CSEP 524: Parallel Computation
(week 3)

Brad Chamberlain
Tuesdays 6:30 – 9:20
MGH 231
Shameless Plug

• The Chapel team is looking to fill two internship positions this summer if someone you know is interested.
What We’ve Discussed

• Why parallelism matters
• A bunch of terminology
• Ways of measuring parallel performance
• How to create/join tasks in C+Pthreads and Chapel
• Block and Cyclic work distributions
• Hopefully you’ve seen speedup firsthand by now
What’s Next?

• At a high level:
  – Discussion/Diagnosis of behavior in Assignment #1
  – Having tasks coordinate with one another
Discussion of Assignment #1
Assignment #1 Discussion

Q1: What kinds of parallel resources did you find?
   – who has highest-core count desktop?
   – what larger-scale systems are available to you?
   – what parallel programming models did you identify?

We should soon have access to a UW CSE 8x4-core VM-based platform for the class to share
Q4: What block distribution strategy did you use?
   – e.g., when dividing 10 items by 4 tasks, did you use:
     • 3 3 2 2
     • 3 2 3 2
     • 2 3 2 3
     • 3 3 3 1
     • other?
Assignment #1 Discussion

Q5: What were your predictions?
   – random vs. ramp
   – negation vs. factorial
   – block vs. cyclic
   – number of tasks

• What were the biggest surprises?

• Did you see linear speedup?
# Summary of Observations

## Block Distribution

<table>
<thead>
<tr>
<th></th>
<th>random</th>
<th>ramp</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>should be faster than factorial</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>factorial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>should be faster than ramp because</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Cyclic Distribution

<table>
<thead>
<tr>
<th></th>
<th>random</th>
<th>ramp</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>factorial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>should be faster than ramp because</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel Programming is Hard
(you may or may not agree with this sentiment yet, but it’s true)

Keep track of your war stories this quarter
• for the purposes of classroom discussion
• because misery loves company
Two Performance Gotchas
Performance Gotcha #1: Memory

Issue #1: *Competition for Memory Locations*

- any time processors have non-shared caches there is the potential for them to compete for memory locations
Performance Gotcha #1: Memory

**Issue #1: Competition for Memory Locations**

- any time processors have non-shared caches there is the potential for them to compete for memory locations
  - read-only accesses should not be an issue
  - once a task/core starts writing to a location, competition may ensue
Example: Competition For Memory

array negation using cyclic distribution:

Task 1

reg = read A[0]
reg = -reg
A[0] = write reg

Task 2

reg = read A[1]
reg = -reg
A[1] = write reg
Example: Competition For Memory

array negation using cyclic distribution:

Task 1
reg = read A[0]
reg = -reg
A[0] = write reg

Task 2
reg = read A[1]
reg = -reg
A[1] = write reg
**Example: Competition For Memory**

- **Task 1**
  - `reg = read A[0]`
  - `reg = -reg`
  - `A[0] = write reg`

- **Task 2**
  - `reg = read A[1]`
  - `reg = -reg`
**Definition: False Sharing**

*False Sharing:* When cache lines must be invalidated not because two tasks are accessing the same data, but because they’re accessing data on the same cache line

– in reality, the data is truly independent, hence “false”
– the details of the granularity at which data is stored within HW is what causes the interdependence (“sharing”)
– **NOTE:** On cache coherent architectures, this is a performance issue, not a correctness issue

– (“true sharing” might be considered when two tasks actually access the same shared variable/data)
False Sharing Implications for Assignment #1

• Writing to an array using a cyclic distribution can result in performance impacts due to false sharing
  – possible fixes:
    • have each task 0 start its cyclic iteration from a skewed position
      – e.g., have task \( t \) starts from element \( t + t\times n/p \)
      – but, results in more complex loop idioms due to need to wrap around
    • use padding/alignment pragmas to spread out array data
      – but, results in wasted space
Issue #2: *Memory is a bottleneck*

- typically, processors increase in speed faster than memory
- having multiple processors share memory doesn’t help
  - there are only so many wires to access memory
  - cache coherence protocols also add overhead/complexity
Performance Gotcha #1: Memory

Issue #2: Memory is a bottleneck

– algorithms with more computational intensity can better amortize these memory overheads
Definition: Computational Intensity

Computational Intensity:
Definition: Computational Intensity

**Computational Intensity:** How much computation is performed per memory access

- high computational intensity: lots of OPS per load/store
  => memory performance is less of an issue
- low computational intensity: few OPS per load/store
  => memory performance is more of an issue
Mem. Performance Implications for Assignment #1

• Computations with greater computational intensity should result in better speedup
  – e.g., factorial should speed up better than negation
Performance Gotcha #2: Load Balance

Negation + Ramp: Computational Intensity per Element
Performance Gotcha #2: Load Balance

Negation + Ramp: Computational Intensity per Element
– Block distribution: green and purple have ~the same work

array elements

computational intensity
Performance Gotcha #2: Load Balance

Factorial + Ramp: Computational Intensity per Element
Performance Gotcha #2: Load Balance

Factorial + Ramp: Computational Intensity per Element
– Block Distribution

array elements

computational intensity
Performance Gotcha #2: Load Balance

Factorial + Ramp: Computational Intensity per Element
– Block Distribution: Purple has ~3x as much work as green
Performance Gotcha #2: Load Balance

Factorial + Ramp: Computational Intensity per Element
– Cyclic Distribution
Performance Gotcha #2: Load Balance

Factorial + Ramp: Computational Intensity per Element
– Cyclic Distribution: Purple only has numItems/2 more work
Performance Gotcha #2: Load Balance

Factorial + Random:

– Block distribution: green has ~1.5x the work of purple
  • (for the data set shown)
Load Balance Implications for Assignment #1

• Block + factorial + ramp exhibits bad *load balance*
  – some tasks had significantly more work than others
  – cyclic/random input sets may result in better load balance

• Keep in mind that many algorithms must be written without knowing their input sets
  – i.e., can’t think “aha, my input will be a ramp so ...”
Assignment #1 Debrief

• Who saw execution time behaviors similar to what I just described?
  – what kinds of things did you “do right” to get this result?
  – what kinds of issues did others do differently to not see it?
  – or perhaps, rather, what did you stumble across then fix?
    • measuring aggregate performance of all threads, not wallclock time
Assignment #1 Summary: Distributions

Block & Cyclic:

+ give each task a similar number of work items
+ reasonably easy to compute

Block:

+ results in good spatial locality (touches adjacent elements)
  – can expose sensitivities to work distribution
    • as in ramp+factorial

Cyclic:

+ less likely to be sensitive to work distribution
  – can result in false sharing issues
Time for a Break/Something Different?
Alternatives to Block and Cyclic

• Other distributions can help address the drawbacks of block and cyclic:
  – Block-Cyclic distribution
  – Dynamic distributions
  – Algorithmically-aware distributions
Distribution #3: Block-Cyclic Distribution

• As the name suggests, a hybrid of Block and Cyclic
  – deals blocks of items out cyclically
    • parameterized by block size, $b$
    • ideally, $b$ should match or exceed cache line size
    • optimal choice of $b$ often depends on algorithm, working set size, ...

  – tradeoffs:
    + gives tasks chunks of work (good spatial locality; less false sharing)
    + like cyclic, results in probabilistically-oriented load balancing
    – results in slightly more complicated loop nests
Dynamic Distributions

Concept:
– don’t deal work out according to a fixed, *a priori* schedule
– instead, deal work out to tasks (or have them grab it) as they become idle

Goal:
– no task gets stuck with more work than it can handle

Challenge:
– what granularity (granularities?) to deal out work?
  • *if too large*: tasks may get unlucky and stuck with too much work
  • *if too small*: too much effort coordinating, not enough computing
Algorithmically-Aware Distributions

Concept:

– For some algorithms, there may be a way to scan the input data in order to compute a good distribution
  • e.g., dynamically sample the input data set to try and predict trends?
  • e.g., examine the placement of zeroes and non-zeroes in a sparse matrix?
  • e.g., compute a dependence graph for the computation and distribute it using a graph partitioning algorithm

Goal:

– use algorithmic-centric knowledge to improve load balance

Challenge:

– Cost:Benefit ratio needs to be taken into account
  • since any overhead in computing a distribution is new work that wouldn’t have been required in a serial version
Multidimensional Distributions

• So far, we’ve looked solely at 1D distributions
• Distributions can also be multidimensional
  – one option is to apply a 1D distribution per dimension
2D Block x Block
(distributed to 2x2 tasks)
2D Block x Block
(distributed to 1x4 tasks)
2D Block x Block
(distributed to 4x1 tasks)
2D Block-Cyclic x Block-Cyclic (distributed to 2x2 tasks)
...and so on and so forth

- Cyclic x Cyclic
- Block x Cyclic
- Cyclic x Block
- Block-Cyclic x Block-Cyclic with different block sizes
- Block-Cyclic x Block
- Block x Block-Cyclic
- etc.
Q: In a Shared-Memory setting, which would you use from the perspective of memory?
Multidimensional Distributions

• So far, we’ve primarily looked at 1D distributions
• Distributions can also be multidimensional
  – one option is to apply a 1D distribution per dimension
  – another is to distribute the items holistically
Holistic Distribution: Recursive Bisection

Note: Can’t be expressed as the conflation of two 1D distributions
Multidimensional Distributions

- So far, we’ve primarily looked at 1D distributions
- Distributions can also be multidimensional
  - one option is to apply a 1D distribution per dimension
  - another is to distribute the items holistically

- Or, even unstructured (e.g., distribute a graph)
  - a topic for another day
Measuring Load Imbalance

• In assignment #1, we used the following pattern to measure the overall execution time of the code:

```cpp
var totTime, maxTime = 0.0;
var minTime = max(real);
start timer
create tasks
do work
join tasks
check timer
```

This essentially measured 
$$\max(\time_{\text{purple}}, \time_{\text{green}})$$
Measuring Load Imbalance

• Imagine instead, pushing the timing into the loop:

```
var totTime, maxTime = 0.0;
var minTime = max(real);
create tasks
start timer
do work
check timer
join tasks
```

This permits us to measure $t_{\text{purple}}$ and $t_{\text{green}}$ distinctly.
Measuring Load Imbalance

• Now, we can compute statistics on a task-by-task basis:

```plaintext
var totTime, maxTime = 0.0;
var minTime = max(real);
coforall tid in 0..#numTasks {
    start timer
do work
    const myTime = check timer
totTime += myTime;
    if myTime < minTime then minTime = myTime;
    if myTime > maxTime then maxTime = myTime;
}
const avgTime = totTime / numTasks;
```

What’s the bug in this code?
Bug of the week

• The previous slide contains a classic bug
  – Code that looks innocuous is actually problematic
  – Cause: reading parallel code as though it were sequential

```c
coforall tid in 0..#numTasks {
    ...
    totTime += myTime;
    ...
}
```

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg = read totTime</td>
<td></td>
</tr>
<tr>
<td>reg = reg + myTime</td>
<td></td>
</tr>
<tr>
<td>totTime = write reg</td>
<td></td>
</tr>
<tr>
<td>reg = read totTime</td>
<td></td>
</tr>
<tr>
<td>reg = reg + myTime</td>
<td></td>
</tr>
<tr>
<td>totTime = write reg</td>
<td></td>
</tr>
</tbody>
</table>
Bug of the week

• Whether or not this bug exhibits itself depends on the scheduling of the tasks
  – the following schedule would be fine:

  Task 1
  reg = read totTime
  reg = reg + myTime
  totTime = write reg

  Task 2
  reg = read totTime
  reg = reg + myTime
  totTime = write reg
Bug of the week

• Whether or not this bug exhibits itself depends on the scheduling of the tasks
  – the following schedule is problematic:

  **Task 1**
  \[
  \text{reg} = \text{read totTime} \\
  \text{reg} = \text{reg} + \text{myTime} \\
  \text{totTime} = \text{write reg}
  \]

  **Task 2**
  \[
  \text{reg} = \text{read totTime} \\
  \text{reg} = \text{reg} + \text{myTime} \\
  \text{totTime} = \text{write reg}
  \]

  \[
  \begin{array}{c}
  3.7 \, \text{myTime} \\
  \hline
  \text{reg} \\
  0.0 \, \text{totTime}
  \end{array}
  \]

  \[
  \begin{array}{c}
  2.3 \, \text{myTime} \\
  \hline
  \text{reg}
  \end{array}
  \]
Bug of the week

• Whether or not this bug exhibits itself depends on the scheduling of the tasks
  – the following schedule is problematic:
Bug of the week

• Whether or not this bug exhibits itself depends on the scheduling of the tasks
  – the following schedule is problematic:

 Task 1
 reg = read totTime
 reg = reg + myTime
 totTime = write reg

 Task 2
 reg = read totTime
 reg = reg + myTime
 totTime = write reg

<table>
<thead>
<tr>
<th>Time</th>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>myTime</td>
<td>3.7</td>
<td>2.3</td>
</tr>
<tr>
<td>totTime</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Bug of the week

• Whether or not this bug exhibits itself depends on the scheduling of the tasks
  – the following schedule is problematic:

```
Task 1
reg = read totTime
reg = reg + myTime
totTime = write reg
```

```
Task 2
reg = read totTime
reg = reg + myTime
totTime = write reg
```

```
3.7 myTime
3.7 reg
```

```
2.3 myTime
0.0 reg
```

```
0.0 totTime
```
Bug of the week

• Whether or not this bug exhibits itself depends on the scheduling of the tasks
  – the following schedule is problematic:
Bug of the week

• Whether or not this bug exhibits itself depends on the scheduling of the tasks
  – the following schedule is problematic:

```plaintext
Task 1
reg = read totTime
reg = reg + myTime
totTime = write reg

Task 2
reg = read totTime
reg = reg + myTime
totTime = write reg
```

3.7 myTime
3.7 reg

3.7 totTime

2.3 myTime
2.3 reg

3.7

2.3
Bug of the week

• Whether or not this bug exhibits itself depends on the scheduling of the tasks
  – the following schedule is problematic:

```
Task 1
reg = read totTime
reg = reg + myTime
totTime = write reg

Task 2
reg = read totTime
reg = reg + myTime
totTime = write reg
```

3.7 myTime
3.7 reg

2.3 totTime

2.3 myTime
2.3 reg
Bug of the week: RRWW (Read-Read-Write-Write)

• Due to interleaving, uncoordinated reads and writes to shared state may cause values to be lost
• The fix is to coordinate such accesses to shared state
  – in this case, totTime, minTime, maxTime
  – e.g., could protect each/all of them by a lock

\[
\text{coforall tid in 0..#numTasks} \\
... \\
grab \text{ totTime lock} \\
\text{ totTime += myTime;} \\
\text{ release totTime lock} \\
... 
\]
Glossary: Synchronization

Synchronization:
Glossary: Synchronization

*Synchronization*: Coordination between tasks
Synchronization Mechanisms in Pthreads

1) mutex: “mutual exclusion” – essentially a lock
   – operations:
     • init, destroy: create and destroy them
     • lock, unlock: grab and release the lock
     • trylock: attempt to grab the lock, but don’t block if you can’t
Synchronization Mechanisms in Pthreads

2) *condition variables*: a “waiting room” for some condition to become true

- operations:
  - *init, destroy*: create and destroy them
  - *wait*: wait for a condition to become true
  - *signal/broadcast*: signal to one/multiple thread(s) that it is

- rationale: avoid spinning on some test in user code
  - e.g., “wait for this variable to take on some nonzero value”
  - such spinning is typically not a wise use of resources
  - *instead*: let the thread library manage who should wake up when
Condition Variables: Fiddly Details

There are some details that complicate condition vars:

- **mutex argument**: must be managed properly
- **spurious wakeups**: verifying that the condition is still true once you’ve awoken from a wait()

See Ch. 6 of the text and/or this tutorial for details:

- [https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables](https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables)
Using Mutexes to fix RRWW bugs

```c
pthread_mutex_t totTimeMutex;
pthread_mutex_init(&totTimeMutex, NULL);

create tasks
...
pthread_mutex_lock(&totTimeMutex);
totTime += myTime;
pthread_mutex_unlock(&totTimeMutex);
...
join tasks

pthread_mutex_destroy(&totTimeMutex);
```
Using Mutexes to fix RRWW bugs

The result is that there are only two legal orderings of the totTime updates:

- **Task 1 grabs the mutex first**
  - Task 1: mutex lock
    - reg = read totTime
    - reg = reg + myTime
    - totTime = write reg
  - mutex unlock
  - (blocks...)
  - reg = read totTime
  - reg = reg + myTime
  - totTime = write reg
  - mutex unlock

- **Task 2 grabs the mutex first**
  - Task 2: mutex lock
    - reg = read totTime
    - reg = reg + myTime
    - totTime = write reg
  - mutex unlock
  - (blocks...)
  - reg = read totTime
  - reg = reg + myTime
  - totTime = write reg
  - mutex unlock

@me task 1 grabs the mutex first

@me task 2 grabs the mutex first
Synchronization Mechanisms in Chapel

1) synchronization variables
2) single-assignment variables
Synchronization Variables

- **Syntax**
  
  ```chapel
sync-type:
  sync type
  ```

- **Semantics**
  - Stores *full/empty* state along with normal value
  - **Defaults to full** if initialized, *empty* otherwise
  - Default read blocks until *full*, leaves *empty*
  - Default write blocks until *empty*, leaves *full*

- **Examples: Critical sections and futures**

  ```chapel
  var lock$: sync bool;
  lock$ = true;
  critical();
  var lockval = lock$;
  ```

  ```chapel
  var future$: sync real;
  begin future$ = compute();
  computeSomethingElse();
  useComputedResults(future$);
  ```
Example: Bounded Buffer Producer/Consumer

```chapel
var buff$: [0..#buffersize] sync real;

cobegin {
    producer();
    consumer();
}

proc producer() {
    var i = 0;
    for ... {
        i = (i+1) % buffersize;
        buff$[i] = ...; // blocks until empty, leaves full
    }
}

proc consumer() {
    var i = 0;
    while ... {
        i= (i+1) % buffersize;
        ...buff$[i]...; // blocks until full, leaves empty
    }
}
```
Single-Assignment Variables

- **Syntax**

  ```
  single-type: single type
  ```

- **Semantics**
  - Similar to sync variables, but stays *full* once written

- **Example: Multiple Consumers of a future**

  ```
  var future$: single real;

  begin future$ = compute();
  computeSomethingElse(future$);
  computeSomethingElse(future$);
  ```
Synchronization Type Methods

- **readFE(): t**  
  block until `full`, leave `empty`, return value
- **readFF(): t**  
  block until `full`, leave `full`, return value
- **readXX(): t**  
  return value (non-blocking)
- **writeEF(v:t)**  
  block until `empty`, set value to `v`, leave `full`
- **writeFF(v:t)**  
  wait until `full`, set value to `v`, leave `full`
- **writeXF(v:t)**  
  set value to `v`, leave `full` (non-blocking)
- **reset()**  
  reset value, leave `empty` (non-blocking)
- **isFull: bool**  
  return `true` if full else `false` (non-blocking)

- **Defaults:** read: **readFE**, write: **writeEF**
Single Type Methods

- **readFE():** block until full, leave empty, return value
- **readFF():** block until full, leave full, return value
- **readXX():** return value (non-blocking)
- **writeEF(v:t):** block until empty, set value to v, leave full
- **writeFF(v:t):** wait until full, set value to v, leave full
- **writeXF(v:t):** set value to v, leave full (non-blocking)
- **reset():** reset value, leave empty (non-blocking)
- **isFull: bool** return true if full else false (non-blocking)

- **Defaults:** read: readFF, write: writeEF
Using Sync vars to fix RRWW bugs

```plaintext
var totTime$: sync real = 0.0;  // starts full

coforall tid in 0..#numTasks {
    ...
    totTime$ += myTime;  // readFE followed by writeEF
    ...
}
```
Summary: Pthreads vs. Chapel Synchronization

Pthreads mutex & condition variables:

+ arguably a reasonable backbone for synchronization
  • based on the endurance of Pthreads
  • use of these concepts in other languages/contexts

– arguably result in complex code for common patterns

Chapel sync/single variables:

+ *data-centric synchronization*: expressing synchronization in terms of the data being accessed

– arguably a little artificial/confusing when used as a mutex
  • e.g., see unused boolean value in previous critical section example

Both approaches also have some common liabilities (stay tuned)
Diagnosing Deadlock/Livelock in Chapel

• If you suspect you have a deadlock problem...
  – re-execute your program using --blockreport
    • adds a certain amount of overhead, but beats deadlocking!
  – if deadlock is detected, the program will...
    • terminate
    • do its best to tell you where the tasks were

• If you suspect you have a livelock problem...
  – re-execute your program using --taskreport
    • again, adds a certain amount of overhead
  – upon hitting Ctrl-C/sending SIGINT, the program will...
    • terminate and do its best to tell you where the tasks are
This week’s assignment

• extend the single-producer/single-consumer bounded buffer pattern shown in lecture to support multiple producers and consumers
  – in Chapel (to get practice with sync/single variables)
  – in Pthreads (to get practice with mutex/condition variables)

• write a dynamic load balancing distribution in Chapel OR Pthreads
  – apply to ramp + factorial case

• some written questions