The Evolution of a New Model of Computation

Brian A. Page[§]
Laboratory of Physical Sciences (LPS)
College Park, MD, USA
bapage@lps.umd.edu
0000-0001-5563-9678

Peter Kogge
Computer Science and Engineering
University of Notre Dame
Notre Dame, IN
kogge@nd.edu

***Abstract*—The conventional model of parallel programming today involves either copying data across cores (and then having to track its most recent value), or not copying and requiring deep software stacks to perform even the simplest operation on data that is "remote", i.e., out of the range of loads and stores from the current core. As application requirements grow to larger data sets, with more irregular access to them, both conventional approaches start to exhibit severe scaling limitations. This paper reviews some growing evidence of the potential value of a new model of computation that skirts between the two: data does not move (i.e., is not copied), but computation instead moves to the data. Several different applications involving large sparse computations, streaming of data, and complex mixed mode operations have been coded for a novel platform where thread movement is handled invisibly by the hardware. The evidence to date indicates that parallel scaling for this paradigm can be significantly better than any mix of conventional models.**

***Index Terms*—multi-threading, thread migration, parallel scaling**

I. Introduction

This paper discusses new models of computation where data does not move, but where the computation itself moves to the data as needed. Such a reversal has two effects on computation. First, it removes the need to track copies of data and to ensure coherence between copies. Second, it reduces the latency of access — instead of a synchronous two-way round trip out and back to access some remote datum, just a one-way asynchronous outwards trip is needed. The latter also has the significant benefit of freeing up the core where the computation resided originally. With relentless multi-threading of both the hardware and the applications, this latter effect means that both hardware utilization and net system throughput can increase considerably for irregular applications, relative to conventional approaches.

The rest of the paper is organized as follows. Section II introduces a new model of computation where thread migration is ubiquitous and an integral part of the underlying hardware. Section III discusses a series of actual platforms that implement this model. Section IV provides evidence of the efficacy of our approach with real-world irregular applications. Section V concludes.

[§]The work reported here was performed while Dr. Page was a graduate student at the University of Notre Dame.

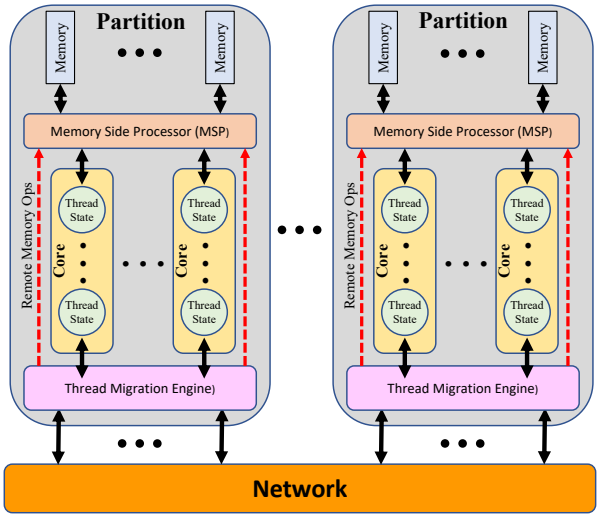

Fig. 1: A generic migrating thread architecture.

II. Intrinsic Migration

Fig. 1 shows a generic "migrating thread" architecture where computation can be migrated freely across a Partitioned Global Address Space (PGAS) memory structure, with direct support from the underlying hardware to perform such migrations automatically as needed. The basic building block of the architecture is a *partition*[1] that contains a *pool* of "cores" that can house thread states that migrate in or are spawned locally. There is also a set of memory blocks, each connected through a separate *memory channel* to the pool of cores. Between the cores and these channels are *memory controllers* that translate access requests from the cores to the timing signals needed by the memory channels. In Fig. 1 they are embedded in a *Memory Side Processor* (MSP) that, depending on the system, may include processing logic to perform operations "at the memory." Also attached to the core pool is a *Thread Migration Engine* (TME) that routes migrating threads from their origin cores to designated network ports. Once launched into the network, the thread state is transported to the target partition where it is unpacked and placed in a selected

[1]Many systems described later use a variety of terms (e.g., "nodelet") to describe a local combination of memory and processing; we use the term "partition" to be generic and consistent with the PGAS model.

core. Not shown in Fig. 1 is some sort of conventional *Stationary Processor* (SP) that can couple the system to the outside world, provide access to file storage, initiate applications, and manage the system as a host processor.

A. Basic Execution Model

With such an architecture, the migration model of computing is straightforward. A thread executing in some partition stays there as long as all memory references it makes are to locations in memory within the partition. As soon as the thread makes a reference to a location that is non-local, the hardware in the partition suspends the thread, packages the thread's state, and ships the package over the inter-partition network to the partition that holds the desired location. At that partition the thread state is unpacked and placed in any of the partition's cores, where execution resumes. Resources used by the migrating thread at the originating partition can be freed up for use by incoming threads.

This migration has two positive effects on overall performance of the parent program owning the thread. First, the migrating thread model halves the latency and avoids idling cores. In a conventional system, accessing a non-local location requires a two-way latency: out and back. Even with a sophisticated out-of-order core and load/store buffers that allow overlaps of memory and computation, very soon such latencies will cause processor stalls. These stalls represent hardware cycles which do not perform useful activities for the program for some period of time. Studies of big data applications on conventional processors (c.f. [1]) indicate that such long latency stalls are major factors affecting the performance of current systems.

The second effect is that there is no need in a migrating thread model for managing remote copies of memory locations in remote caches. The thread moves to the data being accessed rather than making a copy that then must be kept coherent. Thus, there is no need for cache coherency traffic as in conventional processors, freeing up cross-core bandwidth for other purposes. Studies similar to the above have again shown that such coherency traffic is a major limitation in parallel scalability when the data is sparse, and there is little reuse.

B. Pervasive Lightweight Multi-threading

A potential challenge with this approach is that such a system can see loads shift dramatically at unpredictable moments. Huge numbers of threads can descend unexpectedly on a particular partition, in larger numbers than available cores. While they could simply be buffered in queues until a core is free, that could result in threads with just a small amount of work to wait for extraordinarily long times, delaying the entire application.

One approach to addressing this challenge is to deeply multi-thread each core to accommodate very large numbers of thread states. Core microarchitectures that interleave instructions from different threads enable forward progress to be made on all threads, regardless of the number of threads. As threads migrate into a partition, they can be directed to the cores with the least number of active thread states. As threads migrate out, the cores free up their slots, and the remaining threads get more service. For applications that support significant thread-level parallelism, this provides a natural form of automatic load balancing.

"Spawning" new threads is usually a complex operation in conventional systems, often incurring significant software overhead to separate out state that is truly part of the computation (program counter, working registers) versus state that is part of the surrounding infrastructure (TLBs, caches, page tables, ...). In a migrating thread architecture, state related to infrastructure has by necessity already been architected out, and thus the cost of creating a new thread by simply copying working registers is greatly reduced, thereby making it easier to create large numbers of threads dynamically.

C. Memory Controllers and Ultra Lightweight Threads

One of the most productive features of the Cray T3D [2] was a facility whereby a thread could assemble and launch an operation to be performed adjacent to the memory controller of a location anywhere in the system. In particular, operations that performed "atomic" updates[2] to memory were of significant value in eliminating many of the parallel synchronization operations needed to allow parallel programs to execute correctly. Such remote atomic operations have continued to be an important capability supported by conventional systems. However, because of the deep cache hierarchies and the need to maintain cache coherency, the mechanisms needed for this support can get quite complex.

In a migrating thread architecture, another alternative becomes possible. Without the need for coherency, such atomic operations can be performed directly by the memory controller in a guaranteed uninterruptible basis, at very low latency, and with no need to consider possible retries. Thus, instructions can simply be added to the ISA to directly perform remote atomic operations. Further, and of perhaps even more value in future systems, mechanisms can be added to spawn from any thread an *ultra-light weight* (ULW) thread that migrates like any other thread to the target partition. However, instead of a program counter, such threads can contain one of a predefined memory operations, or perhaps a very short sequence of operations, and a bare minimum of operand values. When executed, such threads could operate directly in the memory controller at very high efficiency, thereby bypassing the cores that support longer lived multi-instruction threads.

[2]Such operations perform read-compute-write sequences to memory locations in ways that prevent any other memory operation to the same location from interfering.

III. Hardware Implementations

Table I summarizes a series of real platforms produced by Lucata Inc. (previously Emu Solutions) that have been implemented to support the migration model of computation. Several of these platforms are openly available at the Georgia Tech CRNCH Center[3], and were used for the experiments discussed in Section IV.

A. The Chick system

The Chick system [3] consists of 8 node boards where each node board supported 8 partitions in a single FPGA. Each partition was a single 64-bit multi-threaded *Gossamer core (GC)* and a single memory channel.

In contrast to an earlier prototype, the ISA of the GC dropped the ability to carry code, but was enhanced with more working registers and the ability to spawn ULW threads that would perform memory operations (variations of stores and atomic updates) on remote partitions without migrating the parent thread. The memory controller was upgraded to a smart *Memory Front End* (MFE) engine that was capable of handling atomic operations from either threads in the local GCs or incoming ULW threads from elsewhere.

Six Rapid I/O gen 2.7 network ports were shared among all 8 partitions on a node card, and driven by a *Migration Engine* (ME) that routed threads between partitions and/or through the appropriate network port. A chassis contained 8 such node boards interconnected by these links in a binary hypercube configuration, The result was a system with 64 partitions.

Each node board also held a shared SP consisting of a dual core POWER processor, memory, and an SSD. Each SP ran Linux and was capable of loading and storing the memory in the partitions, launching migrating threads into the system, monitoring progress of a code, and communicating with the outside world via a PCIe board.

B. The Pathfinder systems

The Pathfinder series of designs saw an upgrade in the technology and architecture. Technology saw use of more advanced FPGAs and memory. The architecture saw further expansion of the capabilities of the MFE to a full-fledged *Memory Side Processor* as pictured in Fig. 1 where remote memory threads can be performed directly, without having to use the cores. Also in the Pathfinder-S (currently in the Georgia Tech CRNCH Center) each card has become one partition, so that incoming threads can be placed in any of the partition's cores. There are still multiple memory channels, any of which can be accessed concurrently from any of the partition's cores. In addition, each memory interface has a small memory-side cache to increase effective bandwidth.

Architecturally, features were added, akin to a fence, that allow a thread to know when all outstanding ULWs it has launched have acknowledged completion. The same mechanism is used to prevent threads from migrating if in fact they have outstanding ULWs.

[3]https://crnch.gatech.edu/

C. Enhanced Programming Model

The original prototype system was programmed largely in low level assembly. Starting with the Chick the programming model and software support tool chain was upgraded to a modern LLVM-based C/C++ enhanced by a optimized extension of Cilk [4]. The key extensions include three Cilk keyword extensions reworked to match hardware-supported migration. The `cilk_spawn` keyword is a prefix to a conventional function call that converts it into a non-blocking procedure call that executes independently of the code following the call (called the "continuation"). In conventional implementations the thread performing the non-blocking call is taken from a pool of pre-established threads. In the migrating thread case, it is simply an inline spawn of a new thread that builds a cactus stack like frame on the call stack and provides the child thread with information as to what to do when the call completes. If for some reason a new thread cannot be spawned, then just as in conventional Cilk, the call becomes blocking - the calling program is stalled until the called procedure completes.

The Cilk keyword `cilk_sync` causes the thread executing it to wait for all child threads (created by `cilk_spawn`s in the enclosing block) to complete.

The `cilk_for` keyword looks syntactically like a conventional `for` but acts like a parallel for loop in other languages. The difference is that there is no assumed or implicit ordering to the loop evaluations, as each loop iteration can be executed by a separate thread. Notionally, the loop is converted into a tree of `cilk_spawn`s where each such spawn uses the body of the loop as an anonymous function. As with `cilk_spawn`, there is an implied `cilk_sync` before leaving the loop body.

In addition to the Cilk-based keywords, the tool chain includes intrinsic functions that allow the programmer to directly access parts of the architecture and ISA that are not in the vanilla Cilk model, such as atomic memory operations along with several thread control functions (save state, reschedule, etc.) that allow finer control over the scheduling of threads and memory operations.

D. Typical Program Structure

Many programs written using this tool chain have a similar three-level hierarchy of threads. First is the insertion from an SP of a master thread that migrates through all partitions that will be used by the program. At each partition it spawns a child thread whose job is to initiate any local data structures and fill in any initial replicated data values. Each of these child threads then spawns some number of worker threads to carry out the actual program. These workers may spawn additional workers as required by the program.

Feature	Gen 0 Prototype	Gen 1 Chick	Gen 2 Pathfinder-A	Gen 3 Pathfinder-S
Partition (cores and memories where accesses do not require migration) Characteristics				
Cores/Partition	1	1	1	24
Core Clock (MHz)	100	175	175	220
Runnable Threads/Core	Many	64	64	64
Mem. Channels/Partition	1	1	1	8
Memory/Channel (GB)	8	8	8	8
Memory Protocol	DDR3-166	DDR4-1600	DDR4-1600	DDR4-2400
Mem. Channel Width (Bytes)	8	1	1	2
Mem. Channel Peak B/W (GB/s)	2.6	1.6	1.6	4.8
Mem. Channel Access rate (G/s)	0.032	0.2	0.2	0.3
Channel Cache	No	No	No	Yes
Core Logic FPGA	Virtex 6 LX240T	Altera Arria 10	Altera Arria 10	Stratix 10
Node Card (card with multiple partitions) Characteristics				
Partitions/Node Card	4	8	8	1
Network Ports/Node Card	8	6	6	6
Network Protocol	PCIe-Gen2	SRIO 2.7	SRIO 2.7	SRIO 2.7
Injection B/W/Port (GB/s)	5	2.5	2.5	2.5
Injection B/W/Node Card (GB/s)	10	15	15	15
Chassis (Group of node cards) Characteristics				
Node Cards/Chassis	3	8	8	8
Chassis/System	1	1	1	2
Stationary Processor	Intel PC	Power E5500	Power E5500	Power E6500
SSD	None	1TB/Node card	1TB/Node card	1TB/Node card
Network Topology	Switched	Hypercube	Switched	Switched
Aggregate Characteristics				
Total Chassis	1	1	1	2
Total Partitions	12	64	64	16
Total Cores	12	64	64	384
Other	Coprocessor support			

TABLE I: Generations of Prototypes.

IV. Evidence of Usefulness

A new model of computation is of value only if systems (hardware and software) can together demonstrate that there are at least some problems for which solutions become faster or more efficient or scale better than solutions using conventional models. This section briefly reviews the results of several such comparative studies that used systems available at Georgia Tech.

A. SpMV

The product of a matrix by a vector to form another vector is an important part of many applications. When both the matrix and vector are dense (few zeros), modern architectures can operate at high efficiency. However, when especially the matrix is sparse (few non-zeros, and in irregular locations), processing typically becomes memory bound. Despite the horrible inefficiency, the matrices used in benchmarks like HPCG all fit in the memory accessible to a single core. SpMVs that handle much larger matrices are rapidly becoming important in many areas such as the processing of very large graphs [5], and parallel libraries are being developed to handle them (cf. the GraphBLAS API [6]) and the SuiteSparse implementation [7]). SpMV has been identified as part of the "13 dwarfs" [8] that are representative of the parallel patterns needed for emerging application areas.

1) Prior work

Despite the inefficiencies, a survey of several well-respected SpMV codes against multiple large matrices

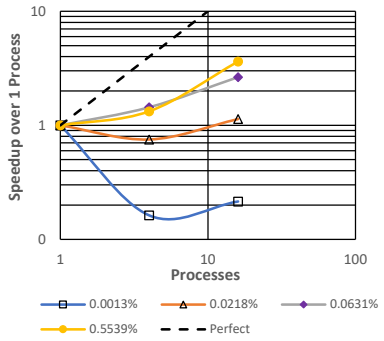

Fig. 2: Hybrid Speedup for 4 sparse matrices.

on 12 core shared memory systems [9] showed that speedups of up to only about 4X are possible[4], and when the time to load and unload the matrices into GPU memory is ignored, up to an 11X speedup using a single GPU (the speedup is negative when transfer times are included). The less than stellar scaling is a result of the underlying architecture relying heavily on coherent caches which are of little use in sparse environments.

The situation gets even worse in distributed memory environments due to load imbalances in the number of non-zeros per core and the relatively few operations that can be performed on one node before communication with another node must occur. One study [10] looked at hybrid algorithms where a shared memory multi-core,

[4]Using as a basis a single core code from Intel's MKL package

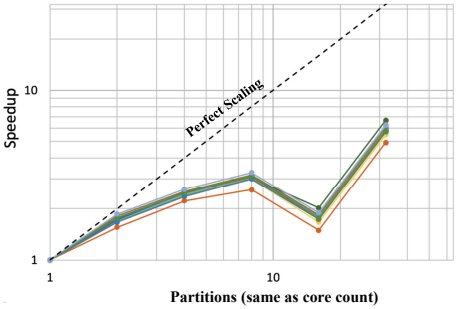

Fig. 3: SpMV with migrating threads.

multi-threaded algorithm was used within a node, and then multiple MPI processes using this code collaborated on the overall problem. Four different sparse matrices were taken from [11] with sparsities from 0.0013% to 0.55% non-zeros. The multi-core algorithm in isolation peaked at about 2 to 5X with somewhere between 6 and 10 threads (3 to 5 cores). The hybrid algorithm results are pictured in Fig. 2, where the different lines correspond to matrices of different sparsity. The x-axis is the number of MPI processes used, where each process was a 4-core multi-core code. The dotted line corresponds to perfect speedup. The key takeaways were that none of the matrices scaled anywhere near perfect, and even worse the most sparse demonstrated negative scaling - more resources slowed down the process.

An expanded study of a similar hybrid algorithm [12] with more (25) matrices and more nodes (72) and more cores per node (16) had similar results: the sparsest matrices had negative scaling immediately, and even the denser ones went negative once parallelism crossed about 16 processes. Instrumentation revealed that `MPI_Reduce` (to accumulate partial results from different processes) rapidly dominated, and by the time parallelism reached 81 processes, the ratio between communication and compute time exceeded 30X.

Similar studies looked at the same matrices but several significantly different architectures [13], [14], and even more matrices (1800 of them) [15], with similar results.

2) A Migrating Thread Implementation

The key takeaway from the above is that for SpMV the cost of collectives and communication very quickly swamps any advantage in processing that advanced core microarchitectures can give us. These effects show up because almost regardless of how a matrix is partitioned, a point is reached where partial sums must be combined, and that is where current architectures bog down.

Given that such "combining" looks like some sort of remote data operation, it is obvious to ask if the migrating thread model can help. An implementation study using the Chick at Georgia Tech [16] considered several variations of migrating thread algorithms over the same matrices discussed above.

The algorithm finally used had each partition compute its partial sum for each segment of each row with non-zeros in it, and send a remote atomic add to accumulate that partial into the correct element of the output vector approach. Two distribution techniques for the matrix were tried: a naive round-robin striping where column i is placed on partition $i \ mod \ P$ where P is the number of available partitions, and use of the hypergraph partitioner used in [14]. There is only one copy of the dense vector which is striped across the partitions. These threads migrated as needed. A remote add ULW thread added the partial sum into the correct output vector location. All 25 prior matrices were used.

Fig. 3 diagrams the results for the naive round-robin matrix distribution and 64 threads per partition. While not perfect, the scaling stays positive the entire time. The dip at 16 partitions represents the points where the number of partitions exceeds a single board and the migrating threads have to use the network to get to the correct partition. However, the performance then goes up considerably as more node boards are used. Better network technology is liable to alleviate this issue.

A second observation is the tight banding of results - there is no major spreading tied to matrix sparsity. Also, not shown are the results from the expensive pre-computation using the hypergraph partitioner that not only did little to improve the best results, but had a significantly larger distribution as a function of matrix sparsity, and are actually worse for several matrices.

The key takeaway is that not only does the migrating thread architecture support better scalability with little sensitivity to sparsity, it does so with very simple code.

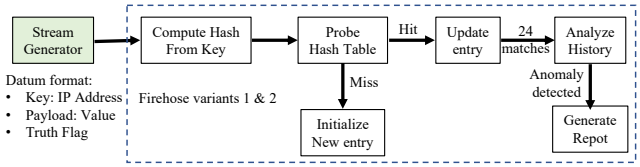

Fig. 4: Data streaming through the Firehose benchmark.

B. Streaming

The processing of continual streams of data has grown in importance as we enter a connected, real-time world. Two aspects of most streaming applications are that data needs to be processed at its arrival, usually using a finite bounded amount of data from prior inputs, and that an answer to some query needs to be considered each time such data modifies this prior data. Such "histories" are often large, complex, irregular data structures such as graphs or big data repositories, and the queries represent the "discovery" of some new property that they possess as a result of the new data (cf. [17]–[20]).

General studies of streaming algorithms include [21]–[24], with a small but growing suite of software support packages [25]–[28]. Unfortunately, when implemented

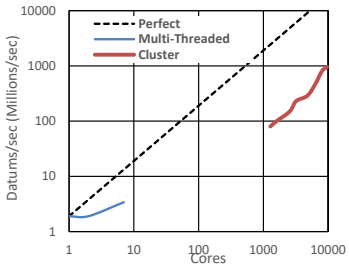

Fig. 5: Firehose scalability on conventional systems.

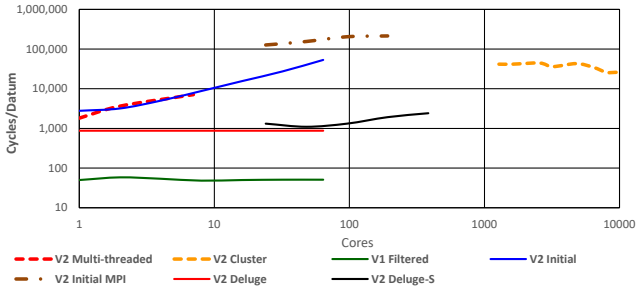

Fig. 6: Machine cycles per datum for different Firehose implementations. The dashed lines are conventional implementations; the solid are from the migrating thread experiments. Lower is better. Perfect scaling is flat.

on conventional architectures such applications often become horribly inefficient, especially when attempts are made to use some sort of parallelism.

The *Firehose* benchmark[5] [29]–[31] is a stand-in for cyber-security streaming applications where information from different incoming internet packets (called *datums*) must be aggregated in some way so that different kinds of "events" can be recognized, and potential "anomalies" detected. The performance metric is "datums/sec:" how many such datums can be pushed through the system per second without dropping too many of them.

Fig. 4 diagrams two of the three variants of Firehose. A stream of synthetic packets are generated, each representing an IP packet holding a hex IP address (expressed as an ASCII string) and a payload (a boolean for the benchmark)[6]. The IP field is used to probe a large hash table, and on a match, a count field in the entry is incremented and the payload added to a sum in the same entry. When the 24th datum is detected, the payload sum is inspected, and if it exceeds some threshold, an *anomaly report* is generated. The difference between the first two variants is that in the first there is a known limit to the number of datums that may be generated; the second has no limit but does constrain the number of unique IP addresses that may be in play at any time.

To show the issues with such a kernel on conventional architectures, Fig. 5 graphs two sets of data from the Firehose website that describe performance (as a throughput) versus the number of cores used in the implementation. The "perfect" line represents the throughput from a single core times the number of cores. The blue line represents implementation on a multi-core chip using shared memory for data exchange. Seven cores gives less than a factor of two throughput. The red line represents throughput measured from a large cluster, starting at a configuration of about 1000 cores. While there is good scaling beyond that, the throughput "per core" on this red curve is about $1/30^{th}$ of that of a single core. The penalty is due solely once again to the communication between cores needed to advance the computation.

The migrating thread FPGA-based hardware forced some simplifications to be made. Because of the lack of high bandwidth inputs, the incoming streams of unprocessed datums were precomputed and placed in memory. Also, because of the lack of string handling instructions, the IP headers in each packet were stored in hex rather than ASCII. Both of these artificially inflate the throughput, but still allow scaling comparisons, as pictured in Fig. 6 which for each experiment shows the number of machine cycles per datum processed as a function of the number of cores[7].

1) Variant 1 Experiments

The first Variant 1 experiments [32] ran on the Chick at Georgia Tech and used an open address hash table that was striped across the partitions in use. Since the maximum number of IP addresses was known in advance, the hash table was big enough so that no probe collisions would occur. Two implementations were tested. The first launched multiple threads on each partition, each of which would handle one of the datums pre-stored on the partition. On average, each thread would migrate once from the partition with the datum to the partition holding the hash table entry, and once to return. The second implementation was similar but included at each partition a "prefilter" to catch datums that have already accumulated enough hits to send a report.

The study varied both the number of partitions and the number of threads initiated on each partition. For the unfiltered case, increasing the former increased throughput until saturation was reached. Higher thread counts per partition lowered the point at which saturation occurred, which indicates that migrations were the bottleneck. The filtered version, however, did not exhibit this problem, and throughput increased as either (or both) initial threads or partitions increased. Overall throughput was over 220 million datums/s - 20X that of the unfiltered case, with migrations reduced by orders of magnitude. As pictured in Fig. 6 the machine cycles per datum were around 50, which is $1/30^{th}$ that of

[5]https://firehose.sandia.gov, https://stream-benchmarking.github.io/firehose/

[6]A "truth flag" is also included so that implementations can check if they got the correct results

[7]This metric was chosen to normalize the difference between ASIC chips and FPGA implementations.

the best conventional (single core) implementation, and approaches $1/1000^{th}$ that from the conventional cluster.

2) Variant 2 Initial Implementation

A number of implementations were created for variant 2. The largest performance hit for this variant is the need to "age out" entries in the hash table over time as the set of active keys changes.

The first of the variant 2 implementations [33] kept the pre-stored datum set, but increased the realism of the implementation by performing the ASCII to binary conversion as in the benchmark spec. Also the implementation divided the pool of threads into two classes: *producers* and *consumers*. The producer threads read out the raw datums, did the string conversion, determined which partition held the hash table entry for that key, and inserted the entry into a partition-local queue. The consumer threads managed aging entries as needed, updated them to reflect new data, generated reports when appropriate. and kept local key tables with LRU data to determine "oldest" entries.

The optimal combination of initial threads per partition was 8 producers and 16 consumers. With a small number of datums per partition, the algorithm achieved near perfect scaling with respect to number of cores (again using the Chick at Georgia Tech where each partition had one core). With larger (but more realistic) numbers of threads, the performance plateaued due to an increase in the number of times aging out of entries was necessary. As pictured in Fig. 6 the performance in terms of machine cycles per datum tracked the shared memory conventional configuration.

As a point of comparison, an MPI version of the same algorithm was implemented and run on a conventional cluster of 24 core 2.66 GHz AMD EPYC 7451 chips. Performance in terms of cycles per datum are worse than the migrating thread by a factor of around 4.

3) Variant 2 - Deluge

The major issue with the prior design was that since each consumer thread handled a distinct set of keys (to avoid inter-thread synchronization), "hot spots" would develop for those key values that appeared in many datums. These consumer threads would thus have far more work to do than the other consumers, and would thus gate the execution.

Another implementation of variant 2 called "Deluge" [34] did not discriminate between producer and consumer, and thus avoided the queuing between them. A more "Actor" model was adopted where separate "producer" partitions hold threads that dynamically generate datums using the official Firehose spec. At completion of the datum generation (corresponding to the arrival in a real system of the datum), the thread will hash the address in the datum and migrate to the appropriate hash entry for processing. Given the now possibility of contention, locks are necessary on each entry, and were handled by built-in atomic memory ops. The implementation again included an LRU list to help when collisions occur and aging needs to be employed. After completing an update, the thread returns to the producer partition to generate another datum. Importantly unlike any of the prior implementations, nodes holding the hash table had no permanently resident threads.

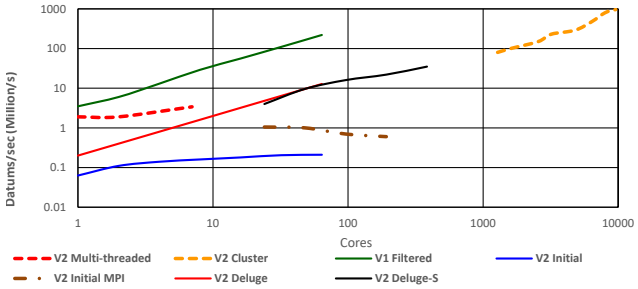

Fig. 7: Firehose Measured Performance in Datums/s. The solid lines are for the migrating thread systems whose core clocks are 10-20X slower than the conventional systems (dashed lines). Higher is better.

As pictured as the red "V2 Deluge" line in Fig. 6, the machine cycles per datum is essentially flat (i.e. near perfect scaling), and considerably lower than any of the prior implementations, particularly including the shared memory on conventional cores.

4) Implementation on a bigger system

All the above ran on the Chick system at Georgia Tech. Recently an upgraded Pathfinder-S system became available, and the Deluge algorithm run on it [35], with a varying number of partitions devoted to the hash tables. All partitions not hosting hash entries were producing sites. The line labelled "V2 Deluge-S" reflects this experiment. As can be seen, the overlap with the prior Deluge curve is very good, and there is only limited growth on cycles per datum as the number of cores increase.

5) Summary Measured Performance

Fig. 7 shows the actual measured datums per second for each of the above implementation versus the number of cores. Remembering that the cores in the migrating thread systems are currently at least an order of magnitude slower than the cores in the conventional (dashed line) systems, the comparison is stunning. The net conclusion from this is that using the Deluge algorithm, the migrating thread architecture has the potential to significantly beat conventional implementations if put on a common technology footing (i.e. cores in an ASIC running at comparable clocks).

C. Machine Learning

Machine learning (ML) is becoming an essential part of much of modern computing. The predicting/inferencing part of ML is often straightforward to perform efficiently, especially by purpose-built hardware (cf. Google's TPU [36]). However, learning is far more complex. One well known ML technique is Stochastic Gradient Descent

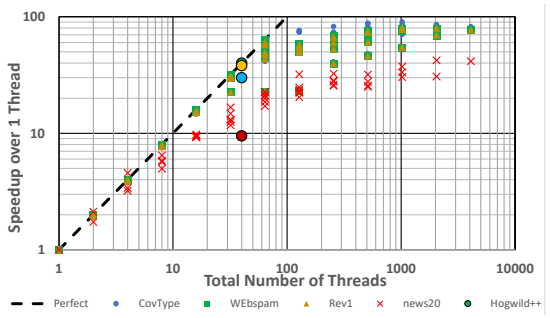

Fig. 8: Wildebeest Speedup.

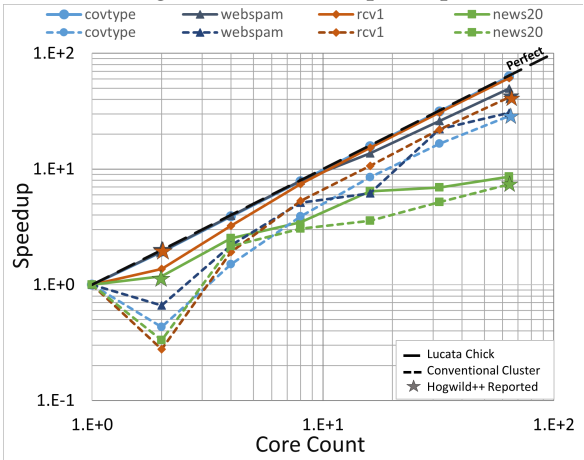

Fig. 9: Passel Speedup.

(SGD) that repeatedly uses training data against a current model vector, determines when a projection (inference) is incorrect, and uses the "direction" of the error to modify the model vector slightly. A series of studies have explored a wide range different parallel SGD algorithms [37]–[40], largely on multi-core chips, with at best limited scalability. Memory issues such as inter-socket coherency traffic and false sharing have been primary. Perhaps the most influential of these algorithms was Hogwild! [41], [42] - a multi-threaded implementation where independent trainers could update the same model vector at the same time with a feature-at-a-time atomic update. This algorithm demonstrated scaling of up to 4.5X on 10 core systems for sparse and very sparse problems where the there were very few updates to make from each training datum. An important follow-on, Hogwild++ [38], provided somewhat higher speedups (up to 9.5X on 40 cores, and again for the sparse problems for which the conventional algorithms do poorly) by creating "clusters" of cores where each cluster had an affinity to some memory channel. Each cluster ran a separate training session using Hogwild! on just local data, but with a rotating "token" whose arrival on a cluster caused it to exchange changes in its model vector since the last token with the next cluster. This exchange is in both directions.

1) Implementation on the Chick

Several migrating thread codes were based on Hogwild! and Hogwild++. In the first, called Wildebeest [43], a Hogwild++ implementation matched a partition with a cluster, and used a variety of migration-based mechanisms for the inter-cluster communication. A remote atomic ULW would be launched by one cluster to set a flag in the next cluster. When one of the training threads detects this, it starts comparing the current model vector with the previous one, and sending out a stream of updates (feature numbers and updates), again using remote atomic ULWs. The updates modify the upstream cluster's model vector; the feature numbers are queued. Then, when the token is passed, a thread on this next cluster starts working through the queue of feature numbers, determining what has changed in that position since the last token passing, and sending an update the other way. Fig. 8 diagrams some results on the Chick. The four colored markers provide speedup as a function of the total number of threads for four different data sets. The outlined circles represent data for the same data sets from the original Hogwild++ paper. As can be seen, the migrating thread had better scalability in all cases, and was significantly better for the sparsest of data sets (the red points).

A second code, called Passel [44], replaced the queue of modified features that was sent upstream with individual threads that performed the updates directly. In this case, the reference data is the same algorithm implemented on a conventional cluster. The results in Fig. 9 are even better than before, with the dotted lines representing the conventional code and the solid ones the migrating threads. Again the latter are clearly superior in all cases.

V. Conclusion

Sparsity and irregularity seems to be becoming the Achilles heel of parallel codes that scale well through large numbers of cores, especially for systems where mixed shared memory multi-threading and distributed message passing are needed to use all the resources efficiently. This paper looked at a variety of different problems that all exhibited this phenomena. Strong scaling of sparse linear algebra operations scale miserably, even while their dense counterparts are the bread and butter of conventional parallel benchmarks. Streaming of data through large complex data structures scales but at horrible inefficiencies. Even the hottest of emerging applications, machine learning, has significant problems when the problems are large and sparse.

In contrast, a new paradigm based on migrating threads seems to scale better in all these cases. These numbers should only get better as implementation of the basic architecture are in comparable ASIC technologies.

A future paper will explore all these cases, and add several more, in more detail, with a detailed projection as exactly how much better implementations in comparable technology will get.

Acknowledgments

This work was supported in part by NSF grant CCF-1822939, 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 Lucata systems there.

References

[1] Z. Jia, J. Zhan, L. Wang, C. Luo, W. Gao, Y. Jin, R. Han, and L. Zhang, "Understanding big data analytics workloads on modern processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1797–1810, 2017.

[2] Cray, "Cray t3d system architecture overview manual." [Online]. Available: ftp://ftp.cray.com/product-info/mpp/T3D_Architecture_Over/T3D.overview.html

[3] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the emu system architecture," in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, 2016, pp. 2–9.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/209936.209958

[5] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. USA: Society for Industrial and Applied Mathematics, 2011.

[6] B. Brock, A. Buluç, T. Mattson, S. McMillan, and J. Moreira, "The GraphBLAS C API Specification version 2.0.0," https://graphblas.org/docs/GraphBLAS_API_C_v2.0.0.pdf.

[7] T. Davis, "Suite sparse," https://github.com/DrTimothyAldenDavis/GraphBLAS.

[8] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, p. 56–67, oct 2009. [Online]. Available: https://doi.org/10.1145/1562764.1562783

[9] M. Grossman, C. Thiele, M. Araya-Polo, F. Frank, F. O. Alpak, and V. Sarkar, "A survey of sparse matrix-vector multiplication performance on large matrices," *ArXiv*, vol. abs/1608.00636, 2016.

[10] B. Bylina, J. Bylina, P. Stpiczyński, and D. Szałkowski, "Performance analysis of multicore and multinodal implementation of SpMV operation," in *2014 Federated Conference on Computer Science and Information Systems*, 2014, pp. 569–576.

[11] T. Davis, "Suite sparse matrix collection," https://sparse.tamu.edu/.

[12] B. A. Page and P. M. Kogge, "Scalability of hybrid sparse matrix dense vector (SpMV) multiplication," in *2018 International Conference on High Performance Computing and Simulation (HPCS)*, 2018, pp. 406–414.

[13] ——, "Scalability of hybrid SpMV on Intel Xeon Phi Knights Landing," in *2019 International Conference on High Performance Computing and Simulation (HPCS)*, 2019, pp. 348–357.

[14] ——, "Scalability of hybrid SpMV with hypergraph partitioning and vertex delegation for communication avoidance," in *Int. Conf. on High Performance Computing and Simulation (HPCS 2020)*, 2021.

[15] S. Usman, R. Mehmood, I. Katib, A. Albeshri, and S. Altowaijri, "Zaki: A smart method and tool for automatic performance optimization of parallel SpMV computations on distributed memory machines," *Mobile Networks and Applications*, 07 2019.

[16] B. A. Page and P. M. Kogge, "Scalability of sparse matrix dense vector multiply (SpMV) on a migrating thread architecture," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 483–488.

[17] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 623–632. [Online]. Available: http://dl.acm.org/citation.cfm?id=545381.545464

[18] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '08. New York, NY, USA: ACM, 2008, pp. 16–24. [Online]. Available: http://doi.acm.org/10.1145/1401890.1401898

[19] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarra-Miranda, C. Hastings, K. Madduri, and S. C. Poulos, "Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation," *Georgia Institute of Technology, Tech. Rep*, 2009.

[20] D. Ediger, K. Jiang, J. Riedy, and D. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 05 2010, pp. 1 – 8.

[21] P. A. Bernstein and N. Goodman, "Timestamp-based algorithms for concurrency control in distributed database systems," in *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, ser. VLDB '80. VLDB Endowment, 1980, pp. 285–300. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286887.1286918

[22] A. McGregor, "Graph stream algorithms: A survey," *SIGMOD Rec.*, vol. 43, no. 1, pp. 9–20, May 2014. [Online]. Available: http://doi.acm.org/10.1145/2627692.2627694

[23] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Theor. Comput. Sci.*, vol. 348, no. 2, pp. 207–216, Dec. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2005.09.013

[24] P. M. Kogge, N. Butcher, and B. Page, "Introducing streaming into linear algebra-based sparse graph algorithms," in *HPCS 2019 (nominated for best paper)*, July 2019.

[25] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," in *Bulletin of the Technical Committee on Data Engineering*, Dec. 2015.

[26] J. Riedy and D. Bader, "Stinger: Multi-threaded graph streaming," 05 2014.

[27] S. J. Plimpton and T. Shead, "Streaming data analytics via message passing with application to graph algorithms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 8, 5 2014.

[28] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao, "Consistency and completeness: Rethinking distributed stream processing in apache kafka," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2602–2613. [Online]. Available: https://doi.org/10.1145/3448016.3457556

[29] K. Anderson, "Streaming benchmarks firehouse and experiences with waterslide," in *Chesapeake Large Scale Data Analytics Conf.*, 2016.

[30] J. Berry and A. Porter, "Stateful streaming in distributed memory supercomputers," in *Chesapeake Large Scale Data Analytics Conf.*, 2016.

[31] J. Eaton, "Firehose, pagerank, and nvgraph: Gpu accelerated analytics," in *Chesapeake Large Scale Data Analytics Conf.*, 2016.

[32] B. A. Page and P. M. Kogge, "Scalability of streaming on migrating threads," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–8.

[33] ——, "Scalability of streaming anomaly detection in an unbounded key space using migrating threads," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 157–175. [Online]. Available: https://doi.org/10.1007/978-3-030-78713-4_9

[34] ——, "Deluge: Achieving superior efficiency, throughput, and scalability with actor based streaming on migrating threads," in

2021 IEEE High Performance Extreme Computing Conference (HPEC), 2021, pp. 1–6.

[35] ——, "Greatly accelerated scaling of streaming problems with a migrating thread architecture," in *2021 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2021, pp. 11–18.

[36] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[37] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231. [Online]. Available: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[38] H. Zhang, C. J. Hsieh, and V. Akella, "Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent," in *2016 IEEE 16th Int. Conference on Data Mining (ICDM)*, Dec 2016, pp. 629–638.

[39] C. D. Sa, C. Zhang, K. Olukotun, and C. Ré, "Taming the wild: A unified analysis of hog wild! -style algorithms," in *Proc. of the 28th Int. Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 2674–2682.

[40] C. Zhang and C. Ré, "Dimmwitted: A study of main-memory statistical analytics," *Proc. VLDB Endow.*, vol. 7, no. 12, p. 1283–1294, Aug. 2014. [Online]. Available: https://doi.org/10.14778/2732977.2733001

[41] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. of the 24th Int. Conference on Neural Information Processing Systems*, ser. NIPS'11. USA: Curran Associates Inc., 2011, pp. 693–701. [Online]. Available: http://dl.acm.org/citation.cfm?id=2986459.2986537

[42] L. M. Nguyen, P. H. Nguyen, M. van Dijk, P. Richtárik, K. Scheinberg, and M. Takáč, "SGD and Hogwild! convergence without the bounded gradients assumption," 2018. [Online]. Available: https://arxiv.org/abs/1802.03801

[43] B. A. Page, "Scalablity of irregular problems," Ph.D. dissertation, Univ. of Notre Dame, October 2020.

[44] B. A. Page and P. M. Kogge, "Passel: Improved Scalability and Efficiency of Distributed SVM using a Cacheless PGAS Migrating Thread Architecture," in *12th IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (Scala)*. IEEE, 2021, pp. 1–8.