Challenges and Opportunities of Parallelism

Lawrence Snyder

www.cs.washington.edu/homes/snyder

8 April 2009
The Situation Today …

- Consumer and business software cannot achieve further performance improvements without parallelism.
- Is that a problem … or an opportunity?
- My goal today is not to “sell” new and wondrous results, but to give a global perspective on the parallel “problem” answering:
  - Why is it so hard?
  - Where do we go from here?
Outline Of This Talk

- Small Parallelism: today’s technology facts
- A Closer Look -- Facts of Life
- Large Parallelism: today’s technology facts
- A Closer Look -- Living with Parallelism
- A trivial exercise focuses our attention
- A little “science” can help
Today’s Facts

Single Processor

Opportunity
Moore’s law continues, so use more gates

Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter & Burton Smith
Today’s Facts

☐ Laptops, desktops, servers, etc. now are parallel (multicore) computers … why?

☐ Multi-core gives “more” computation and solves difficult hardware design problems
  - What to do with all of the transistors: Replicate
  - Power Issues
  - Clock speeds

… consider each issue
Multi-core Replicates Designs

- Traditionally we have used transistors to make serial processors more aggressive
  - Deeper pipelining, more look-ahead
  - Out of order execution
  - Branch prediction
  - Large, more sophisticated TLBs, caches, etc.
- Why not continue the aggressive design?
  … Diminishing returns, limit on ILP, few new ideas

Bottom Line: Sequential instruction execution reaching limits
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
## Multi-core Design Space

- Smaller, slower implies more modest design

### Traditional vs. Core Design

<table>
<thead>
<tr>
<th>micro-architecture</th>
<th>out of order</th>
<th>in order</th>
<th>mm^2</th>
<th>W</th>
<th>GHz</th>
<th>relative perf</th>
<th>words wide</th>
<th>Gflop/s</th>
<th>Gflops/mm</th>
<th>Gflops/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>50</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>power</td>
<td>37.5</td>
<td>6.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frequency</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>threads</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>single thread</td>
<td>1</td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td>relative perf</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector operations</td>
<td>4</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>words wide</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peak throughput</td>
<td>32</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td>Gflop/s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>area capacity</td>
<td>0.6</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>Gflops/mm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>power capacity</td>
<td>0.9</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>Gflops/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: Doug Carmean, Intel
Multi-cores Have Their Problems

☐ Single threaded computation doesn’t run faster (it may run slower than a 1 processor per chip design)

☐ Few users benefit from m-c ||ism today

- Existing software is single threaded
- Compilers don’t help and often harm parallelism
- It’s often claimed that OS task scheduling is one easy way to keep processors busy, but there are problems
  ☐ limited numbers of tasks available
  ☐ contention for resources, e.g. I/O bandwidth
  ☐ co-scheduled tasks often have dependences -- no advantage to or prevented from running together
Legacy Code

☐ Even casual users use applications with a total code base in the 100s of millions LOC … and it’s not parallel

☐ There are not enough programmers to rewrite this code, even if it had || potential

☐ With single processor speeds flat or falling, this code won’t improve

Challenge

How to bring performance to existing programs
What Can Microsoft Do?

“Parallelism requires adjustments at every level of the stack … the repartitioning of different tasks to different layers … So look for a rebalancing of roles and runtimes. We need to formalize that in the operating system. Expect the first pieces in the next generation of Windows.” Craig Mundy, Microsoft Chief Research & Strategy Officer, 3 October 2008
Legacy Code

- Even casual users use applications with a total code base in the 100s of millions LOC … and it’s not parallel
- There are not enough programmers to rewrite this code, even if it had || potential
- With single processor speeds flat or falling, this code won’t improve

**Challenge**
How to bring performance to existing programs

**Opportunity**
Much legacy code supports backward compatibility -- ignore
Potential of Many Threads - Amdahl

- Maximum performance improvement by parallelism is $S$-fold if sequential part is $1/S$

\[ T_P = \frac{1}{S} \cdot T_S + \left(1 - \frac{1}{S}\right) \cdot \frac{T_S}{P} \]

- “Everyone is taught Amdahl’s Law in school, but they quickly forget it” -- T. R. Puzak, IBM

- More complex in multi-core case: programs that are 99% parallel get speedup of 72 on 256 cores [Hill & Marty]
Summary So Far

- **Opportunities --**
  - Moore’s law continues to give us gates
  - Multi-core is easy design via replication
  - Replicating small, slower processors fixes power problems & improves ops/second

- **Challenges --**
  - Smaller, slower designs are smaller, slower on 1 thread
  - Huge legacy code base not improved
  - Parallelism doesn’t speed-up sequential code. Period.
Outline Of This Talk

- Small Parallelism: today’s technology facts
- A Closer Look -- Facts of Life
- Large Parallelism: today’s technology facts
- A Closer Look -- Living with Parallelism
- A trivial exercise focuses our attention
- A little “science” can help
But There’s More To Consider

- Two processors may be ‘close’ on the silicon, but sharing information is still expensive
- $r_1 \rightarrow L_1 \rightarrow L_2 \rightarrow$ coherency protocol $\rightarrow L_2 \rightarrow L_1 \rightarrow r_2$
- Opteron Dual Core: more than 100 instruction times
- **Challenge**
  
  On Chip Latency

- Latency between cores only gets worse
Latency -- Time To Access Data

- Latencies (relative to instruction execution) have grown steadily over the years, but (transparent) caching has saved us

- No more

  - Interference of multiple threads reduces the benefits of spatial and temporal locality
  - Cache coherence mechanisms are
    - good for small bus-based SMPs
    - slower and complex as size, distance and decentralization increase

- Thus, latency growth is a problem
Bandwidth On/Off Chip

- Many applications that are time-consuming are also data intensive e.g. MPEG compress
- $C$ cores share the bandwidth to memory: $\frac{available\_bandwidth}{C}$
- Caching traditionally solves BW problems, but Si devoted to cache reduces number of cores
- A problem best solved with better technology
Memory Model Issues

- When we program, we usually think of a single memory image with “one” history of transitions: $s_0, s_1, \ldots, s_k, \ldots$
- Not true in the parallel context
  - Deep technical problem
  - Two cases
    - Shared memory
    - Distributed memory
- The consequences of this fact are the largest challenges facing parallel programming
Facts of Life Summary: Challenges

- Latency on chip will increase as core count increases
  - significant already
  - both logic complexity and “electronic” distance
- Bandwidth to L3 and memory shared
- Memory model for programmers
  - Presently broken
  - Extensive research has failed to find alternate
  - May be a “show stopper”
Outline Of This Talk

- Small Parallelism: today’s technology facts
- A Closer Look -- Facts of Life
- Large Parallelism: today’s technology facts
- A Closer Look -- Living with Parallelism
- A trivial exercise focuses our attention
- A little “science” can help
Large Parallel Facts

- A parallel computer (IBM Roadrunner) has achieved 1 Peta Flop/S ($1.1 \times 10^{15}$ Flop/S)
Large Parallel Facts

☐ A parallel computer (IBM Roadrunner) has achieved 1 Peta Flop/S (1.1×10^{15} Flop/S)
Roadrunner Specs

- 12,562 dual-core AMD Opteron® chips and 12,240 STI PowerXCell 8i chips (25.6 GF SP, 12.8 GF DP per SPE) … each AMD core gets a Cell chip
- 98 terabytes of memory
- 3.8 MW total delivering 376 MFlops/W
- 278 refrigerator-sized racks; 5,200 ft^2 footprint

Other Data:
- 10,000 connections – both Infiniband and Gigabit Ethernet
- 55 miles of fiber optic cable
- 500,000 lbs.
- Shipped to Los Alamos in 21 tractor trailer trucks
Performance: Top 500

Source: Top 500 Supercomputers
Performance: Top 500

We are on track for exa-scale computing

Source: Top 500 Supercomputers
Performance: From 1st To Last

6-8 Years
Earth Simulator

- Started at #1 June 2002, 35.8 TF
- Decommissioned Sept 2008; still at #73 #500=12.6TF
Speed Increase: An Opportunity

- 1000 Yrs: 1 Gflop/s
- 1 Year: 1 Tflop/s
- ~8 Hrs: 1 Pflop/s
- ~1 Sec: 1 Eflop/s
Node Structure of RoadRunner
Hybrid Machine

3 ISAs
- Opteron
- PPC
- SPE
Three Programming Levels: Challenge

- Programming Multiple Levels is not simply just 3 compilations
  - SPE is difficult to program -- few HW goodies
  - PPC is mostly orchestrating local data flow
  - Opteron (contributes 3% of performance) mostly orchestrates more global data flow
  - Library support different for each ISA

- LINPAC has very favorable work/word character

- LINPAC benchmark has been developed over many years; writing new HPC programs for this architecture will be time consuming
Interprocessor Communication

- Communication between two arbitrary processors (latency) is a serious problem
- (Not considered today, but in Lecture1)

<table>
<thead>
<tr>
<th>System</th>
<th>Processor Model</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>AMD</td>
<td>100</td>
</tr>
<tr>
<td>SMP</td>
<td>Sun Fire E25K</td>
<td>400-660</td>
</tr>
<tr>
<td>Cluster</td>
<td>Itanium + Myrinet</td>
<td>4100-5100</td>
</tr>
<tr>
<td>Super</td>
<td>BlueGene/L</td>
<td>5000</td>
</tr>
</tbody>
</table>

Lge range => cannot be ignored
Summary So Far, Large Scale

- **Opportunities**
  - Moore’s law continues
  - Large Scale on track for exa-scale machines by about 2019, though there is much to do
  - The advancement will be significant
    \[1\text{Kyears} \rightarrow \text{seconds}\]

- **Challenges**
  - Hybrid design requires difficult multilevel programming
  - Hardware “lifetimes” are short -- be general
  - Latencies will continue to grow
Outline Of This Talk

- Small Parallelism: today’s technology facts
- A Closer Look -- Facts of Life
- Large Parallelism: today’s technology facts
- A Closer Look -- Living with Parallelism
- A trivial exercise focuses our attention
- A little “science” can help
Life Times / Development Times

- Hardware -- has a “half-life” measured in years
- Software -- has a “half-life” measured in decades
- Exploiting specific parallel hardware features risks introducing architecture dependences the next platform cannot fulfill
- In parallel programming architecture “shows through” … don’t make the view too clear
What makes good sequential and parallel algorithms are different

- Resources
  - S: Reduce instruction count
  - P: Reduce communication

- Best practices
  - S: Manipulate references, exploit indirection
  - P: Reduce dependences, avoid interaction

- Look for algorithms with
  - S: Efficient data structures
  - P: Locality, locality, locality
Good Algorithms Not Well Known

- What is the best way to multiply two dense matrices in parallel?  
  **Ans: In Lecture 1**

- We all know good serial algorithms and sequential programming techniques

- Parallel techniques not widely known, and because they are different from sequential techniques, should we be teaching them to Freshmen in college?
Outline Of This Talk

- Small Parallelism: today’s technology facts
- A Closer Look -- Facts of Life
- Large Parallelism: today’s technology facts
- A Closer Look -- Living with Parallelism
- A trivial exercise focuses our attention
- A little “science” can help
Programmer Challenge: Education

- Most college CS graduates have
  - No experience writing parallel programs
  - No knowledge of the issues, beyond concurrency from OS classes
  - Little concept of standard parallel algorithms
  - No model for what makes a parallel algorithm good

- Where do the programmers come from?
Illustrating the Issues

Consider the trivial problem of counting the number of 3s in a vector of values on m-c
Try # 1

Assume array in shared memory; 8 way ||-ism

```c
void count3s() {
    int count=0; int i=0; /* Create t threads */
    for (i=0;i<t;i++){
        thread_create(count3s_thread,i);
    }
    return count;
}

void count3s_thread(int id){
    int j;
    int length_per_thread=length/t;
    int start=id*length_per_thread;
    for (j=start; j<start+length_per_thread;j++) {
        if (array[j]==3)
            count++;
    }
}
```

```
array: [2 3 0 2 3 3 1 0 0 1 3 2 2 3 1 0 ...]
```

Thread 0: [2 3 0 2 3 3 1 0 0 1 3 2 2 3 1]
Thread 1: [3]
Thread 2: [2 3 1 0]
Thread 3: [0]
Try #1 Assessment

☐ Try #1 doesn’t even get the right answer!
☐ The count variable is not protected, so there is a race as threads try to increment it

Thread i
...  
 lw   $8,count-off(gp)  
 addi $8,1  
 sw   $8,count-off(gp)  
...
Try #2

- Protect the shared count with a mutex lock

```c
void count3s_thread(int id){
    int j;
    int length_per_thread=length/t;
    int start=id*length_per_thread;
    for (j=start; j<start+length_per_thread; j++) {
        if (array[j]==3) {
            mutex_lock(m);
            count++;
            mutex_unlock(m);
        }
    }
}
```

- This solution at least gets the right answer
Try #2 Assessment

- It doesn’t perform, however
Try #3

☐ Include a private variable

```c
void count3s_thread(int id){
    int j;
    int length_per_thread=length/t;
    int start=id*length_per_thread;
    for (j=start; j<start+length_per_thread; j++) {
        if (array[j]==3) {
            private_count[id]++;
        }
    }
    mutex_lock(m);
    count += private_count[id];
    mutex_unlock(m);
}
```

☐ Contention, if it happens is limited
Try #3 Assessment

- The performance got better, but still no ||-ism
Try #3 False Sharing

- The private variables were allocated to one or two cache lines

- Cache coherency schemes, which keep everyone’s cache current operate on the cache-line granularity … still contention
  - Suppose two processors have the cache line
  - When one writes, other is invalidated; refetch
Try #4

- By simply adding padding to give each private count its own cache line Try #3 works

```c
struct padded_int {
  int val;
  char padding [60];
} private_count[t];
```

- Notice that false sharing is a sensitivity hardware dependent on f, the cache line size
- Machine features “show through”
Try #4 Assessment

- Finally, speed-up over serial computation

<table>
<thead>
<tr>
<th>Processors</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 processor</td>
<td>0.91</td>
</tr>
<tr>
<td>2 processors</td>
<td>1.6</td>
</tr>
<tr>
<td>4 processors</td>
<td>3.1</td>
</tr>
<tr>
<td>8 processors</td>
<td>3.2</td>
</tr>
</tbody>
</table>

- It wasn’t so easy, and it wasn’t so great
Programming Challenges

- Today’s programmers not ||-programmers
- Much to worry about
  - Standard abstractions (locks, etc.) too low level
  - Memory Model (mentioned before) busted
  - Parallelism “shows through”
  - Hardware sensitivity (false sharing)
  - Heavy intellectual investment
    - Small task took serious effort
    - Modest performance achieved
    - Not yet a general solution

Houston, We Have Problem
Outline Of This Talk

- Small Parallelism: today’s technology facts
- A Closer Look -- Facts of Life
- Large Parallelism: today’s technology facts
- A Closer Look -- Living with Parallelism
- A trivial exercise focuses our attention
- A little “science” can help
Overcoming Sequential Control

- Many computations on a data sequence seem to be “essentially sequential”

- Prefix sum is an example: for \( n \) inputs, the \( i^{th} \) output is the sum of the first \( i \) items

  - Input: 2 1 5 3 7
  - Output: 2 3 8 11 18

- Given \( x_1, x_2, \ldots, x_n \) find \( y_1, y_2, \ldots, y_n \) s.t.

  \[
y_i = \sum_{j \leq i} x_j
\]
Naïve Use of Parallelism

- For any $y_i$, a height $\log i$ tree finds the prefix: find it, add $x_i$
  - Much redundant computation
  - Requires $O(n^2)$ parallelism for $n$ prefixes
- Look closer at meaning of tree’s intermediate sums
Naïve Use of Parallelism

- For any $y_i$, a height $\log i$ tree finds the prefix: find it, add $x_i$
  - Much redundant computation
  - Requires $O(n^2)$ parallelism for $n$ prefixes
- Look closer at meaning of tree’s intermediate sums

![Diagram of a tree with nodes and values to illustrate the concept.](image)
Naïve Use of Parallelism

- For any $y_i$, a height $\log i$ tree finds the prefix: find it, add $x_i$
  - Much redundant computation
  - Requires $O(n^2)$ parallelism for $n$ prefixes
- Look closer at meaning of tree’s intermediate sums

\[ \begin{array}{c}
\text{root summarizes its leaves} \\
\end{array} \]
Speeding Up Prefix Calculations

- Putting the observations together
  - One pass over the data computes global sum
  - Intermediate values are saved
  - A second pass over data uses intermediate sums to compute prefixes
  - Each pass will be logarithmic for $n = P$
  - Solution is called: The parallel prefix algorithm

R. E. Ladner and M. J. Fischer
Parallel Prefix Computation
Parallel Prefix Algorithm

Compute sum going up
Parallel Prefix Algorithm

- Compute sum going up
- Figure prefixes going down

Introduce a virtual parent, the sum of values to tree’s left: 0
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is sum of elements to left of subtree
Parallel Prefix Algorithm

Each prefix is computed in $2\log n$ time, if $P = n$
Available || Prefix Operators

- Most languages have reduce and scan (|| prefix) built-in for: +, *, min, max, &&, ||
- A few languages allow users to define || prefix operations themselves … do they?
- Parallel prefix is MUCH more useful
Available || Prefix Operators

- Most languages have reduce and scan (|| prefix) built-in for: +, *, min, max, &&, ||
- A few languages allow users to define || prefix operations themselves … do they?
- Parallel prefix is MUCH more useful

- Length of Longest Run of x
- Number of Occurrences of x
- Histogram
- Mode and Average
- Count Words
- Length of Longest Increasing Run
- Binary String Space Compression
- Run Length Encoding
- Balanced Parentheses
- Skyline

**Why is there no standard programming abstraction?**
Conclusion +

- Most computers sold today are \(||\), and nearly all are under-utilized
  - No silver bullet will save us (Lecture 1)
  - Languages, tools, libraries are not ready

- New software is needed to exploit \(||)ism
  - Naïve parallel programming is difficult, subtle
  - Higher abstractions help (Lecture 1 & 2)

- The “payoff” is to keep riding performance wave
Homework

**HW:** Use ||-prefix to test for balanced parentheses: (((()))())