CS 140: Non-numerical Examples with Cilk++

• Divide and conquer paradigm for Cilk++
• Quicksort
• Mergesort

Thanks to Charles E. Leiserson for some of these slides
Work and Span (Recap)

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \quad T_\infty = \text{span}\]

*Also called critical-path length or computational depth.*

Speedup on \( p \) processors

\[ \frac{T_1}{T_p} \]

Parallelism

\[ \frac{T_1}{T_\infty} \]
Scheduling

- Cilk++ allows the programmer to express potential parallelism in an application.
- The Cilk++ scheduler maps strands onto processors dynamically at runtime.
- Since on-line schedulers are complicated, we’l explore the ideas with an off-line scheduler.
Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A strand is *ready* if all its predecessors have executed.
Greedy Scheduling

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

**Definition:** A strand is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ strands ready.
- Run any $P$. 
Greedy Scheduling

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

**Definition:** A strand is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ strands ready.
- Run any $P$.

**Incomplete step**
- $< P$ strands ready.
- Run all of them.
Analysis of Greedy

**Theorem** [G68, B75, BL93]. Any greedy scheduler achieves
\[ T_P \leq T_1/P + T_\infty. \]

**Proof.**
- # complete steps \( \leq T_1/P \), since each complete step performs \( P \) work.
- # incomplete steps \( \leq T_\infty \), since each incomplete step reduces the span of the unexecuted dag by 1.
Optimality of Greedy

**Corollary.** Any greedy scheduler achieves within a factor of 2 of optimal.

**Proof.** Let $T_P^*$ be the execution time produced by the optimal scheduler. Since $T_P^* \geq \max\{T_1/P, T_\infty\}$ by the Work and Span Laws, we have

\[
T_P \leq T_1/P + T_\infty \\
\leq 2 \cdot \max\{T_1/P, T_\infty\} \\
\leq 2T_P^*. 
\]
Corollary. Any greedy scheduler achieves near–perfect linear speedup whenever \( P \ll T_1/T_\infty \).

Proof. Since \( P \ll T_1/T_\infty \) is equivalent to \( T_\infty \ll T_1/P \), the Greedy Scheduling Theorem gives us

\[
T_P \leq T_1/P + T_\infty 
\approx T_1/P .
\]

Thus, the speedup is \( T_1/T_P \approx P \).

Definition. The quantity \( T_1/PT_\infty \) is called the parallel slackness.
Sorting

- Sorting is possibly the most frequently executed operation in computing!
- **Quicksort** is the fastest sorting algorithm in practice with an average running time of $O(N \log N)$, *(but $O(N^2)$ worst case performance)*
- **Mergesort** has worst case performance of $O(N \log N)$ for sorting $N$ elements
- Both based on the recursive *divide-and-conquer* paradigm
Parallelizing Quicksort

- Serial Quicksort sorts an array S as follows:
  - If the number of elements in S is 0 or 1, then return.
  - Pick any element v in S. Call this pivot.
  - Partition the set $S-\{v\}$ into two disjoint groups:
    - $S_1 = \{x \in S-\{v\} \mid x \leq v\}$
    - $S_2 = \{x \in S-\{v\} \mid x \geq v\}$
  - Return quicksort($S_1$) followed by v followed by quicksort($S_2$)

Not necessarily so!
Parallel Quicksort (Basic)

• The second recursive call to *qsort* does not depend on the results of the first recursive call
• We have an opportunity to speed up the call by making both calls in parallel.

```c++
template<typename T>
void qsort(T begin, T end) {
    if (begin != end) {
        T middle = partition(
            begin,
            end,
            bind2nd( less<typename iterator_traits<T>::value_type>(),
                      *begin )
        );
        cilk_spawn qsort(begin, middle);
        qsort(max(begin + 1, middle), end);
        cilk_sync;
    }
}
```
Performance

- ./qsort 500000 -cilk_set_worker_count 1 >> 0.122 seconds
- ./qsort 500000 -cilk_set_worker_count 4 >> 0.034 seconds
- Speedup = \( T_1 / T_4 = 0.122 / 0.034 = 3.58 \)

- ./qsort 50000000 -cilk_set_worker_count 1 >> 14.54 seconds
- ./qsort 50000000 -cilk_set_worker_count 4 >> 3.84 seconds
- Speedup = \( T_1 / T_4 = 14.54 / 3.84 = 3.78 \)
Measure Work/Span Empirically

- **cilkview ./qsort**
  Work : 6,008,068,218 instructions
  Span : 1,635,913,102 instructions
  Burdened span : 1,636,331,960 instructions
  Parallelism : 3.67

  ...

- Only the qsort function, exclude data generation
  (page 112 of the *cilk++ programmer’s guide*)
  qsort only ws 1 1.000000
  qsort only ws 2 1.764697
  ...
  qsort only ws 16 5.333179
  qsort only ws infinity 12.749447
Assume we have a “great” partitioner that always generates two balanced sets.
Analyzing Quicksort

• Work:
  \[ T_1(n) = 2T_1(n/2) + \Theta(n) \]
  \[ 2T_1(n/2) = 4T_1(n/4) + 2 \Theta(n/2) \]
  ....
  ....
  \[ n/2 T_1(2) = n T_1(1) + n/2 \Theta(2) \]
  \[ T_1(n) = \Theta(n \lg n) \]

• Span recurrence: \( T_\infty(n) = T_\infty(n/2) + \Theta(n) \)
  Solves to \( T_\infty(n) = \Theta(n) \)

\[ \text{Partitioning not parallel!} \]
Analyzing Quicksort

Parallelism: $\frac{T_1(n)}{T_\infty(n)} = \Theta(lg n)$

- Indeed, partitioning (i.e., constructing the array $S_1 = \{x \in S - \{v\} \mid x \leq v\}$) can be accomplished in parallel in time $\Theta(lg n)$
- Which gives a span $T_\infty(n) = \Theta(lg^2 n)$
- And parallelism $\Theta(n/lg n)$

- Basic parallel qsort can be found under $\text{cilkpath/examples/qsort}$
- Parallel partitioning might be a final project
The **Master Method** for solving recurrences applies to recurrences of the form

\[ T(n) = a \ T(n/b) + f(n) \]

where \( a \geq 1 \), \( b > 1 \), and \( f \) is asymptotically positive.

**IDEA:** Compare \( n^{\log_b a} \) with \( f(n) \).

* The unstated base case is \( T(n) = \Theta(1) \) for sufficiently small \( n \).
Master Method — CASE 1

\[ T(n) = a \cdot T(n/b) + f(n) \]

Specifically,

\[ f(n) = O(n^{\log_b a - \varepsilon}) \]

for some constant \( \varepsilon > 0 \).

**Solution:** \( T(n) = \Theta(n^{\log_b a}) \).
Master Method — **CASE 2**

\[
T(n) = a \ T(n/b) + f(n)
\]

Specifically, \( f(n) = \Theta(n^{\log_b a} \lg^k n) \) for some constant \( k \geq 0 \).

**Solution:** \( T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)) \).

Ex(qsort): \( a = 2, b=2, k=0 \) \( \Rightarrow T_1(n) = \Theta(n \ lg \ n) \)
Master Method — Case 3

\[ T(n) = a \, T(n/b) + f(n) \]

Specifically, \( f(n) = \Omega(n^{\log_b a} + \epsilon) \) for some constant \( \epsilon > 0 \), and \( f(n) \) satisfies the regularity condition that \( a \, f(n/b) \leq c \, f(n) \) for some constant \( c < 1 \).

Solution: \( T(n) = \Theta(f(n)) \)

Example: Span of qsort
Master Method Summary

\[ T(n) = a T(n/b) + f(n) \]

**CASE 1:** \( f(n) = O(n^{\log_b a - \epsilon}) \), constant \( \epsilon > 0 \)

\[ \Rightarrow T(n) = \Theta(n^{\log_b a}) . \]

**CASE 2:** \( f(n) = \Theta(n^{\log_b a \lg^k n}) \), constant \( k \geq 0 \)

\[ \Rightarrow T(n) = \Theta(n^{\log_b a \lg^{k+1} n}) . \]

**CASE 3:** \( f(n) = \Omega(n^{\log_b a + \epsilon}) \), constant \( \epsilon > 0 \), and regularity condition

\[ \Rightarrow T(n) = \Theta(f(n)) . \]
Mergesort is an example of a recursive sorting algorithm. It is based on the divide-and-conquer paradigm. It uses the merge operation as its fundamental component (which takes in two sorted sequences and produces a single sorted sequence).

Simulation of Mergesort

Drawback of mergesort: Not in-place (uses an extra temporary array)
template <typename T>
void Merge(T *C, T *A, T *B, int na, int nb) {
  while (na>0 && nb>0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na>0) {
    *C++ = *A++; na--;
  }
  while (nb>0) {
    *C++ = *B++; nb--;
  }
}

Time to merge n elements = \Theta(n).
template <typename T>
void MergeSort(T *B, T *A, int n) {
if (n==1) {
    B[0] = A[0];
} else {
    T* C = new T[n];
    cilk_spawn MergeSort(C, A, n/2);
    MergeSort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
    delete[] C;
}
}

Parallel Merge Sort

A: input (unsorted)
B: output (sorted)
C: temporary

merge
merge
merge
Work of Merge Sort

```cpp
template <typename T>
void MergeSort(T *B, T *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        T* C = new T[n];
        cilk_spawn MergeSort(C, A, n/2);
        MergeSort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        Merge(B, C, C+n/2, n/2, n-n/2);
        delete[] C;
    }
}
```

**CASE 2:**

\[
\begin{align*}
\text{n}^{\log_b a} &= \text{n}^{\log_2 2} = \text{n} \\
\text{f}(\text{n}) &= \Theta(\text{n}^{\log_b a} \log^0 \text{n})
\end{align*}
\]

**Work:**

\[
\begin{align*}
T_1(n) &= 2T_1(n/2) + \Theta(n) \\
&= \Theta(n \log n)
\end{align*}
\]
template <typename T>
void MergeSort(T *B, T *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        T* C = new T[n];
        cilk_spawn
        MergeSort(C, A, n/2);
        MergeSort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        Merge(B, C, C+n/2, n/2, n-n/2);
        delete[] C;
    }
}

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(n)$

$= \Theta(n)$
Parallelism of Merge Sort

**Work:** \( T_1(n) = \Theta(n \lg n) \)

**Span:** \( T_\infty(n) = \Theta(n) \)

**Parallelism:** \( \frac{T_1(n)}{T_\infty(n)} = \Theta(lg n) \)

*PUNY!*

We need to parallelize the merge!
Parallel Merge

**KEY IDEA:** If the total number of elements to be merged in the two arrays is $n = na + nb$, the total number of elements in the larger of the two recursive merges is at most $(3/4)n$. 

Throw away at least $na/2 \geq n/4$
Parallel Merge

template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    P_Merge(C, B, A, nb, na);
  } else if (na==0) {
    return;
  } else {
    int ma = na/2;
    int mb = BinarySearch(A[ma], B, nb);
    C[ma+mb] = A[ma];
    cilk_spawn P_Merge(C, A, B, ma, mb);
    P_Merge(C+ma+mb+1, A+ma+1, B+mb, na-ma-1, nb-mb);
    cilk_sync;
  }
}

Coarsen base cases for efficiency.
Span of Parallel Merge

```c
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
    if (na < nb) {
        int mb = BinarySearch(A[ma], B, nb);
        C[ma+mb] = A[ma];
        cilk_spawn P_Merge(C, A, B, ma, mb);
        P_Merge(C+ma+mb+1, A+ma+1, B+mb, n-
        ma-
        1, nb-
        mb);
        cilk_sync;
    }
}
```

**CASE 2:**
\[ n^{\log_{4/3} 1} = 1 \]
\[ f(n) = \Theta(n^{\log_{4/3} 1} \log 1 n) \]

**Span:**
\[ T_\infty(n) = T_\infty(3n/4) + \Theta(\log n) \]
\[ = \Theta(\log^2 n) \]
template<typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
    if (na < nb) {
        int mb = BinarySearch(A[ma], B, nb);
        C[ma+mb] = A[ma];
        cilk_spawn P_Merge(C, A, B, ma, mb);
        P_Merge(C+ma+mb+1, A+ma+1, B+mb, na-ma-1, nb-mb);
        cilk_sync;
    }
}

**Work:** \( T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(lg n) \),
where \( 1/4 \leq \alpha \leq 3/4 \).

**Claim:** \( T_1(n) = \Theta(n) \).
Analysis of Work Recurrence

**Work:** \( T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(lg\ n), \)

where \( 1/4 \leq \alpha \leq 3/4. \)

**Substitution method:** Inductive hypothesis is \( T_1(k) \leq c_1 k - c_2 lg\ k, \) where \( c_1, c_2 > 0. \) Prove that the relation holds, and solve for \( c_1 \) and \( c_2. \)

\[
T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(lg\ n) \\
\leq c_1(\alpha n) - c_2 lg(\alpha n) \\
+ c_1(1-\alpha)n - c_2 lg((1-\alpha)n) + \Theta(lg\ n)
\]
Analysis of Work Recurrence

**Work:** \( T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \),
where \( 1/4 \leq \alpha \leq 3/4 \).

\[
T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \\
\leq c_1(\alpha n) - c_2 \lg(\alpha n) \\
+ c_1(1-\alpha)n - c_2 \lg((1-\alpha)n) + \Theta(\lg n)
\]
Analysis of Work Recurrence

**Work:**\[ T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n), \]
where \( 1/4 \leq \alpha \leq 3/4. \)

\[
T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \\
\leq c_1(\alpha n) - c_2 \lg(\alpha n) \\
+ c_1((1-\alpha)n) - c_2 \lg((1-\alpha)n) + \Theta(\lg n) \\
\leq c_1 n - c_2 \lg(\alpha n) - c_2 \lg((1-\alpha)n) + \Theta(\lg n) \\
\leq c_1 n - c_2 ( \lg(\alpha(1-\alpha)) + 2 \lg n ) + \Theta(\lg n) \\
\leq c_1 n - c_2 \lg n \\
- (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
\leq c_1 n - c_2 \lg n
\]

by choosing \( c_2 \) large enough. Choose \( c_1 \) large enough to handle the base case.
Parallelism of P_Merge

**Work:** \( T_1(n) = \Theta(n) \)

**Span:** \( T_\infty(n) = \Theta(lg^2 n) \)

**Parallelism:** \( \frac{T_1(n)}{T_\infty(n)} = \Theta(n/lg^2 n) \)
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        T C[n];
        cilk_spawn P_MergeSort(C, A, n/2);
        P_MergeSort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        P_Merge(B, C, C+n/2, n/2, n-n/2);
    }
}

CASE 2:
\[ n^{\log_b a} = n^{\log_2 2} = n \]
\[ f(n) = \Theta(n^{\log_b a \log^0 n}) \]

Work: \[ T_1(n) = 2T_1(n/2) + \Theta(n) \]
\[ = \Theta(n \log n) \]
Parallel Merge Sort

template <typename T>
void P_MergeSort(T *B, T *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        T C[n];
        cilk_spawn P_MergeSort(C, A, n/2);
        P_MergeSort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        P_Merge(B, C, C+n/2, n/2, n-n/2);
    }
}

CASE 2:
\[ n^{\log_b a} = n^{\log_2 1} = 1 \]
\[ f(n) = \Theta(n^{\log_b a} \lg^2 n) \]

Span: \[ T_{\infty}(n) = T_{\infty}(n/2) + \Theta(\lg^2 n) = \Theta(\lg^3 n) \]
Parallelism of P_MergeSort

**Work:** \( T_1(n) = \Theta(n \lg n) \)

**Span:** \( T_\infty(n) = \Theta(\lg^3 n) \)

**Parallelism:** \( \frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n) \)