Futures, Scheduling, and Work Distribution

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

(Some images in this lecture courtesy of Charles Leiserson)
How to write Parallel Apps?

• Split a program into parallel parts
• In an effective way
• Thread management
Matrix Multiplication

\[ (C) = (A) \cdot (B) \]
Matrix Multiplication

\[ c_{ij} = \sum_{k=0}^{N-1} a_{ki} b_{jk} \]
Matrix Multiplication

\[ c_{ij} = \sum_{k=0}^{N-1} a_{ki} \uparrow b_{jk} \]
Matrix Multiplication

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

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

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

Which matrix entry to compute
Matrix Multiplication

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

```java
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();
}
```
Matrix Multiplication

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();
}
Matrix Multiplication

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

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();
}
Thread Overhead

- Threads Require resources
  - Memory for stacks
  - Setup, teardown
  - Scheduler overhead
- Short-lived threads
  - Ratio of work versus overhead bad
Thread Pools

- More sensible to keep a pool of
  - long-lived threads
- Threads assigned short-lived tasks
  - Run the task
  - Rejoin pool
  - Wait for next assignment
Thread Pool = Abstraction

- Insulate programmer from platform
  - Big machine, big pool
  - Small machine, small pool

- Portable code
  - Works across platforms
  - Worry about algorithm, not platform
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>**

```java
Callable<T> task = ...;
...
Future<T> future = executor.submit(task);
...
T value = future.get();
```
Submitting a `Callable<T>` task returns a `Future<T>` object
Future\(<T>\)

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

The Future’s \textbf{get()} method blocks until the value is available
Runnable task = …;
...
Future<?> future = executor.submit(task);
...
future.get();
Submitting a `Runnable` task returns a `Future<?>` object
Future<?>

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

future.get();

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

- Executor Service submissions
  - Like New England traffic signs
  - Are purely advisory in nature
- The executor
  - Like the Boston driver
  - Is free to ignore any such advice
  - And could execute tasks sequentially …
Matrix Addition

\[
\begin{pmatrix}
C_{00} & C_{00} \\
C_{10} & C_{10}
\end{pmatrix} = 
\begin{pmatrix}
A_{00} + B_{00} & B_{01} + A_{01} \\
A_{10} + B_{10} & A_{11} + B_{11}
\end{pmatrix}
\]
Matrix Addition

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

4 parallel additions
Matrix Addition Task

class AddTask implements Runnable {
    Matrix a, b; // add this!
    public void run() {
        if (a.dim == 1) {
            c[0][0] = a[0][0] + b[0][0]; // base case
        } else {
            (partition a, b into half-size matrices a_{ij} and b_{ij})
            Future<?> f_{00} = exec.submit(addTask(a_{00}, b_{00}));
            ...
            Future<?> f_{11} = exec.submit(addTask(a_{11}, b_{11}));
            f_{00}.get(); ...; f_{11}.get();
            ...
        }
    }
}
Matrix Addition Task

class AddTask implements Runnable {
  Matrix a, b; // add this!
  public void run() {
    if (a.dim == 1) {
      c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
      // partition a, b into half-size matrices a_{ij} and b_{ij}
      Future<?> f_{00} = exec.submit(addTask(a_{00}, b_{00})), ...
      Future<?> f_{11} = exec.submit(addTask(a_{11}, b_{11}));
      f_{00}.get(); ...; f_{11}.get();
    }
  }
}
Matrix Addition Task

class AddTask implements Runnable {
    Matrix a, b; // add this!
    public void run() {
        if (a.dim == 1) {
            c[0][0] = a[0][0] + b[0][0]; // base case
        } else {
            // partition a, b into half-size matrices a_{ij} and b_{ij}
            Future<?> f_{00} = exec.submit(addTask(a_{00}, b_{00}));
            ...
            Future<?> f_{11} = exec.submit(addTask(a_{11}, b_{11}));
            f_{00}.get(); ...; f_{11}.get();
            ...
        }
    }
}
Matrix Addition Task

class AddTask implements Runnable {
    Matrix a, b; // add this!
    public void run() {
        if (a.dim == 1) {
            c[0][0] = a[0][0] + b[0][0]; // base case
        } else {
            (partition a, b into half-size matrices a_{ij} and b_{ij})
            Future<?> f_{00} = exec.submit(addTask(a_{00}, b_{00}));
            ...
            Future<?> f_{11} = exec.submit(addTask(a_{11}, b_{11}));
            f_{00}.get(); ...; f_{11}.get();
            ...
        }
    }
}
Matrix Addition Task

class AddTask implements Runnable {
    Matrix a, b; // multiply this!
    public void run() {
        if (a.dim == 1) {
            c[0][0] = a[0][0] + b[0][0]; // base case
        } else {
            (partition a, b into half-size matrices a_{ij} and b_{ij})
            Future<?> f_{00} = exec.submit(addTask(a_{00}, b_{00}));
            ...
            Future<?> f_{11} = exec.submit(addTask(a_{11}, b_{11}));
            f_{00}.get(); ...; f_{11}.get();
            ...
        }
    }
}
Dependencies

• Matrix example is not typical
• Tasks are independent
  – Don’t 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
Disclaimer

• This Fibonacci implementation is
  – Egregiously inefficient
    • So don’t try this at home or job!
  – But illustrates our point
    • How to deal with dependencies

• Exercise:
  – Make this implementation efficient!
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;
        }
    }
}
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;
        }
    }
}
Multithreaded Fibonacci

```java
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;
        }
    }
}
```

Pick up & combine results
The Blumofe-Leiserson DAG Model

• Multithreaded program is
  – A directed acyclic graph (DAG)
  – That unfolds dynamically

• Each node is
  – A single unit of work
Fibonacci DAG

fib(4)
Fibonacci DAG

fib(3) → fib(4)
Fibonacci DAG

fib(4)

fib(3)

fib(2)

fib(2)
Fibonacci DAG

- fib(4)
  - fib(3)
    - fib(2)
      - fib(1)
      - fib(1)
  - fib(2)
    - fib(1)
    - fib(1)
  - fib(1)
    - fib(1)
    - fib(1)
Fibonacci DAG

call

get

fib(4)  

fib(3)  

fib(2)  

fib(1)  

fib(2)  

fib(1)  

fib(1)  

fib(1)
How Parallel is That?

• Define work:
  – Total time on one processor

• Define critical-path length:
  – Longest dependency path
  – Can’t beat that!
Unfolded DAG
Serial fraction = \( \frac{3}{18} = \frac{1}{6} \) …

Amdahl’s Law says speedup cannot exceed 6.
Work?

$T_1$: time needed on one processor

Just count the nodes ....

$T_1 = 18$
Critical Path?

\[ T_\infty : \text{time needed on as many processors as you like} \]
Critical Path?

$T_\infty$: time needed on as many processors as you like

Longest path …. 

$T_\infty = 9$
Notation Watch

- $T_P = \text{time on } P \text{ processors}$
- $T_1 = \text{work (time on 1 processor)}$
- $T_\infty = \text{critical path length (time on } \infty \text{ processors)}$
Simple Laws

• **Work Law**: \( T_P \geq T_1/P \)
  – In one step, can’t do more than \( P \) work

• **Critical Path Law**: \( T_P \geq T_\infty \)
  – Can’t beat infinite resources
Performance Measures

• *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)
  – \( T_1 / T_\infty \)
Sequential Composition

Work: $T_1(A) + T_1(B)$

Critical Path: $T_\infty (A) + T_\infty (B)$
Parallel Composition

Work: $T_1(A) + T_1(B)$

Critical Path: $\max\{T_\infty(A), T_\infty(B)\}$
Matrix Addition

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

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

4 parallel additions
Addition

• 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 4 spawned additions

\[ A_1(n) = 4 A_1(n/2) + \Theta(1) \]

Partition, synch, etc
Addition

• Work is

\[ A_1(n) = 4 \cdot A_1(n/2) + \Theta(1) \]

\[ = \Theta(n^2) \]

Same as double-loop summation
Addition

- Critical Path length is

\[ A_\infty(n) = A_\infty(n/2) + \Theta(1) \]

spawned additions in parallel

Partition, synch, etc
Addition

- Critical Path length is

\[ A_\infty(n) = A_\infty(n/2) + \Theta(1) \]
\[ = \Theta(\log n) \]
Matrix Multiplication Redux

\[(C) = (A) \bullet (B)\]
Matrix Multiplication Redux

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
= 
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\cdot 
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]
First Phase …

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
= \begin{pmatrix}
A_{11}B_{11} & + & A_{12}B_{21} \\
A_{21}B_{11} & + & A_{22}B_{21}
\end{pmatrix}
+ \begin{pmatrix}
A_{11}B_{12} & + & A_{12}B_{22} \\
A_{21}B_{12} & + & A_{22}B_{22}
\end{pmatrix}
\]

8 multiplications
Second Phase ...

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
= \begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} \\
A_{21}B_{11} + A_{22}B_{21}
\end{pmatrix}
\begin{pmatrix}
A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
\]

4 additions
Multiplication

• Work is

$M_1(n) = 8 \cdot M_1(n/2) + A_1(n)$

8 parallel multiplications

Final addition
Multiplication

• Work is

\[ M_1(n) = 8 \cdot M_1(n/2) + \Theta(n^2) \]
\[ = \Theta(n^3) \]

Same as serial triple-nested loop
Multiplication

- Critical path length is

\[ M_\infty(n) = M_\infty(n/2) + A_\infty(n) \]

Half-size parallel multiplications

Final addition
Multiplication

- Critical path length is

\[
M_\infty(n) = M_\infty(n/2) + A_\infty(n)
= M_\infty(n/2) + \Theta(\log n)
= \Theta(\log^2 n)
\]
Parallelism

- $M_1(n)/M_\infty(n) = \Theta(n^3/\log^2 n)$
- To multiply two 1000 x 1000 matrices
  - $1000^3/10^2 = 10^7$
- 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 & go dynamically
Ideal Scheduling Hierarchy

- Tasks
  - User-level scheduler
- Processors
Realistic Scheduling Hierarchy

Tasks

User-level scheduler

Threads

Kernel-level scheduler

Processors
For Example

• Initially,
  – All $P$ processors available for application

• Serial computation
  – Takes over one processor
  – Leaving $P-1$ for us
  – Waits for I/O
  – We get that processor back ….
Speedup

• Map threads onto $P$ processes
• Cannot get $P$-fold speedup
  – What if the kernel doesn’t cooperate?
• Can try for speedup proportional to $P$
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 $i$
Greedy Scheduling

- A node is *ready* if predecessors are done
- *Greedy*: schedule as many ready nodes as possible
- *Optimal* scheduler is greedy (why?)
- But not every greedy scheduler is optimal
Greedy Scheduling

There are $P$ processors

Complete Step:
- $>P$ nodes ready
- run any $P$

Incomplete Step:
- $< P$ nodes ready
- run them all
Theorem

For any greedy scheduler,

\[ T_P \leq T_1/P + T_\infty \]
Theorem

For any greedy scheduler,

$T_P \leq \frac{T_1}{P} + T_\infty$

Actual time
Theorem

For any greedy scheduler,

$$T_P \leq \frac{T_1}{P} + T_\infty$$

No better than work divided among processors
Theorem

For any greedy scheduler,

\[ T_P \leq \frac{T_1}{P} + T_\infty \]

No better than critical path length
Proof:

Number of *incomplete steps* ≤ $\frac{T_1}{P}$ …

… because each performs $P$ work.

Number of *complete steps* ≤ $T_\infty$ …

… because each shortens the unexecuted critical path by 1

QED
Near-Optimality

*Theorem:* any greedy scheduler is within a factor of 2 of optimal.

*Remark:* Optimality is NP-hard!
Proof of Near-Optimality

Let $T_P^*$ be the optimal time.

$T_P^* \geq \max\{T_1/P, T_\infty\}$

$T_P \leq T_1/P + T_\infty$

$T_P \leq 2 \max\{T_1/P, T_\infty\}$

$T_P \leq 2 T_P^*$

From work and critical path laws

Theorem

QED
Work Distribution
Work Dealing

Yes!
The Problem with Work Dealing

D’oh!
Work Stealing

Yes!

No work…
Lock-Free Work Stealing

• Each thread has a pool of ready work
• Remove work without synchronizing
• If you run out of work, steal someone else’s
• Choose victim at random
Local Work Pools

Each work pool is a Double-Ended Queue
Work DEQueue\(^1\)

1. Double-Ended Queue

- pushBottom
- popBottom

\{ work \}
Obtain Work

- Obtain work
- Run task until
- Blocks or terminates

popBottom
New Work

- Unblock node
- Spawn node

pushBottom
Whatcha Gonna do When the Well Runs Dry?
Steal Work from Others

Pick random thread’s DEQqueue
Steal this Task!

popTop
Task DEQueue

• Methods
  – pushBottom
  – popBottom
  – popTop

  \{ \text{Never happen concurrently} \}
Task DEQueue

- Methods
  - pushBottom
  - popBottom
  - popTop

Most common – make them fast (minimize use of CAS)
Ideal

• Wait-Free
• Linearizable
• Constant time

Fortune Cookie: “It is better to be young, rich and beautiful, than old, poor, and ugly”
Compromise

• Method `popTop` may fail if
  – Concurrent `popTop` succeeds, or a
  – Concurrent `popBottom` takes last task
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded Problem
Dreaded ABA Problem

Uh-Oh …

CAS

Yes!
Fix for Dreaded ABA
Bounded DEQueue

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

Bounded DEQueue

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

Index & Stamp (synchronized)
Bounded DEQueue

```java
public class BDEQueue {
    AtomicStampedReference<Integer> top;
    volatile int bottom;
    Runnable[] deq;
    ...
}

index of bottom task no need to synchronize memory barrier needed
```
Bounded DEQueue

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

Array holding tasks
public class BDEQueue {
    ...
    void pushBottom(Runnable r) {
        tasks[bottom] = r;
        bottom++;
    }
    ...
    }

pushBottom()
pushBottom()

```java
public class BDEQueue {
    ...
    void pushBottom(Runnable r) {
        tasks[bottom] = r;
        bottom++;
    }
    ...
}
```

Bottom is the index to store the new task in the array
public class BDEQueue {
    ...  
    void pushBottom(Runnable r){
        tasks[bottom] = r;
        bottom++;
    }
    ...
    }

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

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

Quit if queue is empty
public Runnable popTop() {
    int[] stamp = new int[1];
    int oldTop = top.get(stamp), newTop = oldTop + 1;
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    if (bottom <= oldTop)
        return null;
    Runnable r = tasks[oldTop];
    if (top.CAS(oldTop, newTop, oldStamp, newStamp))
        return r;
    return null;
}
public Runnable popTop() {
    int[] stamp = new int[1];
    int oldTop = top.get(stamp), newTop = oldTop + 1;
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    if (bottom <= oldTop)
        return null;
    Runnable r = tasks[oldTop];
    if (top.CAS(oldTop, newTop, oldStamp, newStamp))
        return r;
    return null;
}
Runnable popBottom() {
    if (bottom == 0) return null;
    bottom--;
    Runnable r = tasks[bottom];
    int[] stamp = new int[1];
    int oldTop = top.get(stamp), newTop = 0;
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    if (bottom > oldTop) return r;
    if (bottom == oldTop){
        bottom = 0;
        if (top.CAS(oldTop, newTop, oldStamp, newStamp))
            return r;
    }
    top.set(newTop, newStamp); return null;
    bottom = 0; }

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

Take Work

Make sure queue is non-empty
Runnable popBottom() {
    if (bottom == 0) return null;

    bottom--;
    Runnable r = tasks[bottom];

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

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

Take Work

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

Take Work

Try to steal last task.
Reset bottom because the DEQueue will be empty even if unsuccessful (why?)
Runnable popBottom() {
    if (bottom == 0) return null;
    bottom--;
    Runnable r = tasks[bottom];
    int[] stamp = new int[1];
    int oldTop = top.get(stamp), newTop = 0;
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    if (bottom > oldTop) return r;
    if (bottom == oldTop) {
        bottom = 0;
        if (top.CAS(oldTop, newTop, oldStamp, newStamp))
            return r;
    }
    top.set(newTop, newStamp); return null;
    bottom = 0;}

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

Take Work
Failed to get last task
(bottom could be less than top)
Must still reset top and bottom since deque is empty
Old English Proverb

• “May as well be hanged for stealing a sheep as a goat”
• From which we conclude:
  – Stealing was punished severely
  – Sheep were worth more than goats
Variations

• Stealing is expensive
  – Pay CAS
  – Only one task taken

• What if
  – Move more than one task
  – Randomly balance loads?
Work Balancing

\[
\left\lfloor 22 + 5 \right\rfloor / 2 = 4
\]

\[
\left\lfloor 2 + 5 \right\rfloor / 2 = 3
\]
public void run() {
    int me = ThreadID.get();
    while (true) {
        Runnable task = queue[me].deq();
        if (task != null) task.run();
        int size = queue[me].size();
        if (random.nextInt(size+1) == size) {
            int victim = random.nextInt(queue.length);
            int min = ..., max = ...;
            synchronized (queue[min]) {
                synchronized (queue[max]) {
                    balance(queue[min], queue[max]);
                }
            }
        }
    }
}
Work-Balancing Thread

```java
public void run() {
    int me = ThreadID.get();
    while (true) {
        Runnable task = queue[me].deq();
        if (task != null) task.run();
        int size = queue[me].size();
        if (random.nextInt(size+1) == size) {
            int victim = random.nextInt(queue.length);
            int min = ..., max = ...;
            synchronized (queue[min]) {
                synchronized (queue[max]) {
                    balance(queue[min], queue[max]);
                }
            }
        }
    }
}
```

Keep running tasks
Work-Balancing Thread

```java
public void run() {
    int me = ThreadID.get();
    while (true) {
        Runnable task = queue[me].deq();
        if (task != null) task.run();
        int size = queue[me].size();
        if (random.nextInt(size+1) == size) {
            int victim = random.nextInt(queue.length);
            int min = ..., max = ...;
            synchronized (queue[min]) {
                synchronized (queue[max]) {
                    balance(queue[min], queue[max]);
                }
            }
        }
    }
}
```

With probability \(\frac{1}{|\text{queue}|}\)
public void run() {
    int me = ThreadID.get();
    while (true) {
        Runnable task = queue[me].deq();
        if (task != null) task.run();
        int size = queue[me].size();
        if (random.nextInt(size + 1) == size) {
            int victim = random.nextInt(queue.length);
            int min = ..., max = ...;
            synchronized (queue[min]) {
                synchronized (queue[max]) {
                    balance(queue[min], queue[max]);
                }
            }
        }
    }
}
public void run() {
    int me = ThreadID.get();
    while (true) {
        Runnable task = queue[me].deq();
        if (task != null) task.run();
        int size = queue[me].size();
        if (Math.random().nextInt(size + 1) == size) {
            int victim = Math.random().nextInt(queue.length);
            int min = ..., max = ...;
            synchronized (queue[min]) {
                synchronized (queue[max]) {
                    balance(queue[min], queue[max]);
                }
            }
        }
    }
}

Lock queues in canonical order
public void run() {
    int me = ThreadID.get();
    while (true) {
        Runnable task = queue[me].deq();
        if (task != null) task.run();
        int size = queue[me].size();
        if (random.nextInt(size+1) == size) {
            int victim = random.nextInt(queue.length);
            int min = ..., max = ...
            synchronized (queue[min]) {
                synchronized (queue[max]) {
                    balance(queue[min], queue[max]);
                }
            }
        }
    }
}
Work Stealing & Balancing

- Clean separation between app & scheduling layer
- Works well when number of processors fluctuates.
- Works on “black-box” operating systems
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - [http://creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/).
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.
TOM MARVOLO RIDDLE