Scalability of Streaming on Migrating Threads

Brian A. Page
Dept. of Computer Science and Engr.
Univ. of Notre Dame
Notre Dame, IN, USA
bpage1@nd.edu

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

***Abstract*—Applications where streams of data are passed through large data structures are becoming of increasing importance. Unfortunately, when implemented on conventional architectures such applications become horribly inefficient, especially when attempts are made to scale up performance via some sort of parallelism. This paper discusses the implementation of the Firehose streaming benchmark on a novel parallel architecture with greatly enhanced multi-threading characteristics that avoids the conventional inefficiencies. Results are promising, with both far better scaling and increased performance over previously reported implementations, on a prototype platform with considerably less intrinsic hardware computational resources.**

***Index Terms*—Emerging Architectures, Streaming, Scalability, Communication Overhead, HPC**

I. Introduction

Applications where streams of data are passed through large data structures are of increasing importance. Particular areas include graphs and big data [4], [5], [3], [15], [11], [12]. Prior studies of streaming include [6], [14], with a small but growing suite of software support packages [8], [17], [16]. Unfortunately, when implemented on conventional architectures such applications become horribly inefficient, especially when attempts are made to scale up performance via some sort of parallelism.

This paper focuses on the Firehose [2], [7], [10] benchmark as a framework for study. The benchmark is a stand-in for 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. Much of the data reported on the Firehose website[1] demonstrates a variety of issues with scaling it to use multiple cores. If an architecture cannot do well on Firehose, then its unlikely to do well on more complex apps. This makes it a good target for attempt similar solutions on alternative architectures[2].

One novel architectural approach that avoids the scaling limitations found in conventional architectures uses a suite of advanced multi-threading in a highly scalable parallel architecture, including the ability for a thread handling a particular stream datum to *migrate* with the datum to whatever hardware node holds the part of the data structure to be accessed. Such migrations are handled directly in the hardware, without any intervening software, and provide very efficient support for mobile atomic memory operations. Our results show this architecture obtains vastly superior performance while using less traditional "horsepower" than conventional systems.

This paper discusses initial scaling results from the implementation of a variation of the Firehose benchmark on such a new architecture. Section II provides background. Section III discusses the algorithms. Section IV reviews the experimental setup. Section V evaluates the results, and Section VI concludes.

[1]https://firehose.sandia.gov

[2]An earlier version of this paper was presented as an extended abstract at the 2020 Int. Workshop on Innovative Architectures (IWIA), in Feb. 2020. The implementation in this paper is a major revision with a much deeper analysis.

II. Background

A. Firehose Streaming Benchmark

Firehose resembles a cyber-security like streaming function where incoming IP packets are to be monitored. When some number of packets with the same IP address have been detected, the payload fields are examined for potential anomalies, and if detected, a report issued. The benchmark has three versions. The first two assume incoming packets have three fields formatted as ASCII strings. The first, the *key*, is an IP address that when converted from ASCII represents a 64-bit unsigned integer. The second, the *payload*, is a value of "1" or "0." The third is a *truth flag* that indicates if this packet is part of an "anomaly" sequence. This field is only used when the implementation makes a call to verify if the call was correct.

For the first two versions, the key field is used to look for matches in a giant hash table. At each match, a "match count" field is incremented in the table entry. In addition if the payload is a "1," a separate payload count is incremented. When the match count reaches 24, the payload field is tested. If it is 4 or less, an anomaly report is generated. No IP matches in the hash table causes a new hash entry to be created.

The first benchmark variant is primarily for testing, and the data generator ensures that there will never be more than 128K unique key values. The second is similar in that at one time there will not be more than 128K unique keys, but it has no constraint in the total number of unique keys over time. The third version is a more complex two-phase process described in the website.

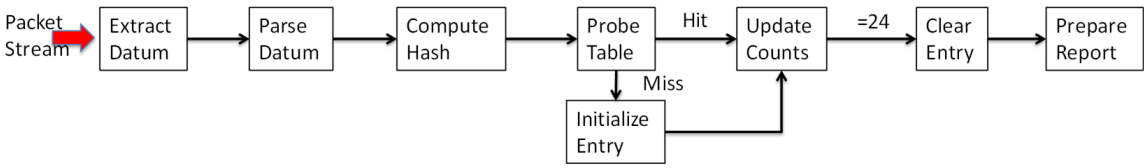

Fig. 1: The Firehose Data Flow.

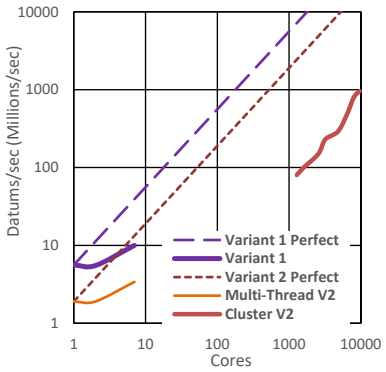

Fig. 2: Reported Scaling Numbers (Mostly from Variant 2).

For all three benchmarks, a "biased power law" data generator[3] creates synthetic data streams. The datum stream associated with a key may have two distributions of payload values. In the normal case, the payload is chosen equally randomly from a "1" or a "0". In the anomalous case, the payload values are biased toward "0."

Fig. 2 diagrams scaling results obtained in previous Firehose studies [1], [2], [10] and which we used from comparison against our implementation. Table I summarizes the major characteristics of the microprocessors used in the studies shown in Fig. 2. The curves for small core counts represent scaling data for multi-threaded implementations of both Variant 1 and 2 as taken from the Firehose website. The system for the Variant 1 result was a dual socket node where each socket was an Intel X5690 six-core processor. This data shows relatively poor scaling, with 7 cores providing 10 million datums per sec, less than twice that of a single core (5.6 million datums/s). The discrepancy is most likely due to a combination of coherency traffic and the need for expensive guaranteed atomic memory operations when the hash table entries are to be updated.

The Variant 2 data has two parts: data from the same system as the Variant 1, and data from a multi-node dual 6-core socket Cray CS-300 using Intel E5-2670 processors [7]. The former achieves 1.9 million datums/s on one core. The latter curve shows good weak scaling, but at an equivalent performance level per core of 0.6-0.1 million datums/s per core. This is up to 30X less than what perfect scaling from one core would have brought. The reason for the huge loss in efficiency per core is the software stack needed to handle the queuing and streaming of data from one physical node to another.

[3]see https://firehose.sandia.gov/doc/generators.html\#bias

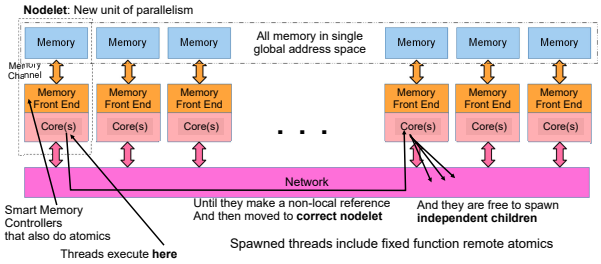

Fig. 3: The Migrating Thread Architecture.

B. Migrating Thread Architecture

A migrating thread architecture [13] is one where the underlying hardware, not software, moves the state of a thread as required during execution. Fig. 3 diagrams such an architecture as implemented by Emu Solutions [9] (now Locata, Inc.). The basic unit, a **nodelet**, is a memory module, its controller and some number of multi-threaded cores. All the memory in the collection of nodelets reside in a common address space. A network connects all nodelets. A thread runs in a core until it makes a memory reference that is not contained in that nodelet's memory. The hardware then puts the thread to sleep, packages it, and moves it over the network to the correct nodelet, where it is unpacked and restarted. A thread can spawn independent child threads. Also, the memory controller can perform directly atomic operations as close to memory as possible. Finally, very lightweight threads can be spawned to perform remote memory operations without moving the parent.

The current prototype used in this study is housed at Georgia Tech's CRNCH center[4]. It has up to 64 nodelets, each with 8GB of memory and one 175MHz multi-threaded core. These nodelets are packaged 8 to a **node board** which supports a RapidIO-based network. A dual core POWER microprocessor (called an **SC**) on each node board runs Linux, manages a local SSD, and launches migrating threads into the system. The nodelet logic on each board is implemented in an FPGA. The last row of Table I summarizes the characteristics of a node board. A larger system is in development.

In comparison to either of the two microprocessors used in the reference data, the aggregate compute cycles (number cores times clock rate) of the nodelet cores on a node card is $1/14$'th of either the other two. The actual comparison is probably lower than this as the nodelet cores are single issue and both the Intel cores are multi-issue. The node board aggregate memory bandwidth is about $1/3$ to $1/4$'th the others, but, because of the memory channel design used

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

Processor	Cores	Core Clock	Memory Channels	Per Channel		Total per Module	
				Bandwidth GB/s	Access Rate G/s	Bandwidth GB/s	Access Rate G/s
Intel Xeon X5690	6	3.46 GHz	3	10.7 GB/s	0.166 G/s	32 GB/s	0.5 G/s
Intel E5-2670	8	2.6 GHz	4	12.8 GB/s	0.200 G/s	51.2 GB/s	0.8 G/s
Emu Chick Node Card	8	0.175 GHz	8	1.6 GB/s	0.2 G/s	12.8 GB/s	1.6 G/s

TABLE I: Processor Characteristics.

in the nodelets, the ability of a node board to handle different independent memory accesses is between 2X to 3X higher that of either microprocessor.

The programming tool chain is based on Cilk, C with a prefix to function calls to spawn new threads, a sync primitive to wait for a set of children to complete, and a parallel *forall* to have a set of independent threads cooperate on a loop. Supported intrinsics include a rich set of atomic operations. Threads may spawn either child threads of equal capabilities to themselves, or **remote atomic threads** that migrate to target memory locations independent of the parent thread, but can only perform a limited set of atomic operations at the target.

III. Algorithms

We developed two implementations of the Firehose benchmark variant 1 analytic engine. Our first implementation was based on the C++ version from the Firehose website. Our second version significantly reduced migrations. In both cases, because of limitations in the prototype's I/O, the incoming data is placed in nodelet memories exactly as it would have if the data can come in externally through the SCs.

A. Baseline Implementation

Alg. 1 assumes each nodelet has a share of the datums which have been stored into a local $datums$ array on each nodelet. A single shared hash table called $state$ is visible to all threads and nodelets. It also assumes T threads are spawned on each nodelet, with each thread having a thread id i from the range 0 to $T-1$. $state$ is an open address hash table which is represented via a 1-Dimensional array striped across all nodelets. Open address hash tables rectify hash collisions by performing linear probing until an empty entry is located. However in our current implementation we set the size of $state$ equal to that of the max possible number of unique keys generated. In doing so we guarantee every possible key has a unique hash table entry and therefore no collisions are possible. This is consistent with variant 1 rules, but will change when the code is upgraded to variant 2 of the benchmark.

Each thread processes a single datum at a time in the while loop on line 2. During processing, a thread accesses the datum array on the nodelet the thread was spawned and acquires the key, payload, and bias flag values for the current datum. The $state$ entry for the current key is determined by computing the hash function $hash = key \ \% \ sizeof(state)$. This is again consistent with variant 1 rules, but will change when the code is upgraded to variant 2 of the benchmark.

Once the $state$ table entry has been determined the state stored within $state[hash]$ must be updated. Updating a key state requires incrementing the hit counter as well as adding the payload value of the current datum to the payload sum for the given key as these values are used to check if an event and or anomaly are to be triggered. All of these updates are done atomically, preventing race conditions between threads handling different datums. The hit counters and the payload are combined in a single 64b word to allow simultaneous updating.

Upon updating the state for the current key, line 8 checks if any key is seen 24 times. When such an event is triggered, the payload sum is checked for a value less than 4, and if so a true anomaly has been triggered. The flag field in each datum is then used to check the accuracy of the analysis by indicating if the report agreed with the expected results. Lines 11 to 14 update (again atomically) global statistics based on what the datum's bias flag indicated was the true answer.

Algorithm 1 Multithreaded FireHose variant 1:
T = number of worker threads spawned
i = thread id (initially $i < T$)
$datumCount$ = num datums assigned to nodelet
$datums$ = array storing all datums assigned to nidelet
key = key value for current datum
p = payload value for current datum
f = bias flag for current datum
$hash$ = state element id for current datum's key
$state$ = array containing payload and hit counts for all keys
$hits$ = observed key occurrences

```
1:  procedure AnalyzeDatums(i)
2:      while i < datumCount do
3:          key ← datums[i].key; p ← datums[i].payload
4:          f ← datums[i].bias; hash ← key%100000
5:          if state[hash]=empty() then initstate(hash)
6:          state[hash].hits++
7:          state[hash].pSum += p
8:          if state[hash].hits = 24 then
9:              events++
10:             if state[hash].pSum < 4 then
11:                 if b = true then truePositives++
12:                 else falsePositives++
13:             else if b = true then falseNegatives++
14:             else trueNegatives++
15:         i += T
16:     end while
```

B. Prefiltering Implementation

The major remaining issues with Alg. 1 lie in the updates to the hash table. Given the range of possible IP values, it is likely that a large percentage of updates from any thread will be to parts of the hash table that are on a that is on a

physically different nodelet. Additionally the data generator utilizes a power law distribution to produce the keys, meaning approximately 20% of keys will be present in 80% of all datums generated. Given the number of datums generated along with the number of nodelets used, it may be possible to flood a nodelet with updates to high occurrence keys.

To decrease the likelihood of this overloading we developed Alg. 2, which limits updates to the global hash table $globalState$ by first checking a local hash table $localState$. Prefiltering remote updates has been used in previous studies, such as breadth first search in which the graph contains vertices with very high out-degree. Both tables are implemented in the same manner as striped $state$ table used in Alg. 1, except that each nodelet's $localState$ table is only accessible from the nodelet on which it resides.

Finally, it is possible that threads on nodelets other than the one that found the $= 24$ event will not have their $deadFlag$ set, and try to update the global state. When they do, they will find the global count > 24 (line 20), and instead of continuing the update simply set their own local state entry to dead (line 21). For keys that may be at the far end of the power law distribution, this means that any global entry is modified at most 24 plus the number of threads times, and probably closer to 24 plus the number of nodelets. As will be seen this has a dramatic effect on performance.

Algorithm 2 Filtered FireHose:
Same parameters as Algorithm 1, with additions:
$globalState$ = global payload and hit counts for all keys
$localState$ = local status of each key
$dead$ = key exhaustion status

```
1:  procedure AnalyzeDatumsFiltered(i)
2:      while i < datumCount do
3:          key ← datums[i].key; p ← datums[i].payload
4:          f ← datums[i].bias; hash ← key%100000
5:          if globalState[hash] = empty() then initglobal(hash)
6:          if localState[hash] = empty() then initlocal(hash)
7:          deadFlag ← localState[hash]
8:          i += T
9:          if deadFlag ≠ 0 then continue
10:         globalState[hash].hits++
11:         globalState[hash].pSum += p
12:         if globalState[hash].hits = 24 then
13:             localState[hash]++
14:             events++
15:             if globalState[hash].pSum < 4 then
16:                 if b = true then truePositives++
17:                 else falsePositives++
18:             else if b = true then falseNegatives++
19:             else trueNegatives++
20:         else if globalState[hash].hits > 24 then
21:             localState[hash]++
22:     end while
```

IV. Experimental Setup

A. Program Execution

The EMU system we evaluated consists of 8 node cards, each containing 8 nodelets and a dual-core SC capable of performing higher level OS functions. When an application is run, a single thread is spawned on nodelet 0 which begins instruction execution as per the program's design. This thread then spawns a single thread on each nodelet. These threads then spawn any number of additional worker threads locally. The number of threads generated on each nodelet does not need to be the same, however only 64 threads may execute concurrently on any nodelet, with additional threads placed into an execution queue.

Threads migrate throughout the system as described in Sec. III until all datums have been exhausted. At this point they return control back to the original thread on $n0$, where statistical information is gathered and output. This final migrating thread then signals the SC that execution has completed.

B. Dataset Generation and Placement

Because of the lack of string handling instructions in the current prototype, the key, value, and flag fields for each datum generated are implemented as 8 byte integers. Space is allocated for $datumCount$ number of datums on every nodelet utilized. For our strong scaling tests we used 250 million datums, the maximum number that can fit on a single nodelet, and them spread uniformly across all nodelets used in the given run. While the stock maxium key range is 100,000 for a single generator we chose to set the key range to 6,400,000, or 100,000 per nodelet. This max unique key range was used for all runs and all nodelet counts tests to insure consistency throughout. Each nodelet received $250Mil/nc$ datums, where nc is nodelet count.

During the initialization phase nodelet 0 spawns nc number of threads locally on nodelet 0. Each thread is then given a distinct seed value which is used to generate randomized datum key values thereby ensuring that each thread does not generate identical keys. When generating random keys each thread inputs the same power law distribution so that all keys generated, though from disjoint seeds, still abide by the global power law distribution. As threads generate datums they are written to remote addresses corresponding to the nodelet on which the datums were assigned.

C. Scaling Tests

For our scaling tests we vary the number of nodelets from between 1 and 64 in powers of 2, as well as the number of threads spawned on each nodelet for datum processing again between 1 and 64 in powers of 2. For the reference case we utilize a single nodelet with a single thread processing all datums. The upper limit of our tests is 64 nodelets with 64 threads each, generating a total of 4096 concurrent threads.

Generation and placement of datums onto their associated nodelet occurs during the un-timed initialization phase. Run time measurements are started before the recursive spawn

which generates worker threads on each nodelet. A Cilk_sync prevents further program execution until all nodelets have completed, upon which the stop time is measured and total runtime determined. The time required by the asynchronous updates to statistic counters is included, as discussed in the benchmark specification.

To gain insight into the improvement from performing key filtering, we gathered information regarding the number of migrations that occur by making separate un-timed runs, with measurements taken that allowed counting migrations.

V. Evaluation

A. Strong Scaling Performance

Fig. 4 shows the multi-threaded speedup for Alg. 1 over a single thread. Each line represents a different nodelet count, while the x-axis represents the increase in threads spawned on each nodelet. The best performance is achieved by using 1 nodelet and increases as thread count increases until approximately 32 where performance begins to flatten. Overall as the number of nodelets increases the thread count at which performance stagnation occurs decreases with 64 nodelets flattening after 2 threads per nodelet.

The power law distribution used to generate datums is the primary reason for this behavior. As the number of nodelets increases the number of $globalState$ table entries that are local to a arbitrary nodelet decreases proportionally. Because of this as we increase system size we also increase the likelihood that a thread must migrate to a remote processing element in order to update a key's state. Greater migration counts means greater communication overhead and therefore reduced performance.

Additionally for Alg. 1 due to the power law distribution a small subset of keys will be "hit" with much greater frequency than others. As we increase the likelihood of migration as nodelet counts increase, we also increase the probability that threads will be inundating the nodelets which control the $globalState$ entries for these high occurrence keys. Since all $globalState$ entry updates are performed using atomic operations, threads must wait until they are able to perform their update. Thus as thread migrations intended to update the same memory locations pile up, their associated execution wait time also increases. In our tests the max occurrence of any key was 83,359,938 meaning that the number of thread migrations for this key was upwards of 65 million depending on nodelet count. Fig. 4 illustrates the impact excessive thread migration had on performance.

Fig. 5 displays overall speedup vs sequential (1 nodelet with 1 thread). Perfect speedup was observed for all thread counts as the number of nodelets used increased up to the point after which speedup flattens. Tests using 1 and 2 threads per nodelet experienced the longest period of perfect scaling up through 32 nodelets. Higher thread counts saw speedup stagnation much sooner, with 64 threads per nodelet flattening out after 2 nodelets. Despite this, the maximum speedup obtained was 110.1X using 64 nodelets and 64 threads per nodelet.

Throughput for Firehose is measured as the number of datums processed per second, and is the key comparison

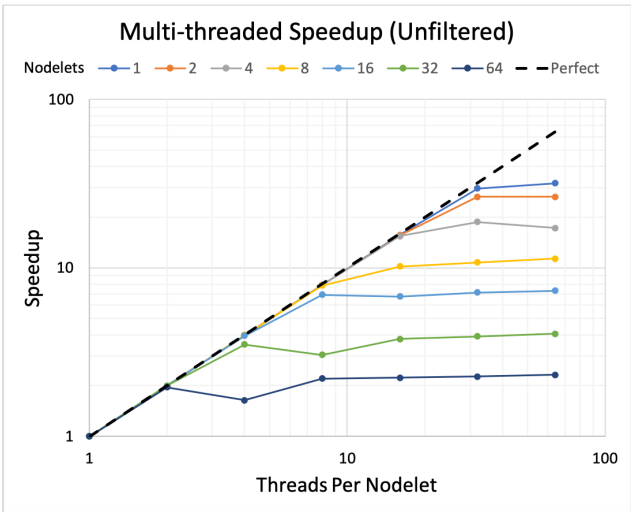

Fig. 4: Speedup for Alg 1 using 250M datums. Calculated as varying thread counts per nodelet vs a single thread per nodelet.

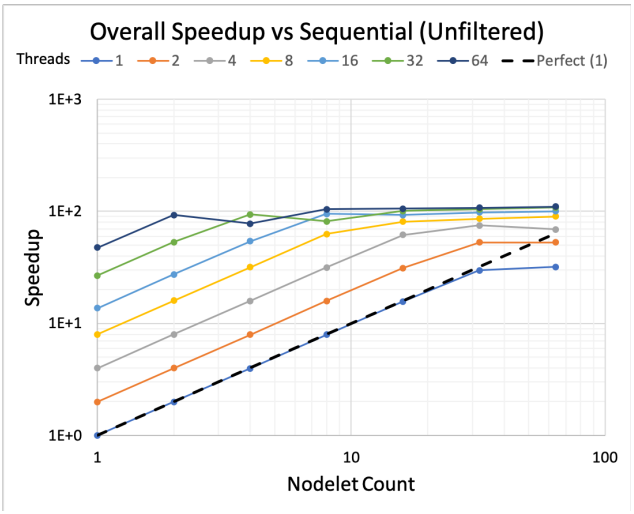

Fig. 5: Overall computational speedup compared to results observed for use of 1 nodelet with 1 thread for Alg 1 using 250M datums.

characteristic. Fig. 6 shows the throughput achieved by Alg. 1 as we scale strongly. While throughput for the 1 nodelet sequential tests is low at 102,100 datums per second the maximum achieved throughput was 11.2 million datums per second. Throughput behavior follows that of overall speedup seen depending on nodelet and thread counts despite the increased thread migration count as system size increases. As explained in Sec. V-B the number of migrations required does not vary enough as nodelet counts increase beyond $nc = 2$ to be a significant factor.

Alg. 2 was designed to eliminate meaningless migrations by checking a $localState$ table prior to performing any atomic operation for updating a key's state. Fig. 7 shows the multithreaded speedup for Alg. 2 with each nodelet count comparing runtimes to the single thread run times for the same nodelet count. Each line represents a different nodelet count, while the x-axis represents the increase in threads spawned on each nodelet. We observed that all nodelet counts experienced

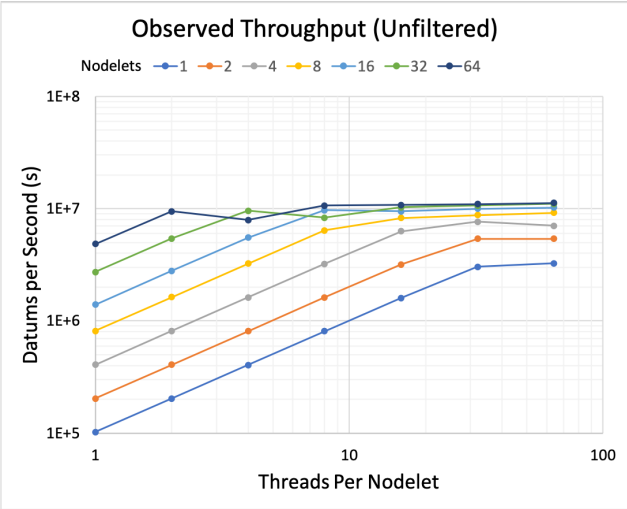

Fig. 6: Observed throughput for Alg 1 using 250M datums measured in datums per second.

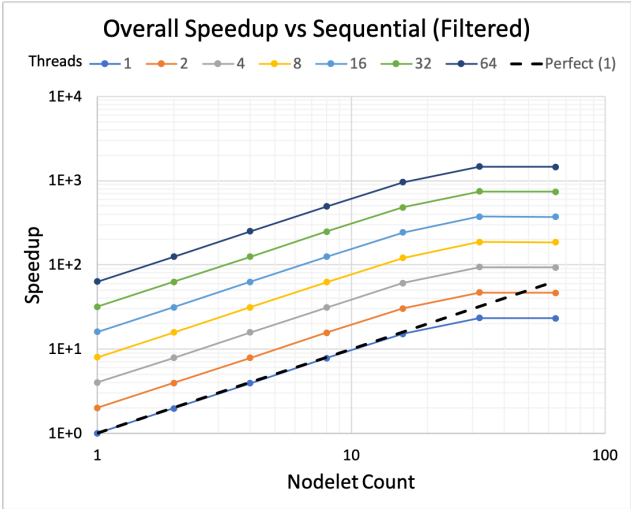

Fig. 8: Overall computational speedup compared to results observed for use of 1 nodelet with 1 thread for Alg 2 using 250M datums.

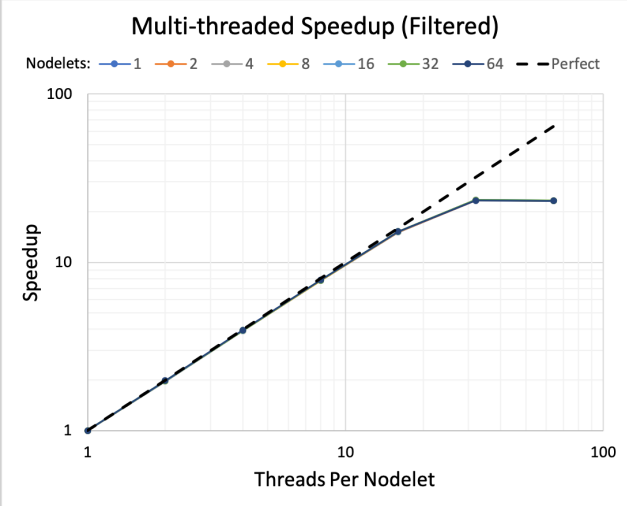

Fig. 7: Mutlithreaded speedup for Alg 2 using 250M datums is calculated as varying thread counts per nodelet vs a single thread per nodelet.

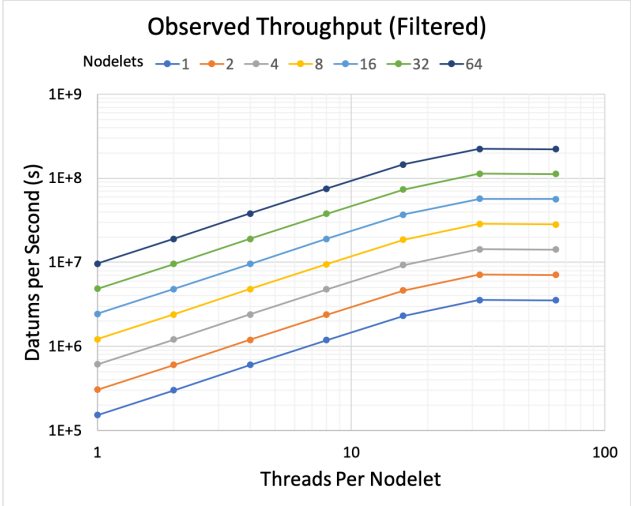

Fig. 9: Observed throughput for Alg 2 using 250M datums measured in datums per second.

nearly identical behavior as threads were increased.

Additionally, speedup is nearly perfect for all nodelet counts until approximately 16 threads per nodelet, continuing to increase though at a reduced rate through 32, and finally flattening at 64. The dramatic reduction in thread migrations is the primary factor causing this behavior. Since the number of migrations has been reduced so to has the the corresponding overhead. Along with this, once a key has been marked "dead" in the $localState$ table, no further ATOMIC_ADDM operations to update the global state will be conducted.

The overall scaling performance of Alg. 2 can be seen in Fig. 8. All thread counts increase near perfectly until nodelet count reaches 32 after which speedup flattens. Here 64 nodelets with 64 threads per nodelet obtained the maximum speedup over the sequential case of **1458.8X**!

Similar to overall speedup, Alg. 2 achieved superior throughput. Fig. 9 outlines the near perfect scaling of throughput as system size increases. Datums per second increases near perfectly for all nodelet counts, as threads per nodelet increases. As can be seen the rate at which through increases begins to decline at 16 threads per nodelet, followed by flattening after 32 threads per nodelet. Thanks to the increased performance of Alg. 2 the maximum throughput observed was **222.89 Million datums per second**, or approximately 20X that of Alg. 1.

Performance improvement due to the filtering performed in Alg. 2 compared to its non-filtered counterpart Alg. 1 is shown in Fig. 10. The performance for Alg. 2 was greater than that of Alg. 1 regardless of the number of nodelets or threads used during testing. Thread counts of 1 and 2 did begin to experience a downturn in comparative speedup after as nodelet counts increased beyond 16 but remained positive.

In general as the number of threads increased the level of comparative speedup obtained increased. **Maximum algorithmic improvement of 20.42X** was achieved using 32

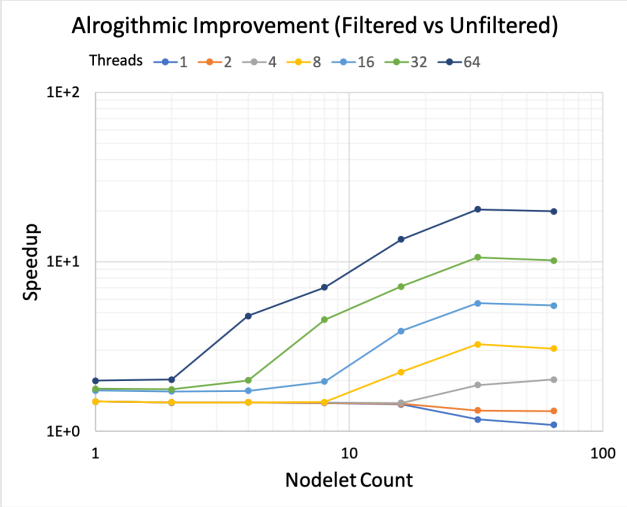

Fig. 10: Comparative algorithmic performance for Alg. 1 and 2

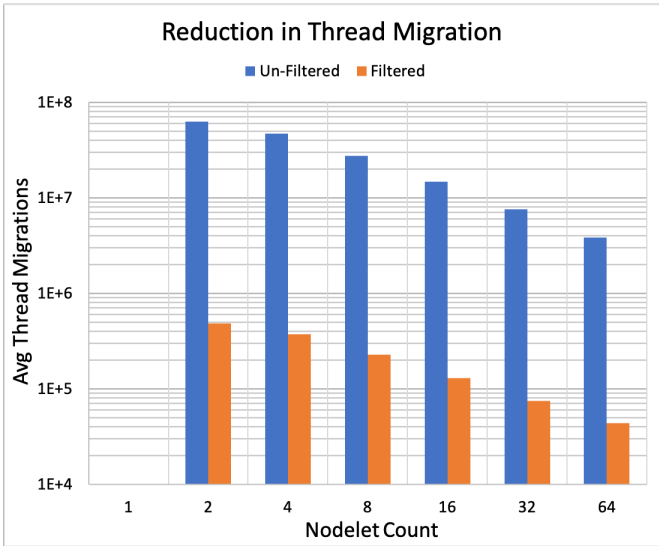

Fig. 11

nodelets with 64 threads per nodelet. This is likely due to the impact filtering has on performance as the probability of thread migration increases with system size. It is clear that eliminating migrations and atomic operations while they would otherwise be increasing in quantity allowed for significant throughput and overall speedup gains.

B. Migration Reduction

Fig. 11 shows the effectiveness of Alg. 2 at reducing the number of thread migrations. For 1 nodelet all hash table entries reside on nodelet 0, therefore no migrations take place regardless of algorithm. However for all other nodelet counts, some portion of datum evaluations will require thread migration. As shown in Fig. 11 Alg. 2 experienced an average of **3 orders of magnitude fewer thread migrations** than Alg. 1 for the same nodelet count. This pattern held true regardless of nodelet count tested.

VI. Conclusion

Streaming is of growing importance, and understanding what architectural features affect scalability (both positive and negative) is important. Firehose, while a small benchmark, is an excellent vehicle for focusing on the core of streaming applications.

The results of this experiment indicate that the migrating thread architecture has much better scalability than the previous reported data, and even significantly better performance (well over 100 million datums/s vs 5.6 for one core or 10 million datums/s for 7 cores), even given the lower computational performance present in the nodelet cores. There are of course caveats: this experiment pre-placed the data in nodelet memory and did not include the ASCII keys, but a fixed length binary version. The former was because the current prototype did not have sufficient real I/O capability, and the latter because of the limitations of the nodelet cores and the need to keep experimental run times down to a manageable level. Even so, the observed performance gains are significant. It is not hard to imagine that the performance would be at least similar even with the ASCII keys if the nodelet cores were replaced with realistic ASIC versions running 10X or faster than the current FPGA.

In looking at the implementation the gains seem to come from a variety of aspects of the migrating thread architecture. First is the lack of cache coherency traffic and the need for complex routines to perform atomic updates to the hash table, either locally or remotely. Second is the multi-threading that allows the memory channels of each nodelet to be fully utilized, regardless of the non-memory computations needed for each datum. Last but not least is the avoidance of explicit messaging software needed to communicate between physically separate nodes.

Near-term future work will focus on Variant 2 of the benchmark, where more complex interactions at the hash table are needed to weed-out entries that are too old. The observed performance drops in the website-reported data (roughly a factor of 2.5X) indicates there are even more issues present with conventional architectures. Additionally, modifications to the code can insert fake “writes” to simulate memory cycles needed in a real system to accept data from an input port. Also, performance runs will be made on the larger system under development that has a different and more even interconnect topology that the 8-node CRNCH system. Finally, as the core technology improves, going back to the ASCII keys will be included.

Acknowledgements

This work was supported in part by the Department of Energy, NNSA, under the Award No. DE-NA0002377 as part of the Predictive Science Academic Alliance Program II, 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, and for aiding in helping us the learning curve of using the available tool chains.

REFERENCES

[1] Firehose benchmarks. http://firehose.sandia.gov/.

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

[3] 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.

[4] 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*, SODA '02, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[5] 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*, KDD '08, pages 16–24, New York, NY, USA, 2008. ACM.

[6] 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*, VLDB '80, pages 285–300. VLDB Endowment, 1980.

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

[8] 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.

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

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

[11] D. Ediger, K. Jiang, J. Riedy, and D. Bader. Massive streaming data analytics: A case study with clustering coefficients. pages 1 – 8, 05 2010.

[12] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2):207–216, Dec. 2005.

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

[14] P. M. Kogge, N. Butcher, and B. Page. Introducing streaming into linear algebra-based sparse graph algorithms, July 2019.

[15] A. McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.

[16] S. J. Plimpton and T. Shead. Streaming data analytics via message passing with application to graph algorithms. *Journal of Parallel and Distributed Computing*, 74(8), 5 2014.

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