

Quantifying Performance

Mark Greenstreet

CpSc 418 – Sept. 20, 2012

Outline:

- Measuring Time
- Count 3's performance

Questions from last lecture:

- How does having the sender and receiver of a parallel program being different threads on the same processor allow us to have one paradigm for multi-core and clusters?
- How does doing extra computation (in the prime sieve example) save time?
- What is Amdahl's Law?

The `time_it` module

`time_it` provides the following functions

- `t(Time_This, ...)` evaluate the runtime of `TimeThis()`.
 - ▶ The function has many variations: `t/1`, `t/2`, `t/3`, and `t/5`.
 - ▶ `t(Time_This)` invokes `Time_This` repeatedly until it has used a total of 1 second. `t(Time_This)` then returns a list of the form `[{mean, Mean}, {std, Std}]`, where `Mean` is the average, measured runtime, and `Std` is the standard deviation.
 - ▶ Other versions give the user finer control over how many times to invoke `Time_This` and what data is gathered from the runs.
- `log([LogIn,] Format [, Data])` Create a tuple recording the current time and the pid of the caller. `Format` and `Data` are arguments to `io_lib:format` to construct a string to identify the event being logged.
- `print_log(Log)` Print an event-log.

Example: Head vs. Tail Recursion

- Two implementations of `sum`

- ▶ Head-recursive version:

```
hr([]) -> 0;  
hr([Head | Tail]) -> Head + hr(Tail).
```

- ▶ Tail-recursive version:

```
tr(List) -> tr(List, 0).  
tr([], Sum) -> Sum;  
tr([Head | Tail], Sum) -> tr(Tail, Sum+Head).
```

- Tail-recursion:

- ▶ A function is **tail-recursive** if the recursion occurs at the very end.
 - ▶ This means that for the recursive case, the return value of the function is the value returned by the recursive call.

Optimization: Head vs. Tail Recursion

- A compiler can convert a tail-recursive function into a while-loop.

- ▶ For example, the tail-recursive version of `sum` becomes

```
tr(List) {  
    Sum = 0;  
    while(List.has_next())  
        Sum = Sum + List.next();  
    return(Sum);  
}
```

- ▶ Note: I also did the function inlining for

```
tr(List) -> tr(List, 0).
```

- The same optimization doesn't work for head-recursive functions.

- ▶ For example, to evaluate

```
hr([Head | Tail]) -> Head + hr(Tail).
```

a real function call is needed.

- ▶ The activation for the current call to `hr` must stay around until the recursive call completes to perform the final addition.

- Questions:

- ▶ Does the Erlang compiler optimize tail-recursive functions?
- ▶ Does it matter?

Timing: Head vs. Tail Recursion

- The code, two implementations of `lists:sum`:

- ▶ Head recursive:

```
hr_time(List) -> time_it:t(fun() -> hr(List) end).
```

- ▶ Tail recursive:

```
tr_time(List) -> time_it:t(fun() -> tr(List) end).
```

- The results:

```
1> c(sum).
```

```
{ok, sum} 2> R = misc:rlist(100000, 1000), ok.
```

```
ok
```

```
3> {sum:hr(R), sum:tr(R)}.
```

```
{50084422, 50084422} % good – they agree. 😊
```

```
4> sum:hr_time(R).
```

```
[[mean, 0.001237131025958], {std, 1.782649536562e-4}]
```

```
5> sum:tr_time(R).
```

```
[[mean, 0.001033205578512], {std, 7.669878290256e-5}]
```

- Conclusion:

- ▶ The tail-recursive version is $\sim 20\%$ faster.
- ▶ Tail-recursion is also important for deeply recursive functions: using tail-recursion prevents stack overflows.

List Comprehensions

- Basic version: `[Expr || X <- List , etc.]`

- ▶ `Expr` is evaluated for each element, `X`, of `List`, to produce a list.
- ▶ Example:

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

- A list comprehension can apply to multiple lists:

- ▶ Example:

```
7> [ 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:

```
8> [ 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:

```
9> 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:

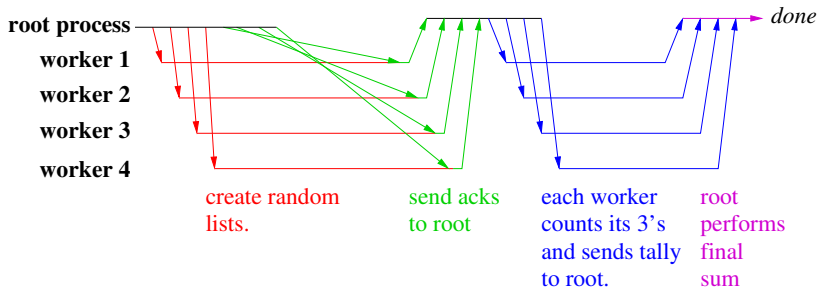
```
qsortp(List) -> qsortp(List, []).  
qsortp([X], Suffix) -> [X | Suffix];  
qsortp([Pivot | T], Suffix) ->  
    {Lo, Hi} = partition(Pivot, T, {[], []}),  
    qsortp(Lo, [Pivot | qsortp(Hi, Suffix)]);  
qsortp([], Suffix) -> Suffix.  
  
partition(_Pivot, [], {Lo, Hi}) -> {Lo, Hi};  
partition(Pivot, [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 $\sim 45\%$ 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)).

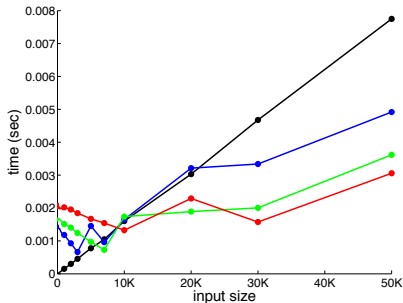
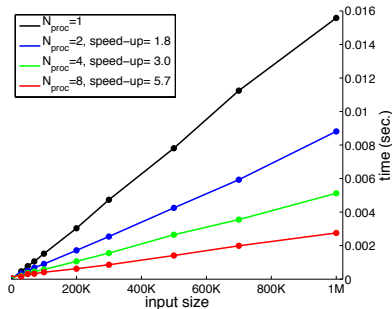
test(N, NWorkers) ->
  W = workers:create(NWorkers),
  rlist(W, N, 10, 'R'), % make random lists
  workers:retrieve(W, fun(_) -> ok end), % sync
  N3S = count3s(W, 'R'), % count the 3's
  workers:reap(W), % clean-up
  N3S.
```

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.

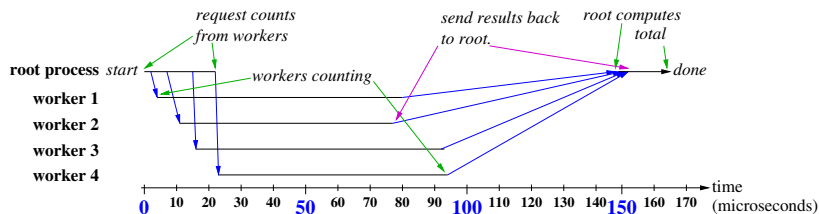
Performance



Parallel execution: 8 processes on quad-core i7

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

The Overhead

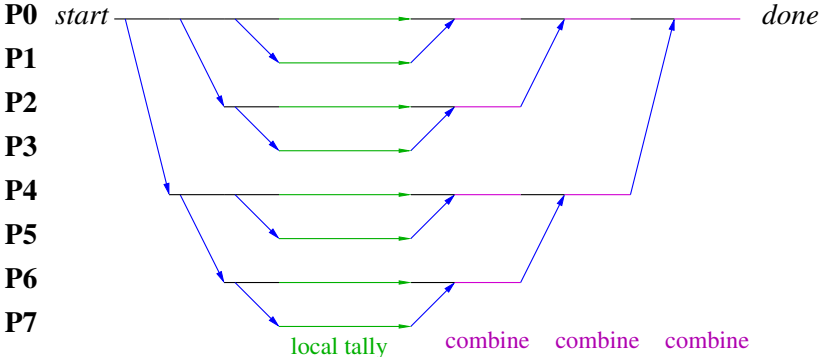


- The biggest overhead is the thread scheduler (OSX).
- Many cores are idle while there are threads waiting for work.
- The scheduler is trying to avoid unnecessary thread migration.
- Similar results when running under linux.

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



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?