Louisiana State University
LSU Digital Commons

LSU Doctoral Dissertations | Graduate School

2011

Advanced semantics for accelerated graph processing

Dylan Thomas Stark
Louisiana State University and Agricultural and Mechanical College, dstark@cct.lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Computer Sciences Commons

Recommended Citation
Stark, Dylan Thomas, "Advanced semantics for accelerated graph processing" (2011). *LSU Doctoral Dissertations*. 3131.
https://digitalcommons.lsu.edu/gradschool_dissertations/3131

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contactgradetd@lsu.edu.

ADVANCED SEMANTICS FOR ACCELERATED GRAPH PROCESSING

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Dylan Stark
B.S., Louisiana State University, 2004
M.S., Louisiana State University, 2007
May, 2011

Acknowledgements

I have made it as far as I have with the untiring support of my parents, Daniel and Beth Stark, and my brother Michael. They have always supported me in my various endeavors, no matter how *long* it takes. Also, I am very grateful for the good friends I made along the way. In many ways they made the trip worth taking.

I thank my advisor, Professor Thomas Sterling, for providing me with such a great opportunity to work with him in the ParalleX group. His vision and determination were inspiring, and his mentoring and support unwaivering. I am indebted to him for the chance to the see through my personal goals. I would also like to thank my committee for the guidance and support they have shown me throughout my graduate career.

I thank Dr. Gabrielle Allen and others at a newly formed Center for Computation and Technology at LSU for giving me the opportunity to pursue graduate research in High Performance Computing. Special thanks go to the Cactus Team: Dr. Allen, Dr. Seidel, Tom Goodale, Thomas Radke, Eric Schnetter, and Yaakoub El-Kahmra.

Finally, I thank my colleagues in the ParalleX group at the Center for Computation and Technology at Louisiana State University and Sandia National Laboratories. In particular, Hartmut Kaiser, Chris Michael, and Phillip LeBlanc at LSU, for humoring me, and taking me seriously, only at the appropriate times; and Rich Murphy, Arun Rodriguez, and Kyle Wheeler, for the real world introduction to so many things multithreaded, message-passing, and graph.

This work was partially supported by Sandia National Laboratories. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Table of Contents

Acknowledgements . . . . **ii**

List of Tables . . . . **v**

List of Figures . . . . **ix**

List of Algorithms . . . . **x**

Abstract . . . . **xi**

Chapter 1: Introduction **1**
- 1.1 Research Objective 3
  - 1.1.1 Goal 3
  - 1.1.2 Hypothesis 3
  - 1.1.3 Objectives 5
- 1.2 Technical Strategy 6
- 1.3 Dissertation Outline 7

Chapter 2: Background **9**
- 2.1 Graph-Based Application Processing 9
  - 2.1.1 Multithreaded 14
  - 2.1.2 Large Scale Parallel 15
- 2.2 State of High Performance Computing System 16
  - 2.2.1 Shared Memory Systems 17
  - 2.2.2 Distributed Memory Systems 19
- 2.3 Future Directions 21

Chapter 3: Data-Driven Computation with the ParalleX Execution Model **24**
- 3.1 The Execution Model Elements 24
  - 3.1.1 Global Perspective 24
  - 3.1.2 Event-Driven Dynamic Multithreading 26
- 3.2 An Enabling Software Runtime System HPX 29
  - 3.2.1 Implementation of the ParalleX Feature Set 30
  - 3.2.2 Application Programming 32

Chapter 4: Advanced Semantics for Graph Processing **41**
- 4.1 Asynchronous Actions and Continuation Migration 43
- 4.2 Constraint-Based Coordination through Phased Use 48
- 4.3 Graph Traversal Strategies through Embedded Coordination 50

4.4 Large-Scale Data through Distribution . . . 52

Chapter 5: A ParalleX Graph Library . . . 58
5.1 Decentralized Data Structures . . . 58
5.1.1 Construction . . . 59
5.1.2 Initialization . . . 60
5.2 Application Example: Scalable Data Generator . . . 61
5.3 Auxiliary Structures . . . 64
5.3.1 Regions . . . 64
5.3.2 Distributions . . . 65
5.3.3 Containers . . . 66
5.4 Graph Structures . . . 68
5.4.1 Graphs . . . 68
5.4.2 Property Maps . . . 70
5.5 Data-Directed Constructs . . . 71
5.5.1 For-all . . . 71
5.5.2 Map . . . 72
5.5.3 Reduce . . . 72

Chapter 6: Experiments . . . 73
6.1 Large Set Classification . . . 76
6.1.1 Implementation . . . 77
6.1.2 Analysis and Results . . . 80
6.2 Subgraph Extraction . . . 83
6.2.1 Implementation . . . 93
6.2.2 Analysis and Results . . . 96
6.3 Computing Betweenness Centrality . . . 99
6.3.1 Implementation . . . 101
6.3.2 Analysis and Results . . . 104

Chapter 7: Conclusions . . . 111
7.1 An Approach to Graph Processing . . . 111
7.2 Role of the Execution Model . . . 113
7.3 Future Directions . . . 114

Bibliography . . . 116

References . . . 116

Vita . . . 121

List of Tables

2.1 Shared Memory model 17

2.2 Communicating Sequential Processes model 19

3.1 The ParalleX model 24

List of Figures

2.1 The graph concept 11

2.2 Notional graph example 11

2.3 Example semantic network 12

2.4 Parallel traversal of a graph 13

2.5 Shared memory system with four cores per socket 17

2.6 Distributed memory system with four single-core nodes 19

2.7 Dominance of distributed memory computing 22

3.1 System with $n + 1$ locality domains and an AGAS providing global naming . 25

3.2 ParalleX mapped to conventional systems 27

3.3 ParalleX work queue model with two policies 28

3.4 HPX software system architecture 30

3.5 Plain action definition for Fibonacci 32

3.6 Usage of HPX eager futures 34

3.7 Multithreaded execution of Fibonacci code 35

3.8 Multilocality execution of accumulator code 37

3.9 Component server definition for accumulator 38

3.10 Component stub definition for accumulator 39

3.11 Component client definition for accumulator 40
4.1 Execution of the recursive Fibonacci process $Fib(4)$ 44
4.2 Execution of the CPS Factorial process $Fact(3, k)$ 46
4.3 Execution of dataflow-variable example Algorithm 4.5 47
4.4 Comparison of graph kernel action execution 49
4.5 Execution of the parallel traversal of a graph structure 52
4.6 Illustration of the distribution construct 54
4.7 Execution of the distributed processing of an edge list 55
4.8 Execution of the distributed traversal of a graph structure 57
5.1 Decentralized construction process for a distributed structure in the PXGL . 60
5.2 Decentralized initialization process for a distributed structure in the PXGL . 62
5.3 Main thread definition for Scalable Data Generator example 63
5.4 High-level dependencies in the Scalable Data Generator 64
6.1 Graph construction 75
6.2 Illustration of SGAB Kernel 1 75
6.3 Large set classification action definition 78
6.4 Edge filter action definition 79
6.5 Filter edges partition action definition 81
6.6 Illustration of SGAB Kernel 2 on a single locality 82
6.7 Kernel 2 strong scaling with 1 partition 84
6.8 Kernel 2 strong scaling with 2 partitions 84
6.9 Kernel 2 strong scaling with 4 partitions 85

6.10 Kernel 2 strong scaling with 8 partitions 85
6.11 Kernel 2 strong scaling with 16 partitions 86
6.12 Kernel 2 strong scaling with 32 partitions 86
6.13 Kernel 2 strong scaling with 64 partitions 87
6.14 Kernel 2 strong scaling with 128 partitions 87
6.15 Kernel 2 strong scaling for data set scale 19 88
6.16 Kernel 2 strong scaling for data set scale 20 88
6.17 Kernel 2 strong scaling for data set scale 21 89
6.18 Kernel 2 strong scaling for data set scale 22 89
6.19 Kernel 2 strong scaling for data set scale 23 90
6.20 Illustration of data-directed PSearch-enabled execution pattern 92
6.21 Illustration of SGAB Kernel 3 on a single locality 93
6.22 Kernel 3 main thread action definition 94
6.23 Subgraph extraction action definition 94
6.24 Subgraph generation action definition 95
6.25 Kernel 3 total number of ParalleX threads 97
6.26 Kernel 3 ParalleX threads per second 98
6.27 Kernel 3 strong scaling run time 98
6.28 Kernel 4 main thread action definition 102
6.29 Betweenness centrality scoring action definition 103
6.30 Betweenness centrality process begin action definition 105
6.31 Betweenness centrality process expand source action definition 106

6.32 Betweenness centrality process expand target action definition 107

6.33 Betweenness centrality process contract target action definition 108

6.34 Betweenness centrality process contract source action definition 108

6.35 Illustration of SGAB Kernel 4 on a single locality 109

List of Algorithms

4.1 Recursive Fibonacci action $Fib(n)$ 44

4.2 Continuation-passing style Factorial action $Fact(n, k)$ 45

4.3 Continuation-passing style Factorial action $Fact'(n, a, k)$ 45

4.4 Example using CPS Factorial action 45

4.5 Example using dataflow-variable for anonymous producer/consumer execution 46

4.6 Example using **forall** to print multiples of 2 47

4.7 Example implementation of graph kernel action with future-values 48

4.8 Example implementation of graph kernel action with continuation-passing . . 49

4.9 Depth First Search action $DFS(G_C, s)$ 51

6.1 A betweenness centrality algorithm 100

Abstract

Large-scale graph applications are of great national, commercial, and societal importance, with direct use in fields such as counter-intelligence, proteomics, and data mining. Unfortunately, graph-based problems exhibit certain basic characteristics that make them a poor match for conventional computing systems in terms of structure, scale, and semantics. Graph processing kernels emphasize sparse data structures and computations with irregular memory access patterns that destroy the temporal and spatial locality upon which modern processors rely for performance. Furthermore, applications in this area utilize large data sets, and have been shown to be more data intensive than typical floating-point applications, two properties which lead to inefficient utilization of the hierarchical memory system. Current approaches to processing large graph data sets leverage traditional HPC systems and programming models, for shared memory and message-passing computation, and are thus limited in efficiency, scalability, and programmability.

The research presented in this thesis investigates the potential of a new model of execution that is hypothesized as a promising alternative for graph-based applications to conventional practices. A new approach to graph processing is developed and presented in this thesis. The application of the experimental ParalleX execution model to graph processing balances continuation-migration style fine-grain concurrency with constraint-based synchronization. A collection of parallel graph application kernels provide experiment control drivers for analysis and evaluation of this innovative strategy. Finally, an experimental software library for scalable graph processing, the ParalleX Graph Library, is defined using the HPX runtime system, providing an implementation of the key concepts and a framework for development of ParalleX-based graph applications.

Chapter 1

Introduction

Computation has been well established as a necessary component of scientific research in the 21st century [1]. Theory and physical experimentation are enhanced by the ability to simulate increasingly complex systems at greater modeling resolutions and with higher fidelity. Recently, projects such as the Sloan Digital Sky Survey [2] and the Human Genome Project [3, 4], and data center proliferation for Web search engines, by Google, Microsoft, and others [5, 6], serve as evidence for growing importance of large-scale data. Thus, data-intensive applications — defined as those focused on the analysis of large data sets — have emerged as a "fourth pillar" of science, necessary to manage the deluge of data and derive information and understanding from it [7, 8].

In particular, so called graph-based applications are emerging as important. Traditional science and engineering applications, out of necessity, are becoming more dynamic to deal with higher resolution. Prime examples of this are Adaptive Mesh Refinement (AMR) for Astrophysics and molecular dynamics [9, 10, 11] and high-fidelity, coupled oceanic and atmospheric simulations, computational fluid dynamics, and other numeric applications [12]. In the case of AMR, dynamic tree structures of computational spaces adapt to provide variable levels of resolution across an evolving simulated space.

Another important class of graph-based applications is those dealing primarily with information or knowledge-processing problems. Examples of these include ontology-based semantic analysis, social network analysis, data mining methods, natural language processing, and AI pattern matching. In the case of ontology-based semantic analysis, queries are posed in the form of a semantic network that is matched against a knowledge base (a dynamic semantic network). The ontology provides valid transformations for deriving additional in-

formation from both the query and the knowledge base, and is used to dynamically update both networks, expanding and contracting semantic relationships, as the process evolves.

Many problems arise when considering the characteristics of graph applications in the context of high-performance computing. Graph-based applications are defined with concurrent activities in the form of simultaneous queries, at the highest level, and independent branching of subproblems, at the low level. During execution, the amount of concurrent actions (the instantaneous workload) will vary dependent on the structure of the graph and the stage of the algorithm. Also, with the processing of graph structures, the flow control of the execution is dictated by the structure of the graph data set (i.e., it is data-directed) and limited in available computation, with the basic operations defined in terms of, and at the scale of, the structural relationships between vertices and edges.

Irregular data access patterns lead to inefficient use of the memory hierarchy and program control flow, and increased latencies. The lack of work relative to communication load exacerbates the effects of the memory wall (the disparity between processor and memory speeds). Data-directed execution over an irregular data structure produces time-varying workloads at runtime that the programmer is unable to predict or manage using static threading and data-parallel programming constructs. And, high levels of exploitable concurrency over a shared data structure increase the performance-limiting effects of contention.

Performance is determined by the ability to make effective use of the memory system by exploiting available parallelism and managing synchronization of concurrent activities, and above all, minimizing and mitigating the effects of latencies and overhead. These application characteristics challenge approaches on conventional high-performance computing systems that either rely on point-to-point message-passing between compute nodes, limiting efficiency, or rely on a single shared memory, limiting scalability. With the ever expanding sizes of data sets, the critical challenge is to improve scalability of problem size and efficiency in resource utilization through the use of alternative techniques for extracting and managing parallelism to improve performance.

1.1 Research Objective

1.1.1 Goal

This research establishes a new approach to scalable, highly concurrent, distributed graph processing on high-performance computing systems. The core data structures and programming constructs provided by the new graph library, PXGL, establish a framework for developing specific algorithms across all graph-based application domains. Researchers in proteomics will have a tool for studying the complex interactions of protein structures, making possible breakthroughs in the treatment of cancer and other systemic illnesses not addressed with modern medical practices. And analysts and scientists in fields from social networks for productization and market research, and ecology and large-scale food webs for environmental conservation will be able to model and analyze interactions at greater scales and resolutions for increased understanding. The co-design of algorithms, data structure, and application, with the ParalleX model of computation means that the work is positioned to lead graph algorithm design and application development to extreme-scale system architectures through the peta-flops regime and into the exa-scale performance domain over the coming decades.

1.1.2 Hypothesis

Graph-based applications require distributed memory systems to achieve scalability. Computations require dynamic medium- to fine-grain multithreading to achieve efficient utilization of system resources. And concurrent processing of shared mutable data structures requires localized, embedded coordination mechanisms.

Current and anticipated application scales suggest that a shared memory approach cannot provide the level of scaling necessary for this problem class. The necessary component that must be carried over from the shared memory model is the notion of a global address space. This must be supported for both locating and referencing individual elements of compound distributed data structures (e.g., the vertices of a graph) from anywhere in the system. An explicit notion of locality, providing the ability to reason about whether data is colocated

with an action, is necessary. Finally, an explicit concept of data distribution, providing the logical mapping of global structure to system topology, must be a fundamental component of the any approach.

Visiting the neighbors of a vertex is a common operation that can yield a number of concurrent actions on many independent paths. Thus, the structural meta-data of the graph data set dictates both the next set of tasks, and what activities can be done in parallel. The guiding principle for graph algorithms, kernels, and applications utilizing fine-grain data-directed threading over graph data structures must be to support continuously moving the computation forward. Actions must carry with them the means of continuing execution dependent on the data structure, producing follow-on actions that can independently progress the computation. At finer-granularities, this will be expressed in a continuation-passing style of computation. But at higher levels, data-directed techniques, such as parallel-forall and map constructs express the pattern of computation, while leaving the particulars of dynamic execution to an active runtime system adapting to the particular data structure.

Simultaneous actions accessing the same data create the possibility of race conditions and increase the likelihood of contention to degrade performance. An example of this is two separate search paths converging on the same vertex at the same time. In order to keep the data structure and the processes in a coherent state, access to the data item must be coordinated, and this must be addressed at two levels. Distributed data structures must provide the interface and mechanisms for efficient, locality-aware, processing by algorithms, and each member of such a structure must be able to maximize local information to work in a decentralized manner whenever possible to reduce the effects of latency and overheads. Likewise, when locally managing a collection of concurrent actions, the use of global barriers to enforce some notion of coherence across the application must be replaced with localized coordination mechanisms embedded in the data structure that support independent progress at finer granularities.

1.1.3 Objectives

This dissertation studies the use of a novel execution model to enable high-performance graph processing on current large-scale compute systems through consideration of the trade-offs in latency, overhead, contention, and starvation necessary to achieve scalability. The ParalleX model [13, 14] (Chapter 3) represents a break from the HPC conventions of message-passing computation on distributed memory systems, and static multithreaded computation on shared memory systems. This work builds upon the execution model, effectively integrating into the co-design of software runtime and operating systems for conventional architectures, and future extreme-scale systems. A path towards addressing current needs for scaling-challenged problems is provided, beginning to answer the challenges posed by peta- and exa-scale development platforms. In particular, this work defines semantic constructs for graph processing independent of any particular language or library. Instead of matching standard practice or an incremental development approach, the programming techniques match the problem, leaving their exact implementation open; though one experimental software system is defined as a proof of concept and vehicle for exploration. Mechanisms for managing concurrency and synchronization of fine-grain, data-directed computation in a message-driven, dynamic multithreading, distributed memory system are designed to extract and manage parallelism for scalable execution. Specifically, the following objectives are outlined:

1. Move work to data and utilize local work queues with message-driven multithreading;
2. Manage locality for highly dynamic and distributed applications with a global address space;
3. Replace global barriers with localized synchronization primitives embedded into the graph structure;

4. Address contention on data and network with locality-aware algorithms combined with a work-moving execution strategy; and,
5. Define constructs around graph meta-data that allow a programmer to express how the computation should be parallelized with respect to the graph structure.

1.2 Technical Strategy

The technical strategy of exploring the application of a novel execution model for large-scale graph processing comprises:

1. Introduction of advanced semantic constructs for graph processing;
2. Mapping of those semantics into the message-driven, multithreaded ParalleX model;
3. Development of an experimental graph library for the ParalleX model;
4. Implementation of exemplar application kernels in the ParalleX Graph Library; and,
5. Experimentation and analysis of initial performance and scalability.

Reconsideration of the semantics of graph processing addresses the challenges related to parallelism and efficiency through the design of new graph data structures and parallel constructs. There are two dimensions to the problem of developing constructs for graph processing. The first deals with semantic constructs for expressing and controlling parallelism, from a programmatic standpoint. We address this by identifying and employing high-level techniques for programming near fine-grain event-driven threads. An example is the adaptation of data-directed techniques for traversal of graphs. The second dimension is concerned with the data structures that are used to represent the graph concept. Extensions will be made to support fine-grain event-driven multithreading in a distributed memory, global address space system, with lightweight synchronization primitives. Examples include the embedding

of synchronization primitives into the graph data structure itself to localize contention and to facilitate constraint-based coordination of concurrent access and update.

Mapping of the new semantic constructs for graph processing into the ParalleX execution model defines the foundation for efficient parallel execution. Dynamic event-driven multi-threading and lightweight synchronization primitives in a global address space model support the data-directed graph processing. This also enables the development of the ParalleX Graph Library. This software implementation provides a framework for the development of graph data structures and algorithms and that will support future research and development of graph-based applications utilizing the ParalleX model. The PXGL will be implemented using the high-performance software runtime system HPX.

A proof-of-concept for the PXGL as a programming framework for graph processing is a set of graph-based problem benchmarks which may be used further for experimentation and analysis. The benchmark used here is based on the SSCA#2 Graph Analysis Suite that was developed as part of the DARPA HPCS project [15, 16]. The four kernels stress important aspects of large scale graph processing. Finally, testing and evaluation of the graph constructs, implementation, and kernels will involve experimentation to establish initial results on conventional systems.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows: Chapter 2 provides an overview of scalable graph processing in high-performance computing, outlining current approaches and the key challenges. Chapter 3 defines the ParalleX model of computation, and the software runtime system, HPX. Chapter 4 presents the new semantic constructs for concurrent processing of distributed graphs. Chapter 5 defines the ParalleX Graph Library (PXGL), a software library for coding graph applications in the ParalleX model. Chapter 6 presents a set of important graph kernels, the experimentations performed, and an analysis of the

results. Finally, Chapter 7 concludes by distilling the key contributions and impact of this work, and identifying directions for future efforts in this area.

Chapter 2

Background

The emerging area of *data-intensive* applications, specifically graph processing, is gaining in importance, and establishing new requirements for high-performance computing (HPC) systems. Conventional HPC systems and applications have been focused on exploiting static parallelism for processing regular (easily partitionable) data sets for scalable execution on commodity-based systems. However, the recent proliferation of multicore and GPU technologies will not sustain the continued performance gains seen over the past 20 years. With this intersection of changing application requirements and HPC system capability comes a new set of problems and challenges for achieving high performance, both on available conventional systems, and future extreme-scale systems. This chapter introduces the increasingly important area of graph-based applications (Section 2.1), and the conventional high-performance computing (Section 2.2) that supports them, to establish the necessary context for the work undertaken.

2.1 Graph-Based Application Processing

Graph-based applications are emerging as important for STEM[1] and symbolic informatics problems, possibly replacing regular matrix computations as the dominant form of computing in the next decade. Traditional science and engineering applications, out of necessity, are becoming more dynamic to deal with higher resolution. Prime examples of this are Adaptive Mesh Refinement (AMR) for Astrophysics and molecular dynamics [9, 10, 11] and high-fidelity, coupled oceanic and atmospheric simulations [12], computational fluid dynamics, and

[1] Science, Technology, Engineering, and Math.

other numeric applications. In the case of AMR, dynamic tree structures of computational spaces adapt to provide variable levels of resolution across an evolving simulated space.

Another important class of graph-based applications is those dealing primarily with information or knowledge-processing problems. Examples of these include ontology-based semantic analysis, social network analysis, data mining methods, natural language processing, and AI pattern matching. In the case of ontology-based semantic analysis, queries are posed in the form of a semantic network that is matched against a knowledge base (a dynamic semantic network). The ontology provides valid transformations for deriving additional information from both the query and the knowledge base, and is used to dynamically update both networks, expanding and contracting semantic relationships, as the process evolves.

Semantic networks are the base model for such applications, which are defined as those focusing on the processing of data sets which originate from information sources [17]. A typical underlying representation, and the one which is the focus of this research, is the attributed relational graph. A graph is a general abstraction for representing entities and relationships between entities. The entities are represented by vertices, and the relationships by pairs of vertices, called edges. In the field of *Graph Theory*, a graph is formally defined as an ordered pair (V, E), where $V = \{u, v, w, \ldots\}$ is the set of vertices and $E = \{(u, v) : u, v \in V\}$ is the set of edges. For each edge $(u, v) \in E$, we refer to the u as the *source vertex* and v as the *target vertex*. A graph can be either *undirected* or *directed*, indicating whether the edge represents a symmetric relationship or not. Figure 2.1 shows three example directed graphs, a list (G_1), a tree (G_2), and a general graph (G_3), along with their mathematical definitions. This graph concept is sufficient for constructing arbitrarily complex graph structures, but to make it useful for real-world problems, the definition is extended to include attributes on the vertices and edges. An attributed graph is an ordered quadruple (V, E, t, w), with V and E as before, $t : V \to L$ a functional mapping from vertices to vertex labels, and $w : E \to T$ a functional mapping from edges to edge labels. Figure 2.2 shows a simple attributed graph, a possible representation as an adjacency list, and the possible structure of a vertex. Semantic

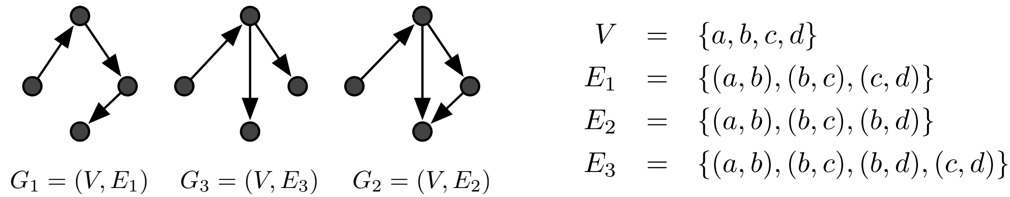


Figure 2.1. The graph concept, graphical and mathematical notations.

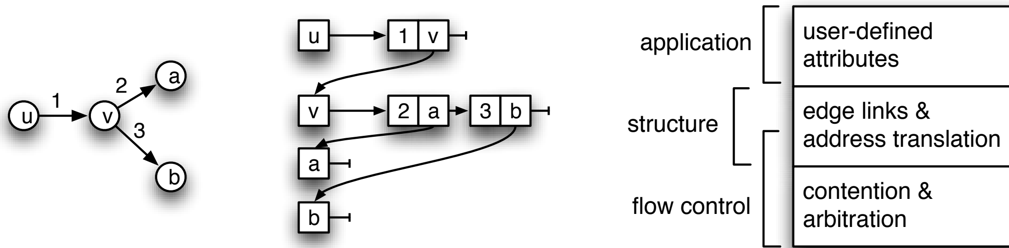


Figure 2.2. Notional examples of an attributed graph, it's adjacency list representation, and vertex structure.

networks are directed attributed graphs that encode knowledge, or information, by defining a set of typed entities and a set of typed relationships between them.

A semantic network, such as the one in Figure 2.3, can be modeled using an attributed graph, where the vertices are people, places, and things, and the edges denote particular relationships between them. From the example diagram, the top left subgraph, labeled "Pattern", would have vertex and edge label sets:

$$L = \{\text{Factory}, \text{Person}, \text{House}, \text{Truck}, \text{Fertilizer}\}, \text{and}$$

$$T = \{\text{observe}, \text{reside}, \text{rent}, \text{buy}\}.$$

Graph-based applications for semantic network analysis all share similar structure and characteristics. These applications are typically composed from kernels which provide traversal/search, component-finding, and metric generation. Graph processing consists of fine-grain actions and irregular memory accesses. These problems are notoriously difficult to optimize because of their strong run-time dependencies. Computation is typically limited to testing some predicate condition, and possibly updating a value, local to a single vertex. Thus, most

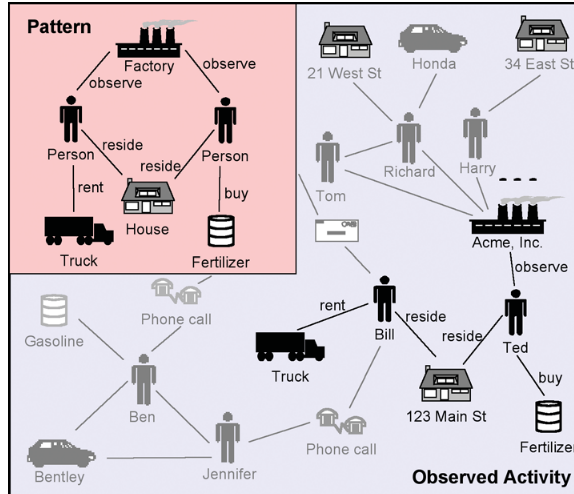


Figure 2.3. Example semantic network modeling the interconnections between a set of people, places, and activities. *Source: Graph-based Technologies for Intelligence Analysis, Coffman et. al.* [18].

of the time in a graph kernel is spent accessing memory as it traverses from one vertex to another, following the graph structure.

An example of a general graph kernel is a parallel traversal, where the computation proceeds recursively by visiting a vertex, testing some predicate, and then visiting all the adjacent vertices. Figure 2.4 illustrates a parallel traversal of an undirected graph. The graph is on the left, with distinguished start vertex s. The paths taken through the graph are depicted by the tree in the middle. The spawning of "visit vertex" operations is depicted by the tree on the right: dark circles denote when the operation was the first to visit a particular vertex, in which case they spawned additional visits for each adjacent vertex; the light circles are operations that visited a vertex after it had already been seen, in which case they did not continue the traversal.

These graph-based applications constitute an important subfield of high-performance computing, distinct from the more traditional simulations of physical phenomenon [17]. In those cases, where the data sets are typically dense and/or well-structured, matrices and matrix computations dominate the solution space. Conventional architectures and programming models are particularly well suited for the physics applications which exhibit good temporal

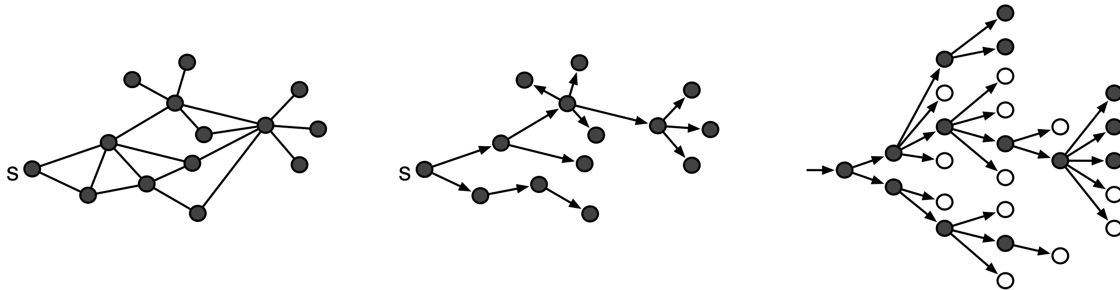


Figure 2.4. Parallel traversal of a graph from a start vertex, s, to all other vertices: the undirected graph structure (left), the directed paths through the graph (center), and the tree of visit actions (right).

and spatial locality and greater amounts of computation versus communication. The sparse and irregular structure of the input data set meta-data, however, reduces the degree of spatial locality. The heavy reliance on structural meta-data to direct the computation destroys useful temporal locality. And the typical graph application is dominated by search or traversal, doing very little computation, relative to the amount of memory accesses (communication). Furthermore, graph applications have been shown to be more *data intensive*, distinct from the more traditional matrix-based simulations [7]; i.e., the number of memory accesses of unique data is very high with respect to the number of numeric operations performed. For these reasons, graph processing needs an HPC system capable of supporting intense, irregular access to the memory system, with limited computation to cover access latencies.

Robust support for large scale graph processing applications requires software libraries specifically engineered for HPC systems. Two such solutions exist: the Parallel Boost Graph Library (PBGL) [19, 20] and the Multi-Threaded Graph Library (MTGL) [21, 22]. The PBGL is an extension of the sequential Boost Graph Library (BGL) [23] for distributed memory clusters. The Multi-Threaded Graph Library targets so-called massively-multithreaded shared memory systems, while attempting to stay compatible with the BGL library interface. Sections 2.1.1 and 2.1.2 introduce these two systems and their support for graph processing. Together these libraries establish the current and future efforts in this area.

2.1.1 Multithreaded

The Multi-Threaded Graph Library was designed and developed at Sandia National Laboratories to support research in high-performance graph processing. Their approach was to deliver a graph library that presents the familiar BGL interface, but is tailored for massively-multithreaded architectures, such as the Cray MTA/XMT (Section 2.2.1). As these systems are specifically designed for addressing the requirements of processing large, sparse, irregular data sets, the MTGL is particularly well suited for high-performance graph processing.

Recent work has been done to port the MTGL to commodity systems with new multi/many-core processors, such as Sun's Niagara T2 and Tilera's TilePro64 [22]. This involved porting the code base to use Qthreads [24], a low-level library for programming with lightweight threads. While the MTGL/Qthreads implementation does not achieve the same performance as with the Cray XMT, it does provide a means for developing and testing of graph applications on commodity systems before moving them to more expensive resources.

The key to achieving performance on this class of system is to always have enough work to keep the processors and memory channels busy and to minimize the opportunity for contention between concurrent tasks. The XMT's parallelizing compiler is the principle means of extracting parallelism from native MTGL codes. In the Qthreads version, the parallelizing compiler is replaced by a combination of handwritten threads and parallel loop constructs provided by Qthreads.

Ultimately, scalability is limited by the amount of shared memory that can be supported by a single system. Though, given the scaling challenges for distributed memory machines, and the ever increasing aggregate main memory capacity in shared memory systems, there is limited discussion of scaling beyond shared memory What is needed is a solution that extends the lessons learned from MMT to distributed memory systems.

2.1.2 Large Scale Parallel

The Parallel Boost Graph Library [20], developed in the Open Systems Laboratory at Indiana University, is an extension of the Boost Graph Library, a popular graph library for sequential graph processing. The PBGL is targeted at distributed memory message-passing supercomputers, using the Message Passing Interface (MPI) [25] as the parallel abstraction and communication layer. This approach has the immediate advantage of leveraging the ubiquity of commodity clusters and MPPs across academic departments and research organizations in government laboratories and industry, as well as the wide availability of efficient MPI implementations and the portability of MPI code bases.

However, there are recognized drawbacks which limit scalability and performance of graph processing with commodity cluster computing [17]. First, the MPI programming model favors static allocation of long-running processes: typically each processor runs one process for the entire duration of the application. Therefore, this approach does not lend itself to the exploitation of fine-grain parallelism inherent in graph processing. Also, dynamic load balancing of time-varying work is difficult to implement over static processes.

Partitioning the graph data structure over a distributed memory is cumbersome. Partitioning the graph means that there will be some edges where the source vertex belongs to one system node, and the target to another. The typical approach in such an environment is to use *ghost cells* to cache information about the target local to the source. However, the irregular structure of graph data means that there is little opportunity for finding the optimal partitions that minimize the number of cross-process edges, in order to minimize the amount of cached data and messages over the network.

A particular issue with power-law graphs [26, 27] is the existence of extremely high degree vertices. Following the ghost cell strategy, a vertex with a high-degree vertex could easily require more local memory for caching distant vertices than for its own local vertices. As a result, it has been found that distributed memory machines cannot support graphs as large as those on significantly smaller shared memory systems, which do not require such

ghost cell schemes. Though it should be noted that ghost cells are not a requirement, and large-scale graph applications have been developed using other techniques, such as increasing messaging to transfer data or hashing the mapping of vertices to nodes [28, 29]. But these approaches still limit scalability by increasing the message buffers and necessary bandwidth, and significantly increasing the computational overheads.

Finally, the key to achieving higher performance with the PBGL is to keep as much work local as possible and to make effective use of spatial and temporal locality in the local data. But this ultimately limits scalability by forcing an ill-suited programming model that does not address the requirements of the application area, but, instead, the requirements of the conventional computing platform. Ultimately, from a performance standpoint, the PBGL demonstrates that the CSP/BSP model, and the use of MPI, over distributed memory systems is a poor match for graph applications. Yet, it does show the benefits of a generic programming for designing a robust and flexible library, enabling productive programmers.

2.2 State of High Performance Computing System

Conventional high-performance computing systems are those that serve the most extensive sets of applications with the highest level of capability. Two classes of HPC systems establish the convention in HPC — a broad, and always evolving, field with the introduction (and, in some cases, reintroduction) of new technologies and strategies such as custom massively parallel processors (MPPs), manycore processing elements, and graphics processing unit (GPU) accelerators. The first class, comprising shared memory multithreaded systems, is the most widely used for highly threaded computing. The second class, distributed memory commodity clusters [30, 31, 32], is the most widely used platform for large-scale parallel computing. Sections 2.2.1 and 2.2.2 introduce these two classes of systems.

2.2.1 Shared Memory Systems

The general system architecture of any shared memory system is a single main memory shared by some number of processing units. Logically, each processing unit is able to access any address in memory. Figure 2.5 illustrates a possible scenario for a uniform memory access (UMA) shared memory system with 16 cores: each core has a private L1 and L2 cache, and sets of four cores share an L3 cache on a socket. This is the basic architecture of both modern high-end symmetric multiprocessors (SMPs) and multicore processor workstations. So called non-uniform memory access (NUMA) shared memory systems provide an alternative system architecture that preserves the shared access to all of main memory, but incorporates a notion of proximity between a core and a given region of memory. For these systems, the specifics of the system topology can dictate different performance penalties for access to different regions of the address space.

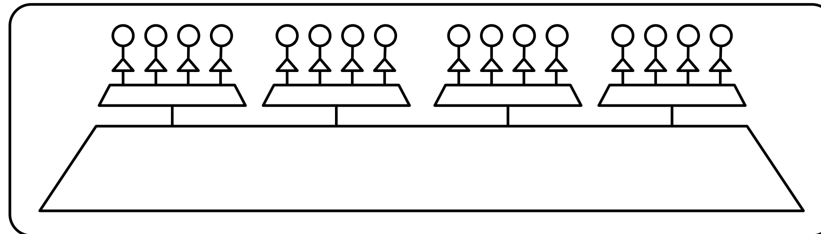


Figure 2.5. Shared memory system with four cores per socket. The memory hierarchy shows each core with private L1 and L2 cache, and each socket with its own L3 cache.

Table 2.1. Shared Memory model.

	Cache-coherent shared memory	**MMT**
Concurrency	Medium-grained static threading	Fine-grained dynamic threading
Coordination	Shared state, locks, critical sections	Shared state, mutex, FEB, etc.
Movement	—	—
Naming	hardware-supported GAS	hardware-supported GAS

The dominant performance bottlenecks of these systems come from contention for shared resources, associated penalties in access latency, and the overheads associated with the managing of threads. One approach is the use of static allocation of a fixed number of threads in an application, combined with a programming model that supports simple data-parallel fork-

join semantics for partitioning the work over disjoint regions of the memory. This reduces overheads in thread creation and destruction, and minimizes contention by providing guarantees of non-overlapping memory accesses. This style of shared memory processing has been referred to as *cache-coherent* shared memory [29], exemplified by the OpenMP API [33, 34]. The primary limitations come in the form of starvation, as it is difficult to generate sufficient parallelism to keep all processing units actively engaged [29], and the inability to mitigate the effects of latencies associated with blocking threads.

Another approach is to use dynamic allocation and management of threads using a work queue model, combined with a programming model that supports spawning concurrent tasks independent of the number of processing units. This provides the opportunity for increased levels of parallelism to keep the processing units engaged, and latency mitigation through overlapping memory access with computation. The increased overheads imposed by the management of dynamic threads are reduced by specialized hardware support. This style of processing has been referred to as *massively multithreaded* (MMT) shared memory [29]. Examples include the high-end Cray MTA/XMT [35, 21] architectures, as well as the Sun Niagara processor for workstations. Though not as well formalized as the CSP and BSP models for distributed memory processing, these two approaches are seen as the dominant execution models for shared memory processing, and their properties are summarized in Table 2.1.

Application development for both classes of systems is at first aided by the shared memory view, which removes the need for explicit data partitioning. However, the major hurdle becomes properly synchronizing concurrent operations to prevent race conditions, and subsequently managing performance degradation incurred from managing contention on shared resources. A race condition is present when an execution depends on the ordering of simultaneous operations by two or more parallel threads. If not properly managed with atomic operations, locks, or critical sections, the resulting execution can be left in an inconsistent state. Controlling shared access to simple data values is typically handled by the use of atomic operations. Critical sections are imposed for managing execution of sequences of op-

erations that can only be executed by one thread at a time. Furthermore, the misuse of these synchronization constructs can have the effect of raising contention, and even serializing apparently parallel codes. All of this increases the burden on the programmer to manage the sharing of resources in multithreaded applications both for insuring correct execution and performance, which is notoriously difficult.

2.2.2 Distributed Memory Systems

The general system architecture of any commodity cluster is a collection of compute nodes, linked by a dedicated network interconnect, as illustrated in Figure 2.6. Originally, compute nodes presented little or no exploitable internal parallelism, favoring coarse-grain node-level parallelism, with message-passing between nodes for coordination and data exchange: i.e., the Communicating Sequential Processes (CSP) model of computation[2] [36, 37]. This model supports general distributed computing, but for HPC, the focus on efficiency means the reduction/management of latency and overheads.

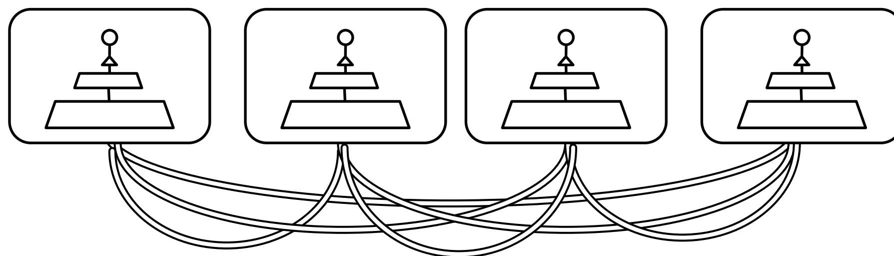


Figure 2.6. Distributed memory system with four single-core nodes with interconnect.

Table 2.2. Communicating Sequential Processes model.

Concurrency	Coarse-grained node-level parallelism
Coordination	Message-passing
Movement	Message-passing for data movement
Naming	Disjoint local address spaces

The network interconnect provides the dominant performance bottleneck in terms of (relatively) high communication latency and overhead. This favors an approach which minimizes

[2]See Table 2.2 for a synopsis of the CSP model.

communication and maximizes message payload (within some additional constraints, such as messaging overheads and bisection bandwidth) through coarsening of work to reduce the number of messages. Likewise, the dominant overheads would be associated with the management of sharing state (i.e., coordination and cooperation) between nodes. This favors an approach where each participant (node) acts independently from the others, and synchronization is kept to a minimum. These requirements are captured in the Bulk Synchronous Parallel (BSP) [38] model, which became the primary strategy for expressing parallelism in applications on large-scale commodity clusters.

A clean mapping between the model of computation and the application area led to the success of commodity cluster computing for the last two decades. Traditional simulations of physical phenomena [17] have large data sets that are well-structured. These exhibit good temporal and spatial locality and greater amounts of computation versus communication. Furthermore, the structure of the data set was not likely to change, so static partitioning into equal sizes for load balancing could be done by the programmer to evenly distribute the workload across the system; with only a relatively small amount of data needing to be shared between time steps, for realistic size runs, resulting in scalable use of memory and message payload sizes. All of this allowed applications such as physics simulations to take advantage of the CSP/BSP model and scale to previously challenging problem sizes.

The success of CSP/BSP with distributed memory systems has meant the widespread deployment of commodity clusters and the development of mature but modest software runtime systems and libraries supporting a wide range of applications. In particular, the Message-Passing Interface (MPI) specification [39] has become the dominant way to program these systems for STEM[3] science problems. So much so, that many equate distributed memory applications using MPI with the CSP/BSP model. Though it is not necessary to design an MPI application following the BSP model, it must be acknowledged that this is by far the

[3]Science, Technology, Engineering, and Math.

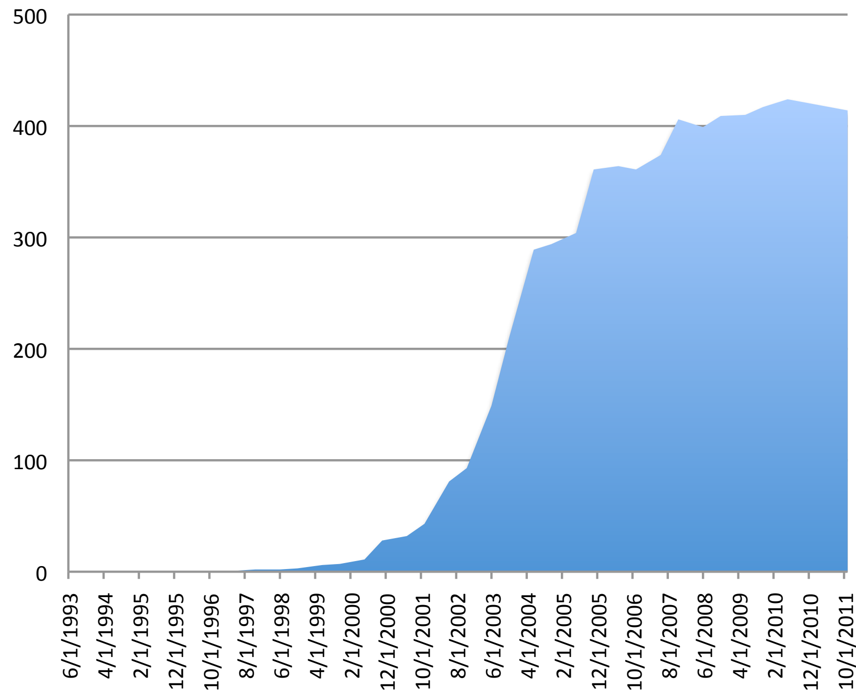
dominant trend. Indeed, the advanced features of MPI-2 receive little attention, outside of the I/O package.

2.3 Future Directions

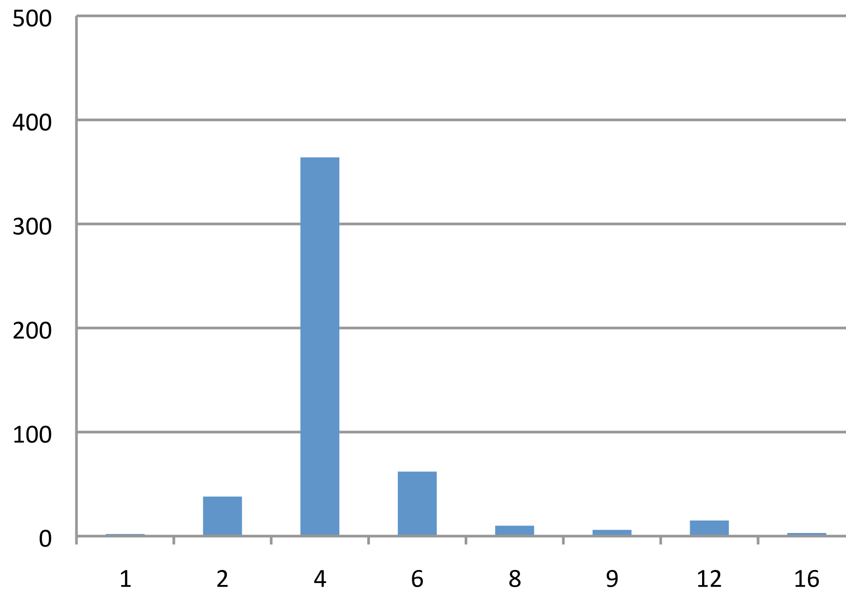
The principal leads of the Parallel Boost Graph Library are actively working to support the use of active messages over the point-to-point message-passing and bulk data movement provided by MPI. Their target is the AM++ communication library for general distributed memory programming [40]. Though there are other active message [41] based software systems available, such as GASNet and Charm++ [42, 43], the developers of AM++ viewed the available options as either too low-level or too high-level, meaning the end user either received too little or too much support — with both cases hindering productivity. Initial results by Lumsdaine, Edmonds, and Willcock for graph processing with AM++ [40] are promising. Yet, further work is needed to see how the PBGL is adapted from a model focusing on coarse-grained static parallelism and bulk message transfer, to a model where the asynchronous work-moving communication primitives are best served by medium- to fine-grained dynamic threading.

The proliferation of multicore and accelerator technologies [44] provide new opportunities and challenges for distributed memory programming. Figures 2.7a and 2.7b show the dominance of distributed memory computing, with both commodity cluster and MPP systems, and the degree of exploitable intra-node parallelism in these systems, respectively, over the previous two decades of Top500 supercomputers [45].

There are two current approaches to leveraging the potential of intra-node parallelism: mapping CSP processors to cores and taking a hybrid model approach. While semantically clean, the former must account for the introduction of heterogeneous latencies between CSP processors which will exacerbate the penalties associated with the use of global barriers — the longest running CSP processors will always determine the runtime of a BSP fork-join



(a) Proportion of Top500 systems over previous three decades that were distributed memory commodity clusters or MPPs.



(b) Distribution of cores per node of all distributed memory systems on the November 2010 Top500 list.

Figure 2.7. Dominance of distributed memory computing over the past two decades and the rise of intra-node parallelism.

style application, and those processors will still be the ones that must incur the inter-node latencies.

Use of hybrid approaches and/or ad hoc solutions are typically limited to only specialized single use codes, and of these, most are demonstrations or proof-of-concept codes, not for general use. As example, one hybrid approach is to use CSP/BSP, specifically MPI, to orchestrate inter-node processing, and cache-coherent shared memory, specifically OpenMP, to manage intra-node multithreading. Again, this suffers from managing the relatively high latency inter-node communication. Though there is potential benefit from using a multithreading model locally, success has been limited on conventional applications [46, 47], and the combination of models is not expected to be readily programmable as both number of nodes and intra-node parallelism increase in the future [8]. Further complicating the issue is code portability, which is limited by approaches that attempt incremental extensions to only a portion of the computing stack.

Recently, efforts have begun to reconsider the execution model to directly address the computation requirements and goals in terms of performance and efficiency, not adherence to legacy codes or conventional practice [8]. It is considered that execution model co-design will offer the best chance for future success, and that future efforts for extending graph libraries, such as the PBGL and MTGL will eventually align with a future convergent model of computation for scalable parallel computing. The ParalleX model discussed in Chapter 3 is proposed as one such approach, and is the model with which this work is aligned.

Chapter 3

Data-Driven Computation with the ParalleX Execution Model

A parallel execution model provides a conceptual framework for considering design decisions across the computing stack: programming languages, compilers, runtime, operating system, as well as system architecture and hardware technology. The semantics of the execution model allows for the co-design of the different layers in the stack by defining the core concepts that are fundamental to computation and invariant across any implementation. The experimental ParalleX execution model [13, 14] represents a break from the HPC conventions of message-passing computation on distributed memory systems, and static multithreaded computation on shared memory systems. Instead, it defines an approach for enabling massively multi-threaded, message-driven computation on current high-end machines and future extreme-scale systems. This chapter describes the major semantic elements of ParalleX (summarized in Table 3.1), and the use of policies for managing aspects of the model that vary across systems and applications.

Table 3.1. The ParalleX model.

Concurrency	Dynamic event-driven multithreading
Coordination	Lightweight, localized synchronization primitives
Movement	Asynchronous one-sided messages for work migration
Naming	Hierarchical processes and active global address spaces
Introspection	First-class entities for run-time adaptivity

3.1 The Execution Model Elements

3.1.1 Global Perspective

An Active Global Address Space (AGAS) is defined as an extension to the Partitioned Global Address Space (PGAS) model [48, 49] for naming and referencing objects across the

system. The AGAS provides the convenience of a shared memory model and the scalability of a distributed memory programming model. In ParalleX, processes, threads, and data are all named objects in this global namespace. The *active* element of the AGAS refers to the support for migration of data across disjoint memory spaces without renaming. This is important because it allows for an object to move from one physical domain to another without invalidating any references to it that might be held by other data objects or processes.

The computational space is partitioned into a set of locality domains mapped onto a physical system. For each locality, the response time for action execution is bounded, and compound atomic operations on local state are guaranteed, providing an upper bound on the extent of a locality on a given system. The notion of locality aids the programmer, or dynamic runtime heuristic, in reasoning about the relative costs of local and remote actions. Figure 3.1 illustrates the relationship between locality domains and AGAS, providing a ParalleX system view to the application. Between the localities is an asynchronous communication domain through which messages pass from one locality to another. These messages, which are called *parcels*, communicate actions to be performed from one locality to another.

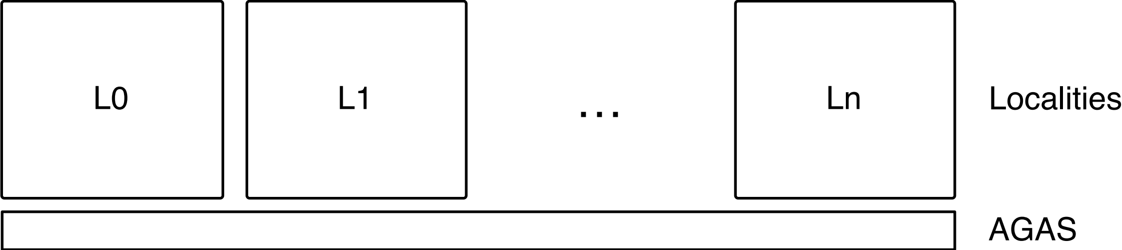


Figure 3.1. System with $n+1$ locality domains and an AGAS providing global naming.

ParalleX processes are a departure from the dominant practice in HPC, enterprise server, and low-end computing. Most models reflect the notion of a process directly from their low-level system operating environment, e.g., the concept of a process in a multi-processing OS. This practice provides for one or more large, heavy-weight processes per physical device (a node in a commodity cluster). In contrast to this, ParalleX defines the concept of parallel

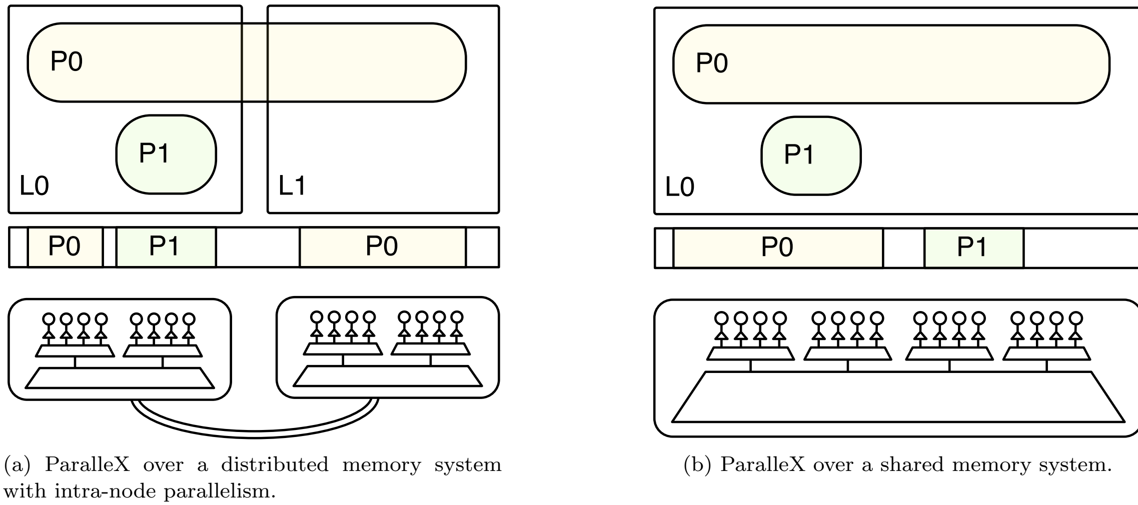
processes that can span multiple physical domains (logically identified as localities). The process provides the context for all computation and data in an execution. They also serve as the front line for resource management, by interfacing the active elements of the computation with the physical system in which it is executing.

Each ParalleX application starts a main process that serves as the initial context for data and execution for the application. During the lifetime of the process, other subprocesses can be instantiated in parallel to further encapsulate data and execution. The process hierarchy provides protection for data and other named objects in the AGAS by defining the visible scope of names across the system. Also, these processes are able to span multiple localities, which is significantly different from other models, where processes are limited to a single physical domain.

Policies deal with those specific aspects of a parallel computing system that vary across application demands, algorithmic complexity, system architecture, and device technology. A policy defines the operational characteristics of a specific aspect of the system for a specific execution of an application. Policies can be asserted at both compile- and run-time. A particular strategy for the run-time load balancing of data is an example of a policy: the choice of strategy will depend on the choice of underlying system and the characteristics of the chosen application. Other examples include thread scheduling strategies, consistency models for distributed shared data, and run-time load balancing of time-varying data sets.

3.1.2 Event-Driven Dynamic Multithreading

Messaging in the ParalleX model takes the form of parcels. Parcels enable the movement of work (continuation of execution) from one locality to another, akin to an active message [41]. A parcel carries an action to perform, the name of a target object upon which to apply the action, the necessary parameters for the action, and possibly a continuation to be performed after the action completes. When an action is applied to remote data, a parcel is created which communicates the action to the locality where the target data resides. Upon arrival, a parcel



(a) ParalleX over a distributed memory system with intra-node parallelism.

(b) ParalleX over a shared memory system.

Figure 3.2. ParalleX mapped to conventional systems. The top-most block(s) depict localities mapped to nodes, and the lower block the AGAS spanning all localities. The colored regions labeled P_0 and P_1 depict processes.

is converted into the appropriate action to be performed local to the target data. The sending of continuations with actions allows for migrating the flow of continued execution control avoiding round-trip messages when unnecessary. Data can also be sent in bulk transfers via parcels; typically modeled as remote *gets* and *puts*.

Locally, actions are managed using a work-queue model. When a new action is initiated from an active thread or the arrival of a parcel, a new thread is created. If all preconditions for the thread's execution are met, then it is considered *pending*, and enqueued in the work queue awaiting the allocation of a compute resource. Otherwise, the thread is considered to be *suspended* and buffered, awaiting the satisfaction of some conditions before it is made pending. Figure 3.3 depicts the flow of tasks through such a work queue model. Note that ParalleX threads are expected to be short-lived, in comparison with other models that favor statically allocated, long-running threads that can last for the lifetime of an application (e.g., OpenMP and UPC [33, 50]).

The realization of execution in the ParalleX model takes the form of a split-phase transaction. A split-phase transaction is an execution instance that can span multiple localities. Each phase is realized by a thread, which acts only on local data (threads do not span local-

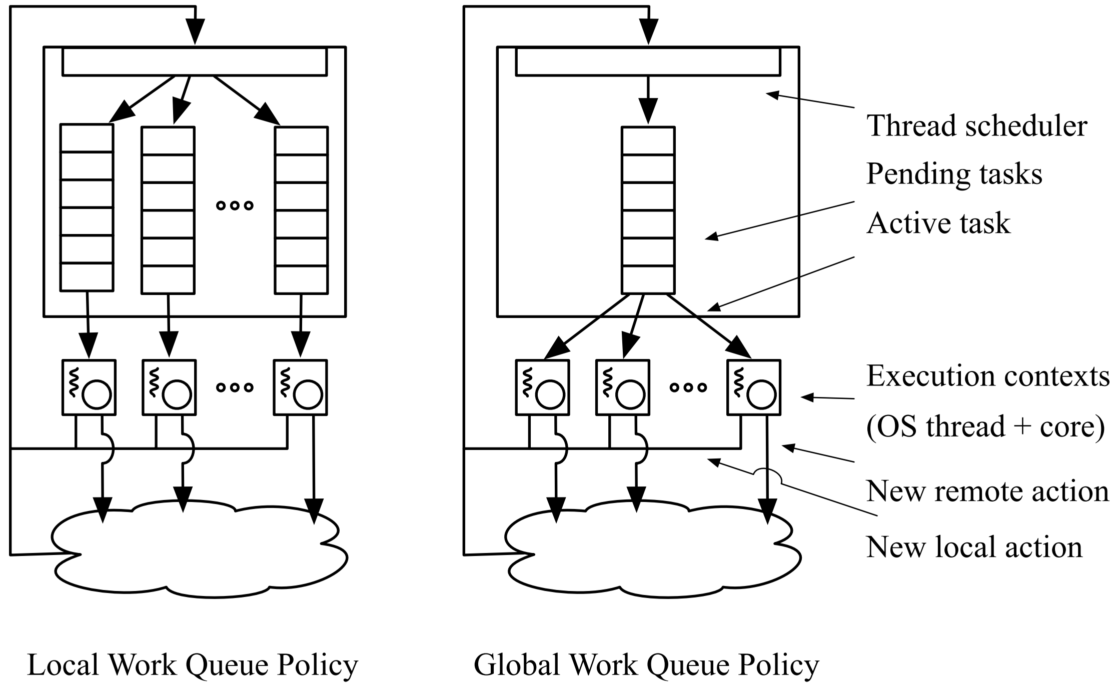


Figure 3.3. ParalleX work queue model with two policies. The *local* policy (left) depicts one work queue of pending tasks per execution context. The *global* policy (right) depicts one queue for all available execution contexts.

ities, though they can access remote data). The transaction is *split* by potential long latency requests, such as those for I/O or waiting for a result of separate (possibly remote) action. This phasing, when combined with the work queue model, allows for exploiting medium- to fine-grain parallel workloads.

Coordination of concurrent actions is managed through the use of lightweight primitives called Local Control Objects (LCOs). LCOs are named objects that capture the semantics of a variety of synchronization primitives, such as dataflow templates, full-empty bits, and futures [51, 52, 53]. LCOs provide the framework for defining constraint-based coordination of process execution by managing the conditions under which execution can proceed, selectively suspending and invoking actions based on the local state information. This use of local synchronization primitives replaces the global barrier constructs used by other models with localized lightweight synchronization points that can be adapted to the particulars of the algorithm to decrease unnecessary starvation of compute resources.

3.2 An Enabling Software Runtime System HPX

The HPX runtime system is a portable and modular, low-level software system implementing the ParalleX model [14]. HPX provides a low-level interface for writing ParalleX applications using a common abstraction over commodity multicore distributed memory clusters and shared memory systems. Portability is stressed in two dimensions: operating system and system architecture. HPX is tested and deployed across Linux, Mac OS X, and Windows HPC operating systems. Also, it runs on systems ranging from single or multicore laptops and workstations, to larger shared memory systems and commodity clusters — making it an excellent platform for initial code development, debug testing, and large-scale science runs. The experimental ParalleX Graph Library and all graph kernels demonstrated in this work are implemented using the low-level programming interface exposed by HPX.

The design philosophy behind HPX is to provide a modular and tunable implementation of the major semantic elements of the ParalleX model (See Figure 3.4). Modularity allows for the selection of point implementations optimized for a given target architecture. An example of this is the removal of the software implementation of global address translation on shared memory systems in favor of the available hardware support. This also opens up opportunities for integrating specialized hardware devices, such as an FPGA-based thread scheduler or accelerators in the form of GPGPUs. Each module also exposes sets of tunable parameters as policy interfaces. The thread scheduler alone allows for choosing the number of operating system threads and work queues, the mapping between these, and different local work stealing policies. All of this leads to a highly flexible system that can be tailored for high performance.

The following sections document implementation and programming details relevant to this work.

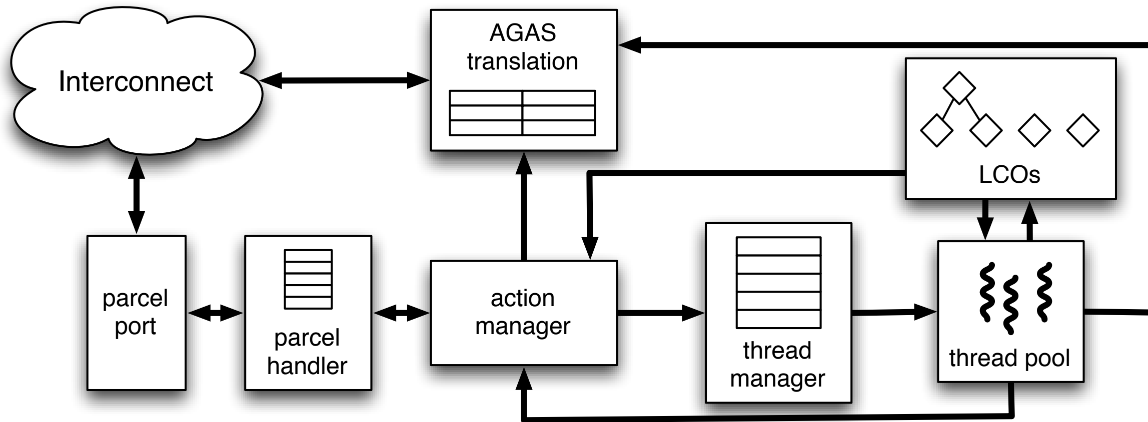


Figure 3.4. HPX software system architecture showing the primary modules providing key functionality.

3.2.1 Implementation of the ParalleX Feature Set

Every instance of a running HPX executable is a locality, effectively mapping the notion of a locality to the system-specific concept of a conventional process. On a shared memory system, a single HPX runtime is instantiated, so the entire shared memory node is treated as a single locality. In the case of a commodity cluster environment, a single HPX runtime is instantiated on each node, so that each node is treated as a separate locality. HPX handles the discovery and coordination of independent runtimes for multilocality runs, presenting a single system view to the user application. This is done by setting up all AGAS and parcel services, prior to executing the application's main thread.

Every instance of a named object in ParalleX, and so in HPX, is associated with a globally-unique identifier (GID). A GID is an unstructured 128-bit integer that can be used to determine whether the object is local to the caller, which locality it is on, and what its physical address is. This is all done through the AGAS module. HPX implements the AGAS using a client-server model. One locality hosts an AGAS network service, implemented over TCP/IP (for Ethernet), that maintains the mapping of all global identifiers (GIDs) to named objects in the system (e.g., Threads, LCOs, etc.). All other localities participate as clients of this AGAS service, sending queries to resolve the current location of an object. Local caching of GID translation information is done when possible to improve scalability.

Parcel messaging support in HPX is divided between two modules: a parcel handler and a parcel port. Each locality hosts a network service, implemented over TCP/IP, called a parcel port and another module called the parcel handler. This service is responsible for receiving incoming parcels from the network and sending new parcels generated from its locality. Received parcels are either forwarded to the parcel port of a different locality or passed to the parcel handler. The parcel handler interfaces between the parcel port and the action manager, translating newly arrived parcels into actions to be executed on the locality, or receiving newly created parcels destined for remote localities.

The action manager module translates action definitions (actions for short) into parcels or threads. When receiving an action definition, the action manager queries the AGAS to determine whether the action is to be performed locally or not. If it is to be executed remotely, it passes the action to the parcel handler. Otherwise, it passes the action to the thread manager, which is responsible for generating a ParalleX thread and enqueing it in the work queue.

HPX implements the multithreaded work queue in a similar fashion as other popular software systems, such as Cilk++, Intel's Threaded Building Blocks, and Sandia's Qthreads packages [54, 55, 24]. Each thread manager has associated with it a set of OS threads and a set of work queues. Each OS thread is mapped to a hardware resource (core), and provides the actual execution context for a ParalleX thread. Each work queue is simply a data structure holding threads which are ready to run, but have not yet been allocated an execution resource. Depending on the policy selection, there will be one work queue serviced by all OS threads, or a work queue per OS thread. In the later case, local work stealing is used for load balancing work across the OS threads.

An executing thread will run non-preemptively until it reaches a synchronization point or completes the action. A synchronization point can be reached by interacting with an LCO, which has the ability to yield control of an execution resource by suspending the calling thread. After completing an action, the thread will transfer the result to the action's

continuation object (actually a reference to another LCO) and terminate. HPX provides a collection of basic LCOs implemented on top of low-level system-dependent synchronization primitives. These include mutexes, semaphores, full-empty bits [56], and futures [52, 53, 57], among others.

3.2.2 Application Programming

Given that HPX is a generic C++ library, application development is necessarily restricted to such. There is a separate effort underway to create a C language interface for development of ParalleX applications, called the XPI, but it was not yet available for use in this work. Instead, all development efforts for this work were coded through direct use of the HPX library using the core constructs of plain actions and components, discussed here.

A plain action[1] defines a task to be executed as a ParalleX thread. This is a wrapper around a C++ function, defining type information for the return value and arguments, and holding a reference to the function. If the function is to return a value, it is of type `hpx::actions::plain_result_action<>`, otherwise it is simply a `hpx::actions::plain_action<>`. Figure 3.5 shows the definition of a simple function and an associated plain action description.

Definition of multithreaded Fibonacci

```
1  int fib(int);
2
3  typedef actions::plain_result_action<                // Action type
4      int,                                             // Return type
5      int,                                             // Argument type
6      fib                                              // Function name
7  > fib_action;                                        // Action name
```

Figure 3.5. Plain action definition for Fibonacci. Line 4 specifies the return type of the action, Line 5 the argument type, Line 6 the name of the function, and Line 7 the name of the action.

[1]The term *plain action* is used to differentiate it from a *component action*, discussed later.

HPX defines the *apply* construct as a low-level mechanism used for spawning actions. The semantics are that an action is *applied* to a target object. If the target is local to the action, a thread is created and added to the local work queue. Otherwise, in the case where the object is not local, a parcel is created and the action is sent to the destination locality, where a thread will be created and added to that remote work queue. This has the benefit of hiding the tasks of determining the locality of an object and creating threads/parcels, and allowing for optimization of those operations by the runtime system implementation. A plain action, which does not return a result, can be invoked by calling `hpx::applier::apply<act>(tgt, ...)` method, where `act` is the action to complete, `tgt` is the target object, and `...` is a list of zero or more arguments for the action.

The apply construct can also take a continuation object for handling the result of an action. In HPX, a continuation is a type of object that can be *triggered* (updated) and, as a result, can cause other actions to occur. In the case of applying an action that returns a value, the continuation is used as the destination for the result value returned by the executing function. The interface for this version of apply is `hpx::applier::apply_c<act>(tgt, ..., cont)`. These are the same arguments associated with the `apply<>()` version, but with the addition of the `cont` argument providing the continuation.

The application programmer is not likely to use the `apply_c()` interface directly. Instead, most HPX codes utilize the higher-level *futures* construct to spawn actions and retrieve the result value. The semantics here are that the action is spawned and the caller immediately receives a future-value LCO and continues execution. This future value acts as a promise of the result from the action. When the calling thread needs this value, it evaluates the future value, resulting in one of two scenarios, depending on whether the result value is available. If the spawned action completed and triggered its continuation (the future value) with the result value, then evaluating the future value simply returns the value to the calling thread. However, if the spawned action has not completed, then evaluating the future value will cause the calling thread to suspend, pending the availability of the result value — the thread

is sometimes referred to as being in a *depleted* state. Eventually, when the spawned action completes and triggers the future value with its result, the future value LCO will reactivate the suspended caller. Figure 3.6 shows the use of futures in implementing a multithreaded version of the Fibonacci function in HPX. HPX defines two classes for working with futures:

Definition of multithreaded Fibonacci

```
int fib (int n)
{
    if (n < 2)
        return n;

    hpx::lcos::future_value<int> n1 =
        hpx::lcos::eager_future<fib_action>(here(), n - 1);
    hpx::lcos::future_value<int> n2 =
        hpx::lcos::eager_future<fib_action>(here(), n - 2);

    return n1.get() + n2.get();
}
```

Figure 3.6. Usage of HPX eager futures in calculating the nth Fibonacci number.

`hpx::lcos::eager_future<>` and `hpx::lcos::future_value<>`. In order to create a future value, the user calls `hpx::lcos::eager_future<act>(tgt, ...)`. (Note: all the arguments are the same as those for the `apply<>()` method.) The result of this call is an instance of a `future_value<>`. In order to retrieve the value of the future (i.e., to evaluate it), the user can call the `future_value<>::get()` method. From the perspective of the calling thread, such a call always returns the result: any suspension of the thread is handled transparently.

This support is sufficient for designing and executing multithreaded codes in HPX. Figure 3.7 illustrates the execution of the example Fibonacci code. The top-most ParalleX thread represents the initial execution of `fib(4)`. The two arrows originating from this ParalleX thread indicate the spawning of the `fib(3)` and `fib(2)` actions, corresponding to lines 6–9 in Figure 3.6. The initial thread is then suspended, at line 11, when it attempts to retrieve the value of the `fib(3)` action, which is not ready. This pattern is recursively applied for values

of `n` greater than or equal to 2, at which point final values become ready and the suspended threads are reactivated. As illustrated, the circles and diamonds labeled 'v' denote the future LCOs providing the coordination for this multithreaded computation.

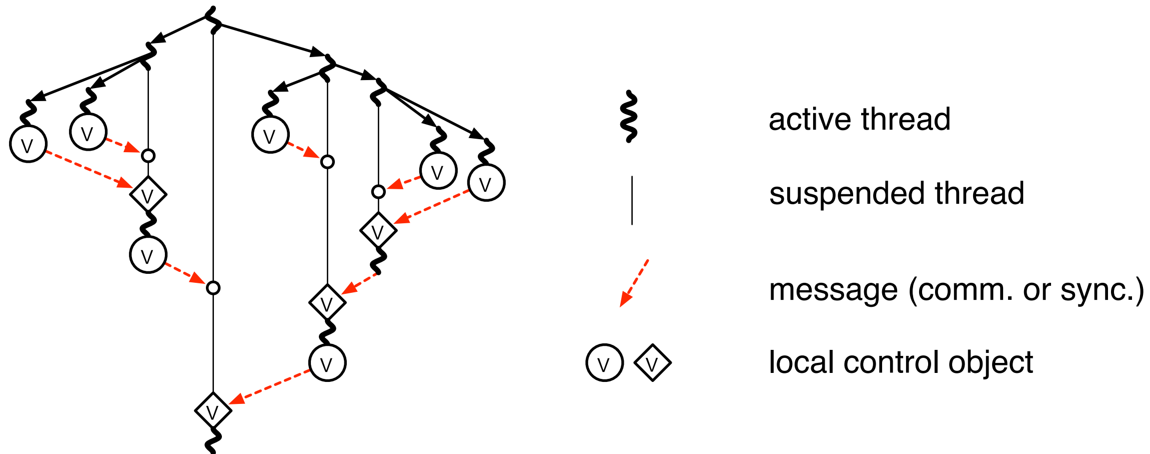


Figure 3.7. Multithreaded execution of Fibonacci code showing the spawning of ParalleX threads interdependencies through suspension and synchronization, for an execution of `fib(4)`.

However, plain actions and LCOs do not provide the means for defining application-specific persistent named data structures. That is where HPX's component model comes in. A component encapsulates data and exposes a set of actions that can be used to query and/or update the component. The component actions are effectively the same as the plain actions discussed earlier, except that they can only target an object of a specific component type.

Coding of a component requires the definition of the core functionality and the actions. These are typically defined in a single class, by convention called the component's *server* implementation. Figure 3.9 is an example of a server implementation of a component. This component has associated with it a single integer value and actions for initializing, adding to, querying, and printing this value. Given the GID for an instance of this component, the programmer could add 42 to the internal value and then query for the updated value as follows:

```
hpx::applier::apply<accumulator_server::add_action>(gid, 42);
x = hpx::lcos::eager_future<accumulator_server::query_action>(gid).get();
```

Optionally, the programmer also can code *stub* and *client* implementations. The stub implementation is a class providing the component interface as a set of static C++ methods that encapsulate the use of the `apply<>()` and `eager_future<>()` methods for interacting with the server (seen above). The client implementation is another class providing the same interface, but this time it encapsulates the use of the stub interface. Figures 3.10, and 3.11 round out the full implementation of the component in HPX. The following code shows how to create a new instance of the component, initialize it, add 42, and then query for the internal value, and retrieve the GID using the client interface (lines 2–9); the use of the stub interface is also shown (lines 12 and 13).

```
// Client interface
accumulator_client accu;
accu.create(next());

accu.init(0);
accu.add(42);
long x = accu.query();

hpx::naming::id_type gid(accu.get_gid());

// Stub interface
accumulator_stub<accumulator_server>::add(gid, 42);
x = accumulator_stub<accumulator_server>::query(gid);
```

The execution of the above user code for a multilocality run is illustrated in Figure 3.8. The single ParalleX thread in locality L_0 (far left) represents the execution of the user code. Creation of the accumulator component (line 3) is a synchronous action that spawns a ParalleX thread on L_1[2] to create the new component instance there; hence the suspension of the thread during the remote action. When the component is ready, lines 5–7 spawn

[2]In a multilocality configuration, HPX maps the set of N localities to a vector $\langle L_0, L_1, \ldots, L_{N-1}\rangle$. The method `next()` is a helper routine that, when called from locality L_i, returns the GID of locality L_j, where $j = (i+1) \bmod N$.

two asynchronous remote actions and one synchronous remote action. These details are obscured by the high-level interface exposed by the client implementation, but review of the stub implementation (Figure 3.10) shows that the `init()` and `add()` methods use the `apply<>()` interface for spawning their respective component actions, while `query()` uses an `eager_future<>()` LCO.

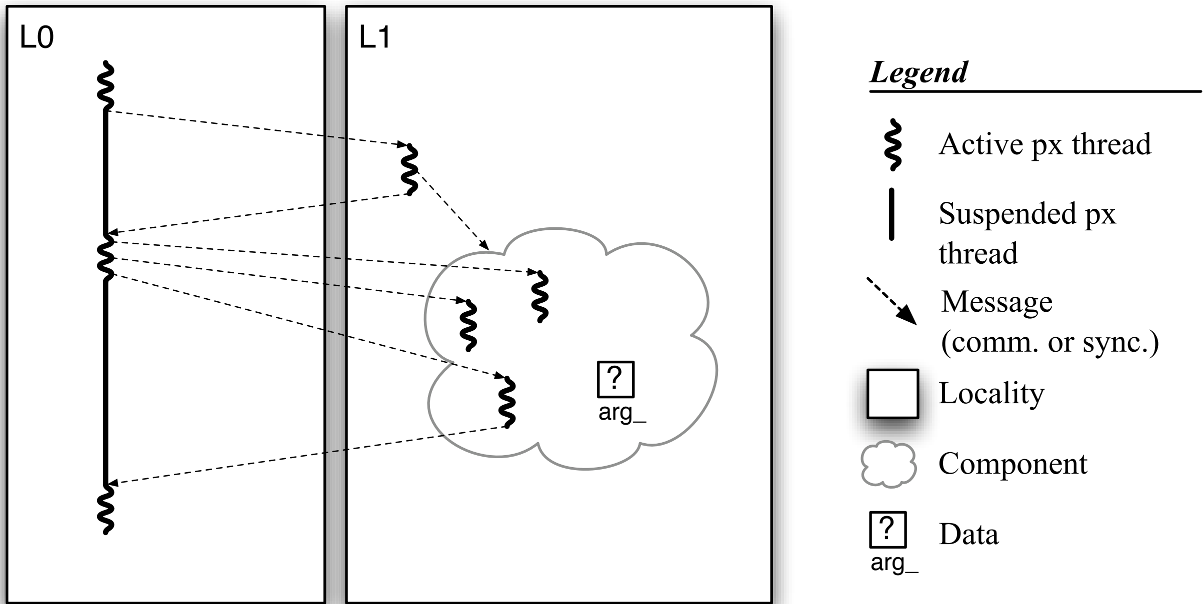


Figure 3.8. Multilocality execution of accumulator code.

Definition of Accumulator component

```
class accumulator_server
  : public hpx::components::detail::managed_component_base<accumulator>
{
public:
    enum actions
    {
        accumulator_init = 0,
        accumulator_add = 1,
        accumulator_query_value = 2,
        accumulator_print = 3
    };

    void init() { arg_ = 0; }

    void add (long arg) { arg_ += arg; }

    long query() { return arg_; }

    void print() { std::cout << arg_ << std::endl; }

    typedef hpx::actions::action0<
        accumulator, accumulator_init, &accumulator::init
    > init_action;

    typedef hpx::actions::action1<
        accumulator, accumulator_add, long, &accumulator::add
    > add_action;

    typedef hpx::actions::result_action0<
        accumulator, long, accumulator_query_value, &accumulator::query
    > query_action;

    typedef hpx::actions::action0<
        accumulator, accumulator_print, &accumulator::print
    > print_action;

private:
    long arg_;
};
```

Figure 3.9. Component server definition for accumulator. Source: HPX code base `examples/accumulator/accumulator/server/accumulator.hpp`.

Definition of Accumulator component

```
1  struct accumulator_stub : hpx::components::stub_base<accumulator_server>
2  {
3      static long query(hpx::naming::id_type const& gid)
4      {
5          typedef accumulator_server::query_action action_type;
6          return hpx::lcos::eager_future<action_type>(gid).get();
7      }
8
9      static void init(hpx::naming::id_type gid)
10     {
11         hpx::applier::apply<accumulator_server::init_action>(gid);
12     }
13
14     static void add (hpx::naming::id_type gid, long arg)
15     {
16         hpx::applier::apply<accumulator_server::add_action>(gid, arg);
17     }
18
19     static void print(hpx::naming::id_type gid)
20     {
21         hpx::applier::apply<accumulator_server::print_action>(gid);
22     }
23 };
```

Figure 3.10. Component stub definition for accumulator. Source: HPX code base `examples/accumulator/accumulator/stubs/accumulator.hpp`.

```
                Definition of Accumulator component
class accumulator_client
  : public client_base<accumulator, accumulator_stubs>
{
    typedef client_base<accumulator, accumulator_stubs> base_type;

public:
    accumulator(hpx::naming::id_type gid)
      : hpx::components::base_type(gid)
    {}

    void init()
    {
        this->base_type::init(gid_);
    }

    void add (long arg)
    {
        this->base_type::add(gid_, arg);
    }

    void print()
    {
        this->base_type::print(gid_);
    }

    long query()
    {
        return this->base_type::query(gid_);
    }
};
```

Figure 3.11. Component client definition for accumulator. Source: HPX code base `examples/accumulator/accumulator_client.cpp`.

Chapter 4

Advanced Semantics for Graph Processing

A directed graph is a general abstraction for representing entities and relationships between entities. The entities are represented by vertices, and the relationships by pairs of vertices, called edges. In the field of *Graph Theory*, a directed graph is formally defined as an ordered pair (V, E), where $V = \{u, v, w, \ldots\}$ is the set of vertices and $E = \{(u, v) : u, v \in V\}$ is the set of edges. Vertices are the principal structural element of the graph. In a directed graph, each vertex has associated local meta-data defining the sets of out-going edges and neighboring vertices. The total number of vertices in a graph is refered to as the *order* of the graph. An edge $e = (s, t)$ represents a single piece of structural meta-data, the relationship between two vertices. In this case, e is said to have source s and target t. Likewise, s is said to have out-going edge e and neighboring vertex t. The total number of edges in a graph is refered to as the *size* of the graph.

An attributed graph is defined as a graph with one or more associated attribute maps. An attribute map is a functional mapping either from vertices to vertex labels or from edges to edge labels. This definition supports many separate views of the same structure by keeping the structure of the graph seperate from any particular labeling of vertices and/or edges. A social network indicating friendship between individuals has the same structure independent of whether the vertices are labeled with the individual's name, home zipcode, or some other value relevant to an end user analyst. This is important because multiple simultaneous views of the network support different concurrent algorithms: a Breadth First Search might color vertices as it traverses the network finding shortest paths from an individual, while a component finding algorithm would label vertices with their respective component.

Common operations on graph structures can be classified as simple actions that access elements of the graph or return structural metrics, and compound actions that traverse or search the graph. Simple actions are primarily concerned with querying the graph for certain measures, such as the order of the graph or the degree of a vertex, or accessing individual vertices and edges. Traversal operations either involve the vertex and edge sets, or the structural meta-data. An example of the former is searching for a specific vertex in the vertex set. Graph algorithms for solving the Single Source Shortest Paths problem or performing a Breadth First Search are examples involving the latter.

Graph-based applications compose many graph operations along with auxiliary actions and data structures for constructing and storing problem solutions. An example graph kernel is used throughout this chapter to define the requirements for scalable graph processing and present the approach taken in this work. The kernel is concerned with performing many concurrent subgraph extractions[1]. A graph $G = (V, E)$, a set of edges $L \subseteq E$, and a depth value d are given. The solution is a collection of subgraphs $S = \{H_0, H_1, \ldots, H_{|L|-1}\}$. The problem is to populate S with subgraphs H_i corresponding to each $e_i \in L$.

The following sections define a new approach to graph processing designed to support these requirements using the ParalleX execution model. Section 4.1 establishes the use of asynchronous actions in a continuation-migration style of algorithm design for graph application programming. Section 4.2 presents the *phased use* semantics for coordinating highly concurrent execution over shared data structures. Section 4.3 defines a technique for supporting graph traversal strategies with embedded control objects. And Section 4.4 discuses supporting scalable execution over distributed memory systems.

[1]This is the third graph application kernel defined in the parallel graph benchmark, and is covered in more detail in Section 6.2.

4.1 Asynchronous Actions and Continuation Migration

The approach taken to graph processing in this work encourages fine-grain asynchronous actions that carry continuations. A basic **apply** construct [58] is used for invoking an asynchronous action. This is the same apply construct provided by the HPX `apply<>()` and `apply_c<>()` methods (Section 3.2), and has the general form:

```
apply(<target> : <action> [, <arg>]* [-> <continuation>]);
```

A *target* specifies the object on which the action should be applied. The target can be a data object, such as a vertex, or a locality domain. Either way, the target determines where the action is executed. An optional argument list and *continuation* can be specified. The continuation is embodied as an local control object (LCO) that will receive the return value $result$ through an implicit follow-on call to $\textbf{apply}(continuation : set, (result))$.

The futures construct [52, 53] can be used for invoking an asynchronous action and synchronizing on the result. This is done by creating a future-value LCO and passing that as the continuation of an *apply*. When the result of the action is required, the future-value is evaluated. If the result value is not available, evaluation will suspend the calling thread until the value becomes available. The creation of the future-value and the spawning of the action are encapsulated by using the `eager` keyword[2], as seen in the following example.

The naive Fibonacci action introduced in Section 3.2 provides a good example of the use of futures. Algorithm 4.1 defines the $Fib(n)$ action in terms of the semantics just defined. Lines 4 and 5 spawn asynchronous actions to compute $fib(n-1)$ and $fib(n-2)$. Line 6 synchronizes on the two future-values, sums them, and returns its value. Figure 4.1 illustrates the execution pattern using futures. The circles labeled v denote the setting of a future-value, while the diamonds and small empty circles denote an evaluation of a future-value. The dashed arrow

[2]This corresponds to the `eager_future<>()` method in the HPX runtime system.

identifies a causal relationship between the setting and evaluation of a particular future-value. The straight lines show suspension of a thread pending the availability of a future-value.

input : integer n

1 **if** $n < 2$ **then**
2   **return** n
3 **end**
4 $n_1 \leftarrow$ eager Fib$(n-1)$
5 $n_2 \leftarrow$ eager Fib$(n-2)$
6 **return** $eval(n1) + eval(n2)$

Algorithm 4.1: Recursive Fibonacci action $Fib(n)$.

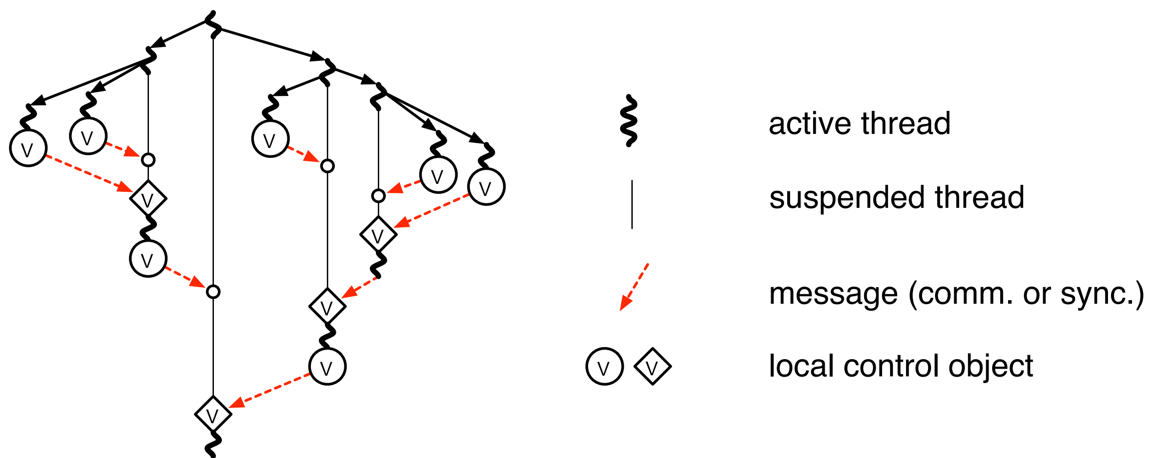


Figure 4.1. Execution of the recursive Fibonacci process $Fib(4)$.

Futures provide a useful method for spawning asynchronous actions and synchronizing on their associated return values. However, the binding of action and return value is not applicable in all situations. Some actions are better designed using a more general strategy of moving the continuation into the action, as opposed to passing the continuation along with the action as the *continuation* parameter of an **apply**. This can increase flexibility by allowing an action to choose between multiple continuation paths. It also removes the need to design actions around return values, which is required by the future-value approach. The dataflow-variable construct [59] is introduced as a replacement for future-values when anonymous producer/consumer execution is preferred. A dataflow-variable is similar to a

future-value in that it can be assigned a value, and evaluation will suspend the caller until a value is available. The difference is that there is no producer action associated with a dataflow-variable.

A simple example of anonymous producer/consumer continuation migration is demonstrated by a recursive algorithm for calculating the factorial of a positive integer, as defined by the $Fact$ and $Fact'$ actions (Algorithms 4.2 and 4.3, resp.). Algorithm 4.4 shows the use of a dataflow-variable k to synchronize on the result of a call to $Fact(6, k)$ (Figure 4.2 illustrates the execution). The dataflow-variable k is passed from one short-lived action to the next, until the tail of the recursive process is reached, and k is assigned the solution value. No call chain is maintained between threads, and it is likely that the system will have retired nearly all previous instances of the $Fact$ and $Fact'$ actions by the time final action executes. This leaves the dataflow-variable k as the connection between the caller and the solution.

input : integer n, LCO k

1 **apply** $(Fact', (n, 1, k))$

Algorithm 4.2: Continuation-passing style Factorial action $Fact(n, k)$.

input : integer n, integer a, LCO k

1 **if** $n = 1$ **then**
2   **apply** $(k, set, (a))$
3 **else**
4   **apply** $(Fact', (n - 1, n \cdot a, k))$
5 **end**

Algorithm 4.3: Continuation-passing style Factorial action $Fact'(n, a, k)$.

1 $k \leftarrow$ dataflow-variable
2 **apply** $(Fact, (3, k))$

`// Intermediate work not depending on the value of Fact(3)`

3 $v \leftarrow$ **eval** (k)

Algorithm 4.4: Example using CPS Factorial action.

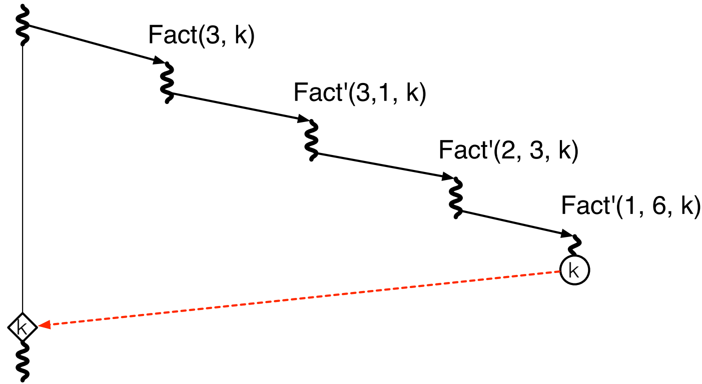


Figure 4.2. Execution of the CPS Factorial process $Fact(3, k)$.

The execution pattern seen in Figure 4.2 is not possible using the future construct. This is because the futures construct provides a simple synchronization point on the completion of a specified action. The spawning of the action is combined with the creation of the LCO, effectively tying creation and initialization. The dataflow variable construct provides a more general synchronization point where creation, initialization, and use are all independent. Algorithm 4.5 defines a classic example demonstrating this point. A new dataflow-variable d is created and passed off to four asynchronous threads. The first three threads (spawned on Lines 2–4) attempt to print the value of the d. The final thread (spawned on Line 5) sets the value d. The order of execution of these four threads does not matter because each of the printing threads waits for the value of d to be set by the final thread. Figure 4.3 illustrates this independence in the execution pattern.

```
  // Create dataflow variable 'd'
1 d ← dataflow-variable
  // Spawn actions that use value of 'd'
2 apply (λ x.(print eval (x)), (d))
3 apply (λ x.(print eval (x)), (d))
4 apply (λ x.(print eval (x)), (d))
  // Spawn action to initialize value of 'd'
5 apply (λ x.(apply (x : set, (42))), (d))
```

Algorithm 4.5: Example using dataflow-variable for anonymous producer/consumer execution.

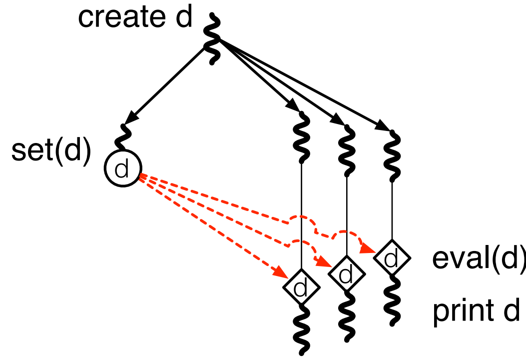


Figure 4.3. Execution of dataflow-variable example Algorithm 4.5.

Now that the basic constructs for asynchronous actions, continuation migration, and synchronization have been introduced, loop abstraction is provided to complete the consideration of basic support necessary for this approach to graph processing. Loop abstractions encapsulate patterns of concurrent execution. By raising the level of abstraction they provide the programmer with a tool that simplifies the program and enhances their ability to reason about their concurrent execution. The policy and runtime implementers are able to use this raised abstraction to provide optimized solutions that best match the requirements of the programmer with the reality of the target operating environment (e.g., OS, architecture, technologies).

The **forall** construct is an example of a loop abstraction that maps an action to each item in a container. The container can be a set of edges, as in the graph example, or a range of integers. An example of the latter is shown in Algorithm 4.6, which is an extension of Algorithm 4.5. The print actions are now spawned from a **forall** over the range $[0, 1000)$. As before, the synchronization is managed by a dataflow-variable, demonstrating the use of this construct as a general barrier. More loop abstractions are covered in Section 5.5, and used extensively in the implementation of the benchmark graph application kernels in Chapter 6.

1 **forall the** $i \in [0, 1000)$ **do**
2   **apply** $(\lambda\ x.(\text{print } \mathbf{eval}\ (x) \cdot i)), (d))$
3 **end**
4 **apply** $(\lambda\ x.(\mathbf{apply}\ (x : set, (2))), (d))$

Algorithm 4.6: Example using **forall** to print multiples of 2.

4.2 Constraint-Based Coordination through Phased Use

Returning to the example, the subgraph extraction graph kernel is considered in terms of the approach outlined in the previous section. The main kernel action can be reduced to the spawning of $|L|$ asynchronous subgraph extractions, since each subgraph H_i can be computed in parallel. The typical means for this is to *fork* asynchronous subgraph extraction actions, and wait on each to *join* back to the calling thread. Each returned subgraph is then inserted into the solution set of subgraphs S. The action terminates when all subgraphs have been added. This execution pattern can be expressed concisely with future-values as in Algorithm 4.7.

```
  // F an empty set of future-values
1 forall the e ∈ L do
2     F ← eager ExtractSubgraph(e, d)
3 end
4 forall the f ∈ F do
5     S ←eval (f)
6 end
```

Algorithm 4.7: Example implementation of graph kernel action with future-values.

To increase the efficiency in a dynamic multithreaded model such as ParalleX, opportunities for increased asynchronous actions and continuation-migration must be exploited. If the action of inserting H_i in S is moved into the corresponding subgraph extraction action as the continuation, then the insertions will happen as soon as possible in parallel. Algorithm 4.8 presents the updated approach. Note that there is no longer any synchronization performed in this action. The future-values were removed, and a simple asynchronous apply is used. Proper synchronization is still maintained, but now it is between the producers of the subgraphs which initialize S, and the consumers of S. Figure 4.4 shows the difference in the execution pattern between these two approaches, highlighting S (the blocks in the figure) as a synchronization point.

1 **forall the** $e \in L$ **do**
2 **apply** $(ExtractSubgraph, (e, S, d))$
3 **end**

Algorithm 4.8: Example implementation of graph kernel action with continuation-passing.

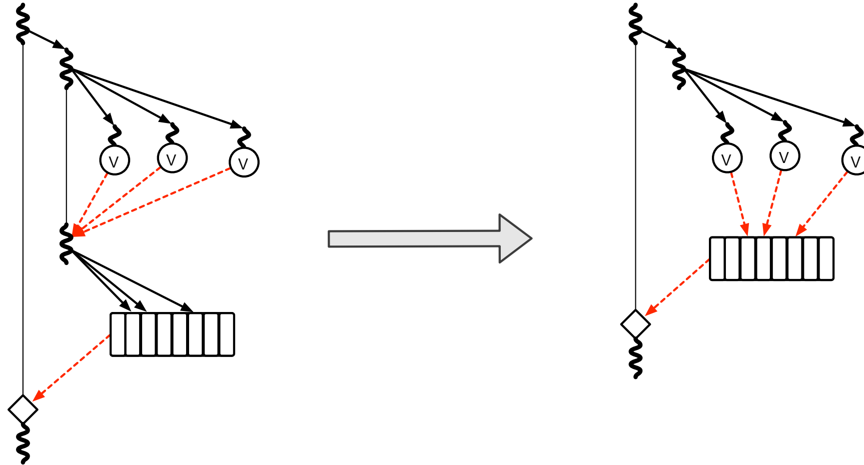


Figure 4.4. Comparison of graph kernel action execution.

This approach encourages the design of self-coordinating actions in place of programmer-managed actions. There is no longer a need to explicitly synchronize on the completion of the subgraph extraction actions, as all actions must have completed before S is fully initialized. Moving the continuation of execution into the subgraph extraction action makes S the natural synchronization point for the application, not the *joining* of the asynchronous extraction actions. This also means that the action which spawned the subgraph extraction kernel does not need to wait on the action to complete before moving forward. Instead, it only waits (synchronizes) if it needs to use S before it is initialized.

Synchronization on S is supported by a so-called *phased use* semantics, which is an extension of the dataflow-variable semantics to compound data structures. Phased use refers to two properties of a data type. First, an instantiation of a data type may be in one of three phases: *construction*, *initialization*, and *use*. Second, all actions over the data type are associated with one or more phases. The set S supports insertion of items, but that action is only permitted when S is in the initialization phase. This prevents attempts to insert before

the data structure has been constructed. Likewise, S supports access of items, but that is not permitted until S has reached the use phase. Invoking an action before the target data has reached the associated phase will cause the instance of the action to suspend. When the container reaches a new phase, it releases all actions suspended on that phase. Properly used, the phased use semantics can increase the level of exploitable parallelism while removing a principal opportunity for observable nondeterminism and associated data races when processing shared data structures such as graphs.

4.3 Graph Traversal Strategies through Embedded Coordination

The implied linear ordering of items in an array allows for traversing the array either forward or backward from any given item. The **foreach** loop construct can be used to access each item in the array in order. A set lacks any defined ordering over items. However, the **foreach** construct can still be used over a set, with the order in which items are accessed generally undefined. A graph, which is a collection of vertices and edges, does not have a single ordering, but many possible orderings. The two mechanisms for traversing a graph are iteration through the vertex or edge sets, and path finding through the graph meta-data structure. The set of vertices and the set of edges can be traversed when the goal is to simply access each vertex or edge, regardless of order and irrespective of the graph structure in both cases. In the case of identifying all edges with a weight greater than a specified value, it is likely that the action would work over the edge set of the graph.

Simply processing the vertex and edge sets is common in graph applications, but the traversal of graph meta-data is at the core of many important graph algorithms. The task of finding all shortest paths from a given vertex s is often described in terms of a traversal from s to all reachable vertices using a Breadth First Search (BFS). The solution for finding components in a graph can use a Depth First Search (DFS) from a vertex to identify the

component. In each case, the graph algorithm is an adaptation of a particular traversal strategy. The point is demonstrated by a simple algorithm for coloring all vertices found in a DFS from start vertex s, such as in Algorithm 4.9, that could be used to color all vertices in a component[3].

```
  input : attributed graph G_C, vertex s
1 if C[s] = white then
2     C[s] ← black
3     forall the v ∈ s.neighbors() do
4         DFS(G_C, v)
5     end
6 end
```

Algorithm 4.9: Depth First Search action $DFS(G_C, s)$.

A given traversal strategy can be adapted by defining actions that should occur at certain event points. This is captured in the *algorithm visitor* [60, 21] pattern. A visitor is an object that embodies a specific traversal pattern and exposes a set of event points that can be used to invoke specified actions at different stages of the traversal. This work leverages the Local Control Object (LCO) construct to embed the visitor pattern into the graph structure. This integrates the control of the traversal and management of continuations into the graph structure using the constraint-based coordination provided by the ParalleX model.

In the running example, the subgraph extraction involves identifying a subgraph in G about a given edge e_i and creating a new graph structure H_i. A subgraph is identified as all vertices and edges along any path of length d originating from e_i. This traversal strategy is embodied by a Parallel Search (PS). The necessary modifications to the PS are ending traversal when a path has reached the specified depth d and inserting new vertices and edges into H_i as they are found in G. This only requires adapting three event points: first visit to a vertex, subsequent visit to a vertex, and first encounter of an edge. When a path first visits a vertex in G, the vertex is labeled with the remaining depth of the path and inserted

[3]Note that a graph G with attribute map $C : V \rightarrow \{black, white\}$ is denoted G_C.

into H_i, and the search is continued along each out-edge in parallel. Subsequent visits to a vertex will check the label to determine if it should continue to the neighboring vertices: if a longer search path has already encountered the vertex, this search path is abandoned; otherwise, the label is updated with the remaining depth of this path, and the search is continued. When a path crosses an edge for the first time, the edge is inserted into H_i These simple extensions of PS provide a concise definition of subgraph identification and extraction process. Figure 4.5 illustrates such a parallel search over a graph.

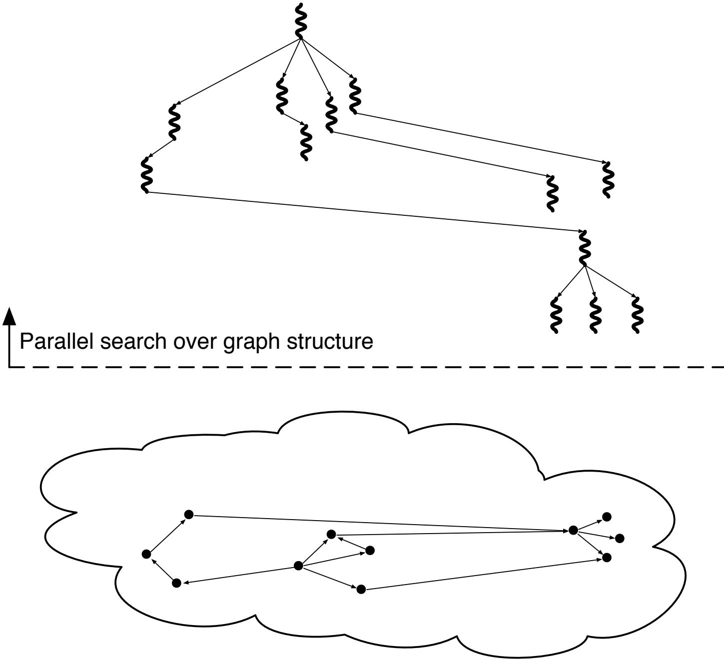


Figure 4.5. Execution of the parallel traversal of a graph structure.

4.4 Large-Scale Data through Distribution

Large-scale shared data structures spanning the entire system are prevalent in graph-based applications. As problem data set sizes continue to increase, graph-based applications will require distributed memory solutions. Yet the use of the previously discussed semantic constructs should be uneffected by the specifics of the system topology or scale.

Transparently scaling applications to multiple locality domains introduces two primary challenges: work migration for colocation of work and data, and distribution of the data structures. The ParalleX model supports work migration through the use of parcels. A parcel carries the necessary information to migrate the continuation of work local to the target data residing on a different locality. The **apply** semantics is sufficiently high-level so that the use of parcels is not seen to the programmer. If the programmer issues a query for the degree of a vertex using "**apply** $(v : degree(f))$", then the future-value f will eventually hold this value. If v resides in the same locality, a new ParalleX thread will be instantiated and the value of f set locally. If v resides on a different locality, a parcel will be created and sent to instantiate the action local to v; the value of f will then be set using another parcel that will migrate the *set* action local to the future-value. This means that the **apply** construct is independent of the locations of the caller, target data, and subsequent continuation LCO.

The distribution construct is defined to aid in managing the physical distribution of containers (e.g., an edge list or graph) over the system. The distribution comprises a key space, coverage, and distribution policy. The key space is the set of identifiers for the items in the container. The coverage is the set of localities over which the container extends. And the distribution policy defines the mapping from the key space into the coverage. Figure 4.6 depicts the role of a distribution in partitioning a container over a set of localities.

The key space refers to the domain of index values for items in a container. For an array of size N, the key space is the integer range $[0, N)$. For the directed graph, the distribution policy is applied over the vertices, so the key space is the vertex set V. This is done because a distribution over the edge set E might map edges (u, v) and (u, w) to different localities in its coverage, effectively distributing the vertex u, and any edge distribution that prevents such mapping is necessarily a distribution over V.

The coverage is the collection of locality domains allocated to a distributed structure. This allows for both bounding the coverage to prevent two containers from overlapping or to guarantee that two containers overlap. The latter is critical for *aligning* containers. Consider

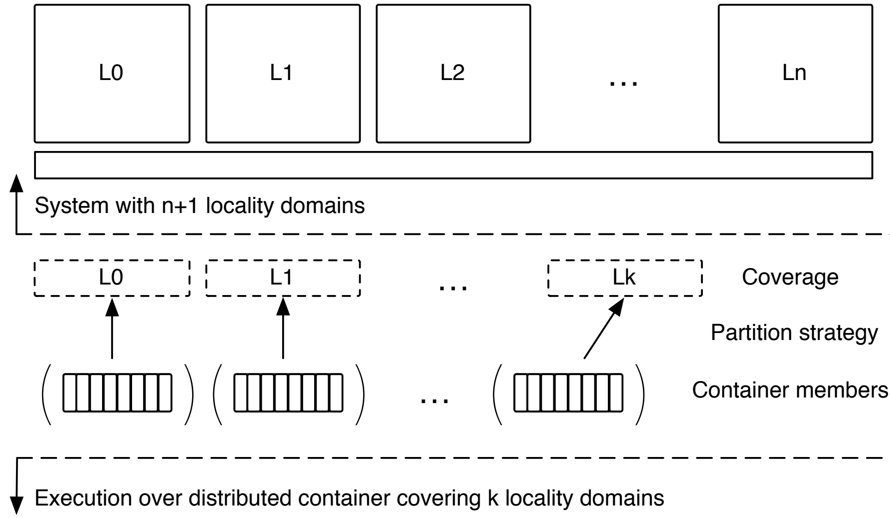


Figure 4.6. Illustration of the distribution construct.

the case of a graph $G = (V, E)$ and an attribute map $a : V \to \mathbb{R}$ defining real-valued weights on the vertices of G. A process traversing the vertices of G is going to access both the weight of the vertex and the structural information contained in the vertex. Therefore, it is important to colocate the two. This is easily achieved by having the graph and attribute map share a distribution over V, effectively aligning the local partitions of each distributed structure.

The distribution policy defines the partitioning strategy for items in a container by mapping item identifiers to localities in the coverage. This is intentionally left as a policy decision to allow for maximum flexibility and applicability to a broad range of applications. A simple example would be selecting between a partitioning strategy that uses the modulo of the integer identifier to map an item to a specific locality in the coverage. A more complex example would be a partitioning strategy that utilizes a full graph partitioning suite, such as Zoltan or ParMetis [61, 62], to define a robust partitioning scheme minimizing edge cut set for a particular data set. Either way, a balance must be found with respect to application and data set that may not always be known a priori.

From the perspective of the programmer, no modifications are necessary to the use of the concepts discussed in the previous section. The only difference is that the distribution must be specified for a structure spanning multiple localities. In the running example, distributing G, L, and S requires defining the distribution of each container. Since $L \subseteq E$, both L and E should be distributed such that corresponding edges are located in the same physical domain. Likewise, since each $H_i \subseteq G$, the subgraphs should also be *aligned* with G and L. This is achieved by defining a distribution over the vertex set D_V covering the available set of localities, and sharing this distribution across G, L, and S.

Loop abstractions can utilize the distribution of a structure to cover multiple locality domains. Given the edge list L has distribution D_V, the **forall** in Algorithm 4.7 will partition the work based on D_V. This is illustrated in Figure 4.7 when the edge list is distributed over $k+1$ localities. The algorithm is unchanged, as the **forall** manages the intermediate branching of the execution to spawn local **forall**s to each locality.

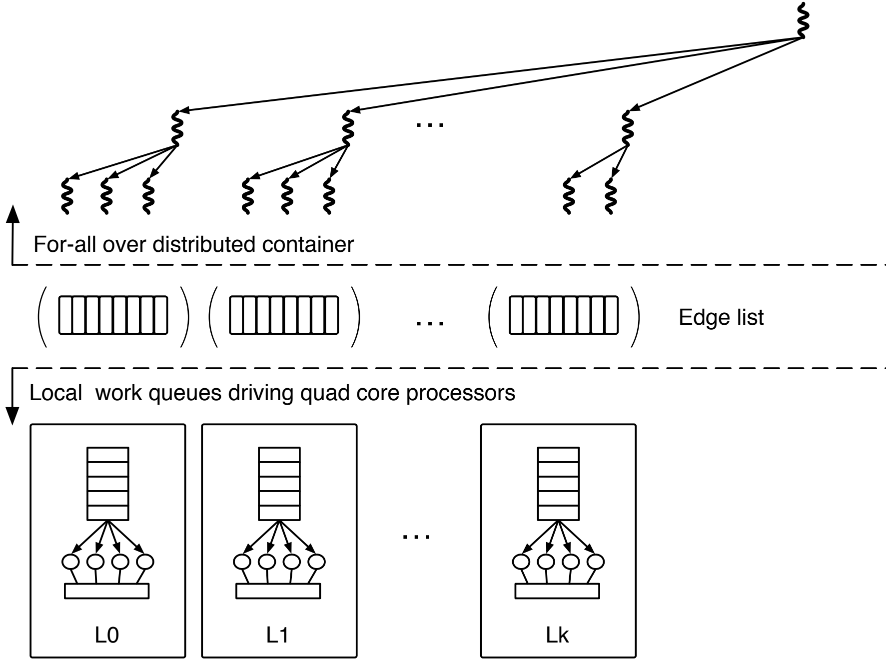


Figure 4.7. Execution of the distributed processing of an edge list.

The phased use semantics applied directly to distributed structure potentially creates a distributed barrier. A decentralized approach is taken to reduce the effects of synchronization. The container is defined as a collection of autonomous members. Each member is resident on a single locality in the container's coverage, and manages the local partition there. But the member interface provides a global view of the container. Queries for information about the container or access to items are always made to the member local to the caller. If a requested item is managed by a different member, the action will be migrated to that member (located using the container's distribution), transparent to the caller. Properly implemented, this will allow members on each locality domain to progress independently, only synchronizing during construction and initialization to ensure that the container is internally consistent and global information is propagated to each member. A particular implementation in the ParalleX Graph Library for graph and vector containers is discussed further in Chapter 5.1.

Finally, any distribution of vertices and edges will also distribute the traversal strategy. Embedding the coordination in the vertex means that the traversal strategy is naturally distributed with the vertices. The previous parallel search example is illustrated again in Figure 4.8 for the same graph distributed over $k + 1$ localities. Each graph member contains a disjoint subset of vertices and edges. When the parallel search is started on locality L_1, it spawns initial paths to three local vertices and one remote vertex on L_0. The execution pattern is the same as in Figure 4.5, showing the transparency of the distribution of the data structure on execution.

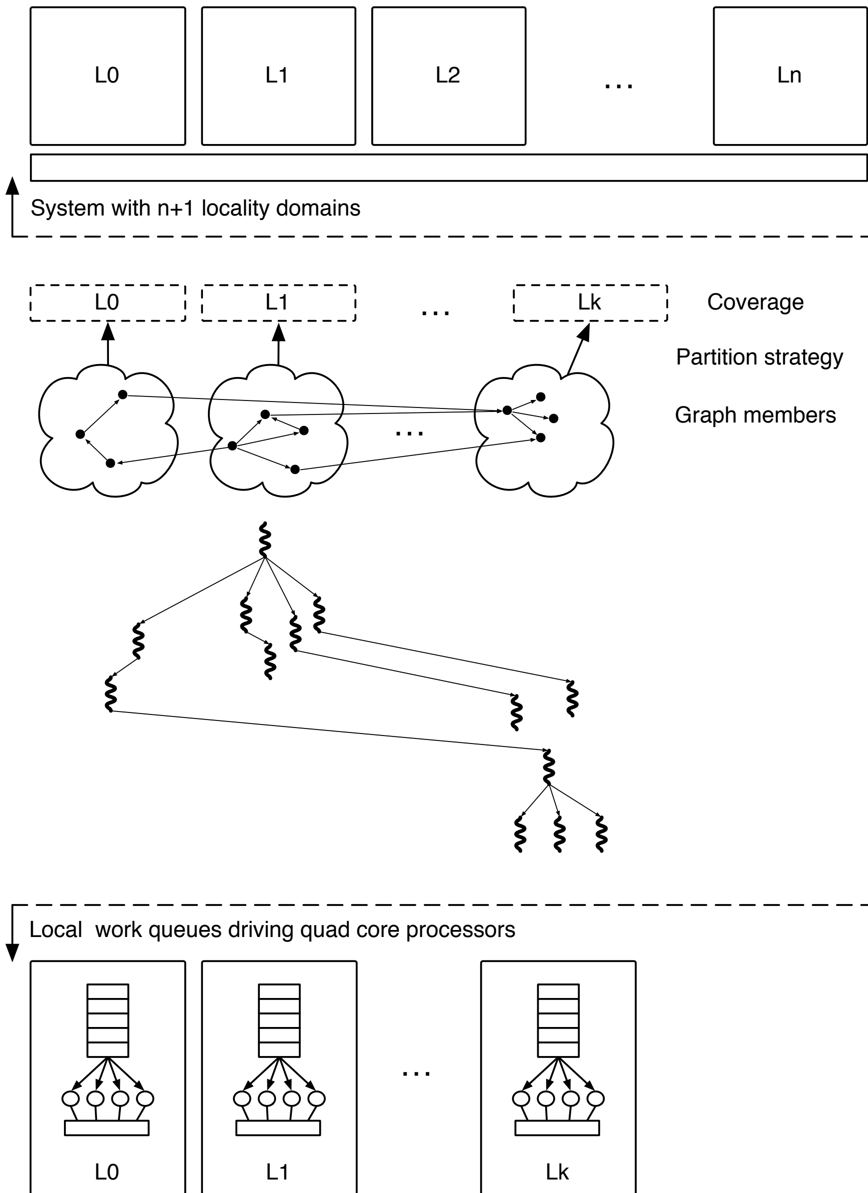


Figure 4.8. Execution of the distributed traversal of a graph structure.

Chapter 5

A ParalleX Graph Library

The ParalleX Graph Library (PXGL) is a software system focused on the utilization of intrinsic phased use semantics and data-directed techniques for highly scalable message-driven, dynamic multithreaded graph processing based on the ParalleX model of computation. The initial implementation of the PXGL was developed to directly leverage the High Performance ParalleX (HPX) runtime system. Section 5.1 discusses the implementation of distributed data structures in the PXGL employing phased use and decentralized coordination. Section 5.2 presents an example application code to demonstrate the use of the PXGL. Sections 5.3–5.5 define the data structures and constructs supporting graph application development with the PXGL.

5.1 Decentralized Data Structures

The distributed structures provided by the PXGL follow the phased use semantic introduced in Section 4.2. An instance of a structure is always in one of three phases: *construction*, *initialization*, or *use*. Each action on a data structure is associated with at least one of these phases. If an action is applied to the instance before it has reached the required phase, the action is suspended until the instance reaches that phase. This allows for actions over distributed structures to be invoked without regard to the current phase of the target, as the necessary coordination is integrated into the distributed structure.

The primary challenges presented by distribution are exploiting locality when possible and localizing synchronization points to reduce global barriers. The approach taken in the PXGL is to implement decentralized distributed structures. A distributed structure is decomposed into a set of autonomous *members*, where each member acts as a proxy for a local portion of

the distributed structure. Internally, each member maintains a mapping to all other members (its siblings). Global information such as the size and order of a distributed graph is also kept at each member. This provides a local target for actions on each locality covered by the distributed structure, preventing remote operations whenever possible.

Embedding the phased use semantic into a distributed structure increases the potential for the blocking phases to create global barriers in the application code. The decentralized approach affords the opportunity to manage the phases for each member, instead of managing a single phase across the entire coverage. Thus the construction and initialization processes are designed to allow each member to advance phases independently. Some internal coordination is necessary to establish connections among members and distribute global information. The following discusses balancing the necessary synchronization with the ability to advance phases from construction to initialization and initialization to use.

5.1.1 Construction

A distributed structure must be constructed through a coordinated effort between concurrent actions in order to create all members and establish the internal mappings between members. Declaration of a distributed structure instantiates a new named member of that type on a specified locality. The construction process begins when a distribution is defined for the initial member. Using the distribution, new members (siblings) are declared on each locality in the coverage. In order to construct the siblings, the initial member spawns an asynchronous replicate action on each. After spawning the replicate actions, the member is finished with construction and moves into the initialization phase.

The replicate action carries the distribution and the collection of sibling GIDs to each member. Each replicate action sets the distribution and the internal siblings map for the target member. Once completed, the member has finished the construction phase and moves into the initialization phase.

Figure 5.1 depicts this decentralized construction process for a distributed structure over four localities. The initial member is declared on locality L_2. The only synchronization involved in this process is the spawn and sync associated with creating members and collecting their GIDs, which is necessary to establish the internal mappings between members. The shaded areas denote the lifetime of each member from declaration to construction. And the circles labeled c correspond to the completion of the construction phase for each member.

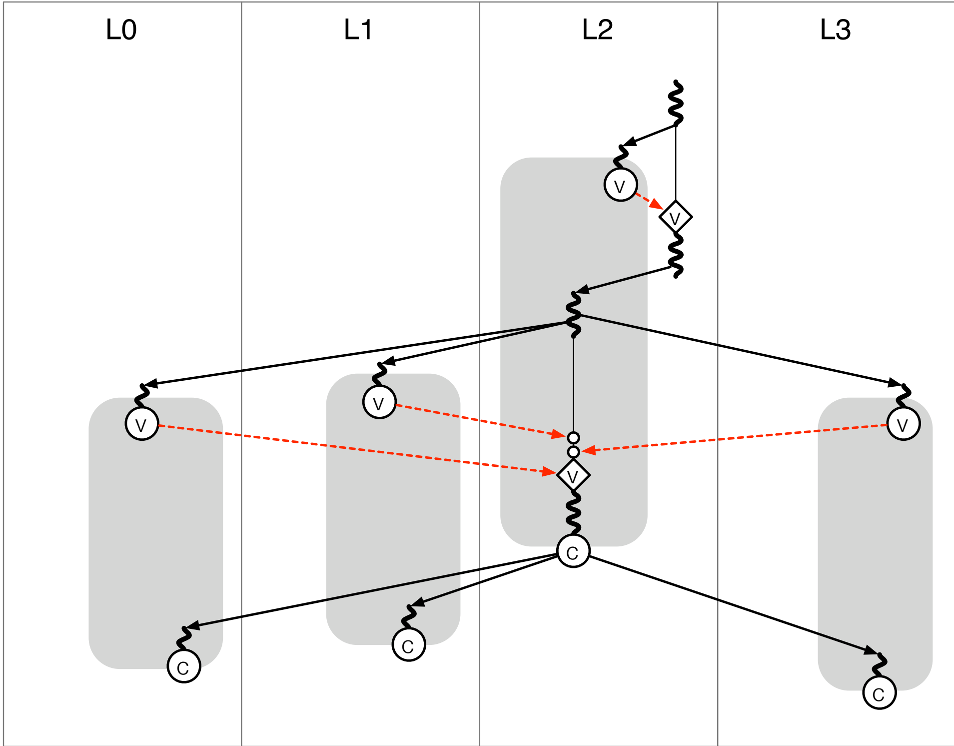


Figure 5.1. Decentralized construction process for a distributed structure in the PXGL.

5.1.2 Initialization

Distributed structure initialization is organized as a coordinated effort of independent members to establish a consistent global state across all members. The exact information that needs to be exchanged is specific to the distributed structure, but the basic patterns of execution and coordination are the same across all implemented structures in the PXGL. In the case of a vector, the global state is the total number of items in the global vector. When a

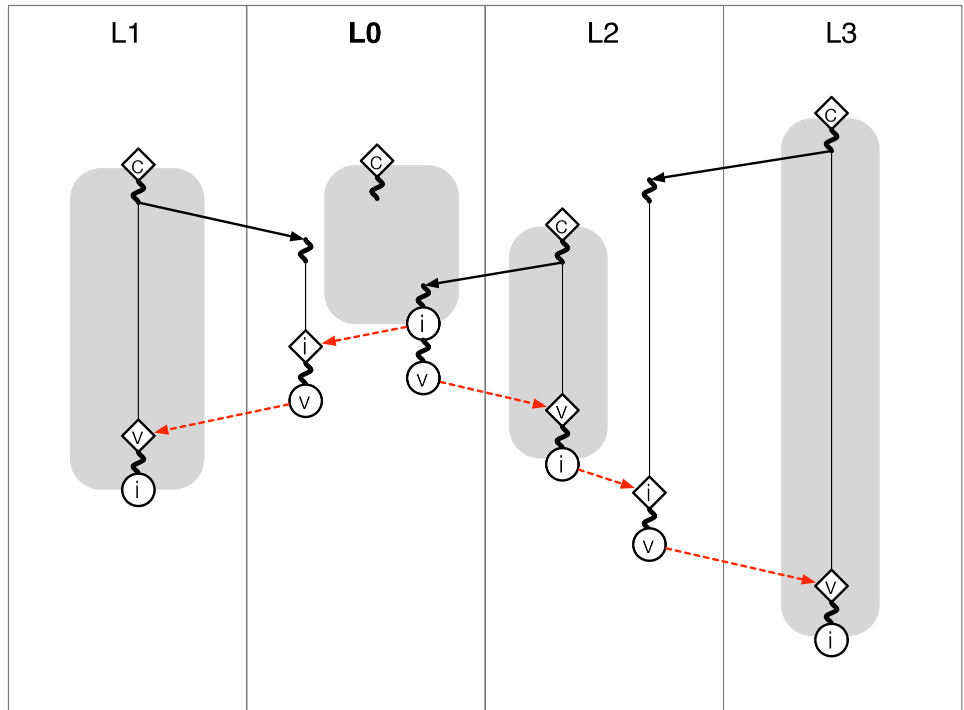
member is ready to use, this count will be available when querying any member, providing fixed-cost look up of global size, with no chance of a remote request being generated.

The initialization process is more involved than that for construction. The process is not initiated by one member, but independently for each member. Also, the members are mapped to a logical d-ary spanning tree defining a single *leader* and at most d *followers* for each member (note that the root of the tree has no leader). The process proceeds by synthesizing local information up this logical tree to the root, and then inheriting the global aggregate down the tree to all of the members.

When a member begins initialization it builds and populates the local data. Then it waits to receive local information from each of its followers. Once the local information is received by all followers, a synchronous update action is applied to the leader. This action is responsible for both sending the aggregate of local information to the leader and communicating the global information back to this member. This pattern can be seen in Figure 5.2. The spawns represent the synthesis of local information to the root member on L_0. The synchronization of future values (the v's) represents the inheritance of the global information to the non-root members. The shaded areas denote the lifetime of the initialization process for each member. The diamonds labeled c represent the precondition that the distributed structure must have been constructed before it can be initialized. And the circles labeled i correspond to the completion of the initialization phase.

5.2 Application Example: Scalable Data Generator

The Scalable Data Generator (SDG) procedure provides an example of how to write a simple application that takes advantage of the decentralized distributed structure and phased use semantics. The procedure involves generating a graph data set and writing the edge list to a file. The three main components to this procedure are the construction of a vector to store the generated edges, the generation of the edge tuples and initialization of the edge tuples



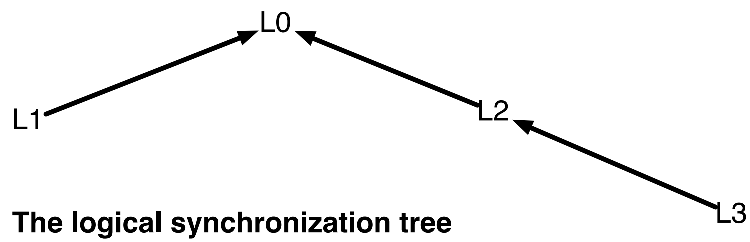


Figure 5.2. Decentralized initialization process for a distributed structure in the PXGL.

vector, and the writing of the edges to the file. Code listing 5.3 provides the definition of the main thread of execution. The application begins by constructing the edge tuple vector (Line 5). This is an asynchronous action allowing the main thread to continue execution while the distributed vector is built concurrently. Next, the `generate_rmat_data` action is spawned (Line 7), again asynchronously. This action is responsible for generating the edge data and initializing the edge tuples vector. Finally, the write action is applied to the vector, this time using the `blocking_for_all` data-directed construct (Lines 8–10). This is done to write each local portion of the distributed edge tuples vector to a local file.

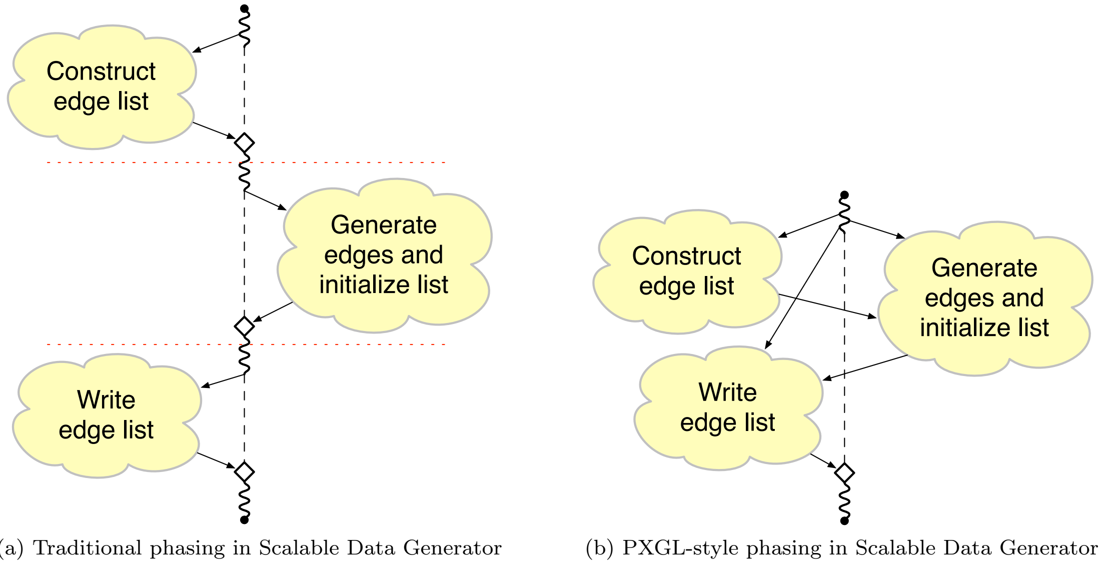
Definition of Scalable Data Generator example

```
1  arbitrary_distribution<gid_type, range> et_dist(localities);
2
3  container_type edge_tuples;
4  edge_tuples.create();
5  edge_tuples.construct(et_dist);
6
7  apply<generate_rmat_data_action>(here, scale, edge_tuples);
8  blocking_for_all<
9      container_type, write_data_action
10 >(edge_tuples, filename);
```

Figure 5.3. Main thread definition for Scalable Data Generator example.

The only explicit synchronization in the entire SDG process is the use of the blocking **forall**[1]. All other necessary synchronization is implicit in the use of the distributed edge tuples vector. The data generator is able to run in parallel with the construction of the vector because any attempt to initialize the vector before it is ready will simply suspend until the vector finishes construction. Likewise, the file writer can be run concurrently because any attempt to use the edge tuple vector will suspend until the vector has completed construction. Figure 5.4a depicts a more traditional phasing of the three components, where construction, initialization, and use are handled sequentially. This PXGL solution is shown in Figure 5.4b.

[1]The current runtime system implementation requires that the programmer guarantee termination of all threads before application shutdown. The blocking **forall** is used to meet this requirement.



(a) Traditional phasing in Scalable Data Generator (b) PXGL-style phasing in Scalable Data Generator

Figure 5.4. High-level dependencies in the Scalable Data Generator.

5.3 Auxiliary Structures

5.3.1 Regions

A region models the set of identifiers for the items in a container. The PXGL defines the Range class representing the integer range $[0, N)$.

Associated Types

index_type
The type of an item identifier.

size_type
The type of a count.

Valid Expressions

region<Range>(n)
Constructs a region of `n` items modeling the range $[0, n)$.

has_index(i)
Returns a Boolean value indicating whether the index `i` is in the region.

size()
Returns the number of items in the region.

5.3.2 Distributions

A distribution comprises a region, set of localities, and partitioning strategy. The region models the set of identifiers for the items in the container. The set of localities defines the coverage over which the container extends. And the partitioning strategy defines the mapping from the indices in the region to localities in the coverage.

The PXGL defines three partitioning strategies: ConstantDistribution, ArbirtraryDistribution, and BlockDistribution. The ConstantDistribution maps all items to a single locality, and works over all regions. The ArbitraryDistribution maps items by index into congruence classes modulo the extent of the distribution. Given coverage $C = \langle L_0, L_1, \ldots, L_m \rangle$, the ArbitraryDistribution strategy simply maps items with index i to locality L_k, where $k = (i \mod |C|)$. This works over regions, such as Range, where the index type supports modulo arithmetic. Finally, the BlockDistribution maps contiguous disjoint segments of the index space to disjoint localities. Each locality will have either $N/|C|$ or $N/|C| + 1$ contiguous items. This works over regions, such as Range, where the index type supports the concept of sequential ordering.

Associated Types

index_type
The type of an index into the distribution.

locality_id_type
The type of identifier for a locality. In HPX, this is the GID type `hpx::naming::id_type`.

locality_ids_type
The type of a collection of locality identifiers.

region_type
The type of region.

Valid Expressions

distribution<P>(l,r)
Creates a new distribution over locality set `l` and region `r`, modeling partitioning strategy `P`.

distribution(l)
Creates a new distribution over locality set `l`. No region is specified.

coverage()
Returns the collection of localities covered by this distribution.

locale_id(i)
Returns the index in the coverage of the locality where the item with index `i` resides.

locale(i)
Returns the identifier of the locality where the item with index `i` resides.

ready()
Suspends the caller until this member of the structure reaches the use phase.

ready_all()
Suspends the caller until all members of the structure reach the use phase.

region()
Returns the region defining the index space for the distribution.

size()
Returns the number of localities covered by this distribution.

5.3.3 Containers

A container is a distributed object with the phased use semantic. Each local member of the distributed container holds a disjoint subset of items. The PXGL defines the Vector container for holding indexed items. The Graph is another type of container and is introduced in Section 5.4.1.

Associated Types

distribution_type
The type of distribution used for this container.

id_type
The type of a GID.

ids_type
The type of a collection of GIDs.

item_type
The type of item stored in the container.

items_type
The type of a collection of items in the container.

size_type
The type of a non-negative count.

Valid Expressions

container(d)
Initiates construction of the container with the distribution `d`.

construct()
Sets the container as constructed.

clear()
Clears the contents of the distributed container and frees all associated memory.

constructed()
Synchronizes on the construction of the container. The caller will suspend if the container has not been constructed, and re-activate when it is ready.

get_distribution()
Returns the distributions of the container.

init(i)
Initializes the container with a collection of items to add to this member.

items()
Returns the collection of items held by this container member.

local_to(i)
Returns the GID of the container member on locality `i`.

ready()
Suspends the caller until this member of the structure reaches the use phase.

ready_all()
Suspends the caller until all members of the structure reach the use phase.

size()
Returns the size of the container.

5.4 Graph Structures

5.4.1 Graphs

A graph is an ordered pair (V, E), where $V = \{u, v, w, \ldots\}$ is the set of vertices and $E = \{(u, v) : u, v \in V\}$ is the set of edges. This is a distributed object with the phased use semantic. A graph is initialized by either providing a (distributed) list of edge tuples, or inserting individual vertices and edges. The PXGL defines the CSRGraph graph representation using a partitioned Compact Sparse-Rows data structure.

Associated Types

Associated types for the distributed graph.

distribution_type
: The type of distribution used for this graph.

edge_descriptor
: The type of an edge identifier.

edge_tuple_type
: The type modeling a tuple of edges.

edge_tuples_type
: The type modeling a collection of edge tuples.

id_type
: The type of a GID.

ids_type
: The type of a collection of GIDs.

size_type
: The type of a non-negative count.

vertex_descriptor
: The type of a vertex identifier.

weight_type
: The type of an edge weight.

Valid Expressions

The directed graph interface.

graph<Representation>(d)
Creates a new graph following distribution `d`. `Representation` defines the internal data structure to use to store the graph data.

add_edge(e)
Adds the edge `e` to the graph.

add_vertex(v)
Adds the vertex `v` to the graph.

clear()
Clears the contents of the distributed graph and frees all associated memory.

construct()
Sets the graph as constructed.

constructed()
Synchronizes on the construction of the graph. The caller will suspend if the graph has not been constructed, and re-activate when it is ready.

edges()
Returns the edge set local to the target member.

get_distribution()
Retrieves the distribution of graph.

init(L)
Initializes the graph from the collection of edges tuples `L`.

local_to(i)
Returns the GID of the graph member on locality `i`.

neighbors()
Returns the neighbor set local to the target member.

order()
Returns the number of vertices in the graph.

ready()
Suspends the caller until this member of the structure reaches the use phase.

ready_all()
Suspends the caller until all members of the structure reach the use phase.

size()
Returns the number of edges in the graph.

vertices()
Returns the vertex set local to the target member.

5.4.2 Property Maps

A property map defines a mapping of keys to values. This is used in the context of graph processing to associate values with the structural elements of the graph; e.g., the vertices and edges. This is a distributed object with the phased use semantic. The PXGL defines the PropertyMap container that supports vertex and edge attributes.

Associated Types

distribution_type
The type of distribution used for this property map.

id_type
The type of a GID.

ids_type
The type of a collection of GIDs.

key_type
The type of key, either vertex_descriptor or edge_descriptor.

size_type
The type of a non-negative count.

value_type
The type of value stored in the map.

values_type
The type of a collection of values in the map.

Valid Expressions

PropertyMap<Key,Value>(d)
Initiates construction of the map with the distribution `d`.

clear()
Clears the contents of the distributed map and frees all associated memory.

construct()
Sets the property map as constructed.

constructed()
Synchronizes on the construction of the property map. The caller will suspend if the property map has not been constructed, and re-activate when it is ready.

get_distribution()
Returns the distributions of the property map.

init(i)
Initializes the map with a collection of items to add to this member.

item(i)
Returns a reference to the item held by this map with index `i`.

local_to(i)
Returns the GID of the property map member on locality `i`.

ready()
Suspends the caller until this member of the structure reaches the use phase.

ready_all()
Suspends the caller until all members of the structure reach the use phase.

size()
Returns the size of the distributed map.

5.5 Data-Directed Constructs

5.5.1 For-all

A **forall** spawns an action for each member in a distributed object. In the case of a container, the action can be applied to each local item in parallel. A **forall** has the general form:

```
for_all([<object>]+, <action> [, <arg>]*);
```

If invoked with more than one distributed *object*, each object must have aligned distributions. The *action* is passed the GID of each object along with the argument list. The PXGL defines the for_all and blocking_for_all methods. The for_all is asynchronous, returning control to the caller immediately after spawning each action. The blocking_for_all is synchronous on the spawned actions, suspending the caller until all actions have completed.

5.5.2 Map

The **map** construct takes one container (the source) and builds a new container (the target). A specified action is applied to each item in the source and the result is inserted into the corresponding index in the target. A **map** has the general form:

```
map(<target>, <source>, <action> [, <arg>]*);
```

The *target* and *source* must have compatible distributions. The PXGL defines the map and blocking_map methods. The map is asynchronous, returning control to the caller immediately after spawning each action. The blocking_map is synchronous on the target structure, suspending the caller until the target completes the initialization phase and is ready for use.

5.5.3 Reduce

The **reduce** construct performs a reduction over items in a given container with a specified operator. This is an extension of the **forall** that uses an additional operator argument to reduce return values. A **reduce** has the general form:

```
reduce([<object>]+, <operator>, <action>, <value> [, <arg>]*);
```

The *operator* is used to reduce a final result value, which is stored in *value*. The value should be an LCO, such as a dataflow-variable, to provide synchronization. The PXGL defines the reduce method. The reduce is asynchronous, returning control to the caller immediately after spawning each action.

Chapter 6

Experiments

This chapter tests the semantics of parallelism postulated in the ParalleX model for graph problems to dynamically expose near fine grain parallelism for adaptive scheduling to improve efficiency and scalability. The PXGL provides the programming framework for the implementation of a parallel graph benchmark suite. Each application kernel is evaluated through consideration of the source code and ability to generate fine-grain actions and execution patterns that match the graph data structure.

The Scalable Synthetic Compact Applications (SSCA) benchmark suite [15, 16] was defined as part of the DARPA High Productivity Computing Systems (HPCS) project. The goal was to define benchmarks for different application classes, each one comprised of number of *kernels*. The second benchmark in this suite, SSCA2, is concerned with the analysis of sparse graph structures. The final revision of the SSCA2 benchmark was in 2007. In 2009, the Scalable Graph Analysis Benchmark (SGAB) was introduced as a revision of the SSCA2.

The specification defines four kernels that operate over a static, weighted, directed graph. The first kernel constructs the graph from a list of "`source target weight`" edge tuples read in from a given file. The second kernel traverses the edge set of the graph to build a set of edges with maximal weight. The third kernel uses the edges from the previous kernel to find and extract subgraphs. And, the fourth kernel computes the betweenness centrality measure of a subset of the vertices in the graph. This set of representative graph-based application kernels are used to stress a broad set of application characteristics.

The first kernel, Kernel 1, involves generating a new graph data structure from given input data. The input data format is a sequence of edge tuples. Each edge tuple includes an identifier for the source vertex, an identifier for the target vertex, and a value for the edge

weight. The identifiers and the value are all of integer type. The form of the solution to the problem is a new directed graph with integer edge weightings.

The goal of any approach to this problem is to quickly build a graph data structure that is useful across the three subsequent kernels. Unfortunately, each of the kernels will stress different aspects of graph processing, and, as the specification suggests, there is likely to be no optimal data structure across all three kernels. Furthermore, the specification requires that the graph data structure is not modified before or during any of the additional kernels.

The benchmark specification provides the following guidance concerning preprocessing of the input data and the generation of the graph. The source of the input data may be a file loaded from secondary storage, or a graph generator process, which generates the data at execution time, given some set of parameters. Either way, the kernel implementer has the option to first load all of the data into an in-memory model before beginning the timed portion of the kernel. When building out the graph data structure, the kernel implementer is also permitted to ignore input data that would create self-loops, multi-edges, or isolated vertices. Finally, the specification states that there is no concurrent execution of subsequent kernels over the graph, and the implementer is free to allocate additional structures in the graph for handling marking and/or locking of the graph data structure.

The implementation of Kernel 1 (Code listing 6.1) begins by defining an arbitrary distribution[1] for the graph (Lines 1 and 2). The coverage is set so that the graph structure will span all available localities. Next, the graph is declared (Lines 4 and 5). This is a synchronous action that creates a named entry in the AGAS for the graph.

The graph is constructed following the procedure outlined in Section 5.1.1 (Line 6). Finally, the graph is initialized from a distributed edge list following the procedure outlined in Section 5.1.2 (Line 7). Both construction and initialization are asynchronous actions, so the main thread completes immediately after the call to `graph_type::init()` returns.

[1]This distribution type is used because of the number of vertices and edges are not known beforehand. If that information were available, a block distribution would have been preferable.

```
Graph Construction
1  arbitrary_distribution<gid_type, range>
2      dist(my_proc.localities());
3
4  graph_type graph;
5  graph.create(here);
6  graph.construct(dist);
7
8  graph.init(edge_tuples);
```

Figure 6.1. Graph construction.

The execution pattern for this process on a single localitity is given in Figure 6.2. The shaded portions represent the construction and initialization phases for the graph structure. It can be seen that the construction and initialization of the graph are concurrent, but self-synchronized. This guarantees that the construction completes before the initialization begins. In fact, ordering of Lines 7 and 8 in the code listing can be swapped with no effect on the execution.

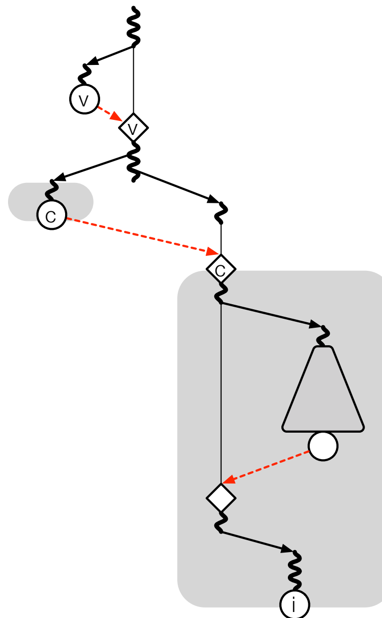


Figure 6.2. Illustration of SGAB Kernel 1.

The following sections cover the three remaining kernels. The design of each algorithm is discussed in the context of the benchmark specification and requirements. The PXGL implementation of the each kernel is then presented. Results of preliminary analysis are also included to both verify the achievement of fine-grain data-directed execution and to assess immediate opportunities for optimiation in this model.

6.1 Large Set Classification

Large set classification (Kernel 2) involves finding the maximum edge weight in a graph and collecting the set of edges which have that weight. The input data is the graph generated by Kernel 1. The form of the solution to the problem is a new set of edges. This so-called *large-set* will be used as input for Kernel 3.

The goal of any approach to this problem is to quickly identify the set of edges in the graph that have maximum weight. This requires identifying the maximum edge weight across the graph and generating the collection of edges that have that weight. The specification does not explicitly state how this large-set should be represented; i.e., as a list of edge tuples or simply a set of edge identifiers. The use of the large-set in Kernel 3 implies that the requirements for that kernel should be considered when deciding how to build this data structure.

This kernel involves processing the edge list of the graph. The process of finding the maximum edge weight and collecting the set of corresponding edges is naturally parallelizable. A conventional divide-and-conquer approach is appropriate for this kernel, as local maxima can be found and edge sets formed in parallel, and merged to form the final collection of edges. The approach uses the meta-data associated with the distribution of the graph to partition the process across each locality. The local workload is again partitioned across a collection of local actions. This local partitioning follows a simple block partitioning heuristic that is controlled by a runtime parameter specifying the number of local partitions.

In the case of a multi-locality run, a new local control object is defined for controlling the distributed reduction. The so-called *have-max* LCO is designed as a coordination point for determining the global maximum value from a set of local maxima. The LCO is initialized with the total number of participants N. When a participant determines its local maximum, it signals the LCO with that value and suspends while waiting for the return value (the global maximum). When the LCO is signaled, the new value is used to update an accumulated maximum. If this is not the Nth signal, the future-value (return value) is not set. When the Nth signal arrives, the global maximum in known, and all pending future-values are set, effectively resuming all suspended participants. The have-max LCO returns a Boolean value to each participant indicating whether or not their local value is the global maximum.

The kernel completes after the global maximum is determined. Each set of local edges with maximum weight are then added to the large-set. This operation is done in parallel for two reasons. First, local insertion of the data does not require the expensive operation of moving a collection of edges between localities. Second, by keeping edges local to where they were found, the large-set is aligned with the graph, which preserves locality of data. Maintaining this alignment between the graph and the large-set will be important in Kernel 3.

6.1.1 Implementation

The main thread of the large-set classification process is defined in Code listing 6.3. This begins by constructing a new container of edge tuples (Lines 1–3). The distribution of the graph is used to align the container with the graph. This provides the large-set with the same coverage over the set of localities and the same partitioning strategy as the graph. This property is critical for designing a process that can effectively utilize locality information to minimize inter-locality actions.

The have-max local control object is also constructed with the distribution of the graph (Lines 5–7). This provides it with the extent of the search, which is also the number of participants in the distributed reduction. Note that in this case, the construction of the

```
Large Set Classification
1  container_type large_set;
2  large_set.create(here);
3  large_set.construct(graph.get_distribution());
4
5  have_max_type is_max;
6  is_max.create(here);
7  is_max.construct(large_set.get_distribution());
8
9  for_all<
10     graph_type, container_type, filter_edges_action
11 >(graph, large_set, is_max);
```

Figure 6.3. Large set classification action definition.

LCO includes initialization. The final step of the process is to asynchronously spawn the `filter_edges_action`s over the graph to initialize the large-set. A **forall** is used to distribute the action (Lines 9–11). The main thread completes after the asynchronous **forall**, though the full large-set classification process is logically finished when the large-set is initialized and ready for use.

Code listing 6.4 defines the `filter_edges_action`. An instance of this action is executed on each locality covered by the graph and large-set. The local portion of the edge set is accessed though the local graph member (Line 6). Since the action is aligned with the graph, it can use that locality information to retrieve a reference to the local edges, instead of a full copy. Note that this call uses the graph structure, so if the graph has not finished initialization, this action will suspend until the graph is ready.

In order to increase the parallelism and reduce the granularity of the search space, another **forall** is used to partition the local edge list. This time, a blocking **forall** is used to wait for all of the searches to complete. The `filter_edges_part_action`, defined in Code listing 6.5, simply iterates over a contiguous block of the edge list and updates the local maximum and edge set. Once completed, the local maximum is known and the have-max LCO is triggered (Line 14). This initiates the distributed coordination procedure described

```
Filter Edges
void filter_edges(
    graph_type graph,
    container_type large_set,
    have_max_type is_max)
{
  graph_type::edge_tuples_type edges = graph.edges();

  size_type max_weight = 0;
  graph_type::edge_tuples_type local_edges;
  blocking_for_all<
      filter_edges_part_action
  >(edges, &local_edges, &max_weight);

  if (is_max.signal(max_weight))
  {
    local_edges.clear();
  }

  large_set.init(local_edges);
}
```

Figure 6.4. Edge filter action definition.

previously, and the thread may suspend. Once the signal have-max LCO is set by the final signal, any suspended thread is resumed and the signal call releases. The local edge set is discarded if that participant did not find the global maximum. Finally, all large-set members are initialized with the local set of edges with the maximum weight (Line 19). The local initialization of the large-set is an asynchronous call entering this large-set member in the coordinated initialization procedure described in Section 5.1.1. The `filter_edges_action` completes immediately after call to initialize, having moved the continuation of action forward to the large-set initialization process. Figure 6.6 illustrates the entire process for a single-locality execution.

6.1.2 Analysis and Results

The design of the algorithm and implementation of Kernel 2 demonstrates the ability to use conventional data-parallel programming techniques with the PXGL. Parallelism is exploited at both the system node and processing core levels. The distributed vector data structure and the parallel **forall** construct enabled partitioning a set of actions over multiple locality domains. Execution was further subdivided on each locality by a runtime variable parameter controlling the task granularity. Continuation migration in the algorithm design is demonstrated by propagating the work of filtering the edge list out and performing the initialization of the large-set from the leaves of the execution tree.

Constraint-based synchronization was demonstrated by both the phased use semantics of the global data structures and the custom have-max local control object. The phased use semantics guaranteeing correct execution was maintained transparent to the runtime allocation and execution of individual tasks. The use of an explicit global synchronization point to reach consensus on the maximum weight value provided coordination for independent actions through a high-level abstraction. This abstraction relieves the burden on the programmer for implementing the control logic for the application. Also, optimization of the underlying synchronization pattern is possible. The current implementation only uses a flat

```
                      Filter Edges Partition
void filter_edges_part(
    graph_type::edge_tuples_type &edges,
    size_type start, size_type stop,
    graph_type::edge_tuples_type &local_edges,
    size_type &max_weights)
{
  graph_type::edge_tuples_type part_edges;
  size_type part_max = 0;
  for (size_type i  = start; i < stop; i++)
  {
    graph_type::edge_tuple_type e = edges[i];

    if (e > part_max)
    {
      part_edges.clear();
      part_edges.push_back(e);
      part_max = e.weight();
    }
    else if (e.weight() == part_max)
    {
      part_edges.push_back(e);
    }
  }

  {
    scoped_feb l(use_feb);

    if (max_weight < part_max)
    {
      max_weight = part_max;

      // Replace current set of local edges with this part
    }
    else if (max_weight == part_max)
    {
      // Add this part to set of local edges
    }
  }
}
```

Figure 6.5. Filter edges partition action definition.

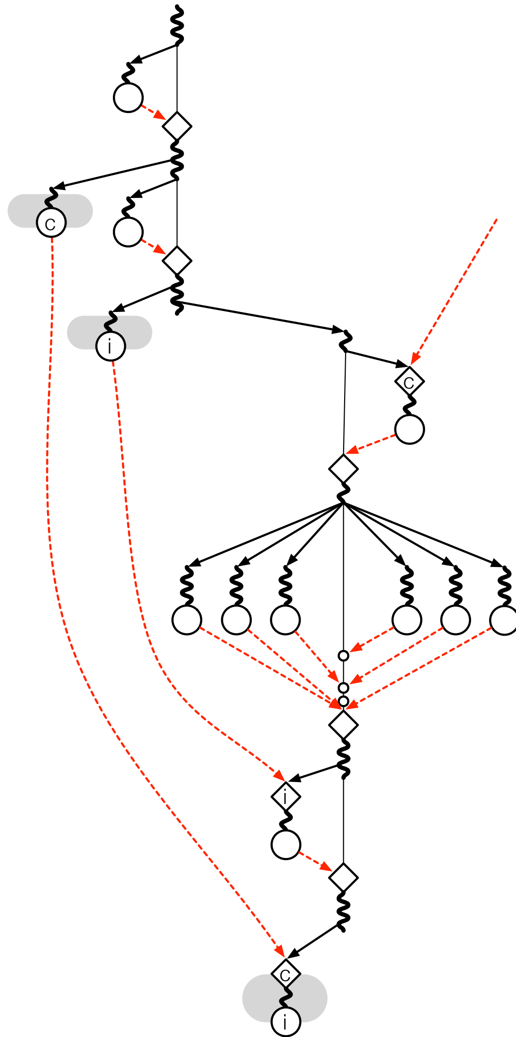


Figure 6.6. Illustration of SGAB Kernel 2 on a single locality.

tree, or broadcast style. Future version for large numbers of localities could employ better synchronization structures, such as the d-ary tree used in the initialization process.

Initial strong scaling results are now presented for this approach. The total number of cores is varied from 1 to 32. The scales under consideration range from 19 to 23, which is 4 million edges to 67 million edges[2]. All scaling figures show total running time of the kernel as a function of operating system thread count. Each OS thread corresponds to an HPX work queue associated with a core on the system.

The first set of figures shows the effect on execution time and scaling for different numbers of partitions. As the number of partitions is increased, the number of parallel tasks increases and the workload per task decreases. No scaling is achieved when the workload is not partitioned (Figure 6.7). Figures 6.8–6.14 show strong scaling is achieved as the number of partitions increases. A different view of scaling is given in Figures 6.15–6.19, which show execution time for individual data sets. This allows for comparing the effect of varying the number of partitions for a particular data set. It is again clear that execution time decreases as parallelism (in terms of number of partitions) increases.

6.2 Subgraph Extraction

Subgraph extraction (Kernel 3) involves parallel identification and generation of subgraphs. The input data is the graph generated by Kernel 1, the edge collection generated by Kernel 2, and a given *depth* value used to bound the diameter of a subgraph. The depth parameter is a positive integer and the specification suggests a setting of 3. The form of the solution to the problem is a new set of subgraphs.

The goal of any approach to this problem is to build a collection of subgraphs. The edges found in Kernel 2 are used as seeds for subgraphs. For each edge, a subgraph is found which comprises all paths originating from that edge and having length equal to the specified depth

[2] For the data sets studied, the number of edges $|E| \approx 8 \cdot 2^{Scale}$.

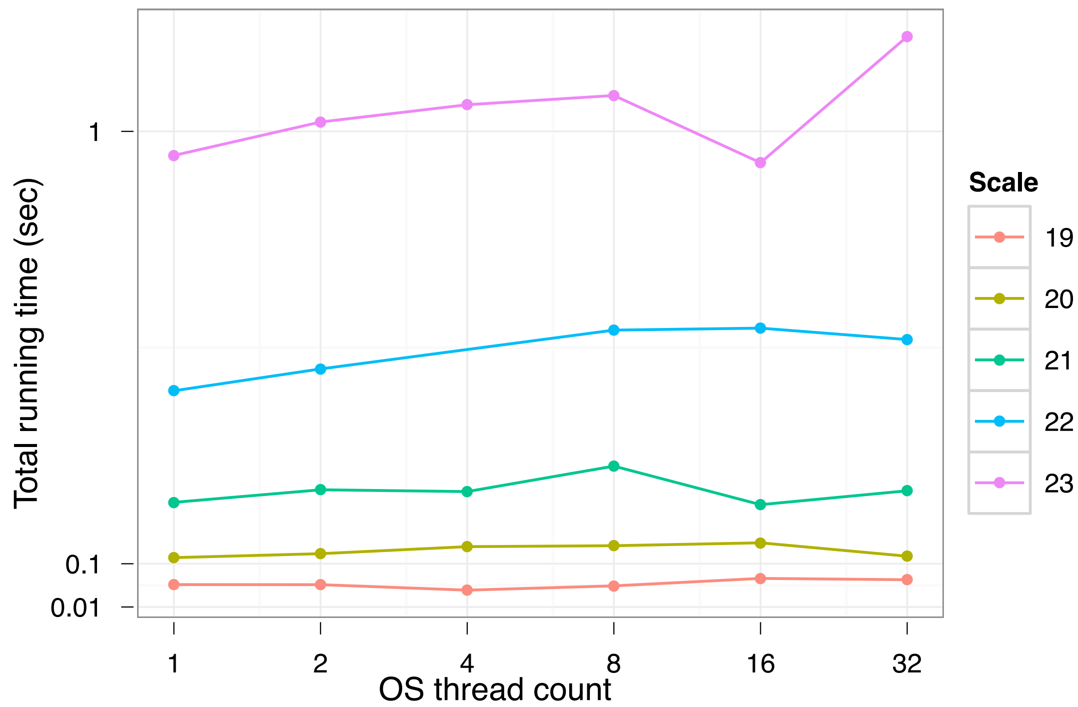


Figure 6.7. Kernel 2 strong scaling with 1 partition.

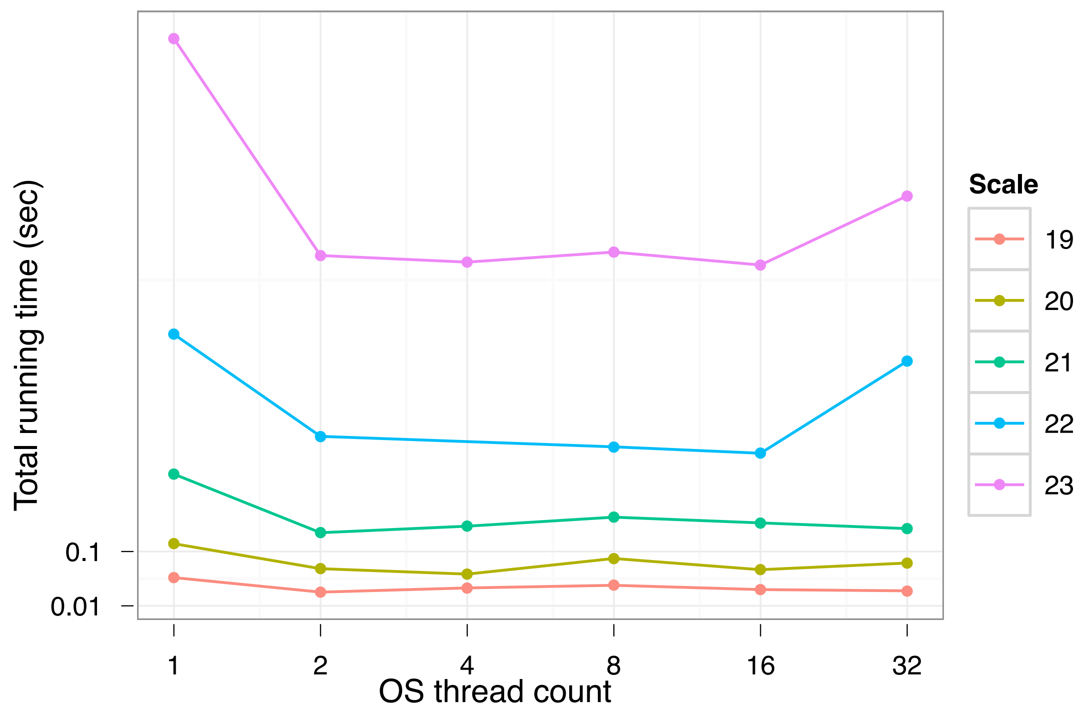


Figure 6.8. Kernel 2 strong scaling with 2 partitions.

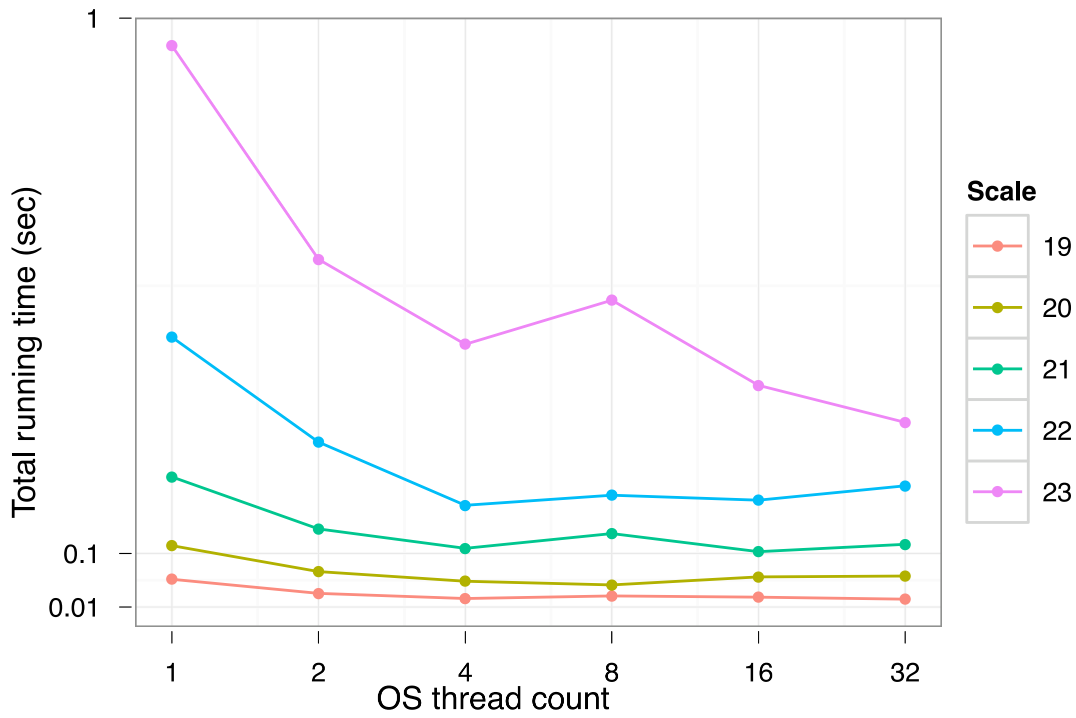


Figure 6.9. Kernel 2 strong scaling with 4 partitions.

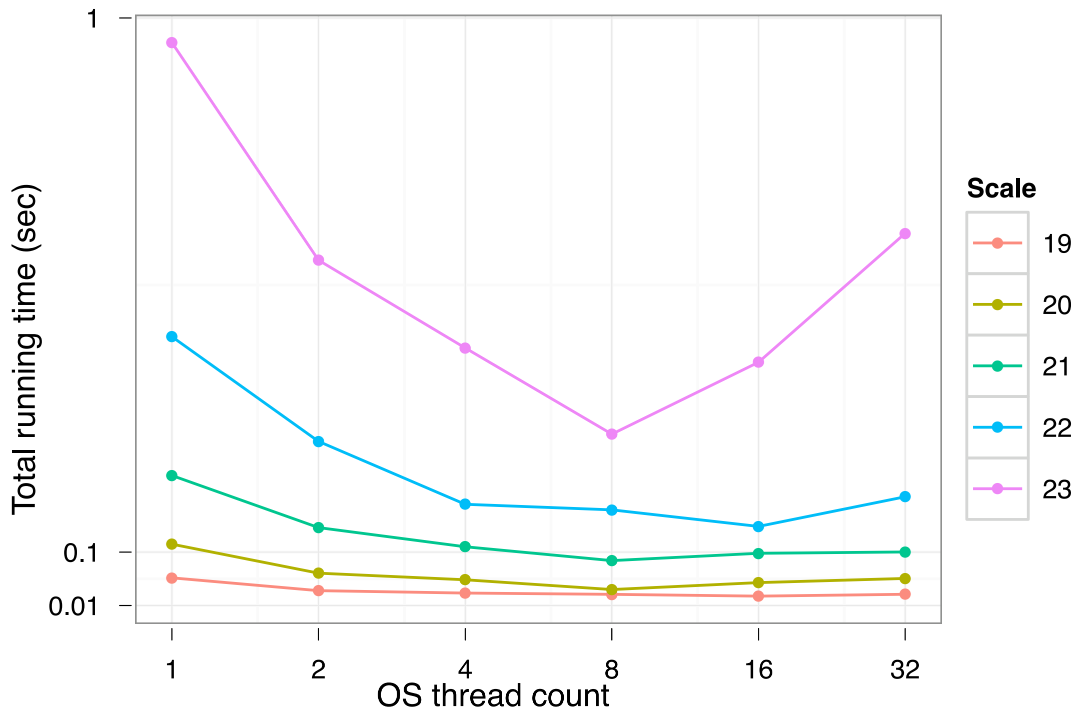


Figure 6.10. Kernel 2 strong scaling with 8 partitions.

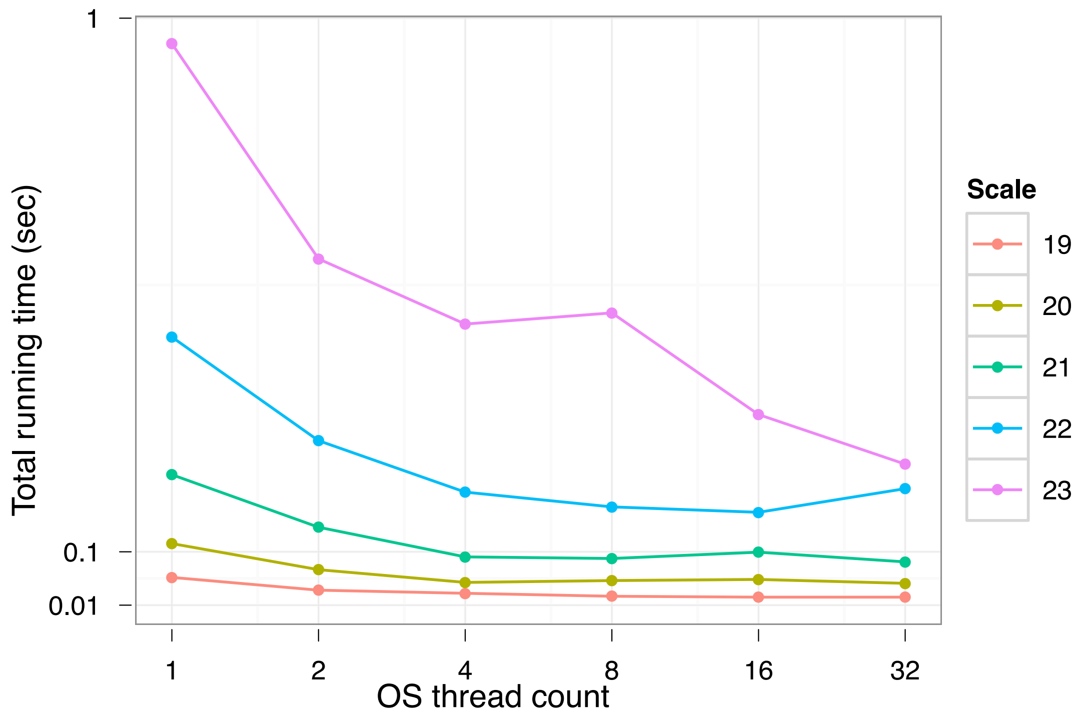


Figure 6.11. Kernel 2 strong scaling with 16 partitions.

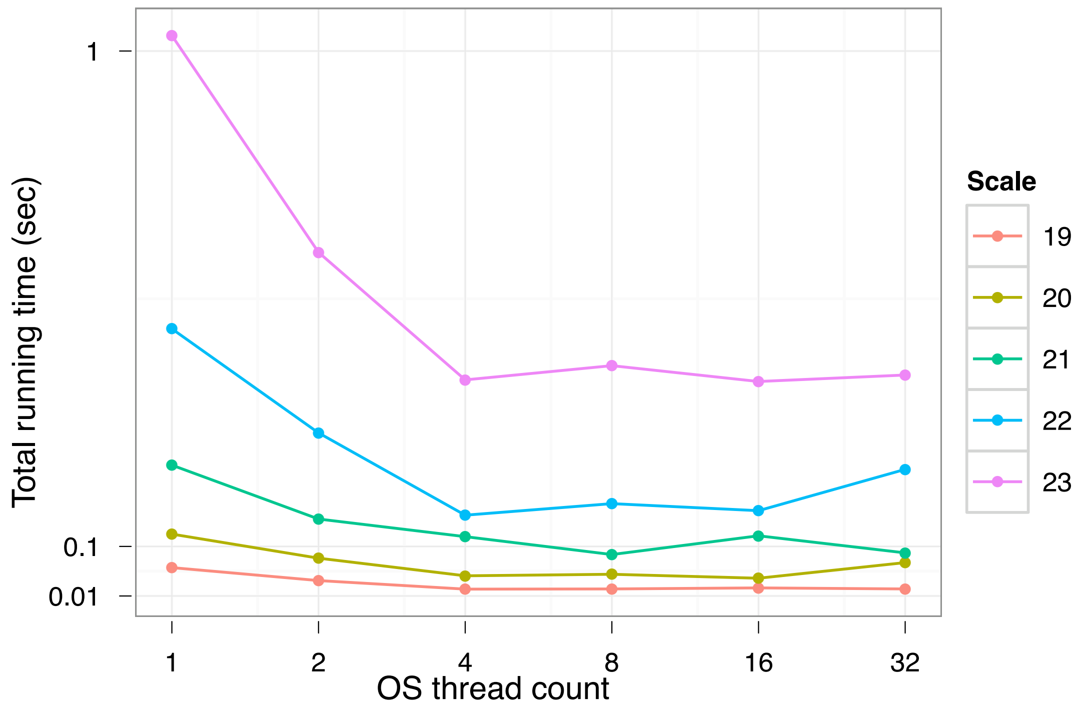


Figure 6.12. Kernel 2 strong scaling with 32 partitions.

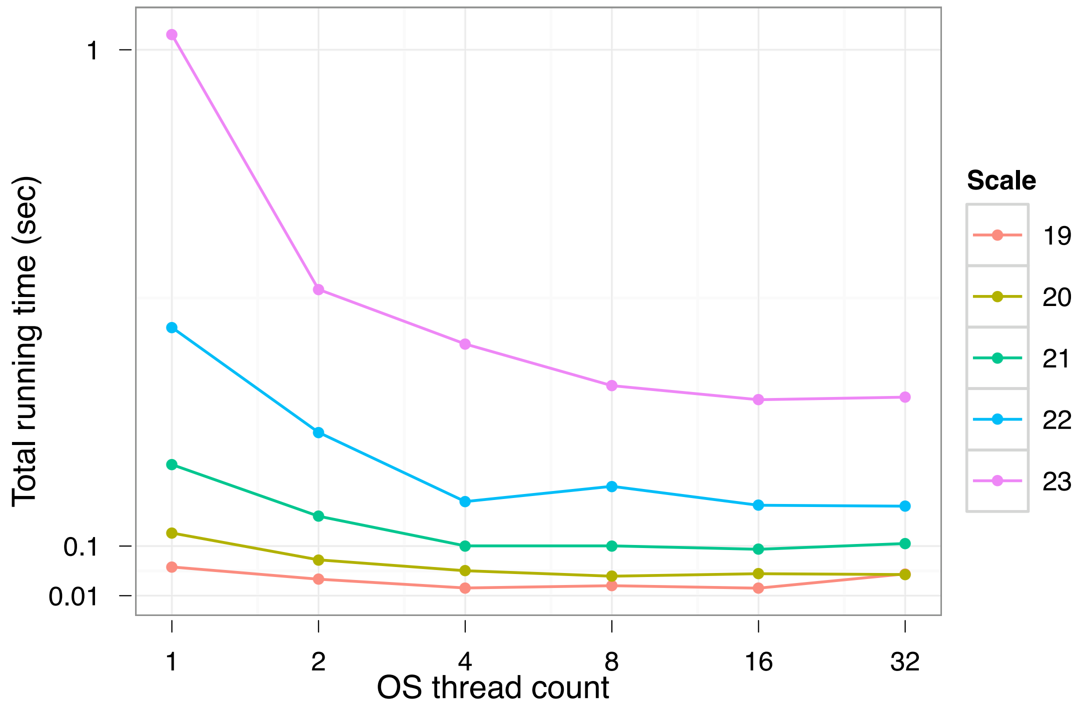


Figure 6.13. Kernel 2 strong scaling with 64 partitions.

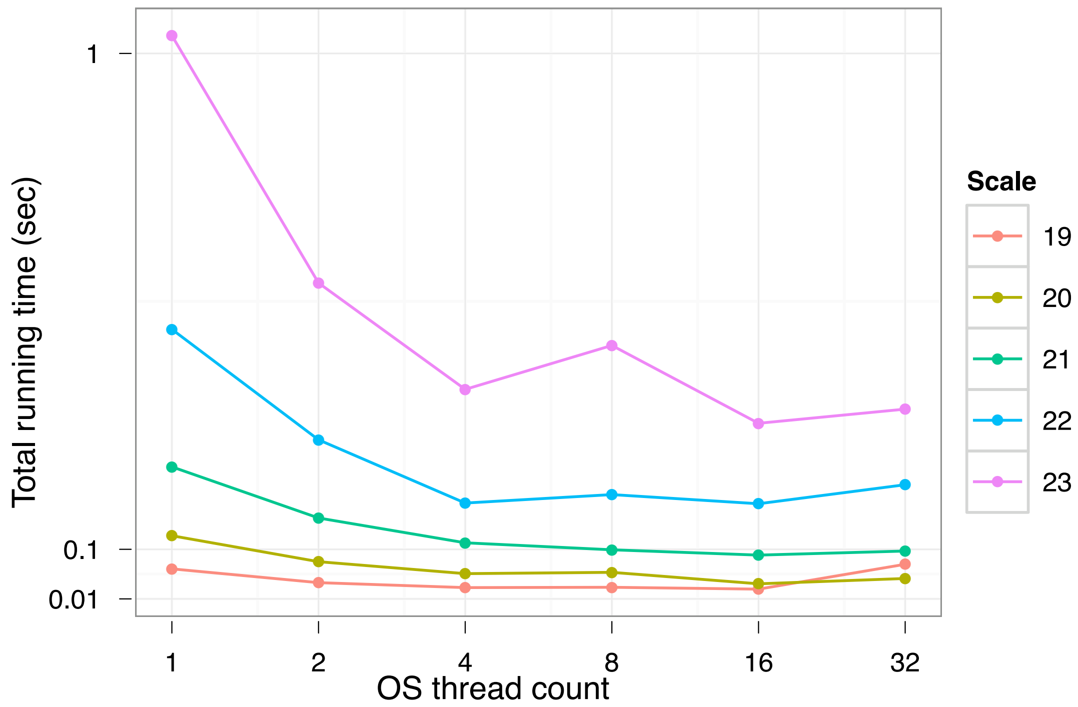


Figure 6.14. Kernel 2 strong scaling with 128 partitions.

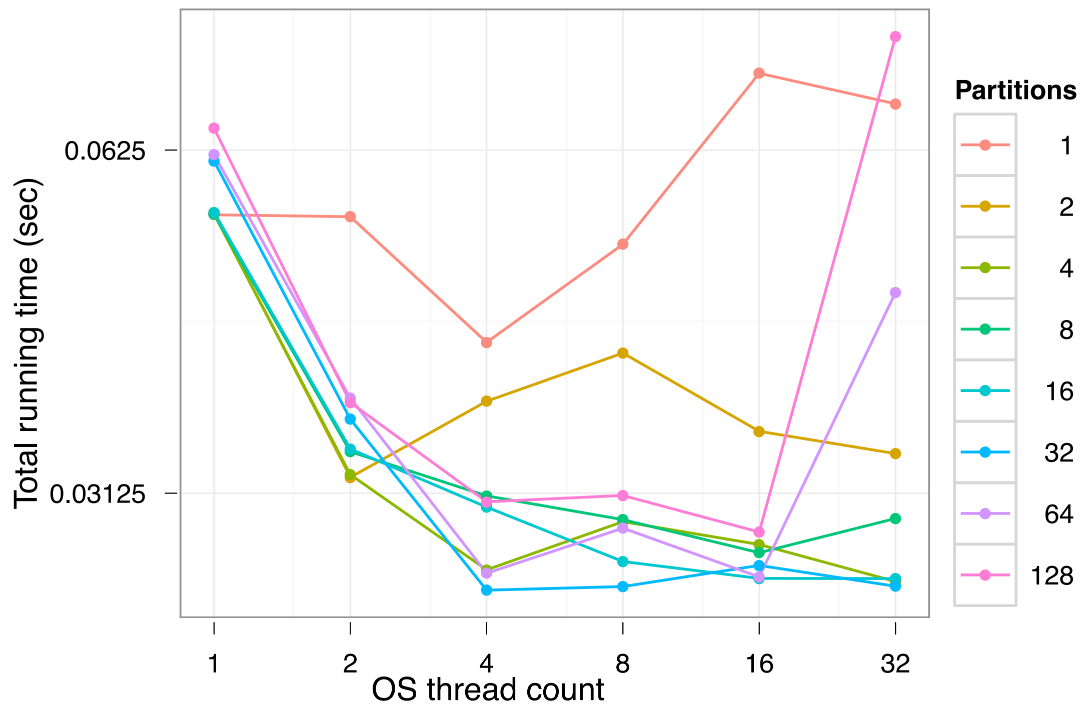


Figure 6.15. Kernel 2 strong scaling for data set scale 19.

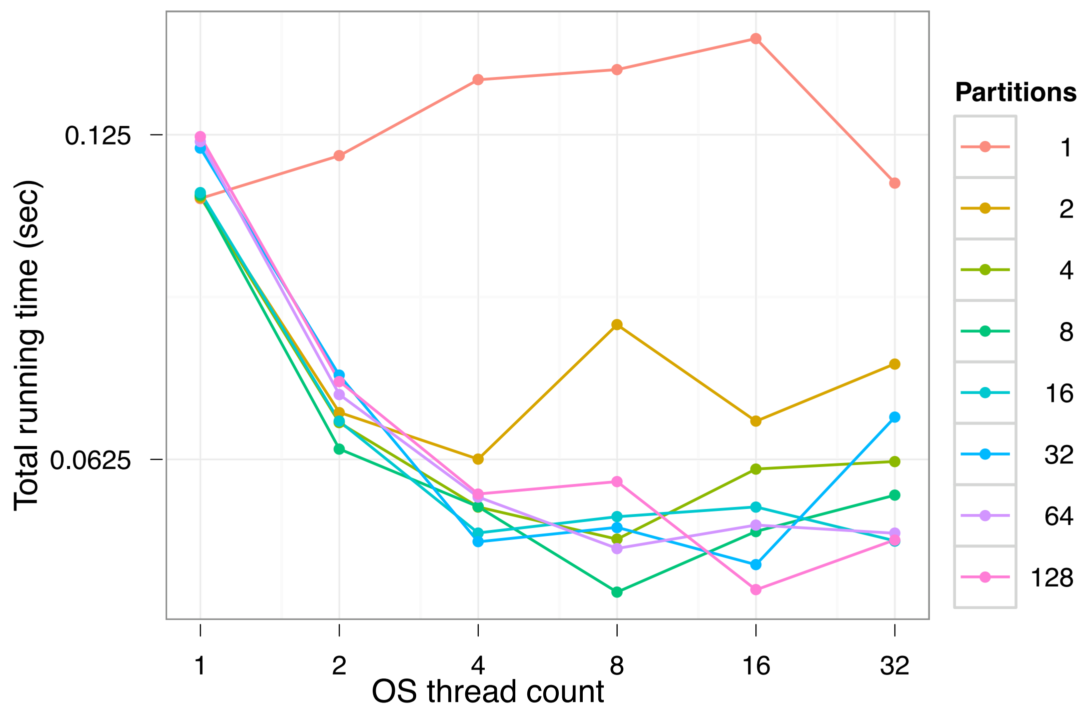


Figure 6.16. Kernel 2 strong scaling for data set scale 20.

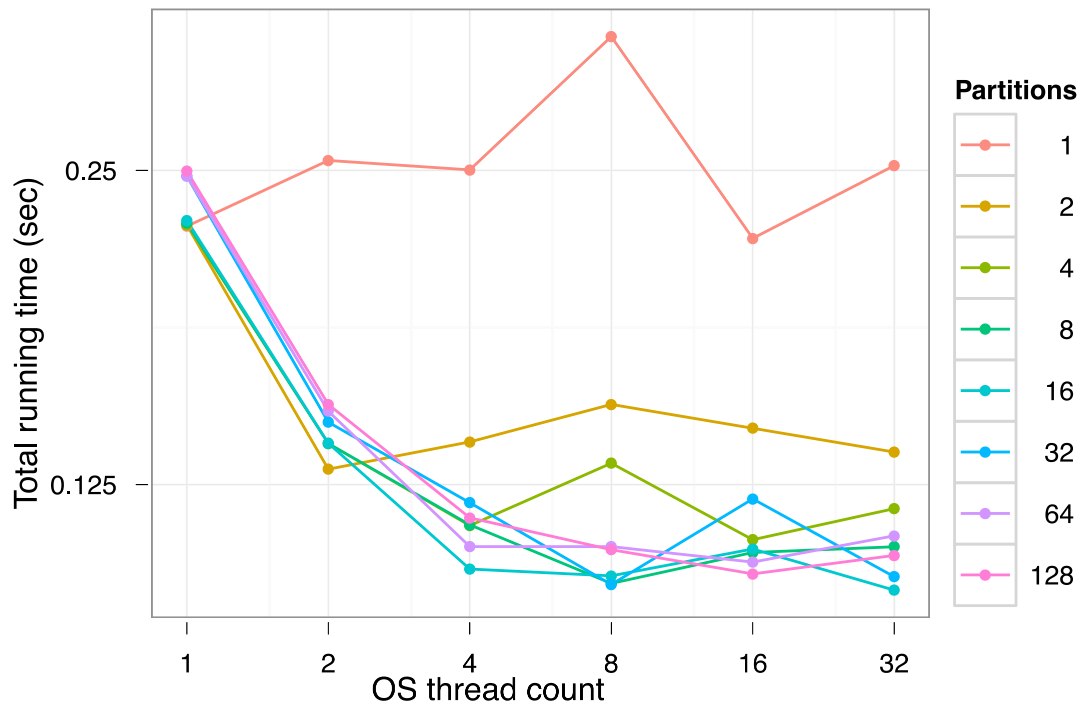


Figure 6.17. Kernel 2 strong scaling for data set scale 21.

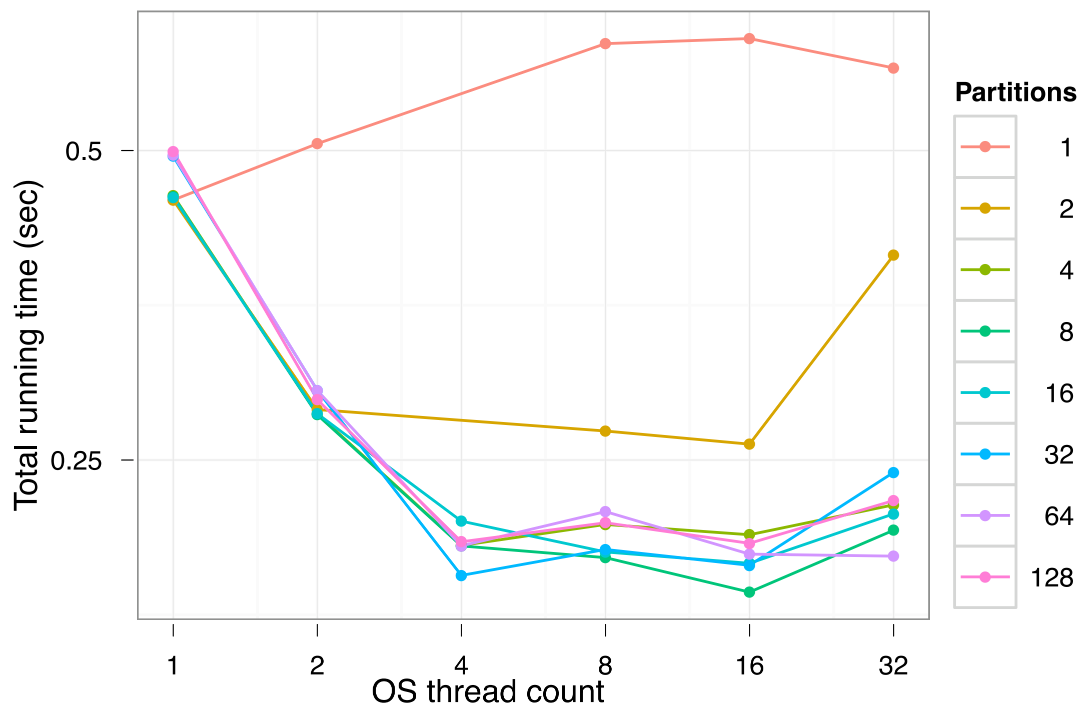


Figure 6.18. Kernel 2 strong scaling for data set scale 22.

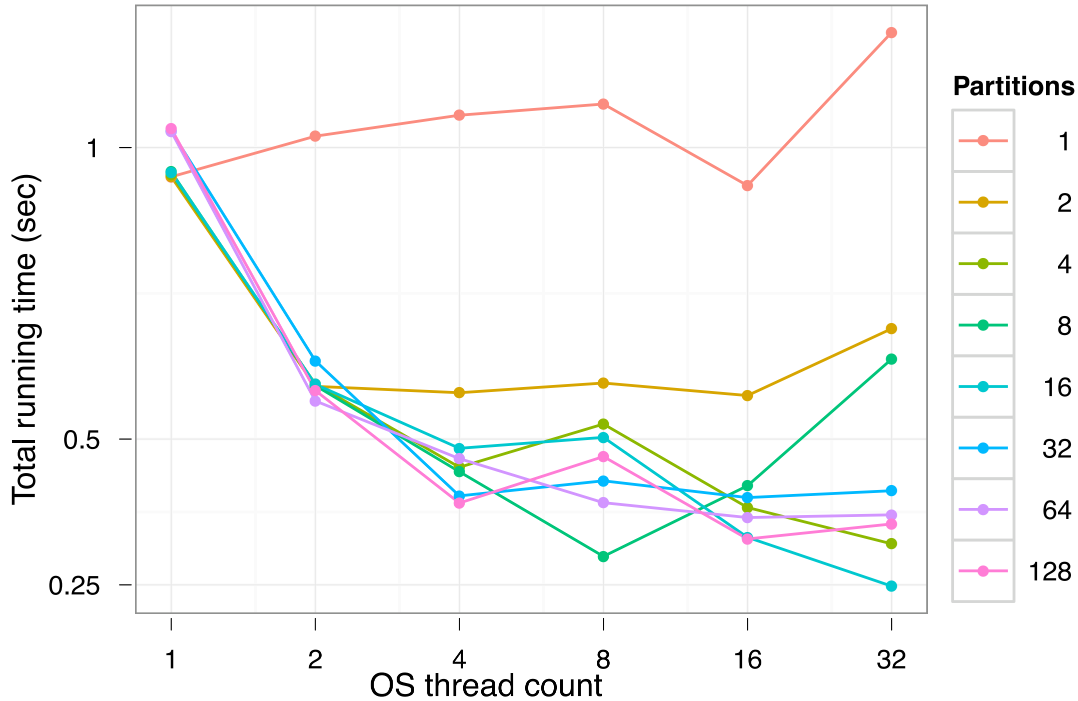


Figure 6.19. Kernel 2 strong scaling for data set scale 23.

value. Unlike the previous two kernels, the subgraphs found by this kernel are not used in subsequent kernels.

The process of this kernel is considerably more involved than that of Kernel 2. The large-set is processed in parallel and, for each edge, a new subgraph is extracted. This is a similar pattern to that already seen in Kernel 2 and in earlier examples. The subgraph extraction uses a parallel search of the graph to find vertices connected to the end point of the edge in the large-set. This process uses the graph meta-data to direct the actions. Also, edges and vertices are inserted into the new subgraph during the traversal.

Each subgraph will be aligned with the original graph. Therefore, the subgraph will have the same distribution as the graph. However, the subset of localities that will actually hold vertices and edges for the subgraph is not known until after the parallel search completes. The coverage of the new subgraph will match the implied coverage of the vertices discovered during the parallel traversal. In the case of a graph covering a large number of localities, the

subgraph could potentially only span a relatively small number of localities. Given the latency associated with coordinated construction and initialization, a lazy construction approach is taken, which builds out the collection of distributed members only as needed.

The data structure used to model the subgraph is a partially dynamic graph data structure. The phased use semantics still apply, but the construction and initialization phases are integrated to support growing the coverage along with the addition of new vertices and edges. Only the initial member is created when the subgraph is first constructed. Insertion of an edge (s, t) in a dynamic graph structure may extend the physical coverage of the structure. If the target vertex t resides on a remote locality, then the action must determine if a sibling exists on that locality. If not, the structure is extended to cover the locality, and an add-vertex operation is invoked to add the vertex. The add-vertex action has a similar effect for vertices mapped to different localities.

Since this initialization proceeds incrementally, a new strategy is needed for finalizing the initialization phase so that the subgraph can be used. The init action is adapted for this purpose to take no arguments and to simply signal that the structure can now be considered as initialized. The only requirement is that there are no pending updates to the subgraph that will be executed after the initialization action. Currently, this must be enforced by the application code. This implementation presented here uses synchronous calls to add-edge and add-vertex to handle this.

The actual creation of each subgraph is controlled by a parallel search over the graph. A parallel search is started from the target vertex t of each edge (s, t) in the large-set. Each search is performed concurrently, and many searches will possibly overlap. For a given search, every path from t is traversed in parallel up to a given depth d. Figure 6.20 shows a single parallel traversal. The directed graph (above) has labeled start vertex s. The parallel traversal (below) begins from s and visits new vertices along all available paths. The dashed arrows indicate a subsequent visit to a vertex, in which case the embedded LCO stops the traversal. Note that this is the data-directed execution pattern described in Section 4.3.

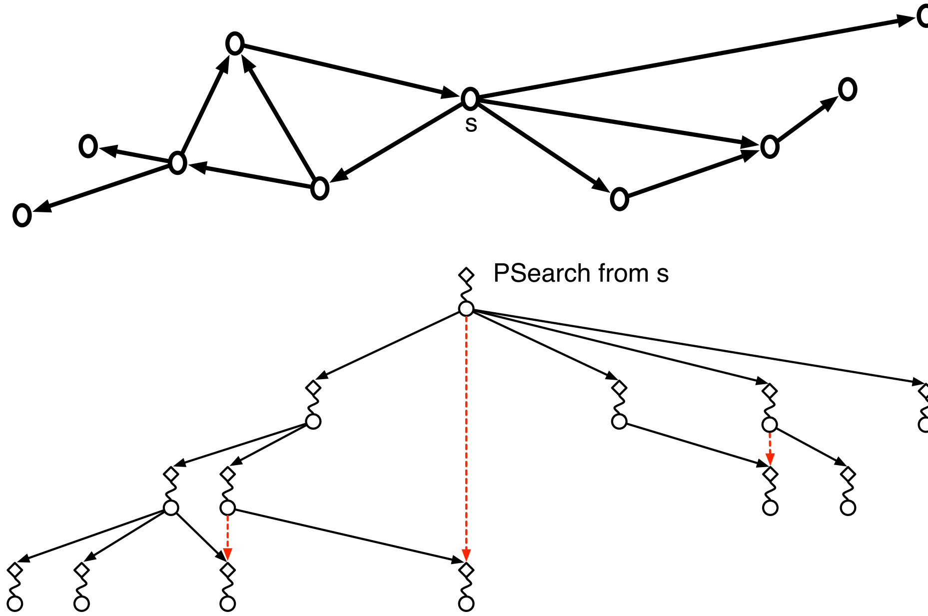


Figure 6.20. Illustration of data-directed PSearch-enabled execution pattern.

A so-called *PSearch* local control object is used to control the traversal of the graph. These LCOs are embedded in the graph structure using an attribute, or property, map. This property map also uses the lazy initialization strategy discussed for the subgraph. The partially dynamic structure is used again for the same reason of forgoing the cost of coordinated initialization over the entire coverage when only a fraction of the coverage might actually be utilized. In this case, the PSearch LCOs are also built in a lazy fashion that matches the traversal of the graph. This is crucial for space considerations when a parallel search of a given depth may only cover 1000 vertices in a graph that contains over 25 million vertices.

The PSearch LCO is extended to insert vertices and edges in an associated subgraph as they are discovered. This utilizes the algorithm visitor pattern to ease the burden on the programmer. The programmer is only responsible for extending the appropriate event points and starting the search. From there the traversal discovers the subgraph and propagates the actions to build the subgraph.

Figure 6.21 illustrates the subgraph extraction process for a single-locality execution. The initial thread spawns asynchronous actions to construct the set of subgraphs and to spawn

the parallel search, then completes. The shaded triangles each represent a distinct parallel traversal of the graph and generation of a new subgraph. The thread that spawns the concurrent traversals waits for each to finish. The entire process logically completes when the collection of subgraphs has been initialized with all of the newly formed subgraphs.

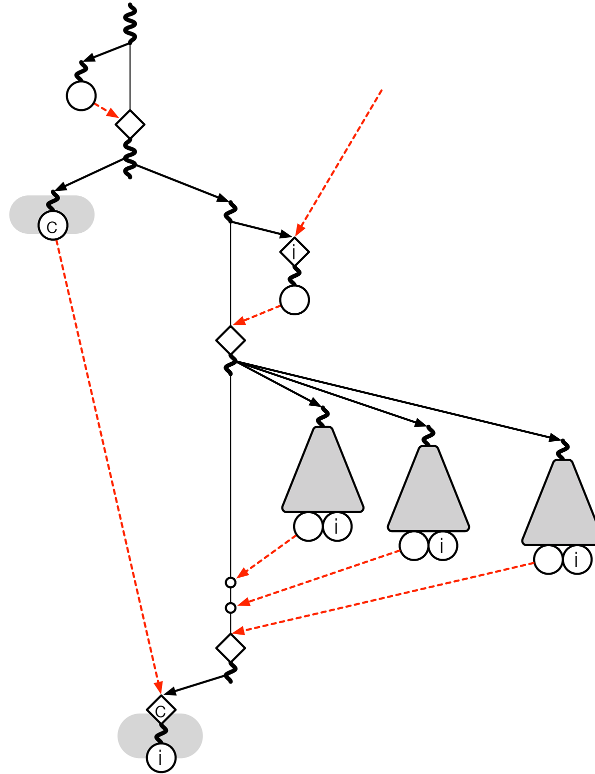


Figure 6.21. Illustration of SGAB Kernel 3 on a single locality.

6.2.1 Implementation

The main thread of the subgraph extraction process is defined in Code listing 6.22. The thread begins by constructing a new container to hold the individual subgraphs (Lines 1–3). This container is aligned with the large-set generated in Kernel 2. This is done because each subgraph is generated from one edge in the large-set. Following the spawn of the asynchronous construct action, a **forall** is used to spawn individual `extract_subgraph_actions` (Lines 5–8). As with the previous examples, the main thread terminates after spawning actions but, logically, the process is finished when the subgraph container has finished initialization.

Kernel 3

```
graph_container_type subgraphs;
subgraphs.create(here);
subgraphs.construct(large_set.get_distribution());

for_all<
    container_type, graph_type, graph_container_type,
    extract_subgraphs_action
>(large_set, graph, subgraphs);
```

Figure 6.22. Kernel 3 main thread action definition.

The `extract_subgraphs_action` is defined in Code listing 6.23. The purpose of this action is to further partition the local portions of the large-set. The blocking **map** construct is used to spawn `find-subgraph-action`s and collect all of the GIDs for each subgraph built from an edge on that locality (Lines 5–7). The local subgraph GIDs are then used to initialize the container of subgraphs (Line 9). Note that a blocking **map** is used because `local_subgraphs` is a standard vector, not a PXGL container with phased use semantics.

Subgraph Extraction

```
void extract_subgraphs(
    container_type large_set,
    graph_type graph,
    graph_container_type subgraphs)
{
  ids_type local_subgraphs;

  container_type::items_type edges = large_set.items();

  blocking_map<
      find_subgraph_action
  >(edges, local_subgraphs, large_set, graph);

  subgraphs.init(local_subgraphs);
}
```

Figure 6.23. Subgraph extraction action definition.

The `find_subgraph_action` is defined in Code listing 6.24. This action is responsible for spawning a parallel search over the graph and subsequent subgraph generation. For each edge (s, t) in the large-set, this action is invoked local to the source vertex s. The action begins by creating a new subgraph locally. The distribution of the graph is supplied to begin the lazy construction/initialization phase introduced earlier (Lines 4–6).

Find Subgraphs

```
1  id_type find_subgraph(
2      graph_type graph,
3      container_type::item_type edge,
4      size_type depth)
5  {
6    graph_type::distribution_type graph_distribution =
7        graph.get_distribution();
8
9    subgraph_type subgraph;
10   subgraph.create(here);
11   subgraph.set_distribution(graph_distribution);
12
13   vmap_type vmap;
14   vmap.create(graph_distribution.locale(edge.target()));
15   vmap.construct(graph_distribution, graph, subgraph);
16
17   subgraph.add_edge(edge);
18
19   vmap.visit(edge.target(), depth);
20
21   vmap.init();
22   subgraph.init();
23
24   return subgraph.get_gid();
25 }
```

Figure 6.24. Subgraph generation action definition.

An embedding of the PSearch local control object is generated for the graph (Lines 8–11). This again follows the lazy initialization semantics introduced earlier. Note that the property map is actually created local to the target of the edge, which might be remote (Line 9). This

is done because the parallel search might not cover the locality where this action is executing. The subgraph was created locally because it will necessarily include the start edge and start vertex, which are local.

After the property map is established, the start edge is added to the subgraph (Line 12). The parallel search is initiated by visiting the target vertex (Line 14). The entire process of the parallel search and corresponding subgraph generation is handled by the embedded PSearch local control object described earlier. The calls to `vmap.init()` and `subgraph.init()` are done to synchronize on completion of the parallel search and subgraph initialization (Lines 16–17). Finally, the GID of the new subgraph is returned (Line 19).

6.2.2 Analysis and Results

This design of the algorithm and implementation of Kernel 3 demonstrates the ability to implement fine-grain data-directed execution using the graph meta-data. Parallelism is exploited at all levels by reducing the granularity of actions to the processing of individual vertices and allowing the execution flow control to be determined by the graph structure. Continuation migration in the algorithm design is demonstrated by migrating the initialization of the subgraph into the extract-subgraphs action, and the migration of the subgraph creation into parallel search.

Constraint-based synchronization is demonstrated by both the phased use semantics of the distributed structures and the embedding of local control objects in the graph data structure. The phased use semantics synchronized the construction, initialization, and use of the large-set and graph structures. The algorithm visitor pattern implemented as a custom local control object shows how a highly parallel algorithm with dynamic flow control can be programmed. In both cases, the responsibility for managing flow control is moved from the application code to the PXGL.

Initial strong scaling results are now presented for this approach. The total number of cores is varied from 1 to 32. The scales under consideration range from 16 to 21, which is 0.5 million edges to 16 million edges. The actual workload for Kernel 3 is best expressed as the number of ParalleX threads executed. This is because of the fine granularity of actions, and the fact that the choice of start edges determines quantity and size of subgraphs extracted. Figure 6.25 shows the total number of ParalleX threads executed for each scale. The important observation is that the total number of ParalleX threads is not directly proportional to the scale of the data set. Also, note that the increase in ParalleX threads with OS thread count is attributed to HPX runtime management. Figure 6.26 shows the average rate of ParalleX threads processed per second. Finally, Figure 6.27 shows the execution time of the kernel. Both of these figures show that the application is able to utilize increased in system resources.

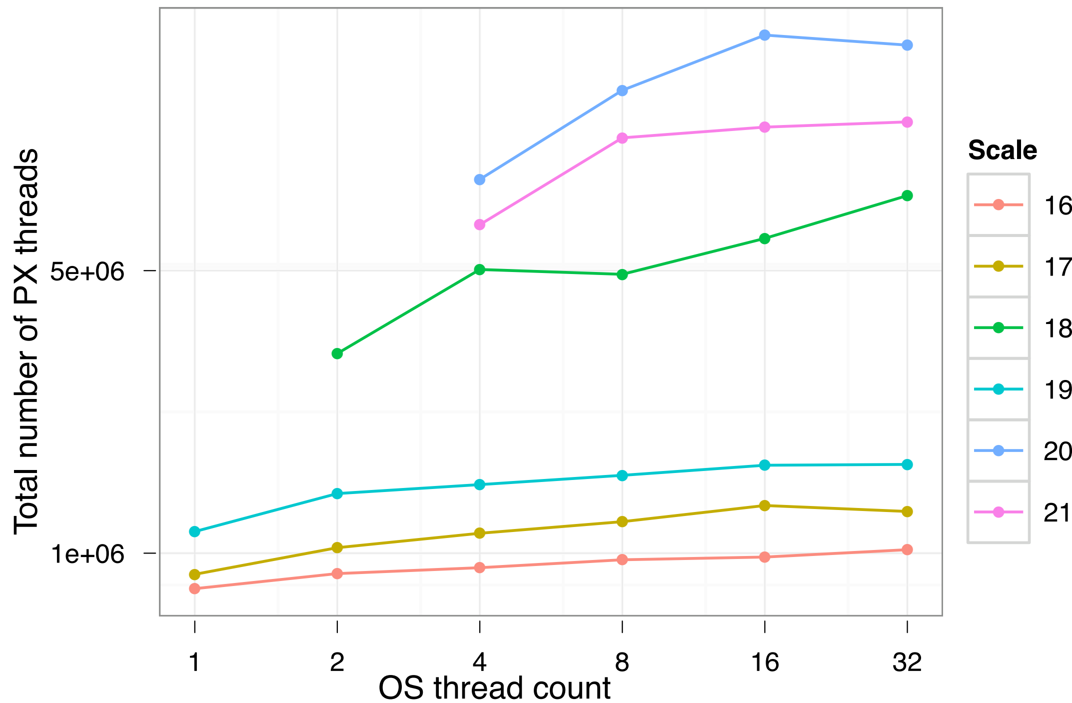


Figure 6.25. Kernel 3 total number of ParalleX threads.

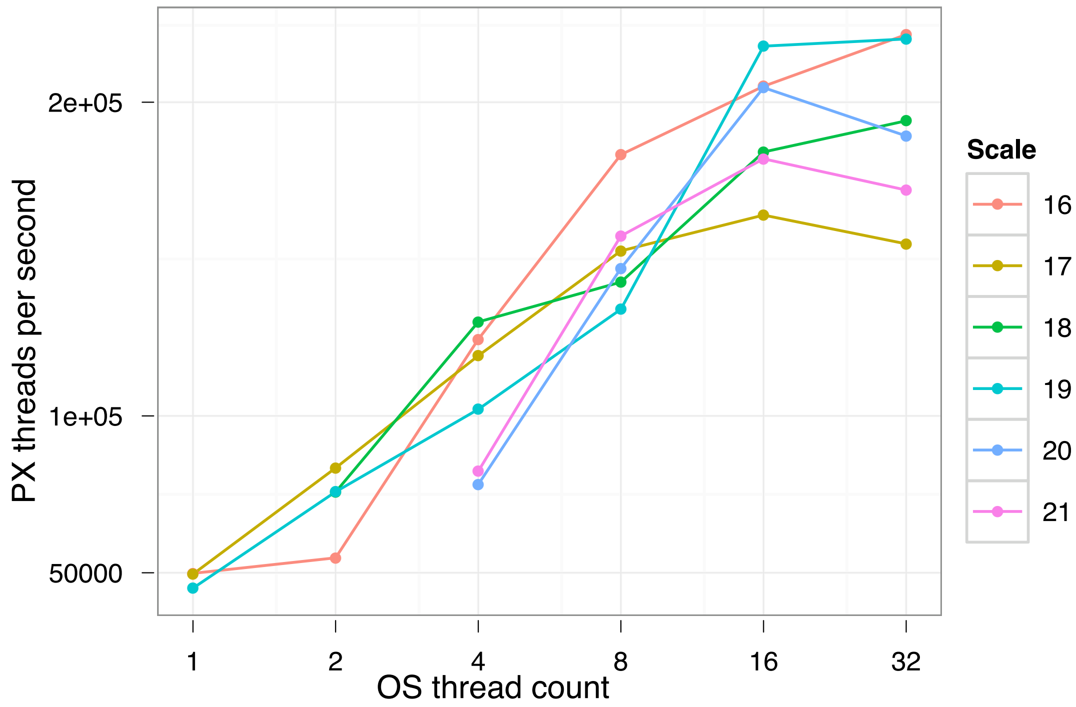


Figure 6.26. Kernel 3 ParalleX threads per second.

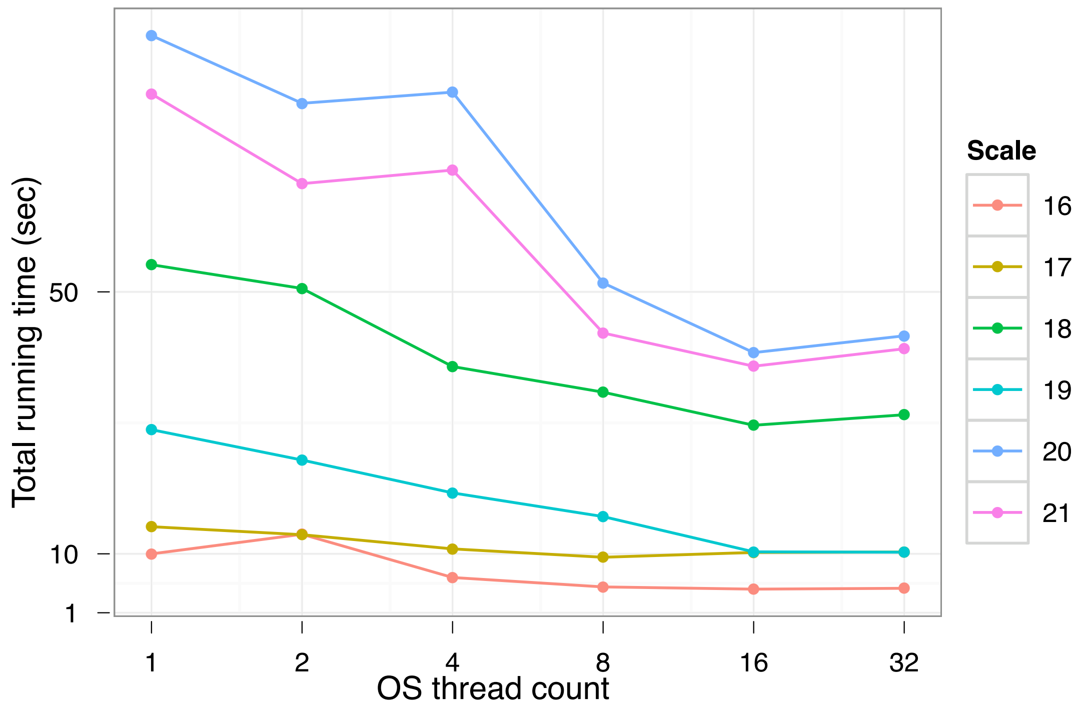


Figure 6.27. Kernel 3 strong scaling run time.

6.3 Computing Betweenness Centrality

Kernel 4 involves calculating a betweenness centrality (BC) score for each vertex in a graph and collecting the set of edges which have the highest score. The input data is the graph generated by Kernel 1 and a specified *K4Approx* value, used to limit the problem size. The *K4Approx* parameter is an integer in $[1, SCALE]$. When this value is less than $SCALE$ a random subset of the graph vertex set, of size $2^{K4Approx}$ is used to provide an approximate solution to the problem. The form of the solution is a scoring of each vertex and a new set of vertices with the highest score. The scoring process works by calculating shortest paths in the graph, with each vertex scored according to the number of shortest paths that pass through it.

The specification for the benchmark suggests a solution formulated by Brandes [63, 64]. The process uses an adapted single source shortest paths (SSSP) algorithm to compute the betweenness centrality scores. The definition of the algorithm from the benchmark specification is reproduced as Algorithm 6.1. The specification authors note two levels of exploitable parallelism: each SSSP computation can proceed in parallel, and fine-grain parallelism within the SSSP can be exploited. The approach taken here extends this to exploit a third level of parallelism by distributing the computation over a distributed memory system.

The approach described here demonstrates the adaptation of this graph algorithm to the methodology outlined in this dissertation. The computation is characterized as a distributed ParalleX process. The execution is still fine-grain data-directed actions, but the process model is used to maintain persistent data structures. The distributed ParalleX process is aligned with the graph G. The bookkeeping data structures σ, d, δ, and P are logically divided across the members of the process. The queue Q and stack S are maintained in the initial member of the process. The BC structure is not included in the process. It is instead implemented as a property map labeling the vertices of the graph. This is done independent of the process because BC represents the solution that will be used by the parent process and other actions in post-processing.

```
input  : graph G = (V, E)
output: array BC[1 ... n]
1  forall the v ∈ V do
2      BC[v] ← 0
3  end
   // Let V_S be a random subset of V of size 2^{K4Approx}
4  forall the s ∈ V_S do
5      S ← empty stack
6      P[w] ← empty list, ∀w ∈ V
7      σ[t] ← 0, d[t] ← −1, ∀t ∈ V
8      σ[s] ← 1, d[s] ← 0
9      Q ← s
10     while Q ≠ ∅ do
11         v ← Q
12         v → S
13         forall the w ∈ neighbors(v) do
14             if d[w] < 0 then
15                 w → Q
16                 d[w] ← d[v] + 1
17             end
18             if d[w] = d[v] + 1 then
19                 σ[w] ← σ[w] + σ[v]
20                 v → P[w]
21             end
22         end
23         δ[v] ← 0, ∀v ∈ V
24         while S ≠ ∅ do
25             w ← S
26             for v ∈ P[w] do
27                 δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w])
28             end
29             if w ≠ s then
30                 BC[w] ← BC[w] + δ[w]
31             end
32         end
33     end
34 end
```

Algorithm 6.1: A betweenness centrality algorithm.

The actual SSSP algorithm is implemented as a set of actions that can be called on the ParalleX process. This is done to support moving work to data. The algorithm defines an expansion phase where new vertices are visited in a breadth-first traversal out from a start vertex. A subsequent contraction phase synthesizes intermediate values up the spanning tree generated by the first phase. In a multi-locality execution of the algorithm, vertices might not be colocated. Therefore these two phases cannot be stated as simply as they are in the original algorithm. Instead, they must be recast as actions that migrate work to the target vertices. The following implementation maps this algorithm into the PXGL.

6.3.1 Implementation

The main thread of the betweenness centrality implementation is defined in Code listing 6.28. Lines 2–6 generate the distributed set of start vertices, `bc_vertices`. This process follows a similar parallel filter pattern as the one used for Kernel 2, and is not discussed further. The relevant information is that the start vertices container is aligned with the graph structure. Following this, the `bc_scores` property map is constructed. This will map the calculated betweenness centrality scores to the vertices in the graph. A blocking **forall** is used to start the individual SSSP processes for each start vertex (Line 13). The remainder of the main action filters the vertices to generate the set of maximum weight (betweenness centrality score) vertices. This again follows a similar pattern as those already discussed and is left out for brevity.

The `score_bc_vertices` action is defined in Code listing 6.29. This action spawns the parallel SSSP computations and corresponds to the **forall** on Line 4 of Algorithm 6.1. The action is invoked once for each member of the `bc_vertices` container. For each vertex associated with the local member, a SSSP process (`bc_sssp_type`) is instantiated (Lines 4, 7, and 8). ParalleX process instantiation follows a similar pattern as the distributed containers such as the vector and graph discussed earlier. This is because the distributed ParalleX process is designed with the same decentralized approach and phased use semantics. The instantia-

Computing Betweenness Centrality

```
1  // Find the set of starting vertices
2  bc_vertices_type bc_vertices;
3  bc_vertices.create(here);
4  bc_vertices.construct(graph.get_distribution());
5
6  for_all<select_bc_vertices_action>(graph, bc_vertices);
7
8  // Calculate the betweenness centrality scores
9  bc_scores_type bc_scores;
10 bc_scores.create(here);
11 bc_scores.construct(bc_vertices.get_distribution());
12
13 blocking_for_all<score_bc_vertices>(graph, bc_vertices, bc_scores);
14
15 // Find set of vertices with the max betweenness centrality score
16 bc_max_vertices_type bc_max_vertices;
17 bc_max_vertices.create(here);
18 bc_max_vertices.construct(bc_vertices.get_distribution());
19
20 have_max_type is_max;
21 is_max.create(here);
22 is_max.construct(bc_max_vertices.get_distribution());
23
24 for_all<find_max_bc_scores_action>(bc_scores, bc_max_vertices, is_max);
```

Figure 6.28. Kernel 4 main thread action definition.

tion takes care of the allocation and initialization of the bookkeeping structures and general setup for the process. Each process is started with the call on Line 10, which supplies the start vertex. Lines 13–17 block the action from completing until all parallel processes have finished.

Score BC Vertices

```
1  void score_bc_vertices(
2      graph_type graph,
3      bc_vertices_type bc_vertices,
4      bc_scores_type bc_scores)
5  {
6    bc_vertices_type::items_type vertices = bc_vertices.items();
7    size_type num_vertices = vertices.size();
8
9    std::vector<bc_sssp_type> bc_sssps(num_vertices);
10   for (size_type i = 0; i < num_vertices; i++)
11   {
12     bc_sssps[i].create(here);
13     bc_sssps[i].instantiate(graph, bc_scores);
14
15     bc_sssps[i](vertices[i]);
16   }
17
18   for (size_type i = 0; i < num_vertices; i++)
19   {
20     bc_sssps[i].ready_all();
21     bc_sssps[i].ended();
22   }
23 }
```

Figure 6.29. Betweenness centrality scoring action definition.

The betweenness centrality process implements the core of the algorithm (Lines 5–33). Code listing 6.30 defines the start action for the process. The call to `ready()` is a standard idiom in the distributed structures in the PXGL. The call is used to prevent the action from executing before the structure is in the use phase (i.e., ready for use). The code follows the algorithm almost explicitly, except for Lines 16 and 30. Line 16 corresponds to the **forall**

in the algorithm that visits all of the neighbors of a vertex. That process is defined across two actions: `expand_source` and `expand_target` (Code listings 6.31 and 6.32, resp.) The two actions are needed to manage the locality of the vertices. The `expand_source` action is invoked with respect to the member of the process that is colocated with the source vertex. Likewise for the `contract_target` action that is invoked with respect to the process member aligned with the target vertex.

The process of visiting neighboring edges to build out the breadth-first traversal of the graph is implemented across the `expand_source` and `expand_target` actions. As stated, the former is invoked on the process member aligned with the source vertex. When executing `expand-source`, the current d and σ values for source are available because this is the process member aligned with source. Next, a blocking **map** is used to visit all neighboring vertices in parallel. The `expand_target` action executes on the process member aligned with the target vertex. This is the action that actually implements the conditional update of $d[w]$ and $P[w]$, corresponding to Lines 14–21 in the original algorithm.

The contraction phase of the algorithm involves propagating the intermediate σ and δ values back up the spanning tree P generated during the expansion phase. The actions are designed along the same rationale as the expansion actions. The `contract_target` action executes local to the target vertex. The local values for $\sigma[target]$ and $\delta[target]$ are collected and sent along as arguments to the `contract_source` action. This subsequent action executes local to the source vertex and computes the updated $\sigma[source]$ value. When all source (predecessor) vertices have been updated, the betweenness centrality score associated with the target vertex is updated (Line 14).

6.3.2 Analysis and Results

The complex interactions between parallel actions is illustrated in Figure 6.35. The three shaded triangles represent the parallel ParalleX processes. Inside each parallel process is a fine-grain data-directed execution of the SSSP algorithm over the graph and updating of the

BC SSSP Process Begin

```
void bc_sssp_type::begin(vertex_id_type start)
{
  ready();

  sigma_[start_id] = 1;
  d_[start_id] = 0;

  Q_.push(start);
  while (!Q_.empty())
  {
    size_type source = Q_.front();
    Q_.pop();

    S_.push_back(source);

    vertex_ids_type new_vertices = expand_source(source);

    for (size_type i = 0; i < new_vertices.size(); i++)
    {
      Q_.push(new_vertices[i]);
    }
  }

  // Synthesize BC scores back up the shortest paths tree
  while (S_.size() > 0)
  {
    vertex_id_type target = S_.back();
    S_.pop_back();

    contract_target(target, start);
  }

  set_ended();
}
```

Figure 6.30. Betweenness centrality process begin action definition.

```
BC SSSP Process Expand Source
1  vertex_ids_type bc_sssp_type::expand_source(vertex_id_type source)
2  {
3    ready();
4
5    long d_source = d_[source];
6    size_type sigma_source = sigma_[source];
7
8    vertex_ids_type new_vertices;
9    blocking_map<
10       expand_target_action
11   >(source, d_source, sigma_source, neighbors, new_vertices);
12
13   return new_vertices;
14 }
```

Figure 6.31. Betweenness centrality process expand source action definition.

BC solution structure. Outside of the processes, the rest of the actions are self-synchronized on the use of the *BC* structure.

The design of the algorithm and implementation of Kernel 4 demonstrates the ability to translate traditional PRAM graph algorithms into the ParalleX model. This approach shows how a traditional approach can be parallelized directly in the PXGL. Parallelism is exploited at all levels by distribution of the ParalleX process and the fine-grain data-directed actions over the distributed graph structure, as in Kernel 2. Continuation migration in the algorithm design is demonstrated by the movement of the work to data during the expansion and contraction phases of the algorithm.

Constraint-based synchronization was demonstrated by the phased use semantics of the global data structure. The phased use is demonstrated by the implicit synchronization of the `bc_vertices`, `bc_scores`, and `bc_max_vertices` structures. Each structure is constructed in parallel, and all initializations are invoked in parallel. In the case of the bc-vertices and bc-scores structures, the bc-sssp processes that initialize the bc-scores structure cannot start

BC SSSP Process Expand Target

```
vertex_id_type bc_sssp_type::expand_target(
    vertex_id_type target,
    vertex_id_type source,
    long d_source,
    size_type sigma_source)
{
  ready();

  vertex_id_type new_vertex;

  if (d_[target] < 0)
  {
    new_vertex = target;
    d_[target] = d_source + 1;
  }

  if (d_[target] == d_source + 1)
  {
    sigma_[target] += sigma_source;

    {
      scoped_use l(use_feb_);

      P_[target_id].push_back(source);
    }
  }

  return new_vertex;
}
```

Figure 6.32. Betweenness centrality process expand target action definition.

```
                    BC SSSP Process Contract Target
void bc_sssp_type::contract_target(
    vertex_id_type target,
    vertex_id_type start)
{
  ready();

  size_type sigma_target(sigma_[target]);
  double delta_target(delta_[target]);

  blocking_for_all<
      contract_source_action
  >(P_[target], target, sigma_target, delta_target);

  if (target != start)
  {
    pxgl::util::scoped_use l(use_feb_);

    bc_scores_.init_incr(target, delta_[target]);
  }
}
```

Figure 6.33. Betweenness centrality process contract target action definition.

```
                    BC SSSP Process Contract Source
void bc_sssp_type::contract_source(
    vertex_id_type source,
    vertex_id_type target,
    size_type sigma_target,
    double delta_target)
{
  ready();

  delta_[source] +=
      (sigma_[source] / sigma_target) * (1 + delta_target);
}
```

Figure 6.34. Betweenness centrality process contract source action definition.

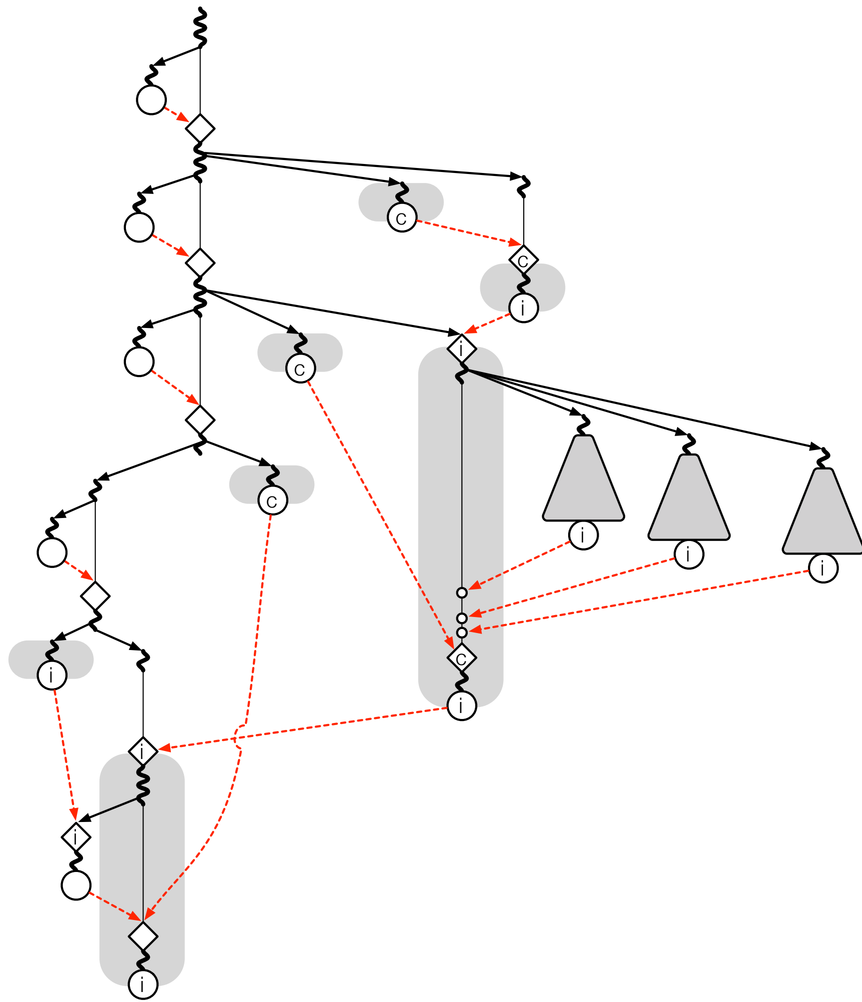


Figure 6.35. Illustration of SGAB Kernel 4 on a single locality.

until the bc-vertices structure has completed initialization. The phased use of the BC structure allows subsequent actions to implicitly synchronize on the completion of the process.

Chapter 7

Conclusions

This research establishes a new approach for highly concurrent, distributed execution for scalable graph processing. Consideration of the requirements of graph-based applications drove the specification of a set of data structures and parallel programming constructs. The features of the ParalleX execution model provided the foundation for the execution of graph algorithms in terms of message-driven dynamic multithreading. Together, this approach to graph programming and the execution model provide a complete strategy for enabling future scalable graph processing.

7.1 An Approach to Graph Processing

Reconsideration of the semantics of graph processing addresses the challenges related to parallelism and efficiency through the design of new graph data structures and parallel constructs. Graph semantics involve the definition of the primitive elements and actions performed on those elements with respect to the objectives of distributed, dynamic multithreading graph processing. The graph, vertex and edge sets, vertices, and edges were represented. Also the fundamental actions, such as visiting a vertex or edge, and the compound operations, such as traversing or searching a graph, were designed to support the objectives of continuation-passing style computation focused on moving work to data and localized constraint-based coordination.

The guiding principle for graph algorithms, kernels, and applications utilizing fine-grain data-directed threading must be to support continuously moving the computation forward. Actions must carry with them the means of continuing execution dependent on the data structure, producing follow-on actions that can independently progress the computation. At

finer-granularities, this is expressed in a continuation-passing style of computation. But at higher levels, data-directed techniques, such as parallel-forall and map constructs express the pattern of computation, while leaving the particulars of dynamic execution to an active runtime system.

Current and anticipated application scales suggest that a shared memory approach cannot provide the level of scaling required for this problem class. The necessary component that must be carried over from the shared memory model is the notion of a global address space. This must be supported for both locating and referencing individual elements of compound distributed data structures (e.g., the vertices of a graph) from anywhere in the system. An explicit notion of locality, providing the ability to reason about whether data is colocated with an action, is necessary. Finally, an explicit concept of data distribution, providing the logical mapping of global structure to system topology, must be a fundamental component of the approach. Distributed data structures must provide interfaces and mechanisms for efficient locality-aware processing by algorithms. And each member of such a structure must be able to maximize local information to work in a decentralized manner whenever possible to reduce the effects of latency and overheads.

With the processing of graph structures, the flow control of the execution is dictated by the structure of the graph data set (i.e., it is data-directed) and limited in available computation, with the basic operations defined in terms of, and at the scale of, the structural relationships between vertices and edges. Thus, the structural meta-data of the graph data set dictates both the next set of tasks, and what activities can be done in parallel. Data-directed execution over an irregular data structure produces time-varying workloads that the programmer is unable to predict or manage using static threading and data-parallel programming constructs.

Likewise, when locally managing a collection of concurrent actions, the use of global barriers to enforce some notion of coherence across the application must be replaced with localized coordination mechanisms embedded in the data structure that support independent progress

at finer granularities. Intrinsic constraint-based synchronization semantics is critical for efficient graph processing. Data-flow variables, and compound structures with the phased use semantics, aid the programmer in managing high degrees of asynchronous, dynamic concurrency. This is powerful because it allows the programmer to reason about concurrent actions over shared values, without needing to explicitly manage synchronized access between producers and consumers. Also, it achieves deterministic concurrency, with no race conditions, no live-locks, and deterministic deadlocks.

7.2 Role of the Execution Model

A parallel execution model provides a conceptual framework for considering design decisions across the computing stack: programming languages, compilers, runtime, operating system, as well as system architecture and hardware technology. The semantics of the execution model allow for the co-design of the different layers in the stack by defining the core concepts that are fundamental to computation and invariant across any implementation. This work builds upon the ParalleX execution model, effectively integrating into the co-design process. Graph algorithm design and application development will thus benefit from a direct path to extreme-scale system architectures through the Petaflops regime and into the Exascale performance domain over the coming decades.

This also enables the development of the ParalleX Graph Library (PXGL). The experimental graph library PXGL was implemented as a proof-of-concept demonstrating the feasibility of this approach for programming graph applications and realizing highly concurrent, distributed execution. The new semantic constructs for graph processing were mapped onto the ParalleX execution model to define the foundation for efficient parallel execution. The PXGL was implemented using the software runtime system HPX to allow for early prototyping on conventional commodity cluster and shared memory systems. This supported the initial experimentation shown which established the viability of the new approach to

graph processing, both in terms of semantic correctness and programmability. Ultimately, the PXGL delivers a framework for the development of graph data structures and algorithms and that will support future research and development of graph-based applications utilizing the ParalleX model.

7.3 Future Directions

The focus on mapping the semantics of programming graphs to the semantics of the ParalleX execution model presents an ideal foundation for future research. Both the data-intensive applications and high-performance systems communities are experiencing a large amount of growth. Graph-based applications will require extensions to the graph model proposed here to support dynamic graphs. For that, the approach must be adapted to support mutating the structural meta-data of the graph throughout the lifetime of the application. The graph interface must be extended and the phased use semantics would have to be reconsidered to support such operations.

Advanced features of the ParalleX model also need to be addressed as support for them is added to the HPX runtime system. Of particular importance to large-scale graph processing is the support for data movement. Data movement supported by the AGAS would enable runtime load balancing of the distributed graph structures. This will be critical for achieving performance as dynamic graph manipulation will introduce data load imbalances, with subsequent work load imbalances following from the data-directed execution style enabled by this approach.

Finally, the implementation of the PXGL defines a solution for commodity cluster and shared memory systems running the Linux operating system and the HPX runtime system. Yet, these semantic constructs defined for graph processing are orthogonal to the choice of hardware technology, system architecture, operating system, and even programming language. By providing a high-level library interface, the PXGL can be used in the implemen-

tation of a broad class of applications. Thus, this work provides a platform for optimization studies over a space of solution vectors for graph-based applications on current and future systems and scalable architectures.

References

[1] J. Dongarra, D.A. Reed, R. Bajcsy, M.A. Fernandez, J.M. Griffiths, R.D. Mott, C.R. Johnson, A.S. Inouye, W. Miner, M.K. Matzke, et al. Computational Science: Ensuring America's Competitiveness, 2005.

[2] D.G. York, J. Adelman, J.E. Anderson Jr, S.F. Anderson, J. Annis, N.A. Bahcall, JA Bakken, R. Barkhouser, S. Bastian, E. Berman, et al. The sloan digital sky survey: Technical summary. *The Astronomical Journal*, 120:1579, 2000.

[3] F.S. Collins, A. Patrinos, E. Jordan, A. Chakravarti, R. Gesteland, and L.R. Walters. New goals for the US human genome project: 1998-2003. *Science*, 282(5389):682, 1998.

[4] F.S. Collins and V.A. McKusick. Implications of the Human Genome Project for medical science. *JAMA: The Journal of the American Medical Association*, 285(5):540, 2001.

[5] L.A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

[6] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. *ACM SIGCOMM Computer Communication Review*, 39(4):51–62, 2009.

[7] R.C. Murphy and P.M. Kogge. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications. *IEEE TRANSACTIONS ON COMPUTERS*, pages 937–945, 2007.

[8] ExaScale Computing Study: Software Challenges in Exascale Systems. Technical report.

[9] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas Wilcox. Extreme-scale amr. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[10] J. Luitjens, B. Worthen, M. Berzins, and T.C. Henderson. Scalable parallel AMR for the Uintah multiphysics code. *Petascale Computing Algorithms and Applications. Chapman and Hall/CRC*, 2007.

[11] L.F. Diachin, R. Hornung, P. Plassmann, and A. Wissink. *Parallel Adaptive Mesh Refinement*. SIAM, 2006.

[12] CD Fry, JV Eccles, and JP Reich. The Ensemble Space Weather Modeling System (eSWMS): Status, Capabilities and Challenges. In *AGU Fall Meeting Abstracts*, volume 1, page 1755, 2010.

[13] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. ParalleX: A Study of A New Parallel Computation Model. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, 2007.

[14] M. Kaiser, H. Brodowicz and T. Sterling. Parallex: An advanced parallel execution model for scaling impaired applications. Submitted to ICPC 2009., 2009.

[15] D.A. Bader. HPCS Scalable Synthetic Compact Applications 2 Graph Analysis, 2006.

[16] D.A. Bader, K. Madduri, J.R. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, 2, 2006.

[17] Bruce Hendrickson and Jonathan W. Berry. Graph analysis with high-performance computing. *Computing in Science and Engineering*, 10(2):14–19, 2008.

[18] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004.

[19] D. Gregor, N. Edmonds, B.W. Barrett, and A. Lumsdaine. The Parallel Boost Graph Library. http://www.osl.iu.edu/research/pbgl, 2005.

[20] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC), July*, 2005.

[21] J.W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and Algorithms for Graph Queries on Multithreaded Architectures. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–14, 2007.

[22] B.W. Barrett, J.W. Berry, R.C. Murphy, and K.B. Wheeler. Implementing a portable Multi-threaded Graph Library: The MTGL on Qthreads. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing-Volume 00*, pages 1–8. IEEE Computer Society, 2009.

[23] J. Siek, L.Q. Lee, A. Lumsdaine, L.Q. Lee, L.S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, M. Heroux, et al. The Boost Graph Library: User Guide and Reference Manual. In *Proceedings of the*, volume 243, pages 112–121. Kluwer, 2002.

[24] K.B. Wheeler, R.C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.

[25] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. the MIT Press, 1999.

[26] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys (CSUR)*, 38(1), 2006.

[27] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, page 262. ACM, 1999.

[28] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L.

[29] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1):5, 2007.

[30] T. Sterling, D.J. Becker, D. Savarese, J.E. Dorband, U.A. Ranawake, and C.V. Packer. BEOWULF: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*. Citeseer, 1995.

[31] T.L. Sterling, P.C. Messina, and P.H. Smith. *Enabling technologies for petaflops computing*. The MIT Press, 1995.

[32] T.L. Sterling. *Beowulf cluster computing with Linux*. The MIT Press, 2002.

[33] L. Dagum and R. Menon. Open MP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[34] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. Scientific and Engineering Computation. The MIT Press, 2007.

[35] K.D. Underwood, M. Vance, J. Berry, and B. Hendrickson. Analyzing the Scalability of Graph Algorithms on Eldorado. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[36] CAR Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):677, 1978.

[37] SD Brookes, CAR Hoare, and AW Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984.

[38] T. Cheatham, A. Fahmy, D.C. Stefanescu, and L.G. Valiant. Bulk synchronous parallel computing-a paradigm for transportable software. In *hicss*, page 268. Published by the IEEE Computer Society, 1995.

[39] MPI Forum. Mpi: A message-passing interface standard. http://www.mpi-forum.org, Sep 2009.

[40] *AM++: A Generalized Active Message Framework*, Vienna, Austria, 09/2010 2010. ACM.

[41] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[42] D. Bonachea. GASNet Specification Version 1.3, 2003.

[43] L. Kale, B. Ramkumar, A. Sinha, and A. Gursoy. The Charm Parallel Programming Language and System: Part I—Description of Language Features. *Urbana*, 51:61801.

[44] R. Keller, D. Kramer, and J.P. Weiss. *Facing the Multicore-challenge: Aspects of New Paradigms and Technologies in Parallel Computing.* Springer-Verlag New York Inc, 2010.

[45] J. Dongarra. Top500 supercomputing sites. http://www.top500.org/.

[46] P.D. Mininni, D.L. Rosenberg, R. Reddy, and A. Pouquet. A hybrid MPI-OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence. *Arxiv preprint arXiv:1003.4322*, 2010.

[47] G. Tang, E.F. D'Azevedo, F. Zhang, J.C. Parker, D.B. Watson, and P.M. Jardine. Application of a hybrid MPI/OpenMP approach for parallel groundwater model calibration using multi-core computers. *Computers & Geosciences*, 2010.

[48] K. Yelick, D. Bonachea, W.Y. Chen, P. Colella, K. Datta, J. Duell, S.L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM New York, NY, USA, 2007.

[49] Yuri Dotsenko. *Expressiveness, Programmability and Portable High Performance of Global Address Space Languages.* PhD thesis, Rice Univesity, January 2007.

[50] T. El-Ghazawi, W. Carlson, and J. Draper. UPC Language Specifications V1. 0, 2001.

[51] J.B. Dennis. Data Flow Supercomputers. *Computer*, 13(11):48–56, 1980.

[52] H.C. Baker Jr and C. Hewitt. The incremental garbage collection of processes. *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, 1977.

[53] R.H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[54] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[55] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2007.

[56] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 218–228. ACM New York, NY, USA, 1986.

[57] R.S. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical report, Technical Report CSG Memo 265, Computation Structures Group, MIT Lab. for Computer Science, Cambridge MA 02139, USA, July 1986.

[58] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and interpretation of computer programs*. Cambridge, Mass.: MIT Press, 1996.

[59] P. Van-Roy and S. Haridi. *Concepts, techniques, and models of computer programming*. The MIT Press, 2004.

[60] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, volume 40, pages 423–437. ACM New York, NY, USA, 2005.

[61] K. Devine, E. Boman, R. Heapby, B. Hendrickson, and C. Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science and Engineering*, pages 90–97, 2002.

[62] G. Karypis, K. Schloegel, and V. Kumar. PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 3.1. *University of Minnesota, Minneapolis*, 2003.

[63] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[64] D.A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 539–550, 2006.

Vita

Dylan Stark was born in Kettering, Ohio, in 1980. He completed his B.S. in 2004 at Louisiana State University, majoring in Computer Science. He completed his M.S. in System Science in 2007 at Louisiana State University. While at Louisiana State University he had the good fortune to work as a graduate researcher at the newly formed Center for Computation and Technology. He also interned at Argonne National Laboratory for one summer and Sandia National Laboratories for the final two years of his doctoral studies. At Argonne he worked on the SPRUCE project under the guidance of Dr. Peter Beckman. At Sandia he worked in the Scalable System Architectures group, where he developed the ParalleX Graph Library.