Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–15

DOI: 10.1177/1094342017695444
journals.sagepub.com/home/hpc
SAGE

DAME: Runtime-compilation for data movement

Tarun Prabhu and William Gropp

Abstract

Modern machines consist of multiple compute devices and complex memory hierarchies. For many applications, it is imperative that any data movement between and within the various compute devices be done as efficiently as possible in order to obtain maximum performance. However, hand-optimizing code for one architecture will likely sacrifice both performance portability and software maintainability. In addition, some optimization decisions are best made at runtime. This suggests that the problem ought to be tackled on two fronts. First, provide the programmer with a declarative language to describe data layouts and data motion. This would allow the runtime system to be tuned for each architecture by a specialist and free the programmer to concentrate on the application itself. Second, exploit the execution time information to optimize the data movement code further. MPI derived datatypes accomplish the former task and Just In Time (JIT) compilation can be used for the latter. In this paper, we present DAME—a language and interpreter designed to be used as the backend for MPI derived datatypes. We also present DAME-L and DAME-X, two JIT-enabled implementations of DAME, all of which have been integrated into MPICH. We evaluate their performance on DDTBench and two mini-applications written with MPI derived datatypes and obtain communication speedups of up to 20 $\times$ and mini-application speedups of up to 3 $\times$.

Keywords

MPI, derived datatypes, data movement, Just In Time compilation, High Performance Computing

1 Introduction

As the exascale-era moves ever closer, data management is fast-becoming the central issue governing performance. The size and complexity of the simulations being run on supercomputers keep growing with the increasing computational power and memory available on these machines. In addition, application scientists are beginning to run data-intensive applications which are characterized by a much smaller flops per byte ratio (Hong et al., 2013; McLaughlin and Bader, 2014; Pearce et al., 2014). This suggests that performance in the future will depend on the efficiently of moving data across the machine, both up and down a complex memory hierarchy on a single node and between unshared memory regions across nodes.

Efficient data movement on a single node itself can be challenging especially when the memory access pattern is non-trivial. Extracting maximum performance requires tailoring the code to the specific characteristics of the machine and might also depend on the system software stack being used. This often comes at the expense of performance portability—a trade-off that is rarely acceptable. Improving the quality of the compilers and runtime systems or providing them with the information to make better decisions about how to carry out the data movement can improve performance while simultaneously addressing the portability concerns.

However, some parameters such as alignment which does impact performance are often only readily obtained at runtime. Obtaining better performance would also then depend on effectively using this runtime information.

One approach to address the first of these is already present in the form of derived datatypes in MPIs. They allow the user to declaratively specify a memory layout. This specification is compiled to some intermediate representation and is then processed in the manner best suited to the target. The derived datatypes are typically

Department of Computer Science, University of Illinois, Urbana–Champaign, USA

Corresponding author:
Tarun Prabhu, Department of Computer Science, University of Illinois, 201 N Goodwin Avenue, Urbana, IL-61801, USA.
Email: tprabhu2@illinois.edu

used in communication to perform pack and unpack operations. These correspond to serialization and deserialization respectively. In the remainder of this paper, any statements that we make regarding packing will also implicitly apply to unpacking which is almost always a straightforward inverse of packing. Any deviations from this will be made clear.

To exploit runtime information, we propose to use just in time (JIT) compilation. This is a technique perhaps best known for its use in the Java Virtual Machine (JVM) (Cramer et al., 1997). When a Java program is compiled, it is compiled to an intermediate representation (bytecode) rather than to native machine code. This program is executed by the JVM which contains a bytecode interpreter. Under certain circumstances, the JVM compiles the bytecode to native machine code which is then directly executed. This step is referred to as JIT compilation. It eliminates the significant interpretation overhead but comes at the expense of some start-up time as the bytecode is compiled. JIT compilation has also been used in a similar manner in languages like Javascript (Gal et al., 2009; Santos et al., 2013) and Python (Akeret et al., 2015; Rigo, 2004; Lam et al., 2015).

As we shall demonstrate later in this paper, the performance of existing implementations of MPI derived datatypes is inconsistent and using them can sometimes be slower than writing manual packing code. This suggested that we needed a higher-quality datatype processing engine. This engine would need to support JIT compilation to take advantage of runtime information. Since JIT'ting incurs a fixed overhead that can only be overcome if there is sufficient reuse of the JIT'ted code, it would also need an efficient interpreter for cases where JIT was unsuitable. This was the primary motivation behind the development of DAME (**DA**ta **M**anipulation **E**ngine)—the major contribution in this work.[1]

DAME is both 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 a datatype processing engine in MPIs, 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, DAME-L and DAME-X. All three implementations have been integrated into MPICH.

The remainder of this paper is organized as follows. We begin with background material about MPI derived datatypes and describe some of the tools that we use to implement DAME. We also discuss related work in this area. We then proceed to describe the DAME language, interpreter and runtime-compilation enabled versions of DAME. We then present performance results obtained by running several micro-benchmarks and mini-applications with the various DAME implementations before concluding and discussing how this work could be expanded upon on in the future.

2 Background and related work

In this section, we provide some background on MPI derived datatypes and discuss related work.

MPI derived datatypes provide a means to declaratively specify data layouts. They are typically used to serialize (pack) and deserialize (unpack) data prior to communication. Each derived datatype is recursively defined as a sequence of datatypes. We shall refer to all datatypes after the first in such a sequence as "inner" datatypes. 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. In the simplest case, a derived datatype would describe a single block of contiguous memory. The next simplest case would be a sequence of blocks separated by a fixed displacement. In the most general case, the layout would consist of variable-length chunks of contiguous data separated by variable-length displacements. The various derived datatypes supported by MPIs are listed in Figure 1. The parameters column lists the parameters that are sufficient to fully describe the type.[2]

Once the datatype has been declared, it must be "committed" before it can be used. This is done 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 mpi). We refer to this phase as "commit-time" in the remainder of this paper.

Figure 2 is an example in pseudo-code showing how derived datatypes can be used to transpose a 5×4 matrix of integers. The vector type `c` initialized on line 5 represents the columns of the matrix. There are 5 rows in the matrix, hence the count of elements is 5. Each block consists of a single integer, so the blocklength is 1. The next element is 16 bytes away (the matrix consists of 4 columns of integers and we assume that **`sizeof`**`(MPI_INT) == 4`). The blockindexed type initialized on line 6 represents the rows.

However, the often 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 better than manual packing for more complex types. In practice, however, this is not always the case. In Figure 3, we compare the communication performance of some micro-applications from the DDTBench suite (see section 'Results' for a description of DDTBench), which use MPI derived datatypes

Name	Parameters	Description
Contiguous	`count, type`	A contiguous sequence of instances of type.
Vector	`count, blocklength, stride, type`	Fixed-length sequence of instances of the inner type separated by a constant stride.
Blockindexed	`count, blocklength, displacements[], type`	Fixed-length sequence of instances of the inner type separated by a variable displacement.
Indexed	`count, blocklengths[], displacements[], type`	Variable-length sequence of instances of the inner type separated by a variable displacement.
Struct	`count, blocklengths[], displacements[], types[]`	Variable length sequence of several different types separated by a variable displacement.

Figure 1. MPI derived datatypes.

```
 1  int mat[4][3];
 2  int transpose[3][4];
 3  MPI_Datatype c, ddt;
 4  long disps[] = {0, 4, 8};
 5  MPI_Type_hvector(4, 1, 12, MPI_INT, &c);
 6  MPI_Type_create_hindexed_block
 7                 (3, 1, disps, c, &ddt);
 8  MPI_Type_commit(&ddt);
 9  // Some parameters omitted for clarity
10  MPI_Pack(mat, 1, ddt, transpose, ...);
11  MPI_Type_free(&c);
12  MPI_Type_free(&ddt);
```

Figure 2. Transpose of a matrix.

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 by, for instance, Hoefler and Gottlieb (2010) and Lu et al. (2004), this inconsistency seems to hamper the widespread use of derived datatypes. This, despite a considerable amount of work by Byna et al. (2006), Ross et al. (2003), Schneider et al. (2013) and Wu et al. (2004) among others addressing the issue.

Schneider et al. (2013) 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. (2013) do not perform any memory layout optimizations 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. When JIT compiling to native code, these optimizations are already performed and are therefore essentially "free".

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 originally intended to be used as the derived datatype engine in the MPI which would be interpreted, we designed the language keeping in mind that the interpretation overhead had to be minimized. We also wanted to be able to perform optimizations on DAME programs which would 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.

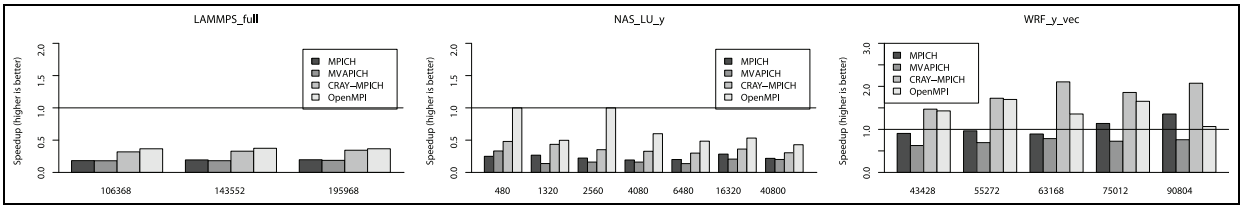

Figure 3. Speedup obtained by using derived datatypes over manual packing. The *x*-axis is the number of bytes packed.

Category	Primitive	Description
Base	CONTIG VECTOR BLOCKINDEX INDEX STRUCT	These primitives correspond respectively to the Contig, Vector, Blockindexed, Indexed and Struct derived datatypes in MPIs.
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 program or 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 4. Complete list of DAME primitives.

Support for runtime compilation. We wished to retain the ability to quickly compile the representation down to machine code and execute that instead of using the interpreter.

3.1 The DAME language

DAME is a simple primitive-based language. Most primitives correspond to an MPI derived datatype and may have 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 4 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 correspond to the Contig, Vector, Blockindexed, Indexed and Struct types in MPI.

Final: These correspond to derived datatypes that satisfy both of the following conditions.

– The datatype has a single inner type. (Since a Struct type in an MPI might have more than one inner type, there is no corresponding **StructFinal** primitive).
– The inner type is *not* a derived datatype.

When the derived datatype is processed, the **Final** types are the ones that actually perform the data moves since their inner types are basic types by definition.

The **CONTIGFINAL** primitive performs a dual function. Since all datatypes are ultimately composed of basic types that are contiguous, the innermost primitive in all DAME programs will always be **CONTIGFINAL**. It also simplifies "partial" pack operations. A common optimization when moving data over a network is to break it up into fixed-size fragments which are pipelined through the network. When a programmer calls, for instance, `MPI_Send`, she might expect that a single pack operation takes place before the data is pushed onto the network. In reality, however, the implementation might consist of several smaller packs—one pack for each fragment of data. We refer to each of these smaller packs as "partial" packs. Each partial pack has a corresponding "resume" operation which packs the next fragment. 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 once processing of the program is complete.
RETURNTO is encountered when processing of a subtype is complete. A subtype is an inner type of a Struct type. It is only used with a **STRUCT** primitive.
BOTTOM marks the end of a program or subprogram. It 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**. Every **EXIT** and **RETURNTO** must have a corresponding **BOTTOM**.

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

$$
\begin{array}{rcl}
Program & \rightarrow & \texttt{EXIT}\ Type\ \texttt{BOTTOM} \\
Type & \rightarrow & Basic \mid OtherFinal \mid ContigFinal \\
Basic & \rightarrow & NonStruct \mid Struct \\
OtherFinal & \rightarrow & \texttt{VECTORFINAL} \mid \texttt{BLOCKINDEXEDFINAL} \mid \texttt{INDEXEDFINAL}\,(Param, \ldots)\ ContigFinal \\
ContigFinal & \rightarrow & \texttt{CONTIGFINAL}(Param, \ldots) \\
NonStruct & \rightarrow & \texttt{VECTOR} \mid \texttt{VECTOR1} \mid \texttt{BLOCKINDEXED} \mid \texttt{BLOCKINDEXED1} \mid \texttt{INDEXED}\,(Param, \ldots)\ Type \\
Struct & \rightarrow & \texttt{STRUCT}(Param, \ldots)\ Subprogram^{+} \\
Subprogram & \rightarrow & \texttt{RETURNTO}(\mathbb{Z})\ Type\ \texttt{BOTTOM} \\
Param & \rightarrow & \mathbb{Z} \mid [\mathbb{Z}, \ldots]
\end{array}
$$

Figure 5. Partial DAME grammar. Whitespace, comments and separators have been omitted. *Z* refers to an integer. *Subprogram*$^{+}$ denotes one or more *Subprogram* . The ellipses indicate a comma-separated sequence.

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 5. 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. Specifically, the "inner" primitives have been indented with the depth of the indentation hinting at the nesting level of the type. Lines beginning with a # are comment lines.

The DAME program for the example of Figure 2 is shown in Figure 6. We shall now describe how this relates to the MPI code of Figure 2.

```
1  EXIT
2  # Args:(count, blklen, disps,
3  #       size(inner), extent(inner))
4  # Arg 1-3: from line 6
5  # Arg 4-5: calculated
6  BLOCKINDEXED1(3,1,[0,4,8],16,40)
7    # Args:(count, blklen, stride,
8    #  size(MPI_INT), extent(MPI_INT))
9    # Arg 1-3: from line 5
10   VECTORFINAL(4,1,16,4,4)
11     # Contiguous bytes in innermost
12     # block: 1 × size(MPI_INT)
13     CONTIGFINAL(4)
14 BOTTOM
```

Figure 6. DAME program for matrix transposing. Line numbers in the comments refer to Figure 2.

Line 1. The first primitive of the program is **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. For ease of programming, we recommend using a more "programmer-friendly" API or language to target DAME as we have done using MPI derived datatypes.

Line 6–13. The entry-point of the DAME program is the first primitive "after" **EXIT**. This is also where the immediate difference can been seen between an MPI program and a DAME program. While MPI datatypes are created in a bottom-up manner, a DAME program is written in a top-down style. In lines 5-6 of Figure 2, we see that the innermost type is declared first followed by the outer one. However, here, we declare the outer type first and move inwards. As mentioned earlier, the innermost primitive of a non-Struct type is always **CONTIGFINAL** (in the case of Struct types, the innermost primitive in each inner type will be **CONTIGFINAL**).

Line 14. The **BOTTOM** primitive follows **CONTIGFINAL** as required and also matches the **EXIT** from Line 1.

```
1 int mat[4][3];
2 int transpose[3][4];
3 MPI_Datatype c, ddt;
4 long disps[] = {0, 4, 8};
5 long blks[] = {1, 1, 1};
6 MPI_Datatype ts[] = { c };
7 MPI_Type_hvector(4, 1, 12, MPI_INT, &c);
8 MPI_Type_struct(3, blks, disps, ts,&ddt);
9 MPI_Type_commit(&ddt);
```

Figure 7. MPI example showing Struct.

Note that each primitive in Figure 6 accepts parameters that are very similar to those in the corresponding constructor of Figure 2. The additional parameters are the size and extent of the inner types. These are tedious to calculate by hand (see Chapter 4 of mpi for details). DAME does not support variables. This again highlights the need for a more programmer-friendly layer above DAME.

In the case of Struct types, the DAME program looks slightly different. Consider Figure 7 which is almost exactly the same as Figure 2 with the exception that the inner vector type `c` is inside a struct instead of a blockindexed type. The corresponding DAME program is in Figure 8. Notice that the inner vector type

```
 1  EXIT
 2  # Arg 4: array of inner sizes
 3  # Arg 5: array of inner extents
 4  STRUCT(3,[1,1,1],[0,4,8],[16],[40])
 5    # Goto statment 4 after each
 6    # inner type has been processed
 7    RETURNTO(4)
 8    VECTORFINAL(4,1,16,4,4)
 9      CONTIGFINAL(4)
10    BOTTOM
11  BOTTOM
```

Figure 8. DAME example showing Struct.

has a **RETURNTO** primitive before it. As we shall describe in more detail later, the DAME interpreter is organized as a stack machine and processing a DAME program essentially consists of push and pop operations on a stack. This is normally fairly straightforward since there is always a single path through the DAME program. The **STRUCT** primitive however introduces "branches" since it may have more than one inner type. In this case, when an inner type is done processing, control must return to the correct "parent" primitive. **RETURNTO** is used for this purpose. The integer parameter given to **RETURNTO** is the line number of the corresponding **STRUCT** primitive. Because every **RETURNTO** and **EXIT** must have a matching **BOTTOM**, there is an additional **BOTTOM** on line 10.

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 the 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 the memory access by reordering or merging them similar to traditional loop optimizations (Padua and Wolfe, 1986). 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 constrained 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 constrained (and faster) vector type instead. This optimization becomes useful in some unexpected scenarios as well. For instance, Kjolstad et al. (2011) 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 challenging. However, if the inner types of the struct have the same 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 behavior can be customized using the MPI_T 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 don't have to be concerned about the impact of this optimization on our ability to perform partial packs.

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 the configure-time (or install-time) of the MPI implementation. Although we have not done so, we could have an optimization that is 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 (for example, the 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. The most important item in the state object is the stack trace. Once restored, the interpreter can resume processing seamlessly. Since the previous partial pack ended at a **`CONTIGFINAL`** primitive, control immediately jumps to the appropriate **`CONTIGFINAL`** primitive in this case. If the call is not a resume, the primitive immediately after **`EXIT`** is processed. At each non-final type, 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

DAME-L is a JIT-enabled DAME implementation. We used LLVM (see Lattner and Adve (2004)) to perform the JIT compilation. LLVM 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 (lazily if required) and manages the buffers containing the code.

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

	Create (μs)						Free (μs)					
	MPICH	MVAPICH	OpenMPI	DAME	DAME-L	DAME-X	MPICH	MVAPICH	OpenMPI	DAME	DAME-L	DAME-X
FFT2	14	9	11	12	153,946	967	5	3	5	7	834	10
LAMMPS_full	351	553	816	72	439,408	2636	1	1	99	13	1383	15
MILC_su3_zd	3	3	5	3	87,115	462	0	1	0	1	308	2
NAS_LU_x	0	0	1	1	31,624	235	0	0	0	1	110	1
NAS_LU_y	3	2	2	2	77,356	425	0	0	1	1	232	2
NAS_MG_x	3	3	7	3	74,376	464	1	1	3	0	248	2
NAS_MG_y	2	1	2	2	81,799	432	0	0	1	0	258	2
NAS_MG_z	1	1	2	1	77,971	431	0	0	1	1	230	2
SPECFEM3D_oc	37	36	221	28	68,603	860	0	0	26	1	225	4
WRF_x_sa	188	188	12	20	502,969	2959	1	1	2	7	2157	7
WRF_x_vec	170	175	7	9	487,067	2634	1	1	1	6	2054	6
WRF_y_sa	32	35	12	37	506,845	3015	1	1	2	10	2066	6
WRF_y_vec	38	39	13	9	483,288	5032	3	2	2	6	1882	11

Figure 9. Datatype creation and deletion overheads (in μ s).

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 9). Despite the datatype being reused, the sheer size of this overhead completely negates any gains obtained by the runtime compilation.

In order to demonstrate that it is possible to do the runtime-compilation efficiently while still obtaining significant performance improvement, we developed a custom backend for DAME which compiles the DAME representation using XED. XED is the opcode generator used in Pin, a binary instrumentation tool for X86 platforms from Intel (see Luk et al. (2005)). 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 making the instruction selector very fast. The only optimizations implemented in this version are unrolling and vectorization.

6 Experimental results

In this section we describe the experiments we ran to test the performance of the various DAME implementations. 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 Open MPI 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 times plotted on the graphs are the median times to complete the ping-pong operation of at least 50 tests. In Figure 10, 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).

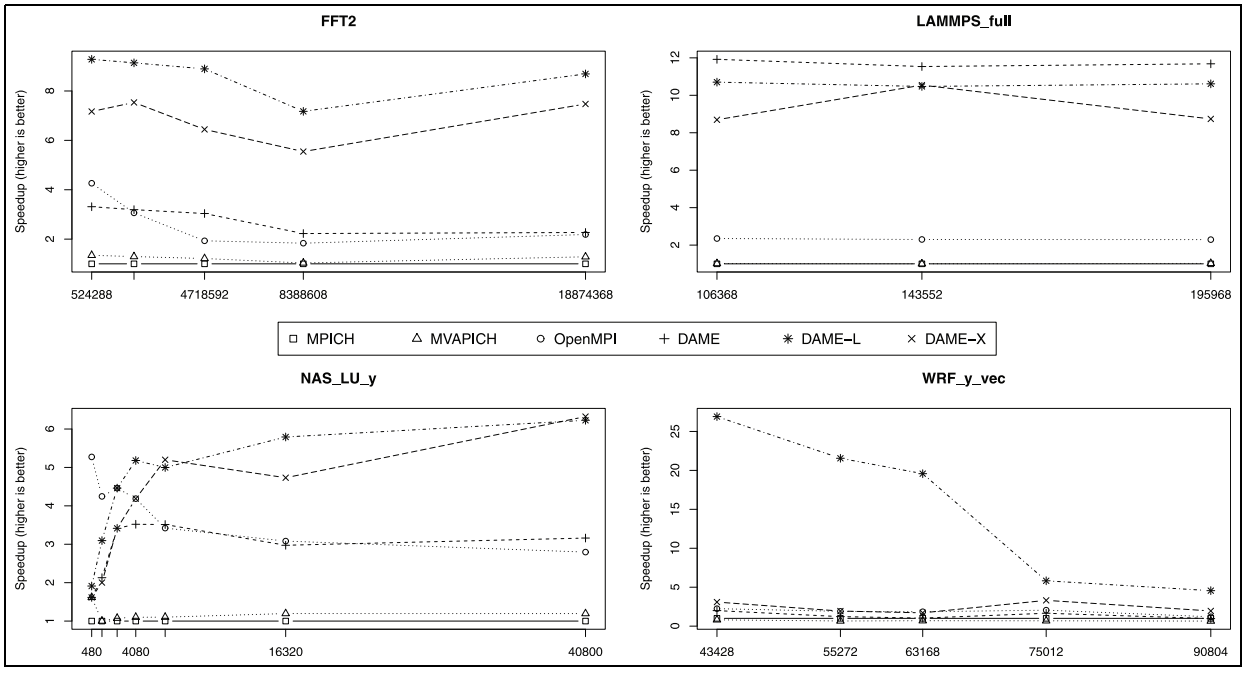

Figure 10. Speedup of communication of micro-applications over MPICH. The *x*-axis is the number of bytes packed.

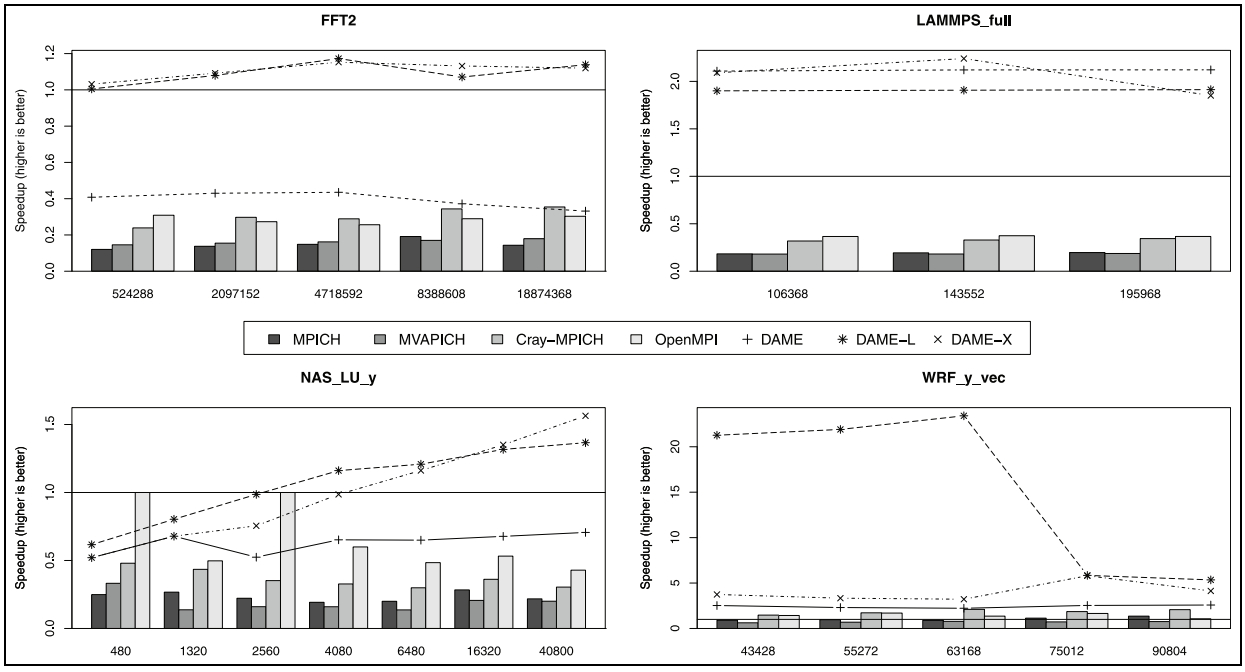

Figure 11. Speedup of derived datatypes over manual packing. The *x*-axis shows the number of bytes packed.

In Figure 11, 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 3 (the bar plot) for comparison[3]. 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

```
 1  for(m = 0; m < 4; m++)
 2   for(k = js; k <= je; k++)
 3    for(l = is; l <= ie; l++)
 4     buffer[c++] = *(a2Ds[m]+i2D(l,k,d1));
 5  for(m = 0; m < 3; m++)
 6   for(k = js; k <= je; k++)
 7    for(l = ks; l <= ke; l++)
 8     for(n = is; n <= ie; n++)
 9      buffer[c++] =
10        *(a3Ds[m]+i3D(n,l,k,d1,d2));
11  for(m = 0; m < 2; m++)
12   for(k = first; k < lim4Ds[m]; k++)
13    for(l = js; l <= je; l++)
14     for(n = ks; n <= ke; n++)
15      for(o = is; o <= ie; o++)
16       buffer[c++] =
17         *(a4Ds[m]+i4D(o,n,l,k,d1,d2,d3));
```

Figure 12. Manual packing code in `WRF_y_vec`.

```
 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 13. DAME program corresponding to Figure 12. Parameters for most primitives have been omitted for clarity.

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 which, as discussed earlier, is an important communication optimization when transferring large amounts of data. We can see the impact of this optimization as the runtime-compiled DAME implementations consistently outperform manual packing even when the interpreted DAME does not. However, when transferring small amounts of data, the optimization is not used and in that case, the overhead of checking whether a partial pack is necessary degrades performance.

In Figures 12 and 13, 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

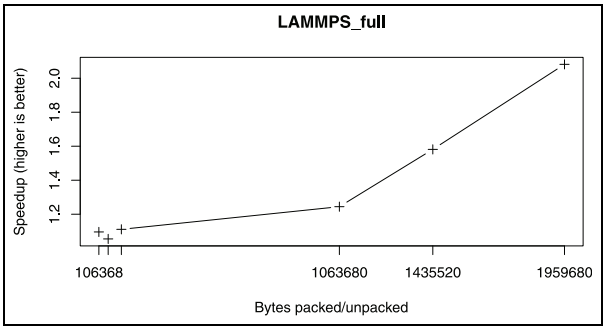

Figure 14. Impact of sorting displacements in indexed types.

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 code is simpler than the manual packing code.

6.2 Impact of displacement sorting

In Figure 14, we show the impact of sorting the displacements in an indexed datatype in regular, interpreted DAME. This does incur both a commit-time overhead to sort a possibly large array and a larger memory footprint. However, the improvement in performance can be substantial for large datatypes which do not fit in the processor's cache. For smaller problem instances, this optimization is not beneficial since the miss penalties are relatively small in the lower levels of cache. This suggests that it would be worthwhile to examine the values of the displacements, their order and distribution and the size of the derived datatype to decide whether or not to use this optimization.

6.3 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. We do not do so, but the thresholds for these options could be customized using MPI-3's MPI_T control variable interface. In Figure 15, we plot the speedup obtained for increasingly higher optimization levels (O1-O3) over no optimizations (O0) in DAME-L in the FFT2 mini-application (described in a later section). 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

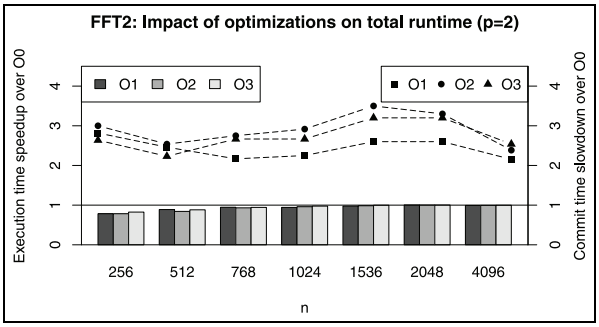

Figure 15. Effect of compiler optimizations on FFT2.

application performance. For the smaller input sizes, the 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.

6.4 Impact of knowing alignment

Figure 16 shows the time taken for some of the DDT benchmarks in DAME-X, one of which uses SSE instructions for aligned-only data (MOVAPS) where possible 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 (Note that in some of the figures, the time taken is in the order of tens of microseconds. The differences are due to system noise. For the tests which run for longer, the running times are almost the same). 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 behavior on Intel Xeon E5's (Taub), AMD Interlagos processors (Blue Waters) as well as a few other local systems to which we had access (running other modern Intel processors). 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 approach should be beneficial in those cases.

6.5 Overcoming overheads

Since JIT compilation incurs an overhead, it is only feasible to pay the cost if there is substantial reuse of the datatype. However, it may not be apparent at datatype construction time how many reuses will be necessary to overcome these overheads. Figure 17 plots the histograms for the number of reuses needed to overcome the compilation overhead for the datatypes in the DDTBench micro-benchmarks. As we can see, reusing the datatype 100 times overcomes the overheads in DAME-X (Figure 17b) for most of the datatypes we examined. However, the tail is quite long. There were also several types (which are not plotted) where JIT compilation was never beneficial. This is similar to the results reported by Schneider et al. (2013). Note that the x-axis in the case of DAME-L (Figure 17a) is approximately 100x that of DAME-X. This is consistent with the orders of magnitude difference in compilation overhead between the two versions reported in Figure 9. This highlights the need for additional tools or performance models to assist the programmer in

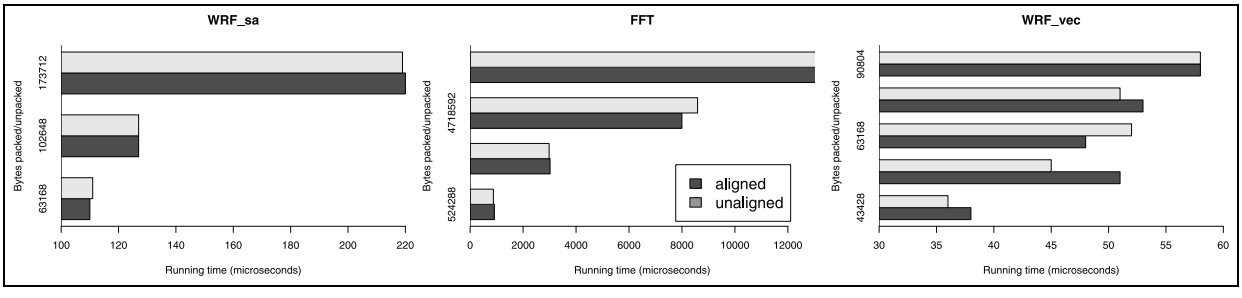

Figure 16. Time taken for calls to MPI_Pack (in μs) with aligned and unaligned instructions.

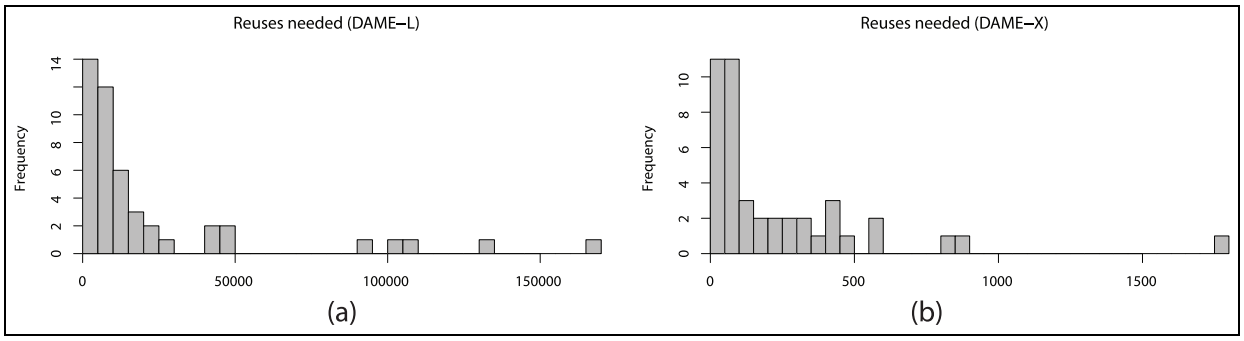

Figure 17. Number of reuses (on the *x*-axis) needed in the JIT-enabled DAME implementations to overcome compilation overhead.

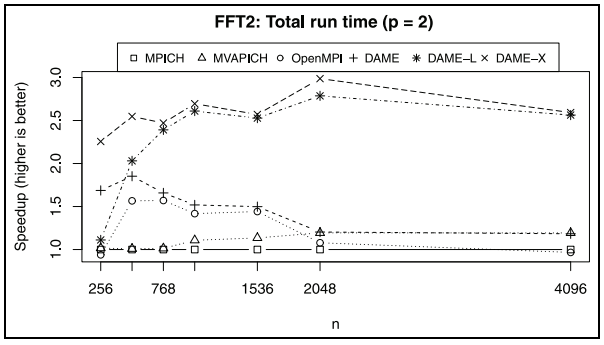

Figure 18. Overall speedup of FFT2 mini-app over MPICH.

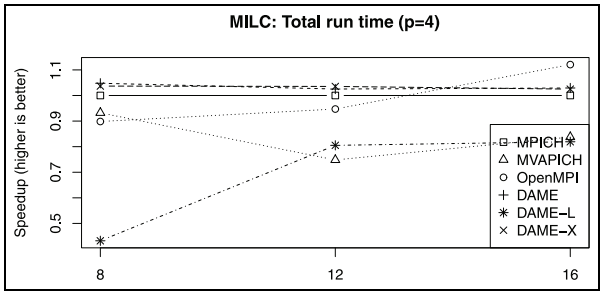

Figure 19. Overall and non-wait speedup for MILC over MPICH. The *x*-axis is the number of points per dimension.

determining when JIT compilation would be profitable for a datatype.

6.6 Mini-applications

We also ran the two mini-applications from Hoefler and Gottlieb (2010) with the various implementations of DAME.

6.6.1 FFT2. FFT2 performs a two-dimensional Fast Fourier transform. In Figure 18, we show the overall speedup obtained for each of the DAME implementations over MPICH. The application was run for $p = 2$ processors and 100 iterations. The input sizes from 256 to 1536 correspond directly to the buffer sizes shown in the communication time graph in Figure 10. The overall speedup is lower than the speedup reported there because the computation cost in the application is not sped up. Since the number of bytes being sent is large, the overheads in the interpretation limit the performance of DAME. However, DAME-L and DAME-X are faster because the interpretation overheads have been eliminated.

6.6.2 MILC. Figure 19 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. This is because the application 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. In Figure 20, we plot the communication speedup (over MPICH) excluding the time spent in `MPI_Wait` for two of the phases of the computation in MILC. As we can see, in both of them, the actual communication is sped up considerably using DAME. We observed similar results for the other phases as well, each of which use different derived datatypes. However, since the actual communication time is very short, there is not enough time to overcome the JIT compilation overheads.

7 Conclusions

We have presented DAME, a declarative platform-independent language to describe data memory layouts to be used in data motion. The language was designed to support both efficient interpretation and JIT-compilation for additional performance. DAME programs can be optimized both at the high-level when compiling

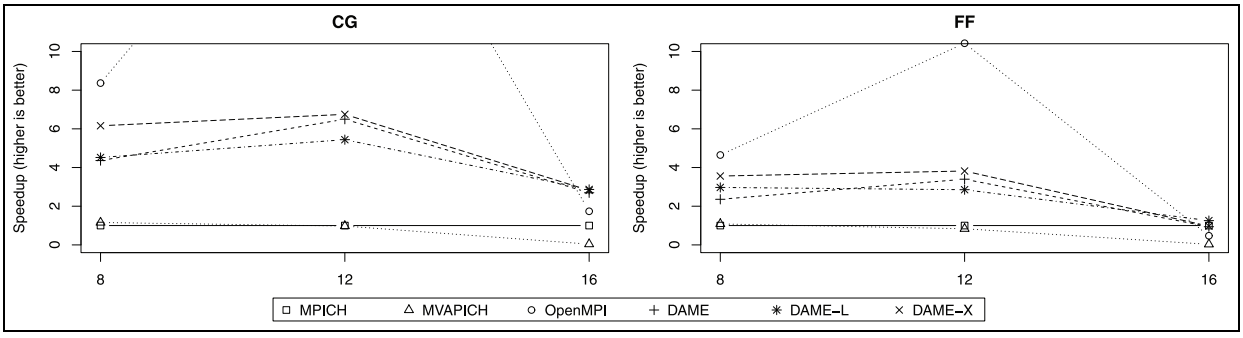

Figure 20. Speedup (over MPICH) of the various phases of computation in MILC, each of which uses different datatypes.

to an intermediate representation to be interpreted and again by the JIT compiler to take advantage of additional machine-specific features and runtime information. We used DAME as the backend for MPIs derived datatypes with the DAME interpreter effectively acting as the datatype processing engine in MPICH. We also implemented two different JIT compilers for DAME—one using industrial-strength LLVM as the backend that incurred significant compilation overheads and another hand-rolled special-purpose code generator using Pin. We tested 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 and two mini-applications that were rewritten to use MPI derived datatypes. DAME's interpreter was competitive with several MPI implementations and even outperformed them in many cases. With the JIT-compiled DAME implementations, we obtained speedups of up to 20x for communication in the micro-benchmarks. When comparing the total execution time of micro-applications, we obtain a speedup of up to 3x for FFT2. For MILC, we did no worse than other MPI implementations. We show that the JIT-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 DAME. Since all the programs that will be compiled have a fixed structure and perform only a limited set of operations, we need only handle a subset of the processor's instruction set. Because of this simplicity, we believe that retargeting it to other platforms should not require significant effort. The fully-featured compiler (DAME-L) performs better on more complicated datatypes. DAME-X's performance can be improved by implementing those optimizations from DAME-L which are shown to be beneficial.

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. `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 that 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 such as Träff (2014) involving a more general approach to the type reconstruction problem might afford insights into how best to normalize datatypes. 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 derived datatypes for a given task.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This material is based (in part) upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

Notes

1. A shorter version of this paper appeared in EuroMPI 2015 (Prabhu and Gropp, 2015).
2. Each of the types has a variant prefixed with an 'H'. In the normal case, the displacements are specified in multiples of the size of the inner type. In the 'H' variants, the displacements are specified as absolute bytes from the start of derived datatype. For a more complete description, see Chapter 4 in mpi.
3. In the subgraph for NAS_lu_y, when only a small amount of data was being sent, 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.

References

MPI Standards Committee (MPI Forum) (2015) A message-passing interface standard. Version 3.0. Technical report. Available at: http://www.mpi-forum.org/docs/ (accessed 23 October 2016).

Akeret J, Gamper L, Amara A, et al. (2015) HOPE: A Python just-in-time compiler for astrophysical computations. *Astronomy and Computing* 10: 1–8.

Byna S, Sun XH, Thakur R, et al. (2006) Automatic memory optimizations for improving MPI derived datatype performance. In: Mohr B, Träff J, Worringen J, et al. (eds) *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (*Lecture Notes in Computer Science*, volume 4192). Berlin Heidelberg: Springer, pp.238–246.

Cramer T, Friedman R, Miller T, et al. (1997) Compiling Java just in time. *IEEE Micro* 17(3): 36–43.

Gal A, Eich B, Shaver M, et al. (2009) Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Notices* 44(6): 465–478.

Hoefler T and Gottlieb S (2010) 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*, Stuttgart, Germany, 12–15 September 2010, pp.132–141. Berlin: Springer–Verlag.

Hong S, Rodia NC and Olukotun K (2013) 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*. Denver, Colorado, USA, 17–22 November 2013, pp. 1–12. New York: ACM.

Kjolstad F, Hoefler T and Snir M (2011) A transformation to convert packing code to compact datatypes for efficient zero-copy data transfer. Technical Report, University of Illinois, USA.

Lam SK, Pitrou A and Seibert S (2015) Numba: A LLVM-based Python JIT compiler. In: *Proceedings of the 2nd workshop on the LLVM compiler infrastructure in HPC*, Austin, Texas, USA, 15 November 2015, New York: ACM.

Lattner C and Adve V (2004) LLVM: A compilation framework for lifelong program analysis & transformation. In: *International symposium on code generation and optimization*, San Jose, California, USA, 22–24 March 2004, pp.75–86. Los Alamitos, CA: IEEE.

Lu Q, Wu J, Panda D, et al. (2004) Applying MPI derived datatypes to the NAS benchmarks: A case study. In: *International conference on parallel processing workshops, 2004. ICPP 2004 workshops*, Montreal, Quebec, Canada, 15–18 August 2004, pp.538–545. Washington DC, US: IEEE.

Luk CK, Cohn R, Muth R, et al. (2005) 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, Chicago, Illinois, USA, 11–15 June 2005, pp.190–200. New York: ACM.

McLaughlin A and Bader D (2014) Scalable and high performance betweenness centrality on the GPU. In: *High performance computing, networking, storage and analysis, SC14: International conference for*, New Orleans, Louisiana, USA, 16–21 November 2014, pp.572–583. Piscataway, NJ: IEEE.

Padua DA and Wolfe MJ (1986) Advanced compiler optimizations for supercomputers. *Communications of the ACM* 29(12): 1184–1201.

Pearce R, Gokhale M and Amato NM (2014) 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*, New Orleans, Louisiana, USA, 16–21 November 2014, pp.549–559. Piscataway, NJ: IEEE Press.

Prabhu T and Gropp W (2015) DAME: A runtime-compiled engine for derived datatypes. In: *Proceedings of the 22nd European MPI users' group meeting*, Bordeaux, France, 22–23 September 2015, pp. 35–45. New York: ACM.

Rigo A (2004) Representation-based just-in-time specialization and the Psyco prototype for Python. In: *Proceedings of the 2004 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, Verona, Italy, 24–25 August 2004, pp.15–26. New York: ACM.

Ross R, Miller N and Gropp W (2003) Implementing fast and reusable datatype processing. In: Dongarra J, Laforenza D and Orlando S (eds) *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, (*Lecture Notes in Computer Science*, volume 2840). Berlin Heidelberg: Springer, pp.404–413.

Santos HN, Alves P, Costa I, et al. (2013) Just-in-time value specialization. In: *Proceedings of the 2013 IEEE/ACM international symposium on code generation and*

optimization, Shenzhen, China, 23–27 February 2013, pp.1–11. Washington DC: IEEE Computer Society.

Schneider T, Kjolstad F and Hoefler T (2013) MPI datatype processing using runtime compilation. In: *Proceedings of the 20th European MPI users' group meeting*, Madrid, Spain, 15–18 September 2013, pp.19–24. New York: ACM.

Träff JL (2014) Optimal MPI datatype normalization for vector and index-block types. In: *Proceedings of the 21st European MPI users' group meeting*, Kobe, Japan, 9–12 September 2014, pp. 33–38. New York: ACM.

Wu J, Wyckoff P and Panda D (2004) High performance implementation of MPI derived datatype communication over InfiniBand. In: *Parallel and distributed processing symposium, 2004. Proceedings of the 18th international*, Santa Fe, New Mexico, USA, 26–30 April 2004, pp. 1–12. volume 1. Los Alamitos, CA: IEEE.

Author biographies

Tarun Prabhu is a PhD student in the Computer Science department at the University of Illinois, Urbana–Champaign. His interests are in compilers, static analysis and high-performance parallel computing. He is the lead developer of Moya, a JIT compiler for C, C++ and Fortran. He holds a Master's degree in Computer Science from the University of Utah and a Bachelor's degree in Computer Engineering from the University of Mumbai.

William Gropp is the Thomas M Siebel Chair in Computer Science at the University of Illinois, Urbana–Champaign and the Acting Director and Chief Scientist of the National Center for Supercomputing Applications. His research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. He has played a major role in the development of the MPI message-passing standard. He is co-author of the most widely used implementation of MPI, MPICH, and was involved in the MPI Forum as a chapter author for both MPI-1 and MPI-2. He is a Fellow of ACM, IEEE, and SIAM and a member of the National Academy of Engineering. He received the Sidney Fernbach Award from the IEEE Computer Society in 2008. He received a PhD in Computer Science from Stanford University, an MS in Physics from the University of Washington and a BS in Mathematics from Case Western Reserve University.