ASSIGNMENT
ONE

SHARED MEMORIES

Batool Shilleh

11923748

CODE EXPLANATION

Libraries that were used in the program

Libraries

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <time.h>
#include <sys/stat.h>
#include <fcntl.h>
```

CODE EXPLANATION

Define the necessary variables in the program

- J , I : for loop
- N : The number of elements of the matrix = university number/10
- M : Process number = the last digit of the registration number + 5
- Remainder , portion_size : calculate remind operation
- Result: array which is not shared.
- Pid[M] , wpid : Parent IP
- start_time, end_time, start_time_shared, end_time_shared , time_taken,time_taken_shared: Speed calculation
- Statr_index , end_index : start point for process and end point

Define variables

```
//Define variables
      int i, j,rank;

      const int N = 11923748/10;

      const int M = 5+8;

      int remainder=N%M;

      int portion_size = N / M;

      double *result = malloc(sizeof(double) * N);

       pid_t pid[M], wpid;

      int state =0;

      clock_t start_time, end_time,
      start_time_shared, end_time_shared;

      double time_taken,time_taken_shared;

      int start_index ,end_index;
```

CODE EXPLANATION

Define the shared memory

sharedarray

```
//sharedarray
const char* SHARED_ARRAY_1_NAME = "/shared_array_1";

const char* SHARED_ARRAY_2_NAME = "/shared_array_2";

const char* SHARED_RESULT_NAME = "/shared_result";

// create shared memory segment for shared_array_1

int shared_array_1_fd =
shm_open(SHARED_ARRAY_1_NAME, O_CREAT | O_RDWR, 0666);

ftruncate(shared_array_1_fd, N * sizeof(double));

double* shared_array_1 =
(double*)mmap(NULL, N * sizeof(double),PROT_READ | PROT_WRITE,
 MAP_SHARED, shared_array_1_fd, 0);

// create shared memory segment for shared_array_2

int shared_array_2_fd =
shm_open(SHARED_ARRAY_2_NAME, O_CREAT | O_RDWR, 0666);

ftruncate(shared_array_2_fd, N * sizeof(double));

double* shared_array_2 =
(double*)mmap(NULL, N * sizeof(double), PROT_READ | PROT_WRITE,
MAP_SHARED,
 shared_array_2_fd, 0);

// create shared memory segment for shared_result

int shared_result_fd =
shm_open(SHARED_RESULT_NAME, O_CREAT | O_RDWR, 0666);

ftruncate(shared_result_fd, N * sizeof(double));

double* shared_result =
(double*)mmap(NULL, N * sizeof(double), PROT_READ | PROT_WRITE,
MAP_SHARED,
 shared_result_fd, 0);
```

CODE EXPLANATION

Fill the two arrays with random elements

Initialize arrays

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

shared_array_1[i] =
(double)rand()/(double)(RAND_MAX/N);

shared_array_2[i] =
(double)rand()/(double)(RAND_MAX/N);
        }
```

CODE EXPLANATION

Compute the sum for the current process

- start_time : to calculate the start time
- We start with the first loop, which will be implemented according to the number of children to be created
- We calculate whether the operations will be divided equally between the children or we will need to perform them in the parent
- end_time : to calculate the end time
- time_taken : Speed calculation

child processes

```
//Create M number of child processes
start_time_shared = clock();
for (i =0 ; i < M; i++){

        pid[i] = fork();
        if (pid[i] == 0) {
        // Determine the portion of the array to process
            if (i < remainder) {
              start_index = i * (portion_size + 1);
              end_index = start_index + portion_size;
                }
           else {
                start_index = i * portion_size + remainder;
                end_index = start_index + portion_size - 1;
    }

    // Compute the sum for the current process
    for (int j = start_index; j <= end_index; j++) {
        shared_result[j] = shared_array_1[j] + shared_array_2[j];
    }
      exit(0);
  }
    // Parent process
      else {
    continue;
    }

}
```

CODE EXPLANATION

parent process waits for all processes to finish

- This loop waits for child processes to finish and uses a waitpid function to check for this
- The perror function is used to print an error when there are problems in the child process
- end_time_shared : to calculate the end time
- time_taken_shared : Speed calculation

Wait

```
for(i=0;i<M;i++){

   wpid = waitpid(pid[i],&state,0);

    if(wpid == -1){

       perror("waitpid");

       exit(1);

       }

     }

end_time_shared = clock();

 time_taken_shared =
 ((double) (end_time_shared - start_time_shared))
 / CLOCKS_PER_SEC;
```

CODE EXPLANATION

Check the result

- We check if the values in an array “result" are equal to the values in an array “result_shared"
- We print the time for normal addition and parallel addition

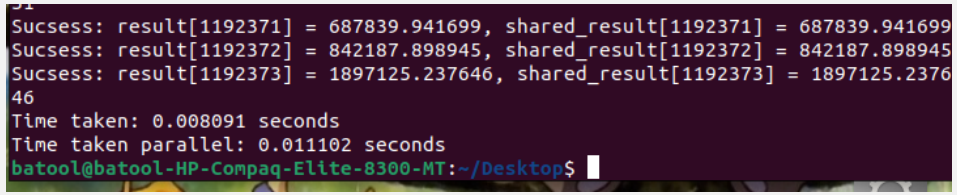
Check the result

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

    if (result[i] != shared_result[i]) {

        printf("Error: result[%d]
        = %lf, shared_result[%d] = %lf\n",
         i, result[i], i, shared_result[i]);

      }

      else{

          printf("Sucsess: result[%d]
          = %lf, shared_result[%d] = %lf\n",
          i, result[i], i, shared_result[i]);

      }
  }

  printf("Time taken: %f seconds\n", time_taken);

  printf("Time taken parallel: %f seconds\n",
  time_taken_shared);
```

OUTPUT

These results are displayed by running the code on a machine running Ubuntu and another using a virtual machine.

Ubuntu computer



virtual machine

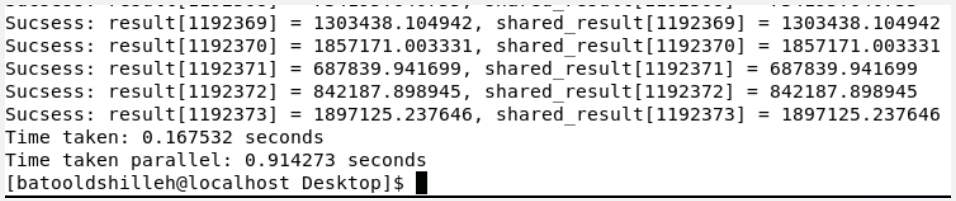
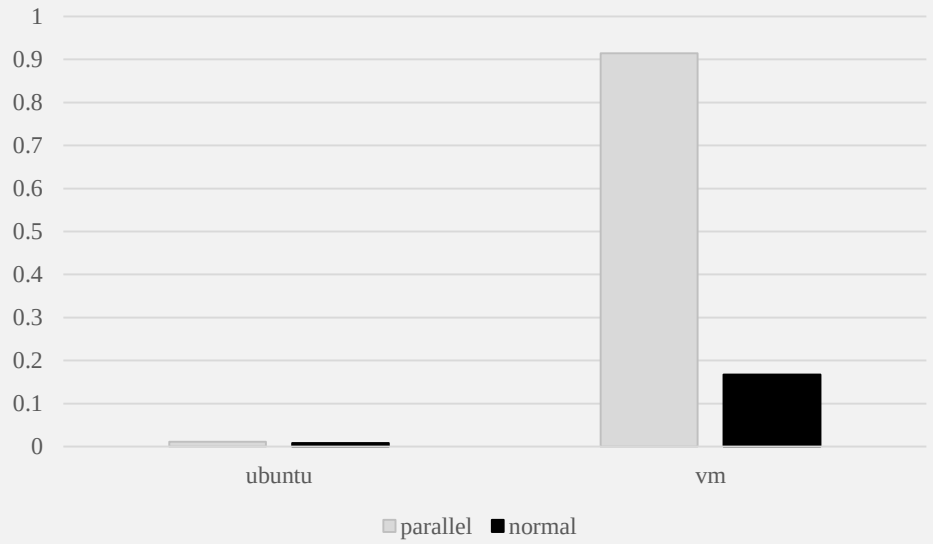


CHART OPTIONS

The graph shows the speed difference between the machine running Ubuntu and the machine running the virtual machine ipsum dolor sit amet, consectetur adipiscing elit.



	parallel	normal
ubuntu	0.0111	0.00809
vm	0.91427	0.16753