Final Project Design Note

Goal:

Exploit multiple processors to implement Merge-Sort and study the performance.

1. Assumptions

- The implementation uses the divide and conquer strategy to divide the problem into multiple tasks, give each task to a separate thread, and then combine the results to get the complete solution.

- The system used to implement the project has multiple cores (logical processors). The number of cores (threads) used in the implementation is a power of two.

- Ideally, we would like as many threads as there are cores, so the threads will execute in parallel, thus reducing elapsed time. If the system has just two cores, the number of threads to use are limited to 4 or 8.

- The number of records in the sort file is divisible by the number of threads (power of 2)

- Each data record is made up of an 8-byte key followed by 56 bytes of data. Total length of the record is 64 bytes.

2. Multithreading models

- Boss/Worker model
  The Boss thread (main thread) assigns tasks to the worker threads to perform the task. The worker threads pass their results to the main thread.
- Work Crew model
  The worker threads cooperate on the task, each performing a small piece. The workers might even divide up the work themselves without direction from the boss.

- Server/Client model
- Pipeline model

3. Strategy

a. The Divide and Conquer or Worker crew model

Use this model for the project in which a single task, sorting a file, is divided into subtasks and delegated to separate threads.

b. Algorithm

The basic idea is to divide the problem into component tasks, give each task to a separate thread, and then combine the results to get the complete solution.

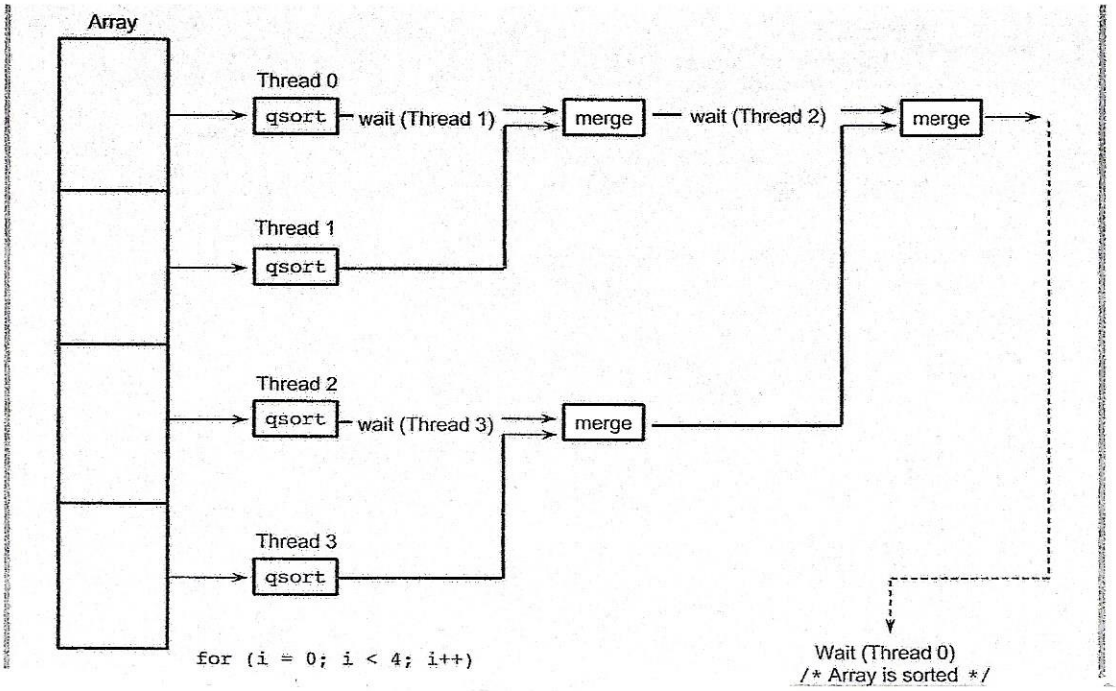
Merge-sort, in which the array to be sorted is divided into smaller arrays, is a divide and conquer algorithm. Each small array is sorted individually using the C library `qsort()` function, so there is no need to develop an efficient `sort` function. The individual sorted arrays are merged in pairs to yield larger arrays. The pairwise merging continues until completion. This is illustrated in the diagram below.

The implementation has three (3) phases:

- Memory mapping phase
- Sort phase, and
- Merge phase

The program starts with relatively large arrays so there is one array for each processor. The program displays the number of cores available on the system so the user can specify the number of threads on the command line.

Pthreads start executing as soon as they are created. This could be a problem here due to the way the threads are assigned to arrays in our design to facilitate pairwise merging of the arrays. Threads are created in a loop. Adjacent arrays are assigned to a pair of threads, the first thread is assigned to the first array and the second thread is assigned to the second (adjacent) array. The first thread must wait for the second


Array
Thread 0
qsort
wait (Thread 1)
merge
wait (Thread 2)
merge
Thread 1
qsort
Thread 2
qsort
wait (Thread 3)
merge
Thread 3
qsort
for (i = 0; i < 4; i++)
Wait (Thread 0)
/* Array is sorted */

thread (using pthread_join) before starting the merge phase. What happens if the first thread is created and waits for the second thread before the second thread is created? In this case the thread ID to wait for does not exist. This is a race condition where two or more threads make unsafe assumptions about the progress of the other threads. What is needed is some capability to start the threads in a suspended state and resume them after all the threads are created. Unfortunately, there is no facility in pthreads to do that. However, pthreads has a facility to *signal changes of state* using *condition variables*.

A condition variable allows one thread to inform other threads about changes in the state of a shared variable and allows the other threads to wait (block) for such notification.

We will use condition variables to suspend each pthread as soon as it is created and resume all the threads after all of them are created.

c. Data Structures used

```
#define KEYSIZE   8
#define DATASIZE 56

struct Record {
      char key[KEYSIZE];
      char data[DATASIZE];
};


#define RECSIZE sizeof(Record)
```

Computing Number of Records

```
// Number of records
int nRecs = FileSize / RECSIZE;

// Number of Records per thread
int nRecsPerThd = nRecs / nThreads;
```

Thread Argument structure passed to the thread function

```
static struct ThdArg {
     int thdNum; // Thread number 0,1,2,3…
     Record * lowRec; // First record of group
     Record * hiRec;   // Last record of group
};
```

Pointer to Array of Thread IDs

```
static pthread_t * p_tids;
```