Performance Wrap-Up

Mark Greenstreet

CpSc 418 – Feb. 24, 2016

Outline:

- Finishing performance loss: non-parallelizable code, etc.
- Real code
- Modeling parallel performance

Objectives

- Learn about main causes of performance loss:
  - Overhead: covered in Feb. 22 lecture.
  - Non-parallelizable code
  - Idle processors
  - Resource contention
- See this with real-code
  - See some `pthreads` code.
  - Compare Erlang and C performance.
  - Learn about some more performance measuring tools.
- Models
  - Wrap-up details from Feb. 12 lecture.
  - Why we like λ.
  - When our simplified CTA model is not enough:
    - location, location, location

Non-parallelizable Code

- Finding the length of a linked list:

```
int length=0;
for(List p = listHead; p != null; p = p->next)
    length++;
```

  - Must dereference each `p->next` before it can dereference the next one.
  - Could make more parallel by using a different data structure to represent lists (some kind of skiplist, or tree, etc.)
- Searching a binary tree
  - Requires 2^k processes to get factor of k speed-up.
  - Not practical in most cases.
  - Again, could consider using another data structure.
- Interpretting a sequential program.
- Finite state machines.

Idle Processors

- There is work to do, but processors are idle.
- Start-up and completion costs.
- Work imbalance.
- Communication delays.

Resource Contention

.

- Processors waiting for a limited resource.
- It's easy to change a compute-bound task into an I/O bound one by using parallel programming.
- Or, we run-into memory bandwidth limitations:
  - Processing cache-misses.
  - Communication between CPUs and co-processors.
- Network bandwidth.

Some Real Code

- Count 3's in C (sequential)
  - The code
  - Timing
  - Compare with Erlang
- Count 3's in C (parallel)
  - `pthreads`
  - The four versions sketched in *Principles of Parallel Programming*

Modeling Performance

- PRAM: ignores communication cost – i.e. it ignores what really matters.
- bloP (a.k.a. logP)
  - simplified version of CTA
  - ignores location and network topology
  - has enough parameters that it worked for small number of machines and a small number of examples in 1993.
- CTA
  - We use a simplified version where λ indicates communication cost.
    - Roughly logP with fewer parameters.
    - Easier to work with, less susceptible to overfitting.
  - The full version includes the network (as a graph).

Why topology matters

- Not all communication costs are equal:
  - Communication between cores on the same chip is relatively fast.
  - Communication between cores on different chips on the same circuit board is slower.
  - Communication over a network is much slower.
- Example: 2D-neighbours vs. all-to-all communication.
  - Assume mesh topology.
  - What happens if every processor sends $N/4$ words to each of its neighbours?
  - What happens if every processor sends N/P words to each of the other processors a large, parallel supercomputer?

What about big messages?

Lecture Summary

- Performance Loss
  - Non-parallelizable code
  - Idle processors
  - Resource contention
- Real-Code: count3s with `pthreads`
  - `pthreads` code is **much** more verbose than Erlang.
  - But, it runs $\sim 4\times$ faster.
  - Demonstrated common parallel pitfalls: races, synchronization overhead, false sharing.
- Modeling:
  - CTA with a term for message length is a nice (adequate) model to get reasonable intuition.
  - Be aware of locality issues, especially for large machines.

Review Questions

- Describe non-parallelizable code and give an example?
- Describe how idle processors and synchronization lead to performance loss?
- Which is faster, Erlang or C? By about how much?
- Which is easier for writing parallel code, Erlang or C, why?
  - Is your answer objective of subjective?
  - Any other observations?
- Compare the PRAM, logP, and CTA models.

Preview

For Feb. 29, read: “The GPU Computing Era”,
http://dx.doi.org/10.1109/MM.2010.41.