Module B: Parallelization Techniques

Course TBD
Lecture TBD
Term TBD

Module developed 2013 - 2014
by Martin Burtscher

This module was created with support form NSF under grant # DUE 1141022
Part 1: Parallel Array Operations

• Finding the max/min array elements
• Max/min using 2 cores
• Alternate approach
• Parallelism bugs
• Fixing data races
Goals for this Lecture

- Learn how to parallelize code
  - Task vs. data parallelism
- Understand parallel performance
  - Speedup, load imbalance, and parallelization overhead
- Detect parallelism bugs
  - Data races on shared variables
- Use synchronization primitives
  - Barriers to make threads wait for each other
Maximum Array Element

- Problem: find the largest element in an array
  - \( arr = \begin{bmatrix} 3 & 2 & 0 & 7 & 6 & 1 & 9 & 5 & 0 & 3 & 8 & 4 & 1 & 2 & 5 & 8 & 6 & 9 & 7 & 4 \end{bmatrix} \)

- Serial code
  ```
  // arr is an array of SIZE elements that are comparable
  max = -infinity;
  for (i = 0; i < SIZE; i++)
    if (max < arr[i]) max = arr[i];
  ```

- Loop iteration order
  - Any order that includes all array elements is correct, making it easy to parallelize this code

Execution time: SIZE iterations
Maximum and Minimum Elements

- Problem: find the largest element in one array and the smallest element in another array
  - \( arrA = \) 3 2 0 7 6 1 9 5 0 3 8 4 1 2 5 8 6 9 7 4
  - \( arrB = \) 6 8 2 9 1 9 0 8 7 3 5 2 0 4 0 6 7 4 5 1

- Serial code

```c
// using two loops
max = -infinity;
for (i = 0; i < SIZE; i++)
    if (max < arrA[i]) max = arrA[i];
min = infinity;
for (i = 0; i < SIZE; i++)
    if (min > arrB[i]) min = arrB[i];
```

Execution time: 2 * SIZE iterations
Max/Min using 2 Cores

- Core 0 computes $max$ while core 1 computes $min$
  - This is called task parallelism (cores run different code)

```
// core 0 code (max)
max = -infinity;
for (i = 0; i < SIZE; i++)
  if (max < arrA[i]) max = arrA[i];

// core 1 code (min)
min = infinity;
for (i = 0; i < SIZE; i++)
  if (min > arrB[i]) min = arrB[i];
```

- Speedup = 2
  - Using 2 cores is twice as fast as using 1 core
  - Can we get more speedup with additional cores?

No, this approach does not scale to more cores as there are only two tasks.
Max/Min using 2 Cores (Version 2)

- Each core processes half of the data
- This is called data parallelism (cores run same code)

```
// core 0 code (lower half)
max = -infinity;
for (i = 0; i < SIZE / 2; i++)
    if (max < arrA[i]) max = arrA[i];
min = infinity;
for (i = 0; i < SIZE / 2; i++)
    if (min > arrB[i]) min = arrB[i];

// core 1 code (upper half)
max = -infinity;
for (i = SIZE / 2; i < SIZE; i++)
    if (max < arrA[i]) max = arrA[i];
min = infinity;
for (i = SIZE / 2; i < SIZE; i++)
    if (min > arrB[i]) min = arrB[i];
```

- Speedup = 2
  - Using 2 cores is twice as fast as using 1 core

This approach is straightforward to scale to larger numbers of cores
Max/Min using $N$ Cores (Version 2a)

• Make code scalable and the same for each core
  • With $N$ cores, give each core one $N^{\text{th}}$ of the data
  • Each core has an ID: coreID $\in$ 0..$N-1$; numCores = $N$

```c
// code (same for all cores)
beg = coreID * SIZE / numCores;
end = (coreID+1) * SIZE / numCores;
max = -infinity;
for (i = beg; i < end; i++)
  if (max < arrA[i]) max = arrA[i];
min = infinity;
for (i = beg; i < end; i++)
  if (min > arrB[i]) min = arrB[i];
```

• Speedup = $N$
  • Using $N$ cores is $N$ times as fast as using 1 core

Execution time: $2 * SIZE / N$ iterations

Compute which chunk of array the core should process
But wait…

- Parallelism bug
  - The code *sometimes* computes an **incorrect** result
  - For illustration, assume the 2-element array $arrA = \begin{bmatrix} 9 & 4 \end{bmatrix}$

```c
// core 0
max = -infinity;
for (i = 0; i < 1; i++)
  if (max < arrA[i]) max = arrA[i];
min = ...

// core 1
max = -infinity;
for (i = 1; i < 2; i++)
  if (max < arrA[i]) max = arrA[i];
min = ...
```

- Problem: both cores read and then write $\text{max}$ without first synchronizing with the other core
- This is a **data race** (which cannot occur in serial code)
Eliminating the Data Race

• Using private variables
  • This bug can be avoided by not sharing \textit{max} and \textit{min}
  • The arrays \textit{arrA} and \textit{arrB} should still be shared

```
// core 0
max0 = -infinity;
for (i = beg; i < end; i++)
  if (max0 < arrA[i]) max0 = arrA[i];
min0 = infinity;
for (i = beg; i < end; i++)
  if (min0 > arrB[i]) min0 = arrB[i];

// core 1
max1 = -infinity;
for (i = beg; i < end; i++)
  if (max1 < arrA[i]) max1 = arrA[i];
min1 = infinity;
for (i = beg; i < end; i++)
  if (min1 > arrB[i]) min1 = arrB[i];
```

• New problem
  • The code now computes 2 minimums and 2 maximums
  • These partial solutions must be combined into 1 solution
Combining the Partial Solutions

- Core 0 reduces partial solution into final solution

```c
// core 0 code
max0 = -infinity;
for (i = beg; i < end; i++)
    if (max0 < arrA[i]) max0 = arrA[i];
min0 = infinity;
for (i = beg; i < end; i++)
    if (min0 > arrB[i]) min0 = arrB[i];
if (max0 < max1) max0 = max1;
if (min0 > min1) min0 = min1;

// core 1 code
max1 = -infinity;
for (i = beg; i < end; i++)
    if (max1 < arrA[i]) max1 = arrA[i];
min1 = infinity;
for (i = beg; i < end; i++)
    if (min1 > arrB[i]) min1 = arrB[i];
if (max1 < max0) max1 = max0;
if (min1 > min0) min1 = min0;
```

- Two new problems
  - Speedup is lowered due to parallelization overhead (extra work) and load imbalance (core 0 does more)
  - Extra code introduces a new data race
Adding Synchronization

- Core 0 must wait for core 1 if necessary
  - Need a synchronization primitive called a **barrier**
  - Barriers make all cores wait for slowest core (thread)

```c
// core 0 code
max0 = -infinity;
for (i = beg; i < end; i++)
  if (max0 < arrA[i]) max0 = arrA[i];
min0 = infinity;
for (i = beg; i < end; i++)
  if (min0 > arrB[i]) min0 = arrB[i];
barrier();
if (max0 < max1) max0 = max1;
if (min0 > min1) min0 = min1;

// core 1 code
max1 = -infinity;
for (i = beg; i < end; i++)
  if (max1 < arrA[i]) max1 = arrA[i];
min1 = infinity;
for (i = beg; i < end; i++)
  if (min1 > arrB[i]) min1 = arrB[i];
barrier();
```

- Now the parallel code works correctly
  - This idea also works with more than 2 cores
Summary of Part 1

- Task parallelism
  - Cores execute different code

- Data parallelism
  - Cores execute same code on different parts of data

- Data race
  - Unsynchronized accesses to shared data (incl. write)

- Private variable
  - Each core (thread, really) gets its own copy

- Reduction
  - Combine multiple partial results into one final result

- Barrier synchronization
  - Wait for all threads of program to reach barrier
Part 2: Parallelizing Rank Sort

- Rank sort algorithm
- Work distribution
- Parallelization approaches
- OpenMP pragmas
- Performance comparison
Goals for this Lecture

- Learn how to assign a balanced workload
  - Chunked/blocked data distribution
- Explore different ways to parallelize loops
  - Tradeoffs and complexity
- Get to know parallelization aids
  - OpenMP, atomic operations, reductions, barriers
- Understand performance metrics
  - Runtime, speedup and efficiency
Rank Sort Algorithm

• Given an array with unique elements, place the elements into another array in increasing order
  • For example, A = [4, 5, 1, 7, 6, 2, 9] B = [1, 2, 4, 5, 6, 7, 9]

• This algorithm counts how many elements are smaller to determine the insertion point
  • For example, there are 5 elements that are smaller than 7, so the 7 will have to go into B[5]
  • Similarly, there are 2 elements that are smaller than 4, so the 4 will have to go into B[2], etc.
Rank Sort Implementation

• Doubly nested loop
  • Outer loop goes over all elements of array A
  • Inner loop counts how many elements are smaller
  • Finally, element is inserted at corresponding index

```c
// A and B are arrays of SIZE elements, A’s elements are unique
for (i = 0; i < SIZE; i++) {
    cnt = 0;
    for (j = 0; j < SIZE; j++) {
        if (A[i] > A[j]) cnt++;
    }
    B[cnt] = A[i];
}
```

Execution time:
SIZE^2 iterations
Balancing the Work Across Threads

- Assume that we have $T$ threads (cores) and that $\text{SIZE}$ is an integer multiple of $T$
  - Each thread has a unique ID = 0, 1, ..., $T-1$
- Then we can easily assign each thread an equal chunk of work of $W = \text{SIZE} / T$ array elements
  - Each thread gets elements $\text{ID} \times W$ through $(\text{ID}+1) \times W - 1$
- E.g., $\text{SIZE} = 20$ and $T = 4$ ($W = 5$) yields this:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
 3 -2  0  7 -6  1 -9  5 10 -3 -8  4 -1  2 -5  8  6  9 -7 -4
```

thread 0  thread 1  thread 2  thread 3
We first attempt to parallelize the outer loop

- Read-only variables can safely be shared: W, SIZE, A[]
- Each thread needs some private variables: i, j, cnt, ID
  - Needed to avoid data races and overwriting each other’s data
- For unique elements in A[], the algorithm guarantees that the writes to shared B[] won’t result in a data race

```
// identical parallel code for each thread (ID is different)
for (i = ID*W; i < (ID+1)*W; i++) {
  cnt = 0;
  for (j = 0; j < SIZE; j++) {
    if (A[i] > A[j]) cnt++;
  }
  B[cnt] = A[i];
}
```

Execution time:
SIZE² / T

Complexity is still O(n²)
Automatic Parallelization with OpenMP

- Many compilers support OpenMP parallelization
  - Programmer has to mark which code to parallelize
  - Programmer has to provide some info to compiler
- Special OpenMP *pragmas* serve this purpose
  - They are ignored by compilers w/o OpenMP support

```c
// parallelization using OpenMP
#pragma omp parallel for private(i, j, cnt) shared(A, B, SIZE)
for (i = 0; i < SIZE; i++) {
    cnt = 0;
    for (j = 0; j < SIZE; j++) {
        if (A[i] > A[j]) cnt++;
    }
    B[cnt] = A[i];
}
```

Pragma tells compiler to parallelize this *for* loop
Pragma clauses provide additional information
Compiler automatically generates ID, W, etc.
Parallel Rank Sort (Versions 2 and 3)

• We now attempt to parallelize the inner loop
  • Outer loop code is run by one thread only
  • Multiple threads are used to run the inner loop
  • Problem: lots of potential data races on cnt

```c
// identical parallel code for each thread (ID is different)
for (i = 0; i < SIZE; i++) {
    cnt = 0;
    for (j = ID*W; j < (ID+1)*W; j++) {
        if (A[i] > A[j]) cnt++;
    }
    B[cnt] = A[i];
}
```

Execution time: 
\[ \frac{\text{SIZE}^2}{T} / \text{iterations} \]
Parallel Rank Sort (Versions 2 and 3)

- Avoiding possible data races on cnt
  - Should cnt be a private or a shared variable?
    - If shared, then the increment may result in a data race
      - We need an atomic (uninterruptable) increment
    - If private, then we have to combine the partial counts
      - We need to reduce (combine) the many counts into one

```c
// parallel code with atomic increment
for (i = 0; i < SIZE; i++) {
    cnt = 0;
    for (j = ID*W; j < (ID+1)*W; j++) {
        if (A[i] > A[j]) atomicInc(cnt);
    }
    B[cnt] = A[i];
}

// parallel code with reduction
for (i = 0; i < SIZE; i++) {
    c[ID] = 0;
    for (j = ID*W; j < (ID+1)*W; j++) {
    }
    cnt = reduce(c, T);
    B[cnt] = A[i];
}
```
OpenMP Code (Versions 2 and 3)

```
// OpenMP code with atomic increment
#pragma omp parallel for private(j)
shared(A, SIZE, cnt)
for (j = 0; j < SIZE; j++) {
    if (A[i] > A[j])
        #pragma omp atomic
cnt++;
}
```

```
// OpenMP code with reduction
#pragma omp parallel for private(j)
shared(A, SIZE) reduction(+ : cnt)
for (j = 0; j < SIZE; j++) {
    if (A[i] > A[j])
cnt++;
}
```

• Performance implications
  • Atomic version prevents multiple threads from accessing cnt simultaneously, i.e., lowers parallelism
  • Reduction version includes extra code, which slows down execution and causes some load imbalance
Which Version is Fastest?

- Need to measure execution time

<table>
<thead>
<tr>
<th>Version</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>version 1: outer</td>
<td>0.505</td>
<td>0.253</td>
<td>0.127</td>
<td>0.066</td>
</tr>
<tr>
<td>version 2: atomic</td>
<td>1.735</td>
<td>2.673</td>
<td>4.401</td>
<td>17.335</td>
</tr>
<tr>
<td>version 3: reduction</td>
<td>0.515</td>
<td>0.291</td>
<td>0.178</td>
<td>0.130</td>
</tr>
</tbody>
</table>

- Version 1 is the **fastest** and also the simplest to write
- Version 3 is slower, especially with more threads
  - The reduction code incurs a significant overhead
- Version 2 is slow and **slows down** with more threads
  - Atomics are convenient but slower than normal operations
  - Many **interleaved** accesses to a shared variable are very bad
### Input Size Dependence

- **Runtime of version 1 for different input sizes**

<table>
<thead>
<tr>
<th>number of values</th>
<th>time to sort (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 thread</td>
</tr>
<tr>
<td>25,000</td>
<td>0.50</td>
</tr>
<tr>
<td>50,000</td>
<td>2.02</td>
</tr>
<tr>
<td>100,000</td>
<td>8.18</td>
</tr>
<tr>
<td>200,000</td>
<td>32.73</td>
</tr>
<tr>
<td>400,000</td>
<td>130.98</td>
</tr>
</tbody>
</table>

- **Absolute runtime**
  - Useful for determining how long the code runs
  - **Difficult** to see how well the parallelization works

- \( O(n^2) \) time complexity is apparent for all thread counts.
Speedup Metric

• **Ratio** of the serial over the parallel runtime

<table>
<thead>
<tr>
<th>number of values</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>25,000</td>
<td>1.00</td>
<td>1.99</td>
<td>3.99</td>
<td>7.68</td>
</tr>
<tr>
<td>50,000</td>
<td>1.00</td>
<td>1.96</td>
<td>3.91</td>
<td>7.58</td>
</tr>
<tr>
<td>100,000</td>
<td>1.00</td>
<td>2.00</td>
<td>3.86</td>
<td>7.58</td>
</tr>
<tr>
<td>200,000</td>
<td>1.00</td>
<td>1.99</td>
<td>3.94</td>
<td>7.71</td>
</tr>
<tr>
<td>400,000</td>
<td>1.00</td>
<td>1.99</td>
<td>3.93</td>
<td>7.71</td>
</tr>
</tbody>
</table>

• **Speedup**
  - Tells us *how much faster* the parallel code runs
  - Depends on the number of threads used

In this example, the speedups are close to the thread counts
## Efficiency Metric

- **Ratio** of the speedup over the number of threads

<table>
<thead>
<tr>
<th>number of values</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>25,000</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.96</td>
</tr>
<tr>
<td>50,000</td>
<td>1.00</td>
<td>0.98</td>
<td>0.98</td>
<td>0.95</td>
</tr>
<tr>
<td>100,000</td>
<td>1.00</td>
<td>1.00</td>
<td>0.96</td>
<td>0.95</td>
</tr>
<tr>
<td>200,000</td>
<td>1.00</td>
<td>0.99</td>
<td>0.98</td>
<td>0.96</td>
</tr>
<tr>
<td>400,000</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
<td>0.96</td>
</tr>
</tbody>
</table>

- **Efficiency**
  - Tells us **how close** performance is to linear speedup
  - Measures how efficiently the cores are utilized

This code **scales very well** to eight threads.
Summary of Part 2

- Rank sort algorithm
  - Counts number of smaller elements
- Blocked/chunked workload distribution
  - Assign equal chunk of contiguous data to each thread
- Selecting a loop to parallelize
  - Which variables should be shared versus private
  - Do we need barriers, atomics, or reductions
- OpenMP
  - Compiler directives to automatically parallelize code
- Performance metrics
  - Runtime, speedup, and efficiency
Part 3: Parallelizing Linked-List Operations

- Linked lists recap
- Parallel linked list operations
- Locks (mutual exclusion)
- Performance implications
- Alternative solutions
Goals for this Lecture

• Explore different parallelization approaches
  • Tradeoffs between ease-of-use, performance, storage

• Learn how to think about parallel activities
  • Reading, writing, and overlapping operations

• Get to know locks and lock operations
  • Acquire, release, mutual exclusion (mutex)

• Study performance enhancements
  • Atomic compare-and-swap
Linked List Recap

• Linked list structure

```
struct node {
    int data;
    node* next;
};
```
Linked List Operations

• Contains
  • `bool Contains(int value, node* head);`
  • Returns true if value is in list pointed to by head

• Insert
  • `bool Insert(int value, node* &head);`
  • Returns false if value was already in list

• Delete
  • `bool Delete(int value, node* &head);`
  • Returns false if value was not in list
Inserting a New Node

- Create new node (*temp) and set value (e.g., 6)
- Find insertion location (between *pred and *curr)
- Set and redirect pointers
  - temp->next = pred->next
  - pred->next = temp
Deleting an Existing Node

- Find predecessor (*pred) and node (*curr)
- Redirect pointer of predecessor
  - pred->next = curr->next
Thinking about Parallel Operations

• General strategy
  • Break each operation into atomic steps
    • Operation = contains, insert, or delete (in our example)
    • Only steps that access shared data are relevant
  • Investigate all possible true interleavings of steps
    • From the same or different operations
    • Usually, it suffices to only consider pairs of operations
  • Validate correctness for overlapping data accesses
    • Full and partial overlap may have to be considered

• Programmer actions
  • None if all interleavings & overlaps yield correct result
  • Otherwise, need to disallow problematic cases
Running Contains in Parallel

- Not a problem as the threads only read the list
- No action needed in absence of writes to shared data
Running Contains and Insert in Parallel

- **Scenario 1**
  - Thread 2 ($T_2$) follows P
  - Thread 1 ($T_1$) updates P

  - Works fine
    - Thread 2 sees list before insertion

```
  head -----> 2 -----> 5 -----> P -----> 9
    ^       |       |       |       |
    |       |       |       |       |
thread 1:  |       |       |       |
          |       |       |       |
thread 1:  |       |       |       |
          |       |       |       |
thread 2:  |       |       |       |
          |       |       |       |
thread 1:  |       |       |       |
          |       |       |       |
thread 1:  |       |       |       |
          |       |       |       |
thread 1:temp
```
Running Contains and Insert in Parallel

- Scenario 2
  - $T_1$ updates $P$
  - $T_2$ follows $P$
  - Works fine
    - $T_2$ sees list after insertion
    - $Q$ must be set before $P$
      - Typically done naturally
Running Insert and Insert in Parallel

- Not a problem if non-overlapping locations
- Locations = all shared data (fields of nodes) that are accessed by both threads and written by at least one

```
thread 2: temp

head  2  3
  5  9

thread 1: temp

  6
```

}[Image 12x452 to 708x462]
Running Insert and Insert in Parallel

- Not a problem if non-overlapping in time
- One thread updates pointer P before the other thread reads P during its traversal to find insertion location

```
thread 1:
      temp

head → 2 → 5 → 9

thread 2:
      temp

      7

      Q

      P

      6

      R
```
Running Insert and Insert in Parallel

- Problem 1 if overlapping in time and in space
  - List may end up not being sorted (or with duplicates)
  - $T_1: R = P; T_1: P = \text{temp}; T_2: Q = P; T_2: P = \text{temp}$

[Diagram of linked list operations with thread 1 and thread 2]
Running Insert and Insert in Parallel

• Problem 2 if **overlapping** in time and in space
  • List may end up not containing one of inserted nodes
  • $T_1: R = P; T_2: Q = P; T_1: P = \text{temp}; T_2: P = \text{temp}$
Locks

- Lock variables
  - Can be in one of two states: locked or unlocked

- Lock operations
  - Acquire and release (both are atomic)
  - “Acquire” locks the lock if possible
    - If the lock has already been locked, acquire blocks or returns a value indicating that the lock could not be acquired
  - “Release” unlocks a previously acquired lock

- At most one thread can hold a given lock at a time, which guarantees mutual exclusion
Avoiding Conflicts Using One Lock

- Preventing conflicts among parallel Inserts
  - In addition to a head, the linked list needs a lock

- Every Insert first has to acquire the lock
  - Guarantees that at most one Insert will take place
  - Serializes all insertions, i.e., no parallelism

- Enhancement
  - Find insertion location first and then acquire lock
    - May have to correct insertion location after lock acquire
    - pred->next may not be equal to curr (should not use curr)
  - Doesn’t work in the presence of concurrent deletes
Avoiding Conflicts Using Many Locks

- Include a lock in every node
  - Significant storage overhead

- Locking nodes during traversal
  - Repeatedly lock curr node, unlock pred node
    - High overhead
    - Prevents faster threads from passing slower threads

- Enhancement
  - Lock only pred node after traversal but before insertion
    - May have to correct insertion location after lock acquire
    - Doesn’t work in the presence of concurrent deletes
Running Contains and Delete in Parallel

- Scenario 1
  - $T_2$ updates $P$
  - $T_1$ follows $P$

- Works fine
  - Thread 1 sees list after del.
Running Contains and Delete in Parallel

- **Scenario 2**
  - $T_1$ follows $P$
  - $T_2$ updates $P$

- **Does not work**
  - Thread 1 sees deleted node for arbitrary long time after deletion
  - If deleted node is freed by $T_2$ and memory is reused before $T_1$ accesses $curr$->$next$, program may crash
Running Delete and Delete in Parallel

- Scenario 1: deleting the same node
  - Seems to work depending on when curr->next is read
  - But second free of current node will fail (and crash)
Running Delete and Delete in Parallel

- Scenario 2: deleting adjacent nodes
  - $T_1$ first: $T_1$’s deleted node is added again but freed
  - $T_2$ first: $T_1$’s node is not deleted but freed
  - Segmentation fault likely if freed memory is reused
Running Insert and Delete in Parallel

• Scenario 1: inserting right after deleted node
  • Inserted node is **not added** to list
  • Segmentation fault possible if P updated after curr freed
Running Insert and Delete in Parallel

- Scenario 2: inserting right before deleted node
  - $T_1$ first: $T_1$’s inserted node is not added to the list
  - $T_2$ first: $T_2$’s node is not deleted but freed (seg. fault)
Freeing Nodes

- After a node’s memory has been freed
  - System may reuse memory for other purposes
  - Updating fields in a freed node can break program
  - Dereferencing a field in a freed node that was reused may result in an invalid address (segmentation fault)

- Not freeing deleted nodes
  - Eliminates seg. faults but causes memory leaks
  - Should free nodes once it is safe to do so
    - Difficult to know when it is safe (who should delete node?)
    - This is automatically done in managed languages like Java

- Alternative: mark deleted nodes but don’t remove
  - Truly remove deleted nodes occasionally (lock list)
Performance Considerations

- Scalable solution requires a lock per node
  - But large overhead in runtime, code, and memory
  - Slower threads can slow down faster threads
- Use a read/write lock in each node
  - Allows many readers or one writer at a time (3 states)
  - Even insert and delete mostly read (during traversal)
    - Insert must lock pred node for writing
    - Delete must lock pred and curr nodes for writing
  - Faster threads can pass slower threads during reading
  - Still large overhead in runtime, code, and memory
- Use more parallelism friendly data structure
  - Skip list, list of arrays, B-tree, etc.
Implementing Locks w/o Extra Memory

- Memory usage
  - Only need one or two bits to store state of lock
  - Next pointer in each node does not use least significant bits because they point to aligned memory address (the next node)

- Use unused pointer bits to store lock information
  - Since computation is cheap and memory accesses are expensive, this reduces runtime and memory use
  - But the coding and locking overheads are even higher
  - Need atomic operations (e.g., atomicCAS) to acquire lock (see later)
Avoiding Read Locks

- Don’t lock for reading, only for writing
  - Essentially no locking overhead (like serial code)
  - Insert must lock pred node
  - Delete must lock pred and curr nodes

- Potential problems
  - Locking must follow a fixed order to avoid deadlock
    - E.g., lock pred node before curr node
  - Insert/delete must restart if they fail to acquire lock
    - Delete must release first lock if it cannot acquire second lock
  - Delete must not free or modify deleted node
    - Causes memory leak
Lock-free Implementation

- Avoiding locks altogether
  - No memory overhead
  - Almost no performance overhead
  - Almost perfect parallelism
  - Some coding overhead
  - Hardware needs to support atomic operations, for example “atomic compare and swap”

- Atomic CAS
  - Allows to redirect pointers atomically if pointer hasn’t changed...
Atomic CAS

- Performs following operations atomically
  - Other threads cannot see intermediate results

```c
int atomicCAS(int *addr, int cmp, int val) {
    atomic {
        int old = *addr;
        if (old == cmp) {
            *addr = val;
        }
    }
    return old;
}
```
**Insertion using Atomic CAS**

```c
int atomicCAS(int *addr, int cmp, int val)
{
    atomic {
        int old = *addr;
        if (old == cmp) {
            *addr = val;
        }
    }
    return old;
}
```

**Insert value “6”:** (deleted nodes are marked but stay in list)

```c
do {
    find insertion point pred, curr;  // traverse list
    temp->next = curr;
} while (atomicCAS(&pred->next, curr, temp) != curr);
```
Summary of Part 3

• Non-overlapping accesses in space
  • No problem, accesses ‘modify’ disjoint parts of the data structure

• Non-overlapping accesses in time
  • No problem, accesses are naturally serialized

• Overlapping accesses (races) can be complex
  • Can have subtle effects
  • Sometimes they work
  • Sometimes they cause crashes much later when program reuses freed memory locations

• Need to consider all possible interleavings
Summary of Part 3 (cont.)

- Locks ensure mutual exclusion
  - Programmer must use locks consistently
  - Incur runtime and memory usage overhead
- One lock per data structure
  - Simple to implement but serializes accesses
- One lock per data element
  - Storage overhead, code complexity, but fine grained
- Lockfree implementations may be possible
  - Often good performance but more complexity
- Should use parallelism-friendly data structure