(cont. from last lecture)

- Serial time: \( T_S \)
- Parallel time (p processors): \( T_P \)
- Overhead: \( T_O = p T_P - T_S \)
- Speedup: \( S = \frac{T_S}{T_P} \)
- Efficiency: \( E = \frac{S}{p} = \frac{T_S}{p T_P} = \frac{1}{1 + T_O/T_S} \)
- Cost Optimal: \( p T_P = O(T_S) \)
(cont. from last lecture)

- Parallel Reduction:
  - The iterative version:
    \[ T_P = O\left(\frac{n}{p} \log p\right) \quad \Rightarrow \quad pT_P = O\left(n \log p\right) \]
  - The serial sum version:
    \[ T_P = O\left(\frac{n}{p} + \log p\right) \quad \Rightarrow \quad pT_P = O\left(n + p \log p\right) \]
Scalability

- How does the system scale up and down with different number of input size (work size)?
- Let's pick parallel reduction as an example, and make up the actual cost:

\[ T_p = \frac{n}{p} + 2 \log p \]

\[ S = \frac{n}{\frac{n}{p} + 2 \log p} \]

\[ T_o = 2p \log p \]
Scalability

- Speedup:
Scalability

- Speedup as a function of $p$ ($n = 256$)
Scalability

- Speedup as a function of $p$ ($n = 256$)

Overhead is a function of $p$, as a result, as $p$ goes up, not only the overhead will take over the computation time, but also it grows with $p$, causing speedup to drop.

$$T_o = 2 \ p \ \log \ p$$
Efficiency

- Efficiency as a function of $p$ ($n = 256$)

\[
E = \frac{1}{1 + \frac{2p \log p}{n}}
\]
Efficiency

- Efficiency as a function of $n$ ($p = 128$)

$$E = \frac{1}{1 + \frac{2p \log p}{n}}$$
Scalability

• In many cases, overhead grows sublinearly with respect to the work size.

• As a result, we can increase efficiency by simultaneously increase the work size and the number of processors to keep efficiency constant.

• Such systems are called scalable systems.

• Scalability reflects a parallel system's ability to utilize increasing resources effectively.
The Isoefficiency Function

- How much should the work size grow to maintain the same efficiency?

\[ E = \frac{1}{1 + \frac{2 \ p \ \log \ p}{n}} \quad \Rightarrow \quad n = \frac{E}{1 - E} (2 \ p \ \log \ p) \]
The Isoefficiency Function

- How much should the work size grow to maintain the same efficiency?

\[ E = \frac{1}{1 + \frac{2p \log p}{n}} \]

- Generally:

\[ W = \frac{E}{1 - E} T_o \]

Isoefficiency function

Problem size should grow proportionally to the overhead function, to maintain the same efficiency.
Minimum Execution Time

- Assume the work size is $n$, how many processors should I throw in to get the minimum execution time?

$$T_p = \frac{n}{p} + 2 \log p$$
Minimum Execution Time

- Assume the work size is $n$, how many processors should I throw in to get the minimum execution time?

$$T_P = \frac{n}{p} + 2 \log p$$

- Differentiate $T_P$ and solve the minimization problem

Turns out: $p = \frac{n}{2}$ and $T_P^{\text{min}} = 2 \log n$
Minimum Execution Time

- Assume the work size is n, how many processors should I throw in to get the minimum execution time?

\[ T_p = \frac{n}{p} + 2 \log p \]

- Differentiate \( T_p \) and solve the minimization problem

  \[
  p = \frac{n}{2} \quad \text{and} \quad T_{p}^{\text{min}} = 2 \log n
  \]

- However, by coupling \( p \) and \( n \), the algorithm is no longer cost-optimal!
Minimum Cost-Optimal Execution Time

• In order for the system to remain cost-optimal:

\[ f(p) = p \log p = O(n) \]

• Hence, for a problem size of \( n \), we can throw in as much as \( f^{-1}(n) \) processors to remain cost-optimal.

• Therefore, the minimum cost-optimal execution time we can achieve is:

\[ O\left( \frac{n}{f^{-1}(n)} \right) \]
Minimum Cost-Optimal Execution Time

Let's take:

\[ p = \frac{n}{\log n} \]

Since \( T_p \) reduces monotonically up till \( p = \frac{n}{2} \), we get:

\[ T_{p}^{\text{cost optimal min}} = \frac{n}{p} + 2 \log p = 3 \log n - \log \log n \]
Prefix Sum (Scan) and Applications

• Problem Definition:
  Given sequence A of n elements and an associative operator \( \oplus \) with identity I

Inclusive scan:
\[
\text{scan}(A, \oplus) = [A_0, (A_0 \oplus A_1), \ldots, (A_0 \oplus A_1 \oplus \ldots A_{n-1})]
\]

Exclusive scan:
\[
\text{prescan}(A, \oplus) = [I, A_0, (A_0 \oplus A_1), \ldots, (A_0 \oplus A_1 \oplus \ldots A_{n-2})]
\]

• Example: \( A = [3 1 7 0 4 1 6 3] \)
Prefix Sum (Scan) and Applications

• Why is this importance?
  Lots of applications: split, radix sort, building trees, subdivision, string comparison, line of sight, histogram, run-length encoding......

• Naïve serial algorithm: linear sum in O(n) time

• Important to study data-parallel version of the algorithm!
Prefix Sum (Scan) and Applications

• Reading List:
  • “Parallel Prefix Sum (Scan)” whitepaper in CUDA SDK
  • “Prefix Sums and Their Applications”:
Prefix Sum (Scan) and Applications

- First try:
Prefix Sum (Scan) and Applications

• First try:

```plaintext
for d := 1 to \log_2 n do
    forall k in parallel do
        if k \geq 2^d then
            x[out][k] := x[in][k - 2^{d-1}] + x[in][k]
        else
            x[out][k] := x[in][k]
    swap(in, out)
```

• What's the number of operations?
Prefix Sum (Scan) and Applications

• First try:

```plaintext
for d := 1 to \log_2 n do
    forall k in parallel do
        if k \geq 2^d then
            x[out][k] := x[in][k - 2^{d-1}] + x[in][k]
        else
            x[out][k] := x[in][k]
    swap (in, out)
```

• What's the number of operations?

\[
\sum_{d=1}^{\log n} (n - 2^{d-1}) = O(n \log n)
\]

More than serial!
Prefix Sum (Scan) and Applications

Second try: think about how to utilize parallel reduction results:

\[ \text{sum}[v] = \text{sum}[L[v]] + \text{sum}[R[v]] \]
Prefix Sum (Scan) and Applications

- Second try: think about how to utilize parallel reduction results:

\[
\text{sum}[v] = \text{sum}[L[v]] + \text{sum}[R[v]]
\]
Prefix Sum (Scan) and Applications

- Second try: think about how to utilize parallel reduction results:

Now let's do “down sweep”!

```
Up Sweep

25
/   \
11   14
/   /   \
4   7   5   9
/   /   /   /   \
3  1  7  0  4  1  6  3

```

```
Down Sweep

0
/   \
0   11
/   /   \
0  3  4  11
/   /   /   /   \
0  4  11 15 16 22

```
Prefix Sum (Scan) and Applications

- Up-sweep
Prefix Sum (Scan) and Applications

- Down-sweep:
Prefix Sum (Scan) and Applications

• Down sweep algorithm:

\[
x[n - 1] := 0
\]
\[
\text{for } d := \log_2 n \text{ down to 0 do}
\]
\[
\text{for } k \text{ from 0 to } n - 1 \text{ by } 2^{d+1} \text{ in parallel do}
\]
\[
t := x[k + 2^d - 1]
\]
\[
x[k + 2^d - 1] := x[k + 2^{d+1} - 1]
\]
\[
x[k + 2^{d+1} - 1] := t + x[k + 2^{d+1} - 1]
\]

• Number of operations?
Prefix Sum (Scan) and Applications

- Down sweep algorithm:

```plaintext
x[n - 1] := 0

for d := \log_2 n \text{ down to } 0 \text{ do }
  for k from 0 \text{ to } n - 1 \text{ by } 2^{d+1} \text{ in parallel } \text{ do }
    t := x[k + 2^d - 1]
    x[k + 2^d - 1] := x[k + 2^{d+1} - 1]
    x[k + 2^{d+1} - 1] := t + x[k + 2^{d+1} - 1]
```

- Number of operations?

\[
\sum_{d=1}^{\log n} (2^d - 1) = O(n)
\]

Same with serial!
Prefix Sum (Scan) and Applications

• Advantages:

Although involves the same number of passes as the first try version, this version does involve less memory reads, and compute “in place”, which eliminates the need for double buffers.

• For Large Arrays:

Similar to parallel reduction: split the input into $p$ partitions → each processor sequentially scans each partition → then a single thread block processes the $p$ partial sums (up-down sweep algorithm) → finally distributes the results back to compute the final scan.
Prefix Sum (Scan) and Applications

- Proof of correctness:

“Prefix Sums and Their Applications”: