REDUCING SCOPE TO INCREASE SCALABILITY:
DECENTRALIZATION IN GRAPH PROCESSING AND UAV SWARMS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by
R. Ryan McCune

Gregory R. Madey, Co-Director

Tim Weninger, Co-Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2016
REDUCING SCOPE TO INCREASE SCALABILITY:
DECENTRALIZATION IN GRAPH PROCESSING AND UAV SWARMS

Abstract
by
R. Ryan McCune

Many systems are designed with a central controller that is responsible for managing and coordinating all system components. Examples include the master/slave model for distributed computing, and the command and control of Unmanned Aerial Vehicles (UAVs). In such systems, the existence of the central controller presents a bottleneck that limits system scalability. For the system to expand beyond these limitations, the scope of control must be reduced, with communication increased among components. In this dissertation, I investigate improving scalability through reducing scope, demonstrated through two applications.

The first application is the command and control of UAV swarms. Bandwidth limitations and increased software complexity prevent conventional centralized UAV control methods from being applied to the remote operation of many UAVs. Mission planners are investigating the application of swarm intelligence to UAV swarms, where UAVs communicate between one another and the environment to self-organize and solve problems, absent of explicit central command. For one set of projects, I explore the application of swarm intelligence to UAVs, with attention to specific behaviors and quantifiable performance.

The second application is the distributed processing of large-scale graphs. Large graphs, common for Big Data, exceed the memory capacity of conventional ma-
chines and must be partitioned across distributed memory in order to be processed. However, typical centralized graph algorithms presume every node to be randomly accessible, leading to high-latency remote data access and prohibitively large intermediate data structures. Recently developed frameworks provide a platform for executing distributed algorithms on graph data, which reduce algorithmic scope to increase scalability. I comprehensively overview these frameworks, and observe the relationship between graph algorithmic scope and scalability.

Together with these two applications, UAV swarms and distributed graph processing, I illustrate how centralized control must be relaxed, and component communication increased, in order for large systems to overcome limitations of central control and successfully scale.
To my brother Austin
CONTENTS

FIGURES ........................................................................................................ viii

TABLES .......................................................................................................... xiii

ACKNOWLEDGMENTS .................................................................................. xiv

CHAPTER 1: INTRODUCTION ...................................................................... 1
  1.1 Two Applications of Decentralized Control ........................................... 2
    1.1.1 Swarms of Unmanned Aerial Vehicles ........................................... 2
        1.1.1.1 Swarm Intelligence and Emergent Behavior ......................... 3
        1.1.1.2 Agent-Based Modeling of Multi-Agent Systems ................. 3
        1.1.1.3 Dynamic Data-Driven Application Systems ....................... 4
    1.1.2 Large-Scale Graph Processing ...................................................... 4
  1.2 Scalability through Scope Reduction .................................................. 6
  1.3 Dissertation Overview ....................................................................... 6

CHAPTER 2: THINKING LIKE A VERTEX: A SURVEY OF VERTEX-
CENTRIC FRAMEWORKS FOR LARGE-SCALE DISTRIBUTED GRAPH
PROCESSING ................................................................................................. 8
  2.1 Overview ............................................................................................ 8
  2.2 Introduction ......................................................................................... 9
    2.2.1 Bulk Synchronous Parallel ......................................................... 10
    2.2.2 Graph Parallel Systems ............................................................. 11
    2.2.3 TLAV Frameworks .................................................................. 13
  2.3 Overview ........................................................................................... 14
    2.3.1 Example: Single Source Shortest Path in TLAV paradigm ......... 16
  2.4 Four Pillars of TLAV Frameworks .................................................... 18
    2.4.1 Timing ....................................................................................... 20
        2.4.1.1 Synchronous ................................................................ 20
        2.4.1.2 Asynchronous ................................................................ 22
        2.4.1.3 Hybrid ........................................................................ 24
    2.4.2 Communication ......................................................................... 26
        2.4.2.1 Message Passing ............................................................ 27
        2.4.2.2 Shared Memory .............................................................. 29
        2.4.2.3 Active Messages ............................................................ 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.2.4</td>
<td>Message Passing Optimizations</td>
<td>33</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Execution Model</td>
<td>36</td>
</tr>
<tr>
<td>2.4.3.1</td>
<td>Vertex Program Implementation</td>
<td>36</td>
</tr>
<tr>
<td>2.4.3.2</td>
<td>Push vs. Pull</td>
<td>39</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Partitioning</td>
<td>40</td>
</tr>
<tr>
<td>2.4.4.1</td>
<td>Distributed Heuristics</td>
<td>41</td>
</tr>
<tr>
<td>2.4.4.2</td>
<td>Streaming</td>
<td>42</td>
</tr>
<tr>
<td>2.4.4.3</td>
<td>Vertex Cuts</td>
<td>44</td>
</tr>
<tr>
<td>2.4.4.4</td>
<td>Dynamic Repartitioning</td>
<td>46</td>
</tr>
<tr>
<td>2.5</td>
<td>Implementation</td>
<td>48</td>
</tr>
<tr>
<td>2.5.1</td>
<td>System Architecture</td>
<td>49</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Multi-Core Support</td>
<td>49</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Fault Tolerance</td>
<td>50</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Single Machine Architectures</td>
<td>51</td>
</tr>
<tr>
<td>2.6</td>
<td>Alternative Graph Granularity</td>
<td>55</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Subgraph-centric Frameworks</td>
<td>55</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Other Scopes: Paths and Sets</td>
<td>57</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Optimizations</td>
<td>58</td>
</tr>
<tr>
<td>2.7</td>
<td>Related Work</td>
<td>59</td>
</tr>
<tr>
<td>2.8</td>
<td>Conclusions</td>
<td>62</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview</td>
<td>64</td>
</tr>
<tr>
<td>3.2</td>
<td>Related Work</td>
<td>66</td>
</tr>
<tr>
<td>3.3</td>
<td>Breadth-First Search</td>
<td>67</td>
</tr>
<tr>
<td>3.4</td>
<td>Lexicographic Breadth-First Search</td>
<td>68</td>
</tr>
<tr>
<td>3.5</td>
<td>Pre-processing for Hardware</td>
<td>70</td>
</tr>
<tr>
<td>3.6</td>
<td>Method I</td>
<td>70</td>
</tr>
<tr>
<td>3.7</td>
<td>Method II</td>
<td>71</td>
</tr>
<tr>
<td>3.8</td>
<td>Technical Discussion</td>
<td>74</td>
</tr>
<tr>
<td>3.9</td>
<td>Experimental Results</td>
<td>76</td>
</tr>
<tr>
<td>3.10</td>
<td>Conclusion</td>
<td>77</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>81</td>
</tr>
<tr>
<td>4.2</td>
<td>Related Work</td>
<td>82</td>
</tr>
<tr>
<td>4.3</td>
<td>Incorporating Agents with Docker</td>
<td>83</td>
</tr>
<tr>
<td>4.3.1</td>
<td>The Case for Docker within Autonomic Computing</td>
<td>84</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Docker In Practice</td>
<td>85</td>
</tr>
<tr>
<td>4.4</td>
<td>Cognitive Environments Lab</td>
<td>85</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Implementation</td>
<td>87</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Fresh Builds versus Layer-reuse</td>
<td>88</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Local Images versus remote images</td>
<td>90</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Serial versus Parallel</td>
<td>91</td>
</tr>
<tr>
<td>4.5</td>
<td>Network Simulation</td>
<td>92</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Network Models</td>
<td>93</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Evaluation</td>
<td>95</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusions and Future Work</td>
<td>97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Overview</td>
<td>98</td>
</tr>
<tr>
<td>5.2</td>
<td>Introduction</td>
<td>99</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Wireless Rechargeable Sensor Networks</td>
<td>100</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Search-and-Return Behavior</td>
<td>101</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Swarm Intelligent Systems</td>
<td>102</td>
</tr>
<tr>
<td>5.3</td>
<td>Prior Work</td>
<td>103</td>
</tr>
<tr>
<td>5.4</td>
<td>System Overview</td>
<td>104</td>
</tr>
<tr>
<td>5.5</td>
<td>Agent-Based Simulation</td>
<td>105</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusion and Future Work</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Overview</td>
<td>116</td>
</tr>
<tr>
<td>6.2</td>
<td>Introduction</td>
<td>117</td>
</tr>
<tr>
<td>6.3</td>
<td>Background and Related Work</td>
<td>119</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Background</td>
<td>119</td>
</tr>
<tr>
<td>6.3.1.1</td>
<td>Swarms</td>
<td>119</td>
</tr>
<tr>
<td>6.3.1.2</td>
<td>Agent-Based Modeling</td>
<td>120</td>
</tr>
<tr>
<td>6.3.1.3</td>
<td>DDDAS</td>
<td>120</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Related Work</td>
<td>121</td>
</tr>
<tr>
<td>6.4</td>
<td>Technical Approaches</td>
<td>121</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Overview</td>
<td>121</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Swarm Mission Scheduling</td>
<td>122</td>
</tr>
<tr>
<td>6.4.2.1</td>
<td>Global-Local Hybrid Planning and Scheduling</td>
<td>122</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Formation Planning</td>
<td>124</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Communication</td>
<td>127</td>
</tr>
<tr>
<td>6.4.5</td>
<td>Swarm Control Through BOIDS</td>
<td>130</td>
</tr>
<tr>
<td>6.4.5.1</td>
<td>Plume Coverage</td>
<td>131</td>
</tr>
<tr>
<td>6.4.5.2</td>
<td>Target Tracking</td>
<td>132</td>
</tr>
<tr>
<td>6.5</td>
<td>Conclusions and Future Work</td>
<td>133</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Overview</td>
<td>135</td>
</tr>
<tr>
<td>7.2</td>
<td>Applications of Swarm Intelligent Systems</td>
<td>136</td>
</tr>
<tr>
<td>7.3</td>
<td>Agent-Based Modeling of Swarms</td>
<td>138</td>
</tr>
</tbody>
</table>
FIGURES

2.1 Example of Bulk Synchronous Parallel execution with 3 tasks/workers over 4 supersteps. Each task may have varying durations after which messages are passed. The barriers control synchronization across the entire system. ................................................................. 11

2.2 Comparison of the Think Like a Vertex (TLAV) and Bulk Synchronous Parallel (BSP) models of computation. Both models are commonly employed for iterative computation. .................................................. 12

2.3 Computing the Single Source Shortest Path in a graph. Dashed lines between supersteps represent messages (with values listed to the right), and shaded vertices are inactive. Edge weights pictorially included in first layer for Superstep 0, then subsequently omitted. .......................... 18

2.4 Distributed communication patterns for common communication implementations. The sample graph is partitioned across two machines (see Section 2.4.4), with vertices A, B, and C residing on machine p1, and vertices D, E, and F on machine p2. Pregel is represented in (b), GraphLab in (c), PowerGraph in (d), and GRE in (e). .................. 28

2.5 Partition-driven optimization strategies for distributed message passing. The Combiner technique employs both Sender-side and Receiver-side Combiners. ................................................................. 34

3.1 An overview of how the graph is preprocessed for the hardware processing kernel. For every node in the graph, the adjacency list set of each node and its neighbors must be provided in LexBFS order. .... 65

3.2 Sample graph and example execution of LexBFS ............................ 70

3.3 Illustration of Method I. For every vertex, LexBFS is executed on the subgraph induced by a node and its neighbors. .................. 71

3.4 Illustration of Method II. LexBFS is first performed on the whole graph. Then, for each vertex $v$, a subgraph is induced for $\{v \cup N(v)\}$, and LexBFS is performed only on nodes residing an additional hop away from the root of the original LexBFS. ........................................ 74

3.5 Method II runs an initial LexBFS over the whole graph to reduce the number of nodes and vertices traversed by subsequent LexBFS on the subgraphs induced by each vertex. Still, some nodes need to be reordered. How and why some nodes need to be re-ordered are depicted. 75
3.6 Processing random graphs generated by the Erdos-Renyi model with 10,000 nodes.

3.7 Processing scale-free graphs generated by the Barabasi-Albert model with 10,000 nodes.

4.1 Two users interact with the CEL system through speech and motion recognition. The CEL incorporates the autonomic computing principle of self-healing to provide a state-of-the-art cognitive experience.

4.2 Runtimes for serially building and running several containers.

4.3 Runtimes for building in parallel with 2 and 4 virtual CPUs.

4.4 Graph Models and Degree Distribution.

4.5 Small-World Graphs.

4.6 Representation of an example network of 10 agents. Agents are nodes, and directed edges between nodes represent agent dependencies, where the bolded ends of edges designate direction. Edges can be bidirectional. In the network simulation, an agent requires messages from neighbors of inbound edges to operate. The simulation tests how the self-healing of malfunctioning agents impacts a system for random, small-world, and scale-free network topology models.

5.1 Search and Return (SEAR) behavior of Wireless Charge Vehicles (WCV).

5.2 The three-tiered architecture is a recurring MANET design pattern. Adapted for ant systems, nests are mobile base stations, ants represent transport nodes, and food are outer nodes.

5.3 Demonstration of nest update mechanism. In each nest, an array stores a running total of distance traveled by returning ants, indexed by the returning direction. When the total exceeds the threshold (an input parameter) the nest moves one space according to the index. In the above figure, food is black, a nest is gray, the arrow is the path of an ant, and the threshold is 4. (i) An ant returns a distance 3, less than the threshold, (ii) An ant returns a distance 5, (iii) the nest moves right one square, and resets the array.

5.4 Test scenario with square patterns of food placed in each quadrant. Ideal system behavior would minimize the cumulative distance between food and nearest nests by updating nests to the center of the square.

5.5 Threshold sweep for random food placement.

5.6 Bar charts for the cumulative distance of food from nearest nests for nest update thresholds of 1 and 10, for 100 simulations with food deterministically placed in 4 square patterns.
5.7 Screenshots from an agent-based simulation of the SM-Swarm. For this scenario, to start, 4 nests are randomly placed, and 16 food objects are deterministically placed in 4 square clusters. For illustrative purposes, images of candy and mobile base stations overlay the simulation’s visualization objects of food and nests, respectively. Ants are the many black dots. No pheromones have been deposited at the start of the simulation in 1, the beginning of pheromone deposits are visible around the nests in 2, and pheromones are visible in varying shades throughout the environment in 3-6. The simulation overlays a Voronoi Tessellation of the environment based on nest locations, and is a good approximation for the nest an ant would return to after locating a given food object. Reduction of cumulative distance from food to nests is observed over time. In addition to predetermined food locations scenario, simulations were also run for random food placement.

6.1 The GUI of the simulation application, displaying a swarm of 3 UAVs executing 2 missions.

6.2 The Parallel Path algorithm, where agents sweep back and forth in parallel to ensure all mobile targets are found. The shaded area represents overlapping paths.

6.3 Execution of the SWEEP Protocol over time. Agents are placed along the perimeter of a search space, sweeping the outside edges moving inward.

6.4 A central controller assigns stationary Sentry agents to strategic locations, displayed in red, to partition the search space into sub-regions. Trade-offs between the two search formations make partitioning the search region an optimization problem. Implemented in a DDDAS framework, the simulation supports dynamic formation reassignment when UAV failure impacts mission objectives.

6.5 Agent mobility rules of the IcoSystem Swarm Game. Aggressor and Defender agents were implemented with the addition of a single Leader agent. The Leader is assigned as the first helper of every other agent, and traverses predefined waypoints throughout the environment to ensure quality coverage.

6.6 The experimental test-bed incorporating DDDAS architecture. Experiments utilized an agent-based simulation developed in Java using the MASON toolkit. Web services support both the real-life representations of UAVs driving simulation input, as well as the simulation output dynamically steering the real-time measurements of the UAVs.
6.7 Illustrative example of swarm command and control with the BOIDS flocking rule set for a contaminant plume surveillance mission. The left-most image shows UAVs gathering around the highly concentrated center of the plume, controlled with a tight turning radius. In the center, real-time statistics reveal that the UAVs are not adequately dispersed about the plume, so the controller reduces the agent turning radius to cover more of the plume and obtain a greater deviation in measurements, seen right \[171\].

6.8 Screen capture of the target tracking scenario. Gray arrowheads represent target vessels in a river, green airplanes represent Searcher UAVs, orange airplanes represent unassigned Pursuers, while red airplanes represent Pursuers actively tracking \[171\].

7.1 Number of English, non-patent research articles returned for search of agent-based

7.2 Parameter sweeps over the coefficients of Equation \[7.1\].

7.3 Parameters conducive to some clustering, as about 80% of agents converged to between 3 and 5 clusters

7.4 A parameter set very conducive to clustering. After 500 steps nearly all agents are apart of a cluster. The entire swarm converged to 1 cluster at around 4500 steps.

7.5 A parameter set not conducive to clustering. Few clusters were formed, and the majority of agents did not cluster.

7.6 Visualization of the Flocker simulation over different time steps, under parameters Cohesion = 2.0, Adhesion = 0.5 and Randomness = 1.0, corresponding to the run depicted in Figure 3(b). Figure 4(a) is at step 0, 4(b) shows step 500, where most agents are apart of 1 of 6 clusters, and 4(c) is at step 4000, where all agents are in 1 of 2 clusters.

A.1 In the Panait Foraging Model (PFM), ants collectively forage food. Ants begin by searching the environment randomly, depositing pheromones (green) that lead back to the nest. Once food is found, ants follow the green pheromone trail, while depositing a different pheromone that leads to food (blue). Ants are more likely to follow the blue pheromone while searching for food. Over time, the colony uncovers the shortest path to food \[201\].

A.2 Test scenario with square patterns of food placed in each quadrant. Ideal system behavior would minimize the cumulative distance between food and nearest nests by updating nests to the center of the square.
B.1 Computing the Single Source Shortest Path in a graph. Dashed lines between supersteps represent messages (with values listed to the right), and shaded vertices are inactive. Edge weights pictorially included in first layer for Superstep 0, then subsequently omitted.

B.2 Utilizing both vertex-centric and centralized algorithms for connected component computation reduces the number of supersteps compared to vanilla vertex programming.

B.3 Results from [262] show how execution time and network messages are reduced by using vertex computing only between vertices that reside on different machines. For subgraphs within a worker, a centralized algorithm is used. Results show improved performance for the hybrid method (HM and GM) is superior to the pure vertex-oriented method (VM) for two different methods of partitioning (HP and GP).

B.4 Results from [295] show how block-centric computing (Block-Centric), where centralized algorithms are executed on the blocks of graphs on a worker and then a vertex-program is used to share results between blocks, is superior in performance to Pregel-like Vertex-Centric computing.

B.5 From top to bottom: A) an example graph, B) an example distributed algorithm where each node is processor/worker, C) 4 workers with 3 nodes each, D) 2 workers with 6 nodes each, E) 1 worker with the whole graph. If it’s assumed that the capacity of a worker can support 6 nodes and also store intermediate computation, then the setup in D is optimal, because a centralized algorithm can execute on each worker and results can be shared between workers through a distributed algorithm. The setup in C is possible, but less efficient than C because there would be more communication overhead. The setup in B represented a true distributed algorithm, but likewise is less efficient than C or D because no centralized algorithm can be utilized.
### TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>EXECUTION TIMING MODEL OF SELECTED FRAMEWORKS</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>FEATURE SUMMARY FOR DYNAMIC REPARTITIONING FRAMEWORKS</td>
<td>48</td>
</tr>
<tr>
<td>2.3</td>
<td>SINGLE MACHINE FRAMEWORKS</td>
<td>52</td>
</tr>
<tr>
<td>2.4</td>
<td>FRAMEWORKS OF ALTERNATIVE SCOPE</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>EXPERIMENTAL RESULTS FOR REAL-WORLD GRAPHS</td>
<td>77</td>
</tr>
<tr>
<td>3.2</td>
<td>RANDOM GRAPH PROFILES</td>
<td>80</td>
</tr>
<tr>
<td>3.3</td>
<td>SCALE-FREE GRAPH PROFILES</td>
<td>80</td>
</tr>
<tr>
<td>4.1</td>
<td>RUNNING A SINGLE CONTAINER WITH LAYER RE-USE</td>
<td>89</td>
</tr>
<tr>
<td>4.2</td>
<td>PULL AND RUN IMAGES FROM REMOTE REPOSITORY</td>
<td>91</td>
</tr>
<tr>
<td>5.1</td>
<td>PARAMETERS FOR THE SM-SWARM AGENT-BASED SIMULATION</td>
<td>109</td>
</tr>
<tr>
<td>A.1</td>
<td>COMPLEXITY OF LOCAL VS. GLOBAL BEHAVIOR FOR ANT MODEL</td>
<td>175</td>
</tr>
<tr>
<td>A.2</td>
<td>COMPLEXITY OF LOCAL VS. GLOBAL BEHAVIOR FOR BOOMERANG SWARM</td>
<td>178</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to acknowledge the gracious support of the Air Force Office of Scientific Research for the DDDAS grant, the GAANN Fellowship provided by the Department of Education via the Notre Dame Computer Science Department, International Business Machines for the prestigious IBM Ph.D. Fellowship, the Cognitive Science group at the IBM T.J. Watson Research Center, the hardware accelerators group at IBM Zurich, and my long-time advisor Dr. Madey.
CHAPTER 1

INTRODUCTION

Centralized systems, predominant across many technological architectures, consist of a distributed system of many components, or computational agents, collectively controlled by a central coordinator. Examples include the master-slave model for distributed computing [174, 181], and the command and control for swarms of Unmanned Aerial Vehicles (UAVs) by a remote operator [178]. Centralized systems offer many advantages including intuitive control and implementation, but are difficult to scale for larger systems [167, 88].

A central controller presents an inevitable bottleneck as a system grows. The controller of a centralized system, whether a master compute node or the remote operator of many UAVs, has practical constraints that prevents any system from scaling infinitely. For a system to scale beyond the inherent limits imposed by a central controller, the system must adopt a decentralized approach, meaning the scope of execution control is reduced, which allows the system to scale.

Increasing scalability through reduction of control scope is illustrated through two applications: i) the command and control of UAV swarms, and ii) large-scale graph processing. In both applications, the existence of a central coordinator limits the ability of the system to grow beyond a certain size while maintaining acceptable performance. In this dissertation, I will explain how, for each application, the existence of a central controller limits scalability, and then demonstrate how scalability is achieved by reducing the scope of control. In this introduction, I will first overview each application, presenting limits to scalability and the decentralized solution. Then
I will present my thesis, and outline the remaining chapters of this dissertation.

1.1 Two Applications of Decentralized Control

I present two applications that illustrate how relaxing the scope of control enables the system to scale beyond the limits imposed by a central coordinator. The first application is the command and control of UAV swarms. The second application is the distributed processing of large-scale graphs. To support my thesis, I first provide an overview of each application, including the scalability problem arising from centralization, and the scalable, decentralized solution. Further detail of each application is provided in respective chapters.

1.1.1 Swarms of Unmanned Aerial Vehicles

Unmanned Aerial Vehicles (UAVs) are remotely operated aerial vehicles that do not require an on-board pilot. Technological advancement has reduced the size and cost of UAVs so that several UAVs can be deployed at once. The commonly held belief is that teams of UAVs can cooperate to accomplish more complex objectives than each UAV operating independently of one another [40].

However, UAVs have typically been controlled by one or more highly-trained remote operators, which cannot scale to teams comprised of many UAVs. Furthermore, centralized control approaches frequently lead to exponential increases in communication bandwidth requirements and software complexity [88], meaning bandwidth and software complexity limit how many UAVs a central coordinator can control. In response, new techniques based on swarm-inspired self-organization are under investigation so a remote operator can control a UAV swarm [273].

I present a swarm-intelligent behavior for the command and control of UAVs in Chapter 5. The behavior is evaluated using an agent-based model, and incorporates the Dynamic Data-Driven Application Systems (DDDAS) paradigm to accomplish
the provided objective [65]. Below I briefly overview swarm intelligence, agent-based 
modeling, and DDDAS.

1.1.1.1 Swarm Intelligence and Emergent Behavior

Decentralized control of UAVs is achieved through swarm intelligence. Swarm 
intelligence is exhibited in systems that utilize emergent behavior to solve problems 
[29]. Emergence can be characterized as a system that is greater than the sum of 
parts, where the relationship between the global behavior of the system and its sub-
components is not well understood. Common examples are cited in nature, such as 
the flocking of birds, or foraging of ants [73]. While each agent behavior is relatively 
simple and local in scope, collectively a greater system level behavior emerges from 
the interaction between agents and the environment.

Swarms demonstrate advantageous system properties. Swarms lack a central co-
ordinator, as each agent executes a simple local behavior that collectively lead to an 
emergent global behavior. Swarms are both scalable and resilient, as the local behav-
ior of each agent allows an agent to be added or removed without significantly im-
pacting overall performance [104]. The emergent problem solving behavior of swarms 
means that the system can collectively solve complex problems, despite the relatively 
simple behavior of each agent. Swarms are also adaptable to dynamic conditions.

1.1.1.2 Agent-Based Modeling of Multi-Agent Systems

Agent-based modeling is an approach for modeling a multi-agent system. A multi-
agent system is comprised of many agents that interact with one another, as well 
as with the environment [289]. An agent can be characterized as an autonomous, 
decision-making entity capable of its own independent behavior [288]. Agents in a 
multi-agent system can be homogeneous or heterogeneous.

Agent-based modeling and simulation is an approach to modeling a multi-agent
system at the agent-level, as opposed to the system level. The modeler designs and programs behaviors for the agents and the environment, and a simulation engine drives interactions between system components, where the global system behavior is perceived by the modeler. Agent-based modeling is a "bottom-up" approach to modeling a system, as opposed to conventional "top-down" approaches, such as equation-based models \[15\]. Agent-based modeling is often employed for modeling swarms \[166\].

1.1.1.3 Dynamic Data-Driven Application Systems

In my UAV swarm application, the Dynamic Data-Driven Application System (DDDAS) paradigm is incorporated for enhancing swarm control by supplementing decision-making with dynamic agent-based simulations. DDDAS entails the ability to incorporate additional data into an executing application \[65\] \[66\]. The data helps drive the decision-making process, which in turn impacts the measurement of future real-time data, affecting future simulations. This synergistic feedback control loop between the simulations and an executing application can tremendously improve analytic and predictive capabilities of the real-time system. Developing a swarm application with DDDAS concepts can alleviate control problems arising from the non-linearity between the agent-level control behavior and application-level emergent behavior.

1.1.2 Large-Scale Graph Processing

The second application of a decentralized approach is large-scale graph processing. Graphs, or networks, are data structures that capture data relationships by representing objects as nodes and objects’ relations as edges. Graph processing is of increasing interest and application \[18\] \[154\]. With the growing amount of quantitative information available, graphs have become a prevalent data structure, and the
processing, or analysis, of graphs can yield useful properties. However, as Big Data and other technological trends have led to graphs of increasing size, processing large graphs is becoming increasingly difficult [167, 174].

Graphs are unstructured and can contain connections between any two nodes. As many fundamental graph operations involve traversing an edge from one node to another, all graph data must reside in memory to be randomly accessible, or otherwise face prohibitively slow communication or external memory access. For large-scale graphs, comprised of billions of nodes and trillions of edges, or more, maintaining the whole graph in random access memory may only be practical on high-end servers, not commodity machines [244].

A graph can be distributed, but aside from the non-trivial problem of efficiently partitioning the graph for distributed memory [251], the aggregate result of a centralized graph algorithm must still be computed and managed by a central node. For distributed memory, this requires repeated high-latency calls to remotely located data, which is orders of magnitude longer than local computation, and renders many conventional graph algorithms impractical to execute at scale. The size and requirements of intermediate data structures are also of critical consideration. Moreover, graph computation is notoriously difficult to distribute and parallelize [167], owed in part to irregular data access patterns of graph computation, which further constrains execution. While graph data can be partitioned for distributed memory, it does not alleviate problems arising from the executing a graph algorithm from one central machine. Maintaining a centralized, global computational perspective, where every node is assumed randomly accessible, is not practical for distributed graphs.

In response, a new framework was recently introduced that reduces the scope of an algorithm to increase scalability. Beginning with Pregel [174], new frameworks have been developed that provide a programming interface for "thinking like a vertex," where the scope of a graph algorithm is at the vertex level, instead of the graph.
These frameworks provide a platform for executing distributed algorithms on graph data, which, like agents in a swarm, use only local information for program execution. Scalability is increased in these systems by decreasing the scope of the algorithm. I present a comprehensive survey of these frameworks in Chapter 2.

1.2 Scalability through Scope Reduction

Centralized systems, where a central coordinator controls system components, are inherently limited in the size the systems can grow. This limitation is inherent in the system controller. For the command and control of UAVs, bandwidth and software complexity prevent a remote operator from managing a large number of UAVs. In graph processing, while graph data can be partitioned to distributed memory, maintaining a central algorithmic perspective, where every node is assumed randomly accessible, is impractical for graphs of scale. Both of these applications illustrate how bottlenecks arise in maintaining control over the whole system.

In order to surpass the limits inherent to centralized control, a decentralized approach can be implemented by reducing the scope of control. My thesis is:

If a system is centrally controlled, and growth is reducing performance, then scalability can be improved by reducing control scope and distributing control to multiple, communicating coordinators.

This thesis is illustrated through two applications, swarms and distributed algorithms. Together the applications illustrate how decentralization improves scalability.

1.3 Dissertation Overview

This dissertation begins with a comprehensive survey of Pregel-like frameworks, presented in Chapter 2, where reducing scope through distributed algorithms improves scalability. A centralized multi-agent system of web service agents is pre-
presented in Chapter 4 which utilizes self-healing functionality for increased autonomy and reliability.

A swarm intelligent behavior for UAVs is presented in Chapter 5. Another swarm intelligent behavior I developed is presented in Chapter 6 within the context of related projects from my group. As swarm intelligence and emergent behavior is difficult to measure, work in Chapter 7 brings together the local and global behaviors of swarms to quantify swarm performance.

Work in Chapter 2 led to further work in graph processing, and Chapter 3 explores reducing the high data access to computation ratio in graph processing, by eliminating random access through pre-processing. This work is related to future graph processing work discussed in the Chapter 8 and addresses parallelism but not scalability.

Two appendices are provided to support chapter content. Appendix A supports Chapter 7 by presenting a model for quantifying swarm performance of ant systems by combing agent-based models and Pregel-like systems. Appendix B supports Chapter 2 by refining the connection between scope and scalability through hybrid frameworks.
2.1 Overview

The vertex-centric programming model is an established computational paradigm recently incorporated into distributed processing frameworks to address challenges in large-scale graph processing. Billion-node graphs that exceed the memory capacity of commodity machines are not well-supported by popular Big Data tools like MapReduce, which are notoriously poor-performing for iterative graph algorithms such as PageRank. In response, a new type of framework challenges one to “think like a vertex” (TLAV) and implements user-defined programs from the perspective of a vertex rather than a graph. Such an approach improves locality, demonstrates linear scalability, and provides a natural way to express and compute many iterative graph algorithms. These frameworks are simple to program and widely applicable, but, like an operating system, are composed of several intricate, interdependent components, of which a thorough understanding is necessary in order to elicit top performance at scale. To this end, the first comprehensive survey of TLAV frameworks is presented. In this survey, the vertex-centric approach to graph processing is overviewed, TLAV

---

1 Published in ACM Computing Surveys, Volume 48 Issue 2, November 2015
2 Presented at DDDAS Primary Investigators Meeting, TJ Watson Research Center 2014
3 Collaborators Tim Weninger and Greg Madey
frameworks are deconstructed into four main components and respectively analyzed, and TLAV implementations are reviewed and categorized.

2.2 Introduction

The proliferation of mobile devices, ubiquity of the web, and plethora of sensors has led to an exponential increase in the amount data created, stored, managed, and processed. In March 2014, an IBM report claimed that 90% of the world’s data had been generated in the last two years [135]. Big Data characterizes the problems faced by conventional analytics systems with this dramatic expansion of data volume, velocity, and variety.

To address the challenges posed by Big Data, analytical systems are shifting from shared, centralized architectures to distributed, decentralized architectures. The MapReduce framework, and its open-source variant, Hadoop, exemplifies this effort by introducing a programming model to facilitate efficient, distributed algorithm execution while abstracting away lower-level details [67]. Since inception, the Hadoop/MapReduce ecosystem has grown considerably in support of related Big Data tasks.

However, these distributed frameworks are not suited for all purposes, in many cases can even result in poor performance [189, 56, 120]. Algorithms that make use of multiple iterations, especially those using graph or matrix data representations, are particularly poorly suited for popular Big Data processing systems.

Graph computation is notoriously difficult to scale and parallelize, often due to inherent interdependencies within graph data [167]. As Big Data drives graph sizes beyond the memory capacity of a single machine, data must be partitioned to out-of-memory storage or distributed memory. However, for sequential graph algorithms, which require random access to all graph data, poor locality and the indivisibility of the graph structure cause time- and resource-intensive pointer-chasing between
storage mediums in order to access each datum.

In response to these shortcomings, new frameworks based on the vertex-centric programming model have been developed with the potential to transform the ways in which researchers and practitioners approach and solve certain problems [174]. Vertex-centric computing frameworks are platforms that iteratively execute a user-defined program over vertices of a graph. The user-defined vertex function typically includes data from adjacent vertices or incoming edges as input, and the resultant output is communicated along outgoing edges. Vertex program kernels are executed iteratively for a certain number of rounds, or until a convergence property is met. As opposed to the randomly-accessible, “global” perspective of the data employed by conventional shared-memory sequential graph algorithms, vertex-centric frameworks employ a local, vertex-oriented perspective of computation, encouraging practitioners to “think like a vertex” (TLAV).

The first published TLAV framework was Google’s Pregel system [174], which, based off of Valiant’s Bulk Synchronous Parallel (BSP) model [268], employs synchronous execution. While not all TLAV frameworks are synchronous, these frameworks are first introduced here within the context of BSP in order to provide foundational understanding of TLAV concepts.

2.2.1 Bulk Synchronous Parallel

After spending a year with Bill McColl at Oxford in 1988, Les Valiant published the seminal paper on the Bulk Synchronous Parallel (BSP) computing model [268] for guiding the design and implementation of parallel algorithms. Initially touted as “A Bridging Model for Parallel Computation,” the BSP model was created to simplify the design of software for parallel hardware, thereby “bridging” the gap between high-level programming languages and multi-processor systems.

As opposed to distributed shared memory or other distributed systems abstrac-
Figure 2.1: Example of Bulk Synchronous Parallel execution with 3 tasks/workers over 4 supersteps. Each task may have varying durations after which messages are passed. The barriers control synchronization across the entire system.

Introductions, BSP makes heavy use of a message passing interface (MPI) which avoids high latency reads, deadlocks and race conditions. BSP is, at the most basic level, a two step process performed iteratively and synchronously: 1) perform task computation on local data, and 2) communicate the results, and then repeat the two steps. In BSP each compute/communicate iteration is called a superstep, with synchronization of the parallel tasks occurring at the superstep barriers, depicted in Figure 2.1.

2.2.2 Graph Parallel Systems

Introduced in 2010, the Pregel system \cite{174} is a BSP implementation that provides an API specifically tailored for graph algorithms, challenging the programmer to “think like a vertex.” Graph algorithms are developed in terms of what each vertex has to compute based on local vertex data, as well as data from incident edges and adjacent vertices. The Pregel framework, as well other synchronous TLAV implementations, split computation into BSP-style supersteps. Analogous to “components” in BSP \cite{268}, at each superstep a vertex can execute the user-defined vertex function and then send results to neighbors along graph edges. Supersteps always end with a synchronization barrier, shown in Figure 2.1 which guarantees that messages sent in a given superstep are received at the beginning of the next superstep. Unlike the original BSP model, vertices may change status between active and inactive, depend-
ing on the overall state of execution. Pregel terminates when all vertices halt and no more messages are exchanged.

A comparison of TLAV frameworks and BSP is presented in Figure 2.2. BSP employs a general model of broad applicability, including graph algorithms at varying levels of granularity. Underlying BSP execution is the global synchronization barrier among distributed processors. TLAV frameworks utilize a vertex-centric programming model, and while Pregel and its derivatives employ BSP-founded synchronous execution, other frameworks implement asynchronous execution, which has been demonstrated to improve performance in some instances [291].

In contrast to TLAV and BSP, MapReduce does not natively support iterative algorithms. Several recent frameworks have extended the MapReduce model to support iterative execution [119], but for iterative graph algorithms, the graph topological data, which remains static, must be transferred from mappers to reducers, resulting in significant network overhead that renders iterative MapReduce frameworks uncompetitive with TLAV frameworks [119]. A theoretical comparison between MapReduce and BSP is presented in [199].
2.2.3 TLAV Frameworks

Since Pregel, several TLAV frameworks have been proposed that either employ conceptually alternative framework components (such as asynchronous execution), or improve upon the Pregel model with various optimizations. This survey provides the first comprehensive examination into TLAV framework concepts, and makes these other contributions:

1. Analyzes 4 principle components in the design of vertex programs execution in TLAV frameworks, identifying the trade-offs in component implementations and providing data-driven discussion

2. Overviews approaches related to TLAV system architecture, including fault tolerance on distributed systems and novel techniques for large-scale processing on single-machines

3. Discusses how the scalability of a graph algorithm varies inversely with the algorithm’s scope, illustrated by vertex-centric and related subgraph-centric, or hybrid, frameworks

This article is organized as follows: First, Section 2.3 overviews the vertex-centric programming model, including an example program and execution. Section 2.4 presents the four major design decisions, or pillars, of the vertex-centric model. Section 2.5 presents details for distributed implementation, as well as novel techniques utilized by TLAV frameworks that enable large-scale graph processing on a single machine. Section 2.6 presents subgraph-centric, or hybrid, frameworks, that adopt a computational scope of the graph that is greater than a vertex (TLAV) but less than the entire graph. Section 2.7 discusses related work. Finally, Section 2.8 presents a summary, conclusions, and directions for future work.

First, a brief note on terminology: The TLAV paradigm is described interchangeably as vertex-centric, vertex-oriented, or think-like-a-vertex. A vertex program kernel refers to an instance of the user-defined vertex program, function, or process that is executed on a particular vertex. A graph is a data structure made up of vertices and edges, both with (potentially empty) data properties. As in the literature, graph and
network may be used interchangeably, as may node and vertex, and edge and link. Network may also refer to hardware connecting two or more machines, depending on context. A worker refers to a slave machine in the conventional master-worker architectural pattern, and a worker process is the program that governs worker behavior, including, but not limited to, execution of vertex programs, inter-machine communication, termination, check-pointing, etc. Graphs are assumed to be directed without loss of generality.

2.3 Overview

Graph processing is transitioning from centralized to decentralized design patterns. Sequential, shared-memory graph algorithms are inherently centralized. Conventional graph algorithms, such as Dijkstra’s shortest path [71] or betweenness centrality [85], receive the entire graph as input, presume all data is randomly accessible in memory (i.e., graph-omniscient algorithms), and a centralized computational agent processes the graph in a sequential, top-down manner. However, the unprecedented size of Big Data-produced graphs, which may contain hundreds of billions of nodes and occupy terabytes of data or more, exceed the memory capacity of standard machines. Moreover, attempting to centrally compute graph algorithms across distributed memory results in unmanageable pointer-chasing [167]. A more local, decentralized approach is required for processing graphs of scale.

Think like a vertex frameworks are platforms that iteratively execute a user-defined program over vertices of a graph. The vertex program is designed from the perspective of a vertex, receiving as input the vertex’s data as well as data from adjacent vertices and incident edges. The vertex program is executed across vertices of the graph synchronously, or may also be executed asynchronously. Execution halts after either a specified number of iterations, or all vertices have converged. The vertex-centric programming model is less expressive than conventional graph-
omniscient algorithms, but is easily scalable with more opportunity for parallelism.

The frameworks are founded in the field of distributed algorithms. Although vertex-centric algorithms are local and bottom-up, they have a provable, global result. TLAV frameworks are heavily influenced by distributed algorithms theory, including synchronicity and communication mechanisms [169]. Several distributed algorithm implementations, such as distributed Bellman-Ford single-source shortest path [169], are used as benchmarks throughout the TLAV literature. The recent introduction of TLAV frameworks has also spurred the adaptation of many popular Machine Learning and Data Mining (MLDM) algorithms into graph representations for high-performance TLAV processing of large-scale data sets [161].

Many graph problems can be solved by both a sequential, shared-memory algorithm as well as a distributed, vertex-centric algorithm. For example, the PageRank algorithm for calculating web-page importance has a centralized matrix form [200] as well as a distributed, vertex-centric form [174]. The existence of both forms illustrates that many problems can be solved in more than one way, by more than one approach or computational perspective, and deciding which approach to use depends on the task at hand. While the sequential, shared-memory approach is often more intuitive and easier to implement on a single machine or centralized architecture, the limits of such an approach are being reached.

Vertex programs, in contrast, only depend on data local to a vertex, and reduce computational complexity by increasing communication between program kernels. As a result, TLAV frameworks are highly scalable and inherently parallel, with manageable inter-machine communication. For example, runtime on the Pregel framework has been shown to scale linearly with the number of vertices on 300 machines [174]. Furthermore, TLAV frameworks provide a common interface for vertex-program execution, abstracting away low-level details of distributed computation, like MPI, allowing for a fast, re-usable development environment. A paradigm shift from centralized
to decentralized approaches to problem solving is represented by TLAV frameworks.

2.3.1 Example: Single Source Shortest Path in TLAV paradigm

The following describes a simple vertex program that calculates the shortest paths from a given vertex to all other vertices in a graph. In contrast to this distributed implementation example, consider a centralized, sequential, shared-memory, or “graph-omniscient,” solution to the single-source shortest path algorithm known as Djikstra’s algorithm [70] or the more general Bellman-Ford algorithm [20].

Both Dijkstra’s and the Bellman-Ford algorithms are based on repeated relaxations, which iteratively replace distance estimates with more accurate values until eventually reaching the solution. Both variants are have a superlinear time complexity: Dijkstra’s runs in $O(|E| \log |E| + |V|)$ and Bellman-Ford’s runs in $O(|E| \times |V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices in the graph and typically $|E| \gg |V|$. Perhaps more importantly, both procedural, shared-memory algorithms keep a large state matrix resulting in a space complexity of $O(|V|^2)$.

In contrast, to solve the same single-source shortest path problem in the TLAV programming model, a vertex program need only pass the minimum value of its incoming edges to its outgoing edges during each superstep. This algorithm, considered a distributed version of Bellman-Ford [169], is shown in Alg. 1. The computational complexity of each vertex program kernel is less than that of the sequential solution, however a new dimension is introduced in terms of the communication complexity, or the messaging between vertices [169]. For TLAV implementation, a user need only to write the inner-portion of Alg. 1 denoted by line numbers; the outermost loop and the parallel execution is handled by the framework. Because lines 1-10 are executed on the each vertex these lines are known as the vertex program.

The TLAV-solution to the single source shortest path problem has surprisingly few lines of code, and understating its execution requires a different way of thinking.
Algorithm 1: Single Source Shortest Path for a Synchronized TLAV Framework

**input:** A graph \((V, E) = G\) with vertices \(v \in V\) and edges from \(i \rightarrow j\) s.t. \(e_{ij} \in E\), and starting point vertex \(v_s \in V\)

**foreach** \(v \in V\) do shrtest\_path\_len\_v \(\leftarrow \infty\); /* initialize each vertex data to \(\infty\) */

send \((0, v_s)\); /* to activate, send msg of 0 to starting point */

repeat /* The outer loop is synchronized with BSP-styled barriers */

**foreach** \(v \in V\) do in parallel /* vertices execute in parallel */

/* vertices inactive by default; activated when msg received */

/* compute minimum value received from incoming neighbors */

\[
\text{minIncomingData} \leftarrow \min(\text{receive (path\_length)});
\]

/* set current vertex-data to minimum value */

if minIncomingData < shrtest\_path\_len\_v then

shrtest\_path\_len\_v \(\leftarrow\) minIncomingData;

**foreach** \(e_{vj} \in E\) do /* send shortest path + edge weight to outgoing edges */

path\_length \(\leftarrow\) shrtest\_path\_len\_v + weight\_e;

send (path\_length, j);

end

end

until no more messages are sent;

Figure 2.3 depicts the execution of Alg. 1 for a graph with 4 vertices and 6 weighted directed edges. Only the source vertex begins in an active state. In each superstep, a vertex processes its incoming messages, determines the smallest value among all messages received, and if the smallest received value is less than the vertex’s current shortest path, then the vertex adopts the new value as its shortest path, and sends the new path length plus respective edge weights to outgoing neighbors. If a vertex does not receive any new messages, then the vertex becomes inactive, represented as a shaded vertex in Figure 2.3 Overall execution halts once no more messages are sent and all vertices are inactive.

With this example providing insight into TLAV operation, particularly the synchronous message-passing model of Pregel, the survey continues by more completely detailing TLAV properties and categorizing different TLAV frameworks.
Superstep 0
message values = 2
and 4

Superstep 1
message values = 4, 3,
and 8

Superstep 2
message values = 6
and 7

Superstep 3
Complete, no new messages

Figure 2.3: Computing the Single Source Shortest Path in a graph. Dashed lines between supersteps represent messages (with values listed to the right), and shaded vertices are inactive. Edge weights pictorially included in first layer for Superstep 0, then subsequently omitted.

2.4 Four Pillars of TLAV Frameworks

A TLAV framework is software that supports the iterative execution of a user-defined vertex programs over vertices of a graph. Frameworks are composed of several interdependent components that drive program execution and ultimate system performance. These frameworks are not unlike an analytic operating system, where component design decisions dictate how computations for a particular topology utilize the underlying hardware.

This section introduces the four principle pillars of TLAV frameworks. They are:

1. Timing - How user-defined vertex programs are scheduled for execution
2. Communication - How vertex program data is made accessible to other vertex programs
3. Execution Model - Implementation of vertex program execution and flow of data
4. Partitioning - How vertices of the graph, originally in storage, are divided up to be stored across memory of the system’s multiple worker machines

\[^4\]TLAV systems generally distribute a graph across multiple machines because of the graph’s prohibitive size. However, regarding the categorization of “single machine frameworks” in Section 2.5.4 while some TLAV frameworks are implemented for a single machine without the specific
The discussion proceeds as follows: the timing policy of vertex programs is presented in Subsection 2.4.1, where system execution can be synchronous, asynchronous, or hybrid. Communication between vertex programs is presented in Subsection 2.4.2, where intermediate data is shared primarily through message-passing or shared-memory. The implementation of vertex program execution is presented in Subsection 2.4.3, which overviews popular models of program execution and demonstrates how a particular model implementation impacts execution and performance. Finally, partitioning of the graph from storage into distributed memory is presented in Subsection 2.4.4.

Each pillar is heavily interdependent with other pillars, as each design decision is tightly integrated and strongly influenced by other design decisions. While each pillar may be understood through a sequential reading of the information provided, a more efficient, yet thorough understanding may be achieved by freely forward- and cross-referencing other pillars, especially when related sections are cited. The inter-relation of the four pillars is unavoidable and indivisible, not unlike a graph data structure itself. The difficulty of independently describing each pillar certainly reflects the challenge of processing a vertex in which a given result depends on the concurrent processing of neighboring vertices. This survey is restricted to a sequential presentation of information in the form of a chapter. However, each pillar, though unique, depends on, and may only be described in relation to, other pillars, so a sufficient understanding of any given pillar may only be achieved by understanding all pillars of a TLAV framework, collectively. Thus one may begin to understand the challenges of processing graphs as in Section 2.2, Section 2.3, and [167].

intention of developing for a non-distributed environment (e.g. the framework is first developed for a single machine before developing the framework for a distributed environment, like the original GraphLab [which published a distributed version 2 years later, see Section 2.4.1.2 or GRACE [see Section 2.4.1.3]), the single-machine frameworks presented in Section 2.5.4 are frameworks that implement particularly novel methods with the stated objective of processing, on a single machine, graphs of size that exceed the single machine’s memory capacity. These single machine frameworks still partition the graph, using framework-specific methods detailed in the respective section.
2.4.1 Timing

In TLAV frameworks, the scheduling and timing of the execution is separate from the logic of the vertex program. The timing of a framework characterizes how active vertices are ordered by the scheduler for computation. Timing can be synchronous, asynchronous, or a hybrid of the two models. Frameworks that represent the different fundamental timing models are presented in Table 2.1.

2.4.1.1 Synchronous

The synchronous timing model is based on the original bulk synchronous parallel (BSP) processing model discussed above. In this model, active vertices are executed conceptually in parallel over one or more iterations, called supersteps. Synchronization is achieved through a global synchronization barrier situated between each superstep that blocks vertices from computing the next superstep until all workers complete the current superstep. Each worker coordinates with the master to progress to the next superstep. Synchronization is achieved because the barrier ensures that each vertex within a superstep has access to only the data from the previous superstep. Within a single processing unit, vertices can be scheduled in a fixed or random order because the execution order does not affect the state of the program. The global synchronization barrier introduces several performance trade-offs.

Synchronous systems are conceptually simple, demonstrate scalability, and perform exceptionally well for certain classes of algorithms. While not all TLAV programs consistently converge to the same values depending on system implementation, synchronous systems are almost always deterministic, making synchronous applications easy to design, program, test, debug, and deploy. Although coordinating synchronization imposes consistent overhead, the overhead becomes largely amortized for large graphs. Synchronous systems demonstrate good scalability, with runtime often linearly increasing with the number of vertices [174]. As will be discussed
in Section 2.4.2.1 synchronous systems are often implemented along with message-passing communication, which enables a more efficient “batch messaging” method. Batch messaging can especially benefit systems with lots of network traffic induced by algorithms with a low computation-to-communication ratio [291].

Although synchronous systems are conceptually straight-forward and scale well, the model is not without drawbacks. One study found that synchronization, for an instance of finding the shortest path in a highly-partitioned graph, accounted for over 80% of the total running time [48], so system throughput must remain high to justify the cost of synchronization, since such coordination can be relatively costly. However, when the number of active vertices drops or the workload amongst workers becomes imbalanced, system resources can become under-utilized. Iterative algorithms often suffer from “the curse of the last reducer” otherwise known as the “straggler” problem where many computations finish quickly, but a small fraction of computations take a disproportionately longer amount of time [257]. For synchronous systems, each superstep takes as long as the slowest vertex, so synchronous systems generally favor lightweight computations with small variability in runtime.

Finally, synchronous algorithms may not converge in some instances. In graph coloring algorithms, for example, vertices attempt to choose colors different than adjacent neighbors [93] and require coordination between neighboring vertices. However, during synchronous execution, the circumstance may arise where two neighboring vertices continually flip between each others’ color. In general, algorithms that require some type of neighbor coordination may not always converge with the synchronous timing model without the use of some extra logic in the vertex program [291].
### TABLE 2.1

EXECUTION TIMING MODEL OF SELECTED FRAMEWORKS

<table>
<thead>
<tr>
<th>Framework</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pregel</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Giraph</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Hama</td>
<td>Synchronous</td>
</tr>
</tbody>
</table>
| GraphLab     | Asynchronous  
               | 162 161    |
| PowerGraph   | Both      |
| PowerSwitch  | Hybrid    |
| GRACE        | Hybrid    |
| GraphHP      | Hybrid    |
| P++          | Hybrid    |

#### 2.4.1.2 Asynchronous

In the asynchronous iteration model, no explicit synchronization points, *i.e.*, barriers, are provided, so any active vertex is eligible for computation whenever processor and network resources are available. Vertex execution order can be dynamically generated and reorganized by the scheduler, and the “straggler” problem is eliminated. As a result, many asynchronous models outperform corresponding synchronous models, but at the expense of added complexity.

Theoretical and empirical research has demonstrated that asynchronous execution can generally outperform synchronous execution [21, 162], albeit precise comparisons for TLAV frameworks depend on a number of properties [291]. Asynchronous systems especially outperform synchronous systems when the workload is imbalanced. For example, when computation per vertex varies widely, synchronous systems must wait for the slowest computation to complete, while asynchronous systems can con-
continue execution maintaining high throughput. One disadvantage, however, is that asynchronous execution cannot take advantage of batch messaging optimizations (see Section 2.4.2.4). Thus, synchronous execution generally accommodates I/O-bound algorithms, while asynchronous execution well-serves CPU-bound algorithms by adapting to large and variable workloads.

Many iterative algorithms exhibit asymmetric convergence. Low et al. demonstrated that, for PageRank, the majority of vertices converged within one superstep, while only 3% of vertices required more than 10 supersteps [162]. Asynchronous systems can utilize prioritized computation via a dynamic schedule to focus on more challenging computations early in execution to achieve better performance [306, 162]. Generally, asynchronous systems perform well by providing more execution flexibility, and by adapting to dynamic or variant workloads.

Although intelligent scheduling can improve performance, schedules resulting in sub-optimal performance are also possible. In some instances, a vertex may perform more updates than necessary to reach convergence, resulting in excessive computation [305]. Moreover, if implementing the pull model of execution, which is commonly implemented in asynchronous systems [162] and described in Section 2.4.3.2, communication becomes redundant when neighboring vertex values don’t change [305, 107].

The flexibility provided by asynchronous execution comes at the expense of added complexity, not only from scheduling logic, but also from maintaining data consistency. Asynchronous systems typically implement shared memory, discussed in Section 2.4.2.2 where data race conditions can occur when parallel computations simultaneously attempt to modify the same data. Additional mechanisms are necessary to ensure mutual exclusion, which can challenge algorithm development because framework users may have to consider low-level concurrency issues [275], like, for example, in GraphLab where users must select a consistency model [162].
2.4.1.3 Hybrid

Rather than adhering to the inherent strengths and weaknesses of a strict execution model, several frameworks work around a particular shortcoming through design improvements. One such implementation, GraphHP, reduces the high fixed cost of the global synchronization barrier using pseudo-supersteps \[48\]. Another implementation, GRACE, explores dynamic scheduling within a single superstep \[275\]. The PowerSwitch system removes the need to choose between synchronous and asynchronous execution and instead adaptively switches between the two modes to improve performance \[291\]. Together, these three frameworks illustrate how weaknesses with a particular execution model can be overcome through engineering and problem solving, rather than strict adoption of an execution model.

As previously discussed, synchronous systems suffer from the high, fixed cost of the global synchronization barrier. The hybrid execution model introduced by GraphHP, and also used by P++ framework \[311\], reduces the number of supersteps by decoupling intra-processor computation from the inter-processor communication and synchronization \[48\]. To do this GraphHP distinguishes between two types of nodes: boundary nodes that share an edge across partitions, and local nodes that only have neighboring nodes within the local partition. During synchronization, messages are only exchanged between boundary nodes. As a result, in GraphHP, a given superstep is composed of two phases: global and local. The global phase, which is executed first, runs the user program across all boundary vertices using data transmitted from other boundary vertices as well as its own local vertices. Once the global phase is complete, the local phase executes the vertex program on local vertices within a pseudo-superstep; the pseudo-superstep is different from a regular superstep in that: 1) pseudo-supersteps have local barriers resulting in local iterations independent of any global synchronization or communication; and 2) local message passing is done through direct, in-memory message passing, which is much faster.
than standard MPI-style messages.

A similar approach to segmented execution, as in GraphHP and P++, is the KLA paradigm \[108\], which creates a hybrid of synchronous and asynchronous execution. For graphs, the depth of asynchronous execution is parameterized, and asynchronous execution is allowed for a certain number of levels before a synchronous round. Similar to how GraphHP implements a round of boundary vertex execution before several rounds of local execution, KLA has multiple traversals of asynchronous execution before coordinating a round of synchronous execution. The trade-off is between expensive global synchronizations with cheap but possibly redundant asynchronous computations. KLA is also similar to delta-stepping used for single source shortest path \[184\].

The single-machine framework GRACE explores dynamic scheduling of vertices from within a single synchronous round \[275\]. To do this GRACE exposes a programming interface that, from within a given superstep, allows for prioritized execution of vertices and selective receiving of messages outside of the previous superstep. Results demonstrate comparable runtime to asynchronous models, with better scaling across multiple worker threads on a single machine.

Knowing \textit{a priori} which execution mode will perform better for a given problem, algorithm, system, or circumstance is challenging. Furthermore, the underlying properties that give one execution model an advantage over another may change over the course of processing. For example, in the distributed Single Source Shortest Path algorithm \[25\], the process begins with few active vertices, where asynchronous execution is advantageous, then propagates to a high number of active vertices performing lightweight computations, which is ideal for synchronous execution, before finally converging amongst few active vertices \[291\]. For some algorithms, one execution mode may outperform another only for certain stages of processing, and the best mode at each stage can be difficult to predict.
Motivated by the necessity for execution mode dynamism, PowerSwitch was developed to adaptively switch between synchronous and asynchronous execution modes [291]. Developed on top of the PowerGraph platform, PowerSwitch can quickly and efficiently switch between synchronous and asynchronous execution. PowerSwitch incorporates throughput heuristics with online sampling to predict which execution mode will perform better for the current period of computation. Results demonstrate that the PowerSwitch’s heuristics can accurately predict throughput, the switching between the two execution modes is well-timed, and overall runtime is improved for a variety of algorithms and system configurations [291].

2.4.2 Communication

Communication in TLAV frameworks entails how data is shared between vertex programs. The two conventional models for communication in distributed systems, as well as distributed algorithms, are message passing and shared memory [297, 163, 169]. In message passing systems, data is exchanged between processes through messages, whereas in shared memory systems data for one process is directly and immediately accessible by another process. This section compares and contrasts message passing and shared memory for TLAV frameworks. A third method of communication, active messages, is also presented. Finally, techniques to optimize distributed message passing are discussed.

Diagrams in Figure 2.4 are referenced throughout this section to illustrate the different communication implementations. A sample graph is presented in Figure 2.4a and Figures 2.4b-2.4e depict 4 TLAV communication implementations of the sample graph. For each implementation, vertices are partitioned across 2 machines, namely, vertices A, B, and C are partitioned to machine p1, and vertices D, E, and F are put on machine p2 (except Figure 2.4d and 2.4e where the graph is cut along vertex C).
Solid arrows represent local communication\(^5\) and dashed arrows represent network traffic.

### 2.4.2.1 Message Passing

In the message passing method of communication, also known as the LOCAL model of distributed computation \([205]\), information is sent from one vertex program kernel to another via a message. A message contains local vertex data and is addressed to the ID of the recipient vertex. In the archetypal message-passing framework Pregel \([174]\), a message can be addressed anywhere, but because vertices do not have ID information of all of other vertices, destination vertex IDs are typically obtained by iterating over outgoing edges.

After computation is complete and a destination ID for each message is determined, the vertex dispatches messages to the local worker process. The worker process determines whether the recipient resides on the local machine or a remote machine. In the case of the former, the worker process can place the message directly into the vertex’s incoming message queue. Else, the worker process looks up the worker-id of the destination vertex\(^6\) and places the message in an outgoing message buffer. The outgoing message buffer in Pregel, a synchronously-timed system, is flushed when it reaches a certain capacity, sending messages over the network in batches. Waiting until the end of a superstep to send all outgoing remote messages can exceed memory limits \([234]\).

Message passing is commonly implemented with synchronized execution, which guarantees data consistency without low-level implementation details. All messages

---

\(^5\)Local communication means communication between vertices residing on the same machine

\(^6\)If the graph is partitioned using random hash partitioning, then the destination worker can be determined by hashing the destination vertex ID. Otherwise, for more advanced partitioning methods (see Section 2.4.4), the worker process typically has access to a local routing table, provided by the master during initialization.
Figure 2.4: Distributed communication patterns for common communication implementations. The sample graph is partitioned across two machines (see Section 2.4.4), with vertices A, B, and C residing on machine p1, and vertices D, E, and F on machine p2. Pregel is represented in (b), GraphLab in (c), PowerGraph in (d), and GRE in (e).
sent during superstep $S$ are received in superstep $S + 1$, at which point a vertex program can access the incoming message queue at the beginning of $S + 1$’s program execution. Synchronous execution also facilitates batch messaging, which improves network throughput. For I/O bound algorithms with lightweight computation, such as PageRank [36], where vertices are “always active” so messaging is high [240], synchronous execution has been shown to significantly outperform asynchronous execution [291].

Message passing is depicted in Figure 2.4b, where vertex $C$ sends (an) inter-machine message(s) to vertices $D$, $E$, and $F$. Technically, messages are first sent from $C$ to the worker process of $p1$, which routes the messages to worker process $p2$, which places the message in a vertex’s incoming message queue, but the worker process-related routing is omitted from the figure without loss of generality. Figure 2.4b represents a general message passing framework, such as Pregel or Giraph. The three messages sent by $C$ across the network can be potentially reduced using optimization techniques in Section 2.4.2.4, namely, Receiver-side Scatter, depicted in Figure 2.5c.

2.4.2.2 Shared Memory

Shared memory exposes vertex data as shared variables that can be directly read or be modified by other vertex programs. Shared memory avoids the additional memory overhead constituted by messages, and doesn’t require intermediate processing by workers. Shared memory is often implemented by TLAV frameworks developed for a single machine (see Section 2.5.4), since challenges to a shared memory implementation arise in the distributed setting [213, 198], where consistency must be guaranteed for remotely-accessed vertices. Inter-machine communication for distributed shared memory still occurs through network messages. The Trinity framework [241] implements a shared global address space that abstracts away distributed memory.

For shared memory TLAV frameworks, race conditions may arise when an adja-
cent vertex resides on a remote machine. Shared memory TLAV frameworks often ensure memory consistency through mutual exclusion by requiring serializable schedules. Serializability, in this case, means that every parallel execution has a corresponding sequential execution that maintains consistency, cf., the dining philosophers problem [162, 94].

In GraphLab [162] border vertices are provided locally-cached ghost copies of remote neighbors, where consistency between ghosts and the original vertex is maintained using pipelined distributed locking [71]. In PowerGraph [94], the second generation of GraphLab, graphs are partitioned by edges and cut along vertices (see vertex-cuts in Section 2.4.4), where consistency across cached mirrors of the cut vertex is maintained using parallel Chandy-Misra locking [45]. GiraphX is a Giraph derivative with a synchronous shared memory implementation [259], which again provides serialization through Chandy-Misra locking of border vertices, although without local cached copies. The reduced overhead of shared memory compared to message passing is demonstrated by GiraphX, which converges 35% faster than Giraph when computing PageRank on a large Web Graph [259]. Moreover, some iterative algorithms perform better under serialized conditions, such as Dynamic ALS [312, 162], and popular Gibbs sampling algorithms that actually require serializability for correctness [93].

Shared memory implementations are depicted in Figure 2.4c and Figure 2.4d. In Figure 2.4c ghost vertices, represented by dashed circles, are created for every neighboring vertex residing on a remote machine, as implemented by GraphLab [162]. One disadvantage of shared-memory frameworks is seen when computing on scale-free graphs which have a certain percentage of high degree vertices, such as vertex C. In these cases the graph can be difficult to partition [153] resulting in many ghost vertices.

Figure 2.4d depicts shared memory with vertex cuts as implemented by Power-
PowerGraph combines vertex-cuts (discussed in Section 2.4.4) with the three-phase Gather-Apply-Scatter computational model (see Section 2.4.3.1) to improve processing of scale-free graphs. In Figure 2.4d, the graph is cut along vertex $C$, where $C1$ is arbitrarily chosen as the master and $C2$ as the mirror. For each iteration, a distributed vertex performs computation where: (i) both $C1$ and $C2$ compute a partial result based on local neighbors, (ii) the partial result is sent over the network from the mirror $C2$ to the master $C1$, (iii) the master computes the final result for the iteration, (iv) the master transmits the result back to the mirror over the network, then (v) the result is sent to local neighbors as necessary. PowerGraph demonstrates how the combination of advanced components, \textit{i.e.,} vertex-cuts and three-phase computation, can overcome processing challenges like imbalances arising from high-degree vertices in scale-free graphs.

Shared memory systems are often implemented with asynchronous execution. Although consistency is fundamentally maintained in synchronous message passing frameworks like Pregel, asynchronous, shared memory frameworks like GraphLab may execute faster because of prioritized execution and low communication overhead, but at the expense of added complexity for scheduling and maintaining consistency. The added complexity challenges scalability, for as the number of machines and partitions increase, more time and resources become devoted to locking protocols.

Dynamic computation addresses asymmetric convergence by only updating necessary vertices. Shared memory with asynchronous execution is an effective platform for dynamic computation, because the movement of data is separated from computation, allowing vertices to access neighboring values even if the values haven’t changed between iterations. This implies the \textit{pull} mode of information flow 2.4.3.2. In contrast, a vertex in a message-passing framework would need all neighboring values delivered in order to perform an update, even if some values had not changed. Dynamic computation is possible with message passing in the Cyclops framework, which
implements a *distributed immutable view*. Cyclops is a synchronous shared memory framework [50], where one of the replicated vertices is designated the master, which computes updates and messages the updated state to replicas at the end of an iteration. Cyclops outperforms synchronous message passing frameworks by reducing the amount of processing performed by each worker parsing messages, and is comparable to PowerGraph by delivering significantly fewer messages.

Significant deterioration in performance was noted in [105, 164] for larger graphs, although admittedly performance largely depends on algorithm behavior [291, 240]. In short, asynchronous shared memory systems can potentially outperform synchronous message passing systems, though the latter often demonstrate better scalability and generalization.

### 2.4.2.3 Active Messages

While message passing and shared memory are the two most commonly implemented forms of communication in distributed systems, a third method called *active messages* is implemented in the GRE framework [297]. Active messaging is a way of bringing computation to data, where a message contains both data as well as the operator to be applied to the data [274]. Active messages are sent asynchronously, and executed upon receipt by the destination vertex. Within the GRE architecture, active messages combine the process of sending and receiving messages, removing the need to store intermediate state, like message queues or edge data. When combined with the framework’s novel Agent-Graph model, described below, GRE demonstrates 20%–55% reduction in runtime compared to PowerGraph across three benchmark algorithms in real and synthetic datasets, including 39% reduction in the execution time per iteration for PageRank on the Twitter graph when scaled across 192 cores.

---

*Active messages as described in this section are different from message-passing messages that activate a vertex, like in Pregel*
over 16 machines when compared to a PowerGraph implementation on 512 cores across 64 machines [297].

The GRE framework modifies the data graph into an Agent-Graph. The Agent-Graph is a model used internally by the framework, but is not accessible to the user. The Agent-Graph adds combiner and scatter vertices to the original graph in order to reduce inter-machine messaging. Figure 2.4e shows that an extra scatter vertex, $C'$, is added to create the internal Agent-Graph model. The $C'$ vertex acts as a Receiver-side Scatter depicted in Figure 2.5c. This is useful because the new $C'$ vertex allows $C$ to only send one message across the network, which $C'$ then disperses to vertices $D$, $E$, and $F$. Combiner vertices are also added to the Agent-Graph in the same way as Server-side Aggregation depicted in Figure 2.5a. The Agent-Graph employed by GRE is similar to vertex-cuts in PowerGraph except that GRE messaging is unidirectional, and active messages are also utilized for parallel graph computation in the Active Pebbles framework [284, 76].

2.4.2.4 Message Passing Optimizations

Message passing can be costly, especially over a network. Thus several message-reducing strategies have been developed in order to improve performance. Some strategies are topology-driven and, as such, exploit the graph layout across machines, while other techniques are applied to specific algorithmic behavior. Three topology-driven optimizations are depicted in Figure 2.5 for messaging between machines $p_1$ and $p_2$ (or messaging from $p_1$, $p_2$, and $p_3$ to $p_4$, for Figure 2.5b).

The Combiner, inspired by the MapReduce function of the same name [67], is a message passing optimization originally used by Pregel [174]. Presuming the commutative and associative properties of a vertex function, a Combiner executes on a worker process and combines many messages destined for the same vertex into a single message. For example, if a vertex function computes the sum of all incom-
Figure 2.5: Partition-driven optimization strategies for distributed message passing. The Combiner technique employs both Sender-side and Receiver-side Combiners. 

When messages are inging messages, then a Combiner would detect all messages destined for a vertex $v$, compute the sum of the messages, then send the new sum to $v$. A Combiner can especially reduce network traffic when $v$ is remote, shown as *sender-side aggregation* (Figure 2.5a). When $v$ is local, a combiner can still reduce memory overhead by aggregating messages before placement into the incoming message queue, shown as *receiver-side aggregation* (Figure 2.5b). For the single-source shortest path algorithm, a combiner implementation resulted in a four-fold reduction in network traffic [174].

A related technique is the *receiver-side scatter*. For instances where the same message is sent to multiple vertices on the same remote machine, network traffic can be reduced by sending only one message and then having the destination worker distribute multiple copies, depicted in Figure 2.5c. The strategy has been employed in multiple frameworks, including the *Large Adjacency List Partitioning* in GPS [231], IBM’s X-Pregel [17], as the *fetch-once* behavior in LFGraph [112], and through *scatter* nodes of the Agent-Graph in GRE [297]. The technique reduces network traffic by increasing memory and processing overhead, as worker-nodes must store the out-going adjacency lists of other workers. With this in mind, GPS maintains a threshold where receiver-side scatter would only be applied for vertices above a certain degree. Experiments showed that as the threshold is lowered, network traffic at first decreases then plateaus, while runtime decreases but then increases, demonstrating
the existence of an optimal vertex-degree threshold. In X-Pregel, a ten-fold reduction in network traffic from Receiver-side Scatter resulted in a 1.5 times speedup \cite{17}. Clearly, the receiver-side scatter strategy can be effective, but unlike the combiner is not guaranteed to improve performance.

The three partition-driven optimizations in Figure \ref{fig:2.5} are related to the messaging structure of a framework, and not specific to algorithm behavior, albeit some assumptions are made regarding message computation. Computation for the combiner must be commutative and associative because order cannot be guaranteed, while messages for the receiver-side scatter must be identical, and independent of the adjacency list. Still, the techniques are oriented around partition-level messaging and apply to the worker process, only requiring certain operational properties in order to work. The Message-Online-Computing model proposed in \cite{311}, which improves memory usage by processing messages in the queue as they are delivered, also requires operations be commutative.

Conversely, algorithm-specific message optimizations have also been developed that restructure vertex messaging patterns for certain algorithmic behaviors \cite{232, 216}. For algorithms that combine vertices into a supervertex, like Boruvka’s Minimum Spanning Tree \cite{55}, the Storing Edges at Subvertices (SEAS) optimization implements a subroutine where each vertex tracks its parent supervertex instead of sending adjacency lists \cite{232}. For algorithms where vertices remove edges, like in the 1/2-approximation for maximum weight matching \cite{212}, the Edge Cleaning on Demand (ECOD) optimization only deletes stale edges when, counter-intuitively, activity is requested for the stale edge \cite{232}. To avoid slow convergence, ECOD is only employed above a certain threshold, e.g., when more than 1\% of all vertices are active. Both SEAS and ECOD exploit a trade-off between sending messages proportional to the number of vertices or proportional to the number of edges. Other strategies for reducing communication, based on aggregate computation, are discussed.
2.4.3 Execution Model

The model of execution for vertex-centric programs describes the implementation of the vertex function, and how data moves during computation.

2.4.3.1 Vertex Program Implementation

Vertex functions have been implemented as 1, 2, or 3 phase-models. Vertex functions have also been implemented as edge-centric functions. While the model choice does not typically impact the accuracy of the final result, combining certain implementations with other TLAV components can yield improved system performance for certain graph characteristics.

One Phase  The vertex programming abstraction implemented as a single function is well-characterized by the Pregel framework [174]. The single compute function of a vertex object follows the general sequence of accessing input data, computing a new vertex value, and distributing the update. In a typical Pregel program, the input data is accessed by iterating through the input message queue (messages that may have utilized a combiner), applying an update function based on received data, and then sending the new value through messages addressed by iterating over outgoing edges. Details based on other design decisions may vary, e.g., input and output data may be distributed through incident edges, or neighboring vertex data may be directly accessible, but in one-phase models the general sequence of vertex execution is performed within a single, programmed function. The Vertex.Compute() function is implemented in several TLAV frameworks in addition to Pregel, including its open-source implementations [14, 236] and several related variants [231, 17, 223]. The One-phase function implementation is conceptually straight-forward, but other
frameworks provide opportunities for improvement by dividing up the computation.

Two Phase   A two-phase vertex-oriented programming model breaks up vertex programming into two functions, most commonly referred to as the Scatter-Gather model. In Scatter-Gather, the *scatter* phase distributes a vertex value to neighbors, and the *gather* phase collects the inputs and applies the vertex update. While most single-phase frameworks *e.g.*, Pregel, can be converted into two phases, the Scatter-Gather model was first explicitly put forward in the Signal/Collect framework [253]. The two phase model is also presented as Scatter-Gather in [228], and is presented as the Iterative Vertex-Centric (IVEC) programming model in [298]. The Scatter-Gather programming model commonly occurs in TLAV systems where data is read/written to/from edges.

Ligra and Polymer are frameworks implemented for single-machines (see Section 2.5.4) that both implement a two-phase model. The user provides two functions, one function that executes across each vertex in the active subset and another function that executes all outgoing edges in the subset. The frameworks adopt a vertex-subset-centric programming model, which is similar to vertex-centric, but the framework retains a centralized view of the graph, where the whole graph is within the scope of computation, which is possible because the entire graph resides on a single machine in this case. The two phase model is executed within a program processing the whole graph.

A related two-phase programming model for message passing called Scatter-Combine is implemented in the GRE framework [297]. This model utilizes active messages, which are messages that include both data as well as the operator to be executed on the data [274]. In the first phase of the model, messages are both sent (Scattered) and the operators in the messages are executed (Combined) at the destination vertex. In the second phase, the combined result is used to update the vertex
value. The Scatter-Combine model incorporates two phases differently than Scatter-
Gather. Instead of the two phase Scatter-Gather model of (i) Gather-Apply, and
(ii) Scatter, the Scatter-Combine model uses active messages to institute (i) Scatter-
Gather, and then (ii) Apply. The GRE framework combines Scatter-Combine with a
novel representation of the underlying data graph, called the Agent-Graph, described
above, to reduce communication and improve scalability for processing graphs with
scale-free degree distributions.

Three Phase A three-phase programming model is introduced in PowerGraph as the
Gather-Apply-Scatter (GAS) model [94]. The Gather phase performs a generic sum-
mation over all input vertices and/or edges, like a commutative associative combiner.
The result is used in the Apply phase, which updates the central vertex value. The
Scatter phase distributes the update by writing the value to the output edges. Power-
Graph incorporates the GAS model with vertex-cut partitioning (see Section 2.4.4.3)
to improve processing of power-law graphs.

Edge-Centric The X-Stream framework provides an edge-centric two phase Scatter-
Gather programming model [228], as opposed to a vertex-centric programming model.
The model is edge-centric because the framework iterates over edges of the graph
instead of vertices. However, the framework may still be considered TLAV because
the two phase program operates on source and target vertices, adopting a similar local
scope. X-Stream leverages streaming edge data instead of random access for efficient
large scale graph processing on a single machine, and is discussed in Section 2.5.4 in
further detail.
2.4.3.2 Push vs. Pull

The flow of information for vertex-programs can be characterized as data being pushed or pulled \[191, 107, 54\]. In push mode, information flows from the active vertex performing the update outward to neighboring vertices, as in Pregel-like message-passing. In pull mode, information flows from neighboring vertices inward to the active vertex, as in GraphLab-like shared memory, when an active vertex reads neighbor’s data. Few TLAV frameworks explicitly adopt a push or pull mode. Instead, the information flow arises from other design decisions. Still, analyzing a system as push or pull allows one to reason about other system properties. For example, asynchronous execution is supported by both modes, but sender-side combining is only possible in push mode \[54\].

Push and pull modes are more commonly associated with databases and transactional processing, though have been more explicitly incorporated in broader graph engines and temporal frameworks (see Section 2.7 for related work). The Galois framework, with a flexible computation model enabling the implementation of a vertex-centric interface, allows users to choose push or pull mode \[145, 191\], as does Kineograph \[54\]. Chronos experiments with how push and pull modes impact caching \[107\].

Ligra is a single-machine graph processing framework that dynamically switches between push and pull-based operators based on a threshold. The framework is in part inspired by a recently developed shared-memory breadth-first search algorithm that achieves remarkable performance by switching between push and pull modes of exploration \[19\]. This algorithm, Ligra, and PowerSwitch from Section 2.4.1.3 exemplify how performance can be improved by dynamically adapting the processing technique to properties of the graph.

The delta-caching optimization, which is introduced in PowerGraph \[94\], which reduces the pulling of redundant data by tracking value changes. In a three phase
model, an accumulator value is the result of gather step. With delta-caching, a cached
copy of the accumulator for each vertex is stored by the worker, requiring additional
storage. If, for a given update, the change in the accumulator is minimal, then
neighboring vertices aren’t activated, and any change can be applied to the cached
copy stored by worker. A neighboring vertex can then use the cached copy during an
update. For delta-caching to be available, the apply function must be commutative,
associative, and have an inverse function. Delta-caching reduces redundant pulling by
not activating neighboring vertices for small changes, and resulted in a 45% decrease
in runtime for computing PageRank on the Twitter graph [94].

2.4.4 Partitioning

Large-scale graphs must be divided into parts to be placed in distributed memory.
Good partitions often lead to improved performance [231], but expensive strategies
can end up dominating processing time, leading many implementations to incorpo-
rate simple strategies, such as random placement [117]. Effective partitioning evenly
distributed the vertices for balanced workload, while minimizing inter-partition edges
to avoid costly network traffic, a problem formally known as \textit{k-way graph partitioning}
that is NP-complete with no fixed-factor approximation [12, 185].

Leading work in graph partitioning can be broadly characterized as (1) rigorous
but impractical mathematical strategies, or (2) pragmatic heuristics used in practice
[265]. Practical strategies, such as those employed in the suite of algorithms known
as METIS [122], often employ a three-phase multi-level partitioning approach [1].
Partition size is often allowed to deviate in the form of a “slackness” parameter in
exchange for better cuts [123].

Graph partitioning with METIS partitioning software is often considered the \textit{de facto}
standard for near-optimal partitioning in TLAV frameworks [251]. Despite a
lengthy preprocessing time, METIS-algorithms significantly reduce total communi-
cation and improve overall runtime for TLAV processing on smaller graphs \[231\]. However, for graphs of even medium-size, the high computational cost and necessary random access the entire graph renders METIS and related heuristics impractical. Alternatives for large-scale graph partitioning include distributed heuristics presented in Section 2.4.4.1, streaming algorithms in Section 2.4.4.2, vertex cuts in Section 2.4.4.3, and dynamic repartitioning in Section 2.4.4.4.

2.4.4.1 Distributed Heuristics

Distributed heuristics are decentralized methods, requiring little or no centralized coordination. Distributed partitioning is related to distributed community detection in networks \[90, 221\], the two main differences being: 1) communities can overlap whereas partitions cannot, and 2) partitioning requires a priori specification of the number of partitions, whereas community detection typically does not. Much distributed partitioning work has been inspired by distributed community detection, namely label propagation \[218\].

Label propagation occurs at the vertex level, where each vertex adopts the label of the plurality of its neighbors. Though the process is decentralized, label propagation for partitioning necessitates a varying amount of centralized coordination in order to maintain balanced partitions and prevent “densification”: a cascading phenomenon where one label becomes the overwhelming preference \[218\]. The densification problem is addressed in \[271\] wherein a simple capacity constraint is enforced that is equal to the available capacity of the local worker divided by the number of non-local workers. In \[267\], balanced vertex distribution is maintained by constraining label propagation and solving a linear programming optimization problem that maximizes a relocation utility function. In \[219\], vertices swap labels, either with a neighbor or possibly a random node, and simulated annealing is employed to escape local optima. The cost of centralized coordination incurred by these methods is much less than the
cost of random vertex access on a distributed architecture, as with ParMETIS.

More advanced label propagation schemes for partitioning are presented in [276] and [249]. In [276], label propagation is used as the coarsening phase of a multi-level partitioning scheme, which processes the partitioning in blocks to accommodate multi-level partitioning for large-scale graphs. In [249], several stages of label propagation are utilized to satisfy multiple partitioning objectives under multiple constraints. [302] use a parallel multi-level partitioning algorithm for k-way balanced graphs that operates in two phases: an aggregate phase that uses weighted label propagation, and then a partition phase that performs the stepwise minimizing RatioCut method.

2.4.4.2 Streaming

Streaming partitioning is a form of online processing that partitions a graph in a single-pass. For TLAV frameworks, streaming partitioning is especially efficient since the partitioning can be performed by the graph loader, which loads the graph from disk onto the cluster. The accepted streaming model assumes a single, centralized graph loader that reads data serially from disk and chooses where to place the data amongst available workers [251] [265]. Centralized streaming heuristics can be adapted to run in parallel [251], however, depending on the heuristic, concurrency between the parallel partitioners would likely be required [196]. One of the first online heuristics was presented by Kernighan and Lin and is used as a subroutine in METIS [131].

GraphBuilder [117] is a similar library that, in addition to partitioning, supports an extensive variety of graph loading-related processing tasks.

A streaming partitioner on a graph loader reads data serially from disk, receiving one vertex at a time along with its neighboring vertices. In a single look at the vertex the streaming partitioner must decide the final placement for the vertex on a worker partition, but the streaming partitioner has access to the entire subgraph of
already placed vertices. In a variant of the streaming model, the partitioner has an available storage buffer with a capacity equal to that of a worker partition, so the partitioner may temporarily store a vertex and decide the partitioning later [251], however this buffer is not utilized by the top performing streaming partitioners. For most heuristics, the placement of later vertices is dependent on placement of earlier vertices, so the presentation order of vertices can impact the partitioning. Thus, an adverse ordering can drastically subvert partitioning efforts, however, experiments demonstrate that performance remains relatively consistent for breadth-first, depth-first, and random orderings of a graph [251] [265].

Two top-performing streaming partitioning algorithms are greedy heuristics. The first is linear deterministic greedy (LDG), a heuristic that assigns a vertex to the partition with which it shares the most edges while weighted by a penalty function linearly associated with a partition’s remaining capacity. The LDG heuristic is presented in [251], where 16 streaming partitioning heuristics are evaluated across 21 different data sets. The use of a buffer in addition to the LDF heuristic has been adapted for streaming partitioning of massive Resource Description Framework (RDF) data [278]. Another variant uses unweighted deterministic greedy instead of linear deterministic greed (LDG), to perform greedy selection based on neighbors without any penalty function; this unweighted variant has been employed for distributed matrix factorization [3]. Further analysis of LDG-related heuristics on random graphs, as well as lower bound proofs for random and adversarial stream ordering, is presented in [250].

Another top-performing streaming partitioner is FENNEL [265], which is inspired by a generalization of optimal quasi-cliques [264]. FENNEL achieves high quality partitions that are in some instances comparable with near-optimal METIS partitions. Both FENNEL and LDG have been adapted to the restreaming graph partitioning model, where a streaming partitioner is provided access to previous stream results...
Restreaming graph partitioning is motivated by environments such as online services where the same, or slightly modified, graph is repeatedly streamed with regularity. Despite adhering to the same linear memory bounds as a single-pass partitioning, the presented restreaming algorithms not only provide results comparable to METIS, but are also capable of partitioning in the presence of multiple constraints and in parallel without inter-stream communication.

2.4.4.3 Vertex Cuts

A vertex-cut, depicted in Figure 2.4d, is equivalent to partitioning a graph by edges instead of vertices. Partitioning by edges results in each edge being assigned to one machine, while vertices are capable of spanning multiple machines. Only changes to values of cut vertices are passed over the network, not changes to edges. Vertex-cuts are implemented by TLAV frameworks in response to the challenges of finding well-balanced edge cuts in power-law graphs \[1, 153\]. Complex network theory suggests power-law graphs have good vertex cuts in the form of nodes with high degree \[5\]. A rigorous review of vertex separators is presented in \[81\].

PowerGraph combines vertex-cuts with the three-phase GAS model (Section 2.4.3.1) for efficient communication and balanced computation \[94\]. For vertices that are cut and span multiple machines, one copy is randomly designated the master, and remaining copies are mirrors. During an update all vertices first execute a gather, where all incoming edge values are combined with a commutative associative sum operation. Then the mirrors transmit the sum value over the network to the master, which executes the apply function to produce the updated vertex value. The master then sends the result back over the network to the mirrors. Finally, each vertex completes the update by scattering the result along its outgoing edges. For each update, network traffic is proportional to the number of mirrors, therefore, breaking up high-degree vertices reduces network communication and helps to balance computation.
Since its initial implementation in PowerGraph, the vertex-cut approach has been adopted by several other TLAV frameworks. GraphX is a vertex programming abstraction for the Spark processing framework [95, 301] where the adoption of vertex-cuts demonstrated an 8-fold decrease in the platform’s communication cost. GraphBuilder [117], an open-source graph loader, supports vertex-cuts and implements grid and torus-based vertex-cut strategies that were later included in PowerGraph. PowerLyra [52] is a modification to PowerGraph that hybridizes partitioning where vertices with a degree above a user-defined threshold are cut, while vertices below the threshold are partitioned using an adaptation of the FENNEL streaming algorithm [265]. PowerLyra also incorporates unidirectional locality similar to GRE framework (see Section 2.4.2.3). BiGraph is a framework developed on PowerGraph that implements partitioning algorithms for large-scale bipartite graphs [51]. LightGraph [309] is a framework that optimizes vertex-cut partitions by using edge-direction-aware partitioning, and by not sending updates to mirrors with only in-edges.

Several edge partitioning analyses and algorithms have recently been developed. A thorough analysis comparing expected costs of vertex partitioning and edge partitioning is presented in [32]. In this study, edge partitioning is empirically demonstrated to outperform vertex partitioning, and a streaming least marginal cost greedy heuristic is introduced that outperforms the greedy heuristic from PowerGraph.

Centralized hypergraph partitioning, including edge partitioning, is NP-hard, and several exact algorithms have been developed [69, 136, 102, 237]. However, because of their complexity, such algorithms are too computationally expensive and not practical for large-scale graphs. Centralized heuristics have been shown to be equally impractical [23]. A large-scale vertex-cut approach for bipartite graphs based on hypergraph partitioning is presented in [186] as part of a vertex-centric program for computing the alternating direction of multipliers optimization technique. A distributed edge partitioner was developed in [220] that creates balanced partitions while reducing
the vertex cut, based on the vertex partitioner in [219]. Good workload balance for skewed degree distributions can also be achieved with degree-based hashing [290]. Finally, as part of a non-vertex-centric BSP graph processing framework, a distributed vertex-cut partitioner is presented in [98] that uses a market-based model where partitions use allocated funds to buy an edge.

2.4.4.4 Dynamic Repartitioning

While an effective partitioning equally distributes vertices among the partitions, for TLAV frameworks, the number of active vertices performing updates on a given superstep can vary drastically over the course of computation, which creates processing imbalances and increases run time. Dynamic repartitioning was developed to maintain balance during processing by migrating vertices between workers as necessary.

Reasons for changing active vertex sets include topological mutations to the graph and algorithmic execution properties. Topological mutations may occur if the framework supports dynamic or temporal graphs (see Related Work in Section 2.7). Topology may also change due to the algorithm, such as graph coarsening [270].

With a static topology, the execution pattern of the algorithm can also change the active vertex set. While vertex algorithms such as synchronous PageRank execute on every vertex for every superstep, other algorithms introduce dynamism. [240] classifies 9 vertex algorithms as either (i) always active, (ii) traversal, or (iii) multi-phase, where the active vertex set of the latter two classifications can vary widely and unpredictably, depending on the graph. For dynamic repartitioning to prove beneficial, the associated overhead must be less than the additional costs stemming from processing imbalance.

According to [231], a dynamic repartitioning strategy must directly address (i) how to select vertices to reassign, (ii) how and when to move the assigned vertices,
and (iii) how to locate the reassigned vertices. Other properties of a strategy include whether coordination is centralized or decentralized, and how the strategy combats “densification” and enforces vertex balance. Densification is akin to the rich-get-richer phenomenon, and can occur in greedy or decentralized protocols for partitioning/clustering, where one partition becomes over-populated as the repeated destination for migrated vertices [269]. In response, protocols often implement constraints that prevent a partition from exceeding a certain capacity. The XPregel framework, for example, only permits the worker with the most vertices and edges to migrate vertices [17].

Table 2.2 presents 6 TLAV frameworks that support dynamic repartitioning: GPS [231], Mizan [134], XPregel [17], xDGP [269], LogGP [294], and the Catch the Wind prototype [240]. The table includes what active vertex set imbalances are targeted by the frameworks, what metrics are used to identify vertices for reassignment, how reassigned vertices are located after migration, how densification is avoided, and whether the protocol is centralized or decentralized.

Among the 6 frameworks that implement dynamic repartitioning, all are synchronous, and repartitioning occurs at the end of a superstep, separate from the updates. When a vertex is selected for migration, the worker must send all associated data to the new worker, including the vertex ID, the adjacency list, and the incoming messages to be processed in the next superstep. To avoid sending all incoming messages over the network, many dynamic repartitioning frameworks implement a form of delayed migration, where the new worker is recognized as the owner of the migrated vertex, but the vertex value remains on the old worker for an extra iteration in order to compute an update. With delayed migration, the incoming message queue doesn’t need to be migrated, but the new worker still receives new incoming messages [134, 231].

Though fundamentally sound, many experiments demonstrate that dynamic repart-
TABLE 2.2:
FEATURE SUMMARY FOR DYNAMIC REPARTITIONING FRAMEWORKS

<table>
<thead>
<tr>
<th>Framework</th>
<th>Cause of Imbalance</th>
<th>Reassignment Metric</th>
<th>How to Locate Migrated Verts</th>
<th>Densification Avoidance</th>
<th>Coordination</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPS</td>
<td>Algorithm</td>
<td>Sent/Recv Msgs and Run Time</td>
<td>Broadcast Worker ID</td>
<td>Swap Min-Set</td>
<td>Decentralized</td>
</tr>
<tr>
<td>Mizan</td>
<td>Algorithm</td>
<td>Sent/Recv Msgs and Run Time</td>
<td>Distributed Hash Table</td>
<td>Metric-based Swap</td>
<td>Decentralized</td>
</tr>
<tr>
<td>XPregel</td>
<td>Algorithm</td>
<td>Sent/Recv Msgs and Run Time</td>
<td>Broadcast Worker ID</td>
<td>Repartition Largest Worker</td>
<td>Centralized</td>
</tr>
<tr>
<td>xDGP</td>
<td>Topology</td>
<td>Labels of Neighbors</td>
<td>Broadcast Worker ID</td>
<td>Fraction of Capacity</td>
<td>Decentralized</td>
</tr>
<tr>
<td>LogGP</td>
<td>Both</td>
<td>Runtime</td>
<td>Lookup Table</td>
<td>Repartition Longest-Running Workers</td>
<td>Centralized</td>
</tr>
<tr>
<td>Catch the Wind</td>
<td>Algorithm</td>
<td>Sent/Recv Msgs and Run Time</td>
<td>Lookup Table</td>
<td>Quota</td>
<td>Decentralized</td>
</tr>
</tbody>
</table>

Dynamic repartitioning is often not worth the high overhead. Results in [17] show that while network I/O is significantly reduced over time, overall runtime shows minor improvements. Independent tests of GPS show dynamic repartitioning to be detrimental for all cases in [164], and similar results are observed for GPS and Mizan in [105]. However, one major shortcoming in these evaluations is the use of the PageRank algorithm for experimentation. Dynamic repartitioning is most effective for dynamic active vertex sets, but with PageRank vertices are always active, so dynamic repartitioning performs predictably poorly. Asynchronous dynamic repartitioning protocols have yet to be explored for TLAV frameworks, but the added complexity and overhead for asynchrony demonstrated in Section 2.4.1 suggest that such an implementation is not practical.

2.5 Implementation

This section overviews implementation details of TLAV frameworks relating to the distributed environment. These details include system architecture and fault tolerance. Additionally, TLAV frameworks that employ novel techniques to process large-scale graphs on single machines are surveyed.
2.5.1 System Architecture

TLAV frameworks generally always employ the master-slave architecture. A master node initializes the slave workers, monitors execution, and manages coordination (and synchronization if invoked) amongst the workers. Generally, the master is responsible for graph loading and partitioning, but with a network filesystem available, the loading and partitioning can be performed in parallel [23]. The master also stores global values, such as aggregators [17]. The workers each execute a copy of the program on the local partitions and inform the master of runtime status.

One notable exception to the general master-slave architecture is XPregel [17], implemented in X10 [46]. X10 implements an Asynchronous Partitioned Global Address Space (APGAS), which is a shared address space but with a local structure that enables highly productive distributed and parallel programming. With APGAS, the number of local “places” is provided at runtime, which the programmer may utilize as necessary. XPregel does implement master-slave, but in X10, the master is actually just place 0, sans hierarchy, and opens the door for alternative architectures, like recursive structures.

2.5.2 Multi-Core Support

For multi-core machines, many BSP-based frameworks including Pregel [17] simply assign a partition to a given core, but frameworks can better utilize computational resources through multi-threading. XPregel [17] supports multi-threading by dividing a partition into a user-defined number of subpartitions, assigning one thread to each subpartition. GraphLab [162] implements multi-threading and avoids deadlocks through scheduler restrictions. GPS [23] implements 3 types of threads: a thread for vertex computation, a thread for communication, and a thread for parsing. Cyclops [50] implements a hierarchical BSP model [31] with a split design to parallelize computation and messaging while exploiting locality and avoiding synchronization.
Cyclops demonstrates that multi-threading can improve runtime relative to single-threaded execution for the same framework, at the expense of added complexity.

### 2.5.3 Fault Tolerance

Distributed systems must often account for the potential failure of one or more nodes over the course of computation. When a node fails, a replacement node may become available, but all data and computation performed on the failed node is lost.

Checkpointing is a common fault tolerance implementation, where an immutable copy of the data is written to persistent storage, such as a network filesystem. Pregel implements synchronous checkpointing, where the graph is copied in between super-steps. When a failure occurs, the system rolls back to the most recently saved point, all partitions are reloaded, and the entire system resumes processing from the checkpoint. The partition of the failed node is reloaded to a new replacement node. If messaging information is also logged, then resources can be saved by only reloading and recomputing data on the replacement node. GraphLab implements asynchronous vertex checkpointing, based on Chandy-Lamport snapshots, which need not halt the entire program and can result in slightly faster overall execution than synchronous checkpointing, minding certain program constraints.

GraphX is a graph processing library for Apache Spark, which is developed based on the Resilient Distributed Dataset (RDD) abstraction. RDDs are immutable, partitioned collections created through data-parallel operators, like map or reduce. RDDs are either stored externally, or generated in-memory from operations on other RDDs. Spark maintains the lineage of operations on an RDD, so upon any node failure the RDD can be automatically recovered. GraphX leverages the RDDs of Spark to create a graph abstraction and Pregel interface.

The Imitator framework implements fault-tolerance based on vertex replicas,
or ghosts/mirrors used in shared memory (see Section 2.4.2.2). The use of replicas for fault tolerance is founded in the observation that the hash partitioning of many real-world directed graphs results in the replication of over 99% of vertices \cite{277}. By replicating every vertex, a full copy of the graph can reside in distributed memory, enabling faster recovery times at the expense of relatively little additional memory consumption and network messaging \cite{277}. The efficiency of Imitator is tied to the effectiveness of the partitioning (see Section 2.4.4). Imitator outperforms checkpointing for large graphs distributed over several nodes, when only one replica per vertex is required. State-of-the-art partitioning methods like METIS, or a smaller number of partitions (Imitator experiments were run on 50 nodes), would likely lead to increased overhead for Imitator. Also, the number of replicas is tied to the degree of fault tolerance. To support the failure of \( k \) machines, then \( k \) replicas are required, increasing overhead for each additional failure supported.

A partition-based checkpoint method for fault tolerance is presented in \cite{242}. During execution, a recovery executor node collects run-time statistics, and upon failure, uses heuristics to redistribute the partitions. Checkpointed partitions of the failed nodes can be reassigned amongst both new and old nodes, parallelizing recovery. Partitions on healthy nodes can also be reassigned for load balancing.

2.5.4 Single Machine Architectures

Like MapReduce, TLAV frameworks are advantageous because they are highly scalable while providing a simple programming interface, abstracting away the lower level details of distributed computing. However, such environments also stipulate the availability of elaborate infrastructure, cluster management, and performance tuning, which may not be available to all users.

Single machine systems are easier to manage and program, but commodity machines do not have the memory capacity to process large-scale graphs in-memory.
This section overviews single machine TLAV frameworks that employ novel methods to process large-scale graphs. The main features of the 4 single machine frameworks in this section are presented in Table 2.3.

Processing large-scale graphs on a single machine requires either substantial amounts of memory, or storing part of the graph out-of-memory, in which case performance is dictated by how efficiently the graph can be fetched from storage. In [244], it’s argued that high-end servers, offering 100GB to 1TB of memory or more, is enough capacity for many real and synthetic graphs reported in the literature. Such machines would be capable of storing large graphs and executing relatively simple graph algorithms, though more complex algorithms would likely exhaust resources.

The recommendation service at Twitter [100], which implements a single machine graph processing system with 144 GB of RAM, finds that in practice one edge occupies roughly five bytes of RAM on average. Compression techniques are further explored for large memory servers in [245]. Yet, graphs of scale are not practical on lower-end machines containing around 8 to 16 GB of memory [148]. Accordingly, single machine frameworks have been developed that implement the vertex-centric programming model and process a graph in parts. Central to many single machine TLAV frameworks are novel data layouts that efficiently read and write graph data to/from external storage. One common representation is the compressed sparse row format, which organizes graph data as out-going edge adjacency sets, allowing for the fast look-up of outgoing edge, and has been implemented in many state-of-the-art shared memory graph processors [204, 111], including Galois [191].
GraphChi The seminal single machine TLAV framework is GraphChi [148], which was explicitly developed for large-scale graph processing on a commodity desktop. GraphChi enables large-scale graph processing by implementing the Parallel Sliding Window (PSW) method, a graph data layout previously utilized for efficient PageRank and sparse-matrix dense-vector multiplication [53, 21]. PSW partitions vertices into disjoint sets, associating with each interval a shard containing all of the interval’s incoming edges, sorted by source vertex. Intervals are selected to form balanced shards, and the number of intervals is chosen so any interval can fit completely in memory. A sliding window is maintained over every interval, so when vertices from one shard are updated from in-edges, the results can be sequentially written to out-edges found in sorted order in the window on other shards. GraphChi may not be faster than most distributed frameworks, but often reaches convergence within an order of magnitude of the performance of distributed frameworks [148], which is reasonable for a desktop with an order of magnitude less RAM. The GraphChi framework was later extended to a general graph management system for a single machine called GraphChi-DB [147].

Storage concepts for single machine graph processing are further explored in [298] through two directions. The first project investigates reducing random accesses in SSDs through prefetching, in a project called RASP that later evolved into PrefEdge [194]. The second project is X-Stream [228], an edge-centric single machine graph processing framework that exploits the trade-off between random memory access and sequential access from streaming data.

X-Stream Streaming data from any storage medium provides much greater bandwidth than random access. Experiments on the X-Stream testbed, for example, demonstrate that streaming data from disk is 500 times faster than random access [228]. X-Stream combines a novel data layout, where an index is built over a storage-
based edge list with an edge-centric Scatter-Gather programming model that includes a shuffle phase. Data is read from, and updates are written to, streaming edge data. Though the framework is edge-centric, a user-defined update function is executed on the destination vertex of an edge. X-Stream reports that it can process a 64-billion edge graph on a single machine with a pair of 3TB magnetic disks attached [175].

* FlashGraph While GraphChi and X-Stream are designed for general external storage, the FlashGraph framework is developed for graphs stored on any fast I/O device, such as an array of SSDs. FlashGraph is deployed on top of the set-associative file system (SAFS) [310], which includes a scalable lightweight page cache, and implements a custom asynchronous user-task I/O interface that reduces overhead for asynchronous I/O. FlashGraph employs asynchronous message-passing and vertex-centric programming with the semi-external memory (SEM) model [204], where vertices and algorithmic state reside in RAM, but edges are stored externally. In experiments comparing GraphChi and X-Stream, FlashGraph outperformed both by orders of magnitude even when the data for GraphChi and X-Stream was placed into RAM-disk [310].

* PathGraph In addition to the path-centric programming model, further discussed in Section 2.6, PathGraph also implements a path-centric compact storage system that improves compactness and locality [300]. Because most iterative graph algorithms involve path traversal, PathGraph stores edge traversal trees in depth-first search order. Both the forward and reverse edge trees are each stored in a chunk storage structure that compresses data structure information including the adjacency set, vertex IDs, and the indexing of the chunk. The efficient computational model and storage structure of PathGraph resulted in improved graph loading time, lower memory footprint, and faster runtime for certain algorithms when compared to GraphChi
2.6 Alternative Graph Granularity

The strengths of the vertex-centric programming model are also its weaknesses. Whereas vertex programs may be relatively simpler to reason about since only local data is available, the algorithms are less expressive than conventional centralized algorithms. While TLAV frameworks exhibit better scalability, execution can be slow because of high overhead from synchronization and message traffic that takes magnitudes longer compared to computation. Several frameworks strive for the best of both worlds by adopting a scope that is greater than a vertex but less than the graph, summarized in Table 2.4.

2.6.1 Subgraph-centric Frameworks

Considering the challenges addressed by TLAV frameworks, taking a subgraph-centric approach is sensible. Conventional graph algorithms require the entire graph in memory, which is not possible with graphs of scale. A subgraph, though, can be partitioned into a size small enough to fit into memory (considering computation) while the connections between subgraphs would be no more, and likely much less, than the total number of edges. The system would better utilize processing while retaining scalability.

The subgraph-centric programming model is implemented in varying degrees by several frameworks. The Giraph++ [262], Blogel [295], and GoFFish [246] frameworks provide a subgraph-centric interface for programming sequential algorithms. Both Giraph++ and Blogel provide a subgraph-centric interface in addition to a vertex-centric interface. The results of the sequential programs can then be shared either through vertex programs on boundary nodes, or in the case of Blogel, results can be shared directly between subgraphs. GoFFish exclusively offers a subgraph-centric
interface, and implements messaging between subgraphs and also from subgraphs to specific vertices, the latter being used for traversal algorithms. By allowing subgraphs to directly message vertices, any vertex-centric algorithm can be implemented by a subgraph-centric framework, maintaining scalability while enabling significant performance improvement. Collectively, subgraph-centric frameworks dramatically outperform TLAV frameworks, often by orders of magnitude in terms of computing time, number of messages, and total supersteps 262, 295.

The GraphHP [48] and P++ [311] frameworks do not implement an interface for sequential programs, but do differentiate between inter-partition nodes to improve performance. In these two frameworks, supersteps are split into two phases: in the first phase messages are exchanged between vertices on partition boundaries, and in the second phase, vertices within a partition repeatedly execute the vertex program to completion, exchanging messages in memory. This method reduces communication and improves performance, however, iteratively executing intra-worker vertex programs is less efficient than executing a sequential algorithm. Message-passing algorithms are typically more scalable than sequential graph algorithms, but P++ is not distributed, nor is Block-based GRACE [293], an extension of [275], although the later demonstrates that executing vertex updates on a subgraph block basis improves locality and cache hits while reducing memory access time, which is a bottleneck for computationally light algorithms like PageRank.

TLAV frameworks illustrate the principal ideas for scalable graph processing, but for the best performance, users may consider subgraph-centric frameworks. Subgraph frameworks leverage principles of TLAV frameworks to execute sequential graph algorithms in a distributed environment. The Giraph++, Blogel, and GoFFish frameworks reduce the scope of sequential graph algorithms for the subgraph to fit in memory while utilizing vertex or subgraph messaging to maintain scalability. Together, the vertex-centric and subgraph-centric programming model, compared to
TABLE 2.4:
FRAMEWORKS OF ALTERNATIVE SCOPE

<table>
<thead>
<tr>
<th>Framework</th>
<th>Programming Model</th>
<th>Sequential Algorithms</th>
<th>Vertex Messaging</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giraph++</td>
<td>Subgraph</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Blogel</td>
<td>Subgraph</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>GoFFish</td>
<td>Subgraph</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>GraphHP</td>
<td>Subgraph</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>P++</td>
<td>Subgraph</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>GRACE (block)</td>
<td>Subgraph</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>PathGraph</td>
<td>Path</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Ligra</td>
<td>Vertex Subset</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Polymer</td>
<td>Vertex Subset</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Galois</td>
<td>User-Defined Set</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

sequential graph algorithms, demonstrate how scalability varies inversely with scope. This idea is explored in more detail in Appendix B and also left for future work.

2.6.2 Other Scopes: Paths and Sets

While subgraph-centric frameworks illustrate the scope/scalability trade-off, several other frameworks adopt alternative computational scopes that demonstrate additional benefits.

A more specific type of subgraph, a traversal tree, is used for the programming model in PathGraph \[300\]. Traversals are a fundamental component of many graph algorithms, including PageRank and Bellman-Ford shortest path. PathGraph first partitions the graph into paths, with each partition represented as two trees, a forward and reverse edge traversal. Then, for the path-centric computational model, path-centric scatter and path-centric gather functions are available to the user to define an algorithm that traverse each tree. The user also defines a vertex update function, which is executed by the path-centric functions during the traversal. Like block-based GRACE, the path-centric model utilizes locality to improve performance through reduced memory usage and efficient caching. PathGraph also implements a path-centric storage model that enables the framework to process billion node graphs on
a single machine (see Section 2.5.4) [300].

Graph processing frameworks designed for single machines can implement interfaces of unique granularity. A vertex subset interface is implemented in Ligra [244]. Ligra argues that high-end servers provide enough memory for large-scale graphs, and thus implements a vertex-centric programming interface while retaining a global view of the graph. Inspired by a hybrid breadth-first search (BFS) algorithm [19], Ligra dynamically switches between sparse and dense representations of edge sets depending on the size of the vertex subset, which impacts whether push or pull operations are performed with the vertex subset. Polymer [303] adopts a similar interface as Ligra, but with several NUMA-aware optimizations. Galois [145] is a shared memory framework that executes user-defined set operators while exploiting amorphous data parallelism [207]. Galois can be implement a variety of programming interfaces, including the vertex-centric paradigm [191].

2.6.3 Optimizations

Two optimizations have been introduced in [232] for TLAV frameworks that improve performance by adopting a scope of the graph other than vertex-centric. The Finishing Computation Serially (FCS) method is applicable when an algorithm with a shrinking set of active vertices converges slowly near the end of execution [232]. The FCS method is triggered when the remaining active graph can fit in the memory of a single machine; in these instances the active portions are sent to the master and completed serially from a global, shared memory perspective of the graph.

Similarly, the Single Pivot (SP) optimization [232], first presented in [216], also temporarily adopts a global view. For algorithms that execute breadth-first search (BFS) across all vertices, e.g., the connected components algorithm, instead of executing BFS from every node, which incurs a high messaging cost, SP randomly selects one vertex from the graph and performs BFS just from that vertex. Since
most graphs have one big component, in addition to many small ones, the BFS from a random node can be executed until the big component is found, then BFS from every vertex that’s not in the big component can execute BFS to complete the algorithm, resulting in significantly fewer total messages. This optimization adjusts scope by randomly selecting a single vertex by utilizing a global aggregator [174], which also adopt a scope beyond vertex.

2.7 Related Work

In this chapter, I surveyed vertex-centric graph processing systems for large-scale graphs. In previous related work, Pregel and GraphLab have been compared [230], and general graph processing systems have been surveyed [133] [195], and 4 TLAV frameworks have been empirically evaluated on 4 algorithms [105]. A tutorial on TLAV frameworks was recently delivered at an international conference [4].

TLAV frameworks intersect several subjects, including graph processing, distributed computing, Big Data, and distributed algorithms. Several graph processing frameworks have been recently developed outside of the vertex-centric programming model. PEGASUS combines the BSP model with generalized matrix-vector multiplication (GIM-V) [121], while TurboGraph introduces the pin-and-slide model to perform GIM-V on a single machine [106]. Combinatorial BLAS [39] and the Parallel Boost Graph Library [97] are software libraries for high-performing parallel computation of sequential programs. Piccolo performs distributed graph computation using distributed tables [210].

Graph databases, such as Neo4j [280], HyperGraphDB [114], and GBASE [121], are decidedly different from TLAV frameworks. Both treat vertices as first class citizens, and both face related problems like partitioning, but the key distinction is that databases focus on transactional processing while TLAV frameworks focus on batch processing [49]. Databases offer local or online queries, such as 1-hop neighbors,
whereas TLAV systems iteratively process the entire graph offline in batch. Some
more general graph management systems, like Trinity [241] and Grace [211], offer
suites of features that include both vertex-centric processing and queries. Sensibly, a
graph processing engine may be developed on top of a graph database. However the
two should not be confused, and performance is incomparable.

A closely related Big Data framework is MapReduce [67, 209]. MapReduce is a
different programming model from TLAV frameworks, but similarly enables large-
scale computation and, when implemented, abstracts away the details of distributed
programming. The programming model is effective for many types of computation,
but addresses neither iterative processing nor graph processing [209, 174]. Iterative
computation is not natively supported, as the programming model performs only a
single pass over the data with no loop awareness. Moreover, I/O is read/written
to/from a distributed filesystem, e.g., HDFS, rendering iterative computation ineffi-
cient [209]. Nonetheless, several frameworks have extended MapReduce to support
iterative computation [77, 83, 301] but such frameworks are still agnostic to the
challenges of graph processing. Graph computation with MapReduce has been ex-
plored [158], but is generally acknowledged to be lacking [56, 174]. A comparison of
MapReduce and BSP is provided in [119]. Still, some argue that MapReduce should
remain the sole “hammer” for Big Data analytics because of the widespread adoption
throughout industry [157].

Similarly, in response to TLAV shortcomings, such as poor out-of-core support
and lengthy loading times, some frameworks rework pre-existing graph database tech-
nologies to provide a vertex-centric interface [80]. However, many of these projects
lose sight of the main problems addressed by the vertex-centric processing. TLAV
frameworks are ultimately Big Data solutions, designed large graphs to be leveraged
against the memory and processing power of several machines, not single machines.
Moreover, TLAV frameworks iteratively process the entire graph, and do not provide
graph queries like 1-hop or 2-hop neighbors. TLAV frameworks are not a universal solution for graph analytics, but rather provide an approach for scalable, iterative graph processing.

Temporal graph processing is beyond the scope of this survey, though a small number of TLAV frameworks have been developed for temporal analysis \[54, 107\]. These frameworks compute temporal properties offline in batch through graph snapshots, necessitating multiple framework components, including a front-end ingress component, an analytics engine, and a storage component such as a graph database. Temporal graph layout optimizations were introduced in Chronos \[107\]. These frameworks illustrate how advanced graph analytics systems utilize the strengths of different graph technologies for different components, \textit{e.g.}, graph databases for storage and online queries, and vertex-centric computation for batch analytics. Dynamic graph algorithms and general analytics systems have also been surveyed \[2, 270\]. Dynamic graphs are supported by many frameworks including Pregel, but the topic was omitted from this survey due to widely varying support by the frameworks and broad scope of the topic.

While coined "vertex-centric" relative to conventional graph processing approaches, the algorithms executed by TLAV frameworks are more formally known as distributed algorithms. Distributed algorithms is a mature field of study \[169\], and further examples beyond Figure \[2.3\] may be found within the referenced frameworks. Some works have explored distributed algorithms within the context of TLAV frameworks \[297\], but researchers and practitioners should be aware that TLAV frameworks execute distributed algorithms \[169\], which come from a field with a considerable body of work, including theory and analysis. The theoretical limits of what can be computed with vertex-centric frameworks, specifically with the synchronous, message-passing LOCAL model, has been studied \[144\].

This chapter surveys and compares the various components of TLAV frameworks,
which are a platform for executing vertex-centric algorithms. Like MapReduce, these frameworks provide an interface for a user-defined function, while abstracting away the lower-level details of cluster computing. Changing the components of the framework will impact system performance and run-time characteristics, but will generally not impact the design or result of the algorithm.

### 2.8 Conclusions

TLAV frameworks have been designed in response to the challenges of processing large graphs. Primary challenges include the unstructured nature of graphs, where an edge may span any two vertices, so the entire graph must be randomly accessible for conventional processing. TLAV frameworks are also developed for ease of use, providing a simple vertex-centric interface while abstracting away the lower level details of cluster computing. MapReduce similarly enables highly scalable computing, but is ill-suited for iterative graph processing.

By adopting a vertex-centric programming model, the scope of computation is dramatically reduced. To perform an update, each vertex only needs data from immediate neighbors. Data residing on a separate machine can be acquired directly between workers, avoiding the bottleneck of central coordination, enabling excellent scalability. The four pillars of the vertex-centric programming model, (i) timing, (ii) communication, (iii) the execution model, and (iv) partitioning, were presented and surveyed in the context of distributed graph processing frameworks. However, vertex-centric algorithms, colloquially known as distributed algorithms, have an established history and are still actively researched [169, 144].

Several related frameworks were explored that similarly adopt a computational scope of the graph at varying granularity. These frameworks of alternative scope

---

8 An exception to this rule is synchronous versus asynchronous execution some algorithms, such as graph coloring Section 2.4.1.
are like a Goldilocks solution to graph processing. Centralized algorithms with the entire graph in scope require too much memory, vertex-centric algorithms can scale but are less expressive and require many relatively slow messages, whereas subgraph-centric algorithms can utilize the two resources just right. A significant contribution of TLAV frameworks is exposing how, for graphs, reducing the scope of a program increases scalability.

Of course, expressing a particular algorithm as subgraph-centric is not trivial. The future of practical large-scale distributed graph processing may be related to finding algorithms that process a graph as independent subgraphs, such as divide-and-conquer, or algorithms that can process graphs at multiple, or even dynamic, scopes [276]. The performance of the subgraph-centric processing is also closely tied to the effectiveness of large-scale graph partitioning, including streaming and distributed partitioning techniques.

TLAV frameworks are a tool for graph processing at scale. Not all graphs are large enough to necessitate distributed processing, and not all graph problems need the whole graph to be computed iteratively. Moreover, there is often more than one way to solve a problem, but these frameworks are simple to program, easy to distribute, and are not a bad choice for the right type of problem. Subgraph-centric frameworks take vertex-centric frameworks a step further for performance. Datasets will continue to grow dramatically into the new age of Big Data, and the design of processing systems should begin asking if they can scale out infinitely. TLAV frameworks illustrate how conventional centralized systems will fail in the Big Data ecosystem, and how decentralized platforms must be embraced.
3.1 Overview

This chapter presents part of on-going work that began while interning at IBM Zurich in the hardware accelerators group. The overall project is to develop a custom hardware architecture for the parallel processing of a well-known graph problem. The overall project includes 2 parts: i) a pre-processing step in software, and ii) a hardware processing step. This chapter focuses on the pre-processing step, and will not discuss in detail the hardware processing step, or graph problem under investigation, in consideration of an on-going patent filing\(^3\).

A custom hardware processing element, or kernel, has been developed for hardware-accelerated parallel processing of a well-known graph problem. The hardware component does not require random memory access, but does require the input data to be specifically ordered, depicted in Figure 3.1. An input set for a kernel contains a given node, and all the node’s neighbors, provided in Lexicographic Breadth First Search (LexBFS) order. To obtain that order, the data must be preprocessed in software. Pre-processing graph data is not uncommon, especially when the pre-processing takes

\(^1\)Work completed at IBM Zurich, with K. Atasu, S. Dragone, and C. Hagleitner

\(^2\)To be submitted for publication pending patent review

\(^3\)A patent for work in this chapter has been submitted, titled "Graph Data Representation and Pre-Processing for Efficient Parallel Search Tree Traversal"
Figure 3.1. An overview of how the graph is preprocessed for the hardware processing kernel. For every node in the graph, the adjacency list set of each node and its neighbors must be provided in LexBFS order.

linear time or less \([251]\). This chapter explores how to efficiently pre-process the data for the given task.

Two pre-processing methods, Method I and Method II, are presented that deliver appropriate data to the hardware components. These two methods are based on LexBFS. The first method naively executes LexBFS on many subgraphs, while the second method is more efficient by i) first running LexBFS on across the whole graph, then ii) using that information to intelligently prune and order vertices to reduce random graph access for subsequent traversals. Graph processing is notoriously difficult to parallelize \([167]\), and often suffers from I/O, rather than computational, bottlenecks, because while the actual computation is relatively simple (e.g., check if a node has been visited, locate an incident edge), graph processing time is often limited by data access. Beyond presentation of a project-specific algorithmic improvement, the graph processing paradigm of reducing random access to graph data is explored.

This chapter is presented as follows: Related work to LexBFS is provided in Section 3.2. An overview of breadth-first search and Lexicographic Breadth-First Search is presented in Section 3.3 and Section 3.4. A high-level overview of the problem is presented in Section 3.5. Method I and Method II are presented in Section 3.6 and
Section 3.7 A brief technical comparison is presented in Section 3.8 Experimental results are presented in Section 3.9 and Section 3.10 concludes with final remarks.

3.2 Related Work

Lexicographic Breadth-First Search (LexBFS) was introduced by Rose and Tarjan as an algorithm for recognizing chordal graphs [227]. While less utilized than the ubiquitous graph traversal methods breadth-first search of depth-first search [258], the unique properties of LexBFS have led to use in a wide variety of graph applications beyond chordality testing [59], including the recognition of restricted graph families like Unit Interval graphs and cographs [58, 35], diameter approximation [62], and finding dominating pairs of AT-free graphs [61]. A general framework for graph search, including LexBFS, is presented in [60].

Two implementations of the LexBFS algorithm have been introduced. The first, original implementation from Rose and Tarjan performs the traversal using a doubly-linked list data structure augmented with pointers [227]. A second, more recent implementation uses partition refinement through pivoting [101]. Both implementations perform in linear time with respect to the graph, though structures from the second method lend themselves to LexBFS variants [59]. One variant of LexBFS breaks ties using a previous LexBFS ordering [247], while another variant produces a LexBFS ordering of the compliment graph, without computing the actual complement [177]. While no distributed algorithm for LexBFS has been discovered [197], parallel LexBFS has been implemented for the GPU [168], based on the partition refinement method. For this work, I implement the original Rose and Tarjan method.

This project uses LexBFS for pre-processing. The other, related project that utilizes the preprocessed data is outside the scope of this chapter. Pre-processing graph data for a subsequent task is not uncommon, especially when simple pre-processing may improve performance of the more complex task. For distributed graph com-
putation, streaming graph partitioning is explored [251, 265, 196], as an alternative to more intensive partitioning methods not suitable for large-scale graphs [12, 122]. Streaming graph partitioning explores graph pre-processing in 2 ways: i) streaming partitioning performance depends on graph ordering [251], and authors explore linear time pre-processing, including breadth-first search and depth-first search, for streaming, and ii) the actual partitioning of a graph can be a pre-processing step for distributed computation, performed by Pregel and related frameworks [174]. For further details, refer to Chapter 2.4.4.2. Linear-time pre-processing is prevalent in graph processing [251], and LexBFS is performed in linear time [227].

3.3 Breadth-First Search

Breadth-first search (BFS) is an approach for traversing a graph, and a fundamental operation in graph theory. Given a graph G=(V,E), with n vertices in V, and m edges in E, and a root vertex s, breadth-first search is a method for systematically exploring every vertex in in the graph, beginning from the root node s.

### Algorithm 2: Breadth First Search ordering

**Input:** Graph G=(V,E) and root vertex s ∈ V  
**Output:** An ordering σ of V  

σ_v = ∅ for all v ∈ V;  
Queue Q={s} ;  
for i=1 to n do  
    v = Q.popfront();  
    σ_v = i;  
    foreach vertex w, where w ∈ Neighbor(v) and σ_w = ∅ do  
        if !Q.contains(w) then Q.append(w);  
    end  
end
In graph exploration, a data structure is used to store the set of vertices that have not yet been explored. When a vertex $v$ is explored, all neighboring vertices $N(v)$ that have not yet been explored are added to the data structure. In BFS, that data structure is a queue. The exploration is initialized by adding the root vertex to the queue, then the vertex is popped from the front of the queue, and all unexplored neighbors are added to the back of the queue. The process repeats until completion, with a runtime complexity of $O(n+m)$ as every node and edge is explored once. The key component that differentiates BFS from Lexicographic Breadth-First Search, explained next in Section 3.4, is that in BFS, unexplored neighbors for a given vertex are enqueued in random order.

A breadth-first search ordering simply represents the order in which the vertices are traversed. The vertex ordering $\sigma$ is a bijective mapping of vertices to integers 1 through $n$, and is ascending beginning with the root vertex of order 1. As unexplored vertices are enqueued in random order, a graph can have many valid BFS orderings. An algorithm for obtaining a BFS ordering is presented in Algorithm 2.

3.4 Lexicographic Breadth-First Search

Lexicographic Breadth-First Search (LexBFS) is a more specific, or more restrictive, form of breadth-first search. Like how every square is a rectangle, but not every rectangle is a square; every LexBFS ordering is a BFS ordering, but not every BFS ordering is a LexBFS ordering. Rather than enqueuing unexplored vertices randomly, LexBFS utilizes augmented information in order to enqueue unexplored vertices according to the order of previously explored neighbors.

Each vertex is augmented with an additional label variable that stores a string. When a vertex is explored, the label string of each neighbor is appended with a lexicographic value corresponding to the order of exploration. On each iteration, the unexplored node with the lexicographically largest label is selected, with ties are
broken randomly. Using a double-linked list data structure, LexBFS also runs in $O(n + m)$ time. LexBFS is presented in Algorithm 3.

**Algorithm 3: Lexicographic Breadth First Search (LexBFS)**

**Input:** Graph $G=(V,E)$ and root vertex $s \in V$

**Output:** An ordering $\sigma$ of $V$

$\sigma_v = \emptyset$, $\sigma'_v = \emptyset$ for all $v \in V$; /* $\sigma'$ is a temp label */

for $i=1$ to $n$ do

select vertex $v$ with lexicographically largest $\sigma'_v$ among all $v$ where $\sigma_v = \emptyset$;

$\sigma_v = i$;

foreach vertex $w$, where $w \in \text{Neighbor}(v)$ and $\sigma_w = \emptyset$ do

$\sigma'_w$.append(n-i);

end

end

An example execution of LexBFS is provided Figure 3.2. The sample graph is provided in Figure 3.2a and the traversal is depicted in Table 3.2b. The traversal is rooted at node A. In the first iteration, A is selected/explored (denoted by the ‘-’ character in Table 3.2b), and it’s lexicographic label is propagated to neighbors. Whereas in Algorithm 3 the lexicographic label is a decreasing integer value, in this example the label is the vertex ID, to better illustrate label propagation in relation to the graph, as well as illustrate the flexibility of lexicographic definition. In iteration 2, node B is chosen randomly among \{B, C, D\}, as all have label 'a'. In iteration 3, node C is chosen as it has the largest lexicographic label, and it’s ID is appended to the label of neighbor E and F. The final ordering is the order in which nodes are selected.
3.5 Pre-processing for Hardware

An algorithm architecture has been developed with 2 patents filed. The algorithm architecture highly parallelizes a graph processing problem. Each processing kernel operates independently of other kernels (embarrassingly parallel). The kernel accepts as input the set of adjacency lists of a node and its neighbors, \( \{v \cup N(v)\} \), executed for every \( v \in V \). Each set \( \{v \cup N(v)\} \) must be LexBFS ordered, rooted at node \( v \). An overview is depicted in Figure 3.1. Two solutions, Method I and Method II, are presented the achieving the pre-processing.

3.6 Method I

Method I naively executes LexBFS on the subgraph induced by a vertex \( v \) and its neighbors \( N(v) \), for every node \( v \in V \). Method I is illustrated in Figure 3.3 and Algorithm 4. An induced subgraph means, given a set of nodes, a graph is...
created with those nodes and just the edges between those nodes. For every node \( v \), a subgraph is induced by \( v \) and its neighbors, containing in total \( 1 + \text{degree}(v) \) nodes. An induced subgraph will contain the number of edges equal to the degree(v) + number of edges between nodes adjacent to \( v \). Summed over all induced subgraphs, LexBFS for Method I is performed \( n \) times, and will traverse a total \( 2 \times m + n \) nodes, and \( 2 \times m + (\text{the number of triangles in } G) \) edges, where \( m \) is the number of edges in \( G \), and \( n \) the number of vertices. Method I is relatively naive, the advantages being: i) simple to conceptualize and implement, ii) easy to parallelize, and iii) minimal memory requirements, as the whole graph need not be stored in memory, and could perhaps take advantage of a graph stored externally, such as in a graph database where node and neighbor queries are practical.

3.7 Method II

Method I is simple to conceptualize and implement, but performs redundant processing and random graph data access through unnecessary traversals. The second method, Method II, first performs LexBFS across the whole graph, then uses that information to reduce the size of induced subgraph, thus reducing random accesses,
**Algorithm 4:** Method I performs LexBFS from every node on the subgraph induced by a node and its neighbors.

**Input:** Graph $G=(V,E)$

**Output:** For every $v \in V$, an ordering $\sigma$ of $\{v \cup N(v)\}$

**foreach** vertex $v \in V$ **do**

- Graph $H = G.$subgraph($\{v \cup N(v)\}$);
- $\sigma = \text{LexBFS}(H,v);

**end**

when ordering the adjacency list sets for a node and its neighbors. LexBFS is still performed for each node, but on smaller sets of nodes. The method is depicted in Figure 3.4 and presented in Algorithm 5.

First, LexBFS is performed over the whole graph, rooted from a random vertex $s$. This builds an initial ordering of the vertices. When building the final adjacency lists sets for each vertex $v$ and its neighbors $N(v)$, vertices in $N(v)$ that are ordered before $v$, according to this initial ordering, do not need to be considered for the final set.

Next, I consider the number of hops (or distance, assuming an unweighted graph) from each node in $\{v \cup N(v)\}$ to $s$, the root node for the initial traversal. Nodes are in the same *layer* if they are the same number of hops, or the same distance, from $s$. For a given node $v$, nodes adjacent to $v$ that reside in the same layer as $v$ are in the correct LexBFS order for a traversal rooted at $v$, even though these neighboring nodes were ordered according to a traversal from $s$. So from the initial LexBFS traversal, when building an adjacency list set for $\{v \cup N(v)\}$, neighboring nodes $N(v)$ ordered before $v$ can be discarded, and nodes ordered after $v$ but in the same layer as $v$ are in the correct LexBFS order, relative to a traversal from $v$.

For a vertex $v$ in layer $k$, neighbors in $N(v)$ residing in layer $k+1$ need to be reordered, despite the initial LexBFS. The problem and solution is depicted in Figure 3.5. To get the neighbors of $v$ in layer $k+1$ in LexBFS order according to a traversal from node $v$, the neighbors of $v$ in layer $k+1$ must be: i) re-ordered relative
to $N(v)$ in layer $k$ with orders greater than $v$, then ii) the $N(v)$ in layer $k+1$ must re-ordered relative to each other, meaning LexBFS is performed on the subgraph induced by $N(v)$ in layer $k + 1$.

The subgraph induced by $N(v)$ in layer $k+1$ is smaller than the subgraph induced by $\{v \cup N(v)\}$. Performance is improved because the graph is smaller and fewer random accesses are required for traversal, relative to Method I. The re-ordering step, where $N(v)$ in layer $k+1$ are re-ordered relative to $N(v)$ in layer $k$, does not require random graph access because previous label information is used from the initial LexBFS, and nodes can be ordered using Radix Sort, which is linear with respect to the word size, in this case bounded by the number of $N(v)$ nodes in layer $k$ and practically small.

After the initial LexBFS which traverses $n$ nodes and $m$ edges, the number of subgraph traversals equals the number of nodes with no leaf nodes, relative to the initial LexBFS. Across all subsequent LexBFS traversals, the number of nodes traversed equals the number of inter-layer edges, or edges between nodes in different layers of the initial traversal. The number of edges traversed equals the number of intra-layer triangle edges, meaning an edge between two vertices in the same layer, with a common neighbor in the previous layer.

\begin{center}
\textbf{Algorithm 5:} Method II algorithm
\end{center}

\begin{verbatim}
Input: Graph $G=(V,E)$ and root vertex $s \in V$
Output: For every $v \in V$, an ordering $\sigma$ of $\{v \cup N(v)\}$
$\sigma_G = \text{LexBFS}(G, s); /* mod tracks dist from root */$
\foreach vertex $v \in V$ do
  $N'(v) = \{w \in V | \sigma_G(w) < \sigma_G(v)\};$
  $N''(v) = \{w \in N'(v) | \text{dist}(w, s) = \text{dist}(v, s) - 1\};$
  Reorder $N''(v)$ using labels from $N'(v)$
  $s' =$ first ordered node in $N''(v)$
  $\sigma = \text{LexBFS}(N''(v), s')$
end
\end{verbatim}
3.8 Technical Discussion

Method I is simple and straight-forward to implement. The graph can be stored externally, such as through a graph database, granted the necessary nodes can be brought into memory. On a graph with n nodes and m edges, Method I will result in \( n \) total LexBFS traversals. In total, \( 2 \times m + n \) nodes will be traversed, and the total number of edges traversed will equal \( 2 \times m + (\text{the number of triangles}) \) in the graph.

Method II is more efficient but requires more storage. While Method I performs LexBFS on subgraphs bounded by the degree of a given vertex, Method II first performs LexBFS across the whole graph. This means the graph must fit in memory. After the initial LexBFS traversal, which traverses n nodes and m edges, the number of subgraph traversals equals the number of nodes in the original traversal with no leaf node. The total number of nodes traversed equals the number of edges between nodes in different layers of the initial LexBFS ordering, plus the \( n \) nodes of the original traversal. The total number of edges traversed, in addition to the original m edges, equals the number of edges between two nodes in the same layer of the initial traversal.
(a) A portion of the original graph, in LexBFS order according to some root node (not pictured). Vertices are ordered right to left, top to bottom. Nodes are organized in layers according to the number of hops to the root. Despite the initial LexBFS order, Node B and its neighbors need to be provided in LexBFS order. While node B and its neighbors in layer \( n \) will be in proper order, the neighbors of node B in layer \( n + 1 \) are not in LexBFS order rooted at B.

(b) To get node B’s neighbors in layer \( n + 1 \) in LexBFS order according to B, first node B’s neighbors in layer \( n \) are ordered according to B’s neighbors in layer \( n \). Since the layer \( n \) nodes have already been labeled by the layer \( n + 1 \) nodes (in the original, full graph LexBFS), the layer \( n + 1 \) nodes can be reordered without having to access graph data adjacency lists. In this instance, node H moves to the front of the layer \( n + 1 \) order.

(c) With node B’s neighbors in layer \( n + 1 \) ordered according to node B’s neighbors in layer \( n \), LexBFS only needs to be performed on node B’s neighbors in layer \( n + 1 \). This takes into account edges between layer \( n + 1 \) nodes. In this case, since G is neighbors with H, G moves up in order.

Figure 3.5. Method II runs an initial LexBFS over the whole graph to reduce the number of nodes and vertices traversed by subsequent LexBFS on the subgraphs induced by each vertex. Still, some nodes need to be reordered. How and why some nodes need to be re-ordered are depicted.
LexBFS traversal, which share a common neighbor in the previous layer.

The number of traversed edges will strongly influence the performance of these 2 methods. While the number of edges in Method I is directly related to the number of triangles in the graph, the number of edges in Method II is equal to a subset of triangles. So Method II will traverse less edges, and should perform faster, than Method I.

3.9 Experimental Results

The two methods are experimentally evaluated using both synthetic and real-world data sets. The synthetic data sets are generated according to two models: the Erdos-Renyi model for random graphs, and the Barabasi-Albert model for scale-free graphs. Graphs for both models were generated with 10,000 nodes and varying edge parameters, as summarized in Table 3.2 and Table 3.3. The real-world graphs of Facebook, Twitter, and Epinions were obtained through the Stanford SNAP network data repository [152]. Experiments were run on a Notre Dame Center for Research Computing general computing node, consisting of a 271 HP DL165 G6 server, with Dual 6-core 2.4 GHz AMD Opteron processors, with 12 GB of RAM and a single 160 GB SATA disk. The pre-processing software was programmed with Python.

Results demonstrate considerable speed-up for Method II over Method I. Real-world graphs of Facebook, Twitter, and Epinions demonstrate considerable speed-up, depicted in Table 3.1. For synthetic graphs, I generated two sets of graphs: i) random graphs according to the Erdos-Renyi model, and ii) scale-free graphs according to the Barabasi-Albert model. Both graphs contained 10,000 nodes, but the edge parameters were varied. Results for the synthetic graphs are depicted in Figure 3.6 and Figure 3.7, which demonstrate a linear increase in run-time for both methods with the number of triangles in a graph. Method II demonstrably outperforms Method I, but the difference is less dramatic for dense graphs with many triangles, such as
TABLE 3.1

EXPERIMENTAL RESULTS FOR REAL-WORLD GRAPHS

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
<th>Triangles</th>
<th>Method I</th>
<th>Method II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>4,039</td>
<td>88,234</td>
<td>1,612,010</td>
<td>34.86s</td>
<td>2.18s</td>
</tr>
<tr>
<td>Twitter</td>
<td>81,306</td>
<td>1,342,310</td>
<td>13,082,506</td>
<td>386.72s</td>
<td>112.30s</td>
</tr>
<tr>
<td>Epinions</td>
<td>75,879</td>
<td>508,837</td>
<td>1,624,481</td>
<td>72.11s</td>
<td>31.17s</td>
</tr>
</tbody>
</table>

scale-free graphs, observed in Figure 3.7

3.10 Conclusion

Two methods for software pre-processing utilizing LexBFS as a central component were presented, to provide designated input for a custom hardware processing kernel. The first method naively performed LexBFS from every specified subgraph, whereas the second method used an initial LexBFS over the entire graph to intelligently prune subsequent subgraphs for more efficient traversal. The second method reduced random graph access, avoiding the I/O bottleneck, by both pruning subgraphs and utilizing previous traversal information to perform re-ordering. Experimental results confirm the performance advantage of Method II, outperforming Method I by an order of magnitude for both real-world and synthetic graphs. For large-scale graph processing, reducing random graph access is performance-critical. Future work will further explore this paradigm by applying these methods to a custom hardware architecture.
Figure 3.6. Processing random graphs generated by the Erdos-Renyi model with 10,000 nodes
Figure 3.7. Processing scale-free graphs generated by the Barabasi-Albert model with 10,000 nodes
### TABLE 3.2

**RANDOM GRAPH PROFILES**

<table>
<thead>
<tr>
<th>Edge Parameter</th>
<th>Nodes</th>
<th>Edges</th>
<th>Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>.04</td>
<td>10,000</td>
<td>1,999,614</td>
<td>10,664,008</td>
</tr>
<tr>
<td>.06</td>
<td>10,000</td>
<td>2,998,680</td>
<td>35,953,764</td>
</tr>
<tr>
<td>.08</td>
<td>10,000</td>
<td>3,999,605</td>
<td>85,307,698</td>
</tr>
<tr>
<td>.10</td>
<td>10,000</td>
<td>4,998,373</td>
<td>166,493,288</td>
</tr>
<tr>
<td>.12</td>
<td>10,000</td>
<td>6,002,328</td>
<td>288,338,448</td>
</tr>
<tr>
<td>.14</td>
<td>10,000</td>
<td>6,998,722</td>
<td>457,102,891</td>
</tr>
<tr>
<td>.16</td>
<td>10,000</td>
<td>7,995,481</td>
<td>681,478,860</td>
</tr>
<tr>
<td>.18</td>
<td>10,000</td>
<td>8,998,052</td>
<td>971,377,215</td>
</tr>
</tbody>
</table>

### TABLE 3.3

**SCALE-FREE GRAPH PROFILES**

<table>
<thead>
<tr>
<th>Edge Parameter</th>
<th>Nodes</th>
<th>Edges</th>
<th>Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10,000</td>
<td>990,000</td>
<td>6,364,019</td>
</tr>
<tr>
<td>200</td>
<td>10,000</td>
<td>1,960,000</td>
<td>37,106,810</td>
</tr>
<tr>
<td>300</td>
<td>10,000</td>
<td>2,910,000</td>
<td>101,792,502</td>
</tr>
<tr>
<td>400</td>
<td>10,000</td>
<td>3,840,000</td>
<td>206,221,809</td>
</tr>
<tr>
<td>500</td>
<td>10,000</td>
<td>4,750,000</td>
<td>353,925,668</td>
</tr>
<tr>
<td>600</td>
<td>10,000</td>
<td>5,640,000</td>
<td>547,748,475</td>
</tr>
</tbody>
</table>
4.1 Introduction

Software systems have reached a level of maturity where focus has shifted from creation and development to function and design. Sophisticated tools and technologies make the realization of any software idea possible, but for such an idea to persist in the modern ecosystem, the technology must be efficient and well-designed. Well-designed means engaging but not confining the user experience, and also complementing or even cooperating with tangential technologies. Gone are the monolithic software systems. The new technological era is defined by cognitive computing systems that are mobile, social, agile, context-aware, and work with both the user and one another. Cognitive computing systems bring together natural language processing, machine learning, and Big Data analytics to improve human decision making.

What cognitive systems aim to achieve on the front-end, autonomic computing seeks to achieve on the back-end. Both cognitive computing and autonomic computing seek to improve the accuracy and efficiency of human decisions. To this end, autonomic computing achieves this goal by building software systems of increased autonomy, so humans can focus on decision-making rather than system maintenance and administration. Autonomic computing takes software autonomy a step beyond

1Ongoing work completed at IBM Watson Research Center with Jeff Kephart, Manager of the Agents & Emergent Phenomena Group

2To be submitted to AAMAS in November 2016
autonomous software *agents*, to an autonomous *system* of software agents, where value is realized through intelligent agent interaction, and the system achieves an additional layer of autonomy through self-management.

Autonomic computing explores software systems capable of self-management [127, 113]. Whereas large, complex software systems often require considerable personnel resources for maintenance, autonomic systems improve efficiency and resource utilization through self-management. A self-managing software system is capable of detecting incorrect system operation and autonomously readjust to fix itself. The four traits of an autonomic computing system are:

1. Self-configuration
2. Self-optimization
3. Self-healing
4. Self-protection

This chapter primarily investigates the third autonomic trait of self-healing. The appropriateness of using the new Docker technology for self-healing is discussed, and the impact of a self-healing Docker system is discussed in regards to the other 3 autonomic traits. A self-healing system using Docker system is presented, developed within the experimental research prototype Cognitive Environment Lab (CEL). To further investigate the impact of the self-healing Docker functionality on complex software systems, a simulation test-bed has been developed using synthetic web-service agents, and experiments further illustrate how self-healing with Docker limits the ”network effect” of downed agents.

4.2 Related Work

Autonomic computing describes self-managing and self-organizing software systems [127, 129, 283, 91]. Similarly, swarm intelligent systems are multi-agent systems
that self-organize into swarms that exhibit emergent behavior capable of solving problems \[179\]. Unlike autonomic systems that are centralized managed, like through an agent-directory-service \[82\], swarms are decentralized, which leads to many advantageous system properties, including scalability, resilience to failure, and adaptability to dynamic conditions. Examples of swarm intelligent systems are often cited in nature, such as the coordinated flocking of birds or the collective foraging behavior of ants. The application of swarm intelligence to self-optimization behavior in autonomic computing has been investigated \[192\].

4.3 Incorporating Agents with Docker

Docker is a new virtualization technology that has gained increasing attention in recent years \[183\]. Docker is a lightweight virtual machine (VM) designed for running a single process or web application on a linux host. Users write a Dockerfile for managing all necessary files and library dependency information, use the Dockerfile to build an Image, then run an instance of the Image, called a Container. A Container is a running instance of a given web application. Docker solves a myriad of problems for complex web applications.

Docker is different from conventional virtual machines in that a single Docker Container only runs a single process or web application. Typical virtual machines emulate a full operating system, consume considerable memory and other resources, are capable of running multiple processes. Docker Containers, on the other hand, are designed to be lightweight and run a single process, where each Docker Container manages dependencies independent of other Containers. So while a single virtual machine hosting several applications would require maintaining consistent library versions across all applications, Docker Containers are isolated from one another so each container can maintain different dependencies. Docker maintains several advantages over conventional server virtualization
1. Rapid application deployment
2. Portability across machines
3. version control and component reuse
4. sharing
5. lightweight footprint
6. simplified maintenance

Docker has gained wide adoption in industry while research explores abilities of the platform. Service discovery and self-healing is explored using Docker in [252].

4.3.1 The Case for Docker within Autonomic Computing

The benefit of utilizing Docker for autonomic systems is discussed in this chapter. Docker is utilized by the CEL to develop a system that is self-healing, one of the four traits of autonomic systems.

Docker is a good choice to use for self-healing systems. In a complex software system, components can fail for any number of reasons. This relates to Fault-tolerant systems [91]. Fault-tolerant systems, such as Spark for large-scale analytics [301], take measures to ensure that the system can proceed despite the failure of one or more components. Self-healing systems constitute a broader approach with consideration for the symbiotic relationship with users and the external environment.

Systems such as Cognitive Systems are open systems that interact with humans and the real world. This means that any number of inputs or external factors can knock system components offline. The things that can negatively impact the system are infinite, and unpredictable. Rather than trying to model or account for every possible fault or event, which is impossible, self-healing systems focus on how to recover from such in case of its occurrence.
Docker is a good technology for incorporating into self-healing systems. My investigation looks at agents that are web services. As is good design practice, a degree of modularity is assumed between agents.

4.3.2 Docker In Practice

Docker is available for Ubuntu, Mac OS X, and Windows. It is based on LXC (Linux Container) technology and requires sudo privileges to run. A lightweight Docker server daemon is installed on the local machine, which is responsible for hosting the Docker containers. So a given host runs the Docker server, which hosts each Docker container.

4.4 Cognitive Environments Lab

The Cognitive Environments Lab (CEL) is a Symbiotic Cognitive System (SCS) prototype under research development at IBM Watson [128, 92]. The CEL incorporates state-of-the-art research in cognitive computing, machine learning, human-computer interaction, and data visualization to provide a dynamic, real-time environment for interactive problem solving across a variety of domains. Users interact with the system through multiple input channels, including, but not limited to, voice and motion, and output is provided through synthesized speech and several large displays. Equipped with advanced data analytics methods, like deep learning, the CEL provides a powerful, yet flexible and frictionless environment for interactive problem solving that is better suited to human thought processes than the conventional computing experience, fostering creativity and collaboration. In one example scenario, two decision-makers use the CEL to analyze company statistical profiles for the mergers and acquisitions business [92]. A CEL demonstration is depicted in Figure 4.1.

The CEL is a complex system composed of many autonomous applications, any
Figure 4.1: Two users interact with the CEL system through speech and motion recognition. The CEL incorporates the autonomic computing principle of self-healing to provide a state-of-the-art cognitive experience.

number of which may be interacting between themselves and/or the user(s) at a given moment. Being a user-focused experience, the management and administration of system applications remain autonomous, albeit challenging. To improve complexity management, reduce the need for human administration of system services, and improve user experience, principles of Autonomic Computing (AC) are incorporated into the CEL. In this chapter, the implementation of self-healing functionality within the CEL is presented.

The experimental CEL system prototype used for this investigation was comprised of 72 web services distributed across 9 hosts running a variety of Unix and non-Unix operating systems. Agents communicate through JSON data objects transmitted through TCP/IP. Agents are implemented in a variety of technologies, in particular Node.js. Agents are ported to Docker containers as part of an on-going project. Self-healing functionality within the CEL is implemented with agents ported to Docker containers.
4.4.1 Implementation

The CEL is a system composed of many interacting web service agents. Broadly, a web service is an application accessible over a network [96, 6], and an agent is an independent entity capable of autonomous action in a given environment [261, 289]. The CEL project is a multi-agent system where each agent is a web service. Throughout the remainder of this chapter, the term web service and agent are used interchangeably within the context of the CEL system.

Self-healing functionality is implemented within the CEL system. The system has the ability to: i) dynamically detect in real-time when a required agent is not functioning; ii) restore the functionality of the given agent; and iii) resume the flow of execution with minimal delay. If a required agent is determined non-responsive, then restoration includes restarting the agent. Running agents within Docker containers improves the restart process. The system architecture implemented for self-healing functionality is presented.

In the CEL system, an agent called the Lifecycle Manager (LM) is primarily responsible for the overall health of the system. In accordance with the FIPA architecture [82], the LM functions as the agent-directory-service. All agents in the CEL register with the LM upon initialization. Agents are provided the LM address, and provide the LM with a name and location for the service. In addition to registration information, the LM also tracks runtime statistics of each agent. The LM continually probes registered agents at fixed intervals to determine whether or not an agent is responsive. The LM is a centralized source of agent information for the CEL, and extra measures and redundancy are enacted to prevent LM failure.

As the agent-directory-service, the LM facilitates inter-agent communication by providing source agents with the IP address and port number of a target agent for which a given source agent is programmed to contact. Upon initialization, each agent provides the LM a list of agents with which the given agent will interact, and the
LM ensures the agents are active. Then, to communicate with a target agent, the source agent contacts the LM to obtain the address and port number of the target agent. If during either agent initialization, or in the process of retrieving a target agent’s location information, the LM discovers that an agent is not functioning, the LM proceeds to heal the downed agent.

The LM utilizes the Discover module to find agents within the agent directory. If an agent is found but non-responsive, the agent is probed to see if the agent can be awoken from sleep. If a probe is unsuccessful, or if an agent is simply not found, then the agent is rebooted. To reboot a Docker container, first the previous container is halted if necessary, then a fresh container is deployed.

As described in Section 4.3, deployment of a Docker container requires that first a Docker image be built from source files, then an instance of the image is deployed as a running container. A running container represents a functioning agent in its final location. A Docker image can be hosted locally or remotely. In the CEL, a Docker image is built locally and then deployed as a container, but experiments were also conducted with importing an image from a remote repository, as well as running a pre-built local image.

4.4.2 Fresh Builds versus Layer-reuse

Docker images are built in layers, making use of a read-only union file system that reduces memory footprints and increases resource utilization by re-using common components in an application. For example, if two web applications run the same software stack, but each run different main Javascript files, the Docker images of the two applications will be built with mostly the same layers, except for the top layer running the main application code. When building an image, the more layers that get reused, the shorter the build time. For self-healing systems, re-using layers will reduce build time and facilitate faster healing and a more responsive system recovery.
The wallclock time for building and running a single agent container is presented in Figure 4.1. When no previous layers of the agent are present on the system, all layers of the image must be built before the agent can be run. For my experimental testbed, agents are homogeneous but have different system configuration files (e.g. for agent IDs), which requires a new image build, but some layers can be re-used from previous agents. If a specific agent image has already been built, then running doesn’t require an image build, just a container run. The average times for building an image and running a container for these 3 scenarios are presented in Table 4.1.

Experiments are also run for building and running several agents in parallel. The building and running of 5, 10, 15, 20, and 25 agents is tested. In these scenarios, the image for the first agent must be built fresh, with no previous layers, but subsequent agent builds can re-use some of the previous layers. The results in Figure 4.2a show wall-clock runtime increases linearly with the number of agents built and run, depicted with the dashed-line where the y-intercept is the runtime for a fresh build, and the subsequent increase is the build time for utilizing some of the previous image layers. Figure 4.2b shows that user runtime and system runtime relative to wall-clock time are roughly constant.
4.4.3 Local Images versus remote images

In addition to storing a built image locally, Docker also supports storing images in remote repositories. Public repositories hosted by Docker include commonly utilized base images like operating systems and web software stacks. Users can publicly or privately store personal images through Docker. Users can also host their own private, remote Docker repositories. I set up a private Docker repository to host agent images and test building and running agents using remote images.

Similar to how the Docker engine re-uses previously built layers when building an image locally, the same principle applies when pulling remote images, in that Docker checks the available local layers to only remotely pull the necessary remote layers.
TABLE 4.2

PULL AND RUN IMAGES FROM REMOTE REPOSITORY

<table>
<thead>
<tr>
<th>Layer Re-use</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>45.78</td>
</tr>
<tr>
<td>Some</td>
<td>10.18</td>
</tr>
<tr>
<td>All</td>
<td>2.20</td>
</tr>
</tbody>
</table>

4.4.4 Serial versus Parallel

We investigate building agents in parallel. First, after building and running the first agent using no previous images, multiple threads are spawned to build and run agents in parallel using the previously built image layers. Since up to this point, all images and containers have been built using the same set of source files with minor container-specific informational changes (e.g. lifecycle-manager registration files), a mechanism is put in place to copy source files into new locations, so parallel processes don’t operate on the same folders. Copying the source folders takes on average an additional 2.82 seconds per image.

To build images in parallel, I modify the Docker server virtual machine. By default, the Docker server uses only one virtual CPU, so parallel run times would be expected to be close to serial run times for the default configuration, plus additional time for copying. The current system uses VirtualBox virtual machine software for the Docker server, and for experiments is reconfigured with 2 and 4 CPUs. Experiments were run for building and running 5, 10, and 15 containers. Each set-up was run five times, and average run-times are presented in Figure 4.3.

Overall, some improvement was observed in increasing the number of cores, however performance was very inconsistent. For building and running 5 containers, a 12% speed-up was observed from 1 to 2 CPUs, and 2 to 3 CPUs. For 10 containers,
run time actually increased. For 15 containers, run time decreased drastically from 1 to 2 CPUs, but then for 4 CPUs the program crashed every time. Some of these results cannot be explained, possibly due to the fact that both the docker server and underlying VirtualBox software are both still under development. A larger number of containers were not extensively tested because of crashing.

4.5 Network Simulation

A simulation test-bed for a network of synthetic web service agents has been developed to further empirically explore the self-healing functionality. The simulation results presented here better illustrate the network-effect of agent healing, because the interaction network of agents can be altered, rather than fixed to a specific system architecture such as the CEL.

The simulation test-bed is comprised of homogeneous synthetic web service agents that communicate with one another according to specifications provided as input. With agents represented as nodes, and messaging represented as edges, the system
can be portrayed as a directed network. An example network depicting a topology
for 10 agents, generated with the NetworkX package for Python and visualized with
matplotlib, is presented in Figure 4.6.

By depicting the system as a network of agents, I can distinctly quantify the
"unhealthy" state of the system in this simulation by tracking the number of missed
messages resulting from a downed agent [91].

4.5.1 Network Models

When designing an autonomic system, it is important to understand the underly-
ing structure of the system in order to illicit top performance. I observe a multi-agent
system of web service agents as a network, where nodes are the agents, and the edges
or connections between nodes represent one agent communicating with another. Var-
ious types of networks exist, and I explore 3 network models to illustrate how software
system architecture can impact autonomic capability.

Random Graph A random graph follows a random distribution of edges according
to a predetermined probability. The random failure of a node will impact the network
depending on its degree.

Scale-Free A scale-free graph follows a power-law distribution of edges. A small
number of hub nodes have high degrees, while the majority of nodes have few degrees.
While the random failure of any given node will likely be minimal, in the case of hub
node failure, such as the agent-directory service [82] or the Lifecycle Manager of the
CEL, the system will be challenged.

Small-World A small-world graph has a small average path length. In the Watts-
Strogatz model, nodes have a uniform degree distribution with few neighbors of
Figure 4.4. Graph Models and Degree Distribution
neighbors, but a few random edges make other clusters of nodes a small number of hops away. Failures can impact node neighborhoods, but not the whole system.

Understanding the topological structure of a system can help with self-healing.

4.5.2 Evaluation

Using the empirical results from previous sections, I evaluate how the different self-healing scenarios (i.e. layer re-use, remote image storage, and parallelization) impact different system architectures.

We consider four variables for a system utilizing this self-healing functionality:

- **Healing Latency** - How long it takes for an agent to be healed. The healing process begins when another agent attempts to contact the non-responsive agent through the Lifecycle-manager. How long it takes an agent to heal, that is to build and run a new Docker container, depends on the factors, including whether or not a fresh build is required, whether the image is stored locally or remotely, and in the case of multiple downed agents, if healing can occur in parallel.
• **System Architecture** - The patterns of interaction amongst agents. I model system architecture using a directed network, with nodes as agents and directed edges representing communication. I consider four models of network topology: random, small-world, scale-free, and a tree. If an agent goes down, it will immediately effect its neighbors, which will then in turn effect the neighbors of neighbors, and so on, in a network effect of cascading failure. The characteristics of a given cascade depend on the network topology.

• **Communication Model** - The frequency of interaction between agents. Agents communicate at fixed intervals, while the length of the interval is varied.

A fourth component for evaluating the network is the failure model, how other agents respond to the failure. However, different failure models are not explored at this time. In this evaluation, once an agent goes down, all dependent agents go down once the next message at the specified interval is missed. For some systems, an application may be able to continue without all required components, but for mission-critical, real-time systems such as the CEL, all dependent agents are required for a given component to function.

Considering these three variables and the failure model, I evaluate how different network topologies handle one or more failed agents. More specifically, if an agent goes down, an agent will fail depending on its distance, or number of hops to the downed agent. If an agent’s hop-distance to downed agent multiplied by the time between messages is less than time it takes to revive the agent, then the agent in question will also go down. In my model, since agent management is centralized with Lifecycle-Manager, the Lifecycle-manager can track when an agent failure is because of a cascading failure, and can thus suspend activity rather than have an agent outright fail. So the agent will remain functional and responsive, while its application will not work.
Figure 4.6. Representation of an example network of 10 agents. Agents are nodes, and directed edges between nodes represent agent dependencies, where the bolded ends of edges designate direction. Edges can be bidirectional. In the network simulation, an agent requires messages from neighbors of inbound edges to operate. The simulation tests how the self-healing of malfunctioning agents impacts a system for random, small-world, and scale-free network topology models.

4.6 Conclusions and Future Work

Self-healing functionality within the CEL Symbiotic Cognitive System is presented. Autonomic computing is employed by the CEL to manage the complex system of interacting web service agents. Self-healing is a critical component in autonomic computing, and utilizing Docker for agent restoration is asserted. Self-healing functionality using Docker containers is implemented within the CEL system. Furthermore, a simulation test-bed is developed to empirically evaluate the network-effect of self-healing with Docker containers. Docker containers are light-weight virtual machines for web applications that as a platform meet several challenges posed by autonomic computing. Future work will apply Docker build latencies to a network simulation model representing relationships between web services.
CHAPTER 5

EMERGENT PROBLEM SOLVING WITH MANET SWARMS

5.1 Overview

The utilization of Wireless Charging Vehicles (WCV) within Wireless Rechargeable Sensor Networks (WRSN) emphasizes a common, recurrent behavior in Mobile Ad Hoc Networks (MANET) - that of a mobile node i) starting from a central, possibly mobile base station, ii) departing the base station in search of an object or location of interest, and, upon detection of the object or location, iii) returning to base. This Search-and-Return (SAR) design pattern is prevalent in many applications. For such systems, I propose a swarm-intelligent approach that utilizes emergent problem solving based on the simple behavior and stigmergic environment of ant systems. The proposed system is robust, scalable, completely decentralized, and requires no non-local communication. Moreover, a simple update mechanism guides nests to areas of higher food density, reducing the cumulative distance between base stations and objects of interest. The system is, to the best of my knowledge, the first biologically-inspired ant system to incorporate moving nests, and clustering capability is evaluated by an agent-based simulation.

---

1Presented at the Agent-Directed Simulation track of the 2014 Spring Simulation Conference

2Awarded Best Student Dissertation

3Developed into "Control of Artificial Swarms with DDDAS", presented at ICCS 2014
5.2  Introduction

Micro-electro-mechanical system advancement has enabled development of low-cost, low-power sensor nodes. Mobile ad hoc networks (MANETs) are wireless sensor networks (WSNs) made up of mobile sensor nodes that communicate without a fixed infrastructure or centralized administrator. Nodes can sense and interact with the environment, perform on-board computation, and communicate with other nodes as well as a central base station. Communication may be routed through intermediate nodes. Nodes can be autonomous or centrally controlled. Since nodes are battery powered, system lifetime is an important consideration for application design.

5.2.1  Wireless Rechargeable Sensor Networks

Breakthroughs in rechargeable lithium batteries and wireless energy transfer have led to Wireless Rechargeable Sensor Networks (WRSNs), where nodes can be wirelessly recharged up to 3 meters \[146\]. Systems have since been developed that utilize wireless charging vehicles (WCVs), mobile vehicles with high volume batteries that wirelessly recharge sensor nodes \[206\]. Progress in such systems is evident in the commercial availability of WCVs \[275\] and the establishment by the Wireless Power Consortium of an international standard for wireless charging interoperability. Well-designed WRSNs offer potentially infinite system lifetimes.

Many wireless charging protocols have addressed different types of WRSNs. Protocols have been designed for a single stationary charger to support many mobile sensor nodes \[299\], for a single WCV to charge many stationary nodes \[206\], for multiple WCVs charging many stationary nodes \[275\], and for sensor nodes to collaboratively charge one other \[304\]. Often, systems incorporating WCVs borrow concepts from Message Ferrying, a technique for efficient data transfer in sparse WSNs. With Message Ferrying, rather than nodes broadcasting data over costly distances, power is conserved via mobile nodes transporting data between stationary nodes and base
stations\[308\]. One prevailing organization for data ferrying is a three-tiered architecture, with a tier of central base stations, a tier of stationary sensor nodes, and an intermediate tier of mobile transport nodes\[238\]. Systems have also been proposed that incorporate mobile nodes for both tasks of data gathering and wireless charging\[307\,99\].

5.2.2 Search-and-Return Behavior

Whether wirelessly charging or data ferrying, mobile nodes transporting resources between a base station and outer nodes perform a simple, recurrent behavior of seeking out a location of interest, then returning to base. Specifically, a mobile node departs a base station in a search state for an object or location of interest. Upon detecting a signal representing the object or location, the node transitions to a return state and heads back to base. The node may stay at the base station for a length of time, until another signal triggers the node to set out again in search of another object or location. The behavior, deemed Search-and-Return (SAR) behavior, is depicted in Figure\[5.1\].

The SAR behavioral design pattern is prevalent in many MANET applications. For example, systems utilizing Unmanned Aerial Vehicles (UAVs) for fire-fighting\[254\] exercise SAR behavior when UAVs search for fires, deploy a fire-retardant payload, then return to base for refill. Moreover, UAV systems delivering medical supplies\[72\] must replenish from a central repository after providing for the field. Almost any system characterized by the three-tiered transport node architecture demonstrates SAR behavior. However, though widespread, SAR systems generally require a high degree of coordination, communication connectivity, and computationally-intensive centralized planning, resulting in a complex, energy-intensive system in an often dynamic environment. Application designers, in contrast, strive for systems that are robust, fault-tolerant, flexible, scalable, and computationally undemanding. For such
a system, inspiration is drawn from biological swarms.

5.2.3 Swarm Intelligent Systems

Swarm intelligent systems are characterized by emergent problem solving capability, where simple behaviors aggregated across many agents give rise to complex phenomena, rendering the collective system greater than the sum of its parts [29]. Common examples are observed in nature, such as the coordinated flocking of birds, or ant colonies uncovering efficient paths to a food source. In ant colonies, ants follow pheromone trails deposited in the environment by other ants. Stigmergy describes systems where agent modifications to the environment impact future actions of other agents. Swarm intelligent systems exhibit many advantageous properties, including robustness, flexibility, scalability, and decentralization.

In this chapter, I present a swarm intelligent system for SAR-based MANETs, a system dubbed the SM-Swarm. The system implements digital pheromones to facilitate a stigmergic environment for MANETs, resulting in a SAR solution that is fault-tolerant, scalable, and completely decentralized, requiring no (non-local) communication, connectivity, central control or global perspective. Furthermore, unlike any previous biologically-inspired ant system, the SM-Swarm implements mobile nests, where nests are guided toward areas of higher food concentration, reducing the cumulative distance between food and nearest nests. An agent-based simulation evaluates the effectiveness of what is essentially a decentralized clustering technique. The agent-based simulation and evaluation are discussed in section 4, preceded by a system overview in section 3, and previous work is discussed next in section 2.

5.3 Prior Work

Most work on the system lifetime performance bottleneck of WSNs and MANETs has focused on methods of energy conservation. An early example is observed in the
LEACH protocol [109], where instead of nodes transmitting sensor data directly to the sink, data is routed through local cluster-head nodes of rotating assignment. Other energy conservation approaches include data ferrying, where mobile relay nodes transport data between source and destination nodes in sparse networks [10].

Still, energy conservation methods fail to solve the problem of finite system lifetime presented by battery powered sensors. Previous efforts to overcome limited energy storage include energy harvesting, where energy is extracted from the environment through means such as solar panels, wind turbines, or from other sources, including heat, light, radio, or vibrations [202]. Such techniques experienced limited success, in part due to heavy dependency on environmental conditions.

Recent breakthroughs in wireless recharging [146] have turned attention to protocols for WRSNs. Common instances of WRSN include a single mobile charger among stationary sensor nodes. In such systems, designers often outline a performance metric, prove an NP-hard reduction, and formulate a centralized heuristic or approximation utilizing global information. like location. In [206], a WRSN prototype is developed, where the path of the WCV is planned using a greedy heuristic for the Traveling Salesman Problem. In [243, 292], the WCV traverses a Hamiltonian cycle to optimize the ratio of idle time to the renewable energy cycle. For systems with stationary chargers, [263] investigates optimal node deployment and routing arrangements through reduction of the 3-CNF SAT problem. In each case, global

Figure 5.1. Search and Return (SEAR) behavior of Wireless Charge Vehicles (WCV)
information is presumed available, and centralized computation is performed. While the comparison between mobile charging and data ferrying is often explicit, some approaches go so far as to combine the techniques, and form a system with a mobile vehicle capable of both recharging power and collecting data. [118] explores maximizing the number of sensor-captured events by jointly scheduling a WCV and node duty-cycling. In [307] the J-MERDG protocol is proposed, where a WCV, utilizing global knowledge of sensor locations, plans an efficient path between sensor nodes. A joint routing-planning scheme is proposed in [155] that implements energy-balanced and energy-minimum routing. [99] improves upon [307] and [118] by incorporating additional time-varying energy consumption models.

Few WRSNs implement multiple mobile chargers. [304] was the first to propose collaborative charging, where mobile nodes coordinate a rendezvous in a one-dimensional system to wirelessly recharge one another. [275] investigates minimizing the total traveling cost of multiple chargers while ensuring no nodes fail, leveraging concepts from Named Data Networking to delivery energy status information. To the best of my knowledge, [173] is the only work to study distributed behaviors of multiple mobile chargers.

In the SM-Swarm, distributed behavior among multiple mobile chargers draws inspiration from swarm intelligent systems. Swarm intelligence is an active subject of interdisciplinary research founded in the study of complex systems. Swarms are capable of self-organization and exhibit decentralized control, rendering the swarm both robust and scalable. The emergent problem-solving ability of ants through pheromone deposits has been developed into an effective meta-heuristic for solving challenging combinatorial optimization problems, such as the Traveling Salesman Problem [74]. Swarm robotics, and more broadly swarm engineering, are disciplines that apply principles of swarm intelligence to engineering problems. Swarm methods for the command and control of UAVs has received attention.
The central nest foraging problem is an instance of Search-and-Return behavior originally observed in ant colonies [255], and more recently applied within a computer science context [201 74], where a nontoroidal grid world consists of a nest location and \( N \) food source locations. Beginning from the nest, ant agents leave in search of food, and upon finding a food source, become laden with food and return to the nest. For my application, the problem is expanded to accommodate \( M \) nest locations, where \( M \geq 1 \).

5.4 System Overview

A system is presented that can be applied to a range of MANET problems. These problems can be or have been addressed through instances of SAR behavior, where mobile transport nodes start from a type of base station, leave the base station to search for objects or locations of interest, and upon discovering an object or location, return to a base station. In the proposed system, the mobile transport node does not need to return to the base station from which the mobile transport node started. In the interest of decentralized design, base stations are non-unique.

The proposed system draws inspiration from swarm-intelligent systems, such as ants. I call this proposed system a SAR MANET Swarm, or SM-Swarm. Agent-based modeling, a natural approach to modeling swarms, is employed to test the SM-Swarm. Following convention, MANET components of the SM-Swarm will be referenced with swarm terminology, in this case an ant systems. Namely, a base station is referred to as nest, a WCV is an ant, and a stationary outer sensor is food. The SM-Swarm is depicted within the three-tiered MANET design patter in Figure 5.2.

The SM-Swarm operates in a stigmergic environment, meaning the ants in the SM-Swarm communicate indirectly through methods facilitated by the environment. Communication is via digital pheromones, represented by a positive integer value
stored locally. The agent-based simulation of the SM-Swarm takes place in a non-toroidal grid, which supports each square in the grid storing the pheromone value. A location’s pheromone value can be sensed from any of the 8 neighboring squares (all 4 sides and 4 diagonals are called the Moore neighbors), and ants can increase the value when positioned on the square. Practically speaking, while research into digital pheromones is on-going, one implementation utilized small RFID tags placed throughout the environment [68].

In general ant systems, ants are more likely to move to locations with a higher pheromone concentration, and over time, higher concentrations emerge along the shortest path from nest to food. Ad hoc methods, such as an internal compass, are then employed for the ant to return home. In [20], two types of pheromones are deposited by ants, so a pheromone gradient is followed in both SAR states (see Figure 5.2).
5.1), leading ants to food and home depending on state. In contrast, the SM-Swarm is unique because pheromone gradient only leads back to the nest.

In the SM-Swarm, pheromones are deposited by ants while searching for food, and then the pheromone gradient is followed when returning to the nest. Leaving the nest, ants search in a random manner that slightly favors moving to a square the ant did not immediately come from. During the random search for food, ants deposit pheromones. The value of the pheromone deposit is the highest, or maximum, value when leaving the nest, and decreases at a constant rate every step. Over time, a pheromone gradient is formed that is highest around the nest, and decreases with distance away from the nest. When food is found, ants stop depositing pheromones, and begin deterministically following the gradient. From every square, an ant can sense the Moore neighbors’ pheromone value, and move to the highest value neighbor. The gradient, the path followed of increasing pheromone, leads back to a nest. At the nest, the ant is reset with the maximum pheromone value and begins searching again. From this system, a gradient emerges that allows any ant to find a way back to a nest, from any location, without any communication other than sensing Moore neighbor pheromone values. For further technical and behavioral details on agent-based models of ant swarms, and effectual visualizations of the pheromone mechanism, refer to [201].

Adapting swarm intelligence techniques to MANETs, in the form of the SM-Swarm architecture, is one contribution of this chapter. The SM-Swarm demonstrates properties of swarms that are desirable in MANETs, including robustness, scalability, and decentralization. The system requires no connectivity, and ants return to nests without any communication. The SM-Swarm can be applied to many SAR problems.

A second contribution is the addition of a simple behavior designed to reduce the cumulative distance between nests and food. Unlike many biologically-inspired ant system models, the SM-Swarm system in this chapter supports mobile nests, since
MANETs can have mobile base stations. The simple behavior, a location update mechanism, is added to nests, which moves nests closer to food.

The behavior is biologically-inspired from ants counting their steps [287]. In the SM-Swarm, once an ant finds food, the ant counts its steps and tracks direction back to the nest. Upon returning to the nest, an ant reports the distance traveled. The nest stores a running total of distance traveled by returning ants, the information is stored in an array of size 8, one for each Moore neighbor, and the array indexed according to the square from which the ant immediately returned. Nests contain a predetermined threshold value, and if any cumulative distance total is greater than the threshold, the nest over moves one square, to the square corresponding to the index of the value that exceeded the threshold. The array is then reset. The process is depicted in Figure 5.3. While information is communicated from ant to nest, the communication is local, from two agents at the same location, not unlike an ant reading a pheromone value. The size of the communicated message from ant to nest is bounded by the diameter of the environment, that is, the distance traveled back to the nest is $O(d)$.

The presented update mechanism is simple for multiple reasons. Ant systems and other swarm intelligent systems emphasize simple, uncomplicated behavior, so for the first ant system (to the best of my knowledge) to implement a moving nest, the mechanism remains simple. The ant reports the Euclidean distance traveled, the nest adds the distance to a value in an array, and if the value exceeds a constant-value threshold, the nest moves one square. Opportunity for enhancement may exist, especially depending on the distribution of the food. Nonetheless, the update mechanism is designed for nests to move closer to food, and ultimately reduce the cumulative distance between food and nests in the system.
5.5 Agent-Based Simulation

An agent-based simulation of the SM-Swarm was developed using MASON, an agent-based modeling toolkit for Java [165]. As a heterogeneous agent system, an agent class was designed to represent the nests, and an agent class was designed to represent the ants. Food locations are static in a 120-unit by 120-unit non-toroidal grid. Two types of food scenarios were tested, one scenario with food placed randomly at the start of a simulation, and another scenario with food placed deterministically into 4 square patterns, with four food to each pattern. Both fixed and random food placement scenarios comprise 16 food locations.

The predetermined layout of food into 4 separate squares is depicted in Figure 5.4 and Figure 5.7. Specifically, 4 20-unit by 20-unit squares are placed in each quadrant of the environment, with top-left food locations of squares located at (20, 20), (20, 80), (80, 20), and (80, 80).

While the number of sensors in WSNs and MANETs can vary by several magnitudes, SM-Swarm simulations were run with 100 ants. Simulations were run for 10,000 time steps. Each simulation has 4 mobile nests randomly placed in the environment at the beginning of the simulation. For a given set of parameters, 100 simulations were executed. These settings were determined acceptable by face validation. With random food placement, food locations are randomly assigned at the beginning of each simulation. For the two scenarios, fixed food placement and random food placement, the movement threshold parameter for nests was varied from 1 to 30. A summary of parameters is presented in Table 5.1. Figure 5.7 shows the progression of a simulation run for fixed food placement.

The SM-Swarm utilizes swarm intelligence and a stigmergic environment for decentralized SAR behavior, and incorporates mobile nests to reduce the distance ants travel from food to the nest. The effectiveness of mobile nests in the SM-Swarm
TABLE 5.1

PARAMETERS FOR THE SM-SWARM AGENT-BASED SIMULATION

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Duration</td>
<td>10,000 time steps</td>
</tr>
<tr>
<td>Environment</td>
<td>120x120, Non-Toroidal</td>
</tr>
<tr>
<td>Number of Agents</td>
<td>100</td>
</tr>
<tr>
<td>Number of Nests</td>
<td>4</td>
</tr>
<tr>
<td>Movement Threshold for Nests</td>
<td>[1,30]</td>
</tr>
<tr>
<td>Number of Food</td>
<td>16</td>
</tr>
<tr>
<td>Food Placement</td>
<td>Placed, Random</td>
</tr>
<tr>
<td>Evaporation Constant</td>
<td>0.999</td>
</tr>
<tr>
<td>Update Cut-Down</td>
<td>0.9</td>
</tr>
<tr>
<td>Obstacles</td>
<td>None</td>
</tr>
</tbody>
</table>
is evaluated by the cumulative distance of food to nearest nests, a reasonable approximation for where pheromones guide an ant after finding food. The Voronoi Tessellation based on nest locations is overlaid on the simulation environment in Figure 5.7, where each food location within a Voronoi cell is closest to the cell’s nest.

Figure 5.5 show simulation results for 5 different nest movement thresholds when food is randomly placed throughout the environment. The cumulative distances for the 100 simulations are depicted as box plots every 500 time steps. While the cumulative distance of food to nearest nests varies between simulations, results show that distance always decreases over time, and the range of distances narrows over time. Results suggest each nests may converge to respective locations based on the initial placement of food. Varying the threshold doesn’t appear to impact the cumulative distance value after 10,000 time steps, but the cumulative distance value is reached at a faster rate with a lower threshold.

For fixed placement of food, the cumulative distance of food to nearest nests follows similar patterns. Regardless of nests’ start locations, the cumulative distance decreases as nests move over time. Cumulative distance appears to roughly converge as well. Cluster purity is an accepted measure of clustering correctness, and is used to evaluate the clusters based on the food closest to each nest. Figure 5.6 shows results for simulations of fixed food placement with nest movement thresholds of 1 and 10. The effectiveness of the SM-Swarm is clearly evident in the uniformly decreasing cumulative distance of food to nearest nests over time. For over half of the simulations with a threshold of 1, the purity is a perfect 1, meaning each square of food is closest to exactly 1 distinct nest. These results demonstrate the simple behavior governing mobile nests can be used for effective decentralized clustering.
5.6 Conclusion and Future Work

A recurrent design pattern in MANET applications, called Search-and-Return behavior, was identified. A system based on principles of swarm intelligence was introduced that implemented simple node behaviors within a stigmergic environment. The proposed system overcame many energy-intensive challenges with current methods by self-organizing into efficiently located clusters. The proposed method is robust, scalable, requires neither connectivity nor (non-local) communication. Agent-based simulations confirm the effectiveness of the system’s decentralized clustering mechanism, as base stations of the MANET converged to locations that significantly reduced the cumulative distance between the base stations and the outer sensor nodes. The proposed system quantitatively demonstrates the capabilities of swarm intelligent systems and is applicable to several MANET problems. The system is one of the first stigmergic environments adapted to MANETS, and the first ant system to incorporate mobile nests for decentralized clustering.
Figure 5.3. Demonstration of nest update mechanism. In each nest, an array stores a running total of distance traveled by returning ants, indexed by the returning direction. When the total exceeds the threshold (an input parameter) the nest moves one space according to the index. In the above figure, food is black, a nest is gray, the arrow is the path of an ant, and the threshold is 4. (i) An ant returns a distance 3, less than the threshold, (ii) An ant returns a distance 5, (iii) the nest moves right one square, and resets the array.

Figure 5.4. Test scenario with square patterns of food placed in each quadrant. Ideal system behavior would minimize the cumulative distance between food and nearest nests by updating nests to the center of the square.
Figure 5.5: Threshold sweep for random food placement
Figure 5.6. Bar charts for the cumulative distance of food from nearest nests for nest update thresholds of 1 and 10, for 100 simulations with food deterministically placed in 4 square patterns.
Figure 5.7: Screenshots from an agent-based simulation of the SM-Swarm. For this scenario, to start, 4 nests are randomly placed, and 16 food objects are deterministically placed in 4 square clusters. For illustrative purposes, images of candy and mobile base stations overlay the simulation’s visualization objects of food and nests, respectively. Ants are the many black dots. No pheromones have been deposited at the start of the simulation in 1, the beginning of pheromone deposits are visible around the nests in 2, and pheromones are visible in varying shades throughout the environment in 3-6. The simulation overlays a Voronoi Tessellation of the environment based on nest locations, and is a good approximation for the nest an ant would return to after locating a given food object. Reduction of cumulative distance from food to nests is observed over time. In addition to predetermined food locations scenario, simulations were also run for random food placement.
CHAPTER 6

INVESTIGATIONS OF DDDAS FOR COMMAND AND CONTROL OF UAV SWARMS WITH AGENT-BASED MODELING

6.1 Overview

The application of Dynamic Data Driven Application Systems (DDDAS) to the command and control of swarms of Unmanned Aerial Vehicles (UAVs) is being investigated. Swarm intelligent systems are not only efficient at solving group-level problems, but also decentralized, controllable by few simple parameters, making possible the command and control of UAV swarms by a single operator. Four separate but related projects are surveyed that explore the command and control of UAV swarms. Each project employs the DDDAS paradigm, entailing the ability of an executing application to incorporate dynamic data into the decision process, and conversely, to steer the measurement process via a central application system. By providing an overview of DDDAS approaches to UAV swarm mission scheduling, UAV swarm communication, UAV swarm formation planning, and flocking applications, general principles of UAV swarms and DDDAS architecture may be observed.

1Presented at the 2013 Winter Simulation Conference

2Collaborators on this work include Dr. Yi Wei, Rachael Purta, Mikolaj Dobski, Artur Jaworski, Alex Madey, and Dr. Brian Blake

3 Developed into book chapter “UAV Swarm Command and Control with Agent-Based Modeling” and accepted for Dynamic Data-Driven Application Systems by Dr Frederica Darema et al.

4 Developed from ”Swarm Control of UAVs for Cooperative Hunting with DDDAS”, ICCS 2013
6.2 Introduction

The applications of Unmanned Aerial Vehicles (UAVs) have proliferated with recent technological advancement. Since UAVs do not require an on-board pilot, vehicles are less expensive than piloted counterparts and offer greater design flexibility, such as smaller size and increased mobility. Control is administered either autonomously by an on-board computer, by one or more remote ground operators, or by a combination of both. Driven by increased accessibility, UAVs are now utilized across a variety of domains, including real estate, agriculture, police surveillance, and parcel delivery. Military applications include reconnaissance of unknown territories, monitoring hazardous environments, and opportunities for minimizing risks to safety of personnel.

As costs reduce and capabilities improve, coordinated teams of UAVs will likely supplant single UAVs for more complex applications. However, current control methods limit the effectiveness of such teams. Each UAV requiring one or more highly-trained ground pilots does not scale well as the number of airborne UAVs increases. Centralized control approaches frequently lead to exponential increases in communication bandwidth requirements and software complexity. To solve problems cooperatively while maintaining scalability, application designers are investigating swarm intelligent methodologies for teams of UAVs.

Swarm intelligent systems are characterized by emergent problem solving capability, where simple behaviors of individual agents give rise to complex phenomena, rendering the whole greater than the sum of parts. Examples are commonly cited in nature, such as the flocking of birds, or ant colonies foraging the shortest path to food. Swarm control is decentralized, with communication performed locally, so the swarm is both scalable and robust. Agents can be added to the swarm with minimal overhead, and the failure of any single agent is of minimal detriment to global performance.
Swarm intelligent behaviors overcome current challenges with UAV teams by offering efficient, cooperative problem solving facilitated by decentralized control. Previously, three challenges to the implementation of swarm command and control were identified [172]. These challenges are (1) the need for near real-time dynamic command and control of the swarm, (2) efficient mission planning and dynamic real-time retasking of the swarm, and (3) the need for improved automation of swarm mission planning and command and control. This chapter presents work addressing these challenges from different focuses (Wei, Madey, and Blake 2013; Purta, Nagrecha, and Madey 2013; Madey and Madey 2013; McCune and Madey 2013).

All projects are part of an on-going investigative effort, sponsored by the Air Force Office of Research, to explore effective modeling and control methods for operating future UAV swarms within a Dynamic Data-Driven Application System (DDDAS) [65]. DDDAS entails the ability of an executing application to incorporate dynamic data into the decision process, and conversely, for the application to dynamically steer the measurement process. A dynamically controlled measurement process means DDDAS supports continual simulation calibration. Since swarm models exhibit nonlinear dynamics stemming from unpredictable agent-agent interaction, incorporating swarm simulations into a framework allowing repeated simulation calibration will improve simulation accuracy, making swarm modeling within DDDAS a compelling topic of investigation.

The rest of this chapter is organized as follows. In Section 2, a brief introduction to the background of DDDAS and UAV swarm control and operation is given. Section 3 starts with an overview of the swarm project, followed by detail descriptions of each individual sub-projects. In Section 4, I summarize this chapter and discuss future work.
6.3 Background and Related Work

Projects surveyed in this chapter apply principles of DDDAS to the command and control of UAV swarms. Each project is designed to incorporate an agent-based simulation into a UAV swarm scenario. Agent-based modeling and simulation is an intuitive paradigm for modeling swarm intelligent systems. Background is provided on swarm intelligence, agent-based modeling and simulation, and DDDAS, then work related to the surveyed projects is presented.

6.3.1 Background

6.3.1.1 Swarms

Though the term "swarm" may often refer to a specific biological instance of swarm intelligence, such as a swarm of bees, a swarm may also more generally refer to any system exhibiting swarm intelligence. Swarm intelligence characterizes a system of simple, interacting agents that collectively exhibit complex phenomena, where the whole is greater than the sum of the parts. Two common examples are observed in nature. The flocking of birds can be modeled as a swarm, where the collective flock formation emerges from each bird adjusting direction and velocity based solely on neighboring birds [225]. Similarly, ants in ant colonies communicate indirectly through pheromones, a mechanism through which the shortest path to food is uncovered while foraging [30]. Both examples illustrate decentralized control, where agents communicate only locally while collectively accomplishing a global task. Decentralization supports system scalability, since agents can be added without creating communicative bottlenecks. Moreover, decentralized systems are robust, since the failure of a single agent will not cascade throughout the system. The emergent problem solving ability of swarms contrasts with traditional, centralized approaches to problem solving.

119
6.3.1.2 Agent-Based Modeling

Agent-based modeling and simulation is an approach to representing a system as autonomous agents that interact amongst one another as well as with the environment. Agent behavior is designed to capture local interactions, through which, after repeated interaction, the collective characteristics of a system emerge over time. Agent-based modeling is an intuitive paradigm for representing swarms. Modeling a system at the agent-level is considered a bottom-up approach, in contrast to conventional top-down models, like equations. Notably, the bottom-up approach can capture the generative nature of system properties, leading researchers to consider agent-based simulation as "a third way of doing science," [16]. While agent-based models offer tremendous explanatory capability, the rapid interaction between agents can quickly lead to unpredictable, non-linear results, requiring careful model calibration. Model verification and validation also challenge the paradigm.

6.3.1.3 DDDAS

DDDAS entails the ability of an executing application to incorporate simulated data into the decision process, while conversely being able to dynamically manage sensors to refine measurements [65]. New sensor data is assimilated into the running simulation, enabling a synergistic feed-back and control-loop between the real-world application and the simulation model. Synergy is realized through a simulation capable of modeling complex, non-linear dynamics in faster than real-time. Sensor controls continually drive the measurement process in order to recalibrate the simulation and provide precise, meaningful results. Since agent-based simulations often require detailed calibration to avoid unpredictable dynamics, incorporating an agent-based model within a DDDAS framework minimizes model inaccuracy by repeatedly recalibrating with new data. An accurate model can support application optimization via simulation.
6.3.2 Related Work

Challenges imposed by UAV swarms have attracted much research attention, including new simulation tools, simulation models, and command and control mechanisms to address all aspects of swarming. Previous simulation work has utilized MASON [166], a general purpose multi-agent simulation library for Java, as well as the Matlab-based UAV simulator MultiUAV2 [222]. In [86], a multi-UAV simulator is implemented on top of a commercial flight simulator called X-Plane, and another tool is presented in [229] that utilizes an existing parallel emulation and simulation tool called SPEEDS. Agent-based models of search and destroy strategies are presented in [89]. Similarly, cooperative search is discussed in [272]. Path planning and vehicle routing problems are explored using multi-objective evolutionary algorithms in [149]. Another decentralized strategy for command and control of UAV swarms is modeled in [88]. DDDAS is applied to multi-UAV scenarios in [132].

6.4 Technical Approaches

6.4.1 Overview

We discuss four projects that investigate swarm control methods with agent-based models. While synthesized swarms aim for fully distributed and decentralized systems, UAV swarms require an element of control from a central operator, no matter how minimal. Each project adapts a swarm behavior to a UAV mission and explores the role of a central controller. The application of the agent-based model to the DDDAS framework is also discussed.

The first project explores mission scheduling within the swarm. The second project augments two swarm algorithms with an additional behavior to optimize the search for mobile targets. The third project compares swarm performance based on communication. The fourth implements two swarm scenarios directly applicable
to the DDDAS framework.

6.4.2 Swarm Mission Scheduling

One of the interesting and challenging problems in UAV swarm control and navigation is mission scheduling. In this study, a long endurance swarm is considered. The swarm is capable of carrying out multiple missions, as well as accepting new missions on the fly. The goal then, is that when given a swarm $S$, either with or without existing missions, and a new mission $MS$, how to schedule the mission onto the swarm so that the total mission cost is minimized.

It is assumed that a mission consists of multiple tasks. And a task can be finished by any UAV. The scheduling process produces a mapping, or an assignment, $M$, from the set of tasks $T$ to the set of UAVs, $S$. The mapping specifies which task is assigned to which UAV. For example, $\{t_1, v_2\}$ means UAV 2 gets task 1.

The mission cost is the sum of all its tasks’ costs. The cost of a task is different for different types of tasks. Generally, it consists of the traveling cost and the task completion cost. The former is the cost for a UAV to travel to the task location from its current location, while the later is the cost for the UAV to carry out the task.

6.4.2.1 Global-Local Hybrid Planning and Scheduling

To solve this problem, a hybrid approach is employed which combines global task assignment and local task scheduling. Two types of agents are used. One is the Swarm Control Agent (SCA), and the other is the UAV agent (UA). A swarm operation environment includes a single SCA and multiple UAs. The SCA is the interface between the human operator and the swarm, it usually runs in the ground station and its design objective is to decompose a newly arrived mission into multiple tasks, assign those tasks to UAVs, and monitor the status of the swarm and missions. On the other hand, a UA represents an operational UAV in the swarm which is
capable of completing different tasks. Agents communicate with each other through messages. These messages either require the recipient to take an action, or contain the latest information about the sender. Six types of messages are used by the agents. For a complete description of these messages and their purposes, refer to Wei et al. [281].

The SCA is also delegated with the tasks to monitor task dependencies as well as mission and swarm status. It needs to decide when a task is ready to be assigned. A ready task is the one that doesn’t have any predecessor tasks or all of its predecessors have been finished. As a task is completed, other tasks that are dependent on it may become ready. For each ready task, the SCA selects the assignment target UAV by the following steps:

1. Calculate, for each UAV, the cost of finishing the task. The SCA will use its latest knowledge about the swarm to estimate the cost of finishing it. This cost is different for different UAVs, because UAVs are on different locations and may or may not have existing tasks. There are two situations:
   1. The UAV doesn’t have scheduled tasks. In this situation, the cost is the traveling cost from UAV’s current location to task location plus the task’s completion cost. The capacity used for comparison is the UAV’s current residual capacity;
   2. There are scheduled tasks on the UAV already. In this case, the estimated final location and estimated final residual capacity values are used to calculate the task cost.

2. Sort the calculated costs and choose the UAV with the smallest cost value as the candidate for the task.

The task assignment gives a task to a UAV, but it does not specify how this task should be scheduled on the UAV. To be more specific, the SCA does not specify whether the new task should be executed immediately or after all existing tasks are completed. The UAV controls this decision. Several policies are implemented to carry out this local task scheduling process. A sorted task list is produced on each UAV after the scheduling process, which represents the completion order of the tasks. The UAV then follows the task list to complete all tasks.
The proposed swarm mission scheduling framework is implemented as a Java multi-threaded simulation program. Both the SCA and all UAs are represented by threads. Figure 6.1 shows the GUI of the simulation. The left part of the GUI has the mission output area on top and the control buttons at the bottom. The testbed visualizes missions by displaying their tasks and task dependencies as DAGs. Each task node also has a color indicating the task status.

At the bottom left of the GUI are the control buttons. The Start button starts a new simulation. The Stop button terminates a running simulation. The Pause button is used to pause/resume the simulation.

The top area of the GUI’s right part is the swarm display area. The swarm is visualized in this area, including all its UAVs, their default waypoints and all tasks scheduled on these UAVs. A UAV is represented by a small solid triangle on the GUI. The capacity of the UAV is shown by a small solid rectangle near the UAV. The area below the swarm display area is used for textual output of simulation information, such as task assignment results, task and mission completion results, and simulation termination message. For more detailed descriptions of the simulation and all the experiments, please refer to Wei et al. [281].

6.4.3 Formation Planning

Swarm intelligent systems utilize simple local behaviors to collectively solve complex problems, where the capability of the group is greater than the sum of its parts. Emergent phenomena in swarms, the mechanism to collectively solve problems, often occur non-linearly and may be difficult to predict. For real-time systems utilizing swarm intelligence, modeling possible control decisions using agent-based simulation can improve performance. I developed a simulation that can be implemented within a DDDAS framework to support search formation assignment [180].

Adapting swarm methods to relevant problems in the UAV domain is challeng-
ing. One recently problem proposed is Cooperative Hunting \[8\], where agents work together and search an environment in such a manner as to guarantee all intelligently evasive targets are found in a finite amount of time. Their work introduced two swarm search algorithms, the Parallel Path algorithm and the SWEEP Protocol, depicted in Figure 6.2 and Figure 6.3 respectively. Both algorithms implement agent formations in such a way that a mobile target, with a predetermined speed, cannot avoid detection.

The two algorithms come with trade-offs. Parallel Paths works as a simple sweep back and forth. The formation is easy to implement and requires little communication between agents other than coordinated movement. However, in current form, the formation is only able to effectively search rectangular environments. Conversely, the SWEEP Protocol is capable of searching any connected shape when provided enough agents, but at the expense of significant increases in complexity. The SWEEP Protocol contains intricate rules, can be difficult to implement, and requires considerably more communication and computational power, since an agent must determine
whether or not its location keeps the search space connected [7].

With these trade-offs in mind, this project implements an additional behavior, the Sentry agent, that allows for the search space to be partitioned. The Sentry is a stationary agent that patrols a single location, equipped with a stronger sensor capable of searching with a larger radius. A central controller can assign Sentries to locations that partition the search space into sub-regions, depicted in Figure 6.4. Considering the trade-offs between each formation, the number of agents and the agents formation can be selected to optimize particular parameters, such as time, communication, or robustness. Furthermore, UAVs may fail over the course of a mission, necessitating formation reassignment. The simulation gives the operator a
tool to test if certain formation assignments will successfully complete the mission.

Figure 6.4. A central controller assigns stationary Sentry agents to strategic locations, displayed in red, to partition the search space into sub-regions. Trade-offs between the two search formations make partitioning the search region an optimization problem. Implemented in a DDDAS framework, the simulation supports dynamic formation reassignment when UAV failure impacts mission objectives [180].

The addition of the Sentry adds flexibility to formation assignment in the Cooperative Hunters problem. While the search swarms act deterministically and autonomously, assignment comes from a central controller. Agents in a UAV swarm may fail, so a simulation tool is helpful for dynamic reassignment during a mission. This agent-based simulation can be implemented within a DDDAS framework so real-time information can update the model, and, if necessary, aid the central controller in dynamic formation reassignment.

6.4.4 Communication

Another project investigated intra-swarm communication and effects on performance within a DDDAS framework [215]. In this experiment, a UAV swarm is tasked with discovering all stationary targets within a 2-D environment. UAV swarm mo-
bility was modeled after the Icosystem Swarm Game \cite{28}, a well-known swarm rule set where each agent associates non-mutually with one or two other agents in the swarm, designated as a helpers of a particular agent. For this implementation, UAVs are initialized as only either aggressors or defenders and position themselves according to their helpers, summarized in Figure \ref{fig65}. To control the swarm, the model was augmented with a leader-follower mechanism, where a designated leader agent is selected as the first helper agent of all other agents. The leader agent visits predefined waypoints in order for the swarm to search the entire environment.

Figure 6.5. Agent mobility rules of the IcoSystem Swarm Game. Aggressor and Defender agents were implemented with the addition of a single Leader agent. The Leader is assigned as the first helper of every other agent, and traverses predefined waypoints throughout the environment to ensure quality coverage \cite{215}.
For efficient searching, UAVs should be positioned far enough away from one another so that search sensors do not overlap. However, for proper mobility, each agent must maintain a communication link with two helpers in order to correctly reposition itself. The baseline implementation required all agents to be within a given communication radius of one another, at the expense of overlapping search coverage and increased risk of collision. Another implementation supported multi-hop communication, permitting agents to be farther away from helpers. Experiments were conducted with the baseline and multi-hop implementations to evaluate performance in terms of number of collisions and number of targets discovered. For the baseline experiments, agents were first positioned randomly about the environment, then later were modified to be evenly distributed around a circle in the center of the environment, with the search radius increased for each agent. Experiments with the multi-hop communication swarm had the same waypoints, initial positions, and sensor parameters as the second set of baseline tests, with communication routes determined by the path costing the least amount of energy.

The architecture of the experimental test-bed employs the DDDAS paradigm. A multi-agent simulation, programmed in Java utilizing the MASON multi-agent simulation toolkit, forms the foundation of the experiments. Simulation input and output is facilitated by RESTful web services running locally on an Apache Tomcat server. Web services communicate between the simulation and a separate user console, capable of displaying real-time UAV data. Real-time data may be any real-world representation of a UAV, whether physical drones or a separate simulation, implemented for proof-of-concept. The architecture is depicted in Figure 6.6.

The mixed results of the experiments support the need for a DDDAS framework. While multi-hop communication was hypothesized to increase the number of targets discovered, results showed that fewer targets were found on average than in baseline tests with non-random initial placement. However, since multi-hop configuration
minimized total energy used by the swarm to communicate, some situations may still find such a design beneficial. Other performance trade-offs were identified, for further discussion of results, refer to [215]. Intra-swarm communication was demonstrated to effect swarm performance in complex ways an operator may not immediately able identify or understand. The DDDAS implementation of this test-bed allows for a simulation to be executed with real-time data in faster than real-time. From the console, an operator is able to seamlessly adjust mission parameters based on simulation output, giving the operator an effective support tool for complex missions.

Figure 6.6. The experimental test-bed incorporating DDDAS architecture. Experiments utilized an agent-based simulation developed in Java using the MASON toolkit. Web services support both the real-life representations of UAVs driving simulation input, as well as the simulation output dynamically steering the real-time measurements of the UAVs [215].

6.4.5 Swarm Control Through BOIDS

Lastly, a fourth project explores UAV swarm command and control with BOIDS flocking behavior [171]. One of the earliest examples of swarming behavior [225], BOIDS models the emergent flocking formation of birds through with three local
rules that govern an agent flight vectors:

- Alignment- agents align their heading with their neighbors
- Cohesion- agents steer towards a group of neighbors
- Separation- agents steer away from their neighbors if they get too close

Subsequent rules were added that refine the flight vector by considering both current and predicted locations of specific neighbors [225, 248]. By manipulating the influence of each rule on the flight vector, the swarm can be controlled. Two surveillance scenarios were developed to evaluate the effectiveness of BOIDS for UAV swarm control.

6.4.5.1 Plume Coverage

The first scenario models the surveillance and tracking of a contaminant plume, building upon previous work in the domain [64, 140]. In a 2D space, the contaminant plume is represented as a circle with the highest contaminant concentration in the center and dissipating outward towards the edges. The objective of the UAV swarm is to provide the controller with an accurate map of the plume and the distribution of the contaminant. Each UAV is capable of measuring the concentration from its current location, and broadcasting measurements to nearby neighbors, within a radius controlled by the swarm operator. Agents follow the cohere rule and navigate towards the neighbor with the highest reading, with a steering radius also controlled by the operator. The simulation displays the standard deviation of sensor measurements in real-time, providing the operator with statistical information to support control decisions. An example of utilizing swarm control to achieve mission objectives is depicted in Figure 6.7.
Figure 6.7. Illustrative example of swarm command and control with the BOIDS flocking rule set for a contaminant plume surveillance mission. The left-most image shows UAVs gathering around the highly concentrated center of the plume, controlled with a tight turning radius. In the center, real-time statistics reveal that the UAVs are not adequately dispersed about the plume, so the controller reduces the agent turning radius to cover more of the plume and obtain a greater deviation in measurements, seen right [171].

6.4.5.2 Target Tracking

The second scenario implements two types of flocking agents, Searchers and Pursuers, to model the search and tracking of mobile targets. Searcher agents navigate through the search space according to the original three BOIDs rules, with the objective to maximize coverage. When a target is detected, a Searcher communicates the location to a Pursuer agent, which implements BOIDS seeking behavior to intercept the target at a predicted position. The ratio of Searchers to Pursuers is controlled by the operator and can be varied based on performance metrics, including total detections, successful pursuits, and the number of identifications for each unique target. Similar to the previous scenario depicted in Figure 6.7, real-time statistics support swarm control decisions of the operator. Figure 6.8 illustrates the target-tracking simulation.
6.5 Conclusions and Future Work

UAVs will continue to play an important role as technology advances. Four projects were presented that explore swarm control of UAVs. While swarms are inherently decentralized, UAV swarms in some manner must be centrally controlled to allow for a ground operator. Each project explored different aspects of swarm control implemented through an agent-based simulation and applied to the DDDAS framework. The mission scheduling project introduced an agent-based control framework for assigning missions based on minimum estimated cost. Through cooperative search, I demonstrated how swarm formations can be centrally assigned before operating autonomously. A third project demonstrated the impact of communication protocol on swarm behavior. Lastly, the swarm control scenarios demonstrated mis-
sion objectives achievable through BOIDS flocking behavior.

Many areas for potential improvement exist. For mission scheduling, improved realism can be obtained by accounting for non-static speed and collision avoidance. For cooperative search formations, an expression of parameters can be formulated in order to automatically optimize formation assignments. For communication, sorting of waypoints and found targets can be implemented to minimize total distance traveled by the swarm. Finally, realism can be improved in the contaminant plume scenario by incorporating weather factors.
CHAPTER 7

QUANTIFYING SWARM PERFORMANCE WITH AGENT-BASED MODELING

7.1 Overview

Swarm intelligent systems are characterized by emergent problem solving capability, where simple individual behaviors give rise to complex phenomena, making the whole greater than the sum of the parts. While common examples exist in nature, recent advancements have led efforts to design, measure, and control artificial swarm intelligent systems, a discipline called swarm engineering. Applications include the command and control of Unmanned Aerial Vehicles (UAVs), allowing a single remote operator to pilot the swarm.

Agent-based modeling and simulation is an approach to modeling a system as autonomous agents interacting between one another, as well as with the environment. The ability of an agent-based model to capture generative behavior of the system under investigation makes the approach a sound choice to model swarm intelligent systems. The agent-based paradigm is not without challenges, but implementing an agent-based model within a Dynamic Data-Driven Application System (DDDAS) for the purpose of swarm engineering can enhance capabilities of both the model and application. By continually recalibrating the model with real-time data, parameters are tuned based on model output allowing the application to be optimized via simulation.

1Revision accepted by the Journal of Artificial Societies and Social Simulation
Quantifying performance for swarm intelligent systems, and then optimizing based on this metric, can be non-trivial, since behaviors are identified at the agent-level while performance is an aggregate measure of a higher-order phenomenon. In this chapter, I design a swarm intelligent system for the command and control of UAVs within a DDDAS framework that utilizes a recently proposed swarm performance model to optimize performance via simulation. I consider swarm performance as a function of the number and size of agent clusters, and with simulation results demonstrate how a single operator can control swarm clustering.

7.2 Applications of Swarm Intelligent Systems

Swarm intelligent systems are groups of autonomous agents that each perform simple behaviors, but together are capable of solving complex problems, generally referred to as swarm intelligence [29]. In a swarm intelligent system, each agent executes a simple behavior that, if acting alone or in a small group, would accomplish relatively little. When combined over many agents, however, a more complex behavior emerges, and together the agents are capable of solving problems of a higher order than the system. Examples of swarm intelligence are most commonly observed in nature, such as the coordinated flocking of birds, and ant colonies uncovering efficient paths to a food source. Though each agent in a swarm interacts locally and with no centralized control, collectively the swarm is capable of solving problems that are outside the scope of any single agent. While the term swarm may refer to a biological instance of swarm intelligence, such as a swarm of bees, a swarm may also more generally refer to any system exhibiting swarm intelligence, and will be referred to as such within the scope of this chapter unless otherwise specified.

In addition to emergent problem-solving, swarms are robust and scalable. For comparison, robustness in scale-free networks is founded in the low probability of random hub-node failure. In the Pareto, or similarly the hub-and-spoke, model of
networks, a commonly occurring, but sparsely connected spoke node can fail with minimal detriment to the system, since the node shares only a few connections. Conversely, hub nodes are rare, but share many, often the majority, of connections. Hub nodes are central to the network and essential for the network to function. Although the probability of a hub node randomly failing is low, such a failure would be catastrophic, like if JFK or Heathrow airport closed. Swarms do not share this vulnerability. Agents are homogenous (at least to a degree depending on the swarm), where no agent is more integral than another, and the failure of any agent will equally impact the system. Agent interaction is also local and distributed, so adding or removing agents doesn’t significantly stress system communication or cause bottlenecks, as is observed in centralized systems.

Swarm intelligence has evolved into a useful problem-solving paradigm across disciplines. The foraging behavior of ant colonies inspired the metaheuristic Ant Colony Optimization for combinatorial optimization problems \[\text{[73]},\] and has been successfully applied to several computationally challenging problems including the Traveling Salesman, a well-known NP-Hard problem. Another technique called Particle Swarm Optimization, inspired by the flocking of birds and related to evolutionary computing, iteratively calculates candidate solutions in a search-space based on results of neighboring solutions sets \[\text{[125]}.\] The technique has been shown to effectively optimize nonlinear equations, including the training of neural networks.

The robustness and scalability of swarms, demonstrated swarm problem solving ability, and technological advancement have broadened the investigation into swarm applications. Swarm engineering describes the development of dependable, swarm intelligent systems for engineering \[\text{[124]},\] including swarm robotics, where autonomous robots are programmed with relatively simple behaviors to cooperatively solve problems. While swarm engineering encompasses the entire process from planning requirements through validation and verification, central to swarm engineering
are behavior design, performance metrics, and swarm control.

Developing an effective swarm is non-trivial. In addition to identifying underlying agent behaviors in order to collectively accomplish a task, swarm engineering encompasses all phases of the engineering life cycle, including the model, design, realization, verification, validation, operation and maintenance of the swarm intelligent applications \[33\]. Swarm performance is also difficult to quantify \[104\]. For example, modeling birds in a flock is possible with only three simple behaviors, but quantifying the performance of the flock is difficult since the emergent behavior takes place on a different perspective than that of the agents.

Researchers are investigating the application of swarm intelligence to the command and control of Unmanned Aerial Vehicles (UAVs). Historically, UAVs have been large, elaborate aircrafts, requiring one or more highly trained remote operators for piloting. Recent advancements have led to smaller, cheaper, sensor-rich UAVs, similar to wireless sensor networks, allowing deployment of several UAVs at once. The number of trained operators cannot scale to the number of deployed vehicles, termed the operator overload problem, leading to the exploration into command and control methods that would allow a single operator to pilot the swarm \[87\].

7.3 Agent-Based Modeling of Swarms

Agent-based modeling and simulation is an approach to representing a system as autonomous agents that interact amongst one another as well as with the environment. Agent behavior is designed to capture local interactions, through which, after repeated agent-agent and agent-environment interactions, the collective characteristics of a system may emerge over time. Modeling a system at the agent-level is considered a bottom-up approach, in contrast to conventional top-down models, like equations. Notably, the bottom-up approach can capture the generative nature of system properties, leading researchers to consider agent-based simulation as a third
This explanatory capability of agent-based modeling, in addition to advancement in computing power establishing large-scale simulations as computationally tractable, has led to a sustained increase in research, least apparent in the number of published articles over the last two decades, depicted in Figure 7.1.

The investigative goals of the system under study influence the approach to modeling. Agent-based modeling is historically rooted in complex adaptive systems, and has been successfully applied across a diverse number of fields including traffic, biology, the stock market, supply chains, the spread of epidemics, and the social sciences \[170, 138\]. Underlying each of these disciplines is the notion of emergent phenomena, where global patterns arise from local interactions, and the collective system possesses greater abilities than the sum of its parts. Agent-based modeling is particularly well suited for the modeling and simulation of swarms.
7.3.1 Challenges of Agent-Based Modeling

Agent-based modeling is not without drawbacks. The high degree of freedom afforded by the agent-based paradigm comes at the expense of added complexity. The objective of an agent-based model is often to uncover the mechanisms causing global patterns to emerge from local behaviors, but no formal frameworks exist to help translate behaviors from micro to macro, leaving crucial design decisions at the discretion of the modeler. Much research has explored formalizing design methodologies, but the adoption of effective frameworks is still an open problem [104]. For agent-based modeling to progress as a paradigm, several challenges need to be addressed that are already well established in other methods, including the choice of appropriate input parameters, how to determine equilibrium for performance analysis, and the validation and verification of the model. Finally, I recommend an approach to agent-based modeling to relax some of these challenges.

7.3.1.1 Input Parameters

The significance of input parameters for an agent-based model underlies the generative nature of the paradigm. In some instances, small changes in a single input can drastically impact the overall result of a simulation, input values characterized as knife edge parameters, [115]. Input parameters can be determined through analytic means or estimated at the discretion of the modeler. Quantitative methods exist to assess the validity of input parameters. Calibration can be considered an iterative process of adjusting poorly characterized parameters in order for the simulation to agree with experimental data [126], but others consider calibration a mere extension of estimation [226] if not model over-fitting. Sensitivity analysis is another approach to evaluating input parameters, and refers to a general family of methods for altering input values in various ways, such as over a range of values, to observe and assess the resulting model behavior. The three major purposes of sensitivity analysis include
corroborating central results, revealing possible variations, and exposing interesting areas warranting further investigation [226].

7.3.1.2 Analysis at Equilibrium

Analysis of a simulation is typically performed at equilibrium [226]. Conventional models, e.g. equation-based models, often impose the assumption of equilibrium from the onset, as aggregate system statistics are provided as input and automatically result in the measure of a stable system. Agent-based modeling challenges the conception of modeling a system at equilibrium, and offer an opportunity to re-evaluate under what perspective to consider a system at equilibrium. Equilibrium may be assessed at the micro level, where behaviors stay constant, or at the macro-level, where a global convergence is achieved. Some have even utilized agent-based models precisely because of their ability to model systems out-of-equilibrium, such as [13] who discusses economics to be continually in flux. The freedom of perspective in terms of equilibrium offers flexibility, but challenges how and under what conditions models may be analyzed. This signals that a more rigorous assessment of underlying assumptions may be necessary if more formal agent-based modeling techniques are to be evaluated.

7.3.1.3 Verification and Validation

Once measurements have been taken, validating and verifying the results are essential. Model verification is the process of determining whether the computational model is the correct implementation of the conceptual model, including debugging and fact checking. Validity, the most important property of a simulation model, means that the right conceptual model is used [137]. Empirical validation and verification of a model is an open challenge [285, 188]. Often a model is subject to face validation, where a domain expert assesses whether the assumptions, behaviors, and
results are reasonable and sufficiently accurate. Empirical validity includes historical data validation and predictive validation, where the simulation results are compared against past and future data, respectively [126]. Data-driven assessment may be more tangential and objective than face validation, but still raises concerns. Namely, since the purpose of an agent-based model is often to explain an underlying mechanism, comparing results of an agent-based model to another data set does not directly assess whether the model is a good representation of an unknown process [285]. Furthermore, Windrum [285] poses Why should empirical validation be the primary basis for accepting or rejecting a model? Do other forms of model validation exist besides the reproduction of stylized facts? Agent-based models are multi-scale, modeling a system locally, globally, and even in states of imbalance. Empirical validation, while certainly not without merit, does not guarantee to capture and validate the entire scope of system activity, most importantly the emergent relationship between micro and macro behaviors.

7.3.1.4 The Explanatory Approach

While a common purpose of modeling is to arrive at some sort of projection, like the future price of a stock or the population of a species, prediction isn’t always the primary purpose of modeling. Epstein [78] lists sixteen alternative reasons to modeling for prediction, and illustrates the difference between explanation and prediction using earthquakes as an example. Models of plate tectonics explain how earthquakes happen, but do little to predict the time and place in which earthquakes will occur. Taking a similar approach with agent-based models, the modeler should try to understand the fundamental relationships between agents and how those interactions affect a greater perspective of the system. By approaching agent-based modeling with an explanatory purpose, aforementioned challenges take on new form.

An explanatory model is more concerned with understanding the systemic foun-
dation of a phenomenon, rather than predicting specific details or results. With this approach, input parameters and initial states become less significant relative to the underlying mechanisms. While it’s important to understand knife-edge parameters, and how critical values can drastically impact the system, the initial states and parameters become less meaningful. Furthermore, because explanatory models emphasize the interaction amongst agents, the notion of equilibrium is challenged. Analysis may be performed under many conditions including equilibrium, or when agent interactions drive an unsteady global state. Finally, data-driven validation is indicative of predictive models, suggesting new methods of validation for explanatory models be explored.

7.4 Swarm Engineering with DDDAS

To develop an effective swarm application, the Dynamic Data-Driven Application System (DDDAS) framework can be employed. DDDAS entails, the ability of a system to incorporate additional data into an executing application, and, in reverse, the ability of applications to dynamically steer the measurement process, [66]. Simulation models can offer valuable insight to a process, but can become insufficient for real-time applications when measured data diverges from simulated prediction. DDDAS offers a solution to divergent simulations by re-parameterizing with real-time data, resulting in more accurate and useful models, which can in turn yield a more precise application. For the DDDAS implementation of a swarm engineering application, an agent-based simulation is a natural choice to model the system. Suitably, DDDAS can address some of the challenges faced by agent-based modeling.

Agent-based modeling can be applied to real-time systems and implemented within a Dynamic Data-Driven Application System (DDDAS). DDDAS is a paradigm where an executing application can incorporate real-time data into the decision process, and conversely, dynamically steer the measurement process [66]. With DDDAS,
simulations are run concurrently with real-world systems at faster than real-time. Applications include wildfire monitoring and prediction, wireless sensor networks, command and control of unmanned aerial vehicles (UAVs), geological phenomena like volcano eruption [75, 203]. DDDAS couples simulations with real-time data to improve predictive capabilities of the models under dynamic conditions, integrating additional information that may selectively enhance or refine the model [66].

7.4.1 DDDAS Overview

A DDDAS framework demonstrates two primary qualities. First, simulations incorporate live, streaming data online. Typically, simulation input is specified offline, and derived from an empirical data source or the modelers subjective estimations. DDDAS challenges modelers and designers to develop effective methods that incorporate real-world data into an executing application. The inclusion, processing, and simulation must collectively execute faster than real-time in order to provide meaningful results.

Second, the application system should incorporate simulation results by adjusting sensors to refine measurements. By focusing on a more relevant subset of measurements, the system can decrease overall costs, reduce the time required for data collection, and improve the overall quality of the data [235].

7.4.2 Agent-Based DDDAS

Many current DDDAS projects incorporate agent-based simulation. Purta et. al [214] utilize agent-based simulation to model the Icosystem swarm game, a well-known example of how small changes in agent parameters can drastically impact emergent global behavior. Wei et. al [281] incorporates an agent-based model in dynamic scheduling of UAV agents. Past projects have also looked at agent-based models within the DDDAS framework [235].
The challenges with agent-based modeling are complimented by the strengths of DDDAS. As previously discussed, agent-based systems often exhibit non-linear dynamics that can be difficult or too complex to capture. Incorporating an agent-based simulation within DDDAS can improve model integrity. Streaming real-time data repeatedly calibrates the simulation so it doesn’t stray too far from observation, and model parameters can be refined to better reflect what’s happening in the real world. Agents also act as sensors, so refining agent parameters and behaviors in turn steers the measurement process.

Implementing an agent-based simulation within a DDDAS framework can help alleviate some of the models shortcomings. Simulation can be used to calibrate input parameters against the real world by comparing measured output against simulated results. Similarly, simulated runs over a range of parameters can identify sensitive inputs and knife-edge parameters. Though I assert agent-based modeling serves the most utility as an explanatory tool, implementing an agent-based model within DDDAS can supplement the model’s ability as a predictor and improve the quality of the application.

7.4.3 Optimization Via Simulation

One existing technique for utilizing simulations in DDDAS is optimization via simulation, as described by Andradttir [11]. The modeler develops a measure of system performance, whether implicit or explicit, and then compares performance of the real-world application against projected, simulated performance. The measured difference is used to identify errors, recalibrate, and optimize the simulation. The real-world parameters are then adjusted accordingly. Examples are abundant in current projects, including with wildfire prediction [37] and volcanic ash dispersion forecasting [203].
7.5 Swarm Performance Metrics

With agent-based models of swarms, measuring performance can be challenging since emergent behavior occurs at a higher order than system parameters. Control of the system is at the agent-level, so identifying how low-level behaviors affect overall performance is also a concern. Recent research by Hamann [104] has sought to identify a universal swarm performance model that is generally applicable to all swarms. A simulation modeling framework for mobile agents traversing network environments executing distributed algorithms (as discussed in Chapter 2) is presented in Appendix A.

7.5.1 Swarm Performance Model

One property introduced by Hamann [104] is the notion of swarm performance relative to agent density. Within a bounded, constant area, a swarm cooperates to accomplish a certain task. More agents yield greater opportunities for cooperation, which in turn improves performance. Conversely, too many agents can saturate the system and generate obstructive interference, counter-productive interactions that degrade performance.

Through these two qualitative properties, the measure of performance in regards to swarm density becomes clear. Performance at first greatly increases with density to a certain peak level, before gradually degrading after further increases. Hamann [104] introduces a simple model for a constant, bounded area $A$, and a number of agents $N$, where the density of the swarm $\rho = N/A$. The performance of the swarm $P$ is based on $N$, and is the combination of the cooperative component $C$ and interference component $I$, defined by

$$P(N) = C(N)(I(N) - d) = a_1N^b \times a_2\exp(cN) \quad (7.1)$$
for parameters $a_1, a_2, b > 0, c < 0,$ and $d \geq 0$. The model for unconstrained cooperation $C$ was previously employed for swarms as early as Breder [34] and more recently in Bjerknes and Winfield [26]. The interference function $I$, augmented with parameter $d$ for the limit to approach zero, models an exponential decrease, exemplified in previous swarms such as Lerman and Galstyan [151]. For further explanation, refer to [104], who provides several examples of model fits to existing swarm performance data sets. Figures 2(a)-(d) depict parameter sweeps over the four coefficients to help visualize how each term effects overall performance.

7.5.2 Swarm Performance in UAV Applications

To support the concepts outlined by this chapter, I offer a demonstrative example of an agent-based simulation to be implemented within a DDDAS framework. This example demonstrates the explanatory emphasis of agent-based modeling, since the
generative behaviors of flocking are modeled, as will be discussed. While a general model of swarm performance has been introduced, defining performance for a particular swarm is a challenge, as previously addressed, and is currently left to the modeler.

This example demonstrates the explanatory power of agent-based modeling, emphasizing the emergent patterns of a swarm from a few low-level behavioral parameters. Prediction involving individual agents is not required. The experiment models the flocking behavior of Boids [224], using the Flockers implementation that is standard with MASON, a discrete-event multi-agent simulation library for Java [165, 193]. Boids is a distributed model for the emergent flocking of birds, where the behavior of agents is governed by a weighted combination of the cohesion and adhesion to neighboring agents, as well as a degree of randomness. For more details on flocking, refer Reynolds [224].

For this DDDAS scenario, suppose there is a central controller for a swarm of UAVs. The mission involves cooperatively controlling the UAVs from a randomly dispersed initial configuration (note that precise location is irrelevant) to a certain number of flocks, using only the weighting coefficients of the three Boids behaviors cohesion, adhesion, and randomness. The scenario could be, for example, to transition from an unclustered configuration, where coverage of the environment is maximal, to a small number of clusters that together contain all agents, minimizing agent exposure vulnerability. The transition from randomly dispersed to clustered UAVs should happen over a given time period, say 5000 time steps, for an optimal trade-off of coverage to effective clustering. If no time frame existed, a controller could then simply maximize the cohesion factor, so I arbitrarily choose a time horizon of 5000 time steps.
7.5.3 Clustering

For the controller to determine a solution, DDDAS can run several simulations in parallel over a range of parameters. In addition to face validation, where the controller can view an agent-based simulation GUI to visualize the flocks, the system can implement a clustering algorithm to quantitatively validate the flocks. Weka is an open-source data-mining library for Java [103], which includes a suite of clustering algorithms, and the DBSCAN clustering algorithm [79] was used to identify agent clusters. DBSCAN is a reasonable choice because the method is unsupervised, so the number of clusters to be identified does not need to be determined ahead of time, unlike supervised clustering techniques like K-Means. Furthermore, DBSCAN is density-based, which is closely related to the neighborhood parameter of the flocking agents, where the algorithm determines clusters by identifying neighboring points within a given input distance $\epsilon$. Also, DBSCAN is capable of identifying odd shaped, non-linearly-separable clusters, as may occur in flocks. The juxtaposition between DBSCAN and Boids flocking behavior is interesting, as both approaches calculate simple arithmetic based on the neighbors within a given radius, offering a unique perspective on the bottom-up versus top-down approach.

For this implementation, $\epsilon$ was decided to be just slightly higher than an agent's neighborhood distance in order to account for random deviations. The minimum number of agents required for a cluster was 3. Three simulations were run over varying parameters for the three Boids behaviors for 5000 time steps. The choice of parameters is not meant to represent a full sweep, but rather demonstrate the wide range of global phenomena emergent from local behaviors, and the simulation length was determined utilizing face validation of the modeler.

From the results in Figures 3(a)-(c), I observe very different global behaviors emerging from each of the three local parameter sets. The first simulation, depicted in 3(a) had an equal weighting of each of the three parameters, and by the end
Figure 7.3. Parameters conducive to some clustering, as about 80% of agents converged to between 3 and 5 clusters.

Figure 7.4. A parameter set very conducive to clustering. After 500 steps nearly all agents are apart of a cluster. The entire swarm converged to 1 cluster at around 4500 steps.
of the simulation run contained about 4 clusters, collectively comprised of most of the agents. The second simulation, in 3(b), implemented behaviors conducive to clustering, with a relatively higher weighting of cohesion and lesser weighting of adhesion. Two tightly formed clusters formed, containing all of the agents, after about 3500 timesteps. A visual depiction of the simulation under these parameters is shown in Figures 4(a)-(c).

The third simulation is depicted in Figure 3(c), representing conditions not conducive to clustering, with low cohesion, and high adhesion and randomness. Over the course of the run, few clusters were formed with a small number of agents for a short period of time.

The model implemented in the DDDAS scenario represents how agent-based modeling can be utilized for its explanatory power while prediction is not required. Agent-based modeling demonstrated how low-level behaviors can lead to emergent global behavior, allowing a central controller to utilize the model for the given scenario without requiring explicit prediction.
7.6 Conclusions

As a modeling paradigm, agent-based modeling demonstrates much potential because of the ability to capture emergent behavior. The explanatory nature of agent-based modeling challenges traditional modeling conceptions such as prediction, input, system equilibrium, and validation. In this chapter, the challenges faced by agent-based modeling were addressed in three parts. First, an overview of agent-based modeling was presented, attempting to distinguish explanatory modeling from predictive modeling. Next, agent-based modeling was discussed in the context of a Dynamic Data-Driven Application System, and recent research introducing a generally applicable swarm performance model was applied to optimization via simulation. Then an example application of DDDAS was proposed, and an agent-based simulation of flocking behavior was coupled with data clustering. The simulation attempted to demonstrate the utility of an explanatory model in the DDDAS framework, and provided an interesting juxtaposition of a bottom-up model analyzed by a top-down clustering algorithm, where both calculated results based on the distance of agent neighbors. Agent-based modeling continues to grow in popularity and application,
and techniques will evolve with understanding.
CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this chapter, I summarize my research contributions presented in this dissertation. I also outline future work from my projects, and discuss open research questions.

8.1 Contributions

8.1.1 Large-Scale Graph Processing

In Chapter 2, I comprehensively overviewed a state-of-the-art Big Data graph processing technique, where distributed algorithms are executed on graph data, instead of centralized algorithms. Beginning with Pregel [174] and then GraphLab [162, 161], a series of new graph processing frameworks were developed that provide a programming interface for ”thinking like a vertex,” where algorithms are designed from the perspective of a vertex, instead of a graph.

Such an approach improves the scalability of graph processing by reducing the scope of the algorithm. Rather than the algorithm presuming random access to every node in the graph, an algorithm is executed from the perspective of a vertex, only requiring data from neighboring nodes. This improves scalability by only requiring inter-machine communication when two adjacent nodes are partitioned to separate machines.

I provide an original organization of the large body of recent literature on these new frameworks. The organization includes breaking down the frameworks into four
fundamental components: communication, timing, execution model, and partitioning. I also survey additional implementation details, such as frameworks developed for single-machine environments. I especially note hybrid frameworks, which combine centralized and decentralized algorithmic approaches for the best performance.

8.1.2 Pre-Processing for Hardware Acceleration

In Chapter 3 I presented an efficient algorithm for pre-processing graph data for a specific application. The specific application is a custom hardware processing element, which only works when the input data is provided in a certain order. While the order can be achieved through using a simple algorithm many times, I developed a slightly more complex algorithm that achieves the same result more efficiently.

The fundamental algorithm underlying both the naive and slightly more complex algorithm is Lexicographic Breadth-First Search (LexBFS). LexBFS is a specific type of the much more common breadth-first search, and has lots of interesting properties. These interesting properties are what allows the custom hardware application to function when the input data is provided in LexBFS order.

While my more complex algorithm can be proven to be more efficient, I programmed both the advanced method and the naive method in Python, and ran experiments on both real-world and synthetic graph data. The results demonstrated that my more complex method indeed outperformed the naive method. Interestingly, both methods scaled linearly with the number of triangles in a graph. This work is the pre-processing component of a larger graph processing project, which includes the hardware component.

8.1.3 Self-Healing System of Web Service Agents

In Chapter 4 I develop a self-healing system of web service agents. "Self-healing" is terminology from the autonomic computing community, where autonomic comput-
ing explores how software systems can become more autonomous.

The project achieves increased software autonomy by incorporating Docker containers. Docker containers are like lightweight virtual machines, except for individual web services instead of whole computers. My research efforts achieve increased system autonomy by porting existing web services into Docker containers, and then utilizing the containers to restart web services when service failures are detected.

I evaluate three different methods for restarting a web-service with Docker. The first method compares restarting the service using previous Docker components of the service, against building the service from scratch. The second method compares building the service locally versus importing the service from a remote repository of Docker containers. The third method compares building Docker containers serially versus in parallel. Collectively, these results help guide design decisions for different system architectures. Additionally, my work positions the use of Docker within the multi-agent software system community.

8.1.4 Emergent Problem Solving with UAV Swarms

In Chapter 5, I present a new swarm intelligent behavior. The system differentiates itself from previous ant systems by incorporating a mobile nest rather than stationary nest [201]. I apply this new swarm behavior to mobile sensor networks that utilize mobile power chargers to charge stationary sensors.

The emergent behavior of the system allows base charging stations in the sensor network to be intelligently relocated based on the location of stationary sensors. I develop an agent-based model of the system, and compare the final placement of the base charging stations to cluster centers obtained by running K-means clustering for the locations of stationary sensors. While the two quantities were equal or similar across many simulations, the difference is that the placement of the base stations was achieved through decentralized, local-only communication, whereas the K-means
algorithm used global information of the system.

I introduce an original swarm behavior with practical application, and develop an agent-based model to evaluate the swarm’s effectiveness. As I also explore in Chapter 7, the emergent behavior is measured through the use of a clustering algorithm. Like Chapter 7, this work explores techniques for measuring the mechanism that leads local agent behaviors to emergent global behaviors.

8.1.5 UAV Swarms with DDDAS and Agent-Based Modeling

My work in Chapter 6 presents my approach for applying swarm intelligence, agent-based modeling, and the DDDAS paradigm for the command and control of UAVs. The approach is illustrated by surveying one of my UAV swarm applications, as well as three other projects from my group. Further background is also provided for swarm intelligence, emergent behavior, agent-based modeling, and DDDAS.

Swarm intelligence can benefit the command and control of UAVs by increasing self-management among UAVs so remote operators are responsible for controlling fewer individual UAV parameters, and can instead control more UAVs. By incorporating the DDDAS paradigm, further precision can be gained by the UAV swarm in dynamic environments. DDDAS entails the ability of an application to improve precision by using simulation data, which in turn impacts the application’s sensor data that drives future simulations, creating a feedback loop. The four projects in this chapter illustrate how agent-based modeling can be used to incorporate the DDDAS paradigm for UAV applications.

My UAV swarm application presented in this chapter is based off of two swarm behaviors for cooperative hunting [8]. In cooperative hunting, a team of agents search an environment in such a way as to always find an evasive target. I developed an agent-based model for each cooperative hunting behavior, and evaluated which behavior works best for a given environment.
8.1.6 Quantifying Swarm Performance

In Chapter 7, I explored how emergent behavior can be quantified. Emergent behavior is difficult to characterize, especially in an engineering context. Emergence is difficult to quantify because of the different scales on which the local and global behaviors occur. Moreover, while the behaviors exist on different scales, the mechanism that connects the two behaviors is unclear. My work in this chapter investigates quantitative relationships between the local and emergent behaviors.

I took an existing swarm intelligent model and added a quantitative analysis component to the emergent behavior. The swarm intelligent model was the well-established Boids flocking behavior [224]. The Boids model simulates the flocking of birds, where each bird positions itself relative to other birds within a local distance parameter, resulting in emergent behavior of a cohesive, adaptive, decentralized flock. The model is an accepted example of emergence, but the emergent behavior has only been characterized visually by an outside observer, not quantitatively. I applied the DBSCAN clustering algorithm [79] to the flock, and compared the result of the clustering algorithm to the local input parameters. Results of the clustering algorithm on the agent-based simulation show what parameters lead to strong flocking behavior.

8.2 Future Work and Open Research Questions

8.2.1 Large-Scale Graph Processing

The processing of large-scale graphs is of increasing importance and representative of data-intensive computing. Graphs will continue to grow in size, as will the systems that are used to process these graphs, driving an on-going need for improvement in techniques.
8.2.1.1 Distributed and Parallel Systems

Many factors are driving research into methods, techniques, principles, and algorithms for processing large-scale graphs in parallel and distributed environments. Big Data and related technological trends continue to yield graphs that are larger and need to be processed more quickly. The exascale national computing initiative explores the requirements for processing extreme scale data, and graphs are representative of data-intensive workloads. The Graph500 benchmark has used competition to drive research into the breadth-first search traversal of large-scale graphs, a fundamental graph processing component. Underlying the need for distributed and parallel implementations are graph algorithms, which themselves can be designed for parallelism and distributed memory.

Open questions include what general techniques and approaches can be extrapolated from algorithm, application, and system-specific processing implementations. These include generalizations for both parallel and distributed environments, as well as heterogeneous environments with GPUs and FPGAs. Extrapolating more abstract techniques from specific implementations will expose the important problems with processing graphs, as well as improve productivity.

8.2.1.2 Streaming and Temporal Graph Processing

Streaming and temporal graph processing both perform computation on a changing graph. Streaming graph processing is performing computations with one or few updates to a graph, which offers a lot of opportunity for future work. Stream graph processing is difficult because of the significant time and graph scope constraints to computation. There is currently no agreed upon streaming model to analyze the effectiveness of a streaming graph processing solution, which illustrates the opportunity for future work to impact the field. Moreover, the Graph500 committee has recently announced the future development of a streaming graph processing bench-
mark, which should drive significant work in the area, as the benchmark has done for large graph supercomputing. Streaming graph processing may be a future frontier of graph processing research.

Temporal graph processing presents many opportunities for further research. The majority of work in temporal graph processing focuses on processing graphs in batches, though a few novel techniques have been introduced. Temporal graph processing systems are more advanced than Pregel-like systems because, at the very least, they must include a temporal graph storage component. Much opportunity for future work exists for novel temporal graph algorithms, processing techniques, and system architectures.

8.2.1.3 Hybrid Graph Algorithms and Frameworks

In Chapter 2, I noted hybrid frameworks, such as Blogel and Giraph++, that combine the best of both Pregel-like frameworks and centralized graph algorithms, resulting in a scalable system that outperforms Pregel-like frameworks. These frameworks open up a graph partition to centralized algorithms, while maintaining the communication component of distributed algorithms. These frameworks unify the two graph processing paradigms and yield top performance, but are significantly under utilized, as illustrated by the dozens if not hundreds of Pregel-like frameworks, compared to six hybrid frameworks.

Future work exists in further exploring the limits and capabilities of these systems, while also developing a theoretical basis. Though a few specialized hybrid algorithms have been introduced, more work exists in developing algorithms and algorithmic building blocks for the hybrid approach.
8.2.2 Quantifying Swarm Intelligent Systems

Future work for swarm intelligent systems includes further exploration of the quantitative relationships between local and global behavior, and the mechanisms leading to emergence. Emergence remains difficult to describe, and the mechanisms of emergence are not well understood. Basic questions, like what is emergence in an engineering context, remain ill-defined.

To address this open question, future work includes development of a simulation engine for mobile agents traversing a network executing a distributed algorithm. This approach would adapt the ant system model to a Pregel-like network environment. The distributed algorithm of the Pregel environment would allow for agents to detect the pheromone value of neighboring nodes. The network environment would allow for the shortest path global behavior to be measured against the local behavior of the mobile agents, for which lots of prior work exists. The system would provide a model for quantifying emergent behavior, and is further outlined in Appendix [A].

8.3 Concluding Remarks

This chapter has provided a summary of my research contributions, as well as outlined future work and open questions addressed by my work. This dissertation has explored decentralized system, and how algorithmic scope can be reduced to increase scalability. I explored decentralization applied to UAV swarms and large-scale graph processing.

Swarm intelligent systems utilize emergent behavior to solve problems, and are scalable, robust, adaptive. UAV swarms are scalable, but still retain a central coordinator. Naturally occurring swarms in nature are completely decentralized, making one consider if such an approach could be applied to computing systems for infinite scalability. The ideas and frameworks of swarms have guided my development of
swarm intelligent applications.

The distributed algorithms executed by Pregel-like frameworks for large-scale graph processing are similar to swarms in that the reduction of scope improves scalability, but differ from swarms because the resulting global behavior of distributed algorithms is provably obtained from local behaviors. I look forward to further exploring the fundamental computational differences between distributed algorithms and emergent systems.
APPENDIX A

EMERGENT COMPUTATION WITH MOBILE AGENTS AND DISTRIBUTED ALGORITHMS

A.1 Overview

Emergent computation characterizes emergent behavior arising from the interaction of parallel computational agents, emergent behavior that is also a computation. Current applications include telecommunications networks and data centers. Emergent computation may also alleviate problems posed by Big Data. However, models of emergence computation, and related swarm intelligence, are lacking.

An algorithmic framework for modeling emergent computation is proposed. The Mobile Agent Computing and Distributed Algorithms Model (MACADAM) captures emergent computing by comparing the algorithmic complexity of the low-level agent and communication behaviors, against the complexity of an omniscient algorithm that represents the emergent computation of the system. The model situates mobile agents operating in a network that executes a distributed algorithm. Two case studies, of ant foraging and Distributed K-medians clustering,

---

1Submitted to ANTS 2014
2Presented at SwarmFest 2014
3Presented at Michigan Complexity Mini-Conference 2014
4Presented at Johns Hopkins Center for Advanced Modeling Graduate Workshop
5Presented to IBM Watson Graph Processing group 2015
illustrate the proposed model for emergent calculus. The MACADAM framework offers quantitative insight into emergent behavior never previously established, that will support the design and evaluation of emergent system engineering.

A.2 Introduction

An improved understanding of emergent behavior will benefit the development of swarm intelligent systems. Currently, the application of swarm intelligence to autonomous robots is under investigation [22, 178]. Further command of emergent behavior may also alleviate computational challenges of Big Data [27]. However, no generally accepted model of emergence exists, and methods to analyze or quantify swarm intelligent systems are lacking [104]. In this appendix, a framework is proposed that models emergence using algorithmic run-time complexity for multi-agent systems that interact with the environment. The framework provides tools for emergent behavior to be studied within a mobile agent computing context, and allows quantitative comparisons to be drawn between the complexity of agent and system behaviors.

A swarm intelligent system utilizes emergent behavior to solve problems. Emergence can describe how simple, local behaviors lead to a complex, macro behavior, a whole greater than the sum of its parts. Inspired by biology, common swarm examples are cited in nature, such as the coordinated flocking of birds, or the collective foraging of ant colonies. Ant colonies utilize stigmergy, where local agent interaction with the environment influences future decisions of other agents. Emergent behaviors are self-organizing and decentralized, where the emergent structure is unknown to agents, but discernible to an outside observer.

The Mobile Agent Computing and Distributed Algorithm Model (MACADAM) for modeling emergence is introduced. Mobile agent computing is one or more autonomous agents interacting in an environment to complete a task. A distributed
algorithm is an algorithm designed to execute locally across several nodes. The proposed framework situates agents traversing a network node to node, utilizing local information provided at each node. The information at each node is computed via a distributed algorithm. Local decisions are modeled in the form of the agent algorithm, and the environmental behavior is modeled in the form of the distributed algorithm. The emergent behavior of the system, the global behavior arising from the interaction of local and distributed algorithms, is verified by an omniscient algorithm. Emergence is captured by comparing the complexities of the agent and environment algorithms against the omniscient algorithm. The MACADAM framework is useful for implementation of swarm intelligence in telecommunications networks or data centers, where control through global information is impractical and links cannot store information. The framework can also be used as a platform for emergent computation in Big Data with tools such as Pregel [174].

The rest of the appendix is organized as follows. Section A.3 discusses prior work in defining and modeling emergence, as well as emergent computing. Section A.4 introduces the proposed framework, providing an overview of both mobile agent computing and distributed algorithms. Section A.5 provides two case studies that model emergence within the proposed framework. Finally, Section A.6 discusses conclusions and future work. The presented research assumes a familiarity with introductory concepts in algorithms and complexity theory, for further reference, see [176].

A.3 Prior Work

Emergent computing describes utilizing interactions between parallel computational agents to achieve an emergent behavior that is also a computation. The idea was first proposed in [84], and subsequently incorporated into a framework for an emergent calculii in [63]. Emergent computing implementaitons, however, have been
few and far between. [43] provides an overview of emergent algorithms and proposes an emergent protocol for clustering in sensor networks. Emergent algorithms are explored within a network security context in [83].

Algorithmic formulations for emergence include [217], which attempts to define emergence with computational complexity theory, based on Conway’s Game of Life [57]. In the Game of Life, simple computational rules underlying cellular automata give rise to macro patterns, such as the gliders and blinkers witnessed by an outside observer. Accordingly, emergence is characterized as the difference in computational complexity between the underlying micro-process and the macro-process pattern compressed by the outside observer.

Information theoretic computational tools are similarly employed in [239] to identify data-driven causal patterns in self-organizing systems. Information theory is used in [187] to automatically detect emergence in agent-based simulations, by first creating an interaction network between agents. A model of emergence in multi-level systems based on agent-based modeling is proposed in [47]. [31] suggests emergent systems may include processes which are not computable, and proposes exploring possible relationships between emergence and incomputability.

A formal definition of emergence is proposed in [260], which extends the work of [141], by defining emergence through a language-theoretic and grammar-based system approach. In a grammar-based approach, agents are represented by grammars, or words, and the collective system behavior is the resulting language, or set of words, arising from agent interactions. Two separate grammars are developed to represent the execution of agents both together and individually. Building off the notion of ”a whole greater than the sum of parts,” emergence is defined as the difference between the resulting languages of two grammars. [260] proposes a grammar-based framework that addresses potential shortcomings of [141], including the explosion of state space, and permits different agent types. Further background of formal languages in
A.4 The MACADAM Framework

A model for emergence is presented. Simply stated, the model is of agents operating in a network environment that executes a distributed algorithm. The agents and environment interact in such a way that network computation emerges, which can be verified by an omniscient algorithm, which operates on the entire network. This section further describes both mobile agent computing and distributed algorithms, two related models of distributed computing. The role of the omniscient algorithm is also described.

A.4.1 Mobile Agent Computing

Mobile agent computing describes one or more agents operating in an environment to accomplish an objective. While the environment can broadly include continuous space such as 2-D or 3-D coordinate systems, the scope of the proposed platform limits the environment exclusively to networks. A network is a collection of nodes connected by links, and within this scope, is equivalent to a graph, made up of vertices connected by edges. These terms may be used interchangeably.

A.4.1.1 The Mobile Agent Computing Model

A mobile agent computing system can be represented by the tuple \((G, \lambda, Q, p, \delta)\), considered a model of the system:

- \(G\) is a graph representing the environment
- \(\lambda\) defines the labeling of nodes in the graph
- \(Q\) represents the set of agents, with \(k = |Q|\) number of agents
- The initial location of agents is denoted by the function \(p\)
The function $\delta$ is the port numbering of links for each node, how outgoing edges are viewed relative to an agent at a particular node.

In addition to the mobile agent computing model described above, system assumptions must be clearly defined. For example, whether the edges of graph $G$ are directed or undirected. Mobile agent computing systems are generally introduced along with underlying model assumptions. The next section elaborates on system assumptions, identifying key assumptions made for the case studies in Section A.5.

A.4.1.2 Model Assumptions

Various assumptions can be made regarding the network model. Assumptions are at the discretion of the modeler. In Section A.5, two preexisting swarm models previously developed for agent-based simulation are adapted to the MACADAM framework, so assumptions regarding the network environment, such as agent communication, are made in order to most accurately transform the system from the simulation environment to the MACADAM.

In certain mobile agent computing problems, different model assumptions can drastically impact the difficulty of the problem, and thus impact the design of agent behavior. For instance, in a system with an agent attempting to traverse every node in an unknown network, whether or not each node possesses a unique label will significantly effect agent behavior. So whenever designing a mobile agent computing system, an understanding of potential network properties is crucial, relative to the problem at hand.

For any mobile agent problem, system properties and assumptions are clearly identified. For further details on mobile agent computing and various system properties, refer to [139, 42]. Several models exist for agent communication. One model facilitates inter-agent communication when occupying the same node. Another model allows for a pebble, or similarly a bread crumb or token, to be dropped at a particular
node, which can be later read or picked up by other agents. One of the most flexible models of communication is the public white board model, where each node contains a white board that agents can read and write information. The public white board model will be utilized by two case studies in Section A.5.

A.4.2 Distributed Algorithms

The MACADAM framework combines mobile agent computing with distributed algorithms, where mobile agents traverse a network executing a distributed algorithm. Distributed algorithms is a subject perhaps less accessible to non-computer scientists. However, only the fundamental ideas of the subject are necessary to understand MACADAM. This section will broadly introduce distributed algorithms, explain how the performance of a distributed algorithm is measured, and briefly discuss a class of distributed algorithms called local algorithms, which will apply to Section A.5. For more detail, refer to a textbook [169].

A.4.2.1 Overview

Distributed algorithms are algorithms designed to run across many interconnected processors, and can be represented by a network of nodes and edges. In the network, nodes are processors that execute a copy of the algorithm, and are able to communicate with other nodes that share an edge. For distributed algorithms, the network is both an input to the problem as well as the system that solves the problem. Problems include leader election, consensus, and graph problems like shortest paths and minimum spanning trees. The challenge lies in processors running concurrently and independently, each with only a limited amount of information.

Like many properties of a network, interprocess communication can be modeled in different ways. One common model of communication is the message passing model, where nodes communicate by sending messages across an edge. In a synchronous
network, the execution of a distributed algorithm occurs in rounds, where in each round a node generates all messages to be sent, sends outgoing messages, and then receives incoming messages. Nodes are permitted unbounded computation, so any local processes, like say sorting, is not measured. The time complexity of a distributed algorithm is typically measured by the number of synchronous communication rounds required to solve the problem, in regards to the size of the network.

A.4.2.2 Local Algorithms

A local algorithm is a distributed algorithm that runs in a constant number of synchronous communication rounds, where the number of communication rounds is independent of the size of the network [256]. Typically, this means that communication is limited to a fixed radius around the node. The MACADAM implementations in Section A.5 implement local algorithms, of time complexity $O(1)$, because only one round of messaging is required for node output. See [256] for a survey of local algorithms.

A.4.3 The Omniscient Algorithm

The proposed platform combines mobile agent computing with distributed algorithms. Autonomous agents are situated in a network environment, traversing nodes based on information local to each node, while the network itself simultaneously executes a distributed algorithm, independent of agents. Emergent behavior can arise from the interplay of the agents and the environment, and is captured by an “omniscient” algorithm, an algorithm that operates on the network as a whole.

An omniscient algorithm is a graph algorithm. The algorithm takes nodes and edges, as input and provides a structured output based on the algorithm objective. Examples of graph algorithms include Dijkstra’s algorithm for computing the shortest path in a graph [70], or brute force search for exhaustively computing every
permutation in a graph, for a problem such as the travelling salesman [150].

Since the environment is a network, a graph algorithm captures the global pattern of system behavior. The choice of algorithm to capture macro-behavior is at the discretion of the modeler, and does not automatically detect the behavior, as explored in [187]. The modeler is in a sense aware of the emergent behavior a priori. The omniscient algorithm provides a method to quantify the emergent, global behavior.

A.4.4 Quantifying Emergence

The MACADAM framework quantifies emergence by providing a platform to compare algorithmic complexities at both the micro- and macro-level. The algorithmic run-time complexity of micro-behaviors are represented the algorithm executed by the autonomous agents, as well as the communication complexity of the distributed algorithm. These two complexities form a comparison against the run-time of the global, omniscient algorithm.

The approach to modeling emergence as a comparison of micro and macro level behavior is not new [217] [141]. The MACADAM framework is novel in the use of algorithmic complexity to form the comparison, or rather, two different approaches to modeling the same phenomenon.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Level</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autonomous Agent</td>
<td>Local</td>
<td>Input Size</td>
</tr>
<tr>
<td>Distributed Network Environment</td>
<td>Local</td>
<td>Communication Rounds</td>
</tr>
<tr>
<td>Omniscient Observer</td>
<td>Global</td>
<td>Input Size</td>
</tr>
</tbody>
</table>
A.4.4.1 Remarks

The MACADAM framework has been presented, detailing mobile agents operating in a network environment that executes a distributed algorithm. An overview has been provided of both mobile agent computing and distributed algorithms, with references for further detail, including [139] for the former and [169] for the latter. MACADAM may be used as a platform for analyzing emergence, by comparing the run-time complexity of the agents and environment at the local level against the run-time of an algorithm capturing emergent behavior at the global level. The next section will provide two examples of quantifying emergent behavior.

One remark before proceeding regards the comparison between mobile agent computing algorithms and distributed algorithms. In [42], the computational power of the mobile agent computing model is explored relative to a distributed algorithm implemented as a message passing system. Namely, it was proven that, in addition to a message passing system being able to mimic an arbitrary mobile agent system, a mobile agent system can be implemented to mimic an arbitrary message passing system. So while mobile agent computing systems and message passing distributed algorithms are modeled, described, and implemented differently (see Section A.4.1 and Section A.4.2), the underlying computational power of each model is equivalent. This implies that MACADAM can be equivalently described as two distributed algorithms, or mobile agent systems, operating on one another.

A.5 Case Studies

Two case studies are presented that quantify emergence by modeling each system according to the MACADAM framework, then comparing the lower level algorithmic complexities of both the agents and the environment against the complexity of an omniscient algorithm that captures emergent behavior. The first case study adapts
biologically-inspired ant foraging to a model of ant agents operating in a network executing a distributed algorithm, where the ant agents collectively uncover the shortest path to a food source. The second case study examines Decentralized K-medians clustering [179], where a modification to the ant foraging behavior produces emergent behavior resembling K-medians clustering, a known NP-hard problem. The two case studies demonstrate how to utilize the MACADAM platform for modeling micro and macro level behaviors.

A.5.1 Ant Foraging

Several artificial ant foraging models have been developed based on the so-called "central nest foraging" problem, where ant agents in a two dimensional environment, featuring a nest and one or more food sources, collectively search for food. For this case study, the foraging model presented in [201], hereby referred to as the Panait Foraging Model (PFM), is adapted to the MACADAM framework. The PFM is unique in the implementation of two types of pheromones by the ants, which establishes pheromone gradients to both food and the nest. The adaptation of the PFM to the MACADAM will be explained, facilitating analysis of the system's emergent behavior. Refer to [201] for further details of the PFM and artificial ant foraging models.

A.5.1.1 Overview

The PFM models ant agents searching for food to bring back to the nest. Pheromones are deposited by ants into the environment, and while ants begin by searching the environment randomly, ants are more likely to follow paths of higher pheromone concentration. While searching for food, ants deposit a pheromone that leads back to the nest, and after finding food, follow the pheromone trail home while depositing another type of pheromone leading to food. As a result, over time, ants uncover the
shortest path to food, depicted in Figure A.1.

Figure A.1. In the Panait Foraging Model (PFM), ants collectively forage food. Ants begin by searching the environment randomly, depositing pheromones (green) that lead back to the nest. Once food is found, ants follow the green pheromone trail, while depositing a different pheromone that leads to food (blue). Ants are more likely to follow the blue pheromone while searching for food. Over time, the colony uncovers the shortest path to food [201].

The model can be adapted to the MACADAM framework. While the PFM is implemented in a non-toroidal grid world, the grid environment can easily be translated to a network, where each grid corresponds to a node, and neighboring cells are connected by links. Pheromone values may be represented by values stored at each node using the public white board model of communication, where the values may be read and written locally.

For ants to decide where to move, the pheromone values of adjacent nodes must be known, which is not immediately possible within the mobile agent computing model. Thus, a distributed algorithm is implemented where each node messages its pheromone value to all adjacent nodes. This is a local algorithm, with runtime $O(1)$, as only one round of communication is needed for all nodes to know the pheromone value of all adjacent nodes. Once all neighboring pheromone values are collected, the
node, which is permitted unbounded computation, arranges the values into a data structure for the ant agent. The ant agent generates a random number in $O(1)$ time, and based on the data structure provided by the current node, is directed to the next node with the appropriate probability.

Each node obtains neighboring pheromone in $O(1)$ rounds, and each agent decides where to go based on random number generation in $O(1)$ time. These two behaviors, the single round of communication between nodes, combined with the action of each agent, occur in a single cumulative system round, similar to the superstep of Les Valiant’s Bulk Synchronous Parallel model [268]. Over time, or a certain number of supersteps, and depending on the number of ant agents, the shortest path to food emerges. An omniscient algorithm can find the single-source shortest path in a graph in $O(n \log n)$ time. The complexities are presented in Table A.1.

<table>
<thead>
<tr>
<th>Agent Environment</th>
<th>Emergent Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

A.5.2 Distributed K-Medians Clustering

A swarm intelligent system for clustering is proposed in [179]. The clustering problem is deemed Decentralized K-medians clustering because a multi-agent system
self-organizes cluster centroids as to minimize the cumulative distance between the
centroids and data objects. Decentralized K-medians is achieved by the Boomerang
Swarm, comprised of an ant system (previous case study Section A.5.1) with modified
behavior. The simply behaved agents and stigmergic environment yield an emergent
behavior capable of clustering in a manner similar to regular K-medians, an NP-hard
problem [182]. Details and discussion of the Boomerang Swarm are provided below.
For system evaluation, refer to [179].

A.5.2.1 System Overview

The Boomerang Swarm is a modified ant system that positions cluster centroids
to minimize the cumulative distance between centroids and data objects. A cluster
centroid is the geometric center of a cluster, where all data objects closest to a
particular centroid are considered part of that centroid’s cluster. A traditional ant
foraging system, as in Section A.5.1 includes a nest, a food source, and many ant
agents. In contrast, the Boomerang Swarm has many nests and many food sources,
along with many ants. The many food sources represent data objects, and remain
stationary. The many nests represent cluster centroids, so for K-medians clustering,
where \( k \) is the number of clusters, there are \( k \) ant hills. Ant hills are mobile, unlike
the biologically-inspired ant system model.

The Boomerang Swarm warrants the name from the Boomerang behavior of the
ants. Ants begin at one of \( k \) nests, randomly searching for a food source, and upon lo-
cating food, return to the nest. Like the previous ant model, ants deposit pheromones
in the environment while searching for food, and after locating food, follow the same
pheromone home. Unlike the previous ant model, no second type of pheromone is
deposited while returning home, so ants have no pheromone trail to follow while
searching for food. As a result, ants are always randomly searching the environment
for one of many food objects, but once food is found, ants have a pheromone trail
that will generally lead to the closest of $k$ nests.

After returning to the nearest nest, the nest will update location according to the returning direction of the ant. The nest will change position to the previous node the ant occupied before returning to the nest. Before relocating, the nest will deposit pheromone into the environment so the pheromone trail will continue leading to the nest. Over time, as the nests are repositioned, the cumulative distance between nests and food sources is significantly reduced, with the global minimum achieved in many simulated instances [179].

Figure A.2. Test scenario with square patterns of food placed in each quadrant. Ideal system behavior would minimize the cumulative distance between food and nearest nests by updating nests to the center of the square.

The Boomerang Swarm for the Decentralized K-medians clustering problem can be adapted to the MACADAM framework in a manner similar to ant foraging in Section A.5.1. The non-toroidal grid environment is translated a network with cells as nodes and neighboring cells connected by edges. Pheromones in the environment
are similarly represented using the public white board model of communication, and pheromone values are shared between adjacent nodes using a local algorithm in $O(1)$ time. Both ant and nest agents execute a $O(1)$ algorithm. In each cumulative system round, or superstep, an ant agent executes a move, a nest agent updates position if necessary, then the environment shares pheromone values with neighbors.

The emergent behavior is an approximation for K-medians. The K-median problem has been proven NP-hard [182], and an efficient algorithm for solving K-medians is not known. Several heuristics and approximation algorithms exist for K-medians and related K-means [116]. Some of these implementations, such as Lloyd’s algorithm [160], are efficient, but suffer drawbacks like failing to converge on centroid location. The Boomerang Swarm experiences similar drawbacks, and research into comparison and evaluation against centralized clustering algorithms is ongoing. However, initial results demonstrate the effectiveness of the decentralized approach [179], and swarm intelligent systems offer distinct advantages over centralized approaches, in that swarms are robust to failure, scale well for large systems, are adaptable in dynamic systems, and are computationally efficient, as illustrated by Table A.2.

\begin{table}
\begin{center}
\begin{tabular}{l|l|l}
Agents & Environment & Emergent Behavior \\
\hline
$O(1)$ & $O(1)$ & NP-hard \\
\end{tabular}
\end{center}
\caption{Complexity of Local vs. Global Behavior for Boomerang Swarm}
\end{table}
A.6 Conclusions and Future Work

At the heart of the MACADAM framework is the idea of trade-offs, that there are different ways to accomplish the same thing. In standard algorithms, while runtime complexity is an important metric, there are other considerations. Dynamic programming, for example, facilitates a faster run-time at the expense of increased memory. So who is to say which is better? Rather than judge, trade-offs are merely identified, allowing the practitioner to appropriately apply to the problem at hand.

The MACADAM framework identifies trade-offs between single-agent and multi-agent algorithms, as both systems accomplish the same objective. A single-agent implementation certainly has advantages, including sensible implementation. Moreover, beyond telecommunication networks and data centers, how or what modern problems can be adapted to agents traversing a network executing distributed algorithm is not entirely clear.

The recent explosion in data, or the Big Data phenomenon, has fueled the development of new technologies to cope with the dramatic increase in data volume, velocity, and variety. One promising technology is Pregel [174], a computational model from Google that transforms large-scale graph computing problems to vertex-centric programming. So rather than one centralized algorithm, like Google’s Page Rank, operating over an entire graph, Pregel allows for programs to be written that execute on each vertice of the graph. Essentially, any graph can be transformed to execute a distributed algorithm. With Pregel providing a distributed algorithm environment, future work will explore how to incorporate mobile.
B.1 Overview

Vertex-centric frameworks such as Pregel demonstrate how vertex-centric computation improves scalability, but the true benefit of vertex programming is realized by frameworks such as Giraph++ [262] that combine vertex programs with conventional, centralized graph algorithms that are executed on partitions wholly contained within a worker. Collectively, these hybrid frameworks illustrate that the scope of a graph algorithm varies inversely with performance, and to efficiently process graphs of scale, both the vertex-centric and graph-centric perspectives should be adopted. By bringing this principle to light, I aim to guide future graph processing research to exploring the interplay between conventional, centralized algorithms and distributed algorithms.

B.2 Introduction

Many frameworks and approaches have been developed to address the challenges of efficiently processing large-scale graphs [39, 174, 167]. Among those approaches is the vertex-centric approach, which was first introduced through the Pregel framework [173], and then recently surveyed [181]. These "Think Like a Vertex" (TLAV) frameworks are platforms for executing distributed algorithms from the perspective

---

1Refined conclusions from "Thinking Like a Vertex" survey
2Submitted to the 2016 Graph Algorithms Building Blocks Workshop
of a vertex, as opposed to conventional \textit{centralized} graph algorithms that adopt a computational perspective of the entire graph. Adopting the vertex-centric perspective presented by the TLAV programming interface dramatically increases scalability, eschewing the inherent limits inherent to conventional perspectives \cite{167}.

However, vertex-centric frameworks, which are really just platforms for executing distributed algorithms \cite{169}, are only one piece of solution for efficient graph processing. Vertex-centric frameworks have evolved into "hybrid" frameworks \cite{262, 295, 246} that achieve top performance by utilizing the best of both the distributed and centralized algorithmic perspective, where inter-worker communication occurs at the vertex level, but worker computation occurs at the sub-graph level. These approaches illustrate the central idea behind vertex programming: that the scalability of graph algorithms vary inversely with granularity. The vertex-centric approach increases scalability, but graph and sub-graph centric perspectives facilitate faster computation, and top performance is elicited by balancing the two.

This appendix aims to elaborate on this idea of balancing graph algorithmic perspectives. This idea was originally posited amongst a comprehensive survey of vertex-centric frameworks \cite{181}, but here is specifically refined and represented. While original results are forthcoming, previous results are presented that clearly illustrate this idea and are appropriate for the workshop. The goal of this appendix is to help direct future graph processing research by bringing together disparate works. While vertex-centric computing compared to Combinatorial BLAS \cite{39} and related linear algebra libraries may seem divergent, this appendix illustrates how they can be brought together, suggesting future work in exploring the relationship between linear algebra and distributed algorithms, and also continuing research into the limits of distributed algorithms, and combining distributed and centralized graph algorithms.

The rest of this appendix is as follows: Section \ref{B.3} discusses related work, Section \ref{B.4} overviews the 2 types of graph algorithms, Section \ref{B.5} presents how frame-
works have evolved beyond vertex-centric programming and incorporate both algorithmic perspectives for improved performance, and Section B.6 discusses conclusions and future work.

B.3 Related Work

Parallel and distributed graph processing is notoriously difficult [167]. In response, a vertex-oriented processing framework was first introduced by Google through the Pregel framework [174], based on the Bulk-Synchronous Parallel model by Valiant [268]. Pregel and related frameworks that provide an API for "Think Like a Vertex" (TLAV) programming are fundamentally platforms for executing iterative, distributed algorithms [169] on data graphs, adopting the LOCAL model of computation (e.g. [159] [142]) where vertices/processors communicate in synchronous rounds by passing messages of unbounded length, as opposed to the CONGEST model where message sizes are bounded. Soon after Pregel, Graphlab [161] and Distributed Graphlab [162] were introduced that offer a similar vertex-centric programming interface, but adopt an asynchronous, shared-memory computational model, which simply explores different properties of distributed algorithms with accompanying performance trade-offs. Since Pregel and Graphlab, a wide variety of frameworks and related techniques have been introduced that explore different properties of the vertex-centric computational approach, which have been comprehensively surveyed [181].

In [181], TLAV frameworks are broken down into 4 distinct but related components: i) timing - synchronous or asynchronous, ii) communication - message passing or shared memory, iii) vertex program execution model - the staging of computation, and iv) partitioning - how the data graph is distributed to workers. Each facet, particularly the first 3, are inter-related, and design decisions for eliciting top performance depend on underlying assumptions about the data and processing goals, e.g.
random vs. power-law network data and the particular algorithms to be executed. How large-scale graphs are partitioned across workers can also heavily impact performance, and while the min-cut problem is NP-hard and the powerful METIS toolkit is impractical for graphs of scale, several practical methods have been introduced for the TLAV environment, notably streaming partitioning \[266\] and re-streaming partitioning \[196\] for their scalability, parallelizability, and low overhead, among others techniques \[181\].

Pregel-like, or TLAV, frameworks are platforms for executing iterative graph algorithms, a specific class of algorithm. Many algorithms such as PageRank, single-source shortest path, connected components, and many other machine-learning algorithms \[162\] can be re-cast as vertex-centric algorithms, or distributed algorithms \[169\]. TLAV frameworks are closely related to other Big Data computational frameworks like Map-Reduce (in some instances, TLAV frameworks are even run on top of Map-Reduce \[14\]) but the underlying data model is fundamentally different. For instance, Map-Reduce runs in a single-pass over the data and has no loop awareness, so cannot execute iterative algorithms, although iterative Map-Reduce has been introduced \[77\].

Efficient graph processing continues to gain increasing research interest, and many frameworks and approaches, other than vertex-centric, have been recently introduced to address the challenges of parallelizing graph computation \[39, 120, 208, 156\]. This project serves to unify the strengths of the vertex-centric model with the strengths of other models such as graph algorithms in the language of linear algebra \[130\].

B.4 2 Types of Graph Algorithms

Among many possible definitions, an algorithm is simply a standard, black-box, provably correct procedure that takes a defined input and produces an ordered output of a particular property. As computer scientists and problem solvers, who is to say one
algorithm is better than another, provided each achieves its stated objective? Still, procedures can be measured, such as how long it takes to complete in relation to input (time complexity), how much intermediate storage is required (space complexity), can it be parallelized, and of course, empirical evaluation of performance in practice.

Problems may be solved by many different algorithms, but even with algorithmic measures, one cannot always say for certain which is best. Take for example the longest increasing subsequence problem. A naive solution can be easily implemented and computed in $O(n^2)$ time in relation to input. Or, with a pre-sorting of the input which takes only $O(n \log(n))$ time, can be solved in a single pass over the input. Even still, given $O(n^2)$ space, dynamic programming can be utilized for a linear time solution. Which is best? Well, at the least, since no algorithm dominates the other two in terms of both time and space, it depends. As a problem solver, I use whichever method best accomplishes the end goal, and in this case that goal is efficient graph processing.

Similarly, for graph algorithms there, there exist 2 distinct approaches. One is the conventional, centralized approach to graph algorithms, where the computational entity presumes to have an omniscient view of the graph, with every node and edge directly accessible. Alternatively, there are distributed algorithms, where the graph is both the input as well as the environment in which the algorithm is executed. Each approach can be utilized to solve the same problem, e.g. PageRank, single-source shortest path, connected components, etc. With different underlying assumptions and measures of complexity, comparing the two approaches is somewhat apples to oranges, so who is to say which is best? These two approaches have remained separate until now, with the evolution of hybrid frameworks [262, 295, 246]. The two algorithmic approaches are briefly discussed below to introduce how hybrid frameworks combine both approaches.
B.4.1 Conventional Graph Algorithms

For graph algorithms, beyond trade-offs between time and space, there are differences in *perspective*, namely two different perspectives. One perspective is the conventional one, adopted by your standard, centralized graph algorithms, such as Djikstra’s shortest path. The input includes an entire graph of nodes and edges, the procedure operates on the entire input, outputting a result in relation to the entire graph. This perspective is omniscient in the sense that the entire graph is within the scope of the computational entity, that that computational agent is disparate from the graph, performing operations upon the graph data. This omnipotence assumes that all graph data is randomly accessible. For example, a graph algorithm such as Djikstra’s may call for the ”next node of shortest distance to a current node” which is a global property, requiring all graph data to be known and accessible, and calling for the computational entity to pick the one node in the whole graph that meets this global criteria. When all data resides in memory, centralized graph algorithms can be computed efficiently with linear algebra [130].

The assumption of global accessibility is valid in theory, but practically and computationally, with Big Data driving large-scale graphs beyond the memory capacity of conventional machines, the assumption of global accessibility may no longer hold. Challenges to graph processing arise when a central computational perspective is adopted but all graph data does not reside in memory [167]. When a graph does not fit in memory, graph operations eventually follow pointers to data not immediately accessible, requiring data to be swapped out from disk leading to excessive overhead, or utilized in a distributed setting, which introduces a host of additional challenges. The centralized graph perspective is now challenged as large-scale graphs are too large for single machines.
B.4.2 Distributed Algorithms

In contrast to centralized graph algorithms are distributed algorithms. Distributed algorithms, a long established subject of study [169], are algorithms designed to run across a network of interconnected "processors", where each processor executes a copy of the given algorithm. Processors are connected to other processors, forming a graph, and only communicate with adjacent processors to which a processor is connected. The graph, or the processor topology, is both the input to the algorithm and the environment in which the algorithm is executed. The output is collective state of all processors after each algorithm has finished executing. An example of a distributed algorithm and its execution is provided in Figure B.1.

The complexity of distributed algorithms are measured in terms of communication rounds between processors. Each processor is permitted unbounded computation, and the complexity in regards to input, as with centralized algorithms, is ignored. Distributed algorithms were foreseen as executing across computer networks, with each computer as a node. As first presented in [181], Pregel-like frameworks are just platforms for executing distributed algorithms across data graphs, where a data graph...
is partitioned across a set of workers, and each worker is responsible for executing the
distributed algorithm across each node in its assigned partition. While Pregel-like
frameworks have inspired new algorithms for the platform [216], these algorithms
are just distributed algorithms. Many problems, such as PageRank, single-source
shortest-path, and connected components, can be solved using both distributed and
centralized algorithms. Distributed algorithms research continues to explore what
can and cannot be computed, in relation to centralized algorithms [143, 190].

B.5 The Tao of Vertex Programming

Scalability is the key benefit of adopting the vertex-centric processing approach, as
illustrated by results in Pregel [174], which show runtime scaling linearly with graph
size for solving single-source shortest path using the vertex-centric approach. Such
scalability is not possible with a centralized approach, because overhead increases as
the graph spread across more than one machine [167]. Such scalability in vertex-centric
frameworks is accomplished by decentralizing computation, to where a computational
agent (the vertex) only requires information from adjacent neighbors, which limits
costly inter-machine communication to instances where vertices are partitioned onto
separate machines (ignoring synchronization overhead, etc.). The decentralization of
vertex programming improves scalability by reducing inter-machine communication.

Considering that the vertex-centric approach increases scalability while limiting
the scope and expressiveness of graph algorithms, why use vertex programming be-
tween vertices residing on the same machine? Whether intra-machine messaging is
managed by local worker agents in the synchronous setting [174], or managed through
locks in asynchronous shared memory [162], the overhead is substantial, especially
when compared to the execution of a conventional centralized algorithm. A number
of hybrid frameworks have responded to this shortcoming by developing frameworks
that combine the centralized and distributed algorithm paradigm.
First published in the Giraph++ framework [262], followed by Blogel [295] and GoFFish [246], researchers recognized that runtime performance can be improved by abandoning the vertex-centric paradigm for subgraphs, or partitions, that are wholly contained within a single machine. For a graph already in memory, why endure the overhead of vertex messages when a centralized algorithm is available? Vertex computing benefits the distributed environment by reducing inter-machine messages and improving scalability, but increases overhead and reduces performance for graphs residing on a single machine. Recognizing this discrepancy, hybrid frameworks combine vertex programming for scalability with centralized algorithms for speed by exposing two programming APIs: i) one for vertex programs between boundary vertices that have adjacent vertices on remote machines; and ii) another API for executing conventional, centralized graph algorithms on subgraphs, or partitions, wholly contained within each worker. An illustration from Tian’s Giraph++ [262] in Figure B.2 depicts how combining the centralized and vertex algorithm paradigm reduces the number of iterations for connected components. For more detail on hybrid frameworks, see [181] and references therein.

Results within respective papers of hybrid frameworks illustrate the impact of the hybrid approach on reducing superstep runtime, number of iterations to convergence, number of messages, and overall runtime [262, 295, 246]. In Figure B.3, the results from [262] show hybrid m and Figure B.4.

The Tao of Vertex Programming is recognizing that the scope of a graph algorithm varies inversely with scalability, and understanding this interplay between: i) distributed algorithms/Pregel-like vertex programming; and ii) conventional, centralized graph algorithms. Computing with centralized graph algorithms is faster, but when graphs become too large and must be processed in a distributed setting, vertex programming can be employed to improve scalability. Vertex programming is more scalable, but less expressive and with more overhead than convention centralized
Figure B.2. Utilizing both vertex-centric and centralized algorithms for connected component computation reduces the number of supersteps compared to vanilla vertex programming. Hybrid frameworks are more efficient than homogeneous vertex-centric frameworks, but more scalable than centralized computation, which is not practical for large-scale graphs. It is postulated that graph processing is most efficient when utilizing hybrid frameworks with the least number of workers necessary to meet the memory requirements for graph data and intermediate storage, and will be pursued in forthcoming experiments. The idea is expressed in Figure B.5. Ultimately, since hybrid frameworks combine centralized algorithms within a worker and distributed algorithms between workers, hybrid algorithms that combine the two paradigms can be expressed as a product of the computational runtime for a graph algorithm within a worker, and the communication complexity of the distributed algorithm between workers, or the number of iterations until convergence.
Figure B.3. Results from [262] show how execution time and network messages are reduced by using vertex computing only between vertices that reside on different machines. For subgraphs within a worker, a centralized algorithm is used. Results show improved performance for the hybrid method (HM and GM) is superior to the pure vertex-oriented method (VM) for two different methods of partitioning (HP and GP).
Figure B.4. Results from [295] show how block-centric computing (Block-Centric), where centralized algorithms are executed on the blocks of graphs on a worker and then a vertex-program is used to share results between blocks, is superior in performance to Pregel-like Vertex-Centric computing.

B.6 Conclusion and Future Work

The scalability of a graph algorithm varies inversely with scope. While vertex-centric computing has dramatically increased the scalability of some graph algorithms by providing a platform for distributed algorithms, recent hybrid frameworks [262, 295, 246, 181] have empirically demonstrated that executing centralized algorithms within a worker while executing vertex-centric programs on vertices between workers improves performance. By bringing this principle to light, I aim to guide future graph processing research into investigating how low-level linear algebra primitives can be adapted for distributed algorithms, further investigate which problems can be solved by hybrid centralized-distributed graph algorithms, and explore the computational limits of graph algorithms. On-going experiments will also demonstrate that top graph processing performance is achieved when utilizing a hybrid framework with the least number of workers necessary to meet storage requirements for graph data and intermediate computation.
Figure B.5. From top to bottom: A) an example graph, B) an example distributed algorithm where each node is processor/worker, C) 4 workers with 3 nodes each, D) 2 workers with 6 nodes each, E) 1 worker with the whole graph. If it’s assumed that the capacity of a worker can support 6 nodes and also store intermediate computation, then the setup in D is optimal, because a centralized algorithm can execute on each worker and results can be shared between workers through a distributed algorithm. The setup in C is possible, but less efficient than C because there would be more communication overhead. The setup in B represented a true distributed algorithm, but likewise is less efficient than C or D because no centralized algorithm can be utilized.


9. Y. Altshuler, V. Yanovsky, I. A. Wagner, and A. M. Bruckstein. The cooperative hunters-efficient cooperative search for smart targets using uav swarms. In Second International Conference on Informatics in Control, Automation and


82. F. FIPA. *FfIP Agents, Editor*, 2002.


211


