Big Data Analytics

A. Parallel Computing / A.1 Threads

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute for Computer Science
University of Hildesheim, Germany

Syllabus

Tue. 10.4.	(1)	0. Introduction
		A. Parallel Computing
Tue. 17.4.	(2)	A.1 Threads
Tue. 24.4.	(3)	A.2 Message Passing Interface (MPI)
Tue. 1.5.	(4)	A.3 Graphical Processing Units (GPUs)
		B. Distributed Storage
Tue. 8.5.	(5)	B.1 Distributed File Systems
Tue. 15.5.	(6)	B.2 Partioning of Relational Databases
Tue. 22.5.	—	— Pentecoste Break —
Tue. 29.5.	(7)	B.3 NoSQL Databases
		C. Distributed Computing Environments
Tue. 5.6.	(8)	C.1 Map-Reduce
Tue. 12.6.	(9)	C.2 Resilient Distributed Datasets (Spark)
Tue. 19.6.	(10)	C.3 Computational Graphs (TensorFlow)
		D. Distributed Machine Learning Algorithms
Tue. 26.6.	(11)	D.1 Distributed Stochastic Gradient Descent
Tue. 3.7.	(12)	D.2 Distributed Matrix Factorization
Tue. 10.7.	(13)	Questions and Answers

Outline

Outline

Processes and Threads

- **process**: a running program
  - each process has its exclusive memory
  - managed by the operating system, heavy weight
  - **distributed computing**: running on different machines

- **multitasking**: run several **processes** in parallel
  - OS switches between processes
  - **multiprocessing**: run in parallel on several processors

Processes and Threads

- **process**: a running program
  - each process has its exclusive memory
  - managed by the operating system, heavy weight
  - **distributed computing**: running on different machines
  - may have several threads
- **multitasking**: run several **processes** in parallel
  - OS switches between processes
  - **multiprocessing**: run in parallel on several processors
- **thread**: a running subprogram
  - threads can share memory (**shared address space**)
  - managed by the program, light weight
- **multithreading**: run several **threads** in parallel
  - may be switched between processors or run on several in parallel

Threads APIs

- POSIX threads (Pthreads; IEEE standard 1995): C
- C++ Standard library thread (2011): C++
- Java Standard library (1995, 2004): Java
  - Thread (java.lang; 1995)
  - Lock, ThreadPoolExecutor etc. (java.util.concurrent; 2004)
- Open Multi-Processing (OpenMP; 1997): C, C++, Fortran
  - Java ports: JOMP, omp4j

Outline

Starting Threads

Runnable:

- interface in java.lang
- only method: **void run()**.
- models a procedure that can be run.

Thread:

- class in java.lang
- constructor **Thread(Runnable)**: a thread to run a given Runnable.
- **Thread.start()**: begin to execute this thread.

Starting Threads / Example

```
public class HelloWorld3 implements Runnable {
    String msg;
    public HelloWorld3(String msg) {
        this.msg = msg;
    }
    public void run() {
        while (true)
            System.out.println(msg);
    }
    public static void main(String[] args) {
        new Thread(new HelloWorld3("A")).start();
        new Thread(new HelloWorld3("B")).start();
        new Thread(new HelloWorld3("C")).start();
        new Thread(new HelloWorld3("D")).start();
    }
}
```

Output

```
A
C
B
A
A
D
⋮
```

Interrupting Threads

- **Thread.interrupt()**:
  - set the thread's interrupted property to true.
  - if the thread is sleeping or waiting, an **InterruptedException** will be thrown.
- **Thread.interrupted()**:
  - get the value of the thread's interrupted property.
- **Thread.isAlive()**:
  - a thread is alive if it has been started, but not yet died.
- **Thread.currentThread()** (static):
  - get the thread executing the current code.
- **Thread.sleep(long)** (static):
  - the current thread sleeps for the given number of milliseconds.
- There is no way to stop a thread externally.
- There is no way to interrupt a thread that does not cooperate.

Interrupting Threads / Example / Example Computation

```
import java.util.*;

public class Primes {
    ArrayList<Long> primes = new ArrayList<>();

    public void compute(long max) {
        primes.add(2L);
        for (long n = 3; n < max; n = n+2) {
            boolean isPrime = true;
            for(Long prime: primes)
                if (n % prime == 0) {
                    isPrime = false;
                    break;
                }
            if (isPrime)
                primes.add(n);
        }
    }
}
```

Interrupting Threads / Example (non coop./broken)

```
import java.util.*;

public class Worker implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; ++i) {
            System.out.println("Work step " + i);
            new Primes().compute(100000);
        }
    }

    public static void main(String[] args) {
        Thread worker = new Thread(new Worker());
        worker.start();
        while (worker.isAlive()) {
            String input = System.console().readLine();
            if (input.equals("interrupt")) {
                worker.interrupt();
                break;
            }
        }
    }
}
```

Output

```
Work step 0
Work step 1
Work step 2
interrupt
Work step 3
.
.
.
```

Interrupting Threads / Example (coop./fixed)

```java
import java.util.*;

public class Worker2 implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; ++i) {
            if (Thread.currentThread().isInterrupted())
                break;
            System.out.println("Work step " + i);
            new Primes().compute(100000);
        }
    }

    public static void main(String[] args) {
        Thread worker = new Thread(new Worker2());
        worker.start();
        while (worker.isAlive()) {
            String input = System.console().readLine();
            if (input.equals("interrupt")) {
                worker.interrupt();
                break;
            }
        }
    }
}
```

Output

```
Work step 0
Work step 1
Work step 2
interrupt
```

Interrupting Threads / Example (sleeping)

```
import java.util.*;

public class Worker3 implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; ++i) {
            System.out.println("Work step " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                break;
            }
        }
    }

    public static void main(String[] args) {
        Thread worker = new Thread(new Worker3());
        worker.start();
        while (worker.isAlive()) {
            String input = System.console().readLine();
            if (input.equals("interrupt")) {
                worker.interrupt();
                break;
            }
        }
    }
}
```

Outline

Synchronization

- Several threads running in parallel may need to exchange some information.
  - can be accomplished through shared variables

Synchronization

- Several threads running in parallel may need to exchange some information.
  - can be accomplished through shared variables
- Several threads running in parallel may need to coordinate, e.g.,
  - a thread needs to wait until another is terminated
  - a thread requires exclusive access to some variable
    - e.g., to increment a counter or to edit an array
  - a thread requires some condition to hold to continue
    - e.g., further input in a stream being available

Synchronization

- Several threads running in parallel may need to exchange some information.
  - can be accomplished through shared variables

- Several threads running in parallel may need to coordinate, e.g.,
  - a thread needs to wait until another is terminated
  - a thread requires exclusive access to some variable
    - e.g., to increment a counter or to edit an array
  - a thread requires some condition to hold to continue
    - e.g., further input in a stream being available
- Called **synchronization**.

Waiting for Termination

- **Thread.join()**:
  - the current thread sleeps until the target thread dies.
- When a program is started, there exists one thread (often called main thread).
- a program terminates once all its threads died.

Waiting for Termination / Example

```
public class Counter {
    int count = 0;
    public void increment() { ++count; }
    public int value() { return count; }
}
```

```
public class ParallelCounters implements Runnable {
    Counter count;
    int num;

    public ParallelCounters(Counter count, int num) {
        this.count = count;
        this.num = num;
    }
    public void run() {
        for (int i = 0; i < num; ++i)
            count.increment();
    }

    public static void main(String[] args) throws InterruptedException {
        Counter count = new Counter();
        Thread a = new Thread(new ParallelCounters(count, 100));
        Thread b = new Thread(new ParallelCounters(count, 100));
        Thread c = new Thread(new ParallelCounters(count, 100));
        Thread d = new Thread(new ParallelCounters(count, 100));

        a.start(); b.start(); c.start(); d.start();

        System.out.println("counter: " + count.value());
    }
}
```

Waiting for Termination / Example

```
public class ParallelCounters2 implements Runnable {
    Counter count;
    int num;

    public ParallelCounters2 (Counter count, int num) {
        this.count = count;
        this.num = num;
    }
    public void run() {
        for (int i = 0; i < num; ++i)
            count.increment();
    }

    public static void main(String[] args) throws InterruptedException {
        Counter count = new Counter();
        Thread a = new Thread(new ParallelCounters2(count, 100));
        Thread b = new Thread(new ParallelCounters2(count, 100));
        Thread c = new Thread(new ParallelCounters2(count, 100));
        Thread d = new Thread(new ParallelCounters2(count, 100));

        a.start(); b.start(); c.start(); d.start();

        a.join(); b.join(); c.join(); d.join();

        System.out.println("counter: " + count.value());
    }
}
```

Synchronized Methods

- Even steps of very simple statements such as **++value** may be interleaved with steps in other threads and lead to corruption.
- For each object and class there exists an implicit lock (called monitor).
- Methods marked **synchronized**
  - try to acquire the monitor of their object and
  - block if the monitor is already taken by another thread until it becomes available.
  - thus, there is at most one thread executing any synchronized method at any time.
- **static synchronized** methods try to acquire the monitor of the class.
- The **synchronized(Object) { . . . }** statement tries to acquire the monitor of the given object/class.
- **Thread.holdsLock(Object)** (static) tests if the current thread holds a given monitor.

Synchronized Methods / Example

```
public class Counter2 {
    int count = 0;
    public synchronized void increment() { ++count; }
    public int value() { return count; }
}
```

```
public class ParallelCounters3 implements Runnable {
    Counter2 count;
    int num;

    public ParallelCounters3 (Counter2 count, int num) {
        this.count = count;
        this.num = num;
    }
    public void run() {
        for (int i = 0; i < num; ++i)
            count.increment();
    }

    public static void main(String[] args) throws InterruptedException {
        Counter2 count = new Counter2();
        Thread a = new Thread(new ParallelCounters3(count, 100));
        Thread b = new Thread(new ParallelCounters3(count, 100));
        Thread c = new Thread(new ParallelCounters3(count, 100));
        Thread d = new Thread(new ParallelCounters3(count, 100));

        a.start(); b.start(); c.start(); d.start();

        a.join(); b.join(); c.join(); d.join();

        System.out.println("counter: " + count.value());
```

Atomic Objects

- Atomic objects provide methods that are executed as a whole (and not interrupted by any other threads).
- **AtomicInteger** provides such operations for a simple int (**java.util.concurrent.atomic**):
  - **set(int)**: set a value.
  - **intValue()**: get.
  - **addAndGet(int)**: atomically adds a value.
  - **incrementAndGet()**: atomically increment.

Atomic Objects

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

public class ParallelCounters4 implements Runnable {
    AtomicInteger count;
    int num;

    public ParallelCounters4(AtomicInteger count, int num) {
        this.count = count;
        this.num = num;
    }
    public void run() {
        for (int i = 0; i < num; ++i)
            count.incrementAndGet();
    }

    public static void main(String[] args) throws InterruptedException {
        AtomicInteger count = new AtomicInteger();
        Thread a = new Thread(new ParallelCounters4(count, 100));
        Thread b = new Thread(new ParallelCounters4(count, 100));
        Thread c = new Thread(new ParallelCounters4(count, 100));
        Thread d = new Thread(new ParallelCounters4(count, 100));

        a.start(); b.start(); c.start(); d.start();

        a.join(); b.join(); c.join(); d.join();

        System.out.println("counter: " + count.intValue());
    }
}
```

Synchronization Issues

- Deadlock:
  - Thread A is waiting for Thread B,
    Thread B is waiting for Thread A.
  - Thread A holds lock 1 and requests lock 2,
    Thread B holds lock 2 and requests lock 1.
  - The program freezes.

Synchronization Issues / Deadlock / Example

```
class Account {
    String id;
    double balance = 0;

    Account(String id) { this.id = id; }

    void withdraw(double amount) { balance -= amount; }
    void deposit(double amount) { balance += amount; }

    static void transfer(Account from, Account to, double amount) {
        synchronized(from) {
            synchronized(to) {
                from.withdraw(amount);
                to.deposit(amount);
            }
        }
    }
}
```

Synchronization Issues / Deadlock / Example (ctd.)

```
class ParallelTransactions implements Runnable {
    Account from, to;
    ParallelTransactions (Account from, Account to) {
        this.from = from;
        this.to = to;
    }
    public void run() {
        while (true) {
            Account.transfer(from, to, 100.00);
            System.out.println(" transfered 100.00 from " + from.id + " to " + to.id);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                break;
            }
        }
    }
    public static void main(String[] args) {
        Account a = new Account("A"), b = new Account("B");
        new Thread(new ParallelTransactions(a, b)).start();
        new Thread(new ParallelTransactions(b, a)).start();
    }
}
```

Synchronization Issues / Deadlock / Example (fix)

```
class Account2 {
    String id;
    double balance = 0;

    Account2(String id) { this.id = id; }

    void withdraw(double amount) { balance -= amount; }
    void deposit(double amount) { balance += amount; }

    static synchronized void transfer(Account2 from, Account2 to, double amount) {
        synchronized(from) {
            synchronized(to) {
                from.withdraw(amount);
                to.deposit(amount);
            }
        }
    }
}
```

Conditions / Guarded Blocks

- Often threads require a specific condition to hold before they can resume their work.

Conditions / Guarded Blocks

- Often threads require a specific condition to hold before they can resume their work.
- **polling**:
  - repeatedly query the condition, proceed if it holds
  - wastes resources
  - possibly sleep between trials
    - sleep time not straight-forward to set

Conditions / Guarded Blocks

- Often threads require a specific condition to hold before they can resume their work.
- **polling**:
  - repeatedly query the condition, proceed if it holds
  - wastes resources
  - possibly sleep between trials
    - sleep time not straight-forward to set
- Condition:
  - a queue of threads to wait for a condition to become true
  - a method to wait on such a condition (**Object.wait**)
  - a method to signal that the condition may have changed (**Object.notifyAll**)
    - The condition itself is not part of the model.

Guarded Blocks / Example (1/2)

```
import java.util.*;
class Store {
    ArrayList<String> store = new ArrayList<>();
    synchronized void put(String item) { store.add(item); }
    synchronized String pop() { String item = store.get(0); store.remove(0); return item; }
}
```

```
class Producer implements Runnable {
    Store store;
    public Producer(Store store) { this.store = store; }
    public void run() {
        while (true) {
            try {
                Thread.sleep(Math.round(Math.random() * 1000));
            } catch (InterruptedException ex) {}
            String item = "A";
            store.put(item);
            System.out.println("produced " + item + ", store = " + store.store);
        }
    }
}
```

Guarded Blocks / Example (2/2)

```
class Consumer implements Runnable {
    Store store;
    public Consumer(Store store) { this.store = store; }
    public void run() {
        while (true) {
            if (store.store.size() >= 2) {
                String item1 = store.pop(), item2 = store.pop();
                System.out.println("consumed " + item1 + " and " + iten
                try {
                    Thread.sleep(Math.round(Math.random() * 1000));
                } catch (InterruptedException ex) {}
            }
        }
    }
}
```

```
class PCExample {
    public static void main(String[] args) {
        Store store = new Store();
        Producer prod = new Producer(store);
        Consumer cons = new Consumer(store);
        new Thread(prod).start();
        new Thread(cons).start();
    }
}
```

Output

```
produced A, store = [A]
produced A, store = [A, A]
produced A, store = [A, A, A]
produced A, store = [A, A, A, A]
produced A, store = [A, A, A, A, A]
.
.
.
```

Guarded Blocks / Example (fix; 1/2)

```
import java.util.*;
class Store {
    ArrayList<String> store = new ArrayList<>();
    public synchronized void put(String item) {
        synchronized(store) {
            store.add(item);
        }
        notifyAll();
    }
    public String pop() {
        String item;
        synchronized(store) {
            item = store.get(0);
            store.remove(0);
        }
        return item;
    }
    public String toString() {
        String s;
        synchronized(store) {
            s = store.toString();
        }
        return s;
    }
}
```

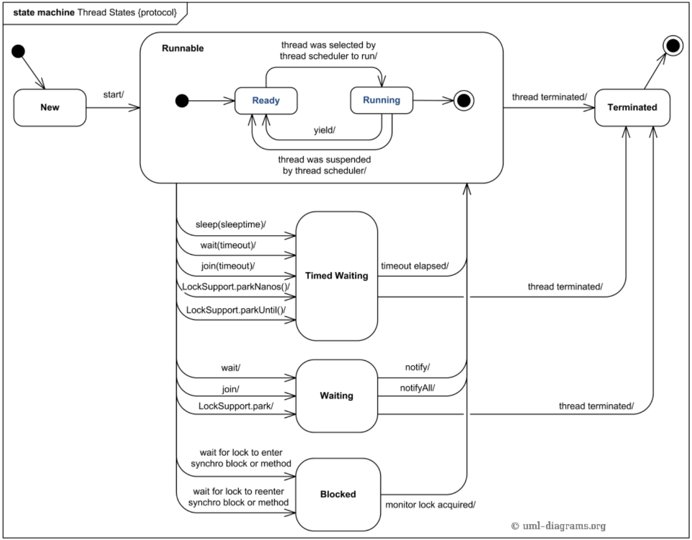
Guarded Blocks / Example (fix; 2/2)

```
class Consumer implements Runnable {
    Store store;
    public Consumer(Store store) { this.store = store; }
    public void run() {
        try {
            while (true) {
                if (store.store.size() >= 2) {
                    String item1 = store.pop(), item2 = store.pop();
                    System.out.println("consumed " + item1 + " and " +
                    Thread.sleep(Math.round(Math.random() * 1000));
                } else
                    synchronized(store) {
                        store.wait();
                    }
            }
        } catch (InterruptedException ex) {}
    }
}
```

Output

```
produced A, store = [A]
produced A, store = [A, A]
consumed A and A, store = []
produced A, store = [A]
produced A, store = [A, A]
consumed A and A, store = []
produced A, store = [A]
produced A, store = [A, A]
consumed A and A, store = []
produced A, store = [A]
```

Thread States



Information about threads at runtime

- **Thread.currentThread** (static): thread executing current code.
- **Thread.getState**: state of the thread.
- **Thread.getId**: get a numeric ID of the thread.
- **Thread.getActiveCount**: get number of concurrent threads.
- **Thread.enumerate**: get all concurrent threads.
- **Thread.getThreadGroup**: get group of the thread.

Outline

Locks

- A lock models mutually exclusive access to a resource.
  - only one thread can hold a lock at any time.
  - locks have methods to acquire and release them.
- **ReentrantLock**: reentrant implementation of interface **Lock**
  - reentrant: bookkeeping for repeated acquisitions and releases by the same thread.
- **Lock.lock**: acquire the lock, if possible, block otherwise until it becomes available.
- **Lock.unlock**: release the lock.
- **Lock.tryLock**: acquire the lock if possible, do nothing otherwise, return success.
  - atomic method for **Thread.holdsLock** followed by **synchronized**.
- in package **java.util.concurrent.locks**

Locks / Example

```
import java.util.concurrent.locks.*;

class Account3 {
    String id;
    double balance = 0;
    private final Lock lock = new ReentrantLock();

    Account3(String id) { this.id = id; }

    void withdraw(double amount) { balance -= amount; }
    void deposit(double amount) { balance += amount; }

    static boolean transfer(Account3 from, Account3 to, double amount) {
        boolean from_lock = from.lock.tryLock();
        boolean to_lock = to.lock.tryLock();
        if (from_lock && to_lock) {
            from.withdraw(amount);
            to.deposit(amount);
        }
        if (from_lock)
            from.lock.unlock();
        if (to_lock)
            to.lock.unlock();
        return from_lock && to_lock;
    }
}
```

Locks / Example

```
class ParallelTransactions3 implements Runnable {
    Account3 from, to;
     ParallelTransactions3 (Account3 from, Account3 to) {
        this.from = from;
        this.to = to;
    }
    public void run() {
        while (true) {
            while (! Account3.transfer(from, to, 100.00)) {
                System.out.println("accounts busy, delay somewhat");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ex) {}
            }
            System.out.println(" transfered 100.00 from " + from.id + " t
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {}
        }
    }
    public static void main(String[] args) {
        Account3 a = new Account3("A"), b = new Account3("B");
        new Thread(new ParallelTransactions3(a, b)).start();
        new Thread(new ParallelTransactions3(b, a)).start();
    }
}
```

Output

```
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
transfered 100.00 from B to A
accounts busy, delay somewhat
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
accounts busy, delay somewhat
transfered 100.00 from B to A
```

Locks / Example

```java
class ParallelTransactions3 implements Runnable {
    Account3 from, to;
     ParallelTransactions3 (Account3 from, Account3 to) {
        this.from = from;
        this.to = to;
    }
    public void run() {
        while (true) {
            while (! Account3.transfer(from, to, 100.00)) {
                System.out.println("accounts busy, delay somewhat");
                try {
                    Thread.sleep(1000);
                } catch ( InterruptedException ex) {}
            }
            System.out.println(" transfered 100.00 from " + from.id + " t
            try {
                Thread.sleep(1000);
            } catch ( InterruptedException ex) {}
        }
    }
    public static void main(String[] args) {
        Account3 a = new Account3("A"), b = new Account3("B");
        new Thread(new ParallelTransactions3(a, b)).start();
        new Thread(new ParallelTransactions3(b, a)).start();
    }
}
```

Output

```
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
transfered 100.00 from B to A
accounts busy, delay somewhat
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
accounts busy, delay somewhat
transfered 100.00 from B to A
```

Locks / Good Practice

- if an exception is thrown after **Lock.lock**,
  in simple sequential code **Lock.unlock** may never be executed.

```
lck.lock();
...   // do something that may throw an exception
lck.unlock();
```

Locks / Good Practice

- if an exception is thrown after **Lock.lock**,
  in simple sequential code **Lock.unlock** may never be executed.

```
lck.lock();
... // do something that may throw an exception
lck.unlock();
```

- better wrap into a **try–finally** block:

```
try {
  lck.lock();
  ... // do something that may throw an exception
} finally {
  lck.unlock();
}
```

Outline

Thread pools

- Avoid creation and destruction of thread objects.
- Recycle thread objects, assigning Runnables to instances from a pool.

Thread pools

- Avoid creation and destruction of thread objects.
- Recycle thread objects, assigning Runnables to instances from a pool.
- **ExecutorService** (interface):
  - **submit(Runnable)**: execute a runnable.
  - **shutdown**: wait for all submitted threads to complete.

Thread pools

- Avoid creation and destruction of thread objects.
- Recycle thread objects, assigning Runnables to instances from a pool.

- **ExecutorService** (interface):
  - **submit(Runnable)**: execute a runnable.
  - **shutdown**: wait for all submitted threads to complete.

- **Executors.newFixedThreadPool(int)** (static):
  - create an **ExecutorService** with a fixed number of threads
    - never run more than given number of threads in parallel.

Thread pools / Example

```java
import java.util.concurrent.*;

public class ExThreadpool implements Runnable {
    String name;
    public ExThreadpool(String name) { this.name = name; }

    public void run() {
        System.out.println("start " + name);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {}
        System.out.println("end " + name);
    }

    public static void main(String[] args) {
        int cores = Runtime.getRuntime().availableProcessors();
        System.out.println("#cores = " + cores);

        ExecutorService pool = Executors.newFixedThreadPool(cores);
        for (int i = 0; i < 2*cores; ++i)
        pool.submit(new ExThreadpool("" + i));
            // pool.execute(new ExThreadpool("" + i));
        pool.shutdown();
    }
}
```

Output

```
cores = 4
start 0
start 1
start 2
start 3
end 0
end 1
start 4
start 5
end 2
start 6
end 3
start 7
end 4
end 5
end 6
end 7
```

Dependency Graphs

- computation composed of several atomic parts: **tasks**
- some tasks require the results of others as input
  $\rightsquigarrow$ dependency graph

Dependency Graphs

- computation composed of several atomic parts: **tasks**
- some tasks require the results of others as input
  $\rightsquigarrow$ dependency graph
- encapsulate access to such results in an object: **Future<T>**
  - **Future<T>.get()**:
    - wait until the producing task is completed
    - then return the result
    - throw an **ExecutionException** if anything goes wrong

Dependency Graphs

- computation composed of several atomic parts: **tasks**
- some tasks require the results of others as input
  $\rightsquigarrow$ dependency graph
- encapsulate access to such results in an object: **Future<T>**
  - **Future<T>.get()**:
    - wait until the producing task is completed
    - then return the result
    - throw an **ExecutionException** if anything goes wrong
- abstract functions as interface: **Callable<T>**
  - like **Runnable**, but
  - returns a **Future<T>** – a function, not a procedure
  - may throw exceptions.

Dependency Graphs

- computation composed of several atomic parts: **tasks**
- some tasks require the results of others as input
  ⇝ dependency graph
- encapsulate access to such results in an object: **Future<T>**
  - **Future<T>.get()**:
    - wait until the producing task is completed
    - then return the result
    - throw an **ExecutionException** if anything goes wrong
- abstract functions as interface: **Callable<T>**
  - like **Runnable**, but
  - returns a **Future<T>** – a function, not a procedure
  - may throw exceptions.
- **ExecutorService.submit(Callable<T>)**: execute a callable.

Dependency Graphs / Example

```
import java.util.concurrent.*;

public class ExFuture2 {
    public static class Constant implements Callable<Double> {
        Double value;
        public Constant(Double value) { this.value = value; }
        public Double call() throws InterruptedException {
            System.out.println("Start computing constant");
            Thread.sleep(Math.round(value * 100));
            System.out.println("Compute constant " + value);
            return value;
        }
    }
    public static class Sum implements Callable<Double> {
        Future<Double> d1, d2;
        public Sum(Future<Double> d1, Future<Double> d2) {
            this.d1 = d1; this.d2 = d2;
        }
        public Double call() throws InterruptedException, ExecutionException {
            System.out.println("Start computing sum");
            Double v1 = d1.get(), v2 = d2.get();
            Thread.sleep(1000);
            System.out.println("Compute sum " + v1 + " + " + v2 + " = " + (v1+v2));
            return v1 + v2;
        }
    }
```

Dependency Graphs / Example

```java
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService pool = Executors.newFixedThreadPool(8);
        Future<Double> c3_res = pool.submit(new Constant(3.0)),
            c5_res = pool.submit(new Constant(5.0)),
            c6_res = pool.submit(new Constant(6.0)),
            sum1_res = pool.submit(new Sum(c5_res, c6_res)),
            sum2_res = pool.submit(new Sum(c3_res, c5_res)),
            sum3_res = pool.submit(new Sum(sum1_res, sum2_res));
        System.out.println("(3+5)+(5+6) = " + sum3_res.get());
        pool.shutdown();
    }
}
```

Output

```
Start computing constant
Start computing constant
Start computing constant
Start computing sum
Start computing sum
Start computing sum
Compute constant 3.0
Compute constant 5.0
Compute constant 6.0
Compute sum 3.0 + 5.0 = 8.0
Compute sum 5.0 + 6.0 = 11.0
Compute sum 11.0 + 8.0 = 19.0
(3+5)+(5+6) = 19.0
```

Further Thread Classes

- concurrent collections:
  - provide atomic thread-safe query and edit operations for collections

Outline

Open MP

- Open Multi-Processing (OpenMP; 1997): C, C++, Fortran
  - Java ports:
    - JOMP: seems no longer available?
    - JAMP:
    - omp4j
- Multithreading directives are added as comments to the code.
  - starting with **omp**
- Special preprocessor **omp4j**:
  - Replace comments by code using the Java threads API.
  - Then compile the code using the standard compiler.

Parallel Sections / Example

```
public class HelloWorld {
    public static void main(String[] args) {
    // omp parallel threadNum(4)
    {
            System.out.print(" hello ");
        }
    }
}
```

Parallel Sections / Example

```
public class HelloWorld {
    public static void main(String[] args) {
    // omp parallel threadNum(4)
    {
        System.out.print("hello ");
      }
    }
}
```

Output

```
hello hello hello hello
```

Parallel Sections / Example / Under the Hood

```
public class HelloWorld {
    public static void main(String[] args) {
        class OMPContext {}
        final OMPContext ompContext = new OMPContext();
        final org.omp4j.runtime.IOMPExecutor ompExecutor = new org.omp4j.runtime.DynamicExecutor(4);

        for (int ompI = 0; ompI < 4; ompI++) {
            ompExecutor.execute(new Runnable(){
                @Override
                public void run() {
                    System.out.print("hello ");
                }
            });
        }
        ompExecutor.waitForExecution();
    }
}
```

Parallel For / Example

```
public class ExParallelFor {
    public static void main(String[] args) {
        // omp parallel for
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
        }
    }
}
```

Parallel For / Example

```
public class ExParallelFor {
    public static void main(String[] args) {
        // omp parallel for
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
        }
    }
}
```

Output

0213458679

Directives

Directives:

Directive	Usage	Behavior
// omp parallel	Before {...}	The statement will be invoked in parallel (as many threads as possible).
// omp parallel for	Before for-loop	The for-loop will be iterated in parallel.
// omp sections	Before {...}	Wrapper for // omp sections directives. It may not contain any other code
// omp section	Before {...}	The statement will be invoked together with other sections in parallel.
// omp critical	Before {...}	At most one thread will access the statement at any particular time.
// omp barrier	Before {}	All threads stop here until the for the last one.
// omp master	Before {...}	Only master thread will execute the statement.
// omp single	Before {...}	Only one thread will execute the statement, no matter which one.

Attributes:

Attribute	Behavior
threadNum(N)	The directive will be invoked with N threads. Default value is set to number of CPUs.
schedule(dynamic|static)	The directive will use dynamic or static executor. Default value is set to dynamic.
public(a,b)	Variables a and b are shared among all threads.
private(a,b)	Variables a and b are created (via parameter-less constructor) for each thread separately.
firstprivate(a,b)	Variables a and b are created (via copy-constructor) for each thread separately.

Outline

Primes / Sequential

```
import java.util.*;

public class Eratosthenes {
    boolean[] is_prime;
    public void compute(int max) {
        is_prime = new boolean[max+1];
        for (int i = 1; i <= max; ++i)
            is_prime[i] = true;
        for (int i = 2; i < Math.floor(Math.sqrt(max)); ++i) {
            if (is_prime[i]) {
                for (int j = 2*i; j <= max; j += i) {
                    is_prime[j] = false;
                }
            }
        }
    }
    public static void main(String[] args) {
        Eratosthenes primes = new Eratosthenes();
        primes.compute(1000000000);
    }
}
```

Primes / Bad Parallelization

```
import java.util.*;

public class Eratosthenes2 {
    boolean[] is_prime;
    public void compute(int max) {
        is_prime = new boolean[max+1];
        for (int i = 1; i <= max; ++i)
            is_prime[i] = true;
        for (int i = 2; i < Math.floor(Math.sqrt(max)); ++i) {
            if (is_prime[i]) {
                // omp parallel for
                for (int j = 2*i; j <= max; j += i) {
                    is_prime[j] = false;
                }
            }
        }
    }
    public static void main(String[] args) {
        Eratosthenes2 primes = new Eratosthenes2();
        primes.compute(1000000000);
    }
}
```

Primes / Recursive

```
import java.util.*;

public class Eratosthenes3 {
    boolean[] is_prime;

    public void compute(int max) {
        is_prime = new boolean[max+1];
        for (int i = 0; i < max+1; ++i)
            is_prime[i] = true;
        do_compute(max);
    }
    protected void do_compute(int max) {
        if (max <= 2)
            return;
        int max_factor = (int) Math.floor(Math.sqrt(max));
        do_compute(max_factor);
        for (int i = 2; i <= max_factor; ++i) {
            if (is_prime[i]) {
                for (int j = 2*i; j <= max; j += i) {
                    is_prime[j] = false;
                }
            }
        }
    }
    public static void main(String[] args) {
        Eratosthenes3 primes = new Eratosthenes3();
        primes.compute(1000000000);
    }
}
```

Primes / Good Parallelization

```
import java.util.*;

public class Eratosthenes4 {
    boolean[] is_prime;

    public void compute(int max) {
        is_prime = new boolean[max+1];
        for (int i = 0; i < max+1; ++i)
            is_prime[i] = true;
        do_compute(max);
    }
    protected void do_compute(int max) {
        if (max <= 2)
            return;
        int max_factor = (int) Math.floor(Math.sqrt(max));
        do_compute(max_factor);
        // omp parallel for
        for (int i = 2; i <= max_factor; ++i) {
            if (is_prime[i]) {
                for (int j = 2*i; j <= max; j += i) {
                    is_prime[j] = false;
                }
            }
        }
    }
    public static void main(String[] args) {
        Eratosthenes4 primes = new Eratosthenes4();
        primes.compute(1000000000);
    }
}
```

Primes / Good Parallelization

```
import java.util.*;

public class Eratosthenes4 {
    boolean[] is_prime;

    public void compute(int max) {
        is_prime = new boolean[max+1];
        for (int i = 0; i < max+1; ++i)
            is_prime[i] = true;
        do_compute(max);
    }
    protected void do_compute(int max) {
        if (max <= 2)
            return;
        int max_factor = (int) Math.floor(Math.sqrt(max));
        do_compute(max_factor);
        // omp parallel for
        for (int i = 2; i <= max_factor; ++i) {
            if (is_prime[i]) {
                for (int j = 2*i; j <= max; j += i) {
                    is_prime[j] = false;
                }
            }
        }
    }
    public static void main(String[] args) {
        Eratosthenes4 primes = new Eratosthenes4();
        primes.compute(1000000000);
    }
}
```

implementation	runtime [s]
sequential	10.4
badly parallel	>120.0
recursive	10.0
OK parallel	6.7
(using 8 cores)	

Matrix Multiplication / Sequential

```java
public class Matrix {
    int N, M;
    double[][] values;
    public Matrix(int N, int M) {
        this.N = N; this.M = M;
        values = new double[N][M];
    }
    public void fill_random() {
        for (int n = 0; n < N; ++n) {
            for (int m = 0; m < M; ++m) {
                values[n][m] = 2 * (Math.random() - 0.5);
            }
        }
    }

    public Matrix mul(Matrix B) throws IllegalArgumentException {
        if (M != B.N)
            throw new IllegalArgumentException("Number of columns and rows does not match in mul.");
        Matrix C = new Matrix(N, B.M);
        for (int n = 0; n < N; ++n) {
            for (int m = 0; m < B.M; ++m) {
                double val = 0;
                for (int k = 0; k < M; ++k) {
                    val += values[n][k] * B.values[k][m];
                }
                C.values[n][m] = val;
            }
        }
        return C;
    }
```

```java
    public static void main(String[] args) {
        Matrix A = new Matrix(1000, 2000),
            B = new Matrix(2000, 3000);
        A.fill_random();
        B.fill_random();
        Matrix C = A.mul(B);
    }
}
```

Matrix Multiplication / Parallelization

```
public class Matrix {
    int N, M;
    double[][] values;
    public Matrix(int N, int M) {
        this.N = N; this.M = M;
        values = new double[N][M];
    }
    public void fill_random() {
        for (int n = 0; n < N; ++n) {
            for (int m = 0; m < M; ++m) {
                values[n][m] = 2 * (Math.random() - 0.5);
            }
        }
    }

    public Matrix mul(Matrix B) throws IllegalArgumentException {
        if (M != B.N)
            throw new IllegalArgumentException("Number of columns and rows does not match in mul.");
        Matrix C = new Matrix(N, B.M);
        // omp parallel for
        for (int n = 0; n < N; ++n) {
            for (int m = 0; m < B.M; ++m) {
                double val = 0;
                for (int k = 0; k < M; ++k) {
                    val += values[n][k] * B.values[k][m];
                }
                C.values[n][m] = val;
            }
        }
        return C;
    }

    public static void main(String[] args) {
        Matrix A = new Matrix(1000, 2000),
            B = new Matrix(2000, 3000);
        A.fill_random();
        B.fill_random();
        Matrix C = A.mul(B);
    }
}
```

Matrix Multiplication / Tiled Sequential

```
public Matrix mul(Matrix B) throws IllegalArgumentException {
    if (M != B.N)
        throw new IllegalArgumentException("Number of columns and rows does not match in mul.");
    Matrix C = new Matrix(N, B.M);
    int T = (int) Math.ceil(Math.sqrt(M));
    for (int n0 = 0; n0 < N; n0+= T) {
        for (int m0 = 0; m0 < B.M; m0+= T) {
            for (int k0 = 0; k0 < M; k0+= T) {
                for (int n = n0; n < Math.min(N, n0+T); ++n) {
                    for (int m = m0; m < Math.min(M, m0+T); ++m) {
                        double val = 0;
                        for (int k = k0; k < Math.min(M, k0+T); ++k) {
                            val += values[n][k] * B.values[k][m];
                        }
                        C.values[n][m] += val;
                    }
                }
            }
        }
    }
    return C;
}
```

Matrix Multiplication / Tiled Sequential

```
public Matrix mul(Matrix B) throws IllegalArgumentException {
    if (M != B.N)
        throw new IllegalArgumentException("Number of columns and rows does not match in mul.");
    Matrix C = new Matrix(N, B.M);
    int T = (int) Math.ceil(Math.sqrt(M));
    for (int n0 = 0; n0 < N; n0+= T) {
        for (int m0 = 0; m0 < B.M; m0+= T) {
            for (int k0 = 0; k0 < M; k0+= T) {
                for (int n = n0; n < Math.min(N, n0+T); ++n) {
                    for (int m = m0; m < Math.min(M, m0+T); ++m) {
                        double val = 0;
                        for (int k = k0; k < Math.min(M, k0+T); ++k) {
                            val += values[n][k] * B.values[k][m];
                        }
                        C.values[n][m] += val;
                    }
                }
            }
        }
    }
    return C;
}
```

implementation	runtime [s]
sequential	21.2
parallel	4.5
tiled sequential	3.9
tiled parallel	1.2
(using 8 cores)	

Summary (1/2)

- **Threads** enable lightweight **concurrency**, i.e., concurrent execution of parts of a program (**tasks**).

- A **scheduler** assigns threads to processors/cores dynamically.
  - If there are more active threads as available processors/cores, the scheduler will do **time slicing**:
    - pick a ready thread and run it for a fixed amount of time,
    - suspend the active thread, then pick another one.
  - In consequence, there are no guarantees about execution order.

- Compared to **processes**,
  - threads have less overhead to setup and start,
  - threads **can share memory**,
  - threads communicate through shared memory (while processes communicate e.g., through pipes)

Summary (2/2)

- Threads require at least elementary synchronization such as
  - one thread waiting for the others to complete (**join**),
  - the possibility to **interrupt** another thread (cooperatively).

- When shared state is updated, more complex **synchronization** is required.
  - to avoid **data races**:
    - = concurrent update of the same variable, leaving it in an undefined state.
    - **atomic objects** offer a set of atomic operations.
    - **monitors** allow more fine-grained synchronization per object.
    - **guarded blocks** / **conditions** allow (possibly many) threads to wait until a condition holds and another thread to signal once this is the case.
    - **locks**/**mutexes** model exclusive access to a resource: only one thread at a time can acquire a lock, others have to wait, until it is released.

Further Readings

- General introduction to parallel computing: [Grama et al., 2003, ch. 1+2]

- Design of parallel algorithms: [Grama et al., 2003, ch. 3]

- Processes, threads and scheduling at operation system level: O'Gorman [2003]

References I

Ananth Grama, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.

John O'Gorman. *The Linux Process Manager: The Internals of Scheduling, Interrupts and Signals*. John Wiley & Sons, Inc., New York, NY, USA, 2003. ISBN 0470847719.