Implementation of the mergesort using thread pool

1. Basic Idea:

Merge sort is a kind of sorting algorithm. By dividing the unsorted numbers into smaller parts, sorting the small subsets, and then merge the sorted subsets into one big array. To make the algorithm works more quickly, parallelism is needed. Multi-thread is an approach.

In this experiment, I implemented a merge sort algorithm using multi-thread. To implement the multi-thread mechanism, I used the so called thread pool, based on Ponix.

2. Steps:

1) Read the unsorted numbers from a file. Add sentinels if the size is not power of 2. Assume the size of data is S now.
2) Initialize N threads, and make them waiting. (Thread pool) N is calculated based on the size of the unsorted numbers.
3) Initialize a Task queue. The sorting mission would be the tasks. Every time there's a new task arrives, it will be added to the task queue and wait for free thread to process the sorting mission.
4) Those threads under waiting statues would be woke up and fetch a task from the task queue, and then execute the sorting process.
5) There will be $\log_2 S$ times loops. For the first loop, the number of active thread will be S/2, and each thread is dealing with only two numbers. Sort and merge the two numbers. After they completed with sorting, all threads will be switch to *pthread_cond_wait()* statue.
6) The main thread will be waiting, until all the worker thread is done and switched to waiting statue (To do this, there is a counter to record the number of active threads. Once the counter is zeroed, a signal will be sent to main thread). And then the main thread is waking up (by cond_signal as well), and put new sorting tasks into the task queue.
7) Iterate the 4-6 steps, and after all array is returned, it is a sorted array.

3. Specific problems:

a) How to implement the thread pool:

Create several threads, and let them wait. These newly initialized threads will check the task queue first. If there are tasks in the queue, process them. Otherwise, let them wait (*pthread_cond_wait*), until new task arrives, and threads will be woke up.

b) How to implement the synchronization:

Many situation requires synchronization, for example, the modification of activeThreadCounter, modification of data array, modification of task queue etc. And I used mutex(*pthread_mutex_lock* and *pthread_mutex_unlock*) and conditional wait(*pthread_cond_wait* and *pthread_cond_signal*) to implement the synchronization. When come up with the modification of public resources, threads will lock related mutex first, modify the data and then unlock it so that other thread can lock and read it.

c) Whether or not the main thread participates in sorting:

No. The main thread in my program is actually a manager, which arranges sorting tasks. It would create enough threads at the beginning, and issue sorting tasks to the task queue. Once new task arrives, the queue will send a signal to those waiting threads, and then a thread will be woke up and continue to sort the data. After all tasks are processed, the last thread will send a signal back to main thread, and then the main thread will wake up and arrange new tasks to the queue.

d) How you organize the intermediate results:

By transferring the address of original data array, the sorting threads will modify the origin array directly. So there's no need to record the intermediate result. It saves much space by doing so.

4. Current problem:

The program is not very stable at this point, and I think the problem is something regard with synchronization. With the increasing of amount of the test number, the program may be stuck and crashed. I've tested the program and it would run well if the size of number is below 128, and it may sometime get stuck if the size grows up to 256. If the program doesn't work, try a little bit more times please... Thank you!