CS 240A : Examples with Cilk++

- Divide & Conquer Paradigm for Cilk++
- Solving recurrences
- Sorting: Quicksort and Mergesort
- Graph traversal: Breadth–First Search

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.*
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
Basic Quicksort sorting an array $S$ works 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 $S \setminus \{v\}$ into two disjoint groups:
  - $S_1 = \{x \in S \setminus \{v\} \mid x \leq v\}$
  - $S_2 = \{x \in S \setminus \{v\} \mid x \geq v\}$
- Return $\text{quicksort}(S_1)$ followed by $v$ followed by $\text{quicksort}(S_2)$
QUICKSORT

Partition around Pivot

13 45 34 14 56
32 21 31 78

13 31 21
14 32

45 56
78
QUICKSORT

Quicksort recursively

13 14 21 31 32

45 56 78

13 14 21 31 32

45 56 78
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 $\text{quicksort}(S_1)$ followed by $v$ followed by $\text{quicksort}(S_2)$

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.

```cpp
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`  
  `>> 0.083 seconds`
- `. `/qsort 500000 -cilk_set_worker_count 16`  
  `>> 0.014 seconds`
- `Speedup = T_1 / T_{16} = 0.083 / 0.014 = 5.93`

- `. `/qsort 50000000 -cilk_set_worker_count 1`  
  `>> 10.57 seconds`
- `. `/qsort 50000000 -cilk_set_worker_count 16`  
  `>> 1.58 seconds`
- `Speedup = T_1 / T_{16} = 10.57 / 1.58 = 6.67`
Measure Work/Span Empirically

- `cilkscreen -w ./qsort 50000000`
  
  \[
  \begin{align*}
  \text{Work} & = 21593799861 \\
  \text{Span} & = 1261403043 \\
  \text{Burdened span} & = 1261600249 \\
  \text{Parallelism} & = 17.1189 \\
  \text{Burdened parallelism} & = 17.1162 \\
  \#\text{Spawn} & = 50000000 \\
  \#\text{Atomic instructions} & = 14
  \end{align*}
  \]

- `cilkscreen -w ./qsort 5000000`
  
  \[
  \begin{align*}
  \text{Work} & = 178835973 \\
  \text{Span} & = 14378443 \\
  \text{Burdened span} & = 14525767 \\
  \text{Parallelism} & = 12.4378 \\
  \text{Burdened parallelism} & = 12.3116 \\
  \#\text{Spawn} & = 50000000 \\
  \#\text{Atomic instructions} & = 8
  \end{align*}
  \]

```cpp
workspan ws;
ws.start();
sample_qsort(a, a + n);
ws.stop();
ws.report(std::cout);
```
Analyzing Quicksort

Assume we have a “great” partitioner that always generates two balanced sets.
Analyzing Quicksort

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

  \[ T_1(n) = \Theta(n \log n) \]

- 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 \( S_1 = \{ x \in S \setminus \{v\} \mid x \leq v \} \)) 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 \$cilkpath/examples/qsort\n
Not much!

Way better!
The Master Method (Optional)

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

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

\[ n^{\log_b a} \approx f(n) \]

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

\[ T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \]

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

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

Example: Span of qsort
Master Method Summary

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

**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
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;
    }
}

Work: \[ T_1(n) = 2T_1(n/2) + \Theta(n) \]
\[ = \Theta(n \lg n) \]
Span of Merge Sort

\[ T_{∞}(n) = T_{∞}(\frac{n}{2}) + Θ(n) \]

\[ = Θ(n) \]

CASE 3:
\[ n^{\log_{b}a} = n^{\log_{2}1} = 1 \]
\[ f(n) = Θ(n) \]
Parallelism of Merge Sort

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

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

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

*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 \( (3/4) n \).
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, na-ma-1, nb-mb);
        cilk_sync;
    }
}
```

**CASE 2:**

\[ n^{\log_b a} = n^{\log_{4/3} 1} = 1 \]

\[ f(n) = \Theta(n^{\log_b a \log \log n}) \]

**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:** \( T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\log n) \), 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: \( T_\infty(n) = \Theta(\log^2 n) \)

Parallelism: \( \frac{T_1(n)}{T_\infty(n)} = \Theta(n/\log^2 n) \)
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:**
\[ n^{\log_b a} = n^{\log_2 2} = n \]
\[ f(n) = \Theta(n^{\log_b a} \lg^0 n) \]

**Work:**
\[ T_1(n) = 2T_1(n/2) + \Theta(n) \]
\[ = \Theta(n \lg n) \]
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:
\[ n^{\log_b a} = n^{\log_2 1} = 1 \]
\[ f(n) = \Theta(n^{\log_b a \log^2 n}) \]

**Span:** \( T_\infty(n) = T_\infty(n/2) + \Theta(\log^2 n) \)

\[ = \Theta(\log^3 n) \]
Parallelism of P_MergeSort

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

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

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

- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$

Graph: $G(E, V)$

- $E$: Set of edges (size m)
- $V$: Set of vertices (size n)
Breadth First Search

- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$
Breadth First Search

- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$
Breadth First Search

- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$
Breadth First Search

- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$
Breadth First Search

- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$
Breadth First Search

- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$
Breadth First Search

- Who is parent(19)?
  - If we use a queue for expanding the frontier?
  - Does it actually matter?

```
Level 1
1 2 3 4

5 6 7 8

9 10 11 12

16 17 18 19

Level 2

Level 3

Level 4

Level 5

Level 6
```
Parallel BFS

- Way #1: A custom reducer

```c
void BFS(Graph *G, Vertex root)
{
    Bag<Vertex> frontier(root);
    while ( ! frontier.isEmpty() )
    {
        cilk::hyperobject< Bag<Vertex> > succbag();
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() )
            {
                if( ! v.unvisited() )
                    succbag() += v;
            }
        }
        frontier = succbag.getValue();
    }
}
```

Bag<T> has an associative reduce function that merges two sets

```
operator+=(Vertex & rhs)
also marks rhs “visited”
```
Parallel BFS

- Way #2: Concurrent writes + List reducer

```c
void BFS(Graph *G, Vertex root)
{
    list<Vertex> frontier(root);
    Vertex * parent = new Vertex[n];
    while ( ! frontier.isEmpty() )
    {
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() )
            {
                if ( ! v.visited() )
                    parent[v] = frontier[i];
            }
        }
    }
}
```

An intentional data race

How to generate the new frontier?
Parallel BFS

void BFS(Graph *G, Vertex root)
{
    ...
    while ( ! frontier.isEmpty() ) {
        ...
        hyperobject< reducer_list_append<Vertex> >
        succlist();
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() )
            {
                if ( parent[v] == frontier[i] )
                {
                    succlist.push_back(v);
                    v.visit();// Mark “visited”
                }
            }
        }
    }
    frontier = succlist.getValue();
}
Parallel BFS

- Each level is explored with $\Theta(1)$ span
- Graph $G$ has at most $d$, at least $d/2$ levels
  - Depending on the location of root
  - $d = \text{diameter}(G)$

**Work:** $T_1(n) = \Theta(m+n)$

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

**Parallelism:** $\frac{T_1(n)}{T_\infty(n)} = \Theta((m+n)/d)$
Parallel BFS Caveats

- \( d \) is usually small
- \( d = \log(n) \) for scale-free graphs
  - But the degrees are not bounded 😞
- Parallel scaling will be memory-bound
- Lots of burdened parallelism,
  - Loops are skinny
  - Especially to the root and leaves of BFS-tree
- You are not “expected” to parallelize BFS part of Homework #5
  - You may do it for “extra credit” though 😊