AMS 250: An Introduction to High Performance Computing

Advanced MPI

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Astronomy & Astrophysics

University of California, Santa Cruz

Outline

- MPI + OpenMP
- One-Sided Communications with MPI

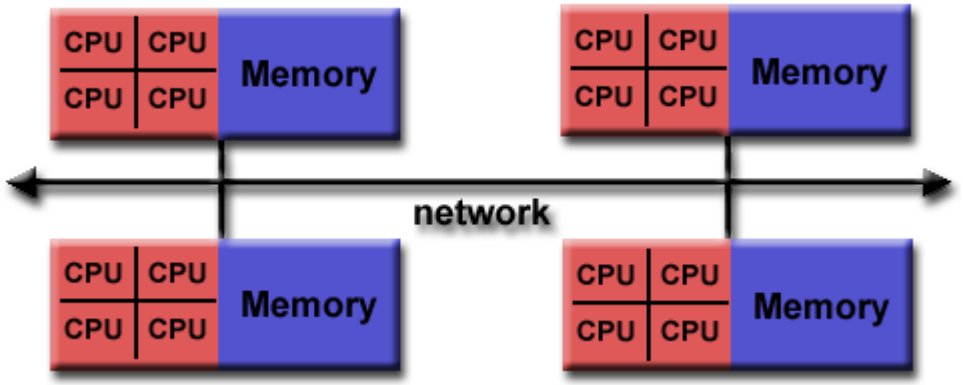
MPI + X

- For the past 20+ years, HPC programing is dominated by
  - either a flat model, with **MPI** across nodes as well as cores within a node
  - or a hybrid model, with **MPI** across the nodes and **OpenMP** shared memory parallelism across the cores in a node
- The promise of **PGAS** (Partitioned Global Address Space) has failed to materialize – more on PGAS later
- MPI continues to be enhanced and evolved – MPI 3.1 released in June 2015
- In the exascale era, the programming model will likely be MPI + X, where X is one of:
  - OpenMP 3 (if accelerators don't persist)
  - OpenMP 4 with *target* directives
  - OpenACC
  - CUDA or OpenCL
  - PGAS

http://www.hpcwire.com/2014/07/16/compilers-mpix/

MPI + OpenMP

- Modern HPC systems are predominantly clusters of SMPs, and increasing heterogeneous:



- Flat MPI model:
  - ☺ simpler programming model: only one level of parallelism and only one API
  - ☹ doesn't take advantage of the shared data across the ranks on the same node, requiring message and buffer management across all ranks
- Hybrid MPI + OpenMP model:
  - ☹ more complex programming model
  - ☺ maps nicely to today's prevalent architecture

MPI and Threads

- MPI libraries can vary in their level of thread support:
  - `MPI_THREAD_SINGLE` – Level 0: Only one thread will execute.
  - `MPI_THREAD_FUNNELED` – Level 1: The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread).
  - `MPI_THREAD_SERIALIZED` – Level 2: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time. That is, calls are not made concurrently from two distinct threads as all MPI calls are serialized.
  - `MPI_THREAD_MULTIPLE` – Level 3: Multiple threads may make MPI calls with no restrictions.
- OpenMP and Pthreads are common models for thread parallelism

MPI and Threads (cont'd)

- An implementation is not required to support levels higher than **MPI_THREAD_SINGLE**; that is, an implementation is not required to be thread safe
- A fully thread-compliant implementation will support **MPI_THREAD_MULTIPLE**
- Call **MPI_Init_thread** (instead of **MPI_Init**) to initialize the multi-threaded MPI execution environment

C	`int MPI_Init_thread(int *argc, char ***argv, int required,` `  int *provided)`
Fortran	`MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)` `  INTEGER  REQUIRED, PROVIDED, IERROR`

- A portable program that does not call **MPI_Init_thread** should assume that only **MPI_THREAD_SINGLE** is supported

An example to query the level of thread support

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

int main(int argc, char* argv[]) {
    int rank;
    int thread_level, thread_is_main;
    /* MPI_Init(&argc, &argv) */
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &thread_level);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        printf("MPI_THREAD_SINGLE = %d\n", MPI_THREAD_SINGLE);
        printf("MPI_THREAD_FUNNELED = %d\n", MPI_THREAD_FUNNELED);
        printf("MPI_THREAD_SERIALIZED = %d\n", MPI_THREAD_SERIALIZED);
        printf("MPI_THREAD_MULTIPLE = %d\n", MPI_THREAD_MULTIPLE);
        printf("I asked for level 3, and I got level %d\n", thread_level);
        MPI_Query_thread( &thread_level );
        MPI_Is_thread_main( &thread_is_main );
        printf("Thread level is %d\n", thread_level);
        printf("thread_is_main = %d\n", thread_is_main);
   }
   MPI_Finalize();
   return 0;
}
```

Try it out on Cori

```
shawdong@cori06:> cc mpi_thread.c -o mpi_thread.x
shawdong@cori06:> salloc -N 1 -p debug -L SCRATCH -C haswell

shawdong@nid12954:> srun -n 2 ./mpi_thread.x
MPI_THREAD_SINGLE = 0
MPI_THREAD_FUNNELED = 1
MPI_THREAD_SERIALIZED = 2
MPI_THREAD_MULTIPLE = 3
I asked for level 3, and I got level 2
Thread level is 2
thread_is_main = 1
```

Cray MPI only supports up to MPI_THREAD_SERIALIZED (level 2)!

Try it out on Hyades (Intel MPI)

```
$ mpiicc -mt_mpi mpi_thread.c -o mpi_thread.x
$ mpirun -n 2 ./mpi_thread.x
MPI_THREAD_SINGLE = 0
MPI_THREAD_FUNNELED = 1
MPI_THREAD_SERIALIZED = 2
MPI_THREAD_MULTIPLE = 3
I asked for level 3, and I got level 3
Thread level is 3
thread_is_main = 1
```

Note:

- You must link with the thread safe version of the Intel MPI library (-mt_mpi)
- Intel MPI offers all 3 levels of thread support

MPI_THREAD_MULTIPLE

- Multiple threads may make MPI calls with no restrictions.
- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
- User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
- “Thread-safe” usually means MPI_THREAD_MULTIPLE

MPI_THREAD_SERIALIZED

Multiple threads may make MPI calls, but only one at a time.

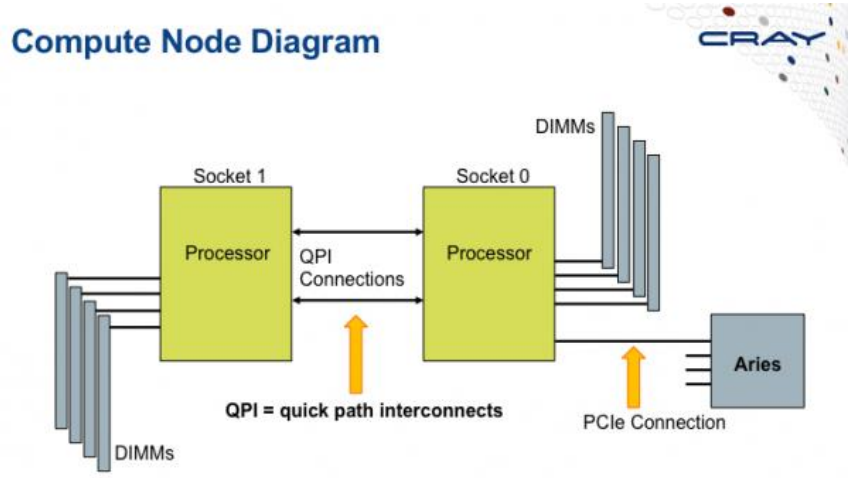
```
#pragma omp parallel
...
#pragma omp atomic
{
... MPI calls allowed here ...
}
```

MPI_THREAD_FUNNELED

- All of the MPI calls are made by the master thread. That is, all MPI calls are:
  - either outside OpenMP parallel regions
  - or inside OpenMP master regions
  - or guarded by an **MPI_Is_thread_main** call (same thread that's called **MPI_Init_thread**)
- Probably sufficient for typical MPI + OpenMP hybrid programs

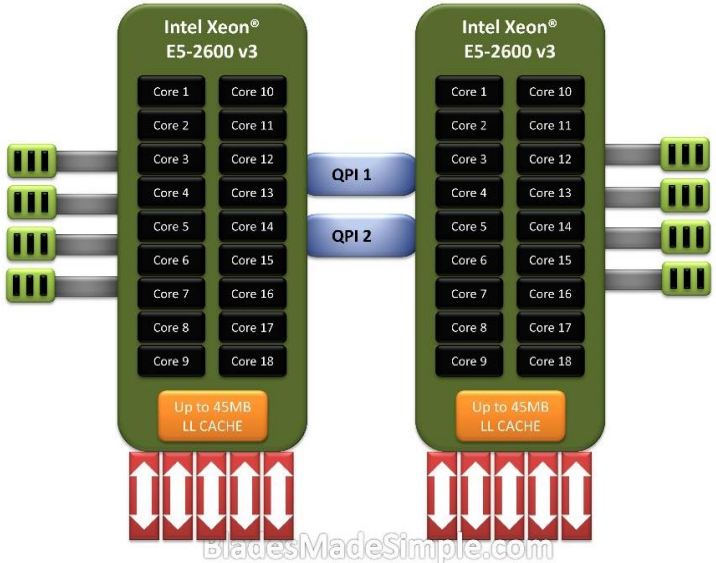
Edison Compute Nodes

- Two Intel "Ivy Bridge" Xeon E5-2695 v2 processors per node
  - 12 cores per processor
  - 256-bit/cycle ring bus interconnect between on-die cores
  - integrated memory controller and PCIe controller
  - 2 QPI links between the 2 processors
- Two NUMA nodes per node
  - one NUMA node per processor/socket
  - local memory bandwidth: 59.7 GB/s
  - remote memory bandwidth: 32.0 GB/s
- An Aries Network Interface Controller (NIC)
  - 0.25 µs to 3.7 µs MPI latency
  - ~8GB/sec MPI bandwidth



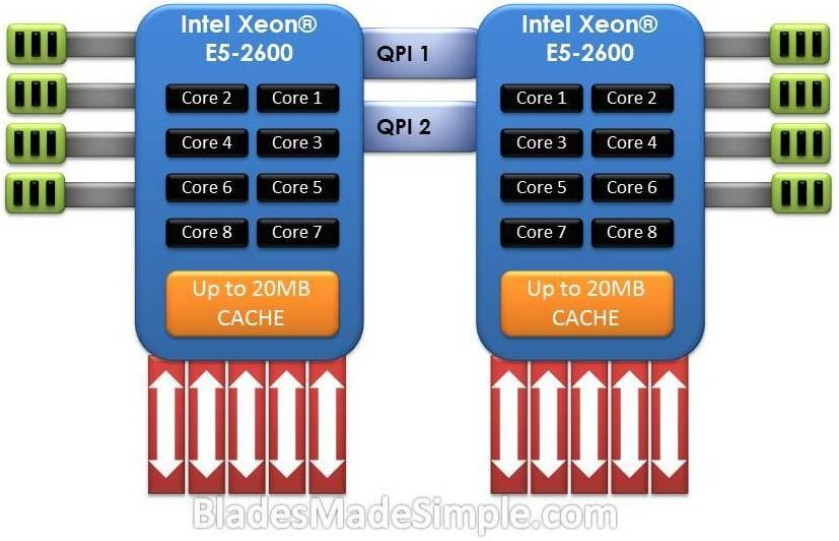
Cori Haswell Compute Nodes

- Two Intel "Haswell" Xeon E5-2698 v3 processors per node
  - 16 cores per processor
  - 256-bit/cycle ring bus interconnect between on-die cores
  - integrated memory controller and PCIe controller
  - 2 QPI links between the 2 processors
- Two NUMA nodes per node
  - one NUMA node per processor/socket
  - local memory bandwidth: 68.3 GB/s
  - remote memory bandwidth: 38.4 GB/s
- An Aries Network Interface Controller (NIC)
  - 0.25 µs to 3.7 µs MPI latency
  - ~8GB/sec MPI bandwidth

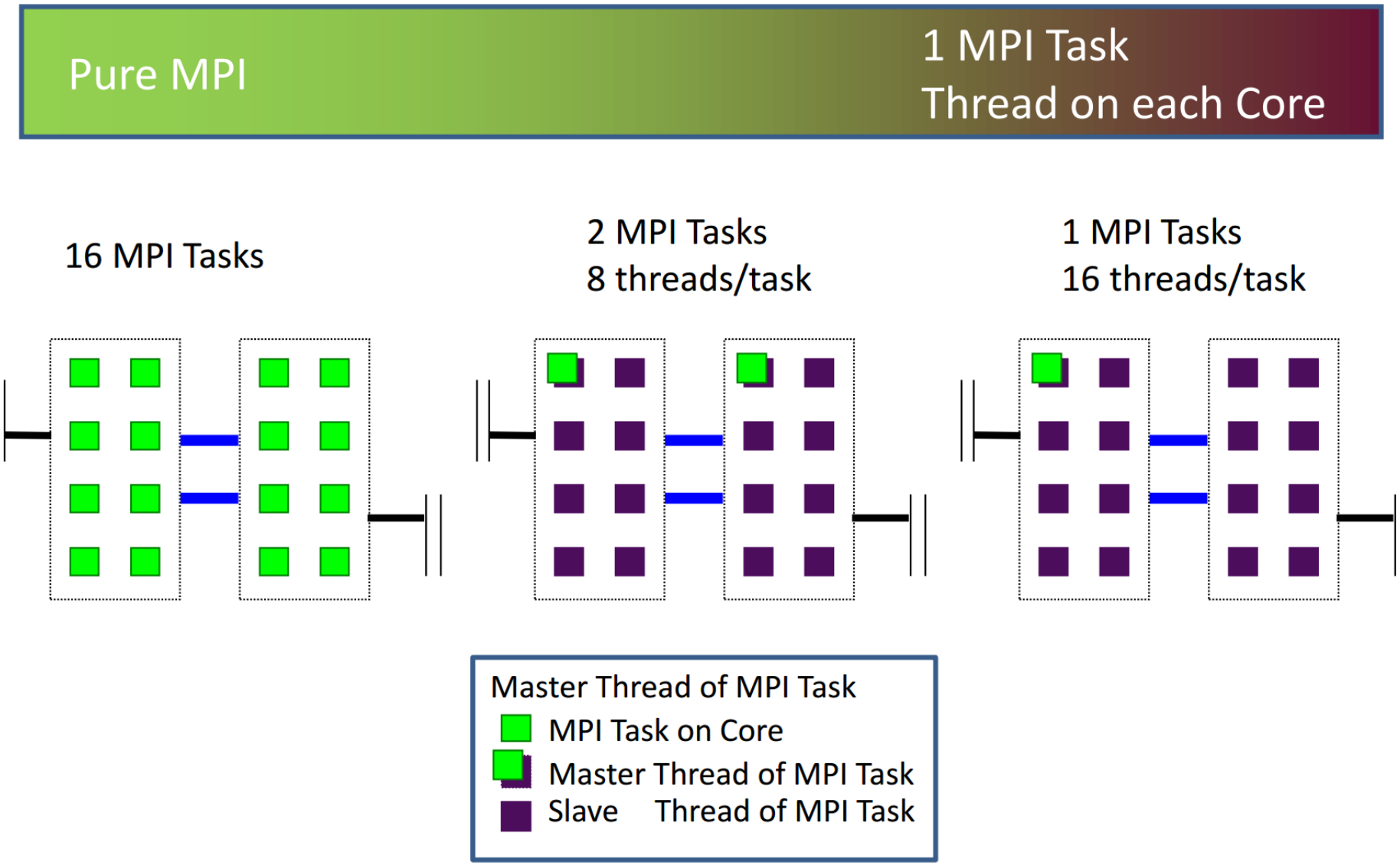


Hyades Compute Nodes

- Two Intel "Sandy Bridge" Xeon E5-2650 processors per node
  - 8 cores per processor
  - 256-bit/cycle ring bus interconnect between on-die cores
  - integrated memory controller and PCIe controller
  - 2 QPI links between the 2 processors
- Two NUMA nodes per node
  - one NUMA node per processor/socket
  - local memory bandwidth: 51.2 GB/s
  - remote memory bandwidth: 32.0 GB/s
- Mellanox ConnectX-2 VPI QDR InfiniBand HCA
  - 5 GB/s signaling rate
  - 4 GB/s data rate
  - for MPI communications across nodes



Running Modes

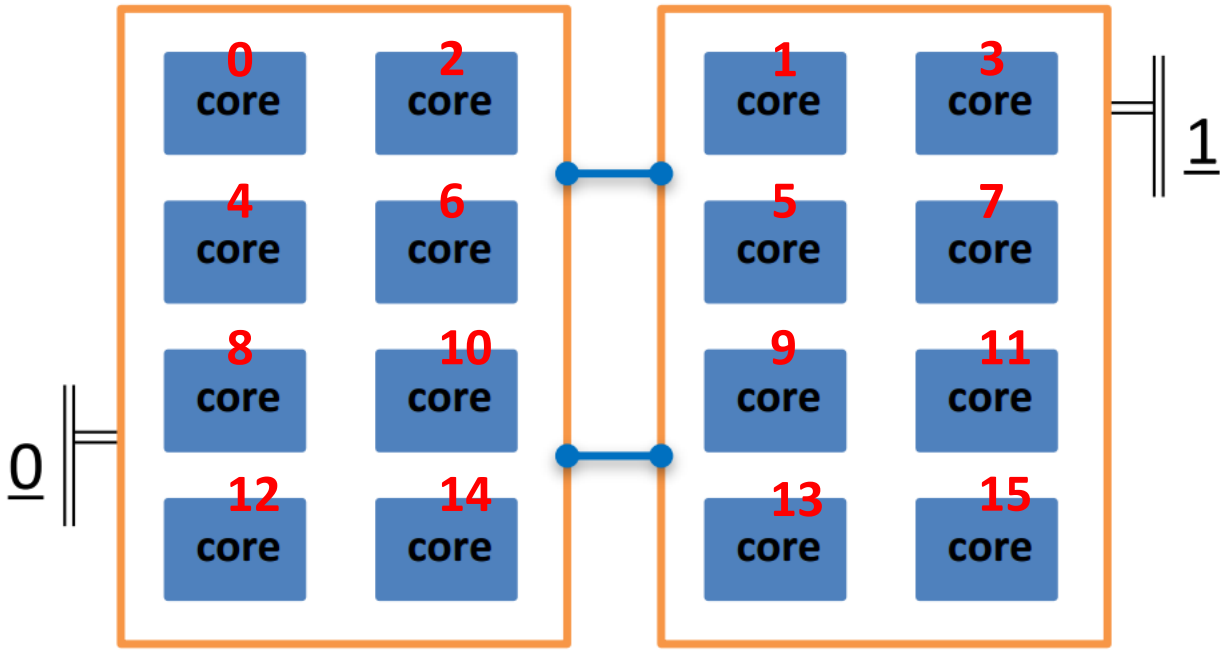


NUMA Operations

- Each process/thread is executed by a core and has access to a certain memory space
  - Core assigned by **process affinity**
  - Memory allocation assigned by **memory policy**
- The control of **process affinity** and **memory policy** using NUMA operations
  - NUMA Control is managed by the kernel (default).
  - Users can alter kernel policies by manually setting **process affinity** and **memory policy**, within a program though C/Fortran API, or using *numactl*.

Processor IDs

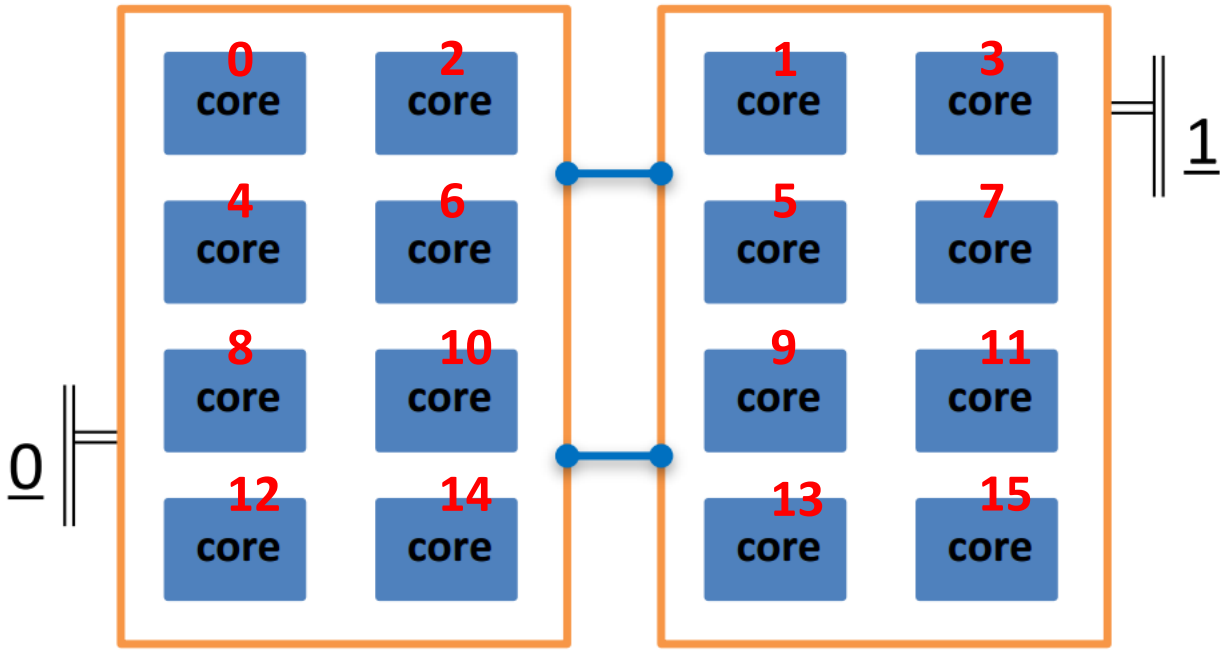
```
$ egrep 'processor|physical id|core id' /proc/cpuinfo
processor       : 0
physical id     : 0
core id         : 0
processor       : 1
physical id     : 1
core id         : 0
processor       : 2
physical id     : 0
core id         : 1
processor       : 3
physical id     : 1
core id         : 1
...
```



Hyades Compute Node

numactl

```
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14
node 0 size: 32722 MB
node 0 free: 24463 MB
node 1 cpus: 1 3 5 7 9 11 13 15
node 1 size: 32768 MB
node 1 free: 26229 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```



Hyades Compute Node

Cori Haswell Compute Node

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -C haswell

nid12954:> numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 0 size: 64430 MB
node 0 free: 62803 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 48 49 50 51 52 53 54 55 56 57 58 59 60 61
62 63
node 1 size: 64635 MB
node 1 free: 63552 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

- Hyperthreading is enabled
- 16 physical cores per CPU
- 16 additional logical cores per CPU
- 2 NUMA nodes (CPUs) per node
- 32 physical cores per node
- 32 additional logical cores per node
- Physical cores will be filled first

Cori KNL Compute Node: SNC4 + Cache

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,snc4,cache

nid12253:~> numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0-17,68-85,136-153,204-221
node 0 size: 24065 MB
node 1 cpus: 18-35,86-103,154-171,222-239
node 1 size: 24232 MB
node 2 cpus: 36-51,104-119,172-187,240-255
node 2 size: 24233 MB
node 3 cpus: 52-67,120-135,188-203,256-271
node 3 size: 24230 MB
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

Cori KNL Compute Node: SNC4 + Flat

```
cori08:> salloc -N 1 -p debug -t 00:10:00 -L SCRATCH -C knl,snc4,flat

nid04893:~> numactl -H
available: 8 nodes (0-7)
node 0 cpus: 0-17,68-85,136-153,204-221
node 0 size: 24065 MB
node 1 cpus: 18-35,86-103,154-171,222-239
node 1 size: 24232 MB
node 2 cpus: 36-51,104-119,172-187,240-255
node 2 size: 24233 MB
node 3 cpus: 52-67,120-135,188-203,256-271
node 3 size: 24233 MB
node 4 cpus:
node 4 size: 4039 MB
node 5 cpus:
node 5 size: 4039 MB
node 6 cpus:
node 6 size: 4039 MB
node 7 cpus:
node 7 size: 4037 MB
```

```
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  21  21  21  31  41  41  41
  1:  21  10  21  21  41  31  41  41
  2:  21  21  10  21  41  41  31  41
  3:  21  21  21  10  41  41  41  31
  4:  31  41  41  41  10  41  41  41
  5:  41  31  41  41  41  10  41  41
  6:  41  41  31  41  41  41  10  41
  7:  41  41  41  31  41  41  41  10
```

Core Affinity Example

```
#define _GNU_SOURCE

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sched.h>
#include <mpi.h>
#include <omp.h>
```

```
static char *cpuset_to_cstr(cpu_set_t *mask, char *str)
{
  char *ptr = str;
  int i, j, entry_made = 0;
  for (i = 0; i < CPU_SETSIZE; i++) {
    if (CPU_ISSET(i, mask)) {
      int run = 0;
      entry_made = 1;
      for (j = i + 1; j < CPU_SETSIZE; j++) {
        if (CPU_ISSET(j, mask)) run++;
        else break;
        }
        if (!run)
          sprintf(ptr, "%d,", i);
        else if (run == 1) {
          sprintf(ptr, "%d,%d,", i, i + 1); i++;
        } else {
            sprintf(ptr, "%d-%d,", i, i + run);
            i += run;
        }
        while (*ptr != 0) ptr++;
      }
    }
    ptr -= entry_made;
    *ptr = 0;
    return(str);
}
```

```
int main(int argc, char *argv[])
{
  int thread_level, rank, thread, duration=0;
  cpu_set_t coremask;
  char clbuf[7 * CPU_SETSIZE], hnbuf[64];

  if (argc == 2) duration = atoi(argv[1]);
  /* MPI_Init(&argc, &argv); */
  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &thread_level);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  memset(clbuf, 0, sizeof(clbuf));
  memset(hnbuf, 0, sizeof(hnbuf));
  gethostname(hnbuf, sizeof(hnbuf));
  #pragma omp parallel private(thread, coremask, clbuf)
  {
    thread = omp_get_thread_num();
    sched_getaffinity(0, sizeof(coremask), &coremask);
    cpuset_to_cstr(&coremask, clbuf);
    #pragma omp barrier
    printf("Hello from rank %d, thread %d, on %s. (core affinity = %s)\n",
           rank, thread, hnbuf, clbuf);
    sleep(duration);
  }
  MPI_Finalize();
  return(0);
}
```

Sample Batch Script for Hybrid Jobs on Edison

Batch script *hybrid.slurm:*

```
#!/bin/bash -l

#SBATCH -J hybrid
#SBATCH -p regular
#SBATCH -N 2
#SBATCH -t 4:00:00
#SBATCH -L SCRATCH
#SBATCH --mail-user=shaw@ucsc.edu
#SBATCH --mail-type=ALL

export OMP_NUM_THREADS=12
srun -n 4 -c 24 ./hybrid_hello.x
```

To submit the job:

```
sbatch hybrid.slurm
```

Comments:

```
### your favorite shell

### job name
### job queue
### request 2 node (48 physical cores)
### and 4 hours walltime
### request a license for scratch file system
### ask SLURM to send emails
### when jobs aborts, starts and ends

### 12 OpenMP threads per MPI task
### -n 4 : 4 MPI tasks (2 tasks per node)
### -c 24 : reserving 24 cores (12 physical plus
###         12 logical) per task;
###         although we only use physical cores
```

http://www.nersc.gov/users/computational-systems/edison/running-jobs/example-batch-scripts/

Sample Batch Script for Hybrid Jobs on Cori Phase I

Batch script *hybrid.slurm:*

```
#!/bin/bash -l

#SBATCH -J hybrid
#SBATCH -p regular
#SBATCH -C haswell
#SBATCH -N 2
#SBATCH -t 4:00:00
#SBATCH -L SCRATCH
#SBATCH --mail-user=shaw@ucsc.edu
#SBATCH --mail-type=ALL

export OMP_NUM_THREADS=16
srun -n 4 -c 32 ./hybrid_hello.x
```

To submit the job:

```
sbatch hybrid.slurm
```

Comments:

```
### your favorite shell

### job name
### job queue
### request a Haswell node
### request 2 node (64 physical cores)
### and 4 hours walltime
### request a license for scratch file system
### ask SLURM to send emails
### when jobs aborts, starts and ends

### 16 OpenMP threads per MPI task
### -n 4 : 4 MPI tasks (2 tasks per node)
### -c 32 : reserving 32 cores (16 physical plus
###         16 logical) per task;
###         although we only use physical cores
```

http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts/

Sample Batch Script for Hybrid Jobs on Cori Phase III: snc4 + flat

```
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -p regular
#SBATCH -C knl,snc4,flat
#SBATCH -S 4                # use core specialization
#SBATCH -t 3:00:00
#SBATCH -L SCRATCH,project

export OMP_NUM_THREADS=16 # 16 OpenMP threads per MPI task, using 64 cores in total
export OMP_PROC_BIND=true #"spread" is also good for Intel and CCE compilers
export OMP_PLACES=threads

# Add the following "sbcast" line here for jobs larger than 1500 MPI tasks:
# sbcast ./mycode.exe /tmp/mycode.exe

# 4 MPI ranks per node for a total of 8 MPI tasks
# executable linked with libmemkind
srun -n 8 -c 68 --cpu_bind=cores ./mycode.exe
```

http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts-for-knl/

Sample Batch Script for Hybrid Jobs on Hyades

Batch script *hybrid.pbs:*

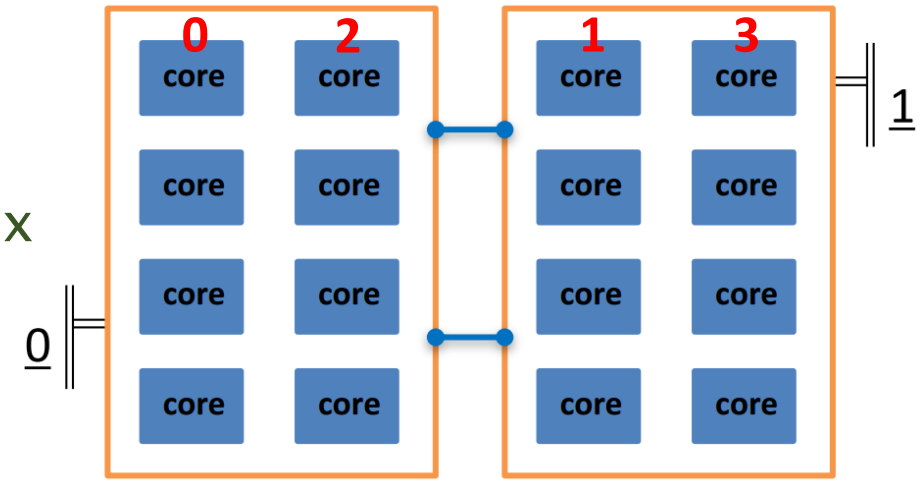
```
#!/bin/bash

#PBS -N hybrid
#PBS -q normal
#PBS -l nodes=2:ppn=16
#PBS -l walltime=0:10:00

cd $PBS_O_WORKDIR
cat $PBS_NODEFILE | sort | uniq > hosts.$PBS_JOBID

export OMP_NUM_THREADS=8
export I_MPI_PIN_DOMAIN=omp
export KMP_AFFINITY=compact

mpirun -machine hosts.$PBS_JOBID \
       -genv I_MPI_FABRICS shm:ofa \
       -n 4 -ppn 2 ./hybrid_hello.x
```

Comments:

```
### your favorite shell

### job name
### job queue
### request 2 node (16 cores per node)
### and 10 minutes walltime

### go to the work directory
### create a machine file

### 8 OpenMP threads per MPI task
### select process pinning scheme
### pin OpenMP threads

### run your hybrid executable
### -n 4: 4 MPI tasks in total
### -ppn 2: 2 MPI tasks per node
```

https://software.intel.com/en-us/articles/hybrid-applications-intelmpi-openmp

Try it out on Hyades

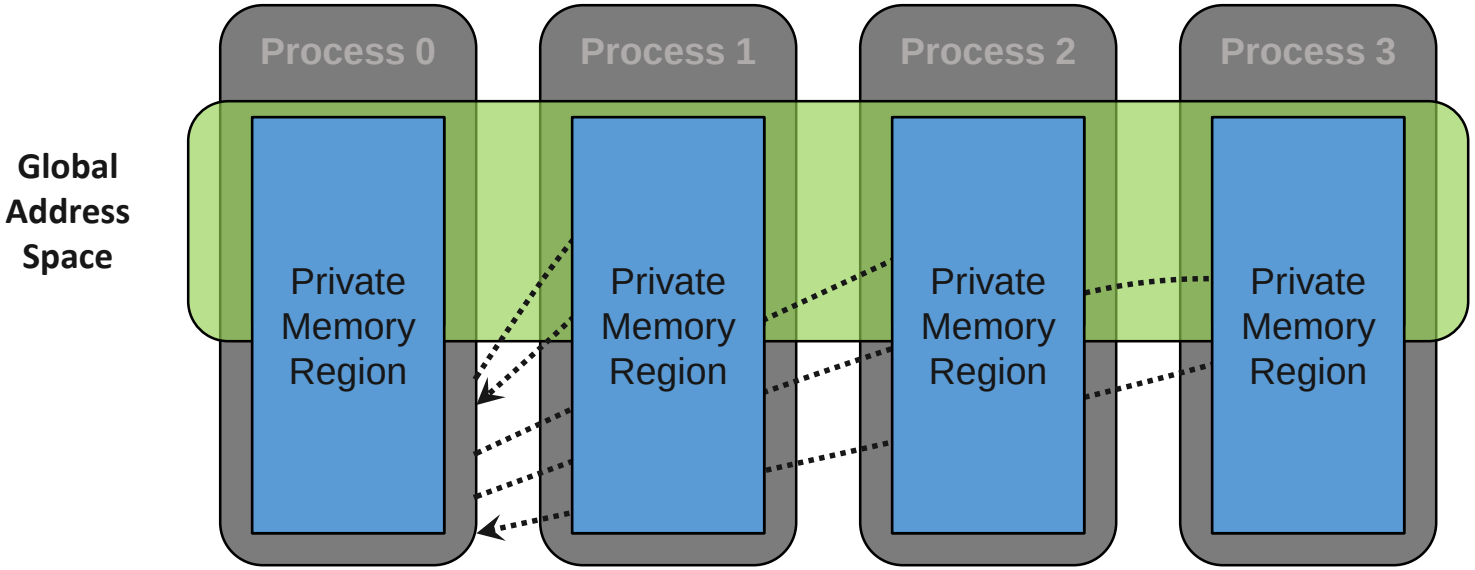



```
$ mpiicc -mt_mpi hybrid_hello.c -o hybrid_hello.x

$ qsub hybrid.pbs

$ cat hybrid.o125827
Hello from rank 0, thread 0, on astro-5-11.local. (core affinity = 0)
Hello from rank 0, thread 1, on astro-5-11.local. (core affinity = 2)
...
Hello from rank 1, thread 0, on astro-5-14.local. (core affinity = 0)
Hello from rank 1, thread 1, on astro-5-14.local. (core affinity = 2)
...
Hello from rank 2, thread 0, on astro-5-11.local. (core affinity = 1)
Hello from rank 2, thread 1, on astro-5-11.local. (core affinity = 3)
...
Hello from rank 3, thread 0, on astro-5-14.local. (core affinity = 1)
Hello from rank 3, thread 1, on astro-5-14.local. (core affinity = 3)
...
```

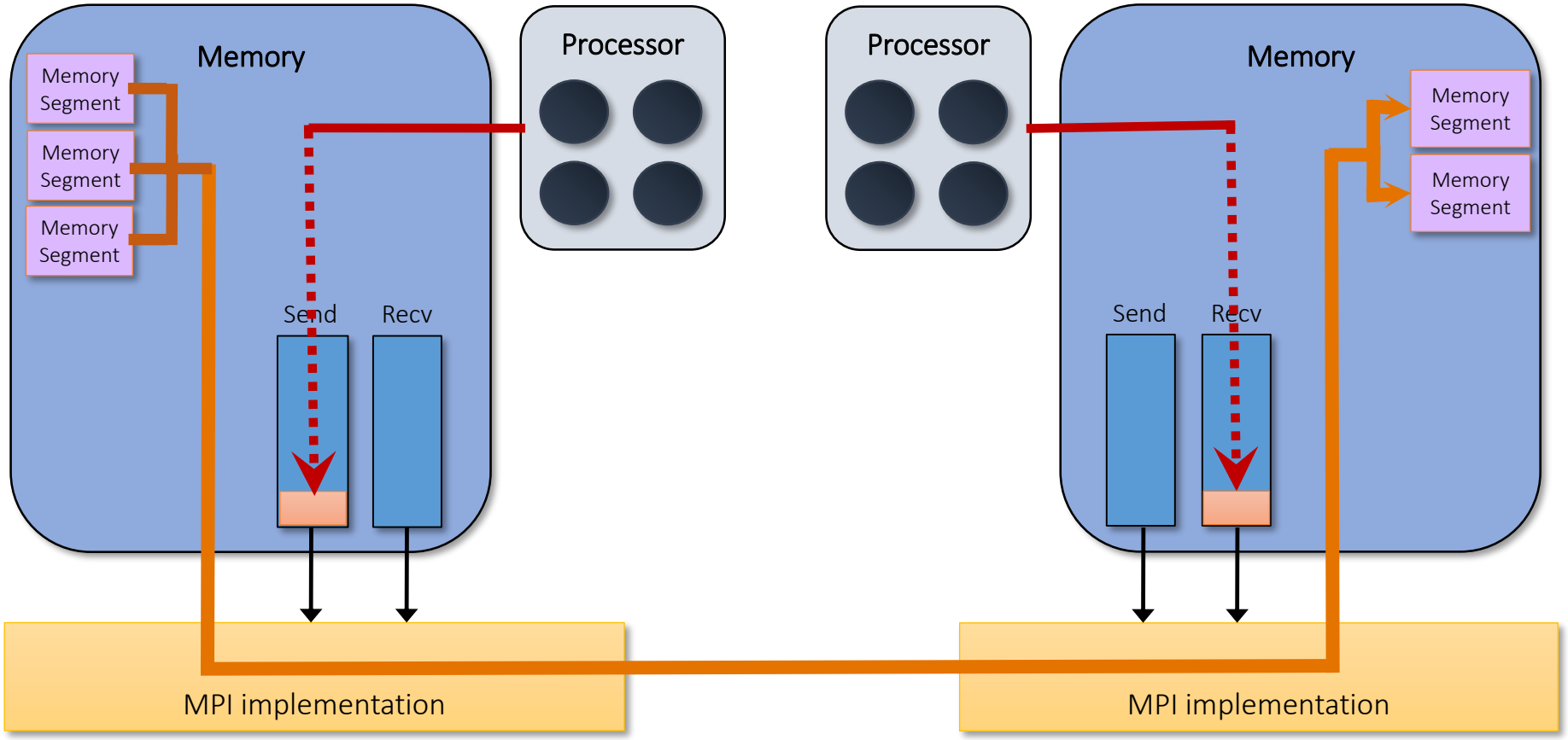
One-Sided Communications

The basic idea of one-sided communication models is to decouple data movement with process synchronization

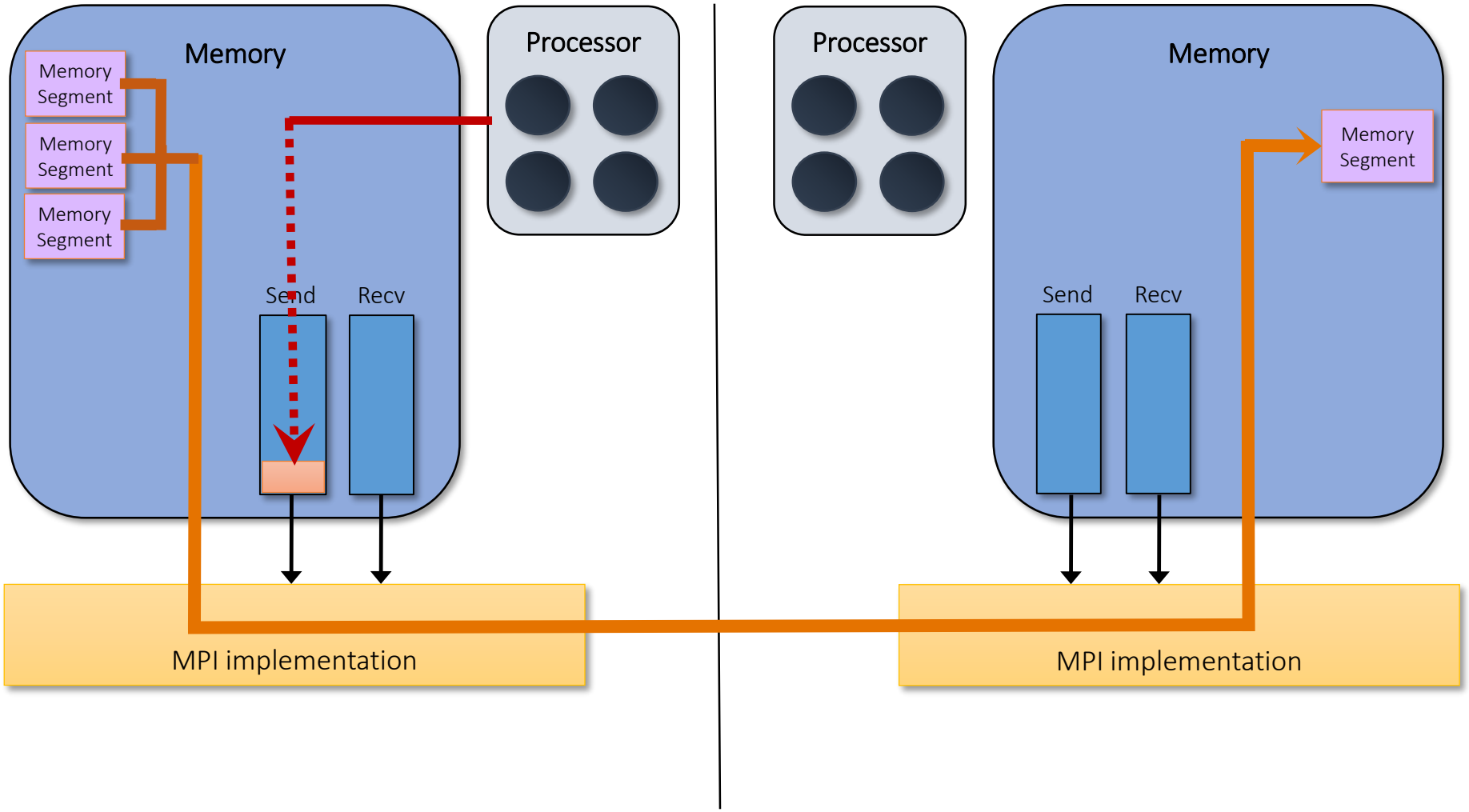
- Should be able move data without requiring that the remote process synchronize
- Each process exposes a part of its memory to other processes
- Other processes can *directly* read from or write to this memory



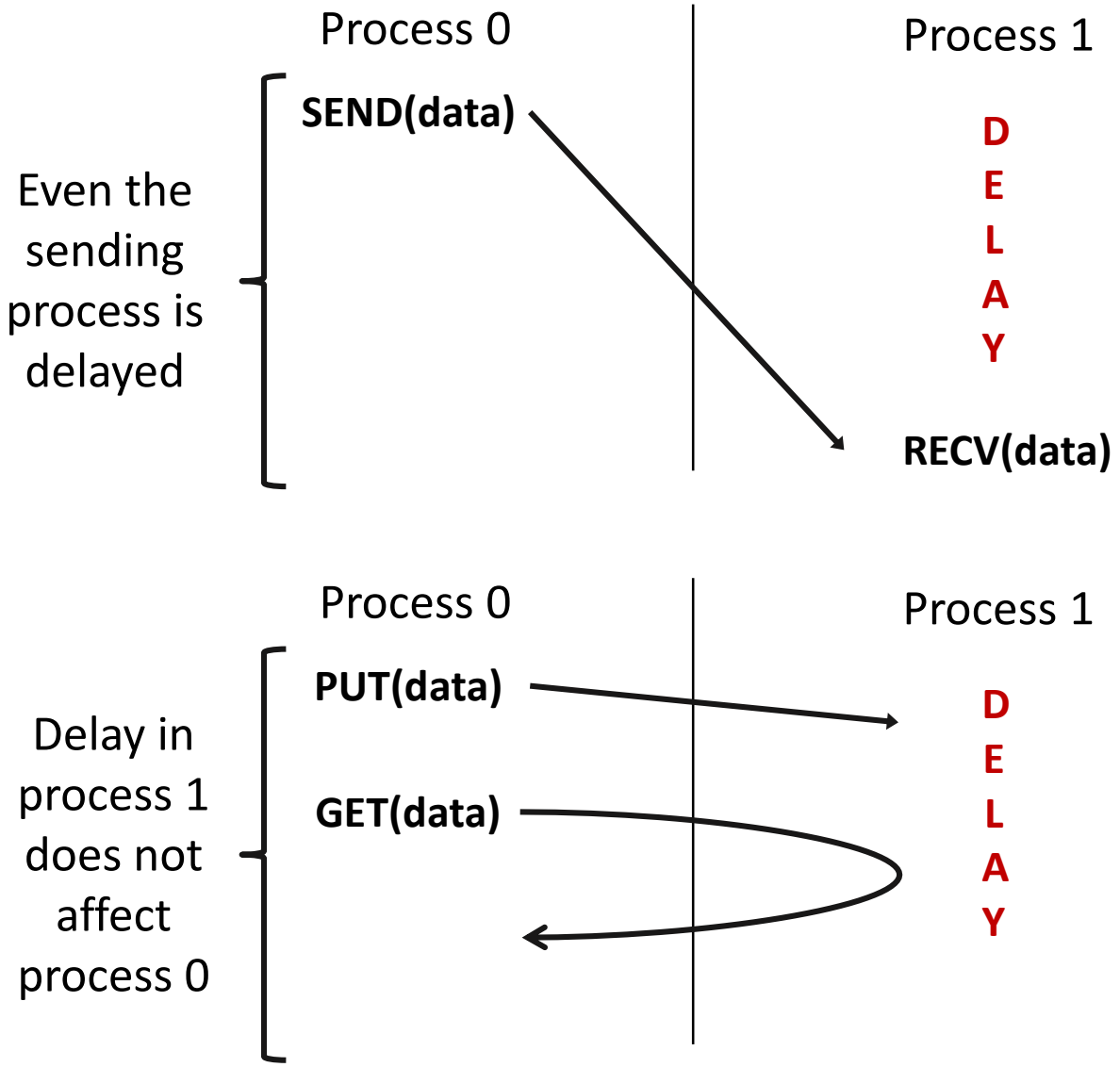
Two-Sided Communication Example



One-Sided Communication Example



One-Sided vs. Two-Sided



One-Sided Communications with MPI

- MPI-2 introduces routines for **one-sided communications**, aka **RMA** (remote memory access)
- MPI-3 adds new one-sides communication operations, to better handle different memory models
- Advantages of RMA operations:
  - Can do multiple data transfers with a single synchronization operation
  - 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 and RDMA

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 Communication 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**)

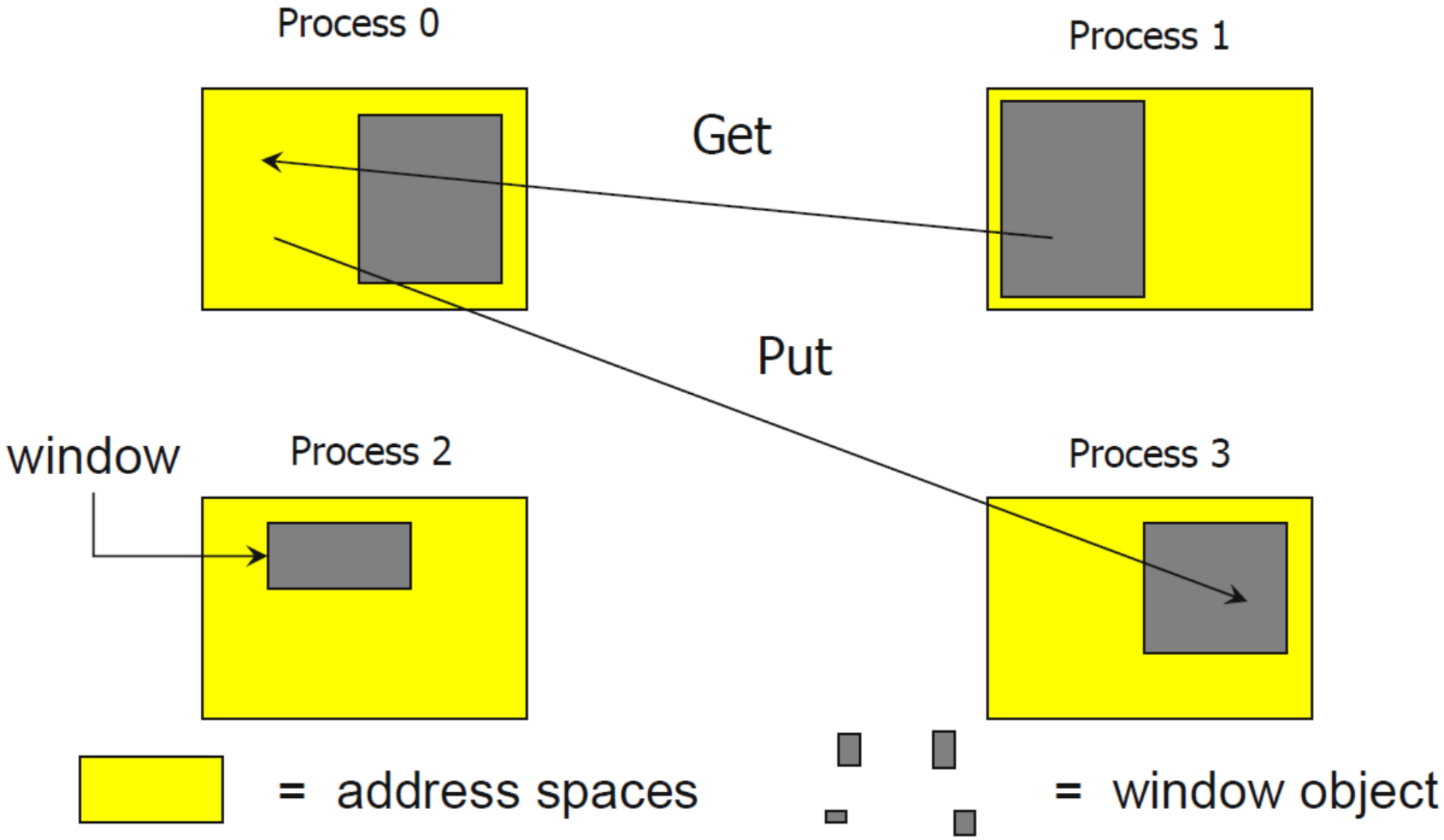
Creating Public Memory

- 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

RMA Windows and Window Objects



Window Creation Models

Four window creation models:

- 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

MPI_Win_create

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

Fortran	`MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)`
	`<type> BASE(*)`
	`INTEGER(KIND=MPI_ADDRESS_KIND) SIZE`
	`INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR`

- Expose a region of memory in an RMA window
- Collective call
- Arguments:
  - base – pointer to local data to expose
  - size – size of local data in bytes
  - disp_unit – local unit size for displacements, in bytes
  - info – info argument (handle)
  - comm – communicator (handle)
  - win – window object returned by the call (handle)

https://www.open-mpi.org/doc/current/man3/MPI_Win_create.3.php

MPI_Win_free

C	`int MPI_Win_free(MPI_Win *win)`
Fortran	`MPI_WIN_FREE(WIN, IERROR)` `    INTEGER WIN, IERROR`

- Frees the window object and returns a null handle
- Collective call

MPI_Win_create Example

```
#include "mpi.h"
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);
    free(a);
    MPI_Finalize();
    return 0;
}
```

MPI_Win_allocate

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

Fortran	`MPI_WIN_ALLOCATE(IZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)`
	`INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR`
	`INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR`

- Allocates memory and returns a window object for RMA operations
- Collective call
- Arguments:
  - size – size of window in bytes
  - disp_unit – local unit size for displacements, in bytes
  - info – info argument (handle)
  - comm – communicator (handle)
  - baseptr – initial address of window
  - win – window object returned by the call (handle)

MPI_Win_allocate Example

```
#include "mpi.h"
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), 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

C	`int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)`
Fortran	`MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)` `INTEGER INFO, COMM, WIN, IERROR`

- Creates an RMA window, to which data can later be attached
- Collective call
- Application can dynamically attach memory to this window
- Application can access data on this window only after a memory region has been attached

MPI_Win_create_dynamic Example

```
#include "mpi.h"
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);
    /* Array 'a' is now accessible from 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

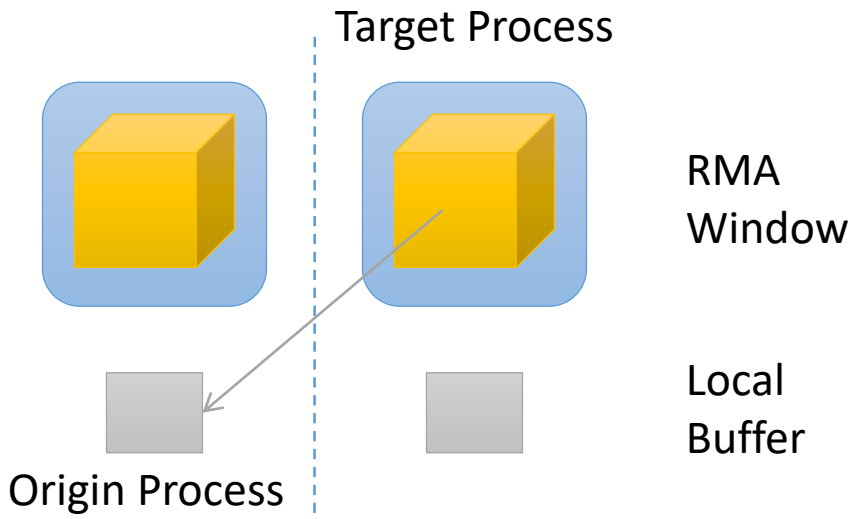
- MPI_Get
- MPI_Put
- MPI_Accumulate
- MPI_Get_accumulate
- MPI_Compare_and_swap
- MPI_Fetch_and_op

MPI_Get

C	`int MPI_Get(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)`

Fortran	`MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,`
	`  TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)`
	`  <type> ORIGIN_ADDR(*)`
	`  INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP`
	`  INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,`
	`  TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR`

- Copies data from the **target** memory to the **origin**
- Separate data description triples for **origin** and **target**

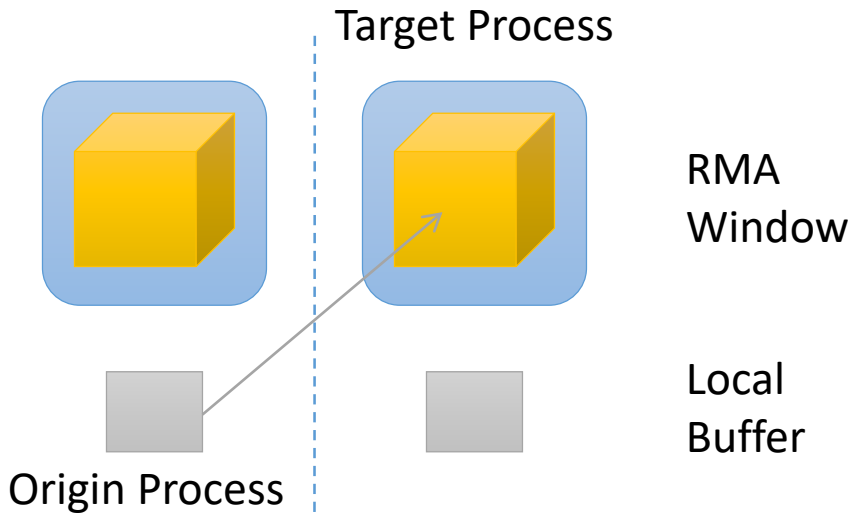


MPI_Put

C	`int 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)`

Fortran	`MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,`
	`  TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)`
	`  <type> ORIGIN_ADDR(*)`
	`  INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP`
	`  INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,`
	`  TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR`

- Copies data from the **origin** memory to the **target**
- Separate data description triples for **origin** and **target**
- Same arguments as MPI_Get

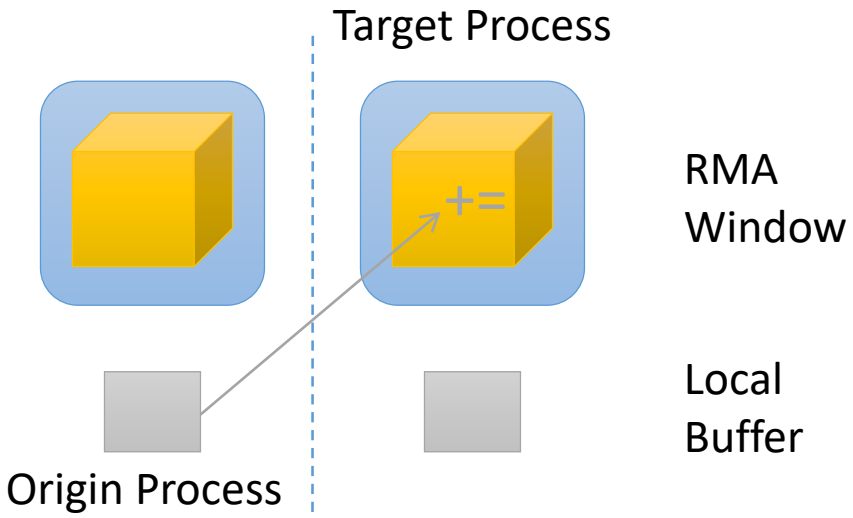


MPI_Accumulate

C	`int 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)`

Fortran	`MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)`
	`<type> ORIGIN_ADDR(*)`
	`INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP`
	`INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR`

- Combines the contents of the **origin** buffer with that of a **target** buffer
- Any of the predefined operations for **MPI_Reduce** can be used. User-defined functions cannot be used.
- Atomic *put* with op = **MPI_REPLACE**

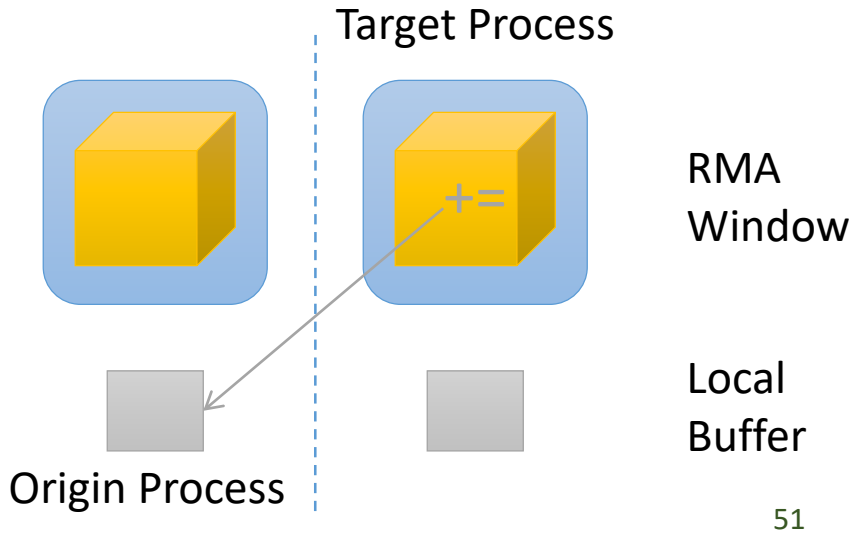


MPI_Get_accumulate

C	`int 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)`

Fortran	`MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)`
	`<type> ORIGIN_ADDR, RESULT_ADDR(*)`
	`INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP`
	`INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_COUNT, TARGET_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR`

- Combines the contents of the **origin** buffer with that of a **target** buffer and returns the **target** buffer value
- Predefined ops only, no user-defined!
- Atomic *get* with op = **MPI_NO_OP**
- Atomic *swap* with op = **MPI_REPLACE**



MPI_Compare_and_swap

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

Fortran	`MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK, TARGET_DISP, WIN, IERROR)`
	`<type> ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR(*)`
	`INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP`
	`INTEGER DATATYPE, TARGET_RANK, WIN, IERROR`

- Performs RMA compare-and-swap
- This function compares one element of type datatype in the compare buffer *compare_addr* with the buffer at offset *target_disp* in the target window specified by *target_rank* and *win* and replaces the value at the target with the value in the origin buffer *origin_addr* if the compare buffer and the target buffer are identical. The original value at the target is returned in the buffer *result_addr*.
- Atomic swap if target value is equal to compare value

MPI_Fetch_and_op

C	`int 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)`

Fortran	`MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK, TARGET_DISP, OP, WIN, IERROR)`
	`<type> ORIGIN_ADDR, RESULT_ADDR(*)`
	`INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP`
	`INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR`

- Combines the contents of the **origin** buffer with that of a **target** buffer and returns the **target** buffer value
- Simpler version of **MPI_Get_accumulate**
  - All buffers share a single predefined datatype
  - No count argument (it's always 1)
  - Simpler interface allows hardware optimization

Ordering of RMA Operations

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get and concurrent Put/Accumulate undefined
  - Can be garbage in both cases
- Result of concurrent accumulate operations to the same location are defined according to the order in which they occurred
  - Atomic put: **Accumulate** with op = **MPI_REPLACE**
  - Atomic get: **Get_accumulate** with op = **MPI_NO_OP**
- Accumulate operations from a given process are ordered by default
  - User can tell the MPI implementation that ordering is not required as optimization hint
  - You can ask for only the needed orderings, e.g., RAW (read-after-write), WAR, RAR, or WAW

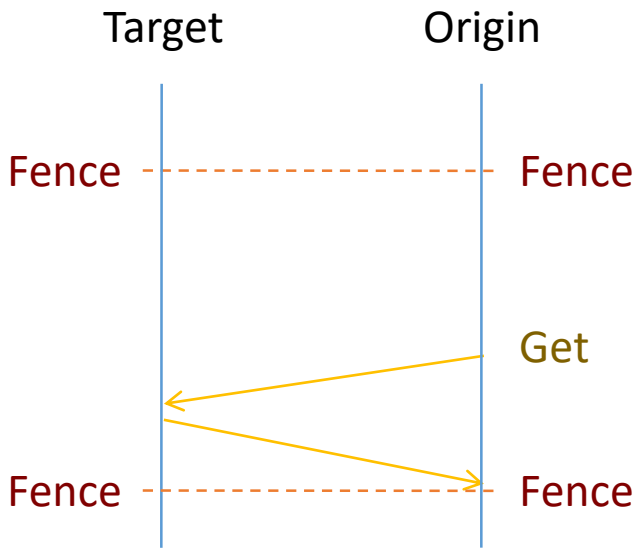
RMA Synchronization Models

- RMA data visibility
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
  - RMA synchronization models define these semantics
- Three synchronization models provided by MPI:
  - **Fence** (*active* target)
  - **Post-Start-Complete-Wait** (generalized *active* target)
  - **Lock/Unlock** (*passive* target)
- Data accesses occur within "epochs"
  - Access epochs: contain a set of operations issued by an origin process
  - Exposure epochs: enable remote processes to access and/or update a target's window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs
    - E.g., starting, ending, and synchronizing epochs

Fence: Active Target Synchronization

C	`int MPI_Win_fence(int assert, MPI_Win win)`
Fortran	`MPI_WIN_FENCE(ASSERT, WIN, IERROR)` `INTEGER ASSERT, WIN, IERROR`

- *Collective* synchronization model
- Starts and ends access and exposure epochs on all processes in the window
- All processes in group of "win" do an **MPI_Win_fence** to open an epoch
- Everyone can issue Put/Get operations to read/write data
- Everyone does an **MPI_Win_fence** to close the epoch
- All operations complete at the second fence synchronization



Example: Calculating π with MPI RMA

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[]) {
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_WIN nwin, piwin;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    if (myid == 0) {
        MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL,
                       MPI_COMM_WORLD, &nwin);
        MPI_Win_create(&pi, sizeof(double), 1, MPI_INFO_NULL,
                       MPI_COMM_WORLD, &piwin);
    } else {
        MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
                       MPI_COMM_WORLD, &nwin);
        MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL,
                       MPI_COMM_WORLD, &piwin);
    }
```

$$\frac{\pi}{4} = tan^{-1}(-1) = \int_0^1 \frac{dx}{1+x^2}$$

MPI_BOTTOM: a predefined constant, indicating the bottom of the address space

https://www.mcs.anl.gov/research/projects/mpi/usingmpi2/examples/starting/cpi-rma_c.htm

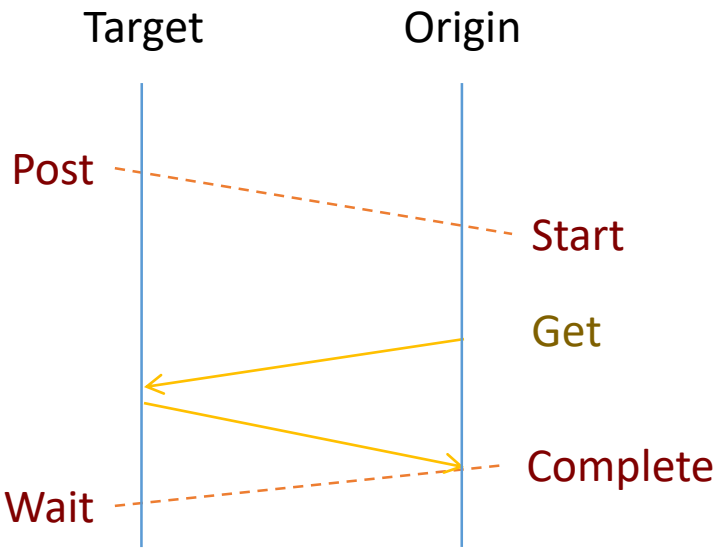
```
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
            pi = 0.0;
        }
        MPI_Win_fence(0, nwin);
        if (myid != 0) MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
        MPI_Win_fence(0, nwin);
        if (n == 0)
            break;
        else {
            h   = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Win_fence(0, piwin);
            MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE, MPI_SUM, piwin);
            MPI_Win_fence(0, piwin);
            if (myid == 0)
                printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
        }
    }
    MPI_Win_free(&nwin); MPI_Win_free(&piwin);
    MPI_Finalize();
    return 0;
}
```

$$\frac{\pi}{4} = tan^{-1}(-1) = \int_0^1 \frac{dx}{1+x^2}$$

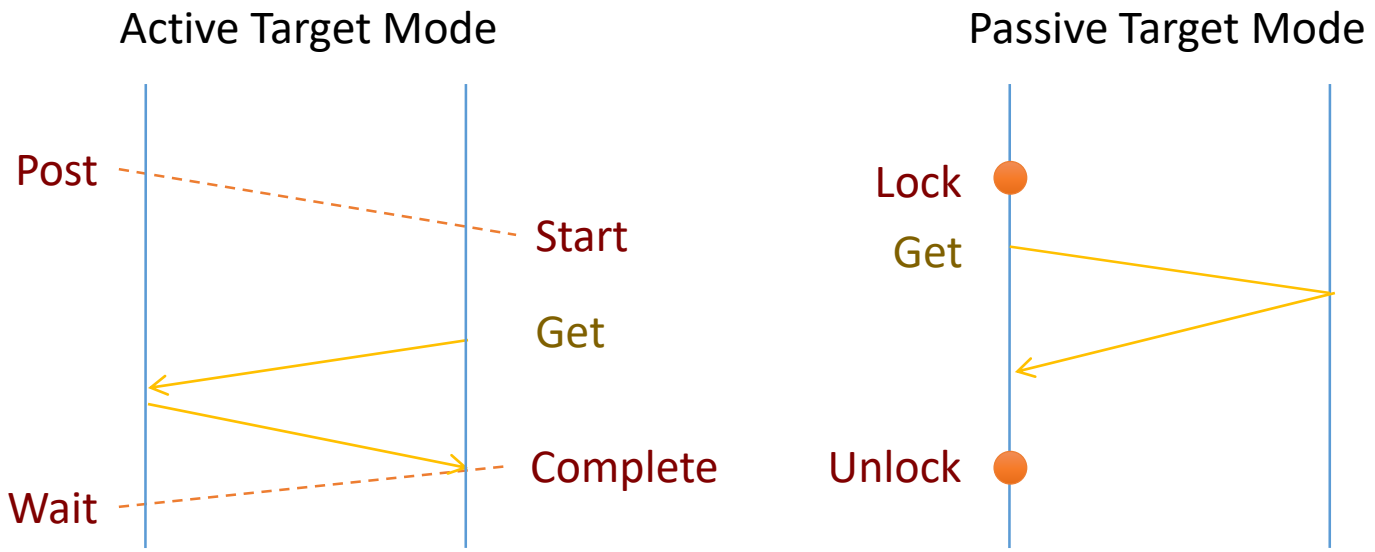
PSCW: Generalized Active Target Synchronization

```
int MPI_Win_post/start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete/wait(MPI_Win win)
```

- Like **Fence**, but origin and target specify who they communicate with
- 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_complete**
- All synchronization operations may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets



Passive Target Mode



- 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)
int MPI_Win_flush/flush_local(int rank, MPI_Win win)
```

- **Lock/Unlock**: Begin/end passive mode epoch
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs to different processes
  - Concurrent epochs to same process not allowed (affects threads)
- Lock type
  - **MPI_LOCK_SHARED**: Other processes using shared can access concurrently
  - **MPI_LOCK_EXCLUSIVE**: No other processes can access concurrently
- **Flush**: Remotely complete RMA operations to the target process
  - After completion, data can be read by target process or a different process
- **Flush_local**: Locally complete RMA operations to the target process

Lock is not Lock

- The name “Lock” is unfortunate
  - **Lock** is really “begin epoch”
  - **Unlock** is really “end epoch”
- An MPI “Lock” does not establish a critical section or mutual exclusion
  - With **MPI_LOCK_EXCLUSIVE** the RMA operations have exclusive access to the data they access/update during the time that they access the remote window
- This is very different than a “lock” in the sense of a thread lock

MPI_Win_lock Example

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

/* tests passive target RMA on 2 processes */

#define SIZE1 100
#define SIZE2 200

int main(int argc, char *argv[])
{
    int rank, nprocs, A[SIZE2], B[SIZE2], i;
    MPI_Win win;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (nprocs != 2) {
        printf("Run this program with 2 processes\n")
        MPI_Abort(MPI_COMM_WORLD,1);
    }
```

```
    if (rank == 0) {
        for (i=0; i<SIZE2; i++) A[i] = B[i] = i;
        MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &win);

        for (i=0; i<SIZE1; i++) {
            MPI_Win_lock(MPI_LOCK_SHARED, 1, 0, win);
            MPI_Put(A+i, 1, MPI_INT, 1, i, 1, MPI_INT, win);
            MPI_Win_unlock(1, win);
        }

        for (i=0; i<SIZE1; i++) {
            MPI_Win_lock(MPI_LOCK_SHARED, 1, 0, win);
            MPI_Get(B+i, 1, MPI_INT, 1, SIZE1+i, 1, MPI_INT, win);
            MPI_Win_unlock(1, win);
        }

        MPI_Win_free(&win);

    else {  /* rank=1 */
        for (i=0; i<SIZE2; i++) B[i] = (-4)*i;
        MPI_Win_create(B, SIZE2*sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
        MPI_Win_free(&win);
    }
    MPI_Finalize();
    return 0;
}
```

Advanced Passive Target Synchronization

```
int MPI_Win_lock_all(int assert, MPI_Win win)
int MPI_Win_unlock_all(MPI_Win win)
int MPI_Win_flush_all/flush_local_all(MPI_Win win)
```

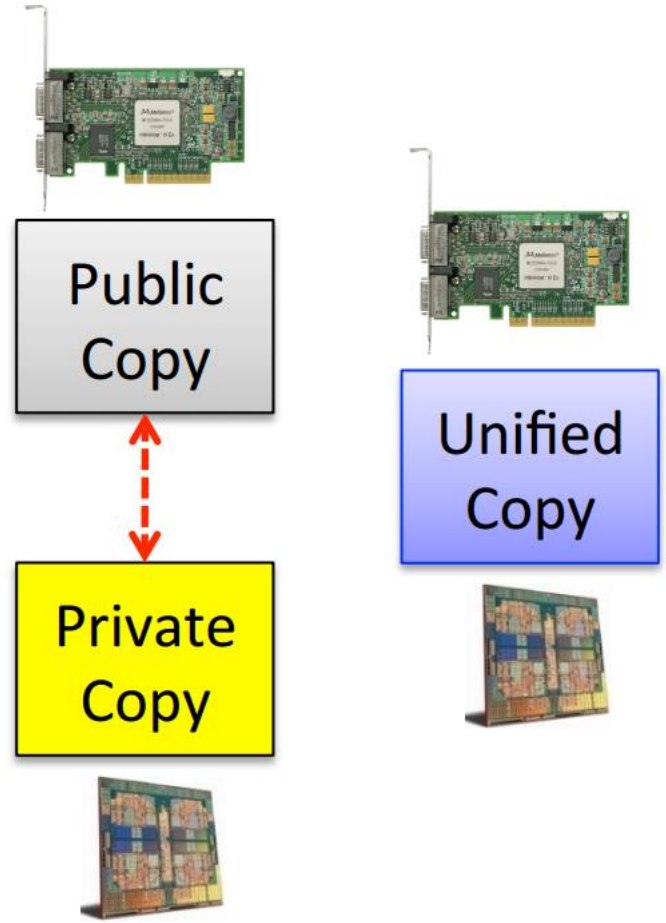
- **Lock_all**: starts an RMA access epoch locking access to all processes in the window, with a lock type of **MPI_LOCK_SHARED**
  - Expected usage is long-lived: lock_all, put/get, flush, …, unlock_all
- **Flush_all**: remotely completes RMA operations to all processes
- **Flush_local_all**: locally completes RMA operations to all processes

Which Synchronization Mode to Use?

- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc.
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g., InfiniBand)
- Active mode: bulk synchronization
  - E.g., ghost cell exchange
- Passive mode: asynchronous data movement
  - Useful when dataset is large, requiring memory of multiple nodes
  - And when data access and synchronization pattern is dynamic
  - Common use case: distributed, shared arrays
- Passive target locking mode
  - Lock/unlock – Useful when exclusive epochs are needed
  - Lock_all/unlock_all – Useful when only shared epochs are needed

Memory Models

- MPI-3 provides two memory models: **separate** and **unified**
  - Type is attached as attribute to window
- MPI-2: **Separate** Model
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable, to systems that don't provide hardware coherence
- MPI-3: New **Unified** Model
  - Single copy of the window
  - System must provide coherence
  - Superset of separate semantics
    - E.g., allows concurrent local/remote access
  - Provides access to full performance potential of hardware



Separate Semantics

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

This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

OVL – Overlapping operations permitted
NOVL – Nonoverlapping operations permitted
X – Combining these operations is OK, but data might be garbage

Unified Semantics

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

This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

OVL – Overlapping operations permitted
NOVL – Nonoverlapping operations permitted

Further Readings

- **Using MPI**, 3rd Edition, *by* William Gropp, Ewing Lusk & Anthony Skjellum, MIT Press, 2014
  http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6981847
- **Using Advanced MPI**, *by* William Gropp, Torsten Hoefler, Rajeev Thakur & Ewing Lusk, MIT Press, 2014
  http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6981848
- Using MPI and Using Advanced MPI:
  http://wgropp.cs.illinois.edu/usingmpiweb/