CpSc 448B **Midterm** October 20, 2011

100 points

Answer **all** of questions 1 through 3. Answer any **four** of questions 4 through 8.

1. **Erlang (14 points).**

   (a) **Read some code** (**5 points**)
   Consider the module `q1a` shown in Figure 1a. Write down an Erlang expression, `E`, your score for this problem will be `q1a:score(E)`.

   Solution: `[1,0,1,'$']`

   Explanation: *The value returned by `q1a:score` is the value sent by the child process back to the main process. This is the value returned by the `helper` function. `helper` is a recursive function that reads a sequence of bits (i.e. `0` or `1` values) that are sent to it and interprets them as an unsigned, binary integer. The bit-order is most-significant bit first (i.e. a "big-endian"). When the atom `'$'` is received, helper returns the value of the integer that it has constructed.*
   Finally, the code is picky. `helper` only accepts the messages `0`, `1`, and `'$'`. `helper` always returns an integer; however, this integer could be larger than 5. To avoid giving unlimited points for this problem, the `score` function only accepts values from helper that are at most 5. Anything larger is treated as zero.

   (b) **Write some code** (**9 points**)
   Consider the module `q1b` shown in Figure 1b. For this problem, you need to write an implementation of function `q1b` such that each call to `q1b:read(P)` returns the last value "written" by a call to `q1b:write(P, Value)`, where P is an Erlang pid returned by a call to `q1b:create()`. For example, here's a transcript of an Erlang session using my solution:

```
1> c(q1b).
{ok,q1b}
2> P = q1b:create().
<0.38.0>

2> q1b:write(P, 2).
{write,2}
3> q1b:read(P).
2
4> q1b:write(P, 17).
{write,17}
5> q1b:write(P, 42).
{write,42}
6> q1b:read(P).
42
7> P2 = q1b:create().
<0.47.0>
8> q1b:read(P2).
empty
```

Solution:

```
q1b(Value) ->
   receive
      {read, Pid} ->
         Pid !  {self(), 'read', Value},
         q1b(Value);
      {write, NewValue} -> q1b(NewValue)
   end.
```

Explanation: *function `q1b` keeps track of the value of the last write operation in its `Value` parameter. Erlang is functional; so, we can't overwrite `Value`. By using a recursive function, each read or write operation results in a fresh call to `q1b`, and each call can have its own value for `Value`. The **last** call is the active one and the one that responds to read and write requests.*
When `q1b` receives a message of the form {`write, NewValue`}, a recursive call is made using `NewValue`, this effects the write. Conversely, when `q1b` receives a message of the form {`read, Pid`}, it sends the value of its `Value` parameter to `Pid`. This is the last value written, thus implementing the specification.
Finally, function `q1b` is tail recursive. This means it can process an unbounded number of read and write requests without causing a stack-overflow.

2. **Dependences** (**21 points**)

   (a) (**6 points**)
   What are the three types of "dependences" defined in Lin & Snyder? Describe each with one or two short sentences.

   Solution:

   Flow Dependence: A read of a memory location (e.g. a variable in a program) follows a write to the same location. The read must be performed after the write to ensure that the read gets the correct value.

   Anti Dependence: A write to a memory location follows a read of the same location. The write must be performed after the read to ensure that the read gets the correct value.

   Output Dependence: A write to a memory location follows another write – call these write-1 and write-2. The second write must be performed after the first one to ensure that the memory location has the correct value for subsequent reads that may occur in the program.

   Explanation: *See Lin & Snyder, p. 69 – that's Chapter 3 → Parallel Structure → Dependences → Data Dependence for those who have a different edition or printing than the one on my desk.*
   Grading: *Three points for each type of dependence that is correctly described. If a solution quotes the Lin & Snyder text directly, then proper attribution must be given.*

   (b) (**6 points**)
   Figure 2 shows the C and assembly code for the inner-loop of matrix multiplication. Identify one example of each kind of dependence.

```
-module q1a.
-export [score/1].

score([H | T]) ->
   MyPid = self(),
   Cpid = spawn(fun() -> MyPid !  helper(0) end),
   send(Cpid, [H | T]),
   receive
      N when N =< 5 -> N;
      _ -> 0
   end;
score(_) -> 0.

helper(V) ->
   receive
      0 -> helper(2*V);
      1 -> helper(2*V + 1);
      '$' -> V
   end.

send(_Pid, []) -> ok;
send(Pid, [H | T]) ->
   Pid !  H,
   send(Pid, T).
```

(a) Module for question 1a

```
-module q1b.
-export [create/0, read/1, write/2].

create() -> spawn(fun() -> q1b(empty) end).

read(Pid) ->
   Pid !  {read, self()},
   receive
      {Pid, read, Value} -> Value
   end.

write(Pid, Value) -> Pid !  {write, Value}.
```

(b) Module for question 1b

Figure 1: Modules for question 1

Solution: *I'll give several examples for each, but only one is required for a correct solution.*

Flow Dependence:

`ld $fa, 0($aptr)` → `fmult $fp, $fa, $fb`

Explanation: *Register* `$fa` *must be updated by the load instruction before its value is used by the floating-point multiply instruction. In the C-code, this means that the value of* `a[i,k]` *must be read before the product* `a[i,k] * b[k,j]` *is computed.*

`add $aptr, $aptr, 8` → `(ld $fa, 0($aptr))`

Explanation: *Here, I wrote* `(instr)` *to denote the occurrence of the instruction instr in the next iteration of the loop. This dependence says that the update of* `$aptr` *in the current loop iteration must complete before* `$aptr` *is used in the next loop iteration.*

A similar dependence exists in the C-code. For the assembly, I assumed that the compiler had performed standard optimizations, this included replacing array index calculations with pointer operations, and replacing the loop index with expressions on these pointers – in other words, the loop index, `k`*, is eliminated by the compiler. In the C-code, a related dependence is*

`k++` → `(a[i,k])`

In English, the increment of `k` *for one loop iteration must complete before the value of* `k` *is used in the next loop iteration.*

`fmult $fp, $fa, $fb` → `(fadd $fsum, $fsum, $fp)`

Explanation: *The floating-point multiplication must be performed before the sum that uses the produce. In the C-code, this means that the multiplication operation for* `a[i,k]*b[k,j]` *must be performed before the addition operation for* `sum += ...`*.*

Anti Dependence:

`ld $fa, 0($aptr)` → `add $aptr, $aptr, 8`

Explanation: *The value of* `$aptr` *must be used for computing the address of the load before* `$apter` *is updated by the add instruction.*

The corresponding dependence in the C-code is `a[i,k]` → `k++`*. This means that the value of* `a[i,k]` *must be read before the variable* `k` *in incremented at the end of the loop iteration.*

`fadd $fsum, $fsum, $fp` → `(fmult $fp, $fa, $fb)`

Explanation: *The floating point addition for the current loop iteration must complete before the floating point multiply for the next iteration is done and overwrites the register holding the product,* `$fp`*.*

Output Dependence:

`fmult $fp, $fa, $fb` → `(fmult $fp, $fa, $fb)`

Explanation: *The floating point multiply for this loop iteration must be performed before the multiply for the next loop iteration so that register* `$fp` *will have the value of the* ***final*** *product at the end of the loop.*

All of the instruction that write registers (i.e. every instruction except for the `bne`*) have an associated output-dependence. I chose the* `fmult` *because the others had flow- or anti-dependences as well. Any instruction (except for the* `bne`*) is an acceptable answer.*

In the C-code, all of the output dependences are also flow dependences. With that in mind, one could list

`sum += ...` → `(sum += ...)`*, or*

`k++` → `(k++)`

as an output dependence (and get full credit).

(c) (**3 points**)

Which of the dependences that you identified for question 2b are true dependences and which are false dependences?

C:

```
for(int k = 0; k < n; k++)
   sum += a[i,k] * b[k,j];
```

assembly:

```
loop:   ld    $fa, 0($aptr)           // fa ← a[i,k]
        add   $aptr, $aptr, 8         // move aptr to next column
        ld    $fb, $bptr              // fb ← b[k,j]
        add   $bptr, $bptr, $bstride  // move bptr to next row
        fmult $fp, $fa, $fb           // fp ← a[i,k]*b[k,j]
        fadd  $fsum, $fsum, $fp       // sum ← sum + a[i,k]*b[k,j]
        bne   $aptr, $atop, loop      // loop test:  k < n
```

Figure 2: C and assembly for the inner-loop of matrix multiplication

Solution: *The flow-dependences are **true** dependences. The anti-flow and output-dependences are **false** dependences.*

Explanation: *See Lin & Snyder, p. 69 – that's Chapter 3 → Parallel Structure → Dependences → Data Dependence for those who have a different edition or printing than the one on my desk.*

(d) (**6 points**)

Briefly explain how a superscalar processor handles each dependence that you identified for question 2b.

Solution:

Flow Dependence: I'll focus on flow dependences for values held in registers. Superscalar machines use "busy bits" for registers to enforce flow dependences. If instruction-1 writes a logical register that should be subsequently read by instruction-2, then the physical register that instruction-1 will write is allocated in the rename state and marked as "busy". When instruction-2 is renamed, it is set to read the same register. Instruction-2 cannot execute as long as the register is marked as busy. Instruction-1 marks the register as "ready" (and thus no longer "busy") when it updates the value of the register. When all of the registers read by instruction-2 are ready, then instruction-2 can execute. Thus, instruction-2 will execute after instruction-1, which is what is required by the flow dependence. physical register tells whether or not the register has been updated by the instruction to which the register was bound (in the renaming process).

Anti Dependence: If instruction-1 reads a logical register that is subsequently written by instruction-2, the renaming process will assign a different physical register to be written by instruction-2 than the one that is read by instruction-1. This allows instruction-1 and instruction-2 to execute in any order, and the write by instruction-2 will not alter the value read by instruction-2.

Output Dependence: If instruction-1 writes a logical register that is subsequently over-written by instruction-2, the renaming process will assign a different physical registers for the two instructions. Instructions that occur in program order after instruction-2 will see the renaming for instruction-2's version of the register. Thus, they will read the correct value regardless of the order in which the two instructions execute.

3. **Multicore, Transactions and the Future of Distributed Computing** (**5 points**)

What was the main problem addressed by the talk?

(a) Persistent memory (e.g. DRAM with battery back-up) will replace disk-based transaction systems in future distributed applications such as web-servers.

(b) Declaring blocks of operations to be atomic is a more natural way to handle synchronization than either coarse or fine grained locks and should lead to simpler programming and more efficient multicore and distributed systems.

(c) Programming of multicore and distributed systems should be built on top of a transactional, relational data base layer to provide a higher-level programming model and robustness against system failures.

(d) Current cache-memory hierarchies work poorly with database servers and web-servers. Memory systems for multicore processors should be designed to directly support database and distributed, web-style transactions.

(e) Multicore computing is vastly increasing the compute power of individual computer systems. To exploit the extra processing power, future distributed systems will need a transaction-based paradigm to replace the client-server framework that is prevalent today.

Solution: *b*

Explanation: *The talk described problems with using locks in shared memory programs including:*

- *Coarse-grain locking can produce poor performance because they inhibit concurrent (i.e. parallel) execution.*
- *Fine-grain locking can produce poor performance because of the high overhead of acquiring and releasing the locks.*
- *Programmers tend to make errors when using locks: too much locking results in deadlock, too little locking results in interference – two threads make inconsistent, overlapping changes to the same data structure.*
- *Locks cannot be composed in a natural way.*

The talk described the use of transactional memory. In the source code, these written as `atomic` *blocks. For example, the speaker described extensions to Java with an* `atomic` *keyword. An* `atomic` *block is executed without locks, and the hardware and/or software perform checks to make sure that other threads do not modify values that were read by this threads, nor do other threads read or write any values written by this thread. If no such interference is detected, the execution of the atomic block is regarded as successful, and execution proceeds. If interference is detected, one (or more) thread(s) has its state restored to what it was when entering the atomic block, and the execution is attempted again.*
Now, for the other choices:

a: *The talk did not discuss persistent memory or battery back-up.*

c: *The talk did not discuss relational database servers.*

d: *The talk did not discuss relational database servers.*

e: *The talk did not discuss the client-server framework.*

I could give more reasons that the other choices don't apply, but this should be sufficient.

Answer **any four** of questions 4 through 8. If you respond to all five, please indicate which four to mark. Otherwise, I will make an **arbitrary** choice.

4. **Sequential Consistency** (**15 points**)
   What is sequential consistency?
   Your answer should be at most 100 words long.

Solution: *A shared-memory multiprocessor is sequentially consistent, if for every program execution there is a total ordering of memory reads and writes such that:*

- *The order is consistent with the local order of reads and writes on each processor; and*
- *Each read gets the value of the preceding write in the global ordering.*

5. **Snooping Caches** (**15 points**)
   What is a snooping cache?
   Your answer should be at most 100 words long.

Solution: *A snooping cache provides memory consistency by observing the actions of the other caches. In particular, multiple caches may have read-only copies of the same cache line. If any cache attempts to write to a cache line, all other caches holding that line must first invalidate their copies.*

Questions 6 through 8 address the speed-ups that can be achieved when multiplying two $N \times N$ matrices using P processors.

6. **Amdahl** (**15 points**)
   Amdahl suggested modeling a computation as having some fraction, s that must be performed sequentially, and the rest $(1 - s)$ that can be run P times faster when run on P processors. Assume that for matrix multiply, $s = 0.01$. According to Amdahl's law, how many times faster is the parallel version than the sequential one for $P = 10$, $P = 100$, and $P = 1000$?

Solution:

$$\begin{aligned} \textit{speed-up} &= \frac{T_{seq}}{T_{par}} \\ T_{par} &= T_{seq}\left(s + \tfrac{1-s}{P}\right) \\ speed-up &= \frac{P}{1 + (P-1)s} \end{aligned}$$

From the problem statement, $s = 0.01$*:*

$$\begin{array}{rcrcrcrcr} P & = & 10 & \rightarrow & \textit{speed-up} & = & 10/1.09 & = & 9.2 \\ P & = & 100 & \rightarrow & \textit{speed-up} & = & 100/1.90 & = & 52.6 \\ P & = & 1000 & \rightarrow & \textit{speed-up} & = & 1000/11 & = & 90.2 \end{array}$$

7. **Algorithm 1** (**15 points**)
   Most of the overhead for matrix multiplication is from the communication time required to send blocks of matrix elements between processors. In class, I sketched an algorithm where each processor sends and receives $P - 1$ messages of size N^2/P.

   - Assume that a single processor can compute the product of two $N \times N$ matrix in N^3 clock cycles.
   - Assume that each processor for the parallel version requires time N^3/P for computation plus the time required to send and receive $P - 1$ messages of size N^2/P.
   - Assume that the total time consumed at the sender and receiver to convey a message of M matrix elements is $1000 + 10M$ processor cycles.

   According to this model, how many times faster is the parallel version than the sequential one for $P = 10$, $P = 100$, and $P = 1000$? Use $N = 1000$ for all three values of P.

Solution:

$$\begin{aligned} \textit{speed-up} &= \frac{P}{1 + \frac{10\sqrt{P}}{N^3}(100P + N^2)} \\ &= \frac{N^3}{(N^3/P) + (P-1)(1000 + (10N^2/P))} \\ &= \frac{P}{1 + \frac{10(P-1)}{N^3}(100P + N^2)} \end{aligned}$$

$N = 1000$*:*

$$\begin{array}{rcrcrcr} P & = & 10 & \rightarrow & \textit{speed-up} & = & 9.2 \\ P & = & 100 & \rightarrow & \textit{speed-up} & = & 50.0 \\ P & = & 1000 & \rightarrow & \textit{speed-up} & = & 83.4 \end{array}$$

8. **Algorithm 2** (**15 points**)
   I also sketched an implementation that only requires sending $\sqrt{P}$ messages of size N^2/P per processor. Using

the same assumptions for compute time and communication time as for question 7, how many times faster is this improved parallel version than the sequential one for $P = 10$, $P = 100$, and $P = 1000$? As before, use $N = 1000$ for all three values of P.

Solution: *This very similar to question 7, except that there are $\sqrt{P}$ communication actions instead of $(P - 1)$. This yields:*

$$\textit{speed-up} = \frac{P}{1 + \frac{10\sqrt{P}}{N^3}(100P + N^2)}$$

$N = 1000$*:*

$$\begin{aligned} P &= 10 \rightarrow \textit{speed-up} = 9.7 \\ P &= 100 \rightarrow \textit{speed-up} = 90.8 \\ P &= 1000 \rightarrow \textit{speed-up} = 741.9 \end{aligned}$$