Lecture 4:
Introduction to parallel machines and models

David Bindel

3 Feb 2010

Logistics

- We have a wiki:
  `https://confluence.cornell.edu/display/cs5220s10/`
- I've been told that you will get accounts today (this morning?).
- The first assignment is ready. I've sent tentative groups; feel free to rearrange them.

Assignment strategy

Three hints:

- Automatically generate the innermost multiply kernel *and* microbenchmark it. Play with loop orders, sizes, etc. I suggest explicitly loading everything into local variables at the start, doing the computations with locals, and saving at the end.
- Play with compiler flags (makes a difference for inner kernel)
- Use multiple levels of blocking (register, L1, L2?)

Read! Get good ideas from anywhere – just attribute them.

Can also share ideas and refs on wiki.

A note on tools

Use your tools. The code in the tarball uses:

1. `gcc` (to build the code)
2. `gfortran` (to build the Fortran reference version)
3. ATLAS (for a fast BLAS for comparison)
4. `make` (to build the code)
5. Shell scripts (to set up SGE submissions)
6. `awk` (to post-process the output files)
7. `gnuplot` (to plot performance)

You could also use MATLAB for plotting, `perl` for auto-generating code, ...

Most high-performance scientific codes run under UNIX. It's worth knowing your way around.

A note for automatic generation

In C code:

```
...
for (i = 0; i < n; i += bs)
  for (j = 0; j < n; j += bs)
    for (k = 0; k < n; k += bs) {
      #include "my_block.c"
    }
```

and generate `my_block.c` by script...

A note on automatic generation

```
#!/usr/bin/perl

... print code to load A and B blocks ...
for ($i = 0; $i < $b; ++$i) {
  for ($j = 0; $j < $b; ++$j) {
    printf "double c%d%d = a%d%d*b%d%d",
           $i, $j, $i, 0, 0, $j;
    for ($k = 0; $k < $b; ++$k) {
      printf "+ a%d%d*b%d%d",
             $i, $k, $k, $j;
    }
    printf ";\n"
  }
}
... print results back to C ...
```

A note on automatic generation

In Makefile:

```
# Recreate my_block.c when my_block_gen.pl changes
my_block.c: my_block_gen.pl
        perl my_block_gen.pl > my_block.c

# my_dgemm.o depends on my_dgemm.c and my_block.c
my_dgemm.o: my_dgemm.c my_block.c

# Copy this from other examples...
matmul-me: $(OBJS) my_dgemm.o
        $(CC) -o $@ $^ $(LDFLAGS) $(LIBS)
```

Class cluster basics

`crocus.csuglab.cornell.edu` is a Linux Rocks cluster

- Six nodes (one head node, five compute nodes)
- Head node is virtual — *do not overload*!
- Compute nodes are dedicated — *be polite*!
- Batch submissions using Sun Grid Engine
- Read docs on wiki...

Class cluster basics

- Compute nodes are dual quad-core Intel Xeon E5504
- Nominal peak per core:
  2 SSE instruction/cycle $\times$
  2 flops/instruction $\times$
  2 GHz = 8 GFlop/s per core
- Caches:
  1. L1 is 32 KB, 4-way
  2. L2 is 256 KB (unshared) per core, 8-way
  3. L3 is 4 MB (shared), 16-way associative

  L1 is relatively slow, L2 is relatively fast.
- Inter-node communication is switched gigabit Ethernet
- 16 GB memory per node

Cluster structure

Consider:

- Each core has vector parallelism
- Each chip has four cores, shares memory with others
- Each box has two chips, shares memory
- Cluster has five compute 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 scientific codes have distinct 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 much 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: Quad-core chips on cluster

Multithreaded processor machine

- May have more threads than processors! Switch threads on long latency ops.
- Called *hyperthreading* by Intel
- Cray MTA was one 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)

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

Why clusters?

- Clusters of SMPs are everywhere
  - Commodity hardware – economics! Even supercomputers now use commodity CPUs (though specialized interconnects).
  - Relatively simple to set up and administer (?)
- But still costs room, power, ...
- Will grid/cloud computing take over next?