Causes of Performance Loss in Parallel Computing

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – January 29, 2018

Table of Contents

1. Parallel Overhead
   - Communication
   - Sychronization
   - Computation
   - Memory
2. Limited Parallelism
   - Serial Dependency
   - Idle Processors
   - Resource Contention

Objectives

At the end of this lecture, you should be able to:

- Describe the main causes of performance loss when parallelizing algorithms.
- Explain how these losses arise in both message passing and shared memory architectures.

Causes of Performance Loss: Overview

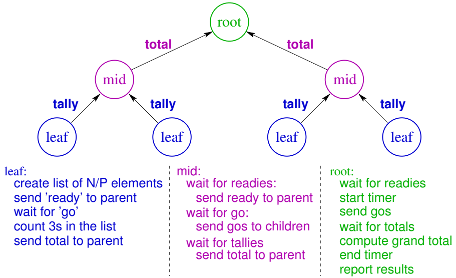
- Ideally, we would like a parallel program to run P times faster than the sequential version when run on P processors.
- In practice, this rarely happens because of:
  - Overhead: Work that the parallel program has to do that is not needed in the sequential program.
  - Limited Parallelism: Not every processor can be kept (usefully) busy all the time.

Causes of Performance Loss: 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.
- Synchronization:
  - The processes (or threads) of a parallel program need to coordinate.
  - This can be to avoid interference, to ensure that a result is ready before it is used, etc.
- Computation:
  - Recomputing a result is often cheaper than communicating 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.
- The time to send and receive data is overhead.
- Communication overhead occurs with both shared-memory and message passing machines and programs.
- Example: Reduce tree (e.g. Count 3s).
- Example: MRR figure 2.13
  - MRR appears to have a narrower definition of "overhead."

Communication overhead (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 / memory than from the local cache / memory.
- False sharing can create communication overhead even when there is no logical sharing of data.
  - False sharing occurs if two processors repeatedly modify different locations on the same cache line.
  - Example: Reduce operation where leaf results are all computed in different elements of a global array.

Communication overhead (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.
  - Desired optimization for common “symmetric multiprocessor” (SMP) hardware.
  - There are implementations of Erlang, MPI, and other message passing parallel programming frameworks tuned for SMPs.
  - The overheads for message passing on an SMP can be very close to those of a program that explicitly uses shared memory.
  - 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.
  - Example: avoiding race conditions (MRR section 2.6.1).
- For shared-memory programs (such as `pthreads` or `Java threads`) there are explicit locks or other synchronization mechanisms.
  - Example: Mutexes / locks (MRR section 2.6.2) leading to strangled scaling (MRR 2.6.4).
- For message passing (such as `Erlang` or `MPI`), synchronization is accomplished by communication.

Synchronization is also a very common source of bugs in parallel implementations.

- Focus for today is on performance loss in correct implementations.

Computation Overhead

A parallel program may perform computation that is not done by the sequential program.

- Algorithm: Sometimes the fastest parallel algorithm is fundamentally different than the fastest sequential one, and the parallel version performs more operations.
  - Example: Bitonic sort (coming soon!).
- Redundant computation: It is sometimes faster to recompute the same thing on each processor than to compute it once and broadcast.
  - Example: Extracting subsequence of prime numbers by sieve of Eratothenes.

Sieve of Eratosthenes

To find all primes $\leq$ N:

```
Let MightBePrime = [2, 3, ..., N].
Let KnownPrimes = [].
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.
    Let P = first element of MightBePrime.
    Append P to KnownPrimes.
    Delete all multiples of P from MightBePrime.
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 (not a prime), 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: In the parallel sieve each process had its own copy of the first $\sqrt{N}$ primes.

Overhead: Summary

Loss of performance due to extra work done by the parallel version not needed by the sequential version, including:

- Communication: Parallel processes may need to exchange data.
- Synchronization: Parallel processes may need to synchronize to guarantee that some operations (e.g. file writes) are performed in a particular order.
  - Sequential programs have their implicit sequential ordering.
- Extra Computation:
  - Sometimes the best parallel algorithm is different than the best sequential algorithm.
  - Sometimes it is more efficient to repeat a computation in several different processes to avoid communication overhead.
- Extra Memory: Data structures may be replicated in several different processes.

Causes of Performance Loss: Limited Parallelism

Sometimes, we cannot 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

Examples:

- 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.
  - A different data structure (eg: skiplists, trees, etc.) might enable more parallelism.
- 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. Common causes:

- Start-up and completion.
- Work imbalance.
- Communication delays.

Also commonly called “load imbalance” (MRR section 2.6.6).

Resource Contention

Processors waiting for a limited resource.

- It is easy to change a compute-bound task into an I/O bound task using parallel programming.
- Shared memory machines often run into memory bandwidth limitations:
  - Processing cache-misses.
  - Communication between CPUs and co-processors.
- Message passing machines often saturate the network bandwidth.

Lecture Summary

Common causes of performance loss in parallel algorithms:

1. Parallel Overhead
   - Communication
   - Sychronization
   - Computation
   - Memory
2. Limited Parallelism
   - Serial Dependency
   - Idle Processors
   - Resource Contention

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?
- Is deadlock a form of parallel performance loss?