CASS Community BoF Days
June 12, 2024

MPICH: A High-Performance Open Source MPI Library for Leadership-class HPC Systems

Agenda

- Argonne Update – Yanfei Guo
- User presentations
  - Jeff Hammond (NVIDIA)
  - Vitali Morozov (Argonne)
  - Wei-keng Liao (Northwestern University)
  - Jiajun Huang (ANL/University of California, Riverside)
  - Junchao Zhang (ANL)
- Wrap Up/Q&A

MPICH: Status and Upcoming Releases
http://www.mpich.org

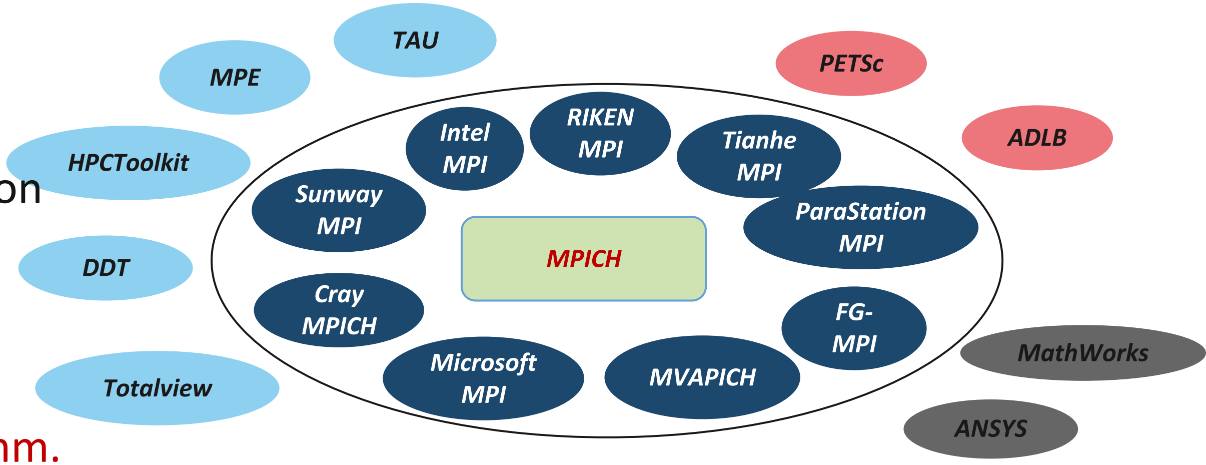
Ken Raffenetti, **Yanfei Guo**, Hui Zhou, Rajeev Thakur

Argonne National Laboratory

MPICH turns 31

The MPICH Project

- Funded by DOE for 31 years
- Has been a key influencer in the adoption of MPI
  - First/most comprehensive implementation of every MPI standard
  - Allows supercomputing centers to not compromise on what features they demand from vendors
- DOE R&D100 award in 2005 for MPICH
- DOE R&D100 award in 2019 for UCX (MPICH internal comm. layer)
- MPICH and its derivatives are the world’s most widely used MPI implementations



MPICH is not just a software

It’s an Ecosystem

MPICH Adoption in Exascale Machines

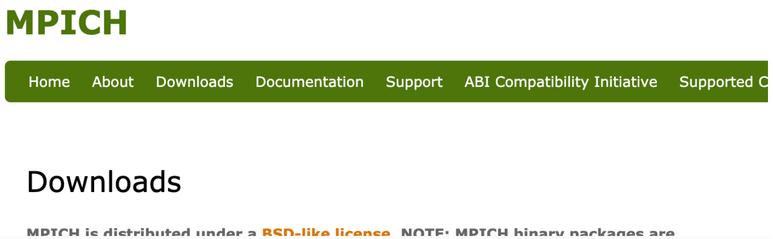
- Aurora, ANL, USA (Intel MPI for Aurora)
- Frontier, ORNL, USA (Cray MPICH)
- El Capitan, LLNL, USA (Cray MPICH)

MPICH ABI Compatibility Initiative

- Binary compatibility for MPI implementations
  - Started in 2013
  - Explicit goal of maintaining ABI compatibility between multiple MPICH derivatives
  - Collaborators:
    - MPICH (since v3.1, 2013)
    - Intel MPI Library (since v5.0, 2014)
    - Cray MPICH (starting v7.0, 2014)
    - MVAPICH2 (starting v2.0, 2017)
    - Parastation MPI (starting v5.1.7-1, 2017)
- Open initiative: other MPI implementations are welcome to join
- http://www.mpich.org/abi
- MPI Standard ABI update in later slides…

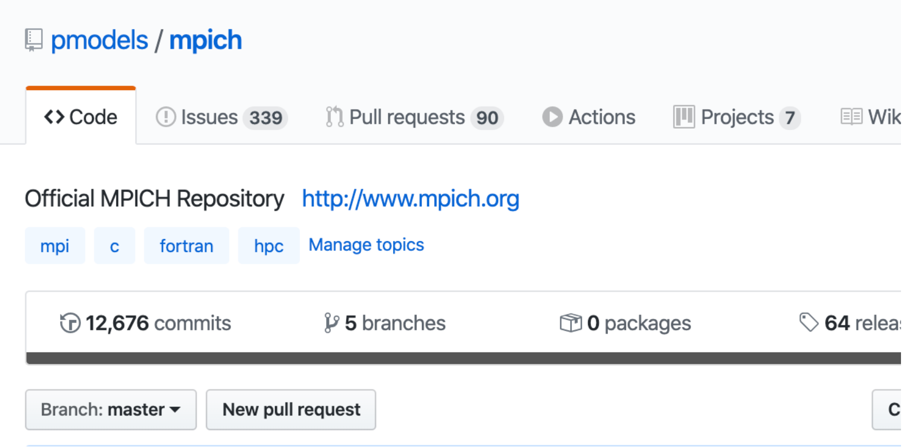
MPICH Distribution Model

- Source Code Distribution
  - MPICH Website, Github
- Binary Distribution through OS Distros and Package Managers
  - Redhat, CentOS, Debian, Ubuntu, Homebrew (Mac)
- Distribution through HPC Package Managers
  - Spack, OpenHPC, E4S
- Distribution through Vendor Derivatives



Downloads

MPICH is distributed under a BSD-like license. NOTE: MPICH binary packages are



Official MPICH Repository http://www.mpich.org

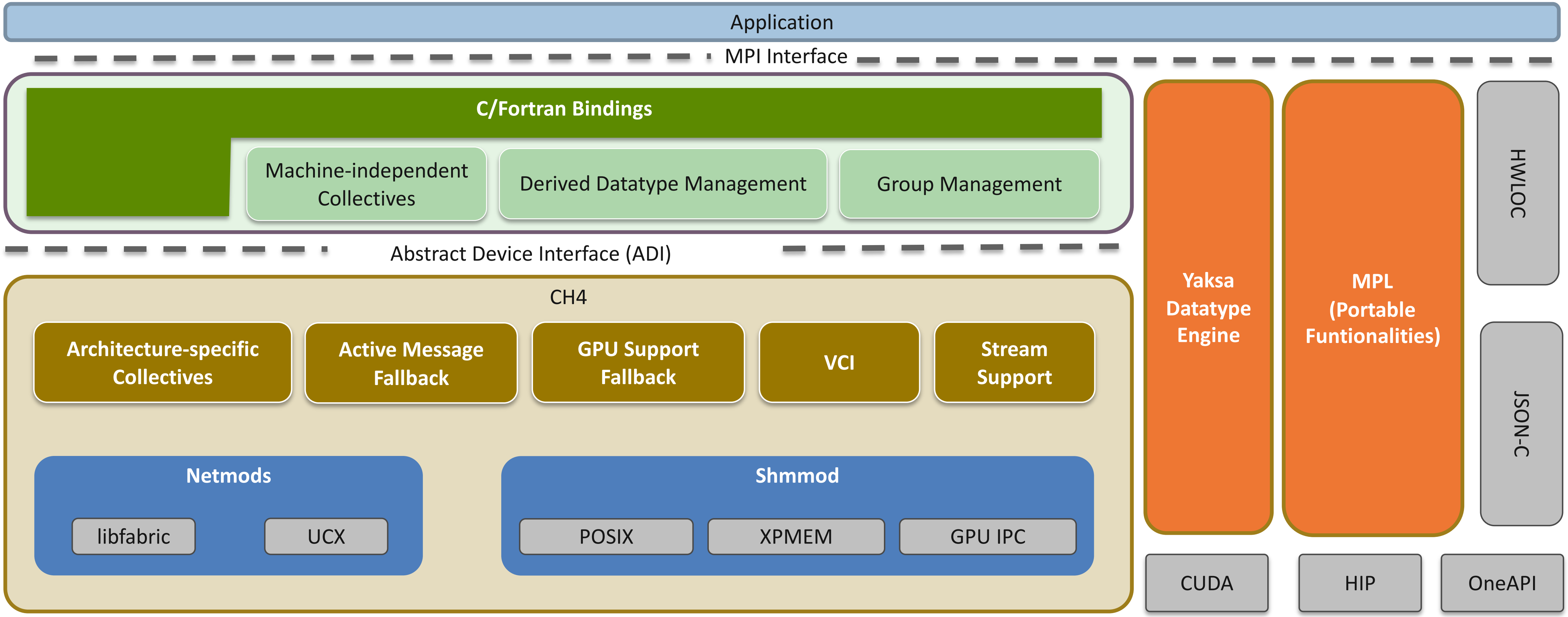
MPICH Support in Spack

- Spack package manager is widely used in HPC
- Many MPICH configurations and features supported
- Recently added options
  - XPMEM variant
  - Improved PMI/PMI2/PMIx variants
- We want to hear from you
  - Are there features missing?
  - Are you unable to build/install on your system?
  - Open an issue on Spack Github (https://github.com/spack/spack), use subject “mpich: <…>” and tag @raffenet, @yfguo, @hzhou

MPICH Releases

- MPICH now aims to follow a 12-month cycle for major releases (4.x)
  - Minor bug fix releases for the current stable release happen every few months
  - Preview releases for the next major release happen every few months
  - Branching off when beta is released (feature freezed)
- Current stable release is in the 4.2.x series
  - mpich-4.2.1 released in March, mpich-4.2.2 release by end of June
- Upcoming major release is in the 4.3.x series
  - mpich-4.3.0b1 release targeted for November @ SC24

MPICH Layered Structure

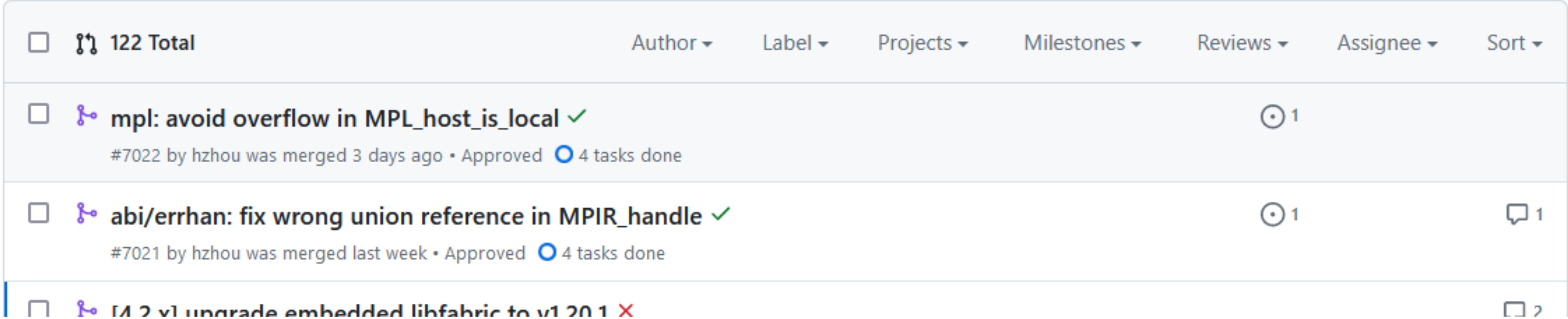


MPICH 4.2

- Full support for MPI 4.1 specification
  - `mpi_memory_alloc_kinds` info hint
  - `MPI_Request_get_status_{all,any,some}`
  - `MPI_Remove_error_{class,code,string}`
  - `MPI_{Comm,Session}_{attach,detach}_buffer`
  - `MPI_BUFFER_AUTOMATIC`
  - Split type `MPI_COMM_TYPE_RESOURCE_GUIDED`
- New experimental features
  - MPI Thread communicator
  - MPI datatype iov query
  - Reduction operator `MPIX_EQUAL`
- Enhanced GPU (esp. ZE) support
- Unified PMI-{1,2,x} support

MPICH 4.3 Update

- Support the new MPI ABI proposal --enable-mpi-abi
- MPIX Async extension – for interoperable MPI progress
  - Custom progress engine can include MPI progress
  - MPI progress can advance custom asynchronous tasks
- Stability and performance issues from Aurora
- Misc fixes and enhancements – 122 merged pull requests so far



□ 122 Total

□ mpl: avoid overflow in MPL_host_is_local ✓
#7022 by hzhou was merged 3 days ago • Approved ○ 4 tasks done
⊙ 1

□ abi/errhan: fix wrong union reference in MPIR_handle ✓
#7021 by hzhou was merged last week • Approved ○ 4 tasks done
⊙ 1 □ 1

□ [4.2.x] upgrade embedded libfabric to v1.20.1 ✗
□ 2

Support for MPI ABI

- Standardized ABI by MPI Forum
  - Portability across different MPI implementations.
  - Simplify package and dependency management of HPC software
- Try today by building MPICH with --enable-mpi-abi
  - Existing MPICH ABI is offered in parallel
- New compiler wrappers
  - mpicc-abi, mpic++-abi

Jeff R. Hammond, Lisandro Dalcin, Erik Schnetter, Marc Pérache, Jean-Baptiste Besnard, Jed Brown, Gonzalo Brito Gadeschi, Joseph Schuchart, Simon Byrne, and Hui Zhou. MPI Application Binary Interface Standardization. In Proceedings of EuroMPI 2023: the 30th European MPI Users' Group Meeting (EUROMPI '23), September 11–13, 2023, Bristol, United Kingdom. ACM, New York, NY, USA. https://doi.org/10.1145/3615318.3615319

New Extension - MPIX_Op_create_x

- The "old" op user function caters to a Fortran calling convention.

```
typedef void (MPI_User_function)(void *invec, void *inoutvec,
                                 int *len, MPI_Datatype *datatype);
```

- It assumes integer handles, which won't work with Fortran.
- It won't work with any non-C/C++ user functions.
- Current MPICH Fortran binding relies on non-standard, language-specific ABIs.

```
void MPII_Op_set_fc(MPI_Op);
void MPII_Op_set_cxx(MPI_Op);
```

- Proposed fix – add a context and a destructor to support binding proxy functions.

```
int MPIX_Op_create_x(MPIX_User_function_x *user_fn_x,
                     MPIX_Destructor_function *destructor_fn,
                      int commute, void *extra_state, MPI_Op *op);
Typedef void (MPIX_User_function_x)(void *invec, void *inoutvec,
                                   MPI_Count len, MPI_Datatype datatype,
                                   void *extra_state);
Typedef void (MPIX_Destructor_function)(void *extra_state);
```

New Extensions to Enable Inter-operable MPI Progress

- Explicit MPI progress

```
int MPIX_Stream_progress(MPIX_Stream stream);
```

- MPIX Async

```
int MPIX_Async_start(MPIX_Async_poll_function poll_fn,
                      void *extra_state, MPIX_Stream stream);

enum {
    MPIX_ASYNC_PENDING = 0,
    MPIX_ASYNC_DONE = 1,
};

typedef struct MPIR_Async_thing *MPIX_Async_thing;
typedef int (MPIX_Async_poll_function)(MPIX_Async_thing);

void *MPIX_Async_get_state(MPIX_Async_thing async_thing);

void *MPIX_Async_spawn(MPIX_Async_thing async_thing,
                        MPIX_Async_poll_function poll_fn,
                        void *extra_state, MPIX_Stream stream);
```

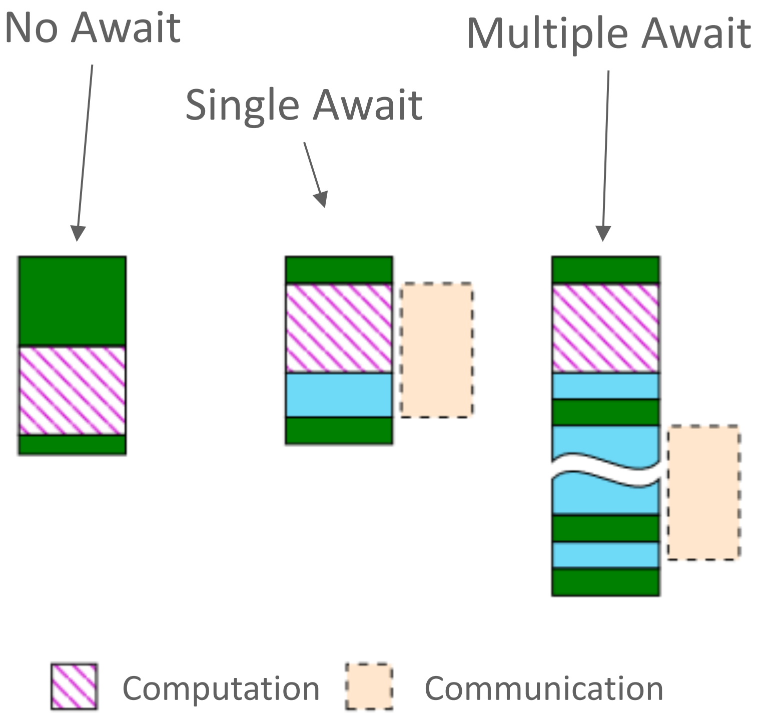
- Lightweight request completion query

```
bool MPIX_Request_is_complete(MPI_Request request);
```

Hui Zhou, Robert Latham, Ken Raffenetti, Yanfei Guo and Rajeev Thakur. MPI Progress For All. https://arxiv.org/pdf/2405.13807

The problem of “fancy” communications

- Three Async Patterns
  - No Await - *e.g.* light weight send
  - Single Await – *e.g.* “strong progress”
  - Multiple Await – *e.g.* fancy schemes require handshakes
- Good computation/communication overlaps are only possible with single await patterns.
- It is more common to require fancy schemes for communication performance due to increasingly hybrid systems.

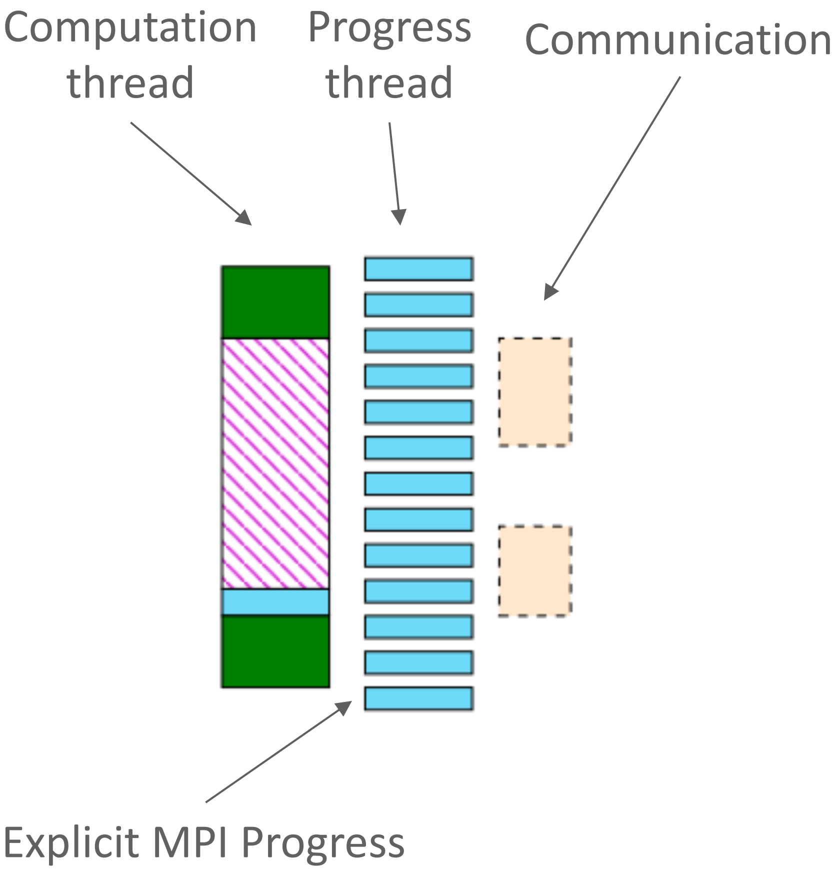


Why we need explicit MPI progress

- To achieve computation/communication overlap, we require a progression scheme, e.g. a progress thread.
- Default global async thread does not work
  - Waste resource when it is not needed
  - Severely degrade performance due to thread contentions
- Solution – explicit MPI progress

```
int MPIX_Stream_progress(MPIX_Stream stream);
```

  - On-demand invocation
  - Per-stream progress



Integrate custom progress hooks into MPI progress

- Enable users to extend MPI by building custom communication algorithms
- Integrate custom progress hooks –
  - Allows for seamless MPI framework, minimize the effort of porting applications
  - Avoid the complexity of building separate progression mechanisms
  - Achieve equivalent performance to a native MPI implementation

```
int MPIX_Async_start(MPIX_Async_poll_function poll_fn,
                     void *extra_state, MPIX_Stream stream);
```

Lightweight request completion query

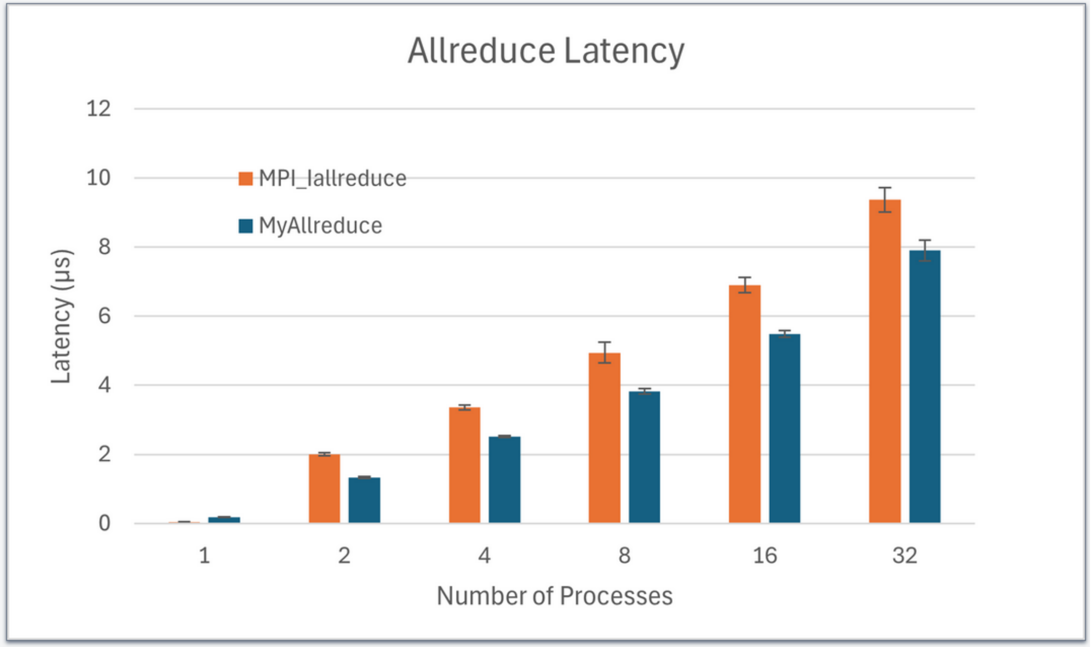
- Asynchronous workflow need to check dependency status
- MPI_Test invokes MPI_Progress
  - It contends with progress engine
  - It does more than what is needed – filling status and freeing requests
- MPIX_Request_is_complete is
  - Lightweight (essentially an atomic query).
  - No side effects.

```
bool MPIX_Request_is_complete(MPI_Request request);
```

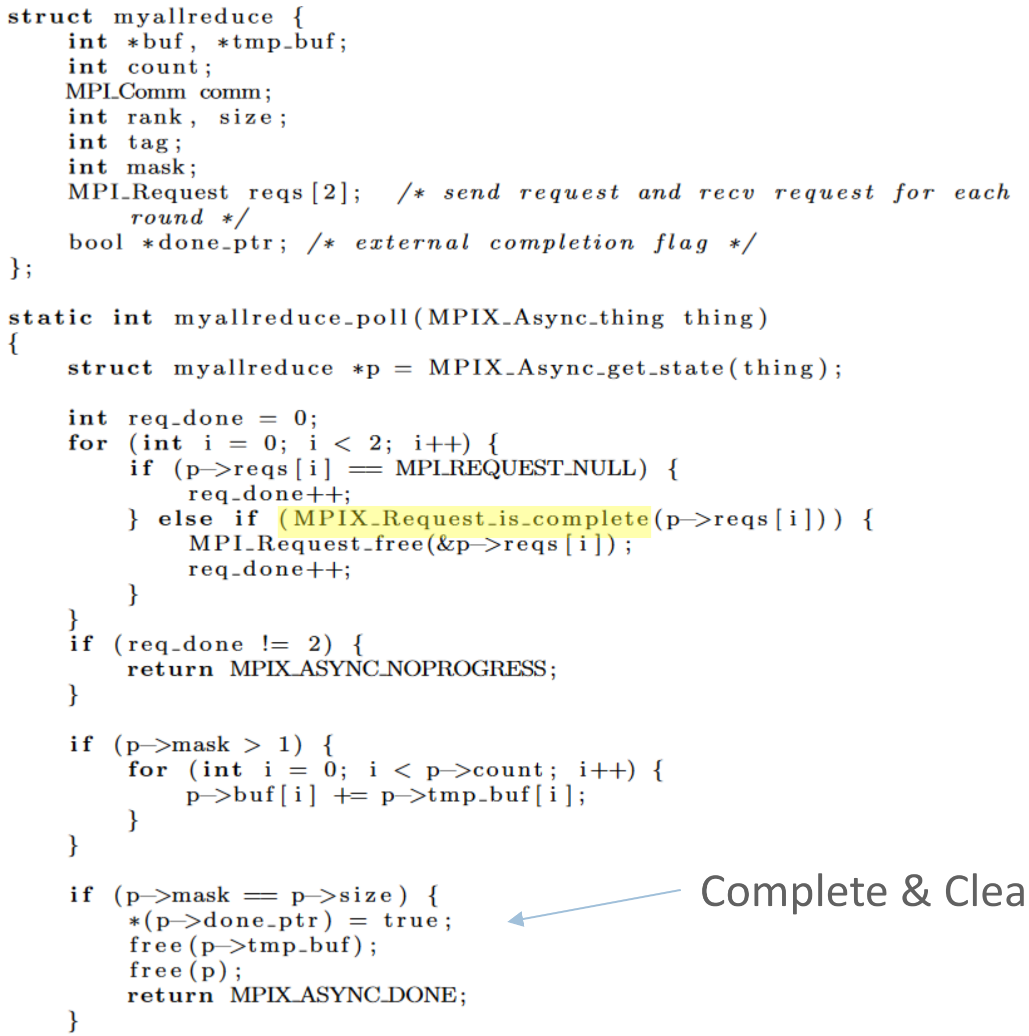
Example: Allreduce Implementation outside of a MPI Library

- Recursive doubling algorithm implemented in outside vs inside an MPI library.

- "MyAllreduce" assumes MPI_IN_PLACE, MPI_INT, MPI_SUM, and a power-of-2 communicator size.
- It out-performs the native implementation due to these assumptions (shortcuts).

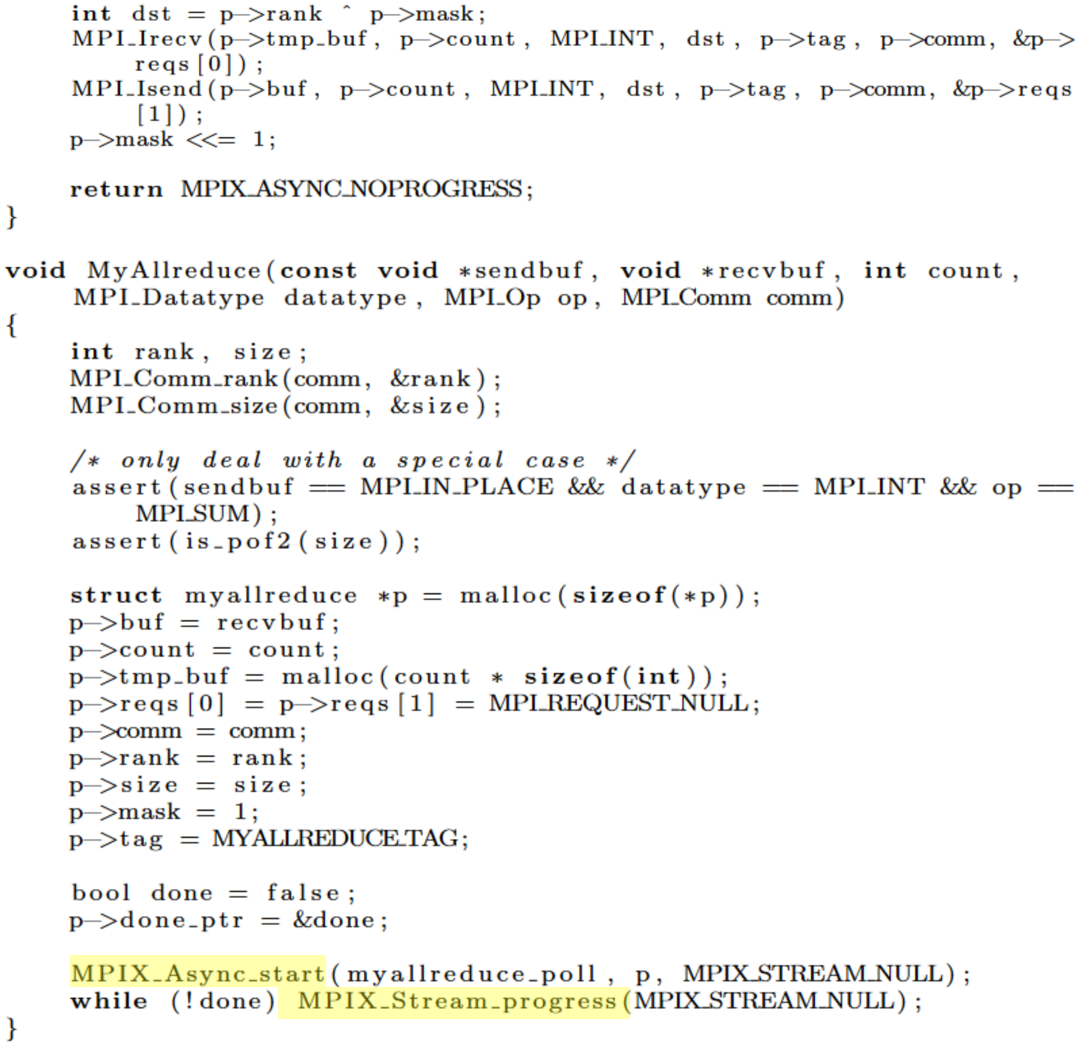


Example: custom Allreduce



```
struct myallreduce {
    int *buf, *tmp_buf;
    int count;
    MPI_Comm comm;
    int rank, size;
    int tag;
    int mask;
    MPI_Request reqs[2];  /* send request and recv request for each
        round */
    bool *done_ptr; /* external completion flag */
};

static int myallreduce_poll(MPIX_Async_thing thing)
{
    struct myallreduce *p = MPIX_Async_get_state(thing);

    int req_done = 0;
    for (int i = 0; i < 2; i++) {
        if (p->reqs[i] == MPI_REQUEST_NULL) {
            req_done++;
        } else if (MPIX_Request_is_complete(p->reqs[i])) {
            MPI_Request_free(&p->reqs[i]);
            req_done++;
        }
    }
    if (req_done != 2) {
        return MPIX_ASYNC_NOPROGRESS;
    }

    if (p->mask > 1) {
        for (int i = 0; i < p->count; i++) {
            p->buf[i] += p->tmp_buf[i];
        }
    }

    if (p->mask == p->size) {
        *(p->done_ptr) = true;
        free(p->tmp_buf);
        free(p);
        return MPIX_ASYNC_DONE;
    }
```

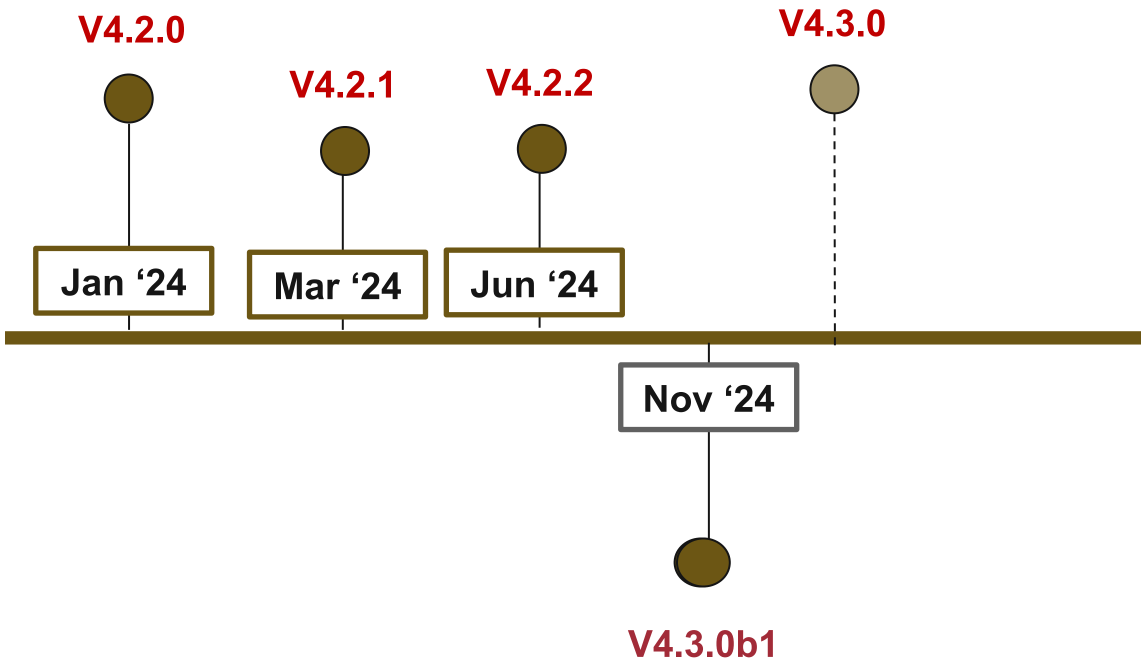
Complete & Cleanup



```
    int dst = p->rank ^ p->mask;
    MPI_Irecv(p->tmp_buf, p->count, MPI_INT, dst, p->tag, p->comm, &p->
        reqs[0]);
    MPI_Isend(p->buf, p->count, MPI_INT, dst, p->tag, p->comm, &p->reqs
        [1]);
    p->mask <<= 1;

    return MPIX_ASYNC_NOPROGRESS;
}

void MyAllreduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
{
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    /* only deal with a special case */
    assert(sendbuf == MPI_IN_PLACE && datatype == MPI_INT && op ==
        MPI_SUM);
    assert(is_pof2(size));

    struct myallreduce *p = malloc(sizeof(*p));
    p->buf = recvbuf;
    p->count = count;
    p->tmp_buf = malloc(count * sizeof(int));
    p->reqs[0] = p->reqs[1] = MPI_REQUEST_NULL;
    p->comm = comm;
    p->rank = rank;
    p->size = size;
    p->mask = 1;
    p->tag = MYALLREDUCE_TAG;

    bool done = false;
    p->done_ptr = &done;

    MPIX_Async_start(myallreduce_poll, p, MPIX_STREAM_NULL);
    while (!done) MPIX_Stream_progress(MPIX_STREAM_NULL);
}
```

MPICH 4.3.0 Roadmap

- MPICH-4.3.0b1 in November 2024
  - 4.3.x branch is created
- GA release in late 2024/early 2025
- Critical bug fixes are backported to 4.2.x



Thank you!

- https://www.mpich.org
- Mailing list: discuss@mpich.org
- Issues and Pull requests: https://github.com/pmodels/mpich
- Weekly development call every Thursday at 9am (central): https://bit.ly/mpich-dev-call

Using MPICH for Fun and Profit

Jeff Hammond
Principal Architect
HPC Software

Outline

1. MPI ABI Collaboration
2. MPI Fortran 2008 (VAPAA)
3. ~~MPI-3 RMA (ARMCI-MPI)~~

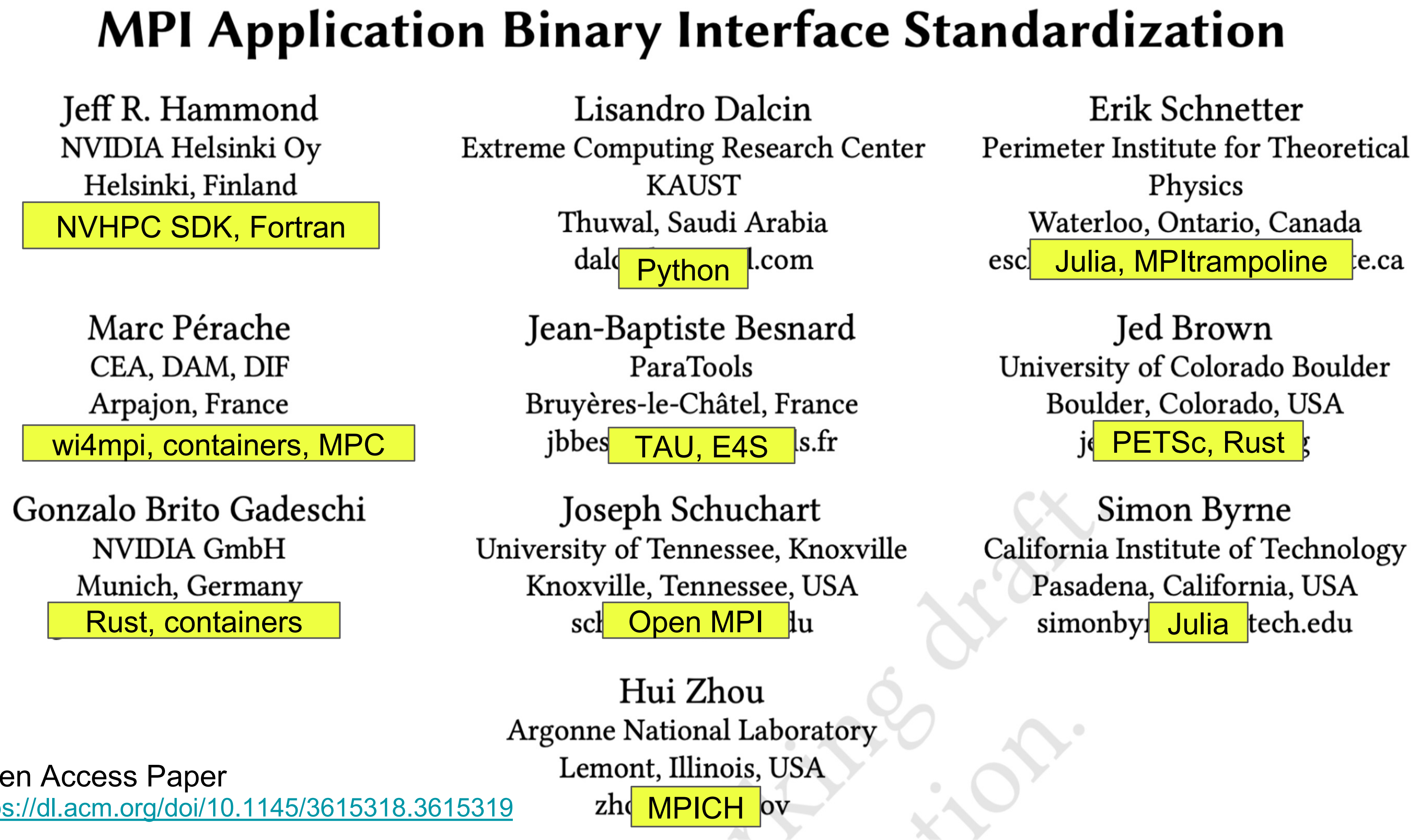
MPI ABI

MPI ABI Standardization

Goal: interoperability between implementations: build once, run many.

History:

- 2006: users want a common or standard ABI
  - 2016: CEA wi4mpi project began
  - 2021: Erik Schnetter creates MPI Trampoline
  - 2021: ABI standardization effort begins
  - 2023: I created Mukautuva, Hui adds ABI prototype to MPICH



MPI Application Binary Interface Standardization

Jeff R. Hammond
NVIDIA Helsinki Oy
Helsinki, Finland
NVHPC SDK, Fortran

Lisandro Dalcin
Extreme Computing Research Center
KAUST
Thuwal, Saudi Arabia
dal Python l.com

Erik Schnetter
Perimeter Institute for Theoretical Physics
Waterloo, Ontario, Canada
esc Julia, MPItrampoline e.ca

Marc Pérache
CEA, DAM, DIF
Arpajon, France
wi4mpi, containers, MPC

Jean-Baptiste Besnard
ParaTools
Bruyères-le-Châtel, France
jbbes TAU, E4S s.fr

Jed Brown
University of Colorado Boulder
Boulder, Colorado, USA
j PETSc, Rust g

Gonzalo Brito Gadeschi
NVIDIA GmbH
Munich, Germany
Rust, containers

Joseph Schuchart
University of Tennessee, Knoxville
Knoxville, Tennessee, USA
sc Open MPI du

Simon Byrne
California Institute of Technology
Pasadena, California, USA
simonby Julia tech.edu

Hui Zhou
Argonne National Laboratory
Lemont, Illinois, USA
zh MPICH ov

Open Access Paper
https://dl.acm.org/doi/10.1145/3615318.3615319

Current Status

MPICH supports the proposed ABI, as defined in the reference header; tested with mpi4py, etc.

MPI Forum still debating fine details of Fortran support.

As a side effect of the ABI effort, MPICH test suite is implementation-agnostic and can be used to test Open MPI, e.g.

https://github.com/mpiwg-abi/header_and_stub_library/

VAPAA

VAPAA

In Finnish, Vapaa means "free", in the sense of "free-range chickens."

What:

Standalone implementation of MPI Fortran support (**MPI_F08**).

Why:

Workaround Fortran compiler and MPI implementation issues to get all the features everywhere.

How:

Use MPI C API; translate subarrays to datatypes using CFI_cdesc_t.
Use MPICH's MPIX_Type_iov instead of tedious and slow type introspection with MPI API.

When:

Common features are available. Features added based on user interest. Code generation will achieve feature-completeness eventually.

https://github.com/jeffhammond/vapaa

NVIDIA

Accelerating Collective Communication With Error-Bounded Lossy Compression

Jiajun Huang

ANL & UC Riverside

Motivation

- MPI collective -> high-performance -> **significant impact on various research fields.**

- Exascale computing -> large-message MPI collectives -> **Scalability challenges.**
  - VGG19 with 143 million parameters -> communication overhead of 83% [1].
  - ResNet-50 with 25 million parameters -> communication overhead of 72% [1].

- Inter-node communications -> **limited network bandwidth -> major bottleneck.**

- How can we solve this bottleneck?

Motivation

- **Designing large message algorithms:** Decrease the overall communication volume.
  - **Allreduce:** Ring: $\frac{2*(N-1)}{N} * D$ **vs** Recursive-doubling: $\lceil \log N \rceil * D$.
- **Lossy compression**: Significantly reduce the message size.
  - To address this issue, prior research simply applies the off-the-shelf fix-rate lossy compressors in the MPI collectives, leading to **suboptimal performance**, **limited generality**, and **unbounded errors** [2].

Design of C-Coll

- C-Coll (Compression-accelerated Collectives): **(IPDPS 24)**
  - Overlap the compression with communication using our developed pipelined SZx in our collective computation framework.
  - Reduce the compression overhead and mitigate error propagation by choosing the appropriate timing of compression.
- **Mathematical proof**: To prove the error-bounded nature -> We perform an in-depth mathematical analysis to derive the limited impact of error-bounded lossy compression on error propagation.

Theoretical Analysis of Error Propagation for C-Coll

- **Collective data movement framework**:
  - The final error for each data point is within $\hat{\boldsymbol{e}}$, where $\hat{\boldsymbol{e}}$ is the compression error bound.

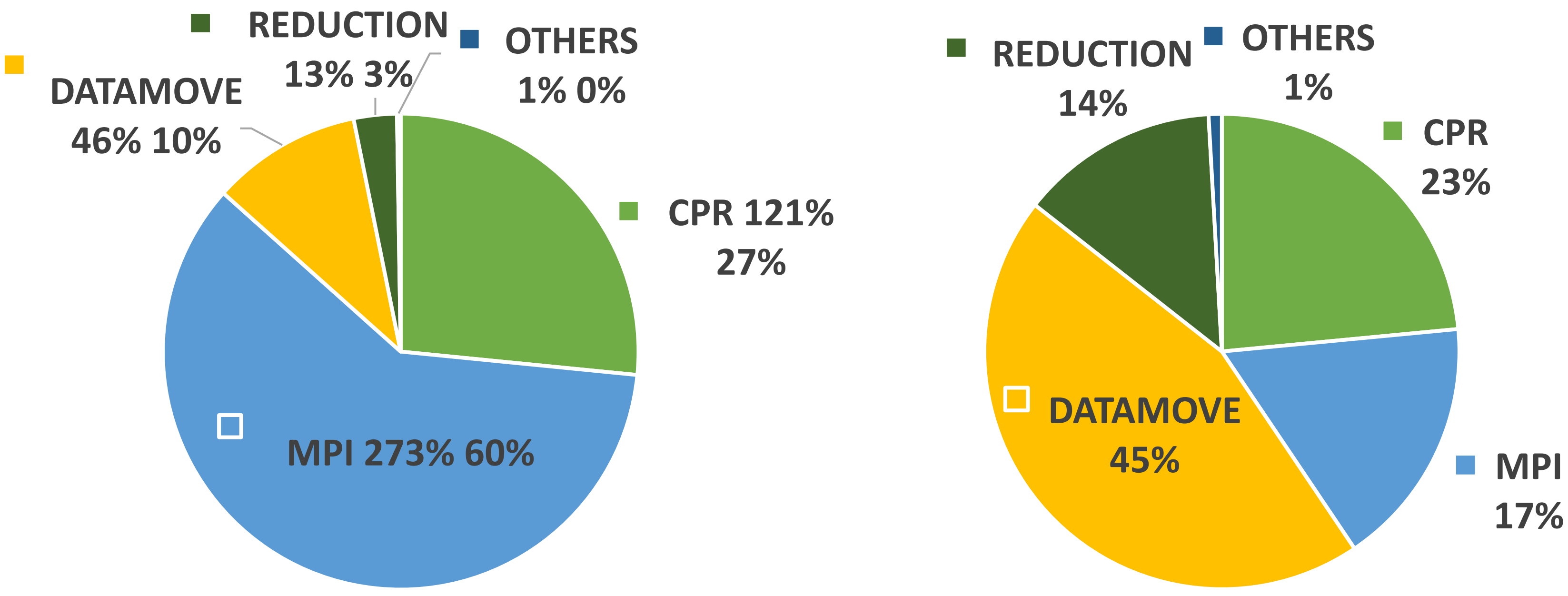
- **Collective computation framework**:
  - The final aggregated error of the most widely used sum operation falls within the interval $[-\frac{\mathbf{2}}{\mathbf{3}}\sqrt{\boldsymbol{n}}\hat{\boldsymbol{e}}, \frac{\mathbf{2}}{\mathbf{3}}\sqrt{\boldsymbol{n}}\hat{\boldsymbol{e}}]$ with the probability of **95.44%**.
  - For example, if there are 100 nodes, and the error bound is 1E-3, the aggregated error is bounded in the range of $[\mathbf{-6.7E-3, 6.7E-3}]$ with a probability of **95.44%**.

Performance of C-Coll

- **Our C-Coll:** a novel design for lossy-compression-integrated MPI collectives that significantly **improves** performance with bounded errors.
  - C-Allreduce is up to 2.1X faster than MPI_Allreduce, while other CPRP2P baselines demonstrate performance degradation.
  - C-Scatter is up to 1.8X faster than MPI_Scatter.
  - C-Bcast is up to 2.7X faster than MPI_Bcast.

Limitation of C-Coll

- C-Coll is optimized for host-centric collective communications. Thus, it faces serious issues in GPU-centric communications [2].
  - Expensive host-device data movements.
  - Underutilized GPU devices.



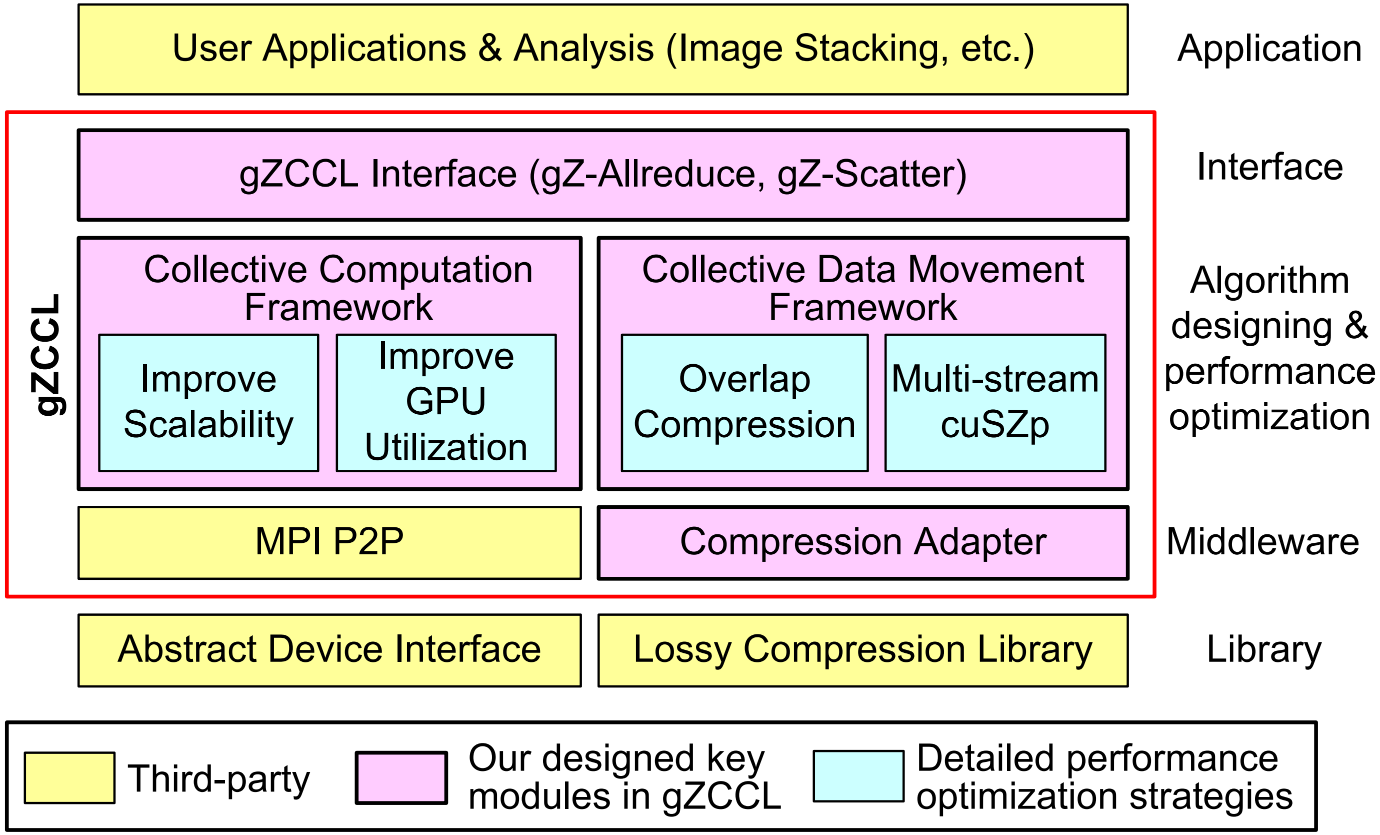
(a) C-Coll

(b) CPRP2P

Figure 1: **Performance breakdown of Allreduce using CPRP2P and C-Coll on 64 A100 GPUs: CPRP2P's first percentage is scaled to C-Coll's runtime, and the second is scaled to its own.**

Design of gZCCL

- gZCCL (GPU-aware Compression-Accelerated Collective Communication Library): **(ICS 24)**
  - Improve the scalability and GPU utilization in the collective computation framework.
  - Overlap compression with our multi-stream cuSZp in the collective data movement framework.



User Applications & Analysis (Image Stacking, etc.)

Application

gZCCL

gZCCL Interface (gZ-Allreduce, gZ-Scatter)

Interface

Collective Computation Framework

Improve Scalability

Improve GPU Utilization

Collective Data Movement Framework

Overlap Compression

Multi-stream cuSZp

Algorithm designing & performance optimization

MPI P2P

Compression Adapter

Middleware

Abstract Device Interface

Lossy Compression Library

Library

🟨 Third-party

🟪 Our designed key modules in gZCCL

🟦 Detailed performance optimization strategies

***Figure 1*: Design architecture (purple box: newly contributed modules)**

Evaluating the Collective Computation Framework of gZCCL

Figure 2 demonstrates that our recursive doubling-based gZ-Allreduce (ReDoub) consistently performs the best, achieving up to **20.2X** and **4.5X** speedups compared to **Cray MPI** and **NCCL** respectively, across varying GPU counts.

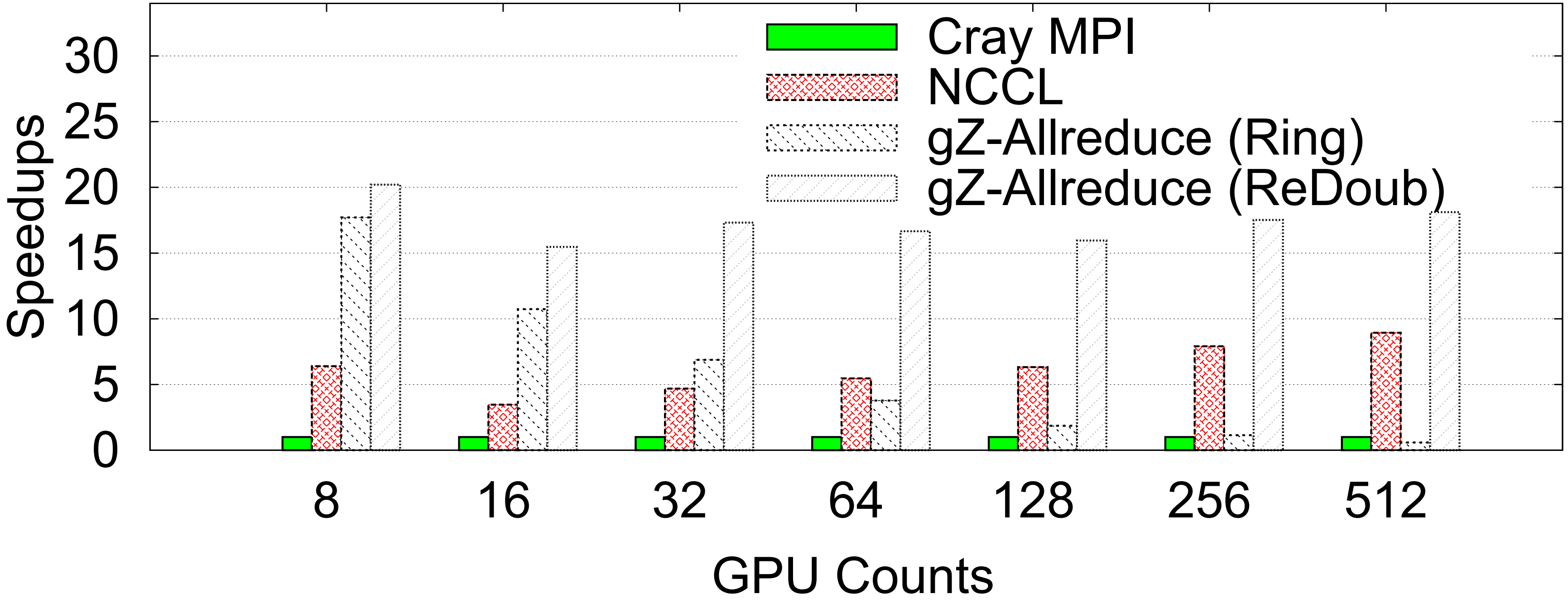
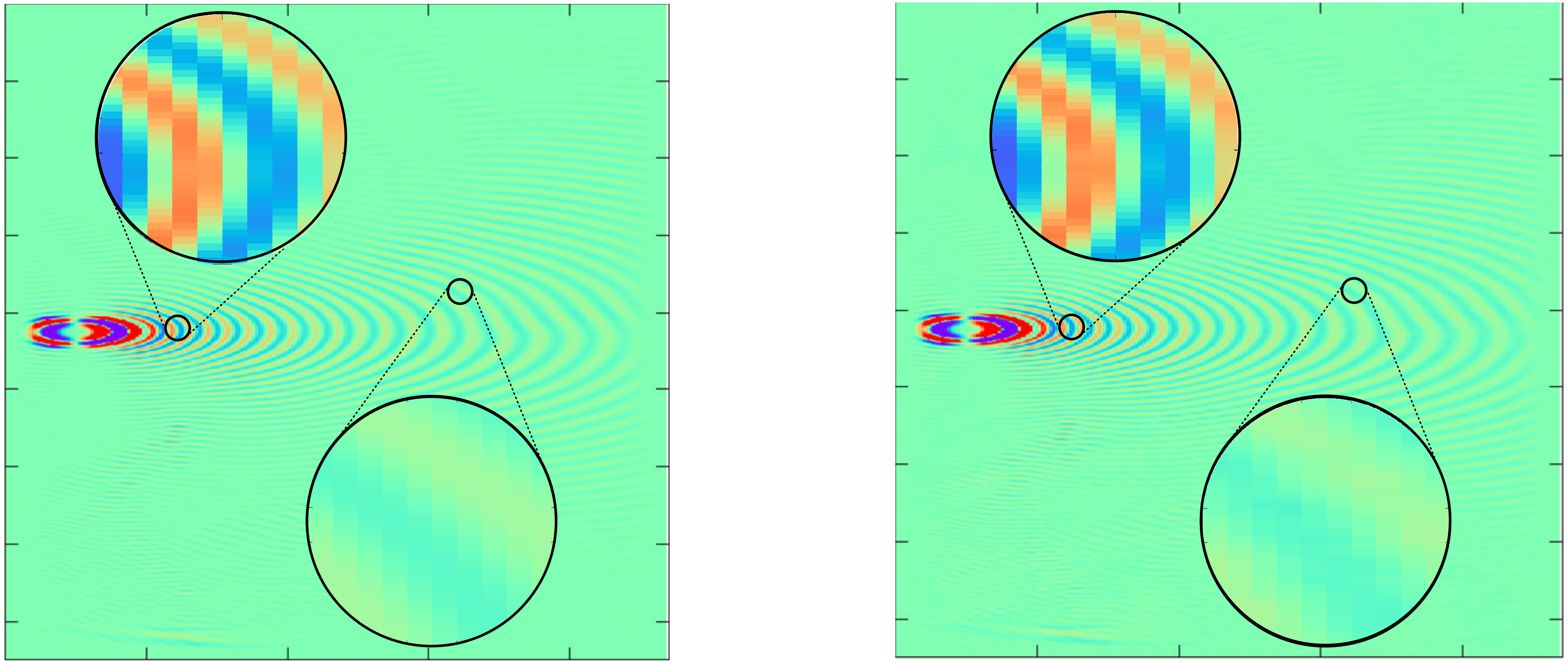


***Figure 2*: Scalability evaluation of our gZ-Allreduce with Cray MPI and NCCL in different GPU counts.**

Image Stacking Accuracy Analysis



(a) Cray MPI/NCCL (lossless)

(b) gZCCL (2E-4)

***Figure 4*: Visualization of final stacking image.**

gZCCL (Ring) (1E-4) reaches a great PSNR of **56.83** and an NRMSE of **1E-3**.

gZCCL (ReDoub) (1E-4) demonstrates better data quality, achieving a PSNR of **57.80** and an NRMSE of **1E-3**.

Summary

- **Performance**: With our C-Coll and gZCCL, compression-accelerated collectives have significantly higher performance than the SOTA communication libraries on both CPU and GPU.

- **Accuracy**: With the accuracy-aware design, C-Coll and gZCCL can demonstrate well-controlled accuracy through both theoretical and experimental analysis.

- **Future Work**: We plan to further optimize compression-accelerated collectives for FPGAs and AI accelerators.

References

1. A. M. Abdelmoniem, A. Elzanaty, M.-S. Alouini, and M. Canini, “An efficient statistical-based gradient compression technique for distributed training systems,” 2021.
2. Jiajun Huang and Sheng Di and Xiaodong Yu and Yujia Zhai and Zhaorui Zhang and Jinyang Liu and Xiaoyi Lu and Ken Raffenetti and Hui Zhou and Kai Zhao and Zizhong Chen and Franck Cappello and Yanfei Guo and Rajeev Thakur. 2023. An Optimized Error-controlled MPI Collective Framework Integrated with Lossy Compression. arXiv:2304.03890 [cs.DC]
3. Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. 2023. CuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23).

Questions?

Thanks for your attention!

C-Coll Overall system design architecture

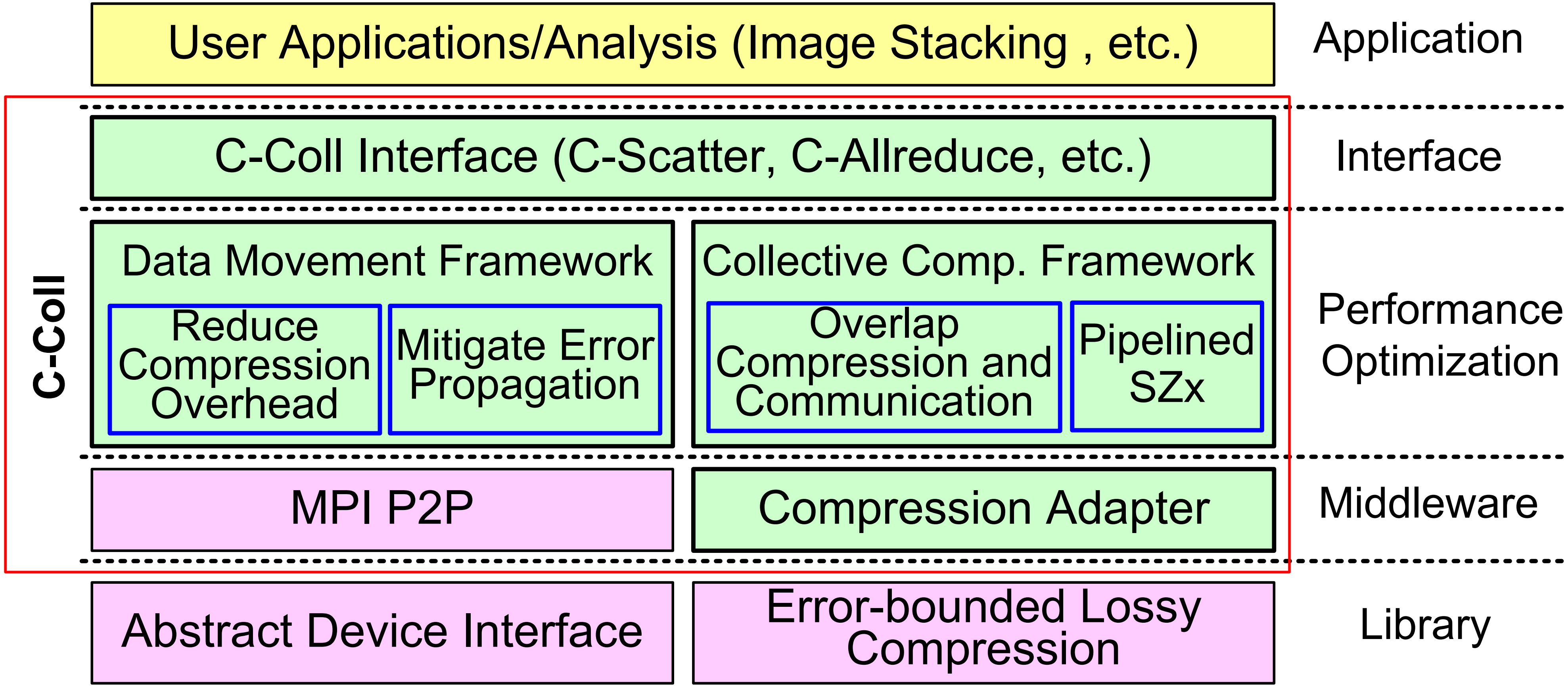
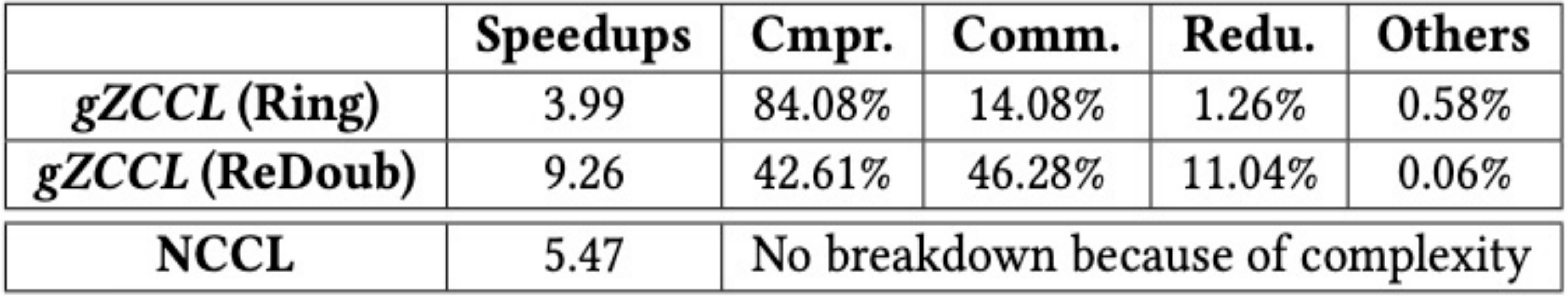

User Applications/Analysis (Image Stacking , etc.)
Application
C-Coll
C-Coll Interface (C-Scatter, C-Allreduce, etc.)
Interface
Data Movement Framework
Reduce Compression Overhead
Mitigate Error Propagation
Collective Comp. Framework
Overlap Compression and Communication
Pipelined SZx
Performance Optimization
MPI P2P
Compression Adapter
Middleware
Abstract Device Interface
Error-bounded Lossy Compression
Library

***Figure 1*: Design architecture (yellow box: applications; green box: new contributed modules; purple box: third-party)**

Image Stacking Performance Evaluation

***Table 1*: Image stacking performance analysis (The speedups are based on Cray MPI. The last four columns are performance breakdowns of our gZCCL).**



	Speedups	Cmpr.	Comm.	Redu.	Others
gZCCL (Ring)	3.99	84.08%	14.08%	1.26%	0.58%
gZCCL (ReDoub)	9.26	42.61%	46.28%	11.04%	0.06%
NCCL	5.47	No breakdown because of complexity			

Evaluating Our Collective Data Movement Framework (gZCCL)

Figure 3 shows that our gZ-Scatter outperforms Cray MPI in all cases. As the GPU count increases, the speedup of gZ-Scatter first increases, peaking at **28.7X**, and then gradually decreases to **4.75X** when the GPU count reaches 512.

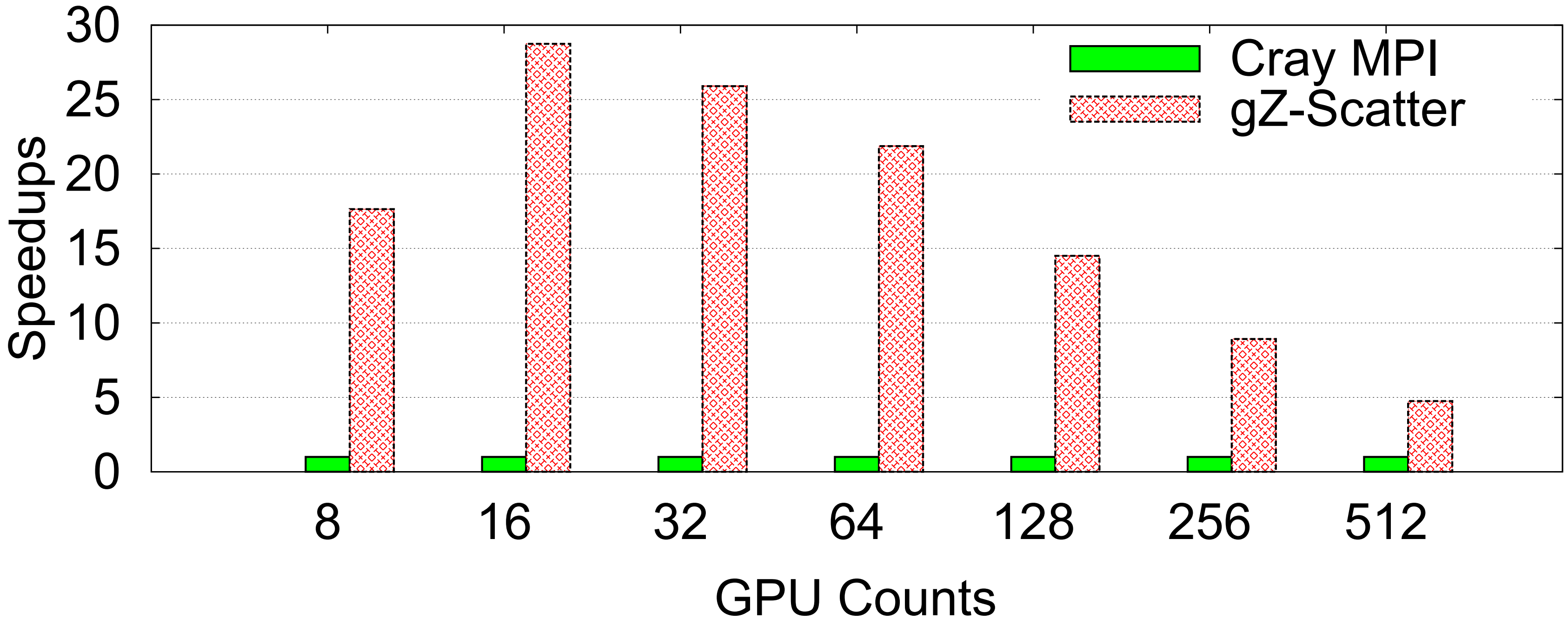


***Figure 3*: Scalability evaluation of our gZ-Scatter with Cray MPI in different GPU counts.**

Evaluating Our Collective Computation Framework

Figure 3 shows that our recursive doubling-based gZ-Allreduce (ReDoub) consistently outperforms across all data sizes, achieving up to a speedup of **18.7X** compared to **Cray MPI** and a **3.4X** performance improvement over **NCCL**.

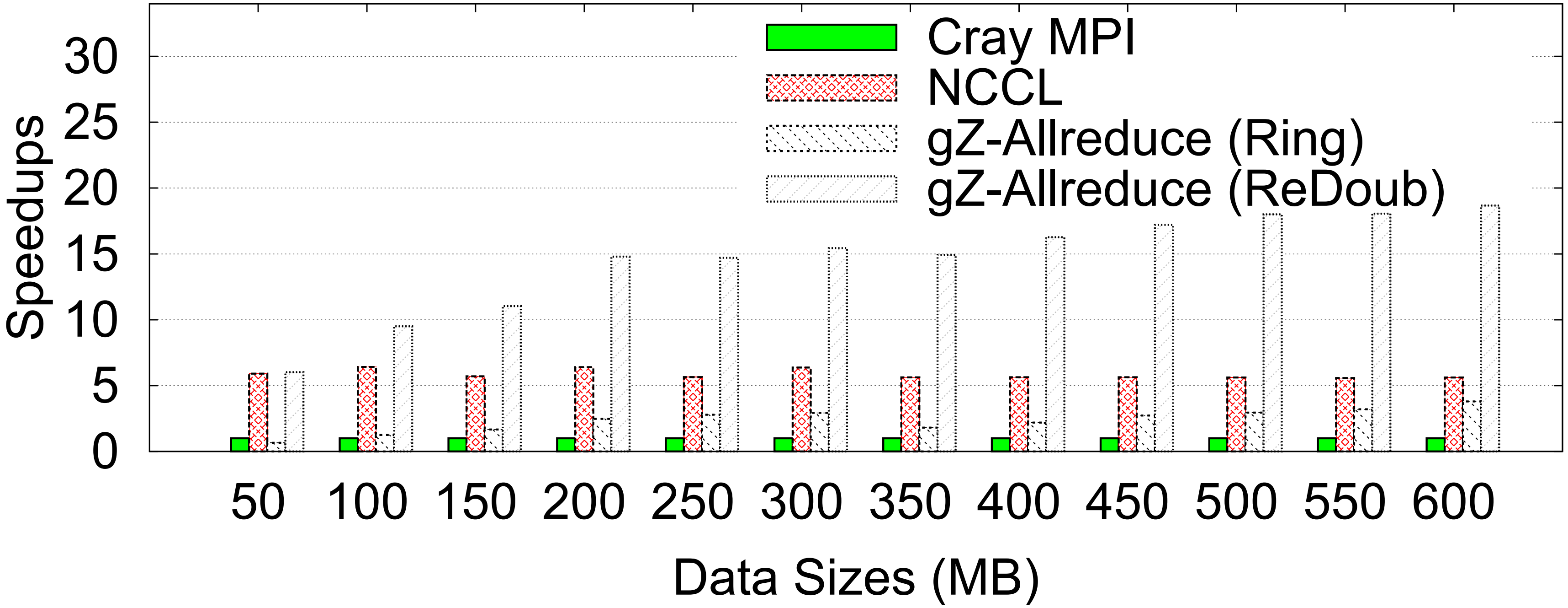


***Figure 3*: Performance evaluation of our gZ-Allreduce with Cray MPI and NCCL in different data sizes.**

Evaluating Our Collective Data Movement Framework

Figure 5 indicates that our gZ-Scatter is able to consistently outperform Cray MPI across all data sizes. The speedup of gZ-Scatter enhances as the data size increases, achieving its maximum **20.2X** at 600 MB.

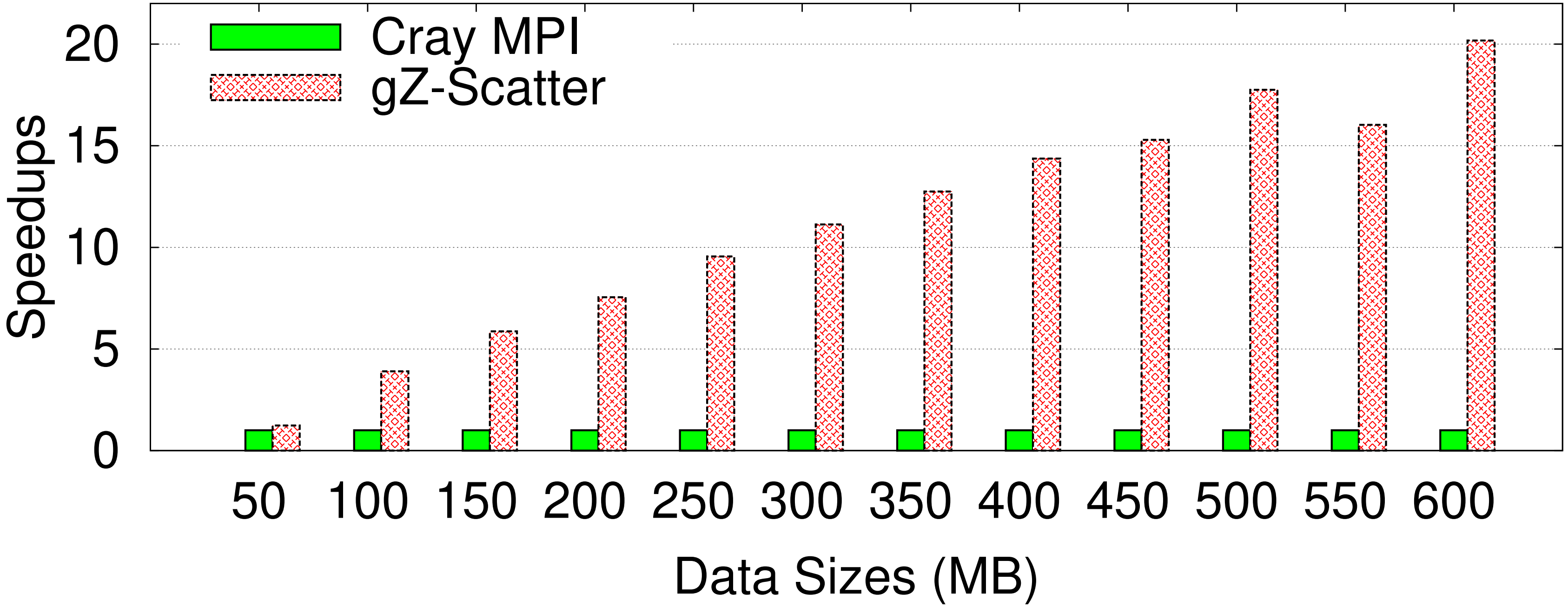


***Figure 5*: Performance evaluation of our gZ-Scatter with Cray MPI in different data sizes.**

About me

- **Two teams in Argonne**:
  - Yanfei Guo and Rajeev Thakur in the **MPICH** project, which is one of the most widely-used MPI libraries.
  - Sheng Di and Franck Cappello in the **SZ** project, which is the leading lossy compressor framework.

ECP

R&D 100™

SZ

2021
R&D 100
WINNER

Experiences with ROMIO

from a PnetCDF developer’s point of view

Wei-Keng Liao, ECE Department, Northwestern University

MPICH: A High-Performance Open Source MPI Library for
Leadership-class HPC Systems
2024 CASS Community BOF Days, June 12, 2024

Northwestern | McCORMICK SCHOOL OF ENGINEERING

PnetCDF

- A parallel I/O library for accessing NetCDF classic file format
  - Relies on MPI-IO
- NetCDF classic file format
  - NetCDF files are popularly used in climate research community
  - Header and data sections
  - Data objects: dimensions, variables, attributes
- PnetCDF APIs
  - Metadata
  - Read and write subarrays of variables
  - Blocking and non-blocking reads and writes

MPI APIs used in PnetCDF

- MPI-IO
  - File open, close, seek, sync, set view
  - Collective and independent I/O APIs
- MPI communication
  - Metadata consistency check: file header
  - Mostly MPI_Allreduce, MPI_Bcast
- MPI derived datatypes to define fileview
  - Commonly used data partitioning patterns
  - Call to set fileview is collective, a problem for switch between independent and collective modes

I/O request aggregation

- User intent for I/O
  - Saving multiple variables in the file during data checkpointing.
  - Safely store all variables, not individual variables, before returning to the computation phase.
  - Such intent can be better realized by high-level libraries (through requestion aggregation feature in PnetCDF, HDF5, and PIO)
- PnetCDF
  - Use nonblocking APIs to post requests + a wait_all call later to flushed all out
- HDF5 multi-dataset APIs
  - H5Dwrite_multi — allows a single call to write multiple variables
- PIO
  - Two aggregation options: subset and box rearrangers
- Challenge for MPI-IO
  - Aggregated amount can become very large. So is the metadata describing the requests.

Challenges — I/O hints

- cb_nodes
  - Number of I/O aggregators (a subset of processes does I/O)
  - Default: one per compute node (GPFS), same as number of OSTs (Lustre)
- cb_buffer_size (default: 16 MiB)
- striping_factor
  - Too small yields poor performance, too large increases interference from other users
- striping_unit
  - Large, contiguous requests prefer large striping size
- Data sieving (romio_ds_write, romio_ds_read, romio_cb_ds_threshold)
  - I/O aggregators check holes in its file domain to determine whether data sieving is necessary, and then read-modify-write

Challenges — future ROMIO improvement

- Memory footprints
  - Fileview and user buffer datatype are flattened into offset-length pairs
  - Internal buffers allocated for storing these pairs can become significant
- Excessive number of memcpy calls
  - Communication phase in collective I/O packs data into contiguous buffers before sending
  - For large number of offset-length pairs, memcpy calls become expensive
- Communication in the two-phase I/O
  - MPI_Isend vs. MPI_Issend — MPI_Issend can prevent message queues from being overwhelmed

MPI on Aurora and Sunspot:

Examples and Best Practices

Vitali Morozov (morozov@anl.gov)

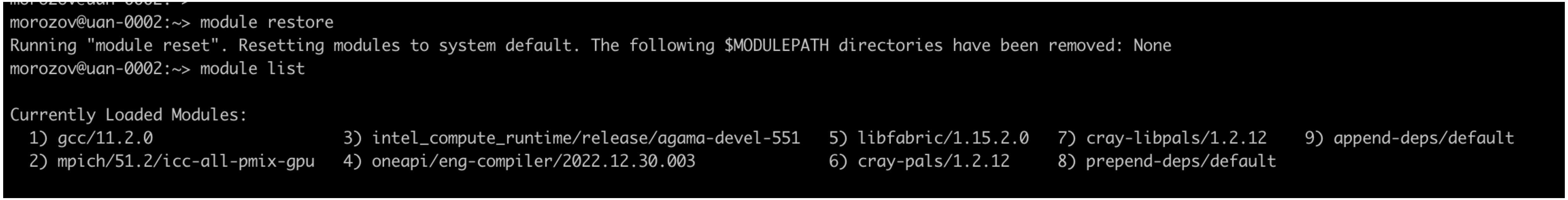
Performance Engineering Team, Argonne Leadership Computing Facility

Consortium for the Advancement of Scientific Software

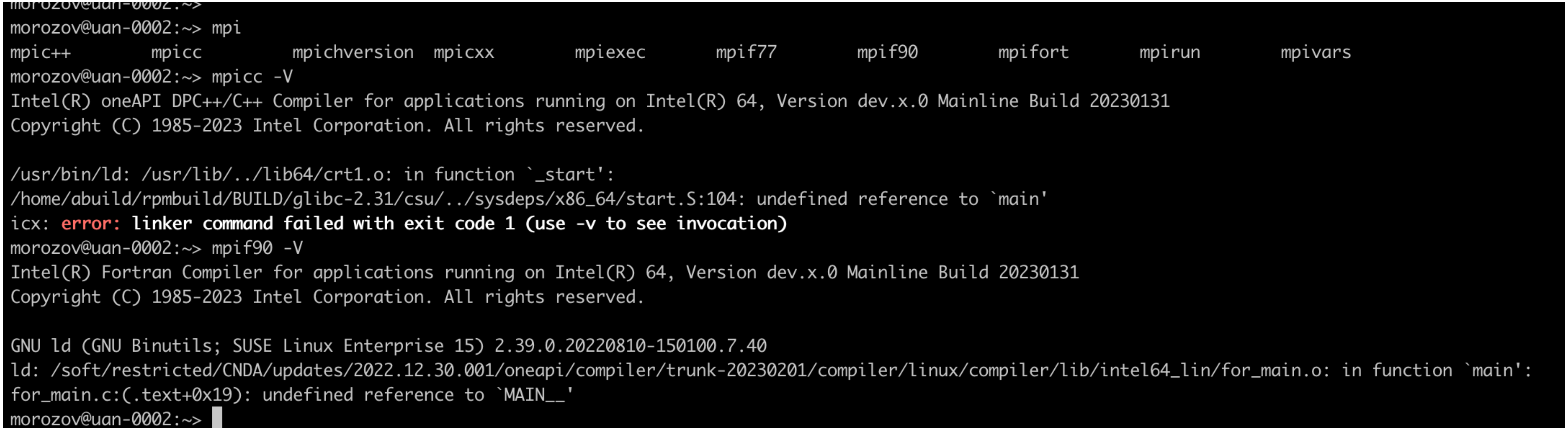
MPICH Birds of a Feather

June 12, 2024

How to Get Started



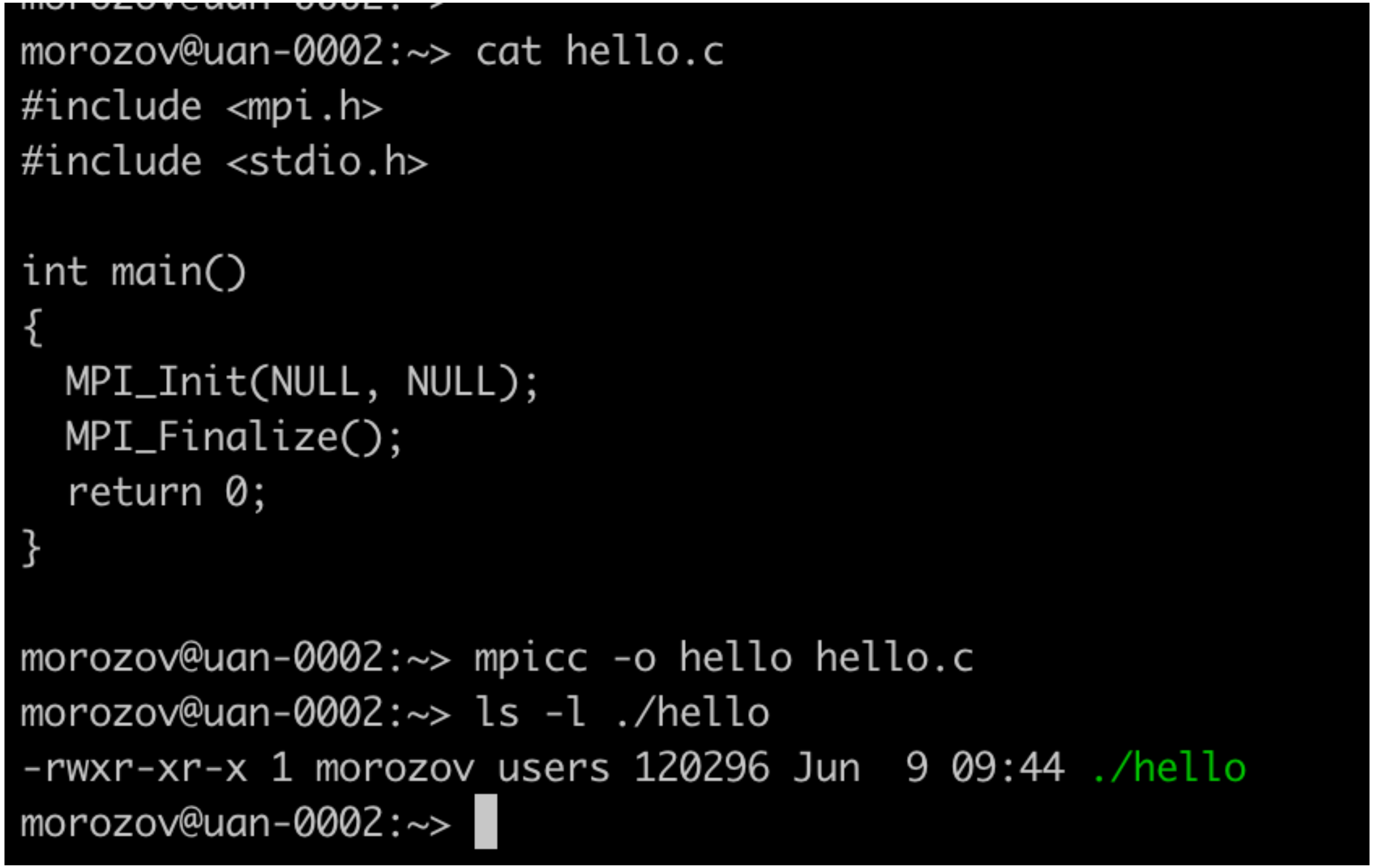
```
morozov@uan-0002:~>
morozov@uan-0002:~> module restore
Running "module reset". Resetting modules to system default. The following $MODULEPATH directories have been removed: None
morozov@uan-0002:~> module list

Currently Loaded Modules:
  1) gcc/11.2.0                 3) intel_compute_runtime/release/agama-devel-551   5) libfabric/1.15.2.0   7) cray-libpals/1.2.12    9) append-deps/default
  2) mpich/51.2/icc-all-pmix-gpu   4) oneapi/eng-compiler/2022.12.30.003           6) cray-pals/1.2.12     8) prepend-deps/default
```



```
morozov@uan-0002:~>
morozov@uan-0002:~> mpi
mpic++        mpicc         mpichversion  mpicxx        mpiexec       mpif77        mpif90        mpifort       mpirun        mpivars
morozov@uan-0002:~> mpicc -V
Intel(R) oneAPI DPC++/C++ Compiler for applications running on Intel(R) 64, Version dev.x.0 Mainline Build 20230131
Copyright (C) 1985-2023 Intel Corporation. All rights reserved.

/usr/bin/ld: /usr/lib/../lib64/crt1.o: in function `_start':
/home/abuild/rpmbuild/BUILD/glibc-2.31/csu/../sysdeps/x86_64/start.S:104: undefined reference to `main'
icx: error: linker command failed with exit code 1 (use -v to see invocation)
morozov@uan-0002:~> mpif90 -V
Intel(R) Fortran Compiler for applications running on Intel(R) 64, Version dev.x.0 Mainline Build 20230131
Copyright (C) 1985-2023 Intel Corporation. All rights reserved.

GNU ld (GNU Binutils; SUSE Linux Enterprise 15) 2.39.0.20220810-150100.7.40
ld: /soft/restricted/CNDA/updates/2022.12.30.001/oneapi/compiler/trunk-20230201/compiler/linux/compiler/lib/intel64_lin/for_main.o: in function `main':
for_main.c:(.text+0x19): undefined reference to `MAIN__'
morozov@uan-0002:~>
```

Key points: module restore, gcc-11, agama driver, oneapi, mpich, mpicc, mpic++, mpif90

Do Sanity Check: Aurora and Sunspot are early machines



```
morozov@uan-0002:~>
morozov@uan-0002:~> cat hello.c
#include <mpi.h>
#include <stdio.h>

int main()
{
  MPI_Init(NULL, NULL);
  MPI_Finalize();
  return 0;
}

morozov@uan-0002:~> mpicc -o hello hello.c
morozov@uan-0002:~> ls -l ./hello
-rwxr-xr-x 1 morozov users 120296 Jun  9 09:44 ./hello
morozov@uan-0002:~> 
```

Key points: make your own tests, identify critical dependencies, test them after each update

Know the topology of the node: Sockets and Cores

- ❑ ***Dual socket with 52 physical Cores on each socket***

  HyperThreading makes each Core as 2 CPUs

- ❑ ***MPI numerates CPUs, not Cores***

```
CPU0 is Socket0, Core0, Thread0
CPU1 is Socket0, Core1, Thread0
….
CPU51 is Socket0, Core51, Thread0
CPU52 is Socket1, Core0, Thread0
CPU53 is Socket1, Core1, Thread0
…
CPU103 is Socket1, Core51, Thread0
CPU104 is Socket0, Core0, Thread1
CPU105 is Socket0, Core1, Thread1
…
CPU155 is Socket0, Core51, Thread1
CPU156 is Socket1, Core0, Thread1
CPU157 is Socket1, Core1, Thread1
…
CPU206 is Socket1, Core50, Thread1
CPU207 is Socket1, Core51, Thread1
```

Key points:

Core changes first, 0 to 51

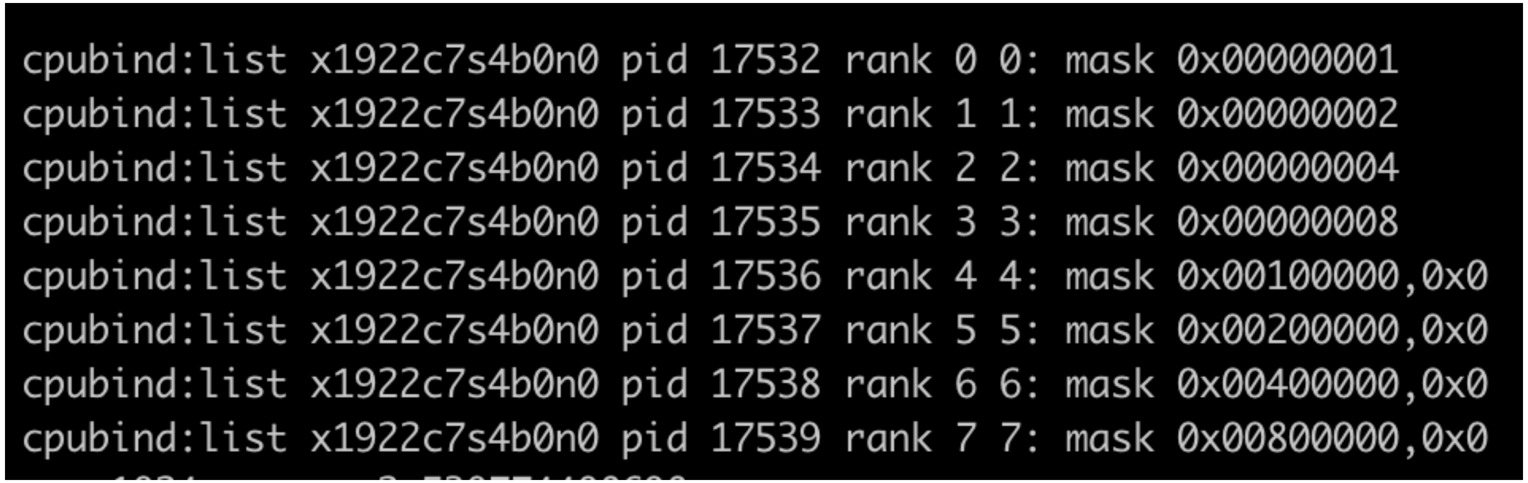
Socket changes next, 0 to 1

Thread changes last, 0 to 1

Use explicit placement and verbose

Thanks Taru Doodi, Intel!

```
mpiexec --np 8 -ppn 8 --cpu-bind verbose,list:0:1:2:3:52:53:54:55 <exe_file>
```



```
cpubind:list x1922c7s4b0n0 pid 17532 rank 0 0: mask 0x00000001
cpubind:list x1922c7s4b0n0 pid 17533 rank 1 1: mask 0x00000002
cpubind:list x1922c7s4b0n0 pid 17534 rank 2 2: mask 0x00000004
cpubind:list x1922c7s4b0n0 pid 17535 rank 3 3: mask 0x00000008
cpubind:list x1922c7s4b0n0 pid 17536 rank 4 4: mask 0x00100000,0x0
cpubind:list x1922c7s4b0n0 pid 17537 rank 5 5: mask 0x00200000,0x0
cpubind:list x1922c7s4b0n0 pid 17538 rank 6 6: mask 0x00400000,0x0
cpubind:list x1922c7s4b0n0 pid 17539 rank 7 7: mask 0x00800000,0x0
```

Key points:

Check the placement

Do not rely on defaults

- ✓ Mask represents cores a task is assigned to. Cores are numerated from 0 to 207
- ✓ Mask is hexadecimal, numerating cores right-to-left
- ✓ One mask digit represents 4 cores, from 0000$_b$ (no cores) to F = 1111$_b$ (all 4 cores)
- ✓ Mask 0x00000001 is 1$_b$, represents Core0
- ✓ Mask 0x00000002 is 10$_b$, represents Core1
- ✓ Obviously, Mask 0x00000003 is 11$_b$, represents Core0 and Core1
- ✓ 0x0 means 0x00000000 no cores from 32 core pool. 0xFFFFFFFF is all 32 cores
- ✓ Example: 0x105,,0x5 means: cores 101b from first 32 core pool, 32 cores empty, cores 1$_b$ 0000$_b$ 0101$_b$ from third pool or Cores 0, 2, 65, 67, 73

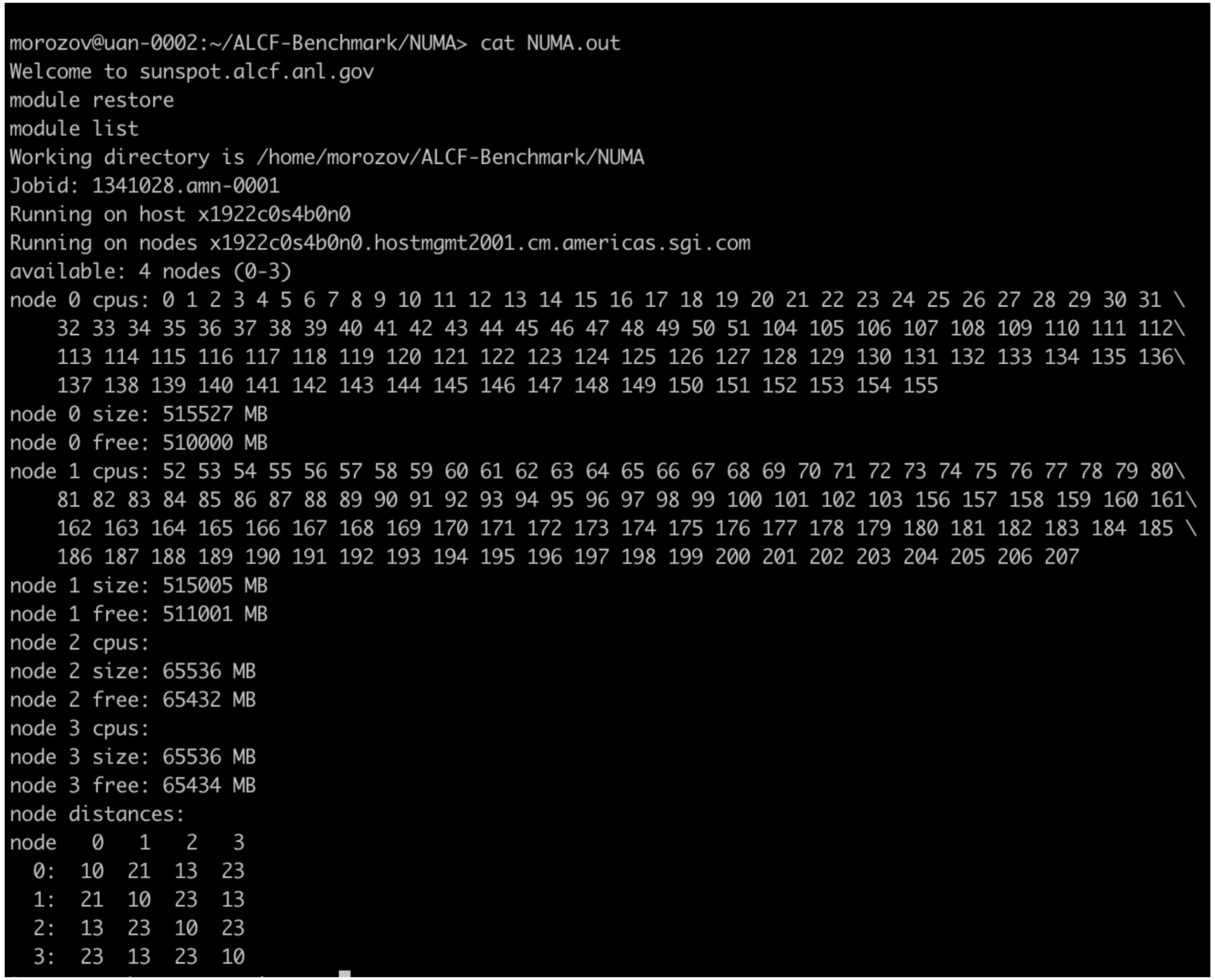
Know the NIC assignments

- ✓ 4 NICs on a socket, 8 NICs on a node
- ✓ Round-robin MPI process to NIC assignment on a socket
- ✓ MPI process does not use multiple NICs*
  - Link bandwidth might be a limitation
- ✓ MPI processes may share the NICs
  - Running 5 or more processes per socket
- ✓ Sockets are connected by UPI bus – might be a limitation
- ✓ Cores are attached to a Network-on-a-chip – might be a limitation

Key points: Distribute processes across cores and sockets to maximize hardware utilization

*** Default, may be changed in experimental builds**

Know the memory domains – Flat mode



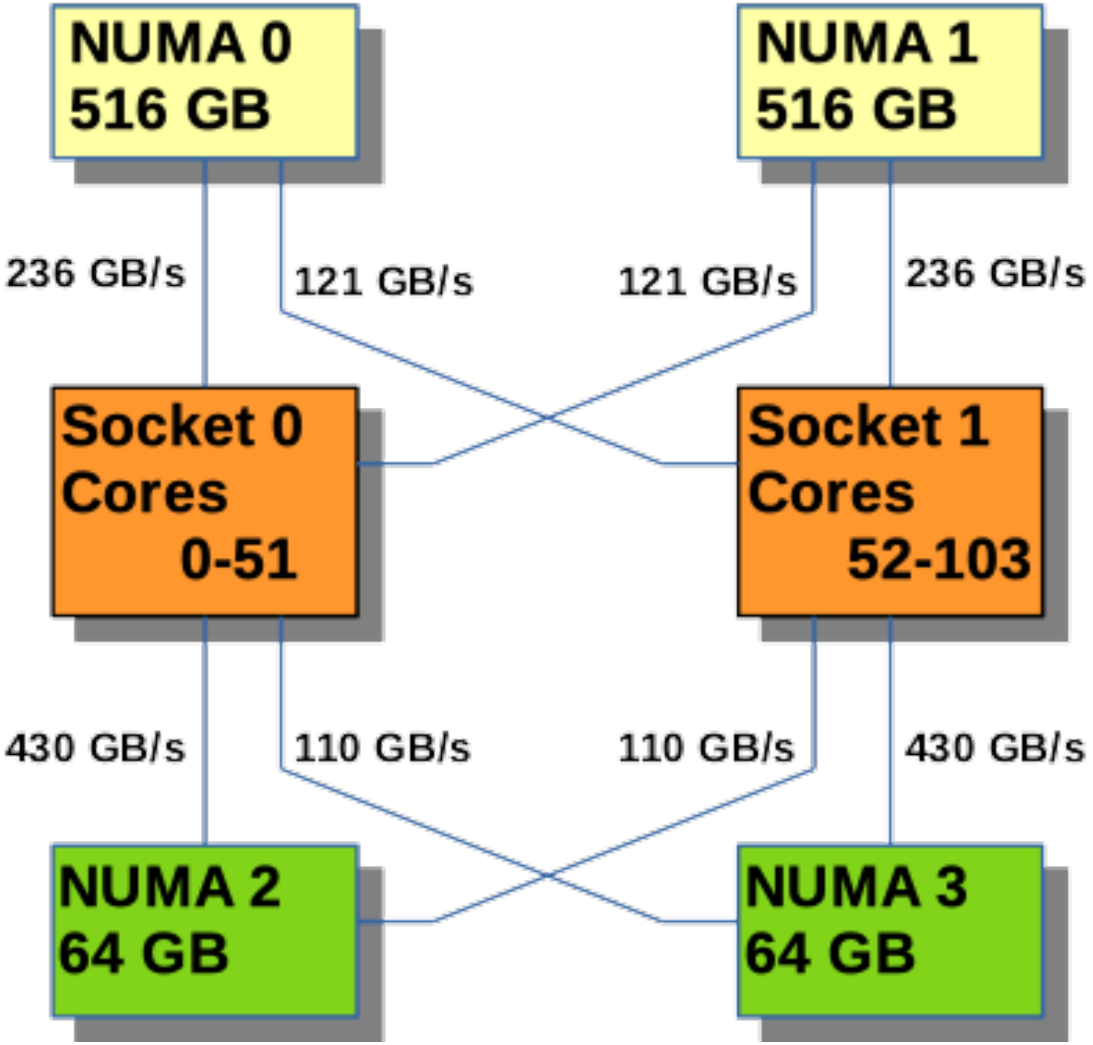
```
morozov@uan-0002:~/ALCF-Benchmark/NUMA> cat NUMA.out
Welcome to sunspot.alcf.anl.gov
module restore
module list
Working directory is /home/morozov/ALCF-Benchmark/NUMA
Jobid: 1341028.amn-0001
Running on host x1922c0s4b0n0
Running on nodes x1922c0s4b0n0.hostmgmt2001.cm.americas.sgi.com
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 \
    32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 104 105 106 107 108 109 110 111 112\
    113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136\
    137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155
node 0 size: 515527 MB
node 0 free: 510000 MB
node 1 cpus: 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80\
    81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 156 157 158 159 160 161\
    162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 \
    186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
node 1 size: 515005 MB
node 1 free: 511001 MB
node 2 cpus:
node 2 size: 65536 MB
node 2 free: 65432 MB
node 3 cpus:
node 3 size: 65536 MB
node 3 free: 65434 MB
node distances:
node   0   1   2   3
  0:  10  21  13  23
  1:  21  10  23  13
  2:  13  23  10  23
  3:  23  13  23  10
```

- ✓ numactl -H before mpiexec
- ✓ 512GB DDR per socket
- ✓ 64GB HBM nodes
- ✓ NUMA domains defined by mode
- ✓ DDR nodes have cores
- ✓ numactl -m 2-3 ./app
- ✓ memkind

```
#include <hbwmalloc.h>
void *hbwmalloc( size_t );
void hbw_free( void *ptr );
-I/home/morozov/include
-L/home/morozov/lib -lmemkind
```

Key points: Meet the system config with your application config

Know the memory domains – Flat mode



- ✓ numactl -H before mpiexec
- ✓ 512GB DDR per socket
- ✓ 64GB HBM nodes
- ✓ NUMA domains defined by mode
- ✓ DDR nodes have cores
- ✓ numactl -m 2-3 ./app
- ✓ memkind

Key points: Meet the system config with your application config

Know the GPU mode and numeration

- ✓ Socket0 has GPU0, GPU1, and GPU2
- ✓ Socket1 has GPU3, GPU4, and GPU5
- ✓ ZE_AFFINITY_MASK variable defines visibility
- ✓ Visibility is defined ”per MPI process”

  Different processes may define different masks

  Use PALS_RANKID, PALS_LOCAL_RANKID with getenv call

  Use MPI_COMM_TYPE_SHARED for comm_split for MPI compliance
- ✓ ZE_AFFINITY_MASK=4 # processes uses 1 device, GPU4
- ✓ ZE_AFFINITY_MASK=0,3 # 2 devices, 0 and 3, different sockets
- ✓ ZE_AFFINITY_MASK=0.0,1.0,2.0,3.0,4.0,5.0å

Key point: Use explicit binding of GPUs to tasks

Local Environment

✓ PALS-defined variables – Cray’s Parallel Application Launch service provided

PALS_RANKID – job process ID, PALS_LOCAL_RANKID – node process ID

```
char *ptr = getenv("PALS_LOCAL_RANKID");
if ( ptr != NULL ) local_id = atoi( ptr );
```

✓ MPI standard for shared memory region

MPI_COMM_TYPE_SHARED predefined

```
MPI_Comm_split_type( MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &shmcomm );
MPI_Comm_size( shmcomm, &shmsize );    // Number of ranks per node
MPI_Comm_rank( shmcomm, &shmrank );    // My ID in a node, might be similar to PALS_LOCAL_RANKID
```

Key point: Use local environment setup constructs to identify itself

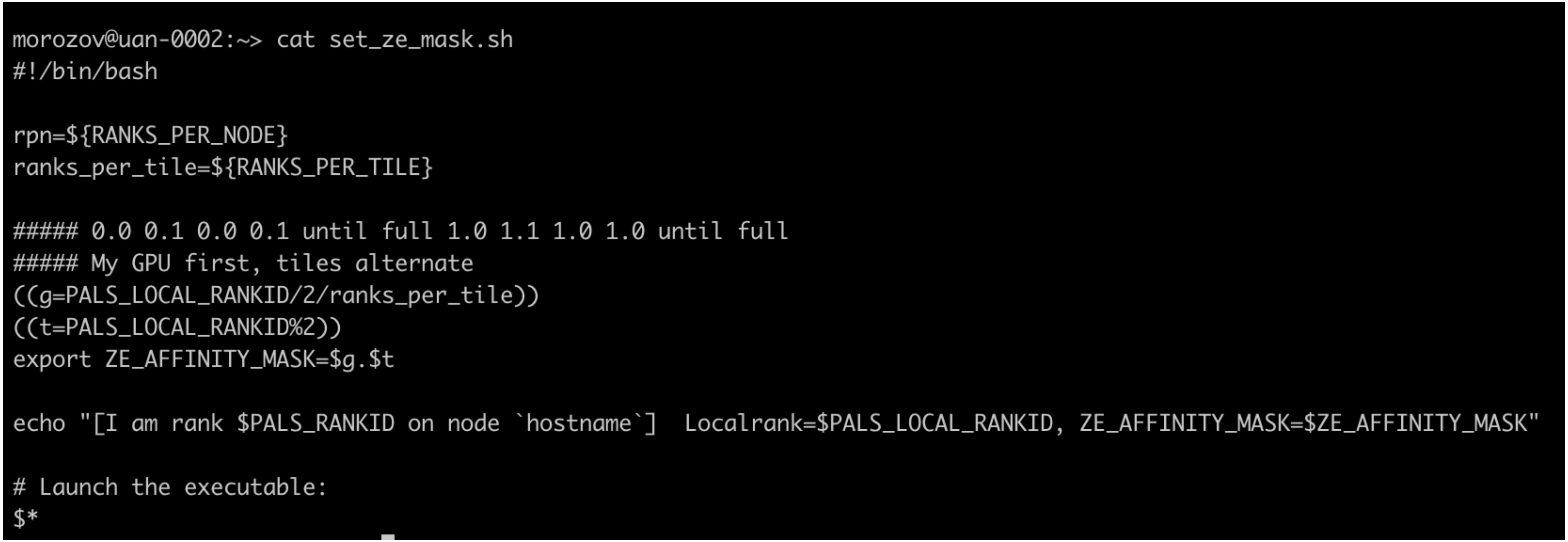
OMP example: Assigning a device

```c
#include <mpi.h>
#include <omp.h>
#include <stdio.h>
int main( int argc, char * argv[] ) {
    int rank, nsize, res_len, shmrank, shmsize, numdevices, deviceid;
    char name[ MPI_MAX_PROCESSOR_NAME ];    MPI_Comm shmcomm;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nsize );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank ) ;
    MPI_Get_processor_name( &name[0], &res_len );
    MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED,
        0, MPI_INFO_NULL, &shmcomm);
    MPI_Comm_size( shmcomm, &shmsize );   // Ranks per node
    MPI_Comm_rank( shmcomm, &shmrank );   // Local rank ID
    numdevices = omp_get_num_devices();
    deviceid = shmrank % numdevices;
    omp_set_default_device( deviceid );
    printf( "Hello from rank %3d of %4d, I am rank %3d of node ranks
        %3d, I know %2d devices, my device id %2d: node %s\n",
        rank, nsize, shmrank, shmsize, numdevices, deviceid, name );
    MPI_Finalize();
    return 0;
}
```

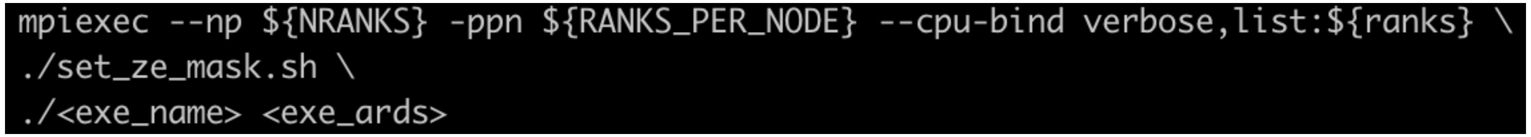
```
mpicc -O2 -g -fiopenmp -fopenmp-targets=spir64  -c hello.c -o hello.o
mpicc -O2 -g -fiopenmp -fopenmp-targets=spir64  hello.o -o hello
```

```
module restore
module list
echo Working directory is $PBS_O_WORKDIR
cd $PBS_O_WORKDIR
echo Jobid: $PBS_JOBID
echo Running on host `hostname`
echo Running on nodes `cat $PBS_NODEFILE`
NNODES=`wc -l < $PBS_NODEFILE`
NRANKS=4            # Number of MPI ranks per node
NTOTRANKS=$(( NNODES * NRANKS ))

echo "NUM_OF_NODES=${NNODES}  TOTAL_NUM_RANKS=${NTOTRANKS}
RANKS_PER_NODE=${NRANKS}

export LIBOMPTARGET_DEVICES=SUBDEVICE  # DEVICE | SUBDEVICE
export LIBOMPTARGET_PLUGIN=LEVEL0

mpiexec --np ${NTOTRANKS} -ppn ${NRANKS} \
    --cpu-bind=verbose,list:0:34:52:86 ./hello
```

Example: MPI process to GPU binding



```
morozov@uan-0002:~> cat set_ze_mask.sh
#!/bin/bash

rpn=${RANKS_PER_NODE}
ranks_per_tile=${RANKS_PER_TILE}

##### 0.0 0.1 0.0 0.1 until full 1.0 1.1 1.0 1.0 until full
##### My GPU first, tiles alternate
((g=PALS_LOCAL_RANKID/2/ranks_per_tile))
((t=PALS_LOCAL_RANKID%2))
export ZE_AFFINITY_MASK=$g.$t

echo "[I am rank $PALS_RANKID on node `hostname`]  Localrank=$PALS_LOCAL_RANKID, ZE_AFFINITY_MASK=$ZE_AFFINITY_MASK"

# Launch the executable:
$*
```



```
mpiexec --np ${NRANKS} -ppn ${RANKS_PER_NODE} --cpu-bind verbose,list:${ranks} \
./set_ze_mask.sh \
./<exe_name> <exe_ards>
```

Key point:
Each task should only see the device it uses

MPI process to GPU binding: to be implemented…

mpiexec **--rankfile <file.txt>** ~~-ppn N~~ --cpu-bind verbose,~~list:0:26:52:78~~ … ~~./set_ze_affinity. sh~~ <exe>

or

export PALS_RANKFILE=<file.txt>

file.txt: examples

```
0 0 0
1 0 5
2 1 12
3 1 78
cycle
```

```
0 0 0 0,2
1 0 5 1,3
2 1 12 2
3 1 78 5
cycle
```

```
0 0 0 0.0,0.1
1 0 5 1.0
2 1 12 2.0,3.1
3 1 78 2.1
cycle
```

Format:

Each line corresponds to one rank
Ranks are sorted in order
The first column - rank number
The second column – host index
The third column – CPUs, the rank should be bound to
The fourth column – GPUs, the rank should be bound to
Cycle – the pattern is replicated to the rest of the hosts

Status as of Aug 30, 2023: Partially implemented
We will update users when this functionality is available

GPU-aware MPI by default

- ✓ MPIR_CVAR_ENABLE_GPU=1 # put 0 if you do not need it
- ✓ MPI message can be located in device memory
- ✓ MPI uses XeLink if necessary
- ✓ MPI uses optimized protocols
  - ○ When implemented and expected to be beneficial
- ✓ Has a price associated with it
  - ○ MPI_Init takes longer
- ✓ Latency for DDR-located messages might by higher

Key point:

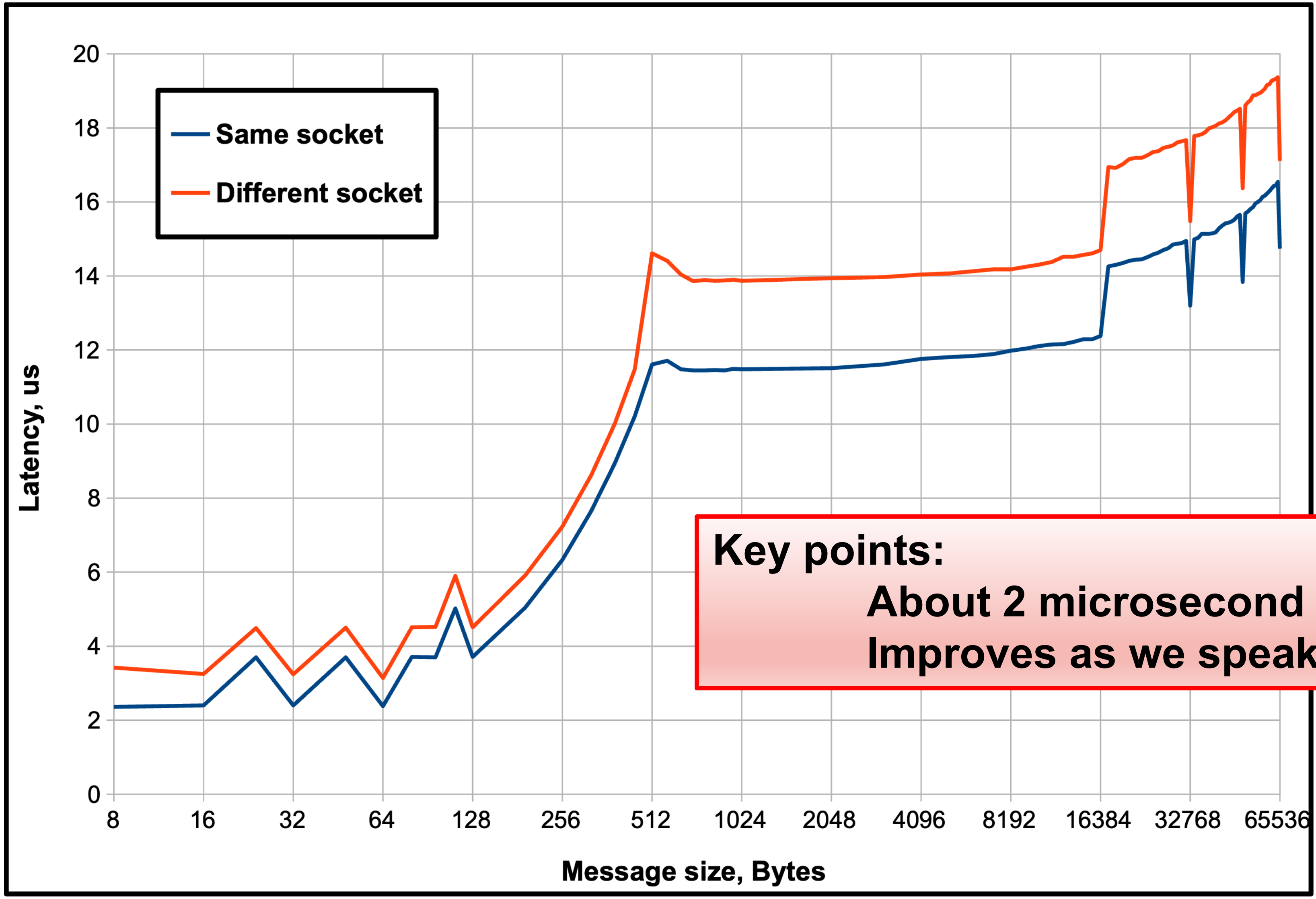
Make the right decision when to use it

Sunspot: Significant intranode improvements

- ✓ Milestones 16, 18 NRE report from Compiler working group
- ✓ Level Zero is used internally for accessing device memory
- ✓ XeLinks are used internally for GPU-GPU transfers
- ✓ CVAR variables for extra tuning
  `$ROOT/share/doc/mpich/tuning_parameters.md`
  - ➢ IPC mechanisms for USM device memory
  - ➢ Fast memory copying control
  - ➢ Pipelining for intern-node communications
  - ➢ The use of compute and link engines
  - ➢ Others

Key point:
Use fine tuning with the CVARs when needed

Benchmarks: GPU to GPU on a node



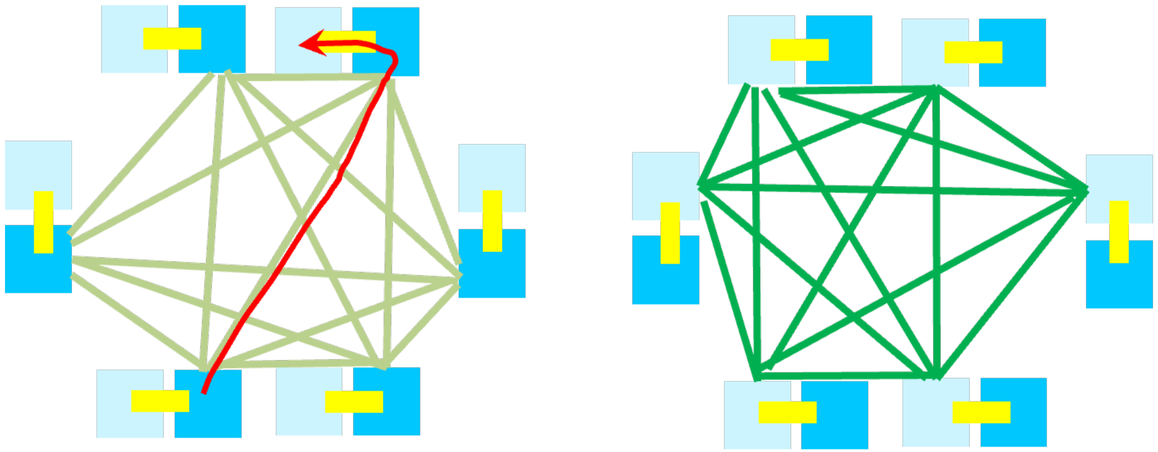
Key points:

About 2 microsecond cost for UPI

Improves as we speak

Benchmarks: GPU to GPU on a node

```
Welcome to sunspot.alcf.anl.gov
Working directory is /home/morozov/OSU_MPI_IntelZE-runs/OSU_pt2pt_1n_bw.Z
Jobid: 4394.amn-0001
Running on host x1921c0s6b0n0

Intra-node: Core 0-25 device 0 -> Core 26-33 device 1
Rank 0/2, device_id = 0/6
Rank 1/2, device_id = 1/6
# OSU MPI-ZE Bandwidth Test v5.6.2
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
# Size      Bandwidth (MB/s)
1024                   49.83
2048                  174.80
4096                  343.54
8192                  678.89
16384                1345.91
32768                2505.71
65536                4465.87
131072               7310.41
262144              10664.40
524288              13929.75
1048576             16370.99
2097152             17954.20
4194304             18874.38
8388608             19370.02
16777216            19436.62
33554432            19273.23
```

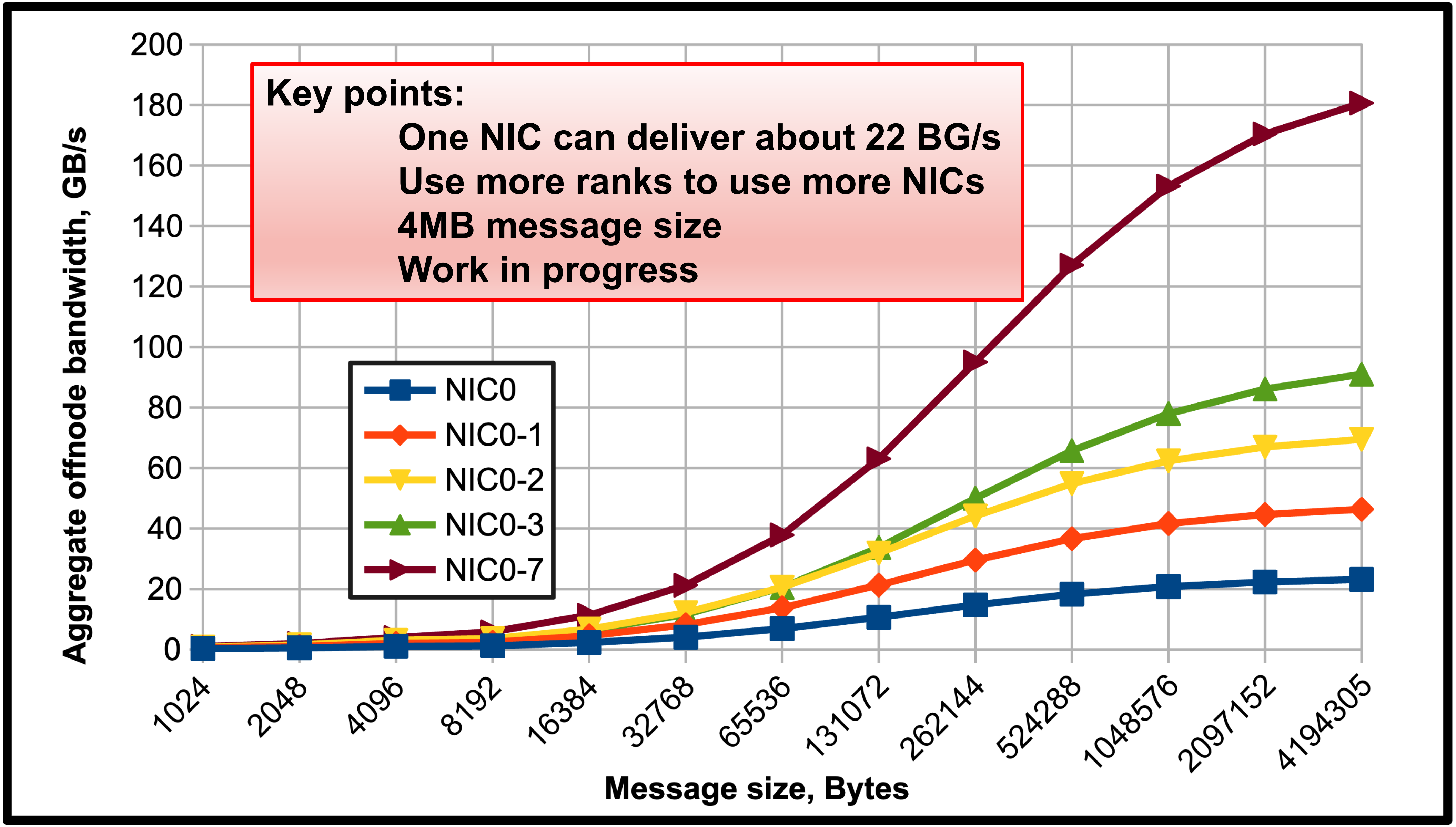


Key points:

About 20 GB/s/link

All Links can be used

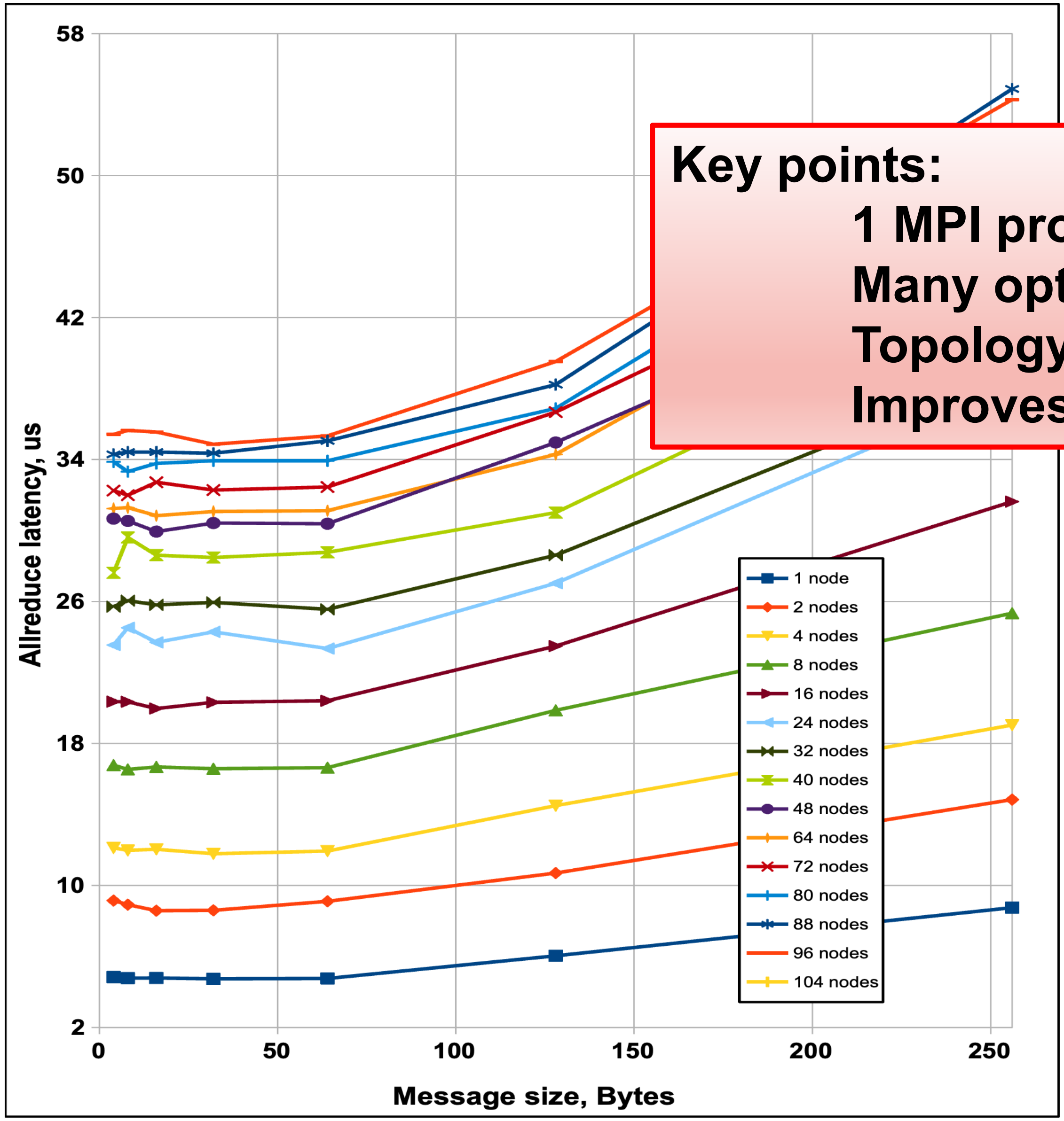
Benchmarks: Aggregate offnode bandwidth



Key points:

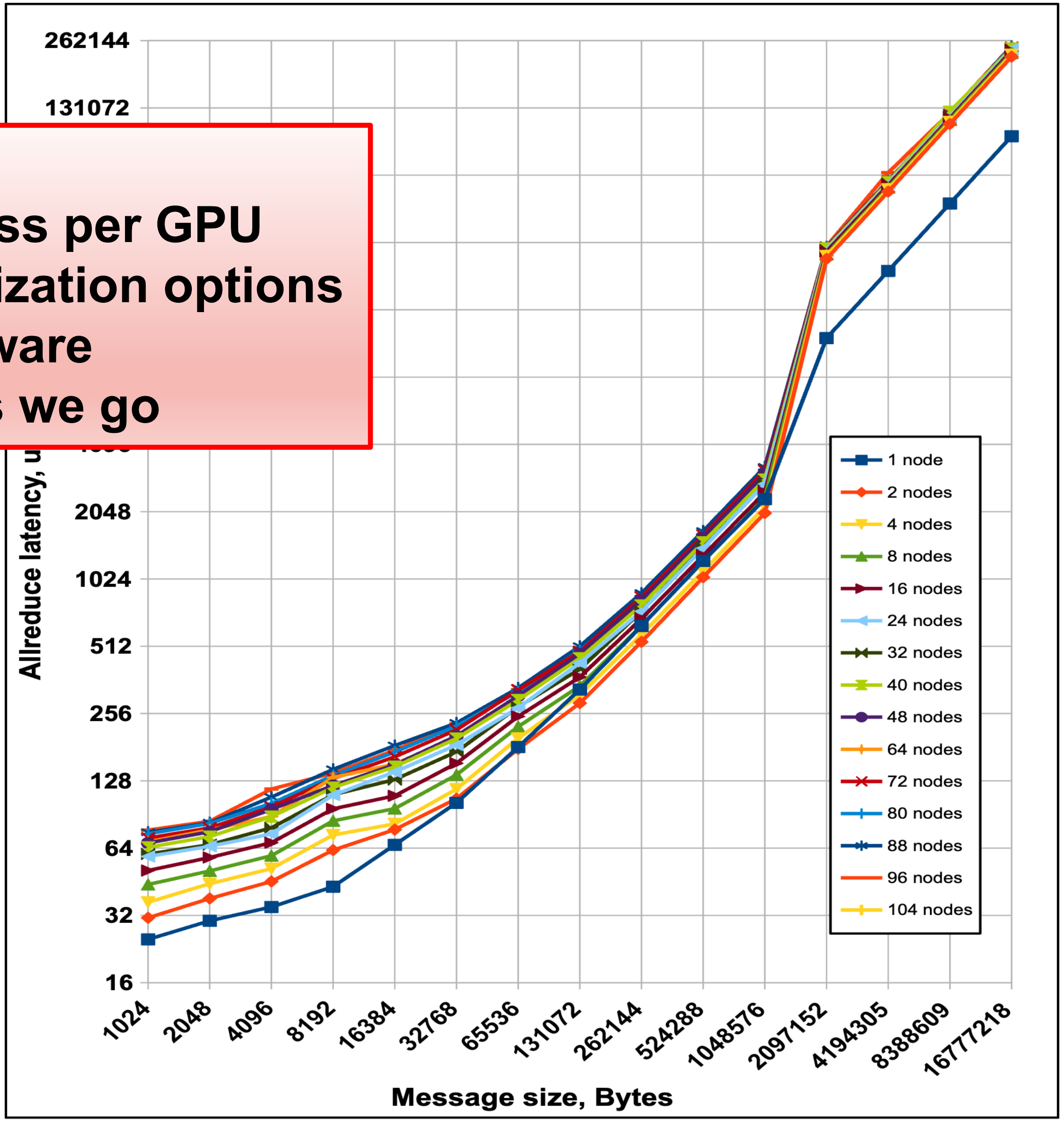
- One NIC can deliver about 22 BG/s
- Use more ranks to use more NICs
- 4MB message size
- Work in progress

Benchmarks: Allreduce Latency from GPU



Key points:

- 1 MPI process per GPU
- Many optimization options
- Topology-aware
- Improves as we go



Aurora-Sunspot Bob Walkup’s MPI profiler

```
Link:  -L/home/morozov/mpitrace/hpmprof -lhpmprof_c \
       -L/home/morozov/binutils-2.39/lib -lbfd -liberty \
       -L/home/morozov/zlib-1.2.13/lib -lz
```

```
Data for MPI rank 0 of 192:
Times and statistics from MPI_Init() to MPI_Finalize().
-----------------------------------------------------------------------
MPI Routine                        #calls     avg. bytes      time(sec)
-----------------------------------------------------------------------
MPI_Comm_rank                          10            0.0          0.000
MPI_Comm_size                           8            0.0          0.000
MPI_Isend                            1092     16424526.9          0.025
MPI_Recv                               52      2508532.2          0.095
MPI_Irecv                            1040     17119641.6          0.027
MPI_Wait                             1456            0.0         10.786
MPI_Waitall                           312            0.0          7.634
MPI_Barrier                            79            0.0          2.338
MPI_Allreduce                          87          373.9          5.652
-----------------------------------------------------------------
total communication time = 26.556 seconds.
total elapsed time       = 150.842 seconds.
user cpu time            = 140.157 seconds.
system time              = 11.229 seconds.
max resident set size    = 1739.879 MBytes.
```

Key points:

Familiar tools are ported

Final checklist

- ✓ If you are happy with the results, continue scaling to bigger hardware
- ✓ Socket: 52 cores, 3 devices, 4 NICs, Memory domains – complete system
  - ➢ Maximize resource utilization, balance, minimize sharing
- ✓ Mapping: ranks to cores, devices to ranks
  - ➢ `mpiexec …-ppn 3  --cpu-bind verbose,list:0-15:16-31:32-51` (1 NIC is unused)
  - ➢ `mpiexec …-ppn 12 --cpu-bind list:4:8:12:32:36:40:48:50:52`
  - ➢ `ZE_AFFINITY_MASK` different for each rank

```
if ((PALS_LOCAL_RANKID==3)); then
    export ZE_AFFINITY_MASK=3
fi
```

- ✓ Always make a sanity check

Key point:
Search for info: slack, support@alcf.anl.gov

MPICH BoF at Consortium for the Advancement of Scientific Software (CASS) Community BoF Days, June 11-13, 2024, Virtual

Argonne NATIONAL LABORATORY

THE COLLABORATION BETWEEN PETSC AND MPICH

Junchao Zhang
(jczhang@anl.gov)

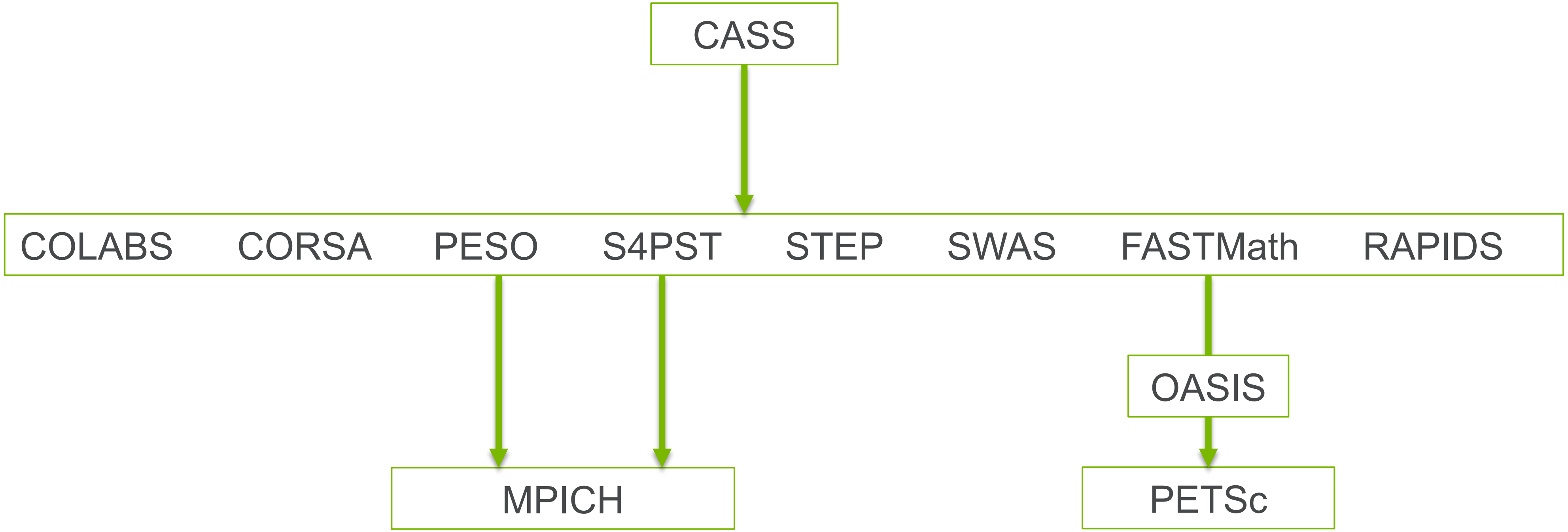
Mathematics and Computer Science Division
Argonne National Laboratory

June. 12, 2024

U.S. DEPARTMENT OF ENERGY

Argonne National Laboratory is a
U.S. Department of Energy laboratory
managed by UChicago Argonne, LLC.

PETSc and MPICH under CASS



MPICH and PETSc at Argonne

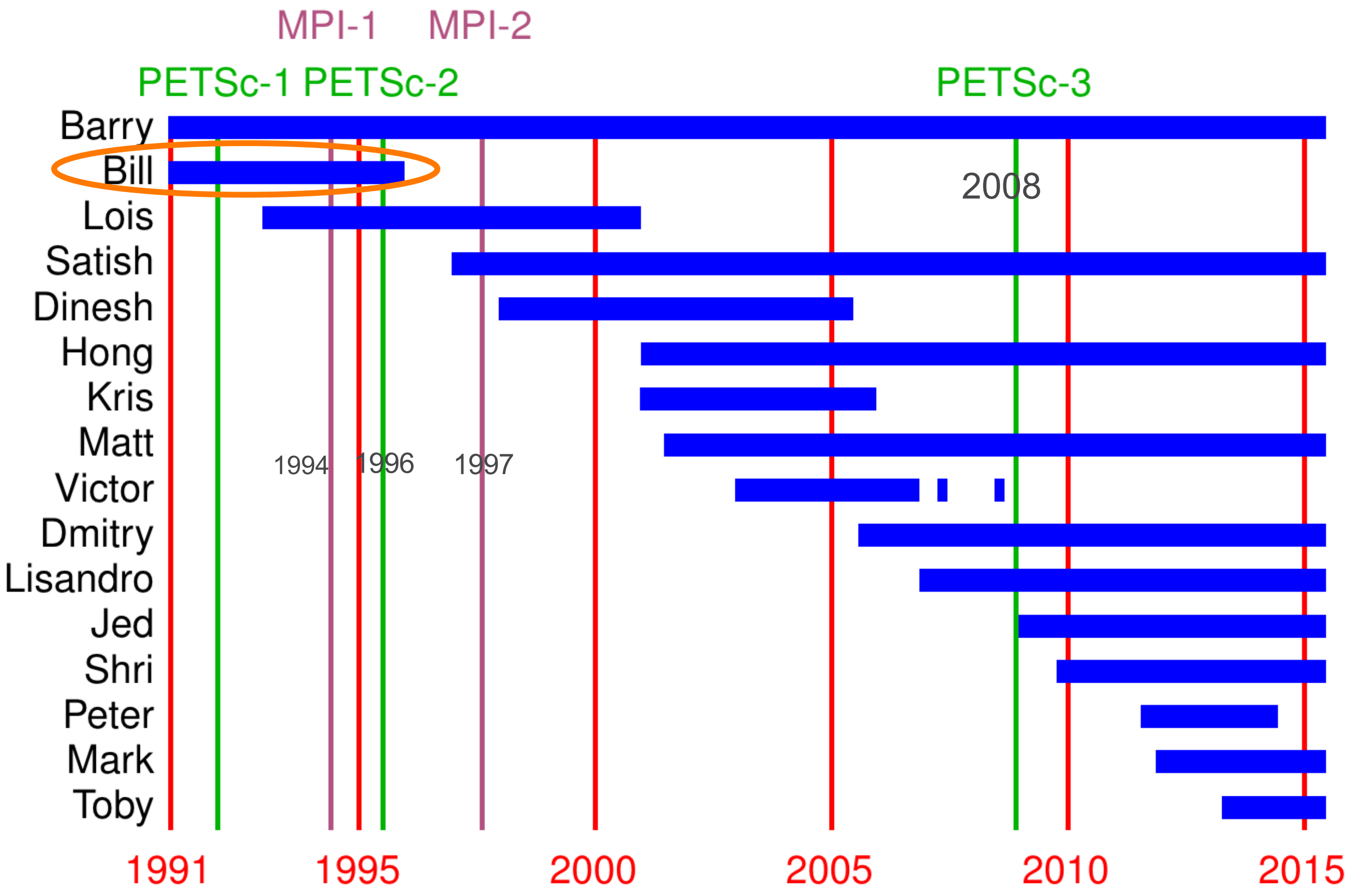
- MPICH: the most widely used MPI implementation and is the implementation of choice for the world's fastest machines

- PETSc: a scalable numerical library for linear and non-linear solvers and more
  - Has C, Fortran, Python, Rust bindings
  - Runs on Linux, Mac and Windows
  - Widely used in academia and industry in dozens of disciplines

Outline

- The PETSc/MPICH collaboration in the MPI-1.0~2.0 era and PETSc's adoption of new MPI features
- The PETSc/MPICH collaboration in the MPI-2.0~4.0 era and PETSc's adoption of new MPI features
- The PETSc/MPICH collaboration in recent years
- Conclusion

MPI-1.0~2.0 era



- MPICH was originally developed during the MPI standards process starting in 1992 to provide feedback to the MPI Forum on implementation and usability issues.
- Bill Gropp was deeply involved in both projects

PETSc’s most successful use of MPI-1.0 features

- MPI communicators and attributes
  - PETSc inner communicator to separate PETSc library messages from callers
  - Sub-communicator in multigrid solvers
- Persistent MPI_Send/Recv
  - Repeated, split-phased sparse neighborhood communication in Krylov solvers
- Various MPI collectives
  - MPI_Allreduce() for VecNorm(); two-sided discovery from one-sided
- MPI datatypes
  - Note derived data types are less used, since we mainly deal with sparse data

“MPI changed everything, by providing an extensive API for message passing and collectives that allowed portable distributed memory scientific libraries to no longer need to be programmed to the lowest common denominator of message passing systems. … The MPI communicator concept made distributed parallel scientific libraries practical in two ways, it eliminated the tag collision problem and (by the use of subcommunicators) allowed applications to simply utilize scientific libraries to perform needed computations on subsets of processes, for example with ‘divide and conquer’ algorithms.”

-- Barry Smith

https://www.hpcwire.com/2017/05/01/mpi-25-years-old/

PETSc’s adoption of new MPI-2.0 features

MPI-IO

✅ MPI Fortran-90 binding

❌ MPI one-sided (RMA) & dynamic process

- Not even tried in the next decade

❌ MPI + multithreading

- PETSc added support for both OpenMP and Pthreads and found the code was never faster than pure MPI and cumbersome to use hence we have removed it

“The PETSc team has no problems with proposals to replace the pure MPI programming model with a different programming model but only with an alternative that is demonstrably better, …

At least for the PETSc package, the concept of being thread-safe is not simple. It has major ramifications about its performance and how it would be used; it is not a simple matter of throwing a few locks around and then everything is honky-dory.”

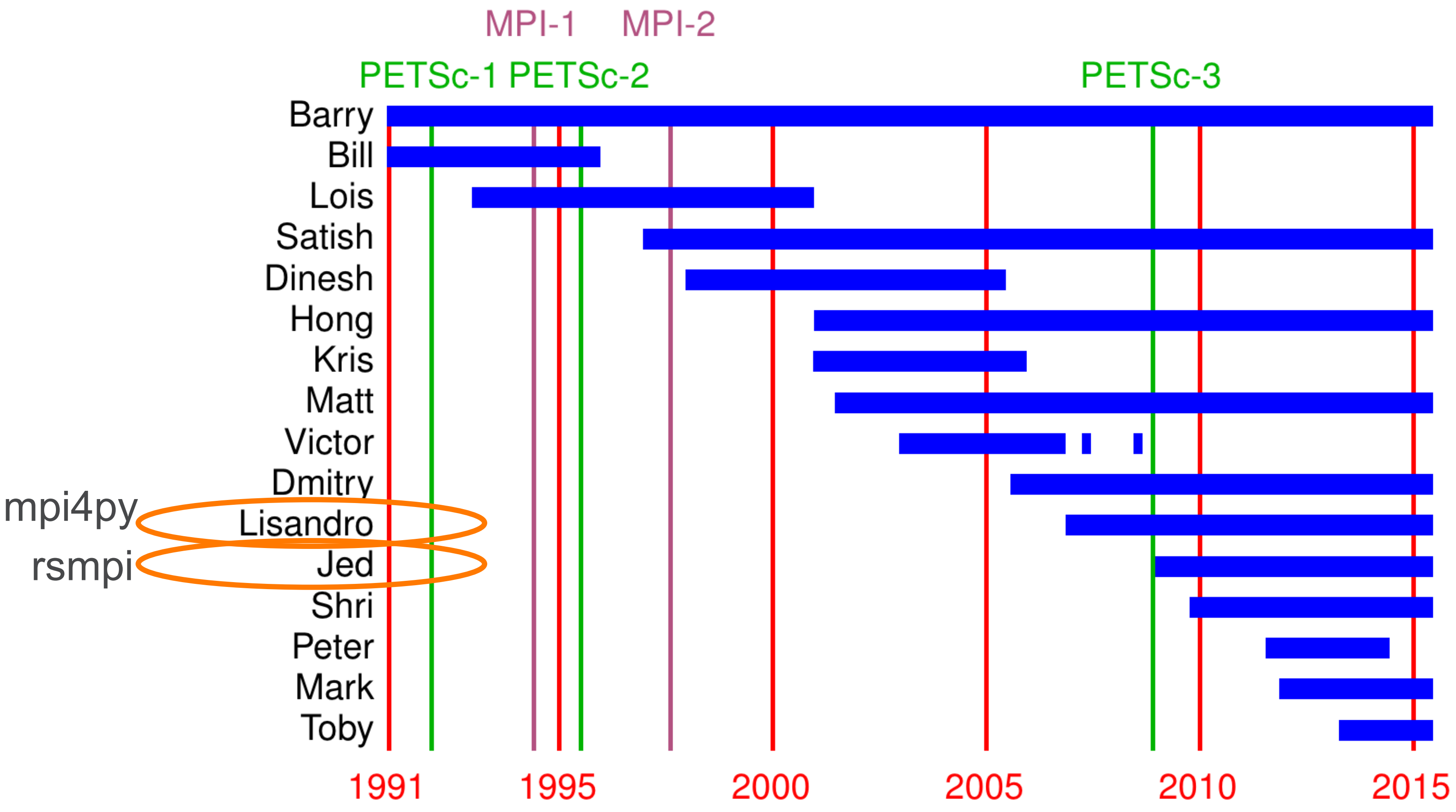
-- Barry Smith

https://www.mcs.anl.gov/petsc/petsc-3.15/docs/miscellaneous/threads.html

MPI-2.0~4.0 era

- As both projects became mature, the close collaboration was almost lost
- PETSc occasionally tried new features introduced in the MPI standard
  - MPI-3.0 process-shared memory to improve intra-node communication
    - Not easier than two-sided for sparse-neighborhood & no obvious performance benefit
  - MPI-3.0 revised one-sided in PetscSF implementation
    - `-sf_type window -sf_window_flavor <create|dynamic| allocate> -sf_window_sync <fence|active|lock>`
    - Yet to show an advantage over two-sided
  - MPI (persistent) neighborhood collectives
    - `-sf_type neighbor –sf_neighbor_persistent <bool>`
  - MPI_Iallreduce() in pipelined CG solver (`–ksp_type pipecg`)
  - MPI_Ibarrier/Iprobe() with `–build_twosided ibarrier*`
    - The ibarrer alg. [hoefler2010] performs better at large scale than the allreduce alg.
    - Less reliable than allreduce, frequently run into errors with Intel MPI
  - MPI large count (`--with-64-bit-indices`)

PETSc developers' contribution to the MPI community -- MPI for Python and Rust maintainers



The enhanced PETSc/MPICH collaboration in recent years

- PETSc CI job coverage with MPICH on GPUs
  - PETSc CI helped MPICH identify its excessive GPU memory usage
  - MPICH helped PETSc discovery a serious GPU stream sync bug
- PETSc is experimenting with the MPICH GPU stream extension
  - `-sf_use_gpu_aware_mpi <bool>` (not steam-aware)
  - `-sf_use_stream_aware_mpi <bool>` (experimental)
- PETSc is experimenting with the MPI-5.0 ABI implemented in MPICH
  - PETSc users might mess up the PETSc build time MPI (e.g., OpenMPI) with user code build time MPI (e.g., MPICH)
  - It is helpful to unify the MPI ABI
- PETSc inspired the MPIX_THREADCOMM extension in MPICH

The “PETSc + OpenMP” dilemma

- PETSc doesn’t support OpenMP because of the complexity and bad performance
- Some OpenMP-only codes want to call PETSc to leverage its tons of solvers
  - Also want PETSc to be run in parallel to make use of the CPU cores

The MPICH MPIX_Threadcomm Solution

```
Mat       A;
Vec       x, b;
int       nthreads = 4;
MPI_Comm comm;
PetscInitialize(&argc, &argv, NULL, NULL);
// user code building A, x, b etc
…
MPIX_Threadcomm_init(MPI_COMM_WORLD, nthreads, &comm);
#pragma omp parallel num_threads(nthreads)
{  Mat A2;
   Vec x2, b2;
   KSP ksp;
   MPIX_Threadcomm_start(comm); // comm’s size is 4
   MatCreate(comm, &A2);
   MatCreateVecs(A2, &x2, &b2);
   // Assemble A2, b2 from the shared A, b
   KSPSolve(ksp, b2, x2);
   // Transfer the solution x2 to x
   MatDestroy(&A2);
   MPIX_Threadcomm_finish(comm)
 }
MPIX_Threadcomm_free(&comm);
PetscFinalize();
```

- Run the test as a regular OMP code:
  `OMP_NUM_THREADS=8 ./test -args`

- User’s sequential code (might use OpenMP)
- PETSc is initialized on a single process
- Build sequential petsc objects such as matrices and vectors

- Build parallel petsc objects on the threadcomm *comm*
- *Somehow* transfer data from the shared sequential A, b to parallel A2, b2
- Other parts of the petsc code work as if they were run by `mpiexec -n 4 ./test`
- Caveats: petsc needs to be thread safe, e.g., in logging
- Future work: provide a new preconditioner type `PCOMP` to wrap around this stuff

Summary: MPI & MPICH's use in PETSc

- In 2024, PETSc can still build with MPI-2.1 without (performance) issues!!
- PETSc users do not need MPI if they only use PETSc sequentially
  - `./configure --with-mpi=0`
  - petsc will use its fake single-process MPI (mpiuni) impl. to provide MPI APIs
  - Maybe MPICH could take it over as others also like it (?)
- MPICH is recommended by PETSc for users needing valgrind
- The *latest* MPICH can be downloaded and installed by PETSc (many users use that!)

  -
    ```
    ./configure --with-cc=gcc --with-cxx=g++ --with-cuda
     --download-mpich –download-mpich-device=<ch3:nemsis|..>
    ```
- PETSc has 8000+ tests and 70+ CI jobs with many using MPICH for testing

Conclusion

- PETSc is an excellent testbed and a real world inspiring example for MPI and MPICH research
- The closer the two projects collaborate, the better they can serve their users