CSCI-580 Advanced High Performance Computing

Lecture 2: Fundamentals of Parallel Processing

Bo Wu
Colorado School of Mines

Part of slides from: Katherine Yelick (UC Berkeley)
Cris Cecka (Harvard) and Scott Baden (UCSD)

Mines CSCI-580, Bo Wu
Why writing parallel programs is hard
Principles of Parallel Programming

- Finding enough parallelism
- Granularity
- Locality
- Load balance
- Coordination and synchronization
Hidden Parallelism in Modern Machines

- Bit-level parallelism
  - within floating point operations
- Instruction-level parallelism (ILP)
  - multiple instructions executed per clock cycle
- Memory system parallelism
  - overlap of memory operations with computation
- OS parallelism
  - Multiple jobs running in parallel on commodity SMPs
Finding Enough Parallelism

- Suppose only part of an application can be parallelized
- Amdahl’s law
  - Let $s$ be the fraction of work done sequentially, $(1-s)$ is fraction parallelizable
  - $P =$ no. of processors
    
    $\text{Speedup} \leq ?$
Finding Enough Parallelism

○ Suppose only part of an application can be parallelized

○ Amdahl’s law
  – Let s be the fraction of work done sequentially, (1-s) is fraction parallelizable
  – P = no. of processors

\[
\text{Speedup} = \frac{\text{Time}(1)}{\text{Time}(P)} \\
\leq \frac{1}{s + \frac{1-s}{P}} \\
\leq \frac{1}{s}
\]

○ Even if the parallel part speeds up perfectly performance is limited by the sequential part
Overhead of Parallelism

- Given enough parallel work, this is the biggest barrier to getting enough speedup.
- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronization
  - extra redundant computation
- Each of these can be in the range of milliseconds on some systems.
- Algorithm needs sufficient large enough units of work to run fast in parallel.
Locality

- Large memories are slow, fast memories are small
- Local memory is fast, remote memory is slow
- Should access small and local memory most of the time
Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to
  - insufficient parallelism (during that phase)
  - unequal size tasks
- Examples of the latter
  - unstructured problems (e.g., graph processing)
- Algorithm needs to balance load
Load Balance Techniques

- Static load balancing
  - Prior knowledge of the parallel tasks (e.g., approx. execution time of each)
  - Fixed assignment of tasks to processors

- Dynamic load balancing
  - Dynamic adjustment of the assignment
Load Imbalance Discussion

- We have 100 boxes of gold here and two workers
- Problem: move the gold to Provost’s office
- Constraint: partition the work beforehand

- Solution 1:
  - Test how strong and how fast they are
  - Build a model and assign the work

- Solution 2:
  - Just evenly partition the work
  - If one finishes earlier, help the other (work stealing)
Measuring Performance
Improving Real Performance

○ Peak performance grows exponentially due to Moore’s law
  – in 1990’s, peak performance increased 100x; in 2000’s, it increased 1000x

○ Efficiency (the performance relative to hardware peak performance) dropped
  – was 40-50% on the vector supercomputers of 1990s
  – now as little as 5-10% on parallel supercomputers of today

○ Close the gap through ……
  – Mathematical methods and algorithms that achieve high performance on a single node and scale to hundreds of thousands of processors
  – More efficient programming models and tools for massively parallel supercomputers
Performance Levels

- Peak advertised performance
  - You can’t possibly compute faster than this speed

- LINPACK
  - The “hello word” program for parallel computing
  - Solve $Ax=b$ using Gaussian Elimination, highly tuned

- Gorden Bell Prize Winning applications performance
  - The right application/algorithm/platform combination + years of hard work

- Average sustained applications performance
  - What one can expect for standard applications

These levels are often confused even in reviewed publications
Performance Levels (for example on NERSC-3)

- Peak advertised performance (PAP): 5 Tflop/s
- LINPACK: 3.05 Tflop/s
- Gordon Bell Prize winning applications performance: 2.46 Tflop/s
  - Material Science application at SC01
- Average sustained applications performance: ~0.4 Tflop/s
  - Less than 10% peak!
Categorizing machines in terms of hardware organization
Flynn’s Taxonomy

- Michael Flynn, 1966
- Dimensions
  - Instruction streams: single (SI) or multiple (MI)
  - Data streams: single (SD) or multiple (MD)
Additional Terms

- **SSE**
  - Streaming SIMD Extensions
  - Intel’s programming model for SIMD

- **SPMD**
  - Single Program, Multiple Data
  - Same program, but not in lockstep like SIMD
  - e.g., MPI

- **SIMT**
  - Single Instruction, Multiple Threads
  - Threads operate on their own data; Threads can diverge
  - e.g., CUDA
Memory Taxonomy

- Shared memory
- Distributed memory
Shared Memory

- All processors share a single address space
- Communicate with each other by writing and reading shared variables
- Hardware automatically performs the global to local mapping using address translation mechanisms
- 2 types, according to uniformity of memory access times
  - UMA: Uniform Memory Access time
  - NUMA: Non-Uniform Memory Access time
Uniform Memory Access (UMA)

- Identical processors share a connection to a common memory
- Symmetric Multiprocessor (SPM) machines
Non-Uniform Memory Access (NUMA)

- Memory is shared, but some blocks are physically closer to some processors
- Significant differences of memory access times between processors
Shared Memory Properties

- **Good**
  - Global address space leads to simple programming models
  - Data sharing between tasks is fast and easy

- **Bad**
  - Complex hardware
  - Not scalable
  - Programmer responsible for “correct” access (e.g., locks, semaphores)
Distributed Memory

- Each processor has its own memory address space
- Processors communicate via a network
- Large multi-processor system
Distributed Memory Properties

- **Good**
  - Scalable with number of processors
  - Cost effective hardware, can use off-the-shelf processors and networking

- **Bad**
  - More complex programming model
  - Programmer responsible for communication
  - Difficult to map some traditional data structures to this memory organization
Hybrid Memory

- Most supercomputers employ both shared and distributed memory systems
- Networking of multiple SMP/GPU machines
Attention Curve
Parallel Programming Models
What is a Parallel Programming Model?

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - What orderings exist between operations?
  - How do different threads of control synchronize?
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations can be used to coordinate parallelism
  - What are the atomic (indivisible) operations?
- Cost
  - How do we account for the cost of each of the above?
Consider applying a function $f$ to the elements of an array $A$ and then computing its sum:

$$\sum_{i=0}^{n-1} f(A[i])$$

Questions:
- Where does $A$ live? All in single memory?
- Partitioned?
- What work will be done by each processor?
- They need to coordinate to get a single result, how?

Simple Example

- $A$ = array of all data
- $fA = f(A)$
- $s = \text{sum}(fA)$
Programming Model 1: Shared Memory

- Program is a collection of threads of control.
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
- Threads coordinate by synchronizing on shared variables.
Model 1a: SMP

- Processors all connected to a large shared memory.
  - SGI, Sun (now part of Oracle), HP, Intel, IBM SMPs
  - Multicore chips, except that all caches are shared

- Difficulty scaling to large numbers of processors
  - $\leq 32$ processors typical

- Advantage: UMA

- Cost: much cheaper to access data in cache than main memory.
Model 1b: Multithreaded Processor

- Multiple thread “contexts” without full processors
- Memory and some other state is shared
- Sun Niagra processor (for servers)
  - Up to 32 threads all running simultaneously
  - In addition to sharing memory, they share floating point units
  - Why? Switch between threads for long-latency memory operations
Model 1c: Distributed Shared Memory

- Memory is logically shared, but physically distributed
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
- SGI Origin is canonical example (+ research machines)
  - Scales to 512
  - Limitation is cache coherency protocols – how to keep cached copies of the same address consistent
Programming Model 2: Message Passing

- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data.

- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is most commonly used SW
Model 2a: Distributed Memory (cluster)

- Each processor has its own memory and cache but cannot directly access another processor’s memory.
- Each “node” has a Network Interface (NI) for all communication and synchronization.
Model 2b: Internet Computing

- SETI@Home
  - >500,000 PCs from Internet
- Goal: detecting intelligent life outside Earth

- Rationale: narrow-bandwidth radio signals from space not known to occur naturally
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope
Programming Model 3: Data Parallel

- Single thread of control consisting of parallel operations.
  - Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - Communication is implicit in parallel operators
  - Elegant and easy to understand and reason about
  - Coordination is implicit – statements executed synchronously

- Drawbacks:
  - Not all problems fit this model
  - Difficult to map onto coarse-grained machines
A large number of (usually) small processors.
- A single “control processor” issues each instruction.
- Each processor executes the same instruction.
- Some processors may be turned off on some instructions.

Originally machines were specialized to scientific computing,
Programming model can be implemented in the compiler.
Vector architectures are based on a single processor
  - Multiple functional units
  - All performing the same operation
  - Instructions may specify large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel

Historically important

Re-emerging in recent years
  - At a large scale in the Earth Simulator (NEC SX6) and Cray X1
  - At a small scale in SIMD media extensions to microprocessors SSE, SSE2 (Intel: Pentium/IA64)

Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn’t have to
Vector Processors

- Vector instructions operate on a vector of elements
  - These are specified as operations on vector registers

- Compiler vectorizes scientific code successfully
  - Failed terribly for commercial code
  - An Evaluation of Vectorizing Compilers
    (polaris.cs.uiuc.edu/~garzaran/doc/pact11.pdf)
Machine Model 4: Clusters of SMPs

- SMPs are the fastest commodity machine, so use them as a building block for a larger machine with a network.

- Common names:
  - CLUMP = Cluster of SMPs
  - Hierarchical machines, constellations

- Many modern supercomputers look like this

- What is an appropriate programming model?
  - Treat machine as “flat”, always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
  - Shared memory within one SMP, but message passing outside of an SMP.
Common problems in shared memory machines
Cache Coherence

- A central design issue in shared memory architectures
- Processors may read and write the same cached memory location
- If one processor writes to the location, all others must eventually see the write

\[
x := 1 \quad \text{memory}
\]
Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system does not have a coherent value for X

```
x:=1    memory

x:=2    P1
x:=1    P2
```
Cache Coherence Protocols

- Ensure that all processors eventually see the same value
- Two options
  - Update-on-write
  - Invalidate-on-write

```plaintext
x := 1       memory

x := 2       P1       x := 2       P2
```
Cache Coherence Protocols

- Ensure that all processors eventually see the same value
- Two policies
  - Update-on-write
  - Invalidate-on-write

\[
\begin{align*}
x &:= 1 & \text{memory} \\
x &:= 2 & P1 & \quad \text{invalidate} & \quad x &:= 1 & P2
\end{align*}
\]
Memory consistency and correctness

- Memory consistency defines shared memory correctness
- Example:
  - The class room is scheduled at 269
  - Registrar’s office chances mind and reschedule it to 169
  - Registrar asks me to update the course page
  - Registrar sends emails to students
  - Students go to?
Memory consistency model

- The memory consistency model determines when a written value will be seen by a reader.
- Sequential Consistency maintains a linear execution on a parallel architecture that is consistent with the sequential execution of some interleaved arrangement of the separate concurrent instruction streams.
  - Expensive to implement.
- Relaxed consistency.
  - Enforce consistency only at well defined times.
  - Useful in handling false sharing.
Next class: More on Parallel Programming Models