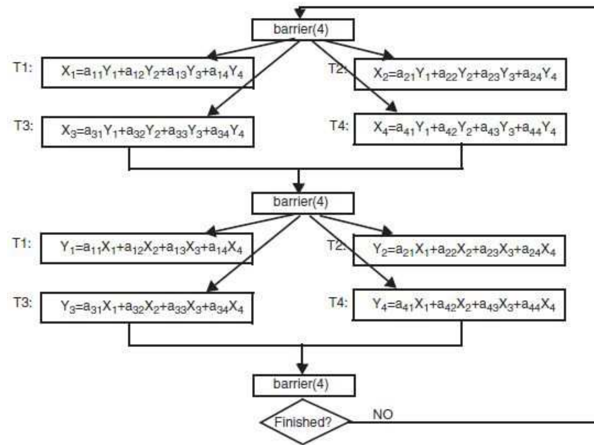
HIGH PERFORMANCE COMPUTER ARCHITECTURE 18-12-2015 MATRICULATION NO.__________________

(FINAL TEST) SURNAME____________________

FIRST NAME____________________

1) (POINTS 20/40) Consider the four-iterate Jacobi method shown as in the following flowchart.



In this flowchart, during every iteration four threads concurrently compute the new values of a vector X_i as a linear function of Y_i values. Then the threads use a barrier synchronization.
Then the four Y_i values are concurrently computed using the X_i values generated prior to the barrier. The threads then wait again to perform a convergence test. If a specific convergence criterion is met, the program exits; otherwise it loops back to the X_i computation.
(1a) First, transform the flowchart into a pseudocode for a parallel algorithm using a barrier synchronization primitive (e.g., with a “BARRIER(BAR)”). Assume that at the end of the second barrier the convergence test function is executed as a single thread before returning to the loop or exiting the program.
(1b) Transform the flowchart into an OpenMP parallel program using compiler directives.
(1c) Transform the flowchart into a P-Thread parallel program using the P-Thread API.
(1d) Transform the flowchart into a Cilk parallel program using the cilk_sync, cilk_spawn, cilk_for directives.

2) (POINTS 20/40)
This problem is about the sensitivity of cache misses to actual timing and to the memory consistency model. We use the Jacobi algorithm and its overhead under various cache coherence protocols to demonstrate this point. A flowchart for this algorithm is shown above.
Assume that the caches have infinite size (i.e., no capacity misses and no conflict misses) and that the algorithm has been running for a while (i.e., no cold misses). However, because of the cache coherence protocol, we now have only coherence misses, i.e., misses due to invalidations (the fourth C in the classification) in invalidation-based protocol or updates in update-based protocols. Because matrix A is read-only, accesses to matrix A will not miss at all. We also ignore misses due to barrier synchronization. Hence, in this problem we focus on accesses to X and Y. All the components of X hold in the same cache block, and all the components of Y hold in the same cache block. Because X and Y are shared writable variables, they have been declared as volatile so that they are not allocated in register by the compiler.
The sequence of memory accesses in thread 1 (T1) to Y and X (in process order) is as follows:

rY1, wX1, rY2, wX1, rY3, wX1, rY4, wX1, rX1, wY1, rX2, wY1, rX3, wY1, rX4, wY1, . . . ,

where r means “read” and w means “write.” The sequences of accesses to X and Y by T2, T3, and T4 are similar. We consider three protocols: MSI-invalidate, MSI-update, MESI protocols.

(2a) Assume first that the system is sequentially consistent. Processors run at exactly the same speed and must globally perform their memory accesses one by one so that the four threads interleave their access to X and Y round-robin. For instance, processors each execute their read of Y1 in turn first, then they execute their write to X in turn, etc. What is the number of coherence misses in all processors for one iteration of the entire loop of Jacobi, for MSI, and MESI invalidate protocols? What is the number of updates in MSI-update?
(2b) Repeat (2a) for different timing. We assume that the system is still sequentially consistent and globally performs each access one at a time. However, the global interleaving of accesses is different. The timing is such that, in both the first and second phases of the iteration, threads execute all their memory access in turn. First T1, then T2, then T3, and finally T4. Then in the second phase we have the same behavior: T1, T2, T3, and T4.

HIGH PERFORMANCE COMPUTER ARCHITECTURE 18-12-2015
(SOLUTION)

EXERCIZE 1)

1a) PSEUDOCODE

```
while (true) {
   BARRIER(BAR); // BAR was initialized for 4 threads.
   parallel_for (i=1; i<=4; ++i) {
      Xi = ai1*Y1 + ai2*Y2 + ai3*Y3 + ai4*Y4
   }
   BARRIER(BAR);
   parallel_for (i=1; i<=4; ++i) {
      Yi = ai1*X1 + ai2*X2 + ai3*X3 + ai4*X4
   }
   BARRIER(BAR);
   // Perform convergence test in a single thread.
   if (convergence_test()) break;
}
```

1b) OPENMP

```
#include <omp.h>
…
while (true) {
   #pragma omp barrier
   #pragma omp parallel for private(i)
   for (i = 0; i < 4; ++i) {
      X[i] = a[i*4+1]*Y[0] + a[i*4+2]*Y[1] + a[i*4+3]*Y[2] + a[i*4+4]*Y[3];
   }
   #pragma omp barrier
   #pragma omp parallel for private(i)
   for (i = 0; i < 4; ++i) {
      Y[i] = a[i*4+1]*X[0] + a[i*4+2]*X[1] + a[i*4+3]*X[2] + a[i*4+4]*X[3];
   }
   #pragma omp barrier
   // Perform convergence test in a single thread.
   if (convergence_test()) break;
}
```

1c) P-THREAD

```
#include <pthread.h>
…
typedef struct { double *x; double *y; double *a;} thread_args;
void *jacobi(void *ptr)
{
    int j; double *x = ((thread_args*)ptr)->x; double *y = ((thread_args*)ptr)->y;
    double *a = ((thread_args*)ptr)->a;
    for (j = 0; j < N; ++j) *x -= a[j] * y[j];
}

While (true) {
    thread_args args1[N], args2[N]; pthread_t thread1[N], thread2[N]; int i;
    for (i = 0; i < N; ++i) {
       args1[i].x = y+i; args1[i].y = x; args1[i].a = R+N*i;
       args2[i].x = x+i; args2[i].y = y; args2[i].a = R+N*i;
    }
    do {
       for (i = 0; i < N; ++i) pthread_create(&thread1[i], NULL, jacobi, (void*) &args1[i]);
       for (i = 0; i < N; ++i) pthread_join(thread1[i], NULL);
       for (i = 0; i < N; ++i) pthread_create(&thread2[i], NULL, jacobi, (void*) &args2[i]);
       for (i = 0; i < N; ++i) pthread_join(thread2[i], NULL);
    } while (convergence_test());
}
```

1d) CILK

```
#include <cilkplus.h>
…
while (true) {
   cylk_sync;
   cilk_for (i = 0; i < 4; ++i) {
      X[i] = a[i*4+1]*Y[0] + a[i*4+2]*Y[1] + a[i*4+3]*Y[2] + a[i*4+4]*Y[3];
   }
   cylk_sync;
   cilk_for (i = 0; i < 4; ++i) {
      Y[i] = a[i*4+1]*X[0] + a[i*4+2]*X[1] + a[i*4+3]*X[2] + a[i*4+4]*X[3];
   }
   cylk_sync;
   // Perform convergence test in a single thread.
   if (convergence_test()) break;
}
```

EXERCISE 2)

We can represent the memory accesses by each thread in a table as follows. Considering also that accesses to Y1, Y2, … are to the same block we can just write Y and similarly for X1, X2, … we can just write X.
We consider a generic iteration after caches are warmed up.

In the case 2a) we can find an equivalent sequential consistent order as specified by the question by "reading" the following table **by rows**:

THREAD1 (P1)	THREAD2 (P2)	THREAD3 (P3)	THREAD4 (P4)
rY	**rY**	**rY**	**rY**
wX	**wX**	**wX**	**wX**
rY	rY	rY	rY
wX	wX	wX	wX
rY	rY	rY	rY
wX	wX	wX	wX
rY	rY	rY	rY
wX	wX	wX	wX

Then:

rX	**rX**	**rX**	**rX**
wY	**wY**	**wY**	**wY**
rX	rX	rX	rX
wY	wY	wY	wY
rX	rX	rX	rX
wY	wY	wY	wY
rX	rX	rX	rX
wY	wY	wY	wY

MSI-invalidate)
In the first part of accesses each cache has a shared copy of X while P4 has a modified copy of Y. So, initially (first row) there will be 3 misses on Y (P4 does not miss). Each of the following writes to X reloads X block and invalidates other caches, thus causing 4 misses (second row). Afterwards, there will be no miss on Y which is only read, but there will be again 4 misses on X (fourth, sixth, eight row). So in total there will be 3+4x4=19 misses. In the second part, X and Y exchange roles so there will be other 19 misses. **In total 38 misses**.
MESI)
The situation is similar to MSI-invalidate since the E state is never visited: the protocol behaves exactly like MSI-invalidate. **In total 38 misses.**
MSI-update)
All copies are constantly updated on writes so the coherence transactions are the write-update: 4x4 in the first part and other 4x4 in the second part. **In total 32 updates.**

In the case 2b) we can find an equivalent sequential consistent order as specified by the question by "reading" the following table **by columns**:

THREAD1 (P1)	THREAD2 (P2)	THREAD3 (P3)	THREAD4 (P4)
rY	**rY**	**rY**	**rY**
wX	**wX**	**wX**	**wX**
rY	rY	rY	rY
wX	wX	wX	wX
rY	rY	rY	rY
wX	wX	wX	wX
rY	rY	rY	rY
wX	wX	wX	wX

Then:

rX	**rX**	**rX**	**rX**
wY	**wY**	**wY**	**wY**
rX	rX	rX	rX
wY	wY	wY	wY
rX	rX	rX	rX
wY	wY	wY	wY
rX	rX	rX	rX
wY	wY	wY	wY

MSI-invalidate)
In the first part of accesses each cache has a shared copy of X while P4 has a modified copy of Y. So, initially (first column) there will be 1 miss on X and 1 miss on Y. Similarly for the second and third column. In the fourth column P4 does not miss on Y which is still there, so we have only the miss on X. These are 7 misses.
In the second part, X and Y exchange roles so there will be other 7 misses. **In total 14 misses**
MESI)
The situation is similar to MSI-invalidate since the E state is never visited: the protocol behaves exactly like MSI-invalidate. **In total 14 misses.**
MSI-update)
All copies are constantly updated on writes so the coherence transactions are the write-update: 4x4 in the first part and other 4x4 in the second part. **In total 32 updates.**