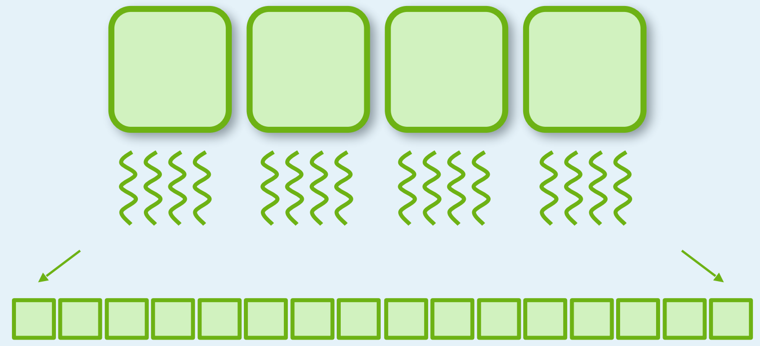
MPI × THREADS



2023 Annual PETSc Meeting

Hui Zhou
The MPICH Team
Argonne National Laboratory

PARALLEL COMPUTING

ELEMENTS OF PARALLEL COMPUTING

- Parallel codes
  - Multiple programs – client/server, master/workers, tasks, events
  - ☆Single program, multiple data – MPI, OpenMP, CUDA
- Parallel environment
  - Manual – multiple programs, listen, connect
  - Launch once – `mpirun -n N prog …`
  - ☆Dynamic spawn -- `#pragma omp parallel`
- Avoid data races
  - Critical sections
  - ☆Privatize data
- Manage synchronizations
  - Barriers, mutexes, atomics
  - ☆Messages, collectives

EMERGED PARADIGMS

MPI

- Distributed processes, private memory
- Static launch – `mpirun -n N prog …`
- Explicit and rich toolkit for synchronization

OpenMP

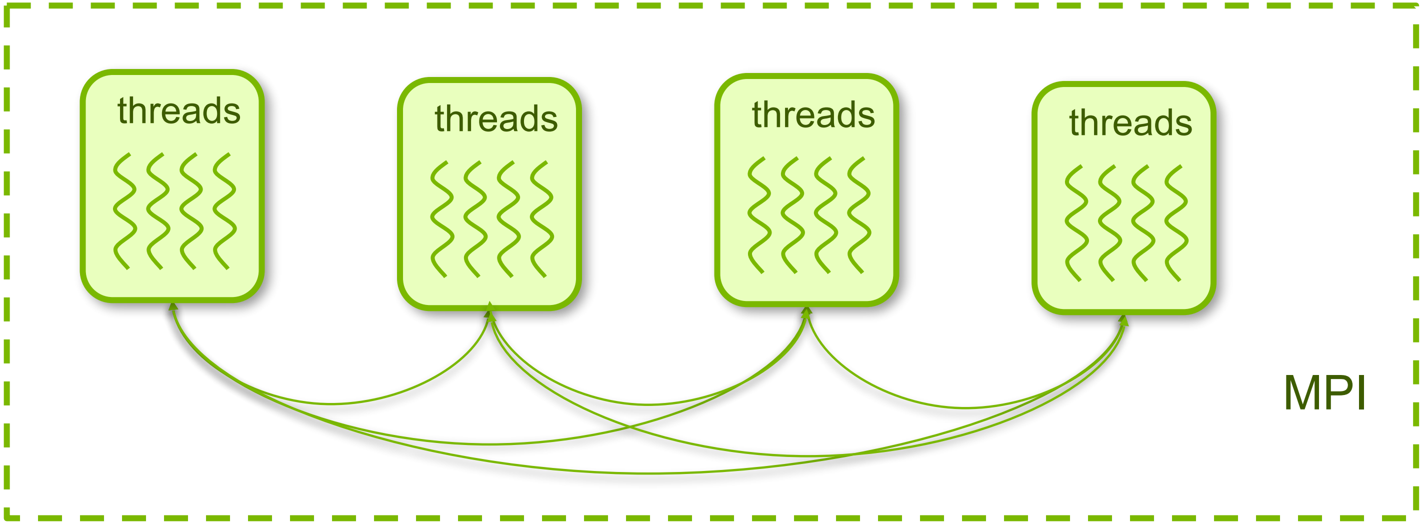
- Threads with shared memory
- Dynamic launch – `#pragma omp parallel`
- Implicit, critical section-based synchronizations

NVIDIA CUDA

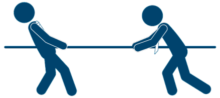
- Threads on offloading devices, shared memory
- Dynamic launch – `compute<<<N,M>>>()`
- Implicit, dependency-based synchronizations

There are advantages and disadvantages, naturally …

MPI + THREADS



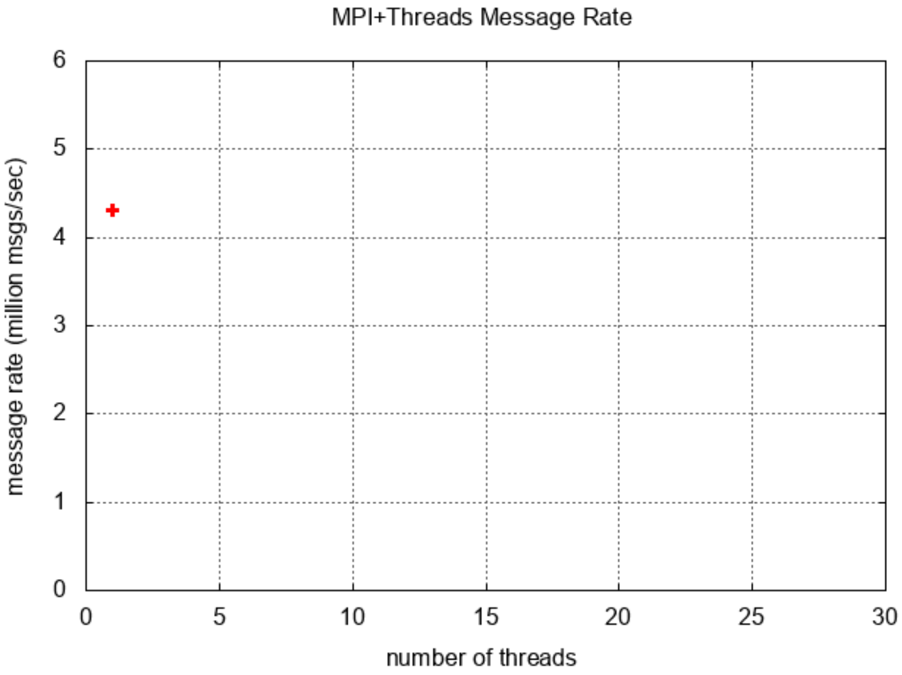
MPI THREAD MODEL



- MPI thread levels
  - `MPI_THREAD_SINGLE` Threads not allowed
  - `MPI_THREAD_FUNNELED` Only the main thread calls MPI
  - `MPI_THREAD_SERIALIZED` Any thread can call MPI, but not concurrently
  - `MPI_THREAD_MULTIPLE` No restrictions

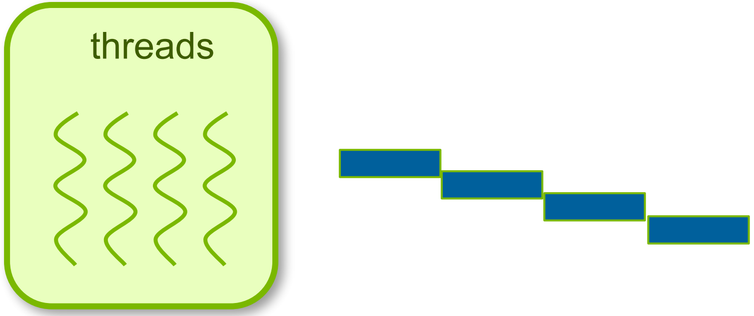
- <u>IMPORTANT NOTES</u>:
  - `MPI_THREAD_MULTIPLE` does not mean free of synchronizations
  - MPI implementations tend to do worse in managing thread synchronizations
  - For later, there is opportunities …

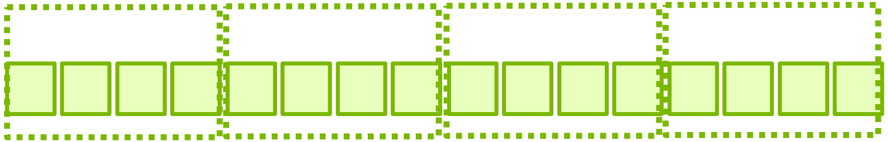
MISERABLE (DEFAULT) PERFORMANCE OF `MPI_THREAD_MULTIPLE` AND SMALL MESSAGES

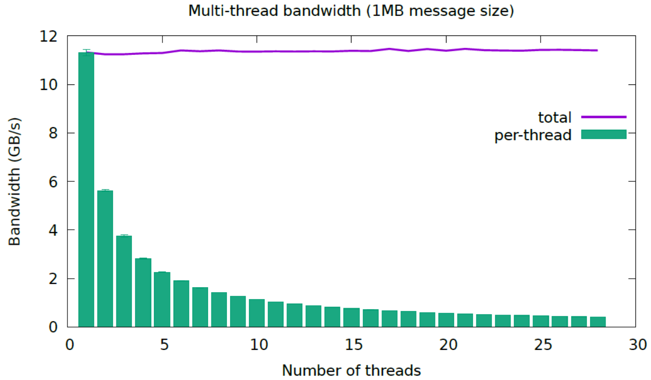


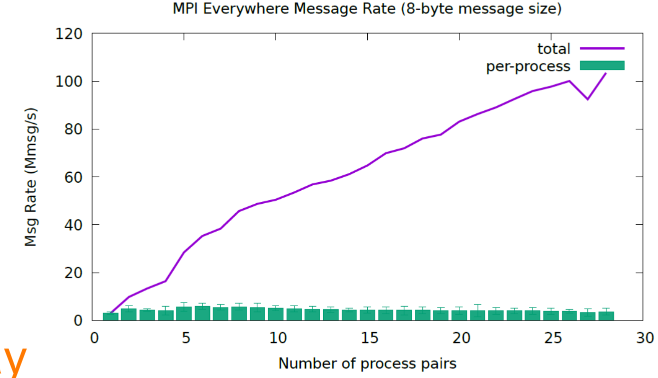
Message size = 8 bytes, Intel Xeon 8176 CPU, Mellanox ConnectX-5

WORKAROUNDS:
LARGE MESSAGE OR MPI-EVERYWHERE







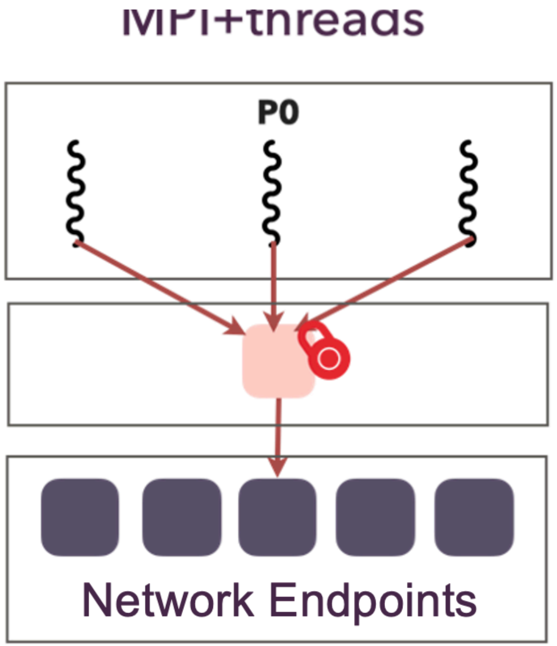


Both workarounds are less applicable today

SYNCHRONIZATIONS INSIDE IMPLEMENTATIONS

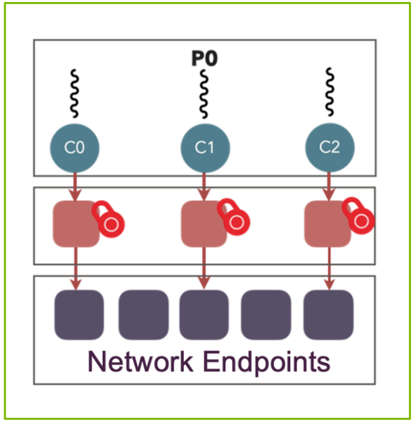
- Request objects
  - Async progress and completion
  - Synchronization required in accessing request pools
- Message queues
  - Required to enforce message order
  - Enqueue and dequeue to message queues
- Low level library and driver
  - Issue packets and polling events
  - Modern NIC can afford limited number of concurrent endpoints

MPI+Thread



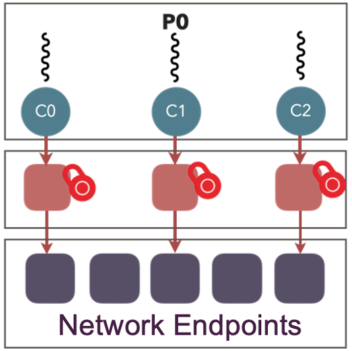
PER-VCI LOCKS

- Align objects into virtual communication interfaces (VCI)
- Start with network endpoints, add request pools and message queues
- Map the communication to VCI and apply per-vci lock
- Operations using different VCIs can go parallel
- Performance depend on successful mapping strategies
- Achieve good strong scaling when all stars aligned



IMPLICIT MAPPING

```
MPI_Send(buf, count, datatype, dest, tag, comm)
MPI_Recv(buf, count, datatype, src, tag, comm,
status)
```

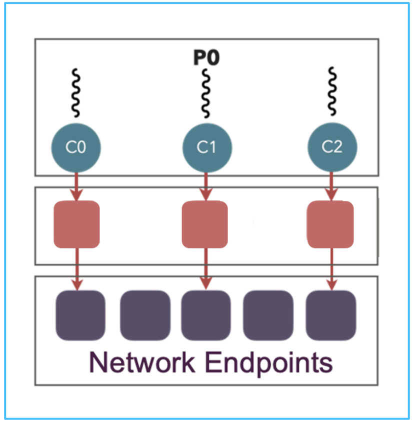


- Scenerios MPI does not require message order
  - Messages in different communicators
  - Messages with different source/destination[1]
  - Messages with different tags[2]
- Application can use comm/rank/tag to express parallelism
- But –
  - communicators (or ranks and tags) are the wrong semantics for execution context
  - Can't match the performance of `MPI_THREAD_SINGLE`

[1] When MPI_ANY_SOURCE is not used.
[2] When MPI_ANY_TAG is not used.

EXPLICIT SOLUTIONS TO MPI+THREAD

- Goal: Let application express parallelism directly to MPI
- Problem: MPI does not acknowledge threads
- Let's address the problem directly!
- *Let's improve MPI!*



EXTENSION: MPIX STREAM

- `MPIX_Stream` identifies a *serial* execution context

```
int MPIX_Stream_create(MPI_Info info, MPIX_Stream *stream)
int MPIX_Stream_free(MPIX_Stream *stream)
```

- `info` can be `MPI_INFO_NULL`, identifies a generic CPU context (i.e. thread)
- It can be specialized into offloading context, e.g. for `cudaStream_t`

```
MPI_Info_create(&info);
MPI_Info_set(info, "type", "cudaStream_t");
MPIX_Info_set_hex(info, "value", &stream, sizeof(stream));

MPIX_Stream_create(info, &mpi_stream);
```

STREAM COMMUNICATOR

- Stream communicator is a communicator with local streams attached.

```
int MPIX_Stream_comm_create(MPI_Comm parent_comm,
          MPIX_Stream stream, MPI_Comm *stream_comm)
```

- The stream communicator specifies both the local and remote network endpoints
- Otherwise, synchronizations are unavoidable at receiver or sender.
- It’s backward compatible
  – Conventional communicators are the same as stream communicators with `MPIX_STREAM_NULL` on every process.

GPU ENQUEUE OPERATIONS

Showing Point-to-Point Communications

- Alias APIs for async launching MPI communications

```
int MPIX_Send_enqueue(buf, count, datatype, dest, tag, comm)
int MPIX_Recv_enqueue(buf, count, datatype, source, tag,
                    comm, status)
int MPIX_Isend_enqueue(buf, count, datatype, dest, tag,
                    comm, request)
int MPIX_Irecv_enqueue(buf, count, datatype, source, tag,
                    comm, request)
int MPIX_Wait_enqueue(request, status)
int MPIX_Waitall_enqueue(count, array_of_requests,
                    array_of_statuses)
```

MPI+THREAD PERFORMANCE

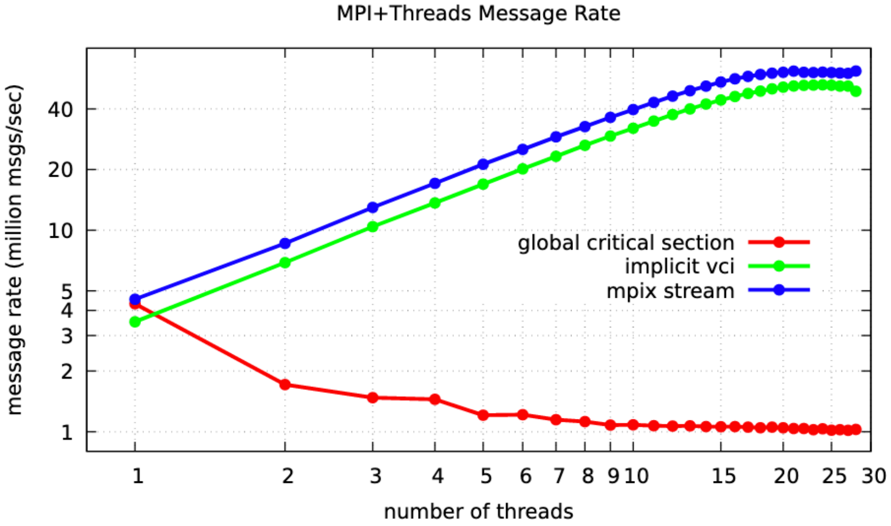
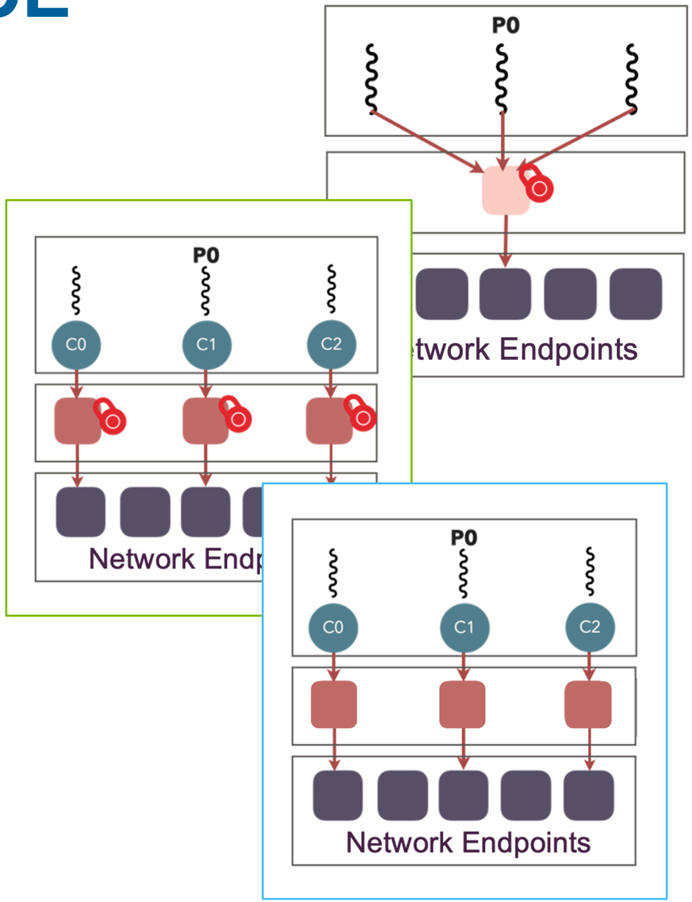
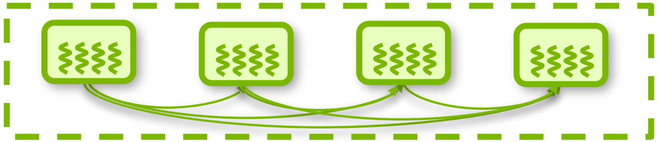


Figure 3: Multithread message rate on 8-byte messages using MPI_Isend/MPI_Irecv.

MPI+Thread



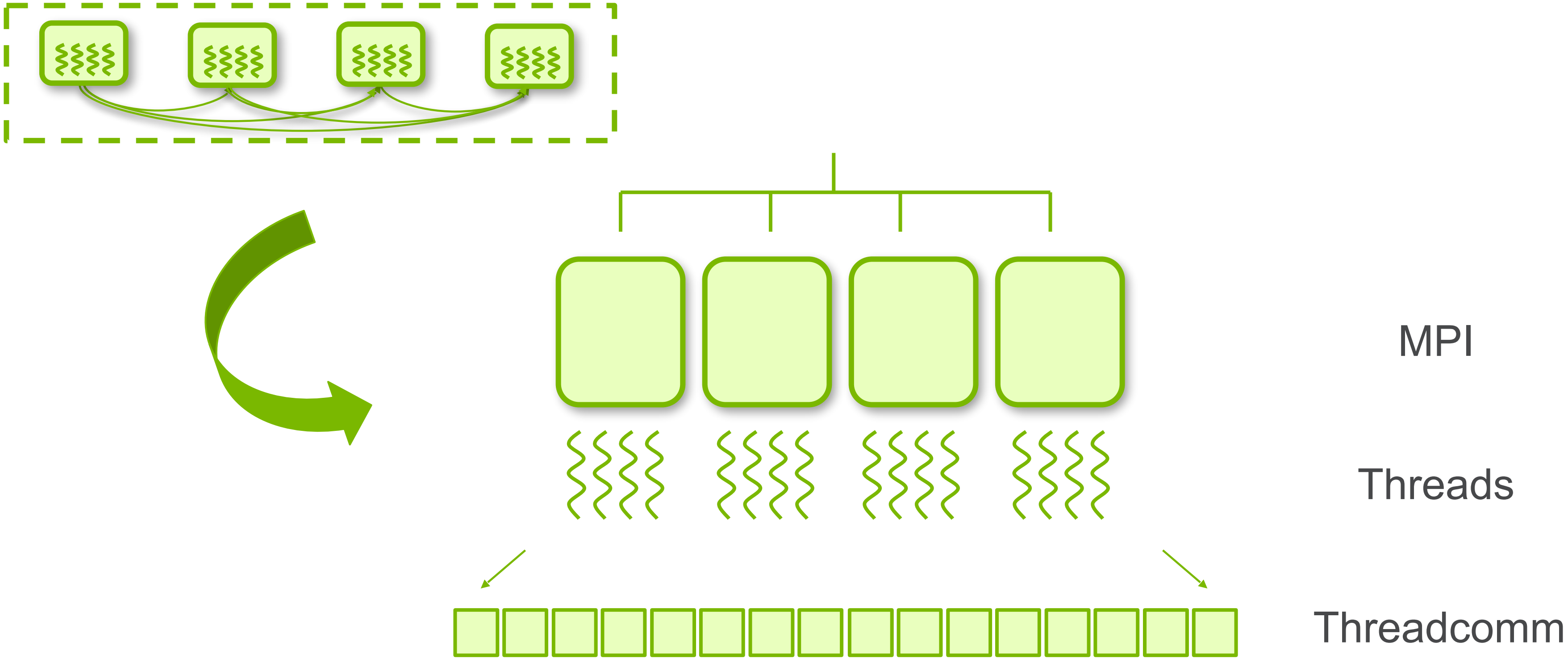
MPIX STREAM IMPROVES MPI + THREADS, HOWEVER …



- MPI and Threads are still not collaborating –
  - Inter-threads cannot use MPI for synchronization
  - MPI cannot use threads to dynamically spawn
  - Similar parallel tasks are re-invented and juxtaposed

- Case in point: PETSc
  - Has tried, and MPI has won
  - Use MPI-everywhere for PETSc
  - But users still use OpenMP
  - How to do OpenMP + PETSc?

INTRODUCING MPI × THREADS



MPIX THREAD COMMUNICATOR

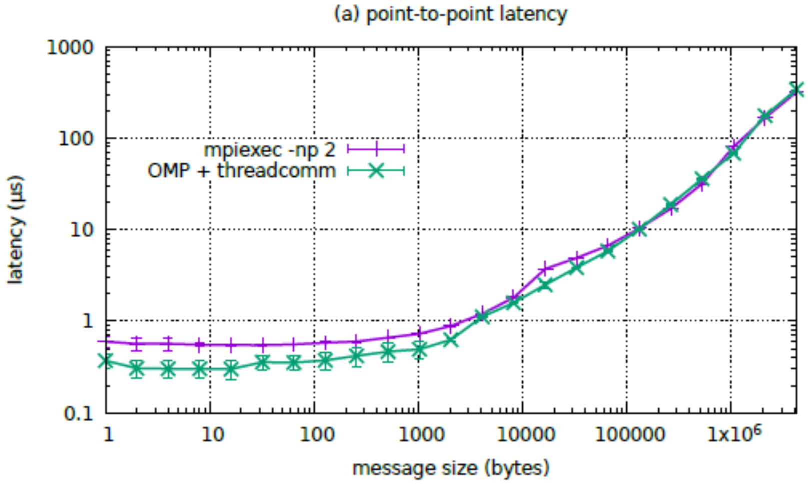
- Synopsis

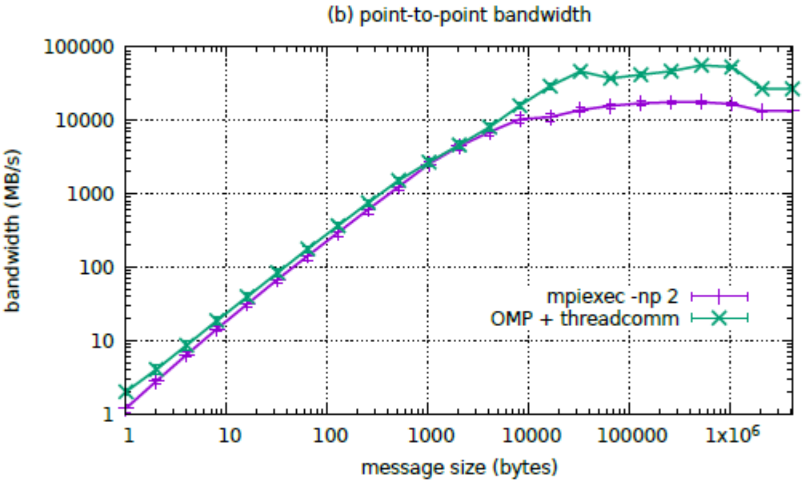
```
int MPIX_Threadcomm_init(MPI_Comm comm, int num_threads,
                         MPI_Comm threadcomm)
```

```
#pragma omp parallel {
    MPIX_Threadcomm_start(threadcomm);
    /* use threadcomm within parallel region */
    MPIX_Threadcomm_finish(threadcomm);
}
```

```
int MPIX_Threadcomm_free(MPI_Comm *threadcomm)
```

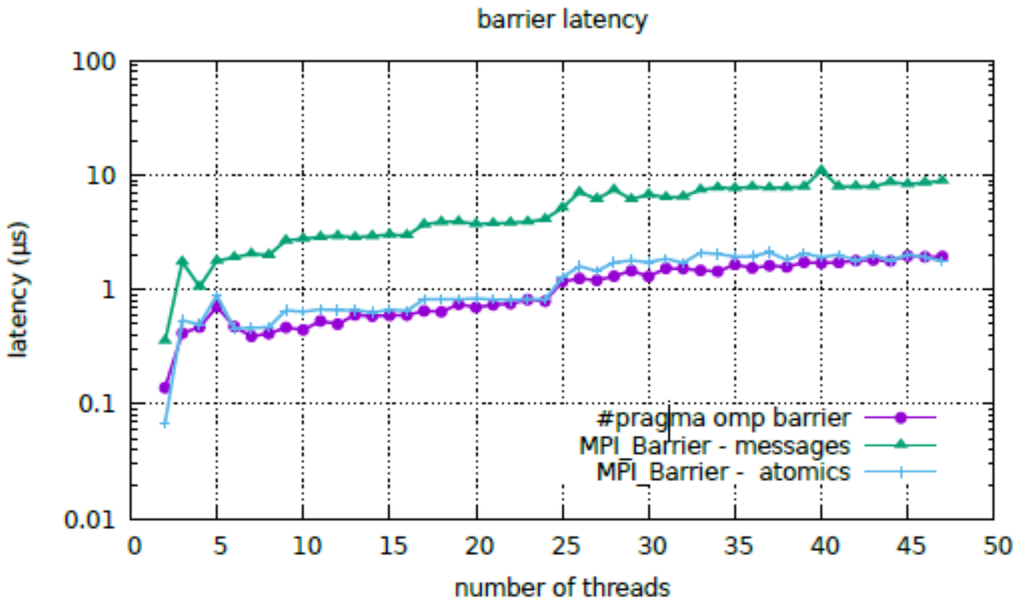
LATENCY AND BANDWIDTH





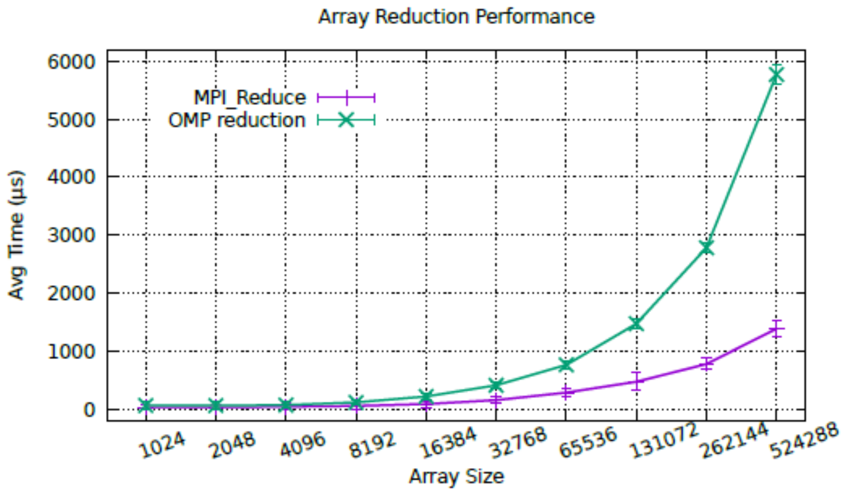
BARRIER

```
#pragma omp parallel
{
    MPI_Threadcomm_start(comm);
  #ifdef USE_MPI
     MPI_Barrier(comm)
  #else
     #pragma omp barrier
  #endif
    MPI_Threadcomm_finish(comm);
}
```



REDUCTION

```
int sum[N];
#ifdef USE_MPI
  #pragma omp parallel
  {
    MPI_Threadcomm_start(comm);
    int my[N];
    int tid = omp_get_thread_num();
    for (int i = 0; i < N; i++) my[i] = tid;
    MPI_Reduce(my, sum, N, MPI_INT, MPI_SUM, 0,
        comm);
    MPI_Threadcomm_finish(comm);
  }
#else
  #pragma omp parallel reduction(+:sum[:N])
  {
    int tid = omp_get_thread_num();
    for (int i = 0; i < N; i++) sum[i] = tid;
  }
#endif
```

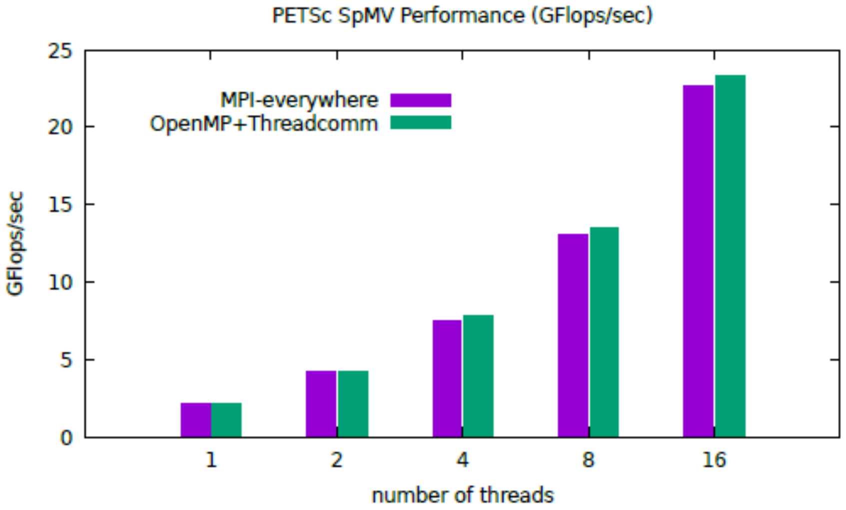


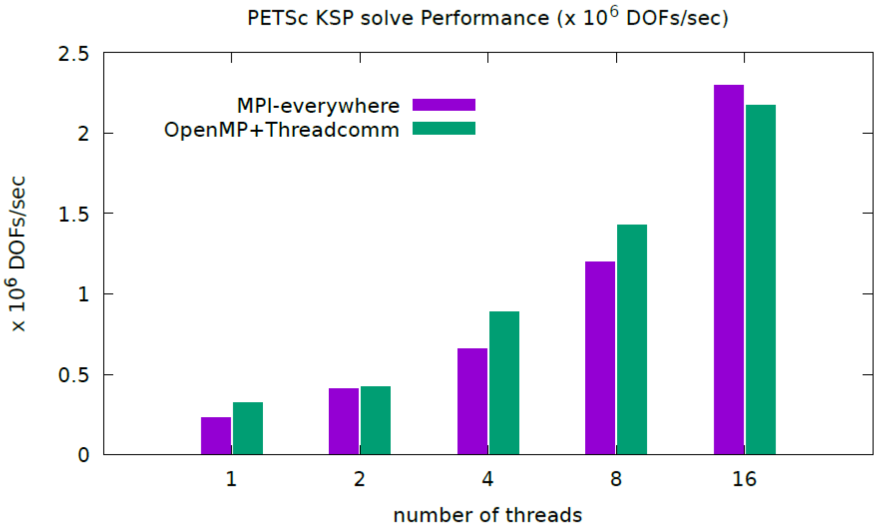
USING PETSC WITH THREADCOMM

```
int       nthreads = 4;
MPI_Comm comm;

MPI_Init(NULL, NULL);
PetscInitialize(&argc, &argv, NULL, NULL);

MPIX_Threadcomm_init(MPI_COMM_WORLD, nthreads,
     &comm);
#pragma omp parallel num_threads(nthreads)
{
    Mat A;
    MPIX_Threadcomm_start(comm);
    MatCreate(comm, &A);
    /* Build matrix A with data from outside
        the parallel region and do parallel
        computation */
    MatDestroy(&A);
    MPIX_Threadcomm_finish(comm);
}
MPIX_Threadcomm_free(&comm);
PetscFinalize();
MPI_Finalize();
```

- PETSc is not thread-safe
  - Use thread-local storage
  - Global init, then read-only
  - Logging and debugging
    - Need mutexes
    - Need threadcomm-aware
- The lessons apply to all MPI-only applications
- The changes required by adaptation are minimal

PETSC+THREADCOMM PERFORMANCE





SUMMARY

- MPI suffers bad performance with MPI_THREAD_MULTIPLE
- Modern MPI implementation capable of achieving good scaling via implicit VCI mapping
- New proposal MPIX Stream adds explicit thread context to MPI
- New proposal to extend MPI to interthread communication
- MPIX Stream API available in MPICH-4.1
- Thread communicator will be available in MPICH-4.2

Q & A