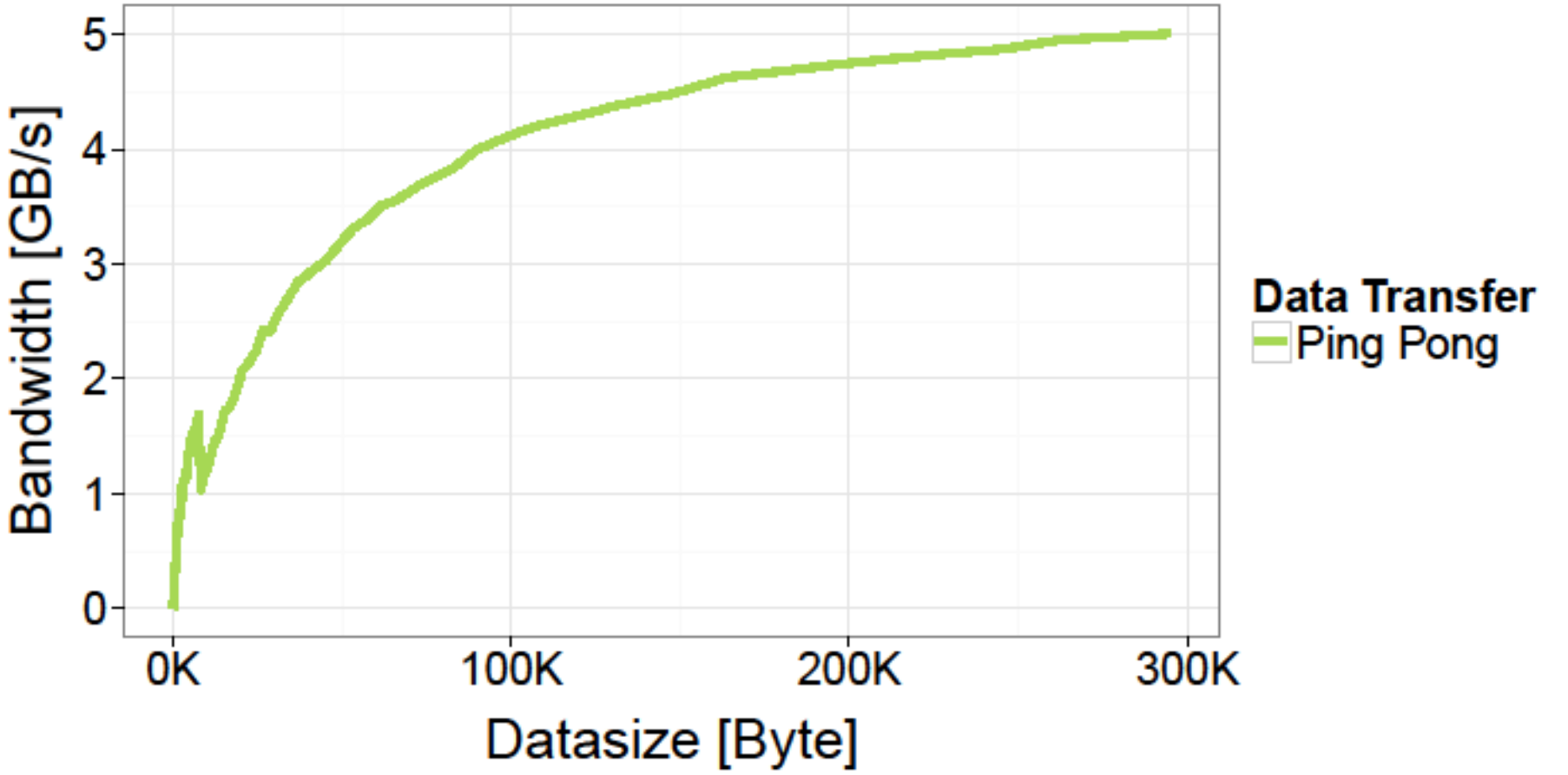
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

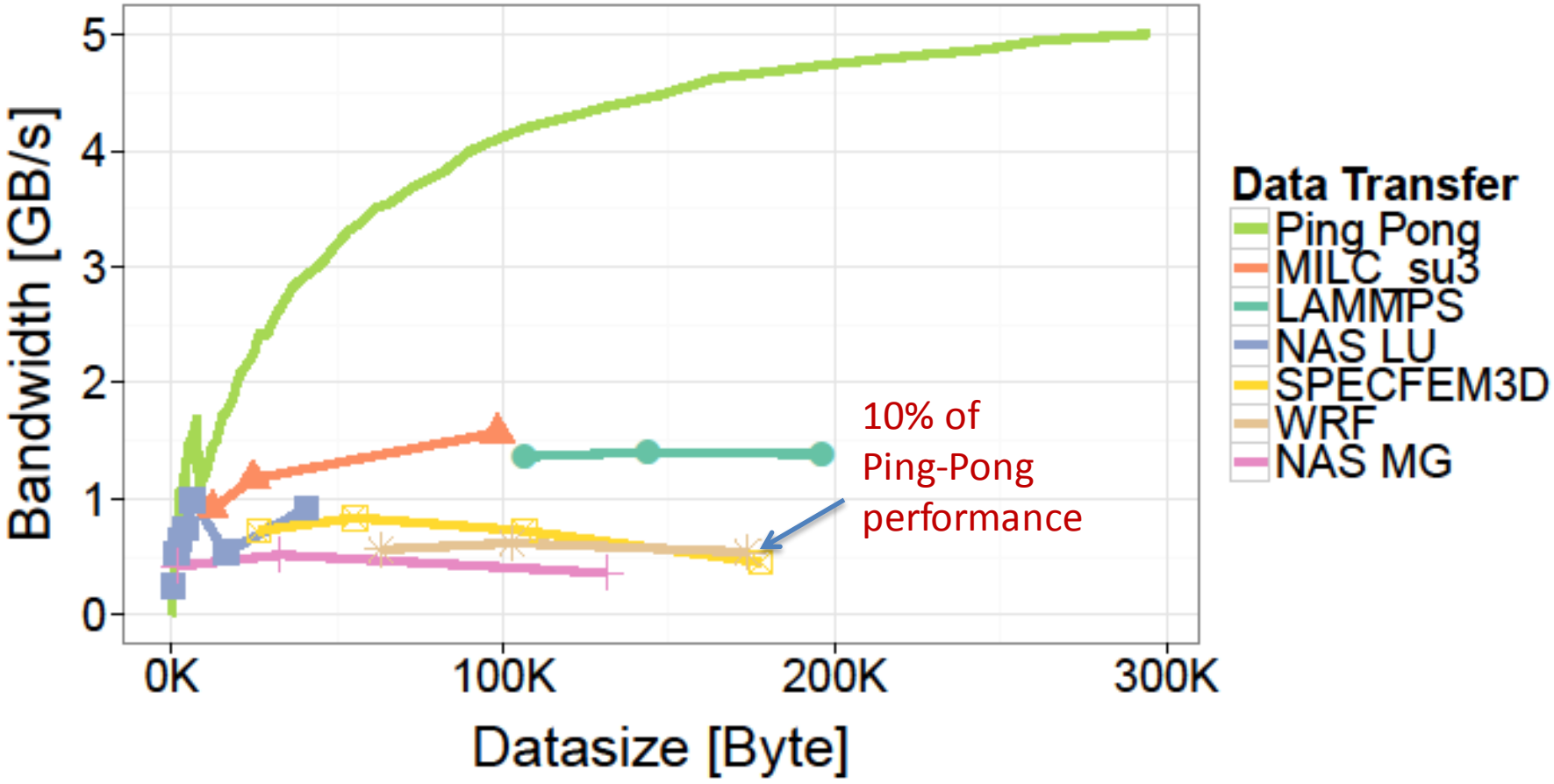
MPI Datatype Processing using Runtime Compilation

Timo Schneider, Fredrik Kjolstad, Torsten Hoefler

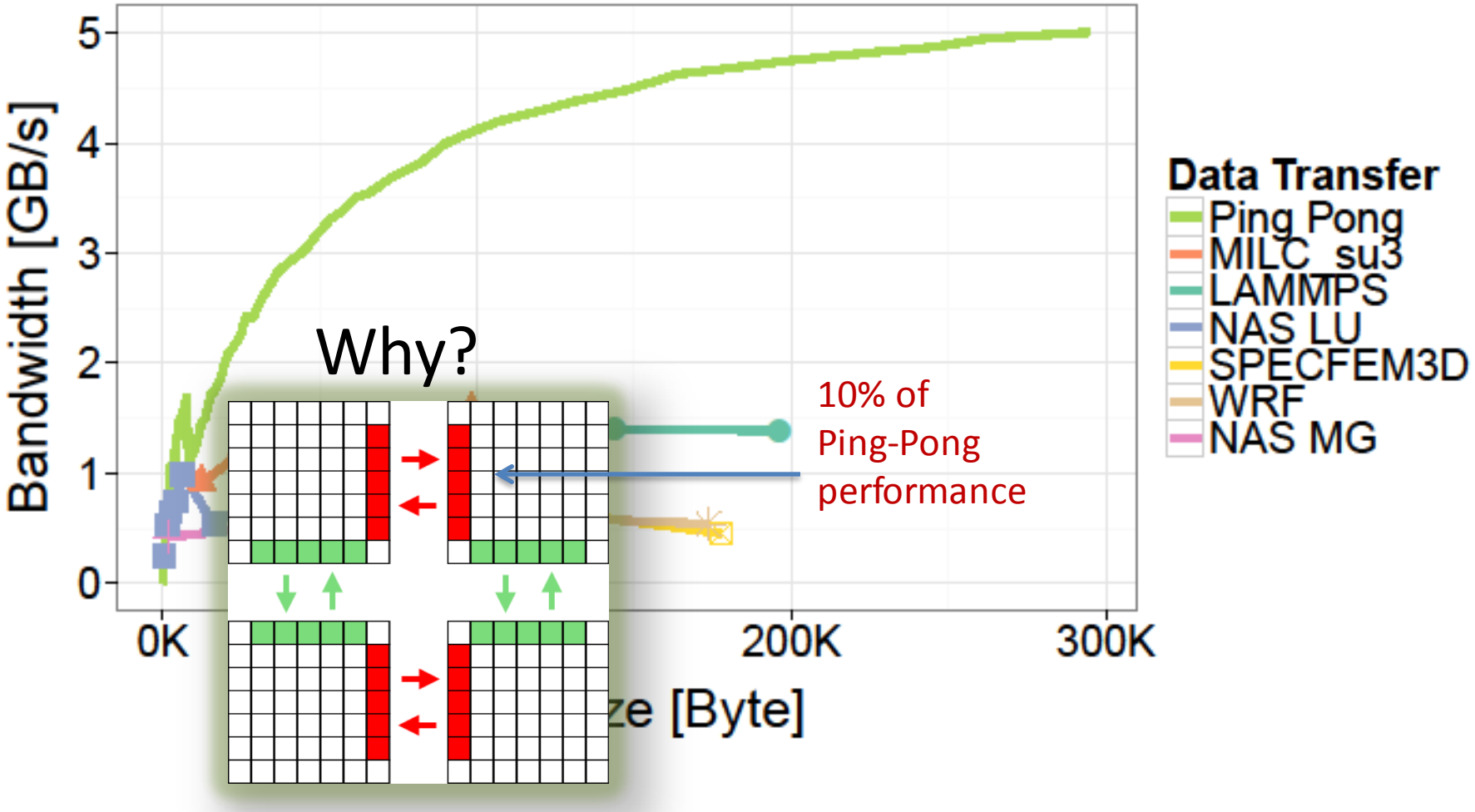
WHAT YOUR VENDOR SOLD



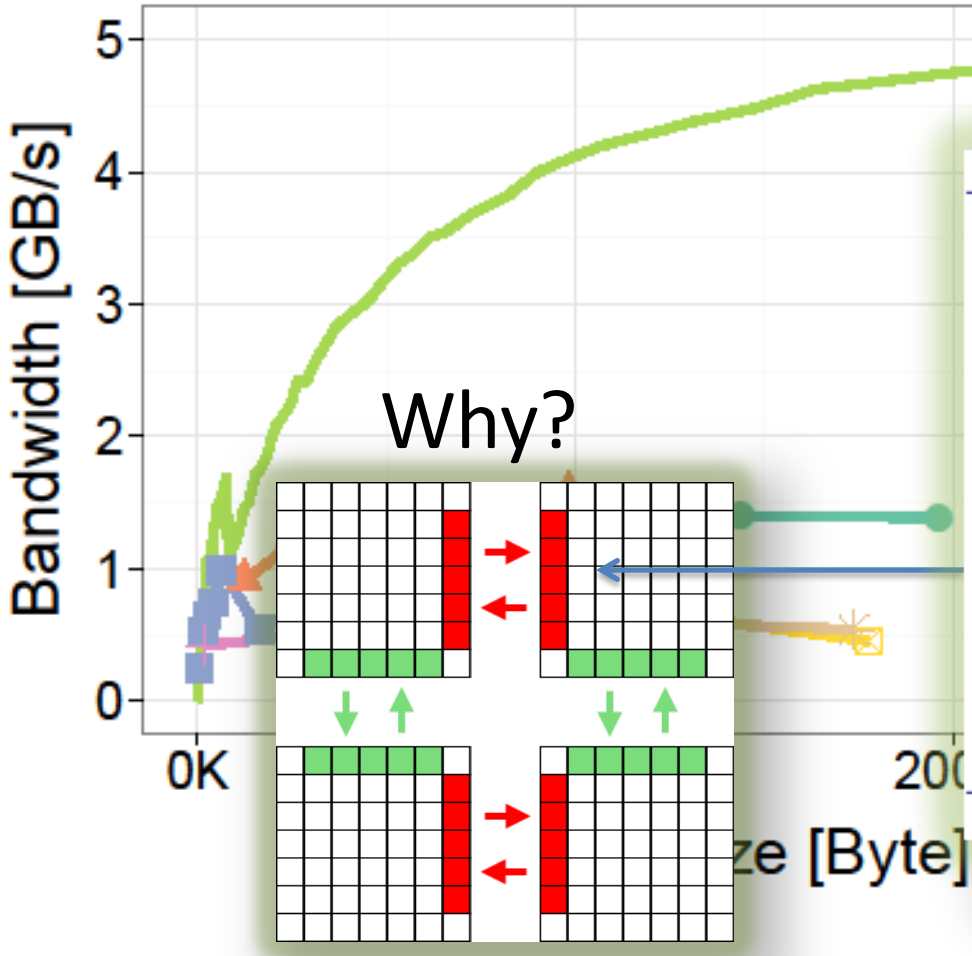
What your Applications get



What your Applications get

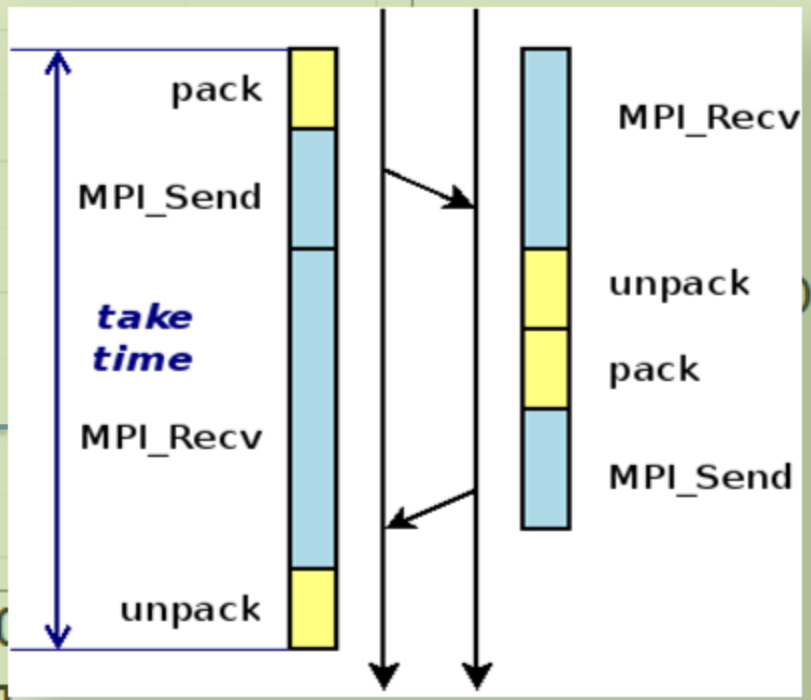


What your Applications get



Why?

How to measure?



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])

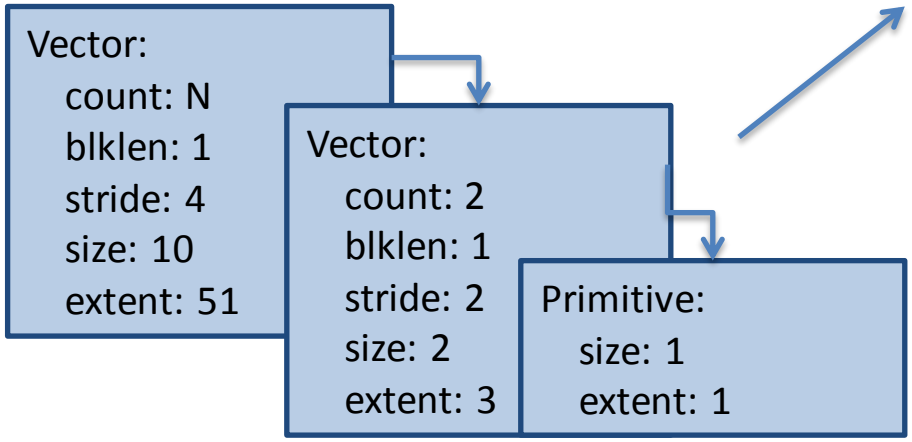
[1] Schneider, Gerstenberger, Hoefler:
Micro-Applications for Communication Data Access Patterns and MPI Datatypes

Interpretation vs. Compilation

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

```
bt = 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

```
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

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

Interpretation vs. Compilation

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

```
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

Interpretation vs. Compilation

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

```
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

Interpretation vs. Compilation

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

```
for (int i=0; i<N; ++i) {
   outbuf[0] = inbuf[0]
   outbuf[1] = inbuf[2]
   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

```
bound = outbuf + 2*N
while (outbuf<bound) {
   outbuf[0] = inbuf[0]
   outbuf[1] = inbuf[2]
   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

```
bound = outbuf + 2*N
while (outbuf<bound) {
   outbuf[0] = inbuf[0]
   outbuf[1] = inbuf[2]
   inbuf += 12
   outbuf  += 2
}
```

- Loop unrolling
- Constant Propagation
- Strength reduction
- Unrolling of outer loop

Interpretation vs. Compilation

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

```
bound = outbuf + 2*N
while (outbuf<bound) {
   outbuf[0] = inbuf[0]
   outbuf[1] = inbuf[2]
   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.

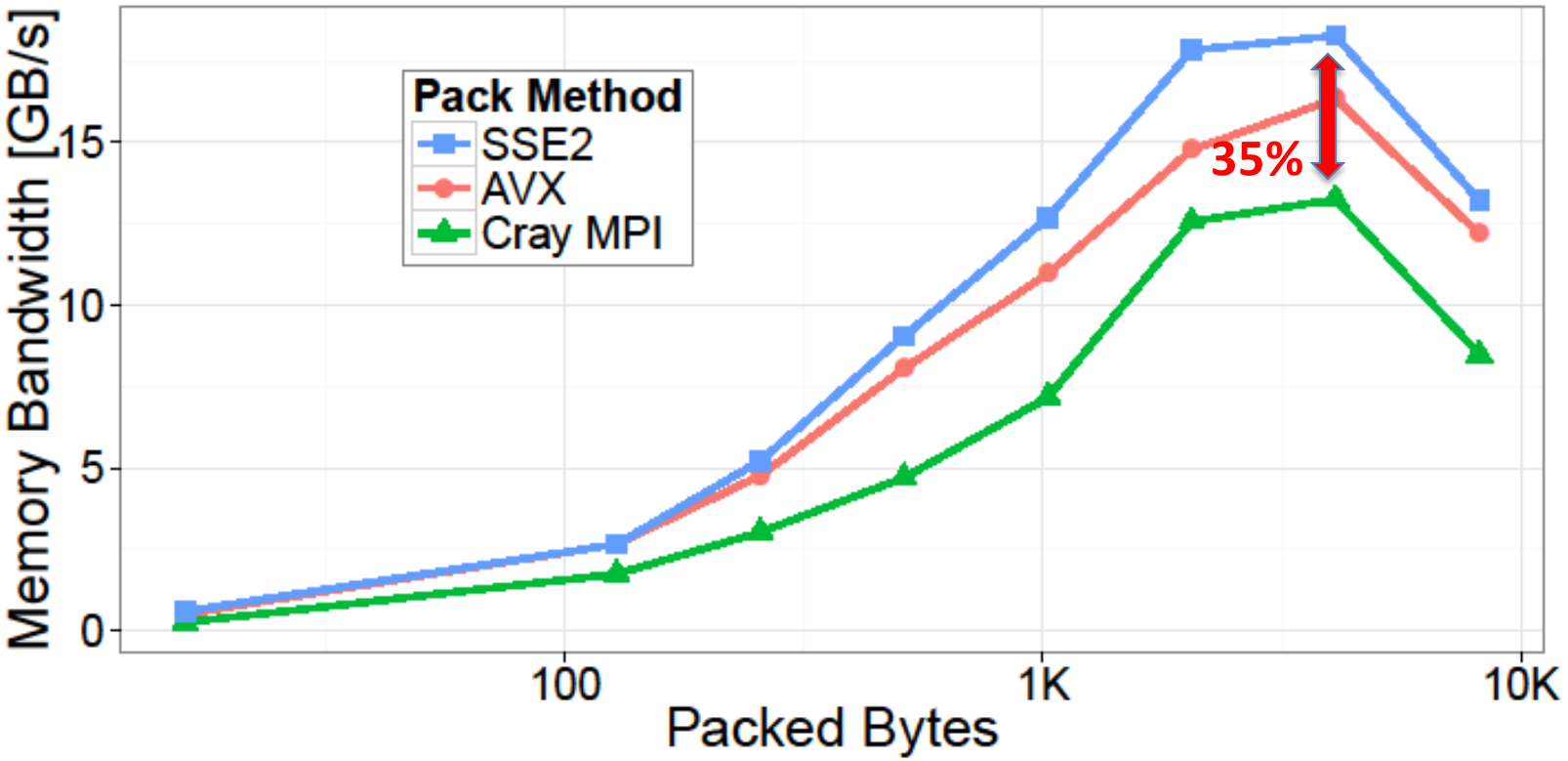
MPI_Send(cnt, buf, new_ddt,…)

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

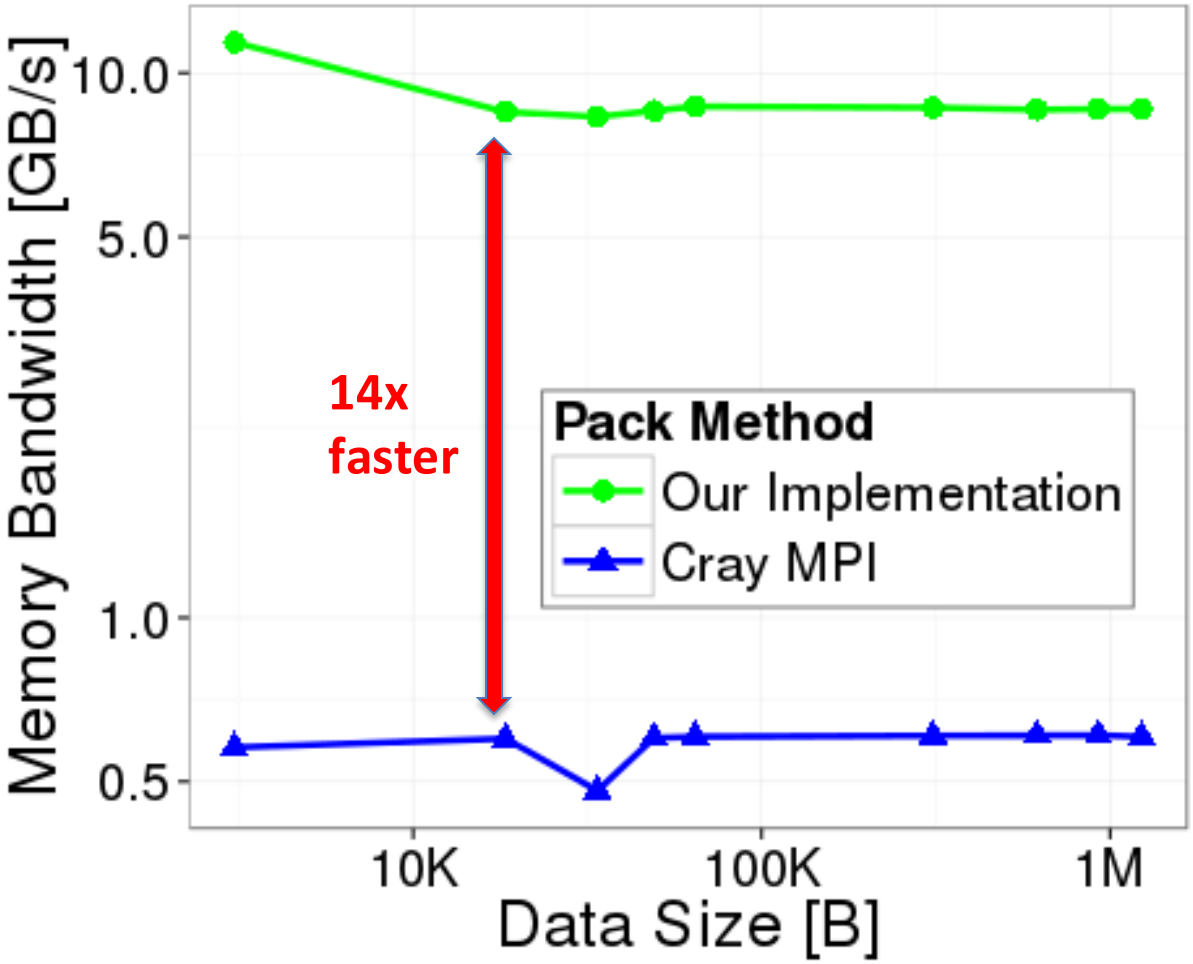


In-cache measurement on AMD Interlagos CPU (Blue Waters test system)

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

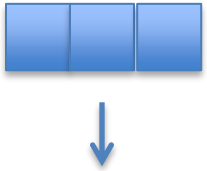
This datatype is used by the Quantum-Chromodynamics code MILC [2]

[2] Studying quarks and gluons on MIMD parallel computers, Bernard, et al.

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

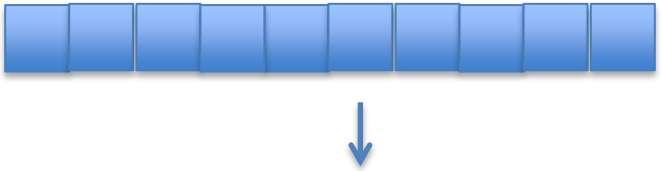
Irregular Datatypes

Depending on index list length:



```
copy(inb+off[0],  outb+…, len[0])
copy(inb+off[1],  outb+…, len[1])
copy(inb+off[2],  outb+…, len[2])
```

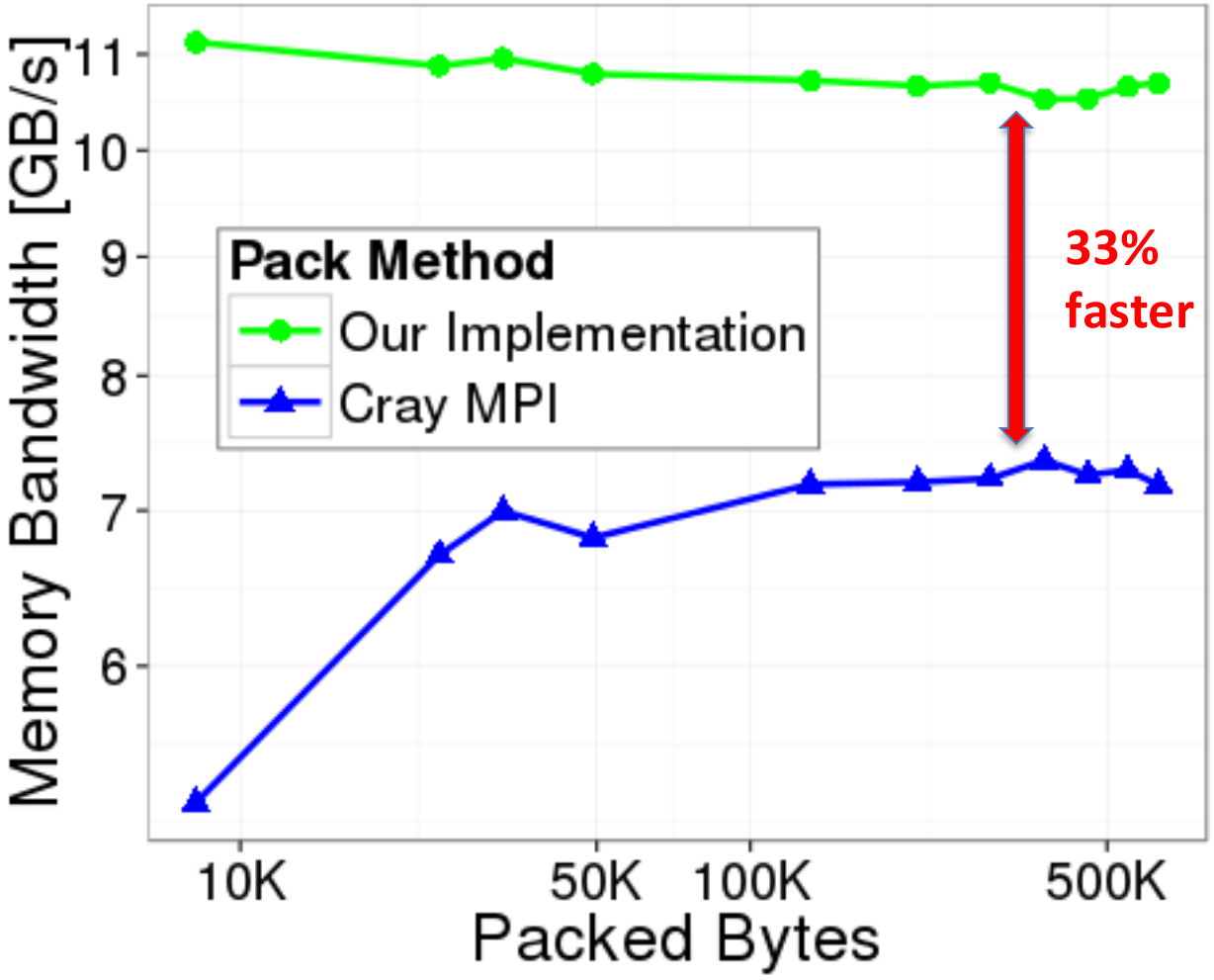
Inline indexes into code



```
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

IRREGULAR PACKING PERFORMANCE

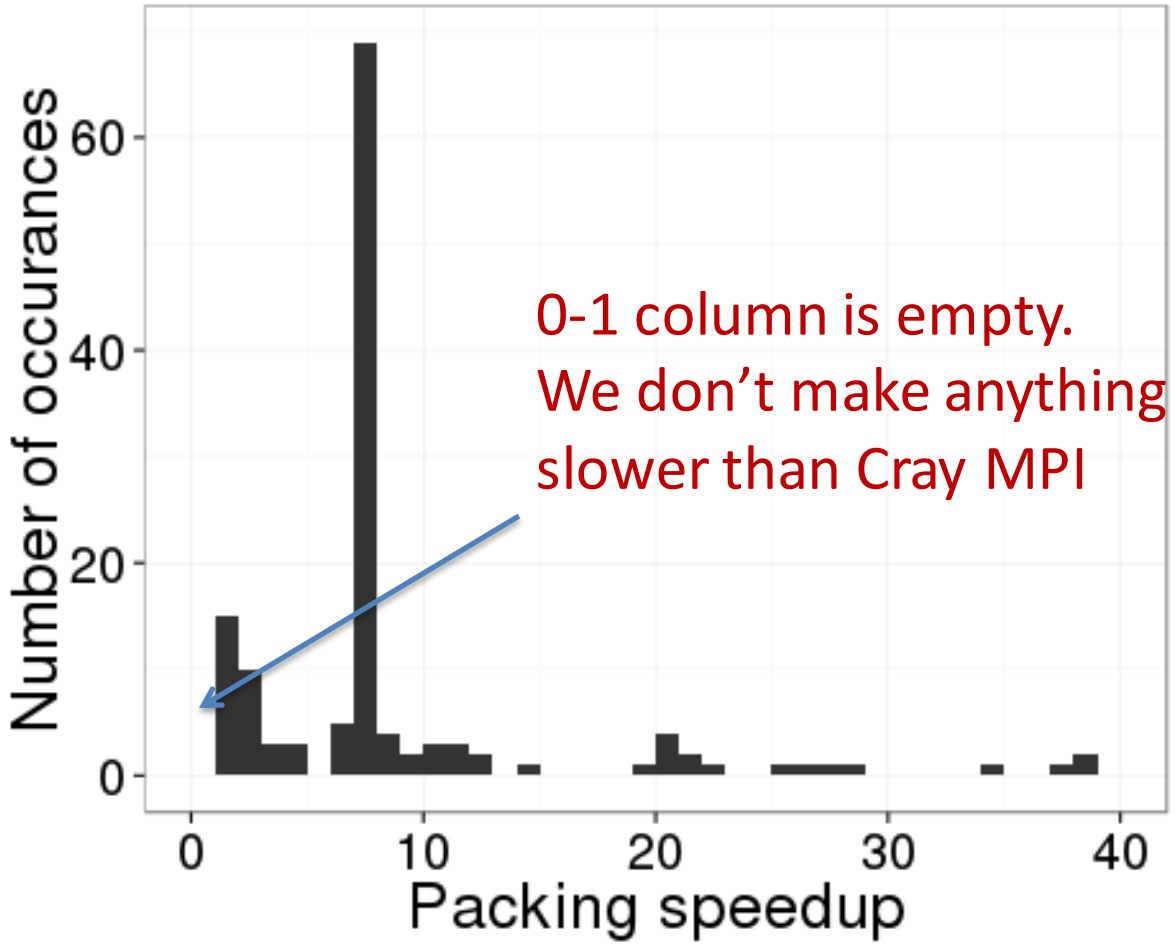


Hindexed DDT with random displacements

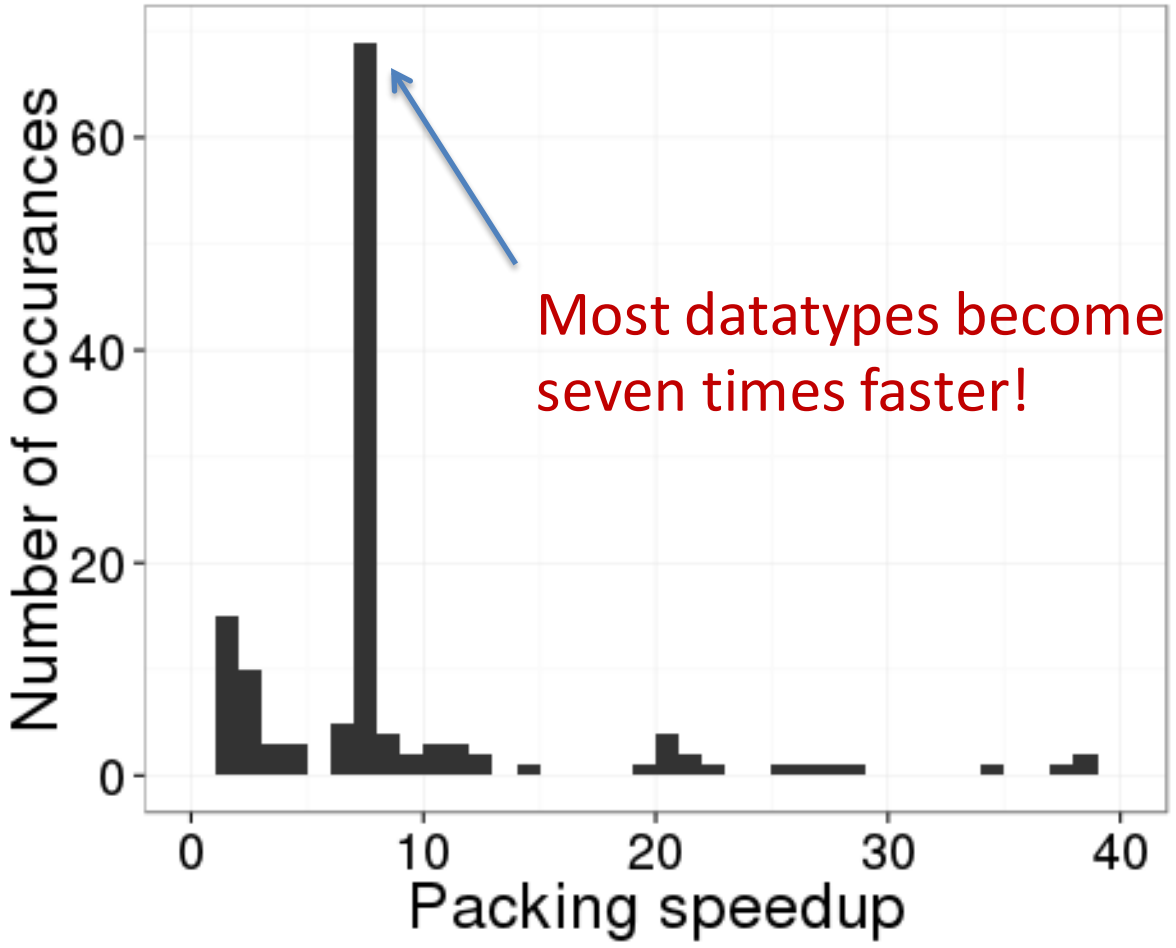
What's the catch?

- 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!

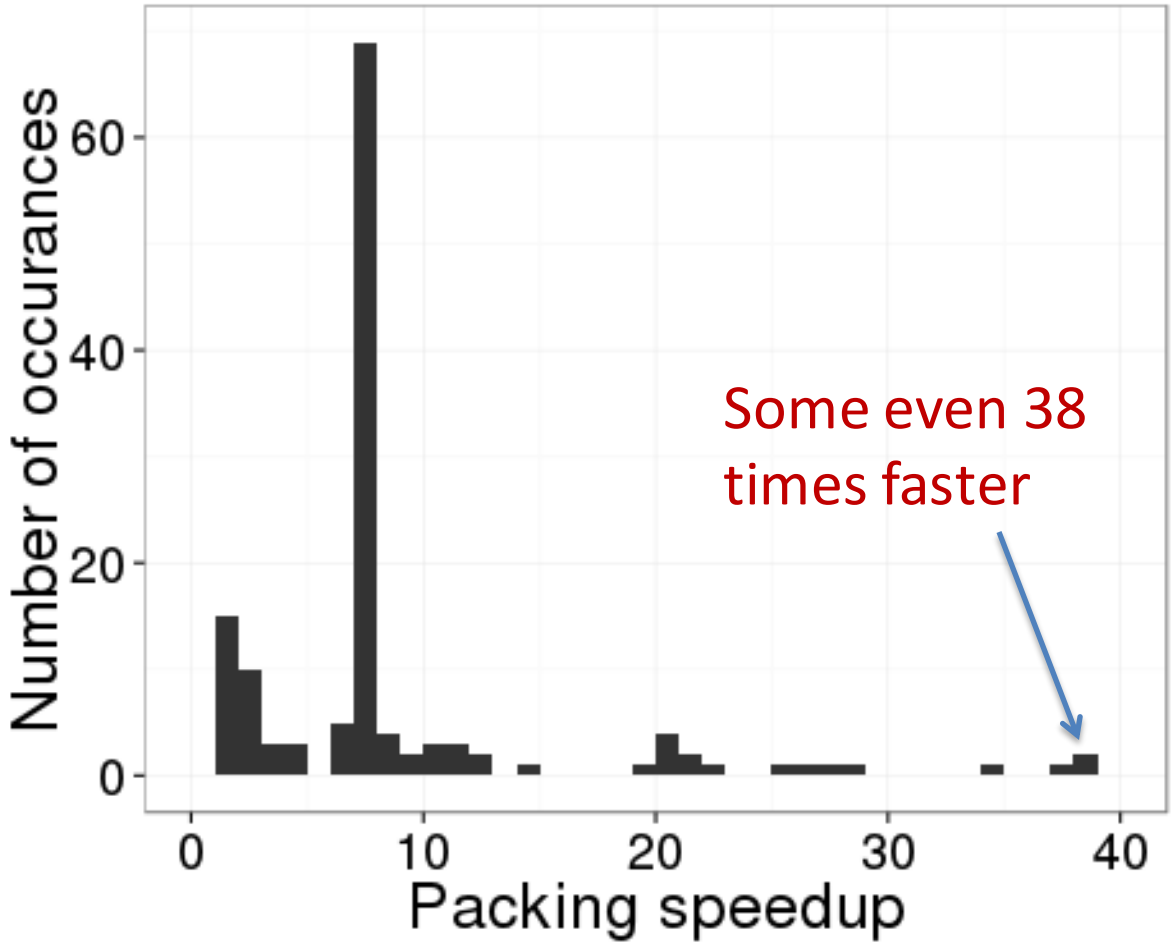
PERFORMANCE STUDY: MILC



PERFORMANCE STUDY: MILC

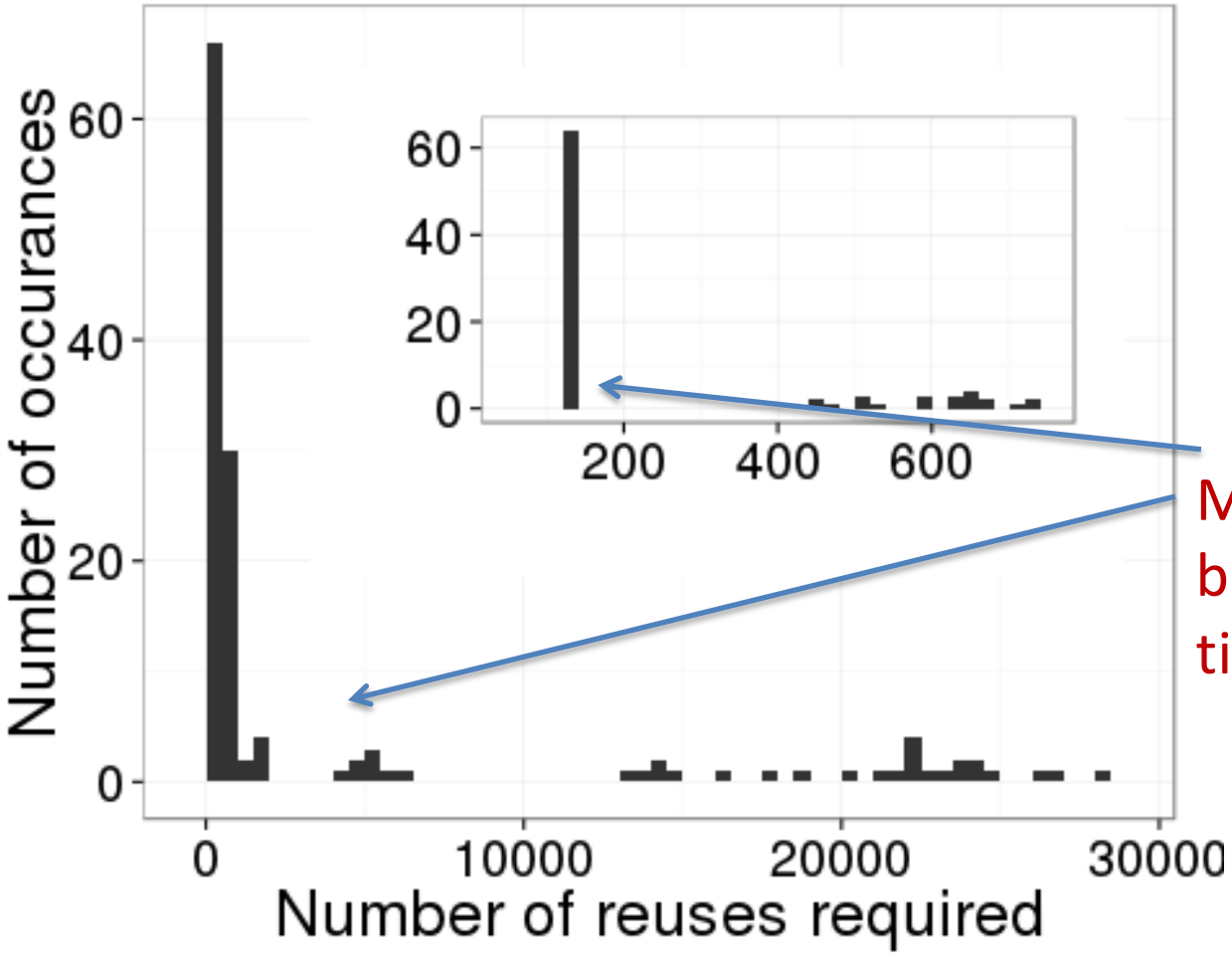


PERFORMANCE STUDY: MILC



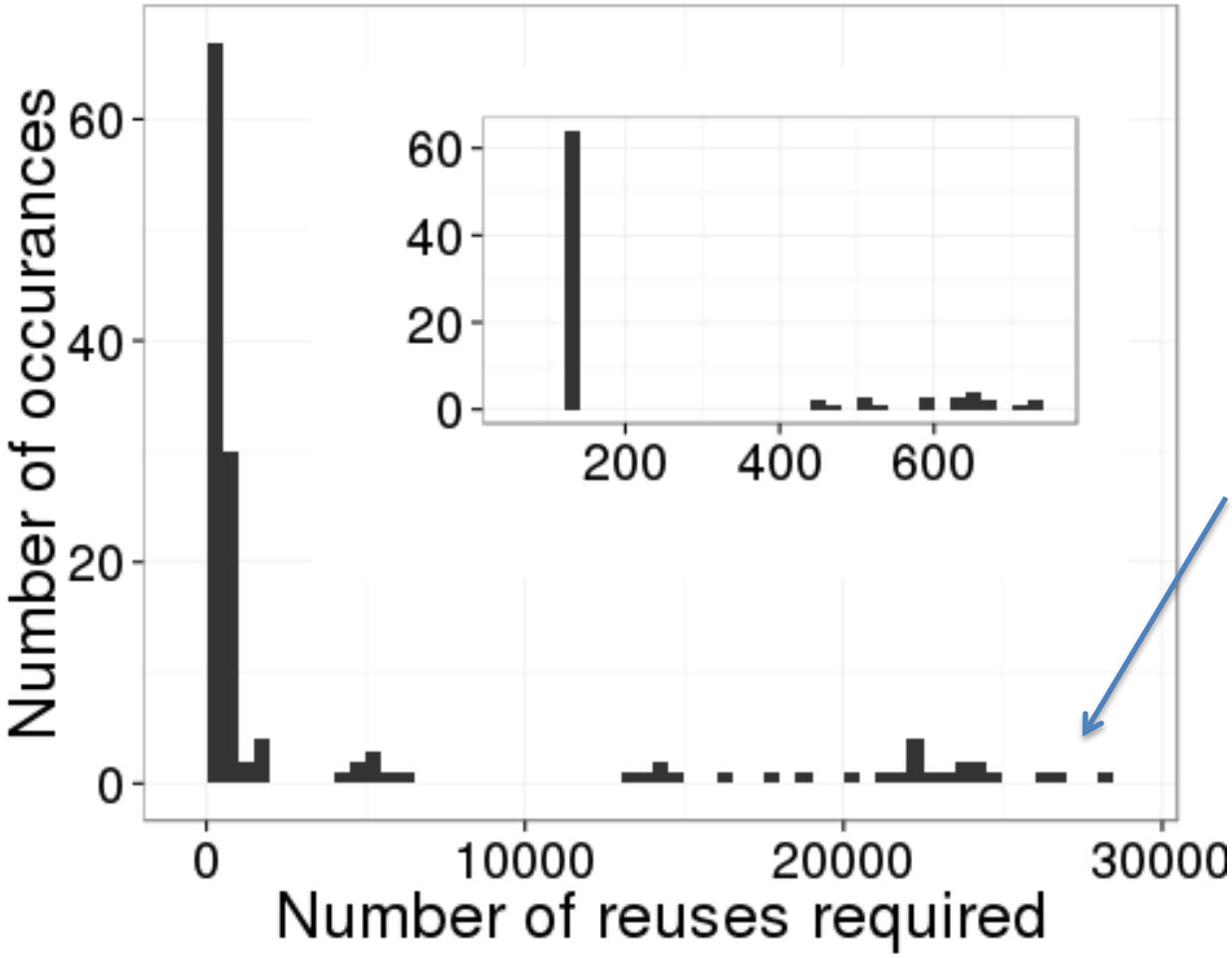
- Packing faster, but commit is now slower
- How often do we need to use a DDT to break even?

PERFORMANCE STUDY: MILC



Most datatypes have to be reused 180-5000 times

PERFORMANCE STUDY: MILC

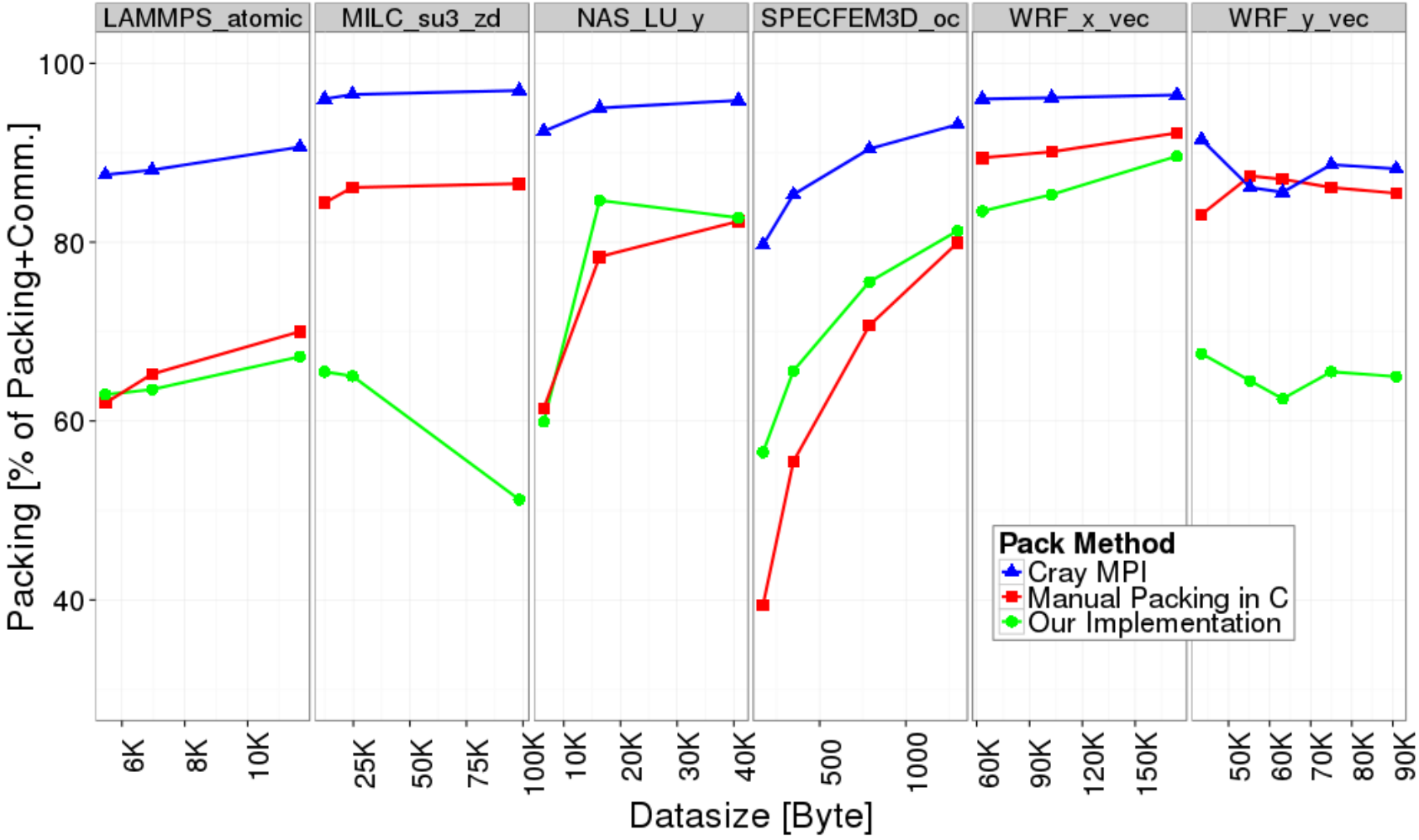


But some need 30000 uses to amortize their costs at commit time

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?



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?