ACTIVE MESSAGES AS A SPANNING MODEL FOR PARALLEL GRAPH COMPUTATION

Nicholas Edmonds

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing
Indiana University
December 2013

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

Arun Chauhan, Ph.D.

Alex Pothen, Ph.D.

Dirk Van Gucht, Ph.D.

Thomas Sterling, Ph.D.

November 1, 2013

This thesis is dedicated to Linda and Lonne Edmonds, who dedicated countless hours to inspiring my love of learning and respect for knowledge.

Acknowledgements

This thesis has been a long process that would not have been possible without the help and support of a number of incredible friends and colleagues.

Firstly I thank my committee: Andrew Lumsdaine, Arun Chauhan, Alex Pothen, Dirk Van Gucht, and Thomas Sterling. Their time and attention and the valuable perspectives they contributed enabled this work to be greater than I could have made it on my own.

Particular thanks are due to Andrew Lumsdaine for his patience and for providing an environment in which to explore my intellectual curiosities. He taught me the difference between science and engineering and encouraged me to be the most intellectually honest scientist that I could be. His ability to provide much needed guidance when I had lost my path and to help me distill the core insights from my research were invaluable.

While I may be receiving my degree from the School of Informatics and Computing due to organizational changes, the formative years of my graduate education took place under the impeccable tutelage of the faculty of the Computer Science Department in the College of Arts and Sciences at Indiana University. They deserve no small amount of credit for this work.

As a newly accepted 20-year old Ph.D. student I wasn't quite sure what I had signed up for, but I knew that I wasn't ready to be done with my formal education. Jeremy Siek and Katie Siek (then Moor) taught me how grad school worked, introduced me to key future colleagues, and proved to be great friends.

The Open Systems Lab was an excellent environment in which to develop my ideas as well as a rich source of opportunities for collaboration. I am indebted to all the OSL members I shared room 415, and later 135, with in Lindley Hall.

In particular, Doug Gregor introduced me to generic programming in C++ and developed the initial version of the Parallel Boost Graph Library which would form the basis for my eventual thesis as well as a number of publications along the way. Without his influence it's a safe bet that this thesis would not exist.

Brian Barrett was tireless when it came to answering my questions about HPC architecture and MPI implementation details.

Joseph Cottam was invaluable in editing and reviewing drafts of my publications that were no doubt much improved by his comments. He also helped generate high-quality visualizations used in many of my publications that were better than anything I could have done alone.

Ben Martin was always willing to pitch in on projects and improved every one in which he participated.

Jeremiah Willcock and Torsten Hoefler were instrumental in making the application stack presented here a reality. The three of us laid a solid foundation in AM++ and Active Pebbles upon which this work is based.

Chris Mueller was instrumental in my joining the OSL and, in addition to fruitful academic collaboration, inspired me to seek adventure. Without his influence I may never have discovered the mountains and foreign lands that so often served as a refuge during my graduate career.

I would be lost without the support of the amazing staff that that made everything run smoothly so that I could focus on research. Lucy Battersby, Rob Henderson, and the rest of the Indiana University Computer Science Department provided an outstanding work environment. I felt Lucy's absence at the end of my graduate career and no doubt many future graduate students will be worse off for not having known her.

Rebecca Schmitt and Kelsey Allen in the Center for Research in Extreme Scale Technologies went above and beyond to help me out of more than a few binds. Their excellent organizational skills played a large part in the success of many grants that supported my research.

Finally, Erin Dobias was resolute in her support and refused to let me settle for anything but success. Nietzsche said that, “Many are stubborn in pursuit of the path they have chosen, few in pursuit of the goal.” She helped me to remember to focus on the goal.

Nicholas Edmonds

ACTIVE MESSAGES AS A SPANNING MODEL FOR PARALLEL GRAPH COMPUTATION

Graph applications are members of an increasingly important class of applications that lack the natural, or domain-induced, locality of traditional computational science problems induced by large systems of PDEs. Rather than being analytically deducible, the dependency structure of graph applications is determined by the input graph itself. This data-carried dependency structure is expressed at run time and offers limited opportunities for static analysis. Graph applications present challenges with regard to load balancing, resource utilization, and concurrency at HPC scales.

This thesis presents a set of parallel programming abstractions and a software-design methodology that allows for the implementation of flexible, scalable, and highly concurrent graph algorithms. The use of active messages as the underlying communication mechanism provides three key performance benefits over more coarse-grained approaches. First, this phrasing reduces global synchronization and exposes asynchrony that can be used to hide communication latency. Second, by executing and retiring messages as they are received, memory utilization is reduced. Finally, each active message represents an independent quantum of work that can be executed in parallel. By ensuring atomicity of the underlying vertex and edge properties manipulated by messages, fine-grained parallelism can be employed at the message level. The implementation of these ideas is presented in the context of the Parallel Boost Graph Library 2.0.

This library is distinguished from other parallel graph implementations by two key features. By moving computation to data, rather than vice-versa, the effects of communication latency are reduced. Simultaneously, runtime optimization separates algorithm specifications from the underlying implementation. This allows optimization to be performed as the structure of the input graph, and thus the computation, is discovered. Separating specification from implementation also provides performance portability and enables retroactive optimization.

The generic design of the library provides a common framework in which to experiment with dynamic graphs, dynamic runtimes, new algorithms, and new hardware resources. Most importantly, this thesis demonstrates that phrasing graph algorithms as collections of asynchronous, concurrent, message-driven fragments of code allows for natural expression of algorithms, flexible implementations leveraging various forms of parallelism, and performance portability—all without modifying the algorithm expressions themselves.

Andrew Lumsdaine, Ph.D.

Alex Pothen, Ph.D.

Arun Chauhan, Ph.D.

Dirk Van Gucht, Ph.D.

Thomas Sterling, Ph.D.

Contents

List of Figures

List of Acronyms

1

Introduction

Coarse-grained, static, and largely data parallel applications have driven the development of the High Performance Computing (HPC) ecosystem since the inception of high performance computing. Computation and communication hardware has evolved to support these applications by focusing on floating point performance, localized communication, and highly-optimized synchronous collective operations.

The existing HPC ecosystem is effective at supporting these traditional compute intensive applications because the applications possess natural locality due to the local nature of the underlying (PDE) operators. The spatial and temporal locality in these applications provides ample opportunities for data reuse which leads to a large number of FLOPs per memory access. The spatial locality allows communication to be largely confined to a local

neighborhood, which allows effective utilization of networks with low bisection bandwidths. Finally, this natural locality allows the dependency structure of the computation to be largely determined before any portion of the computation occurs. Because locality can be determined analytically at compile-time, the granularity of the application can be coarsened through static analysis and the grouping of computations with shared data dependencies. Infrequent global synchronization can then redistribute data according to predefined patterns known to both sender and receiver between computational phases. This class of problems typically possess good separators in the dependency graph which provides balanced computation well suited to parallelization using Single Program, Multiple Data (SPMD) techniques such as the coarse-grained Bulk Synchronous Parallel (BSP) [**164**] "compute-communicate-synchronize" model.

A growing and increasingly diverse group of scientific disciplines including bioinformatics, social network analysis, and data mining are beginning to utilize graph analytics at HPC scales. However, graph problems have a number of inherent characteristics that distinguish them from traditional scientific applications [**117**]. Graph computations are often completely *data-driven*: they are dictated by the vertex and edge structure of the graph rather than being expressed directly in code. Execution paths and data locations are therefore highly unpredictable. Moreover, the connectivity of many graphs is not determined by physical topology (as is the case for discretized PDEs), resulting in data dependencies and computations with *poor locality*. Partitioning fails to provide significant benefit in such situations as no good separators may exist [**63**,**108**] and scalability can be significantly limited by the resulting unbalanced computational loads. This lack of good separators means that partitioning is unlikely to result in a sparser communication graph and provide more spatial locality than a random distribution. Finally, graph algorithms are often based on exploring the structure of a graph rather than performing large numbers of computations on the graph data, which results in *fine-grained data accesses* and a *high ratio of data accesses to computation*. Latency costs (memory accesses as well as communication) can dominate such computations.

This thesis explores a new approach to parallelizing graph applications by expressing algorithms in a fine-grained fashion and applying dynamic transformations at runtime to create efficient communication patterns. It builds on earlier collaborative work by the author and others on programming and execution models. The new work presented represents the top-most layer of a new application stack for parallel graph computation. The fundamental abstraction utilized to achieve these goals is the active message. Active messages were originally developed as part of the Split-C project [**165**] but are widely used today in a number of different areas. Moving control flow to data using active messages rather than using ghost cells to satisfy data dependencies allows resource utilization to be more effectively controlled. Finally, active messages provide a wealth of opportunities for parallelism at many levels provided atomic transactions on graph properties are supported to preserve the consistency of the shared data. The ability of the active message abstraction to support both coarse- (e.g., processes) and fine-grained (e.g., threads or accelerators) parallelism significantly simplifies the process of parallelizing graph applications on hardware platforms with an increasing number of levels of parallelism and diversity of hardware resources.

1.1. Programming Models and Graph Computation

Programming models well suited to traditional HPC applications depend heavily on the locality inherent in these applications, in particular, each node communicating with only a few local peers. Message passing is an effective programming model for these applications because it provides a clear separation of address spaces and makes all communication explicit. The Message Passing Interface (MPI) is the de facto standard for programming such systems [**125**]. However, graph applications need shared access to data structures which naturally cross address spaces. The access patterns induced on these data structures are both fine-grained—operating at the level of individual vertices and edges—and irregular.

A number of choices exist for how to implement fine-grained, irregular remote memory access. The key requirement with regard to graph applications is that the remote memory updates performed by one process must be atomic with regard to those performed by other processes and must support "read-modify-write" operations (e.g., compare-and-swap, fetch-and-add, etc.). More importantly, only the process performing the updates has knowledge of which regions of memory are being updated and thus the process whose memory is the target of these updates cannot perform any sequencing or arbitration of the updates. We refer to this as the "sender knowledge" case, using terminology from Holk et al. [**87**], as distinguished from the case where both the sender and receiver have knowledge of the destination of the updates being performed. Finally, some algorithms require concurrent, dependent updates to multiple, non-contiguous locations in memory, which provides perhaps the greatest challenge to a programming model.

The structure of graph computation is dependent on the structure of the input graph itself which is discovered dynamically at runtime. This fact makes static analysis and compile time optimization difficult and renders computational models which rely on rely on these techniques less effective. The irregular nature of graph computation and the unbalanced computational loads generated by the poor-quality partitions used to distribute them frustrate efforts at coarsening the computation at the algorithm level. Static approaches to hiding memory and communication latency are therefore ineffectual. However, techniques such as multithreading have demonstrated that asynchrony at runtime can be used effectively to hide latency. Active messages provide a mechanism to expose asynchrony both within and across address spaces. Utilizing active messages allows both memory and communication latency to be hidden effectively using a common abstraction. The active message abstraction also supports user-defined "read-modify-write" operations of arbitrarily large granularity to be expressed. In effect, active messages allow transactional memory [**146**] to be generalized to distributed memory.

1.2. Runtime Optimization

Compile time coarsening of graph applications is unlikely to yield balanced, regular units of work which are well suited to coarse-grained parallelism due to the lack of separators required for such a decomposition [**63**, **108**]. However, fine-grained implementations are poorly suited to traditional multi-level memory hierarchies and networks. Both systems perform significantly better when moving a single large data segment vs. moving the same amount of data using multiple smaller segments. A multitude of architectural features could be discussed to support this including prefetching, caches, and network injection rate limitations to name a few. We have argued that a successful programming model for graph applications is fine grained: fundamental "graph operations" consist of individual vertex and edge accesses. Efficient implementations on modern hardware require relatively coarser-grained operations to yield maximum efficiency. This presents two implementation options:

(1) Attempt to minimize communication overhead as much as possible and perform naïve fine-grained communication.
(2) Attempt to apply optimizations at runtime

Given the dynamic nature of graph computations and the infeasibility of static coarsening the only choice that remains is to perform coarsening at runtime. By specifying the fine-grained structure of graph computations using active messages that operate on individual vertices and edges we preserve the full dependency structure of the computation until runtime. This allows runtime coarsening to effectively capture the critical path of the application without artificially extending it as could occur with compile time coarsening. Having the full dependency structure of the computation available allows the runtime system to make the most effective decisions possible with regard to coarsening and other potential optimizations. The drawback is that all analysis performed as a precursor to optimization contributes to the application's execution time. This thesis applies a number of optimizations to the stream of fine-grained messages generated at runtime to attempt to

recover some of the performance benefits available to other classes of applications through static analysis. These optimizations are explored in detail in Chapter 4.

Graph applications also differ markedly from more traditional HPC applications in the way in which data that is communicated is used by the application. Traditional compute-intensive applications are characterized by a high ratio of floating point operations to memory accesses. High rates of date reuse allow efficient caching of operands using traditional multi-level caches and hierarchical memory. Graph applications have much lower ratios of floating point (or integer) operations to memory operations, and thus lower rates of data reuse. Satisfying data dependencies by moving data to the site of computation is an effective strategy for traditional HPC applications because the communicated data is likely to be reused many times. In the absence of data reuse however, moving data to computation and moving computation to data may have similar communication costs.

Satisfying data dependencies by moving data to the computation site has a number of drawbacks. Caching remote data using ghost cells requires additional memory and memory copy operations for the underlying storage. Maintaining the consistency of these ghost cells necessitates complicated consistency models—likely some sort of relaxed consistency to maintain efficiency–and additional communication to enforce these consistency models. Separating data flow from control flow makes reasoning about when computational dependencies have been satisfied difficult or inefficient. In traditional applications these drawbacks are balanced by high rates of data reuse and simplified communication patterns relative to unstructured graph applications.

Moving computation to data using active messages has a number of distinct advantages for graph applications. Active messages communicate control flow, possibly in combination with attached data, unifying data flow and control flow. Once a message arrives, two possible situations exist regarding its data dependencies. If all data dependencies are satisfied via a combination of local data on the receiving process and data attached to the message itself, the message can be executed and retired. If some data dependencies remain unsatisfied locally available data can be appended to the message and the message

forwarded to the owner of one of the remaining data dependencies. In this fashion messages act as closures in which the environment is updated as non-local data dependencies are satisfied. The ability to execute and retire messages during a communication epoch reduces memory utilization. This in turn allows the application as a whole to execute with less total memory, potentially allowing for more locality in the execution environment (e.g., using fewer cluster nodes or more work per node). Finally, because messages are independent and rely only on the data in the message itself and data available locally on the receiving process, greater asynchrony is available vs. programming models which rely on bulk data movement.

1.3. Separating Expression from Execution

The fashion in which graph algorithms are expressed need not map explicitly to the form in which they are executed. In fact, an abstract specification of an algorithm may be adapted to a variety of implementation depending on the hardware resources available. HPC is anything but immune to the many core revolution; where once clusters of workstations dominated, current machines possess an ever increasing number of levels of parallelism. To the familiar Communicating Sequential Processes model has been added multi-core parallelism within a node, hyperthreading within a core, a variety of vector units on-die, and more recently, a variety in PCIe-attached accelerators which are migrating closer to the host CPU . Abstractions that work well within a single one of these levels of parallelism have proliferated with varying degrees of success (e.g., MPI, OpenMP, CUDA , and a host of less prominent systems). Abstractions that work well across multiple levels are significantly more rare. This is evidenced by the vast preponderance of papers on combining MPI and OpenMP and exactly what number of MPI ranks vs. OpenMP threads is best suited to a particular problem.

A suitably abstract specification would be one which is capable of being mapped to all of the aforementioned hardware environments, and others that may be developed in the future, without requiring changes to the algorithm specification itself. Active messages are agnostic as to where they execute and independent of one another provided atomic access

to the data they share is ensured. This fact, in combination with their asynchronous nature, makes it straightforward to steer their execution to hardware contexts of various forms.

A naïve implementation of an active message-based algorithm may be poorly matched to a given hardware context for a variety of reasons. A single message may be too small to make efficient use of communication resources when sent alone. Handling messages individually may lead to excessive overhead or contention. Dense communication graphs between large numbers of ranks may lead to excessive communication buffer space requirements. Performance optimizations to handle these cases and many others are implemented time and again by application programmers. In traditional HPC applications these take the form of static compile-time optimizations. In the graph domain these optimizations must be deferred until runtime when the structure of the computation is discovered. Further discussion of the types of optimizations performed and their effects are discussed in Chapter 4.

These optimizations can also be separated into specification and implementation components. By having application developers specify a set of optimizations which are semantically valid for a given algorithm, users and/or the runtime can select the set of optimizations to be applied without modifying the algorithm specification. This ability to retroactively change the optimizations applied to an algorithm is the key to performance portability. Parameterizing these optimizations (e.g., on message size or routing topology) provides an additional level of control and an opportunity to auto-tune for a particular machine. Suitably generic designs allow algorithms to leverage optimizations that were unavailable when the algorithm was implemented without modification, provided the optimization is a member of a class of which the algorithm is aware. Finally, this optimization framework makes it possible for the runtime to dynamically vary optimization parameters in response to changing system characteristics.

1.4. Elements of an Active Message Graph Library

This thesis includes elements of three separate projects, the Parallel Boost Graph Library (Parallel BGL), AM++, and Active Pebbles. The **Parallel BGL** is a library of parallel

graph algorithms which extends the sequential Boost Graph Library (BGL) to provide distributed memory parallelism. The Parallel BGL builds on the BGL, offering similar data structures, algorithms, and syntax for distributed, parallel computation that the BGL offers for sequential programs.

The Parallel BGL was developed by lifting [**77**] away the implicit requirements of sequential execution and a single shared address space. The original Parallel BGL primarily utilizes a coarse-grained BSP approach communicating over MPI to parallelizing graph computation. The only available mechanism for parallelism within a shared address space is to execute additional processes. This allows algorithms to use additional cores on a single compute node, but has a number of drawbacks. First, communication between processes in the same address space is accomplished via messaging which introduces unnecessary copies compared to a straightforward ownership-transfer approach which requires only the communication of an address in the memory shared by two processes on a single node. Secondly, the Parallel BGL uses a static decomposition of the graph data consisting of a row-wise decomposition of the adjacency matrix. In the case where a single process per node exists, that process owns $\frac{|V|}{p}$ rows of the adjacency matrix, where $|V|$ is the number of vertices in the graph and p is the number of parallel processes. Adding k additional processes to each node to utilize additional cores results in each process owning $\frac{|V|}{p(k+1)}$ rows of the adjacency matrix. This additional discretization of the graph data further reduces locality and exacerbates issues of computational imbalance as well as incurring additional communication overhead. These issues will be further discussed in Chapter 7.

Later in the development of the Parallel BGL, a number of extensions to improve performance and better leverage shared-memory parallelism were explored. The addition of loop-level parallelism and the incorporation of ad hoc active messages [**60**] to more effectively overlap communication and computation were two of the most promising extensions. The theory that re-phrasing the Parallel BGL as a purely active message based library would allow fine-grained parallelism, overlap of communication and computation, and a reduced reliance on global synchronization led to the development of **AM**++. AM++

is a generalized active message library intended for use by non-traditional HPC applications. It is targeted at a "middle ground" between low-level active message libraries such as GASNet [**26**] and object-based runtime systems such as Charm++ [**97**]. AM++ has a number of features that were important to the effort to reimplement the Parallel BGL as an active message based library:

- Allowing message handlers to themselves send arbitrary messages, simplifying the expression of some applications and exposing maximum asynchrony.
- A multi-paradigm implementation, including object-oriented and generic programming, to balance runtime flexibility with high performance.
- Type-safe messages.
- A modular design enabling configurability by the user without substantial application modification.
- A design enabling compiler optimizations such as inlining and vectorization of message handler loops, and multi-threaded message handlers.

In identifying the abstractions required to design an active-message version of the Parallel BGL, two classes of abstractions were discovered. Some abstractions belonged purely to the graph domain, while others were found to be common across a broad range of data-driven applications. Of this second set, some were abstractions related to how algorithms were specified, and others were related to the efficient execution of algorithms. The abstractions common to a variety of data-driven problems were distilled into **Active Pebbles**.

To provide the simultaneous benefits of fine-grained programmability with scalable performance, the Active Pebbles model relies on the following five integrated techniques:

(1) **Fine-grained Addressing** — light-weight global addressing to route messages to targets.

(2) **Message Coalescing** — combining messages to trade message rate for latency and bandwidth.

(3) **Active Routing** — restricting the network topology to trade message throughput for latency for large numbers of processes.

(4) **Message Reductions** — message processing at sources and intermediate routing hops (where possible).

(5) **Termination Detection** — customizable detection of system quiescence.

Messages and targets, in combination with Fine-grained Addressing (1), define an abstract programming model. The techniques in 2–5 describe an execution model which translates programs expressed using the programming model into high-performance implementations. Accordingly, techniques used in the execution model are not simply implementation details: e.g., message reductions in combination with routing cause a decrease in the asymptotic message complexity of some algorithms and are thus essential to the model.

1.5. Evaluation and Contribution

This thesis demonstrates expressive and high-performance techniques for implementing parallel graph algorithms in hybrid-multicore HPC systems using active messages. By combining generic programming with active messages we have developed a conceptually simple parallel graph library (the Parallel BGL 2.0) with flexibility and adaptivity which is superior to competing solutions. By using traditional active messages with a number of novel extensions we are able to capture and utilized the fine-grained dependency structure of graph applications to out-perform coarse-grained models. Finally, the separation of expression from implementation allows a single algorithm specification to be implemented in a performance-portable fashion across a variety of hardware resources and to utilize optimizations developed independently of the algorithm itself.

- Chapter 2 introduces many of the concepts that inspired the software design proposed in this thesis.
- Chapter 3 introduces AM++ and discusses the work required to generalize active messages for use in a user-level graph library.

- Chapter 4 discusses the methods by which specification and implementation of graph algorithms are separated, using (in part) the Active Pebbles programming and execution models, in order to support retroactive optimization, auto-tuning, and dynamic runtimes.
- Chapter 5 introduces a classification of parallel graph algorithms that will be used to demonstrate the broad applicability of active messages for implementing graph algorithms.
- Chapter 6 examines, with examples, how algorithms from the literature are converted to forms suitable for expression using active messages. The methods by which possible runtime optimizations are specified are also discussed.
- Chapter 7 explores how fine-grained phrasings of graph algorithms allow greater asynchrony and parallelism than coarse-grained approaches and reduce resource utilization.
- Chapter 8 discusses the distributed property map, a pseudo-DSM layer in the original Parallel BGL which was retained to provide PGAS semantics in the active message version of the graph library.
- Chapter 9 introduces future directions for active message graph computation, including dynamic graphs and dynamic runtimes. A motivating example which demonstrates the strength of message-driven graph computation is also introduced.
- Chapter 10 summarizes the contributions of this thesis.

1.6. Conclusion

The parallel programming abstractions and software design methodology presented in this thesis represent a departure from traditional HPC programming models. By removing synchronous operations and implementing algorithms in a message-driven form, we enable the critical path of fine grained applications to be accurately captured. Implementing commonly used optimizations in the runtime reduces programmer effort and allows algorithm expression to be separated from implementation.

2

HPC Programming Models, Runtimes, and Parallel Graph Algorithm Implementations

The methodology described in this thesis represents a new approach to implementing parallel graphs algorithms that combines a communication strategy originating in dataflow programming with fine-grained shared memory parallelism and transactional updates to graph properties. This programming model is supported by a runtime system which optimizes the fine-grained message stream emitted by graph algorithms to produce efficient implementations.

In contrast to Partitioned Global Address Space (PGAS) programming models, this approach inverts the typical approach of moving data dependencies to control flow by

moving control flow to data using active messages. This technique was inspired by a number of experimental hardware platforms designed to support fine-grained shared memory programming.

The algorithm implementations discussed in this theses were influenced by a large body of parallel algorithms research as well as implementations of individual algorithms. Significant modification to algorithms was often required in order to phrase them in an active message form.

This chapter reviews many programming models targeting HPC-scale parallel applications implemented at the language, library, and runtime level. Various approaches to parallelizing graph algorithms either in the shared-memory or in the distributed-memory domain are also discussed along with their relation to the techniques in the this thesis.

2.1. Runtime Support for HPC Applications

A variety of relatively low-level runtime systems for supporting high-performance parallel applications have received widespread acceptance. These runtime systems can broadly be grouped into two categories: those targeted at coarse-grained distributed memory parallelism, and those targeted at fine-grained shared memory parallelism. This categories can can also be thought of in terms of data-parallelism and task parallelism respectively, with minor losses in distinction. The literature contains several architecture-focused taxonomies of parallelism which may be employed as well [**57**, **79**, **140**].

Pthreads (POSIX threads) is the most commonly used library on Unix-like systems for thread-based parallelism. Pthreads requires threads to be managed manually; it is designed as a low-level interface with thread creation/deletion and synchronization operations. OpenMP [**46**] is a set of language extensions (implemented as compiler pragmas), typically used for shared-memory computation, which simplify implementation of common threading paradigms. OpenMP is primarily designed for data decomposition. Task parallelism is an alternate method of parallelism which prioritizes work decomposition and scheduling over data decomposition. Task parallelism is often implemented in an ad hoc fashion, but robust runtime systems such as Intel's Threading Building Blocks [**89**],

Microsoft's Task Parallel Library [**112**], and PFunc [**99**] are gaining increased support. Cilk and Cilk++ [**24**] are language extensions to C/C++ which both provide support for task parallelism as well as incorporating runtime scheduling and load balancing.

Messaging-based approaches are commonly utilized when a global shared address space is not available or is impossible to implement efficiently. MPI [**56**, **123**] is the de facto standard library for message passing in parallel computing. It assumes a distributed-memory model, with communication and synchronization explicit and implicit for both the sender and receiver of data. Later versions of MPI are less restrictive; they allow one-sided remote memory operations that only require explicit action by the sender. Classical MPI-1 [**123**] programs are epoch-based and frequently use a strict BSP [**164**] programming model. MPI-2 [**125**] extensions provide extended collectives, one-sided communication, and dynamic process management. Non-blocking collectives from libraries such as libNBC [**85**] further extend the flexibility of MPI and have been included in MPI-3 [**126**], along with more powerful one-sided communication operations. Algorithms for dynamic sparse data exchange such as [**86**] have the potential to improve the performance of graph algorithms implemented using message passing and collective communication.

Alternatives to MPI include a variety of active message libraries which allow user-defined code to be invoked asynchronously on a remote process. Examples of these libraries include IBM's Deep Computing Messaging Framework (DCMF) [**106**], Low-level Application Programming Interface (LAPI) [**143**], and Parallel Active Messaging Interface (PAMI) [**105**]; Myrinet Express (MX) [**74**]; and GASNet [**26**]. Systems such as Unified Parallel C (UPC) [**163**], Co-Array Fortran [**128**], a number of MPI implementations, and object-based programming systems all rely on active messaging for their low-level transport. Using active messages directly in an application is frequently difficult because the interfaces to existing active message systems are very low-level.

2.2. Architectures, Languages, and Programming Models

Research into next-generation parallel computing platforms, languages, and programming models has been focused on two broadly-defined areas. First, a variety of hardware

approaches have sought to provide high levels of on-chip parallelism to both increase instruction throughput and reduce or hide memory and network latency. Second, languages and libraries have been developed to reduce the conceptual burden of programming distributed and heterogeneous systems.

Early research into on-chip parallelism led to fine or interleaved [**109**] multithreading approaches with extremely cheap switching between hardware thread contexts [**151**]. Fine-grained multithreading is effective at hiding memory latency, but suffers when there is inadequate parallelism available to utilize all the available parallel contexts. Simultaneous Multithreading (SMT) is a more recent evolution of this approach to hiding latency. Niagara [**103**] is a processor architecture from Sun (now Oracle) that relies heavily on in-processor SMT. The Cray XMT [**64**] (based on the Tera MTA [**10**] series) is a more extreme version of an SMT approach, designed for programs with thousands of threads. In that system, threading is used to hide memory latency; sophisticated hardware-supported lightweight synchronization operations are provided to communicate between the threads. The IBM Cyclops64 [**9**] is another highly multi-threaded architecture distinguished by the fact that rather than providing inexpensive thread context switching, each thread context has its own execution hardware. Other approaches to addressing memory latency include approaches such as the J-Machine [**47**] which utilize VLSI techniques to embed lightweight processors in memory systems. Later microprocessor designs, such as Sparcle [**5**], combine latency tolerant architectures, optimized fine-grained synchronization, and fast message handling to allow them to be combined into large-scale multiprocessors [**4**].

The PGAS model, as exemplified by Titanium [**174**], UPC [**163**], and Co-array Fortran [**128**], is an approach that mixes shared- and distributed-memory semantics. The use of remote data is explicit in computations, yet global pointers can be used to access remote objects. X10 [**39**] and Chapel [**32**] are PGAS languages which provide support for programming models beyond simple data parallelism. These languages combine PGAS-style support for data parallelism with support for task parallelism via asynchronous spawning primitives, nested parallelism, and fine-grained concurrency. PGAS frameworks have

identified and are addressing limitations in their handling of graph algorithms. For example, Jose et al. introduced UPC Queues, a mechanism for faster producer-consumer communication in the context of Unified Parallel C [**94**]. The goal of this work is to improve UPC's performance on the Graph 500 benchmark [**127**] (breadth-first search).

The Chapel [**32**] language provides direct support for asynchronous active messages using the ***on ... do begin*** idiom [**50**], as well as native atomic blocks. This model inspired one of the intermediate languages described in Chapter 6 § 6.2, and (if combined with an appropriate execution model) would be a suitable abstraction for expressing graph algorithms using the approach described in this paper. The X10 language [**39**] has similar constructs using ***async*** and ***at*** keywords for sending active messages. That language has been used to create the ScaleGraph [**48**] graph library; however their implementation techniques are not described. The default execution models for these languages do not provide efficient support for messages at the fine granularities we use for graph algorithms, however.

Message-driven and dataflow models are similar to the PGAS languages in that they are data-parallel, but emphasize the connections between computations rather than the computations themselves. The Actors [**6**] model uses messages rather than sequential processes to convey both control and data. Dataflow languages similarly treat data as the primary concept behind any program and use data inputs to define control flow. Coordination languages such as Linda [**73**] treat process coordination as a separate activity from computation.

2.3. Parallel Graph Algorithm Implementations

A number of libraries targeted at parallel graph computation exist which utilize a variety of abstractions. The original version of the Parallel BGL [**77**] utilized a coarse-grained Bulk Synchronous Parallel [**164**] programming model and targeted only distributed memory parallelism. Later extensions added some ad hoc forms of active messaging, though the performance of these approaches is limited by the underlying communication layer. By abstracting these features into a separate layer in the application stack we have gained

generalization, flexibility, and through careful design, performance. This separate, thread-safe active message layer also made incorporating fine-grained parallelism (via threading) straightforward. We refer to the new active message-based version of the Parallel BGL as Parallel BGL 2.0.

The Graph Algorithm and Pattern Discovery Toolbox [**76**], later renamed the Knowledge Discovery Toolkit (KDT), provides both combinatorial and numerical tools to manipulate large graphs interactively. KDT runs in parallel over Star-P [**144**] and targets distributed memory parallelism. KDT focuses on algorithms though the underlying sparse matrix kernels are also exposed. Later versions of KDT use the Combinatorial BLAS [**31**] as the computational back-end. The Combinatorial BLAS uses linear algebraic primitives to perform graph computation and incorporates two-dimensional data decompositions. Whether viewing the underlying data structure as a graph or a sparse matrix is more appropriate depends on the operations required by a given algorithm. In some cases, both types of solutions are equally effective, while in others, either a graph or sparse-matrix representation may be preferable. We view sparse-matrix libraries as providing complementary functionality to parallel graph libraries and have previously demonstrated the ability to enable the Parallel BGL to interoperate with other generic libraries such as the Iterative Eigensolver Template Library [**28**, **160**]. Similar techniques may enable the graph library presented here and other libraries, such as the Combinatorial BLAS, to interoperate, thus enabling algorithm implementations which employ primitives from both libraries on the same underlying data. Because both the Parallel BGL 2.0 and the Combinatorial BLAS are generic libraries it should be straightforward to allow the libraries to operate on each other's underlying data representations, providing users a choice of abstractions and allowing them to select the one best suited to their problems, or mix and match the two.

The MultiThreaded Graph Library (MTGL) [**23**] and Small-world Network Analysis and Partitioning (SNAP) [**15**] are shared-memory libraries for manipulating graphs and are thus limited to the resources available on a single symmetric multiprocessor. The MTGL uses loop-level parallelism and complicated dependency analysis to generate efficient code for the Cray XMT and, via virtualization, commodity SMPs [**170**] . MTGL is not as generic

as the Parallel BGL is in terms of supported graph types and algorithms. STINGER [**58**] extends shared-memory parallel graph processing to streaming, dynamic networks. Experimental results indicate that out-of-core approaches are incapable of matching the performance of distributed-memory implementations [**7**] meaning that implementations capable of leveraging both shared and distributed memory parallelism are essential for graph computation at large scale.

Several distributed-memory parallel libraries provide graph algorithms and data structures. The ParGraph library [**84**], also built on top of the sequential BGL, provides generic max-flow and breadth-first search implementations, although it opts for a more explicit representation of communication that does not permit reuse of, e.g., breadth-first search in the parallel context. The CGM*graph* library [**34**, **35**] is an object-oriented library for distributed graph computation. The Standard Template Adaptive Parallel Library (STAPL) [**11**] is a generic, parallel library modeled after the Standard Template Library and providing distributed data structures and parallel algorithms, including a graph data type. Where the Parallel BGL has opted to encode distribution and parallel communication information within the data types and selects efficient algorithms at compile time, STAPL delays such decisions until run-time to adapt to the current execution environment. STAPL is built upon the ARMI active-message runtime system [**158**]. ARMI supports automatic message coalescing to coarsen message granularity for performance, but does not provide routing or message reductions natively. STAPL's parallel graph class pGraph is based on the sequential Boost Graph Library, and some of its algorithms use nested active messages to create distributed-memory parallelism, but details are not provided [**157**].

The popularity of high-level, data-parallel frameworks such as MapReduce [**49**] and Dryad [**91**] and the widespread availability of cloud-computing resources have driven a number of efforts to adapt these resources to application domains with complex dependency structures such as graph processing. Extensions such as Spark [**176**] extend these frameworks to iterative computation. Pegasus [**100**] is a collection of graph algorithms for mining large graphs using Hadoop [**171**]. The vast majority of libraries and tools suitable for parallel graph computation such as Pregel [**120**], Giraph [**3**], and Piccolo [**135**] are based

on the BSP model. Chapter 7 examines in detail why this model is suboptimal for parallel graph computation. GraphLab [**116**] is a framework for machine learning and data mining in the cloud. It attempts to provide additional asynchrony compared to other coarse grained cloud-based graph algorithm implementations. GraphLab relies on the availability of good partitions in the graph being mined or, in the absence of trivial colorings, distributed locking. GraphLab uses a similar combination of ghost cells and *pulling* data dependencies to computation as the original Parallel BGL. Performance results indicate that GraphLab is unable to outperform even highly-synchronous MPI codes when the ratio of communication to computation is small or the input graph does not provide for good partitions with small edge cuts. The MPI codes compared against were running over TCP/IP in Amazon Elastic Compute Cloud (EC2) [**2**], which means that their performance was likely to be significantly worse than what would be expected on a dedicated cluster with a high-speed, low-latency interconnect and collectives optimized for the interconnect's topology. Cloud-based solutions fill an important gap for users without access to dedicated HPC resources but are unlikely to provide competitive performance on latency-bound computations due to the limitations of the IP networks utilized. Data-mining applications are more loosely coupled and suffer less from these high latency networks. Thus, cloud-based solutions are largely focused on these types of applications as opposed to traversal-based algorithms.

A variety of individual implementations of graph algorithms that leverage shared-memory parallelism on both commodity and highly multi-threaded architectures exist, such as those by David Bader and Kamesh Madduri [**16**, **18**, **119**]. Examining a variety of concrete implementations is the first step in the lifting process through which effective abstractions are identified and generic libraries are developed. These existing shared memory implementations can be used to improve algorithm implementations presented here. Individual examples of distributed memory graph algorithms do exist [**175**] as well, and provide useful insight for mapping graph algorithms to distributed memory HPC platforms.

Exposing simplified interfaces through higher-level languages or new abstractions is an excellent way to extend the impact of library implementations to a broader range of application developers and scientists. Various efforts exist to express a high-level interface to unstructured problems [**75**,**104**,**144**]. Effective abstractions are seldom developed in a top-down fashion however. High-level interfaces often benefit from very efficient lower-level library implementations as well. The graph library presented here is beneficial to these types of efforts both in that it provides a flexible, efficient, robust framework for high-level interfaces to target, and because through its development we discover new programming abstractions in a bottom up manner.

2.4. Active Messages

Previous libraries supporting active message or Remote Procedure Call (RPC) semantics can be broadly grouped into two categories: low-level and high-level. Low-level systems are meant for other libraries and run-time systems to build upon, or to be used as a target for compilers; performance is the most important goal, with flexibility less important. These libraries do not provide type safety for messages and often have arbitrary, frequently system-specific, limits on the number and types of parameters to an active message, actions that can be performed in active message handlers, etc. As these libraries assume a sophisticated user, features such as message coalescing are not always provided. Libraries such as IBM's DCMF [**106**], LAPI [**143**], and PAMI [**105**]; MX [**74**]; and GASNet [**26**] are typical examples. GASNet's manual states that it is not intended for direct use by applications [**26**, §1.2], while DCMF, LAPI, and PAMI are intended for such use.

On the other hand, libraries in the RPC tradition are designed for usability, often at the expense of performance. These libraries tend to be more flexible, and more likely to support heterogeneous systems, interface discovery (including sophisticated interface definition languages), and security; run-time dispatch is used heavily, but type checking is usually provided. Some RPC systems expose individual functions for remote access, while others expose objects and their methods. Examples of traditional RPC systems include Java RMI [**166**] , CORBA [**129**] , and ONC RPC [**159**].

Some authors have implemented systems based on RPC principles that are intended for high-performance computing applications. These systems trade off some of the flexibility of fully general RPC in exchange for higher performance. They also include techniques such as message coalescing to reduce message counts, and rely on asynchronous method calls to hide latency. These systems include Charm++ [**97**, **148**] and ARMI [**158**]. The ParalleX system is also based on active message/RPC principles, with sophisticated methods for locating possibly-migrated remote objects and non-message-based synchronization methods [**96**].

Our work differs from these systems in that it does not use objects as the target of remote accesses; we instead use functions registered as handlers. Our messages target nodes, not single objects. Because we allow C++ function objects, methods could be used as AM++ handlers as well by adding an explicit parameter representing the object being addressed, but that is not the intended use case. We provide the flexibility of arbitrary actions (including active message sends) in message handlers like in RPC systems, but use a fully asynchronous model: a message is not guaranteed to be sent until an explicit synchronization operation. We additionally avoid run-time dispatch for our higher-level interfaces using the techniques of generic and generative programming; the compiler is able to resolve that an entire buffer of coalesced messages is all intended for the same handler, and optimize accordingly. We therefore are in the middle, between the low-level Active Message (AM) implementations with their focus on performance and the higher-level systems with their focus on ease of use (Figure 2.1); our goal is a mix of both.

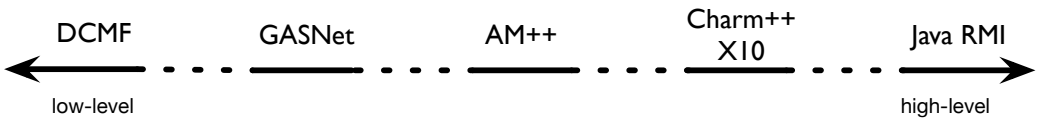


FIGURE 2.1. Comparison of AM++ vs. other active message implementations

The MPI standard defines *one-sided* operations [**125**, §11] which are in some sense similar to active messages with predefined handlers. Those operations allow users to access remote memory (put and get) and perform calculations on remote memory (accumulate).

The latter is probably closest to the concept of active messages; however, MPI specifies a fixed set of possible operations and does not currently support user-defined accumulations. Additions to one-sided operations in MPI-3 [**126**, ch. 11] such as *MPI_Fetch_and_op()* and *MPI_Compare_and_swap()* may also rectify some of the shortcomings of the MPI one-sided communication model, though high performance implementations were unavailable when this work was conducted. MPI-3 does not allow user-defined functions to be used with these additional one-sided operations which prevents their utilization as a basis for generalized active messages.

PGAS languages such as UPC [**163**] and Co-Array Fortran [**128**], are comparable to AM++ in that they are both trying to provide user-level interfaces to AM and one-sided messaging. The Berkeley UPC compiler's runtime system uses GASNet for communication [**41**]. However, these languages primarily use active messages and one-sided operations for data movement, not for the implementation of user algorithms. AM++ exposes the ability to use active messages for algorithm implementation.

2.5. Distributed Shared Memory and Transactional Memory

Distributed Shared Memory (DSM) [**136**] allows applications written for shared-memory models to execute on distributed memory systems by emulating a single shared address space. Many DSM systems treat the disjoint physical address spaces as an additional level in the memory hierarchy and use page-based approaches similar to those utilized by virtual memory systems to move data in the logically shared address space to individual physical address spaces where it can be operated on. While page-based approaches make addressing objects in the shared address space straightforward, they also increase the likelihood of false sharing relative to more fine-grained approaches. This concern is magnified by the fact that many DSM systems use some form of release consistency. DSM models that use weak consistency models suffer from the issues in the original Parallel BGL design where excessive memory barriers group accesses in a fashion that makes exploiting fine-grained dependency structure difficult.

Object-based DSM systems also exist to address the granularity issues some applications experience when using page-based DSM systems. Adsmith [**115**] is an object-based DSM system with an addressing scheme similar to distributed property maps in the original Parallel BGL. However, Adsmith uses release consistency and only implements load-store semantics whereas distributed property maps implement richer semantics for manipulating shared objects and enforces consistency without explicitly acquiring and releasing objects. Object-based DSM extensions to Aleph [**81**] also exist [**95**]. The global associative tuplespace in Linda [**73**] is another example of an object-based DSM. Object-based DSM systems which use weak consistency models can be used to implement graph algorithms, as shown in the original Parallel BGL design, and are similar to distributed property maps. One of the claims of this thesis is that moving data to computation is sub-optimal in unstructured computations and that models which move computation to data have higher performance.

In some sense active messages which can perform arbitrary operations as in AM++ extend memory transactions to distributed memory, provided transactions are supported within address spaces. Memory transactions are, in the simplest case, simply a group of memory operations which are guaranteed to be observed atomically. Memory transactions can be implemented with locks or critical sections, however lock-free and wait-free approaches provide better performance [**82**]. Transactional memories provide support for this behavior directly in the system architecture [**83**]. A variety of methods for emulating transactional memory in software have been explored [**146**]. The programming model and algorithm implementations in this thesis would benefit greatly from hardware support for transactional memory and extending the work to leverage them would be straightforward.

2.6. Conclusion

The work presented in this thesis is based on two primary observations about parallel graph algorithms. First, graph algorithms lack a fixed computation or communication structure that can be analyzed and statically optimized at compile time. The structure of the application is defined by the input graph which is discovered at runtime. Second,

these algorithms have a low ratio of computation to communication making them highly sensitive to memory and network latency. Furthermore, they have little spatial or temporal locality rendering traditional hierarchical caches ineffective.

Coarse-grained, bulk-synchronous parallel implementations of graph algorithms exist in a number of forms. A variety of programming abstractions exist to support this method of parallelism including the MapReduce pattern and PGAS languages[1]. These coarse-grained approaches create artificial dependencies as a result of their static work decompositions and fail to capture the fine-grained critical path of graph applications. Efficient graph algorithm implementations are distinguished by their ability to generate maximal parallelism while minimizing latency due to memory access or contention, including access to remote data in the case of distributed-memory. Inspired by a number of hardware techniques for leveraging asynchrony to tolerate latency we have elected to pursue an approach that combines fine-grained expression with runtime optimization to generate efficient implementations for hybrid environments containing distributed-memory machines with a variety of shared-memory, parallel resources.

Virtual memory **[22]** is an excellent example of how a single abstraction (the virtual address space) can allow users to efficiently manipulate extraordinarily complex, multi-level systems (the physical memory hierarchy). Rather than a two-level approach to parallelism (e.g., MPI + OpenMP) we have been able to extend an existing parallel programming abstraction, active messages, to provide for efficient parallel performance at multiple levels. This reduces the cognitive burden on the user and allows for a broader variety of possible implementation contexts and optimization opportunities.

In order to allow active messages to directly express algorithms without requiring external sequencing and arbitration, a number of extensions were required. These extensions are discussed in Chapter 3. Additionally, in order to optimize communication performance

[1] "PGAS" here refers to fine-grained remote memory accesses (the basic PGAS model), without later asynchronous extensions.

a variety of transformations were implemented to provide for runtime optimization of applications. This allows transformations to leverage information about applications which is unavailable at compile time.

The synthesis of these approaches yields a graph library that allows algorithms to be declaratively specified in a message-driven style with most or all of the control flow encapsulated in message handlers. This specification exposes the maximum asynchrony and parallelism present in an application in a fashion that can be effectively leveraged at runtime to utilize both shared- and distributed-memory parallel resources using a single programming abstraction. Commonly applied optimizations are encapsulated in a runtime layer that maps these algorithm specifications to efficient implementations. This combination of declarative specification and separate optimization combines the best features of the high-level systems discussed with a runtime designed to support fine-grained, message-driven computation.

This design gains programmability from related approaches which utilize a single communication abstraction and logical shared address space. Separately from program specification, the runtime component utilizes information about data location to generate optimized communication patterns. The underlying execution model generalizes one-sided operations to user-defined message handlers in a similar fashion to that in which Transactional Memory generalizes atomic operations to arbitrary transactions.

3

Generalized Active Messages

Active messages are especially well-suited to graph computation because they allow the fine-grained dependency structure of graph computations to be captured directly. This dependency structure is discovered at runtime and is defined by the structure of the input graph as well as its vertex and edge metadata. The ideal active message library for implementing graph algorithms would possess low overheads which would allow optimizations such as message coarsening to be deferred until runtime. It would also have expressive semantics which would allow the bulk of the control flow to be defined in the messages themselves rather than relying on external message buffering.

Runtimes for languages such such as UPC and Co-Array Fortran, a number of MPI implementations, and object-based programming systems all rely on active messaging for

their low-level communications. Unfortunately, using these active message implementations directly in an application is difficult because the interfaces to existing active message systems are either too low-level or impose unreasonable overheads for fine-grained applications. A generalized active message library, AM++, was designed to fill this void and allow fine-grained computations to be expressed directly with low overhead. AM++ is intended for use by non-traditional HPC applications and targets the "middle ground" between low-level active message libraries such as GASNet [**26**] and object-based runtime systems such as Charm++ [**97**]. Particular features of AM++ include the following:

- Allowing message handlers to themselves send arbitrary messages, simplifying the expression of some applications.
- A multi-paradigm implementation, including object-oriented and generic programming, to balance runtime flexibility with high performance.
- Type safety provided for messages.
- A modular design enabling configurability by the user without substantial application modification.
- A design enabling compiler optimizations such as inlining and vectorization of message handler loops, and multi-threaded message handlers.
- Flexible message coalescing and redundant message elimination.
- Performance competitive with existing AM systems. Graph algorithms written using AM++ have higher performance than those using the previous, ad hoc AM framework in the Parallel Boost Graph Library [**78**].

3.1. Design Philosophy

AM++ is designed to be especially flexible, without sacrificing performance. The essential features include a mix of runtime and compile-time configurability in order to trade off runtime flexibility for performance overhead. Another feature is extensive type safety: message data being sent, as well as the handlers receiving messages, use well-defined types to enable both better compile-time error checking and the potential for compiler optimizations. Part of this potential is that message handlers for individual messages within a coalesced block can be inlined into the loop over the whole block (with further optimizations possible), a feature that is difficult to achieve with indirect calls through function pointers registered at runtime. Message coalescing itself can be customized as well: for example, when a particular kind of message is idempotent, duplicate messages can be eliminated by the library—saving the user the effort (and code complexity) of removing them in their application. AM++ also allows handlers to invoke arbitrary actions, including sending active messages (to an arbitrary depth), as well as allocating and freeing memory. One consequence of nested messages is that detecting when all messages have been handled is more difficult (the termination detection problem); we allow the user to specify the algorithm to use and the level of nesting required to trade off flexibility (in nesting depth) for performance (simpler algorithms for limited depths).

AM++ is built with a layered design, as shown in Figure 3.1. To blend performance and flexibility, the lower-level communication layers use an object-oriented design for greater configurability at runtime (for example, an individual subroutine can add message handlers for its own use, with those handlers removed when the subroutine ends). The assumption at the lowest level is that data blocks are typically large (due to the coalescing of messages) and so the overheads of such a design will be amortized across a large number of messages. The upper layers of AM++, however, handle the individual messages; an application will send many of those small messages, making performance much more important. These layers use C++ generic programming techniques to allow compile-time configuration without a loss of performance. The compiler can then use static knowledge

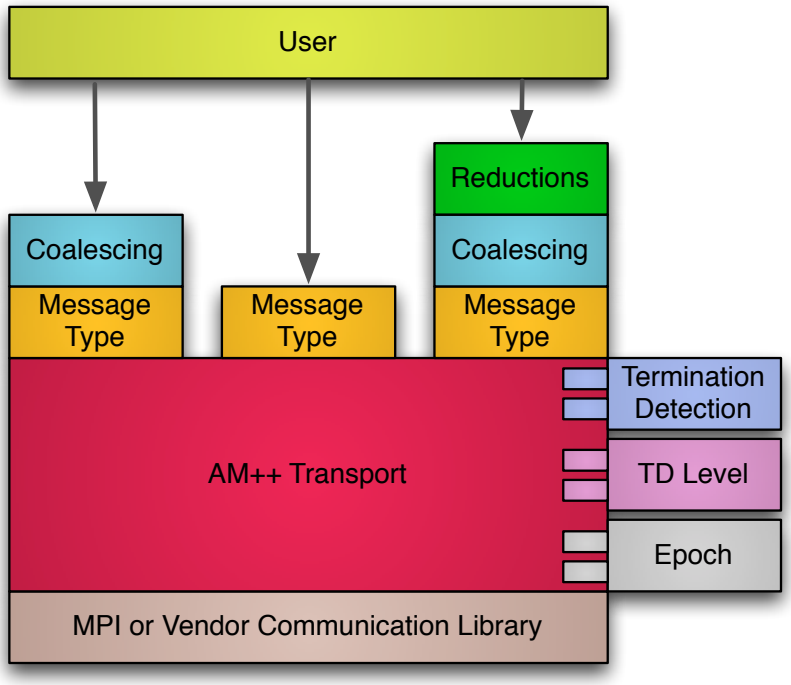


FIGURE 3.1. Design of AM++.

of the configuration of these levels to optimize the code that handles individual messages more effectively.

An AM++ program incorporates the following steps, also shown in Figure 3.2:

(1) Creating a transport to provide low-level communication interfaces (Section 3.2).
(2) Creating a set of message types—type-safe handlers and send operations—within that transport.
(3) Optionally constructing coalescing layers to group messages for higher throughput (Section 3.3).
(4) Beginning a message epoch (Section 3.4).
(5) Sending active messages within the epoch (triggering remote operations in those messages' handlers).
(6) Ending the message epoch, ensuring all handlers have completed.

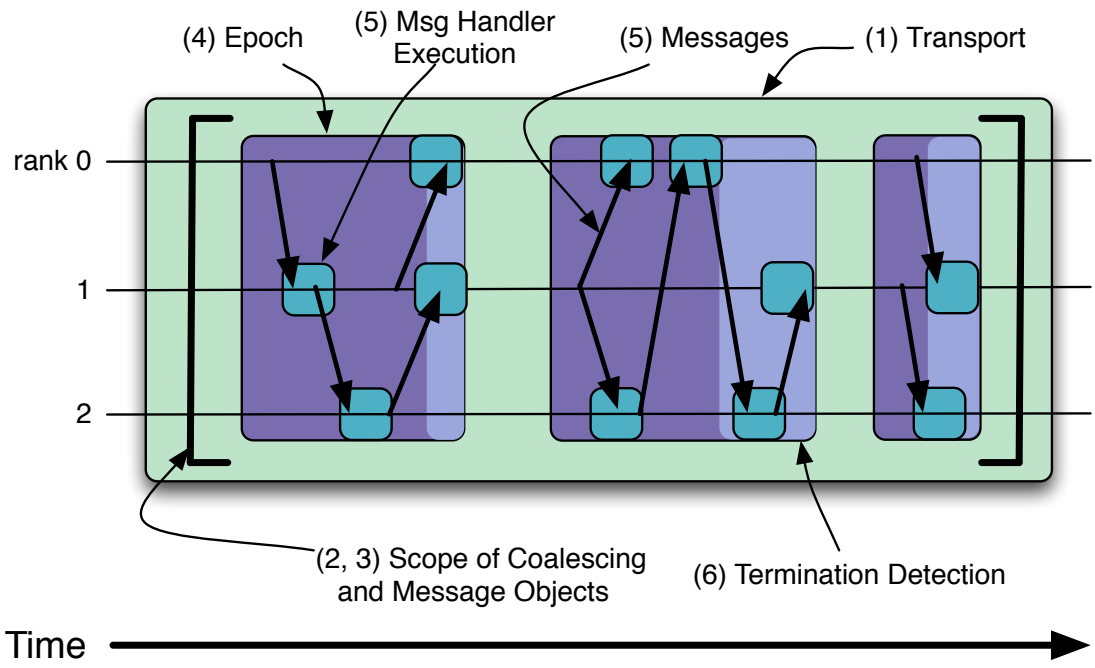


FIGURE 3.2. The overall timeline of a program using AM++. Parenthesized numbers refer to the preceding list of steps.

3.2. Message Transport

At the lowest level, active messages are sent and received using *transports*. These are abstractions that represent sets of handlers, with the ability to add and remove handlers dynamically. The transport manages termination detection and epochs, as described in Section 3.4. Transports also provide various utility functions, such as accessing the number of nodes and the current node's rank.

Several transports can be active at one time, but they will not interact; waiting for a message in one does not test for messages received by the others, which may lead to a deadlock. In this way, transports are somewhat similar to MPI communicators, but more independent. MPI communicators interact in the sense that they all belong to the same MPI implementation, and thus share message progression; AM++ transports are completely independent and so progress on one does not necessarily imply progress on others. Section 3.5 contains more details of progress semantics in AM++.

AM++'s current low-level transport is built atop MPI for portability. However, nothing in the model is tied to MPI or its semantics. Transports could also be built to use lower-level AM libraries such as GASNet [**26**] or direct network interface libraries such as InfiniBand verbs (OFED) or MX. The MPI transport is thread-safe assuming thread-safety in the underlying communication libraries. In our MPI-based implementation, we use MPI datatypes to support heterogeneous clusters without needing explicit object serialization; serialization could be added later to handle more complicated user-defined data types and communication libraries that do not support heterogeneity.

In order to send and handle active messages, individual *message types* must be created from the transport. A transport, given the type of data being sent and the type of the message handler, can create a complete message type object. That message type object then stores the handler and provides a type-safe way to send and handle messages. The interface between the message type object and the underlying transport is internal; the user only sees the methods provided by the message type and its calls to user-defined handlers. When a message is sent, its type is checked to ensure it is valid for that message type object. Messages are also type-checked and necessary conversions are performed before their handlers are called so that handlers do not need to perform type conversion internally. Message type objects are required to be created collectively (i.e., by all processes in the transport at the same time) across nodes, and so no type checking is done between different nodes. Any allocation of message ID numbers or similar objects is done internally by the message type objects; users do not need to do anything to avoid message ID conflicts, even across separate libraries using the same transport. An example of setting up an AM++ transport, creating a message type (built by a coalescing layer), and sending a message is shown in Figure 3.3.

At the transport level, message coalescing and other combining techniques are not applied, and so messages consist of buffers (arrays) of small objects. Sending messages is fully asynchronous to enable overlap of communication and computation, and so the application (or upper messaging layer) passes in a reference-counted pointer that represents ownership of the data buffer. When the buffer can be reused, the reference count of the

```
1  // Create MPI transport
2  mpi_transport trans(MPI_COMM_WORLD);
3
4  // Build coalescing layer (the underlying message
5  // type object is created automatically)
6  basic_coalesced_msg_type<my_message_data, my_handler, mpi_transport>
7    msg_type(trans, /* Coalesced message size */ 256);
8
9  // Set message handler
10 msg_type.set_handler(my_handler());
11
12 // Declare that handler will not send messages
13 // (0 is nested message level)
14 scoped_termination_detection_level_request<mpi_transport>
15   td_req(trans, 0);
16
17 { // The block represents a single epoch
18   scoped_epoch<mpi_transport> epoch(trans);
19   if (trans.rank() == 0)
20     msg_type.send(my_message_data(/*data*/ 1.5), /*destination*/ 2);
21 }
```

FIGURE 3.3. Example usage of AM++.

pointer is decremented, possibly calling a user-defined deallocation function contained in the pointer. This function is effectively a callback indicating when it is safe to reuse the buffer. Buffers for received messages are automatically allocated and deallocated by the message type object using a memory pool; the user is thus required to declare the maximum message size for each message type.

3.3. Message Set Optimization

The modularity of AM++ enables several layers to be built atop the basic message type objects. These layers form generalizations of message coalescing, and thus are important to increase message transfer rates. Because the layers work on single, small messages that are sent frequently, compile-time configuration (expressed using C++ templates) is used to define the compositions of components. Increasing the amount of compile-time knowledge of the application's messaging configuration provides more information to the compiler, allowing a greater level of optimization and thus lower messaging overheads. Here, we describe two layers that can be built atop the basic message types: message

coalescing and vectorization and optimization of message handlers. Further optimizations will be explored in Chapter 4.

3.3.1. Message Coalescing. Message coalescing is a standard technique for increasing the rate at which small messages can be sent over a network, at the cost of increased latency. In AM++, a message coalescing layer can be applied atop a message type; the basic message type does not need to know whether (or what kind of) coalescing is applied. Coalescing can be configured separately for each individual message type. Each message coalescing layer encodes the static types of the messages it sends and of the handler that it calls, leading to the optimizations described in Section 3.3.2.

In our implementation, a buffer of messages is kept for each message type that uses coalescing and each possible destination. Messages are appended to this buffer, and the entire buffer is sent (using the underlying message type) when the buffer becomes full; explicit flush operations are also supported. A timeout is not currently provided, but could be added in the future; a user could also create his/her own coalescing layer supporting a timeout. When a buffer of messages arrives at a particular node, the lower-level handler (provided by the coalescing layer) runs the user's handler on each message in the buffer. The one complication in coalescing is its interaction with threads. We have two implementations for thread-safety of message sends: one uses an explicit lock, while another uses the CPU's atomic operations to avoid explicit locking in the common case (the buffer has been allocated and is not full) and spins in other cases. Message receiving is thread-safe without any difficulty: a single thread is used to call handlers for all of the messages in one buffer, while other threads can simultaneously call handlers for messages in other buffers.

3.3.2. Message Handler Optimizations. The static knowledge of message types and handlers available to coalescing implementations enables several potential optimizations that are not available to other, more dynamic, AM frameworks such as GASNet. The simplest is that the user handler is called for each message in a buffer; this loop's body is simply a call to a known function that can be inlined. After inlining, other loop optimizations can be applied, such as loop unrolling, vectorization, and software pipelining. Hardware

acceleration, such as in a GPU, could potentially be used for especially simple handlers, as could execution in the Network Interface Controller (NIC) hardware.

One particular way in which the handlers for messages in a single buffer can be optimized is through fine-grained parallelization. For example, an OpenMP directive could be applied to mark the handler loop as parallel, or a more explicit form of parallelism could be used. One benefit of AM++'s modularity is that parallelism granularity is message-type-specific and can be varied with minimal application modifications. The fine-grained model is different from handling different buffers in different threads, and its finer granularity of parallelism requires low-overhead thread activation and deactivation. This granularity is applied within the AM system, without requiring modifications to application code (other than thread-safety of handlers).

3.4. Epoch Model and Termination Detection

In AM++, periods in which messages can be sent and received are referred to as *epochs*. Our notion of an epoch is similar to an active target access epoch for MPI 2.2's one-sided operations [**125**, §11.4] in that all AM operations must occur during the epoch, while administrative operations such as modifying handlers and message types must occur outside the epoch. In particular, all nodes must enter and exit the epoch collectively, making our model similar to MPI's active target mode. Epochs naturally structure applications in a manner similar to the BSP model [**164**], except that AM-based applications are likely to do much or all of their computation within the communication regions. AM++ does not provide a guarantee that active messages will be received in the middle of an epoch, but does guarantee that the handlers for all messages sent within a given epoch will have completed by the end of that epoch. This relaxed consistency model allows system-specific optimizations.

One feature that distinguishes AM++ from other AM libraries such as GASNet is that it gives much more flexibility to message handlers. In particular, handlers can themselves send active messages to arbitrary destinations, and are not limited to only sending replies. A major benefit of this capability is that it greatly simplifies some uses of AM; for example,

a graph exploration (such as that shown in Section 3.6.2) can be directly implemented using chained message handlers in AM++, while a separate queue, including appropriate locking, is required when direct sending is forbidden.

More sophisticated message handlers, on the other hand, cannot be implemented using system interrupt handlers; see Section 3.5 for more information on this trade off. Traditional AM systems such as GASNet require each handler to send at most one reply; this restriction can be used to avoid deadlocks [**165**]. Also, end-of-epoch synchronization becomes more difficult with unrestricted handlers: in the event that a message handler itself sends messages, all of these *nested messages* will also need to handled by the end of the epoch. Distributed termination detection algorithms [**67**, **121**] are required to determine this property reliably in a distributed system.

The literature contains several algorithms for termination detection with arbitrary chains of nested messages; our current implementation uses the four-counter algorithm described in [**121**, §4] with a non-blocking global reduction (all-reduce) operation from libNBC [**85**] to accumulate the counts; this approach is similar to the tree-based algorithm by Sinha, Kalé, and Ramkumar [**150**]. General termination detection algorithms allowing arbitrary nested messages can be expensive, however; in the worst case, the number of messages sent by the program must be doubled [**37**, §5.4]. AM++ allows the user to specify the depth of nested messages that will be used; finite depths allow simpler algorithms with lower message complexity to be used, such as generalizations of the algorithms by Hoefler et al. [**86**]. Users can add and remove requests for particular depths, with the largest requested depth used; these requests can be scoped to a particular region of code (see Section 3.5.1).

Because many termination detection algorithms are based on message or channel counting, a user-defined integer value can be summed across all nodes and then broadcast globally without extra messages in many cases. AM++ supports this operation as an optional part of ending an epoch; termination detection algorithms that do not include that feature automatically would need to do an extra collective operation if a summation is requested in a particular epoch. This feature is useful for graph algorithms; many algorithms

consist of a phase of active messages followed by a reduction operation (for example, to determine if a distributed queue is empty), and thus benefit from this capability in the AM library.

3.5. Progress and Message Management

Active messages can be received and processed in several places in the application. For example, GASNet can run message handlers inside a signal handler, leading to restrictions on what operations can be done in a handler (except when a special region is entered) [**26**]. For flexibility, and to avoid the use of operating-system-specific features, we run all handlers in normal user mode (i.e., not in a signal context). As we currently build on top of MPI, our own code also runs as normal user code. Another option would be to use a background thread to process messages, as GASNet allows. We do not mandate the use of threads—and thus thread safety—in user applications, and so we do not create a background thread automatically. The user could spawn a background thread that simply polls the AM engine in order to handle message progress. If there is no progress thread, the user must periodically call into AM++ to ensure that messages are sent and received. In the worst case, ending an epoch will provide that assurance. MPI and GASNet also use this model of progress, and so is likely to be familiar to users.

Our approach to threads is to allow but not mandate them. Actions such as registering and unregistering message types are not thread-safe; it is assumed that the user will perform them from only a single thread. A single epoch can be begun by several threads on the same node; the epoch must then be ended by the same number of threads. Actions such as message sends and handler calls are thread-safe for all of AM++'s standard coalescing layers and duplicate message removers. A compile-time flag can be used to disable locking when only one thread is in use. Our model is thus similar to MPI's *MPI_THREAD_MULTIPLE* or GASNet's *PAR* mode. For our MPI-based transport, we normally assume *MPI_THREAD_MULTIPLE* in the underlying MPI implementation, with a compile-time flag to switch to *MPI_THREAD_SERIALIZED* for MPI implementations that do not support multiple threads in the library simultaneously.

3.5.1. Administrative Objects. Resource Acquisition is Initialization (RAII) [**156**] techniques are used throughout AM++ to simplify applications. For example, handlers are registered by message type objects, and automatically unregistered when the message type is deleted. Requests for particular levels of nested messages (termination detection depth) are also managed using this technique. Epochs are scoped in a similar manner, except that a user-defined summation at the end of an epoch requires it to be managed manually. RAII is used internally for the management of several other types of registration and request objects. RAII prevents many types of resource leaks by giving the compiler responsibility for ensuring exception-safe deallocation of objects.

3.6. Benchmarks

In order to demonstrate that AM++ satisfies the dual goals increased flexibility without sacrificing performance we present a range of experiments. Latency and bandwidth microbenchmarks compare AM++ to GASNet in order to demonstrate that AM++ does not impose unreasonable overheads compared to lower-level communication frameworks. A simple graph exploration benchmark (breadth-first search) compares implementations using AM++ to implementations using GASNet and an existing implementation in the Parallel BGL. This application benchmark demonstrates that AM++'s ability to automatically overlap communication and computation provides better performance and scalability vs. more coarse-grained approaches. Finally, the single-source shortest paths implementation in the Parallel BGL is compared to an AM++ version in a similar style. The single-source shortest paths problem provides the opportunity to not only overlap communication and computation, but to execute the received message and free the associated data before the end of the message epoch.

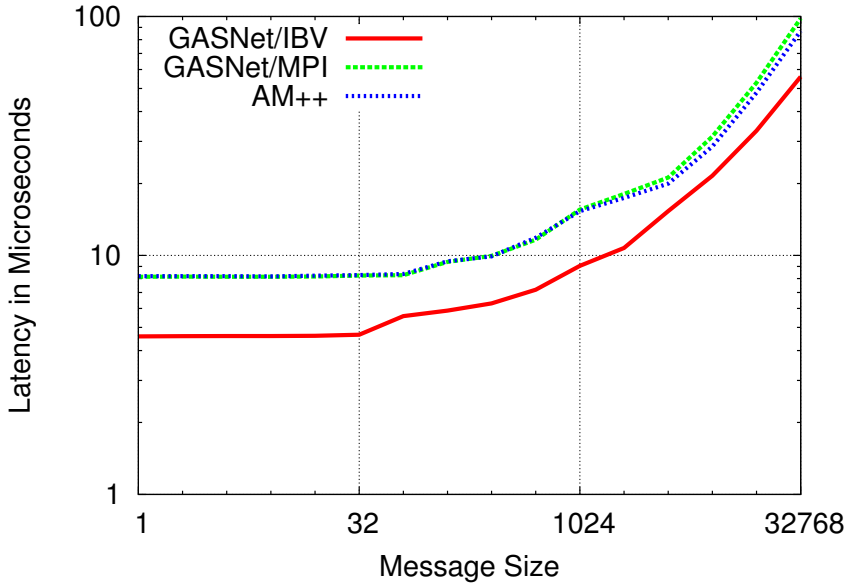
We present performance results on Odin, a 128-node InfiniBand cluster (Single Data Rate). Each node is equipped with two 2 GHz Dual Core Opteron 270 CPUs and 4 GiB RAM. We ran our experiments with Open MPI 1.4.1, OFED 1.3.1, and GASNet 1.14.0. We used the latency/bandwidth benchmark `testam` included in GASNet (recording section L, as it is closest to the messaging model AM++ uses) and a simple ping-pong scheme for

AM++. The compiler used was g++ 4.4.0. We have observed that multiple MPI processes per node, each with a smaller portion of the graph data, is less efficient than a single MPI process with more graph data. This effect is likely due to the increased communication caused by partitioning the graph into more pieces and the overhead of communicating through MPI to perform work on graph data that is present in a node's local memory. Thus, all of our tests used a single MPI process per node.

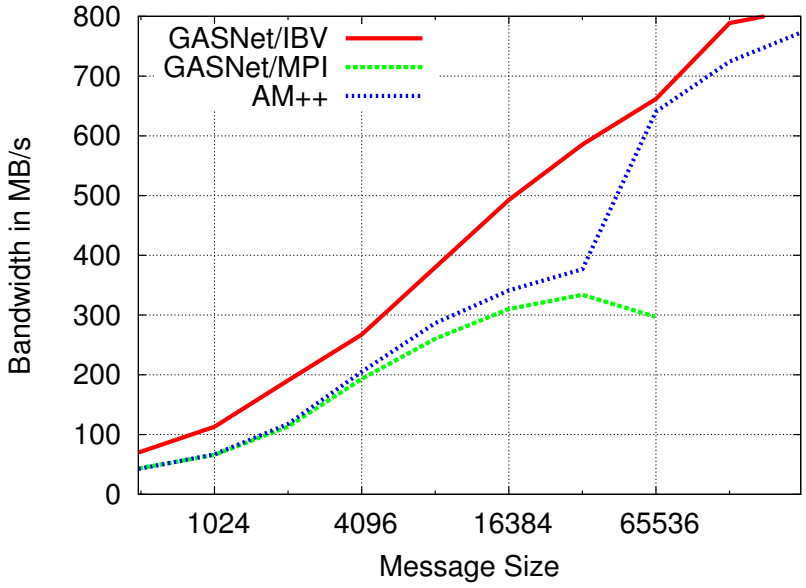
3.6.1. Microbenchmark: Latency and Bandwidth. Figure 3.4 shows a comparison of AM++ and GASNet with respect to latency and bandwidth. We remark that the AM++ implementation runs on top of MPI while the best GASNet implementation uses InfiniBand (OFED) directly. We see a minimal difference in latency between GASNet over MPI and AM++ ($< 0.6\mu s$) and a slightly larger difference ($< 3.1\mu s$) between the optimized GASNet over InfiniBand and AM++ versions. Our design also allows for an implementation on top of OFED, however, we have limited our focus to MPI for increased portability. As with latency, GASNet over InfiniBand performs slightly better than AM++ with respect to bandwidth. We note that GASNet's MPI conduit does not support messages larger than 65 kB.

Our modular design allows for runtime-configurable termination detection and uses dynamic memory management (but with heavy use of memory pools) for send and receive buffers. These capabilities impose a small overhead on all messages, primarily due to virtual function calls.

3.6.2. Kernel Benchmark: Graph Exploration. In order to demonstrate the benefits of sending arbitrary active messages from within a handler context, we now discuss a simple graph exploration kernel. The graph exploration algorithm is similar to breadth-first search in that each vertex of a graph reachable from a given source is explored exactly once, with a color map used to ensure this property. Unlike BFS, however, graph exploration does not place any constraints on the vertex order, and thus avoids internal synchronization.



(A) Latency.



(B) Bandwidth.

FIGURE 3.4. GASNet over MPI or InfiniBand vs. AM++ over MPI with a ping-pong benchmark.

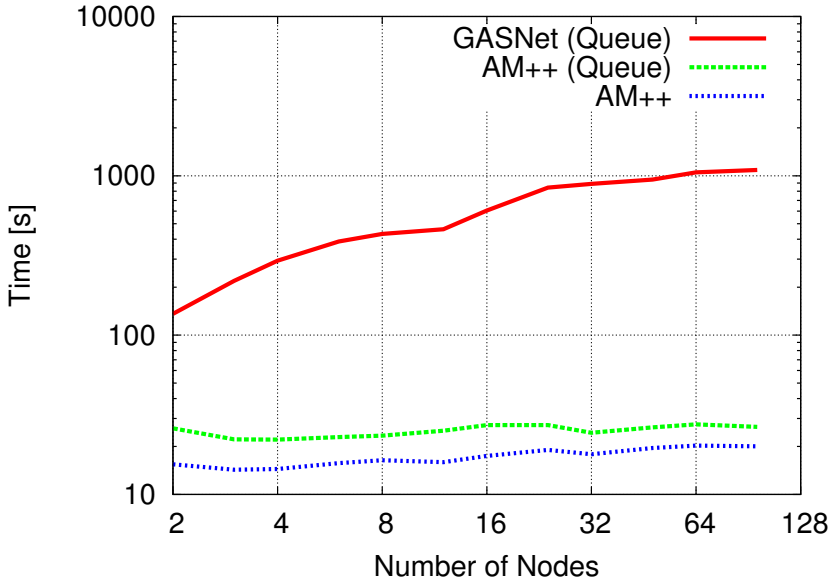
We implemented two versions of the kernel. In the first, *queue-based* implementation, the active message handler adds remote vertices that have been discovered to the local

queue for processing. The second, *nested-message* implementation's active message handler performs the exploration step and sends all remote messages immediately. The nested-message implementation thus has significantly lower overhead (fewer push/pop operations) and is much more agile because it generates new messages without the delay of a queue. However, it requires that handlers be able to send messages to arbitrary processes. While we needed to use the queue-based approach for GASNet, we implemented both versions for AM++.

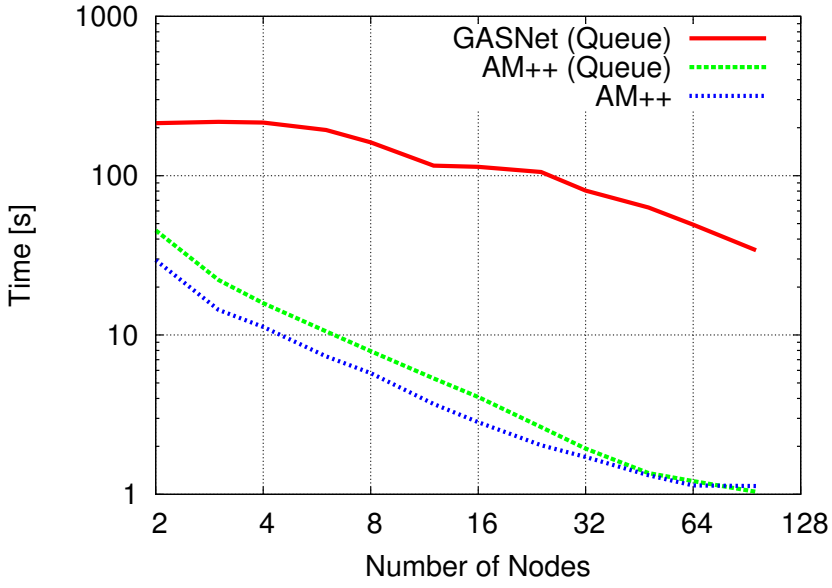
Figure 3.5 shows the results of the benchmark. For each run, a random graph was generated using the Erdős-Rényi model, and then a Hamiltonian cycle was inserted to ensure that the graph was strongly connected. Erdős-Rényi graphs are characterized by a high surface-to-volume ratio and normally-distributed vertex degrees. We seeded the random number generator statically to ensure reproducible results.

One cause for the large performance difference between AM++ and GASNet queue-based implementations is that AM++ includes message coalescing, while GASNet does not. AM++ is designed for user applications that benefit from coalescing, even when it adds latency, while GASNet is designed for optimal latency and assumes features such as coalescing will be built atop it. Although we could have implemented coalescing on top of GASNet, it would have complicated the application; the AM++ version sends its messages through a layer that applies coalescing automatically.

Another difference, as explained above, is that in the nested-message implementation, the handler explores all local vertices reachable from the received vertex and sends messages for any remote neighbors of those vertices. A local stack is used to avoid overflowing the system stack, with a separate stack for each handler call. The queue-based implementations, on the other hand, use a global stack to communicate between the message handler and the code's main loop. The handler pushes vertices onto the queue because it cannot send messages directly. The main loop then processes the elements on the queue, sending messages for those vertices' remote neighbors.



(A) Weak Scaling (5M vertices per node).



(B) Strong Scaling (15M vertices total).

FIGURE 3.5. Comparison of GASNet and AM++ with a simple graph exploration benchmark.

The third difference is termination detection. In AM++, termination detection is included in the library, while the user must implement it on top of GASNet. For the queue-based implementations, we chose a simple scheme based on message counting: each sent

message increases a local counter and the handler generates a reply message that decrements the counter; termination is detected if all counters reach zero. This scheme adds additional overhead in comparison to the optimized termination detection in AM++, used in the nested-message implementation.

Thus, we note that the huge performance differences between AM++ and GASNet on this benchmark stem from the different goals of the two libraries. While GASNet is intended as a low-level interface for parallel runtimes and thus tuned for the highest messaging performance, AM++ is more user-friendly and supports direct implementation of user algorithms and thus enables higher performance with less implementation effort (the nested-messaging AM++ implementation has 40% fewer lines of code than the queue-based GASNet version). In this sense, our results in this section emphasize the different purposes of the two libraries rather than fundamental performance differences (as one could, with significant implementation effort, reproduce most of the features of AM++ on top of GASNet).

3.6.3. Application Benchmarks. Distributed-memory graph algorithms are an excellent application use case for active messages as they can be highly asynchronous and extremely latency-sensitive. The Parallel BGL **[78]** is one of the most successful publicly available distributed-memory graph libraries. The Parallel BGL includes the concept of a Process Group which abstracts the notion of a set of communicating process. The *MPI Process Group* is one implementation of the Process Group concept; it performs message coalescing, early send/receive, and utilizes asynchronous MPI point-to-point operations and traditional collectives for communication.

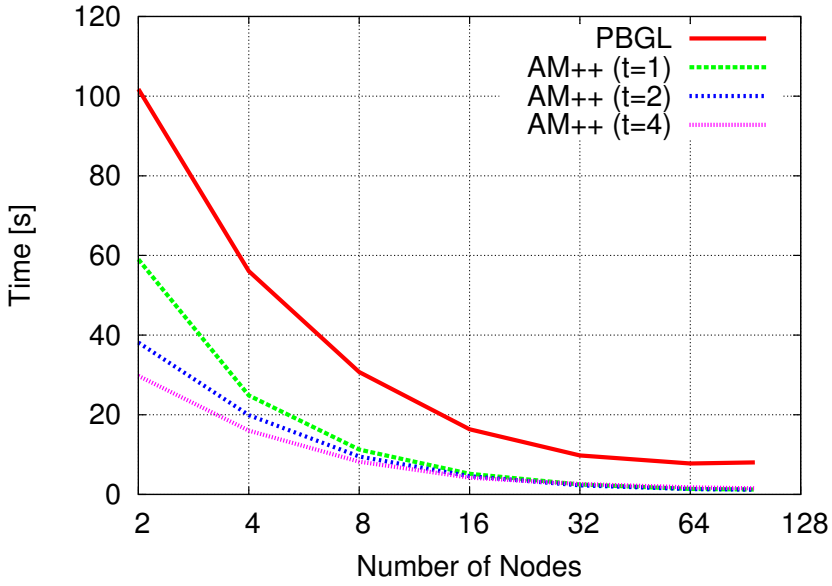
One key benefit of phrasing graph algorithms as message-driven computations is that the work in the algorithm is broken into independent quanta, making fine-grained parallelism straightforward to leverage. Because AM++ is both thread-safe and, more importantly, efficient in the presence of threads, implementing distributed-memory algorithms that utilize fine-grained parallelism on-node was straightforward. This is not the case for

the Parallel BGL: the *MPI Process Group* is not thread-safe and would likely require significant effort to be made so. AM++ operations are performed directly by all threads involved in the computation, rather than being funneled to a single communication thread or serialized. While it would be possible to simply run additional processes to use multiple cores on each node, communicating with other processes on the same node using MPI is less efficient than communicating with other threads in the same address space. Furthermore, additional processes would require further partitioning the graph and associated data structures, leading to poorer load balancing.

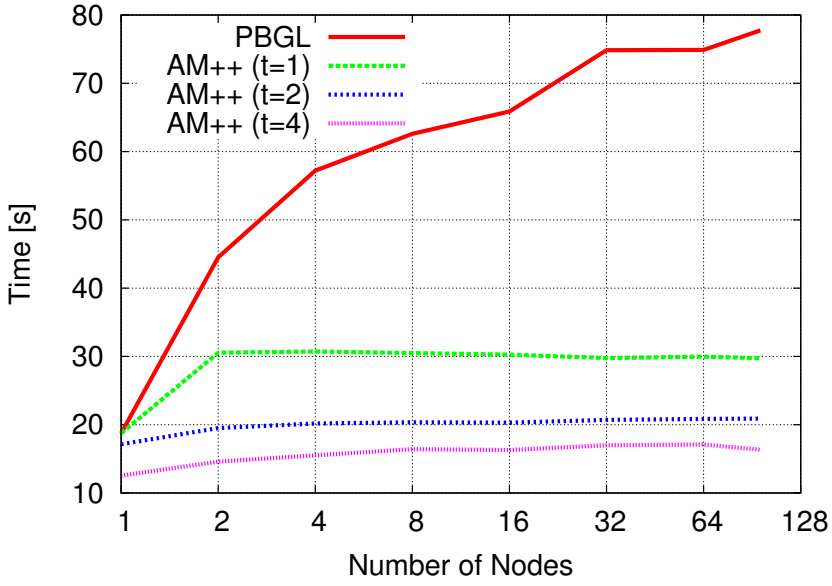
We benchmark two graph algorithms from the Parallel BGL implemented using the *MPI Process Group* against those same algorithms implemented using AM++ and reusing code from the Parallel BGL extensively. The AM++ implementations are benchmarked utilizing various numbers of threads to demonstrate the effectiveness of combining fine-grained parallelism with AM++. The latest development version of the Parallel BGL and a pre-release version of AM++ were utilized in these tests. We present results on Erdős-Rényi graphs as in the graph exploration kernel.

3.6.3.1. *Breadth-First Search.* Breadth-First Search (BFS) is a simple, parallelizable graph kernel that performs a level-wise exploration of all vertices reachable from a single source vertex.

Figure 3.6 shows the performance of the Parallel BGL's BFS implementation, as well as a BFS implemented using similar but thread-safe data structures and AM++. In the case of the AM++ implementation, all data structures that support concurrent access are lock-free. Figure 3.6a shows the performance of both implementations on a constant-size problem as the number of cluster nodes is increased. The Parallel BGL implementation stops scaling at 64 nodes while the AM++ implementation's runtime continues to decrease in all cases as nodes are added. The AM++ implementation also benefits from additional threads in all cases except with 4 threads on 32–96 processors, likely due to contention as the amount of work available on each node decreases.



(A) Strong scaling (2^{27} vertices and 2^{29} edges).



(B) Weak Scaling (2^{25} vertices and 2^{27} edges per node).

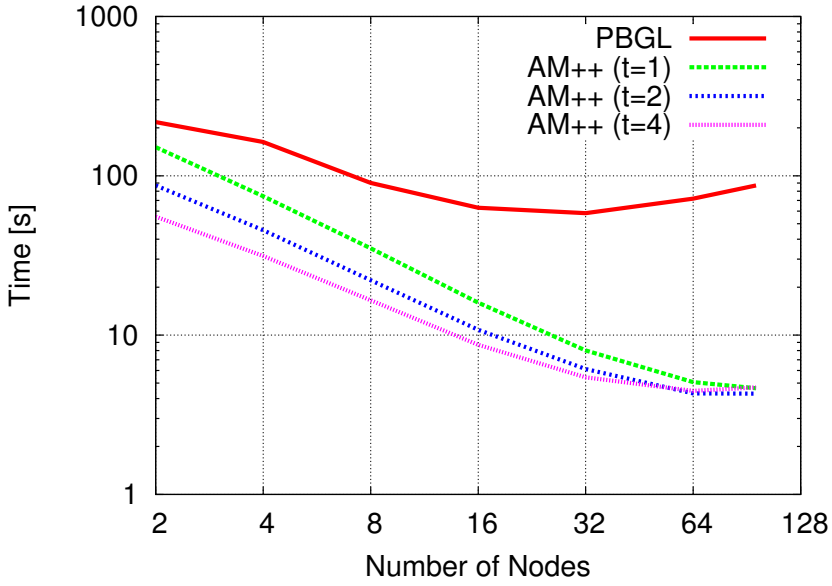
FIGURE 3.6. Performance of the Parallel BGL (using the *MPI Process Group*) and AM++ with various numbers of threads performing a parallel breadth-first search.

Figure 3.6b shows the performance of both implementations on a problem where the work per node remains constant. The Parallel BGL calls the sequential BGL implementation of BFS in the single-processor case, as does AM++ when only one thread is present. Not only does the AM++ implementation perform better and benefit from fine-grained parallelism as threads are added, it also exhibits no increase in runtime as the problem size is increased. This is expected as BFS performs $\mathcal{O}(|V|)$ work, and so increases in the problem size (and work) are balanced by increases in the number of processors available.

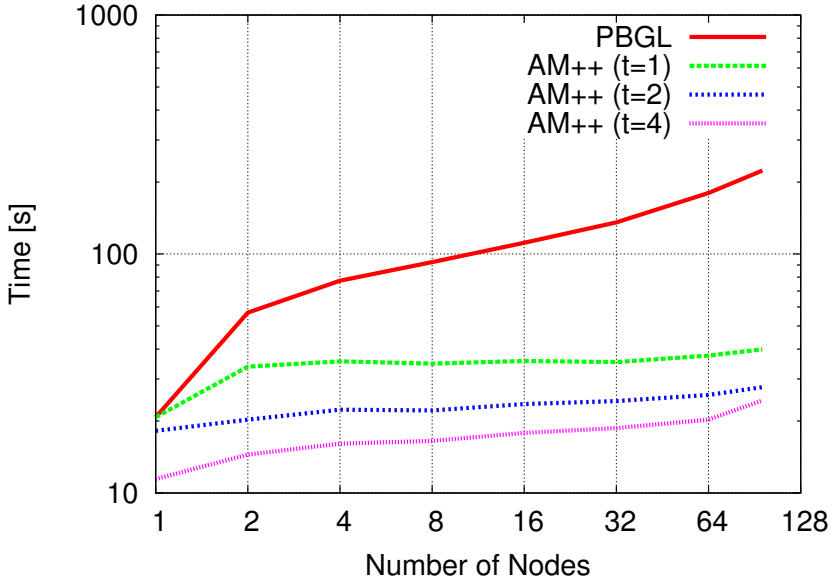
We applied duplicate message removal to our BFS implementation but saw no resulting performance benefit. Our tests used a high-speed interconnect, and the BFS handler is fast for redundant messages (because of the early color map check), so the cost of the cache is likely too large compared to the work and bandwidth it saves. Redundant message combining techniques are likely to show a benefit for other applications and systems, however; more expensive handlers, slower networks, and larger messages obtain greater benefit from removing excess messages. One advantage of AM++ is that various caching schemes can be plugged in without extensive modifications to user code; the user's message type is wrapped in the appropriate caching layer and then sends and handler calls occur as usual.

3.6.3.2. *Parallel Single-Source Shortest Paths.* Single-Source Shortest Paths (SSSP) finds the shortest distance from a single source vertex to all other vertices. A variety of SSSP algorithms exist. The classic algorithm by Dijkstra is *label-setting* in that distance labels are only written once, leading to an inherently serial algorithm. *Label-correcting* algorithms such as **[43, 124]** are common in parallel contexts, as label-setting algorithms do not parallelize well. Edmonds et al. have found Δ-Stepping **[124]** to be the best-performing parallel algorithm on distributed-memory systems **[60]**.

Figure 3.7 shows the performance of the Parallel BGL's Δ-Stepping implementation vs. the same algorithm implemented with AM++. Slight modifications were performed to the Δ-Stepping algorithm which reduce work efficiency and increase redundant communication but which show dramatic performance improvement in practice. In the strong scaling chart in Figure 3.7a the Parallel BGL implementation scales inversely between 32 and 96



(A) Strong Scaling (2^{27} vertices and 2^{29} edges).



(B) Weak Scaling (2^{24} vertices and 2^{26} edges per node).

FIGURE 3.7. Performance of the Parallel BGL (using the *MPI Process Group*) and AM++ with various numbers of threads computing single-source shortest paths in parallel using Δ-Stepping.

nodes while the AM++ implementation displays an increase in performance as both additional nodes and threads are added in almost all cases. In Figure 3.7b, the AM++-based algorithm once again displays almost flat scaling (the runtime is expected to increase proportionally to the problem size because SSSP performs $\mathcal{O}(|V| \, log \, |V|)$ work while the number of nodes increases linearly). The Parallel BGL implementation also exhibits an increase in runtime as the problem size grows, albeit with a significantly steeper slope. AM++ is again able to benefit from additional threads in this case.

Much of the difference in absolute performance is accounted for by the difference in serialization and memory management between the two implementations. AM++'s MPI transport uses MPI datatypes to describe the formats of messages, and the buffer used to accumulate coalesced messages is simply an array of the appropriate type; serialization is done by the MPI implementation if required. On the other hand, Parallel BGL uses explicit serialization using the Boost.Serialization library. In particular, Parallel BGL's message coalescing uses *MPI_Pack* to append to a data buffer, leading to extra function calls and data structure traversals. A 64-node profile of Δ-Stepping on a 2^{27}-vertex, 2^{29}-edge graph (the size used in the strong scaling test) shows that 29% of the runtime is spent in these functions. The Parallel BGL approach can send more general objects, including those that cannot be directly described by MPI datatypes; however, AM++ could use a more flexible serialization library while keeping arrays as coalescing buffers.

Another performance difference between the two implementations is in memory management. AM++ uses various memory pools to reduce the use of MPI's memory management functions (*MPI_Alloc_mem* and *MPI_Free_mem*); these functions are slow for some interconnects that require data to be pinned in memory. Parallel BGL, on the other hand, uses the MPI memory functions more directly and more often, leading to reduced performance; 31% of the profile described above is spent in these calls.

3.7. Conclusion

AM++ provides an active message implementation with performance characteristics comparable to low-level communication frameworks but with a few key extensions. The

ability of messages to themselves send messages to unbounded depth, with termination detection built into the library, allows control to be expressed entirely in active messages without the need for external buffering and ordering constructs. The modular design of AM++ allows for flexible configuration by user-level algorithms and libraries while preserving high performance. Finally, as we will see in Chapter 4, abstracting the semantics—individual active messages—from the underlying mechanism by which the messages are processed and executed allows a host of optimizations to be performed.

4

Separating Specification from Implementation

One of the key goals of this dissertation is to define parallel programming abstractions that allow graph algorithms to leverage the increasing variety of hardware parallelism available. Each of these classes of parallel hardware currently require separate programming models: MPI for distributed-memory parallelism, threading for shared-memory parallelism, and a variety of environments for the increasing variety of accelerators available. This is by no means an exhaustive list. Each of these models come with their own parallel programming challenges. Unifying two or more of them magnifies this challenge and supporting the full cross-product of possible combinations one-by-one requires a monumental effort even for a single algorithm; much less the broad variety of algorithms a useful library would contain.

Taming this hardware complexity requires the development of programming abstractions that can cross hardware boundaries. These abstractions disentangle the algorithm definition from the implementation which is ultimately executed. By discovering transformations common to a class of algorithms these transformations can be developed separately and suitably parameterized such that they can be shared by many algorithms. This modular design allows the development of a flexible framework which is adaptable to new classes of hardware, new optimization techniques, and new algorithms.

The generalized form of active messages provided by AM++ provide a suitable substrate for implementing parallel graph algorithms. Chapter 3 demonstrated that active messages allow parallelization via both shared- and distributed-memory techniques, a topic that will be further discussed in subsequent chapters. However, naïve implementations would be poorly suited to modern HPC cluster hardware. In this chapter we explore a framework, Active Pebbles [1] [**173**], specialized for data-driven computations such as graph algorithms.

Many data-driven problems can scale on traditional HPC hardware. The difficulty is in expressing fine-grained, irregular, non-local computations in such a way as to be able to fully exploit hardware that was designed for coarse-grained, regular, local computations. This can be done (and has been done), with great difficulty, by hand. However, with homegrown solutions like this, the application developer is responsible for developing all layers of the solution stack, not just the application. Furthermore, the application developer is responsible for re-implementing this entire stack when target platforms change, or when new applications must be developed. In many ways, this is similar to the state of affairs that faced the compute-intensive community prior to the standardization of MPI.

Accordingly, an approach is needed that separates data-driven applications from the underlying hardware so that they can be expressed at their natural levels of granularity, while still being able to (portably) achieve high performance. Active Pebbles defines control and data flow constructs for fine-grained data-driven computations that enable low

[1] The term Active Pebbles expresses the idea that messages are active and independent, but without individual identity (transported and processed in bulk).

implementation complexity and high execution performance. That is, with the Active Pebbles programming model, applications can be expressed at their "natural" granularities and with their natural structures. The Active Pebbles execution environment in turn coalesces fine-grained data accesses and maps the resulting collective operations to optimized communication patterns in order to achieve performance and scalability.

The Active Pebbles model has two distinct aspects: a programming model plus an execution model. The programming model provides fine-grained programmability using features such as fine-grained addressing and describes how algorithms are expressed. The execution model incorporates transformations such as coalescing, routing, and reductions as well as termination detection and describes how algorithms are executed. This separation of specification from implementation provides performance portability for algorithms and allows retroactive optimization and tuning without modifying algorithm specifications.

4.1. Active Pebbles Programming Model

Programming models which are well suited to traditional HPC applications depend heavily on the locality inherent in these applications, in particular, each node communicating with only a few local peers. Coarse-grained approaches based on the BSP "compute-communicate" model thus tend to yield scalable solutions on current system scales.

In contrast, graph applications possess no underlying natural locality which can be determined analytically. The locality information is **irregular**, and embedded directly in the data itself. This locality information describes a dependency graph for computations which is **non-local**, i.e., it does not have good separators [**139**]. To complicate matters further, graph problems tend to be **fine-grained**, i.e., they possess a large number of small objects. Performing efficient static coalescing of these objects into larger, coarser-grained objects is hampered by the irregularity of the problems. In contrast to coarse-grained static (or compile-time) approaches, the Active Pebbles programming and execution model is specifically designed for such fine-grained applications. This execution model performs

transformations at runtime which efficiently map applications expressed in the programming model to the underlying machine in an architecture-aware fashion.

Message passing, an effective programming model for regular HPC applications, provides a clear separation of address spaces and makes all communication explicit. The MPI is the de facto standard for programming such systems [**125**]. However, graph applications need shared access to data structures which naturally cross address spaces. A number of libraries for remote memory access exist, some more suited to the fine-grained, irregular access patterns of graph applications than others. The key requirement for graph applications is that these remote memory accesses must be atomic in the presence of concurrent accesses by multiple processes and must support "read-modify-write" operations (e.g., compare-and-swap, fetch-and-add, etc.). Additionally, only the process performing the updates has knowledge of the memory locations being updated which prevents the process whose memory is the target of the update from performing any sequencing or arbitration of the updates. Finally, some algorithms require dependent updates to multiple, non-contiguous memory locations which must be executed atomically. Implementing these distributed, transactional remote memory updates is extremely challenging for a programming model, especially in a manner which avoids the significant overheads inherent in distributed locking.

MPI-2 One-Sided operations [**125**, ch. 11] and PGAS models [**128**, **163**] attempt to fill the gap left by two-sided MPI message passing by allowing transparent access to remote memory in an emulated global address space. However, mechanisms for concurrency control are limited to locks and critical sections; some models support weak predefined atomic operations (e.g., *MPI_Accumulate()*). Stronger atomic operations (e.g., compare and swap, fetch and add) and user-defined atomic operations are either not supported in current versions or do not perform well. Thus, we claim that these approaches do not provide the appropriate primitives for fine-grained graph applications. The designers of those approaches have also realized these limitations in the original models, leading to proposals such as UPC queues [**94**] and Global Futures [**40**] to add more sophisticated primitive operations (including active messages in some cases) to otherwise PGAS programming models.

MPI-3 [**126**] provides fine-grained completion semantics for remote accesses and stronger atomic operations than MPI-2 [**125**] but still fails to provide generalized remote memory transactions and requires polling by the receiver to observe remote writes.

Given these application requirements and the limitations of existing programming models we chose to develop Active Pebbles [**173**], a new programming and execution model based on active messages. Just as Transactional Memory generalizes processor atomic operations to arbitrary transactions, Active Pebbles generalizes one-sided operations to user-defined pebble handlers. This design allows processor atomic operations such as *compare_and_swap()* or *fetch_and_add()* to be invoked remotely as well as providing the potential to leverage more complex synchronization such as Transactional Memory when they become available. The Active Pebbles programming model takes traditional active messages [**165**] and generalizes them in a number of important ways, as well as providing an implementation designed to handle large numbers of tiny messages (both in data size and in computation) efficiently. The most important of the programming model generalizations is that Active Pebbles allows message handlers to directly send additional messages, to unbounded depth. In many cases this feature removes the need for application-level message buffering.

At the core of the Active Pebbles programming model are *pebbles*, light-weight active messages that are managed and scheduled in a scalable way, and which generally have no order enforced between them. In addition to pebbles, the Active Pebbles model includes *handlers* and *distribution objects*. Handlers are functions that are executed in response to pebbles (or ensembles of pebbles) and are bound to data objects with distribution objects to create *targets*. Pebbles are unordered (other than by termination detection), allowing flexibility in processing.

The termination detection features of Active Pebbles allow quiescence to be detected. In Active Pebbles, all messages must be sent within defined *epochs* (similar to the same concept in MPI-2 One-Sided communication), and all messages globally are guaranteed to have been sent and handled—including those messages sent during handlers invoked by previous ones in the epoch—by the time the epoch completes.

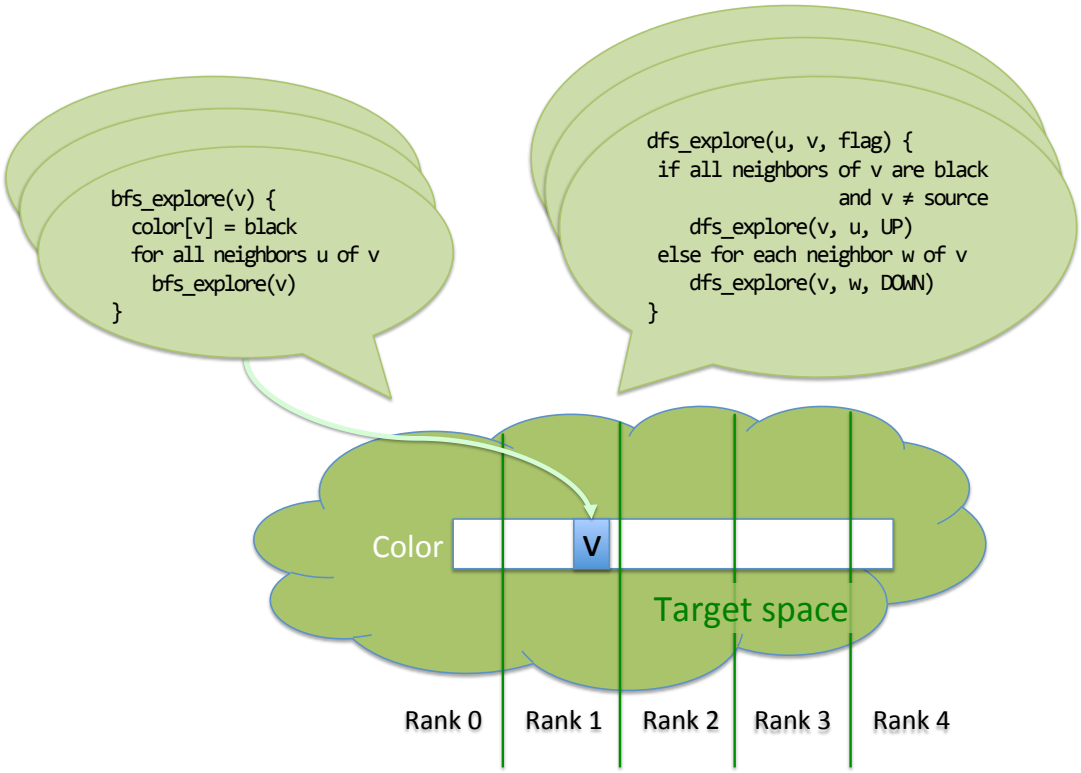


FIGURE 4.1. Active Pebbles programming model.

Figure 4.1 illustrates how the logical target space is distributed across physical ranks according to a distribution object. Light-weight *pebble addressing* allows pebbles to be routed to targets. The underlying source-level construct is called an OwnerMap. In the simplest case, this distribution may be a function evaluated on the input identifier (e.g., a hash). Explicit representation of the distribution is also possible, as are hybrid approaches where blocks of the key space are assigned explicitly. This encapsulation of data distribution allows the Active Pebbles programming model to support a variety of approaches to data distribution. The simple hash-function case supports static addressing with very low storage overhead. Data sets which are amenable to partitioning may utilize an explicit distribution to represent the partition. This partitioning may be at the level of individual keys, or at a coarser block-level to minimize storage overhead. Finally, a dynamic data distribution may be supported via routing tables which provide for eventual location of targets

rather than full location resolution at the sender. Of course many more implementations are possible, possibly involving dimensions of the data distribution problem beyond those discussed here.

The second key element of Figure 4.1 are the message handlers, represented as pseudocode functions here. For many applications, much or all of the program logic can be encoded in these handlers. When bound to a physical location with a distribution object, the handler becomes a target which can be invoked via a pebble. No distinction is made between local and remote targets. Local targets are invoked immediately on the calling thread's stack while remote targets cause a pebble to be sent. The semantics for invoking remote handlers are not unlike a remote procedure call, although in the Active Pebbles case this call is asynchronous and does not return any information to the caller directly. Request-reply semantics may be implemented explicitly by having the the request handler invoked send a separate, asynchronous pebble in reply. The termination detection feature of Active Pebbles ensures that this reply will be received in the same epoch that the original pebble was sent in. The ability of message handlers to themselves send messages is the key to allowing the message structure of an application to encode it's computational dependencies. In many cases this functionality eliminates the need for any external message ordering or control flow. Dependencies expressed in this fashion are much finer-grained than loop-level dependencies often found in BSP or SPMD computations, which is important to hiding latency.

The Active Pebbles programming model was designed with hybrid parallelism in mind. Shared-memory parallelism may be supplied by users in the form of multiple threads making Active Pebbles calls concurrently (similar to *MPI_THREAD_MULTIPLE*). It may also be supplied internally by the runtime using threads or any of a variety of accelerators to handle incoming messages. To support these types of shared-memory parallelism, handlers are required to be reentrant and support many concurrent invocations efficiently.

4.1.1. Comparative Example. We compare and contrast related programming models with a simple example problem. Assume that each of P processes wants to insert n items into a hash table which is statically distributed across the P processes and uses chaining to resolve collisions. The keys for the hash table are uniformly distributed in $[0,N)$, $N \gg P$. We now compare possible implementations in different programming models.

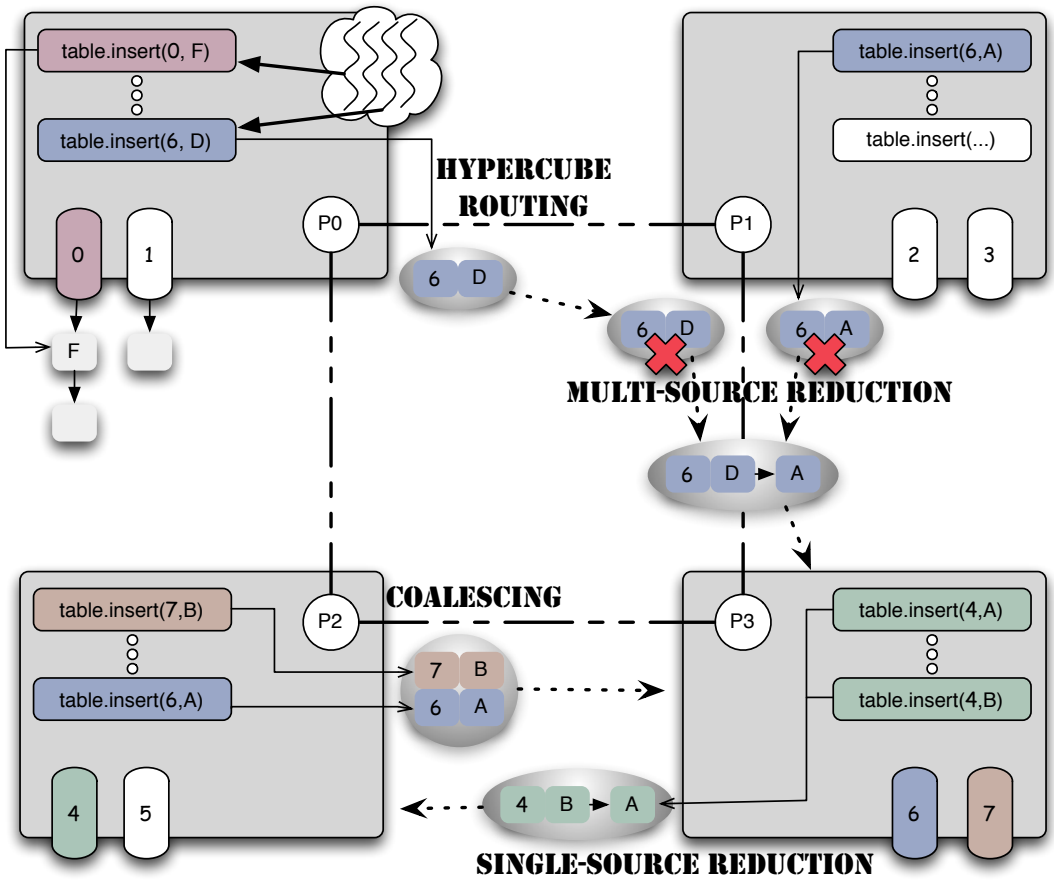


FIGURE 4.2. Active Pebbles execution model.

4.1.1.1. *MPI.* In one possible MPI implementation, each process would collect distinct sets of items, each destined for one remote process. After each process inserted all n requests, all processes would participate in a complete exchange. The uniform distribution guarantees that each process communicates with each remote process with high probability. This communication can either be done with direct sends or with a single

MPI_Alltoallv() operation (plus an *MPI_Alltoall()* stage to determine the pairwise message sizes). Each process would receive and add items to its local portion of the hash table. On average, each would receive and process $\Omega(n)$ items from $\Omega(P)$ peers, incurring a cost of $\Omega(n+P)$.

4.1.1.2. *PGAS.* A possible PGAS implementation would create the hash table in the global address space and each process would add items directly ensuring mutual exclusion by locking. This would need $\Omega(n)$ lock/unlock messages in addition to the $\Omega(n)$ data transfers per process. Resolving collisions is likely to require further messages and locks (e.g., to allocate additional space).

4.1.1.3. *Object-Oriented.* In object-oriented parallel languages, such as Charm++ [**97**] or X10 [**39**], the hash table would be a global object (e.g., a *Chare*). Each item would trigger a member function (e.g., insert) of the hash table object. For large n and P the vast number of remote invocations and their associated management overhead, as well as the small amount of computation per object, would impact performance significantly.

4.1.1.4. *Active Pebbles.* Figure 4.2 shows a schematic view of the Active Pebbles execution model. In an Active Pebbles implementation the user would specify a handler function which adds data items to the local hash table. The user would then send all data elements successively to the handler for each individual key (which is globally addressable). The Active Pebbles framework takes these pebbles and coalesces them into groups bound for the same remote process (two items with keys 6 and 7 sent from process P2 to P3 in Figure 4.2). It can also perform reductions on these coalesced groups of messages to eliminate duplicates and combine messages to the same target key (shown at P3 "single-source reduction" in Figure 4.2). Active routing sends all (coalesced) messages along a virtual topology and applies additional coalescing and reductions at intermediate hops (Figure 4.2 shows routing along a hypercube, i.e., P0 sends messages to P3 through P1 where they are coalesced and reduced, "multi-source reduction," with other messages).

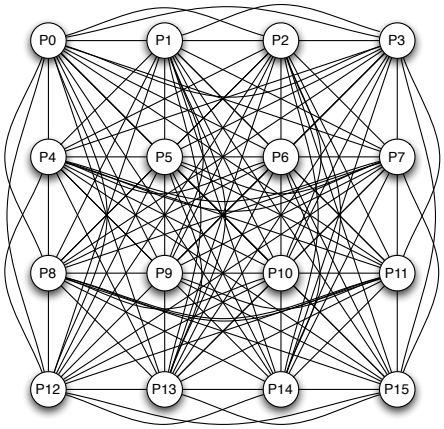
4.2. Active Pebbles Execution Model

Programs written in the Active Pebbles programming model contain the minimum ordering constraints necessary for correct program execution. Computational dependencies are expressed directly via message dependencies, which are discovered dynamically at runtime. This formulation produces maximal parallelism and asynchrony. However, it also results in a stream of tiny messages; the fine-grained execution of which may not be directly suited to certain hardware (e.g., due to message injection rate limits in distributed memory). The Active Pebbles execution model applies a number of runtime transformations to this message stream in order to adapt algorithms to the underlying hardware.

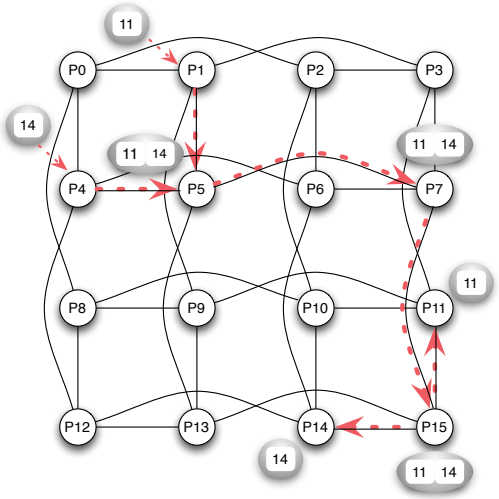
In this section we analyze the performance of each of the Active Pebbles mechanisms in detail. Pebble addressing (Section 4.2.1) is a feature of the programming model but its implementation is an important concern for the execution model, thus it is also discussed here. We utilize the well-known LogGP model [**8**] as a framework for formal analysis. The LogGP model incorporates four network parameters: L, the maximum latency between any two processes; o, the CPU injection overhead of a single message; g, the "gap" between two messages, i.e., the inverse of the injection rate; G, the "gap" per byte, i.e., the inverse bandwidth; and P, the number of processes. We make two modifications to the original model: small, individual messages (pebbles) are considered to be of one-byte size, and we assume that the coefficient of G for an n-byte message is n rather than $n-1$ as in the original LogGP model. LogGP parameters written with a subscript p refer to pebble-specific parameters imposed by our *synthetic network*. For example, L_p is the per-pebble latency, which might be higher than the network latency L due to active routing or coalescing; analogously, o_p, g_p, and G_p are the overhead, gap, and gap per byte for a pebble.

Active Pebbles allows a physical network described by one set of LogGP parameters to be mapped to a synthetic network with a different, more desirable set of LogGP parameters.

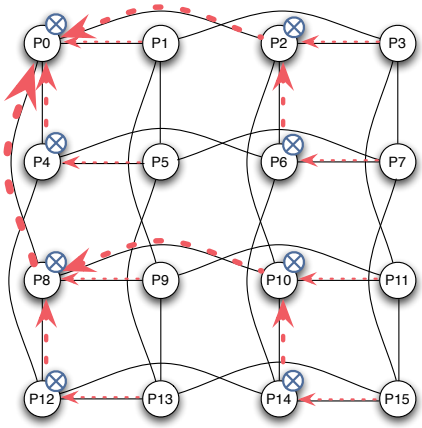
4.2.1. Fine-Grained Pebble Addressing. In the Active Pebbles model, messages are sent to individual targets, rather than process ranks. Target identifiers (which are typically domain-specific) are converted to ranks using a user-defined distribution object. Target



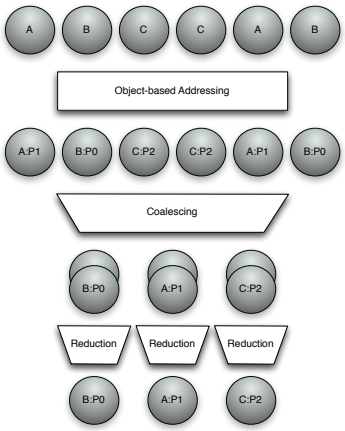
(A) Full system (as exposed by MPI).



(B) Pebble routing in a synthetic hypercube network.



(C) Pebble reduction in a synthetic hypercube network.



(D) Coalescing and reductions on idempotent objects bound for processes *P*0, *P*1, and *P*2.

FIGURE 4.3. Active Pebbles execution model features.

identifiers form a global address space, as in other GAS models. Active Pebbles supports both static and dynamic distributions. When a static distribution is used, the distribution will often require only constant space and constant time for location computations.

A dynamic distribution is likely to require larger amounts of storage; however, applications need not use dynamic distributions if static ones will suffice. Note that, unlike some other object-based messaging systems, Active Pebbles does not require any particular information to be kept to communicate with a target; i.e., there is no setup required to communicate, and a sending thread does not require any local information about the destination target (other than the distribution, which can be shared by many targets). A target identifier can be as simple as a global index into a distributed array; such identifiers can be created and destroyed at will, and are thus very lightweight. This mechanism is similar to *places* in X10 or *Chare arrays* in Charm++ but those mechanisms enable migration and other other advanced management and thus require $\mathcal{O}(n)$ time and space overhead to manage n elements; a statically-distributed array in Active Pebbles would only require $\mathcal{O}(1)$ space overhead to manage n elements, on the other hand. To simplify applications, the fine-grained addressing layer traps pebbles destined for the sending node and calls the corresponding handler directly, avoiding overheads from message coalescing, serialization, etc. In LogGP terms, addressing moves some time from the application to the model's o_p term without an overall change to performance.

4.2.2. Message Coalescing. A standard technique for increasing bandwidth utilization is message coalescing. Message coalescing combines multiple pebbles to the same destination into a single, larger message. This technique is well-known in the MPI arena and often performed manually. In LogGP terms, the packing of n 1-byte messages into one, n-byte message changes the overall message latency from $(n+1)o+L+G+(n-1)g$ to $2o+L+nG$. Thus, coalescing improves performance if $G < o+g$, which is true for all practical networks. However, coalescing works against pipelining. While without coalescing, the first pebble can be processed at the sender after $2o+L+G$, coalescing delays this to $2o+L+nG$. Thus, coalescing improves bandwidth utilization and reduces the use of g and o at cost of per-pebble latency L_p. Finding the optimal coalescing factor is system-dependent and subject to active research. Our LogGP discussion can be used as a good starting point and, in Active Pebbles, a user can specify the coalescing factor at run-time.

Also, it may not be known exactly when the stream of pebbles will end, and so a timeout or similar scheme can be used to know when to stop accumulating pebbles. Active Pebbles includes a *flush()* call to signal to the coalescing implementation that messages in all buffers for one or all destinations should be sent immediately.

One disadvantage of message coalescing is its use of memory. For fast access to buffers, one buffer must typically be kept for each possible message destination. Some implementations also use an individual, preallocated message buffer to receive messages from each possible sender. Those buffers can be established lazily, i.e., allocated at first use, however, reserving space for P buffers per process is impractical for large P if the communication is dense. The next feature, active routing, is an efficient technique to reduce the number of such buffers and increase performance.

4.2.3. Active Routing. Highly irregular applications often send messages from each process to almost every other process. Thus, the packet injection rate on each process's outgoing network links and the number of coalescing buffers become performance bottlenecks. To work around these limitations, our implementation of the Active Pebbles model can route messages through one or more intermediate hops on their ways to their destinations. For example, a hypercube topology can be embedded, with messages routed in such a way that they are only sent along the edges of the hypercube. Thus, each node only sends directly to a small number of other nodes, allowing fewer, but larger, coalesced messages to be sent. Users can also define their own routing schemes, including dynamic routing for fault tolerance.

Several researchers have found (e.g., Bruck et al [**29**]) that software-based routing improves performance on personalized (all-to-all) communications. For example, Bader, Helman, and JáJá show that adding one intermediate node into each message's path leads to a substantial performance improvement [**17**]; Plimpton et al show a performance benefit, both in theory and in practice, of simulating a hypercube topology for the HPCC RandomAccess benchmark [**134**]. Garg and Sabharwal discuss the benefits of manual software routing along a virtual torus on Blue Gene/L [**72**] and Yoo et al. [**175**] use manual routing

and message reductions to optimize breadth-first search at large-scale. The original paper that introduced the LogGP model shows a personalized broadcast (*MPI_Scatter()*) operation that uses tree-based routing [**8**], reducing the number of messages sent from $\mathcal{O}(P^2)$ to $\mathcal{O}(P \log P)$, although the total number of message bytes is increased by a factor of $\log P$. In LogGP terms, the assumption made is that $o + g$ is large enough compared to G that it is acceptable to use extra bandwidth to reduce the number of messages sent.

In many networks, routing increases the number of links used by any given pebble, leading to greater bandwidth utilization. However, routing allows pebbles from multiple sources and/or to multiple destinations to be packed together into a single, larger message, increasing the effectiveness of coalescing compared to a non-routed network. A similar argument applies to reductions across pebbles from multiple source nodes to the same destination target; see Section 4.2.4 for an analysis. Thus, active routing can reduce the number of packets sent across the network, potentially leading to less congestion and a performance improvement.

Active routing also reduces the amount of memory needed for message coalescing because it limits k, the number of other processes that each process communicates with. Without routing, $k = P - 1$, and thus each node requires a large number of communication buffers. Hypercube routing, with $k = \log_2 P$, reduces the number of buffers needed from $\Theta(P)$ to $\Theta(\log P)$.

Active routing, in combination with other Active Pebbles features, effectively converts fine-grained point-to-point messaging operations into coarse-grained optimized "collective" operations. For example, one source sending separate pebbles to different destinations will, by combining routing and message coalescing, actually use a tree-based scatter operation that takes advantage of large message support in the network (as in [**8**]). Similarly, a number of sources sending pebbles to the same target will combine them into a tree-based, optimized gather operation (similar to [**29**]).

Figure 4.3b shows routing along a hypercube (two messages are coalesced at $P5$ and split up at $P15$; routing reduces the message volume significantly). Messages flow only

along the edges of the hypercube compared with (logically) direct all-to-all routing in Figure 4.3a. Active Pebbles' synthetic topologies can be optimized for the topology of the physical communication network, as with MPI collective operations (e.g., [**29**]).

4.2.4. Message Reductions. In many applications, multiple pebbles of the same type to the same object are redundant: duplicate messages can be removed or combined in some way. We call this optimization message reduction; it can occur either at a message's sender or—with active routing—at an intermediate node. Reduction is implemented using a cache. For duplicate removal, previous messages are stored in a cache at each process; messages found in the cache are ignored. As analyzed below, lookup cost is important for the benefit of message reductions, requiring a fast, constant-time cache lookup at the expense of hit rate. In our experiments, we use a direct-mapped cache with runtime-configurable size; a miss replaces the contents of a cache slot with the new pebble.

For messages with data payloads, two cases are possible: the reduction operation is max (in some ordering; min is dual), in which case the cache simply removes messages with suboptimal values; or the reduction operation is something else that requires messages to be combined. In the latter case, message data payloads can be concatenated or combined (e.g., additively) as shown in Figure 4.2. In LogGP terms, reductions replace n messages by $n(1-h)$ messages (where h is the cache hit rate), but they also increase the value of o_p for each pebble to $o_p + c$, where c is the average cost of searching and maintaining the cache. The decrease in messages from n to $n(1-h)$ is equivalent to sending n messages but replacing G by $c + (1-h)G$. With message coalescing, G, c, and the message count are the only important factors in messaging performance; other factors are constant overheads. Without reductions, the time to send one pebble is G; reductions reduce the (expected) time to $c + (1-h)G$, leading to a benefit when $hG > c$.

This analysis only considers the effect of reductions on message latencies and bandwidth consumption; however, the reduction in computation at the target is likely to be even more important. Reductions reduce the expected processing cost p for each pebble to $p(1-h)$. Messages requiring expensive computation thus benefit from reductions by

reducing the number of those computations that occur. In particular, if a message can trigger a tree of other messages, one reduction at the source of that message can prevent many others from being sent at all.

Active routing can increase the benefit of reductions by allowing reductions across messages from multiple sources to the same target. With routing, a message is tested against the cache of every node along its path, increasing the chance of a match being found. For example, in a hypercube, each message is tested against up to $\log_2 P$ caches. Assuming the probabilities of hitting in each cache are independent, the overall hit rate becomes $1-(1-h)^{\log_2 P}$ (approximately $h \log_2 P$) rather than the rate h for a single cache. These multiple checks thus may lead to a greater reduction in message volume than would occur without routing.

When messages are sent from different sources to one target, routing and local message reductions at intermediate nodes combine to synthesize a reduction tree as would be used by an optimized implementation of *MPI_Reduce()*. This emergent property creates an efficient collective operation from point-to-point messages, with the routing algorithm defining the structure of the generated tree. Hypercube routing, for example, would generate binary reduction trees (Figure 4.3c shows an all-to-$P0$ reduction) with a logarithmic number of stages.

4.2.5. Termination Detection. The Active Pebbles model allows the handler for each pebble to trigger new communications. Thus, in a message-driven computation, it is nontrivial to detect the global termination (quiescence) of the algorithm. In the standard model for termination detection [**53**], each process can either be active or passive. Active processes perform computation and can send messages, as well as become passive. Passive processes can only be activated by incoming messages. A computation starts with one or more active processes and is terminated when all processes are passive and no messages are in flight. Many algorithms are available to detect termination [**121**], both for specialized networks and general, asynchronous message passing environments. Some

algorithms rely on special features of the network, such as strict channel ordering, a particular network topology, a global clock, or specialized hardware synchronization, while others work in general asynchronous message passing environments.

The optimal termination detection algorithm for an algorithm can depend on the features of the communication subsystem and on the structure of the communication (dense or sparse). Our framework enables easy implementation of different algorithms by providing several hooks into the messaging layer. In addition to initialization and termination detection itself, a termination detection algorithm can request to be triggered before and after each message send (e.g., for counting messages) and during progress calls. We implement one fully general termination detection scheme (SKR [**150**]), using a nonblocking *allreduce()* operation [**85**], similar to the four-counter algorithm in [**121**].

4.2.5.1. *Depth-Limited Termination Detection.* Some applications can provide an upper bound for the longest chain of messages that is triggered from handlers. For example, a simple application where each message only accumulates or deposits data into its target's memory does not require a generic termination detection scheme. Another example would be an application that performs atomic updates on target locations that return results (e.g., read-modify-write). These examples would require termination detection of depths one and two, respectively. Graph traversals can generate chains of handler-triggered messages of unbounded depth (up to the diameter of the graph), however. Several message-counting algorithms meet the lower bound discussed in [**86**] and detect termination in $\log P$ steps for depth one, while unlimited-depth termination detection algorithms usually need multiple iterations to converge. In the SKR algorithm, termination detection takes at least $2 \log P$ steps (two *allreduce()* operations). Our framework offers hooks to specify the desired termination detection depth to exploit this application-specific knowledge. We implement two depth-one termination detection schemes: a message-counting algorithm based on nonblocking *reduce_scatter()* [**85**], and an algorithm that uses nonblocking barrier semantics and is able to leverage high-speed synchronization primitives [**86**]. Both algorithms are invoked n times to handle depth-n termination detection.

4.2.5.2. *Termination Detection and Active Routing.* In active routing, each message travels over multiple hops which increases the depth of termination detection. For example in hypercube routing, an additional $\log_2 P$ hops are added to the termination detection. This is not an issue for detectors that can handle unlimited depths, but it affects limited-depth detection. With s-stage active routing and a depth-n termination requested by the application, limited-depth termination detection would take $n \cdot s$ steps. However, termination detection could take advantage of the smaller set of possible neighbors from active routing, such as the $\log_2 P$ neighbors of each node in a hypercube. This would reduce the per-round time to $\mathcal{O}(\log \log P)$, for a total time of $\mathcal{O}(\log P \log \log P)$ as compared to $\mathcal{O}(\log P)$ without routing. Routing may benefit the rest of the application enough to justify that increase in termination detection time, however.

4.2.6. Synthetic Network Tradeoffs. The Active Pebbles model uses coalescing, reductions, active routing, and termination detection to present an easy-to-use programming model to the user while still being able to exploit the capabilities of large-scale computing systems. Active Pebbles transforms fine-grained object access and the resulting all-to-all messaging into coarse-grained messages in a synthetic, or overlay, network. The synthetic network transparently transforms message streams into optimized communication schedules similar to those used by MPI collective operations. Various aspects of the Active Pebbles execution model can be adjusted to match the synthetic network to a particular application, such as the coalescing factor, the synthetic topology, and termination detection. The programming interface remains identical for all of those options. Active Pebbles can thus be optimized for specific machines without changes to application source code, allowing performance-portability.

4.3. Application Examples

We examine four example applications using the Active Pebbles model in order to explore the expressiveness and performance of applications written using it. Implementations of these applications in three models are evaluated: Active Pebbles; MPI; and UPC,

which we use to illustrate the programming techniques used in PGAS languages.[2] We first present a simple summary of these applications with more detailed explanations to follow:

RandomAccess: randomly updates a global table in the style of the HPCC RandomAccess benchmark [**118**]. We use optimized reference implementations where appropriate, as well as simplified implementations.

PointerChase: creates a random ring of processes and sends messages around the ring.

ParallelIO: permutes a data array distributed across P processes according to another distributed array, as might be used to redistribute unordered data after loading it.

BFS: is a graph kernel that explores a random Erdős-Rényi [**62**] graph breadth-first.

4.3.1. RandomAccess. The parallel RandomAccess benchmark measures the performance of updating random locations in a globally distributed table [**118**]. The benchmark resembles access patterns from distributed databases and distributed hash tables. It uses a global *table* of N elements distributed across P processes. The timed kernel consists of $4N$ updates to the table of the form *table[ran % N] ^= ran* where *ran* is the output of a random number generator. Processes may not buffer more than 1024 updates locally.

4.3.1.1. *PGAS.* In UPC the *table* can be allocated in the shared space and accessed just as in the sequential version of the algorithm:

UPC

```
1 uint64_t ran;
2 shared uint64_t* table = upc_all_alloc(N ,sizeof(uint64_t));
3 for (int i = 0 ; i < 1024 ; ++i) {
4   ran = (ran << 1) ^ (((int64_t)ran < 0) ? 7 : 0); // compute index
5   table[ran % N] ^= ran; // perform update
6 }
```

The UPC compiler/runtime then performs the necessary communication to perform the update to *table*.

[2] "PGAS" here refers to fine-grained remote memory accesses (the basic PGAS model), without active message extensions.

4.3.1.2. *MPI.* MPI has no notion of shared data structures, so the updates to non-local portions of the table must be explicitly communicated to the remote process which then applies them. Rather than sending individual updates we buffer 1024 updates sorted by destination then communicate them collectively. The MPI implementation of the RandomAccess application first buffers local updates, then communicates the number of updates followed by the updates themselves:

MPI

```
for (int i = 0 ; i < 1024 ; ++i) {
  ran = (ran << 1) ^ (((int64_t)ran < 0) ? 7 : 0); // compute index
  long index = ran % N;
  int owner = index / (N/P);

  // perform local update
  if (rank == owner)
    table[index % (N/P)] ^= ran;
  else // remote
    out_bufs[owner].buf[out_bufs[owner].count++] = ran;
}
// ...  allocate and prepare all-to-all communication buffers
MPI_Alltoall(out_bufs.count,...,in_bufs.count,...);
// ...  allocate and prepare all-to-allv communication buffers
MPI_Alltoallv(out_bufs.buf,out_bufs.count,...,
              in_bufs.buf,in_bufs.count,...);
```

4.3.1.3. *Active Pebbles.* In Active Pebbles we invoke remote handlers using pebbles. To implement RandomAccess we first encapsulate the update operation inside a handler:

Active Pebbles

```
struct update_handler {
  bool operator()(uint64_t ran) const
  { table[ran % (N/P)] ^= ran; } // update to table
};
```

This handler is then invoked from a remote process by creating a pebble type and assigning the handler to it. The pebble type encapsulates pebble addressing (through the *block_owner_map* type) and routing (through the *hypercube_routing* object passed to the type's constructor). A separate operation attaches a particular handler object to the pebble type:

Active Pebbles

```
1 pebble_addressing_dest_hbr<...>
2 update_msg(transport, ..., block_owner_map(N/P),
3          hypercube_routing(rank, size));
4 update_msg.set_handler(update_handler(table));
```

Active Pebbles detects messages which would be sent to the current rank using pebble addressing and simply calls the appropriate handler directly. This eliminates the need for applications to treat local and remote data differently:

Active Pebbles

```
1 for (int i = 0 ; i < 1024 ; ++i) {
2   ran = (ran << 1) ^ (((int64_t)ran < 0) ? 7 : 0);
3   update_msg.send(ran);
4 }
```

4.3.2. PointerChase. The PointerChase application creates a random permutation of $[0,P)$; each processor i then relays a single, small message to element $(i+1) \bmod P$ of the permutation. This benchmark is intended to model the performance of chains of dependent operations in an irregular application. It primarily tests message latency, and thus is expected to favor PGAS models.

4.3.2.1. *PGAS.* In UPC, notifying the next process to relay the message can be implemented by polling on a counter allocated in the shared space and waiting for its value to be updated:

UPC

```
1 shared int* flags = upc_all_alloc(THREADS, sizeof(int));
2 for (int i = 0; i < rounds; ++i) {
3   while (flags[MYTHREAD] != i) {}
```

```
4   flags[next_rank] = i;
5 }
```

4.3.2.2. *MPI.* The implementation of the PointerChase application is similar in MPI, except that messages replace the memory operations and *MPI_Wait()* is used instead of polling (example simplified):

MPI

```
1 for (int i = 0; i < rounds; ++i) {
2   MPI_Recv(&data, 1, MPI_ANY_SOURCE, ...);
3   MPI_Send(&next_rank, 1, next_rank, ...);
4 }
```

4.3.2.3. *Active Pebbles.* In Active Pebbles there is no main loop at all; the control flow is entirely represented in the message handler:

Active Pebbles

```
1 struct msg_handler {
2   bool operator()(int source, const int* data, int count) {
3     if (rank != start) msg→send(round, next);
4     else if (--round > 0) msg→send(round, next);
5 } };
```

The message handler is initialized with the rank which sends the first message (*start*), the number of loops around the ring to perform (*round*), and the next rank in the ring (*next*). After the handlers are initialized all that is required to start the application is for the *start* rank to send the first message:

Active Pebbles

```
1 typed_message<...>::type
2   msg(typed_message<...>
3         ::make(transport, 1, msg_handler(0, rounds, next, msg)));
4 if (rank == start) msg.send(0, next);
```

4.3.3. ParallelIO. The ParallelIO application is designed to be representative of a common task in scientific computing: distributing and permuting unsorted data (e.g., after it has been read from a file). The data distribution may exist to optimize locality, provide load-balancing, or for domain-specific reasons. ParallelIO uses three arrays representing input data (*data*), a permutation (*perm*), and the re-ordered data ($data_{perm}$). These arrays each contain N elements, and each is distributed across P processors. The *data* array contains an index into the *perm* array which functions as a unique identifier for the data element, as well as some associated data. The *perm* array contains the destination for each input element; the inverse of the permutation *perm* is applied. The result satisfies $\{\forall i \in [0,N) : data_{perm}[perm[data[i]]] = i\}$.

4.3.3.1. *PGAS.* In UPC we allocate the arrays in the shared space and use *upc_forall()* to distribute the work:

```
1 upc_forall (uint64_t i = 0; i < N; ++i; &data[i])                    UPC
2   data_perm[perm[data[i]]] = i;
```

In the preceding example we simply store the indices i into $data_{perm}$; a real application using this technique would assign the application data associated with index i.

4.3.3.2. *MPI.* In the MPI implementation of ParallelIO a similar communication pattern to that described in Section 4.3.1 is used. Rather than a single *Alltoall()*/*Alltoallv()* round, two rounds are required for ParallelIO. In the first round, elements of *data* are sent to the process that stores $perm[data[i]]$. In the second round *perm* is applied and the data sent to the process which owns $data_{perm}[perm[data[i]]]$. We have omitted the lengthy code for the MPI implementation of ParallelIO in the interest of brevity.

4.3.3.3. *Active Pebbles.* The Active Pebbles implementation combines the movement of *data* and the application of *perm* into a single phase using dependent messages and depth-two termination detection to detect completion. The MPI implementation must address the situations where some dependent elements of *data*, *perm*, and $data_{perm}$ may be local and others remote. Active Pebbles handles these locality concerns automatically because

pebbles sent to local targets will simply call the correct underlying handler with no performance penalty. The Active Pebbles implementation uses two handlers, the first receives elements of *data*, applies *perm*, and sends a pebble to the owner of the appropriate target in $data_{perm}$:

Active Pebbles

```
1 struct permute_handler {
2   bool operator()(const pair<uint64_t, uint64_t>& x) const {
3     uint64_t target_idx = perm[get(local_map, x.first)];
4     put→send(make_pair(target_idx, x.second));
5 } };
```

The second handler is the *put* handler used by the *permute_handler*. This handler writes values to the specified target in $data_{perm}$:

Active Pebbles

```
1 struct put_handler {
2   bool operator()(const put_data& x) const
3   { data_perm[get(local_map, x.first)] = x.second; }
4 };
```

After the message handlers are initialized, each process simply sends pebbles for all of its local data:

Active Pebbles

```
1 make_coalesced_mt<...>::type data_permute(transport, ...);
2 for (int i = 0; i < N/P; ++i)
3   data_permute.send(make_pair(data[i], N/P * rank + i));
```

Once termination detection completes, all pebbles sent (both originally and from handlers) will have been processed, and so all elements of $data_{perm}$ will have been updated.

4.3.4. BFS. The final application we consider is breadth-first search on a directed graph. Graph algorithms are an excellent application of the Active Pebbles model because they often create many fine-grained asynchronous tasks. All of the implementations use an Erdős-Rényi random graph with a one-dimensional vertex distribution across the P processors.

In a bulk-synchronous implementation BFS is implemented with a single logical distributed queue, composed of logical queues on each process. A *push()* operation on any process results in a vertex being placed on the local queue of the vertex's owning process. Neither MPI nor UPC support dynamic shared data structures such as queues directly. Buffering queue operations at the source and applying them collectively addresses this limitation in both cases. In the case of MPI, this is necessary because *MPI_Accumulate()* is not expressive enough to implement queue update operations; e.g., it cannot atomically fetch and increment a counter. In UPC, vertices could be pushed directly onto the targets' local queues, but the queues would then need to be locked remotely, limiting concurrency.

4.3.4.1. *PGAS + MPI.* The MPI and UPC implementations use similar algorithms:

MPI/UPC Pseudocode

```
1 if (source is local) Q.push(source);
2 while (!Q.empty()) {
3   for (v : Q)
4     if (visited[v] == 0) {
5       visited[v] = 1;
6       for (w : neighbors[v]) {Q2.push(w);}
7     }
8   Q.clear(); swap(Q, Q2);
9 }
```

4.3.4.2. *Active Pebbles.* In Active Pebbles we can choose a formulation of the BFS algorithm that expresses a better mapping to the programming model. Level-wise traversals of the BFS tree are possible using a queue to buffer pebbles, as are versions which compute a BFS numbering using a single-source shortest path algorithm. The latter approach is asymptotically more expensive, but significantly reduces the synchronization required and thus may be desirable in practice. An implementation in Active Pebbles would define a handler which utilized a *distance* property for each vertex:

Active Pebbles

```
1 struct bfs_handler {
2   // x is a ⟨vertex, distance⟩ pair
```

```
3   bool operator()(const pair<Vertex, int>& x) const {
4     if (x.second < distance[x.first]) {
5       distance[x.first] = x.second;
6       explore→send(x.first, x.second + 1);
7  } } }; // explore is an instance of a message type
```

In this formulation there is no need for any queues: all the message buffering and work coalescing performed by the queue in the other implementations is performed by Active Pebbles. Running the algorithm simply requires exploring the source vertex by sending a message to a *bfs_handler()*. The handler at the source vertex will then call the *bfs_handler()* for adjacent vertices, which will in turn make their own recursive calls until all connected vertices are explored.

4.4. Retroactive Optimization

Section 4.2 described a number of transformations that may be applied by the Active Pebbles execution model to increase the performance of applications expressed in the Active Pebbles programming model. The focus of that discussion was on demonstrating that a particular set of transformations could be used to map applications expressed using fine-grained active messages to efficient implementations on modern hardware. This decoupling of expression from implementation has a number of powerful corollaries.

Defining classes of transformations with well-defined semantics allows applications to specify a valid set of possible transformations without knowing about the implementations of those transformations. For example, an application may specify that idempotent messages may be eliminated; with idempotency defined by a user-supplied binary function. The Active Pebbles execution model is "best effort" in the sense that no guarantees are made with regard to every idempotent message being eliminated, or indeed any of them. Many possible implementations are possible with regard to identifying idempotent messages. A naïve implementation may simply maintain a buffer of messages and perform a linear search. More sophisticated implementations may use hashing to implement

some sort of cache. This cache may be direct-mapped or utilize various forms of chaining. It may be write-through or write-back. Selecting from these and other decisions yields a multitude of possible implementations.

The Active Pebbles execution model defines a framework which allows applications to share and reuse implementations of common transformations. It also allows applications to benefit from transformations that may have been written *after* the applications themselves. This retroactive optimization allows existing applications to be adapted to future advances in algorithms and hardware without modifying the application's specification. Chapter 6 discusses a software abstraction that allows library developers to parameterize their algorithms over the Active Pebbles features utilized by their message types. This enables end-users to select the set of transformations most appropriate to their data set and execution environment and pass the resulting description to the library. By allowing retroactive optimization and end-user customization of applications the application specification can far out-live the underlying hardware and the transformations which target it.

The Active Pebbles programming model captures the maximum information about the computational and communication structure of an application. This minimally-constrained implementation is well-suited to separate optimization because there is no need to disentangle "real" dependencies from "artificial" ones which arise as a consequence of the programming model.

4.4.1. Shared-Memory Optimization Opportunities. The classes of transformations described in Section 4.2 are primarily targeted at making communication more efficient. However, the Active Pebbles execution model observes messages addressed to local targets in addition to those bound for remote targets. This presents opportunities to parallelize or otherwise optimize the local portion of a computation as expressed in messages bound for local targets.

The current execution model for messages addressed to local targets is recursive execution within the Active Pebbles runtime until the message handler returns as shown in

Figure 4.4. Message handlers may return because no further messages are sent, or because the only messages sent are addressed to non-local targets. This depth-first recursive execution is inherently sequential. However, because Active Pebbles places no restrictions on message ordering the runtime is free to reorder these computations. This transformation may take the form of a thread pool to handle multiple, concurrent messages or some sort of multi-threaded work stealing [**25**] approach. Thus, the runtime may be extended in the future to parallelize local work in a fashion that is transparent to the application.

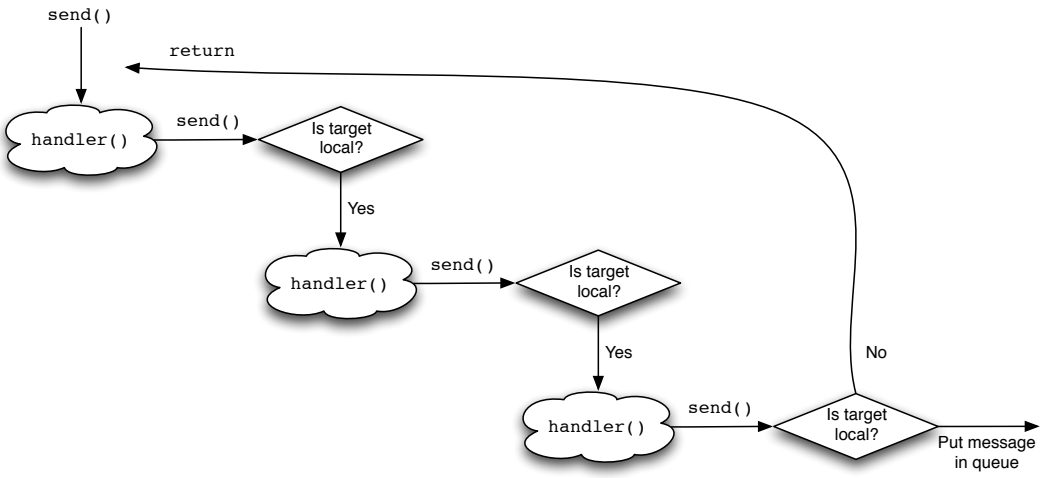


FIGURE 4.4. Flow chart showing result of message sends to local targets.

4.5. Conclusion

The Active Pebbles programming model allows graph algorithms to be expressed at their natural levels of granularity (e.g., individual vertex and edge operations) without imposing artificial computational dependencies. This expression captures essential algorithm logic in a fine-grained fashion that can be implemented using the Active Pebbles execution model. The Active Pebbles execution model provides a framework for runtime optimization which is independent of algorithm specification. As computational dependencies are discovered at runtime, transformations to optimize both the communication

structure of the program and the local, shared-memory execution can be applied. Separating algorithm specification from implementation in this fashion allows common optimizations to be reused, enables performance portability, and simplifies the task of algorithm development by allowing algorithm definition and performance tuning to be performed separately. The Active Pebbles approach of deferring optimization decisions until runtime is especially appropriate for graph algorithms as the structure of the graph determines the computational structure of the algorithm and thus both are discovered dynamically. Active Pebbles yields the expressiveness and programmability needed by a user-level graph library while providing the performance of hand-tuned code.

5

Classification of Parallel Graph Algorithms

Identifying commonly used kernels or patterns has long been recognized as a useful method for enabling portability and high performance. A single common interface gives end users a target to program to and system vendors a target to optimize for. Furthermore a single widely used implementation is more likely to be correct and efficient.

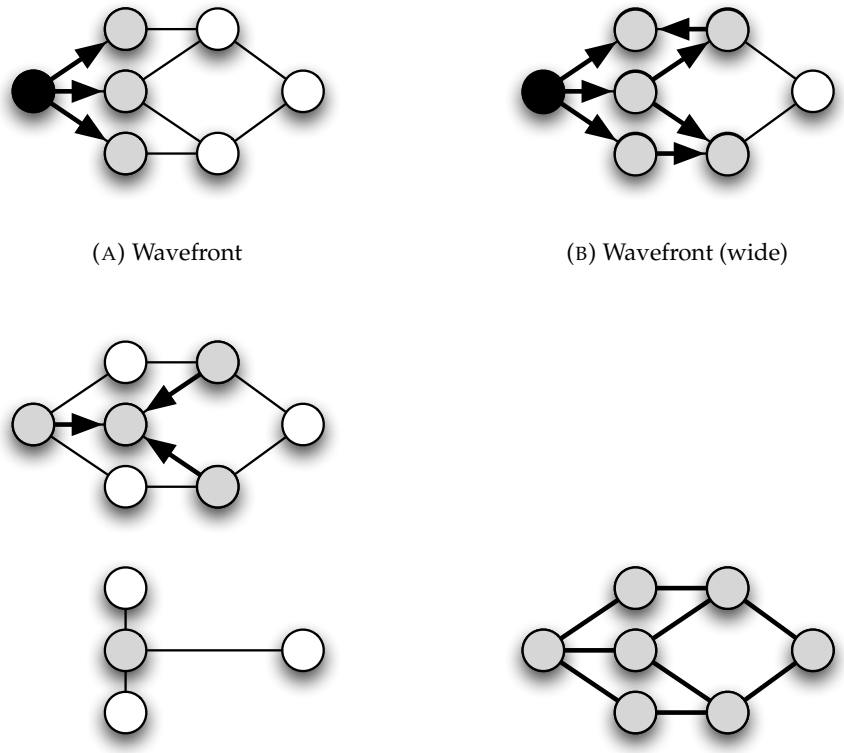
The Basic Linear Algebra Subprograms (BLAS) [**110**] are perhaps the canonical example of such a library. The BLAS were originally defined in order to increase the modularity of scientific software. LINPACK's [**12**] use of the BLAS encouraged experts and vendors to optimize its vector operations. In addition to efficiency, the availability of shared interfaces improved portability. Deeper memory hierarchies and advances in microprocessor

design resulted in memory access becoming much more expensive than floating-point operations. BLAS Level 2 [**55**] and Level 3 [**54**] were developed in the late 1980's to provide higher level blocked primitives which allowed implementations to achieve a higher ratio of floating-point operations to memory accesses. Other higher level patterns of parallelism have also been observed in compute-driven HPC applications [**101**, **122**].

More recently, the Berkeley Report [**13**] has broadened the discussion of common algorithmic kernels with a top-down survey of a broad variety of applications. This report identifies 13 kernels which they call "dwarves" (or "motifs"). This thesis is most concerned with the graph traversal dwarf. Sparse linear algebra is another important dwarf identified by the Berkeley report. Linear algebraic graph primitives have been proposed in [**30**] and Pingali et al. [**133**] have observed higher-level classifications in what they term "amorphous data-parallelism". We have observed recurring kernels in visitor- or traversal-based graph algorithms as well.

In this chapter we distinguish graph traversal from data-mining algorithms. Data-mining algorithms such as subgraph isomorphism [**42**, **162**] are highly data-parallel and can be made embarrassingly parallel for small pattern graphs with appropriate ghost cell strategies along partition boundaries, provided the graph does not have a large expansion factor. In the latter case, the algorithms be more effectively supported using graph databases [**111**, **145**]. We argue here that traversal-based algorithms can be better understood as fine-grained, task-parallel operations. In the case of graph algorithms the granularity of the task is so small that we feel the subcategory "data-driven" is more appropriate.

The patterns shown in Figure 5.1 and described below are representative of a significant fraction of existing graph algorithms (though by no means all of them). These patterns are distinguished not by the raw underlying computations, but rather in the way they expose parallelizable work over time and the dependency structure of this work. We now examine how these patterns are implemented using Active Pebbles and how they might be implemented in other programming models.



(A) Wavefront

(B) Wavefront (wide)

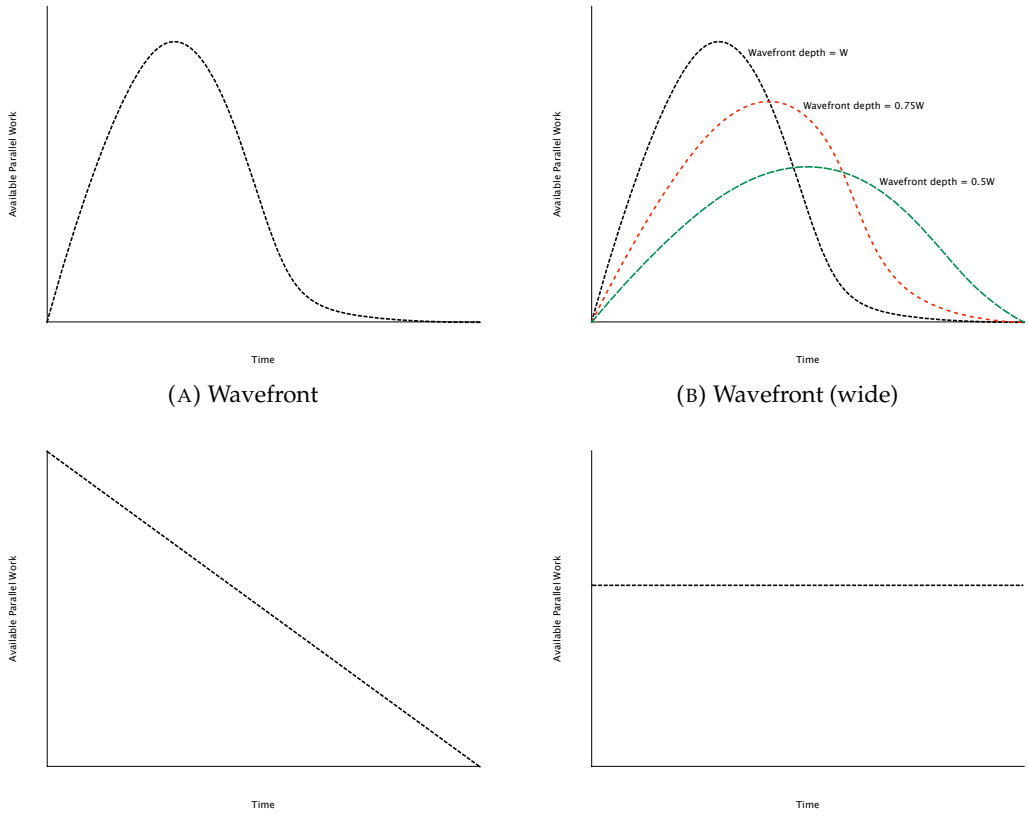
(C) Coarsening and Refinement

(D) Iterative

FIGURE 5.1. Graph algorithm patterns. Grey vertices represent active vertices in the current epoch while black vertices (where present) indicate active vertices in the previous epoch.

5.1. Elements of Graph Algorithm Performance

The performance of a graph application is dependent on two primary factors: the structure of the input graph and the structure of the computation being performed on that input graph. In this chapter we focus on classifying the computational structure of graph algorithms in order to provide some intuition as to how the performance results presented in Chapter 7 would generalize to other algorithms from the same classes. Figure 5.3 contains a set of example algorithms mapping to each of the patterns in Figure 5.1. This list is by no means exhaustive.



(A) Wavefront

(B) Wavefront (wide)

(C) Coarsening and Refinement

(D) Iterative

FIGURE 5.2. Illustration of how parallel work is exposed over time in each of the patterns in Figure 5.1. These charts are intended to illustrate general trends in each class of algorithm and do not correspond to individual empirical data sets.

Much work has also been done to characterize and model the structure of random graphs in a variety of domains [**21**, **33**, **113**, **167**]. Given graph data from a new problem domain, determining which of the existing models, if any, the data matches most closely is a challenging and independent problem. The focus in this work is on providing a general purpose solution for parallel graph processing using traversal-based algorithms. To this end, the algorithms presented by this thesis in the context of the Parallel BGL 2.0 attempt to be as agnostic as possible to the structure of the input graph. Because all layers of the graph application stack utilized, from AM++ to Active Pebbles and the Parallel BGL 2.0, utilize

Breadth first search	Δ-stepping
Strongly connected components	Parallel search
Cuthill-Mckee ordering	Depth-limited search

(A) Wavefront

(B) Wavefront (wide)

Shiloach-Vishkin connected components	PageRank
Dehne-Götz minimum spanning tree	Fruchterman-Reingold layout
Bůruvka minimum spanning tree	Boman et al. graph coloring

(C) Coarsening and Refinement

(D) Iterative

FIGURE 5.3. List of example algorithms for each of the patterns in Figure 5.1. This list is not exhaustive.

generic programming, it should be possible to specialize any of these implementation to take advantage of knowledge about graph structure.

5.2. Wavefront Expansion (Label-Setting)

Perhaps the simplest traversal-based pattern is the exploration of a graph from a single source vertex. As depth-first searches are inherently sequential [**138**], the most basic *parallelizable* search is BFS. The wavefront in this case represents the active vertices at each level in the search. This traversal pattern is *label-setting* in the sense that once a vertex is visited and a label (distance, predecessor, etc.) assigned it will not be revisited or have this label changed.

In order to achieve breadth-first exploration we have to assure that all vertices at distance i from the source vertex are processed before any vertices at distance $i+1$. Messages in Active Pebbles are sent immediately and there is no ordering imposed between messages within the same epoch so this behavior requires one epoch per queue level, where level i in the queue consists of all vertices which have distance i from the source.

Algorithm 1 shows a breadth-first search implementation which uses two queues to force level-wise exploration. We have hidden the fact that in the distributed memory case each logical queue is actually composed of one local queue per process containing vertices

Alg. 1: Basic algorithm for breadth-first search.

```
   In  : Graph 𝒢 = ⟨V, E⟩, vertex s
   Out: ∀v ∈ V: distance[v] = distance of v from s
 1 Q ← empty queue;
 2 nextQ ← empty queue;
 3 level ← 0;
 4 distance[v] ← ∞, ∀v ∈ V;
 5 distance[s] ← 0;
 6 enqueue s → Q;
 7 while Q not empty do
 8     level ← level + 1;
 9     foreach v ∈ Q do
10         foreach neighbor w of v do
11             if distance[w] ≠ ∞ then
12                 distance[w] ← level;
13                 enqueue w → nextQ;
14     Q ← nextQ;
15     nextQ ← empty queue;
```

from the set assigned to that process. Line 6 tests whether all local queues are non-empty, while line 8 iterates over the local queue on each process.

The available parallel work in a label-setting wavefront is equivalent to the size of the queue; at each level every vertex in the distributed queue may be processed in parallel. Figure 5.2a illustrates how the parallel work in algorithms of this form is exposed over time. Initially the queue contains a single source vertex. As the wavefront representing the portion of the graph that has been explored expands, the number of vertices in each successive queue level increases. Finally, as the number of unexplored vertices diminishes the size of the queue at each level decreases until all reachable vertices have been explored.

To implement this algorithm using active messages (Algorithm 2), line 11 of Algorithm 1 becomes an active message that inserts w into the local queue at its target, removing the need for communication inside the distributed queue data structure itself. The messaging layer ensures that each *enqueue()* operation executes on the owning process of the vertex being enqueued. If we make the *enqueue()* method of Q thread-safe and add a

Alg. 2: Parallel active message algorithm for BFS.

```
   In  : Graph G = ⟨V, E⟩, vertex s,
                 ∀v ∈ V: owner[v] = rank that owns v
   Out: ∀v ∈ V: distance[v] = distance of v from s
 1 Q ← nextQ ← empty queue;
 2 level ← 0;
 3 distance[v] ← ∞, ∀v ∈ V;
 4 distance[s] ← 0;
 5 enqueue s → Q;
 6 message handler explore(Vertex v)
 7     if distance[v] ≠ ∞ then
 8         distance[v] ← level;
 9         enqueue v → nextQ;
10 while Q not empty do
11     level ← level + 1;
12     active message epoch
13         parallel foreach v ∈ Q do
14             parallel foreach neighbor w of v do
15                 send explore(w) to owner(w);
16     Q ← nextQ;
17     nextQ ← empty queue;
```

critical section around lines 12–13 of Algorithm 1, this implementation is also thread-safe and the **foreach** statements on lines 8 and 9 can be executed in parallel.

An implementation using MPI collectives would also use one epoch per iteration of the loop on line 6 of Algorithm 1, but rather than communicating immediately on lines 10–11 it would buffer all remote *enqueue()* operations locally and send them at the conclusion of the loop using an *MPI_Alltoall()* operation (to communicate the number of operations) followed by an *MPI_Alltoallv()* operation to send the data. This approach fails to effectively overlap communication and computation in addition to being more complex than the active message formulation. Non-blocking collectives [**85**] and asynchronous point-to-point methods fail to alleviate this problem since we cannot begin the next queue level until the previous one is complete.

Implementing a distributed queue of the sort described above using MPI-2 One-Sided communications or the Remote Direct Memory Access (RDMA) semantics provided by a

number of PGAS languages is complicated by the absence of high-performance atomic operations such as compare and swap or fetch and add. In the absence of these operations, each process must allocate separate input buffers for each other remote process to write into. These buffers must be large enough to contain all remote updates and if this cannot be guaranteed, logic must be provided to ensure all remote updates are received, possibly by using multiple epochs per queue level (with its attendant extra synchronization). At the conclusion of the BFS level, each process must apply the updates from all other processes and clear the input buffers. This approach either requires large amounts of additional memory to be allocated—to ensure that updates fit in the input buffers—or requires additional synchronization to perform multiple rounds of remote update exchanges per queue level. Even in the presence of a high-performance implementation of the RDMA compare and swap operation added to MPI-3 [**126**], multiple network round trip times would be required to increment the head pointer of the queue and insert a new element. This operation can be performed using active messages with much lower latency and less communication bandwidth.

5.3. Wavefront Expansion (Label-Correcting)

While breadth-first search is a simple and concise graph algorithm to use as an example, the strengths of the active message abstraction are more apparent in algorithms with more complex dependency structures. The label-correcting wavefront expansion pattern differs from the label-setting one in one important way. Instead of exploring vertices and edges once and labeling them, algorithms which use a label-correcting wavefront may recompute and correct vertex and edge labels multiple times. Although this approach may be less work-efficient than a label-setting algorithm, it often exposes much more parallelism by allowing updates that are "usually independent" to proceed in parallel with exceptions repaired later.

Some label-correcting algorithms also use "thick" wavefronts: instead of only a single layer (i.e., frontier) of vertices being active at a time, vertices within a given "distance" (for an algorithm-specific definition of distance) are active. The thickness of the wavefront is

controlled in an algorithm-specific manner, often using a parameter Δ to define the thickness of the wavefront. Thicker wavefronts potentially expose more work at a given time and are thus more parallelizable, while thinner wavefronts are likely to be more work efficient. Figure 5.1b illustrates how a single level of the wavefront computation may contain paths of differing lengths in the graph, and thus computational dependencies within the wavefront. The canonical examples of label-correcting wavefront expansion are parallel SSSP algorithms [**43**, **124**].

Alg. 3: Δ-stepping shortest paths algorithms [**124**]

Input: Weighted graph $\mathscr{G} = (V, E)$, vertex s, $c(v,w)$ = weight of edge $v \rightarrow w$
Output: $\forall v \in V$: $distance[v]$ = length of shortest path to v from s

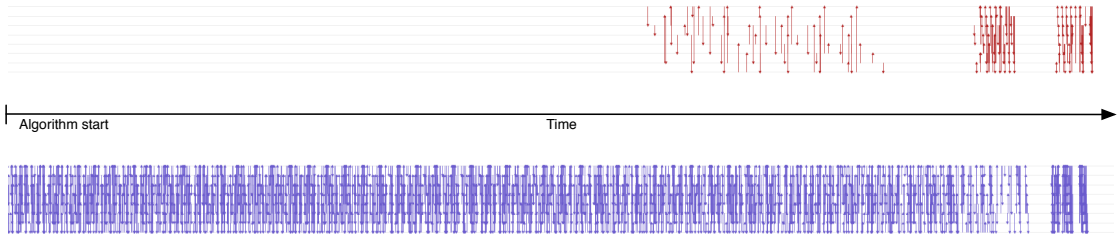
```
1  i = 0;
2  distance[v] = ∞, ∀v ∈ V;
3  Buckets[j] = ∅, ∀j;
4  Buckets[0] = s;
5  distance[s] = 0;
6  while Buckets not empty do
7      D = ∅;
8      while Buckets[i] ≠ ∅ do
9          R = {(v,w) | ∀v ∈ Buckets[i] ∧ c(v,w) ≤ Δ};
10         D = D ∪ Buckets[i];
11         Buckets[i] = ∅;
12         foreach (v,w) ∈ R do
13             relax(v,w);
14     R = {(v,w) | ∀v ∈ D ∧ c(v,w) > Δ};
15     foreach (v,w) ∈ R do
16         relax(v,w);
17     i = i + 1;
18 Procedure relax(v,w)
       Input: Vertices v,w ∈ V
       Output: Updated Buckets and distance[w]
19     dist = distance[v] + c(v,w);
20     if dist < distance[w] then
21         if distance[w] < ∞ then
22             Buckets[⌊distance[w]/Δ⌋] = Buckets[⌊distance[w]/Δ⌋] \ {w};
23         Buckets[⌊dist/Δ⌋] = Buckets[⌊dist/Δ⌋] ∪ {w};
24         distance[w] = dist;
```

Algorithm 3 provides a high-level overview of the Δ-stepping SSSP algorithm. The main idea is to utilize an approximately-sorted data structure to group work into "buckets" such that all vertices in a bucket may be processed in parallel. Rather than each level of the wavefront containing a set of independent vertices with a common label as in the label-setting pattern, work is now grouped into sets of vertices with "tentative" labels in the range $[i\Delta,(i+1)\Delta)$ where i is the current level. Additionally, these vertices may now have dependencies between them (caused by "light" edges with weight less than Δ, using the terminology from [**124**]). The fine-grained structure of these intra-level dependencies provides additional opportunities for parallelism, provided they can be captured efficiently.

As with label-setting wavefronts, the available parallel work at any point in time in a label-correcting wavefront algorithm depends on the size of the distributed queue used to represent candidate paths. Figure 5.2b demonstrates how the amount of work available as a function of time and the total number of epochs required by an algorithm may be varied by manipulating Δ. This parameter also affects the work efficiency of label-correcting algorithms.

Using active messages, dependencies between vertices u, v, and w of the form $u \to v \to w$ can be handled directly by having vertex u send a message to vertex v which then performs the appropriate computation and sends a message to vertex w. Using MPI collectives, each stage in the dependency chain would require a separate *MPI_Alltoall()*/*MPI_Alltoallv()* pair leading to unnecessarily frequent synchronization and $\mathcal{O}(P \log P)$ overhead for each all-to-all communication amongst the P processors. As shown in the message diagrams in Figure 5.4, the BSP-style algorithm communicates in bursts at the end of a computation step (Figure 5.4a) while the active-message-based algorithm sends messages as coalescing buffers fill up, even if computation has not yet finished (Figure 5.4b). Because the BSP-style algorithm waits until all computation is done to do any communication, its communication and computation cannot be overlapped, leading to high usage of the network in some regions of the algorithm and no usage in others.

(A) BSP



(B) Active Messages

FIGURE 5.4. Message trace of (a portion of) a breadth-first search algorithm in BSP and Active Message styles. Process ranks are represented on the Y axis. Each vertical arrow represents a message; arrow size is proportional to message size.

5.4. Parallel Search

Both of the wavefront expansion patterns above use bounded descent (or lookahead) to maintain reasonable work efficiency. This limits the available parallelism by ensuring that only solutions reasonably close to the current optimal solution are explored. More work-efficient solutions synchronize more frequently, while infrequently-synchronizing approaches are often less work-efficient but expose more parallelism. One pathological case of balancing work efficiency and parallelism is to discard the notion of work efficiency entirely. The parallel search pattern does this by phrasing the entire computation as a single epoch with unconstrained parallelism. This pattern is equivalent to the label-correcting wavefront pattern where $\Delta = \infty$.

Reachability, computing the set of vertices reachable from one or more sources, is an example of a parallel search kernel that is very straightforward to express using active messages. A traditional implementation using MPI collectives would most likely utilize the same structure as the breadth-first search example in Section 5.2. This would require a number of epochs equal to the distance of the farthest reachable vertex from the source. Using generalized active messages with message sends allowed inside message handlers, reachability can be performed in a single epoch, as shown in Algorithm 4.

Alg. 4: Reachability determination using the Parallel Search pattern.

```
   Input: Graph 𝒢 = ⟨V, E⟩, vertex s
   Output: ∀v ∈ V: visited[v] = 1 iff v reachable from s
1  message handler explore(Vertex v)
2      if visited[v] = 0 then
3          visited[v] ← 1;
4          parallel foreach neighbor w of v do
5              send explore(w) to owner(w);
6  procedure main()
7      active message epoch
8          if owner[s] = this rank then
9              run handler for explore(v);
```

5.5. Coarsening and Refinement

The coarsening and refinement pattern differs from search or wavefront patterns because rather than starting with a single source vertex and having to discover additional parallelism, algorithms based on this pattern start with a logical work list containing a set of vertices or edges which can be processed in parallel. Vertices and edges are contracted to representative "supervertices" shrinking the work list until it is empty and the algorithm is complete (Figure 5.2c. As the working set of the algorithm becomes smaller, parallelism is reduced and contention increases as each active supervertex represents a larger set of vertices and edges. Figure 5.1c shows an example of this process of contraction. The connected components algorithm due to Shiloach and Vishkin is one classic example of this pattern **[147]**.

The Shiloach-Vishkin connected components algorithm, shown in Algorithm 5, consists of two phases. The first phase, *hooking* (lines 20–24), combines trees if there is an edge between them. The second phase, *shortcutting* (lines 25–31), contracts or flattens trees to a *root* or representative vertex. Component membership is tracked using a *parent* array in which $parent[u] = parent[v]$ iff u and v are in the same component. Hooking must be performed carefully to avoid cycles and requires the ability to scan all the adjacent vertices of each component and access those vertices' *parent* values. In a BSP model,

Alg. 5: Shiloach-Vishkin connected components using active messages.

```
Input: Graph 𝒢 = ⟨V, E⟩, vertex s
Output: ∀v ∈ V: parent[v] is an identifier for the connected component that contains v
1  hooked ← doubled ← false;
2  message handler hook(Vertex pv, Vertex u)
3      pu ← parent[u];
4      if owner(pu) = this rank then
5          ppu ← parent[pu];
6          if pv < ppu then
7              hooked ← true;
8              parent[pu] ← pv;
9      else
10         send hook(pv, pu) to owner(pu);
11 message handler pointer-double(Vertex pv, Vertex v)
12     send pointer-double-reply(v, parent[v]) to owner(v);
13 message handler pointer-double-reply(Vertex v, Vertex ppv)
14     if ppv < parent[v] then
15         doubled ← true;
16         parent[v] ← ppv;
17 procedure main()
18     parent[v] ← v, ∀v ∈ V;
19     do
20         hooked ← false;
21         active message epoch
22             foreach v ∈ V do
23                 foreach neighbor u of v do
24                     send hook(parent[v], u) to owner(u);
25         do
26             doubled ← false;
27             active message epoch
28                 foreach v ∈ V do
29                     pv ← parent[v];
30                     send pointer-double(pv, v) to owner(pv);
31         while doubled on any node;
32     while hooked on any node;
```

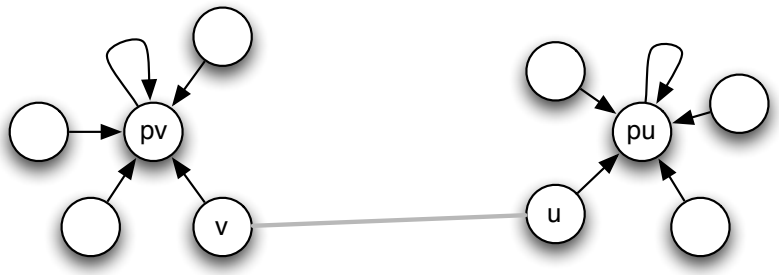
MPI_Alltoall()/*MPI_Alltoallv()* can be used to exchange *parent* values, but two stages of communication (and thus synchronization) may be required when u, v, and $parent[v]$ are all on different nodes. Shortcutting also requires two *MPI_Alltoall()*/*MPI_Alltoallv()* pairs

to perform the operation $\{\forall v \in V : parent[v] \leftarrow parent[parent[v]]\}$ (one to send a request for the parent of each $parent[v]$ value, then a reply to that request).

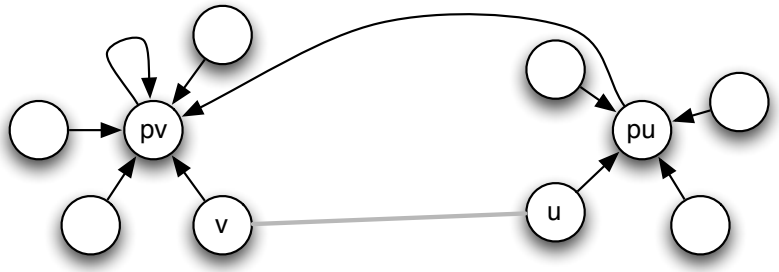
The active message version of this algorithm avoids copying values in the *parent* array between processors and thus uses significantly less storage, in addition to requiring less synchronization. The multiple levels of messages (either requests and replies or something more complicated) can share a single epoch in the active message version, without extra synchronization between the stages. The outer loops in the code use reductions (sums) to determine globally whether parents were changed and whether any hooks occurred; this step is done using the end-of-epoch sum capability in Active Pebbles. Hooking and shortcutting then occur as before.

When hooking two vertices v and u connected by an edge $v \leftrightarrow u$ (Figure 5.5a), up to three processes may be involved: the owners of v, u, and $parent[v]$, since hooking requires sending $parent[v]$ to $owner(parent[u])$. The owner of v fills in the value of $parent[v]$ (Algorithm 5 line 29) and sends a pebble to $owner(u)$ to fill in the value of $parent[u]$ (line 3). The values of $parent[u]$ and $parent[v]$ must then be sent to $owner(parent[u])$ (line 10). At $owner(parent[u])$, $parent[parent[u]]$ may be assigned to $parent[v]$ if $parent[parent[u]] < parent[v]$ (lines 6–10) resulting in the tree shown in Figure 5.5b. Shortcuts are applied for local operations. For hooking to terminate, a monotonic ordering must be defined on vertices to ensure that only one update will ultimately succeed for each target vertex; it can also be used to eliminate redundant updates to the same parent. Though hooking can generate message chains of length 2, the messages can be interleaved arbitrarily and only a single epoch is required.

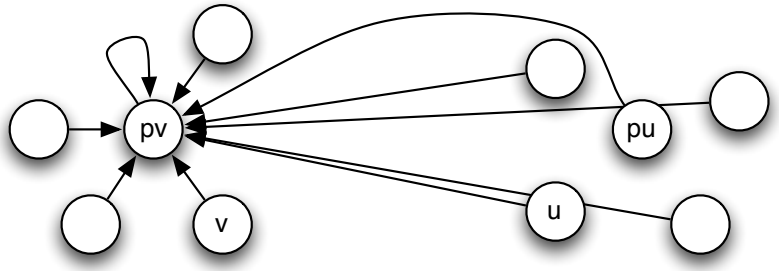
The shortcutting stage involves each process performing pointer-doubling (Figure 5.5c) of the form $\{\forall v \in V : parent[v] \leftarrow parent[parent[v]]\}$ using a two-stage process, all executed in one epoch. The owner of v first sends a request to perform $parent[v] \leftarrow parent[parent[v]]$ to the owner of $parent[v]$, giving it the value of v for the reply (line 30). The owner of $parent[v]$ then fills in $parent[parent[v]]$ and sends a message to perform the update back to the owner of v (line 12), which then processes it (line 16).



(A) Two components connected by a single edge



(B) Hooking of *pu* onto *pv*



(C) Pointer doubling produces a new component rooted at *pv*

FIGURE 5.5. Hooking and pointer doubling in Shiloach-Vishkin connected components. Black arrows indicate the parent of a vertex and the grey line represents an inter-component edge.

5.6. Iterative Methods

Iterative methods such as PageRank [**131**] or graph layout algorithms [**70**, **98**] often iterate computations with the same structure over all the vertices and edges in the graph until a fixpoint is reached (i.e., no changes occur for a round of updates). As illustrated in Figure 5.1d, every vertex and edge in the graph is active in each iteration. This yields

consistent parallelism as the work available remains constant (Figure 5.2d). Parallelization is straightforward using halo-zone communication with MPI or techniques similar to those described for coarsening and refinement algorithms above using active messages. In contrast to other individual algorithm patterns, the fact that these algorithms repeatedly traverse the same graph structure present opportunities for utilizing information discovered in earlier iterations to optimize subsequent traversals.

5.7. Combinations of These Patterns

What makes the algorithm patterns presented in Sections 5.2 - 5.6 useful is that they occur repeatedly in higher-order algorithms. For example, the strongly connected components of a directed graph can be found by finding the intersection of a pair of breadth-first search trees [**65**]. Betweenness centrality can be parallelized using a label-correcting SSSP algorithm combined with a pair of topological-order traversals [**61**]. Optimizing the execution of these patterns which are used as kernels of more complex algorithms is one important part of achieving high performance. Another important element in this goal is scheduling the kernels themselves so as to avoid coarse-grained synchronization between them. Projects in numerical computing [**153**] have demonstrated the effectiveness of scheduling applications consisting of multiple, dependent algorithm kernels in a fine-grained fashion in order to balance computations and ensure less time is wasted waiting for global synchronization. "Fusing" graph kernels at a much finer granularity offers similar benefits with regard to computational balance while also offering temporal locality (frequently absent in graph computations). When active messages are used to express graph algorithms, kernel scheduling is handled automatically by the runtime layer, provided there is no external synchronization preventing overlapping kernels. Kernel fusion can be accomplished by combining the message dependencies of the individual kernels. This makes the active message formulation extremely powerful from an optimization standpoint. Furthermore, these optimizations may be applied dynamically at runtime as the structure of the input data, and thus the computation, is discovered.

5.8. Conclusion

The algorithm classification presented in this chapter groups algorithms according to how the structure of the available parallelism varies over the runtime of the algorithm. This serves to both characterize the types of applications targeted by this work, and to allow the straightforward extension of the techniques presented here to other algorithms. By observing which of the classes described a new, candidate algorithm falls into and examining the implementation of the algorithm from that class presented in this thesis, future algorithm implementers and designers can infer possible design and implementation choices. Perhaps more powerful, the probable *performance* of that algorithm implementation in the active message model may be inferred. This enables vendors and runtime developers to select appropriate benchmarks from this classification and be fairly certain that hardware and software tuning that is effective at accelerating the benchmark algorithm will also provide good performance for other algorithms in the same class. Subsequent chapters will examine algorithms from each of these classes in more detail.

6

Expressing Graph Algorithms Using Active Messages

Chapters 3 and 4 discussed a general purpose active message library, AM++, and a set of techniques for separating the expression of data-driven algorithms from their implementations, Active Pebbles. These tools are applicable to a range of data-driven problems. In this chapter we examine how these techniques apply to graph problems in particular and discuss the architecture of the Parallel BGL 2.0, an active message-based redesign of the original Parallel BGL.

Graph problems have a number of inherent characteristics that distinguish them from traditional scientific applications [**117**]. Graph computations are often completely *data-driven*: they are dictated by the vertex and edge structure of the graph rather than being expressed directly in code. Execution paths and data locations are therefore highly unpredictable. Moreover, the connectivity of many graphs is not determined by 3D space (as is the case for discretized partial differential equations), resulting in data dependencies and computations with *poor locality*. Finding high quality partitions in such situations is computationally impractical since no good separators may exist [**63**, **108**]. The resulting graph distributions have a high ratio of surface area to volume which can significantly limit scalability. Finally, graph algorithms are often based on exploring the structure of a graph rather than performing large numbers of computations on the graph data, which results in *fine-grained data accesses* and a *high ratio of data accesses to computation*. Latency costs (memory accesses as well as communication) can dominate such computations.

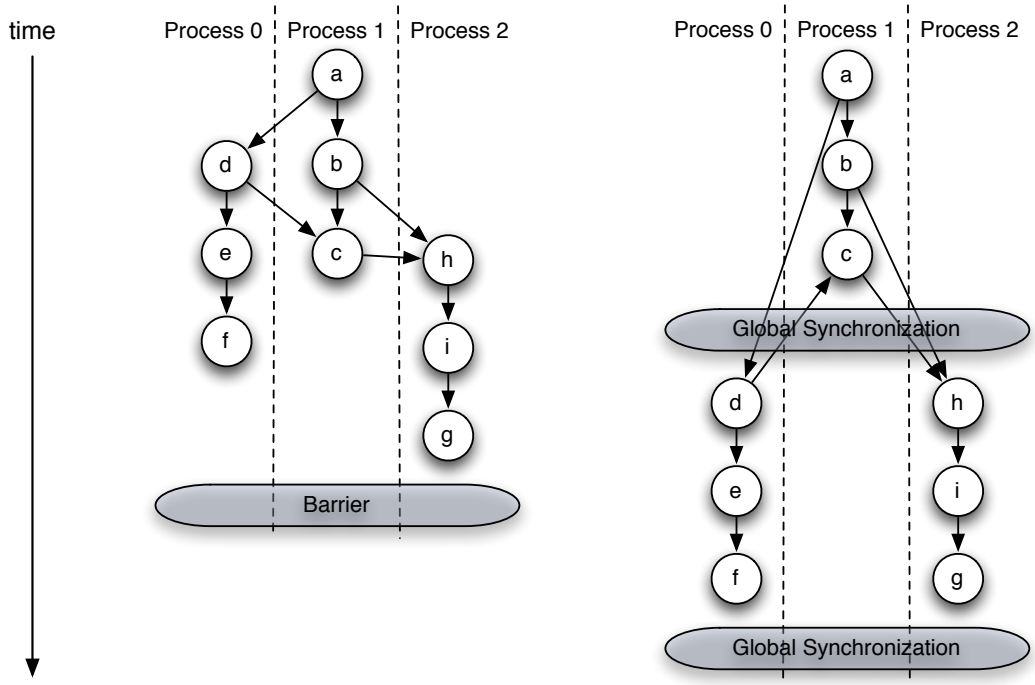
This latency can either be managed—by minimizing overheads in the critical path of the computation—or hidden—by using asynchronous operations in parallel. Minimizing overhead calls for highly optimized kernels and a careful balance between latency and bandwidth in both the local memory subsystem and in the distributed-memory communication layer. Providing asynchronous parallelism in a fashion that is largely transparent to users has spawned a host of hardware features. These architectural features generally rely on the presence of a single shared address space accessible to all hardware resources. Symmetric multiprocessing can be traced at least as far back as the IBM System/360 Model $67-2$ [**88**]. The utility of multiple hardware contexts for tolerating high latency operations was discovered once adequate on-chip real estate was available to support them [**168**]. Simultaneous multithreading [**161**] reduced the overhead of switching between these hardware contexts by allowing multiple threads to issue instructions to a single pipeline simultaneously. A number of architectures pushed the boundaries with regard to the number of concurrent thread contexts and the minimization of context switching overhead, ultimately culminating in systems such as the Tera MTA [**10**]. Regardless of the hardware supporting the logical threads programmed by end users, these techniques are limited in

scalability by the shared memory required to support them. The Tera MTA-2 scaled this shared memory to impressive sizes by using a modified Cayley graph whose bisection bandwidth scales linearly with the number of processors. This custom memory subsystem proved too expensive for commercialization and the next-generation of the architecture, produced as the Cray XMT [**102**], utilized a 3D torus instead. While hardware features can effectively create asynchronous parallelism in the presence of shared memory, achieving similar effects on distributed-memory systems requires new abstractions to encapsulate both data and control flow.

When considering graph algorithms, there is a second dimension to consider beyond the ability of the hardware to support application scalability. As discussed in Chapter 5 there are a number of graph patterns which reoccur. However, these patterns are often customized in a myriad of ways to fit the expected structure of the input graph and the particular algorithm variant desired. Just as special types of sparse matrices—banded, symmetric, triangular, Hermitian, etc.—may require algorithmic modifications to ensure optimal execution, variations in graph structure also motivate algorithmic modifications. While there is a cottage industry in random graph models [**21**, **33**, **62**, **114**, **142**, **167**], classifications are not discrete as with sparse matrix types and thus algorithms often include heuristics rather than discrete variations. Supporting this sort of algorithmic variation and user customization efficiently requires programming abstractions which are both simple and composable. A variety of asynchronous programming constructs exist; As early as 1981, extensions to HEP Fortran [**52**] allowed asynchronous task creation via the *CREATE* and *RESUME* keywords. Programming language constructs such as promises [**68**] and futures [**19**] allow independent, asynchronous operations to be expressed but still require underlying implementations. Active messages are lighter weight than asynchronous tasks while combining the programmability of futures/promises with efficient distributed-memory implementations.

Formulating graph algorithms using active messages has the dual advantages of being both conceptually simple while not over-constraining implementations. To convert a sequential algorithm expressed in an iterative form using loops, the loop bodies become

message bodies. Rather than the coarse-grained, implicit dependencies expressed by the ordering of loop iterations, the active message form of the algorithm expresses the minimal set of dependencies necessary for correctness by allowing message handlers to send other messages directly.



(A) Active Message

(B) Bulk Synchronous Parallel

FIGURE 6.1. Example execution of two programming models for a simple task-graph.

This method of expressing dependencies allows the critical path of the application to be efficiently captured and executed without artificially extending it through coarse-grained programming constructs. Figure 6.1 demonstrates how fine-grained active messages can reduce the execution time of an example task graph spread across three processes by reducing synchronization. Coarse-grained programming models require multiple synchronization steps to ensure progress for remote events (Figure 6.1b) which can delay events on the critical path of the computation and extend application runtime. Fine-grained models such as active messages (Figure 6.1a) reduce total runtime by accurately capturing the

critical path of an application and executing it without artificial delays due to excessive synchronization. Active messages allow the user to directly modify algorithm code in an understandable fashion (individual vertex and edge operations) rather than requiring the use of complicated, vendor-tuned primitives which operate at a coarser level.

With this approach, we are able to capture the fine-grained dependency structure of graph computations at runtime, exposing maximal parallelism. While this fine-grained expression may not be directly suited to certain hardware (e.g., due to message injection rate limits for network adapters), the active message model allows the runtime system to adapt the algorithms to the hardware using coalescing and other transformations. Deferring these decisions until runtime is especially appropriate for graph algorithms as the structure of the graph determines the computational structure of the algorithm and thus both are discovered dynamically.

Finally, the active message phrasing permits both shared- and distributed-memory parallelism, and is amenable to acceleration as well. Separating the expression of an algorithm (the code in the active messages) from the implementation (the messages themselves) enables performance portability across a variety of platforms without having to change the algorithm specification. A single algorithm specification may be executed using an active message library [**26**,**106**,**173**] in a distributed-memory environment, a threading library [**130**, **170**] possibly in combination with work-stealing [**24**] in a shared memory environment, and hardware-specific programming environments [**1**, **155**] targeting various kinds of accelerators. Most importantly, arbitrary combinations of these hardware environments are also supported.

6.1. Generic Graph Library Design

Existing parallel graph libraries can be classified into one of two broad categories. Distributed memory libraries [**34**, **116**, **120**, **137**] utilize coarse-grained work decompositions and parallelization strategies based on BSP models of computation to create efficient communication patterns (primarily through message coalescing). Shared-memory

libraries [**23**, **58**] are able to leverage increasingly-sophisticated fine-grained synchronization primitives to support greater asynchrony and parallelism however, these implementations are limited to the resources available on a single symmetric multiprocessor. By utilizing AM++ and the Active Pebbles programming and execution model features it provides, we have designed a graph library that combines the best features of existing shared and distributed memory implementations. Fine-grained expression of graph applications provides for maximal asynchrony and parallelism while runtime optimization ensures efficient communication patterns are used.

This implementation is based on the original Parallel BGL, though significant architectural modifications have been made to replace to the Process Group communication abstraction with AM++. The original Parallel BGL included separate data-flow and control-flow constructs. Data movement was handled using Distributed Property Maps[1] while control-flow was implemented using distributed queues and other distributed data structures. This architecture relied on a relaxed consistency model for the data in the property maps in order to allow efficient communication patterns. In addition to being somewhat complicated for users to reason about, the relaxed consistency model also made it difficult or impossible to determine if associated data had arrived when control flow messages were received. This prevented possibly actionable control flow from being acted on and retired; instead forcing additional synchronization to ensure that data had arrived before remote control flow referencing it was executed.

The primary architectural modification made in the Parallel BGL 2.0 was the unification of data-flow and control-flow through the use of AM++. Other significant modifications include the extension of the Distributed Property Map component to efficiently handle shared-memory parallelism, as well as a complete restructuring of the way algorithms are expressed and consequently the reimplementation of several algorithms. The philosophy that guided this design was creating a clear distinction between algorithm expression and runtime optimization. The primary goal was to allow algorithms to be specified declaratively and with the minimal number of constraints necessary for correctness. This provides

[1] See Chapter 8 for a thorough discussion of the Distributed Property Map.

maximal freedom for the runtime to optimize algorithm execution. By specifying valid optimization semantics rather than specific mechanisms, runtime optimization is decoupled from algorithm specifications. This allows algorithms to be expressed concisely and common optimizations to be shared amongst many algorithms rather than reimplemented separately in each. Key to this decomposition is the addition of a generic policy mechanism by which end users can control the optimizations utilized by a generic algorithm.

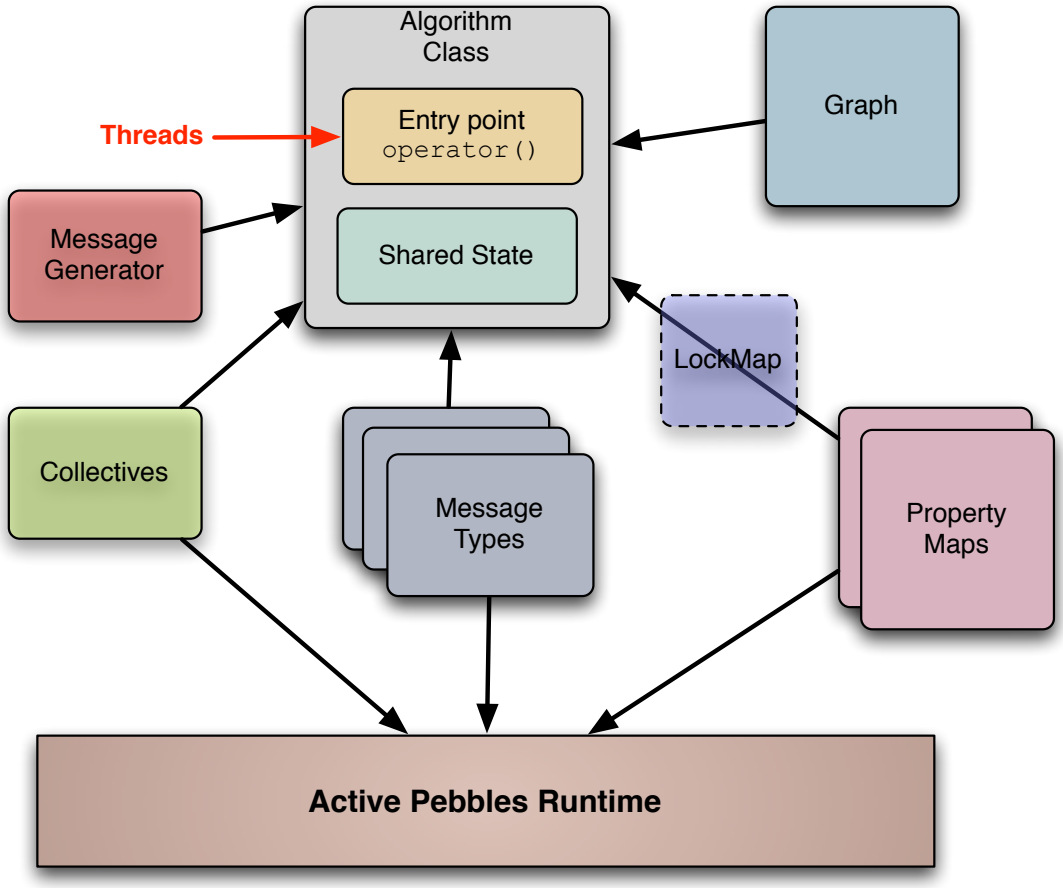


FIGURE 6.2. Structure of Parallel BGL algorithms.

Figure 6.2 illustrates the components that combine to create hybrid-parallel, message-driven algorithms in Parallel BGL 2.0. The algorithm object contains the control flow and message structure of the algorithm as well as any auxiliary data structures for managing message order. Algorithms are generic with regard to the data structures used to store

vertex and edge properties and the Distributed Property Map has been extended to form the Thread-Safe Distributed Property Map. This new concept includes both atomic updates to single property maps, as well as transactional updates to the same key in multiple property maps using the Lock Map concept. The Message Generator concept allows algorithms to create the message types used for communication with a user-defined set of optimization features. These messages are then utilized by the algorithm as a fused control-flow and data-flow mechanism to enable distributed computation. While the Thread-Safe Distributed Property Map supports *put* and *get* operations on remote keys, this functionality is generally reserved for I/O operations. The Parallel BGL 2.0 reuses the distributed graph classes from the original Parallel BGL, with modifications to replace communication which originally used the Process Group with AM++.

Data distribution utilizes the Global Descriptor concept from the original Parallel BGL. A global descriptor is an addressable entity that may reside in a local or remote address space. Global descriptors have a unique owner in addition to a local descriptor which identifies the entity within the address space of the owner. Both AM++ and the Parallel BGL 2.0 utilize the Owner Map concept which is a Distributed Property Map which takes a global descriptor as an argument and returns the rank of the owner of that entity. This abstraction of addressing allows for a wide variety of possible data distribution implementations to be encapsulated in a common interface. Data distributions which are statically computable from the descriptor are commonly used in the Parallel BGL 2.0 because of their $\mathcal{O}(1)$ space requirements. However, the Owner Map abstraction allows for fully-dynamic/fully-specified data distributions with $\mathcal{O}(N)$ space requirements for N elements, as well as a variety of implementations in between these two extremes based on partial local information and software routing.

6.1.1. Algorithm Structure. Rather than stateless polymorphic functions as in the original Parallel BGL, algorithms in the Parallel BGL 2.0 are stateful classes. The stateful

design simplifies a number of interesting use cases and provides convenient encapsulation for the communication and synchronization objects required for multithreaded, active message implementations. Algorithm objects are designed to be used with externally-managed threading. In order to have multiple threads participate in an algorithm, the user would have each thread call the algorithm. This allows users to manage thread life cycles and avoids thread spawning overhead for every algorithm invocation. Algorithm objects provide a simple entry point for threads after the algorithm has been initialized in a serial context.

Figure 6.3 shows portions of the Parallel BGL's implementation of the Δ-Stepping [**124**] single-source shortest paths algorithm. Algorithms in the Parallel BGL 2.0 are polymorphic with respect to the graph data structure (line 2), the data structures used to store vertex and edge properties (line 2), and (thread-safe) auxiliary data structures (lines 3–4). Additionally, each algorithm class is templated on the Message Generator (lines 5–6) which is a policy object that is provided by end users and used by the algorithm to create AM++ messages (lines 8–13) which utilized the specified Active Pebbles features. Algorithms specify semantically-valid optimizations (lines 26–28), which may or may not be be performed by the generated message types (lines 24–28) depending on content of the supplied Message Generator (lines 18–19). This example algorithm illustrates how algorithm classes maintain references to the graph data structures (line 36), vertex and edge properties (lines 38–40), communication constructs (lines 37 and 44), and synchronization mechanisms (line 43). Maintaining this state, possibly in combination with auxiliary information, allows algorithms to be invoked simply by multiple threads (line 32) and provides a basis for incremental algorithms. Incremental algorithms take a partial solution and a set of changes to the graph and its properties and return the updated solution. For many algorithms this approach requires significantly less time than recomputing the solution from scratch.

6.1.2. Thread-Safe Distributed Property Maps. Distributed Property Maps were an integral part of the original Parallel BGL. They provide a distributed data model and enabled

```
template<
  typename Graph, typename DistanceMap, typename EdgeWeightMap,
  typename Bucket =
    append_buffer<typename graph_traits<Graph>::vertex_descriptor,10>,
  typename MessageGenerator = amplusplus::simple_generator<
                 amplusplus::counter_coalesced_message_type_gen> >
class delta_stepping_shortest_paths {
  typedef typename MessageGenerator::template
    call_result<vertex_distance_data, vertex_distance_handler,
              vertex_distance_owner<OwnerMap, vertex_distance_data>,
              amplusplus::idempotent_combination_t<
                boost::parallel::minimum<Dist>, Dist> >::type
  RelaxMessage;

public:
  delta_stepping_shortest_paths(
    Graph& g, DistanceMap distance, EdgeWeightMap weight, Dist delta,
    MessageGenerator message_gen = MessageGenerator(
       amplusplus::counter_coalesced_message_type_gen(1 << 12)))
    : dummy_first_member_for_init_order(
       (amplusplus::register_mpi_datatype<vertex_distance_data>(),0)),
      g(g), transport(g.transport()), distance(distance),
      weight(weight), delta(delta), owner(get(vertex_owner, g)),
      relax_msg(message_gen, transport,
         vertex_distance_owner<OwnerMap, vertex_distance_data>(owner),
         amplusplus::idempotent_combination(
                              boost::parallel::minimum<Dist>(),
                              std::numeric_limits<Dist>::max()))
  { initialize(); }

  void set_source(Vertex s) { source = s; }
  void operator() (int tid) { run(source, tid); }
  void run(Vertex s, int tid = 0);

protected:
  const Graph& g;
  amplusplus::transport& transport;
  DistanceMap distance;
  EdgeWeightMap weight;
  OwnerMap owner;
  Dist delta;
  Vertex source;
  shared_ptr<amplusplus::detail::barrier> t_bar;
  RelaxMessage relax_msg;
};
```

FIGURE 6.3. Excerpt of Parallel BGL Δ-Stepping single-source shortest paths code.

PGAS-style data access in C++ [**59**]. Distributed property maps support user-specified reduction operations for combining writes from multiple processes however, this mechanism is not thread-safe. In most algorithms in the Parallel BGL 2.0, active messages communicate both control-flow and any associated data in a single communication operation thus, processes do not directly access non-local keys via *put* and *get* operations during algorithm execution. However, multiple threads may concurrently access the *local* portion of a Distributed Property Map. By making the property maps which contain the vertex and edge properties manipulated by algorithms thread-safe, we enable fine-grained parallelism at the algorithm level without the need for complex synchronization or partitioning data amongst threads.

In this sense, the "Thread-Safe" portion of the Thread-Safe Distributed Property Map is something of a misnomer. All local operations are thread-safe and an *atomic_put* method is provided to enable thread-safe remote put-accumulate behavior. However, none of the ghost-cell storage classes for caching non-local keys are thread-safe. The Distributed Property Map and thus also the Thread-Safe Distributed Property Map are generic with respect to the manner in which ghost cells are implemented therefore if a thread-safe ghost cell implementation is developed it would be straightforward to integrate. As the remote *put*/*get* interface is largely reserved for IO, the current restriction that it be utilized in a single-threaded context is not considered to be particularly onerous.

Atomic access to property map data is only a portion of the functionality required to allow property maps to support the hybrid parallelism that is a key feature of Parallel BGL 2.0. Not all property types have corresponding processor atomic operations e.g., multi-word types, structures, and other compound types. In other cases, transactional updates to multiple, non-contiguous properties may be required. The Parallel BGL 2.0 implements the Thread-Safe Distributed Property Map concept which extends the Distributed Property Map in the following three ways:

Atomic access to individual properties: The `exchange()` operation extends the `put()`/`get()` interface with an atomic compare and swap operation.

Lock Map: This concept allows a key to be locked, enabling updates to be performed across multiple property maps. Locking granularity is adjustable by modifying the mapping of keys to locks.

Metaprogramming to select appropriate atomic access method: By examining the value type of the property map and the atomic operations available on a platform, Thread-Safe Distributed Property Maps use atomic operations where available and seamlessly fall back to locking via a Lock Map when necessary.

```
1  bool try_hook(Vertex v, Vertex u) {
2    Vertex pv;

4    bool hooked;
5    do {
6      pv = get(parent, v);
7      if (!compare(u, pv)) // if u is not better than v's current parent
8        return false;

10     typename LockMap::value_type lock = locks.template maybe_lock
11                   <boost::parallel::atomics_supported<Vertex> >(v);
12     hooked = maybe_exchange(parent, v, pv, u);
13   } while(!hooked);

15   return true;
16 }
```

FIGURE 6.4. Example from Parallel BGL 2.0 connected components algorithm which illustrates how metaprogramming is used to select between processor atomics and locking automatically.

Figure 6.4 shows a portion of the Parallel BGL 2.0's implementation of the Shiloach-Vishkin connected components algorithm [**147**]. This example illustrates how metaprogramming is used to select the appropriate method for atomic update depending on the value type of the property map being updated. In this example, the property map (*parent*) contains the representative vertex identifying the component to which each vertex belongs. This function takes two vertices *v* and *u* and makes *u* the parent of *v* if the *compare* function indicates that *u* is better than the current parent of *v*. The *Vertex* type depends on the type of the graph being operated upon by this algorithm. The algorithm is generic with

regard to the graph type, so there are an infinite number of possibilities, but in order to illustrate the reason for this design it is sufficient to examine two of them.

In the case of the Parallel BGL's *compressed_sparse_row_graph* graph class, the *Vertex* type would be an integral type of some variety. Modern processors support a variety of compare-and-swap instructions up to and including double-word variants which could perform this update atomically. In this context, the *atomics_supported<Vertex>* statement (line 11) would evaluate to ***true*** *at compile time* and the *maybe_lock* function would become a no-op (using *boost::enable_if* [**93**]) which the compiler would optimize out. The *maybe_exchange* method incorporates a similar approach to select between a compare-and-swap instruction and a simple write. The result is an implementation produced by the compiler that is exactly as if the user had written the compare-and-swap instruction explicitly.

If instead of the Parallel BGL's *compressed_sparse_row_graph* graph class a user were instead to use the *adjacency_list* class then the *Vertex* would be a ***struct***. Depending on the size of the members, memory layout, alignment, and availability of a double-word compare-and-swap instruction it may still be possible to perform the update on line 12 atomically. However, in some cases the available atomic primitives would be inadequate and necessitate an approach which uses locking. In this case the *atomics_supported<Vertex>* primitive would be ***false***, and the *maybe_lock* function will use the Lock Map (*locks* in this example) to find the lock associated with vertex *v* and return it to the caller. The Lock Map uses RAII [**156**] techniques in order to provide safe and simple locking. RAII prevents many types of resource leaks by giving the compiler responsibility for ensuring exception-safe deallocation of objects. In this example, when the *lock* leaves scope on line 13 the underlying mutex is released by the *lock* object's destructor. This prevents users from forgetting to release a lock. As the Lock Map now ensures atomic access to *v*'s entry in *parent*, *maybe_exchange* on line 12 becomes a *put*. The effect in this scenario (using an *adjacency_list*) is that the compiler generates code that is equivalent to a hand-written implementation using locking.

By combining the atomic-update interface to the Thread-Safe Distributed Property Map with the Lock Map, algorithms can be written in a fine-grained and portable fashion. Metaprogramming is used to generate the appropriate implementations at compile-time based on user-provided types and architectural features of the current platform. The example in Figure 6.4 utilized a single property map, but the Lock Map was specifically designed to allow for atomic updates to multiple non-contiguous property map values associated with a single key. Additionally, the mapping of keys to locks and the number of locks can be customized in order to balance contention and storage overhead. This allows locking-granularity to be modified independent of the algorithm or property maps utilized. The Lock Map abstraction enables transactional updates using efficient locking techniques when required, and uses metaprogramming to generate efficient atomic primitives when possible. This local data model provides a suitably rich set of semantics to express a variety of graph algorithms while also providing the efficiency of hand-coded implementations.

6.1.3. Visitors and End-User Customization. Early algorithms in the original Parallel BGL relied heavily on visitors **[71]** to support end-user customization. This design has been very successful in the sequential BGL, allowing both algorithmic composition and customization of existing algorithms. Algorithms such as Dijkstra's Single-Source Shortest Paths and A* search are composed by using Breadth First Search with a visitor that overrides some of the event points specified by the Breadth First Search algorithm. Unfortunately, the relaxed consistency memory model necessary to allow efficient communication in distributed-memory graph algorithms limit the type of visitor event points that can be implemented efficiently. The visitor implementation in the BGL consists of allowing users to read and write graph properties at various points during the *fixed* control flow of an algorithm. This separation of control flow from data flow undermines one of the key reasons for phrasing graph algorithms using active messages: increased asynchrony.

Consider for example, the *white_target* event point in the sequential BGL BFS implementation. This event point is called when the algorithm encounters a previously unseen vertex. In the sequential BGL this event point is called at most once for every vertex. In the

Parallel BGL implementations however, when a process encounters a remote vertex it can only be determined efficiently if that vertex is previously unseen *by this process*. Determining whether a remote vertex is unseen by any process and then acting on the result would require synchronous communication and distributed locking in order to prevent race conditions between the remote memory access and local action. In the absence of synchronous communication only "non-deterministic" visitor event points, which are guaranteed to be called when the condition defined by the event point holds, as well as in some cases when it doesn't, can be implemented. If the operations performed in these event points are associative and commutative then the Distributed Property Map's reduction mechanism can combine them in a fashion that produces a deterministic result. However, arbitrary operations which depend on remote data require some method of capturing local data dependencies and then sending the task to be performed to the owner of the remote data. This technique is equivalent to an active message.

One of the primary goals of formulating graph algorithms using active messages is to reduce synchronization. PGAS-like data movement layers such as the Distributed Property Map rely on synchronization in order to enforce the arrival of remote reads and writes. In the BGL's visitor implementation graph properties and other data can be read and written inside visitor events, but the control flow of the algorithm can only be indirectly influenced through manipulation of properties and exceptions. This means that visitors have limited ability to control the visibility of remote updates. Implementing collective operations necessary for synchronization inside visitor event points would be difficult if not impossible, and would increase synchronization and reduce performance (as shown in Figure 6.1).

The active message formulation of algorithms combines control-flow and data-flow into a single communication model. In this framework, describing a visitor event point requires an amount of work that approaches that required to define an active message and it's associated handler. Thus rather than utilizing the visitor pattern in the Parallel BGL 2.0 we have attempted to simplify the process of using AM++ messages directly by pushing common functionality into the runtime in the form of Active Pebbles features.

This layered approach reduces the barrier to implementing new algorithms significantly. Algorithms utilizing distributed-memory parallelism may be defined entirely in terms of AM++ messages with references to semantically-valid Active Pebbles features as will be discussed in Section 6.4. Auxiliary data structures for message ordering or thread-parallel word distribution are optional.

6.2. Converting Algorithms to Active Messages

As with most parallel programming models some effort is required to transform a sequential algorithm, or even one written for another parallel programming model, into an active message form. Explicit use of active messages is not a familiar form for expressing parallel graph algorithms. However, more standard representations (e.g., with PRAM-like [**66**] semantics) can be converted to active message form with a straightforward translation process, which we describe below. In some cases algorithms themselves contain artificial dependencies, which can be removed to further improve performance. Possible future work would be to automate some or all of this translation process, likely based on programmer annotations of what data structures are distributed and how, and what operations must be atomic, as well as to generate efficient thread-safe code for `atomic` blocks (even for larger ones that do not correspond to processor atomics).

The conversion process has three steps:

(1) Identify distributed data structures and their data distributions.
(2) Annotate where each program statement should execute, and when control flow and data could be moved between nodes.
(3) Lift active message handlers out of the code, replacing control flow movements with sends of active messages.

6.2.1. Identifying Distributed Data Structures. The conversion process starts with a program format we refer to as *pseudo-PRAM*. In this form, sequential vs. parallel loops are explicit, as well as atomic blocks, while the system is assumed to support shared memory. An example of breadth-first search in this form is given in Figure 6.5. The first step in

```
1 while (!empty(Q)) {
2   parfor (v ∈ Q, w ∈ neighbors(v))
3     atomic
4       if (test_and_set(color[w]) == 0)
5         push(nextQ, w);
6   swap(Q, nextQ);
7   clear(nextQ);
8 }
```

FIGURE 6.5. Pseudo-PRAM version of breadth-first search.

converting it to run on a distributed-memory computer is to identify the data distributions of the shared variables; in the example, those are *Q*, *nextQ*, *neighbors*, and *color*. The other variables are scalar local variables and thus are not distributed. Assume that there is a single owner map *owner* applying to all four variables: vertices are in one of the queues on a given process only if the owner map places them on that process, and the neighbors and color map entry for a vertex are only available on its owning process. Given this information, we can then assign statements and expressions to owning processes.

6.2.2. Conversion to Remote Spawns. Each statement must be executed on a process that has ownership of all expressions used in that statement; satisfying this requirement may require moving data around between subexpressions in a larger expression. The example is simpler: the primitive expressions that have data distributions are *neighbors(v)* (which must be run on *owner(v)*), *color[w]*, and *push(nextQ, w)* (the latter two of which must be run on *owner(w)*). Lifting these constraints out to larger blocks of code, the entire ***atomic*** block must be executed on *owner(w)*, while the second loop in the ***parfor*** statement must be executed on *owner(v)*—that is guaranteed by the distribution of *Q*.

In a system without distributed transactions, ***atomic*** blocks are unable to span multiple nodes, and thus having the block must execute in one process. Given information on its execution location, the ***atomic*** block must be wrapped in an ***async_spawn*** statement to move its execution to *owner(w)*. As multi-threaded code is not considered at present, the ***atomic*** statement itself is removed; in a thread-safe implementation, the test and update to *color* would need to be an atomic test-and-set with a thread-safe insertion into *nextQ*

when the test-and-set succeeds. Additionally, an ***epoch*** block needs to be wrapped around the synchronization region in the code; here, that is the outer ***parfor*** loop. Those changes produce Figure 6.6, written in a pseudocode similar to a restricted form of the explicit asynchronous spawn operations in X10 [**39**] or Chapel [**32**].

```
9  while (!empty(Q)) {
10   epoch
11     parfor (v ∈ Q, w ∈ neighbors(v))
12       async_spawn(owner(w))
13         if (test_and_set(color[w]) == 0)
14           push(nextQ, w);
15   swap(Q, nextQ);
16   clear(nextQ);
17 }
```

FIGURE 6.6. Remote-spawn version of breadth-first search.

6.2.3. Conversion to Explicit Active Messages. Given an algorithm expressed using explicit ***async_spawn*** commands, conversion to a set of active message types is similar to closure conversion used to compile functional languages. For every body of an ***async_spawn***, going from inside to outside, identify its set of free variables. For the algorithm in Figure 6.6, this set contains *w*, *color*, and *nextQ*. Of these, *color* and *nextQ* are distributed data structures, and so should be stored in the message handler; *w* is local and should be part of the active message data. Thus, the handler will be an object that contains references to the *nextQ* and *color* variables, and will accept active messages of type *Vertex* (the type of *w*). The destination of each message can be computed using the owner map *owner*. Thus, the active message handler and message type can be constructed. After that, ***async_spawn*** calls are converted into active message sends, and ***parfor*** statements are converted into normal ***for*** statements (possibly with OpenMP annotations in the threaded case). The explicit active message version of breadth-first search after these transformations is given in Figure 6.7.

One optimization that can be made is to determine whether duplicate message removal and/or some other kind of message reduction is semantically allowable. This is an

```
1   queue& nextQ; color_map& color;
2   void handle(Vertex w) {
3     if (test_and_set(color[w]) == 0)
4       push(nextQ, w);
5   }
6 };

8 register_message_type update_msg using
9   type Vertex,
10  handler update_handler(nextQ, color),
11  owner_map owner;

13 push(Q, s);
14 while (!empty(Q)) {
15   epoch
16     for (v ∈ Q, w ∈ neighbors(v))
17       send update_msg(w);
18   swap(Q, nextQ);
19   clear(nextQ);
20 }
```

FIGURE 6.7. Explicit active message version of breadth-first search.

algorithm-specific determination that can be made by checking (either manually or automatically) whether two calls to the handler with argument values related in some way can be replaced by a single call.

6.3. Graph Application Stack

A complicated system stack has evolved to support traditional HPC application development. This stack consists of applications, domain libraries, matrix-vector technical libraries, and array-based performance kernels. The encompassing execution model in this case is Bulk Synchronous Parallelism with MPI as the run-time system.

When expressing graph algorithms using active messages, most of this traditional system stack is too coarse-grained and synchronous to be useful for composing high-performance implementations. To this end, we have developed an alternative set of tools and software components to support graph applications. Figure 6.8 shows the elements of

this graph application stack and their relationships to one another. Each of these components have been discussed individually, here we focus on placing them in the context of a feature-rich graph application stack.

AM++ provides the low level active messaging primitives necessary to support higher-level abstractions while simultaneously being generic with regard to the underlying communication library. Transports exist for a number of communication libraries, with the MPI transport being the most portable. GASNet [**26**] and DCMF [**106**] transports have also been developed. The Active Pebbles programming and execution model translates applications to efficient implementations. The Active Pebbles execution model is implemented in AM++ and contains optimizations common to a broad class of data-driven applications. The programming model provides for straightforward creation of message types and their associated handlers while encapsulating common concerns such as message addressing. The Parallel BGL 2.0 uses the Active Pebbles programming model to express the communication structure of graph applications and relies, in part, on the Active Pebbles execution model to optimize communication. Active Pebbles optimizations can be specified by applications in the form of a Message Generator which is supplied by the application. This policy information is combined with information about semantically-valid optimizations contained in Parallel BGL 2.0 algorithms and data structures to produce AM++ message types which perform the specified optimizations with the supplied parameters where possible (as defined by semantic annotations in the Parallel BGL 2.0). Algorithms in the Parallel BGL 2.0 contain fine-grained synchronization constructs to enable hybrid parallelism. However, rather than spawning threads inside algorithms, applications manage threads and call Parallel BGL 2.0 algorithms concurrently from multiple threads to pass control of the threads to algorithms. This allows applications to minimize thread-spawning overhead and avoid accidental over-subscription of computing resources. As Figure 6.8 shows, applications may also use Active Pebbles features and access the AM++ runtime directly if additional functionality beyond that contained in the Parallel BGL 2.0 is required.

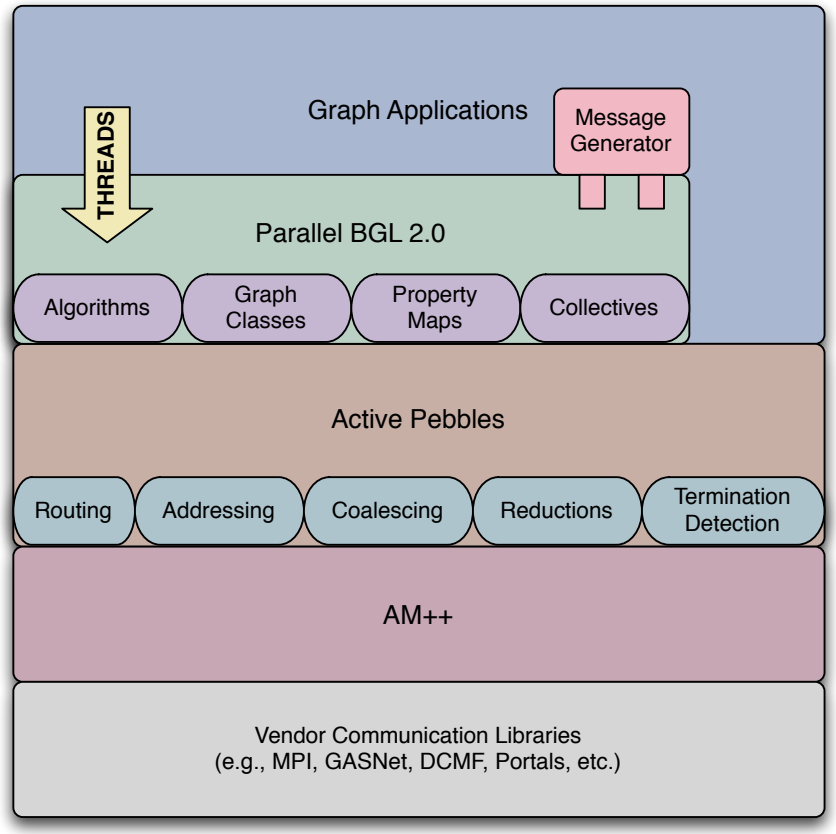


FIGURE 6.8. Application stack using Parallel BGL 2.0, Active Pebbles, and AM++.

6.4. Supporting End-User Behavioral Modification

AM++ and the Active Pebbles model provide a large degree of flexibility with regard to performance portability and tuning communication for different architectures and environments. The Parallel BGL 2.0 applies similar generic programming principles to allow end-user customization of the algorithms and data structures provided in the library. The generic design allows end-users to customize and replace data structures, utility functions which contribute to algorithm behavior, and finally control communication optimizations performed by the Active Pebbles runtime. Customization of data structures and utility functions does not differ markedly from the original Parallel BGL, or indeed the sequential BGL, thus we will focus on the Message Generator mechanism which allows end users to retroactively apply an optimization policy to the AM++ messages used by an algorithm.

6.4.1. Message Generators. Message information in the Active Pebbles model includes the data type to be sent, the C++ type of the message handler, and policies for coalescing, routing, and reductions. While the data type and message handler are algorithm-specific, other aspects are likely to be tied to hardware architecture and system configuration and possibly run-time information about the input data. Thus, we have extended Active Pebbles with a message configuration layer that separates the concerns of message semantics vs. configuration and run-time information.

The configuration layer allows library users to create message configuration objects; each of these objects contains a desired coalescing approach, buffer size, and routing topology. The object also identifies whether the user would like message reductions to be used when applicable to a particular algorithm (and if so, which cache type and parameters to use). This configuration mechanism separates the specification of an algorithm's message semantics from the mapping of those semantics to a particular implementation. Thus optimizations can be applied to algorithm specifications by end users without modifying the algorithm specifications themselves. This mechanism allows algorithms to utilize optimization techniques that were unknown at the time the algorithm was developed, as long as the algorithm's implementation (and semantics) support that type of optimization. This retroactive optimization is key to supporting performance portability across current and future architectures without the need to re-implement algorithms.

Figure 6.9 shows a portion of the Parallel BGL 2.0 implementation of Shiloach and Vishkin's [**147**] connected components algorithm. This example demonstrates how algorithm-specified semantics are combined with a user-supplied Message Generator to generate a concrete implementation. This algorithm contains three different message types (*HookMessage*, *PointerDoubleMessage*, and *UpdateParentMessage*). Lines 32 and 34 stipulate that no reductions may be applied to messages of type *PointerDoubleMessage* and *UpdateParentMessage* respectively. Line 30 indicate that messages of type *HookMessage* may be considered idempotent if they are equivalent according to the *combine* function (which may be modified by the user). If the user-supplied *MessageGenerator* includes

```
1  template <typename Graph, typename ParentMap, typename LockMap,
2         typename MessageGenerator, typename Compare, typename Combine>
3  class connected_components {
4    // This message attempts to hook p(u) onto p(v)
5    typedef typename MessageGenerator::template call_result<
6      vertex_pair, hook_handler, vertex_pair_owner<OwnerMap>,
7      amplusplus::idempotent_combination_t<Combine, Vertex> >::type
8    HookMessage;

10   // This message updates a vertex's parent pointer
11   typedef typename MessageGenerator::template call_result<
12     vertex_pair, pointer_double_handler,
13     vertex_pair_owner<OwnerMap>, amplusplus::no_reduction_t>::type
14   PointerDoubleMessage;

16   // This message is the response to PointerDoubleMessage above
17   typedef typename MessageGenerator::template call_result<
18     vertex_pair, update_parent_handler,
19     vertex_pair_owner<OwnerMap>, amplusplus::no_reduction_t >::type
20   UpdateParentMessage;

22 public:
23   connected_components(Graph& g, const ParentMap& parent,
24                        LockMap& locks, Compare compare, Combine combine,
25                        MessageGenerator msg_gen)
26     : transport(g.transport()), g(g), parent(parent), locks(locks),
27       owner(get(vertex_owner, g)), compare(compare), combine(combine)
28       hook_msg(msg_gen, transport, vertex_pair_owner<OwnerMap>(owner),
29         amplusplus::idempotent_combination(
30           combine, graph_traits<Graph>::null_vertex())),
31       pointer_double_msg(msg_gen, transport,
32         vertex_pair_owner<OwnerMap>(owner), amplusplus::no_reduction),
33       update_parent_msg(msg_gen, transport,
34         vertex_pair_owner<OwnerMap>(owner), amplusplus::no_reduction)
35   { ... }

37 private:
38   amplusplus::transport& transport;
39   Graph& g;
40   const ParentMap& parent;
41   const OwnerMap& owner;
42   LockMap& locks;
43   Compare compare;
44   Combine combine;
45   HookMessage hook_msg;
46   PointerDoubleMessage pointer_double_msg;
47   UpdateParentMessage update_parent_msg;
48 };
```

FIGURE 6.9. Excerpt of Parallel BGL connected components implementation.

a reduction policy then the resulting algorithm instantiation will perform idempotent reductions for messages of type *HookMessage*. However, If the *MessageGenerator* does not include a reduction policy then no reductions will be performed, regardless of the fact that they may be semantically valid.

```
1  typedef amplusplus::simple_generator<
2    amplusplus::counter_coalesced_message_type_gen> MessageGenerator;
3  MessageGenerator msg_gen(
4    (amplusplus::counter_coalesced_message_type_gen(4096)));

6  // Call connected components with no routing and no reductions
7  // using counter-based coalescing buffers of 4096 messages
8  boost::graph::distributed::connected_components<
9    Graph, ParentMap, MessageGenerator>
10 CC(g, parent, locks, msg_gen);

12 typedef amplusplus::per_thread_cache_generator<
13   amplusplus::counter_coalesced_message_type_gen,
14   amplusplus::rook_routing> MessageGenerator2;
15 MessageGenerator2 msg_gen2(
16   amplusplus::counter_coalesced_message_type_gen(4096),
17   10, amplusplus::rook_routing(trans.rank(), trans.size()));

19 // Call connected components with rook routing, per-thread reduction
20 // caches holding 2^10 messages, and counter-based coalescing
21 // buffers of 4096 messages
22 boost::graph::distributed::connected_components<
23   Graph, ParentMap, MessageGenerator2>
24 CC2(g, parent, locks, msg_gen2);
```

FIGURE 6.10. Two possible instantiations of the Parallel BGL 2.0 connected component algorithm with different messaging policies.

Figure 6.10 shows how two instantiations of the Parallel BGL 2.0 connected components algorithm will utilize different Active Pebbles messaging optimizations, determined by the *Message Generator* supplied.

6.5. Conclusion

Rather than using a coarse-grained PGAS-like data movement layer and ad-hoc control flow as in the original Parallel BGL, the Parallel BGL 2.0 is designed from the ground up to use active messages as it's primary communication abstraction *and* unit of control

flow. This provides large amounts of fine-grained asynchronous operations with very low overhead, which is essential for general purpose graph applications. This fine-grained work decomposition and AM++'s thread-safe active messages allow *hybrid parallelism* to be employed in many algorithms. The Parallel BGL 2.0 provides the adaptability and flexibility necessary from a high quality library by retaining, and extending, the genericity of the original Parallel BGL. These same design features enable performance portability, a fundamental requirement for successful parallel programming models.

Both versions of the Parallel BGL are generic with respect to graph data structures, properties, and other auxiliary data structure which allows efficient composition with user-defined data structures. Additionally, the Parallel BGL 2.0 uses AM++—which implements features of the Active Pebbles execution model—as it's underlying communication abstraction. This provides portability across a variety of vendor communication libraries and hardware interconnects due to the fact that AM++ is generic with respect to the underlying data Transport. The Message Generator abstraction allows Active Pebbles optimizations to be applied and tuned retroactively, and even makes it possible to write new optimizations and apply them to existing algorithms. The Parallel BGL 2.0 demonstrates the effectiveness of expressing graph algorithms using active messages by supporting higher-performance, a greater variety and degree of parallelism, and greater flexibility and adaptability than the original bulk-synchronous-parallel Parallel BGL.

7

Exposing Asynchrony and Parallelism

Chapter 6 examined how the active message-based design of the Parallel BGL 2.0 enabled flexibility and adaptability in a broad range of features: the type and degree of parallelism employed, which distributed communication libraries and optimization techniques to employ, and a variety of data structure choices. In this chapter we examine the performance implications of a fine-grained, active message-based phrasing vs. more traditional bulk-synchronous-parallelism.

As the title of this chapter implies, the primary goal of expressing algorithms using active messages is to expose maximal asynchrony and parallelism. System properties such as overhead and contention may prevent the naïve implementation of this expression from being efficient however, access to the full fine-grained expression of an algorithm allows

an optimizing runtime maximum flexibility to generate an efficient implementation. There are a number of dimensions to consider when generating a concrete implementation from the abstract specification of an algorithm. In addition to the system properties discussed above, the properties of the algorithm itself must be considered. Some algorithms express a reasonable degree of asynchrony while others are highly synchronous. Implementing a highly synchronous algorithm (e.g. breadth first search) using active messages often shows little or no performance improvement. This is because the algorithm itself prevents the runtime from leveraging any asynchrony. In synchronous algorithms, messages cannot be acted on until the conclusion of an epoch and thus we have merely moved the responsibility for buffering them from the sender to the receiver. Any additional overhead due to sending multiple small messages rather than a single large message reduces performance which cannot be offset by asynchronous progress on the receiver.

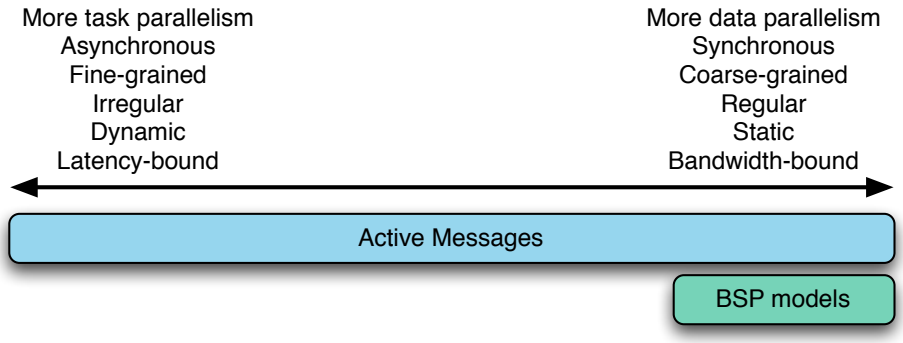


FIGURE 7.1. Range of applications capable of being efficiently expressed by programming models utilizing active messages or bulk synchronous parallelism exclusively.

While this seems to imply that bulk synchronous parallel computation still has an important role to play in graph algorithms, that is in fact not the case at all. Given an algorithm expressed using active messages, the bulk synchronous form can be viewed as a special case capable of being synthesized automatically. In the context of parallel graph algorithms, two primary types of parallelism are encountered. *Data* parallelism exists where

there exists a collection of edges that may be traversed concurrently. *Task* parallelism exists where there are a collection of *paths* which may be traversed concurrently. The distinguishing factor is the presence or absence of asynchronous progress. Let us presume a path $u \to v \to w$ where $u \to v$ belongs to "level" i and $v \to w$ belongs to level $i+1$ where the level of an edge is defined by the number of edges from some source vertex s required to reach the source of the edge. Bulk synchronous parallelism relies on global synchronization events to ensure the receipt of all communication in an epoch. Communication may not be reordered across these synchronization points. This forces all work at level i, as well as the communication round which follows level i, to be completed before any work at level $i+1$ begins. The synchronous communication round enforces the memory consistency semantics of the bulk synchronous parallel model and thus may not be weakened or removed without complicated static analysis to ensure that invalid memory semantics are not observed by the program. The active message model allows work to be executed in an arbitrary order and thus the level-wise order of bulk synchronous parallel models is a proper subset which can be achieved by inserting additional synchronization. Figure 7.1 shows the range of applications covered by active message and bulk synchronous programming models. The key insight is that *fine-grained expressions can become coarse-grained implementations, but coarse-grained expressions cannot be made fine-grained*. In the context of the previous paragraph, this means that task parallelism can *become* data parallelism if we require message recipients to wait until the end of an epoch before processing those messages.

The granularity of applications written using active messages is customizable using features of the Active Pebbles execution model. By manipulating message coalescing factors implementations may be forced to communicate larger messages less frequently. The logical extreme of this approach is to force true bulk synchronous parallelism with no overlap of communication and computation. If message handlers do not themselves send messages but rather insert data received into an auxiliary data structure then any messages received before the end of an epoch will not progress the computation and a single level of messages will be communicated and executed in each epoch. We will examine one

example of this sort of graph algorithm, breadth first search, in Section 7.3.1. The optimal implementation for algorithms of this form is bulk synchronous parallelism, and we will demonstrate that this implementation can be synthesized from an active message-based specification with no loss of performance.

Algorithms specified using active messages present the opportunity for an intelligent runtime to leverage algorithmic asynchrony in a manner that respects system-specific overheads in order to synthesize efficient implementations. Where the algorithm does not possess sufficient fine-grained asynchrony, or the system properties prevent effectively leveraging it, the runtime can synthesize coarse-grained implementations with no loss of performance. Active messages out-perform bulk synchronous parallel execution models on fine-grained, asynchronous algorithms and match their performance on coarse-grained, synchronous ones. Moreover, implementation granularity is customizable to suit applications at both extremes as well as all the applications in between.

7.1. Controlling Granularity and Avoiding Global Synchronization

Choosing algorithm granularity appropriately is the key to generating efficient implementations from algorithms specified using active messages. Algorithms which are too fine-grained suffer from excessive overhead. Conversely, algorithms which are too coarse grained suffer from starvation and poor load balancing. This implementation parameter is independent of, but can be related to, the algorithmically-defined trade-off between work-efficiency and parallelism.

In the LogP [**44**] model each message has an overhead cost of o. We assume here that messages are small enough that the G bandwidth parameter for long messages from the LogGP [**8**] model is unnecessary. This overhead can be split into o_s at the sender and o_r at the receiver [**45**]. This overhead is balanced on the receiver by the opportunity for parallel speedup. If the receiving process has an idle core then o_r and the processing of this message may be overlapped with other concurrent work and removed from the critical path of the application. If processing this message in turn generates additional messages and these messages are processed by cores which would otherwise be idle, then the reduction

in runtime is compounded. If, on the other hand, the process receiving this message is performing useful work then o_r will extend its runtime for no benefit. A similar analysis holds for the sending process and o_s. This makes it clear that the optimal communication granularity is one which sends the minimal number of messages necessary such that all available cores remain saturated.

In the shared memory domain, work-stealing [**25**] is an effective technique for solving exactly this problem. Several factors preclude this approach when implementing graph algorithms in distributed memory. First, the cost to query a remote process for available work is $2L$ (L is the network latency in the LogP model), during which the process performing the query remains idle. Second, only work bound for the process performing the query may be "stolen". The "work" in the distributed case contains data dependencies which may only be satisfied by the process to which the "work", or in this case the message, is addressed. Finding the messages addressed to a given process may be straightforward if one communication buffer per remote process is used. If software routing is used however, finding the messages addressed to a given process may require a linear scan over the communication buffer bound for the network peer on the route to the target process. Finally, in graph algorithms a process does not know a priori which other processes will communicate with it in a given epoch. Thus, an idle process is reduced to randomly polling its peers for available work.

In the absence of an efficient method for idle processes to "pull" work an efficient active message runtime will "push" work (i.e. communicate) often enough to prevent starvation, but not so often as to impose unnecessary communication overhead. Chapter 9 will explore opportunities for dynamic variation of this and other runtime parameters.

7.1.1. Load Balancing. An additional concern is load balancing. Excessive synchronization can lead to wasted cycles while idle processes wait for other processes to complete. In addition, collective operations necessary for synchronization impose overheads of their own. Given a work distribution which is uniform across an application's entire execution (an over-simplification to be sure) but which has significant variance within an

epoch, fewer epochs containing more work per epoch will be more balanced than a larger number of epochs containing less work per epoch.

A simple statistical model can be constructed to demonstrate this fact. It is not unreasonable to presume that the amount of work performed by an algorithm in terms of arithmetic operations, memory accesses, and time is proportional to the number of edge traversals. Indeed this metric has been proposed in [**127**]. While most parallel graph algorithms contain significant deterministic regions, at a given point in time the set of edges to be traversed during some future time interval may be viewed as a random variable. For example, the edges that will be traversed during an epoch are unknown at the beginning of that epoch, rather they are discovered during the epoch itself. In each epoch, some subset of the set of all edges in the graph will be traversed. The probability distribution of choosing edges in epoch $i+1$ is likely to be conditional on the edges chosen in epoch i, and possibly other epochs.

Each edge in a distributed graph may be mapped to a unique process. The question we concern ourselves with here is how balanced is the computation in a given epoch as a function of its length. "Balance" in this case amounts to the variance of the the number of edges traversed on each process. Given a basic distribution of all potential edge traversals, performing an individual edge traversal amounts to sampling from this distribution. If the underlying distribution has variance σ^2 then the variance of n samples is $\frac{\sigma^2}{n}$. The more samples (i.e. edge traversals) we have the more closely the sample variance approximates the variance of the underlying distribution. This example shows that as the amount of work in an epoch grows, the variance more closely approximates the variance of the underlying distribution. The amount of work in an epoch can be increased by keeping the graph size constant and using fewer epochs to perform the same work, or by increasing the graph size. Regardless of the approach, the more work is performed per epoch the more closely the distribution of work in the epoch approximates the underlying global distribution.

Thus longer epochs produce more balanced work distributions, to the degree that the global distribution of work is balanced. Using fewer epochs also reduces the total synchronization overhead of the application. Achieving this reduction in synchronization often requires algorithmic modifications as abstract models for algorithmic analysis often neglect the cost of synchronization.

7.2. Performance Comparison to BSP Graph Algorithms

In order to demonstrate the strengths (and limitations) of phrasing graph algorithms using active messages we have compared a number of algorithms written in an AM style to the same algorithms written in a BSP [**164**] style. While AM graph algorithms have a number of advantages with regard to leveraging on-node parallelism, we compare only the single-threaded case in this work. Multi-threaded versions of most of active message algorithms have also been implemented.

To provide a fair comparison between the programming models rather than the efficiency of the underlying communication libraries, both the AM and BSP implementations use AM++. Previous work has demonstrated that AM++ has minimal latency and bandwidth overheads relative to the data transport layer it is implemented over (MPI in this case) [**172**]. Both the BSP and AM algorithms use the routing features of AM++, as well as message reduction (caching) in some cases. Note that in a traditional BSP implementation using MPI directly, adding routing or caching capabilities would greatly complicate the code. In the AM implementations routing and caching are abstracted and encapsulated in the runtime. The primary distinction in the implementations is that while the active message algorithms overlap communication and computation aggressively and send messages from message handlers, the BSP algorithms defer communication (or at least the handling of communicated data) until the end of the epoch and handle dependent operations using algorithm-level queues to restrict each epoch to a single round of communication. It is very likely that the BSP algorithms presented here perform *better* than versions which do not leverage the routing and message reduction features of AM++; however, disabling

these features would increase memory consumption and make the problems solvable using BSP algorithms very small.

Formulating graph algorithms using active messages allows increased overlap of communication and computation *provided algorithms expose sufficient asynchrony*. Active message graph algorithms also utilize less memory because once messages are executed they can be retired. BSP implementations may communicate data before the end of an epoch, but do not process that data and free the associated memory until the conclusion of the epoch. Our results demonstrate that AM algorithms reduce memory utilization enabling them to scale to larger numbers of nodes than similar algorithms implemented in a BSP style, which fail due to memory exhaustion. In the regions where both styles of algorithms execute successfully, AM algorithms are often more efficient due to their ability to more accurately capture the critical path of the application using fine-grained asynchronous operations. However, some graph algorithms have a very coarse-grained structure that is well suited to the BSP programming model. In these cases AM implementations are unable to outperform BSP implementations because insufficient asynchrony is available in the algorithm. For these algorithms we demonstrate that the overheads imposed by the AM implementations are limited, and represent a reasonable trade off for the benefits this model provides—reduced memory utilization, performance portability, and retroactive tuning of algorithm implementations without changing their specification. In some cases we describe algorithmic modifications which expose additional asynchrony and allow AM implementations to outperform their BSP counterparts. As a fall-back measure it is feasible to synthesize BSP implementations from AM algorithm specifications by deferring message reception in the runtime until the end of an epoch.

7.3. Implementation Details

For this evaluation, we have chosen algorithms from each of the classes discussed in Chapter 5. As previously discussed, breadth-first search is an example of a label-setting wavefront algorithm (§ 5.2). For a label-correcting wavefront (§ 5.3), we chose the Δ-stepping single-source shortest paths algorithm [**124**]. The Shiloach-Vishkin connected

components algorithm consists of contraction of components to rooted "stars" via iterative hooking and pointer-doubling steps; it is an excellent example of a coarsening and refinement algorithm (§ 5.5). PageRank is a well known iterative algorithm (§ 5.6) for ranking vertices in a graph. Where suitable, we have used the Graph 500 generator to generate synthetic graphs of suitable sizes for our weak-scaling experiments. The exception to this is the connected components experiments, for which Graph 500 graphs of the degree used in the other experiments would be very likely to have a single large component and very few other components beyond isolated vertices. Erdős-Rényi graphs have a more interesting component structure, especially around average degree 2 which is the hitting time of the giant component (e.g., the point at which the second largest component contains fewer than $\log n$ vertices with high probability, where n is the number of vertices in the graph). For this reason we have used Erdős-Rényi graphs of average degree 2 for our connected components experiments. All experiments use routing with a rook graph topology (an approximately-balanced rectangle of nodes where nodes in the same row and/or column are connected) to reduce the number of message buffers required for P processes from P to $\sqrt{P}$. In the interest of comparability, each data series utilizes a consistent set of parameters for reductions and message coalescing across all problem sizes. These parameters may be suboptimal or some, or even all, problem scales shown. Further parameter studies are needed to deduce optimal combinations of runtime parameters across problem scales, and also across individual message epochs.

These experiments were performed on Challenger, a 13.6 TF/s, 1 rack Blue Gene/P with 1024 compute nodes. Each node has 4 PowerPC 450 CPUs and 2 GiB of RAM. Our experiments used Version 1.0 Release 4.2 of the Blue Gene/P driver, IBM MPI, and `g++` 4.3.2 as the compiler (including as the back-end compiler for MPI).

All of the runs have much better performance on a single node because of the lack of any messaging overheads and explicit checks for sends-to-self in AM++. Missing data values in the charts indicate that the algorithm failed to complete due to memory exhaustion.

7.3.1. Breadth-First Search. BFS[1] is a very coarse-grained, BSP algorithm. Every vertex at level i must be processed before any vertex at level $i+1$ can be explored. This provides limited asynchrony for the AM implementation to leverage as shown in Figure 7.2. The runs annotated with "1M cache" use duplicate message elimination, implemented in this case with a 2^{20}-message cache per neighbor in the routing topology. With fewer than 32 nodes and no caching we see somewhat worse performance of the AM implementation vs. BSP. Enabling caching for both algorithms eliminates the overhead of the AM version for fewer than 32 nodes. The BSP implementations begin to fail due to memory exhaustion at 256 nodes but scale relatively well to 128 nodes. It is likely that memory exhaustion would occur much sooner due to increased communication buffer utilization were AM++ routing not in use here. The poor scaling performance beyond 32 nodes for the AM implementations is attributable to a number of factors. First, the static choice of coalescing factor (2^{10} messages) is well suited to small-scale execution but provides unreasonable overhead at larger scale. This demonstrates the need to dynamically vary the coalescing factor depending on the problem size. Second, we see that the scaling is worse for the AM implementation using message reductions than for the same algorithm without reductions. As the size of the graph increases, the fraction of data which is non-local increases proportionally. With a statically-sized message cache this means that the hit rate of the cache decreases. Making cache sizes dynamic, or even dynamically enabling/disabling caching, would allow maximal benefit to be derived from message reductions without introducing undesirable overheads. A suitably intelligent runtime would be capable of monitoring the system state and making these decisions appropriately without risking memory exhaustion. This algorithm demonstrates two important points:

(1) Some algorithms lack sufficient asynchrony to benefit from AM implementations. However, features of the AM model (e.g., message reductions) may still improve performance.

[1] See Algorithm 2 in Chapter 5 for pseudocode.

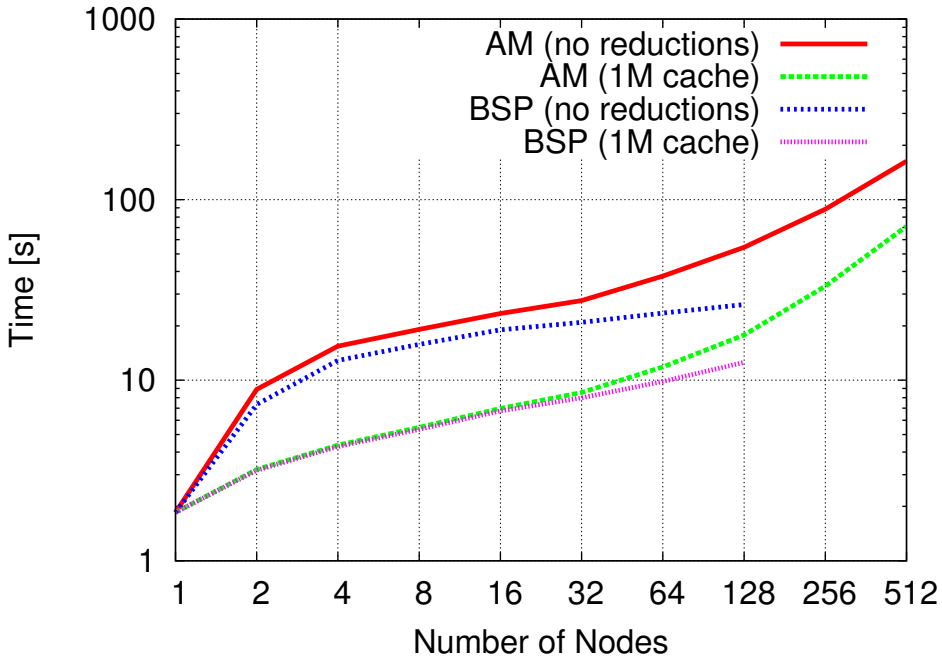


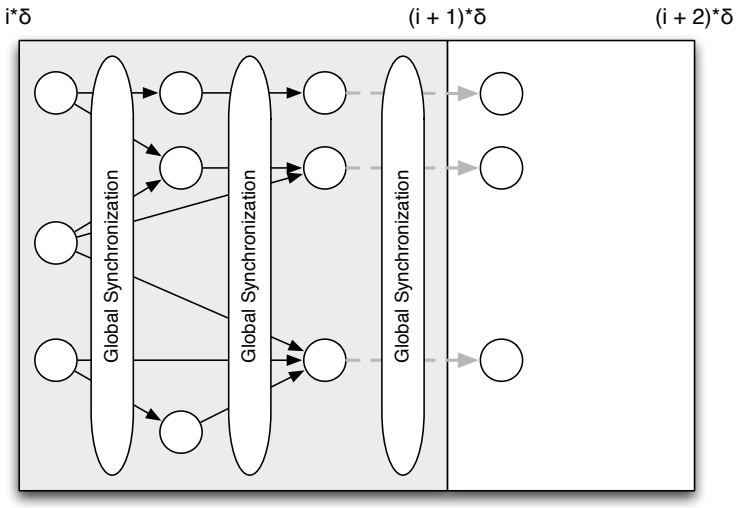
FIGURE 7.2. Breadth-first search weak scaling (Graph 500, 2^{19} vertices per node, average degree of 16, 2^{20}-message caches, average over 16 runs which are guaranteed to visit more than 100 vertices).

(2) Because algorithm execution is dependent on the structure of the input graph, static runtime optimization parameters are suboptimal vs. varying parameters dynamically in response to system state.

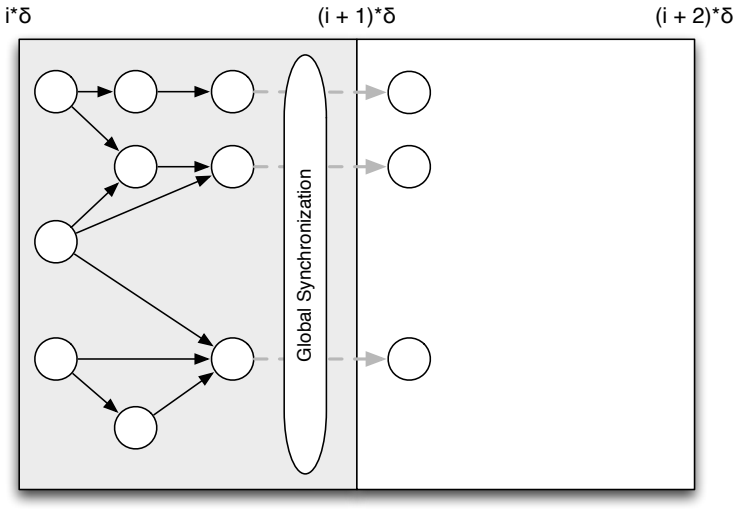
7.3.2. Δ-Stepping Shortest Paths. Figure 7.4 shows the performance of the active message and BSP implementations of Δ-stepping shortest paths[2]. Δ-stepping allows all paths with weights in the range $(i * \delta, (i+1) * \delta]$ to be relaxed concurrently. This means that within a single step of the Δ-stepping algorithm there may be a path composed of multiple incident edges which need to be relaxed in sequence. The BSP implementation will require one epoch for each edge in the path to perform this relaxation due to the fact that the relaxation request for the first edge in the path is not processed until the conclusion of the epoch (Figure 7.3a). This causes a second epoch to be initiated for the second edge in the path,

[2]See Algorithm 3 in Chapter 5 for pseudocode.

and so on for each incident edge until the path distance is at least $(i+1)*\delta$. The AM implementation requires only a single epoch to relax the entire path (Figure 7.3b). The relaxation of the first edge sends a message to the target of that edge, which triggers relaxation of the second edge, and so on until the path length is at least $(i+1)*\delta$. The termination detection feature of AM++ allows the runtime to wait until all dependent messages have been received before concluding the epoch.



(A) BSP processing of bucket $i+1$



(B) Active message processing of bucket $i+1$

FIGURE 7.3. Communication required to process a single bucket in Δ-stepping.

The additional asynchrony in the Δ-stepping algorithm allows the AM implementations to outperform the BSP implementations in almost all cases. As with BFS, the BSP algorithms fail to complete due to memory exhaustion, in this case for more than 16 nodes. The fact that memory exhaustion occurs with fewer nodes than BFS is due to the fact that the Δ-stepping algorithm maintains an auxiliary data structure to order edge traversal which consumes significantly more memory than the BFS queue. In this implementation we do not remove a longer path to a vertex from this data structure when a shorter one is found because the AM algorithm supports multiple threads and this removal is difficult to perform in a manner that is both thread-safe and efficient. The increased memory requirements of the Δ-stepping algorithm require us to use a smaller graph and message reduction cache than in the BFS experiments.

As with BFS we see that message reductions improve performance at small scale but that the benefits decrease as the ratio of cache size to overall graph size shrinks. At larger node counts we see that the reduction caches are ineffectual but that because they are fast, direct-mapped caches their overhead does not adversely affect algorithm performance. The Δ-stepping algorithm also takes a δ parameter, the width of each bucket in vertex distances. The results shown use $\delta = 4$, which performed well across a range of input sizes. In addition to dynamically tuning the message coalescing factor and cache sizes to improve scalability, tuning δ would provide increased control over the amount of parallel work and asynchrony available vs. the likelihood that relaxing an edge contributed to the final solution of the algorithm (e.g., work efficiency). This trade off between work efficiency and parallelism is omnipresent in graph algorithms and careful control of it is the key to high performance.

7.3.3. Shiloach-Vishkin Connected Components. The Shiloach-Vishkin connected components[3] algorithm requires a number of hooking and contraction steps proportional to the diameter of the largest component to complete. Because we know that it is likely that the graphs used have a giant component, we also implement an optimized algorithm

[3]See Algorithm 5 in Chapter 5 for pseudocode.

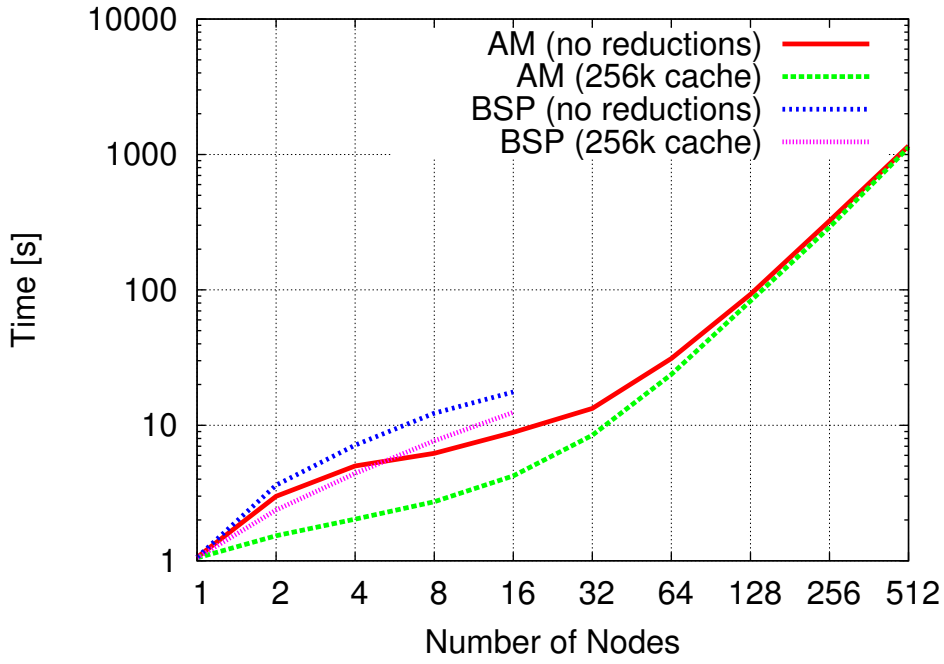


FIGURE 7.4. Δ-stepping shortest paths weak scaling (Graph 500, 2^{16} vertices per node, average degree 16, 2^{18}-element caches).

which does a parallel exploration from the highest degree vertex (which is likely to be in the giant component), and then applies the Shiloach-Vishkin connected components algorithm to the undiscovered portion of the graph to label the remaining components. We call this variant "Parallel Search + Shiloach-Vishkin" (PS+SV)[4].

Figure 7.5 demonstrates that initializing the SV algorithm with a parallel search to discover the giant component ("PS + SV") provides significant performance improvement for both AM and BSP, and improves the scaling of the AM implementation. This transformation is an example of an algorithmic refinement that exposes asynchrony which can be effectively utilized by AM implementations. Message reductions again provide performance improvements in both cases, which diminish as the graph grows. It is likely that dynamically tuning the coalescing factor would mitigate the poor scaling of the the AM

[4]See Algorithm 4 in Chapter 5 for the pseudocode of the "Parallel Search" portion of this algorithm.

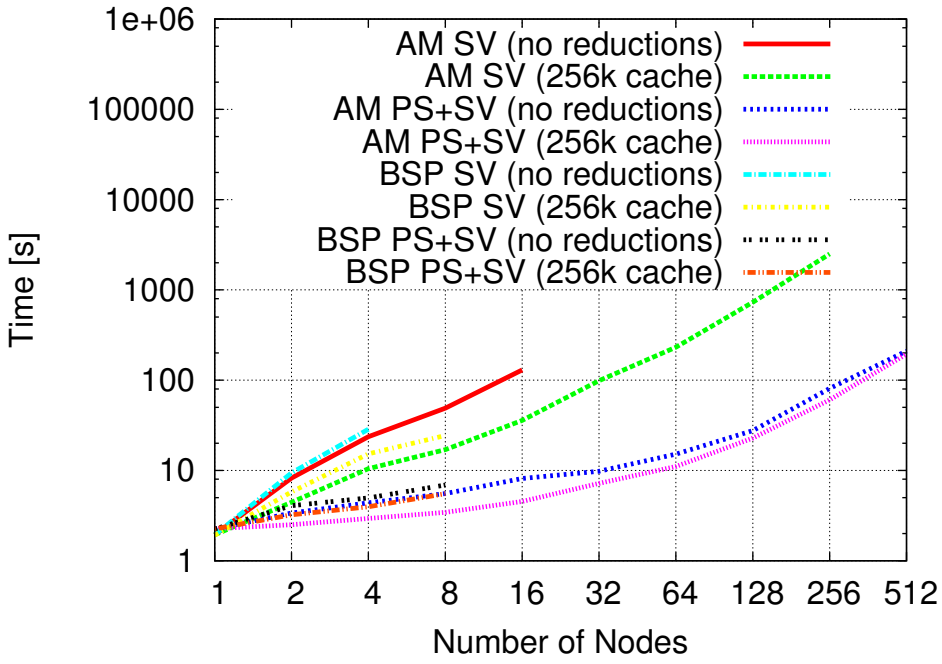


FIGURE 7.5. Shiloach-Vishkin connected components weak scaling (Erdős-Rényi, 2^{18} vertices/node, avg. degree 2, 2^{18}-element caches).[4]

"PS + SV" algorithm between 128 and 512 nodes and result in a scalable algorithm capable of executing on larger numbers of nodes.

The "PS + SV" algorithm demonstrates the ability of the AM implementation to more efficiently leverage the available asynchrony in an algorithm, as well as the ability to perform minor modifications to an algorithm to make it more suitable for AM. Figure 7.5 demonstrates that the AM implementation of the classical Shiloach-Vishkin ("SV") algorithm outperforms the BSP version because hooking and pointer-doubling operations that involve multiple communication stages can be performed in a single AM epoch, while the BSP implementation requires multiple epochs. As with BFS and Δ-stepping the BSP algorithms fail due to memory exhaustion, this time between 4 and 8 nodes.

7.3.4. PageRank. The PageRank algorithm is another very synchronous graph algorithm well suited to BSP execution. This algorithm is equivalent to the power iteration

[4]The missing data points for "AM SV" are due to the wall clock time limit expiring before the algorithm completed, not memory exhaustion. Challenger has a maximum time limit of 1 hour on all jobs.

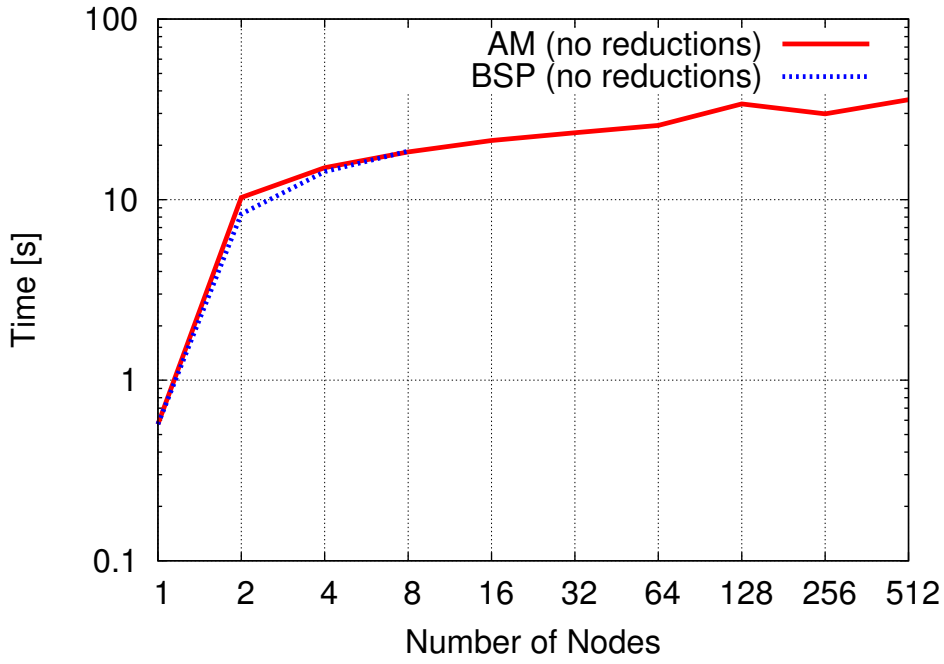


FIGURE 7.6. PageRank weak scaling (Graph 500, 2^{18} vertices per node, average degree 16, average of 20 iterations).

method for computing the largest eigenvector of the adjacency matrix of a graph. In the Parallel BGL this algorithm is implemented using a variant of the distributed property map abstraction [**59**], a PGAS-like data movement layer. Neither implementation uses AM++ reductions in this case because they have yet to be integrated into the distributed property map. Figure 7.6 shows that once again the BSP form of the algorithm fails to complete at larger node counts (at least 16 in this case), while the AM implementation runs successfully up to 512 nodes. In the range where both execute successfully the AM implementation has comparable performance to the BSP version.

7.4. Conclusion

The experiments shown demonstrate that even for static sets of runtime parameters, graph algorithms written in an active message style can out-perform coarser-grained BSP implementations in some cases. Active message algorithms improve performance by reducing global synchronization overhead and exploiting algorithmic asynchrony to hide

memory and network latency. This technique relies on the existence of asynchrony in an algorithm capable of being exploited using active messages. A number of graph algorithms lack this asynchrony due to the algorithm definition itself. In these cases, a coarse-grained, BSP-style implementation can be synthesized from the active message definition of the algorithm. In fact, the data for the BSP algorithms presented in Section 7.3 was generated using AM++ and the active message-based definition of the algorithms from the Parallel BGL 2.0. Active messages can efficiently implement both synchronous and asynchronous graph algorithms and thus represent a more powerful programming model than coarser-grained alternatives such as those offered in traditional PGAS languages [**128**, **163**] and synchronous MPI Collectives [**125**]. In addition to leveraging asynchrony in order to reduce wall clock time, active message-based algorithms also reduce resource utilization as messages may be executed and retired as they are received rather than being buffered until the end of an epoch. Chapter 9 will explore dynamic runtime techniques which may further improve the performance and reduce the memory footprint of algorithms implemented using active messages.

8

Incorporating Traditional PGAS Semantics

The goal of the active message model is to provide maximal asynchrony and parallelism for fine-grained, irregular applications. However, there are times where coarse-grained execution using traditional Remote Memory Access (RMA) techniques are useful and appropriate. Highly-synchronous regions may arise in otherwise irregular applications, especially during I/O. The work presented here is implemented in C++, which currently lacks language support for PGAS semantics. However, C++ is an excellent host language for implementing Domain Specific Embedded Languages (DSELs). These capabilities of C++ have been leveraged to implement the Distributed Property Map [**59**], a DSEL library supporting PGAS semantics, polymorphic partitioned global data structures, and a number of useful extensions. The Distributed Property Map library utilizes template

meta-programming to allow direct mapping at compile-time of high-level semantics to efficient underlying implementations. It combines flexible/extensible semantics, high performance, and portability across different low-level communication interfaces to allow PGAS programs to be expressed in C++. This library was tightly integrated with the original Parallel BGL [**77**] and has been ported, with a number of extensions, to the Parallel BGL 2.0. As in the original library, the Distributed Property Map forms the basis of the property-storage layer in the Parallel BGL 2.0.

8.1. The Case for PGAS Extensions

A wide variety of HPC applications ranging from computational fluid dynamics to N-body simulation have been parallelized using the SPMD technique. Within the SPMD domain there are several potential programming models that vary based on how data is viewed by the concurrent programs. Shared memory models assume that data exists in a shared local address space available to all programs. Message passing models assume that each program has a private address space and that data is moved between address spaces using explicit messages. The DSM model allows programmers to program the conceptually simpler shared memory model, with remote memory accesses implemented via a runtime which utilizes message passing or RMA on behalf of the user. The major shortcoming of the DSM model is that it does not allow programmers to differentiate between local and remote data in order to express the data locality available in the application. This can lead to excessive remote memory accesses and reduced performance. The PGAS model addresses this shortcoming of the DSM model by providing methods to explicitly identify local and remote data and express the spatial locality available in applications to achieve high performance.

The PGAS model has spawned a variety of languages and language extensions [**32**, **39**, **128**, **174**]. UPC [**163**] combines PGAS semantics with the performance of C and is a widely utilized language within the PGAS community. Although sometimes viewed as "C with classes", C++ is a multi-paradigm language that supports a variety of modern software

engineering techniques. While C++ currently lacks language support for PGAS semantics, techniques such as generic programming allow DSELs to be implemented as libraries without loss of performance. The PGAS semantics in a sense *are* a DSEL with respect to C++. We have developed a library-based implementation of this PGAS DSEL by utilizing a performance-preserving composition of existing libraries that support various styles of parallelism with a new interface layer which maps them to the PGAS DSEL. We call this implementation Distributed Property Maps. Distributed Property Maps are based on the Boost Property Map library [**27**]. A Property Map is an abstraction of a map (in a general sense) from keys to values. Property maps abstract data access from the underlying storage implementation, as well as encapsulating data access policies (read-only vs. read/write, etc.). The core element of Distributed Property Maps is a polymorphic global property map partitioned across multiple processes.

Distributed Property Maps consists of three primary layers. The interface layer makes PGAS-style data access semantics available to the user. The communication layer implements data movement, coalescing, and serialization. Between these two layers exists an addressing scheme for mapping data to processes and a mechanism for caching and maintaining the consistency of remote data. These three layers are combined using the generic programming paradigm to map abstract interfaces to efficient, concrete implementations at compile time. This approach allows a new communication layer or caching strategy to be easily incorporated into Distributed Property Maps. Additionally, the generic design enables the performance-preserving composition of Distributed Property Maps with third-party parallel libraries and run time systems such as OpenMP [**46**], Threading Building Blocks [**89**], PFunc [**99**], and others.

Our library based solution has the additional benefits of being flexible and extensible to explore new features that may be useful in the context of the PGAS model. In particular, extending the PGAS semantics provided by Distributed Property Maps is straightforward because it does not require modifying a compiler as with language-based approaches. Because the data access semantics are abstracted from the communication layer, porting the

library to new platforms only requires re-implementing or modifying the underlying communication layer. This made the porting of the Distributed Property Map from the original Parallel BGL to the Parallel BGL 2.0 presented in this thesis straightforward.

8.2. Generic Programming and the Distributed Property Map

Generic programming is a methodology to design software libraries that are as general as possible with the goal of allowing maximal code re-use. The term "generic programming" is perhaps somewhat misleading because it is about much more than simply programming *per se*. Fundamentally, generic programming is a systematic approach to classifying entities within a problem domain according to their underlying semantics and behaviors. The attention to semantic analysis leads naturally to the *essential* (i.e., minimal) properties of the components and their interactions. Basing component interface definitions on these minimal requirements provides *maximal* opportunities for re-use.

Several examples exist of generic programming in C++; the most well known example is the Standard Template Library [**14**, **149**], part of the C++ Standard Library [**90**]. Generic programming works by defining algorithms that are parameterized on the types of input arguments they accept. Each algorithm is *lifted* [**77**]: the minimal constraints (most generic implementation) on its inputs are found. It is then implemented to allow any argument types that satisfy those constraints. The PGAS model is the result of a lifting process on the sequential execution model where the requirement that data exist in a single shared address space is removed. Specialization can be used to take advantage of type-specific information to enable optimal performance. Due to specialization and the sophistication of C++ compilers, generic algorithms can typically be instantiated with comparable performance to non-generic versions.

The basic idea in generic programming is that of a *concept*. A concept defines a set of requirements on a type or set of types; they allow common requirements to be reused across many algorithms. Algorithms then impose requirements that their argument types satisfy the requirements of a given set of concepts. Types then *model* the concept by providing the components needed for a given concept; for example, they can define particular

functions or types. In C++, the requirements in most concepts can be satisfied without any changes to an existing type. External functions and type definitions can be provided by users of a third-party library without modifying its source code. Unlike some competing approaches, the generic programming approach does not require any modifications to existing data structures, and does not require cumbersome wrapper classes. In the STL, for example, normal C/C++ pointers are models of the iterator concepts, even though they are primitive types.

Previous work has illustrated the effectiveness of generic programming in allowing the compositional construction of parallel libraries [77]. This same approach allows us to implement Distributed Property Maps in a manner that is agnostic with respect to the underlying manner in which parallelism is expressed as well as the fashion in which parallel execution contexts communicate. Divorcing the method of achieving parallelism from the interface layer allows us to perform a variety of optimizations in a portable fashion. This flexibility is evidenced by the ease with which the Distributed Property Map was ported from the original Parallel BGL to Parallel BGL 2.0. Distributed Property Maps originally used the Process Group concept which abstracts the notion of a set of coordinating processes. AM++ largely models the Process Group concept with the exception of the manner in which some asynchronous extensions to the original Process Group are implemented (see Section 8.5 for further details). While AM++ could have been wrapped in such a manner as to expose a Process Group interface, in the interest of uniformity it was decided to integrate AM++ into the Distributed Property Maps directly. This integration required only the creation of an AM++ message type for each communication operation in the Distributed Property Map (*put()*, *get()*, etc.) and the replacement of of the original ad hoc, tagged messages with calls to the AM++ replacements. Rather than a Process Group, the updated Distributed Property Map uses the AM++ Message Generator to generate appropriate message types.

8.3. Distributed Property Map Semantics

Distributed Property Maps are designed as a DSM framework with the ability to differentiate between local and remote data where necessary. The keys in a Distributed Property Map are models of the Global Descriptor concept. A global descriptor represents an entity that is owned by some process and may reside in an address space not accessible to the currently-executing process. The global descriptor consists of two parts: the owner of the entity, which is the identifier of the process in which the entity resides, and a local descriptor, that uniquely identifies the entity within the address space of the owner.

Distributed Property Maps use a relaxed consistency model: reads return a default value until a memory barrier loads up-to-date information, and a write may not take effect until a barrier. A user-defined operation combines conflicting writes to the same location.

Figure 8.1 shows the architecture of an application using Distributed Property Maps. Distributed Property Maps originally utilized the Process Group concept which abstracts the notion of a set of coordinating processes. This was later replaced with AM++ in Parallel BGL 2.0. The Process Group concept is in turn modeled by a particular implementation which utilizes a low-level communication library such as MPI [**125**], GASNet [**26**], or a vendor-supplied library such as IBM's DCMF [**106**], LAPI [**143**], or PAMI [**105**]; MX [**74**]; or others. This decomposition allows semantic extensions to be agnostic with respect to the underlying data transport layer.

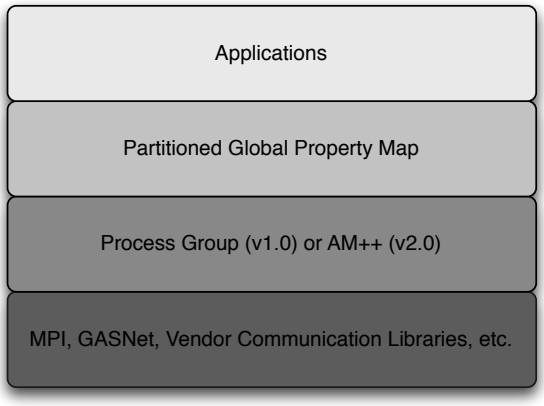


FIGURE 8.1. Distributed Property Map architecture.

Distributed Property Maps lift away the implicit assumption that all mapped values exist in a single logical address space from the original property map library. It provides the same interface functions as the sequential property map library: *get()* and *put()* methods for reading and writing respectively. In the case of remote keys, *get()* and *put()* are non-blocking calls, the implications of this will be explored in Section 8.3.4. A *synchronize()* method acts as a memory barrier across all processes that groups accesses to a Distributed Property Map into epochs and enforces completion of operations performed during an epoch at the end of that epoch. Two additional functions, *request()* and *cache()* are also provided for clarity and performance reasons which are explained in Section 8.5.

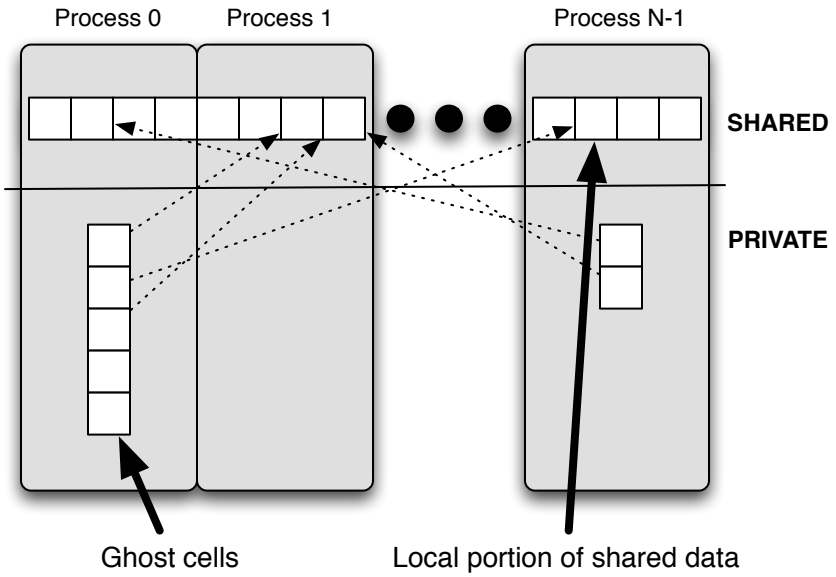


FIGURE 8.2. Logical organization of data for a Distributed Property Map.

The Distributed Property Map is implemented as an adaptor on top of the sequential property map library and consists of the following elements:

- An addressing scheme (i.e., a method of mapping keys to logical address spaces)
- Storage for local values
- Cached copies of remote values
- A method of copying values between logical address spaces
- Methods of resolving conflicting writes

The execution model for Distributed Property Maps is that of concurrent processes with separate logical address spaces. These processes may be mapped onto a variety of system-level parallel constructs, including threads sharing a single physical address space, however semantically the Distributed Property Maps interface treats these processes as if they have separate logical address spaces.

8.3.1. Addressing. In order to allow the underlying storage for a Distributed Property Map shared by all processes to be composed of disjoint pieces of memory from multiple address spaces, a method of mapping keys to address spaces must exist, we call this the Distribution of the Distributed Property Map. The mechanism utilized to perform the mapping is the Owner Map shared by both AM++ and the Parallel BGL 2.0 (see Chapter 4 § 4.1). For performance reasons the distribution is assumed to be static and simple to compute, though extensions to allow distributions that utilize dynamic addressing are possible. In the simplest case, the distribution is nothing more than a hash function. This distribution is assumed to be consistent across all processes and a process must be capable of resolving any key to an address space. In the active message implementation, this distribution is shared with the runtime.

8.3.2. Local Storage. Distributed Property Maps re-uses the sequential property map library to store local values for two reasons. First, this design allows any improvements or extensions to the sequential library to be included in Distributed Property Maps automatically. Second, this allows existing algorithms which use the sequential property map interface to operate directly on the local portion of the Distributed Property Map with no modifications.

8.3.3. Caching Remote Values. For values associated with remote keys, a Distributed Property Map maintains a set of ghost cells that are automatically created for any *put()* or *get()* request and store the most recent value known to this processor. The methods in which these ghost cells are stored and managed can be controlled by the application. The simplest scheme is unbounded growth of the ghost cell set which is stored using an associative container such as a *std::map*. More complex schemes involve caching strategies

of varying associativities combined with an upper limit on the size of the ghost cell set and eviction policies such as least-recently-used.

Figure 8.2 shows the logical organization of data for a Distributed Property Map. Data for keys local to a given process are stored in a memory region that is shared amongst all processes. Additionally, processes can maintain cached copies of remote data in their local memory region. While contiguous storage is shown, Distributed Property Maps places no restrictions on how local data or ghost cells are stored, the types of the underlying storage containers are template parameters. The global and local memory regions are logical constructs and do not imply that direct remote memory access is possible, the actual method by which remote data is manipulated is not specified and depends on the process group implementation.

8.3.4. Conflicting Write Resolution. Two types of potential data consistency issues exist in Distributed Property Maps. First, when a process accesses a remote key via *get()*, a message is generated to request the value associated with that key from the owning process. Because *get()* is a non-blocking call, it must return some value prior to the receipt of the current value as determined by the owning process. If a locally-cached copy of the requested value exists in the ghost cell set it will be returned, however even if no locally-cached copy of the requested value exists *some* value must be returned. While it is possible for a process to use the distribution to determine if a key is local or remote, there are some cases where a sensible default value can be specified and avoid the need to perform this determination. One example would be a shortest-paths computation on a graph where the distance to remote vertices can be specified to be infinity if a lower distance is not known. A sentinel value can also be returned by default to automate the process of determining whether a key is local or remote if such a value is available.

The second possible consistency issue occurs when multiple processes write to the same key during an epoch. Because there is no ordering of operations within epochs, some method of resolving multiple writes to the same key must be specified. This may consist of allowing an arbitrary write to succeed, or combining the writes using a user-specified

reduction operation such as *std::plus()*, *std::min()*, etc. The desired behavior depends on the semantics of the data being stored in a Distributed Property Map and can thus be specified separately for each Distributed Property Map instance.

8.4. Implementation

The features described are sufficient to implement basic PGAS semantics, i.e., the ability to read and write primitive data types in a Distributed Property Map distributed across multiple address spaces. The process group utilized for communication, addressing scheme (*GlobalMap*), local storage container (*StorageMap*), and method of caching remote values (*GhostCellStorage*) are provided as type parameters to the Distributed Property Map using the following class template:

```
1 template<typename GlobalMap,
2          typename StorageMap,
3          typename MessageGenerator,
4          typename GhostCellStorage>
5 class distributed_property_map
```

The version of Distributed Property Map in the original Parallel BGL used Process Group type parameter while the version shown here from the Parallel BGL 2.0 uses an AM++ Message Generator. In this class template the *GlobalMap* is a property map which is typically read-only and maps keys in a Distributed Property Map to the owning address space. This does not imply that each process must explicitly store the location of every element in the Distributed Property Map. The common case is that this property map is a wrapper around a statically computable distribution, however the interface allows any underlying implementation provided that each process can compute the location of every element in the Distributed Property Map. A variety of static distribution objects are supplied with Distributed Property Maps:

Uniform block: uniformly distributes contiguous blocks of keys across processes.

Uneven block: distributes contiguous blocks of keys across the processes with non-uniform block sizes.

Block Cyclic: distributes uniform blocks of keys across processes in a cyclic fashion.

Random Block: distributes uniform blocks of keys across processes in a random fashion.

```
1  // Create MPI environment and transport
2  amplusplus::environment env =
3    amplusplus::mpi_environment(argc, argv, true);
4  amplusplus::transport trans = env.create_transport();

6  amplusplus::rank_type pid = trans.rank();
7  amplusplus::rank_type P = trans.size();

9  // Property map storage for local keys
10 boost::shared_array_property_map
11   storage_map(10);

13 // Use default ghost cell storage,
14 // global_map maps keys to processes
15 distributed_property_map dpmap(trans, global_map,
16                          storage_map);

18 // Make sure that dpmap is initialized on all
19 // processes before remote operations occur
20 synchronize(dpmap);

22 // Shift the values in dpmap around a ring
23 for (size_t i = pid*10 ; i < (pid + 1)*10 % P*10 ;
24      i = i + 1 % P*10)
25   put(dpmap, i + 10 % P*10, get(dpmap, i));

27 // Wait for the above operations to complete
28 synchronize(dpmap);
```

FIGURE 8.3. Example Distributed Property Map usage.

Figure 8.3 shows an example of a program fragment utilizing a Distributed Property Map.

8.5. Extensions

In addition to reading and writing primitive data types, Distributed Property Maps has a number of extensions that allow more complex behavior. The simplest of these is an additional pair of interface functions. The first, *request()*, is merely a *get()* with no return value. This makes the common operation of requesting an up-to-date copy of a remote value more explicit. The second additional interface function, *cache()*, stores a value for a non-local key in the local ghost cell set but does not send this value to the owning process. This can be useful for reducing network communication by storing intermediate values locally and performing reductions on the sender where possible.

```
1  template <typename ValueType>
2  struct append_reducer {

4    // Return the default value of a remote key
5    template<typename Key>
6    ValueType operator()(const Key&)
7    { return ValueType(); }

9    // Combine two values x and y and return the result
10   template<typename Key>
11   ValueType operator()(const Key&,
12                        const ValueType& x,
13                        const ValueType& y)
14   {
15     ValueType z(x.begin(), x.end());
16     z.insert (z.end(), y.begin(), y.end());
17     return z;
18   }
19 };
```

FIGURE 8.4. Example of a reducer which appends to a container.

A more complex set of extensions to the basic PGAS semantics is contained in the conflicting write resolution functionality of Distributed Property Maps. This functionality is encapsulated in a run time assignable *reducer* function object. This reducer object takes two values, performs a user-defined reduction, and returns the result. When invoked with no arguments, it can also (optionally) return the default value for non-local keys. This

relatively simple functionality in combination with the fact that the values stored by a Distributed Property Map need not be primitive data types enables more complex functionality. Figure 8.4 shows an example of a reducer function object which appends values to a Distributed Property Map that stores a container supporting *insert()* (such as *std::vector*) for each key. One-sided accumulate operations such as those defined in the MPI 2 standard [**125**] are also easily implemented with custom reducers on singular data types.

The original Distributed Property Map provided the ability for remote *put()* operations can be buffered until the end of an epoch and explicitly received, or applied as they are received. In the latter case the Distributed Property Map uses a *trigger* interface in the process group to register callbacks which are used to handle remote updates as they arrive. This allowed for a form of ad-hoc active messages to be implemented. With the AM++ immediate processing of messages is the default behavior. The earlier approach of buffering communication until the end of an epoch can now be implemented explicitly by using two property maps; one of which holds data current as of the beginning of the epoch and a second to receive writes within the epoch. This approach may be more memory efficient if the number of writes is greater than the number of keys in the Distributed Property Map.

8.5.1. Extensible Semantics. Distributed Property Maps implements basic PGAS semantics at its core but, due to a layered, flexible, and extensible design, allows the implementation of more complex behavior as well. Some of the extensions described are merely syntactic sugar on top of basic PGAS semantics, while others add completely new functionality. The ability to map keys to arbitrarily complex data types with user-defined reduction operations allows the implementation of functionality beyond simple data movement. Because Distributed Property Maps is implemented as a generic library rather than in a compiler, syntactic extensions are straightforward and implementing new functionality is less complicated than modifying a compiler. This makes Distributed Property Maps an ideal framework in which to explore semantic additions to the PGAS model. Because Distributed Property Maps is a portable, user-level library, functionality implemented within

it can be leveraged on a variety of platforms without requiring a new compiler for each one.

In integrating Distributed Property Maps into the new design of the Parallel BGL 2.0 a number of changes were required to support additional design goals while maintaining the original functionality contained in the Distributed Property Map. In Parallel BGL 2.0, the Distributed Property Map plays two important roles:

(1) Providing a thread-safe local data model for active message-based algorithms.
(2) Supporting synchronous remote memory access for IO and other synchronous regions where a PGAS style is desirable.

The original Distributed Property Map fulfilled the second of these aims with little modification. Extensions described in Chapter 6 § 6.1.2 allowed it to simultaneously fill the first role.

The Distributed Property Map's ghost cell abstraction allows deterministic caching of non-local values to complement AM++'s size-limited reduction caches. Distributed Property Map is generic with regard to the container used to store ghost cells however, no thread-safe ghost cell implementation exists currently. This would be a straightforward extension.

8.6. Conclusion

The Thread-Safe Distributed Property Maps, an extension of the original Distributed Property Map, provides encapsulation of atomic and transactional access to vertex and edge properties in active message-based algorithms. In addition it provides PGAS remote memory access semantics for IO and other synchronous tasks. This combination of functionality allows the Parallel BGL 2.0 to use a single data model for vertex and edge properties across both access modes. Because Thread-Safe Distributed Property Maps are implemented using AM++ rather than compiled language extensions, application regions which use Distributed Property Maps could potentially benefit from the same sort of runtime optimization that allows regions using the Active Pebbles model to react to changes in system

state, resource availability, and input data characteristics within the limitations of the provided memory model. Because the Distributed Property Map is a DSEL embedded in C++ it is much simpler to extend than a compiled language. This fact in combination with the runtime optimization possibilities presented by Active Pebbles make the Distributed Property Map an excellent framework in which to experiment with further extensions to the PGAS model in a similar fashion to the asynchronous features discussed in Chapter 2.

9

Future Directions

In the preceding chapters we have demonstrated that graph applications possess fine-grained task parallelism that is poorly served by existing coarse-grained models of parallel computation. To address this shortcoming we have presented a method of phrasing graph algorithms using active messages as well as methods for converting algorithms expressed in this fashion to efficient implementations. This solution technique has proven to be effective for a number of graph algorithms because it is capable of leveraging fine-grained asynchrony to achieve parallel speedup. Furthermore, this abstraction can be applied effectively across multiple levels of parallelism, thus replacing "MPI + X" with a single unified programming model. In this chapter we discuss the extension of these techniques to

other applications in the graph domain, as well as methods for improving the techniques discussed in this thesis.

9.1. Dynamic Runtimes

Regular applications in which the structure of the computation does not depend on the structure of the input data rely on compile-time optimization for efficient code generation. In contrast graph applications are data-driven, meaning that their computational structure has significant dependencies on the input data (e.g., the graph itself). Active Pebbles allows for the deferral of optimizations that would be performed at compile-time in more traditional applications until runtime. The execution model features described in Chapter 4 comprise broad classes of optimization techniques. Each of these classes has a multitude of possible individual implementations. In addition, these implementations may be parameterized in a variety of fashions. Coalescing is parameterized on the size of the coalescing buffers used as are the caches used for message reductions; routing topologies may be fixed or vary dynamically at runtime. The execution model techniques present in Active Pebbles are designed to react to observations about the input data discovered at runtime. In the results presented in this thesis the parameters for these optimizations have been fixed when message types are created. In the context of the Parallel BGL 2.0 this means the time at which an algorithm class is instantiated, effectively fixing these parameters for the whole course of an algorithm's execution. However, this need not remain the case.

There is no fundamental reason that the parameters to these runtime optimizations may not vary from one message epoch to the next. In fact, preliminary investigation indicates that allowing these runtime parameters to vary dynamically makes possible implementations that out-perform any single static set of runtime parameters. Making the Active Pebbles runtime dynamic opens up a number of interesting possibilities:

(1) The ability to vary runtime parameters in order to minimize execution time based on the structure of input data. This would require some sort of runtime directed feedback or auto-tuning [**169**] framework capable of being executed at runtime.

(2) The ability to vary runtime parameters in response to variations in system state.
(3) The ability to vary the *implementation* of a given optimization from the execution model dynamically at runtime.

Here we examine a few of the many possible use cases in which a dynamic runtime may increase performance and reduce resource utilization. While these cases are described in isolation for illustrative purposes, they are strongly interrelated.

9.1.1. Optimizing Memory Utilization. Many data structures are asymptotically bounded by the number of edges or paths traversed. The number of messages generated in a single level of a breadth first search traversal (see Chapter 5 § 5.2 for algorithm details) depends on the number of edges traversed. The upper bound on the the number of messages is $\mathcal{O}(|E|)$, where $|E|$ is the number of edges in the graph. In practice only a subset of these edges will be traversed in a given breadth first search level. In Δ-stepping single-source shortest paths (Chapter 5 § 5.3) edges may be traversed multiple times if they lie on multiple candidate shortest paths. In this case the upper bound on the size of the bucket data structure used to store candidate paths depends on the distribution of edge weights, the δ parameter of the algorithm, and a possible super-linear factor involving $|E|$.

While the upper bound on memory utilization is large, in practice this upper bound is seldom reached. Predicting the actual memory utilization of graph algorithms is challenging as it depends on the structure of the input graph and often the vertex and edge properties as well. The consequences of main memory exhaustion can range from reduced performance to complete application failure in the absence of virtual memory. Even when virtual memory is present the random reads typical of graph applications may result in paging that reduces performance to the point that application completion is prolonged functionally indefinitely. Some of the runtime optimizations in the Active Pebbles execution model reduce memory utilization—software routing reduces the number of communication buffers required. In other cases, such as message reductions, local memory requirements are increased in an attempt to reduce communication.

While predicting memory utilization a priori is challenging, a dynamic runtime would be capable of varying runtime parameters in response to observations about system state in an attempt to prevent undesirable out-of-core behavior. Message reductions can provide significant reduction in communication, and thus runtime, however the caches utilized can become large. A dynamic runtime could increase cache size when memory is abundant and reduce or eliminate caches when free memory is at a premium. An additional level of introspection to monitor cache hit rate and react to it would also be possible. Flow control is another technique which can impact memory consumption across all the ranks involved in a computation. By increasing coalescing buffer size, communication can be made to occur less frequently. This trades increased memory consumption at the sender for reduced memory utilization at the receiver. If necessary the runtime could even block calling threads in the runtime in order to reduce the message flow at the sender, in effect throttling the computation itself in addition to the communication framework.

9.1.2. Dynamic Routing. Software routing reduces the number of communication buffers required at the expense of increases message latency. It also has a synergistic effect on a number of other Active Pebbles optimizations because it creates the opportunity for message coalescing and reductions to be applied at intermediate routing hops. A dynamic routing topology can adapt to both application characteristics as well as network state. Congestion avoidance **[92]** is a common network-level optimization. Given suitable instrumentation and feedback mechanisms this would be straightforward to integrate into a dynamic runtime.

As of this writing AM++ includes three primary routing topologies: direct (a complete graph), rook, and hypercube. These topologies add successively more intermediate links, increasing latency as well as bandwidth utilization. Direct routing uses $\mathcal{O}(P)$ communication buffers on each of the P processes while rook and hypercube use $\mathcal{O}(\sqrt{P})$ and $\mathcal{O}(1)$ buffers respectively. Graph applications experience different network characteristics over the course of their execution and vary from latency-bound to bandwidth-bound. In latency-bound regions it is of course desirable to minimize latency (e.g., by using a direct

routing topology) while in bandwidth bound regions higher-latency topologies that maximize bandwidth utilization may be desirable. A dynamic runtime would be capable of varying the routing topology to suit the anticipated communication characteristics of the next epoch. It is straightforward to observe that denser routing topologies impose greater memory requirements, and thus an inverse correlation between routing topology density and the size of message coalescing buffers is desirable. This illustrates on of many ways in which dynamic optimizations are often interrelated.

9.2. Dynamic Graphs

Any discussion of dynamic graphs must first address dynamic addressing as any data that is added must be able to be located by remote ranks. In both the original Parallel BGL and the Parallel BGL 2.0, the abstraction of the addressing component and the addressing of messages to members of distributed data structures rather than explicitly to ranks was done to provide maximum flexibility with regard to future data distributions. This mechanism enables dynamic distributions by allowing them to be encapsulated in the addressing component, specifically via the Owner Map concept. The software routing capabilities provided by AM++ and utilized by Parallel BGL 2.0 make this mechanism scalable in the presence of dynamic distributions. Rather than requiring every rank to store the address of every element of a distributed data structure, a process need only be able to locate the next routing hop on the way to the rank on which a target is stored. This may be weakened further to only require a rank be able to address those targets which will be referenced during the course of an algorithm's execution (for example, only structurally adjacent vertices and edges). This support for dynamic data distributions provide a suitable framework in which to implement dynamic graphs and a variety of algorithms which operate on them.

Fundamentally, a "dynamic" graph is a graph where vertices and edges may be added and removed at runtime. In practice this describes a wide range of possible implementations with regard to how and when the graph is modified and how algorithms interact with these modifications. Additionally, "dynamic" graphs may be separated into those

that support both insertion and deletion and "incremental" versions where the vertex and edge sets are monotonically increasing.

9.2.1. Separate modification and analysis. The simplest case is arguably not a dynamic graph at all, but does require similar data structure support. By separating graph computation into structural mutation and analysis phases the concurrency concerns can be avoided. In this mode algorithms from static graph cases can be reused without modification as from the standpoint of the algorithm, the graph is static. This use case requires graphs that can be updated efficiently, possibly in a thread-safe manner. Compact data structures such as compressed sparse matrix formats are efficient from a storage and access standpoint, but are expensive to update. For example, in a compressed sparse row format, edge lookup is $\mathcal{O}(\log d)$ where d is the edge degree of the source vertex of the edge in question. In the sparse or hyper-sparse random graphs this is effectively constant time. However, edge insertion and deletion is $\mathcal{O}(|E|)$. Less compact data structures such as adjacency lists allow $\mathcal{O}(\log d)$ lookup and $\mathcal{O}(d)$ insertion/deletion. Hybrid data structures with the bulk of the graph stored in a compact representation and the remainder stored in a less compact representation to support rapid insertion balance storage efficiency with insertion efficiency in the incremental case. In this latter case the less compact region must periodically be merged into the compact region.

9.2.2. Incremental modification with concurrent analysis. Coarse-grained updates which prevent analysis tasks from running while they are processed may not be suitable for all use-cases, especially latency-sensitive ones. Allowing graph modification and analysis tasks to execute concurrently presents a number of challenges. When concurrent updates are performed on multiple ranks, the updates may contain dependencies. If a vertex v and and edge $u \to v$ are added concurrently the runtime must either guarantee that the addition of v is processed before $u \to v$ or handle the reference to, the as of yet non-existent, v cleanly. A number of efficient distributed algorithms for ordering events in discreet event simulation exist **[20]** which may be suitable for ordering modifications. Additionally, a structural modification and any related vertex/edge property updates must

be performed transactionally to ensure algorithm correctness. Efficient support for these operations would be greatly simplified with transactional memory (hardware or software), though the Lock Map abstraction (Chapter 6) could also be modified to provide the required functionality. The most significant challenge would be in defining algorithm semantics. If an analysis task is initiated at time t_1 and completes at time t_2 it may observe any graph modifications performed in the interval (t_1, t_2). While the analysis task *may* observe any of these modifications, the order in which the analysis task accesses the graph in combination with the portions of the graph that are modified may result in any subset of the modifications being observed and reflected in the results of the analysis task. Additionally, the set of graph modifications observed may not be consistent over the course of the analysis task. An edge added at time $t_x : t_1 < t_x < t_2$ may be observed during phases of an analysis task executing after t_x, but not in earlier phases.

Two options are available when defining the semantics of analysis tasks in the presence of incremental modification. The first option is to observe graph modifications as they occur ordered only by the channel-ordering semantics in the network. This results in the analysis task having a view of the graph that may not correspond to a static view of the graph at any given time, making the solutions computed by analysis tasks "approximate". The degree to which these solutions correspond to the same analysis task executing on static views of the graph in the time range (t_1, t_2) depend on the data being computed by the analysis task and the frequency of graph modifications relative to the size of the graph at t_1.

The second option is to impose a logical order on all analysis tasks and graph modifications such that a "happens-before" relationship can be established between graph modifications and analysis tasks. In practice this could be implemented using any of a number of algorithms for ordering events in distributed systems [**107**]. In this model, the analysis task would have a timestamp that could be compared to the timestamp at which vertices and edges were added to the graph allowing it to filter out vertices and edges added after t_1 and observing the graph as it existed at t_1. A materialized view or snapshot of the graph

at t_1 would also be straightforward to construct [**38**], though the cost of creating such a snapshot would have to be balanced against the duration of its use.

9.2.3. Dynamic modification with concurrent analysis. "Fully" dynamic modification means that in addition to the insert operations allowed in the incremental case, vertices and edges may now be removed from the graph as well. The approaches which allow inconsistent views of the graph or utilize materialized views described above would work similarly in this case. However, if a consistent static view of the graph at some time is to be constructed the timestamp schema above must be extended to include a second timestamp marking the time of deletion. These two timestamps define a time range over which the vertex or edge exists logically. Vertices and edges must be retained until the runtime is able to determine that no analysis task will be issued with a timestamp prior to the deletion time of an element, at which point the element may actually be deleted.

The fine-grained, asynchronous runtime techniques used to express graph algorithms in the Parallel BGL 2.0 provide a suitable framework on which to implement extensions to support dynamic graphs. However, these extensions are decidedly non-trivial. At a minimum the following components would need to be added to support general purpose dynamic graph support:

- Fine-grained, efficient, transactional modification of graph data structures and associated properties.
- A distributed event coordinator for assigning logical timestamps to graph modifications and analysis tasks.
- A garbage collector for vertices and edges that have been removed and will no longer be referenced by analysis tasks.

9.2.4. Persistent algorithms. In the examples in the preceding sections the graph being analyzed was persistent. A use case where the analysis tasks were also persistent could be constructed as well. Maintaining a solution to an analysis task could be effected by "hooking" the process of graph mutation to update the solution based on the individual modification being performed. In the presence of frequent updates and frequent queries of

portions of the solution, this may prove more effective than periodically recomputing the solution from scratch. A simple example would be maintaining the set of shortest paths from a given vertex in a dynamic graph. Each edge addition would most likely require updating only a portion of the solution, though the actual number of distance updates would depend on the structure of the graph and distribution of edge weights. New vertices could be labeled when they were added by examining only their incident vertices and the weights of the edges connecting them. In an dynamic environment where many shortest path queries requiring low-latency were expected this incremental recomputation based on observing graph updates may provide better response times than periodic recalculation of the full shortest paths solution. Dynamic variants exist for a variety of algorithms [**51**, **69**, **80**].

9.3. Runtime-Managed Shared-Memory Parallelism

One of the primary goals of the Parallel BGL 2.0 was to provide support for fine-grained, node-level parallelism in addition to coarse-grained, distributed-memory parallelism. The first step in this process was to make fine-grained communication both thread-safe and efficient. AM++ provides fine-grained thread-safe messaging which meets the requirements of the Parallel BGL 2.0. This support is sufficient to allow threads originating external to AM++ and Parallel BGL 2.0 to execute graph algorithms concurrently. AM++ provides a mechanism to assigning thread ids to user threads and graph algorithm classes in Parallel BGL 2.0 provide thread barriers for synchronizing shared state, but users are still required to create and manage their own threads. This can be advantageous if users are utilizing these threads in other portions of an application because it avoids thread creation and other sources of overhead. However, if threads are created for the sole purpose of being utilized by the Parallel BGL 2.0, the interface could be simplified by having these threads created and managed internally to the library and invisible to the user.

Fine-grained parallelism in the Parallel BGL 2.0 is often initially expressed through loops of one sort or another. In completely message-driven algorithms such as the Parallel Search algorithm in Chapter 5 § 5.4 loops are absent, as are any auxiliary data structures

over which to employ data-parallelism. Techniques such as spawning a new thread for each message sent would succeed in utilizing idle cores, but add overhead which would more than offset the speedup gained through additional parallelism. Balancing overhead and resource utilization is essential to efficiently leveraging fine-grained parallelism in these sorts of algorithms. Fortunately this problem has been well studied and approaches such as thread pools and work stealing [**25**] provide excellent solutions. Integrating these constructs into Parallel BGL 2.0 and/or AM++ would allow the generation of fine-grained parallelism at runtime in a fashion that is transparent to library users. This sort of runtime-managed parallelism has been well received in other contexts (e.g., OpenMP [**46**]) and in some sense this use case is even simpler as the lack of spatial and temporal locality in graph algorithms precludes the need for customizations such as loop scheduling.

As demonstrated in this thesis, the active message abstraction is extremely flexible with regard to its execution context. Extensions to incorporate new parallel hardware within a compute node such as many-core accelerators [**141**]; FPGAs ; and, given appropriate memory models, GPUs is straightforward without modifying the abstractions utilized by the Parallel BGL 2.0. Extending the Parallel BGL 2.0 to leverage accelerators and manage fine-grained parallelism internally would reduce the effort required on the part of algorithm developers and potentially accelerate existing algorithms. Explicit user-supplied threading could remain as a fallback when more control is desired. Converting message handlers to forms suitable for accelerator-based execution would require significant effort in some cases.

9.4. Motivating Application: Low-latency Query Processing in Dynamic Graphs

The graph algorithms discussed in this thesis have largely consisted of coarse grained analysis of static data. Rather than producing results incrementally or accessing a subsection of the graph, these algorithms process the entire graph and produce one or more Distributed Property Maps of size $\mathcal{O}(|V|)$ or $\mathcal{O}(|E|)$ on completion as their result. The fine-grained, active message-based runtime solution to graph computation proposed in this

dissertation is suitable for a wide range of execution contexts and algorithms. One heretofore unexplored class of algorithms to which the Parallel BGL 2.0 is exceptionally well suited is answering low-latency queries on dynamic graphs.

Dynamic graphs can be used to model phenomena such as communication networks, financial transactions, or events in an intrusion detection system [**132**, **152**]. One common task in these domains is to detect anomalous phenomena. On-line anomaly detection is distinguished from traditional data mining by the fact that it is performed in response to external input (which can be modeled as graph modifications) with the intent of producing immediately-actionable data. Rather than off-line bulk analysis of large amounts of data, on-line analysis must consist of low-latency analysis of targeted, localized data to deliver actionable results in an interactive context. Previous approaches to this problem have largely utilized various machine learning approaches to construct Hidden Markov Models [**154**] or train classifiers [**36**]. This would seem to be due, in part, to the fact that the latency demands of the application preclude the use of coarse-grained, on-demand analysis.

The Parallel BGL 2.0 could provide for low-latency, event driven, *in situ* analysis of large scale dynamic graphs. Whether it is the most appropriate solution depends on the types of analysis being conducted. The Parallel BGL 2.0 is optimized for *traversal-based* algorithms of the sort described in Chapter 5. These algorithms expose task-parallelism which the asynchronous architecture of the Parallel BGL 2.0 can exploit. An example query for which the Parallel BGL 2.0 would be well-suited would be depth-limited exploration from a source vertex in order to compute an (ideally associative and commutative) function over the k-neighborhood of that vertex. The architecture of the Parallel BGL 2.0 encapsulates the responsibility for data consistency of the graph in the Thread-Safe Distributed Property Map and AM++ Transports allow partitioning of communication in an application. Utilizing these capabilities it would be straightforward to allow multiple concurrent queries in combination with dynamic graph modification.

10

Conclusion

The active message-based programming model described in this thesis is powerful because, as indicated by the title, it is a *spanning* model. Chapter 4 described a model through which graph applications expressed using active messages can span a range of parallel hardware, including both shared- and distributed-memory architectures, by separating specification from implementation. Runtime transformations allow optimization for a variety of objective functions including latency, bandwidth, resource utilization, etc. As demonstrated in Chapter 6, active messages are suitable for efficiently implementing a wide range of graph algorithms from the classification described in Chapter 5.

Phrasing graph algorithms as collections of asynchronous, concurrently executing, message-driven fragments of code allows for natural expression of algorithms, flexible implementations leveraging various forms of parallelism, and performance portability—all without modifying the algorithms themselves. Most importantly, active message-based implementations are capable of capturing the fine-grained task-parallelism present in many graph algorithms. Asynchronous execution of these fine-grained tasks hides communication latency and reduces global synchronization which yields implementations that out-perform coarse-grained, synchronous methods.

Active messages are an effective abstraction for expressing graph applications because they allow the fine-grained dependency structure of the computations to be expressed directly in a form that can be observed dynamically at runtime as it is discovered. At this point a variety of optimizations are available (coalescing, reductions, etc.) that are difficult or impossible to apply effectively at compile time; if applied manually, they would greatly complicate the structure of the algorithm's implementation. This expression provides users a natural and flexible method to express applications in terms of individual operations on vertices and edges rather than in terms of artificially coarser operations. Furthermore, algorithms expressed in this form can be mapped to a many hardware platforms and parallelized using a variety of techniques (processes, threads, accelerators, and combinations thereof). By allowing algorithms to be expressed in a form that places minimal constraints on the implementation, the active message expression of algorithms can be implemented in the manner that is most efficient for a given execution context. Separating the specification of graph algorithms from the details of their execution using active messages yields flexible and expressive semantics and high performance.

The Parallel BGL 2.0 demonstrates that active messages can be utilized to build a feature-rich framework for parallel graph computation. Active messages are used directly to implement fine-grained synchronization and one-sided operations with arbitrary semantics. This allows many algorithms to be entirely expressed in terms of active messages and their associated handlers. The Active Pebbles execution model decouples common

optimization techniques for improving communication performance from algorithm specifications. This decomposition allows optimizations to be reused and enables retroactive optimization of algorithms without the need to modify their specifications. This iteration of the Parallel BGL represents a major step forward by improving baseline performance through a reduction in global synchronization, adding support for hybrid parallelism, and exposing a generic software architecture suitable for experimentation with next-generation research topics such as dynamic graphs and dynamic runtimes.

Bibliography

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] Amazon Elastic Compute Cloud (EC2), 2010. `http://aws.amazon.com/ec2`.

[3] Apache Giraph, 2012. `http://giraph.apache.org`.

[4] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife machine: a large-scale distributed-memory multiprocessor. Technical Report TM-454, Massachusetts Institute of Technology, Cambridge, MA, USA, 1991.

[5] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.

[6] Gul Abdulnabi Agha. ACTORS: A model of concurrent computation in distributed systems. PhD thesis AITR-844, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Lab, June 1985.

[7] Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Improved external memory BFS implementations. In *SIAM Meeting on Algorithm Engineering and Experiments*, January 2007.

[8] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Symposium on Parallel Algorithms and Architectures*, pages 95–105, New York, NY, USA, 1995. ACM.

[9] George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Henry S. Warren, Jr. Dissecting Cyclops: a detailed analysis of a multithreaded architecture. *SIGARCH Computer Architecture News*, 31(1):26–38, March 2003.

[10] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, New York, NY, USA, 1990. ACM.

[11] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A standard template adaptive parallel C++ library. In *International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors*, page 10, July 2001.

[12] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's users' guide*, 1999.

[13] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[14] Matthew H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[15] D. A. Bader and K. Madduri. SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *International Parallel and Distributed Processing Symposium*, pages 1–12, April 2008.

[16] D.A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *International Conference on Parallel Processing*, pages 523–530, August 2006.

[17] David A. Bader, David R. Helman, and Joseph JáJá. Practical parallel algorithms for personalized communication and integer sorting. *Journal of Experimental Algorithmics*, 1, 1996.

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

[19] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGART Bulletins*, (64):55–59, August 1977.

[20] D. Ball and S. Hoyt. The adaptive Time-Warp concurrency control algorithm. In *SCS Multiconference on Distributed Simulation*, pages 174–177, 1990.

[21] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, October 1999.

[22] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972.

[23] Jonathan W. Berry, Bruce Hendrickson, Simon Kahanz, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *International Parallel and Distributed Processing Symposium*, Long Beach, CA, March 2007. IEEE.

[24] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30(8):207–216, 1995.

[25] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[26] Dan Bonachea. GASNet specification, v1.1. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2002.

[27] Boost. *Boost C++ Libraries*. `http://www.boost.org/`.

[28] Alex Breuer, Peter Gottschling, Douglas Gregor, and Andrew Lumsdaine. Effecting parallel graph eigensolvers through library composition. In *Performance Optimization for High-Level Languages and Libraries*, April 2006.

[29] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1994.

[30] Aydın Buluç. *Linear algebraic primitives for parallel computing on large graphs*. PhD thesis, University of California, Santa Barbara, 2010.

[31] Aydın Buluç and John R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4), November 2011.

[32] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, April 2004.

[33] D Chakrabarti, Y Zhan, and C Faloutsos. R-MAT: A recursive model for graph mining. In *International Conference on Data Mining*, pages 442–446, April 2004.

[34] Albert Chan and Frank Dehne. CGM*graph*/CGM*lib*: Implementing and testing CGM graph algorithms on PC clusters. In *PVM/MPI*, pages 117–125, 2003.

[35] Albert Chan and Frank Dehne. CGM*lib*: A library for coarse-grained parallel computing. `http://lib.cgmlab.org/`, 2004 December.

[36] Philip K. Chan, Wei Fan, Andreas L. Prodromidis, and Salvatore J. Stolfo. Distributed data mining in credit card fraud detection. *IEEE Intelligent Systems*, 14(6):67–74, November 1999.

[37] K. M. Chandy and Jayadev Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[38] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[39] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform

cluster computing. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[40] Daniel Chavarria-Miranda, Sriram Krishnamoorthy, and Abhinav Vishnu. Global Futures: A multithreaded execution model for Global Arrays-based applications. In *International Symposium on Cluster, Cloud and Grid Computing*, pages 393–401. IEEE Computer Society, 2012.

[41] UPC Consortium. Berkeley UPC system internals documentation, version 2.10.0, November 2009. `http://upc.lbl.gov/docs/system/index.html`.

[42] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17:51–64, 1970.

[43] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. Springer, 1998.

[44] David E. Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–12. ACM Press, 1993.

[45] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. Assessing fast network interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[46] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[47] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *International Symposium on Computer Architecture*, pages 189–196, New York, NY, USA, 1987. ACM.

[48] Miyuru Dayarathna, Charuwat Houngkaew, and Toyotaro Suzumura. Introducing ScaleGraph: an X10 library for billion scale graph analytics. In *X10 Workshop*, pages 6:1–6:9, New York, NY, USA, 2012. ACM.

[49] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[50] Steven J. Deitz, Sung-Eun Choi, and David Iten. Five powerful Chapel idioms, May 2010.

[51] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. *Handbook on Data Structures and Applications*. Computer and Information Science. Chapman and Hall/CRC, Boston, MA, USA, 2005.

[52] Denelcor, Inc., Aurora, CO. *HEP Fortran 77 User's Guide, Pub. no. 9000006*, 1982.

[53] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[54] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[55] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[56] Nathan E. Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. An initial implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[57] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, February 1990.

[58] D. Ediger, K. Jiang, J. Riedy, , and D.A. Bader. Massive streaming data analytics: A case study with clustering coefficients. In *Workshop on Multithreaded Architectures and Applications*, Atlanta, GA, April 2010.

[59] Nicholas Edmonds, Douglas Gregor, and Andrew Lumsdaine. Extensible PGAS semantics for C++. In *Fourth Conference on Partitioned Global Address Space Programming Model*, New York, New York, Oct 2010.

[60] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine. Single-source shortest paths with the parallel boost graph library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Piscataway, NJ, November 2006.

[61] Nick Edmonds, Torsten Hoefler, and Andrew Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *International Conference on High Performance Computing*, Goa, India, December 2010.

[62] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[63] Paul Erdős, Ronald L. Graham, and Endre Szemeredi. On sparse graphs with dense long paths. Technical Report CS-TR-75-504, Stanford University, Stanford, CA, USA, 1975.

[64] John Feo, David Harper, Simon Kahan, and Petr Konecny. Eldorado. In *Conference on Computing Frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.

[65] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, volume 1800 of *Lecture Notes in Computer Science*, pages 505–511. Springer, 2000.

[66] Steven Fortune and James Wyllie. Parallelism in random access machines. In *ACM Symposium on Theory of Computing*, pages 114–118, New York, NY, USA, 1978. ACM Press.

[67] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, 1980.

[68] D.P. Friedman and D.S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, 27(4):289–296, 1978.

[69] Daniele Frigionia, Alberto Marchetti-Spaccamelac, and Umberto Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251 – 281, 2000.

[70] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software–Practice and Experience*, 21(11):1129–1164, 1991.

[71] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[72] Rahul Garg and Yogish Sabharwal. Software routing and aggregation of messages to optimize the performance of HPCC RandomAccess benchmark. In *ACM/IEEE conference on Supercomputing*, pages 109–, 2006.

[73] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[74] Patrick Geoffray. Myrinet eXpress (MX): Is your interconnect smart? In *High Performance Computing and Grid in Asia Pacific Region*, pages 452–452, Washington, DC, USA, 2004. IEEE Computer Society.

[75] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. High performance graph algorithms from parallel sparse matrices. In *Applied Parallel Computing. State of the Art in Scientific Computing. 8th International Workshop.*, pages 260–269, 2007.

[76] John R. Gilbert, Viral B. Shah, and Steve Reinhardt. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering*, 10(2):20–25, 2008.

[77] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 423–437, October 2005.

[78] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Workshop on Parallel Object-Oriented Scientific Computing*, July 2005.

[79] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[80] Monika R. Henzinger and Valerie King. Maintaining minimum spanning forests in dynamic graphs. *SIAM Journal on Computing*, 31(2):364–374, February 2002.

[81] M. Herlihy. The Aleph Toolkit: Support for scalable distributed shared objects. In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Orlando, FL., January 1999.

[82] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programing Languages and Systems*, 15(5):745–770, 1993.

[83] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[84] Florian Hielscher and Peter Gottschling. ParGraph. `http://pargraph.sourceforge.net/`, 2004.

[85] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[86] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 159–168. ACM, January 2010.

[87] Eric Holk, William E. Byrd, Jeremiah Willcock, Torsten Hoefler, Arun Chauhan, and Andrew Lumsdaine. Kanor – A Declarative Language for Explicit Communication. In *Thirteenth International Symposium on Practical Aspects of Declarative Languages*, Austin, Texas, January 2011.

[88] IBM. *IBM System/360 Model 67 Functional Characteristics, Third Edition, GA27-2719-2*, 1972. `http://www.bitsavers.org/pdf/ibm/360/funcChar/GA27-2719-2_360-67_funcChar.pdf`.

[89] Intel. Threading Building Blocks. `http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm`, August 2006.

[90] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland, September 1998.

[91] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[92] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, pages 314–329, New York, NY, USA, 1988. ACM.

[93] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In Frank Pfennig and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 228–244. Springer Verlag, September 2003.

[94] Jithin Jose, Sreeram Potluri, Miao Luo, Sayantan Sur, and Dhabaleswar K. Panda. UPC Queues for scalable graph traversals: Design and evaluation on InfiniBand clusters. In *Conference on Partitioned Global Address Space Programming Models*, October 2011.

[95] R. Kaiabachev and B. Richards. Java-based DSM with object-level coherence protocol selection. In *International Conference on Parallel and Distributed Computing and Systems*, pages 648–653, Marina del Ray, CA, USA, November 2003.

[96] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. ParalleX. *International Conference on Parallel Processing Workshops*, pages 394–401, 2009.

[97] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. *SIGPLAN Notices*, 28(10):91–108, 1993.

[98] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.

[99] Prabhanjan Kambadur, Anshul Gupta, Amol Ghoting, Haim Avron, and Andrew Lumsdaine. PFunc: Modern task parallelism for modern high performance computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis, SC09*, Portland, Oregon, November 2009.

[100] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *International Conference on Data Mining*, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[101] Fredrik Berg Kjolstad and Marc Snir. Ghost cell pattern. In *Workshop on Parallel Programming Patterns*, pages 4:1–4:9, New York, NY, USA, 2010. ACM.

[102] Petr Konecny. Introducing the Cray XMT. In *Cray User's Group*, May 2007.

[103] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[104] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *SIGPLAN conference on Programming Language Design and Implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[105] S. Kumar, A.R. Mamidala, D.A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, Dong Chen, and B. Steinmacher-Burrow. PAMI: A parallel active message interface for the blue gene/q supercomputer. In *International Parallel Distributed Processing Symposium*, pages 763–773, 2012.

[106] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark E. Giampapa, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles J. Archer. The Deep Computing Messaging Framework: Generalized scalable message passing on the Blue Gene/P supercomputer. In *International Conference on Supercomputing*, pages 94–103, New York, NY, USA, 2008. ACM.

[107] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[108] Kevin Lang. Fixing two weaknesses of the spectral method. In *Neural Information Processing Systems*, 2005.

[109] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: a multithreading technique targeting multiprocessors and workstations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, New York, NY, USA, 1994. ACM.

[110] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[111] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *International Conference on Very Large Data Bases*, pages 133–144. VLDB Endowment, 2013.

[112] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 227–242, New York, NY, USA, 2009. ACM.

[113] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Knowledge discovery in data mining*, pages 177–187, New York, NY, USA, 2005. ACM.

[114] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 3721 of *Lecture Notes in Computer Science*, pages 133–145. Springer, 2005.

[115] Wen-Yew Liang, Chun ta King, and Feipei Lai. Adsmith: An efficient object-based distributed shared memory system on PVM. In *International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 173–179, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[116] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, April 2012.

[117] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[118] Piotr R Luszczek, David H Bailey, Jack J Dongarra, et al. The HPC Challenge (HPCC) benchmark suite. In *ACM/IEEE Conference on Supercomputing*, pages 213–, 2006.

[119] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Workshop on Algorithm Engineering and Experiments*, New Orleans, LA, January 2007.

[120] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of data*, pages 135–146, New York, NY, USA, 2010. ACM.

[121] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, September 1987.

[122] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Publishers, 2004.

[123] Message Passing Interface Forum. MPI, June 1995. `http://www.mpi-forum.org/`.

[124] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[125] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009.

[126] MPI Forum. MPI: A Message-Passing Interface Standard. v3.0, September 2012.

[127] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. In *Cray User's Group*, May 2010.

[128] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[129] Object Management Group. *CORBA 3.1*, January 2008. `http://www.omg.org/spec/CORBA/3.1/`.

[130] OpenMP Architecture Review Board. Openmp application program interface. Specification, 2011.

[131] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford Digital Library Technologies Project, November 1998.

[132] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, December 1999.

[133] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–25, New York, NY, USA, 2011. ACM.

[134] S.J. Plimpton, R. Brightwell, C. Vaughan, and K. Underwood. A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark. In *International Conference on Cluster Computing*, pages 1–7, 2006.

[135] Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *USENIX Conference on Operating Systems Design and Implementation*, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[136] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, 4(2):63–79, 1996.

[137] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library. In *Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, volume 1511 of *Lecture Notes in Computer Science*, pages 402–410, May 1998.

[138] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, June 1985.

[139] Arnold L. Rosenberg and Lenwood Scott Heath. *Graph Separators, with Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[140] L. R. Scott, T. Clark, and B. Bagheri. *Scientific Parallel Computing*. Princeton University Press, Princeton, NJ, USA, 2005.

[141] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):18:1–18:15, August 2008.

[142] C. Seshadhri, Tamara G. Kolda, and Ali Pinar. Community structure and scale-free collections of erdős-rényi graphs. *Physical Review E*, 85:056109, May 2012.

[143] G. Shah and C. Bender. Performance and experience with LAPI—a new high-performance communication library for the IBM RS/6000 SP. In *International Parallel Processing Symposium*, page 260, Washington, DC, USA, 1998. IEEE Computer Society.

[144] Viral Shah and John R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. In *IEEE International Conference on High Performance Computing*, pages 144–155. Springer, 2004.

[145] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Symposium on Principles of Database Systems*, pages 39–52, New York, NY, USA, 2002. ACM.

[146] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

[147] Yossi Shiloach and Uzi Vishkin. An $\mathcal{O}(log\ n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[148] Wei Shu and L. V. Kalé. Chare kernel—a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11(3):198–211, 1991.

[149] Silicon Graphics, Inc. *SGI Implementation of the Standard Template Library*, 2004. `http://www.sgi.com/tech/stl/`.

[150] Amitabh B. Sinha, L. V. Kalé, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, UIUC, 1993.

[151] Burton Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE Real Time Signal Processing IV*, pages 135–148, May 1981.

[152] Steven R Snapp, James Brentano, Gihan V Dias, Terrance L Goan, L Todd Heberlein, Che-Lin Ho, Karl N Levitt, Biswanath Mukherjee, Stephen E Smaha, Tim Grance, et al. DIDS (distributed intrusion detection system)-motivation, architecture, and an early prototype. In *National Computer Security Conference*, pages 167–176, 1991.

[153] Fengguang Song, Asim Yarkhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Conference on High Performance Computing Networking, Storage and Analysis (Supercomputing)*, Portland, OR, November 2009.

[154] Abhinav Srivastava, Amlan Kundu, Shamik Sural, and Arun Majumdar. Credit card fraud detection using Hidden Markov Model. *IEEE Transactions on Dependable and Secure Computing*, 5(1):37–48, 2008.

[155] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May 2010.

[156] Bjarne Stroustrup. *Design and Evolution of C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[157] Gabriel Tanase, Antal Buss, Adam Fidel, et al. The STAPL parallel container framework. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 235–246, New York, NY, USA, 2011.

[158] Nathan Thomas, Steven Saunders, Tim Smith, Gabriel Tanase, and Lawrence Rauchwerger. ARMI: A high level communication library for STAPL. *Parallel Processing Letters*, 16(02):261–280, 2006.

[159] R. Thurlow. *RPC: Remote Procedure Call Protocol Specification Version 2*. Sun Microsystems, May 2009. `http://tools.ietf.org/html/rfc5531`.

[160] Mathias Troyer and Prakash Dayal. The Iterative Eigensolver Template Library. `http://www.comp-phys.org:16080/software/ietl/`.

[161] Dean Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture*, pages 392–403, 1995.

[162] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23:31–42, 1976.

[163] UPC Consortium. *UPC Language Specification, v1.2*, May 2005. `http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf`.

[164] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[165] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, pages 256–266, New York, NY, USA, 1992. ACM.

[166] Jim Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6:5–7, 1998.

[167] Duncan Watts and Steven Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[168] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. *SIGARCH Computer Architecture News*, 17(3):273–280, April 1989.

[169] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(12):3 – 35, 2001.

[170] Kyle Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *Multithreaded Architectures and Applications*, 2008.

[171] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[172] Jeremiah Willcock, Torsten Hoefler, Nicholas Edmonds, and Andrew Lumsdaine. AM++: A generalized active message framework. In *Parallel Architectures and Compilation Techniques*, September 2010.

[173] Jeremiah Willcock, Torsten Hoefler, Nick Edmonds, and Andrew Lumsdaine. Active Pebbles: Parallel programming for data-driven applications. In *International Conference on Supercomputing*, Tucson, Arizona, May 2011.

[174] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing*, New York, NY, USA, 1998. ACM Press.

[175] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *ACM/IEEE Conference on Supercomputing*, page 25, Washington, DC, USA, 2005. IEEE Computer Society.

[176] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *USENIX conference on Hot topics in cloud computing*, Berkeley, CA, USA, 2010. USENIX Association.

Nicholas Edmonds

296 Tyrella Ave.
Mountain View, CA 94043

Tel: (812) 202-0698
ngedmond@cs.indiana.edu

Objective

To work on interesting HPC and/or scientific problems that span parallel programming, computer architecture, and language development. I prefer an innovative environment where attention is paid to current research as well as modern languages and software design.

Education

Indiana University 2003-Present
M.S. in Computer Science GPA: 3.97/4.00
Anticipated Ph.D. in Computer Science Fall 2013

Vanderbilt University 2000-2003
B.S. in Computer Science and Mathematics GPA: 3.60/4.00
Awards: *magna cum laude*, Dean's List, Clayton Kincaid Memorial Scholar, Sprint Foundation Scholar, National Merit Scholar

Experience

Google Mountain View, CA September 2013 - present
Software Engineer III
Worked on Common Abuse Team developing technologies to detect suspicious content and other undesirable behaviors on Google's public platforms.

Indiana University Bloomington, IN August 2004 - August 2013
Research Assistant in the Open Systems Lab (later renamed CREST)
Worked on three primary (interrelated) projects: **1)** the original Parallel Boost Graph Library (BGL), which offers distributed graphs and graph algorithms to exploit BSP parallelism on distributed memory machines while retaining the same interfaces as the (sequential) BGL. The Parallel BGL was incorporated into the Boost collection of software libraries in Q1 2009. **2)** Active Pebbles, a programming and execution model for message driven computation. **3)** The Parallel Boost Graph Library 2.0 which uses Active Pebbles principles and the AM++ active message library to transparently exploit coarse and fine-grained parallelism. Parallel BGL 2.0 uses message-driven computation and declarative parallelism to improve scalability and increase programmer productivity. Work is underway to extend to dynamic graph problems and implement a dynamically tuned runtime.

Google Mountain View, CA January 2008 - August 2008
Software Engineering Intern
Worked in the Google Application Performance Analysis portion of the Platforms team developing application benchmarks for BigTable. Setup a testing environment consisting of dedicated GFS and BigTable cells running on dedicated hardware. Developed a load generator capable of generating synthetic traffic based on profiles of production applications. Tested effects of Linux kernel resource containers on BigTable tablet servers and GFS chunkservers. Transitioned load generator code to BigTable SREs for use in performance testing as well as modeling the effects of adding new applications to existing BigTable cells by synthetically generating traffic prior to application rollout.

Intel Champaign, IL May 2006 - August 2006
Graduate Intern
Engineered developer tools for multi-threaded software development on next-generation multi-core processors. Worked on detecting high-level data-races in multi-threaded code using access interleaving invariants as part of the Thread Checker project. Also looked at methods to better exercise program scheduling in the testing and debugging stages of development through guided scheduling and other methods.

Sandia National Laboratory Livermore, CA May 2004 - March 2006
Graduate Intern
Developed MASS, a Modular Architecture for Sensor Systems in cooperation with two Sandia employees. MASS is both hardware and software extensible and allows developers to customize both the resources available on, and the power consumption of, their sensor nodes. MASS consists of a networking stack and set of user APIs for performing operations with and discovering information about modules in the system, which allows developers to have a fully functional sensor network application by writing only the application specific code. MASS is written in ANSI C and runs on a variety of microprocessors including the Cygnal C8051 and ARM-7. My contribution to the project included everything from device drivers to software architecture design and software development.

Indiana University Bloomington, IN January 2004 - May 2004
Assistant Instructor
Developed curriculum for graduate Introduction to Computer Security course laboratories. Wrote and presented lab activities on topics ranging from different types of attacks to network sniffing and spoofing and intrusion detection, among other topics. Taught four labs per week as well as performing typical associate instructor duties such as grading, etc.

Pervasive Technology Lab Bloomington, IN May 2003 - August 2003
Graduate Research Assistant
Assisted in setup of new Security for Ubiquitous Resources Groups Lab. Researched infrastructure and device considerations and assembled budget proposal.

HCA Healthcare Nashville, TN February 2002 - May 2003
IT&S Project Management Intern
Oversaw and coordinated the activities of a 2000 member IT/IS workforce. Coordinated operations of multiple Regional Data Centers spread across three countries. Interacted with projects at every level from project management to implementation. Served in an advisory role on $160m replacement of accounts receivable system. Member of team developing Project Management Center to train Project Managers in order to allow more efficient corporate operations through better resource planning and allocation.

Special Skills

Programming Languages : C++, C, Python, (No)SQL, Ruby, Scheme, LISP

Tools and Abilities : Hybrid parallel programming (e.g., threads + MPI), OpenMP and Cray loop-directives based parallelism, Generic C++ (contributor to Boost), OpenGL Development, Unix Networking, POSIX Threads, MPI, Distributed Systems Development, Embedded Systems Development including JTAG debugging and Device Driver development

Platforms : Cray XMT and XT5, IBM BG/P, various flavors of x86 clusters connected via Myrinet and Infiniband, Sun Niagara, Embedded systems (C8051, TI MSP430), MacOS clusters, and a variety of single-user workstations

Selected Publications

Expressing Graph Algorithms Using Generalized Active Messages. Nick Edmonds, Jeremiah Willcock, and Andrew Lumsdaine. International Conference on Supercomputing. Eugene, OR. June 2013.

Expressing Graph Algorithms Using Generalized Active Messages. Nick Edmonds, Jeremiah Willcock, and Andrew Lumsdaine. Principles and Practice of Parallel Programming. Shenzhen, China. February 2013. Poster.

Active Pebbles: Parallel Programming for Data-Driven Applications. Jeremiah Willcock, Torsten Hoefler, Nick Edmonds, and Andrew Lumsdaine. International Conference on Supercomputing. Tuscon, AZ, June 2011.

Active Pebbles: A Programming Model For Highly Parallel Fine-Grained Data-Driven Computations. Jeremiah Willcock, Torsten Hoefler, Nick Edmonds, and Andrew Lumsdaine. Principles and Practice of Parallel Programming. San Antonio, TX, February 2011. Poster.

Scalable Parallel Solution Techniques for Data-Intensive Problems in Distributed Memory. Nicholas Edmonds and Andrew Lumsdaine. SIAM Workshop on Combinatorial Scientific Computing. Darmstadt, Germany. May 2011.

Design of a Large-Scale Hybrid-Parallel Graph Library. Nicholas Edmonds, Jeremiah Willcock, Torsten Hoefler, and Andrew Lumsdaine. International Conference on High Performance Computing, Student Research Symposium. Goa, India. December 2010.

Extensible PGAS Semantics for C++. Nicholas Edmonds, Douglas Gregor, and Andrew Lumsdaine. Conference on Partitioned Global Address Space Programming Model. New York, New York. October 2010.

AM++: *A Generalized Active Message Framework.* Jeremiah Willcock, Torsten Hoefler, Nicholas Edmonds, and Andrew Lumsdaine. Parallel Architectures and Compilation Techniques. Vienna, Austria. September, 2010.

A Space-Efficient Parallel Algorithm for Computing Betweenness Centrality in Distributed Memory. Nick Edmonds, Torsten Hoefler, and Andrew Lumsdaine. International Conference on High Performance Computing. Goa, India. December, 2010.

Single-Source Shortest Paths with the Parallel Boost Graph Library. Nick Edmonds, Alex Breuer, Douglas Gregor, Andrew Lumsdaine. The Ninth DIMACS Implementation Challenge Workshop. Piscataway, NJ. November, 2006.

Method for Module Interaction in a Modular Architecture for Sensor Systems (MASS). Jesse Davis, Doug Stark, and Nicholas Edmonds. In Proceedings International Embedded and Hybrid Systems Conference (IEHSC). Singapore. May 2005.

MASS: Modular Architecture for Sensor Systems. Nicholas Edmonds, Doug Stark, and Jesse Davis. In Proceedings 4th International Workshop on Information Processing in Sensor Networks (IPSN). Los Angeles, California. April, 2005.

Software Application for Modular Sensor Network Node. Jesse Davis, Doug Stark, and Nicholas Edmonds. U.S. Patent Application No. 10/970,684. October 20, 2004.

Grants and Awards

IEEE Technical Committee on Parallel Processing travel grant to attend the International Conference on High Performance Computing (HiPC). Goa, India. December 2010.

IEEE Technical Committee on Parallel Processing *Best Presentation Award* for *Design of a Large-Scale Hybrid-Parallel Graph Library* in the International Conference on High Performance Computing (HiPC) Student Research Symposium. Goa, India. Dec. 2010.

Best Student Poster Award for *Active Pebbles: A Programming Model For Highly Parallel Fine-Grained Data-Driven Computations* in the 16^{th} ACM SIGPLAN Annual Symposium on Principles and Practices of Parallel Programming. San Antonio, TX. February 2011.

Invited Talks

Version 2.0 of the Parallel Boost Graph Library: Message-driven solutions to data-driven problems. Nicholas Edmonds. SIAM Annual Meeting. Minneapolis, MN. July 2012.

The Parallel Boost Graph Library `spawn`*(Active Pebbles).* Nicholas Edmonds. KDT Mind Meld. Santa Barbara, CA. March 2012.