AN ABSTRACT OF THE DISSERTATION OF

William H. Dillon for the degree of Doctor of Philosophy in Computer Science
presented on May 29, 2012
Title: Distributed OpenCL: A platform for Distributed, Heterogeneous Computing for Domain Scientists

Abstract approved: ____

Michael J. Bailey

It is possible to purchase, for as little as $10,000, a cluster of computers with the capability to rival the supercomputers of only a few years ago. Now, users that have little to no experience developing distributed applications or managing a cluster are in a position to do so. To allow domain scientists to effectively utilize these resources, Distributed OpenCL (DOCL) was developed. DOCL is an easy-to-use foundation for peer-to-peer distributed computation on small to medium clusters. It is assumed that the end-user is a domain scientist, familiar with model development in environments such as Matlab, though inexperienced with distributed computation or parallel programming. The scope of this work includes the definition of a peer-to-peer protocol for discovering and establishing relationships with every node within a multicast domain, using the concepts of Zero-Configuration Networking, multicast DNS, and DNS Service Discovery. A problematic edge case of multicast DNS is detailed along with a mitigation technique. An XML schema is also described for basic peer communication and cluster management and inventory. A system for scheduling algorithm tasks on the cluster of heterogeneous compute devices was developed, including an automatic computation and communication cost measurement system. Finally, a graphical programming language was designed and implemented that allows non-expert programmers and modelers to develop new applications in a straightforward, accessible way.

Distributed OpenCL: A platform for Distributed, Heterogeneous Computing for Domain Scientists

by

William H. Dillon

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented May 29, 2012
Commencement June 2012

Doctor of Philosophy dissertation of William H. Dillon on May 29, 2012.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

William H. Dillon, Author

ACKNOWLEDGMENTS

I would like to first thank my wife Katie for being a patient and supportive partner. Without her support, encouragement, and love this work would have never been completed. In addition, I must thank my mother for the many years of support and love and the rest of my family for being there for me. Finally, I want to thank Rachel, my sister, for graciously lending her editing skills.

Secondly, I would like thank my advisor, Mike Bailey, for the support and encouragement over the duration of my graduate career. I would also like to recognize all of my committee members for their time and effort.

Thirdly, I would like to thank the fantastic research computing team at CEOAS. The help provided by Chuck Sears, Tom Leach, and Bruce Marler, was indispensable. Toms mastery of networking protocols is rivaled only by the implementers themselves. The unique approach to research computing management fostered an environment that allowed this work to take place. Had the team not been flexible and understanding, it would not have been possible to complete this work.

Finally, I want to thank Mark Abbott for supporting this work, and providing an environment in which it could be completed, and for financial support, Terri Paluszkiewicz and the Office of Naval Research.

TABLE OF CONTENTS

Page

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

Page

LIST OF FIGURES (Continued)

Figure Page

LIST OF TABLES

Distributed OpenCL: A platform for Distributed, Heterogeneous Computing for Domain Scientists

Introduction

There is a growing divide between the capabilities of modern computing devices and our ability to program them. Gabe Newell of Valve Software recently said "If there were 500 people who could write a good game engine in the last generation, you're really talking 50 people who are going to be good enough to do it in the next generation." (Francis 2010) This is an important observation, but not just within the context of professional software and game development. Scientific applications are, if anything, more susceptible to falling behind the technology curve. Budgets being allocated to maintaining existing models are limited, and there are few opportunities to develop new models that are optimized for recently developed hardware. In addition to limited budgets, talented programmers are enticed away from scientific computing to industries with the higher salaries, such as gaming and financial analytics. To advance scientific computing, solutions that address these realities must be developed. I've responded by developing Distributed OpenCL; a platform that enables scientific application development using commodity and gaming hardware.

Advances in the computer gaming industry are increasingly relevant to any discussion of general and scientific computing. In the 1980s and 1990s, the computing industry was largely focused on providing high-quality tools for professionals, and it was during this time that the majority of supercomputer research and development took place. World governments viewed supercomputer performance as an economic engine as well as an important area of intergovernmental competition. The industry enjoyed the support of considerable government procurements, including national centers and installations for classified work in agencies such as the Department of Energy (DOE) and the Defense Advanced Research Projects Agency (DARPA) (Gilliam 1993).

Although the government remains a significant factor in the computing industry, its ability to influence the direction of computing research and development has been overwhelmed by the expansion of the consumer mobile and entertainment market segments. The market intelligence firm IDC values the 2011 High Performance Computing (HPC) market segment at $10 billion (Joseph and Shirer 2012); it estimates the global value of handheld gaming at $14.7 billion (Ward and Shirer 2012) and the smartphone market at $157.8 billion in the same time period (Llamas, Restivo, and Shirer 2012).

The expansion of the gaming console and smartphone markets pushed the computing industry to develop devices that are appropriate for these markets. Smartphone processors are designed to maximize performance within a very tight power budget. Gaming consoles tolerate greater power draw, but achieving maximum performance is vital. In both cases, cost is a major factor and vendors must provide inexpensive solutions.

Semiconductor products enjoy economies of scale, meaning that the cost to produce an additional unit is less than the average cost to produce all prior units. The majority of the costs associated with a semiconductor product are one-time sunk costs such as research and development, "tape out[1]," and tooling. In consumer markets, these costs are distributed across a large user base. Targeting the consumer market is often a wise business decision; expensive and exotic products, targeted at high-performance computing, are becoming less common.

Healthy competition among vendors drives costs down while improving performance. Microprocessor architecture licensing firms such as MIPS

[1] Tape out is the term used to describe the process of producing the photo-lithographical masks used to define the patterns on a semiconductor wafer during production.

Technologies[2] and ARM[3] produce standard processor designs and Instruction Set Architectures (ISAs). These companies do not manufacture physical products themselves; instead, they license their designs to independent firms. The products based on MIPS and ARM architectures are often compatible within their families, allowing them to be treated as commodities. Using commodity products in scientific applications ensures that scientists are able to pay the lowest possible price for a given level of functionality, making the most out of their fixed budgets.

In addition to the changes in market conditions, we are at a unique time in the evolution of silicon technology. In terms of clock speed, Moore's Law has broken down. During the period of exponential clock speed improvement, software development could remain stagnant; now that clock speeds are mostly constant, improvements in performance must come from increased parallelism. Existing software packages, especially those that are single-threaded, will no longer improve with new hardware. To keep pace with these changes, new software must be developed, and new programming techniques are needed.

Market Changes

The consumerization and commoditization of technology began in the early 1980s, when Compaq reverse-engineered the IBM BIOS and produced the first "IBM Compatible" computer. Competition among vendors producing interchangeable products drove prices down, increasing the number of people who could afford home computers. Home computers were used for entertainment purposes, including computer games, which exploded in popularity with the advent of the personal computer.

[2] MIPS Technologies: http://www.mips.com/, accessed May 12, 2012

[3] ARM Ltd.: http://www.arm.com/, accessed May 12, 2012

By the mid-1990s, computer gaming had become so popular and sophisticated that dedicated 3D graphics processors were developed for gaming. Companies such as 3dfx and Nvidia were founded with talent originating from scientific and enterprise computing companies such as SGI, LSI Logic, Sun Microsystems and AMD. The new 3D graphics processors were the perfect combination of price and performance. Features that were only useful for scientific applications, such as numerical precision, were sacrificed to keep costs low.

Intense competition among the early graphics card companies accelerated product development. Companies were able to increase performance by moving more of the graphics pipeline into hardware. The Transform and Lighting (T&E) engine from the Nvidia GeForce 256 is a good example of this. The T&E engine performed all of the linear algebra operations as well as basic fragment shading in hardware. According to Nvidia[4], this product was the first Graphics Processing Unit (GPU). Two years later, Nvidia added programmability in the GeForce3 product. The programs, called "shaders," were small, extremely constrained programs that could modify the way pixels were computed. In time, shaders were added for vertices, geometry, and tessellation. As the rendering pipeline became more diverse, Nvidia unified the hardware architecture of its processors, discarding the dedicated processing for vertices, primitive assembly, rasterization, and pixels. The new architecture, called Common Unified Device Architecture (CUDA) (Lindholm et al. 2008), uses general-purpose compute elements that are dynamically scheduled to perform any graphics task. In 2006, Nvidia released the CUDA programming language that allowed non-graphics applications to take advantage of the parallel processing power of the GPU.

By generalizing the architecture and programming model, GPUs have become powerful co-processors. Even before general-purpose programmability was in place, the movement toward utilizing the power of GPUs in non-graphics tasks began under

[4] Nvidia corporate history: http://www.nvidia.com/page/corporate_timeline.html, accessed May 12, 2012

the General Purpose GPU (GPGPU) banner. Researchers discovered ways to perform tasks such as large matrix solvers (Bolz et al. 2003), Fourier transforms (Moreland and Angel 2003), and fluid dynamics (Harris 2003) on GPUs by massaging the algorithms to appear as graphics tasks. Technologies such as CUDA, and later OpenCL, allowed algorithms not easily described as graphics tasks to take advantage of GPUs.

The movement of the computer into the home inspired technologies that are now used in scientific applications. Had commodity graphics hardware not been invented, it's hard to imagine that processor architectures inspired by GPUs would have been invented.

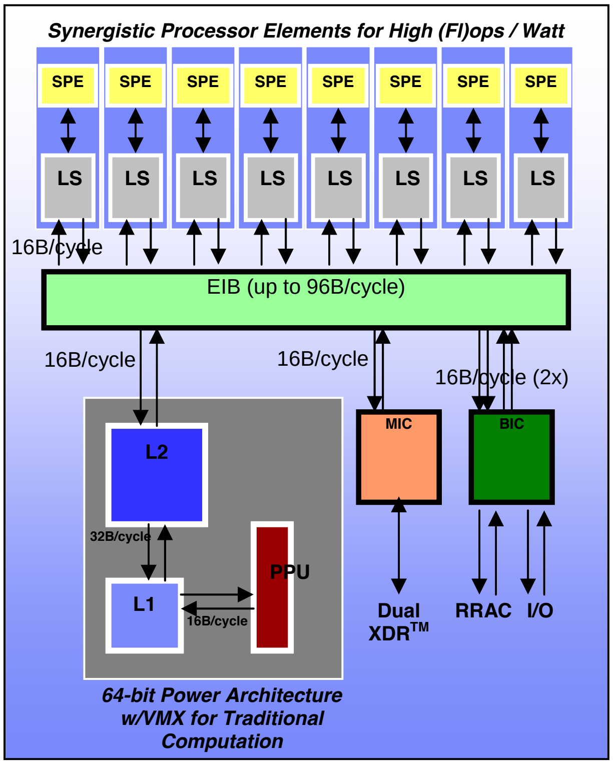


Figure 1: Cell/BE architecture (Chow, Fossum, and Brokenshire 2005)

Coincident with the development of the GPUs, gaming consoles were experiencing great growth in popularity and performance. Strong competition among consoles encouraged rapid development of innovative technologies. In anticipation of its next console, Sony teamed with Toshiba and IBM to found the STI Alliance.

Tasked with developing a “supercomputer on a chip,” they invented the Cell Broadband Engine (Cell/B.E., Figure 1) (Buttari et al. 2007). The Cell/B.E. was a compromise between the parallel processing throughput of a GPU and the single-threaded performance of a CPU. The Cell/B.E. broke ranks with the powerful processors optimized for single thread performance that were common at the time. The new design took steps backward from the traditional tools used to improve instruction-level parallelism, such as out-of-order execution (OOE).

The Cell/B.E. is an Asymmetric MultiProcessor (AMP), meaning that the individual processors are not identical to one another (in contrast to the much more common Symmetric MultiProcessor (SMP)). The POWER Processing Element (PPE) is substantially similar to a PowerPC 970 CPU with the OOE logic removed. In addition to the PPE, several Synergistic Processing Elements (SPE) are married with an Element Interconnection Bus (EIB). The SPEs are designed to be highly efficient, vectorized, throughput-optimized processors.

On an SMP system, the operating system can run itself or any other process on any of the processors in the system, because they’re all identical. However, on the Cell/B.E., the SPEs are architecturally distinct from the PPE and cannot run kernel code. The SPEs can only run specialized code and are scheduled by an application rather than the kernel. Processor features that are necessary for running general-purpose code are expensive in terms of power, die area, and complexity. Discarding these features allowed the Cell/B.E. to achieve dramatically higher throughput than other processors available at the time. The challenge presented by this design, however, was the significant increase in complexity presented to the programmer.

The designers of the Cell/B.E. were ahead of their time in the sense that many of their design choices were used in many subsequent processor designs. The Cell/B.E. was the first in what became a shift in the strategy employed to improve processor performance.

The end of clock rate increases

Processors are physical devices; their capabilities and limitations are ultimately dictated by the material processes used to create them. The dimensions of these physical constraints have been explored since the beginning of semiconductor use in electronics.

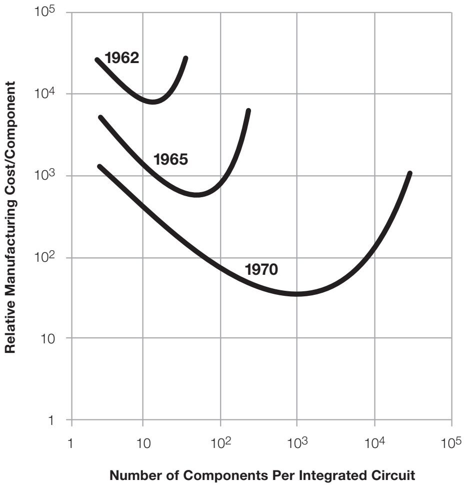


Figure 2: Semiconductor manufacturing trend from 1962 to 1970. (G. E. Moore 1965)

In 1965, Gordon Moore wrote a paper that identified a trend in the semiconductor industry. A widely interpreted quote from that paper is: “The complexity for minimum component costs has [increased] at a rate of roughly a factor of two per year.” (G. E. Moore 1965) His paper included a graph that has been reproduced in Figure 2. The quote refers to the minimum point on each of the relative manufacturing cost curves. There is a range of cost/complexity for silicon devices, and the most efficient among them doubles in complexity approximately every year. Moore assumed that the trend would continue for at least 10 years.

Since the original paper was published, Moore has revisited the relationship that has become known as Moore’s Law several times. In 1975, he saw that integrated circuits had become optimal in terms of area utilization, and he reduced the slope of the curve to a doubling every two years (G. Moore 1975). Later, in 1995, he was unwilling to look to the future past 0.18 micron (10^{-6} meters) technology. The lithography engineers at Intel couldn't conceive of working at this feature size with the techniques available at that time (G. E. Moore 1995).

One interpretation of Moore's Law, which he later endorsed, was that processor speed would roughly double every two years. The two quantities are related; as features become smaller, capacitance and propagation delay decrease. Those properties are the primary factors that determine the maximum clock rate of a device.

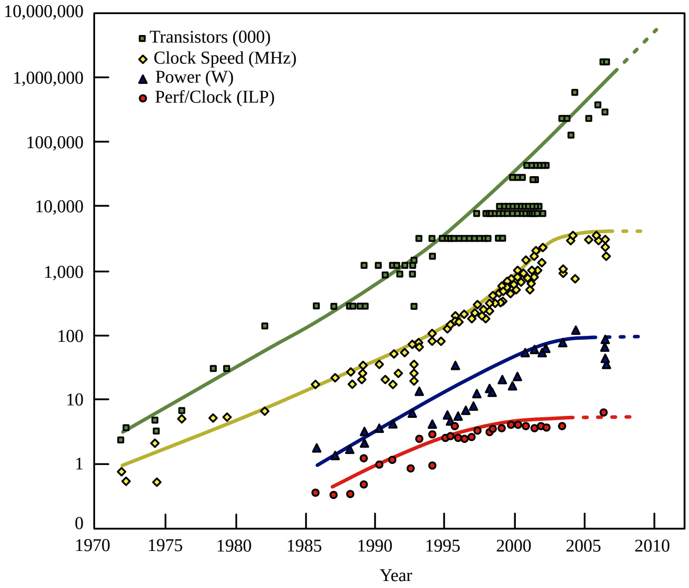


Figure 3: Historical clock speed and transistor count (Source: Shalf et al. 2009)

Once personal computers had standardized on the x86 processor architecture vendors transitioned to using clock rate as a competitive metric. Intel was particularly aggressive, and set high goals for clock rate scaling. Intel strategically architected its processors to achieve higher clock rates than its competitors' processors. If the architecture and feature size are fixed, it is possible to increase clock rate by reducing the duration of the work completed per clock interval. Increasing the number of stages in a pipeline means each stage requires less work and clock rate can increase. Because it isn't possible to know the result of a logical branch condition before its operands have been computed, it’s necessary to predict the outcome. When a branch is mispredicted, all of the speculative work must be discarded. This strategy was tested to its extreme limit by the Intel Netburst architecture, which was used in the Pentium 4 line. The yellow line in Figure 3, clock speed in KHz, shows a noticeable bump between 1999 and 2005, caused by the Pentium 4 and Netburst. The cost of branch mispredictions undermined any increases in performance that could have been gained; the Pentium 4 was noticeably slower than its rivals. Since 2005, clock rates have remained nearly constant. The fundamental limits that constrain clock rate -- power and performance per clock (a proxy for instruction level parallelism) -- prevent further advances in single-thread performance.

In the future, the most reliable way to improve performance will be to increase the number of processors in a system. This shift forces developers to change the way they think about improving the capabilities of computer systems. It was once common wisdom that we could continue using an existing application and expect a doubling of its speed every 18 months. Now, it will take considerably longer to yield similar results without modifying the application or, in extreme cases, re-architecting it from the ground up. It is imperative that we develop applications that are able to take advantage of the proliferation of processors in a computer while tolerating lagging clock rates.

Parallelism is the path forward

Processor vendors have had to embrace other methods for increasing processor performance year after year. Without constant improvements in processor performance, there would be little reason to purchase new products. Innovative architectures and increasing parallelism have become the primary means for increasing performance and driving sales. The continued advance of process technology has enabled greater logic density, allowing for more processors in the same space.

Process technology has advanced beyond the concerns held by Moore and Intel's engineers in 1995. Current technology (as of this writing) is capable of producing chips with 22 nm (.022 micron) features. The current best estimate for the absolute scaling limit for traditional semiconductor techniques is 5 nm. It is estimated that we will reach this limit some time after 2020 (Figure 4). After this point, significant modifications in process technology, such as silicon nanowires and carbon nanotubes[5], will be necessary to continue to improve semiconductor density.

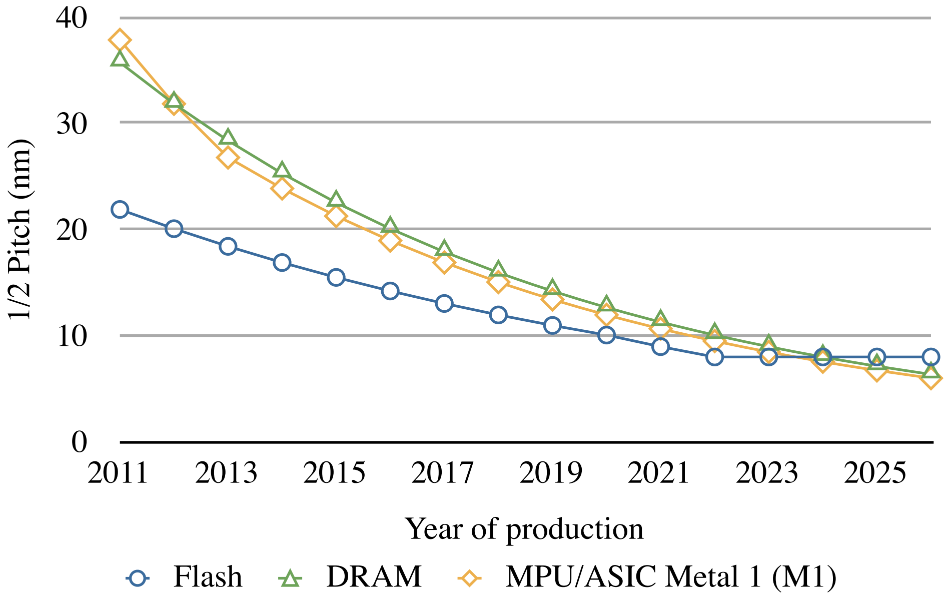


Figure 4: Semiconductor feature size roadmap (Source: ITRS 2011)

In anticipation of the end of feature size miniaturization, and responding to the needs of the mobile device industry, manufacturers have explored other means for increasing logic density. Chip stacking techniques are now a common practice in highly integrated system-on-a-chip (SoC) solutions. The success of these methods is evident in products such as the Apple iPhone.

The Apple A4 processor, which is an ARM derivative, contains the application processor and Synchronous Dynamic Random Access Memory (SDRAM) in one

[5] International Technology Roadmap for Semiconductors, ITRS 2011 report, http://www.itrs.net/Links/2011ITRS/Home2011.htm, accessed May 15, 2012

package. The cross-section image presented in Figure 5 shows the construction of a typical chip-stack system-in-package device. The image was created by Chipworks, in association with iFixit.com. Shortly after the release of the Apple iPad, iFixit.com obtained and disassembled a unit, then sent the mainboard to the Chipworks facility. There, Chipworks cut the A4 in half and ground it smooth.

The photograph is labeled to highlight a few items of interest. The first, label (*a*), is the integrated SDRAM connected to a substrate through several bond wires, one of which is partially visible (*b*). The application processor (*c*) is a flip-chip package mounted to its substrate with solder balls (not labeled). The SDRAM subassembly is electrically and physically connected to the application processor using solder balls (*d*). Finally, the entire package is mounted onto the PCB using a ball-grid array (*e*). The A4 is not an abnormal, or overly advanced, package. The chip stacking approach has become common, and there are several techniques that enhance the level of integration of these system-in-package devices (Bansal et al. 2010).

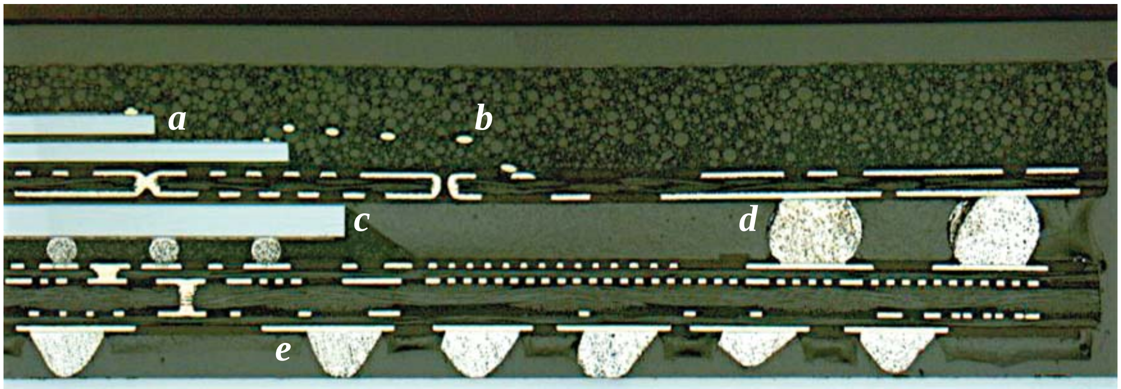


Figure 5: Cross-section of the Apple A4[6]

The widening gap between domain science and computer technology

With the advancement of easy-to-use numerical modeling tools such as Matlab, R and Mathematica, it has become easier for domain scientists to describe their ideas

[6] Ifixit.com Apple A4 teardown, http://www.ifixit.com/Teardown/Apple-A4-Teardown/2204/1, accessed May 15, 2012

and models in computer-readable form. This opened the door for an expansion in the number and variety of computer models and advanced the frontiers of science. Though these tools simplify the creation of scientific models, they do little to improve the complexity of parallel programming; they are explicitly serial. It is possible to develop parallel, or even distributed, applications with these tools, but it is no easier than using a language such as Fortran or C.

Embracing parallelism is the best way to continue to improve performance over time. Existing applications, especially those that are not multithreaded, are not yielding the incremental increases in performance they once were. However, with effectively utilized parallelism and GPU technologies, research that was formerly impractical is now possible. For less than $10,000, it is possible to purchase a computer that would rival the purpose-built supercomputers of only a few years ago (Van der Maar and Batenburg 2009). The power is available, and affordable, but is only useful to those who can harness it.

A common reason for the lack of adoption of parallel and distributed computing is the lack of appropriate training options. This knowledge is often passed between individuals in a workgroup, and sometimes between workgroups. The techniques of parallel and distributed programming can become a type of folk knowledge. The formal training available is, in large part, targeted toward professional programmers and computer science students. Students are expected to be familiar with basic networking concepts and UNIX system administration and have experience programming in C. Not only is the typical user unlikely to possess the skills or prerequisites for these courses, they are also likely to gain little from them. Their goal is not an exploration of the depth of parallel and distributed computing, but to explore the breadth of the field and learn practical ways they can benefit from its adoption.

Distributed OpenCL Overview

Distributed OpenCL (DOCL), the product of this work, was designed to address the issues facing scientific computing today. It is intended to bridge the gap between domain science and computer technology. Future computing devices will be more diverse, and CPUs and GPUs of a wide variety of architectures are already common. As single-thread performance no longer increases at its previous rate, the use of parallel programming is now essential. Though options exist for utilizing these resources with current technology, they are generally not accessible to domain scientists.

To achieve the greatest impact, it is important to re-imagine what an effective programming environment is. It must be capable of producing applications that can work on a variety of new commodity architectures, and even across ad hoc clusters. Distributed OpenCL is a model and platform that allows domain scientists to leverage the advanced computer architectures that are now commonplace, from smartphone processors to high-end GPUs. It was designed from the ground up to allow the creation, management, and utilization of ad hoc clusters of commodity products.

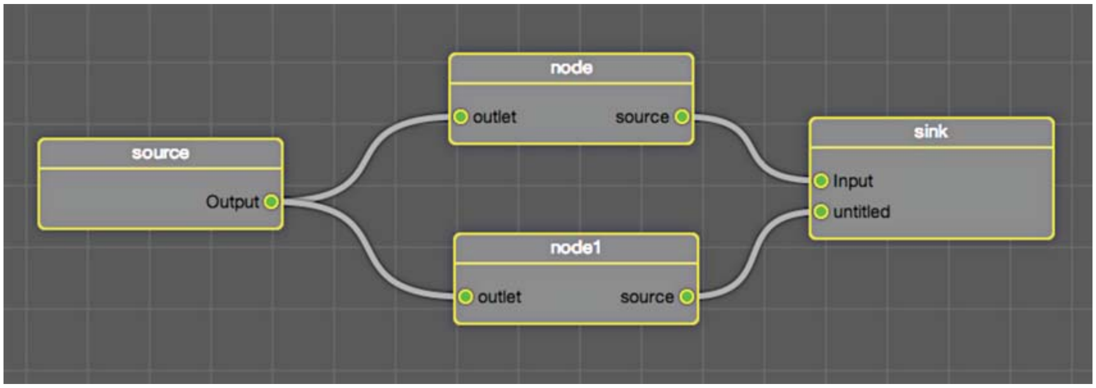


Figure 6: Distributed OpenCL task graph

The programming model chosen was the task graph (Figure 6). A task graph is an explicitly parallel model for describing an algorithm. The concept is used in a number of applications designed for end users. It is used here not only to express parallelism, but to appeal to an intuitive understanding of the user's application. Even without knowing the details of how the tasks work, it is possible to infer the broad form of an application simply by looking at the structure of the graph.

It is valuable to allow the use of any available resources, even when they are part of another computer. A great deal of manual work was required to make distributed resources work together in a cluster. Configuring each computer, diagnosing network issues and developing software are each involved, technical tasks. Distributed OpenCL includes an automatic framework for configuring ad-hoc clusters of available resources. The goal was to make the process of setting up a cluster as easy as running an application.

Solutions for automatically creating a cluster of many computers of greatly differing type and configuration are widely available. These solutions, however, are limited to embarrassingly parallel[7] applications. The foremost example of this technology, the Berkeley Open Infrastructure for Network Computing (BOINC)[8], is only appropriate for enormous problems where no communication between parallel tasks is necessary. BOINC documentation emphasizes that only tasks with thousands to millions of independent work elements are appropriate for this computation model.

The utilization of ad hoc clusters can decrease the cost of computation while improving speed of discovery. More processing power is available on the desktop than ever before, and with the advent of programmable GPUs, it is not uncommon to have several teraFLOPS at each workstation. Distributed OpenCL provides the tools that enable the rapid development of scientific models that can run on a vast array of current hardware, and it functions across an ad hoc cluster of commodity products. The complexity of the underlying processes is hidden from users, allowing them to think about their application rather than the infrastructure that is required to make it function.

[7] Embarrassingly parallel problems are those that are trivially parallelized and typically do not require any interprocess communication.

[8] BOINC, http://boinc.berkeley.edu, accessed May 14, 2012

Architecture

Distributed OpenCL is composed as a stack of loosely coupled software modules (Figure 7). Each module is only dependent on the layers below it. The base layer is responsible for producing a set of network benchmarks, including average latency, UDP packet loss, and throughput. These metrics enable upper levels of the software stack to make informed decisions when creating the network connections used to coordinate cluster nodes, and when opening bulk data channels for transferring intermediate results.

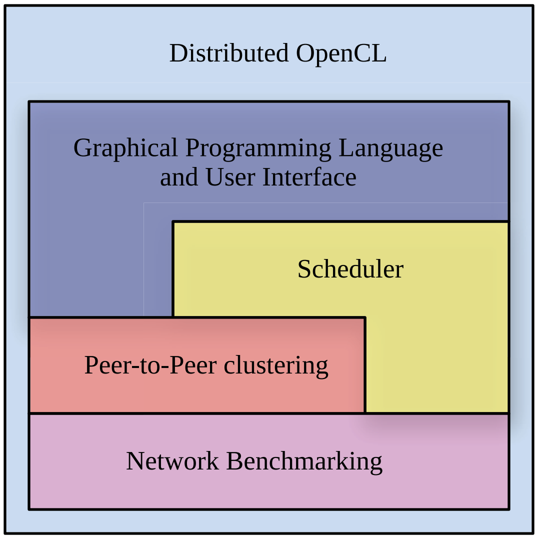


Figure 7: Distributed OpenCL Architectural Diagram

The next-lowest layer of the stack is the peer-to-peer clustering system, which is responsible for opening control channels and reliably passing control messages between peers. A protocol had to be chosen to encapsulate these messages as they are transmitted between peers. Several strategies exist for performing this task. Only those that are well supported by standards were considered, especially Binary

Encoding Rules (BER)[9] and XML[10]. Although not always supported by a standard, object serialization was considered for its simplicity.

Serialization is the process of taking the information in memory (often distributed across several separate regions) and ordering it in a given pattern. On the receiving side, the process is reversed. Most object-oriented programming languages provide tools to aid serialization. The downside of serialization is that the messages are often not interchangeable among languages. This work is intended to be a specification used to develop a suite of compatible implementations; therefore, dependence on a single language is not desirable.

There are standard systems for converting objects into a serial stream of data that are suitable for transmission over a network. The Lightweight Directory Access Protocol (LDAP)[11] uses the Basic Encoding Rules (BER) for extensibly representing structured binary data. BER is a very efficient binary protocol; however, it has relatively few encoders and decoders (relative to XML), and it is much more difficult to debug compared to a text-based protocol.

The eXtensible Markup Language (XML) was decided upon as the container format for messaging in Distributed OpenCL. Though it is inefficient relative to BER, there are many times more implementations of the standard. Because XML is a human-readable text format, it is easy to trace the communication between peers. Finally, XML also includes a syntax and structure verification model. When the document is parsed, its structure is compared to the schema[12]. If the verification

[9] ITU-T X.690: OSI networking and system aspects – Abstract Syntax Notation One (ASN.1)

[10] W3C: Extensible Markup Language (XML): http://www.w3.org/XML/

[11] RFC 4510: Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map

[12] An XML schema is a description of the structure of an XML document.

succeeds, the structure of the document will match the expectations codified in the schema.

The peer-to-peer clustering framework also maintains an in-memory copy of the pertinent statistics and configuration of every other node. This provides other elements in the software stack easy access to the information required for scheduling and diagnostics.

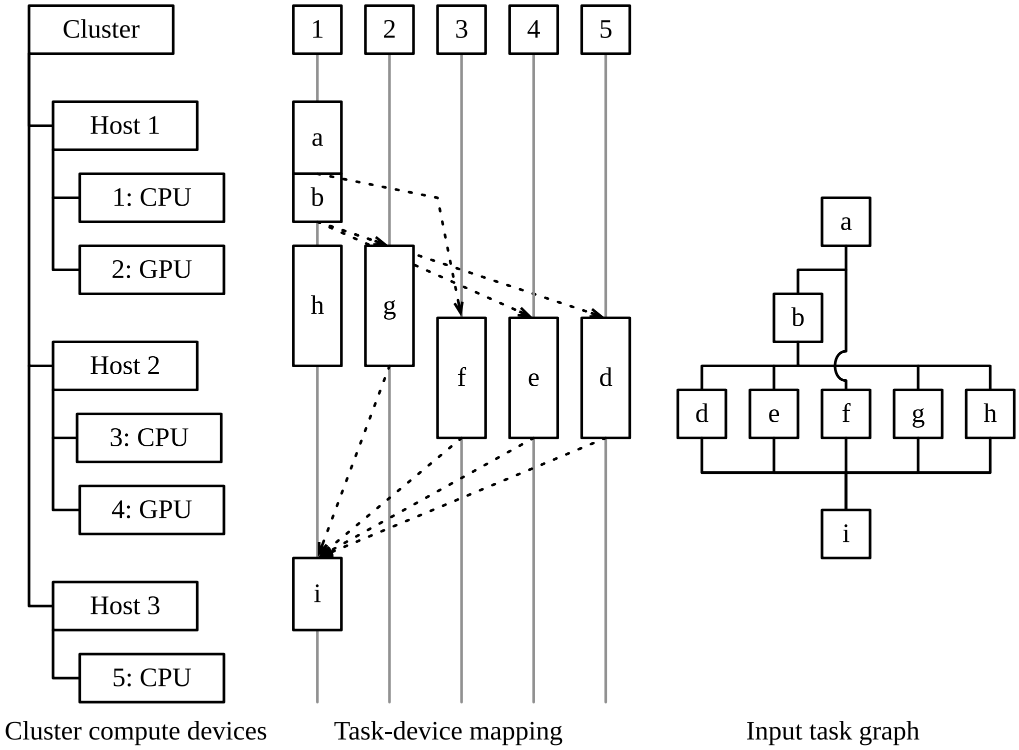


Figure 8: Hierarchal representation of a cluster of five compute devices in three hosts (left). A sample task graph (right) and the mapping between tasks and compute devices (center). The mapping diagram shows the relative duration of compute (box length) and network communication (distance between boxes with dashed lines).

The scheduling layer is responsible for mapping tasks from the user's algorithm to the compute devices responsible for processing them. This mapping is many-to-one, because each task runs on exactly one compute device, and any compute device could be assigned none to many tasks (Figure 8). This layer is extensible and provides a straightforward method for developing new algorithms that generate this mapping. It is also responsible for constructing the concrete manifestation of the abstract representation of the user's algorithm. This includes preparing the compute devices on each of the peers, creating the bulk transfer network connections, and initiating the flow of data between those peers.

At the top of the software stack are the graphical programming language and user interface through which users define their algorithm, monitor the cluster, and submit and monitor jobs. The canonical implementation is written in Apple's user interface and application framework, called Cocoa[13], but is designed so that the core logic is as divorced as possible from the user interface logic. User project files are written to disk in cleartext using XML, reducing the complexity of developing third party tools and editors. Also, by using cleartext document files, it is possible to employ standard version management systems such as Git, CSV, SVN, and Perforce. An XML Schema is also provided to validate document files.

Previous Work

The Message Passing Interface (MPI) (Gropp and Lusk 1993) has been effectively used for nearly two decades. Though it is the de facto standard for cluster computing, it poses significant challenges for new users and non-experts. For a typical user of a community model, such as the Regional Ocean Modeling System (ROMS) (Shchepetkin and McWilliams 2005) or NCAR's Community Climate Model (Kiehl et al. 1998), configuring and troubleshooting MPI is be beyond their abilities. A study done to determine the optimal qualifications necessary for introducing the concepts of MPI found that a course in data communications or networking was required (Apon et al. 2001). The configuration alone of MPI can be a significant challenge to these users, and the knowledge required to set up a cluster using MPI is

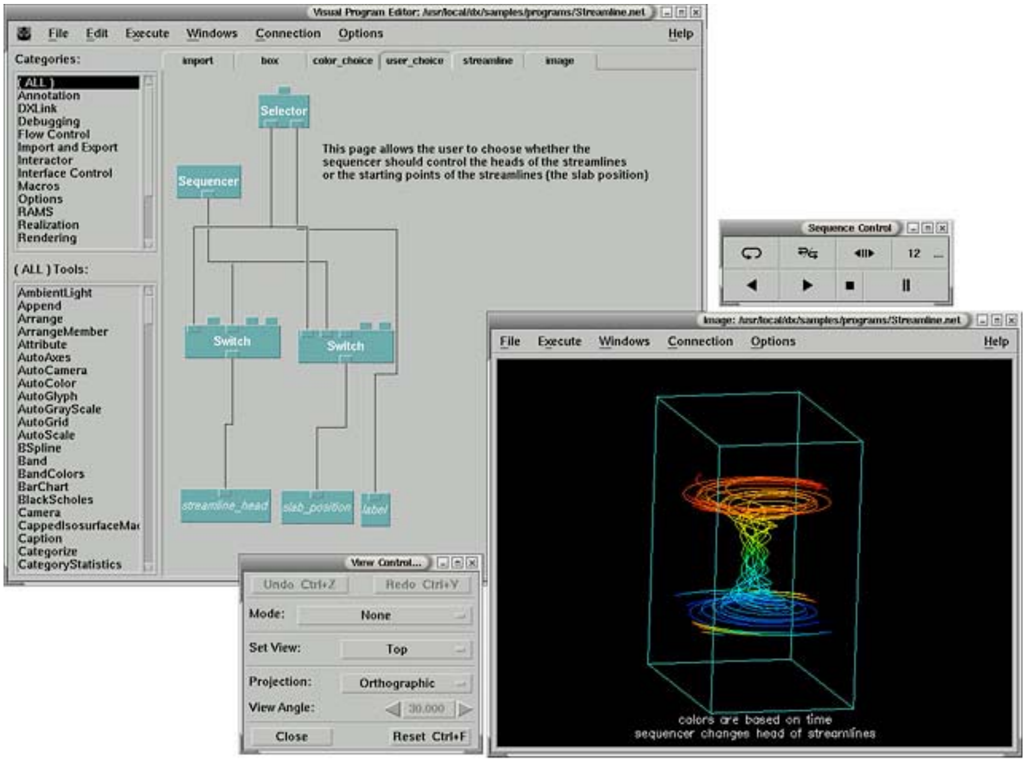
[13] Apple Developer Documentation, Mac OS X Technology Overview, Cocoa Application Layer: http://developer.apple.com/library/mac/navigaion

relatively minor compared with the expertise required to develop new models and applications.

There have been attempts to develop languages that are explicitly parallel. One example, Sequoia (Fatahalian et al. 2006), took the novel approach of explicitly programming to the memory hierarchy. Programmer define their application in terms of ever smaller work units, which are designed to fit into the ever-shrinking memories close to the processing hardware. In the Fatahalian paper, their primary example is a large matrix multiplication. At each level, the task is decomposed into smaller matrix multiplications. For example, a 32x32 matrix could be used as the smallest unit, and it would be able to fit entirely into an example processor's Level 1 cache. Not only are they able to decompose the problem into pieces that perfectly match the underlying hardware, but each of the blocks is intended to execute in parallel. This work unfortunately falls into the same trap as many other programming languages: it is intended for an advanced audience. The cluster support is implemented using MPI, bringing with it additional complexity. Sequoia hasn't made obvious progress in the last six years; though it is occasionally cited in literature, it isn't clear if it is being used for development.

The RapidMind platform (McCool 2008; McCool and Inc 2006) is another explicitly parallel programming package intended to take advantage of GPUs and other emergent massively parallel devices. It is, unfortunately, another example of a programming language intended to reduce the complexity of these applications that doesn't appear to meet its goal. The actual language constructs used in RapidMind are, if anything, more complex and obtuse than those it is intended to replace.

There are several solutions that generate code able to run on the GPU given existing source. Lee et al. describe a method for translating OpenMP applications into CUDA (Lee, Min, and Eigenmann 2009). The Portland Group released a Fortran compiler that can off-load repetitive tasks to the GPU[14]. Another company, AccelerEyes, produced a product called Jacket[15] that can run Matlab code on the GPU. Even Mathworks, the maker of Matlab, added optional GPU support to its platform as part of the parallel computing toolbox[16]. These solutions allow existing applications to incrementally transition to GPU programming. Though these systems simplify the transition to GPU programming, they are limited in their ability to make the most of the platform. Automatic code generators are rarely able to produce solutions as efficiently as humans.



This page allows the user to choose whether the sequencer should control the heads of the streamlines or the starting points of the streamlines (the slab position)

Mode: None

Set View: Top

Projection: Orthographic

View Angle: 30.000

colors are based on time
sequencer changes head of streamlines

Figure 9: OpenDX user interface (opendx.org)

[14] The Portland Group (PGI) CUDA Fortran, http://www.pgroup.com/resources/cudafortran.htm, accessed May 14, 2012

[15] Accelereyes Jacket, http://www.accelereyes.com/products/jacket, accessed May 14, 2012

[16] Mathworks, Matlab Parallel Computing Toolbox, http://www.mathworks.com/products/parallel-computing/, accessed May 14, 2012

There are many examples of graphical programming languages; most are intended to provide a simple and approachable method for defining visualization tasks. OpenDX and Quartz Composer best exemplify these languages. OpenDX (Figure 9) was written in the early 1990s at IBM (Lucas et al. 1992). IBM has since released OpenDX under an open source license. It is intended to be used in conjunction with other scientific tasks and is able to run in a client-server environment. The user interface runs on a lightweight client workstation, with the heavy computation occurring on a mainframe or even a cluster of computers communicating with MPI. It is a powerful visualization tool able to perform complex operations on large datasets. Approaching two decades in age, it has struggled to keep up with current technology. It heavily leverages the X windows toolkit and is difficult for end users to install, requiring third-party solutions. As a visualization tool, OpenDX is not appropriate for general computing tasks; however, the graph-based programming environment is approachable, expressive, and extensible.

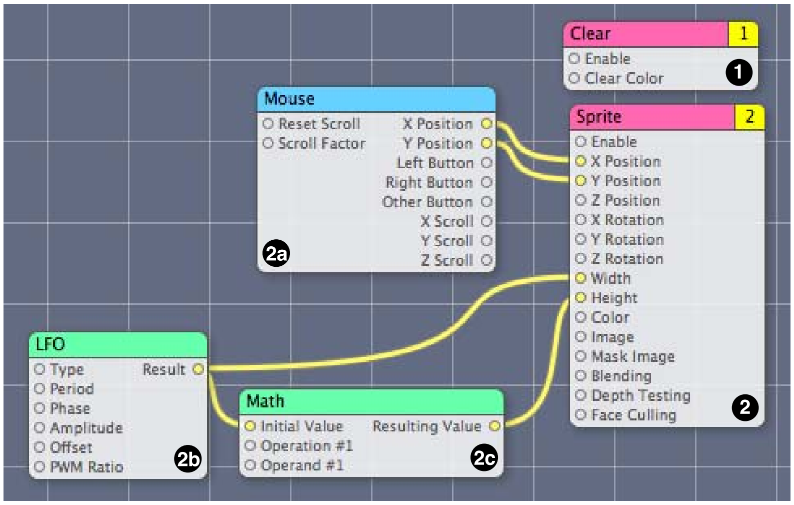


Figure 10: Quartz Composer user interface (Apple)

Another example of a graphical programming language for visualization is Quartz Composer, developed by Apple[17] (Figure 10). It is included in the Xcode Integrated Development Environment (IDE) and is not designed for end users. Quartz Composer is intended to be a tool for testing image transformation filters and developing interactive Quicktime compositions. Like OpenDX, it allows the user to define a task graph with independent operations and explicit dependencies. The pink tabbed nodes (labeled 1 and 2) are output nodes and are responsible for drawing to the screen. The green nodes (2b and 2c) are computation nodes. Finally, the blue node is a user event node. The Quartz Composer runtime system uses this graph to construct a system that evaluates the nodes in parallel whenever possible. OpenCL support was added to the application; there is a node that allows the user to enter custom OpenCL kernels. The graphical programming language used in Quartz Composer influenced the design of the language developed for Distributed OpenCL.

In addition to others' independent work on scheduling algorithms for cluster applications and graphic programming languages, I developed a task graph scheduler for the IBM Cell/B.E. eary in my graduate career. This tool was not able to share work across hosts, but it did serve as the inspiration for this project. The Cell/B.E. scheduler was developed out of necessity; programming for the Cell is notoriously difficult, and the scheduler was intended to abstract some of that complexity away. The programming model was inherently serial, as it was implemented in C. To describe the task graph, the programmer would have provided an SPU kernel task implementation, callback function, and priority. The task implementation was a reference to the compiled object file containing the SPU machine code, and the callback function was a function pointer that was called when the task completed. The callback function's responsibility was to enqueue topologically dependent tasks. The

[17] Apple Inc., Mac OS X Technology Overview, Graphics and Animation: https://developer.apple.com/technologies/mac/graphics-and-animation.html, accessed May 14 2012

priority field was used within a priority heap data structure containing task elements that are eligible to run. This system, while still complicated, significantly improved programmer efficiency during Cell/B.E. software development. In addition, by dynamically mapping tasks to SPUs, the overall efficiency of the system improved. The improvement in system efficiency was due to the balancing effect that task dispatching had on pipelined computation. I realized that the benefits of task graph representations for heterogeneous multiprocessing could be extended by supporting OpenCL and providing support for distributed computation across an ad-hoc cluster. The previous work relating to graphical programming languages provided the inspiration for the form of the task graph representation. Describing these structures graphically leverages more of the human brain than text-based source code is able to. The structure and flow of an algorithm is immediately obvious, and the detailed implementation is available when the user needs it.

Materials and Methods

The network used for the development and analysis of Distributed OpenCL is pictured in Figure 11. A small cluster of Apple MacPros, each of which contains an Intel 82598 (Oplin) 10GBase/T ethernet adapter (Figure 12c), eight 2.66Ghz Intel Xeon cores, and between 6 and 12 GBytes of RAM running MacOS 10.7 (Lion), were used as the ad-hoc cluster of workstations. The MacPros each contain a variety of GPUs, including the Nvidia GeForce GTX285 (Figure 12a) and the ATI Radeon HD4870 (Figure 12b). The configuration of each host is provided in Table 1. Housed in a production computing facility, the machines were loaded into a standard 19-inch computer rack (Figure 12d). Maintenance and management of the machines was completed through Apple Remote Desktop (ARD). Using the ARD interface, it is possible to control the system console and run scripts, either on demand or scheduled.

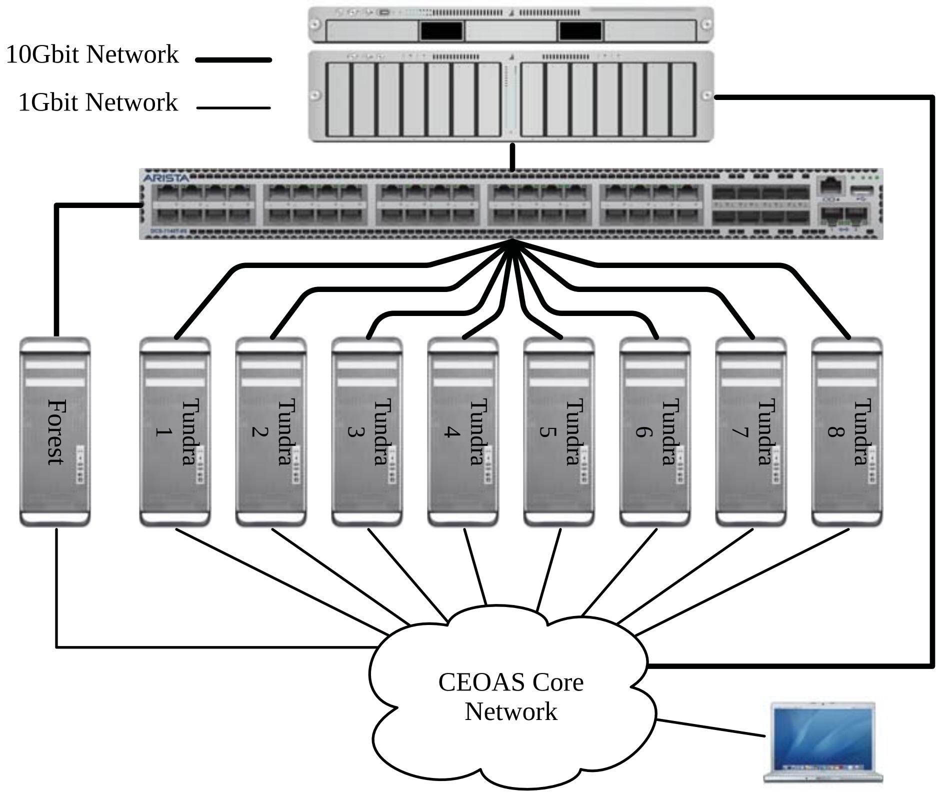


Figure 11: Distributed OpenCL development cluster architecture

Table 1: Development cluster configuration

System	Compute Devices	Memory	Network
Tundra1	2x Intel Xeon Quad-core Nvidia GT120 Nvidia GTX285	6 GBytes	Intel 82598 Oplin, 10Gbit
Tundra2	2x Intel Xeon Quad-core Nvidia GT120 ATI HD4870	6 GBytes	Intel 82598 Oplin, 10Gbit
Tundra3	2x Intel Xeon Quad-core Nvidia GT120 Nvidia GTX285	6 GBytes	Intel 82598 Oplin, 10Gbit
Tundra4	2x Intel Xeon Quad-core Nvidia GT120 ATI HD4870	6 GBytes	Intel 82598 Oplin, 10Gbit
Tundra5	2x Intel Xeon Quad-core Nvidia GTX285	12 GBytes	Intel 82598 Oplin, 10Gbit
Tundra6	2x Intel Xeon Quad-core Nvidia GT120 Nvidia GTX285	12 GBytes	Intel 82598 Oplin, 10Gbit
Tundra7	2x Intel Xeon Quad-core Nvidia GT120 ATI HD4870	12 GBytes	Intel 82598 Oplin, 10Gbit
Tundra8	2x Intel Xeon Quad-core Nvidia GT120 Nvidia GTX285	12 GBytes	Intel 82598 Oplin, 10Gbit

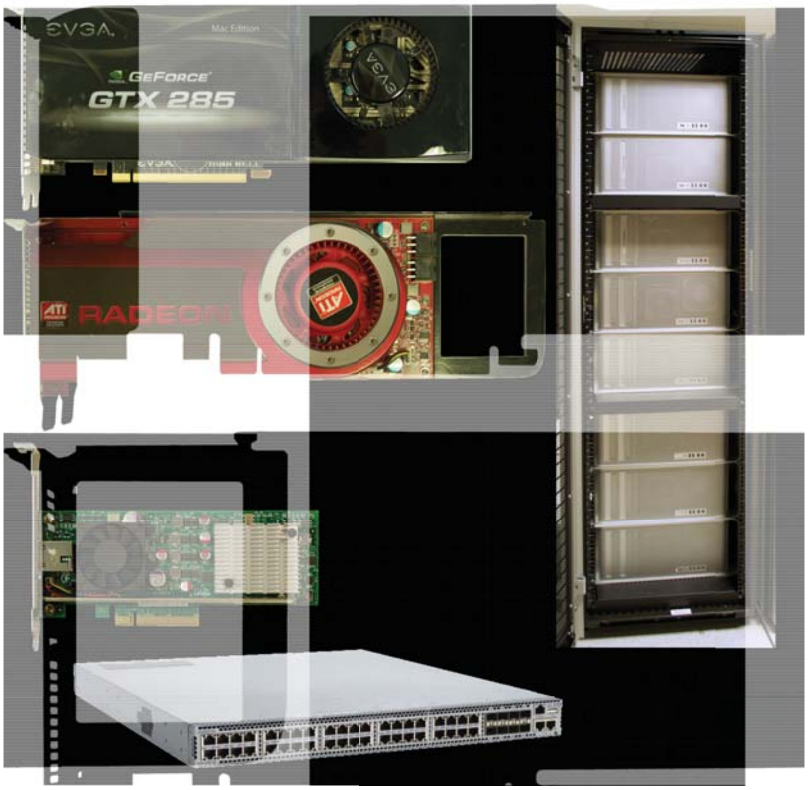


Figure 12: Materials used; (*a*) Nvidia GeForce GTX285, (*b*) AMD/ATI Radeon HD4870, (*c*) Intel 10GBase-T network adapter (82598), (*d*) eight rack-mounted Apple MacPro workstations, and (e) Arista 7140T-8S 10GBase-T switch.

The 10Gbit network fabric used was provided by the Arista networks 7140T-8S 48 port 10Gbit network switch (Figure 12e). The switch has 40 ports of 10GBase-T, and 8 SFP+ module garages. Designed to be low-latency and high-throughput, the 7140T-8S never demonstrated performance less than the 10Gbit line rate. Port-to-port latency is specified to be less than 2.8 microseconds. The largest contribution to the latency is the 10GBase-T physical layer circuitry, which is responsible for producing and receiving the signal used in the twisted pair wiring (Figure 13). Normally, the switch functions in cut-through mode, where packets are switched from source to destination ports without buffering (non-blocking). In some configurations, however, it is necessary to use buffering between ports (store and forward). If there is more than 40Gbit/second throughput from one FM4224 ASIC to another, the switch will transition to store-and-forward mode. To ensure the best performance, all eight cluster nodes were attached to only one of the three ASICs. The cross-sectional bandwidth of the ASIC was sufficient to ensure non-blocking operation at all times.

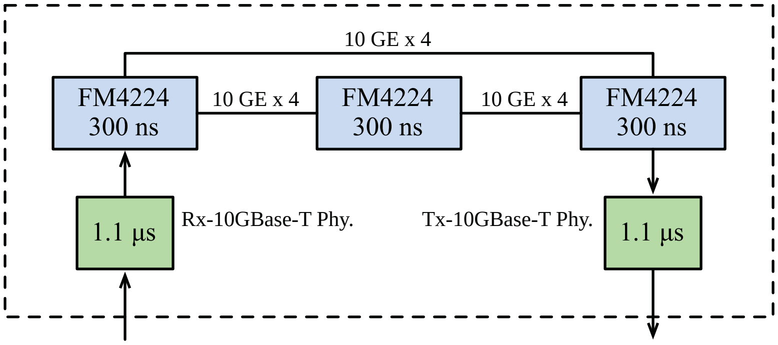


Figure 13: Arista 7040T-8S switch architecture (Source: Arista Networks)

Figure 11 shows the architecture of the network environment used to develop Distributed OpenCL. The architecture was designed to test the platform in a variety of use cases. It was important to identify common scenarios that would be encountered in real-world usage and develop test protocols to verify correct operation. As the platform is intended for ad hoc clusters, it was important to develop a test that demonstrates correct operation when the nodes are attached to the college network in the way that any other workstation would be. Another use case is a purpose-built research cluster used by one or more principal investigators. In this case, a high-speed private network could be designated for the cluster. Cases where the client workstation is and is not a part of this network were evaluated.

In Figure 11, all of the 1Gbit network connections are on the Oregon State University College of Earth, Ocean, and Atmospheric Sciences core network infrastructure. These connections are used in the case of ad-hoc clusters and when the 10Gbit network is used only as a backhaul network. The 10Gbit connections are an entirely private network with an un-routable subnet (172.20.64.0:255.255.240.0). This network functions as the high-speed network that may, or may not, have client access.

It is vital to ensure that the peer-to-peer clustering worked in either case. As it is unconventional to have more than one network connection on a single network node, some protocols made assumptions that do not hold in this case.

The fitness of the algorithms used to implement Distributed OpenCL was evaluated through empirical testing. Whenever possible, comparisons to theoretical best-case scenarios were used. In the case of network benchmarks, comparisons were made against results derived by industry-standard tools, such as Netperf [18].

[18] Netperf, http://www.netperf.org/netperf/, accessed May 15, 2012

Peer Discovery, Resolution and Latency Measurement

Distributed OpenCL is intended to be easy to use and accessible to non-programmers. To achieve these goals, it is important to eliminate any manual configuration, replacing it with automatic resource discovery and configuration. Zero configuration networking (Zeroconf) (Guttman 2001) was chosen for peer discovery and address resolution. Zeroconf is a collection of technologies: Dynamic Configuration of IPv4 Addresses[19], multicastDNS[20], and DNS[21] Service Discovery (DNS-SD) (Steinberg and Cheshire 2005). Apple markets Zeroconf under the Bonjour trademark, and it is intended to eliminate manual configuration of network devices, even on networks that do not have DHCP[22] servers. A device can self-assign an IP address, discover network services such as routers and printers, and resolve IP addresses for these services without configuration or infrastructure. The ease of use that Zeroconf networking promises, if it can be utilized, would dramatically reduce the complexity of configuring an ad-hoc cluster. Though it was developed primarily by Apple, libraries that implement Bonjour on Windows and Linux exist.

Zeroconf networking was designed with consumers in mind, so assumptions were made that are appropriate in that context but troublesome in less common configurations. In a home environment, it is very uncommon for any network device to have more than one IP address, either on the same or multiple network interfaces. In the enterprise, however, this condition is much more common. For example, the research network used during the development of Distributed OpenCL has at least two non-routed subnets on the same VLAN. The first is the standard network that the Internet and file sharing traffic use. The second is used for out-of-band management

[19] RFC3927; Dynamic Configuration of IPv4 Addresses, May 2005

[20]Stuart Cheshire, http://www.multicastdns.org/, Accessed April 24 2012

[21] RFC920; Domain Requirements, October 1984

[22] RFC2131; Dynamic Host Configuration Protocol, March 1997

of servers, commonly marketed under marks such as Integrated Lights-Out Management or Dell Remote Access Console (ILOM and DRAC, respectively). A user that requires access to the internet and remote management networks can either use two network adapters or configure one network adapter with two IP address, one in each subnet. This configuration isn't compatible with Zeroconf networking in its native form. A workaround for this problem was identified, and is presented under the *using mDNS on a network with multiple subnets* subheading.

Peer Discovery with DNS-SD

The peer discovery and resolution processes depend on the mDNS and DNS-SD components of Zeroconf. Normally, DNS servers are specified by the user or automatically through DHCP. Because Zeroconf dispenses with all user configuration and DHCP, mDNS was designed to send DNS queries to a specific multicast group. Every Zeroconf-aware device subscribes to this multicast group and responds to every pertinent query. All mDNS hostname entries are in the virtual domain *local.* and are not accessible from outside the multicast domain. Hostname conflicts with *local.* are prevented by the Zeroconf protocol by requiring new publications to first query *local.* for the existence of a device of the same name. If a conflict is found, a number is appended to the requested host name, and the process is repeated.

In concert with mDNS, DNS-SD adds a record to DNS for service types. DNS-SD allows clients to perform a query for services rather than hosts. See Figure 14 for the hierarchical organization of a Bonjour service name. The service type is the unique designator for a protocol. For example, a query for _ldap._tcp.example.com is a service discovery query for an LDAP server using TCP directed toward the example.com DNS server. Examples of other service types are _http, _ssh, and _ipp for the HyperText Transport Protocol (HTTP), Secure SHell (SSH), and Internet Printing Protocol (IPP), respectively. The underscore characters are prepended to the service type and transport protocol fields to prevent collisions with existing hostnames, as an underscore is an illegal character in DNS hostnames[23]. The Internet Assigned Numbers Authority (IANA) maintains a database with DNS-SD service types[24]. This database is a first come, first served repository for service type identifiers and contains contact names, protocol descriptions, and other information for each service type. The Distributed OpenCL protocol has been registered in the database as _dist-opencl. When used with mDNS, DNS-SD works by performing a similar query, but within the *local.* virtual domain. In this case, the request would be _ldap._tcp.local., and would result in a DNS query placed on the multicast group. Each host participating in mDNS with a matching service would respond.

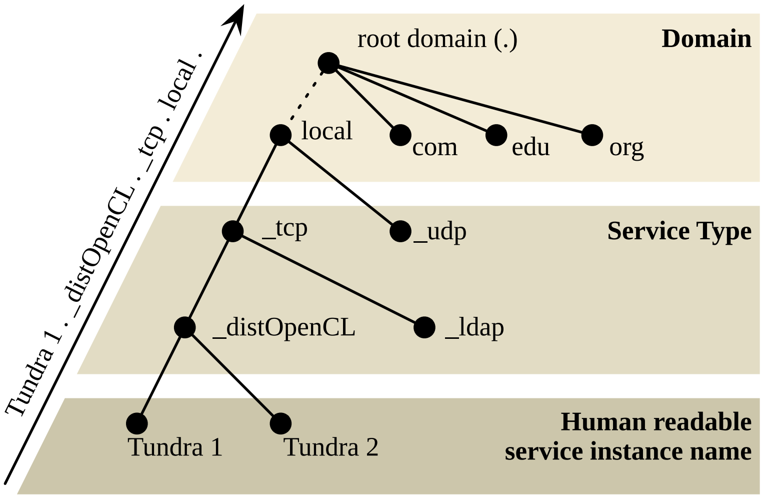


Figure 14: Organization of a Bonjour service name (Source: Apple)

In addition to the service type, DNS-SD allows additional fields to be added to the DNS TXT record, and this protocol defines three such fields, summarized in Table 2. The first field is named UUID and contains the Universally Unique ID (UUID) or Globally Unique ID (GUID) of the host. This field is used to ensure that the node is unique before peering. This field is especially important if a pair of nodes is in a race

[23] RFC2782: A DNS RR for specifying the location of services (DNS SRV)

[24] IANA Service Name and Transport Protocol Port Number Registry: http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml

condition, attempting to peer with one another at the same moment. The UUID field allows for the detection of a duplicate peering and rejects one of them. It is not important that the hardware UUID is used, only that it is unique and remains constant for a given node, though it may change during system reboot.

Table 2: DNS-SD TXT record field descriptions

TXT Record field	Type	Note
UUID	String	Used to ensure unique pairing
TCPendpoint	Integer	Port number for TCP benchmarking
UDPendpoint	Integer	Port number for UDP benchmarking & reachability

The second field added to the DNS-SD TXT record is *UDPendpoint*, which provides the port number of an echo service. The echo service reflects packets back to the sender and is necessary for calculating network latency and mitigating the issues caused by the mDNS edge-case described in the next section. The final field is the *TCPendpoint*, which is substantially similar to the *UDPendpoint*.

Latency Measurement and mDNS Across Non-routed Subnets

Because it uses a multicast group rather than an assigned IP address, mDNS makes no guarantees that a resolved IP address is actually reachable. For example, if one machine is using 10.1.1.2 and another is using 128.193.1.2 on the same VLAN, mDNS will produce query responses between each device, even though there may not be a route between them. A reachability test was devised to address this issue. For each service that is discovered, but is not yet a peer, an mDNS resolution is performed. As IP addresses arrive, they are added to a queue. Each address is then tested for network reachability and latency. This test consists of a series of ping-like UDP packets sent to the UDPendpoint port designated in the DNS-SD service type description. UDP, rather than an ICMP ping, was used because superuser privileges are required to transmit ICMP packets (Wright and Stevens 1995). By using UDP, Distributed OpenCL does not require elevated privileges, increasing security and simplifying application installation.

Table 3: Contents of the UDP ping packet

Offset	Type	Name
0	`uint32`	`ttl (network order)`
4	`uint64`	`timeStamp`
12	`in_addr_t`	`localAddress`

The UDP packets contain a TTL-like field that is decremented each time the packet is reflected. When it reaches ‘0,’ the packet is processed by the reachability algorithm. When initiating a reachability test, this field should only be set with odd numbers; otherwise, it will be processed by the reachability system of the host that did not originate the packet. If this were to occur, the packet would be discarded. The packet also contains a 64-bit, roughly nanosecond precision, time stamp. The content of the timestamp field is flexible in terms of epoch and format. This data is used only on the host that originated the packet, and is only required to be meaningful to that host. Finally, the UDP packet contains an in_addr_t (32 bit integer containing the IPv4 address in network byte order (Wright and Stevens 1995)) field. The purpose of this field is to inform to the sender which address was used to originate the packet. There are no portable APIs that allow user-level applications to know what routing decisions the OS made while sending a packet. The recipient has access to this information, however, when the packet is received; it is in the sender’s address field. By copying this address into the data portion of the packet, we have complete and accurate information.

The resolution/reachability process for each address occurs in parallel. Once one of the available addresses demonstrates a satisfactory packet loss rate, each address is evaluated based on merit. Address resolutions that have poor packet loss rates -- 50 percent or more was used -- are immediately canceled. Other pending address resolutions are allowed to proceed. Once every resolution is complete, the network reachability and latency service provides this information to the upper level in the software stack.

Peer To Peer Clustering

Distributed OpenCL creates compute clusters automatically, using the principles of peer-to-peer (P2P) system design. The central tenet of P2P networking is that the system requires no specialized master node. The definition can be stretched to allow for master nodes, though they are often selected from among the peers. Typically, each peer executes identical code, but masters, or super-peers, assume greater responsibility. The Skype video conferencing system and KaZaA use super peers (Zhang et al. 2010) in their networks. Distributed OpenCL implements a pure peer-to-peer system; at no point are any masters or super-peers required.

There have been other research projects exploring the applicability of P2P systems in the context of science and high performance computing. Most of that research was related to compute grid initiatives (Czajkowski, Foster, and Kesselman 1999). Iamnitchi et al. explored whether P2P architectures can be used for research discovery in grid environments (Iamnitchi, Foster, and Nurmi 2002). Resource management on grids using P2P was attempted (Uppuluri et al. 2005). Resource discovery on grids using super peers (Mastroianni, Talia, and Verta 2005) and resource discovery and membership management (Mastroianni et al. 2005) were explored by Mastroianni. Clusters have also been built using the Gnutella peer-to-peer network protocol (Ripeanu, Lamnitchi, and Foster 2002), which operates over the Internet.

Abbes and Dubacq performed a study that evaluated the applicability of Zeroconf (the approach used in this paper) relative to Pastry[25] for service discovery in a grid environment (Abbes and Dubacq 2009). Pastry is implemented using Distributed Hash Tables (DHT), a common strategy for resource discovery. In their paper, Abbes and Dubacq demonstrated that the Zeroconf architecture is an efficient and reliable protocol for resource discovery, and that Zeroconf is capable of discovering 100 percent of 1,000 nodes in a little as a few hundred milliseconds, easily

[25] Free Pastry, http://www.freepastry.org/, Accessed April 24, 2012

besting DHT. Their work, however, did not include any attempt to initiate TCP connections between each of those nodes. Because their cluster was an in-production compute grid facility, this step wasn't necessary, as cluster homogeneity was assumed. Distributed OpenCL, in contrast, must not make these assumptions. It is necessary to connect with each node to ensure that they are configured correctly, to collect system metrics, and to open a command channel.

By and large, the protocols and techniques in the literature were intended for use with the compute grid initiatives or supercomputer facilities. It is clear that much of the research to date has been to incrementally enhance the capabilities and function of existing distributed computing models. In contrast, DOCL bridges the gap between the complexity of cluster configuration and parallel programming and the user-friendliness of tools such as Matlab.

The P2P system is responsible for establishing control channels between each system that was discovered using the Zeroconf DNS-SD system. Using these control channels, the peers exchange information about their available resources, peers, and network interfaces. This system lays the foundation for the upper layers in the software stack, including cluster management, scheduling, and user interface elements for diagnostics.

In an effort to quantify the wall-clock efficiency of the clustering process, we characterized the time required to complete peering with a variable number of nodes. Theoretically, the time required to build the cluster using the P2P protocol would grow linearly with the number of nodes. The best estimate of the total time required is 14 seconds plus 5 seconds per host after the second host, or $t = 14+\max(0,5*[n-2])$ where t is the wall-clock time and n is the number of hosts. It is possible to reduce the peering time, perhaps to a log factor, but this would increase the risk of duplicate or missed peering. Detailed analysis and empirical data are contained in the results section.

Protocol Details

The peer-to-peer clustering protocol is composed of three layers. The first is responsible for discovering the presence of peers using multicast DNS and DNS Service Discovery, described in the Peer Discovery, Resolution and Latency Measurement chapter.

The next layer is responsible for ensuring an orderly initiation of TCP connections between nodes. It is necessary to have exactly one TCP socket open between each pair of hosts. Reliably enforcing this constraint required the bulk of the engineering effort of the P2P system. There were no examples of previous work appropriate to this task available in the literature. This work likely represents the first example of ad hoc P2P cluster construction using Zeroconf networking. The output of this protocol is a fully connected graph of cluster nodes and TCP sockets.

Finally, the application layer is built using XML. This layer is responsible for ensuring reliable inter-node communication. XML was chosen because it is well known, has many high quality implementations, and provides a mechanism for input sanity checking.

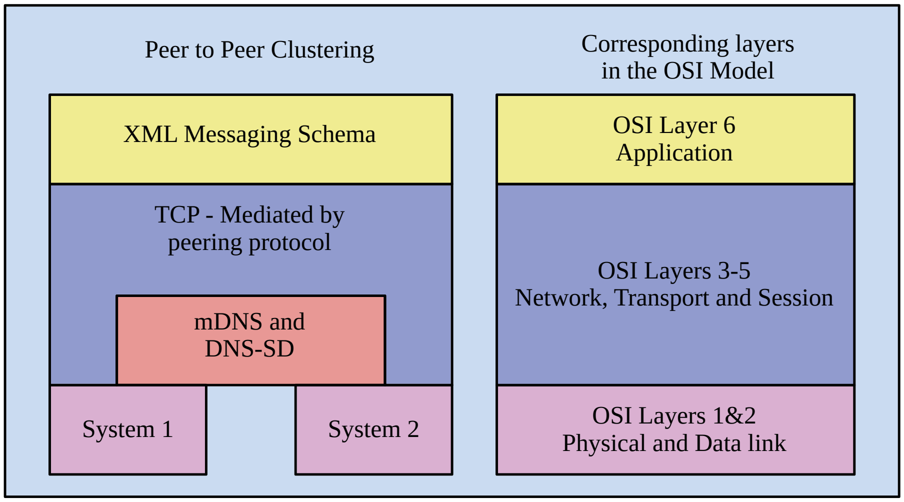

Peer to Peer Clustering
Corresponding layers in the OSI Model
XML Messaging Schema
OSI Layer 6 Application
TCP - Mediated by peering protocol
OSI Layers 3-5 Network, Transport and Session
mDNS and DNS-SD
System 1
System 2
OSI Layers 1&2 Physical and Data link

Figure 15: Architectural makeup of the cluster middleware. The software stack is associated with the corresponding layers in the OSI model. Two peers are shown, but any number of peers could be interconnected using the protocol.

TCP Peering protocol

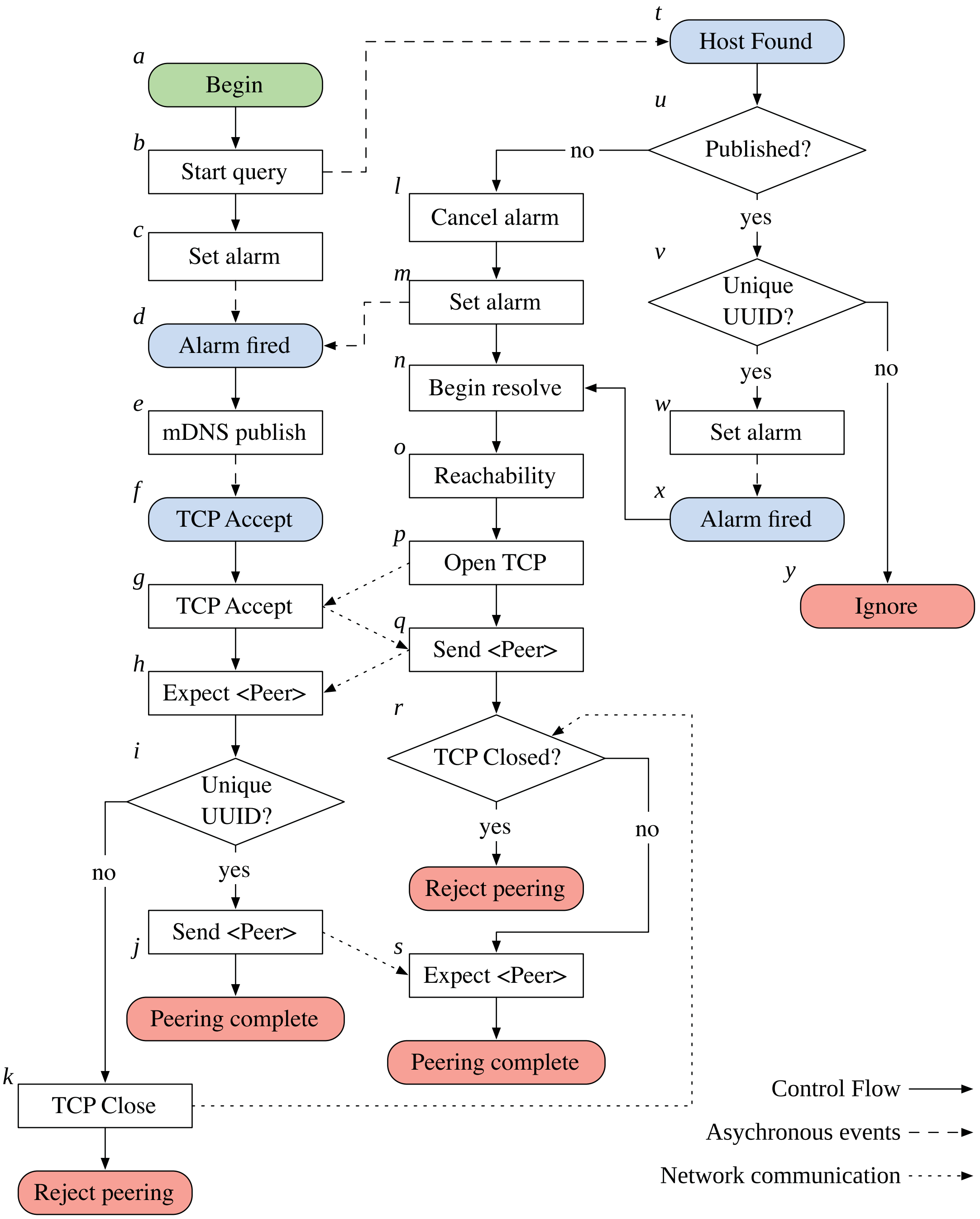


Figure 16: Peering protocol flow chart. Asynchronous events are linked to the action that initiated them with dashed lines. Network communications between peers and their algorithm flow are indicated with dotted lines.

The peering process is implemented as an algorithm that spans a pair of hosts and includes four types of asynchronous events. The complete flowchart for the protocol is presented in Figure 16; and the labels are used to unambiguously reference events. Processing begins (a) with the initiation of an mDNS service type query for _dist-opencl (*b*). Once the query begins, an alarm is set for an interval of fourteen seconds plus a random offset up to five seconds (*c*). Within this period, when a host is discovered (*t*) the alarm is canceled (l) and another is set for five seconds plus a random interval up to five seconds (*m*). Reachable addresses are found using the resolution and reachability processes described above (*n*,*o*). The lowest latency address is chosen, and the TCP control channel is opened (*p*,*g*). The initial *Peer* XML message is sent (*q*,*h*), and the UUID is checked to prevent duplicate peering (*i*). Finally, the peering is confirmed with a reciprocal *Peer* XML message (*j*,*s*).

These alarm intervals were used to mitigate the effects of race conditions during cluster start-up. During development, every host was started at once. With static intervals, every node would transition to the published state at the same time. Newly discovered hosts are only peered with before the local host publishes (*u*). This led to incomplete peering. The random offset reduced the race condition to a degree; however, once the first host (the one with the smallest offset) published, every other host would immediately publish, causing a second race condition. Suppression of this race condition was achieved by deferring the publication by the second random interval. This structure has the effect of serializing the peering process. This serialization forces the scaling of the peering process to a linear factor, rather than constant or logarithmic. The benefit, however, is that the consistent global state of the cluster is maintained.

Even with these strategies, it is still possible for two or more hosts to begin publishing coincidentally. In this case, both nodes will be notified of the new publication after they have stopped connecting to new hosts (*v*). If the UUID of the new host is already known, it is a publication of an existing peer and is ignored (*y*).

However, if the UUID is unique, another alarm is created (*w*). Once this alarm fires, the peering process begins as normal (*x*).

XML Messaging protocol and schema

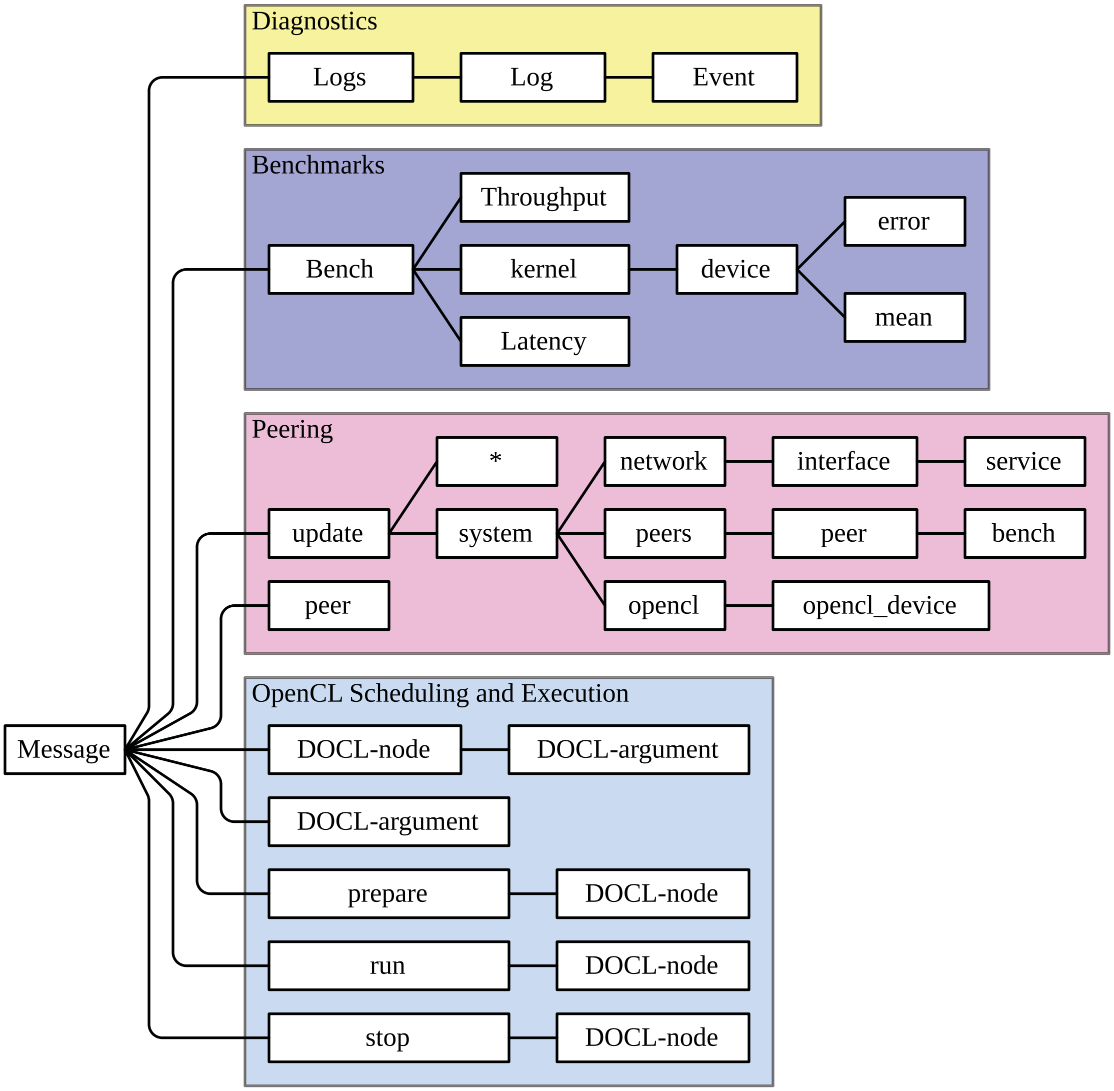


Figure 17: XML Message schema hierarchy. The * node is a wildcard and can contain any node descendant from, and including, the System node.

Once the peer nodes are discovered and TCP control connections are open, it is necessary to define an application layer protocol. XML was chosen for this task because it is sufficiently established and has many robust implementations on virtually every platform, and it is possible to validate XML messages against a reference schema. A side benefit of XML is that the messages are human-readable, even when using packet tracing, simplifying debugging tasks.

The basic vocabulary and hierarchy of the XML messages begins with a *message* node. This node is the parent of every message sent between nodes. A *message* node may have a collection of child nodes, as shown in the hierarchy in Figure 17.

The details of the XML messaging protocol are best illustrated with a brief description of each message and element type. Arbitrary messages can be composed, provided these rules are followed. A value of *null* in the child field indicates that this element can be sent without children. In this case, it could be interpreted with distinct meaning from the variant that contains children. Both meanings, if applicable, are provided in the description.

Logs

Children: *Log*, null

Attributes: *host* - string: the Bonjour host name

Description: The Logs message is a request for the logs from the receiving host if it has no children. In the case where the *Log* child is provided, it is either a response to a previous *Logs* message or an un-prompted update.

Log

Children: *Event*

Attributes: *timestamp* - string: time the log was created

Description: This element is the parent to every log entry. It is best to consider it the console log in its entirety.

Event

Children: String element containing the log entry text

Attributes: *severity* - string: Failure, Warning, Information, or Debug

timestamp - string: time the event occurred

Description: This element encapsulates a single event. The timestamp attribute allows for accurate reconstruction of the sequence of events, assuming the hosts' clocks are synchronized. The severity attributes allow the logs to be filtered.

Bench

Children: *Throughput, Latency, Kernel*

Description: The Bench element contains one or more children. The child elements contain benchmark information relating to their type.

Throughput

Children: String element representing the mean throughput (MB/Sec)

Attributes: *local_address* - string: IPv4 address of the sending interface

remote_address - string: IPv4 address of the receiving interface

Description: This element contains the results of a complete throughput benchmark operation. The local and remote addresses are from the perspective of the sender.

Latency

Children: String element representing the mean latency (in seconds)

Attributes: *local_address* - string: IPv4 address of the sending interface

remote_address - string: IPv4 address of the receiving interface

std_deviation - float: Standard deviation of benchmark samples

packet_loss - float: Observed rate of packet loss

Description: This element contains the results of a complete latency benchmark. In general, this element will be included with the initial *Peer* element in the first message to a remote host. The data represented is likely to be collected during the resolution/reachability testing.

Kernel

Children: *Device*

Attributes: *uuid* - string: UUID of the kernel being benchmarked

Description: This element contains a collection of benchmark results for a single OpenCL kernel. When a kernel update is sent using the *DOCL-Node* element, it is benchmarked on every OpenCL compute device in the local system. These results are added as children to this element. A unique UUID is generated for every kernel. This information is provided when the kernel is updated, and the same value is used in the *uuid* attribute.

Device

Children: *Stats*, string: Type of error; argument, build, enqueue, etc.

Attributes: *type* - string: type of the OpenCL device: GPU, CPU

model - string: device model name as reported by OpenCL

index - integer: OpenCL device index (unique ID)

Description: This element contains basic information about the OpenCL device for a given kernel benchmark. The *type* and *model* strings allow the user interface to display relevant context for the results, and a subset of similar models can be benchmarked to save time. The *index* is used during scheduling to uniquely identify an OpenCL device on a remote machine. If the kernel is unable to run on the given device, the error type is provided as a string element. Reasons for failure include argument, build, enqueue, or unknown failures. If the benchmark is successful, the relevant statistics are provided in the *Stats* child element.

Stats

Children: float: mean runtime for the kernel

Attributes: *min* - float: minimum observed runtime for a single instance

max - float: maximum observed runtime for a single instance *std_deviation* - float: sample standard deviation

samples - integer: number of runs used in the sample

Description: This element contains a digest of the results from a kernel benchmark on a single device.

Peer

Attributes: *name* - string: Bonjour name for the peer

uuid - string: UUID of the peer

Description: The peer element is either a peering request or an acknowledgement. If the recipient has not yet sent a peer message to the sender, it is a peering request. If the recipient has sent a peer message to the sender, it is a peering acknowledgement.

Update

Children: *System* (or any child thereof), string: error

Attributes: *Xpath* - string: Xpath for the update pull or push

Description: The update element allows the peers to transfer their system tree. The system tree contains basic inventory statistics, network information, and cluster data from the remote peer's perspective. The peers structure in the system tree contains a list of every peer the remote system is associated with, including their network benchmark information. This message contains an *Xpath*[26] attribute that allows subtrees to be updated in place, thus increasing efficiency. If this node has no children, it is considered an update request, or pull. An unsolicited update message containing children is an update push.

DOCL-Node

Children: *DOCL-Argument*

[26] http://www.w3.org/TR/xpath/

Attributes: *type* - string: class name for node type

name - string: node name

uuid - string: unique UUID for node

globalWorkSize - string: kernel global work dimension

localWorkSize - string: kernel local work dimension

kernelName - string: OpenCL kernel name (may equal *name*)

buildOptions - string: Build options for the OpenCL compiler

Description: A DOCL-Node element is an XML representation of an OpenCL kernel node. The *type* attribute specifies the class name of potential subclasses. For now, only DAGKernelNode is implemented. The name is the human readable name of the node; *kernelName* is the name of the OpenCL kernel in the source code, and it may or may not be the same as *name*. The *uuid* attribute is the UUID that uniquely identifies the kernel. The *localWorkSize* and *globalWorkSize* attributes specify the size and dimensions of the problem space. These will be described in more detail in the graphical programming language chapter, but they are a 3-tuple of the size of the *x*, *y* and *z* dimensional size of the problem. This is used in OpenCL to spawn work threads.

DOCL-Argument

Attributes: *label* - string: name for the attribute, must be C compatible

type - string: int*n*, float*n*, bool*n*, image2D, image3D

direction - string: input, output

endianness - string: little, big

size - integer: elements in the array (1 for scalar)

uuid - string: unique UUID for the attribute

port - string: port number for source attributes

peer - string: peer UUID for source attributes

Description: The *DOCL-Argument* element contains all the information pertinent to a single kernel attribute. Typically, this element is a child of the *DOCL-Node* element; however, it can be sent alone when *port* or *peer* information is updated or required. Those fields provide the information required to create bulk data connections for passing kernel arguments during execution. The other fields -- *direction, endianness* and *size* -- are used to create OpenCL memory buffers, and to create sample buffers for benchmarking kernels. The *uuid* field is used to uniquely identify the argument.

This schema is the foundation of the cluster management platform. All inter-node communication exists within this framework. The *System* tree was omitted for brevity; its contents are best shown with an example system tree. The listing provided in Appendix B is a sample from one of the development cluster nodes. The opencl subtree contains all of the relevant information collected from the OpenCL library, including software versions and devices. The device entries contain the number of cores, core frequency, memory sizes and maximum work items sizes. The *global_mem_size* attribute provides the quantity of memory available to each of the devices. The CPU device is special, because the global memory is the same as system memory. While the relatively verbose nature of XML may seem inefficient, in practice it doesn’t represent a significant burden. Most XML messages fit within a single 1500-byte packet. If it were to become burdensome, the specification could be extended to use some form of compression.

Prepare, Run and Stop

Children: *DOCL-Node*

Description: The *Prepare* message is a request for the remote peer to initialize a kernel node for processing. Within the prepare method, the kernel arguments are prepared, and the network information is derived and returned to the caller. The *Run* message instructs the remote peer to begin processing for a kernel node, and the *Stop* message halts execution. These commands are described in more detail in the Task Scheduling Framework for Heterogeneous Computing chapter.

Results

To quantify the actual scaling behavior with a variable number of nodes, the number of participating nodes was scaled from three to eight. For each number of nodes, ten tests were run. The peering process was considered complete when every node had peered with every other node, and the peering time was found by taking the delta of the completion time and the time that the first agent process was launched. These results are shown in Figure 18.

The estimated time required to complete the peering process was 14 seconds plus five additional seconds per node after two nodes, or t = 14+max(0,5*(n-2)) where t is the wall-clock time and n is the number of hosts. This estimate was derived by combining the constant time spent searching for published nodes and the additional time added for each discovered node. This additional time does not apply for clusters with two nodes or less, so those cases clamp to the lower bound set by the initial search. The estimated time uses the constant offsets set in the protocol, ignoring the random offset. It was assumed that the random variation would average out. In practice, the value of these constant delays could be tuned. The values chosen are a compromise between reducing latency and ensuring complete discovery on our network.

The observed completion time scaled better than predicted -- 3.37 seconds rather than 5 -- although the y-intercept was higher -- 15.16 rather than 14. The likely cause of the shallower slope in the observed data is that although the nominal delay between each node publishing is 5 seconds, it includes a random variance to reduce the likelihood of collisions. The effect is that the lowest delay time wins, bringing down the average of the total delay. The difference in y-intercept is likely caused by the overhead introduced by resolving the mDNS records and performing the reachability tests.

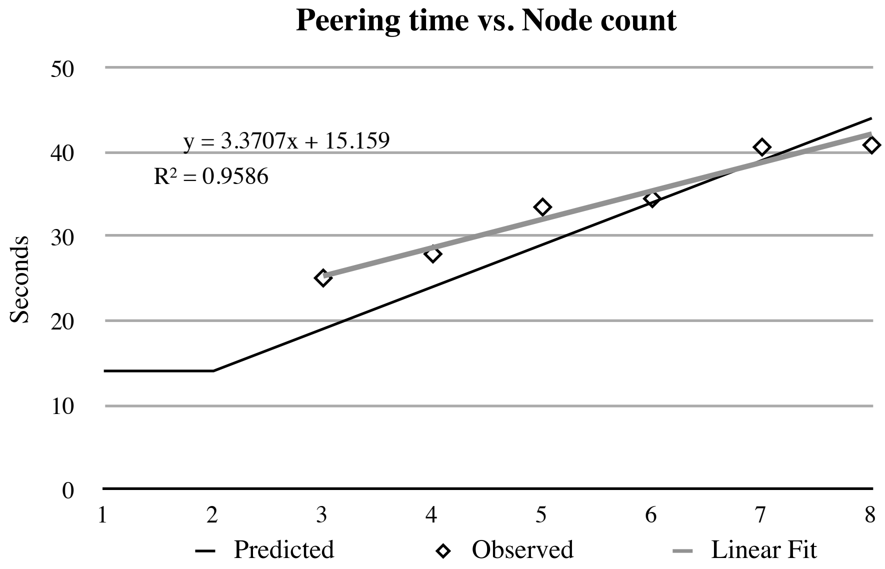


Figure 18: Observed and predicted scaling of cluster creation time.

While the protocol performs well on small clusters, the peering time could become onerous in large clusters. Assuming that the linear trajectory holds through a thousand nodes, it would take approximately an hour for those nodes to peer. Furthermore, because each peer maintains a control connection with every other peer, it is possible to approach the operating system's limit on open file descriptors for a single process. While support for large clusters is outside the scope of this work, addressing these issues remains a topic for future work.

In addition to creating an interconnected cluster of nodes, the XML-formatted system information for every peer is kept current, and an object-based in-memory representation is maintained. This data is used by the task scheduler, and it may be used for asset inventory purposes. The peer graph shown in Figure 19 was created by hand using the cluster report generated by the agent process. A topic for future work is to automatically generate similar diagrams.

A concrete example of the utility of this type of user feedback can be seen in the node description for system named “Tundra 2.” There are two networks listed for this node: the 10 gigabit back-channel (172.20.64.0/24) network and the 1 gigabit administrative interface (128.193.64.0/21). On all the other nodes, the administrative network had been intentionally disabled. This misconfiguration would have likely gone unnoticed without a similar tool.

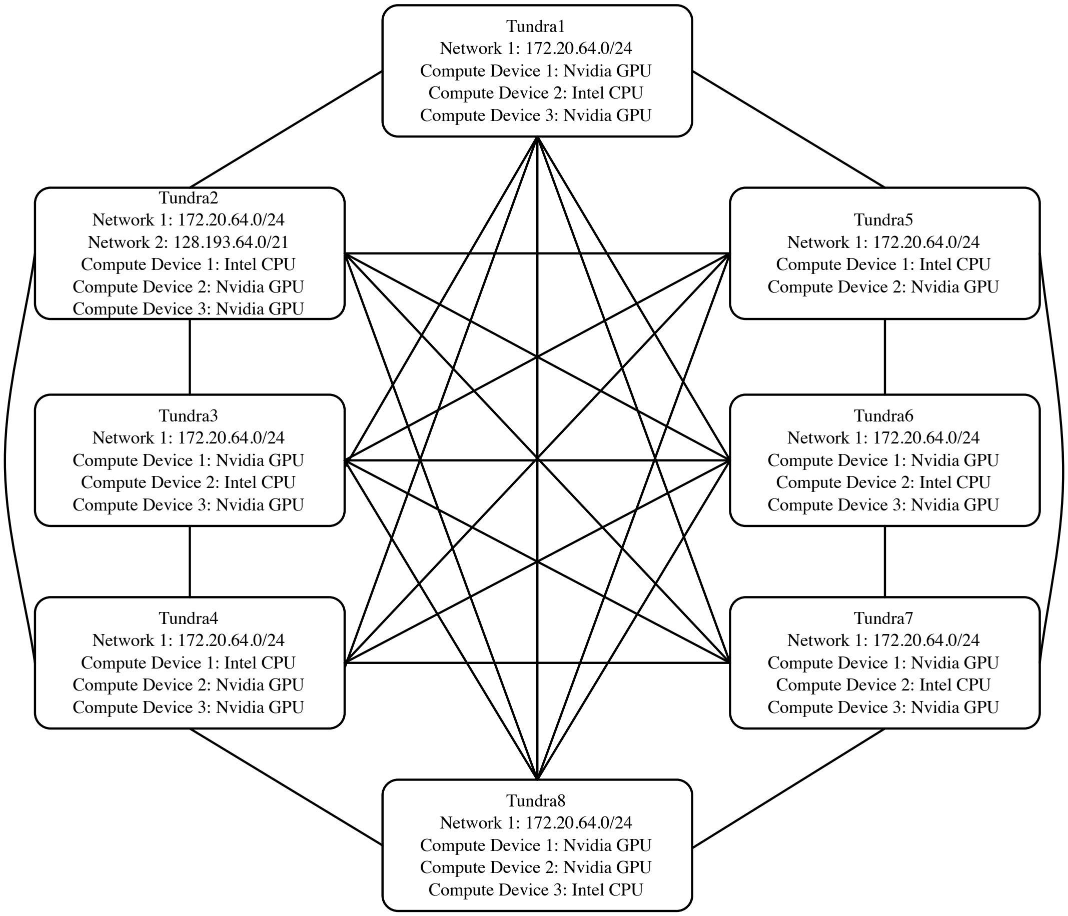

Tundra1
Network 1: 172.20.64.0/24
Compute Device 1: Nvidia GPU
Compute Device 2: Intel CPU
Compute Device 3: Nvidia GPU
Tundra2
Network 1: 172.20.64.0/24
Network 2: 128.193.64.0/21
Compute Device 1: Intel CPU
Compute Device 2: Nvidia GPU
Compute Device 3: Nvidia GPU
Tundra5
Network 1: 172.20.64.0/24
Compute Device 1: Intel CPU
Compute Device 2: Nvidia GPU
Tundra3
Network 1: 172.20.64.0/24
Compute Device 1: Nvidia GPU
Compute Device 2: Intel CPU
Compute Device 3: Nvidia GPU
Tundra6
Network 1: 172.20.64.0/24
Compute Device 1: Nvidia GPU
Compute Device 2: Intel CPU
Compute Device 3: Nvidia GPU
Tundra4
Network 1: 172.20.64.0/24
Compute Device 1: Intel CPU
Compute Device 2: Nvidia GPU
Compute Device 3: Nvidia GPU
Tundra7
Network 1: 172.20.64.0/24
Compute Device 1: Nvidia GPU
Compute Device 2: Intel CPU
Compute Device 3: Nvidia GPU
Tundra8
Network 1: 172.20.64.0/24
Compute Device 1: Nvidia GPU
Compute Device 2: Nvidia GPU
Compute Device 3: Intel CPU

Figure 19: Cluster configuration derived from the XML system report

Automatic Network Cost Measurement

Accurate cost metrics for computation and communication must be available to make good scheduling decisions. The address resolution and reachability testing system described above provides quality latency measurements, but bulk data transfer time is dominated by throughput. With both latency and throughput, it is possible to accurately estimate the total transfer time (network cost).

Supercomputer and purpose-built clusters are designed to have either a well-known or constant network cost structure. A Cray system, for example, is interconnected with a network fabric that is either a 2D or 3D torus; latency increases monotonically as the geometric distance between hosts increases. A cluster built on a single infiniband or ethernet switch will have constant, or near-constant, latency between any pair of hosts.

Ad hoc clusters, however, may have widely varying network costs. If a cluster were constructed using idle workstations in a building, it would be difficult to predict the network cost between hosts. It is common practice to have a “floor switch” for each floor of a building; hosts on that floor will have a relatively low and constant latency between them, and relatively high latency off the floor. Throughput analysis is more complicated in this case. The available bandwidth between floors is often constrained. It is common for the backhaul links, from the floor switches to the core, to be oversubscribed ten to one. In times of low contention, communication between floors will be line-rate; in times of heavy contention, it could be much less.

It is possible to measure throughput by initiating a test transfer of either a fixed size measuring time, or of a fixed time measuring quantity. The constant time method was chosen because it works best across orders of magnitude differences in network speed and allows estimates to be made about the time of completion. A constant quantity benchmark would take far too long on a slow link and would be too short on a fast link. Because every network session on a single interface shares the device throughput, it is vital to guarantee that only one throughput measurement is permitted on any network interface at any time. If this condition is violated, the apparent throughput of the interface will be split between each concurrent session. If the sessions are equivalent, it will appear as though the throughput is approximately 1/*n* with *n* concurrent sessions. An ideal protocol should maximize the number of concurrent benchmark operations occurring across the cluster while minimizing the total time required to measure the throughput through each pair of interfaces.

Theoretical Background and Previous Work

It may be useful to define a few terms used in the analysis of graph coloring and computer algorithms. The complexity of an algorithm is defined as the number of operations required to produce a solution and can be given using a set of bounds. The most common complexity bound is "big-O," which provides the upper bound given the worst-case input. An algorithm that takes, at worst, *n* operations for each *n* input unit would have a bound of $O(n^2)$. The coefficient is omitted for most complexity bounds, with the occasional exception of the tight bound. Problems that have been solved by an algorithm that is upper-bounded by a polynomial (of a small order) are in the *Polynomial* class, or *P*.

A problem is in the *Nondeterministic Polynomial*, or *NP*, class when there isn't a polynomial time algorithm that can solve it, but there is a polynomial time algorithm that can verify whether a solution to the problem is correct. The nondeterministic qualifier is used to signify that random solutions can be chosen and verified in polynomial time. The class *NP-Complete* is a subclass of the *NP* problems. Problems that are shown to be *NP-Complete* are exactly as "hard" as any other problem in *NP-Complete* because they can be transformed from one to another in polynomial time. Also, every problem in *NP* can be transformed into a problem in *NP-Complete*. If a polynomial time solution to any problem in *NP-Complete* was found, every problem in that class could be solved in polynomial time. This would mean that the question "*P=NP?*" is true, and it is generally assumed to be false.

A *complete graph* is a graph that includes an edge between each pair of vertices. The graphs in Figure 20 are two complete graphs with 3 and 4 vertices. The shorthand for a complete graph is K_n, where n is the number of vertices. In the analysis of the benchmarking algorithm, the cluster is assumed to be a complete graph with vertices representing peers, and edges are network connections. This will be referred to as a network graph. It is safe to assume the network graph is connected because that is the form that nearly every cluster will take.

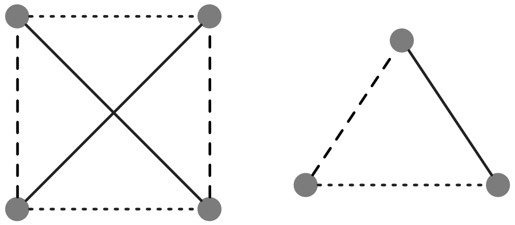


Figure 20: Edge coloring of complete graphs K_3 and K_4

There are only a few, rare exceptions to this. The first is routed networks that were specifically designed to pass mDNS traffic. Because mDNS uses link-local multicast groups, normal routed multicast networks will not pass these packets. Without mDNS traffic, the peering process is not able to discover hosts across the route, and there will be no connections between these routed networks. Allowing mDNS across a router requires configuring network equipment in a way that ignores the link-locality of the mDNS group. This condition is assumed to be very rare, and even if it were to occur, it is likely that the network graph will remain fully connected. Firewall settings on cluster nodes could prevent network connections from other subnets, which would result in other network graphs. If the network was not fully connected, the network benchmarking process would proceed successfully. At no point in the algorithm is the structure of the graph assumed.

The process of assigning the order of benchmark operations is an instance of an *edge coloring* of the network graph. An edge coloring is the labeling of the edges of a graph such that no vertex is adjacent to more than one edge with the same label.

Figure 20 shows two such graphs; in each case, they are labeled with the three colors: dotted, dashed and solid. The *chromatic index* is the number of colors required in the edge coloring of a graph. The chromatic index of a complete graph is *n* when *n* is odd, and *n*-1 when *n* is even. The determination of the chromatic index of general graphs is *NP-Complete*, as is finding the optimal edge coloring (Holyer 1981).

A *multigraph* is a graph in which more than one edge is permitted between any pair of vertices. In the context of networking, a multigraph would occur when one host has more than one interface, each with a different IP address, on the same network. Internet Protocol networking doesn't permit this, so we can safely ignore it. If there are multiple IP addresses assigned to a single physical interface (such as with multi-homed configurations), they will be scheduled independently, which will cause invalid throughput results. Therefore, this configuration is not recommended.

An *online algorithm* is one that produces output before all of the input data are supplied. In contrast, *offline algorithms* only generate results when all of the input data has been read. Offline algorithms for edge coloring that are linear in time exist. This is a desirable trait, and if it were possible to determine when the cluster is in a stable state, it would be appropriate to use one of these algorithms. However, the peer-to-peer and ad hoc nature of Distributed OpenCL make it impractical to detect cluster stability, and once the schedule is set, adding or removing hosts would invalidate the solution. It is possible that the advantage of optimality of the linear-time solution would be lost under dynamic conditions.

Competitive analysis is used to compare the relative performance of online and offline algorithms. The optimality of the decisions made by an online algorithm is affected by the order in which information is presented to it. Therefore, competitive analysis uses a range of input conditions. Bar-Noy et al. found that the greedy algorithm is an optimal solution to the online edge coloring problem (Bar-Noy, Motwani, and Naor 1992). The greedy algorithm produces solutions that use no more than twice the minimum colors required to color the graph.

As the dynamic nature of Distributed OpenCL requires the use of an online algorithm, it cannot attempt to find a minimum coloring of the graph before beginning any benchmark operations. Because the greedy algorithm was shown to be optimal, that was the approach used. As soon as a pair of nodes completes the peering process with one another, they are immediately scheduled.

There are instances in the literature of similar algorithms with applications ranging from load balancing (Sider and Couturier 2008) to switch scheduling (Aggarwal et al. 2003) to wireless channel selection (Duffy, O'Connell, and Sapozhnikov 2008; Leith and Clifford 2006).

Some solutions have been presented that attempt to automatically learn network structure. One example, the NetInventory system, is able to learn the structure of networks, including subnets, routers, and devices (Breitbart et al. 2004). This system, like the others identified, does not collect throughput information.

Significant work was done in Distributed OpenCL to ensure orderly operation and guarantee exclusive access to the interface for a single operation. There do not appear to be references in the literature for a peer-to-peer network throughput measurement tool that behaves in this way. The core contribution described in this chapter is the method for ensuring the highest possible efficiency of throughput benchmarking using network communication as a means for distributed coordination and the protection of a limited resource.

Methods

The success of the cluster throughput measurement system was evaluated by comparing the observed results to the theoretical minimum amount of time required, assuming the minimum edge coloring of the graph. To test the algorithm under the greatest possible stress, a script was developed that starts the software on each host at the same time. The number of hosts was scaled from three to eight to test whether the completion time scaled along with theory. Each number of hosts was tested twenty times, providing enough samples for representative analysis. As each host has two network interfaces (1Gbit/sec and 10Gbit/sec) that are scheduled independently, both results were included in the analysis.

The empirical results were used to evaluate the success of the greedy algorithm and to determine the contribution of protocol overhead to total run time. It was necessary to extract the contribution of each of these factors from the observed signal. To aid in this work, the throughput benchmark duration was set to sixty seconds which is a relatively large value.

Two methods for ensuring mutually exclusive throughput measurements were tried. The first method was designed to leverage the attributes of the TCP handshake to ensure exclusion. This method ultimately proved unsuccessful, though illustrative. It failed because the TCP handshake had been written ambiguously, and it allowed for inconsistent implementations across commonly used operating systems. The second system proved successful and relies on locally locked variables, achieving cluster synchronization through emergent behavior.

Mutual Exclusion using the TCP Handshake

To explain the design strategy of this method, it is useful to briefly discuss the phases of the TCP handshake process and why it was used in an attempt at mutual exclusion. See Figure 21 for the TCP state transition diagram, derived and simplified, from TCP/IP The implementation by Gary Wright and W. Richard Stevens (Wright and Stevens 1995). TCP is a client-server protocol. On the server, a well-known port was opened using the *bind* system call. This port was used as part of the address that the client used to initiate a connection by calling *connect*. With the newly opened port, a *listen* system call was invoked with a user-defined setting for *backlog*, which is the maximum number of clients allowed in the SYN_RCVD state. The *accept* call on the server would have either sent the SYN, ACK response to a client in SYN_RCVD or blocked until the client attempted to connect. On the client side, the *connect* system call used an address and port number pair when it sent a SYN packet to the server. The *connect* call blocked until the SYN, ACK was received from the server.

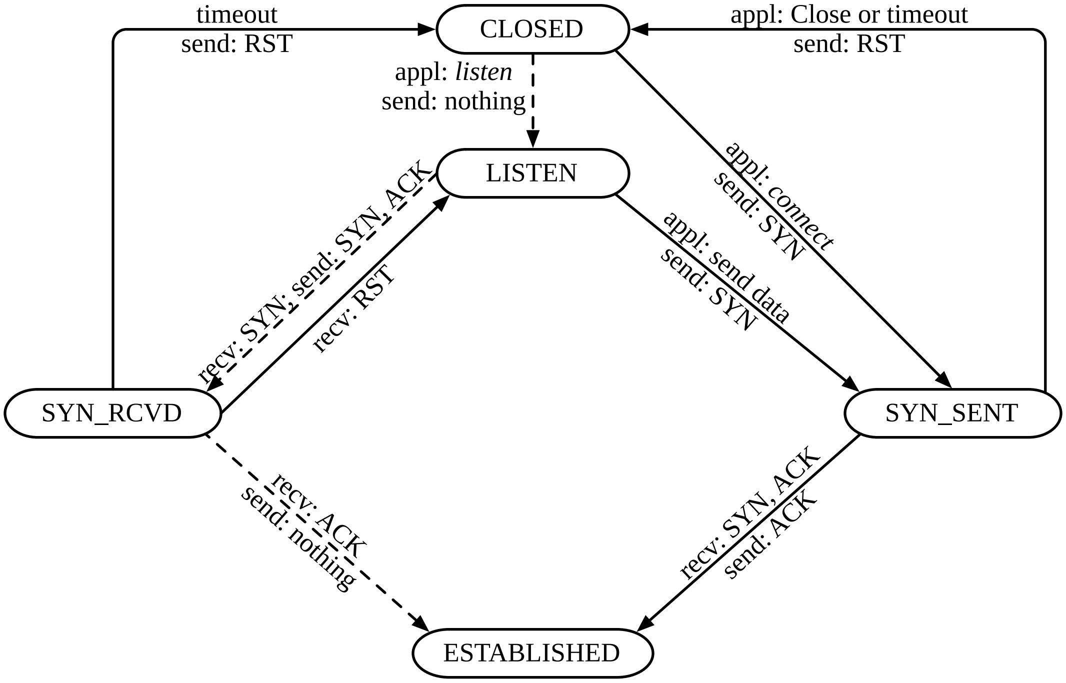


Figure 21: Simplified TCP State Transition Diagram. Derived from (Wright and Stevens 1995).

Under typical TCP use cases, *backlog* is set to a number large enough to balance the dynamic variation of new clients with the speed that the accept thread can accept new clients. It is common to see the backlog value set to five in early textbook examples (Stevens 1990; K. A. Robbins and Robbins 1995). Now, it is more common to see the maximum value the system supports used; the memory required for the backlog structure is small relative to the amount of memory in later systems.

Early specification texts written by W. Richard Stevens (Stevens 1990) and the *listen* man page assert that any new connection attempts will be answered with a TCP reset, causing the client software to receive a ECONNREFUSED error from the *connect* system call. In practice, however, when the number of pending TCP handshakes is equal to or greater than the value set as *backlog*, behavior is essentially undefined. On some systems, including BSD, MacOS, and Linux (with the default behavior), the server ignores connection attempts subsequent to a full backlog. The client would continue sending new connection requests (SYN packets) until the attempt times out, which is typically after one minute. In 1995, Stevens along with Gary Wright (Wright and Stevens 1995), stated that the appropriate action is to ignore subsequent client connections. Between 1990 and 1995, the recommended response to new clients once the backlog is full evolved from sending a TCP reset packet to ignoring it. Finally, in 2004, Stevens and Fenner said this: “For seven different operating systems there are five distinct [...] interpretations about what backlog means!” (Stevens and Fenner 2004) This inconsistency, along with the length of the TCP connect timeout, makes this solution unworkable.

Distributed synchronization using local locks

The eventual solution relied on a locally protected variable to achieve a globally consistent state. The protected variable is a timestamp (TS) that marks the time of completion of the current operation. If there is no current operation, its value is null. This variable is protected using a serial queue. It could be protected using a standard synchronization primitive, such as POSIX locks or semaphores, but these can be a source of inefficiency. Though the serial queue, by definition, causes a serial bottleneck, it is more efficient than the POSIX primitives because it doesn’t require kernel intervention, waking dormant threads, or system calls. Task queues are already widely used in the implementation of Distributed OpenCL, so they were a natural fit. The task queue library used in this implementation, Grand Central Dispatch (GCD), provides serial and parallel queues. With parallel queues, each task is started in order but allowed to run in parallel. Serial queues only permit one task from the queue to execute at a time. In the flow charts presented in Figure 22 and Figure 23, the dashed boxes contain logic that executes on the serial queue that protects access to the TS variable. The logic flow for every client occurs in parallel with every other client and the server. Each peer runs both the client and server logic simultaneously.

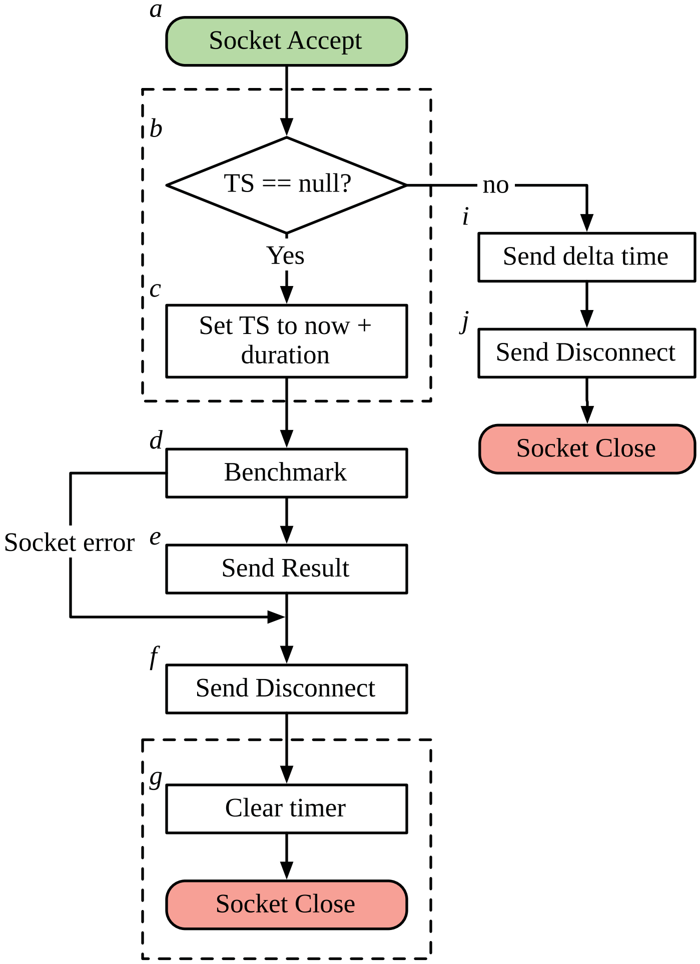


Figure 22: Server side logic for distributed synchronization

Though the system is peer-to-peer, the terms server side and client side are used to separate the control logic for clarity. The server side flow chart is executed when the TCP *accept* system call returns with a new client connection (*a*). When this occurs, a code block is enqueued onto the serial queue that protects the timestamp (TS) variable. This block of code checks whether TS is null (*b*) and, if so, sets the TS to the current time plus the duration of the throughput benchmark (*c*). This value is used within the system as an estimated time of completion for whatever operation is using the resource. If the TS value was non-null, the difference in time between now and TS is sent to the remote host (*i*) along with a disconnect message (*j*), and the socket is closed. Only delta time values or durations are sent to remote hosts,

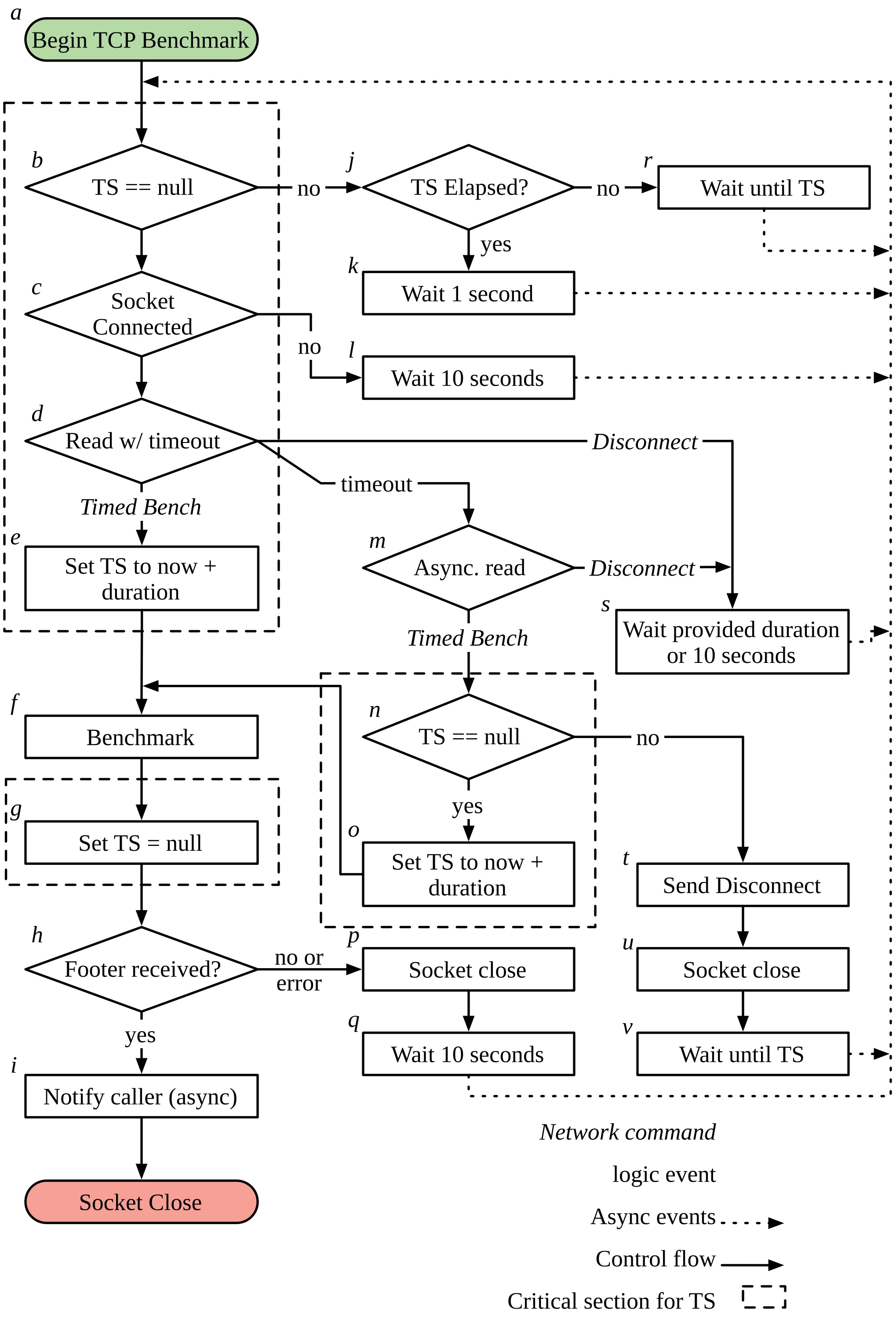


Figure 23: Client side logic for distributed synchronization. Flow begins when a peer is dequeued. Retries are re-enqueued, optionally after a delay. This flow operates in parallel with the server side logic, and is highly multithreaded.

removing the need for coordinated system clocks. Once the TS variable is held, the benchmark is initiated (*d*). When the benchmark is complete, the results are sent to the remote host (*e*), along with a disconnect message (*f*). Finally, a block is enqueued to clear the TS variable (*g*), and the socket is closed.

On the client side, when a throughput measurement is requested (*a*), a code block is scheduled on the serial queue that will check the TS value (*b*). If TS is non-null, its value is checked against the current time (*j*); if TS is still in the future, the same code block is scheduled to run again after TS elapses (*j*). If the time represented by TS has already elapsed (the current operation overran its estimated completion time), the block is re-scheduled after one second (*k*). If TS is null, a socket connection is opened to the remote host (*c*). If this connection fails, the block is scheduled to run again after ten seconds (*l*). Once the socket is open, a network read with a short timeout (one second) is started (*d*). If the read returns with a *Timed Bench* command, which contains the benchmark duration, the TS value is set with the current time plus the duration (*e*). If, however, the network read does not complete within the timeout period, another read is executed outside of the scope of the critical section (*m*). This is to ensure that the network or remote host cannot deadlock the system. If the *Timed Bench* command is received from the asynchronous read, a block is scheduled on the serial queue to test the TS variable (*n*), and set it if possible (*o*). If an error or a disconnect command is received, the first block of the flow chart (*a*) is rescheduled with the delta time provided in the disconnect message if available (*s*), or after ten seconds. If the TS value is not available after the asynchronous read, the client sends a disconnect message (*t*), closes the socket (*u*), and waits the duration of the local TS value (*v*). Finally, if benchmarking is possible, it is run (*f*), and, when complete, the TS variable is cleared (*g*). If the footer containing the results from the server (*h*) is available, the entity that requested the benchmark is notified (*j*). If it is not available, the socket is closed (*p*), and the peer is tried again in ten seconds (*q*).

Through the interaction of the client and server side logic across a collection of hosts, the system reliably converged upon a shared timebase. By sharing the estimated completion time with peers, unnecessary polling was eliminated. The convergence of the cluster occurred within the first time step and, as shown in the results section, was very stable.

Results

The success of the benchmark and distributed mutual exclusion system was measured by collecting benchmarks for a variable number of hosts, as explained in the methods section. For each number of hosts, the test was run twenty times. As each network interface is independently scheduled, the results for the 1Gbit/sec and 10Gbit/sec interfaces were combined to yield forty instances. The results were interpreted with an aim to measure system overhead as well as the optimality of the derived edge coloring in terms of additional colors needed (chromaticity). Finally, the scaling of the algorithm is estimated using these results.

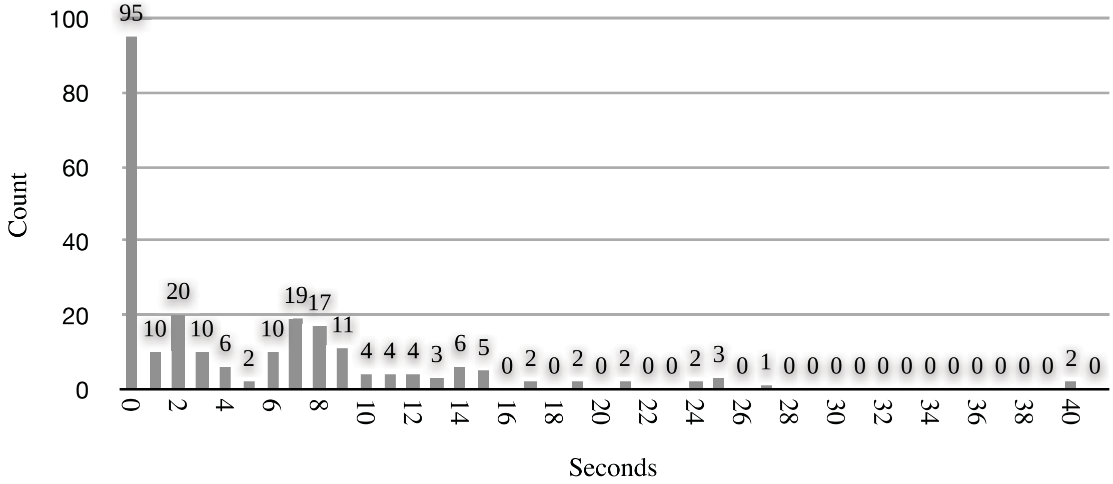


Figure 24: Histogram of total runtime *mod* 60 with 1 second bins, 240 samples total.

To eliminate the contribution of the peer-to-peer clustering protocol to the runtime of this algorithm, time measurement began at the first instance of benchmarking and ended at the time the last pair of hosts completed their benchmark. As the peering process is not instantaneous, and each node in the cluster completed peering serially, there is a small offset in time between when each node is able to begin benchmarking. The implicit synchronization is able to mitigate this effect, as will be seen in the analysis.

The throughput benchmark duration was set to sixty seconds to ensure that the contributions from overhead and chromaticity could be isolated. A histogram of the total run times *mod* 60 is shown in Figure 24. Runtime values were clustered near zero, which suggests that the cluster maintained synchronization with little variation from the 60-second timebase. Thirty-seven percent (37 percent) of all tests completed within 1 second of a multiple of 60 seconds, and 85 percent completed within 10 seconds.

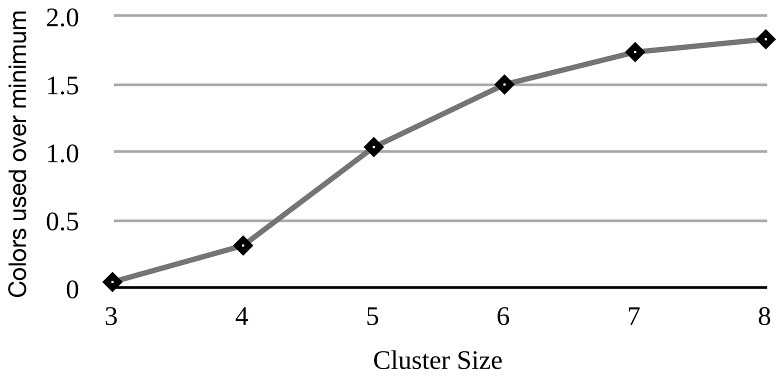


Figure 25: Colors required over minimum versus cluster size

Assuming that chromaticity is the most significant contribution to total run time, it makes sense to evaluate how well the edge coloring algorithm performs relative to theory. The graph shown in Figure 25 demonstrates that the number of extra colors used in the edge coloring follows a roughly logarithmic curve. As the cluster size grows, the number of extra colors per added host approaches zero. Intuitively, this relationship can be explained by considering the expansion of the space of edge coloring solutions for every added color, which is close to exponential.

Finally, in terms of wall clock time, the cost of performing the benchmarks on a variable size of cluster scaled linearly, matching expectations set by theory. Figure 26 shows the relationship between the theoretical and observed increase in runtime versus the size of the cluster. In absolute terms, slope was steeper for the observed results than the theoretical minimum. Omitting the results from the clusters of three and four hosts, as they are trivial coloring problems, the proportion of the time required over theory varied little from about 25 percent.

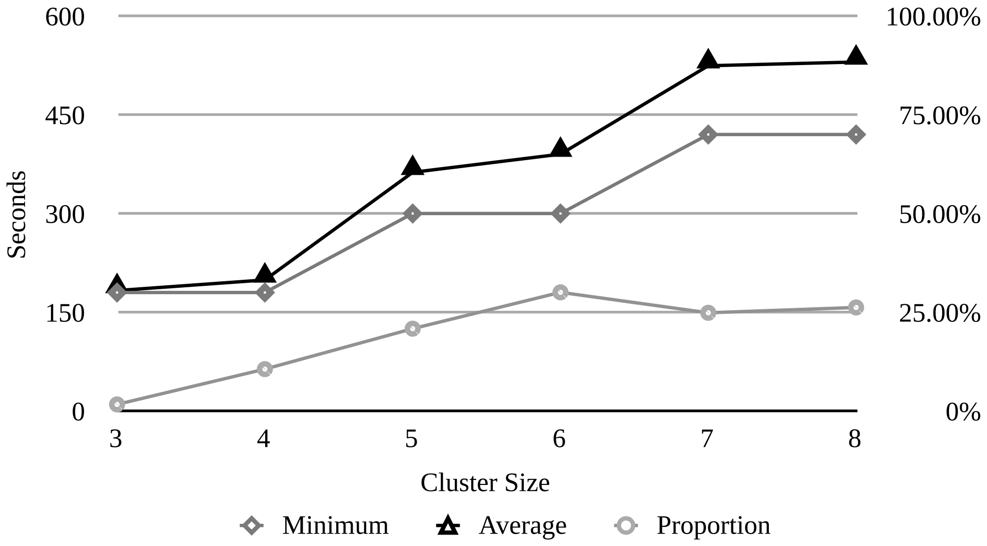


Figure 26: Scaling performance by cluster size

The automatic network cost measurement system functioned well and exhibited linear scaling. It may possible to reduce the time required to determine network cost by selecting a subset of the hosts that require direct measurement. If the topology of the underlying network could be inferred from early measurements, it may be possible to cull later measurements. A nominal throughput value for a network segment could be found, and it would be necessary to only measure the throughput from a given host to the network. The communication cost between a pair of hosts would be equal to the minimum throughput from either host to the network. This is a topic for future work.

Task Scheduling Framework for Heterogeneous Computing

Before performing any work on a cluster, some mapping between independent elements of work (tasks) and the compute device they are to run on must be made. In the case of MPI, this mapping is done somewhat naïvely. The mpirun command assigns threads to processors in the order specified within the host file. If there are five parallel tasks, the first five processors in the host file will be selected and assigned work. This system works well when every cluster node is similar, and when the network fabric is well suited to this purpose. With careful configuration, the network and computation resources can be efficiently utilized. Unfortunately, however, the skills required to maximize the use of resources are often out of reach of the casual user or domain scientist. Many examples of cluster software, such as the Matlab Distributed Computing Toolbox, function in a similar fashion. Specialized software has been developed that manages this process and even allows multiple concurrent users of a cluster if the sum of the requested resources fits within the system's capability.

Distributed OpenCL was designed to provide the best performance possible to non-experts; therefore, a system had to be developed that was able to map tasks to resources in an efficient and easy to use way. By defining an algorithm as a directed acyclic graph (DAG), the user provides the system explicit information about parallel regions of code, data flow, and dependencies. This information is difficult to express using serial code. OpenMP uses compiler directives such as "#pragma omp for" that wrap a parallel section of code. The compiler will then assume that the loop within the directive is parallel and will distribute the work across several threads. Though this may seem straightforward, the details often cause problems for users. The C programming language was designed as an explicitly serial language and the scoping of variables is ambiguous when the loop iterations execute in parallel. To address the scoping problem, OpenMP adds extra keywords that can define whether variables are local to an instance of the loop or shared. The simple concept of parallelizing a for-loop is overwhelmed by the complexity of managing the scope of its variables.

Previous and Related Work

There are several examples of systems that define and schedule work using task graphs. Some of the previous work was covered in the Project Overview chapter, and previous work related to the user interface and graphical programming language will be covered in the next chapter.

Deriving an optimal scheduling of parallel computation is widely known to be *NP-Complete* (Garey, Johnson, and Sethi 1976). Every practical scheduling algorithm must therefore be an approximation. In general, approximation algorithms are a tradeoff between optimality and complexity. The work surveyed here provides context and demonstrates the breadth of the solutions that have been proposed.

Wu and Gajski developed the Hypertool system to ease the user burden involved in programming message-passing systems (Wu and Gajski 1990). Their approach was two-fold. First, they expanded the C compiler's pipeline to produce dataflow graphs, which are substantially similar to the task graphs used in Distributed OpenCL. Using these dataflow graphs, they scheduled and mapped the parallel code segments onto a cluster of homogeneous processing elements. They described two scheduling algorithms: the *Modified Critical-Path Scheduling* algorithm's complexity is $O(n^2 \log n)$, and the complexity of the *Mobility-Directed Scheduling* is $O(n^3)$.

Others have written papers that address parallel scheduling at relatively fine scales. Bannerjee et al. proposed a system for extracting task graph representations of instructions and dataflow from higher level language constructs (BanerJee et al. 1993). Because these automatic techniques generate parallel segments at the instruction level, communication costs would dominate any savings possible from parallel execution. Specialized scheduling strategies were developed to address this issue, such as clustering main segments to run serially (Kim and Browne 1988).

Even machine learning techniques have been used to learn the best scheduler from a suite of options (Wang and O'Boyle 2009). This technique used supervised learning (inferring a function using labeled input data (Duda, Hart, and Stork 2001)) to learn the patterns that exist between parallel code and the performance achieved between scheduling strategies and number of threads.

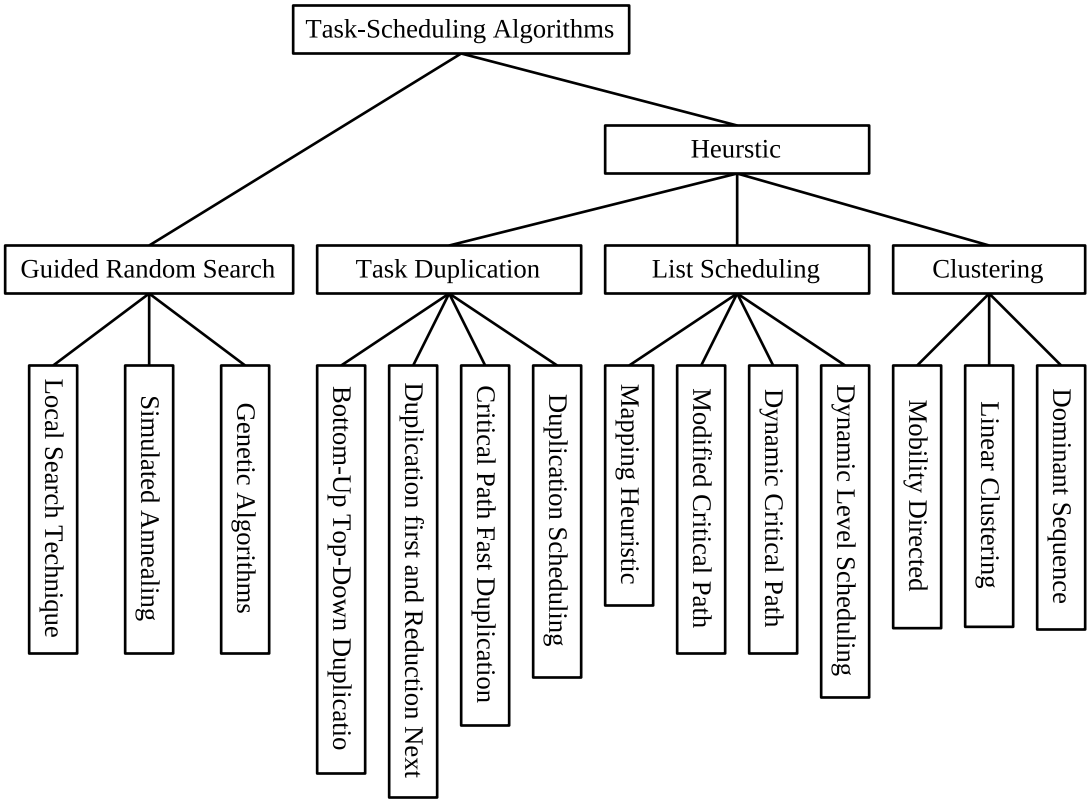


Figure 27: Hierarchy of scheduling techniques

Topcuoglu et al. presented an excellent survey of scheduling techniques and organized them into a hierarchy (Topcuoglu, Hariri, and Wu 2002). This hierarchy is shown in Figure 27. Their work effectively expresses the diversity of available approaches.

Task graph representation and document format

Before discussing the implementation of the Distributed OpenCL scheduling framework, the representation of the task graph, both in memory and on disk, must be fully described. As with the peering messaging format, XML was chosen to structure the task graph representation for storage on disk. In memory, the task graph is represented with a *DAGModel* class that contains the entire task graph.

Task graph Classes and Structure

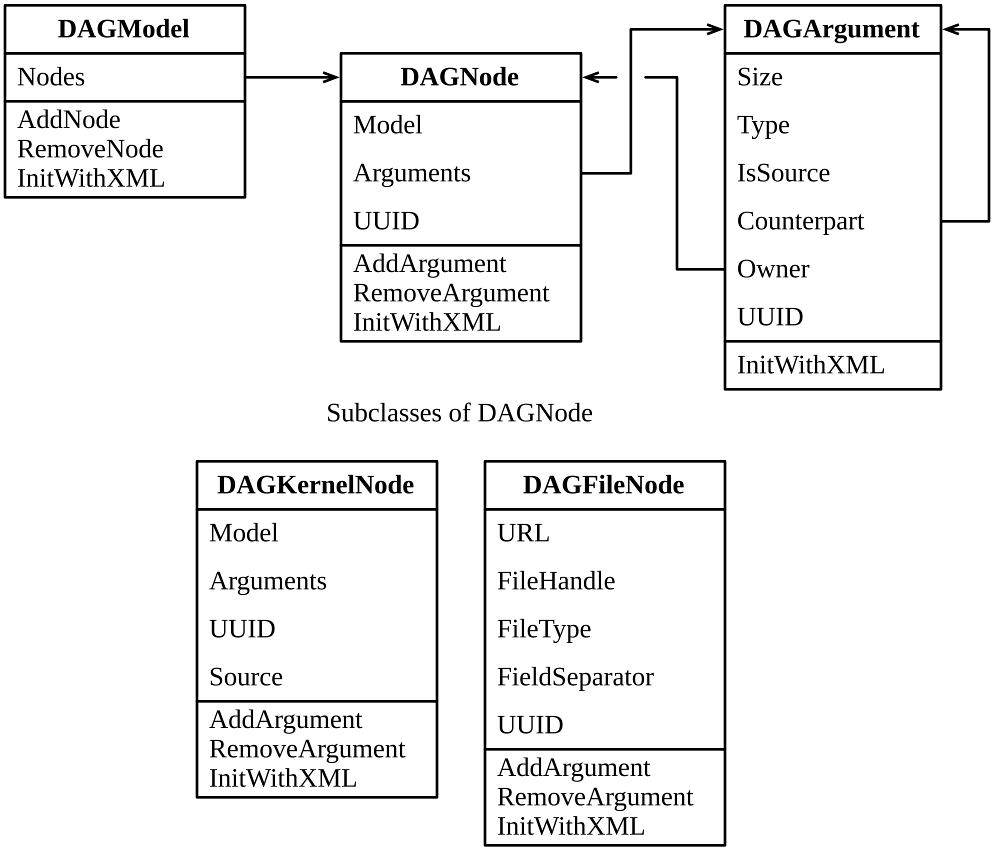


Figure 28: UML[27]-like class diagram of the task graph representation. Simplified for clarity; instance variables relating to scheduling and benchmarking omitted.

[27] UML is a trademark of the Object Management Group, http://www.uml.org/

The *DAGNode* and its subclasses *DAGKernelNode*, *DAGFileSource*, and *DAGFileSink* represent the tasks within the task graph, and *DAGArguments* represents the arguments within the nodes and the links between them. A simplified class diagram, with unrelated information omitted, is shown in Figure 28. Only a subset of the source and sink subclasses are presented.

The classes that represent the task graph maintain the structure of the graph and prevent illogical or degenerate construction. The *DAGArgument* class maintains the size, type and direction of its data; when a connection attempt between arguments occurs, these qualities must match. Conditions such as source-to-source and sink-to-sink connections, as well as multiple sources to a single sink, are detected and prevented. When a connection attempt succeeds, the *Counterpart* instance variable in each *DAGArgument* instance is set with a reference to the other argument; sources maintain an array of counterparts. The *Owner* instance variable contains a reference to the *DAGNode* instance for the node that owns the argument. Through these links, it is possible to traverse the task graph from any arbitrary point to any other point. By performing a graph traversal, it is possible to prevent other illegal conditions such as isolated subgraphs and loops.

XML Document Structure

The XML representation of the task graph is a one-to-one mapping to the in-memory class graph. The XML schema that can be used to validate a document file is presented in Appendix C; and an example XML file with the task graph it represents are included in Appendix D, and the structure of the XML file format is shown in Figure 29. The root of the XML document is the *DOCL-document* element. The children of the document root may either be the *DOCL-node* or *DOCL-connection* elements; these elements define the task graph vertices and edges, respectively.

The *DOCL-node* element represents every subclass of *DAGNode*, including *DAGKernelNode*, *DAGFileSourceNode*, and *DAGFileSinkNode*. The *type* attribute is used to indicate which of the subclasses is being represented. Strictly speaking, the attributes used in the *DAGNode* element are defined by the subclasses, but *name*, *location*, and *type* are created and used by the superclass. The *DAGKernelNode* subclass adds the *globalWorkSize*, *localWorkSize,* and *buildOptions* attributes. When compiling the represented OpenCL kernel, the *buildOptions* attribute may contain compiler options, and the work size attributes are used when invoking the kernel to describe the size of the kernel's problem space.

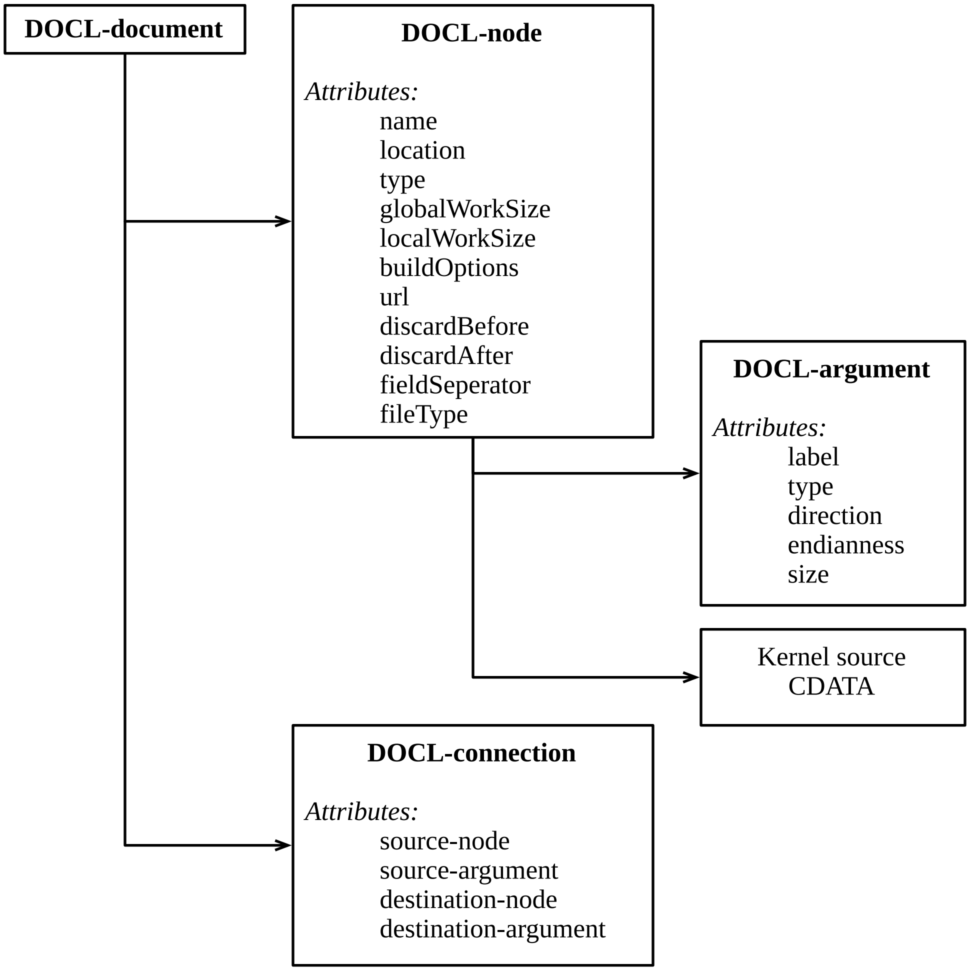


Figure 29: Structure of the XML task graph representation. Subordinate entries are child elements.

The *DAGFileSinkNode* and *DAGFileSourceNode* add several fields, including *url* and *fileType*. The *url* field contains the Uniform Resource Locator (URL) that references the file to be read from or written to. The URL may use a variety of schemes[28], including *file*, *http*, and *ftp*. The current implementation limits URLs to local files, but removing that limitation is straightforward (and a topic for future work). The *discardBefore*, *discardAfter*, and *fieldSeperator* attributes have variable meaning depending on the value for *fileType*. If *fileType* is equal to “binary,” the *discardBefore* and *discardAfter* attributes set the number of bytes at the head and tail of the file to be ignored. If it is equal to “ascii,” these attributes set the number of rows to discard. Providing a number of fields or bytes to discard allows headers or footers to be removed from the data. The *fieldSeparator* attribute is only used with ASCII[29] files, and it indicates whether the fields are separated with commas or tabs. In ASCII files, a record is a row of one or more fields. It is vital that the number of arguments set in the file sink and source nodes is equal to the number of fields in each record, and that the number of fields in a record is constant across the entire file.

The *DAGArgument* element is responsible for representing the arguments for each node. The *type* attribute contains the base type of the argument. Every type supported by OpenCL is a valid setting for this attribute, including *bool*, *int*, *float*, and *image2D*. These types may also be vector types; for example, the *int* type can be *int2*, *int3*, *int4*, etc. Only the image types cannot be vectors. The *size* attribute contains the number of elements in an array of the base type. If the *size* is one, the argument is a scalar. Image types may not be arrays; the size field holds the linear dimensions of the image argument. The *endianness* field is used to ensure that the endianness of the

[28] A scheme is the portion of a URL that defines the protocol used to access the resource. The scheme is presented before the :// in a URL; for example, http in the URL: “http://example.com/path/to/resource.html”

[29] The American Standard Code for Information Interchange (ASCII) character code is the most widely used binary representation for text data.

arguments across different host architectures is respected, though it is only explicitly set within the context of file source and sink nodes. Finally, the *direction* attribute is used to indicate whether the argument is a source or a sink.

To encode the connections between arguments, the root node may have several *DOCL-connection* elements. These elements have no children, but their attributes provide the necessary information to recreate the links between arguments. The names for nodes are unique within a document, and attributes are unique within a node. All that is required, therefore, to uniquely identify a connection between any pair of arguments are the names of each. The *souce-node*, *source-argument*, *destination-node* and *destination-arugment* attributes perform this function.

The XML representation of the task graph is easily processed with any of the widely available libraries, works well with version management systems, and is human-readable.

Automatic Computation Cost Benchmarking

As with communication cost measurement, it is impossible to make good scheduling decisions without computation cost data. Unlike communication cost, however, it is not generally possible to extrapolate computation cost for unknown problems or processors from existing measurements. Both the content of the task and the architectural details of the processor play a significant role in the duration of a unit of computation. To derive accurate computation cost metrics, an automatic benchmarking system was developed. The system distributes task implementations to every device in the cluster, generates test data for each argument, runs the task a number of times, and collects the statistics.

When build or run failures occur, they are collected and returned to the user, providing valuable information about the number of devices that a kernel can target. With OpenCL, it is common for a given kernel to compile and run for some processors and not others. The most salient example of this is the restriction of the maximum local work item size on CPU devices, which may be a single unit, or one-dimensional. Devices such as GPUs typically allow two- or three-dimensional local work groups. The example system tree in Appendix B demonstrates this quality. The maximum work group size for the CPU device is 1024 elements in one dimension, and both Nvidia GPUs support three-dimensional work groups 512 by 512 by 64 elements.

As will be seen in the Graphical User Interface and Programming Language chapter, the OpenCL kernels within tasks are compiled during editing, providing the user with real-time feedback about the correctness of their code. Compiling a kernel is a very fast operation, so it makes sense to do it often. Benchmarking, in contrast, can take tens of seconds. For this reason, it was decided that this operation should be on-demand only; benchmarking is only initiated before processing, or when the user desires benchmark information.

Benchmarking every task on each device is performed through a relatively straightforward procedure. A small set of network transfers, which are depicted in Figure 30, are required. As with all application-layer communication, the *Peer* class is used as a proxy for operations destined for remote hosts. When the benchmark is requested, the *DistOpenCLScheduler* class sends a *benchmarkKernel* message to every *Peer* instance for each kernel. Within the *Peer* class, the XML representation of the kernel is retrieved and sent to the remote host, where it is processed in the matching *Peer* instance. At the destination, a *DAGKernelNode* object is instantiated from the XML representation and is commanded to benchmark itself. Like network benchmarking, the benchmarks must run serially. If kernels were allowed to run in parallel, the results would not accurately reflect the actual performance. A serial dispatch queue is used to enforce this condition. Once the results are available, the *Peer* class instances pass the data back to the caller.

Within the *DAGKernelNode* instance, a sample set of input data must be produced for each of the arguments. This is done with a random number generator. It may be possible to provide sample set generators if the structure of the input data strongly affects kernel runtime. Each kernel is executed ten times to generate basic statistics. This value is variable; however, the results are stable within ten runs. The first execution of each kernel requires significantly more time than subsequent executions. It isn't clear whether this first data point should be discarded. At the moment it is retained.

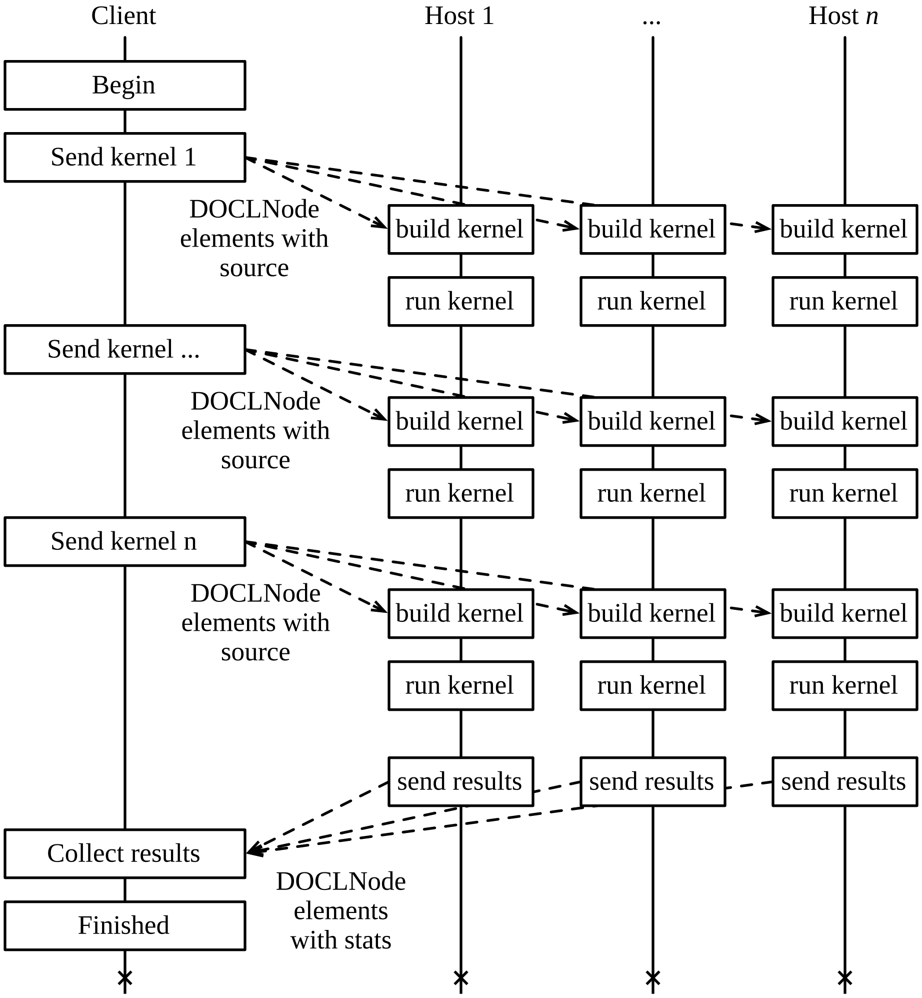


Figure 30: Network flow diagram for task benchmarking.

Scheduling Framework

Given the breadth of the field of scheduling algorithms and the dramatic effect the chosen algorithm can have on the result, a careful decision had to be made about the implementation of scheduling in Distributed OpenCL. It was decided that a modular framework for implementing schedulers should be developed rather than an attempt to derive an ideal solution. The framework approach eases scheduler development by collecting and organizing computation and communication cost metrics into easily queried data structures. To add another scheduling algorithm, a subclass of the *DistOpenCLScheduler* class could be created. The *schedulePendingNodes* method implements the mapping between tasks and devices and provides an easy override point. An example of a very simple scheduler is provided in Appendix E.

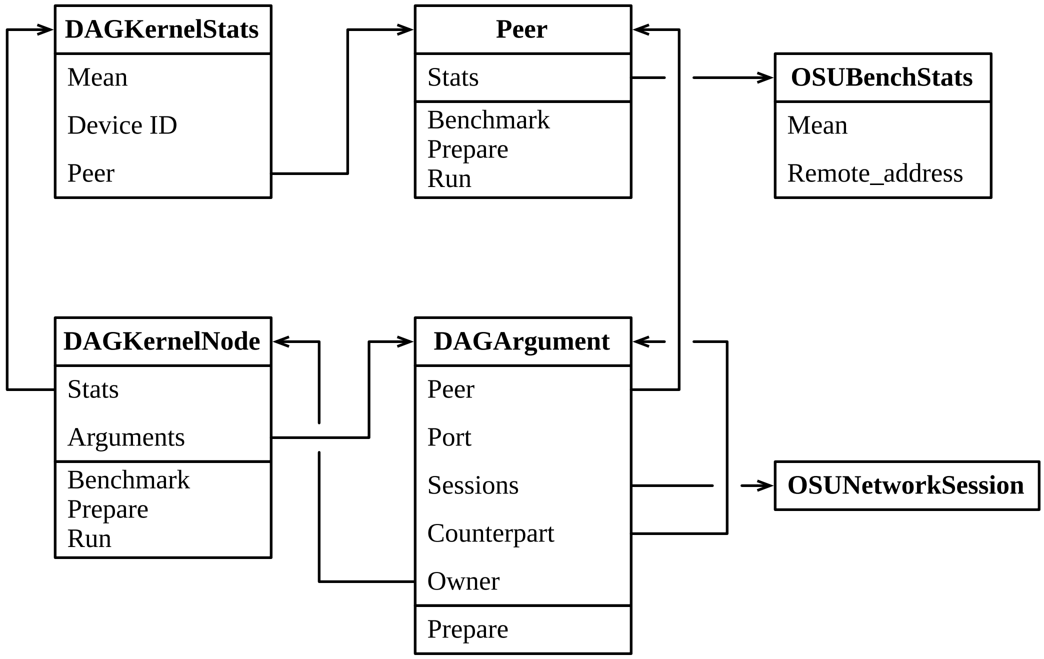


Figure 31: Simplified UML-like class diagram for task scheduling.

The information required to derive a schedule is stored within a collection of classes that encapsulate the scope of each schedulable entity and the costs associated with their interconnections. A simplified class structure of the task graph, omitting structures not related to scheduling, is presented in Figure 31.

An array of *DAGKernelNodes* to be scheduled is created by the *DistOpenCLScheduler* class from the user-supplied task graph. As sink and source nodes run locally to the client, they are not included in the array. Any future subclasses of *DAGNode* are also not included in the scheduling process. Within each *DAGKernelNode*, a collection of *DAGKernelStats* instances are maintained. Each *DAGKernelStats* instance contains the mean runtime for each device and peer that successfully ran the kernel.

The communication cost between, for example, peer1 and peer2 can be found by querying the peer1 class instance for information about peer2. As every *Peer* class instance holds the network throughput and latency information for every other peer and each network, either peer's class instance contains the required information. A convenience method is included in the *Peer* class definition that performs this operation. The results for the best network (highest throughput, then lowest latency) are returned.

When the scheduler decides on a mapping between a task and device, the peer field of the *DAGKernelNode* instance is filled out with the device ID and the UUID of the peer that contains it.

Task Graph Execution

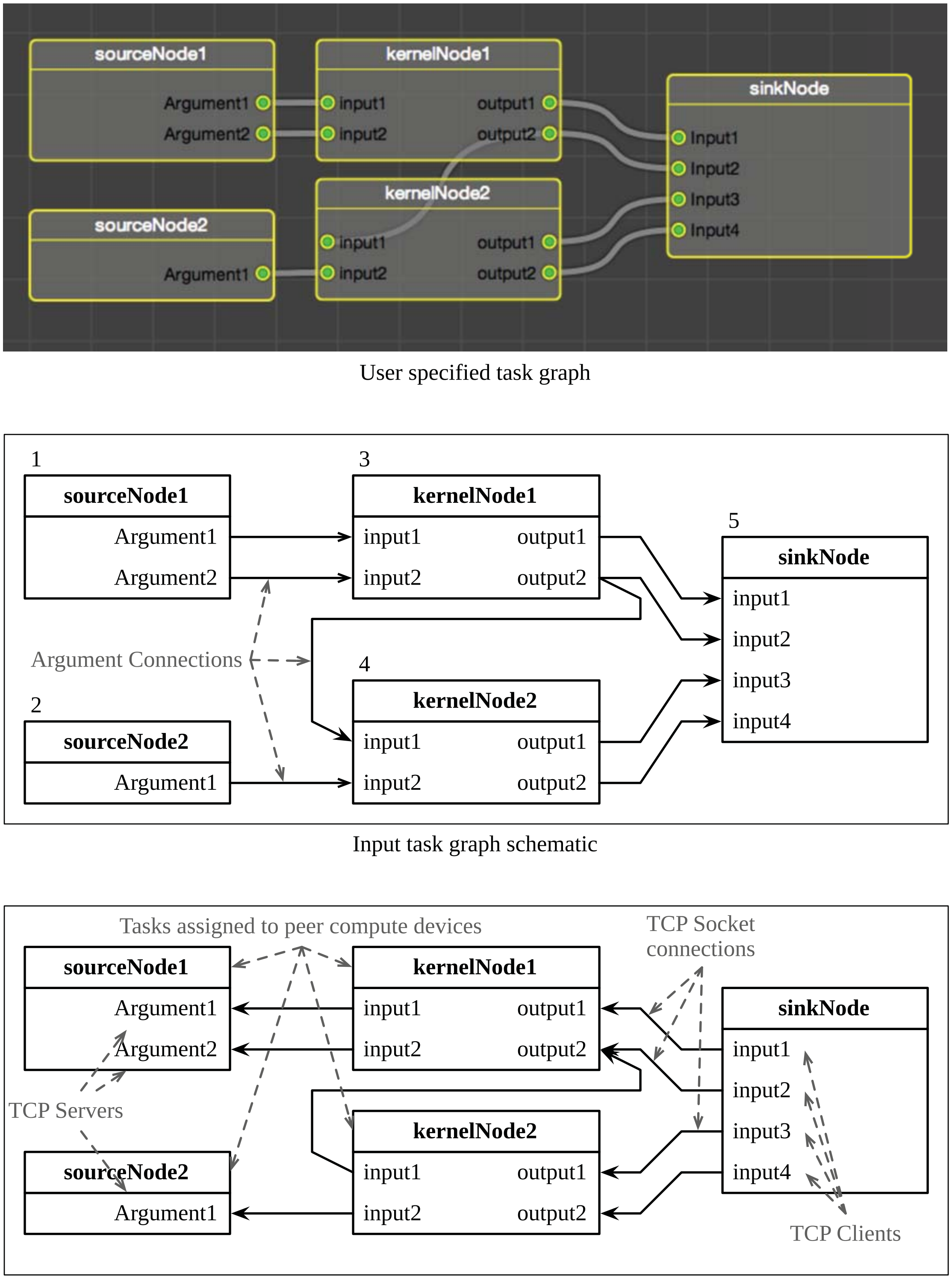
Once the mapping complete is complete, execution can begin. A *topological ordering*[30] of the task graph is generated. The numbers above each node in the input task graph contained in Figure 32 are an example of such an ordering. The topological ordering is used to open network services used to carry data for task arguments. The convention used is that source arguments open the network servers and sink arguments initiate connections as clients. This convention was chosen because it is possible to have several sinks connected to a single source, while the converse isn't true; therefore, the network programming is simplified and better follows the semantics of

[30] A topological ordering is defined such that the starting vertex of every edge is earlier in the ordering than the terminating vertex. A topological ordering is not generally unique; there may be several orderings for a given DAG.

socket programming. In the prepare method, each node is given an opportunity to initialize itself and prepare each argument. At this point, the arguments' network servers and sessions are opened.

Once the preparations are complete, the port number of the source argument is sent back to the calling process, and this information is reflected into the sink arguments downstream. Because the kernel nodes are scattered across the cluster, a mechanism had to be developed to perform these operations on remote peers. The *Peer* class again acts as a proxy; when an instance of this class receives a *prepare* message, the content is encapsulated and sent across the network. Locally to that remote peer, the *prepare* method is executed, and the port number information is collected and returned. Finally, because the sink nodes always run locally, their *prepare* methods make the final network connections. The cluster-embedded graph in Figure 32 describes the structure of the task graph in terms of TCP servers and sockets. The directionality of every edge is reversed to match the semantics of TCP connections to the semantics of task graph connections. In a task graph, multiple sinks can receive data from one source, whereas in TCP, multiple clients can connect to one server.

If the chain of preparation completes successfully, the process is repeated with the *run* methods. The *run* methods are started in reverse topological order. The purpose of the *run* methods is to start the runloop of each kernel node, begin file reading in source nodes, and file writing in sink nodes. Once the End Of File (EOF) marker is encountered in the file source nodes, their network sessions are closed. The close events trigger the cessation of the kernel runloops and downstream session closures. Finally, the sink nodes, running locally, encounter session closures, marking the completion of the graph execution.



User specified task graph

Input task graph schematic

Cluster-embedding of the task graph

Figure 32: Comparison between input and cluster-embedded task graphs

Graphical Programming Language

Reducing the complexity of heterogeneous cluster programming requires careful consideration of the programming model used. A task graph is an instance of a directed acyclic graph in which vertices are tasks and edges are either dependencies or data flow between vertices and edges. Though the implementation details are not necessarily obvious by looking at a task graph image, the structure and flow are easily inferred. As a user, the amount of parallelism present in an algorithm is obvious.

Previous and Related Work

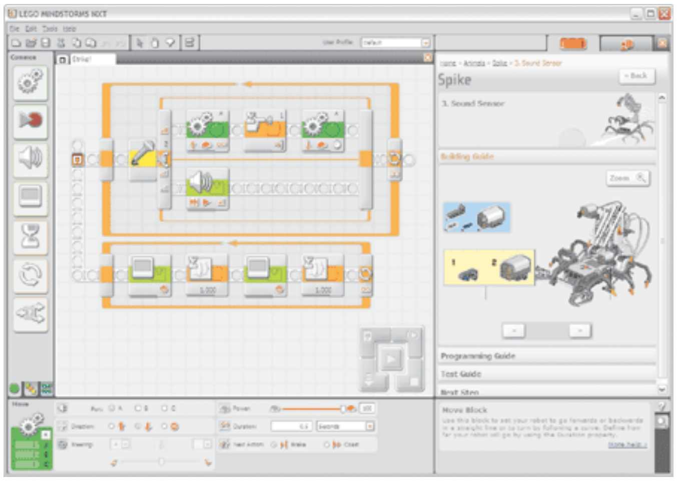


Figure 33: Lego MINDSTORMS programming environment[31].

Graph-based graphical programming languages have been effectively utilized in a vast array of applications while targeting a wide range of user ability. At the very lowest end of user skill is Lego MINDSTORMS, a robotics hardware and software development system intended for children ages ten and up. The programming environment, shown in Figure 33, was developed in conjunction with National Instruments; the core of National Instruments' LabVIEW software was used under a child-friendly front end. Control of robot motion commands, asynchronous event handling and control flow are accessible to the novice computer user. Detailed

[31] National Instruments, http://zone.ni.com/devzone/cda/pub/p/id/8

configuration options for the tasks within the MINDSTORMS software are available underneath the task graph area.

The professional software produced by National Instruments, LabVIEW, targets users ranging from domain scientists to electrical, prototype, and process engineers. The LabVIEW software, while still based on a directed acyclic graph, is designed to mimic a collection of lab instruments and electrical devices rather than tasks. Both the MINDSTORMS and LabVIEW packages support looping constructs within their environments by surrounding looping sections in a “loop” block. This construct enables looped code within a graph that is explicitly loop-free (acyclic). The code within the loop section executes until a stop condition is met. LabVIEW differs from MINDSTORMS and most other task graph languages in that it performs the majority of configurations within the task graph itself. Constant values are represented in the graph alongside tasks. Some of these can be see in Figure 34; the blue boxes containing a number are integer constants, and the yellow boxes are floating point constants.

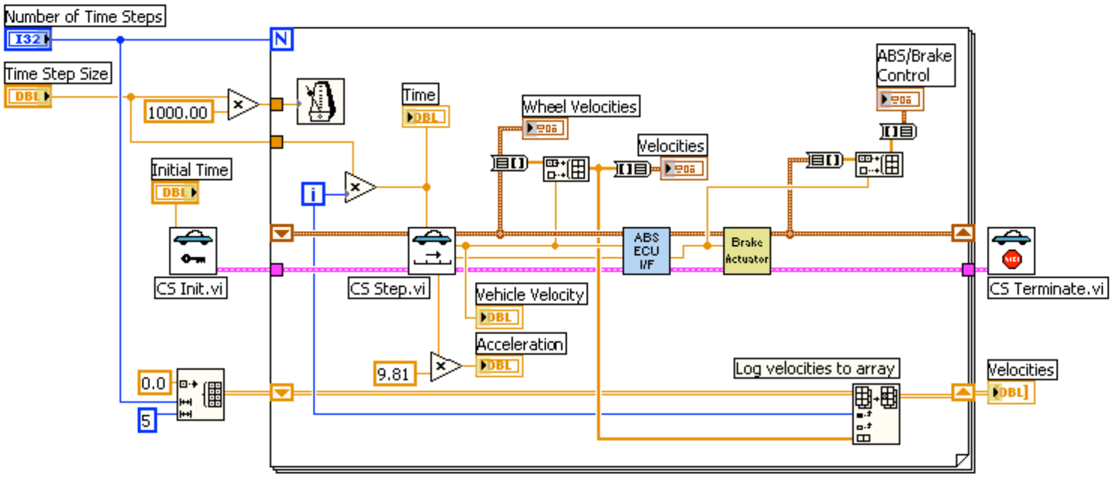


Figure 34: Sample National Instruments LabVIEW graph[32]

[32] National Instruments, Hardware-in-the-loop Evaluation of Vehicle Components with LabView, http://www.ni.com/white-paper/3415/en, accessed May 25, 2012

IBM’s DataExplorer (originally called IBMDX; it was renamed OpenDX when it was released under an open source license), mentioned in the Project Overview chapter, is a graph-based visualization tool. OpenDX took another unique approach for setting constant values within the task graph interface. Every argument associated with a task is represented with a small tab. Tabs on the top of the task are inputs, and tabs on the bottom are outputs. Detailed configuration is available through a separate UI element, like MINDSTORMS, but rather than sharing space on the same application window, and auxiliary window is presented. Task arguments that are set with a constant value through the advanced configuration are represented by a “folded over” tab (Figure 35).

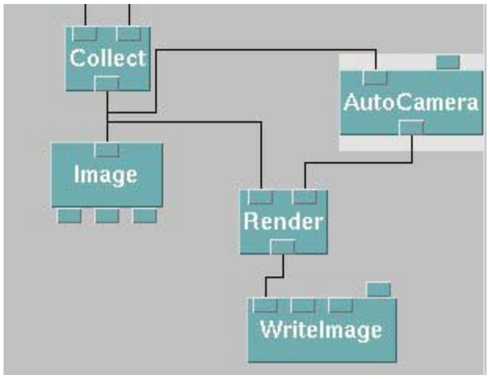


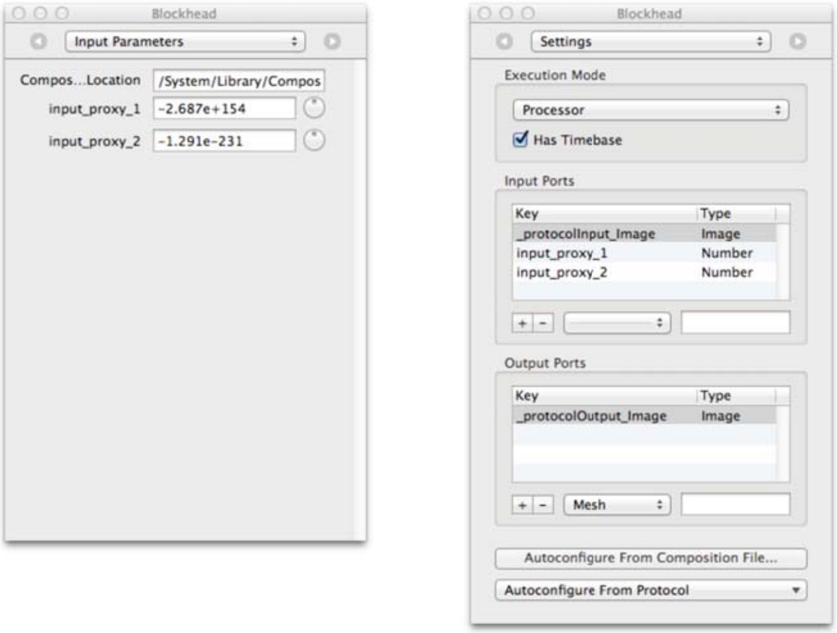
Figure 35: Detail of OpenDX task nodes with folded over tabs. (Source: Numerically Related OpenDX Tutorial[33])

Quartz Composer, also discussed in the Project Overview chapter, implements detailed configuration through an auxiliary window called the “inspector,” shown in Figure 36. Inspector windows are a common user interface construct used in MacOS. The basic strategy is to separate basic and advanced configuration into separate user interfaces. The inspector provides access to patch[34] configuration details and input parameters. Quartz Composer patches can be built-in, OpenCL kernels, or Objective-

[33] http://www.numerically-related.com/tutorials/opendx/opendx_tutorial4.html, accessed May 15, 2012

[34] In Quartz Composer, the tasks are called “patches.”

C plugin modules. Quartz Composer even supports a meta patch which embeds an entire task graph. The patch that is being configured in Figure 36 allows an entire Quartz Composer file (composition) to be imported within the patch. The configuration variables set whether the composition is an input, output or intermediary (Processor), and the number and types of the inputs and outputs.


Blockhead
Input Parameters
Blockhead
Settings
+
-
+
-
Autoconfigure From Composition File...
Autoconfigure From Protocol

Compos...Location: /System/Library/Compos
input_proxy_1: -2.687e+154
input_proxy_2: -1.291e-231

Execution Mode: Processor
☑ Has Timebase

Input Ports

Key	Type
_protocolInput_Image	Image
input_proxy_1	Number
input_proxy_2	Number

Output Ports

Key	Type
_protocolOutput_Image	Image

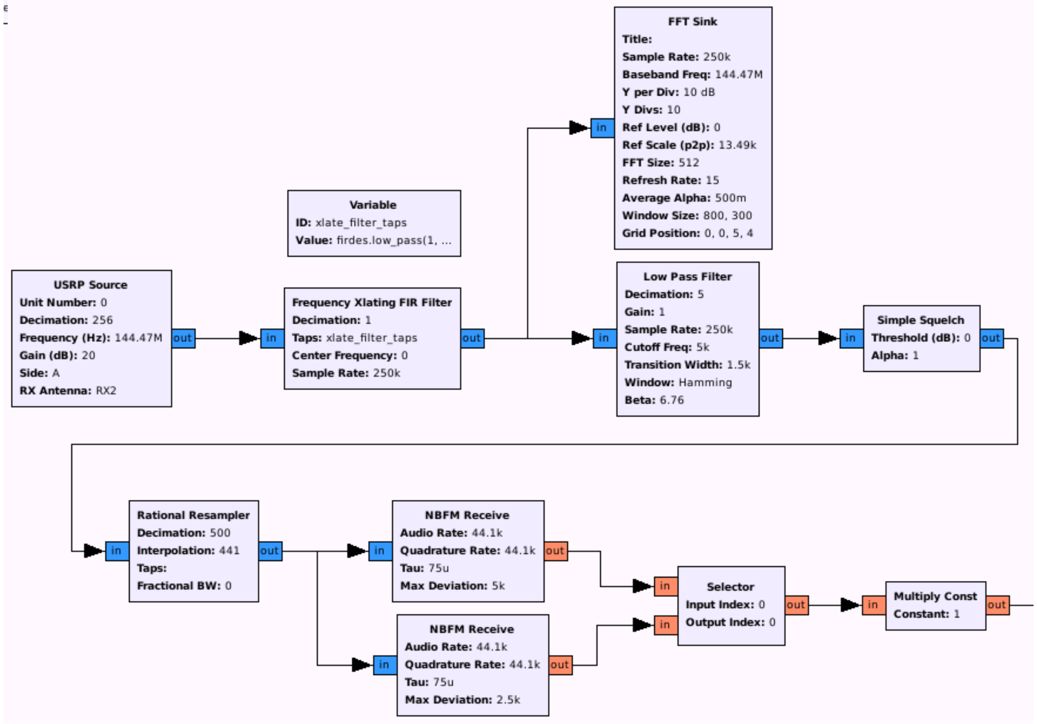
Mesh

Figure 36: Inspector window control for a Quartz Composer patch.

GNURadio is an open source project for software-defined radio. It was implemented using C/C++ and Python[35], and uses the concepts of task graph programming to implement software demodulators and modulators for radio communication. The basic operation of the system consists of defining the task graph within a Python script by instantiating tasks, written in C/C++, and defining the connections between them. A project that started independently from GNURadio, but was later integrated, is GNURadio Companion. The GNURadio Companion software allows the creation of GNURadio Python scripts automatically using a visual task graph. This software allows variables to be set either manually, through an external

[35] Python programming language; http://www.python.org/, Accessed May 3, 2012

editor, or through global variables indexed by name. The object named “Variable” in Figure 37 is an example of such a construct.



FFT Sink
Title:
Sample Rate: 250k
Baseband Freq: 144.47M
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 13.49k
FFT Size: 512
Refresh Rate: 15
Average Alpha: 500m
Window Size: 800, 300
Grid Position: 0, 0, 5, 4

Variable
ID: xlate_filter_taps
Value: firdes.low_pass(1, ...

USRP Source
Unit Number: 0
Decimation: 256
Frequency (Hz): 144.47M
Gain (dB): 20
Side: A
RX Antenna: RX2

Frequency Xlating FIR Filter
Decimation: 1
Taps: xlate_filter_taps
Center Frequency: 0
Sample Rate: 250k

Low Pass Filter
Decimation: 5
Gain: 1
Sample Rate: 250k
Cutoff Freq: 5k
Transition Width: 1.5k
Window: Hamming
Beta: 6.76

Simple Squelch
Threshold (dB): 0
Alpha: 1

Rational Resampler
Decimation: 500
Interpolation: 441
Taps:
Fractional BW: 0

NBFM Receive
Audio Rate: 44.1k
Quadrature Rate: 44.1k
Tau: 75u
Max Deviation: 5k

NBFM Receive
Audio Rate: 44.1k
Quadrature Rate: 44.1k
Tau: 75u
Max Deviation: 2.5k

Selector
Input Index: 0
Output Index: 0

Multiply Const
Constant: 1

Figure 37: GNURadio Companion implementation of a narrowband FM receiver[36]

Graphical Programming Language Design

The graphical programming language developed for Distributed OpenCL was largely inspired by Quartz Composer. Each of the tasks that make up the task graph is represented by a rounded rectangle. The tasks are referred to as “nodes.” The node name is presented in the top region of each rectangle, which is directly editable by clicking on it. The arguments are presented in the lower section of the rectangle. Arguments can be either inputs or outputs; inputs occupy the left side of the rectangle, and outputs occupy the right. The small yellow circle adjacent to each attribute is the click target used for creating, moving and destroying connections between attributes.

[36] http://www.oz9aec.net/index.php/gnu-radio/grc-examples, Accessed May 3, 2012

The connections between attributes are indicated by grey wires following a bezier curve between the click targets; many such wires are shown in Figure 42. When an attribute has at least one connection, the click target is filled in with a green circle. Argument names are editable within the node inspector and are discussed in more detail below.

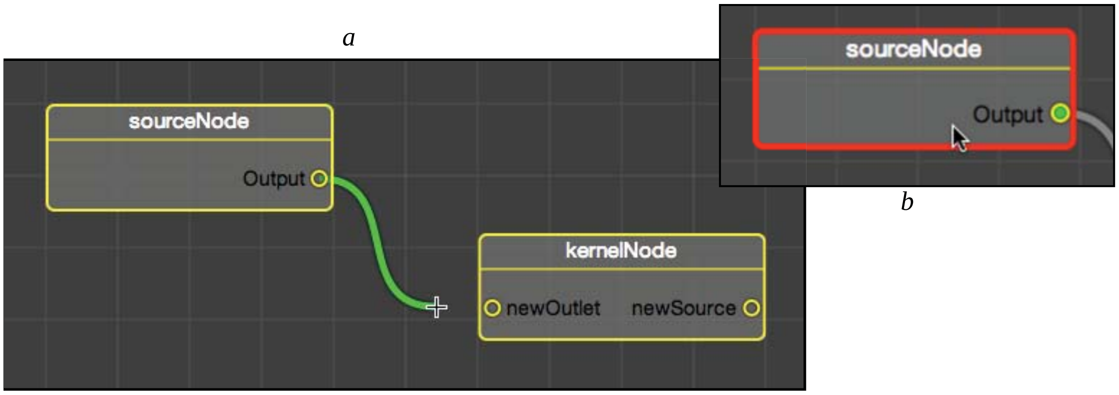


Figure 38: User interaction details; drawing a new connection (*a*) and selecting a node (*b*).

User commands for modifying the graph were designed to maximize usability by leveraging interactions with which users should be familiar. Examples of the method for performing common actions are listed below. For the purposes of this discussion, it is assumed node clicks do not include clicks on the name field or argument click target.

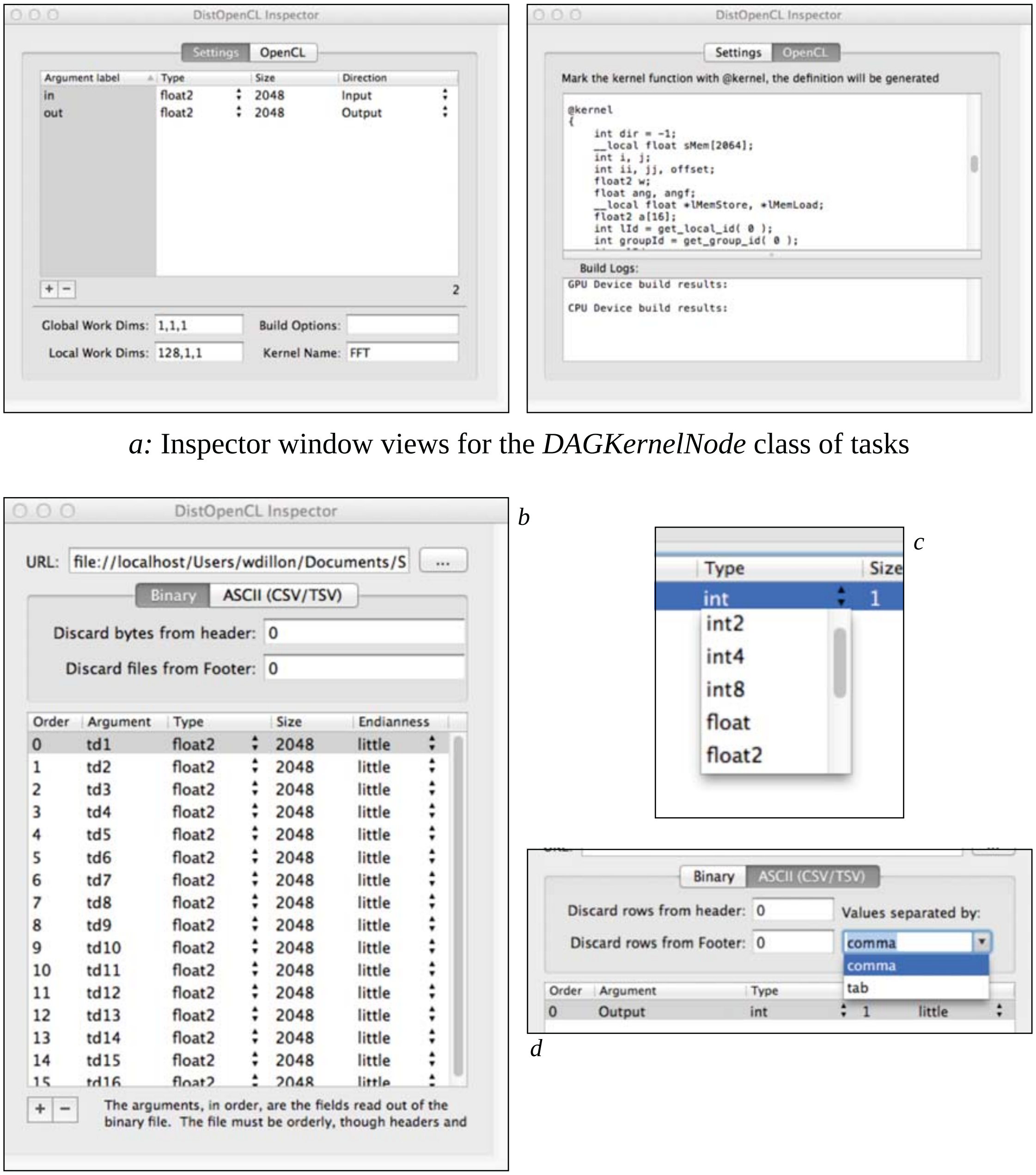
- Moving a node: Click-hold on the node, drag to the destination and release.
- Removing a node: Click on the node once to select, then press the “delete” key.
- Duplicating a node: Click on the node to select, then press the “command” and “d” keys simultaneously.
- Creating a connection: Click-hold on the source argument click target, drag the wire to the destination click target, and release.
- Moving a connection: Click-hold on the destination argument click target, drag wire to the new argument’s click target, and release.

- Removing a connection: The process is the same as moving a connection, but you release the mouse button away from an argument click target.

During wire connection operations, such as creation, movement and destruction, the selected wire is highlighted in green. The color of this highlighting could be used to provide the user constant feedback about the validity of proposed connections before an explicit action is performed. For example, the wire could be colored red while the terminus is not over a click target, yellow while over a click target that would result in an invalid connection (type mismatch or other problem) and green when the connection would succeed. The stoplight color scheme, while convenient for discussion, would ultimately be unsuitable if accessibility for individuals with color blindness is desired. A tooltip dialog that would provide diagnostic information to the user could be provided in cases where a connection attempt would fail. These enhancements are not included in the current implementation, and further usability analysis is a topic of future research.

Following the same model for advanced configuration as several of the graphical programming languages presented in the previous work, an external inspector window is provided. The contents of the inspector window are dynamically loaded and modified to reflect the selected node. Extensibility of the inspector interface was considered during the design phase, and the method for providing custom control is relatively straightforward. The *DAGNode* class provides a method to its subclasses, *loadIViewFromBundle*, which loads an interface specification file from disk. Subclasses can call this method while providing a filename for the interface. When the subclass is instantiated, the inspector view is loaded from disk. Samples of the inspector window contents for a sampling of *DAGNode* subclasses are presented in Figure 39.

The inspector interface allows users to add, remove, and edit arguments. When appropriate, the user may specify as many arguments as are needed. Editing any text field parameter, such as name and size, of the argument is accomplished by double-clicking on the text. The type field is a drop-down selection of every type that is supported. Endianness is also selected through a drop-down. The *DAGSourceFileNode* also provides the ability to select binary and ASCII file types.



Argument label	Type	Size	Direction
in	float2	2048	Input
out	float2	2048	Output

Global Work Dims: 1,1,1

Build Options:

Local Work Dims: 128,1,1

Kernel Name: FFT

Mark the kernel function with @kernel, the definition will be generated

```
@kernel
{
    int dir = -1;
    __local float sMem[2064];
    int i, j;
    int ii, jj, offset;
    float2 w;
    float ang, angf;
    __local float *lMemStore, *lMemLoad;
    float2 a[16];
    int lId = get_local_id( 0 );
    int groupId = get_group_id( 0 );
```

Build Logs:

```
GPU Device build results:

CPU Device build results:
```

a: Inspector window views for the *DAGKernelNode* class of tasks

URL: file://localhost/Users/wdillon/Documents/S

Discard bytes from header: 0

Discard files from Footer: 0

Order	Argument	Type	Size	Endianness
0	td1	float2	2048	little
1	td2	float2	2048	little
2	td3	float2	2048	little
3	td4	float2	2048	little
4	td5	float2	2048	little
5	td6	float2	2048	little
6	td7	float2	2048	little
7	td8	float2	2048	little
8	td9	float2	2048	little
9	td10	float2	2048	little
10	td11	float2	2048	little
11	td12	float2	2048	little
12	td13	float2	2048	little
13	td14	float2	2048	little
14	td15	float2	2048	little
15	td16	float2	2048	little

The arguments, in order, are the fields read out of the binary file. The file must be orderly, though headers and

Discard rows from header: 0

Discard rows from Footer: 0

Values separated by: comma

Order	Argument	Type		
0	Output	int	1	little

Figure 39: Inspector window examples; *DAGKernelNode* class (*a*), *DAGFileSourceNode* class (*b*). Detail views of the type selector (*c*) and the ASCII file delimiter selection (*d*).

The configuration parameters are unique to the file encoding method, so a tab view element is used to separate the distinct, but similar, configuration values. Images of each subview are provided in Figure 39; binary configuration is shown on the left (*b*), and ASCII configuration is on the right (*d*).

The *DAGKernelNode* inspector provides a view for configuring the arguments as well as kernel implementation. The OpenCL tab provides a text area to directly input OpenCL kernels, and the text in the bottom of the view contains the build results from the compilation of the kernel. When the user finishes editing, kernel compilation is immediately started and the results are updated. A macro is provided to automatically generate the kernel function definition. Because the name, directionality and type of every argument and kernel name must match exactly, the automatic creation of the kernel definition should simplify kernel development.

Graphical Programming Language Implementation

The implementation of the graphical programming language and user interface follows the model-view-controller strategy for user interface development. Figure 40 presents a simplified class model for the user interface logic in Distributed OpenCL. The *DAGModelView* and *DAGNodeView* classes (view) produce the on-screen representation and implement user interaction. Maintenance of the task graph structure and correctness is the responsibility of the *DAGModel* and *DAGNode* classes (model). Messages between these classes and the AppDelegate (controller) implement the majority of the application’s functionality.

The AppDelegate is a common class in Cocoa-based applications. Its purpose is to mediate application-level events, such as window creation and destruction, menu item selection and hotkey invocations. For the purposes of the graphical programming language implementation, it is responsible for all those events, as well as coordinating the inspector window contents when the window is created and when the selected node changes.

The apparently redundant functionality between the model and view classes -- for example, the *addNode* methods -- provides entry points to these actions from events originating from within the models or directly from the user. For example, when a node is duplicated, the majority of the processing occurs within the *DAGNode* class; the new instance is passed to the *DAGModel*, which informs the *DAGModelView* of the new *DAGNode* and *DAGNodeView*. If, however, the user selects a node and presses "delete," the *DAGModel* receives the event notification for that action. In this case, the message is passed to the *removeNode* method in the *DAGModel* class instance, removing the node.

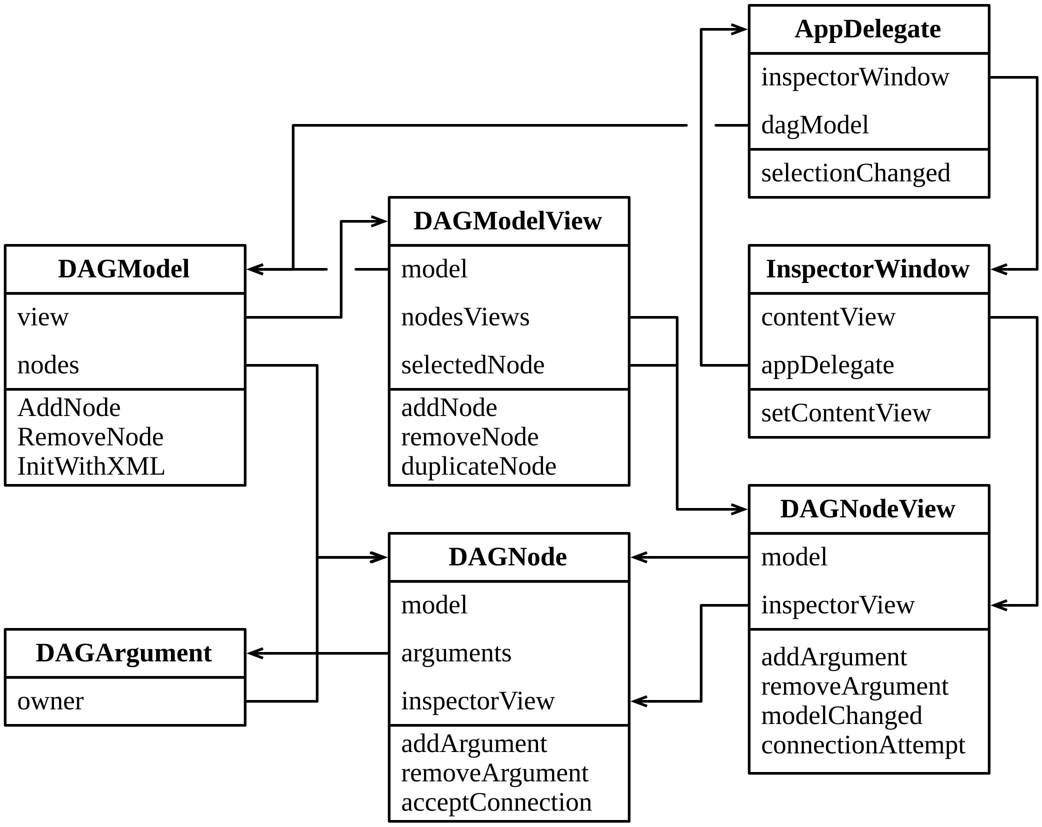


Figure 40: Simplified UML diagram of the user interface mechanics.

When the user draws a candidate for a connection between arguments, as described above, the *DAGModelView* class tracks the mouse movements while drawing the temporary wire. When the mouse button is released, the *DAGModelView* class iterates through the list of *DAGNodeView*s until the coordinates of the pointer are within the bounds of the candidate view. The coordinates are then transformed into the local coordinate space for the node view and tested for intersection with an argument click target. If a click target is selected, the argument represented with the target is sent an *acceptConnection* message with reference to the complementary argument. If the connection is deemed valid, the method returns a boolean true value, and the connection is committed.

Example problems solved with Distributed OpenCL

16 Channel Beamformer

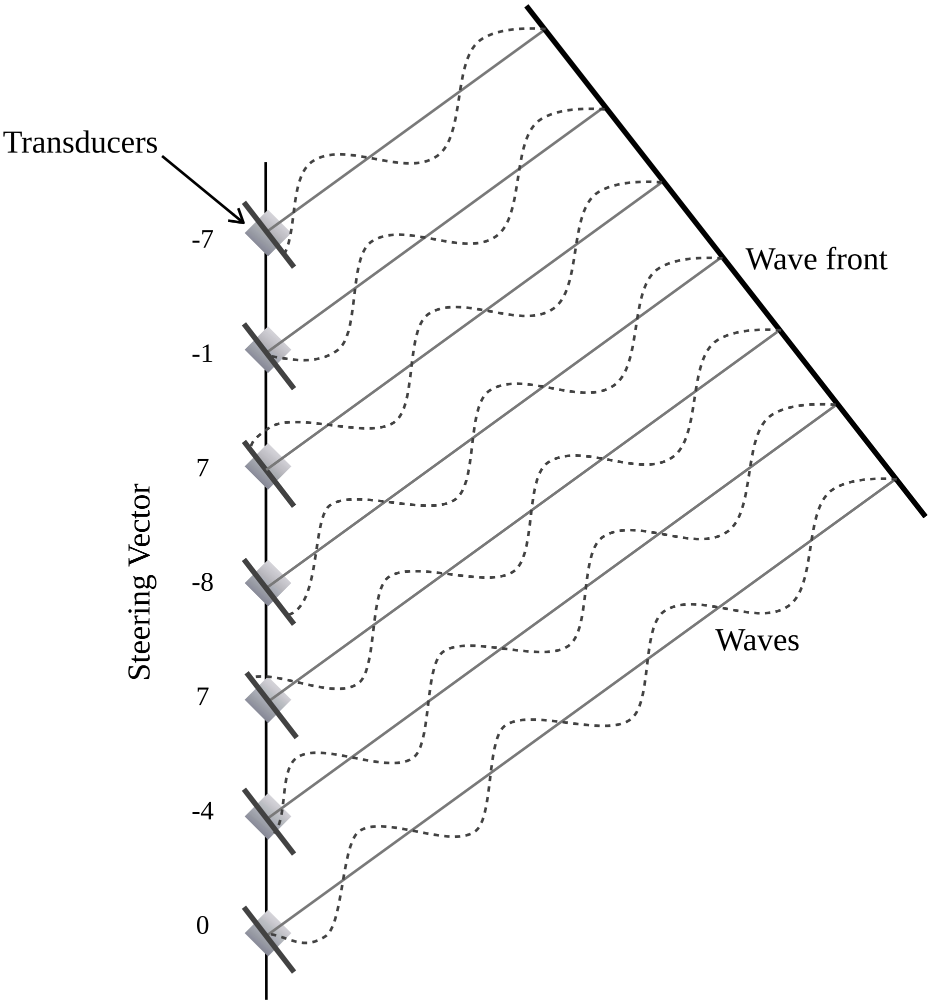


Figure 41: Diagram representing the basic functionality of a beamformer. A planar wave front being applied to an array of transducers is shown. The values of the steering vector are related to the phase of the wavefront at different points in space.

A beamformer is a system for shaping the spatial response of an array of transducers by either manipulating the phase or gain of each transducer (Figure 41). Beamforming can be used in receivers or transmitters, and can be applied to acoustic (Beng, Teck, and Potter 2002), electromagnetic (Curtis et al. 2003), or even seismic (Brind, Goddard, and Whitmarsh 1998) fields. Beamforming systems are commonly implemented using analog electronics, Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), or software running on general purpose computers. The algorithms used in beamforming can be very computationally expensive, especially when the output of every beam is desired; steering an array to a single beam is relatively cheap. Detailed technical information about beamforming is available (Manolakis et al. 2005).

When a beamformer is used without *a priori* information about the signal source and collects information for a broad set of beams, it is called a Conventional BeamFormer (CBF). This example problem was chosen because it has been implemented in three different ways within the context of this work. Also, the CBF algorithm has a variety of different types of parallelism. An FFT [37] is executed on each of the transducer channels, which is a relatively large granular scale of independent parallel work. Each FFT includes smaller scales of parallelism, requiring tight coordination. The results from the FFTs are combined into a matrix that is multiplied with the steering matrix. This is a single operation, but matrix multiplications have internally parallel components.

Exemplar code was provided as a Matlab script, which was first ported to a standard single-threaded application, written in C. Later, the implementation in C was converted to use the resources of the Cell/B.E. processor. Finally, the CBF was ported into the Distributed OpenCL task graph language. The relative duration of porting effort is instructive. Porting from Matlab to C took about a month of human time, and the majority of that time was spent learning the structure of the Matlab script. Matlab treats every variable as a matrix; scalars are simply matrices that are one unit in every dimension. Because every variable is a matrix, it is very easy to overlook their dimensionality. For example, what may appear as a single scalar multiplication may

[37] The FFT, or Fast Fourier Transform, is an efficient implementation of the Discrete Fourier Transform, which transforms input data from the time domain to the frequency domain.

actually be an operation between large matrices. This can have significant consequences for data flow analysis and typing.

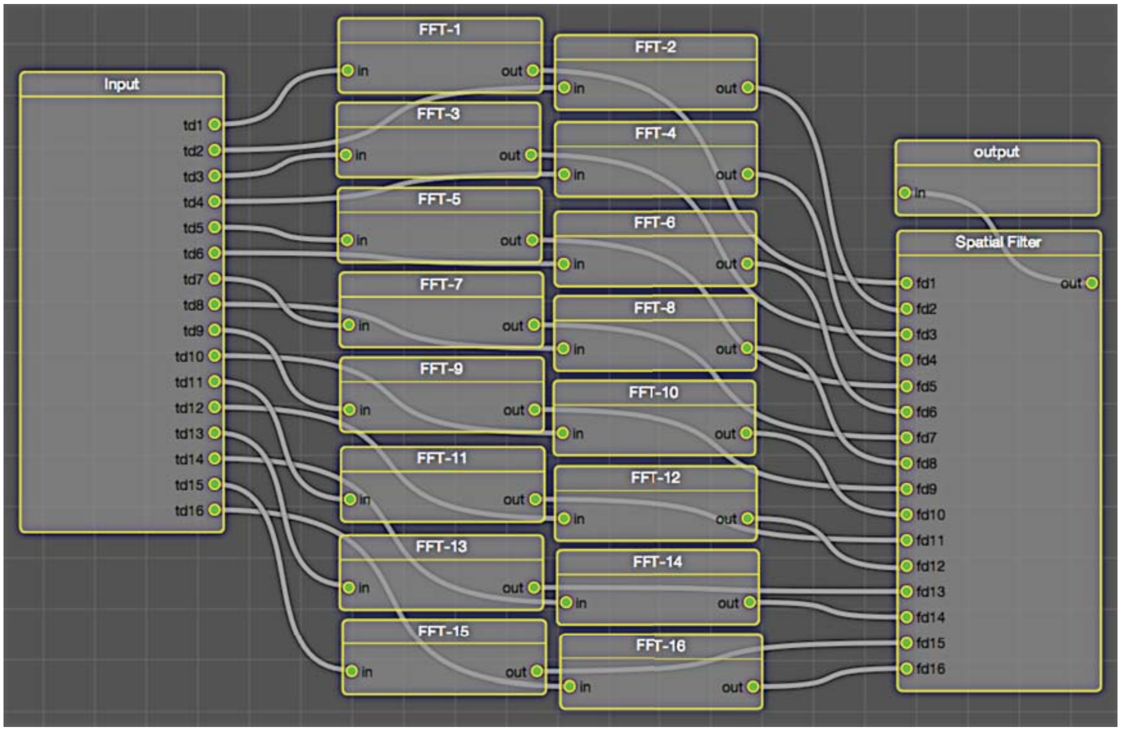


Figure 42: 16 Channel beamformer implemented in DistributedOpenCL

The C-based port of the CBF was used to develop a Cell/B.E. implementation. This work took several months. The reason for the increase in implementation time, though the algorithm was already well understood, is the complexity of the Cell/B.E. architecture. The Asymmetric MultiProcessing nature of the Cell/B.E. requires more management than typical architectures. It is the responsibility of the user application to schedule tasks on the SPE processors and explicitly load memory into the SPE's memory space. The memory operations are implemented as direct commands to the

DMA[38] system. Every memory transfer must be a multiple of 16 bytes and aligned to 16-byte boundaries.

The final implementation is shown in Figure 42. The task graph is a complete implementation of the CBF using Distributed OpenCL, and it took less than a week to produce. The input node is responsible for providing time-domain data for each of the 16 channels and 2048 sample packets. The set of 16 FFT nodes perform an out-of-place complex FFT, and each node's input argument specifies a 2048 element array of float values. The output arguments are 2048 element float2 values representing the complex-valued result. The spatial filter rearranges the output from the FFTs into a matrix, computes a set of steering vectors, multiplies the matrix with the steering vectors and normalizes the result. The output of the spatial filter is a 2D array that is represented as an image type and saved to disk.

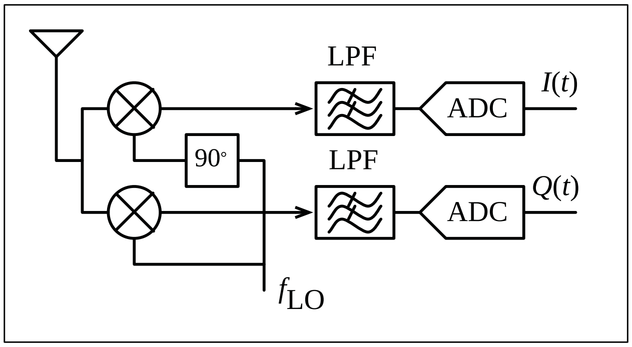
Software Defined Radio

Software Defined Radio (SDR) is a technique for processing and demodulating Radio Frequency (RF) signals. As the capabilities of processors have increased, it has become practical to move ever-greater amounts of the RF signal chain into software. Moving the processing chain into software allows greater flexibility when developing modulation standards and techniques.

The vast majority of SDR architectures are similar to the Direct DownConversion receiver depicted in top part of Figure 43. Radio signals enter the antenna on the left and are split before being fed into a pair of matched mixers. The mixers translate the input frequencies both up and down in frequency according to the Local Oscillator's (LO) frequency.

[38] Direct Memory Access is a system that allows memory transfers to occur without the direct coordination of the main processor. DMA transfers are initiated by providing the bounds of the desired transfer to the DMA processor along with a transfer command. The memory transfer occurs in parallel to other processing and completes asynchronously.

If a signal of 100 MHz is presented at the RF port, and 50 MHz is on the LO port, the output spectra will contain a copy of the input signal at 50 MHz and at 150 MHz (in addition to other signals, but those can be ignored for the purposes of this discussion). In the case of a DDC receiver, the LO frequency is set to match (or is very near) the desired RF frequency. In that case, one copy of the input signal is at (or near) 0 Hz, and the other is two times the LO. The higher frequency components are easily filtered out.



Direct DownConversion (DDC) Block Diagram

Commercial receiver (flipped along the vertical axis)

Figure 43: Sample block diagram of a DDC receiver and commercial device

The two mixers in a DDC receiver are fed with two copies of the LO, one of which is 90° out of phase. The mixers transform the phase of the input signal according to the phase of the LO. By providing copies of the LO that are out of phase, the output pair of signals has a complementary phase relationship: In-phase (I) and Quadrature (Q). These signals can be used as complex numbers. Using the properties of the complex plane, it is possible to isolate positive and negative spectra (on either side of 0 Hz), as well as demodulate signals such as Quadrature-Amplitude Modulation (QAM)[39].

The low-pass filtering also limits the bandwidth of the signal entering the Analog-Digital Converters (ADC). To avoid aliasing artifacts, it is vital to reduce the bandwidth of the input to an ADC to less than half the sample rate, in accordance with Nyquist-Shannon Sampling Theorem (Shannon 1948). The output of the ADCs is the quantized time domain values of the in-phase and quadrature signals. In the case of the commercial SDR receiver (bottom of Figure 43), these are 8-bit samples delivered at up to 3.2 million samples per second.

Figure 44 represents an example application of SDR using Distributed OpenCL. The raw samples are taken directly from the commercial SDR receiver and converted to floating point, which are output from the *Raw Input* task. This data is a 2 million sample per second stream of complex, floating point numbers. The FM carrier is shifted and filtered from anywhere within the two MHz input spectra to zero Hz using the *Frequency xlating FIR* task. This task contains a time-domain operation and uses the DOCL "scratchpad" feature to maintain filter state between blocks. Next, the *NB FM Demodulator* converts the centered and filtered signal from FM to baseband. This signal is tapped off and sent to the audio system of the client workstation for monitoring. It is also sent to a pair of band pass filters, one for 1200 Hz and the other for 2200 Hz. These tones are used to indicate the presence of either a 'one' or a 'zero' in the input stream. These tasks also perform an envelope detection operation, which outputs a DC value proportional to the amplitude of the filtered signal. The *Choose Max* task measures the relative strength of each filter's output and outputs a binary value accordingly. Finally, as each of these processes operate in the time domain at a

[39] QAM modulation uses both the phase and amplitude to super-impose layers of information onto a single "symbol." An analog form of this modulation was used to add color information to NTSC television. The phase and amplitude of the color carrier were used to encode hue and saturation, respectively {Richman:1954hh}.

sample rate much higher than the symbol rate of the input signal, it is useful to discard redundant data in the *decimation* block. This output is the final, demodulated digital data.

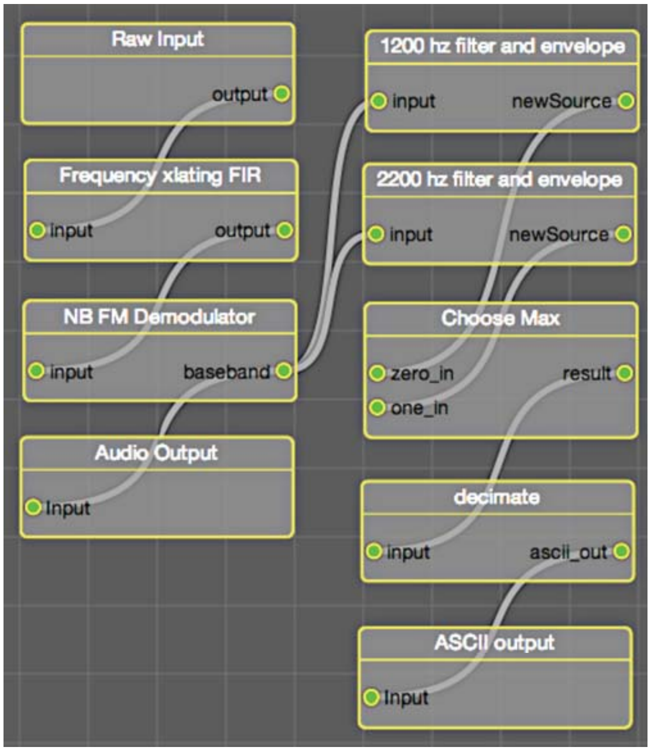


Figure 44: SDR task graph for Bell 202[40] demodulator over narrowband FM.

Distributed OpenCL is a useful tool for implementing a vast array of signal processing tasks, as well as numerical modeling. The bounds of what can be represented using this tool have yet to be fully explored; surely, it will be a challenge to identify applications that cannot be defined within this framework. Furthermore, developing applications using DOCL is easy and rewarding. The tedious, and thankless, glue code that consumes time and mental energy is produced automatically. Users are left to think about their own application, which they probably enjoy.

[40] The Bell 202 modulation scheme was used in 600/1200 baud modems, and is substantially similar to V.23, http://www.itu.int/rec/T-REC-V.23-198811-I/en, accessed May 14, 2012.

Conclusions

As the computing market changes by moving away from expensive, specialized tools in favor of cheap commodity products, new strategies for working must be explored. During the period of rapidly increasing clock speeds, improving the performance and capacity of existing scientific models could be as simple as buying a new computer. Today, as processor clock speeds are mostly constant, the only remaining strategy is to embrace greater parallelism. Often, this means that existing applications need to be rewritten, or at least modified. It is an expensive proposition to rewrite these applications, and with shrinking science budgets, it isn't clear whether the necessary funding exists. There is a distinct opportunity and need for tools that allow the exploitation of cheap and ubiquitous commodity hardware. Giving domain scientists the tools to develop new models that can use the newest technology without crushing their budgets will pave the way for novel research.

Distributed OpenCL (DOCL) addresses these issues. The overarching goal was to mitigate or eliminate every source of frustration for the user while enabling distributed computation on an ad hoc cluster of heterogeneous nodes. Manual cluster creation, configuration, and use were eliminated. Scheduling cost metrics, such as communication and computation cost, were automatically collected. The graphical programming language reduces the complexity of model development, eliminating the need for boilerplate[41] code. Several example applications were implemented using DOCL.

To remain accessible to non-expert computer users, it was necessary to ensure that Distributed OpenCL was easy to install. The initial implementation, which runs on MacOS, is contained within a single application bundle. It appears to the user as a single file and can be installed by dragging and dropping it into the Applications folder. Building a cluster is as easy as installing the application on several computers

[41] Boilerplate code is necessary, but tedious, code that configures the operating system and its APIs, but isn't relevant to the actual task at hand.

and running it. The careful engineering that went into the design of DOCL allows a novice computer user to construct a high-performance computing cluster. DOCL even eliminated the need for superuser privileges; the system can be installed and used without any administrator passwords, which reduces the risk associated with its use.

Unlike typical supercomputer and cluster systems, ad hoc clusters are inherently chaotic and diverse. To make well-informed decisions, the task scheduler requires high-quality information about the time required to transport data across the network, as well as the time required to perform a task on each device. To address this need, two solutions were developed that automatically measure these quantities. In keeping with the overarching project goals, the cost metric measurement systems had to be configuration-free and automatic; this was achieved through the development of novel distributed algorithms. The throughput benchmark scheduling system performed within the bounds set by theory.

Parallel programming has always had a reputation for being a particularly painful activity. This perception appears well founded when using common parallel programming paradigms such as OpenMP, MPI, and Pthreads. These technologies are based on serial programming languages, but they allow multiple instances of the same program run coincidently but independently. Assuring consistent, well-behaved access to system resources is a significant challenge, and it is the root of the majority of errors. Graph-based parallel programming, in contrast, allows parallel execution of independent operations in an inherently safe way. Many of the parallel programming bugs that are common using traditional techniques are not possible using a task graph.

Many task graph implementations operate on a highly granular scale. From a research perspective, this may be an appropriate course of action. From a practical standpoint, however, overly detailed task graphs are visually confusing to the user and are difficult to efficiently translate into a machine executable representation. A compromise was struck between the ease of use of a task graph and the efficiency of traditional, C-based coding. Using OpenCL as the inner implementation of tasks allows efficient compilation and execution across a wide variety of processor types, including data-parallel devices such as GPUs. The outer representation of the task graph summarizes the architecture of the application and allows for automatic parallelization.

The example applications provided demonstrate the power of Distributed OpenCL and its ability to implement complex signal processing algorithms. The beamforming application is an important example of the improvement in the use of human time. The original implementation of the algorithm (from exemplar Matlab source code) on the Cell/B.E. took several months of human time and required the development of a task-graph based library. Using DOCL, the algorithm was developed in a few days and is able to run on a cluster.

I believe that Distributed OpenCL has been a positive exploration of the problems and opportunities encountered while developing a next-generation cluster programming system. Solutions were found that eliminate the complex manual tasks required to construct a traditional cluster, and an effective programming environment was developed and proven to be useful for scientific tasks.

Future Work

Because Distributed OpenCL is a platform rather than an application, there is tremendous opportunity for future enhancement and development. The current implementation relies upon statically linked implementations of the source and sink nodes in the task graph. These nodes are vital for providing input data into the graph as well as performing useful work with the result. In its current form, only a subset of the range of options is included. Part of the strategy for ensuring the greatest impact from this work is supporting unforeseen combinations of input, output, and computation. New avenues of research often begin when existing ideas are combined in new and interesting ways.

To ensure that the greatest diversity in input and output device support is available, a plugin architecture is proposed. Because the inspector window used to configure task nodes is populated at runtime, it is possible to dynamically add additional functionality. Furthermore, because the XML document file specification identifies the node types by their string type name, through introspection, it is also possible to locate class implementations at runtime, allowing for the addition of functionality after compilation and deployment.

In addition to the plugin system proposed for sink and source nodes, it is possible to support task scheduler plugins. The wide range of task scheduling strategies, and their profound effect on performance, requires flexibility within the scheduling system. The provided task scheduler is a basic example intended to demonstrate the scheduling framework. Future work is required to develop a suite of more advanced and powerful schedulers. It is clear that task scheduling algorithm development is a sufficiently broad and deep area whose exploration wasn’t possible in this work; the most appropriate course of action was to provide the mechanics for later development and research.

While the system that generates communication and computation cost performs well, there are enhancements that could accelerate the execution of these measurements. Currently, the network cost measurement system performs a measurement between each pair of hosts. This is the most complete way to determine a communication cost matrix for a cluster. In many cases, however, some of these measurements are redundant. It should be sufficient to model the throughput capability of the network itself, then measure each host’s performance relative to the network. Using this method, the throughput between any pair of hosts is equal to the minimum throughput of each host to the network and of the network itself. The direct measurement of network throughput is not possible because it is always measured through a host, but it can be assumed to be greater than or equal to the highest throughput between any pair of hosts. It should be possible to reduce the number of operations in the communication cost measurement algorithm from $\theta\left(\frac{(n\text{-}1)n}{2}\right)$[42] to better than$O(n \log n)$.

Not only can the number of operations necessary for communication cost measurement be reduced, but many of the operations used in compute cost measurement can be culled. In the case of computation cost, the redundant operations are in the form of additional benchmarks on similar devices. The current system performs a benchmark on every device within the cluster. It is likely that distinct instances of the same device are similar enough that only one measurement is required for the entire class. There is some risk to this optimization, however; namely, that different systems' configurations might affect the performance of the same device in differing hosts. The sensitivity of device performance to host configuration and other tradeoffs would have to be investigated before this relatively simple modification could be made.

There are opportunities for several other enhancements, such as real-time debugging of the task graph. It should be possible to allow the user to interrogate each of the intermediate values as they're passed between nodes in the task graph. A small set of inspection tools would provide a wealth of valuable information to the user. An oscilloscope-like graphing tool could be used to inspect scalar or vector arguments across time, or to graph a vector argument for each task invocation. The same construct could be used after the source data is processed by an FFT. Image and 2D data could be shown by directly mapping the data to the screen or presenting the data in a table. More complex data, like volumetric and 3D data, could be displayed using a raytracing visualizer. Some additional tools, such as triggering, breakpoints, or single-stepping, would have to be provided due to the volume of data that could pass

[42] A θ bound is a tight bound, which is similar to an O bound, except that it is bounded both above and below asymptotically.

through the task graph. These tools would allow the user to make sense of unexpected results and encourage iterative design.

The use of MacOS as a development platform contributed to the rapid implementation of Distributed OpenCL. MacOS contains the canonical implementations of Bonjour and OpenCL; therefore, the tools provided were available earlier and were arguably of higher quality. However, the choices that Apple Inc. made regarding the discontinuation of their rack-mountable server (Xserve), as well as the stagnation of their workstation (MacPro) encourage skepticism regarding their continued investment in the professional product market segment. Furthermore, as GPU vendors have added products that are intended for scientists and that work best in Linux environments, the development of a Linux version of Distributed OpenCL would be beneficial. Only the cluster node agent software would need to be developed initially. The user interface paradigms were designed with the MacOS use model and Cocoa UI libraries in mind; therefore, transitioning these elements would require more careful planning and a complete reimplementation. There are no current plans to port the software to Microsoft Windows, but because the specification is open and relies upon standard technologies, it should be a straightforward exercise.

The production of this dissertation, and its presentation to the committee, do not mark the termination of this work; those are merely signs that the basic foundation has been laid and the associated research is complete. The presented project is capable of performing useful work in a wide range of scientific and data analysis scenarios, and these capabilities will be further enhanced. The described future work is a narrow cross-section of the enormous scope of future enhancements. Many years of work remain, but the end result will be nothing short of amazing.

Bibliography

Abbes, Heithem, and Jean-Christophe Dubacq. 2009. “Analysis of Peer-to-Peer Protocols Performance for Establishing a Decentralized Desktop Grid Middleware.” *Euro-Par 2008 Workshops-Parallel Processing*. doi: 10.1007/978-3-642-00955-6_28.

Aggarwal, G, R Motwani, D Shah, and An Zhu. 2003. “Switch Scheduling via Randomized Edge Coloring.” In, 502–512. doi:10.1109/SFCS. 2003.1238223.

Apon, A, R Buyya, H Jin, and J Mache. 2001. “Cluster Computing in the Classroom: Topics, Guidelines, and Experiences.” In, 476.

BanerJee, Uptal, Rudolf Eigenmann, Alexandru Nicolau, and David A Padua. 1993. “Automatic Program Parallelization.” *Proceedings of the IEEE* 81 (2) (February 5): 211.

Bansal, Samta, Juan C Rey, Andrew Yang, Myung-Soo Jang, LC Lu, Philippe Magarshack, Marchal Pol, and Riko Radojcic. 2010. “3-D Stacked Die: Now or Future?.” In. ACM. doi:10.1145/1837274.1837350.

Bar-Noy, Amotz, Rajeev Motwani, and Joseph Naor. 1992. “The Greedy Algorithm Is Optimal for on-Line Edge Coloring.” *Information Processing Letters* 44 (5) (December): 251–253. doi:10.1016/0020-0190(92)90209-E.

Beng, Koay Teong, Tan Eng Teck, and J.R Potter. 2002. “A Portable, Self-Contained, 5MSa/S Data Acquisition System for Broadband, High Frequency Acoustic Beamforming.” *Oceans '02 Mts/Ieee* 1: 369–378. doi:10.1109/ OCEANS.2002.1193300.

Bolz, Jeff, Ian Farmer, Eitan Grinspun, Peter Schröoder, Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. 2003. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. *ACM Transactions on Graphics (TOG)*. Vol. 22. Conjugate Gradients and Multigrid. New York, New York, USA: ACM. doi:10.1145/882262.882364.

Y. Breitbart, M. Garofalakis, B. Jai, C. Martin, R. Rastogi, A. Silberschatz, Topology discovery in heterogeneous IP networks: the NetInventory system, Networking, IEEE/ACM Transactions on. 12 (2004) 401–414.

Brind, R.J, N.J Goddard, and R.B Whitmarsh. 1998. “Beamforming Performance of an Array of Sea-Bed Geophone Sensors.” In, 2:687–693. doi:10.1109/OCEANS.1998.724326.

Buttari, Alfredo, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, and George Bosilca. 2007. “A Rough Guide to Scientific Computing on the Playstation 3” (May 11).

Chow, Alex Chunghen, Gordon C. Fossum, and Daniel A. Brokenshire. 2005. “A Programming Example: Large FFT on the Cell BE” (September 21).

Curtis, D.D, R.W Thomas, W.J Payne, W.H Weedon, and M.A Deaett. 2003. “32-Channel X-Band Digital Beamforming Plug-and-Play Receive Array.” In, 205–210. doi:10.1109/PAST.2003.1256982.

Czajkowski, Karl, Ian Foster, and Carl Kesselman. 1999. “Resource Co-Allocation in Computational Grids.” *Proc. 10th IEEE International Symposium on High- Performance Distributed Computing.*

Duda, Richard O., Peter E Hart, and David G. Stork. 2001. *Pattern Classification*. Second. John Willey & Sons.

Duffy, K R, N. O. O'Connell, and A Sapozhnikov. 2008. “Complexity Analysis of a Decentralised Graph Colouring Algorithm.” *Information Processing Letters* 107 (2) (July 16): 60–63. doi:10.1016/j.ipl.2008.01.002.

Fatahalian, K, D Horn, T Knight, and L Leem. 2006. “Sequoia: Programming the Memory Hierarchy.” *Supercomputing ’06*.

Francis, Tom. 2010. “Gabe Newell: Next-Gen Game Engines Will Be Ten Times Harder.” *Pcgamer.com*. http://www.pcgamer.com/2010/09/13/gabe-newell-next-gen-game-engines-will-be-ten-times-harder/.

Garey, M.R., D. S. Johnson, and Ravi Sethi. 1976. “The Complexity of Flowshop and Jobshop Scheduling.” *Mathematics of Operations Research* 1 (2) (May): 117–129.

Gilliam, D. 1993. *The Supercomputer Industry Development, Government Involvement, and Implications for the Future*. *Ndu.Edu*. The Industrial College of the Armed Forces, National Defense University.

Gropp, William, and Ewing Lusk. 1993. “The MPI Communication Library: Its Design and a Portable Implementation.” *Scalable Parallel Libraries Conference* (October): 160–165. doi:10.1109/SPLC.1993.365571.

Guttman, Erik. 2001. “Autoconfiguration for IP Networking: Enabling Local Communication.” *Internet Computing* 5 (3) (June): 81–86. doi: 10.1109/4236.935181.

Harris, MJ. 2003. “Real-Time Cloud Simulation and Rendering.”

Holyer, I. 1981. “The NP-Completeness of Edge-Colouring.” *Siam Journal of Computing*.

Iamnitchi, A, I Foster, and D Nurmi. 2002. “A Peer-to-Peer Approach to Resource Discovery in Grid Environments.” *IEEE High Performance Distributed Computing*.

Joseph, Earl, and Michael Shirer. 2012. “HPC Server Market Delivers Record Revenues and 8.4% Growth in 2011.” *Idc.com*.

Kiehl, J, J Hack, G Bonan, and B Boville. 1998. “The National Center for Atmospheric Research Community Climate Model: CCM3*.” *Journal of Climate* 11 (6) (June): 1131–1149. doi: 10.1175/1520-0442(1998)011<1131:TNCFAR>2.0.CO;2.

Kim, SJ, and JC Browne. 1988. “A General Approach to Mapping of Parallel Computation Upon Multiprocessor Architectures.” *International Conference on Parallel Processing* 3 (August 3): 1–9.

Lee, S, S Min, and R Eigenmann. 2009. “OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization.” *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (February 1).

Leith, D.J, and P Clifford. 2006. “A Self-Managed Distributed Channel Selection Algorithm for WLANs.” *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks, 2006 4th International Symposium on*: 1–9. doi:10.1109/WIOPT.2006.1666484.

Lindholm, E, J Nickolls, S Oberman, and J Montrym. 2008. “NVIDIA Tesla: a Unified Graphics and Computing Architecture.” *Micro, IEEE* 28 (2) (March 1): 39–55. doi:10.1109/MM.2008.31.

Llamas, Ramone, Kevin Restivo, and Michael Shirer. 2012. “Smartphone Market Hits All-Time Quarterly High Due to Seasonal Strength and Wider Variety of Offerings.” *Idc.com*.

Lucas, Bruce, Gregory D. Abram, Nancy S. Collins, David A. Epstein, Donna L. Gresh, and Kevin P. McAuliffe. 1992. *An Architecture for a Scientific Visualization System*. IEEE Computer Society Press.

Manolakis, Dimitris G, Dimitris Manolakis, Vinay K Ingle, and Stephen M Kogon. 2005. *Statistical and Adaptive Signal Processing: Spectral Estimation, Signal Modeling, Adaptive Filtering and Array Processing (Artech House Signal Processing Library)*. Artech House Print on Demand.

Mastroianni, C, D Talia, O Verta, and C ICAR. 2005. “A P2P Approach for Membership Management and Resource Discovery in Grids.” *Information Technology: Coding and Computing*.

Mastroianni, Carlo, Domenico Talia, and Oreste Verta. 2005. “A Super-Peer Model for Building Resource Discovery Services in Grids: Design and Simulation Analysis.” Ed. Sloot et al. *Lecture Notes in Computer Science*: 132–143.

McCool, M. 2008. “Scalable Programming Models for Massively Multicore Processors.” *Proceedings of the IEEE* 96 (5) (May 1): 816–831. doi: 10.1109/JPROC.2008.917731.

McCool, M, and R Inc. 2006. “Data-Parallel Programming on the Cell BE and the GPU Using the RapidMind Development Platform.” *GSPx Multicore Applications Conference*.

Moore, GE. 1975. “IEEE Xplore - Abstract Page.” *Electron Devices Meeting*.

Moore, Gordon E. 1965. “Cramming More Components Onto Integrated Circuits.” *Electronics Magazine*. http://ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.

Moore, Gordon E. 1995. “Proceedings of SPIE.” In, 2438:2–17. SPIE. doi: 10.1117/12.210341.

Moreland, Kenneth, and Edward Angel. 2003. *The FFT on a GPU*. Eurographics Association.

Ripeanu, Matei, Adriana Lamnitchi, and Ian Foster. 2002. “Mapping the Gnutella Network.” *IEEE Internet Computing* 6 (1) (January).

Robbins, Kay A., and Steven Robbins. 1995. *Practical UNIX Programming: a Guide to Concurrency, Communication, and Multithreading*. Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithre.

Shalf, John, Jon Bashor, Dave Patterson, Krste Asanovic, Katherine Yelick, Kurt Keutzer, and Tim Mattson. 2009. “MULTICORE COMPUTING: the Manycore Revolution: Will HPC Lead or Follow?.” *Scidacreview.org*. SciDAC Review 14. http://www.scidacreview.org/0904/html/multicore.html.

Shannon, CE. 1948. “A Mathematical Theory of Communication.” *Bell System Technical Journal* 27 (July 28): 379–423, 623–656.

Shchepetkin, Alexander F., and James C. McWilliams. 2005. “The Regional Oceanic Modeling System (ROMS): a Split-Explicit, Free-Surface, Topography-Following-Coordinate Oceanic Model.” *Ocean Modelling*. doi:10.1016/j.ocemod.2004.08.002.

Sider, Abderrahmane, and Raphaël Couturier. 2008. “Fast Load Balancing with the Most to Least Loaded Policy in Dynamic Networks.” *The Journal of Supercomputing* 49 (3) (October 9): 291–317. doi:10.1007/s11227-008-0238-5.

Steinberg, Daniel, and Stuart Cheshire. 2005. *Zero Configuration Networking: the Definitive Guide*. 1st ed. O'Reilly Media.

Stevens, W. Richard. 1990. *UNIX Network Programming*. 1st ed. Prentice Hall.

Stevens, WR, and B Fenner. 2004. *UNIX Network Programming: the Sockets Networking API*. Addison-Wesley Professional.

Topcuoglu, H, S Hariri, and M Wu. 2002. “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing.” *Ieee Transactions on Parallel and Distributed Systems*.

Uppuluri, Prem, Narendranadh Jabisetti, Uday Joshi, and Yugyung Lee. 2005. “P2P Grid: Service Oriented Framework for Distributed Resource Management.” *Services Computing*.

Van der Maar, S, and K Batenburg. 2009. “Experiences with Cell-BE and GPU for Tomography.” *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modelling, and Simulation*: 298–307.

Wang, Z, and MFP O'Boyle. 2009. “Mapping Parallelism to Multi-Cores: a Machine Learning Based Approach.” *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (February 1).

Ward, Lewis, and Michael Shirer. 2012. “Nintendo 3DS and Sony PlayStation Vita Poised to ‘Make Some Noise’ in Gaming in 2012.” *Idc.com*.

Wright, Gary R., and W. Richard Stevens. 1995. *TCP/IP Illustrated, Vol. 2: the Implementation*. 1st ed. Addison-Wesley Professional.

Wu, M, and D Gajski. 1990. “Hypertool: a Programming Aid for Message-Passing Systems.” *Parallel and Distributed Systems, IEEE Transactions on* 1 (3) (July 1): 330–343. doi:10.1109/71.80160.

Zhang, Dongyan, Chao Zheng, Hongli Zhang, and Hongliang Yu. 2010. “Identification and Analysis of Skype Peer-to-Peer Traffic.” In, 200–206. doi:10.1109/ICIW.2010.36.

Appendices

Appendix A. XML Messaging Schema

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://tundra/dist-opencl/peerSchema.xsd"
  xmlns="http://tundra.oce.orst.edu/dist-opencl/peer">

  <!-- Messages relating to DOCL system tree -->
  <xs:attributeGroup name="platform.attributes">
    <xs:attribute name="hw_vendor" type="xs:string"/>
    <xs:attribute name="hw_model" type="xs:string"/>
    <xs:attribute name="sw_vendor" type="xs:string"/>
    <xs:attribute name="sw_version" type="xs:string"/>
  </xs:attributeGroup>

  <xs:complexType name="opencl_device">
    <xs:attributeGroup ref="platform.attributes"/>
    <xs:attribute name="type" use="required"/>
    <xs:attribute name="units" type="xs:positiveInteger"/>
    <xs:attribute name="frequency" type="xs:positiveInteger"/>
    <xs:attribute name="max_workitem_size" type="xs:string"/>
    <xs:attribute name="max_workgroup_size" type="xs:string"/>
    <xs:attribute name="max_image2d_size" type="xs:string"/>
    <xs:attribute name="max_image3d_size" type="xs:string"/>
    <xs:attribute name="global_mem_size" type="xs:decimal"/>
    <xs:attribute name="local_mem_size" type="xs:decimal"/>
  </xs:complexType>

  <xs:complexType name="opencl">
    <xs:sequence>
      <xs:element ref="opencl_device"
        maxOccurs="unbounded" minOccurs="1"/>
    </xs:sequence>
    <xs:attributeGroup ref="platform.attributes"/>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="profile" type="xs:string"/>
    <xs:attribute name="type" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="service">
    <xs:attribute name="address" type="xs:string"/>
    <xs:attribute name="netmask" type="xs:string"/>
    <xs:attribute name="config_method" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="interface">
    <xs:sequence>
      <xs:element ref="service"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
```

```
<xs:complexType name="network">
  <xs:sequence>
    <xs:element ref="interface"
      maxOccurs="unbounded" minOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="system">
  <xs:sequence>
    <xs:element ref="opencl"/>
    <xs:element ref="network"/>
    <xs:element ref="peers"/>
  </xs:sequence>
  <xs:attributeGroup ref="platform.attributes"/>
  <xs:attribute name="timestamp" type="xs:positiveInteger"/>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="uuid" type="xs:string"/>
</xs:complexType>

<xs:complexType name="update">
  <xs:sequence>
    <xs:any namespace="http://tundra.oce.orst.edu/dist-opencl/peer"
      processContents="skip"
      minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="Xpath" type="xs:string"/>
</xs:complexType>

<!-- Messages relating to DOCL benchmarks -->
<xs:attributeGroup name="statistics.attributes">
  <xs:attribute name="min"     type="xs:float"/>
  <xs:attribute name="max"     type="xs:float"/>
  <xs:attribute name="st_dev"  type="xs:float"/>
  <xs:attribute name="samples" type="xs:integer"/>
</xs:attributeGroup>

<xs:complexType name="mean">
  <xs:simpleContent>
    <xs:extension base="xs:float">
      <xs:attributeGroup ref="statistics.attributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:element name="error" type="xs:string"/>

<xs:complexType name="device">
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:element ref="mean"/>
    <xs:element ref="error"/>
  </xs:choice>
  <xs:attribute name="type" type="xs:string"/>
  <xs:attribute name="model" type="xs:string"/>
  <xs:attribute name="index" type="xs:positiveInteger"/>
</xs:complexType>
```

```
  <xs:complexType name="kernel">
    <xs:sequence>
      <xs:element ref="device"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="latency">
    <xs:simpleContent>
      <xs:extension base="xs:float">
        <xs:attributeGroup ref="statistics.attributes"/>
        <xs:attribute name="packet_loss" type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:complexType name="throughput">
    <xs:simpleContent>
      <xs:extension base="xs:float">
        <xs:attributeGroup ref="statistics.attributes"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:complexType name="bench">
    <xs:choice minOccurs="1" maxOccurs="2">
      <xs:element ref="latency"/>
      <xs:element ref="throughput"/>
    </xs:choice>
    <xs:attribute name="local_address" type="xs:string"/>
  <!-- Address of the host sending the benchmark message,
      or address of the localhost (in peerSubtree)-->
    <xs:attribute name="remote_address" type="xs:string"/>
  <!-- Address of the host receiving the benchmark message,
      or address of the remote host (in peerSubtree)-->
    <xs:attribute name="netmask" type="xs:string"/>
  </xs:complexType>

<!-- Messages relating to DOCL peering -->
  <xs:complexType name="peer">
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="uuid" type="xs:string"/>
    <xs:attribute name="client" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="peers">
    <xs:sequence>
      <xs:element ref="peer" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

<!-- Messages relating to DOCL diagnostics -->
  <xs:complexType name="event">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="timestamp" type="xs:string"/>
```

```
          <xs:attribute name="severity"  type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>

    <xs:complexType name="log">
      <xs:sequence>
        <xs:element ref="event" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="timestamp" type="xs:string"/>
    </xs:complexType>

    <xs:complexType name="logs">
      <xs:sequence>
        <xs:element ref="log" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="host" type="xs:string"/>
    </xs:complexType>

  <!-- Messages relating to DOCL preperation and execution -->
    <xs:complexType name="DOCL-node">
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:any namespace="http://tundra.oce.orst.edu/dist-opencl/doc"
          minOccurs="1" maxOccurs="1"/>
      </xs:choice>
    </xs:complexType>

    <xs:complexType name="prepare">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:any namespace="http://tundra.oce.orst.edu/dist-opencl/doc"
          minOccurs="1" maxOccurs="1"/>
      </xs:choice>
      <xs:attribute name="port" type="xs:integer"/>
    </xs:complexType>

    <xs:complexType name="run">
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:any namespace="http://tundra.oce.orst.edu/dist-opencl/doc"
          minOccurs="1" maxOccurs="1"/>
      </xs:choice>
    </xs:complexType>

    <xs:complexType name="stop">
      <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:any namespace="http://tundra.oce.orst.edu/dist-opencl/doc"
          minOccurs="1" maxOccurs="1"/>
      </xs:choice>
    </xs:complexType>

  <!-- Messages relating to encapsulation -->
    <xs:element name="message">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="update"/>
        <xs:element ref="peer"/>
        <xs:element ref="bench"/>
        <xs:element ref="prepare"/>
        <xs:element ref="run"/>
        <xs:element ref="stop"/>
        <xs:any
          namespace="http://tundra.oce.orst.edu/dist-opencl/doc"/>
        </xs:choice>
      </xs:complexType>
    </xs:element>

  </xs:schema>
```

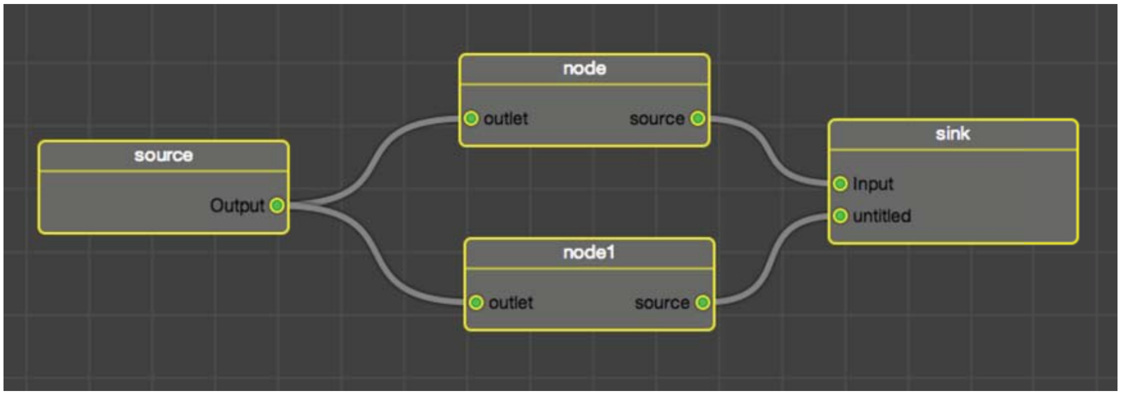
Appendix B. Example Peer XML System Tree

```
<?xml version="1.0" encoding="UTF-8"?>

<system hw_model="MacPro4,1" hw_vendor="Apple" sw_version="85fd753"
     uuid="F184D397-BF40-5097-BF78-53C7F84BEDDA" name="Tundra2">
  <opencl vendor="Apple" version="OpenCL 1.1"
      name="Apple" profile="FULL_PROFILE">
    <opencl_device type="CPU" vendor="Intel" units="16"
        frequency="2659" max_workitem_size="1024,1,1"
        max_workgroup_size="16" max_image2d_size="8192,8192"
        max_image3d_size="2048,2048,2048" local_mem_size="32768"
        global_mem_size="6442450944">
    </opencl_device>
    <opencl_device type="GPU" vendor="NVIDIA" units="30"
        frequency="1476" max_workitem_size="512,512,64"
        max_workgroup_size="30" max_image2d_size="4096,4096"
        max_image3d_size="2048,2048,2048" local_mem_size="16384"
        global_mem_size="1073741824">
    </opencl_device>
    <opencl_device type="GPU" vendor="NVIDIA" units="4"
        frequency="1400" max_workitem_size="512,512,64"
        max_workgroup_size="4" max_image2d_size="4096,4096"
        max_image3d_size="2048,2048,2048"
        global_mem_size="536870912" local_mem_size="16384">
    </opencl_device>
  </opencl>

  <network>
    <interface name="en2">
      <service address="172.20.71.212" netmask="255.255.240.0"
          config_method="DHCP"/>
    </interface>
    <interface name="en0">
      <service address="128.193.71.212" netmask="255.255.248.0"
          config_method="DHCP"/>
    </interface>
  </network>

  <peers>
    <peer name="Tundra1"uuid="97612927-06A2-5876-BF6E-246E0AA72D89">
      <bench local_address="172.20.71.212"
            remote_address="172.20.71.211">
        <latency std_deviation="0.000115" samples="11"
            packet_loss="0.0">0.000186</latency>
        <throughput>549941888.0</throughput>
      </bench>
      <bench local_address="128.193.71.212"
            remote_address="128.193.71.211">
        <latency std_deviation="0.000103" samples="11"
            packet_loss="0.0">0.000300</latency>
        <throughput>333777024.0</throughput>
      </bench>
    </peer>
    <peer name="Tundra3" uuid="E8443EE8-F015-5988-AE4E-0AC1B825715C">
      <bench local_address="172.20.71.212"
            remote_address="172.20.71.213">
        <latency std_deviation="0.000161" samples="11"
            packet_loss="0.0">0.000173</latency>
      </bench>
      <bench local_address="128.193.71.212"
            remote_address="128.193.71.213">
        <latency std_deviation="0.000218" samples="11"
            packet_loss="0.0">0.000338</latency>
      </bench>
    </peer>
  </peers>
</system>
```

Appendix C: XML Document Schema

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://tundra/dist-opencl/docSchema.xsd"
    xmlns="http://tundra.oce.orst.edu/dist-opencl/doc">

  <xs:attributeGroup name="execution.attributes">
    <xs:attribute name="uuid" type="xs:string"/>
    <xs:attribute name="peer" type="xs:string"/>
    <xs:attribute name="port" type="xs:decimal"/>
  </xs:attributeGroup>

  <xs:attributeGroup name="argument.attributes">
    <xs:attribute name="label"      type="xs:string"/>
    <xs:attribute name="type"       type="xs:string"/>
    <xs:attribute name="size"       type="xs:string"/>
    <xs:attribute name="direction"  type="xs:string"/>
    <xs:attribute name="endianness" type="xs:string"/>
  </xs:attributeGroup>

  <xs:complexType name="DOCL-connection">
    <xs:attribute name="source-argument" type="xs:string"/>
    <xs:attribute name="source-node" type="xs:string"/>
    <xs:attribute name="destination-argument" type="xs:string"/>
    <xs:attribute name="destination-node" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="DOCL-argument">
    <xs:attributeGroup ref="execution.attributes"/>
    <xs:attributeGroup ref="argument.attributes"/>
  </xs:complexType>

  <xs:complexType name="DOCL-scratchpad">
    <xs:attributeGroup ref="execution.attributes"/>
    <xs:attributeGroup ref="argument.attributes"/>
  </xs:complexType>

  <xs:complexType name="DOCL-node">
    <xs:sequence>
      <xs:element ref="DOCL-argument"/>
      <xs:element ref="DOCL-connection"/>
      <xs:element ref="DOCL-scratchpad"/>
    </xs:sequence>
    <xs:attribute name="location"       type="xs:string"/>
    <xs:attribute name="name"           type="xs:string"/>
    <xs:attribute name="type"           type="xs:string"/>
    <xs:attribute name="kernelName"     type="xs:string"/>
    <xs:attribute name="localWorkSize"  type="xs:string"/>
    <xs:attribute name="globalWorkSize" type="xs:string"/>
    <xs:attributeGroup ref="execution.attributes"/>
  </xs:complexType>
</xs:schema>
```

Appendix D: Example Task Graph XML representation



Task graph represented

```
<DOCL-document>
  <DOCL-node name="node1" location="397,288" type="DAGKernelNode"
      globalWorkSize="10,1,1" localWorkSize="100,1,1">
    <DOCL-argument label="outlet" type="Int"
        direction="Input" endianness="little" size="1"/>
    <DOCL-argument label="source" type="Int"
        direction="Output" endianness="little" size="1"/>
    <![CDATA[
// This is the kernel source for node1.
// The kernel function definition is generated automatically.
@kernel {
    int i = get_global_id(0);
    int j = get_local_id(0);

    // Do some work for i and j.

    return;
}]]>
  </DOCL-node>

  <DOCL-node name="node" location="393,142" type="DAGKernelNode"
      globalWorkSize="5,2,1" localWorkSize="10,10,10">
    <DOCL-argument label="outlet" type="Int"
        direction="Input" endianness="little" size="1">
    </DOCL-argument>
    <DOCL-argument label="source" type="Int"
        direction="Output" endianness="little" size="1">
    </DOCL-argument>
    <![CDATA[
@kernel {
    // Do some work

    return;
}]]>
  </DOCL-node>
  <DOCL-node type="DAGFileSinkNode" name="sink" location="686,194"
      URL="file://path/to/local/resource.csv" discardBefore="0"
      discardAfter="0" fieldSeperator="comma" fileType="ascii">
    <DOCL-argument label="Input" type="Int"
        direction="input" endianness="little" size="1"/>
    <DOCL-argument label="untitled" type="Int"
        direction="input" endianness="little" size="1"/>
  </DOCL-node>

  <DOCL-node type="DAGFileSourceNode" name="src" location="59,211"
      URL="http://somewhere.com/remote/url/resource.dat"
      discardBefore="0" discardAfter="0" fileType="binary">
    <DOCL-argument label="Output" name="Output" type="Int"
        direction="output" endianness="little" size="1"/>
  </DOCL-node>

  <DOCL-connection source-node="node"
      source-argument="source" destination-node="sink"
      destination-argument="Input"/>

  <DOCL-connection source-node="node1"
      source-argument="src" destination-node="sink"
      destination-argument="untitled"/>

  <DOCL-connection source-node="src" source-argument="Output"
      destination-node="node" destination-argument="outlet"/>

  <DOCL-connection source-node="src" source-argument="Output"
      destination-node="node1" destination-argument="outlet"/>
</DOCL-document>
```

Appendix D. Simple task scheduler

```
- (void)schedulePendingNodes:(NSArray *)nodes
{
// This stores the current assignment (in seconds) for each device
  NSMutableDictionary *devicesDict;
  devicesDict = [[NSMutableDictionary alloc] init];

  // Preferentially schedule the longest nodes
  NSArray *sortedNodes = [nodes sortedArrayUsingComparator:
    ^NSComparisonResult(DAGKernelNode *obj1, DAGKernelNode *obj2) {

      // Find the minimum time of the first object
      float minimumTime1 = FLT_MAX;
      float minimumTime2 = FLT_MAX;
      for (DAGNodeStats *stats in [obj1 stats]) {
        float mean = [stats mean]
        minimumTime1 = (mean < minimumTime1)? mean : minimumTime1;
      }

      for (DAGNodeStats *stats in [obj2 stats]) {
        float mean = [stats mean]
        minimumTime2 = (mean < minimumTime2)? mean : minimumTime2;
      }

      if (minimumTime1 == minimumTime2) return NSOrderedSame;
      if (minimumTime1 < minimumTime2) return NSOrderedDescending;
      else return NSOrderedAscending;
    }];

// For each node in the pending nodes, set the peer UUID and
// device ID fields.  This is the mapping from nodes to peers.
  for (DAGKernelNode *node in sortedNodes) {

    DAGNodeStats *bestStat = nil;
    NSString *leastAssignedDeviceKey = nil;
    float smallestAssignment = FLT_MAX;

    for (DAGNodeStats *stat in [node stats]) {
      float mean = [stat mean];

      NSString *deviceKey = [NSString stringWithFormat:@"%@:%d",
                             [[stat peer] uuid], [stat deviceID]];
      NSNumber *num = [devicesDict objectForKey:deviceKey];

      float assignment = (num == nil)? 0 : [num floatValue];

    // Add the assignment to the mean, and see if it's the smallest
      if (mean + assignment < smallestAssignment) {
        smallestAssignment = mean + assignment;
        leastAssignedDeviceKey = deviceKey;
        bestStat = stat;
      }
    }
  // Now, assign the node to the device
    NSNumber assignmentNumber;
    assignmentNumber = [NSNumber numberWithFloat:smallestAssignment]
    [devicesDict setObject: assignmentNumber
                    forKey:leastAssignedDeviceKey];
    [node setPeerUUID:[[bestStat peer] uuid]];
    [node setDeviceIndex:[bestStat deviceID]];
  }

  [devicesDict release];
}
```