Multithreaded Algorithms [1]

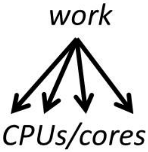
Jie Wang

University of Massachusetts Lowell
Department of Computer Science

[1] Some of the slides presented are borrowed from University of Washington Professor Marty Stepp's slides.

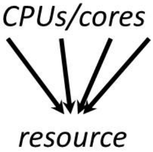
Parallel Computing

- **Sequential algorithms** are algorithms for a computing device with a single processor and random-access memory. These algorithms are also referred to as **singlethreaded algorithms**.
- **Multithreaded algorithms** are algorithms for a computing device with multiple processors (CPUs, cores) and shared memory.
- **Parallel computing**: Multiple processors simultaneously solve a problem.
  For example, split a computing task into several parts and assign a processor to solve each part at the same time.



Concurrent Computing

- **Concurrent computing**: Multiple execution flows (e.g. threads) access a shared resource at the same time.
  For example, concurrent computing allows the same data structure to be updated by multiple threads.



Computing with Multiple Cores

- Run multiple programs at the same time.
  For example: Core 1 runs Chrome; Core 2 runs iTunes; Core 3 runs Eclipse. OS (Windows, OSX, Linux) gives programs "time slices" of attention from cores.
- Do multiple things in the same program with multiple threads.
  - Need to rethink everything about algorithms.
  - Harder to write parallel code, especially in Java, C, C++, and other common languages.
- Each thread is given its own memory for unshared calls and local variables.
  - Global objects are shared between multiple threads.
- Separate processes do not share memory with each other.

Dynamic Multithreading

- Specify parallelism without worrying about communications protocols, load balancing, and other vagaries of static-thread programming.
- The concurrency platform contains a scheduler for load balancing.
  - Two features: (1) nested parallelism; (2) parallel loops.
  - Keywords: **parallel**, **spawn**, **sync**
- Example: Fibonacci numbers (inefficient):

$\text{F}\small{\text{IB}}(n)$

if $n \leq 1$
  return n
else $x = \text{F}\small{\text{IB}}(n-1)$
  $y = \text{F}\small{\text{IB}}(n-2)$
  return $x + y$

Multithreading Fibonacci

- $\text{FIB}(n-1)$ and $\text{FIB}(n-2)$ can be executed independently, and so can be done in parallel.

$\text{P-FIB}(n)$

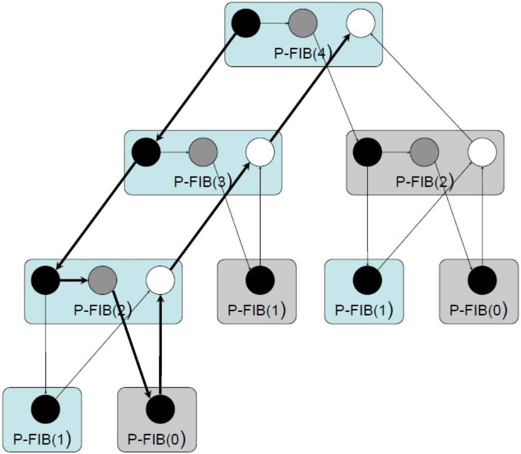
if $n \leq 1$
    return n
else $x =$ **spawn** $\text{FIB}(n-1)$
    $y = \text{FIB}(n-2)$
    sync
    return $x + y$

- Nested parallelism occurs when the keyword **spawn** proceeds a function, where the parent code may continue to execute in parallel with the spawned children.
- Keyword **sync** means to wait for all its spawned children to complete.

Multithreading and Computation DAG

- Multithreaded computation can be modeled as a DAG $G = (V, E)$.
  - Nodes are instructions.
  - Edges are dependencies.
- A **strand** represents a chain of instructions without parallel keywords (i.e., no **spawn**, **sync**, or **return** from a spawn).
- Two strands S_1 and S_2 are **logically in series** if there is a directed path from S_1 to S_2. Otherwise, they are **logically in parallel**.
- A multithreaded computation is a DAG of strands embedded in a three of procedure instances; it starts with a single strand and ends also with a single strand.

Multithreaded Fibonacci Computation

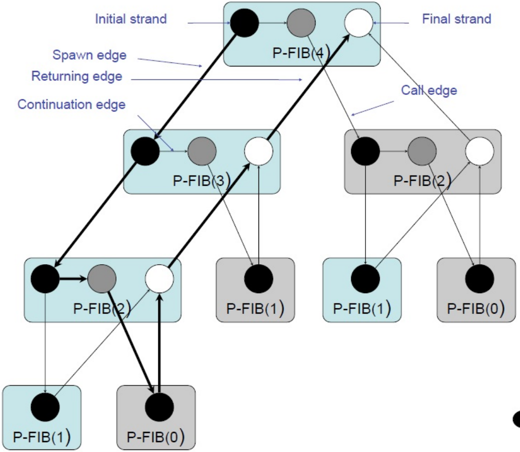


Directed acyclic graph representing P-FIB(4)

- **Circles represent strands.**
- base case or the part of the procedure up to the spawn of P-FIB (n-1) in line 4.
- part of the procedure that calls P_FIB(n-2) in line 5 up to the **sync** in line 6, where it suspends until the spawn of P-FIB(n-1) returns.
- part of the procedure after the **sync** where it sums x and y up to the point where it returns the result.
- Rectangles contain strands belonging to the same procedure
- for spawned procedures
- for called procedures.

```
P-FIB( n )
1  if n ≤ 1
2      return n
3  else
4      x = spawn FIB( n – 1 )
5      y = FIB( n – 2 )
6      sync
7      return x + y
```

Multithreaded Fibonacci Computation Continued



Directed acyclic graph representing P-FIB(4)

- **Circles represent strands.**
- base case or the part of the procedure up to the spawn of P-FIB (n-1) in line 4.
- part of the procedure that calls P_FIB(n-2) in line 5 up to the **sync** in line 6, where it suspends until the spawn of P-FIB(n-1) returns.
- part of the procedure after the **sync** where it sums x and y up to the point where it returns the result.
- Rectangles contain strands belonging to the same procedure
- for spawned procedures
- for called procedures.

```
P-FIB( n )
1  if n ≤ 1
2      return n
3  else
4      x = spawn FIB( n – 1 )
5      y = FIB( n – 2 )
6      sync
7      return x + y
```

Complexity Measures

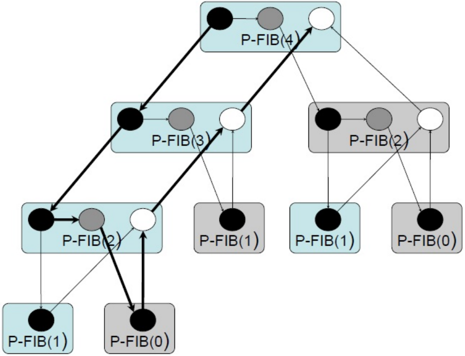
- Assuming ideal parallelism: **sequentially consistent** shared memory among multiple processors.
    - The memory behaves as if each processor's instructions were executed sequentially according to some global linear order.
- **Work**: the total time to execute the entire computation on one processor.
    - The work is equal to the sum of the times taken by each strand.
    - It is simply the number of nodes in a computation DAG if each strand takes a unit time.
- **Span**: the longest time to execute all strands along any path in the DAG.
    - It is the number of nodes on a longest path (a.k.a. critical path) if each strand takes a unit time.
    - A critical path in a DAG can be found in $O(|V| + |E|)$ time.

Work Law and Span Law

- Let T_P denote the runtime of a multithreaded algorithm with P processors.
- Let T_1 denote the work to be done, which is the runtime on a single processor.
- Let T_∞ denote the span.
- **Work Law**: $T_P \geq T_1/P$.
- **Span Law**: $T_P \geq T_\infty$.
- **Speedup**: $T_1/T_P \leq P$.
  - Linear speedup: $T_1/T_P = \Theta(P)$.
  - Perfect speedup: $T_1/T_P = P$.
- **Parallelism**: T_1/T_∞ is the maximum speedup that can be achieved if the number of processors is unbounded.
  - Beyond the parallelism, the more processors are used the less perfect the speedup. This is because if $P \gg T_1/T_\infty$, then $T_1/T_P \ll P$.

The Speedup of Multithreaded Fibonacci Cannot Be Much Larger than 2.

- **Example:** *P-FIB(4)*
  - $W = 17$ *time units*
  - $T_\infty = 8$ *time units*



Directed acyclic graph representing P-FIB(4)

Parallel Slackness

- Slackness measure: $T_1/(PT_\infty)$.
- If $T_1/(PT_\infty) < 1$, then perfect speedup cannot be achieved.
  - This is because $T_1/T_P \leq T_1/T_\infty < P$.
- If the slackness decreases from 1 toward 0, the speedup diverges increasingly further from being perfect.

Greedy Scheduling

- In addition to minimizing the work and span, strands must also be scheduled efficiently onto processors.
- A multithreaded scheduler schedules the computation without advance knowledge of when strands will be spawned or synced.
  - Online distributed scheduler: Ideal but hard to analyze.
  - Online centralized scheduler: not ideal but easy to analyze. Example: greedy schedulers.
- A greedy scheduler assigns as many strands to processors as possible in each step.
  - **Complete step**: At least P strands are ready to execute in the step.
  - **Incomplete step**: Fewer than P strands are ready to execute in the step.
- The work law implies that the best possible runtime is $T_P = T_1/P$.
- The span law implies that the best possible runtime is $T_P = T_\infty$.
- The upper bound of any greedy scheduler is $T_1/P + T_\infty$.

Greedy Scheduler Upper Bound

Theorem 27.1. A greedy scheduler on an ideal parallel computer executes a multithreaded computation in time $T_P \leq T_1/P + T_\infty$.

Proof. Let S_C and S_I denote the number of complete steps and incomplete steps. Then $T_P \leq S_C + S_I$.

We first estimate S_C the number of complete steps. If $S_C \geq \lfloor T_1/P \rfloor + 1$, then the total work of complete steps is at least

$$\begin{aligned} P(\lfloor T_1/P \rfloor + 1) &= P\lfloor T_1/P \rfloor + P \\ &= T_1 - (T_1 \bmod P) + P \\ &> T_1. \end{aligned}$$

This is impossible. Thus, $S_C \leq \lfloor T_1/P \rfloor$.

Greedy Scheduler Upper Bound Continued

Let G be the computation DAG and, w.l.o.g. assume that each strand runs in unit time.

- Let G' be the subgraph yet to be executed at the start of the incomplete step.
- Let G'' be the subgraph remaining to be executed after the incomplete step.

Let L be the length of a longest path in G'. Then the length of a longest in G'' must be $L - 1$.
Since $L \leq T_\infty$, we have $S_I \leq T_\infty$.
This completes the proof.

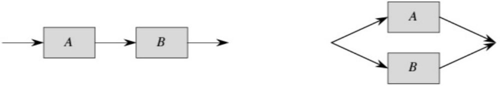
Approximation Ratio

Corollary 27.2. The runtime T_P of a greedy scheduler is an approximation to runtime T_P^* of the optimal scheduler within a factor of 2.
Proof. The work law and span law imply that $T_P^* \geq \max\{T_1/P, T_\infty\}$.
From Theorem 27.1:

$$
\begin{aligned}
T_P &\leq T_1/P + T_\infty \\
&\leq 2\max\{T_1/P, T_\infty\} \\
&\leq 2T_P^*.
\end{aligned}
$$

Corollary 27.3. If $P \ll T_1/T_\infty$, then $T_1/T_P \approx P$.
Proof. Since $P \ll T_1/T_\infty$, we have $T_\infty \ll T_1/P$. Thus,
$T_P \leq T_1/P + T_\infty \approx T_1/P$.
On the other hand, $T_P \geq T_1/P$ by the work law. Thus, $T_1/T_P \approx P$.

Analysis of Parallelism



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(a)

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(b)

To analyze the parallelism of the multithreaded Fibonacci $\text{P-FIB}(n)$:

$$\begin{aligned} T_\infty(n) &= \max\{T_\infty(n-1), T_\infty(n-2)\} + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) \\ &= \Theta(n). \end{aligned}$$

Since $T_1(n) = \Theta(\Phi^n)$, we have

$$T_1(n)/T_\infty(n) = \Theta(\Phi^n/n).$$

Parallel Loops

Consider matrix-vector multiplication, where $A = (a_{ij})_{n \times n}$ is a matrix, $x = (x_j)_n$ is an n-vector, and $y = (y_i)_n$ is the resulting vector with

$$y_i = \sum_{j=1}^{n} a_{ij} x_j .$$

```
MAT-VEC-SEQ( A, x )
n = A.rows
Let y be a new vector of length n
for i = 1 to n
    y_i = 0
for i = 1 to n
    for j = 1 to n
        y_i = y_i + a_ij x_j
return y
```

Serial code for matrix-vector multiplication

```
MAT-VEC( A, x )
n = A.rows
Let y be a new vector of length n
parallel for i = 1 to n
    y_i = 0
parallel for i = 1 to n
    for j = 1 to n
        y_i = y_i + a_ij x_j
return y
```

Parallel code for matrix-vector multiplication

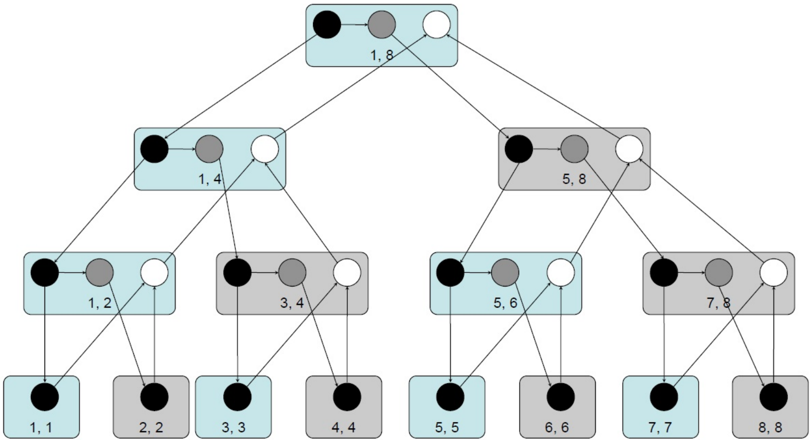
Parallel Loops Continued

- A compiler implements each **parallel for** loop as a divide-and-conquer subroutine using nested parallelism.
- For example, the compiler implements the **parallel for** loop in lines 5–7 with the call to $\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, 1, n)$, where

$$\begin{bmatrix} a_{11} & \cdots & a_{1\,n} \\ a_{mid\;1} & \cdots & a_{mid\;n} \\ a_{mid+1\;1} & \cdots & a_{mid+1\;n} \\ a_{n\;1} & \cdots & a_{n\;n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{mid} \\ x_{mid+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_{mid} \\ y_{mid+1} \\ \vdots \\ y_n \end{bmatrix}$$

```
MAT-VEC-MAIN-LOOP( A, x, y, n, i, i' )
1  if i == i'
2     for j = 1 to n
3          y_i = y_i + a_ij x_j
4  else mid = floor((i+ i')/2)
5          spawn MAT-VEC-MAIN-LOOP( A, x, y, n, i, mid)
6          MAT-VEC-MAIN-LOOP( A, x, y, n, mid+ 1, i' )
7          sync
```

DAG for $\text{MAT-VEC-MAIN-LOOP}(A, x, y, 8, 1, 8)$



Parallelism

It is straightforward that $T_1(n) = \Theta(n^2)$.

$$
\begin{aligned}
T_\infty(n) &= \Theta(\log n) + \max_{1 \le i \le n} iter_\infty(i) \\
&= \Theta(n).
\end{aligned}
$$

Thus, the parallelism $T_1(n)/T_\infty(n) = \Theta(n^2/n) = \Theta(n)$.

Race Conditions

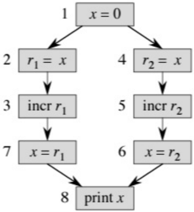
A *determinacy race condition* occurs when two logically parallel instructions access the same memory location and at least one of the instructions perform a write.

```
RACE-EXAMPLE( )
x = 0
parallel for i = 1 to 2
    x = x + 1
print x
```

Load-store steps:
1. Read x from memory to register
2. Increment value in register
3. Write back register to memory

Methods to avoid determinacy races:
Mutual exclusion locks
Others.
Assumption:
Strand that operate in parallel are independent.

Illustration of the Determinacy Race



(a)

step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

(b)

A Chess Lesson

- This is a true story (with simplification on timings)
- A multitreaded chess-playing program was built with $T_1 = 2048$ and $T_\infty = 1$.
- It was prototyped on a 32-processor computer with $T_{32} = 2048/32 + 1 = 65$ and will be implemented on a 512-processor machine.
- Later an optimization was found with $T_1' = 1024$ and $T_\infty' = 8$. Thus, $T_{32}' = 1024/32 + 8 = 40$.
- On a 512-node computer, using the original program we have $T_{512} = 2048/512 + 1 = 5$. With the optimized program we have $T_{512}' = 1024/512 + 8 = 10$.
- Thus, although the optimized algorithms runs better on a 32-node machine, it's worse than the original program on a 512-node machine.

Thread and Runnable

- ```
  public interface Runnable {   // implement this
      public void run();
  }
  ```
- ```
  public class Thread {         // construct one
  ```
  - `public Thread(Runnable runnable)`
  - `public void start()`

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

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

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
```

MultiThread Example: Parallel Summation

- 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

```java
// 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

- 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 with Threads

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

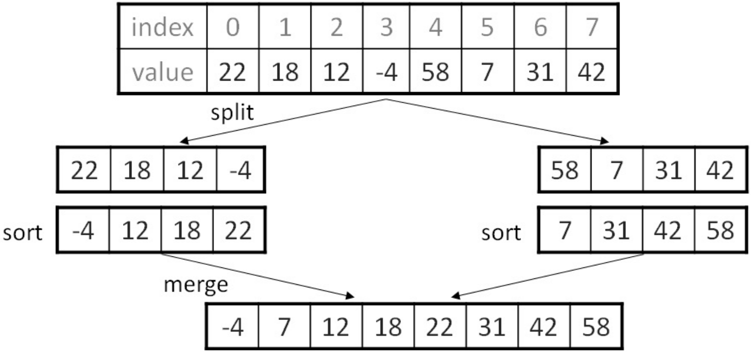
```
// Parallel version (two threads)
public static int sum(int[] a) {
    Summer firstHalf = new Summer(a, 0, a.length/2);
    Summer secondHalf = new Summer(a, a.length/2, a.length);
    Thread thread1 = new Thread(firstHalf);
    thread1.start();
    Thread thread2 = new Thread(secondHalf);
    thread2.start();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

Three or more 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;
}
```

MultiThread Example: Parallel MergeSort

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



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

Runnable MergeSort

- 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);
    }
}
```

MergeSort with Multiple Threads

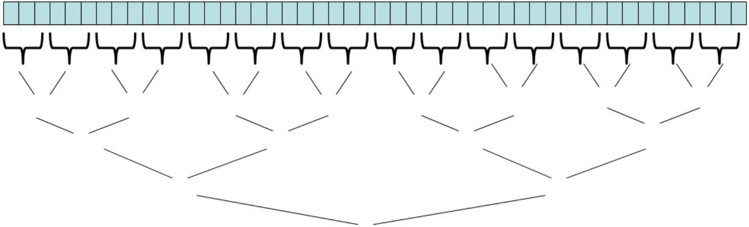
- Now modify the merge sort method to sort in threads:

```
// 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);
}
```

Three or more 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



Map/Reduce (Google's Invention)

- **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*