The Fork-Join Model

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

CS4402 - CS9635, February 9, 2024

The Fork-Join Model

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

CS4402 - CS9635, February 9, 2024

Plan

Outline

Outline

From Cilk to Cilk++, CilkPlus and OpenCilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao Benjamin Schardl.

From Cilk to Cilk++, CilkPlus and OpenCilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao Benjamin Schardl.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.

From Cilk to Cilk++, CilkPlus and OpenCilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao Benjamin Schardl.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of `Cilk` have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.

From Cilk to Cilk++, CilkPlus and OpenCilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao Benjamin Schardl.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of `Cilk` have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to `Cilk++`, developed by Cilk Arts, an MIT spin-off, acquired by Intel in July 2009 and became CilkPlus.

From Cilk to Cilk++, CilkPlus and OpenCilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao Benjamin Schardl.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of `Cilk` have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to `Cilk++`, developed by Cilk Arts, an MIT spin-off, acquired by Intel in July 2009 and became CilkPlus.
- I recommend the following `CilkPlus` documentation

  `https://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf`

From Cilk to Cilk++, CilkPlus and OpenCilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao Benjamin Schardl.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of `Cilk` have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to `Cilk++`, developed by Cilk Arts, an MIT spin-off, acquired by Intel in July 2009 and became CilkPlus.
- I recommend the following `CilkPlus` documentation
  `https://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf`
- `Cilk` is still now developed at MIT with NSF support
  `https://cilk.mit.edu`

From Cilk to Cilk++, CilkPlus and OpenCilk

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo and Tao Benjamin Schardl.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of `Cilk` have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to `Cilk++`, developed by Cilk Arts, an MIT spin-off, acquired by Intel in July 2009 and became CilkPlus.
- I recommend the following `CilkPlus` documentation
  `https://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf`
- `Cilk` is still now developed at MIT with NSF support
  `https://cilk.mit.edu`
- In this course, we will be using OpenCilk which is freely available in source from the above URL.

Cilk

- `Cilk` is a small set of linguistic extensions to `C++` (resp. `C`) supporting fork-join parallelism

Cilk

- `Cilk` is a small set of linguistic extensions to `C++` (resp. `C`) supporting fork-join parallelism
- `Cilk`'s runtime features a provably efficient work-stealing scheduler.

Cilk

- `Cilk` is a small set of linguistic extensions to `C++` (resp. `C`) supporting fork-join parallelism
- `Cilk`'s runtime features a provably efficient work-stealing scheduler.
- A number third-party libraries are known to work with `OpenCilk` out of the box for parallel execution, see OpenCilk-powered libraries.

Cilk

- Cilk is a small set of linguistic extensions to C++ (resp. C) supporting fork-join parallelism
- Cilk's runtime features a provably efficient work-stealing scheduler.
- A number third-party libraries are known to work with OpenCilk out of the box for parallel execution, see OpenCilk-powered libraries.
- OpenCilk includes the Cilkscale performance analyzer.

Nested Parallelism in Cilk

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

Nested Parallelism in Cilk

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

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent executes `fib(n-2)`.

Nested Parallelism in Cilk

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

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent executes `fib(n-2)`.
- `Cilk++` keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.

Loop Parallelism in Cilk

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}$$

```
// indices run from 0, not 1
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 may execute in parallel.

Serial Semantics (1/2)

- `Cilk` is a multithreaded language for parallel programming that generalizes the semantics of `C` by introducing linguistic constructs for parallel control.

Serial Semantics (1/2)

- `Cilk` is a multithreaded language for parallel programming that generalizes the semantics of `C` by introducing linguistic constructs for parallel control.
- `Cilk` is a faithful extension of `C` (resp. `C++`):

Serial Semantics (1/2)

- `Cilk` is a multithreaded language for parallel programming that generalizes the semantics of `C` by introducing linguistic constructs for parallel control.
- `Cilk` is a faithful extension of `C` (resp. `C++`):
  - ↳ The `C` elision of a `Cilk` (resp. `Cilk++`) is a correct implementation of the semantics of the program.

Serial Semantics (1/2)

- `Cilk` is a multithreaded language for parallel programming that generalizes the semantics of `C` by introducing linguistic constructs for parallel control.
- `Cilk` is a faithful extension of `C` (resp. `C++`):
  - ↳ The `C` elision of a `Cilk` (resp. `Cilk++`) is a correct implementation of the semantics of the program.
  - ↳ Moreover, on one processor, a parallel `Cilk` program scales down to run nearly as fast as its `C` elision.

Serial Semantics (1/2)

- Cilk is a multithreaded language for parallel programming that generalizes the semantics of C by introducing linguistic constructs for parallel control.
- Cilk is a faithful extension of C (resp. C++):
  - ↳ The C elision of a Cilk (resp. Cilk++) is a correct implementation of the semantics of the program.
  - ↳ Moreover, on one processor, a parallel Cilk program scales down to run nearly as fast as its C elision.
- To obtain the serialization of a Cilk program

```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```

Serial Semantics (2/2)

```
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);
  }
}
```

Cilk++ source

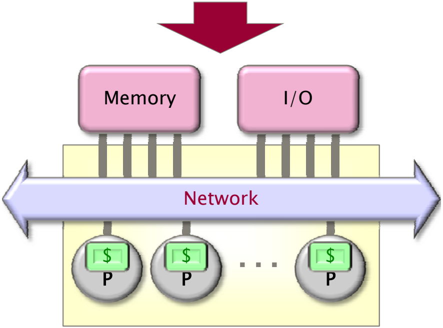
```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

Serialization

Outline

Scheduling (1/2)

```
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);
  }
}
```



Scheduling (2/2)

- `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.

Scheduling (2/2)

- `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.

Scheduling (2/2)

- `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.

Scheduling (2/2)

- `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.

Scheduling (2/2)

- `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.

Scheduling (2/2)

- `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.

Outline

```
template<typename T> void multiply_iter_par(int ii, int jj, int kk,
    T* C)
{
    cilk_for(int i = 0; i < ii; ++i)
        cilk_for(int j = 0; j < jj; ++j)
            for (int k = 0; k < kk; ++k)
               C[i * jj + j] += A[i * kk + k] + B[k * jj + j];
}
```

Does not scale up well due to a poor locality and uncontrolled granularity.

```
template<typename T> void multiply_rec_seq_helper(int i0, int i1, int j0,
    int j1, int k0, int k1, T* A, ptrdiff_t lda, T* B, ptrdiff_t ldb, T* C,
    ptrdiff_t ldc)
{
    int di = i1 - i0;
    int dj = j1 - j0;
    int dk = k1 - k0;
    if (di >= dj && di >= dk && di >= RECURSION_THRESHOLD) {
        int mi = i0 + di / 2;
        multiply_rec_seq_helper(i0, mi, j0, j1, k0, k1, A, lda, B, ldb, C, ldc);
        multiply_rec_seq_helper(mi, i1, j0, j1, k0, k1, A, lda, B, ldb, C, ldc);
    } else if (dj >= dk && dj >= RECURSION_THRESHOLD) {
        int mj = j0 + dj / 2;
        multiply_rec_seq_helper(i0, i1, j0, mj, k0, k1, A, lda, B, ldb, C, ldc);
        multiply_rec_seq_helper(i0, i1, mj, j1, k0, k1, A, lda, B, ldb, C, ldc);
    } else if (dk >= RECURSION_THRESHOLD) {
        int mk = k0 + dk / 2;
        multiply_rec_seq_helper(i0, i1, j0, j1, k0, mk, A, lda, B, ldb, C, ldc);
        multiply_rec_seq_helper(i0, i1, j0, j1, mk, k1, A, lda, B, ldb, C, ldc);
    } else {
        for (int i = i0; i < i1; ++i)
            for (int k = k0; k < k1; ++k)
                for (int j = j0; j < j1; ++j)
                    C[i * ldc + j] += A[i * lda + k] * B[k * ldb + j];
    }
}
```

```
template<typename T> inline void multiply_rec_seq(int ii, int jj, in
    T* B, T* C)
{
    multiply_rec_seq_helper(0, ii, 0, jj, 0, kk, A, kk, B, jj, C, jj
}
```

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

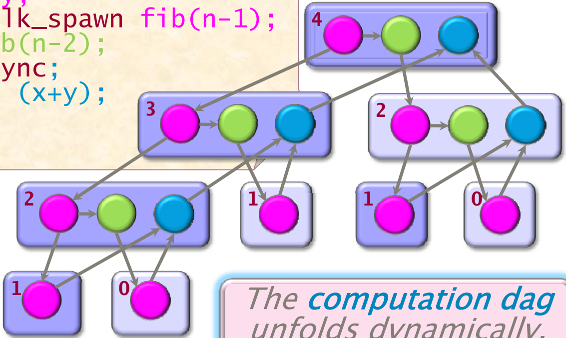
#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83

Outline

The fork-join parallelism model

```
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)`

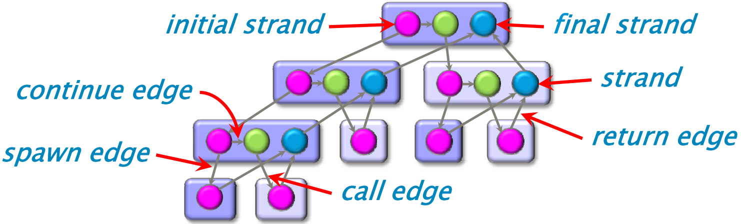


"Processor oblivious"

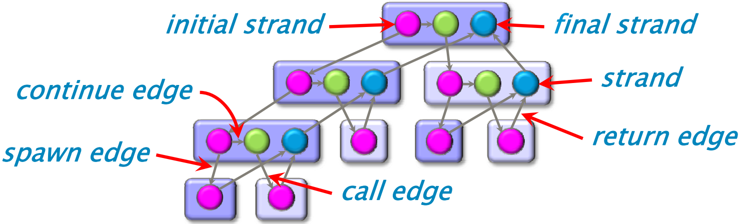
*The **computation dag** unfolds dynamically.*

We shall also call this model **multithreaded parallelism**.

Terminology

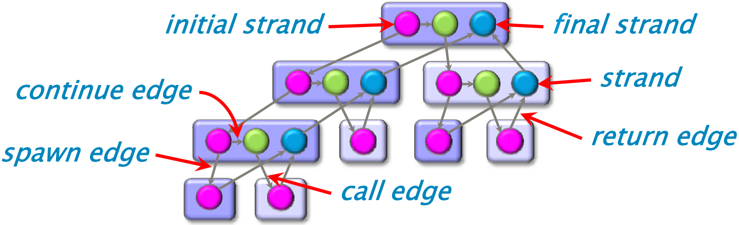


Terminology



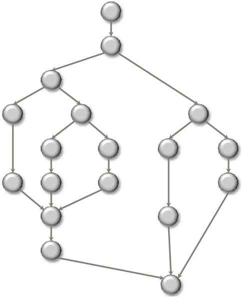
- a **strand** is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` (either explicit or implicit) statement.

Terminology



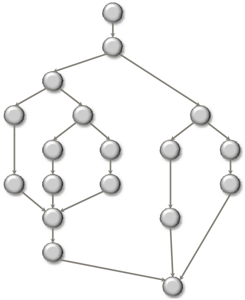
- a **strand** is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` (either explicit or implicit) statement.
- At runtime, the *spawn* relation causes procedure instances to be structured as a rooted tree, called **spawn tree** or **parallel instruction stream**, where dependencies among strands form a dag.

Work and span



We define several performance measures. We assume an ideal situation: no cache issues, no interprocessor costs:

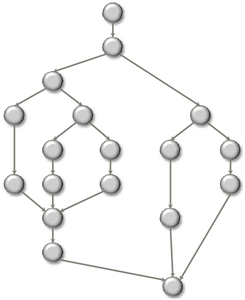
Work and span



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

Work and span

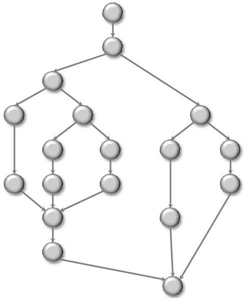


We define several performance measures. We assume an ideal situation: no cache issues, no interprocessor costs:

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.

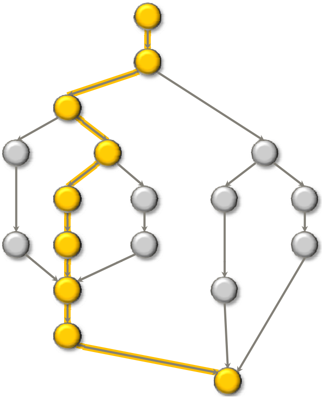
Work and span



We define several performance measures. We assume an ideal situation: no cache issues, no interprocessor costs:

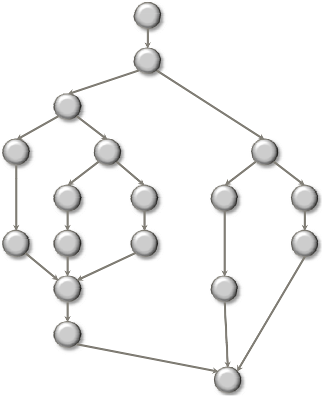
- T_p is the minimum running time on p processors
- T_1 is called the **work**, that is, the sum of the number of instructions at each node.
- T_∞ is the minimum running time with infinitely many processors, called the **span**

The critical path length

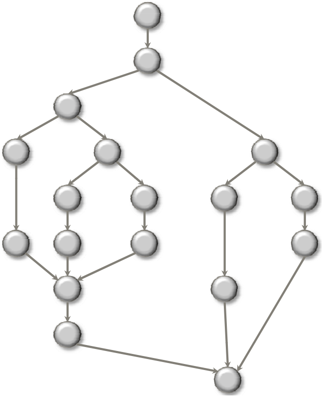


Assuming all strands run in unit time, the longest path in the DAG is equal to T_∞. For this reason, T_∞ is also referred to as the **critical path length**.

Work law

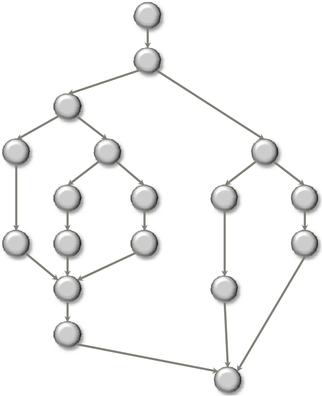


Work law



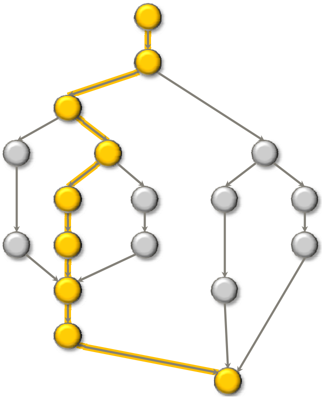
- We have: $T_p \geq T_1/p$.

Work law

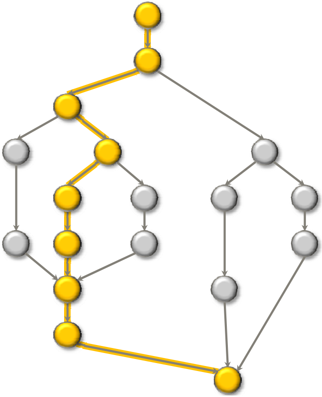


- We have: $T_p \geq T_1/p$.
- Indeed, in the best case, p processors can do p works per unit of time.

Span law

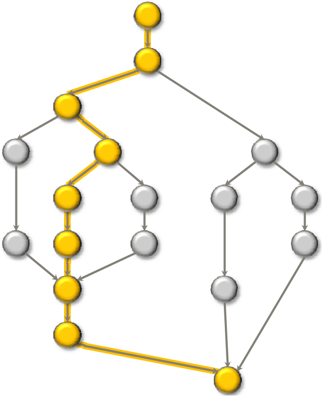


Span law



- We have: $T_p \geq T_\infty$.

Span law



- We have: $T_p \geq T_\infty$.
- Indeed, $T_p < T_\infty$ contradicts the definitions of T_p and T_∞.

Speedup on p processors

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

Speedup on p processors

- T_1/T_p is called the **speedup on p processors**
- A parallel program execution can have:

Speedup on p processors

- T_1/T_p is called the **speedup on p processors**
- A parallel program execution can have:
  ↳ **linear speedup**: $T_1/T_P = \Theta(p)$

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)

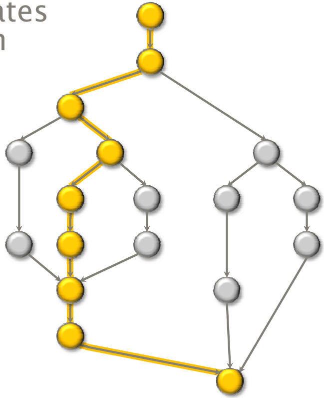
Speedup on p processors

- 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**: $T_1/T_P = o(p)$

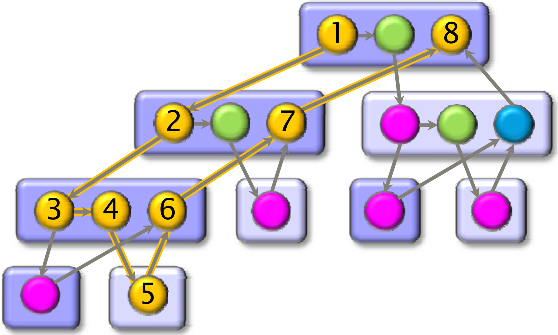
Parallelism

Because the Span Law dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

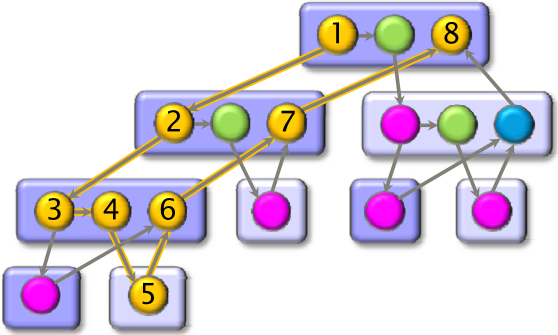
$$\begin{aligned} T_1/T_\infty &= \textit{parallelism} \\ &= \text{the average amount of work per step along the span.} \end{aligned}$$



The Fibonacci example (1/2)

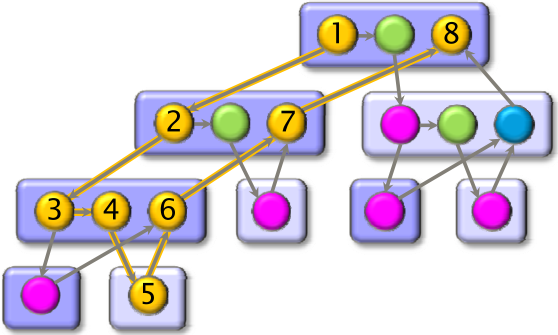


The Fibonacci example (1/2)



- For $\mathrm{Fib}(4)$, we have $T_1 = 17$ and $T_\infty = 8$ and thus $T_1/T_\infty = 2.125$.

The Fibonacci example (1/2)



- For $\mathrm{Fib}(4)$, we have $T_1 = 17$ and $T_\infty = 8$ and thus $T_1/T_\infty = 2.125$.
- What about $T_1(\mathrm{Fib}(n))$ and $T_\infty(\mathrm{Fib}(n))$?

The Fibonacci example (2/2)

- We have $T_1(n) = T_1(n-1) + T_1(n-2) + \Theta(1)$. Let's solve it.

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 can verify by induction that $T(n) \leq aF_n - b$ for $b > 0$ large enough to dominate $\Theta(1)$ and $a > 1$.

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 can verify by induction that $T(n) \le aF_n - b$ 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.

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 can 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 $F_n \leq T(n)$.

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 can 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 $\psi = (1+\sqrt{5})/2$.

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 can 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 $T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$.

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 can 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 $T_\infty(n-1) \geq T_\infty(n-2)$.

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 can 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 $T_\infty(n) = T_\infty(n-1) + \Theta(1)$.

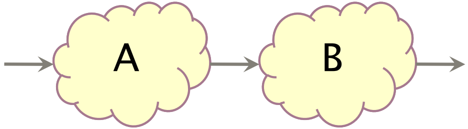
The Fibonacci example (2/2)

- We have $T_1(n) = T_1(n-1) + T_1(n-2) + \Theta(1)$. Let's solve it.
  - ↳ One can 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 $T_\infty(n) = \Theta(n)$.

The Fibonacci example (2/2)

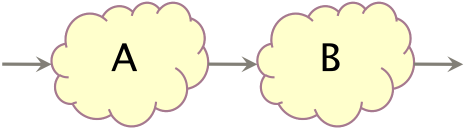
- We have $T_1(n) = T_1(n-1) + T_1(n-2) + \Theta(1)$. Let's solve it.
  - ↳ One can verify by induction that $T(n) \leq aF_n - b$ for $b > 0$ large enough to dominate $\Theta(1)$ and $a > 1$.
  - ↳ We can then choose a large enough to satisfy the initial condition, whatever that is.
  - ↳ On the other 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



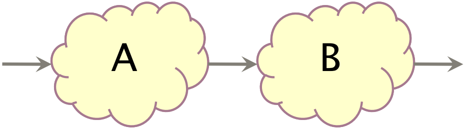
- Work?

Series composition



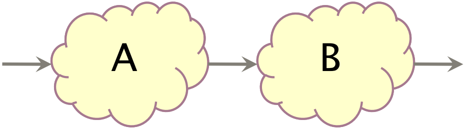
- Work?
- Span?

Series composition



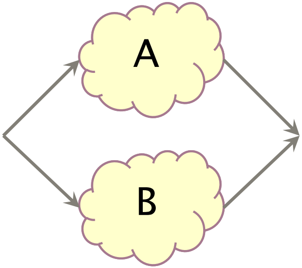
- Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Series composition



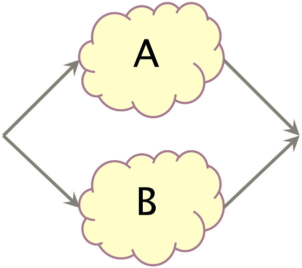
- Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
- Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

Parallel composition



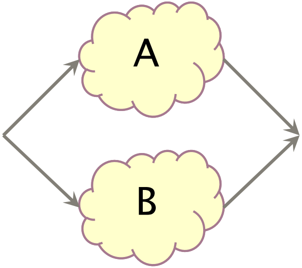
- Work?

Parallel composition



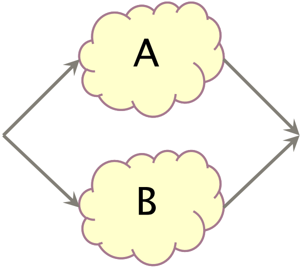
- Work?
- Span?

Parallel composition



- Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

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

Algorithm	Work	Span
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\lg n)$
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$
Tableau construction	$\Theta(n^2)$	$\Omega(n^{\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$
Breadth-first search	$\Theta(E)$	$\Theta(d \lg V)$

We shall prove most of these results in the next sections.

For loop parallelism in Cilk++

$$\underset{A}{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}} \Rightarrow \underset{A^T}{\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}}$$

```
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;
    }
}
```

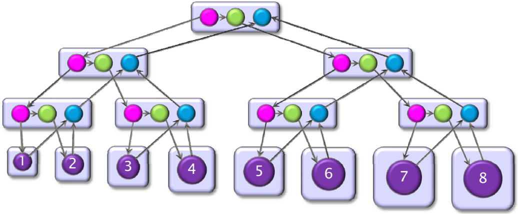
The iterations of a `cilk_for` loop execute in parallel.

Implementation of `for` loops in `Cilk++`

Up to details the previous loop is compiled as follows, using a **divide-and-conquer implementation**:

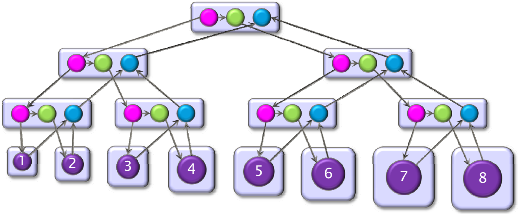
```
void recur(int lo, int hi) {
    if (hi > lo) { // coarsen
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid+1, hi);
        cilk_sync;
    } else
        for (int j=lo; j<hi+1; ++j) {
            double temp = A[hi][j];
            A[hi][j] = A[j][hi];
   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

Analysis of parallel `for` loops



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

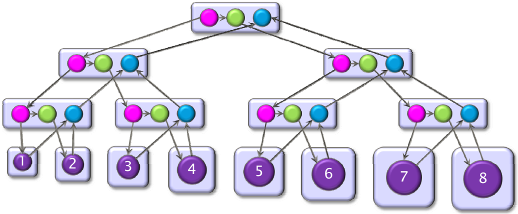
Analysis of parallel `for` loops



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

- **Span of loop control**: $\Theta(\log(n))$

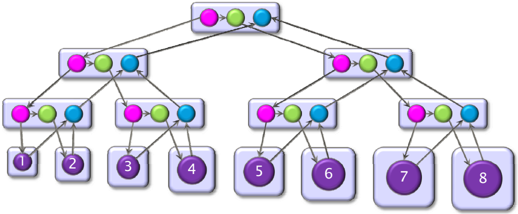
Analysis of parallel `for` loops



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

- **Span of loop control**: $\Theta(\log(n))$
- **Max span of an iteration**: $\Theta(n)$

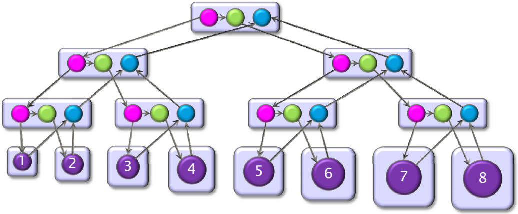
Analysis of parallel `for` loops



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**: $\Theta(n^2)$

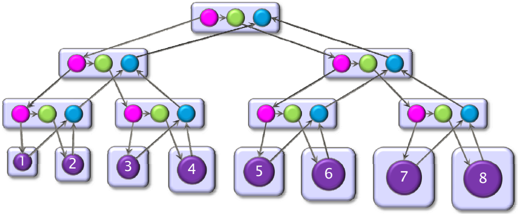
Analysis of parallel `for` loops



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**: $\Theta(n^2)$

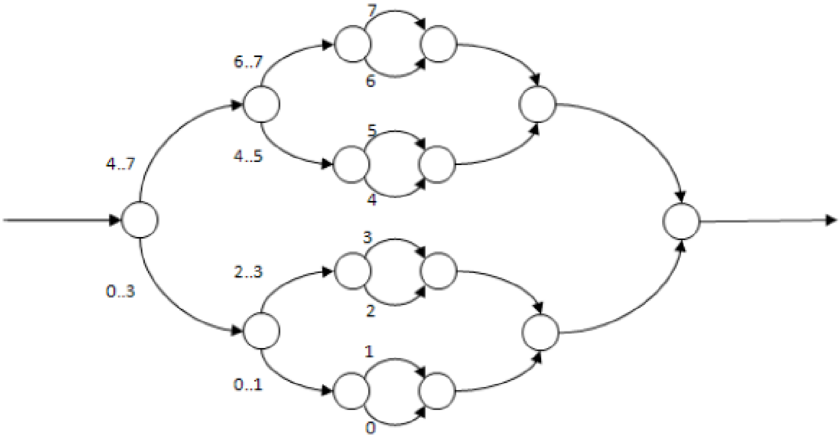
Analysis of parallel `for` loops



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**: $\Theta(n)$

Analysis of parallel `for` loops



Parallelizing the inner loop

This would yield the following code

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

Parallelizing the inner loop

This would yield the following code

```
cilk_for (int i=1; i<n; ++i) {
    cilk_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**: $\Theta(\log(n))$

Parallelizing the inner loop

This would yield the following code

```
cilk_for (int i=1; i<n; ++i) {
    cilk_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**: $\Theta(\log(n))$

Parallelizing the inner loop

This would yield the following code

```
cilk_for (int i=1; i<n; ++i) {
    cilk_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**: $\Theta(1)$

Parallelizing the inner loop

This would yield the following code

```
cilk_for (int i=1; i<n; ++i) {
    cilk_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**: $\Theta(\log(n))$

Parallelizing the inner loop

This would yield the following code

```
cilk_for (int i=1; i<n; ++i) {
    cilk_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**: $\Theta(n^2)$

Parallelizing the inner loop

This would yield the following code

```
cilk_for (int i=1; i<n; ++i) {
    cilk_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**: $\Theta(n^2/\log(n))$

Parallelizing the inner loop

This would yield the following code

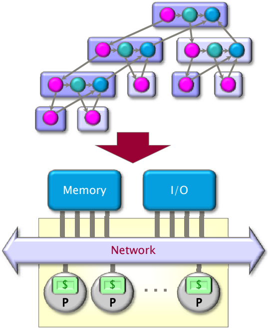
```
cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

- **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.

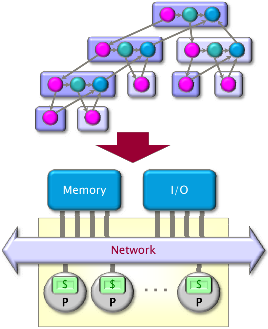
Outline

Scheduling



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

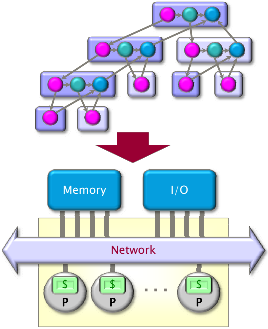
Scheduling



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*

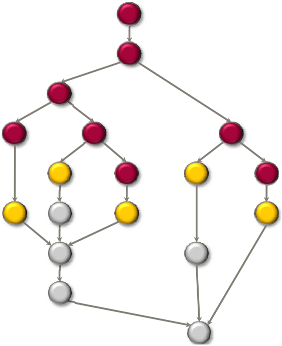
Scheduling



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

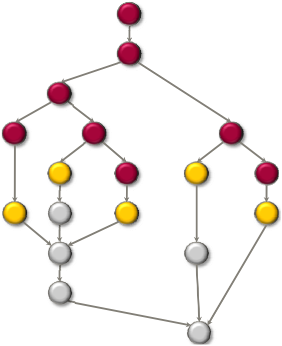
- If decisions are made at runtime, the scheduler is *online*, otherwise, it is *offline*
- `Cilk++`'s scheduler maps strands onto processors dynamically at runtime.

Greedy scheduling (1/2)



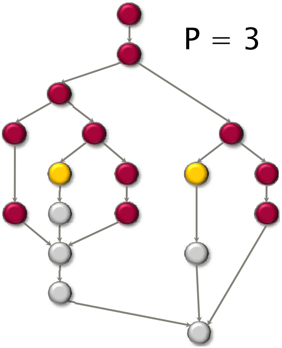
- A strand is **ready** if all its predecessors have executed

Greedy scheduling (1/2)



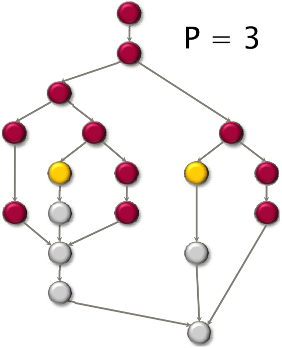
- A strand is **ready** if all its predecessors have executed
- A scheduler is **greedy** if it attempts to do as much work as possible at every step.

Greedy scheduling (2/2)



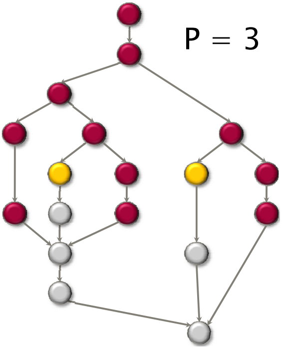
- In any *greedy schedule*, there are two types of steps:

Greedy scheduling (2/2)



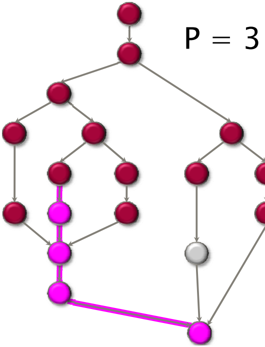
- In any *greedy schedule*, there are two types of steps:
  - ↳ **complete step**: There are at least p strands that are ready to run. The greedy scheduler selects any p of them and runs them.

Greedy scheduling (2/2)



- 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.

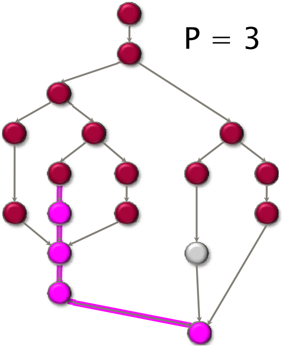
Theorem of Graham and Brent



For any greedy schedule, we have $T_p \;\leq\; T_1/p \;+\; T_\infty$

- $\#\text{complete steps} \leq T_1/p$, by definition of T_1.

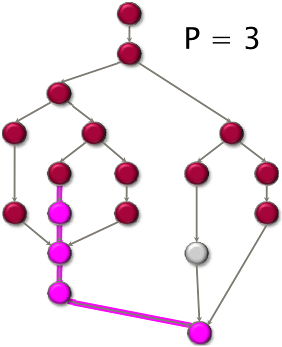
Theorem of Graham and Brent



For any greedy schedule, we have $T_p \leq T_1/p + T_\infty$

- $\#\text{complete steps} \leq T_1/p$, by definition of T_1.
- $\#\text{incomplete steps} \leq T_\infty$. Indeed, let G' be the subgraph of G that remains to be executed immediately prior to an incomplete step.

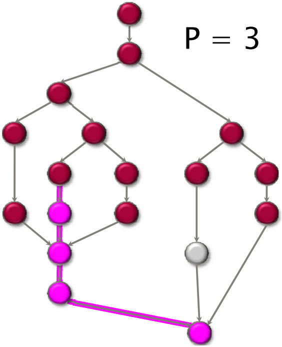
Theorem of Graham and Brent



For any greedy schedule, we have $T_p \leq T_1/p + T_\infty$

- $\#\text{complete steps} \leq T_1/p$, by definition of T_1.
- $\#\text{incomplete steps} \leq T_\infty$. 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

Theorem of Graham and Brent



For any greedy schedule, we have $T_p \leq T_1/p + T_\infty$

- $\#\text{complete steps} \leq T_1/p$, by definition of T_1.
- $\#\text{incomplete steps} \leq T_\infty$. 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_∞ by one.

Corollary 1

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

Corollary 1

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

From the work and span laws, we have:

$$T_P \geq \max(T_1/p, T_\infty) \tag{1}$$

Corollary 1

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

From the work and span laws, we have:

$$T_P \geq \max(T_1/p, T_\infty) \tag{1}$$

In addition, we can trivially express:

$$T_1/p \leq \max(T_1/p, T_\infty) \tag{2}$$

$$T_\infty \leq \max(T_1/p, T_\infty) \tag{3}$$

Corollary 1

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

From the work and span laws, we have:

$$T_P \geq \max(T_1/p, T_\infty) \tag{1}$$

In addition, we can trivially express:

$$T_1/p \leq \max(T_1/p, T_\infty) \tag{2}$$

$$T_\infty \leq \max(T_1/p, T_\infty) \tag{3}$$

From Graham - Brent Theorem, we deduce:

$$\begin{aligned} T_P &\leq T_1/p + T_\infty && (4)\\ &\leq \max(T_1/p, T_\infty) + \max(T_1/p, T_\infty) && (5)\\ &\leq 2\max(T_1/p, T_\infty) && (6) \end{aligned}$$

which concludes the proof.

Corollary 2

The greedy scheduler achieves linear speedup whenever $T_\infty = O(T_1/p)$.

Corollary 2

The greedy scheduler achieves linear speedup whenever $T_\infty = O(T_1/p)$.

From Graham - Brent Theorem, we deduce:

$$\begin{aligned} T_p &\leq T_1/p + T_\infty && (7) \\ &= T_1/p + O(T_1/p) && (8) \\ &= \Theta(T_1/p) && (9) \end{aligned}$$

Corollary 2

The greedy scheduler achieves linear speedup whenever $T_\infty = O(T_1/p)$.

From Graham - Brent Theorem, we deduce:

$$
\begin{aligned}
T_p &\leq T_1/p + T_\infty & (7)\\
&= T_1/p + O(T_1/p) & (8)\\
&= \Theta(T_1/p) & (9)
\end{aligned}
$$

- This result suggests to operate in the range where T_1/p dominates T_∞.

Corollary 2

The greedy scheduler achieves linear speedup whenever $T_\infty = O(T_1/p)$.

From Graham - Brent Theorem, we deduce:

$$\begin{aligned} T_p &\leq T_1/p + T_\infty && (7) \\ &= T_1/p + O(T_1/p) && (8) \\ &= \Theta(T_1/p) && (9) \end{aligned}$$

- This result suggests to operate in the range where T_1/p dominates T_∞.
- As long as T_1/p dominates T_∞, all processors can be used efficiently.

Corollary 2

The greedy scheduler achieves linear speedup whenever $T_\infty = O(T_1/p)$.

From Graham - Brent Theorem, we deduce:

$$
\begin{aligned}
T_p &\leq T_1/p + T_\infty && (7)\\
&= T_1/p + O(T_1/p) && (8)\\
&= \Theta(T_1/p) && (9)
\end{aligned}
$$

- This result suggests to operate in the range where T_1/p dominates T_∞.
- As long as T_1/p dominates T_∞, all processors can be used efficiently.
- The quantity T_1/pT_∞ 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:

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.

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.

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.

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.

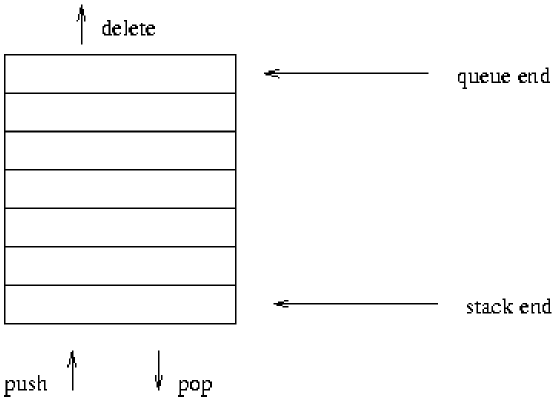
The work-stealing scheduler (1/9)

- `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.

The work-stealing scheduler (1/9)

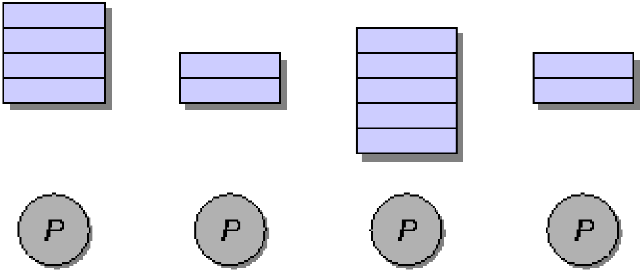
- `Cilk/Cilk++` randomized work-stealing scheduler load-balances the computation at run-time. Each processor maintains a ready deque:
  - ↳ A ready deque is a double ended queue, where each entry is a procedure instance that is ready to execute.
  - ↳ Adding a procedure instance to the bottom of the 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.

The work-stealing scheduler (2/9)

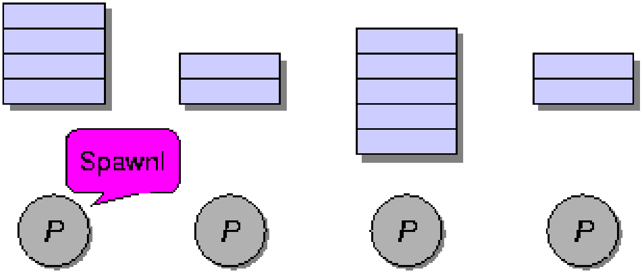


Each processor possesses a deque

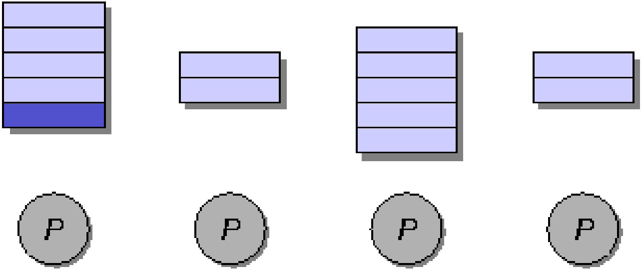
The work-stealing scheduler (3/9)



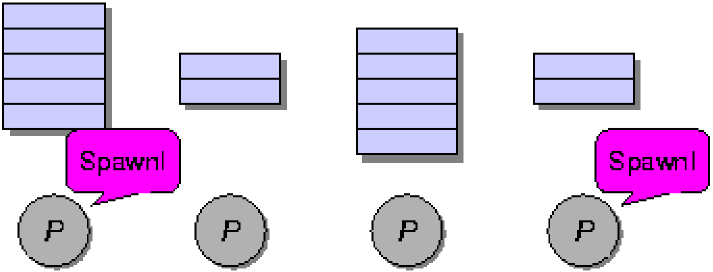
The work-stealing scheduler (3/9)



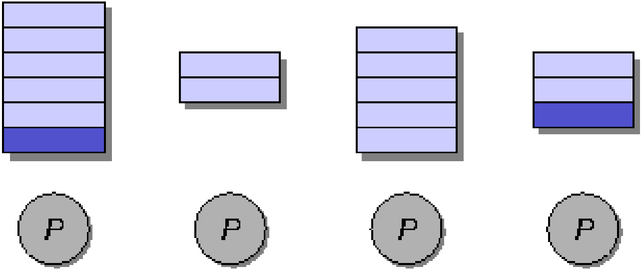
The work-stealing scheduler (4/9)



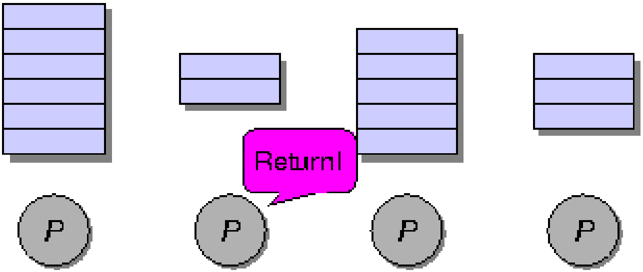
The work-stealing scheduler (4/9)



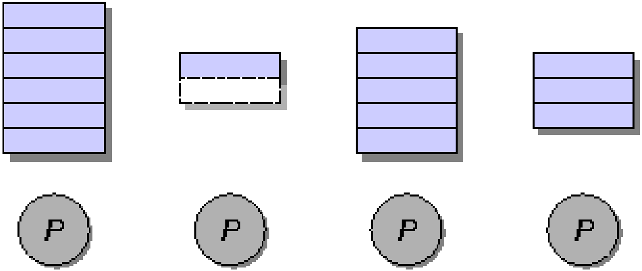
The work-stealing scheduler (5/9)



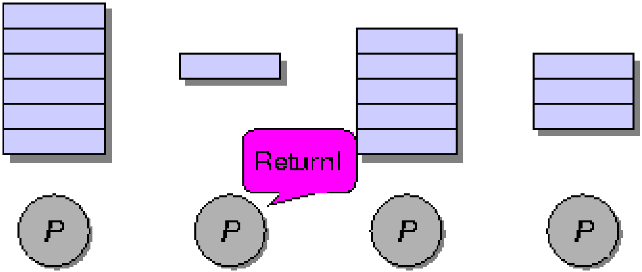
The work-stealing scheduler (5/9)



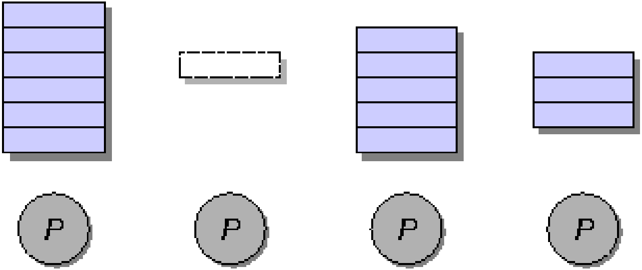
The work-stealing scheduler (6/9)



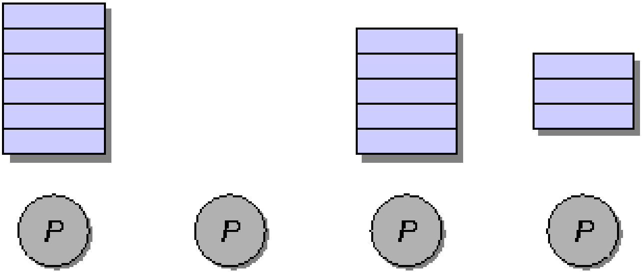
The work-stealing scheduler (6/9)



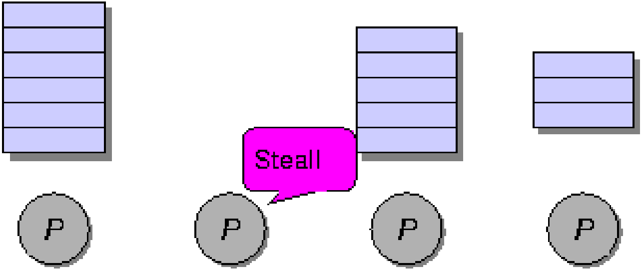
The work-stealing scheduler (7/9)



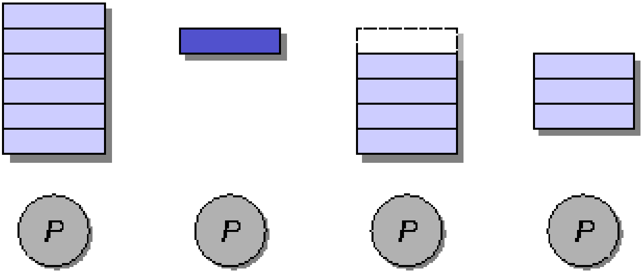
The work-stealing scheduler (7/9)



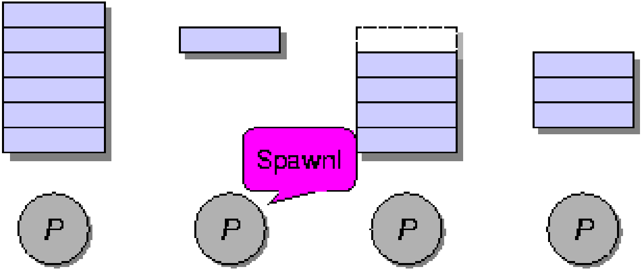
The work-stealing scheduler (8/9)



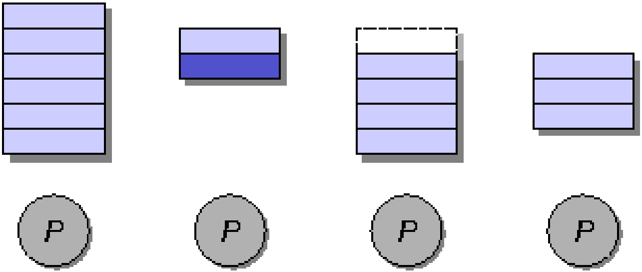
The work-stealing scheduler (8/9)



The work-stealing scheduler (9/9)



The work-stealing scheduler (9/9)



Performances of the work-stealing scheduler

Assume that

- each strand executes in unit time,

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,

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.

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

$$T_P = T_1/p + O(T_\infty)$$

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

$$T_P = T_1/p + O(T_\infty)$$

- A processor is either working or stealing.

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

$$T_P = T_1/p + O(T_\infty)$$

- A processor is either working or stealing.
- The total time all processors spend working is T_1, by definition of T_1.

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

$$T_P = T_1/p + O(T_\infty)$$

- 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.

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

$$T_P = T_1/p + O(T_\infty)$$

- 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 **expected** number of steals is $O(pT_\infty)$.

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

$$T_P = T_1/p + O(T_\infty)$$

- 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 **expected** number of steals is $O(pT_\infty)$.
- Since p processors are working/stealing together, the expected running time

$$T_P = \#\text{steps without steal} + \#\text{steps with steal} = T_1/p + O(pT_\infty)/p.$$

Overheads and burden

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

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.

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:

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*

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

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

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 $T_p = T_1/p + 1.7\ \mathrm{burden\_span}$, where $\mathrm{burden\_span}$ is 15000 instructions times the number of continuation edges along the critical path.

Span overhead

- Let T_1, T_∞, T_p be given. We want to refine the *randomized work-stealing complexity result*.

Span overhead

- Let T_1, T_∞, T_p be given. We want to refine the *randomized work-stealing complexity result*.
- The **span overhead** is the smallest constant c_∞ such that

$$T_p \leq T_1/p + c_\infty T_\infty.$$

Span overhead

- Let T_1, T_∞, T_p be given. We want to refine the *randomized work-stealing complexity result*.
- The **span overhead** is the smallest constant c_∞ such that

$$T_p \leq T_1/p + c_\infty T_\infty .$$

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

Span overhead

- Let T_1, T_∞, T_p be given. We want to refine the *randomized work-stealing complexity result*.
- The **span overhead** is the smallest constant c_∞ such that

$$T_p \leq T_1/p + c_\infty T_\infty.$$

- Recall that T_1/T_∞ is the maximum possible speed-up that the application can obtain.
- We call **parallel slackness assumption** the following property

$$T_1/T_\infty >> c_\infty p \tag{10}$$

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

Span overhead

- Let T_1, T_∞, T_p be given. We want to refine the *randomized work-stealing complexity result*.
- The **span overhead** is the smallest constant c_∞ such that

$$T_p \leq T_1/p + c_\infty T_\infty .$$

- Recall that T_1/T_∞ is the maximum possible speed-up that the application can obtain.
- We call **parallel slackness assumption** the following property

$$T_1/T_\infty >> c_\infty p \tag{10}$$

  that is, $c_\infty\, p$ is much smaller than the average parallelism .
- Under this assumption it follows that $T_1/p >> c_\infty T_\infty$ holds, thus c_∞ 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.

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**

$$c_1 = T_1/T_s$$

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**

$$c_1 = T_1/T_s$$

- Recall the expected running time: $T_P \leq T_1/P + c_\infty T_\infty$. Thus with the parallel slackness assumption we get

$$T_P \leq c_1 T_s/p + c_\infty T_\infty \simeq c_1 T_s/p. \quad (11)$$

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**

$$c_1 = T_1/T_s$$

- Recall the expected running time: $T_P \leq T_1/P + c_\infty T_\infty$. Thus with the parallel slackness assumption we get

$$T_P \leq c_1 T_s/p + c_\infty T_\infty \simeq c_1 T_s/p. \tag{11}$$

- We can now state the **work first principle** precisely

  Minimize c_1 , even at the expense of a larger c_∞.

  This is a key feature since it is conceptually easier to minimize c_1 rather than minimizing c_∞.

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**

$$c_1 = T_1/T_s$$

- Recall the expected running time: $T_P \le T_1/P + c_\infty T_\infty$. Thus with the parallel slackness assumption we get

$$T_P \le c_1 T_s/p + c_\infty T_\infty \simeq c_1 T_s/p. \qquad (11)$$

- We can now state the **work first principle** precisely

  Minimize c_1 , even at the expense of a larger c_∞.

  This is a key feature since it is conceptually easier to minimize c_1 rather than minimizing c_∞.

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

Outline

Outline

Orders of magnitude

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

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

$$0 \leq c_1 \; g(n) \leq f(n) \leq c_2 \; g(n). \tag{12}$$

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
$$0 \leq c_1 \; g(n) \leq f(n) \leq c_2 \; g(n). \qquad (12)$$
- 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
$$0 \leq f(n) \leq c_2 \; g(n). \qquad (13)$$

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

$$0 \;\leq\; c_1 \; g(n) \;\leq\; f(n) \;\leq\; c_2 \; g(n). \tag{12}$$

- 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

$$0 \;\leq\; f(n) \;\leq\; c_2 \; g(n). \tag{13}$$

- 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

$$0 \;\leq\; c_1 \; g(n) \;\leq\; f(n). \tag{14}$$

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
$$c_1 \, n^2 \quad \leq \quad \frac{1}{2}n^2 - 3n \quad \leq \quad c_2 \, n^2. \tag{15}$$
for $n \geq 12$ with $c_1 = \frac{1}{4}$ and $c_2 = \frac{1}{2}$.

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
$$c_1\, n^2 \quad \leq \quad \frac{1}{2}n^2 - 3n \quad \leq \quad c_2\, n^2. \tag{15}$$
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
$$max(f(n), g(n)) \in \Theta(f(n) + g(n)). \tag{16}$$
Indeed we have
$$\frac{1}{2}(f(n) + g(n)) \quad \leq \quad max(f(n), g(n)) \quad \leq \quad (f(n) + g(n)). \tag{17}$$

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
$$c_1\, n^2 \quad \leq \quad \frac{1}{2}n^2 - 3n \quad \leq \quad c_2\, n^2. \tag{15}$$
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
$$max(f(n), g(n)) \in \Theta(f(n) + g(n)). \tag{16}$$
Indeed we have
$$\frac{1}{2}(f(n) + g(n)) \quad \leq \quad max(f(n), g(n)) \quad \leq \quad (f(n) + g(n)). \tag{17}$$
- Assume a and b are positive real constants. Then we have
$$(n + a)^b \in \Theta(n^b). \tag{18}$$
Indeed for $n \geq a$ we have
$$0 \quad \leq \quad n^b \quad \leq \quad (n + a)^b \quad \leq \quad (2n)^b. \tag{19}$$
Hence we can choose $c_1 = 1$ and $c_2 = 2^b$.

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.

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.

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**

$$f(n) \in \mathcal{O}(g(n)) \iff g(n) \in \Omega(f(n)). \tag{20}$$

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**

$$f(n) \in \mathcal{O}(g(n)) \iff g(n) \in \Omega(f(n)). \tag{20}$$

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

$$f(n) = h(n) + \Theta(g(n)) \tag{21}$$

means

$$f(n) - h(n) \in \Theta(g(n)). \tag{22}$$

Another example

Let us give another fundamental example.

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 $a_d > 0$. Let k be an integer. Then we have:

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 $a_d > 0$. Let k be an integer. Then we have:

(1) if $k \geq d$ then $p(n) \in \mathcal{O}(n^k)$,

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 $a_d > 0$. Let k be an integer. Then we have:

(1) if $k \geq d$ then $p(n) \in \mathcal{O}(n^k)$,

(2) if $k \leq d$ then $p(n) \in \Omega(n^k)$,

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 $a_d > 0$. Let k be an integer. 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 $p(n) \in \Theta(n^k)$.

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 $a_d > 0$. Let k be an integer. 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

$$\Sigma_{k=1}^{k=n} k \ \in \ \Theta(n^2). \tag{23}$$

Outline

Divide-and-Conquer Algorithms

Divide-and-conquer algorithms proceed as follows.

Divide-and-Conquer Algorithms

Divide-and-conquer algorithms proceed as follows.

Divide the input problem into sub-problems.

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.

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.

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 $T(n)$.

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.

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.

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:

$$T(n) = a\,T(n/b) + f(n). \tag{24}$$

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.

Tree associated with a divide-and-conquer recurrence

Labeled tree associated with the equation. Assume n is a power of b, say $n = b^p$.

Tree associated with a divide-and-conquer recurrence

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 $\mathcal{A}(n)$ to it as follows.

Tree associated with a divide-and-conquer recurrence

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 $\mathcal{A}(n)$ to it as follows.

(1) If $n = 1$, then $\mathcal{A}(n)$ is reduced to a single leaf labeled $T(1)$.

Tree associated with a divide-and-conquer recurrence

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 $\mathcal{A}(n)$ to it as follows.

(1) If $n = 1$, then $\mathcal{A}(n)$ is reduced to a single leaf labeled $T(1)$.

(2) If $n > 1$, then the root of $\mathcal{A}(n)$ is labeled by $f(n)$ and $\mathcal{A}(n)$ possesses a labeled sub-trees all equal to $\mathcal{A}(n/b)$.

Tree associated with a divide-and-conquer recurrence

Labeled tree associated with the equation. Assume n is a power of b, say $n = b^p$. To *solve* the equation

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

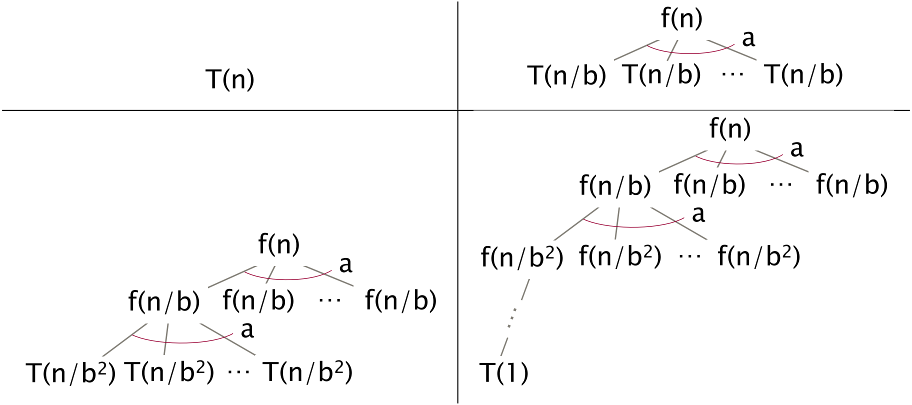
we can associate a labeled tree $\mathcal{A}(n)$ to it as follows.

(1) If $n = 1$, then $\mathcal{A}(n)$ is reduced to a single leaf labeled $T(1)$.

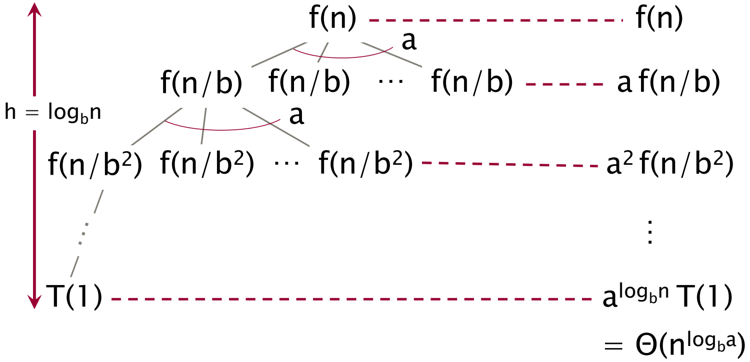
(2) If $n > 1$, then the root of $\mathcal{A}(n)$ is labeled by $f(n)$ and $\mathcal{A}(n)$ possesses a labeled sub-trees all equal to $\mathcal{A}(n/b)$.

The labeled tree $\mathcal{A}(n)$ 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)

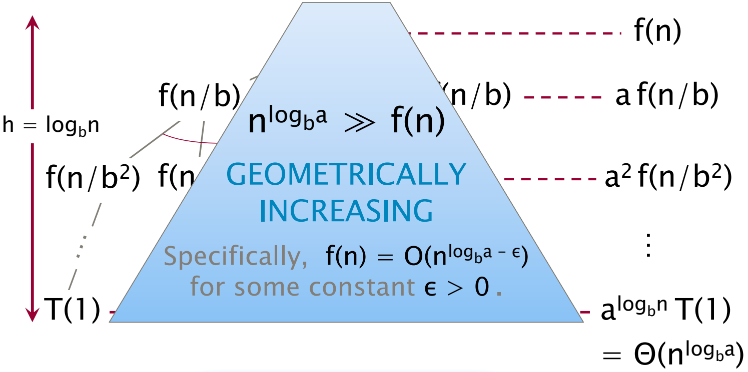


Solving divide-and-conquer recurrences (2/2)



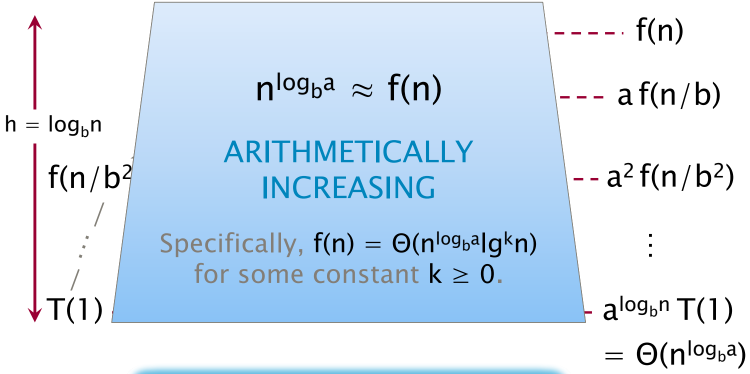
IDEA: Compare $n^{\log_b a}$ with $f(n)$.

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



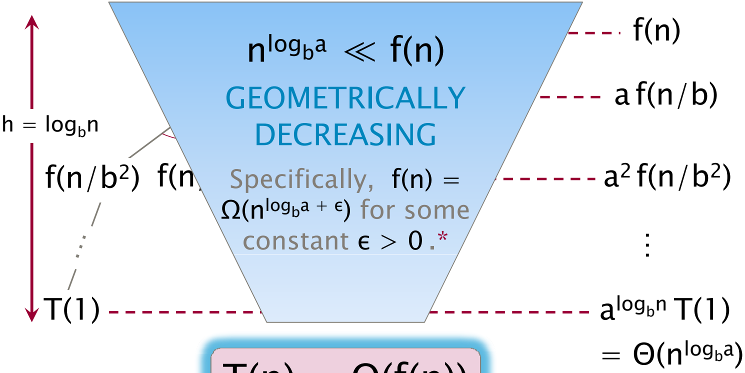
$$T(n) = \Theta(n^{\log_b a})$$

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



$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n))$$

Master Theorem: case where $f(n) \gg n^{\log_b a}$



$$\boxed{T(n) = \Theta(f(n))}$$

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

More examples

- Consider the relation:

$$T(n) = 2\,T(n/2) + n^2. \tag{25}$$

We obtain:

$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots + \frac{n^2}{2^p} + n\,T(1). \tag{26}$$

Hence we have:

$$T(n) \in \Theta(n^2). \tag{27}$$

More examples

- Consider the relation:

$$T(n) = 2\,T(n/2) + n^2. \tag{25}$$

We obtain:

$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots + \frac{n^2}{2^p} + n\,T(1). \tag{26}$$

Hence we have:

$$T(n) \in \Theta(n^2). \tag{27}$$

- Consider the relation:

$$T(n) = 3T(n/3) + n. \tag{28}$$

We obtain:

$$T(n) \in \Theta(\log_3(n)n). \tag{29}$$

Master Theorem when $b = 2$

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

Master Theorem when $b = 2$

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

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

Master Theorem when $b = 2$

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

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

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

Master Theorem when $b = 2$

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

Master Theorem when $b = 2$

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

$$T(n) \leq (2 - 2/n)\, f(n) + T(1) \in \mathcal{O}(f(n)), \tag{30}$$

Master Theorem when $b = 2$

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

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

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

Then for $n = 2^p$ we have:

(1) if $a = 1$ then

$$T(n) \leq (2 - 2/n)\,f(n) + T(1) \in \mathcal{O}(f(n)), \tag{30}$$

(2) if $a = 2$ then

$$T(n) \leq f(n)\log_2(n) + T(1)\,n \in \mathcal{O}(\log_2(n)\,f(n)), \tag{31}$$

Master Theorem when $b = 2$

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

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

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

Then for $n = 2^p$ we have:

(1) if $a = 1$ then

$$T(n) \leq (2 - 2/n) f(n) + T(1) \in \mathcal{O}(f(n)), \tag{30}$$

(2) if $a = 2$ then

$$T(n) \leq f(n) \log_2(n) + T(1) n \in \mathcal{O}(\log_2(n) f(n)), \tag{31}$$

(3) if $a \geq 3$ then

$$T(n) \leq \frac{2}{a-2} \left(n^{\log_2(a)-1} - 1 \right) f(n) + T(1) n^{\log_2(a)} \in \mathcal{O}(f(n) n^{\log_2(a)-1}). \tag{32}$$

Master Theorem when $b = 2$

Indeed

$$\begin{array}{rcl} T(2^p) & \leq & a\,T(2^{p-1}) + f(2^p) \\ & \leq & a\left[a\,T(2^{p-2}) + f(2^{p-1})\right] + f(2^p) \\ & = & a^2\,T(2^{p-2}) + a\,f(2^{p-1}) + f(2^p) \\ & \leq & a^2\left[a\,T(2^{p-3}) + f(2^{p-2})\right] + a\,f(2^{p-1}) + f(2^p) \\ & = & a^3\,T(2^{p-3}) + a^2\,f(2^{p-2}) + a\,f(2^{p-1}) + f(2^p) \\ & \leq & a^p\,T(s1) + \sigma_{j=0}^{j=p-1}\;a^j\,f(2^{p-j}) \end{array} \tag{33}$$

Master Theorem when $b = 2$

Moreover

$$\begin{array}{rcl} f(2^p) & \geq & 2\, f(2^{p-1}) \\ f(2^p) & \geq & 2^2\, f(2^{p-2}) \\ \vdots & \vdots & \vdots \\ f(2^p) & \geq & 2^j\, f(2^{p-j}) \end{array} \tag{34}$$

Thus

$$\Sigma_{j=0}^{j=p-1}\, a^j\, f(2^{p-j}) \;\leq\; f(2^p)\, \Sigma_{j=0}^{j=p-1} \left(\frac{a}{2}\right)^j. \tag{35}$$

Master Theorem when $b = 2$

Hence

$$T(2^p) \ \leq \ a^p\, T(1) + f(2^p)\, \Sigma_{j=0}^{j=p-1} \left(\frac{a}{2}\right)^j. \tag{36}$$

For $a = 1$ we obtain

$$\begin{array}{rcl} T(2^p) & \leq & T(1) + f(2^p)\, \Sigma_{j=0}^{j=p-1} \left(\frac{1}{2}\right)^j \\ & = & T(1) + f(2^p)\, \frac{\frac{1}{2^p} - 1}{\frac{1}{2} - 1} \\ & = & T(1) + f(n)\,(2 - 2/n). \end{array} \tag{37}$$

For $a = 2$ we obtain

$$\begin{array}{rcl} T(2^p) & \leq & 2^p\, T(1) + f(2^p)\, p \\ & = & n\, T(1) + f(n) \log_2(n). \end{array} \tag{38}$$

Master Theorem cheat sheet

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

$$T(n) = a\,T(n/b) + f(n). \tag{39}$$

Master Theorem cheat sheet

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

$$T(n) = a\,T(n/b) + f(n). \tag{39}$$

- We have:

$$(\exists \varepsilon > 0)\ f(n) \in O(n^{\log_b a - \varepsilon}) \implies T(n) \in \Theta(n^{\log_b a}) \tag{40}$$

Master Theorem cheat sheet

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

$$T(n) = a\,T(n/b) + f(n). \tag{39}$$

- We have:

$$(\exists \varepsilon > 0)\ f(n) \in O(n^{\log_b a - \varepsilon}) \implies T(n) \in \Theta(n^{\log_b a}) \tag{40}$$

- We have:

$$(\exists \varepsilon > 0)\ f(n) \in \Theta(n^{\log_b a} \log^k n) \implies T(n) \in \Theta(n^{\log_b a} \log^{k+1} n) \tag{41}$$

Master Theorem cheat sheet

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

$$T(n) = a\,T(n/b) + f(n). \tag{39}$$

- We have:

$$(\exists \varepsilon > 0)\ f(n) \in O(n^{\log_b a - \varepsilon}) \implies T(n) \in \Theta(n^{\log_b a}) \tag{40}$$

- We have:

$$(\exists \varepsilon > 0)\ f(n) \in \Theta(n^{\log_b a} \log^k n) \implies T(n) \in \Theta(n^{\log_b a} \log^{k+1} n) \tag{41}$$

- We have:

$$(\exists \varepsilon > 0)\ f(n) \in \Omega(n^{\log_b a + \varepsilon}) \implies T(n) \in \Theta(f(n)) \tag{42}$$

Master Theorem quizz!

- $T(n) = 4T(n/2) + n$

Master Theorem quizz!

- $T(n) = 4T(n/2) + n$
- $T(n) = 4T(n/2) + n^2$

Master Theorem quizz!

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

Master Theorem quizz!

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

Outline

Matrix multiplication

$$
\underbrace{\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}}_{C} = \underbrace{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix}}_{B}
$$

We will study three approaches:

Matrix multiplication

$$\underbrace{\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}}_{C} = \underbrace{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix}}_{B}$$

We will study three approaches:

- a naive and iterative one

Matrix multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

$$C \qquad\qquad\qquad A \qquad\qquad\qquad B$$

We will study three approaches:

- a naive and iterative one
- a divide-and-conquer one

Matrix multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

$$\qquad C \qquad\qquad\qquad\qquad A \qquad\qquad\qquad\qquad B$$

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];
  }
}
```

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**: ?

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**: ?

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];
  }
}
```

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**: $\Theta(n^3)$

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**: $\Theta(n)$

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**: $\Theta(n^2)$

Matrix multiplication based on block decomposition

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

The divide-and-conquer approach is simply the one based on blocking, presented in the previous 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);
             MMult(D21, A22, B21, n/2, size);
  cilk_sync;
  MAdd(C, D, n, size); // C += D;
  delete[] D;
}
```

Work ? **Span** ? **Parallelism** ?

Divide-and-conquer matrix multiplication

```
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 $n \times n$ $\textsc{Add}$ and $\textsc{Mult}$.

Divide-and-conquer matrix multiplication

```
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 $n \times n$ $\textsc{Add}$ and $\textsc{Mult}$.
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$

Divide-and-conquer matrix multiplication

```
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 $n \times n$ $\textsc{Add}$ and $\textsc{Mult}$.
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$
- $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$

Divide-and-conquer matrix multiplication

```
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 $n \times n$ ADD and MULT.
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$
- $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$
- $M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$

Divide-and-conquer matrix multiplication

```
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 $n \times n$ ADD and MULT.
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$
- $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$
- $M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$
- $M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$

Divide-and-conquer matrix multiplication

```
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 $n \times n$ ADD and MULT.
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$
- $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$
- $M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$
- $M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$
- $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$

Divide-and-conquer matrix multiplication: No temporaries!

```
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;  }
```

Work ? **Span** ? **Parallelism** ?

Divide-and-conquer matrix multiplication: No temporaries!

```
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 $n \times n$ $\textsc{Mult-Add}$.

Divide-and-conquer matrix multiplication: No temporaries!

```
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.
- $MA_1(n) = \Theta(n^3)$

Divide-and-conquer matrix multiplication: No temporaries!

```
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.
- $MA_1(n) = \Theta(n^3)$
- $MA_\infty(n) = 2MA_\infty(n/2) + \Theta(1) = \Theta(n)$

Divide-and-conquer matrix multiplication: No temporaries!

```
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.
- $MA_1(n) = \Theta(n^3)$
- $MA_\infty(n) = 2MA_\infty(n/2) + \Theta(1) = \Theta(n)$
- $MA_1(n)/MA_\infty(n) = \Theta(n^2)$

Divide-and-conquer matrix multiplication: No temporaries!

```
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.

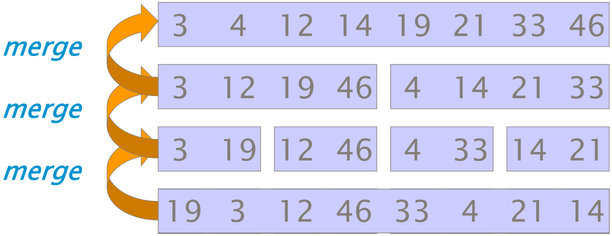
Outline

Merging two sorted arrays

```
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 $\Theta(n)$.

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);
  }
```

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);
  }
```

- **Work**?

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);
  }
```

- **Work**?
- **Span**?

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);
  }
```

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);
  }
```

- $T_1(n) = 2T_1(n/2) + \Theta(n)$ thus $T_1(n) = \Theta(n \lg n)$.

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);
  }
```

- $T_1(n) = 2T_1(n/2) + \Theta(n)$ thus $T_1(n) = \Theta(n \lg n)$.
- $T_\infty(n) = T_\infty(n/2) + \Theta(n)$ thus $T_\infty(n) = \Theta(n)$.

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);
  }
```

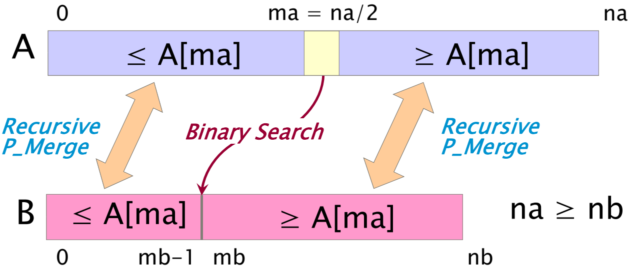
- $T_1(n) = 2T_1(n/2) + \Theta(n)$ thus $T_1(n) = \Theta(n \lg n)$.
- $T_\infty(n) = T_\infty(n/2) + \Theta(n)$ thus $T_\infty(n) = \Theta(n)$.
- $T_1(n)/T_\infty(n) = \Theta(\lg n)$. **Puny parallelism!**

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);
  }
```

- $T_1(n) = 2T_1(n/2) + \Theta(n)$ thus $T_1(n) = \Theta(n \lg n)$.
- $T_\infty(n) = T_\infty(n/2) + \Theta(n)$ thus $T_\infty(n) = \Theta(n)$.
- $T_1(n)/T_\infty(n) = \Theta(\lg n)$. **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;
  }
}
```

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 coarsen the base case for efficiency.

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 coarsen the base case for efficiency.
- **Work**? **Span**?

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; }  }
```

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; }  }
```

- Let $PM_p(n)$ be the *p-processor* running time of $\mathrm{P\text{-}M{\small ERGE}}$.

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; }  }
```

- Let $PM_p(n)$ be the p-*processor* running time of $\text{P-M}\textsc{erge}$.
- In the worst case, the span of $\text{P-M}\textsc{erge}$ is

$$PM_\infty(n) \le PM_\infty(3n/4) + \Theta(\lg n) = O(\lg^2 n)$$

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; }  }
```

- Let $PM_p(n)$ be the *p-processor* running time of P-MERGE.
- In the worst case, the span of P-MERGE is

$$PM_\infty(n) \leq PM_\infty(3n/4) + \Theta(\lg n) = O(\lg^2 n)$$

- 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 $1/4 \leq \alpha \leq 3/4$.

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.

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 $1/4 \leq \alpha \leq 3/4$ and all $n > 1$.

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 $1/4 \leq \alpha \leq 3/4$ and all $n > 1$.
- After substitution, this hypothesis implies: $PM_1(n) \leq a(\alpha + (1-\alpha)n - b\lg(\alpha n) - b\lg n + \Theta(\lg n)$.

Analyzing parallel merge

- Recall $PM_1(n) \le PM_1(\alpha n) + PM_1((1-\alpha)n) + \Theta(\lg n)$ for some $1/4 \le \alpha \le 3/4$.
- To solve this **hairy equation** we use the substitution method.
- We assume there exist some constants $a, b > 0$ such that $PM_1(n) \le an - b \lg n$ holds for all $1/4 \le \alpha \le 3/4$ and all $n > 1$.
- After substitution, this hypothesis implies: $PM_1(n) \le a(\alpha + (1-\alpha)n - b \lg(\alpha n) - b \lg n + \Theta(\lg n)$.
- We can pick b large enough such that we have $PM_1(n) \le an - b \lg n$ for all $1/4 \le \alpha \le 3/4$ and all $n > 1$.

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 $1/4 \leq \alpha \leq 3/4$ and all $n > 1$.
- After substitution, this hypothesis implies: $PM_1(n) \leq a(\alpha + (1-\alpha)n - b\lg(\alpha n) - b\lg n + \Theta(\lg n)$.
- 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 $n > 1$.
- Then pick a large enough to satisfy the base conditions, leading to $PM_1(n) \in O(n)$.

Analyzing parallel merge

- Recall $PM_1(n) \le PM_1(\alpha n) + PM_1((1-\alpha)n) + \Theta(\lg n)$ for some $1/4 \le \alpha \le 3/4$.
- To solve this **hairy equation** we use the substitution method.
- We assume there exist some constants $a, b > 0$ such that $PM_1(n) \le an - b\lg n$ holds for all $1/4 \le \alpha \le 3/4$ and all $n > 1$.
- After substitution, this hypothesis implies: $PM_1(n) \le a(\alpha + (1-\alpha)n - b\lg(\alpha n) - b\lg n + \Theta(\lg n)$.
- We can pick b large enough such that we have $PM_1(n) \le an - b\lg n$ for all $1/4 \le \alpha \le 3/4$ and all $n > 1$.
- Then pick a large enough to satisfy the base conditions, leading to $PM_1(n) \in O(n)$.
- Since we clearly have $PM_1(n) \in \Omega(n)$ (because n array elements are accessed anyway), we finally have $PM_1(n) = \Theta(n)$.

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);
  }
}
```

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);
  }
}
```

- **Work**?

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);
  }
}
```

- **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);
  }
}
```

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 $T_1(n) = \Theta(n\log(n))$.

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

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

.

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 $\Theta(n \lg n)/\Theta(\lg^3 n) = \Theta(n/\lg^2 n)$.

Outline

Tableau construction

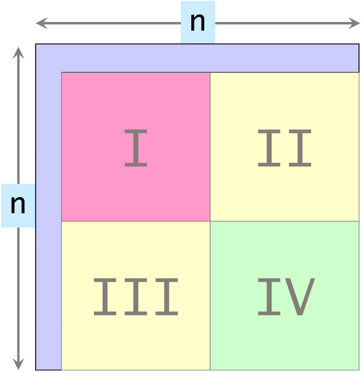
00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Constructing a tableau A satisfying a relation of the form:

$$A[i,j] = R(A[i-1,j], A[i-1,j-1], A[i,j-1]). \tag{43}$$

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

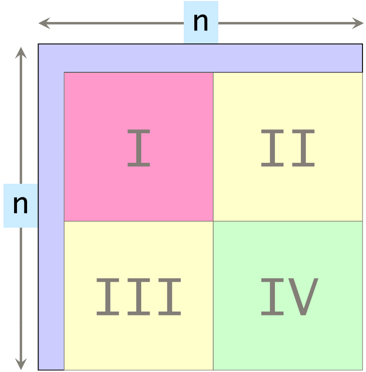
Recursive construction



Parallel code

```
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

Recursive construction

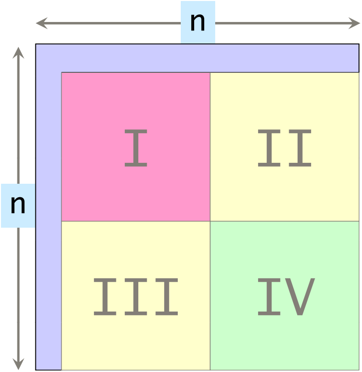


Parallel code

```
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

- $T_1(n) = 4T_1(n/2) + \Theta(1)$, thus $T_1(n) = \Theta(n^2)$.

Recursive construction

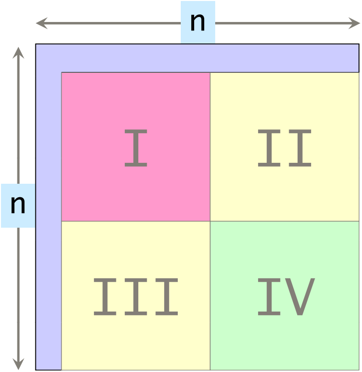


Parallel code

```
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

- $T_1(n) = 4T_1(n/2) + \Theta(1)$, thus $T_1(n) = \Theta(n^2)$.
- $T_\infty(n) = 3T_\infty(n/2) + \Theta(1)$, thus $T_\infty(n) = \Theta(n^{\log_2 3})$.

Recursive construction

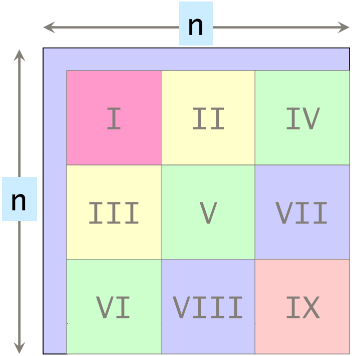


Parallel code

```
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

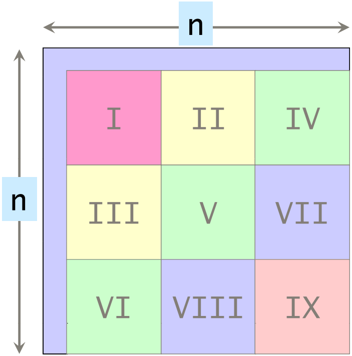
- $T_1(n) = 4T_1(n/2) + \Theta(1)$, thus $T_1(n) = \Theta(n^2)$.
- $T_\infty(n) = 3T_\infty(n/2) + \Theta(1)$, thus $T_\infty(n) = \Theta(n^{\log_2 3})$.
- **Parallelism**: $\Theta(n^{2-\log_2 3}) = \Omega(n^{0.41})$.

A more parallel construction




```
I;
cilk_spawn II;
III;
cilk_sync;
cilk_spawn IV;
cilk_spawn V;
VI;
cilk_sync;
cilk_spawn VII;
VIII;
cilk_sync;
IX;
```

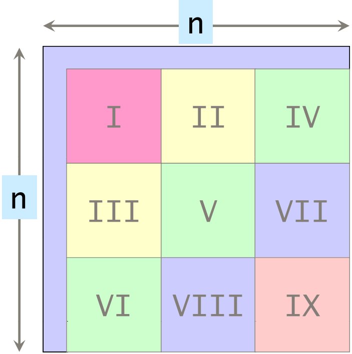
A more parallel construction




```
I;
cilk_spawn II;
III;
cilk_sync;
cilk_spawn IV;
cilk_spawn V;
VI;
cilk_sync;
cilk_spawn VII;
VIII;
cilk_sync;
IX;
```

- $T_1(n) = 9T_1(n/3) + \Theta(1)$, thus $T_1(n) = \Theta(n^2)$.

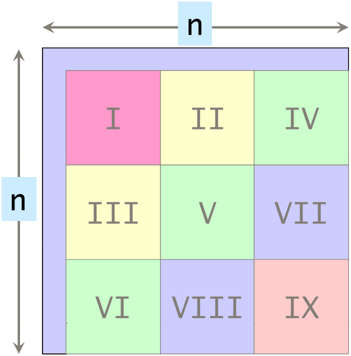
A more parallel construction




```
I;
cilk_spawn II;
III;
cilk_sync;
cilk_spawn IV;
cilk_spawn V;
VI;
cilk_sync;
cilk_spawn VII;
VIII;
cilk_sync;
IX;
```

- $T_1(n) = 9T_1(n/3) + \Theta(1)$, thus $T_1(n) = \Theta(n^2)$.
- $T_\infty(n) = 5T_\infty(n/3) + \Theta(1)$, thus $T_\infty(n) = \Theta(n^{\log_3 5})$.

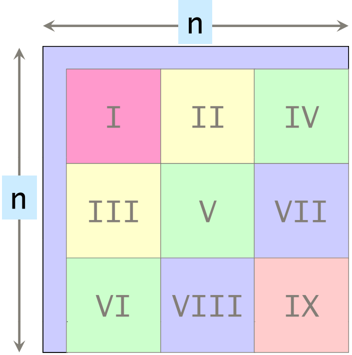
A more parallel construction




```
I;
cilk_spawn II;
III;
cilk_sync;
cilk_spawn IV;
cilk_spawn V;
VI;
cilk_sync;
cilk_spawn VII;
VIII;
cilk_sync;
IX;
```

- $T_1(n) = 9T_1(n/3) + \Theta(1)$, 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**: $\Theta(n^{2-\log_3 5}) = \Omega(n^{0.53})$.

A more parallel construction




```
I;
cilk_spawn II;
III;
cilk_sync;
cilk_spawn IV;
cilk_spawn V;
VI;
cilk_sync;
cilk_spawn VII;
VIII;
cilk_sync;
IX;
```

- $T_1(n) = 9T_1(n/3) + \Theta(1)$, 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.

Acknowledgements

- Charles E. Leiserson (MIT) and Matteo Frigo (Oracle) for providing me with the sources of their lecture notes.
- My former students Yuzhen Xie and Liyun Li or generating the experimental data.

References

- Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, Pages: 212-223. June, 1998.
- Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing, 55-69, August 25, 1996.
- Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM, Vol. 46, No. 5, pp. 720-748. September 1999.