

CSE 160

Lecture 16

MIDTERM Review

Announcements

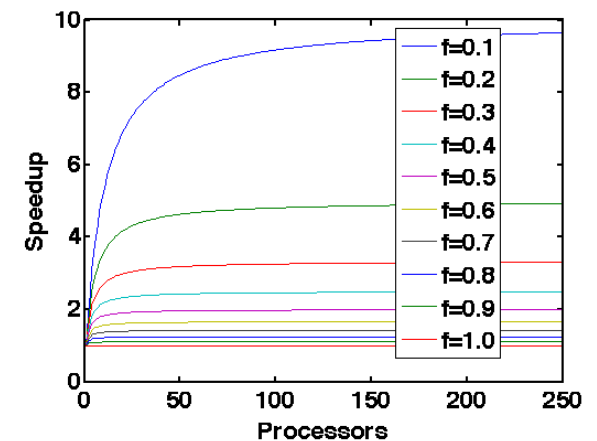
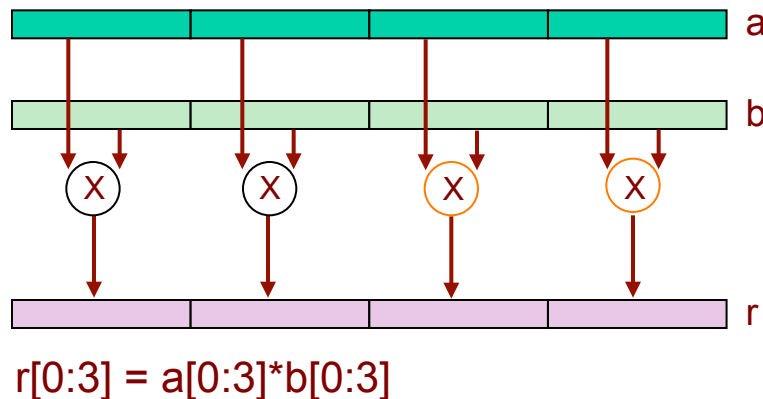
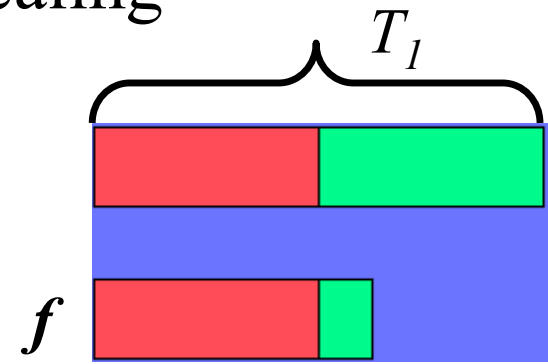
- No lab session Friday
- Midterm Exam will be held in CENTR 222
- Midterm exam is closed book, closed notes
- No Blue Book needed

Today's Lecture

- Midterm Review

Terms and concepts

- Know the definition and significance of
- Parallel speedup and efficiency, super-linear speedup, strong scaling, weak scaling
- Amdahl's law, Gustafson's law, serial bottlenecks
- Strong and Weak Scaling
- SSE



Terms and concepts

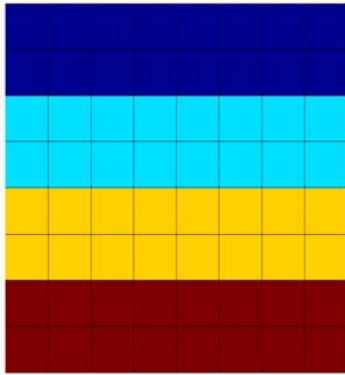
- SPMD, MIMD, SIMD
- Multiprocessors and multicomputers
- NUMAs and SMPs
- Processor Memory Gap
- Cache coherence and consistency
- Snooping
- False sharing
- Data dependencies, loop carried dependence
- Critical sections, race conditions

Implementation techniques

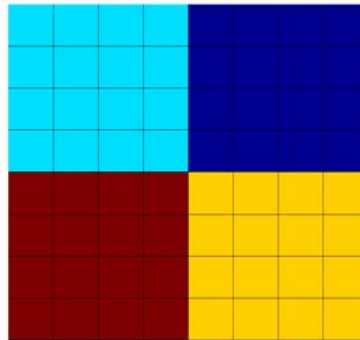
- Threads
- OpenMP
- Mutexes, semaphores, and barriers
- Atomic
- Memory fences
- Block and cyclic decompositions
- Dynamic scheduling

Workload Decomposition

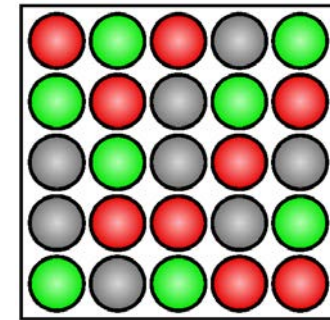
- Block vs. Cyclic
- Static vs. Dynamic Decomposition



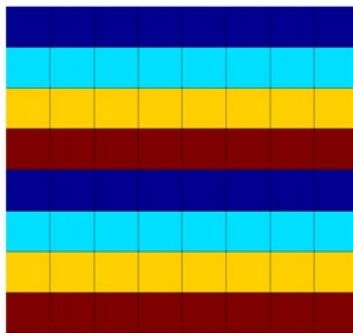
[Block, *]



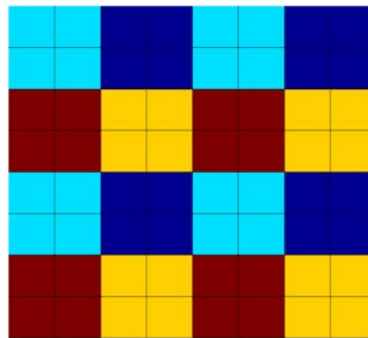
[Block, Block]



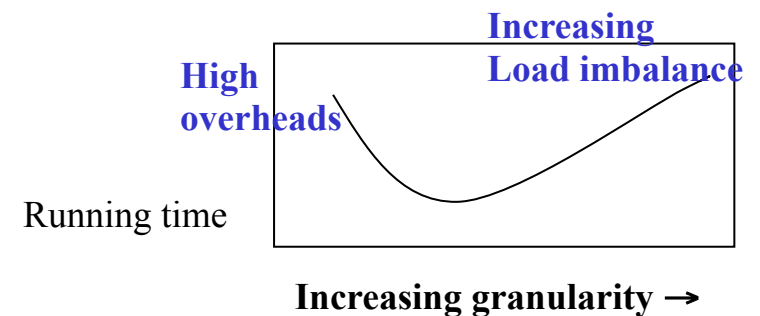
Dynamic



[Cyclic, *]



[Cyclic(2), Cyclic(2)]

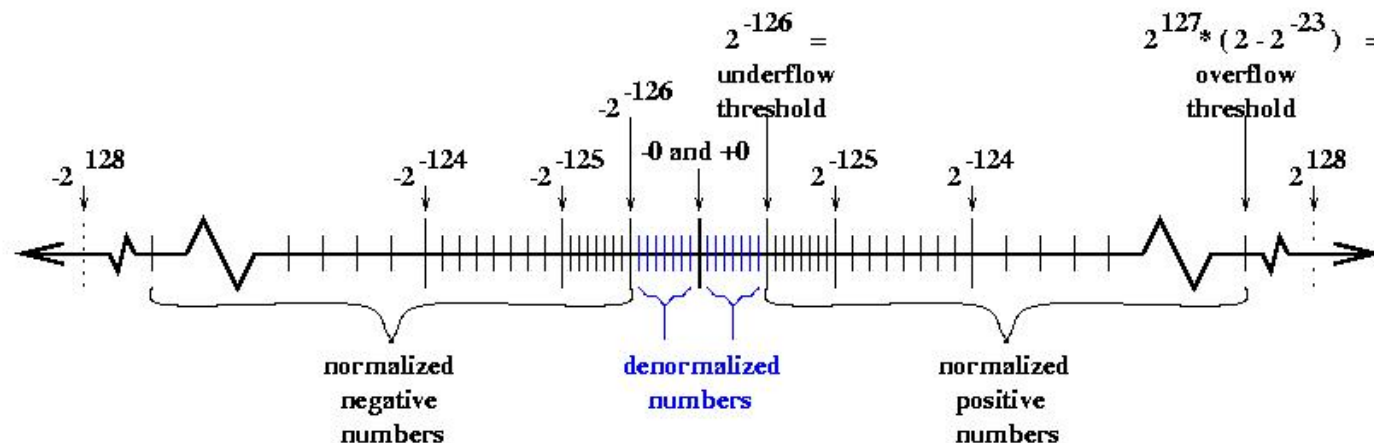


Algorithms

- Image smoother
- Sort
 - Odd-even sort
 - Bitonic
 - Merge

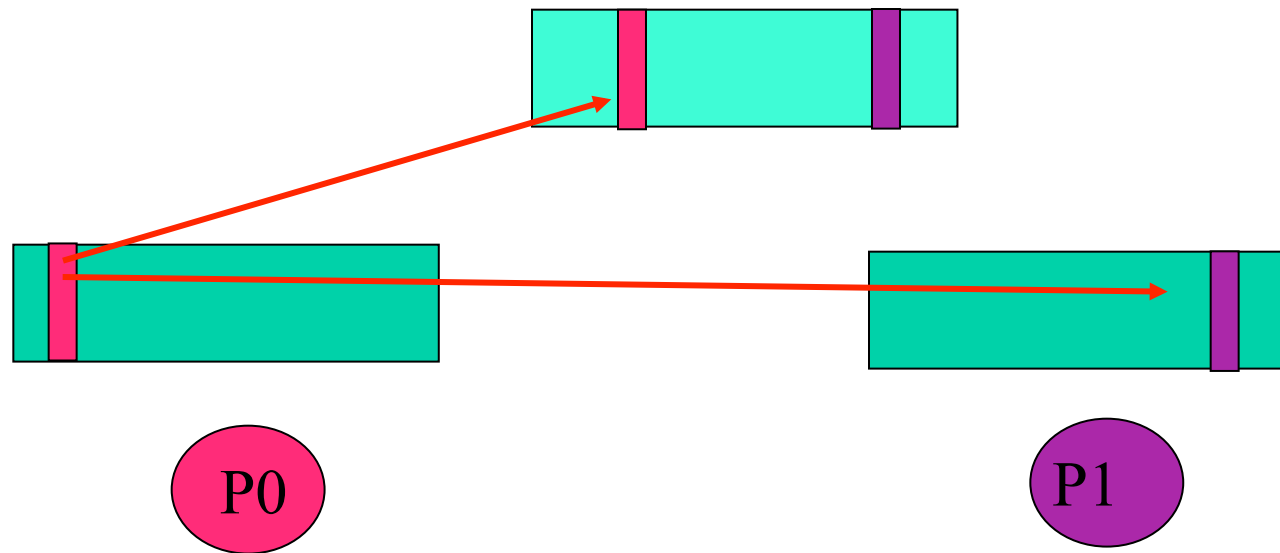
IEEE Floating point

- NaN, $\pm\infty$, signed zeros
- Round to the nearest representable floating point number that corresponds to the exact value (correct rounding)
- Error formula: $\text{fl}(a \text{ op } b) = (a \text{ op } b) * (1 + \delta) \text{ op one of } + - * /$
 - $|\delta| \leq \epsilon$, assuming no overflow, underflow, or divide by zero
- Addition example
 - $\text{fl}(\sum x_i) = \sum_{i=1:n} x_i * (1 + e_i)$
 - $|e_i| \sim (n-1)\epsilon$



False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another's cache



Eliminating false sharing

- Put each counter in its own cache line

0	1	2	3	4	5	6	7	...	31
0	1	2	3	4	5	6	7	...	31
0	1	2	3	4	5	6	7	...	31
0	1	2	3	4	5	6	7	...	31

```
static int counts[];  
for (int k = 0; k < reps; k++)  
    for (int r = first; r <= last; ++ r)  
        if ((values[r] % 2) == 1)  
            counts[TID]++;
```

```
static int counts[][LINE_SIZE];  
for (int k = 0; k < reps; k++)  
    for (int r = first; r <= last; ++ r)  
        if ((values[r] % 2) == 1)  
            counts[TID][0]++;
```

	NT=1	NT=2	NT=4	NT=8
Unoptimized	4.7 sec	6.3	7.9	10.4
Optimized	4.7	5.3	1.2	1.3

Load Balancing

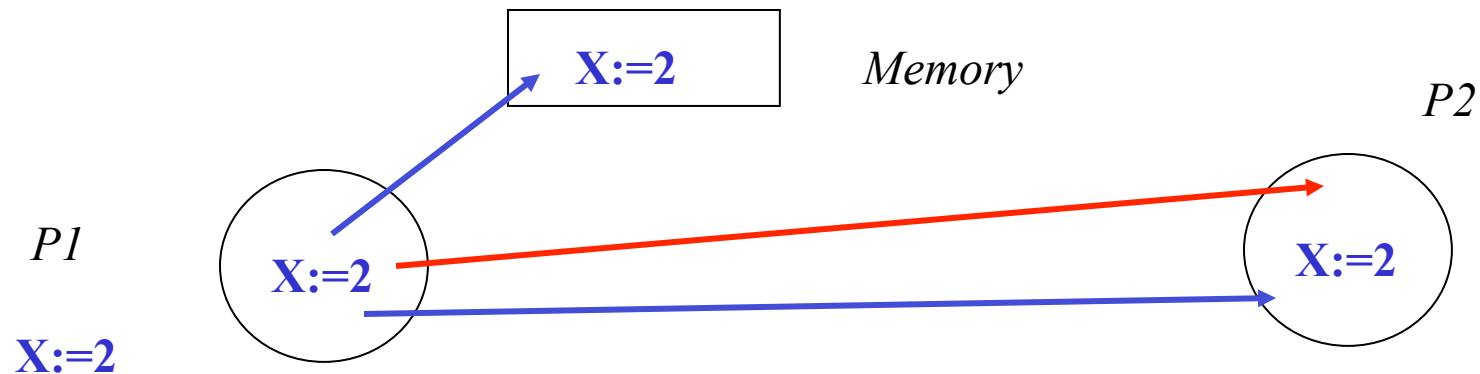
- We have an int array of length 1024, and share the computation evenly among the processors.
The workload consists of updating the array.
 - All the updates take the same amount of time, 1 second
- a. What is the running time on 4 processors?
 - b. What is the running time on 5 processors?
 - c. What is the running time on 16 processors?
 - d. Describe the workload distribution scheme(s) you used to share the workload.

Load Balancing - Part II

- Now, we use a different algorithm to update the array.
 - Some array elements take longer than others to update.
 - In order to balance the workloads, we break the array into 32 contiguous chunks.
 - 8 of these chunks complete in 10 seconds, and the rest 5 seconds.
- a. What is the best possible running time we can expect on 4 processors, using any workload distribution scheme, and ignoring overhead costs?
 - b. What is the best possible running time we can expect on 5 processors?
 - c. What is the best possible running time on 16 processors?
 - d. Describe the workload distribution scheme(s) you used to balance the workload.

Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
 - Update-on-write (implies a write-through cache)
 - Invalidate-on-write



Memory consistency

- Cache coherence tells us that memory will *eventually* be consistent
- The memory consistency policy tells us *when* this will happen
- A memory system is consistent if the following 3 conditions hold
 - Program order (you read what you wrote)
 - Definition of a coherent view of memory (“eventually”)
 - Serialization of writes (a single frame of reference)

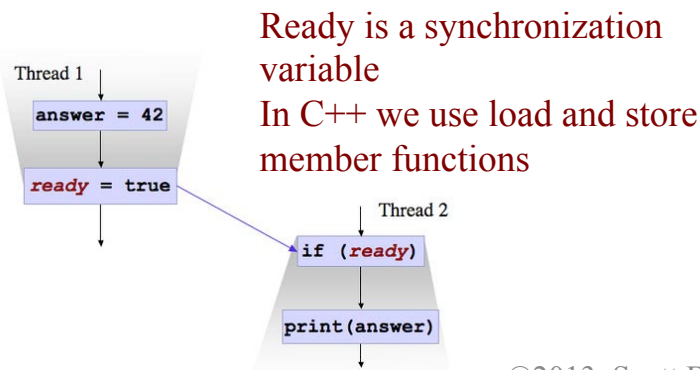
Consistency

- Assume that memory is sequentially consistent
- How many times can procedure `foo()` be run?
- Assume that both `X` and `Y` are shared **int**, and have been initialized to zero
- Too expensive to guarantee sequential consistency all the time

Thread 0	Thread 1
<code>X = 1;</code>	<code>Y = 1;</code>
<code>If (y == 0)</code>	<code>If (x == 0)</code>
<code>foo();</code>	<code>foo();</code>

The C++11 Memory model

- The C++11 memory model makes minimal guarantees about semantics of memory access, bounding effects of optimizations on execution semantics
- Special mechanisms are needed to guarantee that communication happens between threads, that establish the “happens before” relationship
- Memory writes made by one thread can become visible, but no guarantee.
- *Without explicit communication, you can't guarantee which writes get seen by other threads, or even the order in which they get seen*
- C++ atomic variables (and the Java **volatile** modifier) are a special mechanism guaranteeing that communication happens between threads
- When one thread writes to a *synchronization variable*, and another thread sees that write, the first thread is telling the second about all of the contents of memory up until it performed the write to that variable



*All the memory contents seen by T1,
before it wrote to ready,
must be visible to T2,*

after it reads the value true for ready.

<http://jeremymanson.blogspot.com/2008/11/what-volatile-means-in-java.html>

Rules

- **3 rules** that mostly concern when values must be transferred between main memory and per-thread memory
- **Atomicity.** Which instructions must have indivisible effects? Only concerned with instance and static variables, including array elements, but **not** local variables inside methods
- **Visibility.** Under what conditions the effects of one thread are visible to another? The effects of interest are: writes to variables, as seen via reads of those variables
- **Ordering.** Under what conditions the effects of operations can appear out of order to any given thread? In particular, reads and writes associated with sequences of assignment statements
- All changes made in one synchronized variable or code block are atomic and visible with respect to other synchronized variables and blocks employing the same lock, and processing of synchronized methods or blocks within any given thread is in program-specified order

Data races

- We say that a program allows a *data race* on a particular set of inputs if there is a *sequentially consistent execution*, i.e. an interleaving of operations of the individual threads, in which two conflicting operations can be executed “simultaneously” (Boehm)
- We’ll say that operations can be executed “simultaneously”, if they occur next to each other in the interleaving, and correspond to different threads
- We can guarantee sequential consistency only when the program avoids data races
- This program has a data race

Execution 3

```
x = 1;  
y = 1;  
r1 = y;  
r2 = x;  
// r1 = 1 ∧ r2 == 1
```

Thread 1

```
x = 1;  
r1 = y;
```

Thread 2

```
y = 1;  
r2 = x;
```

Using synchronization variables to ensure sequentially consistent execution

- Consider this example where `x` is of type `int`, `x_init` is of type `atomic<bool>`,
- This program is free from data races
- Thread 2 is guaranteed not to progress to the second statement until the first thread has completed and set `x_init`. There cannot be an interleaving of the steps in which the actions `x = 42` and `r1 = x` are adjacent.
- Thus, we are guaranteed a sequentially consistent execution, which guaranteeing that `r1 = 42`.
- Thus implementation must ensure arrange that
 - thread 1's assignments to `x` and `x_init` become visible to other threads in order
 - The assignment `r1 = x` operation in thread 2 cannot start until we have seen `x_init` set.
- In practice these require the compiler to obey extra constraints and to generate special code to prevent potential hardware optimizations, such as thread 1 making the new value of `x_init` available before that of `x` because it happened to be faster to access `x_init`'s memory

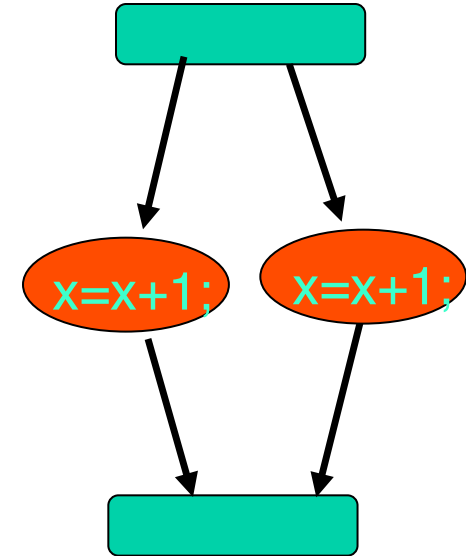
Thread 1	Thread 2
<pre>x = 42; x_ready = true;</pre>	<pre>while (!x_ready) {} r1 = x;</pre>

Visibility

- Changes to fields made by one thread are guaranteed to be visible to other threads only under the following conditions
- A writing thread releases a synchronization lock and a reading thread subsequently acquires that same
 - Releasing a lock flushes all writes from the thread's working memory, acquiring a lock forces a (re)load of the values of accessible variables
 - While lock actions provide exclusion only for the operations performed within a synchronized block, these memory effects are defined to cover all variables used by the thread performing the action
- If a variable is declared as **atomic**
 - Any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation.
 - Readers must reload the values of volatile fields upon each access.
- As a thread terminates, all written variables are flushed to main memory. Thus, if one thread synchronizes on the termination of another thread using **join**, then it is guaranteed to see the effects made by that thread

Under the hood of a race condition

- Assume x is initially 0
 $x = x + 1$;
- Generated assembly code
 - $r1 \leftarrow (x)$
 - $r1 \leftarrow r1 + \#1$
 - $r1 \rightarrow (x)$



- Possible interleaving with two threads

P1
 $r1 \leftarrow x$

$r1 \leftarrow r1 + \#1$

$x \leftarrow r1$

P2
 $r1 \leftarrow x$
 $r1 \leftarrow r1 + \#1$
 $x \leftarrow r1$

*$r1(P1)$ gets 0
 $r1(P2)$ also gets 0
 $r1(P1)$ set to 1
 $r1(P2)$ set to 1
 $P1$ writes its $R1$
 $P2$ writes its $R1$*

Avoiding race conditions

- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness
- We need to take steps to avoid race conditions through appropriate program synchronization
 - Critical sections
 - Barriers
 - Atomics

Correctness and Fairness

Each thread increments a shared variable until reaching a given maximum value
Using an atomic doesn't avoid the race condition

```
atomic<int> sharedVar;  
int* howMany;  
void summer(int TID) {  
    while(sharedVar<MAX_VAL){  
        sharedVar++;  
        howMany[TID]++;  
    } // $PUB/Examples/Threads/whoDunnit.cpp
```

} **Spawning 2 threads**

shared Var: 1024

Thread 1 made 1024 updates

Spawning 2 threads

shared Var: 1025

Thread 0 made 531 updates

Thread 1 made 494 updates

Spawning 2 threads

shared Var: 1024

Updates by thread

Thread 1 made 1024 updates

- If we run with 2 threads, both try to update the shared object
What happens to the shared var?

- If we spawn 100 threads, will all spawned threads get to update the shared variable?

- If one of not, we call that effect "starvation."

- How can we avoid starvation to ensure fairness?

Spawning 100 threads

shared Var: 1024

Thread 0 made 269 updates

Thread 1 made 484 updates

Thread 2 made 190 updates

Thread 3 made 81 updates

shared Var: 1025

Thread 0 made 4 updates

Thread 1 made 735 updates

Thread 2 made 286 updates

Spawning 100 threads

Thread 1 made 1024 updates

Multithreaded Smoother()

Global Change, $I[:,:]$, $I^{\text{new}}[:,:]$

Local mymin = 1 + (\$TID * n / \$NT),
mymax = mymin + n / \$NT - 1;

Local done = FALSE;

while (!done) do

Local myChange = 0;

Change = 0;

update I^{new} and myChange

Change += myChange;

if (Change < Tolerance) done = TRUE;

Swap pointers: $I \leftrightarrow I^{\text{new}}$

end while

update I^{new} and myChange:
for i = mymin to mymax do
 for j = 1 to n do
 $I^{\text{new}}[i,j] = \dots$
 myChange += $(I^{\text{new}} - I^{\text{old}})^2$
 end for
end for

0
1
2
3

Is this code correct?

Correctness

Global Change, $I[:,:]$, $I^{\text{new}}[:,:]$

Local $\text{mymin} = 1 + (\$TID * n / \$NT)$,

$\text{mymax} = \text{mymin} + n / \$NT - 1$;

Local $\text{done} = \text{FALSE}$;

while (!done) do

Local $\text{myChange} = 0$;

BARRIER

Only on thread 0: $\text{Change} = 0$;

BARRIER

update I^{new} and myChange

CRITICAL SEC: $\text{Change} += \text{myChange}$

BARRIER

if ($\text{myChange} < \text{Tolerance}$) $\text{done} = \text{TRUE}$;

Only on thread 0: Swap pointers: $I \leftrightarrow I^{\text{new}}$

end while

0
1
2
3

Does this code use minimal synchronization?

Fin

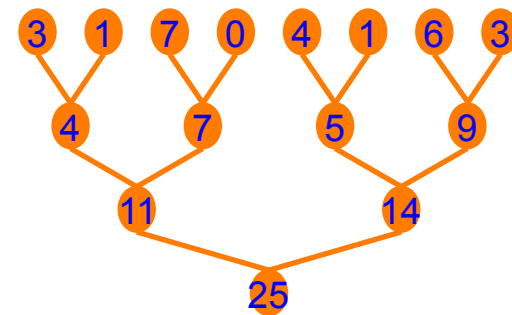
5. Tree Summation

- Input: an array $x[]$, length $N \gg P$
- Output: Sum of the elements of $x[]$
- Goal: Compute the sum in $\lg P$ time

```
sum = 0;  
for i=0 to N-1  
    sum += x[i]
```

- Assume P is a power of 2, $K = \lg P$
- Starter code
for $m = 0$ to $K-1$ {

}



Visualizing the Summation

