Math 4610 Tasksheet 2

Jacob Fitzgerald (A02261889)

Docs

https://jfitzusu.github.io/math4610/

Code

https://github.com/jfitzusu/math4610/tree/main/Assignment07
https://github.com/jfitzusu/math4610/tree/main/mymodules/operations

All test code can be found in the *test07.c* file in the Assignment06 directory, while code for specific functions can be found under the c files named after them in the operations directory.

Task 1

For this task we have to rewrite our python multiplication function in c. This is actually rather difficult, because for some reason c doesn't beleive in syntatic sugar. This means we have to manually allocate memory for each mattrix. As a result, our code actually becomes rather obotuse.

Implementation (Serial):

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

float ** matrixMultiply(float** matrix1, float** matrix2, int s1, int s2, int s3)
{
    float** result = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        result[i] = (float*)malloc(s3 * sizeof(float));
    }

    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s3; j++) {
            for (int k = 0; k < s2; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }

    return result;
}
```

This function accepts the following parameters:
 * matrix1: The First Matrix to Multiply, Represented with a Pointer to a List of Pointers
 * matrix2: The Second Matrix to Multiply, Repersented with a Pointer to a List of Pointers
 * s1: The Rows in the First Matrix
 * s2: The Columns in the First Matrix, and Rows in the Second
 * S3: The Columns in the Second Matrix

This is extremley similar to our python code, however, in this case, we don't actually call the dot product function, instead we just use a third loop. This means that our code will break the second it receives size values/matrices that don't match up.

To make sure our code works, let's test it on some matrices using the following code:

```
void testMatrixMultiply() {
    int s1 = 10;
    int s2 = 30;
    int s3 = 20;
    float matrix1Template[10][30] =
        {{4, 4, 6, 1, 7, 3, 7, 7, 2, 9, 1, 0, 6, 8, 7, 6, 0, 9, 3, 3, 0, 1, 9, 1,
2, 4, 1, 6, 8, 7},
        {1, 6, 0, 5, 6, 0, 0, 8, 5, 6, 0, 7, 0, 6, 4, 4, 5, 4, 4, 5, 9, 1, 0, 8,
8, 1, 8, 6, 7, 1},
        {6, 7, 6, 2, 9, 3, 9, 4, 9, 5, 3, 6, 5, 1, 5, 0, 5, 0, 9, 7, 8, 7, 5, 9,
6, 9, 2, 1, 2, 2},
        {6, 7, 6, 9, 8, 6, 5, 1, 5, 7, 6, 2, 8, 4, 9, 1, 9, 2, 5, 8, 8, 1, 4, 0,
3, 6, 2, 1, 1, 3},
        {4, 4, 3, 0, 6, 4, 8, 4, 9, 2, 4, 8, 7, 8, 1, 1, 1, 6, 1, 1, 2, 2, 1, 3,
5, 2, 6, 2, 6, 5},
        {5, 4, 0, 1, 2, 5, 9, 3, 7, 7, 6, 3, 4, 3, 5, 3, 0, 0, 4, 8, 6, 3, 9, 6,
0, 4, 9, 9, 0, 3},
        {0, 2, 4, 9, 7, 1, 7, 4, 3, 9, 8, 4, 7, 9, 5, 9, 9, 0, 3, 3, 4, 0, 0, 8,
1, 1, 9, 5, 0, 7},
        {7, 2, 0, 4, 1, 0, 9, 6, 1, 8, 7, 4, 2, 8, 1, 1, 7, 3, 8, 1, 7, 0, 6, 6,
9, 7, 6, 7, 4, 6},
        {5, 0, 0, 9, 5, 2, 3, 6, 0, 2, 8, 5, 0, 5, 1, 2, 2, 6, 6, 2, 8, 4, 2, 4,
6, 0, 8, 0, 7, 9},
        {4, 4, 6, 5, 7, 8, 3, 2, 6, 8, 8, 0, 8, 7, 2, 6, 1, 0, 7, 0, 6, 9, 8, 7,
4, 3, 4, 2, 3, 0}};
    float matrix2Template[30][20] =
        {{4, 3, 4, 4, 3, 9, 6, 4, 7, 5, 2, 9, 4, 4, 1, 8, 0, 5, 0, 1},
        {1, 2, 1, 7, 9, 0, 4, 5, 0, 2, 9, 2, 2, 7, 1, 5, 9, 8, 5, 0},
        {9, 3, 7, 0, 7, 9, 9, 6, 8, 2, 6, 1, 0, 8, 2, 8, 9, 1, 9, 7},
        {0, 5, 8, 0, 9, 8, 4, 5, 5, 4, 9, 2, 5, 4, 1, 8, 8, 0, 0, 7},
        {9, 7, 4, 7, 3, 2, 1, 3, 7, 0, 6, 4, 8, 7, 4, 9, 2, 9, 8, 4},
        {8, 5, 4, 1, 3, 0, 3, 7, 1, 6, 9, 5, 9, 4, 7, 0, 3, 9, 1, 5},
        {1, 0, 7, 0, 9, 9, 3, 9, 9, 7, 1, 7, 1, 6, 5, 2, 9, 8, 9, 8},
        {9, 2, 8, 5, 4, 3, 2, 3, 5, 9, 9, 5, 9, 3, 8, 9, 8, 2, 9, 1},
        {8, 6, 5, 2, 7, 9, 7, 4, 4, 2, 4, 8, 2, 6, 8, 8, 8, 6, 9, 8},
        {4, 7, 9, 3, 4, 1, 1, 3, 0, 7, 2, 1, 3, 1, 8, 0, 5, 6, 6, 7},
        {7, 2, 9, 6, 7, 6, 5, 3, 3, 2, 0, 4, 1, 2, 7, 7, 8, 6, 3, 0},
        {4, 0, 1, 6, 0, 3, 8, 6, 3, 9, 7, 2, 5, 1, 5, 8, 1, 8, 8, 5},
        {7, 4, 2, 8, 0, 4, 8, 9, 8, 6, 2, 7, 8, 4, 8, 8, 8, 1, 1, 0},
        {5, 5, 8, 2, 0, 7, 2, 9, 0, 1, 3, 4, 7, 1, 8, 3, 6, 9, 0, 7},
        {6, 5, 0, 1, 1, 8, 0, 3, 3, 2, 6, 9, 8, 5, 3, 6, 9, 3, 1, 0},
        {8, 9, 7, 7, 8, 8, 8, 1, 8, 3, 1, 8, 6, 7, 8, 3, 4, 1, 5, 9},
        {3, 5, 4, 2, 0, 4, 1, 2, 8, 1, 4, 5, 5, 7, 4, 5, 8, 8, 0, 1},
        {5, 1, 5, 7, 6, 0, 6, 4, 8, 4, 2, 5, 2, 2, 2, 7, 7, 0, 8, 5},
        {0, 5, 5, 7, 2, 8, 0, 7, 5, 1, 2, 8, 2, 3, 5, 6, 1, 4, 8, 0},
        {5, 4, 8, 4, 5, 5, 1, 1, 3, 1, 9, 2, 1, 5, 6, 2, 9, 5, 4, 8},
        {1, 1, 7, 4, 2, 1, 0, 7, 3, 8, 5, 9, 6, 9, 3, 9, 7, 7, 6, 8},
        {2, 4, 0, 8, 1, 6, 5, 7, 8, 8, 8, 6, 3, 0, 3, 8, 2, 0, 8, 5},
        {2, 4, 8, 5, 8, 0, 7, 2, 9, 6, 1, 4, 7, 3, 0, 9, 8, 5, 9, 2},
        {1, 0, 5, 2, 4, 8, 2, 6, 4, 9, 2, 9, 0, 2, 0, 3, 1, 3, 8, 5},
        {2, 3, 0, 6, 9, 4, 6, 3, 8, 7, 8, 8, 9, 0, 2, 6, 8, 2, 0, 4},
        {8, 5, 5, 7, 3, 4, 1, 6, 1, 5, 3, 5, 3, 5, 4, 8, 6, 7, 4, 0},
        {0, 3, 2, 9, 4, 7, 2, 6, 2, 8, 5, 3, 9, 4, 9, 6, 7, 8, 8, 4},
        {4, 7, 0, 0, 6, 9, 9, 1, 2, 1, 6, 6, 2, 1, 4, 8, 2, 0, 2, 6},
        {6, 7, 3, 2, 3, 8, 4, 4, 6, 5, 0, 7, 0, 2, 8, 9, 8, 3, 6, 9},
        {5, 0, 7, 9, 9, 9, 7, 6, 2, 4, 6, 0, 4, 5, 6, 6, 3, 3, 0, 2}};

    float** matrix1 = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        matrix1[i] = (float*)malloc(s2 * sizeof(float));
    }

    float** matrix2 = (float**)malloc(s2 * sizeof(float*));
    for (int i = 0; i < s2; i++) {
        matrix2[i] = (float*)malloc(s3 * sizeof(float));
    }

    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s2; j++) {

            matrix1[i][j] = matrix1Template[i][j];
        }
    }

    for (int i = 0; i < s2; i++) {
        for (int j = 0; j < s3; j++) {
            matrix2[i][j] = matrix2Template[i][j];
        }
    }

    float** result = matrixMultiply(matrix1, matrix2, s1, s2, s3);

    printf("Result:\n");
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 20; j++) {
        printf("%i ", (int) result[i][j]);

        }
        printf("\n");
    }



}
```

This code simply sets up two matrices, one 10 x 30, and the other 30 x 20, multiplies them with our funciton, and prints the results. Because of the way we set up our function parameters, it's actually a massive pain to pass anything into it.

Results:

```
Result:
686 554 689 560 641 678 569 601 644 548 519 650 582 504 650 794 804 555 666 579
488 482 552 538 528 630 422 546 510 602 635 671 588 468 601 775 722 594 643 610
643 535 704 664 654 790 532 765 731 709 709 842 608 657 637 947 862 781 846 608
649 571 724 573 617 702 489 681 643 545 689 708 655 671 638 859 920 740 584 551
564 390 520 531 514 626 528 626 540 554 497 599 520 433 622 724 680 604 606 530
512 475 642 536 610 694 494 600 518 610 545 679 544 503 632 726 733 661 679 559
599 545 751 571 622 818 514 665 592 584 588 656 640 581 738 758 812 673 606 624
497 472 715 595 639 754 502 671 614 676 533 739 598 478 639 832 783 685 621 544
438 372 614 577 547 643 431 581 546 591 532 586 546 413 559 767 642 526 550 513
617 583 699 605 604 706 555 708 659 639 570 753 630 527 654 827 763 645 707 598
```

These match up to results obtained by an online matrix multiplcation tool (WRA), and the first index matches with hand calculated values. This means our function most likely works. Now it's time to parellelize it.

Implementation (Paralell):

```c

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define NUM_THREADS 4
float ** matrixMultiply(float** matrix1, float** matrix2, int s1, int s2, int s3)
{
    float** result = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        result[i] = (float*)malloc(s3 * sizeof(float));
    }

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        for (int i = id; i < s1; i += numThreads) {
            for (int j = 0; j < s3; j++) {
                for (int k = 0; k < s2; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
    }

    return result;
}
```

Now we can run the test code again.

Results:

```
Result:
686 554 689 560 641 678 569 601 644 548 519 650 582 504 650 794 804 555 666 579
488 482 552 538 528 630 422 546 510 602 635 671 588 468 601 775 722 594 643 610
643 535 704 664 654 790 532 765 731 709 709 842 608 657 637 947 862 781 846 608
649 571 724 573 617 702 489 681 643 545 689 708 655 671 638 859 920 740 584 551
564 390 520 531 514 626 528 626 540 554 497 599 520 433 622 724 680 604 606 530
512 475 642 536 610 694 494 600 518 610 545 679 544 503 632 726 733 661 679 559
599 545 751 571 622 818 514 665 592 584 588 656 640 581 738 758 812 673 606 624
497 472 715 595 639 754 502 671 614 676 533 739 598 478 639 832 783 685 621 544
438 372 614 577 547 643 431 581 546 591 532 586 546 413 559 767 642 526 550 513
617 583 699 605 604 706 555 708 659 639 570 753 630 527 654 827 763 645 707 598
```

It produces the exact same results as before, so it looks like we haven't messed anything up. As for timing, we'll save that for our future problems.

Task 2

For this task, we have to implement the Hadamard product of two vectors. This is calculated by taking the peicewise product of each vector component, which means that our vectors must be of the same length. As said above, c is actually a massive pain to work with when it comes to dynamic memory, so even though this could be accomplished in a single line in a lot of languages, there's a lot of setup that needs to be done. Additionally, as far as I can find, it's not really possible to get runtime information about dynamic array sizes, so our code will crash the momemnt something goes wrong.

Implementation (Serial):

```
 #include <omp.h>

float * vectorHadamard(float* vector1, float* vector2, float size, double* time)
{
    float* result = (float*)malloc(size * sizeof(float));
    double startTime = omp_get_wtime();
    for (int i = 0; i < size; i++) {
        result[i] = vector1[i] * vector2[i];
    }

    *time = omp_get_wtime() - startTime;
    return result;
}
```

This function takes the following parameters:
* vector1: The first vector to multiply, represented by a pointer to an array of floats
* vector2: The second vector to multiply, represented by a pointer to an array of floats
* size: The length of the two vectors
* time: A pointer to a variable used to store timing information

As you can see, the actual meat of the code is literally just a loop. This means it should be pretty simple to test.

Testing Code:

```
void testVectorHadmard() {
    int size = 10;
    float vector1Template[] = {2, 7, 8, 12, 89, 1, 0, 0, 2, 6};
    float vector2Template[] = {7, 23, 8, 1, 1, 27, 7, 2, 1, 0};
    float* vector1 = (float*)malloc(size * sizeof(float));
    float* vector2 = (float*)malloc(size * sizeof(float));
    for (int i = 0; i < size; i++) {
        vector1[i] = vector1Template[i];
        vector2[i] = vector2Template[i];
    }
    double time;
    float* result = vectorHadamard(vector1, vector2, size, &time);

    printf("Results: ");
    for (int i = 0; i < size; i++) {
        printf("%i ", (int) result[i]);
    }
    printf("\n");
}
```

This function sets up two vectors, and calculates the Hadamard multiple using our code. IT's pretty simple overall, but a lot of work is needed to dynamically allocate the vector sizes.

Results:

```
Results: 14 161 64 12 89 27 0 0 2 0
```

If we go over our original vectors member by member, we can see that the product is exactly what we'd expect, so we can be fairly confident our code works. Next up, let's try making the code parallel.

Task 3

This is actually fairly easy to do. Because our result is already a vector, we don't even have to set up/consolidate results like before. Also, the loop in our code allows us to reapply the same "array slicing" techinique as before.

Implementation (Parallel):

```
#include <omp.h>
#include <math.h>

#define NUM_THREADS 4
float * vectorHadamardOMP(float* vector1, float* vector2, float size, double*
time) {
    float* result =  (float*)malloc(size *  sizeof(float));
    double startTime = omp_get_wtime();
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {



        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        for (int i = id; i < size; i+=numThreads) {
            result[i] = vector1[i] * vector2[i];
        }
    }

    *time = omp_get_wtime() - startTime;
    return result;
}
```

It should be noted that we didn't using the array padding method that we've used before. Thats because we're using dynamic memory allocation this time, which I'm not too familiar with. It should still be possible, it would just take a lot more boiler plate to set up and return the correct solution array. As we'll see soon, however, this doesn't seem to matter too much, as we still see significant speed ups.

Timing Code (Serial):

```
void timeVectorHadmard() {


    for (int i = 10; i < 1000000000; i *= 10) {
        float* vector1 = (float*)malloc(i * sizeof(float));
        float* vector2 = (float*)malloc(i * sizeof(float));
        for (int j = 0; j < i; j++) {
            vector1[j] = 10;
            vector2[j] = 10;
        }

        double time;
        float* result = vectorHadamard(vector1, vector2, i, &time);
        printf("Size %i: (%f)\n", i, time);
    }



}
```

Timing Code (Parallel):

```
void timeVectorHadmardOMP() {


    for (int i = 10; i < 1000000000; i *= 10) {
        float* vector1 = (float*)malloc(i * sizeof(float));
        float* vector2 = (float*)malloc(i * sizeof(float));
        for (int j = 0; j < i; j++) {
            vector1[j] = 10;
            vector2[j] = 10;
        }
        double time;
        float* result = vectorHadamardOMP(vector1, vector2, i, &time);
        printf("Size %i: (%f)\n", i, time);
    }


}
```

In both cases, we simply set up vectors of increasing lengths, multiply them together using our code, and report the timing results. Each time, we fill our arrays will the value 10. Initially I was worried that the c compiler would optimize our code out of existence because of this, but that doesn't seem to be the case while using the debug compiler at least.

Results:

```
(Serial)
Size 10: (0.000000)
Size 100: (0.000000)
Size 1000: (0.000003)
Size 10000: (0.000047)
Size 100000: (0.000378)
Size 1000000: (0.003575)
Size 10000000: (0.036099)
Size 100000000: (0.326706)

(Parallel [4 Cores])
Size 10: (0.000193)
Size 100: (0.000002)
Size 1000: (0.000003)
Size 10000: (0.000022)
Size 100000: (0.000510)
Size 1000000: (0.004614)
Size 10000000: (0.017619)
Size 100000000: (0.168196)
```

At lower sizes, our parellel runs slower. This is likely due to the inherent overhead of omp. However, once we get to big enough arrays, our parellel code starts to run much faster. It only runs about twice as fast on 4 cores, which could be due to the fact that we aren't optimizing for caching with padding this time. However, as mentioned above, doing that with dynamic memory is much more difficult.

Task 4

We can now take the idea of the Hadamard product and apply it to matrices. Rather than looking like our vector code above, this ends up looking more like our vector multiplication code from task 1. This is because a lot of the boilerplate for setting up a solution matrix is the same. The only real difference is how you calculate the value of a given index. In this case, since we're multiplying components together, its actually pretty easy, requiring one less loop than before.

Implementation Code (Parallel):

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define NUM_THREADS 4
float ** matrixHadamard(float** matrix1, float** matrix2, int s1, int s2, double*
time) {
    double startTime = omp_get_wtime();
    float** result = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        result[i] = (float*)malloc(s2 * sizeof(float));
    }

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        for (int i = id; i < s1; i += numThreads) {
            for (int j = 0; j < s2; j++) {
                result[i][j] += matrix1[i][j] * matrix2[i][j];
            }
        }
    }

    *time = omp_get_wtime() - startTime;
    return result;
}
```

This function accepts the following parameters:
 * matrix1: The First Matrix to Multiply, Represented with a Pointer to a List of Pointers
 * matrix2: The Second Matrix to Multiply, Repersented with a Pointer to a List of Pointers
 * s1: The Rows in Both Matrices
 * s2: The Columns in Both Matrices
 * time: A pointer to a variable to store timing info

An interesting thing to note about this code, is that we only divide up the first loop among our different cores. This is because we only need to divide one loop. If we split the data into *nths*, that that means each core of n should have an equal amount of data to work with, giving us the maximum possible benefit from parallelism. This can be done in many different ways, but the easiest way to do it is just by dividing up the outermost loop.

Testing Code:

```
void testMatrixHadamard() {
    int s1 = 5;
    int s2 = 6;
    float matrix1Template[5][6] = {{0, 1, 2, 3, 4, 5}, {6, 7, 8, 9, 10, 11}, {12,
13, 14, 15, 16, 17}, {18, 19, 20, 21, 22, 23}, {24, 25, 26, 27, 28, 29}};
    float matrix2Template[5][6] = {{0, 1, 2, 3, 4, 5}, {6, 7, 8, 9, 10, 11}, {12,
13, 14, 15, 16, 17}, {18, 19, 20, 21, 22, 23}, {24, 25, 26, 27, 28, 29}};

    float** matrix1 = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s1; i++) {
        matrix1[i] = (float*)malloc(s2 * sizeof(float));
    }

    float** matrix2 = (float**)malloc(s1 * sizeof(float*));
    for (int i = 0; i < s2; i++) {
        matrix2[i] = (float*)malloc(s2 * sizeof(float));
    }

    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s2; j++) {
            matrix1[i][j] = matrix1Template[i][j];
            matrix2[i][j] = matrix2Template[i][j];
        }
    }

    double time;
    float** result = matrixHadamard(matrix1, matrix2, s1, s2, &time);

    printf("Result:\n");
    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s2; j++) {
        printf("%i ", (int) result[i][j]);

        }
        printf("\n");
    }

        printf("Time: %f\n", time);



}
```

In this case, we simply multiply the same matrix with itself using our code, and print the results. This makes verification pretty easy, because if our code works, we can expect every element of our new matrix to be a perfect square.

Results:

```
Result:
0 1 4 9 16 25
36 49 64 81 100 121
144 169 196 225 256 289
324 361 400 441 484 529
576 625 676 729 784 841
Time: 0.000174
```

It appears that our code is working correctly, as every element coresponds to it's respective square.

This function is much easier to time than the default multiplication function due to the fact that it contains less loops, making its runtime grow much slower. If we run a quick test with square matrices, we can see that as the size increases, the runtime increases geometrically.

Timing Code:

```
void timeMatrixHadamard() {

    for (int k = 10; k < 100000; k*=10) {
        int s1 = k;
        int s2 = k;

        float** matrix1 = (float**)malloc(s1 * sizeof(float*));
        for (int i = 0; i < s1; i++) {
            matrix1[i] = (float*)malloc(s2 * sizeof(float));
        }

        float** matrix2 = (float**)malloc(s1 * sizeof(float*));
        for (int i = 0; i < s2; i++) {
            matrix2[i] = (float*)malloc(s2 * sizeof(float));
        }

        for (int i = 0; i < s1; i++) {
            for (int j = 0; j < s2; j++) {
                matrix1[i][j] = 10;
                matrix2[i][j] = 10;
            }
        }

        double time;
        float** result = matrixHadamard(matrix1, matrix2, s1, s2, &time);

        printf("Size %i: (%f)\n", k, time);
    }


}
```

Similar to above, we take increasingly larger matrices and try and multiply them together. Each matrix is simply filled will the value 10, but it doens't seem like the compiler optimizes it out, which is nice. We can then report the timing data to see just how fast our runtime grows.

Results:

```
Size 10: (0.000003)
Size 100: (0.000031)
Size 1000: (0.002350)
Size 10000: (0.353971)
```

Increasing the size by 10 seems to increase the runtime by about 100, which makes sense if we consider the double loop.

Task 5

It appears that the outproduct of two vectors is found by performing some "multiplication table" style arithmatic on them. You simple consider one vector as the x axis of such a table, and one as the y. The value of any given cell on the table is found by multiplying the corresponding matrix elements together. This means that for a vector of length *n* and a vector of length *m* the result will be an *nxm* matrix.

Implementation (Serial):

```
#include <omp.h>

float ** outerProduct(float* vector1, float* vector2, float size1, float size2,
double* time) {
    float** result = (float**)malloc(size1 * sizeof(float*));
    for (int i = 0; i < size1; i++) {
        result[i] = (float*)malloc(size2 * sizeof(float));
    }

    double startTime = omp_get_wtime();
    for (int i = 0; i < size1; i++) {
        for (int j = 0; j < size2; j++) {
            result[i][j] = vector1[i] * vector2[j];
        }
    }

    *time = omp_get_wtime() - startTime;
    return result;
}
```

This function accepts the following parameters:
  * vector1: The first vector to multiply, represented by a pointer to an array of floats
  * vector2: The second vector to multiply, represented by a pointer to an array of floats
  * size1: The length of the first vector
  * size2: The length of the second vector
  * time: A pointer to a variable to store timing info

In this case, it looks similar to our Hadamard product, except we have two loops this time. As mentioned above, this is because our result is now a matrix. Overalll, the code itself is very simple, it's just a lot of work to set up the matrix. Lets test it out.

Testing Code:

```
void testOuterProduct() {
        int size = 10;
    float vector1Template[] = {2, 7, 8, 12, 89, 1, 0, 0, 2, 6};
    float vector2Template[] = {7, 23, 8, 1, 1, 27, 7, 2, 1, 0};
    float* vector1 = (float*)malloc(size * sizeof(float));
    float* vector2 = (float*)malloc(size * sizeof(float));
    for (int i = 0; i < size; i++) {
        vector1[i] = vector1Template[i];
        vector2[i] = vector2Template[i];
    }
    double time;
    float** result = outerProduct(vector1, vector2, size, size, &time);

    printf("Results: \n");
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("%i ", (int) result[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

In this case, we're using the same vectors as in our hadamard tests, we're just taking a wildly different product.

Results:

```
Results:
14 46 16 2 2 54 14 4 2 0
49 161 56 7 7 189 49 14 7 0
56 184 64 8 8 216 56 16 8 0
84 276 96 12 12 324 84 24 12 0
623 2047 712 89 89 2403 623 178 89 0
7 23 8 1 1 27 7 2 1 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
14 46 16 2 2 54 14 4 2 0
42 138 48 6 6 162 42 12 6 0
```

The formatting on these is pretty garbage, so it's hard to tell what you're looking at at first. However, once you start testing it, it becomes pretty clear that our code worked. The biggest indicators of this are the rows/columns of zeros, which correspond to an elment in vector 1/2 that was a zero. With this, we can be pretty confident our code works.

When it comes down to it, I'm pretty sure the concept of an outer product can be applied to matrices, its just very hard to visualize. In the case of vectors, you can think of it as a 2D table. However, with matrices, you'd have to go beyond our normal existence of 3 dimensions, so it's pretty much impossible to visualize. However, the basic concept of taking and multiplying similar elements together still stands, so it should be possible. You can simply think of it as going along each elment in one matrix, and multiplying by the other matrix to get an entirley new matrix. The way you stack them together would end up with a higher dimensional matrix of some sort, but it should still work.

NOTE: After looking it up online, it looks like it's possible to have the result just be an extra large, extremly akwardly formatted matrix.

When it comes to parralellization, we can do what we've always been doing. Slicing up the first loop to give equal work to each core. Overall, not very hard.