Analysis of Work-Stealing Scheduler

Yan Gu
6.886 Algorithm Engineering
May 2, 2019
Shared-memory multicore parallelism

6-8 cores

16-40 cores
PC/Workstation

64-100+ cores
High-end servers

50-100x increase in the last 15 years!
Function \text{SUM}(A)

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

In Parallel

\begin{align*}
    a &= \text{SUM}(\text{first half of } A) \\
    b &= \text{SUM}(\text{second half of } A) \\
    \text{return } a + b
\end{align*}
\( A = \begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
+ & + & + & + & + & + & +
\end{array} \\
\text{SUM}(A): \begin{array}{c}
3 & 7 & 11 & 15 & 36
\end{array}
\}

- **Work** \((W)\): the number of operations (ideally it should match the best sequential solution)
  - Less overhead

- **Span** \((D)\): the longest dependence in this computation (ideally to be polylogarithmic)
  - Better scalability
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
parallel_for (int i=0; i<n; i++)
    a[i] = f(a[i]);

How is your code actually executed on hardware?

Why analyzing work and span?
parallel_for (int i=0; i<n; i++)
    a[i] = f(a[i]);
parallel_for (int i=0; i<n; i++)
    a[i] = f(a[i]);
Treat the computation as a DAG

Function SUM(A)
   If |A| = 1 then return A(1)
   In Parallel
      a = SUM(first half of A)
      b = SUM(second half of A)
   return a + b
Greedy scheduler

**IDEA:** Do as much as possible on every step.
Greedy scheduler

**IDEA:** Do as much as possible on every step.

Either execute $p$ operations
**Greedy scheduler**

**IDEA:** Do as much as possible on every step.

Either execute $p$ operations
Or reduce the span by 1

$$T \leq \frac{W}{P} + D$$
Greedy scheduler

Impractical:

- Assumes processors/threads run in lockstep
- Big overhead in context switching
- Different operations have very different costs
Work-stealing scheduler

- Full details in 6.172: Performance Engineering of Software Systems (Cilk implementation)

- If a processor spawns tasks at a FORK, it continues execution with one of the spawned subtasks, and push the other subtask to the front its queue

- If a processor completes a task, it tries to pull a task from the front of its own queue

- If a processor finishes all tasks in its own queue, it randomly selects another processor, and steals a task from the end of the victim queue (retry if failed)
Work-stealing scheduler

P = 3
Work-stealing scheduler
parallel_for (int i=0; i<n; i++)
    a[i] = f(a[i]);
Overhead of work-stealing scheduler

Bound the number of steals (whp):

$O(pD)$
Overhead of work-stealing scheduler

Bound the number of steals (whp): 

\[ O(pD) \]

Running time (whp):

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

Cache reload:

\[ O(pD) \]
Assumptions

Steals come asynchronously

Multiple steals can be made to the same thread, and one wins (adversarially)

A successful steal from thread A would not block two consecutive steals from another thread B
Proof outline

Consider one specific path

Left child: executed directly after the previous node

Right child:
• Stolen by another thread
• Executed when the current thread finishes the left side

Join node: executed when all previous nodes are finished

\[ T = \frac{W + O(pD)}{p} = \frac{W}{p} + O(D) \]
Proof outline

Consider one specific path

Consider the worst case:
- All nodes are right child
- All of them need to be stolen

We want to show that $O(pD)$ steals are sufficient to steal $D$ tasks whp
How many steals do we need?

Challenge: steals happen asynchronously
• They can block each other
Best case: steals are attempted one after another
Each steal has $1/(p - 1)$ probability to steal one task

Chernoff bound: for $n$ independent random variables in \{0, 1\}, let $X$ be the sum, and $\mu = E[X]$, then for any $0 < \delta < 1$,\
\[
\Pr(X \geq (1 - \delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2}}
\]
How many steals do we need?

Best case: steals are attempted one after another

Each steal has \( \frac{1}{\rho - 1} \) probability to steal one task

Let’s say we have \( 2(\rho - 1)(D + \ln(1/\epsilon)) \) steal attempts

The probability that we have at least \( D \) successful steals from \( 2(\rho - 1)(D + \ln(1/\epsilon)) \) attempts is \( 1 - \epsilon \)

\[
\begin{align*}
\frac{\delta^2 \mu}{2} &= \left(\frac{\mu-D}{\mu}\right)^2 \frac{\mu}{2} = \frac{(\mu-D)^2/\mu}{2} < e^{(2D-\mu)/2} = e^{-\ln(1/\epsilon)} = \epsilon
\end{align*}
\]

\[
\Pr(X \geq (1 - \delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2}}
\]
How many steals do we need?

Worst case: $p - 1$ steals are always attempted together.

Probability that none of the steals touch the current thread:

$$\left(1 - \frac{1}{p-1}\right)^{p-1} < \frac{1}{e}$$
How many steals do we need?

Worst case: $p - 1$ steals are always attempted together

One task is stolen by probability at least $1 - 1/e$

Let’s say we have $2e/(e - 1)(D + \log(1/\epsilon))$ rounds of steals

Expected steals: $\mu = 2(D + \log(1/\epsilon))$

If we have less than $D$ steals, then $\delta = (\mu - D)/\mu$, and

$$e^{-\frac{\delta^2 \mu}{2}} = e^{-\frac{((\mu-D)/\mu)^2 \mu}{2}} = e^{-\frac{(\mu-D)^2/\mu}{2}} < e^{(2D-\mu)/2} = \epsilon$$

The probability that we have at least $D$ successful steals from $2(p - 1)e(D + \ln(1/\epsilon))/(e - 1)$ attempts is $1 - \epsilon$
How many steals do we need?

To get $D$ steals with probability $1 - \epsilon$, we need

- **Best case:** $2(p - 1)(D + \ln(1/\epsilon))$ steals
- **Worst case:** $2(p - 1)(D + \ln(1/\epsilon))e/(e - 1)$ steals

We want to guarantee probability with $1 - 1/n^c$

In a DAG with depth $D$, there are in total $\leq 2^D$ paths

Let $\epsilon = 1/(2^D \cdot n^c)$, then $\ln(1/\epsilon) = c \ln n + D \ln 2$

$O(pD)$ steals are sufficient for all possible paths whp
Overhead of work-stealing scheduler

The number of steals (whp): \( O(pD) \)

Running time (whp):
\[
T = \frac{W + O(pD)}{p} = \frac{W}{p} + O(D)
\]

Cache reload: \( O(pD) \)