Rapid Tomographic Image Reconstruction via Large-Scale Parallelization

Tekin Bicer[1], Doga Gursoy[1], Rajkumar Kettimuthu[1,4], Francesco De Carlo[1], Gagan Agrawal[2], and Ian T. Foster[1,3,4]

[1] Argonne National Laboratory, USA
Mathematics and Computer Science Division X-Ray Science Division
{bicer, kettimut, foster}@anl.gov {dgursoy, decarlo}@aps.anl.gov
[2] Dept. of Computer Science and Engineering, Ohio State University, USA
agrawal@cse.ohio-state.edu
[3] Dept. of Computer Science, University of Chicago, USA
[4] Computation Institute, Uni. of Chicago and Argonne National Lab., USA

Abstract. Synchrotron (x-ray) light sources permit investigation of the structure of matter at extremely small length and time scales. Advances in detector technologies enable increasingly complex experiments and more rapid data acquisition. However, analysis of the resulting data then becomes a bottleneck—preventing near-real-time error detection or experiment steering. We present here methods that leverage highly parallel computers to improve the performance of iterative tomographic image reconstruction applications. We apply these methods to the conventional per-slice parallelization approach and use them to implement a novel in-slice approach that can use many more processors. To address programmability, we implement the introduced methods in high-performance MapReduce-like computing middleware, which is further optimized for reconstruction operations. Experiments with four reconstruction algorithms and two large datasets show that our methods can scale up to 8K cores on an IBM BG/Q supercomputer with almost perfect speedup and can reduce total reconstruction times for large datasets by more than 95.4% on 32K cores relative to 1K cores. Moreover, the average reconstruction times are improved from ∼2 hours (256 cores) to ∼1 minute (32K cores), thus enabling near-real-time use.

1 Introduction

As data volumes increase, research success in a growing number of fields depends on the ability to analyze the data rapidly. In scientific computing, this situation is true both for data generated by simulations and instruments. In the context of scientific instruments, for instance, techniques such as time-resolved micro-tomography can produce three or more dimensional data at rates of terabytes per day or more. Moreover, data generation rates are expected to increase with advances in detector technologies and experimental techniques.

The utility of such techniques is severely limited by the hours required to analyze the resulting large datasets [3, 11]. Scientists often want *quasi-instant* feedback so that they can check results and adjust the experimental setup. For instance, the x-ray tomography systems available at the imaging beamlines of the

Advanced Photon Source (APS, located at Argonne National Laboratory) are routinely used in materials science applications, where high-resolution and fast 3D imaging are instrumental in extracting valuable information. Quasi-instant feedback can help identify optimal experimental parameters (beamline condition and sample environment such as temperature and pressure) and accelerate the end-to-end scientific process.

In the absence of such quasi-real-time analysis, precious time on many expensive instruments is used less effectively. In this paper, we focus on improving the efficiency of tomographic image reconstruction, thus enhancing the turnaround time of scientific workflows. Specifically, we address the following issues: (1) how to enable efficient and parallel execution of tomographic reconstruction algorithms and (2) how to ease the rapid development of reconstruction codes.

With regard to the first issue, different parallel reconstruction algorithms have been proposed for multicore machines [1, 14, 17, 23, 24, 26]. Although these works provide reasonable reconstruction times with small datasets, they typically have scalability limitations and are not suitable for high-resolution large datasets such as those generated at synchrotron x-ray light sources (e.g., APS). Another effort in the same direction is to use accelerators such as GPUs [5,16,19]. Accelerators can provide high computational througput and enable the use of compute-intensive algorithms that can operate on fewer projections (i.e., smaller datasets) [12,20,21]. However, these devices can accommodate only a small fraction of data and require repeated communication between host and device, which can limit the performance.

With regard to the second issue, we note that reconstruction algorithms might need to be developed according to different properties, including experimental setup and data acquisition; point of interests in reconstruction object; and total analysis or reconstruction time. These requirements result in various application-specific algorithms that are difficult to modify and maintain [8,9,18]. There are several frameworks that provide workflows and algorithms for tomographic reconstruction [7,22]; however, these typically provide limited support for easy implementation of reconstruction algorithms and parallelization of computation. The data-intensive computing community has also developed frameworks, such as Hadoop [2] and Spark [25], that ease the implementation of parallel algorithms. Although these frameworks show good scalability, they are not always suited for science applications that run on high-end clusters and supercomputers.

To address these issues, we make the following contributions. First, we introduce two parallelization techniques, `per-slice` and `in-slice`, for tomographic reconstruction algorithms. Our `in-slice` technique provides fine-grained high-performance parallelism using *replicated reconstruction objects*, which significantly improves the conventional `per-slice` approach. Second, we extend and optimize a MapReduce-like framework, MATE [13], to help implement these parallel methods efficiently. Third, we extensively evaluate the proposed methods and middleware-based implementations (for different reconstruction algorithms and real-world datasets). Our experimental results show that our middleware can scale almost linearly up to 8K cores and can achieve execution times on 32K cores that are $\geq 95.4\%$ less than those on 1K cores. Moreover, the average reconstruction times are improved from $\sim$2 hours (256 cores) to $\sim$1 minute (32K cores). To the best of our knowledge, this is the first study that examines the parallelization of tomographic reconstruction algorithms at this scale.

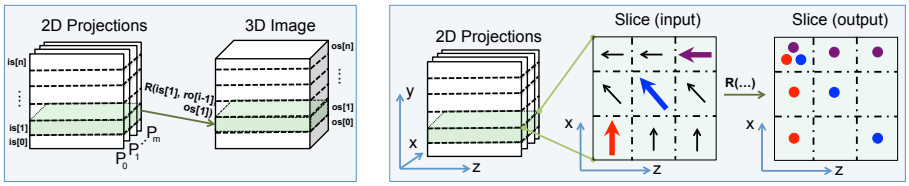


Fig. 1. Per-slice parallelization

2 Parallelization of Image Analysis and Reconstruction

In this section, we first discuss the organization of tomography datasets and reconstruction approach; then we present two parallelization techniques, `per-slice` and `in-slice`, for tomographic image reconstruction.

2.1 Tomography Datasets and Reconstruction

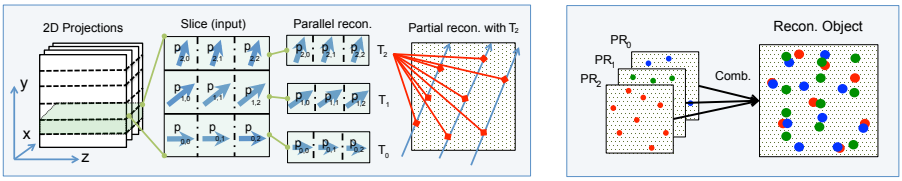
A tomography dataset is a set of 2D projections collected from different directions (θ) of a target sample. Each projection is a 2D array of floating-point numbers, each representing the line integrals of a ray, namely, a *ray-sum*. Therefore, a complete dataset is a 3D array in which the dimensions are projections, rows, and columns, respectively.

The tomographic reconstruction algorithms that we consider in this paper proceed in an iterative manner. At each iteration, rays are simulated according to the *ray-sum* values and reconstructed data from the previous iteration. Since rays in rows corresponding to different projections do not intersect, the reconstruction of individual rows (also referred to as *slices*) can proceed in parallel. We further discuss the tomographic image reconstruction and parallelization techniques in the following sections.

2.2 Per-Slice Parallel Reconstruction

Figure 1(a) illustrates the per-slice parallelization technique. We name the slices in the tomography dataset $IS = \{is_0, is_1, ..., is_n\}$, and denote the reconstructed object generated at the ith iteration as ro_i. Then, we can define a function R that, for iteration $i+1$, determines $ro_{i+1} = R(IS, ro_i)$ by computing each of n output slices os_j from the corresponding input slice is_j and from ro_i, and then setting $ro_{i+1} = \{os_0, os_1, ..., os_n\}$. Since there are no dependencies between slices, each slice can be processed independently.

Figure 1(b) shows a sample reconstruction operation. Each arrow represents a ray-sum value, with the direction of the arrow indicating the ray's θ. Simulating the propagation of the colored rays using the R function leads to the updates on the dotted cells in the output slice. Note that only the updates of the colored rays are shown in the figure. Typically, for each iteration, all rays in the projection (input) dataset are simulated for rapid convergence to the real 3D image. Although there are no dependencies between slices, different rays within a slice can update the same cell in the output slice, a situation that results in a race condition and limits the scalability of the per-slice parallelization technique.



(a) (b)

Fig. 2. In-slice parallelization

2.3 In-Slice Parallel Reconstruction

The per-slice technique can use only as much parallelism as there are slices in a dataset. Thus, for example, a dataset with 2,048 slices cannot be reconstructed with more than 2,048 parallel units (e.g. threads) and hence can take days to finish, depending on dataset size, reconstruction algorithm, and computational resources.

Our *in-slice* parallelization technique addresses this limitation by performing parallel reconstruction for each *ray*, therefore it significantly decreases the granularity of parallelism and increases the number of threads that can be applied. However, two obstacles must be addressed for in-slice parallelism: (1) different rays in the same slice may update the same cell at the same time (i.e., *race conditions* can occur); and (2) threads that operate on the same slice may need to synchronize in order to compute the correct output slice.

One way to address race conditions is to use mutexes. However, the use of mutexes can introduce significant overhead considering the many threads that must perform update operations on the same slice. An alternative approach is to replicate the assigned output slices (reconstruction objects) for each thread so that reconstruction operations can proceed independently. This approach avoids race conditions, and it achieves better performance than using (un)locks on individual reconstruction objects. Figure 2(a) illustrates how replication can be used for parallel reconstruction. First, each row in an input slice is assigned to a thread. Then, each thread simulates its assigned rays on the replicated reconstruction object. For example, in the figure, T_2 simulates rays $p_{2,*}$ and updates results on its own replica. Similarly, other threads perform the same operation on their own replicas.

We address the second issue by grouping and synchronizing threads according to their assigned slices. For example, Fig. 2(b) shows how replicas (PR_0, PR_1 and PR_2) from threads (T_0, T_1 and T_2) are combined in order to generate the correct reconstructed slice. The combination function used may vary depending on the reconstruction algorithm.

In-slice parallelization uses both replicas and combination operations to enable fine-grained parallelism and large-scale 3D reconstruction of tomographic images. However, these techniques require additional resources and introduce overheads that are not required in per-slice parallelization.

3 Our MapReduce-like Middleware

We next describe the processing structure of our middleware and the methods used to port the aforementioned parallelization techniques into this middleware.

Our middleware is built on top of MATE [13], a MapReduce-like middleware that supports *reduction-based processing structure* [4]. The MATE middleware

has been specialized to have a *reconstruction object* similar to the reduction object. The processing structure of the middleware consists of three main phases: *local reconstruction* (specialization of local reduction), *partial combination*, and *global combination*. In this section, we first explain how data management and distribution are performed in our system. Then, we provide details about the processing structure of our middleware. Finally, we introduce **ordered-subsetting** feature and its implementation.

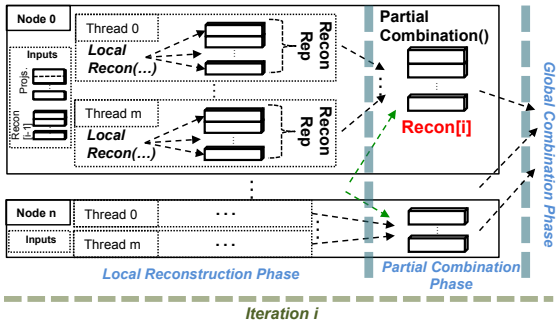
Data Organization and Distribution: Typically, tomography datasets are stored by using a scientific data format such as HDF5 [6]. Before beginning reconstruction, our middleware reads metadata information from the input dataset and allocates the resources required for the output dataset, setting the first dimension of the 3D reconstruction object to the number of slices and the other two dimensions to the number of columns. For instance, if the input dataset's dimensions are 360x2048x1024 (where 360 is the number of projections and 2048 and 1024 are the number of slices and columns, respectively), then the reconstruction object's dimensions are 2048x1024x1024.

Since the parallelization of reconstruction methods is based on slices, the data distribution partitions the input dataset along its second dimension and the 3D reconstruction object along its first dimension. For example, if the system has 128 processes, then the middleware partitions the input dataset and the reconstruction object into subsets of size 360x**16**x1024 and **16**x1024x1024, respectively, where in each case **16**=2048/128. It then assigns each portion of the input data and reconstruction object to a process. If there are more reconstruction processes/threads than there are slices, then portions of the same slice can be distributed to multiple processes.

Reconstruction Object: With per-slice parallelization, our middleware creates a single output object (portion of reconstruction object) in each process and lets all the threads update it. Since each slice is an atomic unit in the per-slice parallelization, threads can perform direct updates.

With in-slice parallelization, however, direct update on output object is not a correct operation. Recall from Section 2.3 that each slice may be shared among several threads, thus introducing the potential for race conditions. Our middleware eliminates these race conditions by creating a replica of the output object for each thread, which we refer to as *ReconRep* in Fig. 3(a). For example, assume that the user sets the number of processes per node to 1 and the number of threads per process to 32. In the aforementioned example, our middleware will allocate a replica of the corresponding input slices (16x1024x1024) for each thread. Therefore, each thread can perform reconstruction on its own replica. This use of replicas provides the greatest reconstruction parallelization among the threads, but does require additional synchronization. We quantify the costs associated with this overhead in our experiments.

Local Reconstruction Phase: Local reconstruction corresponds to the mapping phase of the MapReduce processing structure. The user implements the reconstruction algorithms in the local reconstruction function (*LocalRecon* in Fig. 3(a)); our middleware applies this function to each assigned data chunk. Each data chunk can be a set of slices (for per-slice parallelization) or a subset of rays in a slice (for in-slice parallelization).



(a)

```
Inputs   IS: Assigned projection slices
         Recon: Reconstruction object
         dist: Subsetting distance
Output   Recon: Final reconstruction object

/* (Partial) iteration i */
For each assigned projection slice, is, in IS {
  IR = GetOrderedRaySubset(is, i, dist);
  For each ray, ir, in rays IR {
    (k, off, val)    = LocalRecon(ir, Recon(is));
    ReconRep(k) = Reduce (ReconRep(k), off, val);
  }
}
/* Combine updated replicas */
Recon = PartialCombination(ReconRep)
/* Exchange and update adjacent slices*/
Recon = GlobalCombination(Recon)
```

(b)

Fig. 3. (a) Execution flow of our middleware; (b) pseudocode for in-slice parallelization

The local reconstruction function performs update operations using a 3-tuple, (sliceID, offset, value), where *sliceID* refers to the slice, *offset* is the data point on the slice, and *value* is the computed value. The usage of a 3-tuple is similar to (key, value) pairs in MapReduce, where key corresponds to sliceID and offset. Unlike MapReduce, however, the generated 3-tuple is being reduced/updated on replicas right after its generation.

Partial Combination Phase: After all the rays in the assigned data chunks are processed, the threads that are working on the same slices synchronize and combine their replicas by using a user-defined function (*PartialCombination()* in Fig. 3(a)). Although this phase resembles the *reduce* phase in the MapReduce processing structure, subtle yet important differences exist. First, there is no barrier between the local reconstruction and partial reconstruction phases; thus, idle time occurs only for the threads that need to synchronize. Second, the use of replicas eliminates the need for shuffling, grouping, and sorting operations used in MapReduce. This optimization has been shown to increase application performance significantly [13]. The partial combination is required only for the in-slice technique, since the per-slice technique does not need to use replicas.

Global Combination Phase: At the end of the partial combination phase, processes generate the slices of the 3D image. If the reconstruction algorithm uses neighboring slices, then the processes must exchange border slices. During the global combination phase, processes exchange these slices and then continue next iteration. This phase is required only for reconstruction algorithms that utilize neighboring slices.

Once all the iterations are completed, the final reconstruction object (*Recon[i]* in Fig. 3(a)) is generated. Our middleware, then, writes this reconstruction object using parallel HDF5.

Figure 3(b) gives **pseudocode** for the in-slice parallelization technique. Our middleware also supports **ordered-subsetting**, which lets users perform reconstruction using a subset of the rays in the assigned projection dataset. For example, assume that $PS_{is_i} = \{ps_0, ps_1, ..., ps_m\}$ are the projection rows in slice is_i. If the *GetOrderedRaySubset* function is called with $is = 0$, $i = 0$ and $dist = 2$ values, it sets $IR = \{ps_0, ps_2, ps_4, .., ps_m\}$. These projections' rows are then used for reconstruction. Here $dist$ is used for setting the distances between projections, and i is the current iteration. The iteration number determines the

beginning index of the projection; that is, if $i = 1$ and $dist = 2$, then projections $IR = \{ps_1, ps_3, .., ps_{m-1}\}$ are processed.

While only a subset of the rays is processed in each iteration, the middleware varies the beginning index of the projection so that all rays are eventually processed. Ordered subsetting converges more rapidly to the 3D image than does the sequential approach. After the target rays are determined, they are iteratively reconstructed. Again, notice that the generated values are *reduce*d in *ReconRep* right after the *LocalRecon* function. Once all assigned rays are processed, *ReconRep*s from different threads are combined with *PartialCombination*. The *Recon* object, then, is updated with *GlobalCombine*.

4 Experimental Results

We evaluated our middleware's performance and scalability using four iterative reconstruction algorithms and two real world datasets.

The reconstruction algorithms are ported from TomoPy [10], a widely used tomographic data processing and image reconstruction library. Specifically, we used maximum likelihood expectation maximization (`MLEM`), simultaneous iterative reconstruction technique (`SIRT`), penalized maximum likelihood (`PML`), and accelerated PML reconstruction (`APMLR`). Among these algorithms, `APMLR` requires adjacent slices, whereas `MLEM`, `SIRT`, and `PML` can perform reconstruction using data points on the same slice (i.e., neighboring slices are not needed).

To evaluate our framework, we used two datasets, `Seed` and `Hornby`, from two different APS beamlines. `Seed` is acquired from a seed of *arabidopsis thaliana*, a flowering plant [9]. It consists of 720 projections, each with 2048 rows and 501 columns (i.e., `720x2048x501` single-precision floating-point numbers). `Hornby` is an x-ray microtomography data from a shale sample [15]. It includes 360 projections, each with 2,048 rows and 1,024 columns. The reconstructed 3D images from `Seed` and `Hornby` have dimensions `2048x501x501` and `2048x1024x1024`, respectively.

We conducted our experiments on `Mira`, a 10-petaflops IBM Blue Gene/Q (BG/Q) supercomputer at the Argonne Leadership Computing Facility. Mira is equipped with 49,125 nodes, each with 16 cores (1600 MHz PowerPC A2) and 16 GB memory. The nodes have access to a GPFS file system that provides 24 PB of capacity and 240 GB/sec bandwidth. Moreover, the nodes are connected with a 5D torus proprietary network. [5]

4.1 Multithreaded Performance

We first evaluated the performance of our middleware when using different numbers of threads on a single node. In these experiments, we processed 64 slices (i.e., rows) of the `Seed` dataset using the `MLEM` and `APMLR` reconstruction algorithms.

The `MLEM` column in Fig. 4 shows the single-iteration reconstruction time (y-axis) for varying numbers of threads (x-axis). We observe the best performance with 32 threads, achieving a speedup of 18.76 relative to 1 thread.

[5] For more information, see http://www.alcf.anl.gov/mira

The performance with 64 threads is slightly slower, presumably because of overheads resulting from insufficient resources. Each BG/Q node can provide hardware registers for up to four threads; thus, the maximum number of supported threads is 16x4 per node. Since these threads share the same resources, we observe a performance degradation after 32 threads.

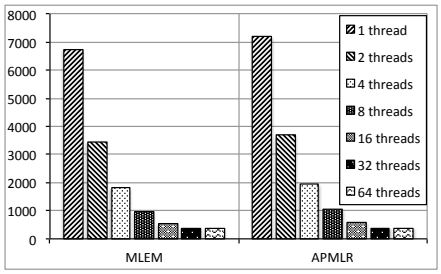
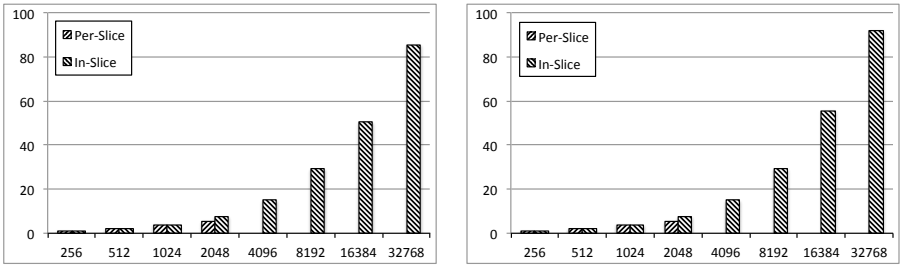


Fig. 4. Multithreaded reconstruction times (in sec) of the `Seed` dataset on a single BG/Q node.

The parallelization performance of `APMLR` follows the same trend as `MLEM`. Here, 32 threads provides the best performance, achieving 18.38 speedup. We observe, however, that the execution time of `APMLR` is slightly longer than that of `MLEM`. Although `MLEM` and `APMLR` use similar library functions from our middleware, `APMLR` performs additional computation and communication operations because of the use of adjacent slices, which introduces overhead during runtime.

4.2 Scalability

We next present the distributed-memory performance of our middleware. For these experiments, we used up to 2K nodes (i.e., 32K cores) and reconstructed both the `Seed` and `Hornby` datasets. We set the total number of iterations to 10.



(a) `MLEM` (b) `PML`

Fig. 5. Speedups achieved (y-axis) when reconstructing the `Seed` dataset on up to 32K cores (x-axis). Speedups are calculated with 256-core `per-slice` configurations.

Figure 5 shows the speedups achieved for the `MLEM` and `PML` algorithms on the `Seed` dataset, when using the `per-slice` and `in-slice` parallelization techniques. For these experiments, we used the 256-core `per-slice` timings as the baseline for speedup calculations, and set the number of threads per core to 2, i.e., 32 threads per node.

Notice that the `per-slice` technique has results for only up to 2K cores, while the `in-slice` technique has results for up to 32K cores. This difference is because the `per-slice` technique can create at most one thread per slice and the `Seed` dataset has 2K slices. Looking more closely, we see that `per-slice`

performs similarly to `in-slice` on up to 1K cores but less well on 2K cores. We attribute this relative decline to the maximum 2K threads that can be created by `per-slice` for `Seed`. Thus, `per-slice` has 16 threads per BG/Q node when running on 2K cores—less than the 32 threads that we showed in Section 4.1 to provide the best CPU utilization.

If we compare the 256- and 1K-core timings of `per-slice`, we see speedups of 3.99 and 3.98 for `MLEM` and `PML`, respectively. Since these speedups are close to the ideal (4x), we conclude that our middleware introduces negligible scalability overhead for these compute-intensive applications. The speedup for 2K-core configuration is 5.18 for both `MLEM` and `PML`, relative to the 256-core configuration. The fact that the `in-slice` technique can scale to more than 2K cores allows it to achieve far better reconstruction performance than does `per-slice`: at least 15.6x faster (on 32K cores) than `per-slice` (on 2K cores) for both `MLEM` and `PML`.

Looking more closely, we see that `in-slice` achieves almost linear speedup on up to 8K cores. Beyond 8K cores, however, the rate of speedup decreases. On 32K cores, for example, we observe a speedup of 85.6 for `MLEM` and 91.9 for `PML` application relative to the times taken on 256 cores. We attribute these less-than-perfect speedups to the short execution times ($\sim$1 min.) with many threads; the time taken by I/O; and the serial computation.

A closer look at the data shows that on up to 1K cores (in which both techniques utilize 32 threads per node), `per-slice` performs slightly better than `in-slice`. We attribute this difference to the need for `in-slice` to (1) synchronize threads that operate on the same slice at the end of each iteration and (2) perform additional computation for correct calculation of intermediate reconstruction objects (i.e., slices of 3D images). However, these overheads are small, ranging between 2.1% and 2.5%.

In Fig. 6, we show the execution times for the same applications, `MLEM` and `PML`, on the `Hornby` dataset, looking only at the more scalable `in-slice` technique in this case. For this set of experiments, we scaled the number of cores from 1K to 32K. The scalability results show a similar trend to that seen in the previous experiments: an almost linear speedup up to 8K cores for both `MLEM` and `PML`, increasing more slowly subsequently because of increased I/O, synchronization, and communication costs. The execution times on 32K vs. 1K cores show speedups of 24.22 and 25.5 for `MLEM` and `PML`, respectively.

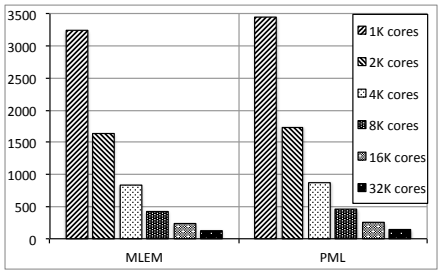
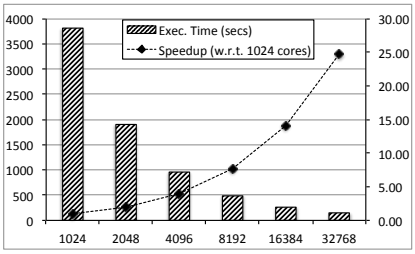


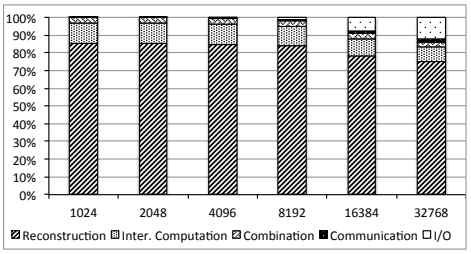
Fig. 6. Reconstruction times (in secs.) of `MLEM` and `PML` with the `Hornby` dataset using up to 32K cores.

Since `Hornby` is larger and thus computationally more demanding than `Seed`, we achieve better scalability than with the former dataset. Specifically, the speedups observed on 32K relative to 1K cores of `MLEM` and `PML` for `Seed` are 21.94 and 23.51, respectively: 7.8–9.5% less than those achieved with `Hornby`.

In Fig. 7, we show the performance achieved when we repeat the same experiments with the `APMLR` reconstruction algorithm. The execution times (left y-axis) and speedups (right y-axis) are presented in Fig. 7(a). Similar to the pre-



(a) APMLR performance



(b) APMLR reconstruction time breakdown: % time spent on different activities

Fig. 7. Performance achieved by the APMLR algorithm for the `Hornby` dataset, using up to 32K cores.

vious experiments, speedups are close to linear for up to 8K cores. Considering 32K vs. 1K cores, the speedup of `APMLR` is 24.77, which decreases the execution time from over 1 hour to less than 2.5 minutes. Note that the 1 hour execution time is with 1K cores; thus, the estimated execution time of the same application on a single BG/Q node (i.e., 16 cores) is more than 67 hours.

In Fig. 7(b), we show the percentage times spent in five different activities for the experiments of Fig. 7(a): (1) `Reconstruction` time, encompassing reconstruction and update operations on replicas; (2) Intermediate computation, i.e., `Inter. Computation`, in which intermediate matrices are calculated for reconstruction; (3) `Combination`, which is the sum of local and global combinations; (4) `Communication`, in which neighbors are updated; and (5) `I/O` time, in which data read and write operations are performed.

We see that `Reconstruction` dominates overall execution time for all core counts. The fraction of time spent in `I/O` increases as the number of cores grows; this result is not surprising since increasing the number of cores decreases per-core computation time and increases synchronization costs. `Communication` also does not scale well, since it requires constant time for updating neighbors. Moreover, the fraction of time spent in `Inter. Computation` and `Combination` remains roughly the same with increasing number of cores; the reason is that since these phases process data structures which are tightly coupled with the size of reconstruction objects, they show good parallelization performance.

4.3 Ordered-Subsetting Performance

In our next experiments, we evaluate the performance of the ordered-subsetting feature of our middleware. We apply the `APMLR` and `SIRT` reconstruction algorithms to the `Hornby` dataset with different distance parameters (d) on 4K cores and for 10 iterations. In contrast to previous experiments, however, these are *partial iterations*; that is, only a portion of the assigned data is processed.

Fig. 8 shows our results. We see that execution times improve with increasing distance configurations for

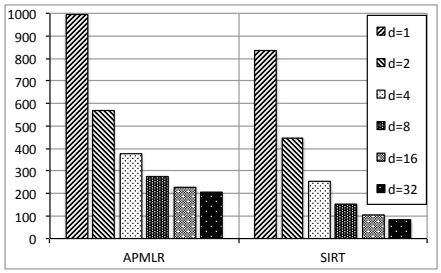


Fig. 8. Reconstruction times (in secs) of APMLR and SIRT using different distances. (`Hornby` dataset; # cores=4K).

both `APMLR` and `SIRT` algorithms. Since the amount of processed data directly affects the overall execution times, the *partial iteration* decreases the reconstruction times. For example, if we set $d = 2$, then each thread processes every other projection in an assigned slice. This effectively decreases the amount of processed input data by half (for each iteration). We note, however, that the reconstruction is performed on the same (full) output data; that is, the computational complexity of updating the 3D object is still high.

Comparing $d = 1$ and $d = 32$, we observe 4.8 and 9.7 speedups for `APMLR` and `SIRT`, respectively. Although the amount of processed data is 1/32 of the original for $d = 32$ configuration, update and reconstruction operations on 3D object still involve significant computation. Moreover, the strided access to the input data degrades the data locality. These effects are more visible in the case of the `APMLR` application, in which reconstruction also requires additional synchronization and communication.

The main advantage of using ordered-subsetting is the high image quality that it achieves with only a small number of *full iterations*. For example, 10 partial iterations with ordered-subsetting (where $d = 2$) provide better image quality than do five full iterations without ordered-subsetting for `SIRT` (normal execution). Note that 10 partial iterations correspond to 5 full iterations where $d = 2$. The ordered-subsetting method is being used extensively to improve reconstruction times and 3D image quality [3].

5 Conclusion

We have described the design and implementation of parallelization methods for tomographic reconstruction algorithms on high-performance clusters. We presented two parallel reconstruction techniques: `per-slice` and `in-slice`. The `in-slice` technique, which provides fine-grained high-performance parallelism using *replicated reconstruction objects*, represents a significant improvement over the conventional `per-slice` approach. We integrated the `per-slice` and `in-slice` techniques in a lightweight MapReduce-like middleware and extended the middleware to make it easy to implement different reconstruction algorithms.

We evaluated the techniques and middleware using four reconstruction algorithms and two large datasets. Our results show that our reconstruction approaches can achieve close to perfect speedups on up to 8K cores (512 BG/Q nodes). Moreover, the execution times of the 32K-core configurations (2K BG/Q nodes) show $\geq 95.4\%$ reduction in execution time relative to a 1K-core configuration. This acceleration enables near-real-time reconstruction of large datasets, such as those generated at synchrotron x-ray light sources.

Acknowledgments: This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under the contract DE-AC02-06CH11357 and the RAMSES project under the Next Generation Networking for Science Program.

References

1. J. Agulleiro and J.-J. Fernandez. Fast tomographic reconstruction on multicore computers. *Bioinformatics*, 27(4):582–583, 2011.

2. Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org, 2014. [Online accessed January-2015].
3. M. Beister, D. Kolditz, and W. A. Kalender. Iterative reconstruction methods in x-ray CT. *Physica Medica*, 28(2):94–108, 2012.
4. T. Bicer. *Supporting Data-Intensive Scientific Computing on Bandwidth and Space Constrained Environments*. PhD thesis, The Ohio State University, 2014.
5. C.-Y. Chou, Y.-Y. Chuo, Y. Hung, and W. Wang. A fast forward projection using multithreads for multirays on GPUs in medical image reconstruction. *Medical Physics*, 38(7):4052–4065, 2011.
6. F. De Carlo, D. Gürsoy, F. Marone, M. Rivers, D. Y. Parkinson, F. Khan, N. Schwarz, D. J. Vine, S. Vogt, S.-C. Gleber, S. Narayanan, M. Newville, T. Lanzirotti, Y. Sun, Y. P. Hong, and C. Jacobsen. Scientific data exchange: a schema for HDF5-based storage of raw and analyzed data. *Journal of Synchrotron Radiation*, 21(6):1224–1230, Nov. 2014.
7. J. Deslippe, A. Essiari, S. J. Patton, T. Samak, C. E. Tull, A. Hexemer, D. Kumar, D. Parkinson, and P. Stewart. Workflow management for real-time analysis of lightsource experiments. In *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*, pages 31–40. IEEE Press, 2014.
8. D. Gürsoy, T. Biçer, J. D. Almer, R. Kettimuthu, S. R. Stock, and F. De Carlo. Maximum a posteriori estimation of crystallographic phases in x-ray diffraction tomography. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 373(2043):20140392, 2015.
9. D. Gürsoy, T. Biçer, A. Lanzirotti, M. G. Newville, and F. De Carlo. Hyperspectral image reconstruction for x-ray fluorescence tomography. *Optics express*, 23(7):9014–9023, 2015.
10. D. Gürsoy, F. De Carlo, X. Xiao, and C. Jacobsen. TomoPy: a framework for the analysis of synchrotron tomographic data. *Journal of Synchrotron Radiation*, 21(5):1188–1193, Sept. 2014.
11. J. Hsieh, B. Nett, Z. Yu, K. Sauer, J.-B. Thibault, and C. A. Bouman. Recent advances in CT image reconstruction. *Current Radiology Reports*, 1(1):39–51, 2013.
12. B. Jang, D. Kaeli, S. Do, and H. Pien. Multi gpu implementation of iterative tomographic reconstruction algorithms. In *Biomedical Imaging: From Nano to Macro, 2009. ISBI'09. IEEE International Symposium on*, pages 185–188. IEEE, 2009.
13. W. Jiang, V. T. Ravi, and G. Agrawal. A Map-Reduce system with an alternate API for multi-core environments. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 84–93, Washington, DC, USA, 2010. IEEE Computer Society.
14. M. Jones, R. Yao, and C. Bhole. Hybrid MPI-OpenMP programming for parallel OSEM PET reconstruction. *Nuclear Science, IEEE Transactions on*, 53(5):2752–2758, Oct. 2006.
15. W. Kanitpanyacharoen, D. Y. Parkinson, F. De Carlo, F. Marone, M. Stampanoni, R. Mokso, A. MacDowell, and H.-R. Wenk. A comparative study of x-ray tomographic microscopy on shales at different synchrotron facilities: Als, aps and sls. *Journal of synchrotron radiation*, 20(1):172–180, 2013.
16. D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, and A. W. Toga. CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms. *Computer Methods and Programs in Biomedicine*, 106(3):175–187, 2012.
17. K. Mohan, S. Venkatakrishnan, J. Gibbs, E. Gulsoy, X. Xiao, M. De Graef, P. Voorhees, and C. Bouman. Timbir: A method for time-space reconstruction from interlaced views. *Computational Imaging, IEEE Transactions on*, PP(99):1–1, 2015.
18. C. Phatak and D. Gürsoy. Iterative reconstruction of magnetic induction using lorentz transmission electron tomography. *Ultramicroscopy*, 150(0):54 – 64, 2015.
19. G. Pratx, G. Chinn, P. Olcott, and C. Levin. Fast, accurate and shift-varying line projections for iterative reconstruction using the GPU. *Medical Imaging, IEEE Transactions on*, 28(3):435–445, March 2009.
20. E. Y. Sidky, C.-M. Kao, and X. Pan. Accurate image reconstruction from few-views and limited-angle data in divergent-beam CT. *Journal of X-ray Science and Technology*, 14(2):119–139, 2006.
21. S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. Hwu, B. P. Sutton, Z.-P. Liang, et al. Accelerating advanced MRI reconstructions on GPUs. *Journal of Parallel and Distributed Computing*, 68(10):1307–1318, 2008.
22. K. Thielemans, C. Tsoumpas, S. Mustafovic, T. Beisel, P. Aguiar, N. Dikaios, and M. W. Jacobson. Stir: software for tomographic image reconstruction release 2. *Physics in Medicine and Biology*, 57(4):867, 2012.
23. J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger, and G. Wellein. Pushing the limits for medical image reconstruction on recent standard multicore processors. *International Journal of High Performance Computing Applications*, 2012.
24. Y. Wang, F. De Carlo, D. C. Mancini, I. McNulty, B. Tieman, J. Bresnahan, I. Foster, J. Insley, P. Lane, G. von Laszewski, et al. A high-throughput x-ray microtomography system at the Advanced Photon Source. *Review of Scientific Instruments*, 72(4):2062–2068, 2001.
25. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, Berkeley, CA, USA, 2010. USENIX Association.
26. K. Zeng, E. Bai, and G. Wang. A fast CT reconstruction scheme for a general multi-core PC. *International Journal of Biomedical Imaging*, 2007, 2007.