Parallelizing Sparse Matrix Multiplication

1. Problem Description

The initial inspiration for our project came from thinking about the earliest published iteration of Google’s pagerank algorithm, which computes an adapted version of eigenvector centrality for the graph of internet websites (vertices are web pages and directed edges are hyperlinks between them). Rather than computing the eigenvectors of the adapted adjacency matrix directly, which is often computationally costly, the matrix can be understood as a Markov matrix and its stable state approximated by raising the matrix to very high powers.

For our project, we took on that last step and wrote code to raise the matrix to high powers in parallel. We noted that internet graphs tend to be very sparse: two randomly chosen pages are extremely unlikely to link to one another. Thus, we decided to parallelize sparse matrix multiplication, in particular taking large powers of sparse matrices. We sought to implement a program that would perform no extra computation (eg would not expand out and multiply any zero matrix entries) and, when using OpenMPI, would not send any zero matrix entries between processes.

1. Our Approach

We split the problem into two parts: finding a fast way to raise the matrix to high powers, and finding a fast (and parallelizable) way to perform the matrix multiplication.

1. Raising matrices to high powers. Rather than taking Ap by computing p multiplications of A \* Ak for each k < p, we simply square A, then square the result, then square the result, and so on. This takes only log2p iterations, which is a huge improvement, and it requires less storage because we do not need to keep the original matrix A after the computation begins. (Note: we actually do keep A, in order to compare our results with a serial computation, but this is not necessary.) Of course, this requires that p be a power of 2, which in a general case it might not be. However, for pagerank and other Markov stable state calculations, the exact value of p is irrelevant; it only matters that the matrix be raised to a sufficiently high power. Thus, we can use the log2 shortcut.
2. Parallelizing matrix multiplication. We used a similar tactic for matrix multiplication as the one found in assignment 3. For each multiplication, we distribute blocks of rows and columns among the processes. The matrices keep their columns throughout the computation, while the rows are passed around in a round-robin style. For the actual multiplication, we created a matrix multiplying kernel that performs a computation only when both the row element and column element are non-zero. In addition, this kernel only stores back to the results matrix the non-zero row dot products.
3. Details of Data Structures, Algorithms, Code Structure
4. Code Structure. Generic pseudocode for our program is as follows:

master:

generate a sparse matrix

distribute its columns to all workers, including self

all:

for each of log\_2(p) iterations {

distribute row blocks to all workers using alltoallv

(see "algorithms", below, for more detail)

for each of num\_procs iterations:

for each of n/num\_procs columns:

for each of n/num\_procs rows:

compute the dot product of row \* col

if > 0, store in results

send rows to left neighbor

receive rows from right neighbor

sort result rows

result matrix becomes new column matrix

}

master:

receive final results from all processes

compare with serial

1. Data Structures. In order to to store sparse matrices, we created two nested structs. The Matrix struct holds one int dimension, n, and an array of Vector structs. Each Vector struct holds an int array of indices, a double array of values, and an int length. The array of Vectors should be considered the n columns of the matrix, and each Vector’s arrays contain the row indices at which the column has nonzero entries, as well as those entries themselves and the total number of nonzero entries in the column. Note that the struct can also be interpreted as containing a Vector array of rows, instead of columns.
2. Raising the matrix to high powers. This required perhaps the most complex and interesting algorithms in order to work successfully. As discussed above, we raised the matrix to power p by squaring A, then squaring the result, then squaring that result, and so on log2p times. Thus, at the end of a single iteration, each process is left with a block of columns of A, a block of rows of A, and a block of columns of A2. At the beginning of the next iteration, it needs to have a block of columns of A2 and a block of rows of A2. The naïve way to accomplish this would be to gather the columns of A2 back to the master process, have the master transpose the columns to get rows, and then redistribute n/num\_procs rows to each process and start again. However, this creates a severe bottleneck at the master and, by adding a serial portion to the code after each multiply, would significantly hurt parallel speedup.

The intuition for improving this is simple: transpose the matrix by using alltoall calls to distribute the row blocks to each process, without going through the master as an intermediary.

The implementation for sparse matrices, however, is quite complex. In brief, we did the following:

for each of the n/num\_procs columns per process:

walk through the column to compute the number of

elements to be sent to each other process

[lines 99-100, functions at lines 386 - 407]

send those elements using alltoallv into a receive buffer

[lines 102-110]

walk through the receive buffer, copying the elements

into their respective rows and computing the appropriate

column index based on which process sent the element and

which iteration is currently in place

[lines 113-126]

sort the rows by column index, because some indices may have come in out of order

[lines 130-132, function at lines 409-432]

1. Summary of Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| serial | n |  |  |  |
| power | 512 | 1024 | 2048 | 8192 |
| 2 | 0.648971 | 5.071271 | 40.627827 | too long |
| 32 | 2.135207 | 18.387104 | 151.394892 | too long |
| 32768 | 5.828904 | 50.924429 | 404.035826 | too long |
|  |  |  |  |  |
|  |  |  |  |  |
| procs = 2 | n |  |  |  |
| power | 512 | 1024 | 2048 | 8192 |
| 2 | 0.416666 | 3.183908 | 25.583827 | 1643.066456 |
| 32 | 1.566698 | 12.149026 | 104.620385 | too long |
| 32768 | 4.522234 | 34.918049 | 294.494091 | too long |
|  |  |  |  |  |
|  |  |  |  |  |
| procs = 8 | n |  |  |  |
| power | 512 | 1024 | 2048 | 8192 |
| 2 | 0.197932 | 0.958325 | 6.461173 |  |
| 32 | 0.509709 | 3.622523 | 26.542911 | too long |
| 32768 | 1.292329 | 9.545945 | 77.165021 | too long |

1. Conclusions
   1. Overall, we found that parallel showed considerable speedup compared to our serial program. Even our serial program is relatively fast, though, because it benefits from both the efficient computation using our sparse matrix data structure and from the technique of only doing log2p computations instead of power p.
   2. We were not able to run n = 8192 serially or for high powers in parallel. With serial, this was simply because the code was too slow; with parallel we were able to compute the timings for p = 2, but greater powers were no longer feasible. We believe this is partly because the sparse matrices, once raised to high powers, are no longer sparse. At that point, we are doing a lot of extra computation to maintain the sparse data structures when they are no longer necessary (and are in fact cumbersome). See limitations section for more thoughts on this.
   3. Speedup and Scalability:  
      Below is the chart showing speedup (serial time / parallel time) across different sizes of matrices and different powers. We see that the algorithm scales well, because not only do we get increased speedup with more processors, but when using many processors, the speedup increases as the matrices get larger.

For two procs:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | n |  |  |
|  |  | 512 | 1024 | 2048 |
| power | 2 | 1.557532892 | 1.592781889 | 1.588027741 |
|  | 32 | 1.362870828 | 1.513463219 | 1.447087888 |
|  | 32768 | 1.288943474 | 1.458398463 | 1.371965816 |

For eight procs:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | n |  |  |
|  |  | 512 | 1024 | 2048 |
| power | 2 | 3.278757351 | 5.291807059 | 6.287995539 |
|  | 32 | 4.189070627 | 5.07577288 | 5.70377876 |
|  | 32768 | 4.510387061 | 5.334666081 | 5.235997098 |

1. Future Steps and Limitations
   1. One limitation of our program that we found after running it is that sparse matrices, when raised to high powers, tend to no longer be very sparse. However, it would be very easy for our program to detect when this happens (simply by counting the non-zero entries as it computes them, and comparing the final count to n2) and then convert the matrix representation to a simple 2-D array. At that point, it could turn the computation over to an adapted version of our assignment 3 (or any other number of parallelized matrix multipliers that we are sure must exist) to finish the computation.

In addition, even if not used for powers of sparse matrices, the core kernel of our program still implements a generic sparse matrix multiplication algorithm, and so could be used to multiply two different sparse matrices quickly and in parallel.

* 1. Another interesting question to pursue is looking at the accuracy of the program. Though our results agreed with our serial computation (indeed, we used this as a guage of whether or not the parallel program was working properly), we did not test the accuracy of our algorithms against a more precise calculator. Thus, floating point errors may have added up over time to produce a somewhat inaccurate result.

However, it is possible that the technique we used to compute high powers of the matrix may have mitigated this problem somewhat. Since we only computed log2p multiplications, we had many, many fewer computations over which floating point errors could add up. Even if the technique we used did not make our computation more accurate (or perhaps made our computation less accurate), the speedup we gain from using this technique is so significant that it can still be worthwhile for applications in which efficiency is favored over accuracy.

* 1. We might have been able to improve our algorithm by thinking more carefully about the sorting algorithm used after transposing the matrix. The array we are sorting is not a random array, but rather has some fixed properties based on the way data is placed into it. The array will always have num\_procs consecutive runs in it that are already sorted, because the processes themselves have columns of increasing index order (column indices of proc 0 < column indices of proc 1 < … < column indices of proc num\_procs). Currently, we use insertion sort, which tends to be fast when dealing with largely pre-sorted data, but if we knew the lengths of the runs, we could get O(n\*log(num\_procs)), where num\_procs = max\_run\_length. If we saved the lengths of the runs as we copied data into the rows, we could simply merge sort each run with the next in linear time.
  2. It might also be worth reconsidering the data structure we used to store sparse matrices. We were efficient in terms of space, and the use of structs allowed us to keep our code somewhat more readable, but other data structures might have been better suited to OpenMPI. In particular, storing each column of the matrix in a separate array slowed down our program because it meant separate sends and receives for each row. A data structure that might have worked better could be three 1-D arrays, one containing the row indices of non-zero entries, one containing the column indices, and one containing the values themselves. However, this probably would have required much trickier code to handle the indices, and might have caused more time-consuming sorting as longer sequences of array elements came in before their final indices were known.

1. Division of labor

We split up the work very evenly and spent a lot of time programming together. We believe that we each spent about the same amount of time working on the project. Below is a vague sense of which tasks were completed by whom, but there was a lot of overlap, revision of the other person’s code, and smaller tasks not mentioned here.

Together:

* researched sparse matrix multiplication and google’s pagerank algorithm
* discussed and pseudocoded different approaches and data structures
* wrote a significant amount of MPI code, discussed how to transpose matrices in parallel
* debugged. A lot.
* ran timings

Amelia:

* wrote serial code for computing sparse matrices
* wrote code for computing sendcounts and senddisplacements for the alltoall’s, as well as the algorithm to sort received alltoall buffers into rows
* wrote report

Jay:

* wrote scaffolding code for generating, allocating and freeing, serially transposing, printing, and comparing sparse matrices
* wrote the MPI code for alltoalls and send/receive (transposing and round-robin)
* additional debugging (seg faults on high matrix powers!)