Salvatore Di Girolamo <digirols@inf.ethz.ch>

DPHPC: Locks

Recitation session
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

**T0**
- write(flag[0]=true)
- read(flag[1])==false
- write(flag[0])=false
- read(flag[0])=false

**T1**
- write(flag[1]=true)
- read(flag[0])==true
- write(flag[0])=false
- read(flag[0])=false
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(flag[0]=true)</td>
<td>read(flag[1])==true</td>
</tr>
<tr>
<td>write(flag[1]=true)</td>
<td>read(flag[0])==true</td>
</tr>
</tbody>
</table>
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
2-threads: Peterson lock

```c
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;  // I'm interested
    victim = tid;  // other goes first
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```
N-threads: Filter Lock

```c
volatile int level[n] = {0,0,…,0}; // highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level

void lock() {
    for (int i = 1; i < n; i++) { // attempt level i
        level[tid] = i;
        victim[i] = tid;
        // spin while conflicts exist
        while ((\exists k \neq tid) (level[k] >= i && victim[i] == tid )) {
        }
    }
}

void unlock() {
    level[tid] = 0;
}
```

- At least one thread trying to enter level L succeeds
- If more than one thread is trying to enter level L, then at least one is blocked (waits at that level)
N-threads: Peterson locks in a binary tree

- Another way to generalize the Peterson lock to $n \geq 2$ threads is to use a binary tree, where each node holds a Peterson lock for two threads.
  - Threads start at a leaf in the tree, and move one level up when they acquire the lock at a node.
  - A thread that holds the lock of the root can enter its critical section.
  - When a thread exits its critical section, it releases the locks of nodes that it acquired.
First-Come-First-Served Locks

- Doorway: bounded number of steps
- Waiting: unbounded number of steps

\[ D^j_A \rightarrow D^k_B, \text{ then } CS^j_A \rightarrow CS^k_B \]
N-threads: Bakery lock

```java
volatile int flag[n] = {0,0,...,0};
volatile int label[n] = {0,0,...,0};

void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
    while ((∃k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {}
}

c
```

- What happens if two threads execute their doorways concurrently?
  - Lexicographical order helps us!

- A thread could see a set of labels that never existed in memory at the same time
Flaky Lock

Programmers at the Flaky Computer Corporation designed the protocol shown below to achieve n-thread mutual exclusion. Does this protocol satisfy mutual exclusion? Is it starvation-free? Is it deadlock-free?

```java
class Flaky implements Lock {
    private int turn;
    private boolean busy = false;
    public void lock () {
        int me = ThreadID.get();
        do {
            do {
                turn = me;
            } while (busy);
            busy = true;
        } while (turn != me);
    }
    public void unlock () {
        busy = false;
    }
}
```
S. Di Girolamo [digirols@inf.ethz.ch]

DPHPC: MPI RMA
Recitation Session

Slides credits: Pavan Balaji, Torsten Hoefler
One-sided Communication

- 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

MPI implementation

Memory

Processor

Memory

Processor

Send

Recv

Memory Segment

Memory Segment

Memory Segment

Send

Recv

Memory Segment

Memory Segment

Memory Segment

MPI implementation
One-sided Communication Example

MPI implementation

Memory

Processor

Memory Segment

Send

Recv

Memory Segment

Memory Segment

Memory Segment

Processor

Memory

Memory Segment

Send

Recv

MPI implementation
Comparing One-sided and Two-sided Programming

Even the sending process is delayed

Delay in process 1 does not affect process 0
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 remotely accessible memory

- Any memory used by a process is, by default, only locally accessible
  - $X = \text{malloc}(100)$;

- Once the memory is allocated, 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
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
    You may want to dynamically add/remove buffers to/from the window
  - MPI_WIN_ALLOCATE_SHARED
    You want multiple processes on the same node share a buffer
MPI_WIN_CREATE

- 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)
  - win - window (handle)
Example with MPI_WIN_CREATE

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

    MPI_Init(&argc, &argv);

    /* create private memory */
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    /* 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_Free_mem(a);
    MPI_Finalize(); return 0;
}
```
MPI_WIN_ALLOCATE

- Create a remotely accessible memory region in an RMA window
  - Only 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
  - win: window (handle)
Example with MPI_WIN_ALLOCATE

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

    MPI_Init(&argc, &argv);

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

    /* Array ‘a’ is now accessible from all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);  // will also free the buffer memory

    MPI_Finalize(); return 0;
}
```
MPI_WIN_CREATE_DYNAMIC

- Create an RMA window, to which data can later be attached
  - Only data exposed in a window can be accessed with RMA ops

- Initially “empty”
  - Application can dynamically attach/detach memory to this window by calling MPI_Win_attach/detach
  - Application can access data on this window only after a memory region has been attached

- Window origin is MPI_BOTTOM
  - Displacements are segment addresses relative to MPI_BOTTOM
  - Must tell others the displacement after calling attach

MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
Example with MPI_WIN_CREATE_DYNAMIC

```c
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 = (int *) 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 accessible from all processes */

    /* undeclare remotely accessible memory */
    MPI_Win_detach(win, a);  free(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_PUT
  - MPI_GET
  - MPI_ACCUMULATE (atomic)
  - MPI_GET_ACCUMULATE (atomic)
  - MPICOMPARE_AND_SWAP (atomic)
  - MPI_FETCH_AND_OP (atomic)
Data movement: **Put**

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

```c
MPI_Put(void *origin_addr, int origin_count,
         MPI_Datatype origin_dtype, int target_rank,
         MPI_Aint target_disp, int target_count,
         MPI_Datatype target_dtype, MPI_Win win)
```
Data movement: Get

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

```c
MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_dtype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_dtype, MPI_Win win)
```
Atomic Data Aggregation: **Accumulate**

- **Atomic update operation, similar to a put**
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, ...
  - Predefined ops only, no user-defined operations

- **Different data layouts between target/origin OK**
  - Basic type elements must match

- **Op = MPI_REPLACE**
  - Implements $f(a,b)=b$
  - Atomic PUT

**MPI_Accumulate**

```c
void *origin_addr, int origin_count,
MPI_Datatype origin_dtype, int target_rank,
MPI_Aint target_disp, int target_count,
MPI_Datatype target_dtype, MPI_Op op, MPI_Win win)
```
Atomic Data Aggregation: Get Accumulate

- **Atomic read-modify-write**
  - $O_p = \text{MPI\_SUM, MPI\_PROD, MPI\_OR, MPI\_REPLACE, MPI\_NO\_OP, ...}$
  - Predefined ops only
- **Result stored in target buffer**
- **Original data stored at result_addr**
- **Different data layouts between target/origin OK**
  - Basic type elements must match
- **Atomic get with MPI\_NO\_OP**
- **Atomic swap with MPI\_REPLACE**

```c
MPI_Get_accumulate(void *origin_addr, int origin_count,
                    MPI_Datatype origin_dtype, void *result_addr,
                    int result_count, MPI_Datatype result_dtype,
                    int target_rank, MPI_Aint target_disp,
                    int target_count, MPI_Datatype target_dtype,
                    MPI_Op op, MPI_Win win)
```
Atomic Data Aggregation: **CAS and FOP**

- **FOP**: 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

- **CAS**: Atomic swap if target value is equal to compare value

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

```c
MPI_Compare_and_swap(void *origin_addr, void *compare_addr,
                     void *result_addr, MPI_Datatype dtype, int target_rank,
                     MPI_Aint target_disp, MPI_Win win)
```
Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get 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 the 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 (s)he does not require ordering as optimization hint
  - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW
Examples with operation ordering

1. Concurrent Puts: undefined

2. Concurrent Get and Put/Accumulates: undefined

3. Concurrent Accumulate operations to the same location: ordering is guaranteed
RMA Synchronization Models

- **RMA data access model**
  - 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 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

- 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

```c
MPI_Win_fence(int assert, MPI_Win win)
```
PSCW: Generalized Active Target Synchronization

- 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

```c
MPI_Win_post/start(MPI_Group grp, int assert, MPI_Win win)
MPI_Win_complete/wait(MPI_Win win)
```
Lock/Unlock: Passive Target Synchronization

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

- **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**
  - **SHARED:** Other processes using shared can access concurrently
  - **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**
Advanced Passive Target Synchronization

- **Lock_all**: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: lock_all, put/get, flush, …, unlock_all
- **Flush_all** – remotely complete RMA operations to all processes
- **Flush_local_all** – locally complete RMA operations to all processes

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

```c
MPI_Win_unlock_all(MPI_Win win)
```

```c
MPI_Win_flush_all/flush_local_all(MPI_Win win)
```
Which synchronization mode should I use, when?

- **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
  - Also, 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
MPI RMA Memory Model

- **MPI-3 provides two memory models: separate and unified**
- **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
MPI RMA Memory Model (separate windows)

- Very portable, compatible with non-coherent memory systems
- Limits concurrent accesses to enable software coherence
MPI RMA Memory Model (unified windows)

- Allows concurrent local/remote accesses
- Concurrent, conflicting operations are allowed (not invalid)
  - Outcome is not defined by MPI (defined by the hardware)
- Can enable better performance by reducing synchronization