CS6235 - Analysis of Parallel Programs

Introduction

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Written assignment = 1 x 10 marks.
- Programming assignments = 2 x 10 marks,
- Paper reading / Project = 10 marks.
- Quiz 1 = 15 marks, Quiz 2 = 15 marks, Final = 30 marks.
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a good question, answer a chosen question, make a good point! Take 0.5 marks each. Max one mark per day per person.
- Attendance requirement – as per institute norms. Non compliance will lead to 'W' grade.
  - Proxy attendance - is not a help; actually a disservice.
- Plagiarism - A good word to know. A bad act to own.
  - Will be automatically referred to the institute welfare and disciplinary committee.

Contact (Anytime) :
Instructor: Krishna, Email: nvk@iitm.ac.in, Office: BSB 352.
TA : Ramya Kasaraneni (cs19d003@smail)

What, When and Why of Program Analysis

- **What**:
  - A process of automatic analysis of computer programs regarding different program properties.
- **How?**
  - Analyze the program with or without executing!
- **Why? Study?**
  - Give guarantees about the correctness of program optimization, effectiveness of program optimization, program safety, and so on.
  - Used by compilers, debuggers, verifiers, IDEs, profilers.
  - A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer.
  - Handy, if you care about the programs you write!

Flavors of Program Analyses

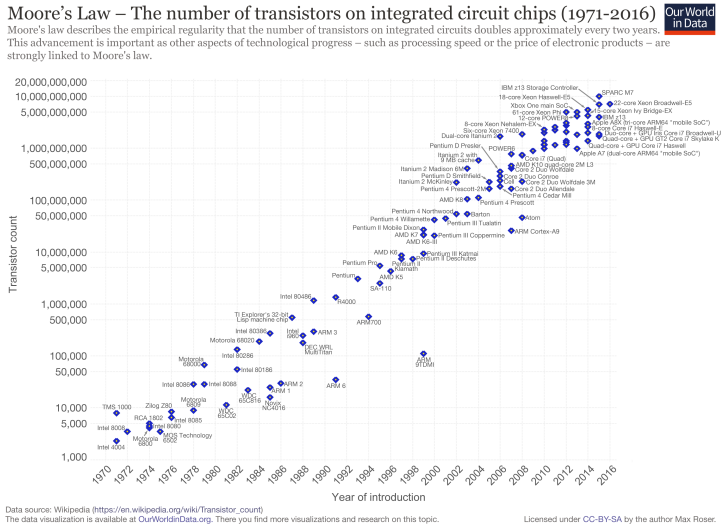
You have a goal? I have an analysis.

1.	Constant Replacement	Constant propagation
2.	Method Inlining	Points-to analysis
3.	Remove Null Pointer checks	Points-to analysis
4.	Loop parallelization	Dependence analysis
5.	Remove array out of bounds checks	Bounds check
6.	Debugging	Program slice
7.	Register allocation	Liveness analysis
8.	Find-definition (IDE)	Def-use analysis
9.	Dead-code elimination	Reaching-definition analysis
10.	Program-safety	Type checking
11.	Barrier elimination	MHP Analysis
12.	Race Detection	MHP Analysis

Why Parallel Program Analysis?

- Parallel systems have become mainstay (Why? - holdon).
- Automatic Extraction of parallelism has not been very successful.
- The community is looking at writing parallel programs.
- Analysis of parallel programs is a natural consequence.

Why Parallel Systems / Multicores?



Focus on increasing the number of computing cores.

What, When Multicores? **Why not** Multiprocessors

- **What** A multi-core processor is composed of two or more independent cores. Composition involves the interconnect, memory, caches.
- **When** IBM POWER4, the world's first dual-core processor, released in 2001.
- **Why not Multi-processors**
  - An application can be "threaded" across multiple cores, but not across multi-CPUs – communication across multiple CPUs is fairly expensive.
  - Some of the resources can be shared. For example, on Intel Core Duo: L2 cache is shared across cores, thereby reducing further power consumption.
  - Less expensive: A single CPU board with a dual-core CPU Vs a dual board with 2 CPUs.

Course outline

A rough outline (we may not strictly stick to this).

- Parallel programming constructs basics.
- Program analysis basics
- Parallel program representation
- MHP analysis and its impact on traditional analysis
- Parallel-Program Specific analysis
- Advanced Topics (depending on time).

Your friends: Languages and Tools

Start exploring

- Java - familiarity a must - Use eclipse to save you valuable coding and debugging cycles.
- JavaCC, JTB – tools you will learn to use.
- Make Ant Scripts – recommended toolkit.
- Find the course webpage:
  http://www.cse.iitm.ac.in/∼krishna/cs6235/

Get set. Ready steady go!

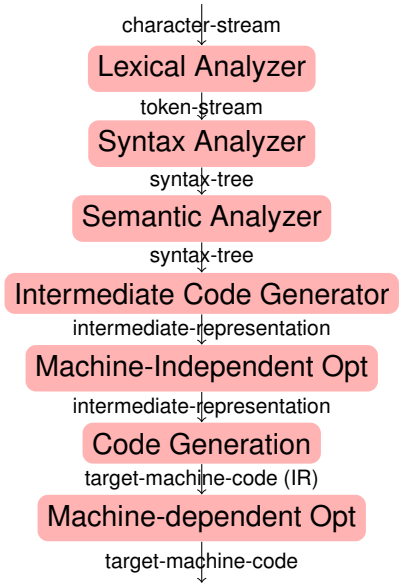
Expectations

What qualities are important in a program analysis?

1. Should identify properties correctly.
2. Should analyze all valid programs.
3. Analysis runs fast
4. Analysis time proportional to program size
5. Support for modular analysis.
6.

Each of these shapes your expectations about this course

Phases inside the compiler



Front end responsibilities:

- Recognize syntactically legal code; report errors.
- Recognize semantically legal code; report errors.
- Produce IR.

Back end responsibilities:

- Optimizations, code generation.

Program Analysis:

- After parsing.

Parallel Programs:

- Impacts both FE and BE

Examples of how Parallelism Impacts Analysis I/III

```
function int Withdraw(int amount){
     if (balance > amount) {
           balance = balance - amount;
           return SUCCESS;
     }
     return FAIL;
}
```

- Say `balance` = 100.
- Two parallel threads executing `Withdraw`(80)
- At the end of the execution, it may so happen that both of the withdrawals are successful. Further `balance` can still be 20!

Race freedom is enough?

```
void deposit(int amt) {
   acquire(m);
   balance = balance+amt;
   release(m);
}
int read_balance() {
   int t;
   acquire(m);
   t = balance;
   release(m);
   return t;
}
```

```
int withdraw(int amt) {
   int t = read_balance();
   acquire(m);
   if (t <= amt) {
      balance = 0;
   } else {
      balance = balance-amt;
      t = amt;
   }
   release(m);
   return t;
}
```

```
// Initial balance = 10.
fork  withdraw(10); ;      // Thread 1
fork  deposit(10); ;       // Thread 2
```

Example taken from Flanagan and Qadeer TLDI 2003.

Examples of how Parallelism Impacts Analysis II/III

```
for (int i = ...) {
  X[f(i)] = ... ;
  async { ... = X[g(i)]; }
}
```

$\Longrightarrow$ // Legal transformation?

```
// After loop distribution
for (int i = ...)
  X[f(i)] = ... ;
for (int i = ...)
    async { ... = X[g(i)]; }
```

Outline

1 Introduction
- Formalities
- Overview
- Parallelism and its impact
- Introduction Parallel constructs
- Java Concurrency

Sources of speedups in Parallel Programs

- Say a serial Program P takes T units of time.
- Q: How much time will the best parallel version P' take (when run on N number of cores)? $\frac{T}{N}$ units?
- Linear speedups is almost unrealizable, especially for increasing number of compute elements.
- $T_{total} = T_{setup} + T_{compute} + T_{finalization}$
- T_{setup} and $T_{finalization}$ may not run concurrently - represent the execution time for the non-parallelizable parts of code.
- Best hope : $T_{compute}$ can be fully parallelized.
- $T_{total}(N) = T_{setup} + \frac{T_{compute}}{N} + T_{finalization}$ ..... (1)
- Speedup $S(N) = \frac{T_{total}(1)}{T_{total}(N)}$. In practice?
- Chief factor in performance improvement : Serial fraction of the code.

Amdahl's Law

- Serial fraction $\gamma = \frac{T_{setup}+T_{finalization}}{T_{total}(1)}$
- Fraction of time spent in parallelizable part = $(1-\gamma)$

$$T_{total}(N) = \underbrace{\gamma \times T_{total}(1)}_{\text{serial code}} + \underbrace{\frac{(1-\gamma)\times T_{total}(1)}{N}}_{\text{parallel code}}$$

$$= \left(\gamma + \frac{1-\gamma}{N}\right) \times T_{total}(1)$$

$$\text{Speedup } S(N) = \frac{T_{total}(1)}{\left(\gamma+\frac{1-\gamma}{N}\right)\times T_{total}(1)}$$

$$= \frac{1}{\left(\gamma+\frac{1-\gamma}{N}\right)}$$

$$\approx \frac{1}{\gamma} \quad \ldots \text{Amdahl's Law}$$

- Max speedup is inversely proportional to the serial fraction of the code.

Implications of Amdahl's law

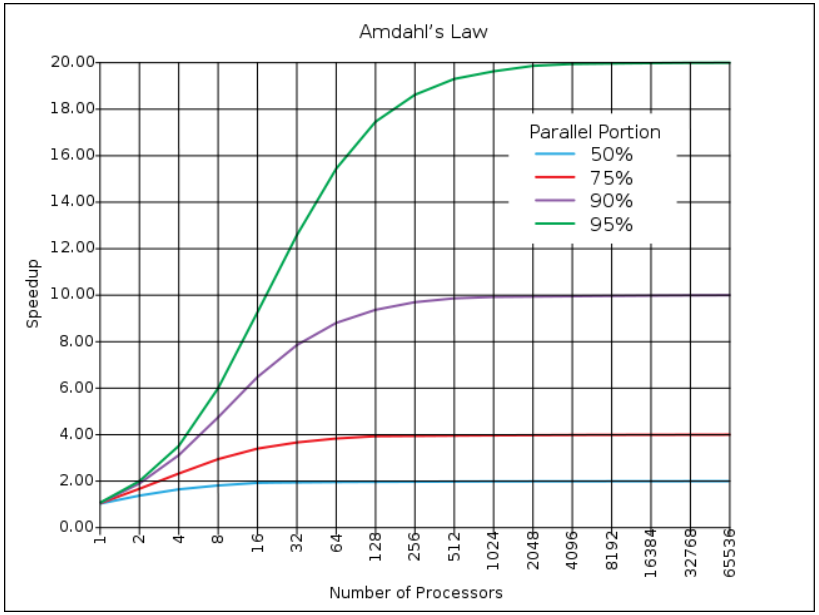
Assume: Ten processors. Goal: 10 fold speedup.

Parallel fraction	Serial fraction	Speedup = $\frac{1}{\left(\gamma+\frac{1-\gamma}{N}\right)}$
40 %	60 %	2.17
20 %	80 %	3.57
10 %	90 %	5.26
99 %	01 %	9.17

Implications of Amdahl's law

- As we increase the number of parallel compute units, the speed up need not increase - an upper limit on the usefulness of adding more parallel execution units.
- For a given program maximum speedup nearly remains a constant.
- Say a parallel program spends only 10% of time in parallelizable code. If the code is fully parallelized, as we aggressively increase the number of cores, the speedup will be capped by ($\sim$) 1.11$\times$.
- Say a parallel program spends only 10% of time in parallelizable code. Q: How much time would you spend to parallelize it?
- Amdahl's law helps to set realistic expectations for performance gains from the parallelization exercise.
- Mythical Man-month - Essays on Software Engineering. Frederic Brooks.

Peaking via Amdahl's law



Limitations of Amdahl's law

- An over approximation : In reality many factors affect the parallelization and even fully parallelizable code does not result in linear speed ups.
- Overheads exist in parallel task creations/termination/synchronization.
- Does not say anything about the impact of cache - may result in much more or far less improvements.
- Dependence of the serial code on the parallelizable code - can the parallelization in result in faster execution of the serial code?
- Amdahl's law assumes that the problem size remains the same after parallelization: When we buy a more powerful machine, do we play only old games or new more powerful games?

Discussion: Amdahl's Law

- When we increase the number of cores - the problem size is also increased in practise.
- Also, naturally we use more and more complex algorithms, increased amount of details etc.
- Given a fixed problem, increasing the number of cores will hit the limits of Amdahl's law. However, if the problem grows along with the increase in the number of processors - Amdahl's law would be pessimistic
- Q: Say a program P has been improved to P' (increase the problem size) - how to keep the running time same? How many parallel compute elements do we need?

Example of how Parallelism Impacts Analysis III/III

```
A = new int[n]; // initialized to 0.
```

T1

```
for (i=1;i<n;++i){
  A[i] = i;
}
```

T2

```
for (j=1;j<n ;++j){
  assert(A[j] >= A[j+1]-1)
}
```

- Q: Can the computation loop be parallelized?
- Is it safe?

Outline

1. Introduction
   - Formalities
   - Overview
   - Parallelism and its impact
   - Introduction Parallel constructs
   - Java Concurrency

Concurrency in programs

	Processes	Threads	Tasks
1	A program in execution	Light weight process	sequence of instructions
2	Shared mem: 1 process/run	1 or more threads per process	one more tasks can be executed by a thread
3	Distributed mem: 1 or more processes	1 or more threads per process	one more tasks can be executed by a thread
4	**C**: `fork`	**Java:**`new Thread()` **C**: `pthread_create()`	**X10:** `async S`
5.	Does NOT share heap/stack	Shares Heap	Share stack + heap
6.	Scheduled by the OS	Scheduled by the runtime	Shared by the threads

Processes - Example

```
int *y;
void main(){
  int done = 0;
  y=calloc(1,4);
  printf("1. Before forking\n");
  if (fork() == 0){
    printf("2a. In the Child\n");
    done = 1;
    while (*y == 0);
    printf("2b. Ending the Child\n");
    exit(0);
  } else {
     printf("3a. After forking\n");
     while (!done) ;
     *y = 1;
     printf("3b. Before waiting\n");
     wait();
  }
  printf("4. Bye\n"); }
```

What is the output?

Threads - Example

```
int *y;
void main(){
int done = 0;
y=calloc(1,4);
printf("1. Before forking\n");
if (create_thread() == 0){ // hypothetical call
  printf("2a. In the Child\n");
  done = 1;
  while (*y == 0);
  printf("2b. Ending the Child\n");

} else {
   printf("3a. After creating thread\n");
   while (!done) ;
   *y = 1;
   printf("3b. Before waiting\n");
   wait();
}
printf("4. Bye\n"); }
```

What is the output?

Tasks - Example

```
int *y;
void main(){
  int done = 0;
  y=calloc(1, 4);
  printf("1. Before forking\n");
  async{ // create task
    printf("2a. In the Child\n");
    done = 1;
    while (*y == 0);
    printf("2b. Ending the Child\n");
  }
   printf("3a. After creating task\n");
   while (!done) ;
   *y = 1;
   printf("3b. Before waiting\n");
   wait();

   printf("4. Bye\n"); }
```

What is the output?

Operations on or by processes / threads /tasks

- Creation.
- Execute in parallel with each other.
- May communicate with each other (data / synchronization).
- Termination.

Outline

1. Introduction
   - Formalities
   - Overview
   - Parallelism and its impact
   - Introduction Parallel constructs
   - Java Concurrency

Java Threads Vs Processes

- Each instance of JVM creates a single process.
- Each process creates one or more threads.
  - Main thread creates the others.

Java Threads

- Each Java thread is an object - instance of the Java Thread class.
- An application that creates an instance of Thread must provide the code that will run in that thread.
  - `implement Runnable` interface.
  - Provide an implementation of the `run` method.

  or
  - `extend Thread` class
  - Provide an overridden implementation of the `run` method.

  Adv/Disadv??
  - (Hint) Java allows single inheritance.
  - Extending thread class ⇒ cannot extend any other class.

What can a thread do?

- Start executing the thread body specified in the `run` method.
- Sleep - `Thread.sleep(..)`
- Wait for child threads to finish: `ch.join()`
- Communicate with other threads.

Communication via synchronized methods

- `synchronized` statements and methods. Making a methods synchronized has two effects:
  - It is not possible for two invocations of synchronized methods on the same object to interleave.
  - When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object.
    - Guarantees that changes to the state of the object are visible to all threads.
- Constructors cannot be synchronized. Why? Consequence - only the object creating threads should call the constructor.
- synchronized methods ensure that there is no thread interference.
- Q: Too many synchronized methods. Disadv?

How do synchronized methods/statements work?

- Each object has an associated intrinsic lock.
- When a thread invokes a synchronized method,
  - automatically acquires the intrinsic lock for that method's object
  - releases the lock when the method returns (normal or via exception).
- What if a thread invokes a synchronized method recursively?

Updating shared variables

Java guarantees that following actions would be atomic.

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared volatile (including long and double variables).
- Any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.
- We can also declare a variable (of some types) as atomic.

Atomic variables

- The `java.util.concurrent.atomic` package defines classes supporting atomic operations on single variables.
- Supports many types of `get` and `set` operations.
- Like `volatile` variables' write operation, the `set` operation has an happens-before relation with the corresponding `get` operation.

Atomic variables - Example

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
                // guarantees thread non-interference.
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

Atomic variables (contd)

- Supported classes:

AtomicBoolean	AtomicInteger	AtomicIntegerArray
...	AtomicLong	AtomicLongArray
...	LongAccumulator	LongAdder

- All support: `boolean compareAndSet(expectedValue, updateValue)`
- CAS operation: How to use it to realize synchronization?
- Building blocks for implementing 'non-blocking' data structures.

Happens before relation

- Sequential order: Each action in a thread happens-before every action in that thread that comes later in the program's order.
- Unlock $\rightarrow$ Lock: An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor.
- Volatile writes: A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.
- A call to start on a thread happens-before any action in the started thread.
- All actions in a thread happen-before any other thread successfully returns from a join on that thread.

Happens-before relation is transitive.

Deadlock, Livelock and Starvation

- Deadlock: two or more threads are blocked forever, waiting for each other.
- Starvation: a thread is unable to gain regular access to shared resources and is unable to make progress.
- Livelock: threads are not blocked, but are not making any progress.

Guarded blocks

- A way to coordinate with others.
- A guarded block
  - polls a condition that has to be true to proceed.
  - other threads set that condition

```
public void guardedEntry(){
    while (!flag) ;
   // flag is set. Inefficient.
}
```

Guarded blocks (contd)

```
public synchronized void guardedEntry() {
    // Check once and "wait"
    while(!flag) { // Loop is needed.
        try {
            wait(); // releases the lock
        } catch (InterruptedException e) {}
    }
    // flag is set. Efficient.
}
```

- Q: Why synchronized?
- `notify` vs `notifyAll`

Immutable Objects

- An object is considered immutable if its state cannot change after it is constructed.
- Helps write reliable code.
- cannot be corrupted by thread interference or observed in an inconsistent state.
- Creating many immutable objects Vs updating existing objects.
  - Cost of object creation, GC
  - Code needed to protect mutable objects from corruption.

Example

```
public class SynchronizedRGB {
  private int red;   // between 0 - 255.
  private int green; // between 0 - 255.
  private int blue;  // between 0 - 255.
  private String name;

  public synchronized void set(int r, int g, int b,
                               String n) {..}
  public synchronized int getRGB() {
      return ((red << 16) | (green << 8) | blue);
  }
  public synchronized String getName() { return name; }
  public synchronized void invert() {
      red=255-red; green=255-green; blue=255-blue;
      name="Inverse of " + name;
  } }
```

Example (contd)

```
...
SynchronizedRGB color =
    new SynchronizedRGB(0, 0, 0, "Black");

int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

What if another threads updates the color object after Statement 1?

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

Such issues do not arise with immutable objects

Mutable to Immutable

General guidelines:

- Don't provide 'setter' methods.
- Make all fields final and private.
- Don't allow subclasses to override methods or provide 'setter' methods.
  - declare the class as final.
  - make constructor final and provide a factory method.
- If the instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't provide methods that modify the mutable objects.
  - Don't share references to the mutable objects. Copy and share if required.

Mutable to Immutable (Example)

```
final public class ImmutableRGB {
  final private int red;   // between 0 - 255.
  final private int green; // between 0 - 255.
  final private int blue;  // between 0 - 255.
  final private String name;

  public /*synchronized*/ int getRGB() {
        return ((red << 16) | (green << 8) | blue); }
  public /*synchronized*/ String getName()
      { return name; }
  public /*synchronized*/ ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                        255 - green, 255 - blue,
                        "Inverse of " + name);
    } }
```

No synchronized methods required!

Deadlocks in Locks

```
public class Deadlock {
 class Friend {
  private final String name;
  public Friend(String name){this.name = name; }
  public String getName() {return this.name; }
  public synchronized void bow(Friend bower) {
   System.out.format("%s: %s" +" has bowed to me!",
     this.name, bower.getName());
   bower.bowBack(this); }
  public synchronized void bowBack(Friend bower) {
   System.out.format("%s:%s"+" has bowed back!",
    this.name, bower.getName()); } }
```

Deadlocks in Locks (contd)

```
public static void main(String[] args) {
    final Friend alpha = new Friend("Alpha");
    final Friend beta = new Friend("Beta");
    new Thread(new Runnable() {
        public void run() { alpha.bow(beta); }
    }).start();
    new Thread(new Runnable() {
        public void run() { beta.bow(alpha); }
    }).start();
} }
```

Avoid deadlocks in Locks

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) { this.name=name;}

        public String getName() { return this.name;}
```

Avoid deadlocks in Locks (cont.)

```
public boolean impendingBow(Friend bower) {
    Boolean myLock = false;
    Boolean yourLock = false;
    try {
        myLock = lock.tryLock();
        yourLock = bower.lock.tryLock();
    } finally {
        if (! (myLock && yourLock)) {
            if (myLock) {
                lock.unlock();
            }
            if (yourLock) {
                bower.lock.unlock();
            } } }
    return myLock && yourLock;
}
```

Avoid deadlocks in Locks (cont.)

```
public void bow(Friend bower) {
    if (impendingBow(bower)) {
        try {
            System.out.format("%s: %s has"
                + " bowed to me!",
                this.name, bower.getName());
            bower.bowBack(this);
        } finally {
            lock.unlock();
            bower.lock.unlock(); }
    } else {
        System.out.format("%s: %s started"
            + " to bow to me, but saw that"
            + " I was already bowing to him.",
            this.name, bower.getName());
    } }
```

Avoid deadlocks in Locks (cont.)

```
    public void bowBack(Friend bower) {
        System.out.format("%s: %s has" +
            " bowed back to me!%n",
            this.name, bower.getName());
    }
}
```

Avoid deadlocks in Locks (cont.)

```
class BowLoop implements Runnable {
    private Friend bower;
    private Friend bowee;

    public BowLoop(Friend bower, Friend bowee) {
        this.bower = bower; this.bowee = bowee;}
    public void run() {
        Random random = new Random();
        for (;;) {
            try {
                Thread.sleep(random.nextInt(10));
            } catch (InterruptedException e) {}
            bowee.bow(bower);
        } } }
```

Avoid deadlocks in Locks (cont.)

```
    public static void main(String[] args) {
        final Friend alpha = new Friend("Alpha");
        final Friend beta = new Friend("Beta");
        new Thread(new BowLoop(alpha, beta)).start();
        new Thread(new BowLoop(beta, alpha)).start();
    }
}
```

Barriers in Java

- `CyclicBarrier`
  - allows a set of threads to wait for each other.
  - can be reused after the end of one phase.

```
class MainClass{
  final CyclicBarrier barrier;
  MainClass(){
    barrier = new CyclicBarrier(NumThreads,
                      new Runnable() {
                         public void run (){
                           execute-at-the-end-of-phase;
                         }});
   for(int i=0;i<NumThreads;++i){
     new Thread(new mThread().start());
   } } }
```

Barriers in Java (cont.)

```
class mThread implements Runnable {
  public void run(){
    while (not-done)
      do-some-computation;
      barrier.await();
      // may throw InterruptedExecution or
      // BrokenBarrierException
  }
}
```

- All or none breakage model.
- Memory consistency effects and happen before relations:

  `S1 Barrier{Sb} S2`
  `S3 Barrier{Sb} S4`

  `S1` and `S3` happen before `Sb` and `Sb` happens before `S2` and `S4`

Tasks and ThreadPools in Java

- ThreadPool: reuses previously created threads to execute current tasks
- Threads are created once and reused across all the tasks.
  - Less overhead.
  - When task/tasks arrive(s), the threads are ready.
  - Avoids resource thrashing caused by creating threads arbitrarily.

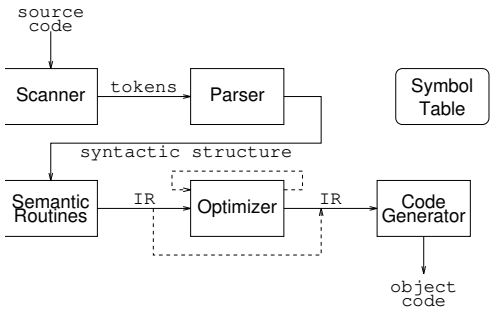
ThreadPools and Tasks example

```
class Task implements Runnable {
  public void run() {...}
}

class Main {
 public static void main(String[] args){
  Runnable r1 = new Task(..);  // Create tasks.
  Runnable r2 = new Task(..);
  ...

  // Create the pool.
  ExecutorService pool=Executors.newFixedThreadPool(max_t);

  // pass the tasks to the pool.
  pool.execute(r1);
  pool.execute(r2);
  ...
  pool.shutdown(); // shutdown - tasks cannot be added
 } }
```

Outline

1. Introduction
   - Formalities
   - Overview
   - Parallelism and its impact
   - Introduction Parallel constructs
   - Java Concurrency

Program Analysis



- Code optimization requires that the compiler has a global "understanding" of how programs use the available resources.
- It has to understand how the control flows (control-flow analysis) in the program and how the data is manipulated (data-flow analysis)
- Control-flow analysis: flow of control within each procedure and across procedures.
- Data-flow analysis: how the data is manipulated in the program.

Example

```
int fib (int m){
   int f0=0, f1=1, f2,i;
   if (m <=1)
      return m;
   else {
      for (i=2; i<=m; ++i) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
}
```

```
1      receive m (val)
2      f0 = 0
3      f1 = 1
4      if (m <= 1) goto L3
5      i = 2
6  L1: if (i<=m) goto L2
7      return f2
8  L2: f2 = f0 + f1
9      f0 = f1
10     f1 = f2
11     i = i + 1
12     goto L1
13 L3:return m
```

- IR for the C code (in a format described in Muchnick book)
- `receive` specifies the reception of a parameter and the parameter-passing discipline (by-value, by-result, value-result, reference). Why do we want to have an explicit receive instruction?– Gives a point of definition for the args.
- What is the control structure? Obvious?

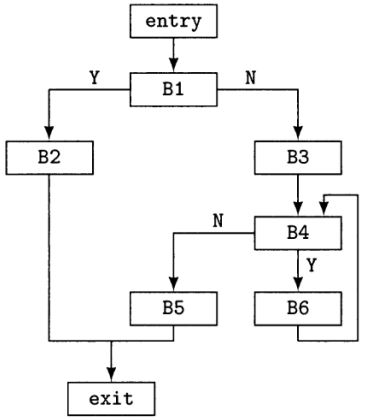
Example - flow chart and control-flow

```
int fib (int m){
   int f0=0, f1=1, f2,i;
   if (m <=1)
      return m;
   else {
      for (i=2; i<=m; ++i) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
}
```

```
1      receive m (val)
2      f0 = 0
3      f1 = 1
4      if (m <= 1) goto L3
5      i = 2
6  L1: if (i<=m) goto L2
7      return f2
8  L2: f2 = f0 + f1
9      f0 = f1
10     f1 = f2
11     i = i + 1
12     goto L1
13 L3:return m
```

- The high-level abstractions might be lost in the IR.
- Control-flow analysis can expose control structures not obvious in the high level code. Possible? Loops constructed from `if` and `goto`
- A basic block is informally a straight-line sequence of code that can be entered only at the beginning and exited only at the end.

Basic blocks - what do we get?



- `entry` and `exit` are added for reasons to be explained later.
- We can identify loops by using dominators
  - a node A in the flowgraph dominates a node B if every path from `entry` node to B includes A.
  - This relations is antisymmetric, reflexive, and transitive.
- back edge: An edge in the flow graph, whose head dominates its tail (example - edge from `B6` to `B4`.
- A loop consists of subset of nodes dominated by its entry node (head of the back edge) and having exactly one back edge in it.

Deep dive - Basic block

Basic block definition

- A basic block is a maximal sequence of instructions that can be entered only at the first of them
- The basic block can be exited only from the last of the instructions of the basic block.
- Implication:First instruction can be a) entry point of a routine,b) item target of a branch, c) item instruction following a branch or a return.
- First instruction is called the leader of the BB.

How to construct the basic block?

- Identify all the leaders in the program.
- For each leader: include in its basic block all the instructions from the leader to the next leader (next leader not included) or the end of the routine, in sequence.

What about function calls?

- In most cases it is not considered as a branch+return. Why?
- Problem with setjmp() and longjmp()? [self-study]

CFG - Control flow graph

Definition:

- A rooted directed graph $G = (N, E)$, where N is given by the set of basic blocks + two special BBs: `entry` and `exit`.
- And edge connects two basic blocks b_1 and b_2 if control can pass from b_1 to b_2.
- An edge(s) from `entry` node to the initial basic block(s?)
- From each final basic blocks (with no successors) to `exit` BB.

CFG continued

- `successor` and `predecessor` – defined in a natural way.
- A basic block is called branch node - if it has more than one successor.
- `join` node – has more than one predecessor.
- For each basic block b:

$$Succ(b) = \{n \in N | \exists e \in E \text{ such that } e = b \rightarrow n\}$$
$$Pred(b) = \{n \in N | \exists e \in E \text{ such that } e = n \rightarrow b\}$$

- A region is a strongly connected subgraph of a flow-graph.

Data Flow Analysis

Why:

- Provide information about a program manipulates its data.
- Study functions behavior.
- To help build control flow information.
- Program understanding (a function sorts an array!).
- Generating a model of the original program and verify the model.
- The DFA should give information about that program that does not misrepresent what the procedure being analyzed does.
- Program validation.

Reaching Definitions

A particular definition of a variable is said to reach a given point if

- there is an execution path from the definition to that point
- the variable might may have the value assigned by the definition.

In general undecidable.

Our goal:

- The analysis must be conservative – the analysis should not tell us that a particular definition does not reach a particular use, if it may reach.
- A 'may' conservative analysis gives us a larger set of reaching definitions than it might, if it could produce the minimal result.

To make maximum benefit from our analysis, we want the analysis to be conservative, but be as aggressive as possible.

Different types of analysis

- Intra procedural analysis.
- Whole program (inter-procedural) analysis.
- Generate intra procedural analysis and extend it to whole program.

We will study an iterative mechanism to perform such analyses.

Iterative Dataflow Analysis

- Build a collection of data flow equations – specifying which data may flow to which variable.
- Solve it iteratively.
- Start from a conservative set of initial values – and continuously improve the precision.
  Disadvantage: We may be handling large data sets.
- Start from an aggressive set of initial values – and continuously improve the precision.
  Advantage: Datasets are small to start with.
- Choice – depends on the problem at hand.

Example program

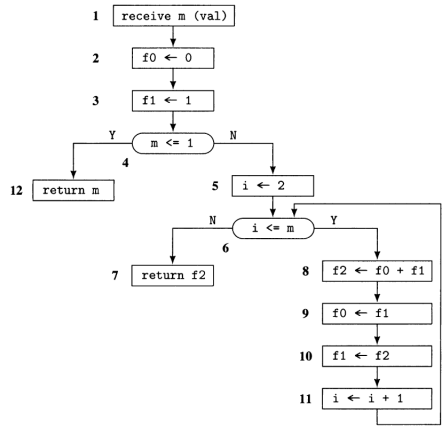
```
1 int g (int m, int i);

2 int f(int n) {
3     int i = 0, j;
4     if (n == 1) i = 2;
5     while (n > 0) {
6         j = i + 1;
7         n = g(n, i);
8     }
9     return j;
10 }
```

- Does def of `i` in line 3 reach the uses in line 6 and 7?
- Does def of `j` in line 6 reach the use in line 9?

Definitions

- GEN : GEN(b) returns the set of definitions generated in the basic block b; assigned values in the block and not subsequently killed in it.
- KILL : KILL(b) returns the set of definitions killed in the basic block b.
- IN : IN(b) returns the set of definitions reaching the basic block b.
- OUT : OUT(b) returns the set of definitions going out of basic block b.
- PRSV : Negation of KILL

Representation and Initialization



Bit Pos	Definition	Basic Block
1	`m` in node 1	B1
2	`f0` in node 2	
3	`f1` in node 3	
4	`i` in node 5	B3
5	`f2` in node 8	B6
6	`f0` in node 9	
7	`f1` in node 10	
8	`i` in node 11	

	Set rep	Bit vector
GEN(B1)	= {1, 2, 3}	⟨11100000⟩
GEN(B3)	= {4}	⟨00010000⟩
GEN(B6)	= {5, 6, 7, 8}	⟨00001111⟩
GEN(.)	= {}	⟨00000000⟩

Populating PRSV, OUT and IN

	Set rep	Bit vector
$PRSV$(B1)	= {4, 5, 8}	⟨00011001⟩
$PRSV$(B3)	= {1, 2, 3, 5, 6, 7}	⟨11101110⟩
$PRSV$(B6)	= {1}	⟨10000000⟩
$PRSV$(.)	= {1, 2, 3, 4, 5, 6, 7, 8}	⟨11111111⟩

Dataflow equations

A definition may reach the end of a basic block i:

$$OUT(i) = GEN(i) \cup (IN(i) \cap PRSV(i))$$

or with bit vectors:

$$OUT(i) = GEN(i) \vee (IN(i) \wedge PRSV(i))$$

A definition may reach the beginning of a basicblock i:

$$IN(i) = \bigcup_{j \in Pred(i)} OUT(j)$$

- GEN, $PRSV$ and OUT are created in each basic block.
- $OUT(i)$ = {} // initialization
- But IN needs to be initialized to something safe.
- $IN(entry)$ = {}

Solving the Dataflow equations: example

Itr 1:

$OUT(entry)$	$= \langle 00000000 \rangle$	$IN(entry)$	$= \langle 00000000 \rangle$
$OUT(B1)$	$= \langle 11100000 \rangle$	$IN(B1)$	$= \langle 00000000 \rangle$
$OUT(B2)$	$= \langle 11100000 \rangle$	$IN(B2)$	$= \langle 11100000 \rangle$
$OUT(B3)$	$= \langle 11110000 \rangle$	$IN(B3)$	$= \langle 11100000 \rangle$
$OUT(B4)$	$= \langle 11110000 \rangle$	$IN(B4)$	$= \langle 11110000 \rangle$
$OUT(B5)$	$= \langle 11110000 \rangle$	$IN(B5)$	$= \langle 11110000 \rangle$
$OUT(B6)$	$= \langle 00001111 \rangle$	$IN(B6)$	$= \langle 11110000 \rangle$
$OUT(entry)$	$= \langle 11110000 \rangle$	$IN(exit)$	$= \langle 11110000 \rangle$

Itr 2:

$OUT(entry)$	$= \langle 00000000 \rangle$	$IN(entry)$	$= \langle 00000000 \rangle$
$OUT(B1)$	$= \langle 11100000 \rangle$	$IN(B1)$	$= \langle 00000000 \rangle$
$OUT(B2)$	$= \langle 11100000 \rangle$	$IN(B2)$	$= \langle 11100000 \rangle$
$OUT(B3)$	$= \langle 11110000 \rangle$	$IN(B3)$	$= \langle 11100000 \rangle$
$OUT(B4)$	$= \langle 11111111 \rangle$	$IN(B4)$	$= \langle 11111111 \rangle$
$OUT(B5)$	$= \langle 11111111 \rangle$	$IN(B5)$	$= \langle 11111111 \rangle$
$OUT(B6)$	$= \langle 10001111 \rangle$	$IN(B6)$	$= \langle 11111111 \rangle$
$OUT(entry)$	$= \langle 11111111 \rangle$	$IN(exit)$	$= \langle 11111111 \rangle$

Dataflow equations: behavior

- We specify the relationship between the data-flow values before and after a block – transfer or flow equations.
  - Forward: $OUT(s) = f(IN(s), \cdots)$
  - Backward: $IN(s) = f(OUT(s), \cdots)$
- The rules never change a 1 to 0. They may only change a 0 to a 1.
- They are monotone.
- Implication – the iteration process will terminate.
- Q: What good is reaching definitions? undefined variables.
- Q: Why do the iterations produce an acceptable solution to the set of equations? – lattices and fixed points.

Lattice

- **What** : Lattice is an algebraic structure
- **Why** : To represent abstract properties of variables, expressions, functions, etc etc.
  - Values
  - Attributes
  - ...
- **Why "abstract?** Exact interpretation (execution) gives exact values, abstract interpretation gives abstract values.

Lattice definition

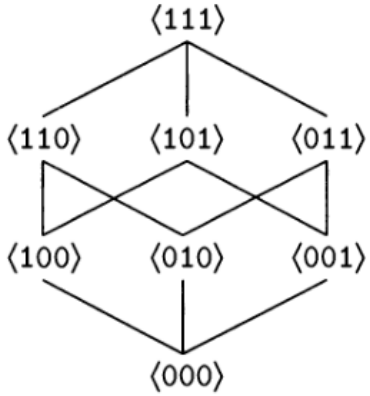
A lattice L consists of a set of values, and two operations called *meet* ($\sqcup$) and *join* ($\sqcap$). Satisfies properties:

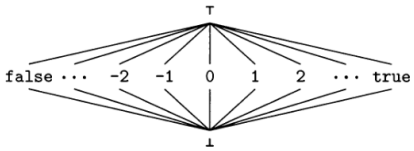
- **closure**: For all $x, y \in L$, $\exists$ a unique z and $w \in L$, such that $x \sqcap y = z$ and $x \sqcup y = w$ – each pair of elements have a unique lub and glb.
- **commutative**: For all $x, y \in L$, $x \sqcap y = y \sqcap x$, and $x \sqcup y = y \sqcup x$.
- **associative**: For all $x, y, z \in L$, $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$, and $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
- There exists two special elements of L called bottom ($\bot$), and top ($\top$).
  $\forall x \in L$, $x \sqcap \bot = \bot$ and $x \sqcup \top = \top$.
- **distributive** : (optional). $\forall x, y, z \in L$, $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$, and $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

Lattice properties

- Meet (and join) induce a partial order ($\sqsubseteq$):
  $\forall x, y \in L$, $x \sqsubseteq y$, iff $x \sqcap y = x$.
- Transitive, antisymmetry and reflexive.

Example Lattices:





Monotones and fixed point

- A function $f : L \to L$, is a monotone, if for all $x, y \in L$, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.
- Example: bit-vector lattice:
  - $f(x_1 x_2 x_3) = \langle x_1 1 x_2 \rangle$
  - $f(x_1 x_2 x_3) = \langle x_2 x_3 x_1 \rangle$
- A flow function models the effect of a programming language construct. as a mapping from the lattice for that particular analysis to itself.
- We want the flow functions to be monotones. Why?
- A fixed point of a function $f : L \to L$ is an element $z \in L$, such that $f(z) = z$.
- For a set of data-flow equations, a fixed-point is a solution of the set of equations – cannot generate any further refinement.

Meet Over All Paths solutions

- The value we wish to compute in solving data-flow equations is – meet over all paths (MOP) solution.
- Start with some prescribed information at the entry (or exit depending on forward or backward).
- Repeatedly apply the composition of the appropriate flow functions.
- For each node form the meet of the results.

A worklist based implementation (a forward analysis)

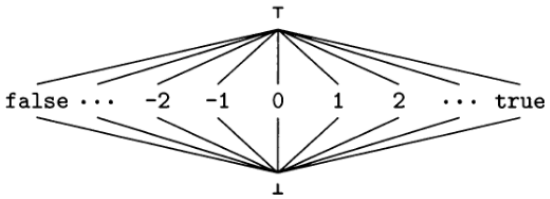
```
procedure Worklist_Iterate(N,entry,F,dfin,Init)
   N: in set of Node
   entry: in Node
   F: in Node × L ⟶ L
   dfin: out Node ⟶ L
   Init: in L
begin
   B, P: Node
   Worklist: set of Node
   effect, totaleffect: L
   dfin(entry) := Init
   Worklist := N - {entry}
   for each B ∈ N do
      dfin(B) := ⊤
   od
   repeat
      B := ♦Worklist
      Worklist -= {B}
      totaleffect := ⊤
      for each P ∈ Pred(B) do
         effect := F(P,dfin(P))
         totaleffect ⊓= effect
      od
      if dfin(B) ≠ totaleffect then
         dfin(B) := totaleffect
         Worklist ∪= Succ(B)
      fi
   until Worklist = ∅
end    || Worklist_Iterate
```

Example: Constant Propagation

Goal: Discover values that are constants on all possible executions of a program and to propagate these constant values as far forward through the program as possible

Conservative: Can discover only a subset of all the possible constants.

Lattice:



Constant Propagation lattice meet rules

- $\perp$ = Constant value cannot be guaranteed.
- $\top$ = May be a constant, not yet determined.
- $\forall x$
  - $x \sqcap \top = x$
  - $x \sqcap \perp = \perp$
  - $c_1 \sqcap c_1 = c_1$
  - $c_2 \sqcap c_1 = \perp$

Simple constant propagation

- Gary A. Kildall: A Unified Approach to Global Program Optimization - POPL 1973.
- Reif, Lewis: Symbolic evaluation and the global value graph - POPL 1977.
- **Simple constant** Constants that can be proved to be constant provided,
  - no information is assumed about which direction branches will take.
  - Only one value of each variable is maintained along each path in the program.

Kildall's algorithm

- Start with an entry node in the program graph.
- Process the entry node, and produce the constant propagation information. Send it to all the immediate successors of the entry node.
- At a merge point, get an intersection of the information.
- If at any successor node, if for any variable the value is "reduced, the process the successor, similar to the processing done for entry node.

Constant propagation - equations

- Let us assume that one basic block per statement.
- Transfer functions set F - a set of transfer functions. $f_s \in F$ is the transfer function for statement s.
- The dataflow values are given by a map: m: $Vars \rightarrow ConstantVal$
- If m is the set of input dataflow values, then $m' = f_s(m)$ gives the output dataflow values.
- Generate equations like before.

Constant propagation: equations (contd)

- Start with the entry node.
- If s is not an assignment statement, then f_s is simply the identity function.
- If s is an assignment statement to variable v, then $f_s(m) = m'$, where:
  - For all $v' \neq v$, $m'(v') = m(v')$.
  - If the RHS of the statement is a constant c, then $m'(v) = c$.
  - If the RHS is an expression (say $y\ op\ z$),

$$m'(v) = \begin{cases} m(y)\ op\ m(z) & \text{if } m(y) \text{ and } m(z) \text{ are constant values} \\ \bot & \text{if either of } m(y) \text{ and } m(z) \text{ is } \bot \\ \top & \text{Otherwise} \end{cases}$$

  - If the RHS is an expression that cannot be evaluated, then $m'(v) = \bot$.
- At a merge point, get a meet of the flow maps.

Constant Propagation - example I

```
x = 10;
y = 1;
z = 5;
if  (cond) {
    y = y / x;
    x = x - 1;
    z = z + 1;
} else {
    z = z + y;
    y = 0;
}
print x + y + z;
```

Constant Propagation - example II

```
x = 10;
y = 1;
z = 1;
while (x > 1) {
    y = x * y;
    x = x - 1;
    z = z * z;
}
A[x] = y + z;
```