Final Project

A Parallel Algorithm for Finding Sub-graph Isomorphism

Ashish Sharma, Santosh Bahir, Sushant Narsale, Unmil Tambe

Department of Computer Science, Johns Hopkins University
Index

1. Abstract 3
2. Overview 4
3. Parallel Design 6
4. Evaluation 10
5. Discussion 13
6. References 14
Abstract

There are a number of algorithms and programs present to find Subgraph Isomorphism. For real world applications of graphs like imaging, biocomputing, information retrieval, the graph often consists of large number of nodes. Algorithms for finding subgraph isomorphism for graphs of these scales are processor intensive and time consuming. In this project, we attempt to solve this problem by parallelizing a popular algorithm for finding subgraph isomorphism by Ullmann. A serial implementation of the algorithm is written and benchmarked. The algorithm is studied to find parts that can be parallelized and a new parallel implementation is developed to be run on Google's Map-Reduce. The comparison of the efficiency of both methods is discussed at the end of this report.
Overview

Introduction

Graphs are a commonly used data structure in computer science. There are many real world problems which are represented in term of graphs. This leads to the various sorts of manipulations of graphs such as searching, adding and deleting the elements etc. One of the few interesting and challenging problems in graph theory is comparing the graphs. This project handles one such problem – finding Subgraph Isomorphism.

Subgraph Isomorphism

Subgraph isomorphism is the problem of determining if one graph is present within another graph i.e. if some portion of a large graph is identical in structure to some other smaller graph.

More formally,

“Given two graphs, G₁ and G₂ there is subgraph isomorphism from G₁ to G₂ if there exists a subgraph S ⊆ G₂ such that G₁ and S are isomorphic”

An isomorphic graph can be defined as follows:

“Let G₁ and G₂ be two graphs and M₁ and M₂ be their corresponding adjacency matrices. Then, G₁ and G₂ are isomorphic if there exists a permutation matrix P for the graph G₁ such that

\[ M₂ = P \cdot M₁ \cdot P^T \]

Where, \( P^T \) = Transpose of matrix P.”

Importance of Subgraph Isomorphism Problem

The applications of subgraph isomorphism are enormous. It is used in pattern recognition, computer vision, information retrieval, scene analysis, image processing, graph grammar and graph transformation, biocomputing, finding network topology over large networks.

The vastness and diversity of the domains where it is used make finding subgraph isomorphism a problem worthwhile to solve. Furthermore, parallelizing the solution will reduce the run time.
Existing Algorithm to find Subgraph Isomorphism

Various algorithms are present for finding subgraph isomorphism. The common idea in most of them is similar to the basic algorithm described by Ullmann. Graphs are represented in the the form of adjacency matrices. A permutation matrix is then built one step at a time to generate all possible subgraphs of the large matrix. At each step the resulting matrix representing a subgraph of the large matrix is compared to the smaller graph being searched to find a match.

This process is computationally intensive and time consuming. The worst case, the time complexity of this algorithm is:

$$O(I^M M^2)$$

where, $M$ = Number of vertices in the main graph

$I$ = Number of vertices in the sub-graph

Main Thesis

The worst case complexity of Ullmann’s algorithm for subgraph isomorphism is $O(I^M M^2)$. A serial implementation of this algorithm when used for large graphs of the order of hundreds of nodes takes infeasible amount of time for real world applications. In this project, we parallelize the algorithm in an attempt to reduce the worst case time complexity. The algorithm is computationally intensive. By distributing the work among multiple compute nodes we aim to increase the scalability and performance.

Experimental Methods Used to Evaluate Thesis

A serial version of Ullmann’s algorithm was implemented and later modified to parallelize it. The parallel version was designed to run on Google’s MapReduce Framework. A comparison was made for the time take to run the serial and the parallel versions of the algorithm for a commonly available set of large graphs used for testing and benchmarking graph related programs.

Findings

The worst case time complexity of Ullmann’s algorithm is reduced to $O(I^M M)$. For large graphs this is a significant improvement. Moreover, since the algorithm is computationally intensive, there is a measurable increase in scalability of the problem in the parallel version due to division of the computational steps into multiple nodes.
Parallel Design

The problem of finding subgraph isomorphism is handled by Ullmann’s algorithm. We first describe here the original serial version of the algorithm, and then the modified parallelized version.

Ullmann’s Algorithm

The Ullmann’s algorithm uses permutation matrices to find isomorphism between two graphs (subgraph). Permutation matrices are generated for the main graph and then compared with the subgraph to find a match. The algorithm is as follows.

Inputs:
1. The larger graph $G = (V; E; \mu; \Box; L_v; L_e)$
2. The smaller graph to find a match for $G_I = (V_I; E_I; I; I; L_v; L_e)$

Output: Permutation matrix for each match found

Algorithm steps:
1. Let $P = (p_{ij})$ be a $n \times n$ permutation matrix, $n = |V|$, $m = |V_I|$ and $M$ and $M_I$ denote the adjacency matrices of $G$ and $G_I$, respectively.
2. call Backtrack($M; M_I; P; 1$)
3. procedure Backtrack(adjacency matrix $M$, adjacency matrix $M_I$, permutation matrix $P$, counter $k$)
   (a) if $k > m$ then $P$ represents a subgraph isomorphism from $G_I$ to $G$. Output $P$ and return.
   (b) for all $i = 1$ to $n$
      i. set $p_{ki} = 1$ and for all $j = i$ set $p_{ij} = 0$
      ii. if $S_{k,k}(M) = S_{k,k}(P) M (S_{k,k}(P))^T$ then
          call Backtrack($M, M_I, P, k+1$)
The inputs to this algorithm are the 2 graphs in the form of their adjacency matrices – the main graph / model graph and the sub-graph / target / input graph. The algorithm outputs the permutation matrices for which a match was found.

The backtrack method recursively builds the permutation matrix element by element, checking for a match with the input graph at each step.

**Design Patterns for Parallel Implementation**

For implementing the parallel algorithm, the original serial version was examined for identifying design patterns as follows:

a) Indentifying Concurrency

In the serial algorithm, the backtrack procedure is called at step 2 which is called with k = 1. k is the last parameter to the backtrack procedure. In backtrack, at step b, this procedure is called recursively with k = k + 1. All these calls with k = k + 1 made in the loop are independent of each other. Here n is the total number of vertices in the main graph. Since these n recursive calls are independent, we can execute them concurrently.

b) Algorithmic Structure

To exploit the concurrency mentioned above, there are a few changes to the serial version of the algorithm. Before dividing the input among different processing entities, we execute the first set of backtrack procedure calls, ie, we create N permutation matrices of dimensions N x N and one distinct column of first row is set to one and rest to zero.

Now instead of calling Backtrack procedure with k = 1, we will start executing it with iteration 2 on N different processing entities with one if the N permutation matrices calculated as mentioned above.

In short, the algorithm would be:

1) First step is the same

2) Set N permutation matrices, for each of which the first distinct column is set to 1

3) Call backtrack with k = 2

The 3rd step will be executed on different nodes and will do actual computation.
c) Supporting Structures

   i) Program Structure:
      The pattern used for program structure is loop parallelism as we are splitting the
      computation intensive loop for execution on independent compute nodes.

   ii) Data structures:
      The data structures used are of type distributed array. As we are distributing the copies of
      permutation matrix. A distributed cache is also used for passing the adjacency matrices
      for main graph and input graph for making them available to each mapper process.

Implementation mechanism

   a) UE Management:
      The Map-Reduce Framework of Hadoop inherently supports multiple UE’s and transparently
      manages their creation and destruction.

   b) Synchronization:
      For the two steps of execution of the parallel algorithm, two mapper processes are chained
      together. The output of first mapper is passed on as the input to the second mapper. Unless
      first mapper finishes execution, the second does not start which ensures the required
      synchronization.

      The first mapper calculates the dimensions (N) from the main graph which is its input – this
      file is read from the distributed cache. It then sets the distinct columns of first row to 1. This
      is the output of first mapper. Internally, all these permutation matrices are kept in file each
      one of which is taken by the second mapper for further calculation and comparison to find
      matches with the input graph.

   c) Communication:
      Communication between the processes is handled internally by Map-Reduce. The format for
      communication between the two mappers and the identity reducer is specified in the code.
Implementation in Map-Reduce

The algorithm is divided into two steps for parallel execution of the steps involved in generation of permutation matrix. The size of the permutation matrix is computed by the first mapper by looking at the dimensions of the adjacency matrix of the main graph that is read from the distributed cache. The first row of all permutation matrices are computed with 1 at distinct places. The output is passed to the second mapper with the help of ChainMapper class introduced in Hadoop 0.19.0. The second mapper computes the further elements of the permutation matrix, and at each step compares the resulting matrix after calculation to the input matrix, which again it reads from the distributed cache.

The output of the second mapper is the number of matches and for each match, the permutation matrix that led to the match is printed. This output is passed through an identity reducer.

Fig: Flow Diagram of the MapReduce Process
Evaluation

Method for evaluation

The serial code was written in Java for the original Ullmann’s algorithm. This code was benchmarked using the standard library of graphs for finding subgraph isomorphism. The serial code was then modified as discussed above to run on the Hadoop MapReduce framework and run on a cluster and benchmarked against the same graph library.

Properties of system used for evaluation

A Hadoop multi-node cluster was created using four 1.8 GHz dual core laptops for running MapReduce. A distributed cache was setup in Map-Reduce for sharing the common input files that are basically used as read-only by the two mappers. These files contain the adjacency matrices for the main and the input graphs. An intermediate file is created internally for the passing the output from the first to the second mapper process. This file contains a different permutation matrix on each line. Each of the second stage chained mapper process gets a single line of this file as input, i.e., a different permutation matrix for doing further calculations and comparisons. The reducer used is an identity reducer, which pipes the output of the second mapper process to the final output.

Thus, in the system, there are 4 compute nodes with the same processor and memory configuration.

Experiments

A standard graph library (from http://amalfi.dis.unina.it/graph/) was used for benchmarking the serial and parallel versions of the program. The graph library consists of pairs of graphs – a main graph and a smaller graph which is present in and is to be searched for in the main graph. A result file is provided for checking correctness of the program. It contains the number of matches found for each graph pair. The graphs range from 20 to 1000 nodes.

The serial version of the code and the parallel version were both tested on the same set of graphs. The execution time was noted and compared, and the results are as below.

Results

The time taken to execute the serial and the parallel code is listed below for different number of graph nodes.
The time taken by the algorithm is dependent only on the number of nodes in the main graph since the size of the permutation matrix is dependent only on the dimensions of the main graph. The Y axis represents the same. As we can observe in the graph, the time taken by the parallel code is initially slightly greater than that of the serial code run on a single node. This is due to the overheads introduced in communication, synchronization and the setup of the MapReduce framework. The real gain in processing time shows up only for larger number of nodes of the graph.

We can also observe that as the increase in time is almost constant till a threshold, in this case 80 nodes. After that there is a massive and sudden increase in time for the serial code. The parallel algorithm comes into effect at this stage by handling the large number of computations in parallel on multiple nodes.
Till 80 nodes in the graph, the time taken by the parallel code is greater than the serial code by a constant magnitude. The advantage of multiple nodes working simultaneously is visible significantly towards larger nodes. The reduction in time by the parallel environment is more than a factor of 2.

Thus the speedup obtained in running the algorithm for large number of graph nodes is significant enough to make it worthwhile to accept the overhead involved in the initial stages for smaller number of nodes, especially since this algorithm is mostly used in applications where the graph size is usually large.
**Discussion**

The results support the thesis to a great extent with respect to the performance of the parallel algorithm for large number of nodes as compared to that of the original Ullmann’s algorithm. This is visible largely from the results of the experiment to compare the performance of the two implementations. For larger number of nodes, the parallel implementation is faster than the serial one almost by a factor of 2. The thesis claimed that on paper, the time complexity of the algorithm was reduced from $O(I^M M^2)$ to $O(I^M M)$. This is proved to a large extent by the results of the experiment.

**Future work**

The scalability of this algorithm remains to be verified due to limited computing resources. Also, due to the same reason, the gain in performance for an even larger number of nodes is not noted. We expect the graph to show the same trend for larger number of nodes as compared to the serial version. But after a threshold number of nodes, the graph will be sublinear due to the overheads induced by the communication, synchronization along with the memory overhead.

**Conclusion**

The project gave an insight into the patterns that are commonly observed in algorithms that can be exploited to make use of the parallel computing resources that are becoming increasingly available. The results of the experiments were as expected and as studied in the Parallel Programming theory. The speedup was obtained only after a certain threshold where the intensive computation overwhelms the serial algorithm, while becoming more and more natural to the parallel paradigm with increase in size of the problem set.
References

1. Subgraph Isomorphism in Polynomial Time – B.T. Messmer and H. Bunke, University of Bern, Neubrückstr. 10, Bern, Switzerland