Project 6 Report

110511010 楊育陞

Part I

Q: Explain how you implement your code clearly with screenshots of your functions.

Step 1. After reading the list of data, I split the list into two halves and create two threads running function 'multi_thread_morge_sort' to sort them respectively.

```
// Split the list into two halves and sort them in parallel
pthread_t threads[2];
ThreadData thread_data[2];

thread_data[0].thread_id = 0;
thread_data[0].l = 0;
thread_data[0].r = list_size / 2 - 1;
pthread_create(&threads[0], NULL, multi_thread_merge_sort, &thread_data[0]);

thread_data[1].thread_id = 1;
thread_data[1].l = list_size / 2;
thread_data[1].r = list_size - 1;
pthread_create(&threads[1], NULL, multi_thread_merge_sort, &thread_data[1]);
```

Step 2. When running 'multi_thread_morge_sort', I use merge sort algorithm to sort the list.

```
void *multi_thread_merge_sort(void *arg)
{
    ThreadData *data = (ThreadData *)arg;
    merge_sort(data->l, data->r);
    pthread_exit(NULL);
}
```

Step 3. Use 'pthread_join' to wait for the two threads to finish sorting.

```
// Wait for the two halves to be sorted
pthread_join(threads[0], NULL);
pthread_join(threads[1], NULL);
```

Step 4. Merge the two sorted lists into one sorted list.

```
// Merge the two sorted halves
merge(0, list_size / 2 - 1, list_size - 1);
```

Step 5. Write the sorted list into the output file.

Following is the screenshot of merge & merge_sort function: merge:

```
void merge(int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1; // size of left subarray
    int n2 = r - m;     // size of right subarray

    // Create temporary arrays
    int L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = list[l + i];
    for (j = 0; j < n2; j++)
        R[j] = list[m + 1 + j];

    // Merge the temporary arrays back into list[l..r]
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j]) // Change this to L[i] > R[j] for descending order
        {
            list[k] = L[i];
            i++;
        }
        else
        {
            list[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if there are any
    while (i < n1)
    {
        list[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2)
    {
        list[k] = R[j];
        j++;
        k++;
    }
}
```

merge_sort:

```
void merge_sort(int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2; // Same as (l+r)/2, but avoids overflow for large l and r
        merge_sort(l, m);
        merge_sort(m + 1, r);
        merge(l, m, r);
    }
}
```

Part II

Q1: Describe which lines of code you modify with detailed explanations.

A1: I mainly modify the function "calculate_innner_product" as following:

```
// Function to calculate the inner product for a portion of the vectors
void* calculate_inner_product(void* arg) {
    ThreadData* data = (ThreadData*)arg;

    int temp_inner_product = 0;
    for (int i = data->start; i < data->end; i++) {
        temp_inner_product += vector1[i] * vector2[i];
    }

    // Lock the mutex before updating the global inner product
    pthread_mutex_lock(&mutex);
    inner_product += temp_inner_product;
    pthread_mutex_unlock(&mutex);


    pthread_exit(NULL);
}
```

The original code add the result of multiplication of each element directly to global variable "inner_product". I modify the code to add the result of multiplication of each element to local variable "temp_inner_product". After finishing the loop, I lock the mutex and add "temp_inner_product" to "inner_product" then unlock the mutex. I also add some code to initialize and destroy the mutex as following: In global domain:

```
pthread_mutex_t mutex;

```

In main function:

```
    // Initialize the mutex
    pthread_mutex_init(&mutex, NULL);
```

```

    pthread_mutex_destroy(&mutex);
```

Q2: After implementing modifications to the sample code in Part II, consider whether the multi-threaded version will outperform the serial version provided below. What factors contribute to the difference in performance?

A2: The multi-threaded version will outperform the serial version. In this case, NUM_THREADS = 8, so the multi-threaded version will split the two vectors into 8 parts and calculate the inner product of each part respectively. The multi-threaded version will add the results of each part to final result only after finishing the loop and get the mutex lock. So the multi-threaded version will be 8 times faster than the serial version ideally (ignore overhead).