CSCI 654 – Advanced Computer Architecture
Homework 4

Due: October 14, 2016

Alexander Powell

1. **Implement the classical test-and-set instruction using the *load-linked/store-conditional* instruction pair.**

   The test-and-set instruction using load-linked and store-conditional would look something like the following which would be placed inside a loop with a stopping condition of R3 having a value of 1.

```
MOV R3, 1
LL R2, 0(R1)
SC R3, 0(R1)
```

2. (a) **At the end of the code segment, what is the value you would expect for C?**
   The value of C should be 2000, since it will have the same value as A.

   (b) **A system with general-purpose interconnection network, a directory-based cache coherence protocol, and support for non-blocking loads generates a result where C is 0. Describe a scenario where this result is possible.**
   This result could be possible if the write to $flag$ in P2 was somehow faster than the write to A.

   (c) **If you wanted to make the system sequentially consistent, what are the key constraints you would need to impose?**
   One would need to make sure that memory operations are atomic and a barrier should be placed right after the write to A and before the write to $flag$.

3. (a) As it stands, the given code cannot be parallelized without first making some modifications. The main problem with the code if the dependency created from remembering the value of pi from the previous iteration, to determine when the series begins to converge. Also, it's easier to parallelize if the value of $sign$ was determined based on the iterating variable, i, instead of flipping the sign from the previous iteration. It would be much simpler to parallelize by changing the while loop into a for loop, and simple iterate an arbitrarily larger number of times (say 10 million). In that case it becomes a simple reduction on the pi variable, which can easily be multi-threaded. The changed code will look something like this:

```
#pragma omp parallel private(i)
{
    #pragma omp for reduction(+:pi) schedule(static)
    for (i = 0; i < 10000000; i++)
    {
        if (i%2 == 0) // i is even
            pi += 1.0/(2 * i + 1);
        else // i is odd
            pi += -1.0/(2 * i + 1);
    }
}
pi = 4 * pi;
```

(b) The table below shows the execution times of the algorithm above with the number of threads ranging from 1 to 16. In this case, the number of iterations of the for loop was 100000000. The times listed are in microseconds.

# Threads	Exec Time	Speedup
1	1017844	1
2	564696	1.8
3	395932	2.57
4	313846	3.24
5	262619	3.88
6	238164	4.27
7	209364	4.86
8	203396	5.0
9	190259	5.35
10	176321	5.77
11	151033	6.74
12	141701	7.18
13	133361	7.63
14	126000	8.08
15	116863	8.71
16	129289	7.87

As you can see, we observe quite a noticeable speedup by invoking a multi-threaded model.

4. (a) 1000×10000 **inner loop**

Number of Threads	Time (microseconds)	Speedup
1	48246	1
2	26685	1.81
3	18933	2.55
4	17549	2.75
5	16452	2.93
6	14830	3.25
7	14824	3.25
8	13054	3.7
9	12012	4.02
10	14317	3.37
11	11265	4.28
12	9304	5.19
13	8590	5.62
14	7611	6.34
15	6036	7.99
16	6061	7.96

10000×1000 **inner loop**

Number of Threads	Time (microseconds)	Speedup
1	61867	1
2	38606	1.6
3	51269	1.21
4	51838	1.19
5	48501	1.28
6	38251	1.62
7	25776	2.4
8	21835	2.83
9	18364	3.34
10	16394	3.77
11	15738	3.93
12	16543	3.74
13	14058	4.4
14	12954	4.78
15	13531	4.57
16	11560	5.35

(b) 1000×10000 **outer loop**

Number of Threads	Time (microseconds)	Speedup
1	50007	1
2	35685	1.4
3	26785	1.87
4	20545	2.43
5	15717	3.18
6	14118	3.54
7	12124	4.12
8	10625	4.71
9	9617	5.2
10	8615	5.8
11	7987	6.26
12	7378	6.78
13	6953	7.19
14	6493	7.7
15	6036	8.28
16	5747	8.7

10000×1000 **outer loop**

Number of Threads	Time (microseconds)	Speedup
1	41391	1
2	36128	1.15
3	29801	1.39
4	27230	1.52
5	25338	1.63
6	18490	2.24
7	18059	2.29
8	17079	2.42
9	13689	3.02
10	19252	2.15
11	13944	2.97
12	14445	2.87
13	13749	3.01
14	10687	3.87
15	12264	3.38
16	11557	3.58

(c) So, from observing the results of the tables above, threading the inner and the outer loops both display decent speedups. As more threads are added, the results are more noticeable in the 1000×10000 matrix-vector multiplications. These tests were performed on the bg4 machine. When queried, it says that there are 32 cores present, although I believe there are really only 16 since hyperthreading is present. Also, it should be noted that the formula used for speedup is $\frac{\text{Time}_{\text{orig}}}{\text{Time}_{\text{threaded}}}$.

5. (a) To parallelize the loop, it was broken into three separate loops inside a parallel region. Below is the segment of code incorporating openMP directives.

```
#pragma omp parallel
{
    const int ithread = omp_get_thread_num();
    const int nthreads = omp_get_num_threads();

    float sum = 0;
    #pragma omp for schedule(static)
    for (int i=0; i<N; i++) {
        sum += A[i];
        A[i] = sum;
    }
    sum_a[ithread+1] = sum;
    #pragma omp barrier
    float offset = 0;
    for(int i=0; i<(ithread+1); i++) {
        offset += sum_a[i];
    }
    #pragma omp for schedule(static)
    for (int i=0; i<N; i++) {
        A[i] += offset;
    }
}
```

(b) The table below displays the performance results from the prefix sum algorithm run sequentially and parallelized with 8 threads. As you can see, the overhead from incorporating openMP directives begin to be amortized after the size of A grows larger than one million. Both times are given in microseconds.

N	**Sequential Time**	**Threaded Time**
10	49	3280
100	59	7252
1000	68	3359
10000	136	3774
100000	954	8977
1000000	9312	10347
10000000	93048	26797
100000000	953620	148621