Cody Adams

CSC478-01

Exam 2

1. [--done

Transaction memory (TM) works in the same way as traditional locks do to manage shared resources and synchronize processes/threads, but is used in an abstract way. Rather than requiring sequential code to be re-written to be parallelized, it allows programmers to specify a block of code that needs atomicity. Where programmers using traditional locks need to use great care to eliminate race conditions, TM allows multiple simultaneous access attempts to shared memory locations in an atomic way. While the exact methods vary based on implementation, TM essentially runs all code “optimistically--that is, fully parallelized as though no contention is possible. If contention is detected, then the violating thread/process, and all of the work it has done, is abandoned and the TM implementation is responsible for ensure that its abandonment is transparent to the other processes/threads. Conversely, TM is also responsible for committing successful execution to shared memory in a way that does not negatively affect any other processes/threads. This is possible by flagging a section of code as “transactional”. Once this is done, all execution is done speculatively and then checked for contention. If contention did not in fact occur, the speculative writes are committed, if not, all work is abandoned.

In addition to providing a higher level of abstraction and allowing for less code modification when parallelizing, TM is better than traditional locks in most cases because contention is usually rare in reality. When using traditional locks, only one process/thread can hold a lock at a time, but TM can allow more than one thread atomic access to shared resources. TM is also deadlock free by nature, which is an obvious advantage over traditional threads.

For all of its advantages, TM does not scale as well as parallel programming using good locking techniques.

]

2. [--done

Do x = 1, n, 8

l = min(n, x+7)

for i = k to l {

fork stuff;

}

stuff() {

if (a[i] != b[i])

a[i] = a[i] \* b[i] / (a[i] - b[i]);

}

]

3. [--done

shared scale, M, x0, x1, x[N], h[M];

private i, j;

scale = M/(x1 - x0);

for i = 1 to N-1 fork DOELEMENT;

i = N;

DOELEMENT:

j = leastint(scale\*(x[i] - x0) );

critical

h[j] = h[j] + 1;

end critical

join(N);

]

4.

[--done

(a) data parallel model – data is partitioned into sections to improve locality or easily map data to tasks where the same operations are to be performed on different data. Generally, inputs and outputs are mapped to matrices and the multiplicative decompositions map to threads.

(b) work pool model – used where a significant amount of work is available to keep a pool of worker threads living and just reassigning them to new tasks. Beneficial if overhead of creating and destroying threads is greater than work to be done. In addition, the amount of data used needs to be small enough that the overhead involved in moving the data around is not too high of a price. Example: Web server, where creating a thread for each request would not make sense.

(c) master-slave model – master process generates work and assigns it to slave processes. If data has critical zones, the master is responsible for enforcing synchronization. Data should be assigned to slaves such that the slaves stay busy as much as possible—of course; they’re slaves.

(d) task graph model – tasks are group by common data dependencies in a task-dependency graph and then statically mapped to promote locality. Used when the amount of data is large relative to the tasks to be performed.

(e) pipeline or producer/consumer model – data is processed consecutively by multiple task such that the data travels through a pipeline of processes, from one to the next. It is also called producer/consumer because it can be demonstrated that each process produces data for the next to consume, where each task has its turn as producer and consumer. Data size must be carefully balanced against synchronization penalty (small amounts of data being processed quickly require more synchronization).

]

5.

[--done

S(N) = time 1 PE / time N PEs = N / ((N-1)N/2)

E(N) = S(N)/N = [ N / ((N-1)N/2) ] / N

Since this is not algebra class, I will not foul up what I believe to be a sound answer with algebra I do not understand. It makes sense to me. I hope it is right.

]

6.

[--done

1. r∞ = 1 / clock speed = 20 GFLOPS
2. n1/2 = k – 1 = 9

]

7.

[--done

The variable should be passed by value and not by reference since (a) the function does not write back to it and (b) by value copies the data into the function, thereby eliminating contention.

]

8.

[--done

In the modified code below, I believe that process k will perform m multiply-adds, since there are m processes that run from 1 to m. each process does m, for a total of m\*m multiply-adds for all n processes.

procedure parmain(value id, P)

shared P, n, a[n, n],x[n], c[n];

private id, i, j, sum, priv;

forall i := 1 step m until n

forall j := 1 step 1 until m

void x[i];

barrier;

forall i := 1 step m until n

begin

sum := c[i];

for j := 1 step 1 until m {

copy x[j] into priv;

sum := sum + a[i, j]\*priv;

}

produce x[i] := sum;

end

barrier;

end procedure

]