Parallel Programming Observation

24/4/21

2015103569

1. Write programs involving both OpenMP and MPI

```
#include<stdio.h>
#include<mpi.h>
#include<omp.h>
int main(int argc, char *argv[]){
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int iam=0, np=1;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    if(rank==0){
        omp_set_num_threads(1);
    }
    if(rank==1){
        omp_set_num_threads(3);
    }

    #pragma omp parallel default(shared) private(iam,np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
    }
    MPI_Finalize();
    return 0;
}
```

2. Hybrid programs

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>
#include <omp.h>
float *create_rand_nums (int num_elements) {
    float *rand_nums = (float *)malloc (sizeof(float) *
                                        num_elements);
    assert(rand_nums != NULL);
    int i;
    #pragma omp parallel for num_threads (5)
    for(i=0; i<num_elements; i++){
        rand_nums[i] = (rand() / (float) RAND_MAX);
    }
    return rand_nums;
}
float compute_avg(float *array, int num_elements){
    float sum = 0.f;
    int i;
    #pragma omp parallel for reduction(+:sum)
     num_threads (5)
     for(i=0; i<num_elements; i++){
         sum += array[i];
     }
     return sum/num_elements;
}
int main(int argc, char *argv[]) {
    if (argc != 2){
        fprintf (stderr, "Err");
        exit (1)
    }
```

```
int num_elements_per_proc = atoi(argv[1]);
srand(time(NULL));
clock_t begin, end;
double time_spent;
MPI_Init(NULL, NULL);
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
begin = clock();
float *rand_nums = NULL;
if (world_rank == 0) {
    rand_nums = create_rand_nums(num_elements_per_proc * world_size);
}
float *sub_rand_nums = (float *)
        malloc(sizeof(float) * num_elements_per_proc);
assert(sub_rand_nums != NULL);

MPI_Scatter(rand_nums, num_elements_per_proc,
    MPI_FLOAT, sub_rand_nums, num_elements_per_proc,
    MPI_FLOAT, 0, MPI_COMM_WORLD);
float sum = compute_avg(sub_rand_nums, num_elements_per_proc);

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
}
```

Merge sort

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<mpi.h>
#include<omp.h>
void merge(int *, int *, int, int, int);
void mergeSort(int *, int *, int, int);
int main(int argc, char *argv[]) {

  int thread_num = atoi(argv[1]);
  int n = atoi(argv[2]);
  int *original_array = malloc(n * sizeof(int));
  srand(time(NULL));
  double exec_time;
  struct timespec begin, end;
  int world_rank;
  int world_size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank)
  MPI_Comm_size(MPI_COMM_WORLD, &world_size)
  MPI_Scatter(original_array, size, MPI_INT, sub_array,
      size, MPI_INT, 0, MPI_COMM_WORLD)

#pragma omp parallel num_threads(thread_Num)
{
    #pragma omp single
    mergeSort(sub_array, tmp_array, 0, (size-1));
}

MPI_Barrier(MPI_COMM_WORLD)
MPI_Finalize();
}
```

```
void merge(int *a, int *b, int l, int m, int r)
{
    int h, i, j, k;
    h = l;
    i = l;
    j = m + 1;

    while((h <= m) && (j <= r))
    {
        if(a[h] <= a[j])
        {
            b[i] = a[h];
            h++;
        }
        else
        {
            b[i] = a[j];
            j++;
        }
        i++;
    }
    if(m < h)
    {
        for(k = j; k <= r; k++)
        {
            b[i] = a[k];
            i++;
        }
    }
    else
    {
        for(k = h; k <= m; k++)
        {
            b[i] = a[k];
            i++;
        }
    }
}
void mergesort(int *a, int *b, int l, int r)
{
    int m;
    if(l < r)
    {
        m = (l + r)/2;
```

```
#pragma omp task shared(a,b)
mergeSort(a,b,l,m);

#pragma omp task shared(a,b)
mergeSort(a,b,m+1,r);

#pragma omp task wait
merge(a,b,l,m,r);
}
}
```