Concurrency:
Multi-core Programming & Data Processing

Futures, Scheduling, and Work Distribution

Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/

*Based on slides by **Maurice Herlihy** and **Nir Shavit***

Institut d'informatique
Université de Neuchâtel

How to Write Parallel Apps?

- How to
  - Split a program into parallel parts…
  - In an effective way…
  - With proper thread management?

Matrix Multiplication

$$(C) = (A) \cdot (B)$$

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$$

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int k = 0; k < n; k++)
      dotProduct += a[row][k] * b[k][col];
    c[row][col] = dotProduct;
  }
}
```

A thread

Which matrix entry to compute

Actual computation

Matrix Multiplication

We create a thread for each dotproduct. Shitload of threads

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row, col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Create n × n threads

Start them

Wait for them to finish

What is wrong with this code?

Thread Overhead

- Threads require resources
  - Memory for stacks
  - Setup, teardown
- Scheduler overhead
- Worse for short-lived threads

Thread Pools

not bound to a single thread with start and end

- More sensible to keep a pool of long-lived threads
- Threads assigned short-lived tasks
  - Runs the task
  - Rejoins pool
  - Waits for next assignment

Thread Pool = Abstraction

don't care about numa

- Insulate programmer from platform
  - Big machine, big pool
  - And vice-versa
- Portable code
  - Runs well on any platform
  - No need to mix algorithm/platform concerns

ExecutorService Interface

- In `java.util.concurrent`
  - Task = `Runnable` object
    - If no result value expected
    - Calls `run()` method
  - Task = `Callable<T>` object
    - If result value of type `T` expected
    - Calls `T call()` method

Future<T>

holder for result that will come later

```
Callable<T> task = …;
…
Future<T> future = executor.submit(task);
…
T value = future.get();
…
```

Submitting a `Callable<T>` task returns a `Future<T>` object

The `Future`'s `get()` method blocks until the value is available

Future<T>

```
Runnable task = …;
…
Future<?> future = executor.submit(task);
…
future.get();
…
```

Submitting a `Runnable` task returns a `Future<?>` object

The `Future`'s `get()` method blocks until the the computation is complete

Note

- Executor service submissions…
  - Are purely advisory in nature
- The executor…
  - Is free to ignore any such advice…
  - And could execute tasks sequentially

dont have guarantee that separate threads will be run in different threads

Matrix Addition

$$(C) = (A) + (B)$$

split in 4 different submatrices

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} + B_{00} & A_{01} + B_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

4 parallel additions

Matrix Addition (Pseudo-Code!)

recursively split matrix until we have one element to compute

```
class AddTask implements Runnable {
  int[][] a, b, c; …
  public void run() {
    if (a.length == 1) {
      c[0][0] = a[0][0] + b[0][0];
    } else {
      // Partition into half-size matrices
      Future<?> c00 = exec.submit(add(a00, b00)); …
      Future<?> c11 = exec.submit(add(a11, b11));
      c00.get(); …; c11.get(); …
    }
  }
}
```

Base case: add directly

Constant-time operation

Submit 4 tasks

Let them finish

Dependencies

- Matrix example is not typical
- Tasks are independent
  - Do not need results of one task…
  - To complete another
- Often tasks are not independent

Fibonacci

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

- Note
  - Potential parallelism
  - Dependencies
- This Fibonacci implementation is
  - Egregiously inefficient
  - So do not deploy it!

lets assume this impl is the best and we want to do that in parallel

Multi-threaded Fibonacci

```
class FibTask implements Callable<Integer> {
  static ExecutorService exec = Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg >= 2) {
      Future<Integer> left = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else
      return 1;
  }
}
```

Parallel calls

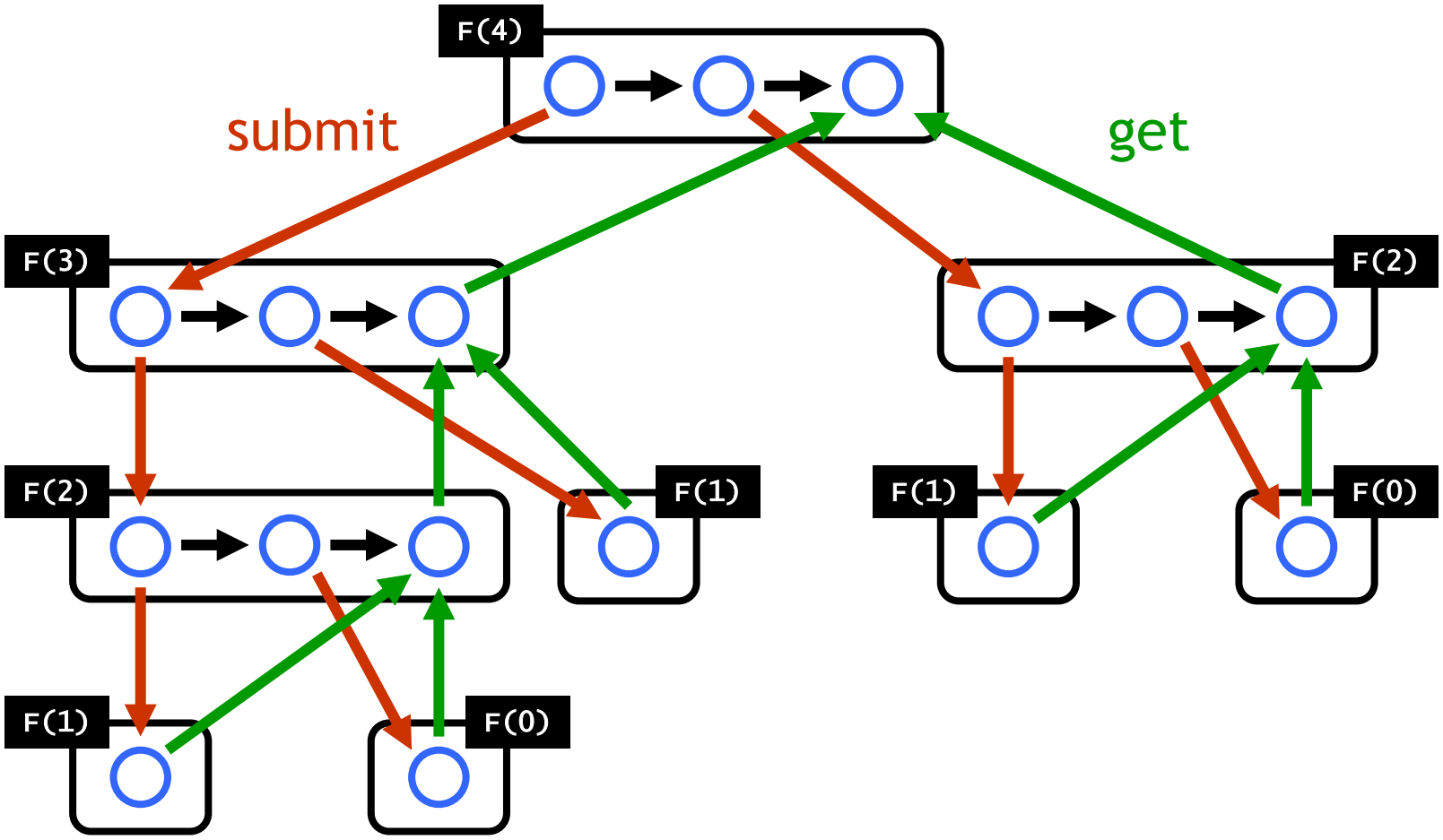
Pick up & combine results

Dynamic Behavior

- Multithreaded program is
  - A directed acyclic graph (DAG)…
  - That unfolds dynamically
- Each node is
  - A single unit of work

Fibonacci DAG

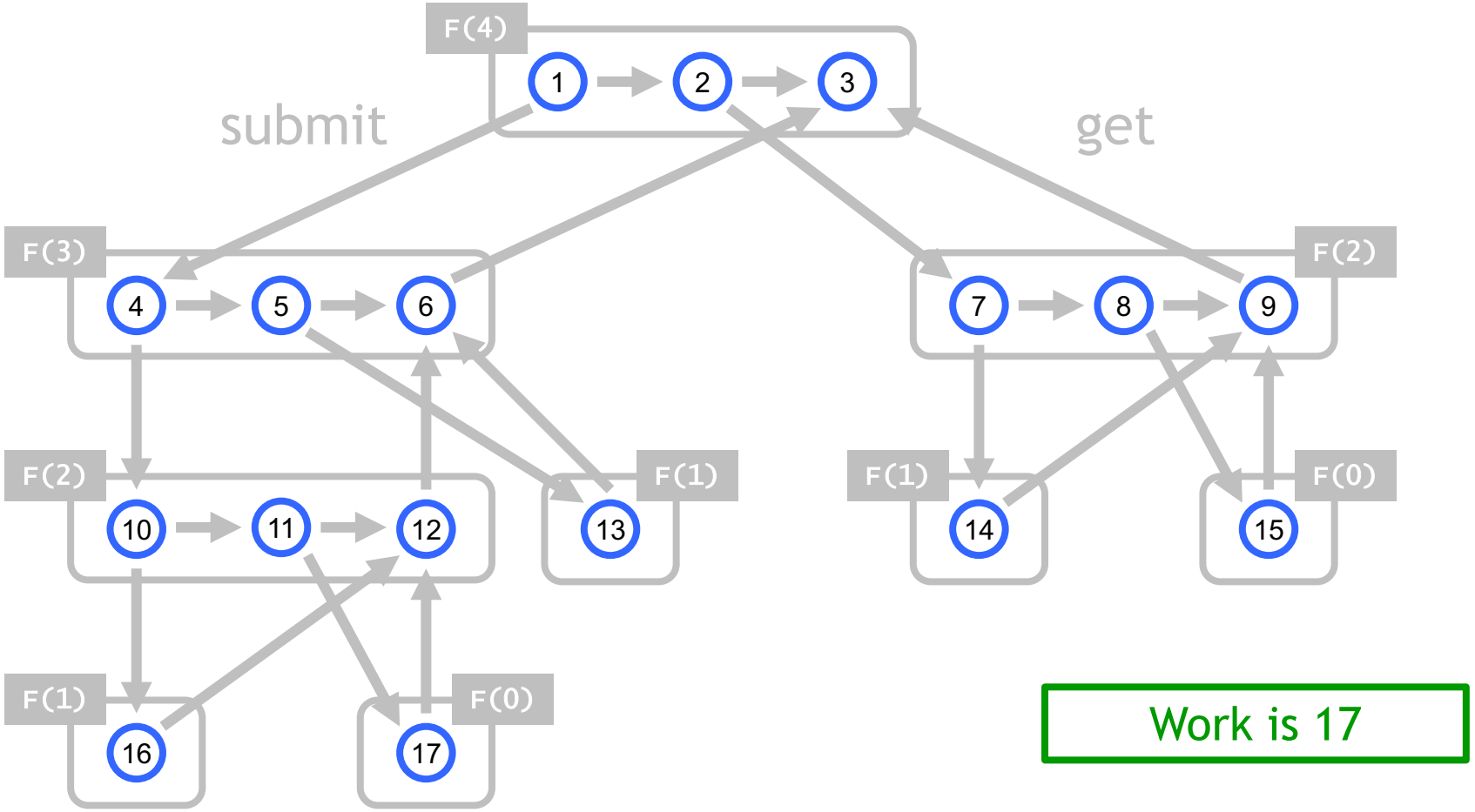
following edges shows deps



How Parallel is That?

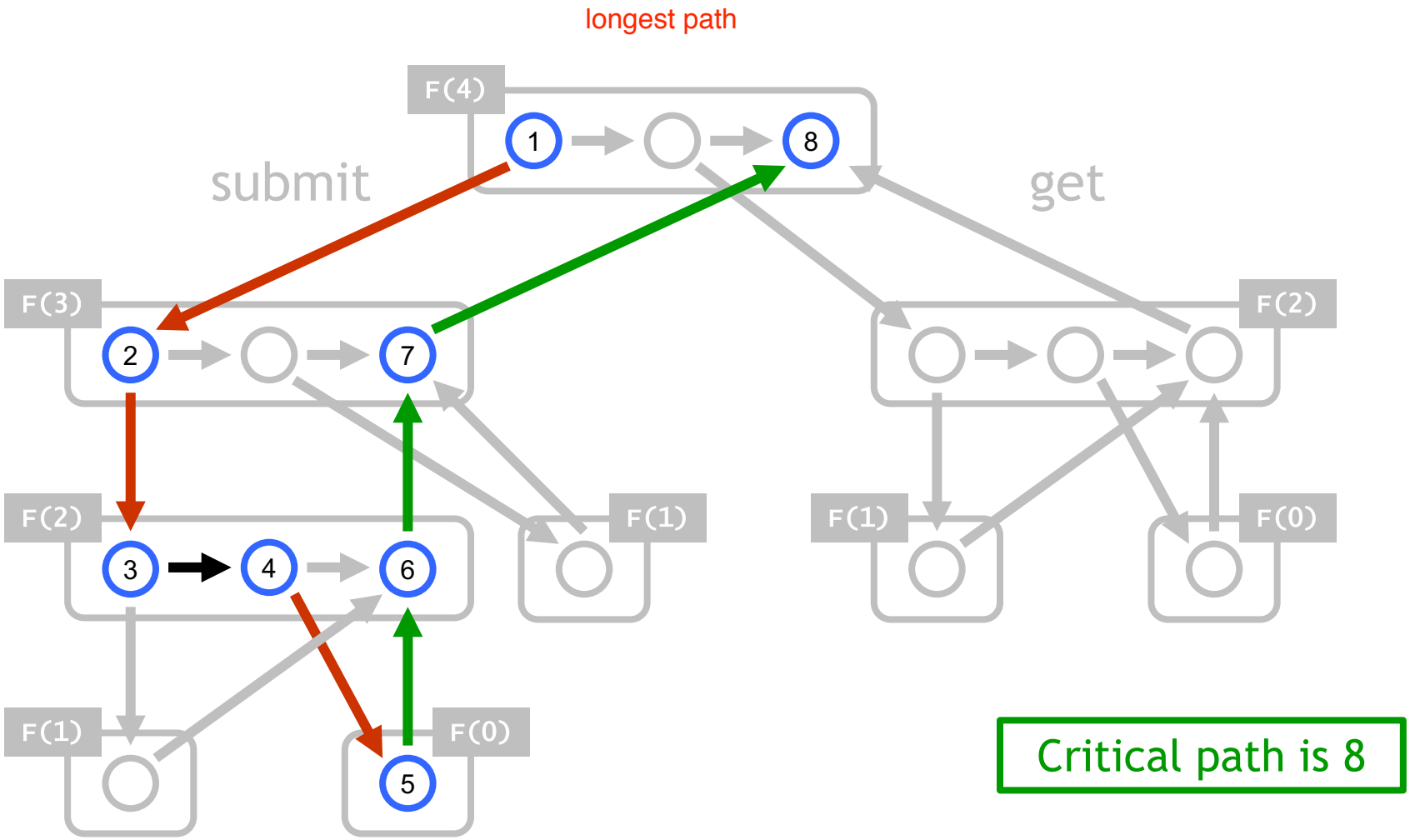
- Two metrics:
- Work
  - Total time on one processor
- Critical path length
  - Longest dependency path
  - Cannot beat that!

Fibonacci: Work



Work is 17

Fibonacci: Critical Path



Notation Watch

- T_P = time on P processors
- T_1 = work
  - Time on 1 processor
- T_∞ = critical path length
  - Time on ∞ processors

Simple Bounds

- $T_P \geq T_1/P$ linear speedup
  - In one step, cannot do more than P work
- $T_P \geq T_\infty$
  - Cannot beat infinite resources

More Notation Watch

- Speedup on P processors
  - Ratio T_1/T_P
  - How much faster with P processors
- Linear speedup
  - $T_1/T_P = \Theta(P)$
- Max speedup (average parallelism)
  - T_1/T_∞

Matrix Addition

$$(C) = (A) + (B)$$

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} + B_{00} & A_{01} + B_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

4 parallel additions

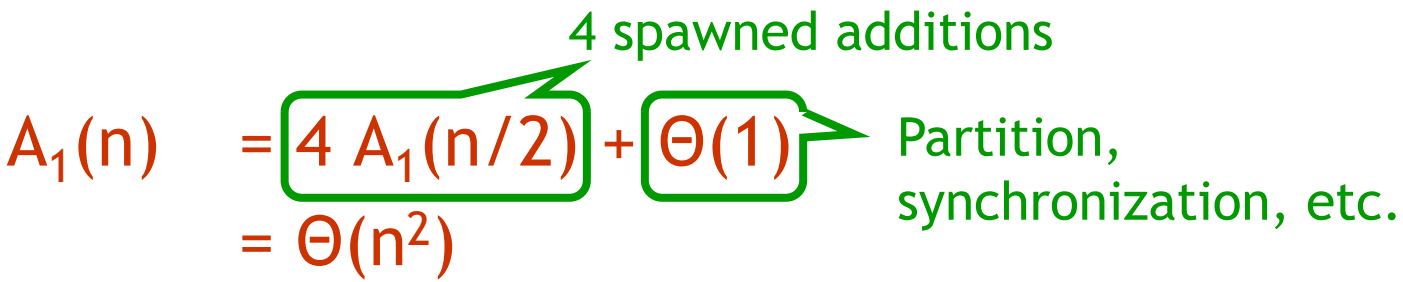
Addition

Addition on p processors

- Let $A_P(n)$ be running time
  - For $n \times n$ matrix
  - On P processors
- For example
  - $A_1(n)$ is work
  - $A_\infty(n)$ is critical path length

Addition

- Work is



(same as double-loop summation)

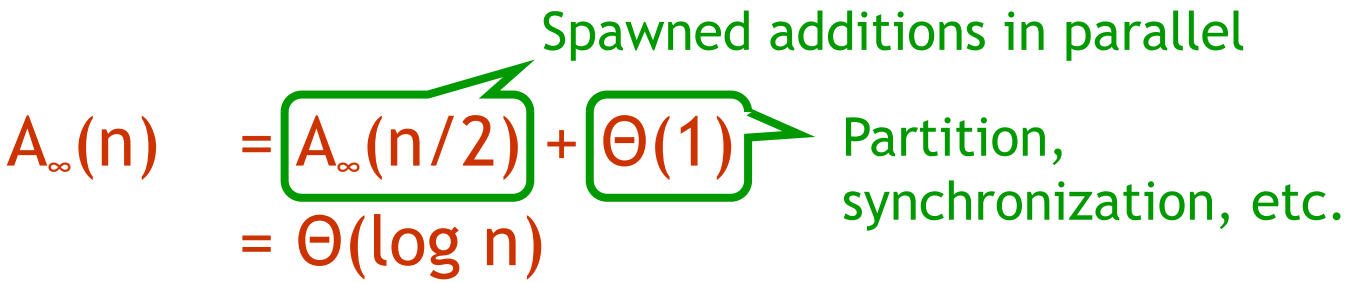
quadratic

herleitung:

a(n) = 4*a(n/2)
= 4 * 4 * a(n/4)
...
= 4^log(n) * 1
= 2^log(n) * 2^log(n) * 1
= n*n*1
= n^2

Addition

- Critical path length is



Matrix Multiplication

more deps

ops need to be done in a seq

$$(C) = (A) \cdot (B)$$

split into submatrices

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Two Phases

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix}$$

First 8 parallel multiplications

Then 4 parallel additions

Multiplication

- Work is

8 parallel multiplications

$$M_1(n) = 8\,M_1(n/2) + A_1(n)$$ Final addition

$$= \Theta(n^3)$$

(same as serial triple-nested loop)

Multiplication

- Critical path length is

Half-size parallel multiplications

$$M_\infty(n) = M_\infty(n/2) + A_\infty(n)$$ Final addition

$$= M_\infty(n/2) + \Theta(\log n)$$

$$= \Theta(\log^2 n)$$

Parallelism

- $M_1(n)/M_\infty(n) = \Theta(n^3/\log^2 n)$
- To multiply two 1000×1000 matrices
  - $1000^3/10^2 = 10^7$
- We can keep 10^7 processors busy!

these are the max number of nodes you can really compute in parallel

  - Much more than number of processors on any real machine

Shared-Memory Multiprocessors

- Parallel applications
  - Do not have direct access to HW processors
- Mix of other jobs
  - All run together
  - Come and go dynamically

Scheduling

- Ideally
  - User-level scheduler
  - Maps tasks to dedicated processors
- In real life
  - User-level scheduler
    - Maps tasks to fixed number of threads
  - Kernel-level scheduler
    - Maps threads to dynamic pool of processors

Speedup

- Map threads onto P processes
- Cannot always get P-fold speedup
  - What if the kernel does not cooperate?
- Can try for speedup proportional to…
  - Time-averaged number of processors…
  - The kernel gives us

Scheduling Hierarchy

- User-level scheduler
  - Tells kernel which threads are ready
- Kernel-level scheduler
  - Synchronous (for analysis, not correctness!)
  - Picks p_i threads to schedule at step i
- Processor average P_A over T steps is $$P_A = \frac{1}{T} \sum_{i=1}^{T} p_i$$

Greed is Good

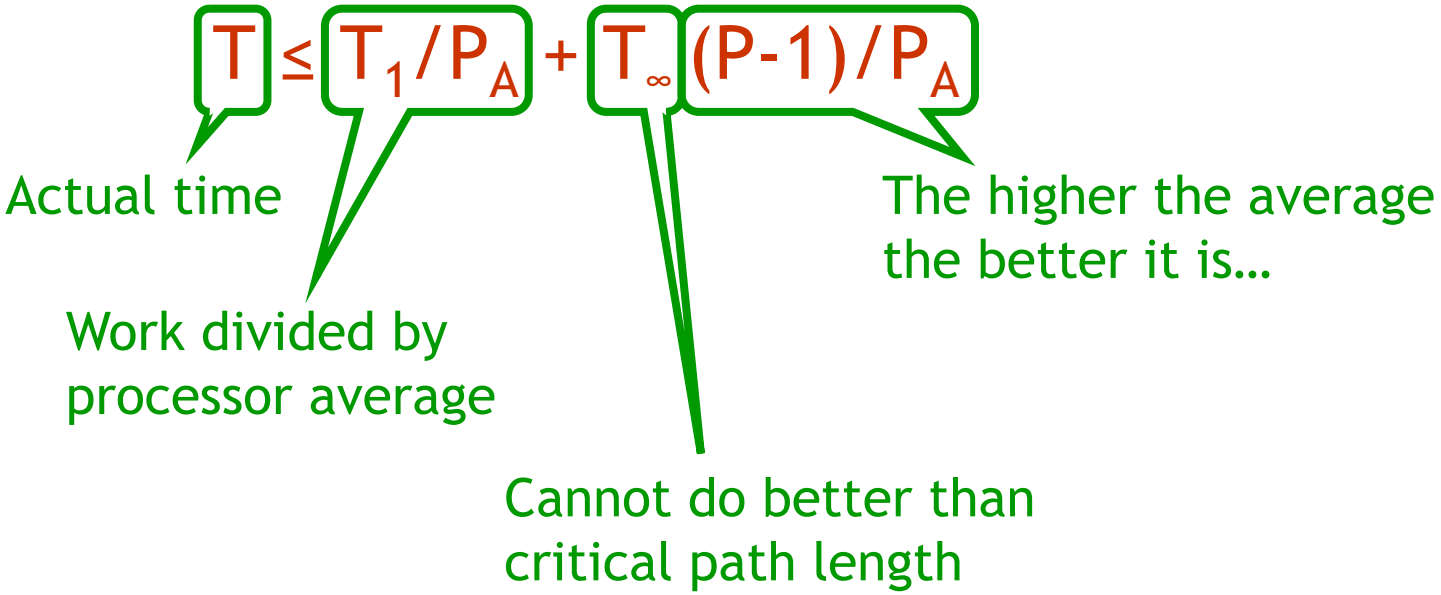
optimal schedule is make all the jobs finish in the least amount of time

- Greedy scheduler
  - Schedules as much as it can
  - At each time step
- Optimal schedule is greedy
- But not every greedy schedule is optimal

also simple to implement

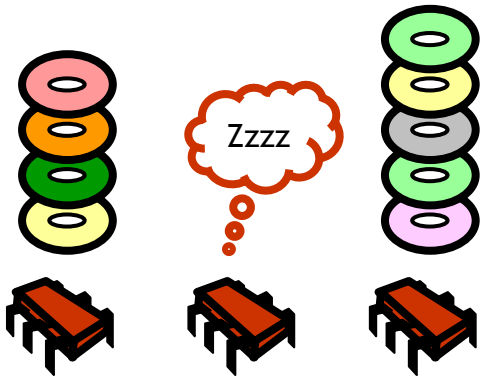
Theorem

- Greedy scheduler ensures that



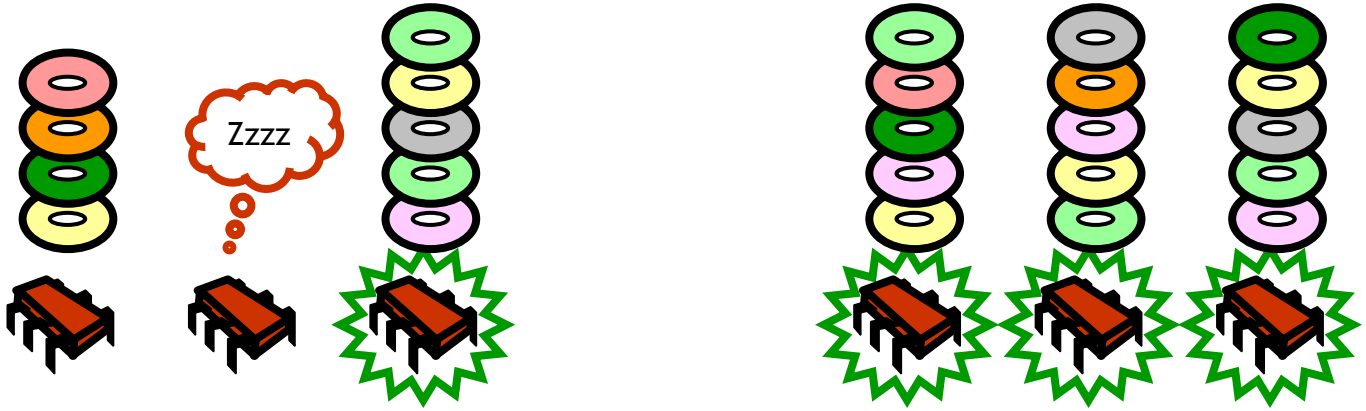
Work Distribution

- Problem: assign ready tasks to idle threads
- Two approaches
  - Work shedding (give work to others)
  - Work stealing (take work from others)



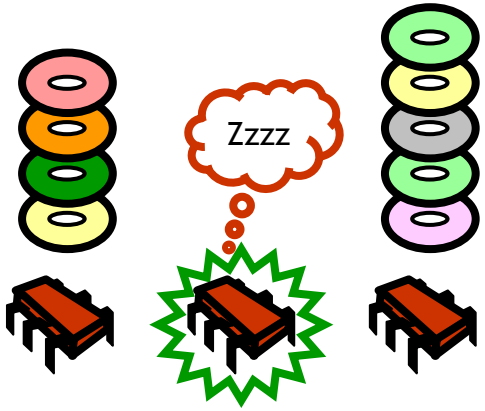
Work Shedding

- Overloaded thread tries to offload surplus tasks on others
- **Cons:** may create cycles of overloaded threads



Work Stealing

- Thread that runs out of work tries to steal work from others
- **Pros:** no coordination is necessary if all threads are overloaded

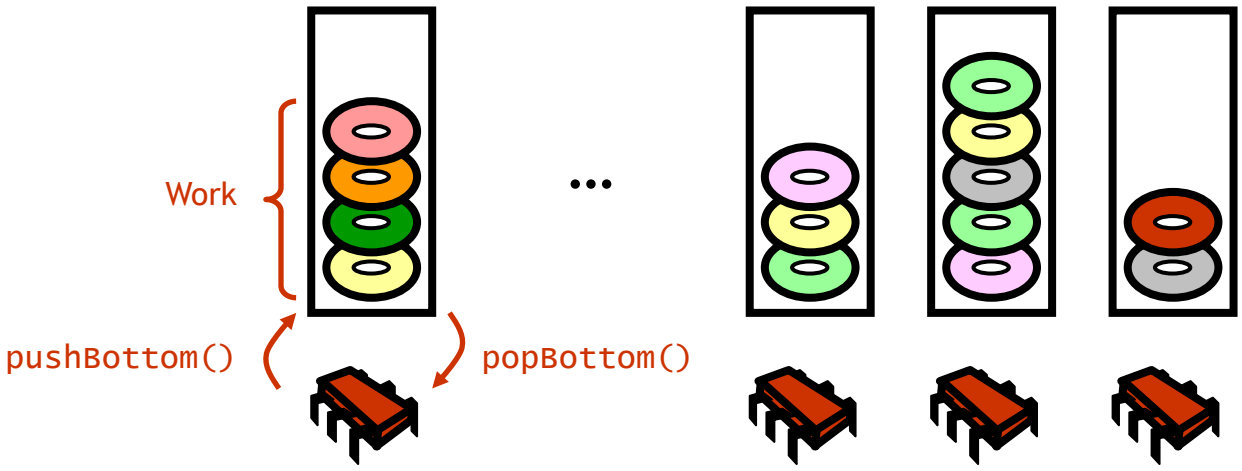


Lock-Free Work Stealing

- Each thread has a pool of ready work
- Remove work without synchronizing
- If you run out of work, steal from someone else
- Choose victim at random

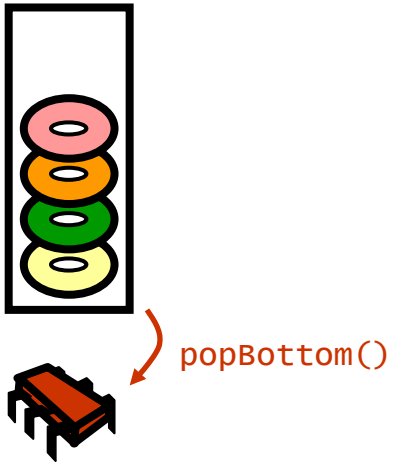
Local Work Pools

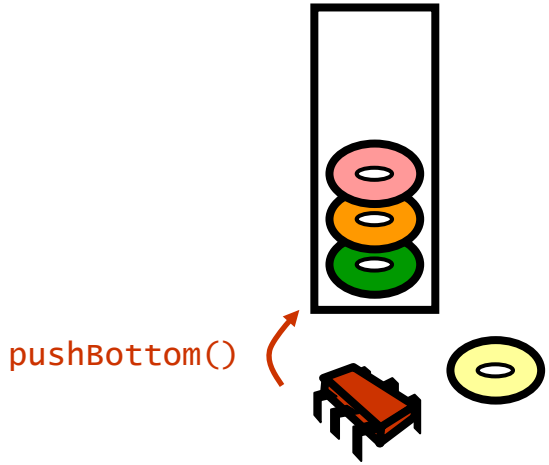
- Each work pool is a double-ended queue



Managing Work

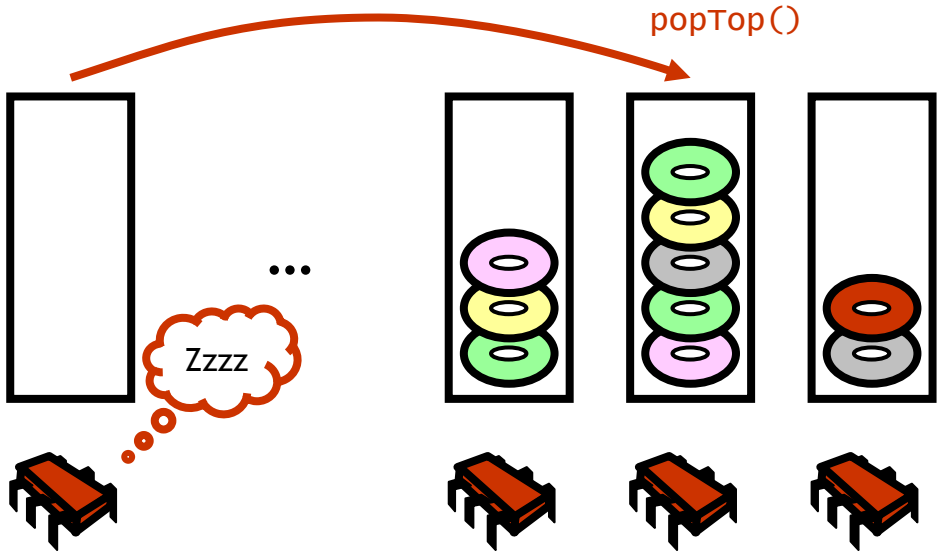
- Obtain work
  - Pop work from bottom
  - Run task until it blocks or terminates
- New work
  - Spawn node (create new task)
  - Push work from bottom





Stealing Work

- When the pool is empty, steal from others
  - Pick random victim
  - Take work from top



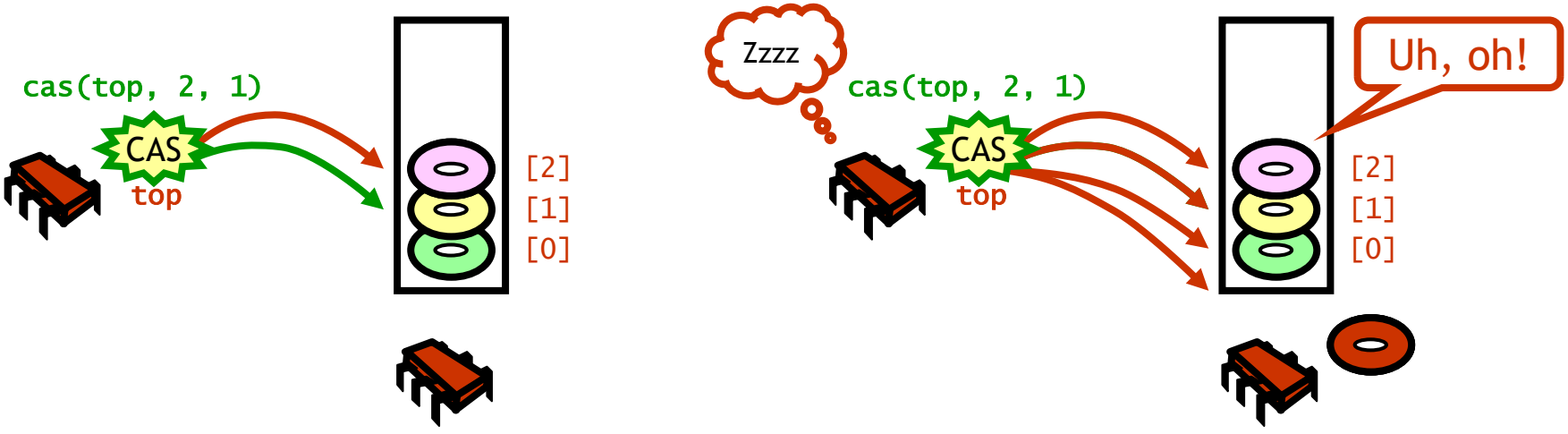
Double-Ended Queue

- Methods
  - `pushBottom()`
  - `popBottom()`
  - `popTop()`

  } Called by owner thread (never happen concurrently)
- Ideally
  - Wait-free, linearizable, constant time
- Method `popTop()` may signal abort if
  - Concurrent `popTop()` succeeds
  - Concurrent `popBottom()` takes last work

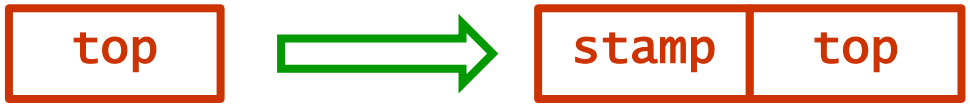
ABA Problem

- To pop, CAS the top pointer down
  - Concurrent pops detected as pointer moved
- What if pointer gets back to former value?



Fix for ABA Problem

- Add "stamp" (counter) to top pointer
  - Stamp incremented each time pointer is modified
  - CAS checks both stamp and top pointer
    - Fails if stamps does not match
  - Is ABA still possible?



Bounded Double-Ended Queue

```
public class BDEQueue {
  AtomicStampedReference<Integer> top;
  volatile int bottom;
  Runnable[] tasks;

  void pushBottom(Runnable r) {
    tasks[bottom] = r;
    bottom++;
  }
  …
}
```

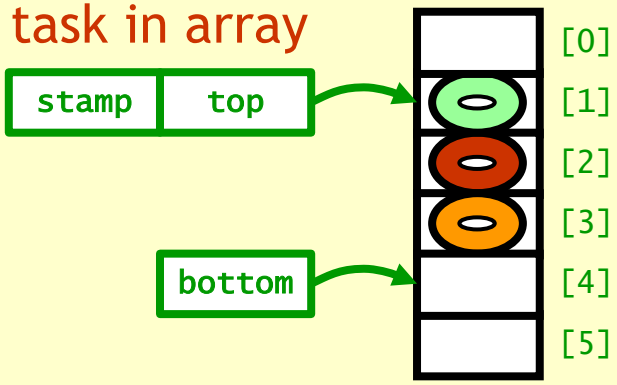
Index & stamp (synchronized)

Index of bottom thread (no need to synchronize but memory barrier)

Array holding tasks

Store new task in array

Adjust bottom index



Steal Work

```
public Runnable popTop() {
  int[] stamp = new int[1];
  int oldTop = top.get(stamp);
  int oldStamp = stamp[0];
  if (bottom <= oldTop)
    return null;
  int newTop = oldTop + 1;
  int newStamp = oldStamp + 1;
  Runnable r = tasks[oldTop];
  if (top.CAS(oldTop, newTop, oldStamp, newStamp))
    return r;
  return null;
}
```

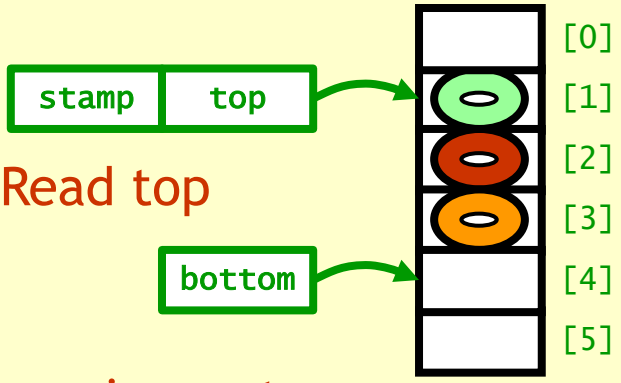
Read top

Quit if queue is empty

Compute new value & stamp

Try to steal the task

Give up if conflict occurs



Take Work

```
Runnable popBottom() {
  if (bottom == 0) return null;
  Runnable r = tasks[--bottom];
  int[] stamp = new int[1];
  int oldTop = top.get(stamp);
  int oldStamp = stamp[0];
  if (bottom > oldTop) return r;
  if (bottom == oldTop) {
    bottom = 0;
    if (top.CAS(oldTop, 0, oldStamp, oldStamp + 1))
      return r;
  }
  top.set(0, oldStamp + 1);
  return null;
}
```

Make sure queue non-empty

Grab bottom task

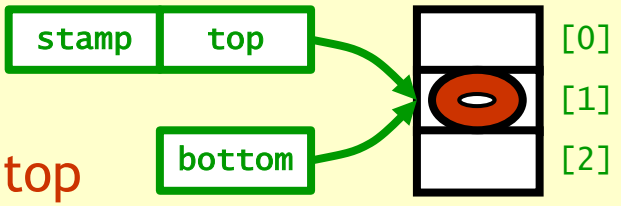
Read top

Top & bottom >= 1 apart: OK

At most 1 item left

Try to take last item (in any case reset bottom as queue will be empty even if unsuccessful)

Failed to get last item: must still reset top



Work Stealing

- Clean separation between application and scheduling layer
- Works well when number of processors fluctuates
- Works on “black-box” operating systems