Pekkilä, Johannes; Väisälä, Miikka S.; Käpylä, Maarit J.; Rheinhardt, Matthias; Lappi, Oskar

**Scalable communication for high-order stencil computations using CUDA-aware MPI**

*Published in:* PARALLEL COMPUTING

*DOI:* 10.1016/j.parco.2022.102904

Published: 01/07/2022

*Document Version*
Publisher's PDF, also known as Version of record

*Published under the following license:*
CC BY

*Please cite the original version:*
Scalable communication for high-order stencil computations using CUDA-aware MPI\textsuperscript{\textcopyright}

Johannes Pekkilä\textsuperscript{a,e}, Miikka S. Väisälä\textsuperscript{b}, Maarit J. Käpylä\textsuperscript{a,c,d}, Matthias Rheinhardt\textsuperscript{a}, Oskar Lappi\textsuperscript{a,e}

\textsuperscript{a} Department of Computer Science, Aalto University, Konemiehentie 2, 02150 Espoo, Finland
\textsuperscript{b} Academia Sinica, Institute of Astronomy and Astrophysics, Roosevelt Road No. 1 Sec. 4, 10617 Taipei, Taiwan
\textsuperscript{c} Max Planck Institute for Solar System Research, Justus-von-Liebig-Weg 3, D-37077 Göttingen, Germany
\textsuperscript{d} Nordita, KTH Royal Institute of Technology and Stockholm University, Hannes Alfvéns Väg 12, SE-10691 Stockholm, Sweden
\textsuperscript{e} Faculty of Science and Engineering, Åbo Akademi University, Tuomiokirkontori 3, 20500 Turku, Finland

A R T I C L E  I N F O

Keywords:
High-performance computing
Graphics processing units
Stencils
Computational physics
Magnetohydrodynamics

A B S T R A C T

Modern compute nodes in high-performance computing provide a tremendous level of parallelism and processing power. However, as arithmetic performance has been observed to increase at a faster rate relative to memory and network bandwidths, optimizing data movement has become critical for achieving strong scaling in many communication-heavy applications. This performance gap has been further accentuated with the introduction of graphics processing units, which can provide by multiple factors higher throughput in data-parallel tasks than central processing units. In this work, we explore the computational aspects of iterative stencil loops and implement a generic communication scheme using CUDA-aware MPI, which we use to accelerate magnetohydrodynamics simulations based on high-order finite differences and third-order Runge-Kutta integration. We put particular focus on improving intra-node locality of workloads. Our GPU implementation scales strongly from one to 64 devices at 50%–87% of the expected efficiency based on a theoretical performance model. Compared with a multi-core CPU solver, our implementation exhibits 20–60× speedup and 9–12× improved energy efficiency in compute-bound benchmarks on 16 nodes.

1. Introduction

Iterative stencil loops (ISLs) belong to a class of algorithms, in which data points are updated by sampling their neighborhood in a fixed pattern called a stencil. ISLs, or more generally, computations on a structured grid, have been identified as one of the major recurring computational patterns in high-performance computing (HPC) due to their prevalence in science and engineering [3]. Common applications include image processing [4,5] and solving partial differential equations (PDEs) [6,7]. Because each data point can be updated independently, ISLs can usually be processed efficiently on parallel computers.

Over the last ten years, compute nodes in HPC have been gradually shifting from homogeneous systems into systems housing multiple general-purpose processors and domain-specific accelerators; graphics processing units (GPUs) are the most commonly used ones. Of the TOP500 HPC systems, 27% house one or more NVIDIA GPUs per node [8]. As specialized co-processors, GPUs can provide multiple times higher throughput in data-parallel tasks than central processing units (CPUs),\textsuperscript{1} which makes them an attractive platform for ISLs. Optimization techniques for accelerating ISLs on a single GPU have been extensively studied in previous works [9,10].

In computational sciences, large stencils are often used to obtain sufficiently accurate results. For example in astrophysical fluid simulations, the fluids are frequently in a state of fully developed turbulence, and high-order difference schemes, high-resolution discretization, and double-precision arithmetic can be useful, or even mandatory, for discerning small-scale details. In large-scale simulations, data movement is a likely bottleneck, as the amount of communication decreases at a
lower rate than computation when the number of parallel processors is increased. This will be elaborated on in Section 3.2.

Reducing the performance impact of data movement is a notable challenge. Wulf [11], Patterson [12], and others [13, 14], observed that arithmetic performance of microprocessors increases at a faster rate relative to the improvements in memory bandwidth, and bandwidth improves at a faster rate than memory access latency. The performance of network interconnects has followed a similar trend. In a ten-year span, the operational performance of a HPC node has increased 26-fold [8], whereas the network interconnect bandwidth has increased only 6.25-fold. As network bandwidth is generally an order of magnitude less than off-chip memory bandwidth, optimizing inter-node communication is critical for achieving efficient scaling to a large number of compute nodes.

In this work, we address two major challenges with data movement in large-scale applications. Firstly, we estimate the upper bound for communication performance of $d$-dimensional stencil computations by defining a communication cost function for idealized hardware and solving an integer program to find the minimum required communication time. Secondly, we implement a scalable communication scheme, in which data movement latencies are hidden by pipelining computation and communication. We apply our implementation to a practical simulation setup commonly used in fluid dynamics research and compare the achieved performance to the theoretical maximum. The simulation setup employs high-order discretizations in space and time based on finite differences and Runge–Kutta integration methods.

The structure of this paper is as follows. In Section 2, we introduce the computational aspects and scaling properties of ISLs. In Section 3.1, we describe the performance model used for finding theoretical performance limits and evaluating the scaling of our implementation. In Sections 3.2 and 3.3, we present a communication cost function for stencil computations, find the upper bound for communication performance, and present the technical details of our implementation. We give a brief description of the magnetohydrodynamics solver used for benchmarks in Section 4. Finally, we present and discuss our results in Sections 5 and 6, and conclude the paper in Section 7.

2. Background

In ISLs, data points are updated by sampling their neighborhoods in a fixed memory access pattern, called a stencil (see Fig. 1). The radius $r$ of a stencil is the maximal Chebyshev distance from $c$

$$r = \max \left( \max_{i \in S} |s_i - c_i| \right).$$

where $c = (c_0, c_1, \ldots, c_d)$ is the spatial index of the point being updated and $S$ the set of stencil points. The exact shape of a stencil depends on the application. In the simplest case, a stencil contains all the points within its radius $S = \{s \in \mathbb{Z}^d : |s| \leq r\}$. In this work, we focus on stencils of this form and its subsets with the same radius.

Data points are stored in a $d$-dimensional array, usually representing a structured grid with regular connectivity. In this context, we refer to data points as cells. During an iteration step, the cells belonging to the computational domain are updated according to some stencil operation. When updating cells near the boundaries, some stencil points fall outside the computational domain. The set of these points is henceforth referred to as the halo. We use the term grid to denote the set of all cells that belong to either the halo or the computational domain.

We use $d$-tuples of the form $\Phi = (\phi_1, \phi_2, \ldots, \phi_d)$ to denote domains, where $\phi_i \in \mathbb{N}$ is the number of cells in dimension $i$ and the total number of elements in $\Phi$ is $C_\Phi = \prod_{i=1}^{d} \phi_i$. Using this notation, the domain of the grid is $M = (n_1 + 2r, n_2 + 2r, \ldots, n_d + 2r)$, where $N = (n_1, n_2, \ldots, n_d)$ is the computational domain (see Fig. 2). When processing ISLs on distributed systems, $M$ must be decomposed into $C_\Phi$ computational subdomains, with $p$ subdomains in dimension $i$. Each computational subdomain $N'$ is also surrounded by a halo, forming a subgrid $M'$. For simplicity, we assume all subdomains to have the same size and require, that each node is assigned exactly one subdomain. This enables us to regard $C_\Phi$ henceforth as the number of nodes. The number of cells in $N'$ and $M'$ can now be written as

$$C_{N'} = \prod_{i=1}^{d} \frac{n_i - 2r}{\phi_i},$$

and

$$C_{M'} = \prod_{i=1}^{d} \left( \frac{n_i}{\phi_i} + 2r \right).$$

When processing ISLs on two or more nodes, a portion of the halo, local to one node, overlaps with the computational subdomain assigned to a neighboring node. After each update of the neighboring subdomain, the data corresponding to the halo segment must be communicated to the initial node. This is called a halo exchange, as communication happens both ways for nodes sharing a boundary.

2.1. Previous work

In previous work, we presented a library for accelerating ISLs on GPUs, called Astaroth [16]. It provides an application programming interface to the GPU resources and a domain-specific language (DSL) for writing stencil kernels. Astaroth was inspired by an earlier proof-of-concept hydrodynamics solver presented in [17, 18], which was originally created for the purpose of exploring how to accelerate the computational methods used by the Pencil Code (PC) [19]. Later, the library was extended to support computations on multiple devices on a single node using CUDA peer-to-peer memory transfers [20]. In this work, we extend Astaroth to support computations on multiple nodes.

There are several libraries and frameworks designed for accelerating stencil codes. The one, which is closest related to Astaroth, is Physis.
which has also been designed for accelerating stencil computations on GPUs and performs source-to-source translation from a DSL to CUDA and MPI. However, communication is carried out explicitly via host memory. Another library close to our work is LibGeoDecomp [22], which is a mature, stencil-focused library supporting hierarchical geometric partitioning and load balancing on heterogeneous systems, including GPUs. Instead of a DSL, LibGeoDecomp provides C++ templates for describing the parameters for stencil kernels. Of PDE-specific libraries similar to our work, Fargo3D [23] is focused on accelerating MHD simulations, supports multiple GPUs and performs communication using CUDA-aware MPI. Instead of handling the memory of each GPU explicitly as in Astaroth, Fargo3D uses Unified Virtual Addressing (UVA) to manage the resources on a node. Yet another framework focused on advection-diffusion type problems is PyFR [24], which provides hierarchical and graph-based partitioning based on the Metis [25] and Scotch [26] software packages. The Cactus Framework has adopted a more generic approach, providing tools for large-scale parallelization of various types of tasks, including stencil computations [27,28].

The main difference of Astaroth to existing libraries is its specialized focus on improving cache reuse in stencil computations, where the working set, that is, the data required to update a small group of cells, is too large to fit into the caches of a GPU. As such, Astaroth is especially suited for multiphysics simulations, which use high-order stencils and double precision, and require data from several coupled fields to update a cell. For more details on the single-GPU optimization techniques and code generation of Astaroth, we refer the reader to [16].

3. Methodology

3.1. Performance modeling

Performance models are useful for estimating theoretical performance limits, which can be used to determine whether further optimizations are needed or to calculate the expected scaling profile without having to queue for compute resources. In this section, we describe a simple performance model, which we use to find the upper bound for scaling performance. While the model has likely been introduced before, we are not aware of an established name.

In the following discussion, we use generic terminology and focus on ISL-specific definitions from Section 3.2 onward. We use the term processing element to refer to a generic computational unit that performs work in parallel, such as a node or a device. The terms host and device are used to refer to the CPU and GPU, respectively. Throughout this work, we use the term CPU to refer to the multi-core processor located on a single CPU socket. Finally, we use the term communication to refer to data movement within or between non-local memory systems.

As processing elements operate in parallel, the running time is the maximum time it takes for an element to complete its task. We denote the computational workload per processing element as $W$ data items and the amount of communication as $Q$ data items. Furthermore, $\alpha$ is the operational capability of the hardware measured as data item updates per second, and $\beta$ the rate at which data elements can be communicated. The time taken by computation and communication is therefore $t_W = W/\alpha$ and $t_Q = Q/\beta$, respectively. In this work, we measure $\alpha$ empirically by benchmarking the program on a single device and calculate $\beta$ based on the theoretical network bandwidth and the size of a data item.

As computation and communication must be carried out in parallel to achieve efficient scaling, the running time of an ideal implementation is $\max(t_W, t_Q)$. Taking inspiration from Amdahl's law, we further include a term $t_0$ to capture the time taken in the sequential portion of the program. We use the term sequential to refer to computations that cannot be carried out in parallel with communication.

We can now model the running time as

$$\max(t_W, t_Q) + t_0 \quad \text{(4)}$$

In this form, the model produces a scaling profile that is familiar from multi-processor benchmarks, see Fig. 3. When the performance of a kernel is limited by compute performance, it is said to be compute bound. In this case, $t_W > t_Q$. The kernel is communication bound when the opposite is true. Alternatively, we can express the performance bounds in terms of operational intensity $I = W/Q$, where the limiter is compute performance if $I > \alpha/\beta$ [29].

3.2. Domain decomposition

There are two major considerations for implementing a communication scheme for distributed applications. Firstly, the problem domain must be decomposed into $P$ subdomains, and secondly, the subdomains must be assigned to processing elements. In this section, we use hierarchical geometric partitioning [30,31] to find a decomposition and processor assignment for ISLs that minimize the communication surface area. In this approach, the partitioning is optimized recursively on each level of the processing element hierarchy. We consider two levels: node- and device-level. Furthermore, we assume that the network topology is a fat tree and the devices within a node are fully connected. These assumptions imply that the bandwidth between any pair of nodes, or devices within a node, is roughly the same and it is possible to form parallel connections to arbitrary many neighbors.

Ultimately, the goal is to balance workloads across processing elements and minimize data movement [32,33]. We use the term critical path to refer to the longest series of dependent operations that need to be performed sequentially to complete an iteration of a parallel algorithm. If we consider Eq. (4) to model running time at sufficient accuracy, we can minimize the communication surface area on the critical path by solving the integer programming problem

$$\arg\min_P \max(t_W, t_Q) + t_0 \quad \text{(5)}$$

subject to $p_i \in \mathbb{N}$, $\sum_{i=1}^P p_i = C_P$.

where $\max(t_W, t_Q) + t_0$ is the worst-case running time for processing a subdomain. Throughout this work, we use the term optimal decomposition to refer to $P$ which solves Eq. (5). First, we assume for the sequential portion of the program $t_0 = 0$. This implies that all computation and communication can be carried out in parallel, which is approximately the case when the implementation is sufficiently pipelined.

Next, we define $W$ and $Q$ for ISLs. In the following proofs, we first find the optimal node-level decomposition, and later expand the reasoning to include heterogeneous nodes containing multiple devices. The amount of local work per update step for each node is

$$W = C_N = \frac{C_N}{C_P} \quad \text{(6)}$$

As $W$ does not depend on the choice of the components of either $N$ or $P$, we can focus on the case $t_W < t_Q$. Furthermore, as $\beta$ is a
constant, the objective function in Eq. (5) simplifies to \( Q \). To simplify the definition of \( Q \), we assume that \( p_i \geq 3 \). \( \forall p_i \in P \). If the boundaries are periodic, which is the case in our tests, then the following definition also holds when \( p_i \geq 2 \). \( \forall p_i \in P \). The number of cells communicated during a halo exchange per node in the worst case is

\[
Q = 2(C_{M''} - C_{N''})
\]

(7)

The worst case behavior is witnessed when all halo cells must be exchanged. This occurs when each of the boundaries of a computational subdomain faces another subdomain.

As \( W \) is inversely proportional to \( C_P \) with coefficient \( C_N \), ISLs are expected to exhibit ideal scaling when the application is compute bound, that is, \( r_W \geq r_Q \). We use the term ideal scaling to refer to the case, where the performance grows linearly with the number of processing elements at 100% efficiency. As \( Q \) scales at a slower rate in comparison to \( W \), scaling efficiency is reduced when the performance is bound by data movement.

For relatively low \( C_P \), it is feasible to conduct an exhaustive search for the optimal decomposition. In other cases, a more sophisticated approach, such as using heuristics to reduce the search space, is likely needed. When using \( P \) as a static mapping, the solution can be stored in a lookup table for quick access. The optimal components of \( P \) for typical choices of \( N \) are listed in Appendix A. On node level, the workloads are inherently balanced, as we require that \( N' \) is the same for all nodes and the bandwidth between any pair is the same.

Next, we consider the case when a node contains multiple devices. The optimal decomposition can be found by recursively subdividing \( N' \) further to \( C_P \) devices available on a node. We denote the sizes of the per-device grid and computational domain as \( C_{M''} \) and \( C_{N''} \), respectively. Similar to the node-level definitions,

\[
C_{N''} = \prod_{i=1}^{d} \frac{p'_i}{p'_i - 1}
\]

(8)

\[
C_{M''} = \prod_{i=1}^{d} \left( \frac{p'_i}{p'_i - 1} + 2r \right)
\]

(9)

For the rest of this section, we refer to \( N' \) as the computational domain and \( N'' \) as the computational subdomain.

To define the amount of communication performed via the intra-node communication fabric, we introduce the concept of a local subgrid \( L'' \). The local subgrid comprises the cells in the computational subdomain that overlaps with the computational subdomain of any of the intra-node neighbors (see Fig. 4). The size of the local subgrid is

\[
C_{L''} \geq \prod_{i=1}^{d} \left( \frac{p'_i}{p'_i - 1} + r \cdot 1_{p'_i \geq 2} \right)
\]

(10)

where \( 1_{p'_i \geq 2} \) is an indicator function. Intuitively, if \( p'_i \geq 2 \), then there is at least one intra-node boundary. As intra- and inter-node communication can be carried out in parallel, the total communication time per device is ideally

\[
\max \left( \beta^{-1}_{\text{inter}} Q_{\text{inter}}, \beta^{-1}_{\text{intra}} Q_{\text{intra}} \right)
\]

(11)

Furthermore,

\[
Q_{\text{inter}} = C_{M''} - C_{L''}
\]

(12)

\[
Q_{\text{intra}} = C_{L''} - C_{N''}
\]

The dominant factor in Eq. (11) is typically \( \beta^{-1}_{\text{inter}} Q_{\text{inter}} \), as \( \beta^{-1}_{\text{inter}} > \beta^{-1}_{\text{intra}} \) almost always on modern systems, and because

\[
C_{L''} + C_{N''} > 2C_{L''}
\]

(13)

holds for \( N, P \in \mathbb{N}^d \) and \( r \in \mathbb{N} \), we have \( Q_{\text{inter}} > Q_{\text{intra}} \). Therefore we can find the optimal intra-node decomposition by minimizing \( C_{M''} - C_{L''} \). By defining \( C_{L''} \) as the lower bound for intra-node communication, minimizing \( C_{M''} - C_{L''} \) maximizes worst-case performance. The optimal intra-node decompositions for \( P' \) are listed in Appendix A.

We can compare the rate of decrease in communication of spatial decomposition schemes by varying the degrees of freedom of \( P \). For example in a one-dimensional decomposition, regardless of the dimensionality of the grid, only one component of \( P \) is free while the others are bound to unity. The scaling of data movement in common spatial decomposition schemes is illustrated in Fig. 5. While a one-dimensional decomposition is easy to implement and scales reasonably well to a low number of nodes [20], it is clear that multi-dimensional decomposition is required for large-scale applications.

3.3. Implementation

In our implementation, we subdivide the computational domain recursively along each axis in succession and use Z-order indexing, also known as Morton order [34], to map processors to subdomains (Fig. 6(b)). The partitioning is given by \( P' = \text{morton}^{-1}(C_{P'} - 1) + (1, 1, 1) \), where \( C_{P'} = C_P C_P \). The function morton(\( \varphi \)) interleaves the binary representation of a multidimensional coordinate \( \varphi \) to obtain a one-dimensional index \( i \), and morton\(^{-1}(i) \) is its inverse. For example, morton\(^{-1}\) maps a binary index \( i = abcd\text{e}f \) to coordinate \( \varphi = (c\text{f}1, b\text{e}2, a\text{d}3) \).

The Z-order curve preserves locality to a relatively high degree, meaning that one-dimensional indices along the curve are likely mapped to multidimensional coordinates that are spatially nearby. If processes within a node are assigned contiguous MPI ranks, Z-order indexing can be used to enhance intra-node locality of the subdomains. Compared with the communication-optimal decomposition discussed in
Section 3.2 and Appendix A, the Z-order mapping minimizes, or nearly minimizes, the data movement on the critical path the case where \( n_x = n_y = n_z \) and \( C_{\text{CPU}} = 4 \).

In contrast to more intuitive processor mappings, locality-preserving space-filling curves can provide better load balancing and reduced data movement. Consider the case where subdomains are assigned to processors in a row-wise scan pattern (Fig. 6(a)). In this case it is possible for neighboring processors to communicate a different amount of data to inter-node neighbors, which incurs a load imbalance. In the three-dimensional case, \( C_{\text{CPU}} = 8 \), \( C_{\text{CPU}} \gg C_{\text{CPU}} \), and with subdomains having equal dimensions, the number of faces shared with inter-node neighbors ranges from four to five. With Z-order indexing, each process has exactly three inter-node-facing edges. The amount of data communicated along the critical path is also larger with row-wise indexing, as the worst-case size of the halo segments is larger when decomposing the intra-node domain in one dimension instead of three.

The use of space-filling curves in large-scale computations has been explored by, for example, Tsuzuki [35] and Li [36]. It should be noted, however, that on practical hardware, pairing our communication cost function with an established graph-based partitioner, such as Scotch [26], may yield even higher-quality decompositions. In this work, we determine that Z-order mapping is sufficiently communication-efficient for our purposes, and leave rigorous comparisons with more established methods for future work.

Several MPI implementations, notably MVAPICH and OpenMPI, provide support for transfers to/from CUDA-allocated device memory. As memory transfers are routed automatically via the fastest communication fabric and the programmer can treat device pointers simply like host pointers, implementing device-to-device communication with CUDA-aware MPI is straightforward. However, special care is needed to ensure the correct pipelining of compute kernels and data transfers.

Furthermore, the GPUDirect remote direct memory access (RDMA) technology has been introduced to enable low-latency inter-node device-to-device communication directly via the network interface controller, bypassing host memory. However, due to lower bandwidth, device-to-device RDMA generally provides better performance only when sending small messages of size 32 KiB or less [37]. Larger messages are buffered through host memory in a pipelined fashion. Almost all messages sent in our implementation are above this threshold.

Before initiating halo exchange, the data elements corresponding to each segment are packed into a contiguous buffer. This has two advantages. Firstly, the bulk of memory operations is performed within the faster local memory and secondly, the throughput for sending a few larger messages is generally higher than sending several smaller ones.

To execute packing, halo exchange, and unpacking in a pipelined fashion, each segment is associated with a distinct non-blocking CUDA stream. To exchange the segments, we use functions \( \text{MPI}_\text{I}send \) and \( \text{MPI}_\text{Irecv} \). The update of the inner segment is launched simultaneously with the halo exchange. It is likely that the packing and integration kernels cannot run in parallel due to high resource usage. Because the halo exchange depends on packed data being available, it is therefore critical to ensure that packing completes before starting the inner segment update. This enables concurrency of computation and communication. We handle this by assigning higher priorities to packing streams, but one could alternatively add an additional synchronization step after packing.

After all data segments have been received, as indicated by the completion of \( \text{MPI}_\text{Waitall} \), we launch the kernels for updating the outer segments. The dependencies and execution order of these from the halo, while the outer segments can be updated only after communication has finished. In three-dimensional grid decompositions, the domain of the inner segment comprises \((n'_x - 2r, n'_y - 2r, n'_z - 2r)\) cells. Similar to the inner and outer segments, we partition the halo into conceptual segments, where each segment overlaps with the computational domain of a distinct neighbor. The halo segments are illustrated in Fig. 7. From largest to smallest, we call the segments sides, edges, and corners. There are 6 side segments, 12 edge segments, and 8 corner segments. Each halo segment can be uniquely identified by the spatial index of its first element, which we use as a message tag. A segment at index \( s \) is mapped to the computational domain of the receiving device as \( s' = ((s_x - r) \mod n'_x) + r \).

To execute packing, halo exchange, and unpacking in a pipelined fashion, each segment is associated with a distinct non-blocking CUDA stream. To exchange the segments, we use functions \( \text{MPI}_\text{I}send \) and \( \text{MPI}_\text{Irecv} \). The update of the inner segment is launched simultaneously with the halo exchange. It is likely that the packing and integration kernels cannot run in parallel due to high resource usage. Because the halo exchange depends on packed data being available, it is therefore critical to ensure that packing completes before starting the inner segment update. This enables concurrency of computation and communication. We handle this by assigning higher priorities to packing streams, but one could alternatively add an additional synchronization step after packing.

After all data segments have been received, as indicated by the completion of \( \text{MPI}_\text{Waitall} \), the launch the kernels for updating the outer segments. The dependencies and execution order of these

![Fig. 6.](image1)

(a) Row-wise scan.

(b) Z-order indexing.

Fig. 6. The mapping of subdomains to devices using row-wise scan and two-dimensional Z-order indexing. Device and node boundaries are indicated with dotted and dashed lines, respectively.

![Fig. 7.](image2)

Fig. 7. Halo segments. Each segment is sent to a unique neighbor. The inner and outer segments of the computational subdomain are not visible.
non-linear problems. Our implementation supports the approach of the Pencil Code: we use a non-standard equations of hydrodynamics. when neglecting the magnetic field, the MHD equations reduce to the same methods can also be used in general fluid mechanics, because Earth’s magnetosphere, interstellar medium, and star formation. It is used to study phenomena such as solar magnetic activity, the time scales of turbulence and the large-scale phenomena of interest, the dynamics of plasma can be approximated as a continuous conducting fluid coupled with a magnetic field. Astrophysical MHD problems are usually highly non-linear and require substantial computing resources because high resolutions in time and/or space, and long integration times, due to the vastly differing time scales of turbulence and the large-scale phenomena of interest, are required in realistic setups. In addition, resource requirements inflate when an extended parameter space has to be scanned. MHD has a wide range of applications in multiple astrophysical domains. It is used to study phenomena such as solar magnetic activity, the Earth’s magnetosphere, interstellar medium, and star formation. The same methods can also be used in general fluid mechanics, because when neglecting the magnetic field, the MHD equations reduce to the standard equations of hydrodynamics.

The MHD code has been utilized in a recent work [20], with single-node parallelization, to explore the kinematic growth phase of turbulent MHD dynamos in the isothermal regime. Our MHD solver follows the approach of the Pencil Code: we use a non-conservative but high-order finite-difference method to explore the non-linear problems. Our implementation supports 2nd-, 4th-, 6th-, and 8th-order central finite differences and time integration by a third-order Runge–Kutta (RK3) method [6,39]. The stencils used to compute the derivatives contain the points

$$S = \{ z(x \pm y) : (x, y, k) \in (i, j, k, 0) \mid |z| \leq r, r \in Z \}.$$  

(14)

where $i, j, k$ is the standard basis of a 3D space. We use the term kth-order stencil to refer to stencils used to compute kth-order accurate central finite differences, where $k = 2r$. The related stencils are illustrated in Fig. 9. The full set of MHD equation is listed in Appendix B. For more details, we refer the reader to [16,20].

5. Results

The tests were conducted on a cluster consisting of a total of 80 SuperServer 1029GQ-TVRT nodes connected in a fat tree network [40]. Each node houses two Intel Xeon Gold 6230 Cascade Lake 20-core processors running at 2.1 GHz and four Tesla V100-SXM2-32 GB GV100GL (rev a1) GPUs running at 1.53 GHz. The stated thermal design power (TDP) of an Intel Xeon Gold 6230 CPU is 125 watts [2], and the stated TDP of a GV100GL GPU is 300 watts [1]. Each GV100GL is connected to the other three GPUs via pairs of NVLink 2.0 connections, providing 91 Gb/s bidirectional bandwidth per pair. The total intra-node NVLink bandwidth per GPU is 270 Gb/s. Each node houses two Mellanox ConnectX-6 InfiniBand HDR100 MT28908 adapters, which provide 23 Gb/s bidirectional inter-node bandwidth per adapter [15]. Error-correcting codes (ECC) were enabled in all tests. We confirmed the transfer rates by measuring the time to transfer 12 MB data blocks, which is the same size as the largest individual halo segment transferred in simulations employing 256³ cells. The effective device-to-device intra-node bandwidth was 86 Gb/s and inter-node bandwidth 40.8 Gb/s.

Astartorh [41], commit 3804e72, was compiled using GCC 8.3.0, CUDA toolkit 10.1.168, and OpenMPI 4.0.3. One MPI task was assigned per GPU. Each multi-core CPU controlled two GPUs closest to it in the node topology and a pair of GPUs shared access to one network interface controller. The rendezvous protocol used by the Unified Communication X (UCX) framework was configured by setting UCX_RDNV_THRES=16384, UCX_RDNV_SCHEME=rocm_copy, UCX_MAX_RDNV_RAILS=4, as this gave the best performance on the tested hardware. It should be noted, that the optimal configuration for the rendezvous protocol is system specific.

To evaluate whether our implementation is competitive with established work used in production, we compared the scaling performance of Astaroth with that of Pencil Code [19], commit 7ddde40. The simulation setup is available at [42].

The Pencil Code was benchmarked on two clusters: Puhti and Mahti. The Puhti benchmarks were run on a CPU-only partition of the same cluster as the GPU tests, which also houses two Intel Xeon Gold 6230 processors but only one Mellanox HDR100 network interface controller (NIC) per node. The effective bidirectional inter-node bandwidth in our experiments was roughly 20 Gb/s. The network topology of the Puhti system is a fat tree [43]. A compute node on the Mahti system houses two AMD Rome 7H12 64-core CPUs running at 2.6 GHz, where each multi-core CPU is split into 4 NUMA domains and the TDP of a single CPU is 280 watts [44], 256 Gb memory, and a single Mellanox HDR200 NIC. The network topology on Mahti is Dragonfly+ [45]. The effective bidirectional inter-node bandwidth was roughly 39 Gb/s. We compiled PC using the highest optimization level (O2) recommended for production runs and used compilers tuned for the clusters. On Puhti, we used Intel compiler version 19.0.4 and HPCX-MPI 2.4.0, and on Mahti, Intel compiler 19.1.1 and OpenMPI 4.0.3. We refer to PC benchmarks run on the respective systems as PC-Puhti and PC-Mahti hereafter.

We chose PC for comparison for the following reasons: First, it is a mature MHD-solver widely used for production in large-scale astrophysical simulations [46]. Secondly, the equations and methods used in Astaroth and PC are otherwise identical, with the exception that Astaroth uses a one-pass modification [17] of the two-pass RK-2N scheme used in PC [19,39]. We measured that the one-pass approach gives roughly a 20% performance advantage in compute-bound tests compared to the two-pass scheme. Thirdly, comparing CPU and GPU scaling gives us an indicator, whether GPU-applications can be competitive with established CPU solvers in large-scale HPC applications.
in terms of throughput and energy efficiency. The perceived bottleneck of CPU-GPU communication has been a cause of concern. Fourthly, PC uses simple axis-wise partitioning supplied by the user and row-wise scan indexing to map subdomains to processors, in contrast to Astaroth’s Z-order mapping that favors the assignment of intra-node neighbors to nearby subdomains. Finally, CPU and GPU applications both use the same communication fabric and implementing a communication scheme for either architecture is code-wise nearly identical. While computation throughput is higher on GPUs in our test case, communication-wise there is little difference with CPU applications.

In the benchmarks, we measured the time to complete an RK3 integration step. The simulation variables were initialized to random values in the range [0, 1] and the timestep was set to a constant \( \Delta t = 1.19209 \times 10^{-7} \). The simulation was run for 100 warm-up steps before measuring the running time of 1000 integration steps. As an exception, we timed only 100 steps in tests which included \( \geq 512^3 \) cells due to longer integration times. Double precision was used in all tests.

We verified the results by comparing the simulation state after an integration step with a model solution, which was obtained with a single-core CPU solver logically equivalent with the GPU solver. For a model value \( m \), the unit in the last place (ulp) is

\[
\epsilon = 2^\lceil \log_2 |m| - p + 1 \rceil .
\]  

where \( p \) is the precision of the floating-point number. For double-precision, \( p = 53 \). The absolute error in ulps for a candidate value \( c \) is calculated as

\[
\frac{|m - c|}{\epsilon} .
\]

We verified the solver on 1–16 devices in problem sets consisting of \( 256^3 \) cells. In all cases, the maximum absolute error was \( \leq 2 \) ulps. We deemed this to be within acceptable limits, as slight round-off errors are expected to accumulate when rounded intermediate values are used in calculations.

We tested our implementation in six benchmarks. In all of the following figures, the performance is shown as the median time per cell per integration step, where each step comprises three ISL iterations. Sixth-order stencils were used unless otherwise mentioned. We confirmed that processes within a node were assigned contiguous MPI ranks as assumed in Section 3.3.

Firstly, we measured the effective integration time per cell for various grid sizes (Fig. 10). For problem sizes consisting of \( 256^3 \), \( 512^3 \), and \( 1024^3 \) cells, we saw scaling to 64 devices at 18%, 43%, and 87% parallel efficiency, respectively, compared to ideal scaling. Theoretical strong scaling, calculated using the performance model discussed in Section 3.1, is shown in Fig. 11. We used \( \varepsilon^{-1} = 2.2 \) ns as the computational performance, which we determined empirically by measuring the integration step time on a single device. The communication performance \( \beta^{-1} = 3.9 \) ns was determined theoretically based on the data size per cell in bytes and the maximum network bandwidth of 46 GiB/s. Compared with the theoretical model, the effective performance was 50%, 59%, and 87% of the theoretical maximum on 64 devices. Our implementation exhibited near-ideal scaling when compute-bound, whereas there was a notable drop in scaling efficiency when communication started to dominate.

In our second test, we isolated computation and communication to study the performance of the system further. Our results are presented in Fig. 12. As can be seen, communication on a single node (up to four devices) is completely hidden, which is enabled by the relatively high device-to-device bandwidth and the devices being fully connected within a node. Inter-node communication is more sensitive to increases in the amount of communicated data. The portion of the halo communicated to inter-node neighbors increases gradually from 1 to 8 nodes (4 to 32 devices). This can be seen as a stagnation in communication times between 4 to 16 devices. The effect from 16 to 32 devices is significantly weaker and no longer visible. Ideally, the measured integration time is the maximum of computation and communication times. This is not the case, and we can see an overhead of roughly 15–20% compared to computation and communication benchmarked individually. The overhead is likely caused by the sequential portion of our implementation. We also measured the overhead caused by synchronization, which was 0.02%–2% of the integration time. The use of this work did not require the communication of the corner halo segments, but this is not the case for all applications. For completeness, the benchmark employing the communication of the corner segments is also shown. The corner segments were small enough to trigger RDMA. The added communication of the corner halo segments increased running times roughly by 0.03%–10%.

In our third test, we measured weak scaling by assigning each device the same amount of work (Fig. 13). The interconnect boundaries are
clearly visible in the figure. The onset of intra-node communication can be seen as the transition between one and two devices, and the onset of inter-node communication between four and eight. The proportion of the node-level halo communicated to inter-node neighbors increases gradually from four devices, reaching the maximum at 32 devices (8 nodes).

In our fourth test, we measured scaling of varying stencil orders in a 256^3-cell simulation (Fig. 14). As expected, low-order stencil computations require less communication and scale more efficiently than higher-order ones. Second-order stencils scale relatively efficiently to 32 devices, whereas 8th-order stencils computations become bandwidth-bound at 8 devices.

In our fifth test, we compared the strong scaling of Astaroth and PC (Fig. 15(a)). To produce fair benchmarks, PC-Puhti was used with grid dimensions 280^3, 520^3, and 1040^3, which were multiples of the per-node core count and close to the dimensions 256^3, 512^3, and 1024^3 used with Astaroth and PC-Mahti. For clarity, we use the terms small, medium, and large to denote the benchmarks using 256^3 or 280^3, 512^3 or 520^3, and 1024^3 or 1040^3 cells, respectively. For decompositions, we chose common-sense processor numbers in the x, y, and z directions, based on the recommendations of the PC developers and its user manual. However, as the decomposition must be supplied by the user and trying each possible choice was not feasible within the bounds of this work, our PC benchmarks may not present the best performance possible.

On a single node, Astaroth exhibited a 18–53x speedup compared to PC. On 16 nodes, the speedups with Astaroth in the small, medium, and large benchmarks were 4–15x, 11–32x, and 20–60x, respectively. The scaling efficiency of PC remained relatively high in all tests.

In our final test (Fig. 15(b)), we measured the energy efficiency of the implementations in terms of cell updates per second per watt. On a single node, we saw 8–11x improved energy efficiency with Astaroth compared to PC, whereas on 16 nodes, the energy efficiency improved 2–3x, 5–6x, and 9–12x in the small, medium, and large benchmarks, respectively.

6. Discussion

The landscape of high-performance computing is evolving: modern compute nodes are heterogeneous, housing multiple specialized accelerators that can provide several times higher throughput in domain-specific tasks. As network bandwidth and latency lag behind operational performance, communication-heavy applications must be carefully tuned to reduce data movement to enable efficient scaling to a large number of nodes.

In this work, we have addressed the main questions regarding the implementation of large-scale ISL-based physical simulations on HPC nodes containing multiple GPU devices. By Z-order-based partitioning, and executing computation and communication in parallel, our implementation scaled to 16 nodes at ≥50% efficiency compared to the theoretically achievable performance.

Compared with traditional CPU computations, GPUs can provide competitive throughput and energy efficiency in multi-node applications despite additional device-to-host and host-to-device communication latency (Figs. 15(a) and 15(b)). When comparing the operational performance and memory bandwidth available on a single HPC node as used in this work, we expect that GPUs can provide an order of magnitude higher throughput in equivalently optimized data-parallel programs when the performance is bound by compute. In large-scale simulations, the benefits of using GPUs diminish when the network bandwidth becomes the performance limiter. Furthermore on a single node using PC-Puhti, we expected a speedup of roughly 13x with Astaroth based on the available memory bandwidth. Instead, we measured a speedup of 44–53x, which suggests that the solvers were not equally optimized.

On current hardware in the test cases presented in this work, the benefits of GPUDirect RDMA in high-order stencil computations are modest, as the majority of the messages are above the 32 KiB threshold (Fig. 12, communication with and without corners). We expect to see more notable benefits from RDMA in larger-scale simulations, where each subgrid contains ≤32^3 cells.

In previous work, we spawned a single process per node, and used CUDA peer-to-peer memory transfers and one-dimensional decomposition for intra-node communication instead of MPI [20]. In this work, we saw no notable performance difference in single-node performance between our previous implementation and our implementation with multiple processes and MPI. The integration time per cell was 0.65 ns in previous work, in contrast to 0.53 ns measured in this work. This was expected, as both implementations were compute bound in our tests. Further tests with smaller grid sizes are required to see whether there is a difference when the performance is bound by bandwidth.

Our work has the following limitations. Firstly, we assumed that the bandwidth between any pair of nodes, or devices within a node, is always the same. This holds if the network topology is a fat tree and the devices within a node are fully connected. However, further analysis may be needed to find the optimal decomposition on more complex interconnect topologies. Furthermore, in our model, we assumed that communication can be carried out simultaneously with all neighbors. This is generally not the case on current hardware and the communication cost function should be extended to take this into account for more precise scaling estimates.

Secondly, our performance model does not account for the communication overhead, or latency, which can take a significant part of the communication time if message sizes are especially small. In this work where large messages were used, we deemed the model to provide sufficiently robust performance estimates.

Thirdly, our implementation is not fully pipelined, which implies lower parallel efficiency in all test cases. This is caused by starting the
update of the outer computational segments only after all communication has finished. Scaling efficiency could be improved with a more fine-grained communication scheme, in which the update of an outer segment is started immediately after its data dependencies have been satisfied [47].

Fourthly, while Z-order indexing provides a good approximation of the optimal decomposition in our test case on an ideal machine, it may not be communication-optimal on practical hardware. Graph-based algorithms for optimizing network communication is a well-studied field [26] and integrating this knowledge into Astaroth is a subject of future work.

As the performance of our implementation was ≥50% the theoretical maximum, more aggressive techniques to reduce data movement are required to see significant improvements in scaling efficiency. One such technique, albeit controversial, is real-time data compression. Lossless compression would reduce data movement without loss of precision, however, achieving satisfactory throughput and compression ratio with noisy data may be difficult. Lossy compression typically provides better compression ratios than lossless compression, but may require tuning in accuracy-sensitive applications. A simple form of lossy compression is mixed precision, supported for example by GROMACS [48], where lower precision is used in calculations where it is known not to result in a catastrophic loss in accuracy. More sophisticated approaches have also been suggested, for example by Lindstrom [49], where four-fold lossy compression in shock hydrodynamics simulations was reported to result in a relative error of 0.06% after 1000 timesteps. There has been notable interest towards hardware-accelerated compression [50,51], as it could provide higher throughput than software implementations. Preliminary studies have also been conducted on using machine learning to reconstruct physically accurate turbulent flows from low-resolution data [52,53].

Whether the disproportional growth of operational, memory, and network performance continues is an open question. At the current rate, the performance of all implementations will eventually become bound by data movement. However, the off-chip memory bottleneck has been notably alleviated, but not eliminated, with the introduction of 3D-stacked CMOS technology [54]. Technologies based on silicon photonics are expected to bring similar improvements to network performance [55]. Moreover, the physical limits for transistor densities in silicon-based microprocessors are expected to be encountered around 2025 [56,57]. As the die size cannot be increased indefinitely to fit more transistors due to lower manufacturing efficiency and issues with heat dissipation [51], new technologies are needed to see continued improvements in operational performance. Several technologies have been suggested, such as graphene-based microprocessors and integrating multiple GPU modules on a single package [58].

Until physical manufacturing limits can be overcome, we expect to see increased use of domain-specific accelerators, such as GPUs, TPUs, and ASICs, due to their ability to provide higher throughput and energy efficiency with existing manufacturing technologies than general-purpose processors. In this light, we believe that data movement will continue to be a major challenge in large-scale computations.

7. Conclusion

Computing centers have started to incorporate specialized accelerators to HPC nodes to improve throughput and power efficiency in domain-specific tasks. This has further increased the gap between computational performance and the rate at which data can be transferred via the network. To enable efficient scaling to multiple nodes in communication-heavy applications, minimizing inter- and intra-node communication is of utmost priority.

In this work, we presented an analysis of the scaling of iterative stencil loops and applied established techniques to implement a scalable multi-GPU communication scheme, which we evaluated in high-order MHD simulations. In our benchmarks, we saw that the per-node performance improvement from GPUs outweighs the added device-to-host and host-to-device communication latencies, and that strong scaling to at least 64 GPUs is possible at high efficiency with sufficiently large problem sizes. Because inter-node bandwidth is an expensive resource, improving intra-node data locality and reducing overall data movement, even at the cost of redundant computations, is likely required to see more efficient scaling in communication-bound problems.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.parco.2022.102904. All funding has been received from academic non-commercial sources.

Acknowledgments

This study is a product of Astaroth collaboration. The authors thank Fredrik Robertsén (CSC) for valuable discussion and technical support. J.P., M.J.K., and M.R. acknowledge the support of the Academy of Finland ReSoLVE Centre of Excellence (grant number 307411). This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (Project UniSDyn, grant agreement no 818665). M.V. acknowledges funding support from CHARMS within ASIaA from Academia Sinica, Taiwan.
Appendix A. Decomposition

Solutions to Eq. (5) for typical problem sizes, in which the stencil contains all points within its radius and the boundaries are periodic, are shown in Tables A.1 and A.2. We evaluated each valid decomposition using a brute-force search. Only one decomposition is listed if there is more than one solution. The optimal inter-node level decompositions \( P \) are shown in Table A.1. Intra-node level decompositions \( P' \) are shown in Table A.2.

Appendix B. MHD equations

The basic physical quantities updated during each RK3 timestep are shown in Table B.1. We use the standard MHD equations for an ideal gas in non-conservative form:

\[
\frac{D\ln \rho}{Dt} = -\nabla \cdot \mathbf{u} \quad \text{(B.1)}
\]

\[
\frac{D\mathbf{u}}{Dt} = -c_s^2 \nabla \left( \frac{1}{\rho} \nabla \ln \rho \right) + \frac{\mathbf{B}}{\rho} + \nabla \left( \mathbf{u} \cdot \nabla \mathbf{u} \right) + \frac{1}{2} \nabla \left( \mathbf{u} \cdot \mathbf{u} \right) \quad \text{(B.2)}
\]

\[
\rho \frac{D\mathbf{A}}{Dt} = \mathbf{F} - \mathbf{e} + \nabla \cdot (K VT) + \eta \mu_0 \mathbf{B}^2 + \frac{2}{3} \rho \mathbf{S} \otimes \mathbf{S} + \zeta \rho (\mathbf{u} \cdot \mathbf{u})^2 \quad \text{(B.3)}
\]

\[
\frac{\partial \rho \mathbf{A}}{\partial t} = \mathbf{u} \times \mathbf{B} + \eta \nabla^2 \mathbf{A} \quad \text{(B.4)}
\]

See Table B.2 for symbol explanations. We refer the reader to [6] for a more detailed discussion on the physical system and related computational aspects.

Table A.1

<table>
<thead>
<tr>
<th>( N )</th>
<th>( P )</th>
<th>( P_0 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ((512, 512, 512))</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(b) ((1024, 512, 512))</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(c) ((1024, 1024, 512))</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Table A.2

<table>
<thead>
<tr>
<th>( N' )</th>
<th>( P' )</th>
<th>( P'_0 )</th>
<th>( P'_1 )</th>
<th>( P'_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ((512, 512, 512))</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(b) ((1024, 512, 512))</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>(c) ((1024, 1024, 512))</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Table B.1

<table>
<thead>
<tr>
<th>Field</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logarithmic density</td>
<td>( \ln \rho )</td>
</tr>
<tr>
<td>Velocity</td>
<td>( \mathbf{u} )</td>
</tr>
<tr>
<td>Specific entropy</td>
<td>( s )</td>
</tr>
<tr>
<td>Magnetic vector potential</td>
<td>( \mathbf{A} )</td>
</tr>
</tbody>
</table>

Table B.2

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laplace operator</td>
<td>( \nabla^2 )</td>
</tr>
<tr>
<td>Curl operator</td>
<td>( \nabla \times )</td>
</tr>
<tr>
<td>Adveective derivative</td>
<td>( D / Dt = \partial / \partial t + \mathbf{u} \cdot \nabla )</td>
</tr>
<tr>
<td>Magnetic field</td>
<td>( \mathbf{B} = \nabla \times \mathbf{A} )</td>
</tr>
<tr>
<td>Magnetic diffusivity</td>
<td>( \eta )</td>
</tr>
<tr>
<td>Magnetic vacuum permeability</td>
<td>( \mu_0 )</td>
</tr>
<tr>
<td>Electric current density</td>
<td>( j = \mu_0^{-1} \nabla \times \mathbf{B} )</td>
</tr>
<tr>
<td>Traceless rate-of-shear tensor</td>
<td>( \mathbf{S} )</td>
</tr>
<tr>
<td>Specific heat capacity at constant pressure</td>
<td>( c_p )</td>
</tr>
<tr>
<td>Specific heat capacity at constant volume</td>
<td>( c_v )</td>
</tr>
<tr>
<td>Kinematic viscosity</td>
<td>( \nu )</td>
</tr>
<tr>
<td>Bulk viscosity</td>
<td>( \zeta )</td>
</tr>
<tr>
<td>Adiabatic speed of sound</td>
<td>( c_s )</td>
</tr>
<tr>
<td>Adiabatic index</td>
<td>( \gamma )</td>
</tr>
<tr>
<td>Explicit heating term</td>
<td>( H )</td>
</tr>
<tr>
<td>Explicit cooling term</td>
<td>( C )</td>
</tr>
<tr>
<td>Radiative thermal conductivity</td>
<td>( K )</td>
</tr>
<tr>
<td>Temperature</td>
<td>( T )</td>
</tr>
</tbody>
</table>
J. Pekkilä et al.

Miikka S. Väisälä is a Postdoctoral Fellow at Academia Sinica Institute of Astronomy and Astrophysics in Taiwan. He did his Ph.D. in Astronomy of the topic of magnetic field in interstellar medium, and how to use GPU-methods to enhance the related computations; graduating from the University of Helsinki in 2017. He started the development of Astaroth as a part of his PhD work. His research interests include computational magneto-hydrodynamics of interstellar medium and star formation, radiative transfer to connect such data to observations and scientific data visualization.

Oskar Lappi is a PhD student at the University of Helsinki doctoral school of computer science. Oskar is a software engineer turned HPC researcher who has developed an asynchronous Task Graph scheduling system for Astaroth. In addition to Astaroth, Oskar is currently working on other HPC projects in his role as a performance researcher at the EUROFusion Advanced Computational Hub in Helsinki. His research interests include: performance modeling, computer and software architecture, distributed systems and programming language design.

J. Pekkilä is a doctoral candidate at Aalto University School of Science. He majored in big data and large-scale computing and has worked on accelerating various physical simulations with GPUs since 2014, from fluid dynamics to particle-particle interactions. His current research efforts are focused on accelerating fundamental computational patterns on massively parallel systems. His research interests include parallel and quantum computing, hardware architecture, programming languages, computer graphics, and machine learning.