Parallel Performance Theory - 1

Parallel Computing
CIS 410/510
Department of Computer and Information Science
Outline

- Performance scalability
- Analytical performance measures
- Amdahl’s law and Gustafson-Barsis’ law
What is Performance?

- In computing, performance is defined by 2 factors
  - Computational requirements (what needs to be done)
  - Computing resources (what it costs to do it)
- Computational problems translate to requirements
- Computing resources interplay and tradeoff

\[
\text{Performance} \sim \frac{1}{\text{Resources for solution}}
\]

... and ultimately

Hardware

Time

Energy

Money
Why do we care about Performance?

- Performance itself is a measure of how well the computational requirements can be satisfied.
- We evaluate performance to understand the relationships between requirements and resources.
  - Decide how to change “solutions” to target objectives.
- Performance measures reflect decisions about how and how well “solutions” are able to satisfy the computational requirements.

“The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”

Charles Babbage, 1791 – 1871
What is Parallel Performance?

- Here we are concerned with performance issues when using a parallel computing environment
  - Performance with respect to parallel computation
- Performance is the *raison d’être* for parallelism
  - Parallel performance versus sequential performance
  - If the “performance” is not better, parallelism is not necessary
- *Parallel processing* includes techniques and technologies necessary to compute in parallel
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …
- Parallelism must deliver performance
  - How? How well?
Performance Expectation (Loss)

- If each processor is rated at $k$ MFLOPS and there are $p$ processors, should we see $k*p$ MFLOPS performance?
- If it takes 100 seconds on 1 processor, shouldn’t it take 10 seconds on 10 processors?
- Several causes affect performance
  - Each must be understood separately
  - But they interact with each other in complex ways
    - Solution to one problem may create another
    - One problem may mask another
- Scaling (system, problem size) can change conditions
- Need to understand performance space
**Embarrassingly Parallel Computations**

- An embarrassingly parallel computation is one that can be obviously divided into completely independent parts that can be executed simultaneously
  - In a truly embarrassingly parallel computation there is no interaction between separate processes
  - In a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way

- Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms
  - If it takes $T$ time sequentially, there is the potential to achieve $T/P$ time running in parallel with $P$ processors
  - What would cause this not to be the case always?
**Scalability**

- A program can scale up to use many processors
  - What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation
  - If double the number of processors, what to expect?
  - Is scalability linear?
- Use parallel efficiency measure
  - Is efficiency retained as problem size increases?
- Apply performance metrics
Performance and Scalability

- Evaluation
  - Sequential runtime ($T_{seq}$) is a function of
    - problem size and architecture
  - Parallel runtime ($T_{par}$) is a function of
    - problem size and parallel architecture
    - # processors used in the execution
  - Parallel performance affected by
    - algorithm + architecture

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
Performance Metrics and Formulas

- \( T_1 \) is the execution time on a single processor
- \( T_p \) is the execution time on a \( p \) processor system
- \( S(p) \) (\( S_p \)) is the speedup
  \[
  S(p) = \frac{T_1}{T_p}
  \]
- \( E(p) \) (\( E_p \)) is the efficiency
  \[
  \text{Efficiency} = \frac{S_p}{p}
  \]
- \( \text{Cost}(p) \) (\( C_p \)) is the cost
  \[
  \text{Cost} = p \times T_p
  \]
- Parallel algorithm is cost-optimal
  - Parallel time = sequential time (\( C_p = T_1 \), \( E_p = 100\% \))
Amdahl’s Law (Fixed Size Speedup)

- Let $f$ be the fraction of a program that is sequential
  - $1-f$ is the fraction that can be parallelized
- Let $T_1$ be the execution time on 1 processor
- Let $T_p$ be the execution time on $p$ processors
- $S_p$ is the speedup
  \[ S_p = \frac{T_1}{T_p} = \frac{T_1}{(fT_1 + (1-f)T_1/p)} = \frac{1}{f + (1-f)/p} \]
- As $p \to \infty$, $S_p = \frac{1}{f}$
Amdahl’s Law and Scalability

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- When does Amdahl’s Law apply?
  - When the problem size is fixed
  - *Strong scaling* ($p \to \infty$, $S_p = S_\infty \to 1 / f$)
  - Speedup bound is determined by the degree of sequential execution time in the computation, not # processors!!!
  - Uhh, this is not good … Why?
  - Perfect efficiency is hard to achieve
- See original paper by Amdahl on webpage
Gustafson-Barsis’ Law (Scaled Speedup)

- Often interested in larger problems when scaling
  - How big of a problem can be run (HPC Linpack)
  - Constrain problem size by parallel time

- Assume parallel time is kept constant
  - \( T_p = C = (f + (1-f)) \times C \)
  - \( f_{seq} \) is the fraction of \( T_p \) spent in sequential execution
  - \( f_{par} \) is the fraction of \( T_p \) spent in parallel execution

- What is the execution time on one processor?
  - Let \( C=1 \), then \( T_s = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par} \)

- What is the speedup in this case?
  - \( S_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par} \)
Gustafson-Barsis’ Law and Scalability

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

- When does Gustafson’s Law apply?
  - When the problem size can increase as the number of processors increases
  - Weak scaling \( S_p = 1 + (p-1)f_{par} \)
  - Speedup function includes the number of processors!!!
  - Can maintain or increase parallel efficiency as the problem scales

- See original paper by Gustafson on webpage
Amdahl versus Gustafson-Baris

Amdahl

serial work

parallelizable work

Time

P=1  P=2  P=4  P=8
Amdahl versus Gustafson-Baris

Gustafson-Baris

serial work

parallelizable work

P=1  P=2  P=4  P=8

Time
DAG Model of Computation

- Think of a program as a directed acyclic graph (DAG) of tasks
  - A task can not execute until all the inputs to the tasks are available
  - These come from outputs of earlier executing tasks
  - DAG shows explicitly the task dependencies
- Think of the hardware as consisting of workers (processors)
- Consider a greedy scheduler of the DAG tasks to workers
  - No worker is idle while there are tasks still to execute
**Work-Span Model**

- $T_P = \text{time to run with } P \text{ workers}$
- $T_1 = work$
  - Time for serial execution
    - execution of all tasks by 1 worker
  - Sum of all work
- $T_\infty = span$
  - Time along the critical path
- Critical path
  - Sequence of task execution (path) through DAG that takes the longest time to execute
  - Assumes an infinite # workers available
Work-Span Example

- Let each task take 1 unit of time
- DAG at the right has 7 tasks
- $T_1 = 7$
  - All tasks have to be executed
  - Tasks are executed in a serial order
  - Can the execute in any order?
- $T_\infty = 5$
  - Time along the critical path
  - In this case, it is the longest pathlength of any task order that maintains necessary dependencies
Lower/Upper Bound on Greedy Scheduling

- Suppose we only have $P$ workers
- We can write a work-span formula to derive a lower bound on $T_P$
  - $\max(\frac{T_1}{P}, T_\infty) \leq T_P$
- $T_\infty$ is the best possible execution time
- Brent’s Lemma derives an upper bound
  - Capture the additional cost executing the other tasks not on the critical path
  - Assume can do so without overhead
  - $T_P \leq (T_1 - T_\infty) / P + T_\infty$
Consider Brent’s Lemma for 2 Processors

- $T_1 = 7$
- $T_\infty = 5$
- $T_2 \leq (T_1 - T_\infty) / P + T_\infty$
  \[ \leq (7 - 5) / 2 + 5 \]
  \[ \leq 6 \]
Amdahl was an optimist!
Estimating Running Time

- Scalability requires that $T_\infty$ be dominated by $T_1$

$$T_P \approx \frac{T_1}{P} + T_\infty \text{ if } T_\infty << T_1$$

- Increasing work hurts parallel execution proportionately
- The span impacts scalability, even for finite $P$
Parallel Slack

- Sufficient parallelism implies linear speedup

\[ T_p \approx \frac{T_1}{P} \quad \text{if} \quad \frac{T_1}{T_\infty} \gg P \]
Asymptotic Complexity

- Time complexity of an algorithm summarizes how the execution time grows with input size.
- Space complexity summarizes how memory requirements grow with input size.
- Standard work-span model considers only computation, not communication or memory.
- Asymptotic complexity is a strong indicator of performance on large-enough problem sizes and reveals an algorithm’s fundamental limits.
Definitions for Asymptotic Notation

- Let \( T(N) \) mean the execution time of an algorithm
- **Big O notation**
  - \( T(N) \) is a member of \( O(f(N)) \) means that
    \[ T(N) \leq c \cdot f(N) \] for constant \( c \)
- **Big Omega notation**
  - \( T(N) \) is a member of \( \Omega(f(N)) \) means that
    \[ T(N) \geq c \cdot f(N) \] for constant \( c \)
- **Big Theta notation**
  - \( T(N) \) is a member of \( \Theta(f(N)) \) means that
    \[ c_1 \cdot f(n) \leq T(N) < c_2 \cdot f(N) \] for constants \( c_1 \) and \( c_2 \)