Abstract

abstract goes here

Acknowledgements

Acknowledgements goes here

Contents

1 Introduction

1.1 Contributions

A framework for implementing distributed algorithms and applications, created in and for C++, using UPC++, a library that provides PGAS mechanisms for communication. The framework provides an alternate programming model for distributed algorithms, in which a computational task, its data and its communication routines, is represented as a single object.

1.2 Outline

123

2 Background

2.1 State of High Performance Computing

Software performance gains from increased frequency in single processing units reached its limits many years ago, and multi-core systems has been prominent in personal computing since. Up until the early 2000s, the speed at which computers could execute sequential programs would steadily increase as clock speed and semiconductor fabrication improved [1]. Heat dissipation and energy consumption caused development of the maximum achievable CPU frequency to stall, effectively freezing the number of tasks that can be completed within a certain time frame on a single processing unit [2]. Yet, the size of computational tasks and the amount of data is still increasing, resulting in single-core systems not being able to keep up with the demands of problems found in modern science, engineering and business.

The answers to these problems were multi-core and many-core systems, which bypasses the aforementioned performance wall by utilizing several processing units. Within the field of high performance computing, however, multi-processor systems were not a novelty. Using networks of workstation computers for parallel computation was an attractive alternative to traditional supercomputers, partly because new processor technology could easily be incorporated without replacing the entire system [3]. While the field of parallel programming was relevant in high-end scientific applications before halt in clock speed development and the following *concurrency revolution*, homogeneous distributed systems now dominate the field of high performance computing.

Multi-core and many-core architectures achieve parallelism through explicit task distribution and scheduling, managed by the programmer. Exploiting parallelism is often an application-specific issue, and can pose several challenges. In the fields of distributed- and high performance-computing, aiming for scalability further increases the complexity of the programming task. Several programming models have been devised to support development of distributed, parallel applications, and the current leading models can be clustered into different language paradigms: *shared memory* and *message passing* [4].

2.2 Programming models

On top of these models, abstractions can be applied to hide complexity from the programmer in order to reduce development time and ease the debugging process.

There exists a perception that abstraction carry an inherent performance penalty, but abstractions can also be powerful tools in high performance computing [5].

For example, computing at a large scale can introduce issues relating to handling large amounts of data, scaling to a large or arbitrary number of computation nodes, or developing algorithms that utilizes communication routines and synchronization.

Identifying common patterns in different problems allows for development of frameworks and libraries that are optimized to handle classes of problems matching the same patterns. For example, the MapReduce framework hides significant programming overhead of problems that can be expressed within semantics provided by the framework and the programming model it represents. In its most primitive, data is supplied to the *Map* function and a function to execute on the data is supplied to the *Reduce* function. The MapReduce programming model is designed for algorithms and problems that can be expressed using these two functions, and the MapReduce framework is implemented to provide functionality that can be applied generally to problems within that programming model. In addition to implementing communication routines and data distribution logic, the framework can supply scalability and fault tolerance, due to the information of the application that is implied by the programming model it is utilizing. Other examples of systems specialized in specific programming models include Dryad [6], which models an application's data flow into directed acyclic graphs, and Pregel [7], a framework for graph processing.

2.3 Data intensive computing

2.4 PGAS

[8]

2.5 GASNet

2.6 UPC++

3 Design

Saddlebags is a framework for parallel computing, aimed at applications and algorithms that can be implemented in C++. It uses UPC++ for thread spawning, communication, and data distribution, which provides PGAS mechanisms and language constructs such as remote procedure calls, virtual shared memory, and distributed objects. Saddlebags provides a programmer with objects and functions that are inherently distributed, and can be extended with methods and data to express different algorithms.

This section will cover the design and architecture of the system, outlining its different components, its programming model, and data flow. Section 3.1 explains the core concept and design philosophy of Saddlebags, Section 3.2 explains the programming model and data flow, while Section 3.3 outlines the architecture.

3.1 Motivation

Saddlebags is designed with several key features in mind, all of them related to what responsibilities a parallel computing framework can take off the an application developer's hands. Some of the underlying concepts and the motivation behind expanding upon the functionality of UPC++ follows.

1. Computational tasks should be executed in the same partition in which the data it requires is located

Problems that can be expressed with data parallelism [**?**] can be distributed across computing units as individual tasks related to specific data. A key concept within PGAS programming is that a private memory segment has affinity to a single computing unit. To exploit data locality, a computational task should only be performed on the unit of computing which the data it requires has affinity to. Expressing behavior like this is trivial when the distribution of data is deterministic. Another wording of the same behavior is that computational tasks have affinity to the data it requires, the same way data has affinity to a computing unit and its private memory segment.

2. A multitude of data should be implicitly distributed across available partitions

Following the principle of a computational task's affinity to its data, full utilization of available processing units is only possible when a program's data is distributed. This means that data used in parallel processing should always be distributed across available memory partitions, so that the computational tasks that require the data is equally distributed.

3. The programmer should not need to write communication routines such as messages or remote procedure calls

The purpose of building a framework on top of UPC++ is to provide interfaces for the implementation of distributed applications without the complexity of communication routines. While computation methods are application-specific, communication patterns can be implemented for a general case. When claiming that programmers are relieved of writing communication routines, it refers to the need for knowing where data is physically stored, which partition to communicate with, and the need for ensuring that messages are delivered before continuing computation. Data movement will still be explicit, but expressed in an abstraction level in which data locality and memory partitioning is not considered.

Saddlebags presents a programming model that is devised from a specific communication model, and it is assumed that algorithms that can be expressed within this model can also be implemented to communicate in a general pattern. Predictable communication patterns also allows for optimizations and reduces the need for application-specific synchronization. This also makes the physical distribution of data transparent to the programmer.

The goal of building a framework on these specific features is to allow computational parallelism to be expressed with the same semantics as data parallelism, while still allowing tasks to communicate data between each other. The complexity of implementing applications that fit this classification is reduced by utilizing a general communication pattern that relieves programmers of implementing specialized communication. Programmers will in this framework use communication routines by referring to data rather than physical location in the form of partition or computing unit.

3.2 Programming model

3.3 Architecture

3.3.1 Item

3.3.2 Table

Distributor

3.3.3 Worker

The communication patterns of a push/pull paradigm is simple enough that computational costs of communication in an application can be reasoned about, despite

implementation complexity being hidden.

The Lighght framework is designed with the goal that abstractions provided through C++ mechanisms such as inheritance and type templates should not affect algorithmic complexity: This will only be true for algorithms that can be tailored to the push/pull/work paradigm.

4 Related work

4.1 Husky

4.2 Spark

5 Implementation

5.1 Distributed objects

5.2 Templates

5.3 Table

5.4 Synchronization

barriers in cycle

5.5 Usage

6 Evaluation

6.1 Expressiveness

6.2 Energy efficiency

6.3 Abstraction costs

6.4 Comparison with other systems

7 Future work

7.1 Fault tolerance

7.2 View based serialization

8 Conclusion

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, p. 54, Jan 2005.

[2] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, p. 1369–1386, 2012.

[3] B. Wilkinson and M. Allen, *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, 2 ed., 2005.

[4] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter, "Partitioned Global Address Space Languages," *ACM Comput. Surv.*, vol. 47, pp. 62:1–62:27, May 2015.

[5] B. C. Mccandless and A. Lumsdaine, "The role of abstraction in high-performance computing," *Lecture Notes in Computer Science Scientific Computing in Object-Oriented Parallel Environments*, p. 203–210, 1997.

[6] "Dryad - Microsoft Research." `https://www.microsoft.com/en-us/research/project/dryad/`. [Online; accessed 03-April-2017].

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," *Proceedings of the 28th ACM symposium on Principles of distributed computing - PODC 09*, 2009.

[8] J. Lagraviere, J. Langguth, M. Sourouri, P. H. Ha, and X. Cai, "On the performance and energy efficiency of the pgas programming model on multicore architectures," *2016 International Conference on High Performance Computing & Simulation (HPCS)*, 2016.