Crunching Large Graphs with Commodity Processors

Jacob Nelson†, Brandon Myers†, A. H. Hunter†, Preston Briggs†, Luis Ceze†, Carl Ebeling†,
Dan Grossman†, Simon Kahan†‡, Mark Oskin†

†University of Washington, ‡Pacific Northwest National Laboratory

{nelson, bdmyers, ahh, preston, luisceze, ebeling, djg, skahan, oskin}@cs.washington.edu

Abstract

Crunching large graphs is the basis of many emerging applications, such as social network analysis and bioinformatics. Graph analytics algorithms exhibit little locality and therefore present significant performance challenges. Hardware multithreading systems (e.g., Cray XMT) show that with enough concurrency, we can tolerate long latencies. Unfortunately, this solution is not available with commodity parts.

Our goal is to develop a latency-tolerant system built out of commodity parts and mostly in software. The proposed system includes a runtime that supports a large number of lightweight contexts, full-bit synchronization and a memory manager that provides a high-latency but high-bandwidth global shared memory. This paper lays out the vision for our system and justifies its feasibility with a performance analysis of the runtime for latency tolerance.

1. Introduction

Many important emerging applications such as social network analysis, bioinformatics, and sensor networks rely on crunching very large graphs. Unfortunately, the computational cost of these applications gets quickly out of hand. While tools to analyze social networks and query semantic graphs with billions of vertices and edges exist today [6, 10, 17, 18, 24], graphs of interest to defense applications are expected to have trillions of vertices and edges [15, 19]. Speeding up these applications at a low cost would have a significant impact in how we analyze large data-sets and make them even more valuable.

The most interesting computational challenge comes from large, low-diameter, power law graphs: this combination makes extracting performance difficult. They do not fit in a single commodity machine's memory. They are difficult to lay out with locality, since every vertex needs to be near every other vertex. They are difficult to partition in a balanced way [21, 22], leading to hotspots and load imbalance. Consequently, high-latency inter-node communication becomes an important limiting factor in scalability [35].

Multithreading is a technique that has been used successfully to implement efficient computations for these graphs [4]. The Cray XMT is an example of such an approach: it solves the memory latency problem through concurrency rather than caching. Each XMT processor supports 128 hardware contexts and 1024 outstanding memory operations, and is able to switch contexts every cycle [3, 13]. This ability comes at a cost: the XMT is an expensive, non-commodity machine with low single-thread performance.

We believe we can build a system based mostly on commodity parts that can attain XMT-like performance with a familiar, XMT-like, programming model, but at a fraction of the cost. Therefore, our goal is to build a system that has good performance on low-locality graph codes but is implemented using cheap commodity processors with the possible additional support of an FPGA. This approach has the added benefit of not sacrificing general-purpose performance.

Figure 1 shows an overview of our proposal. It is composed of multiple nodes built with commodity processors, communicating over an Infiniband network. We add two components: a runtime system, responsible for executing and managing user threads, and a global memory manager, responsible for facilitating memory requests to the global memory space shared across the nodes. We are exploring a mix of hardware (FPGA) and software implementations of the memory manager.

This paper describes the vision of our system and explores the feasibility of our ideas using a single-node implementation of a lightweight threading runtime. We use prefetch instructions together with lightweight threads to tolerate the latency of DRAM access on the local node. We use worst-case pointer chasing benchmarks to verify our runtime's overhead is acceptable. We also model the effects of pointer chasing with a remote node by artificially increasing latencies to several microseconds.

The rest of the paper is organized as follows. We briefly describe our programming model in section 2. We describe the implementation of our latency-tolerant runtime and plans for extending to multiple nodes in section 3. We evaluate our runtime in section 4. We present related work in section 5 and conclude in section 6.

2. Programming model

Our goal is to preserve the XMT programming model: large shared global address space, explicit concurrency with a large number of threads, and full-bit synchronization. We partition the overall address space into a global shared address space and per node private address spaces. Locality may be exploited directly by the programmer in the node private address spaces, just as in a conventional cache-coherent multiprocessor node. In contrast, locality cannot be exploited in the global space: its value is in providing high bandwidth random access to any word in a shared data structure by any processor.

The model promises efficiency subject to the presence of sufficient concurrency. Exactly how that concurrency is expressed by the programmer is language dependent. The goal

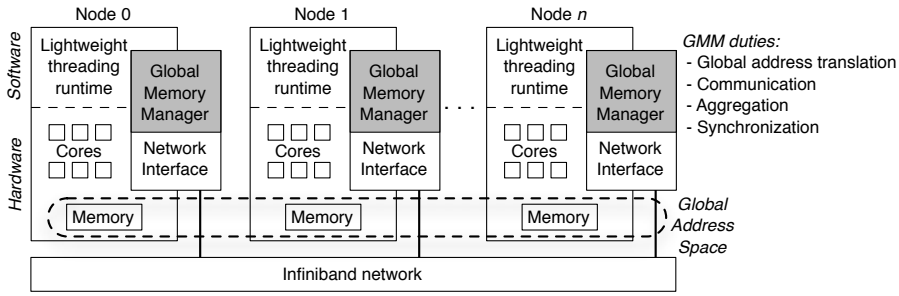

Figure 1: Overview of our proposed system design. The base system is a cluster of commodity nodes and interconnect. We add a latency-tolerant runtime and a global memory manager to obtain efficient access to the global address space. Only the shaded components may require hardware design.

of our system is to support that concurrency and consequently provide latency tolerance on a large address space. We do so via software multithreading: computation that is about to execute a long latency operation (e.g., a remote memory reference) initiates the operation and then yields to the scheduler, which quickly resumes execution of another computation. Concurrency beyond that required for latency tolerance can also be used to make more efficient use of network bandwidth: it enables aggregation of short memory requests into coarser requests, leading to large network messages and consequently better bandwidth utilization.

Synchronization on locations in the private address space of a node works the same way as on conventional systems. Synchronization on global addresses works differently: we provide atomic operations such as `int_fetch_add` as well as operations using full-bits, as on the Cray XMT. As is true for other long latency operations, synchronization latency is tolerated by yielding to the scheduler. Even non-blocking global atomics yield so the core can switch to another computation while the synchronization is accomplished outside of the pipeline. In addition, the system can also exploit aggregation when concurrency is abundant to increase the throughput of global synchronization.

3. Implementation

Our goal is to use multithreading to tolerate the latency of random accesses to memory in a global address space spread across multiple nodes. Our implementation must provide three features:

Concurrency in global memory references We must support many outstanding references to our long-latency global memory to fully utilize its bandwidth.

Lightweight multithreading We expect to need hundreds of threads to tolerate a cluster's network latency, so our threading implementation must have low overhead.

Low-overhead synchronization Long-latency synchronization operations should not block the processor's pipeline, so that other threads can proceed.

We discuss our approach to solving each of these challenges in turn. For each one, we discuss both our current single-node implementation and ideas for a multi-node implementation.

3.1 Concurrency in global memory references

To enable concurrency in memory references, we break each read or write into two operations: the *issue* operation and the *complete* operation. The read issue operation starts data moving towards the processor and returns immediately, like a prefetch. The read complete operation blocks until data is available. The write issue operation takes data along with an address, and starts an asynchronous write. The write complete operation blocks until the write is committed.

A latency-tolerant read operation in threaded user code then consists of three steps: a *read issue*, which causes the data to start moving; a *yield*, which causes another thread to start executing, overlapping the read latency; and, once the reading thread has been rescheduled, a *read complete* blocks until the data is available. A write operation follows the same pattern, blocking until the data has been committed.

In our single-node implementation, we use prefetch instructions for the issue operation, and regular blocking reads and writes for the complete operation. Writes use the prefetch instruction to gain cache line ownership before modification.

As we develop our multi-node implementation, we believe we will need hundreds of outstanding memory references per processor to cover the latency of remote references in a multi-node system. Unfortunately, commodity processors have much smaller limits on memory concurrency: section 4.1.1 finds a limit of 36 concurrent loads. We will have to manage memory concurrency on our own to bypass this limit. We describe approaches in both software and hardware.

One approach is for user code to delegate global references to a special *global memory manager thread*, running on another core in the chip. This thread translates the global address into a network destination, makes the request through the network card to the remote node, checks for completion, and delivers the returned data to the requesting thread. On the remote node, the remote memory manager thread performs the memory operation and return the data to the requesting node.

Another approach moves the global memory management to a hardware device. We can add a coprocessor FPGA in a processor socket, listening to coherence messages on the bus, similar to [27]. Note that this accelerator is only a memory proxy, not a compute coprocessor. The accelerator maps all global shared memory into a segment of each node's local

address space; when it detects a reference to memory on a remote node, it does the address-to-node translation and issues the request through the network controller, delivering the data to the requesting thread when ready. We encode commands in the upper bits of the address; a read from a location's *issue* address starts the remote request and returns immediately, and a read from a location's *complete* address blocks until the data is available.

With either approach, the question of request aggregation will be important. Each memory request we make is small—perhaps 8 bytes—but most networks are optimized for bulk transfers of a few thousand bytes. To improve network utilization, we will explore the aggregation of multiple memory operations to different addresses on the same remote node.

3.2 Lightweight multithreading

Our approach to supporting many lightweight threads uses a cooperative, user-level multitasking system built using coroutines. Much work has been done on similar user-level systems [5, 26], but we have more stringent requirements: coroutines must use little space so that many can be active without trashing the cache; and context switches must be fast (a small fraction of memory latency) so that we can overlap memory requests and achieve concurrency.

We treat context switches as function calls, as in [36]. This allows us to depend on the compiler to save and restore caller-save registers while we explicitly save and restore all the callee-save registers. We minimize context switch time by doing all switching and scheduling in user space.

In our single-node implementation, we implemented coroutines as described, with a round-robin scheduler.

As we develop our multi-node implementation, we will modify our scheduler to reactivate threads only after their long-latency operations have completed.

3.3 Low-overhead synchronization

Just as with reads and writes, we enable concurrency around synchronization operations by splitting them into *issue* and *complete* operations.

Implementing full-bit support efficiently on a platform not designed for them is a challenge. Previous work [34] has enabled support for full-bit synchronization on arbitrary words by allocating additional storage for the full-bits and implementing atomic full-bit operations as library routines.

One potential optimization is to limit full-bit synchronization to pointers to aligned data types, and reuse wasted space in the low-order bits of the pointer for full-bit storage. These bits would be masked out when the pointer is returned to the user. This allows us to synchronize any data type through one level of indirection.

In our single-node implementation, we prefetch and yield before performing a synchronization operation. We support only the synchronization operations supported by our development platform; we have not yet implemented full-bits.

In a multi-node implementation, synchronization operations can be delegated to a memory manager thread or accelerator. Synchronization on remote data would be performed by the remote memory manager. This may simplify the implementation: if only a single core (or single accelerator) is accessing the data being synchronized, the use of memory fences may be reduced or eliminated.

4. Evaluation

Our goal is to evaluate the feasibility of our proposed runtime. We want to determine two things: whether coroutines can generate memory concurrency while incurring minimal performance overhead, and whether the system can support the level of concurrency required to tolerate the latency that will be present in a multi-node system.

We focused the evaluation on one component of the runtime: lightweight context switching. We ran pointer-chasing benchmarks on a single-node implementation of our runtime. These pointer chasing experiments are intended to model a particular "worst-case" behavior of irregular applications, where each memory reference causes a cache miss.

We ran these experiments on a Dell PowerEdge R410 with two Xeon X5650 (Nehalem microarchitecture) chips and 24GB of RAM, with hyperthreading disabled. These chips use a NUMA memory architecture, where each chip has its own integrated memory controller and DIMMs; references to other chips' memory are carried over Intel's cache-coherent QuickPath Interconnect (QPI) [16].

Our evaluation consists of two parts. First, we demonstrate that the runtime system can achieve the same performance as when explicit memory concurrency is available. Second, we look at the runtime within the multi-node picture and show that it can tolerate large latencies. In each part, we begin by studying relevant aspects of our test machine's memory system so we can interpret our runtime results.

4.1 Single-node performance

4.1.1 Base memory system performance

We measured two parameters: the maximum random reference rate that can be issued by the cores in one chip and the maximum random reference rate that can be serviced by one chip's memory controller.

To find the maximum random reference rate that one chip's cores can issue, we ran pointer chasing code following the model of Figure 2a. Each core issues n list traversals in a loop; we call $n * \textit{number of cores}$ the number of *concurrent offered references* since the memory system may not be able to satisfy them all in parallel. Since our goal is to find a baseline for evaluating our coroutine library, we depend on the cores' exploitation of ILP for memory concurrency, rather than our coroutine library.

The lists were laid out randomly with pointers spaced at a cache line granularity, maximizing the probability of each reference being a miss. We allocated the lists on 1 GB huge pages to minimize overhead due to TLB refills.

```
while (count-- > 0) {
  list1 = list1->next;
  list2 = list2->next;
  ...
  listn = listn->next;
}
```
(a)

```
while (count-- > 0) {
   readIssue(&(list->next));
   yield();
   list = readComplete(&(list->next));
}
```
(b)

```
while (count-- > 0) {
  readIssue(&(remoteList->next));
  yield();
  remoteList = readComplete(&(remoteList->next));
  for( i in 1 to num_local_updates ) {
    localList->data++;
    localList = localList->next;
} }
```
(c)

Figure 2: Pseudocode for: (a) pointer chasing without coroutines, (b) pointer chasing using coroutines, (c) pointer chasing with local updates.

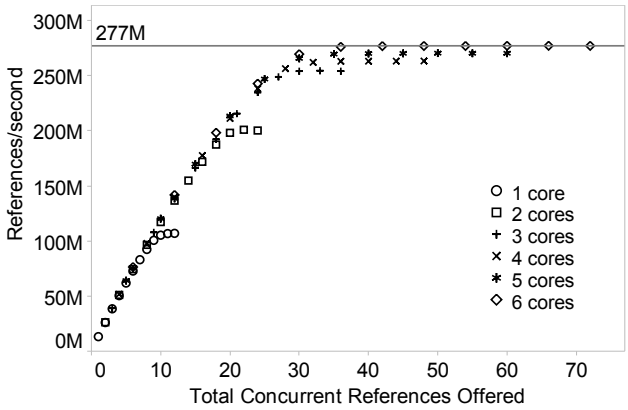

Figure 3: Pointer chasing throughput versus total number of concurrent references offered by the cores. Total concurrent references offered is *number of lists per core * number of cores*.

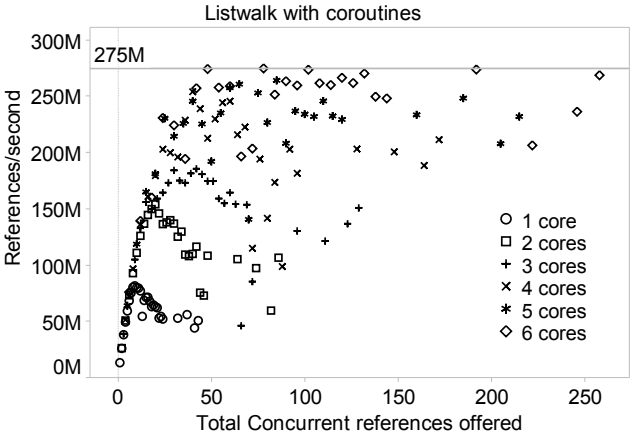

Figure 4: Pointer chasing with coroutines, with one reference per coroutine. Number of coroutines per core is *total concurrent references/number of cores*.

Figure 3 shows the result. Each point represents the maximum rate pointers are traversed for a given number of concurrent offered references. We see a maximum rate of 277 million references per second (Mref/s), which agrees with the measurements in [25]. This rate is achieved when the number of offered references is 36. Note that a single core cannot support this level of memory concurrency; the maximum reference rate for a single core is 107 Mref/s. We believe this limit is due to the core having only enough *line fill buffers* [32] to store 10 concurrent private cache misses.

The memory controller in the chip has more bandwidth than its cores can saturate. To measure the memory controller's maximum random reference rate, we extended the previous experiment so that cores in both chips traversed lists allocated in the first chip's memory. With this configuration, we observed a maximum rate of 360 Mref/s. We believe this difference is due to the chip having only 32 buffers in its *Global Queue* [23] for tracking concurrently-executing read misses from all the cores' private caches.

4.1.2 Coroutine performance

To evaluate the performance of our runtime, we investigated two effects: the maximum reference rate using coroutines to obtain memory concurrency and the effects of cache pressure from the coroutines' context storage.

To find the maximum reference rate obtainable using our coroutine library, we modified our pointer chasing benchmark as shown in Figure 2b. Recall that the baseline code in Figure 2a relied solely on ILP to achieve memory concurrency, which may not be abundant in real applications. Using coroutines, there are two sources of offered concurrent references: the ILP exploited by the processor, and the memory concurrency enabled by prefetching and switching to a new coroutine. As with our first experiment, we allocated the lists in the same chip as the cores doing the traversal.

Figure 4 shows the result. For this experiment, the only source of memory concurrency is the use of coroutines. We are able to obtain a rate of 275 Mref/s with 48 concurrent misses, or 8 coroutines per core. More concurrent requests are required to saturate memory bandwidth than in the ILP-only experiment.

We observe a gradual decrease in reference rate once the number of concurrent references per core exceeds 10; we believe this is due to later prefetches squashing earlier ones in the line fill buffers. In the multi-node runtime, `readIssue` will not be implemented with a prefetch instruction. We also observe repeatable data points that are off the trend and fall below the max. This effect is harder to explain, but we suspect it may be due to resource contention in the cores' pipelines once the memory system is full of requests.

The stacks for the coroutines are stored in the data cache, where they will compete for space with an application's data. To characterize the effects of this cache pressure, we modified our list chasing benchmark to include random updates to a per-coroutine local data structure that is small enough to fit in cache. Figure 2c shows the general idea.

We varied the size of the per-coroutine local working set from 64 B to 4 MB, and ran 0 to 32 local updates for each remote pointer traversal. We measured the remote reference rate (i.e., accesses to `remoteList` from Figure 2c).

Figure 5a shows the result for 8 coroutines per core. With 1, 2, and 4 updates per remote reference for smaller working sets, the runtime is able to achieve near maximum throughput.

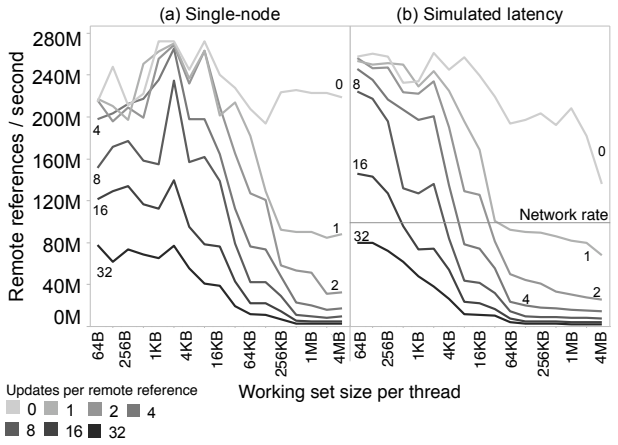

Figure 5: Cache pressure with coroutines, on six cores. The vertical axis is remote reference rate, with (a) 8 coroutines per core, and (b) 48 coroutines per core, about the number required for the network rate in the simulated delay experiments.

As the number of updates and the working set size increase, the data of other coroutines is likely to be evicted from cache, leading to decreased performance.

4.2 Performance with multiple nodes

To evaluate the potential for the lightweight contexts to perform in the multi-node case, we simulated a network delay to see whether the runtime could tolerate the larger latency.

Any network communication must travel over the QPI link in our test system. We estimated the bandwidth of this link by allocating lists in the first chip's memory and traversing those lists on the second chip's cores. We found that use of the QPI link limits throughput to 175 Mref/s.

To simulate the performance of pointer chasing in a multi-node system we referenced the first chip's memory from the second chip's cores and modified our coroutine scheduler to include a delay before a coroutine can be reactivated, imitating the network transit delay. We assume 1.1 μs interface and 400 ns switch latencies in each direction [20, 31], and thus estimate a 3 μs round-trip delay. In analyzing the results, we assume a network issue rate of 100 Mref/s, based on modern network interfaces [30].

The results of the experiment with six cores are shown in Figure 6. The runtime can still reach a maximum rate of 173 Mref/s,[1] saturating the QPI link. Little's Law predicts that 300 concurrent references are required to achieve the estimated network rate of 100 Mref/s when there is $3\mu s$ latency. The results agree: 100 Mref/s is reached with about 53 coroutines per core. Although there will be other overheads in a full runtime, this experiment suggests there is potential to tolerate multi-node system latencies.

[1] The throughput keeps increasing up to around 564 total concurrent references offered (at which point QPI is saturated), far more than the 36 we know the chip to support. This is because we have modeled the network latency by forcing coroutines to wait the extra time before using the data, so the physical miss buffers on our machine are not tied up any longer than usual.

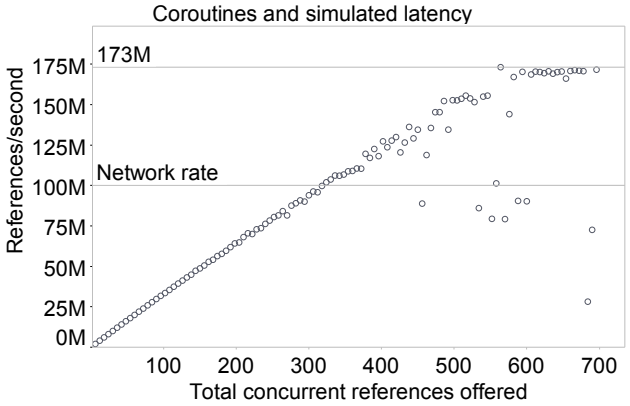

Figure 6: Pointer chasing across QPI link with simulated network delay of $3\mu s$, six cores. Using six cores, 53 coroutines per core are required to reach the assumed network rate of 100 Mref/s.

Figure 5b shows the results of the cache pressure experiment for supporting 48 coroutines per core, about the number of contexts needed to tolerate latency in the network experiment. We observe that when performing up to 16 local updates per remote reference, smaller working sets still allow the desired throughput. When performing 1 to 2 updates, the runtime can maintain throughput for working sets of up to 32 KB.

5. Related work

Much work exists on using multithreading to tolerate latency. Hardware implementations include the Tera MTA [3] and Cray XMT [13], Simultaneous Multithreading [33], MIT Alewife [1], Cyclops [2], and even GPUs [12]. Software-only approaches exist as well; the Software Controlled Multithreading project [28], QThreads [34], MAESTRO [29], and Charm [36] all use variants of this idea.

There has also been much work on user level threading, including the First-Class User Level Threading project [26] and Capriccio [5]. We have a different goal; we want many lightweight contexts to tolerate memory latency.

Our goal of presenting a global view of distributed memory to the programmer is shared by the PGAS community, and is used in languages like Chapel [7], X10 [8], and UPC [11]. They obtain performance by minimizing references to remote nodes, whereas we design for remote references.

The idea of processing large graphs on distributed machines has been explored by projects such as Pregel [24], the Parallel Boost Graph Library [14], and CGMLib [9]. Our goal is to develop infrastructure to aid implementation of similar libraries.

6. Conclusion

We presented our plans for building a system for large graph computation using commodity parts. We leverage decades-old ideas on using a large amount of concurrency to tolerate latencies, but we do so mostly in software. We developed a runtime system for latency tolerance, and our results showed that it supports enough concurrency to tolerate the long latencies of our large high-bandwidth global memory system.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife machine: architecture and performance. In *22nd Annual International Symposium on Computer Architecture*, 1995.

[2] George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Henry S. Warren, Jr. Dissecting Cyclops: a detailed analysis of a multithreaded architecture. *SIGARCH Computer Architecture News*, 31:26–38, March 2003.

[3] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[4] David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *International Conference on Parallel Procesing*, aug. 2006.

[5] Rob Von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *In Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281. ACM Press, 2003.

[6] Matthias Bröcheler, Andrea Pugliese, and V.S. Subrahmanian. COSI: Cloud oriented subgraph identification in massive social networks. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, aug. 2010.

[7] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applcations*, 21:291–312, August 2007.

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[9] Frank Dehne, Alfonso Ferreira, Edson Cáceres, Siang W. Song, and Alessandro Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, 33:183–200, 2002. 10.1007/s00453-001-0109-4.

[10] David Ediger, Karl Jiang, Jason Riedy, and David A. Bader. Massive streaming data analytics: A case study with clustering coefficients. In *2010 IEEE International Symposium on Parallel Distributed Processing*, april 2010.

[11] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

[12] Kayvon Fatahalian and Mike Houston. A closer look at GPUs. *Communications of the ACM*, 51:50–57, October 2008.

[13] John Feo, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[14] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC*, 2005.

[15] Timothy D. R. Hartley, Umit Catalyurek, Füsun Özgüner, Andy Yoo, Scott Kohn, and Keith Henderson. MSSG: A framework for massive-scale semantic graphs. In *2006 IEEE International Conference on Cluster Computing*, sept. 2006.

[16] Intel. An introduction to the Intel QuickPath Interconnect, January 2009. `http://www.intel.com/technology`.

[17] Cliff Joslyn, Bob Adolf, Sinan al Saffar, Jon Feo, Eric Goodman, David Haglin, Greg Mackey, and David Mizell. High performance semantic factoring of giga-scale semantic graph databases. In *Semantic Web Challenge*, 2010.

[18] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *Ninth IEEE International Conference on Data Mining*, December 2009.

[19] Tamara Kolda, David Brown, James Corones, Terence Critchlow, Tina Eliassi-Rad, Lise Getoor, Bruce Hendrickson, Vipin Kumar, Diane Lambert, Celeste Matarazzo, Kevin McCurley, Michael Merrill, Nagiza Samatova, Douglas Speck, Ramakrishnan Srikant, Jim Thomas, Michael Wertheimer, , and Pak Chung Wong. Data sciences technology for homeland security information management and knowledge discovery. Technical Report UCRL-TR-208926, Lawrence Livermore National Laboratory, 2004.

[20] Matthew J. Koop, Wei Huang, Karthik Gopalakrishnan, and Dhabaleswar K. Panda. Performance analysis and evaluation of PCIe 2.0 and quad-data rate Infiniband. *Symposium on High-Performance Interconnects*, 0:85–92, 2008.

[21] Kevin J. Lang. Fixing two weaknesses of the spectral method. In *Advances in Neural Information Processing Systems*, pages 715–722, 2005.

[22] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6, 2009.

[23] David Levinthal. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors. Technical report, 2009. `http://software.intel.com/sites`.

[24] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[25] Anirban Mandal, Rob Fowler, and Allan Porterfield. Modeling memory concurrency for multi-socket multi-core systems. *Performance Analysis of Systems Software (ISPASS)*, March 2010.

[26] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 110–121, New York, NY, USA, 1991. ACM.

[27] Jace Mogill. Extended memory sematics on hybrid FPGA-x86 architectures. Unpublished.

[28] Todd C. Mowry and Sherwyn R. Ramkissoon. Software-controlled multithreading using informing memory operations. In *Sixth International Symposium on High-Performance Computer Architecture*, 2000.

[29] Allan Porterfield, Nassib Nassar, and Rob Fowler. Multi-threaded library for many-core systems. In *IEEE International Symposium on Parallel Distributed Processing*, May 2009.

[30] Mellanox Technologies. Mellanox Infiniband delivers world record message rate performance for high-performance applications, June 2010. Press Release.

[31] Mellanox Technologies. Infiniband switch systems, 2011. `http://www.mellanox.com`.

[32] Michael E. Thomadakis. The architecture of the Nehalem processor and Nehalem-EP smp platforms. Technical report, December 2010. `http://sc.tamu.edu/systems/eos/nehalem.pdf`.

[33] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[34] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[35] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, Washington, DC, USA, 2005. IEEE Computer Society.

[36] Gengbin Zheng, Laxmikant V. Kale, and Orion Sky Lawlor. Multiple flows of control in migratable parallel programs. *Proceedings of the 8th International Conference on Parallel Processing Workshops*, 0:435–444, 2006.