Fall 2014 Computer Architecture Test 2 Name: Mark F.

Question 1. (15 points) When writing a parallel program (e.g using pthreads) work can be statically allocated to each thread initially, or dynamically allocated (e.g., via a work queue) as the computation proceeds.

a) When does it make sense to allocate work statically?

When work can be allocated to threads evenly to avoid idle processors.

b) When does it make sense to allocate work dynamically?

When the amount of work cannot be determined or created dynamically, e.g., TSP - some subtrees have more pruning

Question 2. (20 points) Consider the correct `barrier` code (given in HW #6) uses the condition variable `all_here` and global variables:

- `count` - used to count the number of threads that have arrived at the barrier (initially set to 0), and
- `t` - used to hold the total number of threads to wait for at the barrier

```
void barrier(long id) {
  pthread_mutex_lock(&barrier_lock);
  count++;
  // printf("count %d, id %d\n", count, id);
  if (count == t) {
    count = 0;
    pthread_cond_broadcast(&all_here);
  } else {
    while(pthread_cond_wait(&all_here, &barrier_lock) != 0);
  } // end if
  pthread_mutex_unlock(&barrier_lock);
} // end barrier
```

a) Why does the `pthread_cond_wait` call need the `&barrier_lock` parameter?

5

Needs to release the barrier_lock so other threads can get into the "top" mutex lock.

b) What's the purpose of the `while`-loop in the `else`?

5 In case the threads "erroneously" get signaled and return an error code (non-zero), then they "wait" again.

c) Consider the following barrier implementation. What's wrong with this implementation?

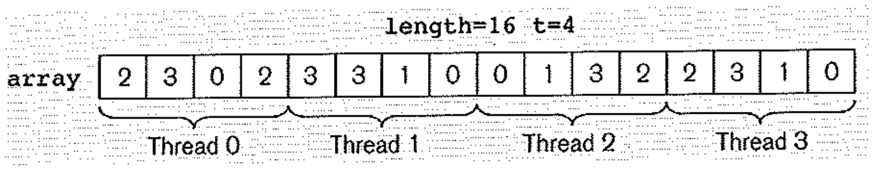
```
void barrier(long id) {
  pthread_mutex_lock(&barrier_lock);
  count++;
  pthread_mutex_unlock(&barrier_lock);
  // printf("count %d, id %d\n", count, id);
  while(count < t);  // NOTE: EMPTY LOOP BODY
  if (id == 0) {
    count = 0;
  } // end if
} // end barrier
```

10

The code is inefficient due to busy-wait in while-loop, but more importantly it has a race-condition if the barrier is used in a loop. The global variable "count" is changed by thread 0 without a mutex lock

35

Question 3. Consider two 1D array summation programs both using a block allocation of the array to threads:



```
void * threadPartialSumA(void * args) {

 int i;
 int start_index = ((RANGE *) args)->start_index;
 int end_index = ((RANGE *) args)->end_index;
 int threadId = ((RANGE *) args)->threadId;
 float * threadSums = ((RANGE *) args)->threadSums;
 float * myArray = ((RANGE *) args)->arrayToSum;
 float localSum = 0.0;
 for (i=start_index; i <= end_index; i++) {
    localSum += myArray[i];
 } /* end for (i */
 threadSums[threadId] = localSum;

} // end threadPartialSumA
```

```
void * threadPartialSumB(void * args) {

 int i;
 int start_index = ((RANGE *) args)->start_index;
 int end_index = ((RANGE *) args)->end_index;
 int threadId = ((RANGE *) args)->threadId;
 float * threadSums = ((RANGE *) args)->threadSums;
 float * myArray = ((RANGE *) args)->arrayToSum;
 threadSums[threadID] = 0.0;
 for (i=start_index; i <= end_index; i++) {
    threadSums[threadID] += myArray[i];
 } /* end for (i */

} // end threadPartialSumB
```

a) (10 points) Why do neither versions need to use a mutex when the threads are updating the threadSums array with their partial sums? Each thread has its own index (i.e., its threadId) so only one thread is changing any threadSums item.

10

b) (10 points) The `threadPartialSumB` code potentially suffers from *false-sharing* forcing a thread to access main memory frequently instead of being able to keep its `threadSums[threadID]` value in its cache. Why does `threadPartialSumA` not suffer from *false-sharing*?

threadPartialSumA updates a local variable (localSum) on its run-time stack. Since each thread has its own stack they are not in the same cache line as in threadPartialSums.

10

Question 4. (25 points) My Homework #6 (2D SOR) solution's `do-while` loop in `thread_main` was:

```
void* thread_main(void * arg) {
   ...
   do {
     ...

     pthread_mutex_lock(&update_lock);
     if (maxLocalDelta > globalDelta) {
        globalDelta = maxLocalDelta;
     } // end if
     pthread_mutex_unlock(&update_lock);

     barrier(id);
   } while (globalDelta > threshold);
} // end thread_main
```

25

a) Why is the mutex used when updating the `globalDelta`?

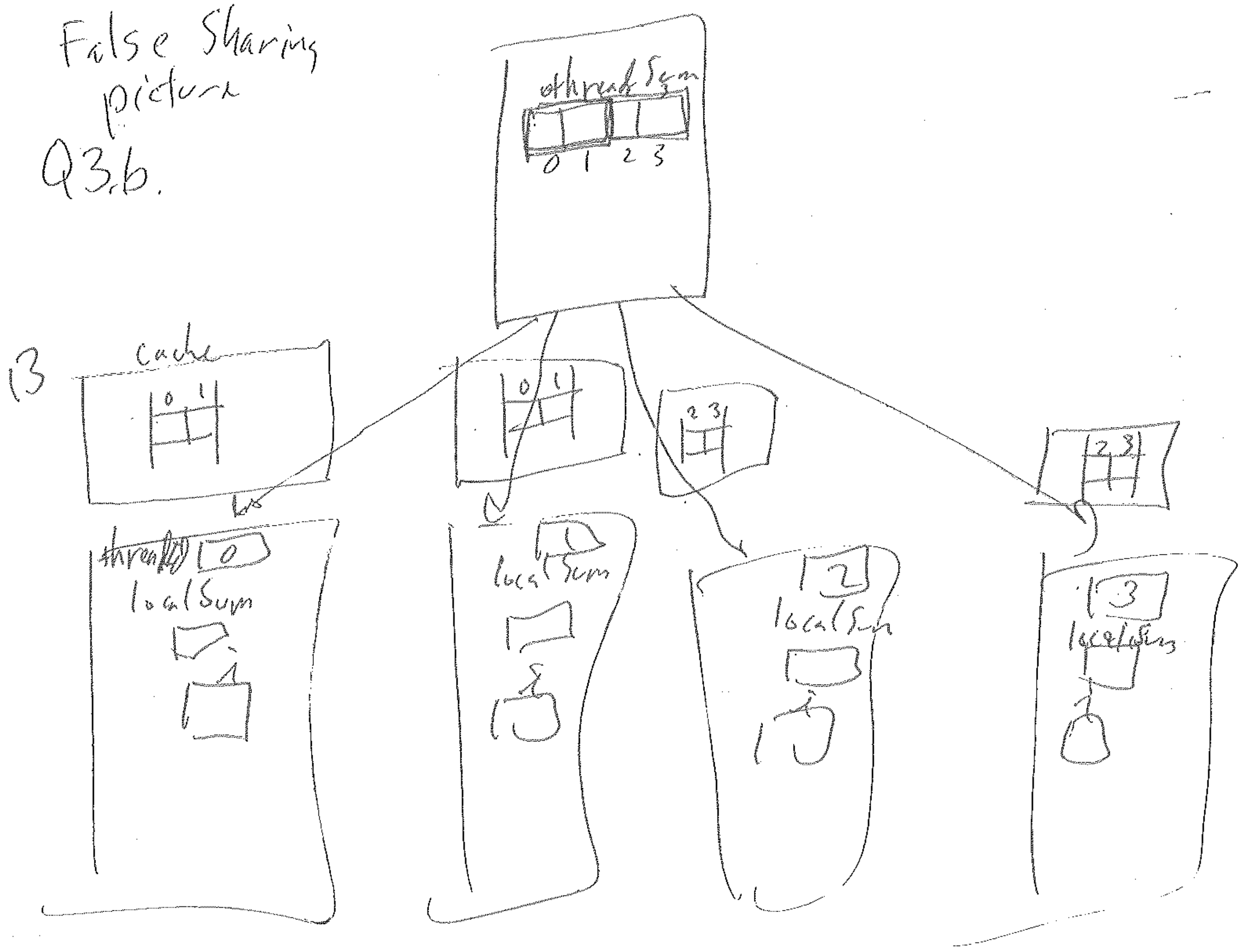
12 The globalDelta variable needs to be updated mutually exclusively or it might be wrong.

b) Why is the barrier necessary before the `do-while` condition is checked?

13 To make sure all threads have updated globalDelta before any thread checks the while-loop condition, so all threads loop again or all drop-out together.

45

False Sharing
picture
Q3.b.



Question 5. (20 points) We have looked at the 2-D SOR (Successive Over-Relaxation) problem in HW #6. In this problem consider solving a 3-D SOR problem of size n x n x n (e.g., 1024 x 1024 x 1024) interior floating-point values all initially 0.0. The surrounding boundary values are always 1.0s across one face of the cube and always 0.0s across the other five faces. The next interior value is the average of its 6 "neighboring" values.

a) Before creating the threads, the **2-D SOR** main program allocates the `val` and `new` arrays by:

```
new = (double **) malloc((n+2)*sizeof(double *));
val = (double **) malloc((n+2)*sizeof(double *));

for (i = 0; i < n+2; i++) {
  new[i] = (double *) malloc((n+2)*sizeof(double));
  val[i] = (double *) malloc((n+2)*sizeof(double));
} // end for i
```

What would be the C code for the **3-D SOR** allocation of the `val` and `new` arrays?

```
new = (double ***) malloc((n+2)*sizeof(double **));
val = (double ***) malloc((n+2)*sizeof(double **));

for (i=0; i<n+2; i++) {
   new[i] = (double **) malloc((n+2)*sizeof(double *));
   val[i]=(double **) malloc((n+2)*sizeof(double *));
   for (j=0; j<n+2; j++) {
       new[i][j]= (double *) malloc((n+2)*sizeof(double));
       val[i][j]= (double *) malloc((n+2)*sizeof(double));
   } // end for j
} // end for i
```

7

b) On an 8-core processor desktop computer, how many pthreads you would use to solve this problem? (Justify you answer)

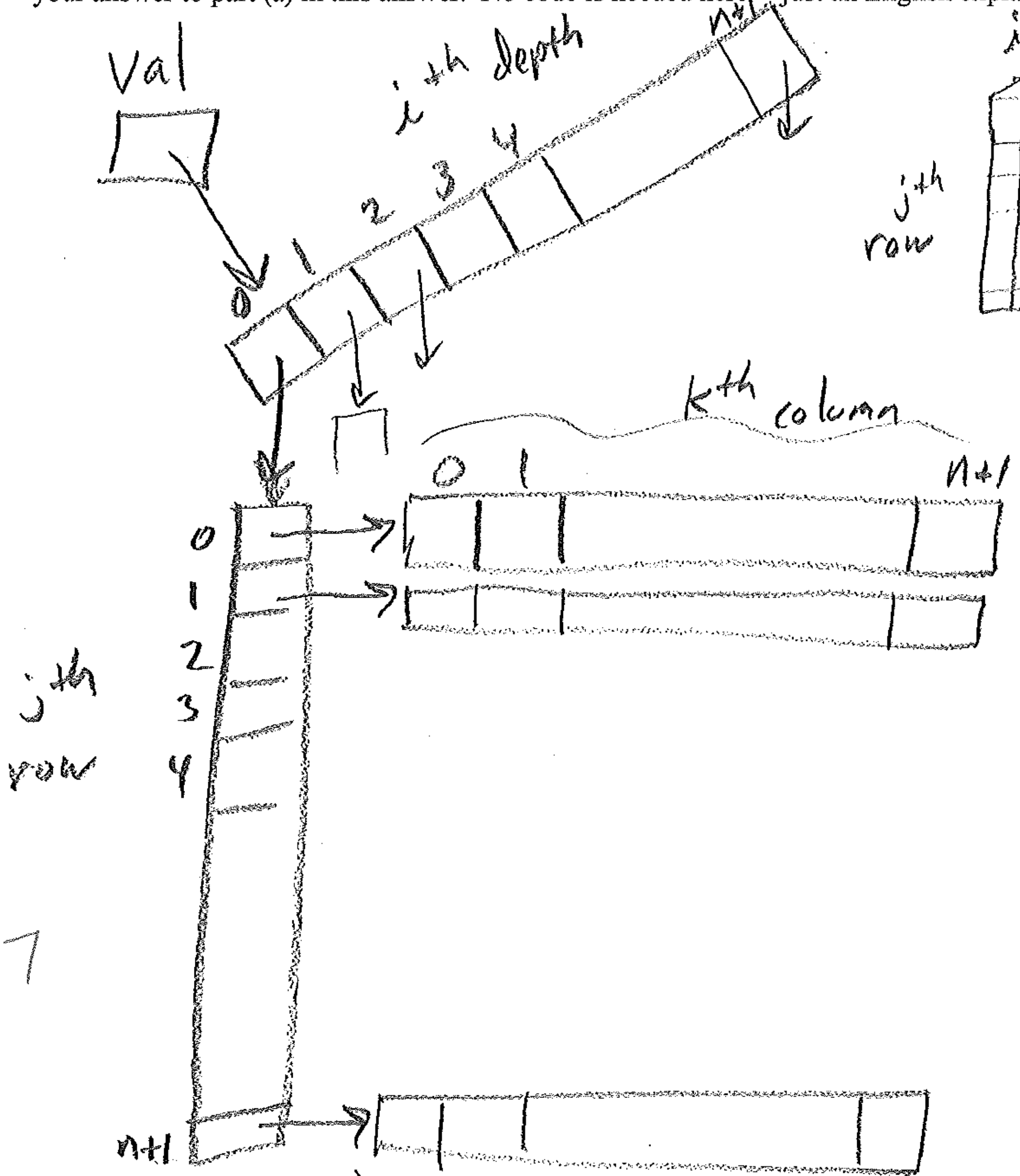
8 pthreads so there is one per core. If < 8 pthreads, then some cores sitting idle. If > 8 pthreads, then multiple pthreads assigned to a core(s). Since only one pthread can execute on a core at a time, some pthreads are idle.

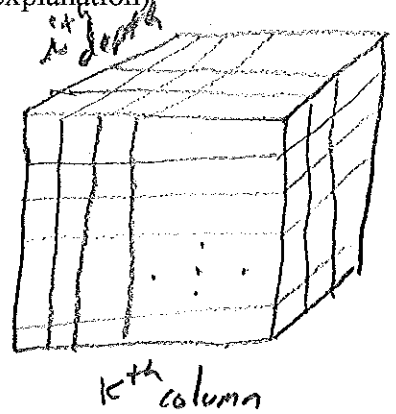
6

13

(test continued on the next page)

c) On an 8-core processor desktop computer, how would you partition the work among the pthreads? (Consider your answer to part (a) in this answer. No code is needed here -- just an English explanation)

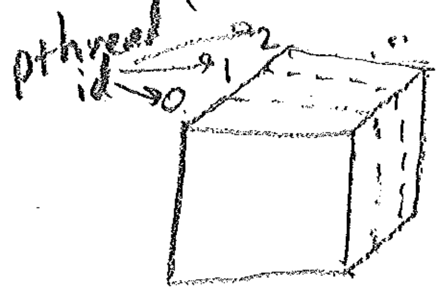




Main thing is to have whole "rows" (last index varies) assigned to a pthread

Two possibilities:

(1) blocks of whole "depths"



(2) blocks of whole "rows"