Exercise 4.1: When we discussed matrix-vector multiplication, we assumed that both m and n, the number of rows and the number of columns, respectively, were evenly divisible by t, the number of threads. How do the formulas for the assignments change if this is not the case?

This is the original code for matrix multiplication:

```c
void* Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m / thread_count;
    int my_first_row = my_rank * local_m;
    int my_last_row = (my_rank + 1) * local_m - 1;
    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++) {
            y[i] += A[i][j] * x[j];
        }
    }
    return NULL;
} /* Pth_mat_vect */
```

Answer: In the case where t is not evenly divisible by m we must do the following modifications to the algorithm:

```c
void * Pth_mat_vect(void * rank) {
  long my_rank = (long) rank;
  int i, j;
  int my_first_row, my_last_row;
  get_offset(my_rank, thread_count, m,
    &my_first_row, &my_last_row);

  for (i = my_first_row; i < my_last_row; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++) {
      y[i] += A[i][j] * x[j];
    }
  }
  return NULL;
} /* Pth_mat_vect */

void get_offset( int rank,    /* Input */
                 int threads, /* Input */
                 int rows,    /* Input */
                 int * start, /* Output */
                 int * end    /* Output */ ) {
  int r = rows % threads;
  int offset, step, idx;
  if (rank < r) {
    offset = 0;
    step = (rows / threads) + 1;
    idx = rank;
  } else {
    offset = ((rows / threads) + 1) * r;
    step = rows / threads;
    idx = rank - r;
  }
  *start = offset + (step * idx);
  *end = (*start) + step;
} /* get_offset */
```

The new algorithm boils down to the following concepts. If the number of rows m is larger than the the number of threads (t), then each thread is assigned at least one row. Then, the remainder of rows is distributed one by one among all threads starting with the first one. Therefore every thread will have either $\frac{m}{t}$ or $\frac{m}{t}+1$ rows assigned to it. In the case where m is smaller than t then $t-m$ threads will be idle while the restcomputes one row.

Exercise 4.2 If we decide to physically divide a data structure among the threads, that is, if we decide to make various members local to individual threads, we need to consider at least three issues:

a. How are the members of the data structure used by the individual threads?

b. Where and how is the data structure initialized?

c. Where and how is the data structure used after its members are computed?

We briefly looked at the first issue in the matrix-vector multiplication function. We saw that the entire vector x was used by all of the threads, so it seemed pretty clear that it should be shared. However, for both the matrix A and the product vector y, just looking at (a) seemed to suggest that A and y should have their components distributed among the threads. Let's take a closer look at this.

What would we have to do in order to divide A and y among the threads? Dividing y wouldn't be difficult–each thread could allocate a block of memory that could be used for storing its assigned components. Presumably, we could do the same for A–each thread could allocate a block of memory for storing its assigned rows. Modify the matrix-vector multiplication program so that it distributes both of these data structures. Can you "schedule" the input and output so that the threads can read in A and print out y? How does distributing A and y affect the run-time of the matrix-vector multiplication? (Don't include input or output in your run-time.)

Answer: Below are two implementations for a matrix multiplier with local and global variables. The first program stores A and y as global variables which are shared across threads. The second program has one local variable A and one local y for each thread.

The input and output can be synchronized by using semaphores. We first initialize one semaphore for every worker thread. Then we call `sem_wait` on all semaphores. We then setup each thread to request a `sem_wait` on their own semaphore before starting to collect input. This means that every thread will block because each semaphore has already being decreased in the initial setup. Next, the main thread posts to the first semaphore belonging to the first thread. Upon completing collecting input, the first thread posts the semaphore for the next thread. This creates a chain reaction that will trigger every thread in rank order. Resulting in all threads collecting input in the correct order. The same strategy is used for printing the output.

After running tests with the two programs, I collected the following data:

	Matrix Dimension (μ *seconds*)					
	800 x 800		80,00 x 8		8 x 80,000	
Threads	Local	Global	Local	Global	Local	Global
1	3662	4642	4164	4292	3736	3931
2	2093	2128	2320	2423	2091	2201
4	1289	1344	1445	1257	**1233**	**2757**
8	1294	1019	1313	1399	**1241**	**1709**
16	1170	597	1294	852	1727	2007

The above table shows that there is no distintc time difference for most matrix and thread number combination of the local vs global implementation. With the exception of the third matrix (8 x 80,000). This matrix turns out to be much slower in the global implementation. A likely cause for this is the false sharing occurring on y across all threads. Since in this matrix y only has 8 components it is likely that it fits in a single cache line. Then, as every thread updates y they invalidates the cache for the other threads. Thus causing an increase in execution time.

Problem 4.2 Global

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#include <semaphore.h>

void   pthread_barrier    ();
void   create_threads     ();
void   join_threads       ();
void   read_x             ();
void   read_a             ();
void   print_y            ();
void * Pth_mat_vect       (void * rank);
void   get_offset         (int rank, int threads, long rows, long * start,
    long * end);

int                 thread_count;
pthread_t         * t_ids;
sem_t             * semaphores;
long                m, n;
double            * x, * A, * y;
pthread_mutex_t     mutex;
pthread_cond_t      cond_var;
int                 counter = 0;

int main(int argc, char ** argv) {
  if (argc != 2) {
    printf("Invalid arguments\n");
    exit(1);
  }
  // Read m and n values
  scanf("%ld %ld", &m, &n);

  struct timeval start, end;
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond_var, NULL);
  thread_count = strtol(argv[1], NULL, 10);
  t_ids        = (pthread_t *) malloc(sizeof(pthread_t) * thread_count);
  x            = (double *) malloc(sizeof(double) * n);
  A            = (double *) malloc(sizeof(double) * (n * m));
  y            = (double *) malloc(sizeof(double) * m);

  read_x();
  read_a();

  create_threads();

  pthread_barrier();
  gettimeofday(&start, NULL);

  pthread_barrier();
  gettimeofday(&end, NULL);
  long t = (end.tv_sec * 1000000 + end.tv_usec) -
           (start.tv_sec * 1000000 + start.tv_usec);

  join_threads();

  print_y();
  printf("m: %ld n: %ld Time: %ld\n", m, n, t);

  free(x);
  free(t_ids);
  return 0;
}

void pthread_barrier() {
  pthread_mutex_lock(&mutex);
  counter++;
  if (counter == (thread_count + 1)) {
    counter = 0;
    pthread_cond_broadcast(&cond_var);
  } else {
    while (pthread_cond_wait(&cond_var, &mutex) != 0);
  }
  pthread_mutex_unlock(&mutex);
}

void create_threads() {
  long i, err;

  for (i = 0; i < thread_count; i++) {
    err = pthread_create(&t_ids[i], NULL, Pth_mat_vect, (void *) i);
    if (err) {
      perror("Could not create thread");
      exit(1);
    }
  }
}

void join_threads() {
  int i, err;
  for (i = 0; i < thread_count; i++) {
    err = pthread_join(t_ids[i], NULL);
    if (err) {
      perror("Could not join thread");
      exit(1);
    }
  }
}

void read_x() {
  int i;
  for (i = 0; i < n; i++) {
```

```c
    scanf("%lf", &x[i]);
  }
}

void read_a() {
  int i;
  for (i = 0; i < (n * m); i++) {
    scanf("%lf", &A[i]);
  }
}

void print_y() {
  long i;
  for (i = 0; i < m; i++) {
    printf("y[%ld] %lf\n", i, y[i]);
  }
}

void * Pth_mat_vect(void * rank) {
  long my_rank = (long) rank;
  int i, j;
  long my_first_row, my_last_row;
  get_offset(my_rank, thread_count, m, &my_first_row, &my_last_row);
  pthread_barrier();

  // Calculate y value
  for (i = my_first_row; i < my_last_row; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++) {
      y[i] += A[i * n + j] * x[j];
    }
  }

  pthread_barrier();
  return NULL;
} /* Pth_mat_vect */

void get_offset( int rank, int threads, long rows, long * start, long * end
    ) {
  long r = rows % threads;
  long offset, step, idx;
  if (rank < r) {
    offset = 0;
    step = (rows / threads) + 1;
    idx = rank;
  } else {
    offset = ((rows / threads) + 1) * r;
    step = rows / threads;
    idx = rank - r;
  }
  *start = offset + (step * idx);
  *end = (*start) + step;
} /* get_offset */
```

Problem 4.2 Local

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#include <semaphore.h>

void   pthread_barrier    ();
void   init_semaphores    ();
void   lock_semaphores    ();
void   destroy_semaphores ();
void   create_threads     ();
void   join_threads       ();
void   read_x             ();
void * Pth_mat_vect       (void * rank);
void   get_offset         (int rank, int threads, long rows, long * start,
    long * end);

int                 thread_count;
pthread_t         * t_ids;
sem_t             * semaphores;
long                m, n;
double            * x;
pthread_mutex_t     mutex;
pthread_cond_t      cond_var;
int                 counter = 0;

int main(int argc, char ** argv) {
  if (argc != 2) {
    printf("Invalid arguments\n");
    exit(1);
  }
  // Read m and n values
  scanf("%ld %ld", &m, &n);

  struct timeval start, end;
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond_var, NULL);
  thread_count = strtol(argv[1], NULL, 10);
  t_ids         = (pthread_t *) malloc(sizeof(pthread_t) * thread_count);
  semaphores    = (sem_t *) malloc(sizeof(sem_t) * thread_count);
  x             = (double *) malloc(sizeof(double) * n);

  read_x();
  init_semaphores();
  lock_semaphores();
  sem_post(&semaphores[0]); // Start the first thread
  create_threads();

  pthread_barrier();
  gettimeofday(&start, NULL);

  pthread_barrier();
  gettimeofday(&end, NULL);
  long t = (end.tv_sec * 1000000 + end.tv_usec) -
           (start.tv_sec * 1000000 + start.tv_usec);

  join_threads();
  destroy_semaphores();

  printf("m: %ld n: %ld Time: %ld\n", m, n, t);

  free(x);
  free(t_ids);
  free(semaphores);
  return 0;
}

void pthread_barrier() {
  pthread_mutex_lock(&mutex);
  counter++;
  if (counter == (thread_count + 1)) {
    counter = 0;
    pthread_cond_broadcast(&cond_var);
  } else {
    while (pthread_cond_wait(&cond_var, &mutex) != 0);
  }
  pthread_mutex_unlock(&mutex);
}

void init_semaphores() {
  int i;
  for (i = 0; i < thread_count; i++) {
    sem_init(&semaphores[i], 0, 1);
  }
}

void lock_semaphores() {
  int i;
  for (i = 0; i < thread_count; i++) {
    sem_wait(&semaphores[i]);
  }
}

void destroy_semaphores() {
  int i;
  for (i = 0; i < thread_count; i++) {
    sem_destroy(&semaphores[i]);
  }
}

void create_threads() {
  long i, err;

```

```c
  for (i = 0; i < thread_count; i++) {
    err = pthread_create(&t_ids[i], NULL, Pth_mat_vect, (void *) i);
    if (err) {
      perror("Could not create thread");
      exit(1);
    }
  }
}

void join_threads() {
  int i, err;
  for (i = 0; i < thread_count; i++) {
    err = pthread_join(t_ids[i], NULL);
    if (err) {
      perror("Could not join thread");
      exit(1);
    }
  }
}

void read_x() {
  int i;
  for (i = 0; i < n; i++) {
    scanf("%lf", &x[i]);
  }
}

void * Pth_mat_vect(void * rank) {
  long my_rank = (long) rank;
  int i, j;
  long my_first_row, my_last_row;
  get_offset(my_rank, thread_count, m, &my_first_row, &my_last_row);

  // Initialize A array
  long num_rows = my_last_row - my_first_row;
  long arr_sz = num_rows * n;
  double * A = (double *) calloc(arr_sz, sizeof(double));
  double * y = (double *) calloc(num_rows, sizeof(double));

  // Read in array
  sem_wait(&semaphores[my_rank]);
  for (i = 0; i < arr_sz; i++) {
    scanf("%lf", &A[i]);
  }
  if (my_rank < thread_count - 1) {
    sem_post(&semaphores[my_rank + 1]);
  } else {
    sem_post(&semaphores[0]);
  }

  pthread_barrier();

  // Calculate y value
  for (i = 0; i < num_rows; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++) {
      y[i] += A[i * n + j] * x[j];
    }
  }

  pthread_barrier();

  // Output results
  sem_wait(&semaphores[my_rank]);
  for (i = 0; i < num_rows; i++) {
    printf("y[%2ld] %lf\n", i + my_first_row, y[i]);
  }
  if (my_rank < thread_count - 1) {
    sem_post(&semaphores[my_rank + 1]);
  }
  free(y);
  free(A);
  return NULL;
} /* Pth_mat_vect */

void get_offset( int rank, int threads, long rows, long * start, long * end
  ) {
  long r = rows % threads;
  long offset, step, idx;
  if (rank < r) {
    offset = 0;
    step = (rows / threads) + 1;
    idx = rank;
  } else {
    offset = ((rows / threads) + 1) * r;
    step = rows / threads;
    idx = rank - r;
  }
  *start = offset + (step * idx);
  *end = (*start) + step;
} /* get_offset */
```

Exercise 4.6 Modify the mutex version of the π calculation program so that it uses a semaphore instead of a mutex. How does the performance of this version compare with the mutex version?
This is the original mutex version of the π calculation:

```c
void * Thread_sum(void * rank) {
    long my_rank = (long) rank;
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n / thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0) {
        factor = 1.0;
    } else {
        factor = -1.0;
    }

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        my_sum += factor / (2 * i + 1);
    }

    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);

    return NULL;
} /* Thread_sum */
```

Answer: This is the modified version of the program using semaphores.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>
#include <semaphore.h>

double sum;
long thread_count;
long long n;
sem_t semaphore;

void * Thread_sem_sum(void * rank) {
    long my_rank = (long) rank;
    double factor;
    double my_sum = 0.0;
    long long i;
    long long my_n = n / thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0) {
        factor = 1.0;
    } else {
        factor = -1.0;
    }

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        my_sum += factor / (2 * i + 1);
    }

    sem_wait(&semaphore);
    sum += my_sum;
    sem_post(&semaphore);

    return NULL;
} /* Thread_sum */

int main(int argc, char ** argv) {
    if (argc != 2) {
        printf("Invalid parameters\n");
        exit(1);
    }

    int power = strtoll(argv[1], NULL, 10);
    n = (long long) pow(2.0, (double)power);
    sum = 0.0;
    thread_count = 16;
    long i, err;
    struct timeval start, end;
    pthread_t * t_ids =
        (pthread_t *) malloc(sizeof(pthread_t) * (thread_count - 1));

    sem_init(&semaphore, 0, 1);


    gettimeofday(&start, NULL);

    for (i = 0; i < thread_count - 1; i++) {
        err = pthread_create(&t_ids[i], NULL, Thread_sem_sum, (void *)i);

        if (err) {
            perror("Could not start thread");
            exit(1);
        }
    }

    Thread_sem_sum((void *) (thread_count - 1));

    for (i = 0; i < thread_count - 1; i++) {
        err = pthread_join(t_ids[i], NULL);
        if (err) {
            perror("Could not join thread");
            exit(1);
        }
    }

    gettimeofday(&end, NULL);
    printf("thread: %2ld n: %lld est: %2.10f time: %ld\n", thread_count, n,
        sum * 4,
        ((end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 +
           start.tv_usec)));

    return 0;
} /* main */
```

To compare the performance of the two versions I ran a simple test with increasing number of sums. The results are below:

Thread	n-size	estimate	Time (μ seconds)	
			Mutex	Semaphore
16	1024	3.1406160913	988	949
16	4096	3.1413485130	491	389
16	16384	3.1415316184	297	339
16	65536	3.1415773948	299	510
16	262144	3.1415888389	580	547
16	1048576	3.1415916999	1634	1548
16	4194304	3.1415924152	5914	5878
16	16777216	3.1415925940	20390	22756
16	67108864	3.1415926387	76306	70947
16	268435456	3.1415926499	331277	286420
16	1073741824	3.1415926527	1153217	1151170

As you can see from the results there is no noticeable difference between using a `mutex` and using a `semaphore`. The lack of difference maybe because every time the program runs there are only 16 contentions for the lock. With this in mind I devised a new test. This time instead of each thread having their own local `sum`, they will add directly to the global sum. This way there will be n contentions for the lock every time the program runs. Below are the results from that experiment:

Thread	n-size	estimate	Time (μseconds)	
			Mutex	Semaphore
16	1024	3.1406160913	395	447
16	4096	3.1413485130	627	1457
16	16384	3.1415316184	1810	5732
16	65536	3.1415773948	6450	21676
16	262144	3.1415888389	24927	78439
16	1048576	3.1415916999	94784	344271
16	4194304	3.1415924152	389920	1631059
16	16777216	3.1415925940	1542686	5746180
16	67108864	3.1415926387	5908414	22728867
16	268435456	3.1415926499	24040131	89201647
16	1073741824	3.1415926527	97204717	324027725

This time around, there is a clear distinction between using a `mutex` and using a `semaphore`. The `semaphore` is clearly slower than the `mutex`

Exercise 4.8 (a) If a program uses more than one mutex, and the mutexes can be acquired in different orders, the program can **deadlock**. That is, threads may block forever waiting to acquire one of the mutexes. As an example, suppose that a program has two shared data structures–for example, two arrays or two linked lists–each of which has an associated mutex. Further suppose that each data structure can be accessed (read or modified) after acquiring the data structure's associated mutex.

a. Suppose the program is run with two threads. Further suppose that the following sequence of events occurs:

Time	**Thread 0**	**Thread 1**
0	`pthread_mutex_lock(&mut0)`	`pthread_mutex_lock(&mut1)`
1	`pthread_mutex_lock(&mut1)`	`pthread_mutex_lock(&mut0)`

What happens?

Answer: In the above scenario thread 0 gets a lock on `mutex` 0. Then thread 1 gets a lock on `mutex` 1. Afterwards thread 0 attempts to get a lock on `mutex` 1. Because thread 1 already has a lock on `mutex` 1 thread 0 has to wait. Then thread 1 attempts to get a lock on `mutex` 0. Thread 0, who is waiting on a lock, also owns the lock on `mutex` 0. This means that both thread 0 and 1 are waiting on each other. This leads to a deadlock where either thread is waiting on the other thread and no one can move forward.

Exercise 4.11 (a, c, e) Give an example of a linked list and a sequence of memory accesses to the linked list in which the following pairs of operations can potentially result in problems:

a. Two deletes executed simultaneously

c. A member and a delete executed simultaneously

e. An insert and a member executed simultaneously

Answer: Assume the following linked list for all three answers. Where $N1$ is the head node and $N4$ is the tail node.

$$H \rightarrow \boxed{\text{N1}} \rightarrow \boxed{\text{N2}} \rightarrow \boxed{\text{N3}} \rightarrow \boxed{\text{N4}}$$

a. In this scenario threads 1 and 2 are trying to delete node 2 and 3 respectively.

Timer	Thread 1	Thread 2
0	`N1 =` *H*	`N1 =` *H*
1	`N2 = N1.next`	`N2 = N1.next`
2	`N1.next = N2.next`	`N3 = N2.next`
3	`free(N2)`	
4		`N2.next = N3.next`
5		`free(N3)`

On time step 4, thread 2 will try to write to the pointer to `N2`. Since `N2` was freed in time step 3 this will cause an error.

c. In this scenario thread 1 is trying to delete N2 and thread 2 is trying to find N3

Timer	Thread 1	Thread 2
0	`N1 =` *H*	`N1 =` *H*
1	`N2 = N1.next`	`N2 = N1.next`
2	`N1.next = N2.next`	
3	`free(N2)`	
4		`N3 = N2.next`

On time step 4, thread 2 will try to access the pointer to N2. Since in the previous time step N2 was freed, this might cause and error.

e. Consider the two scenarios depicted below. In both scenarios thread 1 is trying to insert node `T` between `N1` and `N2`. While thread 2 is tying to find member `T`.

Timer	Thread 1	Thread 2
0	`N1 =` H	`N1 =` H
1	`N2 = N1.next`	`N2 = N1.next`
2	`T.next = N2`	No match
3	`N1.next = T`	

Timer	Thread 1	Thread 2
0	`N1 =` H	`N1 =` H
1	`N2 = N1.next`	
2	`T.next = N2`	
3	`N1.next = T`	
3		`T = N1.next`
3		Match found

In the first scenario thread 2 is unable to find member `T`. In the second scenario however, thread 2 is able to successfully find member `T`. The two scenarios above have different outcomes depending on the timing of the functions and so have a race condition.

Exercise 4.16 Recall the matrix-vector multiplication example with the 8000x8000 input. Suppose that the program is run with four threads, and thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or eight doubles, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector y? Why? What about if thread 0 and thread 3 are assigned to different processors–is it possible for false sharing to occur between them for any part of y?

Answer: No, it is not possible for any false sharing to occur at any part of the vector y. This is because thread 0 is assigned indices 0–1999. Whereas thread 2 is assigned the indices 4000–5999. These two index ranges are far enough from each other that a cache line could not contain indecis in both ranges simultaneously. The same applies for threads 0 and 3.

Exercise 4.17 Recall the matrix-vector multiplication example with an $8\text{x}8,000,000$ matrix. Suppose that doubles use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

a. What is the minimum number of cache lines that are needed to store the vector y?

b. What is the maximum number of cache lines that are needed to store the vector y?

c. If the boundaries of cache lines always coincide with the boundaries of 8-byte doubles, in how many different ways can the components of y be assigned to cache lines?

d. If we only consider which pairs of threads share a processor, in how many different ways can four threads be assigned to the processors in our computer? Here we're assuming that cores on the same processor share a cache.

e. Is there an assignment of components to cache lines and threads to processors that will result in no false sharing in our example? In other words, is it possible that the threads assigned to one processor will have their components of y in one cache line, and the threads assigned to the other processor will have their components in a different cache line?

f. How many assignments of components to cache lines and threads to processors are there?

g. Of these assignments, how many will result in no false sharing?

Answer:

a. You only need one cache line to store y where the number of rows is 8.

b. The maximum number of caches needed to store y is two. This happens when the cache line is not aligned to the start of y but it is stored in a contiguous block of memory. Therefore y would extend into the next cache line.

c. The components of y can be assigned eight different ways. Assuming that y is stored as a contiguous block of memory, it can only be offset by multiples of 8 bytes. This is because the boundaries of the cache line always align with the boundaries of an 8-byte double. Therefore y can be offset by increments of 8-bytes producing eight possible configurations.

d. Four threads can be assigned in three possible ways.

Processor 1	Processor 2
0, 1	2, 3
0, 2	1, 3
0, 3	1, 2

e. Yes. The following configuration does not result in false sharing:

   - Processor 1 is assigned to thread 0 and thread 1
   - Processor 2 is assigned to thread 2 and thread 3
   - The output array y is split across two cache lines. With the first half of y on one cache line and the second half on the next cache line.

f. There are 24 possible assignments of components to cache lines and threads to processors. This is because there are 3 possible assignments for threads to processors, and 8 possible assignments of components to cache lines. Therefore $3 * 8 = 24$.

g. Only *one* of the 24 assignments will result in no false sharing. The only possible arrangement of components to cache lines that will not cause false sharing is for four components on one cache line and four components on the other. Given this, the only possible arrangement of threads to processors that will not have false sharing is for threads 0 and 1; and 2 and 3 to be paired in different processors. Therefore, there is only one possible arrangement that does not result in false sharing.