Multi-threading Semantics for Highly Heterogeneous Systems Using Mobile Threads

Peter M. Kogge
Dept. of Computer Sciene and Engr.
Univ. of Notre Dame
Notre Dame, IN USA
kogge@nd.edu

***Abstract*—Heterogeneous architectures are becoming the norm. The results are nodes that are not only multi-threaded, but simultaneously multi-threaded across several different instruction sets and core designs. Unfortunately, programming models for such systems are still evolving, and are nowhere near adequate as we move into an era of extreme heterogeneity with many new accelerator designs. This paper discusses the current range of multi-threading models and what features are liable to be needed for such future architectures. In addition, we suggest the potential value of using a new threading model, termed migrating threads, that may be an excellent match for a common "glue" to efficiently combine all the emerging heterogeneity.**

***Index Terms*—Fine-Grain Parallelism and Architectures, HPC Architectures for Mobility, Instruction-, Thread- and Memory-Level Parallelism, Programming Languages**

I. Introduction

The 2008 DARPA-sponsored Exascale technology report [28] projected that systems with exascale capabilities will of necessity have to support billions of concurrent threads running on cores that individually are not much faster than today. To reach these levels, recent years have seen the emergence of **heterogeneous systems** employing a mix of high-end server and GPGPU chips that support large numbers of threads. Such systems employ multiple core microarchitectures, meaning that different parts of a program that uses multiple core types must be compiled differently, with code required to bridge the gaps. Followup reports [25]–[27], [29] have confirmed that heterogeneity is in fact the dominant trend. Looking forward we see continued heterogeneity as new technology accelerators come into play. The explosion in neural nets and machine learning is just one example.

Today the dominant memory model is also increasingly heterogeneous. Within a single node there is a large main memory space with hardware-enforced coherency between the general purpose cores and their increasingly deep cache hierarchies. However, the memory used by the GPUs is both physically and logically separate from that main memory. Explicit transfers must be made to move data between them. In contrast, the memory spaces between nodes are totally disjoint, with relatively complex software protocols needed to communicate between them, either for data transfers or to signal a remote computation. For the dense, regular computations of the past, this has been sufficient. However, recent apps are exhibiting much more sparsity and irregularity where today's deep cache hierarchies are of little use for programs within a node (cf. HPCG [15], [34]), and the software-based inter-node messaging protocols often swamp the computation time for multi-node algorithms (cf. [40]).

Looking forward, the emergence of 3D stacked memory [41] has suggested positioning cores on the bottom of individual memory columns within that stack, and then coupling those stacks together in a "sea of stacks", each of which is a "node" [35], [36], [42]. Different stacks may have different types of cores simply by swapping out the processing die, This will implement heterogeneity at a much finer level.

The major issue is the need for a program to explicitly handle references to data that is "here" vs "there," and to be able to start new threads where they have most easy access to the required data and/or have a processing core most attuned to the needed work. Underlying software (as in the UPC language [18]) and/or hardware (as in HPE's "The Machine" [1]) is needed to maintain the illusion of a shared space. This, however, does not address the heterogeneity in processing that we see at the same time.

A common attribute of virtually all of today's models is that threads are largely "stationary." Once started on a core, they seldom move. There is "invisible" migration of threads between cores of the same type for such things as load-balancing, power-moderation, and wear-leveling [7], but that is all managed by software. Techniques like **RDMA**[2] support data access and "active messages" [50] for a remote function spawn, but again need expensive software implementations such as SHMEM [11] and GASNet [4].

Providing a thread of computation the ability to "migrate" much more freely to different execution sites may very well help solve both problems. Reducing the need to distinguish between "here" and "there" both drastically reduces a program's complexity and possibly improves performance. This is even more true when the "migration" can cross an ISA boundary as in a heterogeneous processing system. Architectures have been proposed to implement such capabilities both in multi-node systems and in cores integrated very near or inside the

[1] https://www.labs.hpe.com/the-machine
[2] Remote Direct Memory Access

978-1-7281-4484-9/19/$31.00 ©2019 IEEE

memory hierarchy, but to date have not addressed the "cross-ISA"/"cross-core type" issues.

This paper makes a first attempt at defining more completely the dimensions found across all threading models, and integrating them into a common framework, where both PGAS considerations, multiple core types, and potential mobility of threads are considered. Section II provides some definitions. Section III goes through a brief history of architectures with migrational features. Section IV defines the different dimensions under discussion. Section V then suggests a common paradigm. Section VI discusses what must happen when various types of code invocations are made in such systems. Section VII suggests a possible model, with language features, that incorporates all the above. Section VIII concludes.

II. DEFINITIONS

For this paper, a **thread** is a set of state information that is sufficient to guide a path of execution through a program. This state may include register values, a call history stack, or heap memory allocated to the thread. A thread is **spawned** when its state is first assembled and given to a core for execution. A **thread execution** is the updating of the thread state as the thread performs the actions called out by the thread's program. The **lifetime** of a thread is the time from its spawning until the time when it gives up its state and terminates.

A **core** is the logic that can advance the state for a thread's execution. A **locale** is the memory and core(s) that are packaged together so that threads running in the locale's core(s) can access the locale's memory more easily than the memory associated with some other locale. An **ISA** (Instruction Set Architecture) is the set of instructions a core recognizes.

A **PGAS** (**Partitioned Global Address Space**) system is one where the memory is physically partitioned into locales, but where it is all in the same address space, so that an address prepared by a thread anywhere can be uniquely associated with the physical memory holding the associated memory.

A **distributed address space** is one where are multiple physical locales, and a thread running in one cannot directly generate an address to the memory in another.

A locale's cores thus have an **affinity** to the locale's memory, as does any thread executing on those cores. Further, a particular thread may have two kinds of affinity: **affinity of creation** as to which locale it was spawned (and thus calls **home**), and an **affinity of execution**: the locale(s) where it actually executes. These need not be the same. In particular, for this paper, a **stationary thread** is one whose site of execution in terms of locales never changes from where it was spawned. A **mobile thread** is one whose execution site may change one or more times in its lifetime. A **migration** occurs when a mobile thread switches its locale of execution.

An **AMO** (**atomic memory operation**) is typically when one thread performs a sequence of operations against some memory location(s) in a way that appears to be "instantaneous" to any other thread attempting to access the same location(s).

There are also variations in how threads monitor the progress of other threads. The term **barrier** denotes an operation that is executed at some point by a group of threads, with no thread permitted to advance until all threads have checked in, at which all threads continue again. A **sync** operation is subtly different in that it is typically used by a parent thread to determine when a group of child threads it has spawned have completed their execution. The term **fence** is similar, and used typically when the child "threads" are very short, very transient, operations such as accessing memory remotely.

An **active message** [50] involves the specification of some function to be run against the data associated with some locale other than the current one. Such spawns look like conventional calls but semantically are **non-blocking** where the calling thread need not wait for the called computations to complete. The "called" function thus may run as a separate thread in a target locale specified by the caller. A **reply** capability allows such a function, when it has completed execution at its remote location, to return something to the calling node, where the results are returned to the caller's space back into the caller's memory and/or synchronize with the parent thread.

III. THREADING ARCHITECTURES

There is a standard taxonomy of core microarchitecture that support multi-threading in hardware [49]. Real microarchitectures that support multi-threading in some form date back to the CDC-6600's I/O Processor [47]. The Denelcor HEP [21] carried this into a general-purpose framework, and the later Tera MTA [45] and Cray XMT [31] went further to provide a PGAS environment where any thread on any node could make load/store access to any memory location.

Typical GPUs such as from Nvidia support a dual level of multi-threading: **thread blocks** execute on **stream processors** that in turn host a large number of **worker threads** that follow instructions issued by the stream processor.

The idea of imbuing threads with a sense of "mobility," where at least a large part of its state can move between cores, is beginning to become more serious. Hardware-supported spawning and migration for small pre-defined operations such as remote atomic memory operations go back decades, such as in the Cray T3D [12], [23]. The J-Machine [13], [14] provided each core with hardware support for something akin to active messages, where one core could explicitly send a function spawn to another node. Architectures have been proposed to implement thread mobility in cores integrated very near or inside the memory hierarchy [6], [19], [24], [32], [37], and some have actually been demonstrated mobility in prototype hardware [5], [48].

More advanced in terms of mobility is the **Emu Context Flow Architecture** [16], Fig. 1. Here the unit of parallelism is a small locale-like unit called a **nodelet**: a single memory channel, its controller, and multi-threaded core-like logic to execute migrating threads. All memory in the system is in a common address space. What distinguishes this from a conventional multi-core shared memory system is that threads are not stationary. Instead whenever a thread executing on a nodelet tries to access a memory location not on the nodelet, the hardware suspends the thread, packages its state, and ships

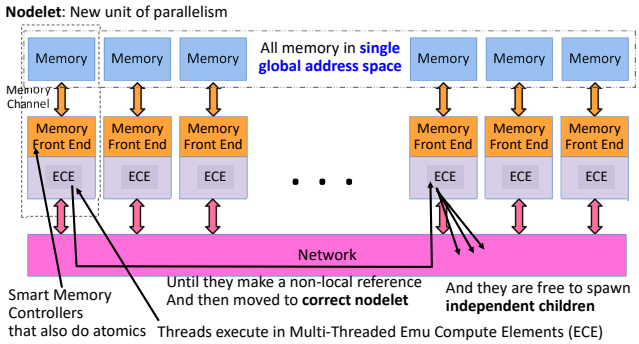

Fig. 1. The Emu Context Flow Architecture.

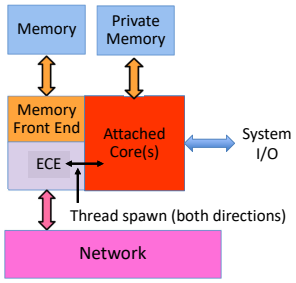

Fig. 2. A Hybrid Nodelet.

it to the correct nodelet and restarted on a local core, with no knowledge that it had moved. In the Emu case, the state that is migrated are a set of hardware registers akin to those in any conventional system. Most of these registers hold arguments and working data; others hold pointers back to larger parts of the thread's state, such as the call stack. In addition, a thread can spawn, with as little as a single instruction, a child thread that is free to pursue its own execution path. The compute logic on each nodelet is multi-threaded so that with sufficient threads there is always one that is ready to access local memory, and thus keep the local memory channel busy. There is a rich set of AMO instructions that are executed directly in the memory controller without any of the complexity or performance hits found with conventional architectures. A thread may also spawn an even lighter weight thread to perform remote AMOs. Applications well-suited for such an architecture include those that are memory intensive, irregular in access, or highly sparse, such as in big data or big graph problems [30].

Not shown in Fig. 1 are conventional cores with interfaces into a nodelet, as approximated in Fig. 2. Today these conventional cores can transfer data between nodelet memory and either their own memory or file systems. They can also inject threads into the nodelet system to initialize computation.

Looking ahead, there is nothing to prevent the spawning of a thread to "go the other way," particularly when the attached core is a GPU or accelerator of some kind that can perform specialized computations on the local memory. A mobile thread that lands on some nodelet because of a reference to the memory in that nodelet could decide to spawn a child thread, but that child could be a thread for the attached core. The attached core may then use the arguments in the mobile thread's state as parameters to tell it what to do. The mobile thread could then either continue execution or wait at the attached core's interface for it to finish. In the latter case, at completion of the spawned function, the attached core could return results to the waiting mobile thread and release it.

This technique could greatly simplify programming large heterogeneous systems with many cores of a variety of types. A common mobile thread infrastructure provides a standardized "glue" to stitch together all the disparate compute elements. Different attached core types could have their processing defined as libraries of functions annotated as callable from a mobile thread spawn. Thus different functions on different nodelets could be called by a mobile thread representing the main flow of an application. The local cores need never know how their computation is tied into a bigger stage.

IV. Threading Dimensions and Current Models

Following is a list of different threading characteristics:

- when may new threads be spawned in a program,
- how many such threads may be spawned at a time,
- what is the lifetime of a thread relative to its program,
- are threads "named," and if so to whom is the name visible,
- what is the affinity of a thread when it is spawned,
- what parts of the system's memory are visible to a thread,
- how mobile is a thread during its execution,
- how does a thread interact with shared memory,
- how do threads interact, especially with parents.

The following subsections overview three distinct classes of such models. Table I summarizes some key characteristics of a cross-section of real languages and libraries for each class. These examples were chosen for either historical or feature set reasons. We use the term "package" to refer to either a complete language with threading features "built in," or a library used with a conventional language.

A. The SPMD Threading Model

The **SPMD** (Single Program Multiple Data) model has multiple threads started at the beginning of a program execution, with each thread running the same code but on a distinct region of memory. Such threads are long-lived - for the lifetime of the program, are "stationary" on a locale established on their creation, and typically do not spawn child threads.

Packages in this class separate into two groups depending on whether or not the memory associated with one locale is addressable by a thread in another. The primary package today supporting SPMD distributed memory models is **MPI** [20]. Libraries such as **SHMEM** (and more recently **OpenSHMEM** [11] and **ARMCI** [39]) support a PGAS-like memory model where some memory in each locale has been contributed to a larger pool against which a variety of access operations is available via library calls. Most implementations invoke remote threads on the target locales to perform the operation, that in turn invoke another transient thread on the requesting locale to return the data back to the requestor's data structures.

UPC [18] is an SPMD language with explicit PGAS support. The original UPC memory model assumed one thread

TABLE I
SOME RELEVANT PROGRAMMING LANGUAGE AND LIBRARY CHARACTERISTICS.

Language	Thread Class			Type		Lifetime						Coordinations			Atomic Operations			
	SPMD	Team	Asynchronous	Stationary	Mobile	Long	Moderate	Short	Transient	PGAS-aware	Named Threads	Barriers	Sync Points	Fences	Basic AMOs	Locks	Reductions	Atomic Sections
Primarily SPMD Threading																		
MPI	E			E	I	E				No	Yes	Yes		Yes	Yes		Yes	
SHMEM	E		I	E	I	E			I	Yes	No	Yes		Yes	Yes	Yes	Yes	
Split-C	E			E	I	E			I	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
UPC	E		I	E	I	E			I	Yes	Yes	Yes		Yes	Yes	Yes	Yes	
Chapel	E	E	E	E	I	E	E	E	I	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes
X10	E		E	E	E	E	E	E	E	Yes	No	Yes	Yes		Yes	Yes	Yes	Yes
Primarily Team Threading																		
OpenMP		E		E		E	E	E		No	No	Yes	Yes		Yes	Yes		Yes
CUDA		E		E		E	E	E		No	Yes		Yes		Yes			
Intel TBB		E		E		E	E	E	E	No	No	Yes		Yes	Yes	Yes	Yes	
Primarily Asynchronous Threading																		
Smalltalk			I		I	E	E	E	E	I	No		Yes		Yes			
P-threads			E	E		E	E			No	Yes	Yes	Yes		Yes	Yes		
Multilisp		E	E	E		E	E			No	No							
GASNet	E		E	E	E	E	E	E	I	Yes	Yes	Yes	Yes	Yes		Yes		
Cilk		E	E	E		E	E	E		No	No		Yes			Yes		Yes
Charm			E	E	E	E	E	E	E	Yes	Yes						Yes	Yes

E = Explicitly stated by programmer. I = Implicit in implementations.

per locale, with access to two memory spaces: a "private" memory accessible only to that thread, and a shared memory space accessible by all. Although any thread can access freely any location in shared memory, there is a program-definable sense of **affinity** for some specific data (especially arrays) to a specific thread, where it is typically far faster for a thread to access data for which it has affinity than to access other shared data with affinity to other threads.

Chapel [8], [9] and **X10** [17], [43] were both programming languages that came out of the DARPA HPCS project with roots in an SPMD model but with significant capabilities for asynchronous threads. Chapel has its roots in a combined multi-mode threading and PGAS model with significant team and asynchronous capabilities that is richer than UPC, including more explicit definition of locales and a variety of distribution options for arrays striped among various locales.

In X10, a prefix *async* in front of a statement creates a new asynchronous thread called an **activity** that shares the parent's heap but has its own control stack. Affinities to locales (termed **places**) are specifiable on a per statement block basis by an *at* (<place-name>) <statement>. When such a statement is encountered, part of the current thread's state are migrated to the new place, and unpacked into a new control stack that is used by a local thread. When the thread completes, execution is restarted back at the original place. Nothing is copied back from the new stack to the old stack.

B. The Team Threading Model

The **Team** model has groups of threads spawned by the program at logically the same time and locale to work their way through some programmer-specified set of work, such as through the different iterations of a *for* loop. Team threads are anonymous. They typically use some sort of **work-sharing** or **work-stealing** [3] where the next available thread picks off (atomically) the next unit of work to do. The lifetime of a team thread falls somewhere between short to moderate.

OpenMP [10] and **Intel Thread Building Blocks** (**TBB**) [22] are two common team packages that are designed for multi-threading within a shared memory environment with fairly heavy-weight long-lived threads.

CUDA [38] (and the newer **OpenCL**[3]) assume there are two distinct kinds of cores involved in two distinct types of locales: a conventional one and one built around GPUs. There are two address spaces, for which CUDA provides RDMA operations to transfer data between the two. Function calls made from a conventional host processor to a GPU define an implicit grid of indices into GPU-resident data structures that can be executed by different threads of different blocks.

C. The Asynchronous Threading Model

SPMD and team threading largely hide the act of spawning a new thread from the programmer. In an **asynchronous** model, not only is the act of creation a programmer-specified event, but so is the specification of what code the new thread is to execute and the key parts of the initial state. Further, unlike SPMD and team, there need be no commonality between the code of the parent and that of the child(ren). However. there must be a way for the parent to know when the child has completed, and for the child to return results.

P-threads [33] is perhaps the most well-known threading package for shared memory system. There is no explicit affinity to threads when they are created, and they may run for arbitrarily long periods of time. It is a library with

[3]https://www.khronos.org/opencl/

four categories of calls: thread management, interaction via mutexes, communication via condition variables,and thread synchronization via locks and barriers.

GASNet [4] is a library that provides the ability to spawn threads remotely. An argument on the spawn provides the affinity of the child thread, expressed as a locale id coupled with a local memory address. The lifetime of these threads is shorter than the threads that spawned them.

Cilk [2] is an extension of C that provides dynamic multithreading where threads are spawned, merged, and die as needed by the program. It is the base language for programming the Emu system discussed earlier. The core Cilk adds to C three keywords dealing with thread spawning:

- *cilk*: an attribute to a function definition that identifies that function as being capable of being called and executed as an independent thread, without blocking the caller's execution.
- *spawn*: a keyword preceding a function call that specifies a "non-blocking" call to be executed by a separate thread.
- *sync*: all threads spawned within the enclosing block must complete before execution continues past the sync.

A new generation of compiler technology, **Tapir** [44], is enhancing the ability of Cilk compilers to extract fork/join parallelism automatically.

V. Unifying the Paradigms

No single language from Table I encompasses all of the attributes needed for all models. Further, as we move towards very heterogeneous systems with many more locales, especially PGAS-based ones, we will need a richer and more unified paradigm. This section moves towards such a unification.

A. Stationary versus Mobile Threads

First, at least two kinds of threads should be definable: stationary and mobile. The stationary class accounts for almost all of today's threading, except for active messages and the largely hidden implementation of RMOs.

Mobile threads are best at attacking problems that either have significant remote accesses, little reuse, and/or change the type of core to be used. They fall into several categories:

- **spawn-time mobility** where a thread is explicitly created by some parent thread to be executed on some other locale or a different core type on the current locale,
- **run-time directed mobility** where a thread is migrated "under the covers" by a run-time for some system reason.
- **programmer-directed mobility** where a thread migrates only under programmer control.
- **location-specific mobility** where directives on data definitions specify when an access should cause a migration.
- **fully self-migrating** where a thread moves from locale to locale as non-local run-time data references are made.

In addition, mobile threads should be capable of spawning other threads (of any kind), just like stationary threads. Such a capability is not currently allowed in, for example, GASNet active messages, although in many cases this is an implementation, not necessary semantics, limitation.

Finally, if a thread is to be created as mobile, an attempt should be made to minimize the state that it must carry with it. Thus it may make sense to resurrect something akin to the old C syntax *register* on function arguments to try to keep those arguments in registers rather than in memory. It may also make sense to separate any call stack frame associated with a thread from its working registers, but provide mechanisms for the thread to remotely access the call stack. The Emu system does this by keeping a thread's stack in some home nodelet memory and migrating back when needed to access it.

B. Other Variations in Thread Types

While both team and SPMD thread groups can be crafted as a series of explicit one-at-a-time asynchronous spawns, their ability to create at least teams with simple syntax is a significant programming simplification. Once a programmer has the ability to define a team with a thread per locale affinity, then SPMD-style programs are also built easily.

The final consideration in thread typing is the ISA of the core where the thread is to run. We note that this is actually orthogonal to the stationary versus mobile discussion. A single CUDA program [38], for example, describes the code for both conventional and GPU core types in a single program, with the compiler capable of identifying from annotations which kind of core will run which section of code, and compile those sections differently. Looking forward, we will see more of this kind of inhomogeneity, especially when spawning non-blocking threads to run on other core types.

The net effect of this is that any unified paradigm must allow the compiler to determine which pieces of code are to be compiled for which ISAs, and to be able to use that information to compile in the right kind of transition code when a change in ISA is encountered.

C. Affinity

For PGAS systems, the "affinity" of a thread is also important. Today, affinity appears in SPMD and team languages as a filter for loop iterations for groups of stationary threads. A few languages like Chapel [9], OpenMP 4.0 [10], and GASNet [4] provide an affinity argument to control where a new thread is to be spawned and then run. Looking forward, for team and asynchronous stationary threads, this affinity is at least the locale where a spawned thread should execute. For mobile threads, this affinity should include both the locale that the new mobile thread calls "home," and what should be the correct affinity during execution.

A logical choice for specifying affinity is the name of the desired locale at spawn time. This is the MPI model. For PGAS systems, specifying an address in PGAS space can serve the same purpose without a programmer having to know which locale holds which data. However, this does require mechanisms to make this mapping between addresses and locales. The Emu system does this in hardware.

D. Thread Naming and Families

As discussed above, there is a real dichotomy in whether or not new threads should have "names". SPMD languages

almost uniformly have intrinsics such as *MYTHREAD* that return a value unique to each thread instance, usually tied to the thread's locale. Others such as GASNet use opaque "handle" objects created at the time of a spawn. This gets difficult to manage when many threads are to be created, as in a team construct, and/or in distributed memory systems where a handle is accessible only from the locale where the object was built. Still other languages such as Cilk provide no such programmer-accessible identifier whatsoever.

Almost all threading languages, however, need some way for the parent of a thread to be able to track the progress of the child and know at least when they have completed. What we propose here is a variant of of the opaque handle but designed to handle "families" of related threads. When a thread wishes to spawn a new child thread, it creates/specifies what we tentatively call a **Family Control Block** (**FCB**)[4]. This FCB is accessible to the parent, with a reference given to the child in a way that it can also access it. When the child is spawned, the FCB is marked as tracking a new live thread. When the child completes its execution, it resets this marking, and may optionally either restart a waiting parent or simply die and allow the parent to check at its convenience.

This is virtually the same as today's handles in packages like GASNet, except that a program may re-use the same FCB for different spawns, even before the first child completes. If each child of a parent thread gets a unique FCB, then the parent can track each individually. If, however, a team is created with a single FCB, then a count is established in the FCB for the number of outstanding children, and the parent can track the status of all of them in aggregate. Again, options can tell the "last" child to complete to do something special such as awaken the parent.

While obviously useful for a team, the same mechanism could be used for asynchronous spawns launched for related but not identical computations. Consider a search tree where using the same FCB for all the randomly-created children allows the parent to track progress as if it were a team, even though each child may be executing a different set of code based on the tree vertex it is covering.

Additionally, there is be no reason why a parent thread can not reuse the FCB given it by its parent, thus allowing a thread to dynamically create a "sibling" thread that is tracked not by it but by its parent. This may prove to be an extraordinarily valuable simplification for large irregular programs such as Breadth First Search[5]. There should be no restriction on the type of all members of a thread family (stationary or mobile), only that they interact with other family members and report completion via a single standard mechanism.

In addition, a parent thread need not have only one thread family active at a time. It should be allowed to create multiple concurrent families, all of which are allowed to run independently, with the parent able to check the status of any one family independently using separate FCBs.

[4]The Habanaro [1] paradigm has something similar

[5]https://graph500.org/

Further, it may be valuable to "link" FCBs together so that the "last" child to return to a particular FCB should be able to detect that it is in fact "the last," and then proceed to a parent FCB and repeat the process. Such tree-like structures would allow large numbers of threads to terminate without hot spots around singleton FCBs. Similar mechanisms would be useful for barriers where threads don't die but are suspended.

These FCBs are also convenient places to put information as to the type of thread to be spawned, or to simplify the process of building teams or swarms of similar threads. In addition, an FCB could also be used for some inter-family communication, such as a common place to put an *abort* or *eureka* flag that could be set by either a parent or child to tell all other threads to truncate their executions early. Whether this abort is a command that goes to all children (possible for stationary threads) or is simply an advisory that can be tested by a child at its convenience (as in a mobile thread whose location is unknown) is up to the language implementation.

E. ISA-Specific Specifications

In terms of the scope of the code to be given to a new thread, there are two major approaches: as a function (Cilk, GASNet) and as a block of statements (OpenMP, CUDA). Languages that use the former typically include prefixes for the function definition. Languages that use the latter allow either special syntax (as in CUDA) or simply placement within a bigger syntactic unit (as in *forall* loops in team languages).

For other thread characteristics such as core type/ISA or stationary/mobile, it is unlikely that a compiler can figure out from context, so that some sort of annotation is appropriate, most probably as prefixes to statement blocks. First, as with Cilk, there may be blocks of code that need to be compiled in multiple ways. Second, a programmer may want to specially annotate specific statements within a statement block that has some annotations already. An example might be some loop within some larger code block that might be best run as a mobile thread (for example, a loop doing pointer chasing). When the outer block is compiled as a mobile thread, no special considerations need to be given to the inner loop, but if the outer loop is stationary, the annotation on the inner loop triggers code for a thread creation.

F. Atomic Operations

Virtually all current multi-threaded languages have a suite of built-in functions for performing operations where the update to a memory location is to be done atomically in relationship to other threads. This is limited, as in most cases there is no way to specify new atomic functions without complex locks.

One exception is Chapel, which has the *atomic* prefix for a block. OpenMP has a similar key word as a prefix to a single statement and the keyword *CRITICAL* for a block. We know from experience that the single statement form is too limiting, while the general atomic block is quite powerful but very difficult to implement in practice (see for example [46]). A reasonable compromise might be to use Chapel's *atomic*,

but explicitly limit its atomicity to a particular programmer-specified subset of variables (including just one).

G. Value Return from Child Functions

A variation of the issue of atomic operations is the return of values from a child thread that is created to run some function asynchronously. Consider a Cilk-like example:

```
sum += spawn foo(...);
... continuation...;
```

At the spawn, the parent continues with the code after the spawn, while the child concurrently executes *foo*. The issue is that the code to implement the $sum+=$ is part of neither threads' scope, and cannot be started until the non-blocking call to *foo* completes. There are potentially three threads: the parent executing the spawn, the thread executing *foo*, and the thread executing the $sum+=$. This latter thread must be spawned (or unsuspended) by the completion of *foo*, but must run in the environment of the parent. If the parent and child threads are stationary threads running on different locales, then this third thread could be a very short mobile thread.

This is actually very close to the semantics of GASNet, where a "reply" thread must be spawned by an active message thread to return values to its parent's context. In GASNet it is up to the programmer to explicitly encode such a reply function and call it from the active message thread.

A further complication occurs if the parent either updates *sum* itself, or has gone on to make additional spawns that all want to update *sum*. There may be a burst of updates to the same variable at approximately the same time from different threads. GASNet ensures atomicity by providing a programmer with the ability to "block interrupts," thus guaranteeing unfettered access during the update process. The original Cilk allowed a programmer to identify an *inlet* function that runs in the parent's environment atomically.

A unified model should simplify such expressions by encapsulating the whole statement, not just the code to be executed in parallel, so that it is clear what code is executed by whom.

VI. Boundary Crossing Invocations

This paper has introduced the notion of inhomogeneous thread types, with the ability of one thread (either mobile or stationary, and running in a core supporting some ISA), to create a thread of possibly another type (mobile or stationary), and/or in a core supporting a different ISA on possibly a different locale. Before trying to discuss a notation to allow this, an important discussion is what does the code look like that would need be compiled to perform such calls. Table II diagrams several combinations of parent and child types, and for both spawns ("non-blocking thread invocations") and conventional calls ("blocking"). The latter are relevant because when some boundary is crossed (locale, thread type, ISA), even a conventional function call may look like a remote spawn (consider a stationary thread calling a routine to be executed on a different locale). Such actions may also occur in codes for mobile threads where the code explicitly directs a change of locale, but with no change in the "thread" doing the execution. The numbers in Table II refer to code sequences as discussed as follows:

1) No special code for the transition is needed other than a normal procedure call/return.
2) Mobile thread created to migrate to target locale and "call" a stationary thread. Both parent and mobile threads block. Upon completion, stationary thread must return values to mobile thread to return values and unblock parent.
3) Mobile thread created. Parent stationary thread blocks. Upon completion, mobile thread returns to parent locale to return values and unblock parent.
4) Parent thread identifies an FCB, and spawns a child thread of correct type. Both continue execution with no interaction other than through FCB. Note that if child is mobile, it can migrate to correct locale.
5) Parent thread identifies an FCB, and spawns a mobile thread to migrate to target locale and "spawn" a stationary thread. Mobile thread may quit after spawn, or may block where child stationary thread lives. Parent stationary thread does not block. Upon completion, child stationary thread must either re-create a new mobile thread or unblock the blocked mobile child, which then returns to locale holding FCB to signal completion.
6) Parent thread migrates to correct locale, creates stationary child thread, and then suspends itself on that locale so that child thread can resume parent thread.
7) Parent thread identifies an FCB, migrates to target locale, "spawns" a stationary thread, and continues execution. Parent stationary thread does not block. Upon completion, child stationary thread creates a mobile thread, which then returns to modify the FCB and signal completion.

In many of these cases, mobile threads are at the heart of implementing the transitions, even when the threads doing the computation are stationary on both sides. The architecture of Fig. 2 is an ideal match for such implementations.

VII. A Possible Unified Language Model

This section suggests a unification that if present in a language would provide all the capabilities discussed in the last section. To crystallize the discussion, Fig. 3 suggests some possible syntax to incorporate the thoughts developed in the prior sections. Anything in italics is treated as verbatim keywords; anything in $< ... >$ represents a syntactic token.

A. Thread Spawning

A keyword like *spawn* in front of a statement block would create a new thread to execute the block in a non-blocking asynchronous manner. The material inside the [...] represents the combination of an FCB and an expression that specifies a creation affinity for the spawned thread. An intrinsic such as *MYFCB* might be used to simply specify the parent's FCB.

Fields in the FCB might identify the type of thread to be spawned, how many current children are still active, what to do when the last child reports in, where is the FCB for the parent of this family, etc.

TABLE II
INHOMOGENEOUS INVOCATIONS.

Type of Invocation:		Blocking				Non-Blocking			
Type of Child:		Stationary		Mobile		Stationary		Mobile	
Affinity of Child:		Same	Different	Same	Different	Same	Different	Same	Different
Parent	Stationary	1	2	3	3	4	5	4	4
	Mobile	6	6	1	1	4	7	4	4

A *team_forall* is essentially a for loop surrounding a nested set of *spawn*s, and creates a family of threads that perform the same code. The same FCB is used for all created threads. Using notation from UPC, the last argument in the loop header is an affinity expression for each thread. It is inside the loop because it may very well be a function of the loop variable.

Such a loop could also be used to start a family of SPMD threads, with the body of the loop the SPMD program.

B. Affinity-based Migration

Fig. 3 suggests use of a keyword like Chapel's *on* to specify an explicit migration, with an argument for affinity, and optionally a second keyword argument such as *fully_mobile*, *controlled_mobile*, or *stationary*. Use of *on* as a stand-alone statement has a similar effect, but controls the locale of execution of the entire rest of the thread's execution.

C. Affinity-based Work Allocation

The *spmd_forall* is formatted similarly to *team_forall* except that it doesn't spawn any threads, and thus doesn't need an FCB. The affinity expression is used by each thread that executes it to test loop iterations and identify just those for which the thread's affinity matches. This is the same as UPC's *upc_forall* and is there to support SPMD codes.

D. Atomics

The $<=>$ notation from Chapel is suggested to provide an atomic swap of two variables. As with normal C semantics, the value returned from the statement (and thus usable if the assignment is nested inside a bigger expression) is the value being placed in the variable. A second version is suggested that has an expression on the right side. In this case, the "swap" is with the value returned out of the assignment, and is the prior value of the left hand side expression. Thus $z = (x <=> (y = y + 1))$; saves the value resulting from the incrementation of y into x but atomically passes the value in x at the time of the store out to be saved in z.

The prefix *atomic* in front of a single assignment guarantees that there has been no change in the value in the left hand side variable from the start of the statement until the new value is written back into that variable.

The *atomic* prefix in front of a block of statement is very similar to Chapel, except that we optionally allow a list of variables to be defined whose values must be maintained before any updates (the read and write sets for the update). The reason is to allow far simpler implementation than the more general case, as discussed earlier.

spawn $[< FCB >, < affinity\_expression >]$
$< statement\_block >$

team_forall $[< FCB >]$ $(< loop\_iteration >,$
$< affinity\_expression >) < statement\_block >;$

on $[< affinity\_expression >] < statement\_block >;$
on $[< affinity\_expression >];$

spmd_forall$(< loop\_iteration >,$
$< affinity\_expression >) < statement\_block >;$

$< var > <=> < var >;$
$< var > <=> < expression >;$

atomic $< var >=< expression >;$
atomic $[< var\_list >] < statement\_block >;$

Fig. 3. Notional Syntax.

VIII. CONCLUSIONS

This paper has explored the kinds of threading that may be relevant if we move into an environment where there is no longer a single type of thread, and we want to be able to express a wide variety of threading paradigms within the same program. This heterogeneity is both in the possible ISAs of the cores on which the threads execute and in beginning the evolution of the kinds of threads suggested by active messages into full-fledged mobile thread types. As discussed here, mobile threads have a far-reaching effect on the structure of programs that wish to utilize them.

In the process of describing such unified paradigms, the concept of identifying "families" of threads and their "affinities" in an organized manner was surfaced.

There is certainly work left to be done before beginning to consider actual packages based on these ideas. As was done by the UHPC language developers, a significant set of sample programs should be developed to explore both the potential savings in program complexity and the additional considerations that must be made. Also a reference definition of an interface as suggested in Fig. 2 should be prototyped. The Emu architecture outlined in [16] is a good starting point.

One such consideration that was not touched on here is extended syntax for data declarations, especially for PGAS systems where there is a great deal of flexibility in how and where parts of large data structures find their affinity. Additional expressiveness for such things as how either mobile or stationary threads should approach different data structures is one such area needing additional thought.

IX. Acknowledgements

This material is based in part upon work supported by the National Science Foundation under Grant No. CCF-1822939, and in part by the Univ. of Notre Dame.

References

[1] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Tasirlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero multicore software research project. In *Proc. of the 24th ACM SIGPLAN conf. companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system, 1995.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[4] D. Bonachea. GASNet Specification, v1.1. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2002.

[5] J. Brockman, P. Kogge, S. Thoziyoor, and E. Kang. PIM Lite: On the road towards relentless multi-threading in massively parallel systems. Technical report, 2003.

[6] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge. A low cost, multithreaded processing-in-memory system, 2004.

[7] J. Brown, L. Porter, and D. Tullsen. Fast thread migration via cache working set prediction, Feb 2011.

[8] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language, 2004.

[9] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[10] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[11] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community, 2010.

[12] Cray. CRAY T3D System Architecture Overview Manual.

[13] W. J. Dally, A. Chien, S. Fiske, W. Horwat, R. Lethin, M. Noakes, P. Nuth, E. Spertus, D. Wallach, D. S. Wills, A. Chang, and J. Keen. Retrospective: The J-machine, 1998.

[14] W. Dayy and et al. Architecture of a message-driven processor, 1987.

[15] J. Dongarra and M. Heroux. Toward a new metric for ranking high performance computing systems. Sandia Report SAND2013 4744, Sandia National Labs, June 2013.

[16] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. B. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein. Highly scalable near memory processing with migrating threads on the emu system architecture, Nov. 2016.

[17] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access (extended abstract), 2015.

[18] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[19] S. E. Frost, A. F. Rodrigues, C. A. Giefer, and P. M. Kogge. Bouncing threads: Merging a new execution model into a nanotechnology memory, 2003.

[20] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1999.

[21] R. E. Hiromoto, O. M. Lubeck, and J. Moore. Experiences with the Denelcor HEP. *Parallel Comput.*, 1(3):197–206, Dec. 1984.

[22] Intel. Intel threading building blocks: (Intel TBB), 2015.

[23] R. Kessler and J. Schwarzmeier. Cray T3D: a new dimension for Cray Research, Feb 1993.

[24] P. Kogge. Of piglets and threadlets: Architectures for self-contained, mobile, memory programming. *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 130–138, Jan. 2004.

[25] P. Kogge. Tracking the effects of technology and architecture on energy through the Top 500, Green 500, and Graph 500, 2012.

[26] P. Kogge and T. Dysart. Using the TOP500 to trace and project technology and architecture trends, 2011.

[27] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the new normal for computer architecture. *Computing in Science and Engineering*, 15(6):16–26, 2013.

[28] P. M. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report CSE 2008-13, Univ. of Notre Dame, Sept. 2008.

[29] P. M. Kogge and D. R. Resnick. Yearly update: Exascale projections for 2014. Technical Report SAND2014-18651, University of Notre Dame, Sandia National Laboratories, Sept. 30 2014.

[30] P. M. Koggw and S. K. Kuntz. A Case for Migrating Execution for Irregular Applications, Nov. 2017.

[31] A. Kopse and D. Vollrath. Overview of the next generation Cray XMT, May 2011.

[32] P. A. La Fratta and P. M. Kogge. Models for generating locality-tuned traveling threads for a hierarchical multi-level heterogeneous multicore, 2010.

[33] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[34] V. Marjanovic, J. Gracia, and C. W. Glass. *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, chapter Performance modeling of the HPCG benchmark, pages 172–192. Springer Int. Publishing, Nov. 2014.

[35] R. Murphy. X-factors: The X-caliber approach to codesign, data movement, and pico-joules, Oct. 2010.

[36] R. Murphy. X-caliber and exascale grand challenge research summary, Sept. 2011.

[37] R. C. Murphy. *Traveling Threads: A New Multithreaded Execution Model*. PhD thesis, University of Notre Dame, Notre Dame, IN, USA, 2006. AAI3221330.

[38] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[39] J. Nieplocha, M. Krishnan, M. Krishnan, D. Panda, and et al. High performance remote memory access comunications: The armci approach. *Int. Journal of High Performance Computing and Applications*, 20:2006, 2006.

[40] B. Page and P. M. Kogge. Scalability of Hybrid Sparse Matrix Dense Vector (SpMV) Multiplication, July. 2018.

[41] J. T. Pawlowski. Hybrid memory cube: a re-architected DRAM subsystems, August 2011.

[42] A. Rodrigues. The UHPC X-Caliber project architecture, design space, and codesign, Nov. 2010.

[43] V. A. Saraswat, O. Tardieu, D. Grove, D. Cunningham, M. Takeuchi, and B. Herta. A brief introduction to X10 (for the high performance programmer), 2008.

[44] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation, 2017.

[45] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multi-processor performance on the Tera MTA, Nov 1998.

[46] S. Sridharan. *Compiler and Runtime Techniques for Software Transactional Memory in Partitioned Global Address Space Languages and Runtime Libraries*. PhD thesis, CSE Dept., Univ. of Notre Dame, Oct. 29 201.

[47] J. E. Thornton. *Design of a Computer: The Control Data 6600*. Scott Foresman & Co, 1970.

[48] S. Thoziyoor, J. Brockman, and D. Rinzler. pim lite.

[49] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, Mar. 2003.

[50] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation, 1992.