Multicore Programming – MPRI 2.37.1

Deterministic Parallel Programming

Adrien Guatto
adrien@guatto.org

IRIF

2019-2020

https://www.irif.fr/~guatto/teaching/multicore/

Why Parallel Programming?

In an ideal world, software should be:

1. *correct* — doing the thing it has been designed to do,
2. *maintainable* — easy to modify, evolve, and port to new systems,
3. *performant* — as fast as possible.

Our main tool to achieve the first two goals is abstraction:

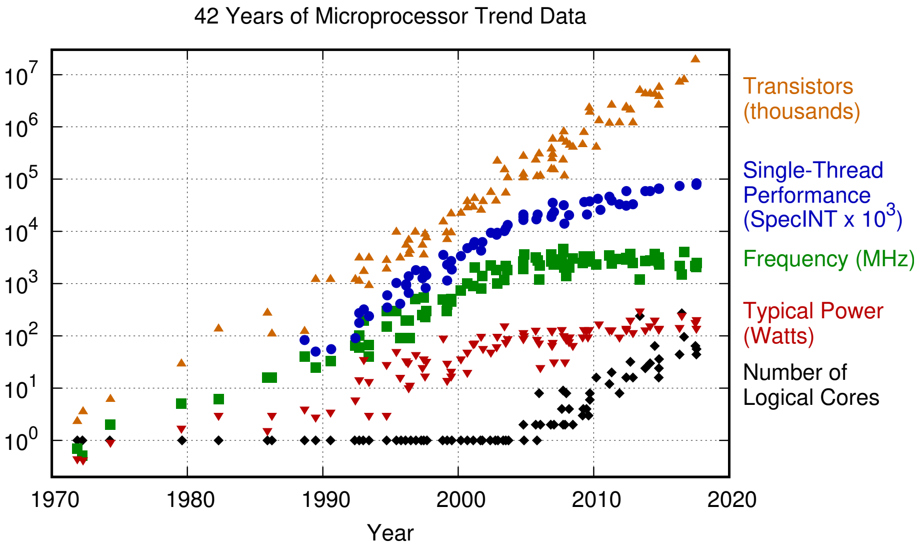
- hardware-agnostic programming languages,
- modular program construction (via modules, objects, packages…).

However, the third goal requires staying reasonably close to the hardware.

A not-so-innocent question

What does high-performance hardware look like these days?

Why Parallel Programming?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Parallel Programming Methodologies

There are many approaches, some competing, some complementary, to the construction of parallel software. Here are some examples:

- Shared-memory programming.
  - Classic system programming with OS threads, locks, semaphores, etc.
    - Add *weak memory models* for a new twist! See Luc's part of the course.
  - Deterministic parallelism à la Cilk, OpenMP, Intel TBB, etc.
- Message passing.
  - Various kind of actor languages and libraries, e.g., Erlang or Akka.
  - Cluster and grid computing, e.g. with MPI.
- Dataflow computing.
  - Futures, promises, I-structures, Kahn networks (cf. MPRI 2.23.1).
- Automatic parallelization of sequential code.
  - Dependence analysis, polytope model.

Parallel Programming Methodologies

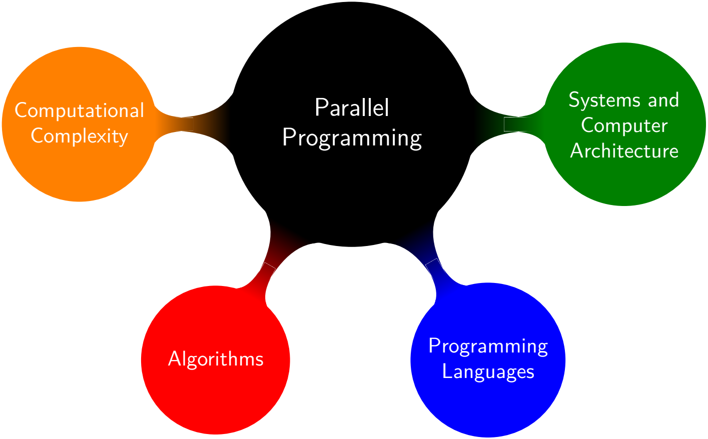
There are many approaches, some competing, some complementary, to the construction of parallel software. Here are some examples:

- Shared-memory programming.
  - Classic system programming with OS threads, locks, semaphores, etc.
    - Add *weak memory models* for a new twist! See Luc's part of the course.
  - Deterministic parallelism à la Cilk, OpenMP, Intel TBB, etc.
- Message passing.
  - Various kind of actor languages and libraries, e.g., Erlang or Akka.
  - Cluster and grid computing, e.g. with MPI.
- Dataflow computing.
  - Futures, promises, I-structures, Kahn networks (cf. MPRI 2.23.1).
- Automatic parallelization of sequential code.
  - Dependence analysis, polytope model.

This part of the course

We will focus on deterministic parallel programming in Cilk.

Deterministic Parallel Programming: Outline



1. [Jan. 16] Programming task-parallel algorithms, an introduction.
2. [Jan. 30] Implementing task parallelism on multicore processors.
3. [Feb. 20] Formalizing task parallelism, its semantics and its cost.

Lecture 1

Programming Task-Parallel Algorithms: An Introduction

Back to Basics: MergeSort

```
void mergesort_seq(int *B, int *A,
                   size_t lo, size_t hi) {
  switch (range_size(lo, hi)) {
  case 0: break;
  case 1: B[lo] = A[lo]; break;
  default:
    {
      size_t mid = midpoint(lo, hi);
      mergesort_seq(A, B, lo, mid);
      mergesort_seq(A, B, mid, hi);
      merge_seq(B + lo, A, lo, mid, A, mid, hi);
      break;
    }
  }
}
```

MergeSort Performance Results

Setup

All our experiments run on `ginette`:

- 40-core Intel Xeon E5-4640 (2.2 Ghz) with 2-way SMT,
- 750 GB of memory,
- GNU/Linux Debian 4.9 with GCC 6.3 and glibc 2.24.

MergeSort Performance Results

Setup

All our experiments run on `ginette`:

- 40-core Intel Xeon E5-4640 (2.2 Ghz) with 2-way SMT,
- 750 GB of memory,
- GNU/Linux Debian 4.9 with GCC 6.3 and glibc 2.24.

Sorting a 16 MB file of random 32-bit integers with this naive merge sort takes ~2.47 s on `ginette`. Can we exploit parallelism to do better?

MergeSort Performance Results

Setup

All our experiments run on `ginette`:

- 40-core Intel Xeon E5-4640 (2.2 Ghz) with 2-way SMT,
- 750 GB of memory,
- GNU/Linux Debian 4.9 with GCC 6.3 and glibc 2.24.

Sorting a 16 MB file of random 32-bit integers with this naive merge sort takes ~2.47 s on `ginette`. Can we exploit parallelism to do better?

We are simple people: let's use pthreads to parallelize the *divide* step.

Naive Pthreaded MergeSort: Divide Step

```
size_t mid = midpoint(lo, hi);
// Do one recursive call in its own thread.
pa->A = A;
pa->B = B;
pa->lo = lo;
pa->hi = mid;
if (pthread_create(&t, NULL,
                   mergesort_par_stub, pa))
  die("pthread_create()");
// Do the second call sequentially.
mergesort_par(A, B, mid, hi);
// Wait for the spawned thread to terminate.
if (pthread_join(t, NULL))
  die("pthread_join()");
```

Not so Fast

Good, let's try sorting a large-ish array, e.g., 64 MB.

Not so Fast

Good, let's try sorting a large-ish array, e.g., 64 MB.

```
aguatto@ginette$ ./msort-par-pthread.bin -i /tmp/16777216
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pppthread_create(): Cannot allocate memory
tpthread_create(): Cannot allocate memory
ththread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
```

Not so Fast

Good, let's try sorting a large-ish array, e.g., 64 MB.

```
aguatto@ginette$ ./msort-par-pthread.bin -i /tmp/16777216
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pppthread_create(): Cannot allocate memory
tpthread_create(): Cannot allocate memory
ththread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
```

On `ginette`, the failure occurs nondeterministically after creating around 50k pthreads. How could we try to fix this?

- Create fewer pthreads. *But how do we know which ones to create?*
- Lower their cost, e.g., shrink their stacks. *Not enough here, I tried!*

Not so Fast

Good, let's try sorting a large-ish array, e.g., 64 MB.

```
aguatto@ginette$ ./msort-par-pthread.bin -i /tmp/16777216
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
pppthread_create(): Cannot allocate memory
tpthread_create(): Cannot allocate memory
ththread_create(): Cannot allocate memory
pthread_create(): Cannot allocate memory
```

On `ginette`, the failure occurs nondeterministically after creating around 50k pthreads. How could we try to fix this?

- Create fewer pthreads. *But how do we know which ones to create?*
- Lower their cost, e.g., shrink their stacks. *Not enough here, I tried!*

We will rather use higher-level abstractions.

Task-Parallel Programming

We've seen that pthreads are...

- heavy: each pthread mobilizes a lot of resources (e.g., a stack), even for suspended pthreads,
- not programmer-friendly: using the right amount of threads is hard.

How would we like to program instead?

Task-Parallel Programming

We've seen that pthreads are...

- heavy: each pthread mobilizes a lot of resources (e.g., a stack), even for suspended pthreads,
- not programmer-friendly: using the right amount of threads is hard.

How would we like to program instead?

Parallel Programming: The Ideal Model

- Express only *logical* parallelism, abstract hardware parallelism away.
- Reason about asymptotic performance in an analytic fashion.

Task-Parallel Programming

We've seen that pthreads are...

- heavy: each pthread mobilizes a lot of resources (e.g., a stack), even for suspended pthreads,
- not programmer-friendly: using the right amount of threads is hard.

How would we like to program instead?

Parallel Programming: The Ideal Model

- Express only *logical* parallelism, abstract hardware parallelism away.
- Reason about asymptotic performance in an analytic fashion.

Task parallelism, as implemented in Cilk [Frigo et al., 1998], offers lightweight, 2nd-class threads that we can almost use as in the ideal model.

Cilk in a Nutshell

Cilk is an implementation of task parallelism in C and C++.

We only use three simple constructs.

Cilk in a Nutshell

Cilk is an implementation of task parallelism in C and C++.

We only use three simple constructs.

1. `cilk_spawn`: create a new parallel task capable of executing immediately, in parallel with the spawning task.

Cilk in a Nutshell

Cilk is an implementation of task parallelism in C and C++.

We only use three simple constructs.

1. `cilk_spawn`: create a new parallel task capable of executing immediately, in parallel with the spawning task.
2. `cilk_sync`: wait for all parallel tasks created since either the beginning of execution or the last `cilk_sync` call.

Cilk in a Nutshell

Cilk is an implementation of task parallelism in C and C++.

We only use three simple constructs.

1. `cilk_spawn`: create a new parallel task capable of executing immediately, in parallel with the spawning task.
2. `cilk_sync`: wait for all parallel tasks created since either the beginning of execution or the last `cilk_sync` call.
3. `cilk_for`: run a for loop in parallel (expressible via `spawn`/`sync`).

Cilk in a Nutshell

Cilk is an implementation of task parallelism in C and C++.

We only use three simple constructs.

1. `cilk_spawn`: create a new parallel task capable of executing immediately, in parallel with the spawning task.
2. `cilk_sync`: wait for all parallel tasks created since either the beginning of execution or the last `cilk_sync` call.
3. `cilk_for`: run a for loop in parallel (expressible via `spawn`/`sync`).

Tasks have access to shared memory but data races disallowed.

Cilk in a Nutshell

Cilk is an implementation of task parallelism in C and C++.

We only use three simple constructs.

1. `cilk_spawn`: create a new parallel task capable of executing immediately, in parallel with the spawning task.
2. `cilk_sync`: wait for all parallel tasks created since either the beginning of execution or the last `cilk_sync` call.
3. `cilk_for`: run a for loop in parallel (expressible via `spawn`/`sync`).

Tasks have access to shared memory but data races disallowed.

Which Cilk Implementation?

I use the latest open-source implementation of Cilk, available at

`http://cilk.mit.edu`.

It is currently a bit fiddly to install (ask me in case of trouble!).

Merge Sort in Cilk: Divide Step

```
size_t mid = midpoint(lo, hi);
cilk_spawn mergesort_par(A, B, lo, mid);
mergesort_par(A, B, mid, hi);
cilk_sync;
```

Cilk Programs and Their Serial Elisions

Here is an inefficient yet functionally correct implementation of Cilk:

Cilk Programs and Their Serial Elisions

Here is an inefficient yet functionally correct implementation of Cilk:

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

Cilk Programs and Their Serial Elisions

Here is an inefficient yet functionally correct implementation of Cilk:

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

Of course, real implementations of Cilk map tasks onto hardware parallelism, as we will see in later parts of the course.

Cilk Programs and Their Serial Elisions

Here is an inefficient yet functionally correct implementation of Cilk:

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

Of course, real implementations of Cilk map tasks onto hardware parallelism, as we will see in later parts of the course.

Yet, the above implementation supports an important idea.

Cilk Programs and Their Serial Elisions

Here is an inefficient yet functionally correct implementation of Cilk:

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

Of course, real implementations of Cilk map tasks onto hardware parallelism, as we will see in later parts of the course.

Yet, the above implementation supports an important idea.

A Core Principle of Cilk

- Every Cilk program has a canonical *serial elision*.
- A correct Cilk program and its serial elision have the same result.

Corollary: External Determinism

All the executions of a correct Cilk program compute the same result.

From External to Internal Determinism

Many Cilk programs enjoy an even strong property: *internal determinism*.

- Key to a well-defined notion of asymptotic performance.
- Intuitively: for a fixed input, gives rise a unique "parallel computation".
- We need to make the latter more formal.

From External to Internal Determinism

Many Cilk programs enjoy an even strong property: *internal determinism*.

- Key to a well-defined notion of asymptotic performance.
- Intuitively: for a fixed input, gives rise a unique "parallel computation".
- We need to make the latter more formal.

Computation Graphs of Cilk Programs [Blumofe and Leiserson, 1994]

Every run of a Cilk program induces a directed acyclic graph:

- vertices are unit-time operations performed during execution,
- edges are dependencies induced by program order, spawn, and sync.

From External to Internal Determinism

Many Cilk programs enjoy an even strong property: *internal determinism*.

- Key to a well-defined notion of asymptotic performance.
- Intuitively: for a fixed input, gives rise a unique "parallel computation".
- We need to make the latter more formal.

Computation Graphs of Cilk Programs [Blumofe and Leiserson, 1994]

Every run of a Cilk program induces a directed acyclic graph:

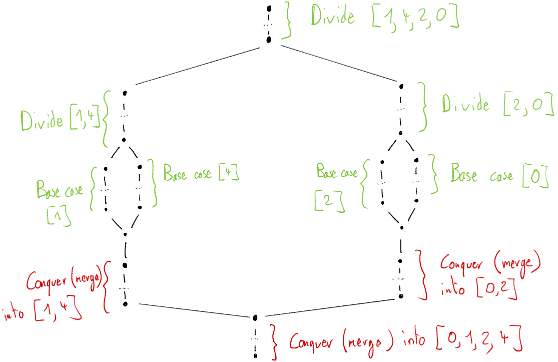
- vertices are unit-time operations performed during execution,
- edges are dependencies induced by program order, spawn, and sync.

Internal Determinism

A Cilk program is *internally deterministic* when, for fixed inputs, all its executions induce the same computation graph.

Computation Graph of our Cilky Merge Sort

Imagine running our sort on $[1, 4, 2, 0]$. Its computation graph looks like:



- The goal of the Cilk runtime system is to schedule this graph, i.e., execute each node after its predecessors have been executed.
- It strives to minimize running time by exploiting hardware parallelism.
- Which structural features of the graph control parallel efficiency?

The Work/Span Model and Brent's Theorem (1/2)

The two main parameters of a computation graph are its *work* and *span*.

- Its work W is the total number of nodes in the graph.
- Its span S is the length of its longest path.

Additionally, we define its *parallelism* P as simply W/S.

The Work/Span Model and Brent's Theorem (1/2)

The two main parameters of a computation graph are its *work* and *span*.

- Its work W is the total number of nodes in the graph.
- Its span S is the length of its longest path.

Additionally, we define its *parallelism* P as simply W/S.

Theorem (Brent [1974])

A computation can execute on p cores within T_p units of time, with

$$T_p \leq \frac{W}{p} + S.$$

The Work/Span Model and Brent's Theorem (1/2)

The two main parameters of a computation graph are its *work* and *span*.

- Its work W is the total number of nodes in the graph.
- Its span S is the length of its longest path.

Additionally, we define its *parallelism* P as simply W/S.

Theorem (Brent [1974])

A computation can execute on p cores within T_p units of time, with

$$T_p \leq \frac{W}{p} + S.$$

This justifies trying to maximize P while keeping W under control: since

$$T_p \leq \frac{W}{p}\left(1 + \frac{p}{P}\right),$$

when p is much smaller than P, we get an optimal (linear) speedup.

The Work/Span Model and Brent's Theorem (2/2)

Informal proof.

1. Say that a node is *at depth* i if it has exactly $i - 1$ predecessors.

The Work/Span Model and Brent's Theorem (2/2)

Informal proof.

1. Say that a node is *at depth i* if it has exactly $i - 1$ predecessors.
2. Consider an idealized scheduler working in rounds, each of which executes all the nodes at depth $i \in [1, S]$.

The Work/Span Model and Brent's Theorem (2/2)

Informal proof.

1. Say that a node is *at depth* i if it has exactly $i - 1$ predecessors.
2. Consider an idealized scheduler working in rounds, each of which executes all the nodes at depth $i \in [1, S]$.
3. All cores proceed independently within the same round, since all the predecesors of any node at that depth have already been executed.

The Work/Span Model and Brent's Theorem (2/2)

Informal proof.

1. Say that a node is *at depth* i if it has exactly $i - 1$ predecessors.
2. Consider an idealized scheduler working in rounds, each of which executes all the nodes at depth $i \in [1, S]$.
3. All cores proceed independently within the same round, since all the predecesors of any node at that depth have already been executed.
4. Thus, writing W_i for the number of nodes at depth i, we have

$$T_p = \sum_{i=1}^{S} \left\lceil \frac{W_i}{p} \right\rceil \leq \sum_{i=1}^{S} \left(\frac{W_i}{p} + 1 \right) = \frac{\sum_{i=1}^{S} W_i}{p} + S = \frac{W}{p} + S. \quad \square$$

The Work/Span Model and Brent's Theorem (2/2)

Informal proof.

1. Say that a node is *at depth i* if it has exactly $i - 1$ predecessors.
2. Consider an idealized scheduler working in rounds, each of which executes all the nodes at depth $i \in [1, S]$.
3. All cores proceed independently within the same round, since all the predecesors of any node at that depth have already been executed.
4. Thus, writing W_i for the number of nodes at depth i, we have

$$T_p = \sum_{i=1}^{S} \left\lceil \frac{W_i}{p} \right\rceil \leq \sum_{i=1}^{S} \left(\frac{W_i}{p} + 1 \right) = \frac{\sum_{i=1}^{S} W_i}{p} + S = \frac{W}{p} + S. \quad \square$$

Important Caveats

- Rigorous proofs use abstract machine models, e.g., PRAM.
- This scheduler is too centralized to be realistic. Wait for lecture 2!

Analysis of Cilky Merge Sort

Analysis

Assume the merge step is linear in work and span, and n is a power of 2. This leads to the following recurrence equations:

$$W(n) = 2W(n/2) + O(n), \qquad S(n) = S(n/2) + O(n).$$

Solving them using standard techniques, we get

$$W(n) = O(n \log n), \qquad S(n) = O(n).$$

Analysis of Cilky Merge Sort

Analysis

Assume the merge step is linear in work and span, and n is a power of 2. This leads to the following recurrence equations:

$$W(n) = 2W(n/2) + O(n), \qquad S(n) = S(n/2) + O(n).$$

Solving them using standard techniques, we get

$$W(n) = O(n \log n), \qquad S(n) = O(n).$$

Observations

- We perform no more work than sequential sorts (*work-efficiency*).
- The algorithm contains $W/S = O(\log n)$ parallelism, which is small.

Analysis of Cilky Merge Sort

Analysis

Assume the merge step is linear in work and span, and n is a power of 2. This leads to the following recurrence equations:

$$W(n) = 2W(n/2) + O(n), \qquad S(n) = S(n/2) + O(n).$$

Solving them using standard techniques, we get

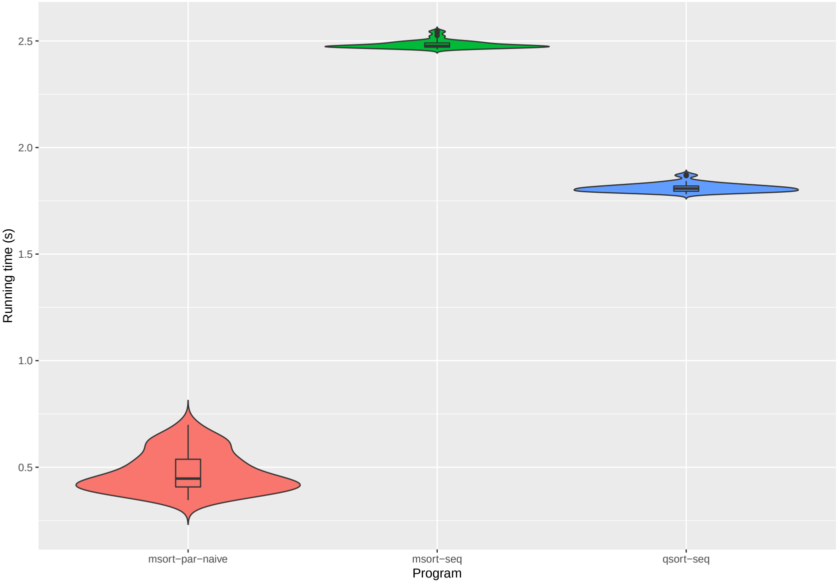
$$W(n) = O(n \log n), \qquad S(n) = O(n).$$

Observations

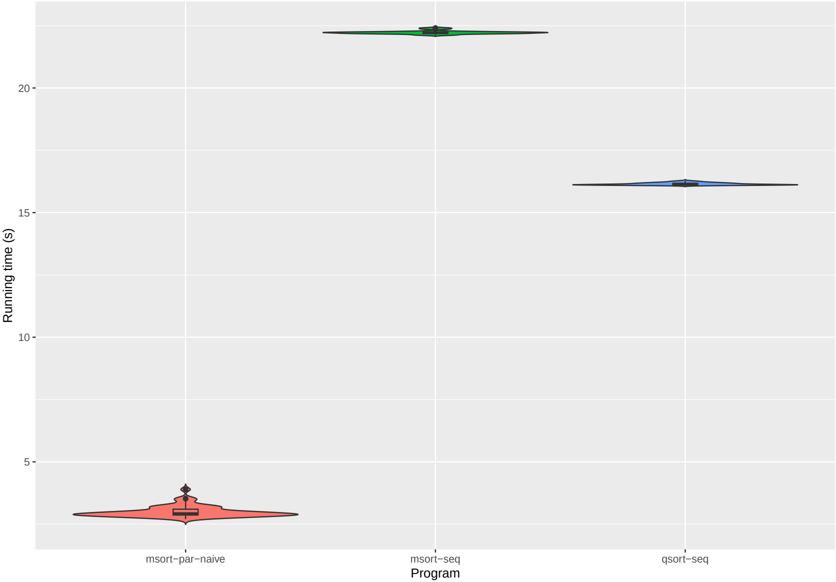
- We perform no more work than sequential sorts (*work-efficiency*).
- The algorithm contains $W/S = O(\log n)$ parallelism, which is small.

Ok, we have an asymptotic analysis. What about empirical performance?

Merge Sort in Cilk: Results on 16 MB Arrays



Merge Sort in Cilk: Results on 512 MB Arrays



Observations

Data size	qsort	msort	msort-par	Speedup	Self-speedup
16 MB	1.81 s	2.48 s	0.45 s	4x	5.5x
512 MB	16.12 s	22.21 s	2.93 s	5.5x	7.6x

Table: Synthetic Results

- Not awful, but disappointing on a 40-cores processor.
- How can we improve our algorithm and its implementation?
  - Coarsen the implementation to amortize bookkeeping costs.
  - Reduce the span of the algorithm to expose more parallelism.

An Important, Heuristic Technique: Coarsening

- In practice, it is detrimental to performance to spawn very small tasks.
- It is better to call optimized sequential code for small input sizes.
- This does not change the asymptotics of work and span.

An Important, Heuristic Technique: Coarsening

- In practice, it is detrimental to performance to spawn very small tasks.
- It is better to call optimized sequential code for small input sizes.
- This does not change the asymptotics of work and span.

Divide-and-conquer algorithms are easy to coarsen by changing base cases.

```
if (range_size(lo, hi) <= MSORT_CUTOFF) {
  quicksort_seq(B, lo, hi);
  return;
}
```

An Important, Heuristic Technique: Coarsening

- In practice, it is detrimental to performance to spawn very small tasks.
- It is better to call optimized sequential code for small input sizes.
- This does not change the asymptotics of work and span.

Divide-and-conquer algorithms are easy to coarsen by changing base cases.

```
if (range_size(lo, hi) <= MSORT_CUTOFF) {
  quicksort_seq(B, lo, hi);
  return;
}
```

Caveats

- The parameter `MSORT_CUTOFF` has been fixed experimentally to 4096.
- This is not portable: depends on the system, or even on inputs.
- One can try computing it online [Acar et al., 2016a, for ex.].

Improving Parallelism (1/3): Idea

- How to reduce the span of our merge sort? By parallelizing the merge.
- Divide-and-conquer algorithms are easy to program in Cilk. How could we express the merge in such a recursive fashion?

Improving Parallelism (1/3): Idea

- How to reduce the span of our merge sort? By parallelizing the merge.
- Divide-and-conquer algorithms are easy to program in Cilk. How could we express the merge in such a recursive fashion?

Recursive Merge

To merge two sorted arrays A and B of size $n_A \geq n_B$:

1. split A in two, and find the value a of its midpoint i_a,
2. find the position i_b of the smallest value of B larger than a,
3. recursively merge $A[0..i_a)$ with $B[0..i_b)$ and $A[i_a..n_a)$ with $B[i_b..n_b)$.

Improving Parallelism (1/3): Idea

- How to reduce the span of our merge sort? By parallelizing the merge.
- Divide-and-conquer algorithms are easy to program in Cilk. How could we express the merge in such a recursive fashion?

Recursive Merge

To merge two sorted arrays A and B of size $n_A \geq n_B$:

1. split A in two, and find the value a of its midpoint i_a,
2. find the position i_b of the smallest value of B larger than a,
3. recursively merge $A[0..i_a)$ with $B[0..i_b)$ and $A[i_a..n_a)$ with $B[i_b..n_b)$.

There are two key optimizations:

- perform step 2 using binary search,
- coarsen the algorithm to merge sequentially for small enough arrays.

Improving Parallelism (2/3): Implementation

```
void merge_par(int *C,
               const int *A, size_t A_lo, size_t A_hi,
               const int *B, size_t B_lo, size_t B_hi) {
  assert (C);
  assert (A);
  assert (B);

  if (range_size(A_lo, A_hi) < range_size(B_lo, B_hi)) {
    merge_par(C, B, B_lo, B_hi, A, A_lo, A_hi);
    return;
  }

  if (range_size(A_lo, A_hi) <= 1
      || range_size(A_lo, A_hi) + range_size(B_lo, B_hi) <= MERGE_CUTOFF) {
    merge_seq(C, A, A_lo, A_hi, B, B_lo, B_hi);
    return;
  }

  size_t A_mid = midpoint(A_lo, A_hi);
  size_t B_mid = search_sorted(B, B_lo, B_hi, A[A_mid]);
  cilk_spawn merge_par(C, A, A_lo, A_mid, B, B_lo, B_mid);
  merge_par(C + range_size(A_lo, A_mid) + range_size(B_lo, B_mid),
            A, A_mid, A_hi, B, B_mid, B_hi);
  cilk_sync;
}
```

Improving Parallelism (3/3): Analysis

Preliminary Remarks

- Write n for the problem size, i.e., $n_A + n_B$. Assume $n_a \geq n_b$ w.l.o.g.

Improving Parallelism (3/3): Analysis

Preliminary Remarks

- Write n for the problem size, i.e., $n_A + n_B$. Assume $n_a \geq n_b$ w.l.o.g.
- The recursive calls have size $n' \triangleq n_A/2 + i_B$ and $n'' \triangleq n_A/2 + n_B - i_B$.

Improving Parallelism (3/3): Analysis

Preliminary Remarks

- Write n for the problem size, i.e., $n_A + n_B$. Assume $n_a \geq n_b$ w.l.o.g.
- The recursive calls have size $n' \triangleq n_A/2 + i_B$ and $n'' \triangleq n_A/2 + n_B - i_B$.
- In the worst case, $\max(n', n'') = n_A/2 + n_B \leq 3n/4$.

Improving Parallelism (3/3): Analysis

Preliminary Remarks

- Write n for the problem size, i.e., $n_A + n_B$. Assume $n_a \geq n_b$ w.l.o.g.
- The recursive calls have size $n' \triangleq n_A/2 + i_B$ and $n'' \triangleq n_A/2 + n_B - i_B$.
- In the worst case, $\max(n', n'') = n_A/2 + n_B \leq 3n/4$.

Writing S_{m} and W_{m} for the span and work of merge, we have:

$$S_{\mathrm{m}}(n) = S_{\mathrm{m}}(3n/4) + O(\log n),$$
$$W_{\mathrm{m}}(n) = W_{\mathrm{m}}(\alpha n) + W_{\mathrm{m}}((1-\alpha)n) + O(\log n) \text{ where } 1/4 \leq \alpha \leq 3/4.$$

Improving Parallelism (3/3): Analysis

Preliminary Remarks

- Write n for the problem size, i.e., $n_A + n_B$. Assume $n_a \geq n_b$ w.l.o.g.
- The recursive calls have size $n' \triangleq n_A/2 + i_B$ and $n'' \triangleq n_A/2 + n_B - i_B$.
- In the worst case, $\max(n', n'') = n_A/2 + n_B \leq 3n/4$.

Writing S_m and W_m for the span and work of merge, we have:

$$S_\mathrm{m}(n) = S_\mathrm{m}(3n/4) + O(\log n),$$
$$W_\mathrm{m}(n) = W_\mathrm{m}(\alpha n) + W_\mathrm{m}((1-\alpha)n) + O(\log n) \text{ where } 1/4 \leq \alpha \leq 3/4.$$

Solving them leads to $S_\mathrm{m}(n) = O(\log^2(n))$ and $W_\mathrm{m}(n) = O(n)$.

Improving Parallelism (3/3): Analysis

Preliminary Remarks

- Write n for the problem size, i.e., $n_A + n_B$. Assume $n_a \geq n_b$ w.l.o.g.
- The recursive calls have size $n' \triangleq n_A/2 + i_B$ and $n'' \triangleq n_A/2 + n_B - i_B$.
- In the worst case, $\max(n', n'') = n_A/2 + n_B \leq 3n/4$.

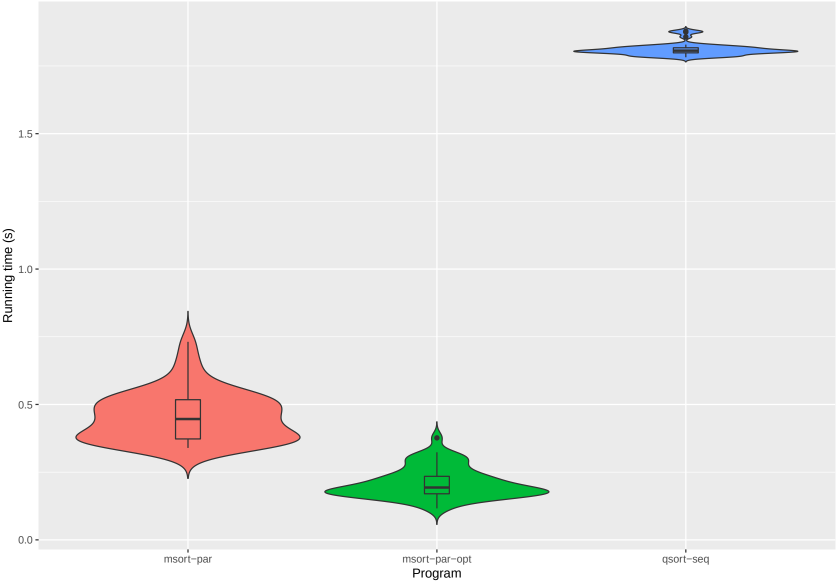
Writing S_{m} and W_{m} for the span and work of merge, we have:

$$S_{\mathrm{m}}(n) = S_{\mathrm{m}}(3n/4) + O(\log n),$$
$$W_{\mathrm{m}}(n) = W_{\mathrm{m}}(\alpha n) + W_{\mathrm{m}}((1-\alpha)n) + O(\log n) \text{ where } 1/4 \leq \alpha \leq 3/4.$$

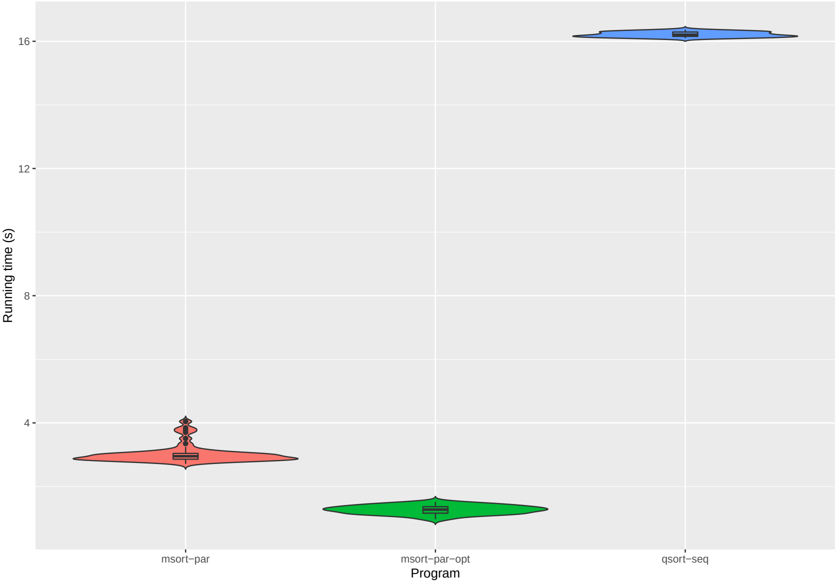
Solving them leads to $S_{\mathrm{m}}(n) = O(\log^2(n))$ and $W_{\mathrm{m}}(n) = O(n)$.

Our merge sort now has $S(n) = O(\log^3 n)$, hence $n/\log^2(n)$ parallelism!

Optimized Merge Sort: Results on 16 MB Arrays



Optimized Merge Sort: Results on 512 MB Arrays



Merge Sort: Conclusion

Data size	qsort	msort-par	msort-par-opt	Spd. (par)	Spd. (opt)
16 MB	1.81 s	0.45 s	0.19 s	4x	9.5x
512 MB	16.12 s	2.93 s	1.27 s	5.5x	12.7x

Table: Synthetic Results

- Pleasingly better than the initial version.
- Still naive, in no way a very fast parallel sort.
- Good enough for our first Cilk program!

Interlude: How General is Internal Determinism?

The Parallel Paradise of Divide-and-Conquer Algorithms

- Merge sort was easy to parallelize.
- In particular, its serial elision is the sequential algorithm!
- Work-efficiency and internal determinism are immediate consequences.

Interlude: How General is Internal Determinism?

The Parallel Paradise of Divide-and-Conquer Algorithms

- Merge sort was easy to parallelize.
- In particular, its serial elision is the sequential algorithm!
- Work-efficiency and internal determinism are immediate consequences.

This works well, but isn't it a bit disappointing? Are there examples were coming out with an internally-deterministic algorithm is harder?

Interlude: How General is Internal Determinism?

The Parallel Paradise of Divide-and-Conquer Algorithms

- Merge sort was easy to parallelize.
- In particular, its serial elision is the sequential algorithm!
- Work-efficiency and internal determinism are immediate consequences.

This works well, but isn't it a bit disappointing? Are there examples were coming out with an internally-deterministic algorithm is harder?

Yes! Let's finish with a more complex example.

Fisher-Yates Shuffling

- Simple process for shuffling arrays/generating uniform permutations.
- Rediscovered multiple times since the 1930s: Durstenfeld, Knuth.

Fisher-Yates Shuffling

- Simple process for shuffling arrays/generating uniform permutations.
- Rediscovered multiple times since the 1930s: Durstenfeld, Knuth.

```
void fy_shuffle_seq(size_t n, int *A) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[rand() % (i + 1)], A[i]);
}
```

Fisher-Yates Shuffling

- Simple process for shuffling arrays/generating uniform permutations.
- Rediscovered multiple times since the 1930s: Durstenfeld, Knuth.

```
void fy_shuffle_seq(size_t n, int *A) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[rand() % (i + 1)], A[i]);
}
```

Pulling out random number generation, we get:

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

Parallelizing Fisher-Yates?

Desiderata

Implement parallel shuffling code that:

- is internally deterministic,
- given H, generates the exact same permutation as Fisher-Yates.

Parallelizing Fisher-Yates?

Desiderata

Implement parallel shuffling code that:

- is internally deterministic,
- given H, generates the exact same permutation as Fisher-Yates.

This problem was studied by Shun et al. [2014]. Why is it interesting?

- At first glance, the code does not seem to contain a lot of parallelism.
- Yet, we'll see that it fits into a general framework for parallelizing certain sequential algorithms, proposed by Blelloch et al. [2012].

Parallelizing Fisher-Yates?

Desiderata

Implement parallel shuffling code that:

- is internally deterministic,
- given H, generates the exact same permutation as Fisher-Yates.

This problem was studied by Shun et al. [2014]. Why is it interesting?

- At first glance, the code does not seem to contain a lot of parallelism.
- Yet, we'll see that it fits into a general framework for parallelizing certain sequential algorithms, proposed by Blelloch et al. [2012].

So, first, how sequential is this code really?

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	a	b	c	d	e	f	g	h

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	a	h	c	d	e	f	g	b

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	a	h	c	g	e	f	d	b

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	a	h	f	g	e	c	d	b

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	a	e	f	g	h	c	d	b

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	a	e	f	g	h	c	d	b

Dependence Structure of Fisher-Yates

```c
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	a	f	e	g	h	c	d	b

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	f	a	e	g	h	c	d	b

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	f	a	e	g	h	c	d	b

Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	f	a	e	g	h	c	d	b

Dependence forest (tree) induced by H:

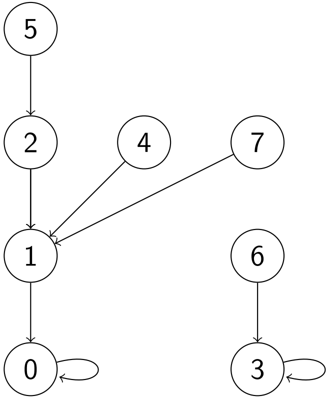
Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	f	a	e	g	h	c	d	b

Dependence forest (tree) induced by H:

1. start with edges $i \to H[i]$,



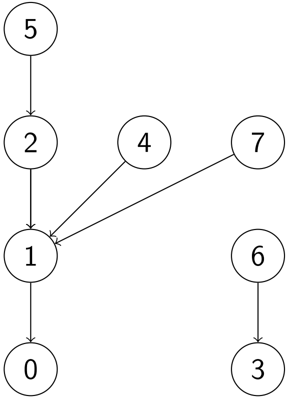
Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	f	a	e	g	h	c	d	b

Dependence forest (tree) induced by H:

1. start with edges $i \to H[i]$,



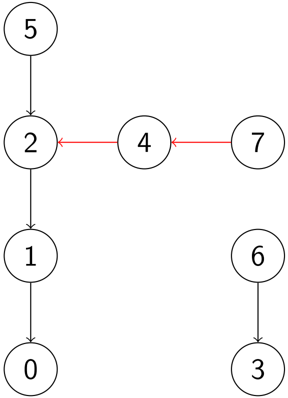
Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	f	a	e	g	h	c	d	b

Dependence forest (tree) induced by H:

1. start with edges $i \rightarrow H[i]$,
2. chain immediate predecessors,



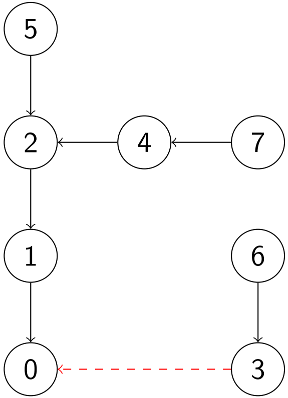
Dependence Structure of Fisher-Yates

```c
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	f	a	e	g	h	c	d	b

Dependence forest (tree) induced by H:

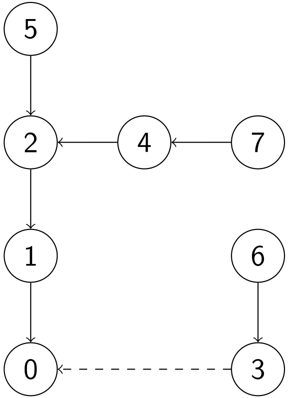
1. start with edges $i \to H[i]$,
2. chain immediate predecessors,
3. (chain roots.)



Dependence Structure of Fisher-Yates

```
void fy_shuffle_seq(size_t n, int *A, const int *H) {
  for (size_t i = n - 1; i > 0; i--)
    swap(A[H[i]], A[i]);
}
```

i	0	1	2	3	4	5	6	7
$H[i]$	0	0	1	3	1	2	3	1
$A[i]$	f	a	e	g	h	c	d	b



Dependence forest (tree) induced by H:

1. start with edges $i \to H[i]$,
2. chain immediate predecessors,
3. (chain roots.)

Theorem (Shun et al. [2014])

Shaped like a random binary search tree, hence height of $\Theta(\log n)$ w.h.p.

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Deterministic Reservations

- A general framework for parallelizing iterative algorithms.

```
speculative_for (reserve, commit, lo, hi);
```

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Deterministic Reservations

- A general framework for parallelizing iterative algorithms.

```
speculative_for (reserve, commit, lo, hi);
```

- Processes every iteration in $[lo, hi)$, in a round-per-round fashion.

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Deterministic Reservations

- A general framework for parallelizing iterative algorithms.

```
speculative_for (reserve, commit, lo, hi);
```

- Processes every iteration in $[lo, hi)$, in a round-per-round fashion.
- Every round proceeds as follows:

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Deterministic Reservations

- A general framework for parallelizing iterative algorithms.

```
speculative_for (reserve, commit, lo, hi);
```

- Processes every iteration in $[lo, hi)$, in a round-per-round fashion.
- Every round proceeds as follows:
  1. pick a subset of remaining iterations,

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Deterministic Reservations

- A general framework for parallelizing iterative algorithms.

```
speculative_for (reserve, commit, lo, hi);
```

- Processes every iteration in $[lo, hi)$, in a round-per-round fashion.
- Every round proceeds as follows:
  1. pick a subset of remaining iterations,
  2. run `reserve` in parallel on every iteration of the chosen subset,

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Deterministic Reservations

- A general framework for parallelizing iterative algorithms.

```
speculative_for (reserve, commit, lo, hi);
```

- Processes every iteration in $[lo, hi)$, in a round-per-round fashion.
- Every round proceeds as follows:
  1. pick a subset of remaining iterations,
  2. run `reserve` in parallel on every iteration of the chosen subset,
  3. run `commit` in parallel on every iteration of the chosen subset,

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Deterministic Reservations

- A general framework for parallelizing iterative algorithms.

```
speculative_for (reserve, commit, lo, hi);
```

- Processes every iteration in $[lo, hi)$, in a round-per-round fashion.
- Every round proceeds as follows:
  1. pick a subset of remaining iterations,
  2. run `reserve` in parallel on every iteration of the chosen subset,
  3. run `commit` in parallel on every iteration of the chosen subset,
  4. mark every iteration where `commit` succeeded as done.

Parallel Fisher-Yates via Deterministic Reservations (1/2)

- In average, the code does contain parallelism, after all. Great!
- Yet, extracting this parallelism does not seem easy.
- (Wrong idea: generate the dependence forest and traverse it.)
- Following Blelloch et al. [2012], we'll use deterministic reservations.

Deterministic Reservations

- A general framework for parallelizing iterative algorithms.

```
speculative_for (reserve, commit, lo, hi);
```

- Processes every iteration in $[lo, hi)$, in a round-per-round fashion.
- Every round proceeds as follows:
  1. pick a subset of remaining iterations,
  2. run `reserve` in parallel on every iteration of the chosen subset,
  3. run `commit` in parallel on every iteration of the chosen subset,
  4. mark every iteration where `commit` succeeded as done.
- This is internally deterministic if the choice of subset is deterministic.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	0	0	0	0	0	0	0
A[i]	a	b	c	d	e	f	g	h

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume R is a boolean array of the same size as A and H.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	0	0	0	0	0	0	0
A[i]	a	b	c	d	e	f	g	h

Round: 1.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	7	5	6	4	5	6	7
A[i]	a	b	c	d	e	f	g	h

Round: 1.
Phase: reserve.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume R is a boolean array of the same size as A and H.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	7	5	6	4	5	6	7
A[i]	a	h	f	g	e	c	d	b

Round: 1.
Phase: commit.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	0	0	0	0	0	0	0
A[i]	a	h	f	g	e	c	d	b

Round: 2.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	4	2	3	4	0	0	0
A[i]	a	h	f	g	e	c	d	b

Round: 2.
Phase: reserve.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume R is a boolean array of the same size as A and H.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	4	2	3	4	0	0	0
A[i]	a	e	f	g	h	c	d	b

Round: 2.
Phase: commit.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume R is a boolean array of the same size as A and H.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	0	0	0	0	0	0	0
A[i]	a	e	f	g	h	c	d	b

Round: 3.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	1	2	2	0	0	0	0	0
A[i]	a	e	f	g	h	c	d	b

Round: 3.

Phase: reserve.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume R is a boolean array of the same size as A and H.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	1	2	2	0	0	0	0	0
A[i]	a	f	e	g	h	c	d	b

Round: 3.
Phase: commit.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	0	0	0	0	0	0	0
A[i]	a	f	e	g	h	c	d	b

Round: 4.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	1	1	0	0	0	0	0	0
A[i]	a	f	e	g	h	c	d	b

Round: 4.

Phase: reserve.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	1	1	0	0	0	0	0	0
A[i]	f	a	e	g	h	c	d	b

Round: 4.

Phase: commit.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume R is a boolean array of the same size as A and H.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	0	0	0	0	0	0	0
A[i]	f	a	e	g	h	c	d	b

Round: 5.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume R is a boolean array of the same size as A and H.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	0	0	0	0	0	0	0
A[i]	f	a	e	g	h	c	d	b

Round: 5.

Phase: reserve.

Parallel Fisher-Yates via Deterministic Reservations (2/2)

(Assume `R` is a boolean array of the same size as `A` and `H`.)

```
bool fy_reserve(int i) {
  write_max(&R[i], i); write_max(&R[H[i]], i);
}
bool fy_commit(int i) {
  if (R[H[i]] == i && R[i] == i) {
    swap(A[H[i]], A[i]);
    return 1;
  }
  return 0;
}
```

i	0	1	2	3	4	5	6	7
H[i]	0	0	1	3	1	2	3	1
R[i]	0	0	0	0	0	0	0	0
A[i]	f	a	e	g	h	c	d	b

Round: 5.
Phase: commit.

Implementing Deterministic Reservations (1/3)

What's missing from our informal implementation of parallel Fisher-Yates?

- Implement `write_max()`.
- Implement `speculative_for()`.

```
static inline int res_write_max_sc(reserve_t *ptr, int val) {
  assert (val >= 0);

  int current = atomic_load(ptr);
  while (current < val &&
         !atomic_compare_exchange_strong(ptr, &current, val)) {
    asm volatile ("pause");
  }

  assert (atomic_load(ptr) >= val);
  return max(current, val);
}
```

Implementing Deterministic Reservations (3/3)

- The code for `fy_shuffle_par` is a bit too long to display on a slide.

Implementing Deterministic Reservations (3/3)

- The code for `fy_shuffle_par` is a bit too long to display on a slide.
- Key issue: which iterations should be retried next round?

Implementing Deterministic Reservations (3/3)

- The code for `fy_shuffle_par` is a bit too long to display on a slide.
- Key issue: which iterations should be retried next round?
- We maintain an array of boolean flags to record failed iterations.

Implementing Deterministic Reservations (3/3)

- The code for `fy_shuffle_par` is a bit too long to display on a slide.
- Key issue: which iterations should be retried next round?
- We maintain an array of boolean flags to record failed iterations.
- At the end of each round, we repack failed iterations.

Implementing Deterministic Reservations (3/3)

- The code for `fy_shuffle_par` is a bit too long to display on a slide.
- Key issue: which iterations should be retried next round?
- We maintain an array of boolean flags to record failed iterations.
- At the end of each round, we repack failed iterations.

This repacking is a key operation in deterministic reservations.

```c
/* Given two arrays of integers dst and src, and an array of booleans keep, all
   of size n, pack_par(dst, src, keep, n) copies into dst all the elements
   src[i] such that keep[i] is true, in order. It returns the number of copied
   elements. */
size_t pack_par(int *dst, const int *src, const bool *keep, size_t n);
```

Implementing Deterministic Reservations (3/3)

- The code for `fy_shuffle_par` is a bit too long to display on a slide.
- Key issue: which iterations should be retried next round?
- We maintain an array of boolean flags to record failed iterations.
- At the end of each round, we repack failed iterations.

This repacking is a key operation in deterministic reservations.

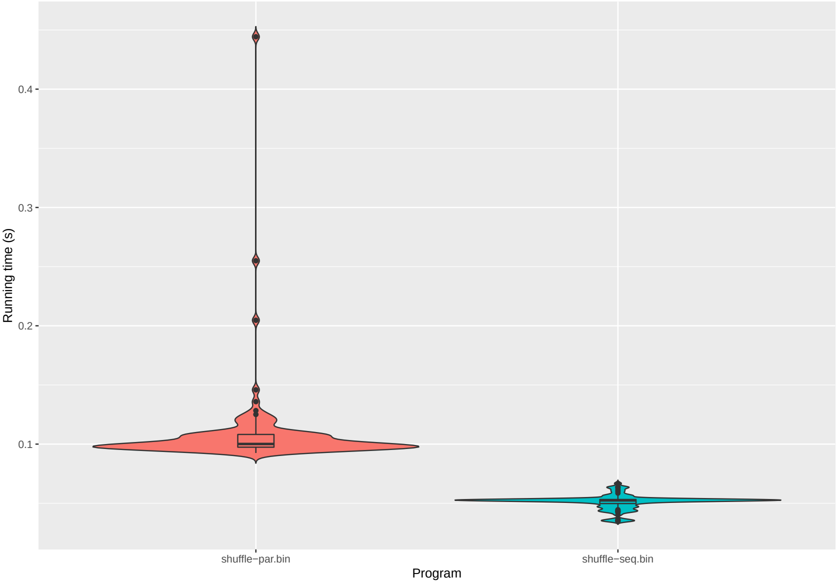
```
/* Given two arrays of integers dst and src, and an array of booleans keep, all
   of size n, pack_par(dst, src, keep, n) copies into dst all the elements
   src[i] such that keep[i] is true, in order. It returns the number of copied
   elements. */
size_t pack_par(int *dst, const int *src, const bool *keep, size_t n);
```

We can implement it efficiently on top of an exclusive prefix sum.

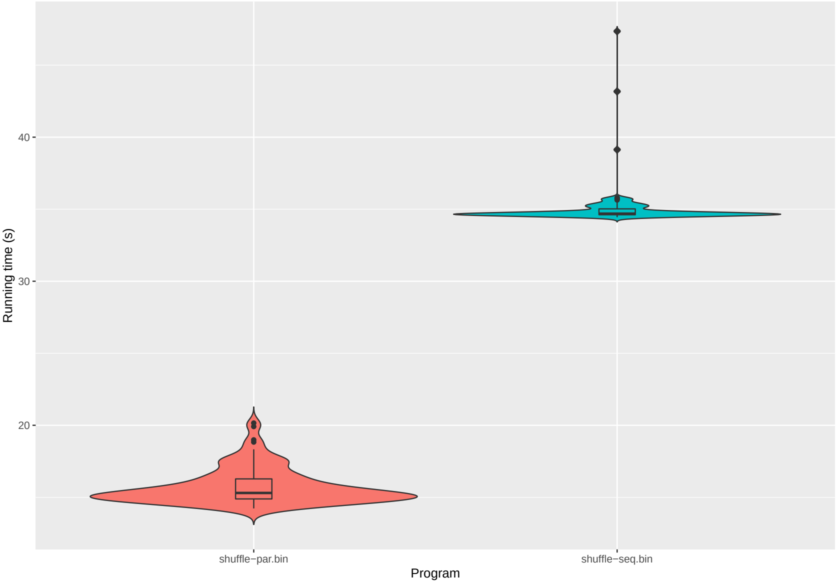
src	a	b	c	d	e	f	g
keep	1	1	0	1	0	0	1
dst	a	b	d	g			
prefix sum of keep	0	1	2	2	3	3	3

Look at `array.c` for a simple parallel implementation.

Fisher-Yates: Results on 16 MB Arrays



Fisher-Yates: Results on 4 GB Arrays



Fisher-Yates: Conclusion

Data size	seq	par	Speedup
16 MB	0.05 s	0.10 s	0.2 x
4 GB	34.69 s	15.30 s	2.26 x

Table: Synthetic Results

- There is extractable parallelism in Fisher-Yates, for large arrays.
- The original paper announced higher numbers...
  - Is my implementation bad?
  - We may need to experiment on even larger arrays.

Fisher-Yates: Conclusion

Data size	seq	par	Speedup
16 MB	0.05 s	0.10 s	0.2 x
4 GB	34.69 s	15.30 s	2.26 x

Table: Synthetic Results

- There is extractable parallelism in Fisher-Yates, for large arrays.
- The original paper announced higher numbers...
  - Is my implementation bad?
  - We may need to experiment on even larger arrays.

Exercise: improve my implementation of parallel Fisher-Yates.

Conclusion of Lecture 1

We've written our first internally-deterministic parallel programs in Cilk.

- Good parallel algorithms are work-efficient, low-span.
- Their performance can be analyzed using classical tools.
- Divide-and-conquer algorithms are easy to parallelize.
  - But you generally need to parallelize the conquer phase!
- Even algorithms that look sequential can contain parallelism.
  - It can often be extracted using ideas like deterministic reservations.

Lecture 2

Implementing Task Parallelism on Multicore Hardware

Previously on MPRI 2.37.1

Last week, we've discussed several ideas:

- The work/span model for performance analysis.
- Internally-deterministic, task-parallel algorithms.
- Their implementation in the Cilk extension to C and C++.

Previously on MPRI 2.37.1

Last week, we've discussed several ideas:

- The work/span model for performance analysis.
- Internally-deterministic, task-parallel algorithms.
- Their implementation in the Cilk extension to C and C++.

Why Task Parallelism?

- Convenient, high-level parallel programming model.
- Useful performance model close to that of sequential algorithms.
- Well-understood implementations efficient in theory and in practice.

Previously on MPRI 2.37.1

Last week, we've discussed several ideas:

- The work/span model for performance analysis.
- Internally-deterministic, task-parallel algorithms.
- Their implementation in the Cilk extension to C and C++.

Why Task Parallelism?

- Convenient, high-level parallel programming model.
- Useful performance model close to that of sequential algorithms.
- Well-understood implementations efficient in theory and in practice.

And what about Cilk itself?

- Cilk is just a vehicle, our programs could be written using many other technologies: OpenMP, `java.util.concurrent`, Rayon (Rust)...
- The implementation techniques invented for Cilk are now standard.

Previously on MPRI 2.37.1

Last week, we've discussed several ideas:

- The work/span model for performance analysis.
- Internally-deterministic, task-parallel algorithms.
- Their implementation in the Cilk extension to C and C++.

Why Task Parallelism?

- Convenient, high-level parallel programming model.
- Useful performance model close to that of sequential algorithms.
- Well-understood implementations efficient in theory and in practice.

And what about Cilk itself?

- Cilk is just a vehicle, our programs could be written using many other technologies: OpenMP, `java.util.concurrent`, Rayon (Rust)...
- The implementation techniques invented for Cilk are now standard.

Today: the standard implementation of Cilk and its formal properties.

The Simplest Bad Cilk Program

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

The Implementations We Know

The Simplest Bad Cilk Program

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

The Implementations We Know

- Its serial elision, obtained by erasing `cilk_spawn` and `cilk_sync`.
    - Not a *provably-efficient* implementation.

The Simplest Bad Cilk Program

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

The Implementations We Know

- Its serial elision, obtained by erasing `cilk_spawn` and `cilk_sync`.
  - Not a *provably-efficient* implementation.
- Its direct implementation using one pthread per task.
  - Slow or even unworkable in practice (cf. last lecture).

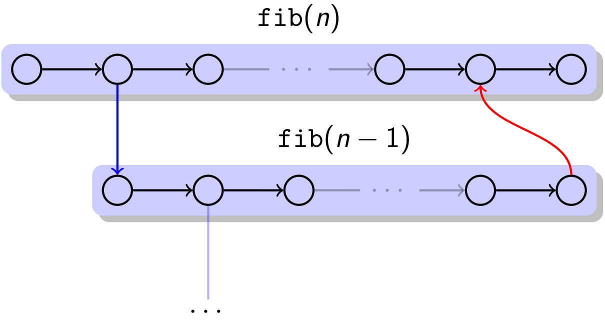
The Simplest Bad Cilk Program

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

The Implementations We Know

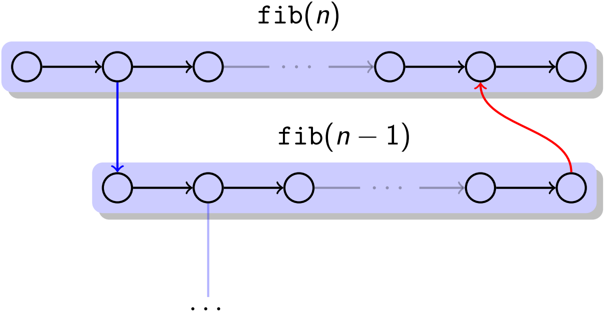
- Its serial elision, obtained by erasing `cilk_spawn` and `cilk_sync`.
  - Not a *provably-efficient* implementation.
- Its direct implementation using one pthread per task.
  - Slow or even unworkable in practice (cf. last lecture).
- The scheduler seen in the proof of Brent's theorem
  - Slow in practice because of its centralized structure.

Computation Graph of Fib



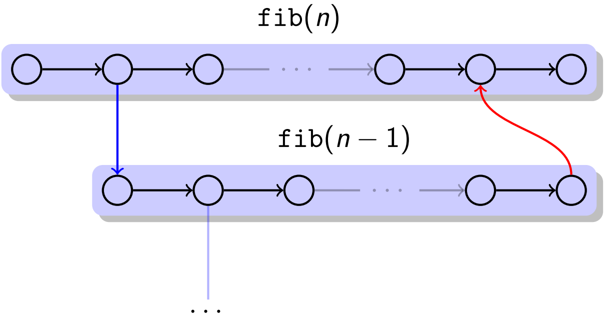
- Three kinds of edges: *seq* (**black**), *spawn* (**blue**), and *join* (**red**).

Computation Graph of Fib



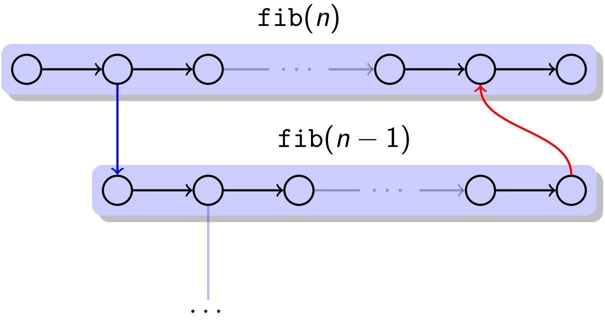
- Three kinds of edges: *seq* (**black**), *spawn* (**blue**), and *join* (**red**).
- At most one incoming/outgoing seq/spawn edge per node.

Computation Graph of Fib



- Three kinds of edges: *seq* (**black**), *spawn* (**blue**), and *join* (**red**).
- At most one incoming/outgoing seq/spawn edge per node.
- Tasks are maximal seq-chains, represented as light-blue boxes.

Computation Graph of Fib



- Three kinds of edges: *seq* (**black**), *spawn* (**blue**), and *join* (**red**).
- At most one incoming/outgoing seq/spawn edge per node.
- Tasks are maximal seq-chains, represented as light-blue boxes.
- Together, tasks and join edges form the program's *activation tree*.

Formal Interlude: Schedules

Formal Interlude: Schedules

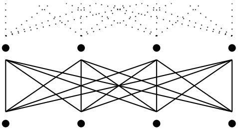
- A (finite) computation graph gives rise to a (finite) poset **C**.

Formal Interlude: Schedules

- A (finite) computation graph gives rise to a (finite) poset **C**.
- We abstract our p-core machine as the locally finite poset **P**, where

$$|\mathbf{P}| = [p] \times \mathbb{N}^+, \qquad (i, n) <_{\mathbf{P}} (j, m) \Leftrightarrow n < m.$$

For example, the two lowest levels of **4** look like:

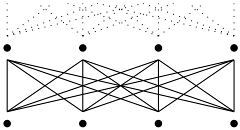


Formal Interlude: Schedules

- A (finite) computation graph gives rise to a (finite) poset **C**.
- We abstract our p-core machine as the locally finite poset **P**, where

$$|\mathbf{P}| = [p] \times \mathbb{N}^+, \qquad (i, n) <_{\mathbf{P}} (j, m) \Leftrightarrow n < m.$$

For example, the two lowest levels of **4** look like:



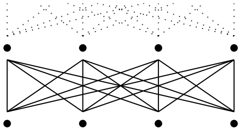
- We denote $T : \mathbf{P} \to \omega$ the map $T(x) \triangleq 1 + \max\{ T(y) \mid y < x \}$.

Formal Interlude: Schedules

- A (finite) computation graph gives rise to a (finite) poset **C**.
- We abstract our p-core machine as the locally finite poset **P**, where

$$|\mathbf{P}| = [p] \times \mathbb{N}^+, \qquad (i, n) <_{\mathbf{P}} (j, m) \Leftrightarrow n < m.$$

For example, the two lowest levels of **4** look like:



- We denote $T : \mathbf{P} \to \omega$ the map $T(x) \triangleq 1 + \max\{ T(y) \mid y < x\}$.

Definitions

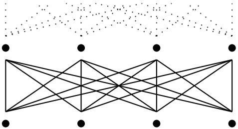
- A *schedule* $\mathcal{X}$ is a monotonic injective function $\mathbf{C} \to \mathbf{P}$.

Formal Interlude: Schedules

- A (finite) computation graph gives rise to a (finite) poset **C**.
- We abstract our p-core machine as the locally finite poset **P**, where

$$|\mathbf{P}| = [p] \times \mathbb{N}^+, \qquad (i, n) <_{\mathbf{P}} (j, m) \Leftrightarrow n < m.$$

For example, the two lowest levels of **4** look like:



- We denote $T : \mathbf{P} \to \omega$ the map $T(x) \triangleq 1 + \max\{\, T(y) \mid y < x \}$.

Definitions

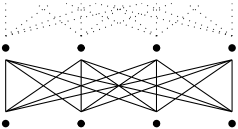
- A *schedule* $\mathcal{X}$ is a monotonic injective function $\mathbf{C} \to \mathbf{P}$.

Formal Interlude: Schedules

- A (finite) computation graph gives rise to a (finite) poset **C**.
- We abstract our p-core machine as the locally finite poset **P**, where

$$|\mathbf{P}| = [p] \times \mathbb{N}^{+}, \qquad (i, n) <_{\mathbf{P}} (j, m) \Leftrightarrow n < m.$$

For example, the two lowest levels of **4** look like:



- We denote $T : \mathbf{P} \to \omega$ the map $T(x) \triangleq 1 + \max\{ T(y) \mid y < x\}$.

Definitions

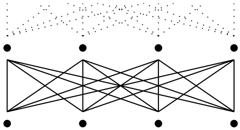
- A *schedule* $\mathcal{X}$ is a monotonic injective function $\mathbf{C} \to \mathbf{P}$.

Formal Interlude: Schedules

- A (finite) computation graph gives rise to a (finite) poset **C**.
- We abstract our p-core machine as the locally finite poset **P**, where

$$|\mathbf{P}| = [p] \times \mathbb{N}^+, \qquad (i, n) <_{\mathbf{P}} (j, m) \Leftrightarrow n < m.$$

For example, the two lowest levels of **4** look like:



- We denote $T : \mathbf{P} \to \omega$ the map $T(x) \triangleq 1 + \max\{\, T(y) \mid y < x \}$.

Definitions

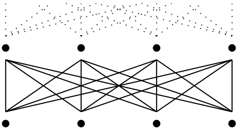
- A *schedule* $\mathcal{X}$ is a monotonic injective function $\mathbf{C} \to \mathbf{P}$.
- The *scheduling time* of $a \in \mathbf{C}$ in $\mathcal{X}$ is defined as $T_{\mathcal{X}}(a) = T(\mathcal{X}(a))$.

Formal Interlude: Schedules

- A (finite) computation graph gives rise to a (finite) poset **C**.
- We abstract our p-core machine as the locally finite poset **P**, where

$$|\mathbf{P}| = [p] \times \mathbb{N}^+, \qquad (i, n) <_{\mathbf{P}} (j, m) \Leftrightarrow n < m.$$

For example, the two lowest levels of **4** look like:



- We denote $T : \mathbf{P} \to \omega$ the map $T(x) \triangleq 1 + \max\{ T(y) \mid y < x\}$.

Definitions

- A *schedule* $\mathcal{X}$ is a monotonic injective function $\mathbf{C} \to \mathbf{P}$.
- The *scheduling time* of $a \in \mathbf{C}$ in $\mathcal{X}$ is defined as $T_{\mathcal{X}}(a) = T(\mathcal{X}(a))$.
- The *length* of $\mathcal{X}$ is defined as $T(\mathcal{X}) \triangleq \max\{ T(x) \mid x \in \mathcal{X}(\mathbf{C})\}$.

Formal Interlude: Greedy Scheduling Theorem

Definitions

- The set of instructions *ready* in $\mathcal{X}$ at $t \in \omega$ is defined as

$$R_{\mathcal{X}}(t) \triangleq \{a \in \mathbf{C} \mid \forall b <_{\mathbf{C}} a, T_{\mathcal{X}}(b) < n\}.$$

- The set of processor steps *active* in $\mathcal{X}$ at $t \in \omega$ is defined as

$$A_{\mathcal{X}}(t) \triangleq \{i \in [p] \mid \exists x \in \mathbf{C}, \mathcal{X}(x) = (i, t)\}.$$

- The schedule $\mathcal{X}$ is *greedy* if $\#A_{\mathcal{X}}(t) = \min(p, \#R_{\mathcal{X}}(t))$.

Formal Interlude: Greedy Scheduling Theorem

Definitions

- The set of instructions *ready* in $\mathcal{X}$ at $t \in \omega$ is defined as

$$R_{\mathcal{X}}(t) \triangleq \{a \in \mathbf{C} \mid \forall b <_{\mathbf{C}} a, T_{\mathcal{X}}(b) < n\}.$$

- The set of processor steps *active* in $\mathcal{X}$ at $t \in \omega$ is defined as

$$A_{\mathcal{X}}(t) \triangleq \{i \in [p] \mid \exists x \in \mathbf{C}, \mathcal{X}(x) = (i, t)\}.$$

- The schedule $\mathcal{X}$ is *greedy* if $\#A_{\mathcal{X}}(t) = \min(p, \#R_{\mathcal{X}}(t))$.

Theorem (Blumofe and Leiserson [1998])

Any greedy schedule $\mathcal{X}$ achieves

$$T(\mathcal{X}) \leq \frac{W(\mathbf{C})}{p} + S(\mathbf{C}).$$

Formal Interlude: Greedy Scheduling Theorem (Proof)

Let $\mathbf{D}_t$ and $\mathbf{W}_t$ be the two subposets of $\mathbf{C}$ such that $\mathbf{C} \cong \mathbf{D}_t + \mathbf{W}_t$ and the elements of $\mathbf{D}_t$ are $\bigcup_{t' \leq t} A_{\mathcal{X}}(t')$. We prove, by induction on $t \leq T(\mathcal{X})$,

$$t \leq \frac{W(\mathbf{D}_t)}{p} + S(\mathbf{C}) - S(\mathbf{W}_t).$$

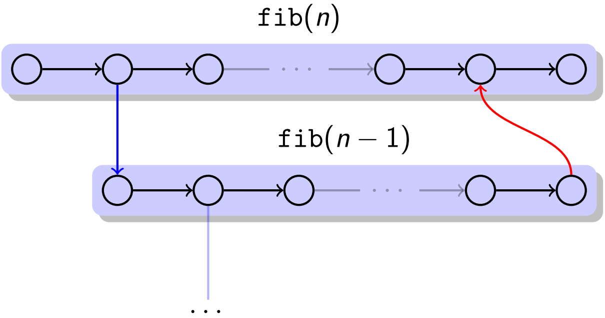
- If $\#A_{\mathcal{X}}(t+1) = p$, we have

$$\begin{aligned} t + 1 &\leq \frac{W(\mathbf{D}_t)}{p} + S(\mathbf{C}) - S(\mathbf{W}_t) + 1 && \text{(I.H.)} \\ &= \frac{W(\mathbf{D}_t)}{p} + S(\mathbf{C}) - S(\mathbf{W}_t) + \frac{\#A_{\mathcal{X}}(t+1)}{p} \\ &\leq \frac{W(\mathbf{D}_{t+1})}{p} + S(\mathbf{C}) - S(\mathbf{W}_{t+1}). \end{aligned}$$

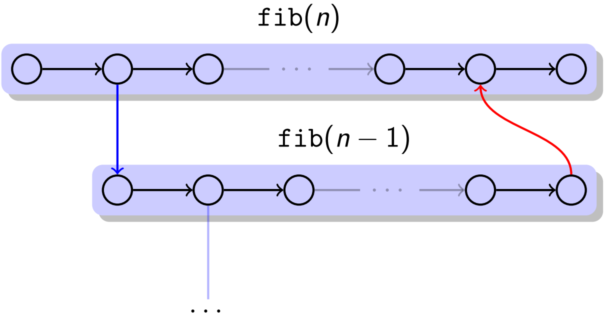
- If $\#A_{\mathcal{X}}(t+1) < p$, since $\mathcal{X}$ is greedy we have $\#R_{\mathcal{X}}(t+1) < p$. This entails $S(\mathbf{W}_t) = S(\mathbf{W}_{t+1}) + 1$ and, as a consequence,

$$t + 1 \leq \frac{W(\mathbf{D}_t)}{p} + S(\mathbf{C}) - S(\mathbf{W}_t) + 1 \leq \frac{W(\mathbf{D}_{t+1})}{p} + S(\mathbf{C}) - S(\mathbf{W}_{t+1}).$$

Strictness and Full Strictness



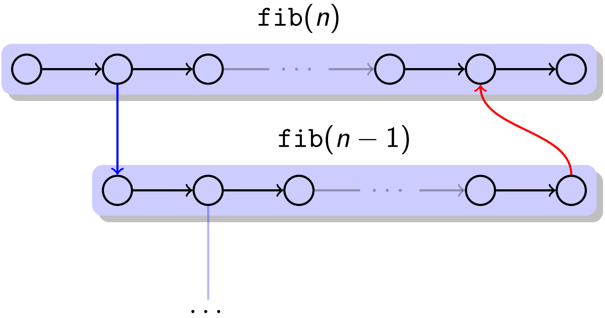
Strictness and Full Strictness



A Key Invariant

A program is *strict* (resp. *fully strict*) when its join edges always connect a task to one of its ancestors (resp. to its parent) in the activation tree.

Strictness and Full Strictness

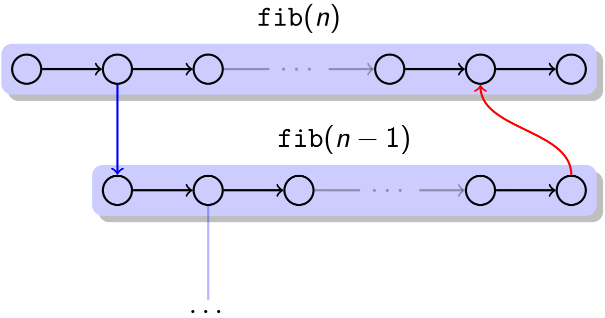


A Key Invariant

A program is *strict* (resp. *fully strict*) when its join edges always connect a task to one of its ancestors (resp. to its parent) in the activation tree.

- Can you write a Cilk program that is not fully strict?

Strictness and Full Strictness

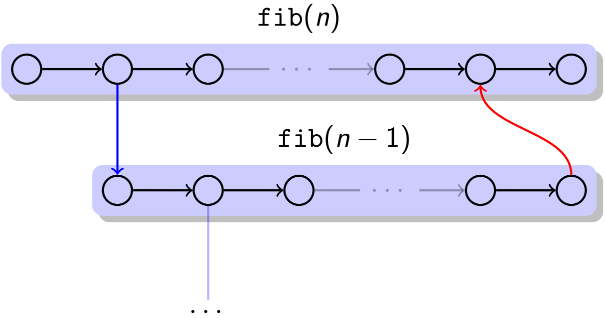


A Key Invariant

A program is *strict* (resp. *fully strict*) when its join edges always connect a task to one of its ancestors (resp. to its parent) in the activation tree.

- Can you write a Cilk program that is not fully strict? No.

Strictness and Full Strictness

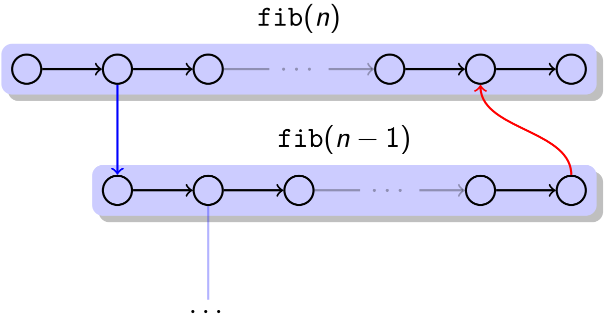


A Key Invariant

A program is *strict* (resp. *fully strict*) when its join edges always connect a task to one of its ancestors (resp. to its parent) in the activation tree.

- Can you write a Cilk program that is not fully strict? No.
- How would you write a non-strict program?

Strictness and Full Strictness

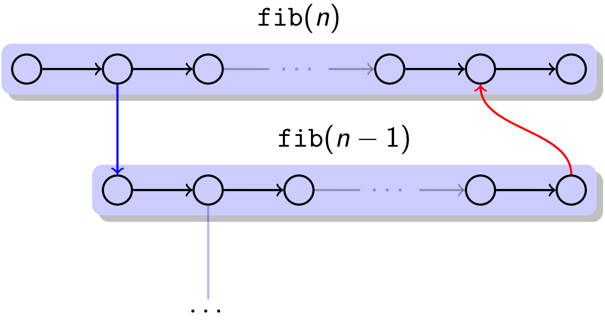


A Key Invariant

A program is *strict* (resp. *fully strict*) when its join edges always connect a task to one of its ancestors (resp. to its parent) in the activation tree.

- Can you write a Cilk program that is not fully strict? No.
- How would you write a non-strict program? Futures, raw pthreads...

Strictness and Full Strictness

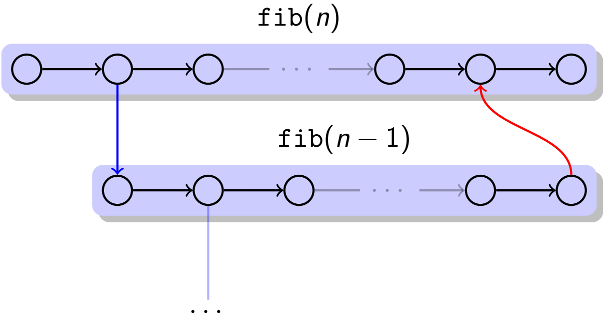


A Key Invariant

A program is *strict* (resp. *fully strict*) when its join edges always connect a task to one of its ancestors (resp. to its parent) in the activation tree.

- Can you write a Cilk program that is not fully strict? No.
- How would you write a non-strict program? Futures, raw pthreads...
- Why is strictness important?

Strictness and Full Strictness



A Key Invariant

A program is *strict* (resp. *fully strict*) when its join edges always connect a task to one of its ancestors (resp. to its parent) in the activation tree.

- Can you write a Cilk program that is not fully strict? No.
- How would you write a non-strict program? Futures, raw pthreads...
- Why is strictness important? It safeguards memory usage.

Thinking About Space Usage

Space Usage in the Task-Parallel Model

Thinking About Space Usage

Space Usage in the Task-Parallel Model

- We assume every task reserves a certain amount of memory.
  - It is natural to think of it as stack usage, but this is not essential.

Thinking About Space Usage

Space Usage in the Task-Parallel Model

- We assume every task reserves a certain amount of memory.
  - It is natural to think of it as stack usage, but this is not essential.
- We assume this memory cannot be freed before the task and all its children in the activation tree terminate.

Thinking About Space Usage

Space Usage in the Task-Parallel Model

- We assume every task reserves a certain amount of memory.
  - It is natural to think of it as stack usage, but this is not essential.
- We assume this memory cannot be freed before the task and all its children in the activation tree terminate.
- The schedule $\mathcal{X}$ thus determines space usage $M(\mathcal{X})$.

Thinking About Space Usage

Space Usage in the Task-Parallel Model

- We assume every task reserves a certain amount of memory.
  - It is natural to think of it as stack usage, but this is not essential.
- We assume this memory cannot be freed before the task and all its children in the activation tree terminate.
- The schedule $\mathcal{X}$ thus determines space usage $M(\mathcal{X})$.

For the sake of simplicity, we henceforth assume unit-size stack frames.

Thinking About Space Usage

Space Usage in the Task-Parallel Model

- We assume every task reserves a certain amount of memory.
  - It is natural to think of it as stack usage, but this is not essential.
- We assume this memory cannot be freed before the task and all its children in the activation tree terminate.
- The schedule $\mathcal{X}$ thus determines space usage $M(\mathcal{X})$.

For the sake of simplicity, we henceforth assume unit-size stack frames.

A Lower Bound

Writing $D(\mathbf{C})$ for the height of the activation tree of $\mathbf{C}$, we have

$$D(\mathbf{C}) \leq M(\mathcal{X}).$$

Thinking About Space Usage

Space Usage in the Task-Parallel Model

- We assume every task reserves a certain amount of memory.
  - It is natural to think of it as stack usage, but this is not essential.
- We assume this memory cannot be freed before the task and all its children in the activation tree terminate.
- The schedule $\mathcal{X}$ thus determines space usage $M(\mathcal{X})$.

For the sake of simplicity, we henceforth assume unit-size stack frames.

A Lower Bound

Writing $D(\mathbf{C})$ for the height of the activation tree of **C**, we have

$$D(\mathbf{C}) \leq M(\mathcal{X}).$$

Clearly there are computations for which linear speedup require linear expansion of space (e.g., p independent tasks). But can it get worse?

Non-strict Computations and Memory Usage

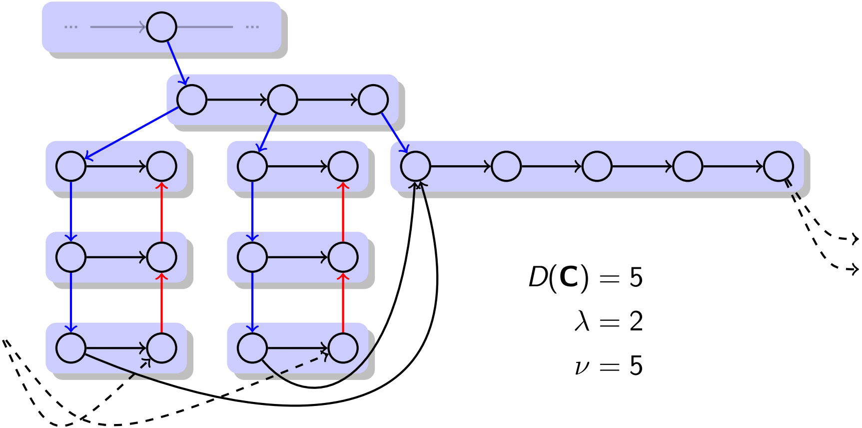
Theorem (Blumofe and Leiserson [1998])

There exists a family of computations for which linear speedup can only be obtained at the cost of a superlinear increase in space usage.

Non-strict Computations and Memory Usage

Theorem (Blumofe and Leiserson [1998])

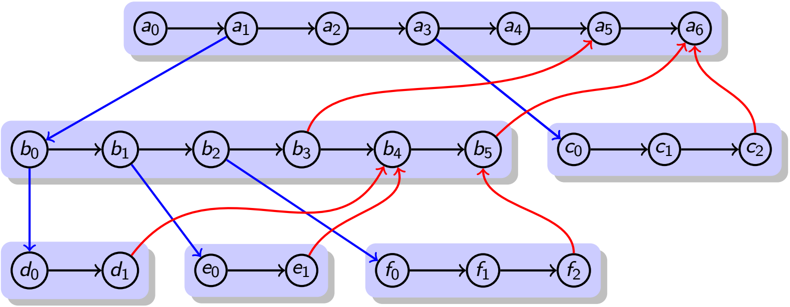
There exists a family of computations for which linear speedup can only be obtained at the cost of a superlinear increase in space usage.



$$D(\mathbf{C}) = 5$$
$$\lambda = 2$$
$$\nu = 5$$

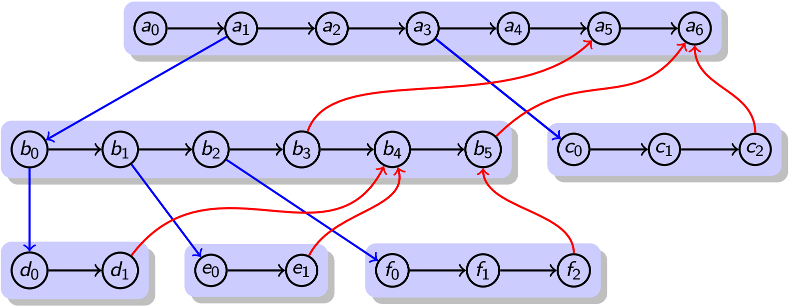
Strict Programs And Eagerness

A fully-strict computation with six tasks $\alpha \in \{a, b, c, d, e, f\}$:



Strict Programs And Eagerness

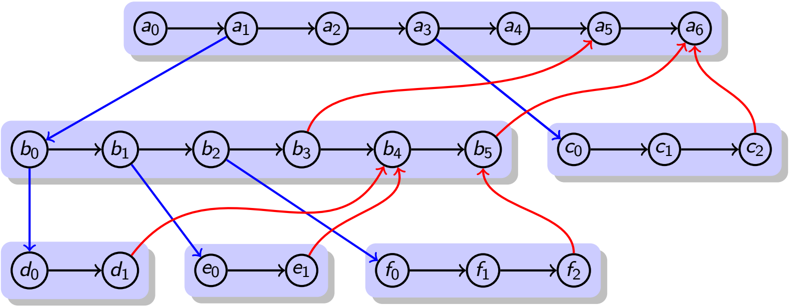
A fully-strict computation with six tasks $\alpha \in \{a, b, c, d, e, f\}$:



- Strictness has important consequences:
  - every task subtree, once started, can be finished by a single processor,
  - a ready leaf task α cannot *stall*, i.e., block on incoming dependencies.

Strict Programs And Eagerness

A fully-strict computation with six tasks $\alpha \in \{a, b, c, d, e, f\}$:

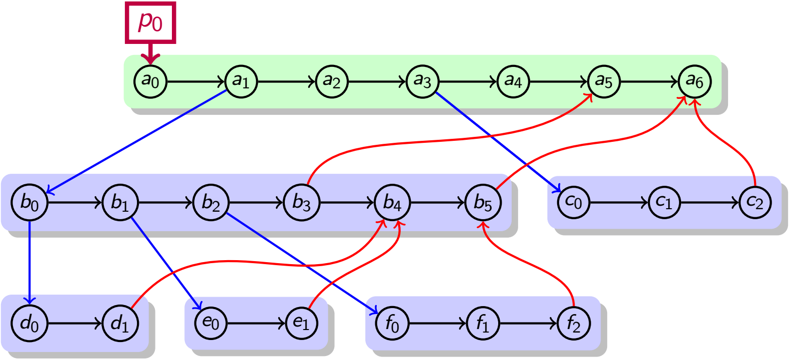


- Strictness has important consequences:
  - every task subtree, once started, can be finished by a single processor,
  - a ready leaf task α cannot *stall*, i.e., block on incoming dependencies.
- We will exploit such properties in a scheduler that guarantees at worst linear space expansion: the *busy-leaves algorithm*.

The Eager Algorithm

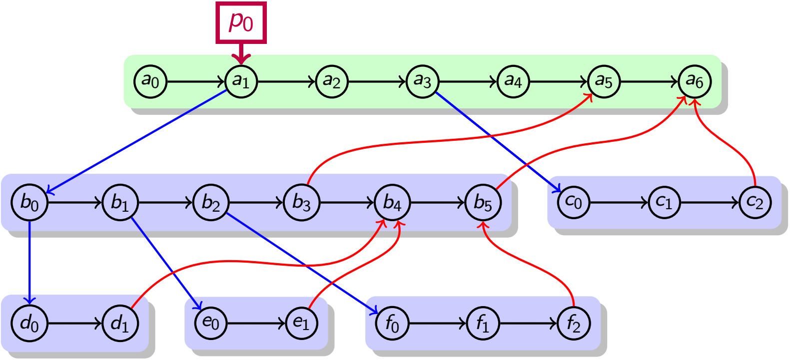
$\alpha_i \leftarrow nil$ for all $i \in [p]$; $R \leftarrow \{\alpha_{init}\}$
while α_{init} is not finished **do**
  for $i \in [p]$ **parallel do**
    if $\alpha_i = nil$ and $R \neq \emptyset$ **then**
      $\alpha_i \leftarrow$ some *ready* task from R; $R \leftarrow \{\alpha_i\} \setminus R$
    end if
    if $\alpha_i \neq nil$ **then**
      execute the next instruction of α_i; let γ be the parent task of α_i
      if α_i has *spawned* β **then**
        $R \leftarrow R \cup \{\alpha_i\}$; $\alpha_i \leftarrow \beta$
      else if α_i is now *stalled* **then**
        $R \leftarrow R \cup \{\alpha_i\}$; $\alpha_i \leftarrow nil$
      else if α_i has *died* **then**
        if γ has no *living* children and $\forall j, \gamma \neq \alpha_j$ **then**
          $\alpha_i \leftarrow \gamma$
        else
          $\alpha_i \leftarrow nil$
        end if
      end if
    end if
  end for
end while

The Eager Algorithm: Example



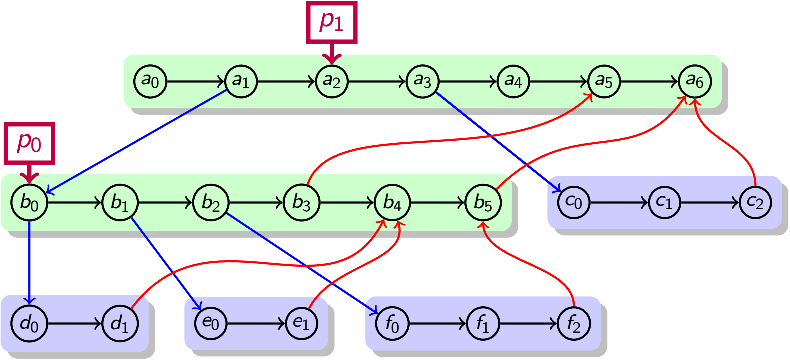
$$R = \emptyset$$

The Eager Algorithm: Example



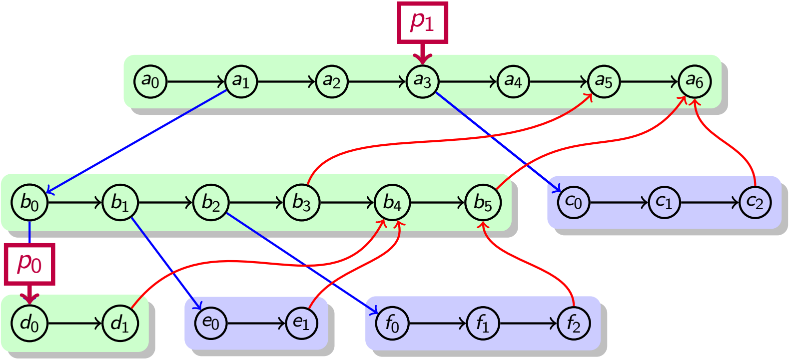
$$R = \emptyset$$

The Eager Algorithm: Example



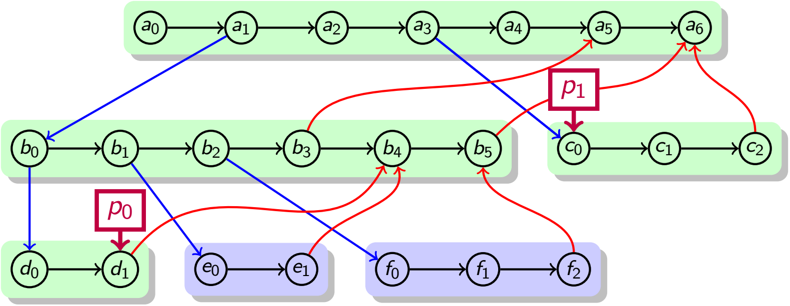
$$R = \emptyset$$

The Eager Algorithm: Example



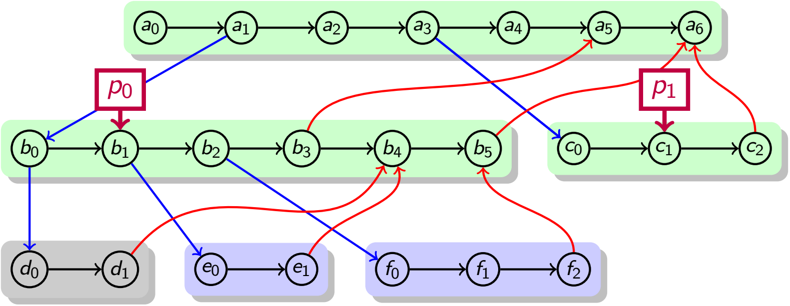
$R = \{\mathbf{b_1}\}$

The Eager Algorithm: Example



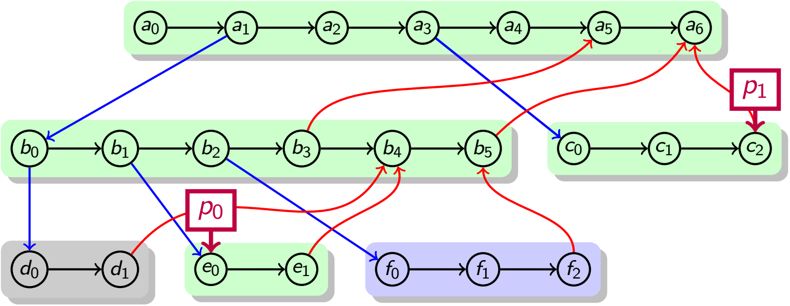
$$R = \{\mathbf{b_1}, \mathbf{a_4}\}$$

The Eager Algorithm: Example



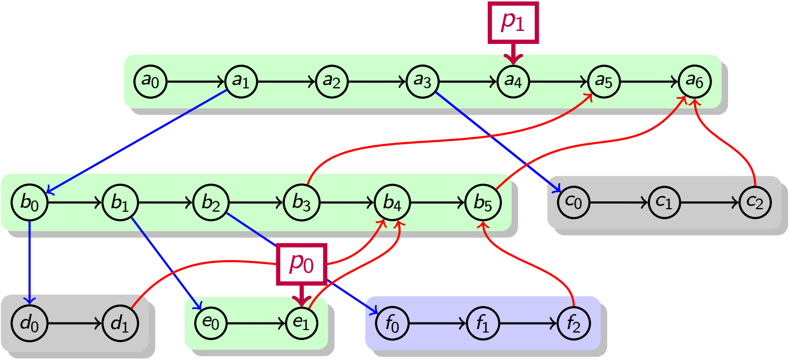
$$R = \{\mathbf{a_4}\}$$

The Eager Algorithm: Example



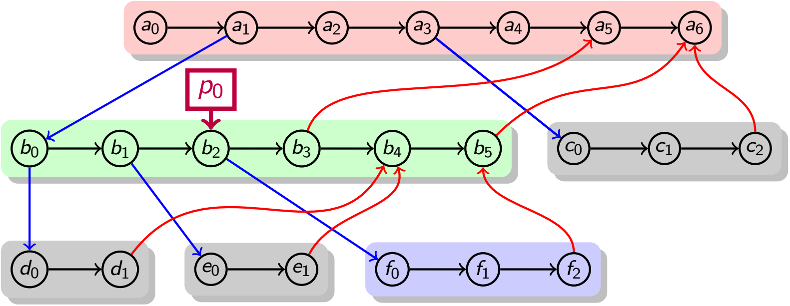
$$R = \{\mathbf{b_2}, \mathbf{a_4}\}$$

The Eager Algorithm: Example



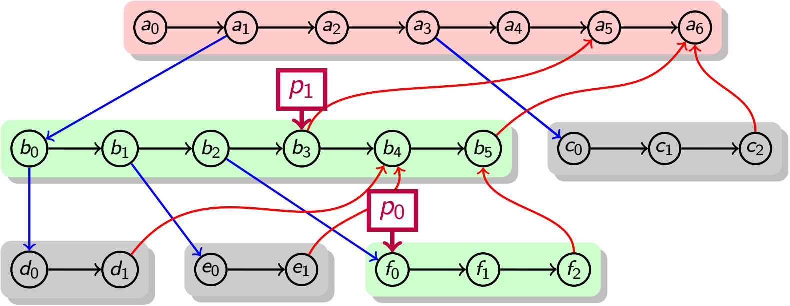
$$R = \{\mathbf{b_2}\}$$

The Eager Algorithm: Example



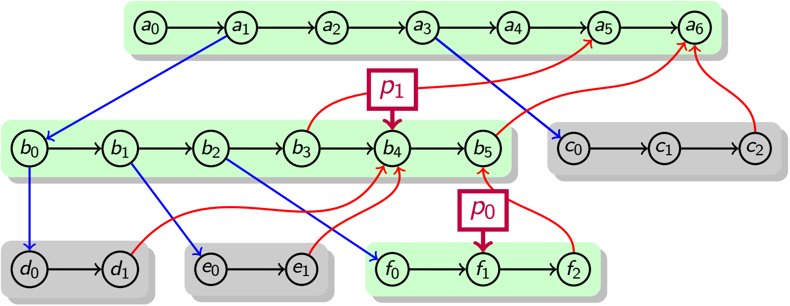
$$R = \{a_5\}$$

The Eager Algorithm: Example



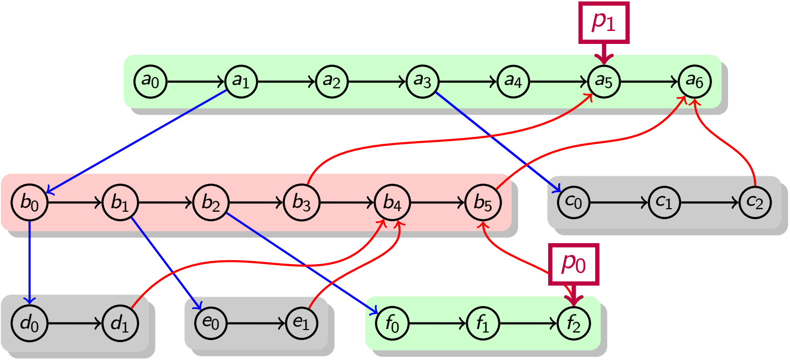
$$R = \{a_5\}$$

The Eager Algorithm: Example



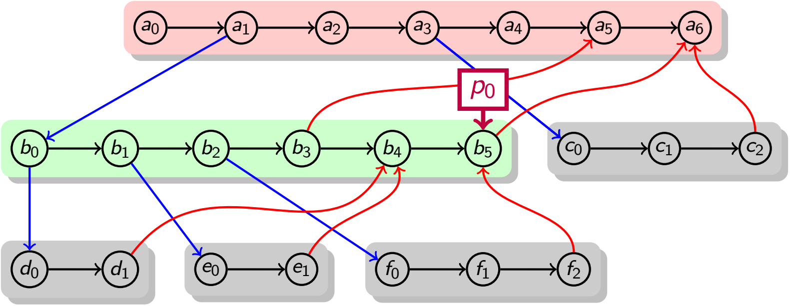
$$R = \{\mathbf{a_5}\}$$

The Eager Algorithm: Example



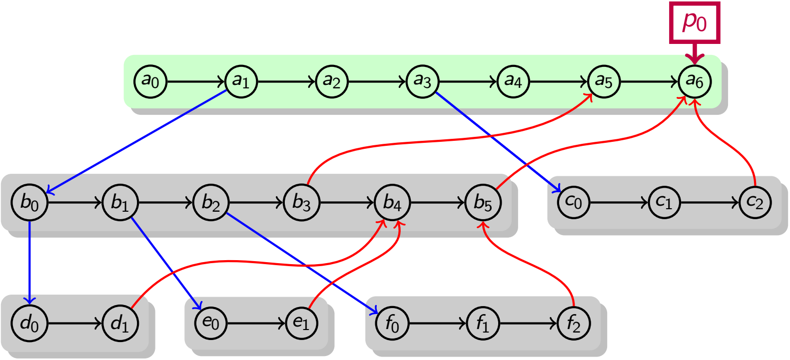
$$R = \{b_5\}$$

The Eager Algorithm: Example



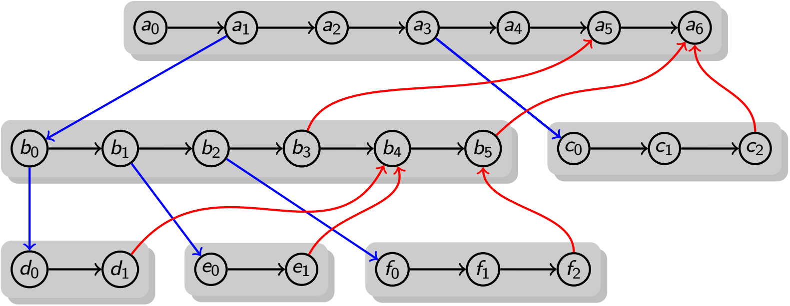
$$R = \{a_6\}$$

The Eager Algorithm: Example



$$R = \emptyset$$

The Eager Algorithm: Example



$$R = \emptyset$$

The Eager Algorithm: Properties

An Online, Serially-Consistent, Greedy, Eager Scheduling Algorithm

- *Online*: does not rely on global graph properties.
- *Serially-consistent*: follows the serial order exactly when $p = 1$.
- *Greedy*: computes greedy schedules, hence $T(\mathcal{X}) \leq \frac{W(\mathbf{C})}{p} + S(\mathbf{C})$.
- *Eager*: computes *eager* schedules, with *busy leaves*.

The Eager Algorithm: Properties

An Online, Serially-Consistent, Greedy, Eager Scheduling Algorithm

- *Online*: does not rely on global graph properties.
- *Serially-consistent*: follows the serial order exactly when $p = 1$.
- *Greedy*: computes greedy schedules, hence $T(\mathcal{X}) \leq \frac{W(\mathbf{C})}{p} + S(\mathbf{C})$.
- *Eager*: computes *eager* schedules, with *busy leaves*.

Definition (Eager Schedules)

At each round, each living task that has no living children is active.

The Eager Algorithm: Properties

An Online, Serially-Consistent, Greedy, Eager Scheduling Algorithm

- *Online*: does not rely on global graph properties.
- *Serially-consistent*: follows the serial order exactly when $p = 1$.
- *Greedy*: computes greedy schedules, hence $T(\mathcal{X}) \leq \frac{W(\mathbf{C})}{p} + S(\mathbf{C})$.
- *Eager*: computes *eager* schedules, with *busy leaves*.

Definition (Eager Schedules)

At each round, each living task that has no living children is active.

Theorem (Blumofe and Leiserson [1994])

If $\mathcal{X}$ is eager then $M(\mathcal{X}) \leq pD(\mathbf{C})$.

The Eager Algorithm: Properties

An Online, Serially-Consistent, Greedy, Eager Scheduling Algorithm

- *Online*: does not rely on global graph properties.
- *Serially-consistent*: follows the serial order exactly when $p = 1$.
- *Greedy*: computes greedy schedules, hence $T(\mathcal{X}) \leq \frac{W(\mathbf{C})}{p} + S(\mathbf{C})$.
- *Eager*: computes *eager* schedules, with *busy leaves*.

Definition (Eager Schedules)

At each round, each living task that has no living children is active.

Theorem (Blumofe and Leiserson [1994])

If $\mathcal{X}$ is eager then $M(\mathcal{X}) \leq pD(\mathbf{C})$.

Proof Sketch.

At any t, at most p leaves are active, each using $D(\mathbf{C})$ space at most. □

The Eager Algorithm: Limitations

Concurrency Issues in the Eager Algorithm

- The pseudocode is, by design, fuzzy on concurrency issues. ▷ pseudocode
- What part of that algorithm should be executed atomically?

The Eager Algorithm: Limitations

Concurrency Issues in the Eager Algorithm

- The pseudocode is, by design, fuzzy on concurrency issues. ▷ pseudocode
- What part of that algorithm should be executed atomically?

Limits of Centralization

- An implementation needs to make accesses to R mutually exclusive.
- This will be difficult to scale beyond a few processors.
- Instead, we would like to use per-processor data structures...

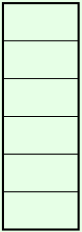
Towards Work Stealing

- We've seen how the eager algorithm strives to mimick serial execution.
- Serial execution relies on a stack to implement its LIFO policy.
- What similar data structure could we use to compute eager schedules?

Towards Work Stealing

- We've seen how the eager algorithm strives to mimick serial execution.
- Serial execution relies on a stack to implement its LIFO policy.
- What similar data structure could we use to compute eager schedules?

Double-Ended Queues and Work-Stealing Scheduling



Organization of the algorithm:

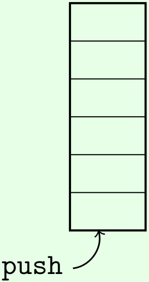
- Each processor has its owns double-ended queue (deque) in which it stores tasks.
- Deques hold the live subset of the spawn tree.

Outline of the scheduler:

Towards Work Stealing

- We've seen how the eager algorithm strives to mimick serial execution.
- Serial execution relies on a stack to implement its LIFO policy.
- What similar data structure could we use to compute eager schedules?

Double-Ended Queues and Work-Stealing Scheduling



Organization of the algorithm:

- Each processor has its owns double-ended queue (deque) in which it stores tasks.
- Deques hold the live subset of the spawn tree.

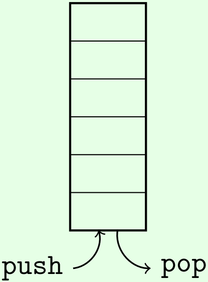
Outline of the scheduler:

1. Push continuation tasks to the bottom (`push`).

Towards Work Stealing

- We've seen how the eager algorithm strives to mimick serial execution.
- Serial execution relies on a stack to implement its LIFO policy.
- What similar data structure could we use to compute eager schedules?

Double-Ended Queues and Work-Stealing Scheduling



Organization of the algorithm:

- Each processor has its owns double-ended queue (deque) in which it stores tasks.
- Deques hold the live subset of the spawn tree.

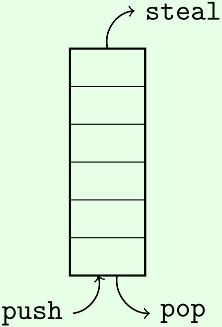
Outline of the scheduler:

1. Push continuation tasks to the bottom (`push`).
2. When out of work, pop from bottom (`pop`),

Towards Work Stealing

- We've seen how the eager algorithm strives to mimick serial execution.
- Serial execution relies on a stack to implement its LIFO policy.
- What similar data structure could we use to compute eager schedules?

Double-Ended Queues and Work-Stealing Scheduling



Organization of the algorithm:

- Each processor has its owns double-ended queue (deque) in which it stores tasks.
- Deques hold the live subset of the spawn tree.

Outline of the scheduler:

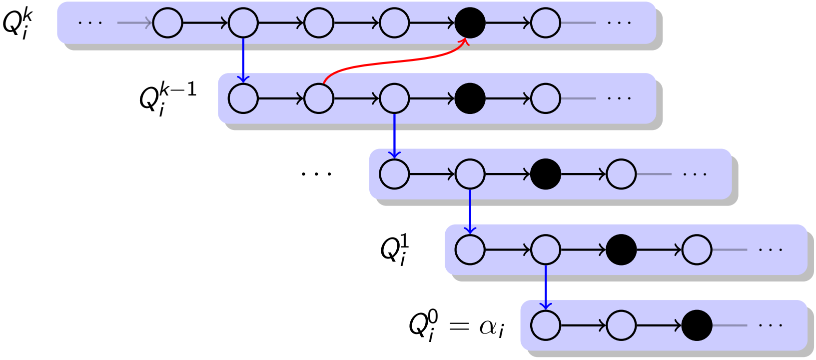
1. Push continuation tasks to the bottom (`push`).
2. When out of work, pop from bottom (`pop`),
3. If pop fails, pick some other processor at random and try to `steal` a task from the top of its deque.

The Work-Stealing Algorithm

```
Q_0 ← {α_init}; Q_i ← empty for all 0 < i < p; alpha_i ← nil for all i ∈ [p];
while α_init is not finished do
    for i ∈ [p] parallel do
        if α_i = nil then
            α_i ← pop(Q_i)
        end if
        if α_i = nil then
            α_i ← steal(Q_j) with j picked randomly in [p];
        end if
        if α_i ≠ nil then
            execute the next instruction of α_i
            if α_i has spawned β then
                push(Q_i, α_i); α_i ← β
            else if α_i is now stalled or has died then
                α_i ← nil
            else if α_i has enabled a stalled β then
                push(Q_i, β)
            end if
        end if
    end for
end while
```

The Work-Stealing Algorithm: Space Usage

Consider the contents of a deque of size k during work-stealing.



Invariant: Q_i^{m+1} has spawned Q_i^m, and only Q_i^k may have worked since.

Theorem

Work-Stealing computes eager schedules.

The Work-Stealing Algorithm: Time Bounds

- Schedules $\mathcal{X}$ computed by work-stealing are eager but not greedy.
  - Steals may fail; Blumofe and Leiserson [1994] also consider contention.
- The length of $\mathcal{X}$ thus depends on the number of steal attempts.

The Work-Stealing Algorithm: Time Bounds

- Schedules $\mathcal{X}$ computed by work-stealing are eager but not greedy.
  - Steals may fail; Blumofe and Leiserson [1994] also consider contention.
- The length of $\mathcal{X}$ thus depends on the number of steal attempts.

Theorem

The expected number of steals is $O(pS(\mathbf{C}))$.

The Work-Stealing Algorithm: Time Bounds

- Schedules $\mathcal{X}$ computed by work-stealing are eager but not greedy.
  - Steals may fail; Blumofe and Leiserson [1994] also consider contention.
- The length of $\mathcal{X}$ thus depends on the number of steal attempts.

Theorem

The expected number of steals is $O(pS(\mathbf{C}))$.

Proof.

Intricate!

The Work-Stealing Algorithm: Time Bounds

- Schedules $\mathcal{X}$ computed by work-stealing are eager but not greedy.
  - Steals may fail; Blumofe and Leiserson [1994] also consider contention.
- The length of $\mathcal{X}$ thus depends on the number of steal attempts.

Theorem

The expected number of steals is $O(pS(\mathbf{C}))$.

Proof.

Intricate! Intuitively, a large number of steal attempts is unlikely because it would be reflecting the persistent presence of certain instructions, deemed *critical*. But such instructions cannot occur deep in the deque, and hence are eliminated with high probability by steal attempts. □

The Work-Stealing Algorithm: Time Bounds

- Schedules $\mathcal{X}$ computed by work-stealing are eager but not greedy.
  - Steals may fail; Blumofe and Leiserson [1994] also consider contention.
- The length of $\mathcal{X}$ thus depends on the number of steal attempts.

Theorem

The expected number of steals is $O(pS(\mathbf{C}))$.

Proof.

Intricate! Intuitively, a large number of steal attempts is unlikely because it would be reflecting the persistent presence of certain instructions, deemed *critical*. But such instructions cannot occur deep in the deque, and hence are eliminated with high probability by steal attempts. □

Theorem

The expected length is $W(\mathbf{C})/p + O(S(\mathbf{C}))$.

An Important Metric: Deviations

Interacting with the scheduler is both a blessing and a curse.

- It is necessary to exploit parallelism, obviously.
- It incurs high costs: bookkeeping overheads, loss of locality.

Fortunately, one can bound the number of *deviations* [Acar et al., 2000].

An Important Metric: Deviations

Interacting with the scheduler is both a blessing and a curse.

- It is necessary to exploit parallelism, obviously.
- It incurs high costs: bookkeeping overheads, loss of locality.

Fortunately, one can bound the number of *deviations* [Acar et al., 2000].

Definition (Deviation)

A *deviation* is a pair $(a, b) \in \mathbf{C}^2$ such that b occurs immediately after a in the serial execution but either $\mathcal{X}(a).\mathrm{p} \neq \mathcal{X}(b).\mathrm{p}$ or $\mathcal{X}(b).\mathrm{t} > \mathcal{X}(a).\mathrm{t} + 1$.

An Important Metric: Deviations

Interacting with the scheduler is both a blessing and a curse.

- It is necessary to exploit parallelism, obviously.
- It incurs high costs: bookkeeping overheads, loss of locality.

Fortunately, one can bound the number of *deviations* [Acar et al., 2000].

Definition (Deviation)

A *deviation* is a pair $(a, b) \in \mathbf{C}^2$ such that b occurs immediately after a in the serial execution but either $\mathcal{X}(a).\mathrm{p} \neq \mathcal{X}(b).\mathrm{p}$ or $\mathcal{X}(b).\mathrm{t} > \mathcal{X}(a).\mathrm{t} + 1$.

Theorem (Acar et al. [2000])

Work-Stealing incurs an expected number of $O(pS(\mathbf{C}))$ *deviations.*

Proof Sketch.

Each steal induces at most two deviations. Can you guess which?

An Important Metric: Deviations

Interacting with the scheduler is both a blessing and a curse.

- It is necessary to exploit parallelism, obviously.
- It incurs high costs: bookkeeping overheads, loss of locality.

Fortunately, one can bound the number of *deviations* [Acar et al., 2000].

Definition (Deviation)

A *deviation* is a pair $(a, b) \in \mathbf{C}^2$ such that b occurs immediately after a in the serial execution but either $\mathcal{X}(a).\mathrm{p} \neq \mathcal{X}(b).\mathrm{p}$ or $\mathcal{X}(b).\mathrm{t} > \mathcal{X}(a).\mathrm{t} + 1$.

Theorem (Acar et al. [2000])

Work-Stealing incurs an expected number of $O(pS(\mathbf{C}))$ deviations.

Proof Sketch.

Each steal induces at most two deviations. Can you guess which?
Steals themselves, as well as enablings of stalled tasks. □

Towards An Implementation

Ok, are we ready to implement Cilk?

Towards An Implementation

Ok, are we ready to implement Cilk? Still missing:

- Manipulating programs rather than abstract graphs.
- Implementing deques.
- Clarifying dependence resolution, as used when enabling stalled tasks.

Naive Compilation

What do we put in work-stealing deques, concretely? *Frames*.

Naive Compilation

What do we put in work-stealing deques, concretely? *Frames*.

Frames and Their Usage

- Frames package the data necessary to run Cilk tasks.
- They typically bundle a code pointer, a parent-task pointer, local data used by the program, and bookkeeping data used by the scheduler.
- Portable implementations store a shadow stack in the heap.
  - This suffers from restrictions Mainly, C code cannot call Cilk functions.
- Ambitious Cilk implementations use actual stack frames.
  - The system stack becomes a cactus stack, i.e., a tree. Systems aficionados may want to read Yang and Mellor-Crummey [2016].

Naive Compilation

What do we put in work-stealing deques, concretely? *Frames*.

Frames and Their Usage

- Frames package the data necessary to run Cilk tasks.
- They typically bundle a code pointer, a parent-task pointer, local data used by the program, and bookkeeping data used by the scheduler.
- Portable implementations store a shadow stack in the heap.
  - This suffers from restrictions Mainly, C code cannot call Cilk functions.
- Ambitious Cilk implementations use actual stack frames.
  - The system stack becomes a cactus stack, i.e., a tree. Systems aficionados may want to read Yang and Mellor-Crummey [2016].

Naive Compilation

What do we put in work-stealing deques, concretely? *Frames*.

Frames and Their Usage

- Frames package the data necessary to run Cilk tasks.
- They typically bundle a code pointer, a parent-task pointer, local data used by the program, and bookkeeping data used by the scheduler.
- Portable implementations store a shadow stack in the heap.
  - This suffers from restrictions Mainly, C code cannot call Cilk functions.
- Ambitious Cilk implementations use actual stack frames.
  - The system stack becomes a cactus stack, i.e., a tree. Systems aficionados may want to read Yang and Mellor-Crummey [2016].

Where do Code Pointers Come From?

The Cilk compiler performs a source-to-source, partial CPS translation, outlining every continuation of `cilk_spawn` into its own C function.

Naive Deques and Counters

How do we implement deques? How do we do dependence resolution?

Naive Deques and Counters

How do we implement deques? How do we do dependence resolution?

Not so Concurrent Deques

- Protect your favorite sequential deque with your favorite lock.
- Important: in the implementation of `steal`, use `try_lock`.

Naive Deques and Counters

How do we implement deques? How do we do dependence resolution?

Not so Concurrent Deques

- Protect your favorite sequential deque with your favorite lock.
- Important: in the implementation of `steal`, use `try_lock`.

Dependence Resolution

- Frames include a *join counter*, manipulated as follows.
  1. It is initialized to one.
  2. Spawning a task increment the counter of its frame.
  3. Joining a task decrements the counter of its parent's frame. If it reaches zero, the parent is pushed to the bottom of the local deque.
  4. Syncinc a task decrements the counter of its frame. If it reaches zero, `sync` proceeds. Otherwise, it returns to the scheduler.
- **Key invariant**: every frame present in a deque has a counter > 1.

The Work-First Principle (1/2)

We have a working, provably-efficient Cilk implementation.

The Work-First Principle (1/2)

We have a working, provably-efficient Cilk implementation. Also, a quite slow one. What should we optimize?

The Work-First Principle (1/2)

We have a working, provably-efficient Cilk implementation. Also, a quite slow one. What should we optimize? Let's exploit our time bound.

The Work-First Principle (1/2)

We have a working, provably-efficient Cilk implementation. Also, a quite slow one. What should we optimize? Let's exploit our time bound.

Notations for overheads

- Let T_p denote $\min_{\mathcal{X}} T(\mathcal{X})$ where $\mathcal{X}$ is a p-processor schedule.
- Let T_s denote the running time of the program's serial elision.
- Write c_1 for T_1/T_s and c_∞ for the hidden constant in $O(S(\mathbf{C}))$.

The Work-First Principle (1/2)

We have a working, provably-efficient Cilk implementation. Also, a quite slow one. What should we optimize? Let's exploit our time bound.

Notations for overheads

- Let T_p denote $\min_{\mathcal{X}} T(\mathcal{X})$ where $\mathcal{X}$ is a p-processor schedule.
- Let T_s denote the running time of the program's serial elision.
- Write c_1 for T_1/T_s and c_∞ for the hidden constant in $O(S(\mathbf{C}))$.

The work-stealing time bound becomes as

$$\begin{aligned} T_p &\leq c_1 T_s/p + c_\infty S(\mathbf{C}) \\ &\approx c_1 T_s/p && \text{assuming } P/p \gg c_\infty. \end{aligned}$$

The Work-First Principle (1/2)

We have a working, provably-efficient Cilk implementation. Also, a quite slow one. What should we optimize? Let's exploit our time bound.

Notations for overheads

- Let T_p denote $\min_{\mathcal{X}} T(\mathcal{X})$ where $\mathcal{X}$ is a p-processor schedule.
- Let T_s denote the running time of the program's serial elision.
- Write c_1 for T_1/T_s and c_∞ for the hidden constant in $O(S(\mathbf{C}))$.

The work-stealing time bound becomes as

$$\begin{aligned} T_p &\leq c_1 T_s/p + c_\infty S(\mathbf{C}) \\ &\approx c_1 T_s/p && \text{assuming } P/p \gg c_\infty. \end{aligned}$$

This has an important consequence, called the *work-first principle*:

- c_1 matters more for performance than c_∞,
- therefore we should minimize c_1, even at the cost of a larger c_∞.

Fast Clone, Slow Clone (1/3)

The General Idea

Actual compilers compile every Cilk function into two distinct C functions:

- the *fast clone* contains almost very little parallel bookkeeping,
- the *slow clone* contains bookkeeping, but is only called after a steal.

Since steals are rare, fast clones dominate: we are putting work first.

Fast Clone, Slow Clone (1/3)

The General Idea

Actual compilers compile every Cilk function into two distinct C functions:

- the *fast clone* contains almost very little parallel bookkeeping,
- the *slow clone* contains bookkeeping, but is only called after a steal.

Since steals are rare, fast clones dominate: we are putting work first.

How do Slow Clones Work?

- Created on steals. Spawns fast clones itself.
- Its `spawn`/`join`/`sync` are implemented as in previous frames.

Fast Clone, Slow Clone (1/3)

The General Idea

Actual compilers compile every Cilk function into two distinct C functions:

- the *fast clone* contains almost very little parallel bookkeeping,
- the *slow clone* contains bookkeeping, but is only called after a steal.

Since steals are rare, fast clones dominate: we are putting work first.

How do Slow Clones Work?

- Created on steals. Spawns fast clones itself.
- Its `spawn`/`join`/`sync` are implemented as in previous frames.

How do Fast Clones Work?

- **Invariant**: a fast clone has never been stolen.
- `spawn`: normal C call.
- `join`: as in the slow clone.
- `sync`: completely free!

Fast Clone, Slow Clone (2/3)

Here is the fast clone for our `fib` function from the first slides. ▷ code

```
unsigned int fib_fast(unsigned int n) {
  unsigned int x, y;
  if (n <= 1)
    return 1;
  fib_frame *f = alloc_fib_frame(); /* Prepare frame by... */
  f->continuation = FIB_CONT_0;     /* ... storing the continuation... */
  f->n = n;                         /* ... and live data. */
  deque_push(f);                    /* Push it onto the work-stealing deque. */
  x = fib(n - 1);                   /* Run the work. */
  if (fib_pop(x) == FAILURE)        /* Has the parent been stolen? */
    join_and_return_to_scheduler(); /* Get back to scheduling loop. */
  y = fib_fast(n - 2);              /* Run the code sequentially. */
  ;                                 /* Sync is free! */
  destroy_fib_frame(f);             /* Deallocate the frame. */
  return x + y;                     /* Return to caller. */
}
```

Fast Clone, Slow Clone (3/3)

Here is the slow clone for our `fib` function from the first slides. ▷ code

```
void fib_slow(fib_frame *self) {
  unsigned int n;
  switch (self->continuation) {
  case FIB_CONT_0: goto L_FIB_CONT_0;
  case FIB_CONT_1: goto L_FIB_CONT_1;
  }
  ...;                           /* Same code as in fast clone, except... */
  if (fib_pop(x) == FAILURE)
    join_and_return_to_scheduler();
  if (0) {                       /* ... at continuation labels. */
  L_FIB_CONT_0:
    n = self->n;                 /* Reload live data. */
  }
  ...;
  if (sync() == FAILURE)         /* Check join counter. */
    join_and_return_to_scheduler();
  if (0) {
  L_FIB_CONT_1:
    x = self->x; y = self->y;   /* Reload live data. */
  }
  ...;                           /* Run the continuation of sync, free frames. */
  return x + y;                  /* Return to caller. */
}
```

Concurrent Deques?

- Sequential deques with locks are fast in the absence of contention. This might be good enough for Cilk, where steals are infrequent.

Concurrent Deques?

- Sequential deques with locks are fast in the absence of contention. This might be good enough for Cilk, where steals are infrequent.
- Frigo et al. [1998] used the *THE* protocol for concurrent deque access.

Concurrent Deques?

- Sequential deques with locks are fast in the absence of contention. This might be good enough for Cilk, where steals are infrequent.
- Frigo et al. [1998] used the *THE* protocol for concurrent deque access.
- Many general-purpose concurrent deques have been studied since.

Concurrent Deques?

- Sequential deques with locks are fast in the absence of contention. This might be good enough for Cilk, where steals are infrequent.
- Frigo et al. [1998] used the *THE* protocol for concurrent deque access.
- Many general-purpose concurrent deques have been studied since.

The Key Issues

- Steals might race with `push`.
- Steals might race with `pop`.
- Steals might race with each other.
- The deque might need to be resized.

Concurrent Deques?

- Sequential deques with locks are fast in the absence of contention. This might be good enough for Cilk, where steals are infrequent.
- Frigo et al. [1998] used the *THE* protocol for concurrent deque access.
- Many general-purpose concurrent deques have been studied since.

The Key Issues

- Steals might race with `push`.
- Steals might race with `pop`.
- Steals might race with each other.
- The deque might need to be resized.

The State of the Art

We will have a glance at the dynamic circular deques proposed by Chase and Lev [2005], in a presentation due to Lê et al. [2013].

Chase-Lev Deques: Push

```
void push(deque *q, int x) {
  size_t b = atomic_load(&q->bottom);
  size_t t = atomic_load(&q->top);
  array *a = atomic_load(&q->array);
  if (b - t > a->size - 1) { /* Full queue. */
    resize(q);
    a = atomic_load(&q->array);
  }
  atomic_store(&a->buffer[b % a->size], x);
  atomic_store(&q->bottom, b + 1);
}
```

Chase-Lev Deques: Pop

```
int pop(deque *q) {
  size_t b = atomic_load(&q->bottom) - 1;
  array *a = atomic_load(&q->array);
  atomic_store(&q->bottom, b);
  size_t t = atomic_load(&q->top);
  int x;
  if (t <= b) { /* Non-empty queue. */
    x = atomic_load(&a->buffer[b % a->size]);
    if (t == b) { /* Single last element in queue. */
      if (!compare_exchange_strong(&q->top, &t, t + 1))
        x = EMPTY; /* Failed race. */
      atomic_store(&q->bottom, b + 1);
    }
  } else { /* Empty queue. */
    x = EMPTY;
    atomic_store(&q->bottom, b + 1);
  }
  return x;
}
```

Chase-Lev Deques: Steal

```
int steal(deque *q) {
  size_t t = atomic_load(&q->top);
  size_t b = atomic_load(&q->bottom);
  int x = EMPTY;
  if (t < b) {
    /* Non-empty queue. */
    array *a = atomic_load(&q->array);
    x = atomic_load(&a->buffer[t % a->size]);
    if (!compare_exchange_strong(&q->top, &t, t + 1))
      /* Failed race. */
      return ABORT;
  }
  return x;
}
```

(Chase-Lev Deques: Weak Memory Models)

Lê et al. [2013] actually gave a *relaxed* version of Chase-Lev deques.

(Chase-Lev Deques: Weak Memory Models)

Lê et al. [2013] actually gave a *relaxed* version of Chase-Lev deques.

Look at the code and its proof, if you dare!

All These Things We Didn't Talk About

State-of-the-art implementations of task parallelism may use:

- dedicated compiler transformations [Schardl et al., 2017],
- efficient join counters such as SNZI [Ellen et al., 2007],
- lower-level primitives than `spawn`/`sync` [Acar et al., 2016b],
- disciplined, provably-efficient uses of futures [Lee et al., 2015].

Conclusion

We've had a look at the design of Cilk runtime systems.

- The work-span model allows us to design *provably-efficient* runtimes.
- Time efficiency is generally easy to obtain.
  - Be it via Brent's scheduler, or greedy scheduling, or work-stealing...
- At-most-linear space expansion is impossible in general.
  - This justifies the restriction to (full) strictness in Cilk.
- Work-Stealing schedulers use double-ended queues to store ready tasks in a decentralized way.

Read `https://github.com/OpenCilk/cilkrts` for more!

Lecture 3

Formal Semantics of Task Parallelism

See `semantics-intro.pdf`.

Bibliography

References I

Matteo Frigo, Charles Leiserson, and Keith Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Programming Language Design and Implementation (PLDI'98)*. ACM, 1998. URL `http://supertech.csail.mit.edu/papers/cilk5.pdf`.

Robert Blumofe and Charles Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Symposium on Foundations of Computer Science (FOCS'94)*. IEEE, 1994. URL `http://supertech.csail.mit.edu/papers/steal.pdf`.

Richard P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM*, 21(2):201–206, 1974. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.9361&rep=rep1&type=pdf`.

References II

Umut Acar, Arthur Charguéraud, and Mike Rainey. Oracle-Guided Scheduling for Controlling Granularity in Implicitly Parallel Languages. *Journal of Functional Programming*, 26, November 2016a. URL `https://hal.inria.fr/hal-01409069/document`.

Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Sequential Random Permutation, List Contraction and Tree Contraction are Highly Parallel. In *Discrete Algorithms (SODA'15)*. Society for Industrial and Applied Mathematics, Dec 2014. ISBN 9781611973730. doi: 10.1137/1.9781611973730.30. URL `https://www.cs.cmu.edu/~guyb/papers/SGBFG15.pdf`.

Guy Blelloch, Jeremy Fineman, Phillip Gibbons, and Julian Shun. Internally Deterministic Parallel Algorithms Can Be Fast. *Principles and Practice of Parallel Programming (PPoPP'12)*, 47(8):181, Sep 2012. ISSN 0362-1340. doi: 10.1145/2370036.2145840. URL `https://www.cs.cmu.edu/~guyb/papers/BFGS12.pdf`.

Robert D. Blumofe and Charles E. Leiserson. Space-Efficient Scheduling of Multithreaded Computations. *SIAM Journal on Computing*, 27(1): 202–229, Feb 1998. ISSN 1095-7111. URL `https://epubs.siam.org/doi/abs/10.1137/S0097539793259471?journalCode=smjcat`.

Umut Acar, Guy Blelloch, and Robert Blumofe. The Data Locality of Work Stealing. *Symposium on Parallel Algorithms and Architectures (SPAA'00)*, 2000. doi: 10.1145/341800.341801. URL `https://www.cs.cmu.edu/~guyb/papers/locality2000.pdf`.

Chaoran Yang and John Mellor-Crummey. A Practical Solution to the Cactus Stack Problem. In *Parallelism in Algorithms and Architectures (SPAA'16)*. ACM, 2016. ISBN 9781450342100. doi: 10.1145/2935764.2935787. URL `http://chaoran.me/assets/pdf/ws-spaa16.pdf`.

David Chase and Yossi Lev. Dynamic circular work-stealing deque. *Symposium on Parallel Algorithms and Architectures (SPAA'05)*, 2005. URL `https://www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-dequeue.pdf`.

Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *Principles and Practice of Parallel Programming (PPoPP '13)*, 2013. URL `https://www.di.ens.fr/~zappa/readings/ppopp13.pdf`.

Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Principles and Practice of Parallel Programming*. ACM, 2017. ISBN 9781450344937. doi: 10.1145/3018743.3018758. URL `https://papers.wsmoses.com/tapir.pdf`.

References V

Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: Scalable NonZero Indicators. In *Principles of Distributed Computing (PODC'07)*. ACM, 2007. ISBN 9781595936165. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.169.6386&rep=rep1&type=pdf.

Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. Dag-Calculus: A Calculus for Parallel Computation. volume 51, page 18–32. ACM, Sep 2016b. doi: 10.1145/3022670.2951946. URL https://hal.inria.fr/hal-01409022/file/dag_calculus_icfp16.pdf.

I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. On-the-Fly Pipeline Parallelism. *Transactions on Parallel Computing*, 2015. doi: 10.1145/2809808. URL https://www.cse.wustl.edu/~angelee/home_page/papers/pipep-topc.pdf.