CS 5220

Parallel HW and Models

David Bindel

2024-09-05

Logistics

Partner finding

- Please post on Ed if you need someone!
- Looking for groups of 2-3

Instance setup

- Let's use `c4-standard-2` (vs `e2`)
  - C4 machines are Intel Emerald Rapids
  - Also gives PMU access
  - Recommend a larger boot disk (20 GB)
- Change of plans for Proj 1: C4 for timing

Instance setup

- You can use Intel oneAPI tools
    - Need 20 GB disk in setup (tools take 10 GB)
    - Make sure you set up your environment for it
    - Intel Advisor and compilers are nice
- Intel Advisor
    - Gives a variety of reports (including Roofline!)
    - Note offline HTML report mode

Roofline

Basic idea

Log-log plot showing memory/compute bottlenecks.

- Y axis: Performance (usu Gflop/s)
- X axis: Operational intensity (usu flops/byte read)
- Diagonals: Memory bottlenecks
- Horizontals: Compute bottlenecks
- Performance sits “under the roof”

One core (GCP C4, Emerald Rapids)

See source example

References

Roofline: An Insightful Visual Performance Model for Multicore Architectures, Communications of the ACM, 2009, 52(4).

Parallel Models and HW

Hardware

Basic components: *processors*, *memory*, *interconnect*.

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

Model

Programming model through languages, libraries.

- What are the control mechanisms?
- What data semantics? Private, shared?
- What synchronization constructs?

For performance, need cost models (involves HW)!

Dot product

```c
double dot(int n, double* x, double* y)
{
    double s = 0;
    for (int i = 0; i < n; ++i)
        s += x[i] * y[i];
    return s;
}
```

Dot product

```c
double pdot(int n, double* x, double* y)
{
    double s = 0;

    // Somehow parallelize over this loop
    for (int p = 0; p < NPROC; ++p) {
        int i = p*n/NPROC;
        int inext = (p+1)*n/NPROC;
        double partial = dot(inext-i, x+i, y+i);
        s += partial;
    }
    return s;
}
```

Basic considerations

How can we parallelize dot product?

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

Shared Memory Model

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
- Coordination by synchronizing on variables
- Example: OpenMP

Dot product

Consider `pdot` on $p \ll n$ processors:

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

Of course, it can't be that simple...

Race condition

A *race condition* is when:

- Two threads access the same variable
- At least one is a write.
- Accesses are concurrent
  - No ordering guarantees
  - Could happen “simultaneously”!

Race to the dot

Consider `s += partial` on two CPUs (`s` shared).

Race to the dot

Processor 1	Processor 2
load S	…
add partial	…
…	load S
store S	…
…	add partial
…	store S

Sequential consistency?

Implicitly assumed *sequential consistency*:

- Idea: Execution is as if processors take turns, in some order
- Convenient for thinking through correctness
- Hard to implement in a performant way!
- Will talk about "*memory models*" later

Locks

Can consider `s += partial` a *critical section*

- Only one thread at a time allowed in critical section
- Can violate invariants locally
- Mechanisms: lock or mutex, monitor

Shared dot with locks

Dot product with mutex:

- Create global mutex `l`
- Compute `partial`
- Lock `l`
- `s += partial`
- Unlock `l`

Still need to synchronize on return...

A problem

Processor 1

1. Acquire lock 1
2. Acquire lock 2
3. Do something
4. Release locks

Processor 2

1. Acquire lock 2
2. Acquire lock 1
3. Do something
4. Release locks

What if both processors execute step 1 simultaneously?

Barriers

- Many scientific codes have phases (time steps, iterations)
- Communication only needed at end of phases
- Idea: synchronize on end of phase with *barrier*
    - More restrictive than small locks
    - But easier to think through (no deadlocks)!
- Sometimes called *bulk synchronous programming*

Dot with barriers

```
// Shared array partials
partials[omp_get_thread_num()] = partial;
#pragma omp barrier

double s = 0;
for (int i = 0; i < omp_get_num_threads(); ++i)
    s += partials[i];
```

Punchline

Shared memory *correctness* is hard

- Too little synchronization: races
- Too much synchronization: deadlock
- And both can happen at once!

And this is before we talk performance!

Shared Memory HW

Uniform shared memory

- Processors and memories talk through a bus
- Symmetric multiprocessor
- Hard to scale to lots of processors
    - Bus becomes bottleneck
    - But *cache coherence* via snooping

Distributed shared memory

- Non-Uniform Memory Access (NUMA)
  - Includes most big modern chips
  - Also many-core accelerators
- Memory *logically* shared, *physically* distributed
- Any processor can access any address
- Close accesses (affinity) faster than far accesses
- Cache coherence is a pain

Punchline

Shared memory is expensive!

- Uniform access means bus contention
- Non-uniform access scales better
  - But now access costs vary
- Cache coherence is tricky regardless
- May forgo sequential consistency for performance

Message-Passing Programming

Programming model

- Collection of named (indexed) processes
- Data is *partitioned*
- Communication by send/receive of explicit messages
    - *One-sided* put/get verges on shared memory
- Lingua franca: MPI (Message Passing Interface)

Dot product (v1)

Processor 1

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

What could go wrong?

Processor 2

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

Dot product (v2)

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?

- This is part of why we have `MPI_Sendrecv`
- Also, `MPI_Allreduce`

MPI: The de facto standard

- Pro: *Portability*
- Con: Feels like assembly language for communication
    - So use higher-level libraries on top

Punchline

- Message passing hides less than shared memory
- But correctness is still subtle

Distributed Memory Machines

Hardware setup

- Each node has local memory
  - … and no direct access to memory on other nodes
  - Except maybe RDMA (remote direct memory access)
- Nodes communicate via network interface
- Example: most modern clusters!

Speed of light

- One light-ns is 30 cm (about one foot)
- A big machine is often over 300 feet across
- May still be dominated by NIC latency (microseconds)
- Across a big machine will always be much slower than local memory accesses
- Another reason locality matters!

Paths to performance

What do we want?

- High-level: solve bit problems fast
- Start with good *serial* performance
- Given p processors, could then ask for
  - Good *speedup*: serial time $/p$
  - Good *scaled speedup*: $p\times$ serial work in serial time
- Easiest to get speedup from bad serial code!

Story so far

Parallel performance is limited by:

- Single core performance
- Communication and synchronization costs
- Non-parallel work (Amdahl)

Overcome these limits by understanding common patterns of parallelism and locality in applications.

Parallelism and locality

Can get more parallelism / locality through modeling

- Limited range of dependency between time steps
- Neglect or approximate far-field effects

Parallelism and locality

Often get parallelism at multiple levels

- Hierarchical circuit simulation
- Interacting models for climate
- Parallelizing experiments in MC or optimization

Next week

More about parallelism and locality in simulations!