Performance Loss

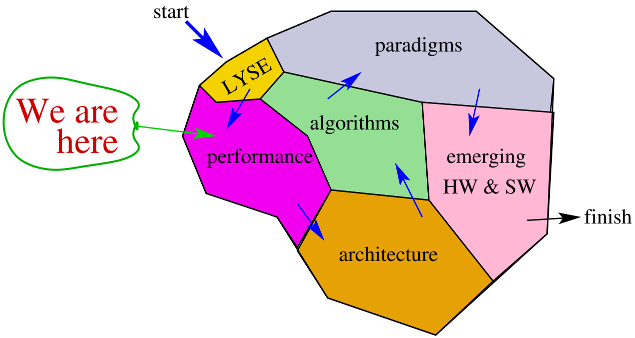
Mark Greenstreet

CpSc 418 – Sept. 17, 2013

Outline:

- Measuring Performance
- Count 3's performance

Where are we?



Parallelandia

- We've completed a quick introduction of Erlang programming.
- Now, we will look at performance:
  - How to measure it.
  - Major performance limiters and how to identify them.

Objectives

- Understand key measures of performance
  - Time: latency vs. throughput
  - Time: wall-clock vs. operation count
  - Speed-up: slide 5
- Understand key performance losses for parallel programs:
  - Overhead: Extra “work” in the parallel version
  - Inherently sequential code: Amdahl’s law
  - Idle processors
  - Resource contention

Measuring 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:

$$\textit{speed\_up} \quad = \quad \frac{\text{time(sequential\_execution)}}{\text{time(parallel\_execution)}}$$

- 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:

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

- 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.
  - This 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

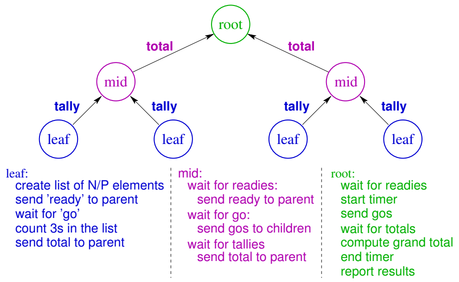
- Ideally, we would like a parallel program to run *P* times faster than the sequential version when run on *P* processors.
- In practice, this rarely happens because of:
  - Overhead: work that the parallel program has to do that isn't needed in the sequential program.
  - Non-parallelizable code: something 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.

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: 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 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.

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 runable 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 `pthreads` code to measure this.
- For count 3s, we just see the scheduler overhead, not the communication time.

Erlang process start-up delays (experiment design)

- `main(P, N)` spawns `P` processes, each of which does $O(\texttt{N})$ work.
- Processes send a value back to main.
- Use `time_it:log(FormatString, Args)` to log key events.
- Print the event log at the end.

Erlang process start-up delays (the code)

```
main(N, P) ->
Log1 = time_it:log("main:  starting"),
{_W, Log2} = start_workers(N, P),
{_V, Log3} = get_results( P),
Log4 = get_logs(P),
time_it:print_log([Log1, Log2, Log3, Log4]).

worker(Ppid, N, MyIndex) ->
Log1 = time_it:log("worker ~b:  starting", [MyIndex]),
V = busy_work(N),
Log2 = time_it:log("worker ~b:  result ready to send",
                   [MyIndex]),
Ppid ! {result, MyIndex, V},
Log3 = time_it:log("worker ~b:  done", [MyIndex]),
Ppid ! {event_log, MyIndex, [Log1, Log2, Log3]}.
```

For the complete code, see

http:www.ugrad.cs.ubc.ca/~cs418/2013-1/lecture/09.17/src/msg1.erl

Erlang process start-up delays (execution)

```
> msg1:main(4, 100000).
            0 <0.35.0>: main: starting
  1.10000e-5 <0.35.0>: main: just spawned worker 4
  1.40000e-5 <0.35.0>: main: just spawned worker 3
  1.70000e-5 <0.35.0>: main: just spawned worker 2
  2.00000e-5 <0.35.0>: main: just spawned worker 1
  2.80000e-5 <0.10174.0>: worker 4: starting
  3.28000e-4 <0.10175.0>: worker 3: starting
  6.28000e-4 <0.10176.0>: worker 2: starting
  9.29000e-4 <0.10177.0>: worker 1: starting
  1.77940e-2 <0.10174.0>: worker 4: result ready to send
  1.77990e-2 <0.10174.0>: worker 4: done
  1.78080e-2 <0.35.0>: main: received result from worker 4
  1.94000e-2 <0.10175.0>: worker 3: result ready to send
  ...
```

Observe long time for process start-up:

- No process starts until after `main` has spawned all children
- About 0.3ms delay between starting consecutive processes.

Measure message passing overhead (experiment design)

- `main(P, N_work, N_rounds)` spawns `P` worker processes:
  - Each process performs `N_rounds` of:
    - Perform $O(N\_work)$ work.
    - Send a message to another worker processes.
  - Our code ensures:
    - Each process receives one message per round;
    - No process sends a message to itself;
    - Different permutations for different rounds.
- Measure the time for different values for `P`, `N_work` and `N_rounds`
- For example, by comparing two runs where each process does the same total amount of local work but sends and receives different numbers of messages, we can determine the time per message.
- Code: next slide. Or download from:

  http:www.ugrad.cs.ubc.ca/~cs418/2013-1/lecture/09.17/src/msg2.erl

Erlang message passing overhead (the code)

```
main(P, N_work, N_rounds) ->
   MyPid = self(),
   One_to_P = lists:seq(1, P),
   I_dst = make_dst_lists(P, 17),
   Args = [[MyPid, I, N_work, N_rounds] || I <- One_to_P],
   {W, _Log} = start_workers(Args),
   P_dst = [ [ lists:nth(I, W) || I <- D ] || D <- I_dst],
   T0 = now(),
   [    CPid ! {busy_busy_busy, DstList}
     || {CPid, DstList} <- lists:zip(W, P_dst) ],
   {_V, _Log2} = get_results(P),
   T1 = now(),
   io:format("elapsed time = ~12.6e~n",
             [1.0e-6*timer:now_diff(T1, T0)]).

worker(Ppid, MyIndex, N_work, N_rounds) ->
   receive {busy_busy_busy, DstList} -> ok end,
   V = busy_work(N_work, N_rounds, DstList),
   Ppid ! {result, MyIndex, V}.
```

Erlang message passing overhead (the results)

`P`	`N_work`	`N_rounds`	elapsed time
4	100	1000	28.9ms
4	50	2000	30.7ms
8	100	1000	38.5ms
8	50	2000	41.2ms
48	100	1000	131.8ms
48	50	2000	144.9ms

Examples run on quad-core, two-way multithreaded, 2GHz, Intel Core i7 processor.
Each value is median from five runs.

- 4 processes: time/(message/core) $= 1.80\mu$s.
- 8 processes: time/(message/core) $= 1.35\mu$s.
- 48 processes: time/(message/core) $= 1.18\mu$s.

Messages take $1 - 2\mu$s. Time decreases with more processes!

Erlang message passing overhead (remarks)

The total number of messages sent (and received) when running the code with `P` processes and `N_rounds` rounds is `P` $\cdot$ `N_rounds`. The total amount of sequential work is `P` $\cdot$ `N_roundsN_work`. Thus, if we have two runs where:

run 1 has P processes, each performing `N_rounds` of “work” with `N_work` per round; and

run 2 has P processes, each performing 2`N_rounds` of “work” with $\frac{1}{2}$`N_work`,

then run 1 and run 2 do the same amount of sequential work, but run 2 sends and receives `P` $\cdot$ `N_rounds` more messages than run 1. Let T_1 and T_2 denote the times for executing run 1 and run 2 respectively.
We ran this code on a machine with Q cores (where $Q = 4$ for the results reported on the previous slide). This means that each core handles `P` $\cdot$ `N_rounds`$/Q$ more messages for run 2 than it did for run 1 – where “handling a message” is the total of the send and receive time. To get the time that it takes a core to handle one message, I divided the time difference by the number of messages per core:

$$T_{msg} = \frac{Q(T_2 - T_1)}{\texttt{P} \cdot \texttt{N\_rounds}}$$

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.

Need a lock (code)

```
main() ->
   MyPid = self(),
   C1 = spawn(fun() -> spin_then_print(MyPid, poem1(), 1) end),
   C2 = spawn(fun() -> spin_then_print(MyPid, poem2(), 1) end),
   msg1:busy_work(random:uniform(8000)),
   C1 ! (C2 ! go),
   receive done -> receive done -> ok end end,

spin_then_print(PPid, Lock, Poetry, V) ->
   receive
      go -> print_poem(Lock, Poetry), PPid ! done
      after 0 ->
        spin_then_print(PPid, Lock, Poetry, math:cos(V) + V)
   end.

print_poem([]) -> ok;
print_poem([Hd | Tl]) -> io:format(Hd), print_poem(Tl).
```

For the complete code, see

http:www.ugrad.cs.ubc.ca/~cs418/2013-1/lecture/09.17/src/lock.erl

Need a lock (execution)

```
Mary had a little lamb.
Twinkle, Twinkle, little star,
It's fleece was white as snow.
How I wonder what you are.
And everywhere that Mary went,
Up above the world so high,
Her lamb was sure to go.
Like a diamond in the sky.

Twinkle, Twinkle, little star,
How I wonder what you are.

ok
```

Implementing a lock in Erlang

```
lock_create() -> spawn(fun() -> unlocked() end).

unlocked() ->
   receive
      {lock, Pid} ->
         Pid ! {locked, self()},
         locked(Pid);
      {unlock, _} -> error("bad unlock: not locked");
      exit -> ok
   end.

locked(Pid) ->
   receive
      {unlock, Pid} -> unlocked();
      {unlock, Imposter} ->
         error(lists:flatten(io_lib:format(
            "bad unlock: attempted by ~w; lock held by ~w",
            [Imposter, Pid])))
   end.
```

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 ≠ []) 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).
```

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?

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 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.
- Finite state machines.

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:

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

- Consequences:
  - Define

$$speed\_up \quad = \quad \frac{T_{sequential}}{T_{parallel}}$$

  - 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, 46 years later

Amdahl's law is not a physical law.

- Amdahl's law is mathematical theorem:
  - If $T_{parallel}$ is $\left(s + \frac{1-s}{P}\right) T_{sequential}$
  - and $speed\_up = T_{sequential} / T_{parallel}$,
  - then for $0 < s \leq 1$, $speed\_up \leq \frac{1}{s}$.
- Amdahl's law is also an economic 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, 46 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: see next slide

Amdahl's Law, 46 years later

- Amdahl's law is an economic law, not a physical law.
- Amdahl's law assumes a fixed problem size
  - 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.

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.

Lecture Summary

Causes of Performance Loss in Parallel Programs

- Overhead
  - Communication, slide 10.
  - Synchronization, slide 21.
  - Computation, slide 25.
  - Extra Memory, slide 30.
- Other sources of performance loss
  - Non-parallelizable code, slide 32
  - Idle Processors, slide 37.
  - Resource Contention, slide 38.
- Quantifying speed-up, slide 5
  - Throughput vs. Latency.
  - $speed\_up = \frac{T_{sequential}}{T_{parallel}}$
  - Amdahl's Law, slide 34.

Review Questions

- What is speed-up? Give an intuitive, English answer and a mathematical formula.
- What is Amdahl's law? Give a mathematical formula. Why is Amdahl's law a concern when developing parallel applications? Why in many cases is it not a show-stopper?
- 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?

Preview

September 19: Performance Measurement

Homework:	Homework 2 goes out – parallel programming with Erlang
Reading:	Lin & Snyder, chapter 3, pp. 68–77
Homework:	September 23, Homework 1 deadline for early-bird bonus

September 24: Matrix Multiplication

Reading:	Lin & Snyder, chapter 3, pp. 77–85
Homework:	Homework 1 due

September 26: Superscalars and compilers

Reading:	The MIPS R10000 Superscalar Microprocessor (Yeager)

October 1: Shared Memory Multiprocessors

Reading:	Lin & Snyder, chapter 2, pp. 30–43.
Homework:	Homework 3 goes out

October 3: Message Passing Multiprocessors

October 8: Models of Parallel Computation

Reading:	Lin & Snyder, chapter 2, pp. 43–59.

October 10: Peril-L, Reduce, and Scan

Reading:	Lin & Snyder, chapter 3, pp. 87–97.