Improving Communication by Optimizing On-Node Data Movement with Data Layout

Tuowen Zhao
Mary Hall
ztuowen@cs.utah.edu
mhall@cs.utah.edu
School of Computing
University of Utah
Salt Lake City, Utah, USA

Hans Johansen
Samuel Williams
hjohansen@lbl.gov
swwilliams@lbl.gov
Computational Research Division
Lawrence Berkeley National Lab
Berkeley, California, USA

Abstract

We present optimizations to improve communication performance by reducing on-node data movement for a class of distributed memory applications. The primary concept is to eliminate the data movement associated with packing and unpacking subsets of the data during communication. With the rapid rise in network injection bandwidth reducing off-node data movement cost, on-node data movement can be significantly more expensive than computation and network communication. This data movement is especially costly for small domains - as in memory-intensive multi-physics codes or when strong scaling to reduce time-to-solution. The optimizations presented include (1) optimizing data layout through indirection to enable pack-free communication; (2) creating contiguous views of memory using memory mapping thus minimizing the number of messages; and (3) applying these techniques to intra-node data movement including CPU-GPU data movement. The benefits of these optimizations are demonstrated in stencil benchmarks against a highly-optimized baseline, reducing communication time by up to 14.4×.

CCS Concepts: • **Computing methodologies** → **Distributed algorithms**; • **Software and its engineering** → *Software libraries and repositories.*

Keywords: MPI Communication, Data Layout, Data Blocking

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441598

1 Introduction

Many parallel distributed algorithms, such as finite difference/element methods and Krylov sub-space methods, exhibit a common communication pattern involving nearest-neighbor data. This communication pattern has the following properties: (1) *Static:* the addresses and size of the data to be communicated are static across iterations of a simulation; (2) *Subsets:* communication involves a subset of data, some of which is not adjacent in memory; and, (3) *Overlapping:* the subsets to be communicated are overlapping, so that the same data is sent to multiple neighbors.

As an example, *stencil computations* exhibit this communication pattern; stencils are ubiquitous in scientific simulations that solve partial differential equations. To calculate stencils on logically regular grids in a distributed environment using MPI, the regular domain may be divided into subdomains that are distributed among MPI processes. These subdomains are then extended with a *ghost zone* that contains copies of the values from neighboring subdomains that are needed each time a stencil is applied. After each stencil application on a subdomain, called the *timestep*, the values from the *surface* of the subdomain are used to populate the ghost zones of neighboring subdomains, by sending the data from the local surface to the corresponding remote ghost zone, using MPI. Often, surface and ghost zones are fixed across multiple timesteps, and a given surface region is sent in multiple messages to different neighbors. This required communication to update ghost zone values is called a *ghost zone exchange*. As some of the surface regions may not be contiguous in memory, a common practice to reduce the number of messages is to *pack* data from the surface into MPI messages and *unpack* from the MPI messages to the ghost zone. (For simplicity, we will describe both collectively as *Packing* in this paper.) *Packing* introduces significant on-node data movement, which increases overall communication costs.

To illustrate this overhead, consider performance of a canonical 7-point stencil. Figure 1 shows results using the state-of-the-art YASK stencil code generator [25] running on 8 Intel KNL nodes of the Theta supercomputer (see end of Section 2, below). We hold the number of nodes constant and vary the subdomain size. This test is a proxy for strong

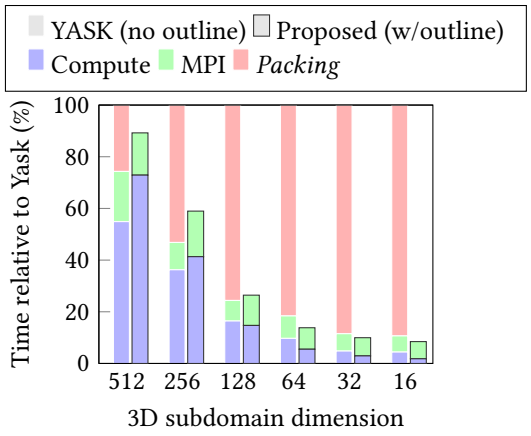


Figure 1. Time breakdown per timestep on 8 Theta nodes (KNL 7230), where each node is given one subdomain size $(N = 512)^3$ down to $(N = 16)^3$, laid out in a periodic 3D cube $(2 \times N)^3$. For comparison, YASK [25] (left bars) uses an autotuned 7-point stencil, without overlapping MPI communication (green) and computation (blue) using the `-no-overlap_comms` option. For all but the largest subdomain sizes, a majority of the time is in *Packing* (red), which is a type of on-node data movement that our approaches entirely avoid.

scaling, where the overall problem size is fixed as the number of nodes increases. As we shrink the subdomain dimension by 2×, the resulting subdomain is comparable to scaling the number of nodes by 8× for a fixed size domain. This also corresponds to minimizing time-to-solution and/or memory footprint, as the case in many multi-physics or iterative solver applications. When shrinking the subdomains, the subdomain size per node (volume of operations) shrinks faster than the surface (volume of data communicated), resulting in an increased fraction of time being spent on communication. In fact, Figure 1 indicates that on smaller subdomains, the time spent on *Packing*, as a step of communication, becomes the majority of the ghost zone exchange time, while the time spent in computation decreases rapidly. The total communication time, which includes MPI and on-node data movement such as *Packing*, exceeds the computation time, even for the larger 256^3 subdomain sizes.

Packing is expensive because ghost zones vary in size, and may express unit-stride, stanza, or strided memory access patterns. These patterns fight against the hardware trends in SIMD and GPU-accelerated computing, resulting in expensive memory operations. This data movement cost applies to scientific applications beyond the domain of stencils that also require *Packing* to communicate non-contiguous data [14, 19].

In this paper, we introduce optimizations to eliminate *Packing* to achieve a *pack-free ghost zone exchange*. As the subdomain size gets smaller, Figure 1 shows that our approach becomes much faster than YASK, with as much as a 14.4× speedup. Previous strategies for optimizing ghost zone exchange have not attempted to eliminate on-node data movement and instead focused on hiding or reducing the costs during communication. These include: (1) overlapping communication and computation, possibly combined with time skewing, to hide latencies in the network and increase parallelism [18, 23, 25]; (2) communication-avoiding optimizations to reduce message frequency to amortize the cost of communication [3, 5, 7, 15, 21]; and, (3) using MPI derived types to describe strided ghost zone regions and the MPI library to optimize packing performance [2, 4, 8].

We achieved *pack-free ghost zone exchange* based on logical-to-physical indirection, so that underlying data ordering can be rearranged while still preserving logical computation, i.e., *layout optimization*. For heterogeneous systems using GPUs, our methods also reduce data movement cost during communication as a whole. Currently, data must be moved between host CPU and GPU during the ghost zone exchange. If *Packing* occurs on the CPU, the entire subdomain must be moved between CPU and GPU; otherwise *Packing* occurs on the GPU and only the packed buffers are moved. The application programmer then needs to write corresponding optimized *Packing* routines and shuttle data explicitly. In prior work [29], MPI was found to take up only half of the communication time, with *Packing* and shuttling data on node taking up the rest. New techniques such as CUDA-Aware MPI[1] and Unified Memory facilitate CPU-GPU data movement by allowing direct MPI communication on memory accessed by the GPU. We use pack-free ghost zone exchange in conjunction with these techniques to eliminate manual host and GPU data movement, resulting in substantial performance improvements.

The contributions of this paper are as follows: (1) we eliminate data movement from *Packing* using `Layout` optimization, to achieve pack-free communication for 3D stencil computations; (2) we combine this with virtual memory mapping, `MemMap`, to further improve communication performance and optimize data movement between CPU and GPU, and between subdomains on the same rank; (3) we implemented these communication methods as library functions that extend the brick library [27] to multiple nodes; and (4) we provide a detailed analysis of pack-free communication performance on both CPU and GPU platforms through scaling experiments, where we achieve up to 14.4× faster communication.

2 Motivation and Overview

Ghost zone exchange for stencil computations is a prominent communication pattern with the three properties (*Static*, *Subsets* and *Overlapping*) targeted by our optimizations. To demonstrate this, Figure 2(L) shows the ghost zone exchange

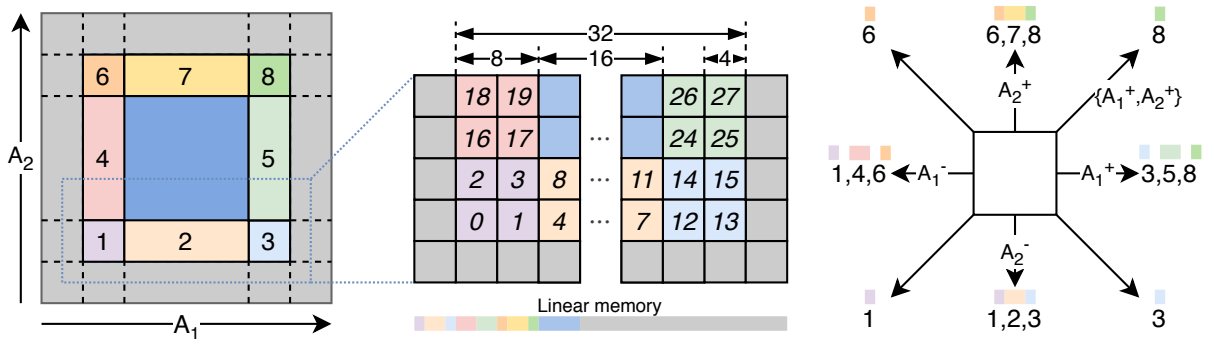


Figure 2. (**L**) An example 32×32 subdomain with an 8-cell-wide ghost zone is decomposed into surface regions ($1-8$, colored separately) and ghost zone (grey) regions. (**C**) shows the dotted logical area decomposed into 4×4 data blocks. Blocks are indexed (*0, 1,* ...) so that physically each surface region is contiguous with regions laid out in sequence according to their numbers (linear color bar, bottom). (**R**) Each surface region is sent to one or more neighbors. Using this layout, 12 messages are required because the side messages are not contiguous in memory (indicated by spaces), but an optimal layout requires only 9 messages (see Section 3.2).

regions for a 2D stencil kernel. The inner blue and colored regions represent the logical subdomain assigned to a particular process, while the grey outer rim represents the ghost zone of the subdomain. The rectangles numbered 1–8 represent the different-sized surface regions that will be sent to neighboring subdomains, potentially on other nodes. For example, region 4 is sent to only the left neighbor, while region 1 is sent to the left neighbor and two others. Thus, they are disjoint surface regions.

Although a *lexicographical* array layout has one contiguous axis, e.g. *i* in *i-j-k*, none of these surface or ghost zone regions is a contiguous array in that case, and so communication requires *Packing*. These *Packing* operations result from the need for a different *physical ordering* of the data than the *logical ordering* used for computation. Instead, we eliminate the need for *Packing* using layout optimization, so that the physical ordering of the data is optimized for communication, while preserving the logical ordering used for computation.

The optimized layout is realized using fine-grained data blocking. The original structured data is broken into fixed-size data blocks, such as the 4×4 blocks in Figure 2(C). The data in each block is stored contiguously, and the block is assigned an index that reflects the physical order that blocks are stored in memory. The logical order is represented as a graph-like data structure consisting of all the blocks. To clarify this point, consider how unstructured data is typically represented in a graph: data associated with each vertex is stored in an array based on its index. Layout optimization on the unstructured data is achieved by simply reordering the graph storage and its corresponding indices, while still preserving its logical organization. Similarly, layout optimization of structured data adds a level of indirection on top of the fine-grained data blocking.

Applying fine-grained data blocking to ghost zones requires ghost zones that are multiples of the block size. Several factors govern the size of ghost zones. A computation may apply multiple stencils in a sequence, and the ghost zone for each stencil can share a data block. Also, stencil applications increasingly use high-order stencils [17, 26] where the number of inputs to compute an output is large. In either of these cases, a single timestep will require a wide ghost zone that can be decomposed into these fine-grained data blocks. For applications that use a low-order stencil, such as a 2D 5-point stencil, even a 4×4 data block is larger than the 1-cell-wide ghost zones. For such stencils, we can expand the ghost zone to a multiple of the block size (such as $8 = 2 \times 4$ in the example), and use communication avoiding *ghost cell expansion* [7]. With ghost cell expansion, we exchange all of the elements in the expanded ghost zone, thus exchanging roughly 8× the communication volume. Ghost cell expansion improves performance by reducing communication frequency (by a factor of 8), trading off redundant computation to use all the data exchanged. Using ghost cell expansion, even this low-order stencil has a ghost zone large enough for fine-grained data blocking.

During computation, these fine-grained data blocks represent aggregate units of parallel work where vectorization can be easily implemented. The logical organization identifies neighboring blocks that contribute values to the computation. Examples of fine-grained data blocking and associated code generators for stencil computations include briquettes [11] and bricks [27, 28]. These showed that fine-grained data blocking provides performance portability for stencil computations by improving vertical data movement, and reducing cache and TLB pressure.

This indirection approach is shown in Figure 2(C), where each region of the surface or the ghost zone is stored contiguously to avoid *Packing* within them. The data blocks are indexed (*0*, *1*, ...) so that the indices are contiguous within each surface region and the indices from different regions follow the numbering of the regions in Figure 2(L). The indices are the physical ordering in memory, which is illustrated with a colored bar at the bottom of Figure 2(C).

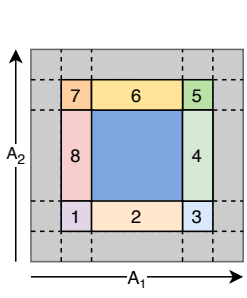
As illustrated in Figure 2(R), subsets of data that participate during the communication can have more than one destination, and so must appear in more than one message. For example, surface region 1 is sent to three different neighbors in 2D. Without *Packing*, we can send each instance of the region independently and increase the number of messages without increasing the volume of communication. We can also leverage the storage order across regions, such as 1-2-3, to reduce the message count, as discussed in Section 3, below. This is dimension-dependent; we will show in Section 3.3 that layout optimization is more effective for lower-dimension problems. In addition, Section 4 includes an additional optimization using virtual memory mapping which allows multiple regions to be mapped to the same section of data.

We have developed these two ideas targeting stencils on structured data. This is because, while structured data needs fine-grained data blocking to introduce indirection, it is also the most representative and easiest to analyze due to fixed dimensionality. In our experiments we use 3D stencils as the prototypical application, however the same analysis in this paper can be applied to many other applications with similar repeated communication patterns.

The remainder of this paper is organized as follows. First, in Section 3 we introduce the pack-free ghost zone exchange via layout optimization for stencil computations using fine-grained data blocking. In Section 4, we show how memory mapping extends its applicability and potential to improve the performance of pack-free ghost zone exchange. Section 5 discusses how both layout optimization and memory mapping apply to CPU-GPU communication.

Our results and analysis are from experiments on two large-scale supercomputers:

• The *Theta* supercomputer uses Cray XC40 nodes, with a single Intel Xeon Phi Knights Landing (KNL) 7230 on each node. The processor has 64 physical cores, each with 4-way multithreading and two AVX-512 vector processing units (VPUs). The processor can operate at a turbo frequency of 1.50 GHz that may downclock to around 1.1 GHz with AVX-512-intensive applications. This results in an effective sustained double-precision performance of 2.2 TFlop/s. The processor also has 16 GB of MCDRAM which is used as separate addressable memory in "flat" mode that has a STREAM [13] performance of 467 GB/s. The network uses the Cray Aries interconnect with a Dragonfly topology. MPI used is Cray-MPICH version 7.7.10.



(a) Illustration

$r(\{A_1^-, A_2^-\}), r(\{A_2^-\}), r(\{A_1^+, A_2^-\}),$
$r(\{A_2^+\}), r(\{A_1^+, A_2^+\}), r(\{A_2^+\}),$
$r(\{A_1^-, A_2^+\}), r(\{A_1^-\})$

(b) Notational representation

```
1 std::vector<BitSet>
2 surface2d={
3   {-1,-2},{-2},{1,-2},{2},
4   {1,2},{2},{-1,2},{-1}
5 };
```

(c) Represented in code

Figure 3. Optimized 2D surface layout.

• The *Summit* supercomputer has IBM AC922 nodes. 6 NVIDIA Volta V100 GPUs are connected with each other through NVLink, with two IBM Power9 CPUs acting as the host processor. Computations are performed on the V100 GPUs. The V100 has 16 GB HBM2 memory with a bandwidth of 828.8 GB/s [24]. Each GPU has 5120 CUDA cores that can operate at up to 1.5 GHz producing a theoretical peak double-precision performance of 7.8 TFlop/s. The network is connected with Mellanox EDR 100G InfiniBand in a fat-tree configuration. MPI used is Spectrum-MPI 10.3.1.2.

3 Optimizing Layout for Communication

We can leverage indirection in the data representation to optimize the layout for communication, which we have termed `Layout` optimization; this section describes how optimized layouts are encoded in the library implementation. Second, we present an analysis of messages required for ghost zone exchange using `Layout` for structured data. Finally, we discuss the effectiveness of `Layout`.

3.1 Describing Layout for Communication

As each surface region has the same set of destination neighbors, they should be stored together so that they can be sent together as a unit. Figure 2(C) illustrates this, with regions of the surface stored contiguously through reordering the data blocks, which applies to ghost zone regions as well.

We use a system of notations to encode the regions of the data and the neighboring nodes. D-dimensional data has d axes, $A_1, A_2, \ldots, A_D$. Each axis, A_i, has a positive A_i^+ (up or right for 2D) and a negative A_i^- (down or left for 2D) direction. A set of axes and a direction can identify a neighbor and surface region in those directions. For example, the the northeast neighbor in Figure 2(C) is identified as neighbor $N(\{A_1^+, A_2^+\})$, and surface region labeled 8 is identified as region $r(\{A_1^+, A_2^+\})$.

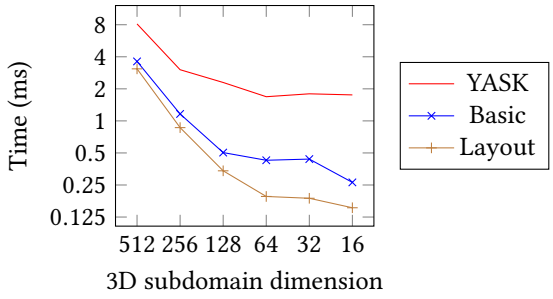


Figure 4. Time required for communication during one 3D stencil loop on 8 KNL nodes vs. subdomain size. YASK includes time taken for *Packing*, which our approach avoids entirely (Section 2). Sending each region independently requires 98 messages (`Basic`), while our optimized layout (`Layout`) requires only 42 messages which speeds up communication for smaller subdomains.

This notation system is also used in the library implementation. Figure 3 shows the notation and corresponding code of an optimized 2D layout.

3.2 Layout Optimization

We can improve communication performance by sending consecutive surface regions in the same message. We note that ghost zone regions are disjoint and owned by one process, and so do not overlap when receiving data from another process. In contrast, surface regions do overlap when being sent to other processes. For example, if the surface regions are stored according to the numbering in Figure 2(L), regions `1-3` are contiguous in memory and can be sent with exactly one message to neighbor $N(\{A_2^-\})$. Using this layout requires 12 messages for ghost zone exchange with 8 neighbors.

By permuting the surface regions and leveraging the relative ordering of the regions, we can reduce the total number of messages. For example, Figure 3 permuted the 2D regions in Figure 2, so that its regions `3-5` can be sent with one message to neighbor $N\{A_1^+\})$. Thus, only 9 messages are required for 8 neighbors.

The possible layouts result from permuting the surface regions. These layouts can be compared using the resulting number of messages required for ghost zone exchange. Using this criterion, we can find a 3D layout that only uses 42 messages to communicate with 26 neighbors through `Layout` optimization. This can be compared with the `Basic` approach, which sends each region individually. This `Basic` approach does not consider the relative order of the regions and needs no layout optimization. As a result, `Basic` needs 98 messages instead. The timing results in Figure 4 show that `Layout` is up to 2.3× faster than `Basic`.

Table 1. Impact of the number of dimensions on the number of messages. `Layout` optimization is more effective for dimensions lower than 5.

Dimensions	1	2	3	4	5
Number of neighbors (Eq. 2)	2	8	26	80	242
`Layout` (Eq. 1)	2	9	42	209	1042
`Basic` (Eq. 3)	2	16	98	544	2882

3.3 Asymptotic Analysis

We have derived a lower bound on the number of messages required with `Layout` optimization to send values from a D-dimensional domain's surface to all its neighbors, including diagonals, as shown in Eq. 1.

$$\frac{5^D}{3} + \frac{(-1)^D}{6} + \frac{1}{2} \qquad (D \geq 1) \tag{1}$$

For brevity, the detailed proof is provided as supplemental material to this paper.

Packing creates one message for each neighbor requiring data from this subdomain. The total number of messages is shown in Eq. 2, which equals the number of neighbors.

$$3^D - 1 \tag{2}$$

An upper bound on the number of messages with fine-grained data blocking is with the `Basic` approach. As long as each region is stored contiguously in memory, this approach will result in the number of messages shown in Eq. 3.

$$5^D - 3^D \tag{3}$$

Table 1 compares these three alternatives for different numbers of dimensions. When comparing against `Basic`, asymptotically, layout optimization could at most reduce messages by 2/3. If we consider the number of neighbors, Eq. 1 also implies that the number of messages grows exponentially with respect to the number of neighbors. This growth rate makes layout optimization most effective when dimension is less than 5.

Eq. 1 also proves that the 3D layout from the previous section that results in 42 messages is an optimal layout when optimizing for the number of messages. Using this optimal 3D layout, we are able to trade the expensive *Packing* operation with 16 extra messages. This optimal 3D layout is provided as a constant, `surface3d`, in our library, and described analogously to `surface2d` in Figure 3.

4 Memory Mapping for Communication

We can reduce the number of messages for ghost zone exchange and send just a single message to each neighbor through the use of virtual memory mapping, which we will call as `MemMap`. Virtual memory mapping provides an extra layer of indirection from virtual to physical address. This

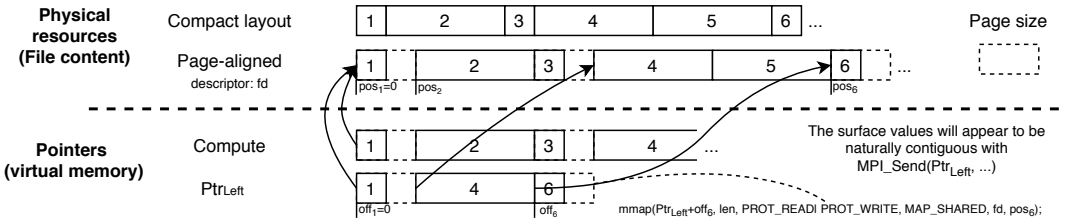


Figure 5. Using memory mapping (*MemMap*) we can avoid both *Packing* and sending extra messages. Whereas sending the 1, 4, 6 surface regions in Figure 2 would nominally require 3 messages, memory mapping exploits virtual memory and requires only one message.

level of indirection is in addition to layout optimization, which only modifies the logical organization of the data within the virtual address space. Additional virtual memory manipulation results in little added overhead as virtual addresses are resolved implicitly through a combination of hardware units and kernel mechanisms.

To clarify, files in Linux can represent a chunk of physical memory. Virtual memory mapping to these files creates a link from virtual memory to physical memory. These files, once opened, are referred to by file descriptors in a program. Using these file descriptors, the `mmap` function allows manual mappings of segments of files into an application's virtual memory space. Consecutive mappings can map to different file segments, while the same file segments can also be mapped multiple times to different memory regions. Thus the underlying data can appear to be reordered when following different pointers, allowing us to create different orderings of the data for different communication and computation needs. As seen in Fig 2(L), this enables some regions of the surface to be sent to multiple neighbors while having only one copy during computation. To ensure the use of main memory, these files are created through `memfd_create` or `shm_open`. To create corresponding views, the `mmap` function option `MAP_SHARED` is used to carry all changes in the mapped virtual memory regions to the underlying files, thus making all changes visible across different threads and processes.

For each neighbor that needs data from a given subdomain, we create a view of that data in which regions are mapped consecutively. Figure 5 shows how regions 1, 4, and 6 are mapped to a view of data that is sent to the left neighbor in Fig 2(R). These views can be reused throughout the application until the communication pattern changes. Most significantly, these views do not incur on-node data movement. This is because when MPI is called with these pointers, the virtual memory system will automatically feed the correct set of regions as if they were contiguous in memory. Therefore, `MemMap` achieves pack-free communication and also minimizes the number of messages.

There are two potential concerns when using memory mapping. First, all parts have to be aligned to page boundaries, regardless of size. When a surface region is smaller than a page, it needs to be padded to occupy the whole page. For example, a 4^3 surface region of doubles might waste $\frac{7}{8}$ of a 4KiB[1] page. This also wastes network bandwidth each time the surface region is transmitted. However, while the baseline amount of network bandwidth is a function of subdomain size, the amount of waste is not. Another limitation is the maximum number of mapped regions one process can have, with a default value of 65530 that typically requires super user privileges to change. To alleviate this problem, the number of mappings can be minimized using the layout optimizations in Section 3.

5 Data Movement for NVIDIA GPUs

On-node data movement for NVIDIA GPUs not only consists of *Packing*, but also data movement between CPU and GPU for MPI communication. Managing this CPU-GPU data movement is crucial for achieving good scaling performance. Layout optimization separates on-node data into interior, surface, and ghost zones. This avoids *Packing* and allows the ghost zone and surface data to be moved independently from the interior subdomain data, which enables independent setup of MPI calls. This allows both `Layout` and `MemMap` to use new technologies like CUDA-Aware MPI (CA) [1] with GPUDirect and Unified Memory (UM) with System Allocator. As a result, we are able to also avoid staging data on the CPU, which can add significant communication latency.

Also, with CUDA-Aware MPI and GPUDirect, communication can use remote direct memory access (RDMA) to a device memory region created by `cudaMalloc` and avoid staging any data through the CPU. `Layout` can be used with these MPI implementations to entirely avoid temporary staging of data, although currently device memory allocated through `cudaMalloc` does not support `MemMap`[2].

On supported systems, Unified Memory with System Allocator allows memory mapping for GPU applications by

[1]Unit symbol under International Electrotechnical Commission: 1 Kibibyte (KiB) = 1024 Bytes

[2]CUDA release 10.2 onward provides `cuMemMap` which may permit memory mapping using device memory. However, currently this is not supported on Summit.

granting GPU access to the page tables and access mappings created by the CPU. We can then setup memory using `MemMap` as discussed in Section 4. A page fault mechanism is used on the GPU when data from a page is not already on the GPU. Similarly, the CPU is able to use page faults to automatically move data from the GPU. While a page fault handler is processing in the background, other non-blocked threads can proceed with other operations. Even when MPI data must be staged through the CPU, the page fault mechanisms allow much better overlapping of CPU-GPU data movement with computation and network operations. For Power9 systems (Summit), Unified Memory with System Allocator is accessed with Address Translation Service (ATS); for x86 systems, this may be supported with Heterogeneous Memory Management (HMM) in the future.

6 Library Implementation

We have implemented `Layout` optimization and `MemMap` by extending our prior work, the `Brick` library [27, 28], to support multiple nodes and ghost zone exchange. The brick library provides a data layout that implements fine-grained data blocking, where each data block is called a brick. Bricks are stored consecutively based on an index in `BrickStorage`, and the logical order of bricks is implemented using an adjacency list. This logical ordering is stored in `BrickInfo`. The library implements an interface that combines the storage and logical layout data into `Brick`. Each brick is an n-dimensional array of fixed size that can be accessed by its index using square brackets. Accesses outside of the current brick will be automatically translated to access the appropriate neighboring brick. Stencil computations can then be expressed as computing on all elements within each brick from bricks specified using a list of brick indices. The brick library API interfaces to create `Brick` and compute a 7-point stencil are shown in Figure 6.

The `Brick` library includes a code generator to optimize for stencil applications. The code generator produces stencil code that is highly optimized for vectorization within a brick, and achieves significant data reuse using associative reordering. It employs vector align operations to realize accessing vectors across brick boundaries. The code generation strategy allows breaking down of complex stencils into simpler ones and enables reuse across stencil iterations. This code generation can target both CPUs using various vector instruction sets and NVIDIA GPUs with CUDA.

The `Brick` library realizes the use of an individual brick as both a unit of data and a unit of parallel work; the implementation makes it convenient to operate a subset of bricks that execute on a thread or node. In [27], code produced by the `Brick` library for three different architectures demonstrated significant performance improvements for complex high-order and multi-stencil computations as compared to a highly-optimized tiled version using the same code generation strategy. This difference as compared to tiling results from a significant reduction in the cost of vertical data movement through the memory hierarchy. Vertical data movement is reduced because a 3D brick is stored in a single contiguous address stream, while a 3D tile may touch multiple cache lines and possibly multiple pages.

The work in this paper extends the definition of `BrickInfo` and allocation of `BrickStorage` to incorporate a layout selection and introduces new APIs for the MPI communication. For applications already using the library, the computation specification is unmodified, same as Figure 6. Figure 7 shows the new library function calls that enable `Layout` and `MemMap`

```
Brick<Dim<8,8,8>,Dim<4>> a(&bInfo, bStorage, 0);
Brick<Dim<8,8,8>,Dim<4>> b(&bInfo, bStorage, 512);
```

(a) Defining `Brick` to access storage.

```
for (auto brickIndex: listofBrickIndices)
for (int k = 0; k < 8; ++k)
for (int j = 0; j < 8; ++j)
for (int i = 0; i < 8; ++i)
a[brickIndex][k][j][i] =
   c0 * b[brickIndex][k][j][i]
 + c1 * b[brickIndex][k-1][j][i]
 + c2 * b[brickIndex][k+1][j][i]
 + c3 * b[brickIndex][k][j-1][i]
 + c4 * b[brickIndex][k][j+1][i]
 + c5 * b[brickIndex][k][j][i-1]
 + c6 * b[brickIndex][k][j][i+1];
```

(b) `Brick` has one extra index dimension to denote which brick it is currently accessing. The library automatically resolves indices to the correct memory location in the current brick or another brick.

Figure 6. Basic `Brick` interface to compute a 7-point stencil. Brick computation is layout-agnostic as long as proper brick indices is supplied.

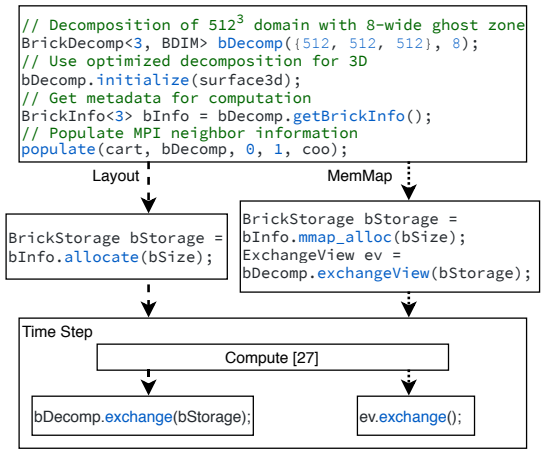


Figure 7. Communication interface in the `Brick` library.

optimizations. The communication methods apply generally to any specified data layout (invoking functions on the left hand side of the figure) or can use memory mapping (functions on the right hand side) to create views of the data for communication. For the 3D experiments in this paper, we specify an optimized `surface3d` layout, analogous to the 2D optimized version in Figure 3. For NVIDIA GPUs, we provided interfaces for `Layout` (using either CUDA-Aware MPI or Unified Memory) and `MemMap` (using Unified Memory).

The brick library also enables interleaving values from multiple fields in one `BrickStorage`, allowing multiple fields to be organized as an array-of-structure-of-array, in Figure 6a. Interleaving multiple fields this way enables communicating them all at once in a single `BrickStorage` exchange.

7 Experiments

To demonstrate the performance of our proposed method, we have performed scaling experiments on two supercomputers: Theta with Intel Xeon Phi Knight Landing, indicated with (*K#*), and Summit with Power9 plus NVIDIA Volta V100, (*V#*). Details for these platforms are discussed in Section 2.

We use two double-precision stencils as proxies for two different extremes of Arithmetic Intensity (AI) [22]. One is a star-shaped 7-point stencil that has an AI of 8/16 (flop/byte). The other is a 5^3 cube-shaped 125-point stencil, which has 10 constant coefficients (due to symmetries) and an AI of 139/16 (flop/byte). Unless otherwise noted, all experiments use $8 \times 8 \times 8$ data blocking ("8^3" bricks) and all ghost zones have a width of 8 using ghost cell expansion [7].

We evaluate four different implementations:

- `YASK` is an optimized stencil framework for CPUs [25]. It has a built-in autotuner and supports overlapping communication and computation.
- `MPI_Types` from MPI supports *Packing* internally within MPI, avoiding explicitly *Packing* by the application programmer and incorporating optimizations not available to the application programmer [8, 12].
- `Layout` optimization, described in Section 3, uses indirection to eliminate data movement from *Packing*.
- `MemMap`, described in Section 4, uses virtual memory mapping to eliminate redundant messages. In these experiments, `Layout` optimization is used to reduce the number of mappings. However, `MemMap` does not depend on using an optimized layout.

Time spent in the main stencil loop (timestep) is decomposed into communication and computation. Computation, `Comp`, includes the time taken to apply the stencil for all points owned by a specific MPI rank, as well as any redundant computation necessary for communication avoiding. Communication, `Comm`, includes the time taken to (a) copy data into MPI buffers, (b) call MPI, and (c) wait for communication to finish. Both of our proposed methods `Layout` and `MemMap` avoid (a). The time reported is the per-timestep

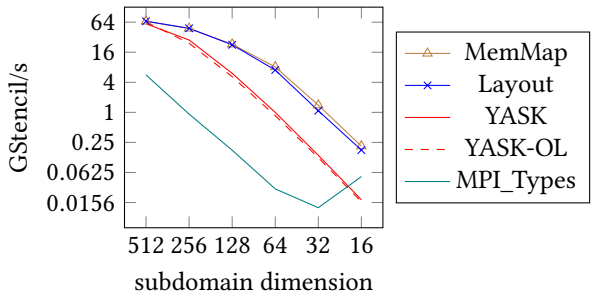


Figure 8. (*K1*) 7-point stencil scaling on 8 KNL nodes. Note that a 2× reduction in subdomain size corresponds with an 8× decrease in total subdomain points.`Layout` is competitive with `MemMap`, and both attain the best performance by minimizing on-node data movement. Overlapping communication with computation with YASK (`YASK-OL`) makes little difference for smaller subdomains (red lines).

average across all nodes, repeated for a sufficient number of timesteps so that the results have minimal variation across multiple runs.

The empirically minimum time required for communication can be obtained by only measuring the time taken to communicate message-sized buffers. This time is represented by `Network` in the experiments.

For experiments *K1* and *V1*, we used 8 nodes and 1 MPI rank per node to form a periodic 3D cube, 2^3. With this configuration, the minimum 8 nodes is required to have each MPI rank neighboring another MPI rank on another node, in all directions. This reduces the network variability and allows us to examine on-node effects, since the ranks form a cubical grid with an even decomposition of the domain with reduced variability in computation.

7.1 Theta with Intel KNL

(K1) 8-node/rank scaling. In this first experiment (*K1*), we analyze performance by reducing the overall problem domain size while using 8 nodes with 1 rank per node. Figure 8 shows the performance scaling relative to the size of the subdomain on each node. Figure 9 shows the communication time spent per-timestep in milliseconds for different implementations, with `Comp` is shown just for reference. `Layout` achieves competitive performance while `MemMap` achieves nearly the same performance as `Network` without any *Packing*; it essentially eliminates on-node data movement with no discernible added cost. In contrast, overheads of `YASK` and `MPI_Types` are significantly higher for any subdomain size; `MemMap` is up to 14.4× faster than YASK and 460× faster than `MPI_Types`.

For subdomain dimension S, per timestep, the amount of computation should scale proportional to S^3, while the volume of data communicated scales with S^2, as would communication time for a fixed bandwidth. The surface-to-volume

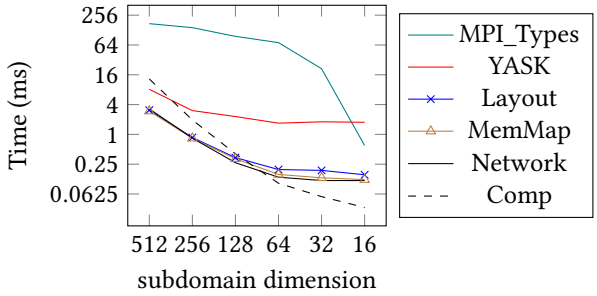


Figure 9. (*K1*) Communication time. `Layout` and `MemMap` are able to achieve competitive communication performance almost achieving the minimum `Network` communication time. For comparison, `Comp` shows computation per timestep in `MemMap`, which is much less than communication time for small subdomains.

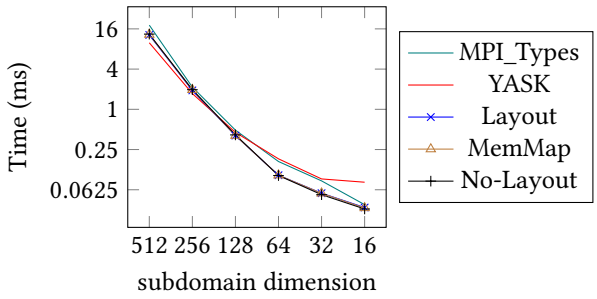


Figure 10. (*K1*) Compute time. Different layouts result in no significant difference. `No-Layout` refers to fine-grained data blocking with no layout optimization, which uses lexicographical ordering for the data blocks.

ratio of $S^2 : S^3 = 1 : S$ suggests that for smaller subdomains, increased time is spent on communication relative to computation. However, for subdomains smaller than 64^3 we also noticed that the communication time is constrained more by communication startup time than network bandwidth (trend to flat time for small subdomains in Figure 9).

Figure 10 shows there is no discernible difference in compute time for different orderings of fine-grained data blocks. This also shows that optimizing the layout for communication does not hurt computation performance. This is because fine-grained data blocking is inherently much better at reducing pressure on cache, TLB, and prefetchers [27]. The difference in performance against YASK arises from different choices of parallelization strategy. YASK uses autotuned two-level OpenMP parallelism that is inefficient for small subdomains. Our method uses a one-level OpenMP parallel schedule that is not optimized for the cache hierarchy, which is why we are slightly slower than YASK on larger subdomains.

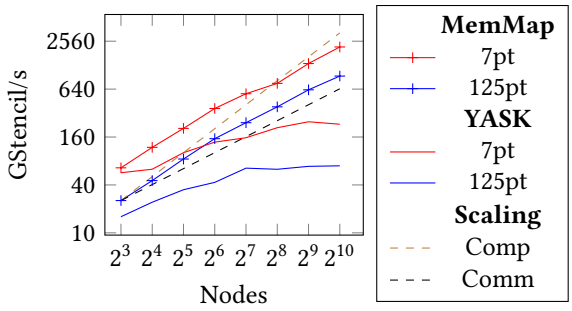


Figure 11. (*K2*) Strong scaling performance on a 1024^3 domain. Strong scaling performance at 1024 nodes is 9.3× (7-point) and 13.4× (125-point) better than YASK. Theoretic scaling for computation (`Comp`) scales with volume, while communication (`Comm`) scales with surface size (dashed lines), with our 125-point `MemMap` results in between.

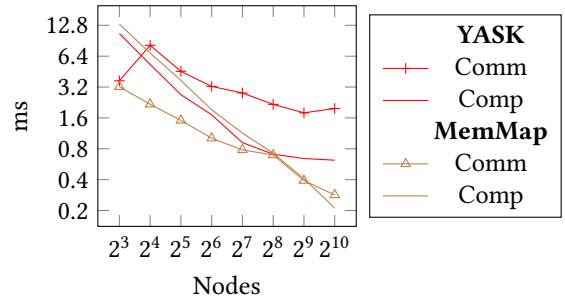


Figure 12. (*K2*) Per timestep communication (`Comm`) vs computation (`Comp`) time decomposition for Strong scaling of 7-point stencil on a 1024^3 domain. Communication time reduction contributes majorly to the significant speedup observed in Figure 11.

(K2) Strong scaling. Figure 11 shows log-log scale strong scaling performance for a 1024^3 domain with both 7- and 125-point stencils, using between 8 and 1024 nodes. For the 7-point stencil, we attain 2166 GStencil/s, while the 125-point stencil achieves 934 GStencil/s on 1024 nodes. With a baseline of 8 nodes, the approach shows good strong scaling, and is compute-bound for smaller number of nodes but scales with communication cost at larger numbers of nodes. In contrast, YASK's throughput is significantly worse at smaller number of nodes and does not scale well at larger numbers of nodes.

Figure 12 separates the average time spent in each timestep into communication and computation. We see that optimizing on-node data movement results in significant performance improvement at larger node counts.

7.2 Summit with NVIDIA Volta GPU

Address translation service (ATS) is supported on Summit, which enables memory mapping through Unified Memory (UM). Spectrum-MPI version 10.3.1.2 is also CUDA-Aware

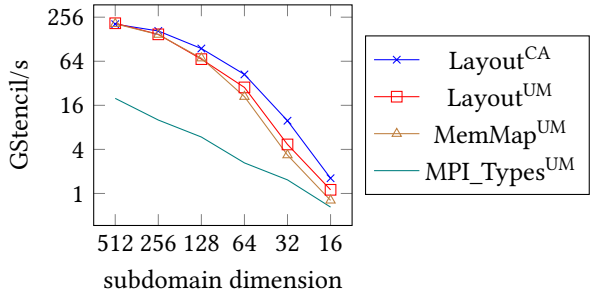


Figure 13. (*V1*) 7-point stencil scaling on 8 NVIDIA V100 GPU nodes. `Layout` and `MemMap` achieves much better performance than `MPI_Types`.

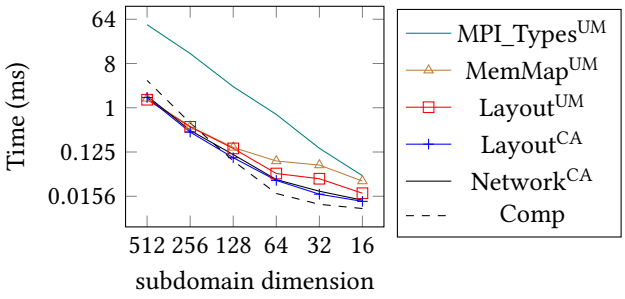


Figure 14. (*V1*) Communication time. `Layout` optimization with CUDA-Aware MPI, `Layout`^{CA}, achieves the best performance, close to the minimum `Network`^{CA} communication time. Different methods have similar compute time; For reference, `Comp` represents time taken to compute in `MemMap`^{UM}.

(CA), which allows direct communication between device pointers.

(V1) 8-node/rank scaling. Figure 13 is a log-log plot that shows the results of experiment (*V1*) on 8 Summit nodes. As with previous experiments, each node is has only one MPI rank which uses just one of the 6 GPUs, resulting in a total of 8 MPI ranks. While MPI_Types can be used directly on CUDA pointers with CUDA-Aware MPI, `MPI_Types`^{CA}, and on host-allocated pointers with Unified Memory, `MPI_Types`^{UM}, only the latter is shown. In our experiments `MPI_Types`^{CA} is more than 50× slower than `MPI_Types`^{UM}.

The main difference in scaling can be attributed to the per-timestep communication time, shown in Figure 14. The layout-based optimization directly uses memory regions allocated with `cudaMalloc` for CUDA-Aware MPI, which provides the best performance (indicated with `Layout`^{CA}). In this case, GPUDirect RDMA can be used by the MPI implementation to bypass host memory for MPI communication.

Figure 15 shows that `Layout`^{CA} and `MemMap`^{UM} result in very similar per-timestep computation time. `Layout`^{UM} and `MPI_Types`^{UM} exhibit worse computation performance because the communicated regions are not aligned to page

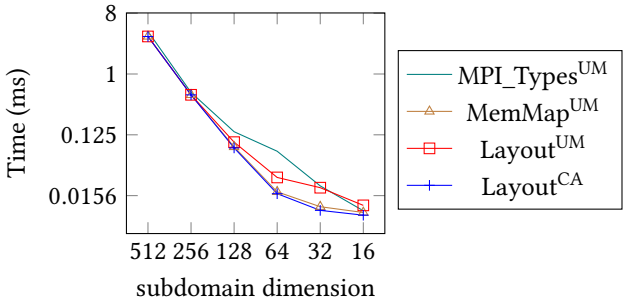


Figure 15. (*V1*) Compute time. `Layout`^{CA} and `MemMap`^{UM} achieves the best computation performance.

Table 2. (*V1*) Network transfer from padding and achieved bandwidth.

Subdomain	**512**	**256**	**128**	**64**	**32**	**16**
Increased network transfer from padding %						
Layout	0	0	0	0	0	0
MemMap	2.4	9.3	35.0	176.9	652.0	883.9
Achieved Bandwidth GB/s						
Layout^{CA}	16.0	21.0	18.6	15.2	9.1	4.7
Layout^{UM}	17.7	16.4	12.0	11.0	4.4	3.2
MemMap^{UM}	17.1	17.6	15.4	16.9	17.3	17.7

boundaries. We can modify the `Layout`^{UM} implementation by padding each communication region to page boundaries, which will result in performance similar to `MemMap`^{UM}. However, this modification then results in significantly worse communication performance due to padding that as a whole slower than `Layout`^{UM}.

`Layout`^{CA} and `Layout`^{UM} perform better than `MemMap`^{UM} due to `MemMap` wasting bandwidth from aligning surface and ghost zone regions to page boundaries. In contrast to 4KiB pages on Theta, Summit uses a larger 64KiB page size. One data block, which is the minimum unit that needs to be transferred on each of the 8 corners of the cube, is 8^3 doubles and yet only 1/16 of a page. Table 2 shows the resulting wasted memory over the Layout optimization, along with the achieved bandwidth for each approach. When comparing the achieved bandwidth, `MemMap`^{UM} is shown not to degrade MPI performance. Also note that layout optimization using unified memory, `Layout`^{UM}, achieves the worst bandwidth performance with smaller subdomains. This is likely due to smaller messages being less efficient to communicate when using unified memory. However, after considering the padding needed for memory mapping, `Layout`^{UM} is still more efficient than `MemMap`^{UM} by communicating far less data.

(V2) Strong scaling. Figure 16 shows strong scaling performance of 2048^3 domain on 8 to 1024 Summit nodes. We used 6 MPI ranks per node where each rank uses 1 V100 GPU, allowing us to scale from 48 to 6144 MPI ranks. Using

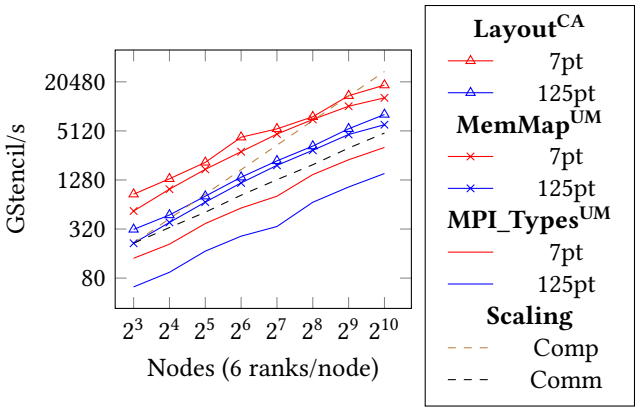


Figure 16. (*V2*) Strong scaling on a 2048^3 domain, `Layout`^{CA} and `MemMap`^{UM}, achieved performance of up to 5.8× and 4.1× at 1024 nodes compared to `MPI_Types`^{UM}. Theoretic scaling for computation (`Comp`) scales with volume, while communication (`Comm`) scales with surface size (dashed lines), with our 125-point `MemMap` results in between.

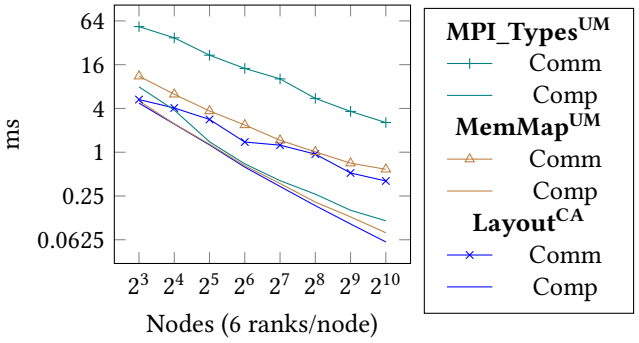


Figure 17. (*V2*) Per timestep communication (`Comm`) vs computation (`Comp`) time decomposition for strong scaling of 7-point stencil on a 2048^3 domain. The application time is dominated by communication even at 8 nodes. Optimizing communication is critical to achieve high performance on Summit and is the source of our significant speedup.

approximately one quarter of Summit, we achieved 18.3 TStencil/s with the 7-point stencil and 8.1 TStencil/s with the 125-point stencil on a 2048^3 domain. This performance is significantly better than `MPI_Types`^{UM}, and `Layout`^{CA} also does not yet appear to be at the strong scaling limit.

Figure 17 decomposes the average time spent in each timestep into communication and computation, and shows that communication is the bottleneck at all scales.

7.3 Impact of Page Size on `MemMap`

As discussed in Section 4, using `MemMap` requires alignment of the surface and ghost regions to page boundaries, which may require padding. This padding results in reduced communication performance for larger page sizes, whereas page

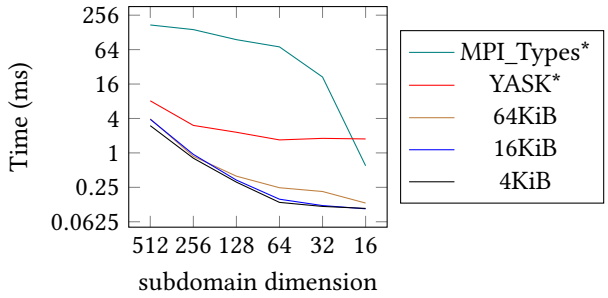


Figure 18. Estimated pagesize scaling effect on communication time with 8 KNL nodes. Even with very large (64KiB) pages, `MemMap` still outperforms both `YASK` and `MPI_Types`.

size does not impact `Layout`. We can estimate the performance impact of different page size on communication by introducing superfluous padding that corresponds to larger page sizes in our 8 node/rank experiments (*K1*), shown in Figure 18. The page sizes are selected from the base page size options in Linux 5.7, where x86 support includes 4KiB, 64bit; for Power, 4KiB, 64KiB; and 64bit ARM can use 4KiB, 16KiB, or 64KiB. Note that for NVIDIA GPUs, when using Unified Memory with System Allocator, the page size is determined by the host page size, which is 64KiB on the 64bit Power9 CPU as configured on Summit. Figure 18 shows that the performance impact of larger page sizes is not significant, and continues to outperform `YASK` and `MPI_Types`.

Seeing that, even for 64KiB pages, `MemMap` still significantly outperforms `YASK`, we believe there are two reasons why page size should not be a significant factor. First, prior work [27] has shown that smaller pages does not limit the computation performance with fine-grained data blocking. Using fine-grained data blocking can reduce key page-related metrics such as TLB misses by up to 49×. Second, larger page sizes like 2MiB (on x86) can be achieved through *hugetlbfs* with Linux. This mechanism is available on x86, ARM, and Power architectures. With this mechanism, multiple page sizes can coexist in the same program or even within the same data structure. Smaller base page sizes, such as 4KiB page on x86, are always available.

8 Related Work

For stencil applications, the impact of MPI communication has been investigated extensively, but in general the impact of memory layout optimization to eliminate on-node data movement has not. There are a few investigations of fine-grained data blocking, such as Briquettes [11], folded vectors in YASK [25], and bricks [27, 28], but none take advantage of reordering memory to improve communication performance. Many MPI benchmarks with stencil loops exist, but these mostly consider latency-bandwidth tradeoffs in inter-node communication, while the cost of on-node data movement is not explicitly addressed (see for example: COMB, SMB,

Cost Type	Array	Layout	MemMap
Strided *Packing*	High	-	-
Extra Msgs	-	Low*	-
Manual CPU-GPU	High	-	-
Large Page	-	-	Low**

Table 3. Compare standard communication practices using arrays with methods in this paper (Layout and MemMap). * Section 3.3. ** Section 7.3.

Intel MPI, CUDA-Aware MPI, and summary in [9]). Several stencil frameworks offer ways to express distribution and parallelization of stencil computations; see S3D-Legion [18] and distributed Halide [6], among many others. Communication avoiding techniques, such as the ghost cell expansion method [7] for stencils and [5, 15] for sparse matrix solvers, explore redundant computations to enable fewer, larger messages. Communication-computation overlap [18, 25] and techniques like time skewing [23] can be used to hide latencies in the network and increase parallelism.

This paper exchanges all neighbors at once, often described as *Put* [16]. Another approach is *Shift* [7, 16] which exchanges ghost zones along each dimension consecutively, excluding corner neighbors. *Shift* avoids latency-dominated small messages through increased synchronization, and is straightforward to implement using memory mapping.

Memory mapping is a common technique in other domains, such as file I/O [20] and distributed shared memory systems [10]. We believe we are the first to combine it with indirection of the application data to optimize the data layout and eliminate on-node data movement during MPI communication.

Implementations of MPI also exist that improve the handling of strided memory access in MPI_Types. Falcon [8] optimizes data type translation for intra-node communication. MPI can also use low-level networking features, like User-Mode Memory Registration [12] in Infiniband, to avoid packing and unpacking. In comparison, these attempts only alleviate the memory access problem, while we have eliminated the problem altogether with layout optimization inside the application.

9 Conclusion and Future work

We have illustrated that optimizing on-node data movement can greatly improve communication performance for stencil applications. Instead of just attempting to hide this overhead, we have demonstrated ways to eliminate on-node data movement on two different architectures. Table 3 compares the methods in this paper, `Layout` and `MemMap`, with standard practices using lexicographical ordered arrays. Both of our methods eliminate the high cost of *Packing* and manual CPU-GPU data movement, by trading these costs with few extra messages (`Layout`) or with increased network transfer from padding (`MemMap`). We provided detailed analysis of our methods using experiments and demonstrated significantly improved communication performance — up to 14.4× compared to YASK.

Overall, this work suggests other opportunities to reduce data movement in current and future systems. For example, using a fine-grained data layout has previously been shown to dramatically reduce vertical data movement through the memory hierarchy [27, 28]; in this paper, it plays a central role in optimizing horizontal data movement across nodes for structured domains. We propose that novel data layouts with indirection can play an important role in achieving performance portability in the face of growing architectural diversity.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract DE-AC05-00OR22725. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] Steve Abbott. 2018. GPUDIRECT, CUDA Aware MPI, and CUDA IPC. In *Summit Training Workshop*. https://www.olcf.ornl.gov/wp-content/uploads/2018/12/summit_workshop_CUDA-Aware-MPI.pdf

[2] Pavan Balaji, Anthony Chan, William Gropp, Rajeev Thakur, and Ewing Lusk. 2008. Non-data-communication Overheads in MPI: Analysis on Blue Gene/P. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 13–22.

[3] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker. 2013. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *20th Annual International Conference on High Performance Computing*. 452–461. https://doi.org/10.1109/HiPC.2013.6799131

[4] Alexandra Carpen-Amarie, Sascha Hunold, and Jesper Larsson Träff. 2017. On expected and observed communication performance with MPI derived datatypes. *Parallel Comput.* 69 (2017), 98 – 117. https://doi.org/10.1016/j.parco.2017.08.006

[5] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. 2008. Avoiding communication in sparse matrix computations. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2008.4536305

[6] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed Halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. ACM, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/2851141.2851157

[7] Chris Ding and Yun He. 2001. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems. In *Proceedings of Supercomputing '01* (Denver, Colorado). ACM, New York, NY, USA,

1. https://doi.org/10.1145/582034.582084
[8] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda. 2019. FALCON: Efficient Designs for Zero-Copy MPI Datatype Processing on Emerging Architectures. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 355–364. https://doi.org/10.1109/IPDPS.2019.00045
[9] S. Hunold and A. Carpen-Amarie. 2019. MPI Benchmarking Revisited: Experimental Design and Reproducibility. https://arxiv.org/pdf/1505.07734.pdf
[10] Ayal Itzkovitz and Assaf Schuster. 1999. MultiView and Millipage — Fine-Grain Sharing in Page-Based DSMs. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 215–228.
[11] Jagan Jayaraj. 2013. *A strategy for high performance in computational fluid dynamics*. Ph.D. Dissertation. University of Minnesota.
[12] M. Li, H. Subramoni, K. Hamidouche, X. Lu, and D. K. Panda. 2015. High Performance MPI Datatype Support with User-Mode Memory Registration: Challenges, Designs, and Benefits. In *2015 IEEE International Conference on Cluster Computing*. 226–235. https://doi.org/10.1109/CLUSTER.2015.41
[13] John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/
[14] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. 2009. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/1654059.1654096
[15] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. 2009. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–12.
[16] Bruce J. Palmer and Jarek Nieplocha. 2002. Efficient Algorithms for Ghost Cell Updates on Two Classes of MPP Architectures. In *IASTED PDCS*.
[17] Björn Sjögreen and N Anders Petersson. 2012. A fourth order accurate finite difference scheme for the elastic wave equation in second order formulation. *Journal of Scientific Computing* 52, 1 (2012), 17–48.
[18] Sean Treichler, Michael Bauer, Ankit Bhagatwala, et al. 2017. S3D-Legion: An Exascale Software for Direct Numerical Simulation of Turbulent Combustion with Complex Multicomponent Chemistry. In *Exascale Scientific Applications*. Chapman and Hall/CRC.
[19] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489.
[20] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. 2015. DI-MMAP—a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing* 18, 1 (01 Mar 2015), 15–28. https://doi.org/10.1007/s10586-013-0309-0
[21] Samuel Williams, Dhiraj D. Kalamkar, Amik Singh, Anand M. Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. 2012. Optimization of Geometric Multigrid for Emerging Multi- and Manycore Processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 96, 11 pages.
[22] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785
[23] D. Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. 171–180. https://doi.org/10.1109/IPDPS.2000.845979
[24] Charlene Yang, Rahulkumar Gayatri, Thorsten Kurth, Protonu Basu, Zahra Ronaghi, Adedoyin Adetokunbo, Brian Friesen, Brandon Cook, Douglas Doerfler, Leonid Oliker, et al. 2018. An empirical roofline methodology for quantitatively assessing performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 14–23.
[25] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK-Yet Another Stencil Kernel: A Framework for HPC Stencil Code-generation and Tuning. In *Proceedings of WOLFHPC '16* (Salt Lake City, Utah). 10.
[26] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software* 4, 37 (May 2019), 1370. https://doi.org/10.21105/joss.01370
[27] T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen. 2019. Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. ACM, New York, NY, USA, Article 52, 44 pages. https://doi.org/10.1145/3295500.3356210
[28] T. Zhao, S. Williams, M. Hall, and H. Johansen. 2018. Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks. In *2018 IEEE/ACM International Workshop, P3HPC*. 59–70. https://doi.org/10.1109/P3HPC.2018.00009
[29] Christopher Zimmer, Scott Atchley, Ramesh Pankajakshan, Brian E. Smith, Ian Karlin, Matthew L. Leininger, Adam Bertsch, Brian S. Ryujin, Jason Burmark, André Walker-Loud, M. A. Clark, and Olga Pearce. 2019. An Evaluation of the CORAL Interconnects. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 39, 18 pages. https://doi.org/10.1145/3295500.3356166

A Artifact Appendix

A.1 Abstract

The artifact associated with this paper includes the brick library enhanced with the ghost zone exchange methods described in this paper using layout optimization. The artifact package also contains relevant code and job scripts to obtain the experiment section results. A `README.md` file is included to describe each experiment's compile instructions and parameters.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Distributed stencil computation.
- **Compilation:** C++ compiler with C++11 and OpenMP support. CMake 3.13 or higher. MPI. Python 3.6 or higher. Optional CUDA Toolkit for GPU experiments.
- **Run-time environment:** Distributed.
- **Hardware:** Optional NVIDIA GPU.
- **Metrics:** Per-timestep timings and overall stencil throughput.
- **How much disk space required (approximately)?:** 10MiB

- **How much time is needed to prepare workflow (approximately)?:** 5 minutes.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License
- **Archived?:** https://doi.org/10.5281/zenodo.4380975

A.3 Description

A.3.1 How to access. The artifact package is available on Github, https://github.com/CtopCsUtahEdu/bricklib/tree/artifact. Clone the repository and check out the `artifact` branch will give local access to the code and documentation.

The package is also archived on Zenodo, https://doi.org/10.5281/zenodo.4380975. One can access the code from Zenodo by download the zip file and extract it.

A.3.2 Hardware dependencies. Any CPU with a paged memory management unit can run the CPU experiments.

NVIDIA GPU is needed for GPU experiments. When utilizing unified-memory with host allocator, corresponding to `MemMap`$^{\texttt{UM}}$ and `Type`$^{\texttt{UM}}$, Power9 host processor with graphics driver supporting Address Translation Service (ATS) is required.

Distributed environment is required for scaling experiments.

A.3.3 Software dependencies.

- A C++11 compatible compiler
- CMake (>= 3.13)
- MPI
- OpenMP
- Python (>= 3.6)
- (Optional) CUDA Toolkit (>= 9)

A.4 Installation

The code can be compiled using CMake. After extracting the code to `<srcdir>`, the following script illustrates a quick way to build the source files.

```
cd <srcdir>
# Prepare
mkdir build && cd build
# Configure
cmake .. -DCMAKE_BUILD_TYPE=Release
# Build
make
```

A.5 Experiment workflow

See the `README.md` file in the artifact package for detailed information and instructions.

A.6 Evaluation and expected results

After building the artifact, `<srcdir>/build/weak` directory will contain executables for running the experiments. Each executable takes command-line options to change the domain size and the number of timing iterations. These options are shown by running it with option `"-h"`. Running one such executable using the appropriate experiment parameters will return the following five performance metrics in the format of `[minimum, average, maximum]` (σ: `standard derivative`).

- **calc** Time spent (in seconds per timestep) for computation
- **pack** Time spent (in seconds per timestep) doing packing and unpacking (not used for MPI_Types)
- **call** Time spent (in seconds per timestep) doing MPI calls (MPI_Isend/MPI_Irecv)
- **wait** Time spent (in seconds per timestep) in MPI_Waitall
- **perf** Overall throughput based on the average of per iteration time

All experiments in this paper require at least 8 nodes for MPI communication. We have supplied example job scripts in the `scripts` directory of the artifact to illustrate how we ran these experiments on the Theta and the Summit supercomputers.

For all experiments, the communication performance of `brick` using the optimizations in the paper is expected to outperform that of `MPI_Types`.