Module:	Introduction to Parallel Programming Techniques
Module ID:	EE4107
Student Name:	Umid Narziev
Student ID:	200234832
Assignment Number:	5
Academic Year:	2020 - 2021

Contents

System Specification

Below, shown the system which was used to run and get the result of the simulation:

CPU:	Intel i5-7200U
Architecture:	Kaby Lake
Segment:	Mobile Processors
The number of cores:	2
Number of threads	4
Clock Frequency	2.50-3.10GHz (Turbo Boost)
Cache levels:	3
Cache level 1 size:	128KBytes
Cache level 2 size:	512Kbytes
Cache level 3 size:	3MBytes
RAM	12 GB
SSD:	250 GB
Operating System:	Ubuntu 20.04.2 LTS
Compiler:	Gcc and its libraries
IDE:	Clion (2020.03)

TASK 1

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include "pthread.h"
#include <semaphore.h>
#include "timer.h"

/*
 * compiling -> gcc -g -Wall -o 5.1 5.1.c -pthread
 * No argument is required
 */
long long int a, b, n;
int thread_count;
double global_result_busy_waiting = 0;
double global_result_mutex = 0;
double global_result_sempahore = 0;
int flag;
pthread_mutex_t mut;
sem_t semaphores;

void *trap_busy_waiting(void *rank);

void *trap_mutex(void *rank);

void *trap_semaphore(void *rank);

int main(int argc, char *argv[]) {

   double start, end;
   pthread_t *thread_handles;
   for (int thread_number = 1; thread_number < 128; thread_number <<= 1) {
      thread_count = thread_number;
      a = 0;
      b = 10000;
      n = 1000000000;
      thread_handles = malloc(thread_count * sizeof(pthread_t));
      pthread_mutex_init(&mut, NULL);
      sem_init(&semaphores, 1, 1);
      global_result_busy_waiting = 0;
      global_result_mutex = 0;
      global_result_sempahore = 0;
      //-----------------------------------------Busy-Waiting-----------------------------------------
      GET_TIME(start);
      for (long thread = 0; thread < thread_count; thread++) {
         pthread_create(&thread_handles[thread], NULL, trap_busy_waiting, (void *) thread);
      }

      for (long thread = 0; thread < thread_count; thread++) {
         pthread_join(thread_handles[thread], NULL);
      }
      GET_TIME(end);
      printf("The elapsed time -> %e, estimated value is %lf from busy waiting function, thread_count ->
%d\n", end - start,
            global_result_busy_waiting, thread_count);

      //-----------------------------------------Mutex-----------------------------------------
      GET_TIME(start);
      for (long thread = 0; thread < thread_count; thread++) {
         pthread_create(&thread_handles[thread], NULL, trap_mutex, (void *) thread);
      }
```

```
    for (long thread = 0; thread < thread_count; thread++) {
      pthread_join(thread_handles[thread], NULL);
    }
    GET_TIME(end);
    printf("The elapsed time -> %e, estimated value is %lf from mutex function, thread_count -> %d\n", end -
start,
           global_result_mutex, thread_count);
    //---------------------------------Semaphore-----------------------------------------------------
    GET_TIME(start);
    for (long thread = 0; thread < thread_count; thread++) {
      pthread_create(&thread_handles[thread], NULL, trap_semaphore, (void *) thread);
    }

    for (long thread = 0; thread < thread_count; thread++) {
      pthread_join(thread_handles[thread], NULL);
    }
    GET_TIME(end);
    printf("The elapsed time -> %e, estimated value is %lf from semaphore function, thread_count -> %d\n",
end - start,
           global_result_sempahore, thread_count);
    printf("\n");
  }
  free(thread_handles);
  pthread_mutex_destroy(&mut);
  sem_destroy(&semaphores);

  return 0;
}


void *trap_busy_waiting(void *rank) {
  double my_rank = (long) rank;

  double h = (double) (b - a) / n;
  double local_n = (double) n / thread_count;
  double local_a = a + my_rank * local_n * h;
  double local_b = local_a + local_n * h;

  double estimated = local_a * local_a + local_b * local_b;
  double x;
  for (int i = 1; i < local_n - 1; ++i) {
    x = local_a + i * h;
    estimated += (x * x);
  }
  estimated = estimated * h;

  while (flag != my_rank);
  global_result_busy_waiting += estimated;
  flag = (flag + 1) % thread_count;


  return NULL;
}


void *trap_mutex(void *rank) {
  double my_rank = (long) rank;

  double h = (double) (b - a) / n;
  double local_n = (double) n / thread_count;
  double local_a = a + my_rank * local_n * h;
```

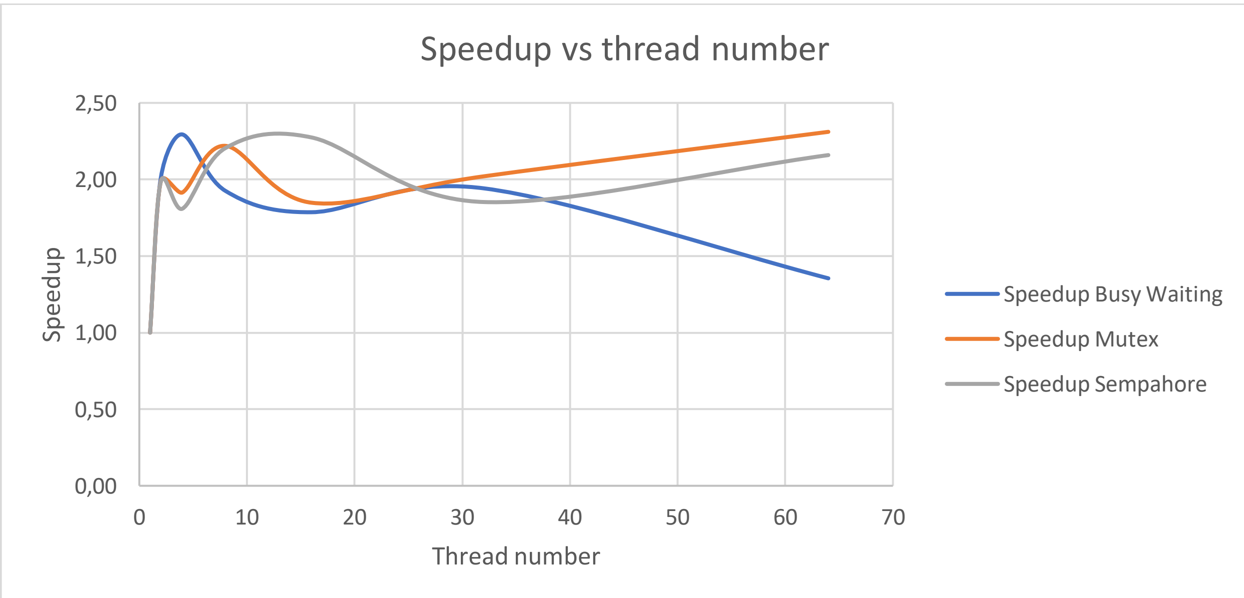
```
    double local_b = local_a + local_n * h;

    double estimated = local_a * local_a + local_b * local_b;
    double x;
    for (int i = 1; i < local_n - 1; ++i) {
        x = local_a + i * h;
        estimated += (x * x);
    }
    estimated = estimated * h;

    pthread_mutex_lock(&mut);
    global_result_mutex += estimated;
    pthread_mutex_unlock(&mut);

    return NULL;
}


void *trap_semaphore(void *rank) {
    double my_rank = (long) rank;

    double h = (double) (b - a) / n;
    double local_n = (double) n / thread_count;
    double local_a = a + my_rank * local_n * h;
    double local_b = local_a + local_n * h;

    double estimated = local_a * local_a + local_b * local_b;
    double x;
    for (int i = 1; i < local_n - 1; ++i) {
        x = local_a + i * h;
        estimated += (x * x);
    }
    estimated = estimated * h;

    sem_post(&semaphores);
    global_result_sempahore += estimated;
    sem_wait(&semaphores);

    return NULL;
}
```

Result

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 5/5.1$ ./5.1
The elapsed time -> 2.963744e+00, estimated value is 333333332833.367920 from busy waiting function, thread_count -> 1
The elapsed time -> 2.929868e+00, estimated value is 333333332833.367920 from mutex function, thread_count -> 1
The elapsed time -> 2.932599e+00, estimated value is 333333332833.367920 from semaphore function, thread_count -> 1

The elapsed time -> 1.472920e+00, estimated value is 333333332833.375854 from busy waiting function, thread_count -> 2
The elapsed time -> 1.472441e+00, estimated value is 333333332833.375854 from mutex function, thread_count -> 2
The elapsed time -> 1.476471e+00, estimated value is 333333332833.375854 from semaphore function, thread_count -> 2

The elapsed time -> 1.291968e+00, estimated value is 333333332833.401001 from busy waiting function, thread_count -> 4
The elapsed time -> 1.529313e+00, estimated value is 333333332833.401001 from mutex function, thread_count -> 4
The elapsed time -> 1.620882e+00, estimated value is 333333332833.401001 from semaphore function, thread_count -> 4

The elapsed time -> 1.539121e+00, estimated value is 333333332833.388367 from busy waiting function, thread_count -> 8
The elapsed time -> 1.319570e+00, estimated value is 333333332833.388367 from mutex function, thread_count -> 8
The elapsed time -> 1.329967e+00, estimated value is 333333332833.388367 from semaphore function, thread_count -> 8

The elapsed time -> 1.659178e+00, estimated value is 333333332833.356689 from busy waiting function, thread_count -> 16
The elapsed time -> 1.584640e+00, estimated value is 333333332833.356628 from mutex function, thread_count -> 16
The elapsed time -> 1.289370e+00, estimated value is 333333332833.356689 from semaphore function, thread_count -> 16

The elapsed time -> 1.526100e+00, estimated value is 333333332833.360291 from busy waiting function, thread_count -> 32
The elapsed time -> 1.449422e+00, estimated value is 333333332833.360291 from mutex function, thread_count -> 32
The elapsed time -> 1.582835e+00, estimated value is 333333332833.360229 from semaphore function, thread_count -> 32

The elapsed time -> 2.188034e+00, estimated value is 333333332833.343872 from busy waiting function, thread_count -> 64
The elapsed time -> 1.267225e+00, estimated value is 333333332833.343689 from mutex function, thread_count -> 64
The elapsed time -> 1.358213e+00, estimated value is 333333332833.343811 from semaphore function, thread_count -> 64
```

Speedup table

Thread count	Busy waiting	Mutex	Semaphore	Speedup Busy Waiting	Speedup Mutex	Speedup Sempahore
1	2,96	2,93	2,93	1,00	1,00	1,00
2	1,47	1,47	1,48	2,01	1,99	1,99
4	1,29	1,53	1,62	2,29	1,92	1,81
8	1,54	1,32	1,33	1,93	2,22	2,21
16	1,66	1,58	1,29	1,79	1,85	2,27
32	1,53	1,45	1,58	1,94	2,02	1,85
64	2,19	1,27	1,36	1,35	2,31	2,16



Conclusion:

- The Busy-waiting loop is the simplest of all but has the worst performance among the three that’s serialize the multiple programs.
- Mutex and semaphore gave similar speedup overall, however, because in my program I have implemented semaphore as a mutex, a semaphore can be applied to a broader concept.
- Mutex is no prone to error but semaphore is, so mutex is simple to operate and implemented easily.

TASK 2

Code:

```c
/* File:
 * Compile:
 *    gcc -g -Wall -o 5.2 5.2.c -lpthread
 * Usage:
 *     5.2 <thread_count> <m> <n>
 */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "timer.h"

//#define PRINT1 //-----------------> define to see output of one-dim-matrix
//#define PRINT2 //-----------------> define to see output of two-dim-matrix
/* Global variables */
#define mm 8192
#define nn 8192
int m = 8192, n = 8192;
int     thread_count;
double* A;
double AA[mm][nn];
double* x;
double* y;
double* yy;
/* Serial functions */
void Usage(char* prog_name);
void Gen_matrix(double A[], int m, int n);
void Gen_vector(double x[], int n);
void Print_matrix_one_dim(char* title, double A[], int m, int n);
void Print_matrix_two_dim(char* title, double A[], int m, int n);
void Print_vector(char* title, double y[], double m);

/* Parallel function */
void *Pth_mat_vect_one_dim(void* rank);
void *Pth_mat_vect_two_dim(void* rank);

/*------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);


#  ifdef PRINT
   printf("thread_count =  %d, m = %d, n = %d\n", thread_count, m, n);
#  endif

   thread_handles = malloc(thread_count*sizeof(pthread_t));
   A = malloc(m*n*sizeof(double));
   x = malloc(n*sizeof(double));
   y = malloc(m*sizeof(double));
   yy = malloc(m*sizeof(double));
   Gen_matrix(A, m, n);
#ifdef PRINT1
   Print_matrix_one_dim("Generated one-dim matrix: ", A, m, n);
```

```c
#endif
  Gen_vector(x, n);

#ifdef PRINT1
  Print_vector("Generated one-dim matrix", x, n);
#endif
  double start, finish;
  GET_TIME(start);
  for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL,
            Pth_mat_vect_one_dim,  (void*) thread);

  for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);
  GET_TIME(finish);
#ifdef PRINT1
  Print_vector("The product is", y, m);
#endif
  printf("[One_Dimensional_Matrix] -> Elapsed time is %f\n", finish - start);
//-----------------------------------------------------------------------------------------------------------
  for (int i = 0; i < n; ++i) { // copying matrix to AA
    for (int j = 0; j < m; ++j) {
      AA[i][j] = A[i*m+j];
    }
  }
#ifdef PRINT2
  Print_matrix_two_dim("Generated two-dim matrix: ", A, m, n);
#endif

#ifdef PRINT2
  Print_vector("Generated two-dim matrix", x, n);
#endif

  GET_TIME(start);
  for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL,
            Pth_mat_vect_two_dim, (void*) thread);

  for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);
  GET_TIME(finish);
#ifdef PRINT2
  Print_vector("The product is", yy, m);
#endif

  printf("[Two_Dimensional_Matrix] -> Elapsed time is %f\n", finish - start);

  free(A);
  free(x);
  free(y);

  return 0;
}  /* main */


/*------------------------------------------------------------------
 * Function:  Usage
 * Purpose:   print a message showing what the command line should
 *          be, and terminate
 * In arg :   prog_name
 */
void Usage (char* prog_name) {
```

```
   fprintf(stderr, "usage: %s <thread_count>\n", prog_name);
   exit(0);
}  /* Usage */


/*------------------------------------------------------------------
 * Function: Gen_matrix
 * Purpose:  Use the random number generator random to generate
 *    the entries in A
 * In args:  m, n
 * Out arg:  A
 */
void Gen_matrix(double A[], int m, int n) {
   int i, j;
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         A[i*n+j] = random()/((double) RAND_MAX);
}  /* Gen_matrix */

/*------------------------------------------------------------------
 * Function: Gen_vector
 * Purpose:  Use the random number generator random to generate
 *    the entries in x
 * In arg:   n
 * Out arg:  A
 */
void Gen_vector(double x[], int n) {
   int i;
   for (i = 0; i < n; i++)
      x[i] = random()/((double) RAND_MAX);
}  /* Gen_vector */



/*------------------------------------------------------------------
 * Function:       Pth_mat_vect
 * Purpose:        Multiply an mxn matrix by an nx1 column vector
 * In arg:         rank
 * Global in vars: A, x, m, n, thread_count
 * Global out var: y
 */
void *Pth_mat_vect_one_dim(void* rank) {
   long my_rank = (long) rank;
   int i;
   int j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = my_first_row + local_m;


#  ifdef DEBUG
   printf("Thread %ld > local_m = %d, sub = %d\n",
       my_rank, local_m, sub);
#  endif


   for (i = my_first_row; i < my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++) {
        y[i] +=A[i*n+j];
      }
   }
```

```c
   return NULL;
}  /* Pth_mat_vect */

/*------------------------------------------------------------------
 * Function:       Pth_mat_vect
 * Purpose:        Multiply an mxn matrix by an nx1 column vector
 * In arg:         rank
 * Global in vars: A, x, m, n, thread_count
 * Global out var: y
 */
void *Pth_mat_vect_two_dim(void* rank) {
   long my_rank = (long) rank;
   int i;
   int j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = my_first_row + local_m;




#  ifdef DEBUG
   printf("Thread %ld > local_m = %d, sub = %d\n",
       my_rank, local_m, sub);
#  endif


   for (i = my_first_row; i < my_last_row; i++) {
      yy[i] = 0.0;
      for (j = 0; j < n; j++) {
        yy[i] += AA[i][j];
      }
   }



   return NULL;
}  /* Pth_mat_vect */



/*------------------------------------------------------------------
 * Function:    Print_matrix
 * Purpose:     Print the matrix
 * In args:     title, A, m, n
 */
void Print_matrix_one_dim( char* title, double A[], int m, int n) {
   int   i, j;

   printf("%s\n", title);
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%6.3f ", A[i*n + j]);
      printf("\n");
   }
}  /* Print_matrix */

void Print_matrix_two_dim( char* title, double A[], int m, int n) {
   int   i, j;
```

```c
   printf("%s\n", title);
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%6.3f ", AA[i][j]);
      printf("\n");
   }
}  /* Print_matrix */

/*------------------------------------------------------------------
 * Function:    Print_vector
 * Purpose:     Print a vector
 * In args:     title, y, m
 */
void Print_vector(char* title, double y[], double m) {
   int   i;

   printf("%s\n", title);
   for (i = 0; i < m; i++)
      printf("%6.3f ", y[i]);
   printf("\n");
}  /* Print_vector */
```

Result:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 5/5.2$ ./5.2 8
[One_Dimensional_Matrix] -> Elapsed time is 0.101242
[Two_Dimensional_Matrix] -> Elapsed time is 0.085074
```

Conclusion:

- Indeed, there is an improvement with the two-dimensional array declaration.
- Mostly it is due to the fact in two-dimensional array, elements are stored in row order, and there is no much mathematical calculation involved.

TASK 3:

Code:

```
/*
 * Compile:
 *    gcc -g -Wall -o 5.3 5.3.c -lpthread
 * Usage:
 *     pth_mat_vect no argument is required!!!
 */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "timer.h"

#define A_part
/*
 *        define A_part for 5.3.A
 *        define B_part for 5.3.B
 *
 */

/* Global variables */
int thread_count;
int m, n;
double *A;
double *x;
double *y;

/* Serial functions */
void Usage(char *prog_name);

void Gen_matrix(double A[], int m, int n);


void Gen_vector(double x[], int n);


void Print_matrix(char *title, double A[], int m, int n);

void Print_vector(char *title, double y[], double m);

/* Parallel function */
void *Pth_mat_vect(void *rank);

/*------------------------------------------------------------------*/
int main(int argc, char *argv[]) {
   long thread;
   pthread_t *thread_handles;

   if (argc != 1) Usage(argv[0]);
   thread_count = 4;
   m = 8;
   n = 8000000;

#  ifdef DEBUG
   printf("thread_count =  %d, m = %d, n = %d\n", thread_count, m, n);
#  endif

   thread_handles = malloc(thread_count * sizeof(pthread_t));
   A = malloc(m * n * sizeof(double));
   x = malloc(n * sizeof(double));
```

```
#ifdef A_part
   y = malloc((m + 8 * thread_count) * sizeof(double));
#endif
#ifdef B_part
   y = malloc((m + 8 * thread_count) * sizeof(double));
#endif
   Gen_matrix(A, m, n);
#  ifdef DEBUG
   Print_matrix("We generated", A, m, n);
#  endif

   Gen_vector(x, n);
#  ifdef DEBUG
   Print_vector("We generated", x, n);
#  endif
   double start, finish;
   GET_TIME(start);


   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
                  Pth_mat_vect, (void *) thread);

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);
   GET_TIME(finish);
   printf("Elapsed time = %e seconds\n", finish - start);
#  ifdef DEBUG
   Print_vector("The product is", y, m);
#  endif

   free(A);
   free(x);
   free(y);

   return 0;
}  /* main */


/*------------------------------------------------------------------
 * Function:  Usage
 * Purpose:   print a message showing what the command line should
 *            be, and terminate
 * In arg :   prog_name
 */
void Usage(char *prog_name) {
   fprintf(stderr, "usage: %s no argument is required\n", prog_name);
   exit(0);
}  /* Usage */



/*------------------------------------------------------------------
 * Function: Gen_matrix
 * Purpose:  Use the random number generator random to generate
 *    the entries in A
 * In args:  m, n
 * Out arg:  A
 */
void Gen_matrix(double A[], int m, int n) {
   int i, j;
   for (i = 0; i < m; i++)
```

```
      for (j = 0; j < n; j++)
         A[i * n + j] = random() / ((double) RAND_MAX);
}  /* Gen_matrix */

/*------------------------------------------------------------------
 * Function: Gen_vector
 * Purpose:  Use the random number generator random to generate
 *    the entries in x
 * In arg:   n
 * Out arg:  A
 */
void Gen_vector(double x[], int n) {
   int i;
   for (i = 0; i < n; i++)
      x[i] = random() / ((double) RAND_MAX);
}  /* Gen_vector */




/*------------------------------------------------------------------
 * Function:      Pth_mat_vect
 * Purpose:       Multiply an mxn matrix by an nx1 column vector
 * In arg:        rank
 * Global in vars: A, x, m, n, thread_count
 * Global out var: y
 */
void *Pth_mat_vect(void *rank) {
   long my_rank = (long) rank;
   int i;
   int j;
   int local_m = m / thread_count;
   int my_first_row = my_rank * local_m;
   int my_last_row = (my_rank + 1) * local_m - 1;
   double temp;


#  ifdef DEBUG
   printf("Thread %ld > local_m = %d, sub = %d\n",
       my_rank, local_m, sub);
#  endif


   for (i = my_first_row; i < my_last_row; i++) {

#ifdef A_part
      y[i + (my_rank * 8)] = 0.0;
#endif
#ifdef B_part
      y[i] = 0.0;
      temp = 0.0;
#endif
      for (j = 0; j < n; j++) {
#ifdef A_part
         y[i + (my_rank * 8)] += A[i * n + j] * x[j];
#endif
#ifdef B_part
         temp += A[i * n + j] * x[j];
#endif
      }
#ifdef B_part
      y[i] = temp;
```

```c
#endif
   }


   return NULL;
}  /* Pth_mat_vect */


/*------------------------------------------------------------------
 * Function:    Print_matrix
 * Purpose:     Print the matrix
 * In args:     title, A, m, n
 */
void Print_matrix(char *title, double A[], int m, int n) {
   int i, j;

   printf("%s\n", title);
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%6.3f ", A[i * n + j]);
      printf("\n");
   }
}  /* Print_matrix */


/*------------------------------------------------------------------
 * Function:    Print_vector
 * Purpose:     Print a vector
 * In args:     title, y, m
 */
void Print_vector(char *title, double y[], double m) {
   int i;

   printf("%s\n", title);
   for (i = 0; i < m; i++)
      printf("%6.3f ", y[i]);
   printf("\n");
}  /* Print_vector */
```

Result:

a) → padding

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 5/5.3$ ./5.3
Elapsed time = 5.823994e-02 seconds
```

b) → temp

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 5/5.3$ ./5.3
Elapsed time = 5.856705e-02 seconds
```

c) → original

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 5/5.3$ ./5.3
Elapsed time = 8.051801e-02 seconds
```

Conclusion:

- The original program has the worst performance among the three programs.
- The other two has an almost similar result, with a slightly better towards the padding.