Analysis of Multithreaded Algorithms

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS 4435 - CS 9624
Plan

1. Review of Complexity Notions
2. Divide-and-Conquer Recurrences
3. Matrix Multiplication
4. Merge Sort
5. Tableau Construction
Plan

1. Review of Complexity Notions
2. Divide-and-Conquer Recurrences
3. Matrix Multiplication
4. Merge Sort
5. Tableau Construction
Orders of magnitude

Let $f$, $g$ et $h$ be functions from $\mathbb{N}$ to $\mathbb{R}$.

- We say that $g(n)$ is in the **order of magnitude** of $f(n)$ and we write $f(n) \in \Theta(g(n))$ if there exist two strictly positive constants $c_1$ and $c_2$ such that for $n$ big enough we have

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n). \quad (1)$$

- We say that $g(n)$ is an **asymptotic upper bound** of $f(n)$ and we write $f(n) \in \mathcal{O}(g(n))$ if there exists a strictly positive constants $c_2$ such that for $n$ big enough we have

$$0 \leq f(n) \leq c_2 g(n). \quad (2)$$

- We say that $g(n)$ is an **asymptotic lower bound** of $f(n)$ and we write $f(n) \in \Omega(g(n))$ if there exists a strictly positive constants $c_1$ such that for $n$ big enough we have

$$0 \leq c_1 g(n) \leq f(n). \quad (3)$$
Examples

- With \( f(n) = \frac{1}{2}n^2 - 3n \) and \( g(n) = n^2 \) we have \( f(n) \in \Theta(g(n)) \). Indeed we have
  \[
  c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2. \tag{4}
  \]
  for \( n \geq 12 \) with \( c_1 = \frac{1}{4} \) and \( c_2 = \frac{1}{2} \).
- Assume that there exists a positive integer \( n_0 \) such that \( f(n) > 0 \) and \( g(n) > 0 \) for every \( n \geq n_0 \). Then we have
  \[
  \max(f(n), g(n)) \in \Theta(f(n) + g(n)). \tag{5}
  \]
  Indeed we have
  \[
  \frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n)) \leq (f(n) + g(n)). \tag{6}
  \]
- Assume \( a \) and \( b \) are positive real constants. Then we have
  \[
  (n + a)^b \in \Theta(n^b). \tag{7}
  \]
  Indeed for \( n \geq a \) we have
Properties

- \( f(n) \in \Theta(g(n)) \) holds iff \( f(n) \in \mathcal{O}(g(n)) \) and \( f(n) \in \Omega(g(n)) \) hold together.
- Each of the predicates \( f(n) \in \Theta(g(n)) \), \( f(n) \in \mathcal{O}(g(n)) \) and \( f(n) \in \Omega(g(n)) \) define a reflexive and transitive binary relation among the \( \mathbb{N} \)-to-\( \mathbb{R} \) functions. Moreover \( f(n) \in \Theta(g(n)) \) is symmetric.
- We have the following **transposition formula**

\[
f(n) \in \mathcal{O}(g(n)) \iff g(n) \in \Omega(f(n)). \quad (9)
\]

In practice \( \in \) is replaced by \( = \) in each of the expressions \( f(n) \in \Theta(g(n)) \), \( f(n) \in \mathcal{O}(g(n)) \) and \( f(n) \in \Omega(g(n)) \). Hence, the following

\[
f(n) = h(n) + \Theta(g(n)) \quad (10)
\]

means

\[
f(n) - h(n) \in \Theta(g(n)). \quad (11)
\]
Another example

Let us give another fundamental example. Let $p(n)$ be a (univariate) polynomial with degree $d > 0$. Let $a_d$ be its leading coefficient and assume $a_d > 0$. Then we have

1. if $k \geq d$ then $p(n) \in \mathcal{O}(n^k)$,
2. if $k \leq d$ then $p(n) \in \Omega(n^k)$,
3. if $k = d$ then $p(n) \in \Theta(n^k)$.

Exercise: Prove the following

$$
\sum_{k=1}^{n} k \in \Theta(n^2). \tag{12}
$$
Plan

1. Review of Complexity Notions
2. Divide-and-Conquer Recurrences
3. Matrix Multiplication
4. Merge Sort
5. Tableau Construction
Divide-and-Conquer Algorithms

Divide-and-conquer algorithms proceed as follows.

**Divide** the input problem into sub-problems.

**Conquer** on the sub-problems by solving them directly if they are small enough or proceed recursively.

**Combine** the solutions of the sub-problems to obtain the solution of the input problem.

**Equation satisfied by** $T(n)$. Assume that the size of the input problem increases with an integer $n$. Let $T(n)$ be the time complexity of a divide-and-conquer algorithm to solve this problem. Then $T(n)$ satisfies an equation of the form:

$$T(n) = a \cdot T(n/b) + f(n).$$

(13)

where $f(n)$ is the cost of the combine-part, $a \geq 1$ is the number of recursively calls and $n/b$ with $b > 1$ is the size of a sub-problem.
Labeled tree associated with the equation. Assume $n$ is a power of $b$, say $n = b^p$. To solve the equation

$$T(n) = a \ T(n/b) + f(n).$$

we can associate a labeled tree $A(n)$ to it as follows.

1. If $n = 1$, then $A(n)$ is reduced to a single leaf labeled $T(1)$.
2. If $n > 1$, then the root of $A(n)$ is labeled by $f(n)$ and $A(n)$ possesses a labeled sub-trees all equal to $A(n/b)$.

The labeled tree $A(n)$ associated with $T(n) = a \ T(n/b) + f(n)$ has height $p + 1$. Moreover the sum of its labels is $T(n)$. 
Solving divide-and-conquer recurrences (1/2)
Solving divide-and-conquer recurrences (2/2)

IDEA: Compare $n^{\log_b a}$ with $f(n)$.
Master Theorem: case $n^{\log_b a} \gg f(n)$

Specifically, $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.

$T(n) = \Theta(n^{\log_b a})$
Master Theorem: case \( f(n) \in \Theta(n^{\log_b a} \log^k n) \)

Specifically, \( f(n) = \Theta(n^{\log_b a} \lg^k n) \) for some constant \( k \geq 0 \).
Master Theorem: case where $f(n) \gg n^{\log_b a}$

Specifically, $f(n) = \Omega(n^{\log_b a} + \epsilon)$ for some constant $\epsilon > 0$.*

$T(n) = \Theta(f(n))$

*and $f(n)$ satisfies the regularity condition that $a f(n/b) \leq cf(n)$ for some constant $c < 1$. 
More examples

- Consider the relation:

\[ T(n) = 2 T\left(\frac{n}{2}\right) + n^2. \]  \hfill (14)

We obtain:

\[ T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots + \frac{n^2}{2^p} + n T(1). \]  \hfill (15)

Hence we have:

\[ T(n) \in \Theta(n^2). \]  \hfill (16)

- Consider the relation:

\[ T(n) = 3 T\left(\frac{n}{3}\right) + n. \]  \hfill (17)

We obtain:

\[ T(n) \in \Theta(\log_3(n)n). \]  \hfill (18)
Master Theorem when $b = 2$

Let $a > 0$ be an integer and let $f, T : \mathbb{N} \rightarrow \mathbb{R}_+$ be functions such that

(i) $f(2n) \geq 2f(n)$ and $f(n) \geq n$.

(ii) If $n = 2^p$ then $T(n) \leq aT(n/2) + f(n)$.

Then for $n = 2^p$ we have

(1) if $a = 1$ then

$$T(n) \leq (2 - 2/n) f(n) + T(1) \in O(f(n)),$$

(2) if $a = 2$ then

$$T(n) \leq f(n) \log_2(n) + T(1) n \in O(\log_2(n) f(n)),$$

(3) if $a \geq 3$ then

$$T(n) \leq \frac{2}{a - 2} \left( n^{\log_2(a) - 1} - 1 \right) f(n) + T(1) n^{\log_2(a)} \in O(f(n) n^{\log_2(a) - 1})$$
Master Theorem when $b = 2$

Indeed

\[
T(2^p) \leq a T(2^{p-1}) + f(2^p) \\
\leq a \left[ a T(2^{p-2}) + f(2^{p-1}) \right] + f(2^p) \\
= a^2 T(2^{p-2}) + a f(2^{p-1}) + f(2^p) \\
\leq a^2 \left[ a T(2^{p-3}) + f(2^{p-2}) \right] + a f(2^{p-1}) + f(2^p) \\
= a^3 T(2^{p-3}) + a^2 f(2^{p-2}) + a f(2^{p-1}) + f(2^p) \\
\leq a^p T(s1) + \sum_{j=0}^{p-1} a^j f(2^{p-j})
\]
Master Theorem when $b = 2$

Moreover

$$f(2^p) \geq 2f(2^{p-1})$$
$$f(2^p) \geq 2^2 f(2^{p-2})$$
$$\vdots \quad \vdots \quad \vdots$$
$$f(2^p) \geq 2^j f(2^{p-j})$$

Thus

$$\sum_{j=0}^{p-1} a^j f(2^{p-j}) \leq f(2^p) \sum_{j=0}^{p-1} \left(\frac{a}{2}\right)^j.$$
Master Theorem when $b = 2$

Hence

$$T(2^p) \leq a^p T(1) + f(2^p) \sum_{j=0}^{p-1} \left(\frac{a}{2}\right)^j.$$ \hspace{1cm} (25)

For $a = 1$ we obtain

$$T(2^p) \leq T(1) + f(2^p) \sum_{j=0}^{p-1} \left(\frac{1}{2}\right)^j$$

$$= T(1) + f(2^p) \left(\frac{1}{2^p - 1}\right)$$

$$= T(1) + f(n) \left(2 - 2/n\right).$$ \hspace{1cm} (26)

For $a = 2$ we obtain

$$T(2^p) \leq 2^p T(1) + f(2^p) p$$

$$= n T(1) + f(n) \log_2(n).$$ \hspace{1cm} (27)
Master Theorem cheat sheet

For \( a \geq 1 \) and \( b > 1 \), consider again the equation

\[
T(n) = a \, T(n/b) + f(n).
\] (28)

- for any \( \varepsilon > 0 \) we have:

\[
f(n) \in O(n^{\log_b a - \varepsilon}) \implies T(n) \in \Theta(n^{\log_b a})
\] (29)

- for any \( k \geq 0 \) we have:

\[
f(n) \in \Theta(n^{\log_b a \log^k n}) \implies T(n) \in \Theta(n^{\log_b a \log^{k+1} n})
\] (30)

- for any \( \varepsilon > 0 \) we have:

\[
f(n) \in \Omega(n^{\log_b a + \varepsilon}) \implies T(n) \in \Theta(f(n))
\] (31)
Master Theorem quizz!

- $T(n) = 4T(n/2) + n$
- $T(n) = 4T(n/2) + n^2$
- $T(n) = 4T(n/2) + n^3$
- $T(n) = 4T(n/2) + n^2/\log n$
Plan

1. Review of Complexity Notions
2. Divide-and-Conquer Recurrences
3. Matrix Multiplication
4. Merge Sort
5. Tableau Construction
We will study three approaches:

- a naive and iterative one
- a divide-and-conquer one
- a divide-and-conquer one with memory management consideration
Naive iterative matrix multiplication

cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
        for (int k=0; k<n; ++k {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

• **Work**: ?
• **Span**: ?
• **Parallelism**: ?
Naive iterative matrix multiplication

cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
        for (int k=0; k<n; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

- **Work**: $\Theta(n^3)$
- **Span**: $\Theta(n)$
- **Parallelism**: $\Theta(n^2)$
The divide-and-conquer approach is simply the one based on blocking, presented in the first lecture.
Divide-and-conquer matrix multiplication

// C ← C + A * B
void MMult(T *C, T *A, T *B, int n, int size) {
    T *D = new T[n*n];
    //base case & partition matrices
    cilk_spawn MMult(C11, A11, B11, n/2, size);
    cilk_spawn MMult(C12, A11, B12, n/2, size);
    cilk_spawn MMult(C22, A21, B12, n/2, size);
    cilk_spawn MMult(C21, A21, B11, n/2, size);
    cilk_spawn MMult(D11, A12, B21, n/2, size);
    cilk_spawn MMult(D12, A12, B22, n/2, size);
    cilk_spawn MMult(D22, A22, B22, n/2, size);
    cilk_sync;
    MAdd(C, D, n, size); // C += D;
    delete[] D;
}

Work? Span? Parallelism?
Divide-and-conquer matrix multiplication

```c
void MMult(T *C, T *A, T *B, int n, int size) {
    T *D = new T[n*n];
    //base case & partition matrices
    cilk_spawn MMult(C11, A11, B11, n/2, size);
    cilk_spawn MMult(C12, A11, B12, n/2, size);
    cilk_spawn MMult(C22, A21, B12, n/2, size);
    cilk_spawn MMult(C21, A21, B11, n/2, size);
    cilk_spawn MMult(D11, A12, B21, n/2, size);
    cilk_spawn MMult(D12, A12, B22, n/2, size);
    cilk_spawn MMult(D22, A22, B22, n/2, size);
    MMult(D21, A22, B21, n/2, size);
    cilk_sync; MAdd(C, D, n, size); // C += D;
    delete[] D;  }
```

- $A_p(n)$ and $M_p(n)$: times on $p$ proc. for $n \times n$ \texttt{ADD} and \texttt{MULT}.
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$
- $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$
- $M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$
- $M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$
- $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$
template <typename T>
void MMult2(T *C, T *A, T *B, int n, int size) {
    // base case & partition matrices
    cilk_spawn MMult2(C11, A11, B11, n/2, size);
    cilk_spawn MMult2(C12, A11, B12, n/2, size);
    cilk_spawn MMult2(C22, A21, B12, n/2, size);
    cilk_spawn MMult2(C21, A21, B11, n/2, size);
    cilk_sync;
    cilk_spawn MMult2(C11, A12, B21, n/2, size);
    cilk_spawn MMult2(C12, A12, B22, n/2, size);
    cilk_spawn MMult2(C22, A22, B22, n/2, size);
    cilk_spawn MMult2(C21, A22, B21, n/2, size);
    cilk_sync;  }

Work ? Span ? Parallelism ?
Divide-and-conquer matrix multiplication: No temporaries!

```cpp
template <typename T>
void MMult2(T *C, T *A, T *B, int n, int size) {
    //base case & partition matrices
    cilk_spawn MMult2(C11, A11, B11, n/2, size);
    cilk_spawn MMult2(C12, A11, B12, n/2, size);
    cilk_spawn MMult2(C22, A21, B12, n/2, size);
    MMult2(C21, A21, B11, n/2, size);
    cilk_sync;
    cilk_spawn MMult2(C11, A12, B21, n/2, size);
    cilk_spawn MMult2(C12, A12, B22, n/2, size);
    cilk_spawn MMult2(C22, A22, B22, n/2, size);
    MMult2(C21, A22, B21, n/2, size);
    cilk_sync; }
```

- $MA_p(n)$: time on $p$ proc. for $n \times n$ MULT-ADD.
- $MA_1(n) = \Theta(n^3)$
- $MA_\infty(n) = 2MA_\infty(n/2) + \Theta(1) = \Theta(n)$
- $MA_1(n)/MA_\infty(n) = \Theta(n^2)$
- Besides, saving space often saves time due to hierarchical memory.
Plan

1. Review of Complexity Notions
2. Divide-and-Conquer Recurrences
3. Matrix Multiplication
4. Merge Sort
5. Tableau Construction
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 for merging $n$ elements is $\Theta(n)$. 

(Moreno Maza)
Merge sort

merge

merge

merge

3 4 12 14 19 21 33 46
3 12 19 46 4 14 21 33
3 19 12 46 4 33 14 21
19 3 12 46 33 4 21 14
Parallel merge sort with serial merge

```
template <typename T>
void MergeSort(T *B, T *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        T* C[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);
    }
}
```
template <typename T>
void MergeSort(T *B, T *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        T* C[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);
    }
}

- $T_1(n) = 2T_1(n/2) + \Theta(n)$ thus $T_1(n) = \Theta(n \log n)$.
- $T_\infty(n) = T_\infty(n/2) + \Theta(n)$ thus $T_\infty(n) = \Theta(n)$.
- $T_1(n)/T_\infty(n) = \Theta(\log n)$. **Puny parallelism!**
- We need to parallelize the merge!
Parallel merge

Idea: if the total number of elements to be sorted in \( n = n_a + n_b \) then the maximum number of elements in any of the two merges is at most \( 3n/4 \).
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;
    }
}

- One should coarse the base case for efficiency.
- **Work? Span?**

(Moreno Maza)
Parallel merge

```c++
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; }
}
```

- Let $PM_p(n)$ be the $p$-processor running time of P-MERGE.
- In the worst case, the span of P-MERGE is
  $$PM_{\infty}(n) \leq PM_{\infty}(3n/4) + \Theta(\lg n) = \Theta(\lg^2 n)$$
- The worst-case work of P-MERGE satisfies the recurrence
  $$PM_1(n) \leq PM_1(\alpha n) + PM_1((1 - \alpha)n) + \Theta(\lg n)$$
Analyzing parallel merge

- Recall $PM_1(n) \leq PM_1(\alpha n) + PM_1((1 - \alpha)n) + \Theta(\lg n)$ for some $1/4 \leq \alpha \leq 3/4$.

- To solve this **hairy equation** we use the substitution method.

- We assume there exist some constants $a, b > 0$ such that $PM_1(n) \leq an - b \lg n$ holds for all $1/4 \leq \alpha \leq 3/4$.

- After substitution, this hypothesis implies:
  
  $$PM_1(n) \leq an - b \lg n - b \lg n + \Theta(\lg n).$$

- We can pick $b$ large enough such that we have $PM_1(n) \leq an - b \lg n$ for all $1/4 \leq \alpha \leq 3/4$ and all $n > 1/4$.

- Then pick $a$ large enough to satisfy the base conditions.

- Finally we have $PM_1(n) = \Theta(n)$.
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);
  }
}

- Work?
- Span?
Parallel merge sort with parallel merge

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);
    }
}

The work satisfies $T_1(n) = 2T_1(n/2) + \Theta(n)$ (as usual) and we have $T_1(n) = \Theta(n\log(n))$.

The worst case critical-path length of the MERGE-SORT now satisfies

\[ T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n) = \Theta(\lg^3 n) \]

The parallelism is now $\Theta(n \lg n)/\Theta(\lg^3 n) = \Theta(n/\lg^2 n)$. 
Plan

1. Review of Complexity Notions
2. Divide-and-Conquer Recurrences
3. Matrix Multiplication
4. Merge Sort
5. Tableau Construction
Constructing a tableau $A$ satisfying a relation of the form:

$$A[i,j] = R(A[i - 1,j], A[i - 1,j - 1], A[i,j - 1]).$$  \(32\)

The work is $\Theta(n^2)$. 

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
</tr>
<tr>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
</tr>
<tr>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
</tr>
</tbody>
</table>
Recursive construction

- \( T_1(n) = 4T_1(n/2) + \Theta(1) \), thus \( T_1(n) = \Theta(n^2) \).
- \( T_\infty(n) = 3T_\infty(n/2) + \Theta(1) \), thus \( T_\infty(n) = \Theta(n^{\log_2 3}) \).
- **Parallelism:** \( \Theta(n^{2-\log_2 3}) = \Omega(n^{0.41}) \).
A more parallel construction

- $T_1(n) = 9T_1(n/3) + \Theta(1)$, thus $T_1(n) = \Theta(n^2)$.
- $T_\infty(n) = 5T_\infty(n/3) + \Theta(1)$, thus $T_\infty(n) = \Theta(n^{\log_3 5})$.
- **Parallelism**: $\Theta(n^{2 - \log_3 5}) = \Omega(n^{0.53})$.
- This nine-way d-n-c has more parallelism than the four way but exhibits more cache complexity (more on this later).
Acknowledgements

Charles E. Leiserson (MIT) for providing me with the sources of its lecture notes.

Matteo Frigo (Intel) for supporting the work of my team with Cilk++ and offering us the next lecture.

Yuzhen Xie (UWO) for helping me with the images used in these slides.

Liyun Li (UWO) for generating the experimental data.