Lecture 11: HW3, Rest of Parallel Patterns, Load Balancing

G63.2011.002/G22.2945.001 · November 16, 2010
Outline

Divide-and-Conquer

General Data Dependencies
Outline

Divide-and-Conquer

General Data Dependencies
Divide and Conquer

\[ y_i = f_i(x_1, \ldots, x_N) \]

for \( i \in \{1, \text{dots}, M\} \).

**Main purpose:** A way of partitioning up fully dependent tasks.
Divide and Conquer

\[ y_i = f_i(x_1, \ldots, x_N) \]

for \( i \in \{1, \ldots, M\} \).

**Main purpose:** A way of partitioning up fully dependent tasks.

Processor allocation?
Divide and Conquer: Examples

- GEMM, TRMM, TRSM, GETRF (LU)
- FFT
- Sorting: Bucket sort, Merge sort
- $N$-Body problems (Barnes-Hut, FMM)
- Adaptive Integration

More fun with work and span:

D&C analysis lecture
Divide and Conquer: Issues

- “No idea how to parallelize that”
  - → Try D&C
- Non-optimal during partition, merge
  - But: Does not matter if deep levels do heavy enough processing
- Subtle to map to fixed-width machines (e.g. GPUs)
  - Varying data size along tree
- Bookkeeping nontrivial for non-$2^n$ sizes
- Side benefit: D&C is generally cache-friendly
Outline

Divide-and-Conquer

General Data Dependencies
General Dependency Graphs

B = f(A)
C = g(B)
E = f(C)
F = h(C)
G = g(E,F)
P = p(B)
Q = q(B)
R = r(G,P,Q)

Great: All patterns discussed so far can be reduced to this one.
Great: All patterns discussed so far can be reduced to this one.
Cilk

cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;
        
        x = spawn fib (n−1);
        y = spawn fib (n−2);
        
        sync;
        
        return (x+y);
    }
}

Features:

- Adds keywords spawn, sync, (inlet, abort)
- Remove keywords → valid (seq.) C

Timeline:

- Developed at MIT, starting in ‘94
- Commercialized in ‘06
- Bought by Intel in ‘09
- Available in the Intel Compilers

Efficient implementation? D&C General
Cilk

cilk int fib (int n) {
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}

Features:
- Adds keywords spawn, sync, (inlet, abort)
- Remove keywords → valid (seq.) C

Timeline:
- Developed at MIT, starting in ‘94
- Commercialized in ‘06
- Bought by Intel in ‘09
- Available in the Intel Compilers

Efficient implementation?
Work-Stealing

Each processor maintains a **work deque** of ready threads, and it manipulates the bottom of the deque like a stack.
Work-Stealing

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a **work deque** of ready threads, and it manipulates the bottom of the deque like a stack.

Return!
Work-Stealing

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it *steals* a thread from the top of a *random* victim’s deque.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it *steals* a thread from the top of a *random* victim’s deque.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a **work deque** of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it **steals** a thread from the top of a **random** victim’s deque.
Work-Stealing

Each processor maintains a **work deque** of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it *steals* a thread from the top of a *random* victim’s deque.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it steals a thread from the top of a random victim's deque.

Why is Work-Stealing better than a Task Queue?

With material by Charles E. Leiserson (MIT)
General Graphs: Issues

- Model can accommodate ‘speculative execution’
  - Launch many different ‘approaches’
  - Abort the others as soon as one satisfactory one emerges.
- Discover dependencies, make up schedule at run-time
  - Usually less efficient than the case of known dependencies
  - Map-Reduce absorbs many cases that would otherwise be general
- On-line scheduling: complicated
- Not a good fit if a more specific pattern applies
- Good if inputs/outputs/functions are (somewhat) heavy-weight
Questions?