CpSc 418 **Homework 3** Due: Nov. 1, 2013, 11:59pm

5% extra credit if solution submitted by 11:59pm on Oct. 29.

Please submit your solution using the handin program. Submit the your solution as

cs418 hw3

Your submission should consist of the following four files:

`matrix_mult.c` and `merge_sort.c` – C source code (ASCII text).

`hw3.erl` – Erlang source code (ASCII text).

`hw3.txt` or `hw3.pdf` – plain, ASCII text or PDF.

The first three files (`matrix_mult.c`, `merge_sort.c`, and `hw3.erl`) will be your solution to the programming part of the assignment. Unless otherwise stated, you may use any functions that you like from the Erlang library and C libraries.

The second file, `hw3.txt` or `hw3.pdf` should give your solutions to the written parts of the assignment.

No other file formats will be accepted.

1 The Questions

1. **Reduce Redux (30 points)** Given a list of numbers, `L`, we want to find the biggest "gap" `L`. This is the pair of elements, `A, B` such that `A` and `B` are adjacent in `L`, and |`A` − `B`| is the largest for all such pairs of elements. Here's an Erlang implementation:

```
biggest_gap([]) -> 0;
biggest_gap([_]) -> 0;
biggest_gap([A, B | Tl]) -> max(abs(A-B), biggest_gap([B | Tl]).
```

I defined biggest gap for an empty or singleton list to be 0, because that will make the rest of this problem simpler. We can also do this with a tail recursive implementation, which is a good idea because we will want a parallel implementation which really only makes sense if `L` is long enough to make up for the overhead of the parallel version. The tail-recursive version is:

```
biggest_gap(L) -> biggest_gap(L, 0);
biggest_gap([], Gap) -> Gap;
biggest_gap([_], Gap) -> Gap;
biggest_gap([A, B | Tl], Gap) -> biggest_gap([B | Tl], max(Gap, abs(A-B)).
```

Of course, our parallel version will use `wtree:reduce`. Figure 1 shows an example of the computation. Note that the "distributed list" in the figure corresponds to the list:

`[1,2,4,8,9,3,1,1,2,4,8,16,1,1,2,3,5,8,17,19,23,29,31]`

At the leaves, each worker computes a triple. Clearly, each needs to compute the largest element for its piece of the list. We also need to consider the gaps between the last element of this worker's list and the first element of the next worker's list. This will be handled by `Combine`. To do this, `Combine` will need to know the first and last element of the part of the list that is handled by each of its subtrees. In my version, this is done by writing `Leaf` and `Combine` to return triples of the form:

`{Left, Gap, Right}`

where:

`Left` is the value of the leftmost element of the list for the subtree.

`Right` is the value of the rightmost element of the list for the subtree.

`Gap` is the largest gap in the subtree.

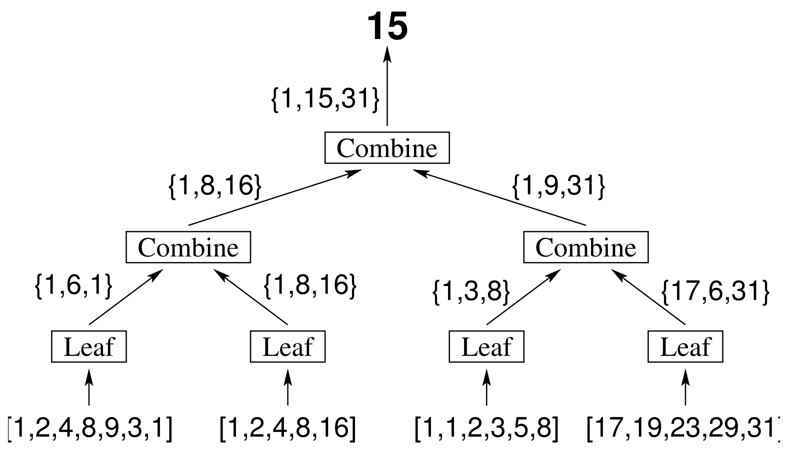

Figure 1: Reduce for `biggest_gap`

As a special case, `Leaf` and `Combine` will return the atom `'empty'` if the list for their subtree is empty. Erlang code for `biggest_gap(W,Key)` is given in `hw3.erl`.

The remainder of this problem addresses how to write a new function,

```
    biggest_gap_i(W, Key) -> {Gap, Index}
```

The value of `Gap` is the same as `biggest_gap(W, Key)`. The value `Index` is the position in the list referred to by `Key` of the first element of the first pair with a gap of size `Gap`. If the list corresponding to `Key` has a length less than two, then `biggest_gap_i(W, Key)` should return the tuple `{0, 'undefined'}`. For the problem shown in Figure 1,

```
    biggest_gap_i(W, Key) -> {15, 12}
```

(a) (**5 points**) Write a *sequential* implementation of `biggest_gap_i` as shown below:

```
biggest_gap_i(List) when length(List) < 2 -> {0, undefined};
biggest_gap_i([Hd | Tl]) -> {Gap, Index}.
```

where you need to replace `{Gap, Index}` with code that computes the specified values for these. For full credit, your solution should be tail-recursive. One point will be deducted for head-recursive implementations.

(b) (**10 points**) Draw a tree similar to the one in Figure 1 showing how `biggest_gap_i` can be computed as a reduce. Your `Leaf` and `Combine` operations should return tuples. Describe the tuple that these functions return – what is the significance of each element of the tuple? Are there any special cases where the functions return something else?

(c) (**15 points**) Write a *parallel* implementation of `biggest_gap_i` as shown below:

```
biggest_gap_i(W, Key) -> {0, undefined};% If the list associated with Key has fewer than two elements.
biggest_gap_i(W, Key) -> {Gap, Index}.  % otherwise
```

where you need to replace `{Gap, Index}` with code that computes the specified values for these. Your solutions should use the `wtree:reduce` function. In particular, write the functions: `biggest_leaf_i`, `biggest_combine_i`, and `biggest_root_i`. You should try to re-use code from your sequential solution where it makes sense possibly making small adaptations as described in Section 2 below.

The file

    http://www.ugrad.cs.ubc.ca/~cs418/2013-1/hw/3/code/hw3.erl

includes templates for thiese functions.

2. **Instruction Level Parallelism.** I have written simple implementations of matrix-multiply and merge-sort in C. You can download them from:

   matrix multiply: `http://www.ugrad.cs.ubc.ca/~cs418/2013-1/hw/3/code/matrix_mult.c`

   merge sort: `http://www.ugrad.cs.ubc.ca/~cs418/2013-1/hw/3/code/merge_sort.c`

   Both should be compiled using `gcc`'s `-std=c99` and `-O4` flags. By using the `-S` flag, `gcc` will generate the assembly code that you can read. For example:

   ```
   gcc -std=c99 -O4 -S matrix_mult.c
   ```

   will produce `matrix_multiply.s` that you can read as you answer the questions below.

   (a) (10 points) How many instructions per clock-cycle (average) does `matrix_multiply` execute when computing the product of two, 100×100 matrices? Please use the function `matrix_rand` from matrix_mult.c to generate the 100×100 matrices for your experiments.

   (b) (10 points) How many instructions per clock-cycle (average) does `merge_sort` execute when sorting a list of $1,000,000$ elements? Please use the function `list_rand` from merge_sort.c to generate the $1,000,000$ list(s) for your experiments.

   (c) (10 points) Explain why the `matrix_multiply` and `merge_sort` have different levels of instruction-level parallelism. Connect the performance you measured with the operation of super-scalar processors from class.

   Hints:

   - Please report instructions/per cycle to two significant digits, e.g., 1.4, 0.012, 42, 230, 0.67, etc. This means that you don't need to account for *every* instruction that gets executed. Give a short description of what you measured and what you didn't, and your guess of how this affects the accuracy of your results.
   - For linux machines,
     `cat /proc/cpuinfo`
     reports lots of things about the machine's CPU including the make, model, and clock frequency. You can get this to determine the clock period.
   - As usual, you're welcome to use a recent (at most 5 years old) machine of your choice. The `linXX@ugrad.cs.ubc.ca` machines are the default for this assignment. If you use another machine, please describe what it is. If you use a machine other than the `linXX` boxes and get strange timings, I will ask you to try your code on a `linXX` box before answering other questions.
   - Mini-assignment 3 provided examples of how to measure the execution time of C code.

3. **Dependencies (20 points)** Using the code for the `merge` or `merge_sort` functions from merge_sort.c

   (a) Describe two RAW dependencies. Identify the dependant operation (e.g. "the read of variable `q` in the statement `p = q+r;`") and the earlier operation that it depends on (e.g. "the write of `q` in the statement `q = sqrt(s);`").

   (b) Describe two WAR dependencies. As for part (a), identify the dependant operation and the operation that depends on.

   (c) Describe two WAW dependencies. As for part (a), identify the dependant operation and the operation that depends on.

   (d) Describe two control dependencies. As for part (a), identify the dependant operation and the operation that depends on.

2 Design of `biggest_gap(W, Key)` using `wtree:reduce`

I'll write a parallel version of `biggest_gap` using `wtree:reduce`. I'd like to call this `biggest_gap(W, Key)` where `W` is a worker tree, and `Key` is the key for the list to process. However, I already defined `biggest_gap(List, Gap)` as part of my tail-recursive version. I'll rename that one `biggest_gap_tr(List, Gap)` so the parallel one can have a simple name. Let's look at the `Leaf`, `Combine`, and `Root` functions for this application of `wtree:reduce`.

`Leaf` needs to get `Key` from the context where it is defined. This observation helps me to sketch-in what `biggest_gap(W, Key)` is going to look like:

```
biggest_gap(W, Key) ->
   Leaf = fun(S) ->
      MySegment = workers:get(S, Key),
      ...
   end,
   ...
```

From Figure 1, I see that `Leaf` should return a tuple of the form `{First, Gap, Last}` if `MySegment` is a non-empty list. If `MySegment` is empty, then `Leaf` should return the atom `'empty'`. I can get `Gap` from `biggest_gap(MySegment)`. I could use `lists:last` to get `Last`, but I don't like the idea of traversing `MySegment` twice. Recognizing that the tail-recursive function `biggest_gap_tr(List, BiggestSoFar)` could easily return the last element, I modify it to do so. Now, I get:

```
biggest_gap_tr([A, B | Tl], Gap) ->
   biggest_gap_tr([B | Tl], max(abs(A-B), Gap));
biggest_gap_tr([A], Gap) -> A, Gap; % A is the last element of the list
biggest_gap_tr([], 0) -> empty.
```

I'll need to adjust `biggest_gap(List)`, but I'll take care of that when I'm done implementing `Leaf`. As noted above, `Leaf` needs to have two cases, one for when `MySegment` is empty, and the other for when it is non-empty. An Erlang case-expression handles this nicely:

```
Leaf = fun(S) ->
   case workers:get(S, Key) of
      [] -> empty;
      MySegment -> erlang:insert_element(1, biggest_gap_tr(L, 0), A)
   end,
end
```

Note that I put the `[]` case first – the pattern `MySegment` would match an empty list if I put it first (or I would have to add a `when` clause).

`Combine` has special cases if either argument is the atom `'empty'`; for these cases, the result is just the other argument. Note that this works if the other element is `'empty'` or if it's a regular tuple. What if both subtrees were non-empty and therefore returned tuples?

The leftmost element of the combined list is the leftmost element of the left sublist.

The rightmost element of the combined list is the rightmost element of the right sublist.

The biggest gap is the largest of:

- The largest gap for the left subtree.
- The largest gap for the right subtree.
- The gap between the two elements where they meet.

I found it clearest to write Combine as a stand-alone function:

```
biggest_combine(LL, LG, LR, RL, RG, RR) ->
   LL, max(max(LG, RG), abs(LR-RL)), RR;
biggest_combine(Left, empty) -> Left;
biggest_combine(empty, Right) -> Right.
```

`Root`: the last combine will return a tuple, but `biggest_gap(W, Key)` should return an integer. I used the `Root` function option for `wtree:reduce` to handle the final conversion. It also has a special case for when the result is the `'empty'` atom. Finally, I added a case for when its argument is of the form `{Gap, Last}` so I could use it with `biggest_gap(List)` as well. The code for my root function is:

```
biggest_root(empty) -> 0;
biggest_root(Gap,_) -> Gap;
biggest_root(_,Gap,_) -> Gap.
```

The complete code is in hw3.erl.

3 x86 assembly code

Most x86 assembly instructions are of the form

```
opcode src dst
```

which typically means `dst` ← `dst op src`, where the operation `op` is determined by the assembly instruction op-code. `src` can be a constant, a register, or a memory read. Constants are written in the obvious way, such as `8`.

Register names on the x86 show the history of the machine. The first eight registers have special names. When used as 32-bit quantities, they are called: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, and `%esp`, and when used as 64-bit quantities they are called: `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, and `%rsp`. Register `%rsp` is the stack pointer (for code with 64-bit pointers, it will be called `%esp` for code with 32-bit pointers). Register `%rbp` (or `%ebp`) is the "base pointer" that points to the where space for the parameters and return-address for a function end, and where the space for local variables begins.

Operands that refer to memory are enclosed in parentheses. The sum of the registers and possible constant inside the parentheses is computed, and that sum is used for the address of the memory read or write. For example,

```
mulsd (%rsi,%rax,8), %xmm0
```

computes the sum of the values in registers `%rsi` and `%rax` and adds eight to that value. The result is used as a memory address. The 64-bit double precision value read from that memory address is multiplied by the value in register `%xmm0`. The product is then stored in register `%xmm0`.

Here are descriptions of all of the x86 instructions that appeared in the assembly code when I compiled matrix_mult.c and merge_sort.c. You don't need to know all of these to do the assignment. Look for clear marker instructions (such as mulsd for a floating point multiply) to find the inner loops. Then, follow the code through the comparison (`cmpl` or `testq`) and branch (`je`, `jg`, `jge`, etc.) instructions to figure out which instructions are in the body of the loop, etc.

`addl`: add long (32 bits)
Example: `addl %r12d %r14d` computes the sum of registers 12 and 14 (treated as 32-bit two's-complement integers) and stores the sum in register 14. In fewer words,

`%r14d` ← `%r12d %r14d`

`addsd`: add scalar-double-precision.
A double-precision, floating-point add. The "scalar" part is to distinguish it from a "packed" operation (MMX) that performs vector operations.
Example: `addsd %ximm0 %ximm1` computes the sum of registers `%ximm0` and `%ximm1` (they are registers of the SSE unit, think floating point). The sum is stored in register `%ximm1`. In fewer words,

`%ximm1` ← `%ximm0 + %ximm1`

`call`: Call a procedure.
The return-address is pushed onto the stack, and execution continues at the target address.
Example: `call _malloc` calls the C-library memory allocation function, `malloc`.

`cmpl`: Compare long (32 bits).
Compare two values and set the processor flags. These flags can be used by a conditional branch instruction such as `je` ("jump equal").
Example: `cmpl %ecx, %r8d` – the result will be "less than" if the value in register `%ecx` is less than the value in register `%r8d`, "equal" if the two are the same, and "greater" if the value in `%ecx` is greater than the value in `%r8d`.

`imul`: integer multiply.

`incl`: add one to a register.

`je`: jump equal.
Example: `je L2` jumps to L2 if the last flag-setting operation (most likely a `cmpl` or `testq`) set the condition flags to "equal".

`jg`: jump greater-than.

`jge`: jump greater-than-or-equal.

`jl`: jump less-than.

`jle`: jump less-than-or-equal.

`jmp`: jump.
Always jump to the target (not conditional).

`jne`: jump not-equal.

`leaq`: Load effective address (64 bits).
The first operand is an pointer calculation. The address is computed, but instead of using it to access memory, the result is stored in the second operand.
Example: `leaq (%rdx,%rax,8), %rbx`, the effect is

```
%rbx ← $rdx + $rax + 8
```

`leave`: procedure return.
A bunch of stuff is done to clean up the stack and restore the stack pointer. Usually appears just before `ret` (procedure return).

`movq`: move.
Copy the value of the `src` operand to the `dst` operand.
Example: `movq %rdi, %r13` – copy register `%rdi` to register `%r13`.

`movsd` move scalar double-precision.
Move a double-precision value.
Example: `movsd (%r10), %xmm0` – register `%r10` is a pointer to memory. Read the double-precision number at that location and store it in register `%xmm0`.

`movslq` move with sign-extension.
This is gnu-speak for what Intel calls `movesxd`. It moves a 32-bit value into a 64-bit register and sign-extends it.
Example: `movslq %r12d,%rax` – takes the 32-bit value in register `%r12d`, sign-extends it to a 64-bit integer, and stores the result in register `%rax`.

`mulsd`: multiply scalar double-precision.
Example: `mulsd (%r14), %xmm1` – register `%r10` is a pointer to memory. Read the double-precision number at that location, multiply it by the double-precision number in register `%xmm1` and store the result in register `%xmm1`.

`popq`: pop (64-bit). Pop a 64-bit word off the stack and store it in the destination. This usually occurs at the end of a function – the values of registers that were used in this function are restored to their previous values (the values from the caller) before returning.

`pushq`: push (64-bit). Push a 64-bit word onto the stack. This usually occurs at the beginning of a function – the values of registers that will be used in this function are saved on the stack so they can be restored (by corresponding `pop` instructions) at the end of the function.

`ret`: return.
Return from a function. The return address is popped off the stack and execution continues from that address.

`salq`: shift arithmetic left (64 bit).

`sarq`: shift arithmetic right (64 bit).

`testl`: test (32 bit).
Compute the bit-wise and of the two operands and sets the sign, zero, and parity flats. The typical use of this (by `gcc`) is to check if the value in a register is positive, negative, or zero.
Example: `testl %ecx, %ecx` – sets the flags according to the value of register `%ecx`. In the `gcc` output from which I grabbed this one, the next instruction was `jle L9` – the code will jump to the instruction at label `L9` if the value in register `%ecx` was less-than-or-equal-to zero.

`testq`: test (64 bit).

`xorl`: bit-wise exclusive-or (32 bit).

`xorpd`: bit-wise exclusive-or (double precision).
The compiler uses this to set the value of a double-precision register to 0.0.
Example: `xorpd %ximm1, %ximm1` – means `%ximm1 ← 0.0`.

I found it helpful to modify the C-source code by adding static variables that count the number of times various blocks of code (e.g. loop-bodies, then-clauses, else-clauses, whole functions) were called. I wrote a `main` function that reports these. Once I knew the number of times various block were executed, I could identify the ones that were executed the most and figure out how many machine instructions they executed. Obviously, adding these counters and incrementing them changes the number of instructions in the code. So, I used the annotated code to figure out how many times each block was executed, and then the unannotated code to figure out how many instructions were executed in each of the most frequently executed blocks.

For further reading:

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2a-manual.pdf
The Intel assembly manual (instructions whose names begin with 'A'–'M').

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf
The Intel assembly manual (instructions whose names begin with 'N'–'Z').

http://en.wikipedia.org/wiki/X86-64 Wikipedia's overview of the `x86_64` instruction set architecture.

http://en.wikipedia.org/wiki/FLAGS_register Wikipedia's description of x86 flags.

http://blog.reverberate.org/2009/07/giving-up-on-at-style-assembler-syntax.html
A blogger ranting about how idiocy of calling an instruction `movslq`.

http://randomascii.wordpress.com/2012/12/29/the-surprising-subtleties-of-zeroing-a-register/
Another blogger examines how to clear a register – a nice look at some of the optimizations done by a superscalar. Thanks to LK (a former student in this course) for sending me the link.