CSE 262
Spring 2007
Scott B. Baden

Lecture 2
Introduction
Announcements

• Lecture schedule for next 2 weeks will be posted by Friday
A theoretical basis: the PRAM

- **Parallel Random Access Machine**
- Idealized parallel computer
  - Unbounded number of processors
  - Shared memory of unbounded size
  - Constant access time
- Access time is comparable to that of a machine instruction
- All processors execute in lock step
- Combining policies for handling writes to the same location
Why is the PRAM interesting?

- Inspires real world systems and algorithms
- Formal basis for expressing fundamental limitations
  - If a PRAM algorithm is inefficient, then so is any parallel algorithm
  - If a PRAM algorithm is efficient, does it follow that any parallel algorithm is efficient?
A natural programming model for a PRAM: the data parallel model

- Apply an operation uniformly over all processors in a single step
- Assign each array element to a virtual processor
- Implicit barrier synchronization between each step

\[
\begin{array}{c}
2 \\
8 \\
18 \\
12 \\
\end{array} = \begin{array}{c}
1 \\
-2 \\
7 \\
10 \\
\end{array} + \begin{array}{c}
1 \\
10 \\
11 \\
2 \\
\end{array}
\]

\[\text{forall } i = 0:n-1 \quad z[i] = x[i] + y[i]\]
Rank sort on a PRAM

- Compute the rank for all possible pairings of inputs in parallel, \( n^2 \)-fold parallelism
- Move each value in position according to the rank: \( n \)-fold parallelism

\[
\text{forall } i=0:n-1, j=0:n-1 \quad \text{if } ( x[i] > x[j] ) \text{ then } \text{rank}[i] = 1 \text{ end if}
\]

\[
\text{forall } i=0:n-1 \quad \text{y}[\text{rank}[i]] = x[i]
\]
Compute Ranks

forall \ i=0:n-1, \ j=0:n-1
if ( x[i] > x[j] ) then rank[i] = 1 end if

O(N²)
parallelism

Update on write: summation

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>-1</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1
5
2
0
3
4

4/5/07 Scott B. Baden/CSE 262/Spring 2007
Route the data using the ranks

\textbf{forall} \ i=0:n-1 \ y[\text{rank}[i]] = x[i]
Parallel Speedup and Efficiency

• How much of an improvement did our parallel algorithm obtain over the serial algorithm?

• Define the parallel speedup, $S_p$

$$S_p = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on P processors}}$$

• $T_1$ is defined as the running time of the “best serial algorithm”

• The speedup is $(n \lg n) / O(1) = O(n \lg n)$

• No matter how many processors we have, the speedup for this workload is limited by the amount of available work

• **Definition:** Parallel efficiency $E_p = S_p/P$
Enter real world constraints

• The PRAM provides a necessary condition for an efficient algorithm on physical hardware

• But the condition is not sufficient; e.g. rank sort

\[
\text{forall } (i=0:n-1, j=0:n-1) \\
\text{if } (x[i] > x[j]) \text{ then } \text{rank}[i] = 1 \text{ end if} \\
\text{forall } (i=0:n-1) \ y[\text{rank}[i]] = x[i]
\]

• Not all computations can execute efficiently in lock-step

• Real world computers have finite resources: memory and communication network capacity

• Fast networks are expensive
SPMD execution model

• Most parallel programming is implemented under the **Same Program Multiple Data** programming model = SPMD
• Other names for this model are “loosely synchronous” or “bulk synchronous”
• Programs execute as a set of P processes (sometimes threads)
  – We specify P when we run the program
  – Each process is usually assigned to a different physical processor
• Each process
  – is initialized with the same code
  – has an associated *rank*, a unique integer in the range 0:P-1
  – executes instructions at its own rate
• Processes communicate via messages or shared memory
• The sequence of instructions each process executes depends on the process’ rank and the outcome of communication
Message Passing

• Message based communication requires that sender and receiver be aware of one another
• There must be an explicit recipient of the message
• Message passing performs two events
  ‣ Memory to memory block copy
  ‣ Synchronization signal on receiving end: “Data arrived”
Send and Recv

- When `Send( )` returns, the message is “in transit”
  - A return doesn’t tell us if the message has been received
  - Somewhere in the system
  - Safe to overwrite the buffer
- `Receive( )` blocks until the message has been received
  - Safe to use the data in the buffer
- Error if the source and destination object don’t have *identical* types

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(x,1)</td>
<td>Send(x,0)</td>
</tr>
<tr>
<td>Recv(y)</td>
<td>Recv(y)</td>
</tr>
<tr>
<td>Print x, y</td>
<td>Print x, y</td>
</tr>
</tbody>
</table>
In order delivery

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- Receipt order undefined when there are multiple senders.
- In order delivery has associated overhead costs.
Non-blocking communication

- *Blocking* calls that cause the program to wait for completion
- There is also asynchronous, *non-blocking* communication
Non-blocking communication

• *Blocking* calls: wait for completion
• Asynchronous, *non-blocking* communication: return doesn’t indicate completion
• Communication is *split-phased*
  – Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call
    \[ \text{iRecv( ), iSend( )} \]
  – Phase 2: synchronize
    \[ \text{Wait( )} \]
  – We can carry out unrelated computations between the two phases
Communication performance

• Let the message have a length $n$
• The simplest communication cost model is transfer time $= \alpha + \beta^{-\infty} n$
  $\alpha = \text{message startup time}$
  $\beta_{\infty} = \text{peak bandwidth (bytes per second)}$
  $n = \text{message length}$
• Short messages
  $T \sim \alpha \gg \beta^{-\infty} n$
• “Long” messages: “Bandwidth limited”
  $T \sim \beta^{-\infty} n \gg \alpha$
Communication Bandwidth on Blue Horizon

\[ N^{1/2} \approx 100\text{KB} \]

\[ 390 \text{ MB/sec} \]

\[ N = 4\text{MB} \]
More about modeling

• LogP model (Culler et al, 1993), is more precise, but the $\alpha$, $\beta$ model is often good enough

• All these models ignore important effects: switch and processor contention
Motivating application

- Poisson’s equation in 3 dimensions
  \[ \Delta \varphi = \rho(x,y,z) \]
  for \((i,j,k)\) in \(1:N \times 1:N \times 1:N\)
  \[
  u[i][j][k] = \frac{(u[i-1][j][k] + u[i+1][j][k] + u[i][j-1][k] + u[i][j+1][k] + u[i][j][k+1] + u[i][j][k-1])}{6.0}
  \]
  \(\rho \neq 0\)
Domain Decomposition Strategy

- Divide problem into subdomains
- Solve each subdomain locally
- Satisfy loop carried dependences: communicate internal boundary data or synchronize at a local barrier
Classic message passing implementation

- Decompose domain into subregions, one per process
  - Transmit *halo regions* between processes
  - Compute *inner region* after communication completes
- Loop carried dependences impose a strict ordering on communication and computation
Surface to volume ratio

As volume increases, S/V drops

1 unit of work
4 units of communication

16 units of work
16 units of communication
The curse of dimensionality

• As we move to higher dimensional spaces, communication becomes relatively more costly
  ▶ In 2D: \( \frac{4N}{N^2} = \frac{4}{N} \)
  ▶ In 3D: \( \frac{6N^2}{N^3} = \frac{6}{N} \)
What’s difficult about masking data transfer delays?

- The required knowledge is beyond the ability of many application programmers
- Complicated flow control
  - Split phase algorithms
  - Uncertain data arrival times
  - Scheduling
- Implementation policies entangled with user code
- Non-robust performance
Latency tolerant variant

- Only a subset of the domain exhibits loop carried dependences with respect to the halo
- Subdivide the domain to remove some of the dependences
- It is now safe to sweep the inner region in parallel with communication
- Sweep the annulus when communication finishes
Shared memory architecture

- Every processor has direct access to all of memory
- The address space is global to all processors
- Hardware automatically performs the global to local mapping using virtual to physical address translation
- Symmetric multiprocessors: uniform memory access times
- Use a snooping strategy to manage cache coherence
NUMA Architectures

- Unlike UMAs, do not rely on a broadcasting
- Point-to-point messages manage coherence
- A directory keeps track of sharers, one for each block of memory
- Stanford Dash; SGI Origin 2000
Shared memory programming

• Shared memory introduces a new set of programming issues
  – Synchronization
  – Cache management
• Certain applications can be automatically parallelized with a compiler
• Others cannot
• The most primitive programming model: threads
• Higher level models: openMP, PGAS, …
Threads programming model

• Fork/join parallelism managed by a scheduler
• A collection of concurrent instruction streams, called *threads*, that share memory
• We get a new kind of storage class: shared data
• A thread is similar to a procedure call with notable differences
  – A procedure call is “synchronous:” a return indicates completion
  – A spawned thread executes asynchronously until it completes
  – Both share global storage with caller
  – Synchronization is needed when updating shared state
Why threads?

- Processes are “heavy weight” objects scheduled by the OS
  - Protected address space, open files, and other state
- A thread, AKA a lightweight process (LWP) is sometimes more appropriate
  - Threads share the address space and open files of the parent, but have their own stack
  - Reduced management overheads
  - Kernel scheduler multiplexes threads
Why threads?

- Processes are “heavy weight” objects scheduled by the OS
  - Protected address space, open files, and other state
- A thread, AKA a lightweight process (LWP) is sometimes more appropriate
  - Threads share the address space and open files of the parent, but have their own stack
  - Reduced management overheads
  - Kernel scheduler multiplexes threads
Threads in practice

• A common interface is the POSIX Threads “standard” (pthreads): IEEE POSIX 1003.1c-1995
  – But there are non-standard features some of which are outside the scope of the standard!
• Another approach is to use program annotations via openMP
Practical issues

- Thread creation costs 10s of µsecs
- Moving data in shared memory is cheaper than passing a message through shared memory

www.llnl.gov/computing/tutorials/pthreads

<table>
<thead>
<tr>
<th></th>
<th>create</th>
<th>MPI Shared Mem BW</th>
<th>Mem to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 2.4 GHz Xeon</td>
<td>34 µs</td>
<td>0.3 GB/s</td>
<td>4.3 GB/s</td>
</tr>
<tr>
<td>Intel 1.4 GHz Itanium 2</td>
<td>42 µs</td>
<td>1.8</td>
<td>6.4</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4</td>
<td>30 µs</td>
<td>2.1</td>
<td>11</td>
</tr>
</tbody>
</table>
Measuring shared memory system performance

• Dot product: 2 mems / 2 flops
• DOT_PROD = SUM(x[:]*y[:])
• On valkyrie, 1M array per processor
  – NT=1: 44.6 MFlops
  – NT=2: 68.0 MFlops
• On a dual 1.8GHz Opteron: 250 and 500 MFlops
• SDSC Datastar, 1.5GHz Power4: 289&586 MFlops
Reducing memory impact

• Insert operations that use registers only

```c
for (int i=0; i < N; i++){
    result += x[i]*y[i];
    for (int j=0; j < K; j++) result *= c;
}
```

• On valkyrie, 1M array per processor

<table>
<thead>
<tr>
<th>NT\K</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.6</td>
<td>121</td>
<td>71</td>
<td>92</td>
<td>113</td>
<td>150</td>
<td>172</td>
<td>165</td>
</tr>
<tr>
<td>2</td>
<td>68.8</td>
<td>157</td>
<td>114</td>
<td>151</td>
<td>198</td>
<td>270</td>
<td>319</td>
<td>322</td>
</tr>
</tbody>
</table>