Speeding Up matrix multiplication using multithreading multiprocessing

PLAGIARISM STATEMENT

Name: Kushagra Indurkhya and Aditya Agrawal

Roll No: CS19B1017 and CS19B1003

Date: 22/11/2020

Signature: [illegible] | CS19B1003 (Aditya)

Single Threaded matrix multiplication

```
1. void row_multiplier(int i)
2. {
3.  for(int j=0; j < ccols; j++){
4.        C[i*ccols+j] = 0;
5.        for(int k=0; k < brows; k++){
6.            C[i*ccols+j] += A[i*acols+k]*B[k*bcols+j];
7.        }
8.    }
9. }
```

```
1. unsigned long long single_thread_mm()
2. {
3.    if(interactive){
4.        input_matrix(A, arows, acols);
5.        input_matrix(B, brows, bcols);
6.    } else {init_matrix();}
7.    time_t start = clock(), end;
8.    for(int i=0;i<crows;i++)
9.        row_multiplier(i);
10.
11.     end = clock();
12.     if(interactive) output_matrix(C, crows, ccols);
13.     return end-start;
14.  }
```

In *single_thread_mm* function we evaluate all the rows of matrix C in a single (main) thread (line-8) by using a helper function (*row_multiplier*- which takes row number of C to be evaluated as argument) looping over crows (ie no of rows of C)

Multithreaded matrix multiplication

```c
void *multi_thread_helper(void *arg)
{
    long t_arg = (long)arg;
    row_multiplier((int)t_arg);
    pthread_exit(NULL);
}
```

```c
unsigned long long multi_thread_mm(){
    if(interactive){
        input_matrix(A, arows, acols);
        input_matrix(B, brows, bcols);
    } else {init_matrix();}
    pthread_t threads[crows];
    int rc;
    long t;
    for(t=0; t<crows; t++){
        pthread_create(&threads[t], NULL, multi_thread_helper, (void*)t);
    }
    time_t start = clock(),end;
    for(int i=0; i<(crows); i++) pthread_join(threads[i], NULL);
    end = clock();
    if(interactive) output_matrix(C, crows, ccols);
    return end-start;
}
```

To evaluate the multiplication result using multiple threads, we made ***crows*** number of threads to evaluate the values in a row.
Making less threads (by not making for each cell) helps in reducing thread creation overhead. We create threads using **pthread library/API** which passes the corresponding row number to *multi_thread_helper*.
The *multi_thread_helper* in turn calls the *row_multiplier* function to evaluate the values.
In this manner the matrix C is evaluated. While this is being computed, the main thread waits for all worker threads to complete their job and exit (This is achieved using pthread_join).

Multi-process matrix multiplication

```c
void mult_process_helper(int i)
{
    row_multiplier(i);
}
```

```c
unsigned long long multi_process_mm()
{
    if(interactive){
        input_matrix(A, arows, acols);
        input_matrix(B, brows, bcols);
    } else {init_matrix();}

    time_t start=clock(),end;
    int status = 0;
    wait(NULL);
    pid_t child_pid, wpid;
    for (int id=0; id<crows; id++) {
        if ((child_pid = fork()) == 0) {
            mult_process_helper(id);
            exit(0);
        }
    }
    while ((wpid = wait(&status)) > 0);
    end = clock();
    if(interactive) output_matrix(C, crows, ccols);
    return end-start;
}
```

To evaluate the multiplication result using multiple processes, we made ***crows*** number of processes to evaluate the values in a row.
Making less processes (not making for each cell) helps in lower memory use and reduces process overhead. We created processes using *fork()* function (Processes are created only in parent process) which passes the corresponding row number to *mult_process_helper.*
The *multi_process_helper* in turn calls the *row_multiplier* function to evaluate the values.
In this manner the matrix C is evaluated. While this is being computed, the *multi_process_mm* function waits for all processes to complete their job and exit (This is achieved using *wait* in line 18).

Testing Using different parameters

Case-1	Case-2
./matmul --ar 4000 --ac 40 --br 40 --bc 3000 *Time taken for single threaded: 5763 us* *Time taken for multi process: 2404 us* *Time taken for multi threaded: 147 us* *Speedup for multi process : 2.40 x* *Speedup for multi threaded : 39.20 x*	*./matmul --ar 30 --ac 300000 --br 300000 --bc 30* *Time taken for single threaded: 4176 us* *Time taken for multi process: 53 us* *Time taken for multi threaded: 15329 us* *Speedup for multi process : 78.79 x* *Speedup for multi threaded : 0.27 x*
./matmul --ar 4000 --ac 40 --br 40 --bc 3000 *Time taken for single threaded: 5763 us* *Time taken for multi process: 2404 us* *Time taken for multi threaded: 147 us* *Speedup for multi process : 2.40 x* *Speedup for multi threaded : 39.20 x*	*/matmul --ar 40 --ac 400000 --br 400000 --bc 40* *Time taken for single threaded: 11507 us* *Time taken for multi process: 277 us* *Time taken for multi threaded: 37177 us* *Speedup for multi process : 41.54 x* *Speedup for multi threaded : 0.31 x*
./matmul --ar 5000 --ac 50 --br 50 --bc 5000 *Time taken for single threaded: 14659 us* *Time taken for multi process: 2153 us* *Time taken for multi threaded: 126 us* *Speedup for multi process : 6.81 x* *Speedup for multi threaded : 116.34 x*	*./matmul --ar 50 --ac 500000 --br 500000 --bc 50* *Time taken for single threaded: 22831 us* *Time taken for multi process: 296 us* *Time taken for multi threaded: 52176 us* *Speedup for multi process : 77.13 x* *Speedup for multi threaded : 0.44 x*

From the above tests we can conclude that whenever the input consists of more computation tasks (crows > acols) the task is better handled when creating multiple processes instead of multiple threads. Although when there is potential of dividing the task more creating multiple threads handle the work better (i.e., crows < acols).