INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Graph Concepts for the DASH C++ Template Library

Stefan Effenberger

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Graph Concepts for the DASH C++ Template Library

Stefan Effenberger

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Tobias Fuchs
Abgabetermin:	ADD DATE

Abstract

ADD ABSTRACT

Contents

1 Introduction

Many scientific projects are largely enabled by simulation. Because such simulations often require huge computational capabilites, single compute nodes with a shared-memory architecture cannot provide enough computation power and storage for numerous cases. For this reason, in High Performance Computing (HPC), work is distributed among multiple, interconnected nodes to facilitate the solving of large problems in a timely manner. Since processors cannot directly access the memory of other nodes, the traditional programming model for such systems requires programmers to explicitly distribute data between nodes via message passing. This imposes high demands on the programming skills of scientists who might not have a background in computer science.

Therefore, with the Partitioned Global Address Space (PGAS) model, a new approach is proposed: The memory space of individual nodes in a system is unified within a global address space so that each node can directly access the memory of all other nodes. Programmers are still required to keep data access between nodes to a minimum because data transferal over an interconnect is costly. To further reduce the demands on the programmer, distributed data structures that handle data distribution and load balancing are needed.

Furthermore, data-intensive tasks have been gaining a continually growing interest in the scientific community. Traditionally, applications in HPC follow a computation-centric approach by solving numerical algorithms in the fastest possible way. As "Big Data" is becoming increasingly important in scientific projects, a shift towards more data-oriented applications can be observed in recent HPC projects [ZZZ$^+$14]. This trend requires distributed data structures that allow for the storage of large amounts of irregular data and cater to the needs of ever-changing dynamic data.

1.1 Problem statement

Data can be represented in numerous ways. The most generic form of data representation is enabled by *graphs*. A graph G(V, E) is a pair with a set of vertices V and a set of edges E that connect the vertices. This allows for the representation of data and its relationships in regular as well as irregular patterns.

On distributed machines, graph data structures can be implemented using a variety of different characteristics. This has lead to many different implementations - usually a new implementation for each algorithm - which are hardly compatible with each other. To overcome this situation, generic programming abstractions to facilitate reuse of existing code and to lower the demands on programmers are needed.

As of today, no generic graph abstractions implementing the PGAS model exist. This work therefore aims to provide a graph abstraction for C++ containers that allows for the implementation of arbitrary graph algorithms following the PGAS model on distributed memory machines.

1.2 Scope and Objectives

In this work, a C++ concept for graph containers following the PGAS model is presented. The concept is part of the DASH C++ Template Library and thus uses concepts already present in the library.

The graph concept is meant to provide a generic framework for the programming of arbitrary graph algorithms in the context of distributed machines and especially the Partitioned Global Address Space model. This means that it meets the following requirements:

- Native support for one-sided communication
- Support for the programming of synchronous graph algorithms
- Support for the programming of asynchronous graph algorithms
- Portability across platforms (PORTABLE EFFICIENCY?)
- Support for heterogenous systems
- FINISH REQUIREMENTS

Furthermore, this work provides concepts for the dynamic allocation of graph data across multiple machines with a focus on optimized data locality. LOAD BALANCING?

A reference implementation is then used to verify the usability, correctness and universality of the given concepts. While the concepts are designed for high performance, the reference implementation is not. This means that the evaluation of this implementation does not account for the performance of this work's concepts.

2 Background

This chapter covers some fundamental background knowledge needed for a better understanding of the following chapters of this thesis. Only explanations directly relevant to the topics of this thesis are provided.

Since the result of this work is a C++ concept, some important language expressions and concepts are firstly discussed, along with a description of the Standard Template Library on which concepts this work is built upon. The reader is then introduced to the domain of High Performance Computing which is the main application area for this work. A brief overview of the Partitioned Global Address Space programming model is then followed by a description of the DASH Library which provides core concepts used in this thesis.

2.1 C++ Concepts

2.1.1 Language Concepts

2.1.2 Standard Template Library

2.2 High Performance Computing

High Performance Computing (HPC) is a broad term describing advances for the fastest possible computation of a given problem. Gustafson's Law [Gus88] suggests that a compute system can linearly grow with the problem size: A problem of two times its original size can be computed on a system with twice as many processors in the same time (best case scenario). This means that very large problems can be computed in an acceptable timeframe if there is a sufficiently large compute system available. Depending on the problem size, two different system architectures are used in HPC:

Shared Memory A shared memory system consists of a single node with multiple processors connected to the same random access memory. Memory access for the different processors can be uniform, but many systems implement a non-uniform memory access (NUMA) design where a part of the memory is assigned to each of the processors. A processor in a NUMA system can access its assigned memory faster than the memory of the other processors. Because processors can access all data at all times, communication between processors has a low cost which simplifies programming on these systems in comparison to distributed memory systems. Achieving high performance on NUMA systems is more problematic because the programmer has to take data locality into account [Lam13].

Distributed Memory Multi-processor systems in which each processor has access to its own memory space are called distributed memory systems. These systems usually consist of several shared memory nodes with the processors of one node not being able to directly access memory of other nodes. While single shared memory systems can only be scaled to a certain extent, the scalability of distributed systems is much higher [PTM96].

The nodes are connected with a network interconnect for communication between the processors. Due to the latency of the interconnect being significantly higher than the latency of a memory bus in a shared memory system, communication is much more costly. This imposes higher demands on the programmers' skills in comparison to shared memory systems.

The largest problems in science are computed on “supercomputers” like the *SuperMUC* at the *Leibniz Rechenzentrum* in Munich. These distributed memory machines consist of hundreds or even thousands of homogeneous nodes that are connected with a specialized interconnect. To this date, *message passing* is the prevalent programming model for such systems.

2.3 Partitioned Global Address Space

Shared Memory and *Message Passing* are the dominant models in HPC as of this writing. As pointed out in section 2.2 however, the usage of Message Passing requires high skills in computer architecture and programming. To ease this problem, the Partitioned Global Address Space (PGAS) model has been proposed. It unifies some of the benefits of both of these models by creating a global address space over the initially local-only address spaces of distributed machines.

Figure 2.1 a) presents the architecture of a shared-memory machine: Multiple processors share a common address space. The processors are attached to the same memory over a bus. In some systems, memory might be local to some processors which means the rest of the processors has a higher latency when trying to access the non-local memory. Still, every processor can access every part of the address space. Communication takes place *implicitly* by writing and reading shared variables. Because data written by one processor can be accessed by another processor in a fast manner, little care has to be taken regarding the decomposition of data. For this reason however, shared memory programs are typically not scalable on distributed machines [SAB$^+$10].

Figure 2.1 b) shows that a distributed memory machine basically consists of several shared memory machines linked to each other via an interconnect. Since processors cannot directly access data stored in the memory of other machines, *explicit communication* is needed in order to synchronize the processors. This is typically done by two-sided communication: The *sending* of a message has to be accepted at the remote machine with a a corresponding *receive* call.

Machines conduct their computations simultaneously and either synchronize in discrete time intervals or exchange data asynchronously. Either way, sending data over an interconnect imposes high latency and low throughput in comparison to the data access over a memory bus in shared memory systems. For this reason, programmers have to carefully decompose data in order to distribute the work load uniformly and minimize communication overhead.

Figure 2.1 c) illustrates the concept of Partitioned Global Address Space: The local portions of memory are unified under a global address space which allows processors to directly access data on remote machines. Data access is performed using one-sided communication: No *receive* call on the remote machine is needed.

Since data transferal over an interconnect is still costly, programmers have to take the same care for data locality as with the traditional message passing approach. To allow for this, the locality of a datum is directly exposed to the programmer.

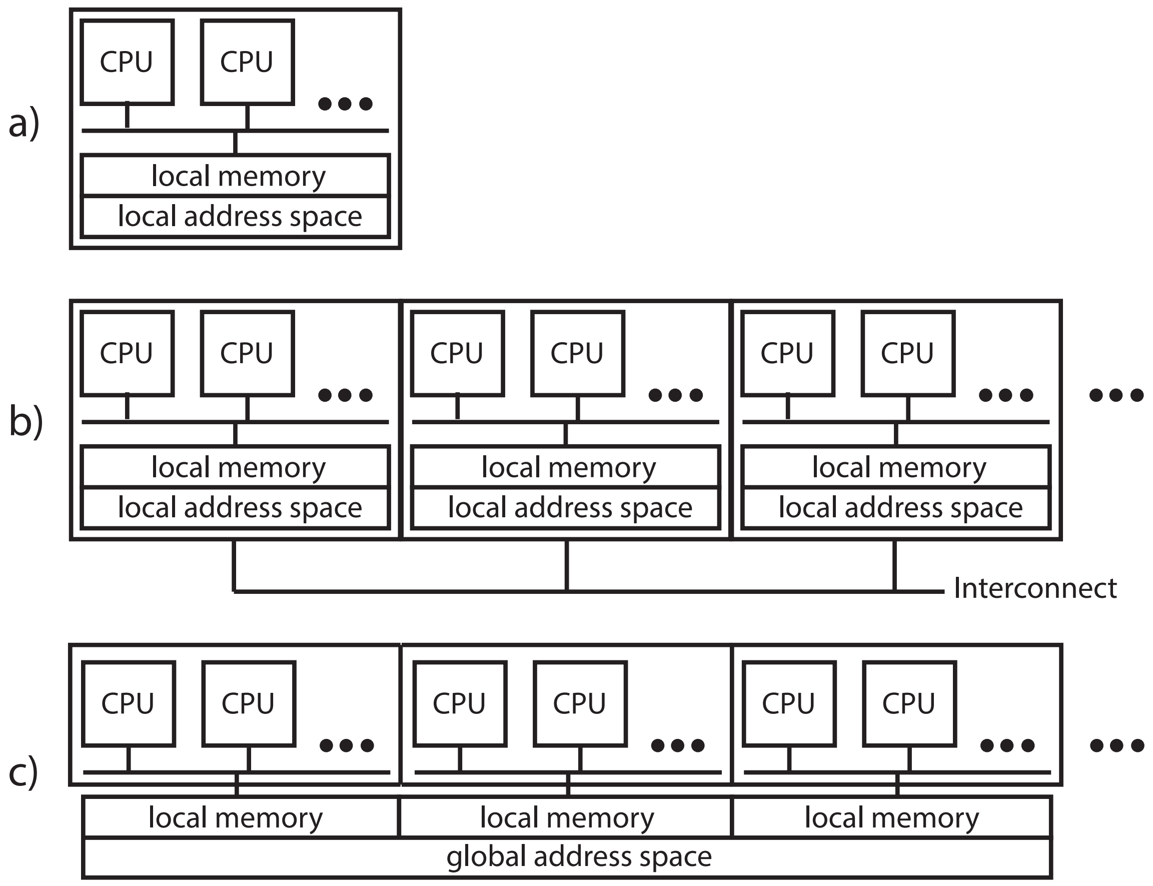

Figure 2.1: View on a) Shared Memory b) Distributed Memory c) Partitioned Global Address Space

Existing PGAS approaches are mainly comprised of dedicated programming languages such as Unified Parallel C (UPC) [C$^+$05], Co-Array Fortran [NR98] or Chapel [CCZ07] that allow for compiler optimizations in respect to distributed machines but lack portability and reach. In contrast to this, efforts exist to create libraries for existing programming languages used by many HPC systems.

2.4 DASH C++ Template Library

DASH [FFK16] is a compiler-free PGAS approach: It consists of a simple C++ library that can be compiled with any C++ compiler and thus can be used out-of-the-box on most HPC systems. The library is part of the Priority Programme "Software for Exascale Computing"

(SPPEXA)[1] which supports research on computing systems achieving 10^{18} floating point operations per second and above. While PGAS languages require existing programs to be completely rewritten from scratch, DASH allows the applications to be incrementally ported and thus facilitates wider adoption of the PGAS model in the HPC community.

DASH operates on top of the *DASH Runtime* (DART) which is a PGAS memory allocation and communication abstraction written in C. DART enables global memory allocation, pointers to remote memory locations and one-sided communication on top of existing libraries like MPI [For12] or GASPI [GS13]. With DART-MPI [ZMI$^+$14], a fully functional DART abstraction on top of MPI-3 is used in DASH releases at the time of this writing.

In DASH, processing elements are referred to as *units*. Units can be any processing element such as threads or processes. DASH programs are implemented using the Single Program Multiple Data (SPMD) model: The data is partitioned onto the participating units and each unit executes the same code on its part of the data. Furthermore, units form *teams* that can be created at runtime. Because HPC hardware topologies become more complex over time (e.g. [KDSA08]), DASH supports hierarchical team creation to allow for a more fine-grained exploitation of data locality compared to the typical local-remote distinction of the PGAS model.

Data is referred to in terms of global pointers and references. A `GlobPtr<T>` object holds information about the unit and local memory location of the referenced datum. It can be dereferenced to a `GlobRef<T>` object which behaves like a C++ reference and can be converted to an object of type `T`. This type conversion triggers a one-sided get operation transferring the data from its remote source to the caller. Similarly, data can be written into the referenced memory location of a `GlobRef<T>` object.

DASH provides a set of containers for distributed data storage. Aside from the static data structures Array and Matrix, dynamic data structures are available. Since the graph concepts of this work belong into the latter category, details of it are discussed in the following.

Dynamic allocation in DASH is encapsulated in the `GlobHeapMem` concept. `GlobHeapMem` offers two basic operations to dynamically allocate memory during runtime: `grow` and `shrink`. These operations increase or decrease the local size of the memory allocated on the respective unit. Changes in memory space are not reflected in global address space until the operation `commit` is called which publishes the changes across all units.

A dynamic container in DASH pre-allocates some memory during its initialization. When the memory is completely used, further additions of elements result in `GlobHeapMem.grow` operations. A call to the `barrier` operation of the container results in all newly added elements of the container to be publicly available on all units.

[1]http://www.sppexa.de

3 Related Work

3.1 Shared Memory

3.1.1 STINGER

3.1.2 Ligra

3.2 Distributed Memory

3.2.1 Parallel Boost Graph Library

3.2.2 STAPL Parallel Graph Library

4 Graph Container Concepts

4.1 Memory Space

4.2 Index Space

4.3 Iteration Space

4.4 Semantics

5 Reference Implementation

6 Case studies

6.1 Static structure

6.1.1 Graph traversal

6.1.2 Shortest path evaluation

6.2 Dynamic Structure

6.2.1 Graph partitioning

6.2.2 De Bruijn Graph construction

7 Evaluation

7.1 Micro-benchmarks

8 Conclusion

8.1 Summary

8.2 Assessment

8.3 Outlook

List of Figures

Bibliography

[C^{+}05] Consortium, UPC u. a.: UPC language specifications v1. 2. In: *Lawrence Berkeley National Laboratory* (2005)

[CCZ07] Chamberlain, Bradford L. ; Callahan, David ; Zima, Hans P.: Parallel programmability and the chapel language. In: *The International Journal of High Performance Computing Applications* 21 (2007), Nr. 3, S. 291–312

[FFK16] Fürlinger, Karl ; Fuchs, Tobias ; Kowalewski, Roger: DASH: a C++ PGAS library for distributed data structures and parallel algorithms. In: *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on* IEEE, 2016, S. 983–990

[For12] Forum, Message Passing I.: *MPI: A Message-Passing Interface Standard Version 3.0.* 09 2012. – Chapter author for Collective Communication, Process Topologies, and One Sided Communications

[GS13] Grünewald, Daniel ; Simmendinger, Christian: The GASPI API specification and its implementation GPI 2.0. In: *7th International Conference on PGAS Programming Models* Bd. 243, 2013

[Gus88] Gustafson, John L.: Reevaluating Amdahl's Law. In: *Commun. ACM* 31 (1988), Mai, Nr. 5, 532–533. http://dx.doi.org/10.1145/42411.42415. – DOI 10.1145/42411.42415. – ISSN 0001–0782

[KDSA08] Kim, John ; Dally, Wiliam J. ; Scott, Steve ; Abts, Dennis: Technology-driven, highly-scalable dragonfly topology. In: *ACM SIGARCH Computer Architecture News* Bd. 36 IEEE Computer Society, 2008, S. 77–88

[Lam13] Lameter, Christoph: NUMA (Non-Uniform Memory Access): An Overview. In: *Queue* 11 (2013), Juli, Nr. 7, 40:40–40:51. http://dx.doi.org/10.1145/2508834.2513149. – DOI 10.1145/2508834.2513149. – ISSN 1542–7730

[NR98] Numrich, Robert W. ; Reid, John: Co-Array Fortran for parallel programming. In: *ACM Sigplan Fortran Forum* Bd. 17 ACM, 1998, S. 1–31

[PTM96] Protic, J. ; Tomasevic, M. ; Milutinovic, V.: Distributed shared memory: concepts and systems. In: *IEEE Parallel Distributed Technology: Systems Applications* 4 (1996), Summer, Nr. 2, S. 63–71. http://dx.doi.org/10.1109/88.494605. – DOI 10.1109/88.494605. – ISSN 1063–6552

[SAB+10] Saraswat, Vijay ; Almasi, George ; Bikshandi, Ganesh ; Cascaval, Calin ; Cunningham, David ; Grove, David ; Kodali, Sreedhar ; Peshansky, Igor ; Tardieu, Olivier: The asynchronous partitioned global address space model. In: *The First Workshop on Advances in Message Passing*, 2010, S. 1–8

[ZMI+14] Zhou, Huan ; Mhedheb, Yousri ; Idrees, Kamran ; Glass, Colin W. ; Gracia, José ; Fürlinger, Karl: DART-MPI: an MPI-based implementation of a PGAS runtime system. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* ACM, 2014, S. 3

[ZZZ+14] Zhao, D. ; Zhang, Z. ; Zhou, X. ; Li, T. ; Wang, K. ; Kimpe, D. ; Carns, P. ; Ross, R. ; Raicu, I.: FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: *2014 IEEE International Conference on Big Data (Big Data)*, 2014, S. 61–70