NETAJI SUBHASH UNIVERSITY OF TECHNOLOGY
SECTOR-3, DWARKA, NEW DELHI-110078

High Performance Computing
(COCSC18)

Practical File | Semester 6

Virender Singh Nayal
2020UCO1545
COE-1

Index

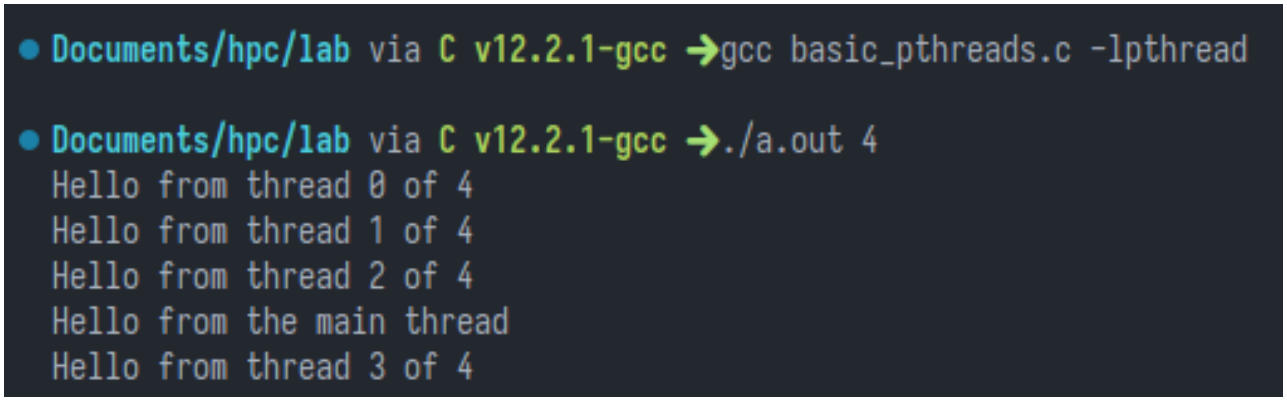
Experiment 1

Basic hello world program using pthreads.

Code :

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

int thread_count;
void *Hello(void *rank);

int main(int argc, char *argv[])
{
    long thread;
    pthread_t *thread_handles;
    thread_count = strtol(argv[1], NULL, 10);
    thread_handles = malloc(thread_count * sizeof(pthread_t));
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Hello,
(void *)thread);
    printf("Hello from the main thread\n");
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
    free(thread_handles);
    return 0;
}

void *Hello(void *rank)
{
    long my_rank = (long)rank;
    printf("Hello from thread %ld of %d\n", my_rank,
thread_count);
    return NULL;
}
```

Output :


Documents/hpc/lab via C v12.2.1-gcc ➜gcc basic_pthreads.c -lpthread
Documents/hpc/lab via C v12.2.1-gcc ➜./a.out 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from the main thread
Hello from thread 3 of 4

Experiment 2

Program to find the sum of all elements of an array using 2 processors.

Code :

```
#include "/usr/include/mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char **argv)
{
    MPI_Init(NULL, NULL);
    int num_procs;
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
    {
        int n;
        printf("Enter number of elements : ");
        scanf("%d", &n);
        int arr[n];
        for (int i = 0; i < n; i++)
        {
            arr[i] = rand() % 10000 + 1;
        }
        printf("Array is -\n [ ");
        for (int i = 0; i < n; i++)
        {
            printf("%d ", arr[i]);
        }
        printf("]\n");
        int elem_to_send = n / 2;
```

```
        if (n % 2)
            elem_to_send++;
        MPI_Send(&elem_to_send, 1, MPI_INT, 1, 0,
MPI_COMM_WORLD);
        MPI_Send(&arr[n / 2], elem_to_send, MPI_INT, 1, 1,
MPI_COMM_WORLD);
        float t1 = clock();
        int local = 0;
        for (int i = 0; i < n / 2; i++)
            local = local + arr[i];
        int s_rec = 0;
        float t2 = clock();
        printf("Time taken by process %d : %f\n", rank, (t2 -
t1) / CLOCKS_PER_SEC);
        MPI_Recv(&s_rec, 1, MPI_INT, 1, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        local = local + s_rec;
        printf("Total sum of array is %d\n", local);
    }
    else
    {
        float t1 = clock();
        int size;
        MPI_Recv(&size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int arr[size];
        MPI_Recv(arr, size, MPI_INT, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        float t2 = clock();
        printf("Total time for recieving : %f", (t2 - t1) /
CLOCKS_PER_SEC);
        t1 = clock();
        int local = 0;
        for (int i = 0; i < size; i++)
            local = local + arr[i];
```

```
        printf("\nProcess %d sending sum %d back to main...\n",
rank, local);
        t2 = clock();
        printf("Time taken by process for addition %d : %f\n",
rank, (t2 - t1) / CLOCKS_PER_SEC);
        MPI_Send(&local, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Output :

```
● Documents/hpc/lab via C v12.2.1-gcc ➜mpicc sum_array_two_p.c

● Documents/hpc/lab via C v12.2.1-gcc ➜mpirun -np 2 ./a.out
  4
  Enter number of elements : Array is -
   [ 9384 887 2778 6916 ]
  Enter number of elements : Total time for recieving : 1.832200
  Process 1 sending sum 9694 back to main...
  Time taken by process for addition 1 : 0.000014
  Time taken by process 0 : 0.000003
  Total sum of array is 19965
```

Experiment 3

Compute the sum of all the elements of an array using p processors.

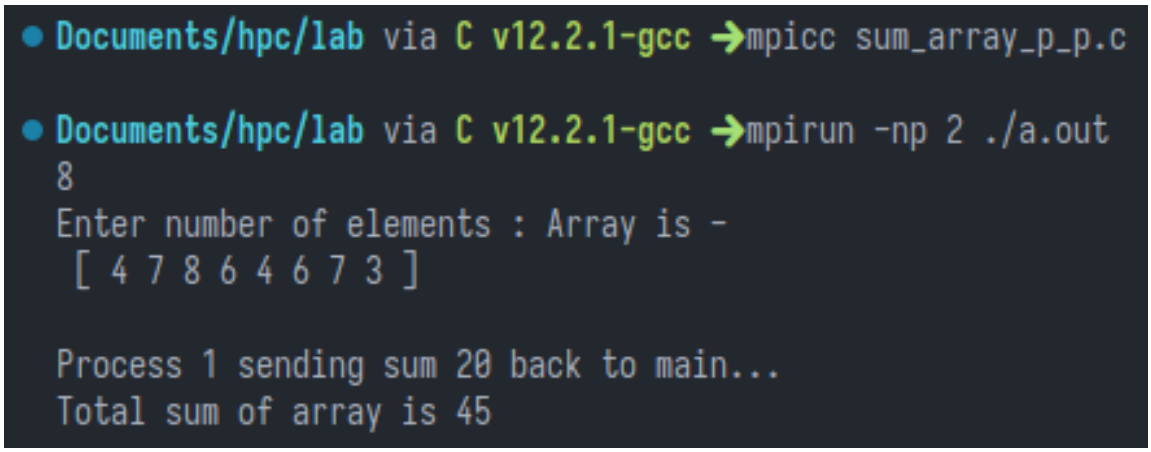
Code :

```
#include "/usr/include/mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(int argc, char **argv)
{
    MPI_Init(NULL, NULL);
    int num_procs;
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
    {
        int n;
        printf("Enter number of elements : ");
        scanf("%d", &n);
        int arr[n];
        for (int i = 0; i < n; i++)
        {
            arr[i] = rand() % 10 + 1;
        }
        printf("Array is -\n [ ");
        for (int i = 0; i < n; i++)
        {
            printf("%d ", arr[i]);
        }
        printf("]\n");
        int elem_to_send = n / num_procs;
        int tag = 0;
```

```
        for (int i = 1; i < num_procs; i++)
        {
            if (i != num_procs - 1)
            {
                elem_to_send = n / num_procs;
                MPI_Send(&elem_to_send, 1, MPI_INT, i, i +
num_procs, MPI_COMM_WORLD);
                MPI_Send(&arr[i * (elem_to_send)], elem_to_send,
MPI_INT, i, i + num_procs + 1, MPI_COMM_WORLD);
                continue;
            }
            elem_to_send = n / num_procs + n % num_procs;
            MPI_Send(&elem_to_send, 1, MPI_INT, i, i +
num_procs, MPI_COMM_WORLD);
            MPI_Send(&arr[(num_procs - 1) * (n / num_procs)],
elem_to_send, MPI_INT, i, i + num_procs + 1, MPI_COMM_WORLD);
        }
        int ans = 0;
        for (int i = 0; i < n / num_procs; i++)
            ans += arr[i];
        int s_rec;
        for (int i = 1; i < num_procs; i++)
        {
            s_rec = 0;
            MPI_Recv(&s_rec, 1, MPI_INT, i, i + num_procs + 2,
MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
            ans += s_rec;
        }
        printf("Total sum of array is %d\n", ans);
    }
    else
    {
        int size;
```

```
        MPI_Recv(&size, 1, MPI_INT, 0, rank + num_procs,
MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        int arr[size];
        MPI_Recv(arr, size, MPI_INT, 0, rank + num_procs + 1,
MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        int local = 0;
        for (int i = 0; i < size; i++)
            local = local + arr[i];
        printf("\nProcess %d sending sum %d back to main...\n",
rank, local);
        MPI_Send(&local, 1, MPI_INT, 0, rank + num_procs + 2,
MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Output :



Experiment 4

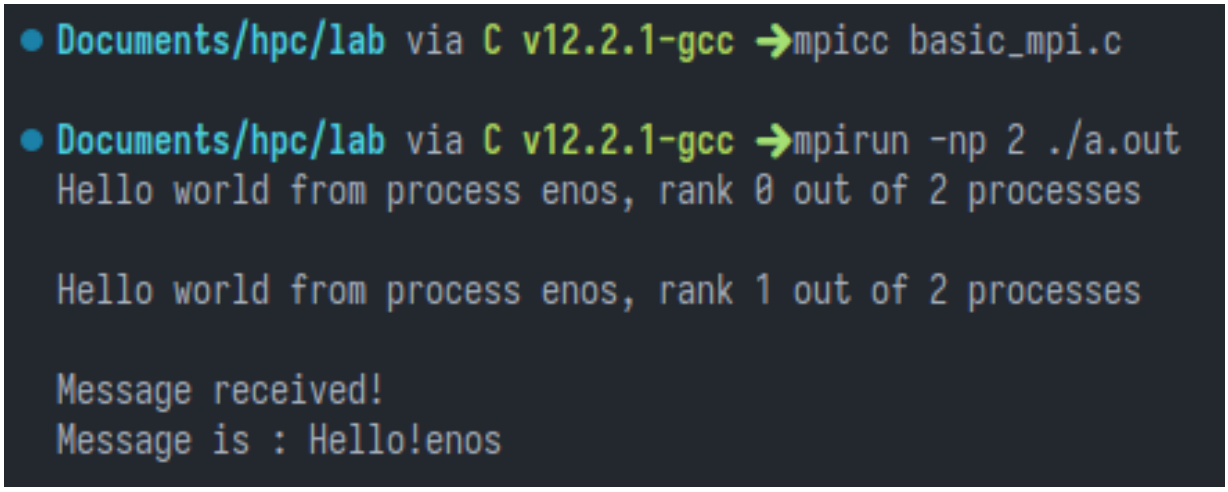
Program to illustrate basic MPI communication routines.

Code :

```
#include "/usr/include/mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    MPI_Init(NULL, NULL);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    printf("Hello world from process %s, rank %d out of %d
processes\n\n",
           processor_name, world_rank, world_size);
    if (world_rank == 0)
    {
        char *message = "Hello!";
        MPI_Send(message, 6, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else
    {
        char message[6];
        MPI_Recv(message, 6, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Message received!\n");
        printf("Message is : %s\n", message);
    }
    MPI_Finalize();
```

```
    return 0;
}
```

Output :



Experiment 5

Parallel program for matrix multiplication and show logging and tracing MPI activity.

Code :

Experiment 6

C program with openMP to implement loop work sharing

Code :

```
#include <omp.h>
#include <stdio.h>
void reset_freq(int *freq, int THREADS)
{
    for (int i = 0; i < THREADS; i++)
        freq[i] = 0;
}
int main(int *argc, char **argv)
{
    int n, THREADS, i;
    printf("Enter the number of iterations :");
    scanf("%d", &n);
    printf("Enter the number of threads (max 8): ");
    scanf("%d", &THREADS);
    int freq[THREADS];
    reset_freq(freq, THREADS);
#pragma omp parallel for num_threads(THREADS)
    for (i = 0; i < n; i++)
    {
        freq[omp_get_thread_num()]++;
    }
#pragma omp barrier
    printf("\nIn default scheduling, we have the following
thread distribution :- \n");
    for (int i = 0; i < THREADS; i++)
    {
        printf("Thread %d : %d iters\n", i, freq[i]);
    }
    int CHUNK;
    printf("\nUsing static scheduling...\n");
    printf("Enter the chunk size :");
    scanf("%d", &CHUNK);
    reset_freq(freq, THREADS);
#pragma omp parallel for num_threads(THREADS) schedule(static,
```

```
CHUNK)
    for (i = 0; i < n; i++)
    {
        freq[omp_get_thread_num()]++;
    }
#pragma omp barrier
    printf("\nIn static scheduling, we have the following thread
distribution :- \n");
    for (int i = 0; i < THREADS; i++)
    {
        printf("Thread %d : %d iters\n", i, freq[i]);
    }
    printf("\nUsing automatic scheduling...\n");
    reset_freq(freq, THREADS);
#pragma omp parallel for num_threads(THREADS) schedule(auto)
    for (i = 0; i < n; i++)
    {
        freq[omp_get_thread_num()]++;
    }
#pragma omp barrier
    printf("In auto scheduling, we have the following thread
distribution :- \n");
    for (int i = 0; i < THREADS; i++)
    {
        printf("Thread %d : %d iters\n", i, freq[i]);
    }
    return 0;
}
```

Output :

```
● Documents/hpc/lab via C v12.2.1-gcc ➜./a.out
  Enter the number of iterations :6
  Enter the number of threads (max 8): 7

  In default scheduling, we have the following thread distribution :-
  Thread 0 : 1 iters
  Thread 1 : 1 iters
  Thread 2 : 1 iters
  Thread 3 : 1 iters
  Thread 4 : 1 iters
  Thread 5 : 1 iters
  Thread 6 : 0 iters

  Using static scheduling...
  Enter the chunk size :20

  In static scheduling, we have the following thread distribution :-
  Thread 0 : 6 iters
  Thread 1 : 0 iters
  Thread 2 : 0 iters
  Thread 3 : 0 iters
  Thread 4 : 0 iters
  Thread 5 : 0 iters
  Thread 6 : 0 iters

  Using automatic scheduling...
  In auto scheduling, we have the following thread distribution :-
  Thread 0 : 1 iters
  Thread 1 : 1 iters
  Thread 2 : 1 iters
  Thread 3 : 1 iters
  Thread 4 : 1 iters
  Thread 5 : 1 iters
  Thread 6 : 0 iters
```

Experiment 7

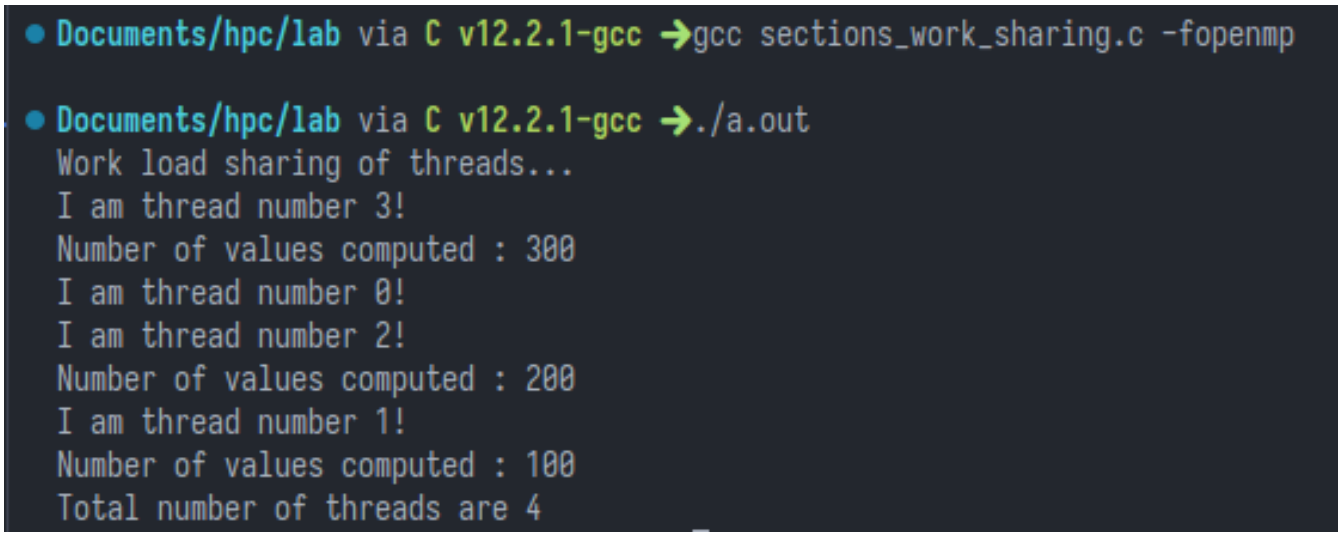
C program with openMP to implement sections work sharing

Code :

```
#include <omp.h>

#include <stdio.h>

int main(int *argc, char **argv)

{

    int num_threads, THREAD_COUNT = 4;

    int thread_ID;

    int section_sizes[4] = {

        0, 100, 200, 300};

    printf("Work load sharing of threads...\n");

#pragma omp parallel private(thread_ID)
num_threads(THREAD_COUNT)

    {

        thread_ID = omp_get_thread_num();

        printf("I am thread number %d!\n", thread_ID);

        int value_count = 0;

        if (thread_ID > 0)

        {

            int work_load = section_sizes[thread_ID];

            for (int i = 0; i < work_load; i++)

                value_count++;

            printf("Number of values computed : %d\n",
```

```
value_count);

        }

#pragma omp barrier

        if (thread_ID == 0)

        {

            printf("Total number of threads are %d",
omp_get_num_threads());

        }

    }



    return 0;

}
```

Output :



Experiment 8

Program to illustrate process synchronization and collective data movements.

Code :

```
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>


int thread_count;


struct arguments

{

    int size;

    int *arr1;

    int *arr2;

    int *dot;

};


void *add_into_one(void *arguments);


void print_vector(int n, int *arr)

{
```

```
    printf("[ ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("] \n");
}

int main(int argc, char *argv[])
{
    long thread;
    pthread_t *thread_handles;
    thread_count = 2;
    thread_handles = malloc(thread_count * sizeof(pthread_t));
    printf("Enter the size of the vectors : ");
    int n;
    scanf("%d", &n);
    printf("Enter the max_val of the vectors : ");
    int max_val;
    scanf("%d", &max_val);
    struct arguments *args[2];
    for (int i = 0; i < 2; i++)
    {
        args[i] = malloc(sizeof(struct arguments) * 1);
        args[i]->size = n;
```

```
        args[i]->arr1 = malloc(sizeof(int) * n);
        args[i]->arr2 = malloc(sizeof(int) * n);
        args[i]->dot = malloc(sizeof(int) * n);
        for (int j = 0; j < n; j++)
            {
                args[i]->arr1[j] = rand() % max_val;
                args[i]->arr2[j] = rand() % max_val;
            }
    }
    printf("Vectors are : \n");
    print_vector(n, args[0]->arr1);
    print_vector(n, args[0]->arr2);
    print_vector(n, args[1]->arr1);
    print_vector(n, args[1]->arr2);
    int result[n];
    memset(result, 0, n * sizeof(int));
    for (thread = 0; thread < thread_count; thread++)
    {
        printf("Multiplying %ld and %ld with thread %ld...\n",
thread + 1, thread + 2,
               thread);
        pthread_create(&thread_handles[thread], NULL,
add_into_one, (void *)args[thread]);
    }
```

```
    printf("Hello from the main thread\n");
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
    for (int i = 0; i < 2; i++)
    {
        printf("Multiplication for vector %d and %d \n", i + 1,
i + 2);
        print_vector(n, args[i]->dot);
        printf("\n");
    }
    free(thread_handles);
    for (int i = 0; i < n; i++)
        result[i] = args[0]->dot[i] + args[1]->dot[i];
    printf("Result is : \n");
    print_vector(n, result);
    return 0;
}

void *add_into_one(void *argument)
{
    struct arguments *args = argument;
    int n = args->size;
    for (int i = 0; i < n; i++)
        args->dot[i] = args->arr1[i] * args->arr2[i];
```

```
    return NULL;
}
```

Output :