ABSTRACT

AWESOME C++ DISTRIBUTED LIBRARY FOR EFFECTIVELY UTILIZING PARALLEL MEMORY AND STUFF

by John David Rudie Jr.

This is a really great paper about some great stuff. Keywords: awesome, really cool, great

AWESOME C++ DISTRIBUTED LIBRARY FOR EFFECTIVELY UTILIZING PARALLEL MEMORY AND STUFF

A Thesis (or Thesis Proposal)

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Science

by

John David Rudie Jr

Miami University

Oxford, Ohio

2021

Advisor: Dr. Dhananjai Rao

Reader: Prof. Norm Krumpe

Reader: Dr. Alan Ferrenberg

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Use this chapter to introduce readers to your research.

Discuss motivations, contributions and other introductory material.

Note: In general, the suggestions provided in this template (for content, chapter breakdown, etc.) are just that - suggestions. Consult with your advisor to determine the appropriate thesis structure for your sub-discipline, and adjust accordingly.

1.1 Motivation

1.2 Contributions

This thesis will make the following contributions:

- an improved C++ memory allocator designed for effictively utilizing big memory
- modular distributed data structures that can easily be inserted into existing programs with little change in code complexity and a resultant increase in performance
  - vector
  - unordered multi-map
  - heap
- garbage collector designed for memory hungry programs running on distributed systems

Chapter 2

Background & Related Work

According to the International Data Corporation, the size of global data is expected to grow from 33 in 2018 to 175 zetabytes in 2025 as shown in Figure 2.1. [3] To put that into perspective, the variations in one human genome can be compressed in a lossless fashion to 4 megabytes of data [4], so the size of our data in 2025 will be equivalent to the digital representation of over 4 quadrillion humans.

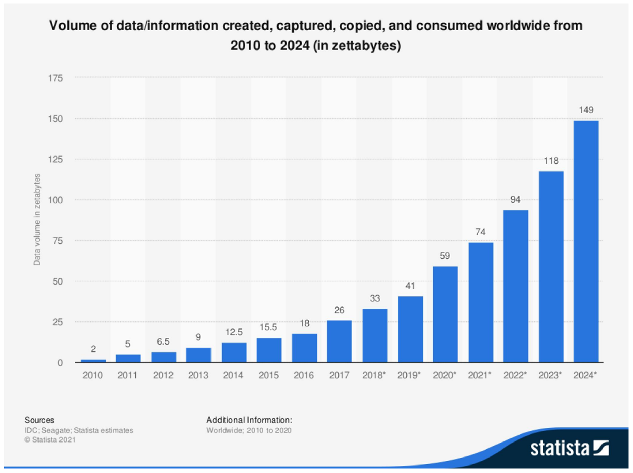


Figure 2.1: Global Data Growth through 2024. Source: International Data Corporation

Along with all of this growth in data comes a growth in the amount of memory with which applications must work. In high performance computing (HPC), we define a computing cluster as a co-located set of computers, each individually called a node, configured for accomplishing a shared task. Administrators have begun to include ”big memory” nodes in these clusters which contain an above average amount of Random Access Memory (RAM) in order to address workloads that require excessive in-memory data. Prominent examples of these workloads include applications that utilize in-memory databases, graph analytics [5], and large simulations.

C++ is one of the most popular languages for distributed computing solutions, partly owing owing to the fact that it offers support for useful object-oriented abstractions in conjunction with highly optimized code. [6] While much prior work has been dedicated to creating distributed libraries in C++ in [7] [8] [9] [10] [11] [12] [13], other separate work has attempted to improve performance on big memory applications through various optimizations [5], and still other work has been dedicated to efficiently managing distributed memory

[14] [15] [16] [17] few, if any, attempts have been made to create a library of standard data structures that are designed with big memory capabilities and work loads in mind. Additionally, few general purpose distributed computing libraries are available to the public under open source distribution licenses, as many provide some sort of proprietary value add. This research will be an attempt to create such an open source distributed computing library optimized for big memory utilization, experimenting with novel implementation techniques for standard data structures and drawing from existing literature when necessary.

2.1 Parallel vs. Distributed Computing

Before detailing the most prominent continuing projects in this space, it is important to draw a distinction between parallel and distributed computing. In order to do that, clear definitions for parallel and distributed computing must be provided. While this thesis project will primarily be geared toward distributed applications running in high-performance computing (HPC) settings, it will also attempt to exploit parallelism, so understanding both is essential for a holistic view of the space in which the proposed project will exist. Definitions included below are a mixture of universal standards used in literature and what will be considered distributed/parallel computing for the purposes of this project.

Parallel Computing There are two main modes of parallelism in parallel computing: instruction-level parallelism and thread-level parallelism.

Instruction-level parallelism refers to design techniques at the processor/compiler level that speed the execution of sequential programs by allowing individual machine instructions, e.g. additions, floating point multiplications, memory stores/loads, to execute in parallel [18]. This is typically done using different components of a processor to perform different parts of a given instruction at the same time. Modern processors typically have, at the very least, several floating point units and several integer units, which can handle different parts of the same instruction.

The main distinction between instruction-level parallelism and thread-level parallelism is that instruction-level parallelism occurs at the processor level whereas thread-level parallelism exploits concurrency among multiple processors within the same computer [19]. The thread in thread-level parallelism refers to a thread of execution in which some part of a program runs its code. Threads are different from processes in that threads share the same memory space and are all added to the same existing process. These attributes give threads a distinct advantage over processes when one is working with multiple processors, as an arbitrary number of threads can work on the same shared data structure, either through synchronization or through splitting the structure up into discrete parts [19].

Distributed Computing The distinction between parallel computing and distributed computing is much like the distinction between instruction-level parallelism and thread-level parallelism. Whereas instruction-level parallelism occurs within one processor while thread-level parallelism occurs in a system with multiple processors, parallel computing occurs within one computer while distributed computing occurs across multiple computers. Naturally, new and unique problems arise when a project migrates from a single system of computers to a cluster consisting of multiple computers that are just as challenging as the problems that arise when making sequential code parallel. Common problems include, but are not limited to heterogeneity in networks, hardware, programming languages, or implementation of standard constructs, openness of components within a distributed system, scalability, fault tolerance, concurrency of different resources across different machines, and unique quality of service (QoS) issues relating to availability, reliability, and performance that do not affect applications running on a single computer [20]. Distributed computing can refer to anything from blockchain networks, which distribute verification of information and transactions across many different systems, to web services architectures, in which a web application is broken into modular, discrete components which then run on different computers and communicate across the network, to cloud computing, in which innumerable virtual private servers are spun off and destroyed in tandem across many machines in a massive data center. This project will mainly focus on HPC settings in which memory-intensive scientific applications are run, but hopefully the deliverable will be able to be extended to cloud computing environments, and perhaps even to desktop computer environments if the memory management methods are applicable at this smaller scale. For the purposes of this project, computers with a discrete GPU or other processing component will not be considered distributed computing since many of the aforementioned challenges are less relevant when communicating between components within the same machine - offloading operations to a GPU is analogous to offloading to another CPU core or offloading cache contents to RAM.

2.2 First-Class Distributed Computing in C++

While the most powerful computers in research in industry continue to expand their processor and GPU counts, distributed computing is not yet included as a first class component of the C++ STL [6]. This forces many developers of high-performance scientific applications to reinvent the wheel for each individual project. Common parallel computing constructs, like threads, mutexes, and other locks, went through a similar process in C++. Exploring the history of other attempts to standardize distributed data structures, as well as analogous constructs, is essential to understand the full picture of distributed computing in C++. First, we will detail the existing constructs for parallel and distributed computing and the history of their rise from community-driven tools to first class members of the STL. Next, we will explore recent attempts to add distributed computing paradigms into the C++ STL.

Existing Constructs Despite the lack of distributed constructs and C++, there are many objects used in heterogeneous/parallel computing that serve as important references for any distributed data structures/algorithms. Most important among the aforementioned constructs are std::async, std::future, std::mutex, std::unique_lock, and std::thread, which are detailed in proposals N1682, N1815, N1875, N1883, N1907, N2043, N2090, N2094, N2096, N2139, N2178, N2184, N2885, and N2889, among others [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32]. A great summary and amalgam of all these proposals can be found in N2320 [33].

To briefly summarize this history, before the ISO C++ STL had any capacity for threading or any concurrency at all, Boost, a popular library that extends the STL, had its own version of threads, locks, and other essentials for concurrency. After the entire Boost community and development team had troubleshooted, iterated, and improved these features in boost, they had an outstanding product with fairly widespread industry adoption. Since the C++ working groups (WGs) have less flexibility and agility in the implementation of a new feature, they did not get around to discussing threading in C++ until after Boost was a full-formed library. As a result, the multi-threading library in the STL is opaquely influenced by Boost. C++ threads, just like boost threads, can be waited on, cooperatively cancelled, quereied, joined with other threads, detached, and otherwise managed. std::mutex in C++ is the mechanism by which threads are locked and synchronized. This ISO adoption of Boost standards is encouraging for any developer who wants to spin up a useful open-source library for use in the community, as it demonstrates that libraries with enough widespread community support can eventually become first-class members of the C++ STL.

The idea of a standardized object for resolving an asynchronous subroutine call arose as a result of the fact that though there were many different multi-threading APIs proposed for the C++ standard, each of these libraries made simply calling another subroutine in parallel a difficult task with high code complexity. The intended domain for std::async was extracted concurrency in existing programs. There existed, and still exist, a number of applications that are purely sequential, having been written before multi-threading was widely possible, and inserting a function call in a couple of places is usually a more realistic goal than rewriting entire applications with concurrency in mind. N2889 gives the example of quicksort: std::async would be appropriate for recursive calls to quicksort, but not to the iteration in a partition, as it was not intended to compete with existing loop-parallelism solutions. std::future is the type returned by a call to std::async, and was proposed for this purpose instead of std::unique_future to avoid unnecessary overhead with the unique_future type. future can be thought of as a more primitive thread, in that it doesn't not provide any method for synchronization.

Threads are relevant for the purposes of this project since thread-level parallelism will be necessary to maintain any concurrent data structure at a node level. Additionally, passing, cancelling, or locking threads from node to node may be essential depending on the implementation of our concurrent data structures, which seems to indicate that either using some implementation of thread pools or creating our own may be necessary. std::future is relevant for work that does not require synchronization, and a distributed implementation of future could be interesting, as it could potentially offer easy speed improvements to poorly written HPC applications.

Recent attempts Among the most compelling recent attempts to include distributed constructs in the STL is the proposal P0443 [34]. Driven by the increasing diversity and heterogeneity of hardware within almost every system from the newest smart phone to HPC clusters, this proposal suggests the creation of a work execution interface, std::executor, and representations of work/the relationships between work, std::sender and std::receiver. The executor interface could be used to represent anything from a thread pool to SIMD units to GPU runtimes to the current thread. Programmers could author executors by defining their own execution functions for each of these very different environments, and the executor interface is likely robust enough to provide support for any other execution environments that might arise in the future. Senders and receivers can represent almost any relationships between work in a flow diagram, allowing for the development of more generic code that acts on an asynchronous dependency graph and can be moved seamlessly in-and-out of systems with significant hardware differences.

An interesting addition to P0443 is the application of affinity to executors as described in [6]. In this context, affinity refers to memory access performance between running code and the data accessed by said code. In this context, a resource has higher affinity with a section of memory if access of that section comes with lower latency and/or higher bandwith [6]. The authors suggest tailoring the executors detailed in P0443 to allow for application developers to preemptively place data/threads in the right place according to affinity, making it so that applications do not have to rely on the operating system to allocate each particular part of memory to the highest affinity segment, perhaps speeding memory accecss and/or achieving more bandwith.

While both of these ideas would likely mesh excellently with this project, unfortunately neither have made it into any recent C++ standards. The current C++ machine model is still strictly CPU focused [6], and thus does not account for GPUs or other heterogeneous components, much less other computers. However, the approach of allocating memory based on affinity within an application instead of relying on the operating system is incredibly relevant for this project, as we will likely have to design our solution to optimize affinity on both a node level and a cluster level, where affinity with each big memory node must be accounted for.

2.3 Memory in Distributed Applications

Since the beginning of the information era, there has been rapid development of better CPUs with increased cores and faster clock times in conjunction with an increase in demand of data accesses in many applications [35]. Since the affordability and speed of RAM has not followed that of CPUs, this means that the memory resources in a distributed system are now much more expensive relative to CPU cycles. As a result of this, any interventions that improve the efficiency of accessing and sharing memory, whether at an operating system level or user level, could greatly improve the profitability and efficiency of existing distributed applications. Literature presents several existing improvements to memory usage in distributed applications like [5] [35], this solution is unique in that it attempts to resolve many of the

Project	Paradigm	Architecture	Nested	Adaptive	Data Dist.	Scheduling	Continued development
STAPL	S/MPMD	Shared/Dist	Yes	Yes	Auto/User	Customizable	Yes
PSTL	SPMD	Shared/Dist	No	No	Auto	Tulip RTS	No
Charm++	MPMD	Shared/Dist	No	No	User	prioritized execution	Yes
CILK	S/MPMD	Shared/Dist	Yes	No	User	work stealing	No
NESL	S/MPMD	Shared/Dist	Yes	No	User	work and depth model	No
POOMA	SPMD	Shared/Dist	Yes	No	User	pthread scheduling	No
SPLIT-C	SPMD	Shared/Dist	Yes	No	User	user	No
X10	S/MPMD	Shared/Dist	No	No	Auto	-	Yes
Chapel	S/MPMD	Shared/Dist	Yes	No	Auto	-	Yes
Titanium	S/MPMD	Shared/Dist	No	No	Auto	-	No
Intel TBB	SPMD	Shared	Yes	Yes	Auto	work stealing	Yes

common pitfalls in distributed memory management at a language level.

2.4 Related Work

There exist a number of comparable libraries that have attempted to standardize commonly used distributed/parallel structures, or otherwise create a standard library for distributed computing in C++. Out of all of the libraries initially created to address distributed computing in C++, however, few are still in continued development in 2021. The table below compares libraries for distributed and parallel computing in C++. Some of these libraries have created their only language (Chapel/Cilk) based heavily on C/C++ specifically to support their run-time system (RTS). Others, like Charm++, have created their own interoperable message passing interface for communication between processors and nodes. One additional significant difference not readily visible on the table is that none of these libraries attempt to solve any of the difficulties inherent in memory-bound distributed applications. Additionally, there are numerous solutions dedicated to speeding distributed computing with large in-memory components, including Spark, Memcached, and Globally Addressable Memory (GAM). Brief overviews of the architecture and contributions of each of these solutions are provided below.

2.4.1 STAPL

Description The Standard Template Adaptive Parallel Library (STAPL) was developed by researchers at Texas A&M University in the late 1990s, far before the C++11 standard provided a set of standardized tools for concurrency. The library was originally created as a super-set of C++'s STL, with the intent to possibly replace the STL on small to medium multiprocessing systems [7]. STAPL is capable of running both single program multiple data (SPMD) and multiple program multiple data applications, and its scheduling algorithm is customizable. STAPL, like the C++ STL, is a generic library, offering data structures and algorithms that can be exploited by a large number of heterogeneous applications. [7]. STAPL implements each distributed data structure as a thread-safe, concurrent object called a pContainer [1].

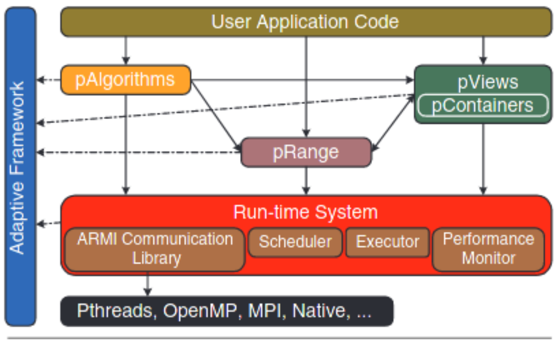


Figure 2.2: STAPL Architecture [1]

STAPL pContainers consist of a finite collection of typed elements, storage space, and an interface of methods/procedures that can be applied to the pContainer. [1] Each pContainer is globally addressable, meaning it provides shared memory address space, and can be composed with other containers to create new structures [1]. Distributed data structures implemented by STAPL include an array, a vector, a list, a matrix, a graph, a map, and a set [36] [37]. The library also includes common algorithms for these data structures like those one would find in the C++ STL.

STAPL facilitates communication between data structures and algorithms by having algorithms communicate with pViews through pRanges in the same way that algorithms in the C++ STL communicate with iterators [1]. STAPL represents most algorithms using compositions of algorithmic skeletons, which include many common interaction patterns in parallel programs; for example, map, reduce, zip, and gather [38]. Providing standard, efficient implementations of these skeletons allows users of the framework to focus on the business logic of their program instead of generic patterns. The code sample below includes an example usage of a few of these algorithmic skeletons.

Like many of the other C++ libraries designed with distributed/parallel computing in mind, STAPL provides its own custom runtime system detailed in [39]. One of the main ideas in STAPL's runtime system is that of a paragraph, which is essentially a directed task graph in which each work items are represented by vertices and dependencies are represented by edges. The runtime system provides executors and schedulers for the tasks detailed in pRanges [40].

Contributions

- **Adaptive Remote Method Invocation (ARMI)**. STAPL provides primitives for registering parallel objects and using Remote Method Invocation on them from any cluster machine. This allows for the asynchronous transfer of data and work throughout the distributed system. ARMI utilizes the future and promise constructs detailed in 2.2.

- **Generic Distributed Containers (pContainers)**. The STAPL parallel container framework provides a good way of abstracting away the implementation details of a given container on a distributed system, giving users a generic interface that works with any parallel container regardless of the type of system or network on which it is exists. In addition to a good interface, STAPL provides its own implementations of containers thatstack up well against other industry-standard implementations in scalability and performance trials. [1].

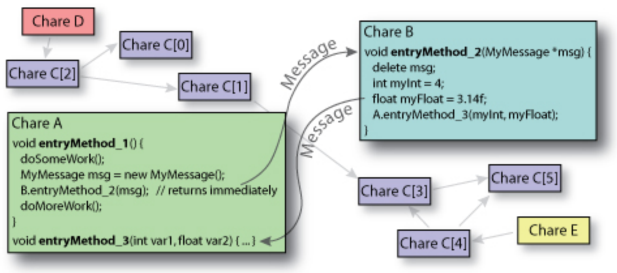
- **Shared Memory Abstraction**. For users who are not working on lower-level applications, STAPL provides an abstraction of the global memory. For more advanced users, STAPL exposes a partitioned global addres space (PGAS) architecture.

  Big Data Graphs. This is a particularly interesting contribution for this project. STAPL's graph library [37] provides a novel approach to out-of-core processing for big data graphs in memory-constricted systems, allowing algorithms implemented in this library to efficiently process graphs that do not fit entirely in RAM. STAPL's parallel graphs use a novel approach that combines RAM and hard disks/SSDs, and works exceedingly well on large distributed memory networks. The approach is comparable to paging, loading subgraphs of a larger graph into available RAM until the whole graph is processed.

Code sample This code solves the first project euler problem, which asks for the sum of all numbers below n that are multiples of 3 or 5. [41]. The code was retrieved from the examples section of STAPL's Gitlab repository [42].

```
typedef unsigned long long ulong_type;

struct three_five_divisor
{
  template<typename T>
  bool operator()(T i)
  {
    return !((i % 3) == 0 || (i % 5) == 0);
  }
};


stapl::exit_code stapl_main(int, char** argv)
{
  ulong_type num = boost::lexical_cast<ulong_type> (argv[1]);

  // Creates array container of unsigned integers that will be used for storage.
  stapl::array<ulong_type> b(num);

  // Creates view over container.
  stapl::array_view<stapl::array<ulong_type>> vw(b);

  // Fills the container with values from 1 to n.
  stapl::iota(vw, 1);

  // For numbers in the container that return true to the three_five_divisor
  //   functor, they are set to 0.
  stapl::replace_if(vw, three_five_divisor(), 0);

  // Adds the total of all elements in container.
  ulong_type total = stapl::accumulate(vw, (ulong_type)0);

  // Prints the total sum.
  stapl::do_once ([&] {
    std::cout << "The total is: " << total << std::endl;
  });

  return EXIT_SUCCESS;
}
```
Listing 2.1: STAPL code sample for Project Euler number 1. Headers and doxygen comments removed for brevity

2.4.2 Charm++



```
Chare A
void entryMethod_1() {
  doSomeWork();
  MyMessage msg = new MyMessage();
  B.entryMethod_2(msg);  // returns immediately
  doMoreWork();
}
void entryMethod_3(int var1, float var2) { ... }
```

```
Chare B
void entryMethod_2(MyMessage *msg) {
  delete msg;
  int myInt = 4;
  float myFloat = 3.14f;
  A.entryMethod_3(myInt, myFloat);
}
```

Figure 2.3: Charm++ Architecture [2]

Description Charm++ is a parallel programming framework designed to ease the development, execution, migration, and decomposition of parallel applications [11]. First developed in the early 1990s, Charm++ may have been one of the earliest frameworks to attempt to address distributed computing issues with object-oriented solutions. When Charm++ was first brainstormed, both object orientation and concurrency were still relatively novel. Charm++ was especially innovative among early object-oriented concurrent frameworks in that it allowed for abstractions of modes of information sharing and communication, provided support for load balancing and prioritization, and put forth a new "chare" object used for programming common data parallel applications [43]. A chare is essentially a parallel process which can spin up more chares and send messages to other chares. In this sense, the Charm++ framework is only a step above MPI in that it provides additional features on top of the bare-bones parallel process management system. Some of these features include data abstractions for information sharing, such as read-only data, accumulators, monotonic/atomic variables, and distributed tables [43]. Additionally, over a decade before the STL accepted future and async as members, Charm++ included its own implementation of future, which is basically follows the same idea as the STL implementation discussed in 2.2 in that the calling process blocks until the value in the future is computed. Unlike STAPL and some other framework's in this domain, Charm++'s runtime system is message driven, and every Charm program can broadly be defined, much like MPI, as an initialization and a message-driven loop [44]. In the initialization, the user defines a main chare and branch chare, initializes a memory manager, crafts queue management devices, and deals with load balancing, among other things. In the mesage driven loop, termed a ”pick and process loop”, the runtime system essentially acts as a work pool manager, comparable to an executor in a thread pool except with messages instead of threads. Users choose how messages are picked, defining distinct message priorities as necessary for different applications.

Contributions

- **Message driven runtime system**. Whereas many frameworks abstract away the concept of messaging such that the user does not have to deal with it, Charm++ is entirely based around laying messaging bare, and all objects are built around sending and processing messages. While this can induce a learning curve for Charm++, it also makes programs written in Charm++ more readable for advanced users.

- **Latency tolerance through futures**. As mentioned in 2.4.2, Charm utilizes largely the same future construct that is used in the C++ STL. This makes transitioning trivially data parallel applications to Charm much easier, as the futures can simply be swapped out.

Code sample This code prints hello world on a single processor using a single chare in Charm++. It is difficult to show any other simple programs, as Charm++ programs often exist across a header file, source file, interface file, and makefile, and involve a nontrivial

[2]

Listing 2.2: Hello World in Charm++

```
// File: main.h
#ifndef __MAIN_H__
#define __MAIN_H__

class Main : public CBase_Main { (1)

  public:
   Main(CkArgMsg* msg); (2)
   Main(CkMigrateMessage* msg); (3)

};

#endif //__MAIN_H__


// File: main.c

#include "main.decl.h" (6)
#include "main.h"

// Entry point of Charm++ application
Main::Main(CkArgMsg* msg) {

   // Print a message for the user
   CkPrintf("Hello World!\n"); (4)

   // Exit the application (5)
   CkExit();
}

// Constructor needed for chare object migration (ignore
// for now) NOTE: This constructor does not need to
// appear in the ".ci" file
Main::Main(CkMigrateMessage* msg) { }

#include "main.def.h" (6)
```

degree of code complexity.

[2]

```
// File: main.ci (interface)
 mainmodule main { (7)

   mainchare Main { (8)
     entry Main(CkArgMsg* msg); (9)
   };

};

// File: makefile
CHARMDIR = [put Charm++ install directory here] (10)
CHARMC = $(CHARMDIR)/bin/charmc $(OPTS) (11)

default: all
all: hello

hello : main.o
   $(CHARMC) -language charm++ -o hello main.o (12)

main.o : main.C main.h main.decl.h main.def.h
   $(CHARMC) -o main.o main.C (13)

main.decl.h main.def.h : main.ci
   $(CHARMC) main.ci (14)

clean:
   rm -f main.decl.h main.def.h main.o hello charmrun
```

Listing 2.3: Hello World in Charm++

2.4.3 X10

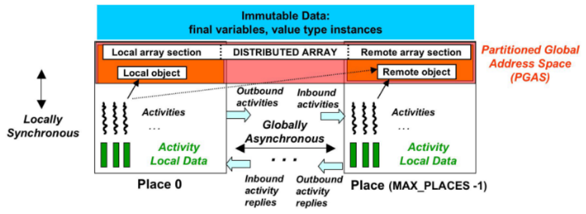


Figure 2.4: X10 Architecture

contributions

2.4.4 Chapel

Figure 2.5: Chapel Architecture

References

[1] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The stapl parallel container framework. *SIGPLAN Not.*, 46(8):235–246, February 2011.

[2] Charm++: Tutorial. `charmplusplus.org/CharmConcepts.html`.

[3] David Reinsel, John Gantz, and John Rydning. The digitization of the world: From edge to core. *International Data Corporation*, page 3, November 2018.

[4] Scott Christley, Yiming Lu, Chen Li, and Xiaohui Xie. Human genomes as email attachments. *Bioinformatics*, 25(2):274–275, 09 2008.

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. *SIGARCH Comput. Archit. News*, 41(3):237–248, June 2013.

[6] Gordon Brown, Ruyman Reyes, and Michael Wong. Towards heterogeneous and distributed computing in c++. In *Proceedings of the International Workshop on OpenCL*, IWOCL'19, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi. Standard templates adaptive parallel library (stapl). In *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '98, page 402–409, Berlin, Heidelberg, 1998. Springer-Verlag.

[8] Maurizio Drocco, Vito Giovanni Castellana, and Marco Minutoli. Practical distributed programming in c++. HPDC '20, page 35–39, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin D. F. Wong. A modern c++ parallel task programming library. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19, page 2284–2287, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Shameem Akhter and Jason Roberts. *Multi-core programming: increasing performance through software multi-threading.* Intel Press, 2006.

[11] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 647–658. IEEE Press, 2014.

[12] D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*, pages 52–60, 2004.

[13] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[14] J. Fu, J. Sun, and K. Wang. Spark – a big data processing platform for machine learning. In *2016 International Conference on Industrial Informatics - Computing Technology, Intelligent Technology, Industrial Information Integration (ICIICII)*, pages 48–51, 2016.

[15] Patrick Hunt, Mahadev Konar, Yahoo Grid, Flavio Junqueira, Benjamin Reed, and Yahoo Research. Zookeeper: Wait-free coordination for internet-scale systems. *ATC. USENIX*, 8, 06 2010.

[16] B. Fitzpatrick. a distributed memory object caching system.

[17] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[18] B Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1), 1993.

[19] Kevin Dowd and Charles Severance. *High Performance Computing Chapter 5.* OpenStax-CNX, 2010.

[20] George F. Coulouris. *Distributed systems: Concepts and design, Chapter 1.* Addison-Wesley, 2011.

[21] Pete Becker. N1682: A multi-threading library for standard c++. `wg21.link/n1682`.

[22] Lawrence Crowl. N1815: Iso c++ strategic plan for multithreading. `wg21.link/n1815`.

[23] Lawrence Crowl. N1875: C++ threads. `wg21.link/n1875`.

[24] Kevlin Henney. N1883: Preliminary threading library proposal for tr2. `wg21.link/n1883`.

[25] Pete Becker. N1907: A multi-threading library for standard c++, revision 1. `wg21.link/n1907`.

[26] Ion Gaztañaga. N2043: Simplifying and extending mutex and scoped lock types for c++ multi-threading library. `wg21.link/n2043`.

[27] Peter Dimov. N2096: Transporting values and exceptions between threads. `wg21.link/n2096`.

[28] Anthony Williams. N2139: Thoughts on a thread library for c++. `wg21.link/n2139`.

[29] Thoughts on a Thread Library for C++. N2178: Proposed text for chapter 30, thread support library [threads]. `wg21.link/n2139`.

[30] Howard E. Hinnant. N2184: Thread launching for c++. `wg21.link/n2184`.

[31] Pete Becker. N2285: A multi-threading library for standard c++, revision 2. `wg21.link/n2285`.

[32] Lawrence Crowl. N2889: An asynchronous call for c++. `wg21.link/n2889`.

[33] Howard E. Hinnant, Beman Dawes, Lawrence Crowl, Jeff Gardland, and Anthony Williams. N2320: Multi-threading library for standard c++. `wg21.link/n2320`.

[34] et. al. Hoberock. P0443: Multi-threading library for standard c++. `wg21.link/p0443`.

[35] Xiaodong Zhang, Yanxia Qu, and Li Xiao. Improving distributed workload performance by sharing both cpu and memory resources. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 233–241, 2000.

[36] Gabriel Tanase, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The stapl parray. In *Proceedings of the 2007 Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture*, MEDEA '07, page 73–80, New York, NY, USA, 2007. Association for Computing Machinery.

[37] Harshvardhan, Nancy M. Amato, and Lawrence Rauchweger. Processing big data graphs on memory-restricted systems. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 517–518, New York, NY, USA, 2014. Association for Computing Machinery.

[38] Mani Zandifar, Mustafa Abdul Jabbar, Alireza Majidi, David Keyes, Nancy M. Amato, and Lawrence Rauchwerger. Composing algorithmic skeletons to express high-performance scientific applications. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 415–424, New York, NY, USA, 2015. Association for Computing Machinery.

[39] Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. Stapl-rts: An application driven runtime system. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 425–434, New York, NY, USA, 2015. Association for Computing Machinery.

[40] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Stapl: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, New York, NY, USA, 2010. Association for Computing Machinery.

[41] Project euler: Problem 1. `projecteuler.net/problem=1`.

[42] Gitlab.com: Standard adaptive parallel templating library. `gitlab.com/parasol-lab/stapl`.

[43] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, October 1993.

[44] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *Parallel Programming Laboratory Technical Report #95-03*, 1994.