École Polytechnique

Traitement des données massives

INF 442

Finding Strongly Connected Components

X2016
Eduardo Caldas
Gabriel Oliveira Martins

May 30, 2018

1 Introduction

The problem of finding strongly connected components in a graph is one of great interest, pedagogically, to show the power of DFS, and in the industry as it can reduce the complexity of various problem, by considering an SCC as an unique node. A sub-graph of G is a strongly connected component of G if it is strongly connected and maximal with this property. Any graph has a unique decomposition into strongly connected components, which partitions the set of nodes. We developed in this project a sequential and a parallel algorithm to find SCC's as well as various utils pre-treat and generate graphs, specially ER-Graphs

2 Sequential Algorithm

2.1 The choice of the algorithm

We chose to implement the Kosaraju's algorithm. This algorithm has a much simpler proof and the same time complexity, when compared to Tarjan's. Since reading the input adds an overhead of $O(m)$ already, we decided that the constant runtime difference between the two wasn't important.

2.2 Brief Description

This algorithm uses properties of the topological sorting of a graph to find its strongly connected components:

1. Create an empty stack 'S' and do post-fix DFS traversal of a graph. But, in this case, for a call `DFS(v)` push 'v' to 'S' in the end of the call, after exploring all its neighbors.

2. Generate the transposed graph G', with reverse-order edges.

3. Perform another DFS, this time following the order on the stack, the elements reached in each DFS are the SCC's.

2.3 Complexity analysis

In the brief description above we have two major operations, 'DFS' and 'reversal of edges'. DFS has time complexity of $O(n + m)$ and reversal of

edges of $O(m)$. The total complexity is $O(n+m)$. As stated in the project description this is optimal. Also considering that reading the input demands $O(m)$ already, this algorithm is already very fast.

3 Erdõs-Rényi Random Graphs

3.1 Brief Description

We call an Erdõs-Rényi Graph, and note it $G(n,p)$, a graph with n nodes such that for each ordered pair of nodes (u,v), $u \neq v$ with probability p there exists and edge $u \to v$.

3.2 Link with connected components

These graphs have, asymptomatically, some interesting properties regarding connected components [4], for instance:

- If $np < 1$, then a graph in $G(n,p)$ will almost surely have no connected components of size larger than $O(log(n))$.
- If $np = 1$, then a graph in $G(n,p)$ will almost surely have a largest component whose size is of order $n^{2/3}$.
- If $np \to c > 1$, where c is a constant, then a graph in $G(n,p)$ will almost surely have a unique giant component containing a positive fraction of the vertices. No other component will contain more than $O(log(n))$ vertices.
- If $p < \frac{(1-\varepsilon)\ln n}{n}$, then a graph in $G(n,p)$ will almost surely contain isolated vertices, and thus be disconnected.
- If $p > \frac{(1+\varepsilon)\ln n}{n}$, then a graph in $G(n,p)$ will almost surely be connected.

Thus in addition to using them to test our algorithms we can also simulate these properties.

3.3 Sequential

The implementation of this generator module was quite straight-forward. We just did $n(n-1)/2$ calls to `rand()`, one for each possible edge, and if it was smaller than p, we created the corresponding edge.

3.4 Parallel

Although showing an $O(n^2)$ complexity, this generator is highly parallelizable, since the probability of existence of an edge is independent of the rest of the graph. We used MPI to parallelize it, by sharing the work among processors by groups of origin nodes.

4 Parallel Algorithm

4.1 Forward and Backward Trim

First of all, we perform a forward and backward trim in the digraph G in order to reduce its size by eliminating the vertices that have no ancestors or descendants and thus cannot be part of a larger SCC, as described it [2].

For the forward trim, we begin by finding all the vertices that have no ancestors and eliminating them from the set of vertices and adding them to a queue. While the queue is not empty, we remove a vertex from the queue and delete all of its out-edges, and then if the vertex reached by each edge has no ancestor we add it to the queue.

The backward trim is analogous but changing ancestors with descendants and out-edges with in-edges.

4.2 Divide-and-Conquer Strong Components (DCSC)

We used the *Divide-and-Conquer Strong Components* (DCSC) algorithm described in [1, 2]. We pick a vertex v at random in the set of vertices and then find $\mathrm{Succ}_G(v)$ and $\mathrm{Pred}_G(v)$, respectively the sets of successor and predecessor vertices of v, using a breadth-first search. Then we partition the nodes of the digraph G into four sets:

- $S_1 = \mathrm{Succ}_G(v) \cap \mathrm{Pred}_G(v)$
- $S_2 = \mathrm{Succ}_G(v) \setminus \mathrm{Pred}_G(v)$
- $S_3 = \mathrm{Pred}_G(v) \setminus \mathrm{Succ}_G(v)$
- $S_4 = V \setminus (\mathrm{Succ}_G(v) \cup \mathrm{Pred}_G(v))$

The set S_1 is the SCC that contains v. We proceed by applying the algorithm recursively in the sets S_2, S_3 and S_4 in a parallel, until the set of vertices is composed of a unique vertex or is empty.

4.3 Parallelization with MPI

In order to perform the parallelization with MPI, we have a *root* processor that is the responsible to allocate the tasks to the others processors and group all the connected components. In the `main` function, all the processors call the function `DCSC` to manage the allocation of tasks and vertices.

The `root` processor has a stack to store the processor that are available and a queue to store the graphs waiting to be analyzed. At first it puts all the other processors in the stack and the original graph after the Trim in the queue. Then it starts a loop where it will allocates as many set of vertices in the queue as possible to the available processors, with a `MPI_Send`, or if it's a set with a single vertex, add it to the list of SCC. Still in the loop, the root waits for a message from one of the the other processors with a `MPI_Recv`. This message can be of three different types, which is indicated by its tag.

- $tag = 0$: The processor p finished its job and is now available. The `root` adds it to the stack with all the other available processor. If all the processors are then available and there queue is empty, it will exit the loop. It will then inform all the others processors that the job is done through a `MPI_Send` with a $tag = 0$.
- $tag = 1$: The `root` receives a set of vertices from the processor p that will be added to the queue.
- $tag = 2$: The root receives a SCC from the processor p and adds it to the list.

If the processor is not `root`, it will attend in a loop with `MPI_Recv` a message from the `root`. If it receives a message with $tag = 1$ and a set of vertices, it will call the function `DCSC_Rec` with this vector. If $tag = 0$, it means that the job is done and it exits the loop. The function `DCSC_Rec` executes the algorithm of `DCSC` described in the last section. The difference consists in the recurrence. After finding S_1, S_2, S_3 and S_4 it sends a message to the `root` with S_1 and $tag = 2$. It sends the S_2 and S_3 with $tag = 1$ or $tag = 2$, depending on its size, to be managed by the `root`. Finally, it calls recursively `DCSC_Rec` with S_4 in the same processor.

References

[1] Lisa K. Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In: José Rolim et al. (eds.) *Proceedings of the 15 IPDPS 2000 Workshops*, Lecture Notes in Computer Science vol. 1800, Springer, Heidelberg, 2000, pp. 505–511.

[2] William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing* 65(8):901–910, 2005.

[3] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. ACM.

[4] Erdős, P.; Rényi, A. (1960). On the evolution of random graphs . In *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*