# A Distributed Graph-Theoretic Framework for Automatic Parallelization in Multi-core Systems

Guixiang Ma [1] Yao Xiao [2] Theodore L. Willke [1] Nesreen K. Ahmed [1] Shahin Nazarian [2] Paul Bogdan [2]

**Abstract**

The rapid demand for memory and computational resources by the emerging complex applications requires multi-core parallel systems capable to scale the execution of these applications. In this paper, we propose a distributed graph-theoretic framework for automatic parallelization in multi-core systems, where the goal is to minimize the data communication while accounting for intrinsic functional interdependence and balancing the workloads among cores to improve the overall performance. Specifically, we design a general and flexible greedy-based vertex cut framework for partitioning LLVM IR graphs into clusters while taking into consideration the data communication and workload balance among clusters. Then, we map the clusters generated by the vertex cut algorithms onto a non-uniform memory access multi-core platform. Experimental results demonstrate that our proposed WB-Libra algorithm provides performance improvements of 1.56x and 1.86x over existing state-of-the-art approaches for 8 and 1024 clusters running on a multi-core platform, respectively.

## 1 Introduction

The massive and growing number of complex machine learning and big data analytics applications (Chen & Zhang, 2014) require highly efficient and scalable multicore platforms. Sequential programs running on single-core systems fail to provide the necessary performance. However, the parallel execution in multicores is not a cure-all solution because it can experience performance degradation due to load imbalance, synchronization overhead, and resource sharing contention. The parallel execution performance is determined by the worst execution time among spawned threads. Therefore, load imbalance can severely impact the overall performance. On the other hand, threads compete for the underlying shared hardware resources, which increases synchronization overhead if not properly handled.

Therefore, it is crucial to study how to optimize the parallel execution of applications in multi-core systems. The recent related work has focused on fine-grained parallelism and various task-to-core mapping strategies for minimizing the execution overhead (e.g., run-time, communication cost) on multi-core systems. For example, (Hendrickson & Leland, 1995a) proposed graph partitioning algorithms based on spectral graph theory to partition coarse-grained dataflow graphs into parallel clusters for mapping large problems onto different nodes while balancing the computational loads. (Devine et al., 2006) develops hypergraph partitioning algorithms to better model communication requirements and represent asymmetric problems to divide computations into clusters. Other related research (Murray et al., 2013; Yu et al., 2008; Murray et al., 2011) designed various systems for general-purpose distributed data-parallel computing.

Despite the large number of works in this area (Hendrickson & Kolda, 2000; Hendrickson & Leland, 1995a; Verbelen et al., 2013; Hendrickson & Leland, 1995b), very few of them have considered instruction-level fine-grained parallelism, which offers novel analytical opportunities for discovering the optimal degree of parallelization and minimizing data communication in multi-core platforms. In this paper, we explore the instruction-level parallelism using advanced graph partitioning techniques on the low level virtual machine (LLVM) intermediate representation (IR) (Lattner & Adve, 2004) graphs and cluster-to-core mapping for optimizing the parallel execution of applications on multicores. The recent work (Xiao et al., 2019; 2017) studies a similar problem and proposed an edge-cut approach via a community-detection-inspired optimization framework to partition dynamic dependency graphs and automatically parallelize the execution of applications while minimizing the inter-core traffic overhead. Although this work achieved better performance in the multi-core parallelism optimization compared to other baseline methods (e.g., thread-based), the graph partition approach in (Xiao et al., 2017) does not consider some important structural properties of the LLVM IR graphs, such as the power-law degree distribution.

[1]Intel Labs, Intel, USA [2]Department of Electrical and Computer Engineering, University of Southern California. Correspondence to: Guixiang Ma <guixiang.ma@intel.com>.

*Proceedings of the 4th MLSys Conference*, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

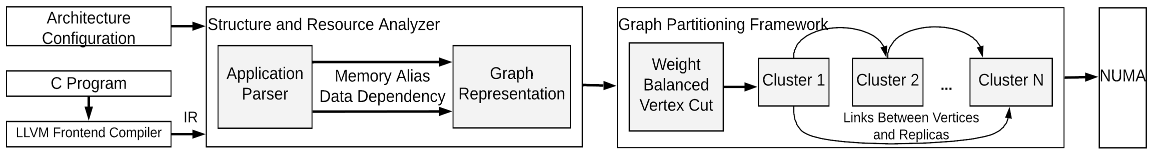

Architecture Configuration
C Program
LLVM Frontend Compiler
IR
Structure and Resource Analyzer
Application Parser
Memory Alias
Data Dependency
Graph Representation
Graph Partitioning Framework
Weight Balanced Vertex Cut
Cluster 1
Cluster 2
...
Cluster N
Links Between Vertices and Replicas
NUMA

Figure 1: Overview of the Proposed Framework. We first pass programs into the structure and resource analyzer to construct LLVM graphs that capture spatial and temporal data communication. Next, we propose a vertex-cut based graph partitioning framework for LLVM graphs to obtain balanced clusters with minimized inter-cluster data communication. Finally, we schedule clusters onto a multi-core non-uniform memory access (NUMA) platform.

This may lead to less-than-ideal graph partitions identified by the optimization model. In this paper, we consider the power-law degree distribution when designing a graph partition framework for LLVM graphs, and propose vertex-cut strategies that partition graphs for better load balancing and parallelism in multi-core systems.

Generally, there are three major challenges in designing a vertex cut framework for LLVM IR graphs: (1) How to formulate the goal of reducing data communication and optimal balanced workloads among multiple cores into the vertex cut graph partitioning problem? (2) How to incorporate edge weights into the vertex cut optimization problem (though most of the existing vertex cut methods are designed for unweighted graphs)? However, the LLVM IR graphs are naturally weighted graphs, where vertices and edges represent instructions and dynamic data dependencies among the instructions, respectively. The edge weights encode the estimated execution time for memory operations, which are crucial for measuring the expected workloads for executing instructions. (3) How to map the graph partitions (i.e., clusters) generated by the vertex-cut approach to the system's processors at run-time.

**Contributions** To address these challenges, we propose a **general and flexible distributed graph-theoretic (vertex cut) framework of LLVM IR graphs for optimal load balancing and parallel execution of applications on multi-core systems** as shown in Fig. 1. The proposed framework has an advantage in incorporating a power-law degree distribution into graph partitioning and can achieve extremely balanced partitions. Therefore, it is an ideal framework for balanced workloads based on graph partitioning. More specifically, we introduce and formalize a new problem, called the *weight-balanced $p$-way vertex cut*, by incorporating edge weights into the optimization of vertex cut-based graph partitioning for load balancing. In addition, we propose novel greedy algorithms for solving this problem, and map the graph clusters to multi-core architectures. Our contributions can be summarized as follows:

- We introduce a vertex cut-based framework for partitioning LLVM graphs, which reduces data communication and achieves optimally balanced workloads amongst a set of cores.

Table 1: Summary of Notations

| | |
|---|---|
| $G$ | Input LLVM graph |
| $V$ | The set of vertices in a graph $G$ |
| $E$ | The set of edges in a graph $G$ |
| $W$ | The weight matrix for graph $G$ |
| $M(e)$ | The set of clusters that contain edge e |
| $A(v)$ | The set of clusters that contain vertex v |
| $w_e$ | The weight of edge e |
| $\alpha$ | The power parameter for the power-law graphs |
| $\lambda$ | The edge weight imbalance factor |

- We model each application as a graph by applying static and dynamic compiler analysis to identify computations and dependencies.
- We prove that the formulated optimization problem obeys a submodular property enabling us to design a greedy algorithm for the Weight-Balanced $p$-way vertex cut problem with guaranteed optimality.

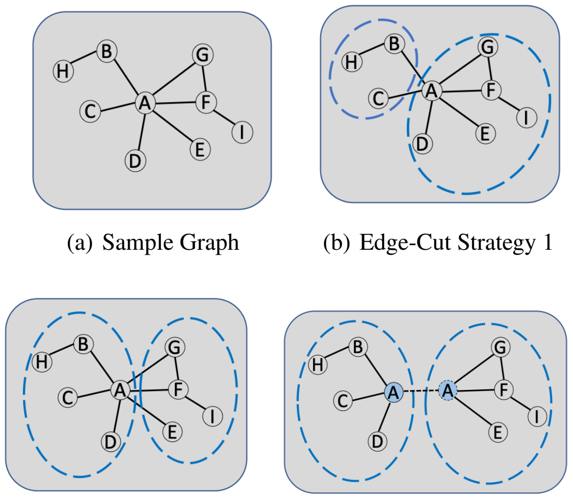
## 2 Notation & Preliminaries

In this section, we introduce notations and provide background for some fundamental concepts used throughout this paper. See Table 1 for a summary of notations.

**Power-law Graphs** Let $G = (V, E)$ denote a graph, where $V$ is the set of vertices and $E \subseteq V \times V$ is the set of edges in $G$. The graph $G$ is a power-law graph if its degree distribution obeys a power law (Adamic & Huberman, 2002):

$$\mathbf{P}(d) \propto d^{-\alpha}, \tag{1}$$

where $\mathbf{P}(d)$ is the probability that a vertex has a degree $d$ and $\alpha$ is a positive constant exponent. The power-law degree distribution means that most vertices in the graph have few neighbors while very few vertices have a large number of neighbors. The exponent $\alpha$ controls the "skewness" of the vertex degree distribution, where a higher $\alpha$ implies a lower ratio of edges to vertices. Many real-world graphs exhibit power-law degree distributions (e.g., social networks). The LLVM graphs that we aim to analyze in this paper are also power-law graphs and will be introduced in Sections 3 and 4. Some examples of LLVM graphs are shown in Fig. 3. The skewed degree distributions in



(a) Sample Graph (b) Edge-Cut Strategy 1

(c) Edge-Cut Strategy 2 (d) Vertex-Cut Strategy

Figure 2: Illustrative Example of Edge Cut vs. Vertex Cut

power-law graphs challenge graph partitioning (Gonzalez et al., 2012), especially for LLVM graphs with a goal of balanced clusters and minimized data communication in parallel computing.

**Edge-Cut** Given a graph $G = (V, E)$, an edge-cut on $G$ is a partition of $V$ into two subsets $S$ and $T$ by cutting some edges in $E$, which results in two clusters with a set of inter-cluster edges $(u, v) \in E|u \in S, v \in T$. Examples of edge-cut based graph partitioning problems include the max-flow min-cut problem (Dantzig & Fulkerson, 2003) in flow graphs and community detection in social networks (Bedi & Sharma, 2016). In parallel computing, calculations can be considered as graphs where nodes represent a series of computations and edges represent data dependencies. Edge-cut based methods have also been studied in this area for partitioning graphs into interconnected clusters to be mapped onto parallel computers (Hendrickson & Kolda, 2000; Hendrickson & Leland, 1995a; Verbelen et al., 2013). In this paper, we will also discuss the state-of-the-art edge-cut based methods and use them as baselines for the optimal parallelism and load balancing in multi-core systems.

**Vertex-Cut** A vertex-cut on $G$ represents a partition of $E$ into two subsets $S$ and $T$ by cutting some vertices in $V$. Whenever a vertex is cut, this vertex will be replicated, and its replica along with a subset of adjacent edges are placed into a different cluster. Instead of having inter-cluster edges like an edge-cut does, vertex-cut partitions only have an inter-cluster connection between each vertex that has been cut and its replicas. Due to these characteristics, a vertex-cut strategy can offer more optimal solutions for graph partitioning tasks compared to an edge-cut strategy.

Fig. 2 illustrates the difference between edge-cut and vertex-cut on a simple graph partitioning scenario, with the goal of minimizing inter-cluster communication while balancing the workloads (i.e., the number of edges) among clusters. Since the vertex $A$ in the graph has a high degree while the other vertices have lower degrees, it is challenging for edge-cut approaches to deal with the edges associated with $A$ in order to achieve low inter-cluster communication (i.e., cross-cluster edges) and a good balance between clusters. Fig. 2(b) shows an edge-cut strategy with low inter-cluster communication but a high imbalance between clusters, while the strategy in Fig. 2(c) achieves a good balance but with more inter-cluster communication cost. On the other hand, the vertex-cut strategy in Fig. 2(d) perfectly addresses the issues by cutting the vertex $A$, where the original $A$ is assigned to the cluster on the left, and a replica of $A$ is assigned to the cluster on the right. The connection between $A$ and its replica is the only communication cost between the two clusters, and the two clusters are well balanced. These examples demonstrate the advantage of vertex-cuts over edge-cuts on a graph with skewed node degrees, and this motivates us to propose a vertex-cut based graph partitioning framework on power-law LLVM graphs to discover the optimal execution and minimal data communication.

## 3 LLVM Graph Construction

We consider each application as an LLVM graph generated from a dynamic trace. The dataflow representation of LLVM graphs requires the advanced graph partitioning algorithms discussed later to find balanced clusters in parallel computing. In this section, we discuss the workflow of LLVM graph construction in three steps: (1) static IR generation via the LLVM front-end from an input program; (2) dynamic IR generation from static IR combined with instrumentation; (3) LLVM graph construction via dependency analysis.

**Definition 3.1.** Let $G = (V, E, W)$ denote an LLVM graph, where each node $v \in V$ represents an LLVM IR instruction, $N = |V|$ is the number of nodes, each edge $e = (u, v) \in E$ represents the data dependency among two nodes, and the corresponding edge weight $w_{uv} \in W$ characterizes the data dependency between node $u$ and node $v$ to guarantee the strict program order.

The LLVM graph in Definition 3.1 captures the spatial and temporal data communication, since the weight $w_{uv}$ measures the amount of time required to transfer data from node $u$ to node $v$ during memory operations. Therefore, we could measure the cost of data communication, which facilitates us to propose an optimization model to partition the LLVM graph into clusters while taking into account data transfer among clusters.

## 3.1 Static IR Generation

Instruction set architecture (ISA) dependent traces include different characteristics and constraints for a specific ISA, which cannot satisfy the fast-growing hardware specialization and ever-expanding workloads. Therefore, in parallel computing, in order to have well-balanced workloads with non-trivial properties and understand the ISA-independent micro-structures, we first compile high-level programs into static LLVM IR. LLVM is a compiler engine that makes program analysis lifelong and transparent by introducing IR as a common model for analysis, transformation, and synthesis (Lattner & Adve, 2004). IR is an intermediate representation between high-level instructions such as Python/C and low-level assembly. It ignores the low-level hardware details while preserving the dataflow structure of programs.

## 3.2 Dynamic Trace Generation

Once the static IR code is generated, we instrument the code to obtain information such as basic blocks and memory time. Once static IR is instrumented, we use the LLVM backend to execute it and collect the execution order of basic blocks and the amount of time for each memory operation. Combined with the hash table, which can be indexed from the execution order of basic blocks, we obtain dynamic IR trace. See Appendices A.2 for an example.

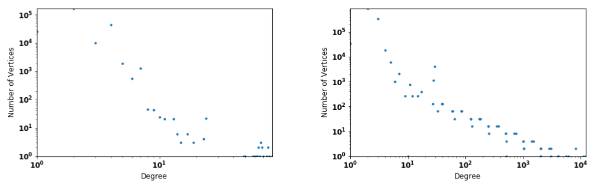
## 3.3 LLVM Graph Construction

The dynamic IR trace captures the dataflow nature of high-level programs. In order to understand the hidden communication structure of the trace and processes that can be potentially be processed in parallel, we construct the LLVM graph by analyzing the data and memory alias dependencies. Data dependency analysis identifies source registers and destination registers for each instruction and checks if source registers of the current instruction match with destination registers of the prior ones. Alias analysis is used to determine if two pointers used in-memory operations have the same address. See Appendices A.3 for an example.

## 4 A Graph-Theoretic Framework

In this section, we introduce our vertex cut framework for partitioning LLVM graphs to optimize the parallel execution of applications in multi-core systems.

By investigating the degree distribution of the LLVM graphs that we construct in Section 3 for some applications, we observe that these LLVM graphs are all power-law graphs, such as the examples shown in Fig. 3 for the Dijkstra algorithm and the fast Fourier transform (FFT) algorithm. The skewed node degree distribution makes the graph partitioning a challenging task on these power-law graphs. As discussed in Section 2, vertex-cut has some advantages over edge-cut on graphs with skewed node degree distributions. Existing works in graph partitioning for distributed graph computing have also shown that vertex-cut methods can achieve better performance in terms of data communication and balance among the partitions than edge-cut methods for power-law graphs (Gonzalez et al., 2012; Xie et al., 2014). In (Gonzalez et al., 2012), a vertex-cut approach called PowerGraph is proposed for solving a balanced $p$-way vertex-cut problem, where the objective is to minimize the average number of vertex replicas while keeping the number of edges balanced among different clusters, so as to minimize the data communication among different clusters while balancing their workloads. In (Xie et al., 2014), a degree-based vertex-cut method called Libra is proposed, which has shown better performance than PowerGraph for the balanced $p$-way vertex-cut task.



(a) Dijkstra (b) FFT
Figure 3: Examples of LLVM Graphs

Although these existing vertex-cut methods have been shown effective in graph partitioning for distributed graph computing, they are designed for unweighted graphs, where the goal of a balanced cut is to keep the number of edges balanced on each cluster. However, LLVM graphs are weighted graphs, where the weights represent the estimated execution time for memory operations, and the goal of a balanced cut is to keep the sum of edge weights in different clusters balanced. Therefore, in this paper, we formulate the Weight Balanced $p$-way Vertex Cut as a new problem and propose a framework for solving this problem.

### 4.1 Weight-Balanced (W-B) $p$-way Vertex Cut

Given an LLVM IR graph $G = (V, E, W)$, our goal is to reduce data communication among different cores (i.e., partitions/clusters) while achieving optimal balanced workloads (i.e., edge weights). We formalize the objective of the weight-balanced $p$-way vertex-cut by assigning each edge $e \in E$ to a cluster $M(e) \in \{1, \cdots, p\}$. Each vertex then spans the set of clusters $A(v) \subseteq \{1, \cdots, p\}$ that contain its adjacent edges. We define the objective as:

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{2}$$

$$\text{s.t.} \quad \max_{m} \sum_{e \in E, M(e)=m} w_e < \lambda \frac{w_{avg}|E|}{p} \tag{3}$$

where $w_e$ is the weight of edge $e$, $w_{avg}$ is the average edge weight of graph $G$, and the imbalance factor $\lambda \geq 1$ is a small constant.

As previously discussed, the balanced $p$-way vertex cut for unweighted graphs has been studied in some works in the literature (Gonzalez et al., 2012; Xie et al., 2014). (Gonzalez et al., 2012) introduces *PowerGraph* and (Xie et al., 2014) proposes *Libra*, both of which are state-of-the-art approaches for vertex cut on unweighted graphs. *PowerGraph* (Gonzalez et al., 2012) first analyzes the randomized vertex cut strategy, and proposes a greedy algorithm for the edge-placement process of the vertex-cuts. In (Xie et al., 2014), a degree-based approach, *Libra*, is proposed for vertex-cut graph partition, which is based on *PowerGraph* but further distinguishes the higher-degree and lower-degree vertices during an edge placement to achieve better performance. Inspired by *PowerGraph* and *Libra*, in the following sections, we will first perform theoretical analysis on the random vertex cut solution for the proposed weighted balanced vertex cut problem and then provide greedy algorithms for it.

## 4.2 Theoretical Analysis

### 4.2.1 *Random Weighted Vertex Cut*

A simple way to perform vertex-cuts is to randomly assign edges to clusters. Based on (Gonzalez et al., 2012), we derive the expected normalized replication factor (Eq.( 2)) in random weighted vertex cut for the weight-balanced $p$-way vertex cut task and obtain the following Theorem.

**Theorem 1.** A random weighted balanced $p$-way vertex cut has an expected replication:

$$\mathbb{E}[\frac{1}{|V|}\sum_{v\in V}|A(v)|] = \frac{p}{|V|}\sum_{v\in V}(1-\mathbb{E}[(\frac{(p-1)}{p})^{\mathbf{D}[v]}]) \quad (4)$$

where $\mathbf{D}[v]$ denotes the degree of vertex $v$. For a power-law graph with exponent $\alpha$, the expected replication is:

$$\mathbb{E}[\frac{1}{|V|}\sum_{v\in V}|A(v)|] = p-\frac{p}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}(\frac{p-1}{p})^d d^{-\alpha} \quad (5)$$

where $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law *Zipf* distribution.

A proof of Theorem 1 is provided in Appendix B.

We can improve the randomly weighted vertex cut with greedy strategies, which assign the next edge onto the cluster that minimizes the conditionally expected replication factor. But before we discuss greedy-based algorithms for the weighted balanced vertex cut, we first prove that the objective function in Eq. (2) is submodular, and a greedy algorithm can provide bounded optimality.

### 4.2.2 *Submodularity of the Objective Function*

**Theorem 2.** The optimization problem is NP-hard.

**Theorem 3.** The objective function (Eq. (2)) is submodular.

**Theorem 4.** The objective function (Eq. (2)) is monotonic.

See Appendix B for the proof of Theorems 2, 3, 4.

**Theorem 5.** Given a monotonic submodular function $f$, the greedy maximization algorithm[1] returns

$$f(A_{greedy}) \geq (1-\frac{1}{e})\max_{|A|<K} f(A) \quad (6)$$

where $K$ is the maximum number of possible assignments. Therefore, even though the optimization problem is NP-hard, algorithm 1 is designed to find an assignment which provides a $(1 - 1/e)$ approximation of the optimal value.

(Krause & Golovin, 2014) proves Theorem 5.

## 4.3 Greedy Algorithms for W-B Vertex Cut

To solve the vertex cut optimization problem defined in Eq. (2) via a greedy approach, we consider the task of placing the $(i + 1)$-th edge after having placed the previous $i$ edges. We define the objective based on the conditional expectation, as shown below.

$$\arg\min_k \mathbb{E}\Big[\sum_{v\in V}|A(v)| \;\Big|\; A_i, A(e_{i+1}) = k\Big] \quad (7)$$

where $A_i$ is the assignment for the previous $i$ edges.

In the following paragraphs, we propose four different greedy solutions for the edge placement of the weight-balanced vertex-cut. We call them Weighted PowerGraph, Weight-Balanced PowerGraph, Weighted Libra, and Weight-Balanced Libra, respectively.

**Weighted PowerGraph** The *PowerGraph* approach is proposed in (Gonzalez et al., 2012) for unweighted vertex cuts, which assigns edges to clusters while balancing the number of edges assigned to each cluster. Inspired by the greedy edge placement in *PowerGraph* and based on the objective of the weighted vertex cut defined in Eq. (2), we define the edge placement rules for our Weighted PowerGraph greedy algorithm as follows. For an edge $(u, v)$,

- Case 1: If $A(u) \cap A(v) \neq \emptyset$, then assign the edge to the least loaded cluster in $A(u) \cap A(v)$, where the workload of each cluster refers to the total weights of all the edges assigned to the cluster.
- Case 2: If $A(u) \cap A(v) = \emptyset$ and $A(u) \neq \emptyset, A(v) \neq \emptyset$, then assign edge $(u, v)$ to the least loaded cluster in

[1]We can easily convert minimization to maximization in this problem by adding a negative sign to the function.

$A(l)$, where $l$ is the vertex from $u, v$ that has more unassigned edges.

- Case 3: If one of $A(u)$ and $A(v)$ is not empty, then assign the edge $(u, v)$ to the least loaded cluster in the non-empty set (i.e., $A(u) \cup A(v)$).
- Case 4: If $A(u) = \emptyset$ and $A(v) = \emptyset$, then assign $(u, v)$ to the least loaded one of the $p$ clusters.

**Weighted Libra** Due to the power-law degree distribution in LLVM graphs, the edge weights associated with high-degree vertices tend to accumulate in a single cluster if these vertices are not cut and spanned over multiple clusters, which can lead to workload imbalance. Moreover, cutting the higher-degree vertices tends to save more communication cost between clusters compared to cutting lower-degree vertices. The Libra unweighted vertex cut approach in (Xie et al., 2014) first proposes a degree-based hashing strategy to address such an issue for cutting power-law graphs, where the higher-degree vertex associated with an edge will be cut with priority if a vertex has to be cut in order to place this edge. Inspired by the degree-based strategy in Libra, we exploit the degree property of vertices during edge placement. Based on Weighted PowerGraph and this degree-based rule, we propose a greedy algorithm called Weighted Libra, which has the following edge placement rules: For an edge $(u, v)$,

- Case 1: If $A(u) \cap A(v) \neq \emptyset$, then assign the edge to the least loaded cluster in $A(u) \cap A(v)$, where the workload of each cluster refers to the total weights of all the edges assigned to the cluster.
- Case 2: If $A(u) \cap A(v) = \emptyset$ and $A(u) \neq \emptyset, A(v) \neq \emptyset$, then assign edge $(u, v)$ to the least loaded cluster in $A(l)$, where $l$ is either $u$ or $v$ whichever has the lower degree.
- Case 3: If one of $A(u)$ and $A(v)$ is not empty, then assign the edge $(u, v)$ to the least loaded machine in the non-empty set (i.e., $A(u) \cup A(v)$).
- Case 4: If $A(u) = \emptyset$ and $A(v) = \emptyset$, then assign $(u, v)$ to the least loaded one of the $p$ clusters.

According to the edge placement rules of Weighted PowerGraph and Weighted Libra, the load balancing among clusters is considered by assigning edges to the least loaded cluster under each case. However, this strategy cannot guarantee the overall balance of the workload (i.e., total edge weights) among different clusters or permit control of the emphasis to put on the balance constraint. To address this issue and further improve load balancing, we incorporate an explicit constraint on the balance of edge weights among clusters into the greedy edge placement rules of the Weighted PowerGraph and Weighted Libra, and have two new greedy algorithms: ***Weight-Balanced PowerGraph*** and ***Weight-Balanced Libra***. Specifically, we incorporate the constraint on the edge weight balance, which is formulated in Eq. (3), into the greedy edge placement rules of the Weighted PowerGraph and Weighted Libra. For cases 1-3 in both algorithms, before placing an edge to the target cluster, we first check if the current sum of edge weights in a target cluster is within the bound given by $\lambda \frac{w_{avg}|E|}{p}$, where $\lambda \geq 1$ is a constant. If it is, then we place the edge into this cluster. Otherwise, we search for another cluster from the remaining set that satisfies this condition as the target cluster for the placement. By setting different values to $\lambda$, we can allow different amounts of emphasis on the workload balance. To illustrate the overall workflow of these greedy algorithms, we summarize the Weighted Balanced Libra greedy algorithm in Algorithm 1 as an example.

**Algorithm 1** Weight-Balanced Libra: A Greedy Algorithm for Vertex Cut Graph Partitioning

```
Input: Edge set $E$; edge weight matrix $W$; vertex set $V$; a set of clusters $C = \{1, 2, \cdots, p\}$; $\lambda$.
Output: The assignment $M(e) \in \{1, 2, \cdots, p\}$ of each edge $e$.
Count the degree $d_i$ for each vertex $v_i, \forall i \in \{1, 2, \cdots, |E|\}$
Compute the cluster weight sum bound $b = \lambda \frac{\sum_{e \in E} w_e}{p}$
for each $e = (v_i, v_j) \in E$ do
  if $A(v_i) = \emptyset$ and $A(v_j) = \emptyset$ then
    $m = leastloaded(C)$
  else if $A(v_i) \neq \emptyset \wedge A(v_j) = \emptyset$ then
    $m = leastloaded(A(v_i))$
    if $load(m) \geq b$ then
      $m = leastloaded(C)$
    end if
  else if $A(v_i) = \emptyset \wedge A(v_j) \neq \emptyset$ then
    $m = leastloaded(A(v_j))$
    if $load(m) \geq b$ then
      $m = leastloaded(C)$
    end if
  else if $A(v_i) \cap A(v_j) \neq \emptyset$ then
    $m = leastloaded(A(v_i) \cap A(v_j))$
    if $load(m) \geq b$ then
      $m = leastloaded(A(v_i) \cup A(v_j))$
      if $load(m) \geq b$ then
        $m = leastloaded(C)$
      end if
    end if
  else
    $s = arg \min_l \{d_l | l \in \{i, j\}\}$
    $t = \{v_i, v_j\} - \{s\}$
    $m = leastloaded(A(s))$
    if $load(m) \geq b$ then
      $m = leastloaded(A(t))$
      if $load(m) \geq b$ then
        $m = leastloaded(C)$
      end if
    end if
  end if
  $M(e) \leftarrow m$; $A(v_i) \leftarrow m$; $A(v_j) \leftarrow m$
end for
```

## 4.4 Discussions

**Time Complexity** According to the workflow of the Weighted Balanced Libra algorithm as shown in Algorithm 1, given a graph $G = (V, E, W)$, for each edge $e$

in $E$, the algorithm retrieves the cluster with the least load (i.e., total edge weights) either from the entire cluster set $C$ or from a subset of $C$. For the former case (line 7, 11, 16, 23, or 33 in Algorithm 1), it takes $O(|C|)$ time, and for the latter case, it takes $O(|C_1|)$ time ($|C_1| \leq |C|$) if the balance constraint is satisfied (line 9, 14, 19, 29), and otherwise it takes $O(|C_1| + |C_2|), |C_2| \leq |C|$ (line 21, 31), or $O(|C_1| + |C|)$ (line 11, 16), or $O(|C_1|+|C_2|+|C|)$ (line 23, 33). Note that line 27 in the algorithm takes $O(1)$. So in the worst case, the algorithm takes $O(3|C|) + O(1) = O(|C|)$ for placing one edge. Therefore, the overall time complexity of Weighted Balanced Libra algorithm is $O(|E| \times |C|)$. Based on the edge placement rules introduced in Section 4.3, this time complexity applies to the three other algorithms as well, although the Weighted Libra and Weighted PowerGraph may take relatively less time in practice compared to Weight-Balanced PowerGraph and Weight-Balanced Libra, since they do not have the weight-balanced constraint. Therefore, they have the time complexity of $O(|C|)$ or $O(|C_1|)$ discussed above for placing one edge.

**Edge Weight Imbalance** Besides the replication factor discussed in Section 4.2.1 as a goal of the optimization model for the weight-balanced vertex cut problem, the edge weight balance among different clusters is another key metric for evaluating the performance, which determines the load balance. As we discussed above in Section 4.3, the degree-based hashing strategy introduced by Libra tends to have more balanced cut results as the higher-degree vertices have a higher priority to be cut than the lower-degree vertices. This statement has also been proved theoretically by (Xie et al., 2014), which shows that Libra can achieve a lower edge imbalance than PowerGraph. By incorporating this degree-based vertex cut rule, our proposed Weighted Libra algorithm is expected to achieve a better load balancing (i.e., a lower edge weight imbalance) than the Weighted PowerGraph algorithm. Furthermore, the proposed Weight-Balanced PowerGraph and Weight-Balanced Libra allow for a further improvement in the load balancing via incorporating a constraint for the edge weight imbalance by the given bound $\lambda \frac{w_{avg}|E|}{p} (\lambda \geq 1)$. If we set $\lambda = 1$, these two algorithms can guarantee near-ideal balanced vertex cut results, with an edge weight imbalance $1 + \epsilon$, where $\epsilon$ is a small non-negative constant.

## 5 Architecture and Mapping

**NUMA Architecture.** Uniform memory access (UMA) is a shared memory architecture for processors running in parallel as shown in Figure 4. It develops a unified vision of the shared physical memory, meaning that access time to a particular memory address is independent regardless of which processor requests data from different memory banks. On the contrary, NUMA allows memory access time dependent on the relative processor location. That is, a processor has fast memory access time to its local memory and slow access to the rest of memory. This non-uniformity enables potential fewer memory accesses with fast access time. Limiting the number of memory accesses and fast memory accesses provides the key to high performance computing. Therefore, we decide to apply the NUMA architecture in the experiments to fully reduce the amount of data communication between processors and memories.

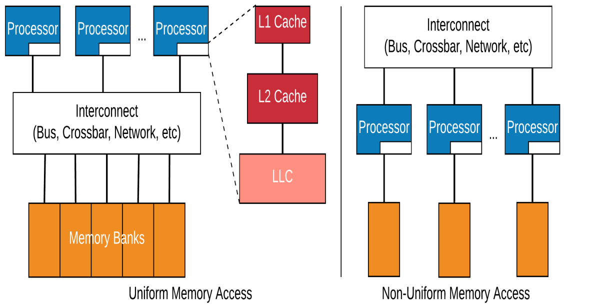


Figure 4: UMA and NUMA Architectures. UMA is a shared memory architecture with uniform memory access whereas NUMA enables fast memory access for a processor to its local physical memory and slow memory access to the rest of memory banks.

Table 2: System Configuration

| | | |
|---|---|---|
| **CPU** | Cores | Out-of-order cores |
| | Clock frequency | 2.4 GHz |
| | L1 private cache | 64KB, 4-way associative 32-byte blocks |
| | L2 shared cache | 256KB, distributed |
| | Memory | 4 GB, 8 GB/s bandwidth |
| **Network** | Topology | Mesh |
| | Routing algorithm | XY routing |
| | Flow control | Virtual channel flit-based |

**Memory-Centric Run-time Mapping.** At run-time, processes/clusters generated in Section 4 from each application are mapped onto processors in a NUMA architecture to be executed. We employ a memory-centric run-time mapping approach for the mapping, which exploits and optimizes the parallelism in clusters while considering data communication between clusters and resource utilization. For a detailed description of the pipeline and algorithm for this run-time mapping, please see Appendix E.

## 6 Evaluations

In this section, we discuss the simulator configurations and present experimental results to investigate the soundness of the proposed methodology.

### 6.1 Simulation Configurations

We use gem5 (Binkert et al., 2011) to simulate a varying number of out-of-order cores with the NUMA architecture. Table 2 shows detailed simulation parameters. We consider the following applications from various benchmarks (Dorta et al., 2005; Guthaus et al., 2001): *FFT*, *Mandel*, *MD*, *Dijkstra*, *NN*, *Neuron*, *CNN*, *Strassen8* and *Strassen16*. For more detailed descriptions of these benchmarks, see Table 8 in Appendix C. We generate LLVM IR graphs from these applications following the procedure introduced in Section 3. For baseline comparisons, we consider four baseline methods for graph partitioning: (1) the work in (Xiao et al., 2017) abbreviated as *CompNet*; (2) *METIS* (LaSalle et al., 2015), which is an edge-cut method that implements various multilevel algorithms by iteratively coarsening a graph, computing a partition, and projecting the partition back to the original graph; (3) the unweighted vertex-cut method *PowerGraph* (*PG*) (Gonzalez et al., 2012); and (4) the unweighted vertex-cut method *Libra* (Xie et al., 2014). We compare these baselines with the proposed four greedy vertex-cut algorithms: *Weighted PowerGraph (W-PG)*, *Weight-Balanced PowerGraph (WB-PG)*, *Weighted Libra (W-Libra)*, *Weight-Balanced Libra (WB-Libra)*.

## 6.2 Experimental Results

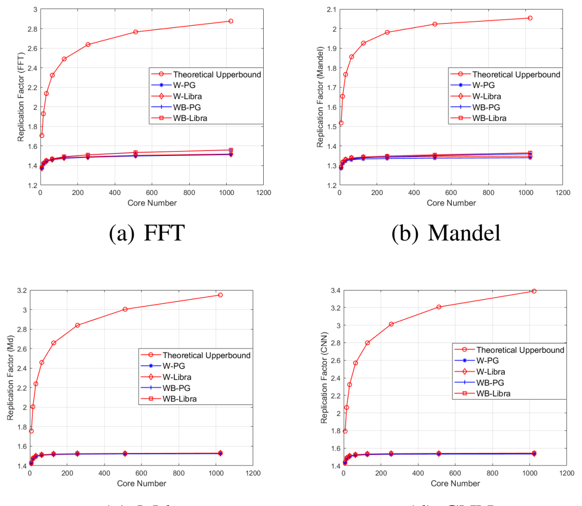
In this section, we evaluate the proposed methods and baselines on the LLVM graphs transformed from the benchmarks listed in Table 8 for the proposed graph partitioning and compare their performance in the graph partition quality (in terms of replication factor and edge weight imbalance). Next, we execute clusters generated from each method to measure the overall execution time and data communication. We also analyze the sensitivity of the parameter $\lambda$ involved in the constraint of load balancing to the execution time.

### *6.2.1 Replication Factor.*

In Section 4.2.1, we have derived the theoretical expected replication factor of the weighted vertex cut with random edge placement, which is in fact a theoretical upper-bound for the replication factor of the proposed greedy algorithms. We now empirically evaluate the performance of the proposed four greedy vertex cut algorithms in replication factor, and compare the results with the theoretical upper-bound we compute by Eq. (5). Fig. 5 shows the results on four graphs. As we can see, the four greedy algorithms achieve comparable performance in the replication factor. All of their replication factors are within the theoretical upper-bound with a considerable gap, which indicates the superior advantage of the greedy vertex-cut algorithms over the random vertex cut strategy.

### *6.2.2 Edge Weight Imbalance.*

As discussed in Section 4.4, edge weight imbalance is a key metric for evaluating the performance of vertex-cuts in load balancing among clusters. The edge weight imbalance is defined by $(\max_m \sum_{e \in E, M(e)=m} w_e)/(\frac{w_{avg}|E|}{p})$, which measures how much the most-loaded cluster deviates from the expected average load between clusters. A good load-balancing vertex-cut method should achieve an edge weight imbalance close to 1, which indicates the absolute balance. We evaluate the edge weight imbalance of all the six vertex cut methods, where we set $\lambda = 1$ in the sum of weights in a cluster (line 4 of Algorithm 1) for WB-PG and WB-Libra, in order to obtain their optimal balance of edge weights for comparisons with the other methods. Table 4 shows the results from edge weight imbalance of the six methods on all ten graphs. We observe from the table that, WB-Libra achieves the best results in most of the graphs, except for Dijkstra, Mandel, and Md, where WB-PG performs the best. Both the two unweighted vertex cut methods, i.e. PowerGraph and Libra, achieve worse results compared to the four weighted vertex cut methods. This is mainly due to the fact that, the unweighted vertex cut was designed to balance the number of edges among clusters for unweighted graphs and therefore they can not properly handle the load balancing for weighted graphs. By comparing between WB-PG and W-PG, and between WB-Libra and W-Libra, we can see that the edge weight balance constraint we incorporate into the weighted balanced algorithms is effective for further improving the edge weight balance among clusters and is able to push the balance to the near-ideal situation.



(a) FFT (b) Mandel

(c) Md (d) CNN

Figure 5: Replication Factor of the Proposed Four Greedy Algorithms with Comparison to the Computed Theoretical Upper-bound by Eq. (5)

### *6.2.3 Execution Time*

Tables 5 and 6 show the execution time for 8 and 1024 clusters, respectively. See Appendix D.1 for more experiments. In general, the vertex-cut methods achieve a better performance than edge-cut baselines CompNet and METIS. This

Table 3: Statistics of Graph Datasets

| Graph Dataset | No. Nodes | No. Edges | power-law $\alpha$ | Avg. Path Length |
|---|---|---|---|---|
| Dijkstra | 248,959 | 291,112 | 2.29 | 136.4 |
| FFT | 109,295 | 143,183 | 2.21 | 194.56 |
| K-means | 98,592 | 119,112 | 2.24 | 479.4 |
| Mandel | 235,051 | 260,042 | 2.43 | 42.67 |
| Md | 1,799,353 | 2,361,213 | 2.17 | 524.61 |
| NN | 124,496 | 161,428 | 2.16 | 171.52 |
| Neuron | 57,883 | 73,431 | 2.20 | 179.25 |
| CNN | 573,694 | 758,712 | 2.13 | 824.37 |
| Strassen8 | 36,831 | 46,756 | 2.21 | 21.22 |
| Strassen16 | 197,827 | 254,392 | 2.20 | 123.94 |

Table 4: Edge Imbalance of the Vertex Cut Methods on Graphs

| Datasets | PG | W-PG | WB-PG | Libra | W-Libra | WB-Libra |
|---|---|---|---|---|---|---|
| Dijkstra | 1.00227 | 1.00092 | **1.00007** | 1.02136 | 1.00106 | 1.00010 |
| FFT | 1.00586 | 1.00831 | 1.00075 | 1.05030 | 1.00400 | **1.00057** |
| K-means | 1.00177 | 1.00469 | 1.00042 | 1.04566 | 1.00180 | **1.00035** |
| Mandel | 1.00730 | 1.00233 | **1.00008** | 1.00749 | 1.00171 | 1.00014 |
| Md | 1.00015 | 1.00007 | **1.00003** | 1.00791 | 1.00008 | **1.00003** |
| NN | 1.00187 | 1.00235 | 1.00028 | 1.03441 | 1.00106 | **1.00019** |
| Neuron | 1.00260 | 1.00487 | 1.00081 | 1.05738 | 1.00236 | **1.00045** |
| CNN | 1.00040 | 1.00035 | 1.00010 | 1.00956 | 1.00027 | **1.00008** |
| Strassen8 | 1.01074 | 1.01307 | 1.00177 | 1.05036 | 1.00787 | **1.00123** |
| Strassen16 | 1.00338 | 1.00352 | 1.00029 | 1.04206 | 1.00170 | **1.00028** |

verifies our expectation that the vertex-cut based graph partitioning methods can work better than edge-cut methods for power-law graphs. Among the six vertex-cut methods, the proposed four methods (i.e., W-PG, WB-PG, W-Libra and WB-Libra) outperform the two unweighted vertex-cut methods. This is reasonable since the unweighted vertex-cuts are not able to handle the load balancing for weighted graphs, as we discussed in Section 4.4, and the load imbalance among clusters will lead to a longer overall execution time for the applications, as the overall execution time depends on the time for executing the cluster with the largest workload. Among the four proposed methods, WB-Libra achieve the best performance in most cases consistently for all different numbers of clusters. This demonstrates the benefit of using the degree-based vertex hashing strategy and the load balancing constraint in the vertex-cuts. These results in execution time indicate that the proposed vertex-cut based graph partitioning framework is effective in load balancing and parallelism optimization for multi-core systems and it has superior performance than the state-of-the-art baselines.

### 6.2.4 *Data Communication*

Tables 7 shows the data communication of each application for 8 clusters. See Appendix D.2 for more experiments.

Table 5: Overall Execution Time (/s) for 8 Clusters From Different Algorithms in a Multi-core Platform

| Datasets | CompNet | METIS | PG | W-PG | WB-PG | Libra | W-Libra | WB-Libra |
|---|---|---|---|---|---|---|---|---|
| Dijkstra | 317.27 | 332.48 | 346.15 | 263.75 | 260.91 | 253.86 | 262.51 | **242.26** |
| FFT | 279.6 | 288.22 | **209.27** | 253.71 | 230.02 | 248.42 | 239.33 | 291.53 |
| K-means | 244.87 | 261.38 | 279.53 | 206.25 | 195.53 | 201.54 | 188.25 | **178.58** |
| Mandel | 341.35 | 373.28 | 265.15 | 289.74 | 257.12 | 277.31 | 256.49 | **245.82** |
| Md | 2568.72 | 2723.71 | 2822.47 | 2313.9 | 1821.68 | 2178.41 | 1824.95 | **1642.18** |
| NN | 351.23 | 376.93 | 354.32 | 311.74 | 256.41 | 297.59 | 278.24 | **253.89** |
| Neuron | 214.75 | 242.68 | 213.95 | 187.23 | 163.63 | 174.54 | 157.69 | **131.4** |
| CNN | 1568.59 | 1736.37 | 1454.88 | 1425.63 | 1221.53 | 1358.61 | 1315.78 | **1175.8** |
| Strassen8 | 142.41 | 155.39 | 131.24 | 121.37 | 104.99 | 112.62 | 111.23 | **96.24** |
| Strassen16 | 326.75 | 351.26 | 323.5 | 304.21 | 274.63 | 285.44 | 264.58 | **248.25** |

Table 6: Overall Execution Time (/s) for 1024 Clusters From Different Algorithms in a Multi-core Platform

| Datasets | CompNet | METIS | PG | W-PG | WB-PG | Libra | W-Libra | WB-Libra |
|---|---|---|---|---|---|---|---|---|
| Dijkstra | 31.08 | 46.48 | 33.5 | 27.79 | 29.27 | 29.58 | 26.43 | **23.4** |
| FFT | 24.92 | 32.4 | 25.08 | 22.64 | 23.96 | 23.2 | 20.31 | **18.59** |
| K-means | 16.77 | 37.23 | 18.26 | 16.37 | 12.54 | 15.3 | 13.92 | **11.26** |
| Mandel | 23.48 | 41.37 | 15.8 | 19.81 | 14.52 | 17.47 | 14.43 | **12.6** |
| Md | 228.43 | 245.61 | 233.02 | 204.53 | 174.23 | 192.23 | 169.18 | **155.71** |
| NN | 33.44 | 52.36 | 34.92 | 29.35 | 25.48 | 29.84 | 27.9 | **23.22** |
| Neuron | 21.3 | 48.32 | 23.73 | 20.93 | 21.62 | 19.19 | 17.97 | **16.11** |
| CNN | 157.5 | 170.92 | 148.48 | 145.87 | 110.67 | 132.22 | 119.43 | **105.29** |
| Strassen8 | 12.51 | 15.03 | 14.57 | 13.55 | 12.39 | 12.17 | 11.11 | **10.31** |
| Strassen16 | 31.81 | 38.14 | 29.23 | 29.62 | 27.36 | 27.22 | 25.25 | **23.42** |

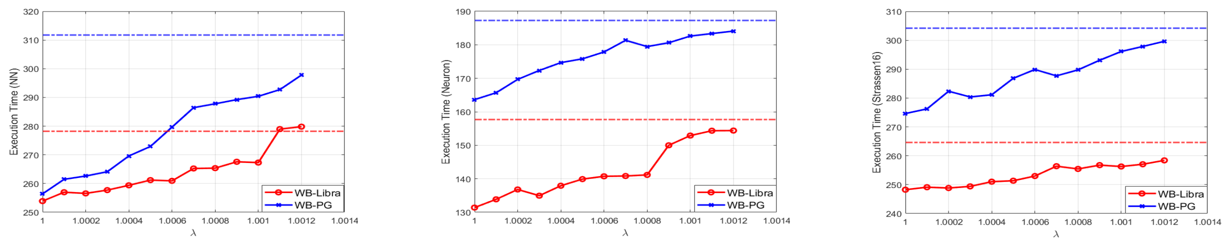
Table 7: Data Communication for 8 Clusters From Different Graph Partitioning Algorithms

| Datasets | CompNet | METIS | PG | W-PG | WB-PG | Libra | W-Libra | WB-Libra |
|---|---|---|---|---|---|---|---|---|
| Dijkstra | 100% | 142% | 60% | 46% | 54% | 50% | 54% | 57% |
| FFT | 100% | 156% | 71% | 53% | 65% | 56% | 62% | 65% |
| K-means | 100% | 135% | 59% | 41% | 52% | 46% | 52% | 55% |
| Mandel | 100% | 158% | 48% | 31% | 41% | 36% | 42% | 45% |
| Md | 100% | 160% | 47% | 33% | 39% | 36% | 40% | 44% |
| NN | 100% | 169% | 63% | 43% | 55% | 48% | 53% | 57% |
| Neuron | 100% | 137% | 59% | 42% | 53% | 47% | 52% | 56% |
| CNN | 100% | 192% | 64% | 50% | 56% | 53% | 55% | 58% |
| Strassen8 | 100% | 171% | 55% | 43% | 50% | 46% | 48% | 52% |
| Strassen16 | 100% | 193% | 54% | 42% | 46% | 45% | 47% | 50% |

In general, all the vertex cut methods reduce more data compared to the edge-cut methods (METIS and CompNet). For example, according to Table 7, the WB-PG reduces the data communication for 8-cores by an average of $48.9\%$ compared to CompNet over the 10 graphs, and WB-Libra reduces it by an average of $46.1\%$. METIS fails to reduce data communication compared to others. However, it is interesting to note that METIS causes less than 120% for the Mandelbrot application whereas the data communication is more than 130% for the rest of applications. It is because Mandelbrot is an embarrassingly parallel application where little effort is required to separate it into a number of parallel clusters. However, traditional edge cut algorithms such as CompNet and METIS still lead to a seriously huge amount of data communication between clusters, while vertex cut methods is able to maintain a much lower communication cost. This is mainly because that the data communication in edge-cut partitions comes from all the inter-cluster edges, while there is no such communication cost in vertex-cut partitions since there is no inter-cluster edges and the only communication for the vertex-cut partitions is the communication between the replicas of the cut vertices across different clusters.

### 6.2.5 *Parameter Sensitivity Analysis*

Fig. 6 shows the execution time for several $\lambda$ values in Eq. (3) on three applications, i.e., NN, Neuron, and Strassen16. $\lambda$ controls the clusters balance. In WB-PG and WB-Libra algorithms, we use $\lambda$ to set a balance bound (line #4 in Algorithm 1), and $\lambda = 1$ indicates an ideal balanced case. When $\lambda$ is large enough, WB-Libra and WB-PG reduce to W-Libra and W-PG, respectively. To analyze the sensitivity of $\lambda$ parameter to the execution time, we evaluate WB-Libra and WB-PG with different $\lambda$ values in the range of 1 to 1.0012 with a step size of 0.0001. The dotted blue line refers to



(a) NN (b) Neuron (c) Strassen16

Figure 6: Execution Time With Different $\lambda$ Values for WB-Libra and WB-PG Algorithms. Dotted lines indicate the execution time for W-Libra and W-PG to which WB-Libra and WB-PG reduce, respectively, when $\lambda$ becomes large enough.

the performance of W-PG and the dotted red line refers to the performance of W-Libra, which can be treated as upper bound for WB-PG and WB-Libra, respectively. There are times when the execution time of applications exceeds the upper bound indicated by the dotted lines because of frequent synchronization such as fetching data from memory and flushing dirty data into memory. In general, the trend is going up, indicating that increasing $\lambda$ causes the performance degradation. It is recommended to set $\lambda = 1$ in WB-Libra and WB-PG to improve the execution time.

## 7 Related Work

Parallel computing enables the continued growth of complex applications (Asanovic et al., 2006; 2009). Most existing work (Murray et al., 2013; Yu et al., 2008; Murray et al., 2011) exploits the coarse-grained parallelism of the dataflow graphs where it is common to represent computations as nodes and data dependencies among them as edges. The work in (Murray et al., 2013) proposes a new computational model, timely dataflow, and captures opportunities for parallelism across different algorithms. Timely dataflow combines dataflow graphs with timestamps to allow vertices to send and receive logically timestamped messages along directed edges for data-parallel computation in a distributed cluster. (Yu et al., 2008) proposes DryadLINQ, a system for general-purpose data-parallel computation. The system architecture incorporates the dataflow graph representation of jobs with a centralized job manager to schedule jobs on clustered computers. (Murray et al., 2011) introduces CIEL, a universal execution engine for distributed dataflow programs. It coordinates the distributed execution of a set of data-parallel tasks and dynamically builds the DAG as tasks execute. Others develop different edge-cut graph partitioning algorithms in parallel computing such as spectral graph theory (Hendrickson & Leland, 1995a), hypergraph models (Hendrickson & Kolda, 2000; Devine et al., 2006), and a multi-level algorithm (Hendrickson & Leland, 1995b). Few (Xiao et al., 2017; 2018) exploit the fine-grained instruction parallelism in high-level programs and propose community detection inspired optimization models to benefit from the underlying hardware such as multi-core platforms and processing-in-memory architectures.

Related works in vertex cut are mainly from the distributed graph computing field, where vertex-cuts are used to partition large power-law graphs for optimizing the distributed execution of real applications such as *PageRank*. The PowerGraph (Gonzalez et al., 2012) and Libra (Xie et al., 2014) discussed in previous sections are two state-of-the-art works in this field. Some other relevant works include (Jain et al., 2013), (Gonzalez et al., 2014), and (Chen et al., 2019).

## 8 Conclusion

In this paper, we explore IR instruction-level parallelism via graph partitioning on universal LLVM IR graphs and cluster-to-core mapping for automatic parallelization in multi-core systems. We propose a vertex cut based framework on LLVM IR graphs for load balancing and parallel optimization of application execution in multi-core systems. Towards solving this problem, we developed Weight-Balanced $p$-way Vertex Cut-inspired greedy algorithms. Our simulation results demonstrate the superior performance of the proposed framework for load balancing and multi-core execution speed-up compared to the state-of-the-art baselines.

**Acknowledgement:** P.B. and his Cyber-Physical Systems group gratefully acknowledge the support by the National Science Foundation under the Career Award CPS/CNS-1453860, the NSF award under Grant Numbers CCF-1837131, MCB-1936775, and CNS-1932620, and the DARPA Young Faculty Award and DARPA Director Award, under Grant Number N66001-17-1-4044. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied by the Defense Advanced Research Projects Agency, the Department of Defense or the National Science Foundation.

## References

Adamic, L. A. and Huberman, B. A. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

Andreev, K. and Racke, H. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., et al. The landscape of parallel computing research: A view from berkeley. 2006.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

Bedi, P. and Sharma, C. Community detection in social networks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 6(3):115–135, 2016.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

Chen, C. P. and Zhang, C.-Y. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information sciences*, 275:314–347, 2014.

Chen, R., Shi, J., Chen, Y., Zang, B., Guan, H., and Chen, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

Dantzig, G. and Fulkerson, D. R. On the max flow min cut theorem of networks. *Linear inequalities and related systems*, 38:225–231, 2003.

Devine, K. D., Boman, E. G., Heaphy, R. T., Bisseling, R. H., and Catalyurek, U. V. Parallel hypergraph partitioning for scientific computing. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pp. 10–pp. IEEE, 2006.

Dorta, A. J., Rodriguez, C., and de Sande, F. The openmp source code repository. In *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 244–250. IEEE, 2005.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 17–30, 2012.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 599–613, 2014.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pp. 3–14. IEEE, 2001.

Hendrickson, B. and Kolda, T. G. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.

Hendrickson, B. and Leland, R. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995a.

Hendrickson, B. and Leland, R. W. A multi-level algorithm for partitioning graphs. *SC*, 95(28):1–14, 1995b.

Jain, N., Liao, G., and Willke, T. L. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, pp. 1–6, 2013.

Krause, A. and Golovin, D. Submodular function maximization, 2014.

LaSalle, D., Patwary, M. M. A., Satish, N., Sundaram, N., Dubey, P., and Karypis, G. Improving graph partitioning for modern graphs and architectures. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pp. 1–4, 2015.

Lattner, C. and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. IEEE, 2004.

Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. Ciel: a universal execution engine for distributed data-flow computing. 2011.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, 2013.

Verbelen, T., Stevens, T., De Turck, F., and Dhoedt, B. Graph partitioning algorithms for optimizing software deployment in mobile cloud computing. *Future Generation Computer Systems*, 29(2):451–459, 2013.

Xiao, Y., Xue, Y., Nazarian, S., and Bogdan, P. A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 217–224. IEEE, 2017.

Xiao, Y., Nazarian, S., and Bogdan, P. Prometheus: Processing-in-memory heterogeneous architecture design from a multi-layer network theoretic strategy. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1387–1392. IEEE, 2018.

Xiao, Y., Nazarian, S., and Bogdan, P. Self-optimizing and self-programming computing systems: A combined compiler, complex networks, and machine learning approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(6):1416–1427, 2019.

Xie, C., Yan, L., Li, W.-J., and Zhang, Z. Distributed power-law graph computing: Theoretical and empirical analysis. In *Advances in neural information processing systems*, pp. 1673–1681, 2014.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P. K., and Currey, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 1–14, USA, 2008. USENIX Association.

## APPENDIX

## A. LLVM GRAPH CONSTRUCTION

### A.1 Static IR Generation

We generate LLVM static IR instructions by applying the front-end compiler clang, as shown in Fig. 8.

### A.2 Dynamic Trace Generation

Once the static IR code is generated, we instrument the code to obtain information such as basic blocks and memory time. First, we use a hash table to keep track of IR instructions within each basic block. For example, in Fig. 8, instructions from ”*%1 = alloca i32*, align 8*” up to ”*br label %5*” should be hashed into the index 1 which represents the first basic block. Second, at the beginning of each basic block, we instrument a *printf* function to record the execution order of blocks. Fig. 8 shows the full instrumentation of *printf* statements in blue. Last, we use the time stamp counter *rdtsc* and the *printf* statements to measure the amount of time for each memory operation ($time = after_{mem} - before_{mem}$). Instructions in red in Fig. 8 show this instrumentation for the first two memory operations. Specifically, we insert *rdtsc* before and after each memory operation and calculate the difference as the amount of execution time. Once static IR is instrumented, we use the LLVM back-end to execute it and collect the execution order of basic blocks and the amount of time for each memory operation. Combined with the hash table, which can be indexed from the execution order of basic blocks, we obtain dynamic IR trace as shown in Fig. 8.

### A.3 LLVM Graph Construction

We construct the LLVM graph by analyzing the data and memory alias dependencies. For example, as shown in Fig. 8, the sixth instruction ”*store i32* %a, i32** %1, align 8*” has the source register %1 which depends on the destination register of the first instruction ”*%1 = alloca i32*, align 8*”. The corresponding LLVM graph manifests this dependency by inserting a directed edge from node 1 to node 6.

## B. PROOF OF THEOREMS

**Theorem 1.** A random weighted balanced $p$-way vertex cut has an expected replication:

$$\mathbb{E}[\frac{1}{|V|}\sum_{v\in V}|A(v)|] = \frac{p}{|V|}\sum_{v\in V}(1-\mathbb{E}[(\frac{(p-1)}{p})^{\mathbf{D}[v]}]) \tag{8}$$

where $\mathbf{D}[v]$ denotes the degree of vertex $v$. For a power-law graph with exponent $\alpha$, the expected replication is:

$$\mathbb{E}[\frac{1}{|V|}\sum_{v\in V}|A(v)|] = p - \frac{p}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}(\frac{p-1}{p})^d d^{-\alpha} \tag{9}$$

where $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law *Zipf* distribution.

According to linearity of expectation, we have:

$$\mathbb{E}[\frac{1}{|V|}\sum_{v\in V}|A(v)|] = \frac{1}{|V|}\sum_{v\in V}\mathbb{E}[|A(v)|] \tag{10}$$

where $\mathbb{E}[|A(v)|]$ is the expected replication number of a single vertex $v$.

Assume vertex $v$ has a degree $\mathbf{D}[v]$, then the expected replication of $v$ can be computed by the process of assigning the $\mathbf{D}[v]$ edges that are adjacent to $v$. Let $X_i$ denote the event that vertex $v$ has at least one of its edges on cluster $i$, then the expectation $\mathbb{E}[X_i]$ is:

$$\begin{aligned}\mathbb{E}[X_i] &= 1 - \mathbf{P}(v \text{ has no edges on cluster } i)\\ &= 1-(1-\frac{1}{p})^{\mathbf{D}[v]}\end{aligned} \tag{11}$$

Then, the expected replication factor for vertex $v$ is:

$$\mathbb{E}[|A(v)|] = \sum_{i=1}^{p}\mathbb{E}[X_i] = p(1-(1-\frac{1}{p})^{\mathbf{D}[v]}) \tag{12}$$

In the power-law graph, $\mathbf{D}[v]$ can be treated as a *Zipf* random variable, therefore Eq.(12) can be further written as:

$$\mathbb{E}[|A(v)|] = p(1-\mathbb{E}[(\frac{(p-1)}{p})^{\mathbf{D}[v]}]) \tag{13}$$

Then:

$$\mathbb{E}[\frac{1}{|V|}\sum_{v\in V}|A(v)|] = \frac{p}{|V|}\sum_{v\in V}(1-\mathbb{E}[(\frac{(p-1)}{p})^{\mathbf{D}[v]}]) \tag{14}$$

In the power-law graph $G$, the probability of a vertex degree being $d$ is $\mathbf{P}(d) = d^{-\alpha}/\mathbf{h}_{|V|}(\alpha)$, where $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law *Zipf* distribution. Then,

$$\mathbb{E}[(\frac{(p-1)}{p})^{\mathbf{D}[v]}] = \frac{1}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}(1-\frac{1}{p})^d d^{-\alpha} \tag{15}$$

By plugging Eq.(15) into Eq.(8), we have:

$$\mathbb{E}[\frac{1}{|V|}\sum_{v\in V}|A(v)|] = p - \frac{p}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}(\frac{p-1}{p})^d d^{-\alpha} \tag{16}$$

**Theorem 2.** The optimization problem is NP-hard.

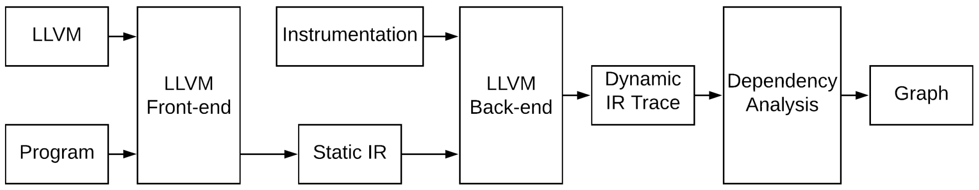


Figure 7: Workflow of LLVM Graph Construction. Each program is first compiled to static IR instructions via the LLVM front-end, which is next translated into dynamic IR trace via the LLVM back-end, combined with instrumentation to obtain information such as memory timing and the sequence of the execution order of basic blocks. Last, we perform dependency analysis to construct a graph based on the dynamic trace where nodes denote IR instructions and edges represent dependencies.

*Proof.* K-balanced graph partitioning (Andreev & Racke, 2006) divides a graph into $k$ equal sized clusters while minimizing the capacity of edges cut, which is NP-hard. It reduces to the optimization problem by having a unit weight for each edge in a graph to be cut. □

**Theorem 3.** The objective function (Eq. (2)) is submodular.

*Proof.* Given an LLVM IR graph $G = (V, E, W)$, define two assignment sets $X, Y = \{A(v_1), A(v_2), ..., A(v_{|V|})\} \subseteq \Omega$ where for any node $v$, $A(v) \subseteq \{1, \cdots, p\}$ and $\Omega$ is the solution space of the problem. We define $f(X)$ as the objective function defined in Eq. (2).
If $X \cap Y = \emptyset$, then

$$\begin{aligned} f(X \cap Y) &+ f(X \cup Y) \\ &= \frac{1}{|V|}\sum_{v\in V}|X(v) + Y(v)| + \cancel{f(\emptyset)}^{\nearrow 0} \\ &= \frac{1}{|V|}\sum_{v\in V}|X(v)| + \frac{1}{|V|}\sum_{v\in V}|Y(v)| \\ &= f(X) + f(Y) \end{aligned} \tag{17}$$

If $X \cap Y = S_c$ where $S_c$ is a set of the common elements, then

$$\begin{aligned} & f(X \cap Y) + f(X \cup Y) \\ &= \frac{1}{|V|}\sum_{v\in V}\{|X(v)| + |Y(v)| - |S_c(v)| + |S_c(v)|\} \\ &= \frac{1}{|V|}\sum_{v\in V}|X(v)| + \frac{1}{|V|}\sum_{v\in V}|Y(v)| \\ &= f(X) + f(Y) \end{aligned} \tag{18}$$

Therefore, by combining Eq. (17) and Eq. (18), we can infer that the objective function is submodular as for any two sets $X, Y \subseteq \Omega$, $f(X) + f(Y) = f(X \cap Y) + f(X \cup Y)$. □

**Theorem 4.** The objective function in the Eq. (2) is monotonic.

*Proof.* Given an LLVM IR graph $G = (V, E, W)$, we define an assignment set $A = \{v_1, v_2, ..., v_{|V|}\} \subseteq \Omega$ and an arbitrary assignment $A' = v_k \cup A$.

$$\begin{aligned} f(A \cup v_k) &= \frac{1}{|V|}\sum_{v\in V}|A'(v)| \\ &= \frac{1}{|V|}\sum_{v\in V}\{|A(v)| + |v_k(v)|\} \\ &= \frac{1}{|V|}\sum_{v\in V}|A(v)| + \frac{1}{|V|}\sum_{v\in V}|v_k(v)| \\ &= f(A) + f(v_k) \end{aligned} \tag{19}$$

Therefore, $f(A \cup v_k) - f(A) \geq 0$. □

## C. Description of Benchmarks

Table 8 provides the detailed descriptions of the application benchmarks used in the experiments.

## D. Experimental Results

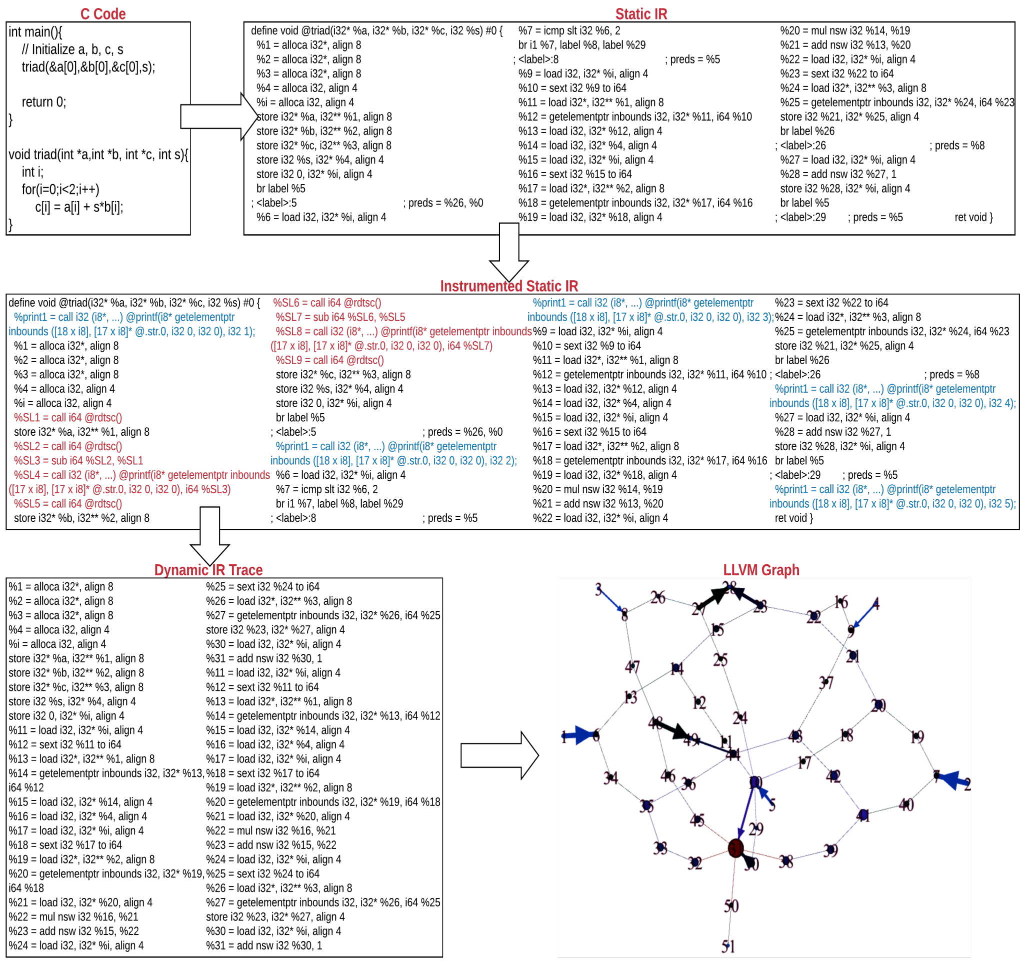
### D.1 Execution Time

Fig. 9 shows the execution time of each application for different graph partitioning algorithms with various cluster numbers.

### D.2 Data Communication

Table 9 lists the data communication of each application for the different graph partitioning algorithms for 1024 clusters.

Fig. 10 shows data communication of each application for the graph partitioning algorithms with different cluster numbers. As shown in the figure, the general trend of data communication from 8 clusters up to 1024 clusters is it first goes down and up again at 128 clusters. The trend of data communication going down is mainly due to the efficient parallelism while minimizing data communication. However, as the number of clusters increases beyond 128 clusters, synchronization starts to take over the impact of data communication because processes are synchronized to allow only one process enter the critical section to modify the shared data structures in main memory. Nevertheless, the least data communication overhead in these cases is still from the proposed vertex-cut algorithms.



C Code

```
int main(){
   // Initialize a, b, c, s
   triad(&a[0],&b[0],&c[0],s);

   return 0;
}

void triad(int *a,int *b, int *c, int s){
   int i;
   for(i=0;i<2;i++)
      c[i] = a[i] + s*b[i];
}
```

Static IR

```
define void @triad(i32* %a, i32* %b, i32* %c, i32 %s) #0 {
  %1 = alloca i32*, align 8
  %2 = alloca i32*, align 8
  %3 = alloca i32*, align 8
  %4 = alloca i32, align 4
  %i = alloca i32, align 4
  store i32* %a, i32** %1, align 8
  store i32* %b, i32** %2, align 8
  store i32* %c, i32** %3, align 8
  store i32 %s, i32* %4, align 4
  store i32 0, i32* %i, align 4
  br label %5
; <label>:5                              ; preds = %26, %0
  %6 = load i32, i32* %i, align 4
  %7 = icmp slt i32 %6, 2
  br i1 %7, label %8, label %29
; <label>:8                              ; preds = %5
  %9 = load i32, i32* %i, align 4
  %10 = sext i32 %9 to i64
  %11 = load i32*, i32** %1, align 8
  %12 = getelementptr inbounds i32, i32* %11, i64 %10
  %13 = load i32, i32* %12, align 4
  %14 = load i32, i32* %4, align 4
  %15 = load i32, i32* %i, align 4
  %16 = sext i32 %15 to i64
  %17 = load i32*, i32** %2, align 8
  %18 = getelementptr inbounds i32, i32* %17, i64 %16
  %19 = load i32, i32* %18, align 4
  %20 = mul nsw i32 %14, %19
  %21 = add nsw i32 %13, %20
  %22 = load i32, i32* %i, align 4
  %23 = sext i32 %22 to i64
  %24 = load i32*, i32** %3, align 8
  %25 = getelementptr inbounds i32, i32* %24, i64 %23
  store i32 %21, i32* %25, align 4
  br label %26
; <label>:26                             ; preds = %8
  %27 = load i32, i32* %i, align 4
  %28 = add nsw i32 %27, 1
  store i32 %28, i32* %i, align 4
  br label %5
; <label>:29       ; preds = %5           ret void }
```

Instrumented Static IR

```
define void @triad(i32* %a, i32* %b, i32* %c, i32 %s) #0 {
  %print1 = call i32 (i8*, ...) @printf(i8* getelementptr
inbounds ([18 x i8], [17 x i8]* @.str.0, i32 0, i32 0), i32 1);
  %1 = alloca i32*, align 8
  %2 = alloca i32*, align 8
  %3 = alloca i32*, align 8
  %4 = alloca i32, align 4
  %i = alloca i32, align 4
  %SL1 = call i64 @rdtsc()
  store i32* %a, i32** %1, align 8
  %SL2 = call i64 @rdtsc()
  %SL3 = sub i64 %SL2, %SL1
  %SL4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([17 x i8], [17 x i8]* @.str.0, i32 0, i32 0), i64 %SL3)
  %SL5 = call i64 @rdtsc()
  store i32* %b, i32** %2, align 8
  %SL6 = call i64 @rdtsc()
  %SL7 = sub i64 %SL6, %SL5
  %SL8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([17 x i8], [17 x i8]* @.str.0, i32 0, i32 0), i64 %SL7)
  %SL9 = call i64 @rdtsc()
  store i32* %c, i32** %3, align 8
  store i32 %s, i32* %4, align 4
  store i32 0, i32* %i, align 4
  br label %5
; <label>:5                              ; preds = %26, %0
  %print1 = call i32 (i8*, ...) @printf(i8* getelementptr
inbounds ([18 x i8], [17 x i8]* @.str.0, i32 0, i32 0), i32 2);
  %6 = load i32, i32* %i, align 4
  %7 = icmp slt i32 %6, 2
  br i1 %7, label %8, label %29
; <label>:8                              ; preds = %5
  %print1 = call i32 (i8*, ...) @printf(i8* getelementptr
inbounds ([18 x i8], [17 x i8]* @.str.0, i32 0, i32 0), i32 3);
  %9 = load i32, i32* %i, align 4
  %10 = sext i32 %9 to i64
  %11 = load i32*, i32** %1, align 8
  %12 = getelementptr inbounds i32, i32* %11, i64 %10
  %13 = load i32, i32* %12, align 4
  %14 = load i32, i32* %4, align 4
  %15 = load i32, i32* %i, align 4
  %16 = sext i32 %15 to i64
  %17 = load i32*, i32** %2, align 8
  %18 = getelementptr inbounds i32, i32* %17, i64 %16
  %19 = load i32, i32* %18, align 4
  %20 = mul nsw i32 %14, %19
  %21 = add nsw i32 %13, %20
  %22 = load i32, i32* %i, align 4
  %23 = sext i32 %22 to i64
  %24 = load i32*, i32** %3, align 8
  %25 = getelementptr inbounds i32, i32* %24, i64 %23
  store i32 %21, i32* %25, align 4
  br label %26
; <label>:26                             ; preds = %8
  %print1 = call i32 (i8*, ...) @printf(i8* getelementptr
inbounds ([18 x i8], [17 x i8]* @.str.0, i32 0, i32 0), i32 4);
  %27 = load i32, i32* %i, align 4
  %28 = add nsw i32 %27, 1
  store i32 %28, i32* %i, align 4
  br label %5
; <label>:29       ; preds = %5
  %print1 = call i32 (i8*, ...) @printf(i8* getelementptr
inbounds ([18 x i8], [17 x i8]* @.str.0, i32 0, i32 0), i32 5);
  ret void }
```

Dynamic IR Trace

```
%1 = alloca i32*, align 8
%2 = alloca i32*, align 8
%3 = alloca i32*, align 8
%4 = alloca i32, align 4
%i = alloca i32, align 4
store i32* %a, i32** %1, align 8
store i32* %b, i32** %2, align 8
store i32* %c, i32** %3, align 8
store i32 %s, i32* %4, align 4
store i32 0, i32* %i, align 4
%11 = load i32, i32* %i, align 4
%12 = sext i32 %11 to i64
%13 = load i32*, i32** %1, align 8
%14 = getelementptr inbounds i32, i32* %13,
i64 %12
%15 = load i32, i32* %14, align 4
%16 = load i32, i32* %4, align 4
%17 = load i32, i32* %i, align 4
%18 = sext i32 %17 to i64
%19 = load i32*, i32** %2, align 8
%20 = getelementptr inbounds i32, i32* %19,
i64 %18
%21 = load i32, i32* %20, align 4
%22 = mul nsw i32 %16, %21
%23 = add nsw i32 %15, %22
%24 = load i32, i32* %i, align 4
%25 = sext i32 %24 to i64
%26 = load i32*, i32** %3, align 8
%27 = getelementptr inbounds i32, i32* %26, i64 %25
store i32 %23, i32* %27, align 4
%30 = load i32, i32* %i, align 4
%31 = add nsw i32 %30, 1
%11 = load i32, i32* %i, align 4
%12 = sext i32 %11 to i64
%13 = load i32*, i32** %1, align 8
%14 = getelementptr inbounds i32, i32* %13, i64 %12
%15 = load i32, i32* %14, align 4
%16 = load i32, i32* %4, align 4
%17 = load i32, i32* %i, align 4
%18 = sext i32 %17 to i64
%19 = load i32*, i32** %2, align 8
%20 = getelementptr inbounds i32, i32* %19, i64 %18
%21 = load i32, i32* %20, align 4
%22 = mul nsw i32 %16, %21
%23 = add nsw i32 %15, %22
%24 = load i32, i32* %i, align 4
%25 = sext i32 %24 to i64
%26 = load i32*, i32** %3, align 8
%27 = getelementptr inbounds i32, i32* %26, i64 %25
store i32 %23, i32* %27, align 4
%30 = load i32, i32* %i, align 4
%31 = add nsw i32 %30, 1
```

Figure 8: Example of LLVM Graph Construction. This is an example of a graph constructed from a C program followed by the workflow in Fig. 7. One thing to note is that in instrumented static IR, instructions in blue keep track of basic blocks whereas instructions in red measure time for memory operations. We only show partial instrumentation for memory time measurement.

## E. Memory-centric Run-time Mapping

At run-time, processes/clusters generated in Section 4 from each application are mapped onto processors in a NUMA architecture in order to be executed. Depending on the mapping (e.g., A process has to fetch data from the farthest memory bank.), data communication is a performance bottleneck. The goal is to improve the amortized time when slow accesses occur only once in a while and fast local accesses happen frequently.

Therefore, without fully observing the structure of clusters with corresponding physical memories, performance would degrade due to these reasons: (1) Waiting for cache update: The multi-core platforms require the cache coherence protocol to have consistent data over private caches. A process later mapped to a different core may increase the time spent for the cache coherence protocol to fetch a cache line from the previous core. (2) Block memory operations between IOs and memory: Block memory operations in IOs constitute a large overhead in the program execution because a large amount of data are referenced and transferred between caches and main memory banks. (3) Core utilization: In an extreme case, processes may be mapped only onto one core to exploit cache temporal and spatial locality. However, the rest cores remain idle for a long time. Therefore, core utilization is another factor for efficient parallelism in multi-core systems.

In order to improve performance, the run-time mapping should exploit and optimize the parallelism in clusters while considering data communication between clusters and resource utilization. Figure 11 shows three important factors to help design a memory-centric run-time mapping.

1. Clusters which reference the same data structures can be mapped onto one core to prevent the time spent on cache coherence protocols and reduce the number of block operations.

2. Clusters which communicate with each other can be mapped to adjacent processors to improve the amortized time by reducing the number of times on fetching data from

Table 8: Summary and Description of Benchmarks

| **Benchmark** | **Description** | **Input Size** | **Source** |
|---|---|---|---|
| *Dijkstra* | Find the shortest path | 50 nodes | MiBench |
| *FFT* | Compute fast Fourier transform | A vector of size 1024 | OmpSCR |
| *K-means* | Partition data into k clusters | 128 2D tuples | Self-collected |
| *Mandel* | Calculate Mandelbrot set | 4092 points | OmpSCR |
| *MD* | Simulate molecular dynamics | 512 particles | OmpSCR |
| *NN* | Neural networks | Three hidden fully connected layers | Self-collected |
| *Neuron* | A list of neurons with the ReLU function | 64 Neurons | Self-collected |
| *CNN* | Convolutional neural networks | Conv-Pool-Conv-Pool-FC | Self-collected |
| *Strassen8* | Strassen's matrix multiplication | Matrices of size 64 | Self-collected |
| *Strassen16* | Strassen's matrix multiplication | Matrices of size 256 | Self-collected |

Table 9: Data Communication for 1024 Clusters From Different Graph Partitioning Algorithms

| **Datasets** | **CompNet** | **METIS** | **PG** | **W-PG** | **WB-PG** | **Libra** | **W-Libra** | **WB-Libra** |
|---|---|---|---|---|---|---|---|---|
| Dijkstra | 100% | 142% | 53% | 34% | 47% | 39% | 46% | 47% |
| FFT | 100% | 163% | 55% | 42% | 50% | 43% | 46% | 49% |
| K-means | 100% | 132% | 47% | 31% | 42% | 32% | 39% | 43% |
| Mandel | 100% | 117% | 33% | 18% | 28% | 19% | 23% | 27% |
| Md | 100% | 156% | 41% | 25% | 35% | 29% | 31% | 35% |
| NN | 100% | 176% | 52% | 35% | 44% | 39% | 45% | 47% |
| Neuron | 100% | 148% | 38% | 27% | 30% | 31% | 33% | 35% |
| CNN | 100% | 183% | 63% | 44% | 55% | 49% | 55% | 57% |
| Strassen8 | 100% | 145% | 52% | 35% | 44% | 38% | 42% | 46% |
| Strassen16 | 100% | 166% | 53% | 38% | 48% | 41% | 45% | 48% |

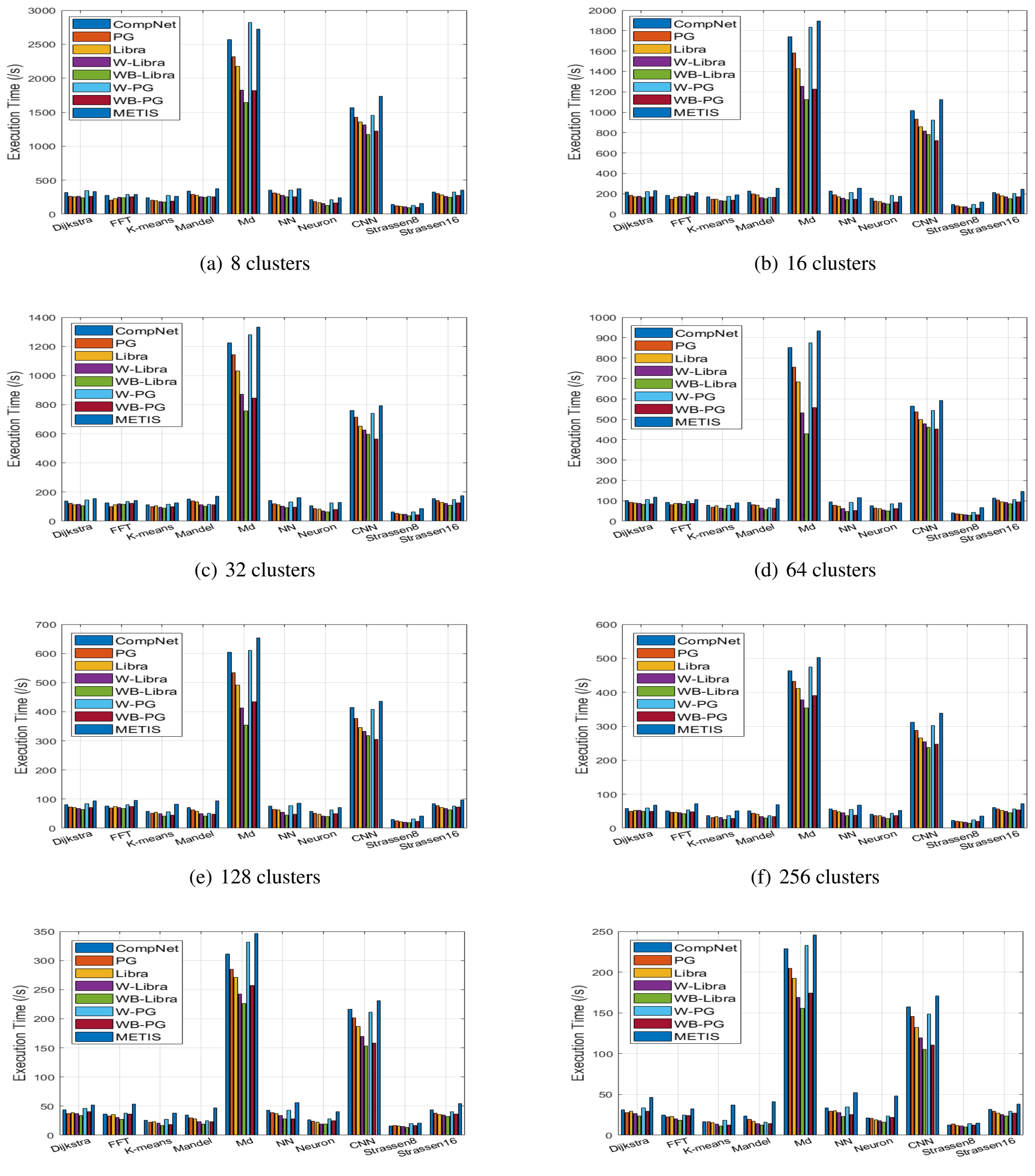
the farthest processor on a NUMA architecture.

3. Clusters which are independent of each other can be mapped to different regions of a multi-core platform (architecture decomposition) to reduce the number of sharing paths of messages.

Therefore, the memory-centric run-time mapping algorithm as shown in Algorithm 2 takes as inputs the clusters, their interactions, and data communication and schedules a mapping from clusters to processors with the objective of improving application performance. The uttermost important criterion for a run-time mapping is the small time complexity. Therefore, we propose a greedy algorithm to achieve high performance with the time complexity of $O(P)$ where $P$ is the number of schedulable clusters. In the algorithm, we first check whether a cluster is ready to schedule, and keep track of all clusters with which this cluster communicates. Next, based on the factor 3, the architecture decomposition is performed to make sure that independent clusters are mapped onto faraway processors to distribute workloads and traffic evenly on hardware communication substrate in a multi-core platform. Then, we calculate the execution time for two clusters with the shared memory to be mapped onto the same processor and different ones, respectively. A mapping of the current cluster is decided based on factors 1 and 3. Mapping clusters with the shared memory onto the same processor could reduce the large overhead for block operations, but at the same time the parallelism may suffer if too many clusters are mapped to a single processor. Therefore, we set an upper threshold of the number of clusters to be mapped to a processor. In the evaluation, the threshold is set to be 4.

**Algorithm 2** Memory-Centric Run-Time Mapping

```
1: INPUT: Clusters and data communication
2: OUTPUT: A mapping from clusters to the architecture
3: Runqueue RQ = $\emptyset$
4: for $cluster$ in $clist$ do
5:   if $cluster\to status$ = $SCHEDULABLE$ then
6:     if C = ClusterFromMem($cluster\to m$) != $\emptyset$ then
7:       $cluster\to p$ = C // Factor 1
8:     end if
9:     if C = ClusterComm($cluster$) != $\emptyset$ then
10:      $cluster\to ipc$ = C // Factor 2
11:    else
12:      $cluster\to ipc$ = $\emptyset$ // Factor 3
13:    end if
14:    RQ.$push$($cluster$)
15:  end if
16: end for
17: Regions = ArchitectureDecomposition()
18: LastCluster = NULL
19: repeat
20:   $cluster$ = RQ.$pop$()
21:   if ClusterFromMem($cluster\to m$)$\to core$ == LastCluster$\to core$ then
22:     // Decide on mapping clusters onto the same processor
23:     if LastCluster$\to core\to num \leq$ threshold then
24:       $cluster\to core$ = $cluster\to p$ // Factor 1
25:       LastCluster$\to core\to num$++
26:     else
27:       $cluster\to core$ = DiffRegion($cluster\to core$)
28:     end if
29:   else if $cluster\to ipc$ != $\emptyset$ then
30:     $cluster\to core$ = Nearby($cluster\to ipc\to core$)
31:   else
32:     $cluster\to core$ = DiffRegion($cluster\to ipc\to core$)
33:   end if
34:   LastCluster = $cluster$
35: until RQ is empty
```



(a) 8 clusters

(b) 16 clusters

(c) 32 clusters

(d) 64 clusters

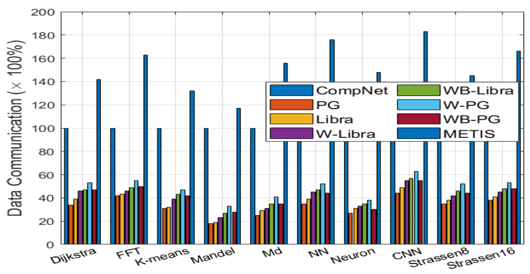
(e) 128 clusters

(f) 256 clusters

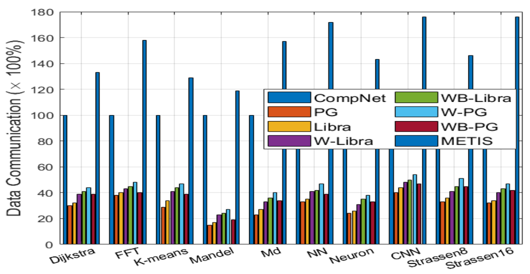
(g) 512 clusters

(h) 1024 clusters

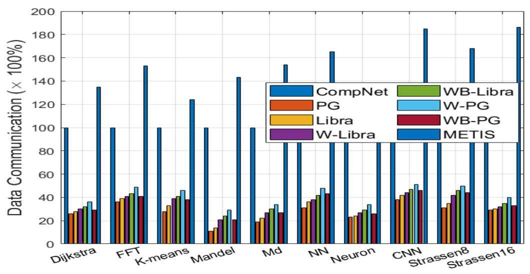
Figure 9: Application Performance From Different Graph Partitioning Algorithms on a Multi-core System



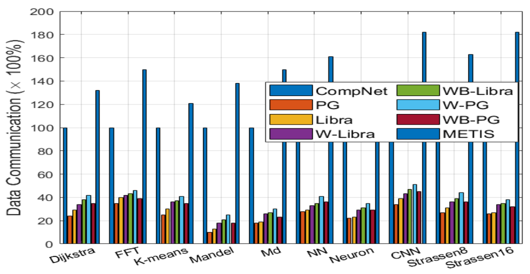
(a) 8 clusters



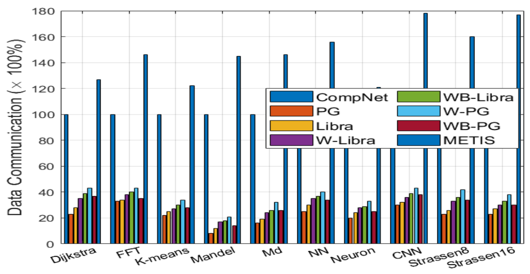
(b) 16 clusters



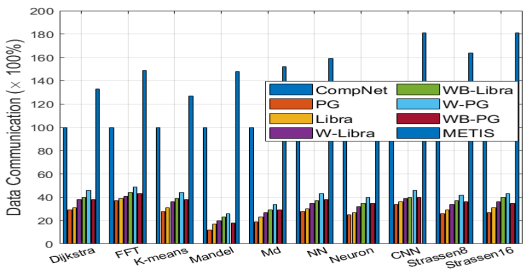
(c) 32 clusters



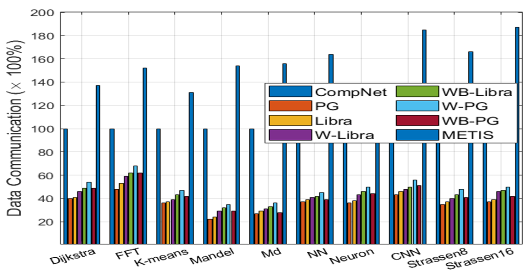
(d)



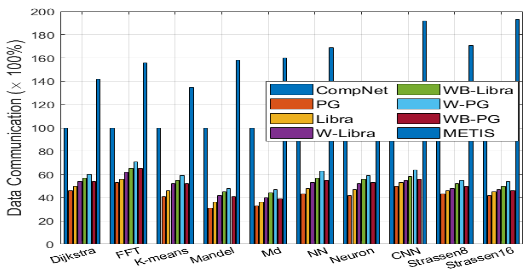
(e) 128 clusters



(f) 256 clusters



(g) 512 clusters



(h) 1024 clusters

Figure 10: Data Communication Cost From Different Graph Partitioning Algorithms in a Multi-core Platform

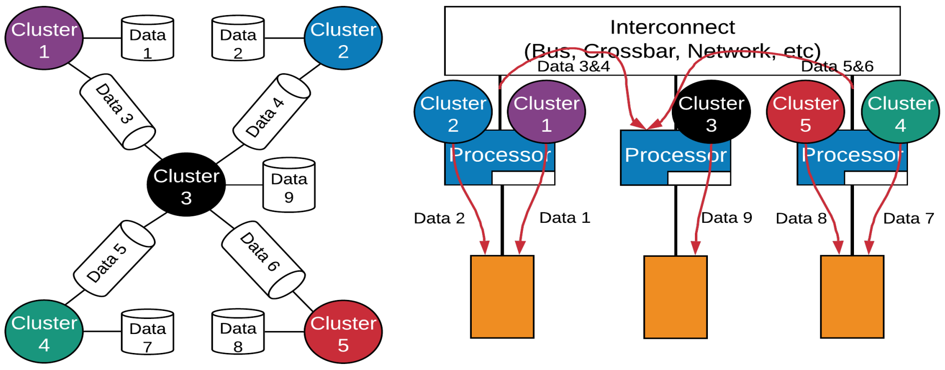


Figure 11: Memory-Centric Run-Time Mapping, which considers memory hierarchy and data communication.