The Asynchronous Partitioned Global Address Space Model

Vijay Saraswat George Almasi Ganesh Bikshandi Calin Cascaval David Cunningham
David Grove Sreedhar Kodali Igor Peshansky Olivier Tardieu
IBM

1. A diversity of parallel architectures

The *multicore discontinuity* is forcing application programmers to deal with a variety of concurrent architectures, such as clusters of SMPs/multicores, hetereogeneous accelerators such as the Cell and GPUs, and integrated high core-count architectures such as Blue Gene.

2. The need for a programming model

The central question in front of us is the design of a *programming model* that can address this diversity of architectures. By a programming model, we mean the central mental constructs around which every activity around the programming enterprise – specification, design, implementation, testing, debugging – is organized. That is, a programming model gives us a coherent set of principles around which to organize our *thinking* about the computation.

While it is relatively easy for the industry to come up with new architectures, it is far more difficult for the industry to adopt new programming models. Indeed, it took about twenty years since the first introduction of object-oriented (OO) programming for the industry to finally accept it. It took about ten years for the US National Labs to move to the MPI programming model. While declarative (functional, logic) programming ideas have been around for several decades — and flourish in certain niche areas — they have not yet established themselves in the main-stream. Application lifetime, number of programmers and users, testing, verification, and maintenance are all factors that contribute to the slow acceptanace.

We believe that each computing era is characterized by a dominant programming model – programming models tend to exhibit a "winner take all" effect. The era of concurrency we now have embarked on essentially requires that a programming model that expects to be dominant must fundamentally account for concurrency, distribution, and composition, and must be able to address heterogeneous architectures. Further, much as the OO revolution did not completely eliminate the previous programming model – procedural programming – rather improved on it in a fundamental way (by providing much better control on structure and extensibility), the new concurrent programming model must improve on the current dominant programming model (OO, procedural programming).

In this note we put forward the APGAS programming model as such a candidate. APGAS is currently being realized through a new programming language (X10), language extension proposals for existing concurrent languages, such as UPC and CoArray Fortran, and through a runtime library which can be invoked from C/C++.

3. Existent approaches

Two broad programming models are currently used: message passing (MPI) and shared memory. Combinations of these models have some penetration, however the burden is on the programmer to isolate them. We discuss each of these programming models with respect to: parallelism (execution model), data structures, and communication.

Message Passing Interface (MPI) [6] is the de-facto standard for programming large distributed memory systems. In MPI, computation is organized around a collection of processes[1] which communicate with each other by sending messages. Each process has a private address space that is inaccessible to other processes (Figure 1a). In most applications, MPI processes are single-threaded, though, increasingly applications are being organized as multi-threaded MPI processes. Single-threaded processes progress sequentially until they encounter a communication action (a call to a function in the MPI API). It is not necessary that all processes execute the same code – however, in practice, many MPI programs are data-parallel SPMD programs. That is, the data on which the computation must be performed is split into roughly equal pieces, each piece is allocated to a process, and all processes execute the same code. Another popular use is in multi-physics codes, in which two or more SPMD codes are linked together by a thin communication layer. There is no explicit mechanism to express global data structures; the programmers are expected to partition the data between the local address spaces of each process and all the abstractions that variables are globally distributed are hidden in the communication patterns. Communication may be *point-to-point*

[1] MPI permits the collection to be dynamically varying. The vast majority of MPI applications, however, spawn a fixed set of processes at the outset. We will mainly discuss such applications.

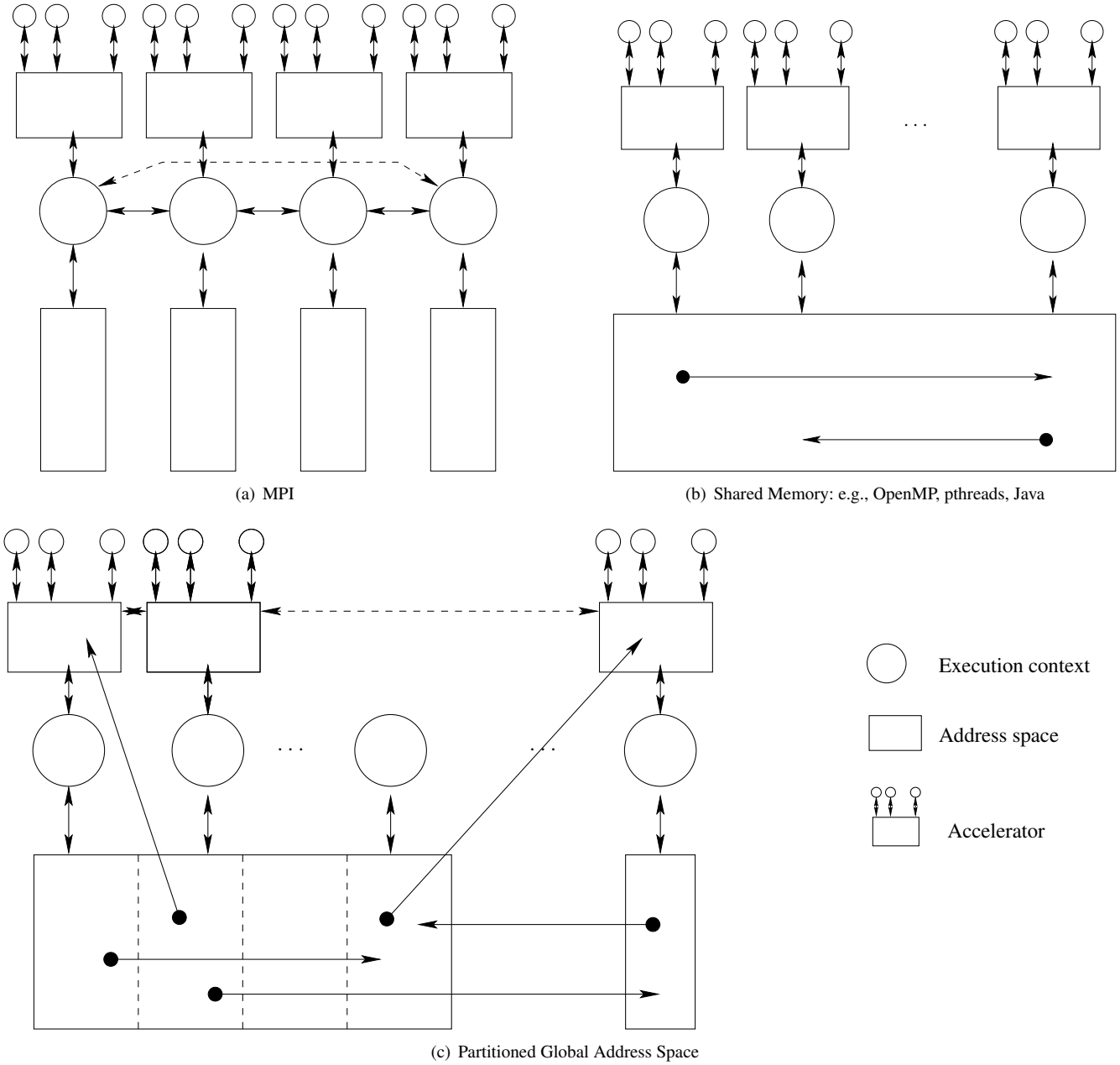

Figure 1. Programmer's view of computation and memory. Arrows represent memory accesses.

(e.g. send/receive) or *collective* – organized via a *communicator* that identifies a subset of processes that perform the same operation simultaneously. Importantly, there is no representation of remote accesses to memory – a "pointer" in the address space of a process may point only to memory allocated in that process.

An orthogonal approach is *shared memory concurrency*. In this model, multiple threads of execution share a *common* address space, communicating with each other by reading and writing shared variables (Figure 1b). This is realized in existing programming languages such as Fortran and C either through libraries such as pthreads and Thread Building Blocks [4], or through language directives, such as *OpenMP* [3]. In general, library approaches are quite flexible, providing the ability to express all types of parallelism: task-, data-, or pipeline parallelism, SPMD or MIMD. Because of this flexibility, there are no guarantees; the programmer is responsible for all aspects of control and synchronization. In OpenMP, constructs are provided for parallel loops and nested parallel regions.

The biggest drawback of this approach is that it is hard to scale on machines that do not support shared memory in hardware. Because by default all data is shared, it is difficult to make this approach scale when the computation must be spread across a cluster and the latency/bandwidth associated with accessing a remote location is very different from that associated with accessing a local location. Attempts to get distributed shared memory have not been very successful.

The *partitioned global address space* (PGAS) model extends the shared memory model to a distributed memory setting (Figure 1c). The execution model allows for computational threads to be distributed across a machine. The existing programming languages in this space – Titanium [9], UPC [7], Co-Array Fortran [2] – all follow the SPMD model of execution. That is, a computation is launched by specifying a single program, which is run in each of the processes that make up the computation. The programming model threads are mapped to processes and threads as supported by the language runtime. The key point here is that most current PGAS runtimes support multiple processes (like MPI), multiple threads in a process (like OpenMP), or a combination. The user decides on the mapping either at compile time or at runtime.

The address space of the processes comprising a PGAS job is unified – so it is possible for a location in the address space of one process to point to a location in the address space of another process. The affinity of a location to a process is enshrined in the programming model, i.e. made explicit – hence we talk of a *partitioned* global address space. Data structures can be allocated either globally (shared by all the computation threads with previously mentioned affinity) or privately. Global data structures are distributed across address spaces, typically under the control of the programmer. Remote global data is accessible to any thread as simple assignment or dereference operations. The compiler and runtime are responsible for converting such operations into messages between processes on a distributed memory machine. While programs may require nothing else but communication through global data structures, most PGAS languages provide library APIs for bulk communication and synchronization: bulk memory transfers, collective communication, and collective synchronization.

An important property of all three of these models is that they are essentially agnostic to the underlying sequential model. That is, they are primarily models of *concurrency* and *distribution*. They may be realized more or less in the same fashion as extensions to members of the Fortran family of languages, or the C family, or object-oriented languages such as C++ or Java. In practice, MPI is mostly used with Fortran and C, and OpenMP with C and C++.[2]

4. The Asynchronous Partitioned Global Address Space (APGAS) Model

There are two drawbacks of the PGAS model that make it difficult to adopt outside the HPC space. First, the PGAS model implicitly assumes that all processes run on similar hardware – only then does the SPMD assumption make sense. Further, the PGAS model does not support dynamically spawning multiple activities. This makes it difficult to handle many non-HPC/non-data-parallel applications, e.g. those that require run-time dynamic load-balancing (as can be expressed in Cilk [1], for example).

The APGAS model can be thought of as being derived from *both* the MPI and OpenMP models by extending the PGAS model with two simple ideas: *places* and *asyncs*. The preferred mechanism of expressing these constructs is the X10 [8] programming language.

4.1 Places

A *place* is simply a coherent portion of the address space – a collection of data together with the threads (activities) that operate on that data. A computation consists of many places.

Places have two important properties. First, they are not required to be single-threaded. That is, multiple activities may be active simultaneously in a single place. More importantly, activities are decoupled from processes and/or threads – they are simply tasks that can be invoked on the data, either locally or remotely. Second, it is not required that all the places in a computation be homogoneous, i.e. be mapped to processors having the same instruction set, or, even, the same number of cores.

This means that places permit expressing computations beyond the SPMD model. Instead we move to a general thread-parallel model, while preserving the locality proper-

[2] In order to handle the hierarchical concurrency offered by clusters of multicores/SMPs, programmers are increasingly turning to the complicated hybrid model of MPI processes each of which is multithreaded and internally is structured as an OpenMP process.

ties of a PGAS programming model. In addition, places can be defined hierarchically, such that the programmer can exploit the hierarchical design of current architectures: clusters of multithreaded SMPs with deep memory hierarchies [3].

4.2 Asyncs

An *async* is the denotational mechanism to express activities that perform computation in a place. We introduce expressions in the language to denote places. Let `p` range over such expressions. Then if `S` is a statement, we introduce the statement `async(p) S`. This statement is executed by launching a new activity at place `p` to execute the statement `S`.

Thus an async is launched at a given place and stays at that place for its lifetime. There are language specific limitations on how asyncs reference remote data. For example, in X10, we require that the async not access locations at remote places. If it desires to operate on remote locations it has access to, it must launch a new async at that place. It is possible to use basic APGAS constructs to get synchronous remote access, so this is not as onerous a restriction as it may sound (see atomics below). In UPC, we allow remote accesses, there restricting only the calling collective operations.

Asyncs may be used not just to run computations at a remote place but also to specify data-transfers (e.g. array copies from an array at a place p to an array at a place q). Such an async while running ostensibly at the remote place may in fact be executed "in the network". Mechanisms specified below (conditional atomics, finish) can be used to determine when the transfer has finished at either the source or the destination, and to group multiple transfers together.

Activities in a place can be spawned locally or remotely. To control their execution, we shall introduce the notion of *finish* – a synchronization construct that allows a parent computation to wait for the completion of all its children activities. If `S` is a statement, `finish S` is another statement. `finish S` executes `S` and waits until all the activities spawned by `S` (recursively) have terminated. Thus `finish` captures the very powerful notion of *distributed termination detection*. It is also extremely easy and natural to use.

Asyncs and finishes represent a tree of computation – each activity has a unique parent (the spawning point) and each parent can have multiple children, which at their turn spawn other children. The finish construct allows synchronization at the desired level in the tree.

Support for a simple form of (nested) data-parallelism is provided through parallel for loops (`foreach` and `ateach`). Both take some form of iteration spaces as arguments. Both spawn an activity simultaneously at every point in the iteration space – the former spawns the activity locally and the latter at a place determined from the point in the iteration space.

4.3 Coordination

Concurrent activities may require tighter coordination and data synchronization than the synchronization offered by the finish construct. In this section we discuss the atomics and conditional atomics.

Atomics. The actions of multiple activities must sometimes be coordinated to ensure the desired result. Typically this is done by introducing locks. However, programming with locks is quite low-level, messy and error-prone. The programmer has to identify a mapping of data to locks, ensure that the same mapping is used always, that locks are acquired in the right order etc. Over-locking can lead to deadlock and under-locking can lead to race conditions.

Instead we propose (as have other language researchers []) to use *atomic blocks*. If `S` is a statement, `atomic S` is a statement that must be executed as if in a single step while no other activities are executing simultaneously. This makes a clean description between specification and implementation – the specification is simple enough for a programmer to work with directly, whereas the mechanism can be realized through a variety of mechanisms. (For instance, a *pessimistic* mechanism may use a system of locks. An *optimistic* mechanism may keep track of dependencies (reads and writes performed in the atomic block) and check that these dependencies are not violated at a commit point.) Since we intend atomics to be used for low-level programming, we shall typically require that the granularity of atomic blocks be small.

In the simplest version of atomic, we also require that all the locations read and written are local, i.e. exist in the same place in which the activity is executing. This leads to a simple cost model for implementing atomics – the programmer can be assured that no communication is involved.[4]

We also permit *conditional* atomics, `when (c) S`. Here `c` is condition and `S` a statement. Such a statement is executed atomically, but only in those memory states which satisfy `c`. That is, execution waits until a state is reached in which `c` is true. In such a state, `S` is executed atomically. Importantly, the evaluation of `c` and execution of `S` is done atomically. This is a very powerful construct (this is the *conditional critical region* of Per Brinch Hansen and Tony Hoare) that can be used to obtain the effect of locks, bounded buffers, communication channels, barriers etc.

Ordering. (Local and remote) Asyncs and conditional atomics offer a powerful concurrent language. However, all ordering between the steps of activity in this language is defined through data-interaction (i.e. through reading and writing variables). While the language is flexible and powerful, it is not easy to analyze statically, e.g. establish that the

[3] The current X10 programming language specification does not support building of hierarchical places, but this will be considered in future revisions of the language

[4] There is an extension to permit atomic blocks to operate on one more more syntactically specified places, other than the local place. This will be dealt with in a fuller version of this note.

program is determinate. However, used in conjunction with the finish construct, it provides the determinacy guarantees that make the APGAS programming model attractive to use.

5. Discussion

The central premise behind APGAS is that the core problems of modern architectures – heterogeneity and non-uniform access to memory – should not be hidden under the rug. Hence APGAS reflects explicit spatial organization of memory and temporal organization of activities. This means that the programmer has to directly confront difficult computational issues. Our basic approach has been to surface simple and elegant ways of addressing these computational issues – places, asyncs, finish, atomicity. In this section we address several details that embellish the model.

5.1 Top-level execution

Computation in an APGAS program is initiated by submitting to the job control system a program and a representation of the places over which the program is to execute.[5] The job control/operating system is responsible for spawning processes with the input information to execute the APGAS runtime. Place 0 then launches a single activity to execute the "`main`" method specified on the command line. The constructs of the APGAS language (or APGAS API) may be used to spawn activities at multiple places and wait for them to terminate. Thus an APGAS program is a "fork join" program rather than an SPMD program.

5.2 Categorizing places

As discussed above, all places need not be equal – a GPU is very different from a Cell SPE and from a core in a multicore chip with coherent caches and from an SMP node. Should the *type* of a place be reflected in the programming notation?

We believe this is necessary for two reasons. First, it helps the programmer to understand that a particular piece of code (async) is intended to be executed in a particular kind of place (e.g. a GPU). Second, the compiler may need to generate different code for different places — the instruction set may be different, a place may have a limited amount of memory. Further, it may need to statically analyze code to ensure that it can run in a particular kind of place. For instance, GPUs typically do not permit an async to invoke a recursive method.

We are currently designing a framework for such a static representation of place types in X10, building on the dependent type framework in X10. A key question is the way in which places can be classified. Should they be classified *by name* (e.g. SPE, Tesla GPU, Intel CoreDuo)? This is clearly not going to be sustainable – technology changes much more rapidly than software. A *property-based* mechanism must be developed. Examples of properties that can be specified for a place are shown in Figure 2. **This is an area of active investigation**.

[5] For X10 v1.7, this representation is similar to that used by MPI to specify which processes must run on which nodes

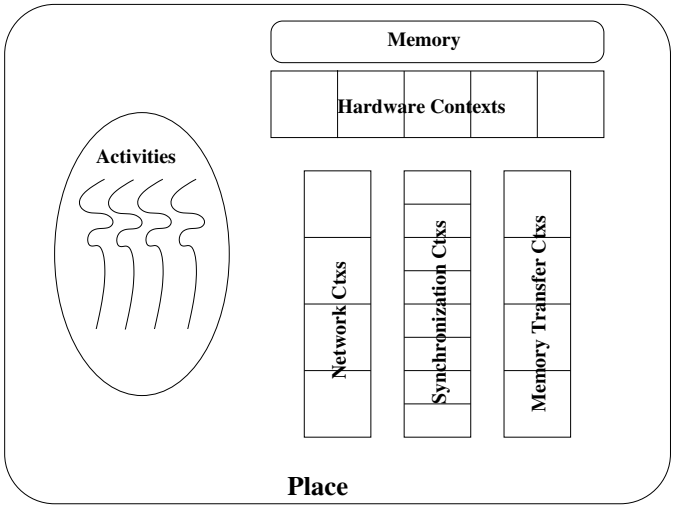

Figure 2. Place resources

5.3 Flat vs structured places

Architectures often are designed as hierarchies: threads, memory, communication are all packaged hierarchically. For instance, the Road Runner architecture has a backbone of Opterons, each coupled with some Cell processors. Thus there is a natural affinity between a Cell processor and the Opteron node it is associated with. The question is: Should this affinity be reflected in the programming model?

Logically, it is not difficult to extend the basic model so that computation runs over a *graph* of places, rather than a flat (unconnected) set of places. One can imagine a simple (declarative) querying model which permits information about the graph to be determined at runtime. The key question is whether this is worth surfacing in the programming model – i.e. will this make a significant difference in the readability of the program or aid the compiler in generating better code? **This is an area of active investigation**

5.4 Granularity of a place

What should the granularity of a place be?

The granularity of a place will depend on architectural constraints. The APGAS runtime, for instance, does not support distributing a single place across multiple processes. However, multiple places may be hosted within a single process. For instance, a computation on a Cell will typically map eight places – one for each of the SPEs – to a single process.

As a rule of thumb, a programmer should look to keep places as fine-grained as possible – given the data requirements of the program – subject to architectural limitations. (For instance it would not make sense to have each thread in a Tesla MT live in its own place.) Note that single-threaded places (places in which only one core is active at any given

time) enjoy the enormous computational advantage that atomic operations are "free" – as long as the core is not interrupted in the middle of an atomic operation.

5.5 Fixed vs dynamic number of places

Should it be possible to spawn places dynamically at runtime? In certain situations (e.g. the program is but one of many jobs running on the operating system) it may make sense for the program to dynamically request more resources from the OS or release resources it can no longer use, reflecting the available concurrency in the program (e.g. see recent work by Kunal Agarwal.) Perhaps the simplest way of requesting more resources is to spawn a new place and launch a computation at that place. The APGAS runtime could then determine — based on the "density" of places over the given set of computational resources — whether to request more resources from the OS.

As an aside, when should a place "die"? It makes most sense to think that a place should cease to exist if it has no resident data that is "live", i.e. that can be accessed from the rest of the computation. However, this would mean that places like GPGPU places – which have no resident data but essentially serve as "compute servers" – would die as soon as they are created. So clearly there should be some mechanism to mark a place as "persistent" and permit it to be available even if it has no live data. (By default, a place would be "transient" – live only as long as it has live data.)

Another illustrative example for dynamic creation of places is provided by web servers. It is most natural to think of a "web session" as associated with its own (transient) place. A place thus serves as a "container" for all the objects created during the course of this web session. When there is no oustanding reference to the web session, the session/place can be garbage collected. **This is an area of active investigation**

5.6 Failure and recovery

Many applications outside the HPC area that are using large systems are designed for clusters of commodity machines that tend to fail. Therefore, middleware such as Hadoop and the like include fault tolerance support. In addition, current work has in MPI has also started addressing fault tolerance. The APGAS places are a natural point to encapsulate checkpointing, but there are a number of open questions with respect to providing fault tolerance at the level of the programming model. For example, should the programmer be involved in specifying the state that needs to be checkpointed? Mark quiescence points in the program or just piggyback on finishes? What are the guarantees provided by the system? **This is an area of active investigation**

5.7 Load balacing of places

While a static mapping of places to computational nodes makes sense for many applications, it is problematic for situations in which load may vary dynamically in a data-dependent fashion. For such situations, some form of load balancing across homogeneous nodes may be appropriate. Here we expect the notion of a place to play a central role – the unit of relocation and redistribution should be a place. **This is an area of active investigation**

5.8 The deadlock-freedom guarantee

Programs written using async, (unconditional) atomic, finish[6] are guaranteed to not deadlock. This property can be established quite simply by realizing that this language does not permit the creation of cyclic dependency structures.

5.9 Global data structures

APGAS languages also offer some capabilities to define global data-structures, i.e. data-structures portions of which live at different places. A canonical example of such a data-structure is a global array, scattered over multiple places.

5.10 Remote Invocation

As discussed before, activities can be invoked on a place locally or remotely. As the APGAS places decouple the notion of activities from the physical processes and threads, we now face the problem of providing fairness guarantees for the execution of activities. One may envision different solutions: priority queues that allow progress in a place on all posted activities, dedicated physical threads to process remote activities and communication, etc. Each solution has different advantages and its benefits depend on the characteristics of the application: its communication and synchronization patterns and computation load balance. **This is an area of active investigation**.

5.11 One-sided and Collective Communication

While atomics provide simplified coordination through memory, most distributed memory machines provide high performance interconnects optimized for bulk transfer. One such example is RDMA communication, which can be well exploited by the one-sided APGAS programming model. Collective communication, in particular collective communication between subsets of activities determined by data distribution (data-centric collectives) are another direction where the APGAS programming model offers advantages over existing PGAS and MPI models. **This is an area of active investigation**.

5.12 Higher-level language

We anticipate that higher-level concurrent languages can be built on top of the APGAS model. Such languages may offer stronger guarantees – e.g. determinacy – and permit the programmer to specify complicated patterns of concurrent computation and communication in a simple fashion (e.g. nested data parallelism).

[6] And a general form of barriers called clocks which are programmable using conditional atomics

We expect that compilers/translators for such high-level languages would be able to use the APGAS tool-chain.

6. Pragmatics

How is the APGAS programming model to be realized? Above, we have outlined a set of language constructs for concurrency and distribution. These constructs can be added to Java or Fortran or C. The X10 programming language may be thought of as being obtained from Java by adding these control constructs. Below we outline two other approaches: a runtime library approach and UPC extensions.

6.1 Runtime library

We provide an API that realizes the APGAS programming model through a runtime library. Similar to the MPI library, the APGAS API provides the following capabilities:

1. Create a *remote reference* to a location. Such a remote reference can be transmitted to other places
2. Spawn an async (at a given place) to execute a given function with given data.
3. Execute a `finish` operation on a given async.
4. Efficiently execute multiple asyncs in a given place, e.g. using dynamic scheduling techniques such as work-stealing.
5. Implementation of atomic, and conditional atomic.

6.2 UPC extensions

UPC already supports the PGAS programming model, therefore, we need to extend the language to handle *aysncs*. We propose the following extensions:

- `upc_async ([(affinity clause]) <stmt block>` — spawns an activity at the remote place specified in the affinity clause (or locally if the affinity clause is not present). The block of statements that executes as an async may execute any arbitrary UPC code, except for code involving collective operations (such as `upc_forall`, `upc_all_alloc` or any of the collective functions defined in the UPC collectives API. In particular, `upc_async` blocks may spawn other asyncs, access remote memory location, etc.
- `upc_finish [<stmt block>]` — waits for the termination of all the asyncs spawned from within the block of statements. That includes all dynamically scoped asyncs, i.e., asyncs spawned by other asyncs, not only the lexically scoped asyncs. If the statement block is not specified, the thread executing `upc_finish` will wait for the completion of all locally spawned asyncs.

In the current proposal, the asynchronous extensions to UPC preserve the SPMD execution model, in which all UPC threads start executing before the main routine is entered. `upc_async` provides a mechanism to easily express fine-grain, "user-level" tasks, akin to the tasks presented in [5], but relying on the compiler and runtime system to schedule and execute the tasks. The asyncs get executed by the worker UPC threads. There are no fairness and progress guarantees, except when `upc_finish` is used. `upc_barrier` implies a `upc_finish`, i.e., the barrier will complete only when all activities spawned in the sychronization epoch complete.

As mentioned before, the only restriction that we put on aysncs is that they can not invoke collective operations. Therefore, strict accesses in `upc_async` regions have fence semantics, not barrier semantics, since barriers can not be called from within an `upc_async`.

7. Conclusion

APGAS offers a simple but powerful concurrency and distribution model. It can bre realized through programming languages or through an API that can be used within any existing sequential language. It can be implemented on top of architectures which support clusters of multicores, symmteric multiprocessors, accelerators (such as the Cell, GPGPUs), and high core-count integrated networks such as the Blue Gene.

Acknowledgements. The APGAS model arose from an abstraction of the X10 programming model, and was first defined in Novemeber 2007. It is being developed in collaboration with several colleagues at IBM and elsewhere, including colleagues on the X10 team, Kevin Gildea, Bob Blainey and others.

Please send comments to Vijay Saraswat, vsaraswa@us.ibm.com.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[2] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1 – 31, 1998.

[3] The OpenMP API specification for parallel programming. `http://www.openmp.org/`.

[4] J. Reinders. *Intel Threading Building Blocks: Multi-core parallelism for C++ programming*. O'Reilly, 2007.

[5] A. G. Shet, V. Tipparaju, and R. J. Harrison. Asynchronous programming in upc: A case study and potential for improvement. In *Proceedings of the First Workshop on Asynchrony in the PGAS Programming Model*, June 2009.

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference, second edition*. The MIT Press, 2000.

[7] *UPC Language Specification, V1.2*, May 2005.

[8] The X10 Programming Language. `http://www.x10-lang.org`.

[9] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13), September-November 1998.