CSE524 Parallel Computation

Lawrence Snyder

www.cs.washington.edu/CSEp524

27 March 2007

Course Logistics

- Teaching Assistant: Nathan Kuchta
- Text at Prof. Copy & Print, 4200, “The Ave”
- Class web page is our headquarters
- Take lecture notes -- the slides will be online sometime after lecture
- Occasional homework problems, mostly programming assignments
- Modest Project during last 2-3 weeks

Please ask questions when they arise
Text: *Principles of Parallel Programming*

- Why use it? For this class it’s better than any book published
- It is a work in progress -- please be patient
- It has benefited from one pass by another class
- You can have a huge impact by commenting on:
  - Mention all organizational issues, confusions, poor explanations, technical errors, e.g. programming errors, etc.
  - Editors will scrub the text: Ignore spelling errors, grammar errors, punctuation errors, etc.
  - Use anonymous email for “private comments”

You may learn a lot about book writing!

---

**Why Study Parallelism?**

- After all, for most of our daily computer uses, sequential processors are plenty fast
  - It is a fundamental departure from the “normal” computer model, therefore inherently cool
  - The extra power from parallel computers is enabling in science, engineering, business, …
Topic Overview

- 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

Familiar Parallel Techniques

- SETI and BOINC techniques -- zillions of independent problem instances
- Pipelining -- perform multiple instances of a multi-step operation
- Overlapping computation and communication -- OS task switching
- Carry Look-ahead Adders -- logarithmic circuit to compute carries
Non-Parallel Techniques

- Distributed computing, e.g. client/server structure, is not usually parallel
- Divide-and-conquer is usually limited by “sending and receiving” the data instances
- Techniques that assume \( n^c \) processors for \( n \) size problem
- Almost all techniques that focus on reducing operation counts and building complex data structures

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>
Changing Paradigms

- Sequential summation to tree-based summation
  - Same number of operations; different order

Parallel Prefix

- Sweep up, sweep down
Fundamental Tool of || Pgmming

- Original research on parallel prefix algorithm published by
  R. E. Ladner and M. J. Fischer
  Parallel Prefix Computation
  *Journal of the ACM 27(4):831-838, 1980*

The Ladner-Fischer algorithm requires twice as much time as simple tournament global sum

Applyes to a wide class of operations

---

Write A Small Parallel Program

- Need to know something about machine … use multicore architecture

```
RAM
  Memory
      L2
          L1
                L1
            P0
                P1
```

Count 3s Problem

- Ideal solution …
  - Sequential program

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

- + compiler magic

Compilers “Can’t” Convert to ||ism

- Effort to compile to ||ism began in 1970s
- Research has occupied some of the best compiler writers
- Progress has been achieved
- Success so far limited to simple, clean cases

**Intuition for low expectations**: Fundamentally, compilers apply correctness preserving transformations … but generally, a parallel solution requires a paradigm shift from the sequential approach.
Try 1: Divide Into Separate Parts

- Threading solution

  \[ \text{length} = 16 \quad \text{t} = 4 \]

  \[
  \begin{array}{cccccccccccc}
  2 & 3 & 0 & 2 & 3 & 3 & 1 & 0 & 1 & 3 & 2 & 2 & 3 & 1 & 0 \\
  \hline
  \text{Thread 0} & \text{Thread 1} & \text{Thread 2} & \text{Thread 3}
  \end{array}
  \]

  ```
  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;
  }
  ```

Try 1: Races

- Two processes interfere on memory writes
Try 2: 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);
    }
}
```

Try 2: Correct Program Runs Slow
- Serializing at the mutex
  - The processors wait on each other

<table>
<thead>
<tr>
<th>Performance</th>
<th>t=1</th>
<th>t=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>Try 2</td>
<td>5.02</td>
<td>6.81</td>
</tr>
</tbody>
</table>
Closer Look

- 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);
    }
}
```

---

Try 3: 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

Try 3: False Sharing

- Private var ≠ private cache-line

Try 3

<table>
<thead>
<tr>
<th>Performance</th>
<th>serial</th>
<th>t=1</th>
<th>t=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.91</td>
<td></td>
<td>0.91</td>
<td>1.15</td>
</tr>
</tbody>
</table>
Try 4: 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

Performance

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

Is this the best solution???
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 programs run well on all parallel platforms
Challenges of Parallel 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

Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class
Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class
- when n > P

Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class
- when n > P
- and communication time = 30 ticks
Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class
- when $n > P$
- and communication time = 30 ticks
- $n = 1024$, $P = 8$
- compute performance

---

Program A Parallel Sum

- Return to problem of writing a parallel sum
- Sketch solution in class
- when $n > P$
- 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 $n \times n$) matrices $A$ and $B$ producing $n \times n$ result $C$
  where $C_{rs} = \sum_{j=1}^{n} A_{rj} \cdot B_{js}$

$C = A \cdot B$

$= \sum_{j=1}^{n} A_{rj} \cdot B_{js}$
Extreme Matrix Multiplication

- The multiplications are independent (do in any order) and the adds can be done in a tree

\[ O(n) \text{ processors for each result element implies } O(n^2) \text{ total} \]

Time: \( O(\log n) \)

\[ O(\log n) \text{ MM in the real world ...} \]

**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

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 model we advocate
PRAM says $O(\log n)$ MM is good

- PRAM allows any # processors $\Rightarrow O(n^2)$ 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 next week

---

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 = high 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 limited to 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

---

Next Week ...

- Read Chapter 2 and study the CTA
  - You may need to answer questions on it …