Principles of Software Construction: Objects, Design and Concurrency

The Perils of Concurrency, part 3

Can't live with it.
Can't live without it.

Jonathan Aldrich
Charlie Garrod
• Problems with your Homework 6 partner?
  ▪ Email me and/or Jonathan

• Homework 6c code due tonight
  ▪ Using a late day allows you to turn in the second part of hw6c late, and also lab 7 late
Last time: Static analysis and JSure

- Annotate design intent for concurrent programs
- Aside: redundancy and robustness
Before that: concurrency

• Basic concurrency in Java
  ▪ Primitive concurrency control mechanisms

• Race conditions
  ▪ check-then-act

• Deadlock

• Livelock

java.util.concurrent.ConcurrentHashMap
Today: Concurrency, part 3

- Higher-level languages, briefly
- Potpourri of parallel algorithms
- Distributed map-reduce frameworks
Recall: work, breadth, and depth

- **Work**: total effort required
  - area of the shape

- **Breadth**: extent of simultaneous activity
  - width of the shape

- **Depth (or span)**: length of longest computation
  - height of the shape
Concurrenty at the language level

• **Consider:**

```java
int sum = 0;
Iterator i = list.iterator();
while (i.hasNext()) {
    sum += i.next();
}
```

• **In python:**

```python
sum = 0;
for item in lst:
    sum += item
```
Parallel quicksort in Nesl

function quicksort(a) =
  if (#a < 2) then a
  else
    let pivot = a[#a/2];
    lesser = {e in a| e < pivot};
    equal = {e in a| e == pivot};
    greater = {e in a| e > pivot};
    result = {quicksort(v): v in [lesser,greater]};
    in result[0] ++ equal ++ result[1];

• Operations in {} occur in parallel

• What is the total work? What is the depth?
  • What assumptions do you have to make?
Prefix sums (a.k.a. inclusive scan)

- **Goal:** given array \( x[0...n-1] \), compute array of the sum of each prefix of \( x \)
  
  \[
  \left[ \text{sum}(x[0...0]), \\
  \text{sum}(x[0...1]), \\
  \text{sum}(x[0...2]), \\
  \vdots \\
  \text{sum}(x[0...n-1]) \right]
  \]

- **e.g.,** \( x = \left[ 13, 9, -4, 19, -6, 2, 6, 3 \right] \)
  
  prefix sums: \( \left[ 13, 22, 18, 37, 31, 33, 39, 42 \right] \)
Parallel prefix sums

- **Intuition:** If we have already computed the partial sums \( \text{sum}(x[0...3]) \) and \( \text{sum}(x[4...7]) \), then we can easily compute \( \text{sum}(x[0...7]) \).

- **Code:**

```python
prefix_sums(x):
    for d in 0 to (\lg n)-1:       // d is depth
        parallelfor i in 2^d to n-1:
            newx[i] = x[i-2^d] + x[i]
        x = newx
```
Parallel prefix sums

- **Intuition:** If we have already computed the partial sums \( \text{sum}(x[0...3]) \) and \( \text{sum}(x[4...7]) \), then we can easily compute \( \text{sum}(x[0...7]) \).

- **Code:**

  ```python
  def prefix_sums(x):
      for d in 0 to (\log_2 n) - 1:
          parallel for i in \(2^d\) to n-1:
              newx[i] = x[i-\(2^d\)] + x[i]
          x = newx
  ```

- **e.g.,** \( x = \left[ 13, 9, -4, 19, -6, 2, 6, 3 \right] \)
Map

- \( \text{map}(f, x[0...n-1]) \)
  - Apply the function \( f \) to each element of list \( x \)

**E.g., in Python:**

```python
def square(x):
    return x*x
map(square, [1, 2, 3, 4]) would return [1, 4, 9, 16]
```

- Parallel map implementation is trivial
  - What is the work? What is the depth?
Reduce

- \texttt{reduce}(f, \ x[0\ldots n-1])
  - Repeatedly apply binary function $f$ to pairs of items in $x$, replacing the pair of items with the result until only one item remains
  - One sequential Python implementation:
    ```python
def reduce(f, x):
    if len(x) == 1: return x[0]
    return reduce(f, [f(x[0],x[1])] + x[2:])
```
  
  - e.g., in Python:
    ```python
def add(x,y): return x+y
reduce(add, [1,2,3,4])
```
    would return 10 as
    ```python
reduce(add, [1,2,3,4])
reduce(add, [3,3,4])
reduce(add, [6,4])
reduce(add, [10]) \rightarrow 10
```
Reduce with an associative binary function

- If the function $\cdot$ is associative, the order $\cdot$ is applied does not affect the result

\[
1 + ((2+3) + 4) \quad 1 + (2 + (3+4)) \quad (1+2) + (3+4)
\]

- Parallel reduce implementation is also easy
  - What is the work? What is the depth?
The distributed map-reduce idea is just:
\[ \text{reduce}(f_2, \text{map}(f_1, x)) \]

Key idea: a "data-centric" architecture
- Send function \( f_1 \) directly to the data
- Execute it concurrently
- Then merge results with reduce
- Also concurrently
Map and Reduce with keys (as told by Google)

- E.g., for each word on the Web, count the number of times that word occurs
  - For Map: \texttt{key1} is a document name, \texttt{value} is the contents of that document
  - For Reduce: \texttt{key2} is a word, \texttt{values} is a list of the number of counts of that word

```java
Map(String key1, String value):
    for each word \texttt{w} in value:
        EmitIntermediate(w, "1");

Reduce(String key2, Iterator values):
    int result = 0;
    for each \texttt{v} in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

\[
\text{Map: (key1, v1)} \rightarrow \text{(key2, v2)*} \quad \text{Reduce: (key2, v2*)} \rightarrow \text{v2*}
\]

\[
\text{MapReduce: (key1, v1)*} \rightarrow \text{(key2, v2*)*}
\]

\[
\text{MapReduce: (docName, docText)*} \rightarrow \text{(word, wordCount)*}
\]
Map and Reduce with keys (as told by Google)

- **Master:**
  - Assigns tasks to map and reduce workers
  - Pings workers to test for failures

- **Reduce workers:**
  - Remote read of key/value pairs
  - Reduce for each key
A map-reduce task for you

- Use map and reduce to generate an inverted index
  - E.g., given (docName, docContents) pairs for each document on the Web, build (word, docNameList) pairs for each word on the web, where docNameList is a list of all the document names containing that word

- Start by figuring out, for map and reduce: what are the keys and what are the values? I.e., what are the intermediate (key, value) pairs?

- Then describe pseudocode for map and reduce
Next time:

- Higher-level Java tools for concurrent programming