Lecture 2

Theoretical basis and performance
Bulk synchronous programming with message passing

Announcements

- Homework #1 has been posted; due next Thursday IN CLASS
- Valkyrie is still not available
- Web board?
Today’s lecture

• Execution models
  – PRAM
  – Hardware control mechanisms
  – SPMD

• Performance
  – Performance metrics
  – Speedup
  – Scaling

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

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

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

\[\forall i = 0:n-1 \quad x[i] = (i \cdot 2.0/n) - 1.0\]

\[\forall i = 0:n-1, j = 0:m-1 \quad H[i,j] = 1.0/(i+j)\]

---

**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

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

Route the data using the ranks

forall i=0:n-1 y[rank[i]] = x[i]
Enter real world constraints

- A PRAM gives us a necessary condition for an efficient algorithm
- But the condition is not sufficient; e.g. rank sort
  
  ```
  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]
  ```
- 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

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

- 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

Typical MIMD computer

- A collection of $P$ processor-memory pairs
- Processors communicate over an interconnect
- There may or may not be a global shared memory

Jim Demmel, U.C. Berkeley CS Division
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 $S_P$

  Execution time on 1 processor = $T_1$
  Execution time on P processors = $T_P$

- Parallel efficiency $E_P$
  
  $E_P = \frac{S_P}{P}$

- $T_1$ 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 \( P \to \infty \), \( S_P \to 1/f \)
- 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 $T_P = 1$
- $f' = \text{fraction of serial time spent on the parallel program}$
- $T_1 = f' + (1 - f') \times P$
- $T_1 = S'_P = \text{scaled speedup}$
- 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 execution model

• Consider the simple SPMD execution model “same program multiple data”
• Programs execute as a set of P processes
  – We specify P 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
• Processes communicate and they execute instructions at their own rate
• In this course, communication will involve passing messages
• 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”
The API

- There are $P = nproc()$ processors
- Every processor has an assigned rank:
  $0 \leq myRank() \leq P-1$
- 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"

Message buffers