This paper talks about how MapReduce has emerged as an enabling technology for large-scale graph processing. However, existing best practices for MapReduce graph algorithms have significant shortcomings that limit performance, especially with respect to partitioning, serializing, and distributing the graph. In this paper, three design patterns are presented that address these issues and can be used to accelerate a large class of graph algorithms based on message passing, exemplified by PageRank. Experiments show that the application of these design patterns reduces the running time of PageRank on a web graph with 1.4 billion edges by 69%.

Large graphs are ubiquitous in today’s information-based society. Two examples include the hyperlink structure of the web spanning many billion of pages (commonly known as the web graph) and social networks that connect hundreds of millions of individuals. MapReduce provides an enabling technology for largescale graph processing. However, there appears to be a paucity of knowledge on designing scalable graph algorithms.

The paper provides an overview of the MapReduce programming model. Then discusses the class of graph algorithms that is the focus of this paper, exempliﬁed by PageRank. Paper also describes standard best practices for large-scale graph processing using MapReduce. Enhanced design patterns are presented for graph algorithms in MapReduce, and evaluated their performance on a large web graph with 1.4 billion links. Finally, summarization of ﬁndings and future directions for improvements.

The original MapReduce paper described several data intensive applications for the programming model, including word count, distributed grep, and inverted index construction, but unfortunately did not discuss graph algorithms. The ﬁrst reasonably detailed explanation of MapReduce graph algorithms can be traced to lecture slides and video recordings of courses sponsored by Google in 2007.2 The materials described implementations of parallel breath-ﬁrst search and PageRank: these have become the de facto best practices for MapReduce graph processing.

The paper discusses about Message passing. Computations in MapReduce operate on input key-value pairs; both keys and values can be primitive types (integers, strings, etc.) or arbitrarily complex records (e.g., tuples with nested structure). For graph processing, it is most natural to adopt an adjacency list representation: graphs are serialized into key-value pairs using the identiﬁer of the vertex as the key, and the record comprising the vertex’s structure as the value.

The paper presented details about Local Aggregation, In-Mapper Combining, Algorithm Optimizations, Schimmy, Range Partitioning.

Based on the results, there are some conclusions and future work. This work presents three design patterns broadly applicable to MapReduce graph algorithms. In-mapper combining can be used in any setting where a standard out-of-memory combiner can be used. Schimmy is useful because it obviates the need to reshuﬄe the graph structure at every iteration. Range partitioning requires only that vertices within a cluster, including approximate clusters derived from some attribute of the vertex, are consecutively labeled.

Future work remains to improve these enhanced design patterns even further. Partitioning could be improved to cluster based on actual graph topology. The challenge is that the graph will, in general, be too large to store entirely in memory, so clustering must be performed in a distributed fashion (perhaps with MapReduce). The schimmy design pattern could be improved by modifying Hadoop’s scheduling algorithm so that a particular key range is consistently assigned to the same machine, which would allow the graph structure to be merged from the local disk of the machine running the reducer (as opposed to a remote network read, as in the current design). This is challenging to implement without compromising the robustness of the system to hardware failures. Finally, the general problem of serialization addressed by in-mapper combining could be improved by storing more of the graph in memory between iterations.