OPERATING SYSTEMS
CS342

Project 2

Ramazan Mert Cinar – Yaman Yagiz Tasbag
21601985 - 21601639
Section 2 – Section 2

Part C)

1)

There are 10 indentical input files in the experiment. As we can see from the table below, batch size does not affect the running times significantly. It may increase the running time if the batch size is too small or too big because there would be many context switches if it is too small and there would be a lot of waiting if it is too big. There is no meaningful job done when context switching so, it may affect the time elapsed in a bad way.

batch size	Time Elapsed
3	31 ms
4	36 ms
5	37 ms
6	32 ms
7	33 ms

Table 1

Appendices

1)

```c
#include  <stdio.h>
#include  <sys/types.h>
#include  <stdlib.h>
#include  <assert.h>
#include  <string.h>
#include  <pthread.h>

int minvalue;
int maxvalue;
int bincount;
int N;
int batch;
char **files;
char *outfile;
double w;
pthread_mutex_t mutex;
int finished_threads = 0;
pthread_cond_t busy;
int working = 0;

struct Node
{
   double value;
   struct Node* next;
};


struct Node* head = NULL;
struct Node* tail = NULL;


void set_working()
{
   pthread_mutex_lock(&mutex);
   if (working == 0)
   {
      working++;
      pthread_mutex_unlock(&mutex);
   }
   else
   {
      working++;
      pthread_cond_wait(&busy, &mutex);
   }
```

```
}

void set_finished_working()
{
   pthread_mutex_lock(&mutex);
   working--;
   pthread_cond_signal(&busy);
   pthread_mutex_unlock(&mutex);
}


void push(double value)
{
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->value = value;
   node->next = NULL;
   if(!tail)
   {
      head = tail = node;
      return;
   }
   tail->next = node;
   tail = node;
}

double pop()
{
   assert(head);
   double value = head->value;
   struct Node* temp = head;
   head = head->next;
   free(temp);
   return value;
}

int is_finished()
{
   pthread_mutex_lock(&mutex);
   int t = finished_threads == N;
   pthread_mutex_unlock(&mutex);
   return t;
}

int set_finished()
{
   pthread_mutex_lock(&mutex);
   finished_threads++;
   pthread_mutex_unlock(&mutex);

}

void consume()
{
   int *histogram = (int*) malloc(bincount * sizeof(int));
   memset(histogram, 0, bincount * sizeof(int));
   int i;

   while(!is_finished())
   {
      pthread_mutex_lock(&mutex);
      int enough = 1;
      struct Node* curr = head;
      for(i = 1; i < batch; i++)
      {
         if(curr == NULL)
         {
            enough = 0;
            break;
         }
         curr =  curr->next;
      }

      if(enough)
      {
         double value = pop();
         assert(value <= maxvalue && value >= minvalue);
         int current_bin = (int)((value - minvalue) / w);
         current_bin = (current_bin >= bincount) ? bincount - 1 : current_bin;
         histogram[current_bin]++;
      }
      pthread_mutex_unlock(&mutex);
   }

   pthread_mutex_lock(&mutex);
   struct Node* curr = head;
   while(curr)
   {
         double value = pop();
         assert(value <= maxvalue && value >= minvalue);
         int current_bin = (int)((value - minvalue) / w);
         current_bin = (current_bin >= bincount) ? bincount - 1 : current_bin;
         histogram[current_bin]++;
         curr = curr->next;
   }
   pthread_mutex_unlock(&mutex);

   FILE *out = fopen(outfile, "w");

   for(i = 0; i < bincount; i++)
      fprintf(out, "%d: %d\n", i + 1, histogram[i]);

   free(histogram);
}




void produce(void * ptr)
{
   int i = *((int*)ptr);
   FILE *f = fopen(files[i], "r");
   double value;
   double *b = (double *)malloc(sizeof(double) * batch);
   memset(b, 0, sizeof(double) * batch);
   int bc;
   int finished = 0;
   bc = 0;
   while(fscanf(f, "%lf", &value) != EOF)
   {
      assert(value <= maxvalue && value >= minvalue);
      int current_bin = (int)((value - minvalue) / w);
      current_bin = (current_bin >= bincount) ? bincount - 1 : current_bin;
      b[bc++] = value;

      if(bc == batch)
      {
         set_working();
         int j;
         for (j = 0; j < bc; j++)
         {
            push(b[j]);
         }
         set_finished_working();
         bc = 0;
      }

   }
   set_working();
   int j;
   for (j = 0; j < bc; j++)
   {
      push(b[j]);
   }
   set_finished_working();
   bc = 0;
   set_finished();


}

int main(int argc, char **argv)
{
   if (argc < 8)
   {
      printf("Usage: phistogram minvalue maxvalue bincount N file1 … fileN outfile batch\n");
      return -1;
   }
   minvalue = atoi(argv[1]);
   maxvalue = atoi(argv[2]);
   bincount = atoi(argv[3]);
   N = atoi(argv[4]);

   if(argc != 5 + N + 2)
   {
      printf("Error: Input files count does not match N \n");
      return -1;
   }
   files = &argv[5];
   outfile = argv[5 + N];
   batch = atoi(argv[5 + N + 1]);

   w = 1.0 * (maxvalue - minvalue) / bincount;
   int i;


   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&busy, NULL);
   pthread_t *threads =  (pthread_t *) malloc(N * sizeof(pthread_t));
   int **is = malloc(N * sizeof(int*));

   for(i = 0; i < N; i++)
   {
      is[i] = malloc(sizeof(int));
      *is[i] = i;
   }

   for(i = 0; i < N; i++)
```

```
      pthread_create( &threads[i], NULL, produce, (void*) is[i]);


   consume();
   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&busy);

   for(i = 0; i < N; i++)
   {
      free(is[i]);
   }
   free(threads);
   free(is);
   printf("%p\n", head);
   printf("%d\n", finished_threads);
   return 0;


}
```

2)

```
          #include  <stdio.h>
#include  <sys/types.h>
#include  <stdlib.h>
#include  <assert.h>
#include  <string.h>
#include  <pthread.h>

int minvalue;
int maxvalue;
int bincount;
int N;
int batch;
char **files;
char *outfile;
double w;
pthread_mutex_t mutex;
int finished_threads = 0;
pthread_cond_t busy;
int working = 0;

struct Node
{
   double value;
   struct Node* next;
};
```

```
struct Node* head = NULL;
struct Node* tail = NULL;


void set_working()
{
   pthread_mutex_lock(&mutex);
   if (working == 0)
   {
      working++;
      pthread_mutex_unlock(&mutex);
   }
   else
   {
      working++;
      pthread_cond_wait(&busy, &mutex);
   }

}

void set_finished_working()
{
   pthread_mutex_lock(&mutex);
   working--;
   pthread_cond_signal(&busy);
   pthread_mutex_unlock(&mutex);
}


void push(double value)
{
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->value = value;
   node->next = NULL;
   if(!tail)
   {
      head = tail = node;
      return;
   }
   tail->next = node;
   tail = node;
}

double pop()
{
   assert(head);
   double value = head->value;
   struct Node* temp = head;
   head = head->next;
   free(temp);
   return value;
}

int is_finished()
{
   pthread_mutex_lock(&mutex);
   int t = finished_threads == N;
   pthread_mutex_unlock(&mutex);
   return t;
}

int set_finished()
{
   pthread_mutex_lock(&mutex);
   finished_threads++;
   pthread_mutex_unlock(&mutex);

}

void consume()
{
   int *histogram = (int*) malloc(bincount * sizeof(int));
   memset(histogram, 0, bincount * sizeof(int));
   int i;

   while(!is_finished())
   {
      pthread_mutex_lock(&mutex);
      int enough = 1;
      struct Node* curr = head;
      for(i = 1; i < batch; i++)
      {
         if(curr == NULL)
         {
            enough = 0;
            break;
         }
         curr =  curr->next;
      }

      if(enough)
      {
         double value = pop();
         assert(value <= maxvalue && value >= minvalue);
         int current_bin = (int)((value - minvalue) / w);
         current_bin = (current_bin >= bincount) ? bincount - 1 : current_bin;
         histogram[current_bin]++;
      }
      pthread_mutex_unlock(&mutex);
   }

   pthread_mutex_lock(&mutex);
   struct Node* curr = head;
   while(curr)
   {
         double value = pop();
         assert(value <= maxvalue && value >= minvalue);
         int current_bin = (int)((value - minvalue) / w);
         current_bin = (current_bin >= bincount) ? bincount - 1 : current_bin;
         histogram[current_bin]++;
         curr = curr->next;
   }
   pthread_mutex_unlock(&mutex);

   FILE *out = fopen(outfile, "w");

   for(i = 0; i < bincount; i++)
      fprintf(out, "%d: %d\n", i + 1, histogram[i]);

   free(histogram);
}




void produce(void * ptr)
{
   int i = *((int*)ptr);
   FILE *f = fopen(files[i], "r");
   double value;
   double *b = (double *)malloc(sizeof(double) * batch);
   memset(b, 0, sizeof(double) * batch);
   int bc;
   int finished = 0;
   bc = 0;
   while(fscanf(f, "%lf", &value) != EOF)
   {
      assert(value <= maxvalue && value >= minvalue);
      int current_bin = (int)((value - minvalue) / w);
```

```
        current_bin = (current_bin >= bincount) ? bincount - 1 : current_bin;
        b[bc++] = value;

        if(bc == batch)
        {
            set_working();
            int j;
            for (j = 0; j < bc; j++)
            {
                push(b[j]);
            }
            set_finished_working();
            bc = 0;
        }

    }
    set_working();
    int j;
    for (j = 0; j < bc; j++)
    {
        push(b[j]);
    }
    set_finished_working();
    bc = 0;
    set_finished();


}

int main(int argc, char **argv)
{
    if (argc < 8)
    {
        printf("Usage: phistogram minvalue maxvalue bincount N file1 … fileN outfile batch\n");
        return -1;
    }
    minvalue = atoi(argv[1]);
    maxvalue = atoi(argv[2]);
    bincount = atoi(argv[3]);
    N = atoi(argv[4]);

    if(argc != 5 + N + 2)
    {
        printf("Error: Input files count does not match N \n");
        return -1;
    }
    files = &argv[5];
    outfile = argv[5 + N];
    batch = atoi(argv[5 + N + 1]);

    w = 1.0 * (maxvalue - minvalue) / bincount;
    int i;


    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&busy, NULL);
    pthread_t *threads =  (pthread_t *) malloc(N * sizeof(pthread_t));
    int **is = malloc(N * sizeof(int*));

    for(i = 0; i < N; i++)
    {
        is[i] = malloc(sizeof(int));
        *is[i] = i;
    }

    for(i = 0; i < N; i++)
        pthread_create( &threads[i], NULL, produce, (void*) is[i]);


    consume();
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&busy);

    for(i = 0; i < N; i++)
    {
        free(is[i]);
    }
    free(threads);
    free(is);
    printf("%p\n", head);
    printf("%d\n", finished_threads);
    return 0;


}
```