CPE419 and CPE466 Joint Project
Parallel Page Rank

Gilbert, Andrew
apgilber@calpoly.edu

Miller, Drew
dmille26@calpoly.edu

Terrell, Josh
jmterrel@calpoly.edu

Yost, Morgan
yost@calpoly.edu

Abstract

We present the results of a sparse-matrix, parallel implementation of PageRank. We explain the differences in the output between this implementation and the previous work in lab 3.

1 Introduction

The page rank algorithm can be implemented in many different ways, as demonstrated by this report. One version of the implementation is the solving of a system of linear equations, one equation at a time as performed by the node based evaluation implementation, refered to as the "original" implementation. This implementation offered the flexability to use multiple threads. Another apporach to the page rank problem is utilizing matrix math to simultaneously solve the system of linear equations and take advantage of optimized matrix libraries. This was the approach used in order to gain imporvement in perfoemance. In the following sections we will discuss how we implemented the page rank algorithm, how we handled memory for large data sets and how we took advantage of parallel architectures available on the MIC and GPU.

2 Implementation Overview

The previous implementation of page rank represented the graph in structs and used threads to parallelize computation. To further increase performance, we vectorized the pagerank computation and used matrix math libraries for the Intel Xeon Phi coprocessor and NVIDIA GPU implementations.

2.1 The non-zero transition probability problem

A problem with vectorizing the pagerank algorithm is that without any memory optimizations, an $N \times N$ (where N is node count) matrix must be created to represent the *transition probability matrix*, P. There is a non-zero probability of transitioning from any node to any other node, so P is throughly dense. For the *live journal* dataset, we simply could not represent a 4M $\times$ 4M matrix in memory, so we got creative.

The equation for the original transition probability matrix is eq. (1), where P_{ij} is the value of the row i and column j representing the probability of transitioning from node i to node j.

$$P_{ij} = \begin{cases} \frac{1-d}{N} + \frac{d}{\text{outdegree}(i)} & \text{if } i \to j \text{ is an edge} \\ \frac{1-d}{N} & \text{if } i \to j \text{ is not an edge} \\ \frac{1}{N} & \text{if } i \text{ is a sink node} \end{cases} \tag{1}$$

In order to represent the matrix as a sparse matrix, we subtracted $\frac{1-d}{N}$ from every cell in the matrix to set one of the above cases to zero. We also set each row to zero if it was a sink node so that the definition of P changed to eq. (2)

$$P_{ij} = \begin{cases} \frac{d}{\text{outdegree}(i)} & \text{if } i \to j \text{ is an edge} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The problem with modifying P like this is that P has a completely different value than it should. Multiplying $\pi^T P$ no longer yields the next π for the iteration. When making these modifications, we had to adjust the equation for the iteration from eq. (3) to eq. (4):

$$\pi = P^T \pi \tag{3}$$

$$\pi = P^T \pi + \frac{1-d}{N} \cdot \Sigma\pi + \frac{d}{N} \cdot \pi^T S \tag{4}$$

where S is a vector such that eq. (5) holds.

$$S_i = \begin{cases} 1 & \text{if row } i \text{ of } P \text{ is a sink node} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Not only did this solve the problem of representing a $N \times N$ matrix, but by reducing the matrix to E (where E is edge count) cells of a sparse matrix, we accidently, conviniently, and significantly reduced the number of operations to be computed on every iteration, assuming nodes have far fewer out edges than the number of nodes on the graph. We were able to further reduce the number of computations by completely eliminating the last term of the addition so that the actual computation that we performed was as follows.

$$\pi = P^T \pi + \frac{1-d}{N} \cdot \Sigma\pi \tag{6}$$

Because the number of sink nodes in our data sets was significantly less than the number of nodes, the last term of the original equation did not significantly alter the results. This was convenient becuase the Intel Math Kernal Library and

NVIDIA Cuda libraries had matrix vector multiplication with vector addition to exactly match the form of the final equation. Furthermore, beause we initialized π to a vector of length N all of value $\frac{1}{N}$, the sum of π should always be 1. So we did not actually need to calculate the sum, but rather, we could simply add a vector of $\frac{1-d}{N}$ to π.

2.2 Algorithm changes

As a result of the change in implementation from a link-based system for lab 3 to a matrix-based system for this project, the handling of sink nodes had to change. As discussed above, we treat sink nodes as always transitioning to another node in the matrix version. If we do not, there is an incorrect probability of transitioning away from them. On the other hand, in lab 3, we were unable to take that route, since we always followed links *to* a node and did not track the links away from each node. This means that the results will differ when sink nodes are involved.

In general, this only can affect the results of directed graph datasets. Undirected graphs, since they have no sink nodes, should match the lab 3 results.

2.3 Memory movement

In our previous implementation, we parsed the datasets in python and used cffi to pass data to C to construct the graph. We optimized the load significantly by performing all the dataset reading in C. Our program performs a mmap of the file into memory then, rather than splitting the file into many substrings, we use C string traversal to construct the graph. This results in much less allocation and writing to memory. In order to take advantage of the CPU's speed for executing this process, we chose to only offload to the MIC the functions that involved actual matrix calculations. We deliberately chose not to run the program natively on the MIC. For the GPU implementation we chose to copy the data to the device for computaion and consider the process complete after 50 iterations. This provides a faster implementation than copying data back to the host to check for convergence every iteration.

2.4 Parallelism

It should be noted that the original implementation for this program utilized pthreads and was significantly faster than most other implementations of the page rank algorithm. After analyzing the timings for the original program (please refer to table 1 on page 6), it was obvious that significant time savings could be made by improving the I/O functionality. For this, we transitioned to a C implementation rather than a python one, and utilized mmap for fast I/O as discussed in the previous section. Another area of concern was obviously the actual page rank calculation. Because the original implementation utilized pthreads, we decided that the best way to see computation improvement was to make a completely new implementation that utilized matrix math to solve the system of equations rather than solving each equation of the system one at a time as was the approach in the previous implementation. This allowed us to use the highly vectorized Intel Math Kernal Library and the Cuda Sparse Matrix Library by NVIDIA. It should be noted that the time improvement seen by implementing the algorithm discussed in section section 2.1 on page 1 was significant enough that no further parallelism was added becuase all major calculations were performed using matrix libraries.

2.5 Memory Layout

Because we are always performing a matrix-vector multiply, we did not need to alter the format of the matrix or vector in memory because they are both stored by default, and accessed in, row major order. As mentioned in section 2.1 on page 1, the shear size of the Live Journal dataset caused us to store our transition probability matrix in a sparse matrix representation. For simplicity and clarity we choose to implement the sparse matrix in Coordinate list form for the Math Kernal Library Implementation. However, because the CUDA library did not support this format, we also created the functionality to allow for the conversion to Compressed Sparse Row Matrix. We then loaded the tansisition probability matrices onto the device, already in this format, for the calculation to occur.

2.6 Stop conditions

Several different stop conditions are used in the various implementations of the algorithms.

3 Results

3.1 NCAA Football

This dataset must be converted to the snap format using `reformat.py` as described in the README (Appendix A). The results are not expected to be the same as the lab 3 results, due to the algorithm changes discussed in section 2.2 on page 3.

3.2 State Borders

This dataset must be converted to the snap format using `reformat.py` as described in the README (Appendix A).

3.3 Karate

This dataset must be converted to the snap format using `reformat.py` as described in the README (Appendix A).

3.4 Dolphins

This dataset must be converted to the snap format using `reformat.py` as described in the README (Appendix A).

3.5 Les Miserables

This dataset must be converted to the snap format using `reformat.py` as described in the README (Appendix A).

3.6 Political Blogs

This dataset must be converted to the snap format using `reformat.py` as described in the README (Appendix A). The results are not expected to be the same as the lab 3 results, due to the algorithm changes discussed in section 2.2 on page 3.

3.7 Wiki Vote

The results are not expected to be the same as the lab 3 results, due to the algorithm changes discussed in section 2.2 on page 3.

3.8 p2p-Gnutella05

The results are not expected to be the same as the lab 3 results, due to the algorithm changes discussed in section 2.2 on page 3.

3.9 SlashdotZoo

This dataset must be converted from Unix (LF only) to DOS (CRLF) format before running. The results are not expected to be the same as the lab 3 results, due to the algorithm changes discussed in section 2.2 on page 3.

3.10 Amazon

The results are not expected to be the same as the lab 3 results, due to the algorithm changes discussed in section 2.2 on page 3.

3.11 LiveJournal

The results are not expected to be the same as the lab 3 results, due to the algorithm changes discussed in section 2.2 on page 3. Further, this dataset only completes processing successfully on the CPU and GPU implementation.

Dataset	Processor	Read time (s)	Process time (s)	Iterations
WIKI	mic-off	0.01	0.3	14
	cpu	0.01	0.001	31
	cuda	0.018	0.043	50
	orig	1.849	0.14	25
P2P-GNUTELLA05	mic-off	0.003	0.294	7
	cpu	0.01	0.001	31
	cuda	0.015	0.043	50
	orig	0.552	0.023	12
AMAZON0505	mic-off	0.398	0.735	26
	cpu	0.404	3.103	138
	cuda	0.519	1.339	50
	orig	39.719	52.776	169
Live Journal	mic-off	N/A	N/A	N/A
	cpu	8.897	54.819	108
	cuda	8.782	39.774	50
	orig	582.99	627.915	103
NCAA Football	mic-off	0	0.307	23
	cpu	0	0	50
	cuda	0	0.141	50
	orig	0.041	0.006	44
Stateborders	mic-off	0	0.302	47
	cpu	0	0	136
	cuda	0	0.029	50
	orig	0.005	0.009	137
Karate	mic-off	0	0.449	21
	cpu	0	0	51
	cuda	0	0.03	50
	orig	0.004	0.009	203
Dolphins	mic-off	0	0.297	19
	cpu	0	0	94
	cuda	0	0.029	50
	orig	0.01	0.011	183
Les Miserables	mic-off	0	0.312	37
	cpu	0	0	73
	cuda	0	0.031	50
	orig	0.008	0.011	198
Political Blogs	mic-off	0.002	0.319	58
	cpu	0.002	0.008	193
	cuda	0.002	0.201	50
	orig	0.263	0.136	154

Table 1: Performance Data

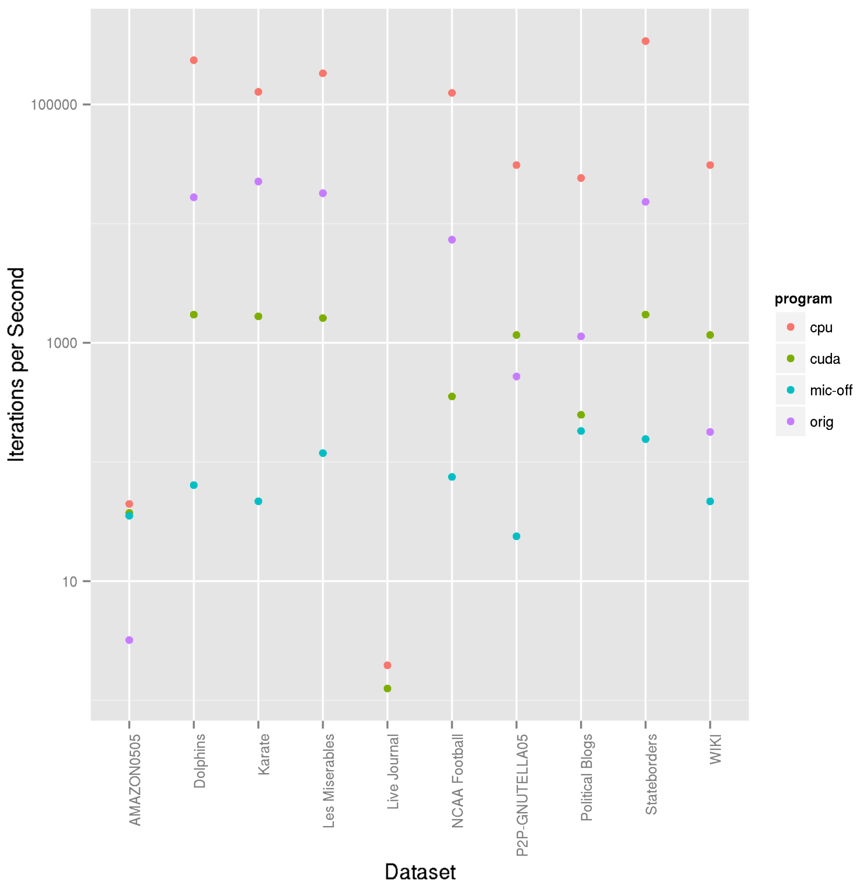

Figure 1: Graph of Performance

4 Performance Summary

The data in table 1 on page 6 shows the effects that the new implementations had on the overall program timing. In all cases it can be seen that the read times are drastically improved. The general trend for process times is also summarized in fig. 1 on page 7. It should be noted that the timings for the original implementation were not gathered on the same machines as the CPU, GPU and MIC data. We chose to consider iterations per second because the number of iterations varied between implementations. It should be noted that the smaller data sets don't show great improvement because the cost of offloading the calculation outweighs the benefit of pthreads on the CPU. We can see that the size of the amazon and Live Journal datasets is large enough to benefit from the cost of offloading the calculations. Note that the LiveJournal numbers for the original version are from a 4-core machine, not the same machine as any of the other numbers.

A README

A.1 Dataset Format

The ranker scripts assume the graph is directed in the following form.

```
# a comment
# a comment
# Nodes: <node count> Edges: <edge count>
# a comment
node_from<tab>node_to
node_from<tab>node_to
node_from<tab>node_to
```

All datasets not in the above format must be converted to this format using the conversion script `reformat.py`.

To convert a dataset, run

```
python3 reformat.py source_file.csv > output_filename.txt
```

This will create a file compatible with the C code, along with a Python pickle named `source_file.csv.pickle` which contains mappings from the node numbers in the new file to the node names in the old file, along with the file load time in the Python code.

Next, run the ranker script as described below, then convert the results back to the named-node format with

```
python3 unformat.py source_file.csv.pickle \
  < output.out > results.txt
```

A.2 Compile

To build for CPU:

```
cd src
make
```

To build for MIC offload:

```
cd src
make offload
```

To build for GPU:

```
cd src/cuda
make
```

A.3 Compute Page Rank

```
cd src
./pageRank <path/to/snap-formatted-dataset.csv>
```

For GPU:

```
cd src/cuda
./pageRank <path/to/snap-formatted-dataset.csv>
```

To compute the page rank on lab3 so it will match that from the MIC offload:

```
ranker --dval 0.95  --epsilon 0.0000001 --fmt <format> \
  --no-scale <path/to/data.txt>
```

You will need to use the correct format for the data you're working with.

See the lab 3 documentation for more on how to run lab 3.