Counting Triangles in Parallel
An Application of Pipelining

Julián Aráoz[1,2] and Cristina Zoltan[1,2]

[1] UNICyT, Panamá
[2] Universitat Politècnica de Catalunya, Espanya

Abstract. The usual approach to producing a parallel solution to a computational problem is to find a way to use the *Divide & Conquer* paradigm in order to have processors acting on their own data so they can all be scheduled in parallel. MapReduce is an example of this approach. We present an alternative program schema that can exploit dynamic pipeline parallelism without having to deal with replication factors. We present the schema through an example: counting triangles in graphs.

Keywords: Pipelining, triangle-counting, parallel algorithms, Data Streams

1 Introduction

With the emergence of commodity multicore architectures, exploiting tightly-coupled parallelism has become increasingly important. Most parallelization efforts are addressed to applications that compute with large amounts of data in memory and in general have a regular behavior.

Frameworks that implement MapReduce are a great success. This may be in part because having a framework helps in testing the goodness of the solution, but also because implementations deal with errors and because the programmer needn't be concerned with deployment in parallel architectures as the implementation takes care of this.

We present an alternative program schema that can exploit dynamic pipeline parallelism without having to deal with replication factors. Applications using a sequence of processes working in parallel without requiring large amounts of local memory are suitable for both multicore and distributed architectures.

We present the schema through an example: counting triangles in graphs. Counting triangles, having as input an unordered sequence of edges, is only one example of the type of problem solvable with this algorithmic schema. Other graph problems include: Elimination of duplicate edges, finding connected components, and finding small circuits, among others.

This is an extended abstract of our paper [1] in which can be found a small example of our algorithm using a NiMo implementation together with a formal description and proof of this type of algorithm.

2 Approaches to solving the problem

Various approaches to parallelizing algorithms for exact triangle counting are found in the literature. [7] uses the MapReduce framework and [3] the hypercube architecture. Both approaches use hashing functions on the vertex names to partition the graph. Both have relatively high replication rate.

A divide and conquer approach is given in [2], using the BSP model to synchronize the parallel workers and MPI to implement the algorithm.

From the input graph we construct a collection of subgraphs, of size equal to the number of processors, such that triangles can be counted by counting on each subgraph and accumulating the results. The improvement over [7] is due to the fact that the latter generates a huge volume of intermediate data, consisting of all possible 2-paths centered at each node. Our improvement over [2] is that we do not collect in a single machine all nodes adjacent to a core (also responsible or dominating) node, but only those that have not been collected by some other responsible node. Furthermore, rather than doing preconditioning in order to balance the workload, our dynamic scheduler is able to achieve this based on the size of a set of adjacent nodes to each responsible node. In [4] a sequential algorithm is presented where a dominating set of nodes is constructed for dividing the graph into subgraphs. The constructed set is different from ours, because it is based on the adjacency presentation of the input graph. In [4] the use of the subgraphs results in a sequential iterative algorithm.

3 Streaming Model of Computation

In [5] we find a good survey of algorithms based on the streaming model of computation.

We use the semi-streaming model of computation, where the input graph, $G = (V, E)$, is presented as a stream of edges (in any order), and the storage space of an algorithm is bounded by $\Theta(n\ polylog\ n)$.

We are particularly interested in algorithms that use only one pass over the input, but for problems where this is probably insufficient, we also look at algorithms using constant, or in some cases, logarithmically many passes.

One measure of amorphous data-parallelism is the number of active nodes (fireable processes) that can be processed in parallel at each step of the algorithm for a given input, assuming that:

- There is an unbounded number of processors,
- An activity takes one time step to execute,
- The system has perfect knowledge of neighborhood and ordering constraints so that it only executes activities that can complete successfully, and
- A maximal set of activities, subject to neighborhood and ordering constraints, is executed at each step.

This is called the *available parallelism* at each step. A function plot showing the available parallelism at each step of execution of an irregular algorithm for a given input is called a *parallelism profile*. This gives an upper bound on obtainable parallelism for the given solution.

4 The Algorithm

4.1 Notation

The input graph is denoted by $G = (V, E)$, where V and E are the sets of vertices (nodes) and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices labeled using labels that can be compared for equality. We use the words node and vertex interchangeably. We assume that the input graph is undirected, without loops or multiple edges [3]. If $(u, v) \in E$, we say u and v are adjacent to each other. The set of all nodes adjacent to $v \in V$ is denoted by N_v, i.e., $N_v = \{u \in V | (u, v) \in E\}$. A triangle is a set of three nodes $u, v, w \in V$ such that there is an edge between each pair of these three nodes, i.e., $(u, v), (v, w), (w, u) \in E$. The number of triangles containing node v (in other words, triangles incident on v) is denoted by T_v. Notice that the number of triangles containing node v is the same as the number of edges among the set of nodes adjacent to v, i.e., $T_v = |\{(u, w) \in E | u, w \in N_v\}|$.

4.2 Informal Algorithm presentation

We assume the input is a large graph given by an unordered enumeration of its edges. We assume a non-oriented graph with no duplicated edges.

We use a two-round schema, first of all partitioning the graph building length two paths, but only one of the three corresponding to a triangle, then identifying and counting the triangles. We do not need to use a special data structure.

In [7] uses also the. The main difference is the way the possible two path are identified and the program structure, also it compute all two paths.

But use the same principle of a single node being responsible for making sure the triangle gets counted. In [6] this is obtained from knowledge of the degree of each node. Dealing with graphs large enough to not fit in memory, this additional knowledge requires an further traversal of the edges of the graph which is not needed in our approach.

The algorithm we present is implemented in NiMo as a sequence of actors (processes), communicating using three unbounded channels. Processes change their role (mutate their behavior) when enough knowledge has been collected. Initially there is a single actor.

An actor's first role is to acquire an edge and become a process that is "responsible" for this first node on the edge. The responsible actor receiving an edge will collect in its memory the node if the edge is adjacent to its responsible node. If the received edge is not adjacent to the responsible node, the actor passes the edge to the next actor in the sequence. If there is no next actor, a newly created actor is added to the sequence that will process the edge.

[3] Oriented edges are a special case of the one treated here.

When there are no more edges in the first input, each actor changes role, once again receiving the sequence of edges using the second input. The new role is to count a triangle whenever the incoming edge forms a triangle with the "responsible " node and two adjacent ones i.e. both ends of the edge are adjacent to the "responsible". Whenever a process has received all the edges, it reads the third input, that carries the total number of triangles identified so far by its provider. It adds its own count, and passes its triangle count to its neighbor and dies. The algorithm ends with no live processes and a single result: the total number of triangles.

The I/O complexity is $O(m)$, The total memory needed is m and the internal memory of each processor is bounded for the maximal degree of the nodes which is bound by n. The number of processes created is the number of responsible nodes and is bounded by $n-1$.

5 Concluding Remarks

We have used the triangle counting problem on a graph described as a sequence of edges as an example of a dynamic pipeline solution of problems. In the example we see that a large amount of processors can be exploited, but the algorithm is correct even in single processor semantics using a dynamic scheduler. The solution can be regarded as a bucket-sorting method that separates the input set into disjoint classes, but in our case the classes are created on the fly. Processes in the pipeline change behavior as soon as enough data has been consumed. In this approach data input is shared with computation, without the barriers present in the BSP model of computation.

References

1. Julián Aráoz and Cristina Zoltan. Parallel triangles counting using pipelining. `http://arxiv.org/pdf/1510.03354.pdf.`, 2015.
2. Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22Nd ACM International Conference on Conference on Information &#38; Knowledge Management*, CIKM '13, pages 529–538, New York, NY, USA, 2013. ACM.
3. Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 63–78, 2015.
4. Shumo Chu and James Cheng. Triangle listing in massive networks. *TKDD*, 6(4):17, 2012.
5. Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.
6. Thomas Schank. *Algorithmic Aspects of Triangle-Based Network Analysis.* PhD thesis, Universität Karlsruhe, 2007.
7. Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM.