Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes

Torsten Hoefler and Steven Gottlieb*

National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign, Urbana, IL, USA,
{htor,sgottlie}@illinois.edu

Abstract. Many parallel applications need to communicate non-contiguous data. Most applications manually copy (pack/unpack) data before communications even though MPI allows a *zero-copy* specification. In this work, we study two complex use-cases: (1) Fast Fourier Transformation where we express a local memory transpose as part of the datatype, and (2) a conjugate gradient solver with a checkerboard layout that requires multiple nested datatypes. We demonstrate significant speedups up to a factor of 3.8 and 18%, respectively, in both cases. Our work can be used as a template to utilize datatypes for application developers. For MPI implementers, we show two practically relevant access patterns that deserve special optimization.

1 Introduction

The Message Passing Interface (MPI) offers a mechanism called derived datatypes (DDT) to specify arbitrary memory layouts for sending and receiving messages. This mighty mechanism allows the integration of communication into the parallel algorithm and data layout and thus is likely to become an important part of application development and optimization. Not only do DDTs save implementation effort by providing an abstract and versatile interface to specify arbitrary data layouts, but they also provide a portable high-performance abstraction for data accesses. It is easy to show that datatypes are complete in that any permutation from a layout on the sender to a layout on the receiver can be expressed (different DDTs at sender and receiver are allowed as long as the *type maps* [1] match).

Zero-copy refers to a mechanism to improve application performance by avoiding copies in the messaging middleware. Several low-level communication APIs, such as InfiniBand [2] or DCMF [3] allow direct copies from a user-buffer on the sender to a user-buffer at the receiver. We extend this definition into the application space and argue that the specification of derived datatypes is **necessary** to enable *zero-copy algorithms*, i.e., no explicit buffer pack/unpack, for parallel applications. It has been shown that non-contiguous data can be transferred without additional copies using InfiniBand [4].

* On leave from Indiana University, Bloomington, IN, USA

Many applications require sending data from non-contiguous locations, so we would expect that many MPI applications use datatypes to specify their communications. However, on the contrary, implementations of the DDT mechanism in MPI have been suboptimal so that manual packing and unpacking of data often yielded higher performance. In the last years, implementations have much improved [4–8] but the *folklore* about low performance still remains. Indeed, the number of success stories is low and limited to application benchmarks with relatively simple datatype layouts [9].

In this work, we demonstrate two complex use-cases for DDTs in parallel applications. The first example shows how to express the local transpose operations in a parallel Fast Fourier Transformation (FFT). The second example shows a complex 4-d stencil code with checkerboard layout.

2 Fast Fourier Transformations

Fast Fourier Transforms (FFT) have numerous applications in science and engineering and are among the most important algorithms today. One-dimensional (1-d) FFTs accept an array of N complex numbers as input and produce an array of size N as output. FFTs can also be done in place with negligible additional buffering. Such 1-d FFTs can be expressed as several multi-dimensional FFTs and application of so called *twiddle factors* [10, §12]. Such a decomposition is often used to parallelize FFTs because applying the twiddle factors is a purely local operation. Naturally multi-dimensional FFTs are also very important in practice, for example, 2-d FFTs for image analysis and manipulation and 3-d FFTs for real-space domains. Such n-d FFTs can be computed by performing 1-d FFTs in all n dimensions.

2.1 A typical parallel FFT implementation

We discuss a typical parallel implementation of a $N_x \times N_y$ 2-d FFT with MPI. We assume that the array is stored in x-major order and distributed along the x dimension such that each process has N_x/P y-pencils. Figure 1 illustrates the whole procedure for a 4×4 FFT on two processes (0 and 1). Each process holds two 4-element y-

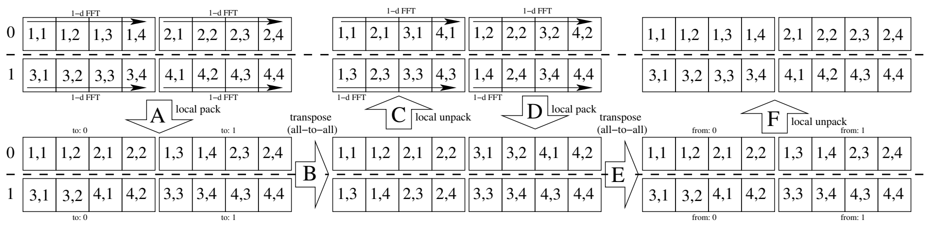


Fig. 1. Parallel two-dimensional FFT on two processes. The steps are explained below.

pencils in its local memory. The elements are shown with (x, y) indices in contiguous memory locations (left→right) in the figure. The two processes are drawn vertically and separated by a dashed line. The steps needed to transform the array and return it in the original layout are:

1. perform N_x/P 1-d FFTs in y-dimension (N_y elements each)
2. pack the array into a sendbuffer for the all-to-all (A)
3. perform global all-to-all (B)
4. unpack the array to be contiguous in x-dimension (each process has now N_y/P x-pencils) (C)
5. perform N_y/P 1-d FFTs in x-dimension (N_x elements each)
6. pack the array into a sendbuffer for the all-to-all (D)
7. perform global all-to-all (E)
8. unpack the array to its original layout (F)

Thus, in order to transform the two-dimensional data, it is rearranged six times. Each rearrangement is effectively a copy operation of the whole data. However, four rearrangements (pack and unpack) are related to the global transpose operation. Since MPI datatypes are complete, we can fold all pack and unpack operations into the communication and thus avoid the explicit copy for packing the data.

2.2 Constructing the Datatypes

We assume that the basic element is a complex number. A datatype for complex numbers can simply be created with MPI_Type_contiguous with two double elements.

The send-datatype can be constructed with MPI_Type_vector because each y-pencil is logically cut into P pieces that need to be redistributed to P processes. Thus, the blocklength is $\frac{N_y}{P}$. Each process typically holds $\frac{N_x}{P}$ pencils, thus, there are a total of $\frac{N_x}{P}$ such blocks. The stride between the blocks is one complete y-pencil of length N_y. The basic vector datatype is shown in Figure 2(a). Sending a single element of this datatype would transmit {(1,1),(1,2),(2,1),(2,2)}. The problem is now that the *comb*-shaped

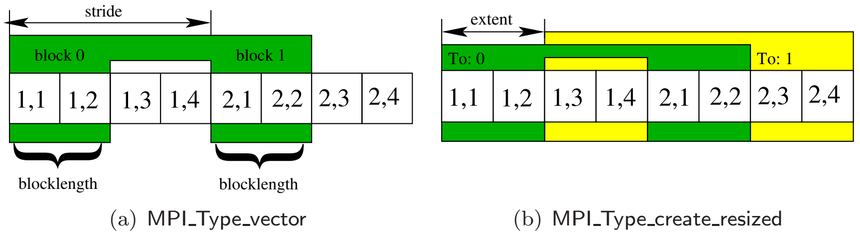
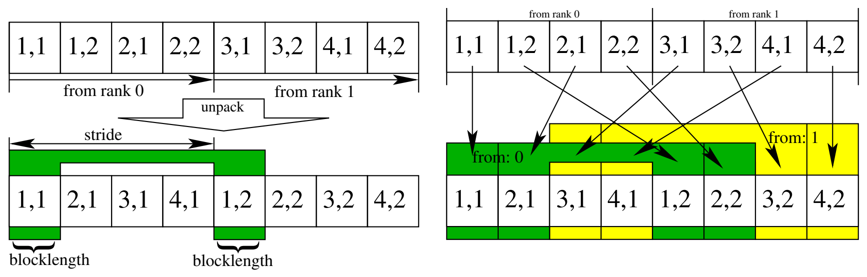


Fig. 2. Visualization of the send datatype creation.

datatypes that need to be sent are interleaved as shown in Figure 2(b). Thus, one can't just sent two of those datatypes (as this would gather two contiguous combs instead of two interleaved combs). MPI allows the user to change the *extent* of a datatype in order to allow such interleaved accesses. In our example we use MPI_Type_create_resized to change the extent to $\frac{N_y}{P}$ times the base-size as shown in Figure 2(b). The resulting datatype can be used as input to MPI_Alltoall by sending `count=1` to each process.

Performing the unpack on the receiver is slightly more complex because the data arrives in non-transposed form from the sender. Thus, the receiver does not only need to unpack the data but also transpose each block locally. This can also be expressed in a single derived datatype. The top of Figure 3(a) shows how the data-stream arrives at the receiver (process 0) and the bottom the desired layout after unpack. Like in the sender-case, we create a MPI_Type_vector datatype. However, the blocklength is now one element because we need to transpose the array locally. We have $\frac{N_x}{P}$ blocks with a stride of N_x between them. The newly created comb-shaped type captures one incoming



(a) MPI_Type_vector (b) MPI_Type_create_resized,_contiguous

Fig. 3. Visualization of the receive datatype creation.

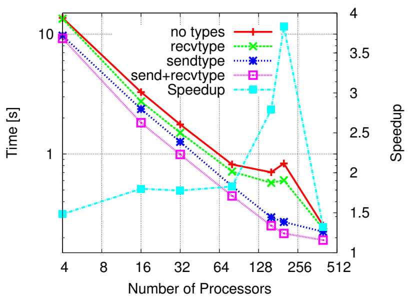
y-contribution of one process. To capture all, we need to create a contiguous datatype with $\frac{N_y}{P}$ elements. We have to change the extent to 1 with MPI_Type_create_resized as for the send datatype. Figure 3(b) shows the final datatypes for our example. Those types can be used as the receive type in MPI_Alltoall with `count=1` per process (note that the send- and receive-types in MPI_Alltoall do not have to be identical as long as the type-map matches).

By using both created datatypes, we can effectively eliminate steps A, C, D, and F in Figure 1 which leads to a zero-copy FFT. An optimized MPI implementation would stream the data items directly from the send buffers into the receive buffers and apply the correct permutation (local transpose). This should lead to significant performance improvements over the state of the art because it avoids four explicit copies of the whole 2-d array. Higher-dimensional FFTs can be treated with similar principles.

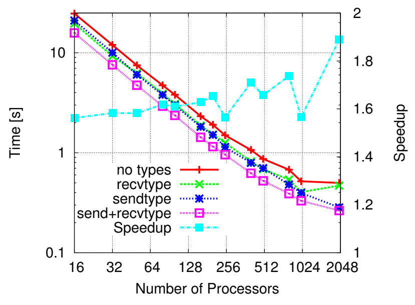
2.3 Experimental Evaluation

We used two systems for our performance evaluation, *Odin* at Indiana University and *Jaguar* at the Oak Ridge National Laboratory. Odin consists of 128 compute nodes with dual-CPU dual-core Opteron 1354 2.1 GHz CPUs running Linux 2.6.18 and are connected with SDR InfiniBand (OFED 1.3.1). We used Open MPI 1.4.1 (openib BTL) and g++ 4.1.2 for our evaluation. Jaguar (XT-4) comprises 150152 2.1 GHz Opteron cores in quad-core nodes connected with a Torus network (SeaStar). Jaguar runs Compute Node Linux 2.1 and the Cray Message Passing Toolkit 3. All software was compiled with `-O3 -mtune=opteron` on both systems.

In all experiments, we ran one warmup round (using the same buffers as for the actual run). We repeated each run three times (in the same allocation) and found a maximum deviation of 4%. We report the smallest measured time for the complete



(a) Odin (N=8000).



(b) Jaguar (N=20000).

Fig. 4. Strong-scaling of a 2-d FFT with and without zero-copy (MPI Datatypes).

parallel 2-d FFT of the three runs. The overhead to create the derived datatypes is included in the measurements that use derived datatypes.

Figure 4 shows the results for a strong-scaling $N \times N$ 2-d FFT on Odin and Jaguar. Using derived datatypes improves the performance of parallel FFTs on both systems by more then a factor of 1.5. The improvement typically grows with the number of processes as local FFTs get smaller. The anomaly at P=200 on Odin (Figure 4(a)) is reproducible. Datatypes also improved parallel scaling on Jaguar as shown in Figure 4(b) where the traditional FFT stopped scaling at 1024 processes and the version using derived datatypes scaled up to 2048 processes.

The performance of derived datatypes is system dependent and might as well not result in any speedup if the implementation performs a complete local pack/unpack. We found only one system, BlueGene/P, where the datatype implementation is slowing down the FFT significantly (up to 40%). The simple representation of the constructed vector datatype should not introduce significant overhead. This might point at an optimization opportunity or performance problem of the MPI implementation on BlueGene/P.

3 MIMD Lattice Computation Collaboration Application

The MIMD Lattice Computation (MILC) Collaboration studies Quantum Chromodynamics (QCD) the theory of the strong interaction [11]. Their suite of applications, known as the MILC code is publicly available for the study of lattice QCD. This group regularly gets one of the largest allocations of computer time at NSF supercomputer centers. One application from the code suite, su3_rmd, is part of the SPEC CPU2006 and SPEC MPI benchmarks. It is also used to evaluate the performance of the Blue Waters computer to be built by IBM.

Lattice QCD approximates space-time as a finite regular hypercubic grid of points in four dimensions. The physical quarks are represented by 3-component complex objects at each point of the grid. The variables that describe the gluons, the carriers of the strong force are represented by 3×3 unitary matrices residing on each 'link' joining points of the grid. Currently, grids as large as $64^3 \times 192$ are in use. Much of the floating point work is involved in multiplying the 3×3 matrices together or applying the matrix to a 3-component vector. Routines for these basic operations are often optimized by assembly code or compiler intrinsics.

The code is easily parallelized by domain decomposition. Once that is done, the program must be able to communicate with neighboring processes that contain off-node neighbors of the points in its local domain. The MILC code abstracts all the communication into a small set of routines: `start_gather`, `wait_gather`, and `cleanup_gather`. These routines are all contained in a single file specific to the message passing library available on the target computer.

The MILC code allows very general assignments of grid points to the processes. At startup, a list of local grid points that need off-node neighbors for their computation is created for each direction $\pm x$, $\pm y$, $\pm z$, $\pm t$. There is one list corresponding to each other process that contains any needed neighbors for a particular direction. There are also similar lists for all the local grid points whose values will need to be sent to other processes. At the time a gather is called, the lists containing data that must be sent to other processes are processed and for each grid point in a list the value of the data to be gathered is copied (packed) into a buffer. The buffer is then sent to the neighboring process. The index list is used to allow for arbitrary decompositions of the grid; however, in practice, the most common data layout is just to break up the domain into hyperrectangular subdomains with checker boarding as described below. It is for this case that we have implemented derived datatypes to avoid copying the data to a buffer before sending it to the destination process. The receive portion does not require datatypes because the computation uses indirect addressing for all grid points. The index list of local grid points with remote data dependencies is set (once during initialization) to point to the correct element in the receive buffer.

3.1 Data Layout and Datatype Construction

The code consists of several computation phases that perform different tasks. There are compilation flags that allow timing and printing performance information for each phase. In this work, we will concentrate on the conjugate gradient (CG) solver since that routine takes the vast majority of the time in production runs. Checkerboarding, or even-odd decomposition is used in the iterative solver. A grid point is even (odd) if the sum of its coordinates is even (odd). Thus, the grid points are stored in memory so that all even sites are stored before the odd sites. If the coordinates of a point are denoted (x, y, z, t), the data is stored so that x is incremented first, then y is incremented, then z and finally t. That means that the edge of the domain in t is (almost) contiguously stored. If the local domain is of size $L_x \times L_y \times L_z \times L_t$, there are $L_x \times L_y \times L_z/2$ even sites stored contiguously and the same number of odd sites stored contiguously. Note that our current implementation of datatypes requires that each of the local dimensions is even. During the CG solver, we are usually only transferring one checkerboard at a time. (In other phases of the code, we operate on all grid points, so we also define datatypes for even-and-odd gathers. These are defined with MPI_Type_hvector in the code example. The blocks of even and odd sites are identical patterns separated by the number of even sites on each process. This is converted to bytes by multiplying by the size of the object.) If we need to fetch values from the z-direction, however, the points are not all stored contiguously. For each value of t, there are $L_x \times L_y/2$ contiguous sites in each checkerboard. The datatype defined for the gathers in the z-direction consists of L_t repetitions of such contiguous data. For the gathers in the y-direction, there are $L_z \times L_t$ regions of $L_x/2$ contiguous sites. Listing 1.1 shows parts of the datatype layout routine which is called during initialization.

```
/* the basic elements */
MPI_Type_contiguous(6, MPI_FLOAT, &su3_vect_dt);
MPI_Type_contiguous(12, MPI_FLOAT, &half_wilson_vector_mpi_t);
MPI_Type_contiguous(18, MPI_FLOAT, &su3_matrix_mpi_t);

/* 48 field types, 3 for su3_vector, half_wilson_vector, and su3_matrix,
   2 for even and even and odd, 8 for directions */
MPI_Datatype neigh_dt_ddt[3][2][8];

/* t-direction, even points */
MPI_Type_contiguous($L_x \cdot L_y \cdot L_z/2$, su3_vect_dt, &neigh_dt_ddt[0][0][3]);
/* t-direction, even and odd points */
MPI_type_hvector(2,1,sizeof(su3_vector)*even_sites, neigh_dt_ddt[0][0][3],
  &neigh_dt_ddt[0][1][3]);

/* z-direction, even points */
MPI_Type_vector($L_t$, $L_x \cdot L_y/2$, $L_x \cdot L_y \cdot L_z/2$, su3_vect_dt,
  &neigh_dt_ddt[0][0][2]);
/* z-direction, even and odd points */
MPI_type_hvector(2,1,sizeof(su3_vector)*even_sites, neigh_dt_ddt[0][0][2],
  &neigh_dt_ddt[0][1][2]);
...
```

Listing 1.1. Datatype Example for the Up Direction and `su3_vector`. MILC uses 48 different data layouts for sending.

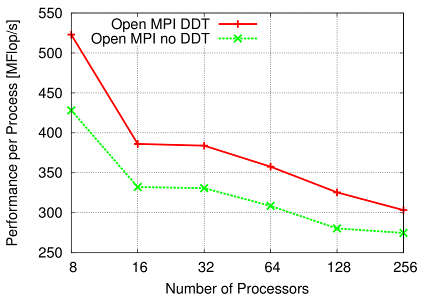
Three other issues are simplified in the code example. We do not show code for negative directions or for gathers of matrices and pairs of vectors. We show the basic definitions for `half_wilson_vector_mpi_t` and `su3_matrix_mpi_t`, but not the corresponding definitions of `field_neigh_dt`[{`1,2`}][ ][ ]. Further, for the CG routine, we also need to gather from sites three grid points away in each direction. These require contiguous blocks three times as long and merely require changing some factors of 1/2 to 3/2.

3.2 Experimental Evaluation

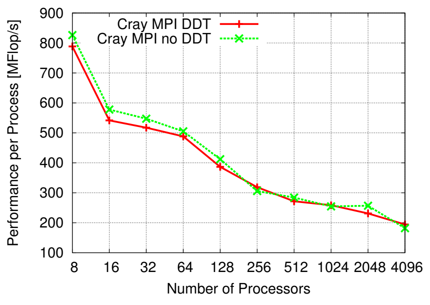
We now present performance results comparing the version the datatype version with the original pack/unpack version. We chose a weak scaling problem of size $L_x = L_y = L_z = L_t = 4$ per process which is similar to the Petascale benchmark problem that will be used to verify the Blue Waters machine on $> 3 \cdot 10^5$ cores. We ran each benchmark multiple times and report the average performance of all CG phases.

Figure 5 shows the performance in MFlop/s of runs on Odin and Jaguar. The CG solver requires global sums in addition to the nearest neighbor gathers. These sums are the biggest impediment to scaling since the global sum time is expected to increase as the logarithm of the number of processes. For a fixed local grid size, *i.e.*, weak scaling, the time for the global sum will eventually dominate the time for the work that must be done on each process. This is reflected in the decreasing performance is the number of processors is increased beyond 16. The sharp dropoff between 8 and 16 is due to the fact the one additional direction has off-node neighbors. Most other parts of the code do not require global sums. We see a speedup up to 18% by using derived datatypes on Odin while we see no benefit, indeed an average performance penalty of 3% on Jaguar.

The performance degradation on Jaguar is surprising because the data access of the MPI_Type_vector definition of the used datatype can be easily expressed as two



(a) Odin.



(b) Jaguar.

Fig. 5. Weak-scaling MILC run with a 4^4 lattice per process.

loops [5, 7] while the original MILC packing routine traverses an array of indices which adds more pressure to the memory subsystem. This points at possible optimization opportunities in Cray's MPI because the simple structure of the datatype should, even in a simple implementation, not introduce significant overheads.

4 Conclusions

We demonstrated two applications that can take significant advantage of using MPI's derived datatype mechanism for communication. Such techniques essentially enable parallel zero-copy algorithms and even allows one to express additional local transformations (as demonstrated for FFT). Performance results of FFT and a CG solver show improvements up to a factor of 3.8 and 18% respectively. However, we also found performance degradation, which indicate optimization opportunities in the MPI libraries on BlueGene/P and Jaguar systems, in some cases.

We expect that our results will influence two groups: (1) application developers are encouraged to use MPI datatypes to simplify and optimize their code, and (2) MPI implementers should use the presented algorithms as examples for practically relevant access patterns that might benefit from extra optimizations. The source code of both applications is publicly available and can be used for evaluating datatype implementations.

Acknowledgments The authors want to thank Bill Gropp (UIUC), Jeongnim Kim (UIUC), Greg Bauer (UIUC), and the anonymous reviewers for helpful comments. This work is supported by the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

References

1. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4th 2009) http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

2. The InfiniBand Trade Association: Infiniband Architecture Specification Volume 1, Release 1.2. InfiniBand Trade Association. (2003)
3. Kumar, S., et al.: The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer. In: ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, New York, NY, USA, ACM (2008) 94–103
4. Santhanaraman, G., Wu, J., Huang, W., Panda, D.K.: Designing zero-copy message passing interface derived datatype communication over infiniband: Alternative approaches and performance evaluation. Int. J. High Perform. Comput. Appl. **19**(2) (2005) 129–142
5. Träff, J.L., Hempel, R., Ritzdorf, H., Zimmermann, F.: Flattening on the fly: Efficient handling of mpi derived datatypes. In: Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, London, UK, Springer-Verlag (1999) 109–116
6. Gabriel, E., Resch, M., Rühle, R.: Implementing and benchmarking derived datatypes in metacomputing. In: HPCN Europe 2001: Proc. of the 9th Intl. Conference on High-Performance Computing and Networking, London, UK, Springer-Verlag (2001) 493–502
7. Gropp, W., Lusk, E., Swider, D.: Improving the performance of mpi derived datatypes. In: in Proceedings of the Third MPI Developer's and User's Conference, MPI Software Technology Press, MPI Software Technology Press (1999) 25–30
8. Byna, S., Gropp, W., Sun, X.H., Thakur, R.: Improving the performance of mpi derived datatypes by optimizing memory-access cost. Cluster Computing, IEEE International Conference on **0** (2003) 412
9. Lu, Q., Wu, J., Panda, D., Sadayappan, P.: Applying MPI Derived Datatypes to the NAS Benchmarks: A Case Study. In: Proc. of the Intl. Conf. on Par. Proc. Workshops. (2004)
10. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical recipes in C (2nd ed.): the art of scientific computing. Cambridge University Press (1992)
11. Bernard, C., Ogilvie, M.C., DeGrand, T.A., DeTar, C.E., Gottlieb, S.A., Krasnitz, A., Sugar, R., Toussaint, D.: Studying Quarks and Gluons On Mimd Parallel Computers. International Journal of High Performance Computing Applications **5**(4) (1991) 61–70