Lecture 13: Analytical Modeling of Parallel Programs, part 1

Concurrent and Multicore Programming

Department of Computer Science and Engineering

Yonghong Yan

yan@oakland.edu

www.secs.oakland.edu/~yan
Topics (Part 1)

- Introduction
- Principles of parallel algorithm design (Chapter 3)
- Programming on shared memory system (Chapter 7)
  - OpenMP
  - PThread, mutual exclusion, locks, synchronizations
  - Cilk/Cilkplus (To be covered after lecture 11 and 12)

Analysis of parallel program executions (Chapter 5)
- Performance Metrics for Parallel Systems
  - Execution Time, Overhead, Speedup, Efficiency, Cost
- Scalability of Parallel Systems
- Use of performance tools
Introduction

• Performance Metrics for Parallel Systems
  – Execution Time, Overhead, Speedup, Efficiency, Cost

• Amdahl’s Law

• Scalability of Parallel Systems
  – Isoefficiency Metric of Scalability

• Minimum Execution Time and Minimum Cost-Optimal Execution Time

• Asymptotic Analysis of Parallel Programs

• Other Scalability Metrics
  – Scaled speedup, Serial fraction
Analytical Modeling: Sequential Execution Time

- The execution time of a sequential algorithm
  - Asymptotic execution time as a function of input size
- Identical on any serial platform

- Big-O Notation
  - $O(1)$
  - $O(N)$
  - $O(N^2)$
  - $O(N\log N)$
  - $O(N^3)$
  - ...

**Example: Matrix Multiplication**

```plaintext
int n = A.length;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        sum = 0;
        for (int k = 0; k < n; k++)
            sum = sum + A[i][k]*B[k][j];
        C[i][j] = sum;
    }
}
```

---

Count the number of operations

Total number of operations:
\[= c_0 + c_1 \cdot n + (c_2 + c_3 + c_6) \cdot n \cdot n + (c_4 + c_5) \cdot n \cdot n \cdot n\]

\[= O(n^3)\]
Parallel Execution Time

- Parallel execution time is a function of:
  - input size
  - number of processors (machine performance)
  - communication parameters of target platform (network)

- Implications
  - must analyze parallel program for a particular target platform
    - communication characteristics can differ by more than $O(1)$
  - parallel program = parallel algorithm + platform
Overhead in Parallel Programs

If using two processors, shouldn’t a program run twice as fast?

– Not all parts of the program are parallelized
– A number of overheads incurred when doing it in parallel

Execution Time

- Essential/Excess Computation
- Interprocessor Communication
- Idling
Overheads in Parallel Programs

• Interprocess interactions:
  – Communication
    • Data movement
  – Synchronization/contention

• Idling:
  – Load imbalance
  – Synchronization
    • Sync itself has overhead
  – Serial components

• Excess computation
  – computation not performed by the serial version
    • E.g. replicated computation to minimize communication.
Topic Overview

• Introduction

• Performance Metrics for Parallel Systems
  – Execution Time, Overhead, Speedup, Efficiency, Cost

• Amdahl’s Law

• Scalability of Parallel Systems
  – Isoefficiency Metric of Scalability

• Minimum Execution Time and Minimum Cost-Optimal Execution Time

• Asymptotic Analysis of Parallel Programs

• Other Scalability Metrics
  – Scaled speedup, Serial fraction
Performance Metrics: Execution Time

Does a parallel program run faster than its sequential version?

- Serial time: $T_s$
  - time elapsed between the start and end of serial execution
- Parallel time: $T_p$
  - time elapsed between first process start and last process end
Performance Metrics: Parallel Overhead

What are the cost to enable parallelism?

- **$T_{all}$**: the total time collectively spent by all the processors
  - $T_{all} = p \times T_P$ (p is the number of processors).

- **$T_S$**: serial execution time

- Total parallel overhead **$T_o$**
  - $T_o = T_{all} - T_{S}$
  - $T_o = p \times T_P - T_{S}$
Performance Metrics: Speedup

What is the benefit from increasing parallelism?

- **Speedup (S):** $T_S / T_P$
  - The ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with $p$ identical processing elements.
Performance Metrics: Example

Adding $n$ numbers

- Sequential: $\Theta(n)$
- Using $n$ processing elements.
  - If $n$ is a power of two, in $\log n$ steps by propagating partial sums up a logical binary tree of processors.
Performance Metrics: Example – cont’d

- $\Sigma_i$ denotes the sum of numbers with consecutive labels from $i$ to $j$

- Analysis:
  - An addition takes $t_c$
  - Communication takes $t_s + t_w$
  - $t_c$ and $(t_s + t_w)$ are constant

- Sequential and parallel time:
  - $T_S = \Theta (n)$
  - $T_P = \Theta (\log n)$

- Speedup $S$:
  - $S = \Theta (n / \log n)$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Σ15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Initial data distribution and the first communication step
(b) Second communication step
(c) Third communication step
(d) Fourth communication step
(e) Accumulation of the sum at processing element 0 after the final communication
Performance Metrics: Speedup

- **The yardstick: \( T_s \)**
  - Many serial algorithms available, each with different asymptotic execution time
  - The parallelization of those algorithms varies too

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input</th>
<th>Output</th>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix multiplication</td>
<td>Two ( n \times n ) matrices</td>
<td>One ( n \times n ) matrix</td>
<td>Schoolbook matrix multiplication</td>
<td>( O(n^3) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Strassen algorithm</td>
<td>( O(n^{2.807}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Coppersmith–Winograd algorithm</td>
<td>( O(n^{2.376}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Optimized CW-like algorithms [14] [15] [16]</td>
<td>( O(n^{2.373}) )</td>
</tr>
</tbody>
</table>

Speedup Example: Sorting

procedure bubbleSort( A : vector)
    n := length( A )
do
    swapped := false
    n := n - 1
    for each i in 0 to n - 1
        if A[i] > A[i + 1]
            swap(A[i], A[i + 1]); swapped := true
    while (swapped)
end procedure

procedure oddEvenSort( A : vector)
    n := length( A )
do
    swapped := false
    for each i in 0 to n - 1 by 2 in parallel
        if A[i] > A[i + 1]
            swap(A[i], A[i + 1]); swapped := true
    for each i in 1 to n - 1 by 2 in parallel
        if A[i] > A[i + 1]
            swap(A[i], A[i + 1]); swapped := true
    while (swapped)
end procedure

• The serial execution time for bubblesort: 150 seconds.
• Odd-even parallel bubble sort: is 40 seconds.
• The speedup: 150/40 = 3.75.
  – But is this really a fair assessment of the system?

• What if serial quicksort only took 30 seconds?
• In this case, the speedup is 30/40 = 0.75
  – A more realistic assessment

• Always consider the best sequential program as baseline
  – Not even the parallel program running with 1 PE

• We do this in our assignment
Performance Metrics: Speedup Bounds

• Speedup, in theory, should be upper bounded by $p$
  – We can only expect a $p$-fold speedup if we use $p$ times as many resources.

• Theoretically, a speedup greater than $p$ is possible only if each processor spends less than $T_s/p$ time solving the problem.
  – Violate the rules of using the best sequential as baseline

• Speedups:
  – Linear
  – Sublinear
  – Superlinear

• In practice, superlinear is possible
Performance Metrics: Superlinear Speedups

Parallel algorithm does less work than its serial versions

- Searching node ‘S’ in an unstructured tree
- Parallel with two PEs using depth-first traversal
  - PE 0 searching the left subtree expands only the shaded nodes before the solution is found by PE 1
  - PE 1 searching the right subtree
- Serial algorithm expands the entire tree
  - Does more work than the parallel algorithm.
Performance Metrics: Superlinear Speedups

Resource-based superlinearity

• Parallel execution:
  – The higher aggregate cache/memory bandwidth can result in better cache-hit ratios, and therefore superlinearity.

• Example: A processor with 64KB of cache yields an 80% hit ratio. If two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory.

• If DRAM access time is 100 ns, cache access time is 2 ns, and remote memory access time is 400 ns, this corresponds to a speedup of 2.43!
Performance Metrics: Efficiency

• Fraction of time for which a process perform useful work

\[ E = S/p = T_S / (p \cdot T_P) \]

• Bounds
  – Theoretically, \( 0 \leq E \leq 1 \)
    • The larger, the better
    • \( E=1 \): 0 overhead
  – Practically, \( E > 1 \) if superlinear speedup is achieved

• Previous example: adding \( N \) numbers using \( N \) PEs
  – Speedup: \( S = \Theta (N / \log N) \)
  – Efficiency: \( E = S/N = \Theta (N / \log N) / N = \Theta (1 / \log N) \)
    • Very low when \( N \) is big
Example: Edge Detection

- Apply 3x3 template to each pixel of the images
  - Stencil computation

- Serial performance: $T_S = 9t_c n^2$
  - Each pixel has 9 multiply-add (MA)
    - Each MA takes constant $t_c$ time
  - An $n \times n$ image for $n^2$ pixels

http://en.wikipedia.org/wiki/Edge_detection
Edge Detection: Parallel Version

- Partitions the image equally into vertical segments, each with $n^2 / p$ pixels.

- Computation by each PE: $T_s = 9 \frac{t_c n^2}{p}$

- Communications by each PE: $2(t_s + t_w n)$
  - The boundary of each segment is $2n$ pixels
  - Two boundaries: left and right
  - Each boundary exchange takes $t_s + t_w n$

- Parallel performance: $T_P = 9t_c \frac{n^2}{p} + 2(t_s + t_w n)$
Edge Detection: Parallel Speedup and Efficiency

- **Serial performance:** $T_s = 9t_c n^2$

- **Parallel performance:**
  \[ T_P = 9t_c \frac{n^2}{p} + 2(t_s + t_w n) \]

- **Speedup:** $S = \frac{T_s}{T_p}$

  \[ S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)} \]

- **Efficiency:** $E = \frac{S}{p}$

  \[ E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}} \]
Performance Metrics: Cost

Product of parallel execution time and number of PEs: \( p^*T_p \)

- The total amount of time by all PEs to solve the problem

- *Cost-optimal*: parallel cost \( \cong \) serial cost
  - \( \sim 0 \) overhead
  - \( E = \Theta(1), \text{ since } E = T_s / p^*T_p \)
Cost: An Example

Adding n numbers on n PEs

- Serial performance: $T_S = \Theta(n)$
- Parallel performance: $T_P = \Theta(\log n)$
- Cost: $p \cdot T_P = \Theta(n \log n)$
- Optimal or not:
  - $E = n/n^* \log n = \Theta(1/\log n)$
  - Not cost-optimal.

- Why not optimal
  - Waste of CPU cycles after step 1
    - Only core 0 is doing all the useful work in logN times
Cost: Impact of Non-Cost-Optimality

- A parallel sorting algorithm uses $n$ PEs to sort a list: $(\log n)^2$
- Serial sorting: $n \log n$
- Speedup $S = n / \log n$, Efficiency $E = 1 / \log n$
- Cost: $p T_p = n (\log n)^2$
  - Not cost optimal by a factor of $\log n$.

- If $p < n$, assigning $n$ tasks to $p$ PEs: $T_p = n (\log n)^2 / p$.
- Speedup: $S = (n \log n) / (n (\log n)^2 / p) = p / \log n$.
- For a given $p$, $n \uparrow \rightarrow S \downarrow$
  - Speedup decreases as we increase problem sizes
- Observation:
  - Non-cost-optimality introduce significant cost (overhead)
  - Cost-optimality is important in practice
Effect of Granularity on Performance

• Scaling down a parallel program
  – Reduce the number of PEs than the max possible
  – Increase granularity → Improve parallel efficiency
  – Naïve scaling-down
    • consider each original processor as virtual PEs
    • map virtual PEs to scaled-down number of PEs

• Impacts:
  – # PE decreases by a factor of n / p
  – computation for each PE increases by a factor of n / p
  – communication cost depends upon what the VPEs do
Sum $n$ Numbers on $p$ PEs

- $P < N$
- $P$ and $n$ are power of 2

- Use parallel algorithm for $n$ (virtual) PEs
  - Assign each PE to $n / p$ virtual Pes

- Simulation: Adding 16 numbers on 4 PEs
Sum Reduction: 1
Sum Reduction: 2

Substep 1

Substep 2

Substep 3

Substep 4
Sum Reduction: 3

Simulation of the third step in two substeps

Simulation of the fourth step

Final result
Sum Reduction Example

- Each of the \( p \) PEs is now assigned \( n / p \) virtual PEs.
- The first \( \log p \) of the \( \log n \) steps of the original algorithm are simulated in \( (n / p) \log p \) steps on \( p \) PEs.
- Subsequent \( \log n - \log p \) steps do not require any communication
  - Local processing
- The overall parallel execution time: \( \Theta \left( (n / p) \log p \right) \).
- The cost is \( \Theta(n \log p) \)
  - Asymptotically higher than the sequential time \( \Theta(n) \)

- Therefore, the parallel system is not cost-optimal.
Cost-optimal Way of Scaling Down: 1

- Each PE locally adds its $n / p$ numbers in time $\Theta(n / p)$
- The $p$ partial sums on $p$ PEs can be added in time $\Theta(\log p)$. 

![Diagram showing the process of summing numbers]
Cost-optimal Way of Scaling Down: 2

- Parallel execution time: \[ T_P = \Theta(n/p + \log p), \]

- Parallel cost: \[ p \times T_P = \Theta(n + p \log p) \]

- **Cost optimal as long as** \[ n = \Omega(p \log p) \]
  - \( f = \Omega(g) \): \( f \) dominate \( g \) in some limit
  - E.g. adding 10,000,000 (\( n \)) number using 14 PEs
Topic Overview

• Introduction
• Performance Metrics for Parallel Systems
  – Execution Time, Overhead, Speedup, Efficiency, Cost
• Amdahl’s Law
• Scalability of Parallel Systems
  – Isoefficiency Metric of Scalability
• Minimum Execution Time and Minimum Cost-Optimal Execution Time
• Asymptotic Analysis of Parallel Programs
• Other Scalability Metrics
  – Scaled speedup, Serial fraction
Amdahl’s Law

Amdahl’s law for overall speedup

\[
\text{Overall Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}
\]

\(F = \text{The fraction enhanced}\)

\(S = \text{The speedup of the enhanced fraction}\)

- The word “law” is often used by computer scientists when it is an observed phenomena (e.g., Moore’s Law) and not a theorem that has been proven in a strict sense.

Using Amdahl’s Law

Overall speedup if we make 90% of a program run 10 times faster.

\[ F = 0.9 \quad S = 10 \]

Overall Speedup \[ = \frac{1}{(1 - 0.9) + \frac{0.9}{10}} \quad = \frac{1}{0.1 + 0.09} = 5.26 \]

Overall speedup if we make 80% of a program run 20% faster.

\[ F = 0.8 \quad S = 1.2 \]

Overall Speedup \[ = \frac{1}{(1 - 0.8) + \frac{0.8}{1.2}} \quad = \frac{1}{0.2 + 0.66} = 1.153 \]
Amdahl’s Law for Parallelism

• The enhanced fraction $F$ is through parallelism, perfect parallelism with linear speedup
  – The speedup for $F$ is $N$ for $N$ processors

• Overall speedup

\[
S(N) = \frac{T_s}{T_p} = \frac{T_s}{(1 - F) * T_s + \frac{F * T_s}{N}} = \frac{1}{1 - F + \frac{F}{N}}
\]

• Speedup upper bound (when $N \to \infty$):
  – $1 - F$: the sequential portion of a program
Amdahl’s Law for Parallelism
Amdahl’s Law Usefulness

- Amdahl’s law is valid for traditional problems and has several useful interpretations.
- Some textbooks show how Amdahl’s law can be used to increase the efficient of parallel algorithms
  \[ E = \frac{1}{1 - (1-F) + \frac{F}{N}} \frac{1}{N} = \frac{1}{N(1-F)+F} \]
  - If we increase N, and the problem size in certain rate (so F increased), we can still keep E constant
- Amdahl’s law shows that efforts required to further reduce the fraction of the code that is sequential may pay off in large performance gains.
- Hardware that achieves even a small decrease in the percent of things executed sequentially may be considerably more efficient.
Amdahl’s Law for Parallelism

• However: for long time, Amdahl’s law was viewed as a fatal flaw to the usefulness of parallelism
  – Focuses a particular algorithm and problem sizes, and does not consider that other algorithms with more parallelism may exist, or scalability issues
  – Amdahl’s law applies only to “standard” problems were superlinearity can not occur
  – **Gustafon’s Law**: The proportion of the computations that are sequential normally decreases as the problem size increases.

• Currently, it is generally accepted by parallel computing professionals that Amdahl’s law is not a serious limit the benefit and future of parallel computing.

References

• Adapted from slides “Principles of Parallel Algorithm Design” by Ananth Grama

Project Ideas and Teams

• Performance measurement and analysis based on PAPI performance counters
  – how OpenMP schedule loop chunks
  – how OpenMP schedule tasks

• Application development combining the use of different programming model (OpenMP, MPI and GPU)
  – Artificial intelligence and deep learning application
  – Computer vision
  – Scientific simulation

• New parallel programming experiments: Chapel

• New computing paradigm: neuromorphic

• Other topics