Partitioned Global Address Space Languages

MATTIAS DE WAEL, STEFAN MARR, BRUNO DE FRAINE, TOM VAN CUTSEM, and WOLFGANG DE MEUTER, Vrije Universiteit Brussel, Belgium

The Partitioned Global Address Space (PGAS) model is a parallel programming model that aims to improve programmer productivity while at the same time aiming for high performance. The main premise of PGAS is that a globally shared address space improves productivity, but that a distinction between local and remote data accesses is required to allow performance optimizations and to support scalability on large-scale parallel architectures. To this end, PGAS preserves the global address space while embracing awareness of non-uniform communication costs.

Today, about a dozen languages exist that adhere to the PGAS model. This survey proposes a definition and a taxonomy along four axes: how parallelism is introduced, how the address space is partitioned, how data is distributed among the partitions and finally how data is accessed across partitions. Our taxonomy reveals that today's PGAS languages focus on distributing regular data and distinguish only between local and remote data access cost, whereas the distribution of irregular data and the adoption of richer data access cost models remain open challenges.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Concurrent, distributed, and parallel languages; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures

General Terms: Design, Languages

Additional Key Words and Phrases: Parallel programming, HPC, PGAS, message passing, one-sided communication, data distribution, data access, survey

ACM Reference Format:
Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Comput. Surv.* x, x, Article x (January 2015), 29 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

High Performance Computing (HPC) is traditionally a field that finds itself at the crossroads of software engineering, algorithms design, mathematics, and performance engineering. As performance and speed are key in HPC, performance engineering aspects such as optimizing a program for data locality often thwart software engineering aspects such as modularity and reusability. Traditional HPC is based on a programming model in which the programmer is in full control over the parallel machine in order to maximize performance. This explains the continuing dominance of languages such as Fortran, C, or C++; usually extended with MPI for message-passing across

Mattias De Wael is supported by a doctoral scholarship of the agency for *Innovation by Science and Technology* in Flanders (IWT), Belgium.
Stefan Marr was funded by the SAFE-IS project in the context of the *Research Foundation, Flanders* (FWO). He is currently affiliated with INRIA, Lille, France.
Tom Van Cutsem is a post-doctoral fellow of the *Research Foundation, Flanders* (FWO).
Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter are affiliated with the *Flanders ExaScience Lab, Intel Labs Europe*.

DOI: http://dx.doi.org/10.1145/0000000.0000000

parallel processes. The way these processes access and manipulate their data, e. g., the way in which large arrays are divided among processors, and the patterns they use for communication, need to be manually encoded. However, this programming model is slowly losing momentum:

First, we observe a continuing trend towards ever more complex hardware architectures. Today, machines have very heterogeneous designs with respect to processor connectivity as well as memory architecture. Compare for instance the characteristics of multicore, many-core, GPGPU, accelerators, and clusters. Programmers thus have to deal with issues such as *non-uniform memory access* (NUMA) and *non-uniform cluster computing* (NUCC). Further, in order to reach exascale performance, i. e., 10^{18} flop/s, ever more clever algorithms are required. The combination of both phenomena leads to an unmanageable source of complexity of HPC algorithms and code.

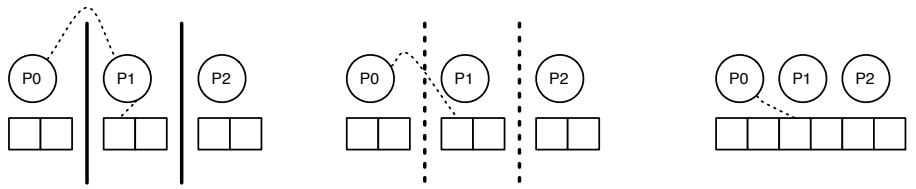
Second, this complexity is becoming a problem for a larger group of people: supercomputers today are easily accessible over the internet and HPC is being used in more and more application domains. Furthermore, the multicore crisis [Sutter 2005] brought the benefits and issues of parallel programming straight to the desktop and even to mobile platforms, and thereby exposes an increasing number of people to languages and programming paradigms formerly restricted to the field of HPC.

Third, as witnessed by the High-Productivity Computing Systems DARPA project (HPCS), programmer productivity is gradually becoming an issue. Quoting Lusk and Yelick [2007]: *"there exists a critical need for improved software tools, standards, and methodologies for effective utilization of multiprocessor computers"*.

The above observations have given rise to a new generation of parallel programming languages explicitly targeted towards HPC that aim to unite performance-aware programming models with high productivity. These programming languages tend to shift focus from isolated processes, which explicitly communicate with one another, to a more global view of an HPC algorithm. It is then the job of the compiler—possibly guided by pragmas or annotations—to map such global specifications onto processors and links between processors. Although some exceptions exist, to date such languages often adopt the *Partitioned Global Address Space* (PGAS) model.[1] In this model a number of parallel processes jointly execute an algorithm by communicating with one another via memory that is conceptually shared among all processes, i. e., there is one single address space. However, at the hardware level this conceptually shared memory is realized by several memories that are interconnected such that not all logical memory addresses have the same access latency. These memories can belong to a processor, a blade of processors, a rack of blades, and even a cluster of racks. Therefore, PGAS languages provide additional abstractions to distinguish between local and remote data access.

Before PGAS, HPC programming models could be clustered into two main groups: message-passing models such as MPI, where isolated processes with isolated memories exchange messages (fig. 1a), and shared-memory models, as exemplified by OpenMP, where multiple threads can read and write a shared memory (fig. 1c). The PGAS model can be situated in-between these models (fig. 1b). From the shared-memory model, it inherits the idea that a parallel program operates on one single memory that is conceptually shared among all its processes. From the message-passing model, it inherits the idea that communication between processes is associated with a certain cost. Figure 1 shows these three memory models and depicts processes as circles and memory locations as rectangles. The dashed lines show memory accesses, which in the case of message-passing explicitly go through a distinct process in the form of a message.

[1] To our knowledge, the current PGAS terminology originates from the work on Split-C, where Culler et al. [1993] describe a *global address space* which is *partitioned over the processors*.



(a) Message-passing (b) Partitioned-memory (c) Shared-memory

Fig. 1: Memory models describe how data can be accessed. The partitioned-memory model, as assumed by PGAS languages, can be situated in-between the message-passing and shared-memory models.

Finally the vertical lines between processes and memory sections, respectively solid, dashed, and absent in the subfigures, illustrate the conceptual distance between memory sections and the associated cost of accessing its data.

Recently, PGAS languages such as Chapel, X10, and Fortress have attracted a great deal of attention. However, the history of PGAS languages is much richer. Over a dozen PGAS languages exist. This article presents the state of the art of this field. We start by outlining the landscape of PGAS languages from a merely historical perspective. We then propose a definition of what a *PGAS language* is and we present a set of definitions for terms we use throughout the text, which allows for the introduction of our taxonomy of PGAS languages. Subsequently, we describe ten exemplary PGAS languages in more detail by discussing their main traits and by presenting a textbook example program. We conclude by pointing out the blank spots in our taxonomy, which indicate interesting avenues for future research.

2. HISTORICAL PERSPECTIVE

This section gives an overview of what we consider to be the most typical PGAS languages. We cluster them into four groups, primarily based on the timeframe and context in which they were invented: the original PGAS languages from the late 1990s, the High Productivity Computing Systems (HPCS) PGAS languages from around 2004, the retrospective PGAS languages from the early 1990s and before, and finally the most recent PGAS languages developed after 2005. Figure 2 visualizes these categories and specifies for each PGAS language if it is considered to be middleware (MW), a language extension (E), a language dialect (D), or a new language (L). The left side of the bounding boxes shows the year in which a language was first presented in an academic paper.

Section 2.5 briefly mentions system-level PGAS libraries. Since these libraries are targeted to language or library implementors instead of application developers, they are outside the scope of this survey and are only mentioned briefly.

2.1. Original PGAS Languages

In the late 1990s, three new languages emerged putting the term PGAS on the map of programming language design. In 1998, the language definitions of *Co-Array Fortran* and *Titanium* were published and one year later *Unified Parallel C* was introduced. These three languages respectively extend Fortran, Java, and C with a partitioned global address space, a *single program multiple data* (SPMD) execution model, and corresponding data structures and communication facilities.

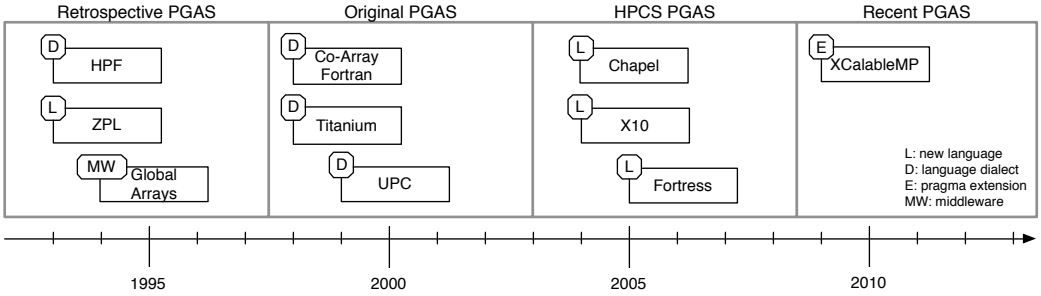


Fig. 2: Overview of PGAS languages and their categorization.

Co-Array Fortran (CAF) is a parallel extension of Fortran 95 and adds the *co-array* as a new construct to the language. The extensions were proposed by Numrich and Reid [1998] and were included in the Fortran 2008 standard in 2005 [Numrich and Reid 2005]. The philosophy of CAF is that the cost of accessing remote data should be manageable and explicit. Therefore, CAF encourages the programmer to write as much local code as possible and to use remote data accesses sparingly. Since remote accesses are syntactically different from classic Fortran code, excessive use of those syntactically conspicuous and expensive operations, can be considered to be a "bad smell" in CAF code and should be avoided as much as possible.

Titanium is a Java dialect designed for high-performance parallel scientific computing [Yelick et al. 1998]. It was conceived at UC Berkeley and provides implementations for symmetric multiprocessing (SMP) as well as for distributed systems. The language is designed to enable explicit parallel programming, while facilitating compiler optimizations for optimal performance. It provides the notion of local and remote references and uses explicit communication primitives to exchange data. One specific design goal was to bring OOP to the field of scientific computing.

Unified Parallel C (UPC) [El-Ghazawi et al. 2005] is a PGAS extension of C, which integrates features from three earlier proposals: PCP [Brooks III et al. 1992], Split-C [Culler et al. 1993], and AC [Carlson and Draper 1995]. The specification of the UPC language is authored by the UPC consortium, which consists of academic and government institutions as well as companies. The first version of UPC, version 0.9, was published in May 1999, while the current version (UPC 1.2) is from 2005 [UPC Consortium 2005]. Well-known implementations of the UPC language include Berkeley UPC, GNU GCC UPC, and HP UPC. UPC programs can make use of *shared* data objects, which is the main PGAS facility of the language. Data values that reside in shared memory are hosted by one of multiple threads but can be accessed in a syntactically transparent way from different threads, even though a 'remote access' normally comes at a communication cost.

2.2. HPCS PGAS Languages

In 2004, roughly half a decade after the advent of the original PGAS languages, in the context of phase II of DARPA's *High Productivity Computing Systems* (HPCS) project *Chapel*, *X10*, and *Fortress* were developed. The HPCS project aimed to create multi-petaflop systems with high productivity, where productivity is seen as the combination

of performance, programmability, portability, and robustness.[2] As opposed to CAF, Titanium, and UPC, these languages do not merely extend existing languages with new concepts, but are designed as new languages based on their corresponding main principles.

Chapel is a parallel programming language developed by Cray as part of the Cray Cascade project. Chamberlain et al. [2007] identify (i) the global view of computation; (ii) the support for both task and data-driven parallelism; and (iii) the separation of algorithm and data structure details as the main programmability concepts of the language. Chapel provides concepts for multithreaded and locality-aware parallel programming. The language also supports many concepts from object-oriented languages and generic programming.

X10 is a programming language developed by IBM Research [Charles et al. 2005]. The name X10 refers to *times 10*, the aim of the language to achieve 10 times more productivity in HPC software development. X10 is described as a modern object-oriented programming language providing an asynchronous PGAS programming model with the goal of enabling scalable parallel programming for high-end computers. X10 extends the PGAS model with asynchronicity by supporting lightweight asynchronous activities and enforcing asynchronous access to non-local state. Its explicit fork/join programming abstractions and a sophisticated type system are meant to guide the programmer to write highly parallel and scalable code, while making the cost of communication explicit. The task parallelism is implemented on top of a work-stealing scheduler.

Fortress is a programming language designed for high-performance computing, originally developed by Sun Microsystems, an effort continued at Oracle Labs until July 2012.[3] The expressive type system facilitates static analysis, while efficient scheduling of implicitly parallel computations is guaranteed by the work-stealing algorithm. Another characteristic of Fortress is its mathematical syntax. The use of unicode for instance for the sum operator Σ and the idea of "typesetting" code give the language a mathematical look-and-feel.

2.3. Retrospective PGAS Languages

There exist a number of languages that have some characteristics of the PGAS approach but that were never explicitly described as a PGAS language when they were released, because they actually predate the term PGAS and with it the original PGAS languages. Thus, we name these *retrospective PGAS languages*.

High Performance Fortran (HPF) emerged in 1993 as one of the first PGAS-like languages. HPF [High Performance Fortran Forum 1993; Koelbel et al. 1994] is a *data-parallel language* for distributed parallel computers, unifying the concepts of older languages such as FortranD [Callahan and Kennedy 1988], Vienna Fortran [Zima et al. 1988], and CM Fortran [Thinking Machines Corporation 1991]. The language was defined through a combined academic and industrial effort. The three main design goals of HPF were according to Kennedy et al. [2007]: (i) a global address space in which large data structures are physically distributed; (ii) an apparently single thread of control where parallelism emerges from parallel operations on the distributed data

[2]*High Productivity Computing Systems (HPCS)*, Defense Advanced Research Projects Agency, access date: Feb. 26th, 2013 http://www.darpa.mil/Our_Work/MTO/Programs/High_Productivity_Computing_Systems_(HPCS).aspx

[3]*Fortress Wrapping Up*, Guy Steele, access date: Feb. 26th, 2013 https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up

structures; and (iii) implicit communication derived from a declarative specification of the data layout. Based on this, one could argue that HPF is a precursor of modern PGAS languages [Kennedy et al. 2007].

ZPL is a data parallel array programming language, developed in the early 1990s at the University of Washington [Lin and Snyder 1994; Snyder 2007]. ZPL provides two classes of data objects, i. e., scalars and arrays, on which the usual operators (e. g., arithmetic and logic) can be applied. In the case of arrays, these operators are *lifted* and are applied point-wise. For arrays ZPL introduces, besides the usual operators, parallel prefix operators. ZPL has sequential semantics, i. e., all parallelism is implicit and arises from applying lifted or parallel prefix operators on the arrays. Also, the partitioning of the global address space is implicit in ZPL.

Global Arrays Toolkit (GA) is a high-level library developed by the Pacific Northwest National Laboratory [Nieplocha et al. 1994]. An example application that uses GA is NWChem, a computational chemistry package. GA provides an API for programming distributed-memory computers using *global arrays*: physically distributed dense multi-dimensional matrices that are shared between processes. Each process can asynchronously access logical blocks of a global array, without need for explicit cooperation by other processes. The locality information for the shared data is available, and a direct access to the local portions of a global array is provided. Because of its high-level nature, we discuss it together with other PGAS languages.

2.4. Recent PGAS Languages

Other PGAS languages have been conceived after 2005 independently of DARPA's HPCS program. One of those *recent PGAS languages* is XCalableMP[4] [Lee and Sato 2010].

XCalableMP (XMP) is a PGAS-extension for both C and Fortran, closely resembling the OpenMP compiler extensions. Like OpenMP, it consists of a set of compiler directives or pragmas that alter the semantics of a sequential program. The XMP Specification Working Group designed XCalableMP based on their experience in developing HPF programs. Besides the influences of HPF, CAF also served as an inspiration in the development of XMP. The large discrepancy in programming style between HPF and CAF seems irreconcilable. Therefore XMP encourages its programmers to pick one style depending on the algorithm.

2.5. System-level PGAS Libraries

The terminology *partitioned global address space* is also used in the context of a number of communication libraries such as MPI-2 [Geist et al. 1996], OpenSHMEM/SHMEM [Chapman et al. 2010; Cray Inc. 1999], GASNet [Bonachea 2002], ARMCI [Nieplocha and Carpenter 1999], and GPI [Pfreundt 2010]. These libraries allow SPMD programs to register memory segments for remote memory access (RMA) through one-sided operations such as `get`, `put`, and `accumulate`. MPI-3, as standardized in 2012, tries to fix the MPI-2 RMA API, as it appeared not to meet the needs of application programmers [Bonachea and Duell 2004]. GASNet is used by Berkeley UPC and other PGAS languages,[5] while ARMCI is used for instance by Global Arrays. Because they provide only low-level functionality these libraries are not meant to be

[4] *XcalableMP Website*, XcalableMP Specification Working Group, access date: 19 November 2014 http://www.xcalablemp.org/

[5] *Systems/Projects using GASNet*, Berkeley, access date: 28 February 2013 http://gasnet.cs.berkeley.edu/

used directly by application developers. Therefore, we do not consider these libraries in this survey.

3. DEFINITIONS

In order to classify programming languages as languages supporting the PGAS programming model, we propose the following definition. A programming language is a PGAS language if:

(1) the language is intended for parallel programming (Parallel Execution Model, section 3.1),
(2) the language makes data access cost explicit by describing a partitioning of the global address space (Places Model, section 3.2),
(3) the language specifies how the data is or can be distributed over the different memory partitions (Data Distribution Model, section 3.3), and
(4) the language allows for data access with the perception of a shared memory (Data Access Model, section 3.4).

These four requirements form the four axes in our taxonomy of PGAS languages. The concepts that arise from these requirements are captured into four *models*, one for each requirement, that can be instantiated in various ways. We describe these four models in greater detail below.

3.1. Parallel Execution Model

The *Parallel Execution Model* describes how the parallel activities within a program are launched and executed. Without imposing any limitations on their implementation, we refer to such a parallel activity as *a thread*. We distinguish three main parallel execution models in current PGAS languages:

Single Program Multiple Data (SPMD). At program startup, a fixed number of threads is spawned. Each thread executes exactly the same program, but is parameterized with a *thread index* that is unique for each thread, allowing computations in each of the threads to diverge. Most early PGAS systems adhere to this execution model, including UPC, Co-Array Fortran, Titanium, and Global Arrays.

If we were to represent a parallel machine with N processing elements as an N-tuple, then the starting configuration of an SPMD program P can be represented by the tuple $(P(0), P(1), \ldots, P(N-1))$.

Asynchronous PGAS (APGAS). At program startup, a single thread starts execution at the program's entry point, which is the `main` function in many languages. Constructs are provided to *spawn* new threads dynamically, running in the same or in remote partitions of the address spaces. Each spawned thread may execute different code. Examples of this approach include X10, Chapel, and Fortress.

The starting configuration of an APGAS program P can be described as $(P, \emptyset, \ldots, \emptyset)$, where $\emptyset$ indicates that a processing element starts out being idle, i. e., it is waiting until the main program P forks additional activities.

Implicit Parallelism. No visible parallelism, or directives to control parallelism, are present in the code, i. e., the program describes a single thread of control. At runtime, multiple threads of control may be spawned to speed up the computation, but this parallelism is implicit in the program's code. An example PGAS language featuring implicit parallelism is High-Performance Fortran, where a statement such as `FORALL` implicitly runs all its iterations in parallel.

The starting configuration of an implicitly parallel program P can be described as a tuple of projections $P|_i$ of the original program P: $(P|_0, P|_1, \ldots, P|_{(N-1)})$. While each

processing element executes a part of P, the precise instructions of each $P|_i$ are typically not controlled by the programmer, but rather by a compiler or language runtime.

3.2. Places Model

In order to account for the NUMA characteristics of large-scale systems with multiple computational nodes, the PGAS model partitions a globally addressable memory space into *places*. Typically, a single place corresponds to a single computational node so that a place can access its local memory with a comparably uniform and minimal cost. Accessing data from different places, however, comes at a higher cost. These *concrete costs* vary depending on the underlying hardware, e. g., communication between blades servers is slower than communication between the chips within a single blade. The places model abstracts away from the concrete underlying hardware by describing a *topology*, i. e., overall structure, of the partitioned address space on the one hand and the *abstract memory access cost function*, i. e., an indication of the different possible costs, on the other hand. Languages that adhere to the PGAS model can differ from each other in both the partitioning topology and the memory access cost function.

Places Interconnection Topology. The relation between different places—as it is observable from the programming language or as it is specified upon program startup—is defined by the *places interconnection topology*, or topology for short. The most common topology being used in PGAS languages today is a flat ordered set of places, i. e., each place can be assigned an index number in the interval $[0, n[$ where n is the number of partitions. This index is observable from the programming language, e. g., by a call to an intrinsic function. Besides the flat ordered set topology, also rectilinear grid, e. g., in HPF and ZPL, and hierarchical tree topologies, e. g., in Fortress, can be found. The topology used in a PGAS language tries to find the middle ground between an abstract representation of physical connections between the underlying (distributed) hardware (e. g., hierarchical tree) and an abstraction representation that is convenient to express the parallel computations (e. g., rectilinear grid to match computations on matrixes).

Abstract Memory Access Cost Function. In traditional programming languages, shared memory is treated as having uniform access cost. Even though PGAS languages give the programmer the perception of a shared memory, they intend to make memory access costs arising from the underlying NUMA model explicit. As noted before, intra-place communication is typically cheap, while inter-place communication can be more expensive when the distance between places gets larger.

The *abstract memory access cost function*, or in this context *cost function* for short, specifies the cost for this inter-place communication. The input variables for the cost function are the place where the data is needed and the place where the data resides. We say this cost function is *abstract* because it only gives an indication of the memory access cost. The concrete memory access costs are hardware specific.

Most PGAS languages today have a cost function with only two possible results, i. e., either *cheap* or *expensive*. We say these languages have a 2-level cost function. The Fortress language intended to support different levels of expensive memory access, i. e., a multi-level cost function, but it was never officially implemented (cf. Allen et al. [2008, chap. 21]). Yan et al. [2009] also investigated places with a hierarchical cost function, i. e., tree topology. A language with a cost function that produces only one outcome implicitly assumes uniform memory access and are therefore not considered to be a PGAS language.

To conclude, the places model allows us to differentiate between *local* and *remote* data. Data are considered local when, according to the *places model*, the access cost is lowest. All other data are said to be remote. This distinction is of course relative to a

place. Orthogonal to the distinction between local and remote, we can also differentiate between *private* and *shared* data, where shared data is data that is accessible from multiple, typically all, places whereas private data can only be accessed from a single place.

3.3. Data Distribution Model

The *data distribution model* defines how data objects are distributed over places. Languages where the programmer has no direct control over this distribution are said to have an *implicit* model. Languages where the programmer can specify the distribution are said to have an *explicit* model.

If a PGAS language allows effectively any kind of data object to be distributed, e. g., by means of remote pointers and/or remote references to data objects, we say the language supports *irregular data distribution*. If a PGAS language allows the distribution of densely packed data structures, such as arrays or matrices, in a different way than other data objects, we say the language supports *regular data distribution*. Note that for the different types of data objects (cf. regular and irregular data) implicit and explicit data distribution may coexist.

(a) (cyclic)-distribution (b) (block-cyclic)-distribution (c) (block)-distribution

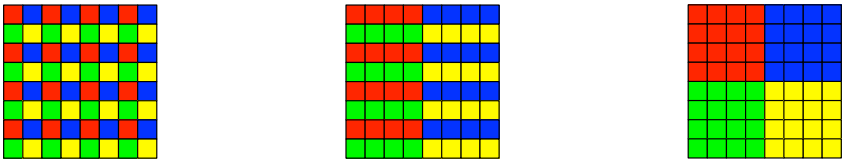
Fig. 3: The three common distributions of a 1-dimensional regular data structure over four places.

In languages with an *explicit data distribution model*, various mechanisms to control the data distribution exist. We differentiate between *ad hoc* (extensional) and *predefined* (intentional) distributions of data structures.

We observe that many PGAS languages provide a set of predefined distributions, often parameterizable by the programmer. In these languages, we identified three recurring distributions of regular data structures. While the distributions are recurring, the nomenclature is not.[6] In 1-dimensional data structures we distinguish between the *cyclic*, *block-cyclic*, and *block* distribution (cf. fig. 3). The cyclic distribution places each consecutive data element on a different place in a cyclic fashion (fig. 3a). The block distribution partitions all data elements in equally large chunks of consecutive data elements and places each chunk on a different place (fig. 3c). The block-cyclic distribution, finally, creates chunks of a parameterised size and places each consecutive chunk on a different place in a cyclic fashion (fig. 3b). In the case of multi-dimensional data structures these three distributions are *lifted*: (1) by applying a distribution per dimension (e. g., figs. 4a to 4c), (2) by applying a distribution per dimension but ignoring one (or more) of the dimensions, e. g., in fig. 4d the column-dimension is ignored (cf. *), or (3) by applying a distribution to a logically flattened data structure, e. g., fig. 4e shows a (cyclic)-distribution applied to a 2D-array as if it was a 1D-array with the rows considered as consecutive memory.

Current PGAS languages typically do not provide predefined distributions of irregular or sparse data structures, such as trees. In UPC, irregular data structures can be distributed by allocating distributed data and sharing pointers explicitly. In X10, one may also allocate objects across different places glued together using remote object references.

[6]The names for the distributions we introduce in this section are used throughout the remainder of this text, but they are not necessarily the same as the names used in the references of the discussed languages.



(a) (cyclic,cyclic)-distribution (b) (cyclic,block)-distribution (c) (block,block)-distribution

(d) (cyclic,*)-distribution (e) (cyclic)-distribution

Fig. 4: Various combinations of the common predefined distributions applied to a 2-dimensional regular data structure.

3.4. Data Access Model

The *Data Access Model* describes what data can be distributed across places, and how that data is represented, declared, and accessed. Naturally, the data access model is intertwined with the data distribution model. The main aspects for our taxonomy are how distributed data is accessed and how access to remote data is realized.

Implicit vs. Explicit Data Access. In general, remote data access to shared data can be either *implicit* or *explicit*. We say remote data access is explicit when accessing remote data requires dedicated syntax. The syntax of the remote data accesses can either imply data retrieval, e. g., in CAF (cf. the co-array syntax in fig. 5a) or a migration of the computation to the place where the data is, e. g., in X10 (cf. the `at`-statement in fig. 5b). If remote data access is syntactically transparent (e. g., UPC in fig. 5c), the PGAS language supports implicit access of remote data.

```
1 ...
2 INTEGER arr(4,M)[*]
3 ...
4 arr(i,j)[k]++
5 ...
```

(a) Data access in CAF

```
1 val d = Dist.makeUnique();
2 val arr = DistArray.make[Int](d, 0);
3 at (arr.dist[i]) {
4     arr[i]++;
5 }
```

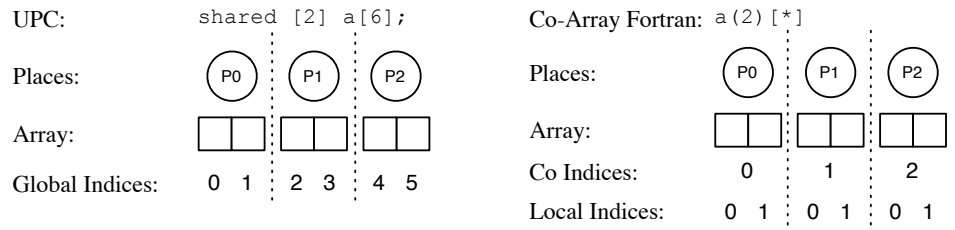
(b) Data access in X10

```
1 ...
2 shared [4] INT arr[N];
3 ...
4 arr[i]++;
5 ...
```

(c) Data access in UPC

Fig. 5: Comparing the syntax used in CAF, X10, and UPC to access remote data.

Local vs. Global Indices. When the distribution of regular data objects, such as arrays and matrices, is supported, a distinction is made between *local indices* and *global indices* to access a specific data element. If global indices are supported, each thread accesses the same memory location for a given index, regardless of the place from where the access is requested. With local indices on the other hand, the memory location associated to a given index is relative to a certain place, which is specified by an additional *place index*.



(a) UPC's indices to shared arrays are *global*. (b) CAF's indices to shared arrays are *local*.

Fig. 6: Indices in a distributed array are either local or global.

Consider fig. 6 as an illustration for these definitions. Assuming a program with three places, the UPC allocation statement in fig. 6a (`shared [2] a[6]`) allocates an array of 6 elements where each places owns two consecutive elements. Then, the expression `a[1]` accesses the second element of `a`, which is located in place P0, regardless of where the expression is executed.

Similarly, the CAF allocation statement in fig. 6b (`a(2)[*]`) allocates 6 elements, which altogether form the co-array `a`. Conversely, the expression `a(1)`, when executed in place P1, accesses the fourth element of `a`. To access the second element of `a`, the more verbose expression `a(1)[0]` should be used.

4. LANGUAGE OVERVIEW

In this section, we discuss the languages introduced in section 2. For each language, we discuss the PGAS properties based on the classification proposed in section 3. We also include some peculiarities of each language that are not necessarily directly related to the PGAS model. Finally, we exemplify some of the language's properties by means of a textbook example program that gives a general idea of the look-and-feel of the language. We conclude the overall section with a summary and an overview table of all discussed languages.

4.1. Original PGAS Languages

4.1.1. Co-Array Fortran. Co-Array Fortran (CAF) is a PGAS extension of Fortran 95 [Numrich and Reid 1998] and has since then been included in the Fortran 2008 standard [Numrich and Reid 2005].[7] CAF adheres to the SPMD execution model. It partitions the global address space into places, which are called *images* and are arranged in a user defined mesh. Each place has a unique id, which can be obtained by calling the `this_image()` built-in function.

```
1  INTEGER n
2  ...
3  n = 5
```

(a) Allocate private integer.

```
1  INTEGER n[*]
2  ...
3  n[p] = 5
```

(b) Allocate shared integer by creating a co-array.

Fig. 7: Both code fragments allocate one integer `n` for each place.

[7]Note that the CAF syntax used in this text is the syntax as presented by Numrich and Reid [1998], which differs from the syntax used in Fortran 2008.

The CAF programmer has no explicit control over the data distribution. Rather, all places own an independent instance of each declared data object: memory is organized the same[8] across all places. The data access model prohibits places to access (read or write) data objects owned by other places. To allow inter-place communication, the data access model introduces *co-arrays*. Such a co-array, accessible by all places, is a one-to-one mapping between a place and the local instance of the data object through which both read and write accesses to data objects owned by other places are allowed. A data object is only accessible via a co-array if the data object is declared with co-dimensions in square brackets immediately following the regular Fortran declaration. Figure 7 shows how to allocate an integer per place. Figure 7a shows a private one and fig. 7b one that is accessible by all places. Introducing an extension of the array notation—regular array indexing uses parentheses—to allow indexing into a co-array makes the remote data access in CAF explicit. For example, to reference the data object `n` on place `p`, one writes `n[p]` (cf. fig. 7b).

In order to achieve correct execution semantics, CAF programs have to be properly synchronized, for instance, by using a call to `sync_all()`, which prevents a thread from proceeding any further until all threads in all other places have reached this barrier. More fine-grained synchronization can be obtained by passing a vector of place ids to the `sync_all()` call, e. g., `sync_all( (/ me-1, me, me+1 /) )`.

```
1  ! global_sum
2  INTEGER :: x(n)[*]                  ! array with a co-array
3  INTEGER :: local_temp(n)            ! array without a co-array
4  INTEGER :: me, mypartner            ! indices of places
5  INTEGER :: n, bit, i, iterations    ! other variables
6
7  iterations = log2_images()
8  bit = 1
9  me = this_image(x)
10 DO i = 1,iterations
11   mypartner = xor(me, bit)
12   bit = shiftl(bit,1)
13   CALL sync_all()                   ! barrier
14   local_temp(:) = x(:)[mypartner]
15   CALL sync_all()                   ! barrier
16   x(:) = x(:) + local_temp(:)
17 ENDDO
```

Listing 1: Sum reduction of arrays in CAF.

Example. Lst. 1 shows how to calculate the sum of all the independent copies of array `x`. Conceptually this computation is similar to row-wise reducing a 2D $n \times p$ matrix with the sum operator and needs $\log_2 n$ steps (line 7). In each step a different partial sum following the butterfly pattern (line 11 and 12) is aggregated with the local value (lines 14 and 16). After the code is executed the values in `x` are equal in all places.

4.1.2. Titanium. Titanium provides an SPMD execution model with a fixed number of threads all executing the same program. Each thread is associated with a *demesne*, Titanium's term for a place. The standard library provides the methods `Ti.numProcs()`

[8]When a different amount of space is needed in different places, it is possible to declare a co-array of a derived type with a component of the type pointer array.

and `Ti.thisProc()` that respectively return the total number of places and the identifier of the local place. The identifier is an integer ranging in $[0, Ti.numProcs()[$, imposing a flat ordering on the places. Note, the ordering does not implying a difference in access cost for adjacent places or places further away from each other.

The data distribution model of Titanium differentiates between local and global references. If a reference type is explicitly specified as local, the referenced object is guaranteed to reside in direct accessible memory. Otherwise, references are implicitly global and thus might reside in any place. The data access model, on the other hand, differentiates between shared and non-shared data objects. Data objects that are shared can be accessed from any thread in any place. Non-shared data objects can only be accessed by threads that run in the same place as the data object. To specify that a reference can point to shared as well as non-shared data objects, it needs to be typed as *polyshared*. Note that the notion of local and global references is independent of the sharing type. Specific to Titanium is the absence of support for distribution of regular data, i. e., arrays or matrixes. While it supports so-called *single* values, they need to be considered as replicated objects. They are identical on all places, either by construction or by communication.

Titanium's interthread communication is designed to facilitate compile-time checks for correctness. To simplify reasoning about program correctness and compiler analyses, the offered *exchange*, *broadcast*, and *barrier* operations have to be executed from the same textual instances in all threads. The `exchange()` operation is defined on arrays and allows threads to exchange data values with each other. The `broadcast` *E* `from` *p* expression allows one thread *p* to send the value *E* to all other threads. Both operations, `exchange` and `broadcast`, act as barriers. For all three operations, it is considered an error if the control flow in any of the threads reaches another textual instance of these operations.

```
1  class ParticleSim {
2    /* ... */
3    public static void main (String[]  argv) {
4      int single allTimeStep = 0;          // single values are equal on all places
5      int single allEndTime  = 100;
6      int single myParticleCount = 100000;
7
8      Particle [1d] single [1d] allParticle =      // 2d array, 1st dimension
9          new Particle [0 : Ti.numProcs - 1][1d];  // equal on all places
10     Particle [1d] myParticle =
11         new Particle [0 : myParticleCount - 1];
12     allParticle.exchange(myParticle);            // distribute 2nd dimension
13
14     for (; allTimeStep < allEndTime; allTimeStep++) {
15       myParticle = applyForces(allParticle, myParticle);
16       Ti.barrier();
17       allParticle.exchange(myParticle);
18 } } }
```

Listing 2: Particle Simulation program in Titanium.

Example. The example in lst. 2 sketches the main loop of an all-pair particle simulation. The `allParticle` array points to the actual particles arrays, which are allocated in the corresponding thread. The `single` keyword indicates that these arrays of pointers are equal for all threads. This is realized by using the `exchange` method to initialize

them with the remote pointers coming from all threads. Each simulation time step results in a new local array of particles, to which the pointers are exchanged at the end of each iteration.

4.1.3. Unified Parallel C. Unified Parallel C (UPC) is a PGAS extension of the language C. It provides an SPMD execution model with a fixed number of places, called *threads*. Each place has an unique id, which can be obtained through the identifier **MYTHREAD**. As in all other languages, the number of places is fixed at runtime. Unique to UPC is that the number of places can be chosen at compile time (*static threads environment*) as well as at startup time (*dynamic threads environment*). When executing in the dynamic threads environment, however, certain constructs are prohibited. Declared data objects are by default *local*, which means that each place has its own independent instance of the data. When a data object is declared using the **shared** qualifier, it is accessible by all places. There exist no restrictions on the base types that can be quantified as shared, which allows the distribution of both irregular (i. e., by means of shared pointers) and regular data structures. In the latter case, the shared qualifier can be accompanied by a *layout qualifier* (`[n]`), that imposes a block-cyclic distribution with blocking factor `n` on the shared array. A block distribution can be specified using `[*]`. Memory accesses to remote elements are syntactically indistinguishable from accesses to local elements. Thus, the remote memory access is implicit.

Besides the iterative control structures inherited from the base language C (**while**, **do while**, and **for**), UPC introduces a fourth iterative control structure **upc_forall**. Inheriting roughly the same semantics as the C **for**-loop, the **upc_forall**-loop is augmented with a fourth header attribute, the *affinity*, that specifies which iteration is to be performed by which place. When the affinity is an integer expression, a place executes all iterations where the affinity evaluates to the place's identifier (modulo the number of places). When the affinity is a pointer expression, a place executes all iterations where the affinity points to a place-local memory location.

```
1  shared [N*N/THREADS] uint8_t orig[N][N], edge[N][N];
2  int Sobel() {
3    int i,j,d1,d2;
4    double magnitude;
5    //          init   cond    step   affinity
6    upc_forall(i=1;  i<N-1; i++;  &edge[i][0]) {
7      for(j=1; j<N-1; j++) {
8        d1  =  (int) orig[i-1][j+1] - orig[i-1][j-1];
9        d1 += ((int) orig[i  ][j+1] - orig[i  ][j-1]) << 1;
10       d1 +=  (int) orig[i+1][j+1] - orig[i+1][j-1];
11       d2  =  (int) orig[i-1][j-1] - orig[i+1][j-1];
12       d2 += ((int) orig[i-1][j  ] - orig[i+1][j  ]) << 1;
13       d2 +=  (int) orig[i-1][j+1] - orig[i+1][j+1];
14       magnitude = sqrt(d1*d1+d2*d2);
15       edge[i][j] = magnitude>255 ? 255 : (uint8_t)magnitude;
16     }
17   }
18   if (MYTHREAD == 0)
19     printf("DONE\n");
20
21   return 0;
22 }
```

Listing 3: Parallel edge detection using Sobel operators in UPC.

Example. Lst. 3 demonstrates how to implement parallel edge detection using Sobel operators in UPC. The example is adapted from Chauvin et al. [2005]. The program is an example of a stencil computation, i. e., the values of the array `edge` are computed based on a fixed pattern of values of the array `orig`. A serial C version could be implemented using nested for-loops iterating over the elements in `edge`. This UPC program is identical, except that the outer loop is replaced by a parallel **upc_forall** loop and both `orig` and `edge` are qualified as **shared** with a block distribution, homogeneous with respect to the column dimension.[9] Therefore, and because the affinity is expressed per row (cf. `&edge[i][0]` on line 6), each place requires mostly local data. Remote accesses into the previous or next place are only needed sporadically.

4.2. HPCS PGAS Languages

4.2.1. Chapel. Chapel is a PGAS language that offers both *task parallelism* and *data parallelism*, without prioritizing one over the other. The stylized statements for launching and synchronizing parallel tasks, e. g., **cobegin** or **coforall**, place Chapel in the category of languages with an *APGAS* execution model, while the data parallel iteration constructs introduce parallelism *implicitly*. Both parallel execution models share the concept of a *locale*, Chapel's terminology for place. These places are accessible as first class citizens through the built-in array `Locales`. Intrinsically, the places are represented as a flat ordered set, but Chapel allows to lift the one-dimensional representation to an n-dimensional mesh trough the use of the `reshape` function. The *domain* construct represents a set of indices that define the size and shape of an array (or iteration space). The actual distribution is achieved by defining a mapping between the indices in a domain and the places, which can be expressed by defining a *domain map*. By default, Chapel provides a set of predefined domain maps.

The distribution of *irregular* data is expressed by the programmer by allocating data objects in the desired place. The code snippet in fig. 8a shows how the **on**-statement is used to allocate an object `c` in the i^{th} place. Besides the ad hoc distribution of irregular data, Chapel also supports the definition of domain maps for *irregular* domains (e. g., associative, sparse, and unstructured domains).

Dereferencing an object that resides in a different place introduces an implicit remote data access. However, remote data accesses can also be made explicit by moving the *computation* to the place owning a certain data object. The latter is shown in fig. 8b, where the **on**-statement moves the computation to the place where `c` resides.

```
1 on Locales[i] {
2     c = new Object();
3 }
```

(a) Ad hoc distribution of irregular data in Chapel using the **on**-statement.

```
1 on c do {
2     c.doSomething();
3 }
```

(b) The **on**-statement used for explicit access of remote data.

Fig. 8: The **on**-statement moves computation to a potentially different place.

Data parallel example. To illustrate the concepts of data parallelism in Chapel, we show an implementation of the Jacobi iteration over a square 2D grid, adapted from the Chapel tutorials.[10]

[9] Note that instead of the declaration on line 1 of lst. 3 also the following declaration, with syntactic sugar, could have been used: **shared** `[*] uint8_t orig[N][N], edge[N][N];`.

[10] *Chapel Tutorials*, Cray, access date: 15 May 2013 http://chapel.cray.com/tutorials/

```
const BigD = {0..n+1, 0..n+1} dmapped Block(boundingBox=[0..n+1, 0..n+1]),
          D: subdomain(BigD) = {1..n, 1..n};
var A, Temp: [BigD] real;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;
  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);
```

Listing 4: Jacobi iteration example in Chapel (data parallel).

The code assumes a grid size n by n and defines a domain `BigD`, which spans the entire grid (ranging in both dimensions from 0 to $n + 1$), and a subdomain `D`, which contains only the inner points of the grid (ranging in both dimension from 1 to n). The data items in the domain `BigD` are distributed across the locales using the `Block` domain map, which realizes a block distribution. The code also defines two arrays, `A` and `Temp`, with floating-point values (datatype **real**) over the entire `BigD` grid domain.

The Jacobi iteration starts from an initial set of values in array `A` and repeatedly computes the average of each grid point's neighbors. The **forall** construct that is used to visit all the grid points is implicitly data-parallel, and it ensures that the operation for grid point i, j is executed by the owning locale, thus minimizing data movement.

Task parallel example. To illustrate the concepts for task parallelism in Chapel, we show a parallel implementation of the Quicksort algorithm, adapted from the Chapel tutorials. The following procedure sorts elements of an array `arr` with an arbitrary domain represented by parameter `D`.

```
proc quickSort(arr: [?D],
               thresh = log2(here.numCores()), depth = 0,
               low: int = D.low, high: int = D.high) {
  if high - low < 8 {
    bubbleSort(arr, low, high);
  } else {
    const pivotVal = findPivot(arr, low, high);
    const pivotLoc = partition(arr, low, high, pivotVal);
    serial(depth >= thresh) do cobegin {
      quickSort(arr, thresh, depth+1, low, pivotLoc-1);
      quickSort(arr, thresh, depth+1, pivotLoc+1, high);
} } }
```

Listing 5: Parallel Quicksort example in Chapel (task parallel).

For arrays with 8 or more elements, the array is partitioned around a pivot element and the Quicksort algorithm is applied recursively on the two sub-arrays. The recursive invocations of the `quicksort` procedure are launched in parallel by employing a **cobegin** block, which spawns a task for each of the contained statements and awaits their completion. The **serial** modifier indicates that parallel execution is no longer employed beyond a certain threshold depth in the recursion hierarchy, because it is assumed that sufficient parallelism has been exposed.

4.2.2. X10. X10 is a class-based object-oriented programming language based on a subset of sequential Java 1.4. The parallel subset of X10 introduces new constructs and built-in primitive types and explicitly assumes a partitioned global address space.

An X10 program starts with a single thread of control. Such a thread of control is called an *activity* and it runs in a fixed place, i. e., it cannot migrate between places. To start computation in other places an activity can asynchronously spawn other activities in remote or local places with the `async` construct (APGAS Execution Model). To synchronize upon termination the `finish` construct can be used. The partitions of the global address space are reified as *places*, allowing the programmer to differentiate between what is local and what is remote. Even though the places in an X10 program do not necessarily map onto physical processors or processing cores, the number of places is fixed and known at program start up. A single object is allocated at a single place. Regular data are allocated across places, the distribution is specified by a *distribution object*, a mapping between array indices and places. X10 allows only for explicit access of remote data by moving the computation to the data. This is shown in fig. 5. Attempting to access a data object from any other place results in a `BadPlaceException`.

```
1  val initializer = (i:Point) => {
2    val r = new Random();
3    var local_result:double = 0.0D;
4    for (c in 1..N) {
5      val x = r.nextDouble();
6      val y = r.nextDouble();
7      if ((x*x + y*y) <= 1.0)
8        local_result++;
9    }
10   local_result
11 };
12 val result_array = DistArray.make[Double](Dist.makeUnique(), initializer);
13 val sum_reducer = (x:Double, y:Double) => { x + y };
14 val pi = 4 * result_array.reduce(sum_reducer, 0.0) / (N * Place.MAX_PLACES);
```

Listing 6: Estimating π using Monte Carlo method in X10.

Example. The example in lst. 6 illustrates both *initializers* and *reducers* and how they can be used to compute an estimation of π using Monte Carlo method, i. e., π can be approximated by computing the chance that a random point in a square is also within the largest enclosed circle. The function stored in `initializer` computes this for `N` random points (cf. lines 1–11). To distribute the work, first an evenly distributed array is created with as many elements as there are places, cf. `Dist.makeUnique()` on line 12. Moreover, each place computes the initial value for one element by calling `initializer`, the second argument in the constructor of the distributed array (line 12). Finally, to compute π, the sum of all local results is needed. Thus, a sum-function (cf. line 13) is used to reduce all values in `result_array` to one global sum (cf. line 14), from there pi can easily be computed.

4.2.3. Fortress. Fortress supports two parallel execution models. Based on the **spawn** keyword, it provides an APGAS model. In addition, it specifies constructs that introduce implicit parallelism. These *potentially parallel constructs* are tuples, parallel blocks (i. e., **do** ... `also` ... **do**), generators and loops over generators, functions, and operators (i. e., Σ and Π as they are known from mathematics). The locality of a data object is governed by *regions*, a hierarchical tree-like representation of the underlying hardware, where each node of a region maps to a place.[11] The actual dis-

[11]Note, the Fortress specifications states that the initial implementation assumed a shared memory machine with a single global region.[Allen et al. 2008]

tribution of the data over the regions is governed by *distribution data structures*. Fortress' standard library provides a number of default distributions. An array for instance is distributed depending on its size, and on the size and locality characteristics of the machine running the program. For APGAS computations, the **spawn** statement allows also for the movement of computation to the data, e. g., the statement **spawn** `x.region` **do** `f(x)` **end** computes `f(x)` in the place where `x` resides.

```
1  var a : RR64 = 0.0
2  var b : RR64 = 0.0
3  var c : RR64 = 0.0
4
5  DELTA = b^2 - 4 a c
6  x_1 = (-b - SQRT DELTA)/(2 a)
7  x_2 = (-b + SQRT DELTA)/(2 a)
```

(a) Small example program in Fortress without unicode characters.

$$\texttt{var}\ a : \mathbb{R}64 = 0.0$$
$$\texttt{var}\ b : \mathbb{R}64 = 0.0$$
$$\texttt{var}\ c : \mathbb{R}64 = 0.0$$
$$\Delta = b^2 - 4\,a\,c$$
$$\mathtt{x}_1 = \frac{-b - \sqrt{\Delta}}{2\,a}$$
$$\mathtt{x}_2 = \frac{-b + \sqrt{\Delta}}{2\,a}$$

(b) Small example program in Fortress that supports unicode characters.

Fig. 9: Example program in Fortress to show the difference between formatted and unformatted code.

In the context of DARPA's HPCS program, a primary focus in the development of Fortress was improved productivity. Facilitating the expression of mathematical and physical formulas is one avenue explored in Fortress towards this goal. To this end, the Fortress syntax allows unicode characters—such as the sum-operator Σ—for both operators and variables. Further, Fortress supports the overloading of the white space character, which allows one to express the juxtaposition, a common mathematical notation to denote a product. Moreover, the static type system supports compile-time checks on physical units and dimensions. Finally, the **atomic** construct allows for the enforcement of correct parallel semantics, especially in implicit parallel constructs.

Example. The example in lst. 7, as taken from the Project Fortress website,[12] estimates π using the method of Buffon. π is estimated using *repeated* random experiments similar to Monte Carlo method. The repetition is obtained by a for-loop of 3000 iterations on line 7. The result is computed by dividing the number of *hits* by the number of experiments, represented as the variables `hits` and `n` respectively. These variables might be subjected to race conditions since **for**-loops are inherently parallel in Fortress. Therefore, it is the programmer's responsibility to wrap racing memory accesses in an **atomic** construct, as is illustrated on line 16.

4.3. Retrospective PGAS Languages

4.3.1. High Performance Fortran. High-Performance Fortran (HPF) [High Performance Fortran Forum 1993] is a *data-parallel language* for distributed parallel computers unifying concepts from various older data-parallel languages [Callahan and Kennedy 1988; Zima et al. 1988; Thinking Machines Corporation 1991]. An HPF program appears to have a single thread of control. However, `for`-statements and argument evaluation happens implicitly in parallel. HPF models its places as a rectilinear mesh of what they call *abstract processors*. When allocating (multi dimensional) arrays, the HPF directives **ALIGN** and **DISTRIBUTE** hint to the compiler how to distribute the

[12] *Project Fortress: source code*, Oracle, access date: 03 March 2013 http://java.net/projects/projectfortress/sources/

```
needleLength = 20
numRows = 10
tableHeight = needleLength numRows
var hits : RR64 = 0.0
var n : RR64 = 0.0

for i <- 1#3000 do
  delta_X = random(2.0) - 1
  delta_Y = random(2.0) - 1
  rsq = delta_X^2 + delta_Y^2
  if 0 < rsq < 1 then
    y1 = tableHeight random(1.0)
    y2 = y1 + needleLength (delta_Y / sqrt(rsq))
    (y_L, y_H) = (y1 MIN y2, y1 MAX y2)
    if ceiling(y_L/needleLength) = floor(y_H/needleLength) then
        atomic do hits += 1.0 end
    end
    atomic do n += 1.0 end
  end
end
probability = hits/n
pi_est = 2.0/probability
```

Listing 7: Estimating π using Buffon's method in Fortress.

elements of the array over the different places. The **DISTRIBUTE** directive can take on one of the following values per dimension: **BLOCK** to distribute equal consecutive blocks over the places, **CYCLIC** to distribute single elements over the places in a round-robin fashion, or $*$, which means no distribution. Additionally, augmenting the distribute directive with a `PROCESSORS` directive refines a distribution such that it only considers a subset of places. In addition to the directives for specifying data layout, HPF provides a library of special global operations such as sum reductions, gather and scatter operations, etc. [High Performance Fortran Forum 1993]. Here, the implicit parallelism adheres to the "owner computes rule", which states that calculations are carried out in the places that own the data objects involved. If a calculation uses data objects from different places, the compiler implicitly generates communication. Although the HPF specification does not require implementing HPF on top of MPI, most existing approaches compile to MPI [Kennedy et al. 2007].

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
         + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
```

Listing 8: Simple relaxation loop in HPF.

Example. As an example, lst. 8 discusses the code for a relaxation loop taken from Kennedy et al. [2007]. HPF code mostly looks like Fortran 90 code, except that the programmer needs to provide a specification of the data layout via **DISTRIBUTE**. In lst. 8, the array `A` is distributed row-wise among the available processors: **BLOCK** distribution along the rows and no distribution along the columns ($*$). The **ALIGN**

directive specifies element-wise matching between different arrays. Hence in our example, array `B` is distributed among the same places as array `A`. Once the data layout is defined, HPF relies on the implicit parallelism provided by the **DO**-loops. In this example, the arrays `A` and `B` are distributed across the available places and each place executes the **DO** loops on *its* portions of the arrays.

4.3.2. ZPL. ZPL is an array programming language that gives the programmer a global view of the computation where all parallelism is implicit. At start-up, one can specify the processor set size (i. e., the number of places, p) and optionally also a (set of) *processor grids*. Such a processor grid is a mesh-shaped view on the set of places, and can be described as $p_1 \times p_2 \times \cdots \times p_d$, where this product equals the number of processors and where d is the dimension (e. g., `Life -p8 -g2x4` would execute the Game of Life example program (lst. 9) on 8 processors arranged in a 2 by 4 mesh). These places are not reified in the ZPL program. However, there exist extensions of ZPL where places are more explicit and are called *locales* [Deitz 2005]. A *region* is a set of indices that can be used to declare a *multi-dimensional array*. Based on such a region, ZPL *implicitly* distributes the regular data structures in a grid-aligned manner. Further, the ZPL specification states that if two regions interact, they must have the same distribution. Programs that utilize regions with different ranks or different scales may use multiple processor grids to represent a different view of the processor set for each computational domain. The communication required by each expression is clearly reflected in its syntax, this is referred to as the *WYSIWYG (what you see is what you get) performance model*, because a program's communication requirements can be determined simply by examining its array operators.

ZPL features a rich collection of operators that, given a direction `D` and a region `R`, form a new region. A *direction* is an offset vector that defines a "shift"in a certain direction. Examples are `D` **in** `R`, `D` **of** `R`, `D` **at** `R`, and `R` **by** `D`. Further, ZPL offers a reduction operator `<<` which reduces a region of values into a smaller region, e. g., `[TopRow] A := +<<[R] B;` sums each column of B and stores the sum in the first row of A. The inverse of reducing is called flooding. The flooding operator `>>` replicates a certain data value in all entries of a given parallel array given some region specifier, e. g., `[R] A := >>[TopRow] B` assigns the first row of B to each row of A.

```
1  program Life;
2    config const n : integer
3    region R = [1..n, 1..n];
4    direction nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
5              w  = [ 0, -1];               e  = [ 0, 1];
6              sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];
7  var  TW : [R] boolean;
8       NN : [R] sbyte;
9  procedure Life();
10 begin -- Initialize the world
11   [R] repeat
12           NN := TW@^nw + TW@^no + TW@^ne
13               + TW@^w  +          TW^@e
14               + TW@^sw + TW@^so + TW@^se;
15           TW := (TW & NN = 2) | (NN = 3)
16        until !(|<< TW);
17 end;
```

Listing 9: Conway's Game of Life in ZPL.

Example. Lst. 9 shows an implementation of Conway's Game of Life in ZPL, an example taken from ZPL Research Group [2003]. The Game of Life involves a two-dimensional grid of booleans that is recomputed in every "generation". A cell in the grid is 'dead' or 'alive'. A living cell dies if it has less than two or more than three living neighbors. A dead cell becomes alive if it has exactly three living neighbors. The program starts with a number of Pascal-styled declarations. The region `R` is used to declare the arrays `TW` (the world) and `NN` (number of neighbors), of respective types `boolean` and `sbyte`. Statements in ZPL are controlled by a region specifier. Instead of specifying the region of each statement in the loop, a single specification `[R]` indicates that all statements of the loop are executed with respect to that part of the parallel arrays. The operator `@^` (pronounced: wrap-at) shifts the parallel array in the corresponding direction. The (missing) boundary values are obtained by wrapping the array. Notice that the `+` operator is available on arrays: all elements of the argument parallel arrays are added in a pairwise fashion. In this example, every element of the `NN` array contains the number of living neighbors of the corresponding element in the `TW` array. The condition of the loop shows the usage of a reduction operator: applying the `|`-operator to all values in `TW` simultaneously, reduces `TW` to a single boolean value.

4.3.3. Global Arrays Toolkit. Global Arrays Toolkit (GA) is a library that provides a PGAS interface for SPMD programs. A thread in a GA program can query for its identifier or the total number of spawned threads by calling the functions `GA_Nodeid` and `GA_Nnodes` respectively. Each thread maps to what we define as a place. The main data objects in the library are *global arrays*, distributed numerical multi-dimensional arrays that are shared by all threads. Creating a new global array requires a call to `NGA_Create` by all threads, otherwise the code hangs. The distribution of a global array over the different places can be specified by the programmer explicitly. The library provides two main types of distributions. On the one hand there is the so called *regular distribution* which is a block-distribution (i. e., approximately the same number of elements per place, see section 3.3) parameterized with a *minimal block-size* for each dimension. On the other hand there is also support for so called *irregular* distributions, where the number of elements assigned to a place may vary. Both distributions are non-cyclic in the sense that each place owns at most one partition of the global array. To obtain a round-robin block-cyclic distribution, however, a call to the `GA_Set_block_cyclic` function is needed. This intrinsic function takes the *exact block-sizes* per dimension and a description of a rectilinear mesh of the set of places (optionally) as parameters. For applications that exploit domain decomposition, e. g., stencil computations, GA introduces *ghost cells* as an integral part of the distribution. Ghost cells are a common technique to reduce communication in these computations. It is clear that global arrays can have a wide range of different predefined distributions. Therefore, it is possible to query for the distribution of the global array using the `NGA_Distribution` function.

The global array library provides the `NGA_Access` function for accessing local data objects. Remote data objects can be accessed using calls to the one-sided comminication functions (i. e., `NGA_Get`, `NGA_Put` and `NGA_Acc`). The algorithm implementor must be aware that local data access is cheaper than remote data access, and should try to optimize data locality. Most other functions in the API (those starting with prefix `NGA_`) are multi-dimensional, meaning that they can be used for global arrays of any dimension. Many of those functions specify collective operations on a (hyper)rectangular region of a global array, called a *patch*.

Example. The code in lst. 10 implements matrix-vector multiplication $A\,\vec{x} = \vec{b}$ where matrix A is stored as a two-dimensional global array, and vectors $\vec{x}, \vec{b}$ are both stored as

```
1  void matvec_multiply(int g_A, int g_x, int g_b) {
2    int    i, j, lo[2], hi[2], ld[1];
3    double *a, *x, *b, scale;
4    int    me = GA_Nodeid(), nprocs = GA_Nnodes();
5    GA_Sync();
6    NGA_Distribution(g_A, me, lo, hi); // Query distribution for local part of g_A
7    NGA_Access(g_A, lo, hi, a, ld);
8    x = malloc((hi[1] - lo[1]) * sizeof(double));
9    NGA_Get(g_x, &lo[1], &hi[1], x, NULL); // Retrieve corresponding part of g_x
10
11   // Calculate local contribution to g_b
12   b = malloc((hi[0]-lo[0]) * sizeof(double));
13   for(i = 0; i < hi[0]-lo[0]; i++) {
14     b[i] = 0.0;
15     for(j = 0; j < hi[1]-lo[1]; j++)
16       b[i] += a[i*ld[0] + j] * x[j];
17   }
18
19   GA_Zero(g_b); // Collectively initialize g_b
20   scale = 1.0;
21
22   NGA_Acc(g_b, &lo[0], &hi[0], b, NULL, &scale); // Update g_b
23   NGA_Release(g_A, lo, hi); // Release resources
24 }
```

Listing 10: Matrix-vector multiplication in GA.

one-dimensional global arrays. The distribution of the global arrays is not unspecified, but since most data elements reside in matrix A, the computation occurs local to this data. The function `matvec_multiply` is executed collectively, and each process retrieves the indices of its local section of matrix A through a call to `NGA_Distribution`. Then, it makes access to the local section available through pointer `a` and stride `ld[0]` with a call to `NGA_Access`. The corresponding parts of $\vec{x}$ and $\vec{b}$ are accessed remotely. First, the relevant portion of $\vec{x}$ is copied to a local buffer using a call to `NGA_Get`. The local part of the multiplication is then executed, storing the result in a local buffer of appropriate size. Finally, the global array for $\vec{b}$ is written, by first initializing its elements to zero, and then adding the contribution from each process at the appropriate place using `NGA_Acc`. Since accumulation is an atomic operation, there is no race when multiple processes try to write their contribution to overlapping parts of the global array. The collective matrix-vector multiplication starts and ends with a call to `GA_Sync`. These barriers are needed to ensure that remote data are ready before they are read with `NGA_Get` and to ensure that all remote writes using `NGA_Acc` have ended. Further, the collective operation `GA_Zero` synchronizes all processes before and after setting the local portion of the global array to zero.

4.4. Recent Approaches

4.4.1. XCalableMP. XCalableMP (XMP) is a PGAS extension for C and Fortran, closely resembling the more widely known OpenMP compiler extensions. Like OpenMP, XCalableMP features a set of compiler directives or pragmas that alter the semantics of a sequential C or Fortran program. For the C extension for instance, the pragmas start with **#pragma** `xmp`. XCalableMP was inspired by HPF and Co-Array Fortran. An XCalableMP program consists of a set of threads each operating in parallel, i. e., SPMD. Like other PGAS systems with an SPMD execution model, XCalableMP provides directives for parallel **for**-loops and barrier synchronization. In XCalableMP, places are referred to as *nodes*.

XCalableMP's data distribution model focuses on the distribution of regular data, i. e., arrays. It has dedicated support for specifying the distribution of array elements over the places. This is a three-step process: (1) One must first define a region or *template* with a name and a size in each dimension, e. g., **#pragma** `xmp template my_template(4, 5)`. (2) One then refines the template by specifying a predefined distribution for each dimension and a target set of places, e. g., **#pragma** `xmp distribute my_template(block, cyclic(2)) on nodes`. This example uses a block and a block-cyclic distribution with blocking factor 2, in the respective dimensions. Uneven distributions are also supported, these require a vector with different block sizes within a dimension. (3) Finally, one must use the `align` directive to specify the correspondence between a template and an array, e. g., **#pragma** `xmp align a[i][j] with t(i,j)`. Remote data access is explicit, by means of an extension of the array indexing notation. For example, to reference array element `a[i]` on node `n`, one writes `a[i]:[n]`.

```
1  int a[YMAX][XMAX];
2
3  #pragma xmp nodes p(*)
4  #pragma xmp template t(YMAX)          // define region
5  #pragma xmp distribute t(block) on p // partition block-wise
6  #pragma xmp align a[i][*] with t(i)  // align i-th row with
7                                        // i-th region
8  main() {
9    int i, j, res;
10   res = 0;
11   #pragma xmp loop on t(i) reduction(+:res)
12   for(i = 0; i < YMAX; i++) {
13     for(j = 0; j < XMAX; j++) {
14       a[i][j] = f(i, j);
15       res += a[i][j];
16     }
17   }
18 }
```

Listing 11: Map and reduce in XCalableMP.

Example. In the example in lst. 11, the sum of applying a function `f` on all elements of a 2D matrix is calculated in parallel. The variable `res` is identified as a reduction variable that accumulates the results computed in parallel in a thread-safe way by summing the partial results.

4.5. Summary

We discussed 10 PGAS languages developed over the last 20 years. While all these languages conform to the definition given in section 3, each of these languages instantiate the four models differently, i. e., parallel execution model, places model, data distribution model, and data access model.

The parallel execution model specifies how parallelism is achieved. All of the considered languages use either SPMD, asynchronous lightweight threads, i. e., APGAS, or implicit parallelism. Furthermore, Chapel and Fortress combine APGAS with implicit parallelism.

Considering the historic perspective, there seems to be a shift away from the SPMD-style of parallelism towards more flexible and more fine-grained asynchronous programming models. Since developer productivity is a key goal of PGAS language de-

Language	Parallel Execution	Topology	Data Distribution	Distributed Data	Remote Access	Array Indexing
Original PGAS languages						
CAF	SPMD	User defined mesh	Implicit	Regular	Explicit	Local
Titanium	SPMD	Flat ordered set	Explicit	Irregular	Expl. + Impl.	not applicable
UPC	SPMD	Flat ordered set	Explicit	Reg. + Irreg.	Implicit	Global
HPCS PGAS languages						
Chapel	APGAS + Impl.	User defined mesh	Explicit	Reg. + Irreg.	Expl. + Impl.	Global
X10	APGAS	Flat ordered set	Explicit	Reg. + Irreg.	Explicit	Global
Fortress	APGAS + Impl.	Hierarchical	Explicit	Reg. + Irreg.	Expl. + Impl.	Global
Retrospective PGAS languages						
HPF	Implicit	User defined mesh	Explicit	Regular	Implicit	Global
ZPL	Implicit	User defined mesh	Implicit	Regular	Explicit	Global
GA	SPMD	Flat ordered set	Explicit	Regular	Explicit	Global
Recent PGAS languages						
XCalableMP	SPMD	Flat ordered set	Explicit	Regular	Explicit	Global

Table I: Classification of all PGAS languages discussed in this survey.

sign, this paradigm shift is the logical consequence of expressing parallelism from the perspective of the problem, instead of expressing parallelism from the perspective of the target hardware, as is the case for classic SPMD where a single thread is used for one node. Another important aspect is the increasing complexity of hardware architectures: more hierarchical and heterogenous. To address this trend, approaches that move performance-related aspects such as scheduling and computation placement to the runtime gain relevance. Together with the increased flexibility and runtime techniques such as work-stealing, more problems can be expressed efficiently, and thus, become solvable in the context of HPC.

The places model captures the very nature of what it entails to be a PGAS language. It specifies how the conceptually shared address space is partitioned and allows to anticipate the cost of accessing remote data.

Currently, all PGAS languages differentiate between local and remote data access and associate those with cheap access and expensive access respectively, i. e., a two-level cost function. Correlated with the lack of richer cost functions is the sparseness of existing inter-place topologies, i. e., how places relate to each other. Most PGAS languages assign to each place a unique identifier resulting the *flat ordered set* topology. While the places themselves usually reflect the partitioned characteristics of the underlying hardware, the flat ordering does not add extra information, e. g., which places belong to the same core, chip, rack, etc. The rectilinear mesh topology, the multi-dimensional extension of the flat ordered set, does not add such information either, but only improves upon the expressibility of distributions of regular data structures of equal dimensions. The first language to adopt the idea that the places interconnection topology should reflect the underlying hardware in more detail, was Fortress via its region construct. The Fortress specification foresees a hierarchical topology to connect its places, but no implementation exists today. Yan et al. [2009] extended the work on X10 with *Hierarchical Place Trees*, and Chapel added initial support for specifying hierarchical locales in version 1.8[13]. From a performance perspective, it might be desirable to enrich cost functions and expose more expressive topologies. However, this could also have a negative impact on portability of programs between different machines with respect to their performance characteristics.

The cost function, as defined in section 3 is not included in table I, because all the PGAS approaches developed hitherto adhere to a 2-level cost function, i. e., they only differentiate between cheap accesses and expensive accesses. Research towards multi-level cost functions and more complex topologies is ongoing [Yan et al. 2009]. Prospective changes on the hardware level that influence the NUMA effects can expedite research towards more expressive topologies and richer cost functions.

The data distribution model describes how data objects can be distributed across the available places. Distribution is either achieved by explicit intervention of the programmer or implicitly, for instance as part of a declaration. This survey shows that both approaches are common in PGAS languages and that they are not mutually exclusive.

All PGAS languages focus on the distribution of dense regular data objects, i. e., (multi-dimensional) arrays. Moreover, most languages adopt the concept of block-cyclic distribution for such data objects, and allow the programmer to explicitly specify a suitable distribution for each regular data object. Less regular distributions exist, i. e., uneven distributions, while arbitrary distributions of arrays are not intrinsically supported by any of the PGAS languages except Chapel. Chapel introduces domain maps, which allow developers to specify the implementation of any array, including the dis-

[13] *Chapel*, Crey, access date: October 17, 2013 http://chapel.cray.com/

tribution over places. The distribution of irregular data objects, e. g., pointers and references that form a complex data structure such as a graph, is only expressible in an ad hoc manner. Thus, the data objects have to be distributed explicitly one by one. Again, Chapel's domain maps allow developers to modularize and re-use such an ad hoc distribution.

Future avenues of research could consider intentional approaches for the distribution of sparse data objects, possibly introducing dynamic distributions that change during the execution of the application in order to have the best fit between computation and data. Such dynamic distributions can be interesting when the number of places changes during the computation.

The data access model specifies how to access data residing in remote places, and how developers can distinguish local and remote accesses. Most PGAS languages opt to make remote accesses syntactically explicit. In UPC, however, remote data accesses is always implicit, i. e., syntactically transparent. Hybrid interpretations of the data access model can be found as well.

The benefit of languages with explicit remote access is that introducing potentially expensive operations in a program requires a special effort by the programmer, moreover, an excessive use of syntactically conspicuous and expensive operations, can be spotted on sight. However, it is an additional cognitive burden for the programmer, even for performance insensitive program parts, which is avoided in languages with syntactically transparent approaches. For these languages, tools such as IDEs or typecheckers could provide similar feedback to highlight performance sensitive operations on demand. Since tools might provide similar clues to the programmer, it remains to mention that implicit solutions typically also have better reuse properties. While an algorithm in a language with explicit remote access might be easier to tune for performance, the same piece of code can typically not be used in a purely local or sequential setting so that reuse can become an issue.

Assessing the productivity gains of PGAS languages and the impact of the aforementioned aspects remain however an open research issue. Recently, Richards et al. [2014] reported on the benefits of X10 comparing it to the use of C and MPI. While they find strong indications that the language constructions of X10 improve productivity, it still remains open how the different design choices within the field of PGAS languages affect programmer productivity.

5. CONCLUSION

The PGAS programming model is a parallel programming model for HPC that aims at improving both productivity and performance. Productivity is improved by providing a shared address space model that is supposed to be more suitable for the majority of HPC applications compared to the prevalent message-passing model. To achieve performance on machines with non-uniform memory accesses, the difference in cost of accessing local and remote data must be taken into account. Therefore, PGAS languages make the cost of accessing remote data explicit. This approach positions PGAS somewhere in the middle of the continuum between shared memory, where communication is completely implicit, and message-passing, where communication is explicit.

For the conceptual classification of PGAS languages, we identified four axes: the *parallel execution model*, the *places model*, the *data distribution model*, and the *data access model*.

For the parallel execution model, we distinguished between languages using the SPMD model, the APGAS model, and implicit parallelism. The places model specifies conceptually how the shared address space is partitioned and what the access cost between different places is. Here we found that a simple distinction between local

and remote accesses is the sole variant explored by today's PGAS languages. However, with the rising complexity of HPC systems, we feel the need for more elaborate models that can guide developers to implement efficient programs on these systems. The data distribution model of a language describes how data objects can be laid out across the available places. Current PGAS languages focus on mechanisms to distributed regular, dense data objects such as arrays. In the light of efforts such as the Graph 500 list,[14] PGAS languages miss the opportunity to support data intensive problems well that require irregular data accesses. Besides some efforts in Chapel, intrinsic programming language support for the distribution of sparse and irregular data objects remains an open issue. The data access model on the other hand has been well explored by PGAS languages. The main distinctions found here is whether the access cost is part of an operation or part of a type or property of a variable. From our perspective, the main questions here are in the field of software engineering and the impact of these different approaches on modularity, maintainability, and portability of the resulting code with respect to preserving performance.

To conclude, we see the PGAS languages as the first group of languages that approach performance as well as productivity as the main goal. With the rising complexity of commodity hardware systems and the multicore revolution in mind, many of the explored ideas are also relevant for general purpose languages in order to enable programmers to optimize program parts that are performance sensitive and require predictable behavior. Thus, we expect that these PGAS languages will influence mainstream programming models in one way or another and that these programming mechanisms will not remain confined to the HPC world.

ACKNOWLEDGMENTS

The authors would like to thank Charlotte Herzeel for her participation in the discussions of PGAS languages, especially for her input on the Fortran-related approaches. Furthermore, we would like to thank the anonymous reviewers for their extensive and insightful feedback, which improved the overall paper and especially the finer technical details.

REFERENCES

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. 2008. *The Fortress Language Specification*. Technical Report. Sun Microsystems, Inc. 262 pages. Version 1.0.

Dan Bonachea. 2002. *GASNet Specification, v1.1.* Technical Report. University of California at Berkeley, Berkeley, CA, USA.

Dan Bonachea and Jason Duell. 2004. Problems with Using MPI 1.1 and 2.0 As Compilation Targets for Parallel Language Implementations. *Int. J. High Perform. Comput. Netw.* 1, 1-3 (Aug. 2004), 91–99. DOI:http://dx.doi.org/10.1504/IJHPCN.2004.007569

Eugene D. Brooks III, Brent C. Gorda, and Karen H. Warren. 1992. The Parallel C Preprocessor. *Scientific Programming* 1, 1 (Jan. 1992), 79–89.

David Callahan and Ken Kennedy. 1988. Compiling Programs for Distributed-Memory Multiprocessors. *The Journal of Supercomputing* 2, 2 (1988), 151–169. DOI:http://dx.doi.org/10.1007/BF00128175

William W. Carlson and Jesse M. Draper. 1995. Distributed data access in AC. In *Proceedings of the 5th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '95)*. ACM, New York, NY, USA, 39–47.

[14]http://www.graph500.org/

Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10)*. ACM, Article 2, 3 pages. DOI:http://dx.doi.org/10.1145/2020373.2020375

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioğlu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. 20th annual ACM SIGPLAN conf. on Object-oriented programming, systems, languages, and applications (OOPSLA'05)*. ACM, New York, NY, USA, 519–538.

Sébastien Chauvin, Proshanta Saha, François Cantonnet, Smita Annareddy, and Tarek El-Ghazawi. 2005. *UPC Manual* (v1.2 ed.). The George Washington University.

Cray Inc. 1999. *Cray T3E C and C++ Optimization Guide*. Technical Report 0042178002. http://docs.cray.com/books/004-2178-002/004-2178-002-manual.pdf

David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. 1993. Parallel Programming in Split-C. In *Proceedings of the Supercomputing '93 Conference*. ACM, Portland, OR, 262–273.

Steven J. Deitz. 2005. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. Dissertation. University of Washington.

Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. 2005. *UPC: Distributed Shared Memory Programming*. Wiley, Hoboken, NJ. 252 pages.

Al Geist, William Gropp, Steven H. Lederman, Andrew Lumsdaine, Ewing L. Lusk, William Saphir, Tony Skjellum, and Marc Snir. 1996. MPI-2: Extending the Message-Passing Interface. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*. Springer-Verlag, London, UK, 128–135.

High Performance Fortran Forum. 1993. *High Performance Fortran Language Specification, version 1.0*. Technical Report CRPC-TR92225. Center for Research on Parallel Computation, Rice University, Houston, Texas.

Ken Kennedy, Charles Koelbel, and Hans Zima. 2007. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*. ACM, New York, NY, USA, 7–1–7–22. DOI:http://dx.doi.org/10.1145/1238844.1238851

Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. 1994. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, USA.

Jinpil Lee and Mitsuhisa Sato. 2010. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPPW '10)*. IEEE Computer Society, Washington, DC, USA, 413–420. DOI:http://dx.doi.org/10.1109/ICPPW.2010.62

Calvin Lin and Lawrence Snyder. 1994. ZPL: An Array Sublanguage. In *Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing (Lecture Notes in Computer Science)*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.), Vol. 768. Springer, 96–114. DOI:http://dx.doi.org/10.1007/3-540-57659-2_6

Ewing L. Lusk and Katherine A. Yelick. 2007. Languages for High-Productivity Computing: the DARPA HPCS Language Project. *Parallel Processing Letters* 17, 1 (2007),

89–102. DOI:http://dx.doi.org/10.1142/S0129626407002892
Jarek Nieplocha and Bryan Carpenter. 1999. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. In *Lecture Notes in Computer Science*. Springer-Verlag, 533–546.
Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. 1994. Global Arrays: a portable "shared-memory" programming model for distributed memory computers. In *Proceedings Supercomputing '94*. 340–349.
Robert W. Numrich and John Reid. 1998. Co-Array Fortran for parallel programming. *ACM FORTRAN FORUM* 17, 2 (1998), 1–31.
Robert W. Numrich and John Reid. 2005. Co-arrays in the next Fortran Standard. *SIGPLAN Fortran Forum* 24, 2 (Aug. 2005), 4–17.
Franz-Josef Pfreundt. 2010. *The building blocks for HPC: GPI and MCTP*. Technical Report. Fraunhofer Institut für Techno-und-Wirtschaftsmathematik (ITWM). 15 pages. http://www.gpi-site.com/cms/sites/default/files/GPI_Whitepaper.pdf
John T. Richards, Jonathan Brezin, Calvin B. Swart, and Christine A. Halverson. 2014. A Decade of Progress in Parallel Programming Productivity. *Commun. ACM* 57, 11 (October 2014), 60–66. DOI:http://dx.doi.org/10.1145/2669484
Lawrence Snyder. 2007. The Design and Development of ZPL. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, 8–1–8–37. DOI:http://dx.doi.org/10.1145/1238844.1238852
Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal* 30 (March 2005), 1–1. Issue 3.
Thinking Machines Corporation. 1991. *CM Fortran Reference Manual, Version 1.0*. Cambridge, Massachusetts.
UPC Consortium. 2005. *UPC Language Specification, v1.2*. Technical Report LBNL-59208. Lawrence Berkeley National Lab.
Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2009. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Languages and Compilers for Parallel Computing*. 172–187. DOI:http://dx.doi.org/10.1007/978-3-642-13374-9_12
Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. 1998. Titanium: A High-performance Java Dialect. *Concurrency: Practice and Experience* 10, 11-13 (1998), 825–836.
Hans P Zima, Heinz-J Bast, and Michael Gerndt. 1988. SUPERB: A Tool For Semi-Automatic MIMD/SIMD Parallelization. *Parallel Comput.* 6, 1 (1988), 1 – 18. DOI:http://dx.doi.org/10.1016/0167-8191(88)90002-6
ZPL Research Group. 2003. *A Comic Book Introduction to ZPL*. Department of Computer Science, University of Washington, Box 352350, Seattle, WA 98195-2350. http://www.cs.washington.edu/research/zpl/comicbook/comicbook.html

Received 2013-06; revised 2014-08, 2014-11; accepted 2015-01