Journal of Scientific Computing (2021) 86:1
https://doi.org/10.1007/s10915-020-01359-x

Parallel Algorithms for Successive Convolution

Andrew J. Christlieb[1] · Pierson T. Guthrey[1,2] · William A. Sands[1] · Mathialakan Thavappiragasm[1,3]

Received: 22 July 2020 / Accepted: 1 November 2020 / Published online: 8 December 2020

Abstract
The development of modern computing architectures with ever-increasing amounts of parallelism has allowed for the solution of previously intractable problems across a variety of scientific disciplines. Despite these advances, multiscale computing problems continue to pose an incredible challenge to modern architectures because they require resolving scales that often vary by orders of magnitude in both space and time. Such complications have led us to consider alternative discretizations for partial differential equations (PDEs) which use expansions involving integral operators to approximate spatial derivatives (Christlieb et al. in J Comput Phys 379:214–236, 2019; Christlieb et al. J Sci Comput 82:52(3):1–29, 2020; Christlieb et al. J Comput Phys 415:1–25, 2020). These constructions use explicit information within the integral terms, but treat boundary data implicitly, which contributes to the overall speed of the method. This approach is provably unconditionally stable for linear problems and stability has been demonstrated experimentally for nonlinear problems. Additionally, it is matrix-free in the sense that it is not necessary to invert linear systems and iteration is not required for nonlinear terms. Moreover, the scheme employs a fast summation algorithm that yields a method with a computational complexity of $\mathcal{O}(N)$, where N is the number of mesh points along a coordinate direction. While much work has been done to explore the theory behind these methods, their practicality in large scale computing environments is a largely unexplored topic. In this work, we explore the performance of these methods by developing a domain decomposition algorithm suitable for distributed memory systems along with shared memory algorithms. As a first pass, we derive an artificial Courant–Friedrichs–Lewy condition that enforces a nearest-neighbor (N-N) communication pattern and briefly discuss possible generalizations. We also analyze several approaches for implementing the parallel algorithms by optimizing predominant loop structures and maximizing data reuse. Using a hybrid design that employs MPI and Kokkos (Edwards and Trott in J Parallel Distrib Comput 74:3202–3216, 2014) for the distributed and shared memory components of the algorithms, respectively, we show that our methods are efficient and can sustain an update rate $> 1 \times 10^8$ DOF/node/s. We provide results that demonstrate the scalability and versatility of our algo-

Last updated on October 29, 2020. The research of the authors was supported in part by AFOSR Grants FA9550-19-1-0281 and FA9550-17-1-0394 and NSF Grant DMS 1912183.

✉ William A. Sands
sandswi3@msu.edu

Extended author information available on the last page of the article

rithms using several different PDE test problems, including a nonlinear example, which employs an adaptive time-stepping rule.

Keywords High-performance computing · Domain decomposition · Method-of-lines-transpose · Integral solution · Fast algorithms · Numerical analysis

1 Introduction

In this work, we develop parallel algorithms using novel approaches to represent derivative operators for linear and nonlinear time-dependent partial differential equations (PDEs). We chose to investigate algorithms for these representations due to the stability properties observed for a wide range of linear and nonlinear PDEs. The approach considered here uses expansions involving integral operators to approximate spatial derivatives. Here, we shall refer to this approach as the Method of Lines Transpose (MOL^T) though this can be more broadly categorized within a larger class of successive convolution methods. The name arises because the terms in the operator expansions, which we describe later, involve convolution integrals whose operand is recursively or successively defined. Despite the use of explicit data in these integral terms, the boundary data remains implicit, which contributes to both the speed and stability of the representations. The inclusion of more terms in these operator expansions, when combined with a high-order quadrature method, allow one to obtain a high-order discretization in both space and time. Another benefit of this approach is that extensions to multiple spatial dimensions are straightforward as operators can be treated in a line-by-line fashion. Moreover, the integral equations are amenable to fast-summation techniques, which reduce the overall computational complexity, along a given dimension, from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$, where N is the number of discrete grid points along a dimension.

High-order successive convolution algorithms have been developed to solve a range of time-dependent PDEs, including the wave equation [5], heat equation (e.g., Allen-Cahn [6] and Cahn-Hilliard equations [7]), Maxwell's equations [8], Vlasov equation [9], degenerate advection-diffusion (A-D) equation [2], and the Hamilton-Jacobi (H-J) equation [1,3]. In contrast to these papers, this work focuses on the performance of the method in parallel computing environments, which is a largely unexplored area of research. Specifically, our work focuses on developing effective domain decomposition strategies for distributed memory systems and building thread-scalable algorithms using the low-order schemes as a baseline. By leveraging the decay properties of the integral representation, we restrict the calculations to localized non-overlapping subsets of the spatial domain. The algorithms presented in this work consider dependencies between nearest-neighbors (N-N), but, as we will see, this restriction can be generalized to include additional information, at the cost of additional communication. Using a hybrid design that employs MPI and Kokkos [4] for the distributed and shared memory components of the algorithms, respectively, we show that our methods are efficient and can sustain an update rate $> 1 \times 10^8$ DOF/node/s. While experimentation on graphics processing units (GPUs) shall be left to future work, we believe choosing Kokkos will provide a path for a more performant and seamless integration of our algorithms with new computing hardware.

Recent developments in successive convolution methods have focused on extensions to solve more general nonlinear PDEs, for which an integral solution is generally not applicable. This work considers discretizations developed for degenerate advection-diffusion (A-D) equations [2], as well as the Hamilton-Jacobi (H-J) equations [1,3]. The key idea of these papers exploited the linearity of a given *differential operator* rather than the underlying

equations, allowing derivatives in nonlinear problems to be expressed using the same representations developed for linear problems. For linear problems, it was demonstrated that one could couple these representations for the derivative operators with an explicit time-stepping method, such as the strong-stability-preserving Runge-Kutta (SSP-RK) methods [10] and still obtain schemes which maintain unconditional stability [1,2]. To address shock-capturing and control non-physical oscillations, the latter two papers introduced quadratures that use WENO reconstructions, along with a nonlinear filter to further control oscillations. In [3], the schemes for the H-J equations were extended to enable calculations on mapped grids. This paper also proposed a new WENO quadrature method that uses a basis that consists of exponential polynomials that improves the shock capturing capabilities.

Our choice in discretizing time, first, before treating the spatial quantities, is not a new idea. A well-known approach is Rothe's method [11,12] in which a finite difference approximation is used for time derivatives and an integral equation solver is developed for the resulting sequence of elliptic PDEs (see e.g., [13–19]). The earlier developments for successive convolution methods, such as [5], are quite similar to Rothe's method in the treatment of the time derivatives. However, successive convolution methods differ from Rothe's method considerably in the treatment of spatial derivatives for nonlinear problems, such as those considered in more recent work on successive convolution (see e.g., [1–3]), as Newton iteration can be avoided on nonlinear terms. Additionally, these methods do not require solutions to linear systems. In contrast, Nyström methods, which are used to discretize the integral equations in Rothe's method, result in dense linear systems, which are typically solved using an iterative method such as GMRES [20]. Despite the fact that the linear systems are well-conditioned, the various collective operations that occur in distributed GMRES solves can become quite expensive on large computing platforms.

Similarly, in [21,22], Bruno and Lyon introduced a spectral method, based on the FFT, for computing spatial derivatives of general, possibly non-periodic, functions known as Fourier-Continuation (FC). They combined this representation with the well-known Alternating-Direction (AD) methods, e.g., [23–25], dubbed FC-AD, to develop implicit solvers suitable for linear equations. This resulted in a method capable of computing spatial derivatives in a rapidly convergent and dispersionless fashion. A domain decomposition technique for the FC method is described in [26] and weak scaling was demonstrated to 256 processors, using 4 processors per node, but larger runs were not considered. Another related transform approach was developed to solve the linear wave equation [27]. This work introduced a windowed Fourier methodology and combined this with a frequency-domain boundary integral equation solver to simulate long-time, high-frequency wave propagation. While this particular work does not focus on parallel implementations, they suggest several generic strategies, including a trivially parallelizable approach that solves a collection of frequency-domain integral equations in parallel; however, a purely parallel-in-time approach may not be appropriate for massively parallel systems, especially if few frequencies are required across time. This issue may be further complicated by the parallel implementation of the frequency-domain integral equation solvers, which, as previously mentioned, require the solution of dense linear systems. Therefore, it may be a rather difficult task to develop robust parallel algorithms, which are capable of achieving their peak performance.

This paper is organized as follows: In Sect. 2, we provide an overview of the numerical scheme used to formulate our algorithms. To this end, we first illustrate the connections among several characteristically different PDEs through the appearance of common operators in Sect. 2.1. Using these fundamental operators, we define the integral representation in Sect. 2.2, which is used to approximate spatial derivatives. Once the representations have been introduced, we briefly discuss the complications associated with boundary conditions

and provide the relevant information, in Sect. 2.3, for implementing boundary conditions used in our numerical tests. Sections 2.4 and 2.5 briefly review the spatial discretization process (including the fast-summation method and the quadrature) and the coupling with time integration methods, respectively. Section 3 provides the details of our new domain decomposition algorithm, beginning with the derivation of the so-called N-N conditions in Sect. 3.1. Using these conditions, we show how this can be used to enforce boundary conditions, locally, for first and second derivative operators (Sects. 3.2 and 3.3, respectively). Details concerning the implementation of the parallel algorithms are contained entirely in Sect. 4. This includes the introduction of the shared memory programming model (Sect. 4.1), the definition of a certain performance metric (Sect. 4.2) used in both loop optimization experiments and scaling studies (Sect. 4.3), the presentation of shared memory algorithms (Sect. 4.4), and, lastly, implementation details concerning the distributed memory algorithms (Sect. 4.5). Section 5 contains the core numerical results which confirm the convergence (Sect. 5.1), as well as, the weak and strong scalability (Sects. 5.2 and 5.3, respectively) of the proposed algorithms. In Sect. 5.4, we examine the impact of the restriction posed by the N-N conditions. Finally, we summarize our findings with a brief conclusion in Sect. 6.

2 Description of Numerical Methods

In this section, we outline the approach used to develop unconditionally stable solvers making use of knowledge for linear operators. We will start by demonstrating the connections between several different PDEs using operator notation, which will allow us to reuse, or combine, approximations in several different ways. Once we have established these connections, we define an appropriate “inverse" operator and use this to develop the expansions used to represent derivatives. The representations we develop for derivative operators are motivated by the solution of simple 1-D problems. However, in multi-dimensional problems, these expressions are still valid approximations, in a certain sense, even though the kernels in the integral representation may not be solutions to the PDE in question. While these approximations can be made high-order in both time and space, the focus of this work is strictly on the scalability of the method, so we will limit ourselves to formulations which are first-order in time. Note that the approach described in Sects. 3 and 4, which considers first-order schemes, is quite general and can be easily extended for high-order representations. Once we have discussed our treatment of derivative terms, we describe the fast summation algorithm and quadrature method in Sect. 2.4. Despite the fact that this work only considers smooth test problems, we include the relevant modifications required for non-smooth problems for completeness. In Sect. 2.5, we illustrate how the representation of derivative operators can be used within a time stepping method to solve PDEs.

2.1 Connections Among Different PDEs

Before introducing the operators relevant to successive convolution algorithms, we establish the operator connections appearing in several linear PDE examples. This process helps identify key operators that can be represented with successive convolution. Specifically, we shall consider the following three prototypical linear PDEs:

- Linear advection equation: $(\partial_t - c\partial_x)u = 0$,
- Diffusion equation: $(\partial_t - \nu\partial_{xx})u = 0$,
- Wave equation: $(\partial_{tt} - c^2\partial_{xx})u = 0$.

Next, we apply an *implicit* time discretization to each of these problems. For discussion purposes, we shall consider lower-order time discretizations, i.e., backward-Euler for the $\partial_t u$ and a second-order central difference for $\partial_{tt} u$. If we identify the current time as t^n, the new time level as t^{n+1}, and $\Delta t = t^{n+1} - t^n$, then we obtain the corresponding set of semi-discrete equations:

- Linear advection equation: $(\mathcal{I} - \Delta t c \partial_x) u^{n+1} = u^n$,
- Diffusion equation: $(\mathcal{I} - \Delta t \nu \partial_{xx}) u^{n+1} = u^n$,
- Wave equation: $(\mathcal{I} - \Delta t^2 c^2 \partial_{xx}) u^{n+1} = 2u^n - u^{n-1}$.

Here, we use $\mathcal{I}$ to denote the identity operator, and, in all cases, each of the spatial derivatives are taken at time level t^{n+1} to keep the schemes implicit. The key observation is that the operator $(\mathcal{I} \pm \frac{1}{\alpha}\partial_x)$ arises in each of these examples. Notice that

$$\left(\mathcal{I} - \frac{1}{\alpha^2}\partial_{xx}\right) = \left(\mathcal{I} - \frac{1}{\alpha}\partial_x\right)\left(\mathcal{I} + \frac{1}{\alpha}\partial_x\right),$$

where α is a parameter that is selected according to the equation one wishes to solve. For example, in the case of diffusion, one selects

$$\alpha = \frac{\beta}{\sqrt{\nu \Delta t}},$$

while for the linear advection and wave equations, one selects

$$\alpha = \frac{\beta}{c \Delta t}.$$

The parameter β, which does not depend on Δt, is then used to tune the stability of the approximations. For test problems appearing in this paper, we always use $\beta = 1$. In Sect. 2.2, we demonstrate how the operator $(\mathcal{I} \pm \frac{1}{\alpha}\partial_x)$ can be used to approximate spatial derivatives. We remark that for second derivatives, one can also use $(\mathcal{I} - \frac{1}{\alpha^2}\partial_{xx})$ to obtain a representation for second order spatial derivatives, instead of factoring into "left" and "right" characteristics.

Next, we introduce the following definitions to simplify the notation:

$$\mathcal{L}_L \equiv \mathcal{I} - \frac{1}{\alpha}\partial_x, \quad \mathcal{L}_R \equiv \mathcal{I} + \frac{1}{\alpha}\partial_x. \tag{2.1}$$

Written in this manner, these definitions indicate the left and right-moving components of the characteristics, respectively, as the subscripts are associated with the direction of propagation. For second derivative operators, which are not factored into first derivatives, we shall use

$$\mathcal{L}_0 \equiv \mathcal{I} - \frac{1}{\alpha^2}\partial_{xx}. \tag{2.2}$$

In order to connect these operators with suitable expressions for spatial derivatives, we need to define the corresponding "inverse" for each of these *linear* operators on a 1-D interval $[a, b]$. These definitions are given as

$$\begin{aligned} \mathcal{L}_L^{-1}[\,\cdot\,;\alpha](x) &\equiv \alpha \int_x^b e^{-\alpha(s-x)}(\cdot)\, ds + Be^{-\alpha(b-x)}, \\ &\equiv I_L[\,\cdot\,;\alpha](x) + Be^{-\alpha(b-x)}, \end{aligned} \tag{2.3}$$

$$\begin{aligned} \mathcal{L}_R^{-1}[\,\cdot\,;\alpha](x) &\equiv \alpha \int_a^x e^{-\alpha(x-s)}(\cdot)\, ds + Ae^{-\alpha(x-a)}, \\ &\equiv I_R[\,\cdot\,;\alpha](x) + Ae^{-\alpha(x-a)}. \end{aligned} \tag{2.4}$$

These definitions can be derived in a number of ways. In Appendix Appendix A, we demonstrate how these definitions can be derived for the linear advection equation using the integrating factor method. In these definitions, A and B are constants associated with the "homogeneous solution" of a corresponding semi-discrete problem and are used to satisfy the boundary conditions. In a similar way, one can compute the inverse operator for definition (2.2), which yields

$$\begin{aligned}\mathcal{L}_0^{-1}[\ \cdot\ ;\alpha](x) &\equiv \frac{\alpha}{2}\int_a^b e^{-\alpha|x-s|}(\cdot)\,ds + Ae^{-\alpha(x-a)} + Be^{-\alpha(b-x)},\\ &\equiv I_0[\ \cdot\ ;\alpha](x) + Ae^{-\alpha(x-a)} + Be^{-\alpha(b-x)}.\end{aligned} \tag{2.5}$$

In these definitions, we refer to "$\cdot$" as the operand and, again, α is a parameter selected according to the problem being solved. Although it is a slight abuse of notation, when it is not necessary to explicitly indicate the parameter or the point of evaluation, we shall place the operand inside a pair of parenthesis.

If we connect these definitions to each of the linear semi-discrete equations mentioned earlier, we can determine the update equation through an *analytic inversion* of the corresponding linear operator(s):

- Linear advection equation: $u^{n+1} = \mathcal{L}_R^{-1}(u^n)$, or $u^{n+1} = \mathcal{L}_L^{-1}(u^n)$,
- Diffusion equation: $u^{n+1} = \mathcal{L}_L^{-1}(\mathcal{L}_R^{-1}(u^n))$, or $u^{n+1} = \mathcal{L}_0^{-1}(u^n)$,
- Wave equation: $u^{n+1} = \mathcal{L}_L^{-1}(\mathcal{L}_R^{-1}(2u^n - u^{n-1}))$, or $u^{n+1} = \mathcal{L}_0^{-1}(2u^n - u^{n-1})$,

with the appropriate choice of α for the problem being considered. We note that each of these methods can be made high-order following the work in [1–3,6,28], where it was demonstrated that these approaches lead to methods that are unconditionally stable to all orders of accuracy for these linear PDEs, even with variable wave speeds or diffusion coefficients. Since the process of analytic inversion yields an integral term, a fast-summation technique should be used to reduce the computational complexity of a naive implementation, which would otherwise scale as $\mathcal{O}(N^2)$. Some details concerning the spatial discretization and the $\mathcal{O}(N)$ fast-summation method are briefly summarized in Sect. 2.4 (for full details, please see [6]). Next we demonstrate how the operator $\mathcal{L}_*$ can be used to approximate spatial derivatives.

2.2 Representation of Derivatives

In the previous section, we observed that characteristically different PDEs can be described in-terms of a common set of operators. The focus of this section shall be on manipulating these approximations to obtain a high-order discretization in time through certain operator expansions. The process begins by introducing an operator related to $\mathcal{L}_*^{-1}$, namely,

$$\mathcal{D}_* \equiv \mathcal{I} - \mathcal{L}_*^{-1}, \tag{2.6}$$

where $*$ can be L, R, or 0. The motivation for these definitions will become clear soon. Additionally, we can derive an identity from the definitions (2.6). By manipulating the terms we quickly find that

$$\mathcal{L}_* \equiv (\mathcal{I} - \mathcal{D}_*)^{-1}, \tag{2.7}$$

again, where $*$ can be L, R, or 0. The purpose of the identity (2.7) is that it connects the spatial derivative to an expression involving integrals of the solution rather than derivatives. In other words, it allows us to avoid having to use a stencil operation for derivatives.

To obtain an approximation for the first derivative in space, we can use $\mathcal{L}_L$, $\mathcal{L}_R$, or both of them, which may occur as part of a monotone splitting. If we combine the definition of the left propagating first derivative operator in equation (2.1) with the definition (2.6) and identity (2.7), we can define the first derivative in terms of the $\mathcal{D}_L$ operator. Observe that

$$\begin{aligned}\partial_x^+ &= \alpha\left(\mathcal{I}-\mathcal{L}_L\right),\\ &= \alpha\left(\mathcal{L}_L\mathcal{L}_L^{-1}-\mathcal{L}_L\right),\\ &= \alpha\mathcal{L}_L\left(\mathcal{L}_L^{-1}-\mathcal{I}\right),\\ &= -\alpha\mathcal{L}_L\left(\mathcal{I}-\mathcal{L}_L^{-1}\right),\\ &= -\alpha\left(\mathcal{I}-\mathcal{D}_L\right)^{-1}\mathcal{D}_L,\\ &= -\alpha\sum_{p=1}^{\infty}\mathcal{D}_L^p, \end{aligned} \tag{2.8}$$

where, in the last step, we used the fact that the operator $\mathcal{D}_L$ is bounded by unity in an operator norm. We use the + convention to indicate that this is a right-sided derivative. Likewise, for the right propagating first derivative operator, we find that the complementary left-biased derivative is given by

$$\partial_x^- = \alpha\sum_{p=1}^{\infty}\mathcal{D}_R^p. \tag{2.9}$$

Additionally, the second derivative can be expressed as

$$\partial_{xx} = -\alpha^2\sum_{p=1}^{\infty}\mathcal{D}_0^p. \tag{2.10}$$

As the name implies, each power of $\mathcal{D}_*$ is *successively* defined according to

$$\mathcal{D}_*^k \equiv \mathcal{D}_*\left(\mathcal{D}_*^{k-1}\right). \tag{2.11}$$

In previous work, [2], for periodic boundary conditions, it was established that the partial sums for the left and right-biased approximations to ∂_x satisfy

$$\partial_x^+ = -\alpha\left(\sum_{p=1}^{n}\mathcal{D}_L^p+\mathcal{O}\left(\frac{1}{\alpha^{n+1}}\right)\right),\quad \partial_x^- = \alpha\left(\sum_{p=1}^{n}\mathcal{D}_R^p+\mathcal{O}\left(\frac{1}{\alpha^{n+1}}\right)\right). \tag{2.12}$$

Similarly for second derivatives, with periodic boundaries, retaining n terms leads to a truncation error with the form

$$\partial_{xx} = -\alpha^2\left(\sum_{p=1}^{n}\mathcal{D}_0^p+\mathcal{O}\left(\frac{1}{\alpha^{2n+2}}\right)\right). \tag{2.13}$$

In both cases, the relations can be obtained through a repeated application of integration by parts with induction. These approximations are still exact, in space, but the integral operators nested in $\mathcal{D}_*$ will eventually be approximated with quadrature. From these relations, we can also observe the impact of α on the size of the error in time. In particular, if we select $\alpha = \mathcal{O}(1/\Delta t)$ in (2.12) and $\alpha = \mathcal{O}\left(1/\sqrt{\Delta t}\right)$ in (2.13), each of the approximations should

have an error of the form $\mathcal{O}(\Delta t^n)$. The results concerning the consistency and stability of these higher order approximations were established in [1,2,28].

As mentioned earlier, this work only considers approximations which are first-order with respect to time. Therefore, we shall restrict ourselves to the following operator representations:

$$\partial_x^+ \approx -\alpha \mathcal{D}_L, \quad \partial_x^- \approx \alpha \mathcal{D}_R, \quad \partial_{xx} \approx -\alpha^2 \mathcal{D}_0. \tag{2.14}$$

This is a consequence of retaining a single term from each of the partial sums in equations (2.8), (2.9), and (2.10). Consequently, computing higher powers of $\mathcal{D}_*$ is unnecessary, so the successive property (2.11) is not needed. However, it indicates, clearly, a possible path for higher-order extensions of the ideas which will be presented here. For the moment, we shall delay prescribing the choice of α used in the representations (2.14), in order to avoid a problem-dependent selection. As alluded to at the beginning of this section, an identical representation is used for multi-dimensional problems, where the $\mathcal{D}_*$ operators are now associated with a particular dimension of the problem. The operators along a particular dimension are constructed using data along that dimension of the domain, so that each of the directions remains uncoupled. This completes the discussion on the generic form of the representations used for spatial derivatives. Next, we provide some information regarding the treatment of boundary conditions, which determine the constants A and B appearing in the $\mathcal{D}_*$ operators.

2.3 Comment on Boundary Conditions

The process of prescribing values of A and B, inside equations Eqs. (2.3) to (2.5), which are required to construct $\mathcal{D}_*$, is highly dependent on the structure of the problem being solved. Previous work has shown how to prescribe a variety of boundary conditions for linear PDEs (see e.g., [5–7,28]). For example, in linear problems, such as the wave equation, with either periodic or non-periodic boundary conditions, one can directly enforce the boundary conditions to determine the constants A and B. The situation can become much more complicated for problems which are both nonlinear and non-periodic. For approximations of at most third-order accuracy, with nonlinear PDEs, one can use the techniques from [1] for non-periodic problems. To achieve high-order time accuracy, the partial sums for this case were modified to eliminate certain low-order terms along the boundaries. We note that the development of high-order time discretizations, subject to non-trivial boundary conditions, for nonlinear operators, is still an open area of research for successive convolution methods.

As this paper concerns the scalability of the method, we shall consider test problems that involve periodic boundary conditions. For periodic problems defined on the line interval $[a, b]$, the constants associated with the boundary conditions for first derivatives are given by

$$A = \frac{I_R[v;\alpha](b)}{1-\mu}, \quad B = \frac{I_L[v;\alpha](a)}{1-\mu}, \tag{2.15}$$

where I_L and I_R were defined in Eqs. (2.3) and (2.4). Similarly, for second derivatives, the constants can be determined to be

$$A = \frac{I_0[v;\alpha](b)}{1-\mu}, \quad B = \frac{I_0[v;\alpha](a)}{1-\mu}, \tag{2.16}$$

with the definition of I_0 coming from (2.5). In the expressions (2.15) and (2.16) provided above, we use

$$\mu \equiv e^{-\alpha(b-a)},$$

where α is the appropriately chosen parameter. Note that the function $v(x)$ denotes the generic operand of the operator $\mathcal{D}_*$. This helps reduce the complexity of the notation when several applications of $\mathcal{D}_*$ are required, since they are recursively defined. As an example, suppose we wish to compute $\partial_{xx}h(u)$, where h is some known function. For this, we can use the first-order scheme for second derivatives (see (2.14)) and take $v = h(u)$ in the expressions (2.16) for the boundary terms.

2.4 Fast Convolution Algorithm and Spatial Discretization

To perform a spatial discretization over $[a, b]$, we first create a grid of $N + 1$ points:

$$x_i = a + i\Delta x_i, \quad i = 0, \ldots, N,$$

where

$$\Delta x_i = x_{i+1} - x_i.$$

A naive approach to computing the convolution integral would lead to method of complexity $\mathcal{O}(N^2)$, where N is the number of grid points. However, using some algebra, we can write recurrence relations for the integral terms which comprise $\mathcal{D}_*$:

$$I_R[v;\alpha](x_i) = e^{-\alpha\Delta x_{i-1}} I_R[v;\alpha](x_{i-1}) + J_R[v;\alpha](x_i), \quad I_R[v;\alpha](x_0) = 0,$$
$$I_L[v;\alpha](x_i) = e^{-\alpha\Delta x_i} I_L[v;\alpha](x_{i+1}) + J_L[v;\alpha](x_i), \quad I_L[v;\alpha](x_N) = 0.$$

Here, we have defined the local integrals

$$J_R[v;\alpha](x_i) = \alpha \int_{x_{i-1}}^{x_i} e^{-\alpha(x_i - s)} v(s)\, ds, \tag{2.17}$$

$$J_L[v;\alpha](x_i) = \alpha \int_{x_i}^{x_{i+1}} e^{-\alpha(s - x_i)} v(s)\, ds. \tag{2.18}$$

By writing the convolution integrals this way, we obtain a summation method which has a complexity of $\mathcal{O}(N)$. Note that the same algorithm can be applied to compute the convolution integral for the second derivative operator by splitting the integral at a point x. After applying the above algorithm to the left and right contributions, we can recombine them through "averaging" to recover the original integral. While a variety of quadrature methods have been proposed to compute the local integrals (2.18) and (2.17) (see e.g., [3,5–7,28,29]), we shall consider sixth-order quadrature methods introduced in [2], which use WENO interpolation to address both smooth and non-smooth problems. In what follows, we describe the procedure for $J_R[v;\alpha](x_i)$, since the reconstruction for $J_L[v;\alpha](x_i)$ is similar. This approximation uses a six-point stencil given by

$$S(i) = \{x_{i-3}, \ldots, x_{i+2}\},$$

which is then divided into three smaller stencils, each of which contains four points, defined by $S_r = \{x_{i-3+r}, \ldots, x_{i+r}\}$ for $r = 0, 1, 2$. We associate r with the shift in the stencil. A graphical depiction of this stencil is provided in Fig. 1. The quadrature method is developed as follows:

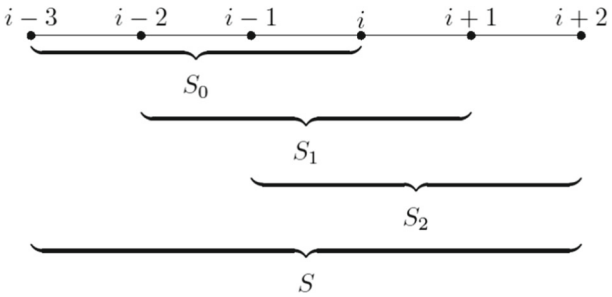

Fig. 1 Stencils used to build the sixth-order quadrature [1,2]

1. On each of the small stencils $S_r(i)$., we use the approximation

$$J_R^{(r)}[v;\alpha](x_i) \approx \alpha \int_{x_{i-1}}^{x_i} e^{-\alpha(x_i-s)} p_r(s)\, ds = \sum_{j=0}^{3} c_{-3+r+j}^{(r)} v_{-3+r+j}, \tag{2.19}$$

   where $p_r(x)$ is the Lagrange interpolating polynomial formed from points in $S_r(i)$ and $c_\ell^{(r)}$ are the interpolation coefficients, which depend on the parameter α and the grid spacing, but not v.
2. In a similar way, on the large stencil $S(i)$ we obtain the approximation

$$J_R[v;\alpha](x_i) \approx \alpha \int_{x_{i-1}}^{x_i} e^{-\alpha(x_i-s)} p(s)\, ds. \tag{2.20}$$

3. When function $v(x)$ is smooth, we can combine the interpolants on the smaller stencils, so they are consistent with the high-order approximation obtained on the larger stencil, i.e.,

$$J_R[v;\alpha](x_i) = \sum_{r=0}^{2} d_r J_R^{(r)}[v;\alpha](x_i), \tag{2.21}$$

   where $d_r > 0$ are called the linear weights, which form a partition of unity. The problems we consider in this work involve smooth functions, so this is sufficient for the final approximation. For instances in which the solution is not smooth, the linear weights can be mapped to nonlinear weights using the notion of smoothness. We refer the interested reader to our previous work [1–3] for details concerning non-smooth data sets.

In Appendix Appendix C, we provide the expressions used to compute coefficients $c_\ell^{(r)}$ and d_r for a uniform grid, although the non-uniform grid case can be done as well [30]. In the case of a non-uniform mesh, the linear weights d_r would become locally defined in the neighborhood of a given point and would need to be computed on-the-fly. Uniform grids eliminate this requirement as the linear weights, for a given direction, can be computed once per time step and reused in each of the lines pointing along that direction.

2.5 Coupling Approximations with Time Integration Methods

Here, we demonstrate how one can use this approach to solve a large class of PDEs by coupling the spatial discretizations in Sect. 2.4 with explicit time stepping methods. In what

follows, we shall consider general PDEs of the form

$$\partial_t U = F(t, U),$$

where $F(t, U)$ is a collective term for spatial derivatives involving the solution variable U. Possible choices for F might include generic nonlinear advection and diffusion terms

$$F(t, U) = \partial_x g_1(U) + \partial_{xx} g_2(U),$$

or even components of the HJ equations

$$F(t, U) = H(U, \partial_x U).$$

To demonstrate how one can couple these approaches, we start by discretizing a PDE in time, but, rather than use backwards Euler, we use an s-stage explicit Runge-Kutta (RK) method, i.e.,

$$u_{n+1} = u_n + \sum_{i=1}^{s} b_i k_i,$$

where the various stages are given by

$$\begin{aligned} k_1 &= F(t^n, u^n), \\ k_2 &= F(t^n + c_2 \Delta t, u^n + \Delta t a_{21} k_1), \\ &\vdots \\ k_s &= F\left(t^n + c_s \Delta t, u^n + \Delta t \sum_{j=1}^{s} a_{sj} k_j\right). \end{aligned}$$

As with a standard Method-of-Lines (MOL) discretization, we would need to reconstruct derivatives within each RK-stage. To illustrate, consider the nonlinear A-D equation,

$$F(t, u) = \partial_x g_1(u) + \partial_{xx} g_2(u).$$

For a term such as g_1, we would use a monotone Lax-Friedrichs flux splitting, i.e., $g_1 \sim \frac{1}{2}(g_1^+ + g_1^-)$, where $g_1^\pm = \frac{1}{2}(g_1(u) \pm ru)$ with $r = \max_u g_1'(u)$. Hence, a particular RK-stage can be approximated using

$$F(t, u) \approx -\frac{1}{2}\alpha \sum_{p=1}^{s} \mathcal{D}_L^p[g_1^+(u); \alpha] + \frac{1}{2}\alpha \sum_{p=1}^{s} \mathcal{D}_R^p[g_1^-(u); \alpha] + \alpha_\nu^2 \sum_{p=1}^{s} \mathcal{D}_0^p[g_2(u); \alpha_\nu].$$

The resulting approximation to the RK-stage can be shown to be $\mathcal{O}(\Delta t^s)$ accurate. Another nonlinear PDE of interest to us is the H-J equation

$$F(t, u) = H(\partial_x u).$$

In a similar way, we would replace the Hamiltonian with a monotone numerical Hamiltonian, such as

$$\hat{H}(v^-, v^+) = H\left(\frac{v^- + v^+}{2}\right) + r(v^-, v^+)\frac{v^- - v^+}{2},$$

where $r(v^-, v^+) = \max_v H'(v)$. Then, the left and right derivative operators in the numerical Hamiltonian can be replaced with

$$\partial_x^- u = \alpha \sum_{p=1}^{s} \mathcal{D}_R^p[u;\alpha], \quad \partial_x^+ u = -\alpha \sum_{p=1}^{s} \mathcal{D}_L^p[u;\alpha],$$

which, again, yields an $\mathcal{O}(\Delta t^s)$ approximation.

In previous work [1,2], for linear forms of $F(t,u)$, it was shown that the resulting methods are unconditionally stable when coupled to explicit RK methods, up to order 3. Extensions beyond third-order are, indeed, possible, but were not considered. For general, nonlinear problems, we typically couple an s-stage RK method to a successive convolution approximation of the same time accuracy, so that the error in the resulting approximation is $\mathcal{O}(\Delta t^s)$.

3 Nearest-Neighbor Domain Decomposition Algorithm

In this section, we provide the relevant mathematical definitions of our domain decomposition algorithm, which are derived from the key operators used in successive convolution. Our goal is to establish and exploit data locality in the method, so that certain reconstructions, which are nonlocal, can be independently completed on non-overlapping blocks of the domain. This is achieved in part by leveraging certain decay properties of the integral representations. Once we have established some useful definitions, we use them to derive conditions in Sect. 3.1, which restrict the communication pattern to N-Ns. Then in Sects. 3.2 and 3.3, we illustrate how this condition can be used to enforce boundary conditions, in a consistent manner, for first and second derivative operators, on each of the blocks. We then provide a brief summary of these findings along with additional comments in Sect. 3.4.

Maintaining a localized stencil is often advantageous in parallel computing applications. Code which is based on N-Ns is generally much easier to write and maintain. Additionally, messages used to exchange data owned by other blocks may not have to travel long distances within the network, provided that the blocks are mapped physically close together in hardware. Communication, even on modern computing systems, is far more expensive than computation. Therefore, an initial strategy for domain decomposition is to enforce N-N dependencies between the blocks. In order to decompose a problem into smaller, independent pieces, we separate the global domain into blocks that share borders with their nearest-neighbors. For example, in the case of a 1-D problem defined on the interval $[a,b]$, we can form N blocks by writing

$$a = c_0 < c_1 < c_2 < \cdots < c_N = b,$$

with $\Delta c_i = c_{i+1} - c_i$ denoting the width of block i. Multidimensional problems can be addressed in a similar way by partitioning the domain along multiple directions. Solving a PDE on each of these blocks, independently, requires an understanding of the various data dependencies. First, we address the local integrals J_*. Depending on the quadrature method, the reconstruction algorithm might require data from neighboring blocks. Reconstructions based on previously described WENO-type quadratures require an extension of the grid in order to build the interpolant. This involves a "halo" region (see Fig. 2), which is distributed amongst N-N blocks in the decomposition. On the other hand, more compact quadratures, such as Simpson's method [29], do not require this data. In this case, the quadrature communication phase can be ignored.

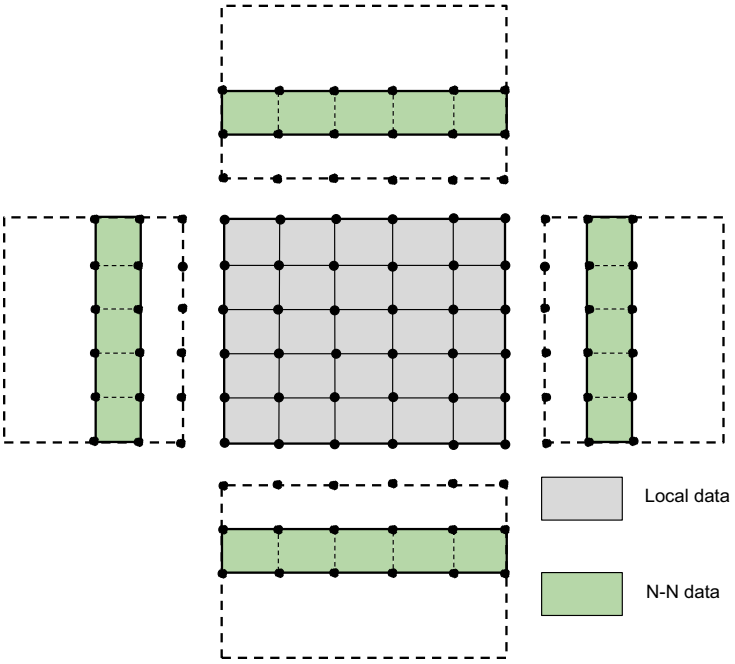

Fig. 2 A sixth-order WENO quadrature stencil in 2-D

The major task for this work involves efficiently communicating the data necessary to build each of the convolution integrals I_*. Once the local integrals J_* are constructed through quadrature, we sweep across the lines of the domain to build the convolution integrals. It is this operation which couples the integrals across the blocks. To decompose this operation, we first rewrite the integral operators I_*, assuming we are in block i, as

$$\begin{aligned} I_L[v;\alpha](x) &= \alpha \int_x^b e^{-\alpha(s-x)} v(s)\, ds, \\ &= \alpha \int_x^{c_{i+1}} e^{-\alpha(s-x)} v(s)\, ds + \alpha \int_{c_{i+1}}^{c_N} e^{-\alpha(s-x)} v(s)\, ds, \end{aligned}$$

and

$$\begin{aligned} I_R[v;\alpha](x) &= \alpha \int_a^x e^{-\alpha(x-s)} v(s)\, ds, \\ &= \alpha \int_{c_0}^{c_i} e^{-\alpha(x-s)} v(s)\, ds + \alpha \int_{c_i}^{x} e^{-\alpha(x-s)} v(s)\, ds. \end{aligned}$$

These relations, which assume x is within the interval $[c_i, c_{i+1}]$, elucidate the local and non-local contributions to the convolution integrals within block i. Using simple algebraic manipulations, we can expand the non-local contributions to find that

$$\int_{c_{i+1}}^{c_N} e^{-\alpha(s-x)} v(s)\, ds = \sum_{j=i+1}^{N-1} \int_{c_j}^{c_{j+1}} e^{-\alpha(s-x)} v(s)\, ds,$$
$$= \sum_{j=i+1}^{N-1} e^{-\alpha(c_j-x)} \int_{c_j}^{c_{j+1}} e^{-\alpha(s-c_j)} v(s)\, ds,$$

and

$$\int_{c_0}^{c_i} e^{-\alpha(x-s)} v(s)\, ds = \sum_{j=0}^{i-1} \int_{c_j}^{c_{j+1}} e^{-\alpha(x-s)} v(s)\, ds,$$
$$= \sum_{j=0}^{i-1} e^{-\alpha(x-c_{j+1})} \int_{c_j}^{c_{j+1}} e^{-\alpha(c_{j+1}-s)} v(s)\, ds,$$

for the right and left-moving data, respectively. With these relations, each of the convolution integrals can be formed according to

$$I_L[v;\alpha](x) = \alpha \int_x^{c_{i+1}} e^{-\alpha(s-x)} v(s)\, ds + \alpha \left(\sum_{j=i+1}^{N-1} e^{-\alpha(c_j-x)} \int_{c_j}^{c_{j+1}} e^{-\alpha(s-c_j)} v(s)\, ds \right), \tag{3.1}$$

and

$$I_R[v;\alpha](x) = \alpha \left(\sum_{j=0}^{i-1} e^{-\alpha(x-c_{j+1})} \int_{c_j}^{c_{j+1}} e^{-\alpha(c_{j+1}-s)} v(s)\, ds \right) + \alpha \int_{c_i}^{x} e^{-\alpha(x-s)} v(s)\, ds. \tag{3.2}$$

From Eqs. (3.1) and (3.2), we observe that both of the global convolution integrals can be split into a localized convolution with additional contributions coming from preceding or successive *global integrals* owned by other blocks in the decomposition. These global integrals contain exponential attenuation factors, the size of which depends on the respective distances between any pair of sub-domains. Next, we use this result to derive the restriction that facilitates N-N dependencies.

3.1 Nearest-Neighbor Criterion

Building a consistent block-decomposition for the convolution integral is non-trivial, since this operation globally couples unknowns along a *dimension* of the grid. Fortunately, the exponential kernel used in these reconstructions is pleasant in the sense that it automatically generates a region of compact support around a given block. Examining the exponential attenuation factors in (3.1) and (3.2), we see that contributions from blocks beyond N-Ns become small provided that (1) the distance between the blocks is large or (2) α is taken to be sufficiently large. Since we have less control over the block sizes e.g., Δc_i, we can enforce the latter criterion. That is, we constrain α so that

$$e^{-\alpha L_m} \leq \epsilon, \tag{3.3}$$

where $\epsilon \ll 1$ is some prescribed error tolerance, typically taken as 1×10^{-16}, and $L_m = \min_i \Delta c_i$ denotes the length smallest block. Taking logarithms of both sides and rearranging

the inequality, we obtain the bound

$$-\alpha \leq \frac{\log(\epsilon)}{L_m}.$$

Our next step is to write this in terms of the time step Δt, using the choice of α. However, the bound on the time step depends on the choice of α. In Sect. 2.1, we presented two definitions for the parameter α, namely

$$\alpha \equiv \frac{\beta}{c_{\max}\Delta t}, \text{ or } \alpha \equiv \frac{\beta}{\sqrt{\nu \Delta t}}.$$

Using these definitions for α, we obtain two conditions depending on the choice of α. For the linear advection equation and the wave equation, we obtain the condition

$$-\frac{\beta}{c_{\max}\Delta t} \leq \frac{\log(\epsilon)}{L_m} \implies \Delta t \leq -\frac{\beta L_m}{c_{\max}\log(\epsilon)}. \tag{3.4}$$

Likewise, for the diffusion equation, the restriction is given by

$$-\frac{\beta}{\sqrt{\nu \Delta t}} \leq \frac{\log(\epsilon)}{L_m} \implies \Delta t \leq \frac{1}{\nu}\left(\frac{\beta L_m}{\log(\epsilon)}\right)^2. \tag{3.5}$$

Depending on the problem, if the condition (3.4) or (3.5) is not satisfied, then we use the maximally allowable time step for a given tolerance ϵ, which is given by the equality component of the relevant condition. If several different operators appear in a given problem and are to be approximated with successive convolution, then each operator will be associated with its own α. In such a case, we should bound the time step according to the condition that is more restrictive among (3.4) and (3.5), which can be accomplished through the choice

$$\Delta t \leq \min\left(-\frac{\beta L_m}{c_{\max}\log(\epsilon)}, \frac{1}{\nu}\left(\frac{\beta L_m}{\log(\epsilon)}\right)^2\right). \tag{3.6}$$

As before, when the condition is not met, then we use the equality in (3.6).

Restricting Δt according to (3.4), (3.5), or (3.6) ensures that contributions to the right and left-moving convolution integrals, beyond N-Ns, become negligible. This is important because it significantly reduces the amount of communication, at the expense of a potentially restrictive time step. Note that in Sect. 5.4, we analyze the limitations of such restrictions for the linear advection equation. In our future work, we shall consider generalizations of our approach, which do not require (3.4), (3.5), or (3.6). In Sects. 3.2 and 3.3, we demonstrate how to formulate block-wise definitions of the global $\mathcal{L}_*^{-1}$ operators using the derived conditions (3.4), (3.5), or (3.6).

3.2 Enforcing Boundary Conditions for ∂_x

In order to enforce the block-wise boundary conditions for the first derivative ∂_x, we recall our definitions (2.3) and (2.4) for the left and right-moving inverse operators:

$$\mathcal{L}_L^{-1}[v;\alpha](x) = I_L[v;\alpha](x) + Be^{-\alpha(b-x)}, \tag{3.7}$$

$$\mathcal{L}_R^{-1}[v;\alpha](x) = I_R[v;\alpha](x) + Ae^{-\alpha(x-a)}. \tag{3.8}$$

We can modify these definitions so that each block contains a pair of inverse operators given by

$$\mathcal{L}_{L,i}^{-1}[v;\alpha](x) = I_{L,i}[v;\alpha](x) + B_i e^{-\alpha(c_{i+1}-x)}, \tag{3.9}$$

$$\mathcal{L}_{R,i}^{-1}[v;\alpha](x) = I_{R,i}[v;\alpha](x) + A_i e^{-\alpha(x-c_i)}, \tag{3.10}$$

where $I_{*,i}$ are defined as

$$I_{L,i}[v;\alpha](x) = \alpha \int_x^{c_{i+1}} e^{-\alpha(s-x)} v(s)\, ds, \quad I_{R,i}[v;\alpha](x) = \alpha \int_{c_i}^{x} e^{-\alpha(x-s)} v(s)\, ds, \tag{3.11}$$

and the subscript i denotes the block in which the operator is defined. As before, this assumes $x \in [c_i, c_{i+1}]$. To address the boundary conditions, we need to determine expressions for the constants B_i and A_i on each of the blocks in the domain. First, substitute the definitions (3.1) and (3.2) into (3.7) and (3.8):

$$\begin{aligned}\mathcal{L}_L^{-1}[v;\alpha](x) = {} & \alpha \int_x^{c_{i+1}} e^{-\alpha(s-x)} v(s)\, ds \\ & + \alpha \left(\sum_{j=i+1}^{N-1} e^{-\alpha(c_j - x)} \int_{c_j}^{c_{j+1}} e^{-\alpha(s-c_j)} v(s)\, ds \right) + B e^{-\alpha(c_N - x)}, \end{aligned} \tag{3.12}$$

$$\begin{aligned}\mathcal{L}_R^{-1}[v;\alpha](x) = {} & \alpha \left(\sum_{j=0}^{i-1} e^{-\alpha(x-c_{j+1})} \int_{c_j}^{c_{j+1}} e^{-\alpha(c_{j+1}-s)} v(s)\, ds \right) \\ & + \alpha \int_{c_i}^{x} e^{-\alpha(x-s)} v(s)\, ds + A e^{-\alpha(x-c_0)}. \end{aligned} \tag{3.13}$$

Since we wish to maintain consistency with the true operator being inverted, we require that each of the block-wise operators satisfy

$$\mathcal{L}_L^{-1}[v;\alpha](x) = \mathcal{L}_{L,i}^{-1}[v;\alpha](x), \quad \mathcal{L}_R^{-1}[v;\alpha](x) = \mathcal{L}_{R,i}^{-1}[v;\alpha](x),$$

which can be explicitly written as

$$B_i e^{-\alpha(c_{i+1}-x)} = \left(\sum_{j=i+1}^{N-1} e^{-\alpha(c_j - x)} I_{L,j}[v;\alpha](c_j) \right) + B e^{-\alpha(c_N - x)}, \tag{3.14}$$

$$A_i e^{-\alpha(x-c_i)} = \left(\sum_{j=0}^{i-1} e^{-\alpha(x-c_{j+1})} I_{R,j}[v;\alpha](c_{j+1}) \right) + A e^{-\alpha(x-c_0)}. \tag{3.15}$$

Evaluating (3.14) at c_{i+1} and (3.15) at c_i, we obtain

$$B_i = \left(\sum_{j=i+1}^{N-1} e^{-\alpha(c_j - c_{i+1})} I_{L,j}[v;\alpha](c_j) \right) + B e^{-\alpha(c_N - c_{i+1})}, \tag{3.16}$$

$$A_i = \left(\sum_{j=0}^{i-1} e^{-\alpha(c_i - c_{j+1})} I_{R,j}[v;\alpha](c_{j+1}) \right) + A e^{-\alpha(c_i - c_0)}. \tag{3.17}$$

Modifying Δt according to either (3.4) or, if necessary (3.6), results in the communication stencil shown in Fig. 3. More specifically, the terms representing the boundary contributions

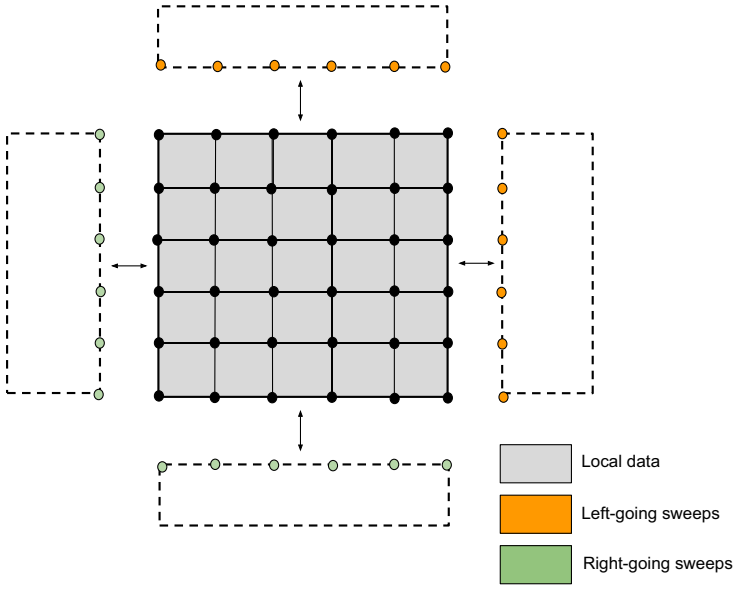

Fig. 3 Fast convolution communication stencil in 2-D based on N-Ns

in each of the blocks are given by

$$B_i = \begin{cases} B, & i = N-1, \\ \\ I_{R,i+1}[v;\alpha](c_{i+1}), & i < N-1, \end{cases}$$

and

$$A_i = \begin{cases} A, & i = 0, \\ \\ I_{R,i-1}[v;\alpha](c_i), & 0 < i. \end{cases}$$

These relations generalize the various boundary conditions set by a problem. For example, with periodic problems, we can select

$$B = I_{L,0}[v;\alpha](c_0), \quad A = I_{R,N-1}[v;\alpha](c_N).$$

This is the relevant strategy employed by domain decomposition algorithms in this work.

3.3 Enforcing Boundary Conditions for ∂_{xx}

The enforcement of boundary conditions on blocks of the domain for the second derivative can be accomplished using an identical procedure to the one described in Sect. 3.2. First, we recall the inverse operator associated with a second derivative (2.5):

$$\mathcal{L}_0^{-1}[v;\alpha](x) = I_0[v;\alpha](x) + Ae^{-\alpha(x-a)} + Be^{-\alpha(b-x)}, \tag{3.18}$$

and define an analogous block-wise definition of (3.18) as

$$\mathcal{L}_{0,i}^{-1}[v;\alpha](x) = I_{0,i}[v;\alpha](x) + A_i e^{-\alpha(x-c_0)} + B_i e^{-\alpha(c_N-x)},$$

with the localized convolution integral

$$I_{0,i}[v;\alpha](x) = \frac{\alpha}{2}\int_{c_i}^{c_{i+1}} e^{-\alpha|x-s|}v(s)\,ds.$$

Again, the subscript i denotes the block in which the operator is defined and we take $x \in [c_i, c_{i+1}]$. For the purposes of the fast summation algorithm, it is convenient to split this integral term into an average of left and right contributions, i.e.,

$$\mathcal{L}_{0,i}^{-1}[v;\alpha](x) = \frac{1}{2}\Big(I_{L,i}[v;\alpha](x) + I_{R,i}[v;\alpha](x)\Big) + A_i e^{-\alpha(x-c_i)} + B_i e^{-\alpha(c_{i+1}-x)}, \tag{3.19}$$

where $I_{*,i}$ are the same integral operators shown in Eq. (3.11) used to build the first derivative. As in the case of the first derivative, a condition connecting the boundary conditions on the blocks to the non-local integrals can be derived, which, if evaluated at the ends of the block, results in the 2×2 linear system

$$\begin{aligned} A_i + B_i e^{-\alpha\Delta c_i} &= \frac{1}{2}\sum_{j=i+1}^{N-1} e^{-\alpha(c_j-c_i)} I_{L,j}[v;\alpha](c_j) \\ &\quad + \frac{1}{2}\sum_{j=0}^{i-1} e^{-\alpha(c_i-c_{j+1})} I_{R,j}[v;\alpha](c_{j+1}) \\ &\quad + Ae^{-\alpha(c_i-c_0)} + Be^{-\alpha(c_N-c_i)}, \end{aligned} \tag{3.20}$$

$$\begin{aligned} A_i e^{-\alpha\Delta c_i} + B_i &= \frac{1}{2}\sum_{j=i+1}^{N-1} e^{-\alpha(c_j-c_{i+1})} I_{L,j}[v;\alpha](c_j) \\ &\quad + \frac{1}{2}\sum_{j=0}^{i-1} e^{-\alpha(c_{i+1}-c_{j+1})} I_{R,j}[v;\alpha](c_{j+1}) \\ &\quad + Ae^{-\alpha(c_{i+1}-c_0)} + Be^{-\alpha(c_N-c_{i+1})}. \end{aligned} \tag{3.21}$$

Equations (3.20) and (3.21) can be solved analytically to find that

$$\begin{pmatrix} A_i \\ B_i \end{pmatrix} = \frac{1}{1-e^{-2\alpha\Delta c_i}} \begin{pmatrix} 1 & -e^{-\alpha\Delta c_i} \\ -e^{-\alpha\Delta c_i} & 1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \end{pmatrix},$$

where we have used the variables r_0 and r_1 to denote the terms appearing on the right-hand side of (3.20) and (3.21), respectively. Under the N-N constraints (3.5) or (3.6), many of the exponential terms can be neglected resulting in the compact expressions

$$A_i = \begin{cases} A, & i = 0, \\ \frac{1}{2} I_{R,i-1}[v;\alpha](c_i) \equiv I_{0,i-1}[v;\alpha](c_i), & 0 < i, \end{cases}$$

and

$$B_i = \begin{cases} B, & i = N-1, \\ \frac{1}{2} I_{L,i+1}[v;\alpha](c_{i+1}) \equiv I_{0,i+1}[v;\alpha](c_{i+1}), & i < N-1. \end{cases}$$

3.4 Additional Comments

In this section, we developed the mathematical framework behind our proposed domain decomposition algorithm. We derived a condition, which reduces the construction of a nonlocal operator to a N-N dependency by leveraging the decay properties of the exponential term within the convolution integrals. We wish to reiterate that this condition is not entirely necessary. One could remove this condition by including contributions beyond N-Ns at the expense of additional communication. This change would certainly result in a loss of speed per time step, but the additional expense could be amortized by the ability to use much a larger time step, which would reduce the overall time-to-solution. As a first pass, we shall ignore these additional contributions, which may limit the scope of problems we can study, but we plan to generalize these algorithms in our future work via an adaptive strategy. This approach would begin using data from N-Ns, then gradually include additional contributions using information about the decay from the exponential. In the next section, we shall discuss details regarding the implementation of our methods and particular design choices made in the construction of our algorithms.

4 Strategies for Efficient Implementation on Parallel Systems

In this section, we discuss strategies for constructing parallel algorithms to solve PDEs. We provide the details related to our work on *thread-scalable*, shared memory algorithms, as well as distributed memory algorithms, where the problem is decomposed into smaller, independent problems that communicate necessary information via message-passing. Section 4.1 introduces the core concepts in Kokkos performance portability library, which is used to develop our shared memory algorithms. Once we have introduced these ideas, we explore numerous loop-level optimizations for essential loop structures in Sect. 4.3, using the performance metrics discussed in Sect. 4.2. Building on the results of these loop experiments, we outline the structure of our shared memory algorithms in Sect. 4.4. We then discuss the implementation of the distributed memory component of our algorithms in Sect. 4.5, along with modifications which enable the use of an adaptive time stepping rule. Finally, we summarize the key findings and developments of the implementation which are used to conduct our numerical experiments.

4.1 Selecting a Shared Memory Programming Model

Many programming models exist to address the aspects of shared memory paralellization, such as OpenMP, OpenACC, CUDA, and OpenCL. The question of which model to use often depends on the target architecture on which the code is to run. However, given the recent trend towards deploying more heterogeneous computing systems, e.g., ones in which a given node contains a variety of CPUs with one, or many, accelerators (typically GPUs), the choice becomes far more complicated. Developing codes which are performant across many computing architectures is a highly non-trivial task. Due to memory access patterns, code which is optimized to run on CPUs is often not optimal on GPUs, so these models address *portability* rather than *performance*. This introduces yet another concern related to code management and maintenance: As new architectures are deployed, code needs to be tuned or modified to take advantage of new features, which can be time consuming. Additionally,

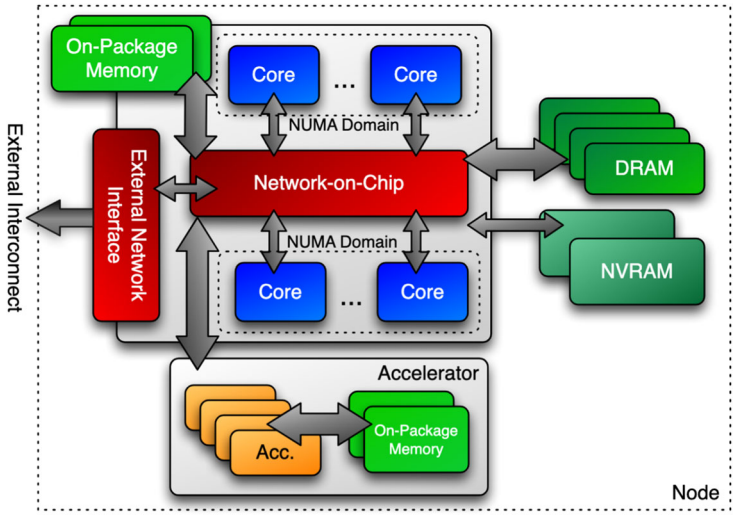

Fig. 4 Heterogeneous platform targeted by Kokkos [4]

enabling these abstractions almost invariably results in either multiple versions of the code or rather complicated build systems.

In our work, we choose to adopt Kokkos [4], a *performance portable*, shared memory programming model. Kokkos tries to address the aforementioned problem posed by rapidly evolving architectures through template-metaprogramming abstractions. Their model provides abstractions for common parallel policies (i.e., for-loops, reductions, and scans), memory spaces, and execution spaces. The architecture specific details are hidden from the users through these abstractions, yet the setup allows the application programmer to take advantage of numerous performance-related features. Given basic knowledge in templates, operator overloads, as well as functors and lambdas, one can implement a variety of program designs. Also provided are the so-called views, which are powerful multi-dimensional array containers that allow Kokkos iterators to map data onto various architectures in a performant way. Additionally, the bodies of iterators become either a user-defined functor or a `KOKKOS_LAMBDA`, which is just a functor that is generated by the compiler. This allows users to maintain one version of the code which has the flexibility to run on various architectures, such as the one depicted in Fig. 4. Other performance portability models, such as RAJA [31], work in a similar fashion as Kokkos, but they are less intrusive with regard to memory management. With RAJA, the user is responsible for implementing architecture dependent details such as array layouts and policies. In this sense, RAJA emphasizes portability, with the user being responsible for handling performance.

4.2 Comment on Performance Metrics

In order to benchmark the performance of the algorithms, we need a descriptive metric that accounts for varying workloads among problem sizes. In our numerical simulations we use a time stepping rule so that problems with a smaller mesh spacing require more time steps, i.e., $\Delta t \sim \Delta x$. Therefore, one can either time the code for a fixed number of steps or track the number of steps in the entire simulation $t \in (0, T]$ and compute the average

time per time step. We adopt the former approach throughout this work. To account for the varying workloads attributed to varying cells/grid points, we define the update rate as Degrees-of-Freedom/node/s (DOF/node/s), which can be computed via

$$\text{DOF/node/s} = \frac{\text{total variables} \times N^d}{\text{nodes} \times \left(\frac{\text{total time (s)}}{\text{total steps}}\right)}, \tag{4.1}$$

where d is the number of spatial dimensions. This metric is a more general way of comparing the raw performance of the code, as it allows for simultaneous comparisons among linear or nonlinear problems with varying degrees of dimensionality and number of components. It also allows for a comparison, in terms of speed, against other classes of methods, such as finite element methods, where the workload on a given cell is allowed to vary according to the number of basis elements.[1] In Sect. 4.3, we shall use this performance metric to benchmark a collection of techniques for prescribing parallelism across predominant loop structures in the algorithms for successive convolution.

4.3 Benchmarking Prototypical Loop Patterns

Often, when designing shared memory algorithms, one has to make design decisions prescribing the way threads are dispatched to the available data. However, there are often many ways of accomplishing a given task. Kokkos provides a variety of parallel iteration techniques — the selection of a particular pattern typically depends on the structure of the loop (perfectly or imperfectly nested) and the size of the loops. In [32], authors sought to optimize a recurring pattern, consisting of triple or quadruple nested for-loops, in the Athena++ MHD code [33]. Their strategy was to use a flexible loop macro to test various loop structures across a range of architectures. Athena++ was already optimized to run on Intel Xeon-Phi platforms, so they primarily focused on approaches for porting to GPUs which maintained this performance on CPUs. Our work differs in that we have not yet identified optimal loop patterns for CPUs or GPUs and algorithms used here contain at least two major prototypical loop patterns. At the moment, we are not focusing on optimizing for GPUs, but, we do our best to keep in mind possible performance-related issues associated with various parallelization techniques. Some examples of recurring loop structures, in successive convolution algorithms, for 3D problems, are provided in Listing 1 and Listing 2. Technically, there are left and right-moving operators associated with each direction, but, for simplicity, we will ignore this in the pseudo-code.

Another important note, we wish to make, concerns the storage of the operator data on the mesh. Since the operations are performed “line-by-line" on potentially large multidimensional arrays, we choose to store the data in memory so that the sweeps are performed on the fastest changing loop variables. This allows us to avoid significant memory access penalties associated with reading and writing to arrays, as the entries of interest are now consecutive in memory.[2] For example, suppose we have an $N-$dimensional array with indices $x_1, x_2, \ldots, x_N$, and we wish to construct an operator in the x_1 direction. Then, we

[1] Note that the update frequency does not account for error in the numerical solution. Certainly, in order to compare the efficiency of various methods, especially those that belong to different classes, one must take into account the quality of the solution. This would be reflected in, for example, an error versus time-to-solution plot.

[2] This is true when the memory space is that of the CPU (host memory). In device memory, these entries will be “coalesced", which is the optimal layout for threading on GPUs. This mapping of indices, between memory spaces, is automatically handled by Kokkos.

would store this operator in memory as `operator`$(x_2, \ldots, x_N, x_1)$. The loops appearing in Listing 1 and Listing 2 can then be permuted accordingly. Note that the solution variable `u`$(x_1, x_2, \ldots, x_N)$ is not transposed and is a read-only quantity during the construction of the operators.

```
for(int ix = 0; ix < Nx; ix++){
  for(int iy = 0; iy < Ny; iy++){
    // Perform some intermediate calculations
    // ...
    // Apply 1-D algorithm to z-line data
    for(int iz = 0; iz < Nz; iz++){
        z_operator(ix,iy,iz) = ...
    }
  }
}
```

Listing 1 Pattern used in the construction of local integrals, convolutions, and boundary steps.

```
for(int ix = 0; ix < Nx; ix++){
  for(int iy = 0; iy < Ny; iy++){
    for(int iz = 0; iz < Nz; iz++){
      z_operator(ix,iy,iz) = u(ix,iy,iz) -
    z_operator(ix,iy,iz);
      z_operator(ix,iy,iz) *= alpha_z;
    }
  }
}
```

Listing 2 Another pattern used to build "resolvent" operators. With some modifications, this same pattern could be used for the integrator step. In several cases, this iteration pattern may require reading entries, which are separated by large distances (i.e., the data is strided), in memory.

In an effort to develop an efficient application, we follow the approach described in [32], to determine optimal loop iteration techniques for patterns, such as Listing 1 and Listing 2. Our simple 2-D and 3-D experiments tested numerous combinations of policies including naive, as well as more complex parallel iteration patterns using the OpenMP backend in Kokkos. Our goals were to quantify possible performance gains attainable through the following strategies:

1. auto-vectorization via `#pragma` statements or `ThreadVectorRange` (TVR)
2. improving data reuse and caching behavior with loop tiling/blocking
3. prescribing parallelism across combinations of team-type execution policies and team sizes

Vectorization can offer substantial performance improvements for data that is contiguous in memory; however, several performance critical operations in the algorithms involve reading data which is strided in memory. Therefore, it is not straightforward whether vectorization would offer any improvements. Additionally, for larger problems, the line operations along certain directions involve reading strided data, so that benefits of caching are lost. The performance penalty of operating on data with the wrong layout depends on the architecture, with penalties on GPUs typically being quite severe compared to CPUs. The use of a blocked iteration pattern, such as the one outlined in Listing 3 (see Appendix B), is a step toward minimizing such performance penalties. In order to see a performance benefit from this approach, the algorithms must be structured, in such a way, as to reuse the data that is read into caches, as much as possible. Naturally, one could prescribe one or more threads (of a team) to process blocks, so we chose to implement cache blocking using the

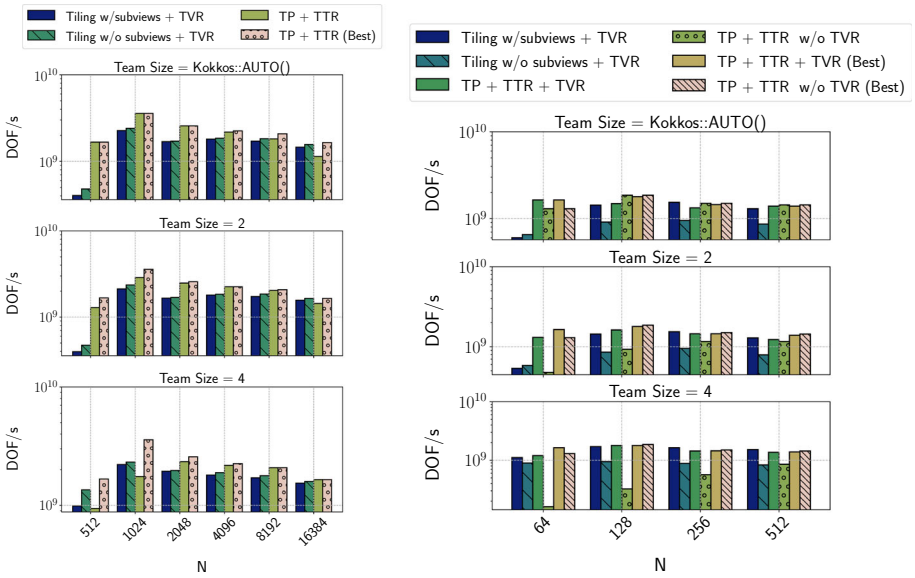

Fig. 5 Plots comparing the performance of different parallel execution policies for the pattern in Listing 1 using test cases in 2-D (left) and 3-D (right). Tests were conducted on a single node that consists of 40 cores using the code configuration outlined in Table 1. Each group consists of three plots, whose difference is the value selected for the team size. We note that hyperthreading is not enabled on our systems, so `Kokkos::AUTO()` defaults to a team size of 1. In each pane, we use "best" to refer to the best run for that configuration across different team sizes. Tile experiments used block sizes of 256^2, in 2-D problems, and 32^3 in 3-D. We observe that vectorized policies are generally faster than non-vectorized policies. Interestingly, among blocked/tiled policies, construction of subviews appears to be faster than those that skip the subview construction, despite the additional work. As the problem size increases, the performance of blocked policies improves substantially. This can be attributed to the large number of idle thread teams when the problem size does not produce enough blocks. In such cases, increasing the size of the team does offer an improvement, as it reduces the number of idle thread teams. For non-blocked policies, we observe that increasing the team-size generally results in minimal, if any, improvement in performance. In all cases, the use of blocking provides a more consistent update rate when enough work is introduced

hierarchical execution policies provided by Kokkos. From coarse-to-fine levels of granularity, these can be ordered as follows: `TeamPolicy` (TP), `TeamThreadRange` (TTR), and `ThreadVectorRange` (TVR). For perfectly nested loops, one can achieve similar behavior using `MDRange` and prescribing block sizes. During testing, we found that when block sizes are larger than or equal to the size of the view, a segmentation fault occurs, so this was avoided. The results of our loop experiments are provided in Figs. 5 and 6. For tests employing blocking, we used a block size of 256^2 in 2-D, while 3-D problems used a block size of 32^3. Information regarding various choices, such as compiler, optimization flags, etc., used to generate these results can be found in Table 1.

As part of our blocking implementation, we stored block information in views, which could then be accessed by a team of threads. After the information about the block is obtained, we compute indices for the data within the block and use these to extract the relevant grid data. Then, one can either create subviews (shallow copies) of the block data or proceed directly with the line calculations of the block data. We refer to these as tiling with and without subviews, respectively. Intuitively, one would think that skipping the block subview creation step would be faster. Among the blocked or tiled experiments, those that created

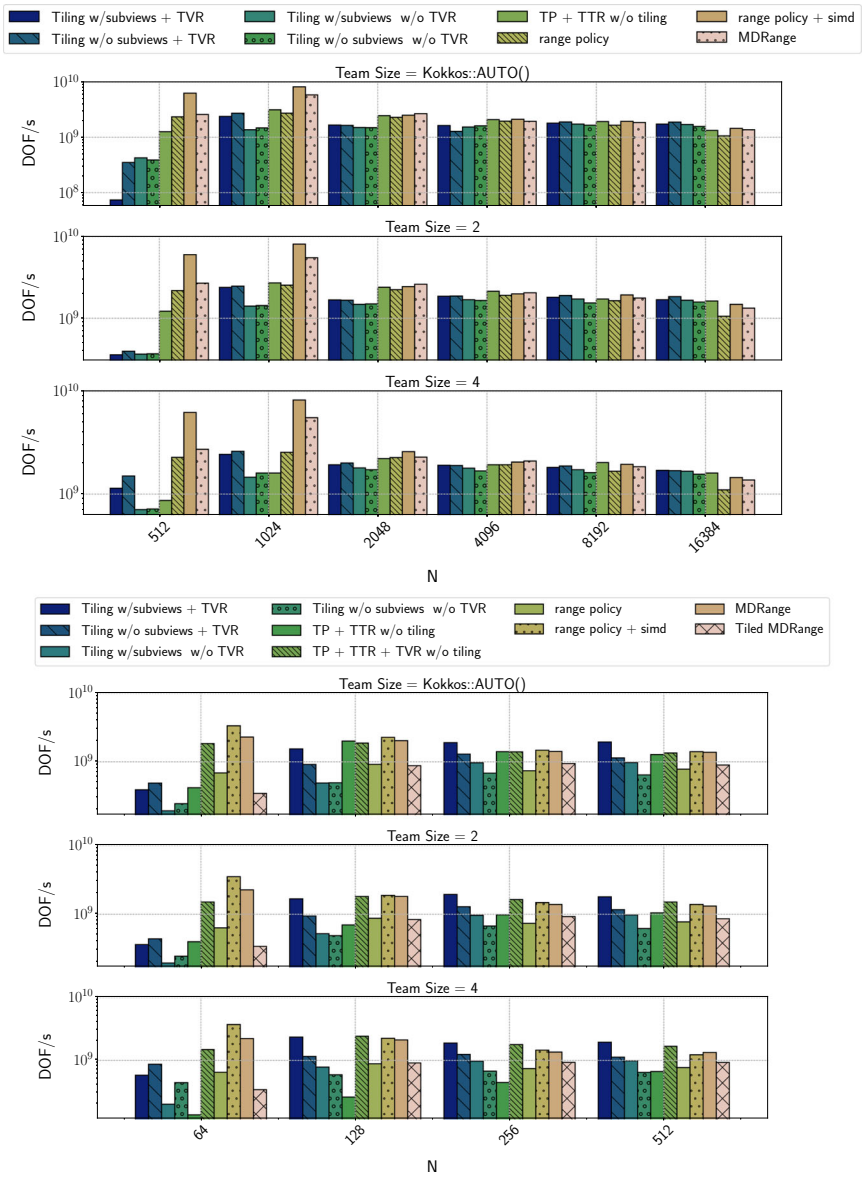

Fig. 6 Plots comparing the performance of different parallel execution policies for the pattern in Listing 2 using test cases in 2-D (top) and 3-D (bottom). Tests were conducted on a single node that consists of 40 cores using the code configuration outlined in Table 1. Each group consists of three plots, whose difference is the value selected for the team size. We note that hyperthreading is not enabled on our systems, so `Kokkos::AUTO()` defaults to a team size of 1. Tile experiments used a block size of 256^2, in 2-D problems, and 32^3 in 3-D. A tiled MDRange was not implemented in the 2-D cases because the block size was larger than some of the problems. The results generally agree with those presented in Fig. 5. For smaller problem sizes, using the non-portable `range_policy` with OpenMP simd directives is clearly superior over the policies. However, when enough work is available, we see that blocked policies with subviews and vectorization generally become the fastest. In both cases, `MDRange` seems to have fairly good performance. Tiling, when used with `MDRange`, in the 3-D cases, seems to be slower than plain `MDRange`. Again, we see that the use of blocking provides a more consistent update rate if enough work is available

Table 1 Architecture and code configuration for the loop experiments conducted on the Intel 18 cluster at Michigan State University's Institute for Cyber-Enabled Research

CPU type	Intel Xeon Gold 6148
C++ Compiler	ICC 2019.03
Optimization flags	`-O3 -xCORE-AVX512 -qopt-zmm-usage=high -qno-opt-prefetch`
Thread bindings	`OMP_PROC_BIND=close, OMP_PLACES=threads`

To leverage the wide vector registers, we encourage the compiler to use AVX-512 instructions. Hardware prefetching is not used, as initial experiments seem to indicate that it hindered performance. Initially, we used GCC 8.2.0-2.31.1 as our compiler, but we found through experimentation that using an Intel compiler improved the performance of our application by a factor of ~ 2 for this platform. Authors in [32] experienced similar behavior for their application and attribute this to a difference in auto-vectorization capabilities between compilers. An examination of the source code for loop execution policies in Kokkos reveals that certain decorators, e.g., `#pragma ivdep` are present, which help encourage auto-vectorization when Intel compilers are used. We are unsure if similar hints are provided for GCC

the subviews of the tile data were generally faster than those that did not. Using blocking for smaller problems typically resulted in a large number of idle threads, which significantly degraded the performance compared to non-blocked policies. In such situations, a user would need to take care to ensure that a sufficient number of blocks are used to generate enough work, i.e., each thread (or team) has at least one block to process. For larger problems, blocking was faster when compared to variants that did not use blocking. We observe that the performance of non-blocked policies begins to degrade once a problem becomes sufficiently large, whereas blocked policies maintained a consistent update rate, even as the problem size increased. By separating the key loop structures from the complexities of the application, we were able to expedite the experimental process for identifying efficient loop execution techniques. In Sect. 4.4, we use the results of these experiments to inform choices regarding the design of the shared memory algorithms.

4.4 Shared Memory Algorithms

The line-by-line approach to operator reconstruction suggests that we employ a *hierarchical design*, which consists of thread teams. Rather than employ a fine-grained threading approach over loop indices, we use the coarse-grained, blocked iteration pattern devised in Sect. 4.3. In this approach, we divide the iteration space into blocks of nearly identical size, and assign one or more blocks to a team of threads. The threads within a given team are then dispatched to one (or more) lines, with vector instructions being used within the lines. As opposed to loop level parallelism, coarse-grained approaches allow one to exploit multiple levels of parallelism, common to many modern CPUs, and load balance the computation across blocks by adjusting the loop scheduling policy. In our implementation, we provide the flexibility of setting the number of threads per block with a macro, but, in general, we let Kokkos choose the appropriate team size using `Kokkos::AUTO()`. If running on the CPU, this sets the team size to be the number of hyperthreads (if supported) on a given core. For GPU architectures, the team size is the size of the warp.

A hierarchical design pattern is used because the loops in our algorithms are not perfectly nested, i.e., calculations are performed between adjoining loops. Information related to blocking can be precomputed to minimize the number of operations required to manipulate blocks. The process of subview construction consists of shallow copies involving pointers to vertices of the blocks, so no additional memory is required. With a careful choice of a base

block size, one can fit these blocks into high-bandwidth memory, so that accessing costs are reduced. Furthermore, a team-based, hierarchical pattern seems to provide a large degree of flexibility compared to standard loop-level parallelism. In particular, we can fuse adjacent kernels into a single parallel region, which reduces the effect of kernel launch overhead and minimizes the number of synchronization points. The use of a team-type execution policy also allows us to exploit features present on other architectures, such as CUDA's shared memory feature, through scratchpad constructs. Performing a stenciled operation on strided data is associated with an architecture-dependent penalty. On CPUs, while one wishes to operate in a contiguous or cached pattern, various compilers can hide these penalties through optimizations, such as prefetching. GPUs, on the other hand, prefer to operate in a lock-step fashion. Therefore, if a kernel is not vectorizable, then one pays a significant performance penalty for poor data access patterns. Shared memory, while slower than register accesses, does not require coalesced accesses, so the cost can be significantly reduced. The advantage of using square-like blocks of a fixed size, as opposed to long pencils,[3] is that one can adjust the dimensions of the blocks so that they fit into the constraints of the high-bandwidth memory. Moreover, these blocks can be loaded once and can be reused for additional directions, whereas pencils would require numerous transfers, as lines are processed along a given dimension. Such optimizations are not explored in this work, but algorithmic flexibility is something we must emphasize moving forwards.

The parallel nested loop structures, such as the one provided in Appendix B (see Listing 3) are applied during reconstructions for the local integrals J_* and inverse operators $\mathcal{L}_*^{-1}$, as well as the integrator update. The current exception to this pattern is the convolution algorithm, shown in Listing 4, which is also provided in Appendix B. Here, each thread is responsible for constructing I_* on one or more lines of the grid. Therefore, within a line, each thread performs the convolution sweeps, in serial, using our $\mathcal{O}(N)$ algorithm. Adopting the team-tiling approach for this operation requires that we modify our convolution algorithm considerably – this optimization is left to future work. Additionally the benefit of this optimizations is not large for CPUs, as profiling indicated that $< 5\%$ of the total time for a given run was spent inside this kernel. However, this will likely consume more time on GPUs, so this will need to be investigated.

If one wishes to use variable time stepping rules, where the time step is computed from a formula of the form

$$\Delta t = \mathrm{CFL} \min\left(\frac{\Delta x}{c_x}, \frac{\Delta y}{c_y}, \ldots\right), \tag{4.2}$$

then one must supply parallel loop structures with simultaneous maximum reductions for each of the wave speeds c_i. This can be implemented as a custom functor, but the use of blocking/tiling introduces some complexities. More complex reducers that enable such calculations are not currently available. For this reason, problems that use time stepping rules, such as (4.2), are constructed with symmetry in the wave speeds, i.e., $c_x = c_y = \ldots$ to avoid an overly complex implementation with blocking/tiling. However, we plan to revisit this in later work as we begin targeting more general problems. Next, in Sect. 4.5, we discuss the distributed memory component of the implementation and the strategy used to employ an adaptive time stepping rule, such as (4.2).

[3] We refer to a pencil as a, generally, long rectangle (in 2-D) and a rectangular prism (in 3-D). The use of pencils, as opposed to square blocks, would require additional precomputing efforts and, possibly, restrictions on the problem size.

4.5 Code Strategies for Domain Decomposition

One of the issues with distributed computing involves mapping the problem data in an intelligent way so that it best aligns with the physical hardware. Since the kernels used in our algorithms consume a relatively small amount of time, it is crucial that we minimize the time spent communicating data. Given that these schemes were designed to run on Cartesian meshes, we can use a "topology aware" virtual communicator supplied by MPI libraries. These constructs take a collection of ranks in a communicator (each of which manages a sub-domain) and, if permitted, attempt to reorganize them to best align with the physical hardware. This mapping might not be optimal, since it depends on a variety of factors related to the job allocation and the MPI implementation. Depending on the problem, these tools can greatly improve the performance of an application compared to a hand-coded implementation that uses the standard communicator. Additionally, MPI's Cartesian virtual communicator provides functionality to obtain neighbor references and derive additional communicator groups, say, along rows, columns, etc. The send and receive operations are performed with *persistent communications*, which remove some of the overhead for communication channel creation. Persistent communications require a regular communication channel, which, for our purposes, is simply N-Ns. For more general problems with irregular communication patterns, standard send and receive operations can be used.

One strategy to minimize exposed communications is to use *non-blocking* communications. This allows the programmer to overlap calculations with communications, and is especially beneficial if the application can be written in a staggered fashion. If certain data required for a later calculation is available, then communication can proceed while another calculation is being performed. Once the data is needed, we can block progress until the message transfer is complete. However, we hope that the calculation done, in between, is sufficient to hide the time spent performing the communication. In the multi-dimensional setting, other operators may be needed, such as ∂_x and ∂_y. However, in our algorithms, directions are not coupled, which allows us to stagger the calculations. So, we can initialize the communications along a given direction and build pieces of other operators in the background.

A typical complication that arises in distributed implementations of PDE solvers concerns the use of various expensive collective operations, such as "all-to-one" and "one-to-all" communications. For implicit methods, these operations occur as part of the iterative method used for solving distributed linear systems. The method employed here is "matrix-free", which eliminates the need to solve such distributed linear systems. For explicit methods, these operations arise when an adaptive time stepping rule, such as equation (4.2), is employed to ensure that the CFL restriction is satisfied for stability purposes. At each time step, each of the processors, or ranks, must know the maximum wave speeds across the entire simulation domain. On a distributed system, transferring this information requires the use of certain collective operations, which typically have an overall complexity of $\mathcal{O}(\log N_p)$, where N_p is the number of processors. While the logarithmic complexity results in a massive reduction of the overall number of steps, these operations use a barrier, in which all progress stops until the operation is completed. This step cannot be avoided for explicit methods, as the most recent information from the solution is required to accurately compute the maximum wave speeds. In contrast, successive convolution methods, do not require this information. However, implementations for schemes developed in e.g., [1–3] considered "explicit" time stepping rules given by Eq. (4.2) because they improved the convergence of the approximations. By exploiting the stability properties associated with successive convolution methods, we can eliminate the need for accurate wave speed information, based on the current state,

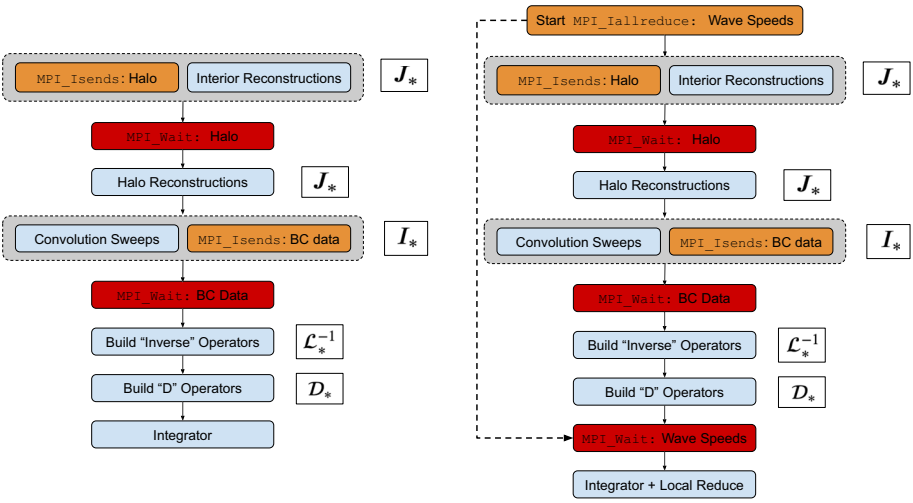

Fig. 7 Task charts for the domain-decomposition algorithm under fixed (left) and adaptive (right) time stepping rules. The work overlap regions are indicated, laterally, using gray boxes. The work inside the overlap regions should be sufficiently large to hide the communications occuring in the background. To clarify, the overlap in calculations for I_* is achieved by changing the sweeping direction during an exchange of the boundary data. As indicated in the adaptive task chart, the reduction over the "lagged" wave speed data can be performed in the background while building the various operators. Note the use of `MPI_WAIT` prior to performing the integrator step. This is done to prevent certain overwrite issues during the local reductions in the subsequent integrator step

and, instead, use approximations obtained with "lagged" data from the previous time step. We present two, generic, distributed memory task charts in Fig. 7. The algorithm shown in the left-half of Fig. 7, which is based on a fixed time step, contains less overall communication, as the local and global reductions for certain information used to compute the time step are no longer necessary. The second version, shown in the right-half of Fig. 7, illustrates the key steps used in the implementation of an adaptive rule, which can be used for problems with more dynamic quantities (e.g., wave speeds and diffusivity). In contrast to distributed implementations of explicit methods, our adaptive approach allows us to overlap expensive global collective operations (approximately) with the construction of derivative operators, resulting in a more asynchronous algorithm (see Algorithm 1).

4.6 Some Remarks

In this section, we introduced key aspects that are necessary in developing a performant application. We began with a brief discussion on Kokkos, which is the programming model used for our shared memory implementation. Then we introduced one of the metrics, namely (4.1), used to characterize the performance of our parallel algorithms. Using this performance metric, we analyzed a collection of techniques for parallelizing prototypical loop structures in our algorithms. These techniques considered several different approaches to the prescription of parallelism through both naive and complex execution policies. Informed by these results, we chose to adopt a coarse-grained, hierarchical approach that utilizes the extensive capabilities available on modern hardware. In consideration of our future work, this approach also offers a large degree of algorithmic flexibility, which will be essential for moving to GPUs.

Algorithm 1 Distributed adaptive time stepping rule

Approximate the global maximum wave speeds $c_x, c_y, \ldots$ using the corresponding "lagged" variables $\tilde{c}_x, \tilde{c}_y, \cdots$

1: Initialize the c_i's and $\tilde{c}_i$'s via the initial condition (no lag has been introduced)
2: **while** timestepping **do**
3: Update the N-N condition (i.e., (3.4), (3.5), or (3.6)) using the "lagged" wave speeds $\tilde{c}_x, \tilde{c}_y, \cdots$
4: Compute Δt using the "lagged" wave speeds and check the N-N condition
5: Start the `MPI_Iallreduce` over the local wave speeds $c_x, c_y, \cdots$
6: Construct the spatial derivative operators of interest
7: Post the `MPI_WAIT` (in case the reductions have not completed)
8: Transfer the global wave speed information to the corresponding lagged variables:

$$\begin{aligned} \tilde{c}_x &\leftarrow c_x, \\ \tilde{c}_y &\leftarrow c_y, \\ &\vdots \end{aligned}$$

9: Perform the update step and computes the local wave speeds $c_x, c_y, \cdots$
10: Return to step 3 to begin the next time step

Finally, we provided some details concerning the implementation of the distributed memory components of the parallel algorithms. We introduced two different approaches: one based on a fixed time step, with minimal communication, and another, which exploits the stability properties of the representations and allows for adaptive time stepping rules. The next section provides numerical results, which demonstrate not only the performance and scalability of these algorithms, but also their versatility in addressing different PDEs.

5 Numerical Results

This section provides the experimental results for our parallel algorithms using MPI and Kokkos, together, with the OpenMP backend. First, in Sect. 5.1, we define several test problems and verify the rates of convergence for the hybrid algorithms described in Sects. 3 and 4. Next, we provide both weak and strong scaling results obtained from each of the example problems discussed in Sect. 5.1. Section 5.4 provides some insight on issues faced by the distributed memory algorithms, in light of the N-N condition (3.4), which was derived in Sect. 3.1. Unless otherwise stated, the results presented in this section were obtained using the configurations outlined in Table 2. Timing data presented in Figs. 9 to 11 was collected using 10 trials for each configuration (problem size and node count) with the update metric (4.1) being displayed relative to 10^9 DOF/node/s. Each of these trials, evolved the numerical solution over 10 time steps. Error bars, collected from data involving averages, were computed using the sample standard deviation.

5.1 Description of Test Problems and Convergence Experiments

Despite the fact that we are primarily focused on developing codes for high-performance applications, we must also ensure that the parallel algorithms produce reliable answers. Here, we demonstrate convergence of the 2-D hybrid parallel algorithms on several test problems, including a nonlinear example that employs the adaptive time stepping rule outlined in Algorithm 1. The convergence results used 9 nodes, with 40 threads per node, assigning 1

Table 2 Architecture and code configuration for the numerical experiments conducted on the Intel 18 cluster at Michigan State University's Institute for Cyber-Enabled Research

CPU type	Intel Xeon Gold 6148
C++ Compiler	ICC 2019.03
MPI library	Intel MPI 2019.3.199
Optimization flags	`-O3 -xCORE-AVX512 -qopt-zmm-usage=high -qno-opt-prefetch`
Thread bindings	`OMP_PROC_BIND=close, OMP_PLACES=threads`
Team size	`Kokkos::AUTO()`
Base block size	256^2
CFL	1.0
β	1.0
ϵ	1×10^{-16}

As with the loop experiments in Sect. 4.3, we encourage the compiler to use AVX-512 instructions and avoid the use of prefetching. All available threads within the node (40 threads/node) were used in the experiments. Each node consists of two Intel Xeon Gold 6148 CPUs and at least 83 GB of memory. We wish to note that hyperthreading is not supported on this system. As mentioned in Sect. 4.3, when hyperthreading is not enabled, `Kokkos::AUTO()` defaults to a team size of 1. In cases where the base block size did not divide the problem evenly, this parameter was adjusted to ensure that blocks were nearly identical in size. The parameter β, which does not depend on Δt is used in the definition of α. For details on the range of admissible β values, we refer the reader to [1,2], where this parameter was introduced. Lastly, recall that ϵ is the tolerance used in the NN constraints

MPI rank to each node, for a total of 360 threads. The quadrature method used to construct the local integrals is the sixth-order WENO-quadrature rule, described in Sect. 2.4, which uses only the linear weights. The numerical solution, in each of the examples, remains smooth over the corresponding time interval of interest. Therefore, it is not necessary to transform the linear quadrature weights to nonlinear ones. According to the analysis of the truncation error presented in [1,2], retaining a single term in the partial sums for $\mathcal{D}_*$ should yield a first-order convergence rate, depending on the choice of α. Convergence results for each of the three test problems defined in Sect. 5.1 are provided in Fig. 8.

Example 1: Linear Advection Equation

The first test problem considered in this work is the 2D linear advection equation

$$\partial_t u + \partial_x u + \partial_y u = 0, \quad (x, y) \in [0, 2\pi]^2,$$
$$u_0(x, y) = \frac{1}{4}(1 - \cos(x))(1 - \cos(y)),$$
s.t. two-way periodic BCs.

We evolve the numerical solution to the final time $T = 2\pi$. In the experiments, we used the same number of mesh points in both directions, with $\alpha_x = \alpha_y = \beta/\Delta t$, with $\beta = 1$. While this problem is rather simple and does not highlight many of the important features of our algorithm, it is nearly identical to the code for a nonlinear example. For initial experiments, a simple test problem is preferable because it gives more control over quantities which are typically dynamic, such as wave speeds. Moreover, the error can be easily computed from the exact solution

$$u(x, y, t) = u_0(x - t, y - t).$$

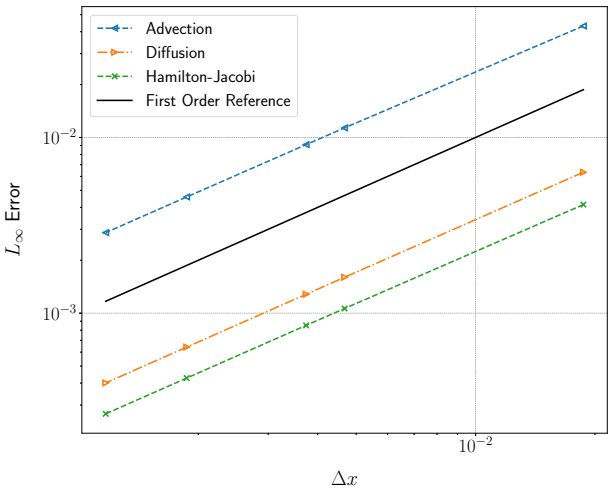

Fig. 8 Convergence results for each of the 2-D example problems. Results were obtained using 9 MPI ranks with 40 threads/node. Also included is a first-order reference line (solid black). Our convergence results indicate first-order accuracy resulting from the low-order temporal discretization. The final reported L_∞ errors for each of the applications, on a grid containing 5277^2 total zones, are 2.874×10^{-3} (advection), 4.010×10^{-4} (diffusion), and 2.674×10^{-4} (H-J)

Example 2: Linear Diffusion Equation

The next test problem that we consider is the linear diffusion equation

$$\begin{aligned} &\partial_t u = \partial_{xx} u + \partial_{yy} u, \quad (x, y) \in [0, 2\pi]^2, \\ &u_0(x, y) = \sin(x) + \sin(y), \\ &\text{s.t. two-way periodic BCs.} \end{aligned}$$

The numerical solution is evolved from $(0, T]$, with $T = 1$, in order to prevent substantial decay. As with the previous example, we use an equal number of mesh points in both directions, so that $\Delta x = \Delta y$. The fixed time stepping rule was used, with $\Delta t = \Delta x$. Compared with the previous example, we used the parameter definitions $\alpha_x = \alpha_y = 1/\sqrt{\Delta t}$, which corresponds to $\beta = 1$ in the definition for second derivative operators. The exact solution for this problem is given by

$$u(x, y, t) = e^{-t}\Big(\sin(x) + \sin(y)\Big).$$

Characteristically, this example is different from the advection equation in the previous example, which allows us to illustrate some key features of the method. Firstly, code developed for advection operators can be reused to build diffusion operators, an observation made in Sect. 2.1. More specifically, to construct the left and right-moving local integrals, we used the same linear WENO quadrature as with the advection equation in Example 1. However, we note that this particular example could, instead, use a more compact quadrature to eliminate the halo communication, which would remove a potential synchronization point. The second feature concerns the time-to-solution and is related to the unconditional stability of the method. Linear diffusion equations, when solved by an explicit method, are known to incur a harsh stability restriction on the time step, namely, $\Delta t \sim \Delta x^2$, making long-time simulations

prohibitively expensive. The implicit aspect of this method drastically reduces the time-to-solution, as one can now select time steps which are, for example, proportional to the mesh spacing. This benefit is further emphasized by the overall speed of the method, which can be observed in Sects. 5.2 and 5.3.

Example 3: Nonlinear Hamilton-Jacobi Equation

The last test problem we consider is the nonlinear H-J equation

$$\partial_t u + \frac{1}{2}\left(1 + \partial_x u + \partial_y u\right)^2 = 0, \quad (x, y) \in [0, 2\pi]^2,$$
$$u_0(x, y) = 0,$$
s.t. two-way periodic BCs.

To prevent the characteristic curves from crossing, which would lead to jumps in the derivatives of the function u, the numerical solution is tracked over a short time, i.e., $T = 0.5$. We applied a high-order linear WENO quadrature rule to approximate the left and right-moving local integrals and used the same parameter choices for α_x, α_y, and β, as with the advection equation in Example 1. However, since the wave speeds fluctuate based on the behavior of the solution u, we allow the time step to vary according to (4.2), which requires the use of the distributed adaptive time stepping rule outlined in Algorithm 1.

Typically, an exact solution is not available for such problems. Therefore, to test the convergence of the method, we use a manufactured solution given by

$$u(x, y, t) = t\Big(\sin(x) + \sin(y)\Big),$$

with a corresponding source term included on the right-hand side of the equation. Methods employed to solve this class of problems are typically explicit, with a shock-capturing method being used to handle the appearance of "cusps" that would otherwise lead to jumps in the derivative of the solution. A brief summary of such methods is provided in our recent paper [3], where extensions of successive convolution were developed for curvilinear and non-uniform grids. The method follows the same structural format as an explicit method with the ability to take larger time steps as in an implicit method. However, the explicit-like structure of this method does not require iteration for nonlinear terms and allows for a more straightforward coupling with high-order shock-capturing methods. We wish to emphasize that despite the fact that this example is nonlinear, the only major mathematical difference with Example 1 is the evaluation of a different Hamiltonian function.

5.2 Weak Scaling Experiments

A useful performance property for examining the scalability of parallel algorithms describes how they behave when the compute resources are chosen proportionally to the size of the problem. Here, the amount of work per compute unit remains fixed, and the compute units are allowed to fluctuate. Weak scaling assumes ideal or best-case performance for the parallel components of algorithms and ignores the influence of bottlenecks imposed by the sequential components of a code. Therefore, for N compute units, we shall expect a speedup of N. This motivates the following definitions for speedup and efficiency in the context of weak scaling:

$$S_N = \frac{NT_1}{T_N}, \quad E_N = \frac{S_N}{N} \equiv \frac{T_1}{T_N}.$$

Therefore, with weak scaling, ideal performance is achieved when the run times for a fixed work size (or, equivalently, the DOF/node/s) remain constant, as we vary the compute units. To scale the problem size, we take advantage of the periodicity for the test problems. The base problem on $[0, 2\pi] \times [0, 2\pi]$ can be replicated across nodes so that the total work per node remains constant. Provided in Fig. 9 are plots of the weak scaling data—specifically the update metric Eq. (4.1) and the corresponding efficiency—obtained from the fastest of 10 trials of each configuration, using up to 49 nodes (1,960 cores).

These results generally indicate good performance, both in terms of the update frequency and efficiency, for a variety of problem sizes. Weak scalability appears to be excellent up to 16 nodes (640 cores), then begins to decline, most likely due to network effects. The performance behavior for advection and diffusion applications is quite similar, which is to be expected, since the parallel algorithms used to construct the base operators are nearly identical. With regard to the Hamilton-Jacobi application, we see that the performance is similar to the other applications at larger node counts. This seems to indicate that no major communication penalties are incurred by use of the adaptive time stepping method shown in algorithm 1, compared to fixed time stepping. Additionally, in the Hamilton-Jacobi application, we observe a sharp decline in the performance at 9 nodes in Fig. 9. A closer investigation reveals that this is likely an artifact of the job scheduler for the system on which the experiments were conducted, as we were unable to secure a “contiguous" allocation of nodes. This has the unfortunate consequence of not being able to guarantee that data for a particular trial remain in close *physical* proximity. This could result in issues such as network contention and delays that exacerbate the cost of communication relative to the computation as discussed in Sect. 4. An non-contiguous placement of data is problematic for codes with inexpensive operations, such as the methods shown here, because the work may be insufficient to hide this increased cost of communication. For this reason, we chose to include plots containing the averaged weak scaling data in Fig. 9, which contains error bars calculated from the sample standard deviation. The noticeable size of the error bars in these plots generally indicates a large degree of variation in the timings collected from trials.

To more closely examine the importance of data proximity on the nodes, we repeated the weak scaling study, but with node counts for which a contiguous allocation count could be guaranteed. We have provided results for the fastest and averaged data in Fig. 10. Data collected from the fastest trials indicates nearly perfect weak scaling, across all applications, up to 9 nodes, with a consistent update rate between $2-4\times10^8$ DOF/node/s. For convenience, these results were plotted with the same markers and formats so that results from the larger experiments in Fig. 9 could be compared directly. A comparison of the fastest timings between the large and small runs supports our claim that data proximity is crucial to achieving the peak performance of the code. Furthermore, the error bars for the contiguous experiments displayed in Fig. 10 show that the individual trials exhibit less overall variation in timings.

5.3 Strong Scaling Experiments

Another form of scalability considers a fixed problem size and examines the effect of varying the number of work units used to find the solution. In these experiments, we allow the work per compute unit to decrease, which helps identify regimes where sequential bottlenecks in algorithms become problematic, provided we are granted enough resources. Applications which are said to strong scale exhibit run times which are inversely proportional to the number of resources used. For example, when N compute units are applied to a problem, one expects the run to be N times faster than with a single compute unit. Additionally, if an

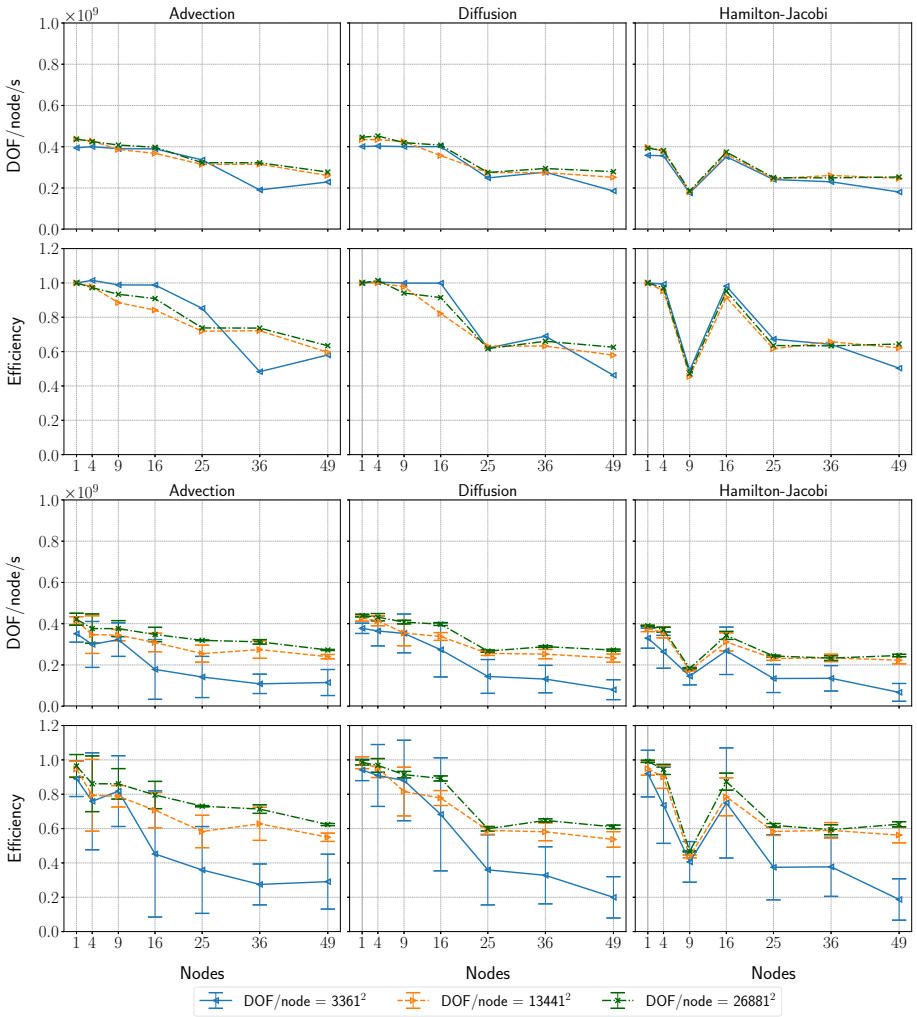

Fig. 9 Weak scaling results, for each of the applications, using up to 49 nodes (1960 cores). For each of the applications, we have provided the update rate and weak scaling efficiency computed via the fastest time/step (top) and average time/step (bottom). Results for advection and diffusion applications is quite similar, despite the use of different operators. The results for the H–J application seem to indicate that no major performance penalties are incurred by use of the adaptive time stepping method. Scalability appears to be excellent, up to 16 nodes (640 cores), then begins to decline. While some loss in performance, due to network effects, is to be expected, this loss appears to be larger than was previously observed. The nodes used in the runs were not contiguous, which hints at a possible sensitivity to data locality

algorithm's performance is memory bound, rather than compute bound, this will, at some point, become apparent in these experiments. Supplying additional compute units should not improve performance, if more time is spent fetching data, rather than performing useful computations. This motivates the following definitions for speedup and efficiency in the context of strong scaling:

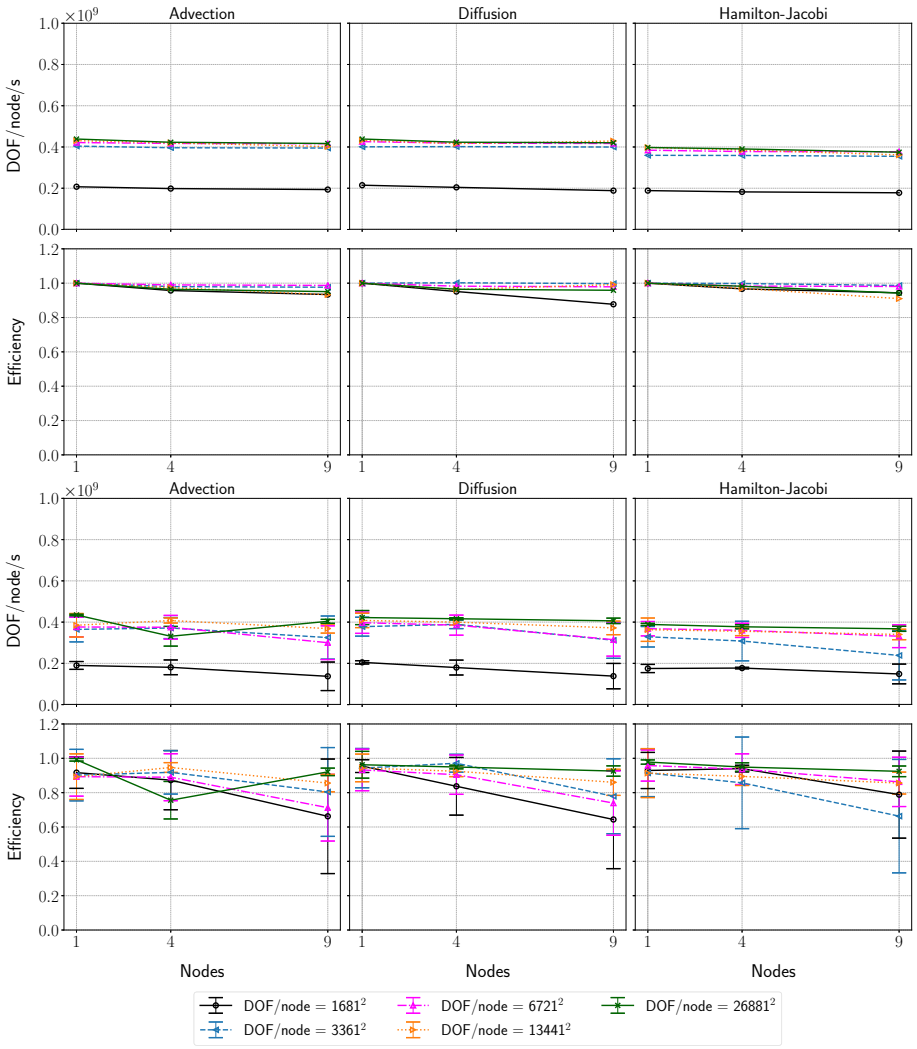

Fig. 10 Weak scaling results obtained with contiguous allocations of up to 9 nodes (360 cores) for each of the applications. For comparison, the same information is displayed as in Fig. 9. Data from the fastest trials indicates nearly perfect weak scaling, across all applications, up to 9 nodes, with a consistent update rate between 2–4 $\times 10^8$ DOF/node/s. A comparison of the fastest timings between the large and small runs supports our claim that data proximity is crucial to achieving the peak performance of the code. Note that size the error bars are generally smaller than those in Fig. 9. This indicates that the timing data collected from individual trials exhibits less overall variation

$$S_N = \frac{T_1}{T_N}, \quad E_N = \frac{S_N}{N}.$$

Here, N is the number of nodes used, so that T_1 and T_N correspond to the time measured using a single node and N nodes, respectively. Results of our strong scaling experiments are provided in Fig. 11. As with the weak scaling experiments, we have plotted the update

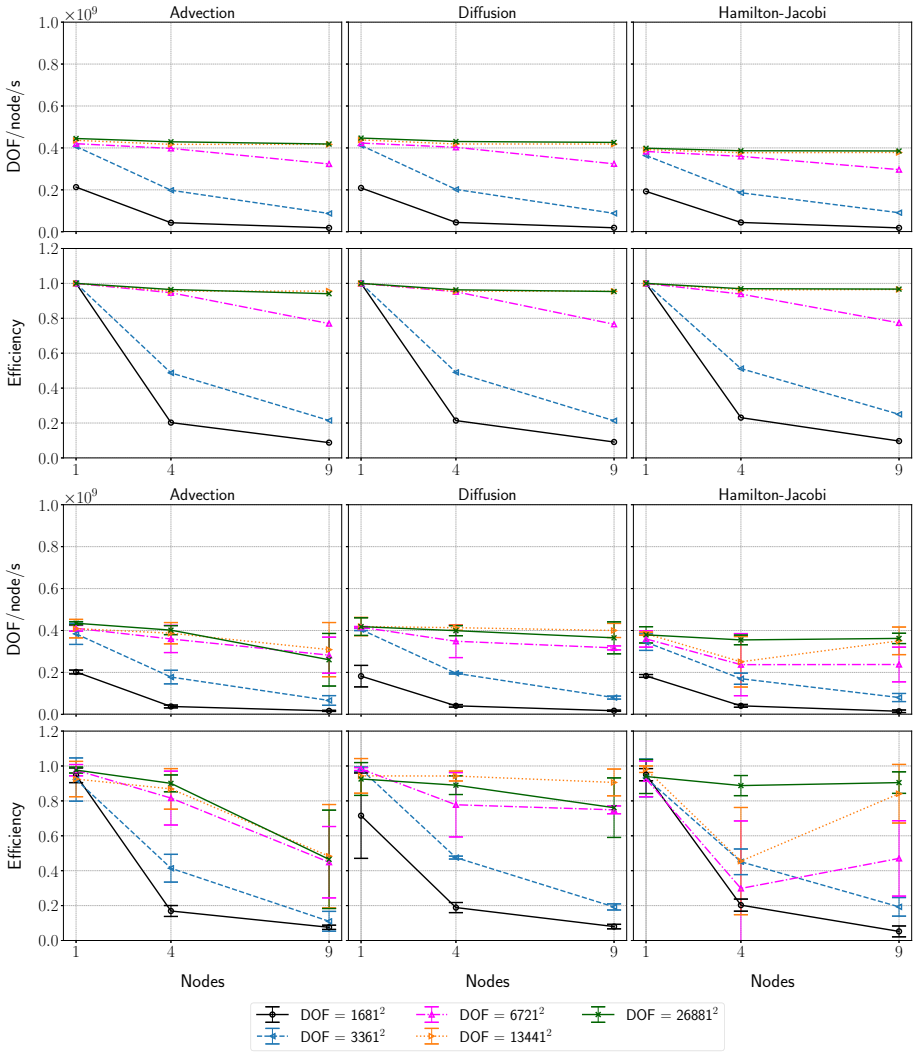

Fig. 11 Strong scaling results for each of the applications obtained on contiguous allocations of up to 9 nodes (360 cores). Displayed among each of the applications are the update rate and strong scaling efficiency computed from the fastest time/step (top) and average time/step (bottom). This method does not contain a substantial amount of work, so we do not expect good performance for smaller base problem sizes, as the work per node becomes insufficient to hide the cost of communication. Larger base problem sizes, which introduce more work, are capable of saturating the resources, but will at some point become insufficient. Moreover, threads become idle when the work per node fails to introduce enough blocks

metric eq. (4.1) along with the strong scaling efficiency using both the fastest and averaged configuration data from a set of 10 trials. In contrast to weak scaling, strong scaling does not assume ideal speedup, so one could plot this information; however, the information can be ascertained from the efficiency data, so we refrain from plotting this data.

Results from these experiments show decent strong scalability for the N-N method. This method does not contain a substantial amount of work, so we do not expect good performance

for smaller base problem sizes, as the work per node becomes insufficient to hide the cost of communication. On the other hand, larger base problem sizes, which introduce more work, are capable of saturating the resources, but will at some point become insufficient. This behavior is apparent in our efficiency plots. Increasing the problem size generally results in an improvement of the efficiency and speedup for the method. Part of these problems can be attributed to the use of a blocking pattern for loops structures discussed in Sect. 4.4. Depending on the size of the mesh, it may be the case that the block size and the team size set by the user result in idle threads. One possible improvement is to simply increase the team size so that there are fewer idle threads within an MPI task. Alternatively, one can adjust the number of threads per task, so that each task is responsible for fewer threads. While these approaches can be implemented with no changes to the code, they will likely not resolve this issue. Profiling seems to indicate that the source of the problem is the low arithmetic intensity of the reconstruction algorithms. In other words, the method is memory bound because the calculations required in the reconstructions are inexpensive relative to the cost of retrieving data from memory. As part of our future work, we plan to investigate such limitations through the use of detailed roofline models. We also plan to consider test problems in 3-D, which will introduce additional work.

5.4 Effect of CFL

In order to enforce a N-N dependency for our domain decomposition algorithm, we obtained several possible restrictions on Δt, depending on the problem and the choice of α. In the case of linear advection, we would, for example, require that

$$\Delta t \leq -\frac{\beta L_m}{c_{\max} \log(\epsilon)}.$$

with the largest possible time step permitting N-N dependencies being set by the equality. Admittedly, such a restriction is undesirable. As mentioned in Sect. 3.1, this assumption can be problematic if the problem admits fast waves ($c_{\max}$ is large) and/or if the block sizes are particularly small (L_m is small). In many applications, the former circumstance is quite common. However, our test problem contains fixed wave speeds so this is less of an issue. The latter condition is a concern for configurations which use many blocks, such as a large simulation on many nodes of a cluster. Another potential circumstance is related to the granularity of the blocks. For example, in these experiments, we use 1 MPI rank per compute node. However, it may be advantageous to consider different task configurations, e.g., using 1 (or more) rank(s) per NUMA region of a compute node.

A larger CFL parameter is generally preferable because it reduces the overall time-to-solution. Eventually, however, for a given CFL, there will be a crossover point, where the time step restriction causes the performance to drop due to the increasing number of *sequential* time steps. This experiment used a highly refined grid and varied the CFL number, using up to 9 nodes. Results from the CFL experiments are provided in Fig. 12. The data was obtained using an older version of our parallel algorithms, compiled with GCC 8.2.0-2.31.1, which does not use blocking. By plotting the behavior according to the number of nodes (ranks) used, we can fix the lengths of the blocks, hence L_m, and change the time step to identify the breakdown region. We observe a substantial decrease in performance for the 9 node configuration, specifically when the CFL number increases from 4 to 5. For more complex problems with dynamic wave behavior, this breakdown may be observed earlier. In response to this behavior, a user could simply increase or relax the tolerance, but the logarithm tends

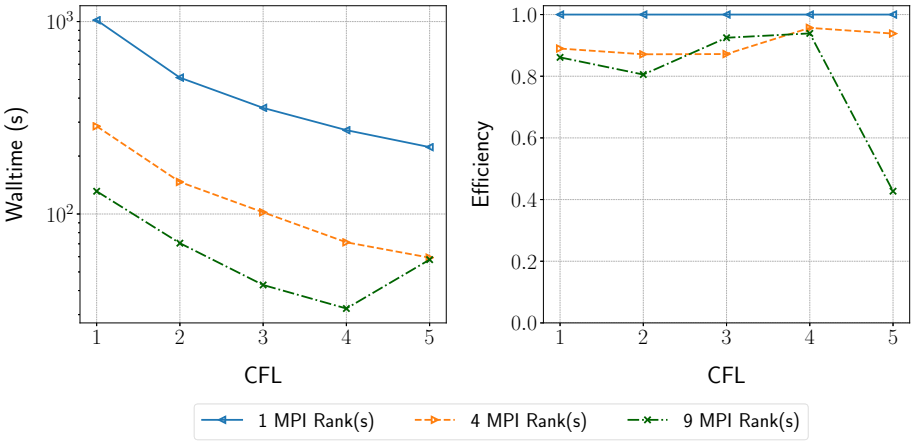

Fig. 12 Results on the N-N method for the linear advection equation using a fixed mesh with 5377^2 total DOF and a variable CFL number. In each case, we used the fastest time-to-solution collected from repeating each configuration a total of 20 times. This particular data was collected using an older version of the code, compiled with GCC, which did not use the blocking approach. For larger block sizes, increasing the CFL has a noticeable improvement on the run time, but as the block sizes become smaller, the gains diminish. For example, if 9 MPI ranks are used, improvements are observed as long as CFL ≤ 4. However, when CFL $= 5$, the run times begin to increase, with a significant decrease in efficiency. As the blocks become smaller, Δt needs to be adjusted (decreased) so that the support of the non-local convolution data not extend beyond N-Ns

to suppress the impact of large relaxations. Another option, which shall be considered in future work is to include more information from neighboring ranks by either eliminating this condition or, at the least, communicating enough information to achieve a prescribed tolerance.

6 Conclusion

In this paper, we presented hybrid parallel algorithms capable of addressing a wide class of both linear and nonlinear PDEs. To enable parallel simulations on distributed systems, we derived a set of conditions that use available wave speed (and/or diffusivity) information, along with the size of the sub-domains, to limit the communication through an adjustment of the time step. Although not considered here, these conditions, which are needed to ensure accuracy, rather than the stability of the method, can be removed at the cost of additional communication. Using these restrictions, boundary conditions are enforced across sub-domains in the decomposition. Results were obtained for 2-D examples consisting of linear advection, linear diffusion, and a nonlinear H-J equation to highlight the versatility of the methods in addressing characteristically different PDEs.

As part of the implementation, we used constructs from the Kokkos performance portability library to parallelize the shared memory components of the algorithms. We extracted essential loop structures from the algorithms and analyzed a variety of parallel execution policies in an effort to develop an efficient application. These experiments considered several common optimization techniques, such as vectorization, cache-blocking, and placement of threads. From these experiments, we chose to use a blocked iteration pattern in which threads (or teams thereof) are mapped to blocks of an array with vector instructions being

applied to 1-D line segments. These design choices offer a large degree of flexibility, which is an important consideration as we proceed with experimentation on other architectures, including GPUs, to leverage the full capabilities provided by Kokkos. By exploiting the stability properties of the representations, we also developed an adaptive time stepping method for distributed systems that uses "lagged" wave speed information to calculate the time step. While the methods presented here do not require adaptive time stepping for stability, it was included as an option because of its ability to prevent excessive numerical diffusion, as observed in previous work.

Convergence and scaling properties for the hybrid algorithms were established using at most 49 nodes (1960 cores), with a peak performance $> 10^8$ DOF/node/s. Larger weak scaling experiments, which used up to 49 nodes (1960 cores), initially performed reasonably well, with all applications later tending to 60% efficiency corresponding (roughly) to 2×10^8 DOF/node/s. While some performance loss is to be expected from network related complications, we found this to be much larger than what was observed in prior experiments. Later, it was discovered that the request for a contiguous allocation could not be accommodated so data locality in the experiments was compromised. By repeating the experiments on a smaller collection of nodes, which granted this request, we discovered that data locality plays a pivotal role in the overall performance of the method. We observe that a large base problem size is required to achieve good strong scaling. Furthermore, when threads are prescribed work at a coarse granularity (i.e., across blocks, rather than entries within the blocks), one must ensure that the problem size is capable of saturating the resources to avoid idle threads. This approach introduces further complications for strong scaling, as the workload per node drops substantially while the block size remains fixed. Finite difference methods which, generally, do not generate a substantial amount of work, are quite similar to successive convolution. Therefore, we do not expect excellent strong scalability. Certain aspects of the algorithms can be tuned to improve the arithmetic intensity, which will improve the strong scaling behavior. At some point, the algorithms will be limited by the speed of memory transfers rather than computation. We also provided experimental results that demonstrate the limitations of the N-N condition in the context of strong scaling.

While we have presented several new ideas with this work, there is still much untapped potential with successive convolution methods. Firstly, optimizations on GPU architectures, which shall play an integral role in the upcoming exascale era, need to be explored and compared with CPUs. A roofline model should be developed for these algorithms to help identify key limitations and bottlenecks and formulate possible solutions. Although this work considered only first-order time discretizations, our future developments shall be concerned with evaluating a variety of high-order time discretization techniques in an effort to increase the efficiency of the method. Lastly, the parallel algorithms should be modified to enable the possibility of mesh adaptivity, which is a common feature offered by many state-of-the-art computing libraries.

Acknowledgements The research of the authors was supported partly through computational resources made available by Michigan State University's Institute for Cyber-Enabled Research. Funding for this work was provided in part by AFOSR grants FA9550-19-1-0281 and FA9550-17-1-0394 and NSF grant DMS 1912183. The authors would like to thank Brian O'Shea and Philipp Grete at Michigan State University for helpful discussions relating to Kokkos and some suggestions for fine-tuning of the parallel algorithms presented here. Additionally, W. Sands would like to thank the organizers of the 2019 Performance Portability Workshop with Kokkos, along with Christian Trott, David Hollman, and Damien Lebrun-Grandie for their assistance with our application. We also wish to express our gratitude to W. Nicolas G. Hitchon for taking the time to review our work and offer numerous suggestions and improvements regarding the presentation of the manuscript.

Appendix A Example for Linear Advection

Suppose we wish to solve the 1D linear advection equation:

$$\partial_t u + c\partial_x u = 0, \quad (x,t) \in (a,b) \times \mathbb{R}^+, \tag{Appendix A.1}$$

where $c > 0$ is the wave speed and leave the boundary conditions unspecified. The procedure for $c < 0$ is analogous. Discretizing (Appendix A.1) in time with backwards Euler yields a semi-discrete equation of the form

$$\frac{u^{n+1}(x) - u^n(x)}{\Delta t} + c\partial_x u^{n+1}(x) = 0.$$

If we rearrange this, we obtain a linear equation of the form

$$\mathcal{L}[u^{n+1}; \alpha](x) = u^n(x), \tag{Appendix A.2}$$

where we have used

$$\alpha := \frac{1}{c\Delta t}, \quad \mathcal{L} := \mathcal{I} + \frac{1}{\alpha}\partial_x.$$

By reversing the order in which the discretization is performed, we have created a sequence of BVPs at discrete time levels. If we had discretized equation (Appendix A.1) using the MOL formalism, then $\mathcal{L}$ would be an algebraic operator. To solve Eq. (Appendix A.2) for u^{n+1}, we analytically invert the operator $\mathcal{L}$. Notice that this equation is actually an ODE, which is linear, so the problem can be solved using methods developed for ODEs. If we apply the integrating factor method to the problem, we obtain

$$\partial_x \left[e^{\alpha x} u^{n+1}(x) \right] = \alpha e^{\alpha x} u^n(x).$$

To integrate this equation, we use the fact that characteristics move to the right, so integration is performed from a to x. After rearranging the result, we arrive at the update equation

$$\begin{aligned} u^{n+1}(x) &= e^{-\alpha(x-a)} u^{n+1}(a) + \alpha \int_a^x e^{-\alpha(x-s)} u^n(s)\, ds, \\ &\equiv e^{-\alpha(x-a)} A^{n+1} + \alpha \int_a^x e^{-\alpha(x-s)} u^n(s)\, ds, \\ &\equiv \mathcal{L}^{-1}[u^n; \alpha](x). \end{aligned}$$

This update displays the origins of the implicit behavior of the method. While convolutions are performed on data from the previous time step, the boundary terms are taken at time level $n+1$.

Now that we have obtained the update equation, we need to apply the boundary conditions. Clearly, if the problem specifies a Dirchlet boundary condition at $x = a$, then $A^{n+1} = u^{n+1}(a)$. We can compute a variety of boundary conditions using the update equation

$$u^{n+1}(x) = e^{-\alpha(x-a)} A^{n+1} + \alpha \int_a^x e^{-\alpha(x-s)} u^n(s)\, ds,$$

where

$$I[u^n; \alpha](x) = \alpha \int_a^x e^{-\alpha(x-s)} u^n(s)\, ds.$$

For example, with periodic boundary conditions, we would need to satisfy

$$u^{n+1}(a) = u^{n+1}(b), \tag{Appendix A.3}$$

$$\partial_x u^{n+1}(a) = \partial_x u^{n+1}(b). \tag{Appendix A.4}$$

Applying condition (Appendix A.3), we find that

$$A^{n+1} = e^{-\alpha(b-a)} A^{n+1} + \alpha \int_a^b e^{-\alpha(b-s)} u^n(s)\, ds.$$

Solving this equation for A^{n+1} shows that

$$A^{n+1} = \frac{I[u^n; \alpha](b)}{1-\mu},$$

with $\mu = e^{-\alpha(b-a)}$. Alternatively, we could have started with (Appendix A.4), which would give an identical solution. While this particular procedure is only applicable to linear problems, this exercise motivates some of the choices made to define operators in the method.

Appendix B Kokkos Kernels

This section provides listings, which outline the general format of the Kokkos kernels used in this work. Specifically, we provide structures for the tiled/blocked algorithms (Listing 3) in addition to the kernel that executes the fast summation method along a line (Listing 4).

```
// Distribute tiles of the array to teams of threads dynamically
Kokkos::parallel_for("team loop over tiles", team_policy(total_tiles, Kokkos::AUTO()),
    KOKKOS_LAMBDA(team_type &team_member)
{
    // Determine the flattened tile index via the team rank
    // and compute the unflattened indices of the tile T_{i,j}
    const int tile_idx = team_member.league_rank();
    const int tj = tile_idx % num_tiles_x;
    const int ti = tile_idx / num_tiles_x;

    // Retrieve tile sizes & offsets and
    // obtain subviews of the relevant grid data on tile T_{i,j}
    // ...

    // Use a team's thread range over the lines
    Kokkos::parallel_for(Kokkos::TeamThreadRange<>
      (team_member, Ny_tile), [&](const int iy)
    {
        // Slice to extract a subview of my line's data and
        // call line methods which use vector loops
        // ...
    }
});
```

Listing 3 An example of coarse-grained parallel nested loop structure.

```
// Distribute the threads to lines
Kokkos::parallel_for("Fast sweeps along x",
    range_policy(0, Ny),
     KOKKOS_LAMBDA(const int iy)
{
     // Slice to obtain the local integrals to which
    we apply
     // the convolution kernel to the entire line
     // ...
});
```

Listing 4 Kokkos kernel for the fast-convolution algorithm.

Appendix C Sixth-Order WENO Quadrature

We provide the various expressions for the coefficients and smoothness indicators used in the reconstruction process for $J_R^{(r)}$. Defining $\nu \equiv \alpha \Delta x$, the coefficients for the fixed stencils are given in [2] as follows:

$$
\begin{aligned}
c_{-3}^{(0)} &= \frac{6 - 6\nu + 2\nu^2 - (6 - \nu^2)e^{-\nu}}{6\nu^3}, \\
c_{-2}^{(0)} &= -\frac{6 - 8\nu + 3\nu^2 - (6 - 2\nu - 2\nu^2)e^{-\nu}}{2\nu^3}, \\
c_{-1}^{(0)} &= \frac{6 - 10\nu + 6\nu^2 - (6 - 4\nu - \nu^2 + 2\nu^2)e^{-\nu}}{2\nu^3}, \\
c_{0}^{(0)} &= -\frac{6 - 12\nu + 11\nu^2 - 6\nu^3 - (6 - 6\nu + 2\nu^2)e^{-\nu}}{6\nu^3},
\end{aligned}
$$

$$
\begin{aligned}
c_{-2}^{(1)} &= \frac{6 - \nu^2 - (6 + 6\nu + 2\nu^2)e^{-\nu}}{6\nu^3}, \\
c_{-1}^{(1)} &= -\frac{6 - 2\nu - 2\nu^2 - (6 + 4\nu - \nu^2 - 2\nu^3)e^{-\nu}}{2\nu^3}, \\
c_{0}^{(1)} &= \frac{6 - 4\nu - \nu^2 + 2\nu^3 - (6 + 2\nu - 2\nu^2)e^{-\nu}}{2\nu^3}, \\
c_{1}^{(1)} &= -\frac{6 - 6\nu + 2\nu^2 - (6 - \nu^2)e^{-\nu}}{6\nu^3},
\end{aligned}
$$

$$
\begin{aligned}
c_{-1}^{(2)} &= \frac{6 + 6\nu + 2\nu^2 - (6 + 12\nu + 11\nu^2 + 6\nu^3)e^{-\nu}}{6\nu^3}, \\
c_{0}^{(2)} &= -\frac{6 + 4\nu - \nu^2 - 2\nu^3 - (6 + 10\nu + 6\nu^2)e^{-\nu}}{2\nu^3}, \\
c_{1}^{(2)} &= \frac{6 + 2\nu - 2\nu^2 - (6 + 8\nu + 3\nu^2)e^{-\nu}}{2\nu^3}, \\
c_{2}^{(2)} &= -\frac{6 - \nu^2 - (6 + 6\nu + 2\nu^2)e^{-\nu}}{6\nu^3}.
\end{aligned}
$$

The corresponding linear weights are

$$d_0 = \frac{6 - \nu^2 - (6 + 6\nu + 2\nu^2)e^{-\nu}}{3\nu(2 - \nu - (2 + \nu)e^{-\nu})},$$
$$d_2 = \frac{60 - 60\nu + 15\nu^2 + 5\nu^3 - 3\nu^4 - (60 - 15\nu^2 + 2\nu^4)e^{-\nu}}{10\nu^2(6 - \nu^2 - (6 + 6\nu + 2\nu^2)e^{-\nu})},$$
$$d_1 = 1 - d_0 - d_2.$$

To obtain the analogous expressions for $J_L^{(r)}$, we exploit the "mirror-symmetry" property of WENO reconstructions. That is, one can keep the left side of each of the expressions, then reverse the order of the expressions on the right. Expressions for calculating one particular smoothness indicator, if interested, can be found in [2].

References

1. Christlieb, A., Guo, W., Jiang, Y.: A kernel-based high order "explicit" unconditionally stable scheme for time dependent Hamilton–Jacobi equations. J. Comput. Phys. **379**, 214–236 (2019)
2. Christlieb, A., Guo, W., Jiang, Y., Yang, H.: Kernel based high order "explicit" unconditionally-stable scheme for nonlinear degenerate advection-diffusion equations. J. Sci. Comput. **82:52**(3), 1–29 (2020)
3. Christlieb, A., Sands, W., Yang, H.: A kernel-based explicit unconditionally stable scheme for hamilton-jacobi equations on nonuniform meshes. J. Comput. Phys. **415**, 1–25 (2020). Art. No. 109543
4. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**, 3202–3216 (2014)
5. Causley, M., Christlieb, A., Ong, B., Van Groningen, L.: Method of lines transpose: an implicit solution to the wave equation. Math. Comput. **83**(290), 2763–2786 (2014)
6. Causley, M.F., Cho, H., Christlieb, A.J., Seal, D.C.: Method of lines transpose: high order l-stable O(N) schemes for parabolic equations using successive convolution. SIAM J. Numer. Anal. **54**(3), 1635–1652 (2016)
7. Causley, M., Cho, H., Christlieb, A.: Method of lines transpose: energy gradient flows using direct operator inversion for phase-field models. SIAM J. Sci. Comput. **39**(5), B968–B992 (2017)
8. Cheng, Y., Christlieb, A.J., Guo, W., Ong, B.: An asymptotic preserving maxwell solver resulting in the darwin limit of electrodynamics. J. Sci. Comput. **71**(3), 959–993 (2017)
9. Christlieb, A., Guo, W., Jiang, Y.: A weno-based method of lines transpose approach for vlasov simulations. J. Comput. Phys. **327**, 337–367 (2016)
10. Gottlieb, S., Shu, C.-W., Tadmor, E.: Strong stability-preserving high-order time discretization methods. SIAM Rev. **43**(1), 89–112 (2001)
11. Salazar, A., Raydan, M., Campo, A.: Theoretical analysis of the exponential transversal method of lines for the diffusion equation. Numer. Methods Partial Differ. Equ. **16**(1), 30–41 (2000)
12. Schemann, M., Bornemann, F.A.: An adaptive rothe method for the wave equation. Comput. Vis. Sci. **1**(3), 137–144 (1998)
13. Biros, G., Ying, L., Zorin, D.: An embedded boundary integral solver for the unsteady incompressible Navier–Stokes equations (preprint) (2002)
14. Biros, G., Ying, L., Zorin, D.: A fast solver for the stokes equations with distributed forces in complex geometries. J. Comput. Phys. **193**, 317–348 (2004)
15. Chiu, S.-H., Moore, M.N.J., Quaife, B.: Viscous transport in eroding porous media. J. Fluid Mech. **893**, A3 (2020). https://doi.org/10.1017/jfm.2020.228
16. Kropinski, M.C.A., Quaife, B.D.: Fast integral equation methods for rothe's method applied to the isotropic heat equation. Comput. Math. Appl. **61**, 2436–2446 (2011)
17. Quaife, B.D., Moore, M.N.J.: A boundary-integral framework to simulate viscous erosion of a porous medium. J. Comput. Phys. **375**, 1–21 (2018)
18. Wang, H., Lei, T., Li, J., Huang, J., Yao, Z.: A parallel fast multipole accelerated integral equation scheme for 3d stokes equations. Int. J. Numer. Meth. Eng. **70**, 812–839 (2007)
19. Ying, L., Biros, G., Zorin, D.: A high-order 3d boundary integral equation solver for elliptic pdes in smooth domains. J. Comput. Phys. **219**, 247–275 (2006)
20. Saad, Y., Schultzn, M.H.: Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **7**, 856–869 (1986)

21. Bruno, O.P., Lyon, M.: High-order unconditionally stable fc-ad solvers for general smooth domains i. basic elements. J. Comput. Phys. **229**, 2009–2033 (2010)
22. Bruno, O.P., Lyon, M.: High-order unconditionally stable fc-ad solvers for general smooth domains ii. elliptic, parabolic and hyperbolic pdes. theoretical considerations. J. Comput. Phys. **229**, 3358–3381 (2010)
23. Douglas Jr., J.: On the numerical integration of $\partial_{xx}U + \partial_{yy}U = \partial_t U$ by implicit methods. J. Soc. Ind. Appl. Math. **3**, 42–65 (1955)
24. Douglas Jr., J.: Alternating direction methods for three space variables. Numer. Math. **3**, 41–63 (1962)
25. Peaceman, D.W., Rachford Jr., H.H.: The numerical solution of parabolic and elliptic differential equations. J. Soc. Ind. Appl. Math. **3**, 28–41 (1955)
26. Albin, N., Bruno, O.P.: A spectral fc solver for the compressible navier-stokes equations in general domains i: Explicit time-stepping. J. Comput. Phys. **230**, 6248–6270 (2011)
27. Anderson, T.G., Bruno, O.P., Lyon, M.: High-order, dispersionless "fast-hybrid" wave equation solver. part i: $\mathcal{O}(1)$ sampling cost via incident-field windowing and recentering. SIAM J. Sci. Comput. **42**, 1348–1379 (2020)
28. Causley, M.F., Christlieb, A.J.: Higher order a-stable schemes for the wave equation using a successive convolution approach. SIAM J. Numer. Anal. **52**(1), 220–235 (2014)
29. Causley, M. F., Christlieb, A. J., Guclu, Y., Wolf, E.: Method of lines transpose: A fast implicit wave propagator. arXiv preprint arXiv:1306.6902, (2013)
30. Shu, C.-W.: High order weighted essentially nonoscillatory schemes for convection dominated problems. SIAM Rev. **51**(1), 82–126 (2009)
31. Hornung, R.D., Keasler, J.A.: The raja portability layer: Overview and status. Lawrence Livermore National Laboratory (LLNL), Livermore. Tech. Rep. (2014). https://doi.org/10.2172/1169830
32. Grete, P., Glines, F. W., O'Shea, B. W.: K-athena: A performance portable structured grid finite volume magnetohydrodynamics code (2019). arXiv: 1905.04341 [cs.DC]
33. White, C.J., Stone, J.M., Gammie, C.F.: An extension of the athena++ code framework for grmhd based on advanced riemann solvers and staggered-mesh constrained transport. Astrophys. J. Suppl. **225**, 2 (2016)

Affiliations

Andrew J. Christlieb[1] · Pierson T. Guthrey[1,2] · William A. Sands[1] · Mathialakan Thavappiragasm[1,3]

Andrew J. Christlieb
christli@msu.edu

Pierson T. Guthrey
guthrey1@llnl.gov

Mathialakan Thavappiragasm
mathialakan@gmail.com

1 Department of Computational Mathematics, Science and Engineering, Michigan State University, East Lansing, MI 48824, USA

2 Present Address: Lawrence Livermore National Laboratory, Livermore, CA 94550, USA

3 Present Address: Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA