Performance analysis of parallel algorithms on multi-core platform
Today’s picture
Cilk

//part A
cilk_spawn foo();
bar();
cilk_sync;
//part B

//part A
In parallel
    foo()
    bar()
//part B
Quicksort

Function QuickSort (A)

\[ p \leftarrow \text{random pivot} \]
\[ s \leftarrow 1, \ t \leftarrow |A| \]
\[ \text{WHILE } s < t \]
\[ \quad \text{WHILE } A_s < p: \ s++ \]
\[ \quad \text{WHILE } A_t > p: \ t-- \]
\[ \quad \text{IF } s < t: \ \text{swap} (A_{s++}, A_{t--}) \]
\[ \text{QuickSort} \ (A_1 \ldots s) \]
\[ \text{QuickSort} \ (A_{s+1} \ldots |A|) \]
**Quicksort**

Function QuickSort (A)

if |A| is small then BaseCase

p $\leftarrow$ random pivot

s $\leftarrow$ 1, t $\leftarrow$ |A|

WHILE s < t

    WHILE A_s < p: s++
    WHILE A_t > p: t--

    IF s < t: swap(A_s++, A_t--)

In Parallel

QuickSort (A_1...s)

QuickSort (A_{s+1}...|A|)
Quicksort

Fork-join Parallelism
(aka. Nested-Parallel Model)

Parallel primitive:

\[ A() \parallel B() \]

Can be used to implement parallel_for with \( \log n \) levels
Parallelism of Randomized Work-Stealing Scheduler
A = 1 2 3 4 5 6 7 8

\[
\begin{align*}
6 & \quad + \quad 15 & \quad + \quad 15 & = 36 \\
A & \quad 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{align*}
\]

Sum(A): 36

- Cut the input array into smaller segments, sum each up individually, and finally sum up the sums

- Picking the appropriate number of segments can be annoying
  - Machine parameter, runtime environment, algorithmic details
Function SUM(A)

If \(|A| = 1\) then return \(A(1)\)

In Parallel

\(a = \text{SUM(first half of A)}\)
\(b = \text{SUM(second half of A)}\)

return \(a + b\)
A = 1 2 3 4 5 6 7 8

Sum(A):

3 7 11 15
10 26
36

Function SUM(A)
If |A| = 1 then return A(1)
cilk_spawn a = SUM(first half of A)
  b = SUM(second half of A)
cilk_sync
return a + b
A = 
\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
+ & + & + & + & + & + & + & + \\
3 & 7 & 11 & 15 & 26 & 36 & & \\
+ & + & + & + & + & + & + & + \\
10 & 17 & 22 & 31 & 41 & & & \\
+ & + & + & + & + & + & + & + \\
36 & & & & & & & \\
\end{array}
\]

Sum(A): 

Fork-join Parallelism

Parallel primitive:

\[A() \parallel B()\]

Can be used to implement parallel_for with log \( n \) levels
\[
\begin{align*}
A &= 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \\
\text{Sum}(A): &\quad 3 \quad 7 \quad 11 \quad 15 \\
&\quad 10 \quad 26 \\
&\quad 36 \\
\end{align*}
\]

**Work** \( W \): total computation \( O(n) \)

**Depth** \( D \): longest chain of all dependencies \( O(\log n) \)
Quicksort

Function QuickSort (A)

\[
p \leftarrow \text{random pivot} \\
s \leftarrow 1, \ t \leftarrow |A| \\
\text{WHILE} \ s < t \\
\quad \text{WHILE} \ A_s < p: \ s++ \\
\quad \text{WHILE} \ A_t > p: \ t-- \\
\quad \text{IF} \ s < t: \ 
\text{swap}(A_{s++}, A_{t--})
\]

In Parallel

QuickSort (A_{1...s})
QuickSort (A_{s+1...|A|})
Quicksort

Quicksort(1,100)

Partition

Quicksort(1,40)
  Partition
  QS(1,15)  QS(16,40)

Quicksort(41,100)
  Partition
  QS(41,70)  QS(71,100)

$O(n)$
A work-stealing scheduler can run a fork-join computation on $p$ cores using time:

$$\frac{W}{p} + O(D)$$

Work $W$: total computation

Depth $D$: longest chain of all dependencies
A work-stealing scheduler can schedule this computation on $p$ cores using time:

$$O \left( \frac{n}{p} + \log n \right)$$
Randomized Work-stealing Scheduler
[Blumofe and Leiserson 1999]

A work-stealing scheduler can run a fork-join computation on \( p \) cores using time:

\[
\frac{W}{p} + O(D)
\]

(The work-stealing scheduler is used in OpenMP, CilkPlus, Intel TBB, MS PPL, etc.)

Work \( W \): total computation

Depth \( D \): longest chain of all dependencies
Quicksort

Function QuickSort (A)
    p ← random pivot
    s ← 1, t ← |A|
    WHILE s<t
        WHILE A_s<p: s++
        WHILE A_t>p: t--
        IF s<t: swap(A_s, A_t)

In Parallel
    QuickSort (A_1...s)
    QuickSort (A_{s+1}...|A|)
Quicksort

Quicksort(1,100)

Partition

Quicksort(1,40)

Partition

QS(1,15)  QS(16,40)

Quicksort(41,100)

Partition

QS(41,70)  QS(71,100)

$O(n)$
QuickSort

Function QuickSort (A)
    p ← random pivot
    s ← 1, t ← |A|
    WHILE s<t
        WHILE A_s<p: s++
        WHILE A_t>p: t--
        IF s<t: swap(A_s, A_t)

In Parallel
    QuickSort (A_1...s)
    QuickSort (A_{s+1}...|A|)

\[ O\left(\frac{n \log n}{p} + n\right) = \frac{T_1}{p} + O(n) \]
Prefix sum

\[
A = \begin{array}{cccccccc}
& 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 1 & 3 & 6 & 10 & 15 & 21 & 28
\end{array}
\]
Prefix sum

A =

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>15</th>
<th>21</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Function PrefixSum(s, t, ps)

If s=t then return P(s) ← ps

In Parallel

PrefixSum(s, mid, ps)
PrefixSum(mid+1, t, ps+leftSum)
Function Select (A, I)

\[
P \leftarrow \text{PrefixSum}(I)
\]

\[
\text{Parallel_for (i} \leftarrow 1 \text{ to } |A|)
\]

\[
\text{if (l(i) = 1) then } O(P(i)) \leftarrow A(i)
\]

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| O  | 1 | 4 | 7 | 8 |
|----|---|---|---|

| P  | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 3 |
Quicksort

Function QuickSort (A)

\[ p \leftarrow \text{random pivot} \]
\[ L \leftarrow \text{Select } (A, <p) \]
\[ M \leftarrow \text{Select } (A, =p) \]
\[ R \leftarrow \text{Select } (A, >p) \]

In parallel

QuickSort (L)
QuickSort (R)

Return \( L + M + R \)

Work: \( O(n \log n) \)
Quicksort

Quicksort(1,100)

Partition

Quicksort(1,40)

Partition

QS(1,15)  QS(16,40)

Quicksort(41,100)

Partition

QS(41,70)  QS(71,100)

$O(\log n)$
Quicksort

Function QuickSort (A)
    p ← random pivot
    L ← Select (A, <p)
    M ← Select (A, =p)
    R ← Select (A, >p)
    In parallel
        QuickSort (L)
        QuickSort (R)
    Return L + M + R

Work: $O(n \log n)$

Depth: $O(\log^2 n)$
(with high probability)
Do most problems can be solved in polylogarithmic depth?

Yes:
Reduce, prefix sum, comparison sort
List / tree contraction / ranking
Most graph algorithms on undirected graphs
Matrix multiplication, FFT, most comp geo.

Yes but require (significant) extra work:
Most graph algorithms on directed graphs, shortest-paths

No polylog depth algorithm is known:
Radix sort, network flow
Wrap up

- As long as the depth of an algorithm is small (i.e. polylogarithmic), randomized work-stealing scheduler can have perfect load balance, and the overhead of scheduling is negligible.

- Two key points:
  - Smaller (polylogarithmic) depth
  - Better sequential code (I/O to the memory)
How to write faster (parallel) code?

Algorithms with lower time complexity (less operations)?

\( (n = 2 \times 10^9) \)

```c
parallel_for (i ← 1 to n)
    a[i] += 1
```

Running time: 0.47s

```c
parallel_for (i ← 1 to n)
    t = rand() % n
    a[t] += 1
```

Running time: 2.22s
Cache complexity
The Random-Access Machine (RAM) model

- Unit cost to read from and write to any specific memory location
  - Therefore unit cost for any operation

- **Time complexity** $T$ of an algorithm is the minimum amount of cost of to run it on the RAM model
The External Memory (EM) model

- Two-level memory hierarchy:
  - A small memory (fast memory, cache) of fixed size $M$
  - A large memory (slow memory) of unbounded size
- Both are partitioned into blocks of size $B$
- Instructions can only apply to data in primary memory, and are free
The EM model has two special \textit{memory transfer} instructions:

- **Read transfer**: load a block from slow memory
- **Write transfer**: write a block to slow memory

The complexity of an algorithm on the EM model (I/O complexity) is measured by:

\[(\text{#(read transfer)}) + \text{(#(write transfer)})\]
The cache can hold whatever $M/B$ blocks YOU like, each with size $B$

The algorithm can freely access the cache and apply computations

Take unit cost to load or store an arbitrary block to the (large) memory

The cache complexity $Q$ of an algorithm is the minimum amount of replacement to finish the computation

(Usually $M \sim 10^6$ for L3 cache, $B$ is 64 bytes)
List access example \((n = 2 \times 10^9)\)

```c
parallel_for (i \leftarrow 1 to n)
    a[i] += 1
```

Running time: 0.47s

\[Q = \frac{2n}{B}\]

**Scan (sequential writes)**

Scatter vs. scan: 4.72

```c
parallel_for (i \leftarrow 1 to n)
    t = rand() \% n
    a[t] += 1
```

Running time: 2.22s

\[Q = 2n\]

**Scatter (random writes)**
Why is this simple model good enough?

- Lemma: If an algorithm incurs $Q$ memory transfers on an ideal cache of size $M/2$, then it incurs at most $2Q$ memory transfers on a fully associative cache of size $M$ with LRU replacement.

- In practice, the real cache is very clever and almost LRU, and the performance matches $Q$ when the computation is I/O-bounded.
Matrix multiplication \((n = 2^{13})\)

\[
\text{parallel\_for (} i \leftarrow 1 \text{ to } n) \\
\quad \text{for (} j \leftarrow 1 \text{ to } n) \\
\qquad \text{for (} k \leftarrow 1 \text{ to } n) \\
\qquad \quad a[i][j] += b[i][k] \times c[k][j]
\]

\[Q = O(n^3), \ n^2 \text{ scatters}\]

Running time: 216s
Matrix multiplication \((n = 2^{13})\)

Transpose(c)

parallel_for (i \leftarrow 1 \text{ to } n)
  for (j \leftarrow 1 \text{ to } n)
    for (k \leftarrow 1 \text{ to } n)
      a[i][j] += b[i][k] \times c[j][k]

\[ Q = O(n^3/B), \quad n^2 \text{ scans} \]

Running time: 69s
Matrix multiplication ($n = 2^{13}$)

Func A*B {
    if A is small then baseCase
    In Parallel
    \[C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}\]
    \[C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}\]
    \[C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}\]
    \[C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}\]
    return C
}

\[A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}\]
\[B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}\]

\[Q = O(n^3 / B \sqrt{M})\]

Running time: 9.8s
Parallel cache complexity
[Ac'ar, Blelloch, Blumofe 2000]

- Practically, as long as the depth is small (i.e. polylogarithmic), the additional cache complexity caused by parallelism is bounded and usually negligible.

- Using work-stealing scheduler and with private caches, the cache complexity of a parallel algorithm using \( p \) cores satisfies (*):

\[
Q_p = Q_{seq} + O(PDM/B)
\]

(*) with high probability

loose upper bound
Goals to design faster parallel algorithms

- The cost of a parallel algorithm:
  - Computations (CPU operations): captured by **time complexity**, should be as good as the best sequential algorithm
  - Memory access (I/Os to main memory): captured by **cache complexity**
    - Can usually be decomposed in the numbers of scatters and scans. Try to minimize them (including the constants)
  - Extra cost by parallel execution: captured by the **depth** (ideally should be polylogarithmic)
Algorithms that are I/O-efficient

- Memory access (I/Os to main memory): captured by cache complexity

- Algorithms that optimize it are called I/O-efficient algorithms. Interesting results:
  - Matrix multiplication: $\Theta\left(\frac{n^3}{B\sqrt{M}}\right)$
  - Comparison sorting: $\Theta\left(\frac{n}{B \log_{M\frac{M}{B}} n}\right)$
  - Searching: $\Theta(\log_B n)$

- Algorithms that are not aware of cache parameters $(M, B)$ are called cache-oblivious algorithms
Goals to design faster parallel algorithms

- The cost of a parallel algorithm:
  - Computations (CPU operations): captured by time complexity, should be as good as the best sequential algorithm
  - Memory access (I/Os to main memory): captured by cache complexity
    - Can usually be decomposed in the numbers of scatters and scans. Try to minimize them (including the constants)
  - Extra cost by parallel execution: captured by the depth (ideally should be polylogarithmic)
Algorithms with low depth

- Extra cost by parallel execution: captured by the depth (ideally should be polylogarithmic)

- Problems that can be solved in $O(1)$ depth are in $NC^0$
- Problems that can be solved in $\tilde{O}(\log n)$ depth are in $NC^1$
- Problems that can be solved in $\tilde{O}(\log^2 n)$ depth are in $NC^2$

$NC^0 \subseteq NC^1 \subseteq NC^2 \subseteq \ldots \subseteq NC$

- Does $NC = P$?
Goals to design faster parallel algorithms

- The cost of a parallel algorithm:
  - Computations (CPU operations): captured by time complexity, should be as good as the best sequential algorithm
  - Memory access (I/Os to main memory): captured by cache complexity
    - Can usually be decomposed in the numbers of scatters and scans. Try to minimize them (including the constants)
  - Extra cost by parallel execution: captured by the depth (ideally should be polylogarithmic)
Useful links:

- CMU 15-210: Parallel and Sequential Data Structures and Algorithms
  - An overview of basic algorithms and data structures that makes no distinction between sequential and parallel

- MIT 6.172: Performance Engineering of Software Systems
  - A more thorough explanation on performance analysis on parallel (multi-core) and distributed setting

- CMU: 15-853: Algorithms in the “Real World”
  - A graduate-level course that provides many useful links to parallel algorithms and I/O-efficient algorithms