System Software Support for Parallel Programming

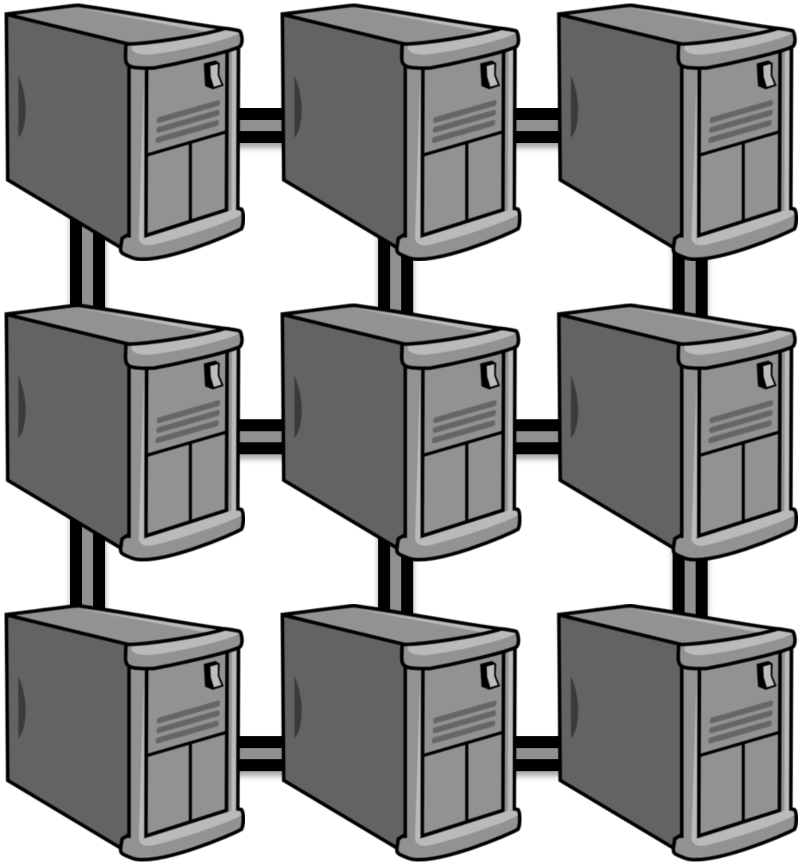
I-Ting Angelina Lee

CSE 591, Fall 2018

What Is Parallel Programming?

- Divide up your computation into multiple components that can be worked on in parallel …
- So that you can simultaneously use multiple compute resources to solve the computational problem.

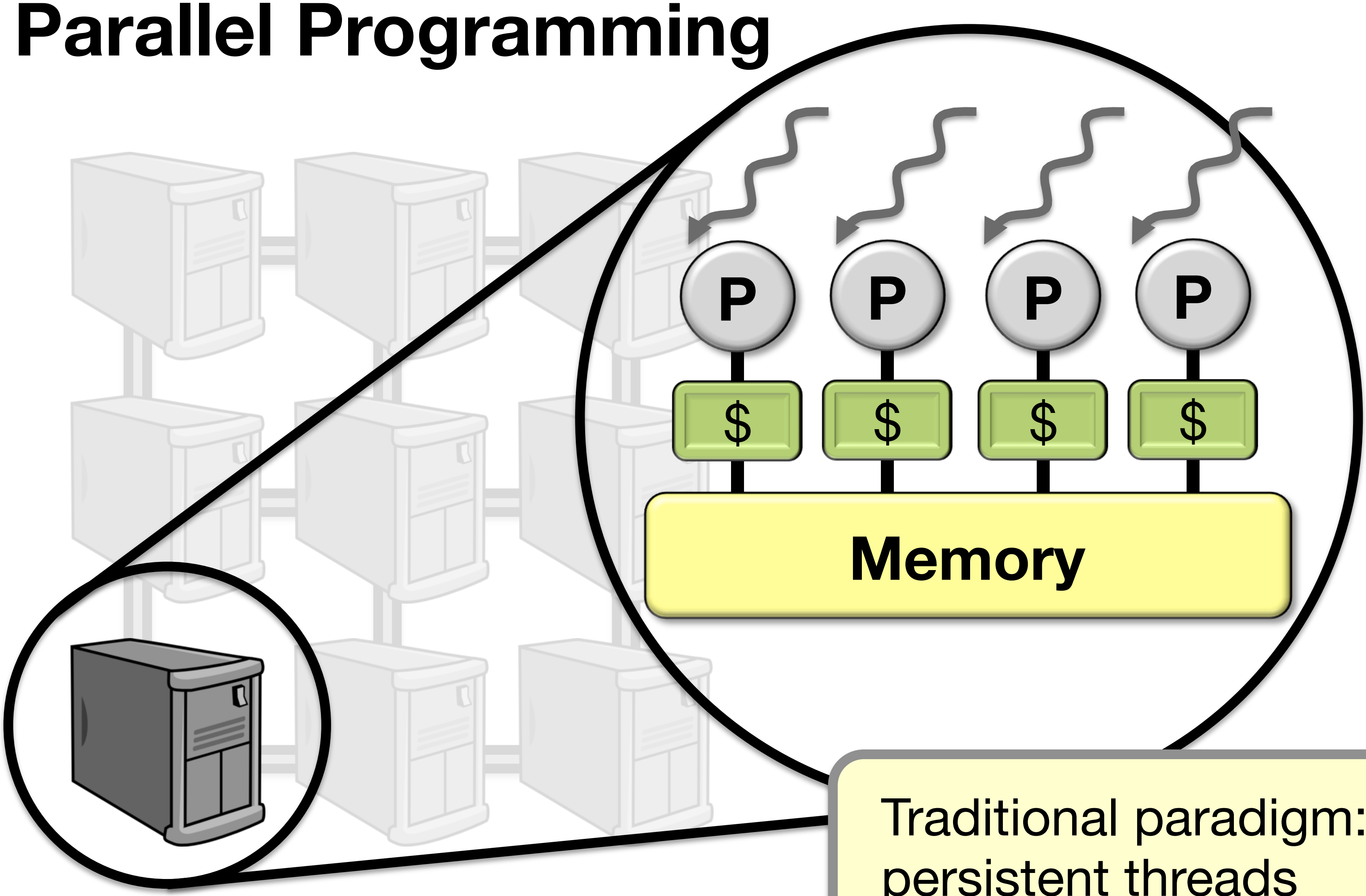
Different Types of Parallel Programming



Program it using MPI
(Message Passing Interface)

Supercomputer: multiple computing nodes connected with high-bandwidth network

Different Types of Parallel Programming



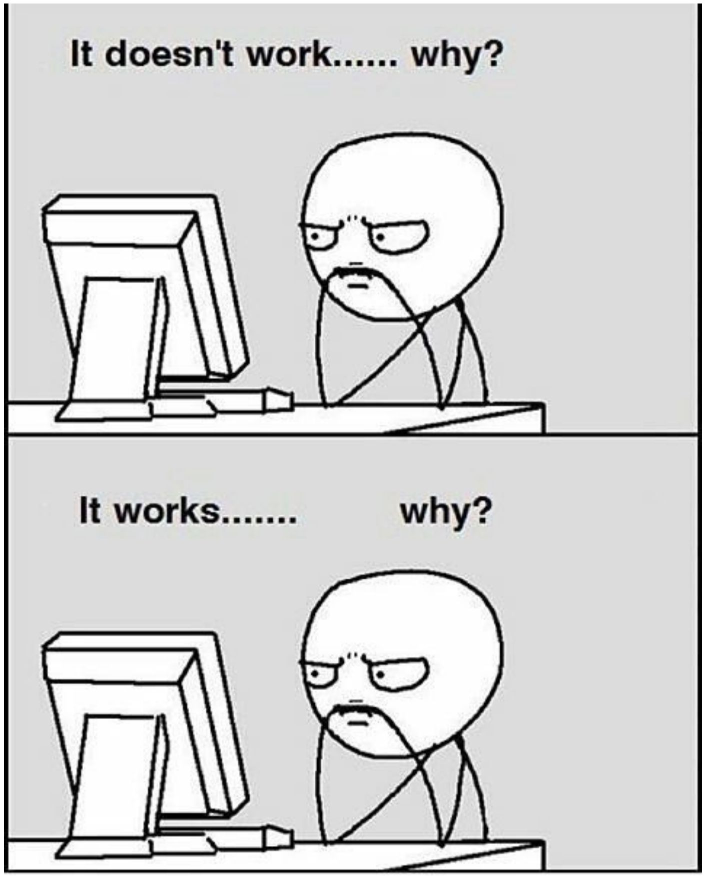
Traditional paradigm:
persistent threads

Within a single node:
multiple cores with shared memory

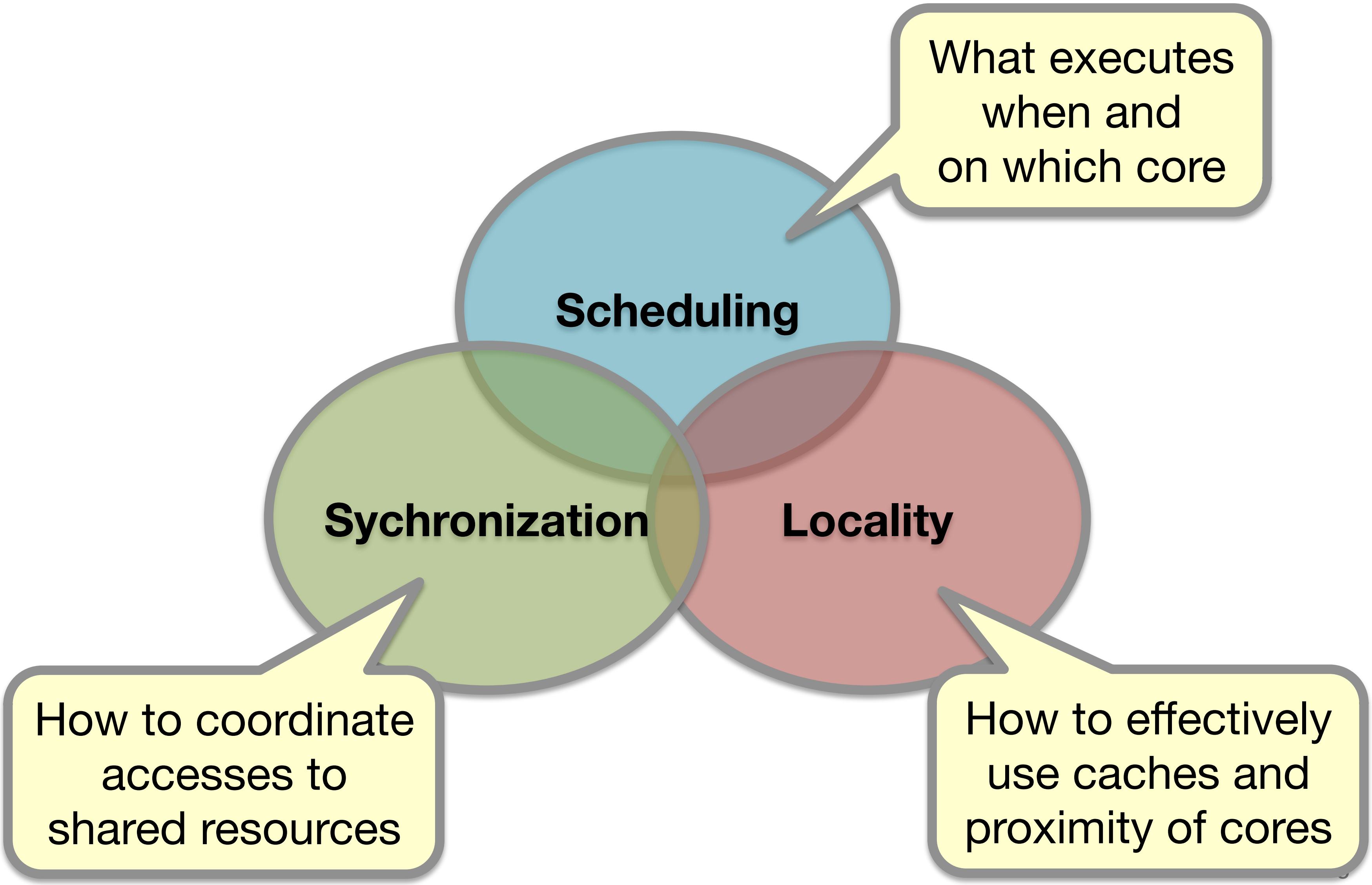
Why Parallel Programming?

- Performance!

Problem: Parallel programming is hard



Challenges in Programming a Multicore Machine



What executes when and on which core

Scheduling

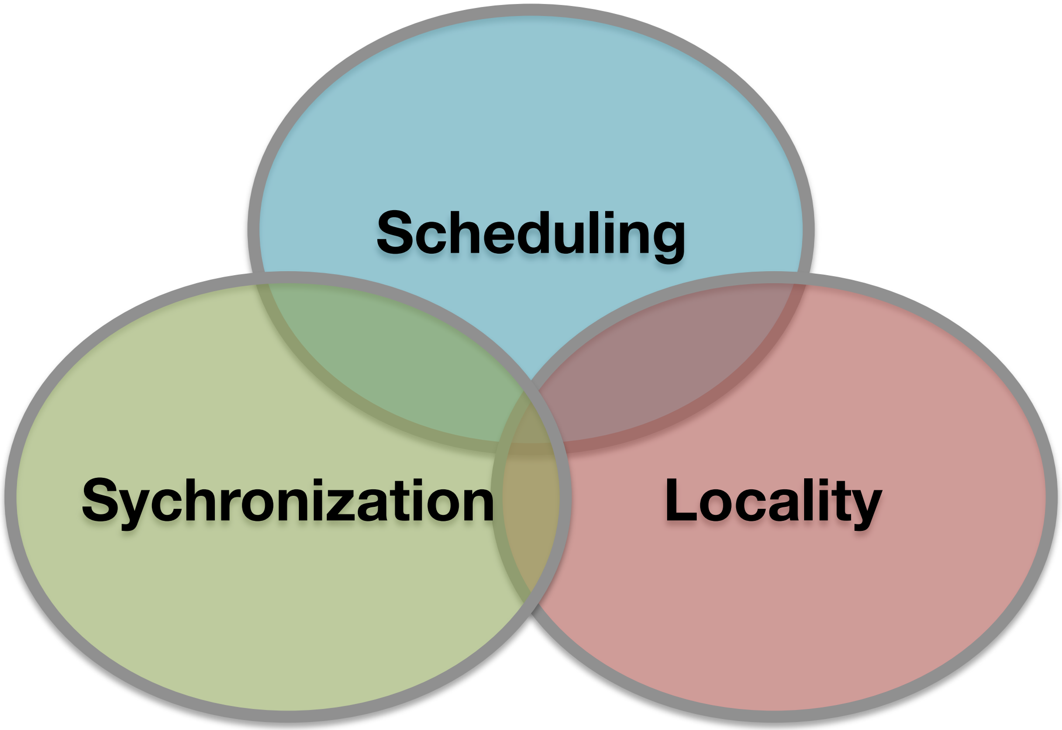
Sychronization

Locality

How to coordinate accesses to shared resources

How to effectively use caches and proximity of cores

Challenges in Programming a Multicore Machine



Scheduling

Sychronization

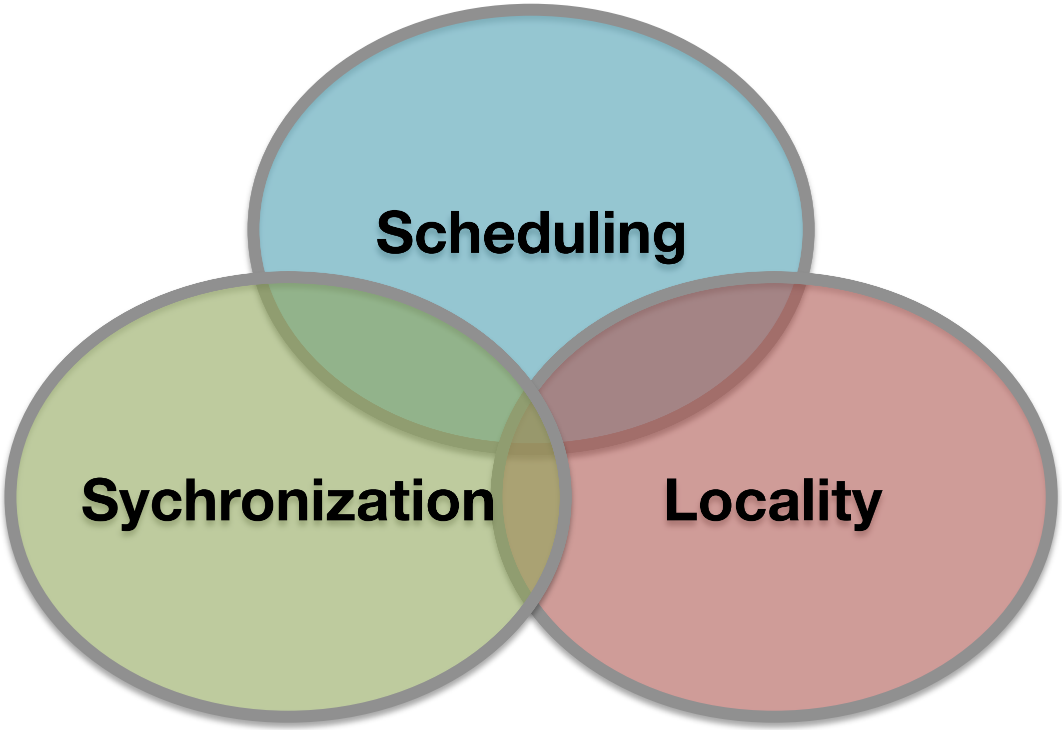
Locality

Traditional paradigm (pthreads) does not address these challenges well.

My Research Goal

Make parallel programming on commodity multicore hardware accessible for everyone.

Challenges in Programming a Multicore Machine



Scheduling

Sychronization

Locality

Traditional paradigm (pthreads) does not address these challenges well.

Example Application: Dedup*

Dedup compresses a stream of data by compressing unique elements and removing duplicates.

```
int fd_out = open_output_file();
bool done = false;
while(!done) {
   chunk_t *chunk = get_next_chunk();
   if(chunk == NULL) { done = true; }
   else {

      chunk->is_dup = deduplicate(chunk);

      if(!chunk->is_dup) compress(chunk);

      write_to_file(fd_out, chunk);
   }
}
```

Stage 0: While there is more data, read the next chunk from the stream.

Stage 1: Check for duplicates.

Stage 2: Compress first-seen chunk.

Stage 3: Write to output file.

*Extrapolated from the PARSEC benchmark [BKS08]

Pipeline Parallelism in Dedup

```
while(!done) {
   chunk_t *chunk = get_next_chunk();
   if(chunk == NULL) { done = true; }
   else {
      chunk->is_dup = deduplicate(chunk);
      if(!chunk->is_dup) compress(chunk);
      write_to_file(fd_out, chunk);
   }
}
```

Stage 0

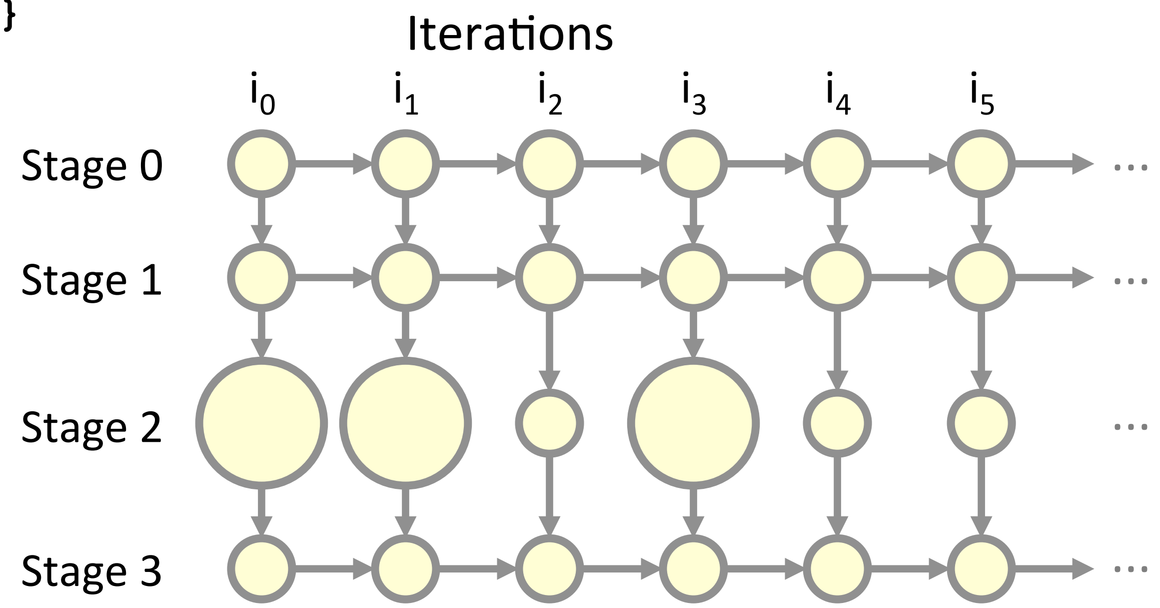
Stage 1

Stage 2

Stage 3

Let's model Dedup's execution as a **pipeline dag**.

- A **node** denotes the execution of a stage in an iteration.
- **Edges** denote dependencies between nodes.



Dedup exhibits **pipeline parallelism**.

→ : **Cross Edge**

Parallelizing Dedup with Pthreads

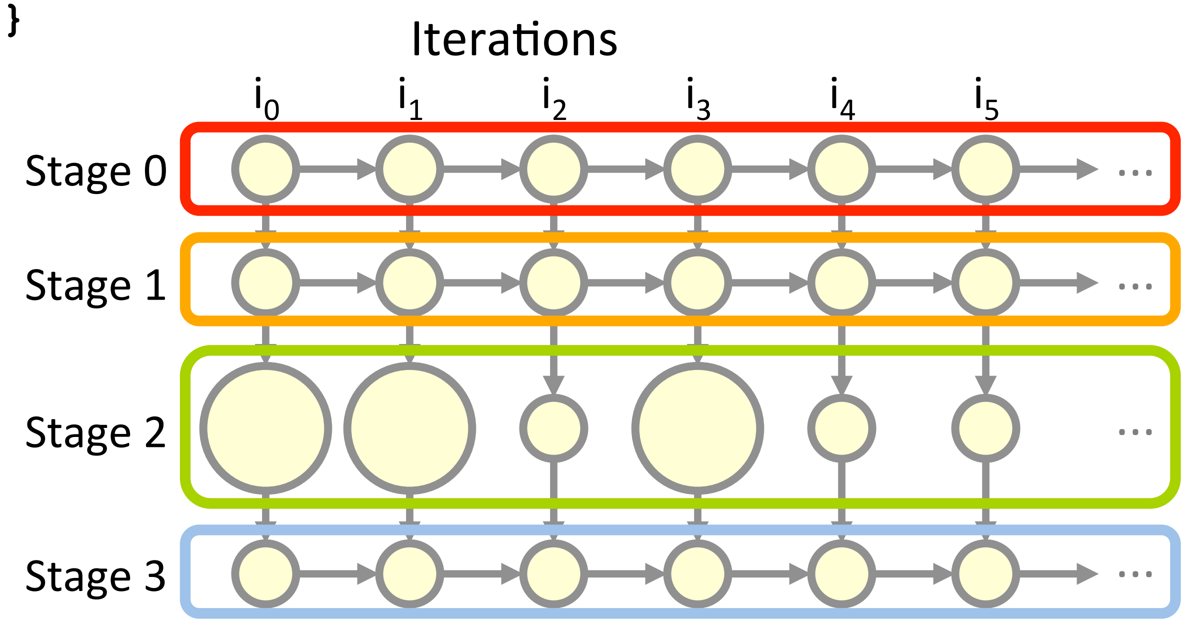
```
while(!done) {
   chunk_t *chunk = get_next_chunk();
   if(chunk == NULL) { done = true; }
   else {

      chunk->is_dup = deduplicate(chunk);

      if(!chunk->is_dup) compress(chunk);

      write_to_file(fd_out, chunk);
   }
}
```

Stage 0

Stage 1

Stage 2

Stage 3



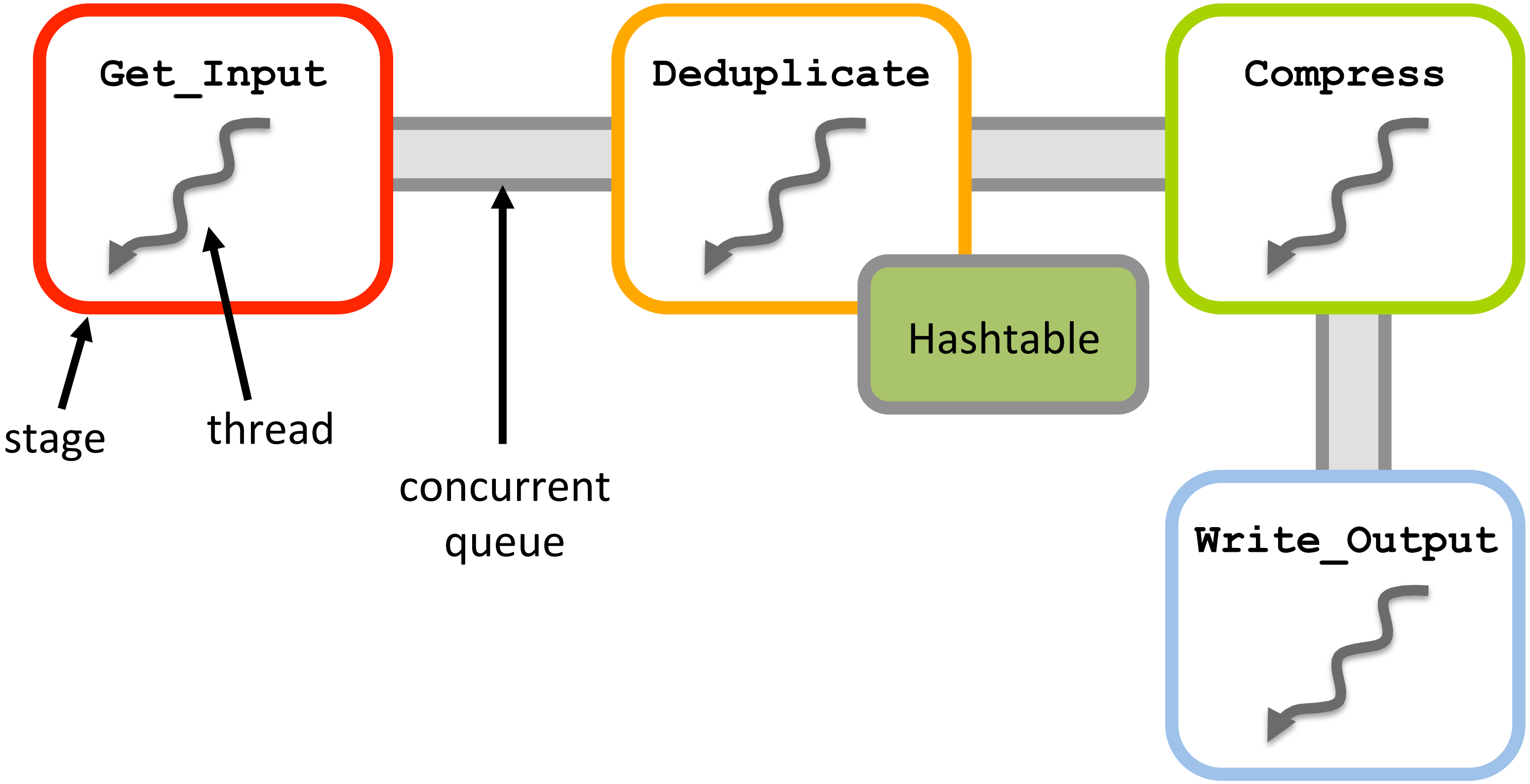
1. Assign threads to stages.

2. Threads communicate via concurrent queues.

(The programmer writes the scheduling code.)

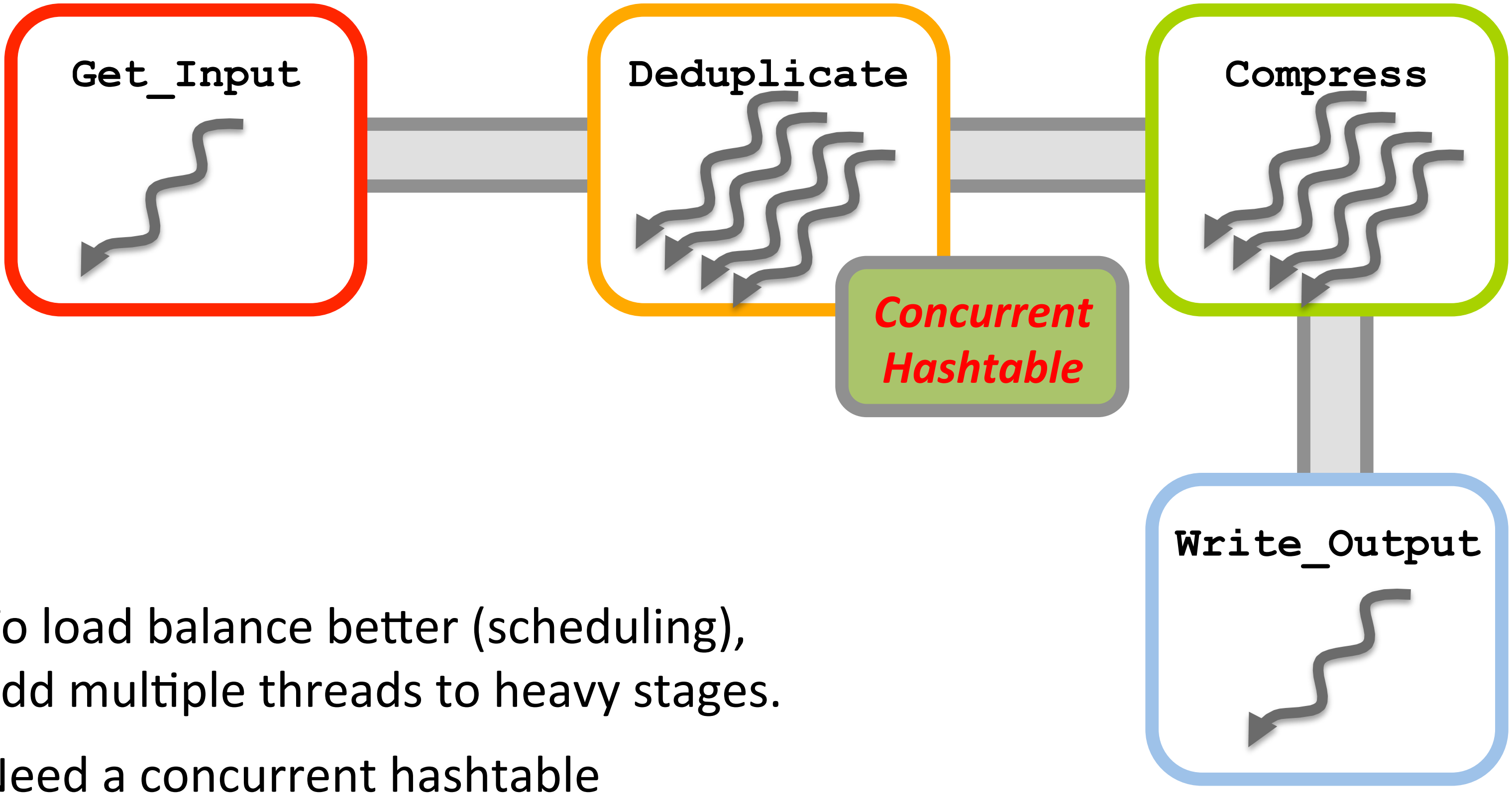
Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.



Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.

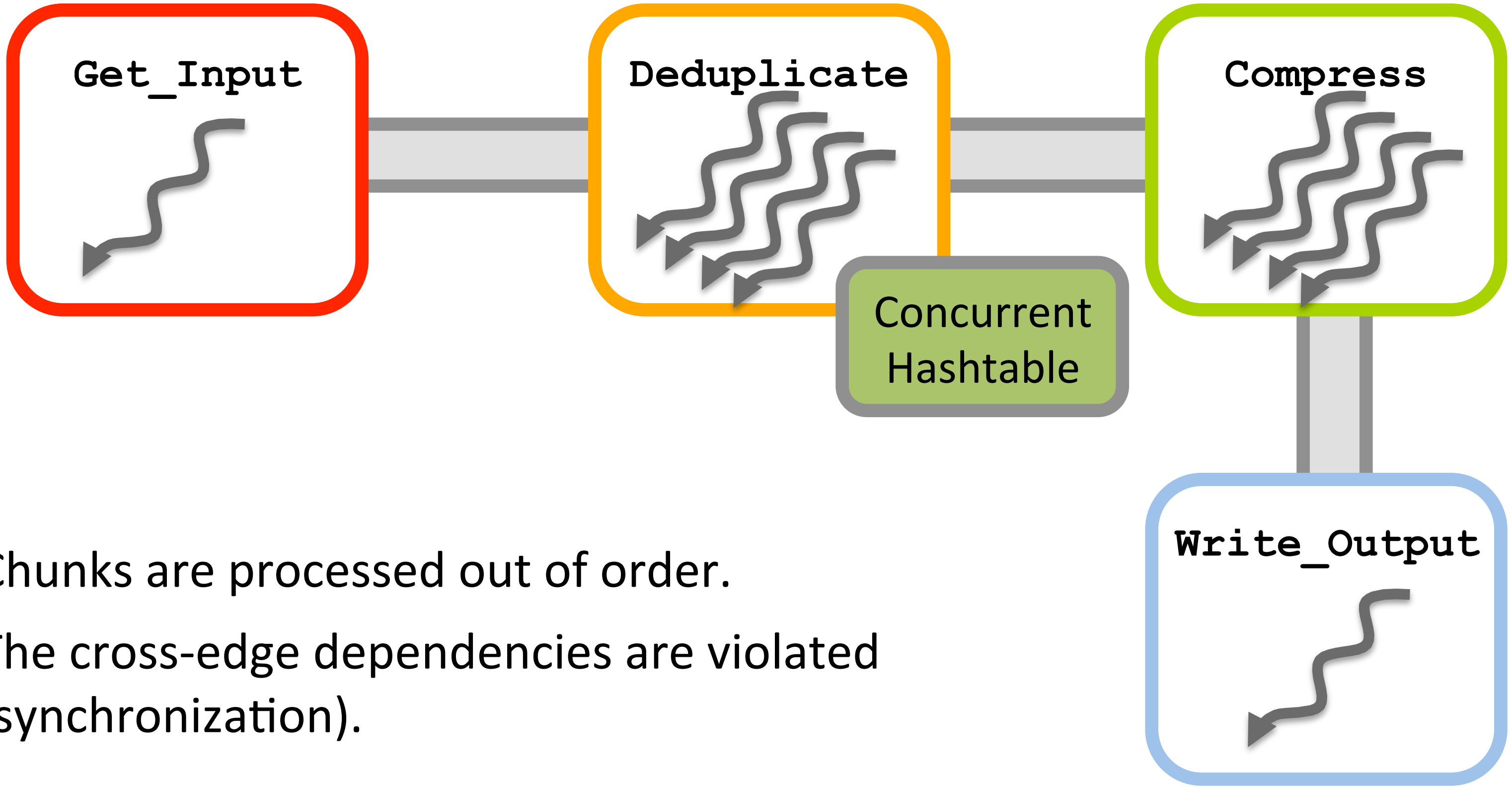


To load balance better (scheduling), add multiple threads to heavy stages.

Need a concurrent hashtable (synchronization).

Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.

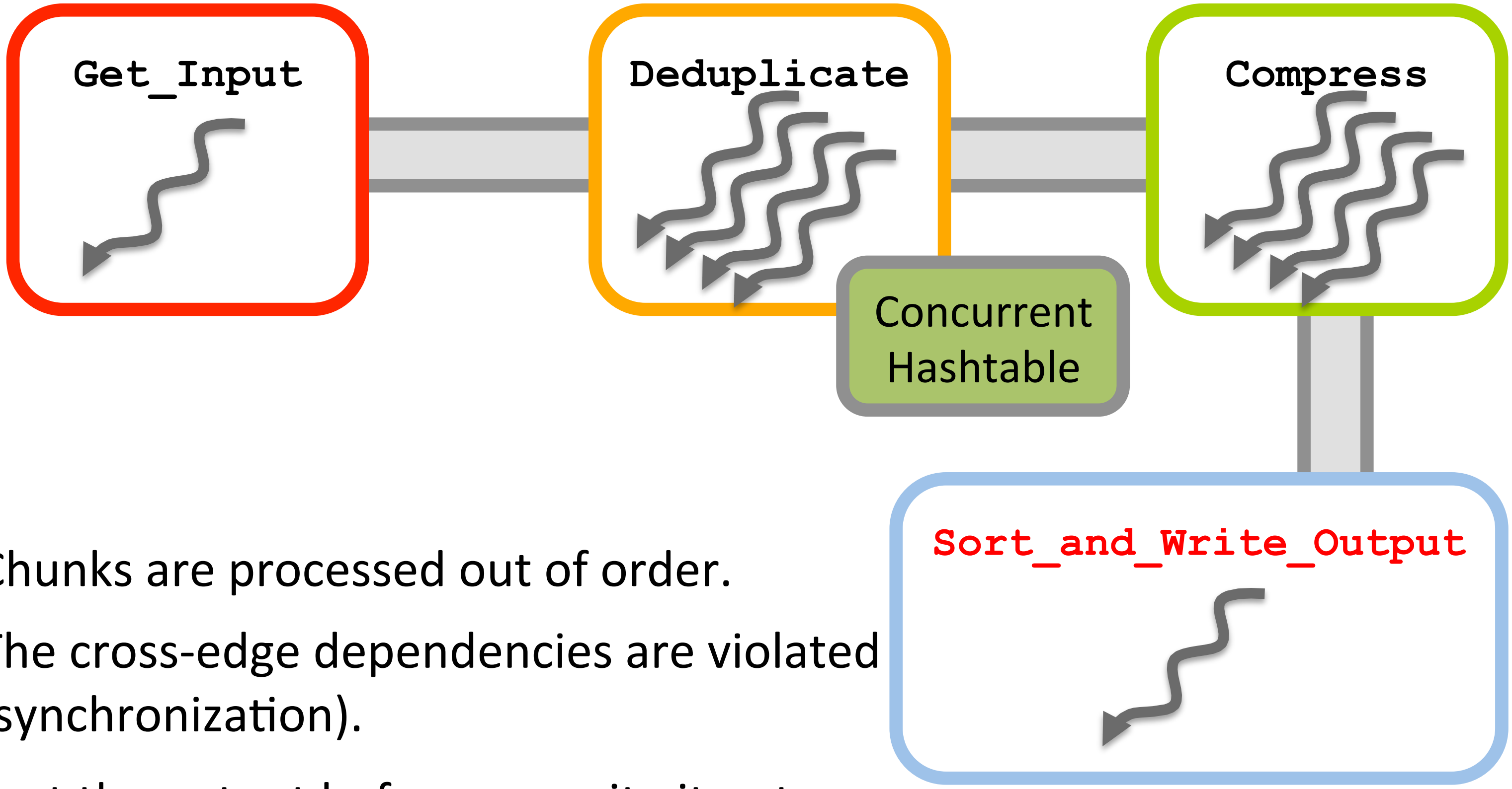


Chunks are processed out of order.

The cross-edge dependencies are violated (synchronization).

Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.



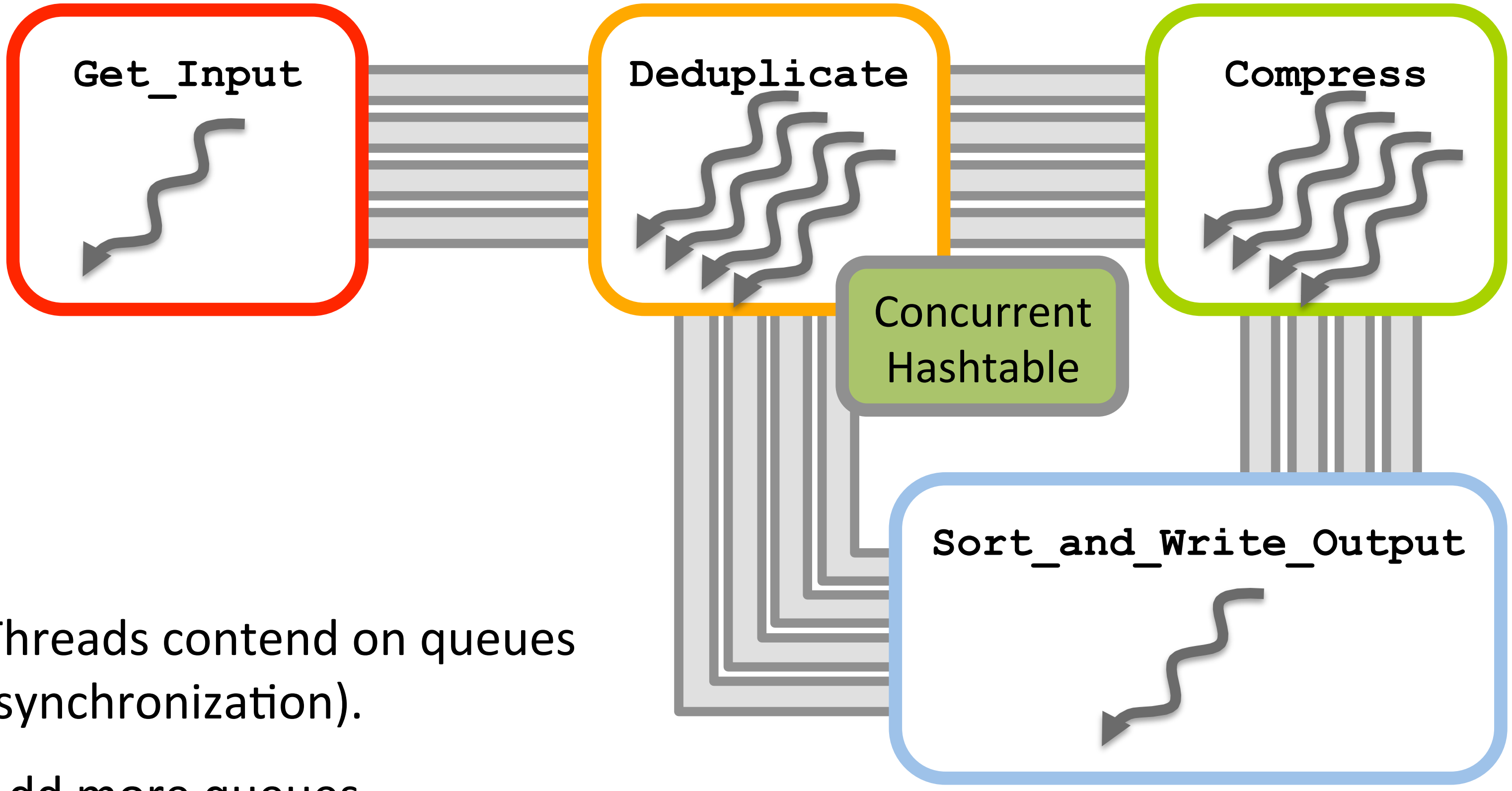
Chunks are processed out of order.

The cross-edge dependencies are violated (synchronization).

Sort the output before we write it out.

Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.

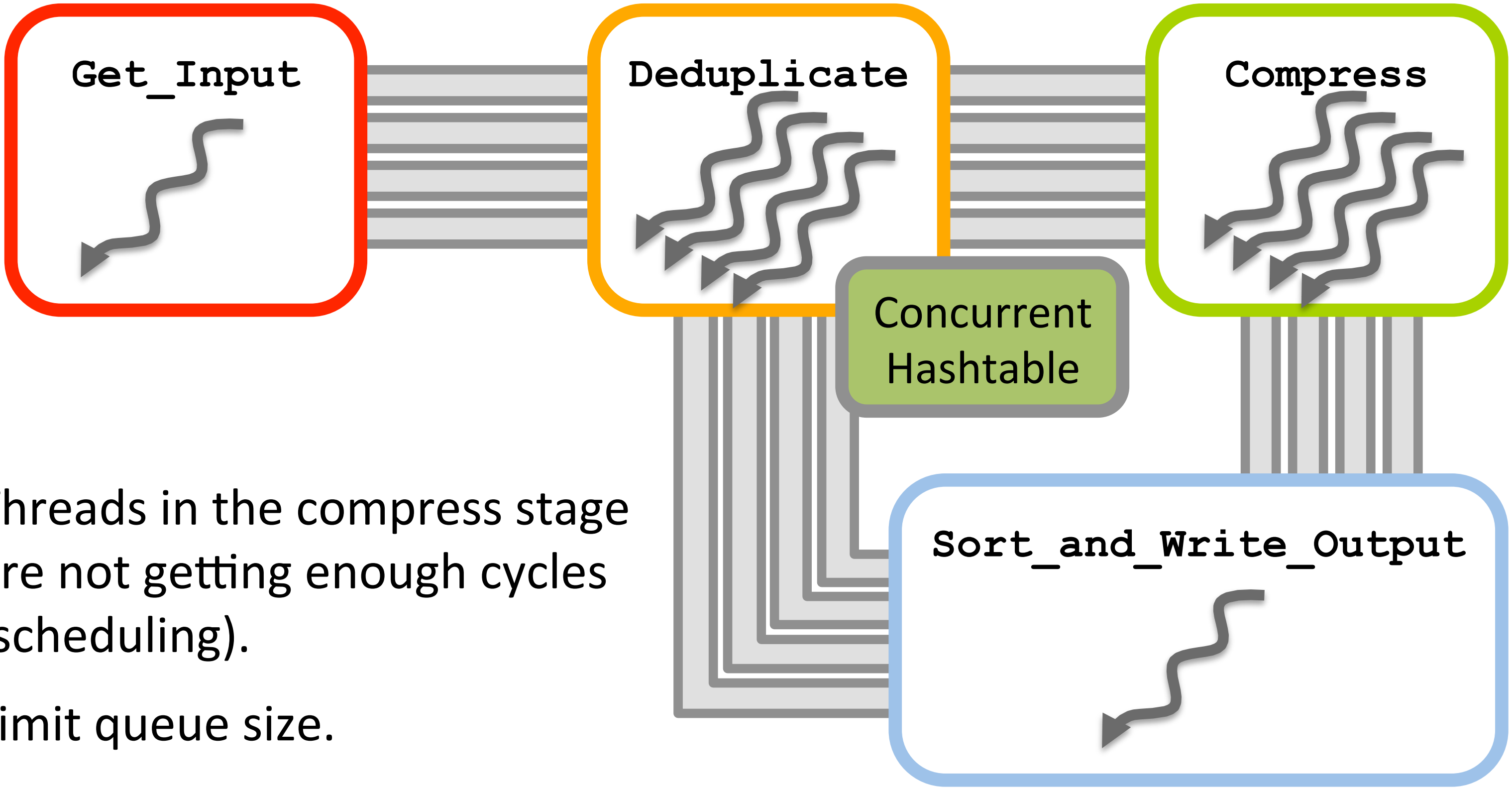


Threads contend on queues (synchronization).

Add more queues.

Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.



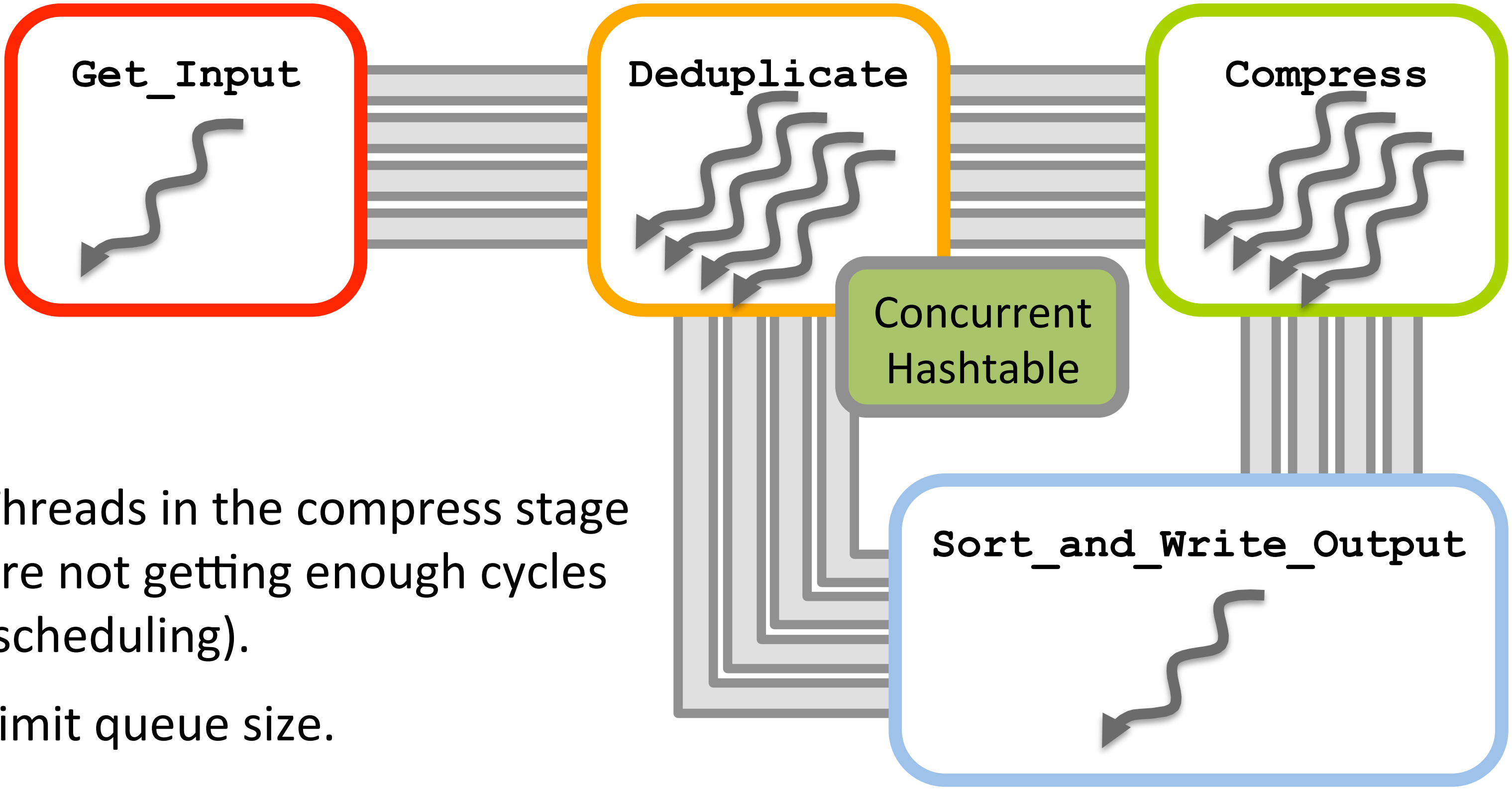
Threads in the compress stage are not getting enough cycles (scheduling).

Limit queue size.

Deadlock!

Parallelizing Dedup with Pthreads*

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.



Threads in the compress stage are not getting enough cycles (scheduling).

Limit queue size.

Deadlock!

* Based on the parallel implementation in PARSEC [BKS08].

```
config_t * conf;
struct hashtable *cache;

static unsigned int hash_from_key_fn( void *k ) {
  return ((unsigned int *)k)[0];
}
static int keys_equal_fn ( void *key1, void *key2 ) {
    return (memcmp(key1, key2, SHA1_LEN) == 0);
}
queue_t *deduplicate_que, *refine_que, *reorder_que,
            *compress_que;

struct thread_args {
  int tid;
  int nqueues;
  int fd_in;
  int fd_out;
  struct {
     void *buffer;
     size_t size;
  } input_file;
};

void Encode(config_t * _conf) {
  struct stat filestat;
  int32 fd_in, fd_out;
  conf = _conf;
  cache = hashtable_create(65536,
                                      keys_equal_fn, FALSE);
  if(cache == NULL) {
    printf("ERROR: Out of memory\n");
    exit(1);
  }
  if(stat(conf->infile, &filestat) < 0)
    EXIT_TRACE("stat() %s failed: %s\n", conf->infile,
                     strerror(errno));
  if(!S_ISREG(filestat.st_mode))
    EXIT_TRACE("not a normal file: %s\n", conf->infile);
  struct thread_args data_process_args;
  int i;
  const int nqueues =
      (conf->nthreads / MAX_THREADS_PER_QUEUE) +
      ((conf->nthreads % MAX_THREADS_PER_QUEUE !=
        0) ? 1 : 0);
  deduplicate_que = malloc(sizeof(queue_t) * nqueues);
  refine_que = malloc(sizeof(queue_t) * nqueues);
  reorder_que = malloc(sizeof(queue_t) * nqueues);
  compress_que = malloc(sizeof(queue_t) * nqueues);
```

```
if( (deduplicate_que == NULL) ||
    (refine_que == NULL) ||
    (reorder_que == NULL) || (compress_que == NULL)) {
  printf("Out of memory\n");
  exit(1);
}
int threads_per_queue;
int throttle = QUEUE_SIZE;
if( conf->throttle != -1 ) {
  throttle = (int)(ceil(conf->throttle / nqueues));
}

conf->throttle = throttle;
for(i=0; i<nqueues; i++) {
  if (i < nqueues -1 ||
      conf->nthreads %MAX_THREADS_PER_QUEUE == 0) {
    threads_per_queue = MAX_THREADS_PER_QUEUE;
  } else {
    threads_per_queue = conf->nthreads

pthread_t threads_anchor[MAX_THREADS],
             threads_chunk[MAX_THREADS],
             threads_compress[MAX_THREADS],
             threads_send, threads_process;
data_process_args.tid = 0;
data_process_args.nqueues = nqueues;
data_process_args.fd_in = fd_in;

pthread_create(&threads_process, NULL, Fragment,
                     &data_process_args);
struct thread_args anchor_thread_args[conf->nthreads];
for (i = 0; i < conf->nthreads; i ++) {
  anchor_thread_args[i].tid = i;
  pthread_create(&threads_anchor[i], NULL,
                       FragmentRefine,
                       &anchor_thread_args[i]);
}
```

```
struct thread_args chunk_thread_args[conf->nthreads];
for (i = 0; i < conf->nthreads; i ++) {
  chunk_thread_args[i].tid = i;
  pthread_create(&threads_chunk[i], NULL, Deduplicate,
                       &chunk_thread_args[i]);
}
struct thread_args compress_thread_args[conf->nthreads];
for (i = 0; i < conf->nthreads; i ++) {
  compress_thread_args[i].tid = i;
  pthread_create(&threads_compress[i], NULL, Compress,
                       &compress_thread_args[i]);
}
struct thread_args send_block_args;
send_block_args.tid = 0;
send_block_args.nqueues = nqueues;
send_block_args.fd_out = fd_out;

pthread_create(&threads_send, NULL, Reorder,
                     &send_block_args);
                      process, NULL);
                      eads; i ++)
                      _anchor[i], NULL);
                      eads; i ++)
                      _chunk[i], NULL);
                      eads; i ++)
                      _compress[i], NULL);
                      end, NULL);

for(i=0; i<nqueues; i++) {
  queue_destroy(&deduplicate_que[i]);
  queue_destroy(&refine_que[i]);
  queue_destroy(&reorder_que[i]);
  queue_destroy(&compress_que[i]);
}
free(deduplicate_que);
free(refine_que);
free(reorder_que);
free(compress_que);

if(conf->infile != NULL)
  close(fd_in);
close(fd_out);
ret = mbuffer_system_destroy();
assert(ret == 0);
hashtable_destroy(cache, TRUE);
}
```

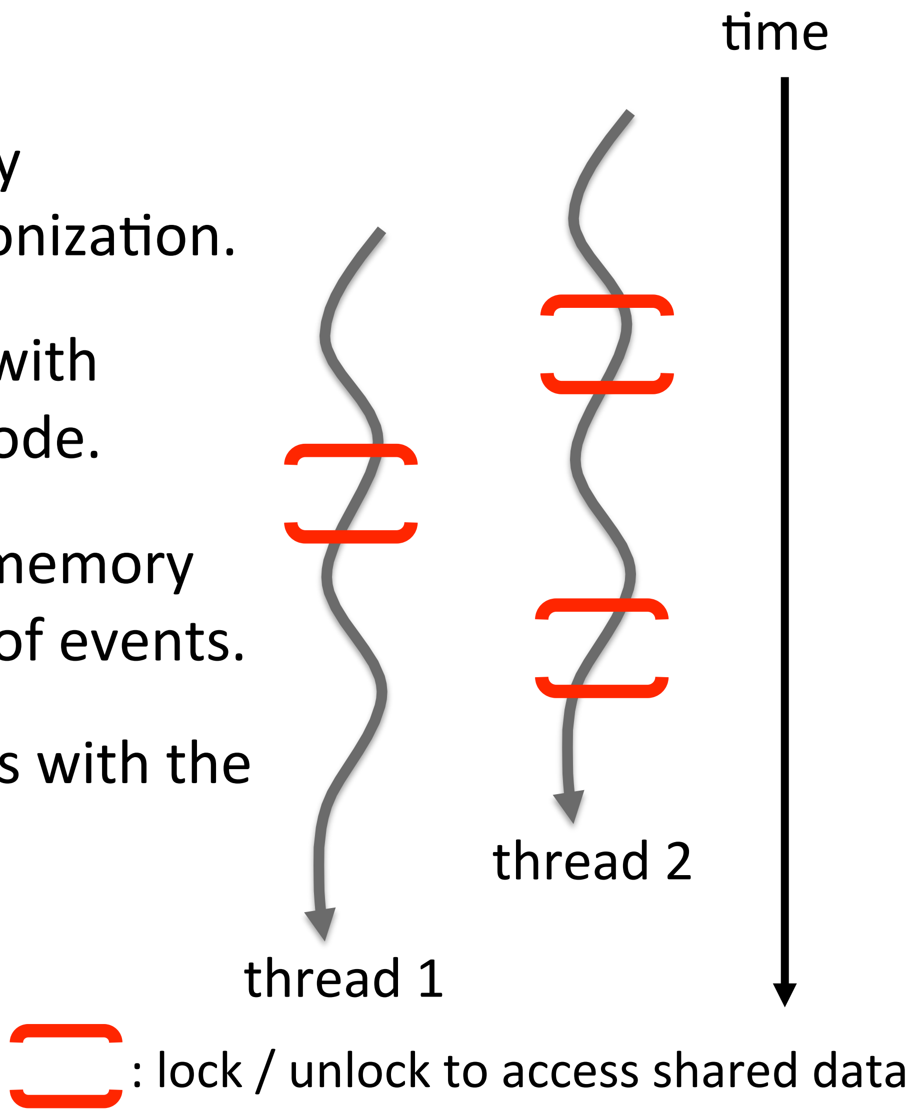
The programmer must manually handle scheduling and synchronization.

The setup code for parallel execution using pthreads.

Problems with Persistent Threads

The programmer must manually manage scheduling and synchronization.

- Scheduling logic intermixed with program logic ⇒ spaghetti code.
- Threads interact via shared memory ⇒ no well-defined ordering of events.
- The scheduling logic interacts with the need for synchronization.



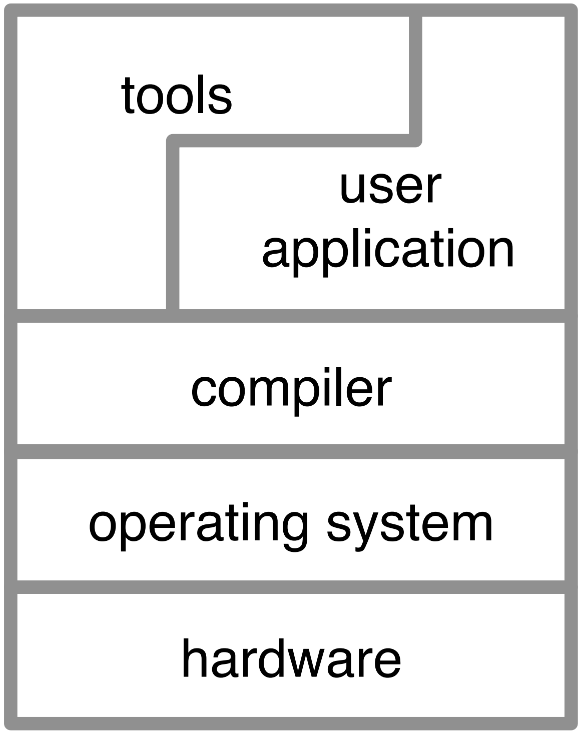
▭ : lock / unlock to access shared data

Structured Parallel Programming

A programming model that allows the the programmer to express the ***logical parallelism*** of the computation to using ***control constructs***.

- separates the scheduling logic from program logic;
- automates scheduling and synchronization; and
- provides a clean mental model for the programmer to reason about parallelism.

Traditional Computing Stack



tools

user application

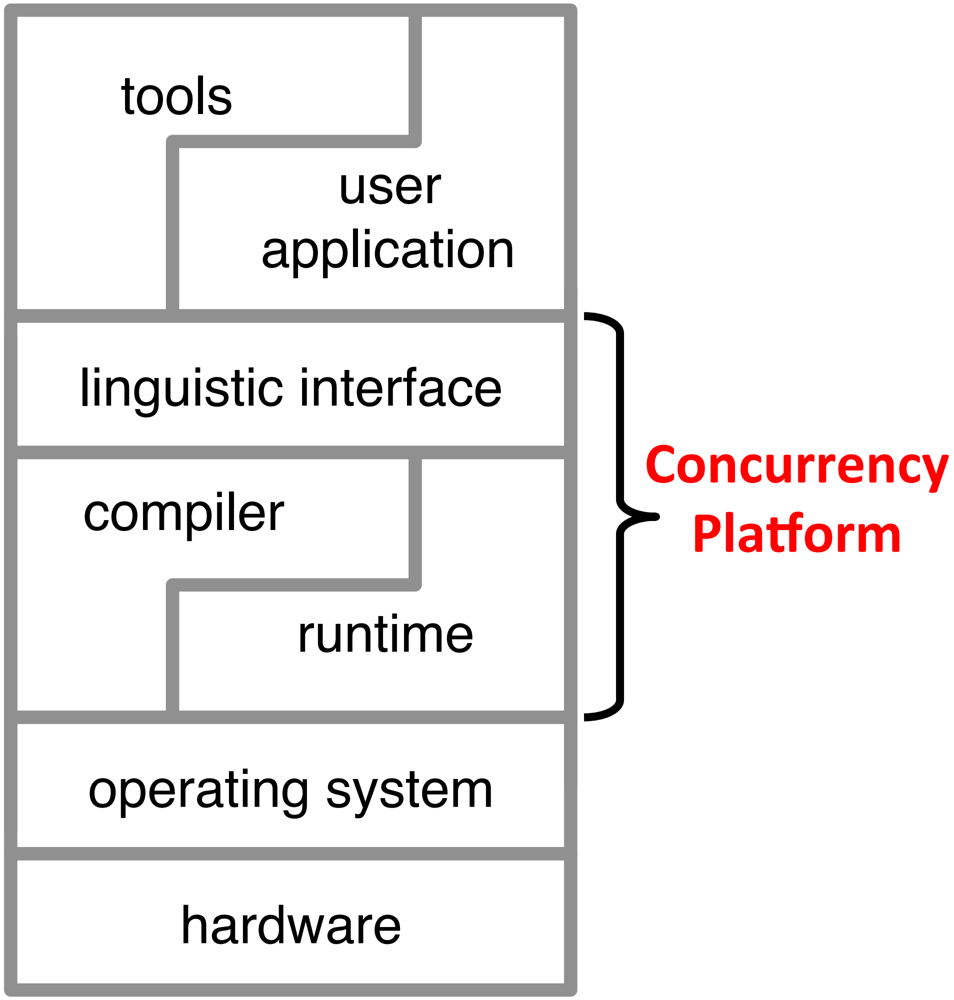
compiler

operating system

hardware

→ Provides the pthread abstraction as surrogates for cores

State of Art: Concurrency Platform



tools

user application

linguistic interface

compiler

runtime

operating system

hardware

Concurrency Platform

A concurrency platform should provide:

- an interface for specifying the ***logical parallelism*** of the computation;
- a runtime layer to automate scheduling and synchronization; and
- guarantees of performance and resource utilization competitive with hand-tuned code.

My Research

- Design language abstractions for structured parallel programming
- Develop efficient system support for these language abstractions
- Design tool support for debugging and performance engineering programs written in theses high-level language abstractions

Cilk-P's Linguistic Support for Pipeline Parallelism

An instance of structured parallel programming

Encode Parallelism of Dedup

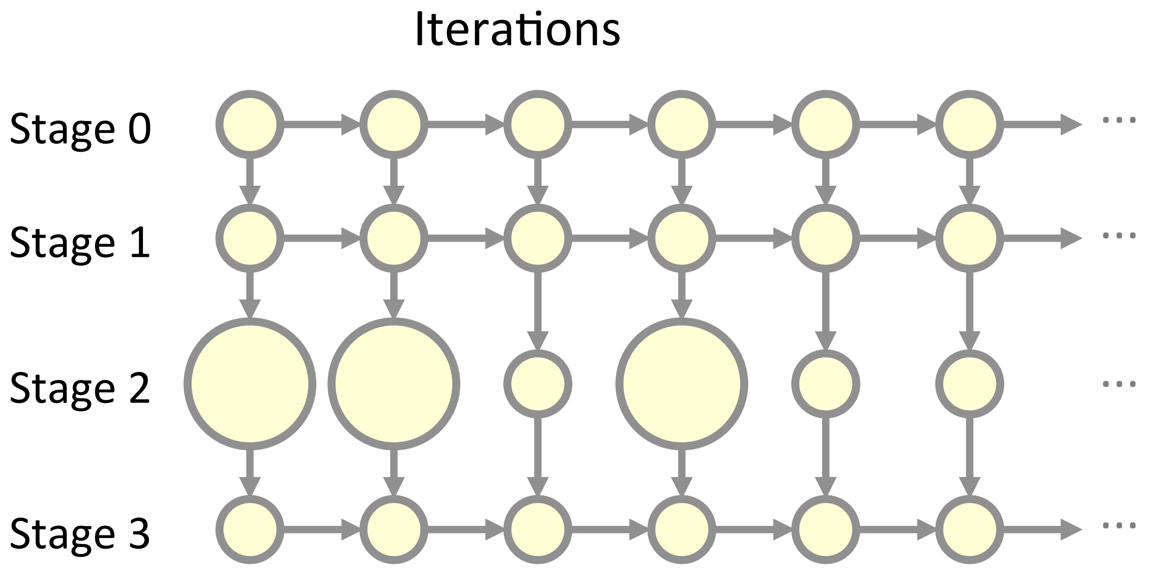
```
while(!done) {
   chunk_t *chunk = get_next_chunk();
   if(chunk == NULL) { done = true; }
   else {

      chunk->is_dup = deduplicate(chunk);

      if(!chunk->is_dup) compress(chunk);

      write_to_file(fd_out, chunk);
   }
}
```

Stage 0

Stage 1

Stage 2

Stage 3

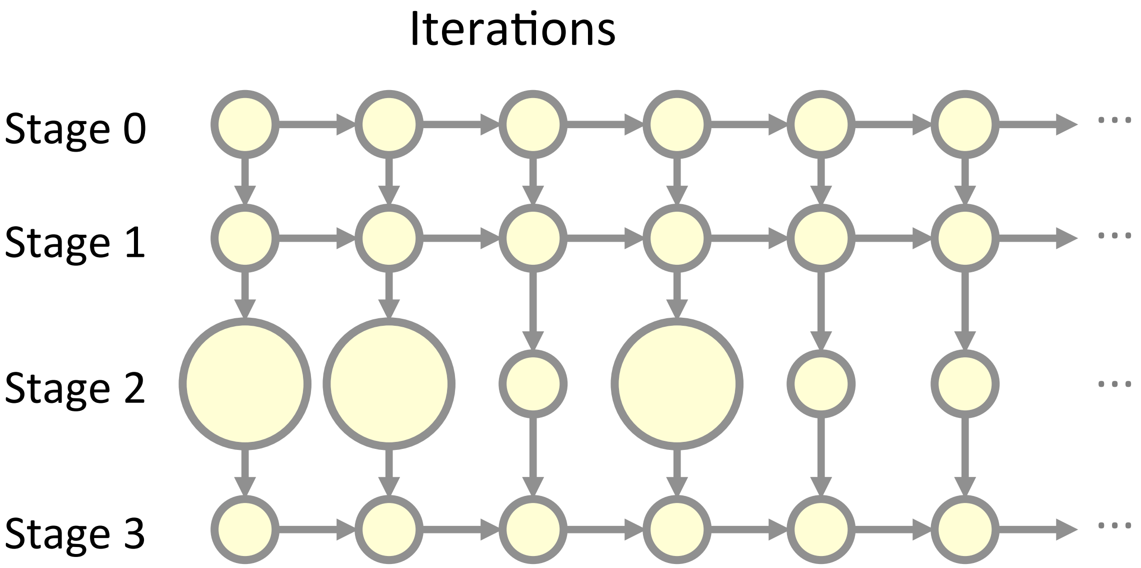


1. Pipeline the loop.

→ : **Cross Edge**

Encode Parallelism of Dedup

```
pipe_while(!done) {
   chunk_t *chunk = get_next_chunk();
   if(chunk == NULL) { done = true; }             Stage 0
   else {

      chunk->is_dup = deduplicate(chunk);         Stage 1

      if(!chunk->is_dup) compress(chunk);         Stage 2

      write_to_file(fd_out, chunk);               Stage 3
   }
}
```



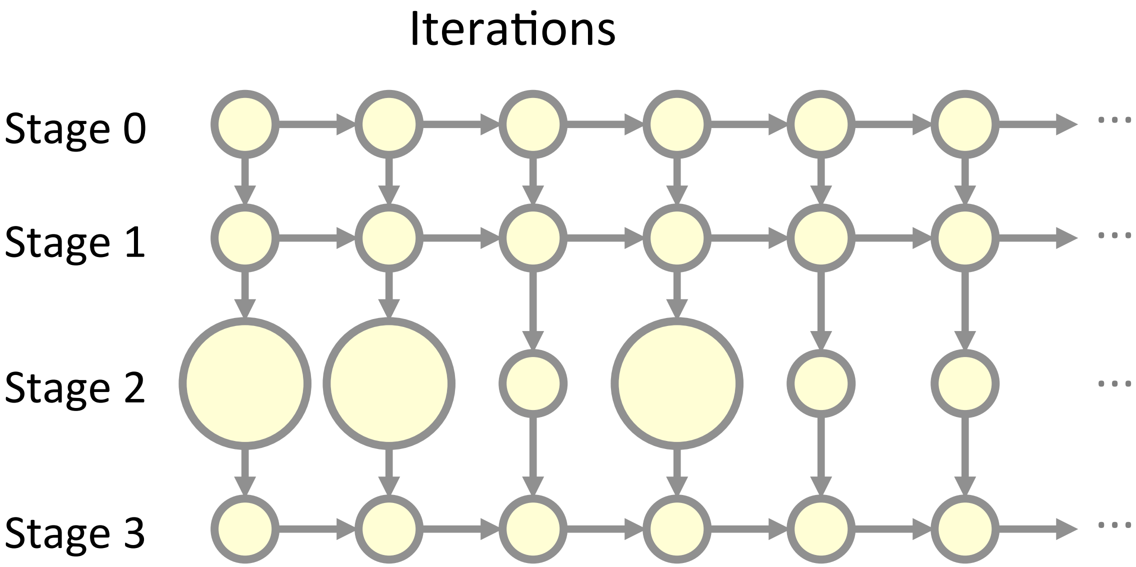
1. Pipeline the loop.

2. Denote stages.

→ : **Cross Edge**

Encode Parallelism of Dedup

```
pipe_while(!done) {
   chunk_t *chunk = get_next_chunk();
   if(chunk == NULL) { done = true; }                Stage 0
   else {
      pipe_stage;
      chunk->is_dup = deduplicate(chunk);            Stage 1
      pipe_stage;
      if(!chunk->is_dup) compress(chunk);            Stage 2
      pipe_stage;
      write_to_file(fd_out, chunk);                  Stage 3
   }
}
```



1. Pipeline the loop.
2. Denote stages.
3. Enforce cross-edge dependencies

→ : **Cross Edge**

Encode Parallelism of Dedup

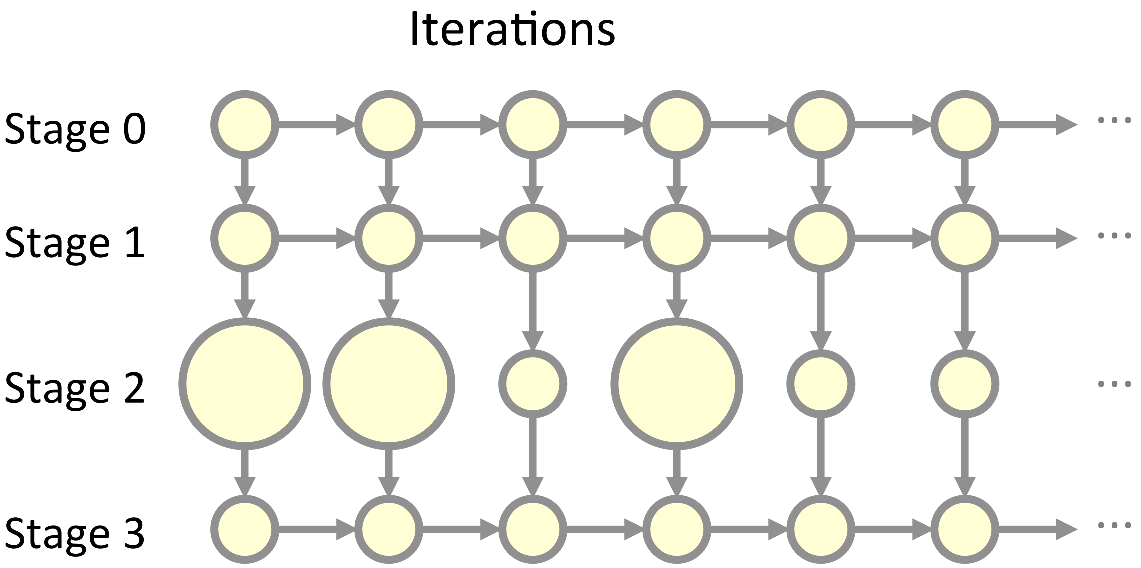
```
pipe_while(!done) {
   chunk_t *chunk = get_next_chunk();
   if(chunk == NULL) { done = true; }
   else {
      pipe_stage_wait();
      chunk->is_dup = deduplicate(chunk);
      pipe_stage();
      if(!chunk->is_dup) compress(chunk);
      pipe_stage_wait();
      write_to_file(fd_out, chunk);
   }
}
```

Stage 0

Stage 1

Stage 2

Stage 3



1. Pipeline the loop.
2. Denote stages.
3. Enforce cross-edge dependencies

→ : **Cross Edge**

The Pipeline Linguistics in Cilk-P

```
int fd_out = open_output_f
bool done = false;
pipe_while(!done) {
   chunk_t *chunk = get_next_chunk();
   if(chunk == NULL) { done = true; }
   else {
      pipe_stage_wait(1);
      chunk->is_dup = deduplicate(chunk);
      pipe_stage(2);
      if(!chunk->is_dup) compress(chunk);
      pipe_stage_wait(3);
      write_to_file(fd_out, chunk);
   }
}
```

Loop iterations may execute in parallel in a pipelined fashion, where stage 0 executes serially.

End the current stage, advance to stage 1, and wait for the previous iteration to finish stage 1.

End the current stage and advance to stage 2.

The Pipeline Linguistics in Cilk-P

```
int fd_out = open_output_file();
bool done = false;
pipe_while(!done) {
  chunk_t *chunk = get_next_chunk();
  if(chunk == NULL) { done = true; }
  else {
    pipe_stage_wait(1);
    chunk->is_dup = deduplicate(chunk);
    pipe_stage(2);
    if(!chunk->is_dup) compress(chunk);
    pipe_stage_wait(3);
    write_to_file(fd_out, chunk);
  }
}
```

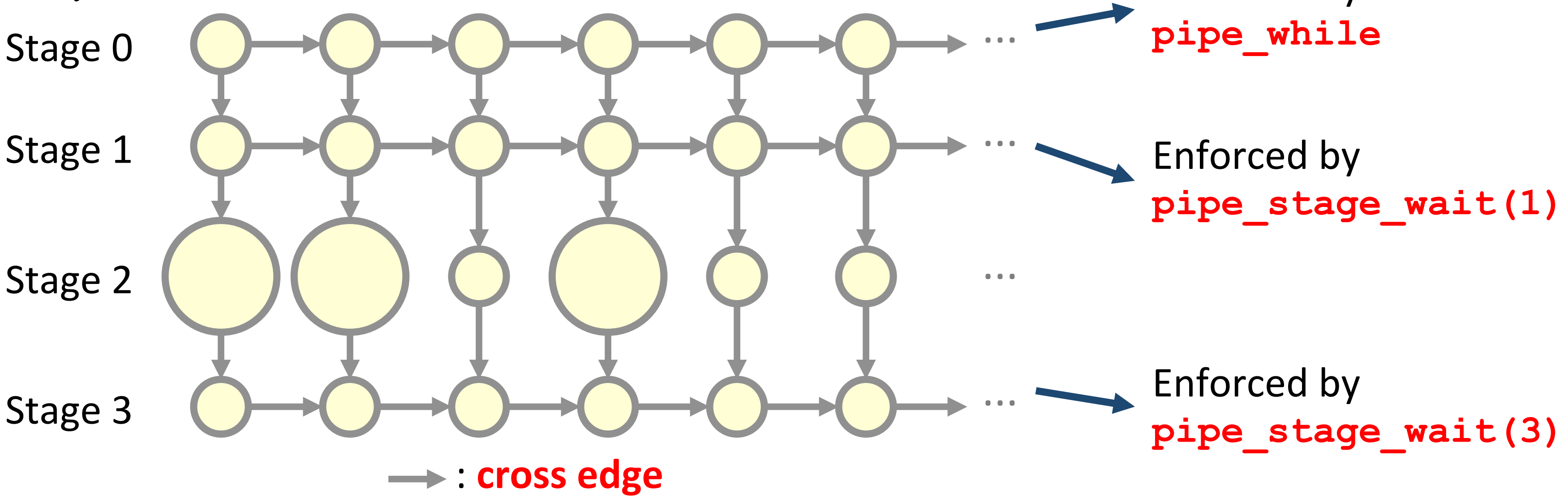
These keywords have **serial semantics** [FLR98].

The Pipeline Linguistics in Cilk-P

```
pipe_while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }
    else {
        pipe_stage_wait(1);
        chunk->is_dup = deduplicate(chunk);
        pipe_stage(2);
        if(!chunk->is_dup) compress(chunk);
        pipe_stage_wait(3);
        write_to_file(fd_out, chunk);
    }
}
```

These keywords allow the user to express the ***logical parallelism***.

Iterations



Enforced by **`pipe_while`**

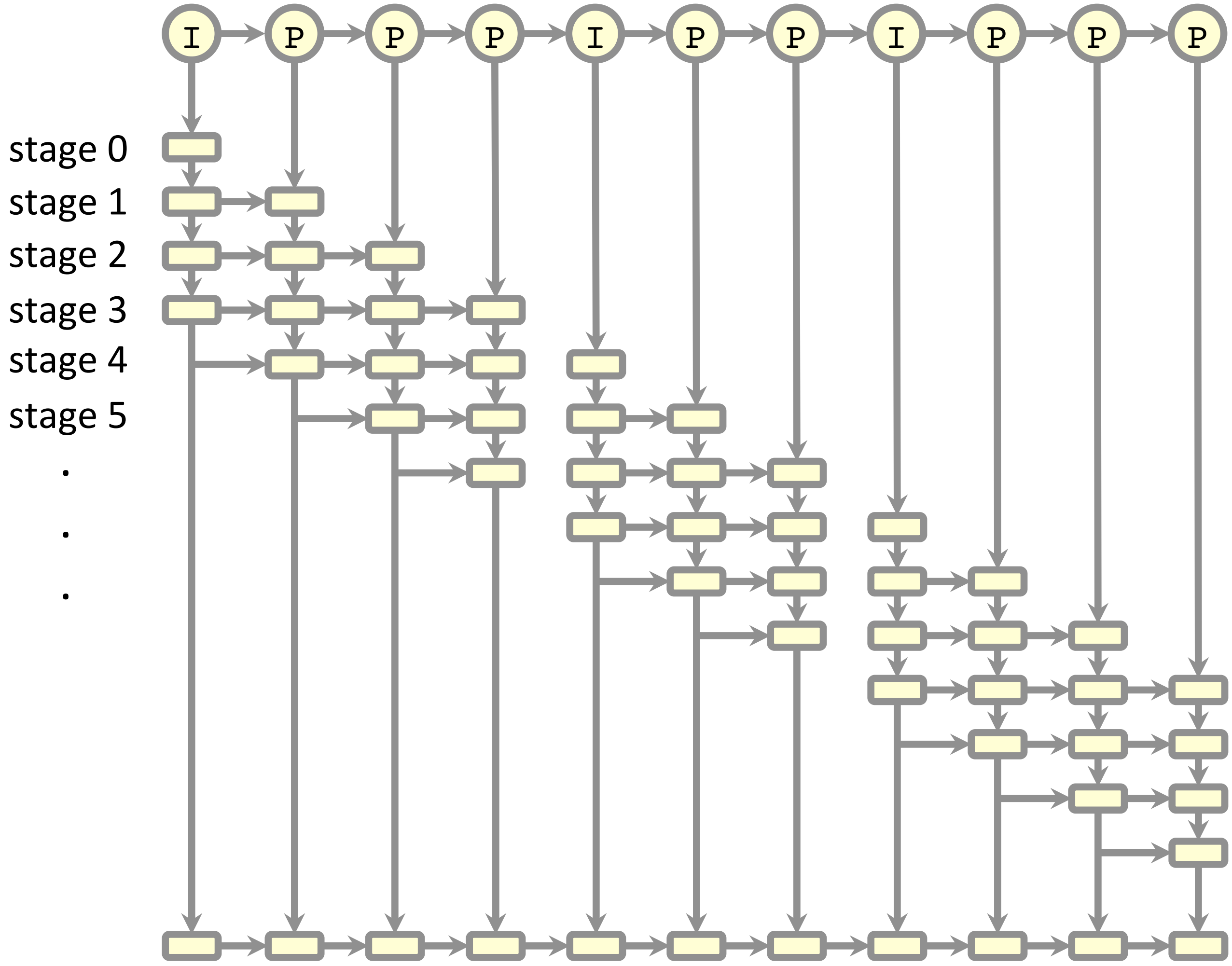
Enforced by **`pipe_stage_wait(1)`**

Enforced by **`pipe_stage_wait(3)`**

→ : **cross edge**

On-the-Fly Pipelining of X264

Cilk-P supports **on-the-fly** pipeline parallelism, where the pipeline is ***constructed dynamically*** as the program executes.



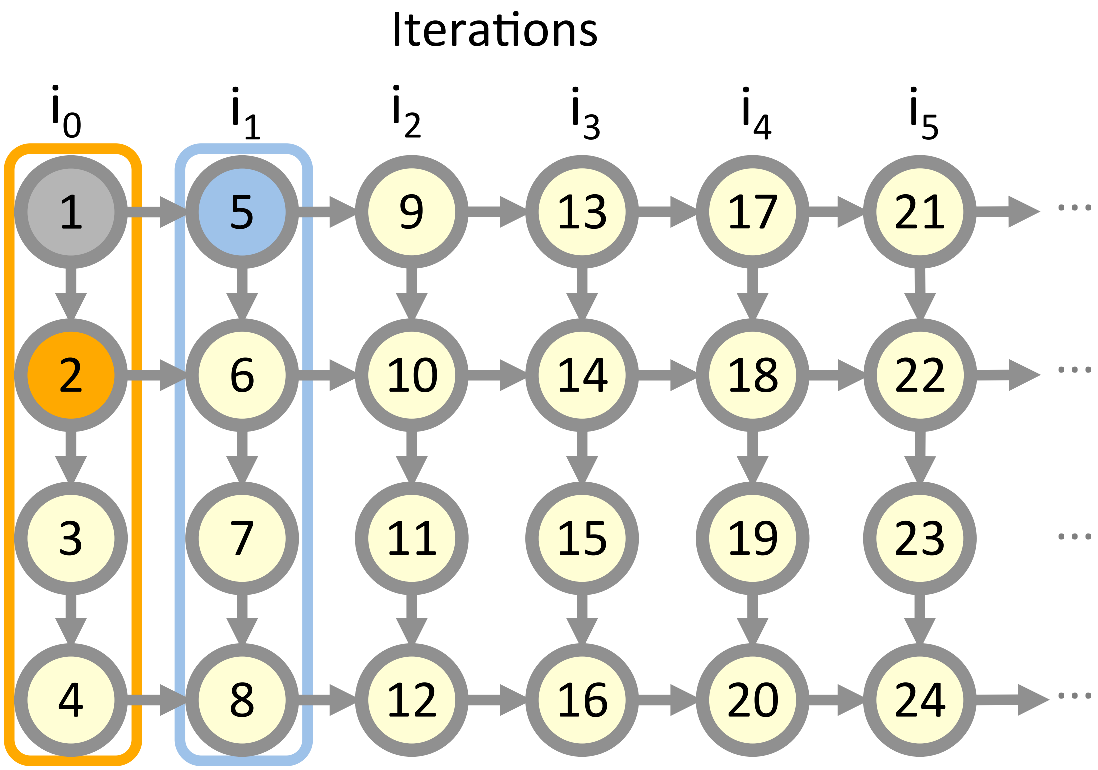
By enclosing **`pipe_stage`** and **`pipe_stage_wait`** statements within other control constructs, one can:

- skip stages;
- make cross edges data dependent; and
- vary the number of stages across iterations.

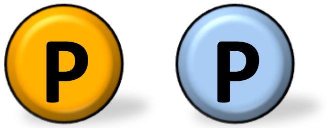
Piper: Cilk-P’s Provably-Efficient Scheduler

Elegant linguistic interface is only half the battle.

PIPER: A Work-Stealing Scheduler



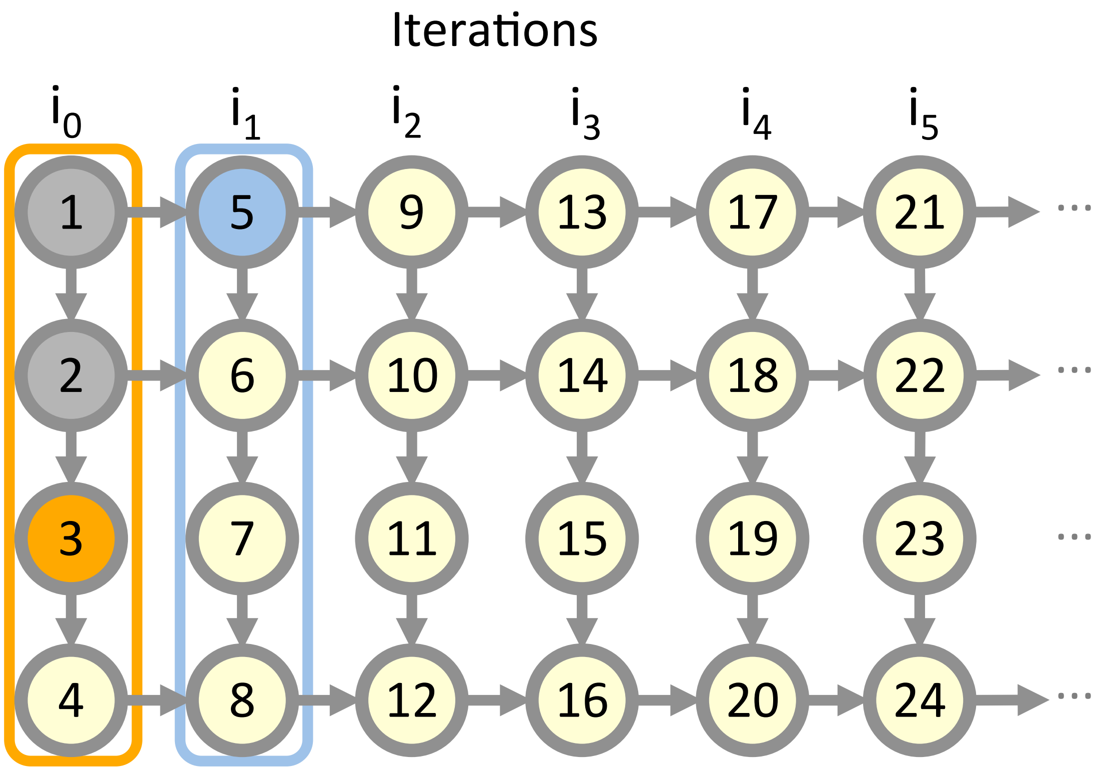
Each iteration
is a "task."



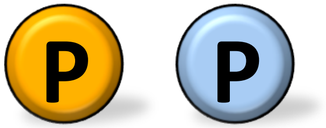
A **worker** (surrogate for a processor)
by default follows the ***serial execution order***.

● : done ○ : not done

PIPER: A Work-Stealing Scheduler



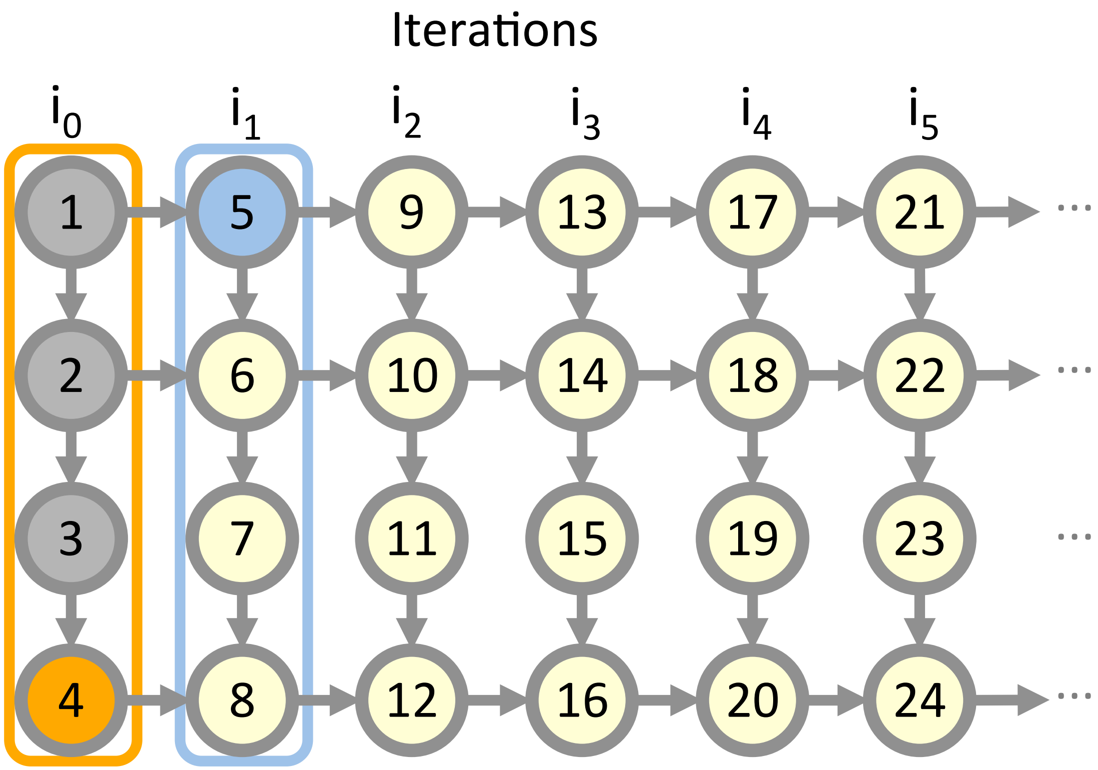
Each iteration
is a "task."



A **worker** (surrogate for a processor)
by default follows the ***serial execution order***.

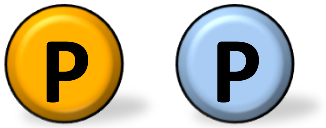
○ : done ○ : not done

PIPER: A Work-Stealing Scheduler



Each iteration is a "task."

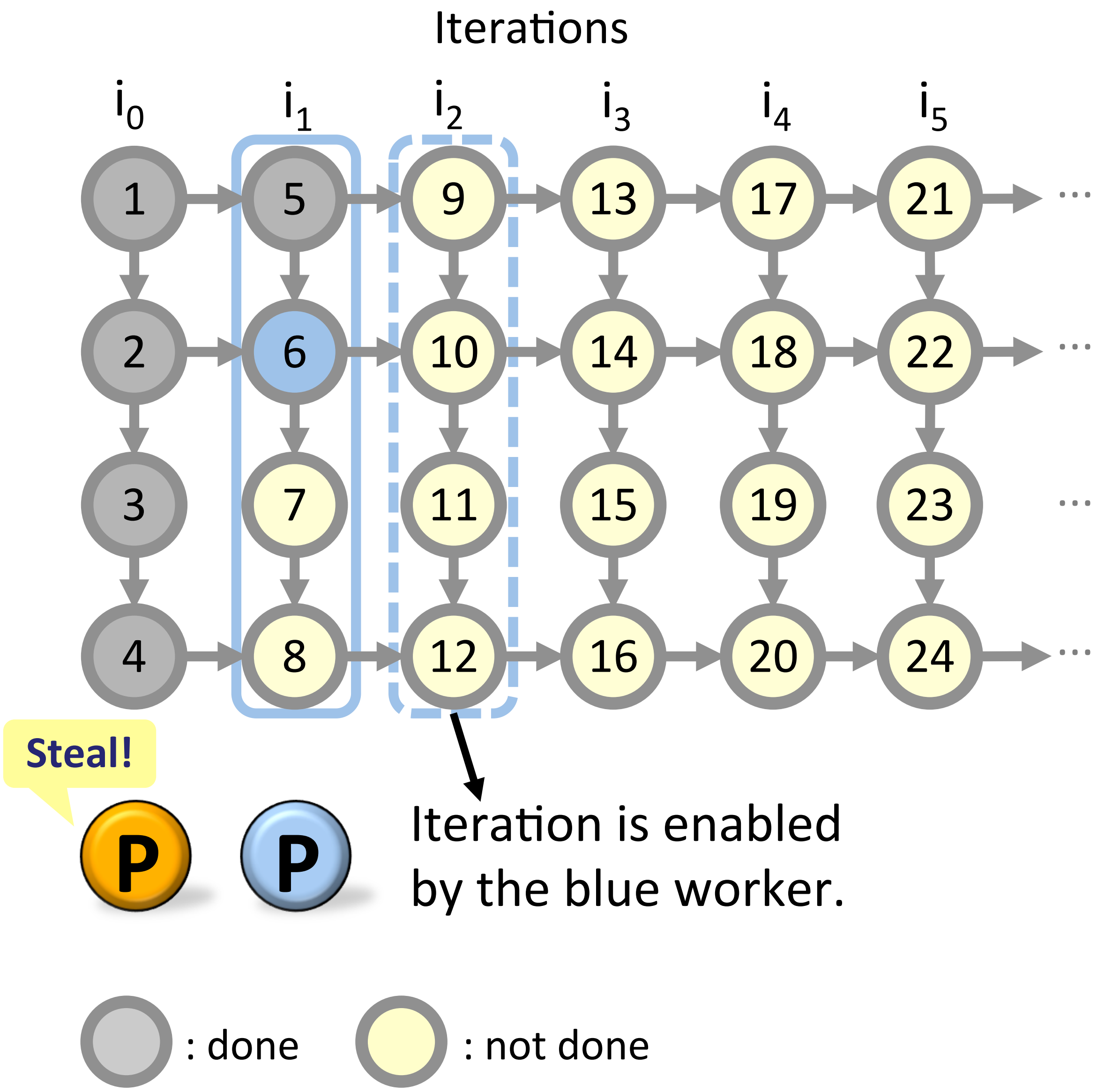
- Serial semantics; and
- Don't need queues to pass elements between stages;
- Potentially better locality.



A **worker** (surrogate for a processor) by default follows the ***serial execution order***.

● : done ○ : not done

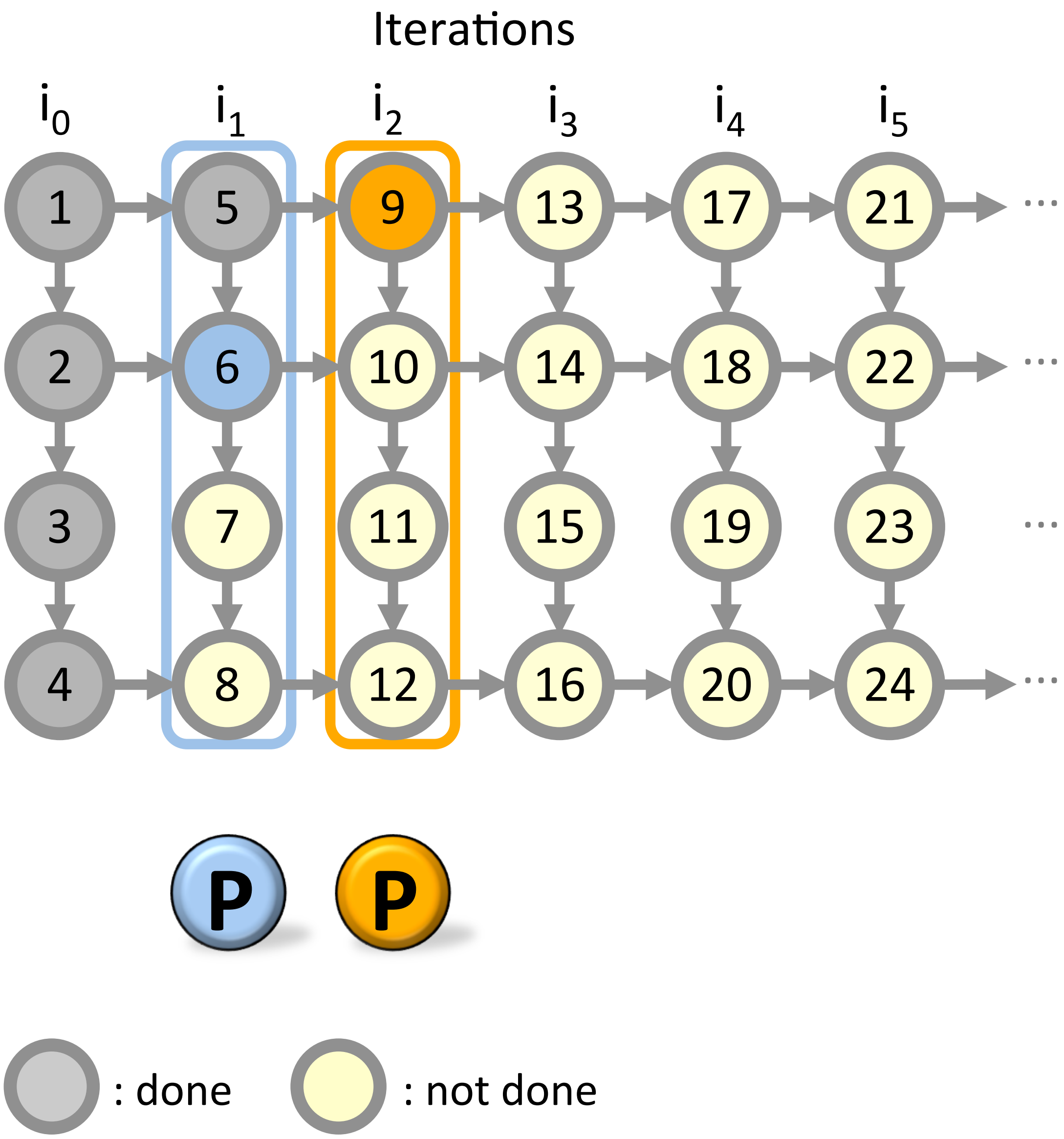
PIPER: A Work-Stealing Scheduler



Iteration is enabled by the blue worker.

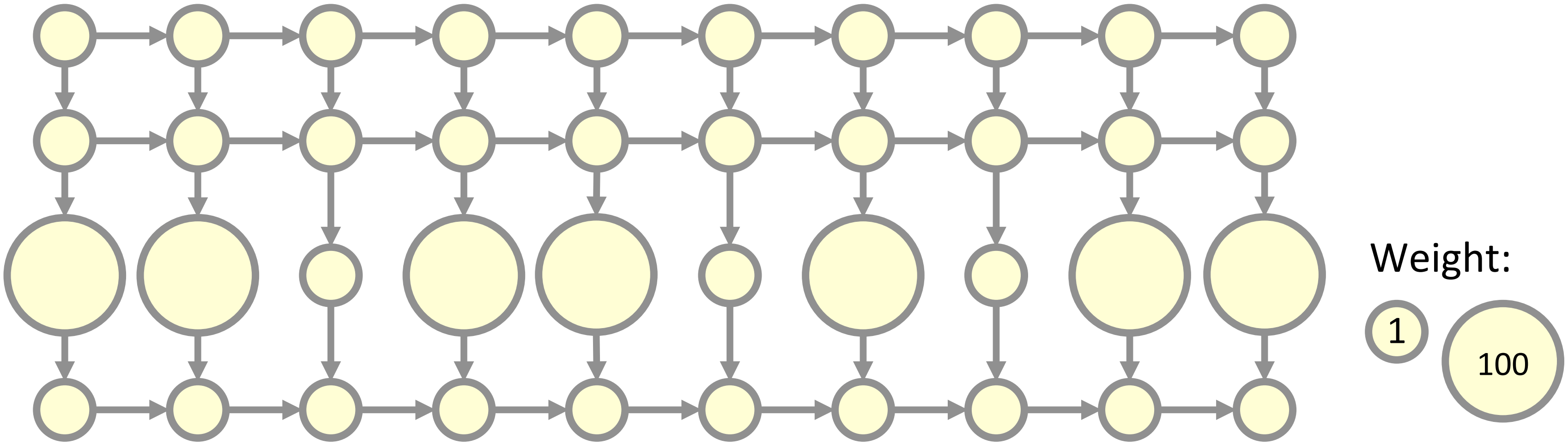
A worker ***steals*** work from a randomly selected victim when it runs out of work to do.

PIPER: A Work-Stealing Scheduler



A worker **steals** work from a randomly selected victim when it runs out of work to do.

Performance Measures [CLRS09]



Let T_P be the time it takes to execute this dag on P processors.

Work T_1 : The sum of the weights of the nodes in the dag. $T_1 = 733$

Span T_∞ : The length of a longest path in the dag. $T_\infty = 112$

Parallelism T_1 / T_∞ : The maximum possible speedup. $T_1 / T_\infty = 6.54$

Work Law : $T_P \geq T_1 / P$ **Span Law** : $T_P \geq T_\infty$

PIPER's Guarantees

Definition. T_P — execution time on P processors

T_1 — work T_∞ — span T_1 / T_∞ — parallelism

S_P — stack space on P processors

S_1 — stack space of a serial execution

K — throttling limit f — maximum frame size

D — depth of nested pipelines

- **Time bound:** Scheduling overhead

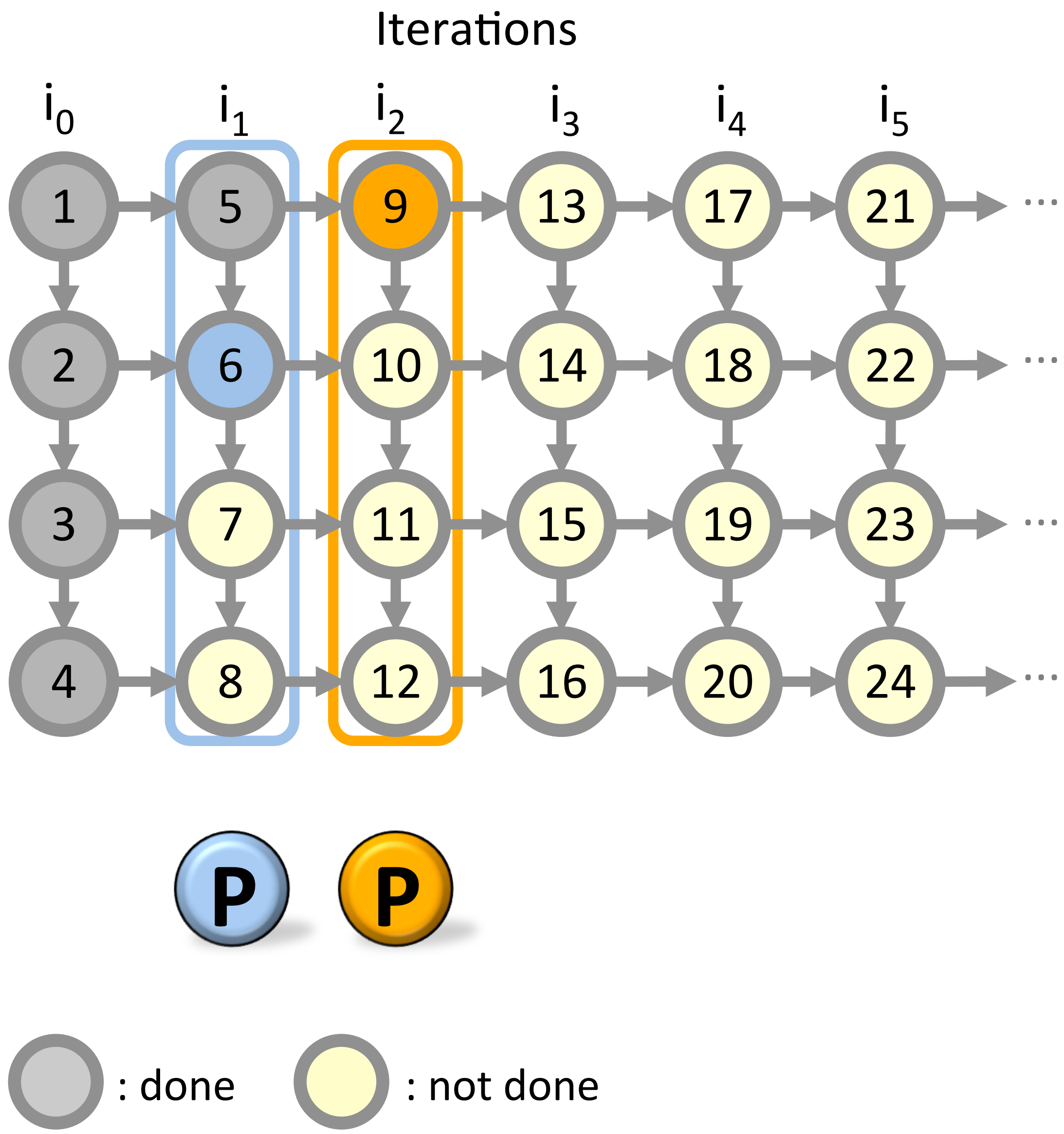
  $T_P \leq T_1 / P + O(T_\infty + \lg P)$ expected time

  $\Rightarrow$ ***linear speedup*** when $P \ll T_1 / T_\infty$ and $T_\infty > \lg P$

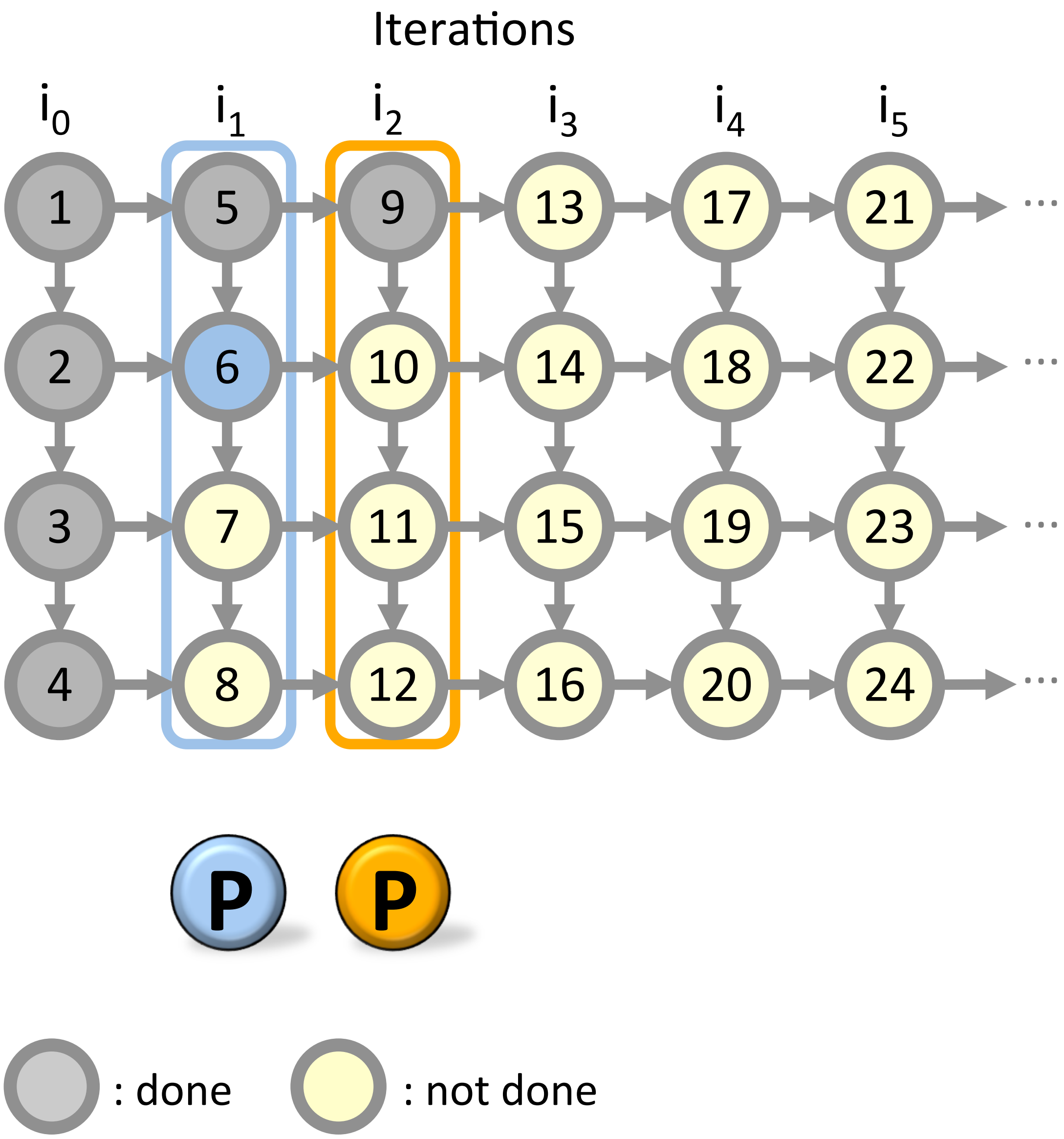
- **Space bound:**

  $S_P \leq P(S_1 + fDK)$

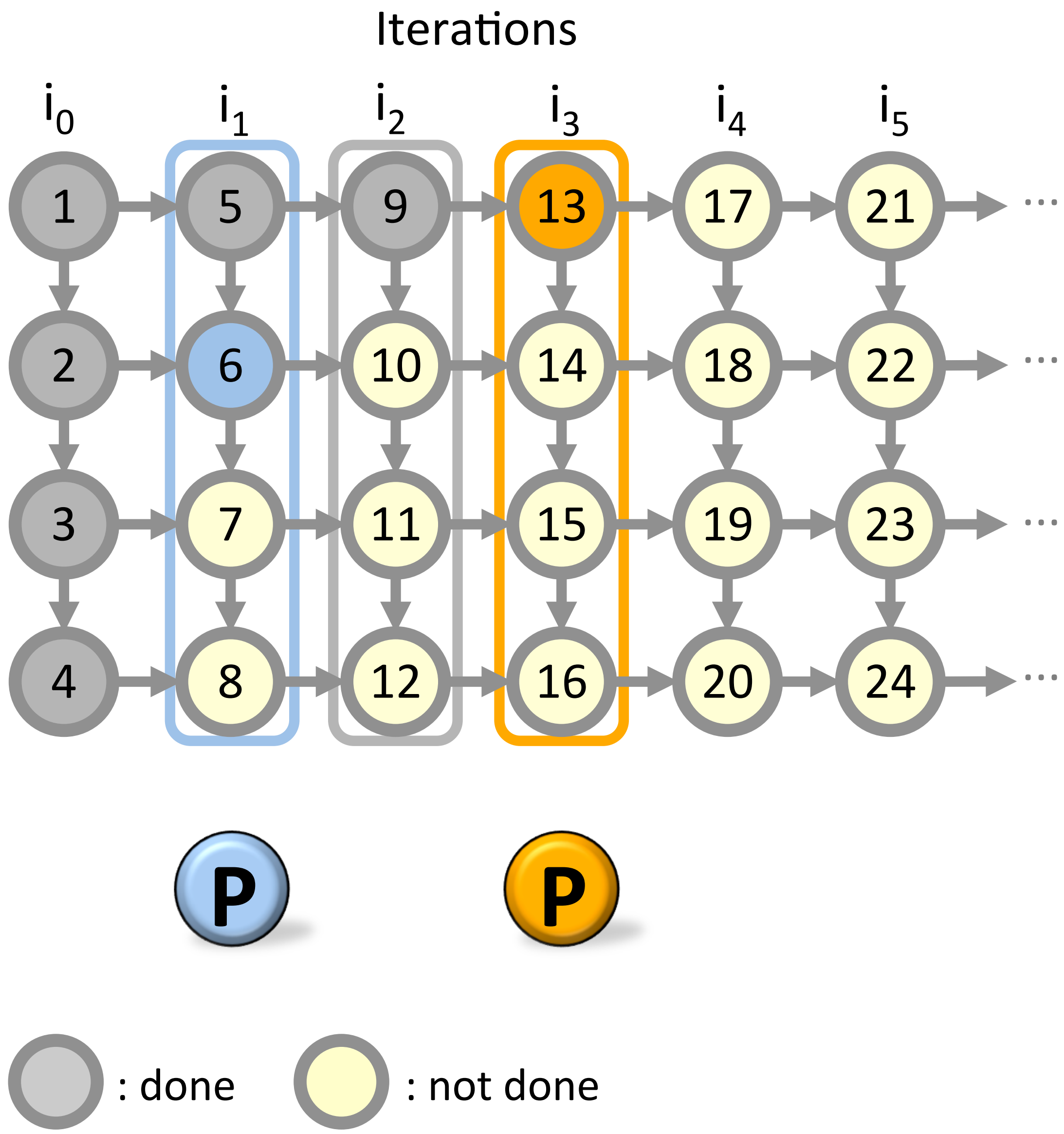
The Check-Next Overhead



The Check-Next Overhead

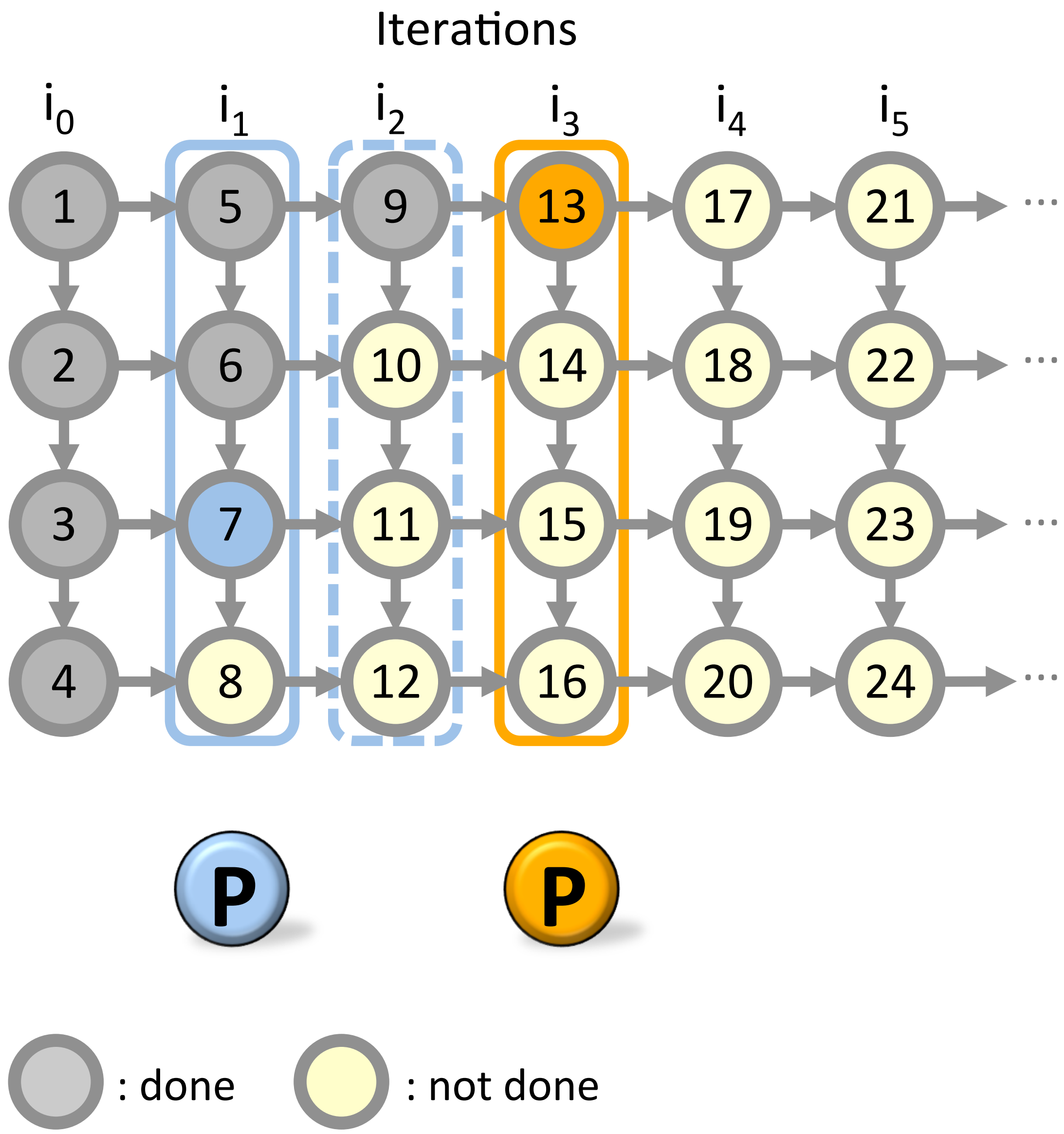


The Check-Next Overhead



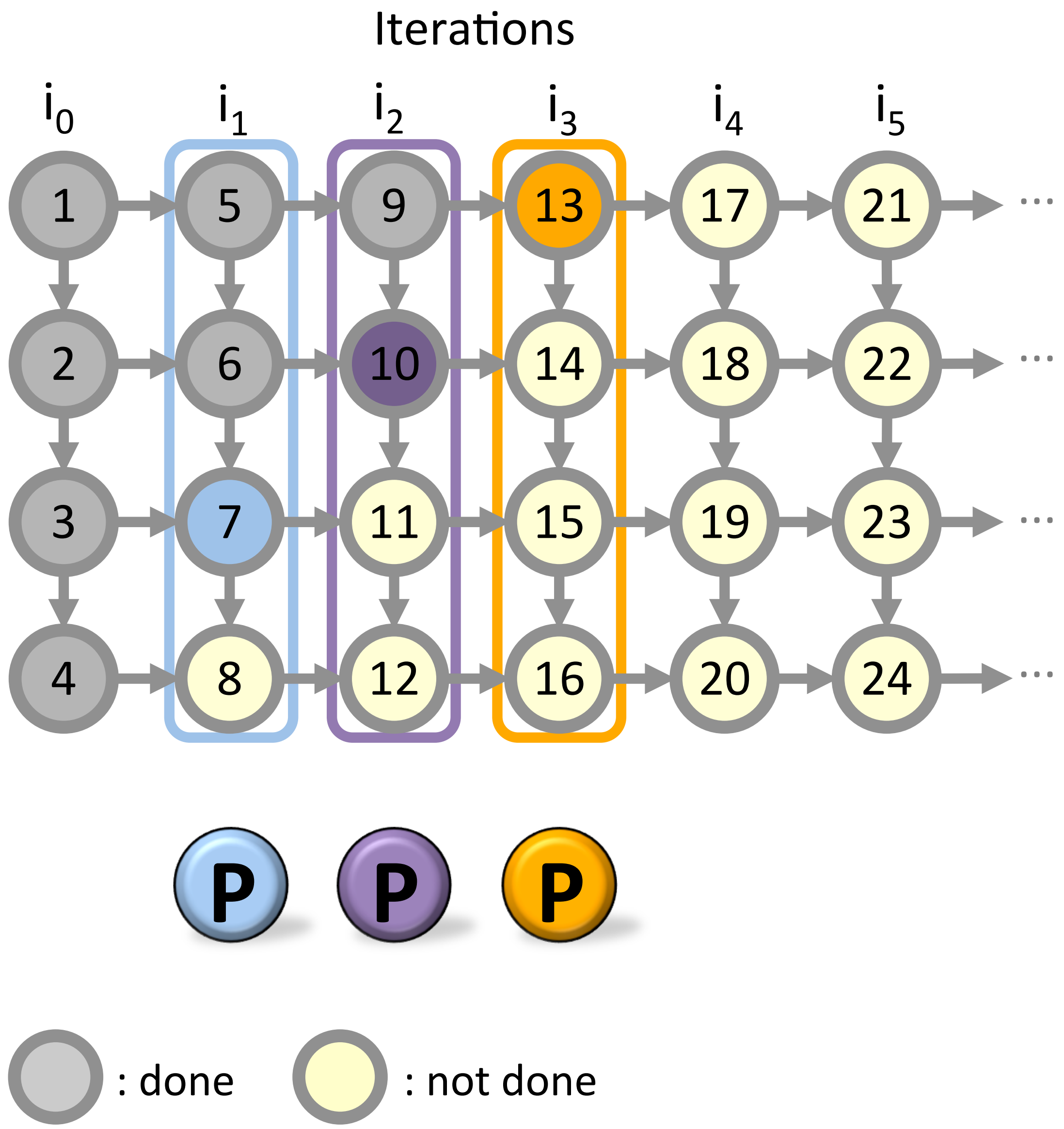
Iteration i_2 gets suspended.

The Check-Next Overhead



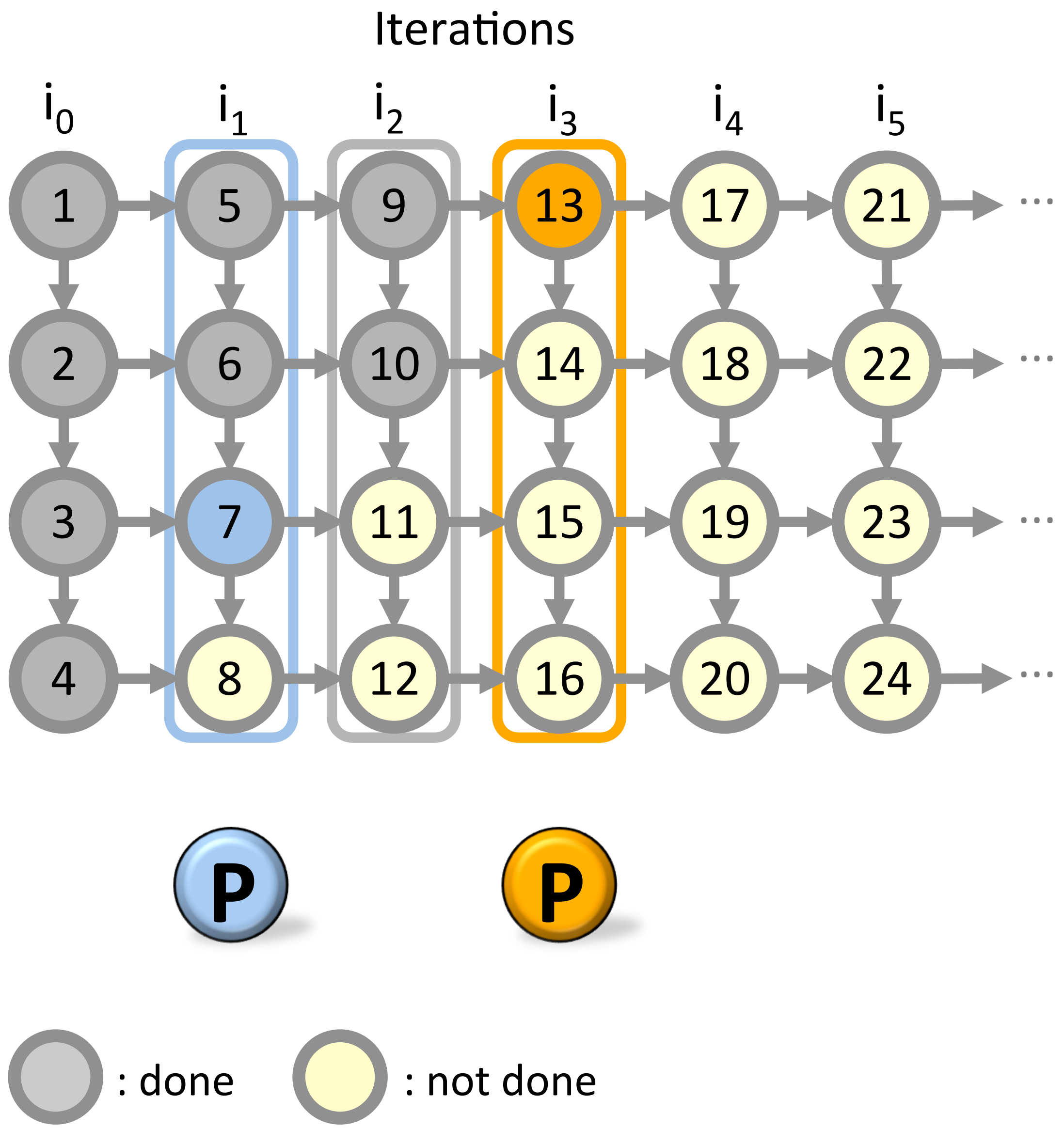
The blue worker re-enables iteration i_2 .

The Check-Next Overhead



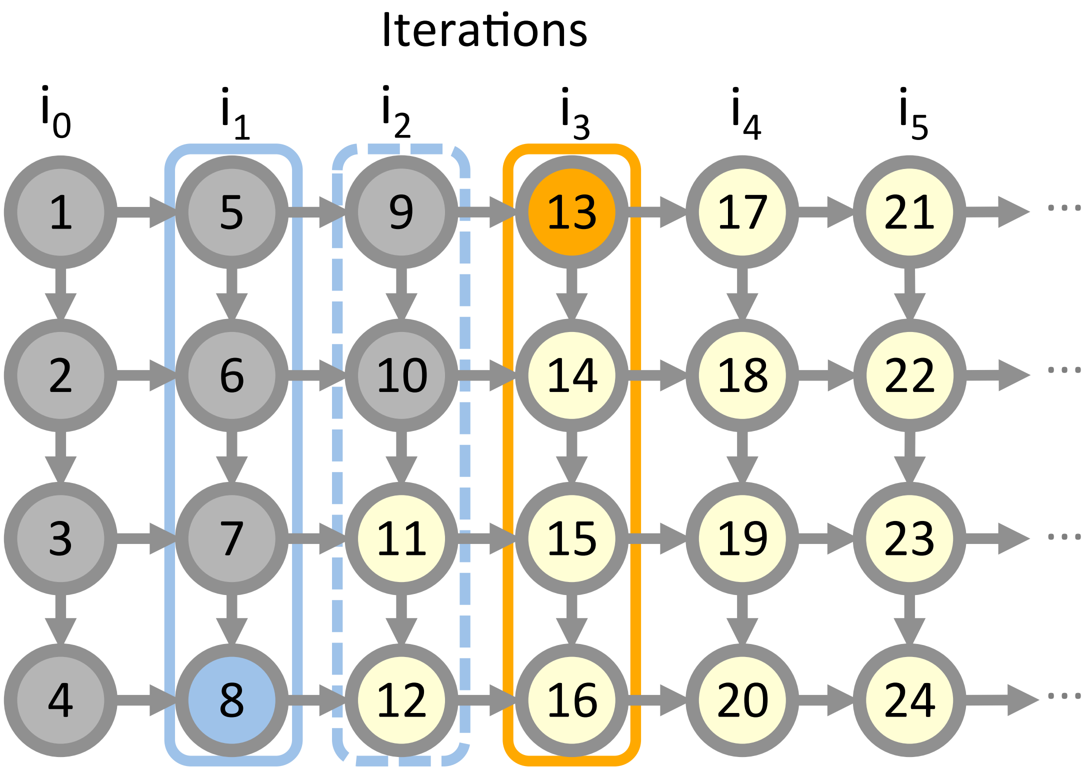
The purple worker steals iteration i_2 .

The Check-Next Overhead



Iteration i_2 gets suspended again.

The Check-Next Overhead



The blue worker re-enables iteration i_2 *again*.

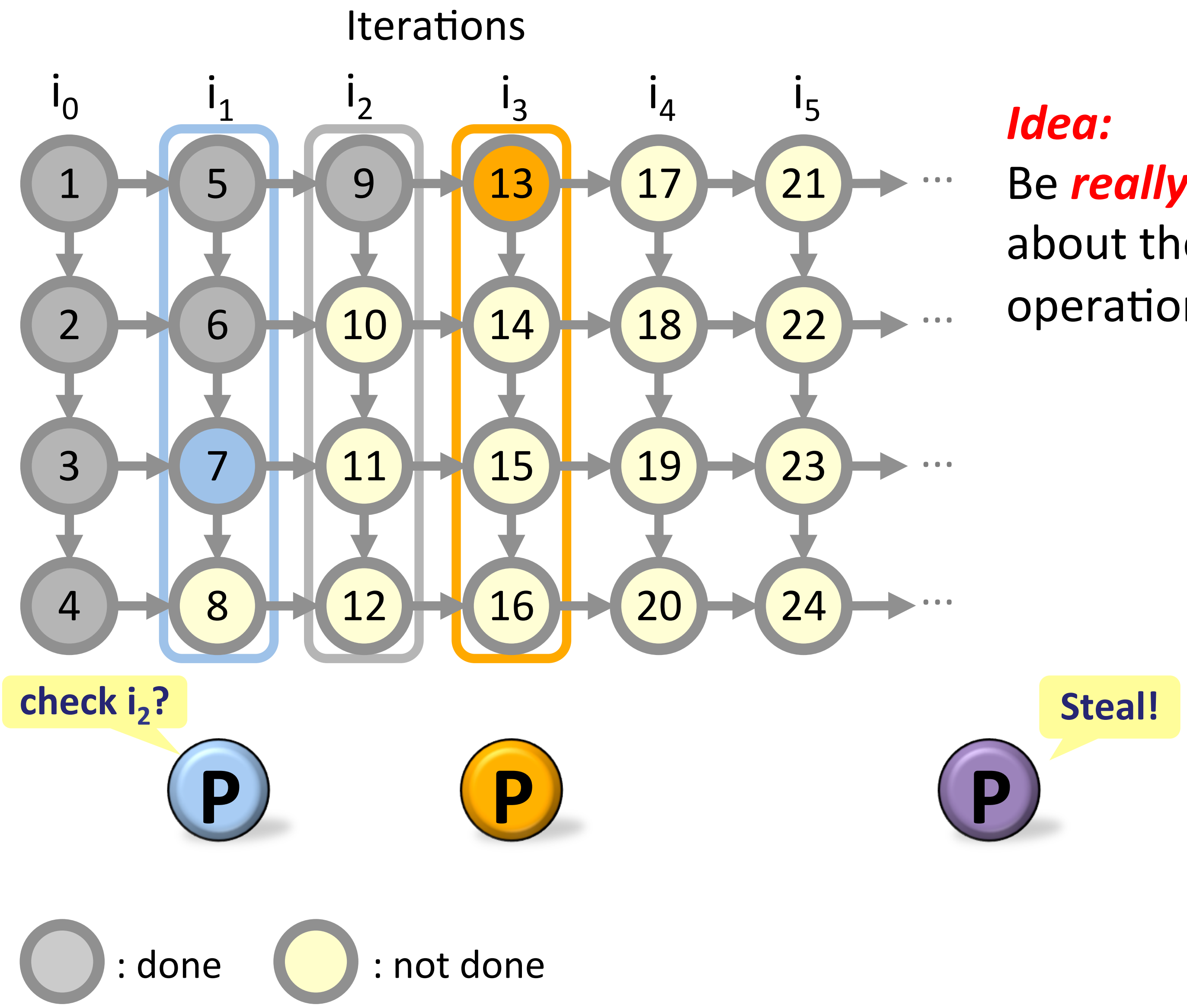
Iteration i_2 can get suspended and re-enabled repeatedly.

⇒ The blue worker must **check next** to re-enable i_2 after every stage!

● : done

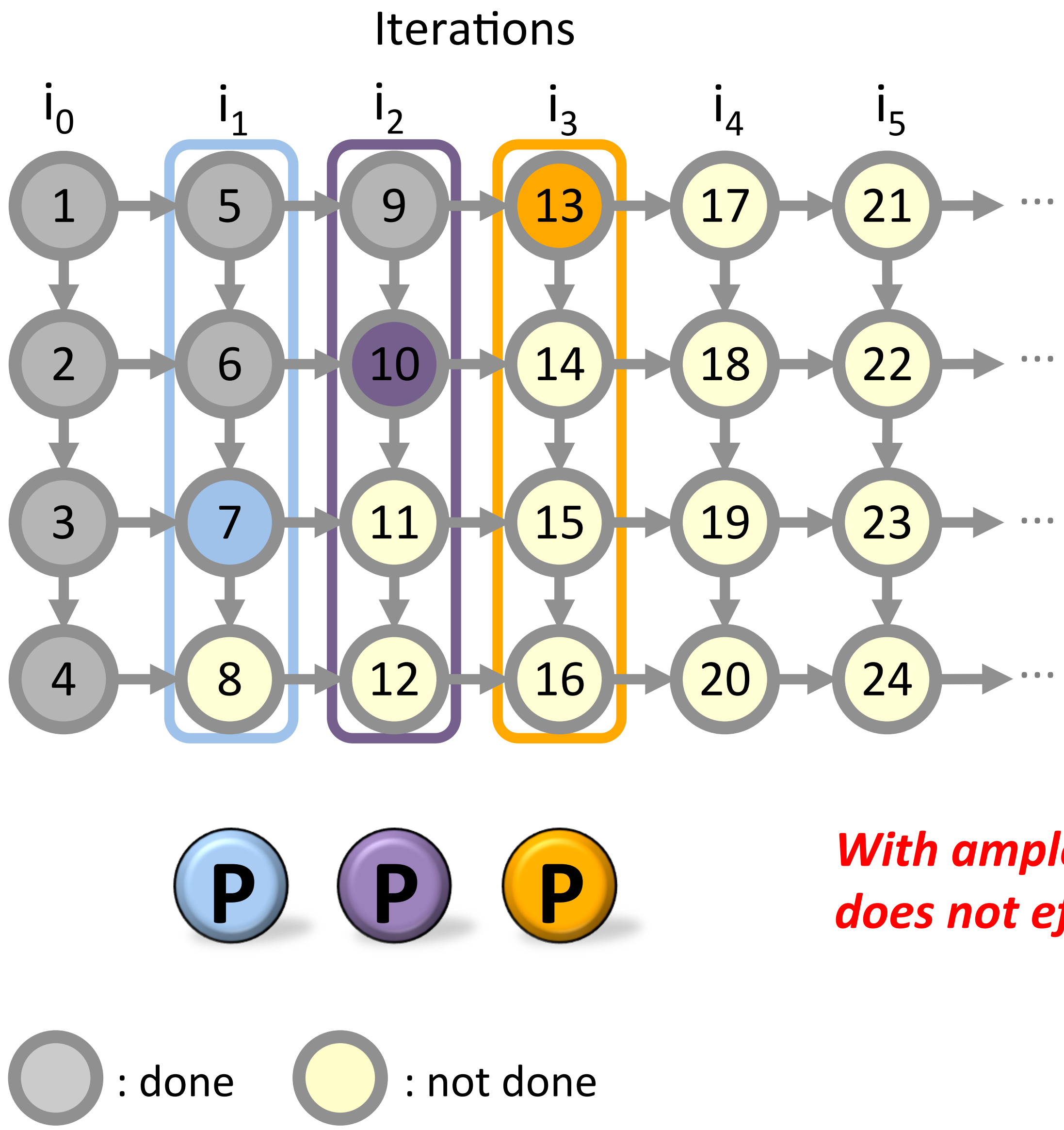
○ : not done

Optimization: Lazy Enabling



Idea:
Be ***really really lazy***
about the ***check-next***
operation.

Optimization: Lazy Enabling



Idea:

Be ***really really lazy*** about the ***check-next*** operation.

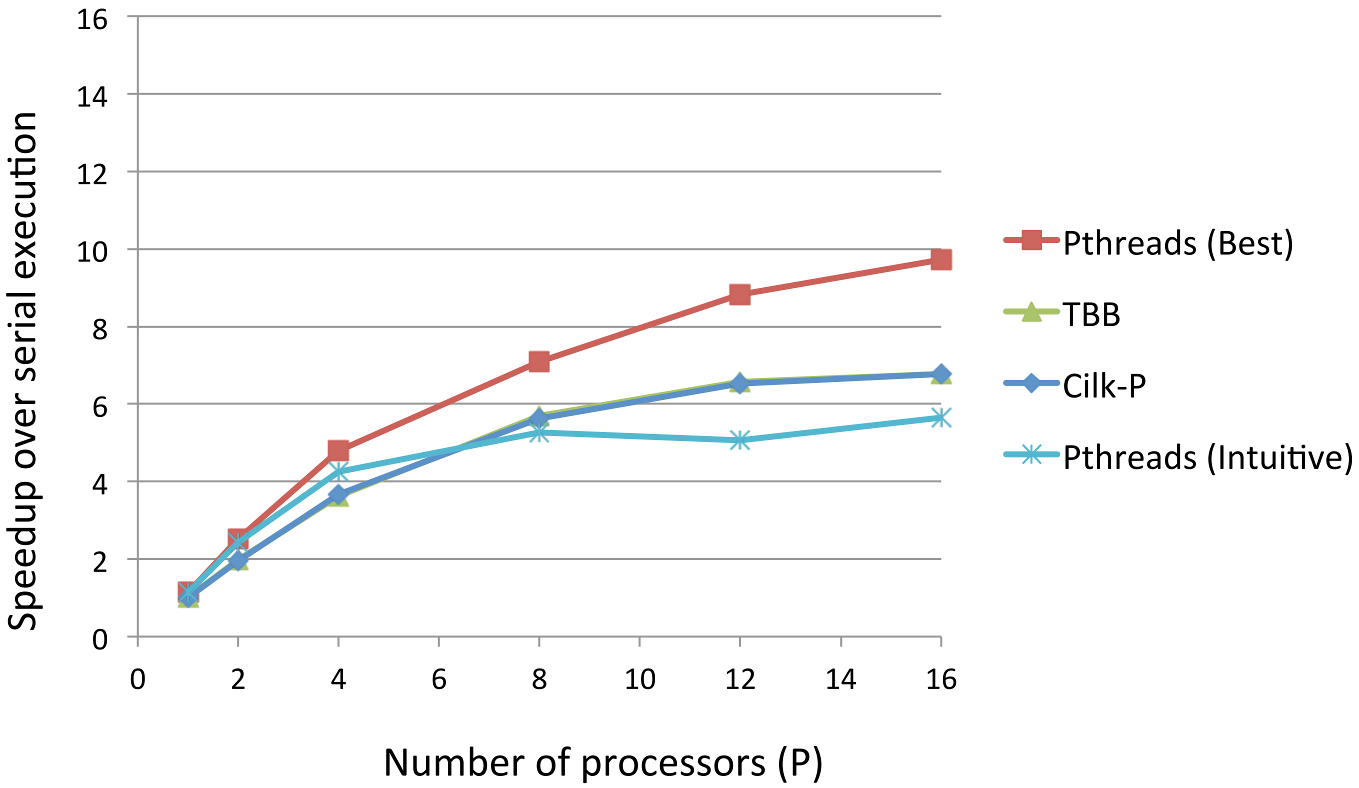
Punt the responsibility of checking next onto a thief stealing or until the worker reaches the end of its iteration.

With ample parallelism, this cost does not effect the performance!

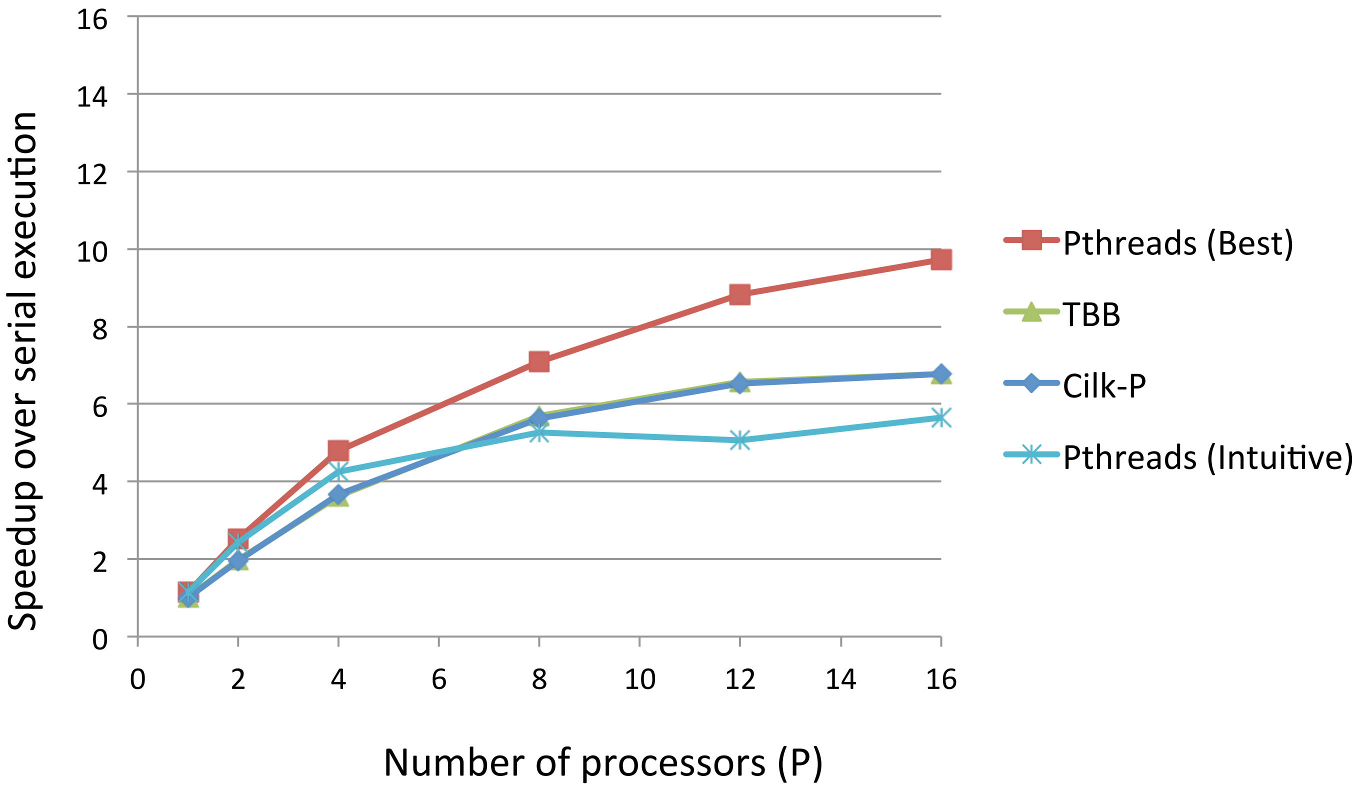
Implementation and Evaluation

Goal: Be competitive with highly-tuned code

Dedup Performance Comparison



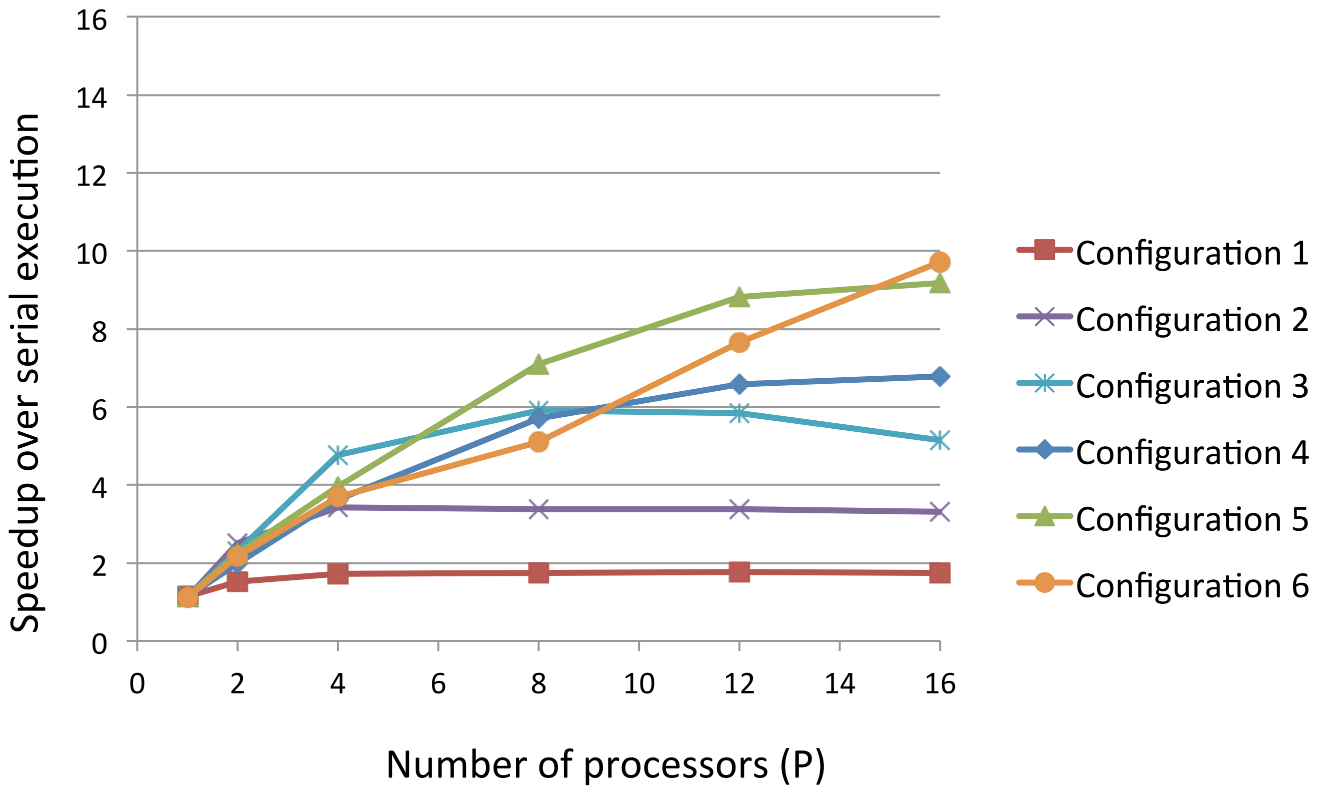
Dedup Performance Comparison



Measured parallelism for Cilk-P (and TBB)'s pipeline is merely 7.4.

The pthreaded implementation has more parallelism due to unordered stages.

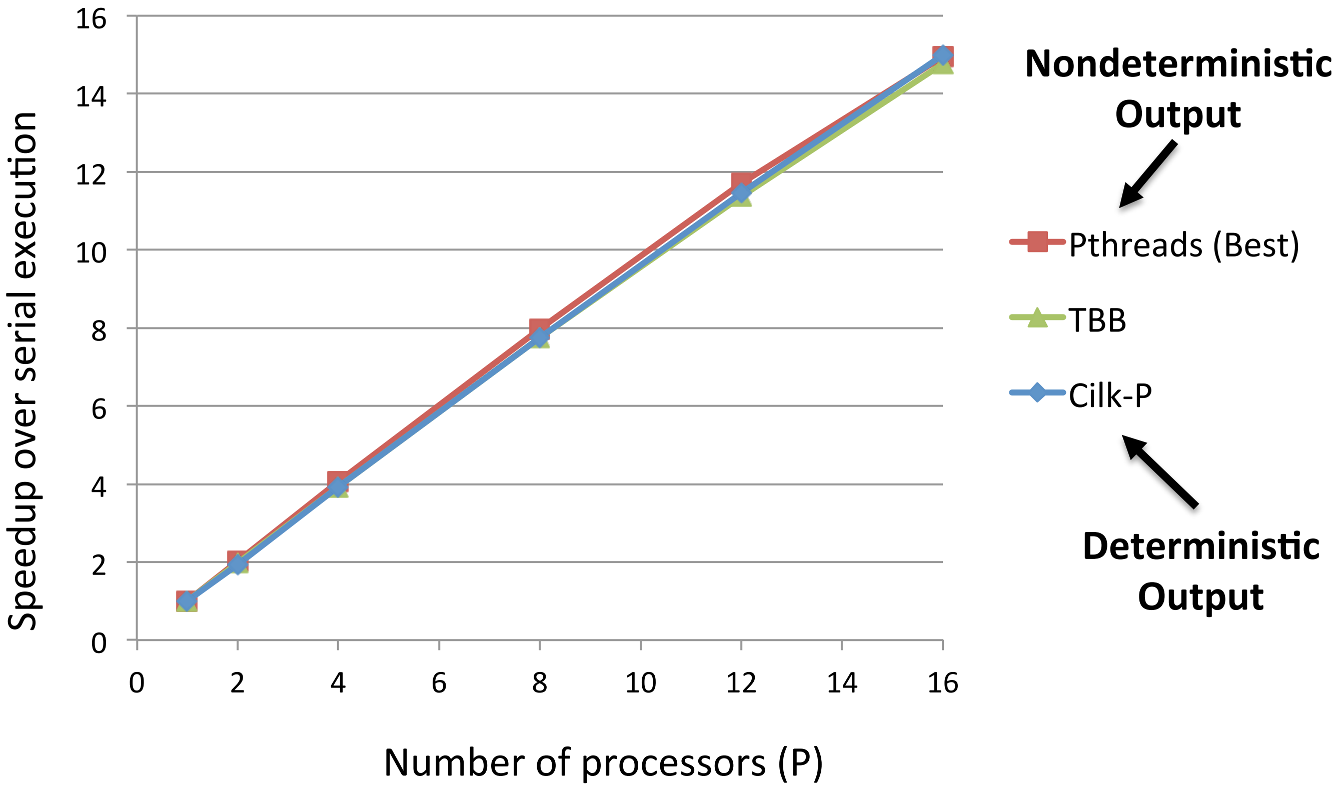
Dedup Performance Using Pthreads



Different configuration (threads per stage) leads to different results.

You don't need to do any of this with Cilk-P!

Ferret Performance Comparison

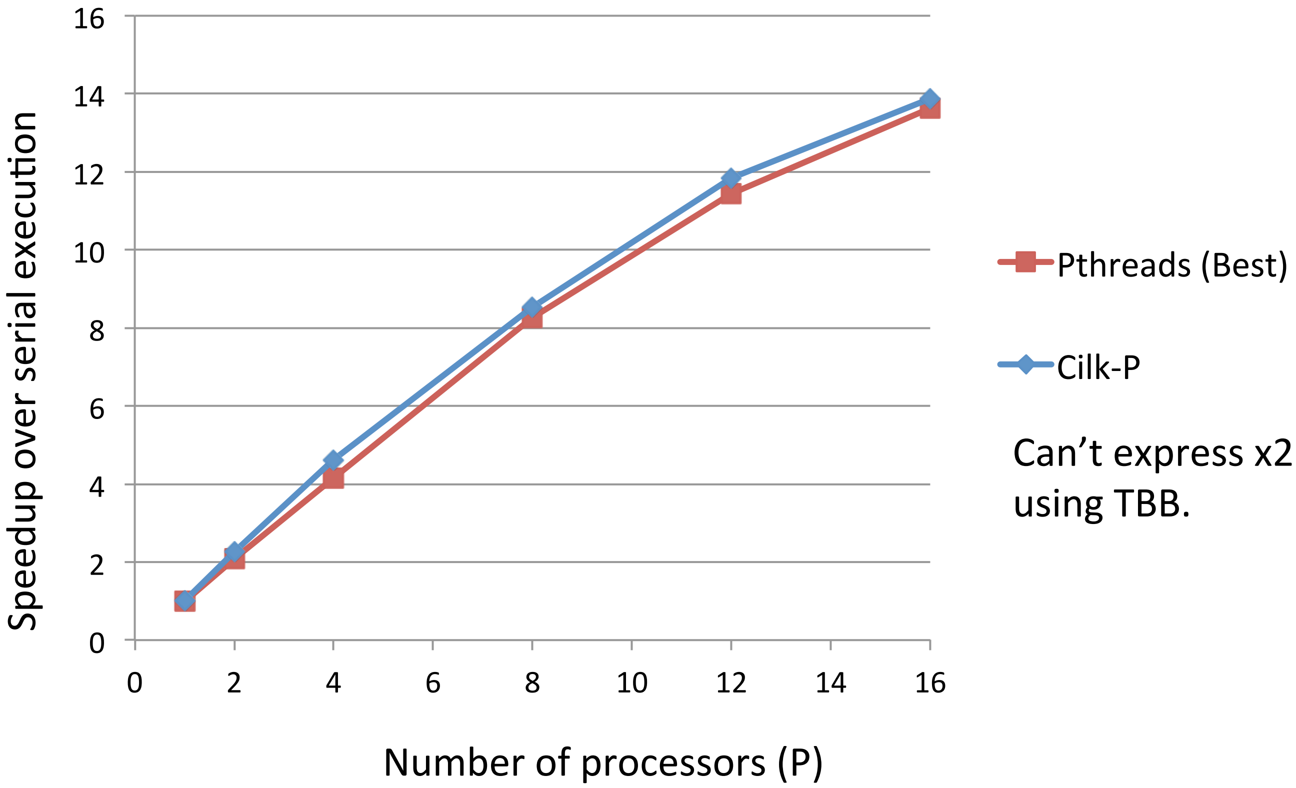


Nondeterministic Output

Deterministic Output

Cilk-P matches the best hand-tuned pthreaded code, and incurs no performance penalty for using the more general on-the-fly pipeline instead of a construct-and-run pipeline.

X264 Performance Comparison



Can’t express x264 using TBB.

Cilk-P matches the performance of hand-tuned pthreaded code, and the application programmer does not need to use any locks and conditional variables.

Pipeline Parallelism in Cilk-P

An instance of structured parallel programming

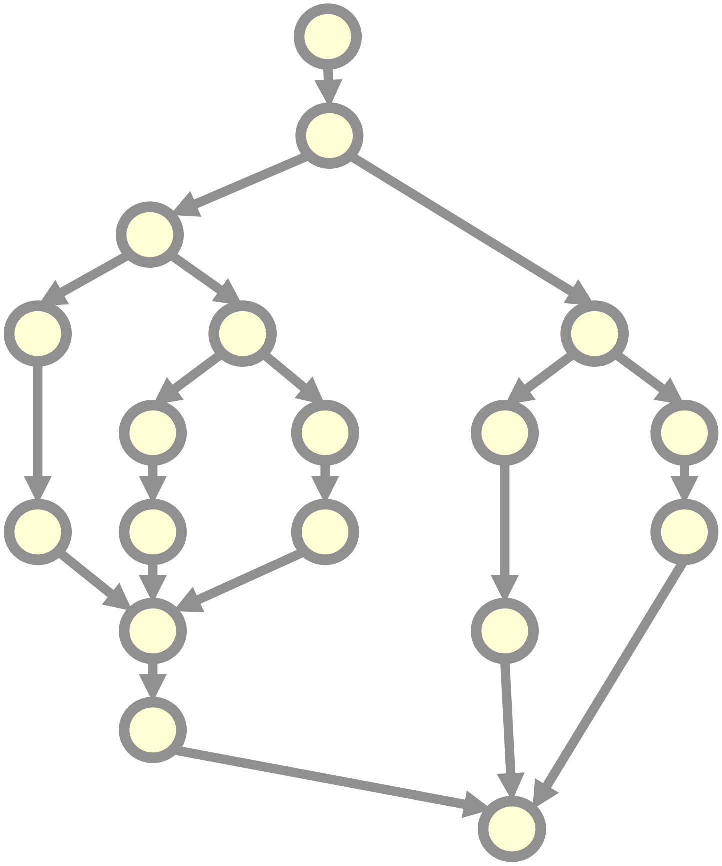
Cilk-P features:

- expressive linguistics for pipeline parallelism that separates the scheduling logic from program logic;
- effectively automates scheduling and synchronization; and
- provides a clean mental model for the programmer to reason about parallelism.

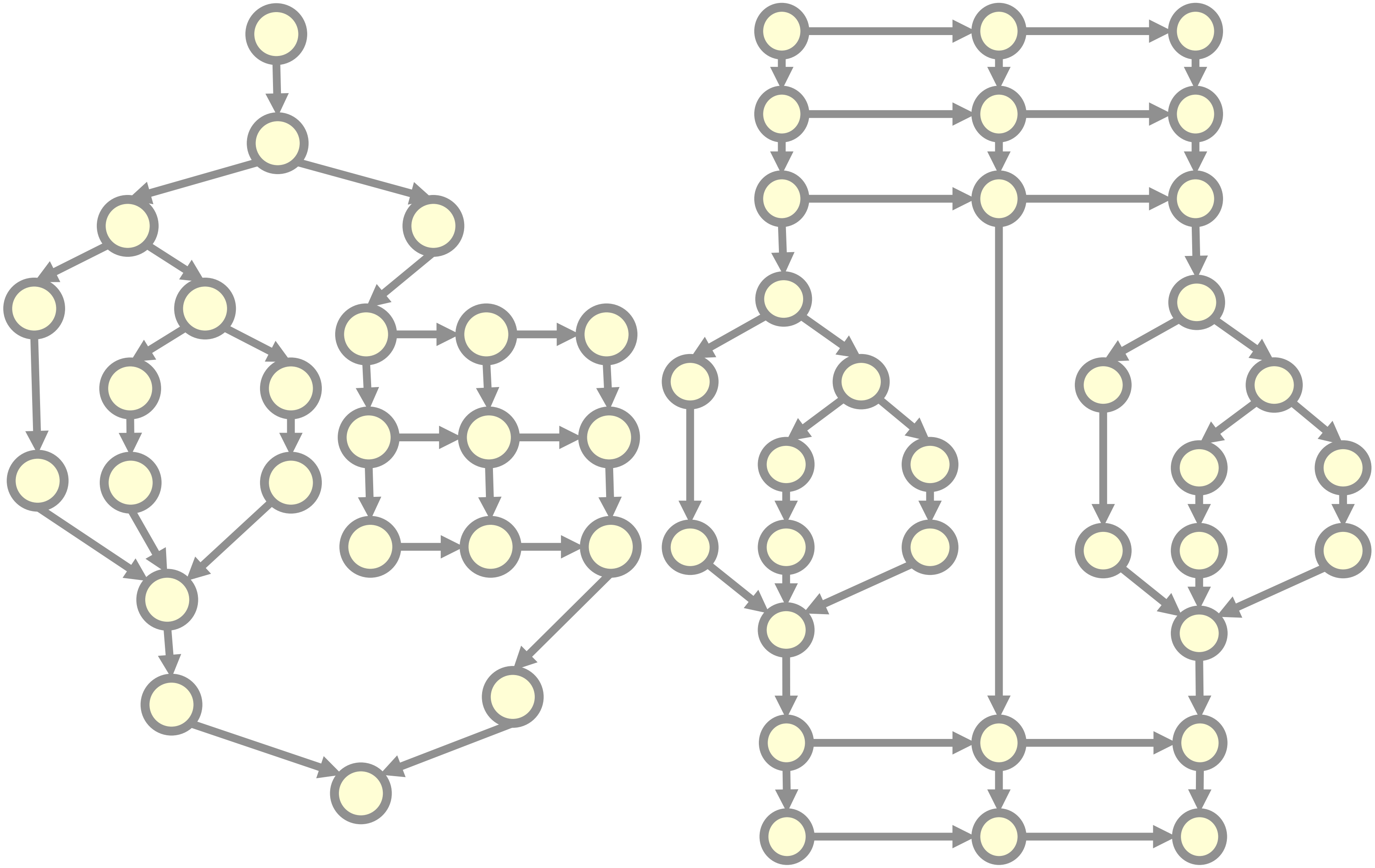
Cilk-P Inherited
Fork-Join Parallelism from Cilk

Cilk's fork-join parallelism [FLR98]:

```
int cilk fib(int n) {
     if(n < 2) { return n; }
     int x = spawn fib(n-1);
     int y = spawn fib(n-2);
     sync;
     return (x + y);
}
```



Cilk-P: A Unified Model



My Research

- Design language abstractions for structured parallel programming

- Develop efficient system support for these language abstractions

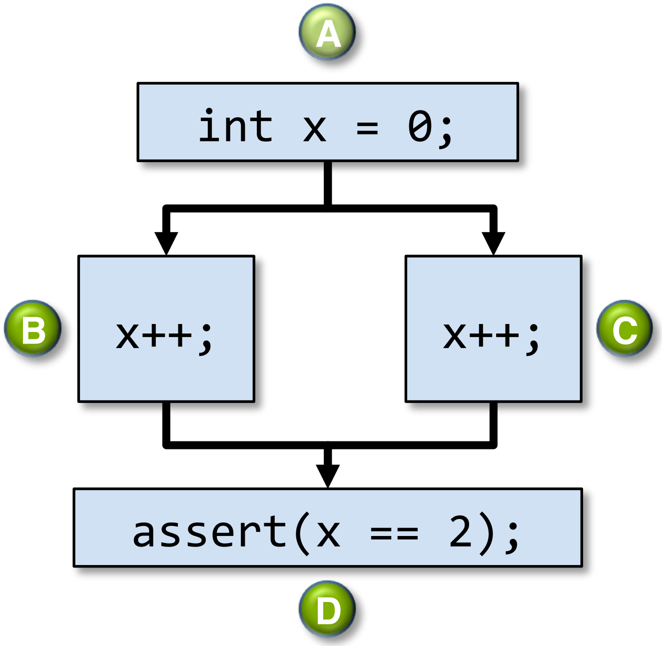
- Design tool support for debugging and performance engineering programs written in theses high-level language abstractions

Determinacy Race

A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

```
int x = 0;
parallel_for(int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```



A

`int x = 0;`

B

`x++;`

`x++;`

C

`assert(x == 2);`

D

dependency graph

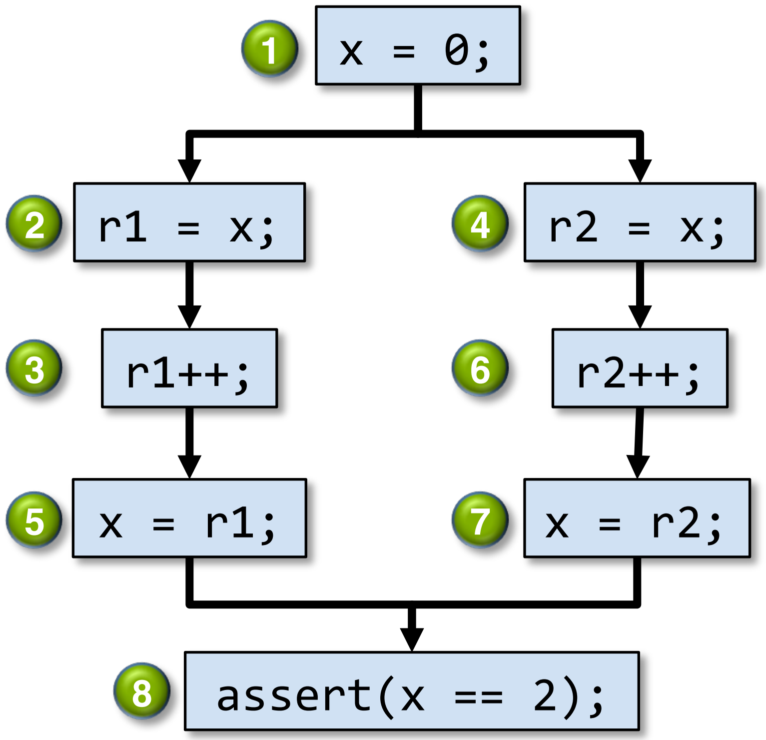
Determinacy Race

A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

```
int x = 0;
parallel_for(int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```

x can be either 1 or 2.



dependency graph

Why Determinacy Race?

In the absence of a determinacy race, a program executes in a deterministic fashion.


Edward Lee

Nondeterminism makes reasoning about parallel programs challenging![1]

Parallel programming must be deterministic by default![2]


Robert L. Bocchino Jr.


Vikram S. Adve


Sarita V. Adve


Marc Snir

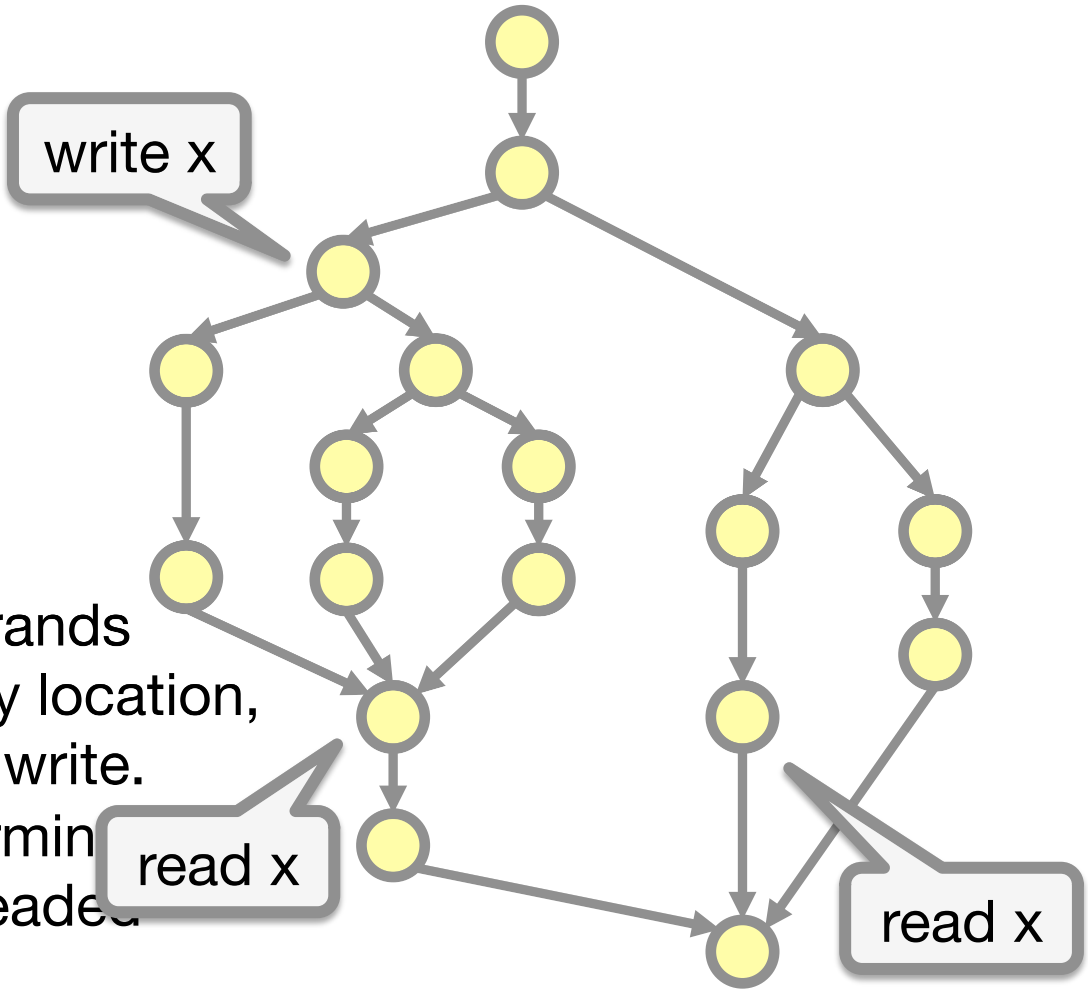

Julian Shun

Deterministic parallel algorithms can be fast![3]

---

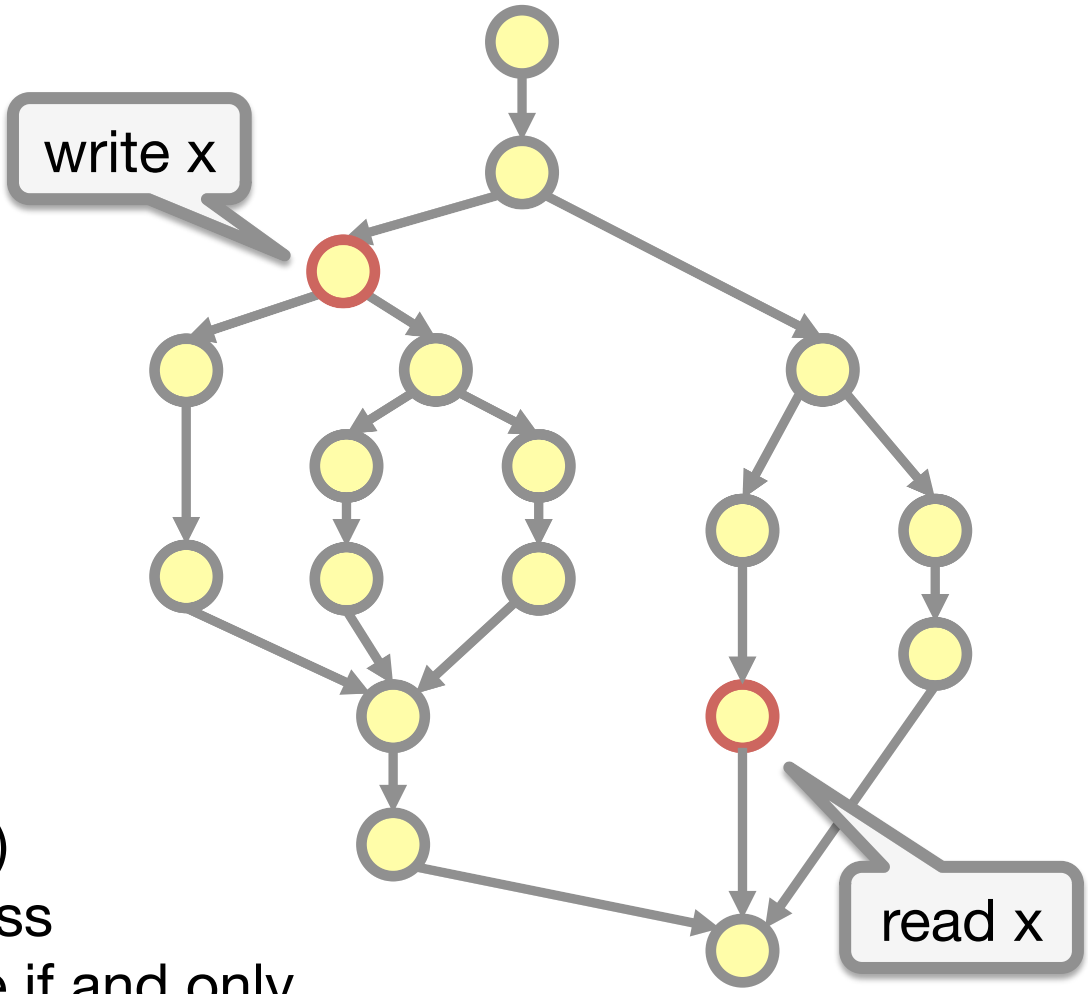
1. The problem with threads. Computer 39 (5), pg 33-42, 2006.
2. Parallel programming must be deterministic by default! HotPar, 2009.
3. Shared-memory parallelism can be simple, fast, and scalable, CMU 2015 (winner of the ACM Doctoral Dissertation Award).

Determinacy Race



- Two ***logically parallel*** strands access the same memory location, with at least one being a write.
- In the absence of a determin race, a dynamic multithreaded computation behaves deterministically.

On-the-fly
Determinacy Race Detection



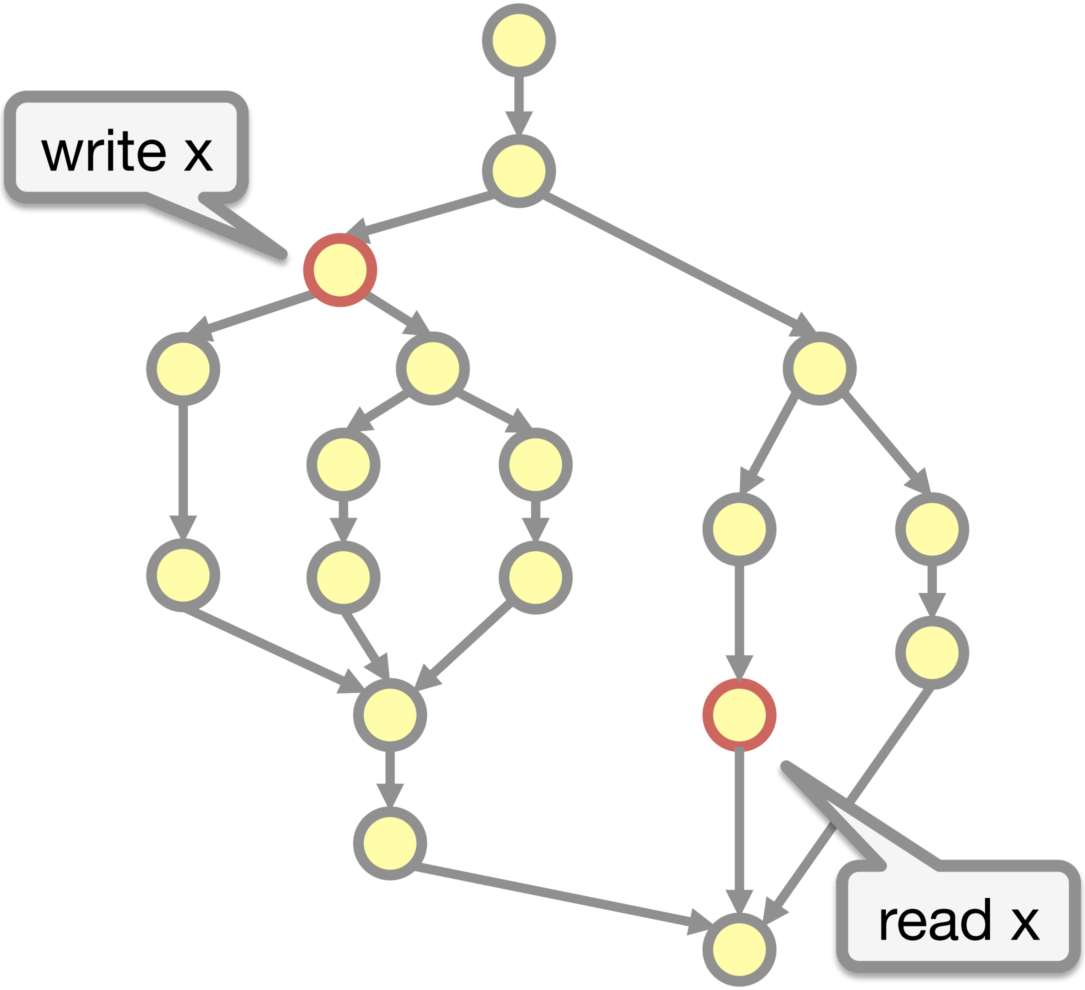
The tool detects races
as the program executes.

Goals:

- Allow the program to **execute in parallel**
- Detect races efficiently (**asymptotically optimal**)
- Provide strong correctness guarantees: report a race if and only if a race exists **for the given input**

Components of On-the-fly Determinacy Race Detection

- Design data structures to maintain **series parallel relationships** that tell us if two nodes are logically in parallel.
- Maintain **access histories** that tell us which nodes accessed the memory location previously.
- ***Challenge:*** Have low overheads and should scale.

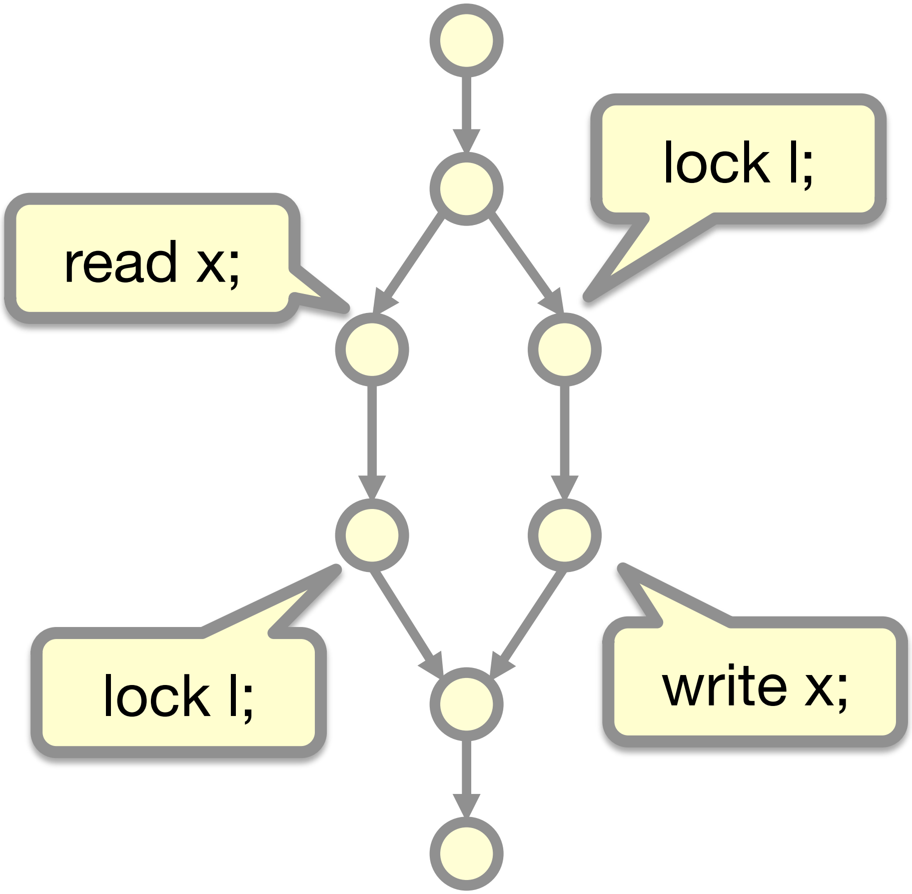


P-Racer

- Provably efficient and correct parallel on-the-fly race detector for both fork-join and pipeline parallelism
- Open problem:
    - Reduce overheads of access history and instrumentation.
    - Generalize to programs with more complex structural properties.
    - Generalize to programs with locks.

Issues with Locks

- Lock operations generate complex dependences, making it difficult to track SP-relationships efficiently.
- Races or not depending on the schedule of lock acquire / release.



PORRidge

- Provably efficient and scalable deterministic record and replayer for fork-join parallel programs that employ locks
  - encapsulate all nondeterminism in the runtime system!
  - Can record and replay on different number of threads
- Open problem:
  - Currently the tool only captures nondeterminism due to lock operations
  - Reduce overhead for logging (both space and time)

Questions?