Multicore Programming

Queues, memory, and the ABA problem

22 Nov 2010

Peter Sewell   Jaroslav Ševčík   Tim Harris
Fine-grain parallel tasks

Work stealing queues & ABA

Memory management
A really bad way to compute fib(n)

```c
int fib(int x) {
    if (x<2) {
        return x;
    } else {
        int f1 = fib(x-1);
        int f2 = fib(x-2);
        return f1+f2;
    }
}
```

- **Base case for x<2**
- **Recursion for x>=2**
A really bad way to compute fib(n)

```c
int fib(int x) {
    if (x<2) {
        return x;
    } else {
        int f1 = spawn fib(x-1);
        int f2 = spawn fib(x-2);
        sync;
        return f1+f2;
    }
}
```
Recursive fib(n)

T_\infty = 4
T_1 = 9
Recursive fib(n)
Work stealing implementation
Limit study

Speedup achieved

Work size threshold (100, 1k, 10k... 10M cycles)
Measured performance

Graph showing measured performance for various benchmarks across different numbers of cores (1, 2, 3, 4). The y-axis represents performance ratio, and the x-axis lists the benchmarks: atom, boyer, bsort-1, bsort-2, cacheprof, calendar, circsin, clasify, compress, fft2, fibheaps, hidden, lcsl, multiplier, para, primetest, write, scs, simple, sphere.
Getting good performance

• Amdahl’s law
• Granularity of work items
• Overheads of the stealing mechanisms
• Interference between items
Fine-grain parallel tasks

Work stealing queues & ABA

Memory management
Work stealing queues

1. Semantics relaxed for “PopTop”

2. Restriction: only one thread ever calls “Push/PopBottom”

3. Implementation costs skewed toward “PopTop” complex
**Bounded deque**

“Top” has a version number, updated atomically with it.

Items between the indices are present in the queue.

“Bottom” is a normal integer, updated only by the local end of the queue.
Bounded deque

```c
void PushBottom(Item i){
    tasks[bottom] = i;
    bottom++;
}
```
void PushBottom(Item i) {
    tasks[bottom] = i;
    bottom++;
}

Item popBottom() {
    if (bottom == 0) return null;
    bottom--;
    result = tasks[bottom];
    <tmp_top,tmp_v> = <top,version>;
    if (bottom > tmp_top) return result;
    ....
    return null;
}
Bounded deque

void PushBottom(Item i) {
    tasks[bottom] = i;
    bottom++;
}

Item popBottom() {
    if (bottom == 0) return null;
    bottom--;
    result = tasks[bottom];
    <tmp_top,tmp_v> = <top,version>;
    if (bottom > tmp_top) return result;
    return null;
}

if (bottom == top) {
    bottom = 0;
    if (CAS(&<top,version>,
             <tmp_top,tmp_v>,
             <tmp_top+1, v+1>)) {
        return result;
    }
}

Item PopTop() {
    if (bottom <= top) return null;
    <tmp_top,tmp_v> = <top, version>;
    result = tasks[tmp_top];
    if (CAS(&<top,version>,
            <tmp_top, tmp_v>,
            <tmp_top+1, v+1>)) {
        return result;
    }
    return null;
}
Bounded deque

void PushBottom(Item i) {
    tasks[bottom] = i;
    bottom++;
}

Item popBottom() {
    if (bottom == 0) return null;
    bottom--;
    result = tasks[bottom];
    <tmp_top,tmp_v> = <top,version>;
    if (bottom > tmp_top) return result;
    return null;
}

if (bottom == top) {
    bottom = 0;
    if (CAS(&<top,version>,
        tmp_top, tmp_v,
        <0,v+1>)) {
        return result;
    }
}

Item PopTop() {
    if (bottom <= top) return null;
    <tmp_top,tmp_v> = <top, version>;
    result = tasks[tmp_top];
    if (CAS(&<top,version>,
        tmp_top, tmp_v,
        <tmp_top+1, v+1>)) {
        return result;
    }
    return null;
}
Item PopTop() {
  if (bottom <= top) return null;
  tmp_top = top;
  result = tasks[tmp_top];
  if (CAS(&top, top, top+1)) {
    return result;
  }
  return null;
}
General techniques

• Local operations designed to avoid CAS
  – Traditionally slower, less so now
• Local operations just use read and write
  – Only one accessor, check for interference
• Use CAS:
  – Resolve conflicts between stealers
  – Resolve local/stealer conflicts
  – Version number to ensure conflicts seen
Fine-grain parallel tasks
Work stealing queues & ABA
Memory management
Lock-free data structures in C

• Java:
  – Explicit memory allocation
  – Deallocation by GC

• C/C++:
  – Explicit memory allocation & deallocation

• When is it safe to deallocate a piece of memory?
Deletion revisited: Delete(10)
De-allocate to the OS?
Re-use as something else?
Re-use as a list node?

Search(20)

H

20 200

30

T

H

30

T
Reference counting

1. Decide what to access

H 10 30 T
0 0 0 0
Reference counting

1. Decide what to access
2. Increment reference count
Reference counting

1. Decide what to access
2. Increment reference count
3. Check access still OK
Reference counting

1. Decide what to access
2. Increment reference count
3. Check access still OK
1. Decide what to access
2. Increment reference count
3. Check access still OK
Reference counting

1. Decide what to access
2. Increment reference count
3. Check access still OK
4. Defer deallocation until count 0
Epoch mechanisms

Global epoch: 1000
Thread 1 epoch: -
Thread 2 epoch: -
Epoch mechanisms

- Global epoch: 1000
- Thread 1 epoch: 1000
- Thread 2 epoch: -

1. Record global epoch at start of operation
Epoch mechanisms

Global epoch: 1000
Thread 1 epoch: 1000
Thread 2 epoch: 1000

1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists

Deallocate @ 1000
Epoch mechanisms

- Global epoch: 1001
- Thread 1 epoch: 1000
- Thread 2 epoch: -

1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists
3. Increment global epoch at end of operation (or periodically)

Deallocate @ 1000
Epoch mechanisms

Global epoch: 1002
Thread 1 epoch: -
Thread 2 epoch: -

1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists
3. Increment global epoch at end of operation (or periodically)
4. Free when everyone past epoch

Deallocate @ 1000
Hazard pointer mechanisms

1. Decide what to access
2. Set hazard pointer
3. Check access still OK

Thread 1 hazards
Hazard pointer mechanisms

1. Decide what to access
2. Set hazard pointer
3. Check access still OK

Thread 1 hazards
Hazard pointer mechanisms

1. Decide what to access
2. Set hazard pointer
3. Check access still OK

Thread 1 hazards
Hazard pointer mechanisms

1. Decide what to access
2. Set hazard pointer
3. Check access still OK

Thread 1 hazards
Hazard pointer mechanisms

1. Decide what to access
2. Set hazard pointer
3. Check access still OK

Thread 1 hazards
Hazard pointer mechanisms

1. Decide what to access
2. Set hazard pointer
3. Check access still OK

Thread 1 hazards
Hazard pointer mechanisms

1. Decide what to access
2. Set hazard pointer
3. Check access still OK
4. Batch allocations and defer deallocation while hazard pointer present

Thread 1 hazards
Fine-grain parallel tasks
Work stealing queues & ABA
Memory management
Summary

• Hard to untangle algorithm design from detailed knowledge of the hardware
  – Memory models
  – Costs of operations
  – Scalability
Summary (2)

• General principle:
  – Avoid introducing contention between things that are logically independent
  – Contention for the same locks
  – Contention in the memory system
Summary (3)

• Linearizability as a way of defining what a concurrently accessed data structure should do

• If building directly from CAS then
  – Ensure that a single CAS really does make an atomic update to the logical state
  – Beware of delays & ABA problems