1 Motivation

Graph analytics has been widely applied in many big data applications, such as social computation, web search and prediction systems. It is an important and computationally demanding class of data analytics. As shared memory systems support terabyte-sized main memory, they provide an opportunity to perform efficient graph analytics on a single machine. Graph analytics is characterized by frequent and fine-grain synchronization [2, 5, 1], which is addressed in part by shared memory systems. Recently, in order to achieve high performance, graph partitioning has been proposed to isolate memory accesses to specific parts of the graph data. However, performance is limited by load imbalance and poor memory locality, which originate in the irregular structure of small-world graphs. This work demonstrates how graph partitioning can be used to optimize (i) load balance, (ii) Non-Uniform Memory Access (NUMA) locality and (iii) temporal locality of graph partitioning in shared memory systems. The developed techniques are implemented in GraphGrind [3, 4], a new NUMA-aware shared memory graph analytics framework.

2 GraphGrind

GraphGrind contains hierarchical parallel decomposition of the computation, NUMA-aware data placement and code scheduling, balanced vertex-cut partitioning and adapting data structures and search direction to the size of the frontier.

2.1 Application Programming Interface

GraphGrind is compatible with the Ligra programming model [2]. It provides two data types: graphs and frontiers. A frontier is a subset of the vertices in a graph. The key functions apply operations to edges or vertices and calculate new frontiers in the process. It also provides two graph traversal methods, *backward* and *forward*. *Backward* traverses all destination vertices, obtains values from active source vertices by following incoming edges. *Forward* traverses all active source vertices, updates values to all destination vertices by following outgoing edges.

2.2 Partition Balancing Criterion

We show that heuristic edge-balanced partitioning to balance CPU load between graph partitions results in an imbalance in the number of vertices per partition. Thus, load imbalance exists between partitions, either for loops iterating over vertices, or for loops iterating over edges. To address this issue, we introduce a classification of graph algorithms to distinguish whether they algorithmically benefit from edge-balanced or vertex-balanced partitioning. This classification supports the adaptation of partitions to the characteristics of graph algorithms. Evaluation in GraphGrind-v1 [3], shows that this outperforms state-of-the-art graph analytics frameworks for shared memory including Ligra [2] by 1.46x on average, and Polymer [5] by 1.16x on average, using a variety of graph algorithms and datasets.

2.3 Locality of Partitioning

We demonstrate that increasing the number of graph partitions is effective to improve temporal locality due to smaller working sets [4]. However, the increasing number of partitions results in vertex replication in some graph data structures. GraphGrind-v2 resorts to using a graph layout that is immune to vertex replication and design an automatic graph traversal algorithm that extends the previously established forward or backward heuristic to a 3-way graph layout choice. This new algorithm furthermore depends upon the classification of graph algorithms in Section 2.2. These techniques achieve an average speedup of 1.79x over Ligra and 1.42x over Polymer. Figure 1 shows that GraphGrind-v2 [4] outperforms Ligra, Polymer and GraphGrind-v1 [3] by a significant margin.

2.4 NUMA-Aware Design

Graph partitions are distributing over NUMA domains. Frontiers represented as bitmaps and application-specific arrays storing attributes of vertices are allocated across the NUMA domains such that the attributes are stored on the NUMA domain that will update those values.
2.5 Graph Ordering

We present VEBO, a novel graph ordering algorithm to challenge the widely accepted heuristic to balance the number of edges per partition and minimize edge or vertex cut. This algorithm balances the number of edges per partition as well as the number of unique destinations of those edges. It optimally balances edges and vertices for graphs with a power-law degree distribution. Moreover, we show that the performance of graph ordering depends upon the characteristics of graph analytics frameworks, such as NUMA-awareness. GraphGrind using this graph ordering algorithm achieves an average speedup of 1.87x over Ligra and 1.51x over Polymer.

3 Contributions

GraphGrind differs from prior work by improving temporal locality and dealing with load imbalance through graph partitioning across NUMA domains, which controls the order of iteration over edges. It deals with load imbalance by tuning the graph partition method based on the characteristics of graph algorithm. Moreover, the state held for each vertex is updated by at most one thread in our design, obviating the need for synchronization and the use of costly hardware atomics. It removes the requirement that programmers decide whether an algorithm runs faster when traversing the graph in a backward or in a forward direction. In fact, we observe that the frontier density and memory access order are important factors.

We also demonstrate that edge-balanced partitioning alone does not create good balancing and that considering vertex-balance along with edge-balance improves load balance significantly. The analysis of performance of VEBO on Ligra, Polymer and GraphGrind has shown that the partitioning criteria should be selected in accordance with the graph processing system that will be used. In particular, Polymer and GraphGrind use partitioning to drive static scheduling of parallel loops. As such, load balance is very important for these systems. Ligra, on the other hand, is fully dynamically scheduled. In this case, it is more important to restructure the graph to improve memory locality.

References