Applying MPI Derived Datatypes to the NAS Benchmarks: A Case Study *

Qingda Lu Jiesheng Wu Dhabaleswar Panda P. Sadayappan
Computer and Information Science,
The Ohio State University
{luq, wuj, panda, saday}@cis.ohio-state.edu

Abstract

MPI derived datatypes are a powerful method to define arbitrary collections of non-contiguous data in memory and to enable non-contiguous data communication in a single MPI function call. In this paper, we employ MPI datatypes in four NAS benchmarks (MG, LU, BT, and SP) to transfer non-contiguous data. Comprehensive performance evaluation was carried out on two clusters: an Itanium-2 Myrinet cluster and a Xeon InfiniBand cluster. Performance results show that using datatypes can achieve performance comparable to manual packing/unpacking in the original benchmarks, though the MPI implementations that were studied also perform internal packing and unpacking on non-contiguous datatype communication. In some cases, better performance can be achieved because of the reduced costs to transfer non-contiguous data. This is because optimizations in the MPI packing/unpacking implementations can be easily overlooked in manual packing and unpacking by users.

Our case study demonstrates that MPI datatypes simplify the implementation of non-contiguous communication and lead to application code with portable performance. We expect that with further improvement of datatype processing and datatype communication such as [10, 24], datatypes can outperform the conventional methods of non-contiguous data communication. Our modified NAS benchmarks can be used to evaluate datatype processing and datatype communication in MPI implementations.

1. Introduction

The MPI (Message Passing Interface) Standard [7, 17] has evolved as a *de facto* parallel programming model for distributed memory systems. As one of its most important features, MPI provides a powerful and general way of describing arbitrary collections of data in memory in a compact fashion. The MPI standard also provides run time support to create and manage such MPI derived datatypes. Other common operations such as regular message passing functions, remote memory access (RMA), and MPI I/O operations can use these user defined datatypes to transfer data with arbitrary layouts.

In principle, there are two main goals in providing derived datatypes in MPI. First, MPI derived datatypes are expected to become a key aid in application development. Typically MPI derived datatypes allow users to have concise representations of many commonly used data layouts [11, 20] such as strided data, indexed data, and the lower triangular portion of a matrix. MPI applications such as (de)composition of multi-dimensional data volumes [2, 8] and finite-element codes [6] often need to exchange data with algorithm-related layouts between two processes. Derived datatypes can be used in these applications to facilitate the development. Second, MPI derived datatypes provide opportunities for MPI implementations to optimize datatype communication. The MPI implementations can either use efficient memory copy algorithms/operations [10, 6, 20] or take advantage of advanced network features [23, 24] to provide high performance non-contiguous data communication.

In practice, however, to transfer non-contiguous data, users often bear the data layouts in mind and pack non-contiguous data into a contiguous buffer at the transmission side. On the receive side, data is first received in a contiguous buffer and then unpacked into user buffers. We term this method *user packing/unpacking*. Despite of requiring significant efforts from users, this method has been widely used in many applications due to the poor performance of traditional MPI implementations with derived datatypes [6, 11, 21, 23].

In recent years, the advent of high performance transport protocols and networking technologies has reduced the gap between the network and the memory subsystems. Some emerging network technologies such as InfiniBand [12] have been able to provide performance comparable to that of the memory system. This trend causes the memory copy cost in the packing/unpacking approach become increasingly significant in transferring non-contiguous data. On the other hand, datatype processing and datatype communication have been improved over years [11, 10, 13, 20, 6, 23, 24]. Applications designed using the user packing/unpacking approach can not take advantage of these improvements.

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052, #CCR-0204429, and #CCR-0311542.

In this paper, we try to answer the following questions:

- Can the performance benefits of MPI datatypes be exploited in MPI applications?
- Do datatypes really ease application development?
- Is the manual packing and unpacking always efficient?

To answer these questions, we employ datatype communication in the NAS Parallel Benchmarks [3]. The NAS benchmarks are derived from computational fluid dynamics code and have gained wide acceptance as a standard indicator of supercomputer performance. Non-contiguous data transfers occur commonly in the NAS benchmarks. However, not surprisingly, the user packing/unpacking approach is used. Our main objectives in this paper are: (1) To study the impact of using derived datatypes on developing scientific applications; (2) To evaluate the impact of using derived datatypes on the performance of the NAS benchmarks; and (3) To provide a set of benchmarks to evaluate MPI implementations with respect to the processing and communication of derived datatypes.

Non-contiguous data communication occurs in the MG, LU, BT and SP benchmarks. We applied derived datatype to these four benchmarks without any change of their algorithms. We performed comprehensive performance evaluation of these modified benchmarks on two cluster systems: an Itanium-2 cluster with Myrinet [5] and a Xeon cluster with InfiniBand [12]. Compared to the performance of the original benchmarks, the performance of the modified NAS benchmarks with datatypes was found to be comparable. In some cases, better performance was achieved due to the optimizations used in the MPI implementations which are easily missed by users in their packing/unpacking. Our experience and performance results demonstrate that using derived datatypes in MPI applications is an easy-to-use way to transfer non-contiguous data with performance comparable to the user packing/unpacking approach.

The rest of the paper is organized as follows. Section 2 presents an overview of MPI derived datatype communication. Section 3 describes how to employ datatypes into the NAS parallel benchmarks. The performance results are presented in Section 4. We examine related work in Section 5 and draw our conclusions and discuss possible future work in Section 6.

2 Overview of MPI Derived Datatype

MPI provides basic pre-defined data types such as MPI_REAL and MPI_INTEGER. Using these basic data types, only contiguous buffers containing a sequence of elements of the same type can be involved in MPI communication and I/O operations. This is too restrictive. Applications often want to pass messages that contain values with different data types (e.g., an integer count, followed by a sequence of real numbers) and non-contiguous data (e.g., a sub-block of a matrix). One solution is that users bear the data layouts in mind and pack non-contiguous data into a contiguous buffer at the send side and unpack it back at the receiver side. Instead, MPI provides a mechanism, called *Derived Datatype Communication*, to specify arbitrary collections of data in memory concisely. In the rest of this paper, we use derived datatypes and datatypes interchangeably. An MPI datatype is an opaque object that specifies two things: (1) a sequence of basic data types and (2) a sequence of displacements [16]. MPI defines a set of functions to dynamically construct and destroy various kinds of datatypes, such as *vector, hvector, distributed array (darray)*, and so on.

A user can conveniently use these calls to construct datatypes needed by the application algorithms. Let us consider a simple example here. Suppose we want to send one or more columns in a two-dimensional 4096×4096 integer array from one process to another process. A derived datatype can be defined using `MPI_Type_vector(4096, x, 4096, MPI_INT, &newtype)`, where x is the number of columns. MPI also provides run time support to use the derived datatypes in other functions. For example, the datatype, *newtype*, can be used in other calls such as `MPI_Send()` and `MPI_Recv()` directly.

MPI datatype communication involves datatype processing and non-contiguous data communication (in this paper, unless stated otherwise, we refer to datatypes as noncontiguous datatypes). The datatype processing and data communication are implementation specific. We focus on the MPICH implementation since it is one of the most popular MPI implementations. MPICH uses internal packing and unpacking techniques in its implementation. This implementation may not be the most efficient one since two extra copies are involved. However, it has few requirements to the datatype processing component and the underlying network communication layer [24]. The MPICH developers have also optimized packing and unpacking algorithm in their MPICH implementation [11, 10, 20]. We expected that this implementation would be able to offer performance comparable to that of the user packing/unpacking approach.

3 Employing MPI Derived Datatypes in the NAS Benchmarks

The NAS parallel benchmarks [4, 3] are a set of programs designed as a part of the NASA Numerical Aerodynamic Simulation (NAS) program originally to evaluate supercomputers. They mimic the computation and data movement characteristics of large-scale computations. The NAS parallel benchmark suite consists of five kernels (EP, MG, FT, CG, IS) and three pseudo applications (LU, SP, BT) programs. Our study is based on the NPB 2.3 implementation written in MPI. There are five classes of NAS benchmarks in NPB 2.3, each of which is characterized by the number of elements in its finest grid and the number of iterations to be performed. There are three classes A, B and C, which are production grade problem sizes. The other two classes S and W are designed for developing and debugging purposes.

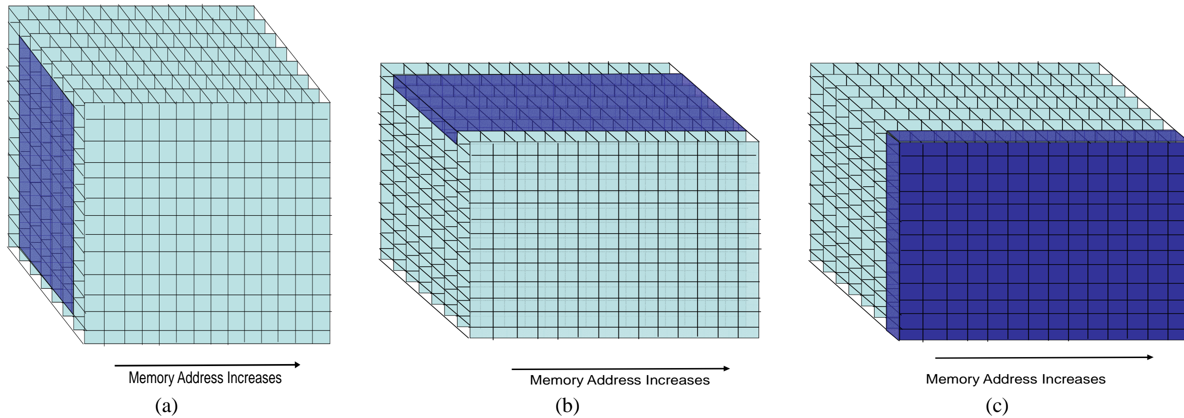

Figure 1. Data Layout in MG Datatype Communication

Non-contiguous communication occurs commonly in the NAS benchmarks, however, no derived datatypes are used in the NAS benchmarks [22]. To transfer non-contiguous data, the NAS benchmarks explicitly pack/unpack non-contiguous data to/from dedicated contiguous buffers. This happens in MG, LU, SP and BT. To the best of our knowledge, there is no user packing/unpacking in the other four benchmarks. In MG, LU, SP and BT, we replaced the combination of contiguous communication calls and manual packing/unpacking in the original benchmarks with MPI derived datatype communication calls. Neither the number of messages nor the content of each message was changed, though the data layouts of messages could have been different compared to the original version. Other additional techniques such as overlapping communication with computation to improve performance were also not applied because we wanted to avoid a radical departure from the original implementation. Therefore, it is meaningful to compare our results with those of the original implementation to evaluate the impact of datatypes.

3.1 MG Benchmark

The MG (multi-grid) benchmark solves Poisson's equation in 3D using a multi-grid V-cycle. The multi-grid benchmark carries out computation at a series of levels and each level of the V-cycle defines a grid at a successively coarser resolution. This test requires a power-of-two number of processors. The partitioning of the grid onto processors occurs such that the grid is successively halved, starting with the z dimension, then the y dimension and then the x dimension, and repeating until all power-of two processors are assigned. The NPB 2.3 code uses a three-step dimensional exchange algorithm to satisfy boundary conditions. In addition to this, point-to-point communication is used in the parallel implementation of these stencils to update boundary values for each dimension that is distributed. `MPI_Allreduce` is also used to concatenate the data and then replicate the result to each process, The time to perform this operation is negligible.

In MG, all non-contiguous data transfers are implemented as packing-then-send or receive-then-unpacking. The subroutine, `comm3`, which exchanges sub-domain boundary data, dominates 99% of the communication time. Therefore, our discussion focuses on it. In this subroutine, six messages are sent and six messages are received by each process, both with two messages in each of the three coordinate dimensions. In the hope that data packing can be overlapped, each process first initializes two non-blocking receives, then packs boundary data to a buffer and sends them using two blocking sends. It waits for the completion of receive operations, and then unpacks the received data to finish the boundary value exchange procedure.

The sub-domain boundary data have three kinds of layouts as depicted by Figure 1. We construct three datatypes accordingly. Other modifications include removing data packing/unpacking code and employing derived datatypes in all communication operations.

3.2 LU Benchmark

The LU benchmark is a simulated CFD application which uses symmetric successive over-relaxation (SSOR) to solve a block lower triangular-block upper triangular system of equations resulting from an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions. This code requires a power-of-two number of processors. A 2-D partitioning of the grid onto processors occurs by halving the grid repeatedly in the first two dimensions, alternately x and then y, until all power-of-two processors are assigned, resulting in vertical pencil like grid partitions on the individual processors. Unlike NPB 2.0 which does diagonal based relaxation which incurs a large number of communications of 40-byte messages, NPB 2.3 does column based relaxation. This improvement significantly reduces the communication cost of LU.

In LU, non-contiguous data transfers are achieved by point-to-point communication calls with manual packing/unpacking. Two communication subroutines `exchange_1` and `exchange_3` are involved in the SSOR

procedure. In exchange_1 which dominates the communication time, there are two two kinds of messages. One is non-contiguous as shown in Figure 2. The data layout is a vector in which the block length is five words. Another is contiguous. The message size is the the same size of the non-contiguous message.

Figure 2. Data Layout in LU Datatype Communication

3.3 BT and SP Benchmarks

SP and BT are simulated CFD applications that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of the Navier-Stokes equations. The BT code solves block-tridiagonal systems of 5x5 blocks; the SP code solves scalar pentadiagonal systems resulting from full diagonalization of the approximately factored scheme. Both have a similar structure.

The NPB 2.3 implementations of SP and BT solve these systems using a multi-partition scheme. In the multi-partition algorithm, each process is responsible for several disjoint sub-blocks of points ("cells") of the grid. The cells are arranged such that for each direction of the line solve phase, the cells belonging to a certain processor will be evenly distributed along the direction of solution. This allows each processor to perform useful work throughout a line solve, instead of being forced to wait for the partial solution to a line from another processor before beginning work. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore the granularity of communications is kept large and fewer messages are sent. Both SP and BT require a square number of processes.

Our discussion is focused on the BT benchmark. BT's communication patterns are mainly noncontiguous. Without changing BT's communication patterns, we can only modify part of noncontiguous communications using derived datatypes. Our modifications include defining several deeply-nested datatypes because of the complicated message shapes in the `copy_faces` procedure and replacing the original non-contiguous data transfer code with datatype communication calls.

4 Performance Results

In this section, we present performance results of the modified NAS benchmarks with derived datatypes on two typical cluster systems. We compare these results with those of the original NAS benchmarks. Unless stated otherwise, the unit megabytes (MB) in this paper is an abbreviation for 2^{20} bytes, or 1024×1024 bytes.

4.1 Experimental Setup and Methodology

We conducted our performance evaluations on the following two clusters.

Cluster 1 : A cluster of 128 nodes (the maximum number of nodes used by the experiments is 16), each with dual 900MHz Intel Itanium 2 processors with 256 KB L2 cache, 1.5MB L3 cache. Each node has a Myrinet 2000 interface card placed in a PCI-X 64-bit 66MHz bus. The Front Side Bus (FSB) runs at 266MHz. The physical memory is 4 GB of multi-channel PC2100 DDR-SDRAM memory. The kernel version is Linux 2.4.21smp. This cluster is provided by Ohio Supercomputer Center. We refer to this cluster as *IA64-Myrinet cluster*.

Cluster 2 : A cluster of 8 SuperMicro SUPER X5DL8-GG nodes, each with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, PCI-X 64-bit 133 MHz bus, and connected to Mellanox InfiniHost MT23108 DualPort 4x HCAs. The nodes are connected using the Mellanox InfiniScale 24 port switch MTS 2400. The kernel version used is Linux 2.4.22smp. The InfiniHost SDK version is 3.0.1 and HCA firmware version is 3.0.1. The Front Side Bus (FSB) runs at 533MHz. The physical memory is 1 GB of PC2100 DDR-SDRAM memory. We refer to this cluster as *IA32-IBA cluster*.

On the IA64-Myrinet cluster, the Intel Fortran 7.1 compiler was used. On the IA32-IBA cluster, the Intel Fortran 7.0 complier was used. The compiler flag was `-O3`. In the following subsections, we show the performance results of both the original and the modified NAS benchmarks (MG, LU, BT, and SP) on both clusters. For each benchmark, nine combinations between 4, 8/9, and 16 processes and A, B, and C classes are carried out. Since we only have 8 physical nodes in the IA32-IBA cluster, we run 2 processes on one physical node to use both processors for test cases which need more than 8 processes. On the IA64-Myrinet cluster, we run only one process per physical node for all test cases. We use the CPU cycle counter to obtain timing information, the total instrumentation overheads are small enough to be negligible.

4.2 MPI and Memory Performance

On the Itanium-2 Myrinet cluster, we use mpich-1.2.4..8a-gm (called *MPICH-GM* in the rest of paper). In the Xeon IBA cluster, we use MVAPICH-0.9.2 [18]. Figures 3 and 4 show the latency and bandwidth results of these two MPI implementations on two clusters. The details of latency and bandwidth tests can be found at [18].

We also tested the memory copying bandwidth in both clusters. In the memory copying test, two buffers, each 20 MBytes, are allocated. We copy data sequentially from one buffer to another buffer. The reported bandwidth is **1200** MBytes/sec on the IA64-Myrinet cluster, and **810** MBytes/sec on the IA32-IBA cluster.

4.3 Performance of MG Benchmark

Figure 5 shows performance results of MG on the IA64-Myrinet cluster. In this figure, we use *old* to refer to the

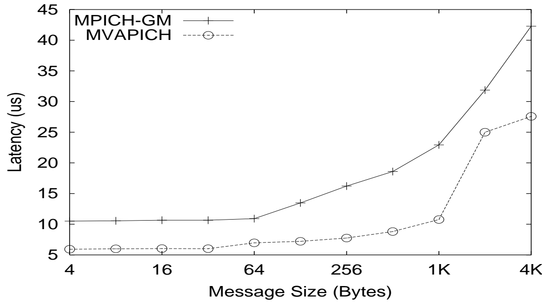

Figure 3. Latency of MPICH-GM and MVAPICH on Two Respective Clusters.

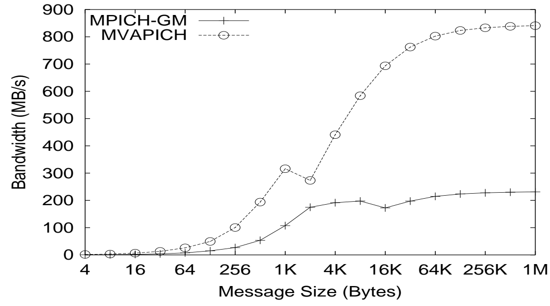

Figure 4. Bandwidth of MPICH-GM and MVAPICH on Two Respective Clusters.

original NAS benchmark, and *datatype* to refer to the modified version NAS benchmark using datatypes. These two legends are also used in the following Subsections. For all combinations, the datatype version outperforms the original version by up to 20% in many cases. For example, with 4 processes, an improvement from 12% to 20% can be achieved for all classes on both clusters. This improvement was unexpected because in both MPICH-GM and MVAPICH-0.92, internal packing and unpacking are deployed in their implementations to perform non-contiguous datatype communication. We tracked down the source of this dramatic improvement, and found that it mainly came from the elimination of a redundant initialization on a communication buffer in the datatype version. To show how much benefit comes from this elimination instead of using datatypes, we also removed this redundant initialization from the original version. We call this modified MG as *modified old* version in the plot. Compared to the old version, the modified old version only removes three lines from 1150 to 1152 in `mg.f`. After this change, the datatype version still outperforms the modified old version, but with much less improvement by up to 5%. This is mainly due to different packing and unpacking algorithms used by the MG developers and the MPICH designers. In both MPICH-GM and MVAPICH, appropriate optimizations are applied to pack and unpack messages [11, 10]. In general, these optimizations are prone to being overlooked in manual packing/unpacking.

Figure 6 shows performance results of MG on the IA32-IBA cluster. It can be observed that the same patterns as those on the IA64-Myrinet cluster also appear.

In the old version, the cost to transfer non-contiguous data includes two parts: the memory copy cost in the manual packing and unpacking; and the communication cost. In the datatype version, the cost of transferring non-contiguous data is reflected completely in the datatype communication cost. There is no explicit (un)packing in the datatype version. Figure 7 shows the breakdown of the communication cost and the (un)packing cost in the above three mentioned versions on both clusters. We choose class B with 4 processes as an example. In the plot, we use *comm* to represent the communication cost and *(un)packing* to represent the (un)packing cost. The (un)packing cost on the IA32-IBA cluster is higher than that on IA64-Myrinet cluster due to the lower memory bandwidth on the IA32-IBA cluster, as mentioned in Section 4.2. The communication cost on the IA32-IBA cluster is lower than that on IA64-Myrinet cluster due to the high communication performance provided by the InfiniBand network, as shown in Figures 3 and 4. It can be observed that using datatypes one can achieve slightly better performance than using manual packing and unpacking plus contiguous communication in the MG benchmark.

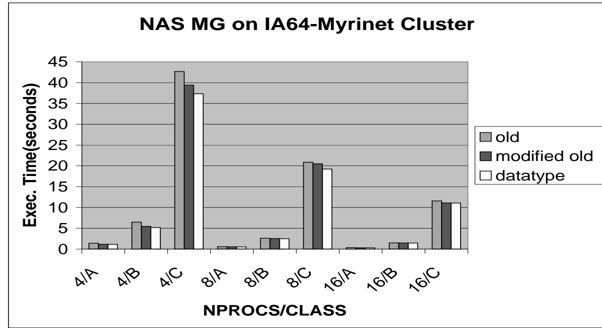

Figure 5. MG on IA64-Myrinet Cluster.

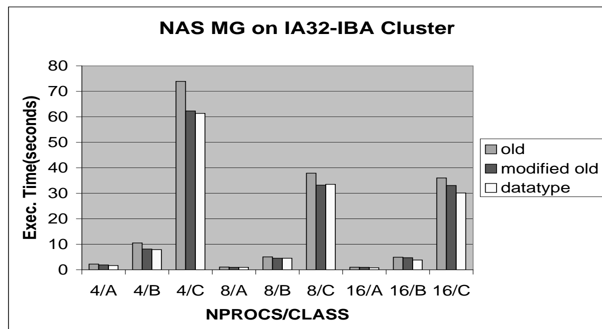

Figure 6. MG on IA32-IBA Cluster.

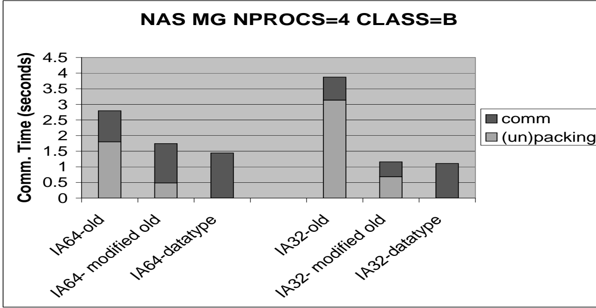

Figure 7. Cost to Transfer Non-Contiguous Data in MG.

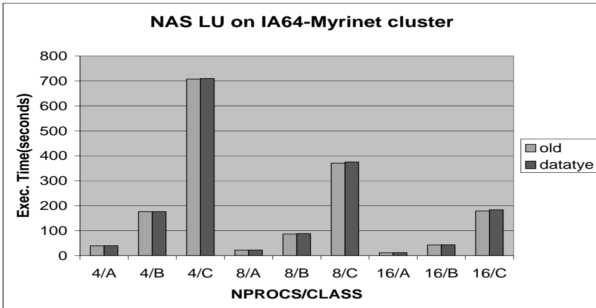

Figure 8. LU on IA64-Myrinet Cluster.

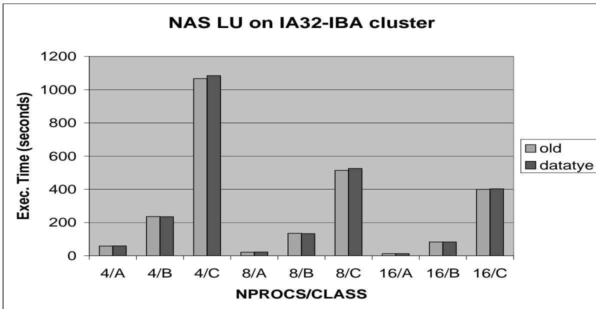

Figure 9. LU on IA32-IBA Cluster.

4.4 Performance of LU Benchmark

Figures 8 and 9 show performance results of LU on the IA64-Myrinet and IA32-IBA clusters. The performance of both the datatype version and the original version is very close. The most used datatypes in this benchmark are quite simple, as mentioned in Section 3.2. The datatype processing overhead is negligible. Figure 10 shows the communication cost and the (un)packing cost in the test case of class B with 4 processes.

4.5 Performance of BT Benchmark

Figures 11 and 12 show performance results of BT on the IA64-Myrinet and IA32-IBA clusters. The performance of both the datatype version and the original version is very close. This is also expected due to the internal packing and

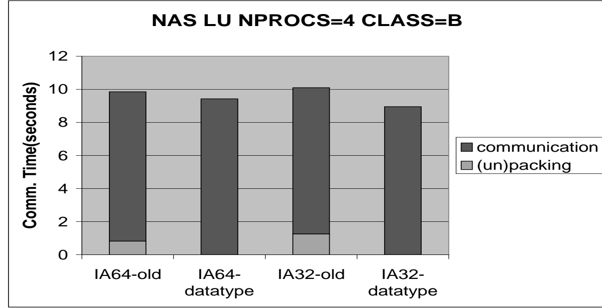

Figure 10. Cost to Transfer Non-Contiguous Data in LU.

unpacking. Note that we could not run class C with 4 processes on the IA32-IBA cluster due to its limited memory capacity.

Figure 13 shows the communication cost and the (un)packing cost in the test case of class B with 4 processes. As mentioned in Section 3.3, the derived datatypes we employed in BT benchmark have complicated nested structures. Presumably the datatype processing cost should be high. However, the results in Figures 13 indicate that the datatype version outperforms the original version in terms of the total cost of transferring non-contiguous data. This is largely because in the original version, the manual packing/unpacking code has loop structures which incur bad cache behavior. The common practice of writing packing/unpacking code is to have a buffer pointer incremented by one after each iteration. This introduces a data dependency between back-to-back iterations which hinders loop transformations. Note that as mentioned in Section 3.3, some non-contiguous communications still use manual packing/unpacking in our modified BT benchmark, therefore, their overhead is also shown in Figure 13.

In the case of class B with 4 processes, using datatypes can reduce the costs to transfer non-contiguous data by 10% on the IA64-Myrinet cluster and by 25% on the IA32-IBA cluster. However, because the communication time is only 2% of the total executime time, we do not see much improvement in terms of the total execution time using datatypes.

4.6 Performance of SP Benchmark

Figures 14 and 15 show performance results of SP on the IA64-Myrinet and IA32-IBA clusters. Like the other applications, the performance of both the datatype version and the original version is comparable.

4.7 Reduction of Communication Code

We found that MPI datatypes are an easy-to-use way to transfer non-contiguous data. As discussed in Section 3, our modification is straightforward. We counted the lines of code related to communication in both the original NAS benchmarks and the modified NAS benchmarks using

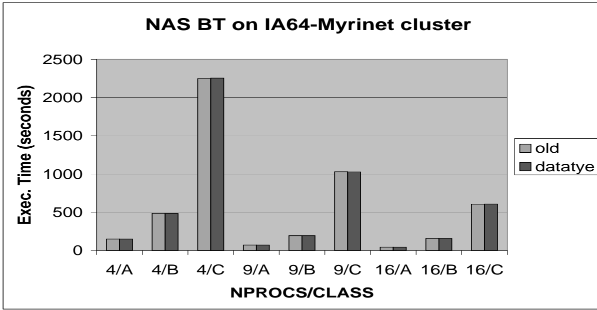

Figure 11. BT on IA64-Myrinet Cluster.

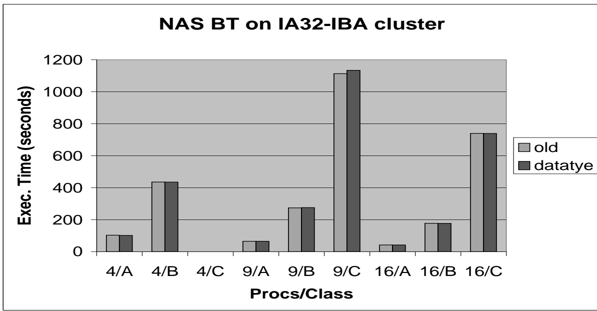

Figure 12. BT on IA32-IBA Cluster.

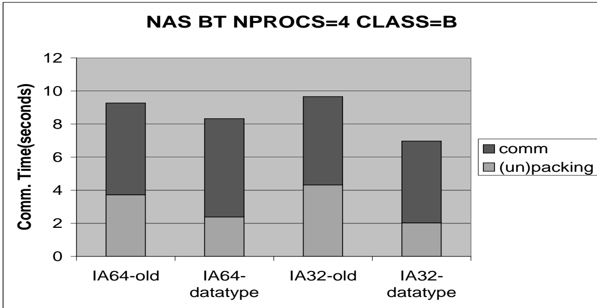

Figure 13. Cost to Transfer Non-Contiguous Data in BT.

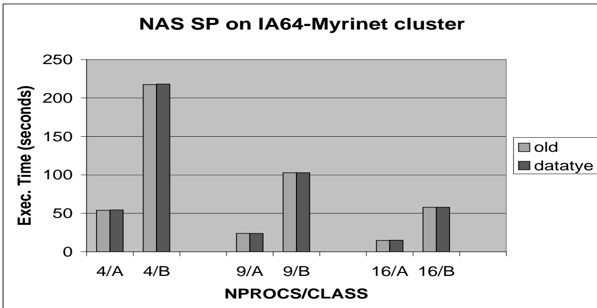

Figure 14. SP on IA64-Myrinet Cluster.

datatypes. Using datatypes can reduce the amount of communication code significantly. Details can be found in [14].

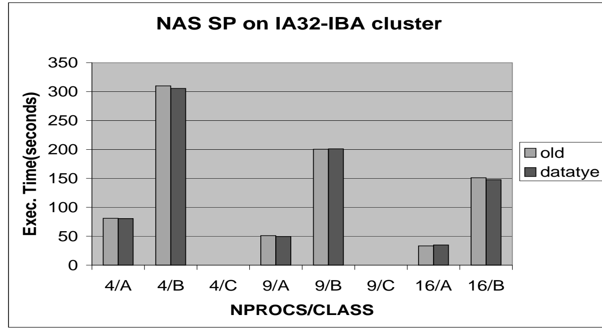

Figure 15. SP on IA32-IBA Cluster.

Our experiences show that MPI datatypes can become a key aid in application development.

5 Related Work

There has been a large amount of research work in improving the datatype processing and datatype communication in MPI implementations. This related work can be grouped into four areas. The first area is to improve the datatype processing system. Gropp *et al.* [11] have provided a taxonomy of MPI derived datatypes according to their memory access patterns and described how to efficiently implement these patterns. Träff *et al.* have described a technique, called flattening on the fly, for improving the datatype processing system [13]. Ross *et al.* [20] have designed a reusable datatype-processing component for the MPICH2 implementation [1].

The second area is to optimize packing and unpacking procedures. Byna *et al.* [6] have presented a technique which selects appropriate packing algorithms with respect to the architecture-specific parameters and the datatype memory access patterns. Recently, MPICH2 [1] has begun to deploy segment pack and unpack in its implementation. The Los Alamos Message Passing Interface (LA-MPI) system [9] has used shared memory regions as pack and unpack buffers in its datatype communication path.

The third area is to take advantage of network features to improve noncontiguous data communication. In [23], Worringen *et al.* have presented a direct copy technique to improve performance of datatype communication using the shared memory region provided by the SCI network. In [24], Wu *et al.* have demonstrated the benefits of using RDMA operations to support noncontiguous data communication. Four schemes are proposed to either overlap the packing, communication, and unpacking or eliminate the packing and/or unpacking.

The fourth area is to benchmark MPI datatype implementations. Work in [15, 19] focuses on using micro-benchmarks to evaluate the performance of datatype in different MPI implementations.

Our work differs from this previous work in the following three aspects. First, we focus on employing MPI datatypes in the NAS benchmarks. We analyze the devel-

opment complexity and effort using MPI datatypes in these application-level benchmarks. Second, we perform comprehensive performance evaluation of the current datatype implementation in MPICH on two typical clusters in the context of the NAS benchmarks. Third, the datatype version of NAS benchmarks complements the micro-benchmarks which have been used in the previous work.

6 Conclusions and Future Work

In this paper, we report on the implementation of four NAS benchmarks using MPI datatypes: MG, LU, BT, and SP. We perform comprehensive performance evaluation of the datatype version of these NAS benchmarks. Our experimental results on two clusters show that MPI datatypes can offer performance comparable to that of using manual packing/unpacking in these four NAS benchmarks. Our results also show that the manual packing and unpacking in the original NAS benchmarks are prone to performance degradation. Using datatypes can reduce or avoid these limitations.

All data type constructors provided in the MPI-1.1 standard are used due to the complicated non-contiguous data communication patterns in these benchmarks. We believe our modified NAS benchmarks can serve as application-level benchmarks to evaluate datatype processing and communication in various MPI implementations.

As for our future work, we plan to use these benchmarks to evaluate performance of the datatype communication schemes proposed in [24]. We are also working on the extension to the datatype processing system to take full advantage of the emerging RDMA communication mechanism.

References

[1] Argonne National Laboratory. MPICH2 Release 0.96p2. `http://www-unix.mcs.anl.gov/mpi/mpich2/`, Jan 2004.

[2] M. Ashworth. A Report on Further Progress in the Development of Codes for t he CS2. In *Deliverable D.4.1.b F. Carbonnell (Eds), GPMIMD2 ESPRIT Pr oject, EU DGIII, Brussels*, 1996.

[3] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165. ACM Press, 1991.

[5] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[6] S. Byna, X.-H. Sun, W. Gropp, and R. Thakur. Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[7] M. P. I. Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[8] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astro physical Thermonuclear Flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

[9] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. Minnich, C. E. Rasmussen, L. Dean Risinger, and M. W. Sukalski. A Network-Failure-tolerant Message-Passing system for Terascale Clusters. In *Proceedings of the 2002 International Conference on Supercomputing*, June 2002.

[10] W. Gropp, E. Lusk, and D. Swider. Toward faster packing and unpacking of mpi datatypes. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[11] W. Gropp, E. Lusk, and D. Swider. Improving the Performance of MPI Derived Datatypes. In *MPIDC*, 1999.

[12] InfiniBand Trade Association. Infiniband architecture specification release 1.1. `http://www.infinibandta.org/specs`, November 2002.

[13] J. L. Träff, R. Hempel, H. Ritzdorf and F. Zimmermann. Flattening on the Fly: Efficient Handling of MPI Derived Datatypes. In *PVM/MPI 1999*, pages 109–116, 1999.

[14] Q. Lu, J. Wu, D. K. Panda, and P. Saday. Employing MPI Derived Datatypes to the NAS Benchmarks: A Case Study. Technical Report OSU-CISRC-02/04-TR10, Dept. of Computer and Information Science, The Ohio State University, Feb. 2004.

[15] G. R. Luecke, S. Spanoyannis, and J. Coyle. The performance of mpi derived types on a sgi origin 2000, a cray t3e-900, a myrinet linux cluster and an ethernet linux cluster.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[17] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[18] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. `http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html`.

[19] R. Reussner, J. L. Träff, and G. Hunzelmann. A Benchmark for MPI Derived Datatypes. *Lecture Notes in Computer Science*, 1908:10+, 2000.

[20] R. Ross, N. Miller, and W. Gropp. Implementing Fast and Reusable Datatype Processing. In *EuroPVM/MPI*, Oct. 2003.

[21] R. Ross, D. Nurmi, A. Cheng, and M. Zingale. A Case Study in Application I/O on Linux Clusters. In *SC2001*, Nov. 2001.

[22] T. Tabe and Q. F. Stout. The use of the MPI communication library in the NAS parallel benchmarks. Technical Report CSE-TR-386-99, 17, 1999.

[23] J. Worringen, A. Gaer, F. Reker, and T. Bemmerl. Exploiting Transparent Remote Memory Access for Non-Contiguous- and One-Sided-Communication. In *Workshop on Communication Architecture for Clusters 2002 (in conjunction with IPDPS)*, April 2002.

[24] J. Wu, P. Wyckoff, and D. K. Panda. High Performance Implementation of MPI Datatype Communication over InfiniBand. In *International Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.