Towards Virtual Shared Memory for Non-Cache-Coherent Multicore Systems

Bharath Ramesh, Calvin J. Ribbens
Center for High-End Computing Systems
Department of Computer Science, Virginia Tech, Blacksburg, VA 24061
Email: {bramesh,ribbens}@cs.vt.edu

Srinidhi Varadarajan
Dell Inc.
Email: Srinidhi_Varadarajan@dell.com

***Abstract*—Emerging heterogeneous architectures do not necessarily provide cache-coherent shared memory across all components of the system. Although there are good reasons for this architectural decision, it does provide programmers with a challenge. Several programming models and approaches are currently available, including explicit data movement for offloading computation to coprocessors, and treating coprocessors as distributed memory machines by using message passing. This paper examines the potential of distributed shared memory (DSM) for addressing this programming challenge. We discuss how our recently proposed DSM system and its memory consistency model maps to the heterogeneous node context, and present experimental results that highlight the advantages and challenges of this approach.**

***Keywords*-virtual shared memory; cache coherence; memory consistency models; multicore systems**

I. Introduction

Shared memory thread-based programming (e.g., Pthreads, OpenMP) has been the popular programming approach for intra-node parallelism. However, emerging node architectures are both highly parallel and increasingly heterogeneous. Furthermore, these emerging heterogeneous architectures do not present the programmer with a cache-coherent view of shared memory at the level of the individual node. It is not clear what programming model is best for these node platforms, which feature multiple traditional processors along with accelerators or coprocessors. One response to the lack of node-wide shared memory is to use the message passing model within a node, essentially viewing each node as a cluster-on-a-chip. Unfortunately, this means giving up the advantage of the shared memory that typically is available within some components of the node. Furthermore, the amount of memory per core in coprocessors is typically low. So treating the cores of the coprocessor as nodes of a mini-cluster, and not fully using the much larger memory associated with the host or general purpose CPU, severely limits the size of problems that can be solved. Another programming option is to completely offload computation to the accelerator or coprocessor. This allows the code running on the coprocessor to take advantage of the programming model that works best for that coprocessor (e.g., thread-based), but it still requires the programmer to manage moving data to and from the coprocessor.

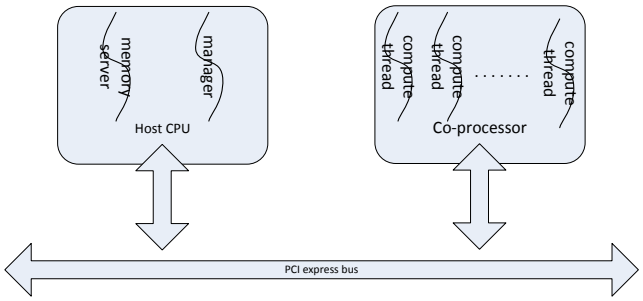

Figure 1. Architectural view of Samhita system consisting of memory servers and managers executing on the host processor, which provide a consistent shared address space for compute threads executing on accelerators or coprocessors connected over the PCI express bus.

Many have noted that the shared memory programming model is easier for most programmers, compared to explicit message passing in MPI [1]. Furthermore, with the rapid adoption of multicore and many-core architectures at all levels of computing—from mobile devices to laptops and high-end servers—there is is a large and growing ecosystem of shared memory programmers. Thus, we have two opposing trends. On the one hand, programming convenience and the existence of some shared memory argues for a shared memory programming model across the entire heterogeneous node. On the other hand, increasingly parallel and heterogeneous nodes that lack cache-coherent shared memory are pushing us towards the message passing programming model.

This is not the first time the computing community has faced these kinds of conflicting priorities and trends. Almost since distributed memory clusters first appeared, researchers have explored ways to provide virtual shared memory, or distributed shared memory (DSM), over such systems. The motivation and challenge are very similar to the present case: programming ease prefers one model, but peak performance (by some measure) prefers another. The goal of DSM systems is to provide shared memory semantics over physically distributed memories. Although the DSM systems proposed 10 or 20 years ago never made a big impact (primarily due to relatively slow interconnects), it is worth considering whether the insights and approaches developed in that work can play a role in our current programming challenge. In

particular, by treating the processing cores of accelerators or coprocessors as individual cores of a cluster, can we provide effective virtual shared memory for these heterogeneous nodes? We have recently been revisiting DSM for clusters in the light of trends in high-performance interconnects [2]. In this paper we consider how the architecture of Samhita, our DSM prototype, maps to today's heterogeneous highly parallel multicore systems. Our main focus is on systems such as the Intel Xeon Phi (Many Integrated Core) [3] family.

Samhita provides virtual shared memory by separating the concept of serving memory from that of consuming memory for computation. At the highest level, Samhita's architecture consists of memory server(s) and compute servers. This maps well to a typical heterogeneous architecture (see Figure 1), which has one or more general purpose host processors associated with a large memory, and one or more multicore coprocessors, all connected by a high-speed bus. The host processors can execute the memory servers, using their memory as the backing store for the global virtual address space. Compute threads can run on the cores of the coprocessor, with the coprocessor's memory used for caching copies of the global address space. Our current Samhita implementation supports the Infiniband switched fabric as the high-performance interconnect. In the scenario of a heterogeneous node, the PCI Express bus connects the host processor to the accelerator or coprocessors.

A critical component of any DSM is the memory consistency model presented to the programmer. Shared-memory programmers are used to the extremely strong consistency provided by true shared-memory hardware. For performance reasons, DSM's typically weaken the consistency model in some way. We have proposed *regional consistency* (RegC) [4], a new memory consistency model that is nearly as strong as hardware shared memory, but allows for performant implementations on physically distributed memory systems. RegC allows existing threaded codes to run on distributed memory clusters with only trivial code modifications.

The goal of this paper is to consider the possibility of using a DSM such as Samhita as a design starting point for providing shared memory semantics across the components of a heterogeneous platform. We do this by first showing how Samhita's architecture and RegC maps to the heterogeneous host/coprocessor context, and then with a series of experiments that highlight the performance issues and tradeoffs that need to be addressed. These experiments use our current implementation of Samhita, with multiple nodes of an Infiniband cluster playing the roles of the components of a single accelerated node. In other words, rather than having a single PCI Express bus between our host and coprocessor(s), we have the Infiniband fabric between cluster nodes (which means Infiniband cards and links, a switch, and a PCI Express bus on each side of every communication). So in one sense, our experiments are quite pessimistic, e.g., an implementation of Samhita directly onto the Intel Xeon Phi (currently being investigated) could reduce the communication costs substantially.

The rest of the paper is organized as follows. We present an overview of Samhita and the RegC model in Section II. We then present previously unpublished performance results for Samhita/RegC in Section III. Related work is presented in Section IV. We conclude and discuss future work in Section V.

II. Overview of Samhita and Regional Consistency Model

Samhita views the problem of providing a shared global address space as a cache management problem. This is done by separating the notion of serving memory and consuming memory for computation. Samhita is architected with compute servers, memory servers, and a manager. The memory servers are responsible for serving the memory required for the shared global address space. The manager is responsible for memory allocation, synchronization and thread placement. The compute servers are where the individual compute threads execute. In our current implementation of Samhita, which targets clusters, each of the components runs on individual cluster nodes (although multiple components can run on the same node as well). In the case of a heterogeneous system, the manager and memory server run on the host processor while the compute threads execute on the accelerators or coprocessors.

Samhita divides the shared global address space into pages. Each compute thread[1] has a local software cache through which it accesses the shared global address space. Samhita uses demand paging to populate the local caches. To reduce the number of misses for applications that exhibit spatial locality, we use cache lines of multiple pages. We also use anticipatory paging or prefetching to exploit spatial locality. The prefetching strategy used by Samhita is simple. When a cache miss is detected Samhita places a request for the missing cache line and an asynchronous request for the adjacent cache line. Since all operations in Samhita occur at the granularity of a page, the impact of false sharing can be significant. To reduce the impact of false sharing, Samhita supports a *multiple-writer* protocol. If a cache becomes full the eviction policy used is biased towards pages that have been written to.

Samhita is implemented as a user-level system requiring no modifications to the operating system kernel. It depends only on standard system libraries. To reduce communication overhead, Samhita interfaces directly with the interconnection network. To support multiple interconnection networks the communication protocol is abstracted as the Samhita

[1]Each Samhita compute thread is actually a process, but we use the term thread throughout.

Communication Layer (SCL). The current implementation of Samhita supports Infiniband switched fabric using the OpenFabrics Alliance [5] verbs API. SCL presents Samhita with a direct memory access communication model instead of a serial protocol. This enables SCL to map easily to the RDMA model of Infiniband.

Samhita keeps the shared global address space consistent using *regional consistency* (RegC) [4]. The RegC consistency model has two primary goals: to enable existing shared memory code to be easily ported to Samhita with trivial code modification, and to allow for a performant implementation of the consistency model. To achieve these goals, RegC explicitly distinguishes between modifications (stores) to memory protected by synchronization primitives and those that are not. This allows implementations to use different update mechanisms to propagate changes made to memory protected by synchronization primitives and those that are not. The details of the model are beyond the scope of this paper.

The idea behind the regional consistency model is to divide an application's memory accesses into two types of regions—*consistency regions* and *ordinary regions*. Memory accesses made in a consistency region (e.g., critical sections) are protected by mutual exclusion variables. Any memory access made outside of consistency regions occur in an ordinary region. Our current implementation of RegC in Samhita uses page granularity for updates made in ordinary regions and fine grain (data object level) updates for memory modifications made in consistency regions. To enable fine grain updates we need to track modifications made by individual stores performed by the application in a consistency region. To do this we use the LLVM compiler framework [6]. We instrument the application by inserting a function call before any store performed in a consistency region. We use static analysis of the application code to determine which stores are performed in a consistency region. Our experimental evaluation has shown that the overhead of this store instrumentation technique is negligible for most applications.

Samhita presents the programmer with APIs for memory allocation, synchronization and thread creation. These APIs are very similar to that presented by Pthreads [7] making it trivial to port existing threaded code to run on a cluster using Samhita. Samhita offers mutual exclusion locks, condition variable signaling and barrier synchronization as synchronization primitives. Samhita performs memory consistency operations whenever the application performs a synchronization operation to ensure that the shared global address space is consistent. This increases the cost associated with synchronization for Samhita. The cost of synchronization is directly affected by the amount of data that is moved during a synchronization operation. This amount depends on the level of false sharing amongst the computation threads.

The level of false sharing is directly related to how memory is allocated by the runtime system. Samhita uses three strategies to allocate memory to reduce both false sharing and the cost to allocate memory. The strategies differ based on the size of the allocation request. The first strategy is used for small allocations. It allocates memory from arenas that are associated with each thread. This allocation is handled locally by the thread, hence removing the cost associated with communicating with the manager for these allocations. This also reduces the amount of false sharing amongst computation threads when a large number of small allocations are performed. When the size of the allocation request crosses a configurable threshold, the second allocation strategy is used. The allocator contacts the manager for the allocation, which then allocates it from a shared zone. For large allocations the third strategy is used—the Samhita allocator directly strides the allocation request across multiple memory servers for reducing hot spots.

```
for (i = 0; i < N; ++i) {
    sum = 0;
    for (j = 0; j < M; ++j) {
        for (k = 0; k < S; ++k) {
            rsum = 0;
            for (l = 0; l < B; ++l) {
                *am(k,l) = r * (*am(k,l));
                rsum += *am(k,l);
            }
            sum += M_PI * rsum;
        }
    }
    LOCK(lock);
    gsum += sum;
    UNLOCK(lock);
    BARRIER_WAIT(barrier);
}
```

Figure 2. Computational kernel of the micro-benchmark code used in the experiments, reflecting local allocation. For global allocation, array indices are a function of thread id. For global strided allocation, the k loop take strides of num_threads.

III. Performance evaluation

To evaluate the performance of Samhita and the regional consistency model we are primarily interested in two important components that contribute to the runtime of an application—compute time and synchronization time. We present the results from micro-benchmarks and two application kernels, Jacobi and molecular dynamics based on the codes from the OmpSCR [8]. The performance evaluation has been carried out on up to six nodes interconnected using quad data rated (QDR) Infiniband switched fabric. Each node is a dual quad-core 2.8GHz Intel Xeon (Penryn Harpertown) with 8GB of main memory. All experiments are conducted using one node acting as a memory server and one as the manager.

The micro-benchmark (see Figure 2) allocates a fixed amount of data (S rows of doubles, each of length B) per

compute thread. An inner compute loop executes M times and does two floating point operations per data element per iteration. At the end of this inner loop we update a global sum protected by a mutex variable, which is followed by a barrier synchronization. In this way, the amount of data per thread can be varied via S and B; and the amount of computation per data element (relative to the frequency of synchronization operations) can be varied using M. An outer iteration repeats the computation N times. We use $N = 10,000$ and $B = 256$ for all experiments reported in this paper.

Since data layout and the potential for false sharing are important to the performance of any cache-based shared memory system, the micro-benchmark also allows us to vary the memory allocation and work distribution strategy. The allocation is performed either *locally* or *globally*. Local allocation means that each thread allocates the memory that will hold its data. Note that this memory is still drawn from the global address space, is served by the memory servers, etc. The Samhita memory allocator ensures that there will be no false sharing among compute threads who do their own local allocation in this way. Global allocation means that only one thread does a single large shared allocation, with each compute thread then working on its own share of that data. Hence, global allocation has a greater risk of false sharing (within a page or within a cache line) among compute threads than local allocation. There are two variations of work distribution and data access for the global allocation case. In the first variation, during the inner compute loop all the data that is accessed in each iteration is contiguous, i.e., S rows of length B, all stored contiguously. We refer to this variation as simply "global" allocation. In the second access pattern variation, the data in each iteration of the inner loop data is stored contiguously for a block of length B, but the next block accessed is strided based on the processor id. In other words, each compute thread accesses S rows of length B, but the rows assigned to the threads are interleaved. These two global allocation variations correspond to block or round-robin allocation of rows of a matrix, for example. Obviously, the potential for false sharing in the "global strided" case is the highest of the three approaches. In our performance evaluation we evaluate how false sharing amongst threads affects compute time and synchronization time in our system. We also evaluate how the amount of data accessed in the ordinary region affects both compute and synchronization time.

The API provided by Samhita is very similar to that of Pthreads [7]. In fact, all our benchmarks share the same code base, with memory allocation, synchronization and thread creation expressed as macros. These macros are processed using the `m4` macro processor. This illustrates how existing shared memory code can run using Samhita/RegC with trivial code modification

In Figures 3, 4 and 5 we compare normalized compute

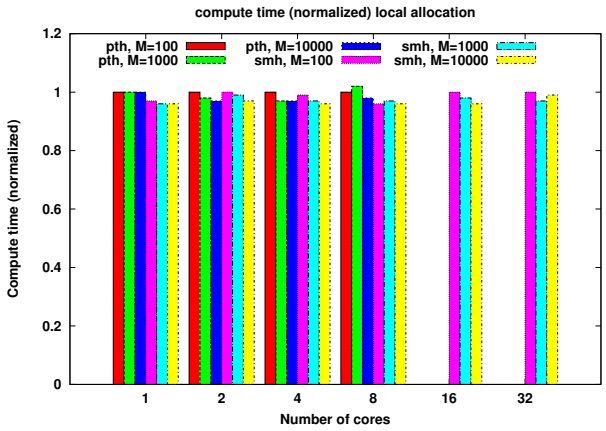

Figure 3. Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated locally.

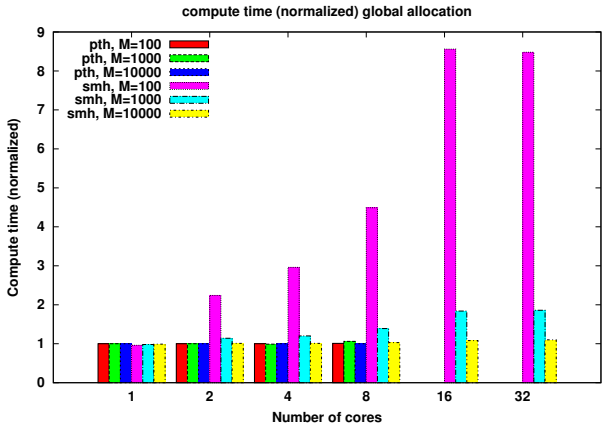

Figure 4. Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated globally.

time per thread between Pthreads (up to 8 threads) and Samhita (up to 32 threads). We normalize the runtime with the equivalent 1-thread compute time for the Pthreads implementation. This experiment compares the compute time between Pthreads and Samhita and how the compute time of Samhita varies with respect to Pthreads depending on the amount of computation performed in the inner loop, and how it is affected by false sharing.

Figure 3 compares the normalized compute time for Pthreads and Samhita as we vary both the number of compute threads and the amount of computation performed in the inner loop, with the data allocated locally. The figure shows that the normalized compute time for Pthreads and Samhita are very similar. In the absence of false sharing the time spent in computation for Samhita is very similar to the equivalent Pthread implementation, even for a relatively

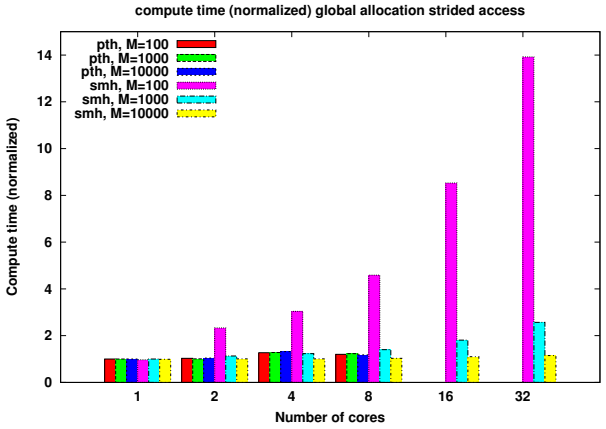

Figure 5. Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated globally, but access using strides.

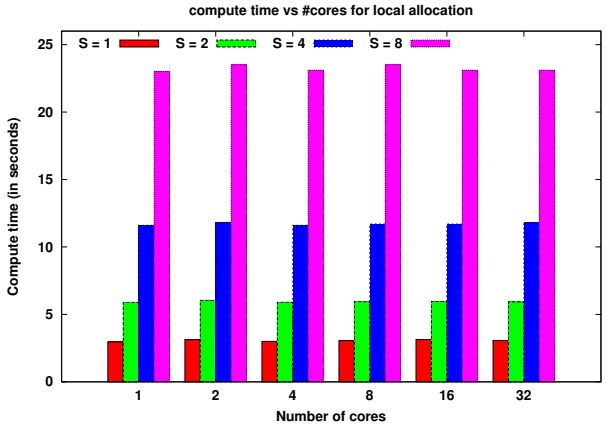

Figure 6. Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated locally.

small amount of computation (small M).

Figure 4 compares the normalized compute time when the data is allocated globally. The figure shows that when the amount of compute performed is low the added penalty incurred by Samhita due to false sharing and other overheads is noticeable. However, as we increase the amount of compute this cost is amortized and the amount of time spent in computation by Samhita is very comparable to Pthreads. This underlines the fact that even if there is some false sharing the penalty can be amortized by the amount of computation performed by the application.

Figure 5 compares the normalized compute time when the data is allocated globally and the access pattern is strided, which increases the amount of false sharing among threads. We see that when the amount of computation performed is relatively small there is a higher penalty compared to the global allocation case. However, once again this cost can be amortized by increasing the amount of compute performed over the data that is shared among the compute threads.

Figures 6, 7 and 8 compare the compute time per thread for the Samhita implementation as we vary the amount of data accessed in the ordinary region versus the number of computation threads. Figure 6 clearly shows how the computation time increases with the amount of work and amount of data accessed in the ordinary region, as expected. However, compute time per thread does not increase as the number of threads increases. This once again underlines the fact that when there is no false sharing the Samhita implementation does not incur any additional penalty.

Figure 7 shows the same trend as local allocation, with the amount of time spent in computation increasing as the amount of data accessed in the ordinary region increases. Due to modest false sharing, the compute time per thread does grow slowly as the number of compute threads in-

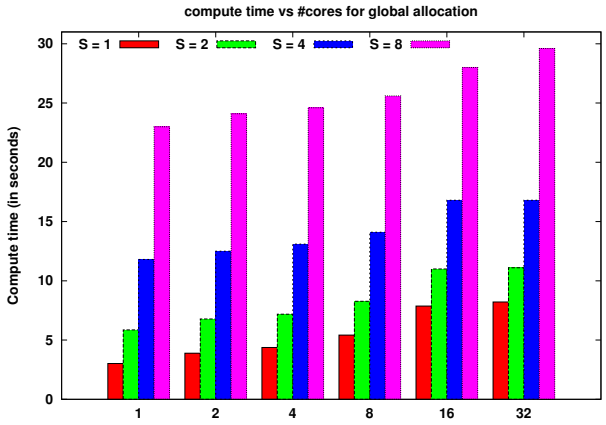

Figure 7. Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated globally.

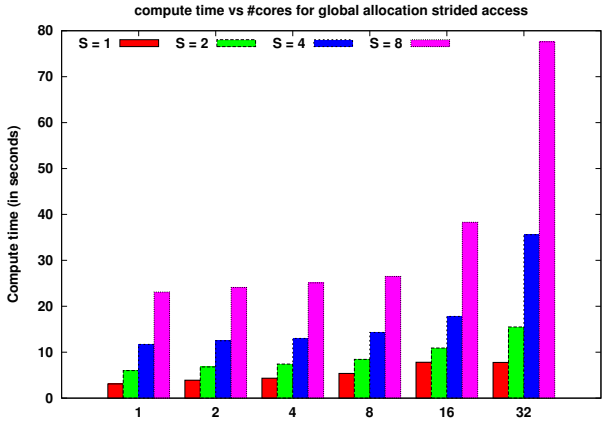

Figure 8. Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated globally, but accesses using strides.

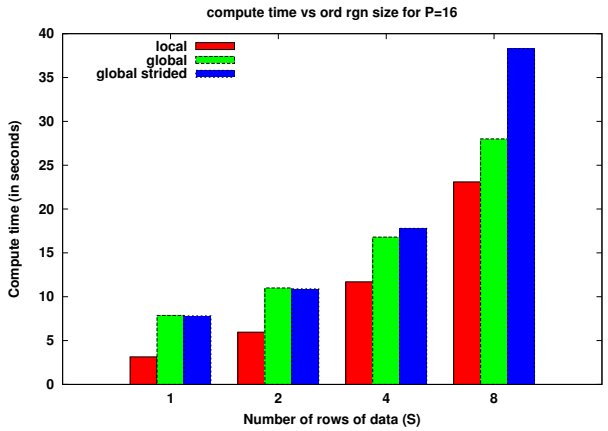

Figure 9. Compute time vs. number of rows of doubles allocated. Compute time for local, global allocation and global allocation with strided access are compared for $S = \{1, 2, 4, 8\}$ for $M = 1000$, $P = 16$ and $B = 256$.

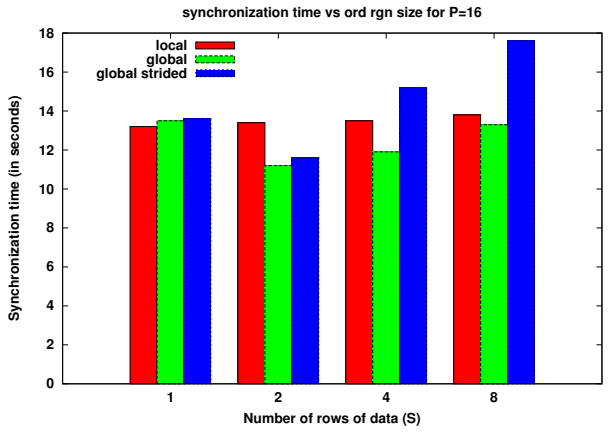

Figure 10. Synchronization time vs. number of rows of doubles allocated. Synchronization time for local, global allocation and global allocation with strided access are compared for $S = \{1, 2, 4, 8\}$ for $M = 1000$, $P = 16$ and $B = 256$.

creases. However, comparing Figure 6 and 7, we see that the penalty is not significant.

Figure 8 shows a similar trend for global strided allocation. However, due to the access pattern which increases false sharing, we see that there is a higher penalty incurred in the compute time. This penalty increases as the amount of data increases, which results in higher data false sharing.

Figure 9 compares the compute time for the Samhita implementation for 16 compute threads, with respect to the number of blocks S assigned to each compute thread. When the number of blocks is one there is no difference in the access pattern between and global and global strided allocations. We see that as the size of the ordinary region grows, the compute time increases as expected, and the penalty incurred in compute time increases based on the amount of false sharing.

Figure 10 compares the synchronization time for Samhita

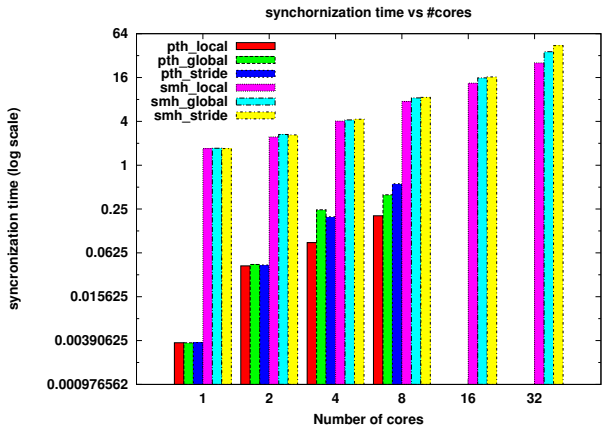

Figure 11. Synchronization time (log scale) vs. number of cores. Synchronization time for Pthreads and Samhita are compared for local, global allocation and global allocation with strided access. $M = 100$, $B = 256$ and $S = 2$ are kept fixed.

for the 16-thread run, again varying S, the number of blocks per thread. The evaluation shows that when there is no false sharing (local allocation) the increase in synchronization cost is hardly noticeable. False sharing does have an impact on synchronization since during synchronization there is increased data movement. However, Samhita's synchronization operations move only the minimum amount of data required, so that even with increased false sharing the increase in synchronization cost is not dramatic.

Figure 11 compares the synchronization time between the Samhita and Pthread implementation, varying the number of threads for $M = 100$ iterations of the inner loop. The figure shows that Samhita does incur an increased cost for synchronization. However, this is expected as the synchronization operations in Samhita perform memory consistency operations which are expensive, unlike Pthreads which performs only synchronization. The graph also shows that Samhita's synchronization overhead is not exceptionally high when compared to Pthreads, and the increase with the number of threads is not dramatic.

Figure 12 compares strong scaling speed-up between the Pthreads and Samhita implementation of the Jacobi application kernel. This kernel corresponds to the Jacobi iteration for solving the linear system corresponding to a discrete laplacian. The memory access pattern for this kernel is representative of many computations with a nearest neighbor communication pattern, i.e., the update at a given grid point depends on previous values at a some small number of near neighbors. The Pthreads and Samhita implementation use a mutex variable to protect a global variable and require three barrier synchronization operations in each outer iteration. We see that the Samhita implementation shows good speed-up up to 16 processors. And within a node Samhita tracks the Pthread implementation very well.

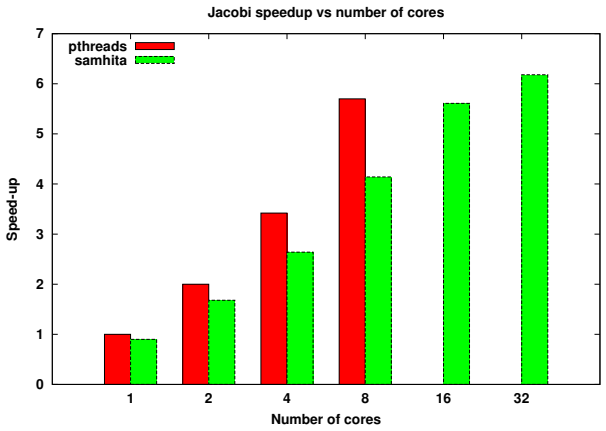

Figure 12. Parallel speed-up vs number of cores for Jacobi. Speed-up is relative to 1-core Pthread execution time.

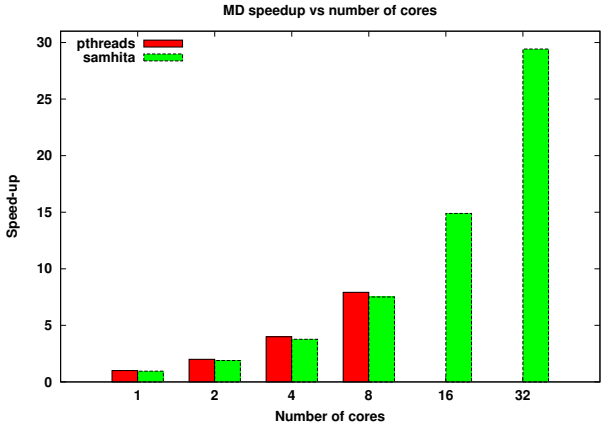

Figure 13. Parallel speed-up vs number of cores for molecular dynamics simulation. Speed-up is relative to 1-core Pthread execution time.

Figure 13 compares the strong scaling speed-up between the Pthreads and Samhita implementation of a molecular dynamics application, which performs a simple n-body simulation using the velocity Verlet time integration method. Both implementations use a mutex variable to protect variables that accumulate the kinetic and potential energies of the particles. Once again three barrier synchronization operations are required per outer iteration. We see that the Samhita implementation tracks the Pthread implementation very closely within a node and continues to scale very well up to 32 cores. This benchmark result clearly indicates that applications that are computationally intensive (the computation per particle is $O(n)$) can easily mask the synchronization overhead of Samhita enabling the applications to scale very well.

IV. Related work

The shared memory programming model has been extended to heterogeneous platforms by extending the Partitioned Global Address Space (PGAS) [9] languages to support heterogeneous architectures. An other approach is by extending the commonly used OpenMP standard [10] to support offload primitives that allow offloading loops and parallel regions to be executed on the accelerator or coprocessor. Another relatively new approach similar to OpenMP is OpenACC [11], which provides a collection of compiler directives to allow offloading of loops and regions to an accelerator. For both OpenMP and OpenACC the compiler has to perform additional work of ensuring that the correct data is offloaded and present on the accelerator or coprocessor before it can execute the code. This adds additional burden on the compiler apart from generating optimal code that can run on the accelerators and coprocessors.

GMAC [12] is an asymmetric distributed shared memory model for heterogeneous platforms. It provides the programmer with a very basic set of APIs to allocate memory on the accelerator and allows the host to access this allocated memory on the accelerator on demand. This approach unburdens the programmer from the responsibility for data movement. The asymmetric model allows the host to read data from accelerator, but does not allow direct write access. This asymmetric memory model makes the programming model complicated as it does not resemble the familiar shared memory programming model. Yan et al. [13] present a shared memory system for heterogeneous platforms. However, the system requires modification at the operating system level, making it less portable for newer architectures.

V. Conclusions

In this paper we present the architecture of Samhita and its associated consistency model, regional consistency (RegC). We evaluated Samhita and RegC using micro-benchmarks. The evaluation shows that the overheads associated with Samhita depend on false sharing, but the penalty can be amortized if the application performs a reasonable amount of computation. The synchronization cost depends on the amount of data accessed in the ordinary region and on the degree of false sharing.

We are currently working on porting Samhita for heterogeneous system that use Intel's Many Integrated Core (MIC) architecture. There are several opportunities presented for further improving the performance on such systems. Currently, Samhita performs all synchronization operations using a manager. This adds additional overhead to the synchronization cost, which also includes memory consistency operations. Samhita on a single node system can avoid contacting the manager for synchronization and reduce the overhead associated with contacting the manager during synchronization. Another orthogonal improvement would be to implement a Samhita communication layer that takes advantage of the Scalable Communication Interface (SCIF). SCIF abstracts the communication between the host processor and the Intel MIC device over the PCI express bus.

This will reduce the communication overheads by directly communicating using the PCI express bus as opposed to using a verbs proxy to communicate between the host and the coprocessor.

REFERENCES

[1] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert, "A pilot study to compare programming effort for two parallel programming models," *Journal of System Software*, vol. 81, no. 11, pp. 1920–1930, November 2008.

[2] B. Ramesh, C. J. Ribbens, and S. Varadarajan, "Is it time to rethink distributed shared memory systems?" in *Proceedings of the 17th International Conference on Parallel and Distributed Systems (ICPADS)*, 2011, pp. 212–219.

[3] "Intel Xeon Phi coprocessor (codename knights corner)," http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, accessed January 18, 2013.

[4] B. Ramesh, C. J. Ribbens, and S. Varadarajan, "Regional consistency: Programmability and performance for non-cache-coherent systems," CoRR, Tech. Rep. 1301.4490, 2013.

[5] "OpenFabrics alliance," https://www.openfabrics.org/, accessed January 18, 2013.

[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 75–.

[7] "POSIX threads," http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html, accessed January 18, 2013.

[8] A. J. Dorta, C. Rodríguez, F. de Sande, and A. González-Escribano, "The openmp source code repository," in *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2005, pp. 244–250.

[9] "Partitioned global address space (PGAS)," http://www.pgas.org, accessed January 18, 2013.

[10] "Open multi-processing (OpenMP)," http://openmp.org/wp, accessed January 18, 2013.

[11] "OpenACC directives for accelerators," http://openacc.org/, accessed January 18, 2013.

[12] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. mei W. Hwu, "An asymmetric distributed shared memory model for heterogenous parallel systems," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 347–358.

[13] S. Yan, X. Zhou, Y. Gao, H. Chen, G. Wu, S. Luo, and B. Saha, "Optimizing a shared virtual memory system for heterogenous cpu-accelerator platform," *ACM SIGOPS Operating System Review*, vol. 45, no. 1, pp. 92–100, January 2011.