

Performance-Loss

Mark Greenstreet

CpSc 418 – Feb. 22, 2016

Outline:

- Measuring Performance
- Count 3's performance

Objectives

- Learn about main causes of performance loss:
 - ▶ Overhead
 - ▶ Non-parallelizable code
 - ▶ Idle processors
 - ▶ Resource contention
- See how these arise in message-passing, and shared-memory code.
- As a bonus: see a bit of `pthread`s programming.

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 before they can work on it.
 - ▶ **Resource contention**: Too many processors overloading a limited resource.

Overhead

Overhead: work that the parallel program has to do that isn't needed in the sequential program.

- Communication:

- ▶ The processes (or threads) of a parallel program need to **communicate**.
- ▶ A sequential program has no interprocess communication.

- Synchronization.

- ▶ The processes (or threads) of a parallel program need to **coordinate**.
- ▶ This can be to avoid interference, or to ensure that a result is ready before it's used, etc.
- ▶ Sequential programs have a completely specified order of execution: no synchronization needed.

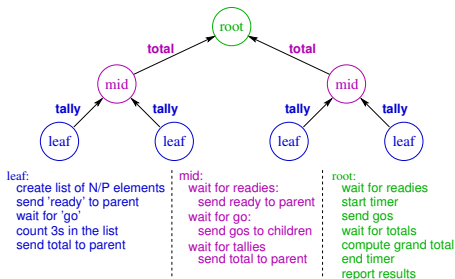
- Computation.

- ▶ Recomputing a result is often cheaper than sending it.

- Memory Overhead.

- ▶ Each process may have its own copy of a data structure.

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.
- Example: Reduce (e.g. Count 3s):
 - ▶ Communication between processes adds time to execution.
 - ▶ The sequential program doesn't have this overhead.

Communication with shared-memory

- In a shared memory architecture:
 - ▶ Each core has its own cache.
 - ▶ The caches communicate to make sure that all references from different cores to the same address look like there 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: example

- The *Principles of Parallel Programming* book considered an example of Count 3s (in C, with threads), where there was a global array, `int count[P]` where `P` is the number of threads.
 - ▶ Each thread (e.g. thread `i`) initially sets its count, `count[i]` to 0.
 - ▶ Each time a thread encounters a 3, it increments its element in the array.
- The parallel version ran much slower than the sequential one.
 - ▶ Cache lines are much bigger than a single `int`. Thus, many entries for the `count` array are on the same cache line.
 - ▶ A processor has to get exclusive access to update the count for its thread.
 - ▶ This invalidates the copies held by the other processors.
 - ▶ This produces lots of cache misses and a slow execution.
- A better solution:
 - ▶ Each thread has a local variable for its count.
 - ▶ Each thread counts its threes using this local variable and copies its final total to the entry in the global array.

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.

Communication overhead: an example

It's hard to measure the communication overhead for Count 3s in Erlang.

- Each process sends and receives 2–6 messages.
- The thread scheduler **avoids parallel execution!**
 - ▶ It assumes that if you have multiple threads, they are GUI event handlers or similar, and that you probably aren't really trying to make your code parallel.
 - ▶ It waits until multiple threads have been runnable for up to a few milliseconds before using multiple cores.
 - ▶ I'm pretty sure this scheduling policy is part of linux/OSX/Windows.
 - ▶ I should write the `ptreads` code to measure this.
- For count 3s, we just see the scheduler overhead, not the communication time.

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. `pthread`s 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

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.

Sieve or Eratosthenes

To find all primes $\leq N$:

```
1.  Let MightBePrime = [2, 3, ..., N].
2.  Let KnownPrimes = [].
3.  while(MightBePrime  $\neq$  []) 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  $\leq N$ .
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  $\leq N$ , 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  $\leq N$ .
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).
```

Prime-Sieve: Parallel Version

- Main idea
 - ▶ Find primes from $1 \dots \sqrt{N}$.
 - ▶ Divide $\sqrt{N} + 1 \dots N$ 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 $1 \dots \sqrt{N}$.
 - ▶ Why does doing extra computation make the code faster?

Memory Overhead

The total memory needed for P processes may be greater than that needed by one process due to replicated data structures and code.

- Example: the parallel sieve: each process had its own copy of the first \sqrt{N} primes.

Overhead: Summary

Overhead is loss of performance due to extra work that the parallel program does that is not performed by the sequential 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.
- Interpreting 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.

Lecture Summary

Causes of Performance Loss in Parallel Programs

- Overhead
 - ▶ Communication, [slide 5](#).
 - ▶ Synchronization, [slide 10](#).
 - ▶ Computation, [slide 11](#).
 - ▶ Extra Memory, [slide 16](#).
- Other sources of performance loss
 - ▶ Non-parallelizable code, [slide 18](#)
 - ▶ Idle Processors, [slide 19](#).
 - ▶ Resource Contention, [slide 20](#).

Review Questions

- What is overhead? Give several examples of how a parallel program may need to do more work or use more memory than a sequential program.
- Do programs running on a shared-memory computer have communication overhead? Why or why not?
- Do message passing program have synchronization overhead? Why or why not?
- Why might a parallel program have idle processes even when there is work to be done?