P1.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include<omp.h>
 // #define SEED 123456

 int main()
{
   int N,I;
   printf("Enter number of iterations :");
   scanf("%d",&N);
   int count=0;
   double pi;

  // srand(SEED);

for(I=1;I<=8;I=I<<1) {
   double x1= omp_get_wtime();
   //Running the "simulation" for N times
   for (int i=0; i<N; i++) {

      omp_set_num_threads(I);
      //Getting the coordinates y,x ε [0,1]
      double x,y;
      x = (double)rand()/RAND_MAX;
      y = (double)rand()/RAND_MAX;

      //Checking if in unit circle
      if (x*x+y*y <= 1)
         count++;
   }

   //Calcuting the ratio and as a result the pi
   pi=(double)count/N*4;

   //printf("Single : # of trials = %14d , estimate of pi is %1.16f AND an absolute error of
%g\n",N,pi,fabs(pi - M_PI));
   double y1= omp_get_wtime();

   printf("Time taken for %d thread(s) for iteration %d: %lf\n",I,N,y1-x1);
}
   return 0;
}
```

P2.

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

/*
void printMatrix(int *arr[], int a, int b)
{
    int i, j;
    for (int i = 0; i < a; i++)
    {
        for (int j = 0; j < b; j++)
            printf("%d ", arr[i][j]);
        printf("\n");
    }
}
*/

int main(int argc, char *argv[])
{
    int a, b, m, n, i, j, k;
    printf("Enter the dimension of the first matrix\n");
    scanf("%d%d", &a, &b);
    printf("Enter the dimension of the second matrix\n");
    scanf("%d%d", &m, &n);

    int *arr1[1000], *arr2[1000];

    for (i = 0; i < a; i++)
        arr1[i] = (int *)malloc(b * sizeof(int));

    for (i = 0; i < m; i++)
        arr2[i] = (int *)malloc(n * sizeof(int));

    for (i = 0; i < a; i++)
        for (j = 0; j < b; j++)
            arr1[i][j] = rand() % 10;

    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            arr2[i][j] = rand() % 10;

    int *arr3[1000];
```

```
    for (i = 0; i < a; i++)
        arr3[i] = (int *)malloc(n * sizeof(int));

    int l;
    for (l = 1; l <= 8; l = l << 1)
    {
        double x = omp_get_wtime();
        omp_set_num_threads(l);
#pragma omp parallel for private(j, k)
        for (i = 0; i < a; i++)
            for (j = 0; j < n; j++)
            {
                arr3[i][j] = 0;
                for (k = 0; k < b; k++)
                    arr3[i][j] += arr1[i][k] * arr2[k][j];
            }
        double y = omp_get_wtime();

        printf("Time taken for %d thread(s) for size %d: %lf\n", l, a, y - x);
    }
    /*
            printf("Matrix A:\n");
            printMatrix(arr1,a,b);

            printf("Matrix B:\n");
            printMatrix(arr2,m,n);

            printf("Matrix C:\n");
            printMatrix(arr3,a,n);*/
    return 0;
}
```

P3.

```
#include <math.h>
#include <string.h>
#include <omp.h>
#include <iostream>
using namespace std;
double t = 0.0;

inline long Strike(bool composite[], long i, long stride, long limit)
{

  for (; i <= limit; i += stride)
    composite[i] = true;
  return i;
}

long CacheUnfriendlySieve(long n)

{

  long count = 0;
  long m = (long)sqrt((double)n);
  bool *composite = new bool[n + 1];

  memset(composite, 0, n);

  t = omp_get_wtime();

  for (long i = 2; i <= m; ++i)

    if (!composite[i])
    {

      ++count;
      // Strike walks array ofsize n here.

      Strike(composite, 2 * i, i, n);
    }

  for (long i = m + 1; i <= n; ++i)
    if (!composite[i])
    {
      ++count;
    }
```

```
  t = omp_get_wtime() - t;

  delete[] composite;

  return count;
}

long CacheFriendlySieve(long n)

{

  long count = 0;

  long m = (long)sqrt((double)n);

  bool *composite = new bool[n + 1];

  memset(composite, 0, n);

  long *factor = new long[m];

  long *striker = new long[m];

  long n_factor = 0;

  t = omp_get_wtime();

  for (long i = 2; i <= m; ++i)
    if (!composite[i])
    {
      ++count;
      striker[n_factor] = Strike(composite, 2 * i, i, m);
      factor[n_factor++] = i;
    }

  // Chops sieve into windows of size ≈ sqrt(n)

  for (long window = m + 1; window <= n; window += m)
  {
    long limit = min(window + m - 1, n);

    for (long k = 0; k < n_factor; ++k)
      striker[k] = Strike(composite, striker[k], factor[k], limit);
```

```
      for (long i = window; i <= limit; ++i)
    if (!composite[i])
      ++count;
  }

  t = omp_get_wtime() - t;

  delete[] striker;

  delete[] factor;

  delete[] composite;

  return count;
}

long ParallelSieve(long n)

{
  long count = 0;

  long m = (long)sqrt((double)n);

  long n_factor = 0;

  long *factor = new long[m];

  t = omp_get_wtime();
#pragma omp parallel
  {

    bool *composite = new bool[m + 1];

    long *striker = new long[m];

#pragma omp single

    {

      memset(composite, 0, m);

      for (long i = 2; i <= m; ++i)

        if (!composite[i])
```

```
        {
          ++count;
          Strike(composite, 2 * i, i, m);
          factor[n_factor++] = i;
        }
    }

    long base = -1;

#pragma omp for reduction(+ : count)

    for (long window = m + 1; window <= n; window += m)
    {
      memset(composite, 0, m);
      if (base != window)
      {
        base = window;
        for (long k = 0; k < n_factor; ++k)
          striker[k] = (base + factor[k] - 1) / factor[k] * factor[k] - base;
      }

      long limit = min(window + m - 1, n) - base;

      for (long k = 0; k < n_factor; ++k)
        striker[k] = Strike(composite, striker[k], factor[k], limit) - m;

      for (long i = 0; i <= limit; ++i)
        if (!composite[i])
          ++count;
      base += m;
    }

    delete[] striker;

    delete[] composite;
  }

  t = omp_get_wtime() - t;

  delete[] factor;

  return count;
}
```

```
int main()

{

  long count1 = CacheUnfriendlySieve(10000000);
  cout << "Unfriendly " << count1 << endl;
  cout << "Time : " << t << endl;

  long count2 = CacheFriendlySieve(10000000);
  cout << "Friendly " << count2 << endl;
  cout << "Time : " << t << endl;

  long count3 = ParallelSieve(10000000);
  cout << "Parallel " << count3 << endl;
  cout << "Time : " << t << endl;
}
```

P4.

```
#include <stdio.h>
#include <gd.h>
#include <error.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int nt = 4;
int tid,tmp,red,green,blue,color,x,h,y,w;
tmp=red=green=blue=color=x=h=y=w=0;
char *iname =NULL;
char *oname = NULL;
gdImagePtr img;
FILE *fp={0};
if(argc!=3)
error(1,0,"format : object_file input.png output.png");
else
{
iname = argv[1];
oname = argv[2];
}
if((fp=fopen(iname,"r"))==NULL)
error(1,0,"error : fopen : %s",iname);
else
{
img = gdImageCreateFromPng(fp);
w=gdImageSX(img);
h=gdImageSY(img);
}
double t=omp_get_wtime();
omp_set_num_threads(nt);
#pragma omp parallel for private(y,color,red,blue,green)
schedule(static,10)/schedule(dynamic,50) schedule(guided,50)/
for(x=0;x<w;x++)
for(y=0;y<h;y++)
{
tid= omp_get_thread_num();

color=gdImageGetPixel(img,x,y);
red=gdImageRed(img,color);
green=gdImageGreen(img,color);
blue=gdImageBlue(img,color);
tmp=(red+green+blue)/3;
red=green=blue=tmp;
```

```
color=gdImageColorAllocate(img,red,green,blue);
gdImageSetPixel(img,x,y,color);

/*if(tid==1)
{
color=gdImageColorAllocate(img,0,green,0);
gdImageSetPixel(img,x,y,color);
}
if(tid==2)
{
color=gdImageColorAllocate(img,0,0,blue);
gdImageSetPixel(img,x,y,color);
}*/

}
t=omp_get_wtime()-t;
printf("\ntime taken : %lf threads : %d",t,nt);
if((fp=fopen(oname,"w"))==NULL)
error(1,0,"error : fopen : %s",oname);
else
{
gdImagePng(img,fp);
fclose(fp);
}
gdImageDestroy(img);
return 0;
}
```

P5.

```
#include<stdio.h>
#include<mpi.h>
#include<string.h>
#define BUFFER_SIZE 32
int main(int argc, char** argv)
{
	int MyRank,Numprocs,Destination,iproc;
	int tag = 0;
	int Root = 0, temp=1;
	char Message[BUFFER_SIZE];
	MPI_Init(&argc,&argv);
	MPI_Status status;
	MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);
	MPI_Comm_size(MPI_COMM_WORLD,&Numprocs);

	if(MyRank==0)
	{
		system("hostname");
		for(temp=1;temp<Numprocs;temp++)
		{

MPI_Recv(Message,BUFFER_SIZE,MPI_CHAR,temp,tag,MPI_COMM_WORLD,&status);
			printf("\n%s in process with rank %d from process with rank
%d\n",Message,temp,Root);
		}
	}

	else
	{	system("hostname");
	  if(MyRank==1){
		strcpy(Message,"Hello");
		MPI_Send(Message,BUFFER_SIZE,MPI_CHAR,Root,tag,MPI_COMM_WORLD);
	   }
	 if(MyRank==2){
		strcpy(Message,"RVCE");
		MPI_Send(Message,BUFFER_SIZE,MPI_CHAR,Root,tag,MPI_COMM_WORLD);
	   }
	 if(MyRank==3){
		strcpy(Message,"CSE");
		MPI_Send(Message,BUFFER_SIZE,MPI_CHAR,Root,tag,MPI_COMM_WORLD);
	   }

	}

	MPI_Finalize();
}
```

P6.

```
#include<stdio.h>
#include<omp.h>
#include<string.h>
#define COUNT 10
#define FILE_NAME "words.txt"

char search_words[20][COUNT] =
{"The","wall","to","from","by","different","any","are","various","of"};
long counts[COUNT];
int line_c = 0;

int is_alpha(char c) {
return ((c >=  65 && c <=  90) || (c >=  97 && c <=  122));
}

int is_equal(char* a,const char* key, int ignore_case) {
char b[20];
strcpy(b,key);
int len_a = strlen(a),len_b = strlen(b);

if(len_a !=  len_b) {
return 0;
}
if(ignore_case != 0) {
int i;
#pragma omp parallel for shared(a) private(i)
for(i = 0; i < len_a; i++) {
if(a[i] > 90)
a[i] -=  32;
}
#pragma omp parallel for shared(b) private(i)
for(i = 0; i < len_b; i++) {
if(b[i] > 90)
b[i] -=  32;
}
}
return (strcmp(a,b) ==  0);
}
void read_word(char *temp, FILE *fp) {
int i = 0;
char ch;
while((ch = fgetc(fp)) !=  EOF && is_alpha(ch) ==  0);
```

```
while(ch !=  EOF && is_alpha(ch) !=  0) {
temp[i++] = ch;
ch = fgetc(fp);
}

temp[i] = '\0';
}

long determine_count(const char *file_name, const char *key, int ignore_case) {
int key_index = 0,key_len = strlen(key);
long word_count = 0;
char ch;
FILE *fp = fopen(file_name,"r");
char temp[40];
int i = 0;
while(feof(fp) ==  0) {
read_word(temp,fp);
if(is_equal(temp,key,ignore_case) !=  0)
word_count++;
//printf("%s ",temp);
}
//printf("\nWord %s: %ld",key,word_count);
return word_count;
}

int main() {
int i;
int nt = 0;
//#pragma omp parallel for shared(counts, search_words) private(i) num_threads(nt)
for(nt=1;nt<=8;nt=nt*2){
#pragma omp parallel for shared(counts, search_words) private(i) num_threads(nt)
for(i = 0; i < COUNT; i++) { counts[i] = 0; }

double t = omp_get_wtime();
#pragma omp parallel for shared(counts, search_words) private(i) num_threads(nt)
for(i = 0; i < COUNT; i++) {
counts[i] = determine_count(FILE_NAME,search_words[i],1);
}

t = omp_get_wtime() - t;

for(i = 0; i < COUNT; i++) { printf("\n%s: %ld",search_words[i],counts[i]); }
printf("\nNo of threads: %d, Time Taken:%lf\n",nt,t);}
}
```

P7.

```
#include <stdio.h>
#include <stdlib.h>
#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#include <time.h>
#endif
#define VECTOR_SIZE 4024

//OpenCL kernel which is run for every work item created.
// TODO: Add OpenCL kernel code here.
const char* saxpy_kernel =
"__kernel                              \n"
"void saxpy_kernel(float alpha,     \n"
"                  __global float *A,       \n"
"                  __global float *B,       \n"
"                  __global float *C)       \n"
"{                                      \n"
"    //Get the index of the work-item       \n"
"    int index = get_global_id(0);          \n"
"    C[index] = alpha* A[index] + B[index]; \n"
"}                                      \n";

int main(void) {
   int i;
   // Allocate space for vectors A, B and C
   float alpha = 2.0;
   float* A = (float*)malloc(sizeof(float) * VECTOR_SIZE);
   float* B = (float*)malloc(sizeof(float) * VECTOR_SIZE);
   float* C = (float*)malloc(sizeof(float) * VECTOR_SIZE);
   for (i = 0; i < VECTOR_SIZE; i++)
   {
      A[i] = i;
      B[i] = VECTOR_SIZE - i;
      C[i] = 0;
   }

   clock_t tStart = clock();
   // Get platform and device information
   cl_platform_id* platforms = NULL;
   cl_uint     num_platforms;
```

```
    //Set up the Platform
    cl_int clStatus = clGetPlatformIDs(0, NULL, &num_platforms);
    platforms = (cl_platform_id*)
       malloc(sizeof(cl_platform_id) * num_platforms);
    clStatus = clGetPlatformIDs(num_platforms, platforms, NULL);

    //Get the devices list and choose the device you want to run on
    cl_device_id* device_list = NULL;
    cl_uint           num_devices;

    clStatus = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, 0, NULL, &num_devices);
    device_list = (cl_device_id*)
       malloc(sizeof(cl_device_id) * num_devices);
    clStatus = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, num_devices, device_list,
NULL);

    // Create one OpenCL context for each device in the platform
    cl_context context;
    context = clCreateContext(NULL, num_devices, device_list, NULL, NULL, &clStatus);

    // Create a command queue
    cl_command_queue command_queue = clCreateCommandQueueWithProperties(context,
device_list[0], 0, &clStatus);

    // Create memory buffers on the device for each vector
    cl_mem A_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY, VECTOR_SIZE *
sizeof(float), NULL, &clStatus);
    cl_mem B_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY, VECTOR_SIZE *
sizeof(float), NULL, &clStatus);
    cl_mem C_clmem = clCreateBuffer(context, CL_MEM_WRITE_ONLY, VECTOR_SIZE *
sizeof(float), NULL, &clStatus);

    // Copy the Buffer A and B to the device
    clStatus = clEnqueueWriteBuffer(command_queue, A_clmem, CL_TRUE, 0, VECTOR_SIZE
* sizeof(float), A, 0, NULL, NULL);
    clStatus = clEnqueueWriteBuffer(command_queue, B_clmem, CL_TRUE, 0, VECTOR_SIZE
* sizeof(float), B, 0, NULL, NULL);

    // Create a program from the kernel source
    cl_program program = clCreateProgramWithSource(context, 1, (const char**)&saxpy_kernel,
NULL, &clStatus);

    // Build the program
    clStatus = clBuildProgram(program, 1, device_list, NULL, NULL, NULL);
```

```
    // Create the OpenCL kernel
    cl_kernel kernel = clCreateKernel(program, "saxpy_kernel", &clStatus);

    // Set the arguments of the kernel
    clStatus = clSetKernelArg(kernel, 0, sizeof(float), (void*)&alpha);
    clStatus = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&A_clmem);
    clStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&B_clmem);
    clStatus = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void*)&C_clmem);



    // Execute the OpenCL kernel on the list
    size_t global_size = VECTOR_SIZE; // Process the entire lists
    size_t local_size = 64;           // Process one item at a time
    clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size,
&local_size, 0, NULL, NULL);

    // Read the cl memory C_clmem on device to the host variable C
    clStatus = clEnqueueReadBuffer(command_queue, C_clmem, CL_TRUE, 0, VECTOR_SIZE
* sizeof(float), C, 0, NULL, NULL);

    // Clean up and wait for all the comands to complete.
    clStatus = clFlush(command_queue);
    clStatus = clFinish(command_queue);

    // Display the result to the screen
    //for (i = 0; i < VECTOR_SIZE; i++)
        //printf("%f * %f + %f = %f\n", alpha, A[i], B[i], C[i]);

    printf("Input : %d \nTime taken: %f\n", VECTOR_SIZE,(double)(clock() - tStart) /
CLOCKS_PER_SEC);

    // Finally release all OpenCL allocated objects and host buffers.
    clStatus = clReleaseKernel(kernel);
    clStatus = clReleaseProgram(program);
    clStatus = clReleaseMemObject(A_clmem);
    clStatus = clReleaseMemObject(B_clmem);
    clStatus = clReleaseMemObject(C_clmem);
    clStatus = clReleaseCommandQueue(command_queue);
    clStatus = clReleaseContext(context);
    free(A);
    free(B);
    free(C);
```

```
    free(platforms);
    free(device_list);
    return 0;
}
```