```
//
// Sorts a list using multiple threads
//

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <limits.h>

#define MAX_THREADS     65536
#define MAX_LIST_SIZE   100000000

#define DEBUG 0

// Thread variables
//
// VS: ... declare thread variables, mutexes, condition varables,
etc.,
// VS: ... as needed for this assignment
//

// Global variables
int num_threads;          // Number of threads to create - user input
int list_size;                    // List size
int *list;                        // List of values
int *work;                        // Work array
int *list_orig;                   // Original list of values, used for
error checking

// Print list - for debugging
void print_list(int *list, int list_size) {
    int i;
    for (i = 0; i < list_size; i++) {
        printf("[%d] \t %16d\n", i, list[i]);
    }

printf("-------------------------------------------------------------
------\n");
}

// Comparison routine for qsort (stdlib.h) which is used to
// a thread's sub-list at the start of the algorithm
int compare_int(const void *a0, const void *b0) {
    int a = *(int *)a0;
    int b = *(int *)b0;
    if (a < b) {
        return -1;
    } else if (a > b) {
```

No change needed

No change needed

```
        return 1;
    } else {
        return 0;
    }
}

// Return index of first element larger than or equal to v in sorted
list
// ... return last if all elements are smaller than v
// ... elements in list[first], list[first+1], ... list[last-1]
//
//   int idx = first; while ((v > list[idx]) && (idx < last)) idx++;
//
int binary_search_lt(int v, int *list, int first, int last) {

    // Linear search code
    // int idx = first; while ((v > list[idx]) && (idx < last)) idx++;
return idx;

    int left = first;
    int right = last-1;

    if (list[left] >= v) return left;
    if (list[right] < v) return right+1;
    int mid = (left+right)/2;
    while (mid > left) {
        if (list[mid] < v) {
            left = mid;
        } else {
            right = mid;
        }
        mid = (left+right)/2;
    }
    return right;
}
// Return index of first element larger than v in sorted list
// ... return last if all elements are smaller than or equal to v
// ... elements in list[first], list[first+1], ... list[last-1]
//
//   int idx = first; while ((v >= list[idx]) && (idx < last)) idx++;
//
int binary_search_le(int v, int *list, int first, int last) {

    // Linear search code
    // int idx = first; while ((v >= list[idx]) && (idx < last)) idx+
+; return idx;

    int left = first;
    int right = last-1;
```

No change needed

No change needed

```
    if (list[left] > v) return left;
    if (list[right] <= v) return right+1;
    int mid = (left+right)/2;
    while (mid > left) {
        if (list[mid] <= v) {
            left = mid;
        } else {
            right = mid;
        }
        mid = (left+right)/2;
    }
    return right;
}

// Sort list via parallel merge sort
//
// VS: ... to be parallelized using threads ...
//
void sort_list(int q) {

    int i, level, my_id;
    int np, my_list_size;
    int ptr[num_threads+1];

    int my_own_blk, my_own_idx;
    int my_blk_size, my_search_blk, my_search_idx, my_search_idx_max;
    int my_write_blk, my_write_idx;
    int my_search_count;
    int idx, i_write;

    np = list_size/num_threads;   // Sub list size

    // Initialize starting position for each sublist
    for (my_id = 0; my_id < num_threads; my_id++) {
        ptr[my_id] = my_id * np;
    }
    ptr[num_threads] = list_size;

    // Sort local lists
    for (my_id = 0; my_id < num_threads; my_id++) {
        my_list_size = ptr[my_id+1]-ptr[my_id];
        qsort(&list[ptr[my_id]], my_list_size, sizeof(int)
compare_int);
    }
if (DEBUG) print_list(list, list_size);

    // Sort list
    for (level = 0; level < q; level++) {

        // Each thread scatters its sub_list into work array
```

Change or replace this routine as needed to implement your multi-threaded parallel merge

sort_list is written in a way that shows how a single processor can “mimic” a multi core by doing the work of each thread by executing a for-loop that goes through each thread’s computation. Variables named “my_ …”

Consider eliminating this loop when creating routine that will be executed by each thread

Barrier may be needed at several places to synchronize threads to ensure values written by a thread before the barrier is available to other threads after the barrier.

```
        for (my_id = 0; my_id < num_threads; my_id++) {

            my_blk_size = np * (1 << level);

            my_own_blk = ((my_id >> level) << level);
            my_own_idx = ptr[my_own_blk];

            my_search_blk = ((my_id >> level) << level) ^ (1 <<
level);
            my_search_idx = ptr[my_search_blk];
            my_search_idx_max = my_search_idx+my_blk_size;

            my_write_blk = ((my_id >> (level+1)) << (level+1));
            my_write_idx = ptr[my_write_blk];

            idx = my_search_idx;

            my_search_count = 0;


            // Binary search for 1st element
            if (my_search_blk > my_own_blk) {
               idx = binary_search_lt(list[ptr[my_id]], list,
my_search_idx, my_search_idx_max);
            } else {
               idx = binary_search_le(list[ptr[my_id]], list,
my_search_idx, my_search_idx_max);
            }
            my_search_count = idx - my_search_idx;
            i_write = my_write_idx + my_search_count + (ptr[my_id]-
my_own_idx);
            work[i_write] = list[ptr[my_id]];

            // Linear search for 2nd element onwards
            for (i = ptr[my_id]+1; i < ptr[my_id+1]; i++) {
                if (my_search_blk > my_own_blk) {
                    while ((list[i] > list[idx]) && (idx <
my_search_idx_max)) {
                        idx++; my_search_count++;
                    }
                } else {
                    while ((list[i] >= list[idx]) && (idx <
my_search_idx_max)) {
                        idx++; my_search_count++;
                    }
                }
                i_write = my_write_idx + my_search_count + (i-
my_own_idx);
                work[i_write] = list[i];
            }
```

Consider eliminating this loop when creating routine that will be executed by each thread

thread writes values from list to work when merging sub lists, then work array is copied back to list for the next iteration of level.

]

```
        }
        // Copy work into list for next itertion
        for (my_id = 0; my_id < num_threads; my_id++) {
            for (i = ptr[my_id]; i < ptr[my_id+1]; i++) {
                list[i] = work[i];
            }
        }
if (DEBUG) print_list(list, list_size);
    }
}

// Main program - set up list of random integers and use threads to
sort the list
//
// Input:
//       k = log_2(list size), therefore list_size = 2^k
//       q = log_2(num_threads), therefore num_threads = 2^q
//
int main(int argc, char *argv[]) {

    struct timespec start, stop, stop_qsort;
    double total_time, time_res, total_time_qsort;
    int k, q, j, error;

    // Read input, validate
    if (argc != 3) {
        printf("Need two integers as input \n");
        printf("Use: <executable_name> <log_2(list_size)>
<log_2(num_threads)>\n");
        exit(0);
    }
    k = atoi(argv[argc-2]);
    if ((list_size = (1 << k)) > MAX_LIST_SIZE) {
        printf("Maximum list size allowed: %d.\n", MAX_LIST_SIZE);
        exit(0);
    };
    q = atoi(argv[argc-1]);
    if ((num_threads = (1 << q)) > MAX_THREADS) {
        printf("Maximum number of threads allowed: %d.\n",
MAX_THREADS);
        exit(0);
    };
    if (num_threads > list_size) {
        printf("Number of threads (%d) < list_size (%d) not allowed.
\n",
           num_threads, list_size);
        exit(0);
    };

    // Allocate list, list_orig, and work
```

Consider eliminating this loop when creating routine that will be executed by each thread

2^k

2^q

```
    list = (int *) malloc(list_size * sizeof(int));
    list_orig = (int *) malloc(list_size * sizeof(int));
    work = (int *) malloc(list_size * sizeof(int));

//
// VS: ... May need to initialize mutexes, condition variables,
// VS: ... and their attributes
//

    // Initialize list of random integers; list will be sorted by
    // multi-threaded parallel merge sort
    // Copy list to list_orig; list_orig will be sorted by qsort and
used
    // to check correctness of multi-threaded parallel merge sort
    srand48(0);  // seed the random number generator
    for (j = 0; j < list_size; j++) {
        list[j] = (int) lrand48();
        list_orig[j] = list[j];
    }
    // duplicate first value at last location to test for repeated
values
    list[list_size-1] = list[0]; list_orig[list_size-1] =
list_orig[0];

    // Create threads; each thread executes find_minimum
    clock_gettime(CLOCK_REALTIME, &start);

//
// VS: ... may need to initialize mutexes, condition variables, and
their attributes
//

// Serial merge sort
// VS: ... replace this call with multi-threaded parallel routine for
merge sort
// VS: ... need to create threads and execute thread routine that
implements
// VS: ... parallel merge sort

    sort_list(q);
```

Replace with multithreaded routine

```
    // Compute time taken
    clock_gettime(CLOCK_REALTIME, &stop);
    total_time = (stop.tv_sec-start.tv_sec)
        +0.000000001*(stop.tv_nsec-start.tv_nsec);

    // Check answer
    qsort(list_orig, list_size, sizeof(int), compare_int);
    clock_gettime(CLOCK_REALTIME, &stop_qsort);
```

Sort a copy of original list via quicksort to compare your sorted list

```
    total_time_qsort = (stop_qsort.tv_sec-stop.tv_sec)
        +0.000000001*(stop_qsort.tv_nsec-stop.tv_nsec);

    error = 0;
    for (j = 1; j < list_size; j++) {
        if (list[j] != list_orig[j]) error = 1;
    }

    if (error != 0) {
        printf("Houston, we have a problem!\n");
    }

    // Print time taken
    printf("List Size = %d, Threads = %d, error = %d, time (sec) =
%8.4f, qsort_time = %8.4f\n",
            list_size, num_threads, error, total_time,
total_time_qsort);

// VS: ... destroy mutex, condition variables, etc.

    free(list); free(work); free(list_orig);

}
```