15-853: Algorithms in the Real World

Parallelism: Lecture 1
Nested parallelism
Cost model
Parallel techniques and algorithms

Outline

Concurrency vs. Parallelism
Quicksort example
Nested Parallelism
  - fork-join and parallel loops
Cost model: work and span
Techniques:
  - Using collections: inverted index
  - Divide-and-conquer: merging, mergesort, kd-trees, matrix multiply, matrix inversion, fft
  - Contraction: quickselect, list ranking, graph connectivity, suffix arrays

Parallelism is here... And Growing!

Andrew Chien, 2008
Parallelism in "Real world" Problems

- Optimization
- N-body problems
- Finite element analysis
- Graphics
- JPEG/MPEG compression
- Sequence alignment
- Rijndael encryption
- Signal processing
- Machine learning
- Data mining

Parallelism vs. Concurrency

- **Parallelism**: using multiple processors/cores running at the same time. Property of the machine
- **Concurrency**: non-determinacy due to interleaving threads. Property of the application.

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>sequential</td>
</tr>
<tr>
<td>parallel</td>
<td>concurrent</td>
</tr>
</tbody>
</table>

Parallelism: nested parallelism

```
nested parallelism = arbitrary nesting of parallel loops + fork-join

- Assumes no synchronization among parallel tasks except at joint points.
- Deterministic if no race conditions

Advantages:
- Good schedulers are known
- Easy to understand, debug, and analyze
```

Nested Parallelism: parallel loops

- **Cilk**
  
  ```
cilk_for (i=0; i < n; i++)
      B[i] = A[i]+1;
  
  #pragma omp parallel
  for (i=0; i < n; i++)
      B[i] = A[i] + 1;
  ```

- **Microsoft TPL (C#, F#)**

  ```
  Parallel.ForEach(A, x => x+1);
  ```

- **Nesl, Parallel Haskell**

  ```
  B = {x + 1 : x in A}
  ```

- **OpenMP**

  ```
  #pragma omp for
  for (i=0; i < n; i++)
      B[i] = A[i] + 1;
  ```
**Nested Parallelism: fork-join**

```
cobegin {
  S1;
  S2;}
```

```
coinvoke(f1,f2)
```

```
Parallel.invoke(f1,f2)
```

```
#pragma omp sections
{
  #pragma omp section
  S1;
  #pragma omp section
  S2;
}
```

Dates back to the 70s or possibly 60s. Used in dialects of Pascal Java fork-join framework

Microsoft TPL (C#, F#)

OpenMP (C++, C, Fortran, ...)

**Serial Parallel DAGs**

Dependence graphs of nested parallel computations are series parallel

Two tasks are parallel if not reachable from each other. A data race occurs if two parallel tasks are involved in a race if they access the same location and at least one is a write.

**Cost Model**

Cost Model:

- **Work:** total number of operations
  - costs are added across parallel calls

- **Span:** depth/critical path of the computation
  - Maximum span is taken across forked calls

- **Parallelism:** \( \frac{\text{Work}}{\text{Span}} \)
  - Approximately \# of processors that can be effectively used.
Combining costs

Combining for parallel for:
\[
p\text{for } (i=0; i<n; i++)
\]
\[
f(i);
\]
\[
W_{pexp}(\text{pfor ...}) = \sum_{i=0}^{n-1} W_{exp}(f(i)) \quad \text{work}
\]
\[
D_{pexp}(\text{pfor ...}) = \max_{i=0}^{n-1} D_{exp}(f(i)) \quad \text{span}
\]

Why Work and Span

Simple measures that give us a good sense of efficiency (work) and scalability (span).

Can schedule in \( O(W/P + D) \) time on \( P \) processors.
This is within a constant factor of optimal.

Goals in designing an algorithm

1. Work should be about the same as the sequential running time. When it matches asymptotically we say it is work efficient.

2. Parallelism \((W/D)\) should be polynomial \( O(n^{1/2}) \) is probably good enough

Example: Quicksort

\[
\text{function quicksort}(S) =
\]
if (#\( S \) <= 1) then \( S \)
else let
\[
a = S[\text{rand}(\#S)];
\]
\[
S_1 = \{e \in S \mid e < a\};
\]
\[
S_2 = \{e \in S \mid e = a\};
\]
\[
S_3 = \{e \in S \mid e > a\};
\]
\[
R = \{\text{quicksort}(v) \mid v \in [S_1, S_3]\};
\]
in \( R[0] \) ++ \( S_2 \) ++ \( R[1] \);

QuickSort Complexity

Sequential Partition and appending
Parallel calls
Work = \( O(n \log n) \)
Partition
Recursive calls
(less than, ...)
Span = \( O(n) \)
Parallelism = \( O(\log n) \)

Not a very good parallel algorithm

*All randomized with high probability
QuickSort Complexity

Now let's assume the partitioning and appending can be done with:
- Work = \(O(n)\)
- Span = \(O(\log n)\)
but recursive calls are made sequentially.

QuickSort Complexity

Parallel partition
Sequential calls

- Work = \(O(n \log n)\)
- Span = \(O(n)\)
- Parallelism = \(O(\log n)\)

Not a very good parallel algorithm

*All randomized with high probability

QuickSort Complexity

Parallel partition
Parallel calls

- Work = \(O(n \log n)\)
- Span = \(O(\log^2 n)\)
- Parallelism = \(O(n/\log n)\)

A good parallel algorithm

*All randomized with high probability

Caveat: need to show that depth of recursion is \(O(\log n)\) with high probability
Parallel selection

\{ e \in S \mid e < a \};

\begin{align*}
S &= [2, 1, 4, 0, 3, 1, 5, 7] \\
F &= S < 4 &= [1, 1, 0, 1, 1, 1, 0, 0] \\
I &= \text{addscan}(F) &= [0, 1, 2, 2, 3, 4, 5, 5]
\end{align*}

where \( F \cdot R[I] = S \)

[2, 1, 0, 3, 1]

Each element gets sum of previous elements. Seems sequential?

Scan code

```python
function addscan(A) =
  if (#A <= 1) then [0]
  else let
    evens = addscan(sums);
    odds = {evens[i] + A[2*i] : i in [0:#a/2]};
    in interleave(evens, odds);
  W(n) = W(n/2) + O(n) = O(n)
  D(n) = D(n/2) + O(1) = O(log n)
```

Parallel Techniques

Some common themes in “Thinking Parallel”

1. Working with collections.
   - map, selection, reduce, scan, collect
2. Divide-and-conquer
   - Even more important than sequentially
   - Merging, matrix multiply, FFT, ...
3. Contraction
   - Solve single smaller problem
   - List ranking, graph contraction
4. Randomization
   - Symmetry breaking and random sampling
Working with Collections

reduce $\odot [a, b, c, d, \ldots $
\quad $= a \odot b \odot c \odot d + \ldots$

scan $\odot$ ident $[a, b, c, d, \ldots$
\quad $= [ident, a, a \odot b, a \odot b \odot c, \ldots$

sort compF $A$

collect $[[2,a), (0,b), (2,c), (3,d), (0,e), (2,f)]$
\quad $= [(0, [b,e]), (2,[a,c,f]), (3,[d])]$