Quantifying Performance

Mark Greenstreet

CpSc 418 – Sept. 19, 2013

Outline:

- Dependencies
- Granularity and Locality
- Performance and Speed-up
- These slides are rather sparse.
  - I'd like to make them more complete, but I won't make any promises of when I'll have them finished.
  - They are based on Lin & Snyder, Chapter 3, pp. 73–85.

Dependencies

- RAW – Read-After-Write
  - The value of a variable must be written before it can be used.
- WAR – Write-After-Read
  - The value of a variable must not be over written before all reads of its current value have completed.
  - This is a false dependency – it can be eliminated by using more memory.
- Write-After-Write
  - The last write to a variable according to the code block must be the last one in execution so we leave the block with the right values for variables. of its current value have completed.
  - This is another kind of false dependency.
- Control
  - Outcomes of branches must be determined so we can execute the right code blocks.
  - These can be mitigated with speculation

Fine-Grain Example: Matrix multiplication

```
for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        sum = 0.0;
        for(k = 0; k < N; k++)
            sum += a[i,k] * b[k,j]
} }
```

Examples of each kind of dependency:

- RAW?
- WAR?
- WAW?
- Control?

Coarse-Grain: Matrix-Multiplication by blocks

Granularity

- Definitions:
  - Fine-grained parallelism performs a small number of operations between communication actions.
  - Coarse-grained parallelism performs large amounts of computation between communication actions.
- Application:
  - If communication is expensive, then coarse-grained approaches are preferable.
    - ★ If there is an OS level context switch involved, make it coarse-grained.
  - Fast, low-overhead communication favors finer grained parallelism
    - ★ Dedicated hardware.
    - ★ GPUs.
  - If the time to complete tasks is hard to predict, that can favor using a finer grain for parallelism.
    - ★ Idle processors can work on small tasks while busy processors finish up big ones.
    - ★ If we're lucky.

Locality

Speed-up, again

- Measures of performance: latency, throughput, and FLOPS
- Sensitivity to technology
- Superlinear speedup

Scalable Speed-up

Let’s write some code

List Comprehensions

- Basic version: `[ Expr || X <- List` *, etc.* `]`
  - `Expr` is evaluated for each element, `X`, of `List`, to produce a list.
  - Example:

```
1> [ X*X || X <- lists:seq(1, 5) ].
[1,4,9,16,25]
```

- A list comprehension can apply to multiple lists:
  - Example:

```
2> [ X*X + Y || X <- lists:seq(1, 5), Y <- [1, 2] ].
[2,3,5,6,10,11,17,18,26,27].
```

  - Note the nesting:

`for each` *First_Comprehension_Variable*
  `for each` *Second_Comprehension_Variable*
    `Expr`

- A list comprehension can have filters
  - Example:

```
3> [ X*X || X <- lists:seq(1, 5), (X rem 2) == 1].
[1,9,25]
```

Two Implementations of QuickSort

- Implementation without list comprehensions:

```
qsort(List) -> qsort(List, []).

qsort([X], Suffix) -> [X | Suffix];
qsort([Pivot | T], Suffix) ->
   {Lo, Hi} = lists:partition(fun(X) -> X < Pivot end, T),
   qsort(Lo, [Pivot | qsort(Hi, Suffix)]);
qsort([], Suffix) -> Suffix.
```

- Implementation with list comprehensions:

```
qsortc([Pivot|T]) ->
   qsortc( [ X || X <- T, X < Pivot]) ++ [Pivot] ++
   qsortc([ X || X <- T, X >= Pivot]);
   qsortc([]) -> [].
```

- Which is faster?
  - The list comprehension version traverses the list twice for each `Pivot`.
  - The list comprehension version uses list concatenation which has a reputation for being slow (when it copies its left operand).
  - Let's try it.

The Quickest QuickSort

- The test set-up:

```
time(N) ->
 R = misc:rlist(N, 1000000),
 TC = time_it:t(fun() -> qsortc(R) end),
 TQ = time_it:t(fun() -> qsort(R) end),
 io:format("N = ~b~n", [N]),
 io:format(
    " with comprehensions:  mean = ~12.6e, std = ~12.6e~n",
    [  element(2, lists:keyfind('mean', 1, TC)),
       element(2, lists:keyfind('std', 1, TC)) ]),
 io:format(
    " plain quicksort:  mean = ~12.6e, std = ~12.6e~n",
    [  element(2, lists:keyfind('mean', 1, TQ)),
       element(2, lists:keyfind('std', 1, TQ)) ]).
 time() -> time(10000).
```

The Quickest QuickSort

- Run the test:

```
4> sort:time().
N = 10000
  with comprehensions:  mean = 8.359e-3, std = 3.385e-4
  plain quicksort:      mean = 9.508e-3, std = 4.236e-4
ok
```

- The list comprehension version is faster!
  - The compiler must be doing some reasonably good optimizations.

I demand a rematch!

- `lists:partition` called the comparator for each element.
- I'll write quicksort with my own partition function:

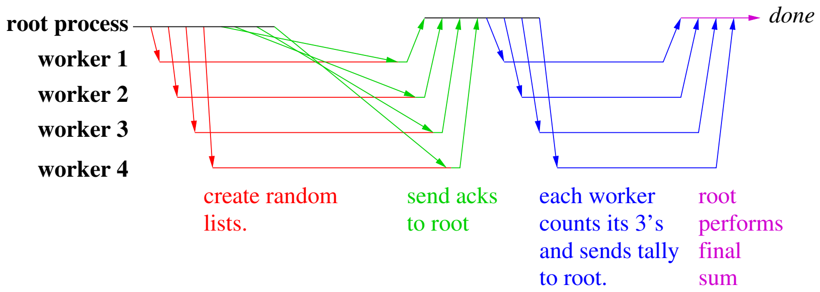
```
qsortp(List) -> qsortp(List, []).

qsortp([X], Suffix) -> [X | Suffix];
qsortp([Pivot | T], Suffix) ->
   {Lo, Hi} = partition(Pivot, T, {[], []}),
   qsortp(Lo, [Pivot | qsortp(Hi, Suffix)]);
qsortp([], Suffix) -> Suffix.

partition(_Pivot, [], {Lo, Hi}) -> {Lo, Hi};
partition(Pivot, [H | T], {Lo, Hi}) ->
   if H < Pivot -> partition(Pivot, T, {[H | Lo], Hi});
      true ->     partition(Pivot, T, {Lo, [H | Hi]})
   end.
```

- Let's try it.

```
with comprehensions:  mean = 9.180e-3, std = 5.090-4
plain quicksort:      mean = 6.372e-3, std = 4.920-4
```

- Now, the hand-coded version is ∼45% faster.
  - But the list-comprehension version is easier to write and read.

Parallel Count3's (version 1)



Parallel Count3's (the code)

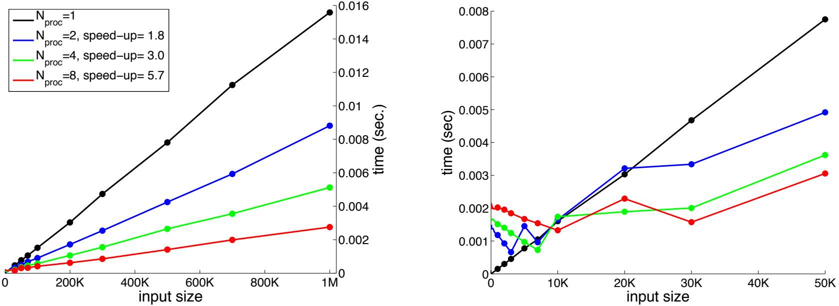
```
count3s(W, Key) ->
   lists:sum(workers:retrieve(W,
      fun(ProcState) ->
         case workers:get(ProcState, Key) of
            undefined -> failed;
            X -> count3s:count3s(X)
         end
      end)).

test(N, NWorkers) ->
   W = workers:create(NWorkers),
   rlist(W, N, 10, 'R'), % make random lists
   workers:retrieve(W, fun(_) -> ok end), % sync
   N3S = count3s(W, 'R'), % count the 3's
   workers:reap(W), % clean-up
   N3S.
```

The Workers Module

Create and manage a pools of processes.

- `workers:create(N)` – create a pool of `N` worker processes.
- `workers:reap(W)` – terminate the processes in pool `W`.
- `workers:broadcast(W, F)` – each worker in `W` executes function `F`.
  - `workers:retrieve(W, Key)` – retrieve the values associated with `Key` in each of the worker processes, and return these values as a list.
    - `workers:retrieve(W, Fun, Args)` – retrieves the value obtained by executing `Fun` in each process with the corresponding element from `Args`.
    - `workers:retrieve(W, Fun)` – retrieves the value obtained by executing `Fun` in each process without any arguments.
- see the on-line documentation for more details.

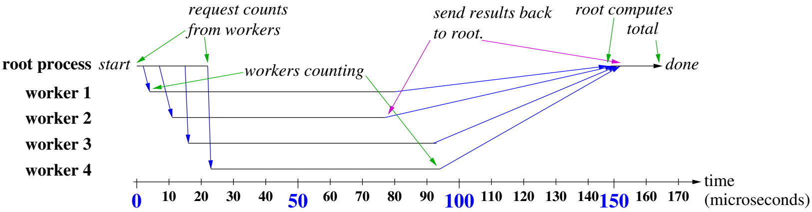
Performance



Parallel execution: 8 processes on quad-core i7

- Speed-up calculated for $N = 1\mathrm{M}$ point (of course).
- The parallel version is faster, but
  - there's a lot of overhead!

The Overhead

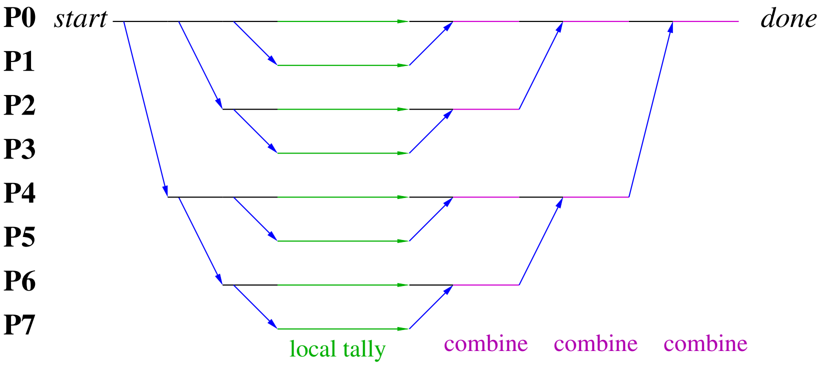


- The biggest overhead is the thread scheduler (OSX).
- Many cores are idle while there are threads waiting for work.
- The scheduler is trying to avoid unneccessary thread migration.
- Similar results when running under linux.

The Reduce Operator

- Count3's is a simple example of a common pattern in parallel computation: reduce.
  - A large vector, array, or other data structure is distributed across many workers.
  - Each worker computes a "tally" of its part of the data.
  - The tally values are combined using some associative operator to produce the final result.
- Examples:
  - Compute the sum of the elements of an array.
  - Find the largest element in an array.
  - Find the largest element in an array and its index.
  - Find the first occurrence of *Key* in an array.

Reduce


P0
start
done
P1
P2
P3
P4
P5
P6
P7
local tally
combine
combine
combine

Summary

- Library modules for parallel programming with Erlang
  - `time_it`: measure elapsed time for computations.
  - `workers`: create and use pools of worker processes.
- Example: `count3s`
  - Can you explain the observed performance loss using the kinds of losses described in the September 18 lecture?