Lecture 2: Basic Performance Models For Extreme Scale Systems

William Gropp

www.cs.illinois.edu/~wgropp
Performance is Key

- Parallelism is (usually) used to get more performance
  - How do you know if you are making good (not even best) use of a parallel system?
- Even measurement-based approaches can be (and all to often are) performed without any real basis of comparison
  - The key questions are
    - Where is most of the time spent?
    - What is the achievable performance, and how do I get there?
  - This latter is often overlooked, leading to erroneous conclusions based on the (immature) state of compiler / runtime / code implementations
Tuning A (Parallel) Code

• Typical Approach
  ♦ Profile code. Determine where most time is being spent
  ♦ Study code. Measure absolute performance, look at performance counters, compare FLOP rates
  ♦ Improve code that takes a long time, reduce time spent in “unproductive” operations

• Why this isn’t the right Approach:
  ♦ How do you know when you are done?
  ♦ How do you know how much performance improvement you can obtain?

• Why is it hard to know?
  ♦ Many problems are too hard to solve without extreme scale computing
  ♦ Its getting harder and harder to provide performance without specialized hardware
Blue Waters Computing System

- **Spectra Logic**: 300 PBs
  - 100 GB/sec

- **Sonexion**: 26 PBs
  - >1 TB/sec

- **10/40/100 Gb Ethernet Switch**: 120+ Gb/sec

- **IB Switch**: >1 TB/sec

- **WAN**
Heart of Blue Waters: Two Chips

**AMD Interlagos**
157 GF peak performance

**Features:**
- 2.3-2.6 GHz
- 8 core modules, 16 threads
- On-chip Caches
  - L1 (I:8x64KB; D:16x16KB)
  - L2 (8x2MB)
- Memory Subsystem
  - Four memory channels
  - 51.2 GB/s bandwidth

**NVIDIA Kepler**
1,400 GF peak performance

**Features:**
- 15 Streaming multiprocessors (SMX)
  - SMX: 192 sp CUDA cores, 64 dp units, 32 special function units
- Memory subsystem
  - Six memory channels
  - 180 GB/s bandwidth
What is an Extreme Scale System Today?

- **Tianhe 2 (China):**
  - 16,000 nodes, each with 2 Intel Ivy Bridge Xeon processors and 3 Xeon Phi coprocessors
  - 3,120,000 cores
  - Interconnect is a “fat tree” of 13 switches, each with 576 ports

- **Sequoia (USA):**
  - IBM Blue Gene/Q. 98,304 nodes, each with 16 (+1) cores
  - Interconnect is 5 dimensional torus
Likely Directions for Extreme Scale Systems

- **5 Years (2020)**
  - Peak performance over 1 ExaFLOPs \(10^{18}\) ops/sec
  - 100k “nodes”
  - Heterogeneous nodes

- **10 Years (2025)**
  - Peak performance over 30 ExaFLOPs
  - Computing distributed throughout node and memory

- **15 Years (2030)**
  - Peak performance over 100 ExaFLOPs
  - Radically different systems emerging
    - New digital logic, e.g., nanotubes
    - New computing models, e.g., quantum or molecular
Why Performance Modeling?

- **What is the goal?**
  - It is *not* precise predictions
  - It *is* insight into whether a code is achieving the performance it could, and if not, how to fix it

- **Performance modeling can be used**
  - To estimate the baseline performance
  - To estimate the potential benefit of a nontrivial change to the code
  - To identify the critical resource
What do I mean by Performance Modeling?

• Actually two different models
  ♦ First, an analytic expression based on the application code
  ♦ Second, an analytic expression based on the application’s *algorithm* and data structures

• Note that a series of measurements from benchmarks are *not* a performance model

• Why this sort of modeling
  ♦ The obvious: extrapolation to other systems, such as scalability in nodes or different interconnect
  ♦ Also: comparison of the two models with observed performance can identify
    • Inefficiencies in compilation/runtime
    • Mismatch in developer expectations
Different Philosophies for Performance Models

- Simulation:
  - Very accurate prediction, little insight beyond specifics of the simulation itself

- Traditional Performance Modeling (PM):
  - Focuses on accurate predictions
  - Tool for computer scientists, not application developers

- PM as part of the software engineering process
  - PM for design, tuning and optimization
  - PMs are developed with algorithms and used in each step of the development cycle

- Performance Engineering
Example

• Lets look at a simple example
• Matrix-matrix multiply
  ♦ Classic example, often used in discussion of compiler optimizations
  ♦ Core of the “HPLinpack” benchmark
  ♦ Simple to express: In Fortran,
    do i=1, n
    do j=1,n
      c(i,j) = 0
      do k=1,n
        c(i,j) = c(i,j) + a(i,k) * b(k,j)
Performance Estimate

• How fast should this run?
  ♦ Standard complexity analysis in numerical analysis counts floating point operations
  ♦ Our matrix-matrix multiply algorithm has $2n^3$ floating point operations
    • 3 nested loops, each with $n$ iterations
    • 1 multiply, 1 add in each inner iteration
  ♦ For $n=100$, $2 \times 10^6$ operations, or about 1 msec on a 2GHz processor :)
  ♦ For $n=1000$, $2 \times 10^9$ operations, or about 1 sec
The Reality

• N=100
  ◆ 1818 MF (1.1ms)

• N=1000
  ◆ 335 MF (6s)

• What this tells us:
  ◆ Obvious expression of algorithms are not transformed into leading performance.
Thinking about Performance

- The performance model assumes the computer looks like the figure on the right
  - Memory is infinitely large
  - Memory is infinitely fast
Thinking about Performance

- We will incrementally improve our performance models by adding features to our model of the computer hardware
  - That model of the computer hardware is a major part of what is often called an execution model
- In the first enhancement, let's make memory not infinitely fast
A Simple Performance Model

• Use the following:
  ♦ Number of operations (e.g., floating point multiply)
  ♦ Number of loads from memory
  ♦ Number of stores to memory

• We are ignoring for now the many features of an architecture that are used to optimize performance
  ♦ We will cover many of them during the class
A Simple Example

- Consider this code:
  
  ```
  Do i=1,n
    y(i) = a*x(i) + y(i)
  enddo
  ```

  - 2n operations (floating add, floating multiply)
  - 2n Loads (x(i) and y(i) for i=1 to n)
  - N Stores (y(i))
Performance Model

• Assume that
  \[ c = \text{time for operation} \]
  \[ r = \text{time to read an element} \]
  \[ w = \text{time to write an element} \]

• Then a very crude estimate of the time for this operation is
  \[ T = n(2c + 2r + w) \]

• Call this a *model* because it is too crude to be an estimate
Some Comments on This Model

- Many analysis of algorithms set r and w to zero
- We will spend much of our time considering different ways to model communication time
  - Load and Store to memory
  - Sharing of data between threads
  - Communication between nodes in a parallel computer
  - Load and Store to a file system
Discussion Topics for Matrix-Matrix Multiply

- Why do you think the algorithm runs slowly at large sizes?
- Why do you think the compiler doesn’t do a better job?
- What about other algorithms such as Strassen’s algorithm?
  - How would that algorithm change this analysis?