DOPC++: EXTENDING C++ WITH DISTRIBUTED OBJECTS AND OBJECT MIGRATION FOR PGAS MODEL

A thesis submitted

to Kent State University in partial

fulfillment of the requirements for the

degree of Master of Science

by

Salwa Aljehane

December,2015

Thesis written by

Salwa Aljehane

B.S., Tibah University, KSA 2007

M.S., Kent State University, USA, 2015

Approved by

Dr. Arvind Bansal ,	Advisor, Master Thesis Committee
Dr. Austin Milton ,	Members, Master Thesis Committee
Dr. Angela Guercio ,	Members, Master Thesis Committee

Accepted by

Dr. Javed Khan ,	Chair, Department of Computer Science
Dr. James L. Blank ,	Dean, College of Arts and Sciences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First of all, Alhamdulillah and Thank to Allah S.W.T. for helping me to complete this thesis successfully. I would like to thank my advisor, Dr. Arvind Bansal for his valuable advice and guidance during this research. I am also thankful to my committee members Dr. Austin Melton and Dr. Angela Guercio for accepting to be my committee. Special thanks goes to my lovely sister Najwa for always being there for me and helping me stay motivated. I want to express gratitude to my dear friends Maha Thafr, Hanan and Safa for listening, offering me advice, and supporting me through this entire process. Last but not the least, I am thankful to Mrs. Marcy Curtiss, the graduate program secretary at the Department of Computer Science at Kent State University, for giving me advice and solving many problems related to graduate program.

Salwa Aljehane

November 13 2015, Kent, Ohio

DEDICATION

I dedicate this thesis and give special thanks to my dearest family: my lovely parents Ayshah and DhifAllah, my husband Salah, my daughter Razan and my son Sultan for their love, support throughout my life.

Abstract

With the growing technology, the number of the processors is becoming massive. Current supercomputer processing will be available on desktops in the next decade. For mass scale application software development on massive parallel computing available on desktops, existing popular languages with large libraries have to be augmented with new constructs and paradigms that exploit massive parallel computing and distributed memory models while retaining the user-friendliness.

Currently, available object oriented languages for massive parallel computing such as Chapel, X10 and UPC++ exploit distributed computing, data parallel computing and thread-parallelism at the process level in the PGAS (Partitioned Global Address Space) memory model. However, they do not incorporate: 1) any extension at for object distribution to exploit PGAS model; 2) the programs lack the flexibility of migrating or cloning an object between places to exploit load balancing; and 3) lack the programming paradigms that will result from the integration of data and thread-level parallelism and object distribution.

In the proposed thesis, I compare different languages in PGAS model; propose new constructs that extend C++ with object distribution and object migration; and integrate PGAS based process constructs with these extensions on distributed objects. Object cloning and object migration. Also a new paradigm MIDD (Multiple Invocation

Distributed Data) is presented when different copies of the same class can be invoked, and work on different elements of a distributed data concurrently using remote method invocations. I present new constructs, their grammar and their behavior. The new constructs have been explained using simple programs utilizing these constructs.

CHAPTER 1

Introduction

1.1 Motivation

As technology continues to grow, the number of processors is increasing. However, productivity is the most important issue that faces high performance computing [1] against the backdrop of existing software library and the existing familiarity with the programming paradigms that programmers know. Currently, tens of thousands of processors are part of a computers generating peta-scale (10^{15} instruction/second), and it is anticipated that in by the end of next decade, we will have exa-scale (10^{18} instructions/second) computing power. This much processing power needs to be fully exploited to solve grand challenge problems such as health science, weather science, agricultural science, space science, managing Internet of things, modeling population related problems at the global and regional scale that will generate huge amount of data.

While, the techniques to handle big data are being developed, processing of big data will require the development of programming tools that can map solutions on high performance massive parallel computers with massively large number of processors. The mapping techniques should be user friendly, known paradigm friendly, and should be downward compatible to use existing library while exploiting the full capability of massive parallel processors for high performance computing.

The exploitation of massive parallel computers requires that the new languages support task parallelism and data parallelism. Task parallelism splits a task into multiple independent subtasks to be executed concurrently using different threads. These threads may be mapped on different processors dynamically. Data parallelism broadcasts instructions on multiple processing elements that work on different data elements in concurrently.

The requirement of paradigm friendliness requires that existing user-friendly programming paradigms be incorporated into the language. Currently, there are three paradigm that are used in user-friendly limited-processors desktops and laptops. The paradigms are: 1) procedural programming; 2) object-oriented programming; and 3) event based programming. Procedural programming has been extended to handle large scale computing using SPMD paradigm (Single Program Multiple Data) – an extension of SIMD (Single Instruction Multiple Data) where instructions have been replaced by threads. The constructs like *forall* and *foreach* are supported by many languages including high performance computing languages such as Parallel Fortran, X10, Chapel and lately UPC++.

The Defense Advanced Research Projects Agency (DARPA) which generate The High Productivity Computing Systems (HPCS) program concentrate its research to provide High Productivity Computing (HPC) systems that could be used for national security and for industrial applications[2]. An HPC system would be able to optimize several aspects in the high performance-computing (HPC) domain, such as its execution time, software development time, system administrative and maintenance fees. In addition, researchs in HPCS help to a reassessment of the performance, programmability, robustness, and productivity definitions and measurements within the programming community.

1.2 Previous Work and Current Limitations

The development of language for large scale parallel computing to exploit data parallelism and task parallelism dates back to early 1980s. The work has been done for programming languages that support: 1) distributed computing using object mobility as in Emerald, Java, PHP and many agent based languages developed on top of Java; 2) procedural languages supporting data parallelism using *forall* and *foreach* constructs; 3) procedural languages supporting concurrency by spawning multiple threads. However, massive parallel processors pose its own problem about how to map problems and groups of activities among the processors.

One recent popular model for inter-process communication is PGAS (Partitioned Global Address Space) model (see Figure 1). In PGAS model, the distributed address space is divided into multiple local spaces which are connected using a global address space. The communication between local address spaces is done using global address spaces and message passing. The local spaces support multiple concurrent threads each with their own data area and a common shared space called heap.

Other underlying model for inter process communication is MPI [3] (Message Passing Interface) that is convenient for use in distributing address space. However, overhead of message passing is an issue that was addressed in MPI [4]. On the other hand, PGAS support local memory as well as distributed address space. Thus PGAS is much more efficient while supporting limited amount of distributed space. The use of partitioned address space and the global synchronization in PGAS based languages increases the productivity [1] and the efficiency of execution.

Using object oriented programming in developing the software on massive parallel computing provides the modality and the reusability features in these languages [4]. With the four main principles of object-oriented programming which are encapsulation, data abstraction, polymorphism, and inheritance, OOP became the solution of the complex software programs being developed. The past two decades have produced programming tools and techniques, which focused on object-oriented applications. Also, many languages that are widely used were created to support OOP, such as Java, Ruby, C++, C#, Objective-C [5] and PHP.

For the last ten years, many object-oriented high performance languages are being developed for PGAS model under DARPA initiative that support large scale computing while retaining object-oriented programming of the single-processor current languages. The major examples of these languages are Chapel[6, 7], X10 [8-10] and UPC++ [11]. All these languages support global partitioning of distributed data like arrays that interface with different local address spaces. Thus, distributed data can be processed concurrently by multiple concurrent threads that execute concurrently. These languages support traditional object oriented programming, distributed arrays, single program multiple data paradigm, asynchronous computation, and invoking remote threads for performing some computations. UPC++ also supports runtime distributed memory allocation.

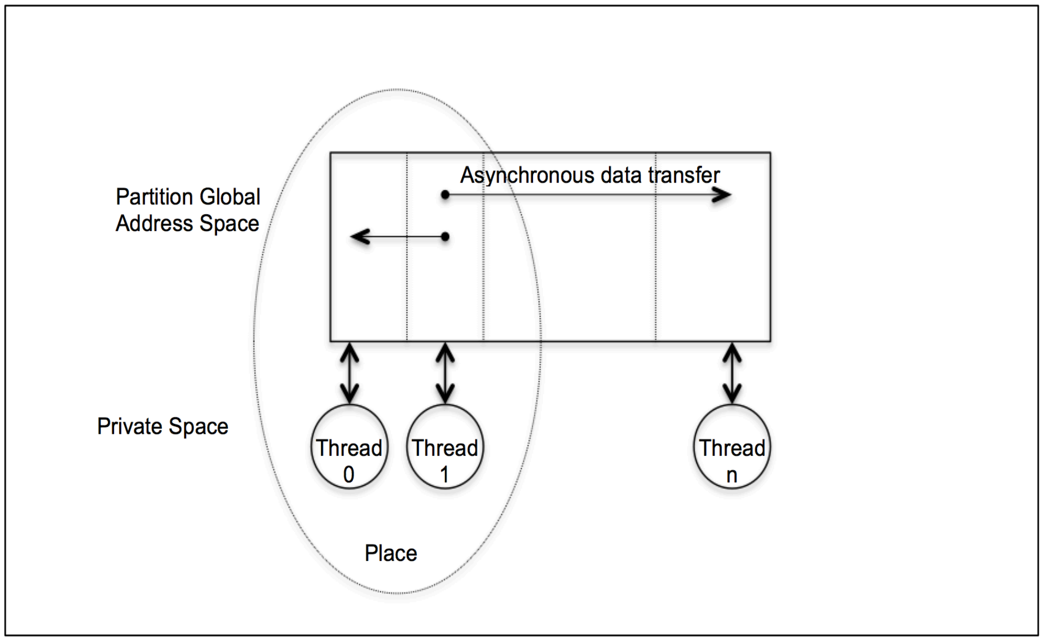

Figure 1:1 Partitioned Global Address Space Model

1.3 Limitations of Existing Languages in PGAS Model

Currently, available object oriented languages for massive parallel computing such as Chapel, X10, UPC++ exploit distributed computing at the process level in the PGAS (Partitioned Global Address Space) memory model. However, their works have some limitations such as the following:

They do not incorporate any extensions at the object level to exploit the PGAS model. For example, these languages do not support constructs that support object migration and remote method invocation described in Emerald and Java, and object cloning used in Java and agent based languages. These languages do not support constructs that support object distribution, dynamic distribution of objects with dynamic change in distribution. Although Chapel, X10 and UPC++ are proposed as object-oriented languages that support parallel programming, the programmer can deal with shared or local variables without any flexibility of migrating or cloning an object between places [1]. One way to provide the task parallelism in PGAS languages, such as spawning a new thread in a remote place without any communication, can be done between places by using the method-invocation. Through a method-invocation, an object in such a place can invoke an operation defined in another place. In addition to the lack of object distribution and object-mobility, these languages do not support a new class of constructs that can be developed by integrating object-mobility with distributed data. One such as class of operations developed during this thesis is MIDD (Multiple Invocation Distributed Data) in which a class template is automatically distributed to a region that is set of logical processing nodes called places in X10, and multiple copies of the same class can be invoked, and work on different elements of a distributed data. The PGAS based languages have also not borrowed some of the object mobility constructs and the notion of flat objects from Emerald that can improve the execution efficiency of program using objects because flat objects can easily migrate, and do not have to carry the inheritance information with them.

1.4 Objectives of this Research:

The main objective of this research is to: 1) comparatively analyze major PGAS languages among themselves and with other languages supporting distributed objects such as Emerald to identify the limitations related to object-based programming with multiple nodes; 2) develop new constructs that are supported by object migration and distribution; 3) development of constructs that integrate object distribution and migration with distributed data objects; and 4) develop high level distributed object-based high performance computing constructs that is user-friendly for better adoption while exploiting massive parallelism.

The specific objectives of this thesis are:

- To extend C++, the language which has been widely used in High-Productivity Computing, to incorporate object distribution, object migration, distributed data arrays, single program multiple data and multiple object invocation on distributed data. The new language DOPC++ (Distributed Object based PGAS for C++) is proposed for the PGAS model.

  The new language DOPC++ has been offered to handle the parallel computing on desktop with distributed objects and object migration. Simple programs have showed to explain the constructs.

Why C++:

In order to design DOPC++, a parallel distributed language is made by adding some high abstraction features on top of an existing language. Thus, they cooperate to present a new language that works in the PGAS model. C++ has been chosen as a base language for different reasons:

1. C++ is known as the most efficient object oriented language [12].

2. C++ is a widely used language. So, that will give a user a chance to quickly understand the extension.
3. C++ supports the object oriented programming. Therefore, using C++ is a good way to simplify the distributed computing that is using the object.
4. A lot of framework and environments for object-distributed programming are built by using C++ such as CORBA ORBs, Network OLE, and HP OODCE, which are commercial tools [1].

1.5 Methodology

In the proposed thesis, the following steps are made and taking in account to achieve the goal:

- Several memory models for massive parallel computing are discussed.
- Different massive parallel languages supporting object oriented programming have been studied.
- A comparative study of different languages in PGAS is proposed.
- New constructs are proposed that extend C++11 with object distribution and object migration; and integrate PGAS based process constructs with these extensions on distributed objects and object migration.
- A DOPC++ grammar and description is presented.
- The new constructs are explained using simple programs utilizing these constructs.

1.6 Contributions

The basic contributions in this thesis are the following:

1. C++ language has been extended. The key idea is to extend the C++ class into three categories: *distributed class*, *local class* (including nested as well as flat classes), and *flat class*. In addition, it supports the migration of objects and the cloning of distributed objects. In this approach, the object can connect through the partition address space.
2. The language DOPC++ supports higher level synchronization primitives by using the sync declaration with shared data, and methods, and borrows and incorporates the notion of monitor to provide synchronization between methods.
3. The language supports dynamic growth (and shrinkage) of the region. It also provides automatic copying of class templates into dynamic regions. These multiple copies of class templates can be dynamically instantiated to create multiple copies of objects executing concurrently. However, the distribution is hidden from the programmer.
4. It provides DOPC++ grammar to include additional features.
5. Simple examples are provided to explain the DOPC++ construct.

1.7 Organization

The overall organization of this thesis is structured into eight main chapters, described as follows:

Chapter 1 is an introduction that presents the motivations, the thesis contribution, and the methodology of this thesis in describing the problem. In addition, the limitation of the previous PGAS language has been explained.

Chapter 2 includes different important definitions that are presented in the background of this thesis.

Chapter 3 covers a literature review for different PGAS languages as well as Emerald as a model of a distributed object based language.

Chapter 4 discusses several PGAS languages and compares them. Also, it describes the Emerald Object based language; and shows the comparison between the Emerald and PGAS languages. In addition, this chapter describes C++11.

Chapter 5 explains the grammar of additional features in DOPC++. This chapter also includes the programming construct and semantics for DOPC++.

Chapter 6 provides a simple example to explain the DOPC++ construct.

Chapter 7 talks about some of the related work in designing a new object oriented parallel programming language.

At the end, **chapter 8** concludes the thesis and discusses limitations and future works.

CHAPTER 2

BACKGROUND AND DEFINITION

2.1 Object Oriented Programming

2.1.1 Class and Objects

Object Oriented Programming (OOP) is centered on the creation and manipulation of *objects*. These objects are the "building blocks" of any Object Oriented program [13]. Each object is comprised of two parts: data and behaviors. The data of the object is referred to its attributes, which are used to show differences between objects. The behaviors of an object are called its methods: procedures, functions, and subroutines that are invoked on the object.

In OOP, the scheme for object is called a *class* and the class is the basis to create the object [13]. It is often referred to as a template for objects used in the program.

2.1.2 Subclass and Inheritance

Object Oriented Programming allows for the creation of new classes through the concept of inheritance. Inheritance allows a class to reuse the attributes and methods of a different class. Through this process, classes can form a hierarchy: some classes can have a more generic definition than others. Instead of creating a completely new class, inheritance provides the means for a convenient and structured model.

Those classes from which information is inherited are called superclasses, or parents; while classes that inherit the data are called subclasses, or children [13]. As an example, the class employee could be a superclass of the subclasses “manager” and “scientist”. These subclasses would inherit all capabilities of “employee”, but also have their different capabilities.

2.1.3 Extending Class Definitions

As discussed in the previous sections, class definitions may be extended in subclasses to provide a specific template for the objects. As the hierarchy of classes becomes extensive, the power of inheritance lies in the abstraction and organization techniques applied to the program [13]. In some OOP languages, such as C++, multiple parent classes are allowed in a single class, while other languages such as .NET and Java only allow single inheritance.

Polymorphism:

The concept of polymorphism is literally translated as “many shapes” [13]. This model allows subclasses to implement their version of a method that already exists in their superclass. This process is called overriding, which means replacing an implementation of a parent with one from a child [13]. Polymorphism allows each subclass to be permitted to respond differently to the same method call.

Casting:

Casting is used in Object Oriented Programming to do type conversion between the parent class and subclass while using the inherited method or inherited data entity in subclass. To avoid error, when using casting the compatibility of data should be taken into account so that information is not lost [4].

2.1.4 Methods

Methods implement the required action of a class [13]. Each object has a set of methods defined in its class. Calling methods from outside of the class are described as public methods, while methods that are visible only within the class are called private methods.

2.2 Concepts in Massive Parallel Computing

2.2.1 "Race condition"

In Parallel Computing, subtasks are referred to as threads, and they modify shared values and data. When two processes operate simultaneously on a variable, the result of both threads will most likely be incorrect. This condition, in which the result of a calculation depends on the speed at which the threads execute, is called the race condition [14].

Synchronization mechanisms such as barriers are used to solve race conditions in parallel programming. A barrier would help processes to wait up until a certain condition. This procedure ensures that prerequisite calculations are completed before proceeding to the next stages to prevent an incorrect result.

2.2.2 Atomicity and Synchronization

Another concern in parallel computing synchronization is the process of contention. Contention relates to the situation when more than one process wants to alter the state or value of a data structure or a shared variable simultaneously [14]. As such, it happens when processes must modify shared variables. To address the problem of contention, spin-locks are used to implement the following rule: only one process should ever be able to update a shared variable at a time [14]. To ensure this rule is followed, the call to the spin-lock must be atomic. Atomicity guarantees that only one process at a time enters the shared region.

2.2.3 Critical Sections and Locks

The code between the program's acquire and release methods which is the body of the atomic operation is called the critical section [15]. When two or more critical sections access the same location and at least one of these sections write to that location, then the same lock must protect the critical section. Programming discipline usually guarantees this property by associating the data with the lock that protects it. A thread should then acquire locks for all data that access in the critical section.

2.3 Models for Massive Parallel Computing

2.3.1 Shared Memory "Open MPI"

An example of a shared memory programming is the threads model. In this model, a single process can have multiple execution paths running simultaneously. These "light weight" thread implementations are usually made up of two components: a library of subroutines being called within parallel source code, and a collection of compiler instructions found in either serial or parallel code[16]. In both cases, the programmer should ascertain parallelism in the code.

OpenMP [16] was created through the collaboration of hardware and software vendors, organizations and individuals. It is an industry standard, compiler-directive-based, portable, and multi-platform implementation available in various languages like C/C++ and Fortran. The implementation of OpenMP provides for “incremental parallelism” and allows an implementation to begin with the serial code.

2.3.2 Distributed Memory “MPI”

The Distributed Memory Model, or Message Passing Model, is a parallel programming construct that allows the exchange of data between tasks residing on different machines through communications by sending and receiving messages. This group of tasks uses their local memory when executing computations and may reside across an arbitrary number of machines[16]. Cooperation between these processes is integral to the data transfer portion of this model; each send operation must have the corresponding receive operation. Also, it is the programmer’s responsibility to determine parallelism. The Message Passing Interface, or MPI, was released in 1994 and is the industry standard for message passing. It exists for nearly all the major parallel computing platforms [16].

2.3.3 Partitioned Global Address Space (PGAS)

The Partitioned Global Address Space (PGAS), memory model has been proposed in order to overcome the limitations presented by the Shared and Distributed Memory models. The Global address space which is the set of all address spaces available to the processors in a massively parallel machine, can be partitioned by the PGAS model by using programmer-defined instructions. In this model, multiple processors work on different partitions simultaneously, and each partition may be accessed locally by different threads or activities[4]. In addition, threads can access remote locations asynchronously.

There are many languages using the PGAS model such as Unified Parallel C, Coarray Fortran, Titanium, X10, and Chapel. They are being developed by large industries such as IBM and Cray Inc. and are progressing in the direction of the application demand.

2.3.4 Single Program Multiple Data (SPMD)

The Single Program Multiple Data[17], or SPMD, is the model of parallelism which is comprised of a set of threads that work in parallel to execute a single program. SPMD is the most widely used on large-scale machines. In SPMD, the programmer determines the number of threads which should be fixed during the program that is running. To keep these threads working in parallel, different operations can be aggregate such as *barrier* [17].

2.4 Distributed Object Based Model

2.4.1 Object Migration

Object Migration in Distributed Object Systems has been addressed in several programming languages such as Emerald. Emerald was the first language to propose and implement the concept of object mobility in a networked environment [18]. The objects within the program are allowed to move from node to node according to the programming language commands. Migration also allows all sizes of objects from small to large processed data to have movement. Each node retains the information of local and remote objects; locating an object is a feature that Emerald provides for simply tracing forward through the path of references[18].

2.4.2 Remote Method Invocation

Remote Method Invocation framework [19] allows the components of the distributed application to communicate between each other by object invocation. Specifically, one node can obtain a remote service by invoking the method of the object that executes this service; with RMI [19] methods on a remote object are invoked. In addition, RMI allows interaction between the server and the client over the net. During the invocation process, the client program sends a request to the server program to access the software then the server reacts by making the software accessible to the client program.

2.4.3 Distributed Objects

The distributed object model is popular because of the natural mapping that exists between the common distributed systems model of communicating entities – such as client-server interaction – and the model of communicating [20].

A challenge in the design of distributed-object systems is scalability, or the capacity of a system to allow working in the increase and decrease of data. This is especially relevant when the system expects a rapid increase in the number of clients of an object; of the number of objects present within the system; or of the distance between the client and objects[20]. Also it is important to take into account the performance and flexibility of the system as well as the possibilities for communication delay and partial failures [20].

2.5 Definitions in Distributed Languages with Object Oriented Programming

2.5.1 Definitions in Emerald

Flat object and conformity:

Emerald doesn't support any hierarchical structure in objects. In Emerald, an object is flat. It is also a clean Object Oriented language with fully integrated distribution facilities. Object has its own code and it is not a member of a class. Instead of inheritance, Emerald uses *conformity* in the meaning of inclusion. In Emerald [4] an abstract type T1 conforms to another abstract type T2 only if we have object of type T1 can substitute for an object of type T2. Also, an object of type T1 may conform to an object of the type T2 under these conditions: T1 has all operation of T2, the number of operations in T1 and argument of types have to be the same as in T2, and the type of argument and result of T1 conforms to the type of argument and result in T2. Since conformity supports the notion of inclusion, it does not have code sharing. This is important in Emerald because the location of object can be maintained so that the ancestor classes are not needed.

2.5.2 Definitions in Chapel

This section explains some definitions of concepts needed to understand Chapel.

Domain:

Domain in Chapel is a way to represent the array's index set. So, a domain is used in Chapel to define arrays. It can also be named and passed between different function. Furthermore, domain supports different features such as iteration, intersection and operation in order to create other domains. Therefore, to declare, slice, and reallocate chapel's arrays, domains are used [21].

Data Parallelism:

Data parallelism refers to how array elements are distributed across multiple partitions of the PGAS model [4].

Task Parallelism:

Task parallelism refers to how to distribute the threads execution (or activities in X10) in different partitions [4].

2.5.3 Definitions in X10

This section explains some definitions of concepts needed to understand X10.

Spawning multiple threads:

When an activity needs to access data in the other place, it must spawn asynchronous activity to process the statement in that place**.** The *async* operation is used in X10 to spawn a new thread in order to access a remote memory [4].

Region:

Region is the set of indices under which an array can be divided across this region [4]**.**

Place:

In X10 [4] the multiple activities that share one partition in the global address space are called place. However, each place can run one or more activities.

Activity:

X10 uses the concept of activities instead of threads of execution. In other words, in X10 the PGAS threads are activities which can be created while the program is running [22].

2.6 List of Acronyms

IBM	International Business Machines Corporation
HPCS	High-productivity computing systems

MPI	Message Passing Interface
PGAS	Partitioned global address space
SPMD	Single program, multiple data
UPC	Unified Parallel C

CHAPTER 3

Literature Review

In this section, literature survey of various popular programming models for high performance computing (HPC) and the corresponding HPC languages are presented. The programming models and the language constructs are compared, and their limitations are explained.

3.1 High Performance Computing Models

There are three popular programming models for high performance, namely the *Message Passing Interface* (*MPI*) *model*, the *Partitioned Global Address Space* (*PGAS*) *model*, and the *shared memory OpenMP model*. The following subsections describe each of the three models, and the last subsection compares these models with regards to supported memory structure, execution, supported programming languages and data structures.

3.1.1 Message Passing Interface (MPI) Model

Message Passing Interface (MPI) is a parallel programming model that uses message interchange to communicate between processes [3]. MPI has been a popular message passing library for High-Performance Computing (HPC) applications on distributed architectures for the last two decades [3]. The library consists of message passing operations that specify names, calling sequence and subroutine results or subprogram functions that can interface with popular languages such as Fortran and C. Computation using MPI is usually organized through a collection of processes that communicate with each other by sending messages [23]. There is private address space for each process that other processes cannot access. Besides message passing, there are no other means to have shared address space.

This model consists of one or more MPI processes per SMP node or multi-core processor, which are also made up of multiple threads. Available Symmetric Multiprocessing (SMP) machines and the multi-core processors message passing are mixed with multi-threading [3]. MPI is best used for portable applications that require parallel tasks, and for support of dynamic data structures [3].

3.1.2 Partitioned Global Address Space (PGAS) Model

The Partitioned Global Address Space (PGAS) model provides a global address space and an explicit SPMD control model [3]. Its implementation is distinctively defined by local as well as remote memory references. The PGAS model extends the shared memory model into a distributed memory setting. This allows for computation to be distributed across a machine using global address space and spawning of remote threads [23]. In the PGAS model, address spaces are shared by SPMD threads, and a part of these shared spaces is local to each process [23]. The data structures in this model can be distributed either privately or globally. Global data structures are distributed across address spaces, and they can be explicitly manipulated by a programmer.

Multiple middleware such as *Global-Address Space Networking* (GASNet), *Aggregate Remote Memory Copy Interface*, and *Kernel Lattice Parallelism* (KeLP) have been developed for the PGAS model [3]. Many high performance and high productivity languages such as X10, Chapel, Titanium and Fortress have been developed for the PGAS model.

In spite of its many advantages, PGAS is not suitable for the general environment due to some of its drawbacks [3]. One of them is that PGAS doesn't have dynamic spawning activities. As a result, there is difficulty in adopting the model to non-HPC applications, which are non-data parallel applications. Due these limitation the APGAS (Asynchronous Partitioned Global Address Space) model [23] extends the PGAS in two attributes: async and place. X10 programming language is a appropriate language that incorporate theses two features.

3.1.3 Shared Memory (OpenMP) Model

The OpenMP Model is a multithreaded shared memory parallel programming model [3]. This model works at a higher concept level compared to a simple thread based model. It has the objective of easing shared memory parallel programming [24]. The OpenMP model supports HPC programs because of its portability on shared memory architectures. OpenMP [3] uses a combination of compiler directives and runtime pragmas to create threads, perform synchronized operations, and manage shared memory.

Through the years, different versions of OpenMP have been released adding new specifications and enhancements. OpenMP version3.0 was launched in 2008. It supports explicit tasks that ease the parallelization of different applications like graph algorithms and dynamic data structures [3]. OpenMP was hindered in the past due to the high cost of traditional multiprocessor machines, but the interest of the computing industry in OpenMP was renewed when multi-core processors were made available [3].

3.1.4 Comparison between MPI Model , PGAS Model and OpenMP

Table 3.1 summarizes the comparison of various HPC models discussed in subsection 3.1. As described in Table 3.1, MPI is a SPMD (Single Program Multiple Data) model working on distributed memory using message passing. There is no shared address space; data structured are logically fragmented; and the programmer has to work on the fragmented data structures. OpenMP supports shared memory and global address space. It supports shared memory arrays that can be accessed by different processes. The execution models support multiple threading. The PGAS model supports local processing as well as distributed processing using a global address space and the spawning of remote processes.

3.2 Chapel

Chapel, or Cascade8 High Productivity Language, is a multithreaded high productivity computing language designed and developed by Cray Inc. Chapel is not built on an existing language to distinguish itself from other sequential languages [25]. Its syntax, however, derives from pre-existing languages such as C, Fortran, Java and Ada. Chapel uses modules to divide its programs and operates on two types of classes: traditional pass-by-reference and pass-by-value classes[25].

Table 3:1 Summary of Comparison of Various HPC model

	MPI	OpenMP	PGAS Model	
			X10	**Chapel**
Memory model	distributed memory	shared memory	PGAS	PGAS
Programming model	SPMD	global-view parallelism	global-view parallelism	global-view parallelism
Supported Data structures	manually fragmented	shared memory arrays	global-view distributed arrays	global-view distributed arrays
Execution model	SPMD	shared memory multithreaded	distributed memory multithreaded	distributed memory multithreaded

3.2.1 Execution Model – Mapping on PGAS Model

PGAS (Partitioned Global Address Space) memory models are supported by Chapel; the user code may refer to any variable as long as it is lexically visible, regardless of its location in memory. According to Chamberlain, [6] for remote variables Chapel's compiler and runtime implements the necessary processing that would allow the variable to be accessed over the network during execution. In addition, the location of a variable may be reasoned statically or dynamically, either through Chapel semantics or execution-time queries respectively. The Chapel structure allows higher-level abstractions to be built upon lower-level concepts in the language.

Tasks are used as units of computation which are expected to execute in parallel. Threads would then execute these tasks, accruing further tasks within the same process or on a remote one [6]. As such, Chapel's execution model is more dynamic and general than SPMD models.

Chapel adopts a global view model which means that the program starts with one thread then, based on the construct written by the programmer, new threads can be spawned [4]. The data distribution and logical partitions are user-defined. The logic behind that is different high level of parallel problems need different architectural configurations.

3.2.2 Synchronization

There are different synchronization mechanisms in Chapel such as *mutual exclusion of shared space* and the *use of synchronization variables*. Synchronization variables have two states: "full" or "empty" that serve as a read and write guards [6]. By default, read blocks until the synchronization-variable-state is deemed “full.” After the value is read, the synchronization variable enters in an “empty” state. In the same way, write blocks until the synchronization-variable reaches in "empty" state. After the value is written, the synchronization variable ends up in a "full" state. In addition, Chapel uses the prefix *sync* which means that all tasks, including during the execute, must be complete before continuance [6].

3.2.3 Data parallelism

Data parallelism [6] is grounded upon the goal of making algorithm implementation independent of its input data. Parallel and distributed data structures are supported by a global view of the data instead of dividing and distributing the data between available processors of the machine. The constructs *forall-loop*, *domains*, *ranges* and *array* are the basic data parallelism features in Chapel. The feature *domain* is a set of indices in array to logically distribute array-indices. The construct *forall* is used for data parallel execution on arrays of data-elements. Chapel also supports *coforall* loops which use one thread in each iteration .For Instance, if a *coforall* loop has ten iterations then ten threads will run in concurrently, while *forall* loop uses any number of threads to execute one iteration. Thus, a *forall* loop is the best way to distribute computations across processors [6].

3.2.4 Task parallelism

Task parallelism is supported through the synchronization variables *single*, which is assigned a value only once in its lifetime, and the prefix *synch*, which may be assigned multiple times [25]. Task-parallel features are present in Chapel [26] through three types of constructs: *cobegin*{stmts}, *coforall,* and *begin{stmt}*. The different between *cobegin*{*stmts*}and *coforall* according to Khaldi et al. [26] is that *cobegin* starts a concurrent non-sharing task for each embedded statement within cobegin … coend construct, while every iteration in *coforall* loop is a different task.

3.3 X10

X10 [10]is a PGAS-based object oriented for high productivity computing language that is currently being developed at IBM . This programming language is part of the goal to design adaptable, scalable systems. X10 also concentrates on the technical objective of hardware-software codesign that unites the advances from both fields [9].

X10 is specifically for large-scale parallel applications, with a target of 10 times improvement in development productivity [9].

3.3.1 Execution Model – Mapping

The execution model of X10 [10] is founded upon five goals: 1) to introduce a new programming language; 2) to use Java to provide homogeneous environment; 3) to develop on Partitioned Global Address Space (PGAS) that had an explicit consideration for the locality by using the notion of "places."; 4) to introduce dynamic and asynchronous activities as the base for concurrency constructs; and 5) to support distributed multi-dimensional arrays.

3.3.2 Spawning Multiple Threads

Central to X10 is the concept of a *place*, a collection of data and resident lightweight threads called "activities" [9]. *Places* are intended to map a data-coherent unit within a large scale system and contain a bounded number of *activities* and a bounded amount of storage. Multiple places may be created in X10, allowing cluster-level parallelism. X10 introduces two constructs for creating a new asynchronous task: *async* followed by <*statement*>, and *future* followed by <*expression*>. These spawn a parallel activity to compute the expression value [26].

3.3.3 Synchronization

X10 introduced the notion of asynchronous activities for creating threads locally and remotely. Java's mechanisms for threads, messages, and processes have the limitation of being heavyweight, X10 addresses the lightweight threads for large-scale

Non-Uniform Cluster Computing systems. In order to coordinate asynchronous activities, several constructs were developed for the X10 language such as *async, future, foreach, ateach, finish, clocks,* and *atomic blocks* [8].

X10 also uses atomic statements in order to secure the values of data limited only to the local scope. Since multiple processes need to be coordinated, it is necessary for X10 to use multiple barriers [25]. The approach of X10 is to use *clocks*, wherein activities are associated with clock numbers during execution time in order to effectively track the activity progress. These clocks advance upon completion of their associated activities' computations. Two types of objects are used in X10: 1) *reference objects*, which have mutable fields but cannot be copied between different places of the machine; and 2) *value objects*, which allow free copying but whose fields cannot be updated [25]. There are also no built-in primitive types in X10; they are within standard libraries as value-classes.

3.4 UPC++

UPC++ [11]is a Partitioned Global Address Space (PGAS) extension to the C++ language. This extension aims to provide three main functionalities: 1) an object-oriented model for the well-known C++ language; 2) a collection of parallel programming idioms not included in UPC in order to support complex scientific applications; and 3) an uncomplicated transition to PGAS programming through interoperability with other similar systems. While UPC has proven to have suitable scalability, it also has some problems having complicated fixes. With this, UPC++ has been created from a clean slate, and this allowed the researchers to “freely enhance UPC” while at the same time retaining its superior features [11].

3.4.1 Execution Model

The execution model of UPC is Single Program Multiple Data (SPMD) which implements independent execution units called threads. UPC++ [11] brings the asynchronous feature to this model through distributed-memory systems similar to the C++11 standard asynchronous library for shared-memory systems. Like other PGAS languages, UPC++ works on both shared- and distributed-memory systems.

3.4.2 Construct:

Synchronization is provided in UPC++ [11] in primitives such as barriers, fences and locks to facilitate precise parallel programs. While synchronization in UPC makes use of keywords, UPC++ utilizes functions and macros. However, there is no observable difference in performance between these two approaches because both underlying implementations are the same.

Shared variables and arrays are implemented in UPC++ [11] via templates since they require different implementation strategies. Shared variables require an element type declaration, while shared arrays require both element type and block size, which may be initialized dynamically during runtime. Any thread may read or write a shared variable or array directly.

Remote function invocation is a feature that is present in UPC++ but not in UPC [11]. The user can use this feature with this syntax:

future<T> f = async(place)(function, args...)

It allows the user to start an asynchronous remote function with a single thread ID which is *place* or a group of threads. An asynchronous call can return a *future* object that may be used to retrieve the return value of the remote function.

3.5 Emerald – Distributed Object Based Language

Emerald is an object-based programming language with the goal to simplify the construction of distributed applications[18]. It was designed and implemented by the Department of Computer Science at the University of Washington in the early and mid-1980s. Despite Emerald being one of the pioneer languages to support distribution explicitly, it reflected simplicity in its execution on the object structure, language design, compiler implementation, and runtime support.

3.5.1 Flat Objects

Emerald was the first language to propose and implement the concept of object mobility in a networked environment [27]. The objects within the program were allowed to move from node to node according to the programming language commands. Location-independent addressing permitted mobility; the addresses of objects and targets were semantically irrelevant to its other counterparts [28].

The concept of Emerald revolves around its objects. All entities present in the system – from the small ones such as integers to the large ones such as compilers – are treated as objects. These objects exhibit identical semantics, and can be manipulated only through invocation.

Emerald objects have four basic components [28]: 1) *name* serves as the identifier for the object within the network; 2) representation contains the data stored in the object; 3) *set of operations* state the allowed functions and methods of the object; and 4) *optional process* allows the object to execute independently and operate with an active existence.

3.5.2 Remote Method Invocation

A significant characteristic of Emerald objects is that they are immutable [28]. This simplifies the sharing as they can be freely copied. Concurrency is supported both between objects and within an object. These objects may execute concurrently within a network; within an object, several invocations may be progressing simultaneously and in parallel with the object's internal process.

Variables shared by different operations are synchronized through a monitor. When an object's process needs access to a shared state, it can invoke monitored operations. Furthermore, an object has an optional *initially* section that executes upon the creation of the object to produce its initial state.

The authors in [28] shared that Since Emerald uses the single object definition mechanism, its compiler chooses among several implementation styles during compile time. An appropriate implementation style is chosen for the object to use. There are three different implementation styles: *global objects, local objects,* and *direct objects*.

Global objects have permission to be moved within the network, and allow invocation by other objects not known during compile time. Global objects are heap allocated by the Emerald kernel and are referenced indirectly. This may require invocations to use a remote-procedure-call. *Local Objects* are confined to another object and never move independently of their enclosing object. These objects may be invoked by local calls or by an inline code and are heap allocated by a compiled code. *Direct Objects* are similar to local objects save for their direct data area allocation in their enclosing object [28]. This implementation style is often used for built-in types, records, and simple objects whose structure may be worked out during compile time.

All objects are manipulated through invocation, and they are created to be location-independent. Emerald's runtime system is responsible for the location and transfer of control to the target object. Remote invocation accomplishes the same end result as remote-procedure-calls. For better remote referencing and mobility, the references to an object must not rely on its location.

3.6 Java

Java is an object-oriented language that has become renowned for its essential features of portable internet communication and built-in synchronization methods [29]. High-performance parallel applications have also become more utilized in Java applications.

Java's programming model uses clean and type-safe methods that make it a primary choice for writing large-scale programs that contain concurrency and parallelism [30]. The multithreading concept is also present in Java; it allows execution-shared memory machines. For distributed memory machines, however, Java makes use of Remote Method Invocation, an object-oriented version of the Remote Procedure Call (RPC).

The Remote Method Invocation standard presents some advantages in both parallel and distributed programming [30]. It is also integrated comprehensively with the object model of Java.

Existing Java implementations of sequential code and communication primitives have substandard output in HPC. As such, efforts are being made to address this disadvantage, especially when it comes to large-scale computing. One of the major points for improvement is the communication overhead of the Java RMI implementations.

In [30] also specify that the large communications overhead of the RMI implementations is primarily caused by inefficiencies in the Java Development Kit. Kernel overhead in the JDK is also present and would account for a part of the delay. Another possible cause is Java's RMI model scheme of prioritizing its interoperability and flexibility.

Since RMI was originally designed for Client-Server programming in web-based systems[30], latencies on the level of several milliseconds were acceptable. Unfortunately, these latencies prove to be a huge shortcoming on parallel machines that are tightly coupled.

CHAPTER 4

COMPARATIVE STUDY OF RELEVANT LANGUAGES

This section presents a comparative study of five high performance computing languages: Chapel, X10 UPC++, and C++11 - the latest version of the popular language being extended in this research. It is important to note, however, that these languages have different paradigms. The respective parallel languages are X10, Chapel and UPC++, which are all based on a partitioned global address space, Emerald is based on concurrent objects and C++11 that is designed based on an object-oriented and procedural paradigm.

4.1 X10

4.1.1 Parallel Construct

Data parallelism

The X10 [8] supports multidimensional arrays, which are associated with a set of region. Arrays are distributed into different regions, which can be done using a built-in distribution. In addition, iteration in X10 [8, 9] utilizes three types of *for-loops*. These loops are: the *"For"* loop to provide a sequential iteration in the same activity; *"Foreach"* loop which provides parallel iteration in the same place; and the *"ateach"* loop that provides parallel iteration in different places.

Task Parallelism

Activities in X10 are running in a place and it can process data that resides in the same place. However, a single activity like accessing the data in another place may spawn asynchronous activity that can affect processing the statement in that place [4]. Using this construct does spawn a new activity:

sync(<place>) <statement>

Also, activity can be spawned locally in the same place. This ability gives X10 a high level of multi-threading [9]. However when activity *A* spawns an asynchronous activity *B*, this then returns a value to activity *A* and it known as *futures* [4].

Example:

f = future (p) expression

The spawning activity computes the value of an expression in that place. However, any other activity that wants to use the value of that same expression will blocked until the spawning activity B is complete and the value of the expression is computed. This will help prevent occurrence of a deadlock situation.

4.1.2 Synchronization in threads

There is different synchronization management in X10. One way to execute the synchronization feature is using a mutual access to shared memory. Synchronization in X10 is presented in these various ways:

- By using the *[finish stmt]* at the end of this clause, the current task wait until all the spawning activity during this execution of statement *stmt* terminate[4].

- By using a concept *clock*, which performs hierarchy of activities to execute. It is considered as robust barrier in X10 [8, 9] as it provides a barrier mechanism and does so in multiple activities. Any set of activities can be registered in a *clock*. A set of activities can be registered with a clock *k* in any phase during the execution of a program. The set of tasks scheduled in clock have to be executed as the clock "runs" or moves to another phase. The clock moves one step after all the registered activities perform "next" operation and successfully terminate [9].

4.1.3 Atomicity

X10 [4, 9] support two (2) atomic structures namely: *atomic block conditional* and *unconditional atomic*.

Unconditional atomic is presented by the construct *atomic <statement>* This atomic block performs one activity while the other concurrent activities are frozen.) In X10, the user can decide about which statements he wants to be executed atomically then the compiler does the lock execution and management. All statements access data in the shared memory have to be within the atomic block.

On the other hand, conditional atomic which is done by the following construct:
When (<condition>) <statement>
When the condition becomes true, then the activity execute the statement <*statement*> atomically.

4.1.4 Object Oriented Features

X10, a statically typed object-oriented language, was driven by the desire to produce a high-end, high-performance, high-productivity computing language. It extends a sequential core language using features called *places*, *activities*, *clocks*, *arrays*, and *struct types*[10]. The language X10 has two types of objects called *reference-objects* and *value-objects*. A *reference-object* allows updating of its fields but does not allow copying between different places of the machine. On the other hand, a *value-object* allows copying on a different machine location but does not have any updatable fields [25].

Like Java and C++, X10 is also a container-based type of OO language, which makes use of *classes, structs*, and *interfaces*. Inheritance is also present in X10. However, X10 supports single inheritance: a class could only have one parent class [4]. Functions in X10 are considered as first-class data, meaning they can be used as values, stored within lists, or passed between activities [10]. Methods can also be inherited and overridden in the subclass. Private, public, protected all these qualifiers are used in X10 to control the visibility of a method at the time of declaration.

4.2 Chapel

4.2.1 Parallel Construct

Data parallelism

Instead of using the single threaded model (SPMD) model, Chapel use multithreaded execution model: each process contains multiple threads and allows the user to determine which task need to execute by threads in parallel. Chapel [4] adopts a global view model which means that the program starts with one thread then based on the construct which written by the programmer a new threads will be spawned. So, the data distribution in chapel is user defined. In chapel, users decide what they want to do with partitions so; it gives the control for the user. The logic behind that is different high level of parallel problems need different architecture to deal with it. When we break up this problem to process in parallel, it may good for certain architecture management and not good for other. Chapel also introduces the term *local*, which refers to the machines that would perform the parallel computations [7]. Chapel also provides a built in array of locals that will execute the program and the programmer can rearrange this array to specify in which part this program will be executed. Another data parallelism features in chapel is *domain* [4, 7]; the set of indices in array to introduce the array-distribution. Moreover, for loops which support the concurrently execution in chapel are two types: *forall* and *coforall*. The difference between them is that *coforall* use one thread in each iteration .for example, if we have *coforall* loop has ten iteration then ten threads will run in parallel. In contrast, *Forall* loop use any number of threads to execute one iteration. Thus, *Forall* loop is the best way to distribute computations across threads because in parallel computing a large number of iterations are needed to perform a task.

Task Parallelism

Chapel presents *cobegin* and *begin* constructs to adopt task parallelism [4]. *Begin* is one way to start new task to execute a statement using this form:

```
Original task.
 Begin
{
        new task start
}
```

The new task performs the statement while the original task continues its execution. So, both tasks will work concurrently. *Cobegin* is the other way to use task parallelism in chapel. By using the same form in begin statement.

,

```
Original task.
Cobegin
{
        Multiple task start on different statement
}
```

The difference in using *cobegin* is using it to assign different tasks to each component statement between the brackets. Also, by using *cobegin* the original task will wait until all of the children-tasks are done. We could do the same thing with *begin* by adding the prefix to the statement with a *sync variable*. Like *coforall* statement, *cobegin* loop assigns a distinct task for each iteration and the main thread will have to wait until all other iterations are done.

4.2.2 Synchronization in threads

There are different management of synchronization in Chapel. *Coforall* and *cobegin* have implicitly synchronization [26]. Chapel also has a keyword: *sync*. Any statement with this prefix should have all tasks completed during the execution before it can continue [26].

Synchronization Variables [4] are adopted in Chapel. The variables' store value and its state are either available, *full* and *empty*. Reading or writing over this type of variable is done as follows: the write operation is blocked until state is emptied and then after writing when it has become full. The read operation is blocked until state is full and then after reading it will be emptied [4].

4.2.3 Atomicity

Accessing the shared memory resources requires atomicity. Like X10, Chapel also provides *atomic* section, which allows group of statements to execute atomically. However, *atomic block* and *sync variable* support parallel computations in Chapel.

4.2.4 Object Oriented Features

Chapel is a programming language that supports optional object-oriented features, and does not require users to use these features [7]. It provides these aspects in order to increase the productivity in a parallel setting.

Chapel's classes make use of heap-allocated storage and can be classified as either *traditional classes* or *value classes*. *Traditional classes*, which are assigned and passed by reference, can be likened to the semantics of Java classes. *Value classes*, which are assigned and passed by value, are like structures of the sequential programming languages, but with method invocation support [7]. Chapel also supports *record* that are defined by using the keyword "**record**", and result in in-place memory allocation. In addition, records support value semantics. They can be considered similar to C++ *structs*. Garbage collection for unused and unreferenced objects is available in Chapel also [6].

4.3 Comparison between X10 and Chapel

Table 4.1 summarizes the comparison between X10 and Chapel.

Table 4:1 Summary of the major different between X10 and Chapel

	X10	**Chapel**
Memory Model	PGAS	PGAS
Base language	Java	-
Create task	future async	Begin cobegin
Synchronization	atomic block conditional atomic clocks	sync atomic block
Atomic section	support	support

Parallel loop	foreach
	for
	ateach
		forall
		coforall

Execution model	multithreaded	Global-view multithreaded

4.4 UPC++

4.4.1 Execution model:

UPC++ [11] adopts a Single Program Multiple Data (SPMD) execution model, where the same program is executed several times with different data.

4.4.2 Parallel Constructs

Allocating memory:

Creating distributed data structures require allocating memory in different memory locations. In UPC++ [11], memory can be allocated in local address space or on remote threads. The construct of allocating memory in the global address space is:

global_ptr<T> allocate<T>(int rank, size_t sz);

*w*here, thread id present by *rank* and *sz* is the number of elements for which to allocate memory, for example, allocate space for 32 integers on thread 4 by using:

global_ptr<int> b= allocate<int>(4, 32);

To deallocate memory from any thread, the *deallocate* function can be used for this purpose [11]. In contrast, Chapel doesn't support pointers and all dynamic allocations are through objects and array.

Also, the program starts with one thread then based on the construct which written by the programmer; new threads will be spawned.

In addition, a *copy* function for large data transfers is a useful feature that UPC++ provides by using this construct:

copy (global_ptr<T> src, global_ptr<T> dst, size_t count); where assume that src and dst buffers are contiguous.

Array:

UPC++ [11]allows the adoption of high-level multidimensional arrays like what X10 has for regular numerical data. The user can also build irregular data structures, distribute data in different positions in the memory by using low-level mechanisms like dynamic remote memory allocation and global pointer.

The array library and domain in UPC++ contains this component: points which the coordinates in N-dimensional space, and the rectangular domain which include a lower bound point, upper bound point, and the stride point. For iteration, UPC++ provides *foreach* macro that allows it to iterate over multidimensional domains in this syntax: *foreach (p, dom)*, the user will specify the variable name and the domain.

The array element should reside on a single thread of the memory location so that iterations occur sequentially on that thread to execute the loop. However, the data can be copied from one array to another with this construct *A.copy(B)*. Copying the array data from B to A has two requirements: 1) the domains of these two arrays are not equal; and 2) they do not link to the same thread [11].

4.4.3 Synchronization

Synchronization is a feature provided in parallel programming languages like UPC++, X10 and Chapel in different ways. In addition, it is one way of solving any critical-section problem. Processes have critical sections – code segments that update or write information in the same shared memory. When a process executes in this critical section, any other process must wait until the process already executing the critical section finishes the execution of critical section to guarantee deadlock free computations. UPC++ provides synchronization primitives, such as *barriers*, *fences*, and *locks* to ensure all preceding storage accesses are completed before continuing. Each one has a different construct to use, and they are the *barrier () and fence( )*.

The construct *barrier()*, as the term implies, blocks until all other threads, and the construct *fence()* is used to ensure that all preceding storage accesses are completed before continuing.

4.4.4 Shared Object

For using shared objects in the global address space, UPC++ [11] has two different shared object types: *shared variable* and *shared array*. Shared variables are defined with the *shared* keyword and allowed for global scope variables. Generally the shared scalars; the store location of the variable; is a single memory location owned by thread 0 but can be accessed by all threads because it is global. So, any thread can read or write a shared variable directly.

About declaring the shared scalar variables, UPC++ uses the *shared_var* template as following: *shared_var<Type> s*.

In contrast, a shared array is an array distributed across all threads. Shared array can is declared in UPC++ using the *shared_array* template in this form: *shared_array<T, BS> sa(Size)*. *T* denotes the element type, and *BS* is the block size for distributing the array. In order to reference shared objects in the global address space, UPC++ [11]provides *global_ptr* type which encapsulates the thread ID and the local address of the shared object referenced by the pointer. The template of global pointer has the following form:

global_ptr<type> sp;

4.4.5 Remote Function Invocation

UPC++ [11] provides a remote function invocation to spawn asynchronous activity in another place. The remote function invocation is similar to X10 "future" construct. It specifies the function along with the arguments and the place where the execution will take place. The template for the construct is as follows:

future<T> f = async(place)(function, args...); where place is a single thread ID or a group of threads.;and *async* to start asynchronous remote *function.*

4.5 Comparison between UPC++ and other PGAS Languages

Table 4.2 shows the comparison between UPC++ ,X10 and Chapel. UPC++, X10 and Chapel have been developed using PGAS model. However, the base language is different for all three languages: UPC++ is built on top of C++ and extends C++; X10 is built on top of Java. The synchronization constructs are similar to introduce mutual exclusion and sequentially to avoid deadlocks. However, clock construct in X10 provides lock-step execution of statements. Parallel loops are used in all three languages to handle distributed arrays. UPC++ uses SPMD model, while X10 and Chapel use multithreaded execution.

Table 4:2 summary of the different between UPC++, X10 and Chapel

	UPC++	**X10**	**Chapel**
Memory Model	PGAS	PGAS	PGAS
Base language	C++	Java	-
Create task	async	future async	begin cobegin

Synchronization	barrier()
	fence()
		atomic block
		conditional atomic
		clocks
			sync
			atomic block

Parallel loop	foreach	foreach
		for
		ateach
			forall
			coforall

Execution model	SPMD	multithreaded	multithreaded

4.6 Emerald

4.6.1 Distributed Constructs

There are multiple approaches to exploit the concurrency in programming languages such as using task parallelism in Chapel and X10 by spawning multiple processes and threads work on different data or distributed data structure; as an object in Emerald; across processors to exploit data parallel computation on distributed data structure. Also in distributed systems, objects can be moved in remote processors to exploit concurrent execution, for example, the object-based programming language –

Emerald [18, 27, 28]. There are two (2) types of concurrency that Emerald can support: concurrency between objects and concurrency within objects [28].

Emerald has three types of objects [18]:

1. A *global* object that can move to other node and could be invoked by any other object regardless of location.
2. A *local* object that could not be referenced from a remote node so it can't move directly instead it can be embedded within another object.
3. A *direct* object which we used to implement objects of "primitive" types such as Boolean, Character, Integer.

4.6.2 Parameter passing and remote communication

All objects in Emerald can be manipulated through invocation [28] and all invocations are location independent. However, the choice of parameter passing is an important issue of the distributed system object based. Emerald language [18] uses the *call-by-object-reference*. Where: a reference to the argument object is passed, and there is a method for parameter passing for local or remote invocation. Because of the mobility object in Emerald, it may move the argument object to the site of the invocation instead of doing remote references. So, Emerald supports call-by-move parameter [18] and at the time of the call the object is relocated to the destination of remote invocation. There are some criteria [18] to test if this way of moving object and avoiding remote references is helpful or not based on an argument object size, other current or future invocations for argument, the number of invocations of the remote object to the argument, and the costs of mobility and invocation either local or remote.

4.6.3 Invoking Objects within Threads

The main goal in Emerald, which is an object-based language, is to support object mobility. Thus, an object in Emerald can freely move within the distributed system. All entities in Emerald are objects that include either small entity such as integers, strings, and arrays or large entities like compiler and directories [28].

One of mobility benefit that is provided by Emerald is the invocation performance by moving the parameter objects to the remote node at the invocation time. So, the object may be invoked remotely and moved from node to another, and it is run time responsibility to locate and transfer the control to target object. In Emerald [28], objects are active when they contain a process while objects without process are called passive data structure.

Objects with processes can make an invocation on other object, which can invoke other object also, and so on. As a result, a thread of control that forms in one object may extend to other object. In other words, a thread of control can spawn multiple objects that may be working on separate thread independently. However, multiple threads of control may be active concurrently within a single object.

A *monitor* construct is provided the synchronization in Emerald to synchronize multiple operation invocations to the same object. Emerald allows the programmer to control the locations of objects, and these locations can be changed – object migration.

These some construct that Emerald present to control the locations of objects:

locate X: To locate an object ; the node where the object is.

move X to Y: To move an object to another node.

fix X at Y: The object X is moved where Y is and stays there.

4.6.4 Flat objects vs. Inheritance

Emerald doesn't support any hierarchal structure in the object. In Emerald, objects are flat, and it is a clean object-oriented language with fully integrated distribution facilities. The objects have their own codes and are not members of any class.

Emerald also exploits the notion of "conformity," which means inclusion [4]. Since the *conformity* supports the notion of inclusion, it doesn't have code sharing. The importance of this feature in Emerald is the location of object can be maintained; therefor the ancestor classes are not needed .

4.7 Comparison between Emerald and PGAS Languages

Table 4.3 summarizes the comparison between Emerald and other three major PGAS based languages. The major difference between Emerald and other current PGAS based languages is the: 1) underlying general distributed programming model used in Emerald; 2) object mobility ad migration in Emerald; 3) remote object invocation and passing of objects as parameters; 4) use of flat objects in Emerald; and 5) the use of high level mutual exclusion construct "monitor" in Emerald . In addition, object-invocation can spawn processes in Emerald. Emerald is more about high-level distribution of objects that is missing in the current high performance computing languages like UPC++, X10 and Chapel. While retaining the previous C++-like object-oriented programming, current PGAS based languages only exploit multithread level parallelism augmented by distribution of arrays, they do not supports high level concepts developed in Emerald and Java about object-migration, object mobility and remote invocation of objects.

Table 4:3 summary of the comparison between Emerald and PGAS languages

	Emerald	**UPC++**	**X10**	**Chapel**
Memory Model	concurrent objects	PGAS	PGAS	PGAS
Base language	-	C++	Java	-
Create task	object invocation	async	future async	begin cobegin
Synchronization	monitor	barrier() fence()	atomic block conditional atomic clocks	sync atomic block
Execution model	Object mobility	SPMD	multithreaded	multithreaded

4.8 C++11

In general, there are multiple requirements for programming language to provide the concurrent programming such as thread-creation and thread-synchronization. New primitives introduced in C++11 as parallel language can provide parallelism in different ways using: 1) multiple thread creation and spawning constructs; 2) the use of mutexes and atomic reference construct to provide mutual exclusion of code segments in critical sections of threads using shared resources.

4.8.1 Parallel Programming Constructs

Thread creation

In C++11[31], a thread library is introduced for manipulating and launching thread. The construct *std::thread* is used by defining an instance of *std::thread* to create a new thread that executes the method between the brackets.

Example for crating s thread in C++11 [31]:

```
# include<thread>
#include <iostream>
void threadMethod () {
        std:: cout << "This is a thread" ;
}
int main () {
        std::thread thrd(threadMethod);
        std:: cout<<"This is the main";
        thrd.join();
}
```

Join function is used to let the current thread wait until the thread that is executing the function terminates successfully.

4.8.2 Programming Issues

Critical section

One of the synchronization issues is when multiple threads attempt to execute a critical section simultaneously. However, mutual execution can be utilized to avoid this problem. So, a thread should a acquire locks for all the data that access in critical section. The most common solutions in C++11 [31]are the use of *mutex* and *atomic reference*.

- *Mutex:*

The use of *mutex* allows for thread to access the shared data and running while other threads trying to access the shared data wait.

How mutex is used in C++11?

An instance of std::mutex is invoked to create a lock followed by the lock( ) function. The invocation of *lock()* function sets the lock. After executing the critical section, the function *unlock( )* is invoked that releases the lock. The unlocking of the lock unblocks the waiting threads.

Example for explain using the mutex in C++11 [31]:

```
struct Counter {
        std:: mutex mutex;
        Int count;
        Counter () : count (0) { }
        void increment () {
                mutex.lock ();
                ++ count;
                mutex .unlock() ;
        }
};
```

- *Atomic*:

Another way to protect the shared data is by using the construct atomic. That implies no other thread can update the result until current thread is finished. C++11 provide this feature by using a template class called std::atomic. Using this construct

Std :: atomic <type> <object>

Example for using atomic in C++11 [31]:

```
# include atomic
struct AtomicCounter {
        std:: atomic <int> count;
        void increment () {
                ++ count;
        }
        void decrement () {
                -- count;
        }
        int get ( ){
                return count.load();
        }
};
```

The distinction between Atomic and Mutex:

Atomic technique is faster when it is used with small data types such as *int*, *long* and *float*. It is more effective than the *mutex* technique, which is more suitable with big data type [11].

4.9 Comparison of C++11 and other High Performance Languages

C++11 has the capability of spawn multiple threads and provide mutual exclusion as discussed in subsection 4.8. However, it has following limitations that make it currently unsuitable for high performance computing:

- It does not have constructs to support parallel execution of multiple elements of a distributed thread for high performance computing that is supported by UPC++, Chapel and X10.
- It does not support distributed data structures such as distributed arrays and distributed vectors. It also does not support flat objects. Many times flat objects provide the abstraction capability, and inheritance is not needed.

- It does not support remote invocation of methods on different logical execution nodes;
- It does not have the capability to group multiple logical executing nodes into a region;
- There is no notion of logical nodes that can be mapped to different physical nodes dynamically by the operating system to balance the load and to provide recovery in case of the failure of a physical node.
- It does not have capability of collaboration between multiple copies of the same object working on the different input data to generate the output.
- It does not support capability of creating a copy of the object or object mobility among various logical processing nodes that can be mapped transparently by the operating system on the physical processors.
- It does not support dynamic growth (and shrinkage) of the region where the objects can map as the problem grows or shrinks.

While UPC++ has borrowed many concepts like place, distributed arrays, processing multiple data elements of the distributed array concurrently, it does not support object mobility, object-migration, distributed objects, object cloning, remote method invocation and the notion of flat objects present in Emerald.

CHAPTER 5

DOPC++: A new Model for Distributed Object

The key point behind the DOPC++ (Distributed Object based PGAS for C++) proposed language is to integrate the notion of objects to exploit the distributed computing in the PGAS memory model, which has global address space and part of the memory local to each processor. DOPC++ introduces a high level abstraction, and hides the low level instructions to increase the usability.

Current PGAS based languages do not support: 1) distribution of objects; 2) migration of objects; 3) and cloning of the objects. The use of PGAS is limited only to spawning threads, distributing arrays, and global view of arrays. Although they have the notion of objects, the full capability of PGAS is not integrated with the notion of objects. The communications with remote processes is low level using spawning of threads. These languages do not use the object message passing concepts for communication between remote objects. Hence, they cannot be called truly high-performance object-oriented languages.

DOPC++ supports distributed creation and dynamic migration of objects; communication between remote objects; and cloning of distributed objects, notion of region associated with a class, notion of subregions associated with methods and array in a class, and the remote invocation of methods in a place for load shedding. Object-migration allows the objects to move during the execution which facilitates load balancing and performance-improvement without the loss of functionality.

The migration of objects can be one place-to-one place, many places–to-one place, one place-to-many places, or many places-to-many places. System level utilities inserted by the compiler takes care of the mapping at run time. When an object migrates from one place to another place <place> in the region <*region*> then the runtime system utilities will creates an alias <*region*>**.**<*place*>**.**<*object-name*> to point to the same object. The migration of objects from a region requires a broadcast of the code part of the object to the destination-region from one of the places in the source region. The data areas are migrated based upon the type of mapping.

Communication between objects is done through a *remote invocation*. The parameters which are passed during the invocation can be objects themselves. Parameter passing uses *call by object reference* or *call by object move* as in Emerald [18]. In *call by object-reference*, the identifier of the object is passed that allows accessing the object remotely. *Call by move* involves migration of an object to the remote place. In addition, the interface of any method to execute remotely is done by accessing the object.

C++ is extended in DOPC++ by supporting: 1) *C++ class; 2) distributed class*; 3) *flat class* as used in Emerald [28]; and 4) *local class.* A "distributed class" has distributed data elements and/or distributed methods. The scope of a distributed class is within a region. Region is a set of places, and a place is a logical node where an activity takes place. The logical node is automatically mapped to the physical processors first statically at compile time, or later dynamically based upon the load balancing and object mapping at runtime by the operating system. Multiple activities may be spawned in a place to solve a task. An activity may include a combination of multiple threads and synchronization based upon monitors. However, these spawning of threads are not explicit; operating system has flexibility to spawn multiple threads, or group the subtasks using a single thread depending upon the available threads and load balancing.

The region that is associated with a *distributed class* can be "static" or "dynamic". A *static region* is a user-defined logical region and is fixed at compile time. Declaring a static region allows every place in that region to get a copy of the class-template and each place within this region will create an object in response to object-creation instruction. Unlike Chapel, DOPC++ also supports *dynamic regions*. A *dynamic region* is a set of places created at runtime, and is can be altered at runtime by the operating system for load balancing.

When declaring a *dynamic distributed class*, a user gives an initial region to start with then the region can expand or shrink. However, distribution of objects is done depending on the load balancing. Within this dynamic region every place get a copy of the class-template automatically. Similarly, the object-creation checks the number of places at runtime, and invokes objects in every place of the dynamic region concurrently. The use of dynamic-regions allows migration of the objects, methods and data elements potentially to any place for load balancing. DOPC++ supports remote invocation of the object and the methods. Multiple Objects could be invoked concurrently within a region, and each can work on an array of data elements independently.

Like Emerald, a process may be embedded in a flat object, and the process starts when the object is invoked. However, An object can be active, and active objects can share the information using a *shared blackboard*. A *blackboard* can be in the global shared address space or it could be distributed among the regions; each time information is written in a blackboard, the related processes are suspended in the corresponding regions using a monitor to achieve synchronization until the process writing on the blackboard finishes using the blackboard. Monitor insures mutual-execution in the shared blackboard to control the synchronization. All methods deal with (access) shared value then executed inside a monitor, to avoid the race-conditions.

5.1 An Overview of New Constructs in DOPC++

This subsection describes an abstract description of the scope-rule extensions, newly added constructs and built-in statements for operation on objects for the language DOPC++ that are supported in PGAS model. There are constructs for the creation of different types of classes, creation of distributed objects, cloning of objects, migration of objects, remote invocation of the objects, communication between remote objects and the new scope rules for extended definition of place and region described in subsection 5.2.

DOPC++ extends C++, and uses C++ syntax and scope rules. In addition, it has borrowed and extended the notion of places and regions from PGAS languages X10 and Chapel. It extends their definition as follows:

- *Place* is a logical entity that maps on a processor, and holds one or more tasks. A place can share the information with other places using PGAS shared address space, and requires constructs to access address space in remote places within the same region. C++ classes are special classes without any region, and can be accessed remotely from any place.

- *Region*: A set of places logically represents the processors, which execute a program, and can be distributed. The regions allows mapping of a *distributed-array*, *distributed-vectors, distributed-objects* and *shared blackboards*. A region could be both static and dynamic. A *static region* is user-defined and the mapping of region to the physical processors is fixed at compile time. A *dynamic region* grows and shrinks based upon the need of the executing task and the load-balance of the physical processors. Objects - method and data elements - migrate between processors dynamically in a dynamic region to balance the process-load. Dynamic region has important properties: dynamic mapping of a place to a physical processor ; a *problem space*: problem space is a fixed set of places where a dynamic region can grow. The dynamic region cannot grow beyond the given problem space. The rationale for the problem space is to limit the spread of dynamic space for very large problems that may affect the execution efficiency of other tasks.

The region provided an added scope rule for the visibility of objects and classes. A class declared within a region is visible only within that region including all the places within that region. This also limits migration of objects to within the region where a class has been declared. However, for dynamic region, the migration pattern of objects also changes dynamically. For the dynamic regions, the operating system keeps the mapping table of logical places and regions to the physical processors.

5.2 Grammar for Additional Features in DOPC++

This subsection describes a grammar for various constructs and built-in statements in DOPC++ that are not part of regular C++11 syntax. As described in subsection 5.1, the new DOPC++ constructs are: 1) for the declaration of place and region, distributed arrays, shared global space and shared blackboard; 2) declaration of different types of classes; 3) invocation, cloning, migration of distributed objects; 4) remote communication between objects; 5) synchronization constructs using monitors.

```
<typed-Class> ::= static <class> | dynamic <class>
<class> ::=  <distributed-class> | <flat-Class> | <local-Class> |
                    <C++ Class>
<distributed-class> ::= distributed <class-declaration>
                                [at region <identifier>]
<flat-class> ::= flat <class-declaration> [at region <identifier>]
<local-class>  ::= local <class-declaration>  [at place <Identifier>]
<C++-Class> ::=  <class-declaration>
<class-declaration> ::= class <identifier>
                              '{'  [{<visibility>] {<class-members>';'}*}*  '}' |
<class-members> ::= <class-variables> | <data-abstractions-decl> |
                              <method-declaration> | <constructor-declaration>
<visibility> ::= public | private | protected
<class-variables>  ::= <data-type> {<identifier>','}*<identifier>
<data-abstractions-declaration>  ::= <distributed-data-abstractions>  |
                                                <C++ data-abstractions>
<distributed_data_abstractions>::= <syncronization_type><compile_type>
                                                <location_type> ( [<distributed_arrays> | <
                                                <distributed_vector]>)
<synchronization-type> ::=  [synchronized]
<compile-type> ::= static | dynamic
<location-type> ::= [ local at place <identifier> | global at region <identifier> |
                              at <identifier> ]
```

```
<distributed-arrays>  := distributedArray <identifier>'['<dimension-list>']'
<dimension-list>::= {<integer>[:<integer>]','}*<integer>
<distributed-vector> ::= distributedVector <identifier> '['<integer>']'
<methods>  ::= <distributed-methods> | <C++ methods>
<distributed-methods> ::= distributed <method-declaration> in <identifier> |
                                remote    <method-declaration>    in    [place]|[region]
                                <identifier>
<method-declaration> ::= <data-type><identifier>'('[<parameters>] ')'
<parameters> := '(' {<typed-parameters>}{";"<typed-parameters>}*  ')'
<typed-parameters> ::= < data-type>< identifier >{','< identifier>}*

<distributed-operations> ::=  clone (<identifier> in <identifier>) |
                                 migrate [(<identifier>] to <identifier> |
                                 <method-invocation>
< method-invocation > ::= < location-identifier > '.'
                              <object-name> '.' <method-name> '['<arguments> ']'
<monitor> := monitor  "{" [<identifier>] {<method>';'}*  "}"

<blackboard> :: <distributed-blackboard> | <local-blackboard>
<distributed-blackboard> ::= 'distributed blackboard' <identifier>
                                 'at region' <identifier>
<local-blackboard>  ::= local blackboard <identifier> [at place <identifier>]

<blackboard-operations>  ::= <bb-sync> (<bb-read> | <bb-write> | <bb_locate>)

<bb-sync> ::=  [(sync | async | sync wait <integer>) ]

<bb-read> ::=  <identifier> '::=' 'bb_read('<identifier>) [at <identifier>]
<bb-write>  ::= 'bb_write('<identifier>, <identifier>')' [at <identifier>]
<bb-locate> ::=  <identifier> ::= bb_locate "(" <identifier> ")"

<loop-statement> ::=<forall_statement>|<foreach-statement>| <for-loop>
<foreach-statement> := foreach [place] <identifier> in <identifier>
<forall-statement> := forall "("<identifier> "=" <identifire> ":" <identifier>
                                 [; <integer>] ")"
<place-decleration> := place <identifier> "=" <integer>
<region-decleration> := region <identifier> "="  (<integer> | <integer-range>)
                                                    {","(<integer> | <integer-range>)}*
<built-in-method> := size | length | indexset
```

Figure 5:1 Grammar for additional features in DOPC++

In DOPC++, a class can be **static** or **dynamic** then define the <*class*>. A <*class*> itself is defined as <*distributed_class*> *,* <*flat_class*> , <*local_class*> or <*C++ class*>. A <*distributed_class*> includes the reserved word **distributed** following by the <*class_definition*> then detect the region. A <*flat_Class*> is defined by the reserved word **flat** then the <*class_definition*> followed by the region. A <*Local_Class*> can be defined as following: the keyword **local** then the <*Class_definition*> come next. A <*class-declaration*> includes the reserved word **class** following by the <*identifier*> then the <*visibility*> of the <*class_members*>. A <*class_members*> could be class variables, declaration of data abstractions, method declaration in the class or constructor declaration.

The visibility <*visibility*> of class member can be *public, private* or *protected***.** A <*class-variable*> is defined as the type information <type> followed by the name of the class-variable <*identifier*>. The data abstraction <*data_abstractions_decl*> can be either <*distributed_data_abstractions*> or regular <*C++ data_abstractions*>. The <*distributed_data_abstractions*> is described by specifying the <*syncronization_type*>, followed by <*compile_type*> followed by <*location_type*> for <*distributed_arrays*> or <*distributed_vectors*>.

A <*synchronization-type*> for the distributed data can be *synchronized* or *asynchronous*. By default, the distributed data is treated as asynchronous. Synchronized type means that only one element in the distributed data can be processed at a time to avoid race-condition; other statements (including statements in concurrently executing threads) using the distributed data have to wait. A data type being *asynchronous* means that multiple operations on the data elements within a distributed data abstraction can be performed concurrently by multiple threads. The granularity of the synchronization is at the data abstraction level: array level in distributed arrays, vector level in distributed vectors, etc.. Each synchronized distributed data is associated with a global lock-variable. When a method writes into synchronized distributed data, then the global lock is set in 'locked' state before performing the write operation. However, this restriction introduces sequentiality. In the presence of SIMD operations, such limitations can cost serious efficiency overhead. To avoid this, the distributed array by default is asynchronous, and it is the programmers' responsibility to ascertain that multiple threads working on the same data use mutually exclusive based upon the use of monitors. The SIMD parallel operations on synchronized data objects such as *forall* and *foreach* are allowed to work in data parallel manner with an understanding that programmer intends to allow data-parallel operation. All the methods declared within monitors are mutually exclusive, and work one at a time. By default, distributed data is asynchronous to provide maximum flexibility of multiple threads working on different places that may be mapped on different processors.

<*Compile-type*> is defined as **static** or **dynamic**. A <*location-type*> describes the way data-abstraction is distributed. A location-type distributed, local or global. The data can be local within a place or within a region. A distributed data-abstraction is distributed within a region with compiler and operating system deciding the granularity-size based upon: 1) available memory; 2) processing speed; 3) processor load; and 4) processor configuration table. The grammar rules defines <*location-type*> has a multi-

definition: 1) reserved word “**local**” followed by the reserved word “**at place**” followed by the place name <*identifier*>; 2) distributed within a region with reserved word “at region” followed by region <*identifier*> or **global**.

A distributed array can have multiple dimensions. The elements are placed in a region with a granularity size being placed in a place. The granularity size for each dimension can be different. A distributed array <*distributed-array*> is defined as the reserved word “**distributedArray”** following by a name of array <*identifier*> followed by <*dimension-list*> within the reserved words “**[“** and **“]”**. The <*dimension-list*> is a sequence of *dimension:granularity* <*integer*>[‘**:’<***integer*>] separated by the reserved word “**,**”. A <*distributed-vector*> includes reserved word “**distributedVector”** followed by the vector-name <*identifier*> followed by vector-size and granularity using the struct “[“ <*integer* >[**:**<*integer*>] “]” followed by the reserved words “**in place**” or “**in region**”

A method in DOPC++ is described as <*distributed-methods*> or a regular <*C++ methods*>. A <*distributed-method*> is tagged by a reserved word “**distributed”** followed by <*method-declaration*> followed by: 1) reserved word “**in region”** following by the region-name <*identifier*>**.** A distributed method <*distributed-methods*> can be declared in remote place by using the reserved word “**remote**” followed <*method-declaration*> followed by the reserved word **“at place”** followed by the place name <*identifier*>.

A <*method-declaration*> is defined as type information of the method <data-*type*> followed by the method-name <*identifier*> followed by the list of parameters

<*parameters*>. The parameter declaration <*parameters*> is a sequence of parameters <typed-*parameters*> separated by the reserved-word "**;**". The declaration <*typed-parameters*> is a type declaration <data-*type*> followed by the multiple parameters <*identifier*> separated by a delimiter "**,**".

A blackboard is a synchronized shared address space that is shared between multiple threads. A blackboard can be *local*: shared between local threads in a place; *distributed*: shared in a region shared between the threads in the places within a static or dynamic region; or *global*: shared between all the threads among multiple regions. If the threads are cooperating across the places within a region then the blackboard is *distributed*. *Local blackboards* are used for sharing information between threads within the same place. Each blackboard is associated with a lock in the global address space. When a thread writes on a distributed blackboard, the global lock is fist captured to ensure that no other thread can write on the blackboard. The lock for a local blackboard is kept locally in the same place where the thread activities are taking place. The blackboard is automatically released after a data has been placed in the blackboard. Distributed blackboard <*distributed_blackboard*> is defined as the reserved word "**distributed blackboard**" followed by the blackboard name <identifier> followed by the reserved word "**at region**" followed by the region-name <*identifier*>. The local blackboard is defined as the reserved word "**blackboard**" followed the blackboard-name <*identifier*>. Global blackboard is defined as the reserved word "**global**" followed by the blackboard name <*identifier*>

There are multiple distributed operations DOPC++ has: *clone*, *migrate*, *remote invocation of methods, get_object_location, move, remote_delete, get_remote_value, bb_put, bb_get*. The operation *clone* is defined by by using **clone** reserved word following by the object-identifier <*object-identifier*> followed by the reserved word "**to**" followed by the destination-identifier <*destination-identifier*> that is a region-name. An object migration operation is done by placing the **migrate** reserved word, followed by the object-identifier <*object-identifier*> followed by the reserved word "**to**" followed by the destination-identifier <*destination-identifier*> that can be a place-name or a region-name. A remote method invocation <*method-invocation*> is defined by <*place-identifier*> followed by the reserved symbol "**.**" to concatenate, followed by the object-identifier <object-identifier> followed by the access operator "." Followed by the method-identifier <method-identifier> and list of parameters to pass the arguments. Monitor based synchronization is achieved by placing the reserved word **monitor** followed by the block of statements forming the critical section within the curly brackets.

Loop statement is a multi-definition: <forall-statement> or <*foreach-statement*> or regular <*for-loop*>. A SIMD statement <*forall-statment*> is defined by reserved word **forall,** followed by left parenthesis "**(**", followed by the index-variable <*identifier*>, followed by the lower-bound <*integer*>, separated by a delimiter "**:**" from the upper bound <*integer*>. The foreach SPMD statement <*foreach-statment*> is defined by the reserved word **foreach,** followed by an optional reserved word "**place**", followed by an identifier **<**identifier**>,** followed by another reserved word "**in**" followed by another identifier <identifier> or a set of identifiers within a curly bracket. The second identifier represents a region if the construct contains the optional reserved word “**place**.”

A region is declared by using a reserved-word **region,** followed then <*identifier*> represent the region name. followed by a sequence of place-identifiers <integer> or a subrange of place-identifiers <integer-range> within a curly bracket.

In DOPC++ three built-in methods are used: **Indexset**, **length** and **size**. **Size** computes the number of array to be allocated to each place by knowing the place capacity. **Length** method returns the length of distributed data at each place which map during the run time. **Indexset** compute the set of indices of a distributed array element in a place.

5.3 Major Constructs and Semantics

5.3.1 Class

- *Distributed class construct:*

```
static distributed  class <class-identifier> at <region> {
           public :
                    // class members
          private:
                    // class members
}
```

Figure 5:2 Declaration of a static distributed class

Semantic:

A class is distributed in a static region and a programmer can define the region in advance. A static region is fixed during the compile time. As a result, a copy of the class template is distributed along all places inside this region. Multiple instances of the objects are created in all the places in the region at runtime when an instruction is given to create object instances. The name of the class should be unique within a region, and the name of the object is qualified by the place-identifier to make it unique.

- *Dynamic distributed class construct*:

```
dynamic distributed  class <class-identifier> {
        public :
                // class members
        private:
                // class members
}
```

Figure 5.3 Declaration of a dynamic distributed class

Semantic:

Dynamic distributed class is the same as the static class but the mechanism of the region is different. In dynamic region, A user could give a initial region to start the distributed class. However, the region grows and shrinks but can not go beyond the the problem space. A class template is created in every place of the initial dynamic region at compile-time. However, the class template migrates at run time as the region grows or shrinks based on the load balance. In case of nested classes (inheritance) when move object to another palace the whole hierarchy should move too.

- *Flat Class:*

```
(static | dynamic) flat class <class-identifier> at <region-identifier> {
        public :
                // class members
        private:
                // class members
}
```

Figure 5.4 Declaration of a flat class

Semantics*:*

Flat class is distributed class in static or dynamic region. A flat class allows objects to be easily distributed since there is no hierarchy. In DOPC++, by default, the class is considered as a hierarchal class unless the declaration is preceded by the reserved word "**flat**". The implementation of this type of class uses the flat object structure as in Emerald [18]. Flat-objects gives more flexibility in migrating object within the region.

- *Local Class:*

```
local class <class-identifier> at <place-identifier>{
        public :
                // class members
        private:
                // class members
        }
```

Figure 5:5 Declaration of a local class

Semantics:

Local class is defined in a specific place. An object is created in the same place. In the local class there is no flexibility of object-migration. In case we want to migrate the class an error message will be given that this class is restricting the boundary. One motivation of using the local class is the privacy issue. Because of the security, this class is important and the information need to be in this place and can not sent to other processors. However, We can use this type of class for assigning a specific operation such as print to a specific processor. Remote method invocation is used to invoke a local method as illustrated in example 6.3.

5.3.2 Distributed arrays:

```
[synchronized] static global distributedArray <identifier>'['<integer>[:<integer>]']'
                         at region <region-identifier>
```

Figure 5.6 Declaration of a distributed array

Semantics:

A distributed Array distributes in all the places of a user-defined region. There are two options: static distributed-array and dynamic distributed-array. In addition a distributed array can be in a sub-region of the basic problem region. A distributed array can also be located locally at a specific place. Each element of a distributed array is controlled by one distributed-object. The elements of distributed-array can be processed in parallel using multiple invocation of a method in the resident copies of the objects in the places of the region. Multiple Invocation distributed data (MIDD) is one feature of the proposed language DOPC++. When a distributed array is defined as global then the elements are shared in global space for the whole places within the region. In this case monitor block is need to control the synchronization. So, different objects can access the data and update these data with out any error. Different dimensions can be used for the distributed array. That means, the user can use distributed region to distribute the array elements.

```
region <regionname>  = [{<(<integer> | <integer-range>)",")*]
                                (<integer> | <integer-range>)
distributedArray <arrayname>[<rows>:<row-grain>][<columns>:<column-grain> ]
                                       at <region-identifier>;
```

Figure 5:7 Declaration of multidimensional distributed array

Each element of a two dimensional array can reside in different places. <row-grain> tells how many rows can be put together in one place; <column-grain> tells how many columns together can be placed in one place. The array is row wise distributed if the <column-grain> is missing; the array is distributed column-wise if <row-grain> is missing.

5.3.3 Distributed Operations:

- *Remote method invocation:*

```
<place-identifier>.<object-identifier>.<method-idenfier>(  );
```

Figure 5.8 Invocation of a remote method

Semantics:

For communication between places, remote method invocation is used. Objects can be passed as arguments. A remote method is invoked using an access operator ".". Whenever a method is located at a specific place and it needs to be accessed to perform some computation for balancing the load or using a specific resource at a specific place, remote method invocation is used.

- *Object-migration:*

```
migrate <object-identifier> to [(<place-identifier> | <region-identifier>)]
```

Figure 5.9 Migration of a method

Semantics:

This construct is used to execute the method of moving object to another place or region. This will give a user more flexibility to use the object as a unit of processing a problem instead of working at thread level. The purpose of object migration is to distribute the workload. When migration is done the data, code-template is broadcast to all the places in the destination region. However, data-area of the object is migrated to balance the data distribution in the places of the destination area.

- *Clone Object*:

```
clone <object-identifier> in [(<place-identifier> | <region-identifier>)]
                          to  [(<place-identifier> | <region-identifier>)]
```

Figure 5.10 Cloning an object

Semantics:

Cloning object is one of the operations that can be used in DOPC++ to duplicate the object in another place or region. However, the object should be clone-able to execute this method. In case of local class for example, the object is fixed in one place and cannot copy it to other place. Provided an object can be cloned, the object is copied from one of the places in the source region a place in the destination region or to a specific place.

5.3.4 Synchronization

```
monitor
{
        // set of operations
}
```

Figure 5:11 Use of a monitor for synchronization

Semantics:

Monitor is a way to impose mutual execution. All methods deal with or access shared value are executed inside this block, to avoid the race of condition. However, a monitor controls synchronization by executing one method at a time. In our example, which is presented, in section 6.3 the monitor is used to control a accessing the shard array to apply the increment and decrement operation. So, with monitor facility all operations become mutually exclusive, and work one at a time to access global data.

5.3.5 Method declaration:

```
distributed <type><method-identifier> in <region-identifier>
remote <type><method-identifier> in (<place-identifier> | <region-identifier>
```

Figure 5.12 A distributed method declaration

Semantics:

In DOPC++ the distributed method can be declared in a region or it can be in a specific place or region remotely. When declaring a method in a region so each object in each place within the region will get a copy. The method can declared and distributed in subregion. A method can be invoked remotely as described in example 6.1.

CHAPTER 6

Programming Illustration

This chapter describes many example program that illustrate various paradigms and constructs described in DOPC++. Specifically, the chapter illustrates object migration, object cloning, synchronization using monitors, remote method invocation along with multiple concurrent object creation and remote method invocation to work on distributed arrays in a region.

6.1 Object Migrations and MIDD

This example illustrates object migration from one region/place R to other region/place S. The object initially resides at one specific place within a region R where it performs some computation. The object includes the distributed array in a region. The object name is unique within a region making it simpler for the programmer to search the object within the region. The purpose of this migration to use an object-method at a place/region that has certain resources the object can use. Declaring a static class in a region allows every place in that region to get a copy of the class. The object-creation automatically creates multiple instances of objects, one in each place, within the region. The migration of an object involves one-to-one, many–to-one, one-to-many, or many-to-many. In case of the overlap of the regions, operating system creates an alias <*region*>**.**<*place*>**.**<*object-name*> to point to the same object. The migration of objects requires broadcasting of code-part to the destination-region from one of the places in the source region. However, data part migrates from the places in the source-regions to places in destination region based upon compiler generated instructions.

Example 6.1

Example 6.1 illustrates the concepts. The program has four functions: main, lookup, store and print. The function creates the distributed objects for distributed static class dictionary, and calls store and lookup methods. The lookup and store are performed in a region R = {*1, 2, 4, 5*}, and the data is printed in a different region S = {*3*}. It performs object migration to print distributed array *word* in a different region. The function *lookup* looks at various places concurrently. The number of threads spawned is operation system responsibility based upon load balancing. The function *store* reads one word at a time, and stores in the words in the distributed array *word*. The function *store* spawns concurrent threads in all four places that is transparent to the programmer, and taken care by the operating system based upon the load. The function *print* prints spawns one thread that prints the dictionary words sequentially using a single printer.

Two regions *R* = {*1, 2, 4, 5*} and S = {*3*} are declared with a global scope within the program. It should not be confused with the "global region." The assumption is that printer is connected to place *3*. A static distributed class *dictionary* is declared within the region R. Four class templates, one for each place within the *region R* are created automatically by compiler generated instructions that is transparent to the programmer. To insert new words, multiple invocations for all objects is done concurrently storing the new words inside the distributed array *word*. Each place gets an instance of the class dictionary that is created concurrently by the code "foreach p in R *dictionary* d."

Migrating the objects to region S will create an object in the place 3, that will print the distributed array word sequentially on the printer.

In the program, the dictionary d is distributed among the places by the compiler using a built-in method *size* that computes the number of words to be allocated to each partition by knowing the relative processing and storage capacity of each place that is stored in a reconfiguration table during mapping of the logical place to the processor.

Another built-in method length gives the *length* of the distributed data at a place during runtime since the number of data-elements change based upon the place to processor mapping.

```
region R = {1, 2, 4, 5}; region S = {3};
static distributed class dictionary at region R;
{         dynamic string distributedArray word[n] at region R;
          distributed void store ()
          {         cout<< "enter new words"
                    foreach place p in R {   // this loop to go over the places within the region
                      m = p.word.size (n);  // get the number of words to be stored at a place
                      for (i=0; i<m; i++) cin>> word[i] ;  // read the words in the dictionary
                    }
          }
          distributed void lookup () in region R
          {         string w;
                    cout<< " what is your word to look up:"; cin>> w;
                    found = false;
                    foreach p in R  {   // enumerate over the places in region
                      foreach i in indexset(p.word)  // search the words in my place
                            if (word [i] == w)  found = true;  // no need for monitor here
                      }
          }

          remote  printData in region S ()
          { foreach p in S {   // this loop to go over the threads within the place5
                m = p.word.size(word.length)
                 for (i = 0; I < m; i++)   // read the words in dictionary one word at a time
                 cout<<  word[i];
                }
          }
}
```

```
int main ()
{         int value;
          foreach p in R    dictionary p.d;  // create object in each place
          cout<<  "1 to store new words, 2 to look up , 3 to print:";  cin>> value;
          switch (value)
          {         case1:  store ();  break;
                    case2: lookup (); break;
                    case3: migrate d to S;  break;
          }
          return 0;
```

Figure 6:1 An example illustration object migration and MIDD

6.2 Object Cloning

This example illustrates Object cloning in specific region. When declaring a dynamic class so the distributed of object will be done depending on the load balance. However, we can conceder the default region is a problem space and the dynamic region cannot grow beyond this default region. Within this dynamic region every place get a copy automatically. Since the dynamic region allows the migration and cloning of object within the dynamic region, cloning is done to create copies of the object across the regions.

Example 6.2

In this example, we define a dynamic distributed class *math* in a dynamic region R with initial value {1, 2, 3, 4}. Two methods *add* and *main* are defined. Multiple objects are created from this class in main method in the dynamic region R. Different class templates, one for each place within the dynamic region, are created automatically. The method add() is used to do *math* operation for every element in the distributed array *num*. The method add() is created in sub region R1 ⊂ dynamic region R. The distributed array is created in the sub region R1. Additional computation on the distributed array is done in the region R2. A static distributed class *test* is created to test the array elements whether they are *even* or odd using the function *isEvenorOdd*(). Cloning clones encapsulated code and data area in all the places in the region R2.

The method *isEvenorOdd( )* works on the distributed cloned array num[n] by filtering even and odd numbers and storing in two distributed arrays: *even* and *odd.*

The distributed array in different places within the region R1 is added concurrently by spawning concurrent threads in different places. The result is stored in individual places in the variable accumulator. There are as many occurrences of the variable accumulator as the number of places in the region. These values of the different variables accumulator from different places are added to the synchronized global variable sum using the construct monito*r.*

In the program, the built-in method *size* is used to allocate the number of words to a partition based upon the reconfiguration table during mapping of the logical place to the processor, and the built-in method *length* gives the number of words in the distributed array *word*. In the method *isEvenorOdd*, the use of monitor is needed as the variables j and k can be updated by multiple concurrent threads, each checking and copying for even or odd numbers from the distributed array *num* to distributed array *even* and distributed array *odd*.

```
dynamic region R =  {1, 2, 3, 4};
region R1 = {1, 2, 3};
synchronized int sum=0;  // A synchronized global variable with final sum-value

dynamic distributed class math in dynamic region R;
int accumulator = 0;
{         static distributedArray num[n] at region R1;

          distributed void add () in R1
          {  foreach place p in R1 {   // this loop to go over the places within the region
                  m = p.num.size(n);  // get the number of words to be stored at a place
                  for (i=0; i < m; i++)       // add the numbers at a place using a thread
                             accumulator = accumulator + num[i];
                    monitor {
                        sum = sum + p.accumulator;  //each place updates global variable sum
                             }
                }
          }
}

int main ()
{
          int value;
          foreach p in R    math p.m;  // create object-instance m in each place
          cout<<  "1 to add, 2 to separate even and odd number in array";
          cin>> value;
          switch (value)
          {         case1:  add (); break;
                    case2:  clone m in  R2;  break;
          }
          return 0;

}
```

```
region R2 = {6, 7, 8};
static distributed class test at region R2
{         synchronized int j, k = 0;
          static distributedArray even [n] at region  R2 = 0;
          static distributedArray odd [n] at region R2 = 0;
          static distributedArray remainder[n] at region R2;
          distributed void isEvenorOdd ()

{         foreach p in R2 {   // this loop to go over the region R2
              m = p.num.size(num.length);
                forall (i = 0:m)  {  // this loop to go over the array indices inside place
                    remainder[i]  = num[i] %2 ;
                       if (remainder[i] == 0)
                             monitor  even [j++] = num[i];
                       else  monitor odd[k++] = num [i];
                    }   //end forall
                }   //end foreach
            }  //end isEvenorOdd
}  //end class

int main ( )
{   foreach p in R2 test p.t ;
    isEvenorOdd( );
return 0:
```

Figure 6:2 A programming example showing object cloning

6.3 Synchronization Using Monitor and Remote Method Invocation

This section illustrates the monitor construct for the synchronization of global data objects using monitor. Global data objects are used in PGAS for sharing the information. Monitor block is used in DOPC++ to control the synchronization. All the methods declared within monitors are mutually exclusive, and work one at a time to access global data. This example illustrates the use of monitor access and write data in a synchronized global array.

Example 6.3

This example illustrates the use of monitor in providing mutual exclusion to the methods in a dynamic distributed class *counter* defined in the dynamic region *R*. The class counter has two mutually exclusive methods – *increment* and *decrement*- operating a shared distributed array *num*. The array *num* is allocated in the global shared partition space. The access to these shared distributed array *num* is done using a monitor that provides exclusion between the methods *increment* and *decrement.* The method *increment* increments each element of the distributed array by 1 in a data parallel manner. The function decrement decrements each element of the distributed array by 1 in a data parallel manner. Synchronization is needed when the mutually exclusive methods increment and decrement attempt to simultaneously write in the data elements.

A remote method printData () is invoked and distributed array *num* is passed to place 5 to print the array element,. A local static class print is created at a specific place 5, which prints a local array *array* using a sequential loop.

In the program, the built-in method *size* is used to allocate the number of words to a partition based upon the reconfiguration table during mapping of the logical place to the processor, and the built-in method *length* gives the number of words in the distributed array *nun*.

```
dynamic region R = {1, 4, 5}; region R2 = {2, 3};
dynamic distributed class counter
{   synchronized static global int distributed array num[n] at region R2;
    monitor
    { distributed void increment () in region R2;
            {  foreach p in R2 {
                     m = p.num.size(num.length);
                     forall (i = 0:m)  ++ num[i];
                    }
            }

     { distributed void decrement () in region R2
            {    m = p.num.size(num.length);
                 forall (i = 0:m)  -- num[i];
            }
     }  //end monitor
}
int main ( )
{           foreach p in R counter p.c ;
            cout << ”1 for increment element, 2 for decrement elemnt, 3 to print the list ”
            switch (value)
            {           case1: increment (); break;
                        case2: decrement(); break;
                        case3:   place5.r.printData  (num);  //  invoke  the  remote  method
                        printData
                                    break;
            }
return 0;
}
```

```
local class print at place 5  // for printing distributed array at specific place
{    static local array  at place 5;
    distributed void printData (array)
        for (i = 0; i < n; i++)   // this loop to go over the array indices inside place
             cout<<  array[i];// // receive the distributed array num

}

int main ()
{          place p = 5;
           print p.r1;  // create the object r1 in the place p
           printData();
           return 0;

}
```

Figure 6:3 programming example illustrating synchronization and RMI

CHAPTER 7

Related Work

There are four classes of languages that have been developed for distributed high performance computing for large scale processors: 1) languages based upon distributed computing like Emerald [27] ; 2) languages supporting distributed computation over the Internet like Java [19, 29, 30]; 3) high productivity languages supporting MPI (Message Passing Interface) model on massive parallel processors such as MPIJava [32] ; and 4) languages supporting PGAS model like Chapel [7], X10 [8], UPC++[11] and more recently POBC++ [5]. The limitations of X10, Chapel and UPC++ regarding object distribution, object migration and object cloning and the lack of distributed object based paradigm present in Emerald with high productivity distributed array based SPMD programming present in X10, Chapel and UPC++ is already described in Chapter 1.

This research has been influenced by Emerald as well as the combination of features present in X10, Chapel and C++. For example, this language adopts the features of object cloning, flat objects, monitor, and blackboard, passing objects as parameters and remote method invocation from Emerald. Although, remote method invocation is presented in UPC++. Similarly, it borrows the concept of place, synchronization and distributed arrays from X10, concept of region and distributed arrays from Chapel, and dynamic distributed memory allocation from UPC++. All these different concept have been integrated in with C++ constructs to extend C++ language.

In addition, this thesis introduces: 1) the concept of dynamic and static regions for better user-transparent load balancing; 2) integration of SPMD model and object based model to introduce a new model MIDD (Multiple Invocation Distributed Data); and 3) multiple constructs that integrate SPMD model, synchronization models of Emerald and X10 with object distribution model to come up as new constructs. Synchronization is provided both at the method level using monitors, and at the variable level and data abstraction level using "synchronized" construct.

MPI based languages such as MPIJava will not provide the same type of productivity as PGAS based languages as described in Chapter 3 and 4 because MPI is based upon message passing, lacks global address space, and uses runtime address computation for remote method invocation and data communication [1].

Another approach is to extend existing popular object oriented languages such as C++ to reduce the learning time and to remain backward compatible with existing code library. Currently, there are two additional efforts: UPC++ [11] being developed by a group in Berkley National Laboratory and PobC++ [5] in 2014. Both these languages are enhancing C++11 constructs to incorporate task and data parallelism while retaining object oriented programming constructs of C++. Unlike X10 that is built on top of Java, their focus is to extend C++ constructs. However, like X10 and Chapel they have integrated parallelism only by extending procedural paradigm, and lack: 1) object distribution; and 2) integration of object distribution with task and data parallelism. As shown in Chapters 5 and 6, I have integrated object distribution as well as task parallelism at the method level with a new paradigm MIDD (Multiple Invocation Distributed Data) that is missing in current PGAS based languages.

The authors of this paper present PObC++ [5] (http://pobcpp.googlecode.com) exploit both object-orientation and distributed-memory parallelism. Like UPC++, PObC++ extends C++ for high performance computing . This work is focused on two different techniques in programming, Message-Passing (MP) and Object-Orientation (OO). The results of this work displayed accepted performance .The data showed that the approach may be able to combine object-orientation and parallelism in a language. Authors state that this programming style allows developers who are well educated in either MPI or OOP to be able to learn the other concept in PObC++ quickly, and to take advantage of the OOPP features.

In our point of view, in designing PObC++, they inspired their work to support a parallel programming by using MPI standard. However, there are some works such as [1] prove that the use of partitioned address space and the global synchronization in PGAS languages that affect on increases the productivity. For example, in X10 and Chapel because of global partitioning, the distributed of data becomes very easy. In contrast, in MPI to access an array element we need to compute its location. Even if the data structure is accessed globally, the process has to define the local storage for this data structure. Also, the overhead of message passing is an issue that addressed in MPI. Due to the integration with MPI and object oriented programming, the language supports: 1) point-to-point communication: 2) process topologies; and 3) dynamic process creation. In contrast to PObC++, the language developed in this thesis supports: place-to-place communication; 2) region-to-region communication; 3) dynamic region growth; 4) MIDD paradigm; 4) multiple paradigms related to object mobility. PObC++ also supports automatic dynamic process spawning when a flat object is created. This feature has been borrowed from Emerald [27].

CHAPTER 8

Conclusion

This chapter concludes the work, discusses some limitations of current work and the future works.

Due to 1) the need of incorporating a user friendly programming paradigm into parallel language, 2) as well as the increasing of interest in HPC applications to solve a grand challenge problems, and 3) the increasing of complexity in HPC applications has triggered the need for the development of new paradigms and languages that could facilitate software development that exploits high level parallelism using user-friendly software development for better productivity and maintenance.

PGAS model supports both local address space as well as global address space to achieve improved productivity over MPI based systems that have less productivity due to the lack of global address space. While there have been many languages such as X10, Chapel, Titanium and UPC++, they have only exploited SPMD model for mapping arrays into distributed domain. These languages still lack: 1) object-mobility, object-migration, object cloning and remote method invocation. The thread level programming supported by these languages is low level, and the languages do not support integration of object distribution paradigm with SPMD paradigm.

The present thesis is a work in that direction. This thesis identified the lack of object mobility, object cloning and object migration in these languages, and identified that the integration of these paradigms with parallel constructs. The integration of object distribution with data parallel programming at method level gives rise to a new higher level paradigm: MIDD (Multiple Invocation Distributed Data). This thesis also introduced new concept of logical dynamic regions that can grow and shrink with the process requirement. The thesis also benefits from the object distribution concepts developed by Emerald such as flat objects, objects passed as parameters, remote method invocation and monitors. C++ class has extended into three categories: *distributed class*, *local class*, and *flat class*. C++ has been extended in different aspects such as support for the migration of objects and the cloning of distributed objects. Many new constructs that are absent in C++ and UPC++ have been designed and their semantics has been discussed.

The grammar and the various constructs have been described, and multiple simple examples using object cloning, object migration and MIDD paradigms illustrated the usefulness of the developed constructs. This thesis is to contribute in the area of parallel language on desktops with distributed objects and object migration.

8.1 Limitations of the Current Work

This thesis has been able to integrate multiple distributed programming paradigm in Emerald, object oriented programming paradigm, distributed object paradigms and SPMD paradigm to come up with many new constructs and MIDD paradigm. It also communicates at object level instead of low level message passing. Multiple objects residing at different places can cooperate to solve a problem within a region. However, there are still multiple limitations such as: 1) objects in multiple regions cooperating to solve a complex tasks by splitting the cooperating subtasks across the regions. In the case of multiple regions, it is still not clear, if the region overlaps can be allowed and to what extent. The language is still not suitable to handle event based programming. Unless event based programming is supported, it can neither become interaction friendly nor real-time events friendly. Some of the issues with constructs will appear when the proposed language is implemented. At this point of time, I am unaware of the overhead of object mobility on the constructs supporting the integration of object-mobility with task and data parallelism. The implementation of “synchronized” distributed arrays, distributed vectors and blackboards will have serious overhead since the distributed arrays will span across multiple processing elements in terms of time delays when low level global lock mechanism is implemented. Solution of some of these synchronization issues will give rise to new synchronization concepts and constructs.

8.2 Future Work

The future work includes: 1) the integration of event based programming paradigm; 2) development of more constructs that integrate task level parallelism, object distribution and dynamic region; 3) implementation of these constructs using UPC++ or Emerald as middleware; and 4) development of high performance applications in the proposed languages. 5) Improve the construct and find out all library we need to manipulate the region.

REFERENCES

[1] S. Spetka, H. Hadzimujic, S. Peek and C. Flynn, "High productivity languages for parallel programming compared to mpi," in *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC,* 2008, pp. 413-417.

[2] J. Kepner, "HPC productivity: An overarching view," *International Journal of High Performance Computing Applications,* vol. 18, pp. 393-397, 2004.

[3] J. Diaz, C. Munoz-Caro and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *Parallel and Distributed Systems, IEEE Transactions On,* vol. 23, pp. 1369-1386, 2012.

[4] A. K. Bansal, *Introduction to Programming Languages.* CRC Press, 2013.

[5] E. G. Pinho and F. H. de Carvalho, "An object-oriented parallel programming language for distributed-memory parallel computing platforms," *Science of Computer Programming,* vol. 80, pp. 65-90, 2014.

[6] B. Chamberlain, *A Brief Overview of Chapel,* 2013.

[7] B. L. Chamberlain, D. Callahan and H. P. Zima, "Parallel Programmability and the Chapel Language," .

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices,* vol. 40, pp. 519-538, 2005.

[9] K. Ebcioglu, V. Saraswat and V. Sarkar, "X10: Programming for hierarchical parallelism and non-uniform data access," in *Proceedings of the International Workshop on Language Runtimes, OOPSLA,* 2004, .

[10] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu and D. Grove, "X10 language specification," *2010—08—28).Http://x10 — Lang.Org,* 2011.

[11] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan and K. Yelick, "UPC : A PGAS extension for C," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International,* 2014, pp. 1105-1114.

[12] M. Shapiro, P. Gautron and L. Mosseri, "Persistence and migration for C objects." in *ECOOP,* 1989, pp. 191-204.

[13] M. Weisfeld, *The Object-Oriented Thought Process.* Pearson Education, 2008.

[14] S. Brawer, *Introduction to Parallel Programming.* Academic Press, 2014, .

[15] D. Padua, "Encyclopedia of Parallel Computing," .

[16] (05 September 2015). *"Parallel Computer Memory Architectures." Introduction to Parallel Computing*. Available: https://computing.llnl.gov/tutorials/parallel_comp/.

[17] A. A. Kamil, "Single program, multiple data programming for hierarchical computations," 2012.

[18] E. Jul, H. Levy, N. Hutchinson and A. Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems (TOCS),* vol. 6, pp. 109-133, 1988.

[19] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. F. Riley, B. Topol and M. Ahamad, "Efficient implementation of java remote method invocation (RMI)." in *COOTS,* 1998, pp. 19-27.

[20] I. Kuz, F. Rauch, M. M. Chakravarty and G. Heiser, "System Architecture," .

[21] B. L. Chamberlain, S. Choi, S. J. Deitz, D. Iten and V. Litvinov, "Authoring user-defined domain maps in chapel," in *In CUG 2011,* 2011, .

[22] R. Cook, E. Dube, I. Lee, L. Nau, C. Shereda and F. Wang, "SURVEY OF NOVEL PROGRAMMINGMODELS FOR PARALLELIZING APPLICATIONS AT

EXASCALE," *Raport Instytutowy LLNL-TR-515971, Lawrence Livermore National Laboratory,* 2011.

[23] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky and O. Tardieu, "The asynchronous partitioned global address space model," in *The First Workshop on Advances in Message Passing,* 2010, pp. 1-8.

[24] B. Chapman, G. Jost and R. Van Der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT press, 2008.

[25] M. Weiland, "Chapel, Fortress and X10: novel languages for HPC," *EPCC, the University of Edinburgh, Tech.Rep.HPCxTR0706,* 2007.

[26] D. Khaldi, P. Jouvelot, C. Ancourt and F. Irigoin, "Task parallelism and data distribution: An overview of explicit parallel programming languages," in *Languages and Compilers for Parallel Computing*Anonymous Springer, 2013, pp. 174-189.

[27] A. P. Black, N. C. Hutchinson, E. Jul and H. M. Levy, "The development of the emerald programming language," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages,* 2007, pp. 11-11.

[28] A. Black, N. Hutchinson, E. Jul and H. Levy, *Object Structure in the Emerald System.* ACM, 1986.

[29] M. Philippsen and M. Zenger, "JavaParty - Transparent Remote Objects in Java," *Concurrency Pract. Exp.,* vol. 9, pp. 1225-1242, 1997.

[30] P. Kumar and R. Yadav, "Java Remote Method Invocation," *International Journal of Research,* vol. 1, pp. 694-699, 2014.

[31] (29 Sept. 2015). *CONCURRENCY IN C++*. Available: http://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/xia.pdf.

[32] M. Baker, B. Carpenter, G. Fox, S. H. Ko and S. Lim, "mpiJava: An object-oriented java interface to MPI," in *Parallel and Distributed Processing*Anonymous Springer, 1999, pp. 748-762.