Exploiting Multiprocessors in Java

Doug Lea

State University of New York at Oswego

`dl@cs.oswego.edu`

`http://gee.cs.oswego.edu`

Outline

Improving service performance

- **Architectures, forces, options**

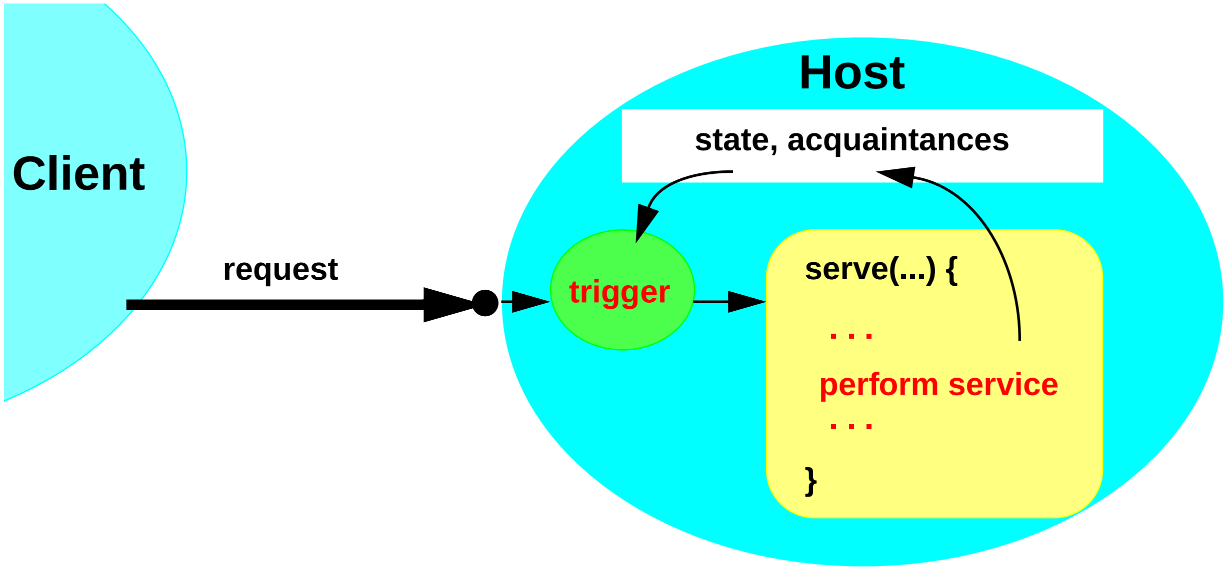
Delegation

- **Worker Threads**

Decomposition

- **Fork/Join designs**

Improving Service Performance



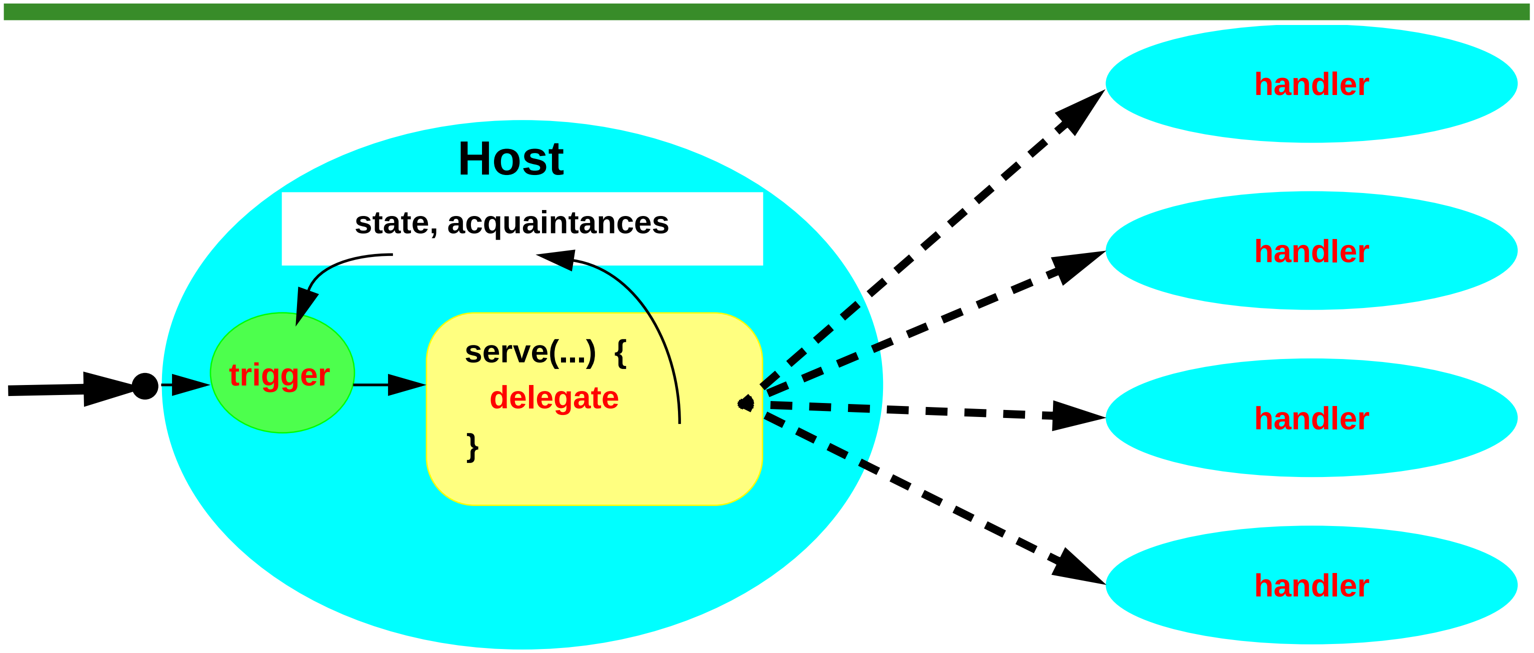
Performance Goals

- **Availability (latency)**
  - **Maximize message acceptance rate**
- **Throughput**
  - **Minimize service times**

Basic Approach

- **Offload tasks to objects running in other threads**

Asynchronously Delegated Services



Hosts are reactive

Of form: `for(;;) { accept and dispatch a request; }`

Handlers are task-based

Of form: `processOneRequest(...)`

Can improve availability and throughput

Host object can quickly respond to next message

Multiple handlers exploit parallelism

Major Design Forces

Problem decomposition

- Maximizing parallelism
- Exploiting multiple CPUs, overlapping IO

Resource management

- Minimizing overhead
- Avoiding resource exhaustion

Concurrency control

- Obeying message semantics; scheduling
- Maintaining safety, liveness

Lead to two sets of patterns, surrounding:

- Dispatching to handlers in other threads
- Breaking up and managing Tasks

Delegation using Open Calls

Event-based programming style:

- **Safely update local state (holding locks)**
- **Issue call to delegate (without holding locks)**

```
class Host { //...
  final Handler handler;

  public void serve(...) {
    updateState(...);
    handler.process(...);
  }

  synchronized void updateState(...) {
    // ...
  }
}
```

Reduces Host as bottleneck, but does not introduce any concurrency

Thread-per-Request Delegation

```
class Host { //...
  final Handler handler;

  public void serve(...) {
    updateState(...);

    Runnable task = new Runnable() { // wrap
      public void run() {
        handler.process(...);
      }
    };

    new Thread(task).start();        // run
  }

  synchronized void updateState(...) {
    // ...
  }
}
```

Messages and Tasks in Java

Direct method invocations

- **Rely on standard call/return mechanics**

Command strings

- **Recipient parses then dispatches to underlying method**
- **Widely used in client/server systems including HTTP**

`EventObjects` and service codes

- **Recipient dispatches**
- **Widely used in GUIs, including AWT**

`Request` objects, asking to perform encoded operation

- **Used in distributed object systems — RMI and CORBA**

`Class` objects (normally via `.class` files)

- **Recipient creates instance of class**
- **Used in Java Applet framework**

`Runnable` commands

- **Basis for thread instantiation, mobile code systems**

Sample Socket-based Server

```
class Server implements Runnable {
  public void run() {
    try {
      ServerSocket socket = new ServerSocket(PORT);
      for (;;) {
        final Socket connection = socket.accept();
        new Thread(new Runnable() {
          public void run() {
            new Handler().process(connection);
          }}).start();
      }
    }
    catch(Exception e) { /* cleanup; exit */ }
  }
}

class Handler {
  void process(Socket s) {
    InputStream i = s.getInputStream();
    OutputStream o = s.getOutputStream();
    // decode and service request, handle errors
    s.close();
  }
}
```

Thread-per-Request Characteristics

+ Simple semantics

- **When in doubt, make a new thread**

- Potentially high overhead

- **Thread start-up overhead impedes host availability**
- **Higher context switch and scheduling overhead**

- Little or no resource or scheduling control

- **Potential resource exhaustion**
- **Live with default saturation characteristics**

Alternative designs can be attractive even on JVMs where overhead is relatively low

Worker Threads

Establish a producer-consumer chain

- Producer
  - Service method just places task in a channel
    - Channel might be a buffer, queue, stream, etc
    - Task might be represented by a `Runnable` command, event, etc
- Consumer
  - Host contains an autonomous loop thread of form:

```
while (!Thread.interrupted()) {
  task = channel.take();
  process(task);
}
```

Worker Thread Example

```
interface Channel { // buffer, queue, stream, etc
  void   put(Object x);
  Object take();
}

class Host { //...
  Channel channel = ...;
  public void serve(...) {
    channel.put(new Runnable() { // enqueue
      public void run(){
        handler.process(...);
      }});
  }

  Host() {   // Set up worker thread in constructor
    // ...
    new Thread(new Runnable() {
      public void run() {
       while (!Thread.interrupted())
        ((Runnable)(channel.take())).run();
      }
    }).start();
  }
}
```

Channel Options

Unbounded queues

- **Can exhaust resources if clients faster than handlers**

Bounded buffers

- **Can cause clients to block when full**

Synchronous channels

- **Force client to wait for handler to complete previous task**

Leaky bounded buffers

- **For example, drop oldest if full**

Priority queues

- **Run more important tasks first**

Streams or sockets

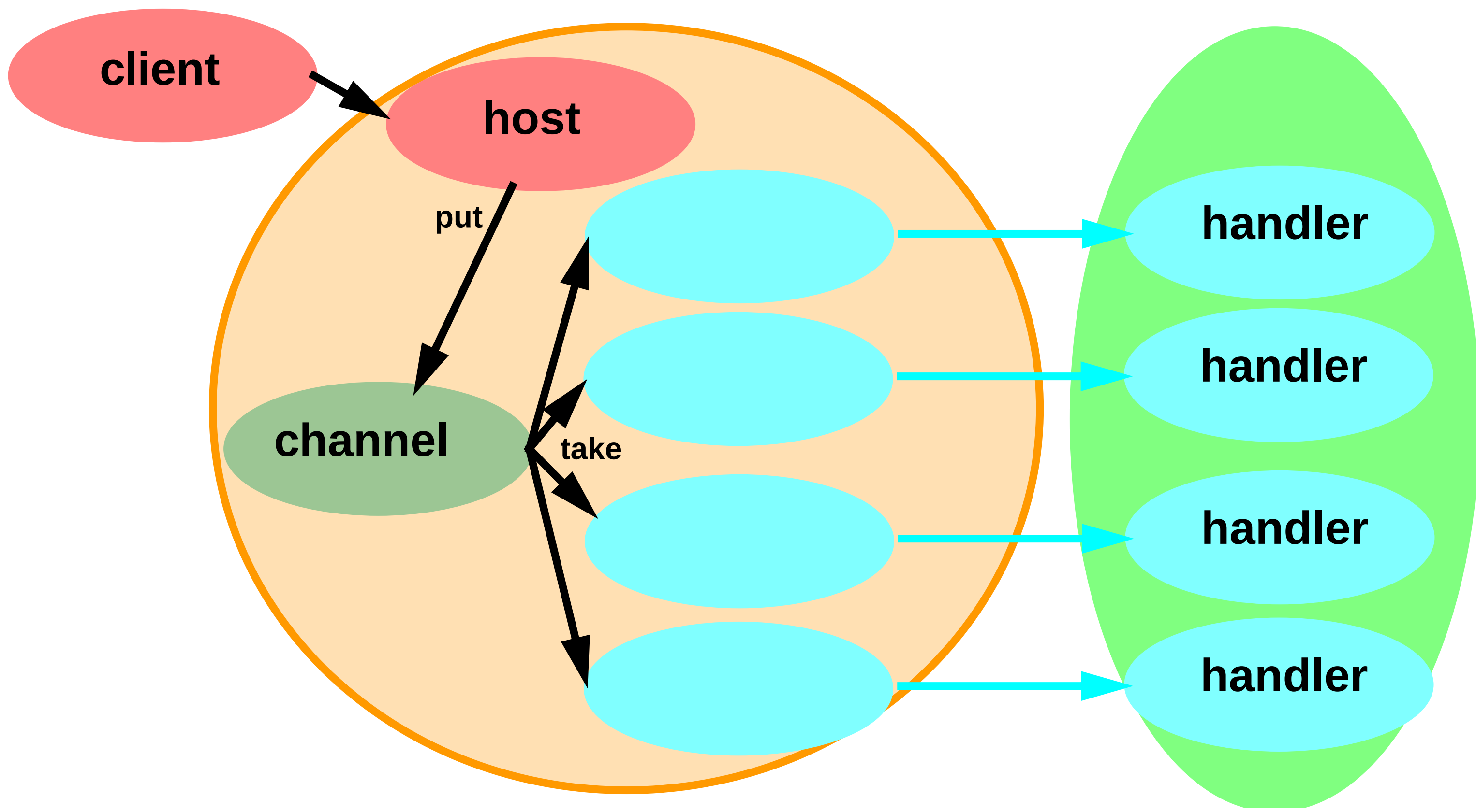
- **Enable persistence, remote execution**

Non-blocking channels

- **Must take evasive action if `put` or `take` fail or time out**

Thread Pools

Use a collection of worker threads, not just one



In simplest cases, set up via a loop in host constructor

But normally, encapsulate as Pool class

Worker Thread Characteristics

+ Tunable semantics and structure

- Somewhat greater coding complexity

    - Requires some Java-level duplication of VM services

+ Less delegation overhead

    - Create and hand off task object instead of new Thread

- May require more work to maintain liveness

    - Queued tasks do not run
    - Need to implement saturation policies

+ Enables bounding of resource usage

    - Can match resource usage to platform characteristics

- May waste threads

- May violate assumptions equating activities with `Threads`

    - Need caution with class `java.lang.ThreadLocal`
    - Can mask locking errors since Java locks are per-thread

Default Worker Thread Pool Policies

Need conservative default policies

- Choose alternatives only when sure you can do better

No queuing

- Avoid lockups due to queued tasks not running
- Usually, the VM can schedule better than you can
- Requires:
  - Synchronous channels
  - Relatively large maximum pool bounds

Run-when-blocked saturation policy

- If cannot immediately hand off, host runs task itself
- Usually, the most graceful degradation policy

Dynamic worker thread management

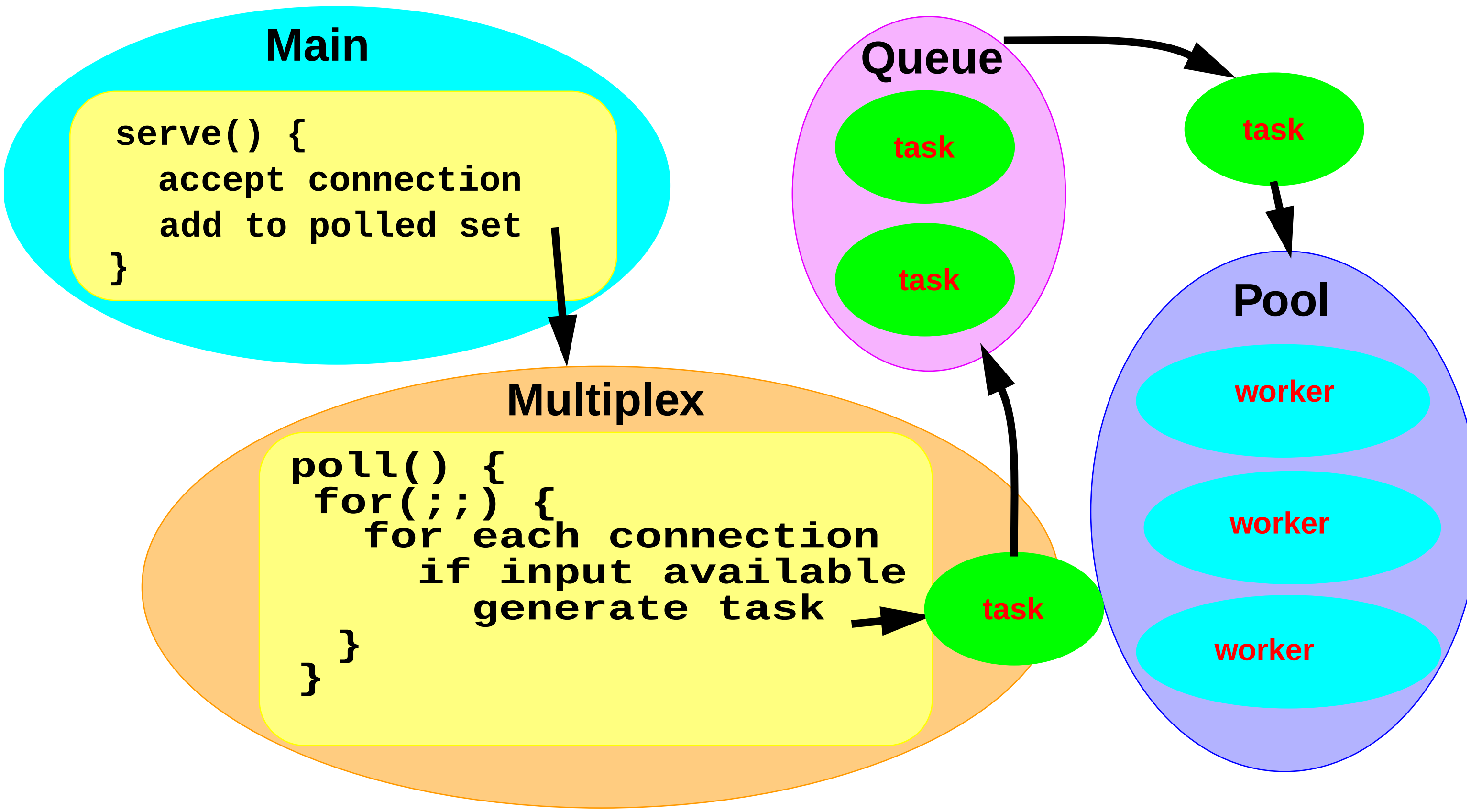
- Lazy construction
- Allow worker threads to die if idle longer than time-to-live threshold. Lazily replace with others if needed later

Pools in Connection-Based Designs

Increasingly common architecture:

- **Many open connections (sockets), but relatively few active at any given time**
- **Service tasks triggered by input on connection**

Multiplex the delegations to worker threads via polling



Event-Driven Tasks

```
class IOEventTask implements Runnable {
  final Socket socket;
  final InputStream input;
  volatile boolean done = false;

  IOEventTask(Socket s) throws IOException {
    socket = s; input = socket.getInputStream();
  }

  public void run() {
   if (done) return;
   byte[] commandBuffer = new byte[BUFFSIZE];
   try {
      int bytes = input.read(commandBuffer, 0, BUFFSIZE);
      if (bytes != BUFFSIZE) done = true;
      else processCommand(commandBuffer, bytes);
    }
    catch (IOException ex) { cleanup(); done = true; }
    finally {
      if (!done) return;
      try { input.close(); socket.close(); }
      catch(IOException ignore) {}
    }
  }
}
```

Parallel Decomposition

Goal: Minimize service times by exploiting parallelism

Approach:

Partition into subproblems

Break up main problem into several parts. Each part should be as independent as possible.

Create subtasks

Construct each solution to each part as a `Runnable` task.

Fork subtasks

Feed subtasks to pool of worker threads. Base pool size on number of CPUs or other resource considerations.

Join subtasks

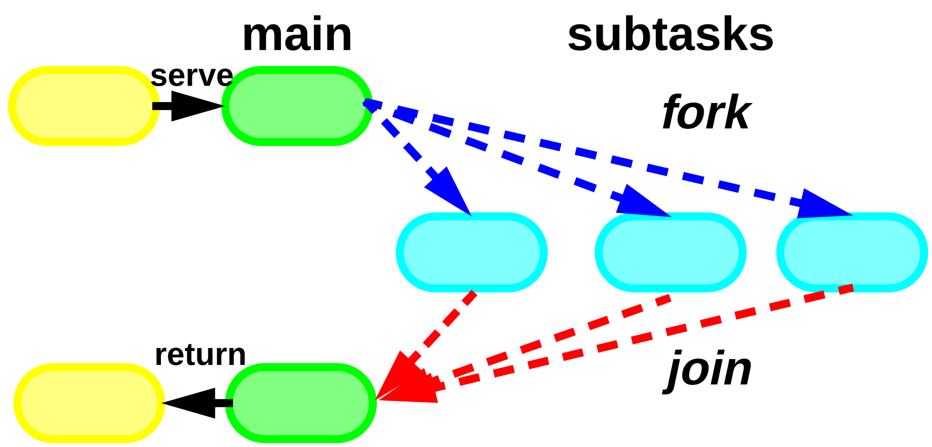
Wait out processing of as many subtasks (usually all) needed to compose solution

Compose solution

Compose overall solution from completed partial solutions. (aka *reduction*, *agglomeration)*

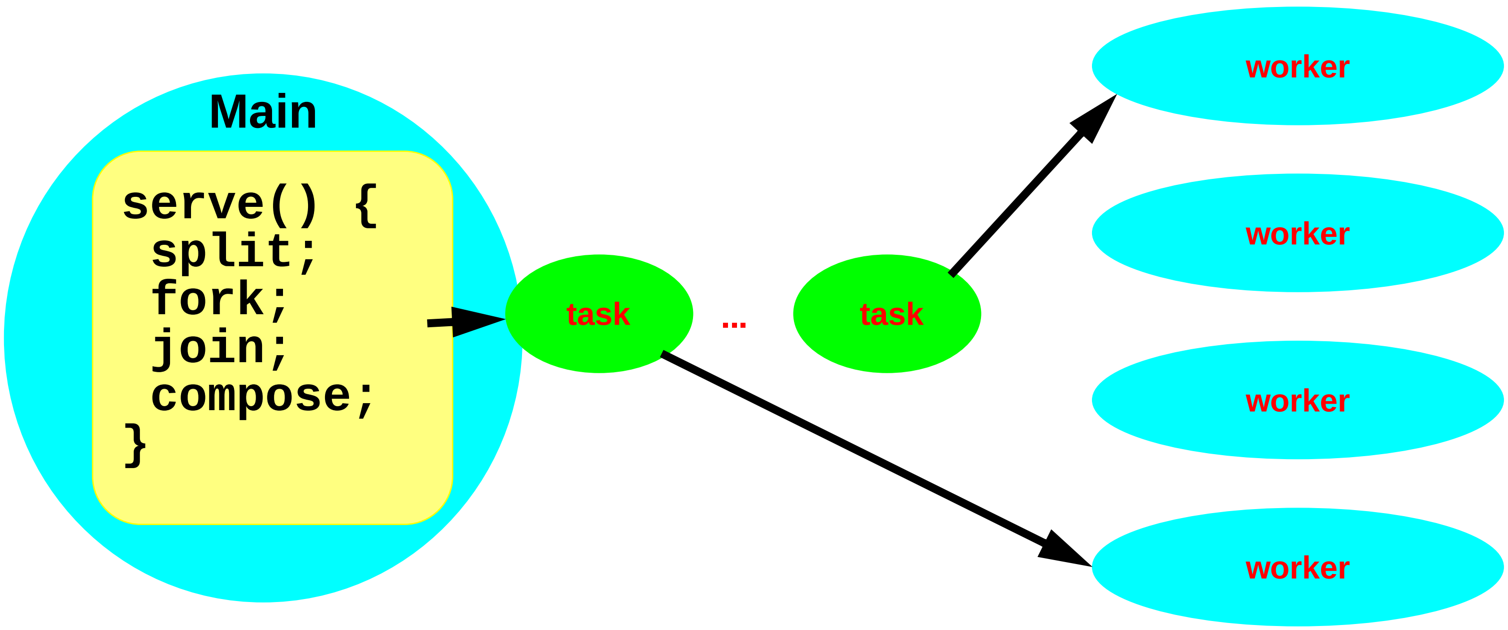
Fork/Join Parallelism

Main task must help synchronize and schedule subtasks




```
public Result serve(Problem problem) {
  SPLIT the problem into parts;

  FORK:
    for each part p
      create and start task to process p;

  JOIN:
    for each task t
      wait for t to complete;

  COMPOSE and return aggregate result;
}
```

Fork/Join with Worker Threads



Similar advantages and disadvantages as before

But further opportunities to improve performance

- **Exploit simple scheduling properties of fork/join**
- **Exploit simple structure of decomposed tasks**

Granularity

How big should each task be?

- **Approaches and answers differ for different kinds of tasks**
  - **Computation-intensive, I/O-intensive, Event-intensive**
- **Focus here on computation-intensive**

Two opposing forces:

- **To maximize parallelism, make each task as small as possible**
  - **Improves load-balancing, locality, decreases percentage of time that CPUs idly wait for each other, and leads to greater throughput**
- **To minimize overhead, make each task as large as possible**
  - **Creating, enqueing, dequeing, executing, maintaining status, waiting for, and reclaiming resources for Task objects add overhead compared to direct method calls.**

Must adopt an engineering compromise:

- **Use special-purpose low-overhead Task frameworks**
- **Use parameterizable decomposition methods that rely on sequential algorithms for small problem sizes**

A Task Framework

Fork/Join Task objects can be much lighter than `Thread` objects

- **No blocking except to join subtasks**
  - **Tasks just run to completion**
  - **Cannot enforce automatically, and short-duration blocking is OK anyway.**
- **Only internal bookkeeping is completion status bit.**
- **All other methods relay to current worker thread.**

```
abstract class FJTask implements Runnable {
  boolean isDone();      // True after task is run
  void fork();           // Start a dependent task
  static void yield(); // Allow another task to run
  void join();                // Yield until isDone
  static void invoke(Task t);    // Directly run t
  static void coInvoke(Task t,Task u);// Fork+join
  static void coInvoke(Task[] v); // Fork+join all
  void reset();                      // Clear isDone
  void cancel();                     // Force isDone
} // (plus a few others)
```

Fork/Join Worker Thread Pools

Uses per-thread queuing with work-stealing

- **Normally best to have one worker thread per CPU**
  - **But design is robust. It scarcely hurts (and sometimes scarcely helps) to have more workers than CPUs**
- **Each new task is queued in current worker thread's dequeue (double-ended queue)**
  - **Plus a global entry queue for new tasks from clients**
- **Workers run tasks from their own dequeues in stack-based LIFO (i.e., newest task first) order.**
- **If a worker is idle, it steals a task, in FIFO (oldest task first) order from another thread's dequeue or entry queue**

Work-Stealing

Original algorithm devised in Cilk project (MIT)

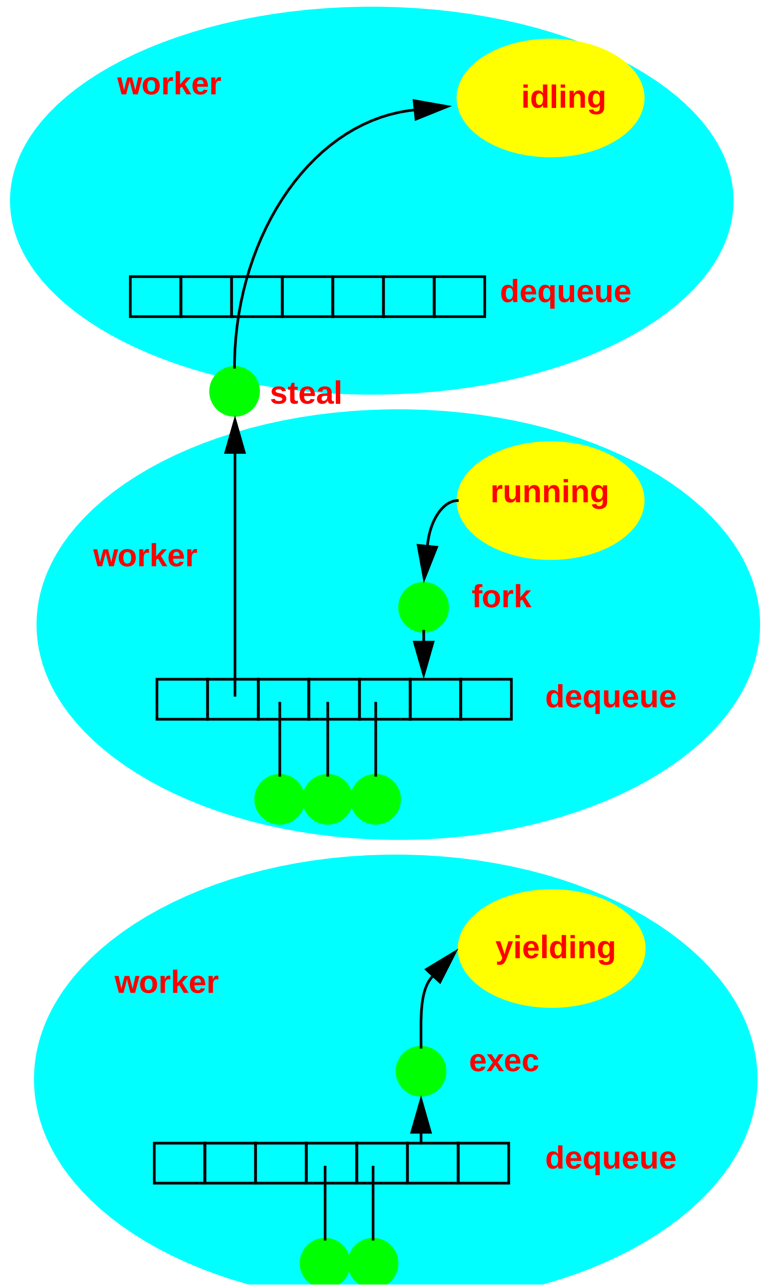
- Several variants
- Shown to scale on stock MP hardware

Leads to very portable application code

Typically, the only platform-dependent parameters are:

- Number of worker threads
- Problem threshold size for using sequential solution

Works best with recursive decomposition



Recursive Decomposition

Typical algorithm:

```
Result solve(Param problem) {
  if (problem.size <= GRANULARITY_THRESHOLD)
    return directlySolve(problem);
  else {
     in-parallel {
       Result l = solve(lefthalf(problem));
       Result r = solve(rightHalf(problem);
     }
     return combine(l, r);
  }
}
```

Why?

- **Support tunable granularity thresholds**
- **Under work-stealing, the algorithm itself drives the scheduling**
- **There are known recursive decomposition algorithms for many computationally-intensive problems.**
  - **Some are explicitly parallel, others are easy to parallelize**

Example: Fibonacci

A useless algorithm, but easy to explain!

Sequential version:

```
int seqFib(int n) {
  if (n <= 1)
    return n;
  else
    return seqFib(n-1) + seqFib(n-2);
}
```

To parallelize:

- **Replace function with Task subclass**
  - **Hold arguments/results as instance vars**
  - **Define `run()` method to do the computation**
- **Replace recursive calls with fork/join Task mechanics**
  - **`Task.coinvoke` is convenient here**
- **But rely on sequential version for small values of n**

  Threshold value usually an empirical tuning constant

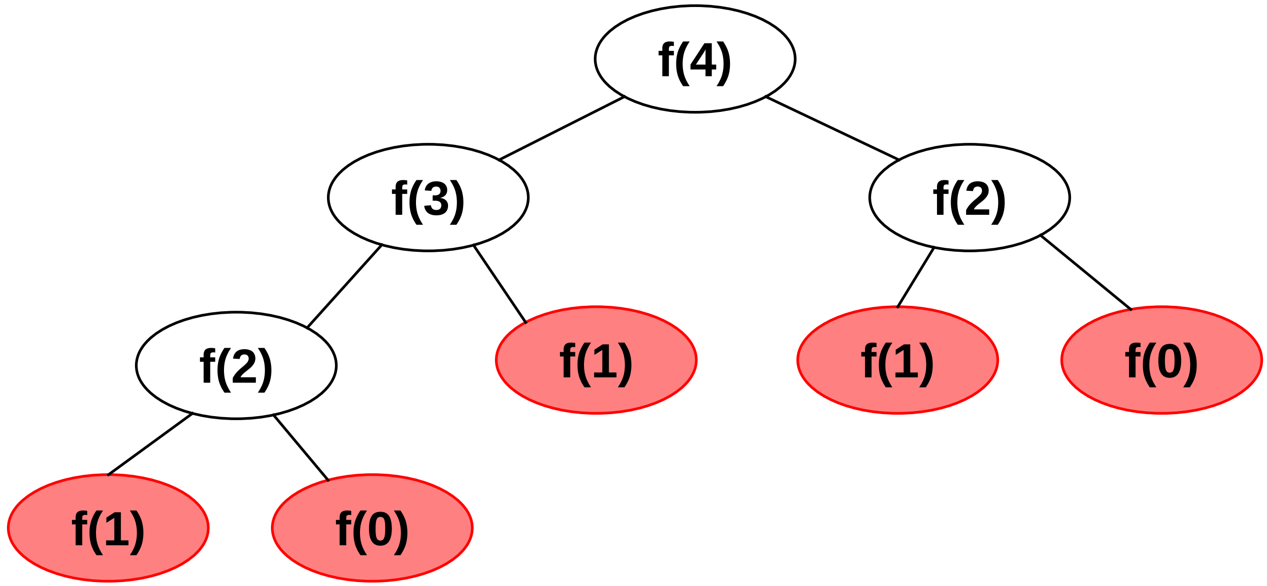
Class Fib

```
class Fib extends Task {
  volatile int number; // serves as arg and result
  Fib(int n) { number = n; }

  public void run() {
    int n = number;
    if (n <= 1) { /* do nothing */ }
    else if (n <= sequentialThreshold) //(12 works)
      number = seqFib(n);
    else {
      Fib f1 = new Fib(n - 1);         // split
      Fib f2 = new Fib(n - 2);
      coInvoke(f1, f2);                // fork+join
      number = f1.number + f2.number; // compose
    }
  }

  int getAnswer() { // call from external clients
    if (!isDone())
      throw new Error("Not yet computed");
    return number;
  }
}
```

Fib Server

```
public class FibServer  { // Yes. Very silly
  public static void main(String[] args) {
    TaskRunnerGroup group = new
      TaskRunnerGroup(Integer.parseInt(args[0]));
    ServerSocket socket = new ServerSocket(1618);
    for (;;) {
      final Socket s = socket.accept();
      group.execute(new Task() {
        public void run() {
          DataInputStream i = new
            DataInputStream(s.getInputStream());
          DataOutputStream o = new
            DataOutputStream(s.getOutputStream());
          Fib f = new Fib(i.readInt());
          invoke(f);
          o.writeInt(f.getAnswer());
          s.close()
      });
      }
    }
  }
} // (Lots of exception handling elided out)
```

Computation Trees



Recursive computation meshes well with work-stealing:

- **With only one worker thread, computation proceeds in same order as sequential version**
  - **The local LIFO rule is same as, and not much slower than recursive procedure calls**
- **With multiple threads, other workers will typically steal larger, non-leaf subtasks, which will keep them busy for a while without further inter-thread interaction**

Iterative Computation

Many computation-intensive algorithms have structure:

Break up problem into a set of tasks, each of form:

- **For a fixed number of steps, or until convergence, do:**
  - **Update one section of a problem;**
  - **Wait for other tasks to finish updating their sections;**

Examples include mesh algorithms, relaxation, physical simulation

Illustrate with simple Jacobi iteration, with base step:

```
void oneStep(double[][] oldM, double[][] newM,
             int i, int j) {
   newM[i][j] = 0.25 * (oldM[i-1][j] +
                        oldM[i][j-1] +
                        oldM[i+1][j] +
                        oldM[i][j+1]);
}
```

Where `oldM` and `newM` alternate across steps

Iteration via Computation Trees

Explicit trees avoid repeated problem-splitting across iterations

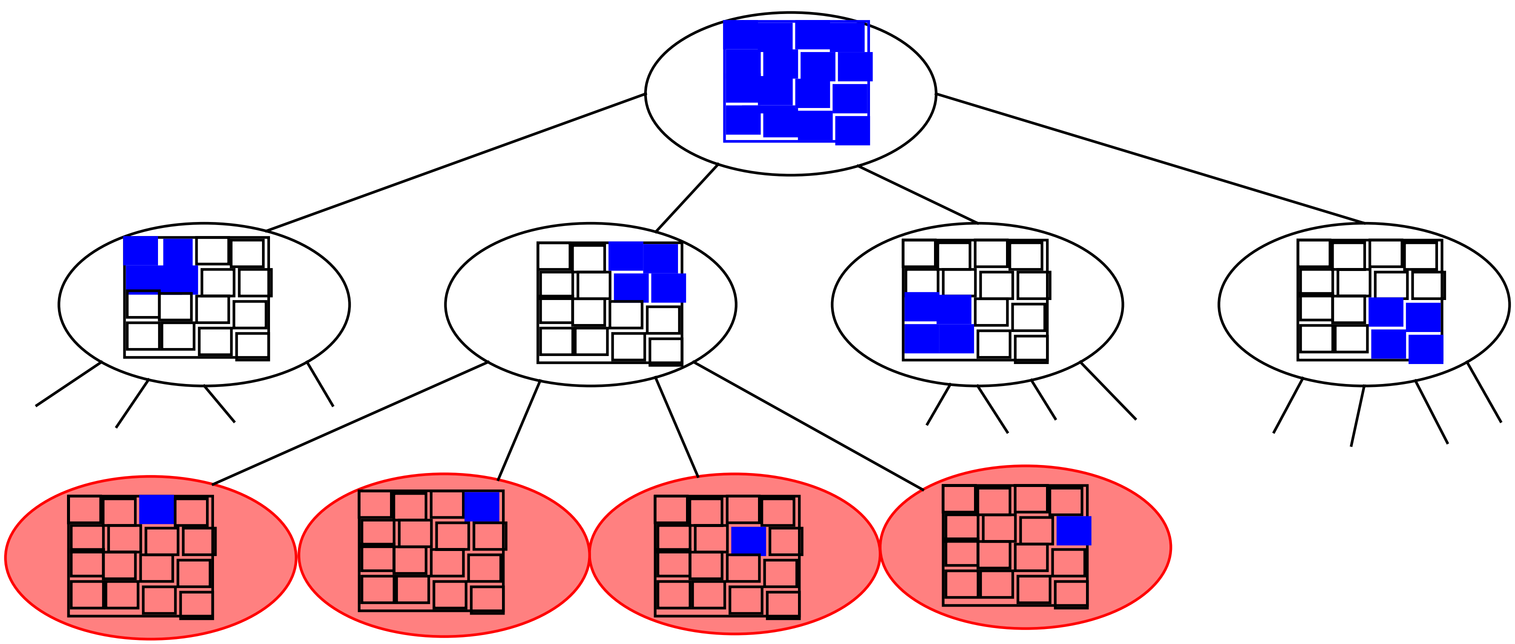
Allow Fork/Join to be used instead of barrier algorithms

For Jacobi, can recursively divide by quadrants

- Leaf nodes do computation;

  Leaf node size (cell count) is granularity parameter

- Interior nodes drive task processing and synchronization



Jacobi example

```
abstract class Tree extends Task {
  volatile double maxDiff; //for convergence check
}


class Interior extends Tree {
  final Tree[] quads;

  Interior(Tree q1, Tree q2, Tree q3, Tree q4) {
    quads = new Tree[] { q1, q2, q3, q4 };
  }

  public void run() {
    coInvoke(quads);
    double md = 0.0;
    for (int i = 0; i < 4; ++i) {
      md = Math.max(md,quads[i].maxDiff);
      quads[i].reset();
    }
    maxDiff = md;
  }
}
```

Leaf Nodes

```
class Leaf extends Tree {
  final double[][] A; final double[][] B;
  final int loRow; final int hiRow;
  final int loCol; final int hiCol; int steps = 0;
  Leaf(double[][] A, double[][] B,
       int loRow, int hiRow,
       int loCol, int hiCol) {
    this.A = A;   this.B = B;
    this.loRow = loRow; this.hiRow = hiRow;
    this.loCol = loCol; this.hiCol = hiCol;
  }
  public synchronized void run() {
    boolean AtoB = (steps++ % 2) == 0;
    double[][] a = (AtoB)? A : B;
    double[][] b = (AtoB)? B : A;
    for (int i = loRow; i <= hiRow; ++i) {
      for (int j = loCol; j <= hiCol; ++j) {
        b[i][j] = 0.25 * (a[i-1][j] + a[i][j-1] +
                          a[i+1][j] + a[i][j+1]);
        double diff = Math.abs(b[i][j] - a[i][j]);
        maxDiff = Math.max(maxDiff, diff);
      }
    }
} }
```

Driver


```
class Driver extends Task {
  final Tree root;  final int maxSteps;
  Driver(double[][] A, double[][] B,
         int firstRow, int lastRow,
         int firstCol, int lastCol,
         int maxSteps, int leafCells) {
    this.maxSteps = maxSteps;
    root = buildTree(/* ... */);
  }

  Tree buildTree(/* ... */) { /* ... */}

  public void run() {
    for (int i = 0; i < maxSteps; ++i) {
      invoke(root);
      if (root.maxDiff < EPSILON) {
        System.out.println("Converged");
        return;
      }
      else
        root.reset();
    }
  }
}
```

Framework Performance

Extremely well-tuned to fork/join designs (only!)

- Fork/join only 4 to 10 times slower than direct call
- Can run about 2 million minimal tasks per second per CPU on 300Mhz sparc
- Requires high-quality garbage collection.
- Supports task granularities of < 1000 instructions without noticeably degrading performance on uniprocessors
  - This is conveniently in the range where the use of special parallelization tools would not be especially helpful. Instead rely on conformance to common decomposition patterns

Impossible to obtain this performance for class `Thread` itself.

- No matter how fast `Threads` are, it is still attractive to build lighter-weight special-purpose executable frameworks.
- Unless standardized versions of these lightweight executable frameworks are also supported.

Sample results

Enterprise 3500, 8x336 CPUs, Solaris 7, Production VM 1.2.1

Tests:

- **Fib 40, Multiply 1024x1024 matrix, Sort 40 million ints, Jacobi with 100 iterations on 2048x2048 matrix**

Times in seconds to nearest tenth, medians of 3 runs

Threads	Fib	MatMul	Sort	Jacobi
1	21.5	40.7	79.8	111.8
2	10.7	20.4	39.7	56.6
3	7.2	13.6	27.0	38.6
4	5.4	10.3	20.2	29.4
5	4.4	8.2	16.3	24.1
6	3.6	6.9	13.8	20.5
7	3.1	5.9	11.9	18.0
8	2.9	5.2	10.7	16.5

Summary

Scalable service designs rely on

- **Reactive hosts**
- **Task-based Delegation**
- **Task-based Decomposition**
- **Resource-conscious programming**
- **Scalable infrastructure — VM, OS, hardware**