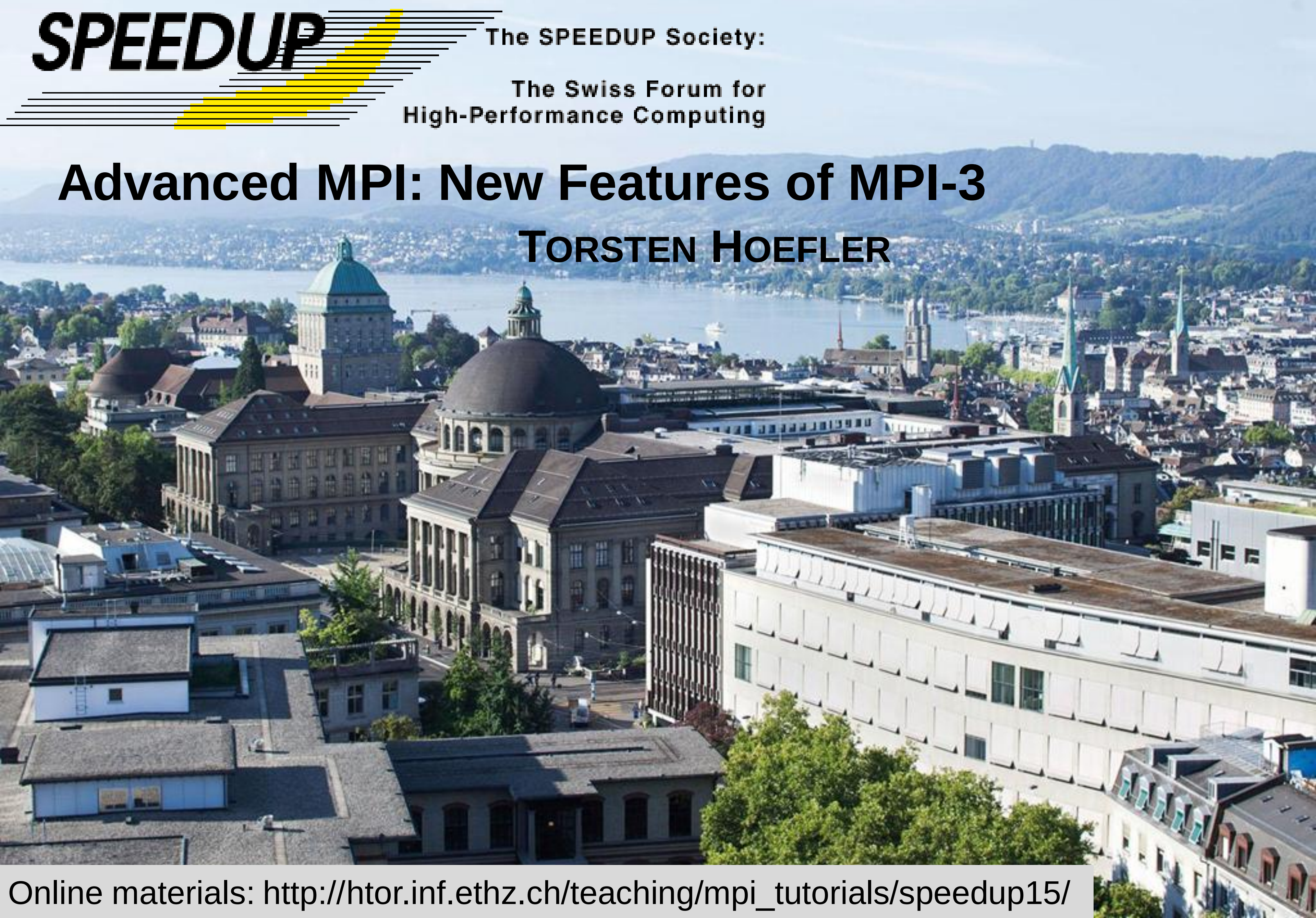
SPEEDUP

The SPEEDUP Society:

The Swiss Forum for
High-Performance Computing

Advanced MPI: New Features of MPI-3

TORSTEN HOEFLER



Online materials: http://htor.inf.ethz.ch/teaching/mpi_tutorials/speedup15/

Tutorial Outline

1. **Introduction to Advanced MPI Usage**
2. **Nonblocking Collective Communication**
3. **One-Sided Communication**
4. **Topology Mapping and Neighborhood Collective Communication**
5. **Bonus Material (only if time)**
    1. Hybrid Programming Primer
    2. Datatypes

- **All materials (slides, code examples) at:**

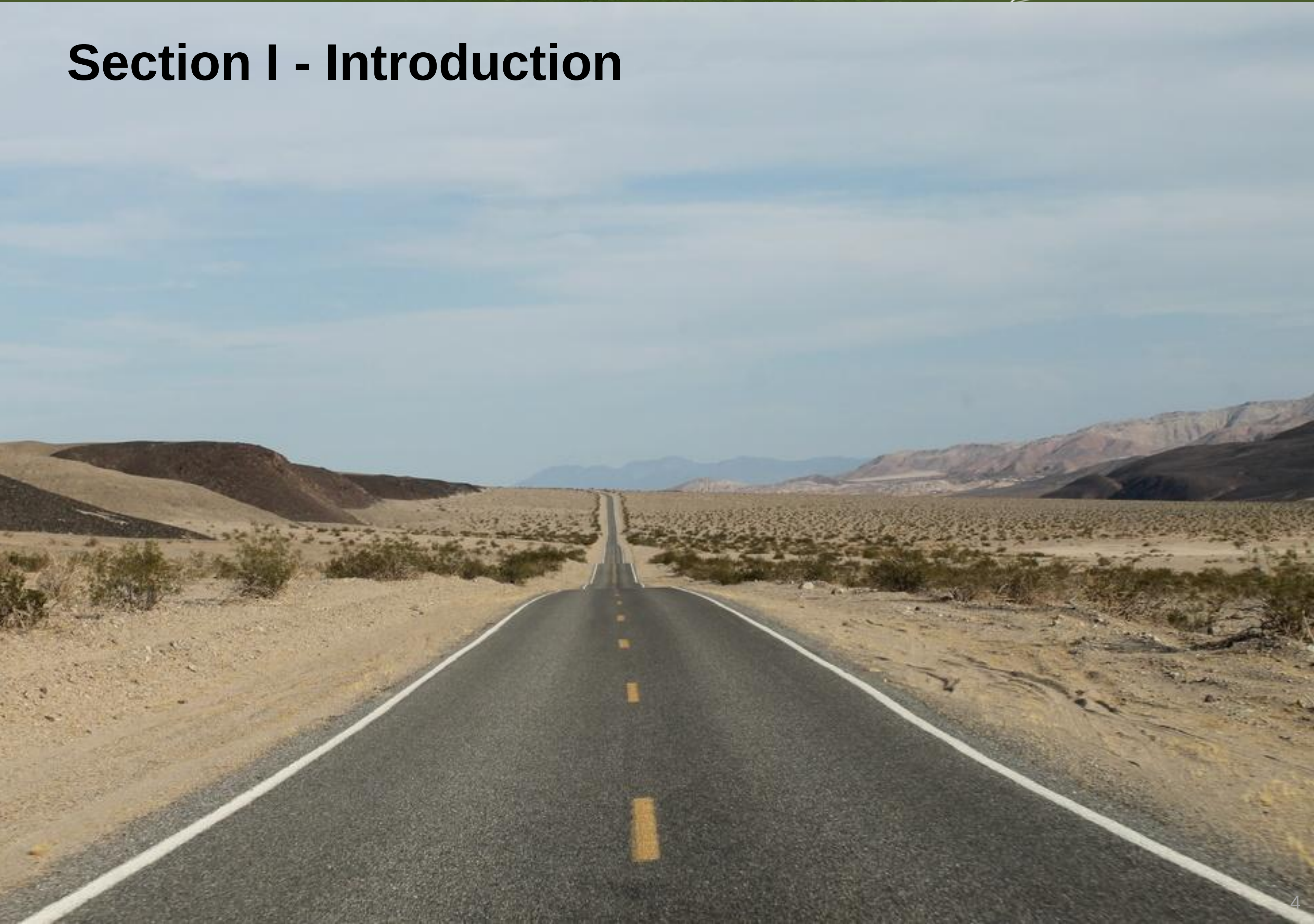
  http://htor.inf.ethz.ch/teaching/mpi_tutorials/speedup15/

Used Techniques

- **Benjamin Franklin *"Tell me, I forget, show me, I remember, involve me, I understand.“***
  - **Tell**: I will explain the abstract concepts and interfaces/APIs to use them
  - **Show**: I will demonstrate one or two examples for using the concepts
  - **Involve**: You will transform a simple MPI code into different semantically equivalent optimized ones

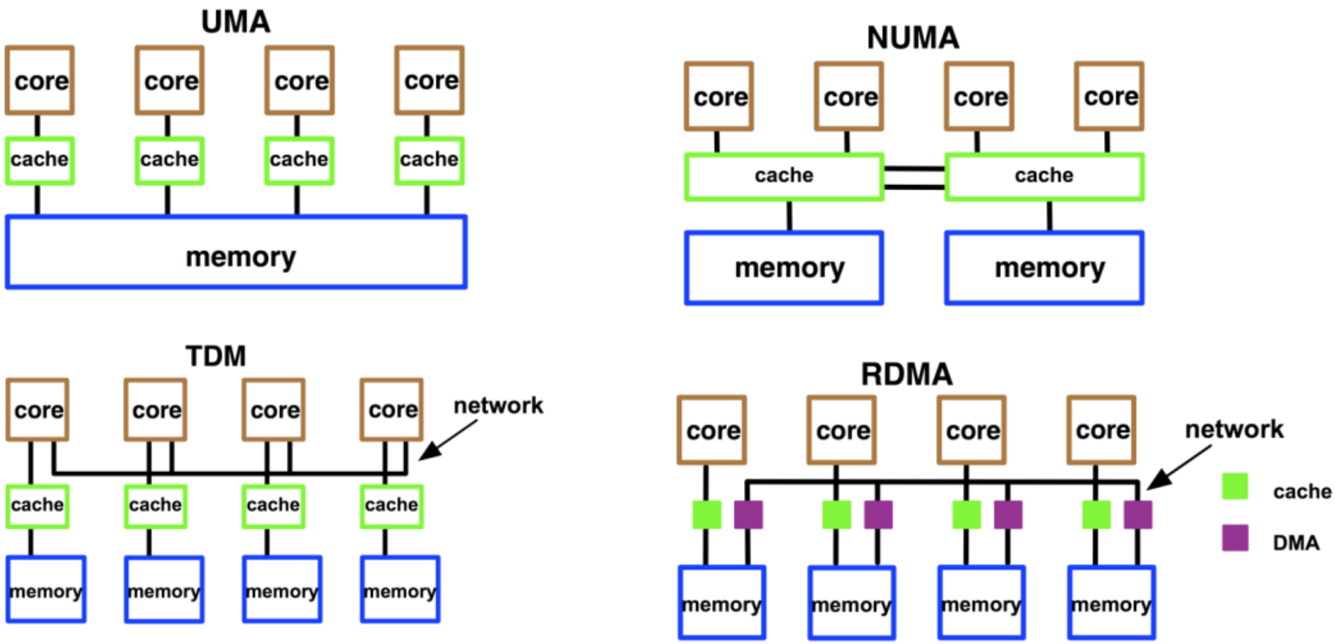
- **Please interrupt me with any question at any point!**

Section I - Introduction



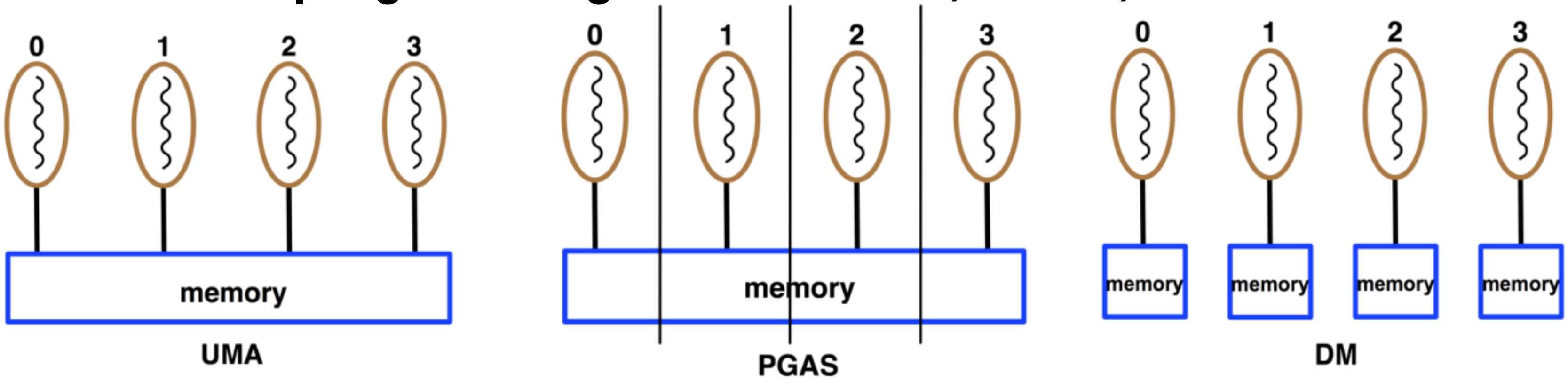
Introduction

- **Programming model Overview**
- **Different systems: UMA, ccNUMA, nccNUMA, RDMA, DM**



Introduction

- **Different programming models: UMA, PGAS, DM**

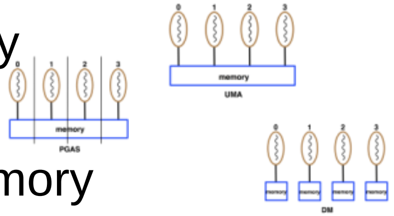


TBB, CILK, OpenMP, MPI-3 SM UPC, CAF, MPI-3 OS MPI-1, PVM

- **The question is all about memory consistency**

Programming Models

- **Provide abstract machine models (contract)**
  - Shared memory
  - PGAS
  - Distributed memory



- **All models can be mapped to any architecture, more or less efficient (execution model)**

- **MPI is not a programming model**
  - And has never been one!

MPI Governing Principles

- **(Performance) Portability**
  - Declarative vs. imperative
  - Abstraction (of processes)

- **Composability (Libraries)**
  - Isolation (no interference)
  - Opaque object attributes

- **Transparent Tool Support**
  - PMPI, MPI-T
  - Inspect performance and correctness

Main MPI Concepts

- **Communication Concepts:**
    - Point-to-point Communication
    - Collective Communication
    - One Sided Communication
    - (Collective) I/O Operations
- **Declarative Concepts:**
    - Groups and Communicators
    - Derived Datatypes
    - Process Topologies
- **Process Management**
    - Malleability, ensemble applications
- **Tool support**
    - Linking and runtime

MPI History

- **An open standard library interface for message passing, ratified by the MPI Forum**
- **Versions: 1.0 ('94), 1.1 ('95), 1.2 ('97), 1.3 ('08)**
  - Basic Message Passing Concepts
- **2.0 ('97), 2.1 ('08)**
  - Added One Sided and I/O concepts
- **2.2 ('09)**
  - Merging and smaller fixes
- **3.0 ('12)**
  - Several additions to react to new challenges
- **3.1 ('15)**
  - Several smaller issues and (hopefully) FT
- **4.0 ('??)**
  - Unclear (come next week to Kobe!!)

What MPI is Not

- **No explicit support for active messages**
  - Can be emulated at the library level
- **Not a programming language**
  - But it’s close, semantics of library calls are clearly specified
  - MPI-aware compilers under development
- **It’s not magic**
  - Manual data decomposition (cf. libraries, e.g., ParMETIS)
    Some MPI mechanisms (Process Topologies, Neighbor Colls.)
  - Manual load-balancing (see libraries, e.g., ADLB)
- **It’s neither complicated nor bloated**
  - Six functions are sufficient for any program
  - 250+ additional functions that offer abstraction, performance portability and convenience for experts

What is this MPI Forum?

- **An open Forum to discuss MPI**
  - You can join! No membership fee, no perks either
- **Since 2008 meetings every two months for three days (switching to four months and four days)**
  - 5x in the US, once in Europe (with EuroMPI → next week)
- **Votes by organization, eligible after attending two of the three last meetings, often unanimously**
- **Everything is voted twice in two distinct meetings**
  - Tickets as well as chapters

Recommended Development Workflow

1. **Identify a scalable algorithm**
    - Analyze for memory and runtime
2. **Is there a library that can help me?**
    - Computational libraries

      PPM, PBGL, PETSc, PMTL, ScaLAPACK
    - Communication libraries

      AM++, LibNBC
    - Programming Model Libraries

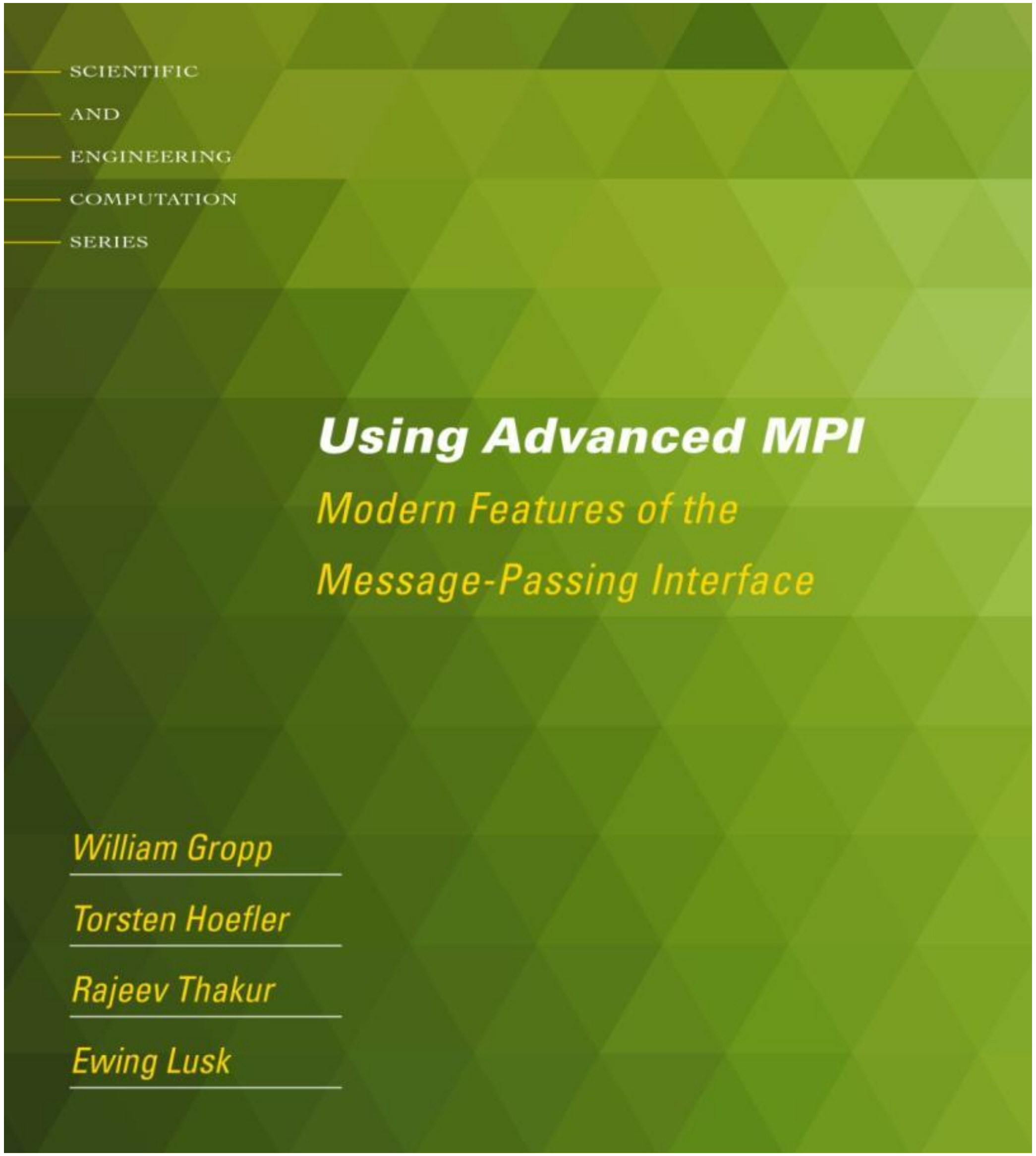
      ADLB, AP
    - Utility Libraries

      HDF5, Boost.MPI
3. **Plan for modularity**
    - Writing (parallel) libraries has numerous benefits

Things to Keep in Mind

- **MPI is an open standardization effort**
  - Talk to us or join the forum
  - There will be a public comment period
- **The MPI standard**
  - Is **free** for everybody
  - **Is not** intended for end-users (no replacement for books and tutorials)
  - **Is** the last instance in MPI questions

Any Deeper Questions – Advanced MPI


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using Advanced MPI
Modern Features of the
Message-Passing Interface

William Gropp
Torsten Hoefler
Rajeev Thakur
Ewing Lusk

includes all of MPI-3.0

appeared November 2014
(on sale on Amazon now)

Section II - Nonblocking and Collective Communication

Nonblocking and Collective Communication

- **Nonblocking communication**
  - Deadlock avoidance
  - Overlapping communication/computation

- **Collective communication**
  - Collection of pre-defined optimized routines

- **Nonblocking collective communication**
  - Combines both advantages
  - System noise/imbalance resiliency
  - Semantic advantages
  - Examples

Nonblocking Communication

- **Semantics are simple:**
  - Function returns no matter what
  - No progress guarantee!

- **E.g., MPI_Isend(<send-args>, MPI_Request *req);**

- **Nonblocking tests:**
  - Test, Testany, Testall, Testsome

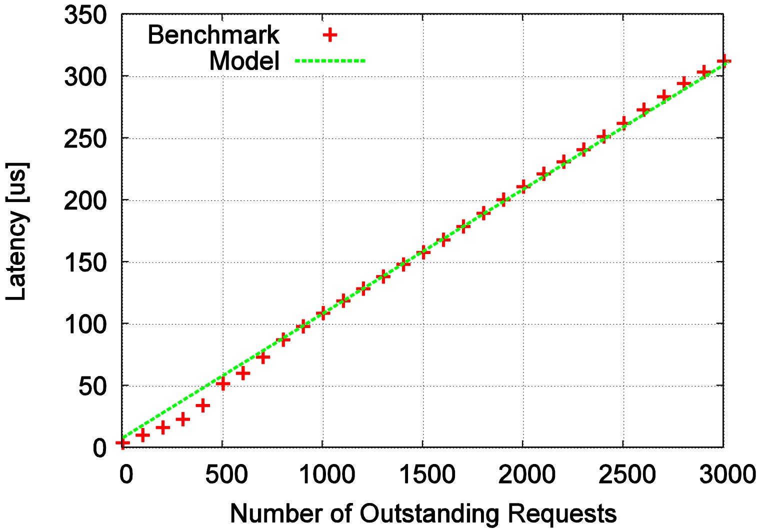
- **Blocking wait:**
  - Wait, Waitany, Waitall, Waitsome

Nonblocking Communication

- **Blocking vs. nonblocking communication**
  - Mostly equivalent, nonblocking has constant request management overhead
  - Nonblocking may have other non-trivial overheads

- **Request queue length**
  - Linear impact on performance
  - E.g., BG/P: 100ns/req

  Tune unexpected queue length!



Nonblocking Communication

- **An (important) implementation detail**
  - Eager vs. Rendezvous

- **Most/All MPIs switch protocols**
  - Small messages are copied to internal remote buffers

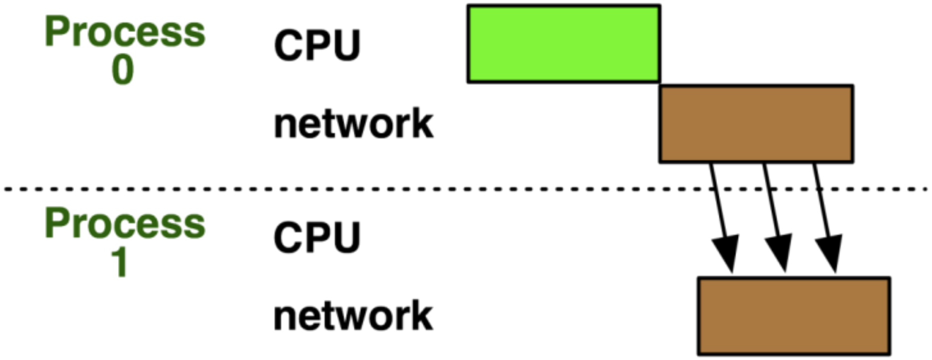
    And then copied to user buffer

    Frees sender immediately (cf. bsend)
  - Large messages wait until receiver is ready

    Blocks sender until receiver arrived
  - Tune eager limits!

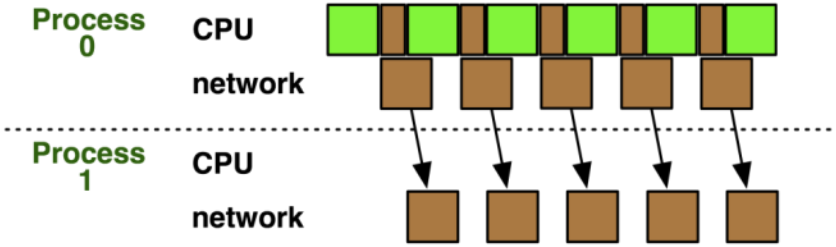
Software Pipelining - Motivation

```
if(r == 0) {
  for(int i=0; i<size; ++i) {
    arr[i] = compute(arr, size);
  }
  MPI_Send(arr, size, MPI_DOUBLE, 1, 99, comm);
} else {
  MPI_Recv(arr, size, MPI_DOUBLE, 0, 99, comm, &stat);
}
```



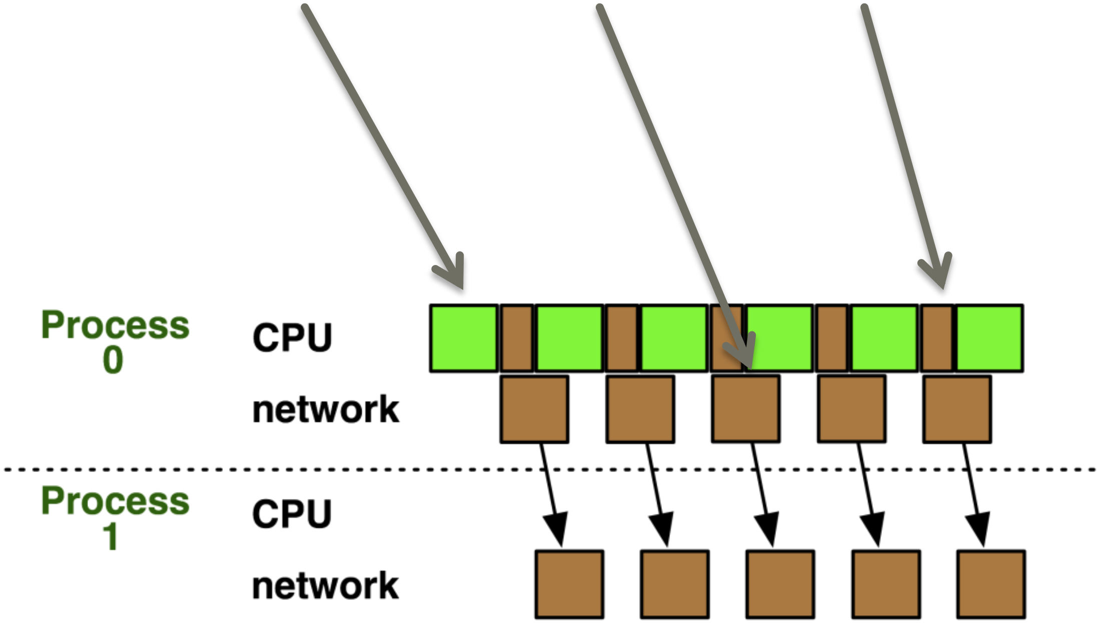
Software Pipelining - Motivation

```
if(r == 0) {
  MPI_Request req=MPI_REQUEST_NULL;
  for(int b=0; b<nblocks; ++b) {
    if(b) {
      if(req != MPI_REQUEST_NULL) MPI_Wait(&req, &stat);
      MPI_Isend(&arr[(b-1)*bs], bs, MPI_DOUBLE, 1, 99, comm, &req);
    }
    for(int i=b*bs; i<(b+1)*bs; ++i) arr[i] = compute(arr, size);
  }
  MPI_Send(&arr[(nblocks-1)*bs], bs, MPI_DOUBLE, 1, 99, comm);
} else {
  for(int b=0; b<nblocks; ++b)
     MPI_Recv(&arr[b*bs], bs, MPI_DOUBLE, 0, 99, comm, &stat);
}
```



A Simple Pipeline Model

- **No pipeline:**
  - $T = T_{comp}(s) + T_{comm}(s) + T_{startc}(s)$
- **Pipeline:**
  - $T = nblocks * \left[\max(T_{comp}(bs), T_{comm}(bs)) + T_{startc}(bs)\right]$

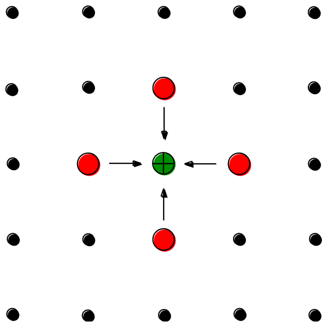


2D Jacobi Example

- **Many 2d electrostatic problems can be reduced to solving Poisson's or Laplace's equation**
  - Solution by finite difference methods
  - $p_{new}(i,j) = (p(i-1,j)+p(i+1,j)+p(i,j-1)+p(i,j+1))/4$
  - natural 2d domain decomposition
  - State of the Art:

    Compute, communicate

    Maybe overlap inner computation



Simplified Serial Code

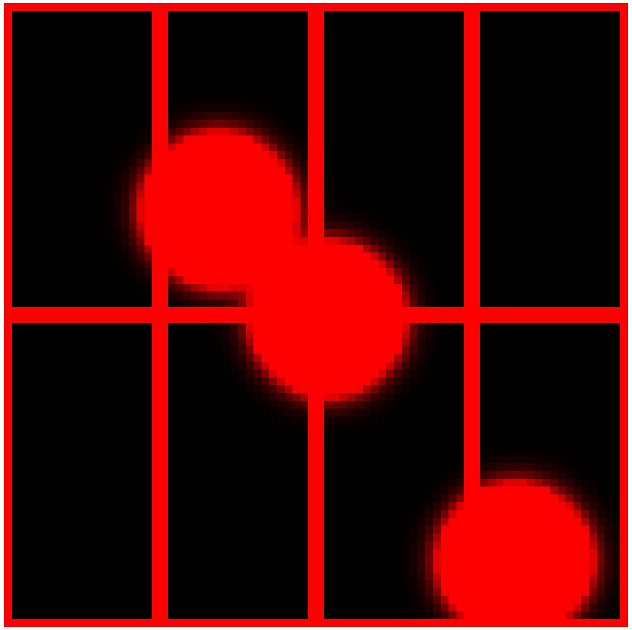
```
for(int iter=0; iter<niters; ++iter) {
    for(int i=1; i<n+1; ++i) {
        for(int j=1; j<n+1; ++j) {
            anew[ind(i,j)] = apply(stencil); // actual computation
            heat += anew[ind(i,j)]; // total heat in system
        }
    }
    for(int i=0; i<nsources; ++i) {
        anew[ind(sources[i][0],sources[i][1])] += energy; // heat source
    }
    tmp=anew; anew=aold; aold=tmp; // swap arrays
}
```

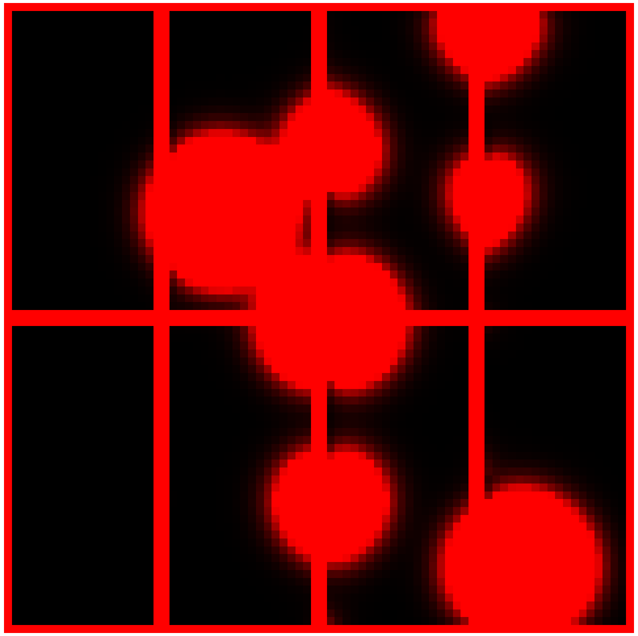
Simple 2D Parallelization

- **Why 2D parallelization?**
  - Minimizes surface-to-volume ratio
- **Specify decomposition on command line (px, py)**
- **Compute process neighbors manually**
- **Add halo zones (depth 1 in each direction)**
- **Same loop with changed iteration domain**
- **Pack halo, communicate, unpack halo**
- **Global reduction to determine total heat**

Source Code Example

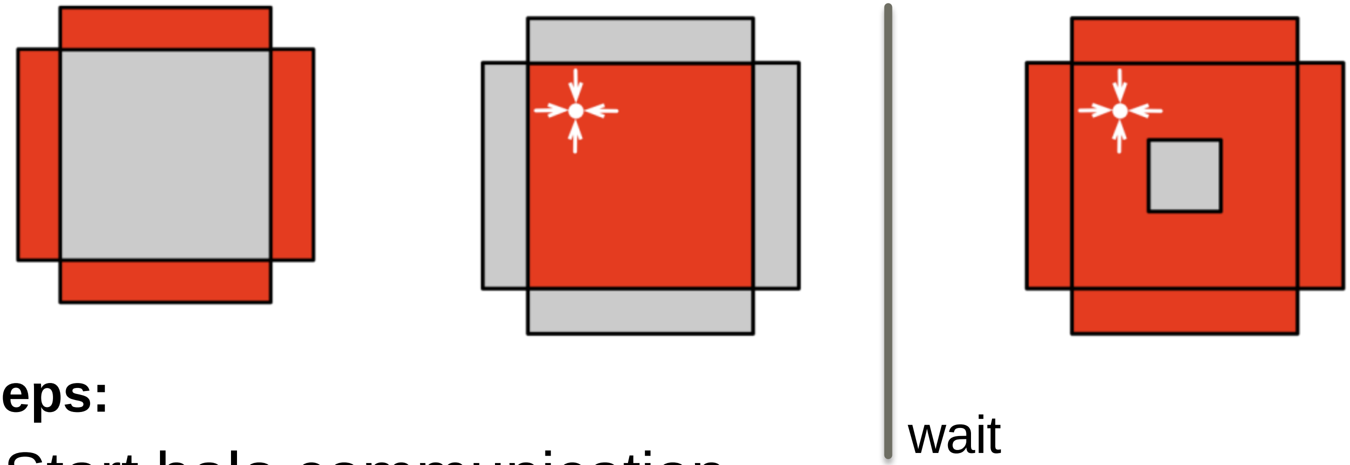
- **Browse through code (stencil_mpi.cpp)**





Stencil Example - Overlap

- **stencil_mpi_ddt_overlap.cpp**



- **Steps:**
  - Start halo communication
  - Compute inner zone
  - Wait for halo communication
  - Compute outer zone
  - Swap arrays

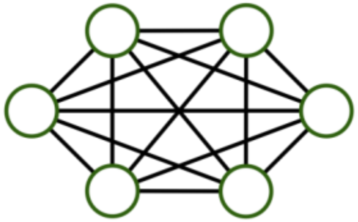
Collective Communication

- **Three types:**
  - Synchronization (Barrier)
  - Data Movement (Scatter, Gather, Alltoall, Allgather)
  - Reductions (Reduce, Allreduce, (Ex)Scan, Reduce_scatter)
- **Common semantics:**
  - no tags (communicators can serve as such)
  - Blocking semantics (return when complete)
  - Not necessarily synchronizing (only barrier and all*)
- **Overview of functions and performance models**

Collective Communication

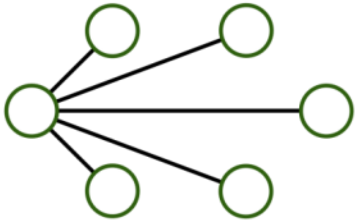
- **Barrier –**
  - Often $\alpha+\beta \log_2 P$

$$\Omega(\log(P))$$



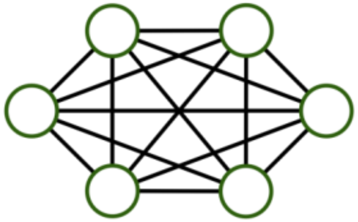
- **Scatter, Gather –**
  - Often $\alpha P+\beta Ps$

$$\Omega(\log(P) + Ps)$$



- **Alltoall, Allgather -**
  - Often $\alpha P+\beta Ps$

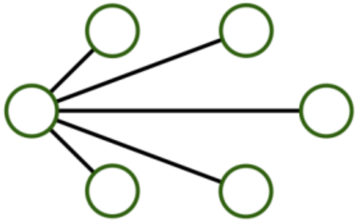
$$\Omega(\log(P) + Ps)$$



Collective Communication

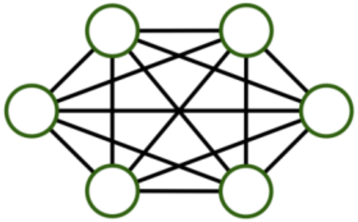
- **Reduce –**
  - Often $\alpha\log_2 P+\beta m+\gamma m$

$$\Omega(\log(P)+s)$$



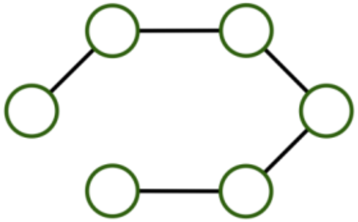
- **Allreduce –**
  - Often $\alpha\log_2 P+\beta m+\gamma m$

$$\Omega(\log(P)+s)$$



- **(Ex)scan –**
  - Often $\alpha P+\beta m+\gamma m$

$$\Omega(\log(P)+s)$$



Nonblocking Collective Communication

- **Nonblocking variants of all collectives**
  - MPI_Ibcast(<bcast args>, MPI_Request *req);
- **Semantics:**
  - Function returns no matter what
  - No guaranteed progress (quality of implementation)
  - Usual completion calls (wait, test) + mixing
  - Out-of order completion
- **Restrictions:**
  - No tags, in-order matching
  - Send and vector buffers may not be touched during operation
  - MPI_Cancel not supported
  - No matching with blocking collectives

TH, Lumsdaine, Rehm: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI, SC07

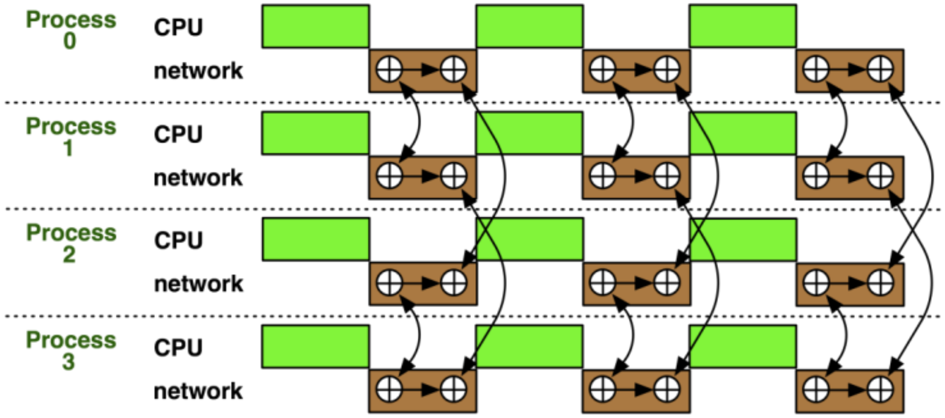
Nonblocking Collective Communication

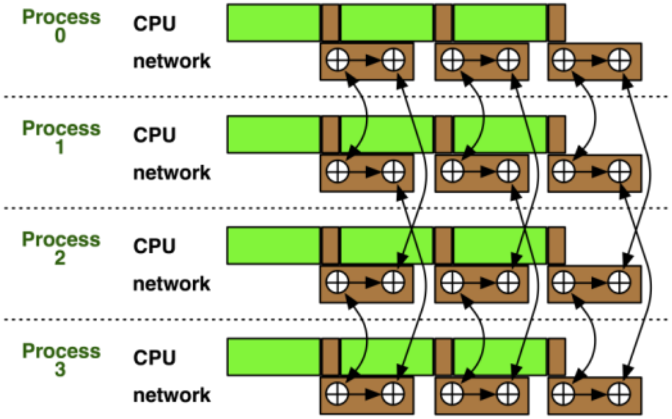
- **Semantic advantages:**
  - Enable asynchronous progression (and manual)
    Software pipelinling
  - Decouple data transfer and synchronization
    Noise resiliency!
  - Allow overlapping communicators
    See also neighborhood collectives
  - Multiple outstanding operations at any time
    Enables pipelining window

TH, Lumsdaine, Rehm: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI, SC07

Nonblocking Collectives Overlap

- **Software pipelining, similar to point-to-point**
  - More complex parameters
  - Progression issues
  - Not scale-invariant





TH, Lumsdaine, Rehm: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI, SC07

Nonblocking Collectives Overlap

- **Complex progression**
    - MPI’s global progress rule!
- **Higher CPU overhead (offloading?)**
- **Differences in asymptotic behavior**
    - Collective time often $\Omega(\log(P) + Ps)$
    - Computation $\mathcal{O}(\frac{N}{P})$

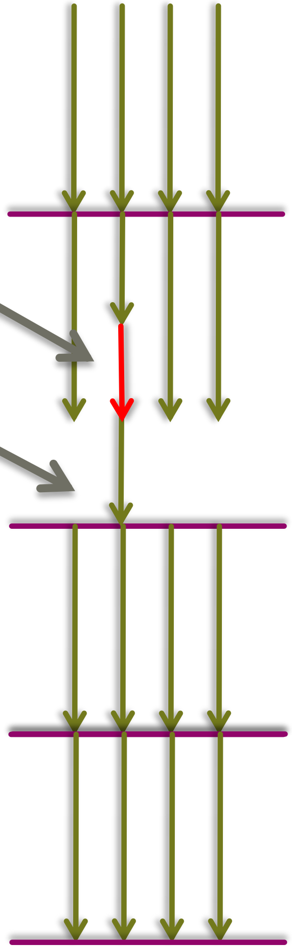
    - → Performance modeling ☺
    - One term often dominates and complicates overlap

TH, Gottschling, Lumsdaine: Leveraging Non-blocking Collective Communication in High-performance Applications, SPAA’08

System Noise – Introduction

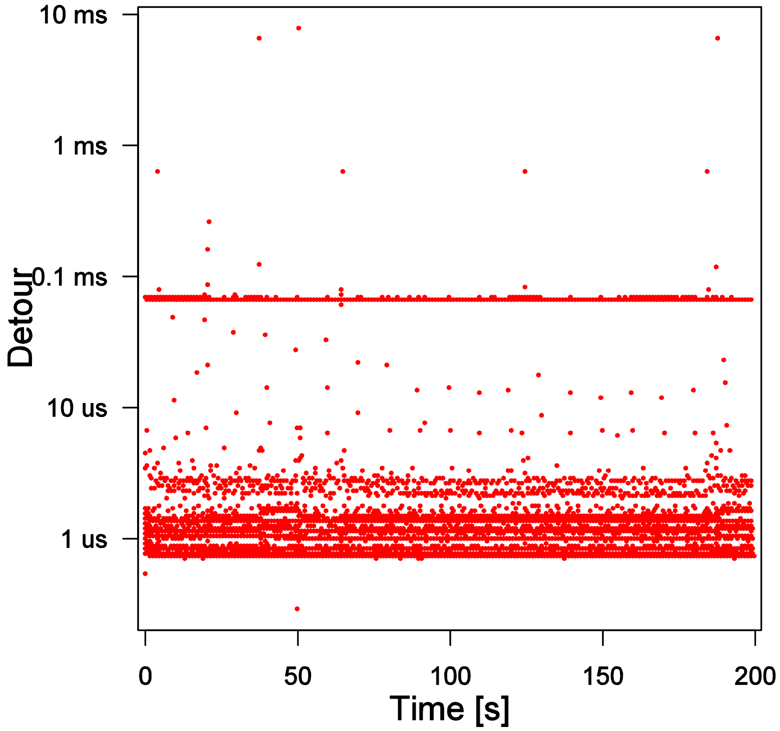
- **CPUs are time-shared**
  - Deamons, interrupts, etc. steal cycles
  - No problem for single-core performance

    Maximum seen: 0.26%, average: 0.05% overhead
  - “Resonance” at large scale (Petrini et al ’03)
- **Numerous studies**
  - Theoretical (Agarwal’05, Tsafrir’05, Seelam’10)
  - Injection (Beckman’06, Ferreira’08)
  - Simulation (Sottile’04)



TH, Schneider, Lumsadine: Characterizing the Influence of System Noise on Large-Scale Applications by Simulation, SC10

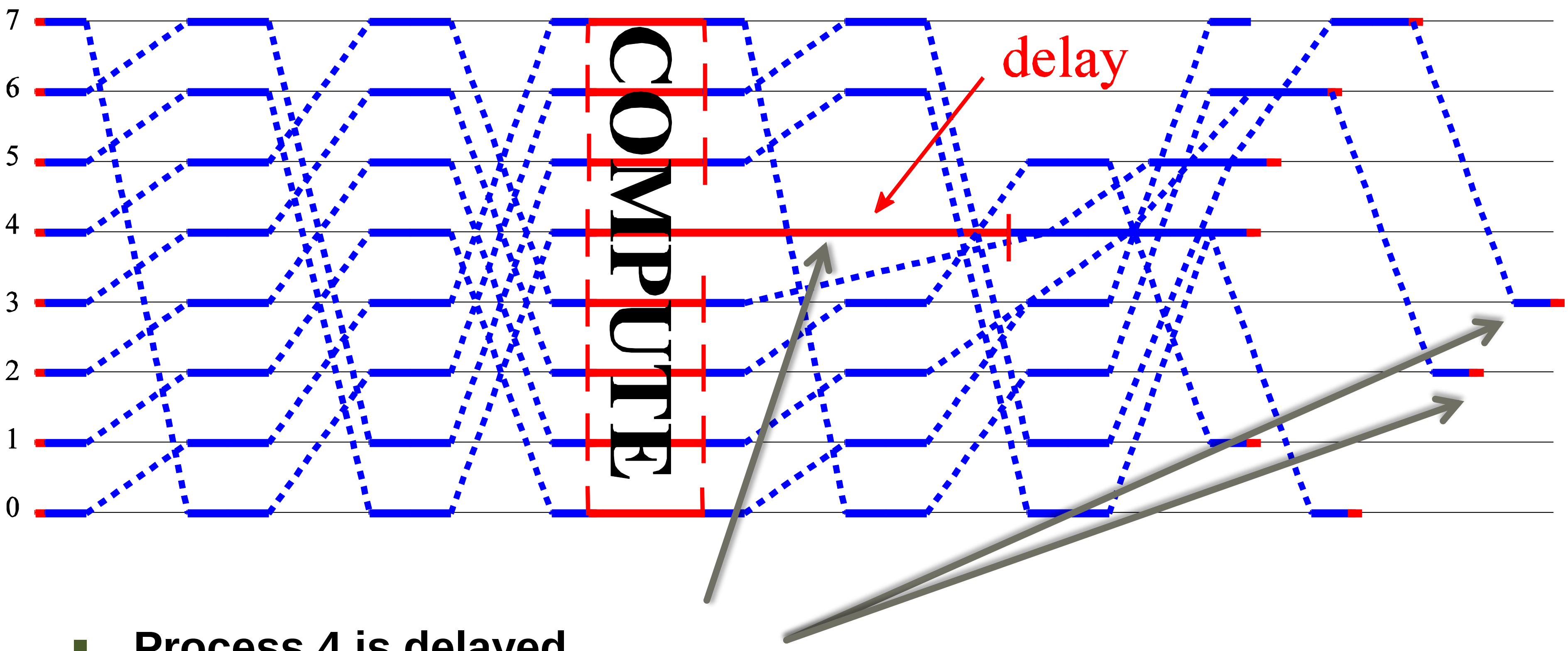
Measurement Results – Cray XE



- **Resolution: 32.9 ns, noise overhead: 0.02%**

TH, Schneider, Lumsdaine: Characterizing the Influence of System Noise on Large-Scale Applications by Simulation, SC10

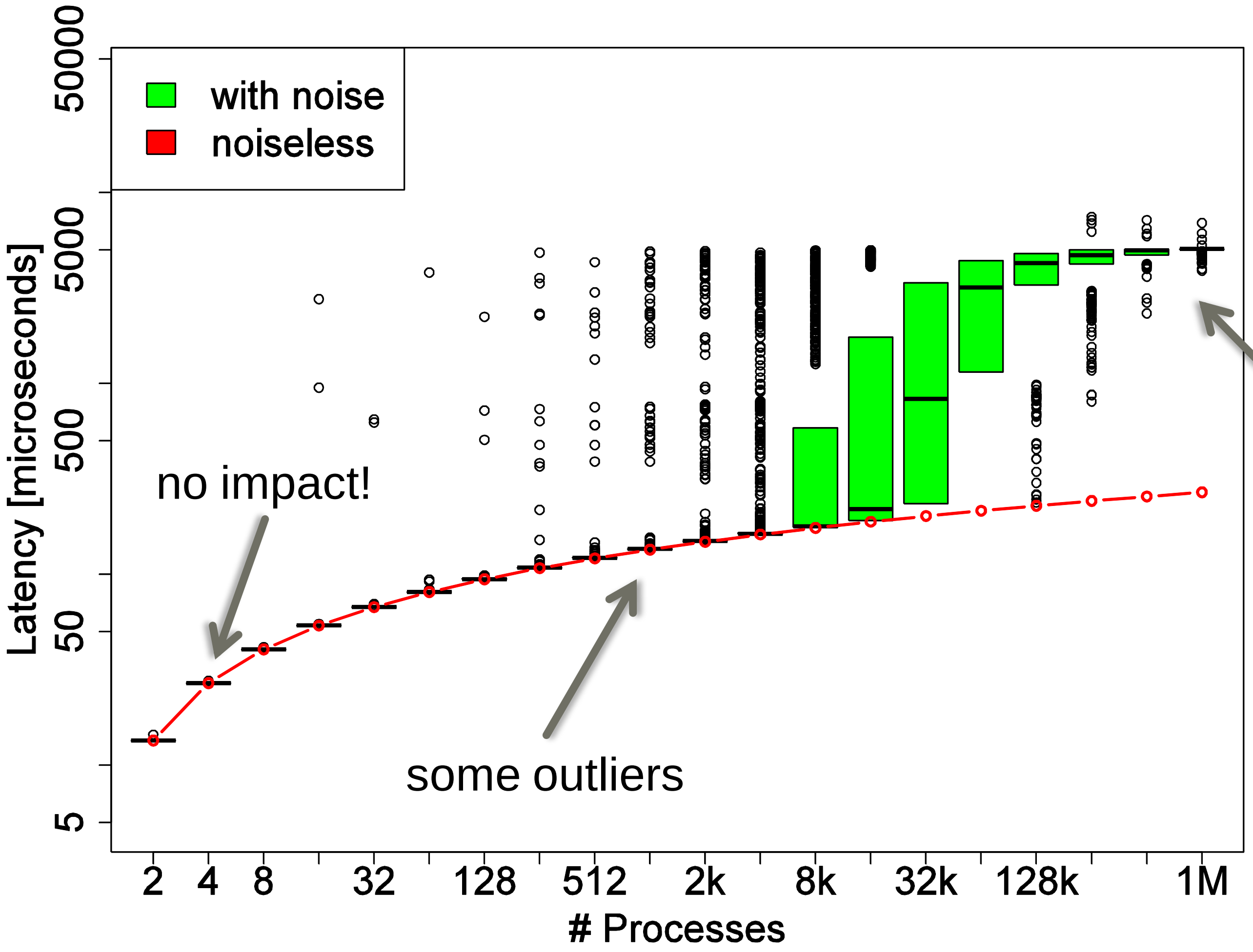
A Noisy Example – Dissemination



- **Process 4 is delayed**
  - Noise propagates “*wildly*” (of course deterministic)

TH, Schneider, Lumsdaine: Characterizing the Influence of System Noise on Large-Scale Applications by Simulation, SC10

Single Byte Dissemination on Jaguar



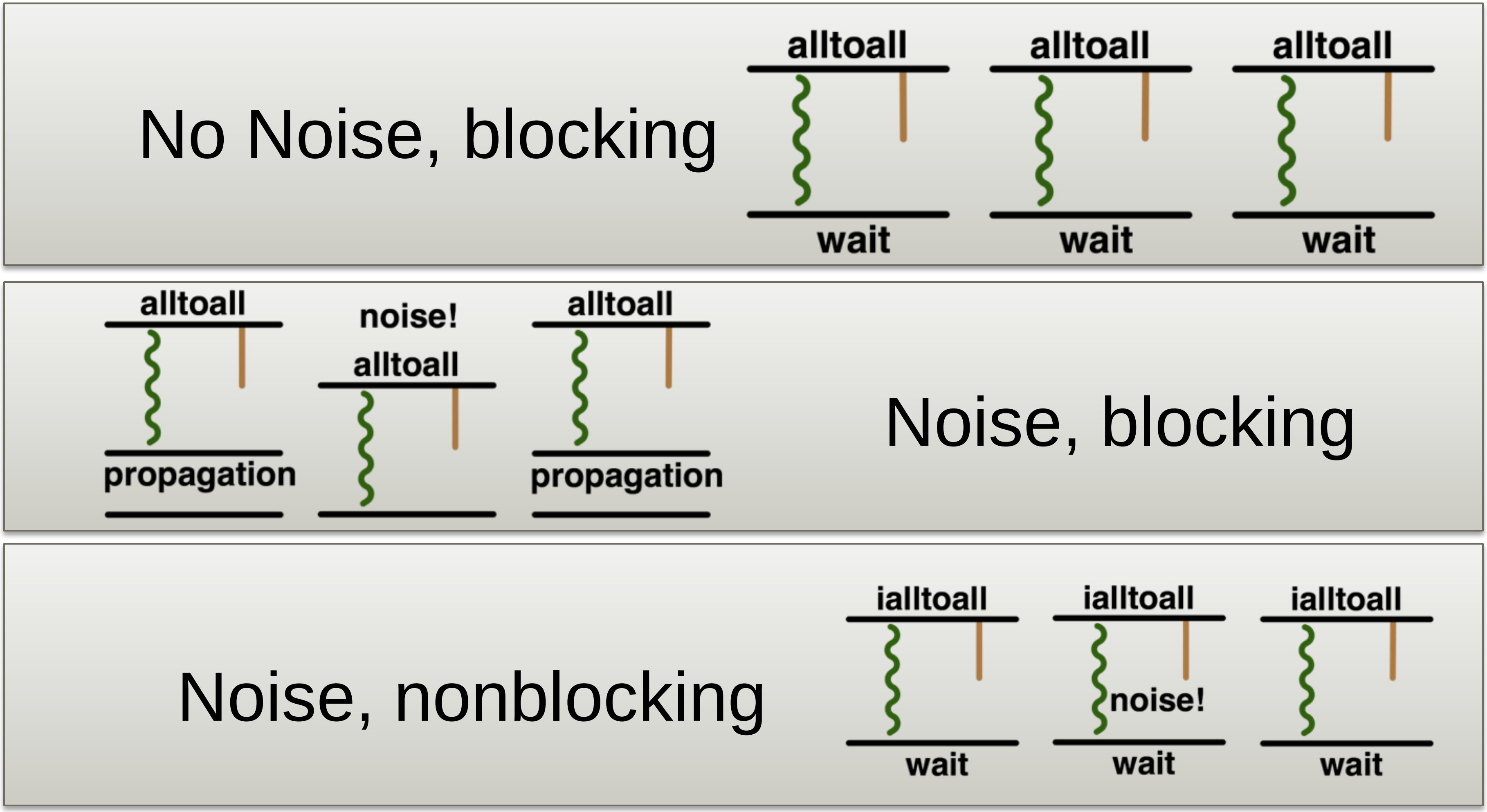
no impact!

some outliers

deterministic slowdown (noise bottleneck)

TH, Schneider, Lumsdaine: *Characterizing the Influence of System Noise on Large-Scale Applications by Simulation*, SC10

Nonblocking Collectives vs. Noise



No Noise, blocking

alltoall alltoall alltoall

wait wait wait

alltoall noise! alltoall alltoall

propagation propagation

Noise, blocking

Noise, nonblocking

ialltoall ialltoall ialltoall

noise!

wait wait wait

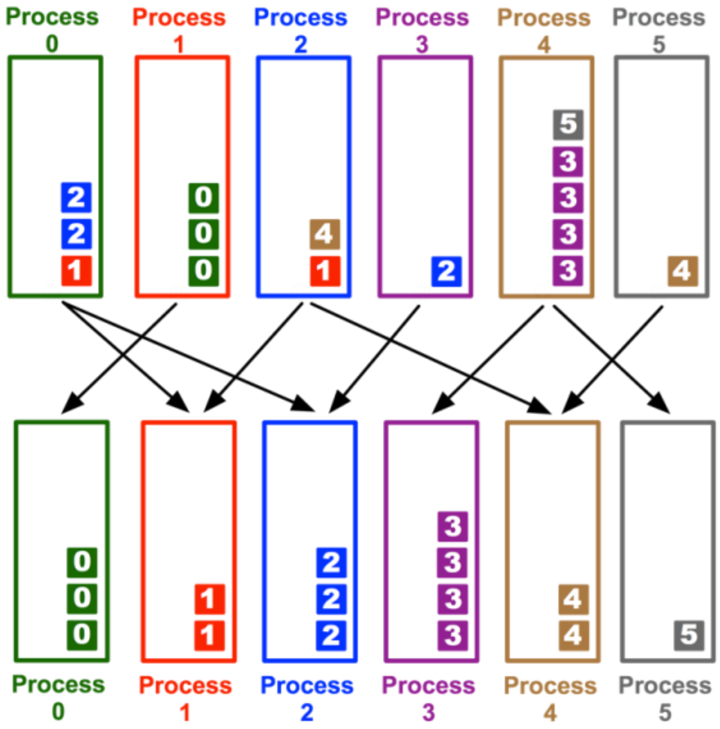
TH, Schneider, Lumsdaine: Characterizing the Influence of System Noise on Large-Scale Applications by Simulation, SC10

A Non-Blocking Barrier?

- **What can that be good for? Well, quite a bit!**
- **Semantics:**
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary
- **Uses:**
  - Overlap barrier latency (small benefit)
  - Use the split semantics! Processes **notify** non-collectively but **synchronize** collectively!

A Semantics Example: DSDE

- **Dynamic Sparse Data Exchange**
  - Dynamic: comm. pattern varies across iterations
  - Sparse: number of neighbors is limited ($\mathcal{O}(\log P)$)
  - Data exchange: only senders know neighbors



TH, Siebert, Lumsdaine: Scalable Communication Protocols for Dynamic Sparse Data Exchange, PPoPP''10

Dynamic Sparse Data Exchange (DSDE)

- **Main Problem: metadata**
  - Determine who wants to send how much data to me
    (I must post receive and reserve memory)

OR:

  - Use MPI semantics:

    Unknown sender

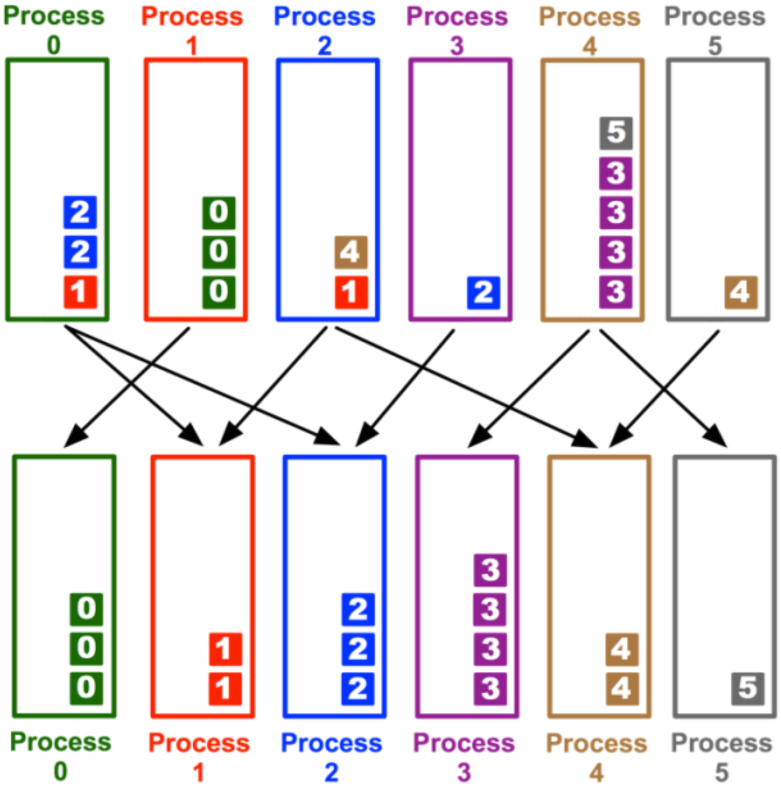
    MPI_ANY_SOURCE

    Unknown message size

    MPI_PROBE

    Reduces problem to counting the number of neighbors

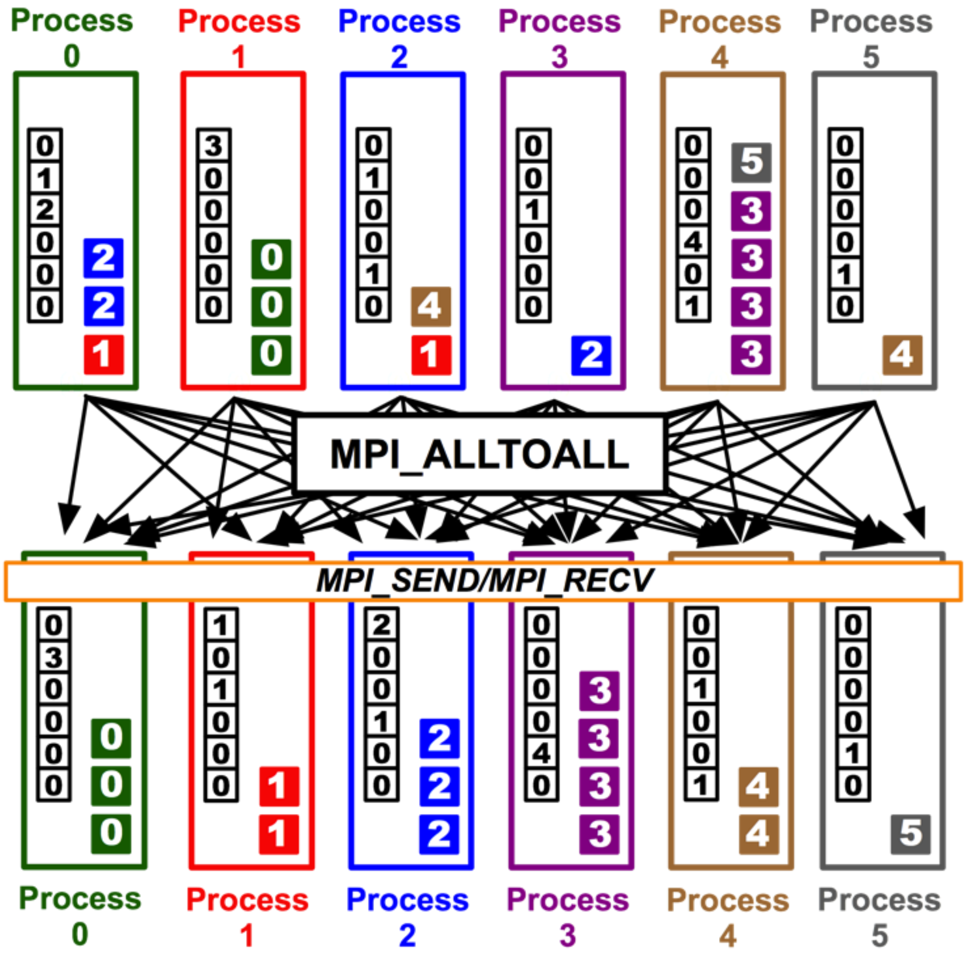
    Allow faster implementation!



TH, Siebert, Lumsdaine: Scalable Communication Protocols for Dynamic Sparse Data Exchange, PPoPP"10

Using Alltoall (PEX)

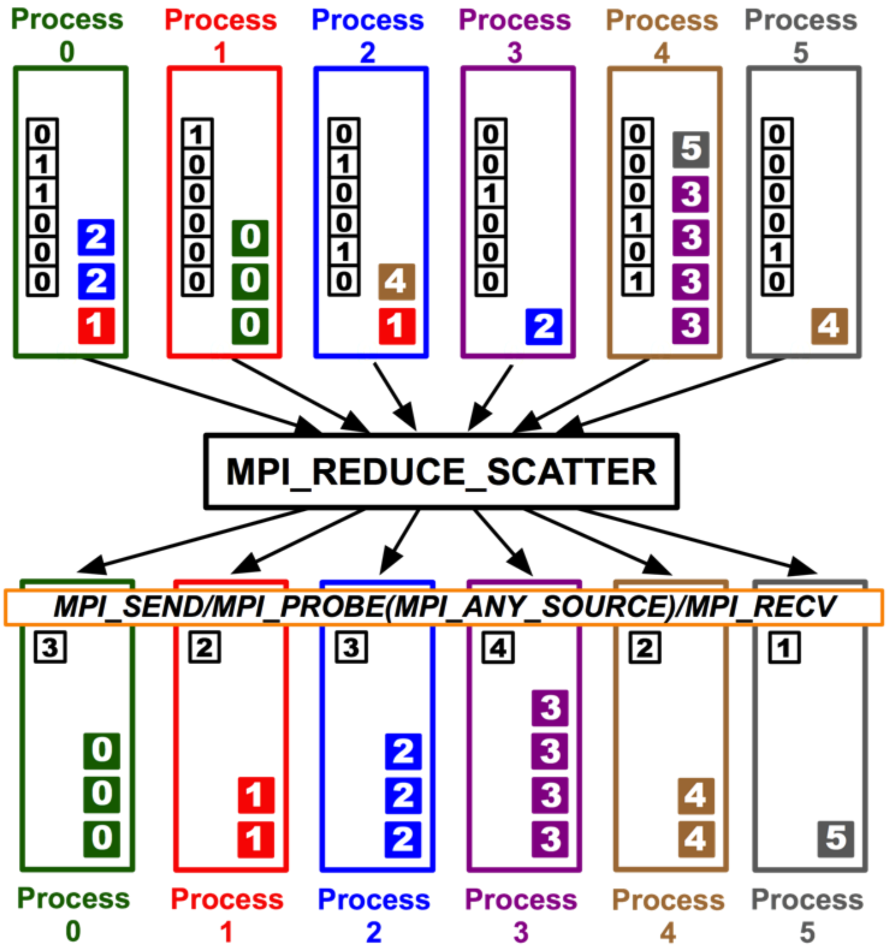
- **Based on Personalized Exchange ($\Theta(P)$)**
  - Processes exchange metadata (sizes) about neighborhoods with all-to-all
  - Processes post receives afterwards
  - Most intuitive but least performance and scalability!



TH, Siebert, Lumsdaine: Scalable Communication Protocols for Dynamic Sparse Data Exchange, PPoPP"10

Reduce_scatter (PCX)

- **Bases on Personalized Census ($\Theta(P)$)**
  - Processes exchange metadata (counts) about neighborhoods with reduce_scatter
  - Receivers checks with wildcard MPI_IPROBE and receives messages
  - Better than PEX but non-deterministic!

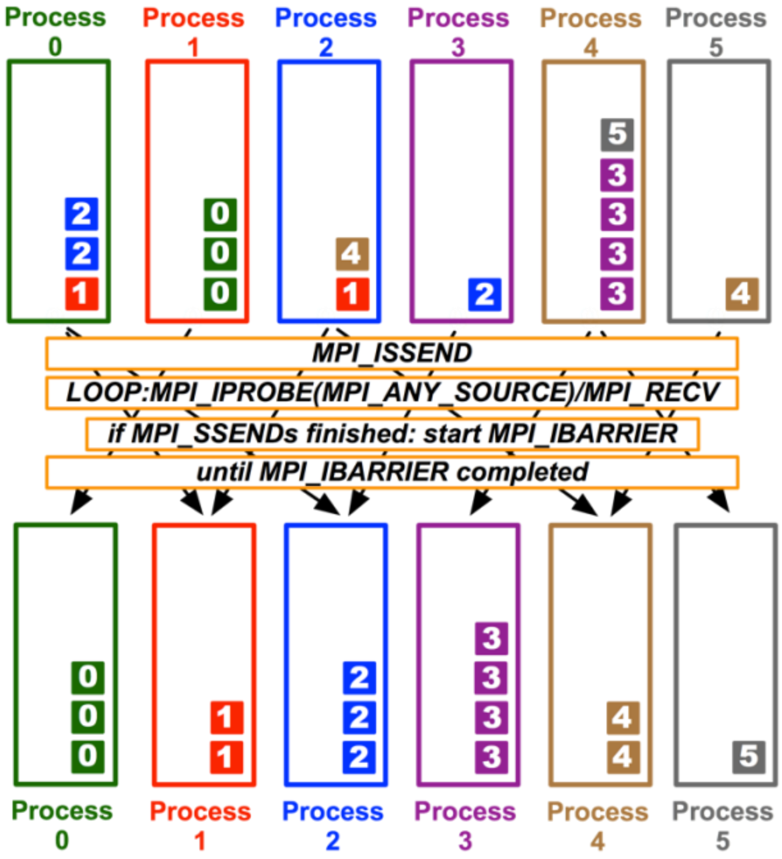


TH, Siebert, Lumsdaine: Scalable Communication Protocols for Dynamic Sparse Data Exchange, PPoPP"10

MPI_Ibarrier (NBX)

- **Complexity - census (barrier): ($\Theta(\log(P))$)**
  - Combines metadata with actual transmission
  - Point-to-point synchronization
  - Continue receiving until barrier completes
  - Processes start coll. synch. (barrier) when p2p phase ended

    barrier = distributed marker!
  - Better than PEX, PCX, RSX!

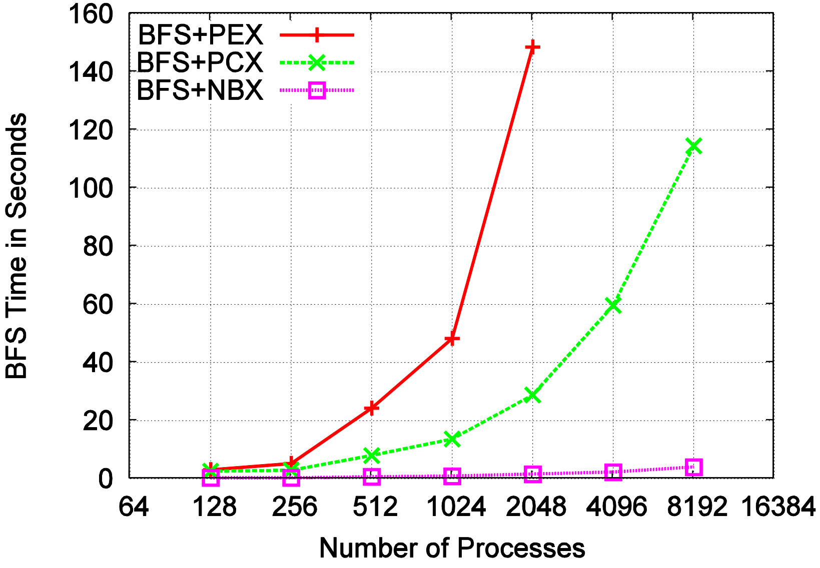


MPI_ISSEND
LOOP:MPI_IPROBE(MPI_ANY_SOURCE)/MPI_RECV
if MPI_SSENDs finished: start MPI_IBARRIER
until MPI_IBARRIER completed

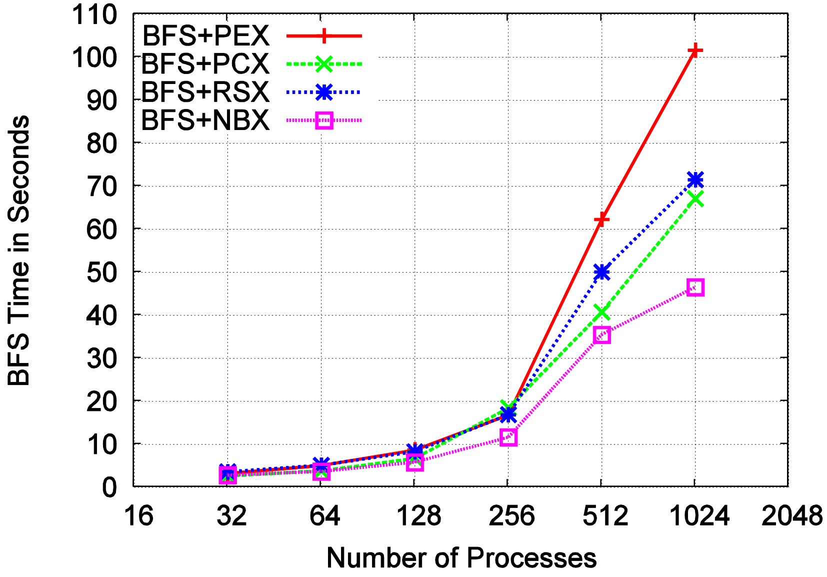
TH, Siebert, Lumsdaine: Scalable Communication Protocols for Dynamic Sparse Data Exchange, PPoPP'10

Parallel Breadth First Search

- **On a clustered Erdős-Rényi graph, weak scaling**
  - 6.75 million edges per node (filled 1 GiB)



BlueGene/P – with HW barrier!



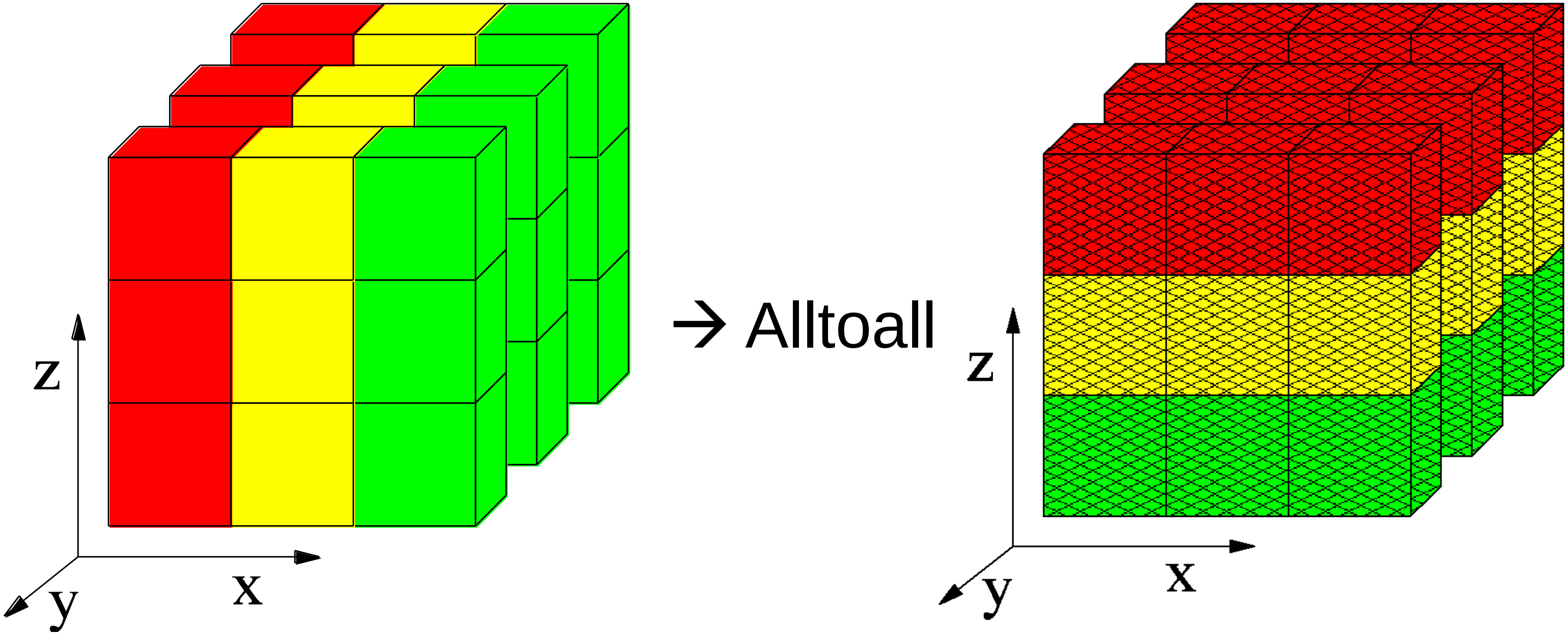
Myrinet 2000 with LibNBC

- **HW barrier support is significant at large scale!**

TH, Siebert, Lumsdaine: Scalable Communication Protocols for Dynamic Sparse Data Exchange, PPoPP"10

Parallel Fast Fourier Transform

- **1D FFTs in all three dimensions**
  - Assume 1D decomposition (each process holds a set of planes)
  - Best way: call optimized 1D FFTs in parallel → alltoall



→ Alltoall

- Red/yellow/green are the (three) different processes!

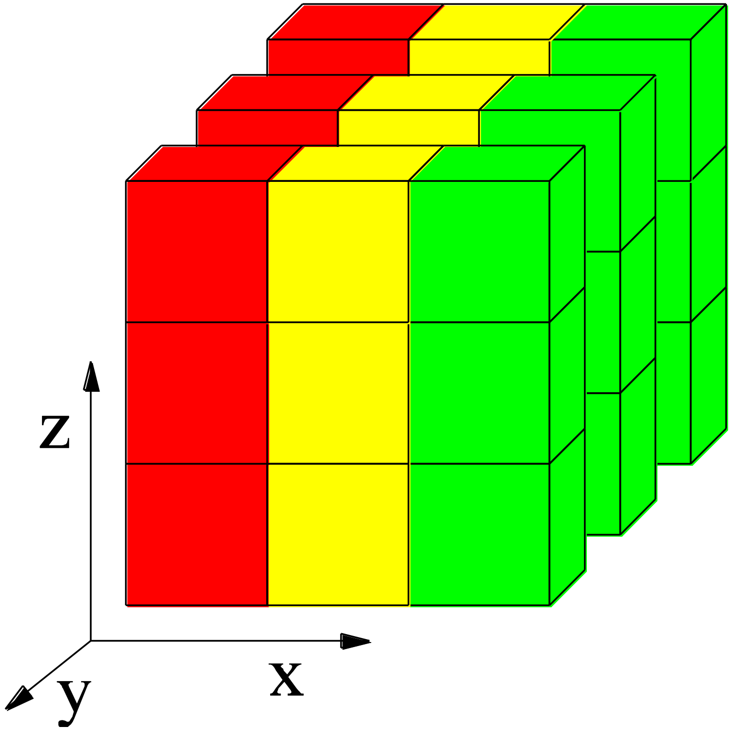
A Complex Example: FFT

```
for(int x=0; x<n/p; ++x) 1d_fft(/* x-th stencil */);

// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose

for(int y=0; y<n/p; ++y) 1d_fft(/* y-th stencil */);

// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose
```

TH, Gottschling, Lumsdaine: Leveraging Non-blocking Collective Communication in High-performance Applications, , SPAA'08

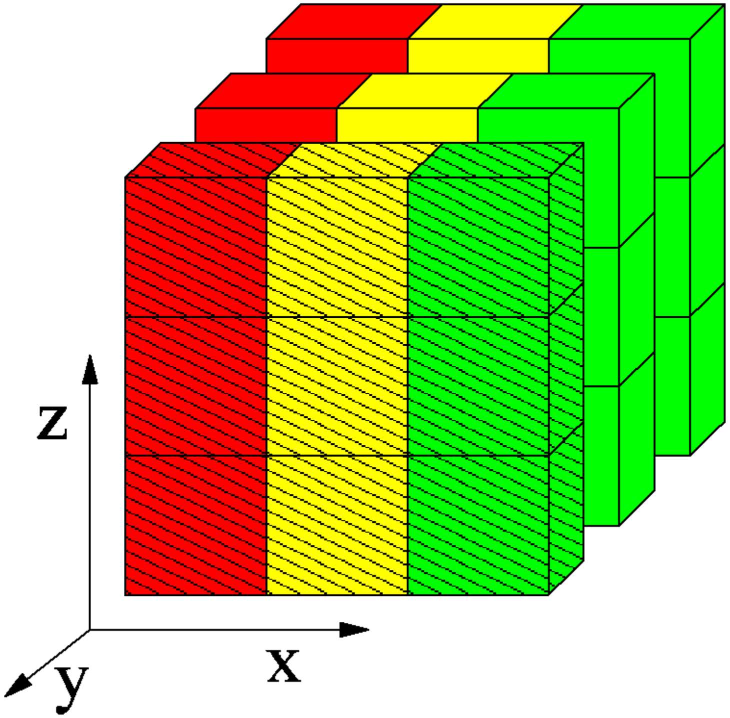
Parallel Fast Fourier Transform

- **Data already transformed in y-direction**



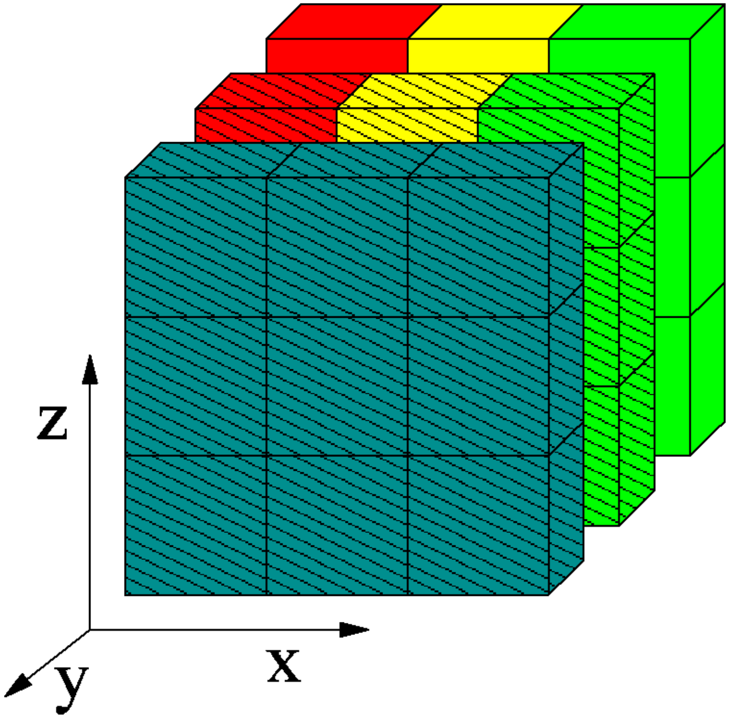
Parallel Fast Fourier Transform

- **Transform first y plane in z**



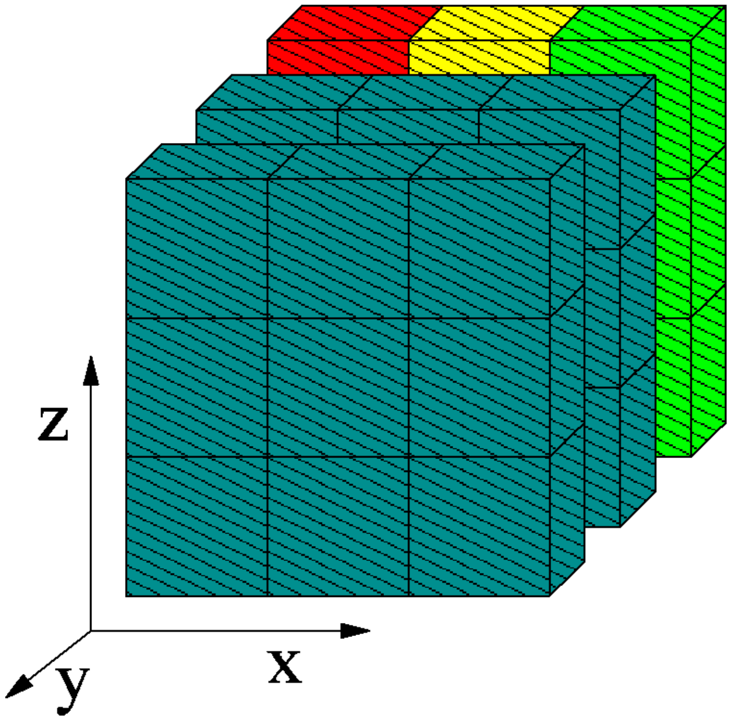
Parallel Fast Fourier Transform

- **Start ialltoall and transform second plane**



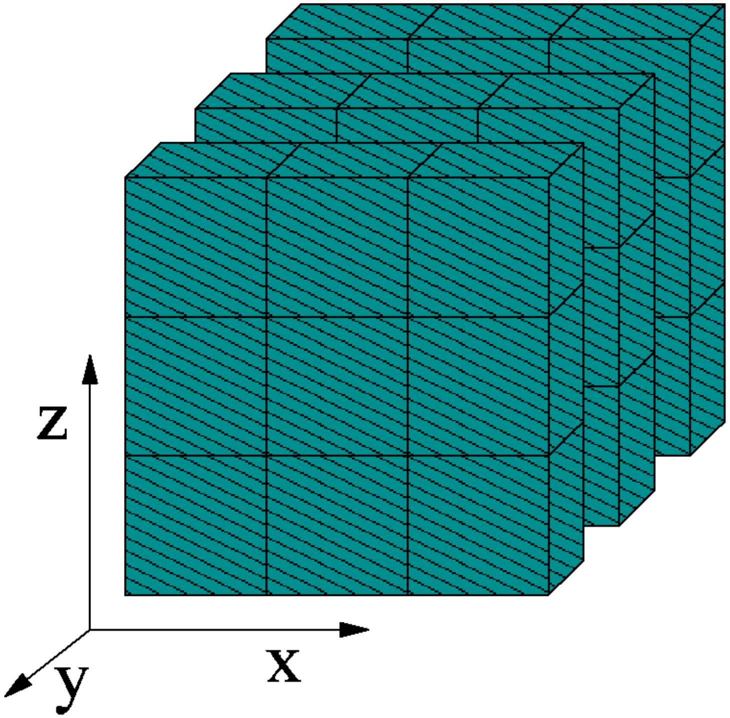
Parallel Fast Fourier Transform

- **Start ialltoall (second plane) and transform third**



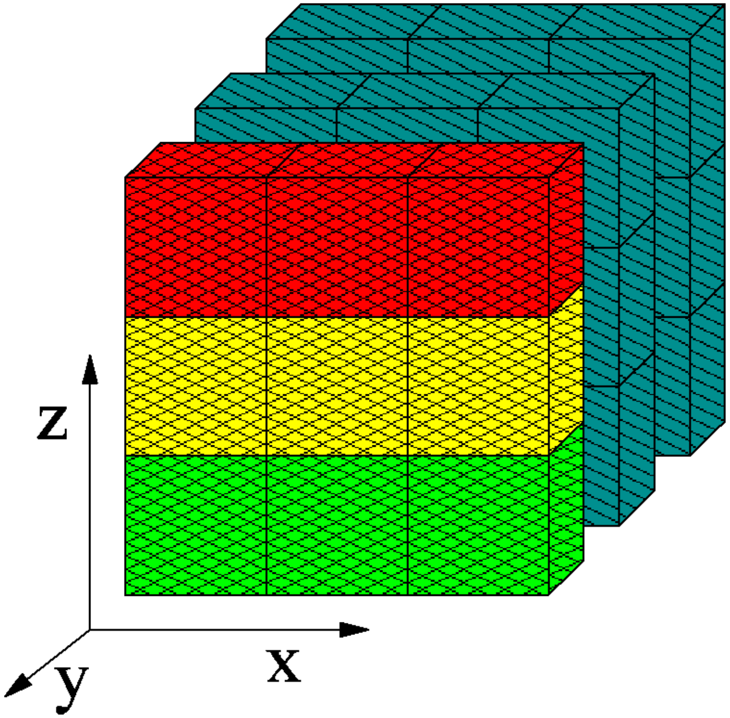
Parallel Fast Fourier Transform

- **Start ialltoall of third plane and …**



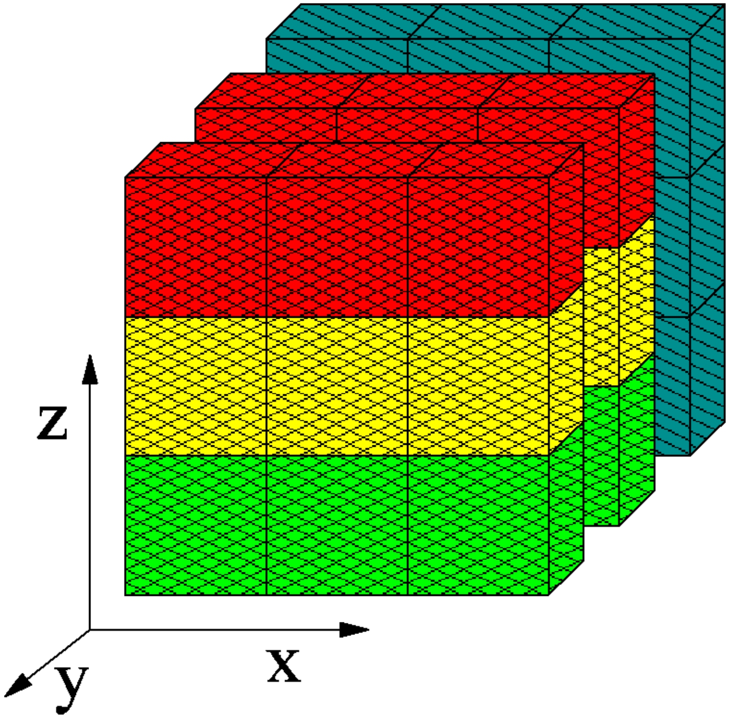
Parallel Fast Fourier Transform

- **Finish ialltoall of first plane, start x transform**



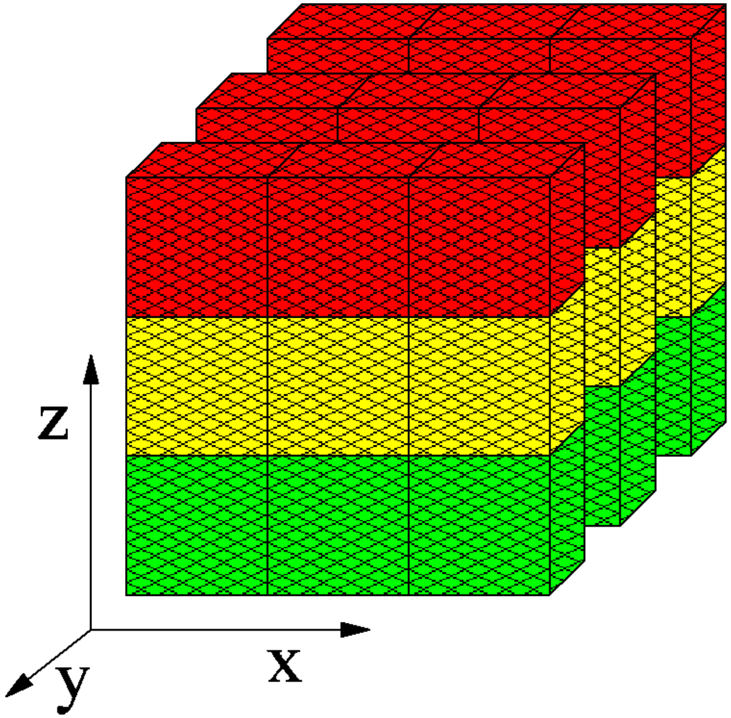
Parallel Fast Fourier Transform

- **Finish second ialltoall, transform second plane**



Parallel Fast Fourier Transform

- **Transform last plane → done**



FFT Software Pipelining

```
MPI_Request req[nb];
for(int b=0; b<nb; ++b) { // loop over blocks
  for(int x=b*n/p/nb; x<(b+1)n/p/nb; ++x) 1d_fft(/* x-th stencil*/);

  // pack b-th block of data for alltoall
  MPI_Ialltoall(&in, n/p*n/p/bs, cplx_t, &out, n/p*n/p, cplx_t, comm, &req[b]);
}
MPI_Waitall(nb, req, MPI_STATUSES_IGNORE);

// modified unpack data from alltoall and transpose
for(int y=0; y<n/p; ++y) 1d_fft(/* y-th stencil */);
// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose
```

TH, Gottschling, Lumsdaine: Leveraging Non-blocking Collective Communication in High-performance Applications, , SPAA’08

Nonblocking And Collective Summary

- **Nonblocking comm does two things:**
  - Overlap and relax synchronization
- **Collective comm does one thing**
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance
- **They can be composed**
  - E.g., software pipelining

Section III - One Sided Communication

One Sided Communication

- **Terminology**
- **Memory exposure**
- **Communication**
- **Accumulation**
    - Ordering, atomics
- **Synchronization**
- **Shared memory windows**
- **Memory models & semantics ☺**

One Sided Communication – The Shock

- **The syntax is weird, really!**
  - It grew – MPI-3.0 is backwards compatible!
- **Think PGAS (with a library interface)**
  - Remote memory access (put, get, accumulates)
- **Forget locks ☺**
  - Win_lock_all is not a lock, opens an epoch
- **Think transactional memory with optional isolation ;-)**
  - That’s really what “lock” means (lock/unlock can be like an atomic region, does not necessarily “lock” anything)
- **Decouple transfers from synchronization**
  - Separate transfer and synch functions

One Sided Communication – Terms

- **Origin process: Process with the source buffer, initiates the operation**
- **Target process: Process with the destination buffer, does not explicitly call communication functions**
- **Epoch: Virtual time where operations are in flight. Data is consistent after new epoch is started.**
  - Access epoch: rank acts as origin for RMA calls
  - Exposure epoch: rank acts as target for RMA calls
- **Ordering: only for accumulate operations: order of messages between two processes (default: in order, can be relaxed)**
- **Assert: assertions about how One Sided functions are used, “fast” optimization hints, cf. Info objects (slower)**

One Sided Overview

- **Creation**
  - Expose memory collectively - Win_create
  - Allocate exposed memory – Win_allocate
  - Dynamic memory exposure – Win_create_dynamic
- **Communication**
  - Data movement (put, get, rput, rget)
  - Accumulate (acc, racc, get_acc, rget_acc, fetch&op, cas)
- **Synchronization**
  - Active - Collective (fence); Group (PSCW)
  - Passive - P2P (lock/unlock); One epoch (lock _all)

Memory Exposure

```
MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
MPI_Comm comm, MPI_Win *win)
```

- **Exposes consecutive memory (base, size)**
- **Collective call**
- **Info args:**
  - no_locks – user asserts to not lock win
  - accumulate_ordering – comma-separated rar, war, raw, waw
  - accumulate_ops – same_op or same_op_no_op (default) – assert used ops for related accumulates

```
MPI_Win_free(MPI_Win *win)
```

Memory Exposure

```
MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
MPI_Comm comm, void *baseptr, MPI_Win *win)
```

- **Similar to win_create but allocates memory**
  - Should be used whenever possible!
  - May consume significantly less resources
- **Similar info arguments plus**
  - same_size – if true, user asserts that size is identical on all calling processes
- **Win_free will deallocate memory!**
  - Be careful ☺

Memory Exposure

```
MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win
*win)
```

- **Coll. memory exposure may be cumbersome**
    - Especially for irregular applications
- **Win_create_dynamic creates a window with no memory attached**

```
MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
MPI_Win_detach(MPI_Win win, const void *base)
```

- **Register non-overlapping regions locally**
- **Addresses are communicated for remote access!**
    - MPI_Aint will be big enough on heterogeneous systems

One Sided Communication

```
MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)
```

- **Two similar communication functions:**
  - Put, Get
  - Nonblocking, bulk completion at end of epoch
- **Conflicting accesses are not erroneous**
  - But outcome is undefined!
  - One exception: polling on a single byte in the unified model (for fast synchronization)

One Sided Communication

`MPI_Rput(…, MPI_Request *request)`

- **MPI_Rput, MPI_Rget for request-based completion**
  - Also non-blocking but return request
  - Expensive for each operation (vs. bulk completion)
- **Only for local buffer consistency**
  - Get means complete!
  - Put means buffer can be re-used, nothing known about remote completion

One Sided Accumulation

MPI_Accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

- **Remote accumulations (only predefined ops)**
  - Replace value in target buffer with accumulated
  - MPI_REPLACE to emulate MPI_Put
- **Allows for non-recursive derived datatypes**
  - No overlapping entries at target (datatype)
- **Conflicting accesses are allowed!**
  - Ordering rules apply

One Sided Accumulation

MPI_Get_accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, void *result_addr, int result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

- **MPI's generalized fetch and add**
  - 12 arguments ☺
  - MPI_REPLACE allows for fetch & set
  - New op: MPI_NO_OP to emulate get
- **Accumulates `origin` into the `target` , returns content before accumulation in `result`**
  - Atomically of course

One Sided Accumulation

```
MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
MPI_Op op, MPI_Win win)
```

- **Get_accumulate may be very slow (needs to cover many cases, e.g., large arrays etc.)**
  - Common use-case is single element fetch&op
  - Fetch_and_op offers relevant subset of Get_acc
- **Very similar to Get_accumulate**
  - Same semantics, just more limited interface
  - No request-based version

One Sided Accumulation

MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Win win)

- **CAS for MPI (no CAS2 but can be emulated)**
- **Single element, binary compare (!)**
- **Compares `compare` buffer with `target` and replaces value at `target` with `origin` if compare and target are identical. Original target value is returned in `result`.**

Accumulation Semantics

- **Accumulates allow concurrent access!**
  - Put/Get does not! They’re not atomic
- **Emulating atomic put/get**
  - Put = MPI_Accumulate(…, op=MPI_REPLACE, …)
  - Get = MPI_Get_accumulate(…, op=MPI_NO_OP, …)
  - Will be slow (thus we left it ugly!)
- **Ordering modes**
  - Default ordering allows “no surprises” (cf. UPC)
  - Can (should) be relaxed with info (accumulate_ordering = raw, waw, rar, war) during window creation

Synchronization Modes

- **Active target mode**
  - Target ranks are calling MPI
  - Either BSP-like collective: MPI_Win_fence
  - Or group-wise (cf. neighborhood collectives): PSCW
- **Passive target mode**
  - Lock/unlock: no traditional lock, more like TM (without rollback)
  - Lockall: locking all processes isn’t really a lock ☺

MPI_Win_fence Synchronization

`MPI_Win_fence(int assert, MPI_Win win)`

- **Collectively synchronizes all RMA calls on win**
- **All RMA calls started before fence will complete**
  - Ends/starts access and/or exposure epochs
- **Does not guarantee barrier semantics (but often synchronizes)**
- **Assert allows optimizations, is usually 0**
  - MPI_MODE_NOPRECEDE if no communication (neither as origin or destination) is outstanding on win

PSCW Synchronization

```
MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
MPI_Win_complete(MPI_Win win)
MPI_Win_wait(MPI_Win win)
```

- **Specification of access/exposure epochs separately:**
  - Post: start exposure epoch to group, nonblocking
  - Start: start access epoch to group, may wait for post
  - Complete: finish prev. access epoch, origin completion only (not target)
  - Wait: will wait for complete, completes at (active) target
- **As asynchronous as possible**

Lock/Unlock Synchronization

```
MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
MPI_Win_unlock(int rank, MPI_Win win)
```

- **Initiates RMA access epoch to rank**
  - No concept of exposure epoch
- **Unlock closes access epoch**
  - Operations have completed at origin and target
- **Type:**
  - Exclusive: no other process may hold lock to rank
    More like a real lock, e.g., for local accesses
  - Shared: other processes may hold lock

Lock_all Synchronization

```
MPI_Win_lock_all(int assert, MPI_Win win)
MPI_Win_unlock_all(MPI_Win win)
```

- **Starts a shared access epoch from origin to all ranks!**
  - Not collective!
- **Does not really lock anything**
  - Opens a different mode of use, see following slides!

Synchronization Primitives (passive)

```
MPI_Win_flush(int rank, MPI_Win win)
MPI_Win_flush_all(MPI_Win win)
```

- **Flush/Flush_all**
- **Completes all outstanding operations at the target rank (or all) at origin and target**
    - Only in passive target mode

```
MPI_Win_flush_local(int rank, MPI_Win win)
MPI_Win_flush_local_all(MPI_Win win)
```

- **Completes all outstanding operations at the target rank (or all) at origin (buffer reuse)**
    - Only in passive target mode

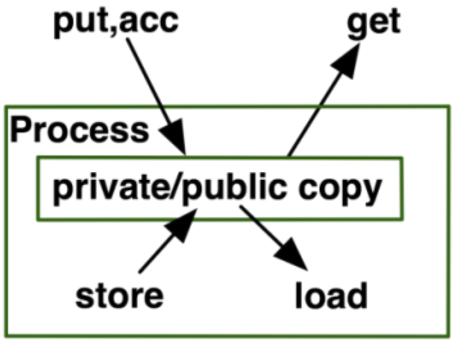
Synchronization Primitives (passive)

```
MPI_Win_sync(MPI_Win win)
```

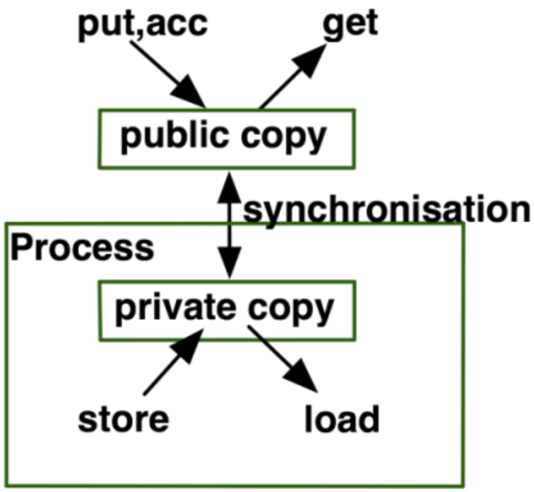
- **Synchronizes private and public window copies**
  - Same as closing and opening access and exposure epochs on the window
  - Does not complete any operations though!
- **Cf. memory barrier**

Memory Models

- **MPI offers two memory models:**
  - Unified: public and private window are identical
  - Separate: public and private window are separate
- **Type is attached as attribute to window**
  - MPI_WIN_MODEL



MPI_UNIFIED



MPI_SEPARATE

Separate Semantics

- **Very complex, rules-of-thumb at target:**

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	X	X
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	X	NOVL	NOVL	NOVL
Acc	NOVL	X	NOVL	NOVL	OVL+NOVL

  - OVL – overlapping
  - NOVL - non-overlapping
  - X - undefined

Credits: RMA Working Group, MPI Forum

Unified Semantics

- **Very complex, rules-of-thumb at target:**

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL+BOVL	NOVL+BOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	NOVL	NOVL
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL+BOVL	NOVL	NOVL	NOVL	NOVL
Acc	NOVL+BOVL	NOVL	NOVL	NOVL	OVL+NOVL

  - OVL – Overlapping operations
  - NOVL – Nonoverlapping operations
  - BOVL – Overlapping operations at a byte granularity
  - X – undefined

Credits: RMA Working Group, MPI Forum

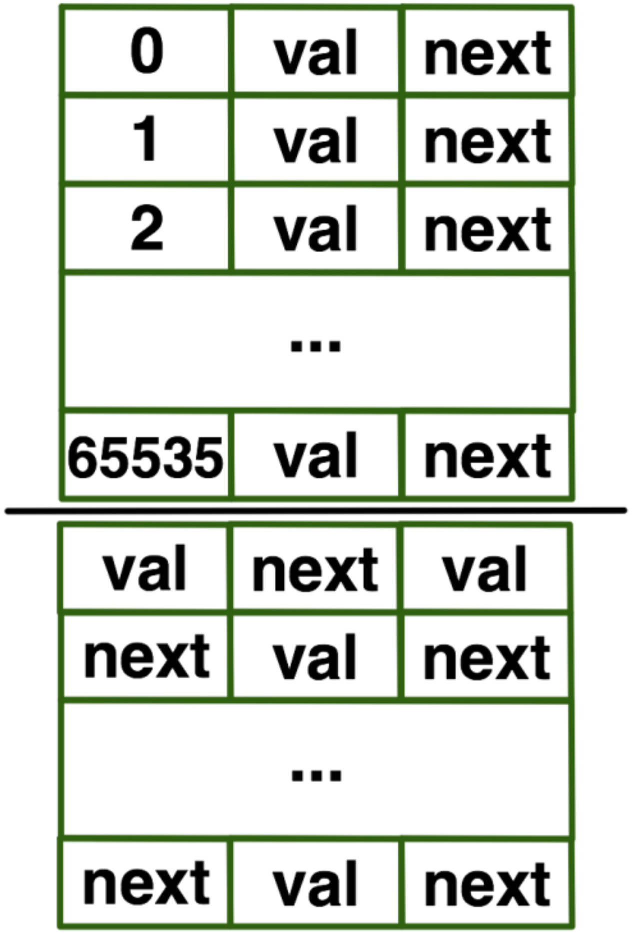
Stencil One-Sided Example

- **stencil_mpi_ddt_rma.cpp**

Distributed Hashtable Example

- **hashtable_mpi.cpp**

- **Use first two bytes as hash**
  - Trivial hash function (2^{16} values)
- **Static 2^{16} table size**
  - One direct value
  - Conflicts as linked list
- **Static heap**
  - Linked list indexes into heap
  - Offset as pointer



0	val	next
1	val	next
2	val	next
...		
65535	val	next

val	next	val
next	val	next
...		
next	val	next

Distributed Hashtable Example

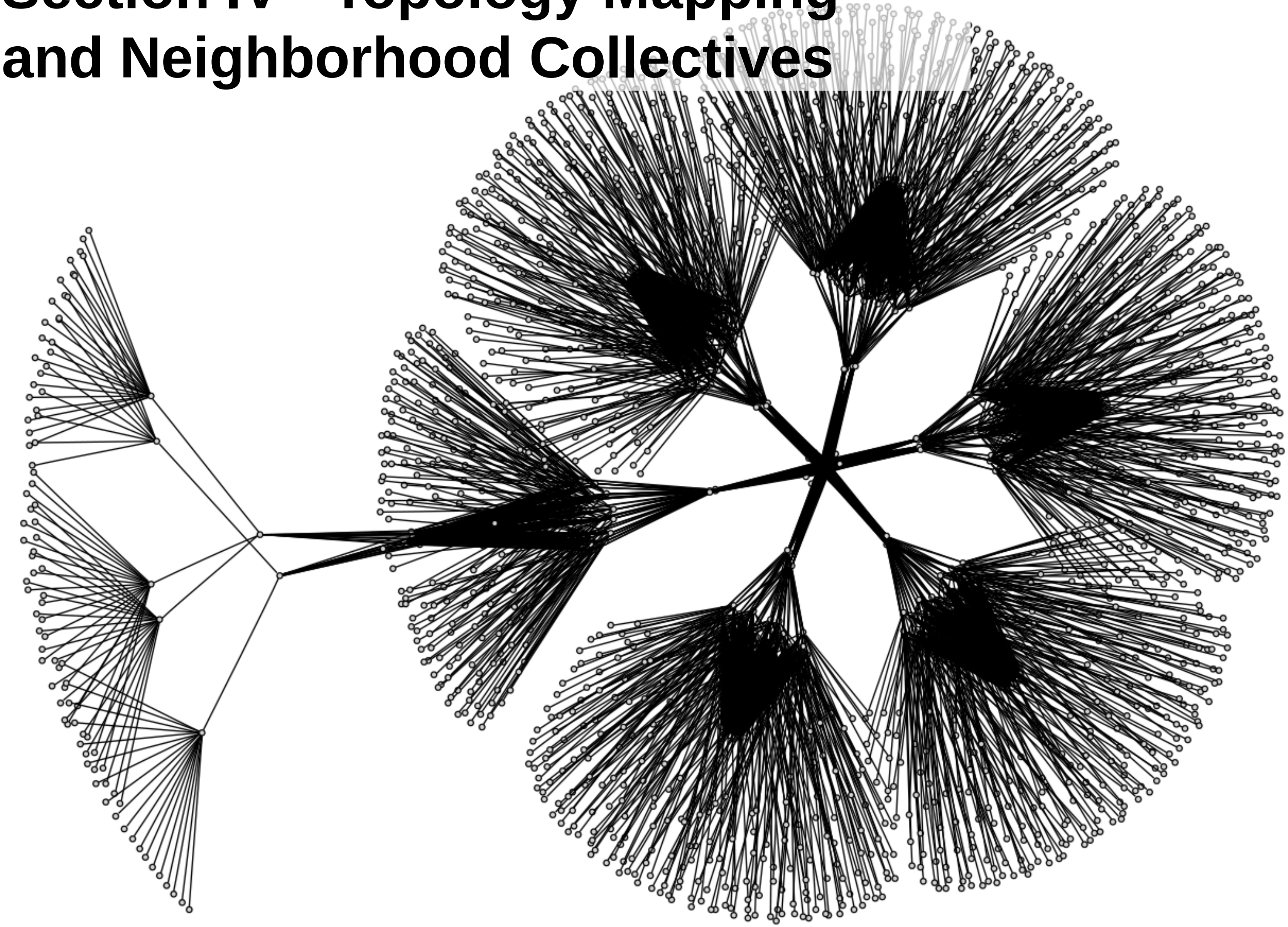
```
int insert(t_hash *hash, int elem) {
  int pos = hashfunc(elem);
  if(hash->table[pos].value == -1) { // direct value in table
    hash->table[pos].value = elem;
  } else { // put on heap
    int newelem=hash->nextfree++; // next free element
    if(hash->table[pos].next == -1) { // first heap element
      // link new elem from table
      hash->table[pos].next = newelem;
    } else { // direct pointer to end of collision list
      int newpos=hash->last[pos];
      hash->table[newpos].next = newelem;
    }
    hash->last[pos]=newelem;
    hash->table[newelem].value = elem; // fill allocated element
  }
}
```

DHT Example – In MPI-3.0

```
int insert(t_hash *hash, int elem) {
  int pos = hashfunc(elem);
  if(hash->table[pos].value == -1) { // direct value in table
    hash->table[pos].value = elem;
  } else { // put on heap
    int newelem=hash->nextfree++; // next free element
    if(hash->table[pos].next == -1) { // first heap element
      // link new elem from table
      hash->table[pos].next = newelem;
   } else { // direct pointer to end of collision list
      int newpos=hash->last[pos];
      hash->table[newpos].next = newelem;
    }
    hash->last[pos]=newelem;
    hash->table[newelem].value = elem; // fill allocated element
  }
}
```

Which function would **you** choose?

Section IV - Topology Mapping and Neighborhood Collectives



Topology Mapping and Neighborhood Collectives

- **Topology mapping basics**
  - Allocation mapping vs. rank reordering
  - Ad-hoc solutions vs. portability
- **MPI topologies**
  - Cartesian
  - Distributed graph
- **Collectives on topologies – neighborhood colls**
  - Use-cases

Topology Mapping Basics

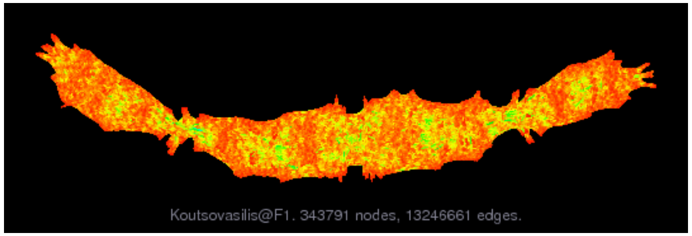
- **First type: Allocation mapping**
  - Up-front specification of communication pattern
  - Batch system picks good set of nodes for given topology
- **Properties:**
  - Not supported by current batch systems
  - Either predefined allocation (BG/P), random allocation, or “global bandwidth maximation”
  - Also problematic to specify communication pattern upfront, not always possible (or static)

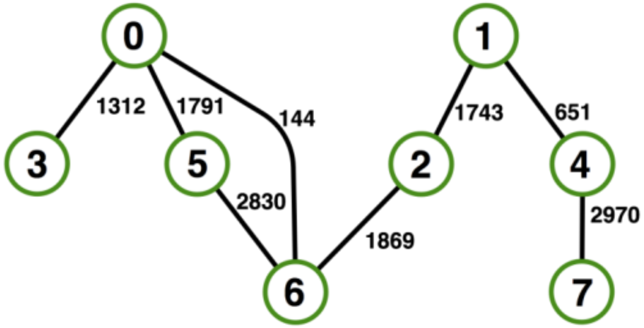
Topology Mapping Basics

- **Rank reordering**
  - Change numbering in a given allocation to reduce congestion or dilation
  - Sometimes automatic (early IBM SP machines)
- **Properties**
  - Always possible, but effect may be limited (e.g., in a bad allocation)
  - Portable way: MPI process topologies

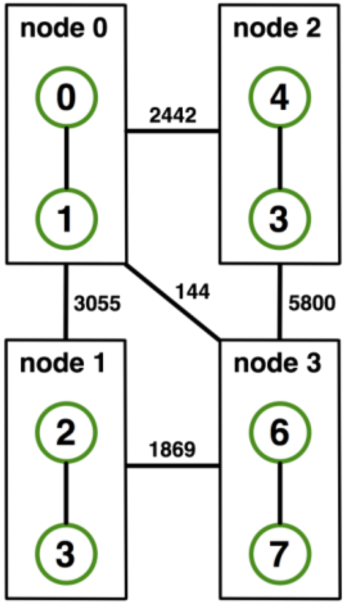
    Network topology is not exposed
  - Manual data shuffling after remapping step

On-Node Reordering


Koutsovasilis@F1. 343791 nodes, 13246661 edges.

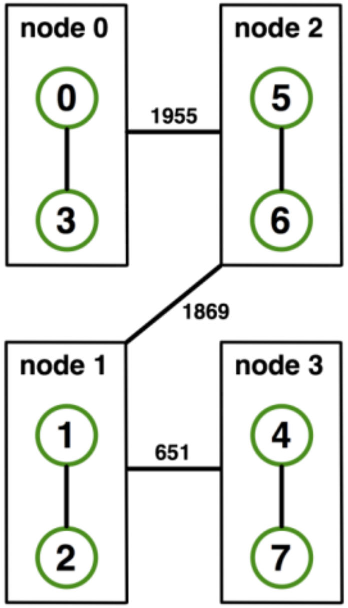


Naïve Mapping



Topomap

Optimized Mapping



Gottschling, TH: Productive Parallel Linear Algebra Programming with Unstructured Topology Adaption, CCGrid'11

MPI Topology Intro

- **Convenience functions (in MPI-1)**
  - Create a graph and query it, nothing else
  - Useful especially for Cartesian topologies

    Query neighbors in n-dimensional space
  - Graph topology: each rank specifies full graph ☹
- **Scalable Graph topology (MPI-2.2)**
  - Graph topology: each rank specifies its neighbors **or** arbitrary subset of the graph
- **Neighborhood collectives (MPI-3.0)**
  - Adding communication functions defined on graph topologies (neighborhood of distance one)

MPI_Cart_create

MPI_Cart_create(MPI_Comm comm_old, int ndims, const int *dims, const int *periods, int reorder, MPI_Comm *comm_cart)

- **Specify ndims-dimensional topology**
  - Optionally periodic in each dimension (Torus)
- **Some processes may return MPI_COMM_NULL**
  - Product sum of dims must be <= P
- **Reorder argument allows for topology mapping**
  - Each calling process may have a new rank in the created communicator
  - Data has to be remapped manually

MPI_Cart_create Example

```
int dims[3] = {5,5,5};
int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- **Creates logical 3-d Torus of size 5x5x5**
- **But we’re starting MPI processes with a one-dimensional argument (-p X)**
  - User has to determine size of each dimension
  - Often as “square” as possible, MPI can help!

MPI_Dims_create

```
MPI_Dims_create(int nnodes, int ndims, int *dims)
```

- **Create dims array for Cart_create with nnodes and ndims**
  - Dimensions are as close as possible (well, in theory)
- **Non-zero entries in dims will not be changed**
  - nnodes must be multiple of all non-zeroes

MPI_Dims_create Example

```
int p;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Dims_create(p, 3, dims);

int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

Cartesian Query Functions

- **Library support and convenience!**
- **MPI_Cartdim_get()**
  - Gets dimensions of a Cartesian communicator
- **MPI_Cart_get()**
  - Gets size of dimensions
- **MPI_Cart_rank()**
  - Translate coordinates to rank
- **MPI_Cart_coords()**
  - Translate rank to coordinates

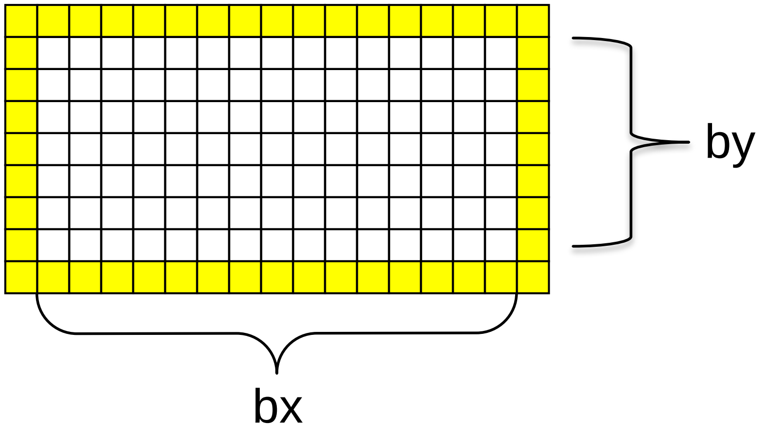
Cartesian Communication Helpers

```
MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
int *rank_source, int *rank_dest)
```

- **Shift in one dimension**
  - Dimensions are numbered from 0 to ndims-1
  - Displacement indicates neighbor distance (-1, 1, …)
  - May return MPI_PROC_NULL
- **Very convenient, all you need for nearest neighbor communication**
  - No “over the edge” though

Code Example

- **stencil_mpi_ddt_overlap_carttopo.cpp**
- **Adds calculation of neighbors with topology**

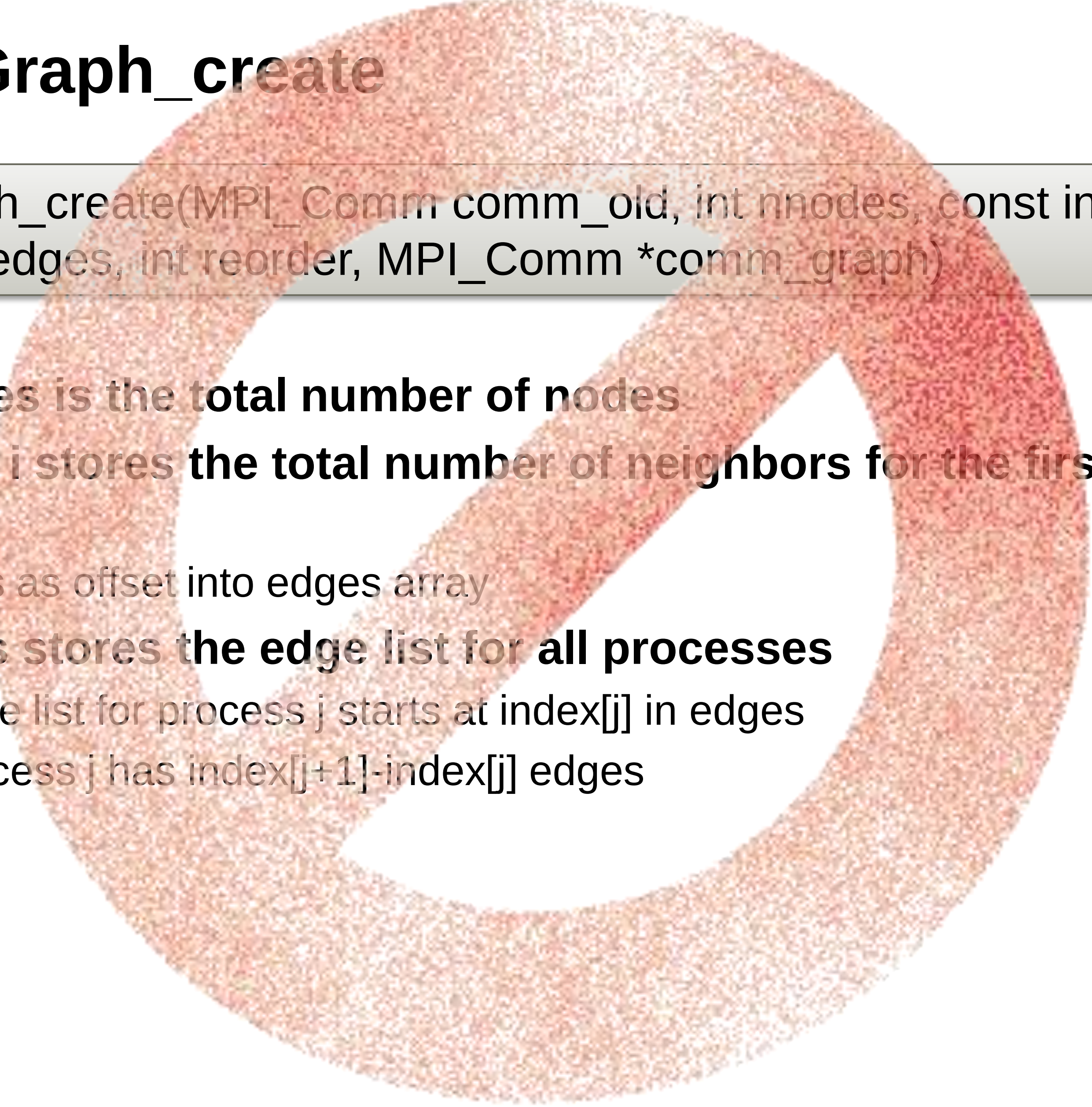


MPI_Graph_create

```
MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int *index,
const int *edges, int reorder, MPI_Comm *comm_graph)
```

- **nnodes is the total number of nodes**
- **index i stores the total number of neighbors for the first i nodes (sum)**
  - Acts as offset into edges array
- **edges stores the edge list for all processes**
  - Edge list for process j starts at index[j] in edges
  - Process j has index[j+1]-index[j] edges

MPI_Graph_create



MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int *index, const int *edges, int reorder, MPI_Comm *comm_graph)

- **nnodes is the total number of nodes**
- **index i stores the total number of neighbors for the first i nodes (sum)**
  - Acts as offset into edges array
- **edges stores the edge list for all processes**
  - Edge list for process j starts at index[j] in edges
  - Process j has index[j+1]-index[j] edges

Distributed graph constructor

- **MPI_Graph_create is discouraged**
  - Not scalable
  - Not deprecated yet but hopefully soon
- **New distributed interface:**
  - Scalable, allows distributed graph specification

    Either local neighbors ***or*** *any edge in the graph*
  - Specify edge weights

    Meaning undefined but optimization opportunity for vendors!
  - Info arguments

    Communicate assertions of semantics to the MPI library

    E.g., semantics of edge weights

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2, CCPE'10

MPI_Dist_graph_create_adjacent

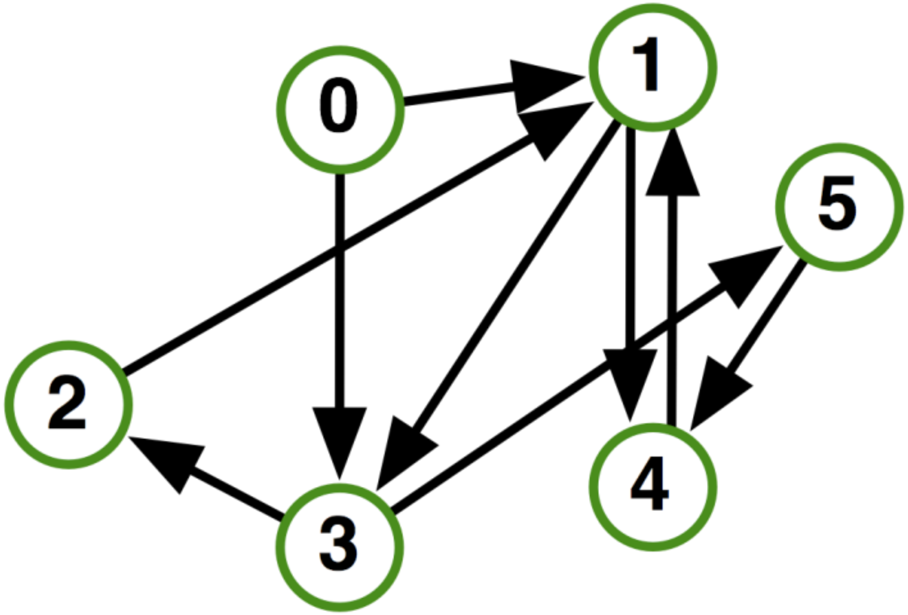
```
MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
const int sources[], const int sourceweights[], int outdegree, const int
destinations[], const int destweights[], MPI_Info info,int reorder, MPI_Comm
*comm_dist_graph)
```

- **indegree, sources, ~weights – source proc. Spec.**
- **outdegree, destinations, ~weights – dest. proc. spec.**
- **info, reorder, comm_dist_graph – as usual**
- **directed graph**
- **Each edge is specified twice, once as out-edge (at the source) and once as in-edge (at the dest)**

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2, CCPE'10

MPI_Dist_graph_create_adjacent

- **Process 0:**
  - Indegree: 0
  - Outdegree: 2
  - Dests: {3,1}
- **Process 1:**
  - Indegree: 3
  - Outdegree: 2
  - Sources: {4,0,2}
  - Dests: {3,4}
- …



Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2, CCPE'10

MPI_Dist_graph_create

```
MPI_Dist_graph_create(MPI_Comm comm_old, int n,
        const int sources[], const int degrees[],
        const int destinations[], const int weights[], MPI_Info info,
        int reorder, MPI_Comm *comm_dist_graph)
```

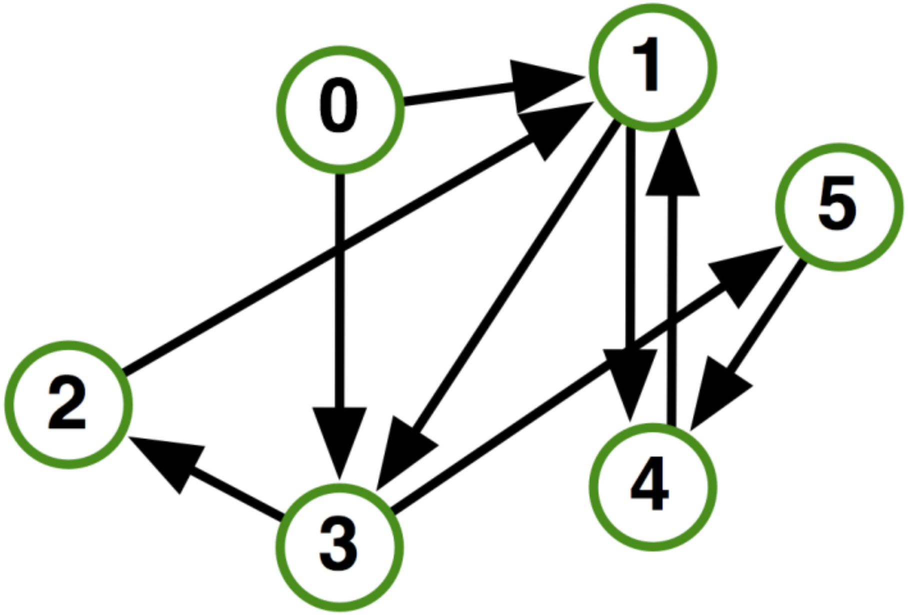
n – number of source nodes

- **sources – n source nodes**
- **degrees – number of edges for each source**
- **destinations, weights – dest. processor specification**
- **info, reorder – as usual**
- **More flexible and convenient**
  - Requires global communication
  - Slightly more expensive than adjacent specification

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2, CCPE'10

MPI_Dist_graph_create

- **Process 0:**
  - N: 2
  - Sources: {0,1}
  - Degrees: {2,1} *
  - Dests: {3,1,4}
- **Process 1:**
  - N: 2
  - Sources: {2,3}
  - Degrees: {1,1}
  - Dests: {1,2}
- …



* Note that in this example, process 0 specifies only one of the two outgoing edges of process 1; the second outgoing edge needs to be specified by another process

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2, CCPE'10

Distributed Graph Neighbor Queries

```
MPI_Dist_graph_neighbors_count(MPI_Comm comm,
          int *indegree,int *outdegree, int *weighted)
```

- **Query the number of neighbors of calling process**
- **Returns indegree and outdegree!**
- **Also info if weighted**

```
MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree,
          int sources[], int sourceweights[], int maxoutdegree,
          int destinations[],int destweights[])
```

- **Query the neighbor list of calling process**
- **Optionally return weights**

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2, CCPE'10

Further Graph Queries

`MPI_Topo_test(MPI_Comm comm, int *status)`

- **Status is either:**
  - MPI_GRAPH (ugs)
  - MPI_CART
  - MPI_DIST_GRAPH
  - MPI_UNDEFINED (no topology)
- **Enables to write libraries on top of MPI topologies!**

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2, CCPE'10

Neighborhood Collectives

- **Topologies implement no communication!**
  - Just helper functions
- **Collective communications only cover some patterns**
  - E.g., no stencil pattern
- **Several requests for “build your own collective” functionality in MPI**
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication patterns!

Cartesian Neighborhood Collectives

- **Communicate with direct neighbors in Cartesian topology**
  - Corresponds to cart_shift with disp=1
  - Collective (all processes in comm must call it, including processes without neighbors)
  - Buffers are laid out as neighbor sequence:

    Defined by order of dimensions, first negative, then positive

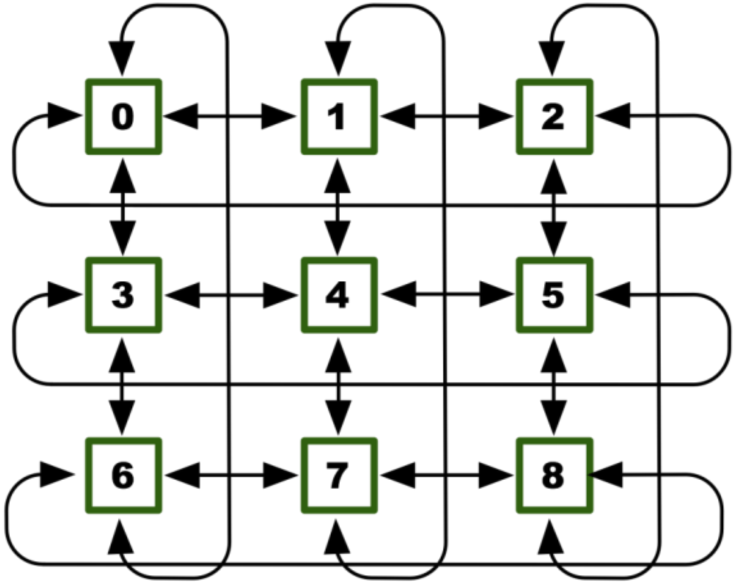
    *2*ndims sources and destinations*

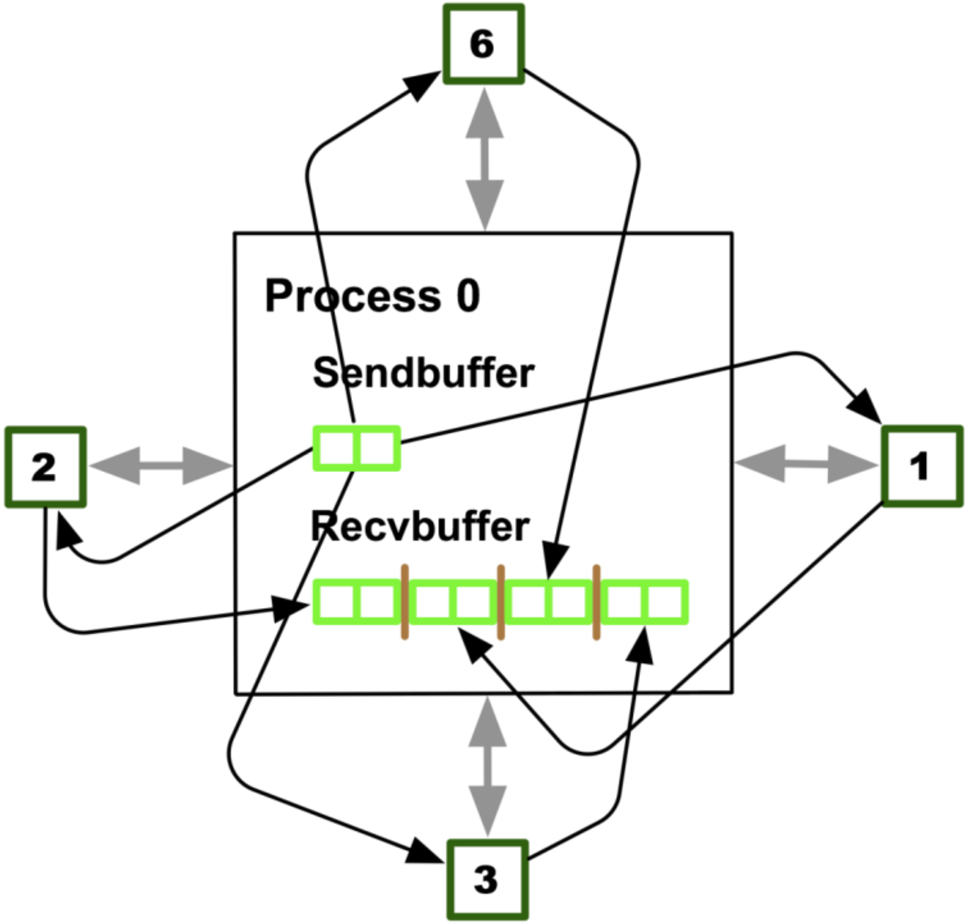
    Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!

TH and J. L. Traeff: Sparse Collective Operations for MPI, HIPS'09

Cartesian Neighborhood Collectives

- **Buffer ordering example:**





TH and J. L. Traeff: Sparse Collective Operations for MPI, HIPS'09

Graph Neighborhood Collectives

- **Collective Communication along arbitrary neighborhoods**
  - Order is determined by order of neighbors as returned by (dist_)graph_neighbors.
  - Distributed graph is directed, may have different numbers of send/recv neighbors
  - Can express dense collective operations ☺
  - Any persistent communication pattern!

TH and J. L. Traeff: Sparse Collective Operations for MPI, HIPS'09

MPI_Neighbor_allgather

```
MPI_Neighbor_allgather(const void* sendbuf, int sendcount,
          MPI_Datatype sendtype, void* recvbuf, int recvcount,
          MPI_Datatype recvtype, MPI_Comm comm)
```

- **Sends the same message to all neighbors**
- **Receives indegree distinct messages**
- **Similar to MPI_Gather**
  - The all prefix expresses that each process is a “root” of his neighborhood
- **Vector version for full flexibility**

MPI_Neighbor_alltoall

```
MPI_Neighbor_alltoall(const void* sendbuf, int sendcount,
          MPI_Datatype sendtype, void* recvbuf, int recvcount,
          MPI_Datatype recvtype, MPI_Comm comm)
```

- **Sends outdegree distinct messages**
- **Received indegree distinct messages**
- **Similar to MPI_Alltoall**
    - Neighborhood specifies full communication relationship
- **Vector and w versions for full flexibility**

Nonblocking Neighborhood Collectives

```
MPI_Ineighbor_allgather(…, MPI_Request *req); MPI_Ineighbor_alltoall(…,
MPI_Request *req);
```

- **Very similar to nonblocking collectives**
- **Collective invocation**
- **Matching in-order (no tags)**
    - No wild tricks with neighborhoods! In order matching per communicator!

Walkthrough of 2D Stencil Code with Neighborhood Collectives

- **stencil_mpi_carttopo_neighcolls.cpp**

Why is Neighborhood Reduce Missing?

```
MPI_Ineighbor_allreducev(…);
```

- **Was originally proposed (see original paper)**
- **High optimization opportunities**
  - Interesting tradeoffs!
  - Research topic
- **Not standardized due to missing use-cases**
  - My team is working on an implementation
  - Offering the obvious interface

TH and J. L. Traeff: Sparse Collective Operations for MPI, HIPS'09

Topology Summary

- **Topology functions allow to specify application communication patterns/topology**
  - Convenience functions (e.g., Cartesian)
  - Storing neighborhood relations (Graph)
- **Enables topology mapping (reorder=1)**
  - Not widely implemented yet
  - May requires manual data re-distribution (according to new rank order)
- **MPI does not expose information about the network topology (would be very complex)**

Neighborhood Collectives Summary

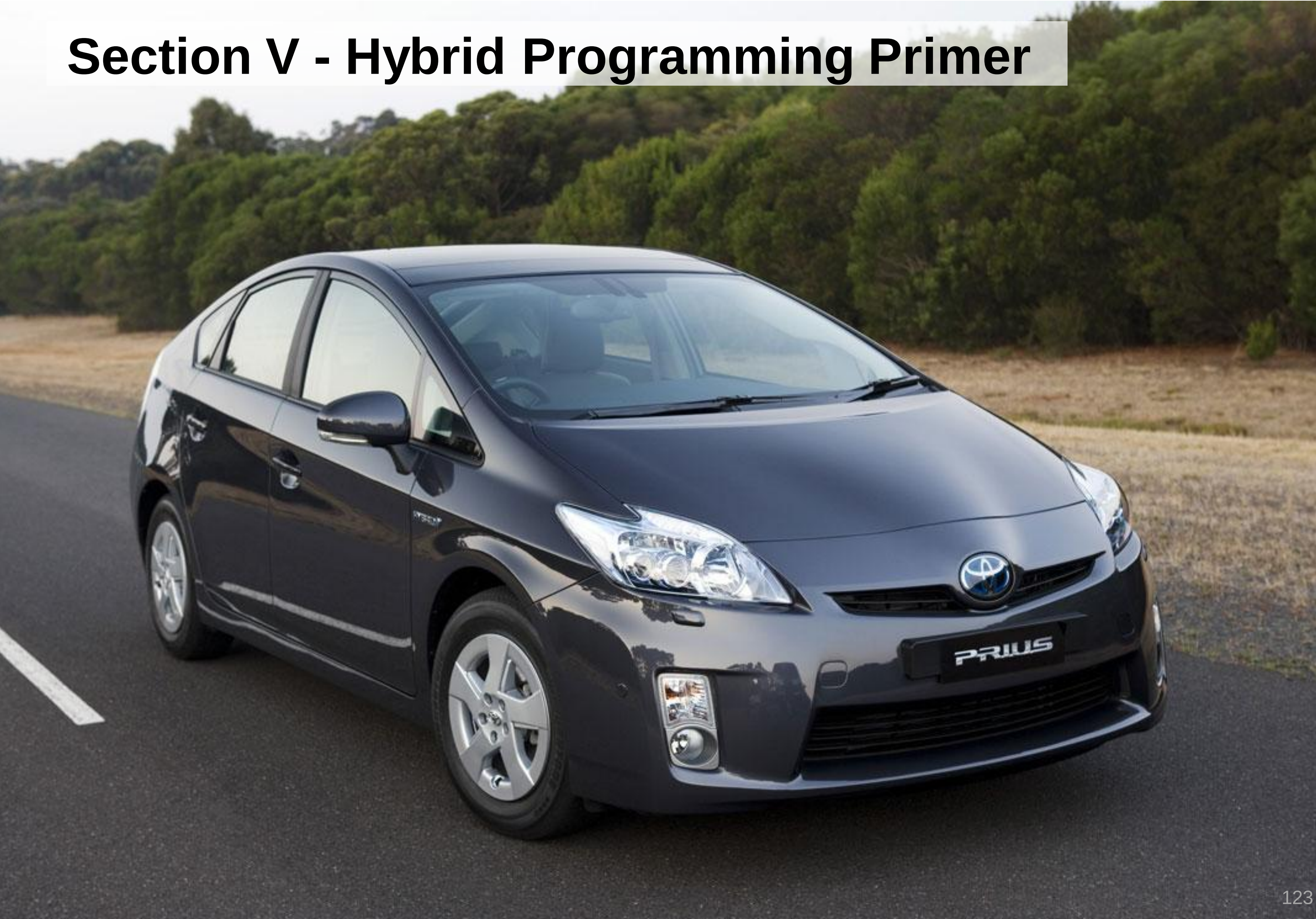
- **Neighborhood collectives add communication functions to process topologies**
  - Collective optimization potential!
- **Allgather**
  - One item to all neighbors
- **Alltoall**
  - Personalized item to each neighbor
- **High optimization potential (similar to collective operations)**
  - Interface encourages use of topology mapping!

Section Summary

- **Process topologies enable:**
    - High-abstraction to specify communication pattern
    - Has to be relatively static (temporal locality)

      Creation is expensive (collective)
    - Offers basic communication functions
- **Library can optimize:**
    - Communication schedule for neighborhood colls
    - Topology mapping

Section V - Hybrid Programming Primer



Hybrid Programming Primer

- **No complete view, discussions not finished**
  - Considered very important!
- **Modes: shared everything (threaded MPI) vs. shared something (SHM windows)**
  - And everything in between!
- **How to deal with multicore and accelerators?**
  - OpenMP, Cuda, UPC/CAF, OpenACC?
  - Very specific to actual environment, no general statements possible (no standardization)
  - MPI is generally compatibly, minor pitfalls

Threads in MPI-2.2

- **Four thread levels in MPI-2.2**
  - Single – only one thread exists
  - Funneled – only master thread calls MPI
  - Serialized – no concurrent calls to MPI
  - Multiple – concurrent calls to MPI
- **But how do I call this function – oh well ☺**
- **To add more confusion: MPI processes may be OS threads!**

Matched Probe

- **MPI_Probe to receive messages of unknown size**
  - MPI_Probe(…, status)
  - size = get_count(status)*size_of(datatype)
  - buffer = malloc(size)
  - MPI_Recv(buffer, …)
- **MPI_Probe peeks in matching queue**
  - Does not change it → stateful object

Matched Probe

- **Two threads, A and B perform probe, malloc, receive sequence**
    - $A_P \rightarrow A_M \rightarrow A_R \rightarrow B_P \rightarrow B_M \rightarrow B_R$
- **Possible ordering**
    - $A_P \rightarrow B_P \rightarrow B_M \rightarrow B_R \rightarrow A_M \rightarrow A_R$
    - Wrong matching!
    - Thread A’s message was “stolen” by B
    - Access to queue needs mutual exclusion ☹

MPI_Mprobe to the Rescue

- **Avoid state in the library**
  - Return handle, remove message from queue

```
MPI_Message msg; MPI_Status status;
/* Match a message */
MPI_Mprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
                       &msg, &status);
/* Allocate memory to receive the message */
int count; MPI_get_count(&status, MPI_BYTE, &count);
char* buffer = malloc(count);
/* Receive this message. */
MPI_Mrecv(buffer, count, MPI_BYTE, &msg, MPI_STATUS_IGNORE);
```

Shared Memory Use-Cases

- **Reduce memory footprint**
  - E.g., share static lookup tables
  - Avoid re-computing (e.g., NWCHEM)
- **More structured programming than MPI+X**
  - Share what needs to be shared!
  - Not everything open to races like OpenMP
- **Speedups (very tricky!)**
  - Reduce communication (matching, copy) overheads
  - False sharing is an issue!

Shared Memory Windows

```
MPI_Win_allocate_shared(MPI_Aint size, MPI_Info info, MPI_Comm
comm, void *baseptr, MPI_Win *win)
```

- **Allocates shared memory segment in win**
  - Collective, fully RMA capable
  - All processes in comm must be in shared memory!
- **Returns pointer to start of own part**
- **Two allocation modes:**
  - Contiguous (default): process i's memory starts where process i-1's memory ends
  - Non Contiguous (info key alloc_shared_noncontig) possible ccNUMA optimizations

Shared Memory Comm Creation

MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info, MPI_Comm *newcomm)

- **Returns disjoint comms based on split type**
  - Collective
- **Types (only one so far):**
  - MPI_COMM_TYPE_SHARED – split into largest subcommunicators with shared memory access
- **Key mandates process ordering**
  - Cf. comm_split

SHM Windows Address Query

MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size, void *baseptr)

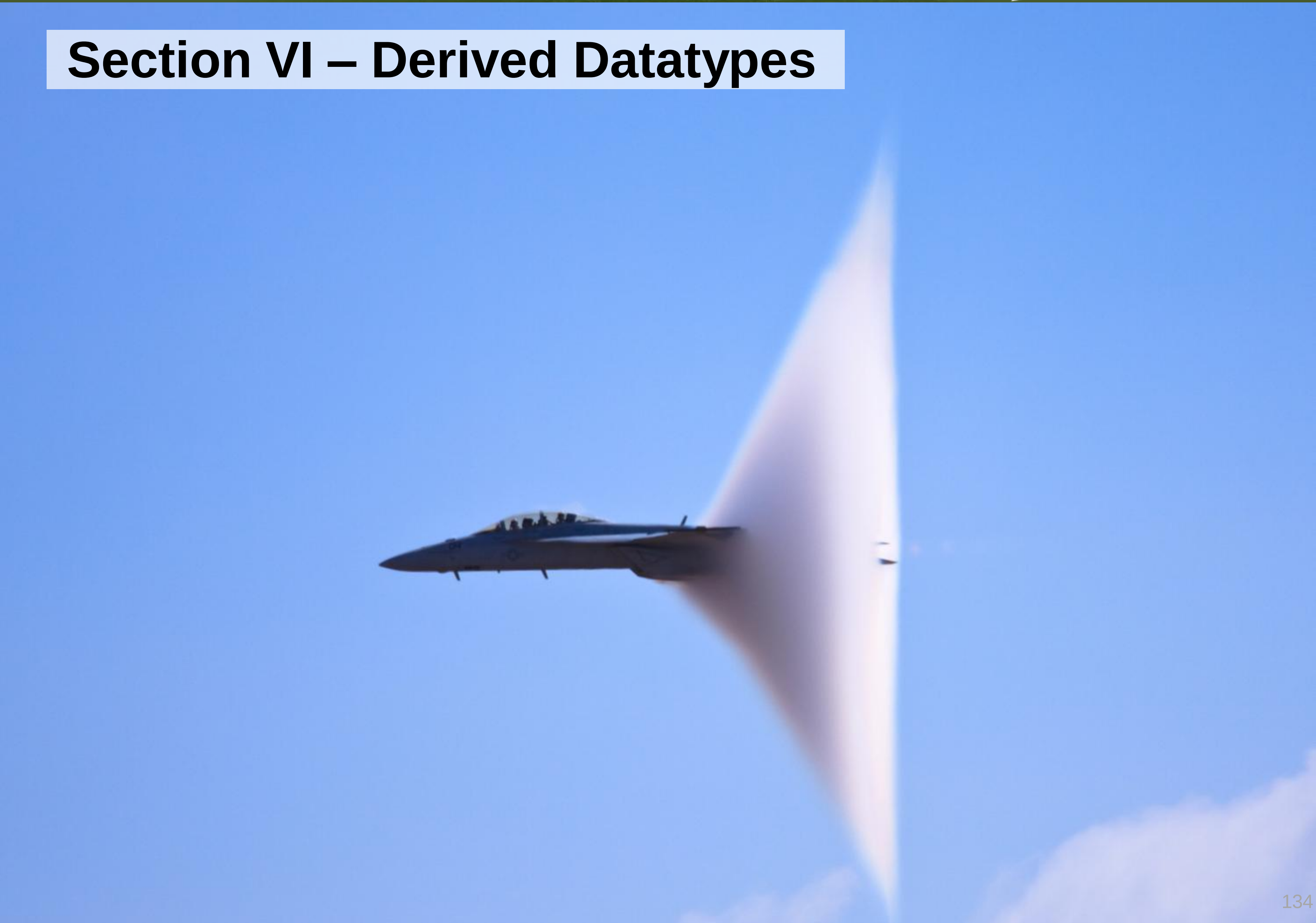
- **User can compute remote addresses in contig case but needs all sizes**
  - Not possible in noncontig case!
  - Processes **cannot** communicate base address, may be different at different processes!
- **Base address query function!**
  - MPI_PROC_NULL as rank returns lowest offset

New Communicator Creation Functions

- **Noncollective communicator creation**
  - Allows to create communicators without involving all processes in the parent communicator
  - Very useful for some applications (dynamic sub-grouping) or fault tolerance (dead processes)
- **Nonblocking communicator duplication**
  - MPI_Comm_idup(…, req) – like it sounds
  - Similar semantics to nonblocking collectives
  - Enables the implementation of nonblocking libraries

J. Dinan et al.: Noncollective Communicator Creation in MPI, EuroMPI'11
T. Hoefler: Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions, Keynote, IMUDI'11

Section VI – Derived Datatypes



Derived Datatypes

Abelson & Sussman: “*Programs must be written for people to read, and only incidentally for machines to execute.*”

- **Derived Datatypes exist since MPI-1.0**
  - Some extensions in MPI-2.x and MPI-3.0
- **Why do I talk about this really old feature?**
  - It is a very advanced and elegant declarative concept
  - It enables many elegant optimizations (zero copy)
  - It falsely has a bad reputation (which it earned in early days)

Quick MPI Datatype Introduction

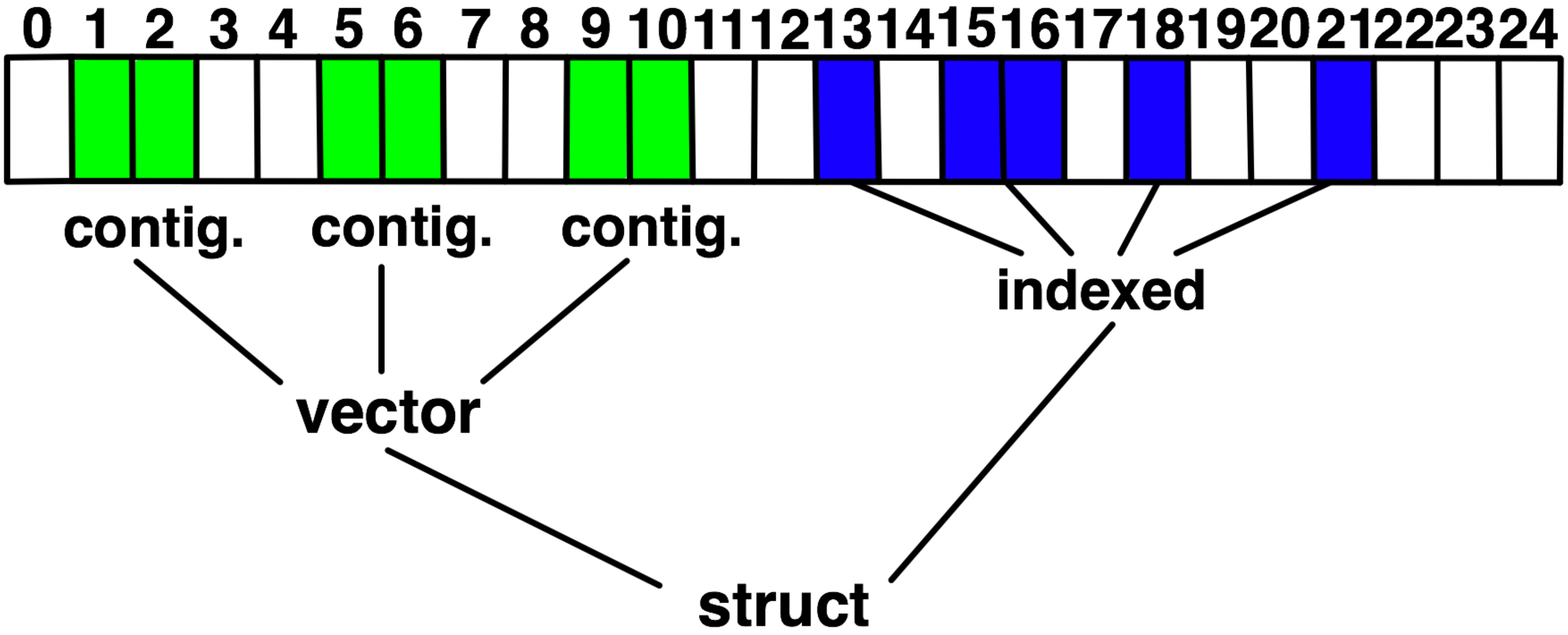
- **Datatypes allow to (de)serialize arbitrary data layouts into a message stream**
  - Networks provide serial channels
  - Same for block devices and I/O
- **Several constructors allow arbitrary layouts**
  - Recursive specification possible
  - *Declarative* specification of data-layout

    "what" and not "how", leaves optimization to implementation (many unexplored possibilities!)
  - Choosing the right constructors is not always simple

Derived Datatype Terminology

- **Type Size**
  - Size of DDT signature (total occupied bytes)
  - Important for matching (signatures must match)
- **Lower Bound**
  - Where does the DDT start
  - Allows to specify “holes” at the beginning
- **Extent**
  - Complete size of the DDT
  - Allows to interleave DDT, relatively “dangerous”

Derived Datatype Example



- **Explain Lower Bound, Size, Extent**

What is Zero Copy?

- **Somewhat weak terminology**
  - MPI forces “remote” copy , assumed baseline
- **But:**
  - MPI implementations copy internally

    E.g., networking stack (TCP), packing DDTs

    Zero-copy is possible (RDMA, I/O Vectors, SHMEM)
  - MPI applications copy too often

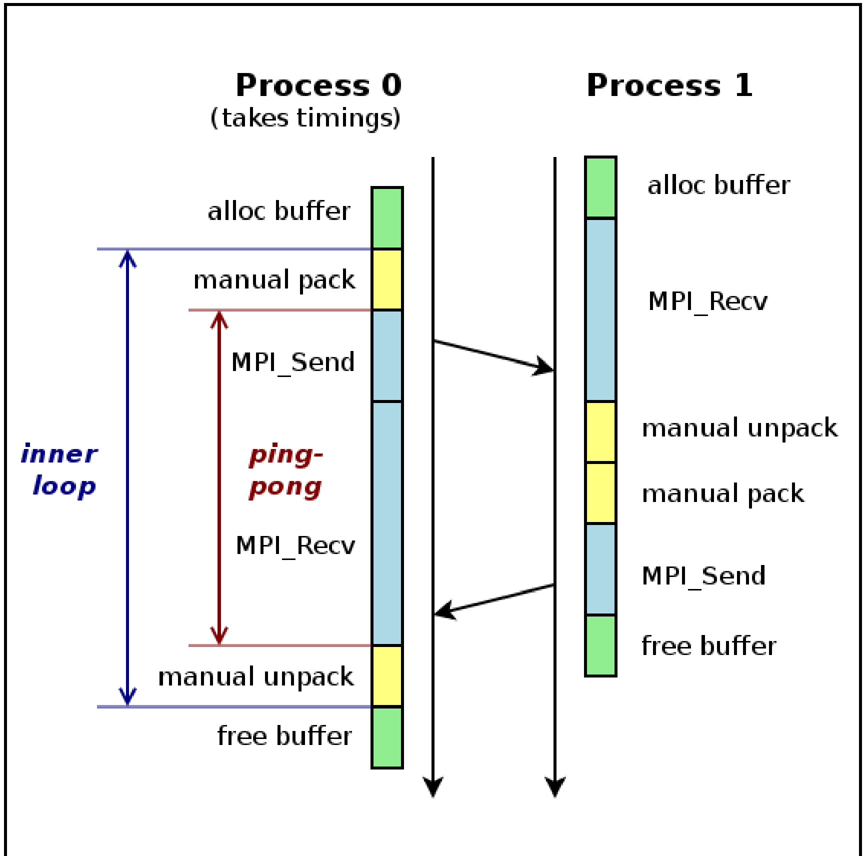
    E.g., manual pack, unpack or data rearrangement

    DDT can do both!

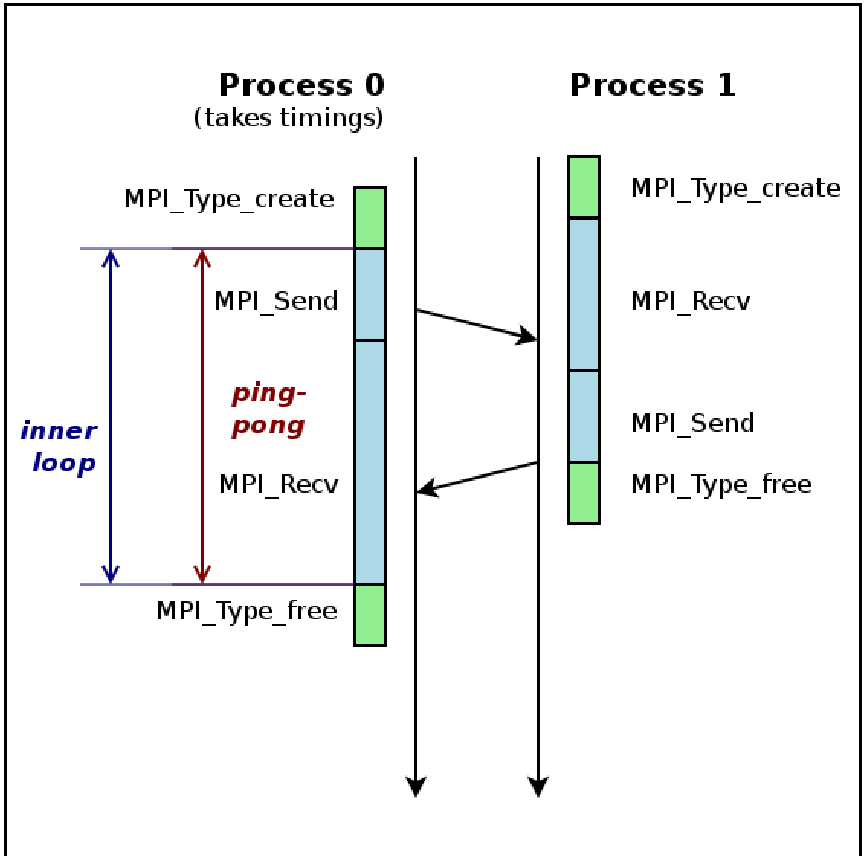
Purpose of this Section

- **Demonstrate utility of DDT in practice**
  - Early implementations were bad → folklore
  - Some are still bad → chicken egg problem
- **Show creative use of DDTs**
  - Encode local transpose for FFT
  - Enable you to create more!
- **Gather input on realistic benchmark cases**
  - Guide optimization of DDT implementations

A new Way of Benchmarking



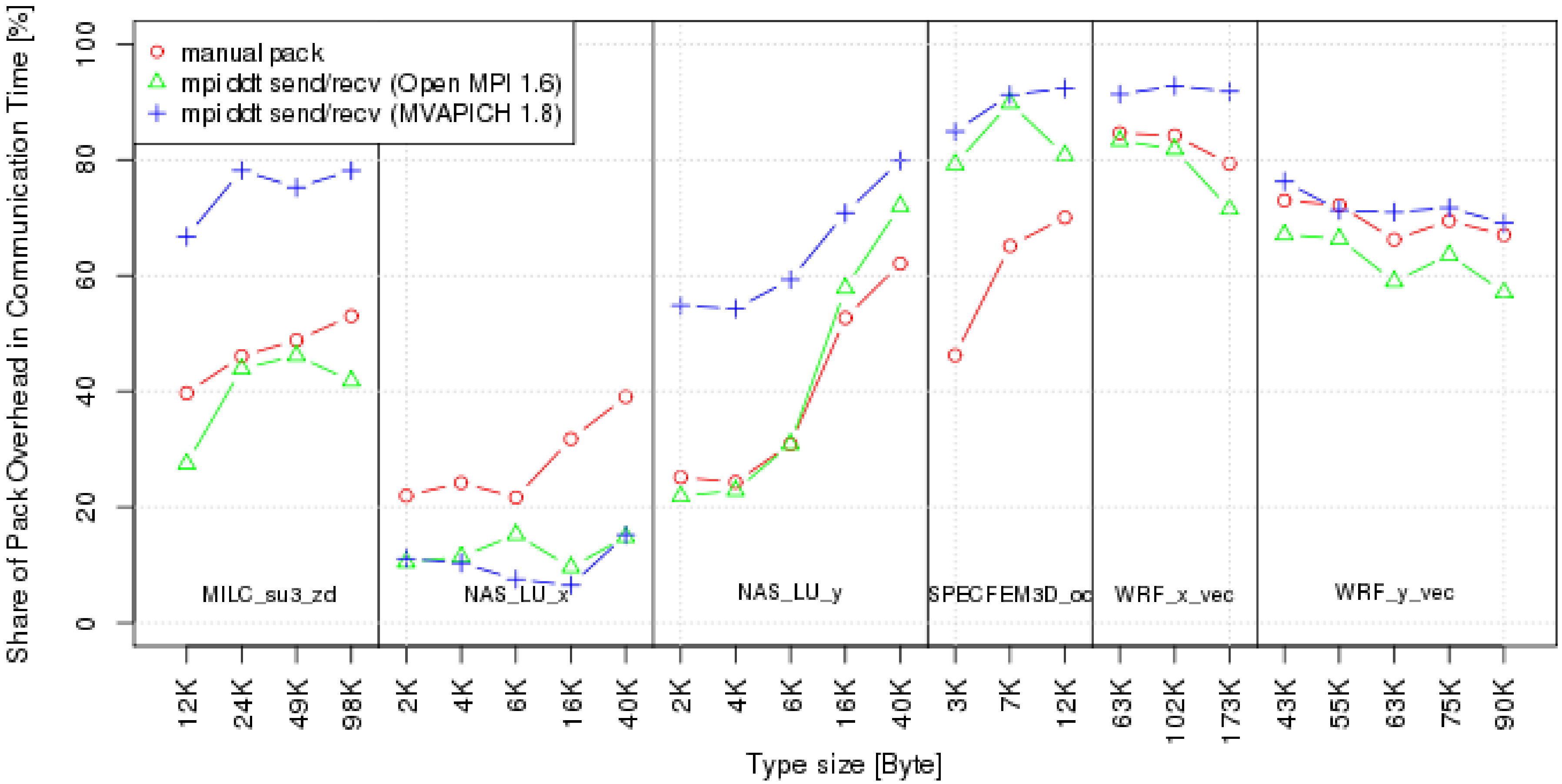
Benchmark for manual packing



Benchmark for MPI DDTs

Schneider, Gerstenberger, TH: Micro-Applications for Communication Data Access Patterns, EuroMPI'13

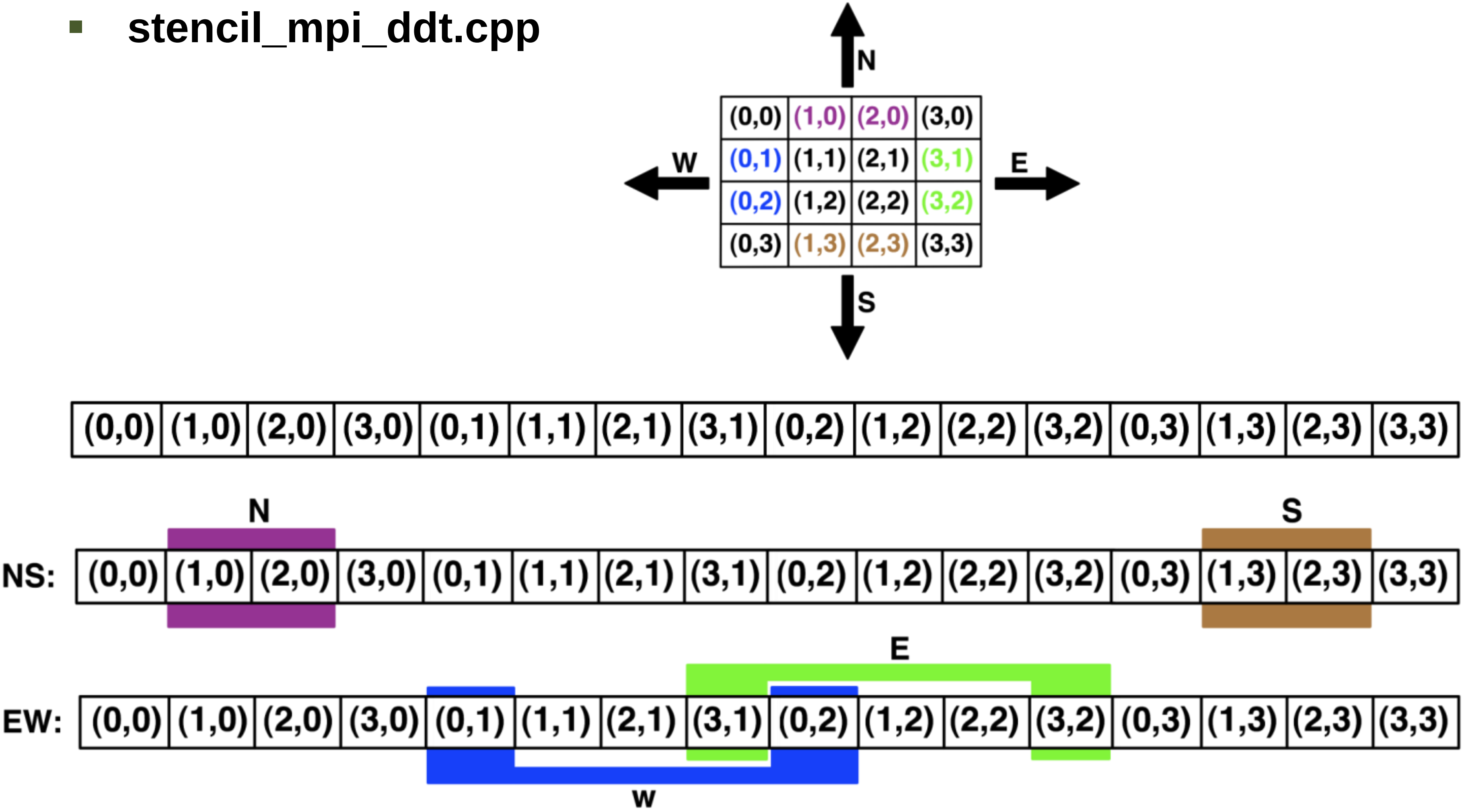
Motivation



Schneider, Gerstenberger, TH: Micro-Applications for Communication Data Access Patterns, EuroMPI'13

Datatypes for the Stencil

- **stencil_mpi_ddt.cpp**



(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)

(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)	(2,3)	(3,3)

MPI’s Intrinsic Datatypes

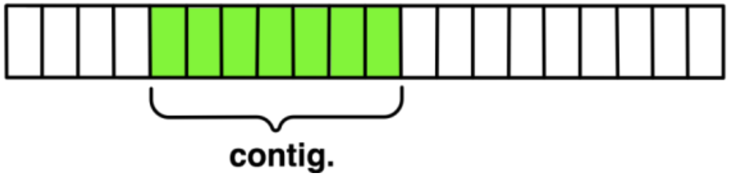
- **Why intrinsic types?**
  - Heterogeneity, nice to send a Boolean from C to Fortran
  - Conversion rules are complex, not discussed here
  - Length matches to language types

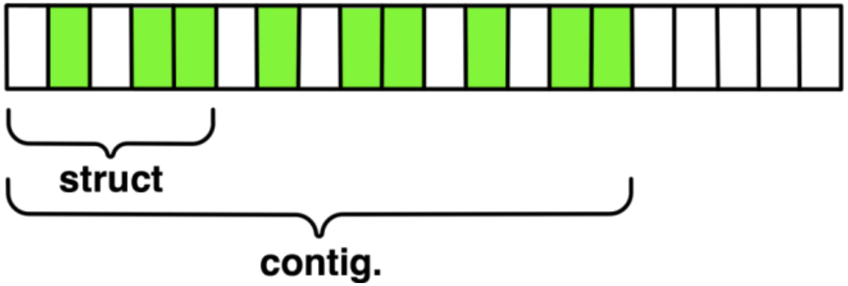
    Avoid sizeof(int) mess
- **Users should generally use intrinsic types as basic types for communication and type construction!**
  - MPI_BYTE should be avoided at all cost
- **MPI-2.2 adds some missing C types**
  - E.g., unsigned long long

MPI_Type_contiguous

```
MPI_Type_contiguous(int count, MPI_Datatype
oldtype, MPI_Datatype *newtype)
```

- **Contiguous array of oldtype**
- **Should not be used as last type (can be replaced by count)**

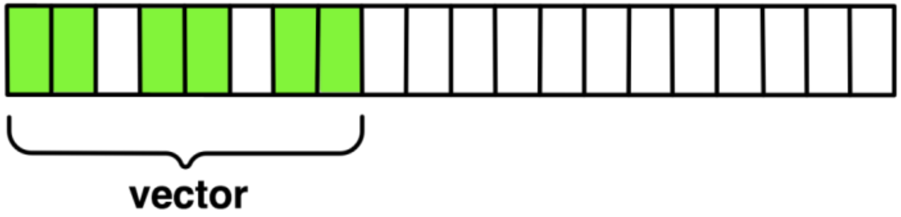


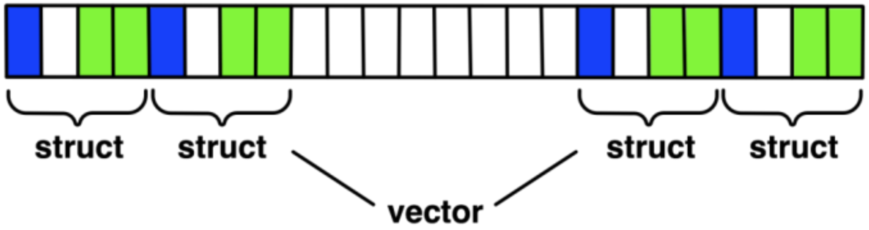


MPI_Type_vector

```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype
oldtype, MPI_Datatype *newtype)
```

- **Specify strided blocks of data of oldtype**
- **Very useful for Cartesian arrays**

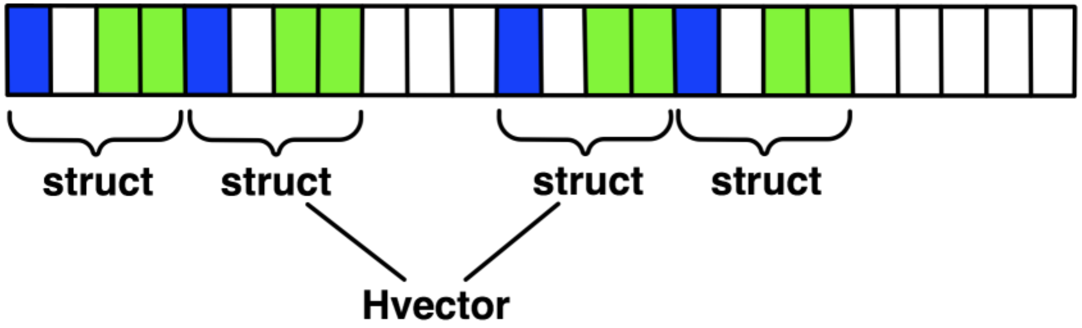




MPI_Type_create_hvector

```
MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- **Create non-unit strided vectors**
- **Useful for composition, e.g., vector of structs**



MPI_Type_indexed

```
MPI_Type_indexed(int count, int *array_of_blocklengths,
int *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)
```

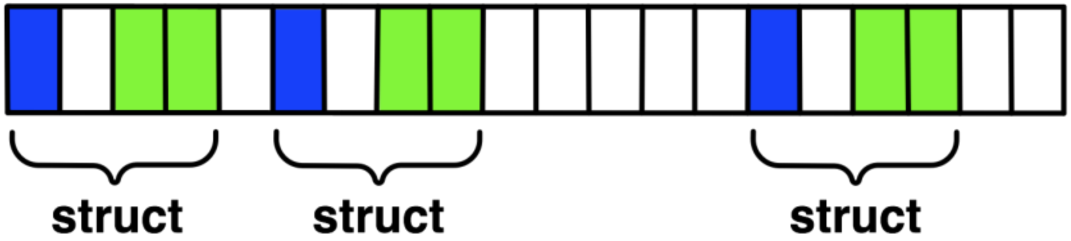
- **Pulling irregular subsets of data from a single array (cf. vector collectives)**
    - dynamic codes with index lists, expensive though!

    - blen={1,1,2,1,2,1}
    - displs={0,3,5,9,13,17}

MPI_Type_create_hindexed

MPI_Type_create_hindexed(int count, int *arr_of_blocklengths, MPI_Aint *arr_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

- **Indexed with non-unit displacements, e.g., pulling types out of different arrays**



MPI_Type_create_indexed_block

MPI_Type_create_indexed_block(int count, int blocklength,
int *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)

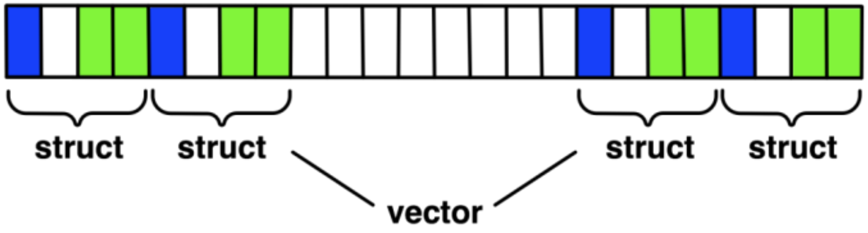
- **Like Create_indexed but blocklength is the same**

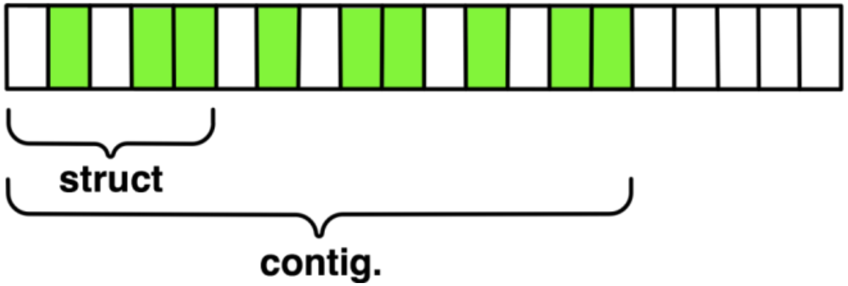
  - blen=2
  - displs={0,5,9,13,18}

MPI_Type_create_struct

```
MPI_Type_create_struct(int count, int array_of_blocklengths[],
MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[],
MPI_Datatype *newtype)
```

- **Most general constructor (cf. Alltoallw), allows different types and arbitrary arrays**

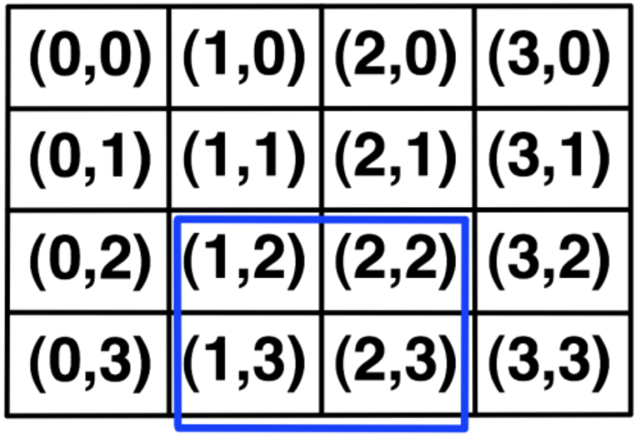




MPI_Type_create_subarray

```
MPI_Type_create_subarray(int ndims, int array_of_sizes[],
int array_of_subsizes[], int array_of_starts[], int order,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- **Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)**



(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)

MPI_Type_create_darray

```
MPI_Type_create_darray(int size, int rank, int ndims,
int array_of_gsizes[], int array_of_distribs[], int
array_of_dargs[], int array_of_psizes[], int order,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

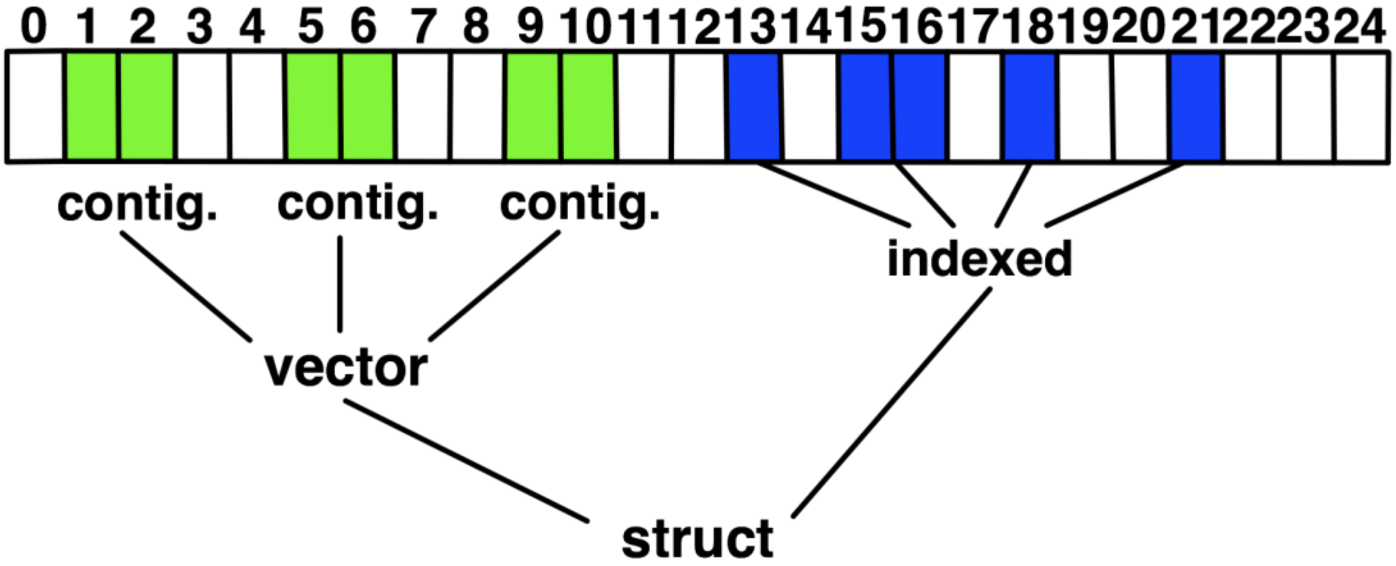
- **Create distributed array, supports block, cyclic and no distribution for each dimension**
  - Very useful for I/O

MPI_BOTTOM and MPI_Get_address

- **MPI_BOTTOM is the absolute zero address**
  - Portability (e.g., may be non-zero in globally shared memory)
- **MPI_Get_address**
  - Returns address relative to MPI_BOTTOM
  - Portability (do not use “&” operator in C!)
- **Very important to**
  - build struct datatypes
  - If data spans multiple arrays

Recap: Size, Extent, and Bounds

- **MPI_Type_size returns size of datatype**
- **MPI_Type_get_extent returns lower bound and extent**



Commit, Free, and Dup

- **Types must be comitted before use**
  - Only the ones that are used!
  - MPI_Type_commit may perform heavy optimizations (and will hopefully)
- **MPI_Type_free**
  - Free MPI resources of datatypes
  - Does not affect types built from it
- **MPI_Type_dup**
  - Duplicated a type
  - Library abstraction (composability)

Other DDT Functions

- **Pack/Unpack**
  - Mainly for compatibility to legacy libraries
  - You should not be doing this yourself
- **Get_envelope/contents**
  - Only for expert library developers
  - Libraries like MPITypes[1] make this easier
- **MPI_Create_resized**
  - Change extent and size (dangerous but useful)

1: http://www.mcs.anl.gov/mpitypes/

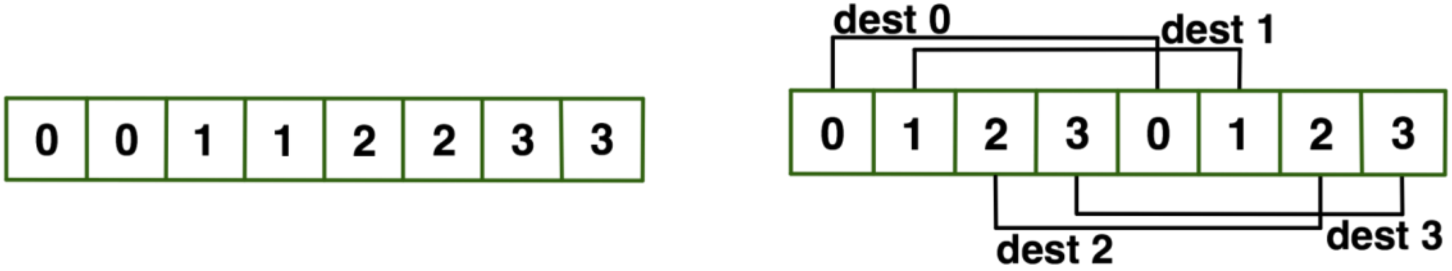
Datatype Selection Tree

- **Simple and effective performance model:**
  - More parameters == slower
- **contig < vector < index_block < index < struct**
- **Some (most) MPIs are inconsistent**
  - But this rule is portable
- **Advice to users:**
  - Try datatype “compression” bottom-up

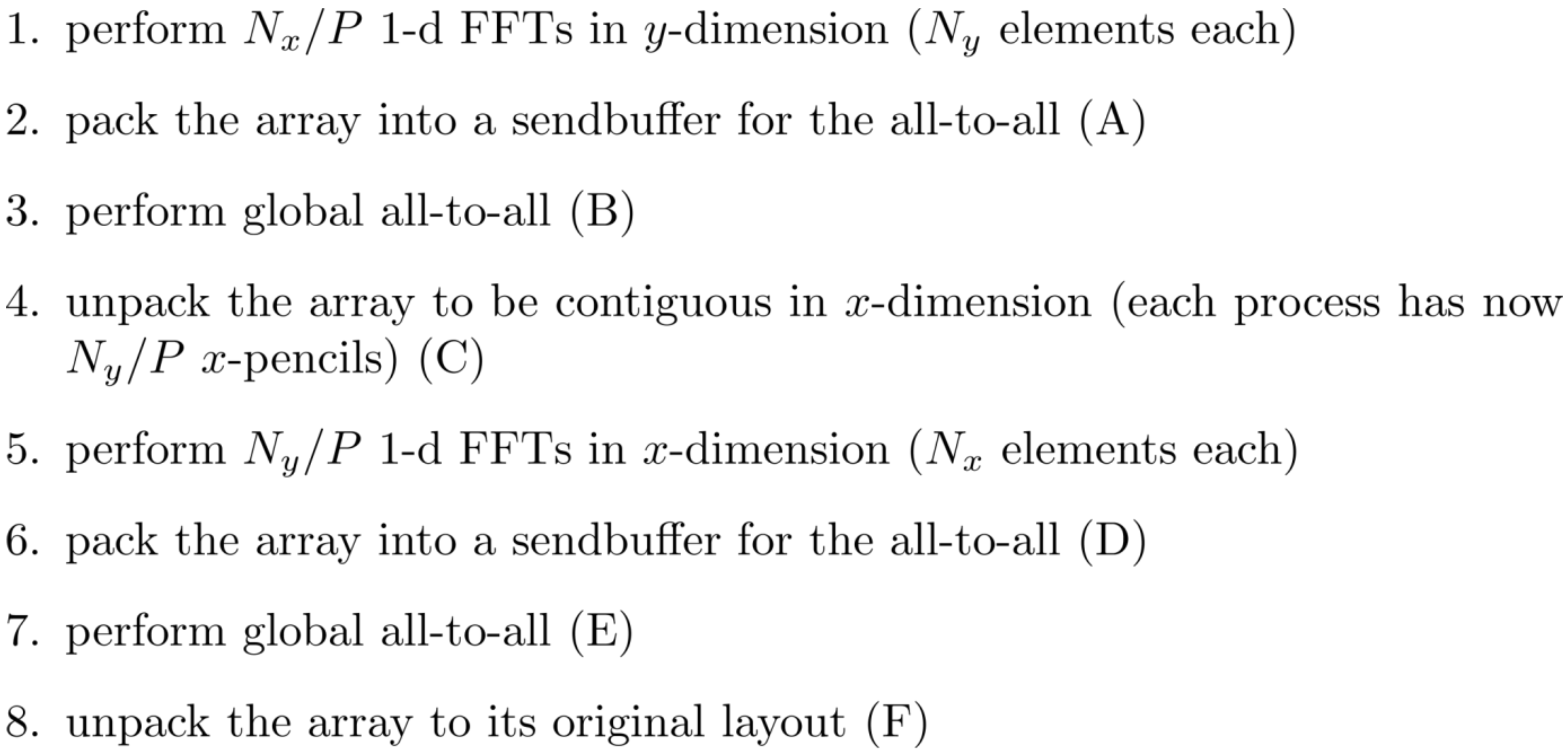
W. Gropp et al.:Performance Expectations and Guidelines for MPI Derived Datatypes, EuroMPI’12

Datatypes and Collectives

- **Alltoall, Scatter, Gather and friends expect data in rank order**
  - 1st rank: offset 0
  - 2nd rank: offset <extent>
  - ith rank: offset: i*<extent>
- **Makes tricks necessary if types are overlapping → use extent (create_resized)**



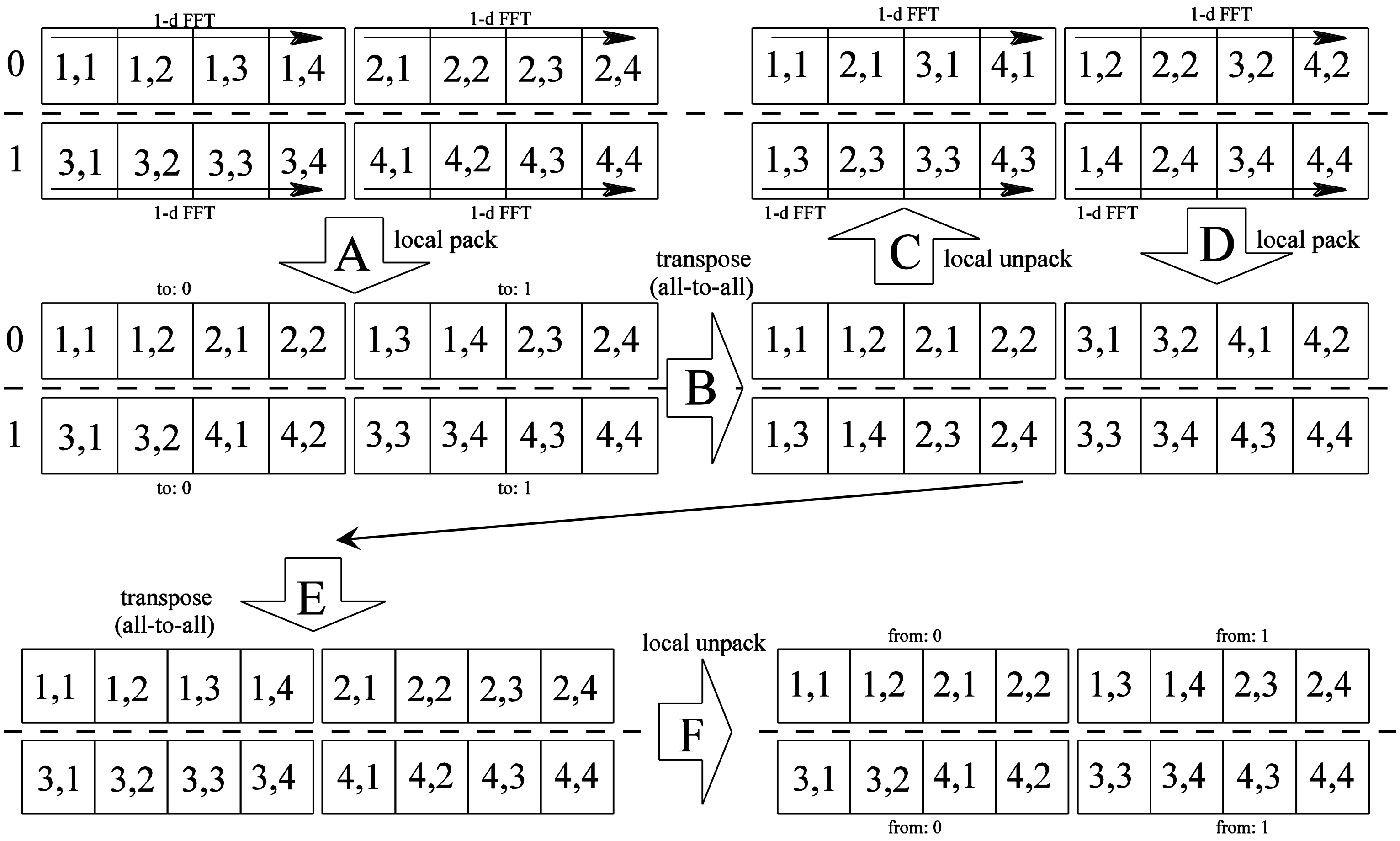
A Complex Example - FFT



1. perform N_x/P 1-d FFTs in y-dimension (N_y elements each)
2. pack the array into a sendbuffer for the all-to-all (A)
3. perform global all-to-all (B)
4. unpack the array to be contiguous in x-dimension (each process has now N_y/P x-pencils) (C)
5. perform N_y/P 1-d FFTs in x-dimension (N_x elements each)
6. pack the array into a sendbuffer for the all-to-all (D)
7. perform global all-to-all (E)
8. unpack the array to its original layout (F)

TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

A Complex Example - FFT



TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

2d-FFT Optimization Possibilities

1. **Use DDT for pack/unpack (obvious)**
   - Eliminate 4 of 8 steps

     Introduce local transpose

2. **Use DDT for local transpose**
   - After unpack
   - Non-intuitive way of using DDTs

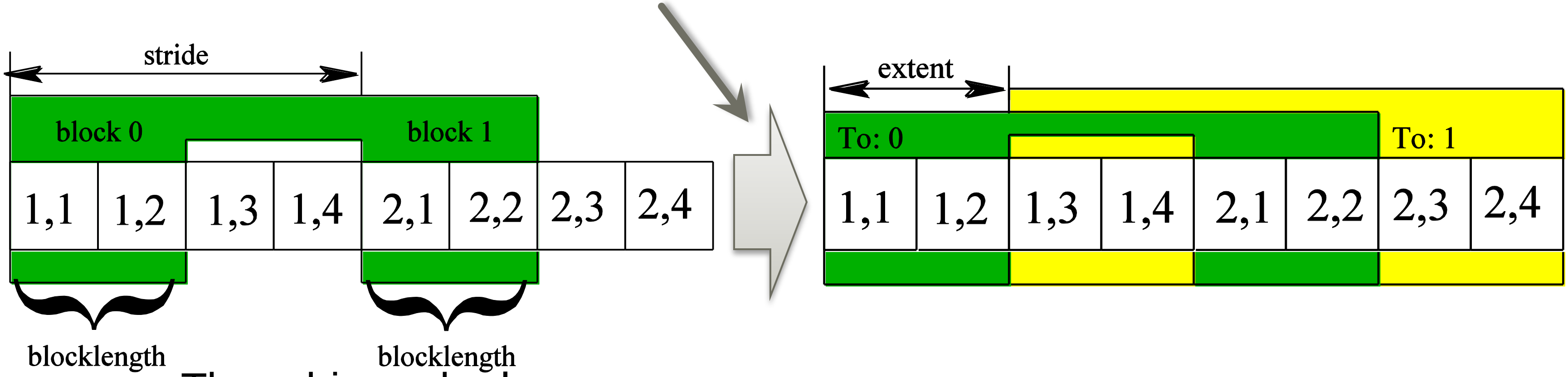
     Eliminate local transpose

TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

The Send Datatype

1. Type_struct for complex numbers
2. Type_contiguous for blocks
3. Type_vector for stride

Need to change extent to allow overlap (create_resized)



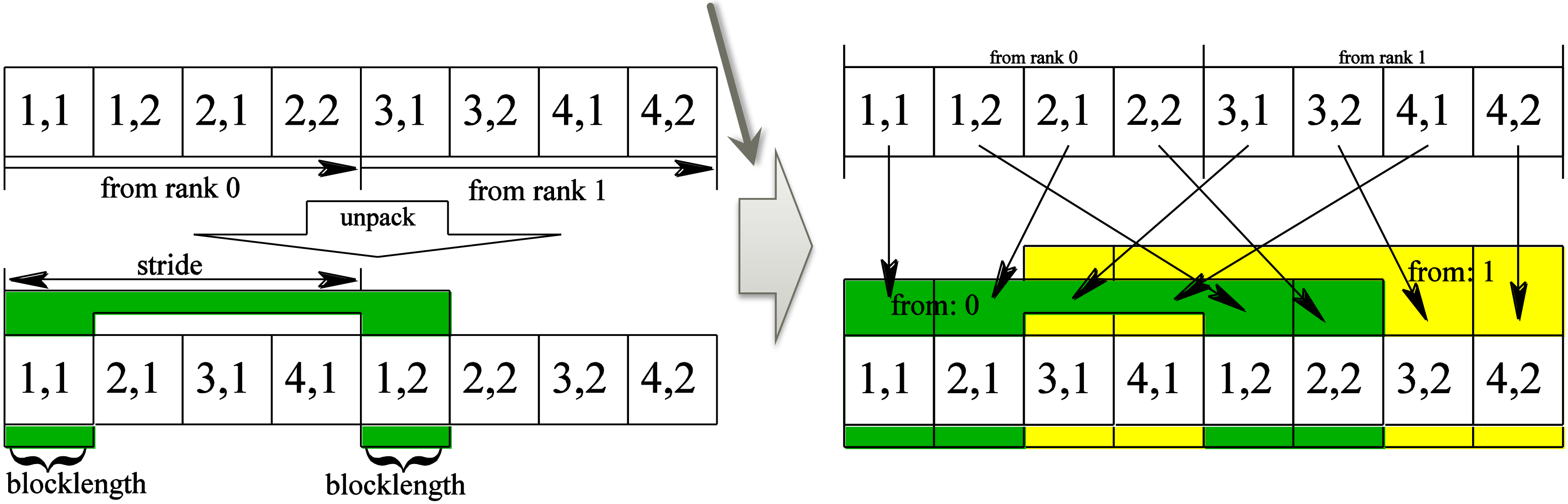
- Three hierarchy-layers

TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

The Receive Datatype

- Type_struct (complex)
- Type_vector (no contiguous, local transpose)

  Needs to change extent (create_resized)



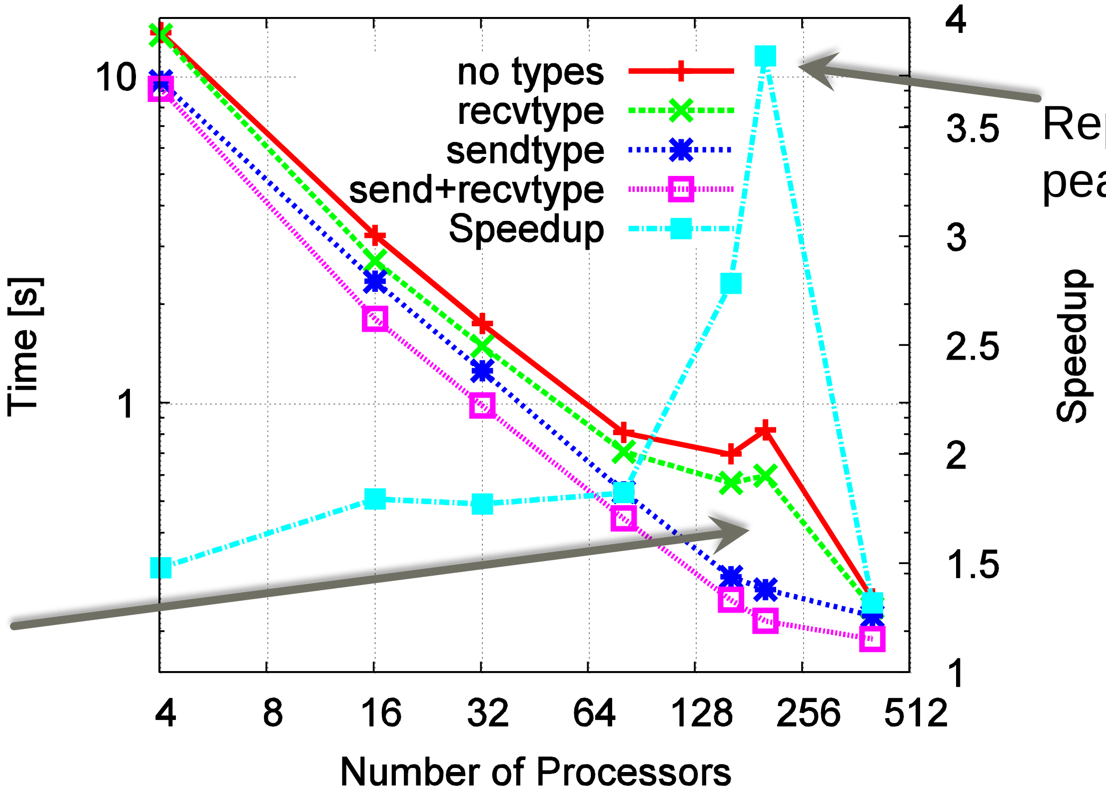
TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

Experimental Evaluation

- **Odin @ IU**
  - 128 compute nodes, 2x2 Opteron 1354 2.1 GHz
  - SDR InfiniBand (OFED 1.3.1).
  - Open MPI 1.4.1 (openib BTL), g++ 4.1.2
- **Jaguar @ ORNL**
  - 150152 compute nodes, 2.1 GHz Opteron
  - Torus network (SeaStar).
  - CNL 2.1, Cray Message Passing Toolkit 3
- **All compiled with "-O3 –mtune=opteron"**

TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

Strong Scaling - Odin (8000^2)



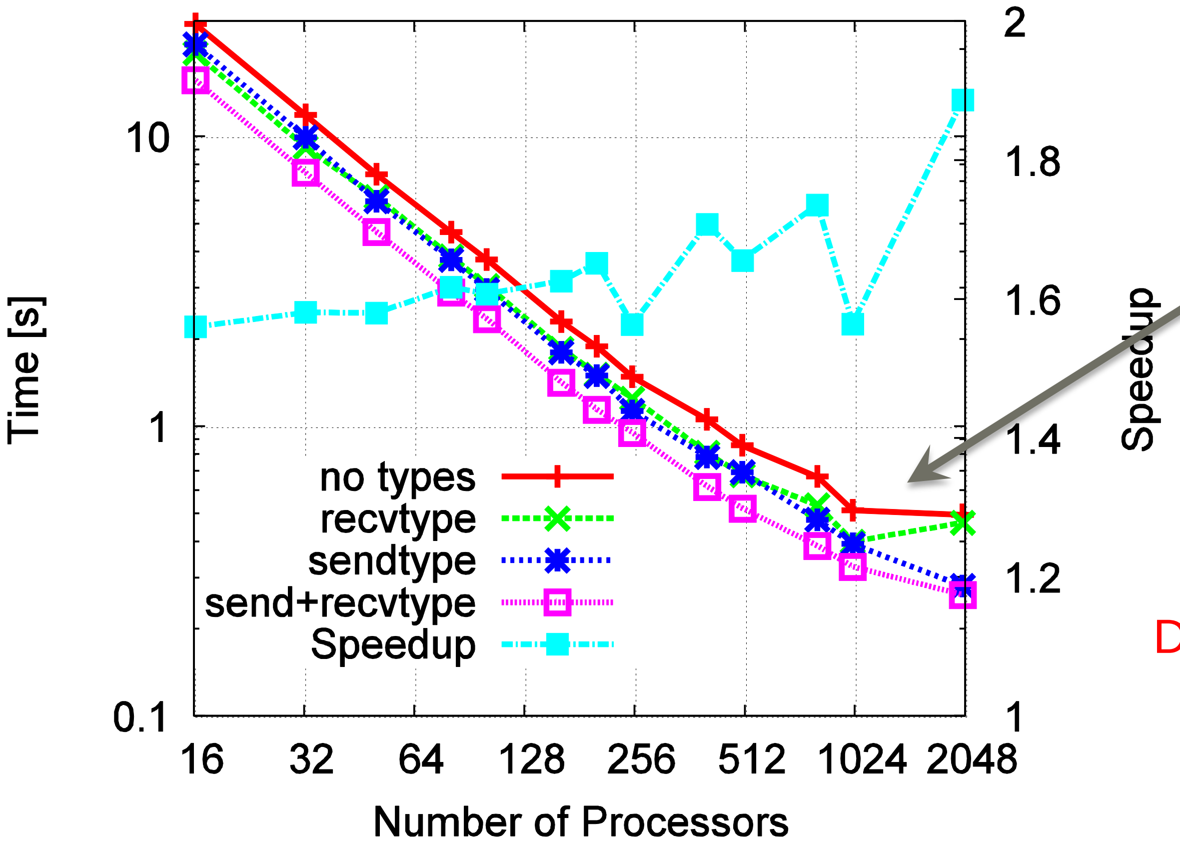
Reproducible peak at P=192

Scaling stops w/o datatypes

- **4 runs, report smallest time, <4% deviation**

TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

Strong Scaling – Jaguar (20k^2)



Scaling stops w/o datatypes

DDT increase scalability

TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

Datatype Conclusions

- **MPI Datatypes allow zero-copy**
  - Up to a factor of 3.8 or 18% speedup!
  - Requires some implementation effort
- **Declarative nature makes debugging hard**
  - Simple tricks like index numbers help!
- **Some MPI DDT implementations are slow**
  - Some nearly surreal (IBM) ☺
  - Complain to your vendor if performance is not consistent!

TH, Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes, EuroMPI'12

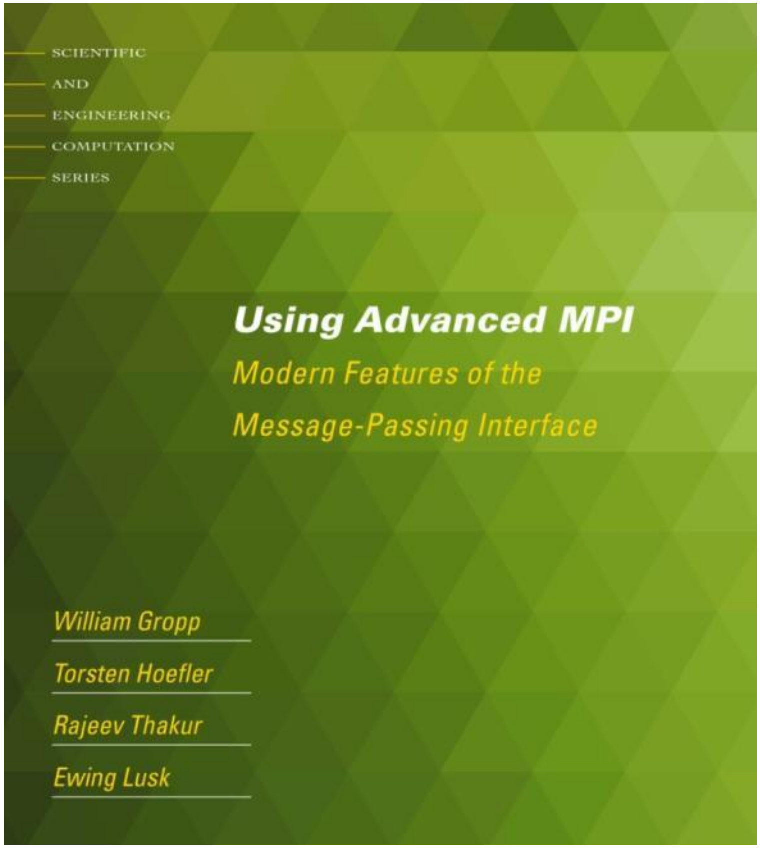
Tutorial Conclusion

- **Thanks for attending!**
  - Ask any questions you have – anytime
  - The book contains all advanced topics *(not datatypes, which are included in the “Using MPI” book)*
  - I hope you enjoyed

SPEEDUP

The SPEEDUP Society:

The Swiss Forum for
High-Performance Computing



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using Advanced MPI

Modern Features of the Message-Passing Interface

William Gropp
Torsten Hoefler
Rajeev Thakur
Ewing Lusk

- All materials (slides, code examples) at:
  http://htor.inf.ethz.ch/teaching/mpi_tutorials/speedup15/