Assume that len % k == 0

Define struct thread_arg (alias thread_data) with attributes thread_no, vec, total, k and sumSquares and function thread_value which takes a struct thread_data as an input and return the sum of squares of components in the vector.

```
/* You may need to define struct here */

typedef struct thread_arg{
  int thread_no;
  const float* vec;
  size_t total;
  int k;
  double sumSquares;
} thread_data;

double thread_value(thread_data td){
  return td.sumSquares;
}
```

The subroutine function l2_norm takes thread_arg as an input and calculate the sum of squares of components in the vector. Moreover, we partition the vector into k parts with total/k elements. Each part corresponds to one thread.

```
/*!
 * \brief subroutine function
 *
 * \param arg, input arguments pointer
 * \return void*, return pointer
 */
void *l2_norm(void *arg) { /* TODO: Your code here */
  // calculate sum of squares
  thread_data* td = (thread_data *)arg;
  for(int i=td->thread_no; i<td->total; i+=td->k){
    td->sumSquares += (td->vec[i])*(td->vec[i]);
  }
  return NULL;
}
```

Before creating the threads, we set the attributes of td[i]. After we create those k threads, we wait all of the threads to finish execution using join. Finally, we sum all the results from every thread and take the square root which gives the l2 norm of a vector.

```
/*!
 * \brief wrapper function
 *
 * \param vec, input vector array
 * \param len, length of vector
 * \param k, number of threads
 * \return float, l2 norm
 */
float multi_thread_l2_norm(const float *vec, size_t len,
                           int k) { /* TODO: your code here */
  int rc;
  double res = 0.0f;
  pthread_t pthread_arr[k];
  thread_data td[k];
  for(int i=0; i<k; ++i){
    td[i].thread_no = i;
    td[i].vec = vec;
    td[i].total = len;
    td[i].k = k;
    td[i].sumSquares = 0.0f;
    rc = pthread_create(&(pthread_arr[i]), NULL, l2_norm, &td[i]);
    if(rc != 0){
      printf("Error\n");
      return 1;
    }
  }

  for(int i=0; i<k; ++i){
    pthread_join(pthread_arr[i], NULL);
  }

  for(int i=0; i<k; ++i){
    res += thread_value(td[i]);
  }
  res = sqrt(res);
  return res;
}
```

Execution time:

We can see that multithreading brings advantages to the execution time when length of the vector is much larger than the number of threads. However, there are more overheads when we increase the number of threads. Therefore, huge number of threads may not greatly improve the performance or even degrade the performance. The key to improve the performance is to choose a suitable number of threads.

```
cylee@workbench:~/Desktop$ ./hw1 1024000 16
Vector size=1024000     threads num=16.
[Singlethreading]start
[Singlethreading]The elapsed time is 1.21 ms.
[Multithreading]start
[Multithreading]The elapsed time is 0.79 ms.
Accepted!
cylee@workbench:~/Desktop$ ./hw1 1024000 8
Vector size=1024000     threads num=8.
[Singlethreading]start
[Singlethreading]The elapsed time is 1.21 ms.
[Multithreading]start
[Multithreading]The elapsed time is 0.48 ms.
Accepted!
cylee@workbench:~/Desktop$ ./hw1 1024000 4
Vector size=1024000     threads num=4.
[Singlethreading]start
[Singlethreading]The elapsed time is 1.21 ms.
[Multithreading]start
[Multithreading]The elapsed time is 0.43 ms.
Accepted!
```