Anncouncements

- PS1 out this evening

Functional Programming and Parallelism

CS149
Lecture 7

TM Implementation Summary 1

- TM implementation
  - Data versioning: eager or lazy
  - Conflict detection: optimistic or pessimistic
    - Granularity: object, word, cache-line, ...

- Software TM systems
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, ...)
    - Transactional record per data (locked/version)

Why is STM Slow?

- Measured single-thread STM performance
  - 1.8x – 5.6x slowdown over sequential
  - Most time goes in read barriers & validation
  - Most apps read more data than they read

TM Implementation Summary 2

- Intel McRT STM
  - Eager versioning, optimistic reads, pessimistic writes
  - Read barriers check version number
  - Write barrier acquire locks
  - Commit validates the read-set and releases locks
  - Periodic validation needed to avoid doomed transactions

- Optimizations
  - Decomposed barriers to allow redundancy elimination
  - No barriers for private or transaction local data
  - Contention management

TM Implementation Summary 3

- STM performance
  - 2x to 8x per thread slowdown due to instrumentation
  - Most time spent on read barriers & validation

- Hardware accelerated TM
  - Conflict detection in HW, data versioning in SW

- Hardware TM
  - Cache to store undo-log or write-buffer
  - Per cache-line R/W bits for read/write set tracking
  - Conflict detection on coherence events
HTM Performance Example

- 2x to 7x over STM performance
- Within 10% of sequential for one thread
- Scales efficiently with number of processors
- Uncommon cases not a performance challenge

Outline

- What’s hard about parallel programming
- Task decomposition
- Functional parallelism
- Functional programming (Lisp, ML) review
- MapReduce

What’s Hard About Parallel Programming

1. Finding independent tasks in the algorithm
2. Mapping tasks to execution units (e.g. threads)
3. Defining & implementing synchronization
   - Races, deadlock avoidance, memory model issues
4. Composing parallel tasks
5. Recovering from errors
6. Portable & predictable performance
7. Scalability
8. Locality management
9. All the sequential issues as well...

The Two Sides of Parallelization

- Dividing Work: Need to chop computation into parallel tasks
  - Tasks are small relatively independent units in a program
  - Goal: enough parallel tasks to keep processors busy
- Partitioning Data: Localizing data onto processors
  - Required on message passing machines
  - Very helpful on shared-memory machines
  - Goal: minimize expensive interprocessor communication

Task Decomposition

- Start from a good understanding of the problem being solved
  - What are the most computationally intensive parts of the problem
  - What are the key data structures
  - How is the data being used as the problem's solution unfolds
- Define the tasks that make up the problem and the data decomposition implied by the tasks
- Problem can sometimes naturally break down into a collection of tasks
- If task decomposition is not evident, data decomposition might be a better starting point

Where Do You Find Tasks?

- Functional decomposition
  - Functional parallelism in the application
  - Independent functions executed as parallel tasks
- Data-driven decomposition
  - Multiple tasks concurrently updating different chunks of a large data structure
  - Usually loop based
- Dataflow decomposition
  - Data assembly lines
  - Producer-consumer chains
Instruction-level Functional Parallelism

- At the instruction level, independent algebraic operations can commute – be processed in any order
- If commutative operations are applied to different memory addresses, then they can also occur at the same time
- Compilers, CPUs often do so automatically

\[ x := (a \times b) + (y \times z) \]

computation A \hspace{1cm} \text{computation B}

Higher-level Functional Parallelism

- Commutativity can apply to larger operations. If foo() and bar() do not manipulate the same memory, then there is no reason why these cannot occur at the same time

\[ x := \text{foo}(a) + \text{bar}(b) \]

computation A \hspace{1cm} \text{computation B}

Parallelism: Dependency Graphs

- Arrows indicate dependent operations
- If foo and bar do not access the same memory, there is not a dependency between them
- These operations can occur in parallel in different threads
- Write x operation waits for predecessors to complete

\[ x := \text{foo}(a) + \text{bar}(b) \]

write x

Dependency Graphs: Full Parallelism?

- Creating dependency graphs requires sometimes-difficult reasoning about tasks
- I/O and other shared resources besides memory introduce dependencies
- Very difficult to do at compile time with imperative languages (e.g. C, C++, Java)
  - Memory state
  - Pointers \Rightarrow pointer disambiguation

Functional Programming Overview/Review

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter: commutative

\[ \text{fun foo(l: int list) = sum(l) + mul(l) + length(l)} \]

Order of sum() and mul(), length() does not matter – they do not modify l
Functional Updates Do Not Modify Structures

fun append(x, lst) =  
  let lst’ = reverse lst in 
  reverse ( x :: lst’ )

The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed ⇒ appends an item

But it never modifies lst

Functions Can Be Used As Arguments

fun DoDouble(f, x) = f (f (x))

It does not matter what f does to its argument; DoDouble() will do it twice

Map

map f lst: ('a -> 'b) -> ('a list) -> ('b list)
Creates a new list by applying f to each element of the input list; returns output in order

Fold

fold f x₀ lst: ('a*'b->'b)->'b->('a list)->'b
Moves across a list, applying f to each element plus an accumulator. f returns the next accumulator value, which is combined with the next element of the list

Example

fun foo(l: int list) = 
  sum(l) + mul(l) + length(l)

How can we implement this?
Example (Solved)

fun foo(l: int list) =
  sum(l) + mul(l) + length(l)

fun sum(lst) = foldl (fn (x,a)=>x+a) 0 lst
fun mul(lst) = foldl (fn (x,a)=>x*a) 1 lst
fun length(lst) = foldl (fn (x,a)=>1+a) 0 lst

A More Complicated Fold Problem

• Given a list of numbers, how can we generate a list of partial sums?

e.g.: [1, 4, 8, 3, 7, 9] \rightarrow [0, 1, 5, 16, 23, 32]

A More Complicated Map Problem

• Given a list of words, can we: reverse the letters in each word, and reverse the whole list, so it all comes out backwards?

["my", "happy", "cat"] \rightarrow ["ym", "ppah", "tac"]

map Implementation

fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)

• This implementation moves left-to-right across the list, mapping elements one at a time

  ... But does it need to?

Implicit Parallelism In map

• In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements

  • If order of application of $f$ to elements in list is commutative, we can reorder or parallelize execution

  • This is the "secret" that MapReduce exploits

Map Reduce Motivation

• Large Scale Data Processing

  • Want to process lots of data (> 1 TB)

  • Want to parallelize across hundreds/thousands of CPUs

  ... Want to make this easy
MapReduce

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers

MapReduce Programming Model

- Borrows from functional programming
- Users implement interface of two functions:
  - `map (in_key, in_value) -> (out_key, intermediate_value) list`
    Processes input key/value pair
    Produces set of intermediate pairs
  - `reduce (out_key, intermediate_value list) -> out_value list`
    Combines all intermediate values for a particular key
    Produces a set of merged output values (usually just one)

Map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).

- `map()` produces one or more intermediate values along with an output key from the input.

Reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list

- `reduce()` combines those intermediate values into one or more final values for that same output key

- (in practice, usually only one final value per key)

Parallellism

- map() functions run in parallel, creating different intermediate values from different input data sets ⇒ data parallelism

- reduce() functions also run in parallel, each working on a different output key

- All values are processed independently

- Bottleneck: reduce phase can’t start until map phase is completely finished.
Example: Count word occurrences

```java
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += parseInt(v);
    Emit(AsString(result));
```

More Examples

- **Distributed Grep:**
  - Map() emits a line if it matches a supplied pattern
  - Reduce() is an identity function that just copies the supplied intermediate data to output

- **Count of URL Access Frequency**
  - Map() processes logs of web page requests and outputs (URL,1)
  - Reduce() adds together all values for the same URL and emits (URL, total count)

Locality

- Master program distributes tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks

Fault Tolerance

- Master detects worker failures
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  - Effect: Can work around bugs in third-party libraries!
Optimizations

- No reduce can start until map is complete:
  - A single slow disk controller can rate-limit the whole process
- Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish
- Why is it safe to redundantly execute map tasks? Wouldn’t this mess up the total computation?

MapReduce on Multicore - Performance

- Phoenix equal to P-threads if algorithm matches MR model
- P-threads is better for algorithms that do not fit MR model
  - Does not use keys, requires multiple MR iterations

Map Reduce on Multicore - Algorithms

- Word count - determine frequency of words in documents
- String match - search file with keys for an encrypted word
- Reverse Index - build reverse index for links in HTML files
- Linear regression - find the best fit line for a set of points
- Matrix multiply - dense integer matrix multiplication
  - MapReduce version introduces coarse-grain coordinate variables
- Kmeans - clustering algorithm for 3D data points
  - Multiple MapReduce invocation with translation step
- PCA - principal component analysis on a matrix
  - MapReduce version introduces coordinate variables
- Histogram - frequency of RGB components in images
  - There is no need for keys in original algorithm

MapReduce Conclusions

- MapReduce has proven to be a useful abstraction for data parallelism
- Greatly simplifies large-scale computations at Google, Yahoo, etc.
- Functional programming paradigm can be applied to large-scale applications
- Easy to use: focus on problem, let library deal with messy details
- Is not efficient for all problems