Locality: The 3rd Wall and The Need for Innovation in Parallel Architectures

Peter M. Kogge[1] and Brian A. Page[1] kogge,bpage1@nd.edu

University of Notre Dame, Notre Dame, IN 46556, USA

Abstract. In the past we have seen two major "walls" (memory and power) whose vanquishing required significant advances in architecture. This paper discusses evidence of a third wall dealing with data locality, which is prevalent in data intensive applications where computation is dominated by memory access and movement – not flops, Such apps exhibit large sets of often persistent data, with little reuse during computation, no predictable regularity, significantly different scaling characteristics, and where streaming is becoming important. Further, as we move to highly parallel algorithms (as in running in the cloud), these issues will get even worse. Solving such problems will take a new set of innovations in architecture. In addition to data on the new wall, this paper will look at one possible technique: the concept of migrating threads, and give evidence of its potential value based on several benchmarks that have scaling difficulties on conventional architectures.

Keywords: architecture · parallelism · multi-threading.

1 Introduction

The Cambrian explosion started about 540 million years ago, and represented the jump from simple multi-cell colonies to complex multi-cell organisms, both plant and animal, with rich internal structures and optimized for a wild variety of environments. In their 2018 Turing Lecture [16], Hennessey and Patterson suggested that we are reaching a similar threshold in computer architecture. We have gone from simple single "cell" cores, through clustered "colonies" of such cells, to the point where we are beginning to see specialized "organs" implemented by accelerators for different functions. While the efficiency and scalability of such systems are often quite good for problems with dense data that is accessed in a regular pattern, this is not so true for "newly evolving" problems that are sparse and irregular in their access patterns.

The thesis of this paper is that the problem is **locality**, which as typically defined can come in several forms. In **temporal locality**, if some memory location is accessed once, there is a high probability that the same location will be accessed again within a short period of time. In **spatial locality**, if one location is accessed, then there is a high probability that nearby locations will be accessed within a short period of time. In parallel systems we also have **physical locality** where it matters whether or not data is in a locally accessible memory.

Table 1. Eras in Parallel Architectures

Era	Slime Mold		Fungi	Moss		
	Prehistory	Archaic	Yesteryear	Last Month	Yesterday	Today
App	3D Mesh-like	Ax=b, A dense $n \times n$ matrix			Ax=b sparse, BFS	ML, Analytics
Time Frame	1980s	1990s	2004	2008	2013	
Computation	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n)$	$O(n^?)$
Memory	$O(n^3)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^?)$
I/O	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$	$O(n^{2/3})$	$O(n^?)$
Issue	Basic Scaling	Memory Bandwidth	Socket Power	Power Efficiency	Low App Intensity	Locality
New Advances	2D topology	Caches	Multi-core	Heterogeneous	GPUs, Hybrid	Massive core
	Moore's Law Clock		Flat Clock	Hybrid	Many Core	Smart NICs
			SOC	Fat Tree	Hi-degree N/W	
				Low precision	Proc. In/Near Mem	

This paper is organized as follows. Section 2 walks briefly through the historical changes. Sections 3 and 4 provide evidence of this new wall. Section 5 discusses one possible approach to solve these issues by allowing threads to migrate. Section 6 provides examples of its effectiveness. Section 7 concludes.

2 Parallel Architectural Archaeology

To understand more fully the forces driving the need for changes in parallel architectures, it is instructive to review briefly the historical record. In analogy with biological evolution, the Proterozoic Eon saw the rise of the first cells with organelles, which in computing terms corresponds to simple single core computers with basic function units. The Phanerozoic Eon saw the growth of multi-cell organisms, or, in computing terms, systems with multiple cores. While parallel function units date back to the Illiac IV (1972), and small-scale shared memory machines date back almost as far, the first highly parallel machines really appeared only in the mid 1980s (the Caltech Cosmic Cube). From then to now we have seen three major eras in parallel systems:

- **Slime Mold era**: In biology, this early era was dominated by loose colonies of simple single-cell organisms. The parallel architecture equivalent was the early systems that had simple single core nodes with simple interconnects.
- **Fungi era**: In biology, this era represented a change to colonies of multi-cell organisms, but where the cells are still mostly undifferentiated. The parallel architecture equivalent was the appearance of multi-core processor chips.
- **Moss era**: In biology, this era began the evolution to organisms with multiple types of cells. The parallel architecture equivalent was the introduction of hybrid nodes with different types of cores and smart network interfaces.

In evolutionary terms, there have been major "extinction events" that caused die-offs of lifeforms, and the emergence of newer ones that were better suited. The same thing has happened with parallel architectures, driven by either technology ("walls") or changes in application needs. Table 1 summarizes how such events have bifurcated the above three eras further into six periods.

Prehistory - the start of the Slime Mold era: The first wave of parallel computers were designed to solve 3D problems that were decomposable into relatively independent sub-cubes that could be solved largely separately, such as 3D mesh equations and N-body problems, where "surfaces" of sub-cubes were exchanged with nearest neighbors. Architecturally, nodes in such systems were simple single cores. The major issue was simply getting enough parallel nodes. The major technology advance was using Moore's Law to move to single-chip cores and scale the clock, with simple 2D or 3D hypercube interconnect topology.

Archaic: The Memory Wall. The second period emerged when processors had accelerated clock speeds to the level where now memory bandwidth was the major impediment. This triggered a shift from mesh-like simulation to the solution of large linear equations of the form $Ax = b$ where A is a large dense matrix. Packages such as LAPACK became quite efficient and easily parallelizable, and led to **HPL** - the benchmark used for the semi-annual TOP500 rankings of supercomputers. An advantage of these algorithms was that these large matrices could be partitioned into smaller sub-matrices, such that a sub-matrix of $O(m^2)$ values could be read into a core once, and $O(m^3)$ operations could be performed on them, for the equivalent of m flops for each value read from memory. This results in high locality, and with a sufficient-sized cache, a core could use nearly all its capability. Moore's Law provided the transistors to do this.

Yesteryear: The Power Wall. For a long time "Dennard scaling" allowed us to reduce the size of transistors, increase the clock, and decrease the supply voltage, all while keeping chip power density relatively constant. This meant faster cores with bigger caches and enhanced ILP. Improving fabrication technology allowed larger die to be produced, resulting in power that was linear in die size, but independent of clock or computational features. Around 2004, however, our ability to scale voltage down slowed tremendously, and chip power dissipation sykrocketed. This forced a major change in architecture. Maximum clock speeds flattened to around 2-3GHz, and processor chips went from holding single complex cores to multiple simpler cores that were more power efficient. Caches continued to grow in size, with additional levels of caching introduced to serve the multiple on-chip cores. Systems-on-a-chip that integrated multi-core processing and networking became practical, such as IBM Blue Gene [11].

Last Month: Energy Efficiency. Even with multi-core processing chips with simple cores, it became obvious that additional energy efficiencies would have to be developed to allow cracking the petaflops (10^{15} flops/sec.) barrier at acceptable power levels. The first such system to do so in 2008, Roadrunner [1], had several unique architectural features, and was a precursor for what became standard in the following decade. First, it employed a **heterogeneous** architecture with two distinct types of processing chips, a conventional chip and one adapted for specialized computation. Further, the accelerator chip was itself a **hybrid** multi-core, where there were two types of cores that shared the same memory space. Finally, the topology of the network changed from a mesh or torus to a switched fat-tree that provided better non-nearest neighbor communication.

At the same time it became obvious that to get the next 1000x ("exascale") at acceptable power and in a reasonable period of time would need even more architectural advances. A major study [21] performed an in-depth projection that determined energy efficiency was the major problem, with memory and interconnect, not computation, as the major culprits. The proposed architecture featured not multi-core but **many-core** chips with hundreds of cores, 3D **stacked memory** with substrate-level chip-to-chip connectivity to the processor cores, on-chip integrated network interfaces, and high-radix interconnect topologies that do not use external switches. Even with all this, the projection for a 2015 exascale machine was 3X over the power goal, and to run at high efficiency for dense problems could not provide anywhere near the relative memory or network bandwidth found in then current machines. In addition, with millions of cores, programming would have to deal with perhaps up to one billion threads.

Finally, this era also saw the rise of "Big Data" with analysis on large data sets that were too unstructured for conventional databases. New parallel execution models like MapReduce staged data partitioned across many nodes through various processing and merge steps. Architecturally, the workload presented by such apps was dramatically different than their original design point (cf. [18]).

Yesterday: Sparsity. By the mid 2010s it became obvious that real algorithms were not performing at anywhere near the floating point rates that HPL achieved on the then top supercomputers. The percentage of non-floating point operations had grown dramatically. Sparsity in data sets required significantly more memory accesses that were not cacheable, and added to memory bandwidth needs. Inter-node communication could no longer be fully overlapped with computation, and thus became significant to compute times. Short point-to-point messages with remote atomic operations were important for collectives that controlled the overall flow of parallel computations. To shed light on these issues, two new benchmarks were introduced. The first, Graph500, (2010) performs a breadth-first search (BFS) through very large random graphs, and emphasizes memory performance, short messages to random targets, and remote atomic operations. Performance is in **Traversed Edges Per Second** (**TEPS**). The second, High Performance Conjugate Gradient (**HPCG**) [17], is the solution of $Ax = b$, but where A is very sparse, and integrated a local sparse matrix-vector product with an iterative algorithm where the full matrix was partitioned across a large number of nodes. Communication between nodes is regular and a secondary performance gate. As with HPL, performance is in flops. Unlike HPL, where floating point efficiencies 80% and above are common, HPCG delivers at best low single digits. The issue is the extra memory accesses to handle the sparsity in the local matrix [23].

This era saw an explosive growth in heterogeneous architectures with the coupling of conventional many-core processors to **GPGPUs** that themselves are hybrid processors capable of running hundreds' to thousands' of threads. At least one system (TaihuLight [10]) had only hybrid chips, each with literally hundreds of cores. While highly efficient for traditional dense applications, these features offered little for the newer sparse applications. However, given their

origins in graphics applications needing less than 64-bit precision, they included capabilities of performing multiple reduced-precision functions in the same time as single 64-bit floating point operations.

Coupled with this were significant changes to the memory hierarchy. Three levels of caching became commonplace. "Scratchpad" memory emerged that was not cache but directly accessible, especially in many-core hybrid chips. 3D memory stacks offered more memory channels and much higher bandwidths. Persistent memory that does not lose contents on power down is blurring the line between main memory and file systems, and led to the term **storage class memory**. Another trend has been to push intelligence out into the network. "Smart NICs" reduced the overhead of message management by performing such functions in hardware at either endpoints or in network switches.

Today: The Locality Wall. In the last few years we have seen a sea-change in the applications driving high end parallel systems, with machine learning (**ML**) leading the way. The emergence of the Internet of Things (**IoT**) has pushed computation out to huge numbers of endpoints. The need to extract and recognize complex connections in such massive, irregular, real-time, and growingly sparse, data sets has become critical to both science and business.

The bulk of today's AI applications take two forms: training and inference. The former takes large data sets and tries to deduce a model. The latter apply such models to real data to make predictions about the data. Architecturally, such apps have triggered a growth in non-traditional parallel accelerators. Nvidia, for example, markets a "DGX^{tm} SuperPOD" that networks over 1500 GPUs. Google is even more application-specific with systems of thousands of Tensor Processing Unit (**TPUs**) chips [19] that have up to 32,000 multiplier-adders on a chip to accelerate matrix-vector products. Wafer-scale integration as in the Cerebus chip now allows over 400,000 AI-tuned cores to be placed on a huge chip with over 1 trillion transistors [15]. Such systems have several common features. First is a use of short floating point to greatly reduce the hardware needed to perform computations. Second is the use of 3D stacked memory to provide sufficient bandwidth to keep these huge numbers of function units fed.

Given the importance of lower precision arithmetic in such ML applications, a new benchmark **HPL-AI** has been developed to continue attacking the $Ax=b$ problem on conventional computers, but using lower than 64-bit precision for the bulk of the computation. Limited results to date indicate a speedup of between 2.5 and 4.5X on the same hardware over native 64-bit performance.

Also there are indications that the old idea of "Processing Near Memory" (PNM) or "Processing/Computing In Memory" (PIM/CIM) may finally demonstrate real advantages, both for increased peak intensity for specialized apps and for lowered energy. IBM's TrueNorth chip [7] mirrors parallel functions found in the human brain. Other chips, c.f. [35], implement low precision linear algebra within a novel memory structure at extraordinarily low levels of energy.

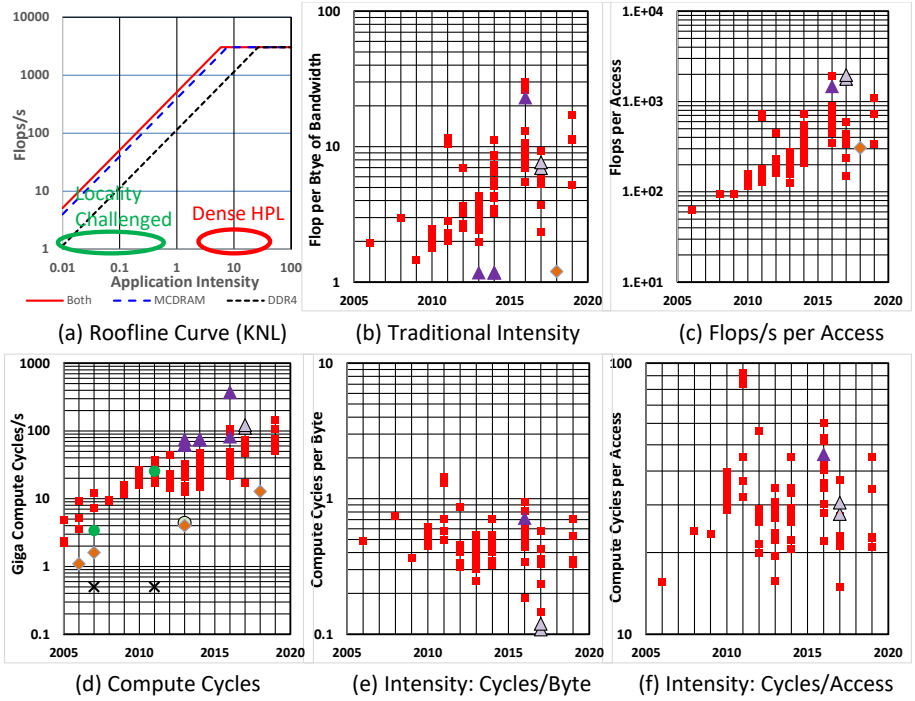

Fig. 1. Today's Architectures in terms of Ridge Point Intensity. The different color points represent different classes of chip architectures.

3 Evidence of a New Wall - Low Intensity Apps

The **roofline** model [36] is a useful visualization of multiple performance bounds, and how close a particular code is to reaching those bounds for a particular architecture. In such charts, the y-axis is a measure of predicted performance in terms of some basic operation count per second (such as flops/s), and the x-axis a measure of "operational intensity," the ratio of the number of such operations performed in the algorithm divided by the minimum traffic from some level of memory needed to support them. The numerator is a property of the program being run, and the denominator a property of the memory hierarchy. A bounding curve for such a chart is a line that represents the maximum performance possible out of a particular system when running a code with a particular intensity.

Fig. 1(a) is an example of a bounding curve for a system using an Intel Knight's Landing many-core chip. In terms of bandwidth between memory and the processor chip there are three possibilities: all data transfers are from the main DDR4 memory, all is from the 3D stacked MCDRAM memory, or data can come from either one at the same time. Each bounding curve consists of two intersecting lines. The upper flat line is the peak performance possible from the system if the computational units in the cores can run at 100%. The sloping

line is a bandwidth constraining bound with a slope equalling the bandwidth of the memory in bytes/sec. The "ridge point" where the two lines intersect is when the peak bandwidth times the application's peak intensity equals the peak system performance. Alternatively, this ridge point is when a code's operational intensity equals the ratio of the processor's peak computational performance divided by the processor's peak memory bandwidth. This ridge point is thus an important characteristic of a processor. Fig. 1(b) charts how such a metric has changed with time when flops is the metric. Modern designs have values of 10 or more, meaning that to utilize all their computational capability, a code must continually find 10 or more flops to perform in the time required to transfer a single byte from memory. This is the direct result of optimizing for dense HPL.

As some real examples, the **SKA** project is a massive radio-telescope project with real-time computation at its core. An estimate of the Science Data Processing chain is that it needs 250 pflops/s peak with a memory bandwidth of 200 pB/s [32,5]. This is an intensity of only 1.25 flops/byte. Further, the HPCG sparse benchmark has an intensity of about 0.2 - even further below modern systems. Also, the Sparse Matrix Dense Vector (SpMV) kernel that is central to HPCG is even lower, at about 0.1 flops/byte. As another example, an ML kernel using Stochastic Gradient Descent to compute a Support Vector Machine model employs a loop involving an inner product and a vector-vector like operation where one of the vectors (the model) has some temporal locality but the other does not. The intensity varies between 0.01 and 0.4 depending on sparsity.

A variant of this is to use access rate as the denominator. Today's architectures are designed to deliver 64 or more bytes per access, regardless of how much is used. This is fine for dense HPL where there is significant spatial locality, but for applications with little such, much of the accessed data is never used, and represents wasted bandwidth. Fig. 1(c) charts this variation of intensity for the same processors as Fig. 1(b). For a modern processor to run at maximum efficiency, it must execute 1,000 or more flops for each memory access it makes.

Looking further, many of the newer locality-sensitive apps are not flop-intensive, so the traditional flop intensity metric does not make sense. An alternative to flops/s might be simply "compute cycles" - clock rate times number of cores. If "operations" are measured as some number of instructions, and each core may have some CPI up to a maximum of its issue width, then such a metric is a decent approximation. Fig. 1(d) diagrams aggregate compute cycles per processor over time, and Figs. 1(e) and (f) diagram equivalent intensity values for both bandwidth and access rates. As can be seen, if apps have high locality then it is easy to stay compute bound. However, with low locality, an app would need 10s to 100s of instructions per operation to not be memory-bound.

An example is the BFS from the Graph500. Dividing the peak TEPS rating for a typical processor by its memory bandwidth yields an intensity of about 0.03 TEPS/byte. The **Firehose**[2] kernel takes streams of internet packets of data containing IP addresses and a payload, correlates them via a huge hash table, and looks for "events" based on the aggregate payloads. The metric is "Datums per second," with an intensity of about 0.01 Datums/byte.

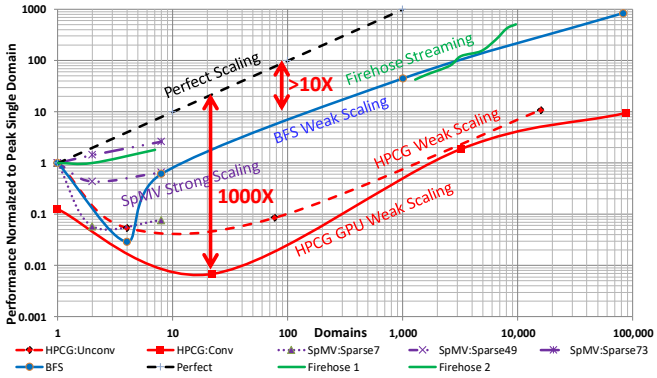

Fig. 2. Some Traditional Scaling Results. BFS comes from www.graph500.org; HPCG from www.hpcg-benchmark.org; SpMV from [4]; Firehose from firehose.sandia.gov.

In terms of PIM-like architectures, a recent study [12] looked at a variety of challenges associated with when to use such concepts and the integration with conventional systems. Such issues included many of the same ones mentioned, especially how to integrate into a larger memory system, handle cache coherency, and support a programming model that allows asynchronous executions.

The key take-away is that in many emerging cases the app intensity is at least an order of magnitude less than the peak capability of a modern core, meaning that performance is dominated by memory not processor architecture.

4 Further Evidence of a New Wall - Poor Scaling

The prior section gave evidence that memory has once again become a gating issue, but there is additional evidence that it is more than just a return to the Memory Wall. Fig. 2 diagrams speedup for several sparse or irregular benchmarks over a wide range of scaling and many different multi-node parallel architectures. Each line represents a different benchmark for which years of reports are available. The x-axis is the number of nodes that have been reported for the same benchmark. The y-axis is the best relative speedup that some system exhibited over the performance reported for the best case of a single node. For each benchmark, the performance at (1,1) is the best reported for a single node (this is often from a multi-threaded algorithm running on a multi-core chip where memory is shared). The curves all have similar characteristics. Going from 1 to multiple nodes causes an immediate performance loss of sometimes 10X or more. While in most cases decent linearity in the scaling is eventually obtained, 10 to 100 nodes are needed to regain the best performance of a single node, and even then there is still significant performance loss over perfect scaling.

To provide even more insight, Fig. 3(a) (from [30]) diagrams SpMV strong scaling results from three different architectures for a variety of matrices of varying sizes and sparsity. The architectures include a conventional cluster, a

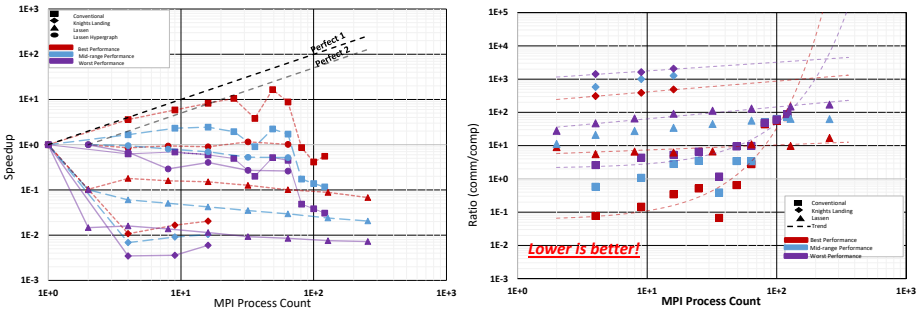

Fig. 3. SpMV Strong Scaling (from [30]).

cluster of Knight's Landing nodes, and Lassen (a smaller version of Sierra with dual socket IBM POWER9 chips and multiple GPUs). The three curves for each included the best results from any matrix in the suite, the worst results, and the median results. For all but the densest matrices, the conventional cluster was the only one to actually achieve even slightly positive speedup. Looking closer, Fig. 3(b) shows why. For most cases, the ratio of the communication time to computation time exceeds 1 and increases as the parallelism increases, meaning that as the computational power of the nodes increases the computation time drops but communication time does not. Making this even worse, as the level of parallelism increases, the size of the local problem also decreases. Only for the conventional case is the ratio ever less than 1, namely because this architecture has relatively less computation capability than KNL or Lassen. Communications delays increase with parallelism and dominates overall execution time. This is true even for the second Lassen case where an expensive hypergraph partitioner was used to optimize data placement to reduce communication by up to 90%.

Another example can be seen in the Firehose data in Fig. 2, especially the leftmost part of the curve where the parallelism involves multiple cores on the same node. Going from 1 to 7 cores increased performance by less than a factor of 2. The issue here is a combination of low intensity and the need to employ multiple expensive atomic updates to non-local hash table entries.

Yet another example is an HogWild!-style algorithm for a parallel ML trainer for sparse SVM problems. Before HogWild! [25], attempts to parallelize via multi-threading suffered from memory contention and excessive cache coherency traffic. HogWild! attacked this by noticing that for sparse problems different trainers are unlikely to be updating the same features at the same time, and as long as individual updates are done atomically, coherency issues are minimized. While this was relatively successful for low levels of parallelism and moderate sparsity, after that the speedup plummeted much as for SpMV. A variant, HogWild++ [38], had each socket in a node operating separately, and used a token-passing mechanism to perform inter-socket updates. This worked relatively well for up to 40 cores except for very sparse data sets. In one case, the *news20* data set with 1.5 million features per sample, but only on average 460 of them non-zero, only

achieved a speedup of about 9.5 in a 40 core system. Once again a combination of sparsity and inter-socket communication caused the scalability issues.

Finally, even experimental CIM/PIM chips such as [35], have scaling issues, as the energy needed to combine two or more memory blocks far exceeds the energy to perform a parallel operation within them, even if on the same chip.

The net effects is that such applications cause memory access issues "in the small" by low intensity within a single (perhaps multi-threaded) node, AND access issues "in the large" where accessing data on remote nodes becomes dominant. Together, these issues form what we claim is the new Locality Wall.

5 An Alternative Architecture - Migrating Threads

As an alternative, instead of keeping the site of a computation stationary at some core and moving copies of data as needed to it (and thus causing locality problems), we might consider what happens if instead we allow the state of a thread to *migrate* as needed to different computational sites. This has been implemented for years in software, from actors and SmallTalk [14], active messages [9], to autonomous objects [3] and messengers [13]. All have been conceptually attractive, but suffered from significant software overhead and scalability.

Limited hardware implementations have been demonstrated, such as the J-machine [26], but only for primarily remote procedure calls. A more complete model would handle all remote memory references in hardware, without any explicit software involvement. When a thread executing on some core tries to access a location not found in a memory to which the core is associated, the hardware suspends the thread from execution, its state is packaged, sent to the appropriate node, unpackaged, and restarted on a more appropriate core - all without any explicit software or additional memory references. Thus all memory references are local, and if any caching is memory and not core correlated, without any coherency traffic. Further, to allow such migrations, all memory must be in a common logical address space. Fig. 4 diagrams such an architecture.

There are at least two reasons why this concept might be an aid to overcoming locality issues. First, while migrating an execution doesn't change the intensity of an application, it does eliminate all the cache coherency traffic required to track the *copies* of data that are shared among traditional threads in a classical shared memory node. Second. and of more impact, it eliminates the entire software stack needed to communicate between nodes in a traditional distributed memory cluster. In most of the above apps with scaling problems, the inter-node communication is usually to request that some small amount of data be shipped to another node and some rather simple set of operations performed there. An example is a remote atomic operation on some memory location, such as a partial sum accumulation. Today, with MPI, SHMEM, or the like, software must copy the data to a buffer, assemble a message around it, read out the message into the sending NIC, write the message back to a buffer on the receiving side, and trigger a thread to interpret and execute the required action. An acknowledgement requires a similar path. When we add in the memory references

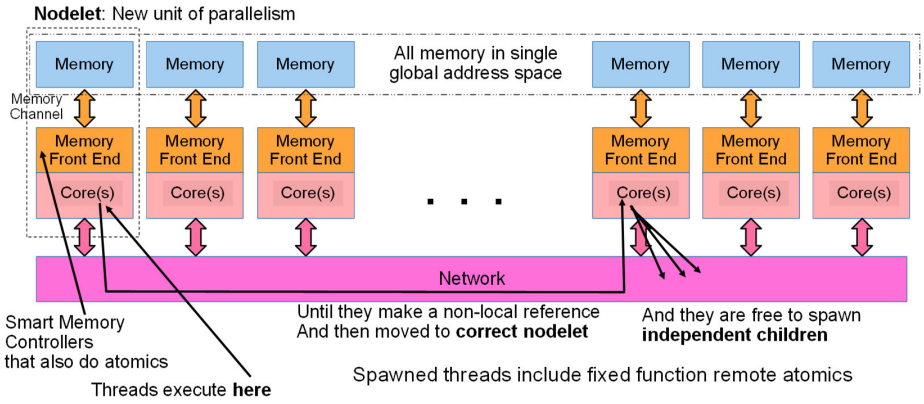

Fig. 4. A Migrating Thread Architecture (from [20]).

needed to save and restore registers at both procedure call/returns and interrupt handling, it should be clear that we have added a huge number of memory references to support what should be a simple operation. With today's architectures we have also idled the core on the sending side and interfered with the core on the receiving side. This is a performance and energy hit.

There are several obvious enhancements. First is to make the cores multithreaded so that arriving and departing threads can be scheduled for execution at the hardware level, and with sufficient threads at each core, the cores can be kept fully utilized. Second is enhancing the memory controllers at each channel to perform **atomic operations** directly on memory, with an absolute guarantee of atomicity in terms of other threads. Next, adding lightweight mechanisms to allow a thread to spawn additional threads provides cheap mechanisms for ad hoc parallelism. Finally, allowing this mechanism to spawn not just full threads but even lighter weight threads with limited functionality (such as perform a remote atomic add) can reduce even further the cost of remote operations.

6 A Real Example

One such machine [8] was designed by Lucata Inc. and is housed at Georgia Tech's CRNCH center[1]. It consists of 64 memory channels, each with a multithreaded core capable of holding 64 active thread states, and interconnected with a RapidIO network designed to transport the register set of a thread between cores without software. Each memory channel holds 8GB and returns only 8 bytes per access, minimizing unused spatial locality. All memory is in a single common logical address space. The memory channel/core combination is packaged 8 to a board, with 8 boards in a hypercube topology. The current implementation is in an FPGA with programs written in Cilk. Each board also

[1] https://crnch.gatech.edu/rogues-emu

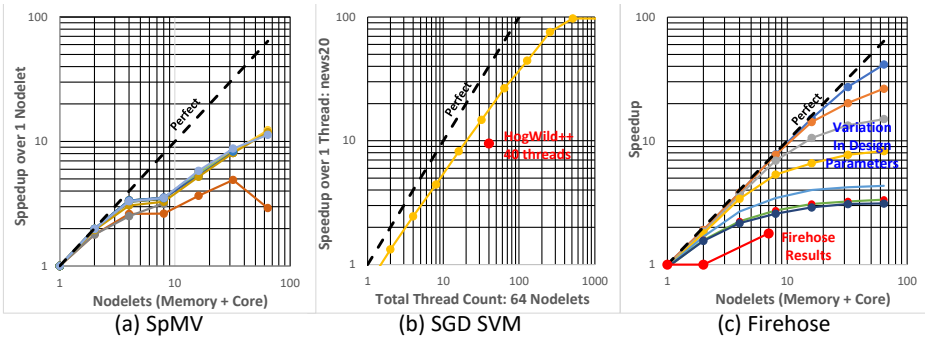

Fig. 5. Speedup Results on the CRNCH System.

houses a conventional dual-core processor that runs Linux, hosts a local SSD, and can interface to a PCIe adapter for communication with the outside world. Applications are launched from a Linux process into the system as a single master thread, which then spawns threads on each nodelet to perform localized initialization and then spawn additional worker threads to perform the parallel computations. At completion, the parent Linux process is signaled.

A variety of demonstration benchmarks have been developed and run on this machine. Fig. 5 diagrams summary speedup for three of them. The SpMV results [28], Fig. 5(a), should be compared to Fig. 3(a). Both use the same set of matrices, and while none of the conventional architectures had good scaling for all cases, the migrating thread system was positive not only for the best cases, but even the worse cases. The key feature used was the ability to launch very light weight threads to perform remote atomic memory operations.

Fig. 5(b) diagrams some results for the migrating thread version ([27] Chap. 4) of the HogWild! style algorithms. The data set is the sparse news20, which achieved only a speedup of 9.5 on a 4-socket 40 core system. The code was similar in that one nodelet corresponded to a single multi-threaded training process. The inter-process algorithm, however, was optimized to use the features of the migrating thread architecture. At 40 threads, the migrating threads version exceeded the classical reported speedup by a factor of 2, and continues to increase.

Finally, Fig. 5(c) diagrams an implementation of the Firehose streaming benchmark for which today's architectures have scaling issues. There are three versions of the benchmark, in increasing complexity, with a metric of datums processed per second. For the simplest version the migrating thread architecture [29] not only had better speedup, but actually beat the conventional architectures in datums/s by a large factor, even though the classical cores had a 20X higher clock. For the intermediate version, [31], the conventional 7 core version achieved a measly 1.8X speedup while an 8 nodelet migrating version achieved near perfect 6.6X. Other examples of benchmarking for this machine include radix sort [24], machine learning using the Random Forest algorithm [34], sparse

linear algorithm kernels [22], pointer chasing [37], approaches to handling sparsity [33], and new compilation techniques [6].

7 Conclusions

In evolutionary terms, parallel architectures have grown from colonies of simple cells to simple plants with a few cell types performing different functions, but with all "life activities" occurring within some limited region. Explicit messaging across an interface is needed to affect computation being performed elsewhere. This paper has presented evidence that today's architectures are highly inefficient at supporting many emerging apps. The paper then suggests that the introduction of primitive "animals" in the form of migrating threads that can move freely around a system can provide better efficiency.

Also, given the current implementation of the prototype migrating thread system in a FPGA, there are some interesting caveats to some of the currently demonstrated benchmarks. First, the core is a single issue design running at 175MHz, vs multi-issue 2-3GHz classical cores. Then each classical socket has 3-4 memory channels for a combined memory bandwidth of 30-50GB/s. This is about 5-6 GB/s per core. The migrating thread implementation has only 1.2 GB/s per core. Thus on a per core basis the FPGA cores are significantly less capable than a classical core. This makes many of the migrating thread results especially striking, especially the throughput numbers for Firehose. It will be interesting to see what happens when faster and larger migrating thread systems become available.

Looking further, the marriage of migrating threads with PIM/CIM and PNM processors placed on the bottom of 3D stacks of memory may usher in an era where computing is done in a sea of "plant-like" stacks, with "animal" threads moving naturally and freely throughout them to manage computation.

Acknowledgements

This work was supported in part by NSF grant CCF-1642280, and in part by the University of Notre Dame. We would also like to acknowledge the CRNCH Center at Georgia Tech for allowing us to use the Emu system there.

References

1. Barker, K., Davis, K., Hoisie, A., et al: Entering the petaflop era: The architecture and performance of roadrunner. In: 2008 SC - Int. Conf. for High Perf. Computing, Networking, Storage and Analysis, SC 2008. p. 1 (11 2008)
2. Berry, J., Porter, A.: Stateful streaming in distributed memory supercomputers. In: Chesapeake Large Scale Data Analytics Conf. (2016)
3. Bic, L.: Distributed computing using autonomous objects. In: Proc. 5th IEEE Workshop on Future Trends of Distributed Computing Systems. pp. 160–168 (1995)

4. Bylina, B., Bylina, J., Stpiczynski, P., Szalkowski, D.: Performance analysis of multicore and multinodal implementation of SpMV operation. In: 2014 Federated Conf. on Computer Science and Information Systems. pp. 569–576 (Sept 2014)
5. Chan, T., Brown, A., Ensor, A.: SDP Memo 54: Compute Node Pipeline Efficiency Assessment Framework. Tech. rep., SKA Square Kilometre Array (Aug 2018)
6. Chatarasi, P., Sarkar, V.: A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System. In: Proc.Workshop on Memory Centric High Performance Computing. p. 37–44. MCHPC'18, Assoc. for Computing Machinery, New York, NY, USA (2018)
7. Cheng, H., Wen, W., Wu, C., Li, S., Li, H.H., Chen, Y.: Understanding the design of IBM neurosynaptic system and its tradeoffs: A user perspective. In: Design, Automation Test in Europe Conf. Exhibition (DATE), 2017. pp. 139–144 (2017)
8. Dysart, T., Kogge, P.M., Deneroff, M., et al: Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In: Proc. of 6th Workshop on Irregular Applications: Architectures and Algorithms. pp. 2–9. IA3 '16, IEEE Press, Piscataway, NJ, USA (Nov 2016)
9. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: A mechanism for integrated communication and computation. In: Proc. 19th Int. Symp. on Computer Architecture. pp. 256–266. ISCA '92, ACM, New York, NY, USA (1992), http://doi.acm.org/10.1145/139669.140382
10. Fu, H., Liao, J., Yang, J., et al: The Sunway TaihuLight supercomputer: system and applications. Science China. Information Sciences **59**, 072001:1–16 (07 2016)
11. Gara, A., Blumrich, M.A., Chen, D., et al: Overview of the Blue Gene/L system architecture. IBM J. of R&D **49**(2.3), 195–212 (2005)
12. Ghose, S., Boroumand, A., Kim, J.S., Gómez-Luna, J., Mutlu, O.: Processing-in-memory: A workload-driven perspective. IBM J. of R&D **63**(6), 3:1–3:19 (2019)
13. Gmelin, M., Kreuzinger, J., Pfeffer, M., Ungerer, T.: Agent-Based Distributed Computing with JMessengers. In: Proc. Int. Workshop on Innovative Internet Computing Systems. p. 134–145. IICS '01, Springer-Verlag, Berlin, Heidelberg (2001)
14. Goldberg, A.: SMALLTALK-80: The Interactive Programming Environment. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1984)
15. Groeneveld, P.: Wafer scale interconnect and pathfinding for machine learning hardware. In: Proceedings of the Workshop on System-Level Interconnect: Problems and Pathfinding Workshop. SLIP '20, Assoc. for Computing Machinery, New York, NY, USA (2020)
16. Hennessy, J.L., Patterson, D.A.: A New Golden Age for Computer Architecture. Comm. ACM **62**(2), 48–60 (Jan 2019)
17. Heroux, M.A., Dongarra, J.: Toward a new metric for ranking high performance computing systems. Sandia Report SAND2013 4744 (June 2013)
18. Jia, Z., Zhan, J., Wang, L., et al: Understanding Big Data Analytics Workloads on Modern Processors. IEEE Trans. on Parallel and Distributed Systems **28**(6), 1797–1810 (2017)
19. Jouppi, N.P., Yoon, D.H., et al: A Domain-Specific Supercomputer for Training Deep Neural Networks. Comm. ACM **63**(7), 67–78 (Jun 2020)
20. Kogge, P.M.: Unifying Threading Paradigms for Highly Scalable PGAS Systems with Mobile Threads. In: Int. Conf. on High Perf. Computing and Simulation (HPCS) (July 2019)
21. Kogge, P.M., Bergman, K., Borkar, S., et al: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Tech. Rep. CSE 2008-13, Univ. of Notre Dame (Sept 2008), http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf

22. Krawezik, G.P., Kuntz, S.K., Kogge, P.M.: Implementing Sparse Linear Algebra Kernels on the Lucata Pathfinder-A Computer. In: IEEE High Perf. Extreme Computing Conf. (HPEC) (Sept 2020)
23. Marjanovic, V., Gracia, J., Glass, C.W.: High Perf. Computing Systems. Performance Modeling, Benchmarking, and Simulation, chap. Performance modeling of the HPCG benchmark, pp. 172–192. Springer Int. Publishing (Nov 2014)
24. Minutoli, M., Kuntz, S., Tumeo, A., Kogge, P.M.: Implementing Radix Sort on Emu 1. In: 3rd Workshop on Near-Data Processing in conjunction with 48th IEEE/ACM Int. Symp. on Microarchitecture (MICRO-48) (Dec 2015)
25. Niu, F., Recht, B., Re, C., Wright, S.J.: HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In: Proc. 24th Int. Conf. on Neural Info/ Proc. Systems. pp. 693–701. NIPS'11, Curran Associates Inc., USA (2011)
26. Noakes, M.D., Wallach, D.A., Dally, W.J.: The J-Machine Multicomputer: An Architectural Evaluation. In: Proc. 20th Int. Symp. on Computer Architecture. p. 224–235. ISCA '93, ACM, New York, NY, USA (1993)
27. Page, B.A.: Scalability of Irregular Problems. Ph.D. thesis, Univ. of Notre Dame, USA (Oct 2020)
28. Page, B.A., Kogge, P.M.: Scalability of Sparse Matrix Dense Vector Multiply (SpMV) on a Migrating Thread Architecture. In: Tenth Int. Workshop on Accelerators and Hybrid Exascale Systems (AsHES) held in conjunction with 34th IEEE Int. Parallel and Distributed Processing Symp. (May 2020)
29. Page, B.A., Kogge, P.M.: Scalability of Streaming on Migrating Threads. In: IEEE High Perf. Extreme Computing Conf. (HPEC) (Sept 2020)
30. Page, B.A., Kogge, P.M.: Scalability of Hybrid SpMV with Hypergraph Partitioning and Vertex Delegation for Communication Avoidance. In: Int. Conf. on High Perf. Computing and Simulation (HPCS 2020) (March 2021)
31. Page, B.A., Kogge, P.M.: Scalability of Streaming Anomaly Detection in an Unbounded Key Space on Migrating Threads. 2021 IEEE Int. Symp. on Parallel Distributed Processing (2021)
32. Rees, N.: SKA and its computing challenges. Tech. rep., SKA Square Kilometre Array (May 2017), https://indico.cern.ch/event/638811/attachments/1460553/2255823/SKA_Computing_Challenges-20170516.pdf/
33. Rolinger, T.B., Krieger, C.D.: Impact of traditional sparse optimizations on a migratory thread architecture. In: 2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3). pp. 45–52 (2018)
34. Springer, P.L., Schibler, T., Krawezik, G., Lightholder, J., Kogge, P.M.: Machine Learning Algorithm Performance on the Lucata Computer. IEEE High Perf. Extreme Computing Conf. (HPEC) (Sept 2020)
35. Wan, W., Kubendran, R., Eryilmaz, S.B., et al: 33.1 A 74 TMACS/W CMOS-RRAM Neurosynaptic Core with Dynamically Reconfigurable Dataflow and In-situ Transposable Weights for Probabilistic Graphical Models. In: 2020 IEEE Int. Solid- State Circuits Conf. - (ISSCC). pp. 498–500 (2020)
36. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Comm. of the ACM **52**(4), 65–76 (2009)
37. Young, J., Hein, E.R., Eswar, S., et al: A Microbenchmark Characterization of the Emu Chick. CoRR **abs/1809.07696** (2018)
38. Zhang, H., Hsieh, C.J., Akella, V.: HogWild++: A New Mechanism for Decentralized Asynchronous Stochastic Gradient Descent. In: 2016 IEEE 16th Int. Conf. on Data Mining (ICDM). pp. 629–638 (Dec 2016)