Lab 10: Threads

Due Date: 4/8/21 (note: extended, but with double credit)

In this lab, you'll get experience in measuring the performance of a compute-intensive program. You'll start off with an unthreaded program and then move on to measure performance of a multithreaded program. The "compute-intensive" operation will be user-scalable matrix multiplication.

Templates and other necessary files are in:

```
http://www.tricity.wsu.edu/~bobl/cpts360/lab10_threads
```

(There is also a link to it in the schedule on the course web page.) Calling it up in a browser will allow you to download all of the files you need for this lab. `writeup.html` in that directory contains an HTML version of this page.

Part 2 depends on Part 1, so be sure that part is working before you move on.

Matrix Multiplication

A very basic operation in numerically intensive (aka "number crunching") programs is matrix multiplication. You should recall the basic formula from Math 220 (Linear Algebra): The product of an *n* by *p* matrix **a** and a *p* by *m* matrix **b** is an *n* by *m* matrix **c**, where:

$$c_{ij} = \sum_{k=0}^{p-1} a_{ik} b_{kj}$$

for all *i* from 0 to *n-1*, inclusive and all *j* from 0 to *m-1*, inclusive.

(The code for doing this is already provided.)

Part 1: Timing an Unthreaded Program

The code for matrix multiplication is maintained in several files:

1. Download the following files: `tspec_diff.h`, `tspec_diff.c`, and `a2d.h`. These will be used unchanged throughout this lab.
2. Download the header file `mat_mul_pt1_tplt.h` and rename it to `mat_mul_pt1.h`. Here are its contents:

   ```
   void mat_mul(double *_c, const int n, const int m,
                const double *_a, const int p, const double *_b);
   ```

   This is the header file for matrix multiplication. You do not need to modify this file for Part 1.
3. Download the source file `mat_mul_pt1_tplt.c` and rename it to `mat_mul_pt1.c`. Here are its contents:

   ```
   #include "a2d.h"
   #include "tspec_diff.h"
   ```

```
#include "mat_mul_pt1.h"


static void multiplyRow(double *_c, const int i, const int n, const int m,
                        const double *_a, const int p, const double *_b)
{
#define c(i,j) _c[I2D(i, n, j, m)]
#define a(i,j) _a[I2D(i, n, j, p)]
#define b(i,j) _b[I2D(i, p, j, m)]
    int j, k;
    double sum;

    sum = 0.0;
    for (j = 0; j < m; j++) {
        sum = 0.0;
        for (k = 0; k < p; k++)
            sum += a(i,k) * b(k,j);
        c(i,j) = sum;
    }
#undef a
#undef b
#undef c
}


void mat_mul(double *_c, const int n, const int m,
             const double *_a, const int p, const double *_b)
{
    int i;

    for (i = 0; i < n; i++)
        multiplyRow(_c, i, n, m, _a, p, _b);
}
```

This file implements matrix multiplication. You do not need to modify it for Part 1 either, but notice how we have put the code to multiply a single row in a separate function `multiplyRow()`. This will be useful when we make this program multithreaded.

4. Download the source file that contains `main()`, `experiment_tmm_pt1_tplt.c`, and rename it to `experiment_tmm_pt1.c`. Here are its contents:

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <sys/time.h>
#include <time.h> // for `clock_gettime()`

#include "a2d.h"
```

```
#include "eprintf.h"
#include "tspec_diff.h"
#include "mat_mul_pt1.h"

char *progname = "*** error: 'progname' not set ***";


// forward reference
static double dot(double *a, double *b, int n);


double *da2d_new(int n, int m)
// instances an n x m array of doubles
{
    double *result = malloc(n * m * sizeof(double));
    assert(result != NULL);
    return result;
}


void da2d_delete(double *d)
// frees an array
{
    free(d);
}


void da2d_printf(double *_d, int n, int m, char fmt[])
// prints a 2D array using a given format `fmt` for each element
{
#define d(i,j) _d[I2D(i, n, j, m)]
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf(fmt, d(i,j));
        }
        printf("\n");
    }
#undef d
}


void da2d_orthonormalize(double *_a, int n, int m)
{
/* makes an n x m matrix `_a` orthonormal via Gram-Schmidt (in-place) */
#define a(i,j) _a[I2D(i, n, j, m)]
    int i, iPrev, j;
    double dotProduct, mag;
    for (i = 0; i < n; i++)
    {
        // For all previous rows...
        for (iPrev = 0; iPrev < i; iPrev++)
        {
            // ...compute the dot product of row i with row iPrev...
            dotProduct = dot(&a(iPrev,0), &a(i,0), m);
            // ...then subtract row iPrev scaled by the dot product.
            for (j = 0; j < m; j++)
            {
                a(i,j) -= dotProduct * a(iPrev,j);
            }
        }
        // Normalize the modified row i.
        mag = sqrt(dot(&a(i,0), &a(i,0), m));
        for (j = 0; j < m; j++)
        {
            a(i,j) /= mag;
        }
    }
#undef a
}


void da2d_fillRandom(double *_a, int n, int m)
/* fills a 2D array with random numbers */
{
#define a(i,j) _a[I2D(i, n, j, m)]
    int i, j;

    // set a to a random matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            a(i,j) = ((double) random()) / RAND_MAX;
        }
    }
#undef a
}


double *da2d_transpose(double *_a, int n, int m)
{
#define a(i,j) _a[I2D(i, n, j, m)]
#define b(i,j) _b[I2D(i, m, j, n)]
    double *_b = da2d_new(m, n);
    int i, j;

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
        {
            b(j,i) = a(i,j);
        }
    }
    return _b;
#undef b
#undef a
}


static double dot(double *a, double *b, int n)
// compute the dot product of two 1-d double arrays of length `n`
{
    double result = 0;
    int i;

    for (i = 0; i < n; i++)
        result += a[i] * b[i];
    return result;
}


/* usage: issue a usage error message */
static void usage(void)
{
    eprintf("usage: %s [{args}]\n", progname);
    eprintf("%s",
            "computes the matrix product c = a x b for two random matrices\n"
            "a[] and b[] of user-specified size, optionally with threads,\n"
            "and prints the times involved\n"
            " {args} are:\n");
    eprintf("%s\n",
            "  -h      this help message");
    eprintf("%s\n",
            "  -n {i}  number of rows of c[] and a[] (default: 4)");
    eprintf("%s\n",
            "  -m {i}  number of columns of c[] and b[] (default: 4)");
    eprintf("%s\n",
            "  -o      test algorithm by forcing a[] to be orthonormal and b[]\n"
            "          to be its transpose (implies '-v' and n == m == p; default: don't)");
    eprintf("%s\n",
            "  -p {i}  number of columns of a[] and rows of b[] (default: 4)");
    eprintf("%s\n",
            "  -s {i}  seed value for srandom (default: no seeding)");
    eprintf("%s",
            "  -v      verbose: print out a and b as well as their product\n");
    return;
```

```
}


int main(int argc, char *argv[])
{
    int n = 4;
    int p = 4;
    int m = 4;
    int verbose = 0;
    int orthonormal = 0;
    int ch, ok;
    double *a, *b, *c;
    int colonIndent;

    ok = 1;
    progname = argv[0];
    while ((ch = getopt(argc, argv, "hm:n:op:qs:v")) != -1) {
        switch (ch) {

        case 'h':
            usage();
            exit(EXIT_SUCCESS);

        case 'n':
            n = atoi(optarg);
            break;

        case 'o':
            verbose = 1;
            orthonormal = 1;
            break;

        case 'm':
            m = atoi(optarg);
            break;

        case 'p':
            p = atoi(optarg);
            break;

        case 's':
            srandom(atoi(optarg));
            break;

        case 'v':
            verbose = 1;
            break;

        default:
            ok = 0;
            break;
        }
    }
    if (!ok || n <= 0 || p <= 0  || m <= 0) {
        usage();
        exit(EXIT_FAILURE);
    }

    /*
     * This value is set empirically so that all the ':'s line up
     * (like movie credits) for readability.
     */
    colonIndent = 30;

    if (orthonormal) {
        // necessary: override any command line specification of m or p
        m = p = n;
    }
    a = da2d_new(n, p);
    da2d_fillRandom(a, n, p);
    if (orthonormal) {
        da2d_orthonormalize(a, n, p);
    }
    if (verbose) {
        printf("%*s:\n", colonIndent, "a");
        da2d_printf(a, n, p, "%8.3f ");
        printf("\n");
    }

    if (orthonormal) {
        b = da2d_transpose(a, n, p);
    } else {
        b = da2d_new(p, m);
        da2d_fillRandom(b, p, m);
    }
    if (verbose) {
        printf("%*s:\n", colonIndent, "b");
        da2d_printf(b, p, m, "%8.3f ");
        printf("\n");
    }

    c = da2d_new(n, m);

    /*
     * ASSIGNMENT
     *
     * Insert clock_gettime(3) calls to get the start times for both
     * the elapsed (`CLOCK_REALTIME`) and CPU
     * (`CLOCK_PROCESS_CPUTIME_ID`) clocks.
     */


    /*
     * ASSIGNMENT
     *
     * Insert call to `mat_mul()` here.
     */

    /*
     * ASSIGNMENT
     *
     * Insert clock_gettime(3) calls to get the stop times for both
     * the elapsed (`CLOCK_REALTIME`) and CPU
     * (`CLOCK_PROCESS_CPUTIME_ID`) clocks.
     */

    if (verbose) {
        printf("%*s:\n", colonIndent, "c (= a x b)");
        da2d_printf(c, n, m, "%8.3f ");
        printf("\n");
    }


    /*
     * ASSIGNMENT
     *
     * print out the cpu time (hint: `tspecDiff()`)
     * print out the wall clock time (hint: `tspecDiff()`)
     */

    da2d_delete(a);
    da2d_delete(b);
    da2d_delete(c);

    return 0;
}
```

Make the changes indicated in the pseudocode, as indicated by the "`ASSIGNMENT`" comments.

5. Create a `Makefile` for `experiment_tmm_pt1`. You will need to compile `tspec_diff.c` into `tspec_diff.o` and include it on the load line. You'll also need to include the "`-lrt`" and "`-lm`" library flags at the end of the load line.
6. Normally, the program generates and uses completely random matrices, but if you run it with a `-o` option, the two matrices will be a random "orthogonal" matrix and its transpose. When two such matrices are multiplied (you may recall), the result is an identity matrix. If an identity matrix is printed out, it will help you verify that your matrix multiplcation is working. Make sure it does before proceeding.

7. Run some tests to make sure that your timing values are accurate. Choose command line arguments for `-n`, `-m`, and `-p` that give you run times of at least two seconds and check them against your computer's clock and/or `bash`'s `time` builtin. How does doubling any of these parameters affect the CPU time?
8. Run the same test several times and note the elapsed (aka "wall clock") and CPU times. Are they always the same? If not, how much do they vary, does one vary more than the other, and why do you think they vary? If you were reporting this to someone else, what would be the best way to describe your result and its variation (if any)?
9. Be sure to include `mat_mul_pt1.h`, `mat_mul_pt1.c`, and your `experiment_tmm_pt1.c` in the submission tarball (see below).

Part 2: Timing a Multithreaded Program

1. Download the header file `mat_mul_pt2_tplt.h` and rename it to `mat_mul_pt2.h`. Here are its contents:

```
/*
 * Statistics for thread computation are kept in an array of these,
 * one per thread.
 */
typedef struct {
    double cpuTime;
    int nRowsDone;
} MatMulThreadStats;

void mat_mul(double *_c, const int n, const int m,
             const double *_a, const int p, const double *_b,
             const int nThreads, MatMulThreadStats *matMulThreadStats);
```

   You do not need to modify this file for Part 2, but note that `mat_mul()` now has an additional (input) argument to specify the number of threads and an another (output) argument to return an array of timing statistics for each thread, using a new `typedef`, `MatMulThreadStats`, that the header file also defines.

2. Download the source file `mat_mul_pt2_tplt.c` and rename it to `mat_mul_pt2.c`. Here are its contents:

```
#include <stdlib.h> // for malloc()
#include <time.h>
#include <assert.h>
#include <pthread.h>

#include "a2d.h"
#include "tspec_diff.h"
#include "mat_mul_pt2.h"

/*
 *  Good practice: This structure contains all of the "globals" each
 *  thread can access.
 *
 *  A pointer to a shared instance of this one struct is shared by all
 *  threads used for a single matrix multiply, so any values that
 *  might be modified by more than one thread should not be changed by
 *  any thread without being protected by some kind of mutex, a la
 *  "nextRow".
 */
typedef struct {
    /*
     * `nextRow` is modified by every thread, so it needs a mutex.
     */
    int nextRow; /* the next row to be done *and* # of rows already done */
    pthread_mutex_t nextRowMutex; /* restricts access to `nextRow` */

    /*
     * As far as a thread is concerned, these are constants, so no
     * mutex is necessary.
     */
    int nThreads;
    int n, m, p; /* matrix dimensions */
    const double *_a, *_b; /* input (1D) matrices */

    /*
     * Each row of this matrix is modified by only one thread, so
     * again no mutex is necessary. (Note how this decision, which is
     * critical to good threaded performance, requires knowledge of
     * the algorithm.)
     */
    double *_c; /* output (1D) matrix */
} ThreadGlobals;


static void multiplyRow(double *_c, const int i, const int n, const int m,
                        const double *_a, const int p, const double *_b)
{
#define c(i,j) _c[I2D(i, n, j, m)]
#define a(i,j) _a[I2D(i, n, j, p)]
#define b(i,j) _b[I2D(i, p, j, m)]
    int j, k;
    double sum;

    sum = 0.0;
    for (j = 0; j < m; j++) {
        sum = 0.0;
        for (k = 0; k < p; k++)
            sum += a(i,k) * b(k,j);
        c(i,j) = sum;
    }
#undef a
```

```
#undef b
#undef c
}


// inThread -- function for each thread
static void *inThread(void *threadGlobals_)
{
    /*
     * ASSIGNMENT
     *
     * Implement the following pseudocode:
     *
     * allocate a MatMulThreadStats instance `matMulThreadStats`
     *  (hint: malloc())
     * set `matMulThreadStats->nRowsDone` to 0
     * get this thread's cpu clock id (hint: pthread_self() and
     *  pthread_getcpuclockid())
     * get the thread clock id's start cpu time (hint: clock_gettime())
     * loop the following indefinitely,
     *     if there are multiple threads,
     *         lock the "nextRow" mutex (hint: pthread_mutex_lock())
     *     set `i` to the current value of `threadGlobals->nextRow`
     *     increment `threadGlobals->nextRow`
     *     if there are multiple threads,
     *         unlock the "nextRow" thread mutex (hint: pthread_mutex_unlock())
     *     if `i` >= `n` (the number of rows),
     *         break out of the loop
     *     multiply row `i` (hint: multiplyRow())
     *     increment `matMulThreadStats->nRowsDone`
     * get the thread clock id's end cpu time (hint: clock_gettime())
     * set `matMulThreadStats->cpuTime` to the difference of thread start and
     *  stop cpu times (hint: tspecDiff())
     * return `matMulThreadStats`
     */
}


void mat_mul(double *_c, const int n, const int m,
             const double *_a, const int p, const double *_b,
             const int nThreads, MatMulThreadStats allMatMulThreadStats[])
{
    /*
     * ASSIGNMENT
     *
     * Implement the following pseudocode:
     *
     * declare a ThreadGlobals struct and set all fields to the
     *  corresponding parameters and `nextRow` to 0. (No rows have
     *  been done yet.)
     * if `nThreads` > 0,
     *     malloc() an array of `nThreads` pthread_t's
     *     initialize the `nextRowMutex` thread mutex (hint:
     *      pthread_mutex_init())
     *     for each of `nThreads` threads,
     *         create a thread calling inThread() and pass it a
     *          pointer to `threadGlobals` (hint: pthread_create())
     *     for each thread `i` of `nThreads` threads,
     *         wait for the thread to complete (hint: pthread_join())
     *         copy its MatMulThreadStats return to `allMatMulThreadStats[i]`
     *         free the returned struct
     *     destroy the thread mutex (hint: pthread_mutex_destroy())
     *     free the pthread_t array
     * otherwise
     *     call the thread function directly (hint: inThread())
     *     copy its MatMulThreadStats return to `allMatMulThreadStats[0]`
     *     free the returned struct
     */
}
```

Make the changes indicated in the pseudocode to add support for timed multithreaded matrix multiplication.

3. Download the source file that contains `main()`, `experiment_tmm_pt2_tplt.c`, and rename it to `experiment_tmm_pt2.c`. Here are its contents:

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <sys/time.h>
#include <time.h> // for `clock_gettime()`

#include "a2d.h"
#include "eprintf.h"
#include "tspec_diff.h"
#include "mat_mul_pt2.h"

char *progname = "*** error: 'progname' not set ***";


// forward reference
static double dot(double *a, double *b, int n);


double *da2d_new(int n, int m)
// instances an n x m array of doubles
{
    double *result = malloc(n * m * sizeof(double));
    assert(result != NULL);
    return result;
}


void da2d_delete(double *d)
// frees an array
{
    free(d);
}


void da2d_printf(double *_d, int n, int m, char fmt[])
// prints a 2D array using a given format `fmt` for each element
{
#define d(i,j) _d[I2D(i, n, j, m)]
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            printf(fmt, d(i,j));
        }
        printf("\n");
    }
#undef d
}


void da2d_orthonormalize(double *_a, int n, int m)
{
/* makes an n x m matrix `_a` orthonormal via Gram-Schmidt (in-place) */
#define a(i,j) _a[I2D(i, n, j, m)]
    int i, iPrev, j;
    double dotProduct, mag;

    for (i = 0; i < n; i++)
    {
        // For all previous rows...
        for (iPrev = 0; iPrev < i; iPrev++)
        {
            // ...compute the dot product of row i with row iPrev...
            dotProduct = dot(&a(iPrev,0), &a(i,0), m);
            // ...then subtract row iPrev scaled by the dot product.
            for (j = 0; j < m; j++)
            {
                a(i,j) -= dotProduct * a(iPrev,j);
            }
        }
        // Normalize the modified row i.
        mag = sqrt(dot(&a(i,0), &a(i,0), m));
        for (j = 0; j < m; j++)
        {
            a(i,j) /= mag;
        }
    }
#undef a
}


void da2d_fillRandom(double *_a, int n, int m)
/* fills a 2D array with random numbers */
{
#define a(i,j) _a[I2D(i, n, j, m)]
    int i, j;

    // set a to a random matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            a(i,j) = ((double) random()) / RAND_MAX;
        }
    }
#undef a
}


double *da2d_transpose(double *_a, int n, int m)
{
#define a(i,j) _a[I2D(i, n, j, m)]
#define b(i,j) _b[I2D(i, m, j, n)]
    double *_b = da2d_new(m, n);
    int i, j;

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
        {
            b(j,i) = a(i,j);
        }
    }
    return _b;
#undef b
#undef a
}


static double dot(double *a, double *b, int n)
// compute the dot product of two 1-d double arrays of length `n`
{
    double result = 0;
    int i;

    for (i = 0; i < n; i++)
        result += a[i] * b[i];
    return result;
}


/* usage: issue a usage error message */
static void usage(void)
{
    eprintf("usage: %s [{args}]\n", progname);
    eprintf("%s",
            "computes the matrix product c = a x b for two random matrices\n"
            "a[] and b[] of user-specified size, optionally with threads,\n"
            "and prints the times involved\n"
            " {args} are:\n");
    eprintf("%s\n",
            "  -h      this help message");
    eprintf("%s\n",
            "  -n {i}  number of rows of c[] and a[] (default: 4)");
    eprintf("%s\n",
            "  -m {i}  number of columns of c[] and b[] (default: 4)");
    eprintf("%s\n",
            "  -o      test algorithm by forcing a[] to be orthonormal and b[]\n"
            "          to be its transpose (implies '-v' and n == m == p; default: don't)");
    eprintf("%s\n",
            "  -p {i}  number of columns of a[] and rows of b[] (default: 4)");
    eprintf("%s\n",
            "  -s {i}  seed value for srandom (default: no seeding)");
    eprintf("%s\n",
            "  -t {i}  number of threads (0 means unthreaded) (default: 4)");
    eprintf("%s",
            "  -v      verbose: print out a and b as well as their product\n");
    return;
}


int main(int argc, char *argv[])
{
    int n = 4;
    int p = 4;
    int m = 4;
    int verbose = 0;
    int orthonormal = 0;
    int ch, ok;
    double *a, *b, *c;
    int colonIndent;
    int nThreads = 4;
    int i;
    MatMulThreadStats *matMulThreadStats;
    double cpuTimeTotal;

    ok = 1;
    progname = argv[0];
    while ((ch = getopt(argc, argv, "hm:n:op:qs:t:v")) != -1) {
        switch (ch) {

        case 'h':
            usage();
            exit(EXIT_SUCCESS);

        case 'n':
            n = atoi(optarg);
            break;

        case 'o':
            verbose = 1;
            orthonormal = 1;
            break;

        case 'm':
            m = atoi(optarg);
            break;

        case 'p':
            p = atoi(optarg);
            break;

        case 's':
            srandom(atoi(optarg));
            break;

        case 't':
            nThreads = atoi(optarg);
            break;

        case 'v':
            verbose = 1;
            break;

        default:
            ok = 0;
            break;
        }
    }
    if (!ok || n <= 0 || p <= 0  || m <= 0 || nThreads < 0) {
        usage();
        exit(EXIT_FAILURE);
    }

    /*
     * This value is set empirically so that all the ':'s line up
     * (like movie credits) for readability.
     */
    colonIndent = 30;

    if (orthonormal) {
        // necessary: override any command line specification of m or p
        m = p = n;
    }
    a = da2d_new(n, p);
    da2d_fillRandom(a, n, p);
    if (orthonormal) {
        da2d_orthonormalize(a, n, p);
    }
    if (verbose) {
        printf("%*s:\n", colonIndent, "a");
        da2d_printf(a, n, p, "%8.3f ");
        printf("\n");
    }

    if (orthonormal) {
        b = da2d_transpose(a, n, p);
    } else {
        b = da2d_new(p, m);
        da2d_fillRandom(b, p, m);
    }
    if (verbose) {
        printf("%*s:\n", colonIndent, "b");
        da2d_printf(b, p, m, "%8.3f ");
        printf("\n");
    }

    c = da2d_new(n, m);

    /*
     * ASSIGNMENT
     *
     * Insert clock_gettime(3) calls to get the start times for both
     * the elapsed (`CLOCK_REALTIME`) and CPU
     * (`CLOCK_PROCESS_CPUTIME_ID`) clocks.
     */

    /*
     * ASSIGNMENT
     *
     * Insert code to `malloc()` an array of max(`nThreads`, 1)
     * MatMulThreadStats.
     */

    /*
     * ASSIGNMENT
     *
     * Modify the `mat_mul()` call to pass `nThreads` and the
     * allocated `MatMulThreadStats` array.
     */

    /*
     * ASSIGNMENT
     *
     * Insert clock_gettime(3) calls to get the stop times for both
     * the elapsed (`CLOCK_REALTIME`) and CPU
     * (`CLOCK_PROCESS_CPUTIME_ID`) clocks.
     */

    if (verbose) {
        printf("%*s:\n", colonIndent, "c (= a x b)");
        da2d_printf(c, n, m, "%8.3f ");
        printf("\n");
    }

    /*
     * ASSIGNMENT
     *
     * Implement this pseudocode:
     *
     * if there are no threads,
     *     print out the cpuTime according to matMulThreadStats[0]
     * otherwise
     *     set the thread cpu time sum to 0.0
     *     for each thread i,
     *         print out the cpuTime used by thread i
     *         increment the cpu time sum by that time
     *         print out the number of rows multiplied by thread i
     *     print out the thread cpu time sum
     */

    /*
     * ASSIGNMENT
     *
     * print out the cpu time (hint: `tspecDiff()`)
     * print out the wall clock time (hint: `tspecDiff()`)
     */
    /*
     * ASSIGNMENT
     *
     * free matMulThreadStats[]
     */

    da2d_delete(a);
    da2d_delete(b);
    da2d_delete(c);

    return 0;
}
```

Again, make the changes indicated in the pseudocode to add support for timed multithreaded matrix multiplication.

4. Modify your `Makefile` to compile `experiment_tmm_pt2` as well as `experiment_tmm_pt1`. You'll need to again include `tspec_diff.o` and put both "`-lrt`" and "`-lpthread`" library flags at the end of the `experiment_tmm_pt2`'s load line.
5. Again, run multiple tests to make sure that your timing values are accurate. Choose command line arguments for `-n`, `-m`, and `-p` that give you run times of at least a few seconds and check them against your computer's clock and/or `bash`'s `time` builtin. How does doubling any of these parameters affect the CPU time?
6. Run the same test several times and note the elapsed and CPU times. Are they always the same? If not, how much do they vary? Why do you think they vary? If you were reporting this to someone else, what would be the best way to describe your result and its variation (if any)? Can you think of any way to reduce the variation?
7. Look at the individual thread statistics for varying numbers of threads (`-t`). Do all threads get about the same amount of time? Do all threads get about the same number of rows to multiply?
8. Investigate the effect of varying the number of threads on the elapsed and CPU times. In general, increasing the number of threads on a multicore CPU should cause the elapsed time to decrease at first. If you don't see this, there's a bug in your program (and points will be deducted). But is there a number of threads after which that decrease stops or slows down a lot? If so, what is that number and can you think of a reason for this? (Hint: hardware)
9. Either use `cslab` or `ssh` to `elec` to run your `experiment_tmm_pt2` experiments (#5 - #8) there. These will be the results you report. Finding a time whent fewer (ideally, no) other people are logged on (using *who(1)*). It might be necessary to log in at odd times for this to happen.
10. With your favorite word processor or text editor, create a 2-3 page double-spaced report in PDF (call it `report.pdf`) on your experimental results for Parts 1 and 2. Your grade will depend on this report. In particular, the grader will have these criteria:

    - This report should be *readable*, as if you were presenting your results to other students who were unfamiliar with this lab. This lab is as much a writing exercise as a programming one. (Think: Writing Portfolio)
    - Don't include your raw results as numbers. Present them as a graph. Think about kinds of graphs: linear, semi-log, and log-log. Choose wisely: One of them is better than the others to prove a point you'll be expected to show for this lab.

   - The report should answer the questions asked above.
   - The grader will be looking for particular insights and will deduct points if they are not there.

11. Be sure to include `report.pdf`, `mat_mul_pt2.h`, `mat_mul_pt2.c`, `experiment_tmm_pt2.c`, the files from Part 1, and any other files necessary for `make` to work for both parts in the submission tarball.