DAME: A Runtime-Compiled Engine for Derived Datatypes

Tarun Prabhu
tprabhu2@illinois.edu

William Gropp
wgropp@illinois.edu

University of Illinois, Urbana-Champaign
Urbana, IL, USA

ABSTRACT

In order to achieve high performance on modern and future machines, applications need to make effective use of the complex, hierarchical memory system. Writing performance-portable code continues to be challenging since each architecture has unique memory access characteristics. In addition, some optimization decisions can only reasonably be made at runtime. This suggests that a two-pronged approach to address the challenge is required. First, provide the programmer with a means to express memory operations declaratively which will allow a runtime system to transparently access the memory in the best way and second, exploit runtime information. MPI's derived datatypes accomplish the former although their performance in current MPI implementations shows scope for improvement. JIT-compilation can be used for the latter. In this work, we present DAME — a language and interpreter that is used as the backend for MPI's derived datatypes. We also present DAME-L and DAME-X, two JIT-enabled implementations of DAME. All three implementations have been integrated into MPICH. We evaluate the performance of our implementations using DDTBench and two mini-applications written with MPI derived datatypes and obtain communication speedups of up to 20x and mini-application speedup of 3x.

CCS Concepts

•Software and its engineering → Runtime environments; *Just-in-time compilers*; Parallel programming languages;

Keywords

MPI, Derived datatypes, Data movement, JIT, HPC

1. INTRODUCTION

As we get ever closer to a full exascale machine, we see greater emphasis being placed on data. Not only are larger instances of compute-intensive problems being tackled, which

usually require larger amounts of memory, but there is also growing interest in data-intensive applications where the number computations performed per byte is lower than traditional numerical codes. See for instance [9, 11, 4]. Supercomputers of the future will almost certainly have to operate within a strict power budget which would require data movement, both within and across nodes, to be performed as efficiently as possible. It becomes apparent then that performance in the coming exascale era will likely be very dependent on making efficient use of the memory hierarchy.

However, modern memory hierarchies are deep and complex. Their performance characteristics vary across systems and extracting maximum performance out requires understanding both the hardware and the system software stack. So while high performance is already hard to extract, achieving portability in performance is significantly harder. Further, some information which can be used to improve performance, such as alignment, can often only be reasonably determined at runtime.

We are therefore faced with two challenges. One is that of describing data movement in a manner that enables the system software (compilers and runtimes) to perform the operation in the most efficient manner possible. The other is to take advantage of information that is available only at runtime in order to improve the performance even further.

One approach to addressing the first of these is already present in the form of derived datatypes in MPI. They allow the user to declaratively specify a memory layout. This specification is compiled to some intermediate representation (which could be optimized for the target platform) and is then processed in the manner best suited to the target. The derived datatypes are typically used in communication to perform pack (serialize) and unpack (deserialize) operations (In the remainder of this paper, when we refer to the pack operation, our statements will also implicitly apply to the unpack operation which is a straightforward inverse of packing. Any deviations from this will be made clear).

JIT compilation is a technique that can be used to exploit the information only available at runtime. Since the datatypes are already compiled to some intermediate representation (IR), choosing an IR which lends itself to JIT compilation would considerably simplify the process. The design of MPI's derived datatypes is such that there are natural “hooks” where this compilation can be performed.

Our contribution is DAME (**D**A**T**a **M**anipulation **E**ngine), a language to describe data moves which closely corresponds to MPI datatypes, and an interpreter to execute “programs” written in the language. DAME was intended to be used as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroMPI '15, September 21-23, 2015, Bordeaux, France

© 2015 ACM. ISBN 978-1-4503-3795-3/15/09...\$15.00

DOI: <http://dx.doi.org/10.1145/2802658.2802659>

```

double matrix[5][4];
double transpose[4][5];
long displs[] = { 0, 8, 16, 24 };
MPI_Datatype c, t_ddt;
MPI_Type_vector(5, 1, 4, MPI_DOUBLE, &c);
MPI_Type_create_hindexed_block
    (4, 1, displs, c, &t_ddt);
MPI_Type_commit(&t_ddt);
MPI_Pack(matrix, 1, t_ddt, transpose, ..);
MPI_Type_free(&c);
MPI_Type_free(&t_ddt);

```

Figure 1: Transpose of a matrix

a datatype processing engine in MPI, although it is general enough to be used in any situation where we might need to declaratively express data movement operations. Apart from the interpreter, we have also implemented two different JIT compilers for DAME to take advantage of information only available at runtime. These two JIT-enabled implementations, DAME-L and DAME-X were developed using LLVM [6] and a custom X86 code generator respectively. All three implementations have been integrated into MPICH.

The remainder of this paper is organized as follows. In Section 2, we present some background about MPI’s derived datatypes and some of the tools that we use to implement DAME and describe related work in this area. In Sections 3, 4 and 5, we describe the DAME language, the interpreter and the runtime-compilation enabled versions of DAME. In Section 6, we describe the results of experiments run with DAME before concluding in Section 7.

2. BACKGROUND AND RELATED WORK

In this section, we provide some background on the tools and platforms that we have used and discuss related work.

2.1 MPI Derived Datatypes

MPI’s derived datatypes provide a means to declaratively specify non-contiguous data layouts. Each derived datatype is recursively defined as a sequence of datatypes (which we shall refer to as “inner” types) and displacements between them. These “inner” datatypes could be basic types which correspond to the datatypes of C and Fortran (char, int etc.), or other derived datatypes. The derived datatypes can be composed to specify arbitrary memory layouts. These are typically used to serialize (pack) and deserialize (unpack) data prior to communication. For a more complete description, see [1]. Figure 1 is an example in pseudo-code showing how derived datatypes can be used to transpose a 5×4 matrix.

Before the derived datatype can be used, it must first be “committed” with a call to `MPI_Type_commit`. The MPI standard explicitly states that implementations could perform optimizations on the type or its representation here (see Chapter 4 of [1]). We refer to this phase as “commit-time” in the remainder of this paper.

However, the poor implementation of derived datatypes can dissuade users from adapting their code to use them. The expectation is that derived datatypes will perform slightly worse than manual packing code for simple types and prefer-

ably better than manual packing for more complex types. In practice, however, this is not always the case. In Figure 2, we compare the communication performance of some micro-applications from the DDTBench suite (for more details, see Section 6) which use MPI’s derived datatypes with several MPI implementations. As we can see, the performance obtained is inconsistent and it could very well hurt application performance to use these derived datatypes. Although better results have been reported [3, 7], this inconsistency seems to hamper the widespread use of derived datatypes. This, despite a considerable amount of work [2, 12, 13, 15] addressing the issue. In [13], Schneider et al also demonstrate runtime compilation for MPI datatypes. Our approach is, we believe, more comprehensive than theirs. That implementation did not support partial packing and was not fully integrated into an MPI implementation. Rather, they intercepted calls to the communication functions using the MPI profiling interface (PMPI). Their implementation also does not exploit partial packing which is used to pipeline packets over the network and is therefore an important optimization. All the experiments reported in this work have been carried out with a patched version of MPICH into which DAME has been fully integrated. Further, Schneider et al do not perform any memory layout optimizations (datatype normalization) when compiling the datatype. Since DAME was designed to be interpreted, the ability to perform such optimizations was taken into account when designing the language. During the runtime-compilation then, such optimizations are essentially “free”.

2.2 LLVM

LLVM [6] is a popular modular compiler framework. It has a type-safe, SSA-based intermediate representation (LLVM-IR) with backends for various architectures. It has a rich API which allows the user to construct an in-memory LLVM-IR and its own JIT compilation engine. This generates machine code on the fly and manages the buffers containing the code.

3. DAME

We now briefly describe the constraints we had in mind when designing DAME. We then describe the DAME language itself with a simple example.

Reduce interpretation overheads: Since DAME was primarily intended to be interpreted, we designed the language keeping in mind that we interpretation overhead had to be minimized. We also wanted to be able to perform optimizations on the language to reduce interpretation time.

Memory access optimizations: Since memory access patterns have a large impact on performance, we needed the ability to transform the program to optimize its memory access pattern.

Support for runtime compilation: Although it was primarily intended to be interpreted, we wished to retain the ability to quickly compile the representation down to machine code and execute that instead.

3.1 The DAME language

DAME is a simple primitive-based language. Most primitives correspond to an MPI derived datatype and may have

Figure 2: Speedup obtained by using derived datatypes over manual packing. The x-axis is the number of bytes packed

Category	Primitive	Description
Base	CONTIG VECTOR BLOCKINDEX INDEX STRUCT	These primitives correspond respectively to the Contig, Vector, Blockindexed, Indexed and Struct derived datatypes in MPI
Final	CONTIGFINAL VECTORFINAL BLOCKINDEXFINAL INDEXFINAL	Specifies that the innermost type is of the corresponding BASE type. Perform the actual moves during processing. In addition CONTIGFINAL simplifies partial packing
Control	CONTIGCHILD EXIT BOTTOM RETURNTO	Counts the number of inner type instances that have been processed Processing of a DAME program terminates when this is encountered Marks the end of a self-contained subprogram Controls where to go after processing an inner type of a STRUCT
Block1	VECTOR1 BLOCKINDEX1	Optimizations for when the blocklength of the corresponding BASE type is 1

Figure 3: Complete list of DAME primitives

parameters which fully describe it. For the primitives that correspond to an MPI derived datatype, the parameters are identical to those needed to describe the corresponding derived datatype. Figure 3 contains a complete list of DAME primitives. They are grouped into several categories, which we now describe.

Base: The **CONTIG**, **VECTOR**, **BLOCKINDEX**, **INDEX** and **STRUCT** primitives obviously correspond to the Contig, Vector, Blockindexed, Indexed and Struct types in MPI.

Final: These correspond to the innermost derived datatype in a derived datatype. For instance, if the datatype `d` was constructed as shown here,

```
MPI_Type_indexed(..., MPI_DOUBLE, &d);
```

the innermost derived datatype would be `d` itself. On the other hand, in the example in Figure 1, the innermost type is the vector type `c`. The inner type of all primitives in this category are basic types by definition. When the derived datatype is processed, the **Final** types are the ones that actually perform the data moves.

The **CONTIGFINAL** primitive performs a dual function. Since all datatypes are ultimately composed of basic types which are contiguous, the innermost primitive in all DAME programs will always be **CONTIGFINAL**. It also simplifies “partial” moves. As mentioned in Section 2.1, the derived datatypes are used to pack/unpack data prior to communication. A common optimization in such cases is to break the data to be

sent into fixed-size buffers which can be pipelined through the network. The operation being performed in this case is then a “partial” pack which will have a corresponding “resume” operation in which the remainder of the data will be packed resuming from the previous stopping point and copied into the next communication buffer. The presence of this **CONTIGFINAL** primitive in the innermost position of all DAME programs simplifies the partial pack and resume operations — the former because all checks for the availability of sufficient buffer space can be pushed into the code for **CONTIGFINAL**, and the latter because all resume operations will, by construction, resume in a **CONTIGFINAL**.

Control: The control primitives are present to simplify the DAME interpreter. **CONTIGCHILD** is present after all **Base** primitives whose blocklength is greater than 1. It counts the instances of the inner type that have been processed. Without this primitive, each of the other non-final types would have been responsible for keeping track of progress resulting in code duplication. **EXIT** is encountered when the program has finished processing the datatype. **BOTTOM** marks the end of a program or subprogram. A subprogram is one of the constituent datatypes of a **STRUCT** primitive. The **BOTTOM** primitive is only present for sanity checks and encountering it in the course of interpretation is an error. All **CONTIGFINAL** primitives must be followed by **BOTTOM**.

Block1: These are optimizations for types whose blocklength is 1. It simplifies their representation by eliminating the need for a **CONTIGCHILD** primitive.

3.2 DAME example

Since there are no conditional moves, a DAME program consists of an ordered list of primitives. A partial grammar for DAME is presented in Figure 4. For clarity we only discuss the primitives and their parameters in the grammar and do not show how comments and whitespace may be used in the program. However, we have formatted the examples of DAME presented in this paper for readability and have also included comments.

The DAME program for the example of Figure 1 is shown in Figure 5. The “inner” primitives have been indented. The depth of the indentation provides an indication of the nesting level of the type. This is only really useful for the **STRUCT** primitive since it has more than one “inner” primitive. Comment lines begin with a #.

Note that the entry point is the first primitive immediately “after” **EXIT**. We acknowledge that having the first instruction of a program be a termination specifier is somewhat counter-intuitive. DAME was primarily designed for ease and efficiency of interpretation. With this design, the program now has a single termination point which simplifies the interpreter (see Section 3.4).

We believe that this language satisfies the design considerations outlined at the beginning of this section. The various **Final** and **Control** primitives result in a relatively clean interpretation loop. The memory copy optimizations get pushed into processing code for the **Final** types. As discussed already, the **CONTIGFINAL** primitive simplifies the partial packing operation. Since the primitives are largely independent of one another, we can optimize memory access by reordering or merging them similar to traditional loop optimizations [10]. For runtime-compilation, since each DAME program is an ordered list of primitives and each primitive can be described as a parameterized sequence of machine instructions, the process of constructing an assembly language representation of the datatype is straightforward.

3.3 Commit-time optimizations

We now describe some of the optimizations that we alluded to in the previous section. These are all performed when a datatype is committed.

Normalization: This is the major optimization that we perform at commit-time. In normalization, for each type, we check if it can be described as more-restrictive type. For instance, when committing an indexed type, we check if the displacements and blocklengths are all constant. If that is the case, we create a more restrictive (and faster) vector type instead. This optimization becomes useful in some unexpected scenarios as well. For instance, [5] describes a method to automatically refactor manual packing code into datatypes. Although they do perform normalization as part of the process themselves, one could imagine an automated tool not carrying out this optimization, which would then be caught at runtime anyway. One can conceive of situations demanding vector types where the stride is usually greater than the blocklength, but on the occasions that they are equal, this will improve performance without the programmer having to introduce special cases.

This normalization is very beneficial in the case of **STRUCT** primitives. The control flow while interpreting these types is complex and compiling them is perhaps even more chal-

lenging. However, if the inner types of the struct have the same “signature” i.e. memory access pattern, they can be normalized into an **INDEXED** primitive. The simplest and most common occurrence of this pattern is when the inner types are all basic types. We see such cases particularly when C-structs are serialized.

Displacement sorting: On some modern architectures, random memory reads outperform random memory writes. If the range of displacements in an indexed type is large, the cache performance of unpacking the datatype is likely to be poor. By sorting the displacements, we can improve this by increasing the number of hits to the cache while writing and trading off poorer performance while reading. However, this does come at the expense of additional memory usage because the MPI standard requires that the user be able to “reconstruct” the datatype at any time i.e. obtain the parameters that were used to construct the type which requires both sorted and unsorted arrays to be saved. This behaviour can be customized using the **MPLT** control variable interface introduced in MPI-3.

Merging: If a **CONTIG** type appears as the inner type of any other type, we can, in most cases, merge the **CONTIG** into the blocklength parameter of the outer type. This reduces the number of primitives in the final program thereby improving the interpretation efficiency. The presence of the **CONTIGFINAL** primitive ensures that we needn’t be concerned about introducing special cases for partial packing.

Buffer size optimizations: Typically, the partial packing takes place in a buffer of a fixed size which is dependent on the communication layer used by the system. This parameter is set at configure-time (or install-time) of the MPI implementation. We could conceivably have an optimization which was aware of this size and transforms and wraps the datatype d into a new datatype d' such that every instance of d' would fit into the partial pack buffer eliminating the need to track partial packs at runtime.

Custom pattern detection: We can also perform pattern detection which can be useful for datatype normalization, or for very platform-specific optimizations. An example of the former is when the displacements of a blockindexed type are such that they result in a constant stride. In that case, if the displacement of the first block is zero, the type can be replaced with an equivalent vector type, and if not, with a **BLOCKINDEX1** with a vector inner type. As an example of the latter, vector types with specific blocklengths and strides can be packed with a single machine instruction on some platforms (e.g. **BLENDPS** instruction on X86). Although better suited to the runtime-compiled versions, this technique can be used in highly optimized, platform-specific implementations of DAME.

3.4 Interpreter

The DAME interpreter is organized as a stack machine. At first, the interpreter checks if the call is resuming from a partial pack. If so, it restores the program’s state from the supplied state object and jumps directly to **CONTIGFINAL** for that datatype. If the call is not a resume, the primitive immediately after **EXIT** is processed. At each non-final type,

<i>Prog</i>	→	<i>Exit</i> <i>TypePrim</i> <i>Bottom</i>
<i>TypePrim</i>	→	<i>Basic</i> <i>OtherFinal</i> <i>ContigFinal</i>
<i>Basic</i>	→	<i>NonStruct</i> <i>Struct</i>
<i>OtherFinal</i>	→	VECTORFINAL BLOCKINDEXEDFINAL INDEXEDFINAL (<i>Args</i>) <i>ContigFinal</i>
<i>ContigFinal</i>	→	CONTIGFINAL (<i>Args</i>)
<i>NonStruct</i>	→	VECTOR VECTOR1 BLOCKINDEXED BLOCKINDEXED1 INDEXED (<i>Args</i>) <i>TypePrim</i>
<i>Struct</i>	→	STRUCT (<i>Args</i>) <i>StructProgs</i>
<i>StructProgs</i>	→	<i>StructProg</i> <i>StructProg</i> <i>StructProgs</i>
<i>StructProg</i>	→	<i>ReturnTo</i> <i>TypePrim</i> <i>Bottom</i>
<i>Exit</i>	→	EXIT
<i>Bottom</i>	→	BOTTOM
<i>ReturnTo</i>	→	RETURNTO (<i>Args</i>)
<i>Params</i>	→	<i>Param</i> <i>Param</i> , <i>Argsx</i>
<i>Param</i>	→	$\mathbb{Z}$ <i>Array</i>
<i>Array</i>	→	[ ] [<i>ArrayElems</i>]
<i>ArrayElems</i>	→	$\mathbb{Z}$ $\mathbb{Z}$, <i>ArrayElems</i>

Figure 4: Partial DAME grammar. Whitespace, comments and separators omitted. $\mathbb{Z}$ refers to an integer

```

1 | EXIT
2 | BLOCKINDEXED1 (4, 1, [0, 8, 16, 24], 40)
3 |   VECTORFINAL (5, 1, 4, 8)
4 |   CONTIGFINAL (4)
5 | BOTTOM

```

Figure 5: DAME program for matrix transpose

the processing that is done is limited to updating pointer values indicating where in the source and destination buffers data is to be copied after which the next “inner” primitive is processed. In the **Final** primitives, the interpreter checks if there is sufficient buffer space to pack the entire type. If so, the entire type is packed and control pops to the previous primitive. Every transition to an inner or outer primitive is recorded in the state object. This process continues until control reaches the **EXIT** primitive, at which point the interpretation terminates successfully. If, while processing a **Final** type, the pack buffer space is found to be insufficient, the interpreter packs as many bytes as it can and returns indicating a partial pack was performed with the state object containing all the information needed to resume the pack correctly on the next call to pack.

4. DAME-L

An interpreter which packs a derived datatype will likely be outperformed by a compiled packing operation. The advantage of derived datatypes when it comes to using runtime-compilation as an optimization is that they represent a self-contained, encapsulated unit within the program. The decision of whether to runtime compile or not does not affect the programmer directly and code written using datatypes can benefit from this without modification. There are several immediate benefits of compilation over interpretation.

Switch Elimination: The `switch` statement that forms the heart of the interpreting loop is redundant since we know

the entire DAME program.

Alignment: The actual transfer of bytes takes place only in the **Final** primitives of a DAME program — all other types merely adjust pointers determining the read/write locations. Therefore, having alignment information is only relevant at the **Final** primitives, yet checking whether the pointers are aligned before every call to `memcpy` is potentially wasteful overhead. To avoid this overhead, we check if the datatype is n -aligned as follows.

- A datatype is n -aligned if and only if all the non-**Control** primitives except **CONTIGFINAL** are n -aligned (since the sole purpose of **CONTIGFINAL** is for partial packing, we assume that it is always unaligned).
- A non-**Final** primitive is n -aligned if the extent of the primitive is a multiple of n .
- A **VECTORFINAL** primitive is n -aligned if the stride is a multiple of n .
- A **BLOCKINDEXFINAL** or an **INDEXFINAL** primitive is n -aligned if all the displacements are multiples of n .
- A **STRUCT** primitive is n -aligned if all the displacements are multiples of n and all the inner types are n -aligned.

If a datatype is n -aligned, then passing an aligned buffer to the type during a pack operation will ensure that the pointer passed to all calls to `memcpy` will be aligned. When we compile the datatype representation, we can hoist the alignment check out of the innermost processing loop, thereby improving its performance.

Prefetching: Since complete information about the type layout and the memory access patterns of the type is known, we can determine whether or not prefetching is profitable. Prefetching will not be profitable if the entire type fits into the L1 cache. Depending on the latencies of the platform

on which the code is executing, the relative latency to other levels of the cache would also be a factor in determining the profitability of prefetching. In the current version of DAME-L, we have not implemented prefetching.

Since the operations performed by each DAME primitive are well-defined, we can easily use LLVM’s Builder API to generate a block of code for each primitive in a DAME program and arrange them in the order in which it would have been processed by the interpreter. We then use LLVM’s MCJIT module (which is a module specifically designed to support JIT compilation including support for lazy code generation) to compile the code and get a pointer to it which we then use whenever we need to pack the type.

5. DAME-X

Although LLVM has a convenient JIT compilation module and the infrastructure to control degree of optimizations, its code-generation pass incurs a significant fixed penalty which cannot be easily avoided (see Figure 6). If the datatype is not reused often, this overhead can completely negate all gains obtained by runtime-compilation.

In order to demonstrate that it is possible to do the runtime-compilation efficiently while still obtaining a significant performance improvement, we developed a custom backend for DAME which compiles the DAME representation using XED as the instruction opcode generator for X86 platforms. XED is the opcode generator used in PIN, a binary instrumentation tool [8] from Intel. The purpose of this version was to demonstrate that a fast, customized compiler could be written since the language that we are compiling is very simple and uses only a small subset of the vast X86 instruction set. We did not add support for optimizations other than unrolling and vectorization in this version. Other optimizations in DAME-L did not have an observable impact on performance (see Section 6.2).

6. EXPERIMENTAL RESULTS

In this section we describe the results of studying the performance of DAME under various conditions. All our DAME-enabled MPICH implementations were tested on the Taub cluster at the University of Illinois. The cluster consists of 12-core Xeon E5 X5650 processors on each node with an Infiniband interconnect. The system provided implementations were MPICH 3.1.3, MVAPICH2 1.6 and OpenMPI 1.8.4. We also ran some tests on Blue Waters with Cray-MPICH 7.0.3. Blue Waters consists of AMD 6267 “Interlagos” processors with Cray’s GEMINI interconnect.

6.1 Communication performance

Since the derived datatypes are primarily used to pack data prior to communication, we first evaluate the communication performance of the interpreted DAME and the runtime-compiled DAME-L and DAME-X. The tests are taken from the DDTBench suite. The suite consists of micro-benchmarks that capture the communication patterns in various applications and benchmarks and describe them using MPI derived datatypes. This datatype is then used in ping-pong tests between two processes (except in the FFT2 test which is an all-to-all operation). The time used to generate the graphs are the median time to complete the ping-pong operation of at least 50 tests. In Figure 7, we plot

the speedup of the three DAME implementations over unmodified MPICH. This shows the impact of DAME over the existing MPI implementation. We also include MVAPICH performance in the graph for comparison. Clearly, in every case, DAME outperforms the existing MPICH implementation. However, the magnitude of the speedup varies (note the different scales on the Y-axis in each of the subgraphs).

In Figure 8, we compare the performance of these same datatypes against the cost of manually packing them. We have included the performance of various implementations used in Figure 2 (the bar plot) for comparison¹. The interpreted DAME does not necessarily outperform manual packing. Manual packing offers superior performance particularly when the packing code is simple (for instance, just a two-deep loop nest with constant strided access). Compilers are easily able to optimize such loop nests. DAME, however, incurs interpretation overhead which degrades its performance relative to manual packing. However, the compiled DAME implementations often outperform manual packing. This is because, for the same simple types, the code generated by the runtime compilers is similar, though not identical, to the manual packing code. In particular, the generated code contains all the support necessary to enable partial packing. As discussed earlier, partial packing is an important communication optimization and for the larger buffer sizes, we can see this be effective as the runtime-compiled DAME implementations consistently outperform manual packing. For the smaller buffer sizes though, this optimization is not used, but the overhead of checking whether a partial pack is necessary negatively impacts performance.

In Figure 9 and 10, we show the manual packing code for the WRF_y-vec test case (this simulates the memory access pattern for WRF — a weather simulation code) and the DAME program for the same. Note that in the manual packing code, we see array dereferences and function calls in the innermost loop both of which can hamper performance. On the other hand, in the DAME code, the innermost loops contain **VECFINAL** primitives which are runtime-compiled to doubly-nested loops in DAME-L and DAME-X. The resulting packing code is thus, much simpler than the manual packing code.

6.2 Impact of compiler optimizations

We enabled only a few optimizations in both DAME-L and DAME-X. For short copies, we generate fully-unrolled, fully-vectorized loops. For medium-length copies, the copy loop is partially unrolled. For large copies, we use the system’s memcpy routine because we assume that it is optimized to perform large copies efficiently on the system. The thresholds for these options could be customized using MPI-3’s MPLT control variable interface. In Figure 11, we plot the speedup obtained for increasingly higher optimization levels (O1-O3) over no optimizations (O0) in DAME-L. We used the FFT2 mini-application (see Section 6.4). The bar-plot is the speedup in execution (left y-axis). The line plot is the slowdown (inverse speedup) in commit-time (right y-axis). Enabling any optimization level beyond O0 resulted in at least a 2x slowdown in commit-time with no improvement in application performance. For the smaller input sizes, the

¹In the subgraph for NAS_lu_y, for the smaller buffer sizes, the measured time was often lower than the resolution of the timer on the system. We chose to list them as having a speedup of 1 instead of omitting them.

	Create(μs)						Free(μs)					
	MH	MV	OM	DM	DM-L	DM-X	MH	MV	OM	DM	DM-L	DM-X
FFT2	14	9	11	12	153946	967	5	3	5	7	834	10
LAMMPS_full	351	553	816	72	439408	2636	1	1	99	13	1383	15
MILC_su3_zd	3	3	5	3	87115	462	0	1	0	1	308	2
NAS_LU_x	0	0	1	1	31624	235	0	0	0	1	110	1
NAS_LU_y	3	2	2	2	77356	425	0	0	1	1	232	2
NAS_MG_x	3	3	7	3	74376	464	1	1	3	0	248	2
NAS_MG_y	2	1	2	2	81799	432	0	0	1	0	258	2
NAS_MG_z	1	1	2	1	77971	431	0	0	1	1	230	2
SPECFEM3D_oc	37	36	221	28	68603	860	0	0	26	1	225	4
WRF_x_sa	188	188	12	20	502969	2959	1	1	2	7	2157	7
WRF_x_vec	170	175	7	9	487067	2634	1	1	1	6	2054	6
WRF_y_sa	32	35	12	37	506845	3015	1	1	2	10	2066	6
WRF_y_vec	38	39	13	9	483288	5032	3	2	2	6	1882	11

Figure 6: Datatype creation and deletion overheads for MPICH, MVAPICH, OpenMPI, DAME, DAME-L and DAME-X

Figure 7: Speedup of communication of micro-applications over MPICH. The x-axis is the number of bytes packed

slowdown in execution time is because the runs were very short and the compilation overhead was comparable to the total execution time of the application. We believe that the ineffectiveness of compiler optimizations is a consequence of the very limited behavior of the code that we are compiling — specifically that it consists entirely of memory copy operations with almost no reuse and very simple pointer arithmetic. Most aggressive compiler optimizations improve performance by maximizing data reuse and hardware resource utilization. In the absence of both, most compiler optimizations have little to no effect.

6.3 Impact of knowing alignment

We also compared the performance of two versions of

DAME-X, one of which uses SSE instructions for aligned-only data where possible (MOVAPS) and the other which exclusively uses the corresponding unaligned instructions (MOVUPS). MOVUPS can be used to copy aligned data as well. Surprisingly, we did not see any significant performance difference between the two. In order to determine if this was a characteristic of our test system, we wrote two different implementations of memcpy in X86 assembly language — the first of which exclusively used MOVAPS and the other which exclusively used MOVUPS. Once again, there was no measurable difference in performance between the two versions when the input arguments were 16-byte aligned. We observed this behaviour on both Taub (Xeon E5) and Blue Waters (AMD Interlagos) as well as a few other lo-

Figure 8: Speedup of derived datatypes over manual packing. The X-axis shows the number of bytes packed

```

for(m = 0; m < 4; m++)
  for(k = js; k <= je; k++)
    for(l = is; l <= ie; l++)
      buffer[c++] = *(a2Ds[m]+i2D(l,k,d1));
for(m = 0; m < 3; m++)
  for(k = js; k <= je; k++)
    for(l = ks; l <= ke; l++)
      for(n = is; n <= ie; n++)
        buffer[c++] =
          *(a3Ds[m]+i3D(n,l,k,d1,d2));
for(m = 0; m < 2; m++)
  for(k = first; k < lim4Ds[m]; k++)
    for(l = js; l <= je; l++)
      for(n = ks; n <= ke; n++)
        for(o = is; o <= ie; o++)
          buffer[c++] =
            *(a4Ds[m]+i4D(o,n,l,k,d1,d2,d3));

```

Figure 9: Manual packing code in WRF_y_vec

cal systems that we had access to. We then ran the same memcpy experiment on an older machine equipped with a Core2Duo T8300 processor from 2008. On this machine, as expected, MOVUPS was approximately 50% slower than MOVAPS when copying between two 16-byte aligned buffers. This suggests that on modern X86 processors by Intel and AMD, there is no benefit in using the aligned-only instructions — the unaligned move instructions for unaligned data are effectively as fast in the aligned data case, and also work with unaligned data. This is a recent development for X86 processors which might not be the case on other platforms. Our JIT compilation approach should be beneficial in those cases.

```

1  # Struct contains 9 inner types
2  EXIT
3  STRUCT(9, ...)
4  # Inner types 0-3 are as below
5  RETURNTO(3)
6  VECFINAL
7  CONTIGFINAL
8  BOTTOM
9  # Inner types 4-6 are as below
10 RETURNTO(3)
11 VECTOR1
12 VECFINAL
13 CONTIGFINAL
14 BOTTOM
15 # Inner types: 7-8 are as below
16 RETURNTO(3)
17 VECTOR1
18 VECFINAL
19 CONTIGFINAL
20 BOTTOM
21 BOTTOM

```

Figure 10: DAME program corresponding to Figure 9. Parameters for most primitives have been omitted for clarity

6.4 Mini-applications

Finally, we ran the two mini-applications from [3] with the various implementations of DAME. FFT2 performs a two-dimensional Fast Fourier transform. The number of processes and the input parameters are mentioned in the captions of the respective figures. In Figure 12, we show the overall speedup obtained for each of the DAME implemen-

Figure 11: Effect of compiler optimizations on FFT2

Figure 12: Overall speedup of FFT2 mini-app over MPICH

tations over MPICH. We also plot the corresponding performance of MVAPICH. The application was run for $p = 2$ processors and 100 iterations. The input sizes from 256 to 1536 correspond directly to the buffer sizes which show the communication time in Figure 7. The overall speedup is lower than the speedup reported there because the computation cost in the application is not sped up.

Figure 13 is a mini-application based on the MIMD Lattice Computation (MILC) quantum chromodynamics code. The mini-application was run with 4 processes. As we can see, there is only a very marginal improvement in performance with DAME and DAME-X. The code spends less than 10% of time communicating, and any improvement in communication performance has very little impact on the overall application performance. However, the high LLVM overheads causes DAME-L to be much slower for the smaller problem sizes. For the larger ones, we observed that nearly 20% of time was spent in the `MPI_Wait` function. We are currently attempting to determine why this is occurring. The right subgraph in Figure 13 shows the relative speedup in total communication while ignoring the time spent in `MPI_Wait`. We can see that the DAME variants perform well with speedup diminishing with problem size.

7. CONCLUSIONS

We have presented DAME, a declarative platform independent language to describe data movement. We present an interpreter for DAME which we have used as the backend to implement MPI’s derived datatypes and integrated this into MPICH. We also implemented two versions of DAME

with support for runtime-compilation — one using LLVM as a backend and the other using a PIN-based custom code generator. We demonstrate DAME’s effectiveness by comparing its performance to both existing MPI implementations and manual packing on a suite of micro-benchmarks extracted from widely used applications where we obtain speedups up to 20x for communication. When comparing the total execution time of micro-applications, we obtain a speedup of up to 3x for FFT2, and match the performance of MILC. We show that the dynamically-compiled DAME implementations often outperform manual packing when the interpreted datatype engines prove to be slower. While evaluating DAME, we observed that several modern X86 processors do not incur a performance penalty when using unaligned load/store instructions on aligned data. We demonstrate that it is both feasible and profitable to implement a custom compiler backend for restricted uses such as this. Since the backend is quite simple, retargeting it to other platforms, we believe, would not require significant effort. The fully-featured compiler (DAME-L) performs better on more complicated datatypes and probably should be used only in those cases. It may be possible to improve the performance of DAME-X in those cases.

We believe that for future exascale machines, this approach will prove effective in both obtaining performance on a given platform and simplifying the task of writing and maintaining performance-portable code.

8. FUTURE WORK

Since the runtime compilation incurs a substantial compilation overhead, the programmer may not wish the datatype to be runtime compiled, especially if it is not used often enough to overcome these overheads. MPI Datatypes can have attributes attached to them. The programmer could use this mechanism to provide hints to the runtime system as to whether the type would benefit from runtime compilation. The `MPI_Info` objects could also be used for this purpose, however this would require changes to the application code since a different routine would have to be used. A more sophisticated approach which does not involve programmer intervention would be to use static analysis to determine how often the derived datatype is likely to be used and pass this information to the MPI runtime using either of the two mechanisms. More recent work involving a more general approach to the type reconstruction problem might afford insights into how best to normalize datatypes [14]. Def-use information from more sophisticated MPI-aware program analyses might also be incorporated to refine the instruction selection phase of the runtime-compilation. Specifically, knowing how soon the buffers used in the pack operation were to be reused, we might choose between temporal and non-temporal store instructions to avoid polluting the cache. This could have a positive impact if communication and computation were well-overlapped and the pressure on the memory system was significant. A more careful analysis of the performance characteristics of the various DAME implementations and their relationship with manual packing would be useful for a programmer to determine whether or not to use MPI’s derived datatypes for a given task.

9. ACKNOWLEDGEMENTS

This material is based [in part] upon work supported by

Figure 13: Overall and non-wait speedup for MILC over MPICH. The x-axis is the number of points per dimension

the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

10. REFERENCES

- [1] MPI: A Message-Passing interface standard. version 3.0. Retrieved from <http://www.mpi-forum.org> (Jan. 2015).
- [2] S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. Automatic memory optimizations for improving MPI derived datatype performance. In B. Mohr, J. Träff, J. Worringer, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 238–246. Springer Berlin Heidelberg, 2006.
- [3] T. Hoefer and S. Gottlieb. Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, pages 132–141, Berlin, Heidelberg, 2010. Springer-Verlag.
- [4] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. ACM.
- [5] F. Kjolstad, T. Hoefer, and M. Snir. A transformation to convert packing code to compact datatypes for efficient Zero-Copy data transfer. Technical report, University of Illinois, Urbana-Champaign, Aug. 2011.
- [6] 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*, volume 0, pages 75–86, Los Alamitos, CA, USA, Mar. 2004. IEEE.
- [7] Q. Lu, J. Wu, D. Panda, and P. Sadayappan. Applying MPI derived datatypes to the NAS benchmarks: A case study. In *Parallel Processing Workshops, 2004. ICPP 2004 Workshops. Proceedings. 2004 International Conference on*, pages 538–545. IEEE, Aug. 2004.
- [8] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 40 of *PLDI '05*, pages 190–200, New York, NY, USA, June 2005. ACM.
- [9] A. McLaughlin and D. A. Bader. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 572–583, Piscataway, NJ, USA, 2014. IEEE Press.
- [10] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, Dec. 1986.
- [11] R. Pearce, M. Gokhale, and N. M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 549–559, Piscataway, NJ, USA, 2014. IEEE Press.
- [12] R. Ross, N. Miller, and W. Gropp. Implementing fast and reusable datatype processing. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 404–413. Springer Berlin Heidelberg, 2003.
- [13] T. Schneider, F. Kjolstad, and T. Hoefer. MPI datatype processing using runtime compilation. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 19–24, New York, NY, USA, 2013. ACM.
- [14] J. L. Träff. Optimal MPI datatype normalization for vector and index-block types. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, New York, NY, USA, 2014. ACM.
- [15] J. Wu, P. Wyckoff, and D. Panda. High performance implementation of MPI derived datatype communication over InfiniBand. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, volume 1, pages 14+, Los Alamitos, CA, USA, Apr. 2004. IEEE.