Lecture 2

Theoretical basis, performance, and an introduction to message passing

Announcements

• Urvashi’s office hours have been posted
• Homework #1 will be assigned today, due next Thursday (4/10) in class
Today’s readings

- Text
  - Chap 2: §2.3, §2.4 (pp 24-32), §2.5 (pp 53-60)
  - Chap 5: §5.1-5.2 (pp 195-200, 202), §5.4 (pp 208-212)
  - Chap 6: §6.1-6.3 (pp 233-240)

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
Why is the PRAM interesting?

- It inspires some research, and some real world designs
- We can use the PRAM to identify some inefficient algorithms
- If a PRAM algorithm is inefficient, then so is any parallel algorithm
- But the opposite is not true!

How do we handle concurrent accesses?

- Our options are to prohibit or permit concurrency in reads and writes
- There are therefore 4 flavors
- We’ll focus on CRCW = Concurrent Read Concurrent Write
- All processors may read or write
CRCW PRAM

- What happens when more than one processor attempts to write to the same location?
- We need a rule for combining multiple writes
  - Common: All processors must write the same value
  - Arbitrary: Only allow 1 arbitrarily chosen processor to write
  - Priority: Assign priorities to the processors, and allow the highest-priority processor's write
  - Combine: the written values in some meaningful way, e.g. sum, max, using an associative operator.

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

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

forall var_0 = <range>, var_1 = <range>, ... <assignment>

Evaluate entire RHS of <assignment> for all index values (in any order) and assign to a temporary

Perform all assignments (in any order) using the temporary

No more than one value for each element on the left hand side

forall i = 0:n-1 x[i] = (i*2.0/n)-1.0
forall i = 0:n-1, j = 0:m-1 H[i,j] = 1.0/(i+j)

---

Sorting on a PRAM

- A 2 step algorithm called rank sort
- Compute the rank (position in sorted order) for each element in parallel
  - Compare all possible pairings of input values in parallel, \( n^2 \)-fold parallelism
  - CRCW model with update on write using summation (+)
- Move each value to its correctly sorted position according to the rank: n-fold parallelism
- O(1) running time
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

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

Rank sort in action

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

```
1  7  3  -1  5  6
```
Compute Ranks

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

Route the data using the ranks

\[ \text{forall } i=0:n-1 \ y[\text{rank}[i]] = x[i] \]
**Speedup**

- How much of an improvement did our parallel algorithm obtain over the serial algorithm?
- Define the *parallel speedup* as the ratio of
  
  \[
  \frac{\text{Running time on 1 processor}}{\text{Running time of the parallel program}}
  \]
  
  – For the running time on 1 processor we use an idealized serial computer called a RAM (hardware assumptions?)
- The speedup is \( \frac{n^2 + n}{2} = O(n) \)

**Limits to performance**

- Some computations either do not parallelize (which might these be?) or they parallelize poorly
- Consider the summation
  
  \[
  \text{sum = 0}
  \]
  
  \[
  \text{forall } i=0:n-1 \quad \text{sum = sum + x[i]}
  \]
- A naïve algorithm runs serially in \( n \) steps
  – Speedup = 1
- A better algorithm does this in \( \log_2(n) \) time
  – Speedup is \( \log_2(n) \)
Enter real world constraints

- A PRAM gives us a necessary condition for an efficient algorithm
- But the condition is not sufficient
  - Real world computers have finite resources including memory and network capacity
  - We cannot ignore communication network capacity, nor the cost of building a contention free network
  - Not all computations can execute efficiently in lock-step
- Interesting things happen when resources are finite!

Control mechanisms

- In addition to address space organization, we also classify architectures according to their control mechanism
- How do the processors issue their instructions?
- Today, most parallel computers execute their instruction streams independently
- Some special purpose machines execute a global instruction stream in lock-step
Flynn’s classification (1966)

SIMD: Single Instruction, Multiple Data

MIMD: Multiple Instruction, Multiple Data

SIMD

- Two notable SIMD designs
  - ILIAC IV (1960s)
  - Connection Machine Model 1 and 2 (1980s)
- These machines excel at operating on regular arrays of data
MIMD

• SIMD machines have a niche market, e.g. signal processing
• Why might have MIMD overtaken SIMD?
• We’ll focus on MIMD architectures in this course, and we’ll program them with message passing

Performance in the real world

• Why measure performance?
• Determine if we have met our design goals
• Obtain feedback to refine a design
• Establish competition and pricing
  – Price/Performance
Measures of Performance

- Completion time for a given workload
- Throughput: amount of work that can be accomplished in a given amount of time
- Relative performance: given a reference architecture or implementation
  AKA Speedup

Parallel speedup and efficiency

- **Definition**
  The parallel speedup on P processors is $S_P$
  \[
  \frac{\text{Execution time on 1 processor}}{\text{Execution time on P processors}} = \frac{T_1}{T_P}
  \]
- Parallel efficiency $E_P$
  \[
  S_P/P
  \]
- $T_1$ is defined as the running time on the “best serial algorithm”
- In general, $T_1$ is not the running time of the parallel algorithm on 1 processor but is a program customized to run on 1 processor
Performance Anomalies

• Important to ensure that you have the best serial algorithm
• Otherwise, you might report a super-linear speedup, that is \( S_P > P \)
• Super-linear speedups are often an artifact of inappropriate measurement technique
• In some cases, when there is a super-linear speedup, a better serial algorithm may be lurking

What’s wrong with speedup?

• Speedup is not always an accurate way to compare different algorithms
• For an individual user the bottom line is running time \( T_P \) or the space time cost \( P T_P \)
• We might be able to obtain a better speedup at the cost of a longer running time
• How might this happen?
• We also have to be careful about comparing speedups across different machines. Why?
Scalability

• Sometimes communication can be a bottleneck that limits performance
• More generally, other factors can limit performance
• We say that a computation is scalable if performance increases as a “nice function” with the number of processors: linear or even $n \log n$

Limits to scalability

• In practice scalability can be hard to achieve
  – “Non-productive” work associated with exploiting parallelism, e.g. communication
  – Serial sections: portions of the code that run on only one processor e.g. (initialization)
  – Load imbalance: work assigned unevenly to processors
• Some algorithms present intrinsic barriers to realizing scalability and in these cases we seek alternatives
Amdahl’s law (1967)

- Sometimes work will not parallelize at all: we call it a **serial section**
- A serial section limits scalability
- Let $T_1 = f \times T_1 + (1-f) \times T_1$
  
  $f$ is the fraction of $T_1$ that runs serially
- $T_P = f \times T_1 + (1-f) \times T_1 / P$
  
  Thus $S_P = 1/[f + (1 - f)/p]$
- As $P \rightarrow \infty$, $S_P \rightarrow 1/f$
- This is known as *Amdahl's Law* (1967)
Scaled Speedup

- Amdahl’s law led many to take a pessimistic outlook on the benefits of parallelism
- Observation: Amdahl’s law assumes that the workload (W) remains fixed
- But parallel computers are used to tackle more ambitious workloads
  - W increases with P
  - f often decreases with W
- Instead of asking what the speedup is, let’s ask how long a parallel program would run on a single processor

Computing scaled speedup

- Let \( T_P = 1 \)
- \( f' \) = fraction of serial time spent on the parallel program
- \( T_1 = f' + (1 - f') \times P \)
- \( T_1 = S'_P = \) scaled speedup
- Scaled speedup is linear in P
Isoefficiency

- Consequence of Gustafson’s observation is that we increase N with P
- Kumar: We can maintain constant efficiency so long as we increase N appropriately
- The isoefficiency function specifies the growth of N in terms of P
- If N is linear in P, we have a scalable computation
- More on this later on

Programming with Message Passing

- We’ll be using the message passing programming model in this course
- A message passing program runs as P processes
- We specify this value when we run the program
- Assume that each process is assigned to a different physical processor
- Each physical process
  - is initialized with the same code
  - has an associated rank, a unique integer in the range 0:P-1
**SPMD programming**

- We call this model “same program multiple data”
- Processes communicate by sending messages and execute instructions at their own rate
- The sequence of instructions each process executes depends on the contents of messages and on the rank
- We sometimes call this model “loosely synchronous” or “bulk synchronous”
- With MPMD ("multiple program multiple data"), each process may run a different program

**MPI**

- We’ll program with a library called MPI
- MPI was designed by committee
- Available in C, C++, Java, Fortran
- See the MPI section of “Software available in the Course”
  
  http://www.cse.ucsd.edu/classes/sp03/cse160/testbeds.html

- Today we’ll work with our more abstract API
The API

• There are $P = \text{nproc}(\ )$ processors
• Every processor has an assigned rank:
  \[ 0 \leq \text{myRank}(\ ) \leq P-1 \]
• Simplest form of communication: point-to-point messages
  – Send a message to another processor
  – Receive a message from another processor

What’s in a message?

• Message passing is like sending email
• To send a message we need
  – A destination
  – A message body (can be empty)
• To receive a message we need similar information, including a receptacle to hold the incoming data
Message Passing

- Recall that message based communication requires that sender and receiver be aware of one another
- There must be an explicit recipient of the message
- In message passing there are two events:
  - Memory to memory block copy
  - Synchronization signal on receiving end: "Data arrived"

The API

- nproc( ) = # processors
- myRank( ) = my process ID
- Send(Object, Destination process ID)
- Receive(Object)
- Object is restricted to types that have a default constructor
- Later we’ll relax this restriction
Some semantic issues

- Receive( ) blocks until the message has been received
  - It is safe to use the data in the buffer

- When Send( ) returns, the message is “in transit”
  - It could be anywhere in the system, and might have been received
  - A return from Send( ) doesn’t tell us if the message has been received
  - It is safe to overwrite the data in the buffer

- It is an error if the source and destination object don’t have compatible types. (What might compatible mean?)

Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent
- But if different processes send messages to the same destination, the order of receipt isn’t defined across processes
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent
- But if different processes send messages to the same destination, the order of receipt isn’t defined across processes
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- But if different processes send messages to the same destination, the order of receipt isn’t defined across processes.

Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- But if different processes send messages to the same destination, the order of receipt isn’t defined across processes.
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- But if different processes send messages to the same destination, the order of receipt isn’t defined across processes.

![Diagram showing causality](image-url)
Some simple programs

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

What is the outcome of this program?
Some simple programs

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

What is the outcome of this program?

Parallel Sorting in practice

- A fundamental algorithm in data processing
- Given an unordered set of keys $x_0, x_1, \ldots, x_{N-1}$
- Return the keys in sorted order
- The keys may be character strings, floating point numbers, integers, or any object for which the relations $>$, $<$, and $==$ hold
- We’ll assume integers here
- Will talk about other algorithms later on
Compare and exchange sorts

- The well known bubble sort algorithm
- Though we’ll look at more efficient sorts, the algorithm is representative of a class of compare and exchange algorithms which can be highly effective
- We can’t execute this formulation of the algorithm concurrently owing to the loop carried dependence in the inner loop

```plaintext
for i = N-1 to 1 by -1 do
    done = TRUE;
    for j = 0 to i-1 do
        if (a[i] < a[j]) { a[i] ↔ a[j]; done=FALSE; }
    end do
    if (done) break;
end do
```

Odd Even Transposition sort

- Divide the values into disjoint sets, odd and even
- Sort each set separately

```plaintext
for i = 0 to N-1 do
    done = OESort(a,0,N-1);
    done &< OESort(a,1,N-1);
    if (done) break;
end do
```

```plaintext
int OESort(key a[ ], int i0, int n)
    done = TRUE;
    for j = i0 to n by 2 do
        if (a[j] > a[j+1]) {
            a[j] ↔ a[j+1];
            done=FALSE;
        }
    end do
```

Odd Even sort in action

0  1  2  3  4  5
1  7  3  -1  5  6

Odd Even sort in action

0  1  2  3  4  5
1  7  -1  3  5  6
Odd Even sort in action

0  1  2  3  4  5
1  7  -1  3  5  6

Odd Even sort in action

0  1  2  3  4  5
1  -1  7  3  5  6
Odd Even sort in action

```
0  1  2  3  4  5
1  -1 7  3  5  6
```

Odd Even sort in action

```
0  1  2  3  4  5
-1  1  7  3  5  6
```
Odd Even sort in action

0 1 2 3 4 5
-1 1 3 5 7 6
Odd Even sort in action

A simple message passing algorithm

• N/P elements per processor
• Perform local sorts using odd/even sort
• After each local sort, exchange edge elements with neighboring processor
• This is a slow algorithm
  – Comparisons involve only nearest neighbors
An improvement: odd-even merge sort

- We can do better if we can look at a larger window of neighbors
- Perform local sorts using a fast serial algorithm, i.e. quicksort
- After each local sort, exchange half the elements with each neighboring processor

Parallel implementation

- Apply a block compare and swap operation
- Processors exchange chunks in odd-even pairs
- Each processor applies a local merge sort extract smallest (largest) N/P values, discards the rest
Odd-even merge sort in action

N values to be sorted
Treat as four lists of M = N/4
Sort each separately
Compare and swap
Compare and swap
Final sorted list

Block compare and swap

- Processor 0: -1 3 7 9 11
- Processor 1: 2 4 8 12 14
- Compare and swap
  - Each processor has sorted merged list
    -1 2 3 4 7 8 9 11 2 14
  - Processor 0 takes 5 smallest values: -1 2 3 4 7
  - Processor 1 takes 5 largest values: 8 9 11 12 14
- Chunking the data allows keys to move a larger distance in each swap operation
Assignment #1

Worth 10 points, due next Thursday in class

1. Go to the Top500 supercomputer site at http://www.top500.org/
   a. What are $R_{\text{max}}$ and $R_{\text{peak}}$ and what is their significance?
   b. Plot $R_{\text{max}}$ and $R_{\text{peak}}$ over the past 10 years for the top 5 machines, noting any trends involving particular manufacturers. Do the same for the number of processors.
   c. Has there been any change in the ratio $R_{\text{peak}} / R_{\text{max}}$ over the past 5 or 10 years (go back as far as you wish, following the ARCHIVE link)?
   d. How about the ratio of $R_{\text{max}}$ to the number of processors?
   e. Follow the trends link on the web pages and note any trends that have taken place over time.
   f. Pick a favorite installation, and discuss an important application that runs on the machine. You may need to look at the installation’s web site

2. Amdahl’s law and speedup
   a. Draw an analogy between Amdahl’s law and some phenomenon you observe in the real world (but not with a computer)
   b. Draw a similar analogy with scaled speedup