CpSc 448B

Homework 2

Due: Oct. 13, 2011

Please submit your solution using the handin program. See homework 1 for more instructions on handing in homework.

For all questions that ask for timing measurements or speed-ups, please use gambier.ugrad.cs.ubc.ca. Of course, you may develop and test your code on any machine that you like. Please run your code on gambier for performance numbers. For the curious, gambier is an 8-core Niagra machine, where each core is eight-way multithreaded. Thus, it appears to the operating systems (and the erlang runtime) as a 64-processor machine.

1. **Quicksort performance (45 points).**
   In the September 22 lecture, we presented two implementations of quicksort: one that uses list-comprehensions, and another that did not. I argued that the list-comprehension version makes about 2.5 times more list traversals than the other version, but the speed difference was only about 20%. This problem explores the performance of the two implementations of quicksort in more detail.

   (a) (**10 points**) Consider the following two functions for finding all elements of a list whose values are less than that of `Pivot`:

   i.
   ```
   less1(Pivot, List) -> less1(Pivot, List, []).
   less1(Pivot, [H | T], List2) ->
      case H < Pivot of
         true ->  less1(Pivot, T, [H | List2]);
         false -> less1(Pivot, T, List2)
      end;
   less1(_Pivot, [], List2) -> List2.
   ```

   ii.
   ```
   less2(Pivot, List) -> [ X || X <- List, X < Pivot].
   ```

   - Which function, `less1(Pivot, List)` or `less2(Pivot, List)` is faster?
   - By how much?
   - Explain how you got your answer including a presentation of your raw data.

   (b) (**10 points**) Consider the following two functions for finding partitioning a list according to the value of `Pivot`:

   i.
   ```
   part1(Pivot, List) -> part1(Pivot, List, {[], []}).
   part1(Pivot, [H | T], {Lo, Hi}) ->
      case H < Pivot of
         true ->  part1(Pivot, T, {[H | Lo], Hi});
         false -> part1(Pivot, T, {Lo, [H | Hi]})
      end;
   part1(_Pivot, [], LoHi) -> LoHi.
   ```

   ii.
   ```
   part2(Pivot, List) ->
      { [ X || X <- List, X <   Pivot],
        [ X || X <- List, X >=  Pivot]
      }.
   ```

   - Which function, `part1(Pivot, List)` or `part2(Pivot, List)` is faster?
   - By how much?
   - Does the erlang compiler recognize that the two list-comprehensions for `part2` are traversing the same list and perform them together with a single traversal of `List`?
   - Explain how you got your answer including a presentation of your raw data.

(c) (**10 points**) Consider the following two functions for concatenating two lists:

i.
```
concat1([H | T], List2) -> [H | concat1(T, List2)];
concat1([], List2) -> List2.
```

ii.
```
concat2(List1, List2) -> List1 ++ List2.
```

- Which function, `concat1(List1, List2)` or `concat2(List1, List2)` is faster?
- By how much?
- Does erlang provide an optimized implementation of list concatenation?
- Measure the runtime for the `lists:append/2` function. Does it use `++` to concatenate lists?

(d) (**15 points**) Finally, consider the two implementations of quicksort from the September 22 lecture:

i.
```
qsort2([Pivot|T]) ->
   qsort2([ X || X <- T, X < Pivot]) ++
   [Pivot] ++
   qsort2([ X || X <- T, X >= Pivot]); qsort2([]) -> [].
```

ii.
```
qsort3(List) -> qsort3(List, []).
qsort3([X], Suffix) -> [X | Suffix];
qsort3([Pivot | T], Suffix) ->
   {Lo, Hi} = part1(Pivot, T, {[], []}),
   qsort3(Lo, [Pivot | qsort3(Hi, Suffix)]);
qsort3([], Suffix) -> Suffix.
```

where `part1(Pivot, List {Lo, Hi}` is the function from question 1b.

- Measure the running times of `qsort2(Pivot, List)` and `qsort3(Pivot, List)` to determine which is faster.
- By how much?
- Explain how you got your answer including a presentation of your raw data.
- Now, use the results from the previous parts of this problem to estimate the speed difference between the two functions.
- Does this more "analytical" approach agree with your measurements? In other words, having measured the runtime of the pieces of the two quicksort algorithms, can you combine these together to accurately predict the runtime of the complete algorithm. If the answer is "yes", then we've identified the critical pieces of the actual runtime. Otherwise, we've still overlooked something. What do you conclude?

Hints:

- You may use the `time_it` module or any of the other modules that I've written for this class. They are available at
  `http://www.ugrad.cs.ubc.ca/~cs448b/2011-1/src/erl/source.html`
- To use the modules from the class library, it's convenient to include them in the code path of the erlang runtime. If you're running erlang on one of the department (ugrad.cs.ubc.ca) machines, you can do this with the function:
  `code:add_path("/home/c/cs448b/public_html/2011-1/src/erl")`
  If you copy the files to your own machine, you can obviously put them in a whatever directory you like and use the appropriate call to `code:add_path/1` so that erlang will use them.
- You may also use any functions from the erlang standard libraries.
- Quantifying speed-up: If P_1 and P_2 are two programs that perform some task using times T_1 and T_2 respectively, with $T_1 < T_2$, then we say that P_1 is

$$\frac{T_2 - T_1}{T_1}(100\%)$$

faster than P_2. Equivalently, we say that P_1 is

$$\frac{T_2}{T_1}$$

times faster than P_2.

For example, if P_1 performs a task in 3.2 seconds, and P_2 takes 4.7 seconds to perform the same task, then we say that P_1 is 47% faster than P_2. Equivalently, we can say that P_1 is 1.47 times faster than P_2. Note that if $T_2/T_1 > 2$, then P_1 is more than 100% faster than P_2. For example, if P_1 performs a task in 3.2 seconds, but P_2 takes 11.1 seconds, then P_1 is 247% faster than P_2, and equivalently, P_1 is 3.47 times faster than P_2.

2. **Message performance (20 points + 10 extra credit)**

How fast can erlang send messages between processes? I wrote two programs to try to answer this question, `msg1`, and `msg2` – the sources for these two programs are listed at the end of this assignment. You can access the source files for `msg1.erl` and `msg2.erl` as well as others related to this assignment at:

`http://www.ugrad.cs.ubc.ca/~cs448b/2011-1/hw/2/src/source.html`

Here's a description of what each program does:

`msg1:time(HowManyMessages, IntsPerMessage)` creates a child process that communicates with the main process in a "ping-pong" fashion. In other words, the main process sends a message to the child. The child receives the message and sends a message to the parent. The parent receives that message and then sends another message to the child, and so on. Each process sends `HowManyMessages` to the other, and each message is a list with `IntsPerMessage` integers. Thus, a total of

$$2 * \texttt{HowManyMessages} * \texttt{IntsPerMessage}$$

integers get sent between the two processes.

`msg2:time(HowManyParentOps, HowManyMessages, IntsPerMessage)` creates a child process. In this case, the main process performs `HowManyParentOps` "operations" (i.e. something to use some time). Before starting each operation, the main process checks to see if it has a pending message from the child process. If so, the main process receives the message from the child process.

Meanwhile, the child process performs `HowManyMessages` of its own operations. Before starting each operation, the child process sends a message to its parent. The child's operations are more computationally intensive than those of the parent. This ensures that the parent's incoming message queue won't fill up. Each message from the child is a list of `IntsPerMessage` integers.

If the parent receives the last message from the child before completing it's `HowManyParentOps` operations, then it succeeds. If the parent succeeds, `msg2:time(...)` reports the mean and standard deviation for the elapsed time to do all of this stuff. If the parent doesn't succeed, `msg2:time(...)` returns the atom `'failed'`.

From the description above, it can be seen that when

`msg2:time(HowManyParentOps, HowManyMessages, IntsPerMessage)`

is executed, the child process sends a total of

$$\texttt{HowManyMessages} * \texttt{IntsPerMessage}$$

integers to the main process, and the main process sends nothing to the child process (actually, there is a small, "synchronization" message that can be ignored for this problem). All of the "operations" take extra time; so to measure the time spent for messages, you'll need to run `msg2:time(...)` with different values for the parameters and take the difference. That's a good idea with `msg1:time(...)` as well.

Now, for the questions:

(a) (**20 points**): For both `msg1` and `msg2` find coefficients t_0 and t_1 to make a model of the form:

$$t_{msg}(N_{ints}) \quad = \quad t_0 + t_1 * N_{ints}$$

wher $t_{msg}(N_{ints})$ is the time to send a message consisting of N_{ints} integers. Show your raw-data and explain how you used it to construct your model.

(b) (up to **10 points**, extra credit): When I did this, I get *much* different results for the time to send messages between the two processes. Explain why this happens. For full credit, devise a way to test your conjecture, try it, and describe the outcome of your experiment.

3. **Primes**

(a) (**20 points**) Here's a simple, sequential implementation of the sieve of Eratosthenes:

```
% primes(N): generate a list of all primes in [2..N]. primes(N) when is_integer(N) and (N > 1) ->
   P0 = lists:seq(2,N),
   M = round(math:sqrt(N)),
   {P1, P2} = do_primes([], P0, M),
   lists:reverse(P1, P2);
primes(N) when is_integer(N) -> [].

do_primes(P1, [P | T], M) when P =< M ->
   do_primes([P | P1], [X || X <- T, (X rem P) /= 0], M);
do_primes(P1, P2, _M) -> {P1, P2}.
```

Implement a function, `par_primes(N)`, that uses parallelism to find the primes up to `N`. Use worker pools as provided in the `workers` module – see `http://www.ugrad.cs.ubc.ca/~cs448b/2011-1/src/erl/source.html` Your function can (and should) leave the primes that it finds distributed amongst the worker processes. You just need to be able to get use them for the next two parts of this problem. Measure the speed-up of your implementation compared to the sequential code described above for finding the primes that are less than 1,000,000.

(b) (**10 points**) Use the function `workers:reduce` to implement a function `sum_primes(N) -> Sum` that computes the sum of the primes that are less than or equal to `N`. Your function should return this sum. Report the sum of the primes that are less than 1,000,000, and report how long it takes `sum_primes` to compute this.

(c) (**5 points**) A pair of primes, (p_1, p_2) are said to be twin primes if $p_2 = p_1 + 2$. I'll declare that a pair of primes, (p_1, p_2) are "century primes" if $p_2 = p_1 + 100$. Write a sequential erlang function `century_primes(PList) -> HowMany` that returns the number of pairs of century primes in `PList`. You may assume that the primes in `PList` are in ascending order. How many pairs of century primes are there where p_2 is less than 1,000,000?

(d) (**10 points**) Use the function `workers:reduce` to implement a function `par_century(...) -> HowMany` that computes the number of pairs of century primes that were computed by a call to `par_primes`. Report the speed-up of your implementation compared with the sequential version that you did above.

msg1.erl

```
-module msg1.
-export [time/2].

debug_timeout() -> 500.  % give up on a receive after 0.5 seconds

time(HowManyMessages, IntsPerMessage) ->
  MyPid = self(),
  M = lists:seq(1,IntsPerMessage),
  ChildPid = spawn(fun() -> child_proc(MyPid, HowManyMessages, M) end),
  T = time_it:t(fun() ->
    case misc:sync(ChildPid, debug_timeout()) of
       ok -> send_before_recv(ChildPid, HowManyMessages, M);
       failed -> failed
    end end),
  ChildPid ! {MyPid, exit},
  T.

child_proc(ParentPid, HowManyMessages, M) ->
  case misc:sync(ParentPid, debug_timeout()) of
     ok ->     recv_before_send(ParentPid, HowManyMessages, M),
               child_proc(ParentPid, HowManyMessages, M);
     exit ->   ok;
     failed -> failed
  end.


% send_before_recv:
%   Exchange messages with Dst.
%   We're the 'active' side -- for each round, we send before we receive.
%   R gets updated with each round, and we return the final tuple for R.
%   If a receive times out, we return the tuple 'failed'.
send_before_recv(_Dst, 0, _M) -> ok;
send_before_recv(Dst, N, M) ->
  Dst ! {send_before_recv, M},
  receive
    { recv_before_send, _ } -> send_before_recv(Dst, N-1, M)
    after debug_timeout() ->
      misc:msg_dump(["send_before_recv: {recv_before_send, V1}"])
  end.

% recv_before_send:
%   Exchange messages with Dst.
%   This is the counterpart to send_before_recv --
%   for each round of messages, we receive before we send.
recv_before_send(_Dst, 0, _M) -> ok;
recv_before_send(Dst,  N, M) ->
  receive
    { send_before_recv, _} ->
      Dst ! { recv_before_send, M },
      recv_before_send(Dst, N-1, M)
    after debug_timeout() ->
      misc:msg_dump(["recv_before_send: {send_before_recv, V1}"])
  end.
```

msg2.erl

```
-module msg2.

-export [time/3].

time(HowManyParentOps, HowManyMessages, IntsPerMessage) ->
  MyPid = self(),
  M = lists:seq(1,IntsPerMessage),
  ChildPid = spawn(
    fun() -> child_proc(MyPid, HowManyMessages, M) end),
  T = time_it:t(fun() ->
    case misc:sync(ChildPid, debug_timeout()) of
       ok ->
         case step2a(M, HowManyParentOps, not_done) of
   failed -> failed;
   _ -> ok
 end;
       failed -> failed
    end end),
  ChildPid ! {MyPid, exit},
  T.

child_proc(ParentPid, N, M) ->
  case misc:sync(ParentPid, debug_timeout()) of
     ok ->     step2b(ParentPid, N, M),
             child_proc( ParentPid, N, M);
     exit ->   ok;
     failed -> failed
  end.

% step2a:
step2a(_M, 0, done) -> ok;
step2a(_M, 0, not_done) -> failed;
step2a(M, N, Status) ->
  receive
    { step2b, V1 }  -> step2a(V1, N, not_done);
    step2b_done -> step2a(M,  N, done)
    after 0 ->
      step2a([ X bxor 123456789 || X <- M], N-1, Status)
  end.

% step2b:
step2b(Dst, 0, _M) ->
  Dst ! step2b_done,
  ok;
step2b(Dst, N, M) ->
  Dst ! {step2b, M},
  step2b(Dst, N-1, grind(M)).

% something to make the child process slower than the parent.
grind(M) -> lists:map(fun(X) -> round(X*math:cos(X)) end, M).

debug_timeout() -> 500.  % give up on a sync after 0.5 seconds
```