Message Passing

Bruno Raffin
-
M2R Parallel Systems

UGA

History

Message Passing Programming first became popular through PVM (Parallel virtual machine), a library initially developed in 1989 by Oak Ridge National Laboratory (USA).

PVM emerged with PC clusters. PVM was easy to understand, portable (TCP/IP in particular) and enabled to run parallel programs on the machines of any lab as long as interconnected through an Ethernet network (no need for an expensive parallel machine).

The Message Passing Interface (MPI) is a standard API, an initiative started in 1992. Since, we when through 3 major releases:

MPI-1.0 (1994) MPI-2.0 (1997) MPI-3.0 (2012)

MPI standardizes the API, not the implementation. Many different implementation exist, open source ones (MPICH, OpenMPI, MPC) as well as closed source ones (from Cray, IBM, Intel)

MPI Success

More than 90% of parallel applications are developed with MPI

MPI is a standard API, implementations are libraries (no specific compiler required)

MPI concepts are easy to understand

MPI contains many functions, but classical applications usually only need a few of them only.

MPI programs are portable across a very large range of machines (from DIY clusters of commodity components up to the #1 top500 supercomputer).

MPI
(Message Passing Interface)

Standard: http://www.mpi-forum.org

Open Source Implementations:

- OpenMPI (https://www.open-mpi.org/): probably the most popular
- MPC (http://mpc.hpcframework.paratools.com/)

Discrepancies between the standard and the implementations:

- A given implementation may not implement the full standard
- Some features may be implemented but not necessarily efficiently

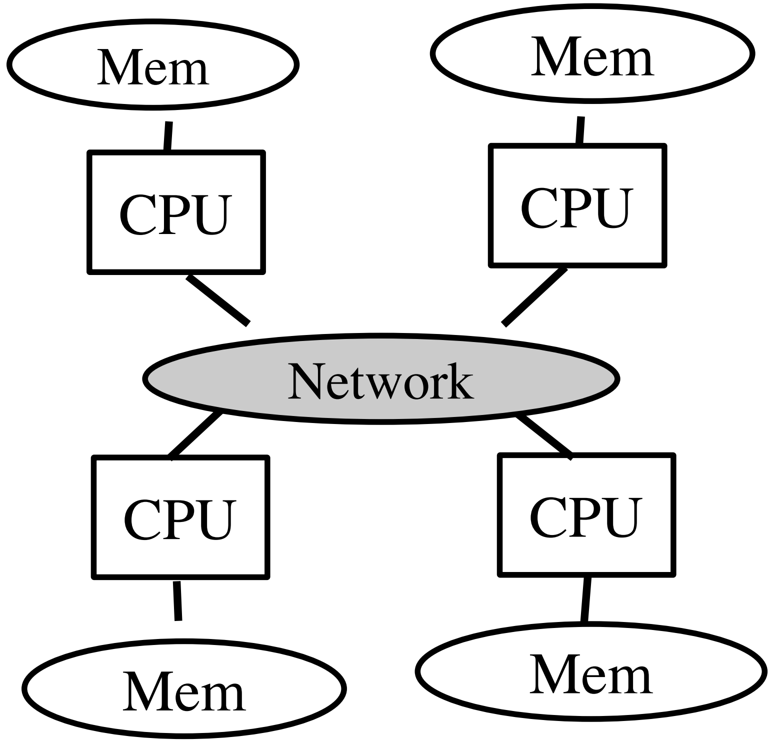
The MPI Model

The MPI (simple) Machine Model:

- Distributed memory
- Single processor/core nodes
- Uniform nodes all fully available at 100%
- Fully interconnected network.

MPI Program: MIMD in SPMD (single program multiple data) mode

We will see that this model is over-simplified and that the programmer actually needs to better take into consideration the architecture of the target machine to get performance.



Distributed memory machine

MPI Hello Word

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

Compile + Exec

To compile MPI : `mpicc  prog.c`

To execute locally with 2 processes:

```
mpirun -np 2  -machinefile fichier_machine  a.out
```

To execute 4 processes on distant machines:

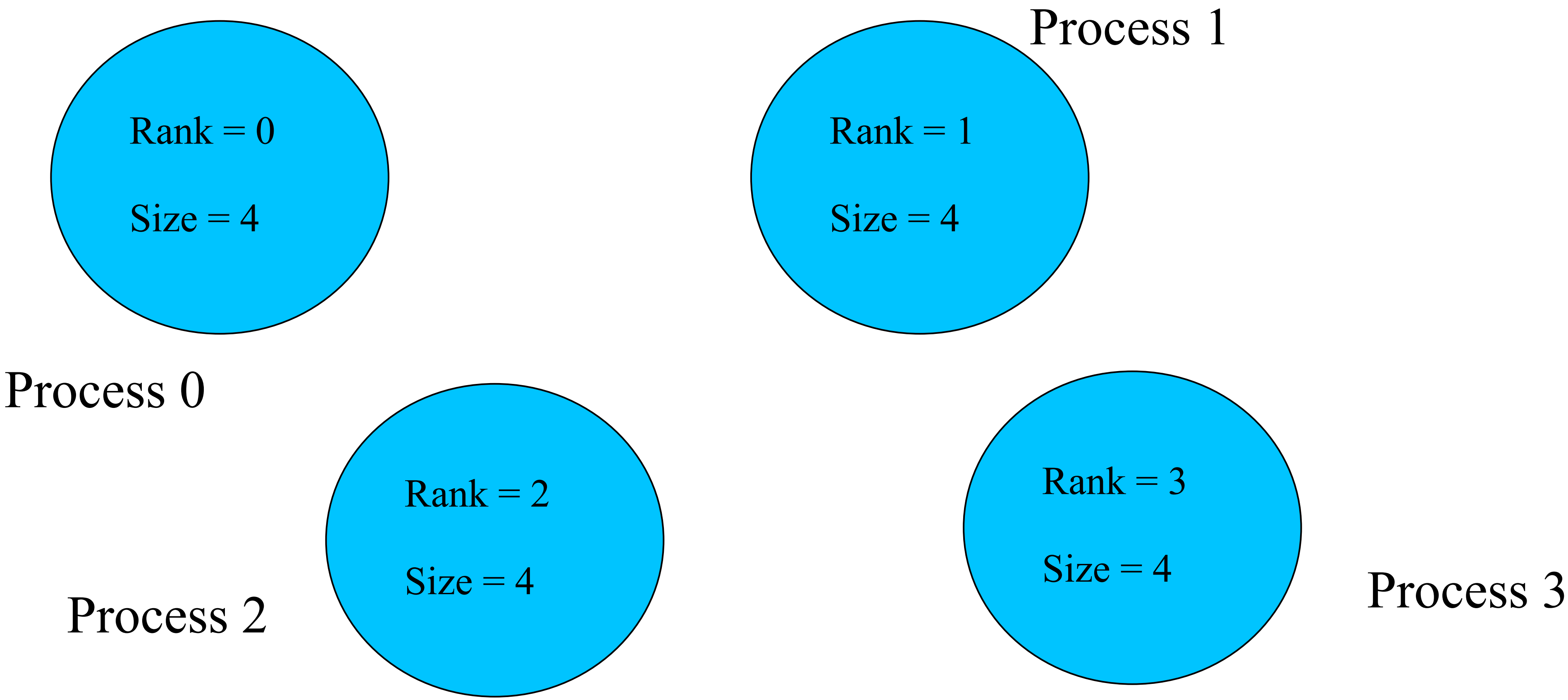
```
mpirun –machinefile hostfile  a.out
```

Remarks:

- mpirun/mpicc are not part of the standard (options may change between implementations)
  (standardized in MPI-2, mpiexec and not mpirun, but not always adopted)
- Parallel machine use a batch scheduler. Some provide an integrated mpi launcher (by the way what is a batch scheduler ?)

Process Identification

MPI rank its processes from 0 to N-1.



Remark: all variables are always local to each process (distributed address space)

MPI Hello Word 2

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Question: how to order outputs using MPI_Barrier (global sync) ?

Point-to-Point Message Exchange

- Send: explicitly send a message (buffer of data) to a given process
- Receive: explicitly receive a message from a given process

```
if (myrank == 2){
X = 1;
Send X to 1;
}
if myrank == 1){
Receive X from 2 into Y;
}
```

Blocking instructions:

- Send call ends when it as the guarantee the message will be properly delivered.
- Receive call ends when message received.

MPI_Send MPI_Recv

Destination rank

```
MPI_Send(&mess_env,3,MPI_INT,2,tag,MPI_COMM_WORLD)
```

Message to send: 3 integers starting at address &mess_env

Message tag (user defined)

Communicator (MPI defined)

Message to receive: 3 integers to be stored at address &mess_rec

Info about the received message

```
MPI_Recv(&mess_rec,3,MPI_INT,1,tag,MPI_COMM_WORLD,&stat)
```

sender Rank

Tags and Communicators

Tag : An integer to define a message class (a matching send/receive require matching tags)

How to make sure than the tag I use is not used in a library (that I also use) – could lead to unexpected communication match ?

Communicator : a set of processes. When starting an MPI application always include all processes in a default communicator called MPI_COMM_WORLD. From this communicator new ones can be created.

- A communicator can be seen as a system defined tag (MPI ensures each communicator is unique)
- Collective communications rely on communicators (ex: broadcast)

Datatypes

MPI comes with several base data types, as well as the possibility to build descriptions of complex data structures.

- Relying on MPI types rather than the host language ones, enables MPI to properly handle communications between processes on machines with very different memory representations and lengths of elementary datatypes.
- Specifying application-oriented layout of data in memory open the way to implementation optimization
  - Serialization/deserialization only when and where "necessary" (less memory copies)
  - Use of special hardware (scatter/gather) when available

Again, MPI is only a standard API, actual implementations may show different degree of efficiency when dealing with complex data structures (sometimes more efficient to serialize/deserialize by hand)

MPI Hello Word 3

```
…
if (rank == 0 ) {

     MPI_Send(&buf, 1, MPI_INT,1,0,MPI_COMM_WORLD);
     printf(”%d sent message to%d\n”,rank,1);

else if (rank == 1) {

      MPI_Recv(&buf, 1, MPI_INT,0,0,MPI_COMM_WORLD,
                                              MPI_STATUS_IGNORE);
      printf(”%d received message from %d\n”,rank,0);
  }
…
```

Communication Matching

Process 1 :

```
MPI_Send(&m1,3,MPI_INT,2,tag_0,MPI_COMM_WORLD);
MPI_Send(&m2,3,MPI_INT,2,tag_1,MPI_COMM_WORLD);
MPI_Send(&m3,3,MPI_INT,2,tag_1,MPI_COMM_WORLD);
MPI_Send(&m4,1,MPI_CHAR,2,tag_0,MPI_COMM_WORLD);
```

Signature of messages sent to process2

1*MPI_CHAR,tag_0,MPI_COMM_WORLD
3*MPI_INT,tag_1,MPI_COMM_WORLD
3*MPI_INT,tag_1,MPI_COMM_WORLD
3*MPI_INT,tag_0,MPI_COMM_WORLD

Process 2 :

Process 2 look for the oldest message with the signature `3*MPI_INT,tag_1,MPI_COMM_WORLD`

```
MPI_Recv(&mess_recu,3,MPI_INT,1,tag_1,MPI_COMM_WORLD,&stat)
```

Does this program deadlock ?

Buffer Management
(implementation dependent)

Message to send

MPI_Send can finish
once the message is
safely stored

Buffer

Signature + Data

Where is this buffer actually located ?

MPI_Recv block as long as
expected data are not stored
in the reception space

Space to store the received message

Buffer Management
(implementation dependent)

Message to send

MPI_Send blocks as long as the message is not stored at its destination

Buffer

Signature + address of the message

MPI_Recv block as long as expected data are not stored in the reception space

When does this implementation make more sense than the previous one ?

Space to store the received message

Strategy 1 is usually used for small messages

- \+ **Send unlocks sooner**
- \- **Need a message copy**

Strategy 2 is usually used for large messages

- \+ **No message copy**
- \- **Send locks more time.**

For some implementations the tilting point between the two strategies can be adjusted.

In the 90's CRAY, shipped his machines with MPI configured to use the strategy 1 for all message sizes. Why ?

MPI Hello Word 4

```
…
if (rank == 0 ) {

      MPI_Send(&buf, 10000, MPI_INT,1,0,MPI_COMM_WORLD);
      MPI_Recv(&buf2, 10000, MPI_INT,1,0,MPI_COMM_WORLD,
                                            MPI_STATUS_IGNORE);

else if (rank == 1) {

       MPI_Send(&buf, 10000, MPI_INT,0,0,MPI_COMM_WORLD);
       MPI_Recv(&buf2, 10000, MPI_INT,0,0,MPI_COMM_WORLD,
                                            MPI_STATUS_IGNORE);
}
…
```

What do you think about this program ?
How would you improve it ?

Asynchronous Send/Receive

```
…
     MPI_Send(&buf, 10000, MPI_INT,(rank+1)%size,
                            0,MPI_COMM_WORLD);
     MPI_Recv(&buf2, 10000, MPI_INT, (rank+size-1)%size,
                            0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
…
```

And this one ?

Asynchronous Send/Receive

```
…
     MPI_ISend(&buf, 10000, MPI_INT,(rank+1)%size,
                          0,MPI_COMM_WORLD,&request);
     MPI_Recv(&buf2, 10000, MPI_INT, (rank+size-1)%size,
                          0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
…
```

MPI comes with non blocking send and receive instructions:

- MPI_ISEND and MPI_IRECV

and blocking or not instructions for querying a reception status.

- MPI_PROBE, MPI_WAIT, MPI_WAITANY, MPI_TESTANY, MPI_TESTALL, MPI_WAITALL

Asynchronous Send/Receive

…

```
MPI_Status status;
MPI_Request request,request2;
MPI_IRecv(&buf, 10000, MPI_INT, (rank+size-1)%size,
                  0,MPI_COMM_WORLD, &request);
MPI_ISend(&buf2, 10000, MPI_INT,(rank+1)%size,
                  0,MPI_COMM_WORLD,&request2);
MPI_WAIT(&request,&status);
MPI_WAIT(&request2,&status);
```

…

Why did I put the reception before the send (immediate does not necessarily mean asynchronous) ?

Communication/Computation Overlapping

…

```
MPI_Status status;
MPI_Request request,request2;
MPI_IRecv(&buf, 10000, MPI_INT, (rank+size-1)%size,
                  0,MPI_COMM_WORLD, &request);
MPI_ISend(&buf2, 10000, MPI_INT,(rank+1)%size,
                  0,MPI_COMM_WORLD,&request2);
// Do some useful computations here ….
something_useful();
MPI_WAIT(&request,&status);
MPI_WAIT(&request2,&status);
```

…

What is the max performance increase I could reach using this kind of idea ?

Try to play with these small programs on your laptop and/or simgrid:

- Put a loop around to repeat the pattern several times
- Use MPI_WTIME to time it (how do you measure the exec time of a parallel program ?)
- Play with the message size, ….

(Non-)Determinism

- What is non-determinism ?

- Are MPI programs deterministic ?
  - On reception, instead of specifying a given message source, MPI provide the wildcard MPI_ANY_SOURCE
  - Clock or random numbers usage ?
  - Other ?

Cost Model for Point-to-point Com. The "Hockney" Model

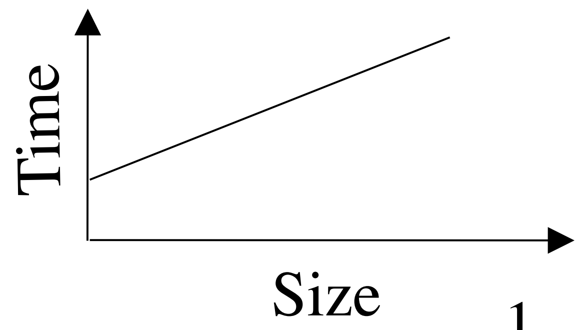
Hockney [Hoc94] proposed the following model for performance eval- uation of the Paragon. A message of size m from P_i to P_j requires:

$$t_{i,j}(m) = L_{i,j} + m/B_{i,j}$$

The homogeneous version is often used for comparing the cost of communication patterns:

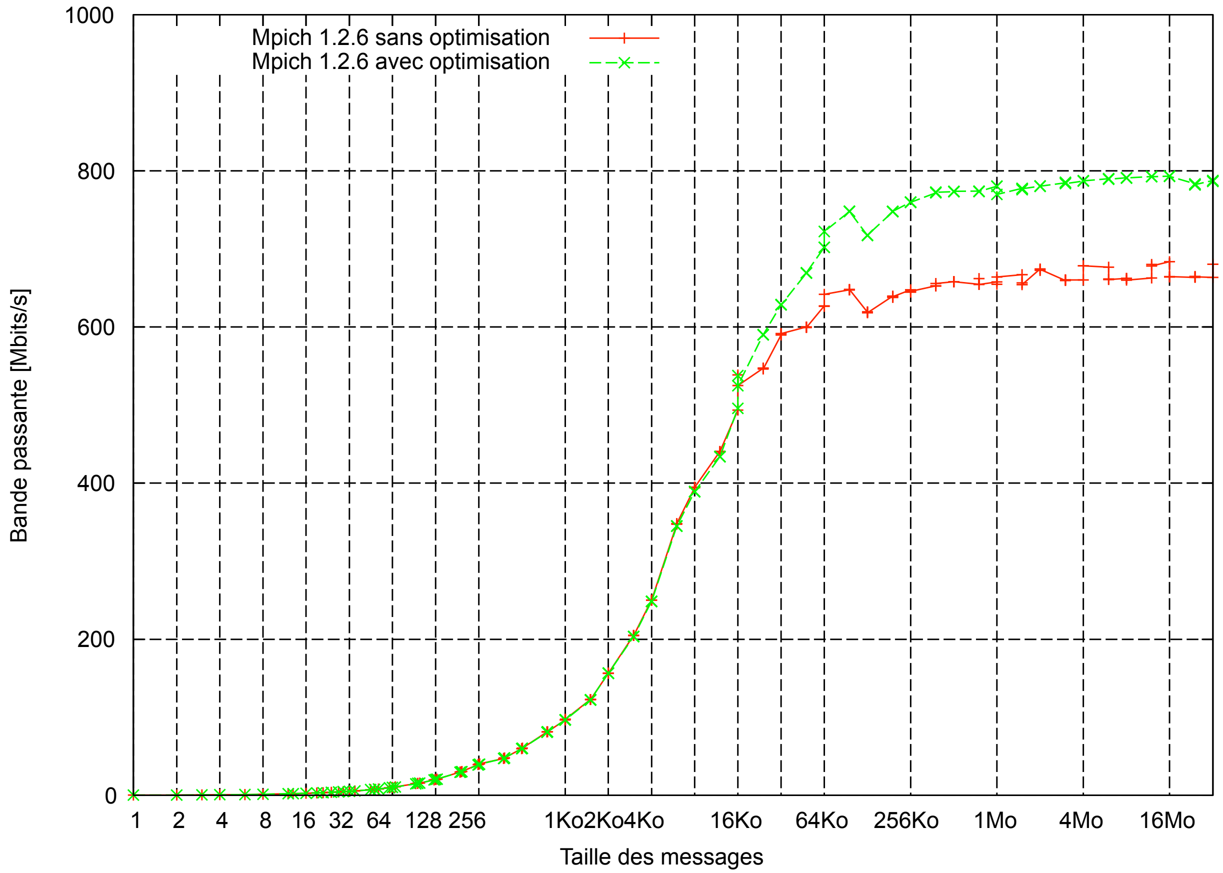
$$t(m) = L + m/B$$

How would you measure L and B with MPI ?



Bandwidth as a Function of Message Size

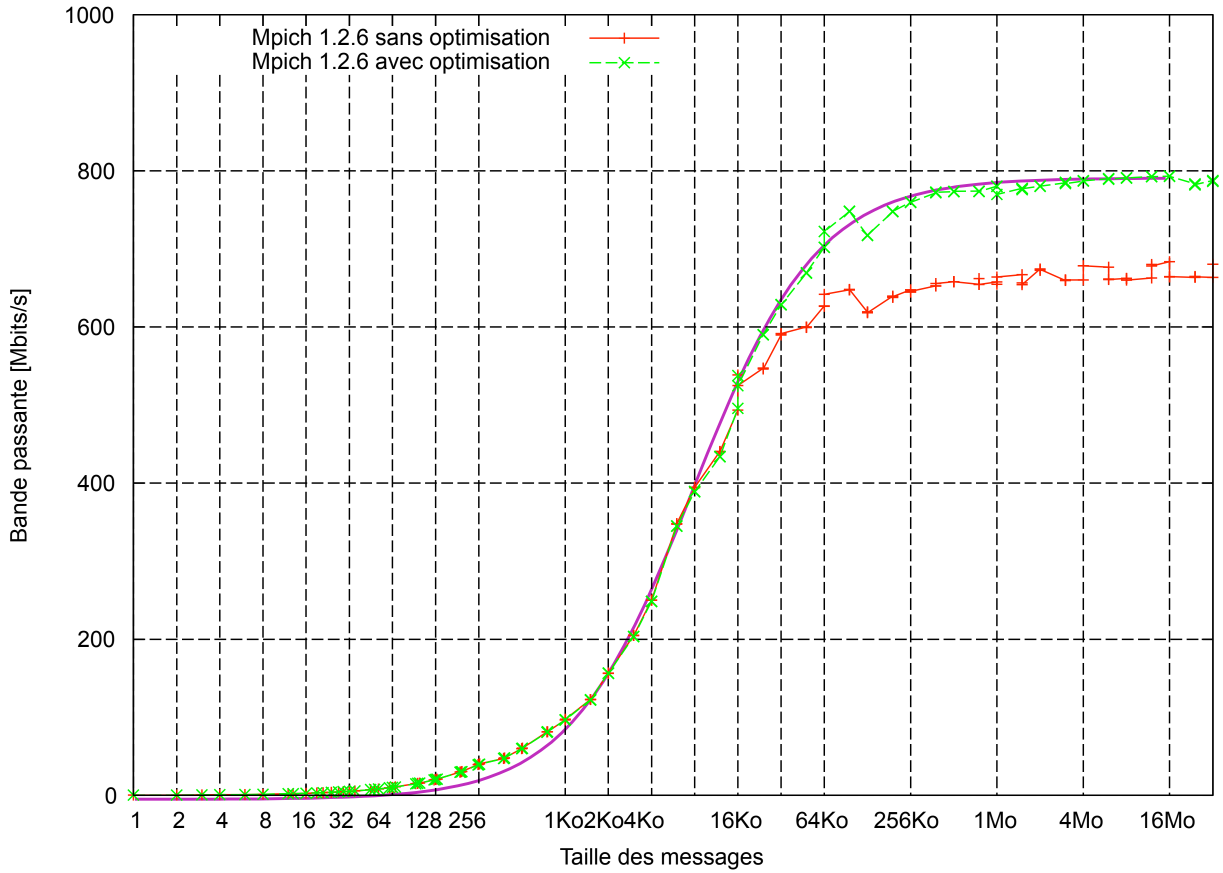
With the Hockney model: $\frac{m}{L+m/B}$.



MPICH, TCP with Gigabit Ethernet

Bandwidth as a Function of Message Size

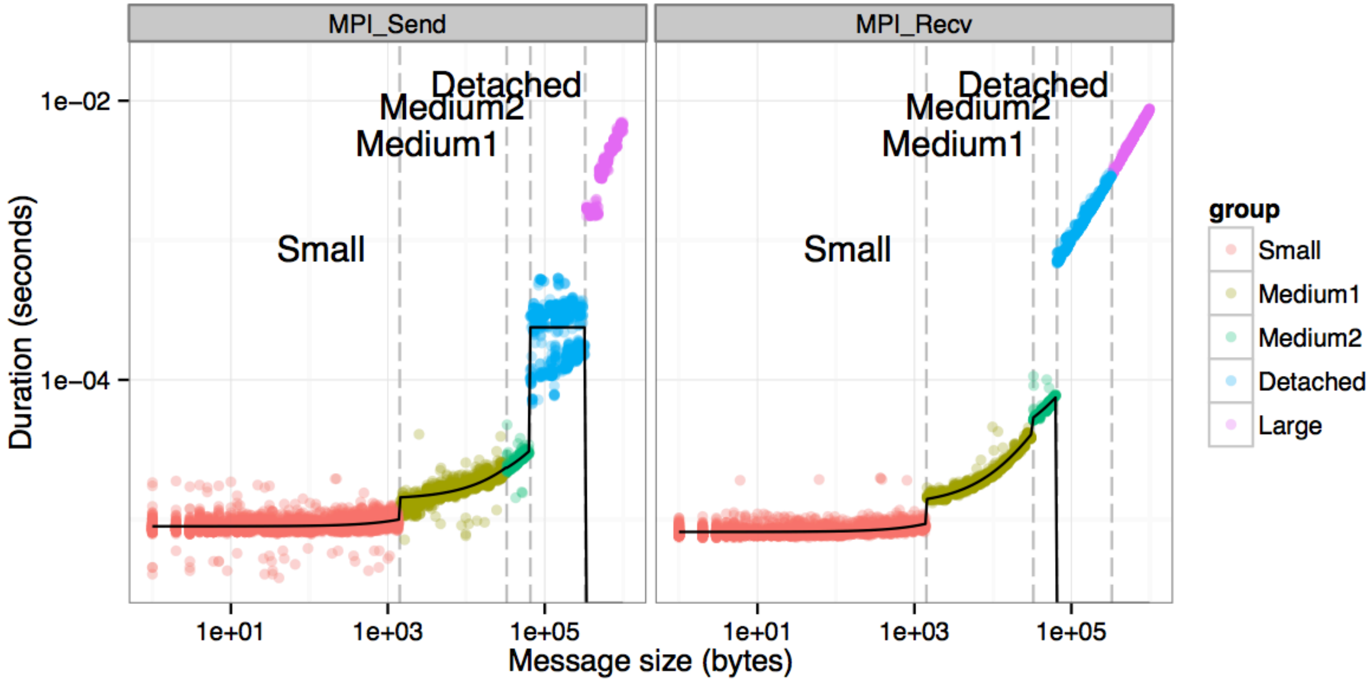
With the Hockney model: $\frac{m}{L+m/B}$.



MPICH, TCP with Gigabit Ethernet

More measures ...

Randomized measurements (OpenMPI/TCP/Eth1GB) since we are not interested in peak performances but in performance characterization

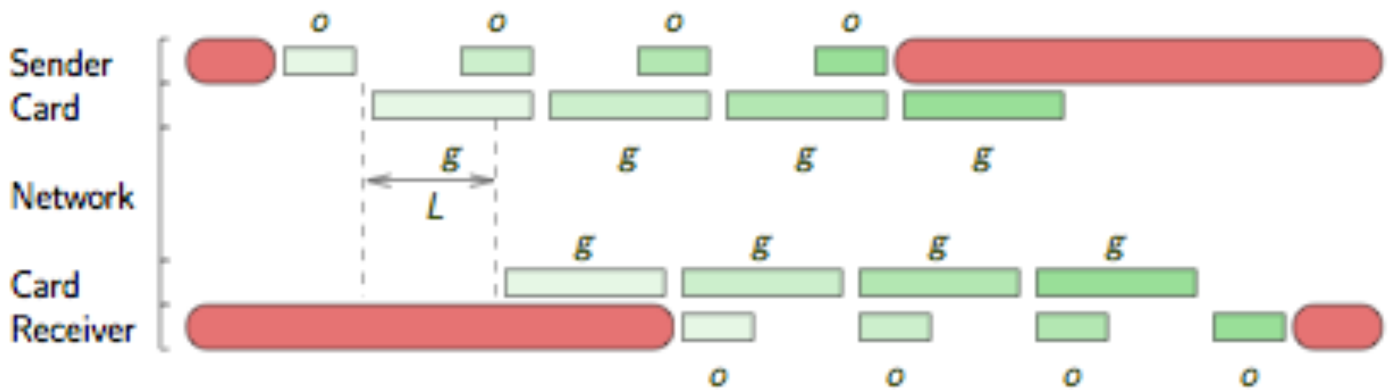


- There is a quite important variability
- There are at least 4 different modes
- It is piece-wise linear and discontinuous

LogP

The LogP model [CKP+96] is defined by 4 parameters:

- L is the network latency (time to communicate a packet of size w)
- o is the middleware overhead (message splitting and packing, buffer management, connection, . . .) for a packet of size w
- g is the gap (the minimum) between two packets of size w
- P is the number of processors



Sending m bytes with packets of size w:

$$2o + L + \left\lceil \frac{m}{w} \right\rceil \cdot \max(o, g)$$

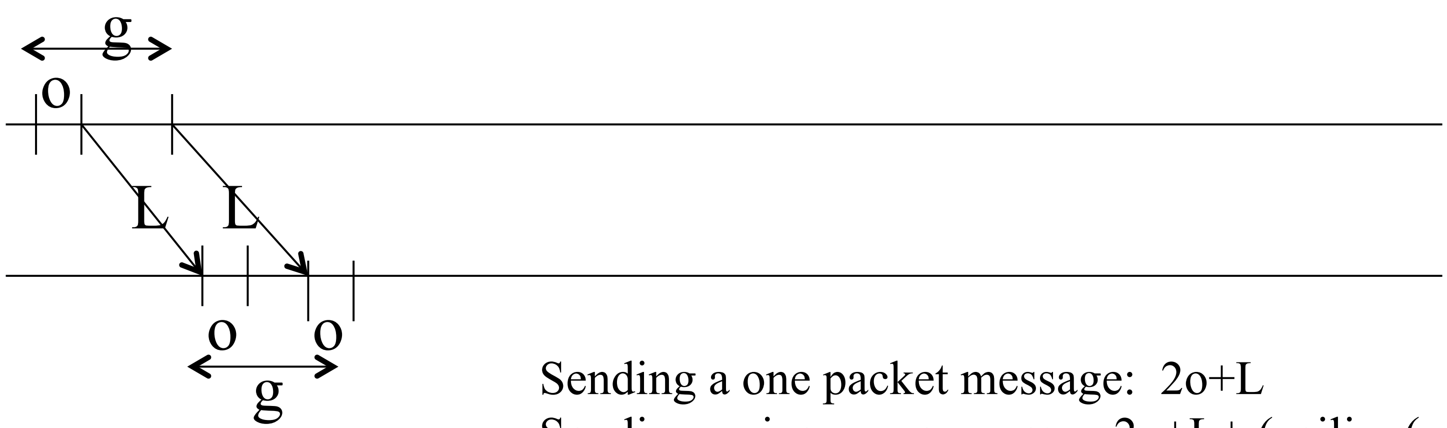
Occupation on the sender and on the receiver:

$$o + L + \left(\left\lceil \frac{m}{w} \right\rceil - 1\right) \cdot \max(o, g)$$

LogP

The LogP model [CKP+96] is defined by 4 parameters:

- *L* is the network latency (time to communicate a packet of size w)
- *o* is the middleware overhead (message splitting and packing, buffer management, connection, . . .) for a packet of size *w*
- *g* is the gap (the minimum) between two packets of size *w*
- *P* is the number of processors



Sending a one packet message: 2o+L
Sending a size m message: 2o+L+ (ceiling(m/w) – 1). max (o,g)
Occupation of sender or receiver: o+ L +(ceiling(m/w) – 1). max (o,g)

LogGP & pLogP

The previous model works fine for short messages. However, many par- allel machines have special support for long messages, hence a higher bandwidth. LogGP [AISS97] is an extension of LogP:

short messages $2o + L + \frac{m}{w} \cdot \max(o, g)$

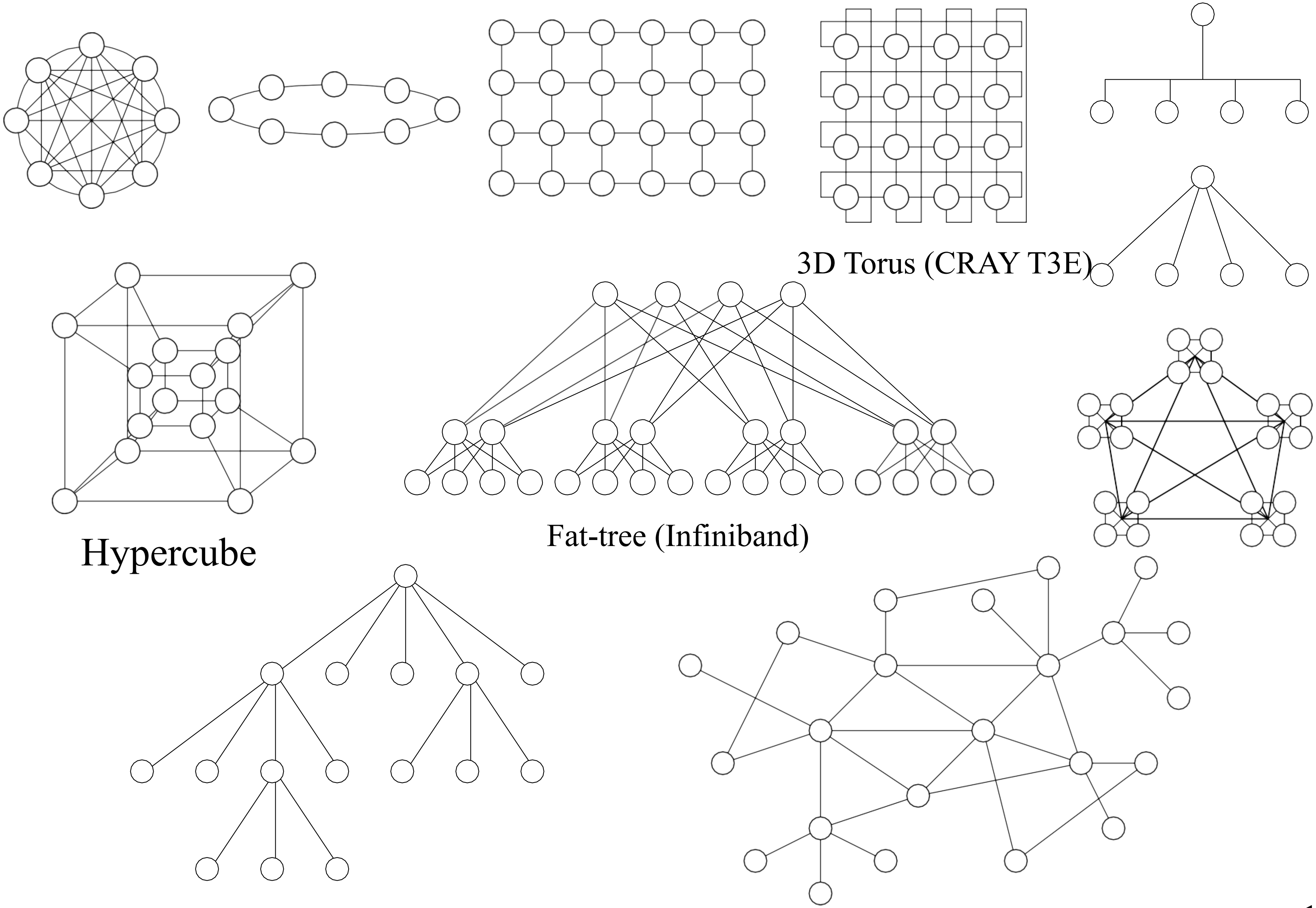
long messages $2o + L + (m - 1)G$

There is no fundamental difference. . .

OK, it works for small and large messages. Does it work for average- size messages ? pLogP [KBV00] is an extension of LogP when L, o and g depends on the message size m. They distinguish o_s and o_r.

This is more and more precise but still don't account for concurrency

Some Network Topologies



MPI for Scalable Computing

Topology Mapping

Bill Gropp, University of Illinoisat Urbana-Champaign

Rusty Lusk, Argonne National Laboratory

Rajeev Thakur, Argonne National Laboratory

Slides from the Argonne Training Program On Extreme Scale Computing 2016:
https://extremecomputingtraining.anl.gov/agenda-2016/

Collective Communications

A collective communication: specific communication instructions for some common communication patterns.

- Write simpler code (single MPI_Bcast call rather than multiple MPI-Send/MPI_Recv)
- MPI implementations can provide efficient arrangement of these communication patterns.

MPI collectives are:

MPI_Bcast() – Broadcast (one to all)
MPI_Reduce() – Reduction (all to one)
MPI_Allreduce() – Reduction (all to all)
MPI_Scatter() – Distribute data (one to all)
MPI_Gather() – Collect data (all to one)
MPI_Alltoall() – Distribute data (all to all)
MPI_Allgather() – Collect data (all to all)

Collective Communication: Broadcast

Broadcast : source process send a message M to all other process of the communicator

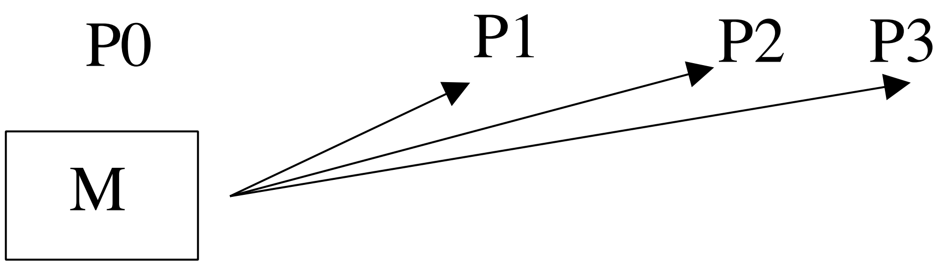
```
MPI_Bcast(&mess,3,MPI_INT,0,MPI_COMM_WORLD)
```

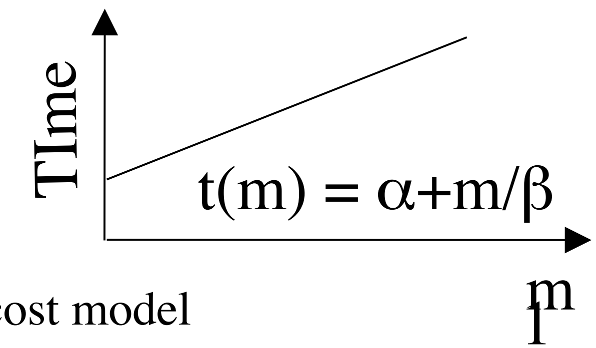
Buffer where to read or write the message to broadcast (depend on the rank)

Message source

Communicator: set of processes that receive the message (minus the source that already has it).

! All processes of the communicator must call this instruction (deadlock otherwise)!





t(m) = (p-1)*(α+1/β*m) for a naïve implementation and simple cost model

Example (PI decimals) 1/2

```
 include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
  int done = 0, n, myrank, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double myrank, pi, h, sum, x, a;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  while (!done)  {
    if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;
```

Example (PI decimals) 2/2

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myrank + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
              pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

Communications

We basically covered MPI-1 functionalities so far, based on the message passing concept.

Ethernet networks are based on the TCP/IP protocol + the socket API

By the way, what is the difference between the socket model and the MPI model ?

Beside PC clusters, supercomputers tend to rely on dedicated high performance networks (Myrinet, Infiniband) relying on specific hardware and protocols. Their goal is performance:

- Short distance cables ensure more reliable communications (no need to enforce safety like with TPC/IP)
- OS bypass - memory pinning – interrupt versus pooling (improve latency)
- Zero copy protocols (improve bandwidth)
- Direct Memory Access (DMA): no need to “bother” the remote CPU.
- Support for some collective operations (barrier, broadcast)

Argonne NATIONAL LABORATORY

MPI for Scalable Computing

One-Sided Communication

Bill Gropp, University of Illinoisat Urbana-Champaign

Rusty Lusk, Argonne National Laboratory

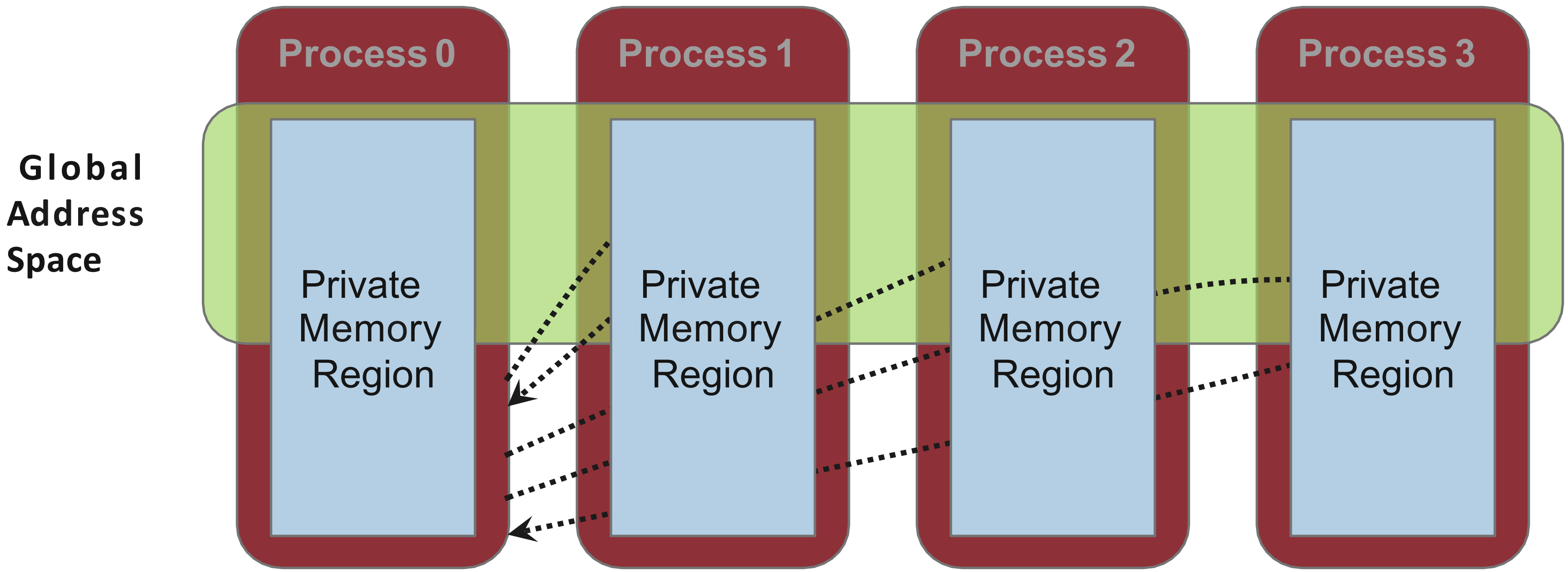
Rajeev Thakur, Argonne National Laboratory

Slides from the Argonne Training Program On Extreme Scale Computing 2016:
https://extremecomputingtraining.anl.gov/agenda-2016/

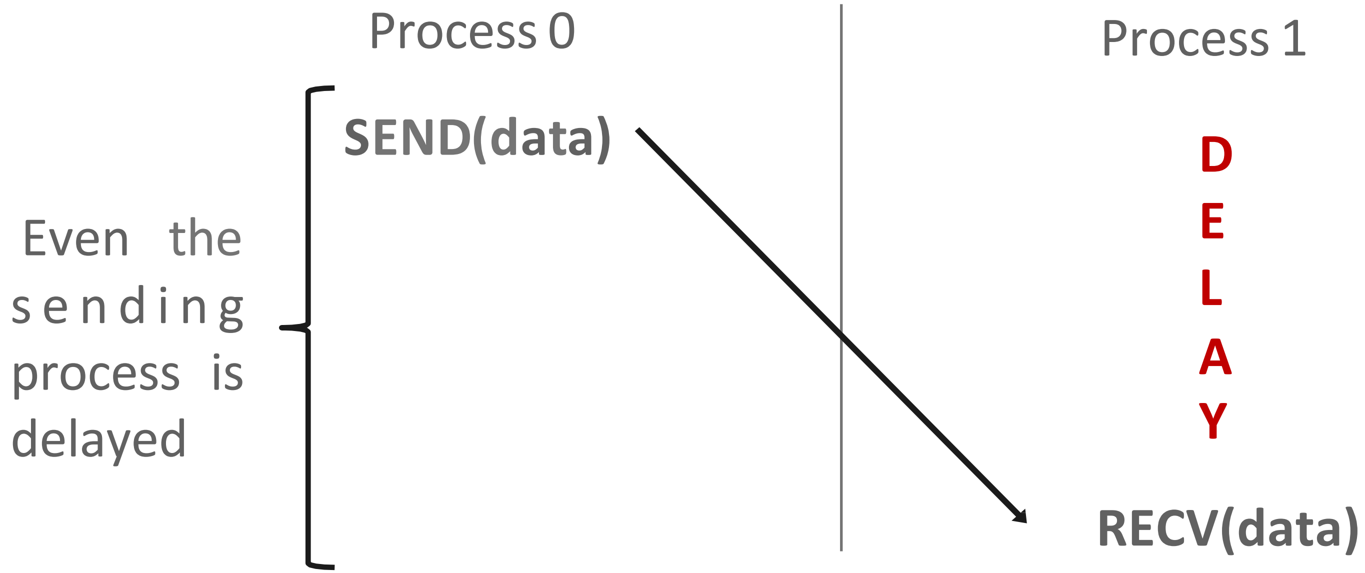
U.S. DEPARTMENT OF ENERGY

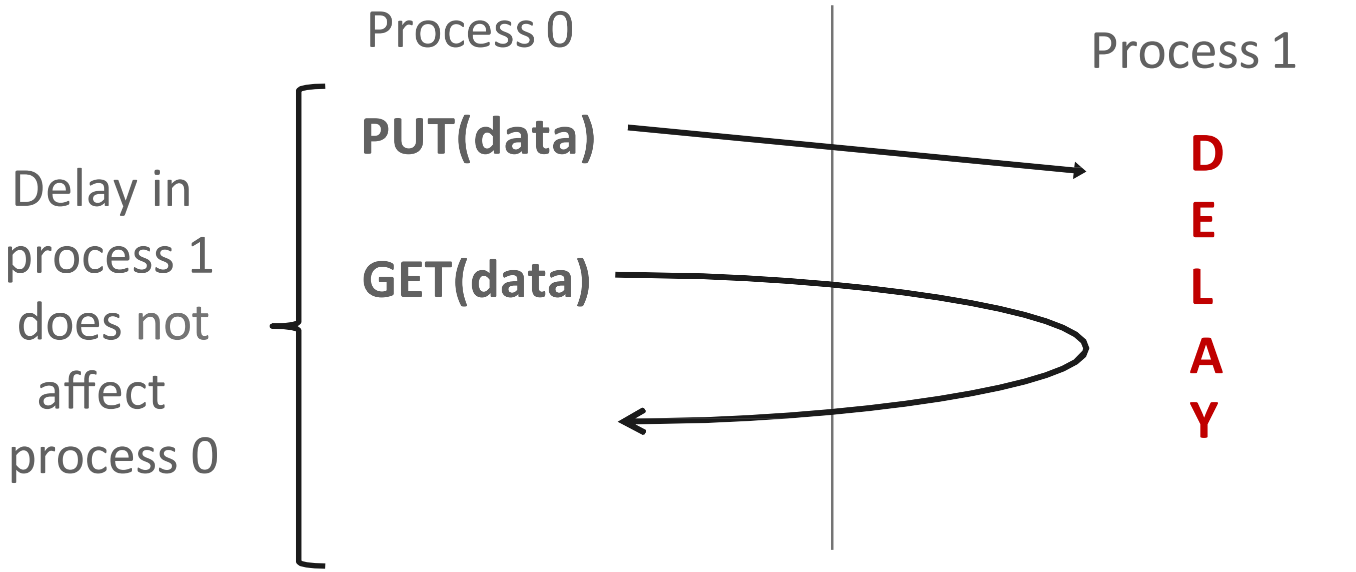
One-Sided Communication

- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory



Comparing One-sided and Two-sided Programming





Advantages of RMA Operations

- Can do multiple data transfers with a single synchronization operation
  - like BSP model
- Bypass tag matching
  - effectively precomputed as part of remote offset
- Some irregular communication patterns can be more economically expressed
- Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems

Irregular Communication Patterns with RMA

- If communication pattern is not known *a priori*, the send-recv model requires an extra step to determine how many sends-recvs to issue
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call
- This makes dynamic communication easier to code in RMA

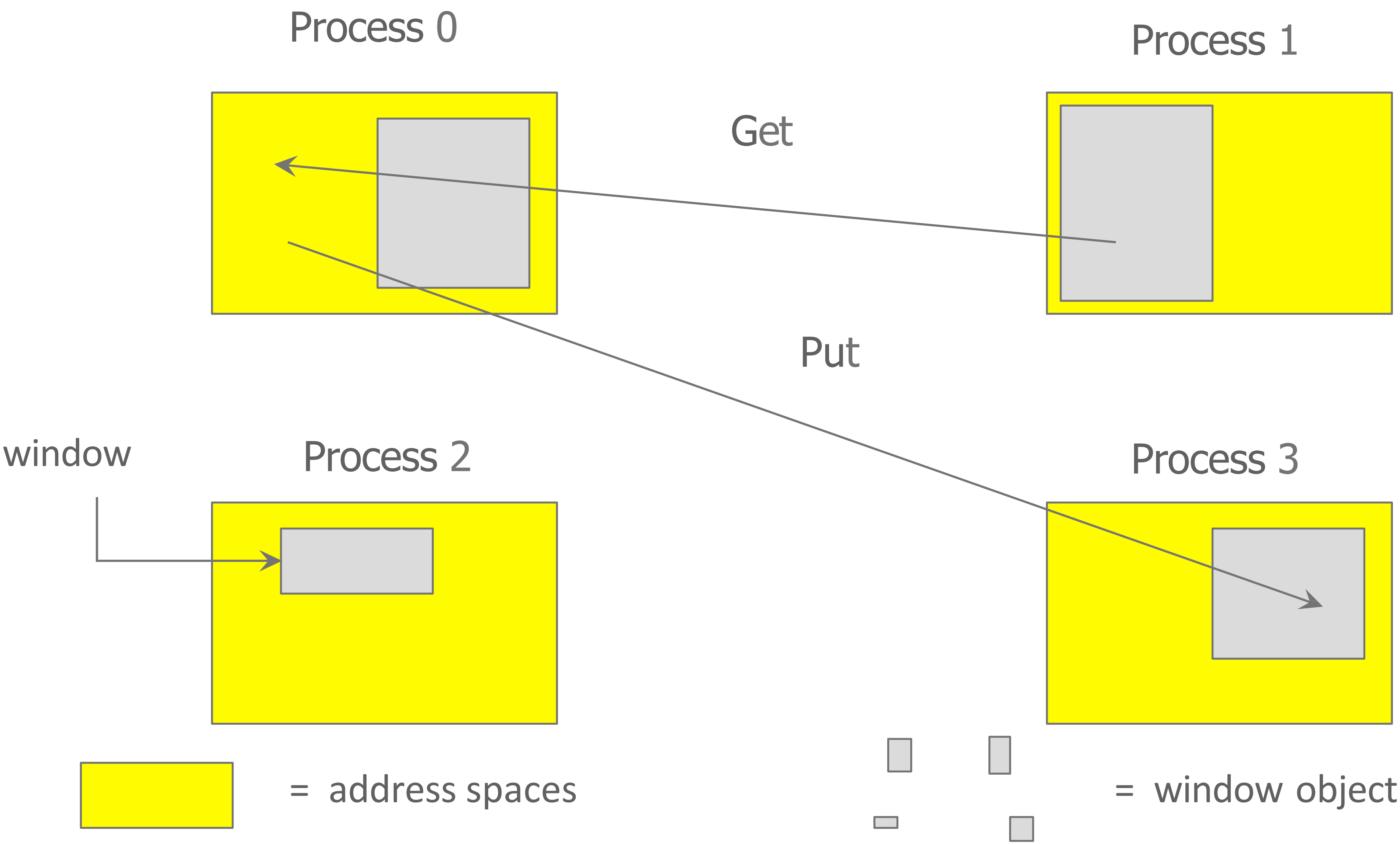
What we need to know in MPI RMA

- How to create remote accessible memory?
- Reading, Writing and Updating remote memory
- Data Synchronization
- Memory Model

Creating Public Memory

- Any memory created by a process is, by default, only locally accessible
  - X = malloc(100);
- Once the memory is created, the user has to make an explicit MPI call to declare a memory region as remotely accessible
  - MPI terminology for remotely accessible memory is a “window”
  - A group of processes collectively create a “window”
- Once a memory region is declared as remotely accessible, all processes in the window can read/write data to this memory without explicitly synchronizing with the target process

Remote Memory Access Windows and Window Objects



Basic RMA Functions for Communication

- `MPI_Win_create` exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- `MPI_Win_free` deallocates window object

- `MPI_Put` moves data from local memory to remote memory
- `MPI_Get` retrieves data from remote memory into local memory
- `MPI_Accumulate` updates remote memory using local values
- Data movement operations are non-blocking
- **Subsequent synchronization on window object needed to ensure operation is complete**

Window creation models

- Four models exist
  - MPI_WIN_CREATE
    - You already have an allocated buffer that you would like to make remotely accessible
  - MPI_WIN_ALLOCATE
    - You want to create a buffer and directly make it remotely accessible
  - MPI_WIN_CREATE_DYNAMIC
    - You don’t have a buffer yet, but will have one in the future
  - MPI_WIN_ALLOCATE_SHARED
    - You want multiple processes on the same node share a buffer
    - We will not cover this model today

MPI_WIN_CREATE

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

- Expose a region of memory in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.
- Arguments:
  - base - pointer to local data to expose
  - size - size of local data in bytes (nonnegative integer)
  - disp_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)

Example with MPI_WIN_CREATE

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory */
    a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                      MPI_COMM_WORLD, &win);

    /* Array 'a' is now accessibly by all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

MPI_WIN_ALLOCATE

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

Allocate a remotely accessible memory region in an RMA window

– Oy data exposed in a window can be accessed with RMA ops.

Arguments:

- size - size of local data in bytes (nonnegative integer)
- disp_unit - local unit size for displacements, in bytes (positive integer)
- info - info argument (handle)
- comm - communicator (handle)
- baseptr - pointer to exposed local data

Example with MPI_WIN_ALLOCATE

```
 int main(int argc, char ** argv)
{
     int *a;    MPI_Win win;

     MPI_Init(&argc, &argv);

    /* collectively create remotely accessible memory   in the
   window */
    MPI_Win_allocate(1000*sizeof(int), sizeof(int),
   MPI_INFO_NULL,
                         MPI_COMM_WORLD, &a, &win);

    /* Array ‘a’ is now accessibly by all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

MPI_WIN_CREATE_DYNAMIC

```
int MPI_Win_create_dynamic(…, MPI_Comm comm, MPI_Win *win)
```

- Create an RMA window, to which data can later be attached
  - Only data exposed in a window can be accessed with RMA ops
- Application can dynamically attach memory to this window
- Application can access data on this window only after a memory region has been attached

Example with

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);  MPI_Win_create_dynamic(MPI_INFO_NULL,
    MPI_COMM_WORLD, &win);

    /* create private memory */
    a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* locally declare memory as remotely accessible */
    MPI_Win_attach(win, a, 1000*sizeof(int));

    /*Array ‘a’ is now accessibly by all processes in MPI_COMM_WORLD*/

    /* undeclare public memory */
   MPI_Win_detach(win, a);
   MPI_Win_free(&win);

   MPI_Finalize(); return 0;
}
```

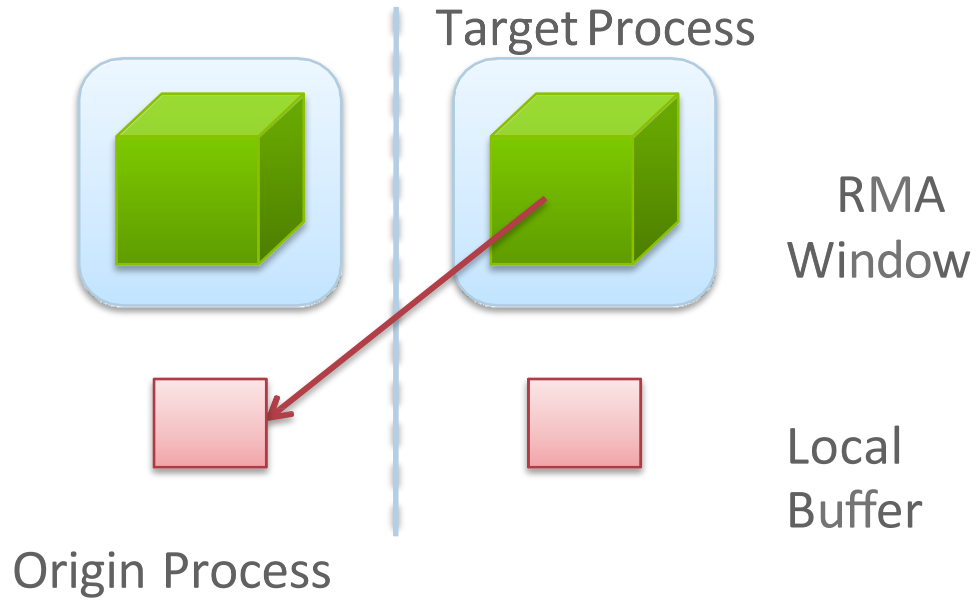
Data movement

- MPI provides ability to read, write and atomically modify data in remotely accessible memory regions
  - MPI_GET
  - MPI_PUT
  - MPI_ACCUMULATE
  - MPI_GET_ACCUMULATE
  - MPI_COMPARE_AND_SWAP
  - MPI_FETCH_AND_OP

Data movement: *Get*

```
MPI_Get(origin_addr, origin_count, origin_datatype,
        target_rank, target_disp, target_count,
target_datatype,
        win)
```

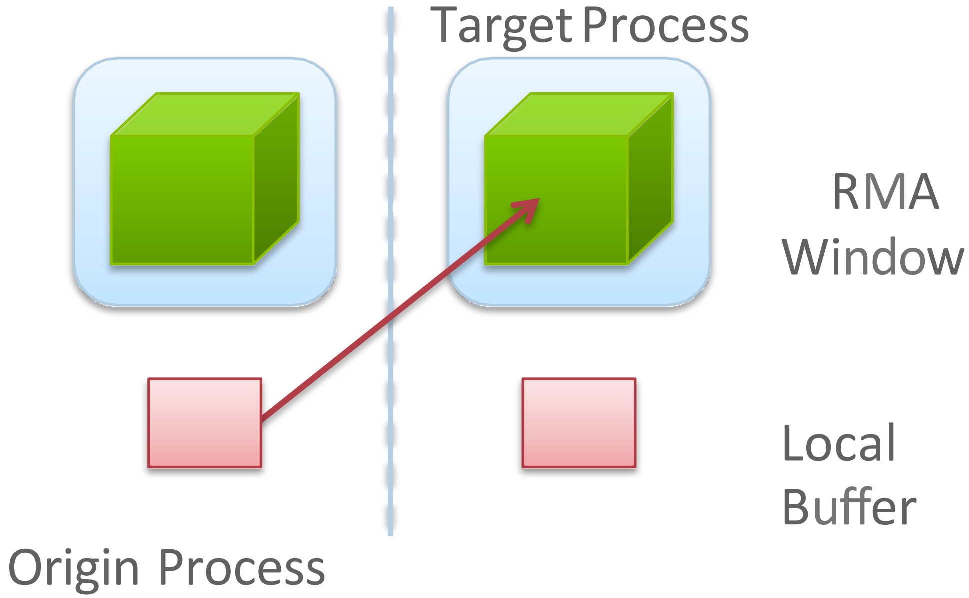
- Move data to origin, from target
- Separate data description triples for origin and target



Data movement: *Put*

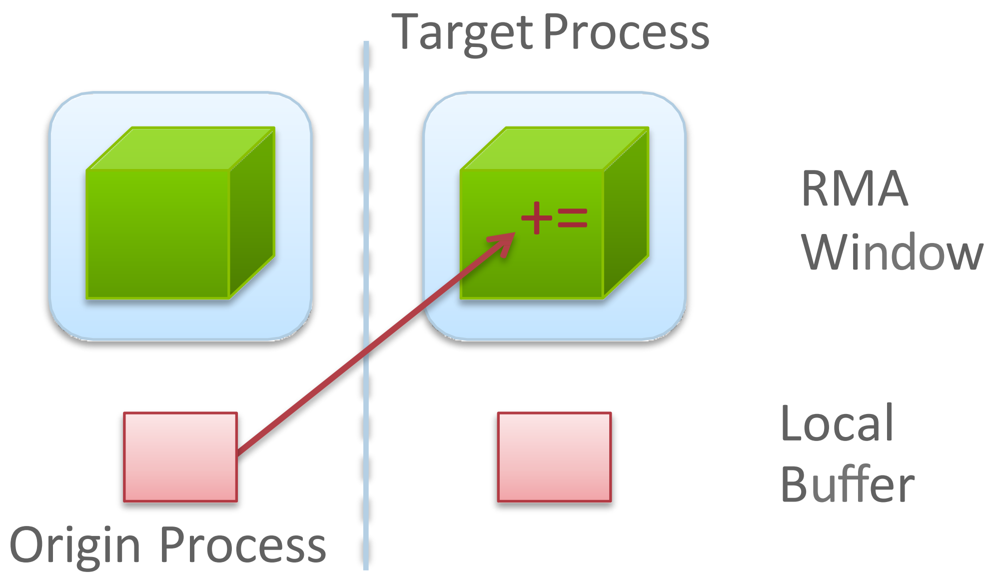
```
MPI_Put(origin_addr, origin_count, origin_datatype,
        target_rank, target_disp, target_count,
target_datatype,
        win)
```

- Move data from origin, to target
- Same arguments as MPI_Get



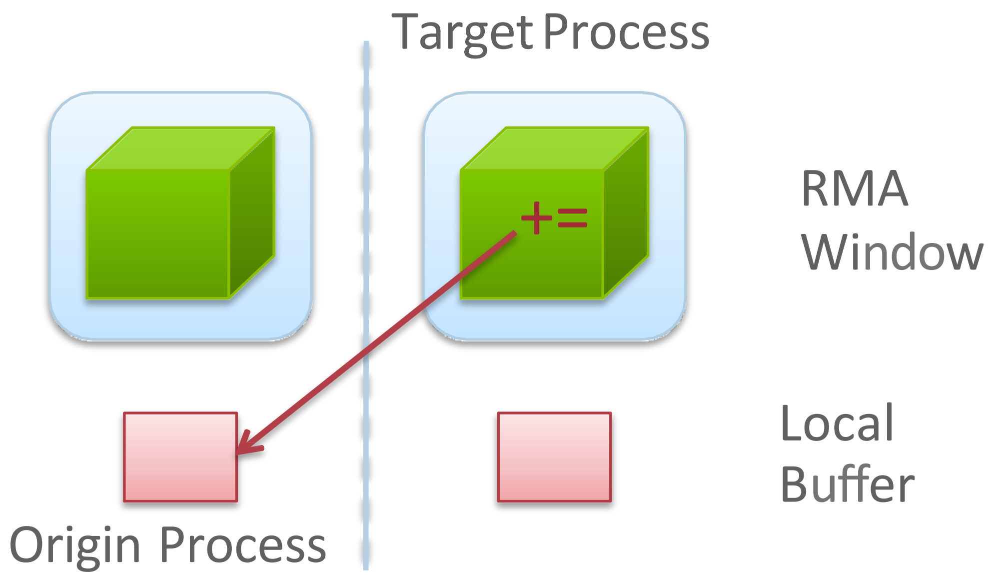
Data aggregation: *Accumulate*

- Like MPI_Put, but applies an MPI_Op instead
  - Predefined ops only, no user-defined!
- Result ends up at target buffer
- Different data layouts between target/origin OK, basic type elements must match
- Put-like behavior with MPI_REPLACE (implements *f(a,b)=b*)
  - Per element atomic PUT



Data aggregation: *Get Accumulate*

- Like MPI_Get, but applies an MPI_Op instead
  - Predefined ops only, no user-defined!
- Result at target buffer; original data comes to the source
- Different data layouts between target/origin OK, basic type elements must match
- Get-like behavior with MPI_NO_OP
  - Per element atomic GET



Ordering of Operations in MPI RMA

- For Put/Get operations, ordering does not matter
  - If you do two concurrent PUTs to the same location, the result can be garbage
- Two accumulate operations to the same location are valid
  - If you want “atomic PUTs”, you can do accumulates with MPI_REPLACE
- All accumulate operations are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hints
  - You can ask for “read-after-write” ordering, “write-after-write” ordering, or “read-after-read” ordering

Additional Atomic Operations

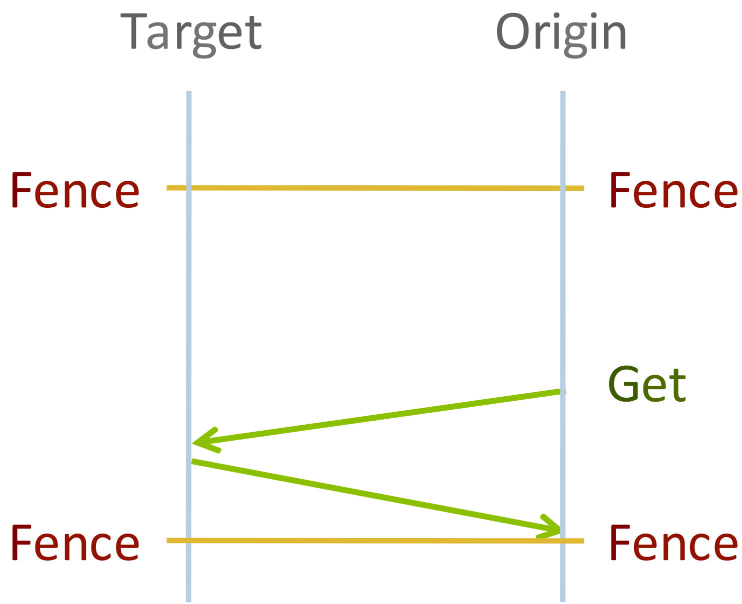
- Compare-and-swap
  - Compare the target value with an input value; if they are the same, replace the target with some other value
  - Useful for linked list creations – if next pointer is NULL, do something
- Fetch-and-Op
  - Special case of Get accumulate for predefined datatypes – (probably) faster for the hardware to implement

RMA Synchronization Models

- RMA data visibility
  - When is a process allowed to read/write from remotely accessible memory?
  - How do I know when data written by process X is available for process Y to read?
  - RMA synchronization models provide these capabilities
- MPI RMA model allows data to be accessed only within an "epoch"
  - Three types of epochs possible:
    - Fence (active target)
    - Post-start-complete-wait (active target)
    - Lock/Unlock (passive target)
- Data visibility is managed using RMA synchronization primitives
  - MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL
  - Epochs also perform synchronization

Fence Synchronization

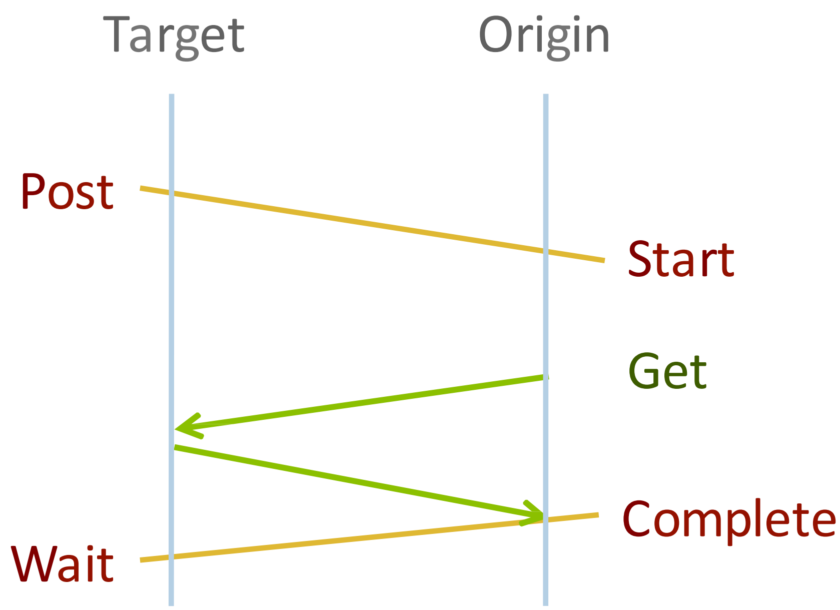
- `MPI_Win_fence(assert, win)`
- Collective synchronization model -- assume it synchronizes like a barrier
- Starts *and* ends access & exposure epochs (usually)



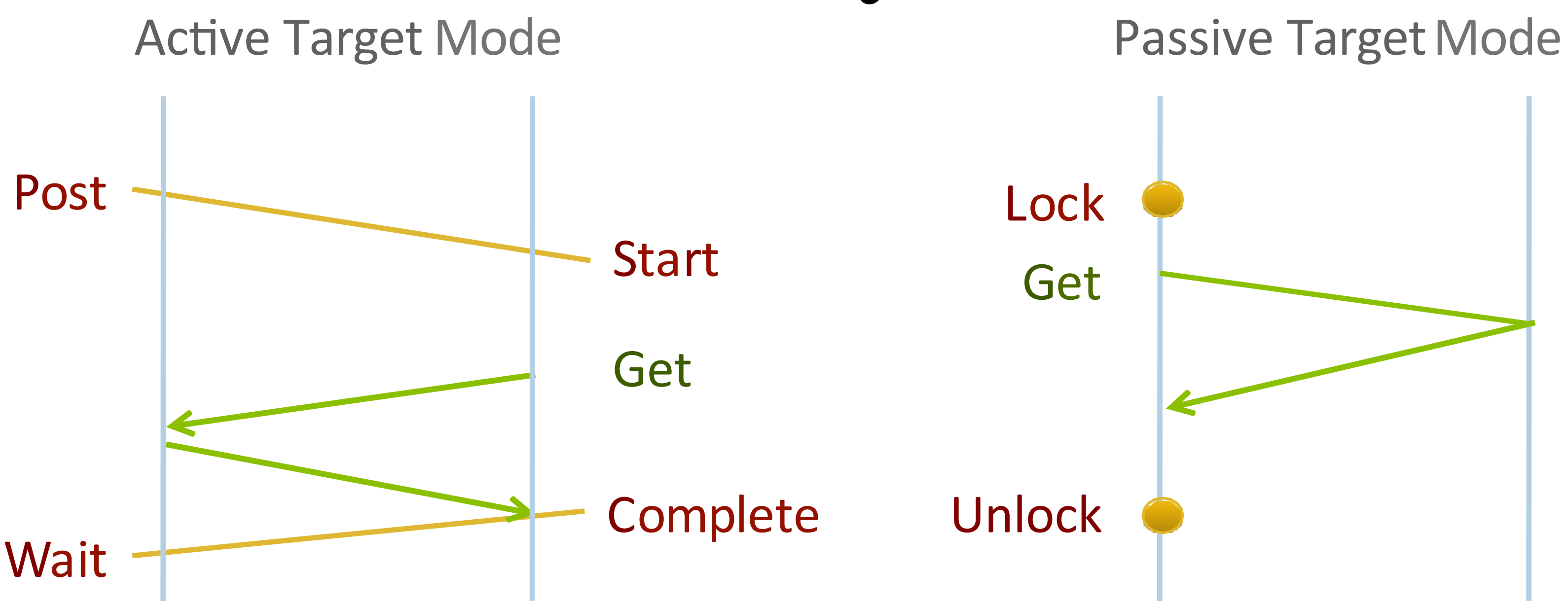
- Everyone does an MPI_WIN_FENCE to open an epoch
- Everyone issues PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch

PSCW (Post-Start-Complete-Wait) Synchronization

- Target: Exposure epoch
  - Opened with MPI_Win_post
  - Closed by MPI_Win_wait
- Origin: Access epoch
  - Opened by MPI_Win_start
  - Closed by MPI_Win_compete
- All may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets
- Like FENCE, but the target may allow a smaller group of processes to access its data



Lock/Unlock Synchronization



- Passive mode: One-sided, *asynchronous* communication
  - Target does **not** participate in communication operation
- Shared memory like model

Passive Target Synchronization

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

int MPI_Win_unlock(int rank, MPI_Win win)
```

- Begin/end passive mode epoch
  - Doesn't function like a mutex, name can be confusing
  - Communication operations within epoch are all nonblocking
- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently

When should I use passive mode?

- RMA performance advantages from low protocol overheads
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc...
  - Direct support from high-speed interconnects (e.g. InfiniBand)
- Passive mode: *asynchronous* one-sided communication
  - Data characteristics:
    - Big data analysis requiring memory aggregation
    - Asynchronous data exchange
    - Data-dependent access pattern
  - Computation characteristics:
    - Adaptive methods (e.g. AMR, MADNESS)
    - Asynchronous dynamic load balancing
- Common structure: shared arrays

Dynamicity in MPI(-2)

MPI_COMM_SPAWN

Each process from MPI_COMM_WORLD spawn one new process that each starts the "child" exec.

```
int rank, err[4];
MPI_Comm children;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_spawn("child", NULL, 4, MPI_INFO_NULL, 0,
               MPI_COMM_WORLD, &children, err);
if (0 == rank) {
    MPI_Send(&rank, 1, MPI_INT, 0, 0, children);
}
```

Child program

```
#include "mpi.h"
int main(int argc, &argv) {
    int rank, msg;
    MPI_Comm parent;
    MPI_Init(&argc, &argv);
    MPI_Comm_get_parent(&parent);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (0 == rank) {
        MPI_Recv(&msg, 1, MPI_INT, 0, 0, parent, MPI_STATUS_IGNORE);
    }
```

Rank 0 of initial processes sends a message to rank 0 of the spawned processes

Dynamicity in MPI(-2)

Client/server mode for MPI

Server side:

```
char myport[MPI_MAX_PORT_NAME];
 MPI_Comm intercomm;
 /* ... */
 MPI_Open_port(MPI_INFO_NULL, myport);
 printf("port name is: %s\n", myport);

 MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
 /* do something with intercomm */
```

On the client side:

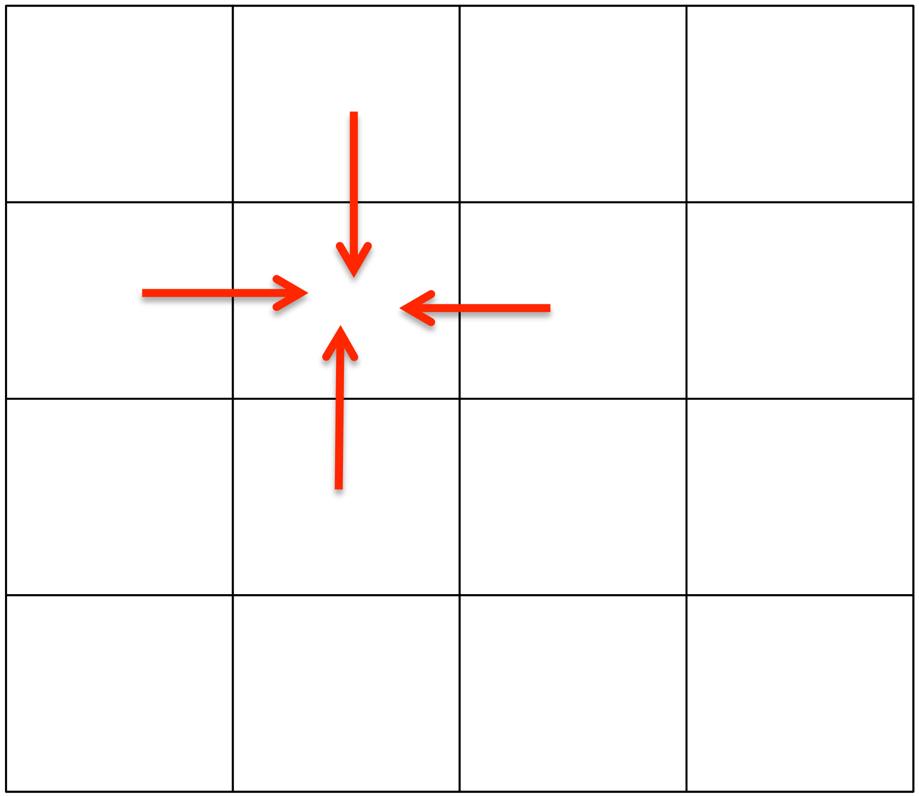
```
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF,&intercomm);
```

Programming with MPI

A simple example, but representative of many numerical simulations (FEM, DEM)

The simulation domain is split in cells.

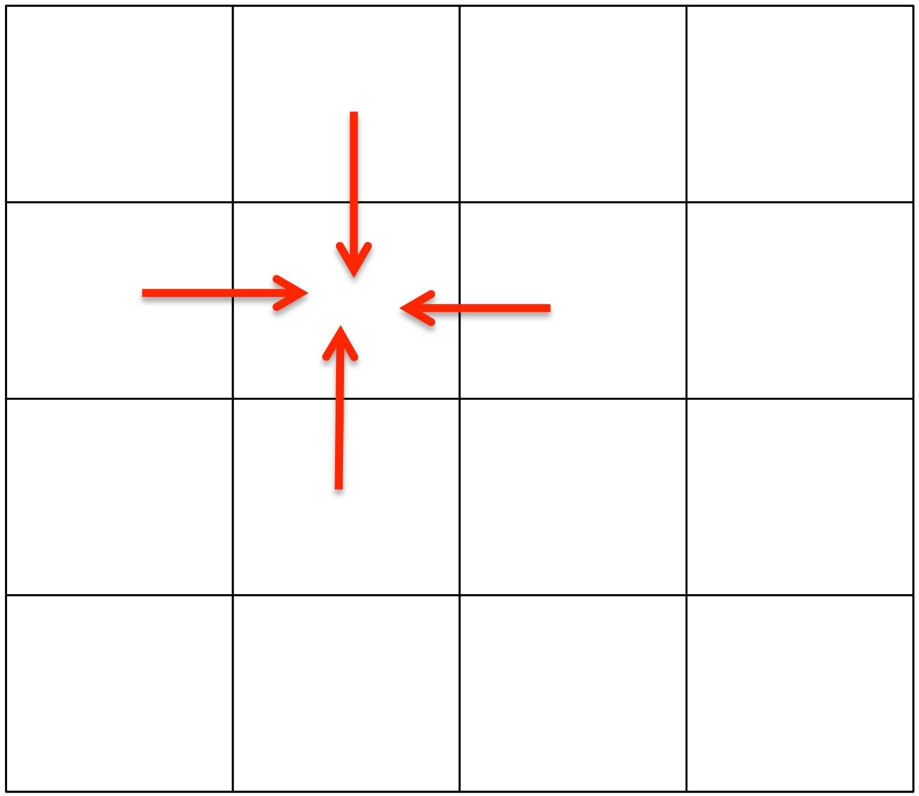
At each iteration t_i, each cell state is updated according to the states of its neighbours at t_{i-1}.



What about the MPI parallelization ?
(We have much more cells than processors)

Programming with MPI

- 1D or 2D domain partitioning ?
- Communication and data structure organisation (phantom cells) ?



Send/Receive Considered Harmful ?

After Dijkstra paper: “Go To Statement Considered Harmful”, 1968

What about the:

- Send/recv constructs as the MPI ones ?
- Collectives operations ?

Qualities and Limits of the MPI model

The MPI model assumes the machine is homogeneous:

- one processor per MPI process each available at 100%
- communication time is about the same between any pair of processors

Pro: This model makes it "easy" to reason about the parallelization of a program.

MPI is based on a distributed memory model with explicit communications

Pro: The programmer is forced to think globally parallel and not just parallelize some parts (case with OpenMP)

But today processors are multi-core, nodes multi-sockets!

Qualities and Limits of the MPI model

The MPI model assumes the machine is homogeneous:

- one processor per MPI process each available at 100%
- communication time is about the same between any pair of processors

Cons: Perturbations that can prevent processes to progress evenly can have a catastrophic effect on performance

- Avoid more than one process per core
- Disable Hyperthreading
- Process pining to avoid OS perturbations
- If you want dynamics load balancing you have to program it by yourself.

Qualities and Limits of the MPI model

MPI is based on a distributed memory model with explicit communications:

Cons: Communication performance is very different for intra-node or inter-node communications

Cons: Data are duplicated (phantom cells) even for processes running on the same node (shared memory)

How to take benefit of multi-cores: **hybrid programming ?**

Hybrid Programming

Hybrid programming = MPI + X (X=openMP, Intel TBB, Cilk, Pthreads)

MPI supports (depend on the implementation) two modes:

MPI_THREAD_FUNNELED: only one thread (master thread) can make MPI calls

MPI_THREAD_MULTIPLE: all threads can make MPI calls. This is the more flexible mode, but current implementation often show important performance issues.

By the way, what does thread-safe mean ?