CS3510: OS-1 - HW ASSIGNMENT 3 - REPORT

PLAGIARISM STATEMENT

We certify that this assignment/report is our own work, based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this assignment/report has not previously been submitted for assessment in any other course, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that we have not copied in part or whole or otherwise plagiarised the work of other students and/or persons. We pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, we understand my responsibility to report honour violations by other students if we become aware of it.
Name: Ananya Mantravadi & Siddharth Saini
Date: 1-12-2020
Signature: AM & SS
Roll Numbers: CS19B1004 & CS19B1024

MULTIPLICATION FEASIBILITY:

```
if(acols!=brows) {
        fprintf(stderr, "Matrix multiplication not feasible\n");
        exit(EXIT_FAILURE);
    }
```

Since matrix multiplication is not feasible if the number of columns of the first matrix and the number of rows of the second matrix are equal, this code snippet was added to the main function. If this condition turns out to be true, the program will exit from main() by returning EXIT_FAILURE to the shell.

MATRIX INITIALIZATION:

```
void init_matrix(int *mat, int rows, int cols)
{
    for (int i=0; i<rows; i++) {
        for (int j=0; j<cols; j++) {
            *(mat+(i*cols+j)) = rand() % 100;
        }
    }
}
```

This function is called within single_thread_mm() and multi_thread_mm() to initialize matrices A and B and insert elements into them (ranging from 0 to 100) randomly using rand().

MEASURING CPU TIME ELAPSED:

```
clock_t begin = clock();                    //note the time when computation begins
/* computational code */
clock_t end = clock();                      //note the time when computation ends
clock_t clockTimeSpent = end - begin;
unsigned long long time_spent = 1000000*((double)clockTimeSpent/(unsigned long
long)CLOCKS_PER_SEC);
```

The time taken for multiplication has been calculated using **clock()** function that is available in time.h header file. The difference between **begin** and **end** divided by CLOCKS_PER_SEC (also determined in time.h) gives the number of microseconds used by the CPU.

MULTI-THREADED MULTIPLICATION:

THREAD PORTION:

```
for(i=0; i<bcols; i++) {
      for(j=0; j<acols; j++) {
        *(C+k*bcols+i) += *(A+k*acols+j) * (*(B+j*bcols+i));
      }
}
```

MULTI_THREAD_MM:

```
pthread_exit(NULL);
pthread_t* threads;
threads = (pthread_t*)malloc(arows*sizeof(pthread_t));
int* dataForThread[arows];
for(i=0; i<arows; i++) {
      dataForThread[i] = (int*)malloc(sizeof(int));
      *dataForThread[i] = i;
      pthread_create(&threads[i], NULL, threadPortion, (void*)(dataForThread[i]));
}

for(i=0; i<arows; i++) {
      pthread_join(threads[i], NULL);
}
```

After initializing matrices A and B, we create threads equal to the number of rows in matrix A. An array is initialized to store the data to be transferred to each thread every time for computation. Then, threads are created and the data for computation of one row at a time is passed to them (threadPortion). Each row in the resultant matrix is computed using these threads. Finally, the time taken for multiplication is returned by the function. At the end of the program execution, main() returns EXIT_SUCCESS to the shell.

OBSERVATIONS ON SPEEDUPS:
(Without using Multi_Process as of now.)

```
> ./main --ar 3 --ac 4 --br 4 --bc 5
Time taken for single threaded: 3 us
Time taken for multi process: 1 us
Time taken for multi threaded: 552 us
Speedup for multi process : 3.00 x
Speedup for multi threaded : 0.01 x
```

Here, a multi-threaded process takes more time than a single-threaded process. When actual computation is less due to smaller inputs, there is a time overhead in the multi-threaded process. This is because the multi-threaded process also includes context-switching, `pthread_join`, and `pthread_exit` calls.

```
> ./main --ar 30 --ac 40 --br 40 --bc 50
Time taken for single threaded: 523 us
Time taken for multi process: 1 us
Time taken for multi threaded: 595 us
Speedup for multi process : 523.00 x
Speedup for multi threaded : 0.88 x
```

Here, we see a slight improvement with the speedup for the multi-threaded process being 0.88 times as compared to 0.01 times in the previous run.

```
> ./main --ar 50 --ac 50 --br 50 --bc 50
Time taken for single threaded: 1220 us
Time taken for multi process: 1 us
Time taken for multi threaded: 1029 us
Speedup for multi process : 1220.00 x
Speedup for multi threaded : 1.19 x
```

For the first time, the speedup for a multi-threaded process is greater than one, with the multi-threaded process being 1.19 times faster than a single-threaded process.

```
> ./main --ar 400 --ac 500 --br 500 --bc 540
Time taken for single threaded: 1479139 us
Time taken for multi process: 1 us
Time taken for multi threaded: 201890 us
Speedup for multi process : 1479139.00 x
Speedup for multi threaded : 7.33 x
```

As the input size increases, the speedup for the multi-threaded process also increases as we can see observe.

```
> ./main --ar 800 --ac 650 --br 650 --bc 600
Time taken for single threaded: 5028017 us
Time taken for multi process: 1 us
Time taken for multi threaded: 474376 us
Speedup for multi process : 5028017.00 x
Speedup for multi threaded : 10.60 x
```

MULTI-PROCESS MULTIPLICATION:

```
int segmentID = shmget(IPC_PRIVATE,
sizeof(int)*((arows*acols)+(brows*bcols)+(arows*bcols)), IPC_CREAT|0666);

int* partition = (int*)shmat(segmentID, NULL, 0);
```

We first create a shared memory segment and allocate it a space equal to the sum of the sizes of Matrix A, Matrix B, and Matrix C. Then we pass the shared memory id to the partition to compute a partition of Matrix C at a time.

```
for(i=0; i<noOfProcesses; i++)
  {
    if(pid != 0)
    {
      pid = fork();
    }
  }
```

we fork() the process equal to the number of times of the rows.

```
if(pid == 0)
  {
    startRow = (*partition * arows)/noOfProcesses;
    endRow = ((*partition+1)*arows)/noOfProcesses;

    for(i = startRow; i<endRow; i++ )
    {
      for(j=0; j<bcols; j++)
      {
        for(k=0; k<acols; k++)
        {
          *(C + (bcols)*i + j) += *(A + (acols)*i + k) * (*(B + (bcols)*k + j));
        }
```

```
        }
      }
    }
```

This child process (when pid is 0) computes each row at a time in every child process. We initialize the row from where we need to start computation and the row where we need to end.

```
else
  {
    /* waits for all the child processes to execute first. */
    wait(NULL);
}
```

We wait in the parent process so that all child processes are executed first.

```
> gcc main.c -pthread -o main
> ./main --ar 3 --ac 4 --br 4 --bc 5
Time taken for single threaded: 3 us
Time taken for multi process: 9 us
Time taken for multi threaded: 359 us
Speedup for multi process : 0.33 x
Speedup for multi threaded : 0.01 x
> ./main --ar 30 --ac 40 --br 40 --bc 50
Time taken for single threaded: 531 us
Time taken for multi process: 30 us
Time taken for multi threaded: 1396 us
Speedup for multi process : 17.70 x
Speedup for multi threaded : 0.38 x
> ./main --ar 300 --ac 400 --br 400 --bc 500
Time taken for single threaded: 629258 us
Time taken for multi process: 2258 us
Time taken for multi threaded: 201049 us
Speedup for multi process : 278.68 x
Speedup for multi threaded : 3.13 x
> ./main --ar 800 --ac 600 --br 600 --bc 500
Time taken for single threaded: 2766812 us
Time taken for multi process: 3677 us
Time taken for multi threaded: 563092 us
Speedup for multi process : 752.46 x
Speedup for multi threaded : 4.91 x
```

These are the outputs obtained after adding multi_process_mm in our program. We observe that multiprocess takes lesser time than multi-threaded and single-threaded processes as it optimizes computational work as opposed to multi-threaded processes which optimize I/O work.