DAME: A Runtime-Compiled Engine for Derived Datatypes

Tarun Prabhu William Gropp

University of Illinois, Urbana-Champaign

What is DAME?

DAME is a language and interpreter specifically designed for data movement

What does it do?

A DAME program lets a user declaratively describe a data layout. The interpreter can then perform pack/unpack operations on this data layout in the most efficient manner possible
A DAME program can also be compiled using a JIT approach for even greater efficiency

Where is it used?

We patched MPICH to use the DAME interpreter as its datatype processing engine

Do we really need a data-movement language?

- Writing packing loops by hand can be cumbersome
- Hand-optimized packing loop nests may not have performance-portability
- Declarative loops allow user to specify only the high-level data layout and allow the runtime to pick the most efficient way of performing the packing

An example: Matrix transpose

```
do i = 1, 5
  do j = 1, 4
    b(i,j) = a(j,i)
```

```
do i = 1, 5
  do j = 1, 4
    b(j,i) = a(i,j)
```

- One implementation has sequential writes and strided reads, the other has strided writes and sequential reads.
- Relative performance is platform-dependent
- Neither efficient for cache-based systems.

MPI Datatypes

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

- Bit more verbose, but implementation can choose between strided writes and sequential writes

Are MPI datatypes always better?

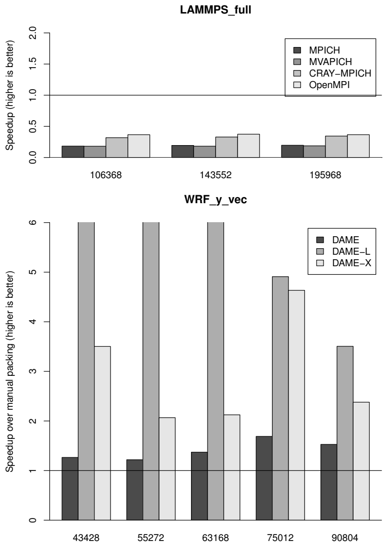


Figure: Communication speedup over manual packing

Why is the performance poor?

- Interpretation overhead
- No optimizations? (in manual packing, compiler can perform some optimizations)
- Poor choice of intermediate representation?
- . . .

These are just some possibilities.

So why runtime compilation?

- Reduce interpretation overhead
- Exploit runtime information. For instance knowing loop bounds can help compiler make better optimization decisions
- Let the compiler handle platform-specific information, e.g., cache sizes, instead of having the programmer do it all
- MPI datatypes are typically created once and reused often. Compilation overhead can be amortized

Design considerations

- Reduce interpretation overhead
- Maximize ability of compiler to optimize code. Expose as much as possible of user's program to compiler
- Simplify partial packing/unpacking as much as possible
  - Data may be transfered in packets; thus the pack/unpack code must be able to pause and resume. Keeping this requirement from impacting performance is key.
- Support for memory access optimizations
- Support for runtime compilation

DAME

DAME is a primitive-based language with an interpreter organized as a stack machine

Matrix transpose revisited

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

- **EXIT** and **BOTTOM** are control primitives
- The **Final** primitives indicate the innermost types. Exposes at least a doubly-nested loop to the compiler
- **CONTIGFINAL** simplifies partial packing
  - Not executed unless partial pack (or unpack)

DAME interpreter

1. Begins at first primitive after **`EXIT`**
2. Each primitive is “pushed” onto the interpreter stack
3. At each non-final primitive, only pointers are updated
4. Actual data is moved at each **`Final`** primitive. If packing can only be partially done, the maximum amount of data is packed including partial blocks
5. Terminate when **`EXIT`** is encountered

DAME — Optimizations made possible

- **`EXIT`** simplifies termination checks
- **`CONTIGFINAL`** simplifies resuming from partial packs because control jumps directly to this primitive to complete the last partially packed block
- In partial packing, the interpretation stack contains the entire state and resuming is as simple as restoring this stack
- Memory access optimizations can be done by shuffling primitives as desired. This is done at "commit" time.
- Other optimizations such as normalization, displacement sorting and merging can also easily be performed at commit-time.

Additional optimizations possible

- Alignment can be determined most accurately and appropriate instructions can be chosen
- Prefetching can be done more accurately because the sizes of the types and the cache are all known
- The main switch statement at the heart of interpretation loops is eliminated

Implementation I: DAME-L

First implementation using LLVM

- All the work of code generation, JIT'ted code management handled by LLVM's MCJIT API.
- Plenty of optimizations available
- Overhead was terrible (commit-time was ≈100000x slower than non-JIT'ted DAME)

Implementation II: DAME-X

Alternate implementation using XED[1]

- Custom opcode generator with support for a very limited subset of x86
- Much lower compile overhead (1000x faster than DAME-L)
- Limited optimizations enabled and will only work on x86

[1] XED is a part of PIN - a binary instrumentation tool

Evaluation

- Evaluated using DDTBench by Schneider et al[2]
- DAME implementation was MPICH patched to use DAME as the datatype processing engine
- Test machine was the Taub cluster at the University of Illinois consisting of 12-core Xeon E5 X5650 processors with an InfiniBand interconnect
- Cray MPICH was tested on Blue Waters to compare performance over manual packing

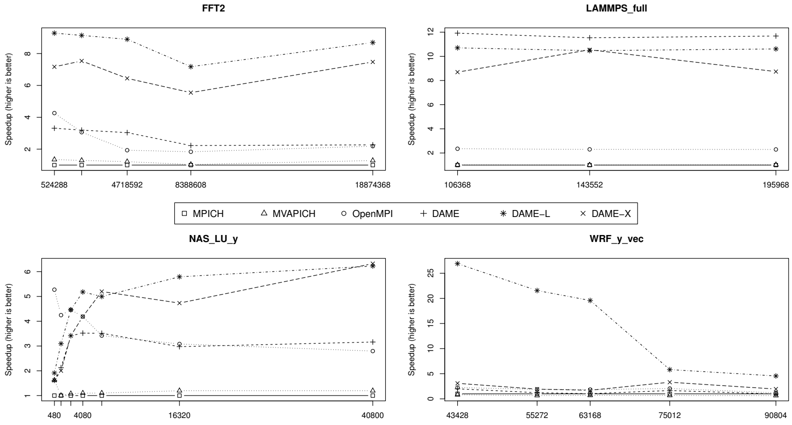
[2] T. Schneider, F. Kjolstad and T. Hoefler. MPI datatype processing using runtime compilation. EuroMPI '13

Datatype create/free overheads

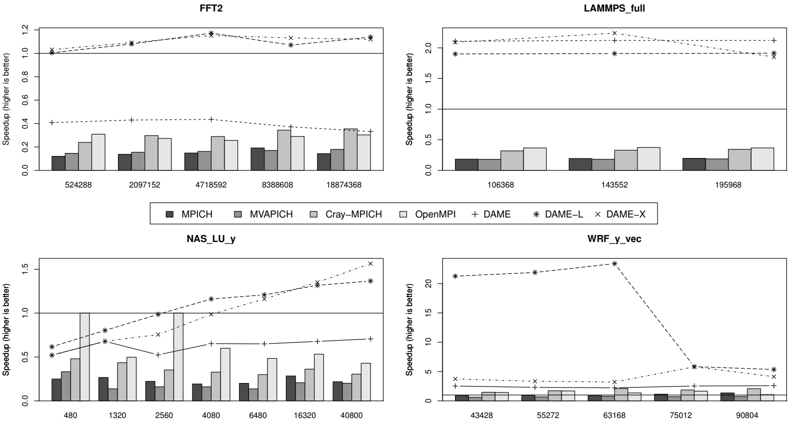
	Create(μs)				Free(μs)			
	OM	**DM**	**D-L**	**D-X**	**OM**	**DM**	**D-L**	**D-X**
FFT2	11	12	153946	967	5	7	834	10
LAMMPS	816	72	439408	2636	99	13	1383	15
MILC	5	3	87115	462	0	1	308	2
NAS_LU_x	1	1	31624	235	0	1	110	1
NAS_LU_y	2	2	77356	425	1	1	232	2
NAS_MG_x	7	3	74376	464	3	0	248	2
NAS_MG_y	2	2	81799	432	1	0	258	2
NAS_MG_z	2	1	77971	431	1	1	230	2

Figure: OpenMPI, DAME, Dame+LLVM, Dame+XED respectively

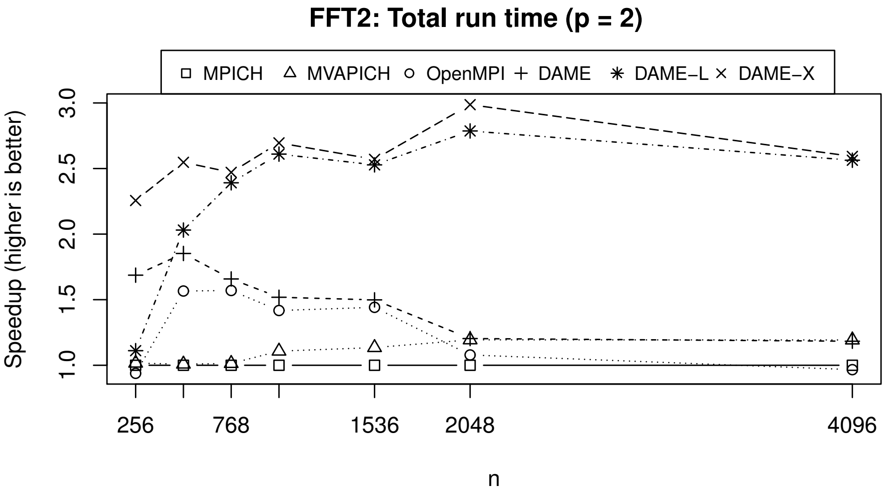
Communication speedup (p=2)



Communication speedup over manual packing (p=2)



Overall speedup in mini-app: FFT2[3]



[3] T. Hoefler and S.GottLieb. Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes. EuroMPI '10

Effect of compiler optimizations (DAME-L with FFT2)

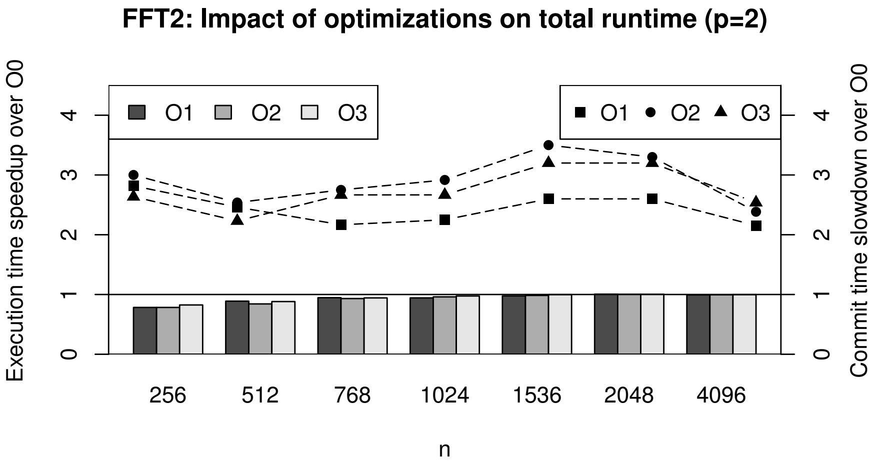


Figure: Bar graph is execution-time speedup over O0. Line graph is commit-time slowdown (inverse speedup)

Conclusions

- Implemented DAME, a JIT-enabled language for data movement as the datatype processing engine in MPICH
- Experiments with DDTBench — a suite of datatype benchmarks taken from real applications — shows consistent improvement in communication performance over existing MPI implementations
- JIT compilation improves the performance of DAME even further in many cases.
- A comparatively low-overhead special-purpose JIT compiler is beneficial and not impractical to implement