Performance Losses

Mark Greenstreet

CpSc 418 – Sept. 18, 2012

Outline:

- Measuring Performance
- Count 3's performance

Parallel Programming and Performance

- The main motivation for parallel programming is performance
  - Time: make a program run faster.
  - Space: allow a program to run with more memory.
- To make a program run faster, we need to know how fast it is running.
- There are many possible measures:
  - Latency: time from starting a task until it completes.
  - Throughput: the rate at which tasks are completed.
  - Key observation:

$$\begin{aligned} \textit{throughput} &= \frac{1}{\textit{latency}}, \quad \text{sequential programming} \\ \textit{throughput} &\geq \frac{1}{\textit{latency}}, \quad \text{parallel programming} \end{aligned}$$

Speed-Up

- Simple definition:

$$speed - up = \frac{\text{time}(\text{sequential} - \text{execution})}{\text{time}(\text{parallel} - \text{execution})}$$

- But beware of the spin:
  - Is “time” latency or throughput?
  - How big is the problem?
  - What is the sequential version:
    - The parallel code run on one processor?
    - The fastest possible sequential implementation?
    - Something else?
- More practically, how do we measure time?

Time complexity

- What is the time complexity of sorting?
  - What are you counting?
  - Why do you care?
- What is the time complexity of matrix multiplication?
  - What are you counting?
  - Why do you care?

Big-O and Wall-Clock Time

- In our algorithms classes, we count "operations" because we have some belief that they have something to do with how long the actual program will take to execute.
  - Or maybe not. Some would argue that we count "operations" because it allows us to use nifty techniques from discrete math.
  - I'll take the position that the discrete math is nifty because it tells us something useful about what our software will do.
- In our architecture classes, we got the formula:

$$\text{time} \;=\; \frac{(\#\text{inst. executed}) * (\text{cycles}/\text{instruction})}{\text{clock frequency}}$$

- The approach in algorithms class of counting comparisons or multiplications, etc., is based on the idea that everything else is done in proportion to these operations.
- BUT, in parallel programming, we can find that a communication between processes can take 1000 times longer than a comparison or multiplication.
  - The may not matter if you're willing to ignore "constant factors."
  - In practice, factors of 1000 are too big to ignore.

Causes of Performance Loss

- Ideally, we would like a parallel program to run P times faster than the sequential version when run on P processors.
- In practice, this rarely happens because of:
  - Overhead: work that the parallel program has to do that isn't needed in the sequential program.
  - Non-parallelizable code: something that has to be done sequentially.
  - Idle processors: There's work to do, but some processor are waiting for something so before they can work on it.
  - Resource contention: Too many processors overloading a limited resource.

Communication Overhead

- In a parallel program, data must be sent between processors.
- This isn't a part of the sequential program.
- The time to send and receive data is overhead.
- Communication overhead occurs with both shared-memory and message passing machines and programs.

Communication with shared-memory

- In a shared memory architecture:
  - Each core has it's own cache.
  - The caches communicate to make sure that all references from different cores to the same address look like their is one, common memory.
  - It takes longer to access data from a remote cache than from the local cache. This creates overhead.
- False sharing can create communication overhead even when there is no logical sharing of data.
  - This occurs if two processors repeatedly modify different locations on the same cache line.

Communication overhead with message passing

- The time to transmit the message through the network.
- There is also a CPU overhead: the time set up the transmission and the time to receive the message.
- The context switches between the parallel application and the operating system adds even more time.
- Note that many of these overheads can be reduced if the sender and receiver are different threads of the same process running on the same CPU.
  - This has led to SMP implementations of Erlang, MPI, and other message passing parallel programming frameworks.
  - The overheads for message passing on an SMP can be very close to those of a program that explicitly uses shared memory.
  - This allows the programmer to have one parallel programming model for both threads on a multi-core processor and for multiple processes on different machines in a cluster.

Synchronization Overhead

- Parallel processes must coordinate their operations.
  - Example: access to shared data structures.
  - Example: writing to a file.
- For shared-memory programs (e.g. `pthreads` or `Java threads`, there are explicit locks or other synchronization mechanisms.
- For message passing (e.g. `Erlang` or `MPI`), synchronization is accomplished by communication.

Computation Overhead

- Computation: a parallel program may perform computation that is not done by the sequential program.
  - Redundant computation: it's faster to recompute the same thing on each processor than to broadcast.
  - Algorithm: sometimes the fastest parallel algorithm is fundamentally different than the fastest sequential one, and the parallel one performs more operations.
- Memory: The total memory needed for P processes may be greater than that needed by one process due to replicated data structures and code.

Sieve or Eratosthenes

To find all primes ≤ N:

```
1.    Let MightBePrime = [2, 3, ..., N].
2.    Let KnownPrimes = [].
3.    while(MightBePrime ≠ []) do
        % Loop invariant: KnownPrimes contains all primes less than the
        % smallest element of MightBePrime, and MightBePrime
        % is in ascending order. This ensure that the first element of
        % MightBePrime is prime.
3.1.    Let P = first element of MightBePrime.
3.2.    Append P to KnownPrimes.
3.3.    Delete all multiples of P from MightBePrime.
4.    end
```

See http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Prime-Sieve in Erlang

```
% primes(N): return a list of all primes ≤ N.
primes(N) when is_integer(N) and (N < 2) -> [];
primes(N) when is_integer(N) ->
   do_primes([], lists:seq(2, N)).

% invariants of do_primes(Known, Maybe):
%    All elements of Known are prime.
%    No element of Maybe is divisible by any element of Known.
%    lists:reverse(Known) ++ Maybe is an ascending list.
%    Known ++ Maybe contains all primes ≤ N, where N is from p(N).
do_primes(KnownPrimes, []) -> lists:reverse(KnownPrimes);
do_primes(KnownPrimes, [P | Etc]) ->
do_primes([P | KnownPrimes],
          lists:filter(fun(E) -> (E rem P) /= 0 end, Etc)).
```

A More Efficient Sieve

- If N is composite, then it has at least one prime factor that is at most $\sqrt{N}$.
- This means that once we've found a prime that is $\geq \sqrt{N}$, all remaining elements of `Maybe` must be prime.
- Revised code:

```
% primes(N): return a list of all primes ≤ N.
primes(N) when is_integer(N) and (N < 2) -> [];
primes(N) when is_integer(N) ->
   do_primes([], lists:seq(2, N), trunc(math:sqrt(N))).

do_primes(KnownPrimes, [P | Etc], RootN)
         when (P =< RootN) ->
   do_primes([P | KnownPrimes],
      lists:filter(fun(E) -> (E rem P) /= 0 end, Etc), RootN);
   do_primes(KnownPrimes, Maybe, _RootN) ->
      lists:reverse(KnownPrimes, Maybe).
```

- If you prefer Java or C, see slide 29.

Prime-Sieve: Parallel Version

- Main idea
  - Find primes from $1 \ldots \sqrt{N}$.
  - Divide $\sqrt{N} + 1 \ldots N$ evenly between processors.
  - Have each processor find primes in its interval.
- We can speed up this program by having each processor compute the primes from $1 \ldots \sqrt{N}$?
  - Why does doing extra computation make the code faster?

Overhead: Summary

Overhead is loss of performance due to extra work that the parallel program does that is not performed by the seqential version. This includes:

- Communication: parallel processes need to exchange data. A sequential program only has one process; so it doesn't have this overhead.
- Synchronization: Parallel processes may need to synchronize to guarantee that some operations (e.g. file writes) are performed in a particular order. For a sequential program, this ordering is provided by the program itself.
- Extra Computation:
  - Sometimes it is more efficient to repeat a computation in several different processes to avoid communication overhead.
  - Sometimes the best parallel algorithm is a different algorithm than the sequential version and the parallel one performs more operations.
- Extra Memory: Data structures may be replicated in several different processes.

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.

Amdahl's Law

- Given a sequential program where
  - fraction s of the execution time is inherently sequential.
  - fraction $1 - s$ of the execution time benefits perfectly from speed-up.
- The run-time on P processors is:

$$T_{parallel} = T_{sequential} * (s + \frac{1-s}{P})$$

- Consequences:
  - Define

$$speed - up = \frac{T_{sequential}}{T_{parallel}}$$

  - Speed-up on P processors is at most $\frac{1}{s}$.
  - Gene Amdahl argued in 1967 that this limit means that parallel computers are only useful for a few special applications where s is very small.

Amdahl's Law, 45 years later

- Amdahl's law is an economic law, not a physical law.
  - Amdahl's law was formulated when CPUs were expensive.
  - Today, CPUs are cheap
    - The cost of fabricating eight cores on a die is very little more that the cost of fabricating one.
    - Computer cost is dominated by the rest of the system: memory, disk, network, monitor, ...
- Amdahl's law assumes a fixed problem size ...

Amdahl's Law, 44 years later

- Amdahl's law is an economic law, not a physical law.
  - Amdahl's law was formulated when CPUs were expensive.
  - Today, CPUs are cheap (see previous slide)
- Amdahl's law assumes a fixed problem size
  - Many computations have s (sequential fraction) that decreases as N (problem size) increases.
  - Having lots of cheap CPUs available will
    - Change our ideas of what computations are easy and which are hard.
    - Determine what the "killer-apps" will be in the next ten years.
      - Ten years from now, people will just take it for granted that most new computer applications will be parallel.
  - Examples:
    - Managing/searching/mining massive data sets.
    - Scientific computation.
      - Note that most of the computation for animation and rendering resembles scientific computation. Computer games benefit tremendously from parallelism.
      - Likewise for multimedia computing.

Software is Expensive

- On the previous slide, I noted that CPUs are essentially free.
  - But programming them isn't.
- Hardware is already free.
  - Software is the problem.
- The challenge in exploiting parallelism is a software problem.
  - We need to understand the architectural issues so we can develop programming abstractions that match performance reality.

Overhead: Idle CPUs

There are idle processors and work to do, but the processors can't do the work, because:

- Load imbalance:
  - A few processors get tasks that take longer than the others.
  - This is especially a problem if it's hard to determine how long a task will take without running it.
- Start-up and ending costs
  - Some problems start with one process that spawns tasks for other processors to execute.
  - Initially, the other processors are idle, waiting for the first processor to spawn tasks.
  - A similar problem can occur collecting results at the end.

Contention

Multiple processors need the same resource.

- Disk access.
- Main memory access with a SMP.
- Network access with a cluster.

On a really good day, you win

- Embarrassingly parallel applications
  - Problems that can run nearly independently on a large number of processors.
  - Monte Carlo simulations, ray tracing, factoring huge numbers, ...
- Superlinear speed-up
  - Occasionally, a parallel program with P processors is more than P times faster than the sequential version.
    - ⋆ More, fast memory:
      multiple CPUs have more total registers, more cache memory, more I/O bandwidth, etc.
    - ⋆ A different algorithm:
      - The natural parallel algorithm may visit a data structure in a different order than the sequential algorithm.
      - This can, for example, result in faster pruning for a search for some applications.
      - If the sequential version is modified to do the same thing, it may be too complicated, resulting in sequential overhead.

Lecture Summary

Causes of Performance Loss in Parallel Programs

- Overhead
  - Communication, slide 7.
  - Synchronization, slide 10.
  - Computation, slide 11.
  - Extra Memory
- Other sources of performance loss
  - Non-parallelizable code, slide 17
  - Idle Processors, slide 22.
  - Resource Contention, slide 23.
- Quantifying speed-up, slide 3
  - Amdahl's Law, slide 19.
  - Super-Linear Speed-up, slide 24
    and "embarrassingly parallel" applications.

Supplementary Material

- The `time_it` module.
- The sieve of Eratosthenes in Java/C.

The `time_it` module

- I wrote some erlang functions for measuring the time it takes a function to execute.
- These functions are available at
  http://www.ugrad.cs.ubc.ca/~cs418/2012-1/src/erl/source.html
- Most of what you need:
  - `time_it:t(Fun, N)`, for integer `N` returns the mean and standard deviation of the execution time for `N` trials of executing `Fun()`.
  - `time_it:t(Fun, T)`, for floating point number T returns the mean and standard deviation by repeatedly executing `Fun()` until a total of `T` seconds have elapsed.
  - `time_it:t(Fun)`, equivalent to `time_it:t(Fun, 1.0)`.

time_it Example

```
1> R1K = lists:map(fun(_) -> random:uniform() end,
                   lists:seq(1, 1000)), ok.
ok
2> code:add_path("/home/c/cs418/public_html/src/erl").
true
3> time_it:t(fun() -> hw2:max_hr(R1K) end).
[{mean,3.553738450603681e-5},{std,6.529345227998487e-6}]
```

Prime-Sieve: Java/C version

```
% Sieve of Eratosthenes
int primes[N];
primes[0] = 0; primes[1] = 0;
for(int i = 2; i < N; i++)
   primes[i] = 1;      % assumed prime until proven composite
int lastp = 1;         % look for primes starting at lastp+1
int top = sqrt(N);     % any composite ≤ N has a factor ≤ top
while(lastp < top) {
   int p;              % next line sets p to next prime
   for(p = lastp+1; (p < N) && (primes[p] == 0); p++);
   for(c = 2*p; c < N; c += p)
      primes[c] = 0;   % c is a multiple of p, hence composite
   lastp = p;
}
% that's it!
```