Analysis and Visualization of Communication/Computation Patterns of High-Performance Applications

A Thesis Presented
by

Eduard Llamosí

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Informatics Engineering

Northeastern University
Boston, Massachusetts

December 2013

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Analysis and Visualization of Communication/Computation Patterns of High-Performance Applications.

Author: Eduard Llamosí.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Dr. Gunar Schirner	Date
Thesis Reader: Dr. David Kaeli	Date
Thesis Reader: Dr. Ningfang Mi	Date
Department Chair: Dr. Sheila Hemami	Date

Director of the Graduate School:

Dean: Dr. Núria Castell Ariño (Facultat d'Informàtica de Barcelona)	Date

To my family.

Contents

List of Figures **v**

List of Acronyms **vii**

Acknowledgments **viii**

Abstract of the Thesis **ix**

1 Introduction **1**
1.1 Motivation . . . 3
1.2 Problems to be Solved . . . 5
1.3 Approach . . . 6
1.4 Benefits Overview . . . 7

2 Background **9**
2.1 The Charm++ Language . . . 10
2.1.1 Structure of a Charm++ Program . . . 11
2.1.2 Chare Collections . . . 13
2.2 Widely Used Charm++ Applications . . . 15
2.2.1 NAMD . . . 15
2.2.2 ChaNGa . . . 15

3 Solution Overview **16**
3.1 Charm++ Trace Generation for Projections . . . 17
3.2 Extending Projections . . . 17
3.2.1 PE-to-PE Communication . . . 17
3.2.2 Communication and Computation Graph . . . 18

4 Graph Visualization of Communication and Computation **19**
4.1 Goals . . . 19
4.2 Approach . . . 20
4.3 Structure . . . 21
4.4 Internal Graph Representation . . . 22
4.4.1 Generation . . . 25

4.4.2 Traversal . . . 31
4.5 Visual Graph Representation . . . 35
4.5.1 User Interface . . . 35
4.5.2 The JGraphX Drawing Library . . . 36
4.5.3 Style Manager . . . 36
4.5.4 Node Placement . . . 37

5 Visualization of PE-to-PE Communication 41
5.1 Goals . . . 41
5.2 Approach . . . 42
5.3 Structure . . . 42
5.4 Data Collection . . . 43
5.4.1 Statistics Collection . . . 45
5.5 Visual Data Representation . . . 45
5.5.1 User Interface . . . 46
5.5.2 Data Plotting . . . 47
5.5.3 Representing and Displaying Tables . . . 47

6 Synthetic Benchmark 50
6.1 Goals . . . 50
6.2 Approach . . . 51
6.3 Structure . . . 52
6.4 Communication Patterns . . . 52
6.4.1 All-to-all Communication . . . 53
6.4.2 Circular Communication . . . 54

7 Experimental Results 56
7.1 Validation through the Synthetic Benchmark . . . 57
7.1.1 Tool Limitations . . . 59
7.2 PE-to-PE Communication Analysis . . . 59
7.2.1 Tool Limitations . . . 65
7.3 Communication Overheads . . . 65

8 Conclusions 66
8.1 Considerations for Future Work . . . 66

Bibliography 68

A Tool Usage Notes 71
A.1 Communication/Computation Graph Tool . . . 71
A.1.1 Statistics Description . . . 71
A.2 PE-to-PE Communication . . . 73
A.2.1 Statistics Description . . . 73
A.3 CCBench User Manual . . . 75
A.3.1 Building CCBench . . . 75
A.3.2 Running CCBench . . . 76

List of Figures

1.1 Illustration of the butterfly communication pattern found in the fast Fourier transform which results from the combination of smaller discrete Fourier transforms into larger ones. 7

2.1 Overview of the files involved in the definition of a Charm++ chare, which is carried out as an extension of a C++ class. 10
2.2 A Charm++ application's execution structure. 11

3.1 General overview of the infrastructure used to gain access to a Charm++ application runtime events. 16

4.1 Simple example of desired output, where an entry within each PE sends a message to the other PE, triggering another entry on the remote PE. It also shows the relative computation (red nodes) and communication (blue nodes) weights through their color intensity. 21
4.2 Conceptual view of the Communication and Computation Graph tool structure, illustrating the flow of information between components. 21
4.3 Class diagram of the Communication and Computation Graph's internal node structure 23
4.4 Diagram of the internal Communication and Computation Graph's building classes 26
4.5 Overview of the parallel graph building process 27
4.6 Structure of the Communication and Computation graph tool's window. 36
4.7 Node placement process for a purely topological view. 38
4.8 Node placement process considering both topological and chronological constrains. 40

5.1 Conceptual view of the PE-to-PE Communication tool structure, illustrating the flow of information between components. 42
5.2 CommunicationTotals class used in matrix data gathering, encapsulation and provision. 43
5.3 Diagram of the communication data gathering classes. 44
5.4 Overview of the parallel communication data gathering process. 45
5.5 Structure of the PE-to-PE Communication tool's user interface. 46
5.6 Extracts of different plot types generated with surfaceplot using the same data. . . . 47
5.7 Class diagram illustrating the extension of Java's TableModel to support tables with titled rows as well as the extension of the JTable component to visualize them. . . . 48

6.1 Depiction of the two implemented communication patterns for 3 PE over the course of a single iteration. 53

7.1 Graph tool displaying the graph resulting from running CCBench with the circular communication pattern on 18 PEs. 57
7.2 Graph tool displaying the graph resulting from running CCBench with the all-to-all communication pattern on 3 PEs. 58
7.3 A scaled down fraction of the graph resulting from a 100ms sample of a NAMD execution. 59
7.4 PE-to-PE communication results for CCBench using the all-to-all communication pattern on 18 PEs. 60
7.5 PE-to-PE communication results for CCBench using the ring communication pattern on 18 PEs. 61
7.6 PE-to-PE communication results for NAMD running in non PME mode, running a 5000 step job on 32 PEs. 61
7.7 PE-to-PE communication results for NAMD running in PME mode, running a 5000 step job on 32 PEs. 62
7.8 PE-to-PE communication results for the Dense LU benchmark, performing a LU factorization of a 4096x4096 matrix on 32 PEs. 62
7.9 PE-to-PE communication results for the Barnes-Hut N-body solver, using an input size of 20,000 particles on 32 PEs. 63
7.10 PE-to-PE communication results for AMR, using a maximum depth of 7 on 32 PEs. 63
7.11 PE-to-PE communication results for the LeanMD molecular dynamics benchmark for 8000 steps on 32 PEs. 64
7.12 PE-to-PE communication results for the OpenAtom molecular dynamics simulator with its medium sized auto test on 16 PEs. 64

List of Acronyms

CPU Central Processing Unit. Hardware within a computer that performs the basic arithmetical, logic and I/O operations from the computer's programs.

GPU Graphical Processing Unit. Specialized processor that offloads and accelerates 2D or 3D graphics rendering from the general purpose processor.

HPC High Performance Computing. A branch of computer science dedicated to the design and development of supercomputers and the software that runs on them.

MPI Message Passing Interface. A standardized and portable message-passing system designed to be used on a wide variety of parallel computers.

PE Processing Element. In Charm++ terminology, a logical element that acts as a unit of mapping and scheduling with an associated pool of messages.

SMP Symmetric Multiprocessor System. A multiprocessor system with centralized shared memory operating under a single operating system.

Acknowledgments

My first and sincere appreciation goes to Gunar Schirner, my supervisor for all I have learned from him and for his continuous help and support in all stages of this thesis, even well outside the boundaries of my exchange program. I would also like to thank him for being an open person to ideas, and for encouraging and helping me to shape my interest and thoughts.

In addition, I would like to thank David Kaeli for accepting to be part of the examining committee in this project and for his interesting Big Data research course, as well as his involvement in making the exchange program I had the pleasure to take part of possible. Also, thanks to Ningfang Mi for accepting to be the external examiner in my committee.

I would like to thank my family, especially my mother and father for always believing in me, for their continuous affection and support in my decisions. Without whom I could not have made it here.

Thanks to my supportive research companion, Chulian Zhang, and others who made the lab a friendly environment for working.

Abstract of the Thesis

Analysis and Visualization of Communication/Computation Patterns of High-Performance Applications

by

Eduard Llamosí

Informatics Engineering in Electrical and Computer Engineering

Northeastern University, December 2013

Dr. Gunar Schirner, Adviser

Energy consumption is an increasingly important concern in high performance computing (HPC) data centers. Today, up to half of the energy in the computing clusters is consumed by the cooling infrastructure. While most of the focus to improve the power efficiency is targeted at the electronics, software can also contribute to such improvements. Thermally-aware job allocation policies are being researched to optimize the cooling costs by minimizing the peak inlet temperatures of the data center server nodes. However, job allocation decisions strongly affect the performance of HPC applications with signicant inter-processor communication. To achieve a balanced tradeoff between thermal performance and application throughput, the actual communication requirements of the applications need to be considered.

This thesis describes the development of two new profiling tools that aim at analyzing the communication and computation patterns of Charm++ based HPC applications. One of the tools is designed to collect and plot the amount of communication between every pair of processing elements, which allows to evaluate how communication is distributed across processors, while the other generates a graph composed of communication and computation nodes and their dependencies. This is achieved through the use of compiler introduced logging features, which allows for any

Charm++ application to be profiled without changes in its source code. The analysis they provide can be applied to the evaluation of the performance impact that new job allocation policies can have in communication performance for a particular application. As part of the validation process of the developed tools, to demonstrate their accuracy, a synthetic benchmarking tool is also developed.

Chapter 1

Introduction

High Performance Computing (HPC) is a branch of computer science dedicated to the design and development of supercomputers and the software that runs on them. These supercomputers are machines meant to surpass the processing capability of their commonly available counterparts. While at the time of their introduction during the 1960s they used a small number of processors, machines with thousands of processors appeared in the 1990s. By the end of the 20th century this trend carried on, and supercomputers went to become massively parallel machines often composed of up to tens of thousands of consumer-like processors networked together. Even within the context of personal computers and mobile devices a similar trend is taking place, where the processors make use of multiple identical cores to provide performance increases.

There are typically two different approaches to the implementation of these supercomputers with a massive number of processors:

- **Grid based Supercomputers** are large number of computers networked together that share their resources to fulfill a task. Within this network, its individual computers are often loosely coupled, heterogeneous, multipurpose and span geographically disperse computing centers. This approach usually aims to maximize the system's scalability.

  An example of such approach is The Worldwide LHC Computing Grid (WLCG) [9], a global collaboration of computing centers tasked with the analysis of the data generated by CERN's Large Hadron Collider.

- **Cluster based Supercomputers** use a large number of processors physically close to each other, typically within a single computing center. The computers and networks within a clus-

ter are often purpose built. This family of supercomputers tends to be more throughput oriented.

Massachusetts Green High Performance Computing Center (MGHPCC) [5] or Barcelona Supercomputing Center's MareNostrum [4] would be examples of this second approach.

These large machines have high power power demands, and almost all of it is converted into heat, which requires cooling. Cooling requirements can amount to a significant fraction of the overall power demands (up to half of them in some cases [22]). This is particularly true in cluster based supercomputers, given the number and density of electronic components to be cooled. For instance, China's Tianhe-1A supercomputer has a peak power consumption of 24 megawatts, out of which 6.2 MW are dedicated solely to cooling [23]. For perspective, 24 MW at $0.10/kWh amount to $2400 an hour which translates to around $21 million per year.

Not surprisingly, considering the growth of such supercomputers, their large power demands sparkled interest in more power efficient approaches to supercomputers to reduce both the computing's power demands as well as the cooling requirements. The teams behind the MGHPCC are focusing their efforts towards the later, designing an advanced cooling system aimed to reduce the need of air conditioning by maximizing the use of cool outside air [6].

However, not all efforts to achieve such objective need to be made from a hardware design standpoint. Adequate knowledge of the underlying hardware by the software can contribute to reduce power requirements and improve throughput by making a more efficient use of it. This can be achieved by different means, such as optimizing the software for the particular architecture being used within the nodes, which can yield both better performing code as well as significant power savings. The practices to accomplish such a thing are typically equivalent to the ones employed for architecture specific optimizations outside of supercomputing contexts, which generally fall between the use of purpose specific instructions offered by the CPU or the use of purpose specific hardware, such as a GPU as a stream processor. However, our focus is a different one which is fairly more HPC specific: the mapping of tasks to particular nodes within a supercomputer.

Task allocation within specific nodes can have different significant impacts on HPC applications. For example, allocating different tasks that require a lot communication between them in different nodes physically distant to each other will result in higher latency, more usage of the network's resources (which at the same time will depend on the network's topology) and decreased throughput as a consequence of both, particularly if the miss-use of network resources result in congestion. At the same time, allocating highly demanding tasks physically close together could

create a thermal hotspot, which might easily increase the overall power required for cooling over that which a more spread out mapping likely would.

As the number of processors increases, so does the reliance of a parallel application on communication with it becoming an increasingly important factor to the the performance of many HPC applications, task mapping shouldn't solely take into account heat management, but seek an parametrized balance between it and the application's throughput. To achieve so, it's important to understand an application's computation and communication requirements. In addition, a good understanding of these requirements could provide clues for an HPC application's developers to enhance their application by resolving bottlenecks due to excessive computation or communication demand at a given point of the application and by maximizing the highly desired overlap of communication and computation for overhead minimization in parallel algorithms.

This thesis aims to analyze the existing profiling tools available for HPC applications, evaluate their capabilities for inspection of an application's computation and communication demands, suggest different approaches that might better suit our needs and develop an implementation of said new approaches.

1.1 Motivation

Most parallel applications spend some of their time waiting for data to become available. If a problem isn't embarrassingly parallel, there are dependencies between the work being performed in different processing elements (PEs). A PE, in this context and outside of Charm++ specific terminology, could be any logical or physical component capable of carrying out a unit of work (a thread, specialized hardware, a computer within a grid...) out of the several of which the parallel program is composed of. Minimizing these dependencies is important to create well performing HPC applications. Not all of these dependencies can be eliminated, though, as some are intrinsic within the problem at hand, but for the ones that remain the idle time spent waiting for external data should be minimized.

To achieve this, the work assigned to each PE should be adequately distributed across the system, otherwise the less heavily tasked PEs will end up having to wait for data from their more heavily loaded counter-parts to continue. This balance should be time focused, rather than based in the amount of work, as the objective would be minimizing the maximum time used by any of the PEs to finish a program stage, a process known as load balancing. This is particularly relevant topic in heterogeneous parallel systems or systems with shared resources.

Even if an application is perfectly load balanced, it might spend significant time waiting for data to be exchanged between PEs, as such transfers can be costly regardless of the underlying transport (a bus, a network, the internet...). Load balancing doesn't prevent a processing element from becoming computationally idle during the time invested in such communication, it just minimizes the chance of a particular PE becoming the application's bottleneck.

In order to minimize the idle time caused by waiting for the data transfers between PEs to be completed, this communication process should overlap with some computation by scheduling additional, independent work. This can not always be achieved, however, as it might be infeasible to find any work that doesn't rely on the data that's being transferred. A. Becker and R. Venkatarama suggest a number of patterns for programs to achieve such overlap based on overdecomposition, non-blocking communication, and speculation [1].

The aforementioned situation suggests looking into the analysis of inter-node communication. However, a combined look at the application's computation and communication stages could provide additional information that could help in optimizing parallel applications while preserving the same data gathering mechanism. Some of the most important issues in parallel applications are minimizing data dependencies between processing elements and maximizing the overlap of communication and computation.

Fulya Kaplan and Jie Meng from the Boston University recently proposed [12] a job allocation policy that optimizes both the application performance in terms of the communication cost and the cooling energy of HPC data centers. Said policy confines the communicating nodes of a job in close proximity, while also selecting the most cooling-efficient locations possible. They formulate and solve the joint optimization problem using binary quadratic programming and demonstrate that for jobs involving intensive communication between the nodes, application performance becomes an important factor in determining the total cooling energy consumed.

Evaluating the real world impact that such thermal aware job allocation policy can have on communication performance suggests investigating the communication demands of real world HPC applications, as learning the communication patterns that such applications rely on can be used to predict the the policy's impact on them. This can be achieved through a prediction model that considers the increased number of network hops that the communication between every communicating pair of nodes needs to go through on the new policy, as well as the amount of communication between them.

Such evaluation has been carried out by Ashfaq Khan in collaboration with the Boston University's group by instrumenting NAMD[2], a molecular dynamics application written in

Charm++charmweb targeted to high performance computers. This was achieved by modifying the application's source code, so it would log the number and size of messages involved in inter-node communication. However, this limits the scope of such evaluation to the communication patterns that this single application employs. To get a broader look at how different HPC applications tend to communicate, the same instrumentation task would need to be performed on a large set of applications. Accurately understanding the behavior of each of these applications or modifying their source in order to instrument them would be extremely time consuming, however.

The developers of Charm++, a programming language intended for HPC environments, also offer Projections[11] [7], a trace-based performance evaluation program designed for use with Charm++ applications. Projections' tool set, however, while it offers tools to quantify the communication that a node has produced or received, it doesn't offer a way that allows quantifying that for a given pair of nodes, which makes it unsuitable to be used to gather data for evaluation of the thermal aware job allocation policy.

1.2 Problems to be Solved

Design and develop a trace-based analysis system that can be easily used to instrument any existing Charm++ based application. This system should accomplish the following goals:

- Provide means to evaluate their inter-PE communication demands in a way that allows to infer their communication patterns. As a single application can be using different communication patterns during its execution, either in a staged or overlapping way, means have to be provided so this evaluation can be performed independently on the stages or in the program's components involved in each pattern.

- Analyze inter-PE dependencies in order to help a developer to pinpoint superfluous dependencies as well as tracking the communication and computation investments of each PE to assist in detecting load balancing issues and provide clues of which parts of the program could be enhanced by scheduling additional work in order to maximize the overlap of communication and computation.

Providing means to approach both problems should enable an HPC application developer to gather data that may be used to predict the impact of using a different job allocation methodology for an application on its communication performance, as well as detecting possible bottlenecks within an application caused by inappropriate load balancing.

1.3 Approach

To obtain the data necessary to accomplish both goals a simulation based approach can be taken by running the arbitrary Charm++ application in an instrumented environment. However, how the data is collected during this process can be approached in two fundamental ways:

- **Per application instrumentation** can be performed by means of providing a framework that can be used by modifying the application's source code that one intends to gather the data from, modifying the points of interest by adding invocations to the framework to register them. It has the advantage of being simple to implement solution, but it has the large disadvantage of being time consuming and complex to use when several possibly complex applications need to be instrumented.

- **Compiler added instrumentation** can allow for any Charm++ program to have the instrumentation relevant code automatically added by the compiler upon generation of the program's binary to monitor particular events that trigger the language's runtime. It has the important advantage that it doesn't require per-application instrumentation. Were the compiler's source need to be modified for such approach, it would be tedious to implement and hard to maintain, as it would become obsolete with the release of a new version of the Charm++ compiler.

The later is an obviously more attractive solution, as it can result in a simple to use tool that doesn't rely on the modification of an application's source code. Fortunately Charm++ already implements a runtime instrumentation option that allows for applications to generate log files for Charm++'s own performance analysis tool, Projections. Using the traces left on these logs most of the information needed for this thesis' analysis tools can be obtained. Also, using Projections' Java) source code was deemed as an appropriate basis to build these tools from, as it allows for readily use of its log reading framework, which given the nature of their non trivial binary format and compression would have made alternative approaches to access these complex and time consuming.

The work of this thesis consists in the development of two complimentary tools for Projections that attempt to provide a solution the stated problems (Chapter 1.2):

- **PE-to-PE Communication:** performs an analysis of the communication between each pair of processing elements, quantifying amount of data transferred and number of messages.

Allows filtering the results according to the method receiving the remote message, as well as also allowing time and PE based filtering.

- **Communication and Computation Graph:** displays a graph representation of the program's inter and intra-PE communication and computation dependencies, highlighting communication or computation steps according to their perceived computation time or transfer size. Also allows for method, time and PE based filtering.

Aside from the addition of two different tools to Projection's portfolio, an independent Charm++ program is also developed for verification purposes: CCBench, a simple synthetic benchmark that allows running an application with configurable communication and computation demands. This allows to contrast the validity of the logged data and the results of the developed tools with the easily predictable results that the benchmark offers.

1.4 Benefits Overview

The proposed approach of extending Projections with the addition of two new tools has the important advantage of being able to re-use the existing infrastructure for the instrumentation of Charm++ applications. By themselves, the tools should offer the following benefits:

- **Quantification of the communication between pairs of nodes** that allows gathering data for use in prediction models for thermal aware task allocation algorithm so estimations can be made of an application's throughput with the different task allocations. This also allows for the detection of communication hot spots between pairs of nodes.

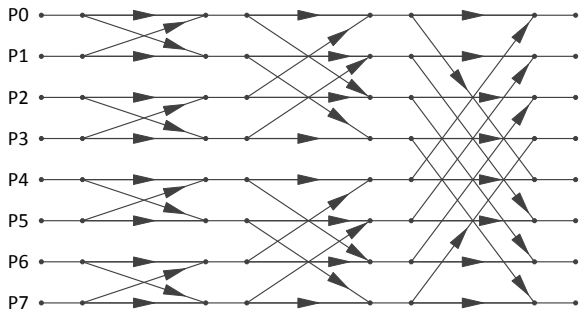


Figure 1.1: Illustration of the butterfly communication pattern found in the fast Fourier transform which results from the combination of smaller discrete Fourier transforms into larger ones.

- **Ability to infer an application's communication patterns** by providing enough information of inter-node communication. Known communication patterns (all-to-all, self, neighboring and more complex ones such as the FFT's butterfly shown in Figure 4.5) should be easy to identify. This can be used to gather data about how frequently each pattern appears in commonplace HPC applications and analyze the behavior of the thermal aware task allocation algorithm on the most predominant ones.

- **Detection of communication and computation hot spots in the program flow**, the latter being particularly useful to help in finding an load balancing issues between processing elements.

- **Display of communication dependencies between PEs**, which can be helpful for the application's developer to pinpoint unnecessary dependencies within their application, as well as finding additional work which could be scheduled in order to maximize the communication and computation overlap.

All these seek to provide a developer with more information to analyze and improve communication related issues that appear within HPC applications and current Charm++ profiling tools don't properly address.

Chapter 2

Background

Programming techniques used for parallel environments differ significantly from the more traditional ones, which are typically focused on sequential program flows. Many imperative languages that are still in common use today, such as C and C++, were designed for single threaded execution. To overcome their limitations, new libraries, frameworks and languages appeared over the years aimed at supporting parallel program development within their feature repertoire.

Within HPC contexts, to achieve inter processor communication a message based model is typically employed due to supercomputers' reliance on computer networks for inter node communication and their distributed memory architecture. While general purpose languages can make use of a Message Passing Interface (MPI) library to build a program suitable for a supercomputer, they typically offer alternative communication models which assume a shared memory architecture that cannot be translated into a binary that makes effective use of a supercomputer's resources.

Charm++ is a C++ derived language targeted at HPC environments developed by the University of Illinois that aims to overcome these issues by building a concurrency model on top of C++ and implementing language level message based communication in the form of asynchronous remote procedure invocations. It also provides several other features, such as fault tolerance and object migration.

This thesis' work is built on Charm++ and related tools, which is an attractive language for HPC application development due to its efficient portability and modularity. Additionally, such choice allows providing results that would corroborate and extend previous work performed by the research group, which also was Charm++ based. An overview of the language follows, as well as brief descriptions of the related tools that are used in one way on another throughout the development of the thesis. An overview of Charm++ follows to introduce the reader to its features

and terminology, which are referenced throughout the description of this work.

2.1 The Charm++ Language

Charm++ is defined as an object-oriented asynchronous message passing parallel programming paradigm [19]. It's built as an extension of C++ rather than as an entirely new language, adding features and structuring elements to offer better support for HPC environments than C++ natively does.

One of the main limitations that Charm++ overcomes is that of offering language level support for parallel applications. These applications have multiple threads of execution that either execute simultaneously or perceptively behave as such. Different threads can share a single processing element through time-division multiplexing. One may have several threads time sharing a single or multiple processing elements (PEs).

As an extension of C++, it preserves its object orientation features in which programs are built as collections of logical objects have associated data fields (attributes) and procedures (methods). In addition, Charm++ introduces a special kind of objects called chares. Each chare object will keep its own internal state, send and receive messages, as well as perform some task upon receiving a message by executing special member functions called entries.

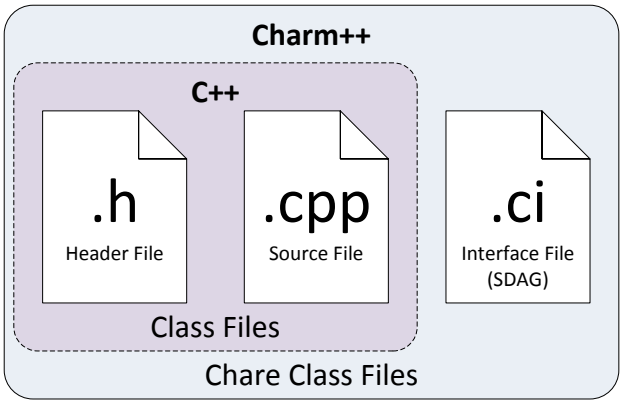


Figure 2.1: Overview of the files involved in the definition of a Charm++ chare, which is carried out as an extension of a C++ class.

Its message passing model allows for communication between chare objects by sending messages to one another in the form of remote method invocation of the special entry methods. This takes place in an asynchronous way, as the code belonging to the chare that creates and sends a message continues execution even though the message might have yet to reach the target chare

object. Receiving a message is an asynchronous process as well, as the target chare can receive a message at any given time, it won't stop the code that it's currently executing to immediately process the received message.

2.1.1 Structure of a Charm++ Program

At a high-level, from the programmer's perspective, a Charm++ program is simply a collection of chare objects. Each chare object has some state associated with it. The chare objects communicate by sending messages to one another. When a particular chare object receives a message, it will execute an entry method to processes the message. This entry method may perform one or more operations/calculations, it may send more messages to other chare objects, it may buffer the contents of the message for later processing, or it may do nothing at all. This is how forward progression is made in the overall application. One chare sends a message to another chare, the receiving chare does some computation and then sends out more messages to other chares, and so on, and so on. Execution begins with a special chare called the main chare (similar to how execution of a C++ program begins with the exection of a special function called main).

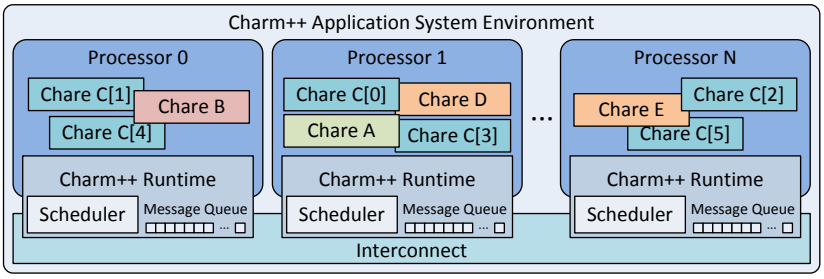


Figure 2.2: A Charm++ application's execution structure.

It is worth pointing out that the description of a Charm++ application does not involve the number of processing elements (i.e. one thread per processing element), what type of processing elements are involved, how many chares there are per processing element, or even how data is transferred between processing elements. Charm++ applications are simply written in terms of the chare objects. The programmer does not have to worry about the number of processing elements that will be available when the program is executed or what type of interconnect (network) will be used to connect the processing elements together. All of these details are left to an intelligent runtime system called the Charm++ Runtime System. The Charm++ Runtime System takes care of mapping each of the chares to the physical processing elements, routing messages over the interconnect, and so on.

2.1.1.1 Chare Objects

Each of the chare objects in the Charm++ application can be thought of as independent entities. Each chare object has it's own state (that is, data local to it). Chare objects in the application cannot directly access the state of another chare objects, however, any chare object can directly communicate with any other chare object. The overall set of chare objects in the application is commonly refereed to as the global object space. Any chare object in the global object space can request information on another chare object's state by sending the chare object a message requesting the information.

For the purposes of this tutorial, we will use two terms: chare class and chare object. These terms have the same basic meaning as their C++ equivalents, class and object, respectively. A chare class defines a type of chare object (the blueprints of a chare object). A chare object is a specific instance of a chare class. Another term that will be used in this tutorial is chare array. A chare array is simply an array of chare objects (just like an integer array is an array of integers in C++). See the section on Chare Collections below.

There is one special chare object in a Charm++ application, the main chare object. The main chare object acts as the entry point of the Charm++ application in a similar manner to the main function in a C++ application (see Entry Methods below).

2.1.1.2 Messages

The chare objects communicate with each other by sending messages to one another. In Charm++, this process is also referred to as remote method invocation since the sending chare object simply calls one of the receiving chare object's entry methods (more on entry methods later). From the programmer's perspective, the code basically looks like it is calling a member function on the target chare object.

2.1.1.3 Entry Methods

Entry methods are special member functions of the chare class. The difference between a normal C++ member function and a Charm++ entry methods is that Charm++ entry methods act as the reception points for messages (or, in other words, they are the member functions that can be remotely invoked by other chare objects). When one chare object does a remote method invocation on another chare object, the data being send is packed into a message and passed to the receiving chare object. Once the receiving chare object receives the message, the entry method that was

specified by the sending chare object is invoked (executed) and the data in the message is passed to that entry method.

Each of the chare classes also has constructor entry methods (basically the same thing as a constructor in C++). When a chare object is created, this constructor entry method is automatically invoked by the Charm++ Runtime System to create the chare object (once again, similar to what is done for an object in C++). The execution of a Charm++ application begins with the execution of the main chare object's constructor.

There are two aspects of entry methods that make the different from standard C++ member functions. First, entry methods do not have return values (i.e. they return void). Second, from the perspective of the calling chare object, the entry method returns immediately and the code of the calling chare object continues to execute. This does not mean that the target entry method has executed, only that a message has been sent to the chare object which will cause the target entry method to execute at some time in the future. Both of these aspects of entry methods are directly related to the message passing nature of Charm++. Invoking an entry method on a chare object does not cause the entry method to execute immediately nor is any value produced immediately. Instead a message is sent to the target chare and the entry method will be executed at some point in the future.

2.1.1.4 Proxies

Because the actual chare objects in the global object space in a Charm++ program are spread out across the various processing elements, it is not always possible for two chare objects to directly communicate with each other. Instead, for one chare object to invoke an entry method on another chare object, the sending chare object must first have a reference called a proxy to the target chare object. The proxy objects hide the details of the actual communication from the programmer. Entry methods are called on the proxy object as if the chare object itself were local to the current physical processor. The Charm++ Runtime System takes care of locating the actual chare object in the global object space on behalf of the chare object. See Figure 2.

2.1.2 Chare Collections

In actual Charm++ applications, it is more common to see collections of chare objects rather than individual chare objects. The collection of chare objects is spread out across the pro-

cessors and all member are commonly performing a similar operation on different pieces of data. There are several types of chare collections.

Each collection within a program is represented by a unique handle. This handle is visible to all processors (can be transferred between processors) and can be used to access the members of the collection.

2.1.2.1 Chare Arrays

As was previously mentioned, a chare array is simply an array of chare objects. The array can indexed using a simple integer index similar to how C ++ arrays are indexed (int arrays, for example). Chare arrays can be single or multidimensional (indexing schemes for 1D through 6D are provided). Chare arrays can also be indexed using more complex schemes such as a bit vector or even use user defined objects to index into the array.

Initially, the chare array elements are spread out according to a mapping scheme (with round-robin being the default). However, because the chare array elements are migratable chare objects, they can be moved between the processors on an individual basis. The Charm++ Runtime System will migrate the chare array elements between the processors based on runtime data that it gathers in an effort to balance the processing load on all of the processors. The programmer does not have to concern themselves with where each of the individual chare array elements are. Even if a chare array element is moved from one processor to another, the programmer still refers to it as the same array element (e.g. myChareArray[i] for the ith element in the myChareArray chare array). The Charm++ Runtime System takes care of making sure the message are redirected to the appropriate processor.

2.1.2.2 Chare Groups

A chare group is a collection where each of the physical processors available to the application has one, and only one, representative (chare object) located on it.

2.1.2.3 Chare NodeGroups

A chare nodegroup is similar to a chare group. The difference is that there is only a single representative per node, instead of per processor.

2.1.2.4 SDAG

Charm++ provides such constructs for structured control flow across an object's entry methods in a notation called Structured Dagger. The basic constructs of Structured Dagger (SDAG) provide for program-order execution of the entry methods and code blocks that they define. These definitions appear in the .ci file definition of the enclosing chare class as a 'body' of an entry method following its signature.

2.2 Widely Used Charm++ Applications

Charm++ is used in two popular HPC applications, NAMD for Molecular Dynamics and ChaNGa for approximating N-body cosmology problems. Both of them will be used to test the effectiveness of the application analysis tools developed as part of this thesis, in conjunction with additional, smaller applications for testing specific algorithms.

2.2.1 NAMD

NAMD (Not (just) Another Molecular Dynamics program) is a Charm++ based parallel molecular dynamics program designed for high-performance simulations in structural biology [20][2]. It has been developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign. NAMD was designed to run efficiently on such parallel machines for simulating large molecules.

Molecular dynamics (MD) simulations compute atomic trajectories by solving equations of motion numerically using empirical force fields, such as the CHARMM (Chemistry at HARvard Macromolecular Mechanics) force field [13], which approximates the actual atomic force in biopolymer systems.

2.2.2 ChaNGa

ChaNGa (Charm N-body Gravity) [3] is a collaborative project by Professors Thomas Quinn from the University of Washington and Marianne Winslett from the University of Illinois to build NChilada [21], a framework for enabling N-Body based parallel simulations, especially those in Cosmology and Astronomy. It involves various algorithms, with the basic parallel data structures being particles organized into trees.

Chapter 3

Solution Overview

The work of this thesis focuses on the development of profiling tools using compiler introduced instrumentation on a program's runtime library. This approach has the advantages of allowing any Charm++ application to be easily instrumented and the possibility of re-using Charm++'s built in infrastructure for its performance visualization tool to do so. Using Charm++'s own infrastructure for application profiling also provides some degree of future proofing, as it allows using new versions of the Charm++ compiler with minimal changes to our profiling tool's code, if any.

The developed tools are built as an extension to Projections' tool set, Charm++'s Java based analysis and visualization framework. This allows easily re-using of the log reading and decoding infrastructure built within it, which can be accessed through its *GenericLogReader* class.

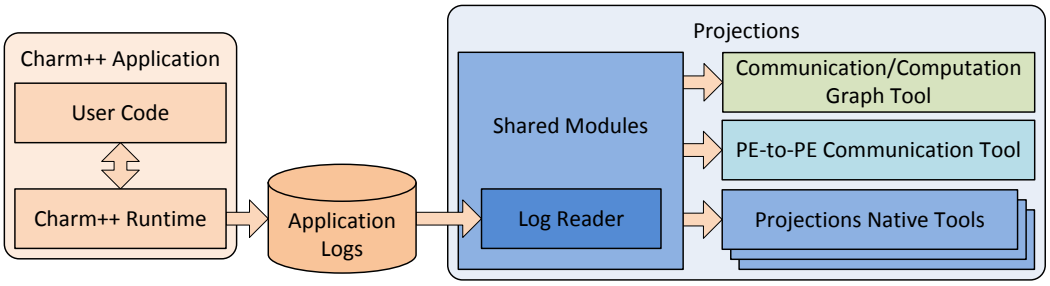


Figure 3.1: General overview of the infrastructure used to gain access to a Charm++ application runtime events.

3.1 Charm++ Trace Generation for Projections

The Charm++ runtime automatically records pertinent performance data for performance-related events during execution. These events include the start and end of entry method execution as well as messages sent from entry methods and the scheduler's idle time. This means most users do not need to manually insert code into their applications in order to generate this trace data.

However an API is also available for user defined events to be captured so additional performance information can be provided. Projections then offers some support for the visualization of such events. In addition greater control over the tracing activities can be achieved through an API which allows, for example, dynamically enabling and disabling the instrumentation.

Each PE creates a series of generally compressed log files containing the application traces for the events produced within that PE. For an application to generate such trace logs, however, it needs to be compiled with the linker option *-tracemode projections*. More detail about the different tracing options can be found in the Projections and Charm++ Manuals [17][16].

3.2 Extending Projections

The generated logs can then be read with the Java-based tool to visually study various performance aspects and locate the performance issues for the execution of that application. The application is structured as a set of different tools that evaluate different aspects of the instrumented Charm++ application which share a number of components for GUI objects and log reading.

Within Projection's modules, the class *GenericLogReader* provides means to read an application's PE log, entry by entry. While creating a new application based on Projection's infrastructure to access the compressed binary log files was considered, Projection's shared modules are tightly coupled and present a complex dependency chain that would have made them time consuming to isolate for re-use. Building the new tools as new additions to Projection's tool set was deemed a more effective approach.

3.2.1 PE-to-PE Communication

Out of these performance evaluation tools the one that provides communication information most closely resembling what is required for performance aware job allocation would be the *Communication Per Processor* tool. However, while it displays the number and size of messages sent or received by a particular processing element, as well as the procedures messages are invoking,

it doesn't quantify the number of messages and their sizes on pairs of PEs, a particularly revealing figure when attempting to allocate the tasks with more communication between them physically close to achieve improved latency and throughput. A new tool is therefore required to be able to collect and display such communication information.

3.2.2 Communication and Computation Graph

That aside, Projections' tool portfolio does not offer the possibility to perform an analysis of communication dependencies between PEs, which limits the ability to detect avoidable dependencies that might have been overlooked during an application's development stages. To overcome this, part of this thesis' work centers around developing a tool that displays a graph representation of the program's inter and intra-PE communication and computation dependencies, also highlighting communication or computation steps according to their perceived computation time or transfer size. While Projections already offers tools that highlight an application's intensive communication or computation spots, displaying such information as part of a graph of the application's dependencies might provide further insight that might otherwise be more difficult to perceive if the dependency context is removed.

Chapter 4

Graph Visualization of Communication and Computation

Most HPC applications spend part of their time waiting for data to become available, as there are dependencies between the work being performed in different PEs. Although eliminating these dependencies might not be possible due to a problem's nature, minimizing them is important to create well performing HPC applications. For the ones that remain, though, the idle time spent waiting for external data should be minimized.

In cases where such dependencies cannot be removed, evening out the workloads on the application's PE's also helps in reducing most application's idle times, as it prevents a PE from supplying data that the others rely one significantly late. Scheduling additional, independent work can also help and can also be performed even during the idle times incurring from PE-to-PE data transfers.

4.1 Goals

This chapter describes a method that provides an analysis of an application's execution that can result in useful insight for reducing such idle times by focusing on realizing the following points:

Assistance in Dependency Analysis

A dependency analysis of an application's communication flow to assist the application's developer in pinpointing superfluous dependencies within their application and therefore al-

lowing to find additional work to be scheduled in order to maximize the communication and computation overlap.

Facilitate the Inference an Application's Communication Patterns

Allow the user to understand how the PE's within an application communicate with each other, to make it possible for them to infer how a 3rd party application communicates in a given stage. This information can then be used for estimations of the application's performance on different network topologies and considering different task allocation methodologies.

Highlighting Communication and Computation Hot Spots

To minimize the time that an application spends waiting to collect an application stage's data an application has to minimize the maximum time used by any of the PEs to finish a task so the application is well load balanced.

The realization of all these goals can provide enough information to the developer to potentially improve an application's load balancing issues and better exploit the overlap of communication and computation.

4.2 Approach

The method addresses the stated goals through a tool that depicts an application's execution tree, representing entry invocations as well as the messages for such invocations as nodes within that tree and their parenthood as edges. The nodes are grouped within the PE they originated from, and be arranged following topological and chronological criteria. Careful node placement allows for the current application's dependencies to be perceivable. Detailed information from the events related to a node is also made available.

Within the graph, nodes are to be colored according to their relative weight in computation or communication. This, in combination with a node placement that considers dependencies (topological) and chronological factors is to allow a developer in pinpointing load balancing issues between PEs in a particular application stage, as well as difference of weights between stages.

The tool has to deal with the challenges of dealing with large application logs, which are commonplace for large applications yielding a significant number of events, as well as being able to correlate events between the logs generated by the different PEs with the information provided in

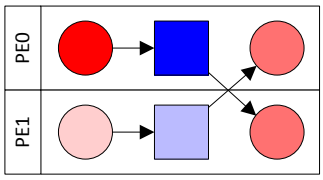


Figure 4.1: Simple example of desired output, where an entry within each PE sends a message to the other PE, triggering another entry on the remote PE. It also shows the relative computation (red nodes) and communication (blue nodes) weights through their color intensity.

them. Its results also have to be presented to the user in a comprehensible manner and offer ways to limit the amount of information presented to them at a given time.

4.3 Structure

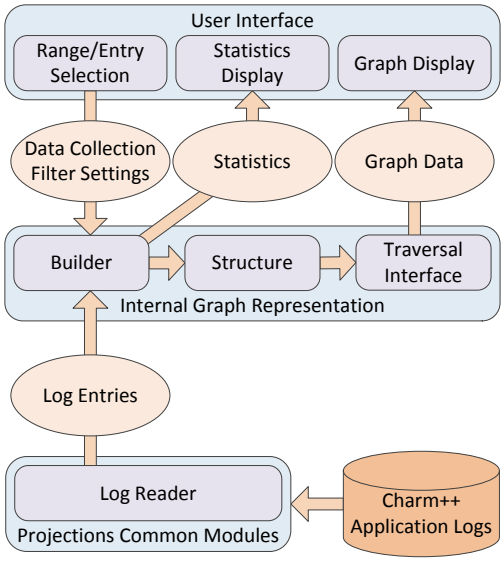


Figure 4.2: Conceptual view of the Communication and Computation Graph tool structure, illustrating the flow of information between components.

Structurally, as depicted in Figure 4.2, the tool follows a triple layered design loosely coupling its UI components from the data gathering and its non-UI related treatment. Such data gathering is achieved by interacting with Projection's shared components, which handle the log file reading and decoding.

Behaviorally, the user starts the tool by providing the analysis range and entry filtering information. Such information is then used by the Graph Builder to retrieve the log information

satisfying the set restrictions and analyze the application's flow, yielding an internal representation of it in the form of a graph. A UI component prepared to visually render graphs accesses this internal representation through an interface, presenting different views of the graph. Said interface abstracts much of the graph's actual implementation. The building process also collects some statistics that provide some information about the fidelity of the built graph.

While the tool, in the end, currently generates a series of trees as an output (graphs without cycles), the tool's nomenclature, internal data representation and algorithms are graph based. The reason for this being expandability. For instance, a possible addition to the tool could be the automatic detection of data gathering points within an application's stage, collapsing the tree's nodes belonging to these points, which would transform the tree into a graph.

4.4 Internal Graph Representation

The implemented design to represent the generated graph attempts to minimize the memory footprint of the resulting objects while at the same time employing common object oriented principles that help in extending the structure and preventing mistakes by the programmer. It attempts to minimize one of the problem's limiting factors, which would be the internal representation's size. All the classes for the graph representation are self contained within the package *eduard.projections.CCGraph*.

As depicted by the class diagram in Figure 4.3 the graph structure is build hierarchically, with each node class adding information that is relevant only to itself and its derivatives, avoiding fields that could otherwise be left empty. A description of these node types

Node

Works as an abstract root type for the rest of the node types and contains the attributes and methods shared across all of them:

- **time:** time stamp of the log entry that resulted in the creation of the node (in ms).
- **pe:** PE to which the node belongs to (for communication nodes, the PE corresponding to the message's origin).
- **entry:** the ID of the entry a computation node is executing or a communication node is invoking. It can be translated to a proper method name using the functionality provided within Projection's *Analysis* class.

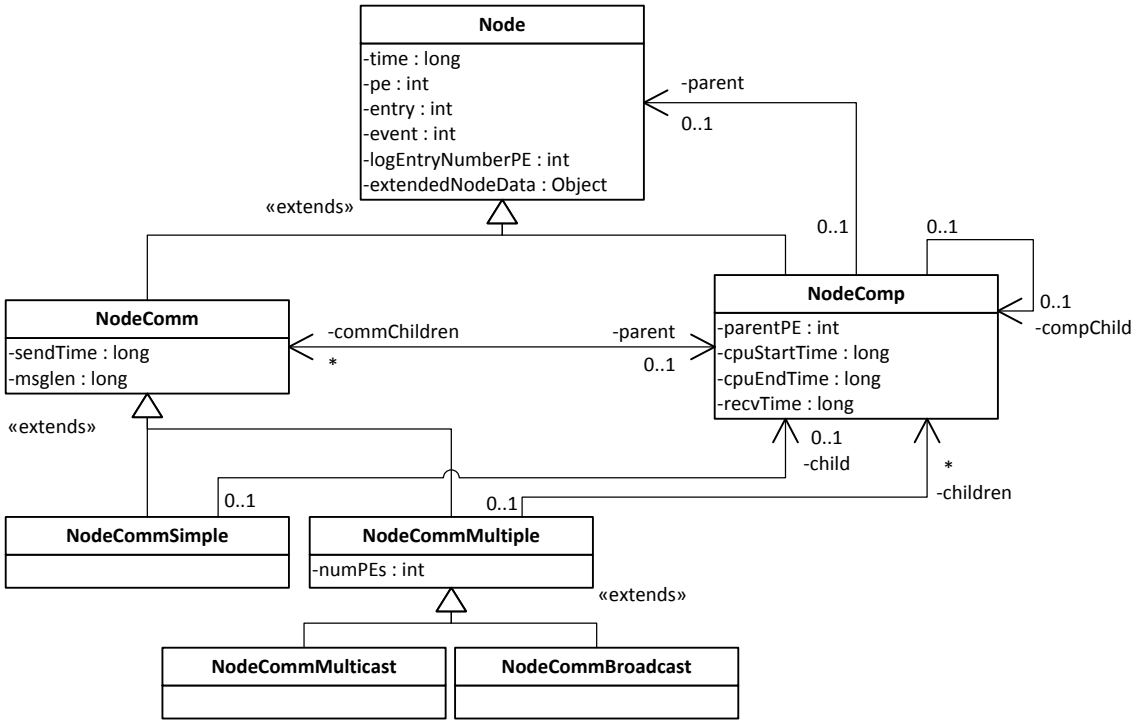


Figure 4.3: Class diagram of the Communication and Computation Graph's internal node structure

- **event:** run-time event ID shared by related events, assigned by the origin's PE.
- **logEntryNumberPE:** number of this log entry within its PE's logs. Used for ordering events that don't have precise enough timing information.
- **extendedNodeData:** attribute that allows storage of additional data to any code that uses the structure.

For simplicity's sake the class diagram does not show *Node* extending *GraphDefs*, which at the same time extends *ProjDefs*. *GraphDefs* contains definitions for use within the graph related classes, particularly exceptions. *ProjDefs* is a Projections class with several definitions which is extended to conveniently expose them within the graph related classes, similarly to how other Projections tools use it.

While it would have been possible to use generics to achieve compile-time checks on the *extendedNodeData*'s type, its addition came late development stages and would have required extensive code re-factoring and in many cases overly long type declarations. The current implementation relies in class casting instead. It might be worth noting that a similar func-

tionality implemented on the jGraph library[15] relies on the later approach as well.

NodeComp

Represents a computation interval by a PE to complete the work of an invoked entry and holds its related information as a computation node.

- **parentPE:** PE that invoked the entry yielding this computation node.
- **cpuStartTime:** time at which this computational work started executing (in ms).
- **cpuEndTime:** time at which this computational work finished executing (in ms).
- **recvTime:** time at which the message invoking this entry was received (in ms).

NodeComm

Abstract class that generalizes any type of communication node and holds the information that is common for any kind of message.

- **sendTime:** time at which the message was sent by the origin PE.
- **msglen:** size of the message (in bytes).

NodeCommSimple

Represents a simple message, that is, a message that targets a single PE. Allows setting a single computation node as a child.

NodeCommMultiple

Abstract class that generalizes messages targeted at multiple PEs. Allows for one to many relationships to computation nodes.

- **numPEs:** Number of PEs that the message has been sent to. Not all targets might have been retrieved from the log data, so this number is stored instead of inferring it from the resulting relations.

NodeCommMulticast

Represents a Multicast message sent to a given number of PEs.

NodeCommBroadcast

Represents a message broadcasted to every PE.

To establish the parent/child relationships between the different nodes, the mutual registration pattern suggested by K. Henney is followed [10] which guarantees to keep the bidirectional relation consistent. However, the *linkChild* and *linkParent* methods are redefined so they can enforce compile time type checks for the valid parenthood relations instead of relying on type abstraction at the root level and raising run-time exceptions on incorrect assignments, as would be more common in most object oriented designs.

In addition, no specific attribute is reserved to mark the visited nodes during a graph traversal, as that would consume additional storage and have a potentially shorter lifetime than the graph structure. The algorithms traversing the graph can make use of data within the *extendedNodeData* attribute or a external structure to keep track of the node marking. Ways to traverse the graph can be found on section 4.4.2.

4.4.1 Generation

The graph builder is responsible of reading the log data and transforming it into the graph structure representing the profiled application's flow. It uses Projection's own *GenericLogReader* module, which decodes the log data into readily usable objects representing each log entry.

A Charm++ application can yield log files totaling a considerable size for relatively short running times if a large number of events are being logged. Easily hundreds of MB worth of compressed log data can be generated in just a minute on applications where the logged events aren't carefully defined by the application itself through the logging API.

To be able to cope with this large volume of data as well as to limit the size of the output graph to something that can be usable by a human viewer, the user is given the option to provide information to the tool to restrict the data samples to certain time range and to only consider event related to some of the application's entries. This allows discarding a substantial amount of information early in the process, reducing the time the builder will require to traverse the logs.

In addition, the builder is also designed to work in parallel in computers with multiple processors, exploiting the fact that each PE generates a different set of log files. This also contributes in making the log reading and graph building process quicker, particularly considering that a significant bottleneck of the entire process if the decompression stage of the log decoding, which needs to be performed on the entirety of the file even if most of its entries end up being discarded. Taking into account that this decompression process does not incur in inter-processor communication it results in a time reduction virtually proportional to the number of available processors.

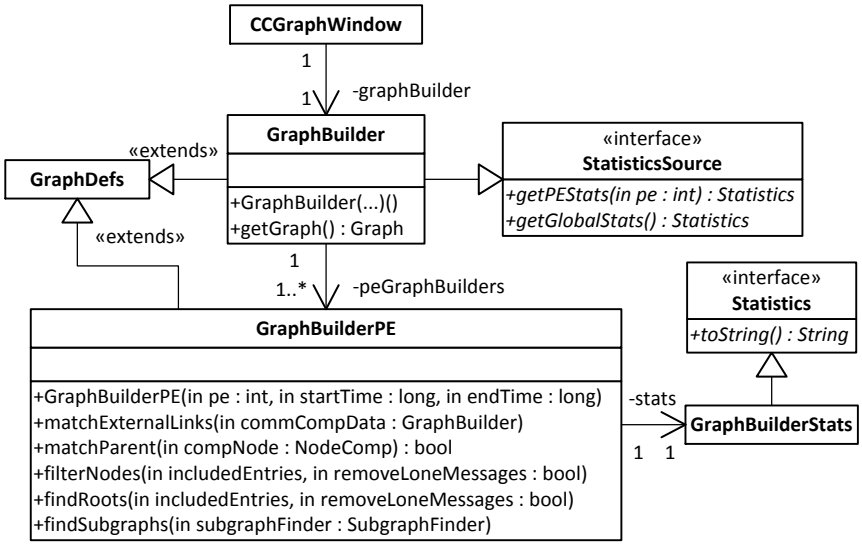


Figure 4.4: Diagram of the internal Communication and Computation Graph's building classes

The reader can appreciate a part of this parallel design from the class diagram on Figure 4.4 the class as a *GraphBuilder* object uses a collection of as many *GraphBuilderPE* instances as PEs the instrumented application was run on. Each of this per-PE builders loads, processes and holds information originated on the PE it's related to, offering a number of methods for them to interact between them and their parent *GraphBuilder* during the building process, which is composed of a mix of p arallel and sequential stages.

As depicted in Figure 4.5, the first step consists in reading the log data, allocating the nodes and creating the links between nodes belonging to the same PE. This first step executes in the per-PE jobs concurrently through the use of a job queue that runs as many simultaneous jobs as processors are available to the tool. Once all the jobs have been completed the following step establishes the cross-PE node links. Its completion is followed by the execution of two more steps which are sequentially executed: filtering the nodes according to the user settings and performing the necessary work to finally encapsulate the graph within an object.

4.4.1.1 Log Reading and Intra PE Matching

The first step within the graph generation process consists in retrieving the data from the application logs and doing an initial treatment of it. These logs can be generated in plain text, binary or compressed binary formats depending on the options used when running the instrumented application with the last format being the default and more prevalent one. To access the logs, the

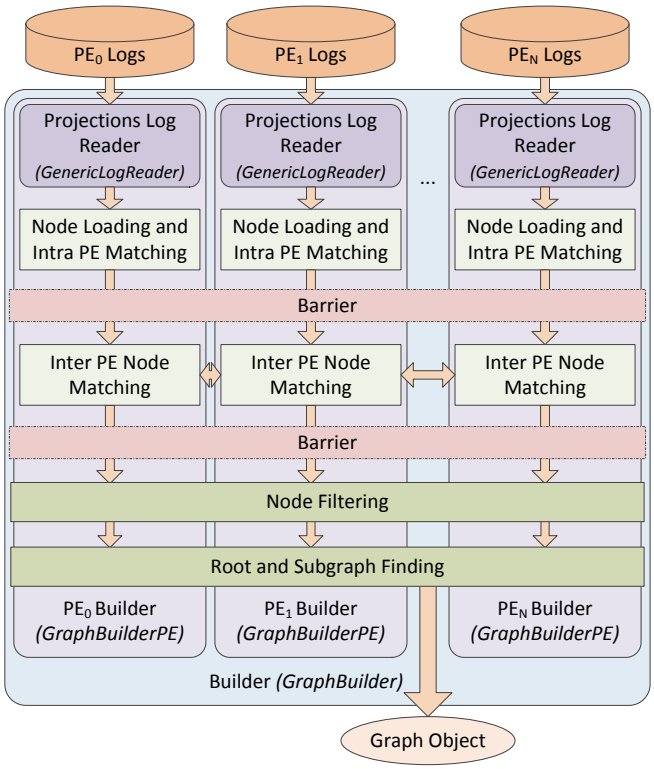


Figure 4.5: Overview of the parallel graph building process

builders use Projection's own *GenericLogReader* module, which transparently decodes the log data into readily usable objects representing each log entry.

These sequences of decoded log entries need to be converted to the graph structure by finding the relations between them. Simplifying, the execution of a chare's entry results in a BEGIN_PROCESSING log entry when it begins the execution of the code and a matching END_PROCESSING one once its execution concludes. Code that attempts to execute another message result in the creation of a message, which yields one of the different CREATION log entries, which will trigger the execution of a potentially remote entry.

Constructing the graph requires matching the begin and end processing entries to complete the information within a computation node as well as finding which message creation triggered the execution of a particular chare entry or which entry created the message.

Considering that this process is executed concurrently for the logs of each PE, some of the nodes that need to be matched on a different PE might have yet to be created. To prevent this, the matching and linking of nodes belonging to different PEs is performed once this first step has been completed for all PEs. This allows for a simpler threading model and prevents the need of

synchronization for any of the operations performed during this stage of the process. This approach, however, slightly raises the process' memory requirements and makes a stream-like implementation unfeasible. Algorithm 1 describes it.

The lookups that need to be carried out are implemented through a hash tables, which make for the additional space that this implementation requires, which grows with the number of open communication and computation nodes accumulated. Said hash tables contain just references to the objects that make for the actual result, but any of the result's nodes may be modified or linked, requiring the entirety of the result data to be kept allocated.

4.4.1.2 Inter PE Node Matching

Once the creation of nodes with their PE-local links has been completed for every PE, the building process can proceed to find and create the links between nodes belonging to different PEs. These links will always be from communication nodes to computation ones, reflecting Charm++'s message based communication model. Waiting for all nodes to be created on the previous step before attempting this simplifies the parallelization of the process as it guarantees their availability for matching.

The matching, as described in Algorithm 2, is performed using a simple lookup of computation nodes on the origin, this lookup is performed through a hash table. Some synchronization takes place at node level to prevent data corruption if multiple computation nodes are linked to the same communication parent concurrently. However, the probability of this happening simultaneously is low and the protected operation (creating the link) quick. Therefore little performance penalties should incur from said synchronization.

4.4.1.3 Node Filtering

While the time range information can be used to discard log data before it's added to the graph structure the same cannot be done for the filtering of data that will translate into nodes within that time range, which is the case when filtering nodes according to the method entries they are related to. The reason for this being that the matching process that determines the application flow relies on this information. The entry filtering needs to be performed after this flow has been found, so the nodes can be removed but the relationships between their parents and children preserved.

The filtering process uses an implementation based on Algorithm 3 which is executes sequentially. It performs several traversals which remove the nodes related to entries that the user

```
C ← empty set that will contain the closed communication nodes
P ← empty set that will contain the closed computation nodes
OC ← empty set that will contain the open communication nodes
OP ← empty set that will contain the open computation nodes
foreach log entry e ∈ CURRENT_PE do
    switch e.type do
        case BEGIN_PROCESSING
            np ← new NodeComp(e)
            lc ← np
            OP.insert(np)
            if e.pe = CURRENT_PE then
                if e.eventID ≠ MESSAGELESS then
                    /* e was triggered by a message                    */
                    nc ← OC.match(e.eventID = nc.eventID ∧ e.entryID = nc.entryID)
                    if ¬nc.hasMissingChildren() then
                        /* All of nc's children have been linked       */
                        OC.remove(nc)
                        C.insert(nc)
                else
                    lc.linkChild(np)
        case END_PROCESSING
            np ← OP.match(e.eventID = np.eventID)
            np.addInformation(e)
            OP.remove(np)
            P.insert(np)
        case CREATION /* Simple message creation                       */
            nc ← new NodeCommSimple(e)
            lc.linkChild(nc)
            OC.insert(nc)
        case CREATION_MULTICAST /* Multicast message creation          */
            nc ← new NodeCommMulticast(e)
            ... /* follows as in a simple message                      */
        case CREATION_BCAST /* Broadcast message creation              */
            nc ← new NodeCommBroadcast(e)
            ... /* follows as in a simple message                      */
```

Algorithm 1: Pseudo-code describing the (per-PE) internal node matching.

```
/* The sets C, P, OC and OP preserve their state from Algorithm 1 */
foreach node np ∈ OP do
    builder ← GraphBuilderPE[np.invokingPE()]
    nc ← builder.OC.match(e.eventID = nc.eventID ∧ e.entryID = nc.entryID)
    nc.linkChild(np)
    builder.OC.remove(nc)
    builder.C.insert(nc)
    OP.remove(np)
    P.insert(np)
```

Algorithm 2: Pseudo-code describing the (per-PE) cross-PE node matching.

selected not to instrument and link their parents to the children the removed nodes were originally linked to. Considering that these traversals access and modify nodes and edges belonging to any of the PEs, attempting several concurrent traversals would result in difficult to prevent racing conditions. However, this process is devoid of disk access and treats a small fraction of the information that was originally available within the logs, so its running time should be comparatively small and not have a large impact on the overall performance of the graph generation process.

4.4.1.4 Root and Subgraph Finding

The final step of the graph building process simply performs a traversal from every unvisited node, which allows identifying disjoint portions of the graph and their roots. This information is used to create the *Graph* object, which stores the resulting set of subgraphs and lists the roots for each of them. This is a fairly straightforward process described in Algorithm 4.

This last step of the building process is also performed sequentially. Similarly to the previous filtering step, it relies on several graph traversals that will be visiting nodes created by the different PE builders. Even if the graph needs not to be structurally modified, there isn't a simple way to prevent racing conditions on the marking process if several traversals are attempted simultaneously. But as in the previous step, the execution time will be small compared to the node loading stage.

4.4.1.5 Statistics Collection

During the graph building process, each of the builders collects a series of statistics about the amount of data collected, number of features of the resulting graph and a rundown of events that

```
/* The sets C and P preserve their state from Algorithm 2 within
   each GraphBuilderPE.                                          */
LIE ← set of instrumented entries provided by the user
foreach GraphBuilderPE pe do
    foreach NodeComp np ∈ pe.P do
        if ¬LIE.contains(np.entry) then
            pe.P.remove(np)
            CL ← empty set of children nodes to re-link
            npc ← np.children
            foreach Node npc ∈ np.children do
                if npc.isNodeComm() then /* Child is a communication node */
                    CL.insert(npc.child)
                    npc.unlinkChild(npc.child)
                    pe.C.remove(npc)
                else /* Child is a computation node                        */
                    CL.insert(npc)
                    np.unlinkChild(npc)
            foreach Node npp ∈ np.parents do
                npp.unlinkChild(np)
                foreach Node cl ∈ CL do
                    npp.linkChild(cl)
```

Algorithm 3: Pseudo-code describing the algorithm used for the node filtering step of the graph generation.

might affect the fidelity of the resulting graph. This is data is collected both on a per-PE and overall basis. A detailed description of the data provided can be found within Appendix A.

The class diagram previously shown in Figure 4.4 shows that the builders are related to a series of interfaces and classes designed for the purpose of collecting and providing statistics. The interfaces *Statistics* and *StatisticsSource* defined in *eduard.projections.gui* allow for any component that collects statistics on a per-PE and global fashion to display them on a re-usable UI component built for this purpose: *StatisticsPanel*.

4.4.2 Traversal

To traverse the graph in a way that abstracts the actual implementation, an interface (*GraphVisitor*) is provided which can be used in conjunction with *GraphTraverser* to perform Depth-First or Breadth-First traversals of a graph based on the defined structure. This interface

```
/* The sets C and P preserve their state from Algorithm 3 within
   each GraphBuilderPE.                                          */
g ← empty graph container
foreach GraphBuilderPE pe do
    foreach node n ∈ pe.P ∨ n ∈ pe.C do
        sg ← empty subgraph container
        if ¬n.isMarked() then
            n.mark()
            foreach Node nc connected to n do
                nc.mark()
                if nc.isRoot() then
                    sg.insert(nc)
        g.insert(sg)
```

Algorithm 4: Pseudo-code describing the algorithm responsible of finding subgraphs and roots to generate the resulting graph object.

is designed just to traverse joint subgraphs out of the overall, possibly disjoint graph.

4.4.2.1 Component Description

GraphTraverser

The generic class *GraphTraverser* can be used with any type (NodeType) that implements the simple *VisitableNode* interface, allowing re-use of the traversing algorithms, changes in the node definitions or usage of the methods just for a particular branch within the node type hierarchy. It offers the methods *traverseBFS* and *traverseDFS* which non-recursively implement both traversal algorithms given a *GraphVisitor* as an input.

GraphVisitor

A *GraphVisitor* object provides the information to start and carry on a traversal: defining the starting node, marking method and actions to take during node visits. As with the *GraphTraverser* class, the interface is generically defined to support graphs based on vertices that implement the *VisitableNode* interface. The interface defines the following methods:

```
NodeType getStart();
```

Provides the node where the traversal should be started.

```
void makeVisited(NodeType node);
```

Marks the given node as visited.

```
boolean isVisited(NodeType node);
```

Checks if the given node is visited. Note that the behavior of this method can be adjusted so a node is visited multiple times if the algorithm requires it.

```
void setCurrent(NodeType current);
```

Can do the work of an actual node visit. Will be invoked on every node, including the starting one regardless which edge direction was used to reach them.

```
void doVisitChild(NodeType childNode);
```

Can do the work of an actual node visit. Indicates that the node was reached following a link forward. Won't be invoked on the starting node.

```
void doVisitParent(NodeType parentNode);
```

Can do the work of an actual node visit. Indicates that the node was reached following a link backwards. Won't be invoked on the starting node.

```
void doVisitedChild(NodeType childNode);
```

The traversal cursor is peeking a node that has already been visited and will be skipped. The node was reached following a link forward.

```
void doVisitedParent(NodeType parentNode);
```

The traversal cursor is peeking a node that has already been visited and will be skipped (not visited again). The node was reached following a link backwards.

```
void done();
```

Signals the end of the traversal.

The usage of these methods is extremely situational. A traversal can be achieved by implementing just a few of them. To simplify the usage of the interface partial implementations are provided which are described in the modules that follow.

AbstractGraphVisitor

Implements the methods defined on the *GraphVisitor* interface with empty bodies. The methods that need to be implemented by an algorithm need then to be overridden, but eases the burden of creating empty bodies for the methods that the algorithm doesn't implement. The marking methods (makeVisited, isVisited), aren't defined, however, as these are essential to

the correct execution of the traversal. Additionally, it defines an attribute and constructor to store the starting node.

DefaultGraphVisitor

Extends *AbstractGraphVisitor* class and defines an actual implementation of the marking methods. It relies on using a *HashSet* structure to perform the node marking. It's not the most performance or memory efficient way to achieve such a thing, but it doesn't rely on the *extendedNodeData* which might already be used for a different purpose. Extending this class is not a recommended way to implement traversals, but it's convenient for rapid algorithm development and testing.

Graph

Graph, aside from being used as the top level container for the graph structure contains two class methods (*traverseBFS* and *traverseDFS*) that can be used to conveniently perform the traversals, without the need to instantiate a new *GraphTraverser* and taking a *Node* as the traversal starting point.

4.4.2.2 Example Traversal

An example of how to perform a traversal on the graph structure that illustrates both the usage of the interface the benefits of the abstraction follows. It first defines a *GraphVisitor*, which for simplicity is implemented by extending the *DefaultGraphVisitor*, which eliminates the need of having to implement the node marking methods.

```
public class ExampleTraversal extends DefaultGraphVisitor<Node> {

    ExampleTraversal(Node startNode) {
        super(startNode);
    }

    @Override
    public void setCurrent(Node current) {
        // Do something on visit
    }

}
```

Finally, a root is selected from the graph, the visitor is instantiated and used to invoke a Breadth-first traversal.

```
static void traverseFirstSubgraph(Graph g) {
        // Obtain a root from the first subgraph
        Node root = g.getSubgraphs().get(0).firstRoot();
        // Traverse the subgraph starting at that root
        Graph.traverseBFS(new ExampleTraversal(root));
}
```

4.5 Visual Graph Representation

User Interface related modules are contained within the package *eduard.projections.Tools.CCGraph*. They allow the user to set the desired instrumented time range, and to choose from a list of the application's entries which are the ones to be instrumented. This information is then used on the graph builder to load the log data and generate the graph with these restrictions. The user interface then uses this graph to generate a visual representation of it, which can be adjusted with some layout options provided to the user.

4.5.1 User Interface

Once the graph has been generated, the user is presented with the window with the structure depicted in Figure 4.6, containing the visual representation of the resulting graph on a tab and the statistics of the building process in another using the components described in Section 4.4.1.5. This visual graph representation is implemented within the *GraphPanel* component, which is accompanied by an additional panel allows adjusting the component's options, while a third displays the available information for the currently selected node.

The visualization options made available to the user include, the node layout, the graph's scale, the edges' shape and the locking state of the graph's nodes; all of which are explained in the following subsections. It's also possible to regenerate the graph or save it as an image from the window's drop down menus.

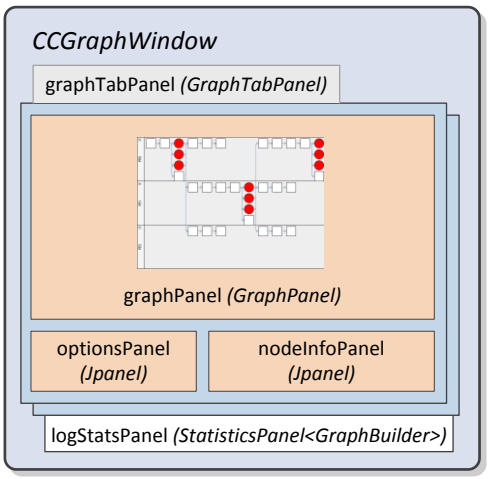


Figure 4.6: Structure of the Communication and Computation graph tool's window.

4.5.2 The JGraphX Drawing Library

JGraphX[15] is a Java Swing graph visualization library available under a BSD license. Out of the freely available Java based graph visualization libraries it was deemed the most flexible, although it comes at a cost of increased complexity on its usage. This flexibility was a necessity, as no other library seemed to offer the capability of displaying the desired swimlane based layout.

As of version 6, however, the library has been rewritten as a port of its commercial JavaScript counter-part, mxGraph. This new version, at the time of this writing, suffers from a severe lack of documentation limited to a single page manual and a few Java and JavaScript based examples. Also, probably as a vestige of its JavaScript origins the library heavily relies on untyped parameters that need to be casted. These factors made the learning process steeper than expected, as achieving the desired behavior mostly became a trial and error process.

4.5.3 Style Manager

JGraphX's components are styled by referencing styles stored within the graph's stylesheet. Considering the number of potential node styles required for the desired output (256 shades for each of the possible node shapes) the tool implements a dynamic style creator: *StyleManager*. When a style is requested for a particular node type and color intensity it creates the style if not present within the stylesheet and returns its reference. That way just as many different styles

are created as the current graph actually needs.

The same class is also used to create and access the non dynamic styles such as those of the swimlanes and edges.

Node Style	Description
	Computation
	Unicast
	Multicast
	Broadcast

Table 4.1: Node style for each node type.

4.5.4 Node Placement

As previously discussed on section 4.2, the aim is to achieve a graphical output where the graph's nodes are distributed in a series of swimlanes, one for each PE. There are several criteria that could be chosen to decide the horizontal placement of the nodes within the swimlanes. Two methods are proposed to perform such node placement, which are described in the subsections that follow.

The tool uses an abstract class, *GraphLayout*, as a basis for different layout implementations. Creating a new layout should be done by extending this class and implementing the draw method, which provides a created visual graph component and the graph data structure to be drawn among other information that can be used to implement the actual node drawing within the swimlanes. Then the layout can be applied by invoking the *setLayout* function on the *GraphPanel* component.

By default, JGraphX's graph drawing component allows the user to edit the displayed graph by moving, adding or removing the edges and nodes. For this tool's purpose these features have been disabled, but the ability to move the nodes has been kept as an option as it might be useful for the user to have the ability of manually re-distributing the nodes in situations that the graph view might otherwise be ambiguous.

4.5.4.1 Topological

The solely topological node layout aims for a simple implementation that separates unrelated application flows (disjoint subgraphs) and draws the nodes within each of them using their topological order to determine their horizontal positions. Separating the independent flows should allow for a clearer examination of each of them. The layout is implemented within the *TopologicalSplitLayout* class.

Placement Rules

- The nodes within a disjoint subgraph will be horizontally sorted according to their topological depth.
- Nodes that share the same topological depth within a disjoint subgraph will be vertically aligned (derived from the previous rule).
- Different disjoint subgraphs won't horizontally overlap.
- The different disjoint subgraphs will be sorted according to the minimum *time* value of their root nodes.

Algorithm

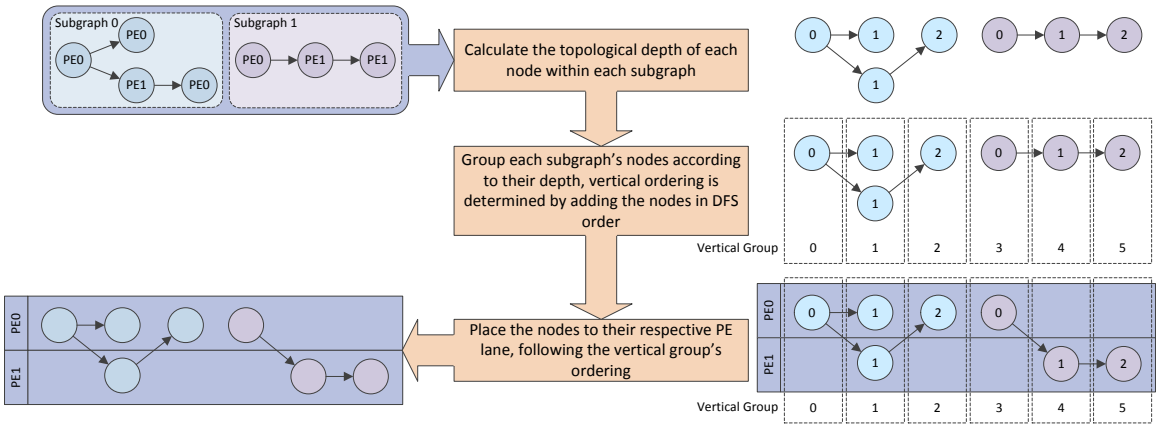


Figure 4.7: Node placement process for a purely topological view.

As shown in Figure 4.7 the algorithm calculates the topological depth of each node within each of the input's subgraphs, which are also sorted by the time value of their first root. The nodes sharing a topological depth within a subgraph are to be vertically grouped, which is done with a DFS traversal of the subgraphs, placing the firstly visited nodes on top of

each group's space reserved for a particular PE. This vertical ordering criteria prevents edge crossing within an application's flow for tree-like graphs.

Complexity Analysis

A fixed number of traversals are required for the realization of the algorithm, which come at a cost that linearly grows with the number of edges ($O(|E|)$). However, the topological sorting stage has a higher complexity than that of a traversal at $O(|V|+|E|)$, raising the overall cost of the algorithm to the cost of a topological sort. In the degenerate case where the input consists of a large number of small subgraphs the cost of the subgraph sorting might prevail, resulting in $O(|S|\log|S|)$ where S would be the set of subgraphs.

The spatial complexity of both a traversal and a topological sort is of $O(|V|)$, leaving the algorithm's space complexity to the number of the graph's vertices.

4.5.4.2 Topological and Timed

This layout attempts to provide a more chronologically accurate overview of the application's events by not splitting the different application flows and considering also timing information for the horizontal placement of the nodes. The nodes are to be horizontally placed according to their timestamp, but in the cases where several nodes share a timestamp topological order will be used to horizontally separate them. This implementation of this layout can be found in the *Topological-TimedLayout* class.

Placement Rules

- All the nodes will be horizontally sorted according to their *time* value.
- Nodes that share the same *time* value will have the horizontal order among them determined by their topological depth within their respective disjoint subgraph.

Algorithm

The algorithm, as shown in Figure 4.8, groups and makes a per-PE chronological sort of the nodes (if the timestamp for two nodes is the same then the in-log order is used). It then sequentially links the nodes within each of the PEs with a special type of link representing their chronological dependencies. With these new links in place a topological sort is performed considering both the topological and chronological links, vertically grouping the nodes with equal depths as in a subgraph of the topological-only placement algorithm.

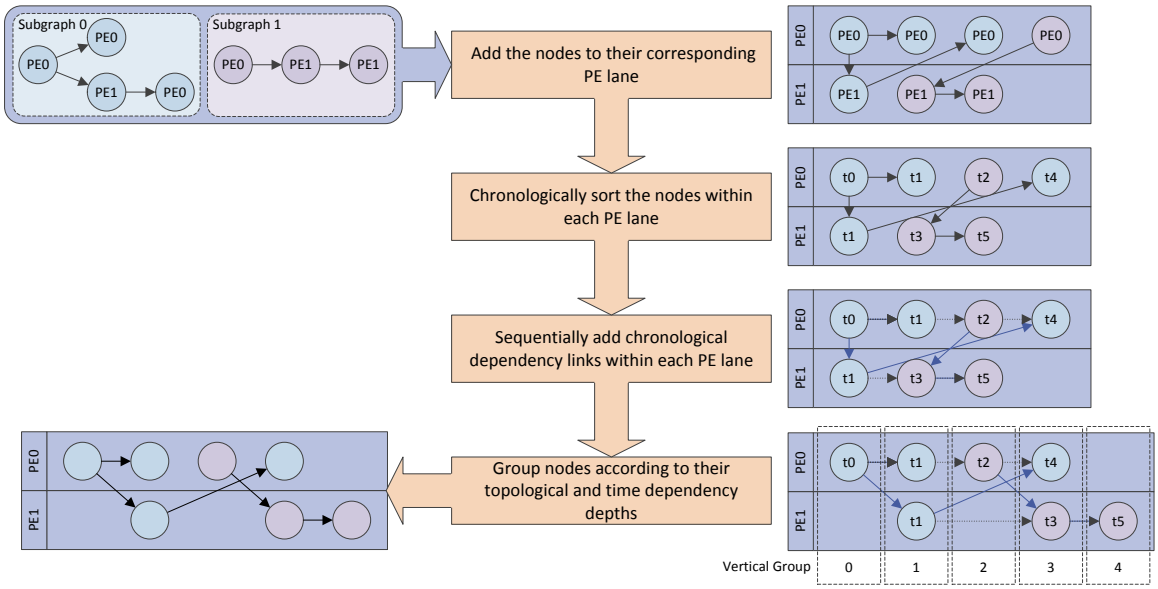


Figure 4.8: Node placement process considering both topological and chronological constrains.

Complexity Analysis

The computational complexity is dominated by the chronological sorting step, which is performed as an average $O(n \log n)$ sort. The rest of the algorithm's stages consist of a set number of traversals with a linear $O(|E|)$ complexity and a topological sort of cost $O(|V| + |E|)$.

Space wise, the complexity is of $O(|V|)$ considering that the space complexities of the traversals, sorting and the number of chronological edges created is of $O(|V|)$.

Chapter 5

Visualization of PE-to-PE Communication

To evaluate a new job allocation policy's impact on an HPC application's communication performance predictive models have been developed, which can allow to estimate a different task allocation algorithm's performance impact for a particular application. These models can consider the network topology and the application's communication behavior to predict the impact that different number of hops that each communicating pair of nodes would need to go through under a new allocation policy in the application's throughput.

Such knowledge of an application's communication behavior might not be readily available, however, but can be inferred by instrumenting the application's communication and building a more abstract model of its communication patterns from it. Nonetheless, a careful look at how an HPC application is communicating might also reveal possible bottlenecks on how the application uses the network.

5.1 Goals

This chapter describes a method that provides an analysis of an application's communication by focusing on the following points:

Quantification of the Communication between Pairs of PEs

Such quantification can be used in prediction models for different task allocation algorithms so estimations can be made of an application's throughput when making use of such algo-

rithms.

Facilitate the Inference an Application's Communication Patterns

Known communication patterns should yield characteristic results when it comes to a PE-to-PE quantification of the communication. However, gasping these patterns from the resulting table of numerical values might prove difficult. A graphical output of such results through surface and density plots should make it potentially easier for an user to notice such patterns.

The realization of these two goals can provide information about an application's communications demands, which can be used to evaluate the suitability of a particular job allocation methodology under the capabilities of a particular interconnect system and its topology.

5.2 Approach

Quantifying the information can be easily performed by counting the number of messages sent between each pair of PEs as well as accumulating their sizes. The resulting numerical values are then displayed and allowed to be exported to other tools for further processing.

The tool also allows plotting the retrieved values in different ways in an attempt to make the communication patterns as noticeable as possible to the user. These plotting methods include a tridimensional surface plot and a flat density one.

5.3 Structure

The tool's structure, as depicted in Figure 5.1, closely mirrors the one employed on the Computation and Communication Graph tool detailed in Section 4.3. It uses a three layer design divided in the UI related components, the ones that gather and represent the internal data, and Projection's own shared modules for log access.

As with the graph tool, the user initiates the tool by providing the analysis range and entry filtering information. This information is used by the data collector to read the logs and filter part of the read data while generating the resulting matrices of accumulated data. The UI then formats and presents this matrices in both table and several plot types. The data collection process also generates some statistics that provide some additional insight to the gathered data.

This tool relies on far simpler internal data than the graph tool does, however. It just stores a number of matrices that accumulate the desired communication data which are wrapped together

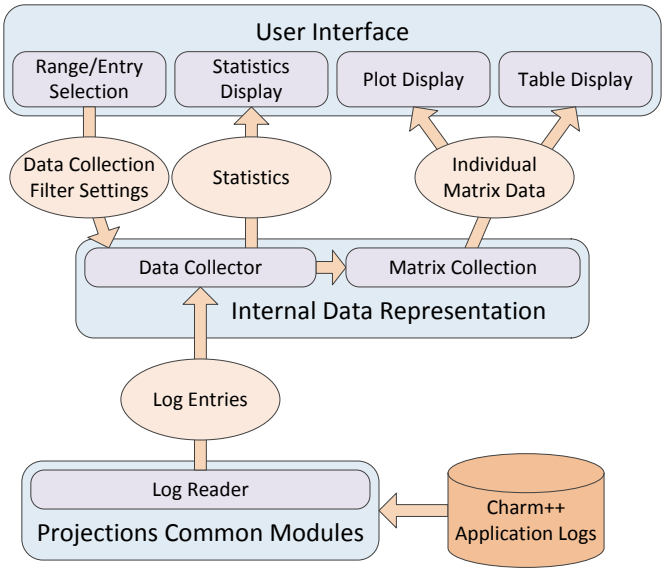


Figure 5.1: Conceptual view of the PE-to-PE Communication tool structure, illustrating the flow of information between components.

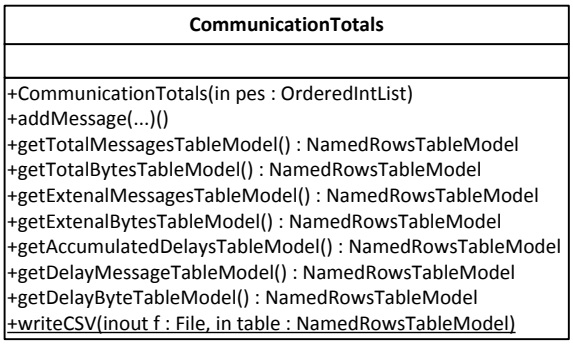


Figure 5.2: CommunicationTotals class used in matrix data gathering, encapsulation and provision.

in the *CommunicationsTotal* object shown in figure Figure 5.2 that allows accessing the different results.

5.4 Data Collection

The module *CommTotalsGenerator* covers the communication data collecting role, by reading the log data and accumulating the desired attributes, relying on Projection's *GenericLogReader* for the reading and decoding of the log data.

This process is executed in parallel so the logs of several PEs can be concurrently read and processed on machines with multiple processor cores, reducing the impact of the time consuming log decompression process for very large log files.

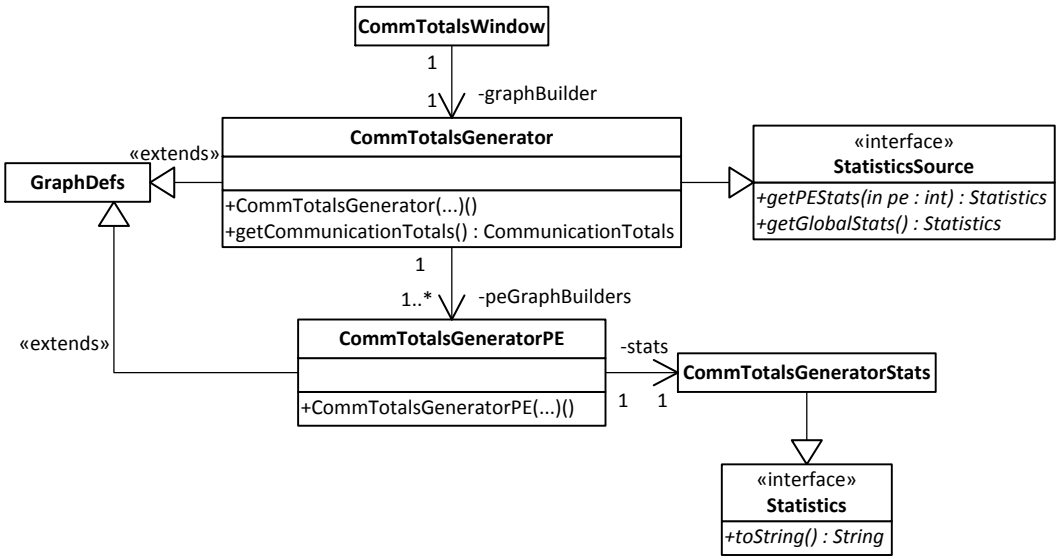


Figure 5.3: Diagram of the communication data gathering classes.

As depicted on class diagram in Figure 5.3 and Figure 5.4 the class *CommTotalsGenerator* creates a series *CommTotalsGeneratorPE*, one per each PE of the instrumented application, which load and process their PE's log data and update a series of shared tables. Each of these generators run as part of a job queue which executes as many of them concurrently as processors are available to Projections. This implies the need of some synchronization on the result tables to prevent the issues associated with concurrent access, which the *CommunicationsTotal* class implements.

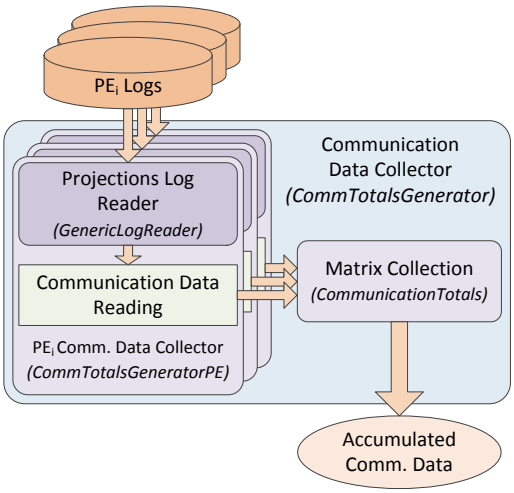


Figure 5.4: Overview of the parallel communication data gathering process.

The processing of a communication log entry to account for a message or its size is straightforward, as it's just a matter of checking its timestamp against the user's supplied sample range and set of instrumented entries and update the byte and message counts on the matrix positions matching the source and target PEs.

Collecting time information to provide latency data would also be desirable, however the limited granularity. Even though the logs store the timing information as 64 bit integers representing milliseconds, the timing values supplied are rounded to the second. Attempting to obtain useful latency information with such data would hardly provide useful results. Additionally it would significantly increase the complexity of the data gathering process as the entries where a message is sent would need to be matched to the entries where these messages are received. This would likely impair the tool's ability to work as effectively with logs of arbitrarily large sizes.

The resulting *CommunicationTotals* containing the accumulated data is then used by the UI components to access the collected data using the implemented extension of Java's *TableModel* described on Section 5.5.3.

5.4.1 Statistics Collection

The generators used in this tool also rely in the set of statistics collection interfaces, as shown in Figure 5.3, and the UI component to display them, *StatisticsPanel*, described in Section 4.4.1.5.

Considering that both Projections-based tools developed as part of this thesis' work rely

on a similar presentation of the statistics, this implementation allowed for the benefit of re-using the related components for both of them. A description of the statistics provided by this tool can be found within Appendix A.

5.5 Visual Data Representation

The tool's user interface components allow the user to define a time range from which to sample log events from, as well as choosing from a list of the application's entries which of them are to be instrumented. These restrictions are passed to the data collector, which read and process the log data, accumulating communication information on a set of tables. The user interface then retrieves these tables to show the data either numerically or in plot form.

5.5.1 User Interface

User Interface related modules are contained within the package *eduard.projections.Tools.CommunicationPEtoPE*, which is shared with part the modules belonging to the internal representation, as the tool is reasonably compact. An exception to this would be the table model described in Section 5.5.3 which has been kept separate for re-use.

After the data has been gathered, the tool displays a window described in Figure 5.5 that presents the gathered data within a tab and the statistics collected during the process in another, the later being based on the components for statistics collection and display described in Sections 4.4.1.5 and 5.4.1.

The tab containing the communication data itself is implemented in the *CommDataPanel* component, which allows selecting the data to be displayed (source) and its representation. For the display of the data a different panel is created for each possible source, with each of these containing a *JSurface* panel for the graphical representations and a *TablePanel* for numerical visualization of the raw data within a table.

The data source panels as well as the visualization ones are encapsulated within a *CardLayout*, which displays just one of them at a time while hiding the others according to the options marked within the selectors. This makes for a simple implementation that allows preserving the views on the hidden panels.

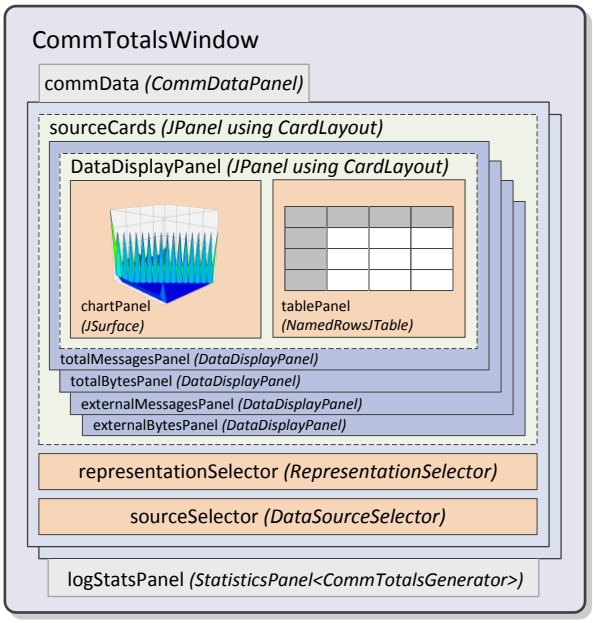


Figure 5.5: Structure of the PE-to-PE Communication tool's user interface.

5.5.2 Data Plotting

Most of Projection's tools use the library JFreeChart [14] for chart display. JFreeChart, however, does not suport tridimensional plots, which was deemed a desirable representation for the communication data that this tool gathers.

To overcome this limitation the tool makes use of the open source plotting library surfaceplotter[8] which is compact and allows generating surface or density plots. The library has been modified to support custom labeled axes which were required to label non contiguous sets of PEs.

(a) Density Plot (b) Contour Plot

Figure 5.6: Extracts of different plot types generated with surfaceplot using the same data.

The tool provides a flat density plot, which is better suited in some circumstances such as printed media. The surfaceplotter library also has limitations of its own, though. Its implementation of the density plots colors the area using average values between markings, taking the four values

where the markings cross as the data samples, obscuring the actual sample values at these points, resulting in plots like the one shown in Figure 6.1(a).

Alternatively it offers a different plotting method: contour, which uses interpolation to mask the lack of sample data between markings. While this has the drawback of the introduction of inaccuracies implicit to such process it preserves the actual values at the sample points, which in Figure 6.1(b) reveals a significantly different value at point $(0,0)$. The tool also offers this plotting mode as to overcome the deficiencies of the density plot.

5.5.3 Representing and Displaying Tables

Java's Swing libraries include a useful component to represent tables, the *TableModel*. It allows abstracting the inner implementation of any component that makes sense to access as one would access a table. An actual implementation of the interface can then choose how to store the data or even calculate it on the fly when queried. This interface is also used to supply data to Java's UI table component: *JTable*.

These two components also allow to conveniently define and render column headers within the tables, but surprisingly offer no support for row headers. To be able to properly represent and display the origin PE and the target PE a cell corresponds to an extension of the model was implemented, *NamedRowsTableModel*, which mirrors the structure and components its original counterpart is based on, as shown in Figure 5.7.

The *NamedRowsTableModel* interface simply adds two new methods to *TableModel*, which allow querying the row header labels and the title for the cornering cell where the column and row headers intersect. *AbstractNamedRowsTableModel* extends *AbstractTableModel*, adding a default implementation of the newly introduced methods. Finally, the *DefaultTableModel* implementation offering a basic working implementation based on vectors, *DefaultNamedRowsTableMode* extends it through object composition to achieve the same functionality on the new model.

On the user interface side, *NamedRowsJTable* extends *JTable* to offer the same functionality with the new model. It implements the drawing of the row headers by transparently creating a new column with cells that borrow their styling from the column headers.

The previously described *CommunicationTotals* component allows access to the resulting tables through this new implementation of the *TableModel* component which is then used on the tool UI's *NamedRowsJTable* and *JSurface* view components, the later being extended to support axis labeling through the headers supplied by the model. Additionally the *CommunicationTotals*

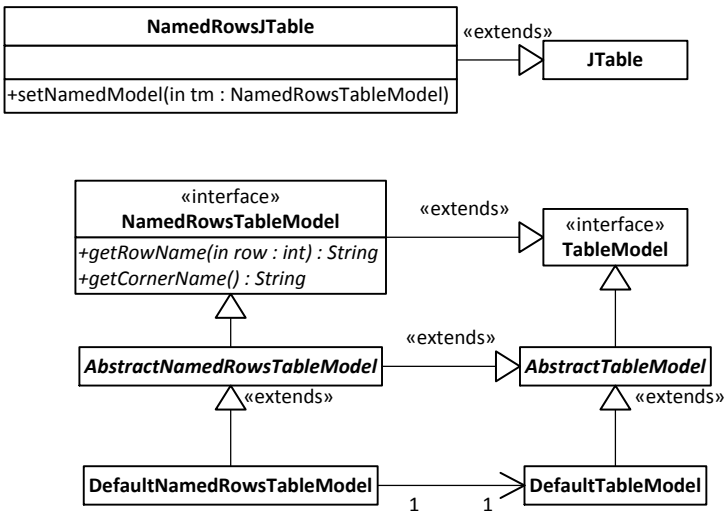


Figure 5.7: Class diagram illustrating the extension of Java's TableModel to support tables with titled rows as well as the extension of the JTable component to visualize them.

class provides the static helper function *writeCSV* to store a given *TableModel* to a file in the simple Comma Separated Values format.

The implementation of this new model is a re-usable and self-contained (outside of Java's standard Swing libraries). It can be found in the package *eduard.projections.gui.table*.

Chapter 6

Synthetic Benchmark

Aside from the addition of two different tools to Projection's portfolio, an independent Charm++ program is also developed for verification purposes: CCBench, a simple synthetic benchmark that allows running an application with configurable communication and computation demands.

When setting up a hardware and software environment for HPC applications, it's desirable to analyze how changes in the setup or its configuration can affect different aspects of an application's performance. During part of the experimental work for this thesis, in particular, there was interest in examining which would be the impact in communication performance of using a Charm++ application enabling the use of shared memory (SMP option) for communication between PEs within the same computing node instead of relying exclusively on network based message passing.

While more elaborate benchmarking methods exist to evaluate both computation and communication performance, that isn't its most important purpose even though it can be used to achieve so to a certain degree. The benchmark is meant to be used as a test suite for the development and validation of tools that work on data extracted from the execution logs discussed in Chapters 4 and 5.

6.1 Goals

This tool aims to complement the two instrumentation tools by providing the following:

Communication and Computation Performance Comparisons

The use of real world HPC applications for benchmarking could easily dilute performance gains or losses that an specific adjustment to an HPC environment might bring. Offering a way independently compare changes in computation throughput, memory bandwidth, communication bandwidth and communication latency (or any weighted combination of them) would allow for more circumstance-specific tests.

Generating Predictable Log Data

Developing and validating the log based instrumentation tools required an application that offered a clean and controllable environment for the generation of logs that would allow to predict the exact number and sizes of the exchanged messages as well as employing different flows of data and test the feasibility of recognizing them on the implemented tools. Accomplishing this can also allow measuring the overheads introduced in both the number of messages and their size.

6.2 Approach

The benchmark is designed to send a controllable number of messages of adjustable sizes to be able to measure the effects of latency and bandwidth on the application's throughput on a given execution environment, as well as allowing the verification of the message counts and sizes from its resulting logs.

It implements two different communication patterns, an all to all communication one which mimics the behavior common in HPC applications and a simple synthetic circular pattern. Both place different stresses on the communication layer, and offer distinct application flows that can then be verified with the implemented instrumentation tools.

Finally, to offer the possibility of creating computational and memory demands, CCBench can perform a number of simple integer based checksums of the arbitrary data allocated within a buffer of adjustable size.

The comparative metric that the benchmark provides is the benchmark's total running time.

6.3 Structure

The program is composed of 2 modules, each of them made up by 3 different files, 2 pure C++ files: the header for the module specification (.h) and the module's body file (.C) as well as the Structured Dagger file describing the inter-PE access to the module's variables and methods (.ci).

- **CCBench** is the program's main module, has the role of reading the command line arguments and setting up the program's environment accordingly through the use of global, shared, read-only variables. It instantiates the workers and starts them as needed, monitors them and finishes the program once they have completed their work, reporting the program's total running time.

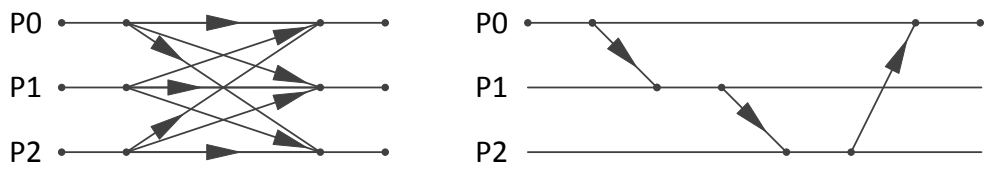
- **WorkerArray** is the program's collection of workers that are responsible of running the CPU bound job and sending data to the next worker/s according to the in-use communication pattern. While modularization principles would suggest implementing the individual worker module and the collection of them separately, Charm++'s design allows declaring a variable as a collection of a particular type of workers and allowing to conveniently interface with them without the need of manually creating a collection interface layer. In all cases as many workers will be created as PEs are allocated for the application.

6.4 Communication Patterns

In addition to adjusting the communication demands of each individual message, the way that these are sent between PEs can have a performance impact on the underlying communication infrastructure. Different communication patterns also allow to validate the capability and recognizability of these in the developed profiling tools.

CCBench offers two different and simple patterns, all-to-all and circular communication. Whichever pattern is selected through the command line arguments will be looped a number of times throughout the benchmark's execution.

Charm++ allows separating the concerns of the actual computational work from the application's flow control through its Structured Dagger (SDAG) notation. This feature allows defining which particular entry invocation a worker will be able to accept at a given point of the application's execution. The benchmark exploits this by implementing the flow of the different communication patterns through this notation, while sharing part of the computational code base.



(a) all-to-all communication (b) circular communication

Figure 6.1: Depiction of the two implemented communication patterns for 3 PE over the course of a single iteration.

Due to a limitation on SDAG where its integers cannot take values larger than that of an unsigned 16 bit integer the patterns are iterated using a double loop. Otherwise, if the application is run with lightweight demands, it can go through all the iterations possible within a single loop in too short of a time for the test's timings to be useful. The number of iterations for both loops (it_{inner} and it_{outer}) can be adjusted through the command line arguments.

6.4.1 All-to-all Communication

In this pattern, each worker scatters a message to all other workers and waits until it has gathered a message from them all before proceeding with computational work for the iteration. It therefore requires a large number of messages. This is actually a realistic communication pattern that repeatedly appears in parallel applications. The implementation performed assumes a situation where the messages to be delivered to each of the PEs are different, which prevents the usage of broadcasting features.

Implementation of Flow Control

In this configuration, the workers start by immediatly sending a message to every other worker through, *sendToAll* which implements the all-to-all communication at the origin and then waits until receiving all the messsages sent to it for that iteration of the pattern, a process which repeats until it has completed until the given number of iterations of the pattern is reached. Then, the worker its end to the main chare through the *finish()* method.

Listing 6.1: Extract of *WorkerArray.ci* depicting a worker's flow in the all-to-all communication pattern.

```
entry void runAllToAll() {
  for (oiter = 0; oiter < nOuterIterations; ++oiter) {
    for (iiter = 0; iiter < nInnerIterations; ++iiter) {
```

```
        serial {
          sendToAll();
        }
        forall [part] (0:nElements-1, 1) {
          when submitPart[iiter](WorkMsg *msg) {
            serial {
              msgBuffer[part] = msg;
            }
          }
        }
        serial {
          workAllToAll();
        }
      }
    }
    serial {
      finish();
    }
  };
```

Volume of Messages

The volume of messages $|M|$ created with this pattern for each iteration is defined by the number of PEs and iterations as:

$$|M| = |PE|^2 \, it_{inner} it_{outer}$$

6.4.2 Circular Communication

In circular communication each node connects to exactly two other nodes, forming a single continuous pathway for signals through each node - a ring. Data travels from worker to worker, with each node along the way receiving a message, performing its computational task and sending a message to the next. This is an unrealistic pattern for an effective parallel application, as it's sequential in nature. However, it provides additional situations to validate the instrumentation tools, as well as allowing to assess the network throughput and latency on a collision free environment.

Implementation of Flow Control

As shown in Listing 6.2 the SDAG code will only allow the first worker to start the messaging chain, which is propagated by the *sendToNext* which handles the one-to-one communication. Once the given number of iterations for a worker is complete, the worker notifies the end of its life to the main chare through the *finish()* method.

Listing 6.2: Extract of *WorkerArray.ci* depicting a worker's flow in the circular communication pattern.

```
entry void runCircle() {
  serial {
    if (thisIndex == 0) sendToNext();
  }
  for (oiter = 0; oiter < nOuterIterations; ++oiter) {
    for (iiter = 0; iiter < nInnerIterations; ++iiter) {
      when submitPart(WorkMsg *msg) {
        serial {
          msgBuffer[0] = msg;
          workCircle();
          sendToNext();
        }
      }
    }
  }
  serial {
    finish();
  }
};
```

Volume of Messages

In this pattern the volume of messages $|M|$ simply grows linearly with the number of PEs and iterations:

$$|M| = |PE|\, it_{inner} it_{outer}$$

Chapter 7

Experimental Results

As mentioned throughout the report, CCBench was developed with the purpose to create synthetic logs for the validation of the instrumentation tools. In this chapter it's put to use to validate and test the effectivity of the implemented tools.

Additionally, to obtain realistic workloads, eight other HPC Charm++ applications are analyzed. The selected applications are actively used at multiple HPC centers or closely resemble such applications. Collectively, these and closely related applications comprise a substantial fraction of all HPC cycles. The communication patterns produced by the selected applications represent a number of different HPC communication types. Some applications themselves encompass several communication types.

In particular, the following applications were chosen (unless otherwise noted, the applications can be obtained from Charm++ Mini-Apps web site [18]): The Fast Fourier Transformation (FFT) is dominated by all-to-all communication from the matrix transposition operations. CharmLU, a linear solver operating on sparse matrices, is representative for applications operating on sparse data sets. NAMD is a widely used parallel molecular dynamics simulation for large bio-molecular systems [2]. We investigate two variants: NAMD-noPME omits the long-range force computation and uses only range limited electrostatics resulting in primarily local communication. Conversely, NAMD-PME includes the long range force computation with additional communication. It adds communication when transforming into vector space and an all-to-all communication stage when performing an FFT. Barnes-Hut and ChaNGa[3] (Charm N-body Gravity) are celestial body simulations with a tree-based implementation. Nonetheless, their detailed realization differs resulting in substantially different communication patterns. OpenAtom, a quantum chemistry simulation framework, has a number of interdependent heterogeneous phases including sparse and dense

FFTs, as well as non-square matrix multiplications. Finally, the adaptive mesh refinement AMR adaptively refines computation through repartition and data migration.

7.1 Validation through the Synthetic Benchmark

When using the graph tool with CCBench, it can properly display both of its communication patterns in a way that allows the user to infer the them, while providing detailed information of the application's behavior.

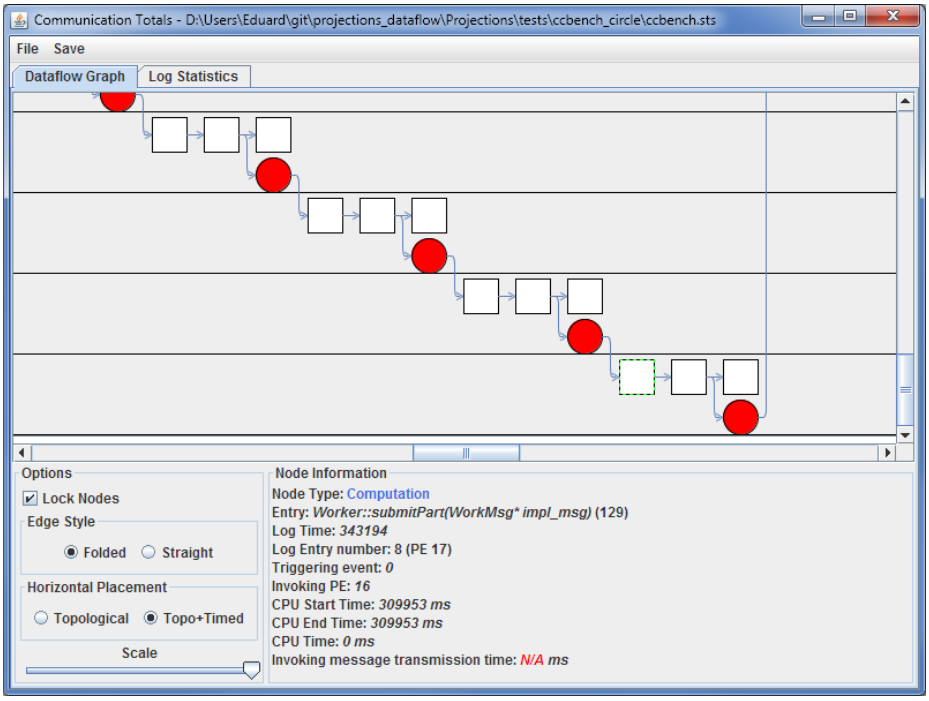


Figure 7.1: Graph tool displaying the graph resulting from running CCBench with the circular communication pattern on 18 PEs.

Figure 7.1 shows an extract of the circular pattern, particularly the step where the last PE messages the first to start the ring again in a new operation. The three computation nodes shown on each step reflect the invocation of the *submitPart* entry (see Chapter 6) while the tailing two represent flow control code that has been generated as result of the SDAG flow notation. These additional nodes could have been filtered out if so desired when loading the graph. The window also displays the information of the selected computation node. Among the displayed node information the application attempts to calculate the latency of the message that invoked the node's entry, however missing log data on that message prevents it from being calculated.

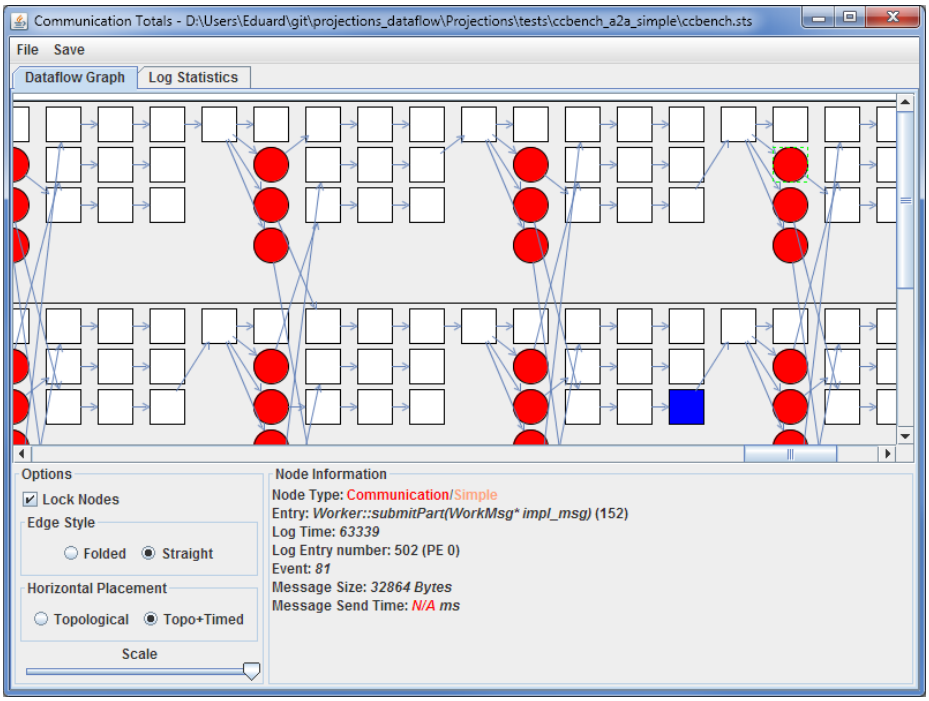


Figure 7.2: Graph tool displaying the graph resulting from running CCBench with the all-to-all communication pattern on 3 PEs.

CCBench's all-to-all communication pattern is shown in Figure 7.2. While the tree-like portrayal of the application's flow does not show the data dependencies, it's possible to make an educated guess and realize the points where the application is gathering data before proceeding to the next step, allowing to infer the pattern, although it's less trivial than the circular one. In here a communication node is selected, which is bound to invoke the *submitPart* entry. Even though the logs are supposed to supply the time when a message was sent, the parameter is consistently marked as invalid, which makes it unfeasible to use it to calculate message latencies. This determined the decision of coloring the communication nodes according to message size instead of transfer time. The computation nodes, however, are colored according to the time they take to run, but face a different problem: the rounded to the second values of the timestamps. In the figure this results in just a single fully colored node (1 second of computation time) while the rest are depicted fully unsaturated (0 seconds) when actually all of them should have some value in between and be colored accordingly, instead of suggesting a false outlier.

7.1.1 Tool Limitations

For simple applications yielding a small volume of log events in the tool accurately portrays the application's flow and presents it in a way that is intelligible to the user, providing timing information about the events as well as relative communication or computation costs.

Unfortunately, the tests with the real world applications consistently result in graphs far too large for a human user to be effectively be able to track its flow effectively, even when severely limiting the sample ranges (see Figure 7.3 for an example). Further tests with other tools are not shown due to their similar issues.

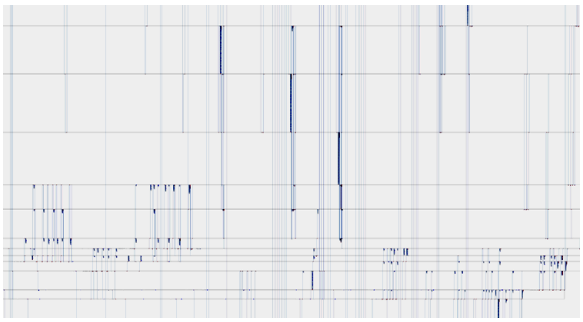


Figure 7.3: A scaled down fraction of the graph resulting from a 100ms sample of a NAMD execution.

This issue appears long before the tool's technical boundaries are reached, as the visualization grows too large to be understood before the graph or the representation's memory footprint are a cause for concern. This implies that it's currently an issue with the way information is portrayed to the user rather than with the process of generating the application's flow graph itself.

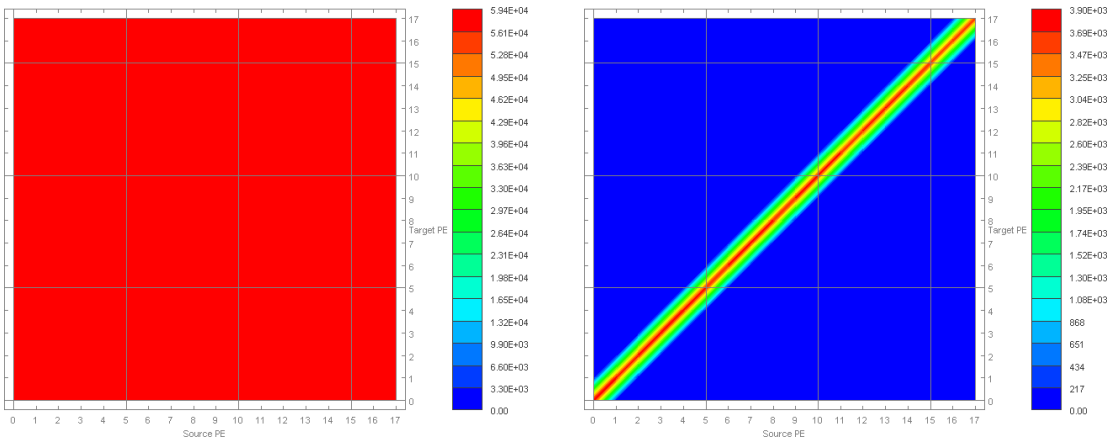
Another issue severely limiting part of its usefulness is the granularity of the timing information provided by the application logs which most of the time is limited to a second. This prevents from accurate comparisons being made on time based data (computation time, message transfer time), although message sizes can still be adequately compared.

7.2 PE-to-PE Communication Analysis

The figures depicted in this section were are all generated by the PE-to-PE communication tool using data from the described tests. They present contour plots where the communication volume between PE pairs (source on x-axis, destination on y-axis) is visualized by color, where red indicates higher volume.

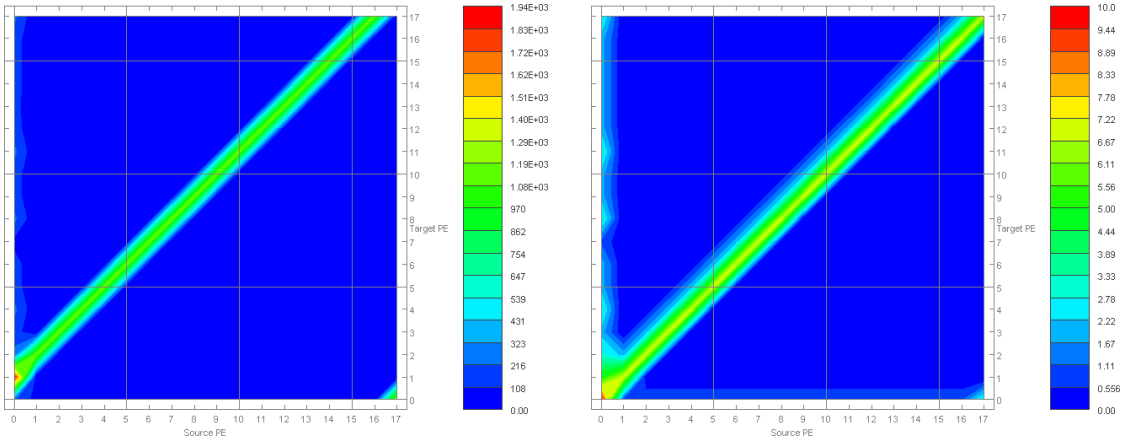
Overall, all the applications show a large degree of self-communication, visible as a diagonal line. Some rely on it almost exclusively, such as LeanMD (Figure 7.11) while most show some spread with their neighbors, as would be the case for NAMD (Figures 7.6 and 7.7), Dense LU (Figure 7.8) and AMR (Figure 7.10). OpenAtom (Figure 7.12) displays a peculiar behavior in which while self communication dominates, all PEs rely on lightweight (most likely control) messages sent by a task in PE 0. CCBench's ring communication pattern (Figure 7.5) is also depicted as a diagonal, but it's vertically displaced, reflecting the one-to-next behavior of the pattern. It exemplifies the easily recognizable all-to-all communication pattern in (Figure 7.5).

Dense LU has mostly local and neighboring, but irregular communication. PE0 is a consistent source of messages, likely for control purposes. Barnes-Hut (Figure 7.9) has a distinct pattern with source PE index being two times of the destination PE index. In addition, it shows an all-to-all communication pattern for PEs 0 through 19. ChaNGa uses Barnes-Hut for much of its computation, but has drastically different communication. NAMD-noPME (Figure 7.6) mostly exhibits local neighbor communication. Including long-range electrostatics (Figure 7.7), NAMD-PME significantly increases neighboring communication, as well as the distance of these neighbors. AMR is dominated by local communication, but with a wider range compared to the other applications. Similarly to Dense LU, PE0 seems to act as a coordinator.



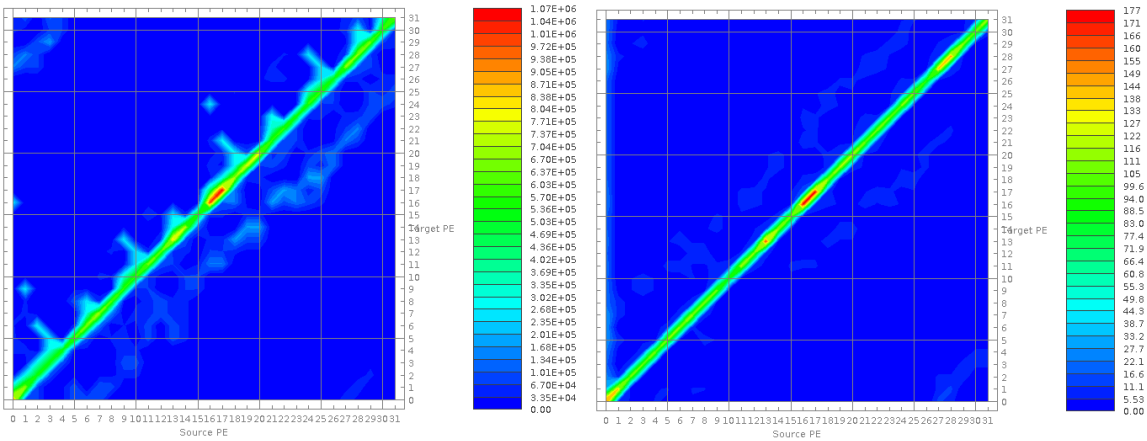
(a) Total Bytes (b) Message Count

Figure 7.4: PE-to-PE communication results for CCBench using the all-to-all communication pattern on 18 PEs.



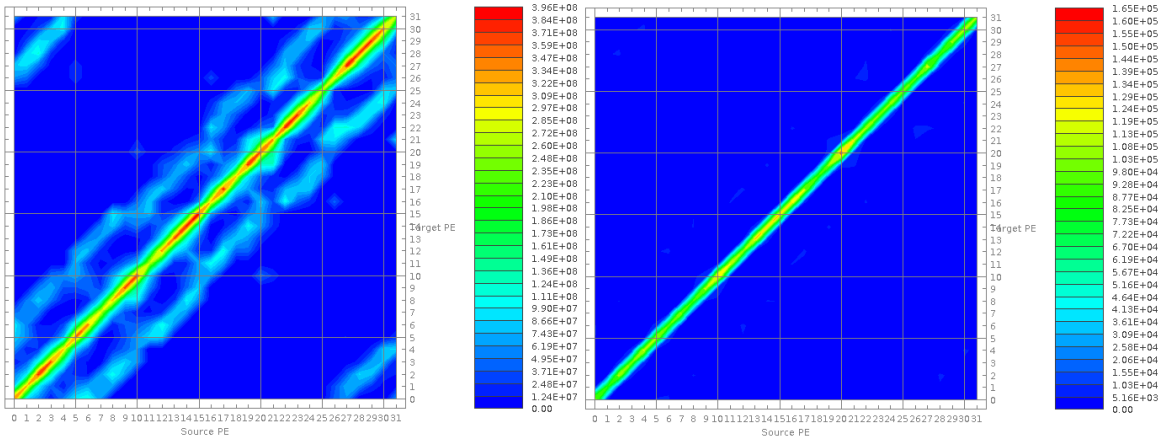
(a) Total Bytes (b) Message Count

Figure 7.5: PE-to-PE communication results for CCBench using the ring communication pattern on 18 PEs.



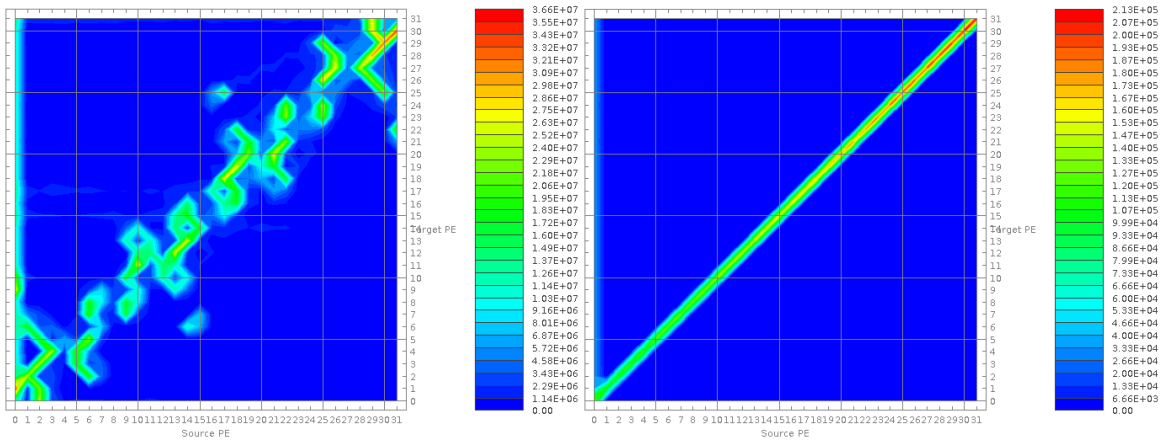
(a) Total Bytes (b) Message Count

Figure 7.6: PE-to-PE communication results for NAMD running in non PME mode, running a 5000 step job on 32 PEs.



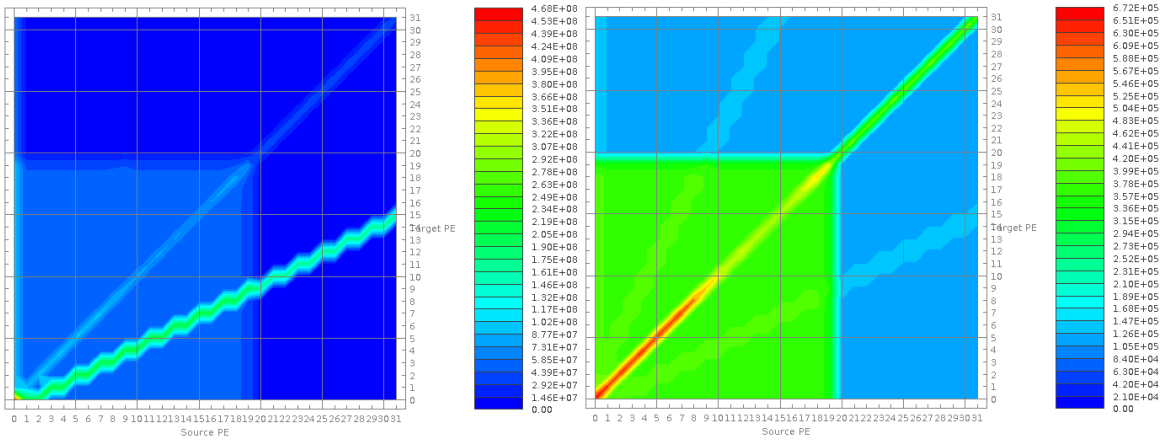
(a) Total Bytes (b) Message Count

Figure 7.7: PE-to-PE communication results for NAMD running in PME mode, running a 5000 step job on 32 PEs.



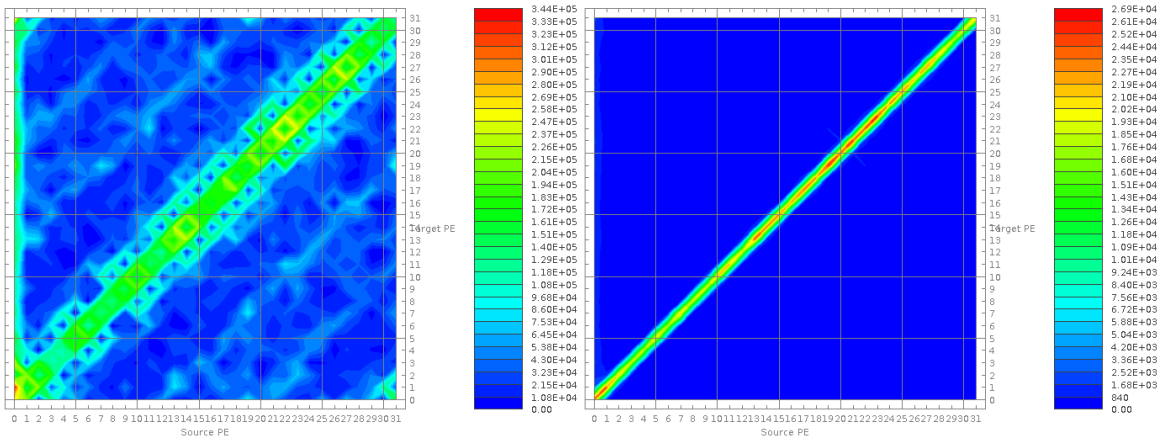
(a) Total Bytes (b) Message Count

Figure 7.8: PE-to-PE communication results for the Dense LU benchmark, performing a LU factorization of a 4096x4096 matrix on 32 PEs.



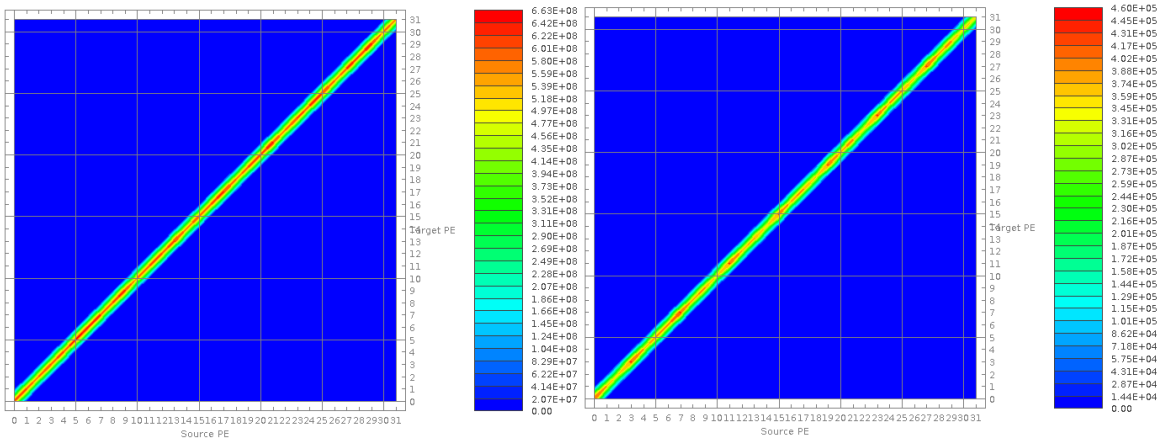
(a) Total Bytes (b) Message Count

Figure 7.9: PE-to-PE communication results for the Barnes-Hut N-body solver, using an input size of 20,000 particles on 32 PEs.



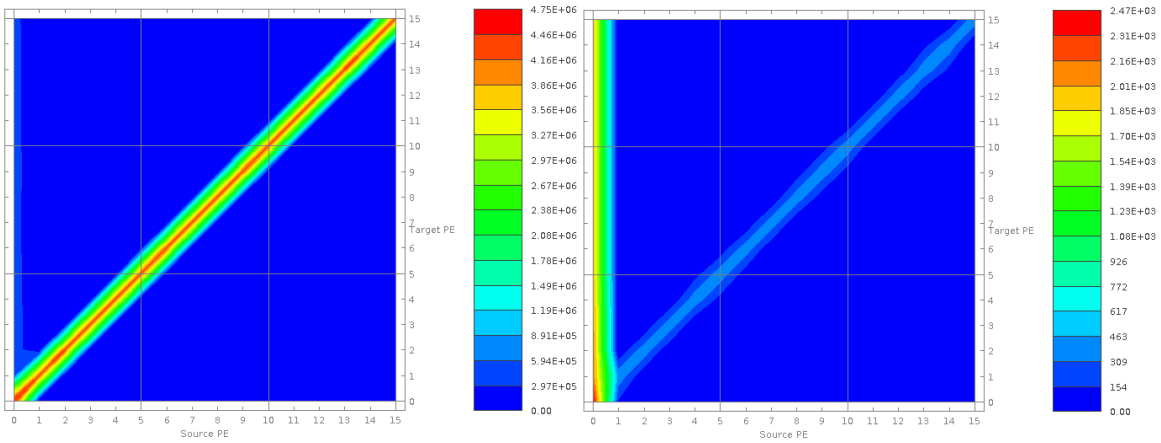
(a) Total Bytes (b) Message Count

Figure 7.10: PE-to-PE communication results for AMR, using a maximum depth of 7 on 32 PEs.



(a) Total Bytes (b) Message Count

Figure 7.11: PE-to-PE communication results for the LeanMD molecular dynamics benchmark for 8000 steps on 32 PEs.



(a) Total Bytes (b) Message Count

Figure 7.12: PE-to-PE communication results for the OpenAtom molecular dynamics simulator with its medium sized auto test on 16 PEs.

7.2.1 Tool Limitations

Measuring the data sent between PE pairs proves to be an effective method to perform educated guesses about their communication patterns. It also scales well to instrument complex applications that yield arbitrarily large logs, thanks to its fixed space complexity and result presentation.

It's limited, however, to measurements of message counts and overall data transferred (both of which can optionally exclude self communication). Communication latency was discarded due to the coarse timing measurements provided within the logs, as well as concerns of the impact it would have on the tool's scalability.

7.3 Communication Overheads

During the collection of test data for the PE-to-PE communication tool, the cumulative message counts and sizes were checked against those provided by Projection's native tools as a way to validate the implemented tool's result. Each check resulted in an exact match.

A different matter, however, is that of predicting the overheads that these figures present in contrast to which might be ideally expected from a given application, as Charm++ introduces additional control communication and the messages themselves are bound to include some metadata.

The overhead in the message sizes can be easily measured using the graph tool and checking the sizes of the messages that CCBench generated against their ideal ones. Charm++ consistently adds 80 bytes for variable size messages and 64 bytes for fixed size ones.

More difficult to predict is the number of control messages introduced to coordinate the compute objects. However, from the performed tests, their number appear to be within a set range for a given application no matter what the total message count of that application is. So any application with a large amount of messaging will easily mask the number of control messages. As reference, in an application with a message count in the thousands typically fewer than 1% of these messages are control ones.

Overall, the instrumented measurements have the significant advantage of reflecting the effective network traffic over the application's ideal values.

Chapter 8

Conclusions

The main focus of this thesis was to provide a way to determine the communication patterns and demands of arbitrary HPC application. Two different tools to accomplish this purpose have been developed and put to test. At this point both tools don't perform equally well to satisfy this goal, with the PE-to-PE communication analyses being far better suited to providing useful results for arbitrary applications. The graph tool, however, while currently more limited in capability, is a foundation for a more detailed analysis of an HPC application's communication, with the potential of revealing issues with more aspects of an HPC application such as dependencies or load balance.

Most applications exhibit communication patterns that significantly differ from the commonly assumed all-to-all pattern, motivating pattern-aware task mapping optimization (i.e., to place highly communicating tasks in close proximity to each other in the cluster) following the job allocation stage. One exception is FFT, which shows a uniform all-to-all communication, yielding limited optimization potential.

8.1 Considerations for Future Work

The work carried out within this thesis establishes an extensible basis which could facilitate further development of the researched instrumentation aspects.

An obvious one would be attempting to overcome the current limitations of the graph tool. This could perhaps be accomplished by an automatic analysis of the graph to reveal unbalanced stages of a program instead of relying on visualization and the user's interpretation. This could very well be complemented by the addition of an automatic detection for certain patterns, such as

data scattering and gathering points, collapsing the flow into the later. This would transform the resulting execution tree into an actual data dependency graph. The tool's infrastructure has been designed with graph support in mind, so the changes it would require would be minimal.

Both profiling tools, however, could see an important benefit in their usefulness if future revisions of Charm++ increase the timing precision of the logged data, or alternative, more accurate, methods to acquire timing information are devised, as this would allow providing latency information and more accurate computation times in the analyses performed.

Bibliography

[1] Aaron Becker, Ramprasad Venkataraman, and L Kale. Patterns for overlapping communication and computation. *Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, IL*, 2009.

[2] Beckman Institute for Advanced Science and Technology at University of Illinois at Urbana Champaign. NAMD - scalable molecular dynamics. `http://www.ks.uiuc.edu/Research/namd/`, 2013.

[3] Beckman Institute for Advanced Science and Technology at University of Illinois at Urbana Champaign. Scientific applications: ChaNGa - computational cosmology and n-body codes. `http://charm.cs.uiuc.edu/research/cosmology`, 2013.

[4] Barcelona Supercomputing Center. About BSC. `http://www.bsc.es/about-bsc`, 2013.

[5] Massachusetts Green High Performance Computing Center. Massachusetts green high-performance computing center. `http://www.mghpcc.org/`, 2013.

[6] Massachusetts Green High Performance Computing Center. What are the green design aspects of the MGHPCC? `http://www.mghpcc.org/about/what-are-the-green-design-aspects-of-the-mghpcc/`, 2013.

[7] I. Dooley, G. Zheng, and J. Lifflander. Runtime systems and tools: Parallel performance analysis, visualization and optimization. `http://charm.cs.uiuc.edu/research/parallel_perf`, 2013.

[8] Eric Atienza (ericaro). surfaceplotter. `https://code.google.com/p/surfaceplotter/`, 2010.

[9] European Organization for Nuclear Research. The worldwide LHC computing grid. `http://home.web.cern.ch/about/computing/worldwide-lhc-computing-grid`, 2013.

[10] Kevlin Henney. Mutual registration: A pattern for ensuring referential integrity in bidirectional object relationships. *EuroPLoP'99*, 1999.

[11] L.V. Kalé and Amitabh Sinha. Projections: A preliminary performance tool for Charm. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, Newport Beach, CA, April 1993.

[12] F. Kaplan, J. Meng, and A. K. Coskun. Optimizing communication and cooling costs in HPC data centers via intelligent job allocation. In *Proceedings of the International Green Computing Conference (IGCC)*, 2013.

[13] Martin Karplus. Chemistry at HARvard Macromolecular Mechanics. `http://www.charmm.org/`.

[14] Object Refinery Limited. Jfreechart. `http://www.jfree.org/jfreechart/`, 2013.

[15] JGraph Ltd. Java graph drawing component. `http://www.jgraph.com/jgraph.html`, 2013.

[16] University of Illinois at UrbanaChampaign. *The Charm++ Parallel Programming System Manual*, 2013.

[17] University of Illinois at UrbanaChampaign. *Projections Manual*, 2013.

[18] University of Illinois at Urbana-Champaign Parallel Programming Lab. Charm++: Mini-apps. `http://charmplusplus.org/benchmarks/`.

[19] University of Illinois at Urbana-Champaign Parallel Programming Lab. Charm++ programming tutorial. `http://charm.cs.illinois.edu/tutorial/CharmConcepts.htm`.

[20] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with namd. In *Journal of Computational Chemistry, 26:1781-1802*, 2005.

[21] Tom Quinn. NChilada collaboration site. http://hpcc.astro.washington.edu/nchilada/bin/view/Nchilada/WebHome, 2006.

[22] Doug Ramsey. UC San Diego's greenlight project to improve energy efficiency of computing. http://ucsdnews.ucsd.edu/newsrel/science/07-08GreenLightProj.asp, 2008.

[23] John Rath. Chinas Milky Way-2 is worlds top supercomputer. https://www.datacenterknowledge.com/archives/2013/06/17/chinas-tianhe-2-topples-top500-supercomputers/, 2013.

Appendix A

Tool Usage Notes

A.1 Communication/Computation Graph Tool

A.1.1 Statistics Description

These statistics apply on a per-PE or overall basis:

Computing Nodes

Total number of computing nodes within the graph or PE.

Total Messages

Total number of communication nodes.

Simple Messages

Total number of communication nodes representing one to one communication.

Multicast Messages

Total number of communication nodes representing a multicast message.

Broadcast Messages

Total number of communication nodes representing a broadcast message.

Comp To Comp Links

Number of assumed links from a computation node to a different computation node without a communication between them, as a message between them was not present within the logs.

Comm To Comp Links

Number of links found between a communication node and a computation node within the same PE.

Comp To Comm Links

Number of links found between a computation node and a communication node.

Missmatched End Processing Entries

Number of End Processing entries for which a matching Start Processing entry couldn't be found within the logs.

Leftover Begin Processing events

Number of Begin Processing entries for which a matching End Processing entry couldn't be found within the logs.

Non Messageless Begin Entries With No Message

Number of Begin Processing entries that should be preceded by a message for which their triggering message couldn't be found.

Last Assumed Parent Skipped

Number of cases where direct local link between computing nodes was assumed but a computation parent candidate couldn't be found.

Creation Entries Without Parent

Number of messages for which a computation parent couldn't be found.

Re-used events

Number of collisions from new events with IDs that match that of older still open IDs.

Inter-PE Comm To Comp Links

Number of links found between a communication node and a computation node across different processing elements.

A.2 PE-to-PE Communication

A.2.1 Statistics Description

These statistics apply on a per-PE or overall basis.

Received Messages

Count of received messages (typically larger or equal to the amount of sent messages due to multicast and broadcast messages).

Received Bytes

Amount of received bytes (as with the number of receiver messages, typically larger or equal to the number of sent bytes).

Sent Messages

Count of sent messages.

Sent Bytes

Amount of sent bytes.

Overall sent Messages (Considering number of targets)

Accumulation of the number of recipients each message was sent to.

Overall sent Bytes (Considering number of targets)

Accumulation of each of the message's sent bytes multiplied by the number recipients each message was sent to.

Sent Simple Messages

Count of unicast messages.

Sent Bytes through Simple Messages

Amount of bytes sent through unicast messages.

Sent Multicast Messages

Count of multicast messages sent.

Sent Bytes through Multicast Messages

Amount of bytes sent through multicast messages.

Overall sent Multicast Messages

Count of multicast messages sent multiplied by the number recipients each message was sent to.

Overall sent Bytes through Multicast Messages

Amount of bytes sent through multicast messages multiplied by the number recipients each message was sent to.

Sent Broadcast Messages

Count of broadcast messages sent.

Sent Bytes through Broadcast Messages

Amount of bytes sent through broadcast messages.

Overall sent Broadcast Messages

Count of broadcast messages sent multiplied by the number recipients each message was sent to.

Overall sent Bytes through Broadcast Messages

Amount of bytes sent through broadcast messages multiplied by the number recipients each message was sent to.

Messages with an invalid source

Messages not marked with a valid source PE within the logs.

Messages with an invalid target

Messages not marked with a valid target PE within the logs.

Skipped messages due to invalid target or source

Number of messages that had to be discarded due to having an invalid source PE, target PE or the combination of both.

Total Messages overflows

This statistic indicates the number of cells that have overflown, nonzero values imply some invalid data in the number of messages table.

Total Bytes overflows

This statistic indicates the number of cells that have overflown, nonzero values imply some invalid data in the total bytes table.

Total Delays overflows

Currently unused.

A.3 CCBench User Manual

CCBench is a simple synthetic benchmark written in Charm++ that allows running an application with configurable communication and computation demands.

While more elaborate benchmarking methods exist to evaluate both computation and communication performance, that isn't its most important purpose even though it can be used to achieve so to a certain degree. The benchmark is meant to be used as a test suite for the development and validation of tools that work on data extracted from the execution logs generated for Projections by Charm++.

A.3.1 Building CCBench

Modify the Makefile setting the CHARMC variable to the path of the Charm++ compiler binary by replacing the bolded fraction of the following line:

```
CHARMC=../charm/bin/charmc $(OPTS)
```

To compile just navigate to the benchmark's directory and run the *make* command, which should generate the benchmark's binary if the Make tool and Charm++ compiler are properly set up.

Keep in mind that this will compile CCBench using Charm++'s default settings, custom compiler flags can be added by setting the *OPTS* variable when invoking the *make* command as in the following example, which enables the generation of the program's trace files for Projections (see the Charm++ and Projections manuals for more information):

```
make OPTS="-tracemode projections"
```

A.3.2 Running CCBench

A.3.2.1 Synposis

`ccbench` *`msgSize outerIterations innerIterations bufferSize bufferTraversals`* `-comma2a|-commcircle`

A.3.2.2 Example Usage

CCBench should likely be launched in conjunction with the *charmrun* utility generated when compiling CCBench to execute the application in several different machines. For more information about charmrun and indications on how to configure it refer to Charm++'s documentation.

Commonly it should be launched similarly to the following example, where it's run using 3 different processing elements:

```
./charmrun +p3 ./ccbench 128 20 10000 128 100 -comma2a
```

A.3.2.3 Parameter Summary

- ***msgSize*** Defines the number of bytes sent from processing element to processing element on each of the pattern's steps. The larger the size the larger the bandwidth requirements of the benchmark are.

- ***outerIterations, innerIterations*** Define the number of iterations that the selected pattern will be repeated for, this number being the result of multiplying both values. Each of these parameters must have a value smaller than 2^{16}. This is a workaround to a SDAG limitation where indexes for iterations are represented as 16 bit unsigned integers preventing a large number of iterations.

- ***bufferSize*** Determines the allocated memory size in bytes of the buffer to perform a checksum from to simulate a processing task. The larger this buffer is the longer this checksum process should take.

- ***bufferTraversals*** Allows setting a number of iterations to checksum the allocated buffer. This allows to increase the time employed by the simulated processing task without increasing its memory demands.

- **`-comma2a`|`-commcircle`**

  `comma2a` Uses an all-to-all communication pattern on each iteration using point to point messaging (not broadcasting). The amount of messages sent on each iteration using this pattern grows to the square of processing elements employed. It's a more bandwidth intensive test.

  `commcircle` Sets a communication pattern where each processing element will send a message to the next until an entire communication ring has been completed for an iteration. It's a more latency focused test.