Communication Optimization for Medical Image Reconstruction Algorithms

Torsten Hoefler[1], Maraike Schellmann[2], Sergei Gorlatch[2], and Andrew Lumsdaine[1]

[1] Open Systems Lab, Indiana University, Bloomington IN 47405, USA,
{htor,lums}@cs.indiana.edu
[2] Institute of Computer Science, University of Münster, Germany
{schellmann,gorlatch}@uni-muenster.de

Abstract. This paper presents experiences and results obtained in optimizing the parallel communication performance of a production-quality medical image reconstruction application. The fundamental communication operations in the application's principal algorithm are collective reductions. The overhead of these operations was reduced by transforming the algorithm to overlap its computation and communication. Several different approaches to communication progress were studied, both user-directed and asynchronous. Experimental results comparing the new approach to the previous implementation show overall application performance improvements of up to 8%, when run on 32 nodes.

1 Introduction

Modern medical methods for diagnosis and treatment require very accurate, high-resolution 3D images of the inside of a human body. In order to provide the required accuracy and resolution, reconstruction algorithms in medical imaging are becoming more complex and time-consuming. In this paper, we study Positron Emission Tomography (*PET*) reconstruction, where one of the most popular, but also most time-consuming algorithms—the list-mode OSEM algorithm—requires several hours on a common PC in order to compute a 3D reconstruction. With advanced algorithms that incorporate more physical aspects of the PET process, computation times are rising even further [1]. This motivates the parallelization of the algorithm on multiprocessor machines [2].

Our current parallel implementation uses Message Passing Interface (MPI) [3] collective operations and OpenMP. Collective operations allow the programmer to express high-level communication patterns in a portable way, such that implementers of communication libraries provide machine-optimized algorithms for those complex communications. Our earlier work showed that many parallel algorithms can be implemented with exclusive use of collective communications and that portability, readability, programmability, code maintenance and performance are often improved in this case [4].

In this paper, we use non-blocking collective operations to reduce the communication overhead of the parallel list-mode OSEM algorithm. Non-blocking

collective operations are a new class of collective operations that combines all benefits of collective operations with the ability to overlap communication and computation [5]. They also relax the tight bond between computation and communication by performing communication tasks in the background. In our case study, the scalability for the fixed problem size (strong scaling) is limited by a collective data reduction operation in which the message size is independent of the number of MPI processes (in our example $48\,MiB$). To reduce the communication overhead, we transform our code to leverage non-blocking collective operations offered by LibNBC [6], which provide—additionally to the overlapping of communication with computation— high-level communication offload using the InfiniBand network. We analyze the code transformations and provide an analytical runtime model that identifies the overlap potential of our approach.

The rest of the paper is organized as follows: we start with an introduction to the list-mode OSEM algorithm in Section 1.1 and describe its current parallelization in Section 1.2. In Section 2, we show the necessary changes to our implementation that allow overlapping of communication and computation, and in Section 3, we discuss the optimization of LibNBC to maximize overlap. Conclusions are presented in Section 4.

1.1 List-Mode OSEM Algorithm

PET is a medical imaging technique that displays metabolic processes in a human or animal body. PET acquisition proceeds as follows: A slightly radioactive substance which emits positrons when decaying is applied to the patient who is then placed inside a *scanner*. The detectors of the scanner measure so-called events: When the emitted positrons of the radioactive substance collide with an electron residing in the surrounding tissue near the decaying spot (up to 3 mm from the emission point), they are annihilated. During annihilation two gamma rays emit from the annihilation spot in opposite directions and form a line, see Fig. 1. These gamma rays are registered by the involved detectors; one such registration is called *event*.

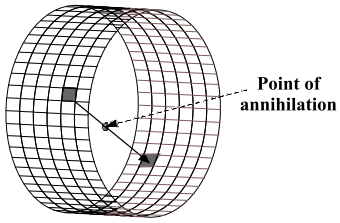


Fig. 1. Detectors register an event in a PET-scanner with 6 detector rings

```
for each(iteration k){
 for each(subiteration l){
  for (event i ∈ S_l){
    compute A_i
    compute c_l += (A_i)^t 1/(A_i f_l^k)}
  f_{l+1}^k = f_l^k c_l}
 f_0^{k+1} = f_{l+1}^k}
```

Listing 1.1. Sequential list-mode OSEM algorithm.

During one investigation, typically 10^7 to $5 \cdot 10^8$ events are registered, from which a reconstruction algorithm computes a 3D image of the substance's distribution in the body.

In this work, we focus on the very accurate, but also quite time-consuming list-mode OSEM (Ordered Subset Expectation Maximization) reconstruction algorithm [7] which computes the image f from the m events saved in a list.

The algorithm works block-iteratively: in order to speed up convergence, a complete iteration over all events is divided into s subiterations (see Listing 1.1). Each subiteration processes one block of events, the so-called subset. The starting image vector is $f_0 = (1, ..., 1) \in \mathbb{R}^N$, where N is the number of voxels in the image being reconstructed. For each subiteration $l \in 0, ..., s-1$, the events in subset l are processed in order to compute a new, more precise reconstruction image f_{l+1}, which is used again for the next subiteration as follows:

$$f_{l+1} = \underbrace{\frac{1}{A^t_{norm}\mathbf{1}}}_{:=a} f_l c_l; \quad c_l = \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l}, \tag{1}$$

where S_l are the indices of events in subset l, $\mathbf{1} = (1, ..., 1)$. For the i-th row A_i of the so-called system-matrix $A \in \mathbb{R}^{m \times N}$, element a_{ik} denotes the length of intersection of the line between the two detectors of event i with voxel k. The so-called normalization vector $a = \frac{1}{A^t_{norm}\mathbf{1}}$ is independent of the current subiteration and can thus be precalculated. In the computation of f_{l+1} the multiplication of $a f_l c_l$ is performed element by element.

After one iteration over all subsets, the reconstruction process can either be stopped, or the result can be improved with further iterations over all subsets (see pseudocode in Listing 1.1). Note that the optimal number of events per subset $m_s = m/s$ only depends on the scanner geometry and is thus fixed (for our scanner [8], it is $m_s = 10^6$).

1.2 Algorithm Parallelization Concept

Two strategies to parallelize the list-mode OSEM algorithm exist: PSD (Projection Space Decomposition) and ISD (Image Space Decomposition). In [2] we showed that PSD outperforms ISD in almost all cases and we therefore chose the PSD strategy that distributes the events among the processes for our parallelization: Since f_{l+1} depends on f_l we parallelize the computations within one subset. We decompose the input data, i.e., the events of one subset into P (=number of nodes) blocks and process each block simultaneously. The calculations for one subset includes four steps on every node j ($\forall\, j = 1, \ldots, P$) (cf. Fig. 2):

1. read m_s/P events
2. compute $c_{l,j} = \sum_{i \in S_{l,j}} (A_i)^t \frac{1}{A_i f_l}$. This includes the on-the-fly computation of A_i for each event in $S_{l,j}$.
3. sum up $c_{l,j} \in \mathbb{R}^N$ ($\sum_j c_{l,j} = c_l$) with MPI_Allreduce
4. compute $f_{l+1} = f_l c_l$

We implemented steps 1 and 3 (i.e., the reading of data and the actual communication of the parallel algorithm) using MPI_File_Read and blocking MPI_Allreduce. We start one process per node and use the SMP node in a cluster by additionally parallelizing steps 2 and 4 using OpenMP.

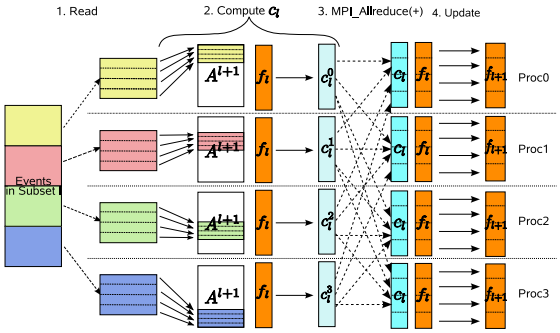


Fig. 2. Parallel list-mode OSEM algorithm on four nodes with the blocking MPI_Allreduce using four OpenMP threads per node

2 Parallel Algorithm with Non-Blocking Collectives

In order to optimize the parallel algorithm, we reduce the overhead arising from the allreduce step by overlapping its communication with computations that are independent of the communicated data. We use LibNBC's [6] non-blocking version of MPI_Allreduce called NBC_Iallreduce, and the MPI_Wait counterpart NBC_Wait.

We overlap the reading of events for subset l and the computation of the corresponding sub-matrix A^l (which is composed of rows $i \in S_l$) with the communication of c_{l-1} of the preceding subset (see Fig. 3). Hence, the non-blocking parallel algorithm on nodes $j\,(\forall\, j = 1, \ldots, P)$ reads as follows:

1. read m_s/P events in the first subset
2. compute $c_{l,j} = \sum_{i \in S_{l,j}} (A_i)^t \dfrac{1}{A_i f_l}$. This includes the on-the-fly computation of A_i for each event in $S_{l,j}$ in the first subset. Beginning from the second subset, rows A_i have already been computed in parallel with NBC_Iallreduce
3. start NBC_Iallreduce for $c_{l,j}$ $(\sum_j c_{l,j} = c_l)$
4. in every but the last subset, each node reads the m_s/P events for subset $l+1$ and computes A_i for subset $l+1$
5. perform NBC_Wait to finish NBC_Iallreduce
6. compute $f_{l+1} = f_l c_l$

Note that in this approach, A^l has to be kept in memory. If not enough memory is available, one part A^l can be computed as in the original version in step 2 and the other part in step 4. Also, since A_i is precomputed, the computation of $c_l += (A_i)^t \dfrac{1}{A_i f_l}$ could cause CPU cache misses that influence the performance.

2.1 Analyzing the Overlap Potential

In order to identify the overlap potential of our approach, we develop an analytical runtime model for the overlappable computations. We denote the sequential time to compute the m_s rows of A^l by $t^1_{A_l}(m_s)$ and the time to read each node's

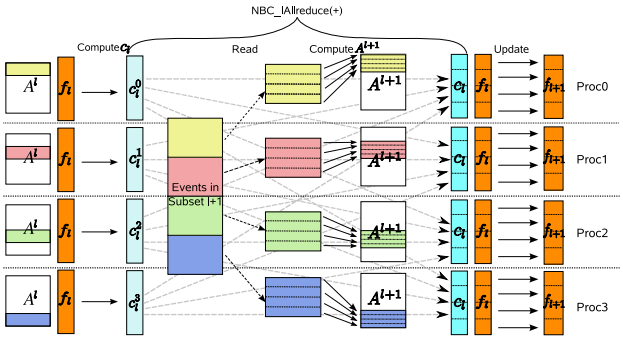


Fig. 3. Parallel list-mode OSEM algorithm on four nodes with the non-blocking NBC_Iallreduce using four OpenMP threads per node

m_s/P events by $t^P_{read}(m_s/P)$. If we assume that $t^P_{read}(m_s/P) \approx t^P_{read}(m_s)/P$, we obtain a computational overlap time per subset with one thread on each of the P nodes of

$$t^P_{CompOver} = t^P_{read}(m_s/P) + t^1_{A^l}(m_s)/P \approx (t^P_{read}(m_s) + t^1_{A^l}(m_s))/P \qquad (2)$$

We will verify our model (2) with experiments in Section 3.2.

On q cores per node, the ideal parallel efficiency with our OpenMP parallelization would be $\beta(q) = t^1_{A_i}/(t^q_{A_i} \cdot q) = 1$. However, with an increasing number of threads sharing the cache, cache misses increase considerably and thus our OpenMP implementation scales worse than ideally on multi-core machines. For example, on a quad-core processor, efficiency is $\beta(4) = 0.5$.

Note that on systems where file I/O and MPI communication share the same network, the overlapping of reading of data and communication might be limited due to the network's bandwidth. Hence, in the worst case, with the network fully loaded by MPI communication, $t^P_{CompOver} = t^1_{A^l}(m_s)/P$.

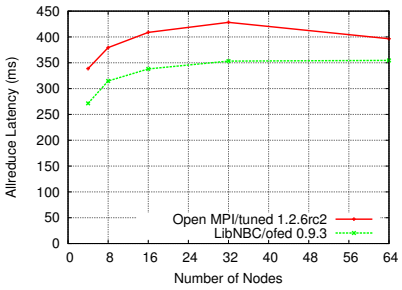
3 Optimization of Non-Blocking Collectives

In this section, we explain the optimized implementation of non-blocking collective operations for the needs of the parallel list-mode OSEM reconstruction algorithm.

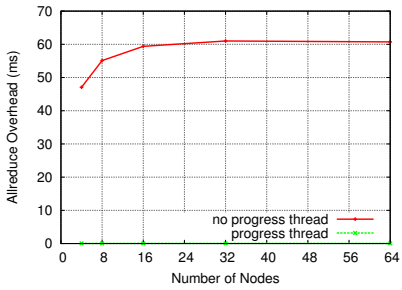
3.1 Implementation with LibNBC

We used the InfiniBand optimized version of LibNBC for this work. This version uses an overlap-optimized InfiniBand transport layer which achieves better computation/communication overlap than open source MPI implementations [9]. The algorithm that is used to all-reduce large messages in LibNBC uses a pipelined communication scheme to maximize overlap and to use the network bandwidth as efficiently as possible. On P processes, it divides the data into P chunks. Every process receives a chunk from its left neighbor, computes it and passes it on to the next neighbor in a ring-like fashion. This algorithm finishes the reduction in $2 \cdot P - 2$ communication/computation cycles.

Fig. 4(a) shows a comparison of the "blocking performance"[1] of LibNBC 0.9.3 with the "tuned" collective module of Open MPI 1.2.6rc2. The measurements were done with NBCBench [10] on the *odin* cluster at Indiana University. *Odin* consists of 128 dual core dual socket 2 GHz AMD Opteron 270 processors connected with SDR InfiniBand, uses NFSv3 over Gigabit Ethernet as file system and the Intel compiler suite version 9.1. LibNBC's allreduce uses multiple communication rounds (cf. [6]). This requires the user to ensure progress manually by calling NBC_Test or run a separate thread that manages the progression of LibNBC (i.e., progress thread). Fig. 4(b) shows the communication overhead with and without a progress thread under the assumption that the whole communication latency can be overlapped with computation (i.e., the overhead is a lower bound) and the progress thread runs on a spare CPU core (the overhead with a progress thread is constantly $3\mu s$, due to the fully asynchronous processing, and thus at the very bottom of Fig. 4(b)).



(a) Comparison of the Allreduce Performance of LibNBC and Open MPI



(b) Comparison of the Allreduce Overheads of different LibNBC Options

Fig. 4. Allreduce Performance Results for a 48MiB Summation of Doubles

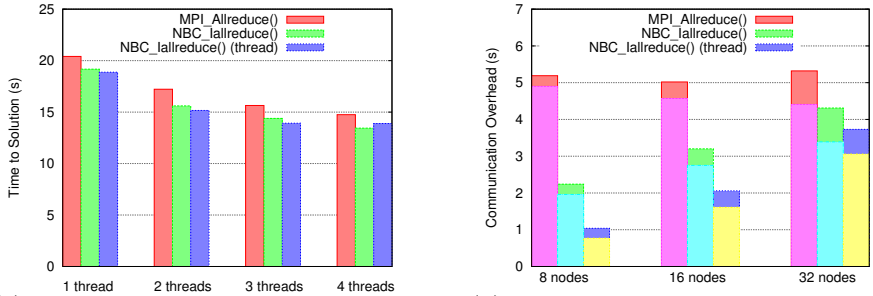
3.2 Benchmark Results

In our benchmarks, we study the reconstruction of data collected by the *quadHIDAC* small-animal PET scanner [8]. We used 10^7 events divided into 10 subsets and performed one iteration over all events. The reconstruction image has the size $N = (150 \times 150 \times 280)$ voxels. We ran a set of different benchmarks on the *odin* system. We compared the non-threaded and threaded versions of LibNBC using the InfiniBand optimized transport. We progressed the non-threaded version with $4 \times P$ calls to NBC_Test that are equally distributed over the overlapped time. The threaded version of LibNBC is implemented by using InfiniBand's blocking semantics and the application did not call NBC_Test at all. We benchmarked all configurations of LibNBC and the original MPI implementation on 8, 16 and 32 nodes with 1, 2, 3 or 4 OpenMP threads per node three times and report the average times across all runs and processes (the variance between the runs was very low).

[1] NBC_Iallreduce immediately followed by NBC_Wait

Computational Overlap The computational overlap time per subset $t^P_{CompOver}$ decreases—as expected from our model—linearly with increasing number of processes P. The average time was 833.5 ms on 8, 469.9 ms on 16 and 241.8 ms on 32 nodes. With reading time t^P_{read} ranging from 55.4 ms on 8 to 11.2 ms on 32 nodes and computation time $t^P_{A_l}$ ranging from 778.1 ms to 230.6 ms on 8 and 32 nodes, respectively, we are able to verify our model (2) with an error of about 6%.

Fig. 5(a) shows the application running time on 32 nodes with different numbers of OpenMP threads per node. We see that the non-threaded version of LibNBC is able to improve the running time in every configuration. However, the threaded version is only able to improve the performance if it has a spare core available because of scheduler congestion on the fully loaded system. The performance gain also decreases with the number of OpenMP threads. This is because we studied a strong scaling problem and the overlappable computation time gets shorter with more threads computing the static workload and is eventually not enough to overlap the full communication. Another issue for smaller node-counts is that our transformed implementation is, as described in Section 2.1, slightly less cache-friendly which limits the application speedup at smaller scale.



(a) Runtime on 32 Nodes with Different Number of OpenMP Threads

(b) Communication Overhead for Different Node Counts and a single Thread

Fig. 5. Application Benchmark for different number of OpenMP threads and nodes

Fig. 5(b) shows the communication overhead for different node counts with one thread per node[2]. Our implementation achieves significantly smaller communication overhead for all configurations. However, the workload per node that can be overlapped decreases, as described above, with the number of nodes, while the communication time of the 48 MiB Allreduce remains nearly constant. Thus, the time to overlap shrinks with the number of nodes and limits the performance gain of the non-blocking collectives.

4 Conclusions

We applied non-blocking collective operations to the mixed MPI/OpenMP parallel implementation of the list-mode OSEM algorithm and analyzed the perfor-

[2] the lower part of the bars denotes the Allreduce overhead

mance gain for a fixed problem size (strong scaling) on different setups of MPI processes and OpenMP threads.

The conducted study demonstrates that the overlap optimization potential of non-blocking collectives depends heavily on the time to overlap (amount of work to do while communicating) which usually decreases while scaling to larger process counts. However, even in the worst case (smallest workload) of our example, running 128 threads on 32 nodes, LibNBC was able to reduce the communication overhead from 40.31% to 37.3%. In the best case, with one thread on 8 nodes (highest workload per process), the communication overhead could be efficiently halved from 12.0% to 5.6%.

Acknowledgments

This work was partially supported by a grant from the Lilly Endowment, National Science Foundation grant EIA-0202048 and a gift from the Silicon Valley Community Foundation on behalf of the Cisco Collaborative Research Initiative. This work was also partly funded by the Deutsche Forschungsgemeinschaft, SFB 656 MoBil (Project B2).

References

1. Kösters, T., Wübbeling, F., F.Natterer: Scatter correction in PET using the transport equation. In: IEEE Nuclear Science Symposium Conference Record, IEEE (October 2006) 3305–3309
2. Schellmann, M., Gorlatch, S.: Comparison of two decomposition strategies for parallelizing the 3d list-mode OSEM algorithm. In: Proceedings Fully 3D Meeting and HPIR Workshop. (2007) 37–40
3. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville (1997)
4. Gorlatch, S.: Send-receive considered harmful: Myths and realities of message passing. ACM Trans. Program. Lang. Syst. **26**(1) (2004) 47–56
5. Brightwell, R., Riesen, R., Underwood, K.D.: Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. Int. J. High Perform. Comput. Appl. **19**(2) (2005) 103–117
6. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07, IEEE Computer Society/ACM (11 2007)
7. Reader, A.J., Erlandsson, K., Flower, M.A., Ott, R.J.: Fast accurate iterative reconstruction for low-statistics positron volume imaging. Phys. Med. Biol. **43**(4) (1998) 823–834
8. Schäfers, K.P., Reader, A.J., Kriens, M., Knoess, C., Schober, O., Schäfers, M.: Performance evaluation of the 32-module quadHIDAC small-animal PET scanner. Journal Nucl. Med. **46(6)** (2005) 996–1004
9. Hoefler, T., Lumsdaine, A.: Optimizing non-blocking Collective Operations for InfiniBand. In: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS). (04 2008)
10. Hoefler, T., Schneider, T., Lumsdaine, A.: Accurately Measuring Collective Operations at Massive Scale. In: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS). (04 2008)