MPI Datatype Processing using Runtime Compilation

Timo Schneider
ETH Zurich
Dept. of Computer Science
Universitätstr. 6
8092 Zurich, Switzerland
timos@inf.ethz.ch

Fredrik Kjolstad
MIT
CSAIL
32 Vassar Street Cambridge,
MA 02139
fred@csail.mit.edu

Torsten Hoefler
ETH Zurich
Dept. of Computer Science
Universitätstr. 6
8092 Zurich, Switzerland
htor@inf.ethz.ch

ABSTRACT

Data packing before and after communication can make up as much as 90% of the communication time on modern computers. Despite MPI's well-defined datatype interface for non-contiguous data access, many codes use manual pack loops for performance reasons. Programmers write access-pattern specific pack loops (e.g., do manual unrolling) for which compilers emit optimized code. In contrast, MPI implementations in use today *interpret* datatypes at pack time, resulting in high overheads. In this work we explore the effectiveness of using runtime compilation techniques to generate efficient and optimized pack code for MPI datatypes at commit time. Thus, none of the overhead of datatype interpretation is incurred at pack time and pack setup is as fast as calling a function pointer. We have implemented a library called libpack that can be used to compile and (un)pack MPI datatypes. The library optimizes the datatype representation and uses the LLVM framework to produce vectorized machine code for each datatype at commit time. We show several examples of how MPI datatype pack functions benefit from runtime compilation and analyze the performance of compiled pack functions for the data access patterns in many applications. We show that the pack/unpack functions generated by our packing library are *seven* times faster than those of prevalent MPI implementations for 73% of the datatypes used in a scientific application and in many cases outperform manual pack loops.

1. INTRODUCTION

Most scientific applications perform some form of domain decomposition to parallelize the effort of solving a specific problem. Each process in an SPMD program is responsible for a patch of the global domain and performs calculations on this local sub-problem independently of other processes. If the underlying problem requires iterations but is not massively parallel, partial solutions of the local domain have to be exchanged with neighboring processes between iterations. A simple example for such a problem are time-stepping stencil codes [13], as shown in Figure 1. In this example the global domain is represented by an $N \times N$ matrix, which gets partitioned into p blocks (one per process) of roughly equal size. After each time step the boundary regions of these partitions have to be exchanged with neighboring processes, before the next time-step can be started.

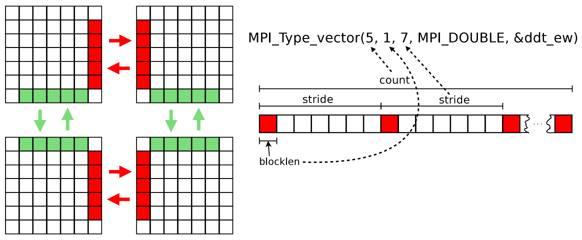


Figure 1: Domain-decomposed 2D stencil. Data exchanged in east-west direction must be packed before it can be sent, e.g. with the given MPI datatype.

If we assume matrices are stored in row-major order, data exchanged in the north-south direction is consecutive in local memory, while data in the east-west direction is not. It either has to be sent and received in multiple small chunks, which can be inefficient due to the constant overhead associated with each send operation, or the data has to be *packed* into a consecutive buffer and sent in one piece. Of course this process has to be reversed (the data has to be *unpacked*) after such data is received. However, in the following we will not separate between packing and unpacking as they are symmetric — when we say packing we mean both packing and unpacking. The packing approach can be found in many codes which are similar to the example above, for example WRF [15], MILC [2], NAS LU, MG, SP and BT [5], and SPECFEM [4]. Most of these applications pack data by looping over it and copying it into a temporary buffer. We call this *manual packing* or *pack loops*.

Manual pack loops have drawbacks. First, and most importantly, they always perform an explicit copy of all the data. However, this is just one of multiple possible strategies. Other possible strategies include pipelining the packing with partial sends, thereby overlapping packing and sending, or better yet, to use emerging network hardware with support for non-contiguous transfers [16]. Replacing manual packing code with data layout descriptions (datatypes) gives MPI implementations flexibility to choose the best strategy for the available hardware. Furthermore, it can detect cases where the data to be sent is contiguous and avoid the copy

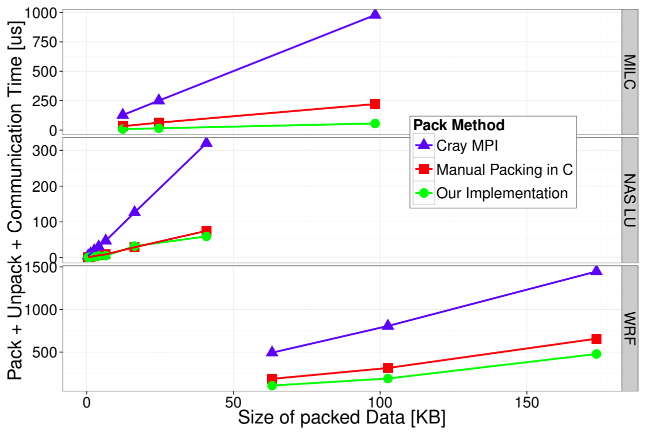


Figure 2: Overhead of exchanging non-continuous data in three codes. The data can be packed explicitly with a manual pack loop or our runtime-compiled pack functions, or it can be packed implicitly using MPI derived datatypes in any MPI communication.

altogether! Finally, the optimal packing strategy depends on the target machine and compiler, which makes it hard for application programmers to write performance-portable pack loops.

For these reasons the MPI standard defines an interface to specify non-contiguous data layouts, called MPI Derived Datatypes (DDTs), which can be used with any communication or IO function [12]. For example, the DDT for the east-west exchange in Figure 1 can be constructed with an MPI vector datatype. Before a DDT can be used to send or receive data, it has to be *committed*. Committing a datatype allows MPI to perform optimizations on that datatype that allow efficient packing and unpacking of data.

We showed in a previous paper that packing and unpacking data can contribute up to 90% of the total communication overhead for non-contiguous sends [14]. We also observed that in many cases the manual pack loops are faster than using MPI derived datatypes. One of the reasons for this is that manual pack loops are written as specific as possible and can be translated by the compiler into very efficient, machine code, while MPI DDTs are *interpreted* at runtime in all current implementations. In this work we attempt to bridge this gap by using runtime compilation techniques to generate efficient native packing code at the moment a datatype is committed. We utilize the fact that, at the time we generate the pack functions, we know the values of all arguments with which the datatype was constructed (including its sub-types). This lets us generate highly specialized code, unroll small loops completely, promote datatypes to more specialized types and merge types. Figure 2 demonstrates that this approach is very effective. Runtime compiled pack functions outperform MPI packing in all tested cases (which are taken from real-world applications such as WRF [15], MILC [2], or the NAS benchmarks [5].

In this work we demonstrate the effectiveness of using runtime compilation to generate fast pack functions for MPI DDTs. However, the same approach can also be used to enable fast packing for bulk transfers in other programming models, such as global address space languages. We show how runtime compilation can be done in a straightforward platform-independent manner. Since we use LLVM for code generation we can generate pack/unpack functions for many architectures, including x86, PowerPC, ARM, and IA64. The goal of this work is to answer whether datatype engines benefit from runtime compilation techniques. Finally, we have made our packing library for MPI DDTs publicly available for use by the community.[1]

2. RUNTIME DATATYPE COMPILATION

In the introduction we mentioned the MPI vector datatype. This datatype is part of a family of three derived datatypes that describe regular packing patterns. These are contiguous, vector, hvector datatypes. Contiguous types take a count specifying the number of sub-types. Note that subtypes can either be primitives, such as doubles and integers, or they can themselves be arbitrary derived datatypes. Vectors take a three-tuple (count, blocklength, stride) and pack *count* blocks of *blocklength* contiguous subtypes where each block is the extent of *stride* subtypes apart. Hvectors are like vectors, with the exception that strides are given in bytes instead of subtype extents.

The other family of derived datatypes are the indexed datatypes that describe irregular pack patters. These are indexed_block, indexed, hindexed and struct. Common to all of these is that the location of the subtype blocks are given as an displacement/index list. Apart from this, indexed_block and indexed datatypes specify displacements in element multiples, while hindexed and struct datatypes specify them in bytes. Furthermore, indexed_block datatypes require each block to have the same length, and struct types allow a different subtypes for each block.

We use LLVM for code generation [11]. LLVM is a modular compilation framework centered around a well documented type-safe intermediate representation (IR). At commit time we use the LLVM code generation APIs to construct an in-memory representation of our packing code in LLVM IR. We then invoke the LLVM JIT code generation backend to produce machine code for the target backend, which returns to us a function pointer to the pack function. Note that, although LLVM supports a rich set of optimizations on its IR, we do not need them or use them. The reason for this is that datatypes are very simple compared to a general purpose languages, and we can therefore output compact and optimized LLVM IR in the first place. Running LLVM optimization passes on this IR increases compilation time and does not improve the code.

2.1 Packing Contiguous Data

Even when packing non-contiguous datatypes, the leaves of the datatype hierarchies are blocks of contiguous data of varying lengths. This is the code that performs the actual copy work and it is critical that it performs well. There is a surprising variety of ways to copy data on modern processors, some of which generalize well across platforms. On X86 there is a choice between the movs instruction that moves data between two memory locations, and instructions that load data into registers followed by instructions that store the data to a different memory location. Load and store instructions come in many forms, from normal one-word loads

[1]`http://spcl.inf.ethz.ch/Research/Parallel_Programming/MPI_Datatypes/libpack/`

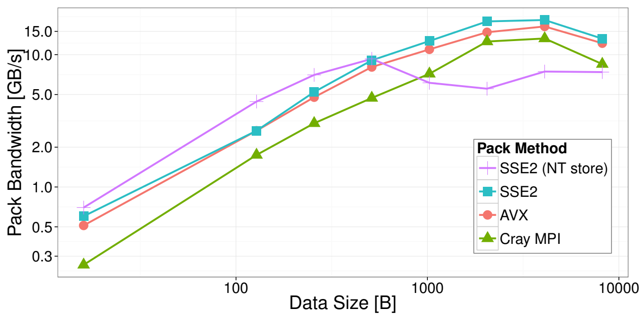


Figure 3: A comparison of different methods to copy contiguous blocks of data on the JYC machine.

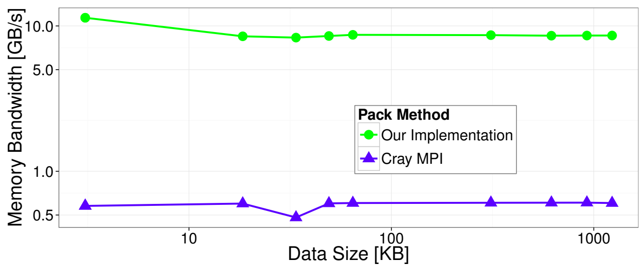


Figure 4: A comparison of Cray MPI with runtime compiled pack functions for packing a nested vector with a small inner vector.

to vector instructions of varying widths. Furthermore, the vector instructions can be aligned or unaligned and temporal or non-temporal. The latter refers to whether the load/stores will bypass the cache or not.

As demonstrated in Figure 3 we compared several different instruction types and found that there is no single optimal copy method per machine, and certainly not across different architectures. Furthermore, we found that for typical small data copies, the overhead of using aligned instructions, which in general require a preamble and postamble to copy the unaligned part of the output buffer with unaligned instructions, is higher than the gain.

Our implementation supports both aligned and unaligned stores and the method can be selected at compile time, but defaults to unaligned stores. Furthermore, we use the LLVM vector type to generate vector instructions automatically. Since, 128-byte stores (SSE2) were superior on our test platforms, we currently use this vector width. In addition, the loops around the vectors are unrolled 16 times and if the size of the contiguous data is less than 256, the loop is completely unrolled and removed. The vector width, unroll factor and unroll threshold are optimized for current X86 processors, but are compile-time configurable. Although LLVM's vector types allow our code to generalize well, future work could include determining these values through experimentation.

2.2 Compiling Vector Datatypes

Multi-dimensional arrays are very common in scientific applications and border exchanges typically require slices of these to be communicated, such as 2D or 3D faces. Such slices can most conveniently be described with vector hierarchies or the subarray datatype and the performance of such datatypes is therefore critical for a solid datatype implementation.

Our runtime compiler generates specialized code for vector datatypes that takes advantage of the fact that the vector count as well as the subtype's extent are known at compile time to reduce the number of induction variables and to precompute the loop bound.

For example, the MILC code creates the following datatype when run with 32 processes: a hvector datatype (count=2, blocklen=1, stride=6144) of vector datatypes (count=8 blocklength=8 stride=32) of contiguous datatypes (count=6), of MPI_FLOATs.

In this example the very low vector loop overhead of the code produced by our runtime compiler, combined with the unrolled code generated for the blocks of the innermost vector, results in very efficient code. In Figure 4 we gradually increase the count of the outermost vector. We see that, even when the datatype becomes larger than the L1 cache size, our runtime compiled pack function always outperforms the pack function of Crays vendor MPI (from Cray Compiling Environment 8.1.6) by a factor of 10.

We believe most of this performance advantage comes from the fact that our runtime compiler determines an efficient inner vector loop at compile time and incurs no overhead at pack time to decide what to do. An interpreted approach would have to incur overhead at pack time to select a good inner loop, and nested vectors would necessitate this overhead multiple times. We see the same effect with MPICH2 where we can inspect the source code, and it highlights the advantage an runtime compilation approach has over an interpreter.

2.3 Compiling Irregular Datatypes

As described above, irregular datatypes such as hindexed, indexed_block and structs contain a list of offsets as well as a description of the lengths of the blocks pack at each offset.

Such datatypes are often used to capture one of two patterns. The first case is when the communicated data lacks structure, such as for graphs, particle codes, or irregular meshes. The SPECFEM application is an example of the latter. The second case is when the user needs to pack data from multiple data structure, such as two arrays. In this case, an hindexed or struct datatype can be used to link the datatypes describing each array together. This is done in the WRF application, where data that logically belongs together is kept in three separate arrays and is packed using a struct of vectors.

Such data-access patterns are hard to pack efficiently with traditional, interpreter-like datatype engines, since they must load the complete index list into registers during packing. Using runtime compilation we can embed the index list into the generated code if the index list is small, as is often the case. Otherwise, a runtime compiler can unroll the loop over the indices to reduce looping overhead.

Figure 5 compares our performance to that of Cray MPI for index lists. Again, we observe that runtime compilation results in significantly lower overheads and improved performance compared to an interpreted approach.

2.4 Compiling Hierarchical Datatypes

As we alluded to earlier, MPI datatypes can be composed in arbitrary hierarchies to describe any layout. For example, while vector datatypes can be used to pack slices of 2D ar-

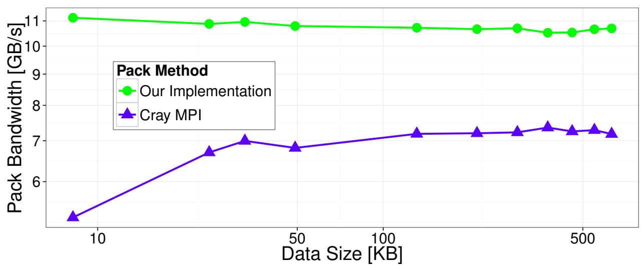
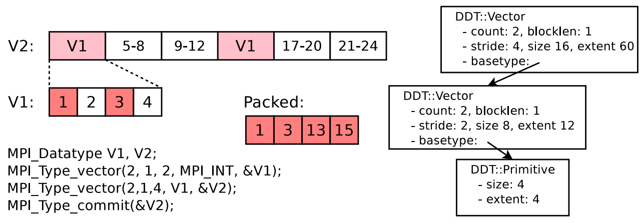


Figure 5: A performance comparison between Cray MPI and runtime compiled pack functions packing hindex datatypes with random displacements.



```
MPI_Datatype V1, V2;
MPI_Type_vector(2, 1, 2, MPI_INT, &V1);
MPI_Type_vector(2,1,4, V1, &V2);
MPI_Type_commit(&V2);
```

Figure 6: An example for a nested vector DDT, its memory-access pattern and internal representation in our object-oriented packing library.

rays, vector-of-vectors allow sub-volumes of 3D arrays to be packed. Figure 6 demonstrates how derived datatypes can be combined to create more complex data-access patterns.

Each datatype is represented as a C++ object, and each object has one or more pointers to its subtype(s). To generate the pack functions, each datatype object has a *codegen* method that emits LLVM IR to pack itself, which includes calling it's subtype's codegen method. A more detailed description of the implementation can be found in the README file which comes with the code of libpack.

However, before code generation a few optimizations are performed bottom up on the datatype tree representation. First, if a contiguous, vector or hvector datatype has a contiguous subtype, then the subtype is merged into the count/blocklength of its parent and removed. This optimization only applies if the size of the contiguous type equals its extent. Second, vector and hvector datatypes, whose stride (in elements of the subtype) equals their blocklength are promoted to contiguous types. These optimizations allow us to produce better code with fewer loops, without complicating the code generators.

When packing large datatypes for a blocking send, it is possible to overlap the packing process with communication by *segmenting* the datatype: it is unnecessary to wait until the whole datatype has been packed, as soon as a chunk of the packed buffer is ready, this chunk can be sent (in a non-blocking manner) and a second buffer can be used to pack the next segment. The same technique can be used to unpack data partially before the full buffer is received. This method keeps the size of the required temporary pack buffer constant, regardless of the size of the datatype. This can be done with our library by splitting a high level datatype into multiple datatypes thereby obtaining different pack functions for the different segments.

2.5 Performance Hints to MPI

We identified several sources of uncertainty which hinder further optimization of derived datatypes in MPI.

First, the user has no way to indicate at commit time if he wishes to reuse the datatype many times, and can therefore tolerate longer optimization times, or if the datatype will not be reused and therefore it is important to minimize the create and commit overhead. Second, when the pack function is used, the output and input buffers are supplied. In many applications the buffers are always the same. If that is the case then the pack function can use absolute instead of (often slower) relative addressing. Furthermore, if it is known which parts of the buffer will be aligned, then aligned memory operations can be generated without the overhead of the mentioned pre- and postamble. Third, in many cases the user always communicates the same number of datatypes. If this is the case, additional opportunities for unrolling the outer loop exists. For example, if the number of datatypes sent is always 1, then the outer loop can be completely removed.

The user could remove these uncertainty factors by using MPI info arguments, if the MPI_Type_commit function would accept them in future versions of MPI.

3. EXPERIMENTS

All experiments in this paper have been carried out on JYC, the Blue Waters test system at the National Center for Supercomputing Applications. JYC consists of a single cabinet Cray XE6 (74 nodes with 2368 Interlagos 2.3-2.6 GHz cores). We use the GNU compiler version 4.7.2 and compiled all benchmarks with -O3 optimization. The Cray Compilation Environment version on the system is 8.1.6, however, our packing library also beats recent versions of MPICH, MVAICH and Open MPI on Intel i5 and i7 CPUs, for which we do not provide results due to space limitations.

3.1 Micro-Applications

In this section, we compare the performance of runtime compiled pack functions to that of Cray's optimized MPI, using DDTBench [14], a micro-application based benchmark for MPI derived datatypes. It contains manual pack routines extracted from real application codes, such as LAMMPS, SPECFEM3D, MILC, and the NAS benchmarks. These pack loops have been converted to MPI datatypes (partly using a tool we developed in previous work [10]) to allow us to benchmark MPI datatype engines with realistic datatypes. We wrote an MPI wrapper for our library, which intercepts the calls to MPI datatype functions as well as those to send, receive and wait functions. If the datatypes used in those functions are not primitive types, it packs/unpacks them using our packing library and uses MPI's PMPI interface to send and received packed data. This wrapper allows us to use DDTBench without modifications to benchmark our runtime compiled pack functions.

DDTBench performs a ping-pong between two processes, and uses either a manual pack loop or the equivalent datatype to pack data each time it leaves a process and unpack it on arrival. By comparing the timings obtained to those of a ping-pong of equal size with consecutive data, we can measure how much packing the non-consecutive data costs, as a fraction of the total round-trip time. Figure 7 shows the result of such a measurement. Runtime compiled datatypes always outperform Cray MPI, and are roughly on

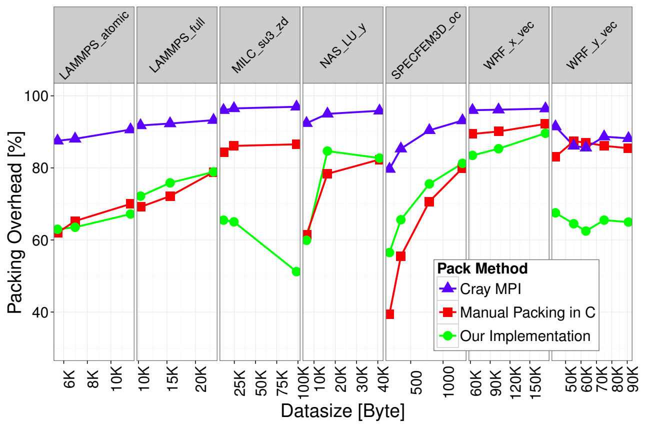


Figure 7: Packing overhead for data access patterns from different applications. Runtime compiled pack functions always outperform Cray MPI and are comparable with manual packing.

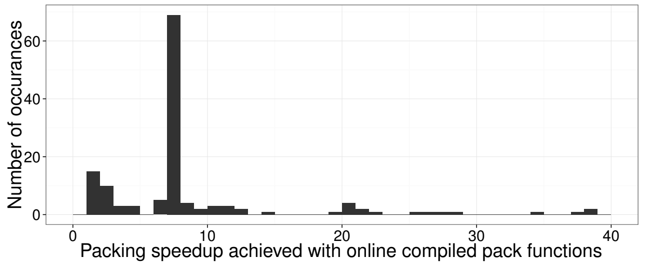


Figure 8: Speedup of DDTs in MILC. More than 73% of them can be packed in less than one-seventh of the time when using runtime-compiled pack functions instead of Cray MPI. None of the used datatypes experienced slowdown.

the same level with manual packing, despite the additional overhead of the MPI wrapper. In case of MILC and WRF, our packing library outperforms manual packing by a large margin, since the original packing code in MILC uses index lists, while the data to be packed is in fact is in a regular form and can be packed with nested vectors.

3.2 Application Benchmarks

In this subsection, instead of analyzing the performance of single datatypes, we analyze all datatypes that occur in a complex application. To do this we intercept all calls to MPI_Type_commit. We create each committed datatype in our packing library and as an MPI datatype and record the overhead for this step. Then we use each datatype for packing and unpacking and again record the required time. To minimize the influence of OS jitter we report the median of several measurements.

Hoefler et al. demonstrated that the performance of MILC can be improved by up to 25% using (interpreted) MPI datatypes instead of the original pack code [8]. In this section we analyze which fraction of the 96 datatypes in MILC can benefit from runtime compilation techniques, and to what degree. Figure 8 shows a histogram of the different speedups achieved for each datatype in MILC when running the sample input file on eight processes. Most of the recorded datatypes can be packed with runtime-compiled

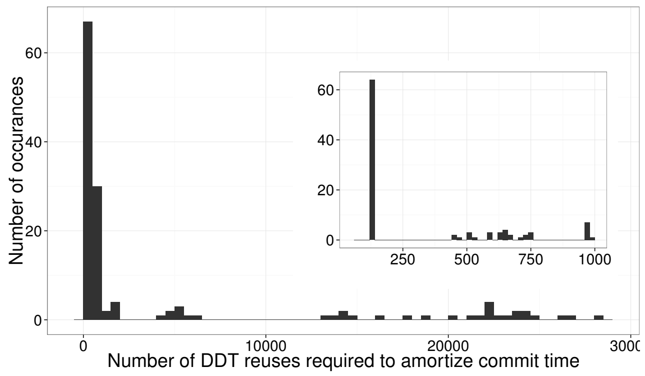


Figure 9: Histogram of necessary number of datatype reuses, until the higher performance of runtime compiled datatypes amortizes the higher cost of runtime compilation. The head of the distribution is plotted again in the smaller plot, with a different scale on the x-axis.

pack functions in less than one-seventh of the time it takes to do the same with MPI, some can be even be packed in 1/38th MPI's pack time. However, even with those high speedups compared to Cray MPI, the absolute improvement due to the usage of runtime-compiled pack functions is around 100μs. Therefore it is also important to take the commit overhead (the time for the runtime compilation) into account. The commit overhead lies between 1 and 10ms for runtime-compiled DDTs in MILC, while it is below 1μs for MPI. In Figure 9, we analyze how often each datatype would have to be reused in order to amortize the commit overhead. The tail of this distribution is quite large, therefore we plot the head separately, with a different scale on the x-axis.

We see that 50% of the datatypes in MILC need to be reused at least 520 times, for runtime compilation to pay off (a typical MILC run consists of thousands of iterations with the same datatypes). A smaller number of types (e.g., some very small types) requires a very high number of reuses. This shows that runtime compilation is not always beneficial and the hints discussed in Section 2.5 may guide the decision what method to apply. It is vital to know how much time can be saved by packing a datatype in a more efficient manner and how often it will be reused. The first information can be estimated using the size of the datatype, while the second information has to be supplied by the user with info arguments, as proposed earlier.

4. RELATED WORK

Since some MPI implementations struggle to fulfill elementary performance expectations when handling derived datatypes [6], their adoption is not widespread yet. More and more success stories about improving performance using MPI DDTs are reported [1,8] and tools are available that enable users to quickly change their code from using manual pack loops to leveraging derived datatypes [10].

There have been many publications on performance improvements of MPI derived datatypes. The naïve approach to interpreting datatypes is to represent them as a tree, where leaf nodes represent primitive datatypes and complex datatypes such as vectors or structs are represented by internal nodes. A pack operation is performed by traversing the tree post-order, recursively calling the interpreter for each intermediate node. The interpreter must keep track of the current input and output buffer positions, which are updated whenever a leaf node is encountered and data is actually moved. The drawback of this method is that inner nodes have to be visited multiple times and the recursive function calls incur a high overhead. Another naïve approach is to flatten the datatype completely. This inflates its representation in memory as it does not utilize regularity present in the datatype. Therefore hybrid approaches have been proposed [7, 17], where small sub-trees are flattened on the fly during packing. These approaches partially transform the datatype tree into a stack which can be processed iteratively. Jenkins et al. proposed a different representation of datatypes which is suitable for processing by a pack-kernel running on a GPU as it exposes available parallelism [9]. Furthermore, Byna et al. show that tuning the datatype interpreter for the memory-hierarchy on the target machine (cache and page sizes, etc.) can lead to performance improvements [3]. They do this by providing different packing functions which are parametrized with information about the actual datatype as well as information about the memory hierarchy of the node. However, the packing functions itself are compiled together with the rest of the MPI code, so this requires branching to select the correct pack variant at runtime and some optimizations, such as loop unrolling, can only be performed for a fixed number of cases.

The runtime compilation approach can be used to implement all of these optimizations. However, in contrast to the previous work, we compile the datatypes at runtime when all parameters are available, so we can generate *specialized* native pack functions.

5. CONCLUSIONS

In this paper we demonstrate that runtime compilation of MPI datatypes can lead to speedups *up to an order of magnitude* in (un)pack performance over the interpretation approach. This makes raw datatype packing performance competitive to manual packing code. The performance comes at a one-time compilation cost of a few milliseconds per datatype at commit time, which we argue is well within the user's expectations and the intent of MPI_Type_commit.

Given these results we believe there is a strong case for adding support for runtime datatype compilation to MPI implementations. We propose adding a runtime datatype compiler with an LLVM backend, perhaps based on our code, to MPI implementations as an alternative to the current interpreted approach. In addition we propose adding the configure switch --with-llvm=<path>. If the path to LLVM is given, the datatype compiler is compiled into the MPI runtime and used to compile datatypes at commit time. This configure option would also have the added benefit of paving the way for other interesting uses of runtime and JIT compilation in MPI implementations in the future.

With runtime compilation techniques, like those outlined in this paper, we believe datatype performance will finally become superior to manual pack code in the vast number of cases. As the performance hit incurred by using datatypes on most systems disappear and datatypes become more widely used, investments in gather-scatter hardware that supports non-contiguous communication becomes viable, leading to even greater performance.

6. REFERENCES

[1] E. Bajrović and J. L. Träff. Using MPI derived datatypes in numerical libraries. In *EuroMPI'11*, pages 29–38. 2011.

[2] C. Bernard, M. Ogilvie, et al. Studying quarks and gluons on MIMD parallel computers. *Int. Journal of High Performance Computing Applications*, 5(4):61–70, 1991.

[3] S. Byna, W. Gropp, X.-H. Sun, and R. Thakur. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *ICCC'03*, 2003.

[4] L. Carrington, D. Komatitsch, et al. High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62K processors. In *ACM/IEEE conference on Supercomputing*, 2008.

[5] R. F. V. der Wijngaart and P. Wong. NAS parallel benchmarks version 2.4. Technical report, NAS Technical Report NAS-02-007, 2002.

[6] W. Gropp, T. Hoefler, R. Thakur, and J. L. Träff. Performance expectations and guidelines for MPI derived datatypes. In *EuroMPI'11*. 2011.

[7] W. Gropp, E. Lusk, and D. Swider. Improving the performance of MPI derived datatypes. In *Proceedings of the Third MPI Developers and Users Conference*. MPI Software Technology Press, 1999.

[8] T. Hoefler and S. Gottlieb. Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes. In *EuroMPI'10 Proceedings*, pages 132–141. 2010.

[9] J. Jenkins, J. Dinan, et al. Enabling fast, noncontiguous GPU data movement in hybrid MPI + GPU environments. In *CLUSTER 2012*, 2012.

[10] F. Kjolstad, T. Hoefler, and M. Snir. Automatic datatype generation and optimization. In *PPOPP'12 Poster Paper*. ACM, 2012.

[11] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004.

[12] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3. available at: `http://www.mpi-forum.org` (Sept. 2012).

[13] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions*, 100(7):845–858, 1987.

[14] T. Schneider, R. Gerstenberger, and T. Hoefler. Micro-applications for communication data access patterns and MPI datatypes. In *EuroMPI'12*, 2012.

[15] W. C. Skamarock and J. B. Klemp. A time-split nonhydrostatic atmospheric model for weather research and forecasting applications. *J. Comput. Phys.*, 227(7):3465–3485, 3 2008.

[16] M. ten Bruggencate and D. Roweth. DMAPP — an API for one-sided program models on Baker systems. In *Cray User Group Conference*, 2010.

[17] J. L. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In *EuroPVM/MPI'99*, pages 109–116. 1999.