The Partitioned Parallel Processing Spaces (PCubeS) Type Architecture

Muhammad N. Yanhaona and Andrew S. Grimshaw

Table of Contents

Background

The beginning of the twenty first century observed few changes that greatly affected the way we do scientific computation. First, the uniprocessor processing speed reached close to its theoretical pick, driving the high

performance scientific community to rely almost exclusively on parallel processing. Second, large scale data analytics has become a common practice in many new domains of interest such as social sciences and economics. Finally, the continuing success of the hardware industry in providing cheap and improved processing capacity to the masses has achieved such a height that now one can own a parallel machine that is computationally more powerful than the supercomputers of just two decades ago and that has terabytes of storage capacity.

There is a two-sided effect in all these changes, on one hand the diversity of needs resulted in significant heterogeneity in present day hardware architectures, in both building block processing units and large scale massively parallel computing systems. On the other hand, algorithm design has become more and more architecture sensitive; so much so that recent papers in this field tend to claim to solve any particular problem on a specific architecture only.

Overall, there is a heightened need to be able to reflect the hardware in a program. By 'reflect' we mean the ability to properly exploit the features of an execution platform in a program. That necessitates the support for exposing those features along with their associated costs by the medium – the programming language under use – to be there to begin with. Unfortunately, we observe several limitations in conventional parallel programming paradigms in their support for exposing hardware features.

Some contemporary paradigms are stuck with their obsolete view of an execution platform as a Von-Neumann style sequential processor or a uniform collection of those, some others expose the features quite accurately but are so hardware-specific that they absolutely deny any portability, and the remaining others are plainly fostering a hardware agnostic high-level mode of programming.

Now, this report is not about programming languages and paradigms that might succeed in striking a balance between the need of hardware modeling; and the efficiency, expressiveness and portability of the language features. There is a precursor to all of these: a mechanism of describing present day heterogeneous architectures in a uniform way. That is the goal of a *Type Architecture.* The type architecture, a term coined by Laurence Snyder [33], is a facility description that standardizes the interface between the hardware and the programming language.

For any type architecture to be meaningful, it has to be applicable to most contemporary hardware architectures, able to accurately expose the costs of described features, and simple enough to be understood and utilized by an average programmer. In our proposed type architecture, the Partitioned Parallel Processing Spaces or *PCubeS* (a shorthand originated from three P's followed by an S in the initials), we strive to balance these requirements.

Before we engage into a deeper discussion on *PCubeS*, we should reflect on current trends in large scale parallel computing architectures to discover similarities within heterogeneity and need some background on type architecture to understand what it brings into the table. Subsequent sections elaborate on these two points before our discussion on *PCubeS* begins.

Current Trends in Parallel Computing Architectures

One of the most significant differences of present day parallel architectures, regardless of being large or small scale, from architectures of just a decade ago is that they tend to be constructed out of building blocks that are parallel computing units themselves. For large scale parallel systems, this trend means that purely shared memory such as SGI's Blacklight or purely distributed memory such as IBM's Blue Gene L systems of yesteryears are now extreme rarities. Machines like Blue Gene P and Q systems, next generations of L and dating only a few years back (2007 and 2012 respectively), or the Ranger supercomputer in TACC (2008) were already offering a hybrid computing environment connecting multicore processor nodes in a distributed manner. This trend is further intensified as recent machines such as *Stampede* and *Titan* have not only multicore CPUs but also hardware accelerators within nodes to offload computations from the former.

This push towards hybrid architectures is there not for any conceptual clarity or programmatic simplicity; on the contrary, hybrid architectures make it more difficult to achieve those objectives. The push is there due to memory and power wall barriers that become [7] major concerns in computer architecture since the beginning of this millennium. As these concerns are unlikely to go away in near future, proliferation of hybrid architectures should continue, and proper attention must be given to their deep processing and memory hierarchies in programs for effective utilization of available capacities.

Let us now briefly examine the constituent's and overall architectures of typical, present day large scale parallel computing platforms to better comprehend their hierarchical nature.

Node architectures

A typical supercomputer of current time has multicore processors, hardware accelerators, or both as a node. The number of processor cores usually ranges from 4 to less than 20. For examples, a IBM Blue Gene Q system uses 18-core PowerPC, *Stampede* 8-core Intel Sandy Bridge, and *Titan* 16-core AMD Opteron as their multicore CPU nodes. Sometimes the individual cores are simultaneously multithreaded such as the 4-way multithreading of PowerPC cores. That further multiplies the parallel processing capacity of the CPU.

These cores are general purpose execution units and typically have their own L1 and L2 caches and share an L3 cache among them. A memory controller connects an external RAM to the CPU to be equally accessible to all the cores. Figure 1 shows the block diagram of a 16-core AMD Opteron CPU as an illustrative example.

We already see a rich hierarchy in the architecture of a single CPU that should be taken into consideration while writing a high-performing program.

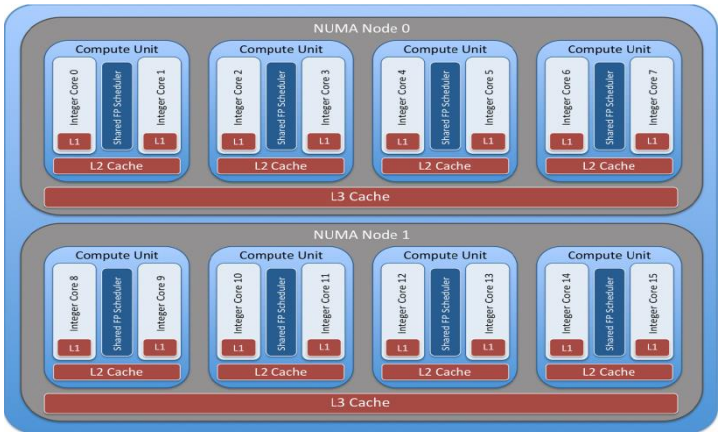

Figure 1: A 16-Core AMD Opteron CPU (Source: AMD Corporation)

At the same time, hardware accelerators that were initially popularized by gaming industry and computer graphics are becoming increasingly common in supercomputers. For examples, *Stampede* has an Intel Xeon Phi co-processor alongside 2 Sandy Bridge CPUs and *Titan* has an NVIDIA Kepler K20 alongside the AMD Opteron CPU in each compute node. The difference between a multicore CPU and an accelerator is that the latter has a large number of streamlined, simple cores running in parallel in a SIMD (Single Instruction Multiple Data) or SPMD (Single Program Multiple Data) manner as opposed to the independent processing model supported in the heavy, general purpose cores constituing the former. A large number of scientific computations have significant regularity and data parallelisms that these accelerators are particularly suitable for, explaining their popularity.

Just like a contemporary microprocessor, accelerators have a hierarchy in the organisation of their processing elements, and it is critical to give proper attention to that for efficient utilization of these hardware. For an

example, if we look at the construction of the NVIDIA Kepler K20 GPU that is been used in *Titan*, we see its scalar cores are grouped under 15 symmetric multiprocessors that work independent of each other but share the onboard DRAM and a small L2 cache (Figure 2 depicts the block diagram of the GPU).

Each symmetric multiprocessor (SM) can run upto 2048 threads by multiplexing them to its 320 scalar cores. An SM has a small, 64 kB, memory to be shared by the cores. Furthermore, these threads do not work independently; rather they get dispatched as groups of 32 lockstep threads that are called warps in NVIDIA terminology. Shared memory access alignment within the threads of a warp, and global memory access alignment among succesive transactions made from an SM are absolutely critical for good performance in this environment.

A similar, albeit different, hierarchical decomposition can be discovered in Intel's Xeon Phi co-processor; the only other accelerator currently been extensively used in high performance computing.

Figure 2: Block diagram of a NVIDIA Kepler GPU having 15 symmetric multiprocessors (Source: NVIDIA Website)

To summarize our discussion on node architecture, we see two points coming out as important. First, there are significant processing power and memory capacity available within a single node; and second, it is inappropriate to view a node as a flat collection of processing units. Apart from in programming large-scale machines, this trend has significance in the domain of personal computing too. As recent personal computers and workstations are regularly equipped with powerful multicore CPUs and one or more accelerators, it is possible to view them as good candidates for small and medium scale scientific computing. Consequently, programming frameworks that can simultaneously support both domains have the potentials for bringing great benefits.

Supercomputer Architectures

Turning our focus from individual nodes to entire supercomputers, the first noticeable thing is their sheer scales. *Titan* has 18,688; *Stampede* has 6,400; and *Mira*, a Blue Gene Q system, has 49,152 compute nodes. Given that the demand for more computation power always outstripped the capacity available, this trend towards larger and larger machines is unlikely to stop.

It is unthinkable to wire such massive number of nodes directly to one another. At the same time, connecting them in a uniform fashion through a sparse network does not give good communication performance. Therefore, a typical supercomputer has multistage interconnection network or networks that divide the nodes into densely connected subgroups that are further connected by high-in-bandwidth communication channels of progressively sparser topologies. Evidently, the interconnection network of a supercomputer has been subjected to a

hierarchical breakdown too. Nature of this breakdown can make a particular application more or less efficient in a particular supercomputer.

For example, the nodes of *Stampede* are connected by a two-level fat tree Infiniband interconnect [3]. Figure 3 illustrates the topology. It is obvious from the Figure that a communication between a pair of nodes within a single leaf should complete significantly faster than that between a pair of nodes belonging to different leaves.

In *Mira*, each compute rack or cabinet hosts 32 compute drawers with 16 drawers forming a mid-plane. A single mid-plane hosts 512 compute nodes that are electrically connected into a 5D torus topology, and beyond the mid-plane level there are only optical connections [1]. Again the differential nature of communication becomes evident.

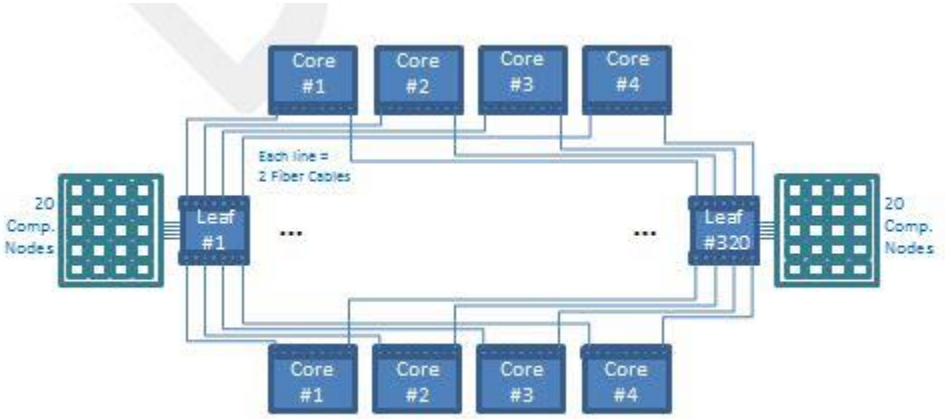

Figure 3: *Stampede*'s Interconnection Structure (Source: Texas Advanced Computing Center)

From the programming perspective, this suggests that treating nodes differently based on their relative positions in the interconnection network may provide performance advantage. Some recent research, e.g. one done on Blue Waters system on topology sensitive MPI codes, bolsters that assumption [29] – here we need to have some reservation though[1].

Type Architecture Fundamentals

The huge capacities of present day massively parallel architectures may appear impressive, but historically their computational power was never quite enough to satisfy the demand made by their contemporary applications. This is because most interesting scientific applications are quadratic or above in their runtime complexity. Therefore, only a modest improvement in problem size and running time can be achieved through a linear increase of processors that parallelism offers. The exponential growth of data in large scale data analytics while each technological advance merely doubling available machine capacities, has made this demand and supply disparity even more intense in recent years.

It is not that we were not aware of this problem before. It has been identified as early as in 80's. Snyder in his 1986's seminal article "Type Architectures, shared memory, and corollary of modest potentials [33]" points out

[1] Although it has been shown that if the nature of communications within a program exactly matches the interconnection topology of its execution platform then the potential for performance advantage is great, that is unlikely to happen for most programs [16]. Furthermore, graph mapping itself is an NP-complete problem; therefore, programming effort on such mapping even when it exists is prohibitive. Rather, we believe, considering positional difference at the group level such as intra-leaf and intra-plane as oppose to across-leaves and across-planes within programs is both manageable and beneficial.

that it is crucial to translate all of the capabilities of a parallel execution platform into useful computation, rather than losing much of that in implementation heat, to even modestly keep up with the growing demand of the applications. He argues for a hardware sensitive programming paradigm for writing parallel applications, and in so doing introduces the notion of a *Type Architecture*.

Snyder describes the endeavor of writing a program as a two steps translation process: from algorithm to program then from program to executable. The programmer is responsible for the first transformation and the compiler for the second. Overheads should be scrupulously avoided in both steps for the sake of efficiency. Given that the programmer did his best in writing the program, unwanted overheads may still arise due to limitations of the abstract machine model exposed by the programming language that works as the medium of communication between him and the compiler. Depending on how the language defines it, the abstraction can be prohibitively expensive so that he has to work against it to get his program run efficiently in a particular environment or it can be so low level that he can write programs for only a particular class of hardware.

There is a rift between high and low level programming techniques regarding the role their underlying abstract machine models play. Most high level languages present a simple abstract machine model of the execution environment to simplify coding and enhance portability of written codes, but that model provides close to no guidance for efficiently exploiting the features of the environment. Thereby, it is often difficult to get good performance in high level languages. On the other hand, low level programming techniques make the programmer deal with even minute details of the execution environment. Their abstract machine model is effectively the bare hardware. Hence efficiency can be attained, but with considerably greater programming effort and at the expense of program portability.

To resolve this tension between high and low level programming techniques and combine the best of both worlds, Snyder proposes to adopt an idealized machine model that should serve as the standard hardware-programming language interface, and thereby provides the foundation for the machine abstraction of any parallel programming language. This interface is called the *Type Architecture*. In other words, the type architecture is a description of the facilities of hardware. To be effective, the type architecture description should bear the following two characteristics.

- A type architecture must expose the salient architectural features of a hardware
- And, it must accurately reflect the costs of those features

It is important to understand the distinction between a type architecture and the programming model, formally known as the abstract machine model, of a programming language. For example, the Von Neumann architecture can be considered a type architecture for sequential machines. FORTRAN [25] and Lisp [24] present two different machine models on top of that. FORTRAN offers a programming style that fosters generic array operations; in contrast, Lisp offers a programming style that relies on recursive list manipulations. The type architecture here tells how the operations and primitives of these languages will be translated and how well they should perform in an execution platform. A successful programmer internalizes the cost of model-to-architecture translation and chooses his primitives and operations accordingly to write an efficient program.

To understand the different roles an abstract machine model and a type architecture play in programming from an alternative angle, consider the pseudo code of an algorithm. A type architecture description tells the programmer how to assess the efficacy of the algorithm on a target execution platform. The abstract machine model, on the other hand, tells him how to implement the pseudo code in a particular programming paradigm supported by that platform.

The further the abstract machine model goes away from the actual type architecture the more difficult it becomes for the programmer to make a correct assessment about the performance of his program. Consequently, the

algorithm he uses may be inappropriate for the underlying execution platform. This is not a problem when performance is not a major concern, and that is often the case for sequential programming paradigms. That is, however, definitely not the case for parallel computing.

Consequences of a Mismatch

A mismatch between the assumptions of the abstract machine model of a parallel programming paradigm and the actual behavior of the features of the execution platform can lead to severe performance degradation. For example, programming in a non-uniform shared memory machine such as Blacklight [23] using a uniform shared memory programming paradigm may result in great performance loss due to memory access inefficiency. Similarly, a distributed memory programming paradigm that treats communications among processes as pure information exchanges that are devoid of any performance characteristics can be grossly inefficient as it leaves the programmer in dark about the proper choice of message size, nature, and frequency for a particular platform.

The ability to fully control the communication characteristics of a program is, probably, the most important factor behind the success of current de-facto standard of parallel computing, the Message Passing Interface (MPI)[36]. However, in the absence of a clear type-architecture foundation, writing good MPI codes often becomes equivalent to knowing a lot of ad-hoc performance tweaking. Therefore, it can be difficult for an average programmer to write high-performing MPI programs for typical building-block scientific problems. For entire applications the problem is justifiably more severe. This problem was already evident when most supercomputers were plain distributed memory machines. The advent of accelerators and multicore CPUs has made this problem only more intense in current decade.

The stakes are always high that we extract good performance out of the machines anyway possible. Therefore the common practice in high performance computing has become to 'program against the machine [33].' MPI has been coupled with various shared memory and offloading computing models such as Pthreads, OpenMP, CUDA, and Cilk [14, 8, 9, 26] that often violate MPI's model of communicating sequential processes. To clarify this point through an example, if two different MPI processes running independently on two CPU cores vie for the same GPU resource during offloading computations using CUDA, the model of isolated processes evidently falls apart. In such a situation, the programmer has to rely on intuition rather than on the machine model to determine what should run efficiently and what not.

For a typical programmer, a hybrid programming model is significantly more difficult to deal with than a singular, holistic language alternative regarding reading, writing, and debugging a program. Unfortunately, that covers almost all aspects of programming. Furthermore, a hybrid model is particularly antagonistic to the idea of portability of knowledge.

To elaborate on the last point, a programmer versed in C [20] can easily rewrite his programs in FORTRAN once he learns latter's syntax. His original programs were not portable but their underlying logic definitely is. Regrettably, a programmer knowing how to write good MPI + Pthreads hybrid programs, on the other hand, may miserably fail if he is given an MPI + CUDA platform. This is because unlike the first sequential programing example, the peculiarities of underlying execution platforms are closely tied with the behavior of low level primitives provided by Pthreads and CUDA in the second. In other words, in the latter example, learning not only a different syntax but also a whole new and quite different abstract machine model is required.

Thus a lack of a unifying modeling framework, a facility description standard, aka a type architecture is badly felt here. As a side note, recent high-level parallel languages such as Chapel [10] or X10 [12] that are striving to be viable alternatives to dominant low-level programming paradigms also need an acceptable type architecture background for their abstract machine models. In the absence of a type architecture, features like 'places,'

'locations,' and so on that are proposed by them for hardware sensitive reasoning may become more like abstract concepts for grouping program segments.

Related Work on Type Architecture

Few parallel type architectures have been proposed so far and there is none that been universally accepted. Some are not even formally presented as type architectures. We mention all of them here for the sake of completeness.

PRAM or paracomputer was proposed in 1980's by Schwartz [31] and implicitly accepted by the theory community as a type architecture. PRAM's view of a parallel execution environment is of an arbitrarily large number of identical processors sharing a common memory where any number of processors can read and write simultaneously at a unit cost. The problem with paracomputer is that it is unrealizable. Although it makes designing parallel algorithm easy, Snyder deftly explains how its impracticality can easily fool someone into unwittingly choosing sub-optimal algorithms [33].

SIMD or Single Instruction Multiple Data type architecture [32] was temporarily popular in 1970s and 1980s during the time of machines like CM-1, CM-2, ILLIAC IV, SIMDA, etc. A SIMD machine is described as a computer system that has a controller unit and a fixed number of processing elements that are connected by some interconnection network. The controller unit broadcasts an instruction to the processing elements and the processing elements that are active at that time all execute that instruction on different pieces of data. Present day accelerators' Single Program Multiple Data (SPMD) execution model can be tracked back to a SIMD origin, but pure SIMD is no longer a candidate for type architecture for modern machines.

CTA is offered by Snyder as an alternative to PRAM as the first Candidate Type Architecture, hence the name. CTA describes a parallel hardware as a finite set of sequential computers connected by a fixed, bounded degree graph, with a global controller [33]. Poker [27] and later ZPL [11] are two languages that are developed using this abstraction and the latter saw some success in few hardware platforms too. Even the programming model of MPI can be viewed as loosely based on CTA. CTA may be a good choice for describing purely distributed memory systems of last decades, for present day hybrid machines though, it is plainly inadequate.

Systolic Architecture gained some momentum in late 1980s and early 1990s [21]. CMU's Warp machines are good examples of this architecture. In a systolic system, data flows out from the computer memory in a regular order, passes through a series of processors, and then returns back to the memory. Multiple independent series of processors, aka assembly lines, work simultaneously to provide parallelism. Systolic systems offer a data-driven programming paradigm that is fundamentally different from the instruction-driven paradigm of Von Neumann architecture that is at the heart of present day parallel machines. Since multiple operations can be done on a single piece of data, systolic systems are better able to balance computation with IO. Regardless, systolic architecture is suitable only for highly specialized systems – not for general purpose parallel computing – as it is difficult to map most algorithms to its restrictive model.

LogP and LogGP type architectures are proposed for distributed memory machines [13][5]; and unlike the previous two examples, they take a parametric approach to type architecture description. LogP describes a machine in terms of four parameters: communication latency (L), communication overhead (o), minimum gap in successive communications (g), and finally processor count (P). Later the proponents added another parameter (G) for capturing long message communication bandwidth to form LogGP. Both of these architectures suffer from their lack of relevance in programming. It is possible to gauge the expected performance of an algorithm to a great precision using LogP or LogGP, but how a programming language's features can embody these parameters is difficult to imagine.

PMH describes a parallel computer as a tree of modules that hold and communicate data with only leaf modules being able to do computations [6]. Each level in the hierarchy has four attributes: the size of memory blocks, memory capacity in terms of blocks, communication delay in transferring a block between a parent and a child, and child count. Although PMH is better suited than the previous examples in capturing the hierarchical nature of recent parallel architectures, its lack of concern for processing capacity and elimination of direct module-to-module communication within a level make it too conservative a type architecture for high performance computing.

PGAS is proposed more as a programming model than as a type architecture [34], but essentially it serves both purposes. PGAS or Partitioned Global Address Space describes a shared memory execution environment composed of sequential processors. The shared memory is, however, not a conventional single unit; rather it is the collective sum of the local memories of individual processors. Each processor has access to the entire memory but access time varies depending on the locality of the reference. PGAS is suitable for describing multicore CPUs, and thus a good number of languages have been developed and currently under development that use this abstraction [15, 17, 28], but it is much less applicable for large scale parallel machines.

OpenCL or Open Computing Language offers a programming model and a hardware abstraction both at the same time [35]. Its type architecture description is of a host CPU and any number of attached OpenCL 'devices.' Each such device contains one or more 'compute units' each of which holds one or more SIMD 'processing elements' to execute instructions in lockstep. There are four types of memory: a global device memory, a small low-latency read-only memory, a per unit shared memory, and finally a per element private memory. One or more of these memories can be missing in a particular platform. This is quite a rigid description to be general purpose. Hence OpenCL is restricted to mostly hardware accelerators, and to some cell processors and multicore CPUs.

Partitioned Parallel Processing Spaces

The formal description of our proposed type architecture, The Partitioned Parallel Processing Spaces (*PCubeS*), is as follows.

> *PCubeS is a finite hierarchy of parallel processing spaces each having fixed, possibly zero, compute and memory capacities and containing a finite set of uniform, independent sub-spaces that can exchange information with one another and move data to and from their parent.*

Before we define parallel processing spaces in more detail, we have two things to add on Snyder's notion of type architecture that are fundamental to our proposal and essential to understand the elements of *PCubeS*.

Programmability: Snyder used the term 'structural feature' and 'facility' interchangeably. This is because he was more concerned about describing hardware features than about how they can be put to meaningful use in a program. We believe this approach of 'describe first then derive programming models' is not right as the type architecture is intended for standardizing the hardware-language interface, and an interface design has to take into account the demands of both sides. Thus, in our opinion, a type architecture description should rather focus on the programmatic usage of hardware's structural features than on their actual working principal. To make the idea concrete through an example, the description should not bother if a vector is implemented as a pipeline or a SIMD lane. That the described hardware has a vector should be the primary concern. Furthermore, if the hardware is good for multiple modes of programming – as many contemporary supercomputers are – there may be multiple type architecture instances for it describing its features from different programming perspectives.

Parameterization: we think parameterization is needed for any type architecture description to enable a programmer to make accurate estimation of the performance of his algorithm for a particular input set. To understand why it is important, imagine a CTA description of a parallel hardware where processors are connected through a complete binary tree network configuration. From the description, a programmer can deduce that on an average communication between a pair of processors should take steps logarithmic to the number of processors. To determine how frequently a processor should communicate and what should be the individual message sizes to balance computation with communication, however, he needs to know the actual latency and bandwidth of the network. Lack of such information hurts CTA's applicability. On the other hand, a type architecture description in terms of such values such as in the case of LogP should be avoided as that obstructs programmability. We believe a PMH like description that has a core model parameterized by actual values is the proper middle ground. So the programmer can use the model to design the algorithm and the parameter values to assess a program's runtime performance.

Elements of PCubeS

Let us now characterize the elements of *PCubeS* to better understand how this type architecture can be used to describe present day parallel machines.

Parallel Processing Space (PPS): the notion PPS is used to describe any part of a parallel hardware or the entire hardware itself where a computation can take place. For example, in case of a multicore CPU, both the CPU and its individual cores are spaces, the latter lying within the former as sub-spaces. From the perspective of the former, the latter are its *Parallel Processing Units (PPU)*. As part of a computation, a space can perform two fundamental operations: floating point arithmetic and data movement.

Capacity of a space: parallel processing and memory access capacities of a space are defined as the number of corresponding fundamental operations that can be done in parallel. For example, a SIMD thread group within an NVIDIA GPU's symmetric multiprocessor has 32 operations per clock cycle as its parallel processing capacity as 32 threads run in lock-step within each group. Note that the actual hardware implementation of a parallelization feature is not important; rather its programmatic manifestation is. This allows us to treat cores, SIMD lanes, vector pipelines all in the same way. We believe the speed of instruction execution is enough to expose their efficiency differences.

The memory, if it exists in the space, is characterized by its size and the number of transactions to and from that can be done on it in parallel. We use the term transaction to represent a single load/store operation or a communication. A transaction is further characterized by the volume of data it caries and its latency. For example, if a read/write operation by a thread within a dual hyper-threaded CPU core involves 8 bytes of data and takes 15 cycles to complete and operations from both threads can take place simultaneously then the core's parallel memory access capacity is 2 transactions of width 8 bytes and latency 15 cycles.

Uniformity and Independence of Sub-spaces: given that an execution platform is viewed as a hierarchy of spaces, it is important that there is a guiding principal for breaking larger hardware components into smaller components to form sub-spaces – otherwise, any hardware can be described as a flat *PCubeS* hierarchy having a single space only. In that regard, we use uniformity and independence as the defining factors. Uniformity requires that not only all sub-spaces of a particular space have the same processing and memory capacities but also their information exchange with one another and data movement to and from their parents have the same average values for transactional attributes. Meanwhile, independence requires that operations done by different sub-spaces are independent. Whenever both of these requirements are met for a hardware component then we divides its capacities into sub-spaces; otherwise we do not.

To understand how this rule works in practice, consider a supercomputer node having two 8-core CPUs. We cannot view a node as a 2-space hierarchy where the node is the higher space containing 16 sub-spaces, one for each core. This violates the uniformity requirement due to difference in intra-CPU and inter-CPU information exchanges among cores. There should be another space in-between that represents the individual CPUs to bring uniformity into the hierarchy. Similarly, the 32 lock-step threads of an NVIDIA GPU thread block cannot form sub-spaces despite being uniform. This is because their operations are not independent.

Information Exchange: it is important that we characterize interactions between sibling spaces as information exchanges instead of as communication or other platform specific term. This allows us to treat shared memory and distributed memory systems in a uniform way. Furthermore, within the umbrella of distributed memory architecture, different execution platforms may have different implementations of communication mechanism. The use of an abstract term enables us to dissolve these differences. An information exchange is characterized solely by its latency. We believe that is enough to capture the efficiency differences among different modes of interaction.

For example, if we consider a shared memory environment of four CPU cores sharing an L-3 cache. Then information exchange between a pair of cores is equivalent to the sending core writing data in the cache and receiving core subsequently reading it. The only difference between such an interaction and a read following a write by a single core is that there should be a mutex operation signaling the completion of writing by the former. So the latency of information exchange in this case is the latency of the mutex operation. For another example, in a distributed memory system supporting two-sided communication, the latency of information exchange would be that of handshaking added to the latency of actual data transfer. For a system with one-sided communication, on the other hand, the handshaking cost is replaced by the cost of setting a flag in the receiver's memory.

Data Movement: unlike in the case of information exchange, in characterizing data movements between a parent space and its sub-spaces all three transactional attributes – latency, width, and bandwidth – are needed. Transaction latency and width are used to understand the cost of a single data motion between the parent and one of its children, and transaction bandwidth is used to determine how many data motion operations can take place in parallel. For example, each symmetric multiprocessor (SM) in an NVIDIA K-20 GPU can read and write global memory at a maximum chunk size of 128 bytes that takes around 100 to 300 clock cycles. Assuming all 15 SMs in the GPU can initiate a global memory operation at the same time, the data movement between the sub-spaces representing individual SMs and the space representing the entire GPU has transactional latency of 100 to 300 cycles, width 128 bytes, and bandwidth 15.

When the space under concern represents a distributed memory segment of the hardware, transactional attributes need to be determined from the capacity of the communication channel and the settings of the communication protocol. For example, if we consider the space represented by a leaf in the fat-tree network of *Stampede* as shown in Figure 3 then the 20 nodes are its sub-spaces. In this case, transaction bandwidth for data movement is the number of concurrent flow that can go in and out of the leaf router, latency is self-explanatory, and transaction width is the maximum amount of data that each packet within a flow can carry.

Design Principles

In designing *PCubeS*, we adopt a few of design principles that have great programmatic significance. We would like to draw readers' attentions to these principles now.

Caches as Memories

PCubeS treats hardware managed caches as if they were programmable memories. This treatment of caches is unusual but, we believe, extremely important to capture and expose the efficiency of present day multi-level cache

based memory designs. Contemporary multicore CPUs and accelerators have significant capacities in their caches, and often time effective utilization of these caches becomes the determining factor for good performance. For example, a single core of an Intel Xeon Phi Co-processor has 512 KB of L-2 cache and the latency ratios between accessing that cache and accessing the on-board DRAM is about 1:10. Evidently, ignoring the L-2 cache is not a good choice.

On the other hand, treating a cache as a memory may raise the concern that the type architecture is describing a feature in a way that differs from its true nature. We believe that this is not a problem for two reasons. First, a cache's size is interpreted as the memory capacity in its *PCubeS* interpretation. That suggests that programs have to be written with the assumption that at runtime the memory can hold up to the cache size amount of data. Although actual loading and unloading of data to and from the cache is beyond programmatic control, since the working set can fit into the cache, it will behave more or less like a memory. The compiler should be able to generate addresses so that cache conflict is minimized or eliminated altogether.

Second, replacing the caches by programmable memories, or allowing a hardware switch for software controlled management of caches is not difficult to implement in the hardware. Such provisions can bring great performance benefits. As a type architecture is a vehicle for not only describing existing architectures but also for guiding future architecture developments, emphasizing programmatic manipulation of the caches, we believe, is the right approach to take.

Exposing caches as memories raises another question. Which memory should be exposed when a space has multiple caches and/or memories? This is an important concern, given that a space in *PCubeS* can have only a single memory. At the end, the hardware manufacturer is responsible for making the correct choice in that regard. Nonetheless, our opinion on that can be expressed as two generic rules

- The largest cache/memory should be exposed as the memory of a space as it is generally more important to handle the largest capacity more efficiently.
- Information exchange among sub-spaces within a space should take place through the closest/smallest memory as it generally represents the fastest path of communication.

If there is some definite advantage in exposing more than one memory then an alternative is to break the single space into a linear hierarchy of spaces with each space holding a memory. Again, the final decision is up to the hardware manufacturer.

Contention Oblivious Modeling

It is noticeable that *PCubeS* does not model contention. We recognize that contention is an important issue in parallel hardware design. For example, tree saturation or hotspot contention problem [30] in interconnection networks is so significant that it has become a standard practice to have separate communication network/channel in supercomputers to avoid such contentions. Memory bank conflict in multi-banked memories is another source of considerable performance loss in many programs.

One can argue that a CTA like description may be better suited in addressing hotspot contentions due to its emphasis on the specific structure of the interconnection network, or PMH is better suited for memory bank conflicts as it has the notion of memory blocks; but we argue otherwise because contentions are very application specific problems and just focusing on hardware features is not enough to deal with them. Therefore including features like memory bank size, memory stride, or network topology in the type architecture description may not pay off. Rather there is a change that they will further confuse the programmer.

Does our approach introduce a potential gap between perceived and actual performance of a program written over the type architecture abstraction? The chance is there as it exists in all other type architectures, but we believe its probability can be greatly reduced using standard programming techniques. For example, the use of primitives such as reduction and scatter-gather are effective in avoiding most hotspot contentions. Similarly, a compiler can do a lot to reduce memory bank conflicts when it has significant control over memory management of a program. Given that we are arguing for type architecture based programming paradigms, we envision that language primitives and compilers will play bigger roles in dealing with problems like contentions.

Symmetrical Space Hierarchy

As we have mentioned earlier, an ideal *PCubeS* description requires that sub-spaces of a space are uniform. This rule implies that sibling spaces at any level of the hierarchy have an equal number of sub-spaces. In other words, the hierarchy is symmetrical. We emphasize symmetry because, in our opinion, a symmetrical system is significantly easier to program than an asymmetrical system.

Many large scale parallel architectures are inherently symmetrical, e.g., *Mira* Blue Gene Q supercomputer. Furthermore, most accelerators and multicore CPUs that are been used in supercomputers have symmetrical organization of internal processing elements and memory modules. Therefore, *PCubeS'* restriction of sub-space uniformity readily applies.

Some hybrid systems, however, cannot be described as symmetrical hierarchies. For example, a compute node of *Stampede* supercomputer has two 8-core CPUs and a 61-core accelerator/co-processor. The accelerator cores can work just like the CPU cores or can be used to offload computations from the latter. Unfortunately, in both programming paradigms sub-space uniformity is directly inapplicable. The first case fails because accelerator cores are widely different in their capacities from the CPU cores. The second case fails because accelerator cores cannot be uniformly distributed under the two CPUs.

Note that *Stampede* supports CPU-only and accelerator-only programming models. Due to the symmetry of the interconnection network, the whole hardware can be described as a *PCubeS* instance for such usages. It is only when all resources of a node are used simultaneously problem occurs. Therefore, *PCubeS* is not appropriate for all parallel architectures or their supported programming paradigms. Nevertheless, often times a conservative *PCubeS* estimate can be drawn by merging sibling spaces, dividing space capacities, and often discarding some spaces altogether.

To give an example of how this can be done, consider the CPU-to-accelerator offloading computation model for a *Stampede* node. A block diagram of that node is given in Figure 4.

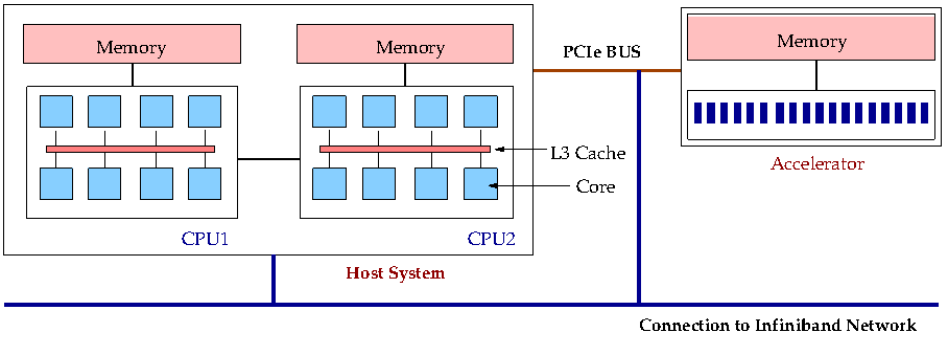

Figure 4: Block Diagram of a *Stampede* Node

In the offloading paradigm, individual cores can offload small pieces of computations to the accelerator from both CPUs, or the two CPUs can offload two relatively larger computations. This gives rise to two sub-cases, and both

can be conservatively described using *PCubeS* by ignoring some accelerator cores as shown in Figure 5(A) and (B). Note that in the second case, only 48 of the 61 accelerator cores can be used in a program.

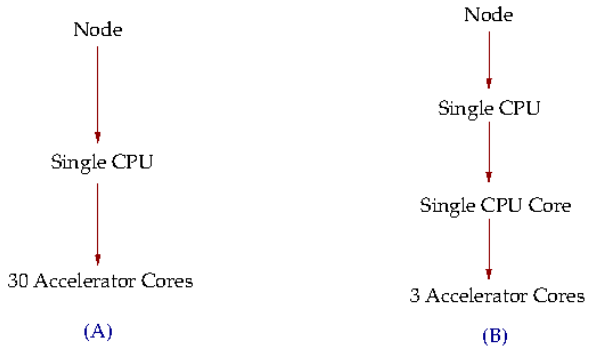

Figure 5: Two Alternative Symmetrical Breakdowns of a *Stampede* Node

Whether a conservative estimate worth consideration or not depends on the hardware and its expected usage, and the hardware manufacturer should decide. In the case of a *Stampede* node, the second description seems to be quite wasteful; thereby should be avoided. Despite this kind of problems, we stick to the uniform sub-space requirement because of programming simplicity.

Furthermore, we would like to encourage parallel hardware manufacturers to adopt symmetrical designs through *PCubeS*. In our opinion, an asymmetric organization of components makes it difficult to reason about actual runtime behavior of a program. For example, all 61 cores of the Intel Xeon Phi accelerator/co-processor can be utilized by the 16 cores of the two Sandy Bridge CPUs in a *Stampede* node, but it is difficult to imagine any offloading scheduler for the co-processor other than a work-pool for such usage. A symmetrical decomposition with 48 or 64 co-processor cores provides more scheduling alternatives. Common wisdom suggests it should not be difficult to restrict the number of cores to 48 or increase it to 64 in the hardware.

Mapping Hardware Architectures to PCubeS

If we compare *PCubeS* with other previously proposed type architectures; we see many machines described by CTA, SIMD, PMH, and PGAS can also be described using *PCubeS*. Most supercomputers of yesteryears such as purely distributed machines and purely shared memory machines are instances of *PCubeS*. Many clusters of workstations and even clusters of such clusters fall under *PCubeS* category. Most present day supercomputers that have only multicore CPUs as nodes such as Blue Gene P and Q systems are *PCubeS* instances. More complicated cases are the ones that have both multicore CPUs and accelerators within nodes. *PCubeS* can describe many of them too. To show how this can be done, we take the Titan Supercomputer [4] of Oak Ridge Leadership Community Facility as a case study for *PCubeS*.

The Titan Supercomputer: a Case Study

Titan has 18,688 compute nodes. Each compute node contains a 16 core AMD Opteron CPU and an NVIDIA Tesla K20 GPU. Two nodes share a single Gemini interconnection router and the interconnection network connects these routers in a 3D torus topology. The underlying philosophy in *Titan*'s design is to let the GPUs do the heavy lifting of a scientific application and use the CPUs mostly as coordinators of activities in the GPUs. Within that paradigm, there are two use cases to consider for a node.

1. A single activity offloaded to a GPU by its accompanying CPU consumes entire GPU capacity
2. Individual cores within a CPU offload relatively smaller, parallel computations to the GPU

Given that GPUs are not efficient in handling irregular parallelism, for some applications, doing the heavy lifting within the CPUs may be the right approach. This gives us the third programming paradigm for the node

3. CPU cores compute pieces of a parallel computation and coordinate their results

The network connecting the nodes is only a medium for communicating intermediate results of computations generated by individual nodes running under one of the above three paradigms. Note that *PCubeS* does not allow different programming paradigms in different nodes to coexist simultaneously nor does it allow a node to have multiple different sub-spaces. This is a limitation of our type architecture. If such behavior is expected, this limitation can be partially circumvented by dividing the capacity of the entire hardware into independent groups of nodes where each group runs a different programming paradigm.

What features of the GPU and CPU of a node the type architecture should expose and how they should manifest in the description very much depend on the choice of programming paradigm. But before we present the salient features of a node from the perspective of the supercomputer as a whole, let us derive the *PCubeS* descriptions of the multicore CPU and GPU when they are treated in isolation.

PCubeS Mapping of AMD Opteron CPU

If we refer back to the block diagram of a 16 core AMD Opteron CPU [19] given in Figure 1, we see that it has a non-uniform memory access (NUMA) model. Individual cores have their exclusive 16 KB L-1 cache, a pair of cores share a 2 MB L-2 cache, a group of 8 cores share a 8 MB L-3 cache, then all cores have uniform access to a 32 GB main memory. CPU cores run at a clock frequency of 2.2 GHz and data paths are 64 bits wide.

Although there are 16 cores in total, two cores sharing an L-2 cache share a single floating point execution unit. Since *PCubeS* measures compute capability in terms of floating point instruction density, in the lowest level of the hierarchy, the processing speed for each of the 16 spaces becomes half of the clock speed, i.e., 1.1 GHz; and the parallel processing capacity is 1 operation per cycle. The memory of a Space-1 unit is the L-1 cache. The immediate higher level of the lowest one is composed of pairs of Space-1 PPUs that share a single 2 MB L-2 cache. Each space in the next level combines four Space-2 PPUs and owns an 8 MB L-3 segment. The final space represents the entire CPU comprising two Space-3 PPUs sharing the 32 GB main memory. A pictorial depiction of the *PCubeS* description is given below in Figure 6.

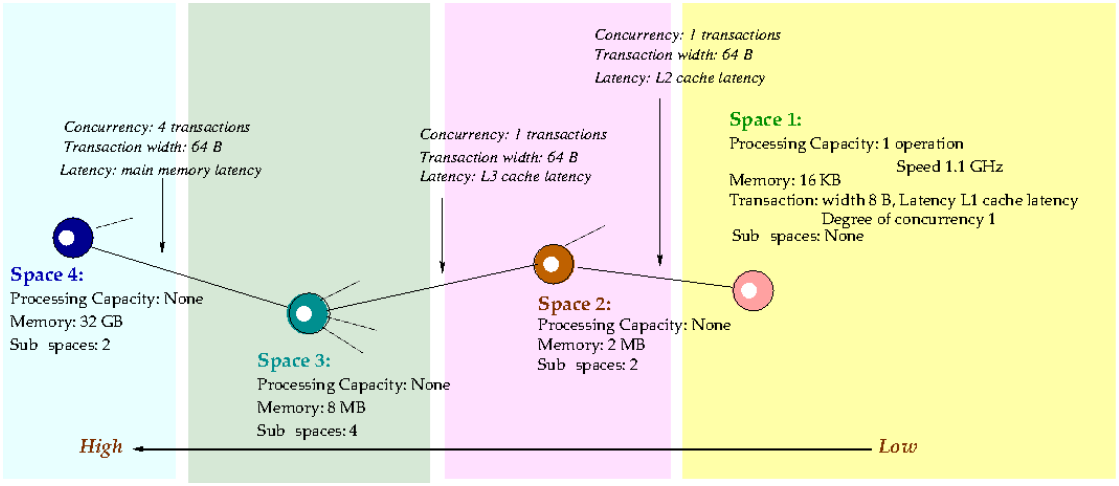

Figure 6: PCubeS Model of AMD Opteron Showing Only 1 Component Par Space

In the above model, the latency of information exchange is not shown because we do not know the cost of doing atomics in the hardware. Furthermore, memory access capacity is irrelevant for a space that has no processing

power. In such cases only the memory size matters. Only one data movement operation at a time can take place between a parent and its sub-spaces up to level 3 as there is only one 64 bits wide data path. Meanwhile, 4 concurrent data movement operations can take place between Space-4 and Space-3 as the main memory is quad channeled. Finally, transaction width is 64 bytes along each arc as the cache lines are 64 bytes long.

PCubeS Mapping of NVIDIA Tesla K20 Accelerator

A Tesla K20 GPU [22], shown in Figure 2, has 2496 streaming cores running at 706 MHz clock speed and distributed within 15 streaming multiprocessors (SM). It has a 6GB on-board DDR2 RAM as the main memory unit. Shared memory per SM is 64 KB but only 48 KB of it is accessible to programs. The streaming cores run in lock steps within each SM as a group of 32 threads known as warps. If we want to use the maximum amount of shared memory programmatically, each SM can roughly run up to 16 warps. Each shared memory load/store operation can process 16 32bit words and a global memory load/store is twice that size. The *PCubeS* description of the hardware depicts a 3 levels space hierarchy. Figure 7 illustrates this description.

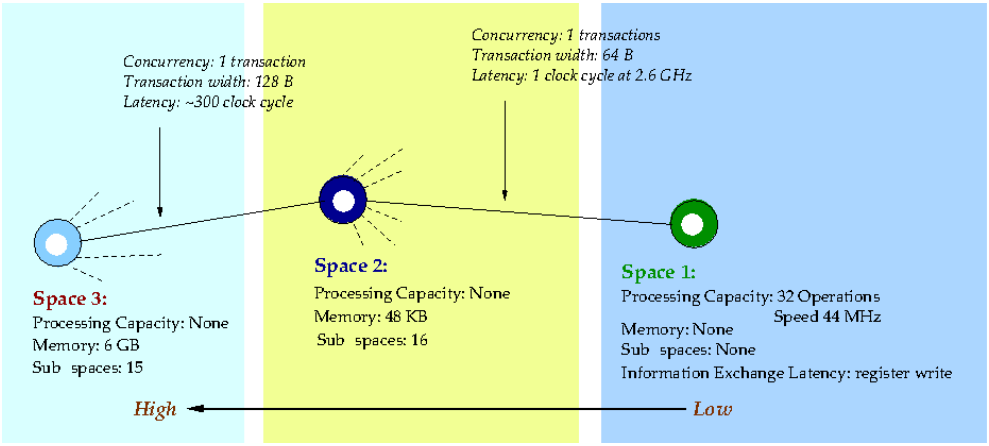

Figure 7: PCubeS Model of NVIDIA K20 Showing Only 1 Component Par Space

A warp represents a Space-1 unit. Parallel processing capacity of a Space-1 is 32 operations per cycle. The clock speed is, however, only 44 MHz instead of 706 MHz as warps are executed as pipeline instead of concurrently and there are 16 of them. This clock speed setting in the *PCubeS* description is in contrast to NVIDIA's advertised value as *PCubeS* is focused on actual performance characteristics of warps rather than mere numbers and structural details. A warp has no memory. This indicates that the result of any computation needs to be moved to the closest space with a memory, which is in this case the parent Space-2.

A Space-2 represents an SM and holds 16 Space-1 PPUs as sub-spaces. Only one Space-1 PPU can transfer data to its parent Space-2 at a time due to the pipelined nature of warp execution. So there is no concurrency, but the latency is minimal. A transaction carries 64 bytes of data and takes only one memory cycle to finish.

Finally, there is only one unit in Level-3 of the hierarchy. A single Space-3 PPU represents the entire GPU. It has 6 GB memory and holds 15 Space-2 PPUs as sub-spaces.

The *PCubeS* description of the accelerator resembles the hardware abstraction popularized by NVIDIA's CUDA programming model that also has 3 levels. Nonetheless, there are important differences to discover as we examine the parameters. We see that *PCubeS* makes the limitations of the hardware more explicit. For example, in CUDA, threads of a warp can diverge and execute different instructions. This gives a programmer more flexibility. In reality, however, divergent paths are executed sequentially. The *PCubeS* description makes divergence impossible by coupling the threads together. Such divergences in a program then need to be implemented as sequential streams of less degrees of parallelism. So the programmer is aware of his wastage of processing capacity. For

another example, data transfers between global and shared memory happen in chunks of 128 bytes, but CUDA threads can issue arbitrarily small and irregular requests to the global memory. In *PCubeS*, any data transfer between global and shared memory is mediated by the Space-2 unit under concern that makes such irregular requests impossible.

To summarize, *PCubeS* provides a more restrictive but accurate representation of the hardware than the CUDA machine model – with some exceptions[2]. As a type architecture description and an abstract machine model are not the same the comparison is not exact, but our point is that a programming paradigm whose machine model resembles the *PCubeS* description more than that of CUDA is more likely to guide the programmer in the right direction regarding efficient use of the GPU hardware.

PCubeS Mapping of Titan

Given that three different programming models are supported by the supercomputer, there should be three different *PCubeS* descriptions for it. In the easiest case, where only the multicore CPUs can be used, we get the description depicted in Figure 8.

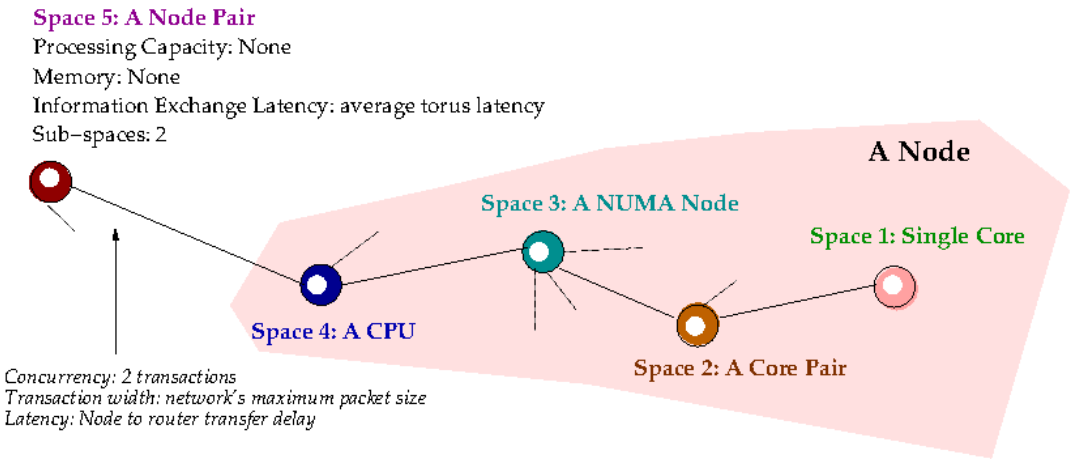

Figure 8: First PCubeS Description of the Titan Supercomputer

On top of the node hierarchy presented in Figure 6, we have one additional level representing the pair of nodes sharing a Gemini interconnect router. There are 18,688/2 = 9,344 such Space-5 PPUs in the machine. Information exchange latency between a pair of Space-5 PPUs is the average latency of a packet transmission in the 3D torus interconnection network. Notice the un-rootedness of the hierarchy. This is expected as there is no global controller or common aggregation point for the nodes of the supercomputer.

Let us now consider the case where the CPU within a node works as a coordinator of computations offloaded to its accompanying GPU and individual computations are large enough to consume the entire GPU capacity. In this scenario, only one core within the CPU does any useful work – the rest remain idle. So the spaces in upper three levels of the 4 levels hierarchy presented in Figure 6 become memory only spaces with branching factor 1. The GPU representation remains as it is. Figure 9 illustrates the *PCubeS* description for this scenario.

One important concern in describing this kind of hybrid environments is 'how to define the transaction attributes for data transfer between the CPU and accelerator.' Note that although Space-4 (a Core) is the parent of Space-3 (a GPU) in the *PCubeS* description, actual data transfer takes place from Space-7 to Space-3. When moving data out of the GPU to CPU core, data flows from Space-3 to Space-7 then from there to Space-4. As *PCubeS* is silent

[2] Additional memories such as constant or texture memories that are visible in CUDA cannot be used in *PCubeS*.

about the actual mechanism of data transfer this is not a problem, but one have to define the transaction attributes appropriately to derive a conservative estimate for the communication channel.

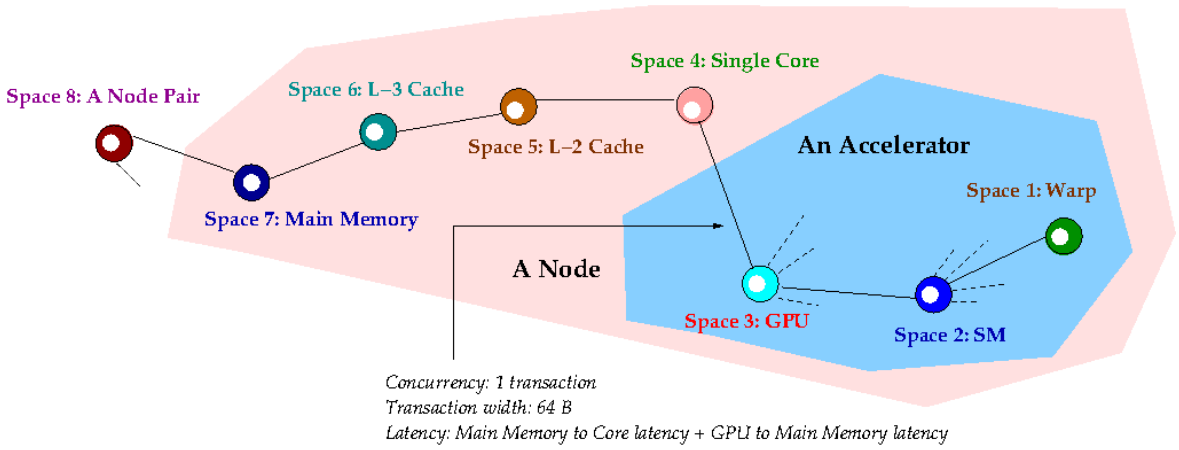

Figure 9: Second PCubeS Description of the Titan Supercomputer

The consistent rule for deriving a conservative estimate is: set the transaction width and concurrency to that of the least capacity link along the physical data transfer path, set the latency as the sum of the latencies of individual links, and if there is any shared link along the path then divide its capacity uniformly. For *Titan*, this gives us 1 concurrent transaction, 64 byte transaction width, and latency of main memory to L-1 cache transfer augmented with main memory to GPU memory transfer latency for data movement between Space-4 and Space-3.

Finally, let us consider the case where individual cores of a CPU offload parallel pieces of computations to the GPU. In this scenario, we have an imbalance as there are 16 cores in the CPU but only 15 SMs in the GPU. Thus, we have to discard one core and adjust the CPU hierarchy of Figure 6 to res-establish symmetry before assigning a segment of the GPU to individual cores.

One way to do this is to merge the two Space-3 PPUs and get rid of Space-2 PPUs altogether. Therefore, the cores share the L-3 cache and own an exclusive L-1 cache but cannot access the L-2. Figure 10 shows the final *PCubeS* description of *Titan* after these modifications.

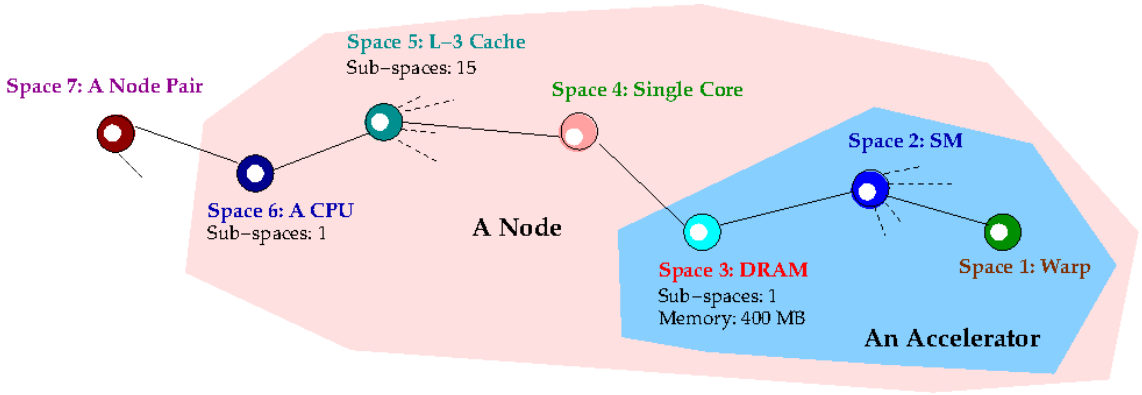

Figure 10: Third PCubeS Description of the Titan Supercomputer

Not shown in Figure 10, but transaction concurrencies and latencies of shared links need to be adjusted in the final description where appropriate.

The Mira Supercomputer: Second Case Study

The *Mira* Supercomputer [2] in Argonne Leadership Computing Facility is a Blue Gene Q system. It has 48 compute racks hosting a total of 49,152 IBM PowerPC [37] A2 nodes. Nodes are connected in a 5D torus interconnection topology and each node has 18 cores running at 1.6 GHz clock speed. This is a symmetric system of homogeneous nodes. Nonetheless, there are important subtleties in the architecture that reveals a rich hierarchy in the *PCubeS* description, as illustrated in Figure 11.

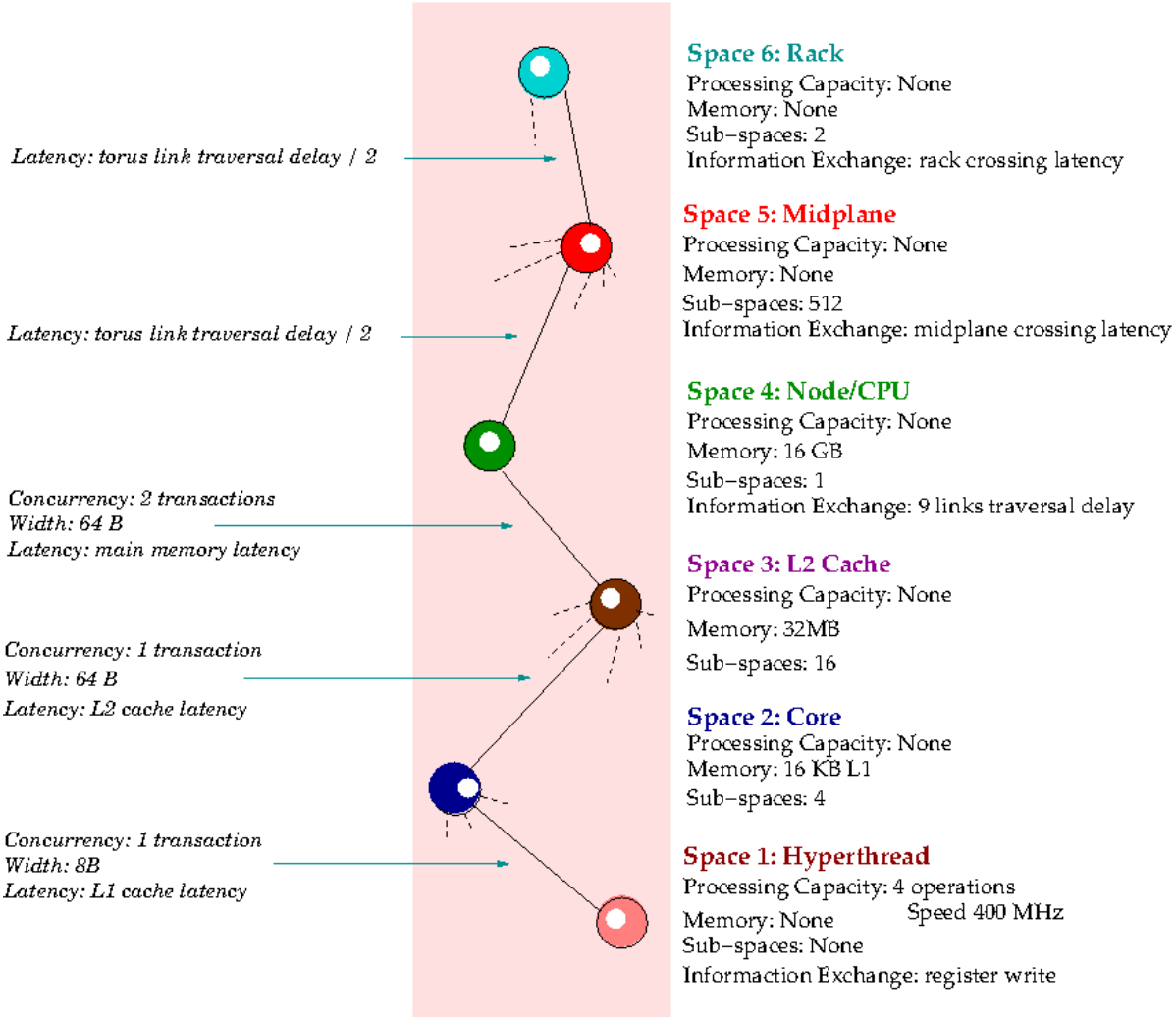

Figure 11: The PCubeS Description of the Mira Supercomputer

Down at the bottom, each core can run up to 4 hyper-threads that have access to a SIMD instruction unit of 4 words wide. So a Space-1 of *Mira* is a hyper-thread with a processing capacity of 4 parallel operations and no memory. The clock speed for an operation is only 400 MHz, instead of 1.6 GHz, as there are 4 Space-1 PPUs.

In the next level, a single core with its 16KB L-1 cache represents a Space-2 PPU. As the data path is 64 bits and hyper-threads' data load/store happens in the L-1 cache, transactions between Space-1 and Space-2 are 8 B wide. Transaction concurrency is 1 as only one hyper-thread issues a load or store at a time.

The single Space-3 PPU in the next level represents the 32 MB L-2 cache shared by all cores. The common cache line size is 64 B. So, that is the width for a transaction between a Space-3 and its Space-2 sub-spaces. A Space-3 has

16 sub-spaces as opposed to 18 because only 16 out of the 18 cores can be used in a program. Above the L-2 cache a CPU or node constitutes a Space-4 with no processing and 16 GB memory capacities.

From Space-4, the 5D torus interconnection network becomes effective. Nonetheless, we get more levels in the hierarchy because of the non-uniform nature of node wiring. A single compute rack of *Mira* holds 1024 nodes. The rack is further divided into two mid-planes containing 512 nodes each. A mid-plane is the smallest full torus configuration where the average distance between a pair of nodes is 9 torus links. Average distance between a communicating pair lying in different mid-planes is thus 10 links.

Therefore, the *PCubeS* description has two additional levels on top of Space-4 for representing a mid-plane and a rack as Space-5 and Space-6 PPUs respectively. There are 48 racks in the system. Hence we have 48 Space-6 PPUs in total. Like *Titan*'s, *Mira*'s hierarchy is also un-rooted as the racks are equivalent.

Notice the Latencies of information exchange among sibling spaces and between a parent and child space starting from Space-4 and above. Space-4 siblings are the 512 nodes connected in a 4×4×4×4×2 torus topology. So an information exchange between a pair of Space-4 PPUs has on average 9 torus links traversal delay. Mid-plane crossing adds one more link in the communication path of two nodes. Therefore, we divide the cost of traversing a single link into half and assign that as the cost of communication between a Space-4 and its parent Space-5 PPU. This augmented with any additional delay for mid-plane crossing – set as the information exchange latency in Space-5 – reflects the true average cost of communication between nodes of opposite mid-planes. The same logic applies for nodes' cross-rack communications in Space-6.

Summary

If we reflect on our discussion in previous sections, we realize that developing *PCubeS* descriptions of a machine is an involved procedure. The choice of spaces and their capabilities depends very much on the expected usage of the underlying features, which requires the developer to have a solid understanding of the expected usage to begin with.

PCubeS' focus on programmability over physical implementation of structural features also makes it difficult to judge a machine as non-*PCubeS* architecture. This is different from Snyder's original idea of type architecture as we see different machines are easily classified as CTA or non-CTA. In *PCubeS*, the efficiency of the description dictates whether the underlying hardware is a *PCubeS* instance or not.

For example, Sun Microsystem's UltraSPARK T1 [18] microprocessors have 8 cores. Each core can handle 4 concurrent threads and has large caches. The system, however, has a single floating point unit to be shared among all cores. With a scheduler that gives round robin access to the floating point unit to all threads, one can give a *PCubeS* description of the system. The sheer inefficiency of the description – not the infeasibility of constructing one – shows that the architecture is not suitable for a *PCubeS* description.

PCubeS as a Basis for Programming Models

Since the primary motivation for having a type architecture is to model architectural features, accurately reflect costs, and serve as the basis for programming models, it is natural to ask how *PCubeS* fares as a basis for efficient parallel programming models and ultimately efficient algorithms on real architectures.

By definition, type architectures standardize the interface between the hardware and the programming language. Therefore any paradigm that operates over a type architecture abstraction is applicable to all hardware the type architecture is capable of efficiently describing.

We have shown the *PCubeS* can model many different, though by no means all, existing parallel architectures. Given that we propose *PCubeS* as the common interface for programming on different parallel platforms, it is important to assess the quality and efficiency of that interface. Does the interface properly expose all the features necessary for efficient parallel programming? Also, does it expose the features in a way that lead to their right usage? To answer these and other similar questions, one needs to understand what a *PCubeS* description conveys about its concerned hardware.

Reflecting back on the *PCubeS* descriptions presented in the previous section, we realize that a hierarchical *PCubeS* description exposes the organization and *qualities* of memories and processors, and the cost of moving data within a machine. Here the term 'quality' deserves particular attention. We believe that the few attributes that characterize *PCubeS* spaces are enough to guide an intelligent programmer to choose the right memory to store his data and the right processor to do his computation for individual pieces of his program. In other words, locality is reflected by the cost to access/manipulate data items.

An example will help clarify this point. Assume we have an application involving many small graph traversals that is to run on *Titan*. One can do the traversals either in the GPU SMs or in the CPU cores. Given that the programmer knows that a typical graph traversal is characterized by irregular memory accesses, it is an easy decision to use the CPU cores for this computation since memory transactions there are only 8 bytes as opposed to 64 bytes as in the SMs. Assuming that graph data structures are small (e.g., 8-16 bytes) and cache aligned, much of the memory bandwidth on an SM will be wasted loading cache lines that will only be partially used with a resulting loss of efficiency due to irregularity. On the other hand, if the application has another part that involves matrix-vector multiplications, the obvious choice for executing that are the GPU SMs due to the enormous parallel computing capability hosted by them in their warp sub-spaces and the regularity of memory accesses in the multiplications.

This logic holds true even if the programmer is using *Stampede* supercomputer instead of *Titan* where the NVIDIA GPU is replaced with a Xeon Phi co-processor and the single AMD Opteron CPU is replaced with 2 Intel Sandy Bridge CPUs. The actual structural details of the involved hardware are not important in the decision making – only their working principle is – and *PCubeS* captures exactly that.

Furthermore, *PCubeS*' depiction of caches as memories has important positive consequences. The programmer is told by it to treat the flow of data from the main memory to a processor as composed of several intermediate data transfer steps going through smaller memories. Sizes of these memories and transaction widths of the data transfers may serve as good hints for automatic cache blocking. For example, assume an iterative refinement problem on a large grid is to be run on an Intel Xeon Phi. The programmer can divide the grid into 60 smaller grids, one for each core. If he, however, pays attention to the *PCubeS* hierarchy of the co-processor; he will see the 512 KB memory (actually an L-2 cache segment) available to each core. Then a smaller grid size that allows the entire data of the concerned region to be held in that memory would be a better choice. As he chooses a fitting small grid size, he gets automatic cache blocking.

Note that conventional interpretation of a system or a portion of it as a shared or distributed memory environment is not appropriate for *PCubeS* descriptions, particularly because of its emphasis on non-uniformity of memory access. In *PCubeS*, memory sharable by the PPUs of a space is the memory of their parent space where those PPUs have uniform access. For example, in Figure 7, SM memory is shared by all warps and global memory is shared by all SMs. The global memory is not a shared memory for the warps. This is in sharp contrast to the CUDA programming model where even a single thread can access an arbitrary global memory address. The *PCubeS* description encourages viewing global memory to warps and vice-versa transfers as been mediated by the SM. So the concurrency limit and collective nature of such transactions are emphasized. This interpretation is closer to the actual working principle of the hardware.

Given that a *PCubeS* based programming paradigm makes the organization and characteristics of memories in the *PCubeS* description of its execution platform visible within a program, we believe, it will be equally efficient regardless of having a distributed or shared memory programming model. In our opinion, knowing the cost of accessing a piece of information from some location – not the actual mechanism of data movement – is important in developing the right algorithm for an execution platform. Hence *PCubeS* presents a simplified, albeit efficient, interface for dealing with memories in parallel programs.

The hierarchical *PCubeS* description is useful beyond optimizing performance within a single node. Although the exact topology of interconnection network is not portrayed in the description; grouping of nodes, cost of communication within and across groups, and degree of concurrency of each communication channel are available. All these are important sources of guidance in decision making regarding inter-node communication frequency, message sizes, and computation-communication overlaps. Sometimes, the hierarchy itself reveals information enough to estimate the benefit of topology sensitive programming. For example, comparing the upper spaces of *Titan*'s *PCubeS* description with that of *Mira*, it is easy to deduce that topology sensitive MPI should be more effective in the latter hardware. This is because the latter has more levels on top of the node level, those levels are densely populated with PPUs, and inter-group PPU-PPU communications are significantly costlier than intra-group PPU-PPU communications.

Nevertheless, *PCubeS* is not devoid of limitations. Specifically, it is silent about IO handling. Efficient IO has increasingly become a significant concern in high performance computing with the proliferation of massive scale databases and data mining applications. We recognize its need, but leave IO as a future concern for *PCubeS*. Similarly, like other models, *PCubeS* does not handle applications that totally fill all available memory on a machine with many levels of memory hierarchy and have random or near random access patterns that vary constantly over the evolution of the computation, for example Automatic Mesh Refinement.

Conclusion

In this report we have presented the *PCubeS* type architecture, its motivation, structure, and limitations. We have argued that *PCubeS* provides important architectural information for parallel machines and applications that other existing type architectures fail to expose. We have also shown that several existing machines can be modeled using *PCubeS*, and what those representations look like.

The next question though is critical.

> *Can PCubeS be efficiently used as the basis for parallel programming models and for parallel programming languages that accurately reflect the costs and capabilities of the underlying machines in such a manner that algorithm developers and programmers can write efficient programs for a variety of machines?*

To address this question we are in the process of developing the *IT* programming language. *IT* is a *PCubeS* language designed around the notion of parallel processing spaces. The *IT* language separates the description of computation occurring in multiple nested spaces from the partitioning of data in each of those spaces and the mapping of those partitioned spaces to the physical spaces of a particular machine. An *IT* program consists of three parts: i) the definition of the computational elements of the application, the loops, conditional statements, and assignments that specify the computations to be performed, ii) the partitioning section that defines how the data structures will be partitioned in each space, and iii) the mapping section that defines how the abstract spaces (computation, data structures, and partitions) are mapped to the physical *PCubeS* spaces of the machine. When moving a code from one machine to another the definition of the computational elements does not change, the partitioning may change, and the mapping will change.

We have executed "hand compiled" *IT* kernels on *PCubeS* modeled NVIDIA cards [38]. We are in the process of developing an IT compiler and run-time environment for large scale machines such as *Stampede*, *Titan*, and *SuperMUC*. The complete language description will be provided in a forth-coming technical report, "*The IT Programming Language and Run-Time Model.*"

Bibliography

[1] Blue gene - wikipedia. http://en.wikipedia.org/wiki/Blue_Gene. Accessed: 2014-08-19.

[2] Mira user guide. http://www.alcf.anl.gov/user-guides/mira-cetus-vesta. Accessed: 2014-08-19.

[3] Stampede user guide. https://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide. Accessed: 2014-08-19.

[4] Titan cray xk7. https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/. Accessed: 2014-08-19.

[5] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation, 1995.

[6] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers, 1993. Proceedings*, pages 116–123, Sep 1993.

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, Meetings Jim Demmel, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY, 2006.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[9] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[10] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[11] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. Zpl: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26:2000, 2000.

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[13] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[14] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, 1998.

[15] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[16] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] Paul N. Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium language reference manual. Technical report, Berkeley, CA, USA, 2001.

[18] Tim Horel and Gary Lauterbach. Ultrasparc-iii: Designing third-generation 64-bit performance. *IEEE Micro*, 19(3):73–85, May 1999.

[19] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March 2003.

[20] Brian W. Kernighan. *The C Programming Languag*e. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[21] HT Kung. Systolic array. 2003.

[22] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[23] Lilia Maliar. Assessing gains from parallel computation on supercomputers. Working Papers. Serie AD 2013-10, Instituto Valenciano de Investigaciones EconÃ³micas, S.A. (Ivie), December 2013.

[24] John McCarthy. History of lisp. *SIGPLAN Not.*, 13(8):217–223, August 1978.

[25] Loren P. Meissner. The fortran programming language: Recent developments and a view of the future. *SIGPLAN Fortran Forum*, 1(1):3–8, July 1982.

[26] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[27] David Notkin, Lawrence Snyder, David Socha, Mary L. Bailey, Bruce Forstall, Kevin Gates, Ray Greenlaw, Willian G. Griswold, Thomas J. Holman, Richard Korry, Gemini Lasswell, Robert Mitchell, and Philip A. Nelson. Experiences with poker. In *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems*, PPEALS '88, pages 10–20, New York, NY, USA, 1988. ACM.

[28] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[29] Antonio J. Peña, Ralf G. Correa Carvalho, James Dinan, Pavan Balaji, Rajeev Thakur, and William Gropp. Analysis of topology-dependent mpi performance on gemini networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 61–66, New York, NY, USA, 2013. ACM.

[30] Gregory F. Pfister and V. Alan Norton. Interconnection networks for high-performance parallel computers. chapter "Hot Spot" Contention and Combining in Multistage Interconnection Networks, pages 276–281. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[31] Jacob T. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, October 1980.

[32] H.J. Siegel. A model of simd machines and a comparison of various interconnection networks. *Computers, IEEE Transactions on*, C-28(12):907–917, Dec 1979.

[33] Lawrence Snyder. Annual review of computer science vol. 1, 1986. chapter Type Architectures, Shared Memory, and the Corollary of Modest Potential, pages 289–317. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[34] Tim Stitt. An introduction to the partitioned global address space (pgas) programming model.

[35] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[36] David W. Walker, David W. Walker, Jack J. Dongarra, and Jack J. Dongarra. Mpi: A standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[37] Shlomo Weiss and James E. Smith. *IBM Power and PowerPC*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[38] Muhammad N. Yanhaona and Andrew S. Grimshaw. It – a simple parallel language for hierarchical parallel architectures. Technical report, Technical Report, UVA Computer Science, 2014.