Lecture 25: Strategies for Parallelism and Halo Exchange

William Gropp
www.cs.illinois.edu/~wgropp

What are some ways to think about parallel programming?

- At least two easy ways:
  - ♦ Coarse grained - Divide the problem into big tasks, run many at the same time, coordinate when necessary. Sometimes called “Task Parallelism”
  - ♦ Fine grained - For each “operation”, divide across functional units such as floating point units. Sometimes called “Data Parallelism”

Example – Coarse Grained

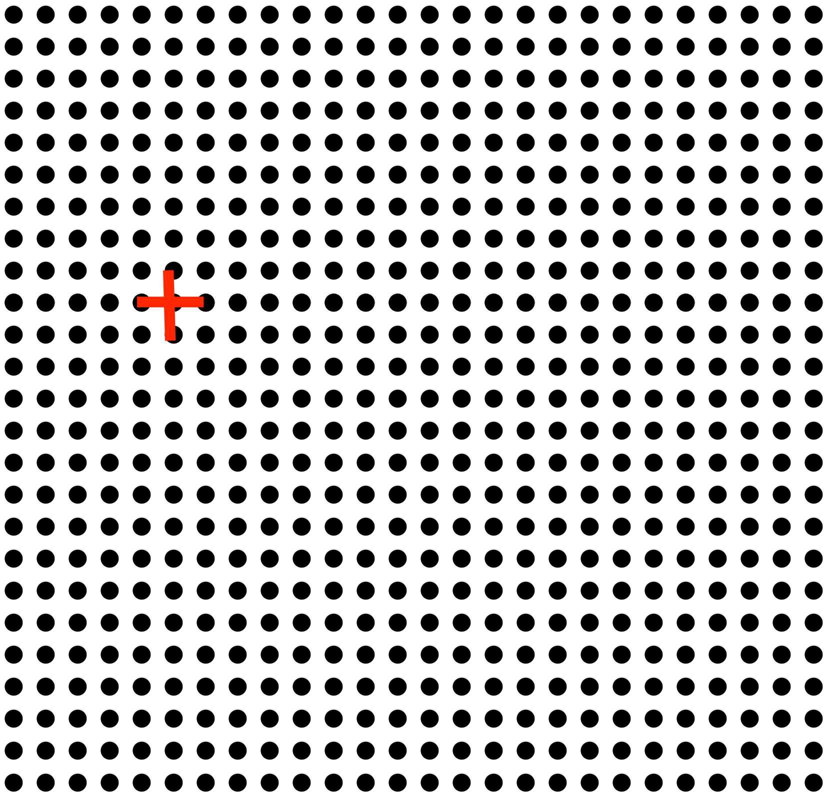
- Set students on different problems in a related research area
  - ♦ Or mail lots of letters – give several people the lists, have them do everything
  - ♦ Common tools include threads, fork, TBB

Example – Fine Grained

- Send out lists of letters
  - break into steps, make everyone write letter text, then stuff envelope, then write address, then apply stamp. Then collect and mail.
  - Common tools include OpenMP, autoparallelization or vectorization
- Both coarse and fine grained approaches are relatively easy to think about

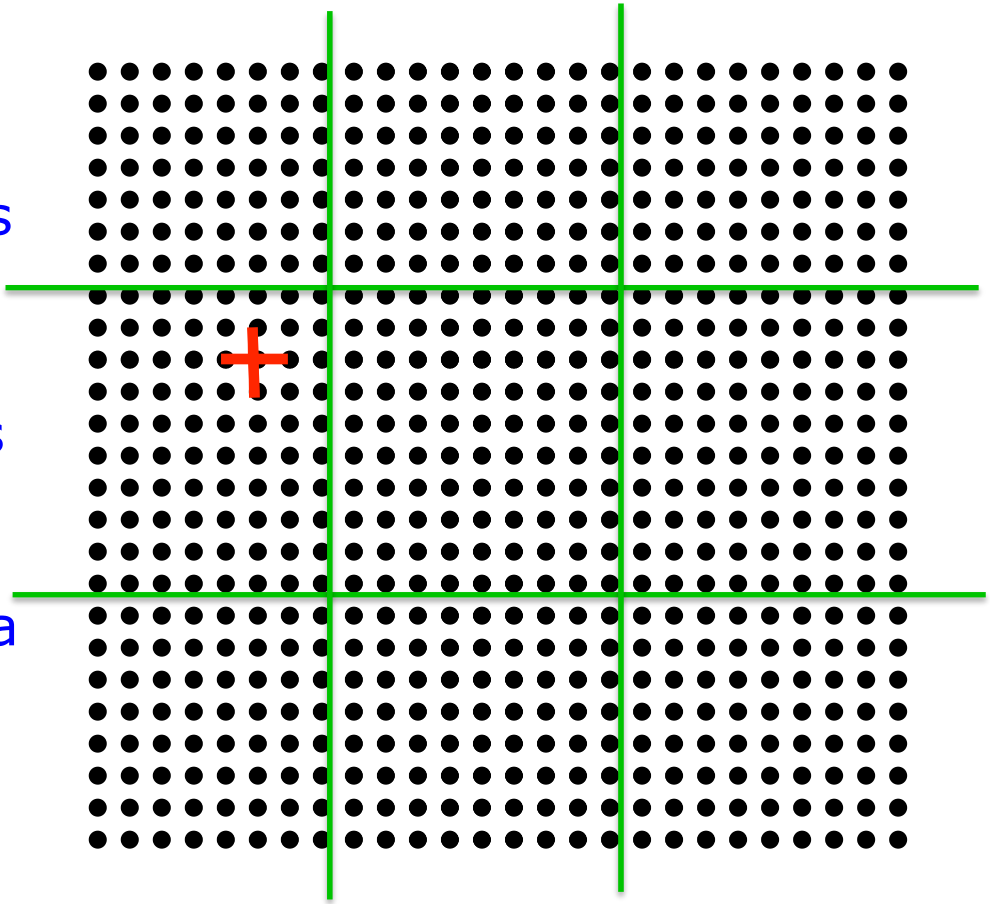
Example: Computation on a Mesh

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.

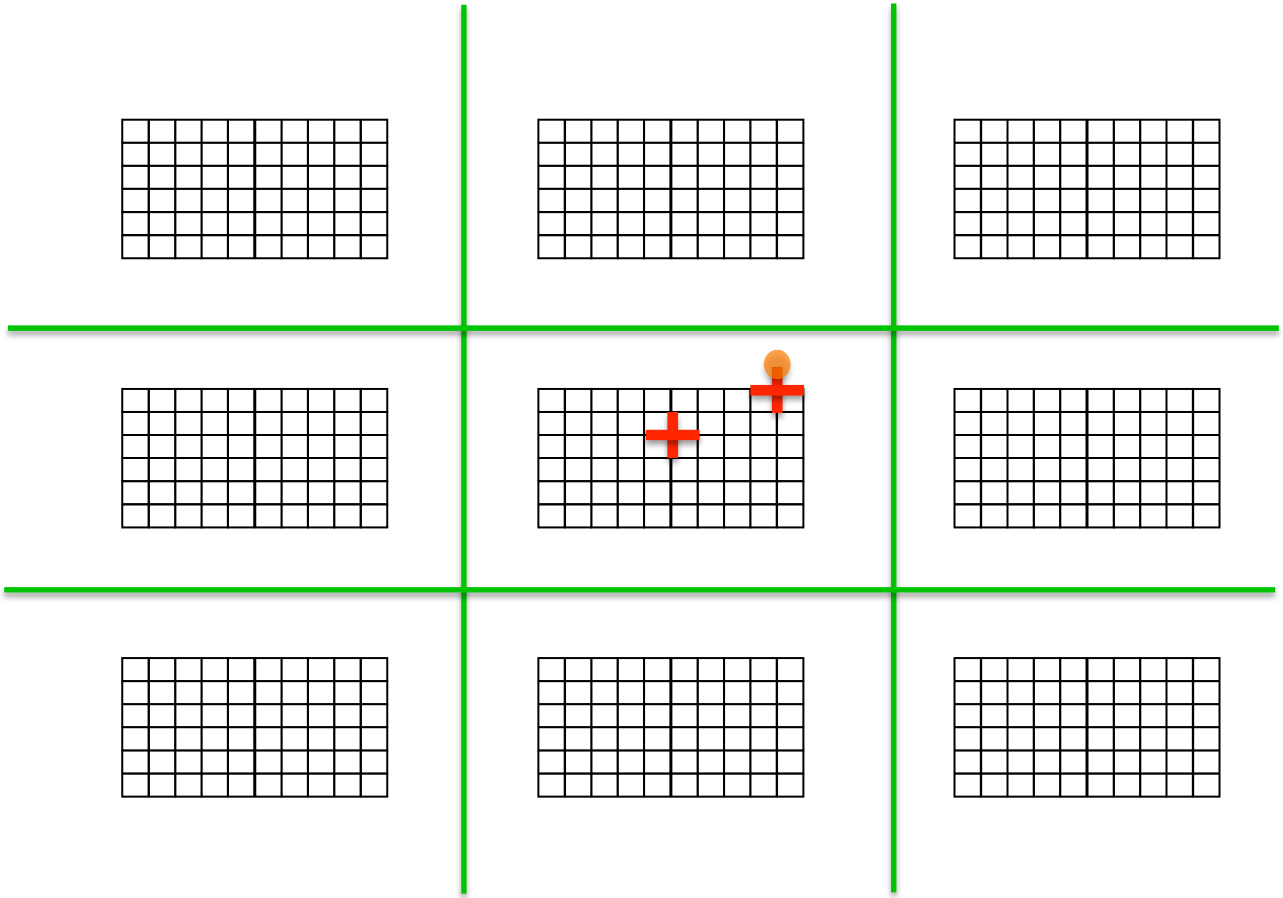


Example: Computation on a Mesh

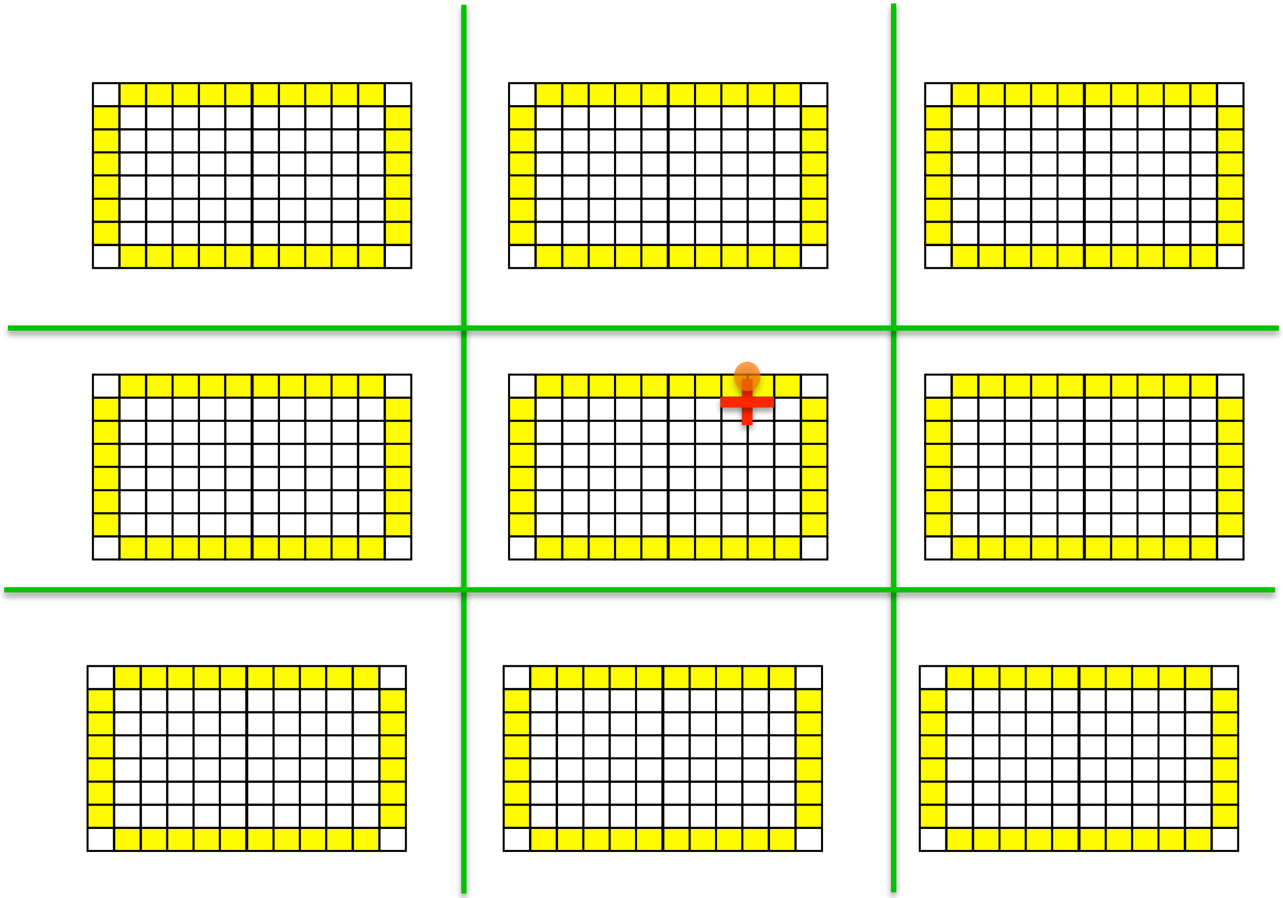
- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red "plus" is called the method's stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.
- Decompose mesh into equal sized (work) pieces



Necessary Data Transfers

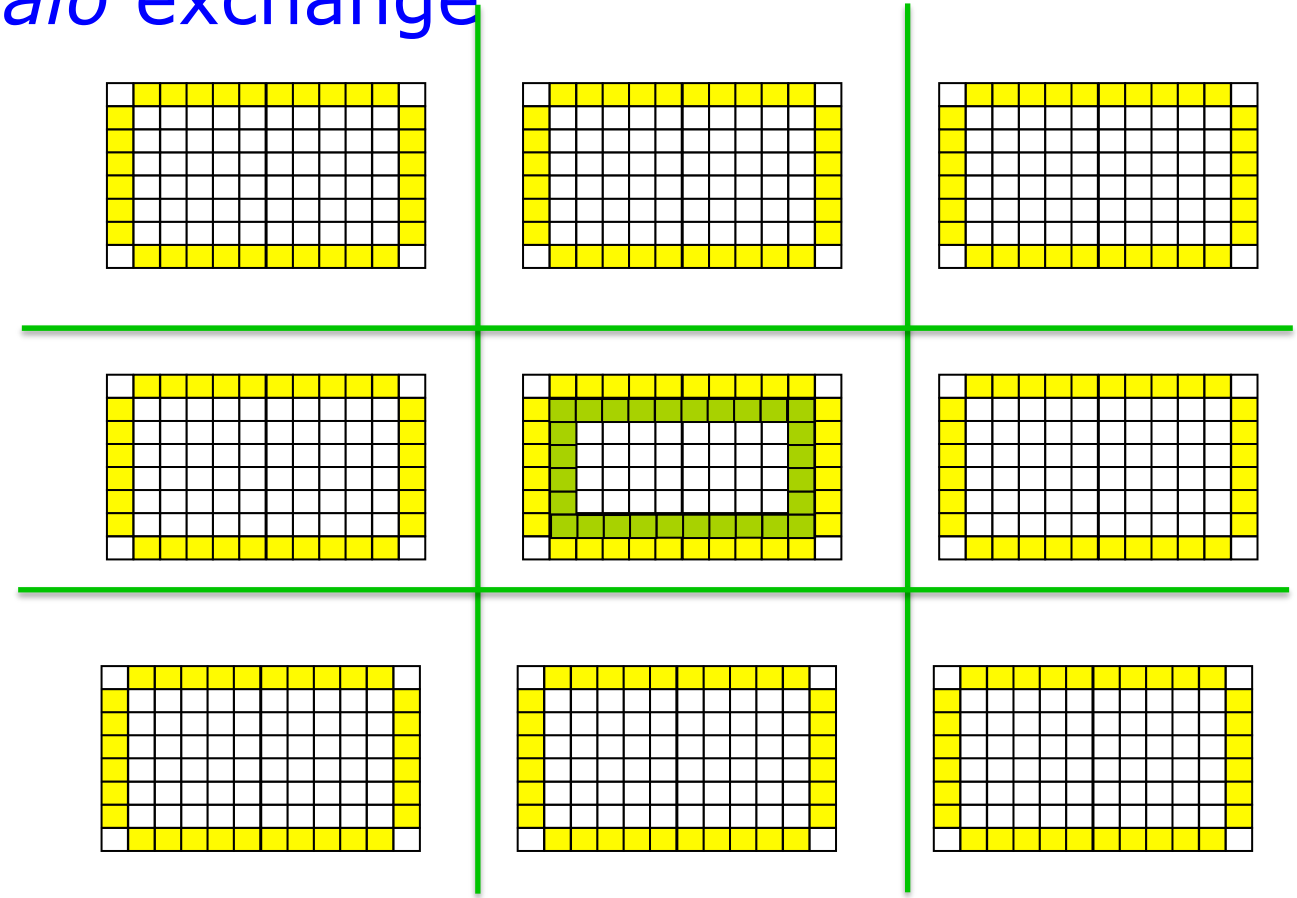


Necessary Data Transfers



Necessary Data Transfers

- Provide access to remote data through a *halo* exchange



PseudoCode

- Iterate until done:
  - ♦ Exchange "Halo" data
    - MPI_Isend/MPI_Irecv/MPI_Waitall or MPI_Alltoallv or MPI_Neighbor_alltoall or MPI_Put/MPI_Win_fence or …
  - ♦ Perform stencil computation on local memory
    - Can use SMP/thread/vector parallelism for stencil computation – E.g., OpenMP loop parallelism

Choosing MPI Alternatives

- MPI offers may ways to accomplish the same task
- Which is best?
  - ♦ Just like everything else, it depends on the vendor, system architecture
  - ♦ Like C and Fortran, MPI provides the programmer with the tools to achieve high performance without sacrificing portability
- The best choice depends on the use:
  - ♦ Consider choices based on system and MPI implementation
  - ♦ Example: Experiments with a Jacobi relaxation example

Tuning for MPI's Send/ Receive Protocols

- Aggressive Eager
  - Performance problem: extra copies
  - Possible deadlock for inadequate eager buffering
  - Ensure that receives are posted before sends
  - MPI_Issend can be used to express "wait until receive is posted"
- Rendezvous with sender push
  - Extra latency
  - Possible delays while waiting for sender to begin
- Rendezvous with receiver pull
  - Possible delays while waiting for receiver to begin

Rendezvous Blocking

- What happens once sender and receiver rendezvous?
  - ♦ Sender (push) or receiver (pull) may complete operation
  - ♦ May block other operations while completing
- Performance tradeoff
  - ♦ If operation does *not* block (by checking for other requests), it adds latency or reduces bandwidth.
- Can reduce performance if a receiver, having acknowledged a send, must wait for the sender to complete a separate operation that it has started.

Tuning for Rendezvous with Sender Push

- Ensure receives posted before sends
  - ♦ better, ensure receives match sends before computation starts; may be better to do sends before receives
- Ensure that sends have time to start transfers
- Can use short control messages
- Beware of the cost of extra messages
  - ♦ Intel i860 encouraged use of control messages with ready send (force type)

Tuning for Rendezvous with Receiver Pull

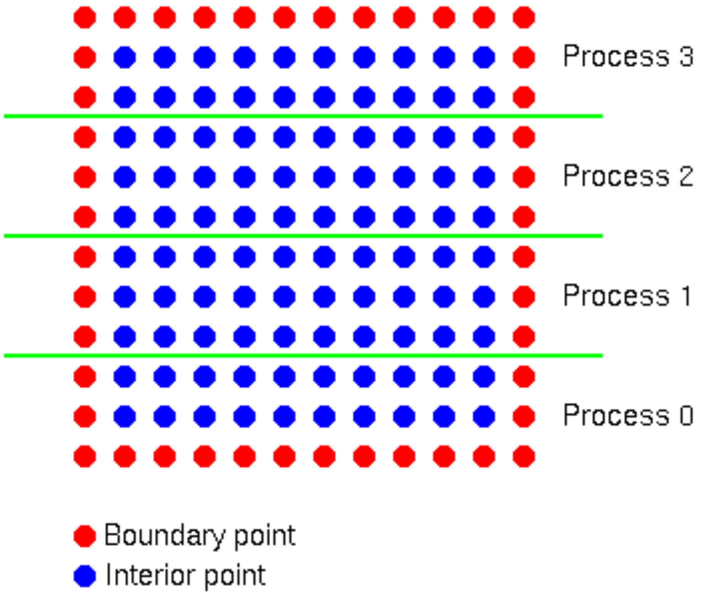
- Place MPI_Isends before receives
- Use short control messages to ensure matches
- Beware of the cost of extra messages

Experiments with MPI Implementations

- Multiparty data exchange
- Jacobi iteration in 2 dimensions
  - ♦ Model for PDEs, Sparse matrix-vector products, and algorithms with surface/volume behavior
  - ♦ Issues are similar to unstructured grid problems (but harder to illustrate)

Jacobi Iteration (C Ordering)

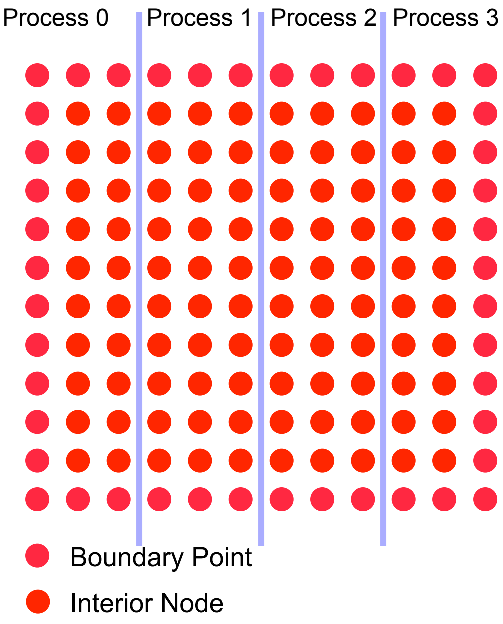
- Simple parallel data structure



- Processes exchange rows with neighbors

Jacobi Iteration (Fortran Ordering)

- Simple parallel data structure



- Processes exchange columns with neighbors
- Local part declared as xlocal(m,0:n+1)

Background to Tests

- Goals
  - ♦ Identify better performing idioms for the same communication operation
  - ♦ Understand these by understanding the underlying MPI process
  - ♦ Provide a starting point for evaluating additional options (there are many ways to write even simple codes)

Some Send/Receive Approaches

- Based on operation hypothesis. Most of these are for polling mode. Each of the following is a *hypothesis* that the experiments test
  - ♦ Better to start receives first
  - ♦ Ensure recvs posted before sends
  - ♦ Ordered (no overlap)
  - ♦ Nonblocking operations, overlap effective
  - ♦ Use of Ssend, Rsend versions (EPCC/T3D can prefer Ssend over Send; uses Send for buffered send)
  - ♦ Manually advance automaton
- Persistent operations

Scheduling Communications

- Is it better to use MPI_Waitall or to schedule/order the requests?
  - Does the implementation complete a Waitall in any order or does it prefer requests as ordered in the array of requests?
- In principle, it should always be best to let MPI schedule the operations. In practice, it may be better to order either the short or long messages first, depending on how data is transferred.

Send and Recv (C)

- Simplest use of send and recv

```
{
   MPI_Status status;
   MPI_Comm ring_comm = mesh->ring_comm;

   /* Send up, then receive from below */
   MPI_Send( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
             ring_comm );
   MPI_Recv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm, &status )
   /* Send down, then receive from above */
   MPI_Send( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm );
   MPI_Recv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
             ring_comm, &status );
}
```

Send and Recv (Fortran)

- Simplest use of send and recv

```
integer status(MPI_STATUS_SIZE)

call MPI_Send( xlocal(1,1), m, MPI_DOUBLE_PRECISION, &
               left_nbr, 0, ring_comm, ierr )
call MPI_Recv( xlocal(1,0), m, MPI_DOUBLE_PRECISION, &
               right_nbr, 0, ring_comm, status, ierr )
call MPI_Send( xlocal(1,n), m, MPI_DOUBLE_PRECISION, &
               right_nbr, 0, ring_comm, ierr )
call MPI_Recv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION, &
                left_nbr, 0, ring_comm, status, ierr )
```

Performance of Simplest Code

- Often very poor performance
  - ♦Rendezvous sequentializes sends/receives

Better to start receives first (C)

- Irecv, Isend, Waitall - ok performance

```
MPI_Status statuses[4];
MPI_Comm ring_comm;
MPI_Request r[4];

   /* Send up, then receive from below */
   MPI_Irecv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm, &r[1] );
   MPI_Irecv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
              ring_comm, &r[3] );
   MPI_Isend( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
              ring_comm, &r[0] );
   /* Send down, then receive from above */
   MPI_Isend( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm,
               &r[2] );
   MPI_Waitall( 4, r, statuses );
}
```

Better to start receives first (Fortran)

- Irecv, Isend, Waitall - ok performance

```
integer statuses(MPI_STATUS_SIZE,4), requests(4)

call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION,&
                        left_nbr, ring_comm, requests(2), ierr )
call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION,&
                        right_nbr, ring_comm, requests(4), ierr )
call MPI_Isend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, &
                        right_nbr, ring_comm, requests(1), ierr )
call MPI_Isend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, &
                        left_nbr, ring_comm, requests(3), ierr )
call MPI_Waitall( 4, requests, statuses, ierr )
```

Ensure recvs posted before sends (C)

- Irecv, Sendrecv/Barrier, Rsend, Waitall

```
void ExchangeInit( mesh )
Mesh *mesh;
{
   MPI_Irecv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm,
              &mesh->rq[0] );
   MPI_Irecv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
             ring_comm, &mesh->rq[1] );
}

void Exchange( mesh )
Mesh *mesh;
{
   MPI_Status statuses[2];

   /* Send up and down, then receive */
   MPI_Rsend( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
              ring_comm );
   MPI_Rsend( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm );

   MPI_Waitall( 2, mesh->rq, statuses );
}

void ExchangeEnd( mesh )
Mesh *mesh;
{
   MPI_Cancel( &mesh->rq[0] );
   MPI_Cancel( &mesh->rq[1] );
}
```

Ensure recvs posted before sends (Fortran)

- Irecv, Sendrecv/Barrier, Rsend, Waitall

```
integer statuses(MPI_STATUS_SIZE,2), requests(2)
! Post initial Irecv’s (not shown)
do while (.not. Converged)
   ….
   call MPI_Rsend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, &
                        right_nbr, ring_comm, ierr )
   call MPI_Rsend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, &
                        left_nbr, ring_comm, ierr )
   call MPI_Waitall( 2, requests, statuses, ierr )
   ! Process ghost points
   call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION,&
                        left_nbr, ring_comm, requests(1), ierr )
   call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION,&
                        right_nbr, ring_comm, requests(2), ierr )
   call MPI_Allreduce( … )
enddo
```

Use of Ssend versions (C)

- Ssend allows send to wait until receive ready
  - At least one (ancient) implementation (T3D) gives better performance for Ssend than for Send

```
void Exchange( mesh )
Mesh *mesh;
{
   MPI_Status  status;

   /* Send up, then receive from below */
   MPI_Irecv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm, &rq );
   MPI_Ssend( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
             ring_comm );
   MPI_Wait( &rq, &status );
   /* Send down, then receive from above */
   MPI_Irecv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
             ring_comm, &rq );
   MPI_Ssend( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm );
   MPI_Wait( &rq, &status );
}
```

Use of Ssend versions (Fortran)

- Ssend allows send to wait until receive ready
  - At least one (ancient) implementation (T3D) gives better performance for Ssend than for Send
-
  ```
  integer status(MPI_STATUS_SIZE), request
  call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION, left_nbr, 0,&
                     ring_comm, request, ierr )
  call MPI_Ssend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, right_nbr, &
                      0, ring_comm, ierr )
  call MPI_Wait( request, status, ierr )
  call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION, &
                     right_nbr, 0, ring_comm, request, ierr )
  call MPI_Ssend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, left_nbr, &
                      0, ring_comm, ierr)
  call MPI_Wait( request, status, ierr )
  ```

Nonblocking Operations, Overlap Effective (C)

- Isend, Irecv, Waitall
- A variant uses Waitsome with computation

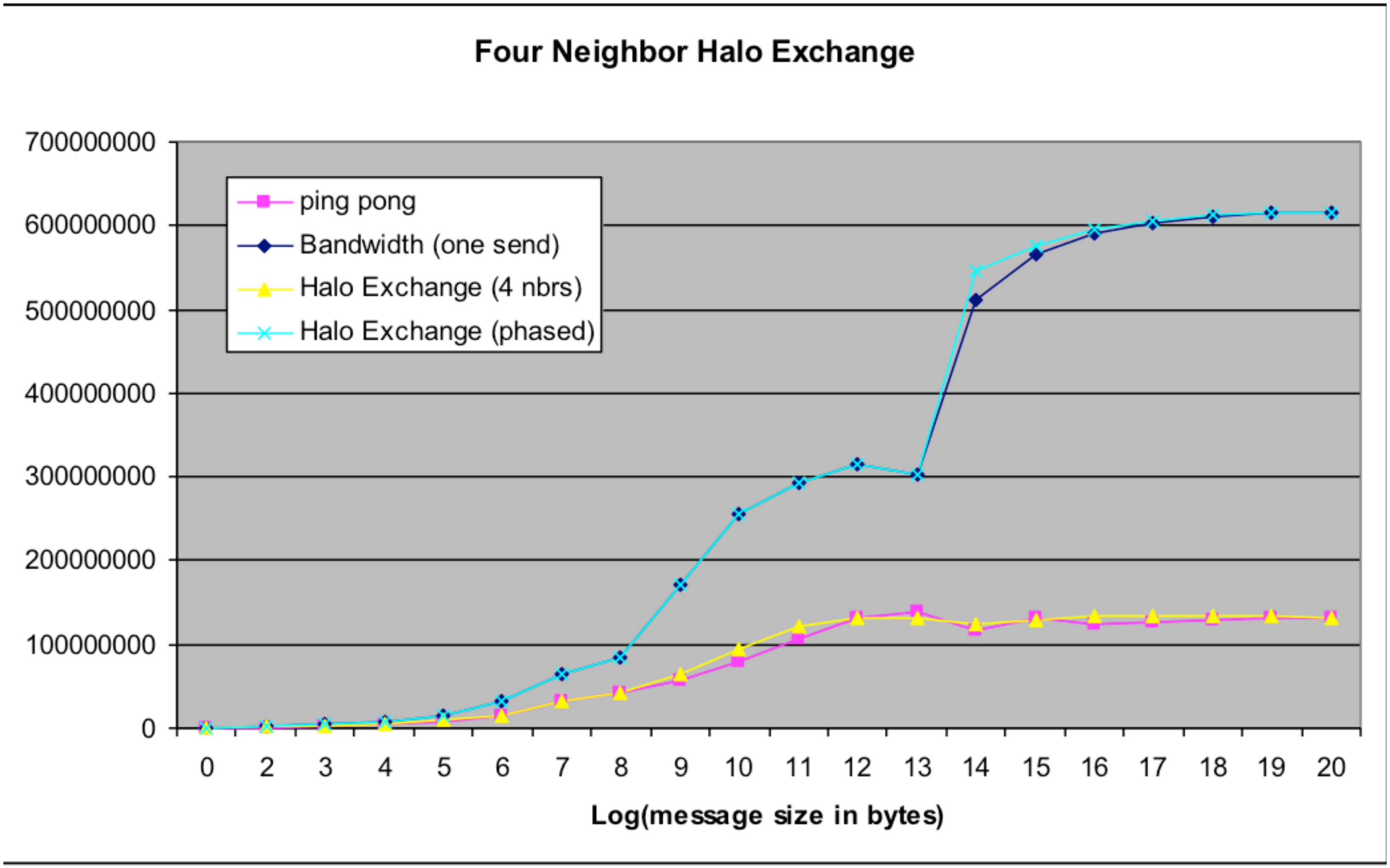
```
void ExchangeStart( mesh )
Mesh *mesh;
{
     /* Send up, then receive from below */
    MPI_Irecv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm,
                   &mesh->rq[0] );
    MPI_Irecv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
                  ring_comm, &mesh->rq[1] );
    MPI_Isend( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
                   ring_comm, &mesh->rq[2] );
    MPI_Isend( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm,
                   &mesh->rq[3] );
}

void ExchangeEnd( mesh )
Mesh *mesh;
{
    MPI_Status statuses[4];
    MPI_Waitall( 4, mesh->rq, statuses );
}
```

Nonblocking Operations, Overlap Effective (Fortran)

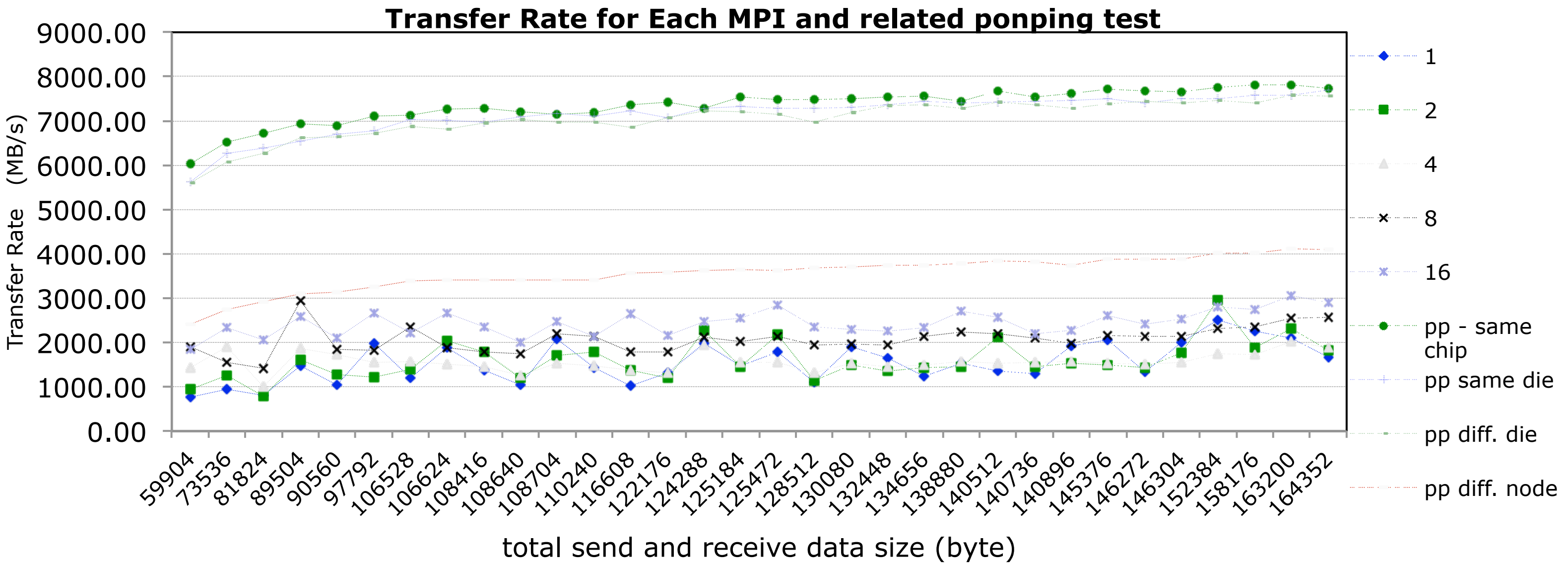
- Isend, Irecv, Waitall
- A variant uses Waitsome with computation

```
integer statuses(MPI_STATUS_SIZE,4), requests(4)
call MPI_Irecv( xlocal(1,0), m, MPI_DOUBLE_PRECISION, left_nbr, 0,&
                         ring_comm, requests(1), ierr )
call MPI_Isend( xlocal(1,n), m, MPI_DOUBLE_PRECISION, right_nbr, &
                         0, ring_comm, requests(2), ierr )
call MPI_Irecv( xlocal(1,n+1), m, MPI_DOUBLE_PRECISION, &
                        right_nbr, 0, ring_comm, requests(3), ierr )
call MPI_Isend( xlocal(1,1), m, MPI_DOUBLE_PRECISION, left_nbr, &
                         0, ring_comm, requests(4), ierr)
… computation ...
call MPI_Waitall( 4, requests, statuses, ierr )
```

How Important is Using Nonblocking Communication?

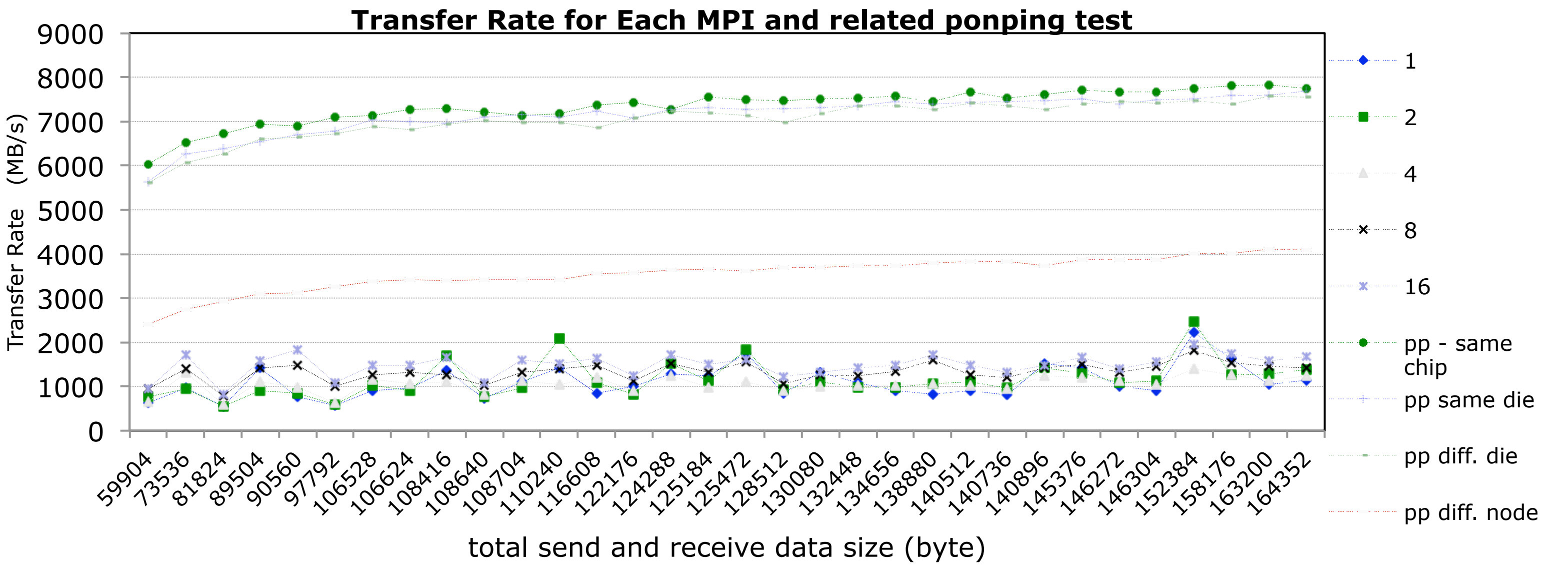


And For Blue Waters: Using Isend/Irecv



- Halo exchange performance lower than ping-pong, but it could be worse …

Blue Waters: Using Send/Irecv



Persistent Operations

- Potential saving
  - ♦ Allocation of MPI_Request
  - ♦ Validating and storing arguments
  - ♦ Fewer interactions with "polling" engine
- Variations of example
  - ♦ sendinit, recvinit, startall, waitall
  - ♦ startall(recvs), sendrecv/barrier, startall(rsends), waitall
- Some vendor implementations are buggy
- Persistent operations may be slightly *slower*
  - ♦ if vendor optimizes for non-persistent operations

Summary

- Many different ways to express the same communication pattern in MPI
  - This is a *feature*, not a flaw
- Different systems will optimize different patterns
  - Try to design application code to allow different implementations
  - Here, used “Exchange” routine

Overdecomposition

- We’ve used a decomposition that assigns exactly one “patch” to an MPI process
- Any load imbalance between processes will result in idle (wasted) time on all but the slowest process.
- One option is to subdivide each patch into smaller patches, and redistribute patches from slower processes to faster ones.
- This approach is called *overdecomposition* because it decomposes the original domain into more parts than is required for distributed memory parallelism to the individual MPI processes

Discussion

- Try the different MPI communication approaches on your system. How do they perform?
- Does the performance depend on whether the MPI processes are on the same chip? Node?