```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>

//gcc -Wall -Werror -O2 -std=gnu90 MatrixMult.c
//valgrind --leak-check=yes -q --vgdb=no ./a.out < input.txt > output.txt

int * * * input;
int * * * results;
int * * dimensions;
int count = 0;
int row0 = 0;
int AAA = 0;
#ifdef _REENTRANT
    int read_wall = -1;
    int write_wall = -1;
    pthread_mutex_t lockwrite, lockread, lockcount, lockresults, lockinput, lockdimension;
#endif

/**
 * while (read_wall < i) {
 *  this is bad idea?
 * }
 * Hint:
 * How many locks? Mutexes?
 * what are the critical sections?
 * we will work on making it actually good next assigment.
 **/

void * read_matrices() {
    int r;
    int c;
    int i, j;
    //printf("Read time bois \n");
    AAA = scanf("%d %d", &r, &c);
    #ifdef _REENTRANT
        pthread_mutex_lock(&lockcount);
        pthread_mutex_lock(&lockdimension);
    #endif
    dimensions[0] = malloc(sizeof(int) * 101);
    dimensions[1] = malloc(sizeof(int) * 101);

    //garuntee intialized
    dimensions[1][0] = 0;
    dimensions[0][1] = 0;

    count = -1;
```

Commented [TD(1]: This doesn't make sense to me. Maybe it's right, but it doesn't feel like you are initializing the right variables.

Commented [TD(2]: This feels weird too. At the very least the name of the variable is wrong.

```
    row0 = r;
    #ifdef _REENTRANT
        pthread_mutex_unlock(&lockcount);
        pthread_mutex_unlock(&lockdimension);
    #endif
    while (r != 0 && c != 0) {
        #ifdef _REENTRANT
            pthread_mutex_lock(&lockread);
            pthread_mutex_lock(&lockinput);
            pthread_mutex_lock(&lockcount);
            pthread_mutex_lock(&lockdimension);
        #endif
        count++;
        dimensions[0][count] = r;
        dimensions[1][count] = c;
        input[count] = malloc(sizeof(int *) * r);
        for (i = 0; i < r; ++i) {
            input[count][i] = malloc(sizeof(int) * c);
            for (j = 0; j < c; ++j) {
                AAA = scanf("%d", &input[count][i][j]);
            }
        }
        #ifdef _REENTRANT
            read_wall += 1;
            pthread_mutex_unlock(&lockread);
            pthread_mutex_unlock(&lockinput);
            pthread_mutex_unlock(&lockcount);
            pthread_mutex_unlock(&lockdimension);
        #endif
        AAA = scanf("%d %d", &r, &c);
    }
    #ifdef _REENTRANT
        pthread_mutex_lock(&lockread);
        read_wall += 1;
        pthread_mutex_unlock(&lockread);
    #endif
    return NULL;
}

void * mult_matrices() {
    sleep(1);
    //printf("I do Start CLAC \n");
    int i;
    int j;
    int k;
    int z = 0;
    #ifdef _REENTRANT
        pthread_mutex_lock(&lockwrite);
        pthread_mutex_lock(&lockread);
        pthread_mutex_lock(&lockresults);
```

Commented [TD(3]: Using lots of locks can lead to deadlock. You should be able to solve this problem with only 2-3 locks.

Also we only want to lock for a tiny window of time. In this case the many calls to malloc() take a LOT of time. Also scanf() uses zillions of instructions.

Basically, figure out how to agree among the threads which data items are “mine” and which are “yours”, instead of “owning them all”.

Commented [TD(4]: This is a reasonable critical section size.

```
        pthread_mutex_lock(&lockinput);
        pthread_mutex_lock(&lockcount);
        pthread_mutex_lock(&lockdimension);
        while(read_wall<0);
    #endif
    results[0] = malloc(sizeof(int *) * row0);
    //printf("I start pre loop \n");
    for (i = 0; i < row0; i++) {
        results[0][i] = malloc(sizeof(int) * dimensions[1][0]);
        for (j = 0; j < dimensions[1][0]; j++) {
            results[0][i][j] = input[0][i][j];
        }
    }
    #ifdef _REENTRANT
            pthread_mutex_unlock(&lockwrite);
            pthread_mutex_unlock(&lockread);
            pthread_mutex_unlock(&lockresults);
            pthread_mutex_unlock(&lockinput);
            pthread_mutex_unlock(&lockcount);
            pthread_mutex_unlock(&lockdimension);
        #endif
    //printf("I done Pre loop \n");
    for (z = 1; z <= count; z++) {
        #ifdef _REENTRANT
        pthread_mutex_lock(&lockwrite);
        pthread_mutex_lock(&lockread);
        pthread_mutex_lock(&lockresults);
        pthread_mutex_lock(&lockinput);
        pthread_mutex_lock(&lockcount);
        pthread_mutex_lock(&lockdimension);
        while(read_wall<z);
    #endif
        results[z] = malloc(sizeof(int *) * row0);
        //printf("I on count %d \n", z);
        for (i = 0; i < row0; i++) {
            results[z][i] = malloc(sizeof(int) * dimensions[1][z]);
            //printf("I on row %d \n", i);
            for (j = 0; j < dimensions[1][z]; j++) {
                //printf("I on init time %d \n", z);
                results[z][i][j] = 0;
                //printf("I on math TIME %d \n", j);
                for (k = 0; k < dimensions[0][z]; k++) {
                    results[z][i][j] += results[z-1][i][k]*input[z][k][j];
                }
                //printf("I on sort TIME %d \n", j);
                int l = j;
                int temp;
                while(l > 0 && results[z][i][l] < results[z][i][l-1]) {
                    temp = results[z][i][l-1];
                    results[z][i][l-1] = results[z][i][l];
```

Commented [TD(5]: Locking all these before you can actually do something (see spin-wait below) means other threads can't get anything done.

Commented [TD(6]: This is a spin-wait—a horrible waste of CPU cycles. Now you know to use a condition variable or a semaphore.

Also, this only "helps" the first time thru?

Commented [TD(7]: Why do you need to copy the whole matrix?

Oh this is the first one only. Still, no need to copy it.

Commented [TD(8]: See comments from above.

```
                    results[z][i][l] = temp;
                    l--;
                }
            }
        }
        #ifdef _REENTRANT
            write_wall += 1;
            pthread_mutex_unlock(&lockwrite);
            pthread_mutex_unlock(&lockread);
            pthread_mutex_unlock(&lockresults);
            pthread_mutex_unlock(&lockinput);
            pthread_mutex_unlock(&lockcount);
            pthread_mutex_unlock(&lockdimension);
        #endif
    }
    #ifdef _REENTRANT
        pthread_mutex_unlock(&lockwrite);
        write_wall += 1;
        pthread_mutex_unlock(&lockwrite);
    #endif
    //printf("I done calc \n");
    return NULL;
}

void * show_matrices() {
    sleep(1);
    int i, j, k;


    #ifdef _REENTRANT
        pthread_mutex_lock(&lockwrite);
        pthread_mutex_lock(&lockresults);
        pthread_mutex_lock(&lockcount);
        pthread_mutex_lock(&lockdimension);
    #endif

    //Free
    for(i=0; i < row0; i++) {
        free(input[0][i]);
        free(results[0][i]);
    }
    free(input[0]);
    free(results[0]);
    #ifdef _REENTRANT
        pthread_mutex_unlock(&lockwrite);
        pthread_mutex_unlock(&lockresults);
        pthread_mutex_unlock(&lockcount);
        pthread_mutex_unlock(&lockdimension);
    #endif
    for(k=1; k <= count; k++) {
```

Commented [TD(9]: That's a lot of shuffling. There are sort algorithms that don't do as many assignment operations.

Commented [TD(10]: So here's the thing. Since we have a variable that tells us where the "dividing line" is (write_wall), we don't actually have to lock all the variables, as long as we agree to "stay on our side".

Commented [TD(11]: This seems weird and unnecessary.

Commented [TD(12]: This is not helping performance at all and probably hiding some race conditions.

Commented [TD(13]: So you copied it but didn't actually do anything with it?

```
        #ifdef _REENTRANT
            pthread_mutex_lock(&lockwrite);
            pthread_mutex_lock(&lockresults);
            pthread_mutex_lock(&lockcount);
            pthread_mutex_lock(&lockdimension);
            while(write_wall<k);
        #endif
        printf("%d %d\n", row0, dimensions[1][k]);
        for(i=0; i < row0; i++) {
            printf("%d", results[k][i][0]);
            for(j = 1; j < dimensions[1][k]; j++) {
                printf(" %d",results[k][i][j]);
            }
            printf("\n");
            free(results[k][i]);
        }
        for(i=0; i < dimensions[0][k]; i++) {
            free(input[k][i]);
        }
        free(input[k]);
        free(results[k]);
        #ifdef _REENTRANT
            pthread_mutex_unlock(&lockwrite);
            pthread_mutex_unlock(&lockresults);
            pthread_mutex_unlock(&lockcount);
            pthread_mutex_unlock(&lockdimension);
        #endif
    }
    printf("0 0\n");
    #ifdef _REENTRANT
        pthread_mutex_lock(&lockwrite);
        pthread_mutex_lock(&lockresults);
        pthread_mutex_lock(&lockcount);
        pthread_mutex_lock(&lockdimension);
    #endif
    free(input);
    free(results);
    for(i = 0; i < 2; i++) {
        free(dimensions[i]);
    }
    free(dimensions);
    #ifdef _REENTRANT
            pthread_mutex_unlock(&lockwrite);
            pthread_mutex_unlock(&lockresults);
            pthread_mutex_unlock(&lockcount);
            pthread_mutex_unlock(&lockdimension);
        #endif
    return NULL;
}
```

```
int main() {
    #ifdef _REENTRANT
        pthread_t readThred;
        pthread_t multThred;
        pthread_t showThred;
        pthread_mutex_init(&lockwrite, NULL);
        pthread_mutex_init(&lockread, NULL);
        pthread_mutex_init(&lockcount, NULL);
        pthread_mutex_init(&lockresults, NULL);
        pthread_mutex_init(&lockinput, NULL);
        pthread_mutex_init(&lockdimension, NULL);
    #endif
    u_int64_t diff;
    u_int64_t read;
    u_int64_t mult;
    u_int64_t show;
    struct timespec start;
    struct timespec startr, endr;
    struct timespec startm, endm;
    struct timespec starts, ends;
    //printf("I do Start \n");
    // initialize any global variables
    input = malloc(sizeof(int * *) * 100);
    results = malloc(sizeof(int * *) * 100);
    dimensions = malloc(sizeof(int *) * 2);

    clock_gettime(CLOCK_REALTIME, &start);

    //run
    #ifdef _REENTRANT
        clock_gettime(CLOCK_REALTIME, &startr);
        pthread_create(&readThred,NULL,read_matrices,NULL);
    #else
        clock_gettime(CLOCK_REALTIME, &startr);
        read_matrices();
        clock_gettime(CLOCK_REALTIME, &endr);
        read = 1000000000 * (endr.tv_sec - startr.tv_sec) + endr.tv_nsec - startr.tv_nsec;
    #endif

    #ifdef _REENTRANT
        clock_gettime(CLOCK_REALTIME, &startm);
        pthread_create(&multThred,NULL,mult_matrices,NULL);
    #else
        clock_gettime(CLOCK_REALTIME, &startm);
        mult_matrices();
        clock_gettime(CLOCK_REALTIME, &endm);
        mult = 1000000000 * (endm.tv_sec - startm.tv_sec) + endm.tv_nsec - startm.tv_nsec;
    #endif

    #ifdef _REENTRANT
```

Commented [TD(14]: Since the size of these is not changing, why do you need to malloc() them?

```
        clock_gettime(CLOCK_REALTIME, &starts);
        pthread_create(&showThred,NULL,show_matrices,NULL);
    #else
        clock_gettime(CLOCK_REALTIME, &starts);
        show_matrices();
        clock_gettime(CLOCK_REALTIME, &ends);
        show = 1000000000 * (ends.tv_sec - starts.tv_sec) + ends.tv_nsec - starts.tv_nsec;
    #endif

    #ifdef _REENTRANT
        pthread_join(readThred, NULL);
        clock_gettime(CLOCK_REALTIME, &endr);
        read = 1000000000 * (endr.tv_sec - startr.tv_sec) + endr.tv_nsec - startr.tv_nsec;
    #endif

    #ifdef _REENTRANT
        pthread_join(multThred, NULL);
        clock_gettime(CLOCK_REALTIME, &endm);
        mult = 1000000000 * (endm.tv_sec - startm.tv_sec) + endm.tv_nsec - startm.tv_nsec;
    #endif

    #ifdef _REENTRANT
        pthread_join(showThred, NULL);
        clock_gettime(CLOCK_REALTIME, &ends);
        show = 1000000000 * (ends.tv_sec - starts.tv_sec) + ends.tv_nsec - starts.tv_nsec;
    #endif

    diff = 1000000000 * (ends.tv_sec - start.tv_sec) + ends.tv_nsec - start.tv_nsec;
    printf("Reading: %luns\n", read);
    printf("Computing: %luns\n", mult);
    printf("Writing: %luns\n", show);
    printf("Elapsed: %luns\n", diff);

    // report time
    return 0;
}
```

Commented [TD(15]: There's a lot of duplicated code here, which means fertile ground for bugs to grow.

About half the time your code generates valgrind errors. This time was with the sample intput.

```
==3170== Thread 4:
==3170== Use of uninitialised value of size 8
==3170==    at 0x109460: show_matrices
==3170==    by 0x4E436DA: start_thread (pthread_create.c:463)
==3170==    by 0x517C88E: clone (clone.S:95)
==3170==
==3170== Invalid read of size 8
==3170==    at 0x109460: show_matrices
==3170==    by 0x4E436DA: start_thread (pthread_create.c:463)
==3170==    by 0x517C88E: clone (clone.S:95)
==3170==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==3170==
==3170==
==3170== Process terminating with default action of signal 11 (SIGSEGV)
==3170==  Access not within mapped region at address 0x0
==3170==    at 0x109460: show_matrices
==3170==    by 0x4E436DA: start_thread (pthread_create.c:463)
==3170==    by 0x517C88E: clone (clone.S:95)
==3170==  If you believe this happened as a result of a stack
==3170==  overflow in your program's main thread (unlikely but
==3170==  possible), you can try to increase the size of the
==3170==  main thread stack using the --main-stacksize= flag.
==3170==  The main thread stack size used in this run was 8388608.
==3170== Thread 1:
==3170== 272 bytes in 1 blocks are possibly lost in loss record 4 of 9
==3170==    at 0x4C31B25: calloc (in vgpreload_memcheck-amd64-linux.so)
==3170==    by 0x40134A6: allocate_dtv (dl-tls.c:286)
==3170==    by 0x40134A6: _dl_allocate_tls (dl-tls.c:530)
==3170==    by 0x4E44227: allocate_stack (allocatestack.c:627)
==3170==    by 0x4E44227: pthread_create@@GLIBC_2.2.5 (pthread_create.c:644)
==3170==    by 0x108AD4: main
==3170==
==3170== 272 bytes in 1 blocks are possibly lost in loss record 5 of 9
==3170==    at 0x4C31B25: calloc (in vgpreload_memcheck-amd64-linux.so)
==3170==    by 0x40134A6: allocate_dtv (dl-tls.c:286)
==3170==    by 0x40134A6: _dl_allocate_tls (dl-tls.c:530)
==3170==    by 0x4E44227: allocate_stack (allocatestack.c:627)
==3170==    by 0x4E44227: pthread_create@@GLIBC_2.2.5 (pthread_create.c:644)
==3170==    by 0x108AF5: main
```

This time was with a rather large input file.

```
==3292== Thread 4:
==3292== Use of uninitialised value of size 8
==3292==    at 0x109460: show_matrices
==3292==    by 0x4E436DA: start_thread (pthread_create.c:463)
==3292==    by 0x517C88E: clone (clone.S:95)
==3292==
==3292== Invalid read of size 8
==3292==    at 0x109460: show_matrices
==3292==    by 0x4E436DA: start_thread (pthread_create.c:463)
==3292==    by 0x517C88E: clone (clone.S:95)
==3292==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

```
==3292==
==3292== Process terminating with default action of signal 11 (SIGSEGV)
==3292==  Access not within mapped region at address 0x0
==3292==    at 0x109460: show_matrices
==3292==    by 0x4E436DA: start_thread (pthread_create.c:463)
==3292==    by 0x517C88E: clone (clone.S:95)
==3292==  If you believe this happened as a result of a stack
==3292==  overflow in your program's main thread (unlikely but
==3292==  possible), you can try to increase the size of the
==3292==  main thread stack using the --main-stacksize= flag.
==3292==  The main thread stack size used in this run was 8388608.
==3292== Thread 1:
==3292== 272 bytes in 1 blocks are possibly lost in loss record 2 of 9
==3292==    at 0x4C31B25: calloc (in vgpreload_memcheck-amd64-linux.so)
==3292==    by 0x40134A6: allocate_dtv (dl-tls.c:286)
==3292==    by 0x40134A6: _dl_allocate_tls (dl-tls.c:530)
==3292==    by 0x4E44227: allocate_stack (allocatestack.c:627)
==3292==    by 0x4E44227: pthread_create@@GLIBC_2.2.5 (pthread_create.c:644)
==3292==    by 0x108AD4: main
==3292==
==3292== 272 bytes in 1 blocks are possibly lost in loss record 3 of 9
==3292==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==3292==    by 0x40134A6: allocate_dtv (dl-tls.c:286)
==3292==    by 0x40134A6: _dl_allocate_tls (dl-tls.c:530)
==3292==    by 0x4E44227: allocate_stack (allocatestack.c:627)
==3292==    by 0x4E44227: pthread_create@@GLIBC_2.2.5 (pthread_create.c:644)
==3292==    by 0x108AF5: main
```

Threaded mode sometimes produces no output at all, for large files.

Unthreaded:
```
Reading:     48036900ns
Computing: 1045454700ns
Writing:   1135320500ns
Elapsed:   2228812400ns
```

Threaded:
```
Reading:     30890000ns
Computing: 1021726100ns
Writing:   1103713500ns
Elapsed:   1103792900ns
```

This seems good, because it is faster than unthreaded but there is something very wrong since compute and write are both approximately 50 slower than read in both versions.