CS4961 Parallel Programming

Lecture 2: Introduction to Parallel Algorithms

Mary Hall
August 26, 2010

Homework 1 – Due 10:00 PM, Wed., Sept. 1

- To submit your homework:
  - Submit a PDF file
  - Use the "handin" program on the CADE machines
  - Use the following command:
    "handin cs4961 hw1 <prob1file>"

Problem 1:

- What are your goals after this year and how do you anticipate this class is going to help you with that? Some possible answers, but please feel free to add to them. Also, please write at least one sentence of explanation.
  - A job in the computing industry
  - A job in some other industry where computing is applied to real-world problems
  - As preparation for graduate studies
  - Intellectual curiosity about what is happening in the computing field
  - Other

Homework 1

Problem 2:

(a) Provide pseudocode (as in the book and class notes) for a correct and efficient parallel implementation in C of the parallel prefix computation (see Fig. 1.4 on page 14). Assume your input is a vector of n integers, and there are n/2 processors. Each processor is executing the same thread code, and the thread index is used to determine which portion of the vector the thread operates upon and the control. For now, you can assume that at each step in the tree, the threads are synchronized.

(b) The structure of this and the tree-based parallel sums from Figure 1.3 are similar to the parallel sort we did with the playing cards on Aug. 24. In words, describe how you would modify your solution of (a) above to derive the parallel sorting implementation.

Homework 1, cont.

Problem 3 (see supplemental notes for definitions):

Loop reversal is a transformation that reverses the order of the iterations of a loop. In other words, loop reversal transforms a loop with header

```
for (i=0; i<N; i++)
```

into a loop with the same body but header

```
for (i=N-1; i>=0; i--)
```

Is loop reversal a valid reordering transformation on the "i" loop in the following loop nest? Why or why not?

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        a[i+1][j+1] = a[i][j] + c;
```

Today's Lecture

- Parallelism in Everyday Life
- Learning to Think in Parallel
- Aspects of parallel algorithms (and a hint at complexity!)
- Derive parallel algorithms
- Discussion
- Sources for this lecture:
  - Larry Snyder, "http://www.cs.washington.edu/education/courses/524/08wi/"

Is it really harder to "think" in parallel?

- Some would argue it is more natural to think in parallel…
- … and many examples exist in daily life
- Examples?

Is it really harder to "think" in parallel?

- Some would argue it is more natural to think in parallel…
- … and many examples exist in daily life
  - House construction -- parallel tasks, wiring and plumbing performed at once *(independence)*, but framing must precede wiring *(dependence)*
    - Similarly, developing large software systems
  - Assembly line manufacture - pipelining, many instances in process at once
  - Call center - independent calls executed simultaneously (data parallel)
  - "Multi-tasking" – all sorts of variations

Reasoning about a Parallel Algorithm

- Ignore architectural details for now
- Assume we are starting with a sequential algorithm and trying to modify it to execute in parallel
  - Not always the best strategy, as sometimes the best parallel algorithms are NOTHING like their sequential counterparts
  - But useful since you are accustomed to sequential algorithms

Reasoning about a parallel algorithm, cont.

- Computation Decomposition
  - How to divide the sequential computation among parallel threads/processors/computations?
- Aside: Also, Data Partitioning (ignore today)
- Preserving Dependences
  - Keeping the data values consistent with respect to the sequential execution.
- Overhead
  - We'll talk about some different kinds of overhead

Race Condition or Data Dependence

- A ***race condition*** exists when the result of an execution depends on the ***timing*** of two or more events.
- A ***data dependence*** is an ordering on a pair of memory operations that must be preserved to maintain correctness.

Key Control Concept: Data Dependence

- **Question:** When is parallelization guaranteed to be safe?
- **Answer:** If there are no data dependences across reordered computations.
- **Definition:** Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the accesses is a write.
- **Bernstein's conditions (1966):** I_j is the set of memory locations read by process P_j, and O_j the set updated by process P_j. To execute P_j and another process P_k in parallel,

$I_j \cap O_k = \phi$ write after read

$I_k \cap O_j = \phi$ read after write

$O_j \cap O_k = \phi$ write after write

Data Dependence and Related Definitions

- Actually, parallelizing compilers must formalize this to guarantee correct code.
- Let's look at how they do it. It will help us understand how to reason about correctness as programmers.
- **Definition:**
  Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write.

  A data dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

- **Source:**
  - "Optimizing Compilers for Modern Architectures: A Dependence-Based Approach", Allen and Kennedy, 2002, Ch. 2. (not required or essential)

Data Dependence of Scalar Variables

True (flow) dependence

```
a     =
    = a
```

Anti-dependence

```
    = a
a   =
```

Output dependence

```
a   =
a   =
```

Input dependence (for locality)

```
= a

= a
```

Definition: Data dependence exists from a reference instance i to i' iff
either i or i' is a write operation
i and i' refer to the same variable
i executes before i'

Some Definitions (from Allen & Kennedy)

- **Definition 2.5:**
  - Two computations are equivalent if, on the same inputs,
    - they produce identical outputs
    - the outputs are executed in the same order
- **Definition 2.6:**
  - A reordering transformation
    - changes the order of statement execution
    - without adding or deleting any statement executions.
- **Definition 2.7:**
  - A reordering transformation preserves a dependence if
    - it preserves the relative execution order of the dependences' source and sink.

Fundamental Theorem of Dependence

- **Theorem 2.2:**
  - Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Simple Example 1: "Hello World" of Parallel Programming

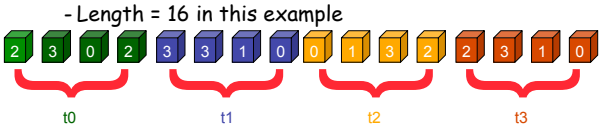
- Count the 3s in *array[]* of *length* values
- Definitional solution … Sequential program

```
count = 0;
for (i=0; i<length; i++) {
  if (array[i] == 3)
  count += 1;
}
```

Can we rewrite this to a parallel code?

Computation Partitioning

- Block decomposition: Partition original loop into separate "blocks" of loop iterations.
  - Each "block" is assigned to an independent "thread" in t0, t1, t2, t3 for t=4 threads
  - Length = 16 in this example

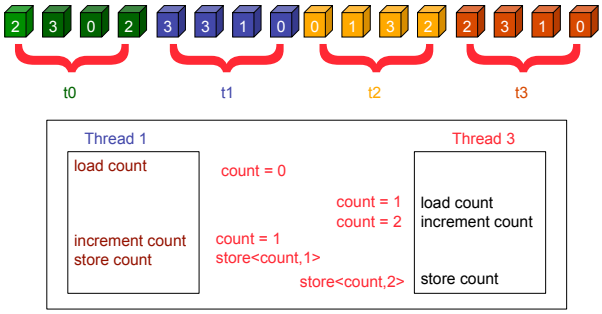


Correct? Preserve Dependences?

```
int block_length_per_thread = length/t;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++) {
    if (array[i] == 3)
        count += 1;
}
```


Data Race on Count Variable

- Two threads may interfere on memory writes



What Happened?

- Dependence on count across iterations/ threads
  - But reordering ok since operations on count are associative
- Load/increment/store must be done *atomically* to preserve sequential meaning
- Definitions:
  - Atomicity: a set of operations is atomic if either they all execute or none executes. Thus, there is no way to see the results of a partial execution.
  - Mutual exclusion: at most one thread can execute the code at any time

Try 2: Adding Locks

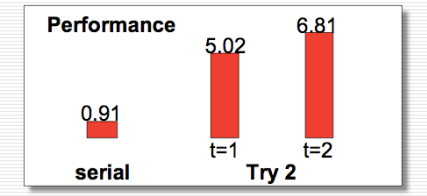
- Insert mutual exclusion (mutex) so that only one thread at a time is loading/incrementing/ storing count atomically

```
int block_length_per_thread = length/t;
mutex m;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++) {
    if (array[i] == 3) {
        mutex_lock(m);
        count += 1;
        mutex_unlock(m);
    }
}
```

Correct now. Done?

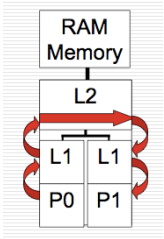
Performance Problems

- Serialization at the mutex
- Insufficient parallelism granularity
- Impact of memory system



Lock Contention and Poor Granularity

- To acquire lock, must go through at least a few levels of cache (locality)
  - Local copy in register not going to be correct
- Not a lot of parallel work outside of acquiring/releasing lock



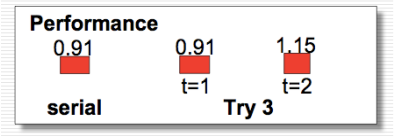
Try 3: Increase "Granularity"

- Each thread operates on a private copy of count
- Lock only to update global data from private copy

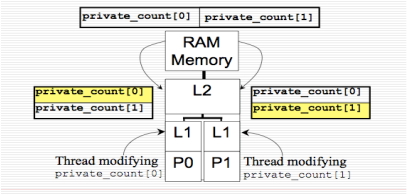
```
mutex m;
int block_length_per_thread = length/t;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++) {
    if (array[i] == 3)
        private_count[id] += 1;
}
mutex_lock(m);
count += private_count[id];
mutex_unlock(m);
```


Much Better, But Not Better than Sequential

- Subtle cache effects are limiting performance

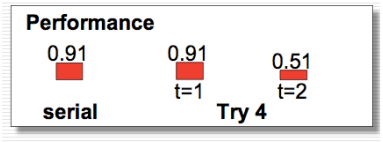


Private variable ≠
Private cache line



Try 4: Force Private Variables into Different Cache Lines

- Simple way to do this?
- See textbook for authors' solution



Parallel speedup when <t = 2>:
time(1)/time(2) = 0.91/0.51
= 1.78 (close to number of processors!)

Discussion: Overheads

- What were the overheads we saw with this example?
  - Extra code to determine portion of computation
  - Locking overhead: inherent cost plus contention
  - Cache effects: false sharing

Generalizing from this example

- Interestingly, this code represents a common pattern in parallel algorithms
- A reduction computation
  - From a large amount of input data, compute a smaller result that represents a reduction in the dimensionality of the input
  - In this case, a reduction from an array input to a scalar result (the count)
- Reduction computations exhibit dependences that must be preserved
  - Looks like *"result = result op …"*
  - Operation *op* must be associative so that it is safe to reorder them
- Aside: Floating point arithmetic is not truly associative, but usually ok to reorder

Simple Example 2: Another "Hello World" Equivalent

- Parallel Summation:
  - Adding a sequence of numbers A[0],…,A[n-1]
- Standard way to express it

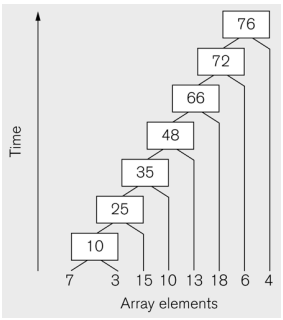
```
sum = 0;
for (i=0; i<n; i++) {
    sum += A[i];
}
```

- Semantics require:

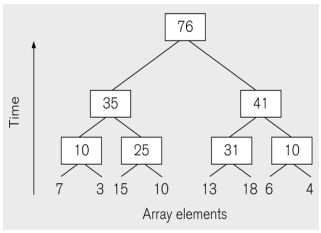
(…((sum+A[0])+A[1])+…)+A[n-1]

- That is, sequential
- Can it be executed in parallel?

Graphical Depiction of Sum Code



Original Order



Pairwise Order

Which decomposition is better suited for parallel execution.

Summary of Lecture

- How to Derive Parallel Versions of Sequential Algorithms
  - Computation Partitioning
  - Preserving Dependences and Reordering Transformations
  - Reduction Computations
  - Overheads

Next Time

- A Discussion of parallel computing platforms
- Questions about first written homework assignment