Parallel and Distributed Computing Final Project

An exploration of matrix vector multiplication

Daniel Muckerman

Abstract

This paper examines the matrix vector multiplication algorithm, and various parallel implementations of said algorithm. Matrix multiplication has numerous applications in mathematics, especially in the fields of applied mathematics, physics, and engineering.

Introduction

Matrix multiplication is an important central operation in many numerical algorithms, as well as having applications in other types of programs like games, and is potentially time consuming, all of which has led it to becoming a well-studied problem in numerical computing. For this paper, I will be focusing on matrix vector multiplication, which is a subset of matrix multiplication.

Implementation

For tackling the problem of multiplying a square matrix of dimensions $n * n$ by a vector with length of n, we can determine the product as follows:

$$A = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix}, B = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \tag{1}$$

$$AB = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax & by & cz \\ px & qy & rz \\ ux & vy & wz \end{pmatrix} \tag{2}$$

And here's an example of a properly calculated resultant vector, given an $n * n$ matrix, and n length vector:

$$A = \begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, B = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 2 \end{pmatrix}, \tag{3}$$

$$AB = \begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 5*1 & 0*0 & 0*0 & 0*2 \\ 0*1 & 1*0 & 0*0 & 0*2 \\ 0*1 & 0*0 & 2*0 & 0*2 \\ 0*1 & 0*0 & 0*0 & 1*2 \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \\ 0 \\ 2 \end{pmatrix} \tag{4}$$

There are numerous practical applications for matrix vector multiplication in linear algebra alone, where the problem arises quite often. By choosing appropriate values, A can be used to represent a variety of transformations like rotations, scaling, reflections and shears. This is helpful in manipulating graphics and camera viewpoints in games and programs, as well as for mathematical purposes.

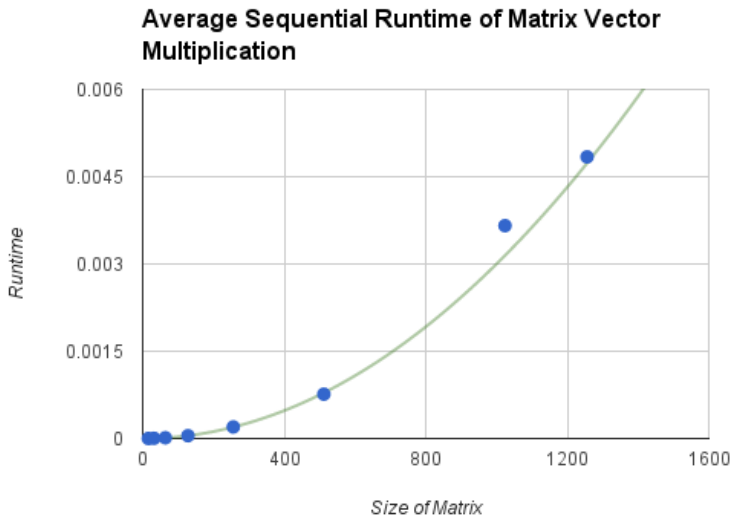
Results

In order to obtain these runtime results, each matrix of size $n*n$ was tested with both implementation 10 times, and averaged, to determine an approximation of the average runtime of both implementations.

Exact Runtime

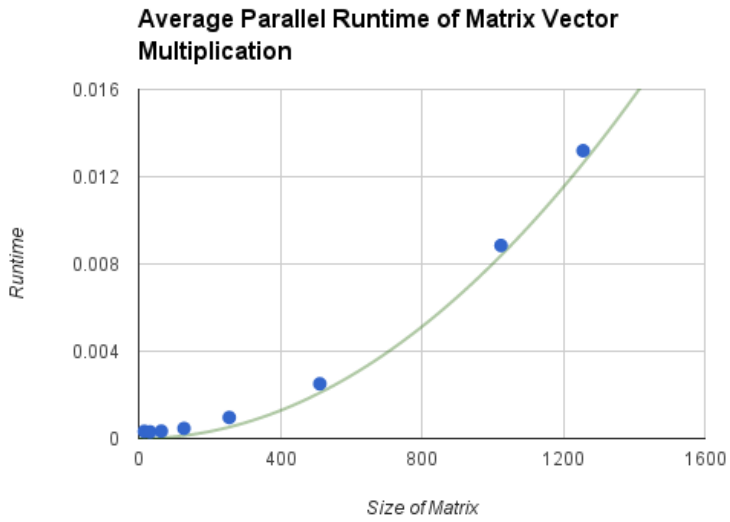
Sequential Using a sequential implementation of the problem I wrote in C, I ran the multiplication on random matrices of steadily increasing sizes, and these are the results:

n	Average Runtime	$3\times10^{-9} * n^2$
16	0.000001	0.000001
32	0.000004	0.000003
64	0.000013	0.000012
128	0.000049	0.000049
256	0.000199	0.000197
512	0.000762	0.000786
1024	0.003659	0.003146
1256	0.004841	0.004733



Parallel And, for comparison, I implemented a parallelized version of the above sequential program, also in C, using pthreads for the multithreading. I ran it through the same gamut of tests, increasingly larger random matrices, and compared the results to the output of the sequential method, to ensure they matched. These are the results, with the program run using 8 threads:

n	Average Runtime	$8\times10^{-9} * n^2$
16	0.000329	0.000002
32	0.000306	0.000008
64	0.000340	0.000033
128	0.000463	0.000131
256	0.000964	0.000524
512	0.002509	0.002097
1024	0.008849	0.008389
1256	0.013193	0.01262



So, at first glance, the parallel implementation seems to take longer to run, at least when it comes to exact runtime. But what about the asymptotic runtime?

Asymptotic Runtime

Sequential The core of the sequential implementation is this set of nested for loops:

```
// For every row in the array
for (int i = 0; i < n; i++) {
    // For every column in the row
    for(int j = 0; j < n; j++) {
        // Update result vector with results of calculation
        arr[i] = arr[i] + (mat[(i * n) + j] * vec[j]);
    }
}
```

Therefore, we can easily determine that the asymptotic runtime of the sequential implementation is $O(n^2)$, since we're dealing with nested loops of size n. But what about the parallel implementation?

Parallel The parallel implementation is set up such that the following worker function is called when each thread is created:

```
// Calculate the resultant vector
// Based on the passed in matrix and vector of size n
```

```
void *worker(void *p) {
    // Initialized Work struct from passed in pointer
    struct Work *work = p;

    // For every row in the chunk
    for (int i = work->start; i < work->end; i++) {
        // For every column in the row
        for(int j = 0; j < work->n; j++) {
            // Update result vector with results of calculation
            work->arr[i] = work->arr[i] + (work->mat[(i * work->n) + j] * work->vec[j]);
        }
    }

    return NULL;
}
```

Looking at the worker function, it should be relatively apparent that it has asymptotically the same runtime as the sequential implementation, $O(n^2)$, seeing as how it has roughly the same nested loops. In theory, it should run faster, or at least, just as fast as the sequential implementation, since multiple threads can concurrently attack smaller chunks of the potentially huge matrix.

Conclusion

So, why did the sequential implementation run faster? My best guess is that the overhead associated with creating, initializing, and then, following execution, waiting for each thread to finish before combining them, outweighs any sort of potential speed gains on matrices of the sizes used in this test.

This actually seems to be true, because in my tests, *decreasing* the number of threads actually *increases* the average runtime of the parallel implementation, which seems to support my hypothesis that the slowdown is due to the overhead from creation and joining of the threads.

For comparison to the above numbers, here's the actual runtime of the parallel implementation using only 2 threads, instead of 8.

n	Average Runtime	4.5×10^{-9} * n^2
16	0.000168	0.000001
32	0.000148	0.000005
64	0.000123	0.000018
128	0.000167	0.000074
256	0.000396	0.000295
512	0.001230	0.00118
1024	0.004784	0.004719

n	Average Runtime	$4.5\times10^{-9} * n^2$
1256	0.007125	0.007099

These numbers are a lot closer to the sequential version than when it runs with 8 threads.

It also seems likely that there is probably some sort of upper limit for **huge** matrices where the parallel implementation will finally overtake the sequential, where the overhead with creating the threads becomes negligible compared to the overall sequential runtime. But for smaller matrices like these, which are already approaching the limit of `int` variables, the overhead of creating and joining the threads seems like too much to reconcile, at least with this simple parallelization of the task.