

# CSL 860: Modern Parallel Computation

# **MEMORY CONSISTENCY**

# Intuitive Memory Model

- Reading an address should **return the last value written**
  - Easy in uniprocessors
  - Cache coherence problem in MPs
- “A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” [Lamport, 1979]

# Memory Consistency Semantics

- Threads always see values written by some thread
  - No garbage
- The value seen is constrained by thread-order
  - for every thread
- Example: *spin lock*

```
initially:  ready=0, data=0

thread1           thread 2

data = 1           while(!ready);
ready = 1          pvt = data
```

**If P2 sees the new value of flag (=1),  
it must see the new value of data (=1)**

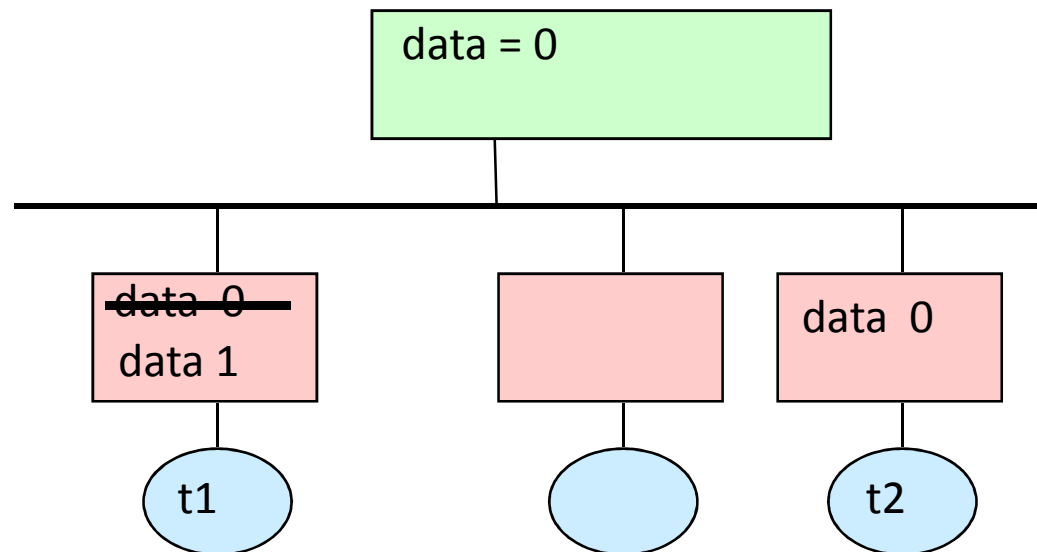
<b>If P2 reads flag</b>	<b>Then P2 may read data</b>
0	1
0	0
1	1

# OpenMP: Weak Consistency

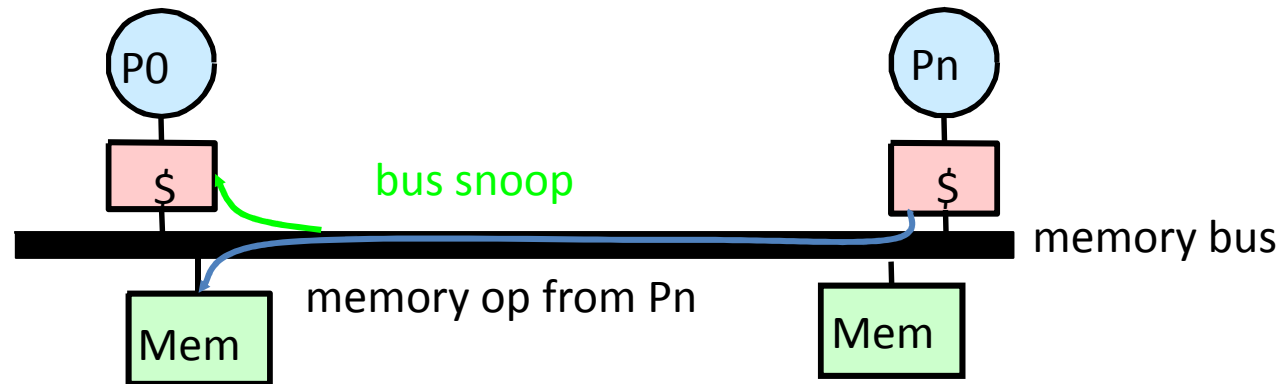
- Special memory 'sync' operations (flush)
- Order only syncs with respect to each other
- Two flushes of the same variable are synchronization operations
- Non-intersecting flush-sets are not ordered with respect to each other

# Cache Coherency

- Different copies of same location do not have the same value
  - thread1 and thread2 both have cached copies of data
- t1 writes data=1
  - But may *not* “write through” to memory
- t2 reads data, but gets the “stale” copy
  - This may happen even if t2 read an *updated* value of another variable



# Snoopy Cache-Coherence Protocol



- All transactions to shared memory visible to all processors
- Caches contain information on which addresses they store
- Cache Controller “snoops” all transactions on the bus
- Ensures coherence if a relevant transaction
  - invalidate, update, or supply value

# Memory Consistency Hazards

- The compiler reorders/removes code
  - The compiler usually sees only local memory dependencies
- Some form of inconsistent cache
  - The compiler can even allocate a register for some *shared* variable
- System may reorder writes to merge addresses (not FIFO)
  - Write X=1, Y=1, X=2
  - Second write to X may happen before Y's, 1<sup>st</sup> write may never happen
- The network can also reorder the two write messages.

## **Solutions:**

- Tell compiler about (asynchronous) update to variable
- Avoid race conditions
  - If you have race conditions on variables, make them volatile

# Review

- Memory consistency
  - Sequential consistency is the natural semantics
  - Lock access to shared variable for read-modify-write
  - Architecture ensures consistency
  - But compilers (and programs) may still get in the way
    - Non-blocking writes, read pre-fetching, code reordering
- Memory performance
  - May allocate data in large shared region
  - Understanding memory hierarchy is critical to performance
    - Traffic can be incoherent
  - Also watch for sharing
    - Both *true* and *false* sharing

# **PERFORMANCE ISSUES**

# Memory Performance

- True sharing
  - Frequent writes to a variable can create a bottleneck
  - OK for read-only or infrequently written data
  - **Solution:** make copies of the value, one per processor, if possible
    - Do not allocate in threads arbitrarily from heap
- False sharing
  - Two distinct variables in the same cache block
  - **Solution:** allocate contiguous block per processor (thread)
    - But best to consider memory bank stride (more later)

# Other Performance Considerations

- Keep critical region short
- Limit Fork-Join (and all synchronization points)
  - e.g., If few iterations, fork/join overhead exceeds time savings from parallel execution of loop
  - Can use conditional for construct
- Invert loops if
  - Parallelism is in the inner loop
  - But be mindful of memory coherence
  - And memory latency
- Use enough threads
  - Easier to balance load
  - Easier to hide memory latency (more on this later)
  - See if worker-queue model applied

# Example: Shorten Critical Region

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    #pragma omp critical
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

# Other Performance Considerations

- Keep critical region short
- Limit Fork-Join (and all synchronization points)
  - e.g., If few iterations, fork/join overhead exceeds time savings from parallel execution of loop
  - Can use conditional for construct
- Invert loops if
  - Parallelism is in the inner loop
  - But be mindful of memory coherence
  - And memory latency
- Use 'enough' threads
  - Smaller tasks make it easier to balance load
  - Easier to hide memory latency
  - See if work-queue model applies

# Example: Work Queue

```
#pragma omp parallel private(task_ptr)
{
    task_ptr = get_next_task (&job_ptr);
    while (task_ptr != NULL) {
        complete_task (task_ptr);
        task_ptr = get_next_task (&job_ptr);
    }
}
char *get_next_task(Job_struct **job_ptr)
{
    Task_struct *answer;
    #pragma omp critical
    {
        answer = (*job_ptr)->task;
        *job_ptr = (*job_ptr)->next;
    }
    return answer;
}
```

# Parallel Program Design

- Partition into “concurrent” tasks
  - Determine granularity
    - Start with  $\gg 10$  times #processors
    - Target similar size
  - Minimize dependence/communication
- Manage data communication (sharing)
  - Determine synchronization points
- Group tasks
  - Balance load
  - Reduce communication
- Map each group to a processor

# Example Decomposition Techniques

- Pipeline decomposition
- Recursive decomposition
  - Divide and conquer
- Data decomposition
  - Partition input/output (or intermediate) data
  - Multiple independent output (or input)
  - Computation follows
- Exploratory decomposition
  - Search problems
- Speculative decomposition
  - Conditional execution of dependent tasks
- Hybrid

# Mapping

- Easier if number and sizes of tasks can be predicted
- Knowledge of data size per task
- Inter-task interaction pattern
  - Static vs dynamic
    - Dynamic interaction is harder with distributed memory
    - Requires polling or support for *signaling*
  - Regular vs Irregular
  - Read-only vs Read-write
  - One-way vs two-way
    - Only one thread actively involved in ‘one-way’

# Processor Mapping

- Goals
  - Reduce no-work time
  - Reduce communication
  - Usually a trade-off
- Approaches
  - Static
    - Knap-sack (NP-complete)
    - Use heuristics
  - Dynamic
    - Apply when load is dynamic
    - Work-queue approach

# Static Processor Mapping

- Data Partitioning
  - Array Distribution
    - Block distribution
    - Cyclic or Block-cyclic
    - Randomized block distribution
  - Graph Partitioning
    - Allocate sub-graphs to each processor
- Task Partitioning
  - Task interaction graph

# Reducing Communication Overhead

- Interaction means ‘wait’
- Communication in shared memory
  - Between levels of memory hierarchy
- Increase locality of reference
  - See if data replication is an option
- Batch communication if possible
  - Locally store intermediate results
  - Design a ‘strided’ communication pattern
- Fill wait time with useful work
  - Which is independent of the communication

# **GENERAL PROGRAMMING TIPS**

# Parallel Programming Tips

- Know your target architecture
  - Indicates the number of threads to use
  - Make design scalable
- Know your application
  - Look for data and task parallelism
  - Try to keep threads independent
  - Low synchronization and communication
  - For generality, start fine grained, then combine
    - Parametrically if possible
  - Make sure “hotspots” are parallelised
- Use thread-safe libraries
- Never assume the state of a variable (or another thread)
  - Always enforce it when required

# Parallel Programming Tips II

- Use 'closer' memory as much as possible
  - Usually requires proper (local) declarations
  - Also indirectly reduces synchronization
  - Do not create dependency by variable reuse
    - Sometimes better to re-compute values
- Lock data at the finest grain
  - Trade off with overhead of number of locks/operations
  - Consider lock-free/non-blocking synchronization
    - Consider 'batching' updates
  - Clearly associate each lock and data it protects
  - Only necessary processing inside critical region
- Avoid malloc/new

# Parallel Programming Tips III

- Number of threads can be
  - Functionality/task based
    - Usually they do not all run together
  - Performance based
    - The count is quite important
- One thread per processor is not always the best option
  - Idle processor due to fetching from memory or I/O
  - Can often reduce idle time
    - Stride I/O, memory access etc.
    - Intersperse computation phases
  - Be mindful of per-thread overhead
  - Beware of too many compute intensive threads
- Consider work-queue paradigm
  - Threads take work from queue and complete them in sequence