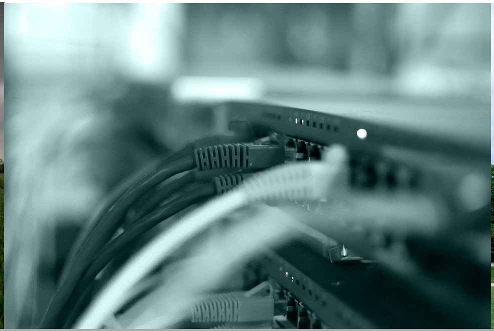
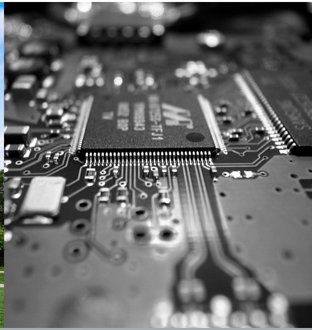
Adding Fast GPU Derived Datatype Handing to Existing MPIs

Carl Pearson (Ph.D candidate, Illinois ECE)
Advisor: Wen-Mei Hwu (Nvidia Research, Illinois ECE)
Kun Wu (Ph.D candidate, Illinois ECE)
I-Hsin Chung, Jinjun Xiong (IBM T. J. Watson Research)



ILLINOIS
Electrical & Computer Engineering
GRAINGER COLLEGE OF ENGINEERING

Carl Pearson

- Compilers & program understanding
- GPU performance programming
  - research, teaching, applications on Blue Waters
- GPU Communication
  - detailed measurement
  - multi-GPU
  - + MPI
- (next...) Scalable Algorithms at Sandia National Lab

cwpearson
lastname at illinois dot edu
carlpearson.net

For ~~the Folks at Home~~ Everyone

- go.illinois.edu/TEMPI
  - paper preprint (PDF)
  - link to code on github
  - these slides (PDF)
- Some diagrams will be 2D instead of 3D
  - Fewer lines and arrows
  - concepts generalize to higher dimensions

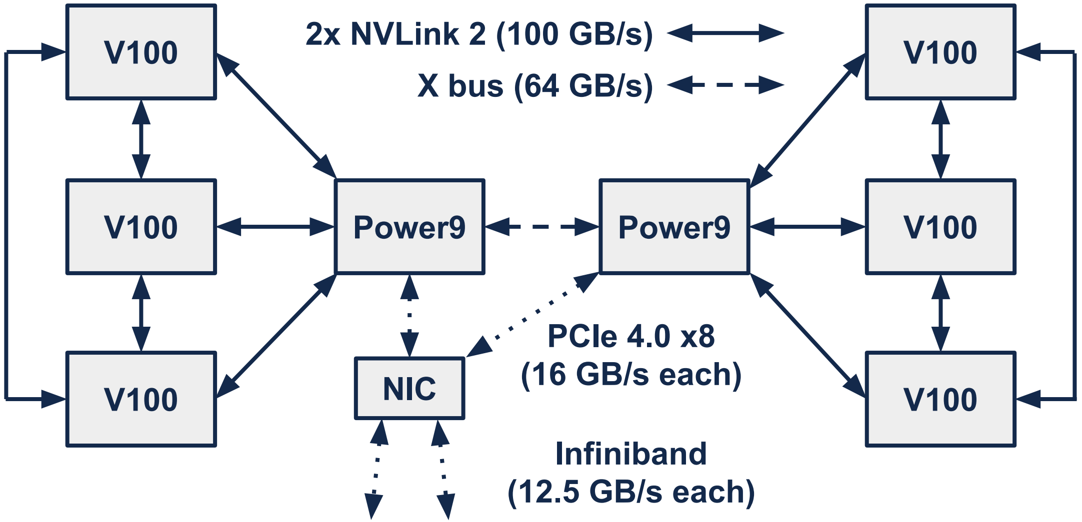
If that URL does not work... https://www.carlpearson.net/talk/20210219_unm_psaap/

Outline

- 3D Distributed Stencil on GPU
- A case study
- TEMPI's approach to derived type handling
  - Translation
  - Canonicalization
  - Kernel Selection
- Some Performance Results
- How TEMPI works with MPI

OLCF Summit

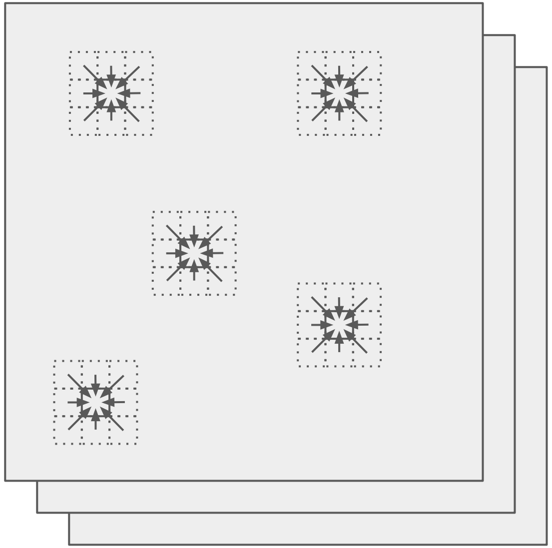
- Similar to LLNL Lassen, 6 GPUs / node instead of 4



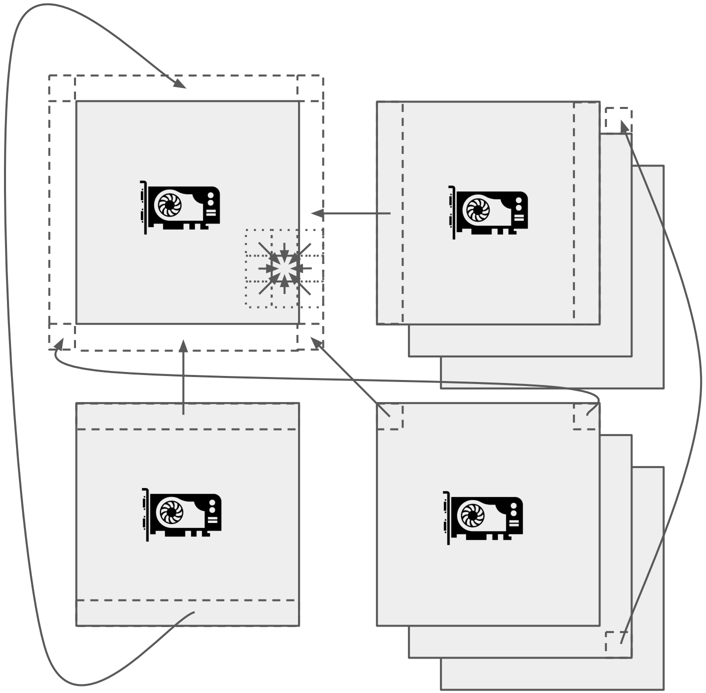
Summit Node
(bidirectional bandwidth)

Distributed Stencil

- periodic boundary conditions
- one sub-grid per GPU



grid distributed to different memories



quantities * # directions independent messages per rank

Astaroth[1] - Stencil on GPU

- 256^3 grid-points per GPU
  - 8 quantities per grid-point
  - double-precision (8 bytes / quantity)
- Kernel radius = 3, periodic boundary conditions
- CUDA 11.0.221, Spectrum MPI 10.3.1.2

<u>Pros</u>

- Regular data access and reuse
  - “easy” to avoid memory bandwidth bottleneck
- Regular computation
  - “easy” to vectorize

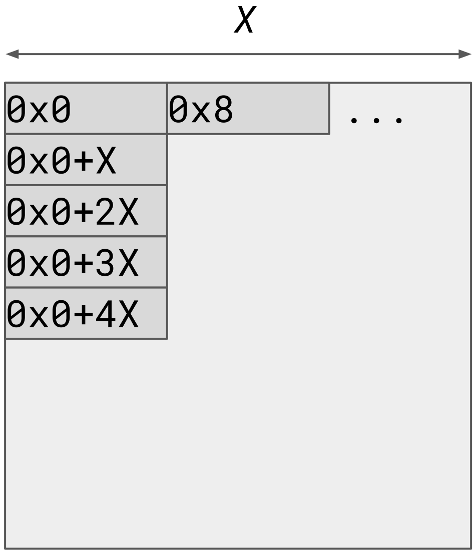
<u>Cons</u>

- Limited GPU memory
  - communication
- High latency of GPU control
- CUDA? OpenCL? Kokkos? etc.

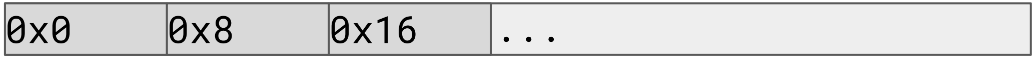
[1] Pekkilä, J., Väisälä, M. S., Käpylä, M. J., Käpylä, P. J., & Anjum, O. (2017). Methods for compressible fluid simulation on GPUs using high-order finite differences. *Computer Physics Communications*, *217*, 11-22.

Contiguous & Non-contiguous Data

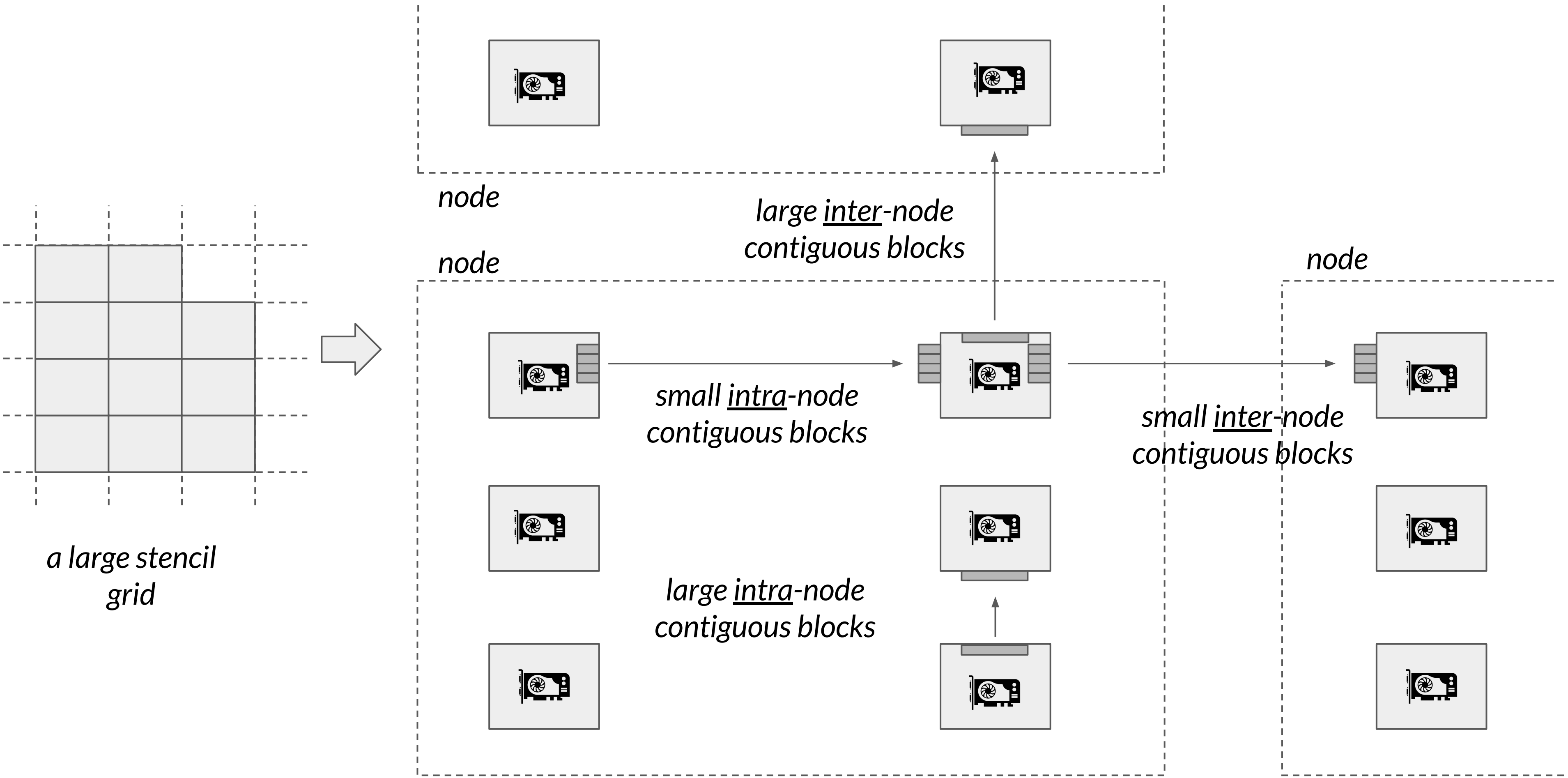
- (Generalizes to 3D)
- “row-major”



“top edge” - contiguous

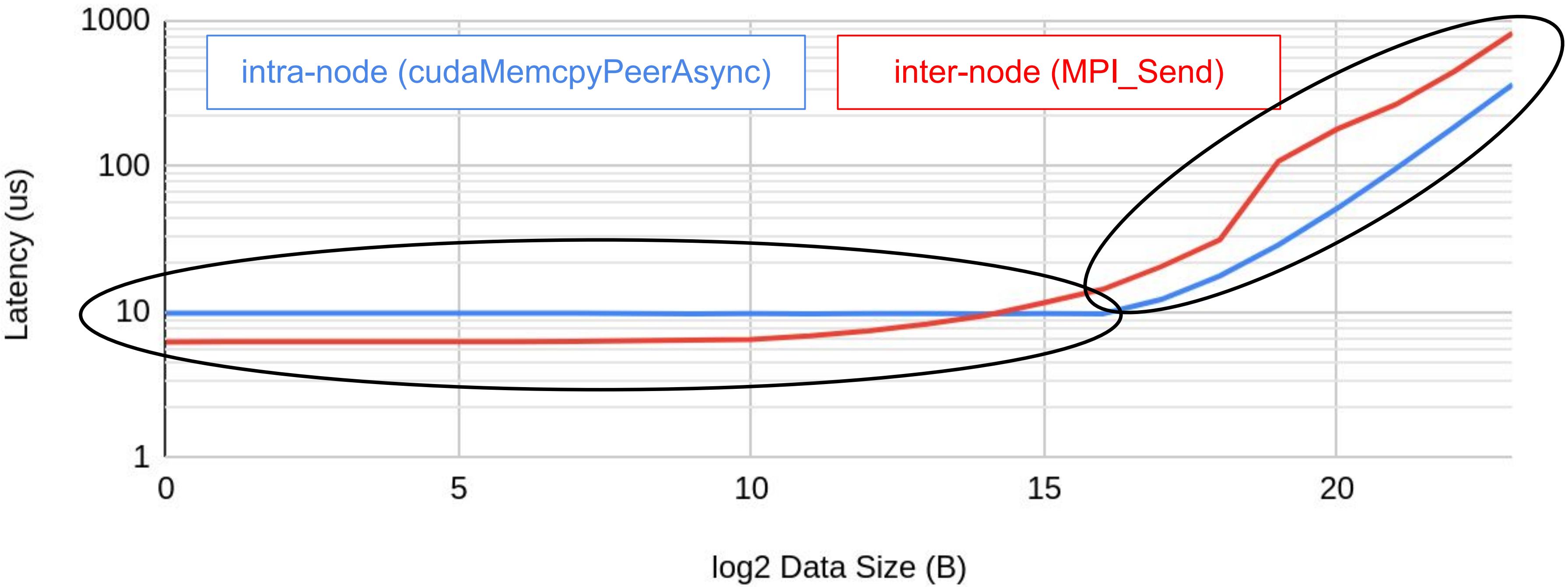


“left edge” - non-contiguous



a large stencil grid
node
node
large inter-node contiguous blocks
node
small intra-node contiguous blocks
small inter-node contiguous blocks
large intra-node contiguous blocks

Latency vs Contiguous Size



small blocks - relatively slow
(latency)

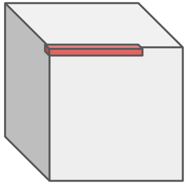
large blocks - relatively fast
(bandwidth)

Regular MPI Derived Datatypes

- MPI_Type_contiguous(*count*, *old*, *new*)
  - *new* is *count* contiguous copies of *old*
- MPI_Type_vector(count, blocklength, stride, old, new)
  - *new* is *count* blocks of *blocklength olds*, with pitch of *stride olds*
- MPI_Type_hvector(count, blocklength, stride, old, new)
  - *new* is *count* blocks of *blocklength olds*, with pitch of *stride* <u>bytes</u>
- MPI_Type_subarray(count, array_of_sizes, array_of_subsizes, array_of_starts, order, old, new)
  - *new* is a subarray of *array_of_subsizes* at offset *array_of_starts* taken from an array with size *array_of_sizes* of *olds*
- Call MPI_Type_commit(type) on a type before it can be used

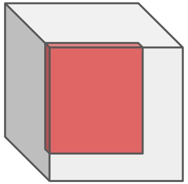
MPI Derived Datatypes

MPI_Type_Contiguous(count, oldtype, newtype)
MPI_Type_Contiguous(E_0, MPI_BYTE, &row)

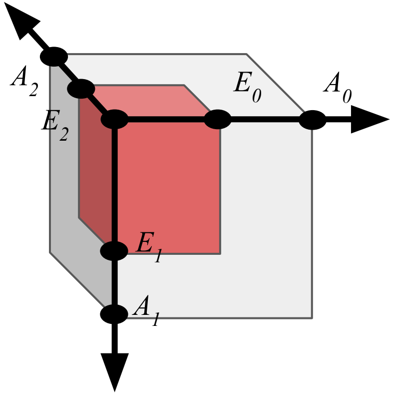


(bytes)

MPI_Type_hvector(count, blocklength, stride, oldtype, newtype)
MPI_Type_hvector(E_1, 1, A_0, row, &plane)

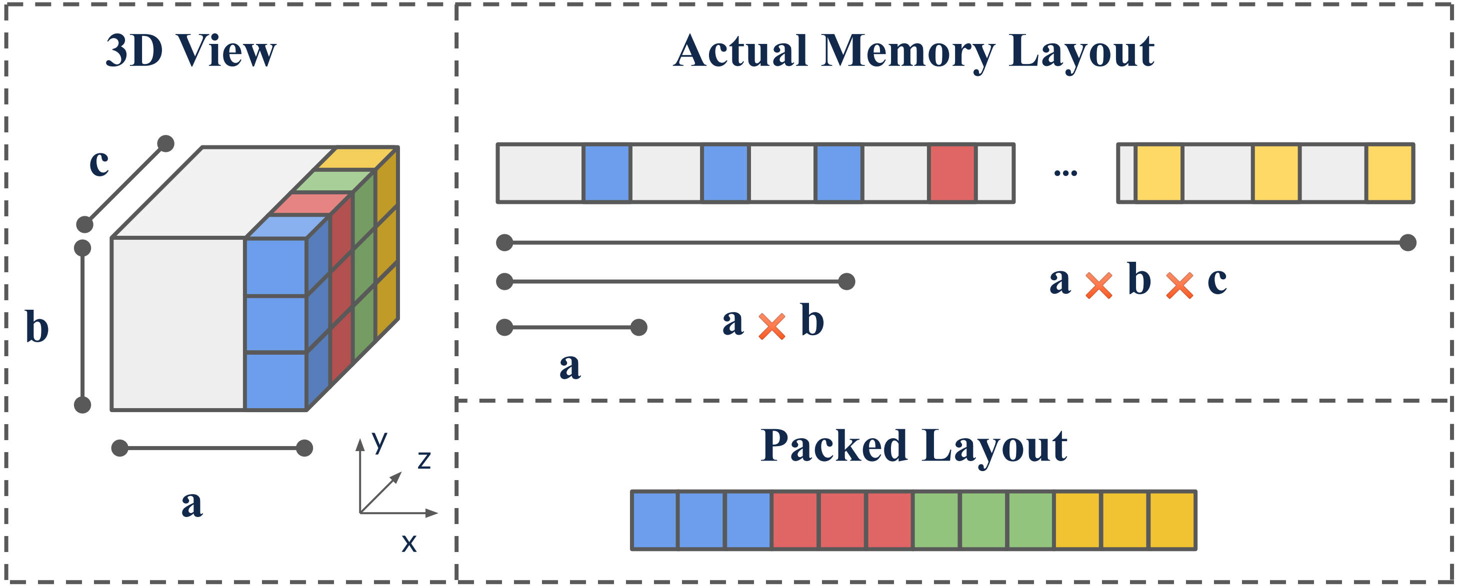


MPI_Type_hvector(E_2, 1, A_1, plane, &cuboid)

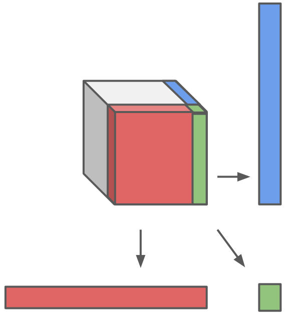


Other applicable types can mix and match too (vector, subarray)

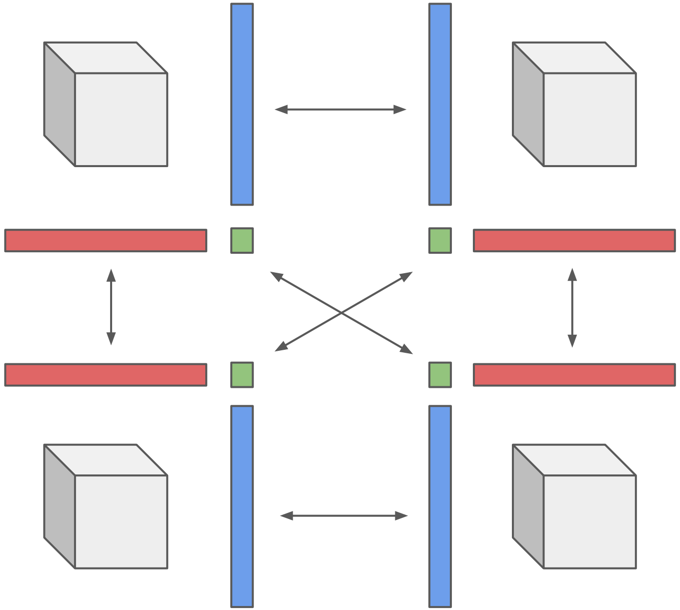
Many Small Blocks into Few Large Blocks



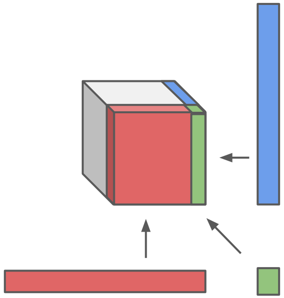
Pack / Alltoallv / Unpack



MPI_Packs



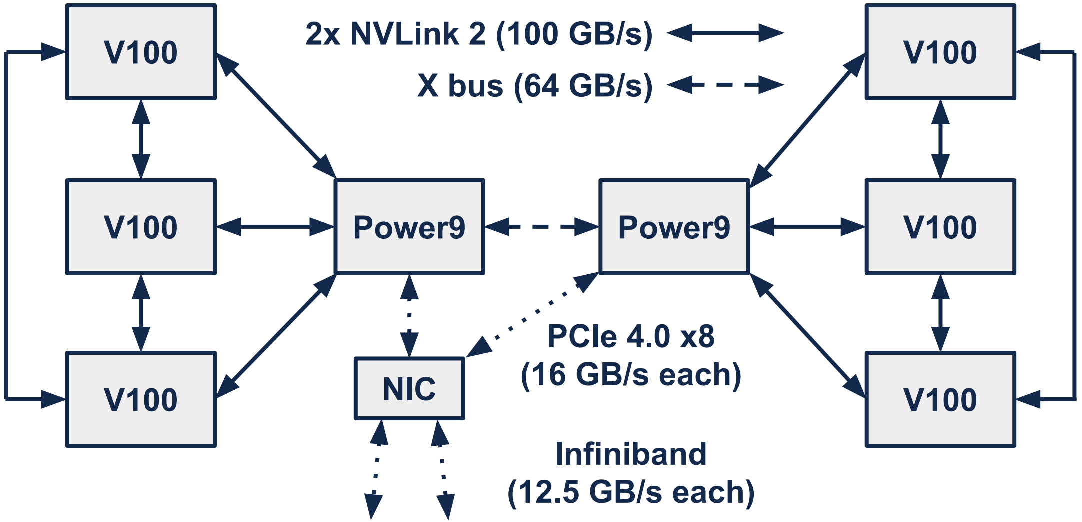
MPI_Neighbor_alltoallv



MPI_Unpacks

OLCF Summit

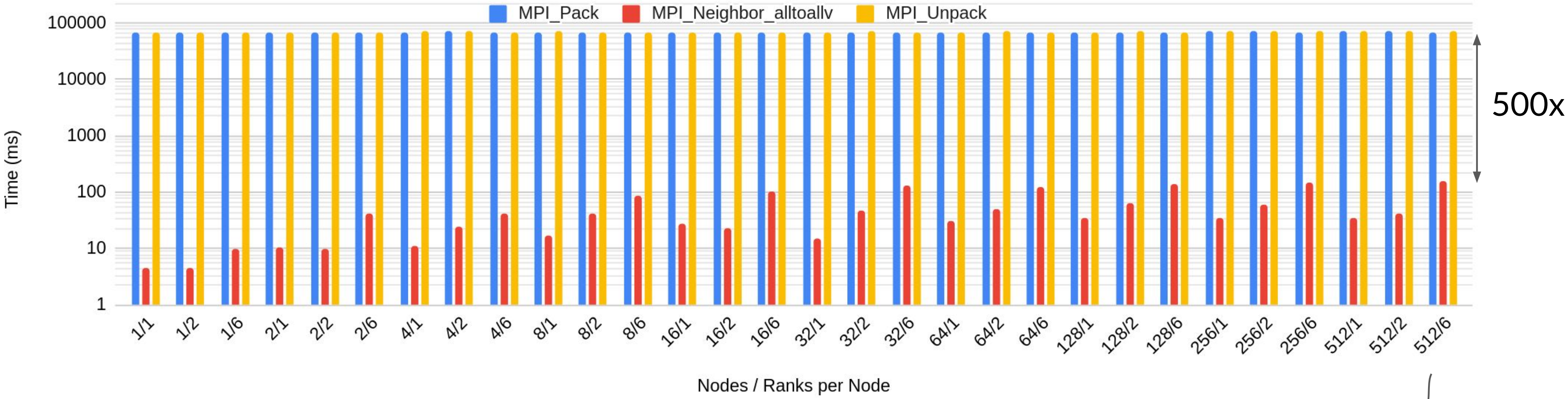
- Similar to LLNL Lassen, 6 GPUs / node instead of 4



Summit Node
(bidirectional bandwidth)

The Problem

- Halo exchange with MPI derived types

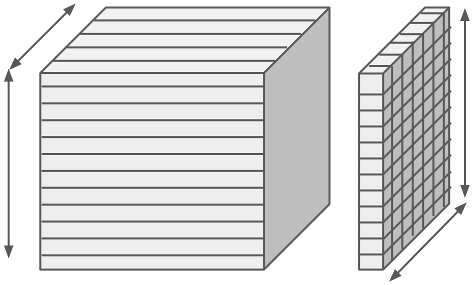


- 73.7 MiB/rank
  - MPI_Neighbor_alltoallv = ~500 MB/s/rank
  - MPI_Pack / MPI_Unpack = **~1 MB/s**

But 70s? (a.k.a the baseline)

- 256 x 256 per quantity x 8 quantities x 3 substeps x 2 directions
    - most of the “non-contiguousness” is in one dimension
- 3,145,728 contiguous blocks (~20us per block)
- one cudaMemcpyAsync per block

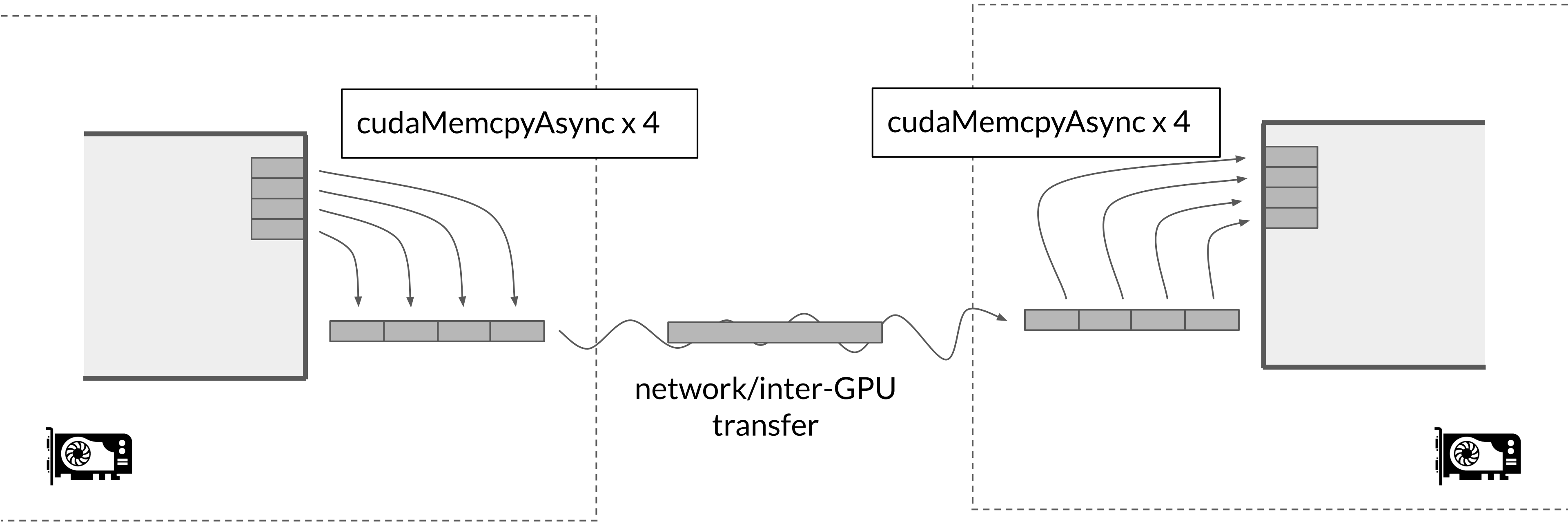
256 contiguous blocks
on each of these faces



256 x 256
contiguous blocks
on this face

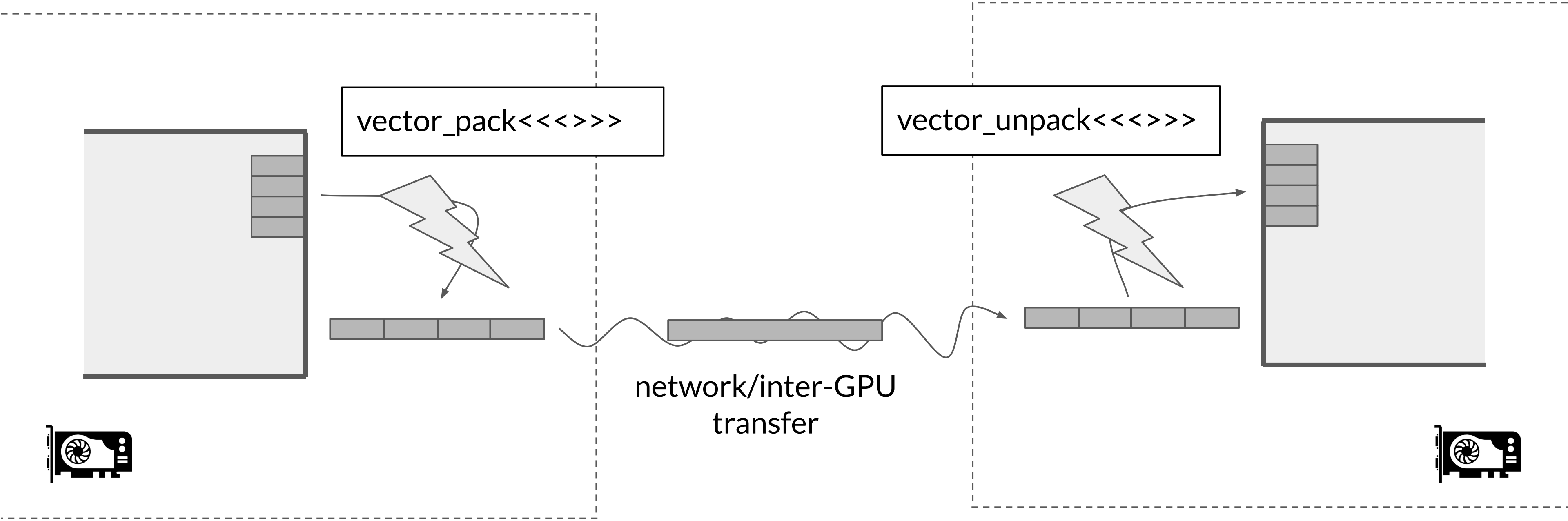
(recall that halo space breaks up otherwise contiguous directions)

MPI_Send (OpenMPI / SpectrumMPI)

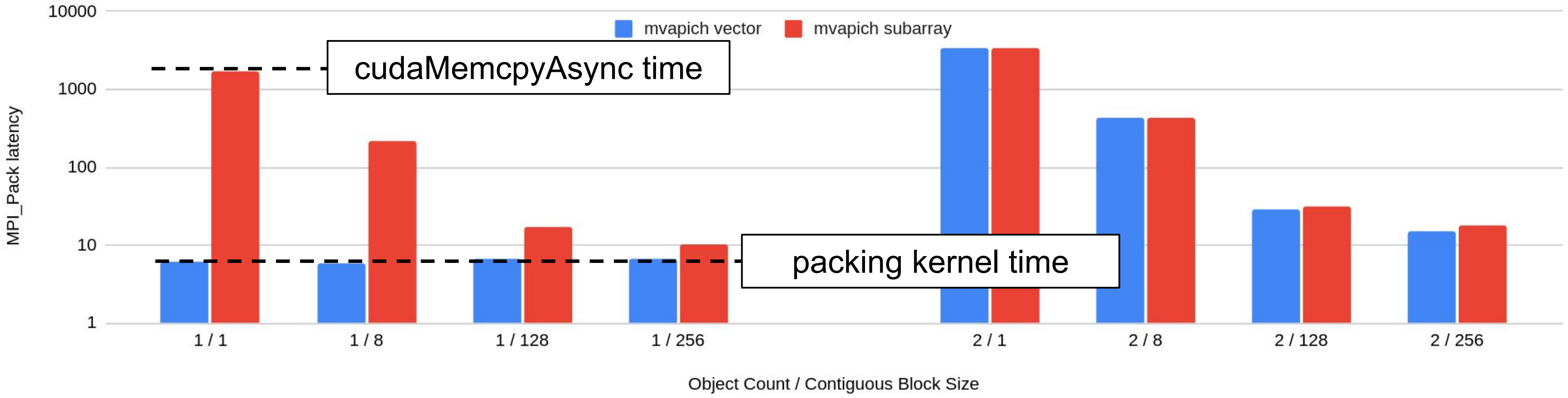


A Partial Solution (MVAPICH)

- GPU Kernels to pack non-contiguous data
- Implemented in MVAPICH (non-GDR)



MVAPICH MPI_Pack (1 KiB)



Works for vector, but not subarray

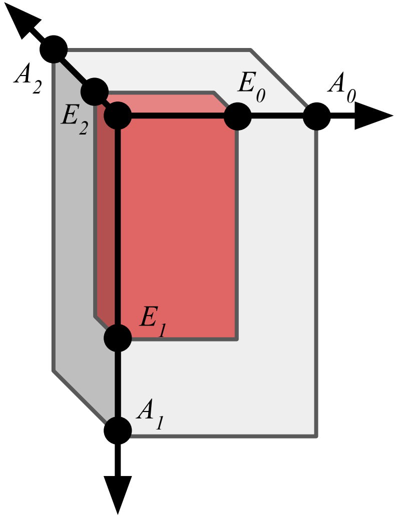
Works for one object, not two

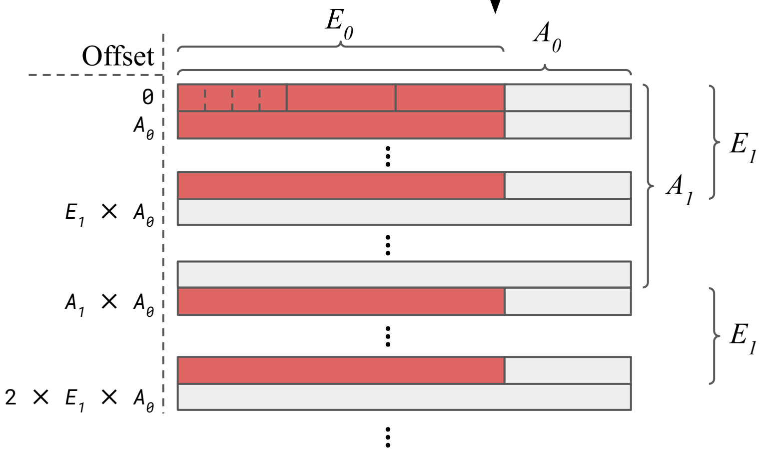
MPI Derived Datatype Equivalence

```
int array_of_sizes[2]{A0, A1};
int array_of_subsizes[2]{E0, E1};
int array_of_starts[2]{0, 0};
MPI_Type_create_subarray(
      2, array_of_sizes, array_of_subsizes,
      array_of_starts, MPI_ORDER_C, MPI_BYTE, &plane);
MPI_Type_vector(E2, 1, 1, plane, &cuboid);
```

```
MPI_Type_vector(E0, 1, 1, MPI_BYTE, &row);
MPI_Type_create_hvector(E1, 1, A0, row, &plane);
MPI_Type_create_hvector(E2, 1, A0 * A1, plane, &cuboid);
```

```
int array_of_sizes[3]{A0, A1, A,};
int array_of_subsizes[3]{E0, E1, E2};
int array_of_starts[3]{0, 0, 0};
MPI_Type_create_subarray(
      3, array_of_sizes, array_of_subsizes,
      array_of_starts, MPI_ORDER_C, MPI_BYTE, &cuboid);
```





Other Work (see paper)

Specialized Kernels

- fragile
  - cartesian product of compound and base types
  - byte vector, float vector, byte subarray, float subarray
  - vector of vector, etc.

Flexible Approaches

- large data representation
  - Each datatype is a list of block offsets and lengths
  - May be as large as the data itself
  - Limits GPU performance
    - split bandwidth between metadata and data

warning: this a very superficial summary of related work

This Work

- Regular types only
  - Compact representation
  - Fast generalized kernels
  - For indexed/struct types, probably some previous approach is better
- No “deep integration” with MPI
  - Shim / translation layer only
  - Leaving some performance on the table
  - Don’t have to touch an MPI implementation

TEMPI Datatype Handling

MPI_Type_commit()

↓

Translation
Convert MPI Derived Datatype into internal representation (IR)

↓

Canonicalization
Convert semantically-equivalent IR to simplified form

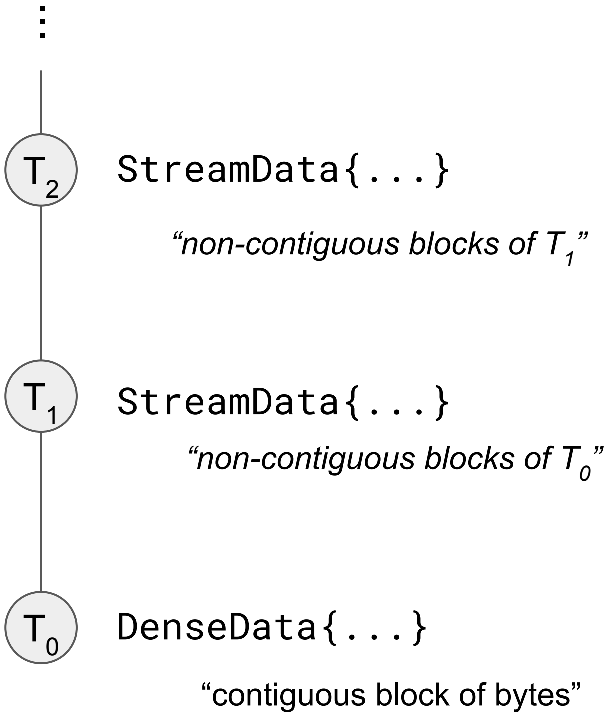
↓

Kernel Selection
Choose packing/unpacking kernel for future operations

IR

```
StreamData {
  integer offset; // offset (B) of the first element
  integer stride; // pitch (B) between element
  integer count;  // number of elements
}

DenseData {
  integer offset; // offset (B) of the first byte
  integer extent; // number of bytes
}
```

Hierarchy of StreamData, rooted at DenseData

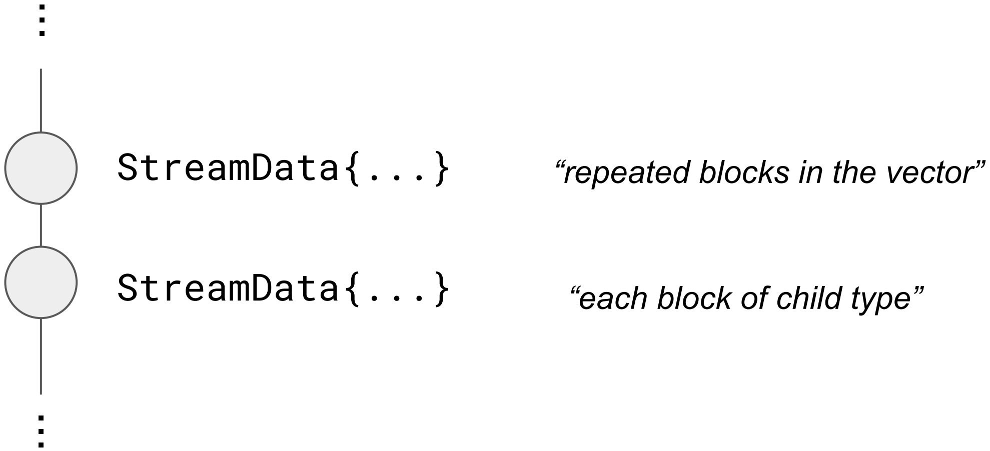


Translation: Named / MPI_Type_contiguous

```
               T0 = MPI_BYTE   DenseData{offset: 0, count: 1}

              T1 = MPI_FLOAT   DenseData{offset: 0, count: 4}

MPI_Type_contiguous(10, T0, &T2)   StreamData{offset: 0, count: 10, stride: 1}
                                   └DenseData{offset: 0, count: 1}

MPI_Type_contiguous(13, T2, &T3)   StreamData{offset: 0, count: 13, stride: 10}
                                   └StreamData{offset: 0, count: 10, stride: 1}
                                    └DenseData{offset: 0, count: 1}
```

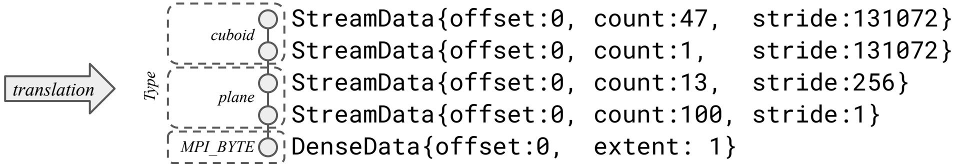
Translation: Vector

```
                     T0 = MPI_BYTE   DenseData{count: 1}

MPI_Type_vector(10, 4, 6, T0, &T1)   StreamData{count: 10, stride: 6}
                                     └StreamData{count: 4, stride: 1}
                                      └DenseData{offset: 0, count: 1}
```

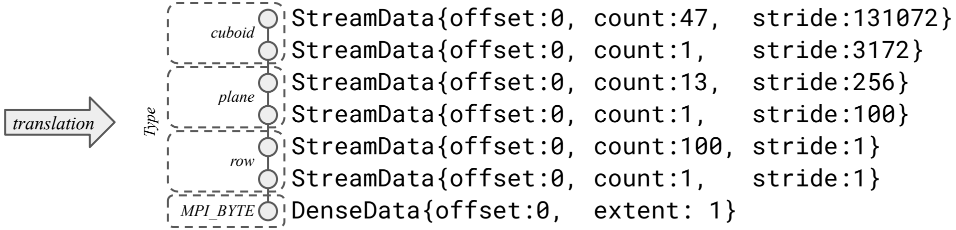


Translation: Three Equivalent Examples

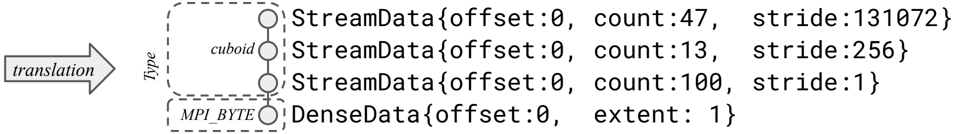
```
int array_of_sizes[2]{256, 512};
int array_of_subsizes[2]{100, 13};
int array_of_starts[2]{0, 0};
MPI_Type_create_subarray(
      2, array_of_sizes, array_of_subsizes,
      array_of_starts, MPI_ORDER_C, MPI_BYTE, &plane);
MPI_Type_vector(47, 1, 1, plane, &cuboid);
```




```
MPI_Type_vector(100, 1, 1, MPI_BYTE, &row);
MPI_Type_create_hvector(13, 1, 256, row, &plane);
MPI_Type_create_hvector(47, 1, 256 * 512, plane, &cuboid);
```




```
int array_of_sizes[3]{256, 512, 1024};
int array_of_subsizes[3]{100, 13, 47};
int array_of_starts[3]{0, 0, 0};
MPI_Type_create_subarray(
      3, array_of_sizes, array_of_subsizes,
      array_of_starts, MPI_ORDER_C, MPI_BYTE, &cuboid);
```



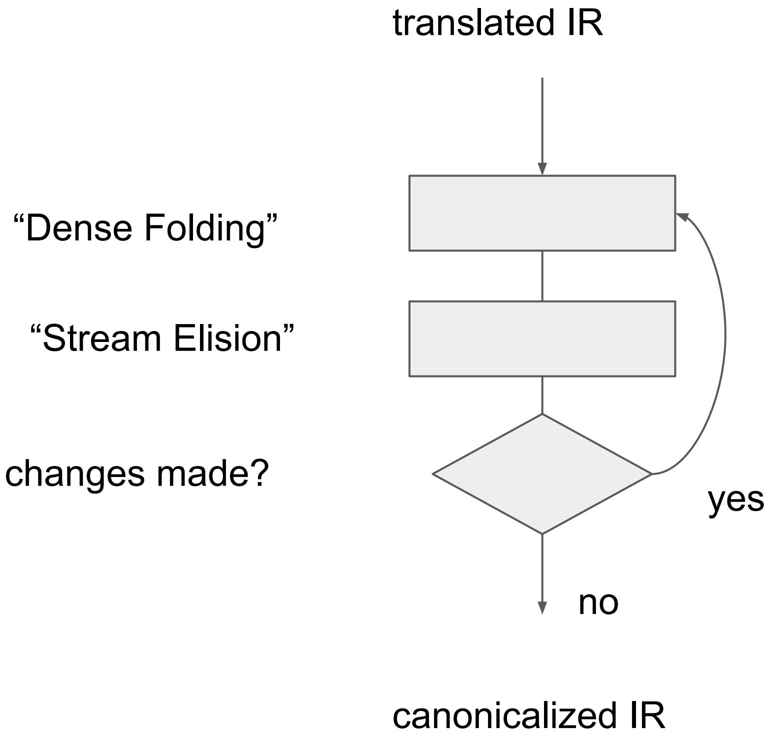
TEMPI Datatype Handling

Translation
Convert MPI Derived Datatype into internal representation (IR)

Canonicalization
Convert semantically-equivalent IR to simplified form

Kernel Selection
Choose packing/unpacking kernel for IR

Canonicalization



Canonicalization: Stream Elision

An MPI vector will commonly have a block of 1 child element

...

StreamData{offset:X, count:5, stride:4}

StreamData{offset:Y, count:1, stride:_}

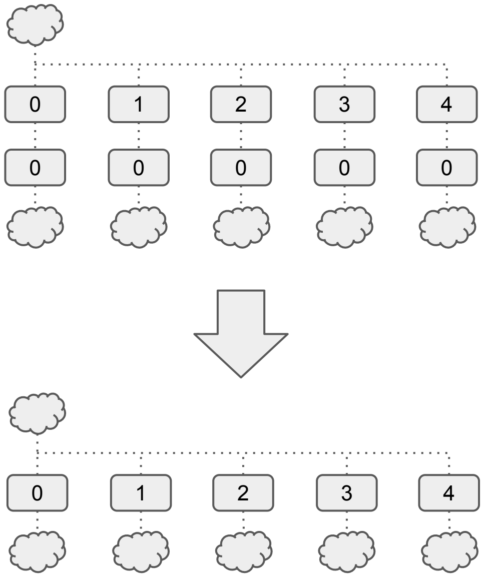
...

↓

...

StreamData{offset:X+Y, count:5, stride:4}

...



Canonicalization: Dense Folding

When a stream is actually multiple dense elements
A parent type of an MPI named type

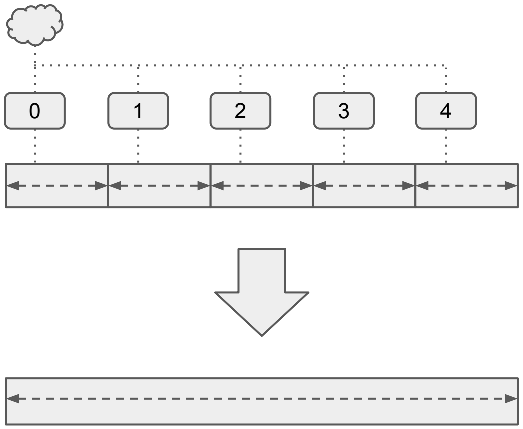
...

`StreamData{offset:X, count:5, stride:4}`

`DenseData{offset:Y, extent:4}`

...

`DenseData{offset:X+Y, extent:4}`



TEMPI Datatype Handling

Translation
Convert MPI Derived Datatype into internal representation (IR)

↓

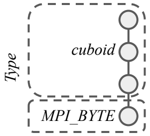
Canonicalization
Convert semantically-equivalent IR to simplified form

↓

Kernel Selection
Choose packing/unpacking kernel for IR

Generalized pack kernels

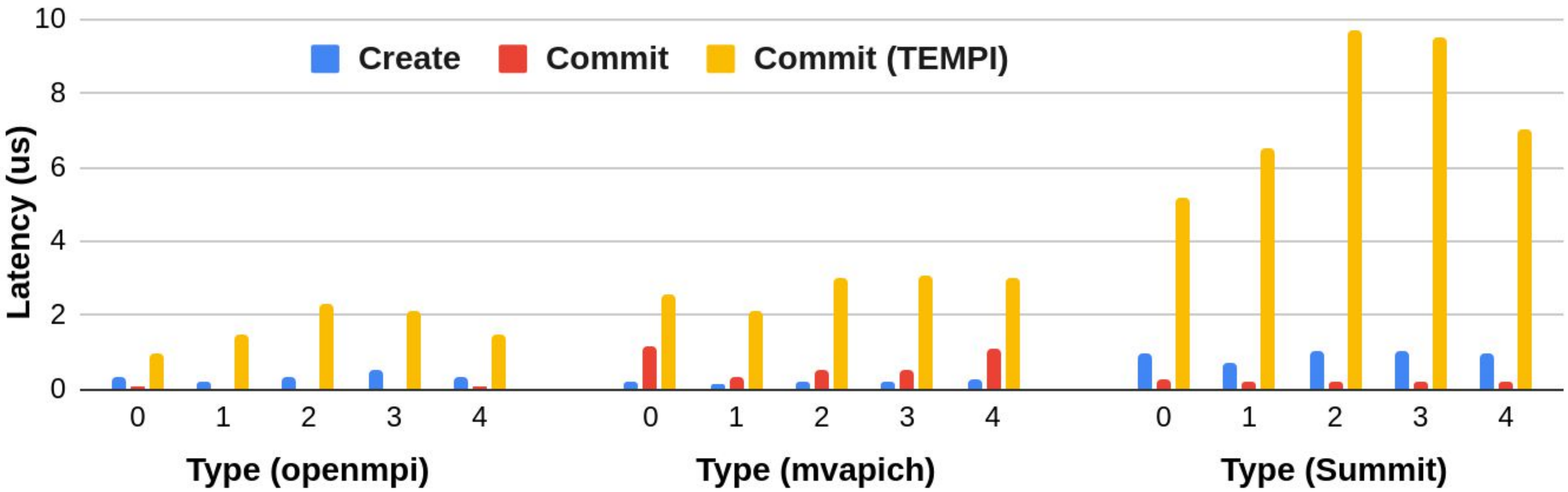
- N-D pack kernel
  - x dimension to lowest StreamData
  - y dimension to next lowest
  - z dimension to next lowest
  - loops after that
  - One thread per word
- Parameterized on word size W
  - specialized to W=1,2,4,8
- Dispatch at run-time by GCF of alignment and contiguous block size



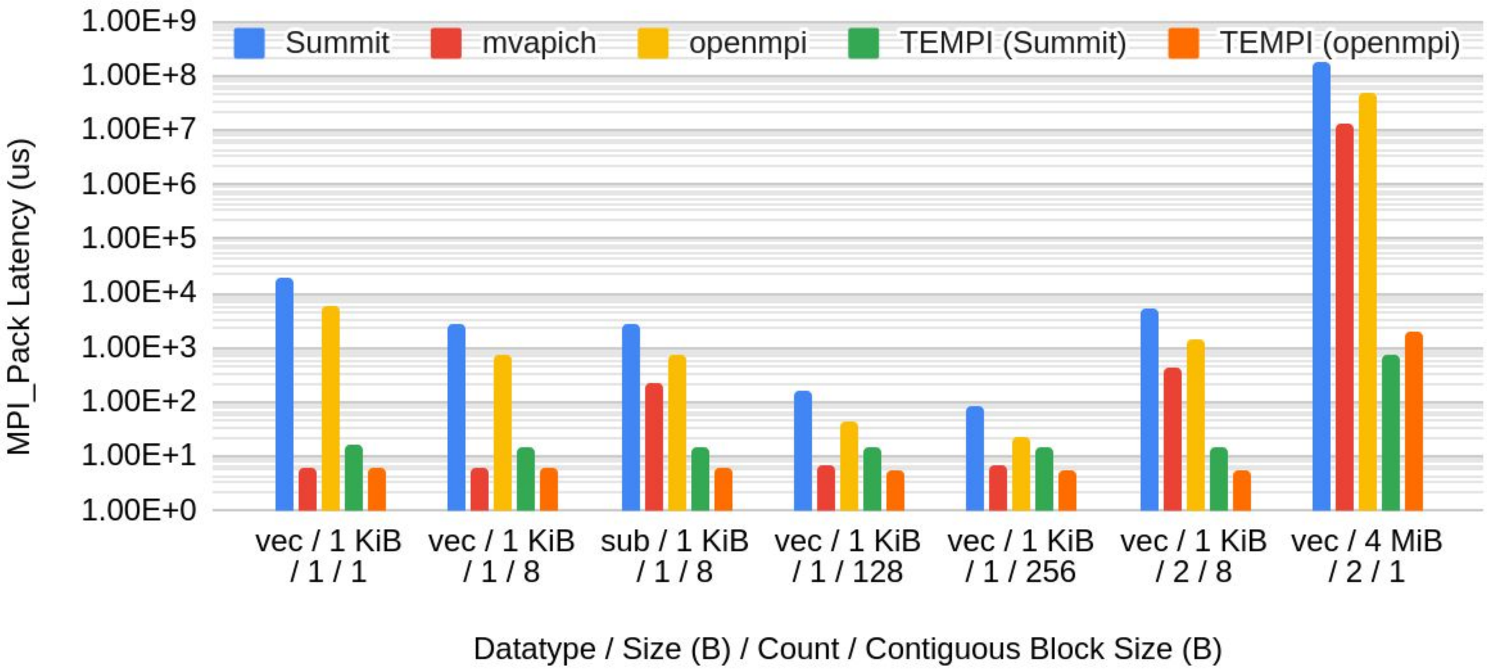

```
StreamData{offset:0, count:47,  stride:131072}
StreamData{offset:0, count:13,  stride:256}
StreamData{offset:0, count:100, stride:1}
DenseData{offset:0,  extent: 1}
```

```
z dimension = 47
y dimension = 13
x dimension = 100 / W
W = 4
```

MPI_Type_commit Time

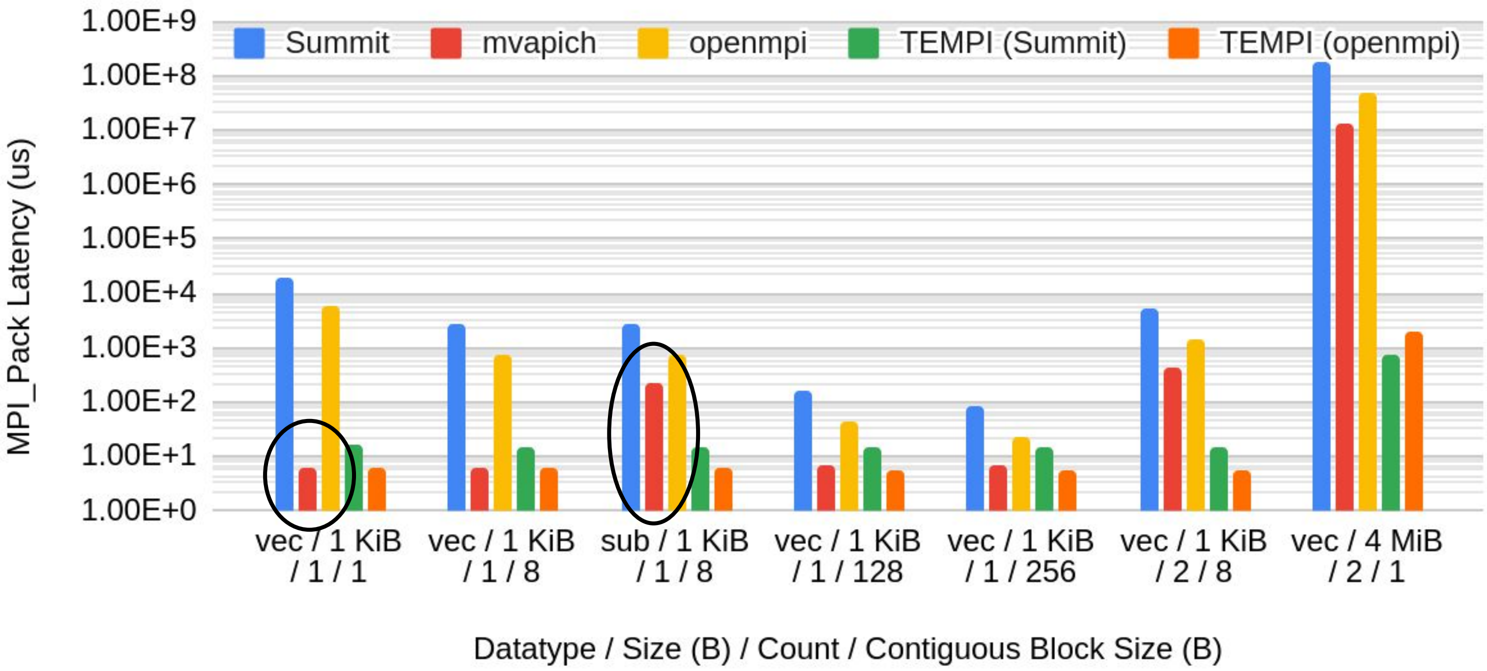


MPI_Pack



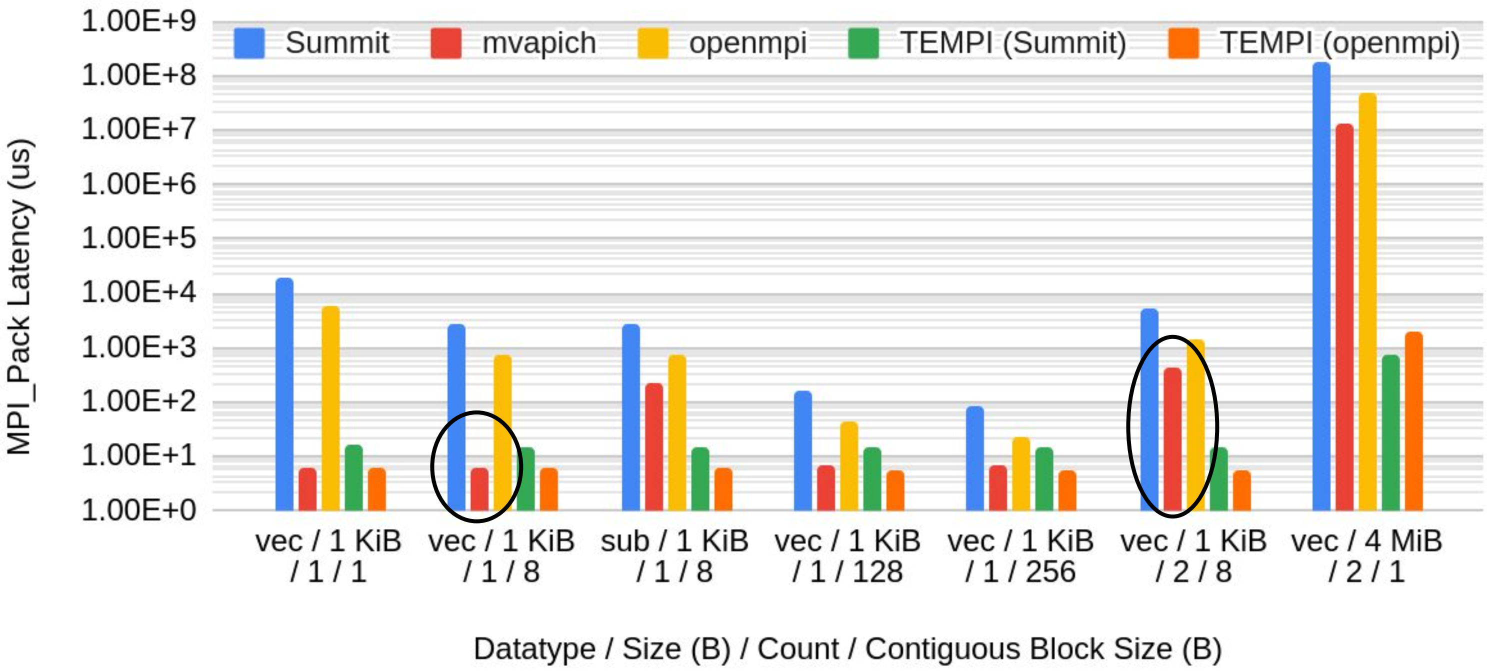
Summit
mvapich
openmpi
TEMPI (Summit)
TEMPI (openmpi)
MPI_Pack Latency (us)
1.00E+9
1.00E+8
1.00E+7
1.00E+6
1.00E+5
1.00E+4
1.00E+3
1.00E+2
1.00E+1
1.00E+0
vec / 1 KiB / 1 / 1
vec / 1 KiB / 1 / 8
sub / 1 KiB / 1 / 8
vec / 1 KiB / 1 / 128
vec / 1 KiB / 1 / 256
vec / 1 KiB / 2 / 8
vec / 4 MiB / 2 / 1
Datatype / Size (B) / Count / Contiguous Block Size (B)

MPI_Pack



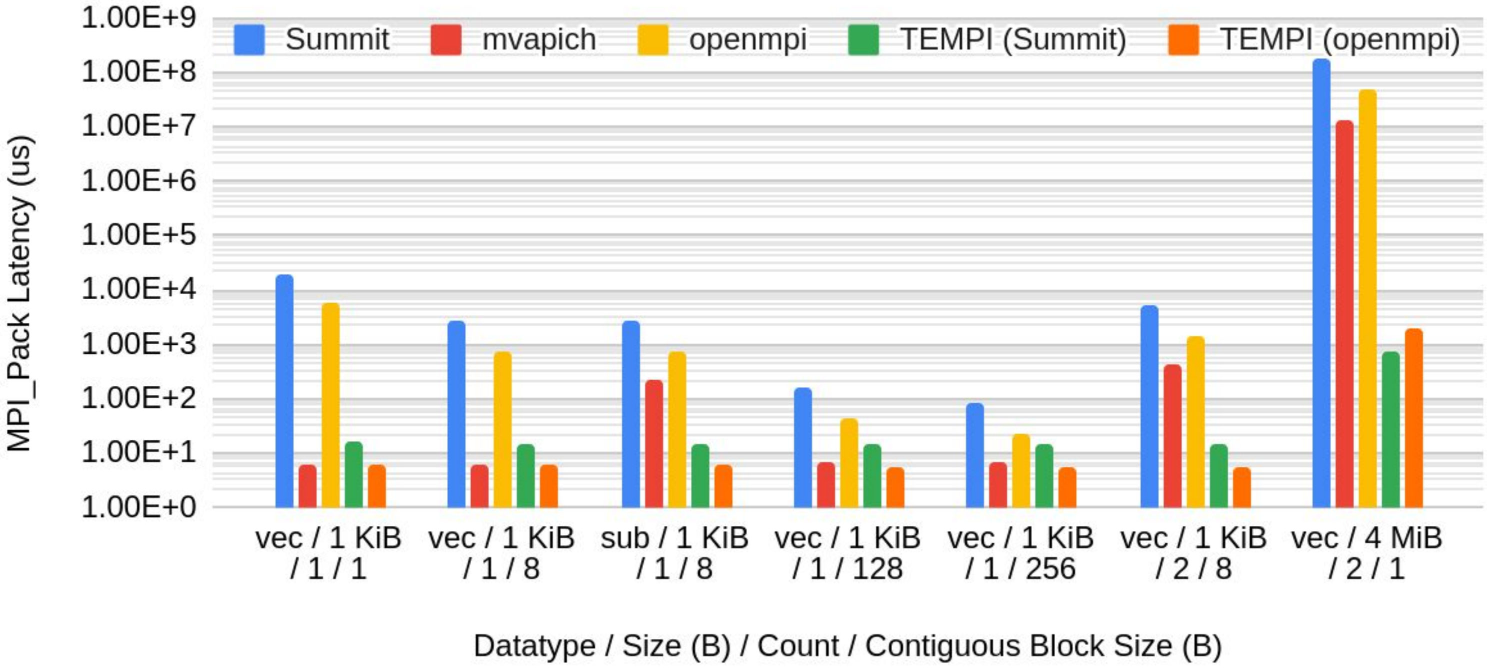
MVAPICH performance depends on type

MPI_Pack



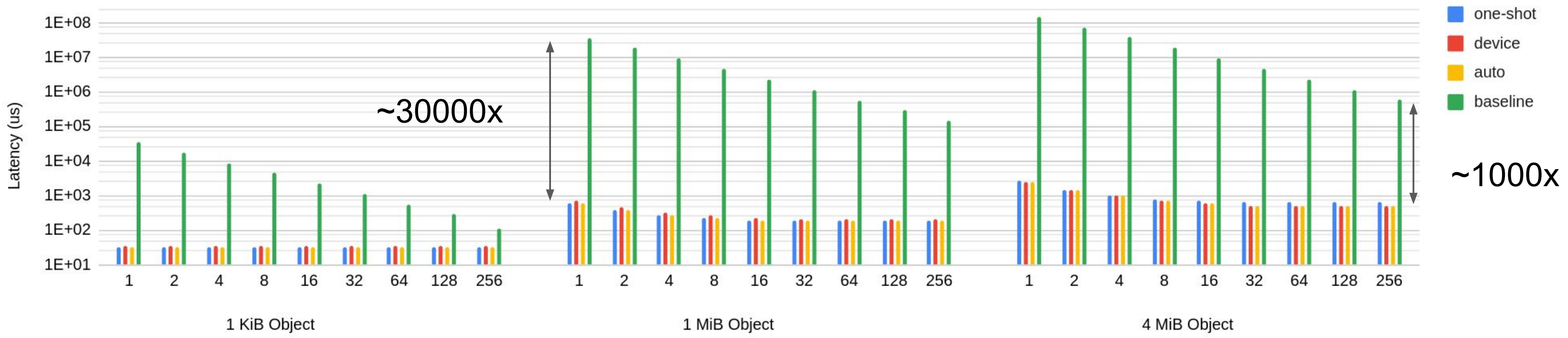
MVAPICH performance depends on count

MPI_Pack



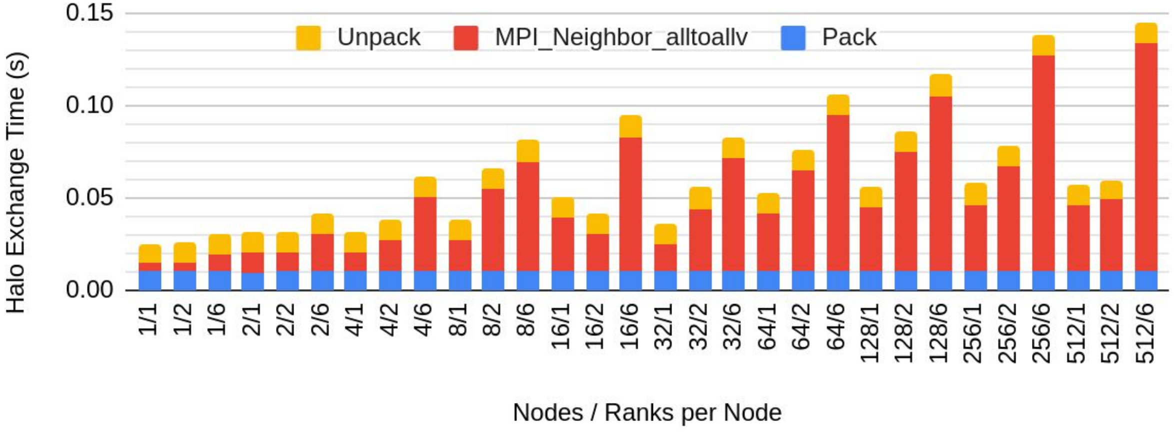
TEMPI is fast regardless of underlying MPI, type, or count

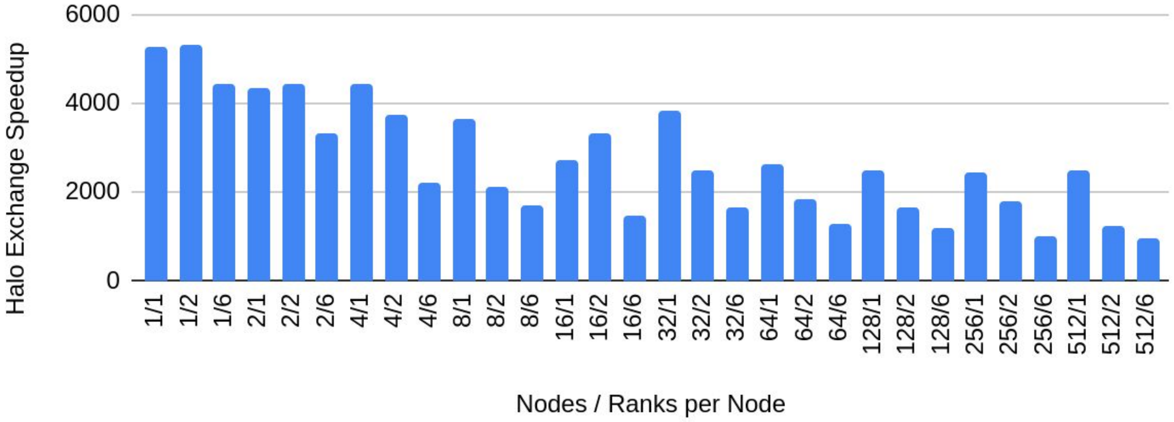
MPI_Send / MPI_Recv



MPI_Send/Recv Latency for 2D objects with different block sizes

Halo Exchange





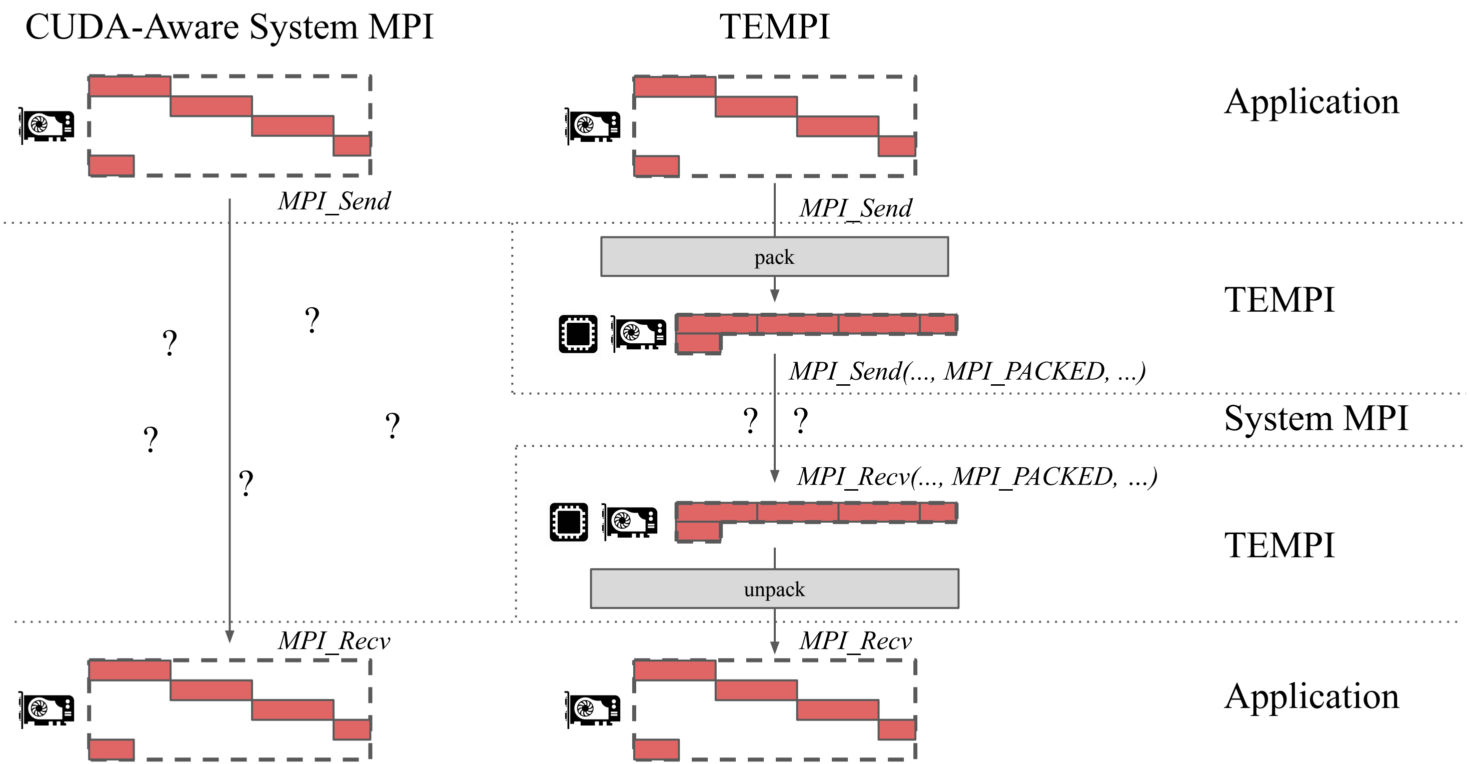
Software

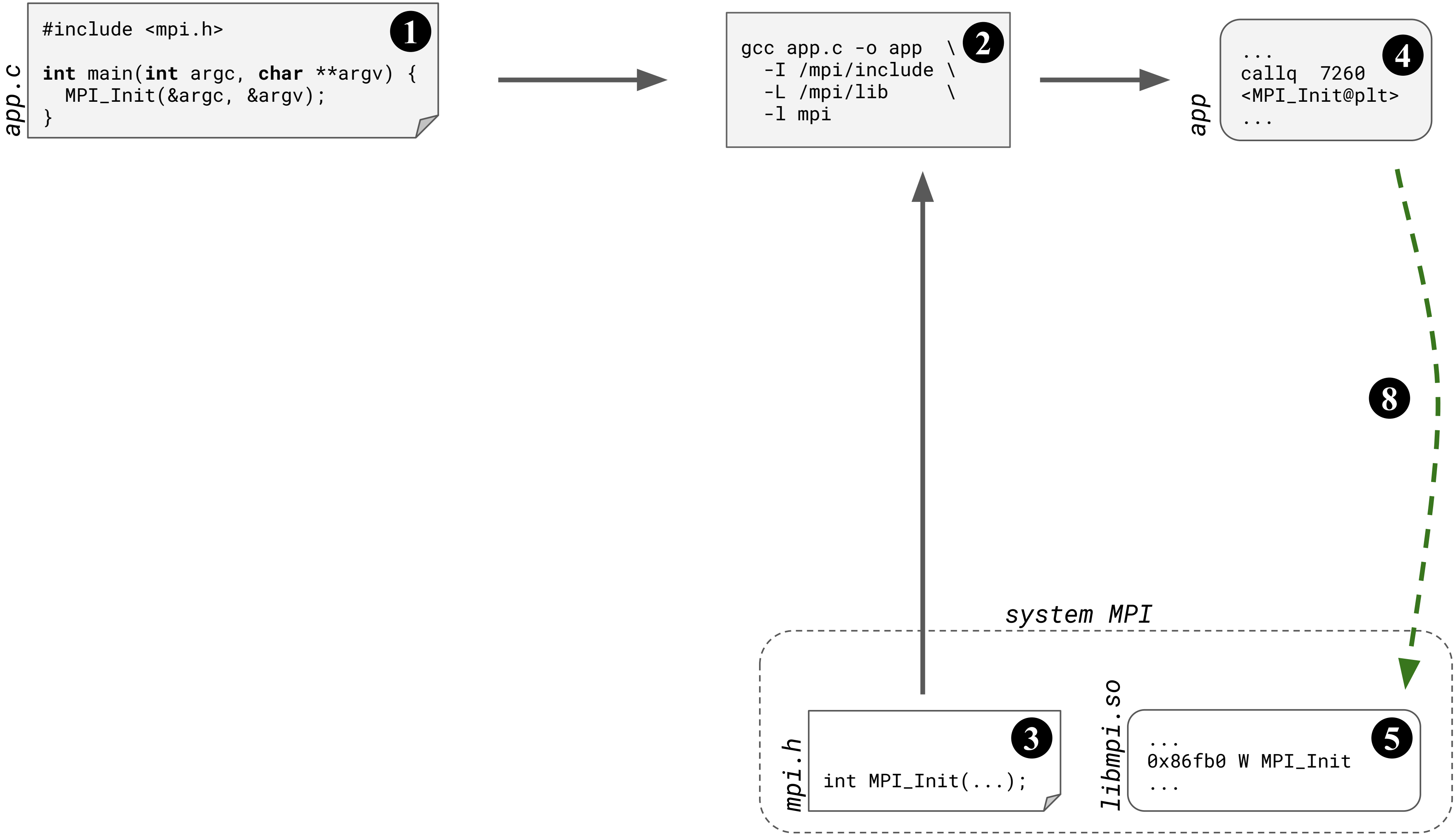
- The life
  - You
  - It ta
  - Fixi / test it
  - I gue
  - ¯_(ツ
- The life
  - You
  - You cale
  - You ations
  - ¯_(ツ

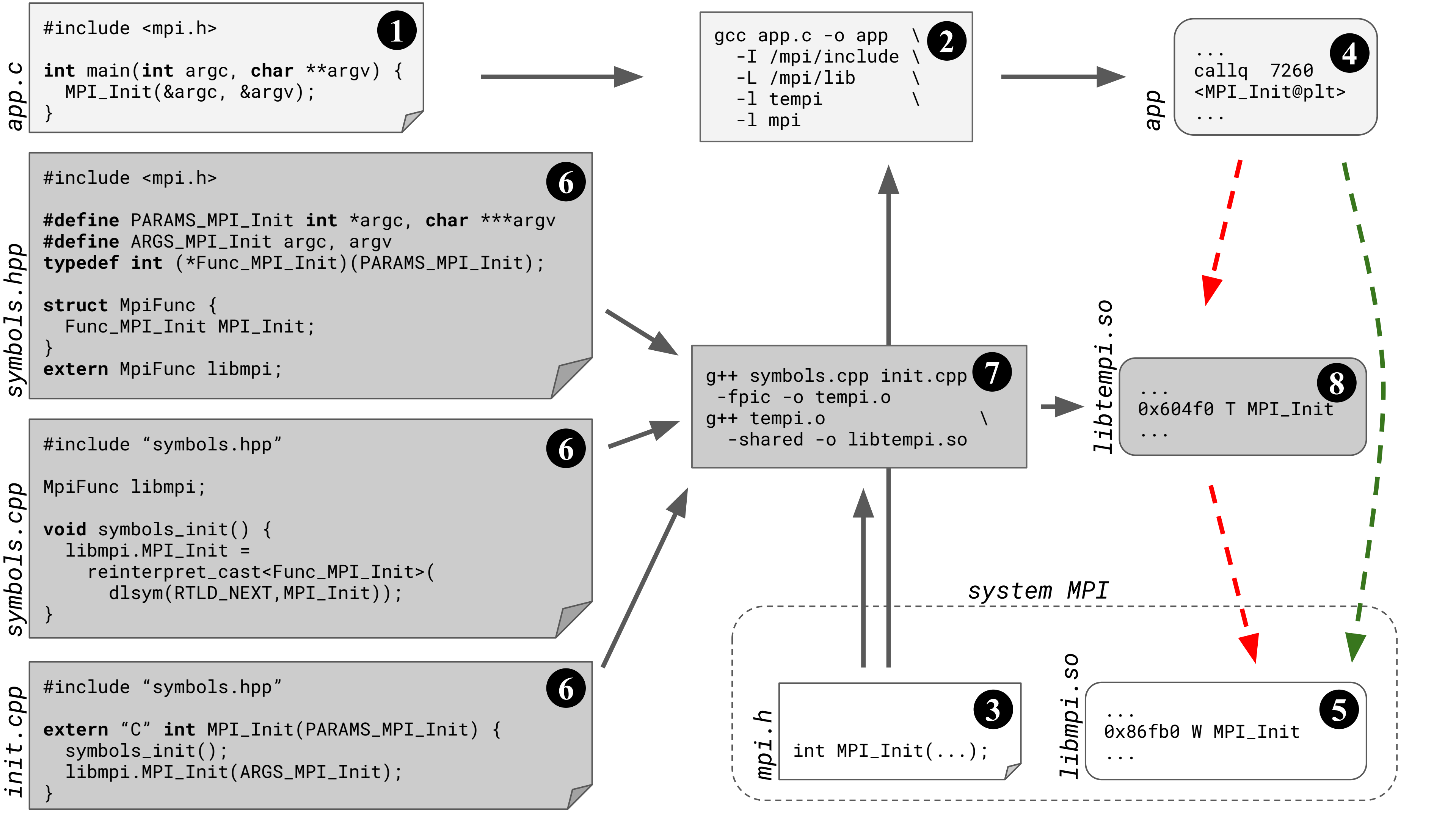
Wonder Woman 1984

TEMPI

- “Temporary MPI” / “Topology Experiments for MPI” / plural of tempo (speed)
- MPI interposer




app.c
#include <mpi.h>
int main(int argc, char **argv) {
MPI_Init(&argc, &argv);
}
1
gcc app.c -o app \
-I /mpi/include \
-L /mpi/lib \
-l mpi
2
app
...
callq 7260
<MPI_Init@plt>
...
4
8
system MPI
mpi.h
int MPI_Init(...);
3
libmpi.so
...
0x86fb0 W MPI_Init
...
5


app.c
#include <mpi.h>
int main(int argc, char **argv) {
MPI_Init(&argc, &argv);
}
1
gcc app.c -o app \
-I /mpi/include \
-L /mpi/lib \
-l tempi \
-l mpi
2
app
...
callq 7260
<MPI_Init@plt>
...
4
symbols.hpp
#include <mpi.h>
#define PARAMS_MPI_Init int *argc, char ***argv
#define ARGS_MPI_Init argc, argv
typedef int (*Func_MPI_Init)(PARAMS_MPI_Init);
struct MpiFunc {
Func_MPI_Init MPI_Init;
}
extern MpiFunc libmpi;
6
symbols.cpp
#include “symbols.hpp”
MpiFunc libmpi;
void symbols_init() {
libmpi.MPI_Init =
reinterpret_cast<Func_MPI_Init>(
dlsym(RTLD_NEXT,MPI_Init));
}
6
init.cpp
#include “symbols.hpp”
extern “C” int MPI_Init(PARAMS_MPI_Init) {
symbols_init();
libmpi.MPI_Init(ARGS_MPI_Init);
}
6
g++ symbols.cpp init.cpp
-fpic -o tempi.o
g++ tempi.o \
-shared -o libtempi.so
7
libtempi.so
...
0x604f0 T MPI_Init
...
8
system MPI
mpi.h
int MPI_Init(...);
3
libmpi.so
...
0x86fb0 W MPI_Init
...
5

Pre-pre-conclusion: Software Engineering

- Compile TEMPI on the system
  - C++17 (std::variant) / CUDA14 (std::make_unique)
- Ensure libtempi.so is first in your link order
  - re-link your application, or LD_PRELOAD
- OS loader will find MPI symbols in TEMPI
  - TEMPI will do something and call system MPI as needed
- Works with unmodified applications
- Requires no elevated privileges
- Modify / implement as much of MPI as you want to, transparently
  - https://github.com/cwpearson/tempi

Pre-conclusion: things I didn't have slides for

- MVAPICH-GDR & other existing work on GPU + MPI datatypes
    - I couldn't get MVAPICH-GDR working after a few weeks (hard to evaluate)
    - Other work doesn't actually seem to be available (hard to compare against...)
    - TEMPI is designed to avoid these problems
- Multiple smaller blocks vs one large block
    - pipelining
    - bandwidth vs overhead tradeoff
- Ideas for accelerator-friendly MPI functions
    - Fewer guarantees (e.g. not allowed to use buffer after return until MPI_Wait)
    - More opportunities to let MPI make optimizations / amortize accelerator overhead
        - Express intent, (not just implementation)
        - e.g. MPI_Dist_create_graph ("these ranks will communicate a lot")
        - e.g. persistent communication ("this communication will happen a lot")
        - Something akin to CUDA streams + CUDA graph API?

Conclusion

- New MPI derived datatype approach
- TEMPI is an MPI research platform / strategy
  - demonstrated orders-of-magnitude speedup on a real system
  - tested with MVAPICH, OpenMPI, Spectrum MPI
  - no modification of application code
  - no modification of existing MPI
    - closed-source / binary distribution
    - open-source and complicated
  - Boost software license
    - Use any of it for any purpose, (usually with attribution)

Thank You

- pearson at illinois dot edu
- https://go.illinois.edu/TEMPI
    - https://github.com/cwpearson/tempi
    - https://carlpearson.net for links to slides / paper

Abstract

MPI derived datatypes are an abstraction that simplifies handling of non-contiguous data in MPI applications. These datatypes are recursively constructed at runtime from primitive Named Types defined in the MPI standard. More recently, the development and deployment of CUDA-aware MPI implementations has encouraged the transition of distributed high-performance MPI codes to use GPUs. Such implementations allow MPI functions to directly operate on GPU buffers, easing integration of GPU compute into MPI codes. Despite substantial attention to CUDA-aware MPI implementations, they continue to offer cripplingly poor GPU performance when manipulating derived datatypes on GPUs. This work presents a new MPI library, TEMPI, to address this issue. TEMPI introduces a common representation for equivalent MPI derived datatypes, and fast kernels for that representation. TEMPI can be used as an interposed library on existing MPI deployments without system or application changes. Furthermore, this talk will discuss a performance model of GPU derived datatype handling, demonstrating that previously preferred "one-shot" methods are not always fastest.