Automatic Memory Optimizations for Improving MPI Derived Datatype Performance

Surendra Byna[1], Xian-He Sun[1], Rajeev Thakur[2], and William Gropp[2]

[1] Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA
{bynasur, sun}@iit.edu
[2] Math. and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
{thakur, gropp}@mcs.anl.gov

Abstract. MPI derived datatypes allow users to describe noncontiguous memory layout and communicate noncontiguous data with a single communication function. This powerful feature enables an MPI implementation to optimize the transfer of noncontiguous data. In practice, however, many implementations of MPI derived datatypes perform poorly, which makes application developers avoid using this feature. In this paper, we present a technique to automatically select templates that are optimized for memory performance based on the access pattern of derived datatypes. We implement this mechanism in the MPICH2 source code. The performance of our implementation is compared to well-written manual packing/unpacking routines and original MPICH2 implementation. We show that performance for various derived datatypes is significantly improved and comparable to that of optimized manual routines.

Keywords: MPI, derived datatypes, MPI performance optimization.

1 Introduction

MPI derived datatypes [7] enable users to describe noncontiguous memory layouts compactly and to use this compact representation in MPI communication functions. Derived datatypes also enable an MPI implementation to optimize the transfer of noncontiguous data. For example, if the underlying communication mechanism supports noncontiguous data transfers, the MPI implementation can communicate the data directly without packing it into a contiguous buffer. On the other hand, if packing into a contiguous buffer is necessary, the MPI implementation can pack the data and send it contiguously.

In practice, however, many MPI implementations perform poorly with derived datatypes—to the extent that users often resort to packing the data manually into a contiguous buffer and then calling MPI. Such usage clearly defeats the purpose of having derived datatypes in the MPI Standard. Since noncontiguous communication occurs commonly in many applications (for example, Fast Fourier transform, array redistribution, and finite-element codes), improving the performance of derived datatypes has significant value.

The performance of derived datatypes can be improved in several ways. Researchers have used data structures that allow a stack-based approach to parsing a

B. Mohr et al. (Eds.): PVM/MPI 2006, LNCS_4192, pp. 238 – 246, 2006.

datatype, rather than making recursive function calls, which are expensive [4], [11], [12]. These works improved the performance of derived datatypes to the level of performance with naïve manual implementations for packing noncontiguous data. (We do better than that in this paper.) Wu et al. [13] improved the performance of MPI derived datatypes by taking advantage of the features in InfiniBand to overlap packing and unpacking a message with network communication.

The performance of derived datatypes can be improved further by using optimized algorithms for packing and unpacking of data. Many implementations of derived datatypes use loops in packing/unpacking noncontiguous data. Utilizing data locality in these loops by applying loop optimizations, which a developer cannot easily do without advanced knowledge of memory hierarchy design and optimizations, is beneficial. This area is the focus of our study. These techniques are useful for MPI implementations on various network channels and the performance gain is not limited to fast networks. Our previous work [1] presents the scope of performance improvement by using MPI's profiling interface (PMPI). In this paper, we present automatic selection of optimized packing/unpacking templates within the MPICH2 source code, based on data access patterns, data size, and memory architecture. Ogawa et al. [9] used optimized templates in improving MPI performance for instantiating partial-evaluation code selection in order to reduce software overhead. We, in contrast, use templates to optimize memory performance.

The rest of this paper is organized as follows. In Section 2, we present the design of our optimization mechanism. In Section 3, we describe the implementation details in selecting optimized templates dynamically. In Section 4, we present our experimental results, followed by conclusions in Section 5.

2 Optimization Mechanism

To choose optimized templates automatically, we developed a systematic approach. Our method first retrieves the data access pattern of a derived datatype from user's definition and verifies whether performance improvement is possible with optimizations for a derived datatype before applying them. If improvement is possible, our optimization method uses an analytical model [2] to predict memory access cost and to find optimization parameters with the lowest access cost. These parameters are passed to templates to pack/unpack noncontiguous data.

Overall procedure of optimizing an MPI communication function using derived datatypes has two steps. In the first step, we verify whether a datatype is optimizable or not, and find optimization parameters. In the second step, MPI communication function calls optimized templates automatically.

In MPI programs, after defining a derived datatype, it has to be committed by calling `MPI_Type_commit`. We modified the implementation of the `MPI_Type_commit` function to verify whether optimization is possible. The modified implementation first retrieves the data access pattern, which includes the type of the user-defined datatype, old datatype, strides between consecutive memory accesses, size of the data items, and depth of the derived datatype. If the old datatype is another derived datatype (that is, when a derived datatype is nested), `MPI_Type_commit` retrieves these values for that inner datatype as well. We use

the datatype decoder functions of MPI-2, namely `MPI_Type_get_envelope` and `MPI_Type_get_contents` to retrieve the pattern. The overhead of decoding datatypes by using these functions is low.

In order to determine whether a datatype is optimizable or not, the modified `MPI_Type_commit` function verifies a series of heuristics that cause cache misses. It verifies whether the datatype is contiguous or noncontiguous, examines whether the data size is more than cache size, and then calculates the factor of cache and TLB reuse. The optimization method reverts back to the original implementation if it determines that the performance cannot be improved at any of these verifications. We use an optimization flag (`is_optimizable`) to keep track of the results of these verifications. If the performance can be improved, `MPI_Type_commit` determines the optimization parameters and sets the flag `is_optimization` to 1.

We developed optimized templates to pack/unpack noncontiguous data by using various loop optimization methods. In our current implementation, these templates use cache blocking [5], loop unrolling, array-padding optimizations, and software-level prefetching [8].

Various parameters are required in using these optimizations. Examples of optimization parameters are: block size for cache blocking, number of padding elements for array padding, and prefetching distance for software-level prefetching. In our approach, we first select these optimization parameters based on heuristics. To determine if these parameters are optimal, we developed a simple, fast, and accurate memory-access-cost prediction model [2]. This model verifies whether the memory access cost is reduced with the selected parameters. A new set of optimization parameters are selected if the cost is not optimized and the prediction model verifies for lowered cost again.

Examples of optimization parameter selection are as follows. For cache-blocking optimization, the block size is selected in a way that each block fits into the cache memory and virtual-to-physical address mappings of that block fit in the TLB (Translation Look-aside Buffer). For software prefetching, the number of loop iterations needed to overlap a prefetching memory access is called the *prefetching distance* [8]. Assuming memory access latency is *l*, and the work per loop iteration is *w*, the prefetch distance is *ceiling (l/w)*. The main loop that packs data is unrolled for all the references that reuse cache lines that are prefetched. An *epilogue loop* is called without prefetching to execute the last few iterations that do not fit in the main loop. We use a special `gcc` function `__builtin_prefetch` to issue these prefetch instructions. A special flag, `-mcpu,` has to be set to compile MPI source code.

In the second step, when the `MPI_Send` function is called to send the data, if the `is_optimization` flag is 1, the `MPI_Send` calls optimized packing templates using the optimization parameters. These templates are also used when the user calls `MPI_Pack` or `MPI_Unpack` to pack or unpack noncontiguous data.

3 Performance Results

We used three sets of benchmarks to evaluate the performance of our optimized implementations.

1. Simple derived datatypes: We chose fixed derived datatypes defined by the SKaMPI benchmark [10]. They describe a memory layout consisting of a number of units of a basic datatype. The number of units depends on the size of data, the size of basic datatype, and strides. We used vector and indexed datatypes.
2. Nested derived datatypes: We use the nested derived datatypes described by Ross et al. in [11]. These datatypes represent a collection of elements from a 3D array. When a 3D array is stored in row-major order, accessing the YZ face and all the YZ faces of the array in X direction is noncontiguous and has poor locality when the size of the YZ face is more than the cache or TLB sizes. We tested a nested datatype describing a 3D cube of YZ planes in the X direction with a vector of vectors (vector of YZ planes in an array).
3. NAS benchmarks: Lu et al. [6] modified four NAS benchmarks to apply MPI derived datatypes for noncontiguous data communication. Among these, LU, BT, and SP have small data transfers and do not benefit from memory optimizations. In the MG benchmark, the data transfers in the `comm3` function are noncontiguous and are implemented as packing-then-sent by a sender process and receive-then-unpacking by a receiver. The datatypes described in the modified code are nested datatypes that represent vectors of vectors. We also tested the performance of the matrix transpose operation from the NAS parallel benchmarks' Fourier Transform (FT) program, using MPI derived datatypes. To describe the transpose operation with a derived datatype, we use a datatype that is a vector of vectors (vector of columns in an array).

Except for the NAS MG benchmark, we obtained the performance results of all other benchmarks with an `MPI_Send/Recv` ping-pong operation. In this operation, a process sends a noncontiguous message that is described by the MPI derived datatypes, and a destination process receives it contiguously. The destination process then sends back the data with the same derived datatype and is received at the first process contiguously. The time is measured at the first process and halved to find the communication cost for one complete data transfer. We ran 20 iterations of each program and calculated the minimum time. We present the performance as transfer rate (MB/s) to normalize the results. The size of the message used in the ping-pong operation is divided by the measured time to find the rate. For the NAS MG benchmark, we compare the execution time of the benchmark.

We compare the performance results for three implementations: manually packing data and sending it (no derived datatypes), MPICH2 version 1.0.3 (unoptimized), and our optimized implementation of the MPICH2 code. The manually implemented pack and unpack codes are written to represent the way a good programmer would write them. Ross et al. [11] showed that the implementation of derived datatypes in MPICH2 outperform those implemented in LAM/MPI. Therefore, we directly compare our results with MPICH2. We compile all manual codes and MPI installations with `gcc` version 3.2.3 with the flags `-O6`.

To test the portability of our optimized implementations, we ran these experiments on two different clusters: a 350-node Linux cluster (*jazz*) at Argonne National Laboratory and an 84-node Sun cluster (*sunwulf*) at Illinois Institute of Technology. The nodes of *jazz* have a 2.4 GHz Pentium-4 processor with 1 GB of memory. These processors have 512 KB of built-in L2 cache, with a 64 byte cache line and 8-way

associative, a TLB of 128 entries, and a page size of 4 KB. The network interconnect of this cluster is Fast Ethernet. Each node of the *sunwulf* cluster is a Sun Blade-100 workstation with one 500MHz UltraSparc-IIe CPU. The L1 cache is 16 KB, with a 16-byte cache line size. The L2 cache has a capacity of 8 MB and its line size is 64 bytes. It has a TLB of 48 entries with 4 KB page size. The network interconnect of *sunwulf* is Gigabit Ethernet.

Figure 1 shows the performance (rate of sending/receiving data in MB/s) of programs using messages formed by vector and indexed datatypes on the *jazz* cluster. Figure 2 shows the performance of the same programs on the *sunwulf* cluster. On both clusters, when the message size is larger than cache size, the performance of the original MPICH2 implementation degrades sharply compared to the manual implementation for both vector and indexed datatypes. With the optimized implementation, this performance is in the same level as that of optimized manual codes. These figures also show that the overhead of optimized implementations is low.

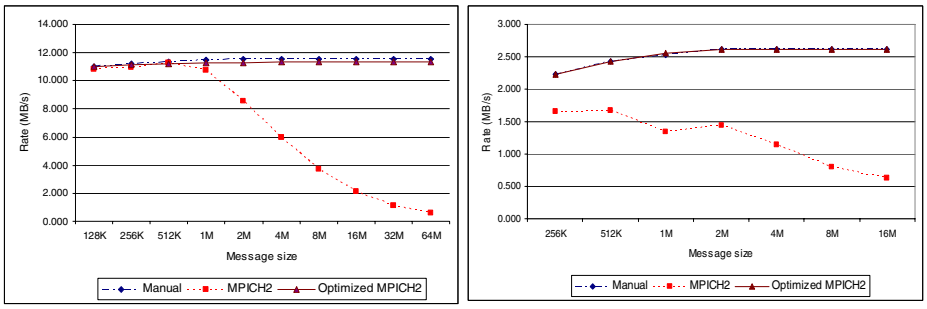

Fig. 1. Bandwidth measurements for vector (left) and indexed (right) datatype on jazz

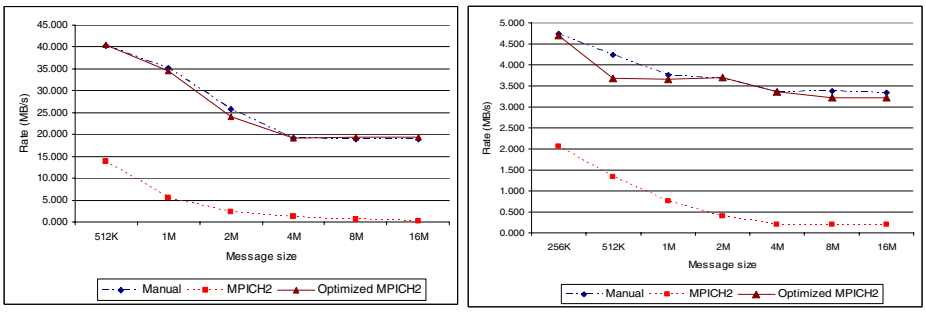

Fig. 2. Bandwidth measurements for vector (left) and indexed (right) on sunwulf

Figure 3 shows the performance of programs communicating messages formed using nested derived datatypes representing a 3D-cube on the *jazz* cluster and Figure 4 shows that on the *sunwulf* cluster. On both clusters, the original MPICH2 performs

similar to manual and optimized implementations for smaller data sizes. As the message size (size of 3D cube) becomes larger compared to the L2 cache size, the performance degrades for MPICH2, whereas the optimized implementation maintains superior performance similar to that of the optimized manual program.

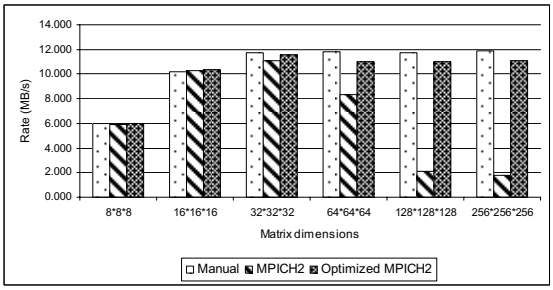

Fig. 3. Bandwidth measurements for the 3D-cube experiment on jazz

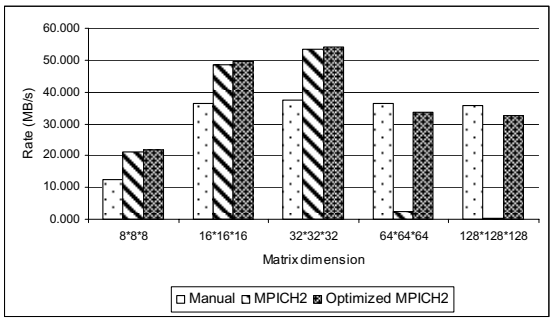

Fig. 4. Bandwidth measurements for the 3D-cube experiment on sunwulf

Figures 5 and 6 show the performance of the NAS MG benchmark on *jazz* and *sunwulf* clusters, respectively. We measured the execution time of the MG benchmark by using 4, 8 and 16 processors with B and C class workloads. The execution time with MPICH2 is higher than that of the original MG benchmark implementation (manual). With optimized MPICH2, the execution time is up to 8% (on average 6%) lower than that of manual implementation, and up to 25% (on average 13%) lower than that of unmodified MPICH2 on the *jazz* cluster. On the *sunwulf* cluster, for 8 and 16 processors, the execution time is up to 12% (on average 7.3%) less than that of the manual implementation. Here, manual implementation is the original NAS MG benchmark, which is not optimized for cache blocking and prefetching. Our optimized MPI derived datatype implementation benefits from using cache blocking in the nested datatypes in the MG benchmark.

Figures 7 and 8 show the performance (rate in MB/s) of the matrix transpose subroutine of NAS FT benchmark on *jazz* and *sunwulf* clusters, respectively. When

the message size is larger than the L2 cache size, the rate degrades severely for unmodified MPICH2 because of the large number of cache misses caused by poor data locality. The optimized MPICH2 implementation benefits from using cache blocking in this program. The performance gain is in the range of 50–60% on *jazz* cluster and 50–114% on the *sunwulf* cluster.

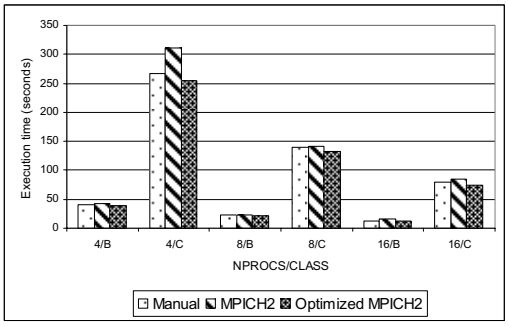

Fig. 5. Execution time of the NAS MG benchmark on jazz (left) and on sunwulf (right)

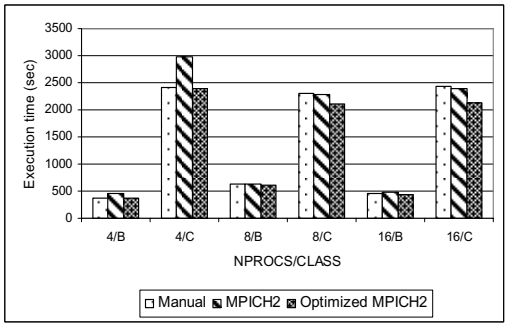

Fig. 6. Execution time of the NAS MG benchmark on sunwulf

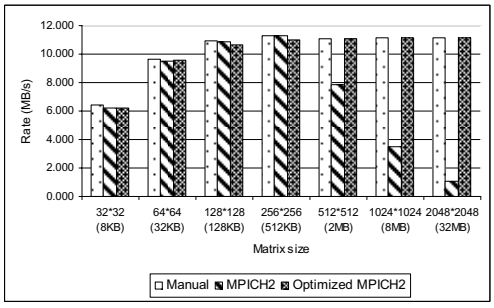

Fig. 7. Bandwidth measurements for matrix transpose experiment on jazz

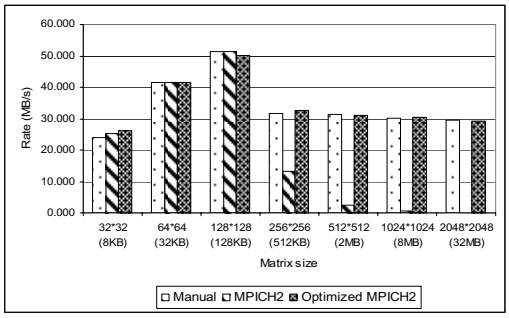

Fig. 8. Bandwidth measurements for matrix transpose experiment on sunwulf

4 Conclusions and Future Work

In this paper, we presented a technique to optimize the performance of MPI derived datatypes. Poor data access performance in dealing with noncontiguous data has been a major performance bottleneck of in packing and unpacking of MPI derived datatypes. Many optimization methods are available in the literature to optimize the data-access performance. However, predicting the optimization parameters with low overhead and automatically applying these optimization strategies is a challenging research issue. We developed models for predicting memory-access cost [2] that can help in dynamically applying optimizations. By combining optimization methods with a memory access model, we have introduced in this paper an approach to optimize memory performance automatically. The optimized implementation of MPI derived datatypes chooses packing templates that are optimized for advanced hierarchical memory systems of modern machines. These templates are parameterized with various architecture-specific parameters (for example, block size and TLB size), which are determined separately for different systems. By using these optimized templates, we obtained significantly higher performance than the existing MPICH2 implementation and manual packing/unpacking by the user. This result is significant because it will improve the performance of `MPI_Pack`/`Unpack` and MPI communication functions in many applications that use MPI derived datatypes in performing noncontiguous communication. We have shown that our optimized implementations are applicable on multiple architectures (Intel and Sun).

The optimizations described in this paper are not yet incorporated into the MPICH2 release, but we plan to do so. We are also looking at other applications of automatically selecting optimization parameters using the analytical prediction model. For example, in scientific applications, major portion of their run time is spent in executing loops. Using optimized templates can improve the performance of those loops. We are also working on incorporating prefetching strategies within PVFS [3] to improve the performance of data movement.

Acknowledgments. This work was supported in part by the National Science Foundation under NSF grants CNS-0509118, CNS-0406328, EIA-0224377, EIA-0130673, and in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

1. Surendra Byna, William Gropp, Xian-He Sun, and Rajeev Thakur, "Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost," In Proceedings of IEEE International Conference on Cluster Computing, December 2003.
2. Surendra Byna, Xian-He Sun, William Gropp and Rajeev Thakur, "Predicting Memory-Access Cost Based on Data-Access Patterns," In Proceedings of IEEE International Conference on Cluster Computing, September 2004.
3. Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur, "PVFS: A Parallel File System for Linux Clusters," In Proceedings of the 4th Annual Linux Showcase and Conference, pages 317--327, Atlanta, GA, 2000, USENIX Association.
4. William Gropp, Ewing Lusk, and Deborah Swider, "Improving the Performance of MPI Derived Datatypes," In Proceedings of the Third MPI Developer's and User's Conference, MPI Software Technology Press, pp. 25–30, March 1999.
5. M. Lam, Edward E. Rothberg, and Michael E. Wolf, "The Cache Performance of Blocked Algorithms," In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 63–74, April 1991.
6. Q. Lu, J. Wu, D. Panda and P. Sadayappan, "Applying MPI Derived Datatypes to the NAS Benchmarks: A Case Study," Technical Report OSU-CISRC-4/04-TR19, Ohio State University.
7. Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", Version 1.1, June 1995. http://www.mpi-forum.org/docs/docs.html.
8. T. Mowry and A. Gupta, "Tolerating Latency Through Software-controlled Prefetching in Shared-memory Multiprocessors," Journal of Parallel and Distributed Computing, Volume 12, Issue 2, June 1991.
9. H. Ogawa and S. Matsuoka, "OMPI: Optimizing MPI Programs using Partial Evaluation," In Proceedings of IEEE/ACM Supercomputing Conference, Pittsburgh, November 1996.
10. Ralf Reussner, Jesper Larsson Träff, and Gunnar Hunzelmann, "A Benchmark for MPI Derived Datatypes," In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting, volume 1908 of Lecture Notes in Computer Science, pages 10-17, 2000.
11. R. Ross, N. Miller, and W. Gropp, "Implementing Fast and Reusable Datatype Processing," In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting, volume 2840 of Lecture Notes in Computer Science, pages 404-413, 2003.
12. Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann, "Flattening on the Fly: efficient handling of MPI derived datatypes. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting, volume 1697 of Lecture Notes in Computer Science, pages 109-116, 1999.
13. Jiesheng Wu, Pete Wyckoff, Dhabaleswar Panda, "High Performance Implementation of MPI Derived Datatype Communication over InfiniBand," In Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004.