Lecture 26:

NT Method +
Course Review

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2014
Limited range force computation

Goal: compute interactions between all particles located within distance $R$

Common strategy:
Partition space into $P$ regions (one region per processor)

All particles in gray box are “owned” by the same processor.

Typical assignment decision: processor owning particle $A$ is responsible for computing interactions between $A$ and $B$
Limited range force computation

Goal: compute interactions between all particles located within distance \( R \)

Consider two sets of particles:

- "Home" region for processor \( P \): \( \sim b^2 \) (particles in gray box, "owned" by processor \( P \))

- "Import" region for processor \( P \): \( \sim 4bR + \pi R^2 \) (space containing particles that must be communicated to processor \( P \) in order to compute forces on particles in \( P \)'s home region)

Number of interactions carried out by processor \( P \) is proportional to product of the two terms:

\( \sim b^2(4bR + \pi R^2) \)
Limited range force computation: scaling
Communication-to-computation ratio grows as $P$ increases

Consider communication-to-computation ratio as $P \to \infty$:
Note: $b \sim 1/\sqrt{P}$

Import region shrinks to $\pi R^2$
Home region shrinks to 0

So communication-to-computation ratio $\to \infty$
“Neutral territory” (NT) methods

Instead of computing interactions in the “home” node for the particles in the home region (and communicating the import region)…

Perform computation in a node that is **neither** the home node for the home region particles or the import region particles!
Neutral territory approach (in 2D)

Particle A interacts with particle B in square with X coord equal to that of A and Y coord equal to that of B. (yellow square)

Import region of the yellow square associated with processor P is blue highlighted ROW and highlighted COLUMN.

Size of import region = $4bR + b^2$

Size of import region as $P \rightarrow \infty = 0$
Neutral territory intuition

Assuming $R > b$ (true for high processor count), most particle interactions computed by processor upon which neither particle resides!

Intuition: In the NT method, the two sets of interacting particles (column and row) are the same size. Recall number of interactions is the product of these sizes.

Analogy: area of a square is greater than that of any other quadrilateral with the same perimeter.
Extending to 3D

- Divide space into $b_x \times b_y \times b_z$ cells
- Want to avoid computing interaction between each particle pair twice
- Half-shell interaction rules for particles A and B (applied in this order):
  - If $A_x < B_x$ interact in home box of A
  - If $A_y < B_y$ interact in home box of A
  - If $A_z < B_z$ interact in home box of A
  - If A and B are in the same box, no movement is necessary for interaction.
NT intuition in 2D

Half-Shell Method

Perform computation in box with smaller $X$ coordinate

Neutral-Territory Method

Perform computation in box with $X$ coordinate from particle with smaller $X$ coordinate and $Z$ coordinate from particle with larger $X$ coordinate

- Green = force computation performed by processor responsible for this region of space
- Light blue = import region for processor corresponding to green region
Extending NT method to 3D

- Divide space into $b_x \times b_y \times b_z$ cells

- NT interaction rules for particles A and B (applied in this order):
  - If $A_x < B_x$ A is in the tower
  - If $A_y < B_y$ A is in the tower
  - If $A_z < B_z$ A is in the plate
  - If A and B are in the same box, choose arbitrarily

- Two particles will interact in box with $x, y$ coordinate of tower and $z$ coordinate of plate
Scaling

Number of Processors

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>512</th>
<th>4K</th>
<th>32K</th>
</tr>
</thead>
<tbody>
<tr>
<td>HS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NT method asymptotics (3D)

- **Half-shell method**
  
  - Import volume $V = 3Rb^2 + (3/2)\pi R^2b + (2/3)\pi R^2$
  
  - As $P \to \infty$: $V = (2/3)\pi R^2$

- **NT-method**
  
  - Import volume $V = 2Rb_{xy}^2 + 2Rb_{z}^2 + (1/2)\pi R^2b_{z}$
  
  - As $P \to \infty$: $V = O(R^{3/2} / p^{1/2})$

Note: $b \sim$ cube root of $p$
NT method summary

- In N-body problems, communication-to-computation ratio increases as number of processors gets large

- Surprising solution: reduce communication requirements by always communicating
  - Key idea: a “neutral processor” performs computation of interaction forces between two particles

- In practice, the NT method wins only when $P$ is very large
Course Review
Exam details (same as exam 2)

- Closed book, closed laptop
- One “post it” of notes (but we’ll let you use both sides)
- **Focuses** on material from lecture 14 (Scaling a Web Site) onward, but you should still know major concepts from the first half of course
  - Yes (from first half): SIMD, multi-threading, latency/bandwidth/throughput, scheduling parallel programs, etc.
  - No (from first half): details of cache coherence, memory consistency models
  - No: details of actual domain-specific languages discussed in class

- **My office hours are:**
  - Saturday 3-4:30pm, or by appointment

- **The TAs will lead a review session on Sunday at 3pm in Rashid**
  - Please come with questions
Consider a multi-threaded web server
Consider a tasking system for a modern multi-core machine

- Tasking system supports bulk launch of N tasks
  - Think: about how OMP parallel for, ISPC launch, of Cilk might be implemented

Example: Eight thread worker pool for my quad-core laptop with Hyper-Threading

```c
while (work_exists()) {
  work = get_new_work();
  work.run();
}
```
Describe the components of this web site architecture (why might they exist?)

- Requests
- Perf. Monitor
- Load Balancer
- Web Server
- Database (potentially multiple machines)
  - DB Slave 1
  - DB Slave 2
  - Master
- memcached servers
  - value = get(key)
  - put(key, value)
What is the motivation for designing lock-free data structures?

What is the ABA problem?
Lock-free stack (first try)

```c
struct Node {
  Node* next;
  int value;
};

struct Stack {
  Node* top;
};

void init(Stack* s) {
  s->top = NULL;
}

void push(Stack* s, Node* n) {
  while (1) {
    Node* old_top = s->top;
    n->next = old_top;
    if (compare_and_swap(&s->top, old_top, n) == old_top)
      return;
  }
}

Node* pop(Stack* s) {
  while (1) {
    Node* old_top = s->top;
    if (old_top == NULL)
      return NULL;
    Node* new_top = old_top->next;
    if (compare_and_swap(&s->top, old_top, new_top) == old_top)
      return old_top;
  }
}
```

Main idea: as long as no other thread has modified the stack, a thread’s modification can proceed.

Note difference from fine-grained locks example earlier: before, implementation locked a part of a data-structure for fine-grained access. Here, threads do not hold lock on data-structure at all.
The ABA problem

Thread 0

begin pop()  (local variable: old_top = A, new_top = B)

begin push(A)

CAS succeeds (sets top to B!)
complete pop()  (returns A)

Thread 1

begin pop()  (local variable old_top == A)
complete pop()  (returns A)

begin push(D)
complete push(D)

modify A: e.g., set value = 42
begin push(A)
complete push(A)

Stack structure is corrupted! (lost D)
Lock-free stack using counter for ABA soln

struct Node {
    Node* next;
    int value;
};

struct Stack {
    Node* top;
    int pop_count;
};

void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, Node* n) {
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

Node* pop(Stack* s) {
    while (1) {
        int pop_count = s->pop_count;
        Node* top = s->top;
        if (top == NULL)
            return NULL;
        Node* new_top = top->next;
        if (double_compare_and_swap(&s->top, top, new_top,
                                     &s->pop_count, pop_count, pop_count+1))
            return top;
    }
}

- Maintain counter of pop operations (technically a counter wrap-around issue still exists)
- Requires machine to support “double compare and swap” (DCAS) or doubleword CAS
- Could also solve ABA problem with node allocation and/or element reuse policies

* Still assuming a sequentially consistent memory system for simplicity

test to see if either have changed (in this example: return true if no changes)
What problems could occur in the previous lock-free code on a relaxed memory consistency system that reorders writes?
What are the semantics of these Cilk primitives?

cilk_spawn foo(args);

Semantics: invoke foo, but unlike standard function call, caller may continue executing asynchronously with execution of foo.

cilk_sync;

Semantics: returns when all calls spawned by current function have completed. ("sync up" with the spawned calls)

Note: there is an implicit cilk_sync at the end of every function that contains cilk_spawn (implication: when a Cilk function returns, all work associated with that function is complete)
Why do we have queues in a parallel system?

To accommodate variable (unpredictable) rates of production and consumption. As long as A and B, on average, produce and consume at the same rate, both workers can run at full rate.

No queue: stalls exist

<table>
<thead>
<tr>
<th>Time</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

With queue of size 2: A and B never stall

<table>
<thead>
<tr>
<th>Time</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Size of queue when A completes a piece of work (or B begins work)
Heterogeneous processing

Example:
Let $n=16$
One core: $r = 4$
12 cores: $r = 1$

$\text{perf}(r) \sim \sqrt{r}$
Speedup (relative to n=1)

X-axis for symmetric architectures gives $r$ for all cores (many small cores to left)

X-axis for asymmetric architectures gives $r$ for the single “fat” core (rest of cores are $r = 1$)

[Source: Hill and Marty 08]
Limits on chip power consumption

- General rule: the longer a task runs the less power it can use
  - Processor’s power consumption (think: performance) is limited by heat generated (efficiency is required for more than just maximizing battery life)

- Electrical limit: max power that can be supplied to chip
- Die temp: (junction temp -- Tj): chip becomes unreliable above this temp (chip can run at high power for short period of time before chip heats to Tj)
- Case temp: mobile device gets too hot for user to comfortably hold (chip is at suitable operating temp, but heat is dissipating into case)
- Battery life: chip and case are cool, but want to reduce power consumption to sustain long battery life for given task

Slide credit: adopted from original slide from M. Shebanow

- iPhone 5 battery: 5.4 watt-hours
- 4th gen iPad battery: 42.5 watt-hours
- 15in Macbook Pro: 95 watt-hours
Yinzer deadlock

Non-technical side note for car-owning students: Deadlock happens in Pittsburgh all the %$***## time

(However, deadlock can be amusing when a bus driver decides to let another driver know he has caused deadlock... “go take 418 you fool!”)
Deadlock due to full queues

Assume buffers are sized so that max queue size is one message.

Outgoing read request (initiated by this processor)

Incoming read request (due to another cache) **

Both requests generate responses that require space in the other queue (circular dependency)

** will only occur if L1 is write back
Flow control using packets

- A packet consists of:
  - Header: containing routing and control information
  - Payload: HW-specific data
  - Error code
    - Generally located at tail of packet so it can be generated on the way out (sender computes checksum, appends it to end of packet)