MPI Datatype Processing Using Runtime Compilation

Timo Schneider, Fredrik Kjolstad, Torsten Hoefler
WHAT YOUR VENDOR SOLD

Bandwidth [GB/s]

0 1 2 3 4 5

Datasize [Byte]

0K 100K 200K 300K

Data Transfer
Ping Pong

Timo Schneider, Fredrik Kjolstad, Torsten Hoefler
WHAT YOUR APPLICATIONS GET

10% of Ping-Pong performance
WHAT YOUR APPLICATIONS GET

Why?

10% of Ping-Pong performance

Bandwidth [GB/s]

Data Transfer
- Ping Pong
- MILC_su3
- LAMMPS
- NAS LU
- NAS MG
- WRF
- SPECFEM3D

Size [Byte]

0K

200K

300K

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Timo Schneider, Fredrik Kjolstad, Torsten Hoefler
WHAT YOUR APPLICATIONS GET

Why?

How to measure?

Why?
WHAT MPI OFFERS

Manual packing

```
sbuf = malloc(N*sizeof(double))
rbuf = malloc(N*sizeof(double))
for (i=1; i<N-1; ++i)
    sbuf[i]=data[i*N+N-1]
MPI_Isend(sbuf, ...)
MPI_Irecv(rbuf, ...)
MPI_Waitall(...)
for (i=1; i<N-1; ++i)
    data[i*N]=rbuf[i]
free(sbuf)
free(rbuf)
```

MPI Datatypes

```
MPI_Datatype nt
MPI_Type_vector(N-2, 1, N, MPI_DOUBLE, &nt)
MPI_Type_commit(&nt)
MPI_Isend(&data[N+N-1], 1, nt, ...)
MPI_Irecv(&data[N], 1, nt, ...)
MPI_Waitall(...)  
MPI_Type_free(&nt)
```

- No explicit copying
- Less code
- Often slower than manual packing (see [1])

[1] Schneider, Gerstenberger, Hoefler: Micro-Applications for Communication Data Access Patterns and MPI Datatypes
MPI DDTs are interpreted at runtime, while manual pack loops are compiled.

```
b = Vector(2, 1, 2, MPI_BYTE)
nt = Vector(N, 1, 4, bt)
```

**Internal Representation**

```
If (dt.type == VECTOR)
    for (int i=0; i<dt.count; i++) {
        tin = inbuf; tout=outbuf
        for (b=0; b<dt.blklen; d++) {
            interpret(dt.basetype, tin, tout)
        }
        tin += dt.stride * dt.base.extent
        tout = dt.blklen * dt.base.size
    }
    inbuf += dt.extent
    outbuf += dt.size
```
**INTERPRETATION VS. COMPILATION**

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

- None of these variables are known when this code is compiled
- Many nested loops

```c
If (dt.type == VECTOR)
  for (int i=0; i<dt.count; i++) {
    tin = inbuf; tout=outbuf;
    for (b=0; b<dt.blklen; d++) {
      interpret(dt.basetype, tin, tout)
    }
    tin += dt.stride * dt.base.extent
    tout = dt.blklen * dt.base.size
  }
  inbuf += dt.extent
  outbuf += dt.size
```

Timo Schneider, Fredrik Kjolstad, Torsten Hoefler
MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
for (int i=0; i<N; ++i) {
    for(j=0; j<2; ++j) {
        outbuf[j] = inbuf[j*2]
    }
    inbuf += 3*4
    outbuf += 2
}
```
MPI DDTs are interpreted at runtime, while manual pack loops are compiled.

```c
for (int i=0; i<N; ++i) {
    for (j=0; j<2; ++j) {
        outbuf[j] = inbuf[j*2]
    }
    inbuf += 3*4
    outbuf += 2
}
```

- Loop unrolling
MPI DDTs are interpreted at runtime, while manual pack loops are compiled.

```c
for (int i=0; i<N; ++i) {
    int j = 0
    outbuf[j] = inbuf[j*2]
    outbuf[j+1] = inbuf[(j+1)*2]
    inbuf += 3*4
    outbuf += 2
}
```

- Loop unrolling
- Constant Propagation
MPI DDTs are interpreted at runtime, while manual pack loops are compiled.

```c
for (int i=0; i<N; ++i) {
    outbuf[0] = inbuf[0]
    inbuf += 12
    outbuf += 2
}
```

- Loop unrolling
- Constant Propagation
- Strength reduction
MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```
bound = outbuf + 2*N
while (outbuf<bound) {
    outbuf[0] = inbuf[0]
    inbuf += 12
    outbuf += 2
}
```

• Loop unrolling
• Constant Propagation
• Strength reduction
**INTERPRETATION VS. COMPILATION**

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
bound = outbuf + 2*N
while (outbuf<bound) {
    outbuf[0] = inbuf[0]
    inbuf += 12
    outbuf += 2
}
```

- Loop unrolling
- Constant Propagation
- Strength reduction
- Unrolling of outer loop
MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
bound = outbuf + 2*N
while (outbuf<bound) {
    outbuf[0] = inbuf[0]
    inbuf += 12
    outbuf += 2
}
```

- Loop unrolling
- Constant Propagation
- Strength reduction
- Unrolling of outer loop
- SIMDization
**Runtime-Compiled Pack Functions**

- **MPI_Type_vector(cnt, blklen, ...)**
  - Record arguments in internal representation (Tree of C++ objects)

- **MPI_Type_commit(new_ddt)**
  - Generate pack(*in, cnt, *out) function using LLVM IR. Compile to machine code. Store f-pointer.

- **MPI_Send(cnt, buf, new_ddt,...)**
  - new_ddt.pack(buf, cnt tmpbuf)
  - PMPI_Send(...tmpbuf, MPI_BYTE)
COPYING BLOCKS

- Even for non-contiguous transfers, the “leaves” of the DDT are consecutive blocks.
- It is important that we copy those blocks as efficiently as possible.
- If the size of the cont. block is less the 256B we completely unroll the loop around it.
- Use fastest available instruction (SSE2 on our test system).
**BLOCK COPY PERFORMANCE**

In-cache measurement on AMD Interlagos CPU (Blue Waters test system)

Timo Schneider, Fredrik Kjolstad, Torsten Hoefler
Packing Vectors

- Vector count and size and extent of subtype are always known
- Use this to eliminate induction variables to reduce loop overhead
- Unroll loop for innermost loop 16 times
**Vector Packing Performance**

HVector(2,1,6144) of Vector(8,8,32) of Contig(6) of MPI_FLOAT

This datatype is used by the Quantum-Chromodynamics code MILC [2]

14x faster

In-cache measurement on AMD Interlagos CPU (Blue Waters test system)

IRREGULAR DATATYPES

Depending on index list length:

```
for (i=0; i<idx.len; i+=3) {
    inb0=load(idx[i+0])+inb
    inb1=load(idx[i+1])+inb
    inb2=load(idx[i+2])+inb
    // load oub and len
    copy(inb0, outb0, len0)
    copy(inb1, outb1, len1)
    copy(inb2, outb2, len2)
}
```

Minimize loop overhead by unrolling the loop over the index list

Inline indexes into code
IRREGULAR PACKING PERFORMANCE

Hindexed DDT with random displacements

33% faster
WHAT’S THE CATCH?

- Emitting and compiling IR is expensive!
- Commit **should** tune the DDT, but we do not know how often it will be used – how much tuning is ok?
- Lets see how often we need to reuse the datatypes in a real application!
0-1 column is empty.
We don’t make anything slower than Cray MPI
**Performance Study: MILC**

Most datatypes become seven times faster!
Some even 38 times faster

- Packing faster, but commit is now slower
- How often do we need to use a DDT to break even?
Most datatypes have to be reused 180-5000 times.
But some need 30000 uses to amortize their costs at commit time.
Performance Hints for DDTs

- How often will the DDT be reused?
- How will it be used (Send/Recv/Pack/Unpack)?
- Will the buffer argument be always the same?
- Will the data to pack be in cache or not?
CAN WE BEAT MANUAL PACKING?

<table>
<thead>
<tr>
<th>LAMMPS atomic</th>
<th>MILC su3 zd</th>
<th>NAS LU y</th>
<th>SPECFEM3D oc</th>
<th>WRF x vec</th>
<th>WRF y vec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Packing [% of Packing + Comm.]

Datasize [Byte]

Pack Method
- Cray MPI
- Manual Packing in C
- Our Implementation
**FUTURE WORK**

- Currently we do not support pipelining of packing and communicating
- Our packing library is not yet integrated with an MPI implementation – we use the MPI Profiling interface to hijack calls

http://spcl.inf.ethz.ch/Research/Parallel_Programming/MPI_Datatypes/libpack
THANK YOU!

- Questions?