A Case for Migrating Execution for Irregular Applications

Peter M. Kogge
Univ. of Notre Dame
Notre Dame, IN
kogge@nd.edu

Shannon K. Kuntz
Emu Solutions, Inc.
South Bend, IN
skuntz@emutechnology.com

ABSTRACT

Modern supercomputers have millions of cores, each capable of executing one or more threads of program execution. In these computers the site of execution for program threads rarely, if ever, changes from the node in which they were born. This paper discusses the advantages that may accrue when thread states migrate freely from node to node, especially when migration is managed by hardware without requiring software intervention. Emphasis is on supporting the growing classes of algorithms where there is significant sparsity, irregularity, and lack of locality in the memory reference patterns. Evidence is drawn from reformulation of several kernels into a migrating thread context approximating that of an emerging architecture with such capabilities.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**; *Massively parallel algorithms*; • **Computer systems organization** → **Multicore architectures**; • **Applied computing**;

KEYWORDS

multi-threading, parallel systems, mobile threads, memory architectures, performance, PGAS

ACM Reference Format:
Peter M. Kogge and Shannon K. Kuntz. 2017. A Case for Migrating Execution for Irregular Applications. In *IA^3'17: 7th Workshop on Irregular Applications: Architectures and Algorithms, Nov. 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3149704.3149770

1 INTRODUCTION

A thread's "site of execution" is the core that contains the key state information needed for a thread to run, and that is capable of updating that state as the thread's program is executed. Today, the ability to move the site of execution of a thread is typically at best limited within a single node. This paper discusses the potential impact on program performance and complexity if that limitation is not only lifted, but augmented by hardware to make "migration" between execution sites largely "invisible" to the program.

IA^3'17, Nov. 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5136-2/17/11…$15.00
https://doi.org/10.1145/3149704.3149770

Such a capability will become more important as applications leave the dense regular world of LINPACK linear algebra and enter one where irregular accesses (both in time and location) become commonplace. In this regime, the effects on conventional cache-based architectures are severe. Similar problems surface as we enter the Internet of Things and demand real-time processing of floods of data. This paper will build evidence that substantial benefits are possible for both classes of problems if migrating thread capability is incorporated into the architecture.

The organization of this paper is as follows: Section 2 provides a definition of thread migration. Section 3 describes some systems that have hardware support for migration, including one where that capability is at the heart of the architecture, and which can scale to very large sizes. Section 4 describes a simple model for bounding performance when the majority of computation is done by migrating threads. Sections 5 and 6 look at two different kernels where reported information indicates that poor scalability is due to memory and execution model issues that may be largely avoided by migrating threads. Section 7 concludes.

2 MIGRATION

In conventional usage a *process* represents an execution stream for a program acting on related data. A *process state* is all the information needed to carry out that stream, including the code, data, call stacks, open files, mappings, etc. A *thread* is the sequential execution of one of perhaps many concurrent paths through a process. Each thread has a *thread state* with two parts - a part common to all threads in the same process, and a part relevant to just that thread. This part may have two pieces: a fixed size piece holding the working register values, and a variable size piece representing memory resources allocated to that thread, such as its call stack.

An *execution site* is the physical hardware where the state of a thread is modified to reflect and track the thread's execution. Such sites are usually a core within a microprocessor chip within a particular node that is within a particular rack. If a core is multithreaded then one core may hold multiple thread states.

The "migration" of a thread refers to a change in its execution site. Working register values are moved from one core to another, and any cached parts of a thread's variable state are made available to that core, with the values as modified by the thread just before the thread moved sites. For the migration to be valid, the process state must be visible to the thread at its new site.

In modern systems, thread migration to different sites happens all the time, but is confined in the range of possible execution sites. A thread may be executing on one core in a multi-core system and makes an I/O request. The thread's register state is saved until the I/O operation completes, and then is re-established in some other core. Any cached memory state, such as a call stack, must be made available at the new site through some coherency mechanism.

Migration via software goes back to the 1990s' when load balancing across nodes in a parallel cluster became important, especially for distributed shared memory systems ([9, 10, 18, 23]).

3 HARDWARE SUPPORTED MIGRATION

While not a central part of current mainstream architectures, there have been significant demonstrations of various forms of hardware-supported migration capabilities. The J-machine [11, 12] was a fine-grained parallel computer that had both hardware and instruction-level support for message-passing that could support remote spawning of functions, as in active messages. A message included operands for the new thread when it was spawned. The Cray T3D [20] and related follow-ons supported a form of a global address space where some atomic operations could be specified to execute against remote memory. DIVA [17] placed cores near each of multiple memory modules, and allowed active-message-like function calls to be spawned on the conventional host, and routed on the basis of memory address. PIM Lite [26] placed cores directly on a memory chip, and allowed explicit migration of threads between them.

A variety of proposals have been made (e.g. [25]) to hardware accelerate the process of saving, moving, and restoring thread states within a conventional multi-core architecture.

The EMU architecture [16] goes much further in integrating migration into all aspects of a scalable system architecture. As discussed in [21], thread states are explicitly separated from process state, and made very light. A single shared address space encompasses the memory in all nodes, with each node made up of multiple *nodelets*. Each nodelet contains a combination of memory and its own set of highly multi-threaded cores. The cores support the automatic migration of execution threads whenever the threads attempt to access a memory address that is not contained in the current nodelet. When a non-local address is detected, the core suspends the thread and sends its state to the nodelet holding the desired address. At this nodelet, the thread state is placed in any available core and restarted. Thus, memory references by a thread are always performed "locally." An extensive set of Atomic Memory Operations (AMO) can take advantage of this locality. In addition, a thread can spawn both other threads and write-based remote atomic memory operations (akin to very efficient special purpose migrating threads). Programming the system is done in Cilk [7]. Caching may be present, but is limited to individual memory channels (effectively upping the local access rate). Thus no coherency across nodelets is needed. Table 1 summarizes some expected design parameters, with the FPGA box configuration completing final test, and the others in design.

	FPGA-based		ASIC-based	
	Box	Rack	Box	Rack
Nodes in System	8	256	8	256
Nodelets per Node	8	8	8	8
Nodelets per System	64	2048	64	2048
Mem B/W per Nodelet	1.6	1.6	3.2	3.2
Cores per Nodelet	3	4	4	4
Core Clock (GHz)	0.3	0.3	2	2
Instr/s per Nodelet	0.9	1.2	8	8
Network Ports per Node	6	6	8	8
Injection B/W per Port	2.5	2.5	5	5
Injection B/W per Nndelet	1.875	1.875	5	5
Bandwidths in GB/s				

Table 1: Emu Design Parameters.

4 THE VISIT MODEL

To date, many applications designed for Emu exhibit three levels of threads. An original *root thread* is spawned at the start of a program execution, and visits all nodelets that will be involved in the computation. At each nodelet a *child thread* is spawned to initialize any local storage (such as local free pools and call stacks) and then spawn some number of third level *worker threads*. These threads perform most of the actual computation and, for the applications considered here, their execution path takes them through data structures spread out across the system.

There are typically many worker threads, and over an extended period most nodelets see a similar number of such visits with similar properties. A simple model can then be developed for estimating performance bounds in terms of maximum "visits per second." When a thread arrives, it stays for some period of time, making memory references and executing instructions. When a reference to a non-local location is encountered, the thread is migrated by the hardware without software intervention. A *visit* thus incorporates an average stay and an average migration.

It is often possible to estimate the number of memory accesses and the number of instructions an average stay performs. Dividing accesses into the maximum access rate possible for a nodelet provides an upper bound on the maximum number of stays that may be executed per second on one nodelet. A similar bound can be made with instructions and peak instruction rate.

It is also possible to estimate the size of the thread state that must migrate and thus the size of the packet that must be injected into a node's network ports. Given the peak injection bandwidth for a node into the system interconnect network, we can compute the average injection bandwidth per nodelet. Dividing the average injection bandwidth by the size of a thread migration packet gives the max number of migrations per second that may leave a nodelet.

Taking the minimum of all three bounds provides an upper bound on the max number of visits per second that a single nodelet can sustain. Multiplying by the number of nodelets gives a bound on the total number of visits supportable by a system per second.

This visits/s number is usually translatable into a more meaningful performance metric for the application in question. For example, Sec. 5 describes a sparse matrix problem where each visit encounters some number of non-zeros, each of which is involved in two floating point operations. Thus multiplying the aggregate visits per second by 2 times the number of non-zeros gives an effective floating point rate.

5 SPMV

The product $y = Ax$ of a sparse matrix A and a dense vector x (often termed **SpMV**) is a staple of many modern codes (see [14]). A key parameter is the total number of non-zeros in the array versus the array's total size. A related parameter is the average **row sparsity**

(termed "s" here), which equals the average number of non-zeros per row. HPCG (High Performance Conjugate Gradient)[15], for example, uses an SpMV at the core of a highly scalable parallel computation where each row has at most 27 non-zeros in it (an s of 27). Variations [19] support many graph algorithms.

For HPCG "floating point efficiency" (ratio of sustained flops over peak flops) is a key indicator of how well an architecture is suited for such problems. Today such numbers are at best a few percent, indicating that there is a serious problem with today's architectures handling such problems. This section explores what appears to be the underlying cause, and projects that migrating threads have the potential for a significant improvement.

5.1 Reported Data

Fig. 1 graphs some reported results[8]for a system with up to 8 nodes of dual-socket Intel X5650 6-core chips[1]. Each chip had an aggregate peak floating point potential of about 64GF/s, and three memory channels had an aggregate peak memory bandwidth of about 32GB/s. The matrices used in the study were taken from a large online collection[13], and had a variety of sparsities from about 7 to 398, with between 72,000 and 0.5M rows. A hybrid algorithm was used, where each 6-core socket supported a separate MPI process, with $P = p^2$ such processes arranged in a square array of p by p processes. The algorithm assumed that the dense n-vectors x and y were striped down the memory of the first column of processes. When more than one process is used, the stripe of x in each of the left column processes is sent to the diagonal process in that row, and then broadcast to each process in the corresponding column. Thus when the actual numeric computation begins, each process would have an n/p by n/p sub-array of A, and a matching stripe of x of length n/p. An OpenMP multi-threaded kernel then performs the local sparse matrix, dense vector product. The resulting stripe of n/p elements of y are then sent to the leftmost process in that process' row, and then atomically added into the y vector stripe residing there.

Fig. 1(a) graphs the reported sustained performance for a single process running on a single 2-socket node as a function of the number of threads (up to 2 per core), with separate lines for matrices with 4 different sparsities. The number in the key for each line is its "s" value. As can be seen, maximum performance occurs around 6 threads (notionally one thread for each core in the socket), is relatively flat thereafter, and is at best a few percent of the maximum floating point capabilities of the chip. This implies the capacity of some other resource has been reached, such as memory bandwidth. Further, the sparser the matrix, the lower the peak performance. From this data, the rest of the referenced study fixed each process at one socket, with each of the 6 cores running a single thread within the process.

Fig. 1(b) graphs sustained performance as a function of the number of processes (1, 2x2, and 4x4). As can be seen, scaling is relatively low, and in fact is negative for the sparser two cases. Fig. 1(c) presents the same data but normalized each curve to the performance achieved at 1 process, and adds a line representing "perfect strong scaling." The sparsest case loses almost an order of magnitude in going from 1 to 4 processes, and the second sparsest at best matches the performance of a single process by the time we get to 16 processes. Again this is an order of magnitude less than what perfect scaling would provide.

A common metric used in many rankings is the "floating point efficiency" (sustained flops over peak flops). Fig. 1(d) diagrams the floating point efficiency for these results. In contrast to LINPACK (dense linear algebra) where efficiencies in the 80s' to 90s' are typical, efficiency here deteriorates rapidly to considerably less than than 1% for the sparser cases. These sparser cases are even much worse than seen in HPCG[2] [15], where efficiency is between often 1 and 4%, and the equivalent sparsity per process remains relatively constant at about 27 regardless of the number of processes.

5.2 The Problem

Clearly there are problems with such a kernel, especially when sparsity is small. Fig. 1(a) shows something other than concurrency is bounding performance even for the single node multi-threaded case. Further, parallelizing the kernel to run on multiple processes makes the situation even worse. In this section we try to abstract out both problems so that they can be understood in general.

We first posit that the performance bounding in the single process case is memory bandwidth[3]. To validate this, we can compute backwards from the peak 6-thread performance at each s value to estimate the number of bytes fetched from memory per row computation as a function of sparsity and assuming that the memory was running at its maximum bandwidth. As can be seen in Fig. 1(e), an excellent fit is a linear relation of $19s + 100$ bytes per row. The constants in this model match almost exactly what we would expect if caches were ineffective (which we might expect in a sparse problem). For each non-zero in a row there are 3 memory references: the value of the non-zero, the column in which the non-zero resides, and then the matching value from x. If each is 8 bytes, and there are no cache hits, the total would be 24 bytes. If the column indecies are 4 bytes rather than 8, the byte count drops to an almost perfect match of 20 bytes. The 100 bytes per row from the curve fit is equivalent to about 12-13 8-byte accesses from memory to select and start each row computation, and then add the resulting sum back into memory. Again this seems quite reasonable.

The dip in performance when we start adding more processes has at least two likely causes. First, this is a strong scaling code, so unlike HPCG, the sparsity seen by each process decreases as the number of processes increase. For p^2 processes the sparsity seen by each process decreases to s/p. This increases the effect of the fixed 100 bytes per row startup costs.

Second, of probably bigger effect, is the communication needed by the algorithm at the beginning and at the end. An MPI process adds at least 6 memory references per word transferred: a read from the memory array as declared by the program, a write to an I/O buffer, a read from that buffer when it is ready for transmission, a write to the corresponding I/O buffer on the other side, a read from that buffer, and a write into the process's memory. This

[1] Data for a more advanced chip are also included, but is not as comprehensive as that reported for this system.

[2] http://www.hpcg-benchmark.org/

[3] [22] came to similar conclusions after performing an analysis of the bigger HPCG application that incorporates SpMV.

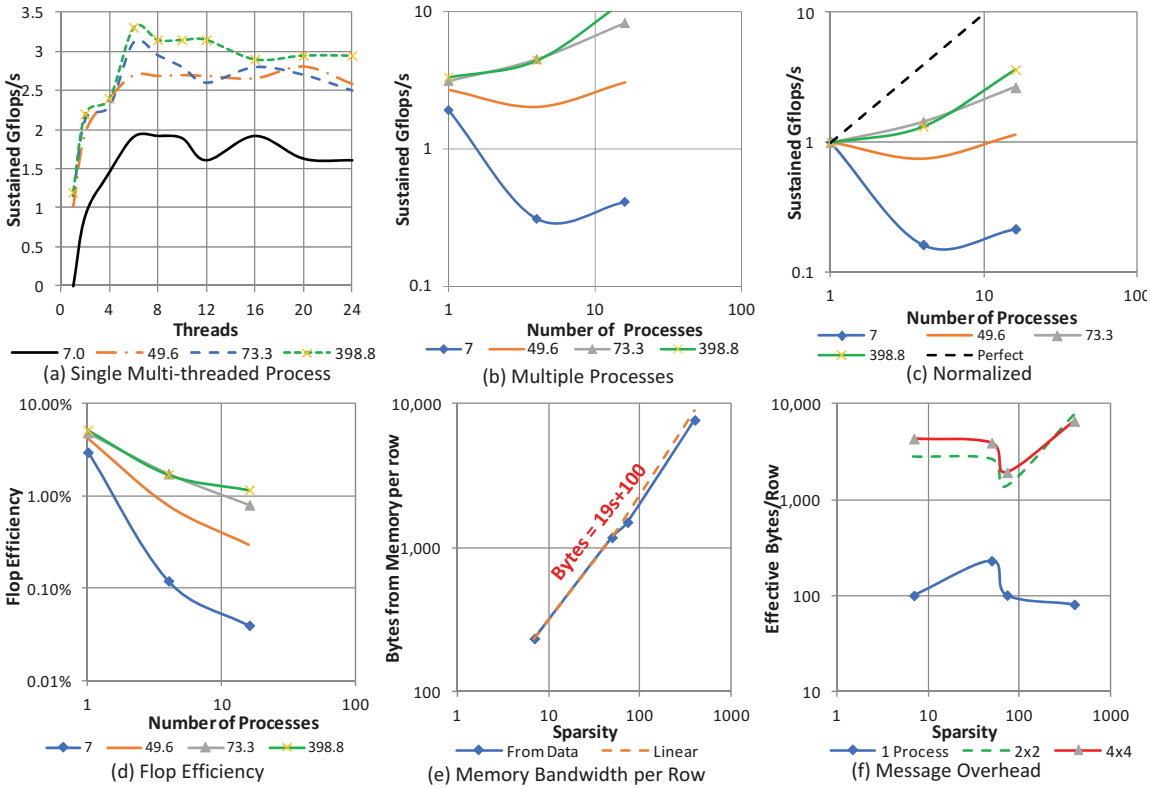

Figure 1: Reported SpMV Single Process Performance.

happens at least twice in the initialization time: from each of column 0 process to the matching diagonal process, and then up to each process in the column (perhaps more depending on the implementation of a broadcast). This is even worse on the cleanup, where each process transmits its stripe of y to its column 0 process, where an atomic add to memory must be performed. Also, any time that a thread in the program spends stalled for an MPI request to complete is equivalent to adding more "accesses" to the constant part of the model.

There are two complications to this effect. First is that the stripes of x and y that are read and written are probably largely in the same memory channel, which in this case has the effect of reducing effective bandwidth by a factor of 3. Further, the write-backs of the y strips all target the same process (column 0). This causes a hot-spot for incoming messages, which means that those transfers are driven by the network's bandwidth and the software stack to handle the MPI calls, not that of memory.

A more detailed analysis[4] of the MPI collective indicates that its implementation determines the bathtub shape of the curves. A simple implementation that transfers the entire segment of y from a process to its leftmost avoids the dip for small p, but at larger p dives without any recovery. Partitioning the y segments into smaller packets provides more performance, but eventually goes negative in the same way. A log-based tree reduction, on the other hand, reproduces the shape almost exactly. The early dip is due to the extra steps needed to combine segments two at a time, but once p gets large enough, the log number of steps, rather than linear in p helps and the performance turns around and climbs with larger ps'. The resulting curve, however, stays well under an order of magnitude of the perfect line.

Fig. 1 (f) assumes the same linear $ms + b$ model as before, but adjusts the s to s/p for p^2 processes (to account for the first effect), and derives backwards an equivalent resulting "constant" that would be added to get to the same measured performance. The $2x2$ and $4x4$ cases show a huge jump over that of a single process (factors of 20x to 80x), and correspond to the aggregate of the factors just discussed. This accounts for the significant dip in sustained gflops/s for low sparsities as parallelism increases.

5.3 SpMV via Migrating Threads

Migrating threads can help not only with both the above problems, but also in reducing the complexity of the code. Consider a straight-forward migrating thread code running on a machine with P nodelets (with all memory in a common address space). Assume that vectors x and y are striped equally across all nodelets so that $x[j]$ and $y[j]$ are in the same nodelet. Assume also that each non-zero $A[i, j]$ is represented as 3 words stored in the same nodelet as $x[j]$. Two of these are the same as in a CSR representation: the column number j and the value of the non-zero. The third is a pointer to the next non-zero triple in row i, which may be on a totally different nodelet. Assume also that there is a third vector co-located

[4]There is insufficient space in this paper for the details.

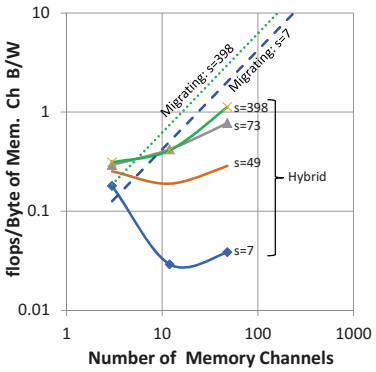

Figure 2: Migrating vs Hybrid.

with x and y where the i'th element is a pointer to the first non-zero in the i'th row of A. Thus each nodelet has the same number of rows from A, and equal length stripes of x and y (including all the y values that will be modified by processing the locally originated rows), and all elements of A that fall under the matching column indices of the local stripe of x values.

An algorithm for SpMV is now essentially the same as for the OpenMP version discussed above. A thread is given a row index, and proceeds down the row, performing each multiply-add for each non-zero, with the final sum written to $y[i]$ at the end. Assuming that the thread for row i originates in the nodelet holding $x[i]$, there are *at most* s+1 migrations in this process: one each to the next non-zero and one to write the sum into $y[i]$.

In terms of memory references, there are only two small additions to the basic OpenMP model described above. Instead of 3 memory reads per non-zero there are 4 (the extra pointer), and there may be an extra read at the beginning of the row to fetch the pointer to the first non-zero in the row. A reasonable upper bound for the number of bytes needed from memory for a row is thus $32s + 108$ bytes, assuming no helpful caching effects and regardless of where the non-zeros are and independent of P.

Significant optimizations are possible, such as by adding CSR-like edge blocks to handle all, and only, non-zeros that are in the same nodelet. The Stinger data structure [5] is an excellent match to such representations.

We note that the characteristics of the aggregate load on the network is better than for the hybrid algorithm. Since rows start and end at all nodelets, there is none of the hot-spotting described above. Further, a variety of tweaks could increase the effective utilization of the bandwidth by having say all the even rows by linked lower to higher and the odd rows linked higher to lower.

We can now compare implementations on the basis of how many sustainable flops are achievable *per byte of memory bandwidth from a single memory channel* (a single nodelet). Multiplying this ratio by the number of memory channels in a system, and then the bandwidth per memory channel gives the sustainable flop rate. Fig. 2 provides curves for both the hybrid algorithm discussed above and the migrating algorithm. For the hybrid case, each socket has three attached memory channels, so one process in Fig. 1 corresponds to 3 channels. Thus the hybrid numbers are computed by taking the original performance data and dividing by the number of channels (3) and the per channel bandwidth (10.66 GB/s). The migrating thread numbers are computed from the observation that 2s flops are computable for each 32s+108 bytes read from any memory channel.

The only place where the hybrid algorithm on the conventional architecture does better is for the 3 channel (single socket) case where the multiplier for the OpenMP case is 19 bytes per non-zero, vs 32 for the migrating thread. After that, as soon as the deleterious effects of inter-node parallelism come into play, the migrating case wins, and by up to 50$^+$X, especially for the sparsest cases.

6 FIREHOSE - A STREAMING APP

Firehose is a benchmark [2] designed to represent cyber-security applications that must process large streams of meta-data from network traffic in real-time, and detect possible anomalies. The input is a high-speed stream of *datums*, where each consist of an ASCII *key* (corresponding to a 64-bit IP address), an ASCII *value* (which depends on the benchmark variant), and a *truth value* (used only to verify that the implementation is correct). Processing consists of tracking the incoming stream for datums with the same key. An *event* is declared when some number (24) with identical keys are found. Further, detecting an event leads to a test if the event is an *anomaly* - when the set of values associated with the detected stream has some characteristics. Performance is measured by the peak *datums per second*[5] that the system can handle.

There are three variations of this benchmark. In *Anomaly 1* there are at most 100K unique keys and the values are either 0 or 1, taken from either an "unbiased" distribution (approximately equal numbers of 0's and 1's) or "biased" distribution (15 to 1 ratio of 0s and 1s). *Anomaly 2* is similar but there are an unlimited number of possible keys in the generated stream. *Anomaly 3* has a two-stage process involving an "outer" and an "inner" key.

Fig. 3 diagrams a generic processing flow for these benchmarks. Potentially multiple concurrent streams provide packets, each with multiple ASCII-encoded datums. After parsing of the ASCII string, a hash table is interrogated to see if the key has been detected before. If not, a new entry in the hash table is formed. If so, a count is updated on the number of observations and the number of times a value of 0 was found. When the observation count reaches 24, the value count is checked to see if the stream was unbiased or not. If the latter, an anomaly message is formed and sent.

6.1 Reported Data

Fig. 4 plots data from the website [2] using a dual-socket 6-core Intel X5690 3.46 GHz machine, with each socket supporting 3 memory channels and a peak memory bandwidth of 32GB/s. Depending on the implementation, some number of these cores are used for generation of the data streams, and the rest for the actual stream processing. Four different programming paradigms are reported: pure sequential Python and C++ implementations, use of the PHISH library [24] (a lightweight framework for streaming data via message-passing), and use of the Waterslide framework [3] for stream multi-threading, based on Sandia's Q-Threads [27] multi-threading library. The hollow markers in Fig. 4 represent data from the Anomaly 1 version of the problem; the filled markers represent data from

[5]The spelling "*datums*" comes from the documentation.

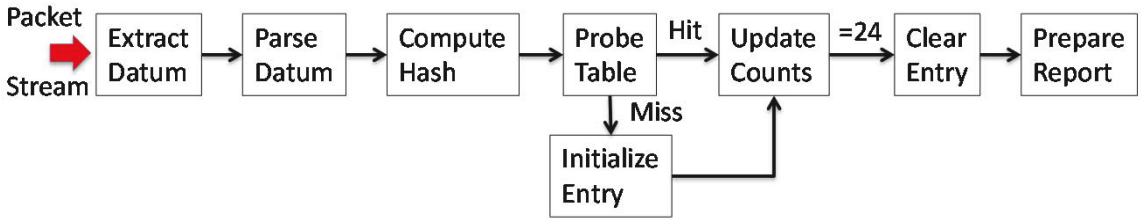

Figure 3: Firehose Processing Flow.

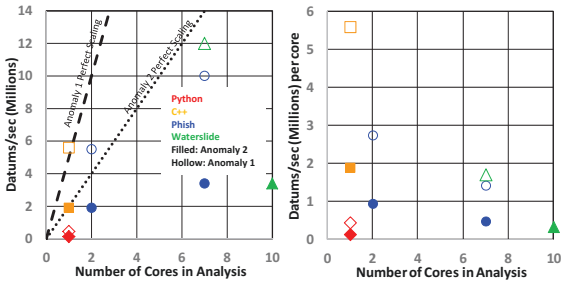

Figure 4: Firehose on Single Node.

the Anomaly 2 version. The dotted lines represent linear projections of performance if the best single core implementations (C++) could scale perfectly to multiple cores.

The Waterslide implementation in particular is at the surface a good match for processing such as shown in Fig. 3. The stream of incoming datums go through a series of functions in a pipelined fashion, with little direct interaction between individual inputs.

Berry [6] reports a bigger study of an MPI+PHISH implementation of Anomaly 1 on a Cray CS300 cluster system [1] with 1848 nodes. Each node has two 16-core sockets, each with 4 memory channels aggregating about 60 GB/s of bandwidth. Each node also has ports into a Dragonfly interconnection network.

The sockets in each node either generate simulated data to be processed, or perform the processing on that data. Sockets devoted to processing used 6 cores for parsing input traffic, 6 cores for key hashing, and 2 cores for anomaly detection. The largest reported run processed 1.1 billion datums per second with about 400 nodes. Scaling was fairly linear up to about 300 nodes handling about 975 million datums per second, for an average of about 3.25 million datums/s per node. At 80 nodes per rack, this is about 260 million datums/s per rack.

6.2 The Problem

Fig. 4(a) indicates that even within a single shared memory environment there are significant scaling issues for both versions of the benchmark. The sequential C++ code provides far more performance per core than any of the parallel forms, and by 7 cores even the best of the parallel paradigms (Waterslide) have maxed out at perhaps twice the performance of a single core. As the right curve shows, the effectiveness of parallelism dies rapidly as core counts increase, for both benchmark versions.

If in fact the bottleneck is memory bandwidth, and the 7 cores use all the memory bandwidth on the node, then for Anomaly 1 processing, each datum requires over 5,000 bytes accessed from memory, and over 18,000 for Anomaly 2.

The cluster results are in line with these conclusions. The use of 14 cores of a more modern chip actually provide only about 1/3 of the performance of the single node 7-core PHISH implementation. If all the memory bandwidth is consumed by analysis, the average bytes accessed per datum is about 18,000 bytes.

As with SpMV, there are probably several memory-related reasons for these issues: the addition of extra memory operations to handle the queueing and de-queueing of the transfer between nodes, and the need to introduce atomicity in updating data entries in the hash table when multiple datums descend on the same entry concurrently. Further, the key hash tables must be large enough to allow 128K+ different keys to be routed to separate bins with high probability. This probably causes significant caching problems, pushing most of the memory references out to the real memories, and requiring significant coherency traffic. Finally, it is likely that the hash table accesses use very little of the bandwidth returned by a typical 64 byte access to a modern DDR DIMM, meaning that memory bandwidth must be inflated significantly above that actually needed.

6.3 Firehose via Migrating Threads

Again as with SpMV, the potential use of a migrating thread for Firehose goes a long way to eliminating the above problems. A single thread could be written to implement one datum passing through Fig. 3 much as was done with the C++ implementation, with atomic operations added when necessary to guarantee correct updates to the hash table in the presence of multiple threads. No extra memory operations need be added to buffer data between processing stages: each thread carries the datum with it as it moves. No extra sequences to call and return from various procedure calls are needed: once launched, a thread can perform the entire processing of a single datum. Only when it can find no new datum packets need the thread return to some call stack to signal its termination and potentially wake a parent.

Further, having a large hash table spread out over a potentially large number of nodelets means that different threads representing different datums have good chances of executing on different nodelets, upping dramatically the potential for good scaling via decent load balancing.

To get a feel for what such memory traffic might entail, Fig. 5 summarizes a spreadsheet model for the processing of a single

24	Datums per report													
1.1	Ave. # of entries in hash table list													
0.05	Prob. of simultaneous free list access													
0.05	Prob. of concurrent inserts into same bucket													
Per Datum	1.000	1.000	1.050	1.155	1.000	0.044	0.044	0.044	0.002	0.042				
	Insert & Parse	Generate Hash	Probe bucket	Search per entry	Update on Match	Get free cell	Init new entry	Insert into bucket	Return entry on conflict	Report Generation	Total MemOps	% of All MemOps	Accesses	% of All Accesses
Reads	3	2	1	2	0	2	0	0	1	3	8.57	65.3%	8.57	59.4%
Writes	3	0	0	0	0	0	3	0	1	3	3.26	24.8%	3.26	22.6%
CAS	0	0	0	0	0	1	0	0	0	1	0.09	0.7%	0.17	1.2%
Atomics	0	0	0	0	1	0	0	1	1	4	1.21	9.2%	2.43	16.8%
MemOps	6.000	2.000	1.050	2.310	1.000	0.131	0.131	0.044	0.006	0.458	13.13	100%	14.43	100%
Accesses	6.000	2.000	1.050	2.310	2.000	0.175	0.131	0.088	0.008	0.667				
% of Total	41.6%	13.9%	7.3%	16.0%	13.9%	1.2%	0.9%	0.6%	0.1%	4.6%				

Figure 5: Memory Access Estimates for a Migrating Firehose Thread.

datum by a single migrating thread. The cells filled with yellow are numbers entered as estimates for each step, based on a hand-assembly of possible code. Each column in the table is a different stage in the processing pipeline. The four rows labelled "Reads", "Writes," "CAS" (Compare and Swap), and "Atomics" (other atomic memory operations such as atomic add to memory) represent different classes of memory operations. Assuming that CAS and Atomics take two memory accesses each (a read and a write), there are about 13.1 memory operations that translate into about 14.4 8-byte memory accesses needed per datum[6].

This table accounts in the "Insert and Parse" column for some outside I/O source having to write the ASCII form of the datum into a buffer and the thread then needing to read it out. Interestingly, these two operations account for over 40% of the memory operations per datum, suggesting that our supposition as to the cause of the slowdown in the conventional implementations due to multiple layers of such extra buffering is probably correct.

For a memory system where each access returns just 8 bytes (as in the systems of Table 1, this 14.4 accesses for an Anomaly 1 datum translates into about 115 bytes of bandwidth per datum (almost 1/50th of that from the conventional single node solutions).

Estimating code length in an architecture designed for migrating threads indicates a path length per datum of about 420 instructions, of which about 3/4 are for the ASCII conversion and hash key computations.

There is usually at least one migration: from the nodelet where the datum is found to the nodelet holding the appropriate hash table entry. Depending on how reports are accumulated, occasionally a thread may migrate to some nodelet that holds an output record queue (this is at most once every 24 datums, and usually a lot less). Another migration would be needed if the nodelet returned to its source node after completing a datum. This could be avoided by having the thread look for work at whatever nodelet it finds itself on after processing. However, such a move may end up introducing load imbalances, as certain nodelets may end up being abandoned by processing threads.

[6]Functions such as CAS and atomics take two accesses per operation

	Per Visit	Peak Visits/sec	
		FPGA	ASIC
Memory Accesses	14.3M	14	28M
Thread Size	128B	14.6M	29M
Instructions	420	2.1M	19M
Bound		2.1M	19M

Table 2: Bounding Firehose on a Single Nodelet.

A better solution is to use another aspect of a migrating thread architecture, namely the ability to spawn child threads cheaply. In this scenario, some number of threads may stay resident in each nodelet, with each simply fetching, parsing, and possibly hashing each datum, and spawning a child thread at that point. This child thread then migrates as necessary, and at its completion can simply die. In stream processing we don't really care about when the "last" thread is done, as there are no barriers before any next steps. Also, this means that we can keep approximately the same number of threads processing incoming datums per nodelet, and select that number to utilize most of the memory bandwidth if no other children are passing through.

Table 2 uses the visit model to provide an upper bound on the potential processing potential of a single nodelet. As can be seen, the bounding condition is the instruction processing rate, and 3/4 of this is related to string handling. It is not hard to believe that some better string handling sequences might double or more the visit limit for this condition, and still leave some memory and network head room. However, using a single node with 8 nodelets as a possible point of comparison to a node in the conventional systems (both have about the same number of memory channels), the takeaway is that even with this lower bound, 8 nodelets implemented

with FPGAs have the potential for an aggregate performance of 16 million datums/s, or about 5X that of a CS300 node. An ASIC implementation ups this ratio to about 40X on a node basis, or upwards of 149X on a rack to rack basis.

7 CONCLUSION

Hybrid parallel codes use one paradigm (OpenMP) for code running inside a single node, and an entirely different paradigm (message-passing) between nodes. This leads to complex codes, especially when patterns of data transfers between nodes are unpredictable in time or irregular in source-destination. It also affects performance by adding significant extra memory traffic, especially when the amount of processing needed on each node is small. The two kernels described here, SpMV and Firehose, are significantly different from each other but both suffer when scaling beyond a single node is attempted.

This paper discusses an alternative based on allowing threads to change their sites of execution invisibly within a large shared address space. While this isn't a universal panacea, in cases where where there is little locality, then as shown here substantial performance gains are possible, with simplified code. For SpMV, the code is simple row traversals, and gains over conventional codes may range as high as 50X, especially when matrices are extremely sparse and irregular. For streaming problems like Firehose, the migrating code paradigm again resembles considerably that of a simple serial code. Any "pipelining" in execution happens automatically as a thread touches different data structures (located on different nodelets) on the computational path.

A simple "visit" model permits performance bounding of many algorithms by counting the basic memory and instruction operations that need to be performed by a typical thread from the time it arrives on a nodelet until the time it migrates away. For many applications there are a large number of "worker" threads performing the same migration pattern, so such an approach is relatively accurate in projecting a potential performance throughput.

Real world proof of the validity of this new paradigm should be be available in the near future when actual codes are run on real hardware supporting the paradigm. In addition, designs of codes for more complex applications such as Breadth First Search [4] have been completed, and will also be tested in the near future.

Looking down the road, the migrating thread paradigm may also make an outstanding match for systems with significant near-memory processing. Such accelerators will need to be told when to start processing, and what functions to perform on what data. A migrating thread can visit a nodelet holding an accelerator, and spawn a child thread to run on the accelerator. Depending on the application, the parent thread could be suspended until the child completes, pick up the results, and then proceed to perform other functions elsewhere in memory, all without ever having to report back to some central node.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the Univ. of Notre Dame, in part by the National Science Foundation under Grant CCF-1642280, and in part by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002377, as part of the Center for Shock-Wave Processing of Advanced Reactive Materials, Univ. of Notre Dame. The hardware system was developed by Emu Solutions, Inc.

REFERENCES

[1] [n. d.]. Cray CS300-AC. http://www.cray.com/Assets/PDF/products/cs/CrayCS300-ACBrochure.pdf. ([n. d.]).
[2] [n. d.]. Firehose Benchmarks. http://firehose.sandia.gov/. ([n. d.]).
[3] [n. d.]. Waterslide Users' Guide. https://github.com/waterslideLTS/waterslide/blob/4c9bf85782a3d8179d03268d62a2da46955f2611/doc/users_guide/waterslide_users_guide.pdf. ([n. d.]).
[4] David Bader and et al. [n. d.]. The Graph 500 List. http://www.graph500.org/. ([n. d.]).
[5] David A Bader and et al. 2009. STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation. *Georgia Inst. of Tech.* (2009).
[6] Jonathan Berry and Alexandra Porter. 2016. Stateful Streaming in Distributed Memory Supercomputers. In *Chesapeake Large Scale Data Analytics Conf.*
[7] Robert D. Blumofe and et al. 1995. Cilk: an efficient multithreaded runtime system. In *Symp. PPOPP*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958
[8] B. Bylina and et al. 2014. Performance analysis of multicore and multinodal implementation of SpMV operation. In *Fed. Conf. on CS and Info. Systems*. 569–576. https://doi.org/10.15439/2014F313
[9] J. Casas and et al. 1994. Adaptive load migration systems for PVM. In *Supercomputing*. 390–399. https://doi.org/10.1109/SUPERC.1994.344302
[10] J. Chase and et al. 1989. The Amber System: Parallel Programming on a Network of Multiprocessors. In *12th ACM Symp. on Operating Systems Principles (SOSP '89)*. ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/74850.74865
[11] W. Dally and et al. 1987. Architecture of a Message-Driven Processor. In *ISCA*.
[12] William J. Dally and et al. 1998. Retrospective: The J-machine. In *25 Years of ISCA (ISCA '98)*. ACM, New York, NY, USA, 54–58. https://doi.org/10.1145/285930.285953
[13] Tim Davis. [n. d.]. The SuiteSparse Matrix Collection. https://www.cise.ufl.edu/research/sparse/matrices/. ([n. d.]).
[14] T. Davis. 2006. *Direct Methods for Sparse Linear Systems*. SIAM. https://doi.org/10.1137/1.9780898718881 arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9780898718881
[15] Jack Dongarra and Michael Heroux. 2013. *Toward a New Metric for Ranking High Performance Computing Systems*. Technical Report SAND2013 4744. Sandia National Labs. http://www.sandia.gov/~maherou/docs/HPCG-Benchmark.pdf
[16] T. Dysart and et al. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. *Innovative Architectures Workshop*.
[17] Mary Hall and et al. 1999. Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture. In *Supercomputing*.
[18] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. 1997. Thread Migration and its Applications in Distributed Shared Memory Systems. *J. of Systems and Software* 42 (1997), 71–87.
[19] J. Kepner and J. Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. SIAM. https://doi.org/10.1137/1.9780898719918 arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9780898719918
[20] R.E. Kessler and J.L. Schwarzmeier. 1993. Cray T3D: a new dimension for Cray Research. In *Compcon Spring '93*. 176–182. https://doi.org/10.1109/CMPCON.1993.289660
[21] P.M. Kogge. 2004. Of Piglets and Threadlets: Architectures for Self-Contained, Mobile, Memory Programming. *IWIA Workshop* (Jan. 2004), 130–138. https://doi.org/10.1109/IWIA.2004.10005
[22] Vladimir Marjanović, José Gracia, and Colin W Glass. 2014. Performance modeling of the HPCG benchmark. In *High Perf. Computing Systems*. Springer Int., 172–192.
[23] Edward Mascarenhas and Vernon Rego. 1996. Ariadne: Architecture of a Portable Threads system supporting Mobile Processes. (06 1996).
[24] Steven J. Plimpton and Tim Shead. 2014. Streaming data analytics via message passing with application to graph algorithms. *J. Parallel and Dist. Computing* 74, 8 (Aug. 2014), 2687–2698.
[25] M. Rodrigues, N. Roma, and P. Tomás. 2015. Fast and Scalable Thread Migration for Multi-core Architectures. In *Int. Conf. on Embedded and Ubiquitous Computing*. 9–16. https://doi.org/10.1109/EUC.2015.36
[26] Shyamkumar Thoziyoor, Jay Brockman, and Daniel Rinzler. 2005. PIM Lite: a multithreaded processor-in-memory prototype. In *Great Lakes Symp. on VLSI (GLSVLSI '05)*. ACM, New York, NY, USA, 64–69. https://doi.org/10.1145/1057661.1057678
[27] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. 2008. Qthreads: An api for programming with millions of lightweight threads. In *IPDPS*. IEEE, 1–8.