Distributed Objects based Programming Constructs for PGAS based High Performance C++

Salwa D. Aljehan and Arvind K. Bansal
Department of Computer Science, Kent State University, Kent, OH 44242, USA

Abstract -- For high performance application software development on massive parallel processing, existing languages have to be augmented with new constructs and paradigms that exploit massive parallel computing and distributed memory models while retaining the user-friendliness. Available object-oriented languages for massive parallel computing such as Chapel, X10 and UPC++ exploit data and task parallelism at the process level in the PGAS memory model. However, they do not support automated class-template distribution, object migration and user-transparent dynamic growth of regions for load balancing. This paper describes new constructs that extends C++ with distributed class template distribution, dynamic regions, object cloning, object migration; and integrates data parallelism, task parallelism, automated class-template distribution, and object migration. The integration supports MIDD (Multiple Invocation Distributed Data) programming paradigm for user-transparent invocations of multiple copies of methods concurrently working on different data elements of the same distributed data.

Keywords: C++, distributed programming, high productivity, object-mobility, programming language, PGAS.

1 Introduction

Currently available supercomputers have peta-scale \((10^{15} \text{ instruction/second})\) capability. It is anticipated that by the end of the next decade, we will have exa-scale \((10^{18} \text{ instructions/second})\) computing power. This processing power needs to be fully exploited to solve big data problems in health science, weather science, agricultural science, space science, managing the Internet of things and modeling the population-related problems at the global and regional scale that will generate huge amount of data. Productivity is the most important issue that faces high performance computing against the backdrop of existing software library and the existing familiarity with the programming paradigms.

The processing of big data will require the development of user-friendly high-productivity programming tools that exploit massive number of processors. The development of such programming tools should be paradigm-friendly, and should be downward compatible to use the existing libraries. The development of such tools requires the integration of user-friendly paradigms such as event-based programming, object-oriented programming, web-based programming in addition to task parallelism and data parallelism currently being exploited on high performance computers.

Task parallelism splits a task into multiple subtasks that run on different processing elements concurrently; data parallelism broadcasts the same instructions to multiple processing elements to perform same operations on multiple data elements concurrently.

Currently available languages supporting large scale concurrent processing exploit data parallelism and task parallelism including spawning of multiple threads, and their integration. However, massive parallel processor configurations and available memory models pose issues about how to map problems and tasks among the processors to preserve efficiency due to synchronization introduced to handle race conditions and message passing overheads.

PGAS (Partitioned Global Address Space) is a popular model for high-performance computing. In PGAS, the distributed address space is divided into multiple local spaces connected through a global address space. The local spaces support multiple concurrent threads each with their own data area and a common shared space called heap. The communication between local address spaces is done by using global address space and message passing. Compared to MPI (Message Passing Interface) that suffers from excessive overhead of message passing, the use of local memory and distributed partitioned address space improves the productivity and execution-efficiency in PGAS model.

Currently available high-performance and high-productivity computing languages on PGAS such as Chapel, X10, UPC++ and SPMD (Single Program Multiple Data) paradigm, task parallelism, asynchronous computation and invocation of remote threads. UPC++ also supports runtime distributed memory allocation. While these languages support object-oriented programming, they do not incorporate object distribution and mobility; and remote method invocation described in Emerald, Java and other agent based languages. These languages also do not support dynamic distribution of objects and class templates for dynamic load management. In addition, they do not support the constructs that can be derived by the integration of object-oriented programming, task parallelism and data parallelism.

This paper describes extension of C++ by developing new programming constructs and paradigms that integrate object-oriented programming, web-based programming in addition to task parallelism and data parallelism currently being exploited on high performance computers.
oriented programming, compile-time user-transparent distribution of class-templates to a set of processing nodes, object migration, object cloning, task parallelism and data parallelism. The integration of these programming paradigms supports MIDD (Multiple Invocation Distributed Data) in which a distributed class-template is automatically distributed to a region (possibly dynamic) that is set of logical processing nodes called places in X10 [10, 11], and multiple copies of methods in the same distributed-class are invoked concurrently to process different elements of distributed data. The major contributions in this research are as follows:

1. Incorporation of sets of static as well as dynamic logical regions that can dynamically grow and shrink to accommodate load balancing in a user-transparent manner.
2. Incorporation of different types of classes for distributed computing: distributed class, local class and Emerald like flat class [4].
3. New programming constructs integrating object-mobility, cloning and distribution; task parallelism; and data parallelism for C++ and UPC++ languages.
4. Incorporation of higher level primitives such as monitor as in Emerald [4].

The rest of the paper is organized as follows. Section 2 describes the PGAS model, current languages for PGAS model and object-mobility in Emerald. Section 3 presents the abstract concepts. Section 4 describes the extended data abstractions. Section 5 presents the extended control abstractions. Section 6 illustrates the new constructs using examples. Section 7 presents related work. The last section concludes the paper.

2 Background

The “Partitioned Global Address Space” (PGAS) [3, 12] memory model has been proposed to overcome the limitations of the shared and distributed memory models. Overall address space is partitioned into multiple local spaces each having their own heap. These local spaces are connected through a shared global address space. Multiple threads execute locally, and can access remote locations asynchronously. The PGAS model supports SPMD control model. The data structures in this model can be distributed across address spaces. A distributed array is a data-space in shared memory such that different subranges are mapped on different places of local activities to exploit data-parallelism.

2.1 PGAS based high-productivity languages

There are many PGAS based high-productivity programming languages such as Chapel [8, 9], X10 [10, 11], UPC++ [18] that support object-oriented programming.

Chapel is a multithreaded high productivity computing language. It adopts a global view model, which means that a program starts with one thread, and based on the construct written by the programmer new threads can be spawned [3]. Data distribution and logical partitions are static and user-defined to map onto different architectural configurations. Chapel supports both data parallelism and task parallelism. Parallel and distributed data structures are supported by shared address space that connects various local partitions. The constructs forall-loop, domains, ranges and array are the basic data parallel features in Chapel.

X10 [10, 11] is a statically typed, object-oriented, high-performance and high-productivity computing language. It extends a sequential core language using features called places, activities, clocks, arrays, and struct types [11]. Like Java and C++, X10 makes use of classes, structs, and interfaces. X10 supports single inheritance [11]. Methods can be inherited and overridden in the subclasses. The reserved words “private”, “public” and “protected” are used to control the visibility of a method.

Central to X10 is the concept of a place, a collection of data and resident lightweight threads called "activities" [10]. Places map to a local processor, and contain a bounded number of activities and a bounded amount of storage. Creation of multiple places allows cluster-level parallelism. X10 introduced the notion of asynchronous activities for creating threads locally and remotely. X10 uses atomic statements to secure data-values limited only to the local scope. Since multiple processes need to be coordinated, it is necessary for X10 to use multiple barriers.

UPC++ [18] provides three main functionalities: 1) an object-oriented model for C++ language; 2) a collection of parallel programming constructs, not included in C++, to support high performance execution; and 3) a transition to PGAS programming through interoperability with other similar systems. The execution model of UPC++ is “Single Program Multiple Data” (SPMD). UPC++ implements asynchronous features through distributed-memory systems similar to C++11 standard asynchronous libraries for shared-memory systems. Synchronization is provided using primitives such as barriers, fences and locks to facilitate parallel programming. Remote function invocation allows asynchronous remote function with a single thread ID that is a place or a group of threads.

2.2 Emerald

Emerald is a flat object-based programming language [4, 14] that supports object-mobility [3] in a networked environment. Location-independent addressing allows object-mobility from node to node [14]. All entities are treated as objects. Concurrency is supported between objects and within an object. Variables shared by operations are synchronized using high level control-abstraction monitor. Emerald’s runtime system is responsible for the location and transfer of control to the target object.

2.3 Notations

We denote the data and control abstractions in italics and reserved words within double quotes “...”. The non-terminal symbols in the data-abstractions are enclosed in angular-brackets, are written in italics, and are self-
explanatory. For example, the non-terminal symbol `<distributed-class>` discusses about the declaration of distributed class. The grammar rules are written using extended BNF when required: sets are written using curly brackets `{...}`; optional use is written using square brackets `[..]`; alternatives are written using parenthesis and vertical bar `|`; multiple occurrences are denoted by `{...}`+.

3 Abstract concepts

The motivation behind the proposed DOPC++ (Distributed Object based Programming for C++) language is to incorporate high level user-friendly constructs that provide integration of object distribution and mobility with task and data parallelism while retaining the user-friendliness in the PGAS model. DOPC++ introduces a high level abstraction, and hides the low level object mapping on physical processors to increase the usability and user-friendliness.

DOPC++ supports: 1) distributed creation and dynamic migration of objects; 2) communication between remote objects; 3) cloning of distributed objects; and 4) extension of the notion of region in Chapel. Unlike static regions in Chapel, the regions in DOPC++ can dynamically grow and shrink. It describes distributed class, distributed methods, dynamic migratory methods that can be remotely invoked for dynamic load balancing. User-transparent dynamic object migration facilitates load balancing and performance improvement without loss of any functionality.

In DOPC++, a `place` is a logical computational entity like a virtual processor that is mapped to physical processor statically or dynamically. A `region` is a set of places and/or sub-regions that allow mapping of distributed data-abstractions.

Regions could be static or dynamic. A `static-region` is fixed during compile time. A `static region` is mapped to physical processors at compile-time, and does not grow or shrink during execution. Declaring a static-region allows every place in that region to get a copy of a distributed class-template and each place within this region will create a copy of the object in response to object-creation instruction. Unlike other PGAS based languages, DOPC++ also supports `dynamic regions`. A `dynamic-region` grows and shrinks at runtime based upon: 1) the computational need of the executing task; 2) resource-availability of the HPC system; and 3) load-balancing of the physical processors. Operating system performs runtime allocation of the places and the physical processors for dynamic regions.

Dynamic regions are bounded by a `problem space`. A `problem space` is a fixed set of places in which a dynamic region can grow. The rationale for the boundedness of dynamic regions is to limit the spread of very large problems that may affect the execution efficiency of other tasks. A dynamic region grows and shrinks during runtime, but cannot go beyond the `problem space`.

A `distributed data-abstraction` is distributed within a region with compiler and operating system deciding the granularity based upon: 1) available memory; 2) processing speed; 3) processor load; and 4) processor configuration table.

System level utilities inserted by the compiler take care of the mapping at run time. Objects methods and data elements migrate between processors dynamically in a dynamic region to balance the process-load. Within a dynamic region every place gets 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. Multiple objects could be invoked concurrently within a region, and each can work on an array of data elements independently.

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

Places exchange information with other places using PGAS shared address space, and require constructs to access address space in remote places within the same region. The migration of objects can be place-to-place, many places-to-one place, one place-to-many places, region-to-place, or region-to-region. 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 part of an object migrates based upon the type of mapping and available physical processor-load.

Communication between objects is done through a `remote invocation`. The parameters that are passed during invocations can be objects themselves. Parameter passing uses `call by object-reference` or `call by object move` as in Emerald [14]. In `call by object-reference`, the identifier that allows accessing the object remotely is passed as parameter. `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.

DOPC++ supports different type of classes: 1) `distributed-class`; 2) `flat-class` as used in Emerald [14]; 3) `local-class`; and 4) `regular C++ class`. A `distributed-class` has distributed data elements and/or distributed methods. The scope of a `distributed-class` is within a declared region. A user gives an initial region to start with.

A method embedded in a `flat-object` is invoked at the time of creation of the object. Active objects can share information using a `blackboard`. A blackboard is a synchronized shared address space that is shared between multiple threads. A `shared blackboard` can be in the global shared address space or it could be distributed among the places in a region. A `local-blackboard` is shared between local threads in a place. A `distributed-blackboard` is shared...
between the threads in the places within a static or dynamic region. A global-blackboard is shared between all the threads among multiple regions. Local-blackboards are used for sharing information between threads within the same place. All other processes sharing the blackboard are suspended to achieve synchronization until the writing process finishes writing on the blackboard. The synchronization-construct monitor ensures mutual-execution in the shared blackboard to serialize writing. The lock for a local blackboard is kept locally in the same place where the thread activities are taking place. The lock for a distributed-blackboard is kept in a shared address-space. When a thread writes on a blackboard, the lock is first captured to ensure that no other thread can write on the blackboard.

4 Extended data abstractions

A distributed data abstraction is declared after the specification of problem space and region (or sub-regions). Problem-space is declared as “problem space” \{<place-identifiers>\}. A region is declared as “region” \{<region-name> \{<place-identifiers>\}\}. A distributed-data-abstraction is declared as [<synchronization-type>] <compile-type><data-abstraction><location-type>.

Compile-type could be static or dynamic. Distributed-class or distributed data structures could be distributed, local, or global. Location-type specifies the region or place where data-abstraction is located.

4.1 Distributed-class declarations

A distributed-class is distributed in a static-region declared at compile-time. The name of the distributed-class is unique within a region, and an object-name is qualified by the place-identifier to make it unique. For dynamic distributed-class, a class-template is created in every place of the initial dynamic region at compile-time. A class-template migrates at run time as the region grows or shrinks based on the load balance. In the case of nested classes (inheritance), the whole class-hierarchy migrates. A flat-class allows objects to be easily distributed since there is no hierarchy. A class is considered hierarchical by default. A local-class resides in a specific place. An object-instance of a local-class is created in the same place without any possibility of object-migration. One motivation of using the local-class is to reduce the overhead of migration.

A distributed-method can be declared in a region or a specific place. A method-declaration in a region results in a copy of the method in each place within the region. The location-name of a method-declaration can be different than the location-name of the corresponding class-declaration if the methods are invoked remotely.

A <distributed-class> is declared as (“static”|“dynamic”) (“distributed” | “flat” | “local”) <class-declaration> <region-declaration>. A <flat-Class> is declared as “flat” <class-declaration> <region-declaration>. A <local-class> is declared as “local” <class-declaration><place-declaration>. A <distributed-method> is declared as “distributed-method” <method-declaration> “in” (“region” | “place”) <location-name>. A distributed-method is declared as <data-type> followed by the <method-name> that is followed by the <typed parameter-list>.

4.2 Distributed data structures declarations

Elements of a distributed-array (or distributed-vector) are split in different places of the region based upon user-defined granularity. Granularity is the number of consecutive elements in an array (or vector) that occur in the same place. The granularity for each dimension can be different.

A <distributed-array> (or <distributed-vector>) is declared as “distributed” (or “global”) followed by “array” (or “vector”) followed by <array-name> (or <vector-name>). Array-name is followed by dimension-list. Dimension-list is a non-empty sequence of the form <dimension>:<granularity>. The distributed-array (or distributed-vector) declaration is followed by location declaration (“in region” | “in place”) <location-name>. A global-array is allocated in the shared partitioned space.

The elements of distributed-array are processed in parallel using multiple invocations of a method in the resident copies of the objects in different places of the region. This paradigm is called MIDD (Multiple Invocation Distributed Data). When a distributed-array is defined as “global” then the elements are shared in global address-space.

4.3 Shared data and synchronization

A <distributed-blackboard> is declared by “distributed blackboard” <blackboard-name> “at region” <region-name>. A <local-blackboard> is declared as “blackboard” <blackboard-name>. A <global-blackboard> is declared as “global” <blackboard-name>.

Data-objects are synchronized by tagging with a reserved word “synchronized” preceding the declared data-objects. The granularity of the synchronization is at the data abstraction level. A monitor control-abstraction is used to provide mutual exclusion of statements working on synchronized data-objects. All the statements occurring within the block following the reserved word “monitor” form the critical section. The use of monitor secures the lock(s) associated with the corresponding synchronized data-objects before executing the embedded methods. Monitor creates a queue for the processes waiting to access the corresponding synchronized data-object(s). This restriction imposes sequentiality. For SPMD operations on distributed data-objects, this restriction can cause serious efficiency overhead. To avoid this, the default mode for aggregate data-abstracts is asynchronous.

5 Control abstraction extensions

Major extensions are migrate, remote invocation of methods, clone, get-object-location, move-object, remotely-delete-object, get-remote-value, blackboard-put, blackboard-get in addition to SPMD statements working on distributed-
arrays, distributed-vectors, and distributed-blackboards.

The purpose of object-migration is to distribute the workload. During object-migration, 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 limited number of places of the destination region. Object-migration is declared as “migrate” <object-name> “to” (<region-name> | <place-name> |).

A remote-method is needed to perform some computation to balance the computational load or to use a specific resource present at a specific place. A remote-method is invoked using <place-name>.<object-name>.<method-name>( ).

Cloning is used to duplicate an object for migration to another place or region. However, the object should be cloneable to execute this method. An object that is an instance of a local-class is fixed in one place and cannot be copied to another place. Cloning of a distributed-object is declared as “clone” <object-name> “in” (<region-name><place-name>) “to” (<region-name> | <place-name>).

The possible name-conflict during migration of objects or cloning of objects across regions needs two global tables for each region: 1) a table of object-names for each region that is checked when an object is migrated to avoid duplication; and 2) a correspondence table between the object-name in the source-region and the object-name in the destination region.

There are two control-abstractions that support SPMD paradigm: 1) standard forall statement that works on every element of a distributed-array (or distributed-vector) concurrently; and 2) foreach statement that works on elements in a subset concurrently. A foreach statement is written as “foreach<variable-name> “in” (<set> | <region-name>). The set could also be a set of places marking a sub-region. Foreach construct invokes multiple copies of a method distributed in a set of places in a region. For communication between places, objects can be passed as arguments.

DOPC++ uses three additional built-in methods: size, length and Indexset. The method size computes the number of elements in a distributed-array that can be allocated to each place by knowing the place capacity and load. The method length returns the number of allocated elements of dynamic distributed array (or vector) at each place. Indexset computes the set of indices of a distributed array element in a place.

6 Programming examples

In this section, we illustrate extended constructs using two examples. Example 1 illustrates static-region, MIDD (Multiple Invocation Distributed Data), object-distribution, object-migration, and the use of distributed-arrays and static distributed-class. Example 2 illustrates dynamic-region, object-cloning, synchronization using monitor, static distributed-class and dynamic distributed-class. We have used extended C++ syntax in the programs.

6.1 Example 1 - object migration and MIDD

The program in Figure 1 illustrates distributed class and object migration from a place in a region R = {1, 2, 4, 5} to another region S = {3} to utilize a printer resource available in the region S. Declaring a “static distributed-class” in region R allows every place in region R to get a copy of the distributed class-template dictionary. Four class templates, one for each place within the region R are created transparently by a compiler. The program has a distributed array word and four functions: main, lookup, store and printData. The program creates distributed objects for static distributed-class dictionary, and calls store and lookup methods in a region R = {1, 2, 4, 5}, and the data is printed in a different region S = {3}.

```cpp
problem space = {1, 2, 3, 4, 5};
region R = {1, 2, 4, 5}; region S = {3};

static distributed class dictionary at region R
{ Public:
  int n = 100; gs = 25; // gs is grain-size of word in a place
dynamic string distributedArray word[n] at region R;
}
distributed void store ()
{ cout<< "Enter new words";
  for (i = 0; i < gs; i++) cin >> word[i]; // read words
}
distributed void lookup () in region R
{ string w;
  Boolean found = false;
  COut << "Enter the word to look up:"; cin >> w;
  foreach i in indexset(word) // search word
  if (word[w] == w) { found = true; break; }
  if (found) return('found'); else return('search fails');
}
remote printData () in region S
{ int i, m = 0;
  foreach p in R // for each place in region R do
  { m = p.word.size(word.length);
    for (i = 0; i < m; i++) // read the words
    cout << word[i]; // print the words
  } // end foreach
}
} // end distributed class dictionary

int main ()
{ int value;
  Boolean keepLooping = true;
  new dictionary d // create object d in every place in R;
  cout << "Enter new words; 2 to look up; 3 to print;";
  while (keepLooping) // read the commands
  { cin >> value;
    switch (value)
    { 1: store (); break;
      2: lookup (); break;
      3: keepLoong = false; d.printData(); break;
    } // end reading the commands
  } // end main
}
```

Figure 1. Illustrating object migration and MIDD

The function lookup looks at various places concurrently. The number of spawned threads depends upon operation system according to the load balancing. The function store reads one word at a time, and stores in the distributed array word. The function store spawns concurrent
threads in all four places of the region R. Migration creates an object in the place 3 that will print the distributed array word iteratively. The object d migrates to place 3, which has a printer resource. The remote-method printData spawns a thread to iteratively print the dictionary words.

The main program invokes three functions: 1) store( ) to store the words in the distributed array word; 2) lookup( ) to lookup a given word in the distributed array word; and 3) print function printData( ) to print the data. The main program terminates after printing the dictionary. The functions are invoked based upon the input value value.

The statement “dictionary d” in main program automatically creates multiple instances of objects 1.d, 2.d, 3.d, 4.d, one in each place, within the region R. Each copy of the object works concurrently on the distributed array word during the execution of functions store( ) and lookup( ). The local Boolean variable found is used to break the foreach loop after the successful search. To insert new words inside the distributed array word, multiple invocations for all objects are done concurrently. A built-in method size computes the number of words allocated in each place.

6.2 Example 2 – cloning and synchronization

The program in Figure 2 illustrates dynamic distributed-class, object-cloning and synchronization. The program has a dynamic distributed-class math in a dynamic region R with four initial places \{1, 2, 3, 4\}. Multiple copies of the class-template math, one for each place within the dynamic region R, are created automatically after the execution of the statement math m in the main program. The method m.add( ) and the distributed array num are created in the places 1, 2, and 3 constituting the sub-region R1.

The distributed array num within the region R1 is added by spawning concurrent threads in different places in the region R1. The result is stored in the local copy of the variable accum in each place of the region R1. The values stored in local copies of accum are added to the synchronized global shared variable sum to get the cumulative total; concurrent threads are spawned in the places within the region R1. The synchronization between concurrent active threads is maintained using a monitor.

A static distributed class test is created in the region R2 to derive even or odd numbers using isEvenOdd(). Cloning of the object m in the main program creates distributes copies of the object testarray in the places \{6, 7, 8\}. The method isEvenOdd( ) filters and stores the numbers in two distributed arrays: even and odd. The built-in method size retrieves the number of data-elements of a distributed-array stored in a place based upon the reconfiguration table. The built-in method length gives the number of data-elements in the distributed-array.

In the method isEvenOdd, monitor is needed for the shared synchronized index-variables j and k that are being updated by multiple concurrent threads, each trying to update distributed-arrays even and odd concurrently.

7 Related works

Three types of high-productivity languages are being developed: 1) MPI based languages such as mpiJava [2] and POoC++ [16]; and PGAS based languages such as Chapel [8, 9], X10 [10, 11], and UPC++ [17]; 3) CSP (Communicating Sequential Processes) based such as OCCAM [13] and Rain[7].

In mpiJava [2] and POoC++ [16], the integration of MPI and object-oriented programming supports: 1) point-to-point communication; 2) process topologies; and 3) dynamic process creation. POoC++ also supports automatic dynamic process spawning when a flat-object is created. However, the overhead of message passing is an issue in MPI. MPI based languages do not provide the same type of productivity as PGAS based languages [15]. In addition, MPI based languages cannot exploit the data abstractions based upon the shared partitioned space.
Both UPC++ [18] and PObC++ [16] extend C++11 to incorporate task and data parallelism while retaining object-oriented programming constructs of C++11. However, they lack: 1) object distribution; 2) object cloning; 3) object migration; and 2) integration of object-distribution with task and data parallelism.

Multiple integrations [5, 7] of C++ with OCCAM [13] have been proposed to integrate object-oriented programming in C++ with CSP-like parallelism [6]. However, these integrations do not support object-migration, object-cloning and automated distribution of distributed-class templates.

This paper extends the work of prior PGAS based languages [8, 9, 10, 11, 18] by introducing: 1) the concept of static and dynamic regions for better user-transparent load balancing; 2) integration of object-distribution and object-migration to exploit MIDD (Multiple Invocation Distributed Data) paradigm; 3) dynamic growth of regions; and 4) separation of logical notion of place and regions from physical processors. Many new constructs have been developed using these extension of concepts.

8 Conclusion and future work

We have extended the PGAS based high performance computing language development effort by integrating class-template distribution, object-mobility and remote method invocation with task and data parallelism. We have also extended the concept of region to include dynamically growing and shrinking regions to take care of load balancing. The integration of object-distribution and remote method invocations supports MIDD (Multiple Invocation Distributed Data) programming paradigm where multiple methods can be automatically distributed to work on distributed data-elements of an object in a distributed class. The separation of logical place and regions from physical processors provides more architecture independent high level programming that we believe will provide better adaptability of object-oriented high performance computing to desktop as high performance computing moves to desktops.

We need to investigate other process invocation paradigms in the models integrating CSP derivatives and C++ [5, 7] to look into alternate communication models. We are also looking into developing a translator that can translate these constructs to languages like UPC++ or X10. We are extending the language further for integrating event-based programming with object distribution.

9 References


