Investigating Data Movement Strategies for Distribution of Repartitioned Data

John-Paul Robinson
University of Alabama at Birmingham
Birmingham, Alabama, USA
jpr@uab.edu

Ke Fan
University of Illinois at Chicago
Chicago, Illinois, USA
kfan23@uic.edu

Steve Petruzza
Utah State University
Logan, Utah, USA
steve.petruzza@usu.edu

Thomas Gilray
University of Alabama at Birmingham
Birmingham, Alabama, USA
gilray@uab.edu

Sidharth Kumar
University of Illinois at Chicago
Chicago, Illinois, USA
sidharth@uic.edu

ABSTRACT

Repartitioning in a parallel setting can be defined as the task of redistributing data across processes based on a newly imposed grid/layout. Repartitioning is a fundamental problem, with applications in domains that typically involve computation on tiles (blocks/patches) of varying resolution, for example, while creating multi-resolution data formats in in situ mode (such as the JPEG format and its variants). This paper explores the performance and tradeoffs of different ways to perform the data redistribution phase. In particular, we explore a greedy scheme that aims to minimize data movement while compromising on load balancing and a balanced scheme that aims to create a balanced load across processes while compromising on data movement. For both these schemes, we measure the impact of buffer size on MPI point-to-point communication performance when using two different communication patterns: a per-patch (staggered data transfer) and a per-rank (aggregated data transfer). Our experimental study finds that the reduced data movement of the greedy scheme leads to reduced transfer times during redistribution. Furthermore, we conclude that the per-patch communication pattern outperforms per-rank communication.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**.

KEYWORDS

data movement, data layout, load-balancing

ACM Reference Format:
John-Paul Robinson, Ke Fan, Steve Petruzza, Thomas Gilray, and Sidharth Kumar. 2024. Investigating Data Movement Strategies for Distribution of Repartitioned Data. In *Practice and Experience in Advanced Research Computing (PEARC '24), July 21–25, 2024, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3626203.3670534

PEARC '24, July 21–25, 2024, Providence, RI, USA

ACM ISBN 979-8-4007-0419-2/24/07
https://doi.org/10.1145/3626203.3670534

1 INTRODUCTION

This paper explores the process of repartitioning, which entails the task of redistributing existing data across the available set of processes based on a new grid layout [12]. The process of repartitioning can be used by any parallel code that performs computational tasks with different data distribution requirements. In particular, it can be used for in-situ tasks where the division of data leveraged by the application is not in sync with the division of data needed by the in-situ operation. For example, an image is a collection of pixels arranged on a regular grid indexed by its global coordinate system. An in-situ operation on the image, like compression, may be applied only in units of a regular, well-defined extent of pixels known as a patch. A patch is a slice- or volume-based subdivision of the image pixels that is transformed by an in-situ operation, e.g. a patch size of 16×16 pixels. Repartitioning is used to support computational operations on patches that represent logical units of data aligned with the algorithmic requirements of a computation. Repartitioning facilitates operations like wavelet convolutions [4], low-pass filtering and image denoising [1], lossy compression [13, 16, 19], efficient I/O [12], and other downstream computations. While repartitioning can be part of many different workflows, our motivation behind studying it in detail stems from its applicability in parallel I/O pipelines, in particular, while writing data in hierarchical format as with JPEG [16] or IDX [12]. Both these data formats usually work on patches of resolution 16^3 or 32^3 or 64^3 and therefore require repartitioning the data domain based on that patch size.

A simple example of repartitioning can be seen in Figure 1. The left sub-figure shows a 2D data domain of resolution 300×200 partitioned across 6 processes (shown by 6 different colors), where every process works on a patch of resolution 100×100. The right sub-figure shows the result after repartitioning the data domain into patches of size 75×75. In this example, since the total number of patches after repartitioning (12) is an integral multiple of the initial number of patches (6), every process gets 2 patches after repartitioning. However, the global domain resolution in the Y-axis (200) is not an integral multiple of the patch size in the Y-dimension (75), therefore a couple of processes end up having patches of smaller sizes (brown and grey process).

Non-uniform distribution of patches can lead to an imbalance in the work performed across processes which leads to communication delays and idle resources as processors with less work wait on those with more work to complete their operations before the next phase

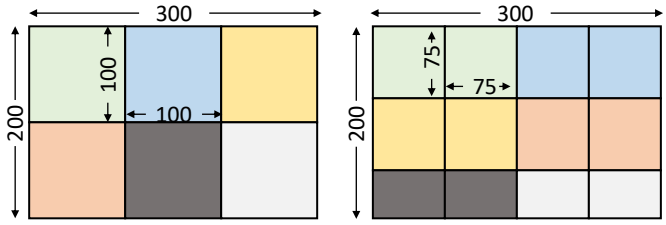


Figure 1: Example of *repartitioning*. (Left) a 2D domain of 300×200 divided across patches of resolution 100×100, (right) the same 2D domain after repartitioning across patches of resolution 75×75.

of a synchronized workflow can begin. Uniform distribution of data to balance the workload ensures optimal forward progress of the workflow [9]. This work explores the data movement costs needed to maintain a balanced workload.

Applications can pursue many different approaches to distributing the new patches across the available set of processes. For example, the scheme shown in Figure 1 follows the round-robin approach where the newly created 12 patches are distributed to the 6 processes in increasing patch-id order. Clearly, this scheme does not lead to optimal load balance as both the brown and gray processes end up with two patches of a smaller size. A load-balanced redistribution scheme would instead ensure allotment of the 4 small-sized patches to 4 different processes.

In this paper, we explore two main patch distribution approaches using MPI point-to-point communication: (a) a greedy approach that aims to minimize data movement across processes, and thereby compromises on load-balancing and (b) a lowest-rank patch placement approach that aims to achieve a balanced load across processes and thus can potentially compromise data movement costs. For both these approaches, we have two implementation strategies: (i) per-rank data exchange which ensures only one point-to-point data exchange between a process pair, and (ii) per-patch exchange, where a process pair has as many point-to-point exchanges as the total number of patches.

In brief, the contributions made by our work are:

- experimental study of the communication costs of a balanced data placement versus greedy placement
- experimental study of the impact of per-patch data exchange using non-contiguous MPI subarry buffer packing versus a per-rank data exchange using contiguous buffers
- scaling study of the proposed repartitioning strategies.

2 RELATED WORK

Existing parallel I/O performance libraries like ADIOS [11] and HDF5 [10] provide abstractions for managed file I/O across process. These frameworks are not focused on data exchange costs leading to the file I/O. Prior studies have explored data repartitioning and parallel I/O workflow performance from an application perspective [12] and included characteristic communication and computation operations that leverage these parallel I/O libraries to measure the throughput benefits of balanced patch [9] or particle [18] distribution. Particle-based workload distribution using task run-time to balance load across processors [3, 20] also demonstrate improved workflow performance. These studies do not isolate the cost of data communication from the total workflow runtime and do not explore the potential impact of large-scale data movement.

Existing benchmarks available for file I/O, like IOR [15], support platform-specific performance assessments of throughput and can serve to isolate the storage performance in parallel I/O workflows. MPI communication benchmarks provide mechanisms to explore inter-process communication costs for specific scenarios. Benchmarks to explore overlapped communication and computation using point-to-point [7] and partitioned communication [17] focus on specific application communication patterns and not the timed exchange of application specific data abstractions. As with file I/O, there are MPI benchmarks, like the Sandia [5] and ACLF [14] benchmarks, to explore the performance of underlying network hardware, but do not present the measures from an application perspective portable across systems.

We orient our micro-benchmark development toward measures of data exchange specific to repartitioned patch exchange with a perspective toward scaled scientific application data sets. We therefore explore large scale patch data exchange to better understand performance characteristics and application requirements required for such a micro-benchmark.

3 DATA REPARTITIONING

Data repartitioning is comprised of two sub-phases: (a) patch-to-rank assignment (Section 3.1), and, (b) inter-process data communication (Section 3.2). The patch-to-rank phase indexes all patches in a newly defined patch grid resulting from the repartitioned global data set using fixed-size patches and determines the rank assignment for each patch. The inter-process data communication uses MPI's non-blocking, point-to-point API for the data transfers that move patches to their assigned rank.

In this section, we present the implementation details of both sub-phases. For simplicity, we base our exposition around a 2D example shown in Figure 2. The initial state shown in Figure 2, comprises a 2×2 process grid (process ids: P0, P1, P2, and P3) with data uniformly distributed across four processes. The processes are further color-coded green, blue, pink, and yellow to allow process tracking during later patch assignment, which drop the process labels in favor of patch labels based on the patch grid imposed by repartitioning. In this example, the goal is to repartition the data into a 3×3 grid of patches (patch ids: p0, p1, . . . , p8) of smaller size. To present a more realistic scenario, we show a patch size that does not align with the original data dimensions. Newly defined patches that overlap the boundaries of the original data (p2, p5, p6, p7, p8) are truncated to the boundaries of the original data. The placement algorithm assigns the patches to the 4 processes during patch-to-rank assignment. The patch data is moved to the assigned process during inter-process communication.

3.1 Patch-to-rank assignment

This paper focuses on uniform-resolution data sets. The newly imposed patch grid resulting from repartioning the original data set

comprises fixed-size patches, except for patches that intersect the boundaries of the original data dimension as described above. Refer to the example of a newly imposed grid of patches patch shown in Figure 2.

Once the new patch layout is imposed, the main task is to optimally assign each of the newly created patches to a process (rank). The patch-to-rank assignment is then followed by inter-process point-to-point communication to move the relevant patch data to each rank. The patch-to-rank assignment algorithm directly impacts data movement costs and overall load balance across all processes.

The main challenge in selecting a rank to which the patch is assigned is considering how to deal with patches that are shared by more than one process, a natural consequence of the patch size defining a patch grid that does not align with the original data process boundaries. We distinguish patches based on this criterion, classifying them into two categories: fully-contained patches *(FCP)* and shared-patches *(SP)*. A *FCP* is fully contained within a process before distribution. In Figure 2, p0, p2, p6 and p8 are *FCPs*. A *SP* is shared by more than one process (i.e., each process holds some portion of the patch). In our example, p1, p3, p4, p5, and p7 are *SPs*, with p4 being shared by all four processes. We minimize data movement costs by enforcing the *FCPs* to stay within their parent rank. For example, data in p0 continues to stay with the blue rank and will not be transferred.

SPs on the other hand, need to have the sub-parts that exist across multiple ranks moved to a common rank to form a complete patch on the assigned rank. This introduces the need for a patch assignment algorithm that determines the assignment of patches to ranks. We consider and compare two approaches: (i) a *balanced placement* and (ii) a *greedy placement* algorithm to determine the impact on communication cost.

3.1.1 Balanced placement . The balanced placement algorithm seeks to maintain an even distribution of patches to maintain computational workload balance for subsequent patch computations [9]. The FCPs are not moved and are thus assigned to their original parent rank. The SPs, instead, will each be assigned to a target rank.

In Figure 2 the balanced patch placement shows the the target patch counts for processes P0-P3 (colored green, blue, pink, and yellow to highlight patch to process assignment) are 3, 2, 2 and 2. Formally, balanced patch placement for a total of M patches distributed across N processes is ensured when every process gets exactly $\lfloor M/N \rfloor$ patches and the remaining $M\%N$ patches are spread out uniformly across the N processes. A process will therefore either hold $\lfloor M/N \rfloor$ or $\lfloor M/N + 1 \rfloor$ patches, which we call the target patch count. To minimize data movement, we attempt to assign a shared-patch to one of the processes that already share that patch-- we choose the process that has currently been assigned fewer patches than its target patch count. If there are multiple candidate, the process with the smallest rank is chosen. If an assigned process for the shared-patch is found this way, then its chunk of the shared-patch will be locally copied (instead of being sent across the network), hence reducing data movement. Alternatively, if all processes that contain a part of the shared-patch have already reached their target patch count, we scan through all processes and assign the patch to the first process that has not reached its target patch count.

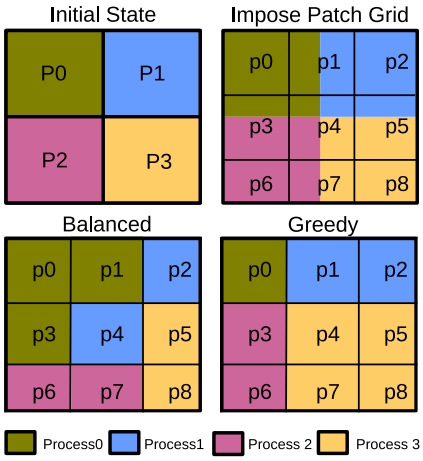


Figure 2: Patch Layout and Distribution: the *Initial State* image tile shows the global data set evenly distributed across four processes, labeled P0-3 and color-coded to highlight patch spanning and placement in later steps. The *Imposed Patch Grid* tile shows the Initial State with a patch division logically imposed across the data set indicating that the new boundaries lead to logical patches now shared across multiple ranks, patches labeled p0-8. Note the *Imposed Patch Grid* leads to patches p3, p5, p6-8 overlapping the original data set boundaries because patch size does not evenly divide the data size. The boundary of these patches is truncated to align with the original data boundaries. The newly defined logical patches need to be physically arranged as undivided physical patches assigned to specific ranks for further computation. The *Balanced* and *Greedy* image tiles highlight the two patch distribution schemes explored leading to even and uneven patch data distribution for subsequent compute stages.

3.1.2 Greedy placement. The greedy placement algorithm dispenses with any attempt to maintain a balanced patch distribution. Its goal is to minimize data movement, therefore, it assigns patches to the rank that contains the largest shared subpart of the patch. If there are multiple candidates, the process with the smallest rank is chosen. The assigned process will have its chunk of the shared-patch locally copied (instead of being sent across the network), further optimizing for minimal data movement.

In the greedy strategy, we do not use any target patch count and in the worst case a process could be receive all the patches shared with its neighbors. It is worth noting, that in the greedy approach, a patch will never be assigned to a rank that does not already contain a portion of the patch.

Patch distribution using this greedy scheme has been used, for example in [12], albeit not for balancing the data transformation per patch but to minimize interleaving of data samples among processes in aggregation buffers. In Figure 2 we show that the greedy scheme can lead to a very imbalanced patch distribution. Here, the greedy assignment results in a distribution of 1, 2, 2, and 4 patches for processes P0, P1, P2, and P4, respectively. In Section 5 we show through experiments that the greedy scheme can, however,

result in an optimal data transfer times. This trade-off between minimized data transfer times and balanced patch workloads needs to be assessed based on workflow throughput goals due to the potential for significantly longer computation times for imbalanced workloads as shown in [9].

3.2 Inter-process data communication

Once a target rank is assigned to every patch by the chosen placement algorithm, we initiate inter-process data communication to appropriately move patches to their target rank. We divide the set of processes associated with a patch into senders and receivers. The receiver is the one process to which the patch is assigned. The rest are senders. In the case of a *FCP*, the sender and receiver are the same rank. In the case of a *SP*, each sender sends the region of the patch that it holds to the receiver.

The patch-to-rank assignment algorithm is run concurrently on every process, ensuring that every process knows exactly whether it is a receiver or sender for a patch and what data it is expected to send or receive. For the actual inter-process data communication to distribute patches to their assigned ranks, we have implemented two different schemes: *per-patch* communication and *per-rank* communication. These schemes explore the impact of buffer size on communication overhead to determine if the per-patch communication introduces inefficiencies in data transfer that can be alleviated by reducing the frequency of inter-process communication events through explicitly prepared per-rank buffers.

3.2.1 Per-patch communication. Per-patch communication interfaces with the MPI point-to-point communication calls at the level of patches. The algorithm iterates over the patches that are to be received and sent by the rank and issues MPI non-blocking receives and sends as each patch is processed. For per-patch communication, the MPI_Type_create_subarray is used to define the non-contiguous sub-patch regions from the original data set that are sent and received over the network. For per-patch communication, we issue the full batch of MPI_Irecv calls for each patch or patch portion of a shared patch assigned to the local rank. This creates a pending MPI request for each patch that will be satisfied by the corresponding MPI_Isend calls of the senders of the corresponding patch data. After all the sub-patch regions are sent and received, every process ends up with their assigned patches, each of which is stored in a separate contiguous memory block, ready for subsequent computational steps. See Figure 2 and earlier discussion of the exhibition example with numbered patches and colored processes to help to visualize the data exchanges resulting from patch assignment.

3.2.2 Per-rank communication. Per-rank communication interfaces with the MPI point-to-point communication calls at the level of manually-packed, per-rank patch buffers. The algorithm iterates over the patches that are to be received by the rank and creates a buffer sized to accept all patch data expected from a specified sender rank. It then issues a non-block MPI_Irecv call for each rank's receive buffer. It then iterates over the patches to be sent by the rank and manually packs the non-contiguous patch data into a per-rank send buffer that will contains all patch data exchanged with a specific rank. As with the receive phase, no MPI communication calls are issued until after the buffer packing phase is complete. After the buffers are packed, the algorithm issues a non-blocking MPI_Irecv call for rank buffer. This reduces the pending MPI request set to one send and one receive request per rank pair.

Note that the MPI_Type_create_subarray calls used to simplify patch data collection for per-patch communication cannot be used for per-rank communication because the aggregate patch structure no longer has a regular structure needed by this call. This requires development of explicit steps to manual gather non-contiguous patch data from the data set and pack it into per-rank buffers, representing additional developer effort to implement buffer packing.

To measure the MPI communication cost for both per-patch and per-rank communication, we initialize the MPI communication timer at the beginning of the the receive and send request generation phase but after the parallel execution of the placement algorithm. Therefore, the timed executions include the time it takes to gather the patch data into buffers to send to peer processes, either implicitily with per-patch or explicitly with per-rank. Once all MPI non-blocking requests have been issued, the ranks enter an MPI_Waitall barrier to block the ranks until all data exchange has completed. We record the end time on the return from the wait call and take the total MPI communication time as the difference between the end time and the time recorded at timer initialization. We record this delta as the total MPI communication time for each rank. In both scenarios, we record the time of the MPI communication as the time of the slowest rank, since the processes will not continue until the slowest rank completes communication.

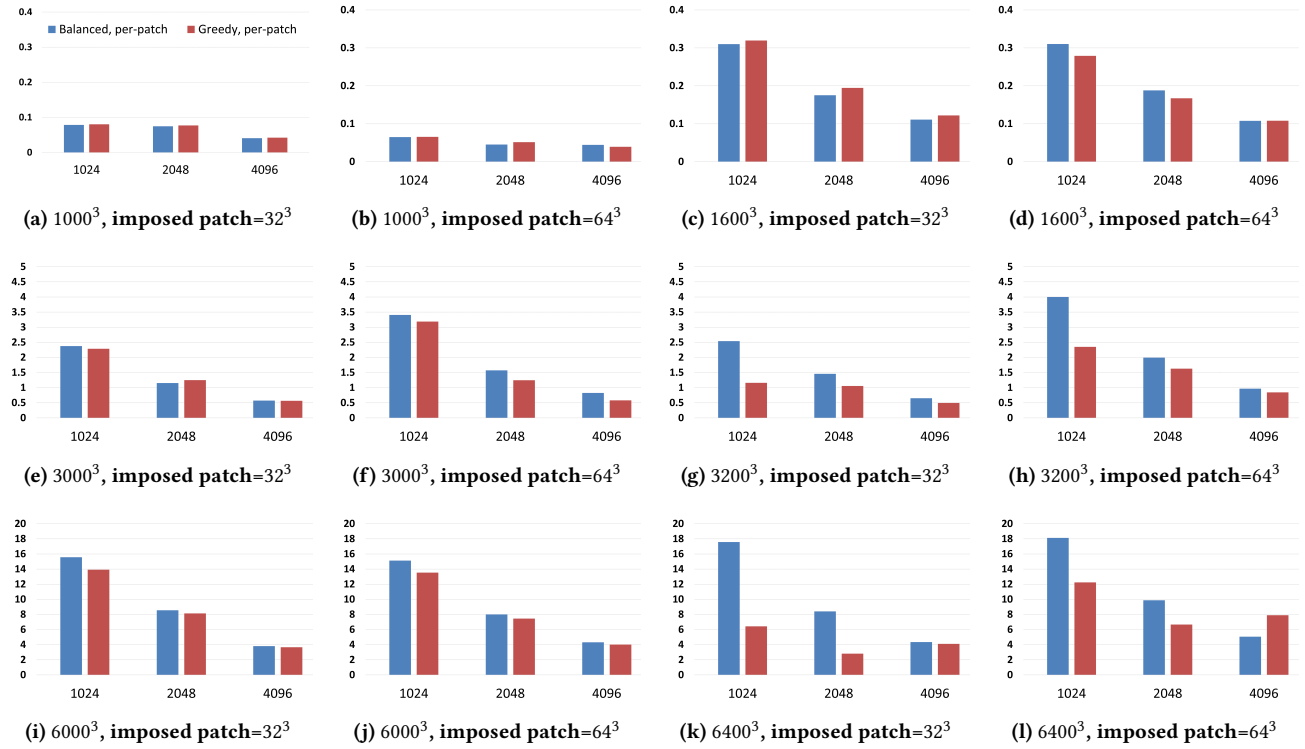
4 EMPIRICAL EVALUATION

4.1 HPC platform

All our experiments are performed on the Theta Supercomputer [8] at the Argonne Leadership Computing Facility(ALCF). Theta is an Intel-Cray XC40 with a peak performance of 11.69 petaflops, 281,088 compute cores, 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM, and 10PiB of online disk storage. The supercomputer has a Dragonfly network topology [2] and a Lustre filesystem. We do not investigate specific performance impacts of non-local communication patterns of the Dragonfly network outside the aggregate communication performance of application workload. The current version of our micro-benchmark does not address system-to-device data transfer performance of GPU based workloads. The focus of this study is exclusively on inter-process communication costs for CPU workloads where each MPI rank is associated with a single processor core. As such, measured data exchange performance does not differentiate between intra-CPU, intra-node, or other aspects of the cluster network topology.

4.2 Experiment setup

We evaluate the performance of our four data repartitioning strategies: (a) greedy, per-rank, (b) greedy, per-patch, (c) balanced, per-rank, and, (d) balanced, per-patch. We measure the performance of the four strategies by recording their time to completion for MPI point-to-point communication. We evaluate the performance of the four strategies using synthetic micro-benchmarks, that simulate real-world application scenarios. The benchmark operates in two phases, data generation, where each process generates data for the rank's portion of the global data set it holds (see P0, P1, P2 and P3



(a) 1000^3, **imposed patch**=32^3 **(b)** 1000^3, **imposed patch**=64^3 **(c)** 1600^3, **imposed patch**=32^3 **(d)** 1600^3, **imposed patch**=64^3

(e) 3000^3, **imposed patch**=32^3 **(f)** 3000^3, **imposed patch**=64^3 **(g)** 3200^3, **imposed patch**=32^3 **(h)** 3200^3, **imposed patch**=64^3

(i) 6000^3, **imposed patch**=32^3 **(j)** 6000^3, **imposed patch**=64^3 **(k)** 6400^3, **imposed patch**=32^3 **(l)** 6400^3, **imposed patch**=64^3

Figure 3: Timing for repartitioning for different workloads. Blue bar is the balanced, per-patch scheme and red bar is for greedy, per-patch. X-axis for all graphs shows three process counts, 1024, 2048 and 4096. Y-axis for the graphs is time in seconds.

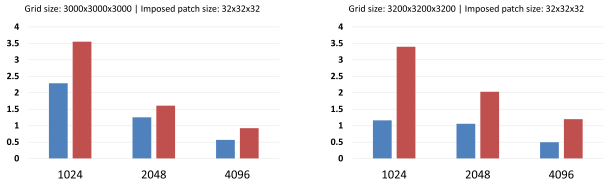
held by ranks green, blue, pink, and yellow in Figure 2) followed by the data redistribution phase. Any subsequent phases of real application workloads would normally exist after data redistribution, for example, wavelet transformation and compression phases documented in earlier work [9, 12]. This investigation, ignores these down stream phases to focus exclusively on the data redistribution behavior and performance.

The data generation phase includes allocation of the rank's portion of the global data set, the calculations to determine how the global data set is divided into patches, and finally execution of the patch placement algorithm, that determines the patch assignments for distribution of patches to the ranks in phase two. In all our experiments we *only* measure the communication time that occurs during the data redistribution phase. Time to completion is our only performance metric. This metric only depends on the nature of data transfers (amount and pattern), rather than the nature of the data itself -- making synthetic benchmarks suitable, as they can be easily modeled to generate different workloads and patch sizes.

We perform a series of scaling experiments with varying workloads and patch sizes. We use patch resolutions of $32 \times 32 \times 32$ and $64 \times 64 \times 64$. We keep the patch resolution to be powers of two. This is characteristic of common wavelet sizes used in image processing to ensure that the patch boundaries are computed correctly. We ran scaling tests at different problem sizes. We used six global resolutions 1600^3, 3200^3, 6400^3,1000^3, 3000^3, and 6000^3. The first three correspond to the case when the global resolution and the newly imposed patch grid align, and the last three correspond to the case where the global resolution does not align with the newly imposed patch grid. We use 8-byte (64-bit) data types for all tests so that patches are either 0.25 Megabytes(MB) for 32^3 patches or 2MB for 64^3 patch sizes. The global data set sizes range from ~ 8Gigabytes (GB) at 1000^3 resolution to ~ 2Terabytes (TB) at 6400^3 resolution. With 6 global resolutions and 2 patch size settings, we have 12 sets of experiments. For each of these 12 experiment sets, we vary the process counts across 1024, 2048 and 4096 (strong scaling experiments). We ran all our experiments for 50 iterations, and have plotted the median.

5 RESULTS

Figure 3 shows the results for balanced and greedy placement with per-patch communication for all 12 experiments. We observe the expected benefits of strong scaling as process counts increase and the data exchange burden of individual tasks diminishes. These two methods perform similarly for global data sets sizes below 3200^3 (i.e. data sets below approximately 250GB). This confirms that using balanced patch placement to facilitate workload balance for downstream tasks is an effective strategy for such data sets. [9] Greedy patch placement outperforms balanced placement as the global resolution increases. While this is a somewhat intuitive result, since greedy placement explicitly favors less data movement,



(a) 3000^3**, imposed patch**=32^3 **(b)** 3200^3**, imposed patch**=32^3

Figure 4: Per-patch vs. Per-rank comm. Blue is per-patch, red is per-rank. Per-patch consistently outperforms per-rank. Time on Y-axis is in seconds.

it demonstrates that data movement impacts on application performance may not be measurable unless data sets of significant size are included in the test scenarios.

It also suggests opportunities for heterogeneous, multi-core environments to pursue an early start on downstream compute tasks for FCPs while SPs are exchanged. FCP are immediately available for local computation without the need for data transfer between ranks. This approach could offset the generally higher workload imbalance that results from greedy placement by reducing the total amount of compute (FCP + SP) remaining after all SPs have been received .

We observe an unexpected inversion of the strong scaling performance pattern at the largest global resolution (2TB) and process count (4096) as seen in Figure 3l, which suggests the potential for unexpected behaviors as global resolution increase. To investigate potential causes of unexpected behaviors at scale, we plot per-rank MPI communication performance using a "box plot per rank" layout that summarizes all sampled runs for each rank, see Figure 5. This has proved a useful debugging tool that enables observation of characteristic performance across the entire collection of MPI processes. This figure includes characteristic performance plots for both balanced and greedy assignment runs as summarized in Figure 3l. It is clear from the greedy plot that there are a set of ranks that consistently report performance well below (slower than) the remaining ranks causing them to dominate the slowest performer metric used to record test performance. We were unable to uncover and resolve the root cause of this anomaly, these ranks do not carry an undue burden of communication nor are they assigned more patches than any other ranks. We suspect an algorithmic error in our micro-benchmark that manifests at this scale in our test. While an unsatisfying result, we note that, in the absence of these rank outliers, the performance trend of the well-behaving ranks for greedy placement is in line with the performance of the results for balanced placement results. This is consistent with our other reported 4096 rank results.

This error represents a characteristic challenge of scaling benchmarks to model emerging large scale data sets, on the way to Exascale. There were several improvements necessary to counters and memory allocations in our micro-benchmark to handle our largest (2TB) tested data set, as prior versions had only been used up to 30GB data sets. Additionally, MPI processes had to be scheduled at half- and quarter node capacity to accommodate per-node memory limits on Theta. Further improvements to benchmark memory management are warranted.

For inter-process communication, we observed that the per-patch communication scheme consistently outperformed the per-rank communication scheme. We highlight this result in Figure 4 for two data set sizes of 3000^3 and 6000^3 for the imposed patch size of 32^3. Our interpretation of the performance advantage for per-patch communication is that the MPI subarray derived data type is an efficient mechanism for gathering non-contiguous data for exchange with other processes. This implementation is able to outperform our basic per-rank communication because the initial per-patch MPI requests are ready for transfer as the algorithm continues to process later patches. This enables MPI to start patch exchange while other patches are gathered (as memory is read). In contrast, our naive per-rank buffer model delays the start of MPI patch exchange until buffer construction is complete (after having read through all memory). Our per-patch and per-rank experiments measure two extremes of communication patterns and clearly demonstrate that implementations that enable an early start to their communication by exchanging smaller buffers can have a significant impact on their overall performance. A more subtle impact of explicitly managed buffers is that they directly compete with the application for scarce memory resources, introducing additional memory management overheads, which complicates and limits scaling to very large data sets, especially for the smaller rank counts where RAM-per-node is limited relative to the data set.

In summary, these results show that larger data sets are necessary to successfully measure the impact of data movement on task completion times. It was only for data sets at $3200^3 \times 8bytes = 244GB$ and above where we observed a significant difference in greedy versus balanced placement. This demonstrates that future performance assessments of placement strategies need to include large scale data sets to get a comprehensive understanding of the performance benefits of different methods. The results further suggest that performance differences may also relate to the rank count and associated per-rank patch processing overhead. The patches-per-rank count drops under strong scaling scenarios and limits the per-rank data exchange effort. This appears to impact the uniformity of measured performance under strong scaling scenarios. This observation warrants further study. Finally, while the per-patch and per-rank communication tests showed per-patch communication as universally superior, we suspect there is potential for tuning the number of patches included in MPI_Isend and MPI_Irecv calls to explore if there is a benefit to batching more than one patch and less than all patches in a single communication transaction. More advanced communication implementations are needed to fully explore this scenario, possibly using MPI partitioned communications.

6 FUTURE WORK

An aspect of the per-rank implementation that was not explored in this work was resize-able send buffers. That is, we either had per-patch communication with an MPI Isend & Irecv for each patch or a per-rank communication that included all the patches destined for a neighboring process. It is possible that tuning the per-rank implementation to issue MPI requests after set amounts of patches

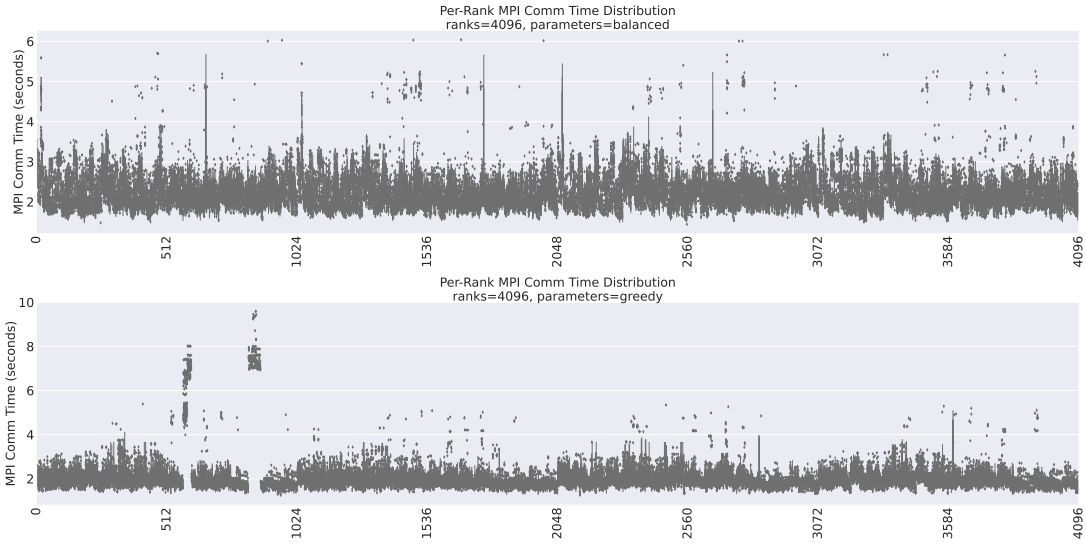


Figure 5: Plot of per rank MPI communication times for balanced (top) and greedy (bottom) patch-to-rank assignment. The data points are per-rank box plots for each of the 50 samples for global size 6400^3, patch size 64^3 and rank count 4096. The greedy placement plots shows that specific ranks contribute to the unexpected loss of strong scaling for the greedy assignment highlighted in Figure 3l. Balanced assignment included for comparison and discussion.

are gathered could improve per-rank performance. MPI partitioned communication to leverage threaded communicators is an area of future research interest. Additionally, while the communication times generally decrease as process counts increase, there are lots of subtle variation between the methods that make it hard to define specific heuristics for performance. This is likely further complicated by the variable compute demand of different application workflows. We plan to investigate the integration of ML algorithms [6] to improve the placement methods and communication parameters that would allow automatic tuning of performance for specific application needs and potentially better account for unexpected behaviors at scale.

ACKNOWLEDGMENTS

This work was partly funded by NSF Collaborative Research Awards 2401274 and 2221812, and NSF PPoSS Planning and Large awards 2217036 and 2316157. We thank the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on Theta supercomputer located at the Argonne National Laboratory.

REFERENCES

[1] Monagi H. Alkinani and Mahmoud R. El-Sakka. 2017. Patch-based models and algorithms for image denoising: a comparative review between patch-based images denoising methods for additive noise reduction. *EURASIP Journal on Image and Video Processing* 2017, 1 (Aug. 2017). https://doi.org/10.1186/s13640-017-0203-4

[2] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. 2012. Cray XC series network. *Cray Inc., White Paper WP-Aries01-1112* (2012).

[3] Samuel J. Araki and Robert S. Martin. 2022. Dynamic load balancing with over decomposition in plasma plume simulations. *J. Parallel and Distrib. Comput.* 163 (may 2022), 136–146. https://doi.org/10.1016/j.jpdc.2022.01.023

[4] A. Averbuch, D. Lazar, and M. Israeli. 1996. Image compression using wavelet transform and multiresolution decomposition. *IEEE Transactions on Image Processing* 5, 1 (Jan. 1996), 4–15. https://doi.org/10.1109/83.481666

[5] Brian W. Barrett and K. Scott Hemmert. 2009. An application based MPI message throughput benchmark. In *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE. https://doi.org/10.1109/clustr.2009.5289198

[6] Riccardo Cantini, Fabrizio Marozzo, Alessio Orsino, Domenico Talia, Paolo Trunfio, Rosa M. Badia, Jorge Ejarque, and Fernando Vazquez. 2022. Block size estimation for data partitioning in HPC applications using machine learning techniques. (Nov. 2022). arXiv:2211.10819 [cs.DC]

[7] Alexandre Denis and Francois Trahay. 2016. MPI Overlap: Benchmark and Analysis. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE. https://doi.org/10.1109/icpp.2016.37

[8] Fahey et al. 2019. Theta and Mira at Argonne National Laboratory. In *Contemporary High Performance Computing*. CRC Press, 31–61. https://doi.org/10.1201/9781351036863-2

[9] Haoyi Fan, Fengbin Zhang, Yuxuan Wei, Zuoyong Li, Changqing Zou, Yue Gao, and Qionghai Dai. 2021. Heterogeneous Hypergraph Variational Autoencoder for Link Prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), 1–1. https://doi.org/10.1109/tpami.2021.3059313

[10] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases (EDBT/ICDT '11)*. ACM. https://doi.org/10.1145/1966895.1966900

[11] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (July 2020), 100561. https://doi.org/10.1016/j.softx.2020.100561

[12] Sidharth Kumar, Venkatram Vishwanath, Philip Carns, Joshua A. Levine, Robert Latham, Giorgio Scorzelli, Hemanth Kolla, Ray Grout, Robert Ross, Michael E. Papka, Jacqueline Chen, and Valerio Pascucci. 2012. Efficient data restructuring and aggregation for I/O acceleration in PIDX. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. https://doi.org/10.1109/sc.2012.54

[13] Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (Dec. 2014), 2674–2683. https://doi.org/10.1109/tvcg.2014.2346458

[14] Vitali Morozov, Jiayuan Meng, Venkatram Vishwanath, Jeff R. Hammond, Kalyan Kumaran, and Michael E. Papka. 2012. ALCF MPI Benchmarks: Understanding Machine-Specific Communication Behavior. In *2012 41st International Conference on Parallel Processing Workshops*. IEEE. https://doi.org/10.1109/icppw.2012.7

[15] Hongzhang Shan, Katie Antypas, and John Shalf. 2008. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic

benchmark. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. https://doi.org/10.1109/sc.2008.5222721

[16] David S. Taubman. 2002. JPEG2000: Image Compression Fundamentals, Standards and Practice. *Journal of Electronic Imaging* 11, 2 (apr 2002), 286. https://doi.org/10.1117/1.1469618

[17] Yiltan Hassan Temucin, Ryan E. Grant, and Ahmad Afsahi. 2022. Micro-Benchmarking MPI Partitioned Point-to-Point Communication. In *Proceedings of the 51st International Conference on Parallel Processing*. ACM. https://doi.org/10.1145/3545008.3545088

[18] Will Usher, Xuan Huang, Steve Petruzza, Sidharth Kumar, Stuart R. Slattery, Sam T. Reeve, Feng Wang, Chris R. Johnson, and Valerio Pascucci. 2021. Adaptive Spatially Aware I/O for Multiresolution Particle Data Layouts. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. https://doi.org/10.1109/ipdps49936.2021.00063

[19] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE. https://doi.org/10.1109/icde51399.2021.00145

[20] Guixun Zhu, Jason Hughes, Siming Zheng, and Deborah Greaves. 2023. A novel MPI-based parallel smoothed particle hydrodynamics framework with dynamic load balancing for free surface flow. *Computer Physics Communications* 284 (mar 2023), 108608. https://doi.org/10.1016/j.cpc.2022.108608