CS 5220: Parallel machines and models

David Bindel

2017-09-07

Why clusters?

- Clusters of SMPs are everywhere
    - Commodity hardware – economics! Even supercomputers now use commodity CPUs (+ specialized interconnects).
    - Relatively simple to set up and administer (?)
- But still costs room, power, ...
- Economy of scale $\implies$ clouds?
    - Amazon and MS now have HPC instances (GCP, too)
    - Microsoft has Infiniband connected instances
    - Several bare-metal HPC/cloud providers
    - Lots of interesting challenges here

Cluster structure

Consider:

- Each core has vector parallelism
- Each chip has six cores, shares memory with others
- Each box has two chips, shares memory
- Each box has two Xeon Phi accelerators
- Eight instructional nodes, communicate via Ethernet

How did we get here? Why this type of structure? And how does the programming model match the hardware?

Parallel computer hardware

Physical machine has *processors*, *memory*, *interconnect*.

- Where is memory physically?
- Is it attached to processors?
- What is the network connectivity?

Parallel programming model

Programming *model* through languages, libraries.

- Control
    - How is parallelism created?
    - What ordering is there between operations?
- Data
    - What data is private or shared?
    - How is data logically shared or communicated?
- Synchronization
    - What operations are used to coordinate?
    - What operations are atomic?
- Cost: how do we reason about each of above?

Simple example

Consider dot product of x and y.

- Where do arrays x and y live? One CPU? Partitioned?
- Who does what work?
- How do we combine to get a single final result?

Shared memory programming model

Program consists of *threads* of control.

- Can be created dynamically
- Each has private variables (e.g. local)
- Each has shared variables (e.g. heap)
- Communication through shared variables
- Coordinate by synchronizing on variables
- Examples: OpenMP, pthreads

Shared memory dot product

Dot product of two n vectors on $p \ll n$ processors:

1. Each CPU evaluates partial sum (n/p elements, local)
2. Everyone tallies partial sums

Can we go home now?

Race condition

A *race condition*:

- Two threads access same variable, at least one write.
- Access are concurrent – no ordering guarantees
    - Could happen simultaneously!

Need synchronization via lock or barrier.

Race to the dot

Consider `S += partial_sum` on 2 CPU:

- P1: Load `S`
- P1: Add `partial_sum`
- P2: Load `S`
- P1: Store new `S`
- P2: Add `partial_sum`
- P2: Store new `S`

Shared memory dot with locks

Solution: consider `S += partial_sum` a *critical section*

- Only one CPU at a time allowed in critical section
- Can violate invariants locally
- Enforce via a lock or mutex (mutual exclusion variable)

Dot product with mutex:

1. Create global mutex `l`
2. Compute `partial_sum`
3. Lock `l`
4. S += partial_sum
5. Unlock `l`

Shared memory with barriers

- Lots of sci codes have phases (e.g. time steps)
- Communication only needed at end of phases
- Idea: synchronize on end of phase with *barrier*
    - More restrictive (less efficient?) than small locks
    - But easier to think through! (e.g. less chance of deadlocks)
- Sometimes called *bulk synchronous programming*

Shared memory machine model

- Processors and memories talk through a bus
- Symmetric Multiprocessor (SMP)
- Hard to scale to lots of processors (think ≤ 32)
    - Bus becomes bottleneck
    - *Cache coherence* is a pain
- Example: Six-core chips on cluster

Multithreaded processor machine

- Maybe threads > processors!
- Idea: Switch threads on long latency ops.
- Called *hyperthreading* by Intel
- Cray MTA was an extreme example

Distributed shared memory

- Non-Uniform Memory Access (NUMA)
- Can *logically* share memory while *physically* distributing
- Any processor can access any address
- Cache coherence is still a pain
- Example: SGI Origin (or multiprocessor nodes on cluster)
- Many-core accelerators tend to be NUMA as well

Message-passing programming model

- Collection of named processes
- Data is *partitioned*
- Communication by send/receive of explicit message
- Lingua franca: MPI (Message Passing Interface)

Message passing dot product: v1

Processor 1:

1. Partial sum s1
2. Send s1 to P2
3. Receive s2 from P2
4. s = s1 + s2

Processor 2:

1. Partial sum s2
2. Send s2 to P1
3. Receive s1 from P1
4. s = s1 + s2

What could go wrong? Think of phones vs letters...

Message passing dot product: v1

Processor 1:

1. Partial sum s1
2. Send s1 to P2
3. Receive s2 from P2
4. s = s1 + s2

Processor 2:

1. Partial sum s2
2. Receive s1 from P1
3. Send s2 to P1
4. s = s1 + s2

Better, but what if more than two processors?

MPI: the de facto standard

- Pro: *Portability*
- Con: least-common-denominator for mid 80s

The “assembly language” (or C?) of parallelism...
but, alas, assembly language can be high performance.

Distributed memory machines

- Each node has local memory
    - … and no direct access to memory on other nodes
- Nodes communicate via network interface
- Example: our cluster!
- Other examples: IBM SP, Cray T3E

The story so far

- Even *serial* performance is a complicated function of the underlying architecture and memory system. We need to understand these effects in order to design data structures and algorithms that are fast on modern machines. Good serial performance is the basis for good parallel performance.
- *Parallel* performance is additionally complicated by communication and synchronization overheads, and by how much parallel work is available. If a small fraction of the work is completely serial, Amdahl's law bounds the speedup, independent of the number of processors.
- We have discussed serial architecture and some of the basics of parallel machine models and programming models.
- Now we want to describe how to think about the shape of

Reminder: what do we want?

- High-level: solve big problems fast
- Start with good *serial* performance
- Given p processors, could then ask for
    - Good *speedup*: p^{-1} times serial time
    - Good *scaled speedup*: p times the work in same time
- Easiest to get good speedup from cruddy serial code!

Parallelism and locality

- Real world exhibits *parallelism* and *locality*
  - Particles, people, etc function independently
  - Nearby objects interact more strongly than distant ones
  - Can often simplify dependence on distant objects
- Can get more parallelism / locality through model
  - Limited range of dependency between adjacent time steps
  - Can neglect or approximate far-field effects
- Often get parallism at multiple levels
  - Hierarchical circuit simulation
  - Interacting models for climate
  - Parallelizing individual experiments in MC or optimization

Basic styles of simulation

- Discrete event systems (continuous or discrete time)
    - Game of life, logic-level circuit simulation
    - Network simulation
- Particle systems
    - Billiards, electrons, galaxies, ...
    - Ants, cars, ...?
- Lumped parameter models (ODEs)
    - Circuits (SPICE), structures, chemical kinetics
- Distributed parameter models (PDEs / integral equations)
    - Heat, elasticity, electrostatics, ...

Often more than one type of simulation appropriate.
Sometimes more than one at a time!