Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost

Surendra Byna[†] **William Gropp**[‡] **Xian-He Sun**[†] **Rajeev Thakur**[‡]

[†]Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
{renbyna, sun}@iit.edu

[‡]Math. and Comp. Science Division
Argonne National Laboratory
Argonne, IL 60439
{gropp, thakur}@mcs.anl.gov

Abstract

The MPI Standard supports derived datatypes, which allow users to describe noncontiguous memory layout and communicate noncontiguous data with a single communication function. This feature enables an MPI implementation to optimize the transfer of noncontiguous data. In practice, however, few MPI implementations implement derived datatypes in a way that performs better than what the user can achieve by manually packing data into a contiguous buffer and then calling an MPI function. In this paper, we present a technique for improving the performance of derived datatypes by automatically using packing algorithms that are optimized for memory-access cost. The packing algorithms use memory-optimization techniques that the user cannot apply easily without advanced knowledge of the memory architecture. We present performance results for a matrix-transpose example that demonstrate that our implementation of derived datatypes significantly outperforms both manual packing by the user and the existing derived-datatype code in the MPI implementation (MPICH).

1. Introduction

The MPI (Message Passing Interface) Standard is widely used in parallel computing for writing distributed-memory parallel programs [1,2]. MPI has a number of features that provide both convenience and high performance. One of the important features is the concept of derived datatypes. Derived datatypes 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 two ways. One way is to improve the data structures used to store derived datatypes internally in the MPI implementation, so that, in an MPI communication call, the implementation can quickly decode the information represented by the datatype. Research has already been done in this area, mainly in using data structures that allow a stack-based approach to parsing a datatype, rather than making recursive function calls, which are expensive [3,4]. Another area for improvement is to use optimized algorithms for packing noncontiguous data into a contiguous buffer in a way that the user could not do easily without advanced knowledge of the memory architecture. This latter area is the focus of this paper. To our knowledge, no other MPI implementations use memory-optimization techniques for packing noncontiguous data in their derived-datatype code (for example, see the results with IBM's MPI in Figure 8).

Interprocess communication can be considered as a combination of *memory communication* and *network communication* as defined in [5]. Memory communication (or memory copying) is the transfer of data from the user's buffer to the local network buffer (or shared-memory buffer) and vice versa. Network communication is the movement of data between source and destination network buffers. Much research has been devoted to improving the performance of networks with the assumption that network communication is the main contribution to communication cost. But with the rapid improvements in network technology, the proportion of these overheads has changed, and memory communication has become a significant factor in the overall communication. Therefore, limiting the cost of memory accesses can significantly improve the overall communication time, as we demonstrate in this paper. The key to improving the memory-access performance is to exploit advanced memory hierarchies in modern computer architectures. Doing so directly is difficult for users because they are often unaware of architectural details, and these details vary from machine to machine. We do the memory optimizations automatically at the library level in our memory-conscious implementation of derived datatypes.

The rest of this paper is organized as follows. Section 2 describes related work in this area. Section 3 presents a motivating example that illustrates the performance improvements possible with memory optimization. In Section 4, we quantify memory-communication costs by using the Memory-LogP model. Section 5 describes how we use memory-optimized algorithms to improve the performance of derived datatypes. Performance results are presented in Section 6, followed by conclusions and a discussion of future work in Section 7.

All the experiments reported in this paper, except for the results in Figure 8 with IBM's MPI, were performed on an SGI Origin2000 at the National Center for Supercomputing Applications. This machine has a cc-NUMA architecture and runs IRIX 6.5.14 as the operating system. Each node of the machine has two MIPS R10000 processors [6] running at 195 MHz. Each processor has a 32 KB two-way set-associative and two-way interleaved primary (L1) cache and a 4 MB off-chip secondary cache. The MIPS R10000 processor has two on-chip 32-bit registers to count 30 distinct hardware events. In our experiments, we measured the events related to total cycles (event 0), graduated instructions (event 17), memory data loads graduated (event 18), memory data stores graduated (event 19), L1 cache misses (event 25), and L2 cache misses (event 26). The MPI implementation we used is MPICH-1.2.5 with the shared-memory device.

2. Related Work

Prior research related to this paper falls into two main categories: improving the performance of MPI derived datatypes and improving the memory performance of algorithms.

Träff et al. describe a technique, called flattening on the fly, for improving the performance of derived datatypes [3]. This method aims to minimize the use of expensive recursive function calls to parse a derived datatype in the MPI implementation by using a stack-based approach. Gropp et al. [4] provide a taxonomy of MPI derived datatypes based on their memory-access patterns and describe how to implement those patterns efficiently, also using a stack-based approach. Neither of these efforts uses advanced memory-optimization techniques for packing derived datatypes as we do.

Much research has been performed by the compiler and algorithms community on improving memory-hierarchy performance by using techniques such as loop transformation, array padding, and cache blocking [7,8,9], and we use some of these optimizations in our optimized packing algorithms. Many compilers use some of these optimizations to improve code performance. However, longer compile times and dynamic behavior of data accesses limit the performance improvement that a compiler can obtain. Many software libraries have been developed, particularly for numerical software, that use advanced memory-optimization techniques, for example, the portable LAPACK library [10] and the ESSL and PESSL libraries on IBM machines [11]. ATLAS (Automatically Tuned Linear Algebra Software) is an approach for automatically generating optimized numerical software on machines with deep memory hierarchies [12].

3. Motivating Example

We use a matrix-transpose example to demonstrate the benefits of memory optimization and to demonstrate that a compiler by itself cannot perform all these optimizations; in other words, the code must be optimized in the first place. We compare the performance of two programs that perform matrix transpose (MT): an MPI program with derived datatypes in which the MPI implementation does its usual unoptimized packing of transferred data (leaving all optimizations to the compiler) and another MPI program in which we manually pack data in a memory-optimized fashion before communication, without using derived datatypes. For both programs, we isolate and measure the memory-communication cost by running the programs as a single-process application, and we measure the performance with different compiler optimizations.

The SGI MIPSpro compiler [13] offers a gamut of general-purpose and architecture-specific optimizations to improve performance, including loop-nest optimization, software pipelining, and interprocedural analysis. The `O2` optimization flag turns on all global optimizations, such as dead-code elimination, loop normalization, and memory-alias analysis. The `Ofast` flag turns on all optimizations related to loop nesting, such as loop unrolling, loop interchange, loop blocking, and memory prefetch.

The manual-packing program uses external array padding and cache blocking. In array padding, an array is extended in size to reduce the number of memory-system conflicts by adding a new column to the original array. Cache blocking aims to reuse as much as possible the data elements already loaded into the cache before they are replaced by new elements. We chose a blocking size such that the whole block fits into the cache. We chose five sizes of matrices: 512*512, 1024*1024, 2048*2048, 4096*4096, and 8192*8192, where each element of the array is an 8-byte word. We measured performance for the following scenarios:

- Simple MPI implementation of MT using derived datatypes
    a. compilation of MT using –O2 optimizations (default)
        i. –O2 compilation of MPICH derived datatype source code [sO2/mpiO2]
    b. compilation of MT using –Ofast optimizations
        i. –O2 compilation of MPICH derived datatype source code [sO2/mpiOfast]
        ii. –Ofast compilation of MPICH derived datatype source code [sOfast/mpiOfast]
- MT with manually optimized pack and unpack (default –O2 MPICH compilation)
    a. compilation of MT using –O2 optimizations (default) [mO2]

b. compilation of MT using –O2 optimizations + cache blocking [mO2cb]
c. compilation of MT using –Ofast optimizations [mOfast]
d. compilation of MT using –Ofast optimizations + cache blocking [mOfastcb]

Figure 1 shows the overall cost in cycles per memory reference (the lower the better) for both programs with various combinations of options of the SGI MIPSpro compiler. Predictably, the manual implementation of the packing algorithm outperforms the compiler-generated code for derived datatypes in MPICH. Code compiled with the default optimizations (-O2) performs the worst for all data sizes; code compiled with –Ofast performs better in most cases. Memory-optimized programming with cache blocking and array padding provides a speedup ranging from 380% to 550% for various data sizes versus the MPICH derived-datatypes version.

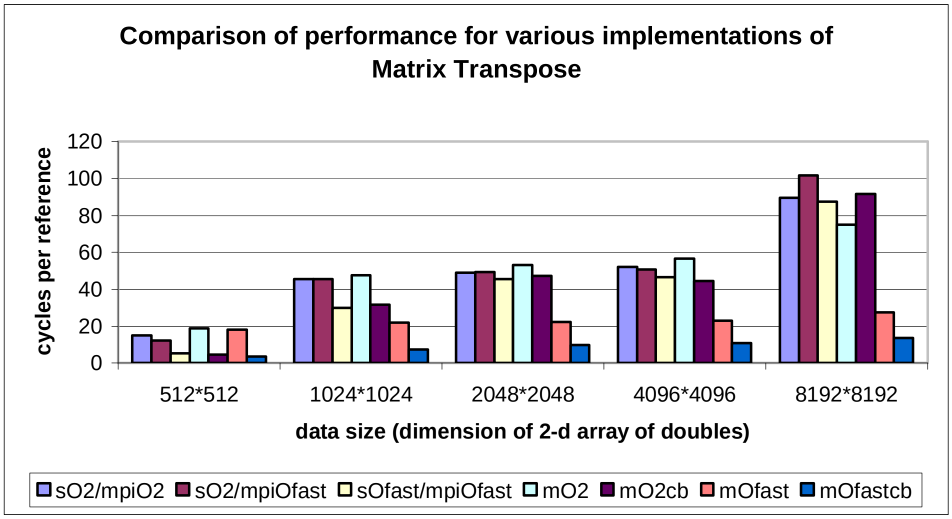

Figure 1. Performance gain with memory-conscious programming. The transpose operation with the use of manual array padding and cache blocking provides a speedup of 380% to 550% versus the original MPICH derived-datatypes version.

4. Quantifying Memory-Communication Cost

Parallel communication models such as LogP [14] focus on network communication, with limited consideration of memory communication. Recently, the LogP model was extended to incorporate memory-communication cost [5]. The Memory-LogP model formally characterizes the memory-communication cost under four parameters: *l*: the effective latency, defined as the length of time the processor is engaged in transmission or reception of a message due to the influence of data size (*s*) and distribution (*d*), *l=f(s,d)*; *o*: the overhead, defined as the length of time the processor is engaged in transmission or reception of an ideally distributed (contiguous) message (during this time, the processor cannot perform other operations); *g*: the gap, defined as the minimum time interval between consecutive message receptions at the processor (the reciprocal of *g* corresponds to the available per processor); and *P*: the number of processor/memory modules (point-to-point communication in the memory hierarchy implies *P*=1). Detailed information about the Memory-LogP model can be found in [5].

The memory-communication cost for sending a data segment depends on architectural parameters, such as cache capacity, and code characteristics, such as data distribution, as explained in the Memory-LogP model. In general, the overall communication cost includes data-collection overhead, the cost of data copying to the network buffer, the cost of data forwarding to the receiver (network-communication cost), and other costs added by the middleware implementation. When data distribution in memory is noncontiguous, the data is typically collected into a contiguous buffer before being copied to the network buffer. This process adds extra buffering overhead to the overall communication cost and is implementation dependent.

We can quantify the memory-communication overhead as follows. According to Memory-LogP (for *P*=1), we divide the overall communication cost into three parts: basic contiguous data-copying cost (o), memory-communication cost (l), and the network-communication cost (L). The memory-communication cost is further classified as data-packing overhead (l_p) and middleware-induced overhead (l_m). The middleware-induced overhead includes all other costs, such as extra buffer-copying cost, handshaking overhead between source and destination processes (if there is any), and load-imbalance costs.

$$Communication\_overhead = o + l_p + l_m + L$$

We measured each of these costs as follows. The basic data-copying cost (o) is the cost of copying a contiguous data between two buffers. The network-communication cost (L) is calculated by subtracting (o) from the cost of communicating a contiguous message between two processes. The cost of reading data noncontiguously from a buffer and writing it into a contiguous buffer includes data-packing cost (l_p) and the basic overhead (o). Subtracting (o) from this cost gives the data-packing cost (l_p). The middleware-induced cost (l_m) is calculated by subtracting the sum of other costs $(o + l_p + L)$ from the overall communication cost. The results for a matrix-transpose algorithm with derived datatypes are shown in Figure 2. It shows that the basic-copying cost (o) and network-communication cost (L) are relatively constant per data reference. In contrast, data-packing and middleware-induced costs grow significantly with data size. In this paper, we aim to reduce the data-packing cost of the overall memory-communication cost.

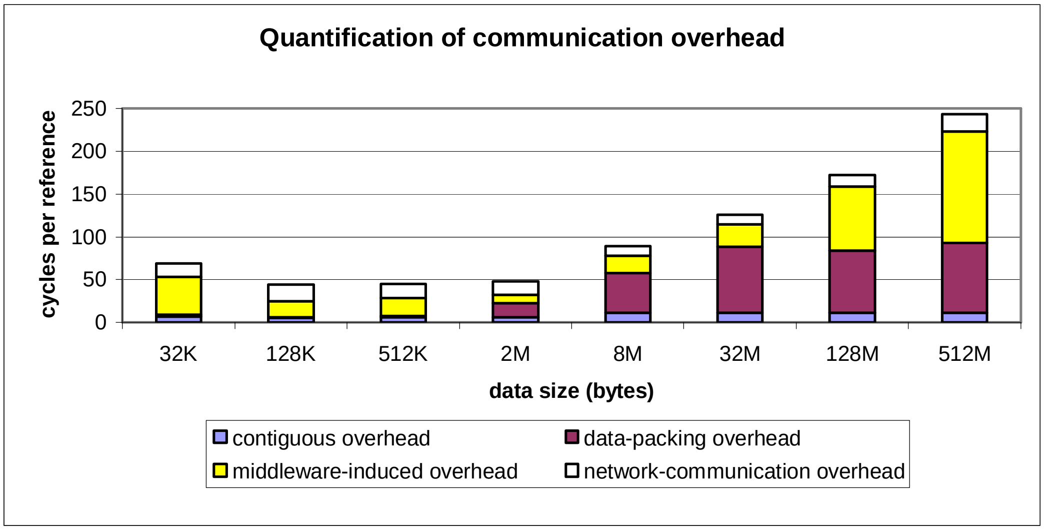

Figure 2. Memory-communication cost for a matrix-transpose algorithm is classified into basic contiguous overhead, data-packing cost, and middleware-induced cost. The network-communication cost is also shown.

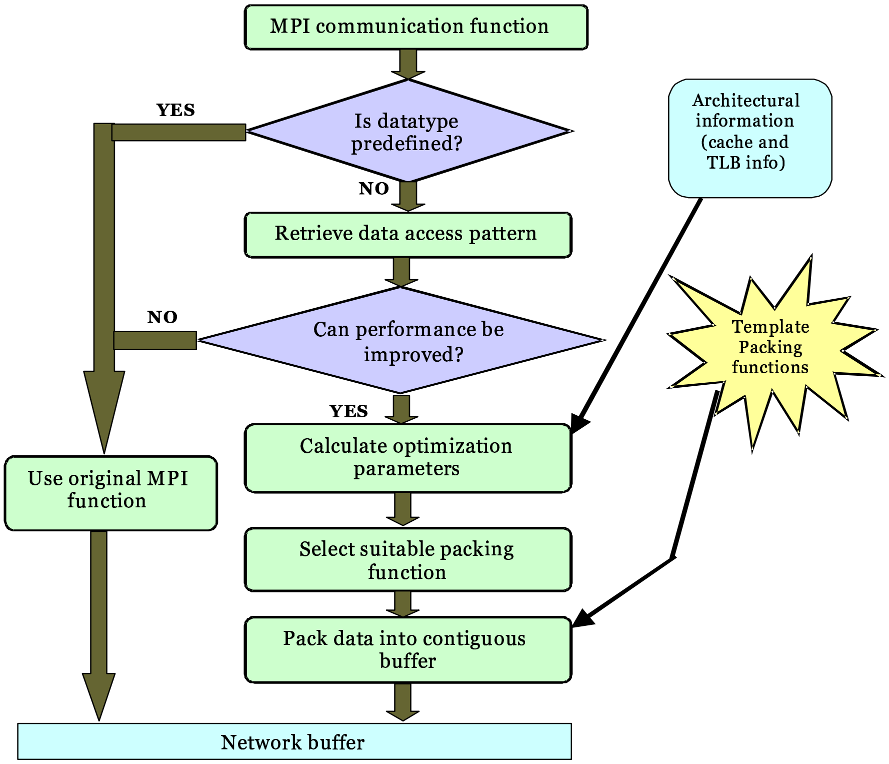
MPI communication function
YES
Is datatype predefined?
NO
Retrieve data access pattern
NO
Can performance be improved?
YES
Architectural information (cache and TLB info)
Template Packing functions
Calculate optimization parameters
Use original MPI function
Select suitable packing function
Pack data into contiguous buffer
Network buffer

Figure 3. Overview of memory-conscious optimization for communication at a sender

5. Optimizing MPI Derived Datatypes

In this section we describe how we automatically use memory-optimized packing algorithms to implement MPI derived datatypes.

5.1. Overview of Optimization Technique

Figure 3 illustrates the procedure for optimizing sends with derived datatypes. The profiling interface in MPI [1] provides a convenient way for us to insert our code in an implementation-independent fashion as follows. Every function in MPI is available under two names, `MPI_` and `PMPI_`. User programs use the `MPI_` version of the function, for example, `MPI_Send`. We intercept the user's call to `MPI_Send` by implementing our own `MPI_Send` function in which we do the datatype packing (if necessary) and then call `PMPI_Send` to do the data communication. As a result, application programs don't need to be recompiled; they need only to be relinked with our version of the MPI functions appearing before the MPI library in the link command line. We currently interpret derived datatypes by accessing the internal data structures used by MPICH. However, we plan to adopt an implementation-independent way by using the datatype-decoder functions from MPI-2, namely, `MPI_Type_get_envelope` and `MPI_Type_get_contents`. In our implementation of `MPI_Send,` we first determine whether the datatype passed is a basic (predefined) datatype or a derived datatype. If it is a basic datatype, we simply call `PMPI_Send` as no packing is needed. For a derived datatype, we first determine whether any performance improvement is possible for the access pattern represented by the datatype as described below.

5.2 Is Performance Improvement Possible?

Memory performance degrades significantly when the program cannot reuse the data already loaded in various levels of the memory hierarchy. We observe in Figure 2 that the data-packing overhead (l_p) increases after data size 2 MB. Cache reuse degrades when the number of cache lines required to be loaded for the working set of data is more than the available number of cache lines. Reuse of the Translation Look-aside Buffer (TLB) degrades when the number of page entries required is more than the number of entries the TLB can hold. The TLB typically contains a small number of entries (128 on the Origin2000), which result in more misses when the number of pages required is higher than this number. Therefore, to determine whether any performance improvement is possible, we use the metric of determining whether there will be any TLB misses for a naïve (unoptimized) data-packing algorithm for the given data size. For this purpose, we need to know the number of page entries that will be required to be loaded into the TLB. For a noncontiguous access with fixed block size and fixed stride, this number can be determined as follows.

Let W be the size in bytes of each contiguous block, S be the stride in bytes between the start of two consecutive contiguous blocks of data, n be the number of references to contiguous blocks of data in the innermost loop, P_s be the page size in bytes, and T_p be the maximum number of entries the TLB can hold. If $P_s \geq S$, each page contains $\lfloor \frac{P_s}{S} \rfloor$ references. The number of pages required R_p can be calculated as follows:

$$R_p = \frac{n}{\lfloor \frac{P_s}{S} \rfloor} \text{ if } P_s \geq S \text{, and } R_p = n \left\lceil W / P_s \right\rceil \text{ if } P_s < S .$$

If $R_p < T_p$, the entire data to be accessed, including the stride, can be mapped by TLB; that is, there are no TLB misses. In this case, we assume that no performance optimization is possible, and we simply call `PMPI_Send`. But if $R_p \geq T_p$ and if the access pattern represented by the datatype includes out-of-order accesses, some of the pages mapped would be replaced before they are completely used. In this case, we use our optimized packing algorithm that uses cache blocking to ensure better reuse of mapped pages.

5.3 Choosing a Block Size

After determining that performance improvement is achievable, performance-optimization parameters (such as block size for cache blocking) are calculated. Lam et al. [7] have shown that the block size has a significant effect on blocking algorithms. Choosing the optimal block size, however, is difficult. Temam et al. [15] propose an analytical approach to calculate block size by taking into consideration all three types of cache misses: compulsory, capacity, and conflict misses. This method has significant overhead in estimating the parameters accurately.

Instead of trying to find the optimal block size, we simply aim to choose a block size that avoids the worst performance. Specifically, because of the high cost of TLB misses [16] and the relatively small size of the TLB, we decided to choose a block size that minimizes TLB misses. In our implementation, we choose a block size that accommodates TLB mapping, noncontiguous array accesses, and the other variables in the program. We use half the TLB entries ($T_p / 2$) to map the block and the other half to accommodate the contiguous buffer and other loop variables. In other words, we use a block size that will consume half the entries in the TLB. We determine the TLB size for a given system by running a microbenchmark developed by Saavedra et al. [17]. The page size is determined by using the system command `getpagesize`.

5.4 Choosing a Packing Function

Based on the data-access pattern represented by a datatype, we choose a predefined packing function and plug in the memory-optimization parameters. To select a function, we classify access patterns into combinations defined by contiguous or noncontiguous accesses with fixed or variable block sizes and fixed or variable strides. For each of these combinations we use a predetermined packing function with architecture-dependent parameters. The data is packed into a contiguous buffer with the selected packing function, and we then call `PMPI_Send` with the contiguous buffer. Figure 4 shows the current implementation of `MPI_Send` in MPICH, and Figure 5 shows our implementation with the packing optimizations.

```
MPI_Send (data, datatype, dest)
{
      if (datatype is basic datatype)
      {
            Send (data) to the network buffer.
      }
      else (datatype is derived datatype)
      {
            /* MPI_Pack () cost is staggeringly high for large
               data sets and powers-of-2 dimension arrays */

            MPI_Pack (data, datatype, buffer);
            Send (buffer) to the network buffer.
      }
}
```

Figure 4. Current implementation of MPI_Send in MPICH

6. Performance Evaluation

This section presents performance results for the matrix-transpose example. We compare the performance of three cases: original MPICH with derived datatypes, original MPICH with manual (unoptimized) packing by the user (no derived datatypes), and derived datatypes with our optimized packing algorithm. To describe the transpose operation with a derived datatype, we use a datatype that is a vector of vectors (vectors of columns in an array). We use a ping-pong operation to measure performance. A process sends a message with a derived datatype representing the transpose, and the destination process receives it contiguously. The destination process then sends back the data with the same derived datatype to the first process, which receives it contiguously. The time is measured at the first process and halved to find the communication cost for one complete data transfer. We run a few iterations of the program and find the minimum time.

```
MPI_Send (data, datatype, dest)
{
        if (datatype is basic datatype)
        {
                PMPI_Send (data, datatype, source, dest);
        }
        else (datatype is derived datatpe)
        {
                packing_algorithm = Select_best_packing_algorithm (data,
                datatype);
                pack (packing_algorithm, data, datatype, buffer) ;
                PMPI_Send (data, datatype, dest);
        }
}

Select_best_packing_algorithm (data, datatype)
{
        if (data fits into cache/TLB)
        {
                packing_algorithm = PMPI_Pack (data, datatype);
        }
        else
        {
                calculate_optimization_params (datatype, system_info, &params);
                choose_packing_algorithm (params, data, datatype,
                &packing_algorithm);
        }
        return (packing_algorithm);
}

pack (packing_algorithm, data, datatype, buffer)
{
        if (packing_algorithm == PMPI_Pack)
        {
                PMPI_Pack (data, datatype, buffer);
        }
        else
        {
                /* Here come the template implementations for various
                   data-access patterns with optimized parameters */
        }
}
```

Figure 5. Memory-conscious implementation of MPI_Send

In the optimized packing algorithm, cache blocking is used only if the number of pages required to be loaded in the TLB is more than the available TLB entries. For the MIPS R10000 processor on the Origin2000, the number of TLB entries is 128, and the page size is 16 KB. For matrix transpose, the number of pages required is more than the available TLB entries for arrays of size larger than 512*512 double-precision numbers (data size is 2 MB).

Figure 6 shows the performance of the three cases in terms of the number of clock cycles per memory reference. For small data sizes, where we do not use cache blocking, the performance of the three methods is almost the same. But once the data size is 8 MB or larger, where cache blocking comes into effect, our optimized implementation significantly outperforms both original MPICH and manual packing, and the performance improvement is greater as the data size increases.

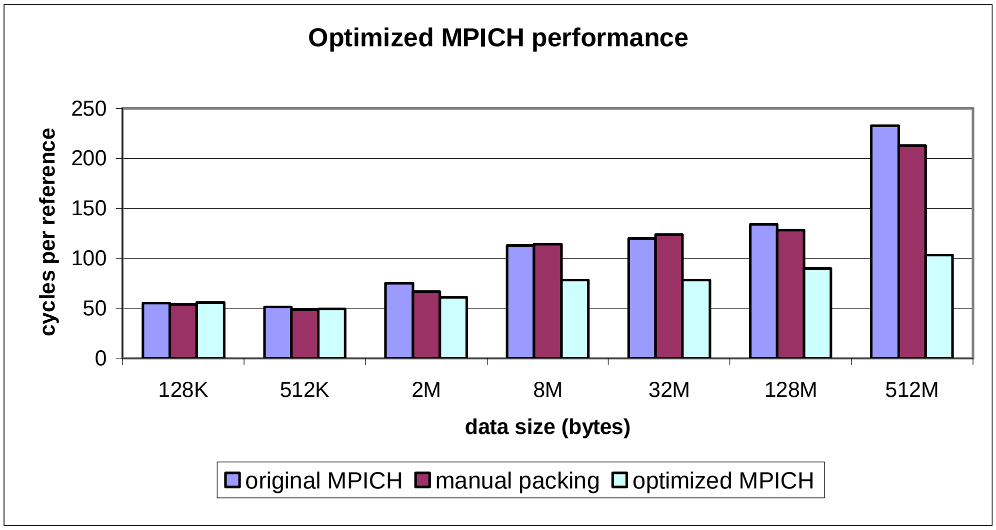

Figure 6. Performance improvement with optimized implementation of derived datatypes

Figure 7 shows the overall communication bandwidth achieved for the same example with original MPICH and the memory-optimized version. For higher data sizes, the bandwidth achieved by original MPICH decreases considerably, whereas the bandwidth with the memory-optimized version decreases only slightly and is as much as 85% higher than the original MPICH bandwidth.

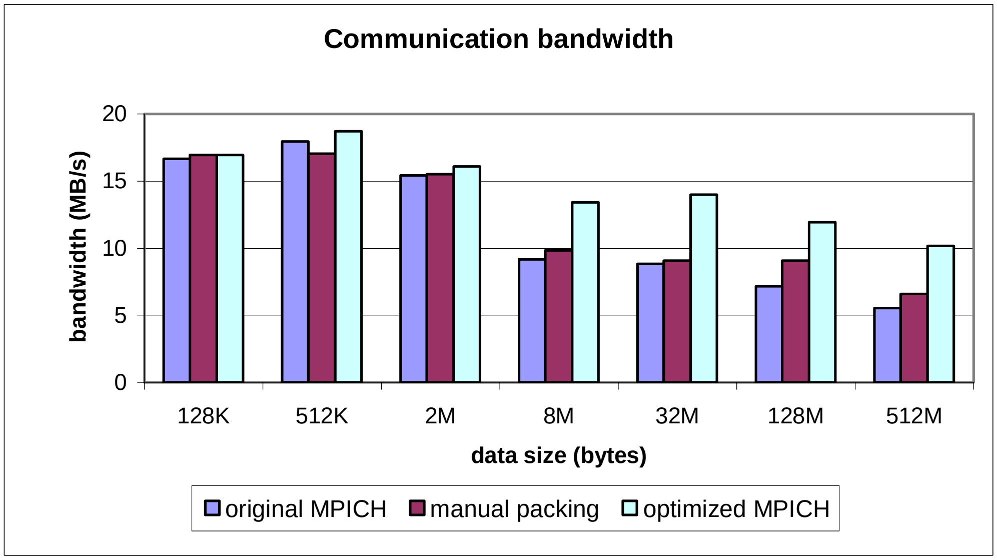

Figure 7. Bandwidth improvement with the optimized implementation

To see how memory-optimized packing performs compared with derived datatypes in a vendor MPI implementation, we ran some experiments on the IBM SP at the San Diego Supercomputer Center. We used IBM's MPI to run three versions of the matrix-transpose program: derived datatypes, manual unoptimized packing, and manual memory-optimized packing. The results are shown in Figure 8. For large data sizes, the program that uses manual memory-optimized packing takes significantly lower time than both the derived-datatypes version and the one with manual unoptimized packing. These results demonstrate that vendor MPI implementations also stand to gain by using memory-optimized packing.

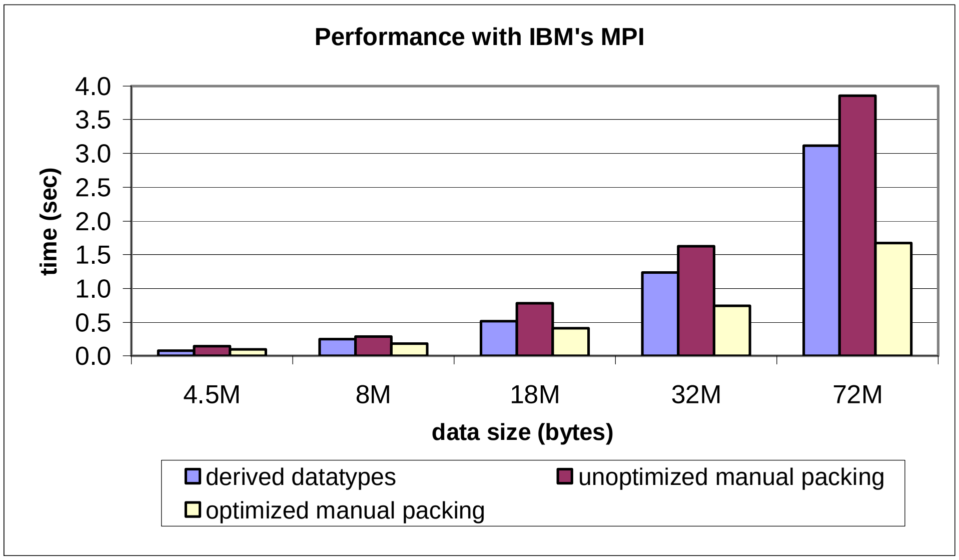

Figure 8. Performance of matrix transpose with IBM's MPI

7. Conclusions and Future Work

We have described a technique for improving the performance of MPI derived datatypes by using packing algorithms that are optimized for advanced hierarchical memory systems on modern machines. The optimized algorithms are selected automatically in the implementation of derived datatypes. The algorithms 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 algorithms, we obtained significantly higher performance than both the MPI implementation and manual packing by the user. This result is significant because it will improve the communication performance of many applications that perform noncontiguous communication.

We plan to extend this work in a number of areas. We will extend the optimizations to include loop optimizations, such as loop interchange and loop unrolling, as well as specific optimizations to reduce L1 and L2 cache misses. We will incorporate this datatype-optimization scheme in MPICH2, which is an all-new implementation of MPI that will eventually replace MPICH-1. MPICH2 uses more efficient data structures and a stack-based method for implementing derived datatypes, compared with the recursive-function-call approach used in MPICH-1. The new scheme is an extension of the work described in [4]. We believe that the new stack-based data structures combined with the memory-optimized data packing described in this paper will result in a very high performance implementation of derived datatypes.

We also plan to study the performance on other systems and with other applications, such as FFT, unstructured mesh codes, 3D nearest-neighbor communication, and array redistribution.

Acknowledgments

This work was supported 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, and in part by a grant from the Office of Advanced Simulation and Computing, National Nuclear Security Administration, U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 1.1," `http://www.mpi-forum.org/docs/docs.html`, June 1995.

[2] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 2nd edition, 1999.

[3] Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann, "Flattening on the Fly: Efficient Handling of MPI Derived Datatypes," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Vol. 1697, Springer, pp. 109–116, 1999.

[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] Kirk W. Cameron, Xian-He Sun, "Quantifying Locality Effect in Data Access Delay: Memory logP," to appear in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[6] National Center for Supercomputing Applications, "Understanding Performance on the SGI Origin 2000," `http://archive.ncsa.uiuc.edu/SCD/Perf/Tuning/Tips/Tuning.html`.

[7] Monica 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.

[8] M. Kandemir, J. Ramanujam and A. Choudhary, "Cache Locality by a Combination of Loop and Data Transformations," *IEEE Transactions on Computers (TC)* 48(2): 159–167, February 1999.

[9] Gabriel Rivera and Chau-Wen Tseng, "Locality optimizations for multi-level caches," in *Proceedings of SC99: High-Performance Networking and Computing*, November 1999.

[10] LAPACK – Linear Algebra Package, `http://www.netlib.org/lapack.`

[11] IBM Corp. Engineering Scientific Subroutine Library (ESSL), `http://www-1.ibm.com/servers/eserver/pseries/library/sp_books/essl.html.`

[12] R. Clint Whaley, Antoine Petitet, and Jack Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing*, 27(1–2):3–25, 2001.

[13] SGI MIPSpro family compilers, `http://www.sgi.com/developers/devtools/languages/mipspro.html`.

[14] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *Proceedings of Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12, May 1993.

[15] O. Temam, C. Fricker, and W. Jalby, "Cache Awareness in Blocking Techniques," *Journal of Programming Languages*, 1998

[16] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström, "Recency-based TLB Preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 117–127, June 2000.

[17] R. H. Saavedra , R. S. Gaines, and M. J. Carlton, Micro Benchmark Analysis of the KSR1, in *Proceedings of Supercomputing '93*, pp. 202–213, December 1993.