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 Partitioned Global Address Space

[8]

Traditionally, parallel programming (and the field of High Performance Computing) has been clustered in two language paradigms: *shared memory* and *message passing* [4]. Shared memory models provide an abstract memory space that is available for access across several concurrent compute threads. Message passing models, such as MPI [9], give a programmer control of the flow of execution by conceptually bundling most of the communication into a message abstraction. This approach can be used to achieve high performance; the programmer can reason about the cost of communication, and messages can be delivered between compute nodes to scale a program to large, multi-node systems.

Programming languages that achieve parallelism through a shared memory abstraction that provides memory access to several compute threads concurrently can be classified as Global Address Space (GAS) langauges. Communication between threads is achieved through reading and writing shared data, rather than explicit message passing [10]. Partitioned Global Address Space (PGAS) languages aim to provide a hybrid approach between previous parallelization paradigms, by utilizing the programmability advantages of the shared memory model, and the communication efficiency of message passing.

2.4.1 GASNet

GASNet is a communication API for GAS/PGAS languages that aims to provide a portable communication layer to systems that would otherwise be platform specific or network implementation dependent. Portability is achieved through translating PGAS programs to an intermediate C representation, which can be compiled into a platform specific application using the system's standard C compiler [10].

GASNet is distributed with several API implementations with support for different underlying network architectures, called *conduits* [11]. This means that systems built on GASNet can be implemented to utilize different communication interfaces, including, among others, MPI, shared memory, UDP/IP, and InfiniBand verbs.

2.5 Unified Parallel C

Unified Parallel C, referred to as UPC, is a superset of the C programming language for writing SPMD programs, utilizing a global address space paradigm for parallelism. A complete UPC specification was first standardized in 2001, and several UPC compilers have been under development since, including GNU UPC, which is an extension of GCC [12], and Berkeley UPC, which is dependent on GASNet [13].

The address space of a UPC program is partitioned into several logical fragments, with each concurrent thread being mapped to one fragment. With this model, each thread has affinity to its own individual memory region, which makes it possible to exploit data locality [14] and utilize parallelism with the same language constructs.

Handling data in a memory partition mapped to a single thread makes it implicit that data is being accessed by a thread that has affinity to the given partition. Memory can also be explicitly distributed across several threads, using shared global memory, which disregards any thread affinity and is logically separated from the aforementioned thread-mapped memory partitions.

The shared memory abstraction of UPC is provided regardless of underlying hardware structure. Partitioned memory segments can be mapped to individual compute nodes in a cluster, utilizing data locality by performing computation on private data close to the location where it is stored, while still providing a shared memory abstraction for communication and programmability. The original UPC language specification states that a driving principle behind UPC is that parallelism and remote memory access should not obfuscate the resulting program; programmers should themselves utilize the language constructs to design programs and data structures, instead of being provided with solutions to specialized tasks. Additionally, the specification claims that a shared memory programming model is attractive for users of distributed memory systems [15]. These two concepts can in part make up the motivation behind the UPC language.

2.6 UPC++

UPC++ is a PGAS extension to the C++ programming language. It is developed at University of California, Berkeley and saw the release of its 1.0 version in September 2017. UPC++ is implemented as a library extension, due to the complexity of developing a separate C++ compiler [16]. The library heavily utilizes C++ template programming [17] for PGAS mechanisms, which allows the creation of library functions that use generic variables that can be adapted to different types.

2.6.1 Programming model

UPC++ follows the SPMD model and like UPC utilizes threads with affinity to memory partitions. While UPC++ allows the use of C++ functionality such

```
void main()
{
  upcxx::init();
  cout << "Hello world! " << upcxx::rank_me() << endl;
  upcxx::finalize();
}
```

```
Output:
Hello world! 0
Hello world! 2
Hello world! 3
Hello world! 1
```

Listing 1: "Hello World"-program in UPC++ and its output

as objects and lambdas, it still aims to provide high performance and explicit communication by leaving much responsibility regarding concurrency and data movement to the programmer.

Each UPC++ partition has access to its own private memory segment, and is identified by its rank, an integer that can range from zero to the total number of partitions. Listing 1 demonstrates a simple program that is started with 4 threads, each printing their rank, illustrating the Single Program Multiple Data property of UPC++ programs. Ranks can communicate through a logical shared memory segment, by utilizing UPC++ communication functionality, including global pointers, remote procedure calls (RPCs), and distributed objects. The number of ranks is static during execution; the total rank count is specified when launching a program and cannot be modified during runtime.

Through communication or internal operations, a single thread may consist of several independent tasks that need access to computing resources for the program to continue its execution. In order to keep communication data movement predictable, a programmer will need to yield control of execution. This means that operations that require communication, such as RPC or global memory accesses, are separated from the main thread of execution and will not expend computing resources until explicitly granted permission to do so. Remote and asynchronous tasks are exposed to the caller through *future* objects, which will change their internal state when the task has completed. This allows the caller to perform operations dependent on the completion of remote tasks or retrieval of remote data, and effectively communicate between ranks.

The 1.0 release of UPC++ introduced the concept of *distributed objects*, which are objects defined by a generic type within UPC++ with the same identifier across all ranks. This means that data stored in the private memory segment of a partition can be referenced by a different rank in RPC calls. The data within the object is not implicitly distributed, but the name of the variable is. Local contents of the distributed objects can as such be explicitly distributed to any rank.

2.6.2 Progress

Outstanding asynchronous operations are not completed automatically, because the program consists of a static number of threads. In order to advance the internal state of UPC++ or complete RPC calls, computing resources are yielded from the main thread of execution via the *progress()* function. This concept puts the responsibility of frequently progressing UPC++ state on the programmer's shoulders. The goal of this is to provide visibility of resource usage and parallelism, the latter being important to achieve interoperability with libraries and software packages that would otherwise restrict the use of multi-threading. Being cautious about where to progress RPCs relieves the needs for concurrency control such as mutexes, and implies atomicity in partition-internal operations of basic data types. More complex data structures that are not designed for concurrent access can be safely referenced in RPCs across partitions as long as the *progress()* function is used only between atomic accesses.

2.6.3 Remote Procedure Calls

RPCs can be used for both modifying and copying data in the private segment of another rank's memory. The future objects returned by UPC++ RPCs can be used to indicate completion of the remote task, and communicate return values. A program that is dependent on data from a different rank, can halt advancement of the main thread by using the *future.wait()* function, during which progress is attempted until the given future indicates that its corresponding remote operation has completed. This is not possible to achieve in tasks executed by progressing – futures can only change state during progress, and progress-callbacks cannot themselves initiate further progress of other tasks. This concept provides a clear distinction between execution via the main thread of a rank, and execution via user-initiated progress of the same rank.

Listing 2 demonstrates communication between two ranks using RPCs, distributed objects, and futures. A distributed object of an arbitrary class is initiated on all

```
int main(int argc, char *argv[])
{
  upcxx::init();

  auto myObj = upcxx::dist_object<myClass>(myClass());
  upcxx::barrier();

  if(upcxx::rank_me() == 1)
  {

    auto f = upcxx::rpc(2, [](
      upcxx::dist_object<myClass> &my_dist_obj])
    {
      return (*my_dist_obj).myProperty;
    }, myObj);

    f.wait();
    std::cout << f.result() << std::endl;

  }

  upcxx::barrier();
  upcxx::finalize();
}
```

Listing 2: Communication through RPC, referencing a distributed object of an arbitrary C++ class. Rank 1 retrieves and prints *myProperty* from rank 2's instance of *myObj*.

ranks, and a RPC is invoked to retrieve a property of the instance of the distributed object *myObj* local to rank 2, back to rank 1. A barrier is used to ensure that the identifier of the distributed object is available on all ranks before it is referenced in the following RPC. The remote procedure is expressed with a lambda function, an unnamed function object. The RPC returns future *f*, which will hold a copy of the remote *myProperty*, the return value of the lambda function passed to the UPC++ *rpc* function. The return value of the lambda will be available as *f.result()* when the function has been executed, which is indicated by returning from *f.wait()*.

A distributed object is used in this example because it can be passed as a reference to the RPC on the sender's side, and it will be translated to the local copy of the distributed object with the same name on the receiving side. Non-distributed objects can only be passed as copy through RPCs, so distributed objects must be used for retrieving or modifying properties of remote objects. Within the RPC, the distributed object can be de-referenced to access properties of the underlying class. This is exemplified in listing 2 through the access of *myProperty*.

2.6.4 Global pointers

Basic built-in C++ datatypes can be transmitted between partitions as RPC arguments or return values, by passing them as a copy. To communicate pointers to other partitions, the shared memory segment must be utilized. UPC++ can create a pointer to memory allocated in the shared memory segment, and copy local data to this location. The global pointer can be passed to other partitions through RPCs. Accessing global memory from a partition other than the one that initiated the pointer has to be done through asynchronous operations. UPC++ provides functions for remote puts and gets, which require internal progress on the partition that is hosting the shared memory in order to complete.

3 Related work

3.1 Husky

3.2 Spark

3.3 DataMPI

3.4 Legion

4 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.

4.1 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.

4.2 Programming model

The programming model and data flow model of Saddlebags is based around coupling data and functionality together into an object. Objects that serve the purpose of executing computational tasks in Saddlebags are named *Items*. Items contain pre-existing general communication methods, and the main effort of developing an application with Saddlebags is implementing the computational tasks within an item, and integrating communication and data-flow into that task.

Items of different types can be implemented in the same application, and stored in different *tables*. The behavior of tables cannot be extended by the programmer, but every application must define their tables by name and item type. Tables are common for every partition, while an instance and name of an item can only exist on a single partition. This is how distribution of data is achieved: items are spread across partitions, while the tables that contain them exist on every partition, and can reference items regardless of physical location.

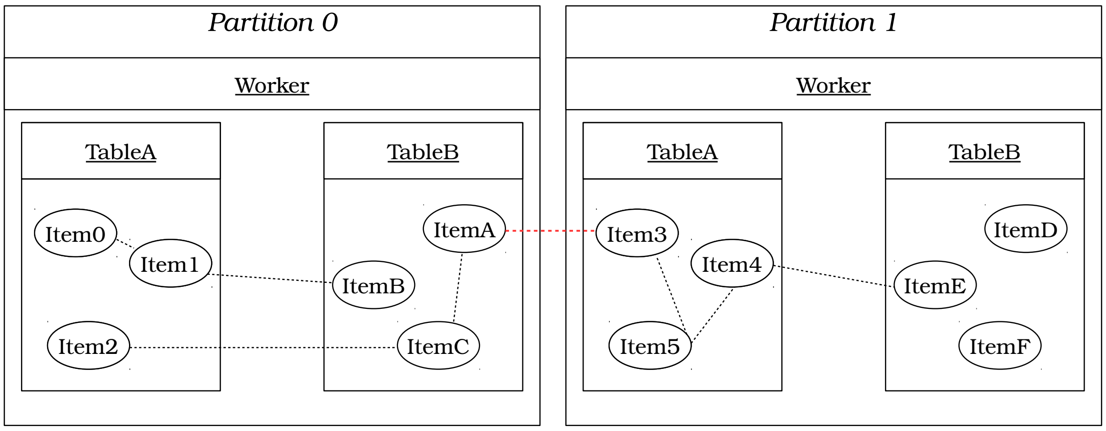

Figure 1: Overview of the key data structures in a Saddlebags application. Underlined names are distributed objects. Black lines indicate example communication within a partition. Red lines indicate example communication between partitions.

The relation between items, tables and partitions are illustrated in figure 1. This illustration shows a basic design of a Saddlebags application with two tables and two UPC++ partitions. Items of different types are stored in different tables, and the same tables are available on both partitions. Item names are unique across all partitions, and a single item have affinity to a single partition.

Tables are stored within a *worker*, a distributed object. Tables are referenced via this object. The behavior of the worker need not be extended by the programmer, and in the context of data flow and execution model the worker is relatively insignificant, and will be mostly ignored in this section. It will be explained further when discussing implementation and interoperability with UPC++.

Items implement a *work-push-pull* model, in which *work* refers to a computational task implemented in the item class, and *push* and *pull* are the two available modes of communication. The work of an item is designed to be executed iteratively, and each iteration of a Saddlebags application involves executing work and performing any communication routines that was called as a result of the work. An iteration applies to every item in every table, and it is with this iterative model that an application can be executed.

The data flow model of Saddlebags is implemented with the concept of *events*. Within the context of this framework, an event refers to the receiving of any communication message, or reaching specific points of the iteration process. Whenever an event is triggered, for example when receiving a *push* message, the item that received the message calls a *received push* method. The contents of this method can be implemented differently in different item types, and it also able to initiate other communication routines. This is directly related to the data flow of an application: when receiving a *receieved pull* event, the event-triggered method determines what data to return. When a pull returns with some data, the method triggered by the *returning pull* event determines how to handle the data.

4.3 Architecture

4.3.1 Item

4.3.2 Table

Distributor

The communication patterns of a push/pull paradigm is simple enough that the cost of data movement as a result of communication in an application can be reasoned about, despite implementation complexity being hidden.

Saddlebags is designed with the goal that abstractions provided by the programming model, through C++ mechanisms such as inheritance and type templates, should not affect algorithmic complexity. This will likely only be true for algorithms that can be tailored to the push/pull/work paradigm.

5 Implementation

5.1 Distributed objects

5.2 Templates

5.3 Table

5.4 Communication

5.5 Synchronization

5.6 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

7.3 File input

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.

[9] "Message Passing Interface." `https://computing.llnl.gov/tutorials/mpi/`. [Online; accessed 27-November-2017].

[10] D. Bonachea and J. Jeong, "GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages," 2002. [Online; accessed 23-November-2017].

[11] D. Bonachea and P. Hargrove, "GASNet Specification, v1.8.1."

[12] "GNU UPC." `http://www.gccupc.org`. [Online; accessed 23-November-2017].

[13] "Berkeley UPC." `http://upc.lbl.gov`. [Online; accessed 23-November-2017].

[14] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An Evaluation of Global Address Space Languages," *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP 05*, 2005.

[15] W. W. Carlson, D. E. Culler, and E. Brooks, "Introduction to UPC and Language Specification," 1999.

[16] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++ : A PGAS Extension for C++," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[17] "Function template, cppreference.com." `http://en.cppreference.com/w/cpp/language/function_template`. [Online; accessed 23-November-2017].