CSE 373

Introduction to Parallel Algorithms

reading: Grossman

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Changing our assumptions

- So far most or all of your study of computer science has assumed that *only one thing happens at a time* in a given program.
    - **sequential programming:** Each statement executes in sequence.

- Removing this assumption creates challenges and opportunities:
    - *Programming:* How can we divide work among threads of execution and coordinate (synchronize) among them?
    - *Algorithms:* How can activities in parallel speed-up a program?
        - (more throughput: work done per unit time)
    - *Data structures:* May need to support concurrent access (multiple threads operating on data at the same time).

Brief arch. history

- **CPU:** Central Processing Unit. The brain of a computer.
  - From ~1980-2005, CPU speed (GHz) got exponentially faster.
  - Roughly doubled every 1.5 years ("Moore's Law").

- But we are reaching limits of classic CPU design.
  - Increasing speeds further generates too much heat.
  - Any single CPU over ~3-4 GHz crashes or burns out in normal usage.

- Current work-around: Use multiple processors.
  - Or, more recently, produce one CPU containing many processors in it.
  - **core:** A processor-within-a-processor.
    - A "multi-core" processor is one with several cores inside.

Using many cores

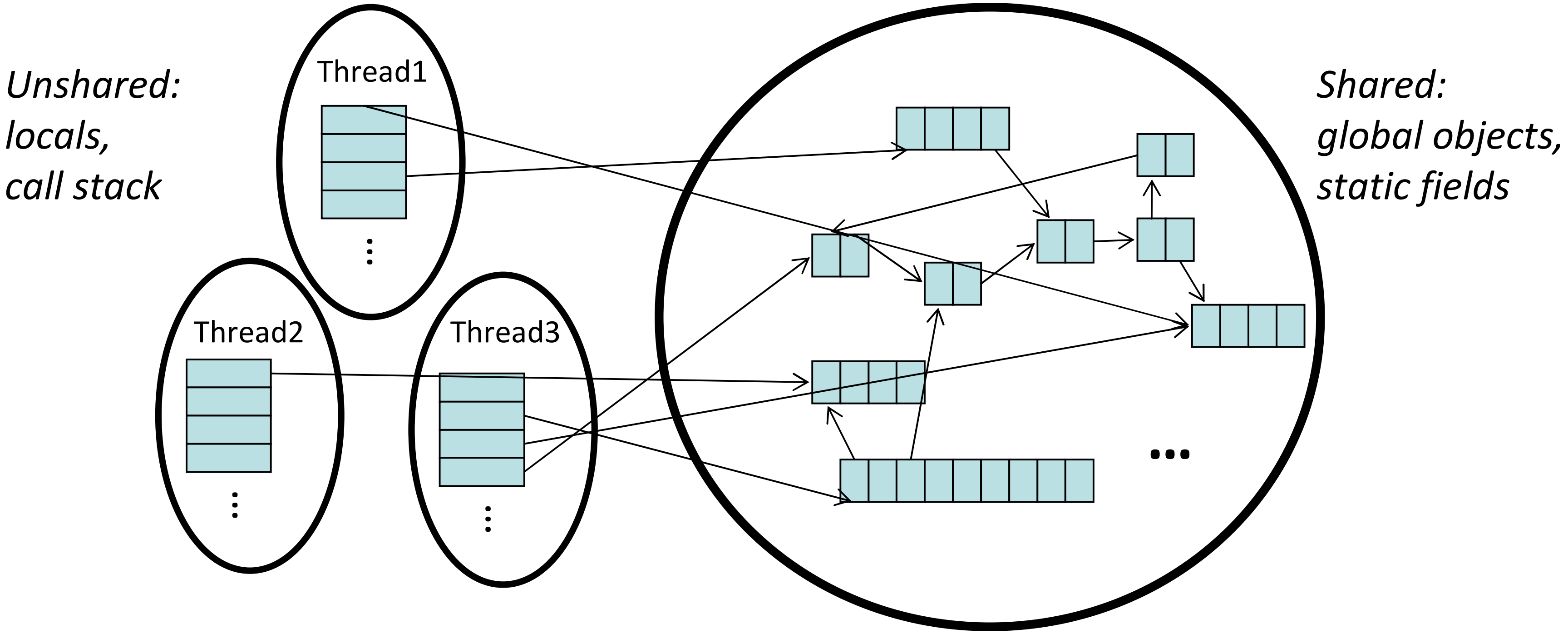
- What can you do with multiple CPUs (or cores)?
    - Run multiple different programs at the same time (**processes**).
        - Example: Core 1 runs Firefox; Core 2 runs iTunes; Core 3 runs Eclipse...
        - Technically, programs receive "time slices" of attention from cores.
        - Your OS (Windows, OSX, Linux) already does this for you.

- *Do multiple things at once within the same program* (**threads**).
    - This will be our focus. More difficult; must be done manually.
    - Requires rethinking everything about our algorithms, from how to implement data-structure operations, to Big-Oh, to ...

- Writing correct/fast parallel code is much harder than sequential.
    - Especially in common languages like Java and C.

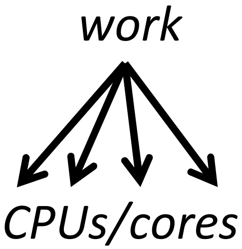
Shared memory model

- Each thread has its own unshared call stack and local variables.
  - Some objects are shared between multiple threads: Any objects declared at a global scope or passed from one to another.
- Separate processes do not share memory with each other.

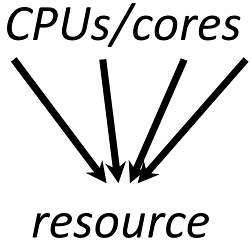


Parallel vs. concurrent

- **parallel:** Using multiple processing resources (CPUs, cores) at once to solve a problem faster.
  - Example: A sorting algorithm that has several threads each sort part of the array.



- **concurrent:** Multiple execution flows (e.g. threads) accessing a shared resource at the same time.
  - Example: Many threads trying to make changes to the same data structure (a global list, map, etc.).



- Many programmers confuse these two concepts.
  - Threads are often used to implement both.

Thread and Runnable

- To run some code in its own thread:
  - Write a class that implements the `Runnable` interface.
    - Its `run` method contains the code you want to execute.
  - Construct a new `Thread` object, passing your runnable to it.
    - Then `start` the thread.

  -
    ```
    public interface Runnable {  // implement this
        public void run();
    }
    ```

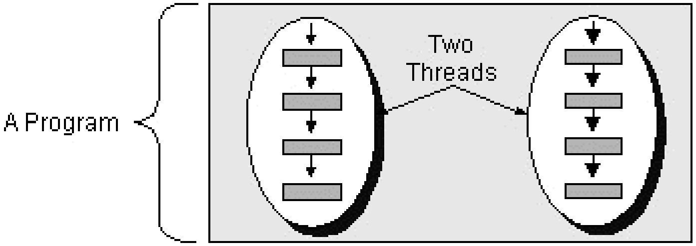
  - `public class Thread {            // construct one`
    - `public Thread(Runnable runnable)`
    - `public void start()`

Runnable example

```
public class MyRunnable implements Runnable {
    public void run() {
        // perform a task...
    }
}

...
Thread thread = new Thread(new MyRunnable());
thread.start();   // returns immediately
```

- Sometimes done with an *anonymous inner class*:

```
new Thread(new Runnable() {
    public void run() {
        // perform a task...
    }
}).start();
```



Waiting for a thread

- The call to `Thread`'s `start` method returns immediately.
  - Your code continues running in its own thread.
  - Cannot assume that the other thread has finished running yet.
- If you want to be sure the thread is done, call `join` on it.
  - Sometimes called a "fork/join" execution model.

```
Thread thread = new Thread(new MyRunnable());
thread.start();
System.out.println("Hello!");      // runs immediately

try {
    thread.join();          // wait for thread to finish
} catch (InterruptedException ie) {} // never happens

System.out.println("Hello!");        // runs afterward
```

Algorithm example

- Write a method named `sum` that computes the total sum of all elements in an array of integers.
  - For now, just write a normal solution that doesn't use parallelism.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98

```
// normal sequential solution
public static int sum(int[] a) {
    int total = 0;
    for (int i = 0; i < a.length; i++) {
        total += a[i];
    }
    return total;
}
```

Parallelizing the algorithm

- Write a method named `sum` that computes the total sum of all elements in an array of integers.
  - How can we parallelize this algorithm if we have 2 CPUs/cores?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98

sum1 = 22+18+12+-4+27+30+36+50 = **191**

sum2 = 7+68+91+56+2+85+42+98 = **449**

sum = sum1 + sum2 = **640**

  - Compute sum of each half of array in a thread.
  - Add the two sums together.

Initial steps

- First, write a method that sums a partial range of the array:

```
// normal sequential solution
public static int sumRange(int[] a, int min, int max) {
    int total = 0;
    for (int i = min; i < max; i++) {
        total += a[i];
    }
    return total;
}
```

Runnable partial sum

- Now write a runnable class that can sum a partial array:

```
public class Summer implements Runnable {
    private int[] a;
    private int min, max, sum;

    public Summer(int[] a, int min, int max) {
        this.a = a;
        this.min = min;
        this.max = Math.min(max, a.length);
    }

    public int getSum() {
        return sum;
    }

    public void run() {
        sum = Sorting.sumRange(a, min, max);
    }
}
```

Sum method w/ threads

- Now modify the overall sum method to run Summers in threads:

```
// Parallel version (two threads)
public static int sum(int[] a) {
    Summer firstHalf = new Summer(a, 0, a.length/2);
    Summer secondHalf = new Summer(a, a.length/2, a.length);
    Thread thread1 = new Thread(firstHalf);
    thread1.start();
    Thread thread2 = new Thread(secondHalf);
    thread2.start();
    try {
        thread1.join();
        thread2.join();
    } catch (InterruptedException ie) {}
    return firstHalf.getSum() + secondHalf.getSum();
}
```

More than 2 threads

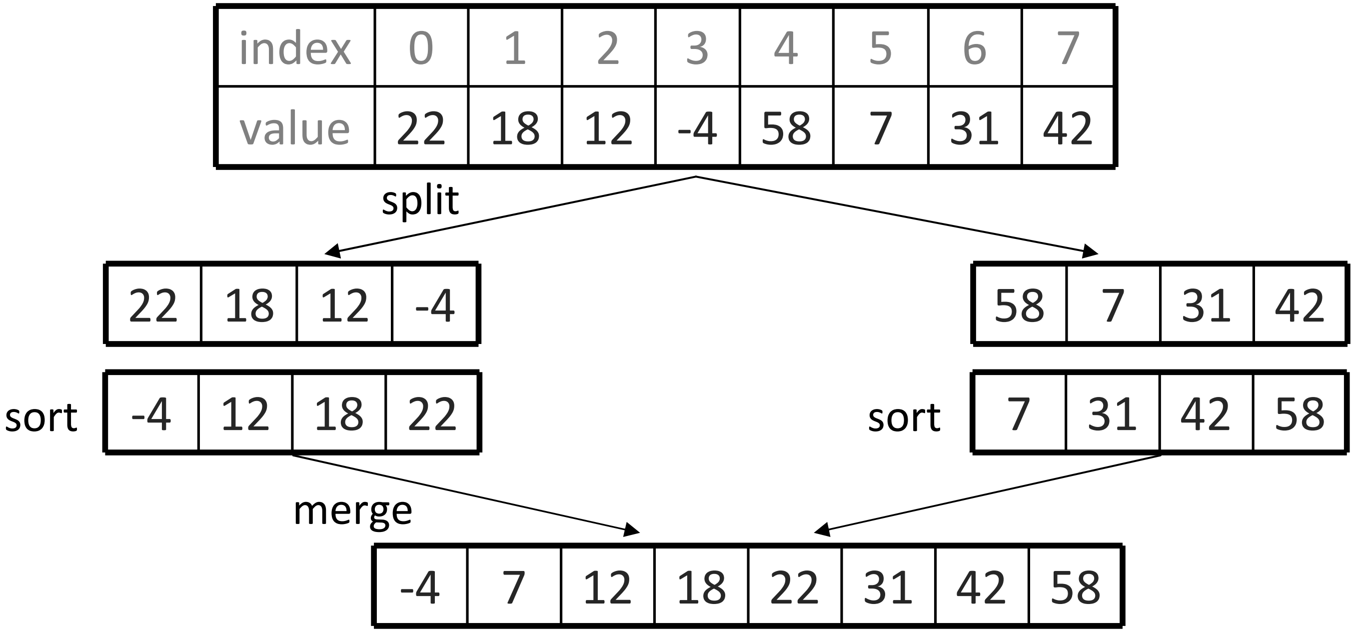
```
public static int sum(int[] a) {  // many threads version
    int threadCount = 5;   // what number is best?
    int len = (int) Math.ceil(1.0 * a.length / threadCount);
    Summer[] summers = new Summer[threadCount];
    Thread[] threads = new Thread[threadCount];
    for (int i = 0; i < threadCount; i++) {
        summers[i] = new Summer(a, i*len, (i+1)*len);
        threads[i] = new Thread(summers[i]);
        threads[i].start();
    }
    try {
        for (Thread t : threads) {
            t.join();
        }
    } catch (InterruptedException ie) {}

    int total = 0;
    for (Summer summer : summers) {
        total += summer.getSum();
    }
    return total;
}
```

How many threads to use?

- You can find out how many cores/CPUs your machine has:
  - `int cores = Runtime.getRuntime().availableProcessors();`
- You'd think that would be the ideal number of threads.
  - Sometimes yes, sometimes no.
  - Your program does not always get all of the cores to use.
- Too few threads can be bad (core(s) sit idle).
- Too many threads can be bad (overhead of creating Threads).
  - A bad ratio can slow the algorithm: e.g. 8 threads for 6 cores.
  - If threads are lightweight to create, making tons of threads can be very effective (e.g. make 1000 threads, set them all loose!).
    - Java's Threads are too heavy-weight for this to be practical.

Parallel merge sort

- How can merge sort be parallelized if we have 2 CPUs/cores?



index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42

split

22	18	12	-4

58	7	31	42

sort

-4	12	18	22

sort

7	31	42	58

merge

-4	7	12	18	22	31	42	58

- Idea:
  - Split array in half.
  - Recursively sort each half **in its own thread**.
  - Merge.

Runnable merge sort

- Write a runnable class that can merge sort an array:

```
public class MergeSortRunner implements Runnable {
    private int[] a;

    public MergeSortRunner(int[] a) {
        this.a = a;
    }

    public void run() {
        mergeSort(a);
    }
}
```

Merge sort w/ threads

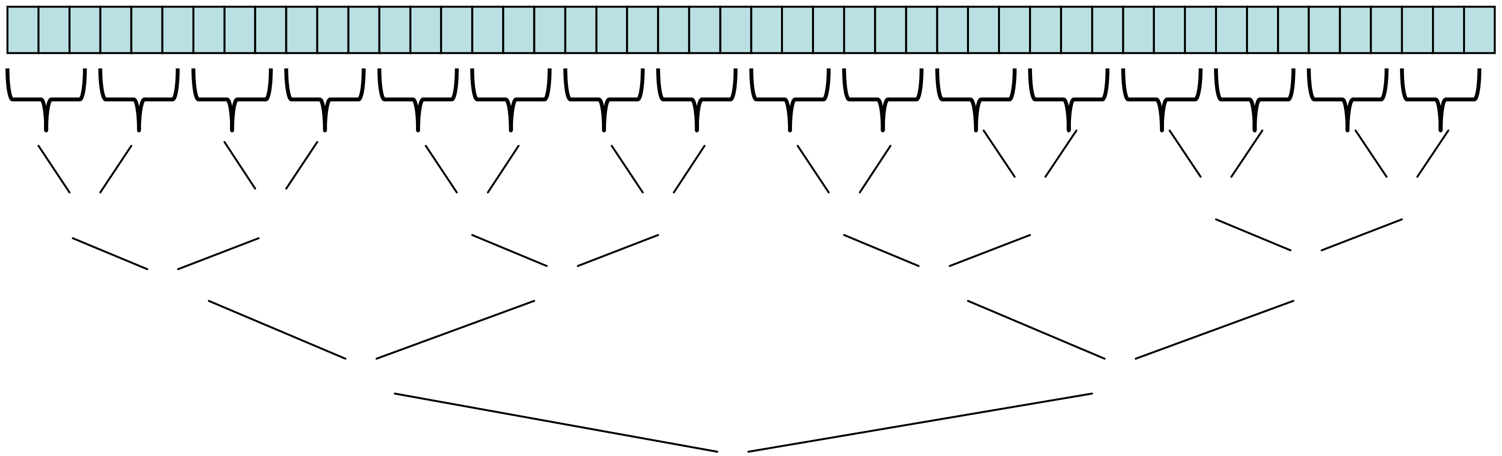
- Now modify the merge sort method to sort in threads:

```
// Parallel version (two threads)
public static void parallelMergeSort(int[] a) {
    if (a.length < 2) { return; }

    // split array in half
    int[] left  = Arrays.copyOfRange(a, 0, a.length / 2);
    int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

    // sort each half (in parallel)
    Thread lThread = new Thread(new MergeSortRunner(left));
    Thread rThread = new Thread(new MergeSortRunner(right));
    lThread.start();
    rThread.start();
    try {
        lThread.join();
        rThread.join();
    } catch (InterruptedException ie) {}

    // merge them back together
    merge(left, right, a);
}
```

More than 2 threads?

- If we want to use more than 2 threads, it is tricky to code.
  - Have to keep an array of threads/runnables.
  - Tough to merge all the partial results together when done.

- A better way: **divide-and-conquer parallelism**
  - Have each call spawn two threads, which spawn two threads, ...
  - Each thread merges its two sub-threads; easier to manage



Modified Runnable

- Modify the runnable class to accept a *level*:
  - Level 0 : base case; just do a sequential merge sort.
  - Level *K* : spawn two threads at level *K*-1 to sort each half.

```
public class MergeSortRunner implements Runnable {
    private int[] a;
    private int level;

    public MergeSortRunner(int[] a, int level) {
        this.a = a;
        this.level = level;
    }

    public void run() {
        parallelMergeSort(a, level);
    }
}
```

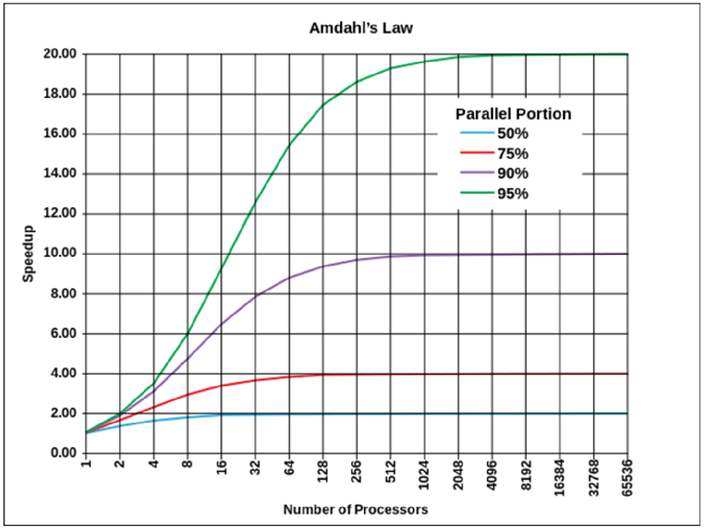
Merge sort w/ threads

- Now modify the merge sort method to use levels:

```
// Parallel version (many threads)
public static void parallelMergeSort(int[] a) {
    parallelMergeSort(a, 3);   // 3 levels => 2^3=8 threads
}

private static void parallelMergeSort(int[] a, int level) {
    if (a.length < 2) { return; }
    if (level == 0)   { mergeSort(a); return; }

    // split array in half
    int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
    int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

    // sort each half (in parallel)
    Thread lThread = new Thread(new MergeSortRunner(left, level-1));
    Thread rThread = new Thread(new MergeSortRunner(right, level-1));
    lThread.start();
    rThread.start();
    try {
        lThread.join();
        rThread.join();
    } catch (InterruptedException ie) {}

    // merge them back together
    merge(left, right, a);
}
```

Amdahl's Law

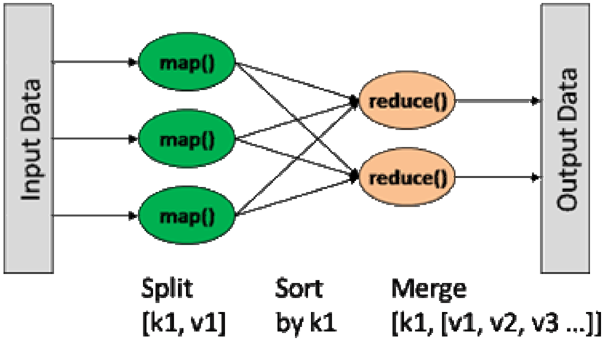
- **Amdahl's Law**: The speedup that can be achieved by parallelizing a program is limited by the sequential fraction of the program.
    - Example: If 33% of the program must be performed sequentially, no matter how many processors you use, you can only get a 3x speedup.
    - An example of *diminishing returns* from adding more processors.
        - "Nine couples can't make a baby in one month."
    - Therefore, part of the trick becomes learning how to minimize the portion of the program that must be performed sequentially.
        - Making better parallel algorithms.



Map/Reduce

- **map/reduce**: A strategy for implementing parallel algorithms.
  - *map*: A master worker takes the problem input, divides it into smaller sub-problems, and distributes the sub-problems to workers (threads).
  - *reduce*: The master worker collects sub-solutions from the workers and combines them in some way to produce the overall answer.
    - Our multi-threaded merge sort is an example of such an algorithm.

- Frameworks and tools have been written to perform map/reduce.
  - MapReduce framework by Google
  - Hadoop framework by Yahoo!
  - related to the ideas of *Big Data* and *Cloud Computing*
  - also related to *functional programming*



Thread object methods

Method name	Description
`getPriority()` `setPriority(`**int**`)`	gets/sets this thread's running priority. Possible values: Thread.MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY
`getName()` `setName(`**name**`)`	gets/sets the name of this thread as a string
`getState()`	thread's state. One of Thread.State.NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, or TERMINATED
`interrupt()`	stops the thread's current time slice
`isAlive()`	returns `true` if the thread is in runnable state
`join()` **`join(ms)`**	waits indefinitely, or for a given number of milliseconds, for the thread to finish running
`start()`	puts a thread into runnable state
`stop()`	instructs a thread to stop immediately (*deprecated*)

Thread static methods

Static method name	Description
`activeCount()`	number of currently runnable/active threads
`dumpStack()`	causes current thread to print a stack trace
`getAllStackTraces()`	returns stack trace data for all currently running threads
`getCurrentThread()`	returns the current code's active thread
`holdsLock(`**obj**`)`	returns `true` if current thread has locked the given object
`sleep(ms)`	causes the current thread to wait for at least the given number of ms before continuing
`yield()`	temporarily pauses the current thread to let others run

Sleeping a thread

```
try {
    Thread.sleep(ms);
} catch (InterruptedException ie) {}
```

- Causes current thread to wait for the given number of milliseconds.
- If the program has other threads, they will be given a chance to run.
- Useful for writing code that checks for an update periodically.

```
// check for new network messages every 2 sec
while (!done) {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException ie) {}
    myMessageQueue.read();
    ...
}
```