High Performance Computing
Homework assignment 1.1

Luca Venturi

February 9, 2017

An efficient and scalable algorithmic method for generating large-scale random graphs

The paper [1] presents a new algorithm for generating large-scale random graphs. Random graphs simulations are used as models in the study of behavior of complex system, such as the Internet, biological networks and social networks. As the complex systems are growing larger and larger, it requires the generation of massive random networks efficiently. Generation of such random graphs requires efficient algorithms, both in space and time requirements. Recently, some efficient sequential and parallel algorithms have been developed. Although, while these algorithms can generate graphs with millions of vertices in a reasonable time, generating graphs with billions of vertices can take an undesirably long time and a prohibitive large memory requirement. The authors propose a novel method, based on grouping the vertices by their degree, that guarantees space and time efficient algorithms. They then describe an efficient parallelization of the proposed algorithm and develop a rigorous computation of the computational cost. With the proposed parallel algorithm, generating a power-law network with 250 billion edges (and ~ 1 billion vertices) takes only 12 seconds using 1024 processors.

The DG algorithm The following algorithm generates a random graph with a given degree distribution, i.e., a random graph with a fixed number of vertices n of expected degree $\{w_i\}_{1 \le i \le n}$. Many models have been proposed to generate random graphs given a desired degree sequence. The DG algorithm proposed in the paper [1] is based on the CL model. In this model any pair of vertices i, j is connected with probability $p_{ij} = \frac{w_i w_j}{S}$, where $S = \sum_k w_k$; the expected degree of a vertex i converges to w_i for large graphs. The basic idea of the DG algorithm is to work independently on groups of vertices with same expected degrees. Let $d_1, \dots, d_\Lambda$ be the distinct expected degrees and n_i be the number of vertices with expected degree d_i. The vertices are then grouped based on their expected degrees: V_i is the group of vertices with expected degree d_i. Now there can be two type of edges: *Intra edges*, if both vertices belong to the same group V_i, *Inter edges*, if the two vertices belong to two different groups V_i, V_j, $i \neq j$. In the first case, the edges are created by generating a geometric random variable of parameter $\frac{d_i^2}{S}$ for group V_i, while in the second case, the edges are created by generating a geometric random variable of parameter $\frac{d_i d_j}{S}$ for groups V_i, V_j. The geometric variable represents the label of selected edges and is generated m_{ij} times, with m_{ij} being the number of generated edges in the group V_i, V_j. The vertices are labeled by indexes $1, \dots, n$ so that $V_i = \{\lambda_i, \dots, \lambda_{i+1} - 1\}$. In this way, for each group V_i it is only needed to store d_i, n_i and λ_i, leading to the space complexity of $O(\Lambda)$ (plus the space needed to store the selected edges $O(m)$, where m is the number of edges generated). On the other hand, this algorithm takes $O(\Lambda^2 + m)$ time. Since, to be applicable, the CL model requires $w_{max}^2 < S$ and since in case of integer expected

degrees one has $\Lambda < w_{max}$, the algorithm has a runtime of $O(m)$. In fact, in most of the real-world graphs Λ^2 is significantly smaller than m.

Parallelization of the DG algorithm To parallelize the above algorithm, it's important to assign to each of the available P processors the same computational load. The algorithm is already decomposed into many independent tasks $\mathcal{T}_{ij}$, where $\mathcal{T}_{ij}$ represents the task of generating edges between groups V_i and V_j. Relabeling these tasks as $\{\mathcal{T}_{ij}\}_{i\leq j} = \{\mathcal{Q}_x\}_x$, where every task $\mathcal{Q}_x$ has as a corresponding computational cost c_x, the easiest way would be to assign to each processor a certain number of tasks $\mathcal{Q}_x$. This can be done using the UCP algorithm, which partitions the tasks using the cumulative cost $\mathcal{C}_x = \sum_{k=1}^{x} c_k$, in such a way that the k-th processor is assigned tasks $\{\mathcal{Q}_{q_k}, \ldots, \mathcal{Q}_{q_{k+1}-1}\}$ according to the condition $\mathcal{C}_{q_k} < \frac{k}{P}\mathcal{C} \leq \mathcal{C}_{q_{k+1}-1}$, where $\mathcal{C} = \sum_x c_x$ is the total cost. However, a task in the UCP algorithm is non-divisible, i.e., the entire task is assigned to a processor; if some boundary tasks are very large, it can lead to imbalanced loads. Nevertheless, in our case these tasks can be broken in arbitrary smaller subtasks, considering the geometric variable in every task taking values on smaller intervals. The authors improved the UCP algorithm, allowing it to assign subtasks intead of entire tasks $\mathcal{Q}_x$. The parallel algorithm so obtained runs in $O(\frac{m+\Lambda^2}{P} + \Lambda + P)$ time.

Experimental results For experiments has been used an 81-node HPC cluster. Each node had two octa-core SandyBridge E5-2670 2.60GHz (3.3GHz Turbo) processors with 64GB RAM. They used MPICH2 (v1.7) for the algorithm. The parallelization performances of the algorithm shows to be better than the best known parallel algorithm (AK algorithm) both for strong and weak scaling. Strong scaling of a parallel algorithm shows its performances with the increasing number of processors while keeping the problem size fixed. The proposed algorithm shows a linear speedup four time faster than the AK algorithm. Also, the DG algorithm requires less memory ($O(\Lambda)$ instead of $O(n)$). The weak scaling measures the performance of a parallel algorithm when the input size per processor remains constant. DG algorithm shows to achieve very good weak scaling compared to AK algorithm, with almost constant runtime. Finally, different graphs are reported in the paper that show how load balancing is perfectly achieved.

Summarizing, paper [1] propose an algorithm for generating large-scale random graphs. Random graphs are used as models in the study of complex system. As the number of the vertices increases, algorithms to efficiently generate such random graphs are needed. The authors propose a new algorithm and describe how to parallelize it. This new method shows to be very efficient and scalable, with an almost perfect load balancing.

References

[1] Alam, M., Khan, M., Vullikanti, F., Marathe, M. (2016). An Efficient and Scalable Algorithmic Method for Generating Large-Scale Random Graphs. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, p. 32.