CPS343 Parallel and High-Performance Computing
Project 2

Introduction

Given an $n \times n$ diagonalizable matrix A, a maximum amount of iterations $M > 0$, and tolerance $\epsilon > 0$, the dominant eigenvalue (called λ) and its corresponding eigenvector (called x) can be approximated utilizing an algorithm known as the Power Method. The Power Method finds the solution by first making an initial guess and slowly iterating towards the approximate solution, hopefully converging before M iterations have been reached. This project seeked to turn a serial version of this algorithm into a parallel one.

The dominant eigenvalue, which is the maximal positive eigenvalue of A, is important for Google's PageRank algorithm. Briefly, PageRank essentially consists of diagonalizing a 'link' matrix and exponentiating that matrix with its eigenvalues. Since the exponent is very large, the diagonalizable matrix is dominated by the largest eigenvalue[1]. Finding this dominant eigenvalue is computationally expensive, especially for large matrices, largely due to the matrix-vector multiplication that is required. Since matrix-vector multiplication is easily parallelized, it is worthwhile to pursue.

To solve this problem, OpenMP was used to parallelize the serial version of the C++ code for use on a 4 core computer. There were two different original serial programs, one that utilized a relatively un-optimized for-loop method to compute the linear algebra operations, and one that used highly optimized CBLAS (a C interface to Basic Linear Algebra Subroutines) function calls. Three different parallel programs were created from these serial programs. Firstly, a version of the for-loop program was written to include a single `#pragma omp parallel for` directive in the matrix-vector multiplication function. The second and third programs used one `#pragma omp parallel` directive around the main power method loop, creating a thread pool to execute the parts of the loop in parallel. The program was written in such a way that the matrix size did not necessarily need to be a multiple of the number of threads[2].

After conducting testing, results indicated that matrices of 100x100 to 1000x1000 experienced significant speed improvement of 50% to 75%. Smaller matrices of size 5x5 to 50x50 experienced performance degradation when running with more than one threads, and matrices of size 2000x2000 to 10000x10000 experienced limited performance improvements of around 15%.

---

[1] A good explanation can be found here: Why is PageRank an eigenvector problem?
[2] Option 2

If these methods were to be implemented in a production system, a serial version should be used for smaller matrices, and a different approach should be used for larger matrices so that the cache can be utilized more efficiently.

Overview

The specific algorithm that was parallelized is as follows:
If $A$is an $n \times n$ diagonalizable matrix, $\epsilon > 0$ is a tolerance, $M > 0$is the maximum allowed number of iterations, $\lambda$is the eigenvalue estimate, $\lambda_0$is the previous eigenvalue estimate, $y$is the eigenvector estimate, and $x$is the previous eigenvector estimate, then the Power Method algorithm proceeds as follows:

// The serial version
$x := (1, 1, 1, \ldots, 1)^T$
$x := x/||x||$
$\lambda := 0$
$\lambda_0 := \lambda + 2\epsilon$
$k := 0$
while $|\lambda - \lambda_0| \geq \epsilon$and $k \leq M$do
    $k := k + 1$
    $\lambda_0 := \lambda$
    $y = Ax$
    $\lambda := x^T y$
    $x = y$
    $x := x/||x||$
end while

// The parallel version
$x := (1, 1, 1, \ldots, 1)^T$
$x := x/||x||$
$\lambda := 0$
$\lambda_0 := \lambda + 2\epsilon$
$k := 0$
Parallel section
while $|\lambda - \lambda_0| \geq \epsilon$and $k \leq M$do
    # Single thread
        $k := k + 1$
        $\lambda_0 := \lambda$
    $y_i := A_i x$
    $\lambda_i := x_i^T y_i$
    # Critical section
    $\lambda = \lambda + \lambda_i$
    $z_i = y_i^T y_i$
    # Critical section
    $z = z + z_i$
    # Barrier
    $y_i = y_i / \sqrt{z}$
    # Critical section

$$x_i = y_i$$

end while

The parallelization begins directly before the main while loop, and continues to the end of the program. To prevent skipping iteration, only one thread needs to increment the loop counter k, while inside this section, it also updates the shared $\lambda_0$variable. Each thread then computes its own section of the $y$vector, denoted y_i. After this, each thread can compute its portion of the eigenvalue estimate λ_i. At this point, the each piece of the eigenvalue estimate must be accumulated, so each thread enters a critical section to update the value λ, which afterwards all threads have access to. Normalizing the eigenvector can also be done in parallel, so each thread computes its part of z, called z_i. Before each thread can use this value to scale its part of the y vector, it must be accumulated, so the program enters a critical section to sum each thread's piece of z. We must be sure that this process is finished before scaling, so each thread waits at a barrier for the accumulation of $z$to complete. Then each thread divides their portion of $y$by $\sqrt{z}$. After this is complete, $x$must be updated from y. Since every thread needs all of $x$for the matrix-vector multiplication, we enter a critical section so that each thread can copy their piece of $y$to the corresponding part of x. After this is finished, each thread now can access all of x, and the loop repeats if the tolerance or maximum number of loops has not been reached.

Implementation

The biggest pieces of data in the algorithm are the three different arrays which make up A, x, and y. A is a $n \times n$ matrix implemented as an array of doubles of size $n \times n$, stored contiguously in memory. The vector x is implemented as an array of size n. Each thread will be computing a piece of y, using part of $A$and all of x.To decide which part of $A$is being used by which thread, we call the function `decompose1d()`[3], which breaks an array into a given amount of pieces of approximately equal size. Since we want to partition rows, we pass in the amount of rows in the matrix (`n`) and the number of threads (`num_threads`), as well as the `rank` of the thread calling the function. It will pass back the starting and ending element indices of an array of the given size in the variables `first_mat_elem` and `last_mat_elem`. We can then do some simple arithmetic on these variables to get the starting and ending indices of the matrix. For example, say we had a matrix of size 5x5, and we wanted to evenly distribute the rows of this matrix over 3 threads. If we put `n=5` and `num_threads=3` into the function, we essentially are partitioning this array (which we will call `x`) :

0	1	2	3	4

[3] Provided by Dr. Senning

The array `x`.

Instead of this matrix (which we will call `A`) :

0	1	2	3	4
5	...	...	...	9
10	...	...	...	14
15	...	...	…	19
20	...	...	...	24

The 5x5 matrix `A`

So the partition of the first thread (`rank=0`) will start at `x[0]` and end at `x[1]`. The second will start at `x[2]` and end at `x[3]`, and the final thread's partition starts and ends at `x[4]`. Computing the corresponding start and end points for the matrix block is fairly simple. The starting point is computed by multiplying the original partition by `n`. So in this example, the first thread starts at $0 \times 5 = 0$, the second starts at $2 \times 5 = 10$, and the third starts at $4 \times 5 = 20$. We can easily verify that this is correct using the above table.

The last index of the partition is important for computing how many rows were partitioned to a given thread, and we can compute it with this code:

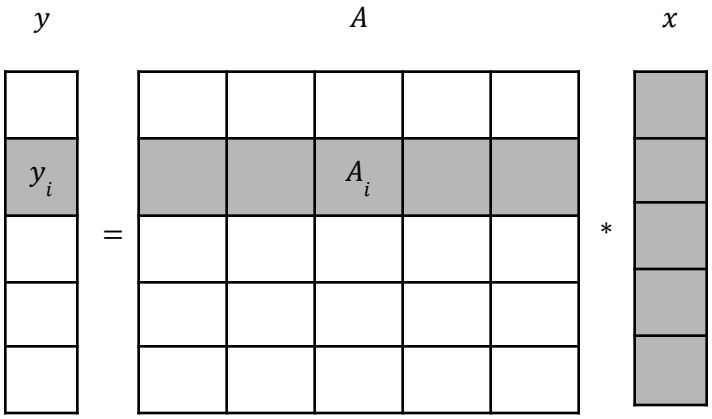
```
last_mat_elem = last_mat_elem * n + n - 1;
```

This calculates the first value of the corresponding row in the matrix, and then finds the last element of that row by adding `n-1`. In the example, the first thread would get $0 \times 5 + 5 - 1 = 4$, and the last thread would get $4 \times 5 + 5 - 1 = 24$. These values can be confirmed by looking at the above matrix.

Computing the number of rows for a given thread is equally simple; we just find the length of the block the thread was just partitioned and divide it by the number of rows in the matrix:

```
thread_num_rows =(last_mat_elem - first_mat_elem + 1) / n;
```

Using our example, the first thread would get $(9 - 0 + 1)/5 = 2$ rows, while the last thread would get $(24 - 20 + 1)/5 = 1$row.



Matrix-vector multiplication for a single thread.

After the matrix has been partitioned, each thread can compute its piece of the vector y. To compute the matrix-vector multiplication, each thread needs access to all of the vector x, and its chunk of the matrix A, with starting indices given as described above. A thread will compute as many elements of $y$as it was partitioned rows of A. The thread can then store the result it receives in a private variable y_i.

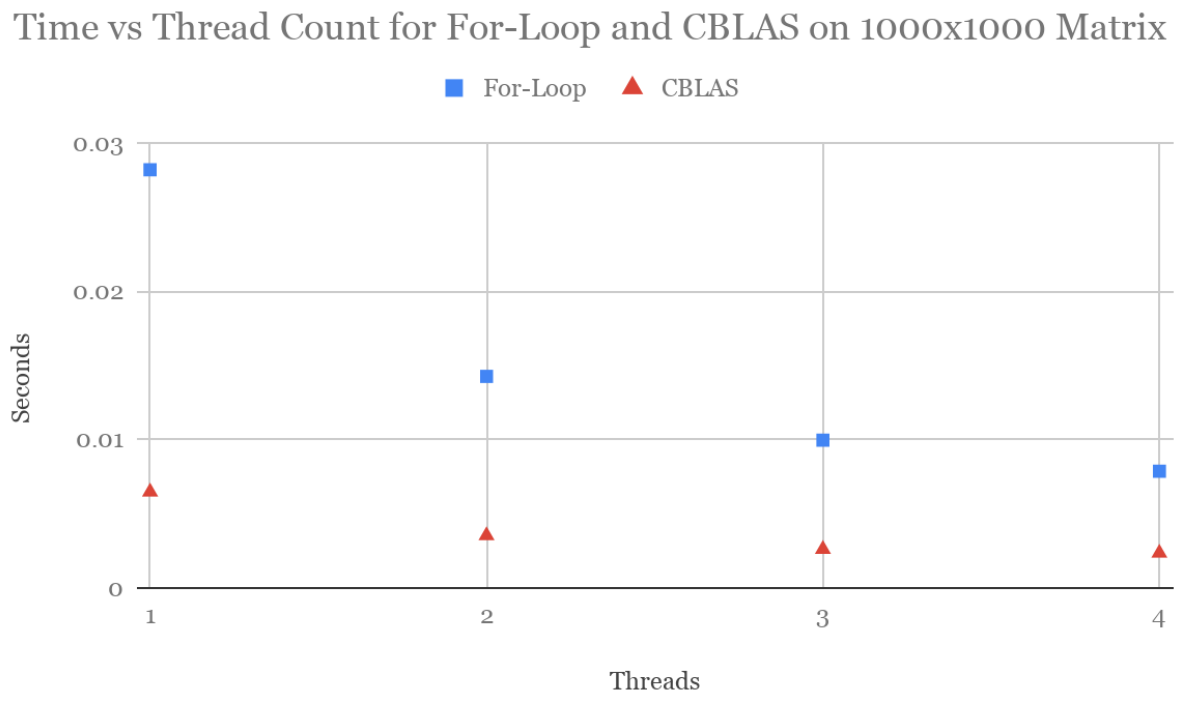
Discussion and Analysis of Results

Tests were conducted on matrices of sizes 5x5, 10x10, 50x50, 100x100, 500x500, 1000x1000, 2000x2000, 5000x5000, and 10000x10000 on Zechariah; one of the 4 processor minor prophets workstations. A script was written to test all 5 programs (3 parallel and 2 serial), running ./proj2_forloop and ./proj2_cblas 4 times each, incrementing the number of threads each time, from 1 to 4.

Matrix size	For-Loop Runtime (sec)	CBLAS Runtime (sec)	Parallelized For-Loop Runtime (sec)
5	0.00000312	0.0000128	0.000852
10	0.0000148	0.0000307	0.001237
50	0.000382	0.000199	0.001019
100	0.001544	0.000522	0.001043
500	0.028016	0.006382	0.007947

1000	0.187654	0.053649	0.055586
2000	0.449971	0.419603	0.349903
5000	3.77253	3.58387	3.20223
10000	14.6205	13.5329	12.3007

Matrix size	1 Thread (For)	2 Threads (For)	3 Threads (For)	4 Thread (For)	1 Thread (CBLAS)	2 Thread (CBLAS)	3 Thread (CBLAS)	4 Thread (CBLAS)
5	0.0000635	0.000118	0.000146	0.000439	0.0000698	0.000121	0.000154	0.000815
10	0.000113	0.00018	0.000221	0.000668	0.000126	0.000184	0.000239	0.000928
50	0.00054	0.000423	0.000421	0.00138	0.000346	0.00037	0.000394	0.00148
100	0.001689	0.001012	0.000847	0.001087	0.000671	0.000528	0.000509	0.000847
500	0.028169	0.014257	0.009955	0.007868	0.006476	0.003533	0.002613	0.002355
1000	0.18891	0.096733	0.068229	0.05427	0.057249	0.043187	0.043624	0.043724
2000	0.451642	0.369518	0.361935	0.348391	0.406202	0.400562	0.414538	0.4179
5000	3.76072	3.21092	3.17824	3.20042	3.55982	3.50142	3.50526	3.55922
10000	14.4836	12.4058	12.228	12.3068	13.5048	13.2589	13.405	13.5114



As this can see, on matrices of size 1000x1000, the for-loop program experiences significantly more speedup than the already heavily optimized CBLAS routines. From the table, we can see that smaller matrices (5x5-50x50) experience slower performance when multithreaded. This is most likely because the overhead required to utilize multiple threads does not necessarily correlate to improved performance when the work that can be parallelized is already small. As the matrix size becomes large (2000x2000-10000x10000) we see that the speedup also decreases. This probably due to cache misses as the matrix becomes huge, a problem that parallelizing does not necessarily solve.

Conclusion

The Power Method is a powerful tool behind arguably one of the most famous and impactful algorithms in the world; Google's PageRank. In order to improve performance of two different versions of the program, OpenMP was used for parallelization. The project was a good exercise in using OpenMP, as well as the parallelization of serial programs. After testing, it was revealed that OpenMP is very useful for matrices with dimensions in the range of 100x100 to 1000x1000, and moderately useful for larger matrices. The tests also showed that smaller matrices are unable to benefit from OpenMP because of the overhead required. In the future, it would be useful to implement blocking techniques to the For-Loop version of the program to try and approach the faster performance of the CBLAS functions, especially for the larger matrices.