The Fork-Join Model and its Implementation in Cilk

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS 4402 - CS 9535
Plan

Parallelism Complexity Measures

cilk_for Loops

Scheduling Theory and Implementation

Measuring Parallelism in Practice

Anticipating parallelization overheads

Announcements
Plan

Parallelism Complexity Measures

cilk_for Loops

Scheduling Theory and Implementation

Measuring Parallelism in Practice

Anticipating parallelization overheads

Announcements
The fork-join parallelism model

```c
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x+y);
    }
}
```

Example:
```
fib(4)
```

"Processor oblivious"

The computation dag unfolds dynamically.

We shall also call this model **multithreaded parallelism**.
a strand is a maximal sequence of instructions that ends with a spawn, sync, or return (either explicit or implicit) statement.

At runtime, the spawn relation causes procedure instances to be structured as a rooted tree, called spawn tree or parallel instruction stream, where dependencies among strands form a dag.
We define several performance measures. We assume an ideal situation: no cache issues, no interprocessor costs:

- $T_p$ is the minimum running time on $p$ processors
- $T_1$ is called the work, that is, the sum of the number of instructions at each node.
- $T_\infty$ is the minimum running time with infinitely many processors, called the span.
The critical path length

Assuming all strands run in unit time, the longest path in the DAG is equal to $T_\infty$. For this reason, $T_\infty$ is also referred to as the **critical path length**.
We have: \( T_p \geq T_1/p \).

Indeed, in the best case, \( p \) processors can do \( p \) works per unit of time.
We have: $T_p \geq T_\infty$.

Indeed, $T_p < T_\infty$ contradicts the definitions of $T_p$ and $T_\infty$. 
Speedup on $p$ processors

- $T_1 / T_p$ is called the speedup on $p$ processors

A parallel program execution can have:

- **linear speedup**: $T_1 / T_p = \Theta(p)$

- **superlinear speedup**: $T_1 / T_p = \omega(p)$ (not possible in this model, though it is possible in others)

- **sublinear speedup**: $T_1 / T_p = o(p)$
Because the **Span Law** dictates that $T_P \geq T_\infty$, the maximum possible speedup given $T_1$ and $T_\infty$ is

$$T_1 / T_\infty = \textit{parallelism}$$

= the average amount of work per step along the span.
The Fibonacci example (1/2)

- For \( \text{Fib}(4) \), we have \( T_1 = 17 \) and \( T_\infty = 8 \) and thus \( T_1 / T_\infty = 2.125 \).
- What about \( T_1(\text{Fib}(n)) \) and \( T_\infty(\text{Fib}(n)) \)?
The Fibonacci example (2/2)

-we have $T_1(n) = T_1(n-1) + T_1(n-2) + \Theta(1)$. Let's solve it.
- One verify by induction that $T(n) \leq aF_n - b$ for $b > 0$ large enough to dominate $\Theta(1)$ and $a > 1$.
- We can then choose $a$ large enough to satisfy the initial condition, whatever that is.
- On the other hand we also have $F_n \leq T(n)$.
- Therefore $T_1(n) = \Theta(F_n) = \Theta(\psi^n)$ with $\psi = (1 + \sqrt{5})/2$.

We have $T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$.
- We easily check $T_\infty(n-1) \geq T_\infty(n-2)$.
- This implies $T_\infty(n) = T_\infty(n-1) + \Theta(1)$.
- Therefore $T_\infty(n) = \Theta(n)$.

Consequently the parallelism is $\Theta(\psi^n/n)$.
Series composition

- Work?
- Span?
Series composition

- **Work:** $T_1(A \cup B) = T_1(A) + T_1(B)$
- **Span:** $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$
Parallel composition

- Work?
- Span?
Parallel composition

- **Work:** $T_1(A \cup B) = T_1(A) + T_1(B)$
- **Span:** $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$
Some results in the fork-join parallelism model

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge sort</td>
<td>(\Theta(n \lg n))</td>
<td>(\Theta(lg^3n))</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>(\Theta(n^3))</td>
<td>(\Theta(lg n))</td>
</tr>
<tr>
<td>Strassen</td>
<td>(\Theta(n^{\lg 7}))</td>
<td>(\Theta(lg^2n))</td>
</tr>
<tr>
<td>LU–decomposition</td>
<td>(\Theta(n^3))</td>
<td>(\Theta(n \lg n))</td>
</tr>
<tr>
<td>Tableau construction</td>
<td>(\Theta(n^2))</td>
<td>(\Omega(n^{\lg 3}))</td>
</tr>
<tr>
<td>FFT</td>
<td>(\Theta(n \lg n))</td>
<td>(\Theta(lg^2n))</td>
</tr>
<tr>
<td>Breadth–first search</td>
<td>(\Theta(E))</td>
<td>(\Theta(d \lg V))</td>
</tr>
</tbody>
</table>

We shall prove those results in the next lectures.
Plan

Parallelism Complexity Measures

cilk_for Loops

Scheduling Theory and Implementation

Measuring Parallelism in Practice

Anticipating parallelization overheads

Announcements
For loop parallelism in Cilk++

The iterations of a `cilk_for` loop execute in parallel.

cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}

The iterations of a `cilk_for` loop execute in parallel.
Implementation of for loops in Cilk++

Up to details (next week!) the previous loop is compiled as follows, using a divide-and-conquer implementation:

```c
void recur(int lo, int hi) {
    if (hi > lo) { // coarsen
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid+1, hi);
        cilk_sync;
    } else
        for (int j=lo; j<hi+1; ++j) {
            double temp = A[hi][j];
            A[hi][j] = A[j][hi];
            A[j][hi] = temp;
        }
}
```
Analysis of parallel for loops

Here we do not assume that each strand runs in unit time.

- **Span of loop control**: $\Theta(\log(n))$
- **Max span of an iteration**: $\Theta(n)$
- **Span**: $\Theta(n)$
- **Work**: $\Theta(n^2)$
- **Parallelism**: $\Theta(n)$
Parallelizing the inner loop

This would yield the following code

```cilk
for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

- **Span of outer loop control**: $\Theta(\log(n))$
- **Max span of an inner loop control**: $\Theta(\log(n))$
- **Span of an iteration**: $\Theta(1)$
- **Span**: $\Theta(\log(n))$
- **Work**: $\Theta(n^2)$
- **Parallelism**: $\Theta(n^2 / \log(n))$

In practice, parallelizing the inner loop would increase the memory footprint (allocation of the temporaries) and increase parallelism overheads. So, this is not a good idea.
Plan

Parallelism Complexity Measures

cilk_for Loops

Scheduling Theory and Implementation

Measuring Parallelism in Practice

Anticipating parallelization overheads

Announcements
A scheduler’s job is to map a computation to particular processors. Such a mapping is called a schedule.

- If decisions are made at runtime, the scheduler is online, otherwise, it is offline

- Cilk++’s scheduler maps strands onto processors dynamically at runtime.
A strand is **ready** if all its predecessors have executed

A scheduler is **greedy** if it attempts to do as much work as possible at every step.
In any greedy schedule, there are two types of steps:

- **complete step**: There are at least \( p \) strands that are ready to run. The greedy scheduler selects any \( p \) of them and runs them.
- **incomplete step**: There are strictly less than \( p \) strands that are ready to run. The greedy scheduler runs them all.
For any greedy schedule, we have $T_p \leq T_1/p + T_\infty$

- #complete steps $\leq T_1/p$, by definition of $T_1$.
- #incomplete steps $\leq T_\infty$. Indeed, let $G'$ be the subgraph of $G$ that remains to be executed immediately prior to an incomplete step.
  
  (i) During this incomplete step, all strands that can be run are actually run.
  
  (ii) Hence removing this incomplete step from $G'$ reduces $T_\infty$ by one.
Corollary 1

*A greedy scheduler is always within a factor of 2 of optimal.*

From the work and span laws, we have:

\[
T_P \geq \max(\frac{T_1}{p}, T_\infty) \tag{1}
\]

In addition, we can trivially express:

\[
\frac{T_1}{p} \leq \max(\frac{T_1}{p}, T_\infty) \tag{2}
\]

\[
T_\infty \leq \max(\frac{T_1}{p}, T_\infty) \tag{3}
\]

From Graham - Brent Theorem, we deduce:

\[
T_P \leq \frac{T_1}{p} + T_\infty \tag{4}
\]

\[
\leq \max(\frac{T_1}{p}, T_\infty) + \max(\frac{T_1}{p}, T_\infty) \tag{5}
\]

\[
\leq 2 \max(\frac{T_1}{p}, T_\infty) \tag{6}
\]

which concludes the proof.
Corollary 2

The greedy scheduler achieves linear speedup whenever

\[ T_\infty = O(T_1/p). \]

From Graham - Brent Theorem, we deduce:

\[
T_p \leq T_1/p + T_\infty \leq T_1/p + O(T_1/p) = \Theta(T_1/p)
\]

The idea is to operate in the range where \( T_1/p \) dominates \( T_\infty \). As long as \( T_1/p \) dominates \( T_\infty \), all processors can be used efficiently. The quantity \( T_1/p T_\infty \) is called the parallel slackness.
The work-stealing scheduler (1/9)

- Cilk/Cilk++ randomized work-stealing scheduler load-balances the computation at run-time. Each processor maintains a ready deque:
  - A ready deque is a double ended queue, where each entry is a procedure instance that is ready to execute.
  - Adding a procedure instance to the bottom of the deque represents a procedure call being spawned.
  - A procedure instance being deleted from the bottom of the deque represents the processor beginning/resuming execution on that procedure.
  - Deletion from the top of the deque corresponds to that procedure instance being stolen.

- A mathematical proof guarantees near-perfect linear speed-up on applications with sufficient parallelism, as long as the architecture has sufficient memory bandwidth.

- A spawn/return in Cilk is over 100 times faster than a Pthread create/exit and less than 3 times slower than an ordinary C function call on a modern Intel processor.
Each processor possesses a deque
The work-stealing scheduler (3/9)
The work-stealing scheduler (4/9)
The work-stealing scheduler (5/9)
The work-stealing scheduler (6/9)
The work-stealing scheduler (7/9)
The work-stealing scheduler (8/9)
Performances of the work-stealing scheduler

Assume that

- each strand executes in unit time,
- for almost all “parallel steps” there are at least \( p \) strands to run,
- each processor is either working or stealing.

Then, the randomized work-stealing scheduler is expected to run in

\[
T_P = \frac{T_1}{p} + O(T_\infty)
\]

- A processor is either working or stealing.
- The total time all processors spend working is \( T_1 \), by definition of \( T_1 \).
- Each stealing processor has a probability of \( 1/P \) to reduce the span by 1.
- Thus, the \textbf{expected} number of steals is \( O(P T_\infty) \).
- Since \( P \) processors are working/stealing together, the expected running time

\[
T_P = \# \text{steps without steal} + \# \text{steps with steal} = \frac{T_1}{p} + O(p T_\infty)/P. \tag{10}
\]
Overheads and burden

- Obviously $T_1/p + T_\infty$ will under-estimate $T_p$ in practice.

- Many factors (simplification assumptions of the fork-join parallelism model, architecture limitation, costs of executing the parallel constructs, overheads of scheduling) will make $T_p$ larger in practice.

- One may want to estimate the impact of those factors:
  1. by improving the estimate of the randomized work-stealing complexity result
  2. by comparing a Cilk++ program with its C++ elision
  3. by estimating the costs of spawning and synchronizing

- Cilk++ estimates $T_p$ as $T_p = T_1/p + 1.7 \text{ burden.span}$, where burden.span is 15000 instructions times the number of continuation edges along the critical path.
Span overhead

- Let $T_1, T_\infty, T_p$ be given. We want to refine the randomized work-stealing complexity result.

- The **span overhead** is the smallest constant $c_\infty$ such that

  \[ T_p \leq T_1/p + c_\infty T_\infty. \]

- Recall that $T_1/T_\infty$ is the maximum possible speed-up that the application can obtain.

- We call **parallel slackness assumption** the following property

  \[ T_1/T_\infty \gg c_\infty p \]  \hspace{1cm} (11)

  that is, $c_\infty p$ is much smaller than the average parallelism.

- Under this assumption it follows that $T_1/p \gg c_\infty T_\infty$ holds, thus $c_\infty$ has little effect on performance when sufficiently slackness exists.
Work overhead

- Let $T_s$ be the running time of the C++ elision of a Cilk++ program.

- We denote by $c_1$ the work overhead
  
  \[ c_1 = \frac{T_1}{T_s} \]

- Recall the expected running time: $T_P \leq \frac{T_1}{P} + c_\infty T_\infty$. Thus with the parallel slackness assumption we get
  
  \[ T_P \leq c_1 \frac{T_s}{p} + c_\infty T_\infty \approx c_1 \frac{T_s}{p}. \quad (12) \]

- We can now state the work first principle precisely
  
  Minimize $c_1$, even at the expense of a larger $c_\infty$. This is a key feature since it is conceptually easier to minimize $c_1$ rather than minimizing $c_\infty$.

- Cilk++ estimates $T_p$ as $T_p = \frac{T_1}{p} + 1.7 \text{ burden}_\text{span}$, where burden_span is 15000 instructions times the number of continuation edges along the critical path.
Plan

Parallelism Complexity Measures

cilk_for Loops

Scheduling Theory and Implementation

Measuring Parallelism in Practice

Anticipating parallelization overheads

Announcements
Cilkview computes work and span to derive upper bounds on parallel performance. Cilkview also estimates scheduling overhead to compute a burdened span for lower bounds.
The Fibonacci Cilk++ example

Code fragment

long fib(int n)
{
    if (n < 2) return n;
    long x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
Fibonacci program timing

The environment for benchmarking:

- model name: Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz
- L2 cache size: 4096 KB
- memory size: 3 GB

<table>
<thead>
<tr>
<th>n</th>
<th>#cores = 1 timing(s)</th>
<th>#cores = 2 timing(s)</th>
<th>speedup</th>
<th>#cores = 4 timing(s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0.086</td>
<td>0.046</td>
<td>1.870</td>
<td>0.025</td>
<td>3.440</td>
</tr>
<tr>
<td>35</td>
<td>0.776</td>
<td>0.436</td>
<td>1.780</td>
<td>0.206</td>
<td>3.767</td>
</tr>
<tr>
<td>40</td>
<td>8.931</td>
<td>4.842</td>
<td>1.844</td>
<td>2.399</td>
<td>3.723</td>
</tr>
<tr>
<td>45</td>
<td>105.263</td>
<td>54.017</td>
<td>1.949</td>
<td>27.200</td>
<td>3.870</td>
</tr>
<tr>
<td>50</td>
<td>1165.000</td>
<td>665.115</td>
<td>1.752</td>
<td>340.638</td>
<td>3.420</td>
</tr>
</tbody>
</table>
Quicksort

code in cilk/examples/qsort

```c
void sample_qsort(int * begin, int * end)
{
    if (begin != end) {
        --end;
        int * middle = std::partition(begin, end,
            std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);
        cilk_spawn sample_qsort(begin, middle);
        sample_qsort(++middle, ++end);
        cilk_sync;
    }
}
```
### Quicksort timing

Timing for sorting an array of integers:

<table>
<thead>
<tr>
<th># of int</th>
<th>#cores = 1</th>
<th>#cores = 2</th>
<th>speedup</th>
<th>#cores = 4</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 \times 10^6$</td>
<td>1.958</td>
<td>1.016</td>
<td>1.927</td>
<td>0.541</td>
<td>3.619</td>
</tr>
<tr>
<td>$50 \times 10^6$</td>
<td>10.518</td>
<td>5.469</td>
<td>1.923</td>
<td>2.847</td>
<td>3.694</td>
</tr>
<tr>
<td>$100 \times 10^6$</td>
<td>21.481</td>
<td>11.096</td>
<td>1.936</td>
<td>5.954</td>
<td>3.608</td>
</tr>
<tr>
<td>$500 \times 10^6$</td>
<td>114.300</td>
<td>57.996</td>
<td>1.971</td>
<td>31.086</td>
<td>3.677</td>
</tr>
</tbody>
</table>
Matrix multiplication

Code in cilk/examples/matrix

Timing of multiplying a $687 \times 837$ matrix by a $837 \times 1107$ matrix

<table>
<thead>
<tr>
<th>threshold</th>
<th>iterative</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>st(s)</td>
<td>pt(s)</td>
<td>su</td>
<td>st(s)</td>
<td>pt (s)</td>
<td>su</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.273</td>
<td>1.165</td>
<td>0.721</td>
<td>1.674</td>
<td>0.399</td>
<td>4.195</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.270</td>
<td>1.787</td>
<td>0.711</td>
<td>1.408</td>
<td>0.349</td>
<td>4.034</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.280</td>
<td>1.757</td>
<td>0.729</td>
<td>1.223</td>
<td>0.308</td>
<td>3.971</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>1.258</td>
<td>1.760</td>
<td>0.715</td>
<td>1.164</td>
<td>0.293</td>
<td>3.973</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.258</td>
<td>1.798</td>
<td>0.700</td>
<td>1.159</td>
<td>0.291</td>
<td>3.983</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>1.252</td>
<td>1.773</td>
<td>0.706</td>
<td>1.267</td>
<td>0.320</td>
<td>3.959</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

st = sequential time; pt = parallel time with 4 cores; su = speedup
The cilkview example from the documentation

Using cilk_for to perform operations over an array in parallel:

```c
static const int COUNT = 4;
static const int ITERATION = 1000000;
long arr[COUNT];
long do_work(long k){
  long x = 15;
  static const int nn = 87;
  for (long i = 1; i < nn; ++i)
    x = x / i + k % i;
  return x;
}
int cilk_main(){
  for (int j = 0; j < ITERATION; j++)
    cilk_for (int i = 0; i < COUNT; i++)
      arr[i] += do_work( j * i + i + j);
}
```
1) Parallelism Profile

Work : 6,480,801,250 ins
Span : 2,116,801,250 ins
Burdened span : 31,920,801,250 ins
Parallelism : 3.06
Burdened parallelism : 0.20
Number of spawns/syncs: 3,000,000
Average instructions / strand : 720
Strands along span : 4,000,001
Average instructions / strand on span : 529

2) Speedup Estimate

2 processors: 0.21 - 2.00
4 processors: 0.15 - 3.06
8 processors: 0.13 - 3.06
16 processors: 0.13 - 3.06
32 processors: 0.12 - 3.06
Inverting the two for loops

```c
int cilk_main()
{
    cilk_for (int i = 0; i < COUNT; i++)
        for (int j = 0; j < ITERATION; j++)
            arr[i] += do_work( j * i + i + j);
}
```
1) Parallelism Profile

Work : 5,295,801,529 ins
Span : 1,326,801,107 ins
Burdened span : 1,326,830,911 ins
Parallelism : 3.99
Burdened parallelism : 3.99
Number of spawns/syncs: 3
Average instructions / strand : 529,580,152
Strands along span : 5
Average instructions / strand on span: 265,360,221

2) Speedup Estimate

2 processors: 1.40 - 2.00
4 processors: 1.76 - 3.99
8 processors: 2.01 - 3.99
16 processors: 2.17 - 3.99
32 processors: 2.25 - 3.99
## Timing

<table>
<thead>
<tr>
<th>version</th>
<th>#cores = 1</th>
<th>#cores = 2</th>
<th>#cores = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>timing(s)</td>
<td>timing(s)</td>
<td>timing(s)</td>
</tr>
<tr>
<td>original</td>
<td>7.719</td>
<td>9.611</td>
<td>0.803</td>
</tr>
<tr>
<td>improved</td>
<td>7.471</td>
<td>3.724</td>
<td>2.006</td>
</tr>
</tbody>
</table>
Plan

Parallelism Complexity Measures

cilk_for Loops

Scheduling Theory and Implementation

Measuring Parallelism in Practice

Anticipating parallelization overheads

Announcements
Construction of the Pascal Triangle: nearly the simplest stencil computation!
The parallelism is $\Theta(n^{2-\log_2 3})$, so roughly $\Theta(n^{0.45})$ which can be regarded as low parallelism.
Let \( B \) be the order of a block and \( n \) be the number of elements.

The parallelism of \( \Theta(n/B) \) can still be regarded as low parallelism, but better than with the divide and conquer scheme.
Estimating parallelization overheads

The instruction stream DAG of the blocking strategy consists of \( n/B \) binary trees \( T_0, T_1, \ldots, T_{n/B-1} \) such that

- \( T_i \) is the instruction stream DAG of the \texttt{cilk}\_for loop executing the \( i \)-th band
- each leaf of \( T_i \) is connected by an edge to the root of \( T_{i+1} \).

Consequently, the burdened span is

\[
S_b(n) = \sum_{i=1}^{n/B} \log(i) = \log\left( \prod_{i=1}^{n/B} i \right) = \log\left( \Gamma\left( \frac{n}{B} + 1 \right) \right).
\]

Using Stirling’s Formula, we deduce

\[
S_b(n) \in \Theta\left( \frac{n}{B} \log\left( \frac{n}{B} \right) \right). \tag{13}
\]

Thus the burdened parallelism (that is, the ratio work to burdened span) is \( \Theta(Bn/\log(\frac{n}{B})) \), that is sub-linear in \( n \), while the non-burdened parallelism is \( \Theta(n/B) \).
Construction of the Pascal Triangle: experimental results

Worker vs Speedup and Parallelism

- speedup dynamic block
- speedup static block
- parallelism dynamic block
- parallelism static block

Core/Workers vs Speedup and Parallelism

Core/Workers

- 2
- 4
- 6
- 8
- 10
- 12

Speedup and Parallelism

- Speedup dynamic block
- Speedup static block
- Parallelism dynamic block
- Parallelism static block
Burdened parallelism

- Parallelism after accounting for parallelization overheads (thread management, costs of scheduling, etc.) The **burdened parallelism** is estimated as the ratio work to burdened span.
- The **burdened span** is defined as the maximum number of spawns/syncs on a critical path times the cost for a `cilk_spawn` (`cilk_sync`) taken as 15,000 cycles.

Impact in practice: example for the Pascal Triangle

- Consider executing one band after another, where for each band all $B \times B$ blocks are executed concurrently.
- The **non-burdened span** is in $\Theta(B^2 n/B) = \Theta(n/B)$.
- While the **burdened span** is

$$S_b(n) = \sum_{i=1}^{n/B} \log(i) = \log(\prod_{i=1}^{n/B} i) = \log(\Gamma(\frac{n}{B} + 1)) \in \Theta\left(\frac{n}{B} \log\left(\frac{n}{B}\right)\right).$$
Plan

Parallelism Complexity Measures

cilk_for Loops

Scheduling Theory and Implementation

Measuring Parallelism in Practice

Anticipating parallelization overheads

Announcements
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++.

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

▶ Liyun Li (UWO) for generating the experimental data.
References

