# Performance Loss

Mark Greenstreet

CpSc 418 – October 5, 2018

# Outline

- Overhead: work the parallel code has to do that the sequential version avoids.
  - Communication and Synchronization
  - Extra computation, extra memory
- Limited parallelism
  - Code that is inherently sequential or has limited parallelism
  - Idle processors
  - Resource contention
- Related topics
  - Super-linear speed-up
  - Embarrassingly Parallel Problems
  - Brent's Lemma

# 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.

# Causes of Performance Loss

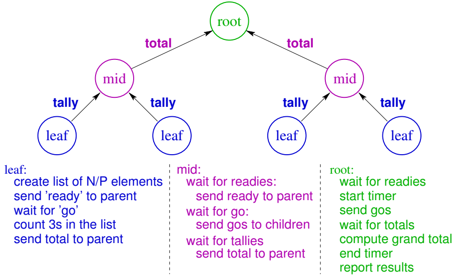
- Ideally, we would like a parallel program to run *P* times faster than the sequential version when run on *P* processors.
- In practice, this rarely happens because of:
  - Overhead: work that the parallel program has to do that isn't needed in the sequential program.
  - Non-parallelizable code: something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

# 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

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 of 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 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.

# Limited Parallelism

Sometimes, we can't keep all of the processors busy doing useful work.

- Non-parallelizable code
  The dependency graph for operations is narrow and deep.
- Idle processors
  There is work to do, but it hasn't been assigned to an idle processor.
- Resource contention
  Several processes need exclusive access to the same resource.

# 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.

# Super-Linear Speedup

Sometimes you win – $SpeedUp > P$. ☺

- But if that is true, wouldn't the best sequential algorithm be to simulate $P$ workers by time-sharing a single processor?
  - Probably not: Time-sharing has overhead.
- Memory: a common explanation
  - $P$ machines have more main memory (DRAM)
  - and more cache memory and registers (total)
  - and more I/O bandwidth, ...
- Multi-threading: another common explanation
  - The sequential algorithm cannot full utilize each CPU's parallel capabilities.
  - A parallel algorithm can make better use through, for example, latency hiding.
- Algorithmic advantages: Some problems are naturally parallel.

BUT: be skeptical, especially if $SpeedUp \gg P$.

# Embarrassingly Parallel Problems

Problems that can be solved by a large number of processors with very little communication or coordination.

- Rendering images for computer-animation: each frame is independent of all the others.
- Brute-force searches for cryptography.
- Analyzing large collections of images: astronomy surveys, facial recognition, ...
- Monte-Carlo simulations: same model, run with different random values.
- **Don't be ashamed if your code is embarrassingly parallel:**
  - Embarrassingly parallel problems are great: you can get excellent performance without heroic efforts.
  - The only thing to be embarrassed about is if you don't take advantage of easy parallelism when it is available.

# The Work-Span Model

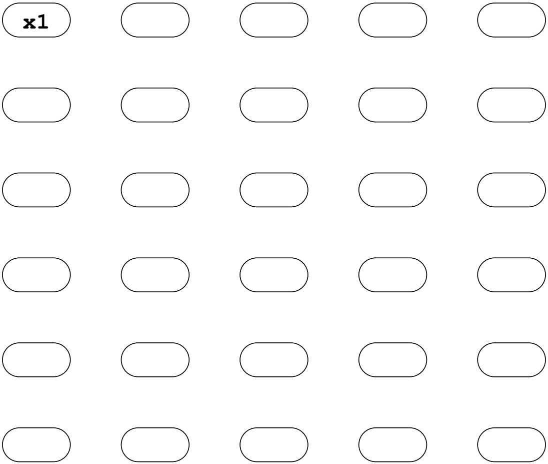
- Model computation using a directed, acyclic, graph (DAG)
  - Vertices correspond to operations
  - Edges represent dependencies.
    - If there is an edge from $Op_i$ to $Op_j$,
    - then, operation $Op_i$ must be performed before $Op_j$.
  - A vertex with no incoming edge(s) is an initial vertex.
    - The operation for an initial vertex can be done without waiting for any other operation.
    - The first operation(s) performed in any execution must correspond to an initial vertex.
  - A vertex with no outgoing edge(s) is a final vertex.
    - The first operation(s) performed in any execution must correspond to a final vertex.
- Work and Span
  - Work: the total number of vertices in the DAG.
    - Work represents the **sequential** execution time.
  - Span: the longest path from an initial vertex to a final vertex.
    - Span represents the ideal **parallel** execution time with an unlimited number of processors.

# The Work-Span Example

$$\{r_1, r_2\} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Ops:

```
  x1 ← b*b
  x2 ← 4*a
  x3 ← x2*c
  x4 ← x1 - x3
  x5 ← sqrt(x4)
  x6 ← -b
  x7 ← x6 + x5
  x8 ← 2*a
  r1 ← x7/x8
  x9 ← x6 - x5
  r2 ← x9/x8
```




Fill in the figure.

Work = ______

Span = ______

$SpeedUp_{\infty}$ = ______

# Brent's Lemma

- Let $T_P$ denote the time to evaluate a task with $P$ processors.
  - $T_1 = SequentialTime = Work$.
  - $T_\infty = UnlimitedParallelismTime = Span$.
- Then, $T_P \leq \frac{T_1}{P} + T_\infty$.
  - Corollary for speed-up:

$$SpeedUp_P = \frac{T_1}{T_P} \geq \frac{T_1}{\frac{T_1}{P} + T_\infty} = \frac{P}{1 + \frac{T_\infty}{T_1} P}$$

- Why Brent's Lemma is awesome:
  - Brent's lemma provides an upper bound on time and thus a lower bound on speed-up.
  - By using the work-span graph, Brent's lemma accounts for computations that have limited parallelism, even if they are not purely sequential.

# Brent's Lemma: Proof

- Construct the work-span DAG.
  - Arrange the graph to have *Span* levels, where vertices in each level have all of their incoming edges from earlier levels.
- Time to execute on $P$ processors:
  - Execute levels one at a time.
  - Let $W_i$ be the number of vertices at level $i$.

$$
\begin{array}{rclcl}
T_p & \leq & \sum_{i=1}^{Span} \left\lceil \frac{W_i}{P} \right\rceil & \leq & \sum_{i=1}^{Span} \frac{W_i}{P} + 1 \\
 & = & \left( \sum_{i=1}^{Span} \frac{W_i}{P} \right) + Span & = & \frac{Work}{P} + Span \\
 & = & \frac{T_1}{P} + T_\infty & &
\end{array}
$$

  - This is an **upper bound** for $T_p$ because we ignored the possiblity that some processors might be able to execute tasks in level $i+1$ while other processors are completing the last tasks in level $i$.

# Lecture Summary

Causes of Performance Loss in Parallel Programs

- Overhead
  - Communication, slide 6.
  - Synchronization, slide 10.
  - Computation, slide 11.
  - Extra Memory, slide 16.
- Other sources of performance loss
  - Non-parallelizable code, slide 19
  - Idle Processors, slide 20.
  - Resource Contention, slide 21.
- Finishing up related topics
  - Super-linear speed-up, slide 22
  - Embarrassingly Parallel Problems, slide 23.
  - Brent's Lemma, slide 26

# Preview

**October 7:** HW 2 earlybird (11:59pm).

**October 9:** HW 2 due (11:59pm).

**October 10: Parallel Performance: Models**

Homework: HW3 released.

**October 12: Energy, Power, and Time**

**October 15: Sorting Networks**

**October 17: The 0-1 Principle**

**October 18:** HW 3 earlybird (11:59pm).

**October 19: Midterm Review**

Homework: HW3 due: **12 noon**.

**October 22: Midterm**

**October 24-26: Sorting (second half)**

**October 29-November 30: Data Parallelism with CUDA**

# Review Questions (1 of 2)

- 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?
- What is super-linear speed-up?
  - Give two common causes for super-linear speed-up.
  - Is it likely to have speed-up that grows as $O(P \log P)$ or faster?
- What is an embarrassingly parallel problem? Give an example?

# Review Questions (1 of 2)

- What is the work-span model?
- Sketch the work-span DAG for $2 \times 2$ matrix multiplication.
- For $N \times N$ matrix multiplication, how does work grow as a function of $N$? For simplicity, you can assume that we use the simple, brute-force algorithm.
- For $N \times N$ matrix multiplication, how does span grow as a function of $N$? You can make the same assumptions as for work.
- What is $SpeedUp_\infty$ for $N \times N$ matrix multiplication?
- Use Brent's Lemma to derive a lower bound for speed up for $N \times N$ matrix multiplication when the number of processors, $P$ is at most $\sqrt{N}$.