CSE524 Parallel Algorithms

Lawrence Snyder

www.cs.washington.edu/CSEP524

30 March 2010
CSE524 Parallel Algorithms

Lawrence Snyder

www.cs.washington.edu/CSE524

30 March 2010
CSE524 Parallel Algorithms

Lawrence Snyder

www.cs.washington.edu/CSE524

30 March 2010
Course Logistics

☐ Teaching Assistants: Matt Kehrt and Adrienne Wang


- There will also be occasional readings

☐ Class web page is headquarters for all data

☐ Take lecture notes -- the slides will be online sometime *after* the lecture

Informal class; ask questions immediately
Expectations

- Readings: We will cover much of the book; please read the text before class.
- Lectures will layout certain details, arguments … discussion is encouraged.
- Most weeks there will be graded homework to be submitted electronically PRIOR to class.
- Am assuming most students have access to a multi-core or other parallel machine.
- Grading: class contributions, homework assignments; no final is contemplated at the moment.
Part I: Introduction

Goal: Set the parameters for studying parallelism
Why Study Parallelism?

- After all, for most of our daily computer uses, sequential processing is plenty fast
  - It is a fundamental departure from the “normal” computer model, therefore it is inherently cool
  - The extra power from parallel computers is enabling in science, engineering, business, …
  - Multicore chips present a new opportunity
  - Deep intellectual challenges for CS -- models, programming languages, algorithms, HW, …
Facts

Single Processor

Opportunity

Moore’s law continues, so use more gates

Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter & Burton Smith
Size vs Power

- **Power5 (Server)**
  - 389mm^2
  - 120W@1900MHz

- **Intel Core2 sc (laptop)**
  - 130mm^2
  - 15W@1000MHz

- **ARM Cortex A8 (automobiles)**
  - 5mm^2
  - 0.8W@800MHz

- **Tensilica DP (cell phones / printers)**
  - 0.8mm^2
  - 0.09W@600MHz

- **Tensilica Xtensa (Cisco router)**
  - 0.32mm^2 for 3!
  - 0.05W@600MHz

Each processor operates with 0.3-0.1 efficiency of the largest chip: more threads, lower power.
Goal: To give a good idea of parallel computation

- Concepts -- looking at problems with “parallel eyes”
- Algorithms -- different resources; different goals
- Languages -- reduce control flow; increase independence; new abstractions
- Hardware -- the challenge is communication, not instruction execution
- Programming -- describe the computation without saying it sequentially
- Practical wisdom about using parallelism
Everyday Parallelism

- Juggling -- event-based computation
- House construction -- parallel tasks, wiring and plumbing performed at once
- Assembly line manufacture -- pipelining, many instances in process at once
- Call center -- independent tasks executed simultaneously

How do we describe execution of tasks?
Parallel vs Distributed Computing

Comparisons are often matters of degree

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Parallel</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Goal</td>
<td>Speed</td>
<td>Convenience</td>
</tr>
<tr>
<td>Interactions</td>
<td>Frequent</td>
<td>Infrequent</td>
</tr>
<tr>
<td>Granularity</td>
<td>Fine</td>
<td>Coarse</td>
</tr>
<tr>
<td>Reliable</td>
<td>Assumed</td>
<td>Not Assumed</td>
</tr>
</tbody>
</table>
Parallel vs Concurrent

- In OS and DB communities execution of multiple threads is **logically** simultaneous.
- In Arch and HPC communities execution of multiple threads is **physically** simultaneous.
- The issues are often the same, say with respect to races.
- Parallelism can achieve states that are impossible with concurrent execution because two events happen at once.
Consider a simple task ... 

- Adding a sequence of numbers \( A[0], \ldots, A[n-1] \)
- Standard way to express it
  
  ```
  sum = 0;
  for (i=0; i<n; i++) {
      sum += A[i];
  }
  ```

- Semantics require: \( \ldots((sum+A[0])+A[1])+\ldots)+A[n-1] \)
  - That is, **sequential**

- Can it be executed in parallel?
Parallel Summation

- To sum a sequence in parallel
  - add pairs of values producing 1st level results,
  - add pairs of 1st level results producing 2nd level results,
  - sum pairs of 2nd level results …

- That is,

Express the Two Formulations

- Graphic representation makes difference clear

- Same number of operations; different order
The Dream ...

- Since 70s (Illiac IV days) the dream has been to **compile** sequential programs into parallel object code

- Three decades of continual, well-funded research by smart people implies it’s hopeless

  - For a tight loop summing numbers, it’s doable

  - For other computations it has proved **extremely** challenging to generate parallel code, even with pragmas or other assistance from programmers
What’s the Problem?

- It’s not likely a compiler will produce parallel code from a C specification any time soon…

- Fact: For most computations, a “best” sequential solution (practically, not theoretically) and a “best” parallel solution are usually fundamentally different …

  - Different solution paradigms imply computations are not “simply” related
  
  - Compiler transformations generally preserve the solution paradigm

Therefore... the programmer must discover the \| solution
Consider computing the prefix sums

```plaintext
for (i=1; i<n; i++) {
    A[i] += A[i-1];
}
```

A[i] is the sum of the first i + 1 elements

What advantage can ||ism give?

Semantics ...

- A[0] is unchanged

... 

Comparison of Paradigms

- The sequential solution computes the prefixes … the parallel solution computes only the last

- Or does it?
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Original research on parallel prefix algorithm published by R. E. Ladner and M. J. Fischer

Parallel Prefix Computation


The Ladner-Fischer algorithm requires $2\log n$ time, twice as much as simple tournament global sum, not linear time

Applies to a wide class of operations
Parallel Compared to Sequential Programming

- Has different costs, different advantages
- Requires different, unfamiliar algorithms
- Must use different abstractions
- More complex to understand a program’s behavior
- More difficult to control the interactions of the program’s components
- Knowledge/tools/understanding more primitive
Consider a Simple Problem

☐ Count the 3s in \texttt{array[]} of \texttt{length} values

☐ Definitional solution …

- Sequential program

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

- Need to know something about machine … use multicore architecture

How would you solve it in parallel?
Divide Into Separate Parts

- Threading solution -- prepare for MT procs

```
length=16  t=4
```

```
array
2 3 0 2 3 3 1 0 0 1 3 2 2 3 1 0
Thread 0  Thread 1  Thread 2  Thread 3
```

```c
int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++)
{
    if (array[i] == 3)
        count += 1;
}
```
Divide Into Separate Parts

Threading solution -- prepare for MT procs

length=16  t=4

array

Thread 0  Thread 1  Thread 2  Thread 3

2 3 0 2 3 3 1 0 0 1 3 2 2 3 1 0

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

Doesn’t actually get the right answer
Races

- Two processes interfere on memory writes

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>count (\equiv 0)</td>
</tr>
<tr>
<td>increment</td>
<td>load</td>
</tr>
<tr>
<td>store</td>
<td>increment</td>
</tr>
<tr>
<td>count (\equiv 1)</td>
<td>store</td>
</tr>
<tr>
<td>count (\equiv 1)</td>
<td></td>
</tr>
</tbody>
</table>

`time`
Races

- Two processes interfere on memory writes

**Thread 1**
- load
- increment
- store

**Thread 2**
- load
- increment
- store

**Try 1**

- `count \equiv 0`
- `count \equiv 1`
- `count \equiv 1`

**Time**
Protect Memory References

- Protect Memory References

```c
mutex m;
for (i=start; i<start+length_per_thread; i++)
{
    if (array[i] == 3)
    {
        mutex_lock(m);
        count += 1;
        mutex_unlock(m);
    }
}
```
Protect Memory References

mutex m;
for (i=start; i<start+length_per_thread; i++)
{
  if (array[i] == 3)
  {
    mutex_lock(m);
    count += 1;
    mutex_unlock(m);
  }
}

Try 2
Correct Program Runs Slow

- Serializing at the mutex

The processors wait on each other
Closer Look: Motion of \texttt{count, m}

- Lock Reference and Contention

```c
mutex m;
for (i=start; i<start+length\_per\_thread; i++)
{
    if (array[i] == 3)
    {
        mutex\_lock(m);
        count += 1;
        mutex\_unlock(m);
    }
}
```
Accumulate Into Private Count

- Each processor adds into its own memory; combine at the end

```c
for (i=start; i<start+length_per_thread; i++)
{
    if (array[i] == 3)
    {
        private_count[t] += 1;
    }
}
mutex_lock(m);
count += private_count[t];
mutex_unlock(m);
```
Accumulate Into Private Count

- Each processor adds into its own memory; combine at the end

```c
for (i=start; i<start+length_per_thread; i++)
{
    if (array[i] == 3)
    {
        private_count[t] += 1;
    }
}
mutex_lock(m);
count += private_count[t];
mutex_unlock(m);
```

Try 3
Keeping Up, But Not Gaining

- Sequential and 1 processor match, but it’s a loss with 2 processors

<table>
<thead>
<tr>
<th>Performance</th>
<th>serial</th>
<th>Try 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t=1</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>t=2</td>
<td>1.15</td>
<td></td>
</tr>
</tbody>
</table>
False Sharing

- Private var ≠ private cache-line

Private var

Private Count

L2

L1

P0

P1

Thread modifying

private_count[0]

Thread modifying

private_count[1]
Force Into Different Lines

- Padding the private variables forces them into separate cache lines and removes false sharing

```c
struct padded_int
{
    int value;
    char padding[128];
} private_count[MaxThreads];
```
Force Into Different Lines

- Padding the private variables forces them into separate cache lines and removes false sharing

```c
struct padded_int {
    int value;
    char padding[128];
} private_count[MaxThreads];
```
Success!!

Two processors are almost twice as fast

<table>
<thead>
<tr>
<th>Performance</th>
<th>serial</th>
<th>Try 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>t=1</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>t=2</td>
<td>0.51</td>
<td></td>
</tr>
</tbody>
</table>

Is this the best solution???
Count 3s Summary

Recapping the experience of writing the program, we

- Wrote the obvious “break into blocks” program
- We needed to protect the count variable
- We got the right answer, but the program was slower … lock congestion
- Privatized memory and 1-process was fast enough, 2-processes slow … false sharing
- Separated private variables to own cache line

Finally, success
Break

- During break think about how to generalize the “sum n-integers” computation for n>8, and possibly, more processors
Variations

- What happens when more processors are available?
  - 4 processors
  - 8 processors
  - 256 processors
  - 32,768 processors
Our Goals In Parallel Programming

- Goal: Scalable programs with performance and portability
  - Scalable: More processors can be “usefully” added to solve the problem faster
  - Performance: Programs run as fast as those produced by experienced parallel programmers for the specific machine
  - Portability: The solutions run well on all parallel platforms
Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class when $n > P = 8$
- Use a logical binary tree?
Return to problem of writing a parallel sum
Sketch solution in class when $n > P = 8$
Assume communication time = 30 ticks
$n = 1024$
compute performance
Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class when $n > P = 8$
- and communication time = 30 ticks
- $n = 1024$
- compute performance
- Now scale to 64 processors
Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class when \( n > P = 8 \)
- and communication time = 30 ticks
- \( n = 1024 \)
- compute performance
- Now scale to 64 processors

This analysis will become standard, intuitive
Matrix Product: || Poster Algorithm

- Matrix multiplication is most studied parallel algorithm (analogous to sequential sorting)

- Many solutions known
  - Illustrate a variety of complications
  - Demonstrate great solutions

- Our goal: explore variety of issues
  - Amount of concurrency
  - Data placement
  - Granularity

Exceptional by requiring $O(n^3)$ ops on $O(n^2)$ data
Recall the computation...

- Matrix multiplication of (square n x n) matrices $A$ and $B$ producing $n \times n$ result $C$ where $C_{rs} = \sum_{1 \leq k \leq n} A_{rk} \times B_{ks}$
The multiplications are independent (do in any order) and the adds can be done in a tree. O(n) processors for each result element implies O(n^3) total. Time: O(log n)
$O(\log n)$ MM in the real world …

Good properties
- Extremely parallel … shows limit of concurrency
- Very fast -- $\log_2 n$ is a good bound … faster?

Bad properties
- Ignores memory structure and reference collisions
- Ignores data motion and communication costs
- Under-uses processors -- half of the processors do only 1 operation
Where is the data?

- Data references collisions and communication costs are important to final result … need a model … can generalize the standard RAM to get PRAM

![Diagram showing memory access patterns]

- Memory access patterns:
  - 
P_0
  - 
P_1
  - 
P_2
  - 
P_3
  - 
P_4
  - 
P_5
  - 
P_6
  - 
P_7

- Memory access:
  - C
  - A
  - B
Parallel Random Access Machine

- Any number of processors, including $n^c$
- Any processor can reference any memory in “unit time”
- Resolve Memory Collisions
  - Read Collisions -- simultaneous reads to location are OK
  - Write Collisions -- simultaneous writes to loc need a rule:
    - Allowed, but must all write the same value
    - Allowed, but value from highest indexed processor wins
    - Allowed, but a random value wins
    - Prohibited

Caution: The PRAM is *not* a model we advocate
PRAM says $O(\log n)$ MM is good

- PRAM allows any # processors $\Rightarrow O(n^3)$ OK
- $A$ and $B$ matrices are read simultaneously, but that’s OK
- $C$ is written simultaneously, but no location is written by more than 1 processor $\Rightarrow$ OK

PRAM model implies $O(\log n)$ algorithm is best … but in real world, we suspect not

We return to this point later
Where else could data be?

- Local memories of separate processors ...

- Each processor could compute block of \( C \)
  - Avoid keeping multiple copies of \( A \) and \( B \)

**Architecture common for servers**
Data Motion

- Getting rows and columns to processors

- Allocate matrices in blocks
- Ship only portion being used
Blocking Improves Locality

- Compute a $b \times b$ block of the result

- Advantages
  - Reuse of rows, columns = caching effect
  - Larger blocks of local computation = hi locality
Caching in Parallel Computers

- Blocking = caching … why not automatic?
  - Blocking improves locality, but it is generally a manual optimization in sequential computation
  - Caching exploits two forms of locality
    - Temporal locality -- refs clustered in time
    - Spatial locality -- refs clustered by address

- *When multiple threads touch the data, global reference sequence may not exhibit clustering features typical of one thread -- thrashing*
Sweeter Blocking

- It’s possible to do even better blocking …

- Completely use the cached values before reloading
Best MM Algorithm?

- We haven’t decided on a good MM solution
- A variety of factors have emerged
  - A processor’s connection to memory, unknown
  - Number of processors available, unknown
  - Locality--always important in computing--
    - Using caching is complicated by multiple threads
    - Contrary to high levels of parallelism
- Conclusion: Need a better understanding of the constraints of parallelism

Next week, architectural details + model of ||ism
Assignment for Next Time

☐ Reproduce the parallel prefix tree labeling to compute the bit-wise & scan

☐ Try the “count 3s” computation on your multi-core computer

- Implementation Discussion Board … please contribute – success, failure, kibitzing, …

- https://catalysttools.washington.edu/gopost/board/snyder/16265/