Tausch: A halo exchange library for large heterogeneous computing systems using MPI, OpenCL, and CUDA☆

Lukas Spies [a],[*], Amanda Bienz [c], David Moulton [b], Luke Olson [a], Andrew Reisner [b]

[a] Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL, USA
[b] Los Alamos National Laboratory, Los Alamos, NM, USA
[c] Department of Computer Science, University of New Mexico, Albuquerque, NM, USA

ARTICLE INFO

ABSTRACT

Exchanging halo data is a common task in modern scientific computing applications and efficient handling of this operation is critical for the performance of the overall simulation. Tausch is a novel header-only library that provides a simple API for efficiently handling these types of data movements. Tausch supports both simple CPU-only systems, but also more complex heterogeneous systems with both CPUs and GPUs. It currently supports both OpenCL and CUDA for communicating with GPGPU devices, and allows for communication between GPGPUs and CPUs. The API allows for drop-in replacement in existing codes and can be used for the communication layer in new codes. This paper provides an overview of the approach taken in Tausch, and a performance analysis that demonstrates expected and achieved performance. We highlight the ease of use and performance with three applications: First Tausch is compared to the halo exchange framework from two Mantevo applications, HPCCG and miniFE, and then it is used to replace a legacy halo exchange library in the flexible multigrid solver framework Cedar.

1. Introduction

A common challenge in many parallel scientific codes is communicating boundary data between different processes. The efficiency of this data exchange or *halo* exchange is critical as it is called many times in an application (e.g. iterative solvers) and impacts the overall performance of a code. In this paper we will address codes that employ MPI+X (or pure MPI) for parallel communication, and show how Tausch can be utilized in such contexts.

Halo exchanges are often embedded directly within a larger application, requiring hand-tuning and creating additional effort to maintain. A goal in the present work is to design a stand-alone exchange library that can be used as a drop-in replacement for those applications. To this end, the Tausch[1] library has several design requirements, including

Ease of use: It should be straightforward to incorporate Tausch into an existing code or to add it to a new code. In contrast, existing halo exchange libraries can be complicated to work with, for example with objects living in a global namespace or with a complex API.

Flexible: It should support any type of geometry and any type of data, ideally allowing for different data types to be combined into one message. This approach allows the tool to adapt to the application it is used for, while taking advantage of specific optimizations.

Heterogeneous: It should support both CPUs and GPUs, and their interaction. Ideally the exchange will require minimal user input on the specifics of the communication. Many modern supercomputers are inherently heterogeneous, thus necessitating an exchange library that can handle multiple disparate compute units.

Performant: Communicating data is a non-trivial task as it requires memory movement (contiguous and strided) and network communication, leading to a potential performance bottleneck in the application. The exchange operation should target efficiency, and performance expectations should be clearly defined.

In this work, we detail the Tausch library. It is a header-only C++ library utilizing MPI for communication, thus relieving the user of

---

☆ Los Alamos Report LA-UR-21-28891.
* Corresponding author.
E-mail addresses: lspies@illinois.edu (L. Spies), bienz2@illinois.edu (A. Bienz), moulton@lanl.gov (D. Moulton), lukeo@illinois.edu (L. Olson), areisner@lanl.gov (A. Reisner).
1 The name *Tausch* comes from the German language and translates into English as *exchange* or *swap*.

having to precompile and link to an additional library and allowing for maximum inlining of its member functions. Its aim is to be as unintrusive as possible, making minimal assumptions about the code and the data. It is agnostic regarding the dimension of the application geometry as it works with the data on a memory level, storing the information about the halos in a compressed format minimizing memory requirements. Tausch also supports halo exchanges on heterogeneous systems: it facilitates halo exchanges across CPUs and GPUs in any combination through a single API. It currently supports both OpenCL [1] and CUDA [2] for communicating with GPUs. There are various performance optimizations implemented in Tausch, all of which can be enabled/disabled with the call to a single member function. Examples of such optimizations include derived data types for MPI, and direct memory copies. All operations in Tausch can be called in a non-blocking (asynchronous) fashion, with each method returning a handler object for managing these operations, checking their status and waiting for their completion. In all, Tausch provides a flexible interface that can adapt to any setup while providing several low-level optimizations to boost performance.

There are several existing solutions for communicating halos. We first discuss several tools that address halo exchanges in a generic way, allowing for their integration into any user code. Then we mention several existing frameworks that include their own halo handling. Most of the existing generic tools are targeted towards a specific use case or situation, with some no longer maintained. The Data Transfer Kit (DTK) [3] is designed primarily for physics applications, where geometric domains may not conform to the same physical space, potentially with mismatched parallel decomposition. These features are valuable when needed, but they also introduce unnecessary complexity. The Generic Communication Layer (GCL) [4] is a library of communication patterns where the halo exchange operation is divided into different layers that can be tweaked and updated independently. It is a templated header-only C++ library and allows for flexibility in how it can be used, leading to a more complex API. GCL is described as "old code" in its GitHub repository [5], with its last update in 2017. We are not aware of any applications making use of GCL.

Raja [6,7] is a library of C++ software abstractions aiming to enable architecture and programming model portability for high performance applications. It also provides constructs for efficient packing and unpacking of data on different computing devices, although it does not facilitate any actual communication. Tempi [8] is another approach that specifically targets MPI+CUDA, improving the performance of MPI using CUDA buffers. This design is achieved through MPI and can thus be easily combined with other tools and libraries that use MPI. A different approach is taken by Kokkos [9], where a new programming model is developed that offers local mapping and execution on a variety of architectures. It does not handle halo exchanges and defers to other codes and frameworks for those. Tpetra [10] is a package for Trilinos [11] implementing linear algebra objects that uses Kokkos for local operations and provides the necessary code for facilitating halo exchanges. PETSc also provides its own handling of halo exchanges as part of its distributed arrays (DMDA). All of these come with their own programming models and require the user's code to adapt to that. Thus, they require the use of their own custom data structures and also typically necessitate large code rewrites. Finally, the MiniGhost [12,13] application in the Mantevo Project [14] is written in Fortran and serves as stand-alone code to explore and experiment different programming models. It was last updated in 2016.

To demonstrate the flexibility and performance of Tausch we perform a number of computational tests and highlight two applications. These computational experiments make use of the *Lassen* supercomputer, part of the Livermore Computing Center. Lassen employs the IBM Power9 CPU architecture with 40 CPU cores per node and a CPU memory bandwidth of 170 GB/s. Each node is equipped with 4 NVIDIA V100 (Volta) GPUs with each GPU having 5120 cores, 7 TFLOPS peak performance, 32 GB memory, and 900 GB/s GPU memory bandwidth.
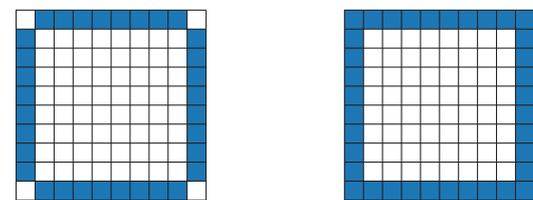
(a) halo not including corners   (b) halo including corners
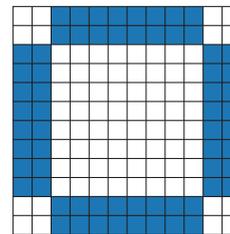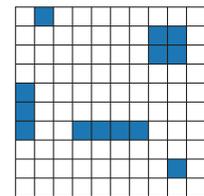
(c) halo of width 2   (d) unstructured halo

Fig. 2.1. Examples of different halos, with the halos highlighted in blue.

The compiler used is GCC 7.3.1 together with Spectrum MPI 10.03 and CUDA 10.1. We present some simple results for the now decommissioned BlueWaters system [15] that was hosted at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana/Champaign in Appendix.

The paper is organized as follows. In Section 2 we provide an overview of Tausch, both on a conceptual level and also detailing the API. We illustrate how Tausch compresses halo information internally in order to reduce the memory requirement and also to improve performance. In Section 3 we provide an overview of the various communication strategies that are currently provided by Tausch. We describe how they have the potential to boost the performance given the right data and hardware. In Section 4 we provide a performance analysis of Tausch for both a three dimensional test case on the CPU and on the GPU. In Section 5 we compare the performance of Tausch to the performance of two established codes, HPCCG [16] and miniFE [17], and we discuss another project, Cedar [18], that incorporates Tausch as their halo communication layer.

2. Tausch overview

Tausch provides a high-level API for halo exchanges using MPI [19], CUDA [2], and OpenCL [1]. The user determines the traffic pattern, where the data to be sent lives in memory and where the received data is to be written to in memory. Tausch then offers various strategies to achieve a high-performance exchange of the specified data, whether the data comprises a halo or any other type of data. It is a header-only library, thus relieves the user of having to precompile and link to an additional library, and it allows for maximum inlining of its member functions. It is written in C++ with a fully compatible C API, and a Fortran interface is also available.

To begin, we first define the notion of a *halo* in the exchange of data. A halo is any structured or unstructured area that is used but typically not owned by the local process. In most applications a halo would lie along the edges of a domain, though this is not a requirement for Tausch. We refer to data that needs to be sent to another partition's halo region as the *send halo* and, conversely, data that needs to be updated locally with values received from another partition as the *receive halo*. Fig. 2.1 illustrates various types of halos, both structured and unstructured, all of which can be handled by Tausch.

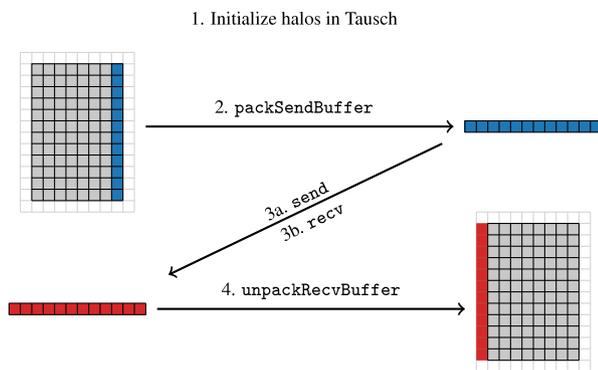*L. Spies et al.*

Parallel Computing 114 (2022) 102973

1. Initialize halos in Tausch

2. packSendBuffer

3a. send
3b. recv

4. unpackRecvBuffer

Fig. 2.2. High-level overview of Tausch.

2.1. High-level overview

Facilitating a halo exchange with Tausch consists of several steps. Fig. 2.2 shows a schematic of that process, where an 8×12 grid resides on one node and the right column of 12 data elements is sent to the node owning the adjacent block of data.

1. *Describe the halo.* Tausch requires a description of the halo regions from where in memory to either read the sending halo data (on the sending process) or where in memory to write the received halo data (on the receiving process).
2. *Pack the data.* Closely following the implementation of a halo exchange in pure MPI, Tausch packs the data into a dedicated send buffer, allowing the user to continue computations on the main buffer as the data to be sent is held separately.
3. *Send and receive data.* This corresponds closely to a call to `MPI_Send` and `MPI_Recv`. Internally Tausch creates a communication channel that is reused on subsequent calls to this particular send.
4. *Unpack the data.* The data is copied from a dedicated receive buffer into the main buffer on the receiving end.

The above four-step process outlines the basic way of handling halo exchanges using Tausch. Depending on the underlying hardware and MPI implementation, the user may choose certain optimizations that modify this four-step process. For example, when calling Tausch to take advantage of MPI derived datatypes, the steps to pack and unpack the data are not required. We will revisit this and other examples in more detail in Section 3.

2.2. Language overview (C++, C)

Tausch is a header-only library written in C++, providing a fully compatible C API and a Fortran interface. This makes the process of integrating it into any project very straightforward. In the following we will detail a subset of the API.

2.2.1. Constructor

```
Tausch(
  const MPI_Comm comm,
  const bool useDuplicateOfCommunicator,
  OutOfSync handling
)
```

The default constructor takes three arguments: The MPI communicator to be used (default: `MPI_COMM_WORLD`), whether to take a duplicate of the communicator (default: `true`), and whether to check for race conditions and what to do then (default: `WarnMe`). Duplicating the communicator isolates Tausch from other communication and avoids any potential interference.

2.2.2. Halo regions

```
int addSendHaloInfo(
  std::vector<int> haloIndices,
  const size_t typeSize,
  const size_t numBuffers,
  const int remoteMpiRank
)
int addRecvHaloInfo(
  std::vector<int> haloIndices,
  const size_t typeSize,
  const size_t numBuffers,
  const int remoteMpiRank
)
```

There are multiple approaches to specifying a halo region. The simplest way is to pass a vector of indices for halo values to Tausch. The second parameter specifies the byte size of that data — e.g., `sizeof(real_t)`. The third parameter specifies how many buffers can be combined along the same communication paths (default: 1). If there is more than one buffer using the same communication path, then they can either all use the same halo specification (for example, multiple variables at each point), or use different halo specifications (for example, two different domains that are used simultaneously). The fourth parameter specifies the receiving MPI rank (default: −1, meaning do not set a fixed remote rank). The receiving MPI rank may be the same MPI rank as the sender, which allows Tausch to optimize the halo exchange accordingly. The function also returns the ID of the halo for subsequent invocations on the halo region.

Internally Tausch does not store the full vector of indices but instead forms a compressed view. Section 2.4 shows how that is done and its implications. At the same time, the compressed form can be sent directly to Tausch rather than a raw vector of indices.

Specifying a *receive* halo region is similar to that of the send halo, with the main difference being the MPI rank passed on as fourth argument, which refers to the sending MPI rank instead of the receiving MPI rank.

2.2.3. Packing and unpacking data

```
Status packSendBuffer(
  const size_t haloId,
  const size_t bufferId,
  unsigned char *buf,
  const bool blocking
)
Status unpackRecvBuffer(
  const size_t haloId,
  const size_t bufferId,
  unsigned char *buf,
  const bool blocking
)
```

Packing the send data requires moving data from the main buffer into a dedicated send buffer, which is internal to Tausch. The halo ID is the integer returned by calls to `addSendHaloInfo` and `addRecvHaloInfo`. The buffer ID (starting at 0 counting up) is important only if more than one buffer is combined as one message. The third parameter is the main buffer where the send halo data is stored. Unpacking the received data is done in an equivalent way to the packing process.

By default, packing and unpacking data is done while blocking the main thread until the operation has completed. With `blocking=false` the packing and/or unpacking is done asynchronously and this function returns a `Status` object for that process. Making sure that the process has completed before its data is used further has

3

to either be taken care of by the user, or Tausch can be set to either print a warning or wait on these processes whenever a send is called after a non-blocking pack, or an unpack is called after a non-blocking receive.

The `Status` object provides a unified way to handle such calls, it provides methods to check its status (`isRunning()` and `isCompleted()`), and `wait()` in order to block the main thread until the connected operation has completed. Inside this object lives a `std::shared_future<void>`, an `MPI_Request`, a `cudastream_t`, or an OpenCL `UserEvent`, but the user does not have to worry about which one it is. However, if desired, the underlying object can be obtained using conversion. For example, assigning the `Status` object to an object of type `std::shared_future<void>` will return the future contained inside `Status`.

When calling Tausch with derived MPI datatypes for communication, these calls are not required as the data is sent directly from and received directly into the main buffers. This optimization avoids the additional copy performed at these steps, however each buffer sends a separate message instead of combining multiple buffers in one message. Additionally, the halo data in the main buffers cannot be touched while communication remains active. These performance trade-offs require careful consideration.

2.2.4. Sending and receiving data

```
Status send(
  size_t haloId,
  const int msgtag,
  int remoteMpiRank,
  const size_t bufferId,
  const bool blocking,
  MPI_Comm communicator
)
Status recv(
  size_t haloId,
  const int msgtag,
  int remoteMpiRank,
  const size_t bufferId,
  const bool blocking,
  MPI_Comm communicator
)
```

Calls to `send` and `recv` move the data between different MPI ranks. The halo ID is the integer returned by the call to `addSendHaloInfo` and `addRecvHaloInfo`. The message tag corresponds to the integer tag required for MPI communication. The remote MPI rank refers to the sender/receiver of the message (default: −1, meaning take the rank specified when adding halo information). When multiple buffers use the same communication path and thus share the same halo ID, then the buffer ID specifies which one of these buffers we are operating on (counter starting at 0). For the `send` and `recv` calls, this parameter only needs to be specified when MPI derived datatypes are used (default: −1). The send operation is by default non-blocking (`blocking=false`) whereas the receive operation is by default blocking (`blocking=true`). The default values for `blocking` are not due to any particular performance consideration, but rather were chosen arbitrarily as sensible defaults. The final parameter enables temporary overwriting of the MPI communicator for the specific call to `send` or `recv`. If not specified (or set to `MPI_COMM_NULL`) the communicator specified during construction of Tausch is used. Both methods return a `Status` object (containing the underlying `MPI_Request` used for each operation).

2.2.5. Other member functions

```
void setSendCommunicationStrategy(
  size_t haloId,
  Communication strategy
);
void setRecvCommunicationStrategy(
  size_t haloId,
  Communication strategy
);
```

This enables any specific communication strategies for sending and receiving. These need to be called after the respective halo has been set up and before any communication happens. See Section 3 for a detailed overview of the different communication strategies.

```
void setOutOfSyncHandling(
  OutOfSync handling
);
```

All the functions in Tausch doing the heavy lifting (pack/unpack and send/receive) have the option to be called without blocking the main thread (`blocking=false`). Even though Tausch cannot guarantee that using this option will not lead to a race condition, it can make sure that a send waits for the corresponding pack, and that an unpack waits for the corresponding receive. There are three possible values of the `OutOfSync` enum, the first one is `DontCheck`. The second and default value is `WarnMe`, all this does is print a message to the screen that a potential race condition has been detected. The third possible value is `Wait` which makes Tausch wait for the pack or receive to complete before proceeding with the send or unpack.

2.3. OpenCL and CUDA

Using Tausch in combination with OpenCL and CUDA is nearly identical to the API described in Section 2.2. In order to use either or both of these technologies, the macros `TAUSCH_OPENCL` and `TAUSCH_CUDA` are required *before* including the header file. Only the process of packing/unpacking data require a different API call, `packSendBufferOCL` and `unpackRecvBufferOCL` (`packSendBufferCUDA` and `unpackRecvBufferCUDA` respectively), which contains OpenCL/CUDA specific code. In addition, before using the OpenCL feature of Tausch, the specific OpenCL environment must be specified, either by passing an existing OpenCL environment to Tausch (`setOpenCL`) or by requesting Tausch to set up an environment (`enableOpenCL`).

In order to facilitate GPU-to-GPU communication, Tausch by default first transfers GPU data to the CPU and then does the data transfer using MPI before transferring the data to the receiving GPU. Most MPI implementations allow GPU-to-GPU communication to happen without going through the CPU. We will revisit this as part of our communication strategies in Section 3.4.

For NVIDIA GPUs using CUDA, Tausch also supports multiple GPUs per MPI rank, as it can work with pointers to memory regions on different GPUs.

2.4. Compressed storage of halo information

Tausch uses a compressed format to store halo information instead of storing a full vector indices. The compressed storage used by Tausch is optimized for structured halo regions, however, it will work for any halo region form or shape.

The user can either directly specify the halo regions using the compressed format, or make use of a convenience function that takes in a set of indices of halo data and converts it into the compressed format. In the latter case, Tausch decomposes that region up into rectangular subregions corresponding to how the data is laid out in memory. Such

Fig. 2.3. Example of compressed storage: 10 integers (40 bytes) stored using 4 integers (16 bytes), halo region highlighted in blue.

Fig. 2.4. Example of rectangular region not corresponding directly to mesh region, halo region highlighted in blue.

a region does not necessarily translate to a rectangular region in the physical setup. Each such subregion is defined using these 4 integers (see Fig. 2.3):

1. Starting index of the region;
2. number of consecutive values (i.e., number of columns);
3. frequency of consecutive values (i.e., number of rows); and
4. stride between the sets of consecutive values.

Using a compressed form allows highly efficient memory operations using `memcpy`, but also using strided copies in OpenCL and CUDA. Additionally, the memory requirement of storing halo information is drastically reduced, particularly in cases of structured data. In the example given in Fig. 2.3 the compressed storage requires $2/5$ of the memory required for a full set of halo indices, and the effect increases with larger halo regions. Yet, in the case of unstructured halo regions, the memory requirement of using the compressed storage might increase if the region does not easily decompose into rectangular subregions.

The rectangular subregions found by Tausch do not necessarily correspond to rectangular regions in the mesh. Instead, in a slightly more abstract sense, they correspond to rectangular regions in the memory — e.g., the example shown in Fig. 2.4 illustrates how a 10×2 rectangular region in the mesh is detected as 20×1 rectangular region in the memory.

The same concept extends to three dimensions, where a three dimensional volume is interpreted by Tausch as a two or possibly one dimensional memory region. It is also possible to directly pass the halo region information in compressed form to Tausch instead of vectors of indices. In the case of the example shown in Fig. 2.4 both representations (10×2 and 20×1) are valid.

In general, a one dimensional compression is preferential to a two dimensional one, as it allows the use of fewer `memcpy` operations and thus offers better performance. Since a halo inherently corresponds to the surface of the domain (i.e., at most two dimensional), a three dimensional compression of the halo brings little to no benefit while increasing the overhead of the actual compression step.

3. Communication strategies

Tausch implements a robust default strategy: always pack the data wherever it lies into a dedicated send buffer on the CPU that is communicated with MPI. Even though this is guaranteed to execute properly, it possibly yields suboptimal performance. We next review several additional methods that can be enabled by the user.

3.1. Derived datatypes

When communicating data between CPU ranks, the use of derived datatypes avoids copying the data-to-be-sent into a dedicated send buffer before handing it off to MPI (step 2 in Section 2.1) and, similarly, on the receiving end skips the intermediate step of receiving the data into a dedicated receive buffer before distributing the data into the main buffer (step 4 in Section 2.1).

Skipping these two copy operations offers the potential for improved performance if implemented efficiently in MPI. However, it also means that the locations of the data-to-be-sent cannot be altered until the send operation has completed. Adding the intermediate step of copying the data into a dedicated send buffer mitigates this caveat (at the possible expense of performance).

3.2. Persistent communication

MPI supports persistent communication, where a communication channel is established between a sender and receiver including the sending and receiving buffers and any information required for the communication. Such a channel can then be re-used repeatedly in subsequent iterations. Bypassing this overhead has the potential for improved performance if it is implemented efficiently in MPI. When enabled, Tausch will manage persistent communication channels without requiring additional user interaction.

3.3. Single-copy and multi-copy

When halo data needs to be moved to or from a GPU, then Tausch can perform this copy in one of two ways:

Single-copy: first transfer all received data as a contiguous memory buffer to the device in a single memory copy followed by a redistribution of data on the device; or

Multi-copy: directly transfer data to the corresponding memory locations on the device using two-dimensional memory copies.

The advantage of the former is that the data movement between the CPU and the GPU is done with a single memory copy and one big chunk of data, independent of the data shape. In applications where the memory movement between the CPU and GPU is a bottleneck, this can optimize the performance of that operation at the expense of an additional memory copy on the device.

The advantage of the latter is that the data is directly written to the appropriate memory locations on the device, requiring each subregion of the halo to execute its own memory copy operation. This can be expensive if the halo region consists of many subregions, but may be less expensive if the halo region is an almost perfect rectangle. Fig. 3.1 shows a comparison of these two approaches in three dimensions on Lassen. The test case is a cube with the halo region along the surface — i.e., 6 halo subregions. Based on the results from Fig. 3.1 there is a reasonable expectation for the single copy strategy to yield better performance, thus Tausch implements this as a default.

3.4. CUDA-aware MPI

A special case arises when two or more GPUs communicate with each other using CUDA. If CUDA-aware MPI is available and supported by the underlying architecture, then moving the data between the GPUs can be done without (explicitly) going through the CPU. In order to use this feature in Tausch, the communication strategy for a sending and/or receiving halo id needs to be set, everything else will be handled internally by Tausch. Fig. 3.2 compares the performance of using CUDA-aware MPI to the default strategy of passing the data through the CPU for 100 iterations of a simple halo exchange using Tausch across 512 GPUs on Lassen for a 7-point stencil. Note that we
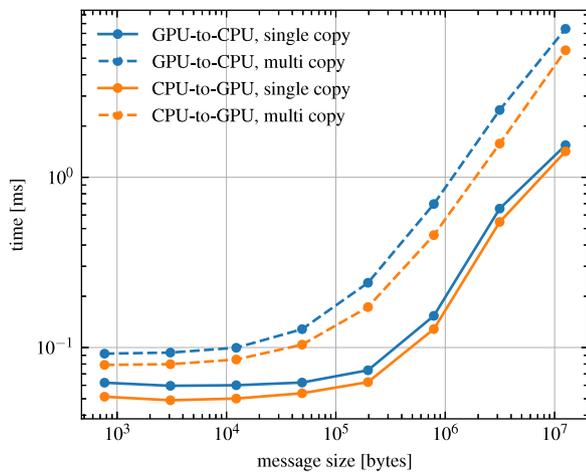
Fig. 3.1. Data movement in three dimensions between the CPU and GPU for NVIDIA V100 (Volta) on Lassen.
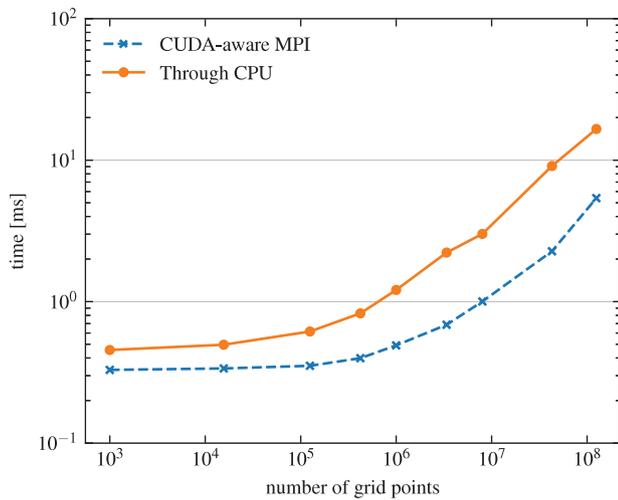

Fig. 3.2. Comparing performance of halo exchange for 7-point stencil using CUDA-aware MPI and passing through CPU on 512 GPUs on Lassen.

do not measure the stencil applications but only the communication required for such a stencil. The colored regions show the range of values (min/max) across all ranks, the lines show the average timings.

Fig. 3.2 shows that using CUDA-aware MPI has the potential to greatly improve the performance, up to an order of magnitude. Since this feature is highly dependent on the underlying MPI architecture, it is hard to predict whether this feature is available and whether it will, in fact, lead to a performance gain. However, it clearly as the potential to do so, thus it is important for Tausch to support either communication path, giving the user maximum flexibility. It is worth pointing out that using CUDA-aware MPI with Tausch is as simple as setting the communication strategy to use such, no other adjustments are necessary.

4. Performance analysis

In this section we explore the performance of Tausch in more detail in order to highlight its efficiency and to expose potential bottlenecks.

4.1. Performance model

We consider two different performance models, one that only models the communication (based on the *max-rate model* [20]) and one that
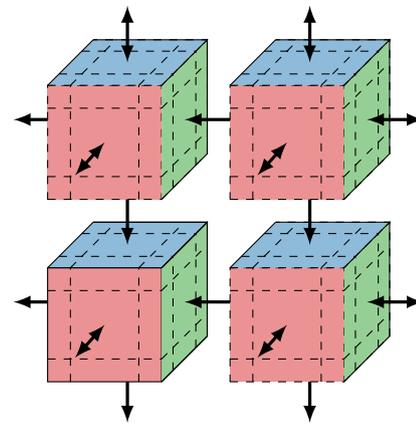

Fig. 4.1. Visualization of three-dimensional halo exchange used as test case.

includes packing/unpacking the data before and after communicating. The max-rate model is an extension of the traditional postal model, and can capture injection limits observed on SMP nodes. For a detailed analysis of the max-rate model we refer the reader to [20].

In order to compare the performance of Tausch to the performance model we consider a test that performs a three-dimensional halo exchange. Fig. 4.1 shows a visualization of this halo exchange test in three dimensions, and Algorithm 4.1 describes the actual algorithm that is being used. The test code implementing this example is run on the Lassen supercomputer.

Algorithm 4.1 Algorithm of test code

1: Create data buffers
2: Compose halo information
3: n_test ← number of tests
4: n_timing ← number of timings per operation
5: **for** test←1, n_test **do**
6:    MPI_Barrier
7:    **Start** pack timer
8:       **for** t←1, n_timing **do**
9:          Pack halo data to be sent off
10:            into dedicated send buffer
11:       **end for**
12:    **Stop** pack timer
13:    **Start** communication timer
14:       **for** t←1, n_timing **do**
15:          Send data off to neighbors
16:           using MPI_Isend
17:          Receive data from neighbors
18:           using MPI_Irecv + MPI_Wait
19:       **end for**
20:    **Stop** communication timer
21:    **Start** unpack timer
22:       **for** t←1, n_timing **do**
23:          Unpack received halo data
24:            out of dedicated receive buffer
25:       **end for**
26:    **Stop** unpack timer
27: **end for**

The max-rate performance model, which focuses on communication, can be expressed using the following equation,

$$T = t_c + r\alpha + \frac{k\,n}{\min(R_N, kR_C)} \tag{4.1}$$

where t_c is the time for copying the data into/out of dedicated send/receive buffers, r is the number of messages a rank is sending, α is the

Table 1
Max-rate model parameters for Lassen, obtained through experiment.

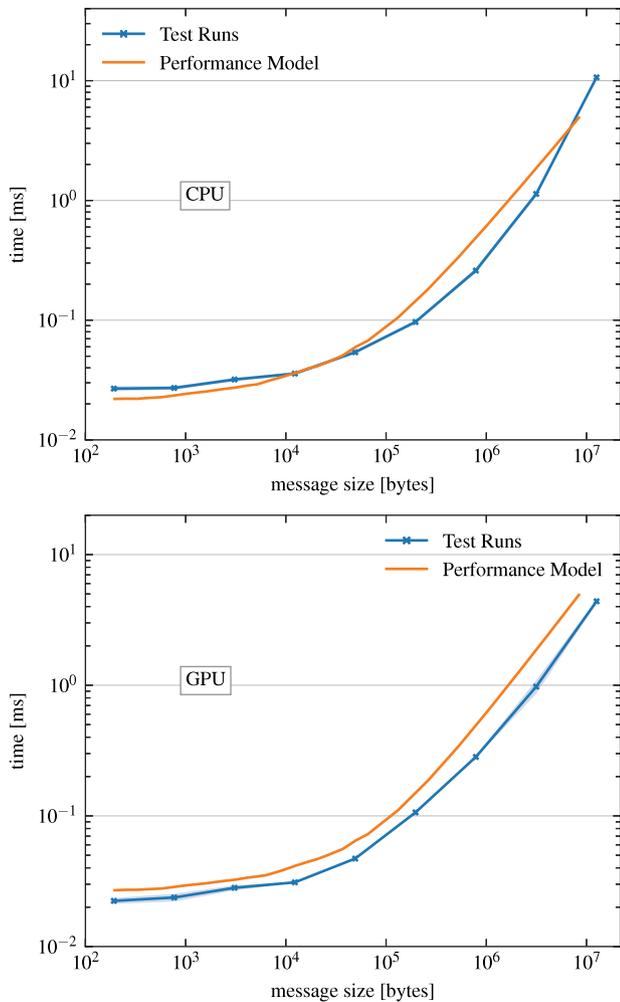| Protocol	α [s]	R_N [B/s]	R_C [B/s]
Short	1.38×10^{-6}	–	3.81×10^{9}
Eager	2.26×10^{-6}	–	2.36×10^{9}
Rendezvous	1.14×10^{-5}	2.28×10^{9}	1.77×10^{10}

Fig. 4.2. Performance Model on CPU using 320 ranks across 8 nodes (top) and GPU using 32 ranks across 8 nodes (bottom) running on Lassen.

Fig. 4.3. MPI_Pack vs packSendBuffer in Tausch, packing surface of three dimensional cube on both the (top) CPU and (bottom) GPU with CUDA, on Lassen.

latency introduced by MPI per message, k is the number of processes, n is the number of bytes sent per process, R_N is the *injection bandwidth* (how fast data can leave or enter the node and leave or enter the network), and R_C is the rate that can be achieved by each process in sending or receiving a message. The values for α, R_N and R_C can be found in Table 1, with the values obtained through experiment. Note that R_N only impacts the rendezvous protocol.

We present a performance evaluation for test runs on both the CPU and the GPU. Note that the only difference between those two is in the copying the halo data into their dedicated send buffers, for the GPU test runs this involves calls to cudaMemcpy.

Fig. 4.2 shows the comparison of the performance model and the test code. The colored regions show the range of values (min/max) across all ranks, the lines show the average timings. In order to get a handle on the average expected performance, the parameters for the performance model are for inter-node communication (i.e., using the network) and for copying only consecutive chunks of memory (without stride). Thus, the model will be slightly too optimistic for memory
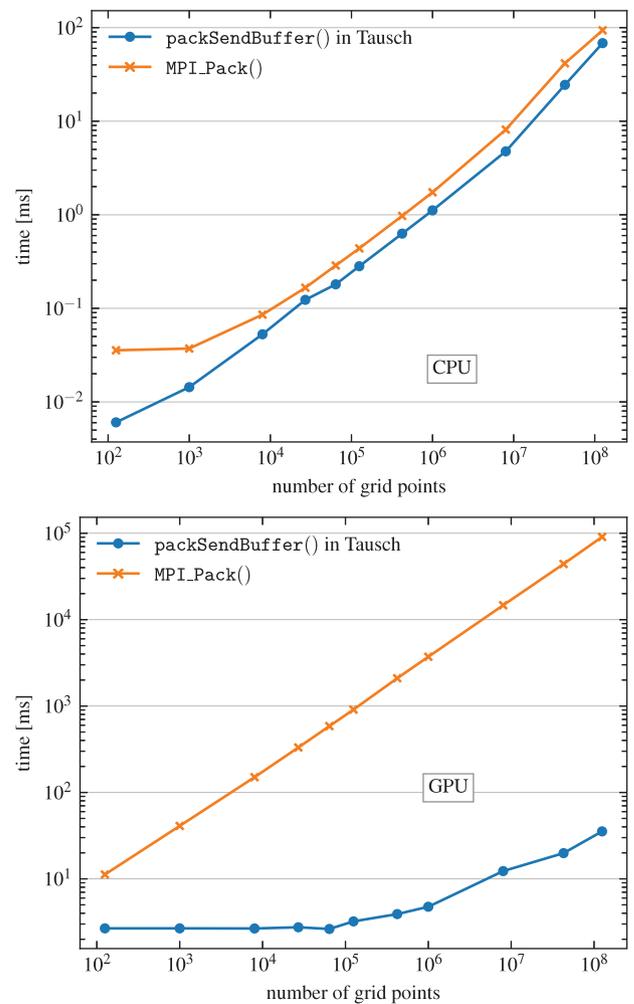
copies, especially for the larger problem sizes. For the smaller problem sizes data located at some stride still falls within one or just a few cache lines resulting in a performance that is near ideal. On the other hand, the communication prediction of the model is slightly pessimistic, ranks that lie on the same node and/or socket will result in faster communication performance than the predicted performance. Overall, the prediction by the performance model will be an average of the best and worst performance between any two ranks.

The minimum values for the test runs are the fastest time for doing a halo exchange between any two ranks, and likely stem from two ranks living on the same socket. Conversely, the maximum values likely stem from two ranks living on different nodes that are far apart. From the results we see that the modeled and actual performance closely align.

4.2. Comparison to MPI_Pack

MPI provides its own routines for packing/unpacking, MPI_Pack and MPI_Unpack. Fig. 4.3 shows a comparison of MPI_Pack to the packing routine in Tausch on both the CPU and GPU using CUDA-aware MPI. The test case is a three dimensional cube whose surface is packed into a six dedicated send buffers (to be sent to its 6 neighbors). The test code for MPI_Pack is implemented using plain MPI and is not used/supported in Tausch. We see that Tausch performs better than MPI_Pack for packing data: up to 5 times faster on the CPU and up to 2500 times faster on the GPU. The speedup is at least in part due to the
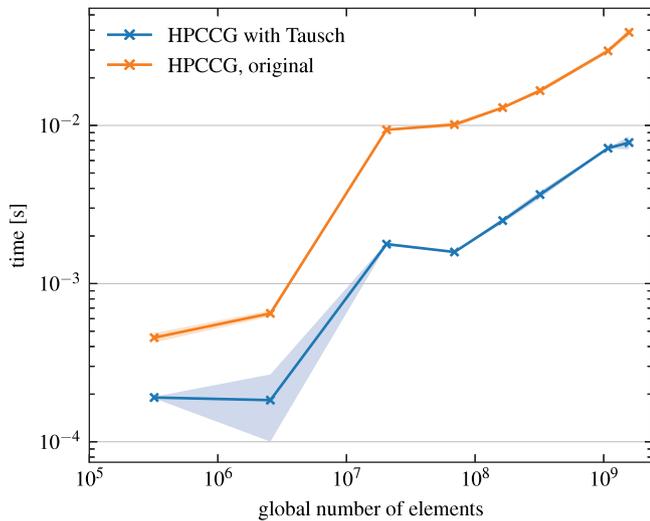
Fig. 5.1. Halo exchange in HPCCG with and without Tausch using 320 MPI ranks across 8 nodes on Lassen (40 ranks per node).

fact that Tausch is optimized for structured communication, whereas `MPI_Pack` is assuming a more general pattern.

5. Example applications

In this section we highlight the ease of use and performance with three applications. First, Tausch is compared to two Mantevo applications, HPCCG [16] (CPU-only) and miniFE [17] (CPU and GPU), and lastly we use it to replace a legacy halo exchange library in the flexible multigrid solver framework, Cedar [18]. Due to the flexibility of Tausch and the simplicity of its API, dropping Tausch into existing code is straightforward.

5.1. Tausch in HPCCG

The Mantevo application HPCCG [16] is a simple conjugate gradient code that generates a 27-point finite difference matrix for a 3D chimney domain on an arbitrary number of processors. It captures the performance of popular Krylov based linear solvers that rely on key linear algebra operations, such as sparse matrix–vector multiplications and dot products. It was chosen for a comparison with Tausch as it exhibits strong similarities to Tausch in the handling of halo data. Thus, this allows us to get an accurate understanding of the performance of Tausch in comparison to an established code. We started out by running the original application on Lassen on 320 CPU cores (spread across 8 nodes). Then we replaced the halo exchange logic with calls to Tausch and re-ran the code with the same configuration. The result it shown in Fig. 5.1.

Fig. 5.1 shows a comparison of HPCCG with and without Tausch. The colored regions show the range of values (min/max) across all ranks, the lines show the average timings. Tausch outperforms HPCCG, often improving performance by an order of magnitude. One reason for this performance boost is the re-use of information about the halos. For example, where HPCCG is re-creating intermediate buffers at each iteration, Tausch is able to re-use buffers of the right size from previous iterations. Tausch also encodes the halo information once during setup allowing for more performant memory operations to be done at each iteration instead of simple looping over each halo data point. These optimizations are not trivial to be implemented by hand, but come for free with the use of Tausch. They all result in a lower overhead per iteration and thus a significant speed-up. This has a noticeable effect on the overall absolute runtime as up to 10% of the total runtime is spent in the halo exchange.
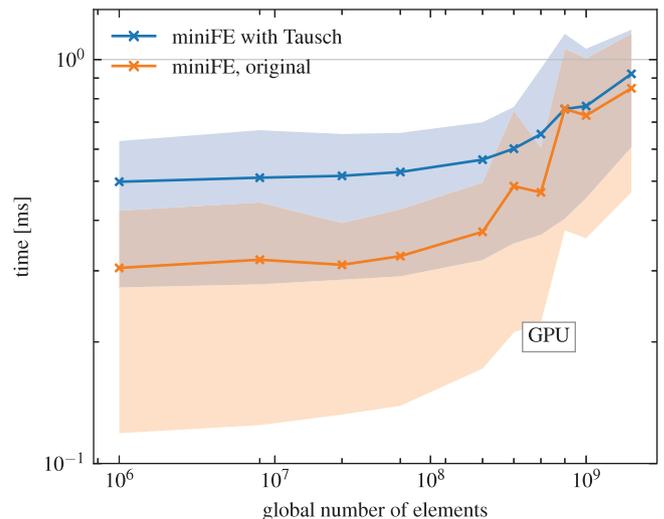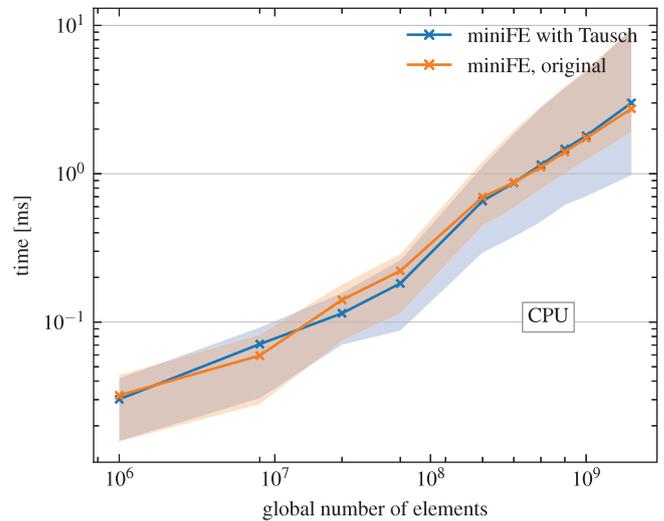

Fig. 5.2. Halo exchange in miniFE with and without Tausch using 320 ranks across eight nodes on the CPU (top) and across 128 nodes on the GPU (bottom).

5.2. Tausch in miniFE

The Mantevo application miniFE [17] provides implementations of an unstructured finite elements code on various platforms. It provides implementations on both the CPU and the GPU (using CUDA) with a clear implementation of its own halo handling, making it very straightforward to switch to using Tausch. Both the CPU and GPU runs are done with 320 ranks, spread across eight nodes for the CPU and 128 nodes for the GPU. We first ran the original code and then the modified version with Tausch. The result is shown in Fig. 5.2.

The colored regions show the range of values (min/max) across all ranks, the lines show the average timings. The upper plot in Fig. 5.2 shows that on the CPU the halo exchange in miniFE takes about the same amount of time with Tausch and the original code, the difference between the two is negligible. On the GPU, the original code performs slightly better than Tausch, yet the difference between the two is still negligible. The clear advantage of using Tausch, however, is the simple interface, the need for fewer lines of code, and the minimization of the potential need of restructuring and refactoring of the code. The overall proportion of time spent in the halo exchange is less than 30% in the GPU case and less than 25% in the CPU case.

It is worth pointing out that the part of the code that handles Tausch is nearly identical in both the version for the CPU and the one for the
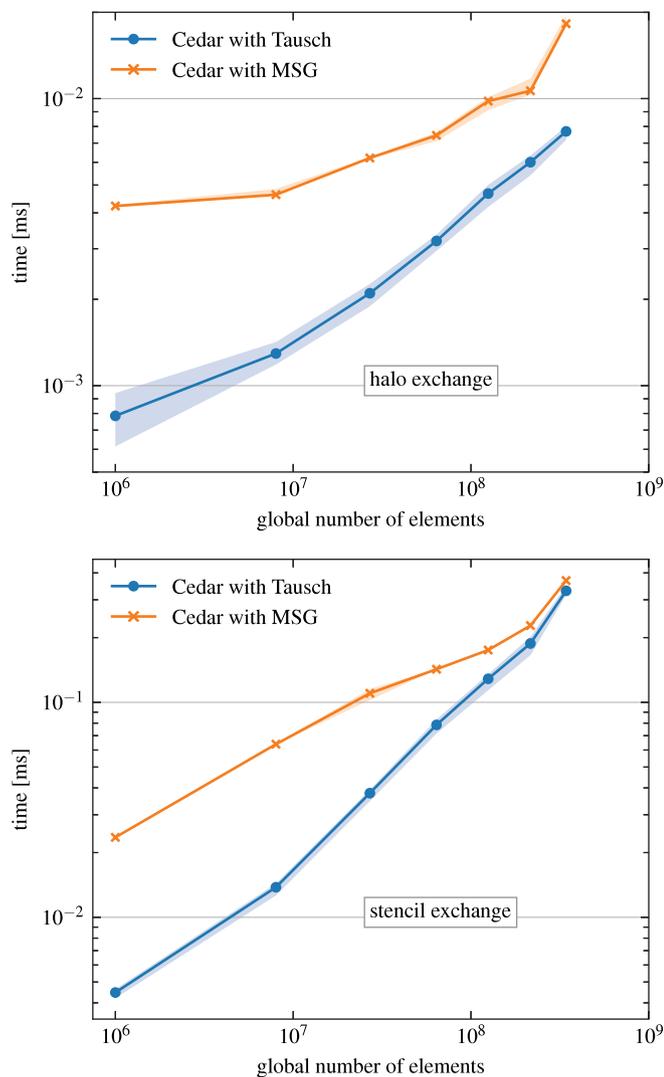
Fig. 5.3. Cedar with Tausch and MSG, 320 MPI ranks across 8 nodes on Lassen (40 ranks per node).

GPU. The only differences being the buffer pointers, and the setting of CUDA-aware MPI strategy (a single line of code) based on a compile-time macro. This highlights yet again the ease of using Tausch in a variety of codes.

5.3. Tausch in Cedar

Tausch plays a crucial role in the structure-exploiting variational multigrid library Cedar [18]. Replacing a legacy halo-exchange library (MSG [21]), Tausch provides structured communication for the solver in two and three dimensions. In addition to providing performant halo communication with predictable performance, Tausch enables parallel plane relaxation with coarse-grid problems redistributed on subcommunicators. Prohibited in the past by the legacy communication library, Tausch supports many non-interfering instances. This is used to create thousands of instances of Tausch for large 3D solves with minimal overhead [22].

Fig. 5.3 shows a comparisons of the performance of halo exchanges in Cedar when using Tausch and the previous solution MSG. The test case is a three-dimensional halo and stencil exchange across 320 MPI ranks spread across 8 nodes on Lassen (40 ranks per node). The colored regions show the range of values (min/max) across all ranks, the lines show the average timings. Tausch consistently performs better than

MSG. A major source of performance gain is the communication of the stencil operator data. MSG communicates each stencil direction in its own message, whereas Tausch is able to combine them into larger messages, as they are sent along the same communication path. Enabling Tausch to combine them into larger messages happens during setup, where different halo specifications that use the same communication paths can be tied together. Since at least around 30% of the time in Cedar is spent in communication [23], this improvement leads to significant performance gains overall.

6. Conclusion

In this paper we have introduced a new tool called *Tausch* that provides a simple API for moving halo data on heterogeneous machines. We have illustrated how its design maximizes performance while minimizing memory requirements. Measuring its performance against a performance model in three dimensions on both the CPU and GPU showed that its performance lies within expectations. Comparing its performance to the Mantevo applications HPCCG and miniFE confirmed its performance as it was able to match or outperform the applications by up to an order of magnitude. Finally, we took the framework Cedar and replaced the legacy communication library, MSG, by Tausch and achieved considerable performance gain of up to an order of magnitude. Increased flexibility in the design of Tausch also enabled scalable parallel plane relaxation in Cedar previously prohibited by the legacy communication library.

7. Resources

Tausch is hosted on GitHub and is licensed under the MIT license: https://github.com/luspi/tausch.

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.2022.102973.

Data availability

Data will be made available on request.

Acknowledgments

Appendix. BlueWaters

BlueWaters [15] was located at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana/Champaign. It had two different types of compute nodes, XE and XK. Each XE compute node had 2 AMD 6276 Interlagos CPUs (each 16 cores, 2.3 GHz operating frequency each, and 4 GB system memory) with a CPU memory bandwidth of 102.4 GB/s. Each XK compute node had a single AMD 6276 Interlagos CPU and one NVIDIA GK110 Keppler GPU (2688 cores, 1.31 TFLOPS peak performance, 6 GB memory, 250
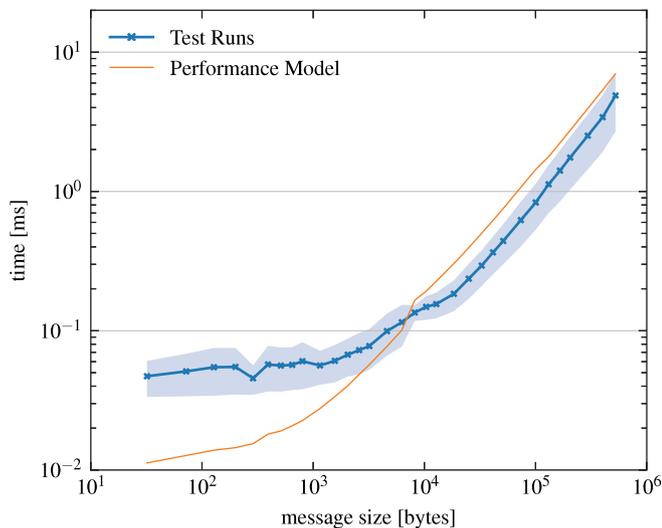
Fig. A.1. Performance Model on CPU using 256 ranks across 16 nodes on BlueWaters.

GB/s GPU memory bandwidth) with a CPU memory bandwidth of 51.2 GB/s.

The system was decommissioned at the start of 2022, but before it went offline we were able to run a simple performance analysis on its XE6 nodes (CPU-only).

The setup for the performance analysis is identical to the setup presented in Section 4. The test runs were run with a total of 256 ranks spread across 16 nodes, resulting in 16 ranks per node. Fig. A.1 shows the resulting comparison between our test runs and our model data. The colored regions show the range of values (min/max) across all ranks, the lines show the average timings. The parameters for the performance model are for inter-node communication (i.e., using the network) and for copying consecutive memory chunks (without any stride). Thus the model will be slightly too pessimistic for communication as the test runs include some on-node/on-socket communication, and slightly too optimistic for memory copies as our test case involves strided memory accesses.

For the test runs the minimum/maximum values are the fastest/slowest performance between any two ranks in the setup. Two ranks on the same node will result in better than average performance, two ranks living on two nodes far apart will result in worse than average performance.

We see that the performance model aligns very well with the data, the observed performance is very close to the expected performance.

References

[1] K.O.W. Group, The OpenCL specification, 2018, Version 1.2. URL https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf.

[2] CUDA C++ programming guide, 2019, Version 10.1. URL https://docs.nvidia.com/cuda/archive/10.1.

[3] S.R. Slattery, P.P.H. Wilson, R.P. Pawlowski, The Data Transfer Kit: A Geometric Rendezvous-Based Tool for Multiphysics Data Transfer, American Nuclear Society - ANS, 2013, URL http://inis.iaea.org/search/search.aspx?orig_q=RN:45033752.

[4] M. Bianco, An interface for halo exchange pattern, 2013, http://dx.doi.org/10.5281/zenodo.831983.

[5] Generic communication layer, 2020, Accessed: 2020-04-27. URL https://github.com/eth-cscs/gcl.

[6] RAJA performance portability layer (C++), 2020, Accessed: 2021-05-29. URL https://github.com/LLNL/RAJA.

[7] D.A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A.J. Kunen, O. Pearce, P. Robinson, B.S. Ryujin, T.R. Scogland, RAJA: Portable performance for large-scale scientific applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019, pp. 71–81, http://dx.doi.org/10.1109/P3HPC49587.2019.00012.

[8] C. Pearson, K. Wu, I.-H. Chung, J. Xiong, W.-M. Hwu, TEMPI: An interposed MPI library with a canonical representation of CUDA-aware datatypes, 2021, URL https://arxiv.org/abs/2012.14363.

[9] H.C. Edwards, C.R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216, http://dx.doi.org/10.1016/j.jpdc.2014.07.003, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. URL http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[10] C.G. Baker, M.A. Heroux, Tpetra, and the use of generic programming in scientific computing, Sci. Program. 20 (2) (2012) 115–128, http://dx.doi.org/10.1155/2012/693861.

[11] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams, K.S. Stanley, An overview of the Trilinos project, ACM Trans. Math. Software 31 (3) (2005) 397–423, http://dx.doi.org/10.1145/1089014.1089021.

[12] MiniGhost halo exchange mini-application, 2020, Accessed: 2020-04-27. URL https://github.com/Mantevo/miniGhost.

[13] R. Barrett, M. Heroux, C. Vaughan, MiniGhost : a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing, 2012, http://dx.doi.org/10.2172/1039405.

[14] Mantevo project, 2020, Accessed: 2020-04-27. URL https://mantevo.github.io.

[15] BlueWaters - system summary, 2021, Accessed: 2021-04-06. URL https://bluewaters.ncsa.illinois.edu/hardware-summary.

[16] High performance computing conjugate gradients (HPCCG), 2021, Accessed: 2021-01-08. URL https://github.com/Mantevo/HPCCG.

[17] miniFE finite element mini-application, 2022, Accessed: 2022-02-22. URL https://github.com/Mantevo/miniFE.

[18] D. Moulton, L.N. Olson, A. Reisner, Cedar framework, 2017, Version 0.1. URL https://github.com/cedar-framework/cedar.

[19] M.P.I. Forum, MPI: A message-passing interface standard, 2015, Version 3.0. URL https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[20] W. Gropp, L.N. Olson, P. Samfass, Modeling MPI communication performance on SMP nodes: Is it time to retire the ping pong test, in: Proceedings of the 23rd European MPI Users' Group Meeting, in: EuroMPI 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 41–50, http://dx.doi.org/10.1145/2966884.2966919.

[21] A. Malevsky, Message-Passing Tools for Structured Grid Communications User's Guide Version 2.0, Centre de Recherche en Calcul Appliqué, 5160 boul. Décarie, bureau 400, Montréal, Québec.

[22] A.R. Reisner, J.D. Moulton, M. Berndt, L.N. Olson, Scalable line and plane relaxation in a parallel structured multigrid solver, Parallel Comput. 100 (2020) http://dx.doi.org/10.1016/j.parco.2020.102705.

[23] A. Reisner, L.N. Olson, J.D. Moulton, Scaling structured multigrid to 500K+ cores through coarse-grid redistribution, SIAM J. Sci. Comput. 40 (4) (2018) C581–C604, http://dx.doi.org/10.1137/17M1146440, arXiv:https://doi.org/10.1137/17M1146440.