Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System

Richard M. Yoo, Anthony Romano, Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{rmyoo, ajromano, kozyraki}@stanford.edu

Abstract—**Dynamic runtimes can simplify parallel programming by automatically managing concurrency and locality without further burdening the programmer. Nevertheless, implementing such runtime systems for large-scale, shared-memory systems can be challenging. This work optimizes Phoenix, a MapReduce runtime for shared-memory multi-cores and multiprocessors, on a quad-chip, 32-core, 256-thread UltraSPARC T2+ system with NUMA characteristics. We show how a multi-layered approach that comprises optimizations on the algorithm, implementation, and OS interaction leads to significant speedup improvements with 256 threads (average of 2.5× higher speedup, maximum of 19×). We also identify the roadblocks that limit the scalability of parallel runtimes on shared-memory systems, which are inherently tied to the OS scalability on large-scale systems.**

I. Introduction

Single-chip multiprocessors (CMPs) are now the norm for all computing systems, from laptops to server farms. However, due to the difficulties of thread-based parallel programming, utilizing multiple hardware cores is still challenging. Specifically, the thread-based model requires the programmer to manually manage synchronization, load balancing, and locality, which is often error-prone (e.g., races and deadlocks) or requires detailed understanding of the underlying hardware.

An alternative approach is to rely on a runtime system for concurrency management [1]–[3]. In this model, the programmer expresses computation in terms of multiple tasks, and the dynamic runtime automatically manages synchronization, load balancing, and locality in order to achieve efficient execution. Therefore, most of the hard work moves away from the programmer and into the system. MapReduce [4], for instance, is a data-parallel programming model that relies on such a runtime approach. In MapReduce, computation is specified by the programmer as a set of *map* and *reduce* functions. The runtime system spawns multiple processes or threads that apply these functions concurrently across the elements of the input dataset. The applicability of the MapReduce model on a wide range of applications has led to various implementations for clusters and CMP systems [5]–[8].

This work focuses on Phoenix [5], a MapReduce implementation for shared-memory CMPs and SMPs. While the original Phoenix performed well on small-scale systems with uniform access latencies [5], we found that the runtime significantly underperformed on large-scale systems with non-uniform memory access (NUMA) characteristics. Due to the popularity of two-socket and four-socket servers, large-scale NUMA configurations are already commonplace. If the number of cores per chip continues to scale, a single CMP chip will soon exhibit NUMA characteristics as well.

Optimizing a parallel runtime for a large-scale NUMA machine is challenging, as the runtime must manage complex interactions with both the application and the operating system. First, the runtime must enhance locality in order to hide the additional latency due to the non-uniform characteristics. Second, to handle parallel computations with large datasets, the runtime must use scalable and low-overhead data structures for input, output, and intermediate data. Lastly, the runtime must be aware that the OS mechanisms for memory management and I/O may not scale well when hundreds of threads are running concurrently. To achieve scalable performance while retaining the simplicity of the runtime-based approach, it becomes crucial to address these issues.

In this paper, we optimize the Phoenix runtime on a quad-chip, 32-core, 256-thread shared-memory system with NUMA characteristics. The specific contributions of this work are:

- We show that efficient execution on a large-scale system requires a multi-layered optimization approach. The runtime developer must carefully select the runtime algorithms, optimize their implementations around NUMA challenges, and deliberately manage the interactions with the operating system.
- We demonstrate the approach with the Phoenix runtime using a 256-thread UltraSPARC T2+ system. The optimized runtime exhibits significantly improved scalability over the original system; for 256-threads, the new runtime improves speedup by 2.5× (maximum improvement of speedup up to 19×).
- We also identify the important roadblocks that limit further scaling of the Phoenix runtime on shared-memory NUMA systems. Specifically, we find that interacting with the operating system for memory allocation and I/O (i.e., sbrk() and mmap()) becomes a crucial issue at large thread counts.

The rest of the paper is organized as follows. Section II provides a brief overview of Phoenix and our NUMA system. Section III details our methodology, while Section IV discusses the effectiveness of the optimizations. Section V identifies and discusses the bottlenecks that were critical to further scale the runtime. Related work is presented in Section VI, and Section VII concludes the paper.

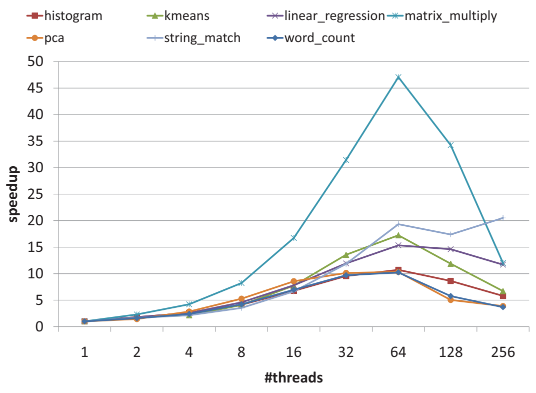


Fig. 1. Application speedup for the original Phoenix system.

Hardware Settings

CPU	4 UltraSPARC T2+ chips 8 cores per chip, 8 HW contexts per core total of 256 hardware contexts on system
Per Core L1 Cache	8 KB, 16 B lines, 4-way assoc write through, physically tagged / indexed
Shared L2 Cache	4 MB, 64 B lines, 8-banked, 16-way assoc write back, physically tagged / indexed
Memory	128 GB over 16 FBDIMM channels 4 channels local to each chip 300-cycle latency for local access 100-cycle overhead for remote access 85 GB/sec for read, 42 GB/sec for write
Interconnect	single external coherence hub 130 GB/sec bisection for coherence

Software Settings

Operating System	Sun Solaris 5.10
Compiler	GCC 4.2.1 with -O3 optimization

TABLE I
THE CHARACTERISTICS OF THE LARGE-SCALE SHARED-MEMORY SYSTEM USED IN THE STUDY.

II. Background and Motivation

MapReduce [4] is a parallel programming framework and runtime system proposed originally for cluster-based systems. Since it is representative of high-level, data-parallel runtimes, the lessons learned from scaling MapReduce can be generalized to other similar frameworks.

A. The Phoenix Implementation of MapReduce

The original MapReduce system was designed for clusters [4], which spawned multiple *processes* to achieve parallelism. In contrast, Phoenix [5] is a shared-memory version of MapReduce targeted for multi-core and multiprocessor systems. Phoenix uses shared-memory *threads* to implement parallelism.

First, a user provides the runtime with the map / reduce functions to apply on the data. The runtime then launches multiple worker threads to execute the computation. In the map phase, the input data is split into *chunks*, and the user-provided map function is invoked on each chunk. This generates intermediate key / value pairs, which reside in memory. In the reduce phase, for each unique key, the reduce function is called with the values for the same key as an argument, to reduce them to a single key / value pair. Results from all the reduce tasks are merge sorted by keys to produce the final output.

Compared to the cluster version, the most striking aspect of Phoenix is that the workers communicate by accessing a shared address space. Unlike the cluster system where communication takes place through the distributed file system and remote procedure calls [4], [7], the communication overheads for shared-memory MapReduce are low. On the other hand, because the threads contend over the single address space, the way threads access memory and perform I/O can have a first-order impact on the overall performance. This can be a limitation in a large-scale parallel system, as we show in Section V.

B. The Large-Scale Shared-Memory System

The most popular form of large-scale, shared-memory machines today are multi-socket servers that use two to four multi-core chips, with caches and main memory channels physically distributed across those chips. Such a system can readily support hundreds of threads in a single unit, but exhibits variable memory access latencies. Next generation multi-core

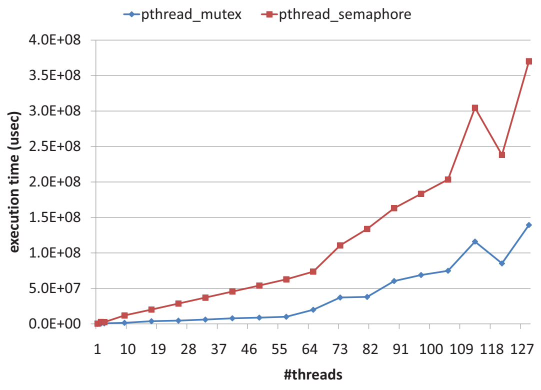
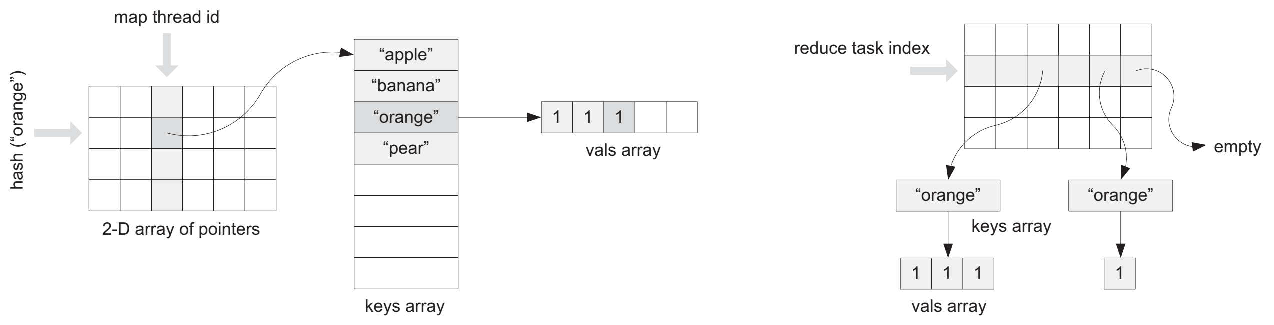


Fig. 2. The latency of synchronization primitives.

chips with hundreds of cores on a single die will likely exhibit similar NUMA characteristics.

The original Phoenix runtime targeted CMP and SMP systems with uniform memory access characteristics and 24 to 32 hardware threads. We target the Sun SPARC Enterprise T5440 system [9], [10], summarized in Table I. Each of the four T2+ chips supports 64 hardware contexts for a total of 256 in the whole system. Each chip has 4 channels of locally attached main memory (DRAM) as well. Notice that the accesses to remote DRAM are 33% slower than the accesses to locally attached memory; any program that uses more than 64 threads will experience such non-uniform latency.

Figure 1 shows the scalability of the original Phoenix runtime measured on T5440 with the released applications. Despite the parallelism available in these applications, none of them scales beyond 64 threads, and most of them actually slow down when more threads are involved. This is the result of the increased memory latency, loss of locality, and high contention when utilizing threads across multiple chips. We explain these issues in detail in Section IV.



(a) Access pattern during the map phase.

(b) Access pattern during the reduce phase.

Fig. 3. Phoenix data structure for intermediate key / value pairs.

It is interesting to also note that the performance of several OS primitives deteriorates when using more than 64 threads on T5440. Figure 2 shows the latency of synchronization operations as we scale the number of threads. We measured the time needed for a half million lock acquisitions and releases in order to increment a shared counter. Similar to the behavior in Figure 1, the cost for synchronization increases drastically as we cross the chip boundary.

We also observed similar behavior with Phoenix and with low-level OS primitives on an 8-chip, 32-core, 32-thread Opteron system running x86 Linux. This verifies that the challenges observed are fundamental to large-scale systems and not the artifacts of the T5440 machine we used. We omit the x86 results due to space limitations. However, the optimizations presented in this paper led to significant performance improvements on this system as well.

III. Optimizing Phoenix for Large-Scale & NUMA

A parallel runtime such as Phoenix continuously interacts with the user application and the operating system. Therefore, it is natural that the optimization strategies for large-scale, NUMA systems are multi-layered. The approach we propose comprises three layers: algorithm, implementation, and OS interaction.

A. Algorithmic Optimizations

To perform well on a NUMA machine, the basic algorithms used in the runtime must be scalable and NUMA aware. For instance, the original Phoenix algorithm is not NUMA aware in that local and remote worker threads are indistinguishable at the algorithm level. While this is not an issue for small-scale systems, it becomes important for locality-aware task distribution in a large-scale, NUMA environment. When the input data for the application is brought into memory via mmap(), Solaris distributes the necessary physical frames across multiple *locality groups*, i.e., chips and their separate memory channels [11]. If Phoenix blindly assigns map tasks to threads, it could end up having a local thread continuously work on remote data chunks, thus causing additional latency and unnecessary remote traffic.

We resolved the issue by introducing a task queue per locality group, and by distributing tasks according to the location of the pertaining data chunk. Map threads retrieve tasks from their local task queue first, and when the queue runs out, they start stealing tasks from remote task queues. Although similar in spirit to the work stealing algorithm utilized in other runtimes [1], the difference here is that we maintain one task queue for each locality group (instead of each thread), hence creating a load balancing approach that is compatible with NUMA memory hierarchies.

B. Implementation Optimizations

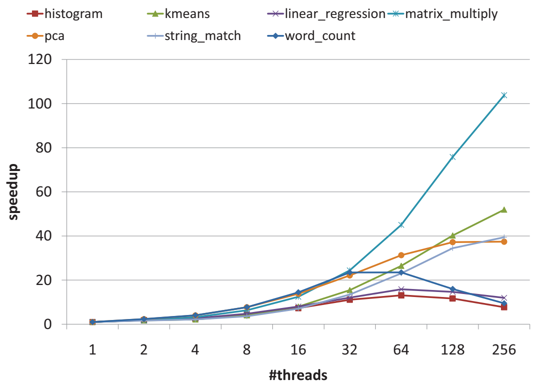
To fully utilize a large-scale machine, applications must include a non-trivial amount of work. This typically implies large input datasets that the runtime system must handle efficiently. Meeting this requirement in MapReduce is quite challenging, since a typical MapReduce application generates a commensurate amount of intermediate and output data as well. Unlike the original MapReduce where data storage and retrieval are limited by the raw bandwidth of the network and the disk subsystem [4], we found that in Phoenix, the design and performance of the in-memory data structures used to store and retrieve intermediate data are crucial in overall system performance.

Figure 3 gives a simplistic view of the core data structure of the original Phoenix, which is used to store the intermediate key / value pairs generated from the map phase. In particular, Figure 3(a) depicts the typical access pattern to this structure in the map phase, where a worker thread is storing an <"orange", 1> intermediate pair. Phoenix internally maintains a 2-D array of pointers to *keys array*, where the width is determined as the number of map workers, and the height is fixed by a default value (256). During the map phase, each map worker uses its thread id to index into this array column-wise. Once the column is determined, the element is indexed by the hash value of the key. All the threads use the same hash function. Therefore, for each thread, a column of pointers acts as a fixed-sized hash table, and one keys array amounts to a hash bucket.

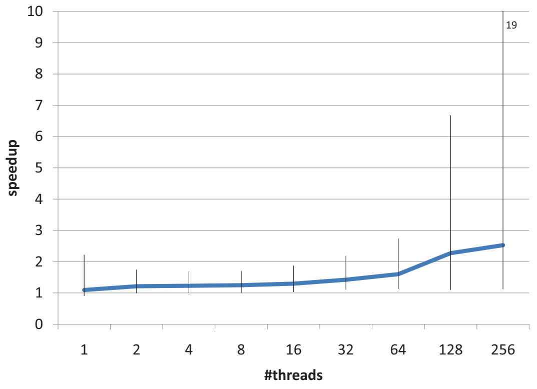
Specifically, each keys array is implemented as a contiguous buffer, and keys are stored sorted to facilitate binary search. Each entry in the keys array also has a pointer to a *vals array*; this structure stores all the values associated with a particular key. To maximize locality, vals array is implemented as a single contiguous array as well. During the map phase, both the keys array and vals array are thread local.

application	input dataset	description
histogram	1.4 GB BMP image	Computes the RGB histogram of an image
kmeans	500,000 data points	Performs k-means clustering analysis over data points
linear_regression	3.5 GB data	Applies linear regression best-fit over data points
matrix_multiply	3,000 × 3,000 matrices	Matrix multiplication
pca	3,000 × 3,000 matrix	Performs principal components analysis over a matrix
string_match	500 MB dictionary	Pattern matches a set of strings against streams of data
word_count	500 MB random texts	Counts the number of unique word occurrences

TABLE II
THE WORKLOADS USED WITH THE PHOENIX MAPREDUCE SYSTEM.



(a) Application speedup with the optimized version of Phoenix.



(b) Relative speedup over the original Phoenix system.

Fig. 4. Performance improvements summary.

Next, during the reduce phase (Figure 3(b)), each row amounts to a reduce task; a reduce thread grabs a row from the matrix and invokes the reduce function over all the keys array in that row. Notice the disparity between how the threads access the 2-D array structure during the map phase (column-wise) and the reduce phase (row-wise); pictorially, every 'crossing' in the access pattern signifies that a worker thread has to access data structures prepared by another thread, which could require remote memory accesses in the NUMA environment. On the other hand, since Phoenix is implemented in C, the 2-D array structure is represented as a row-major array in memory. Hence, the above design optimizes for the locality of reduce phase; map phase exhibits little locality since the sequence of keys confronted during the phase is close to random.

With the medium-sized datasets used on small-scale systems, this data structure design was acceptable. But when the input was significantly increased on the 256-thread NUMA system, we observed some performance pathologies. As described earlier, the keys array structure was implemented as a sorted array to utilize binary search. Although it provided fast lookup, the downside of this implementation was that when the buffer ran out of space, the entire array had to be reallocated. Even worse, when a new key was inserted in the middle of the array, all the keys coming lexicographically after the new key had to be moved. As we increased the input dataset, this problem became more prominent. Similar buffer reallocation issues occurred on the vals array as well.

Improving the keys array structure was more complicated than we first expected. Since massive amounts of intermediate pairs were being inserted into the hash table, slight latency increases such as pointer indirection obliterated any performance gains. For example, we tried a design that implemented keys array as a linked list, but failed to improve performance. Replacing the structure with a tree was not an option either, since frequent rebalancing would have been costly. Instead, we settled by significantly increasing the number of hash buckets so that on average only one key resides in a keys array. Detailed reasoning behind this decision will follow (Section IV-C1), but simply put, increasing the number of hash buckets linear to the number of unique keys was sufficient.

For the vals array, we implemented an iterator interface to the buffer and exposed this interface to the user reduce function. This allowed us to have a buffer that is comprised of few disjoint memory chunks (good for locality purposes) while still maintaining the sequential accessibility through the iterator interface. This design completely removed the reallocation issue. Additionally, to tolerate the increased memory latencies in the NUMA system, we implemented prefetching functionality behind the interface. Note that unlike the map phase where we can avoid accessing remote memory by assigning tasks based on locality, in the reduce phase a worker might not be able to avoid performing remote accesses since the intermediate pairs with the same key may be produced by workers across all the locality groups. Therefore, sequential accesses and prefetching to vals array become important.

We also experimented with combiners [4], where each thread invoked the combiner at the end of the map phase to decrease the amount of reduce phase remote memory traffic. However, with the prefetching in place for the reduce phase, combiners made little difference.

After the data structure was improved, other minor factors such as task generation time were affected by the increased input size as well. We detail the issue in Section IV-C2.

C. *OS Interaction Optimizations*

Once the algorithm and the implementation are optimized, interactions with the OS are apt to become the next bottleneck. Specifically, Phoenix frequently uses two OS services: memory allocation and I/O.

Phoenix exerts significant pressure on memory allocators due to its large memory footprint as well as its peculiar allocation pattern. First, Phoenix generates a significant amount of intermediate and output data. Even worse, the memory needs per key are usually unpredictable. As for the allocation pattern, in Phoenix, the thread that allocates memory (e.g., a map thread) rarely is also the thread that deallocates the memory (e.g., a reduce thread). This mismatch can incur contention on the per-thread heap locks when multiple threads try to manipulate the same thread heap. We experimented with a sizable number of memory allocators to compare their performance. However, at high thread count, we noticed that the parallel allocator performance was limited by the scalability of the sbrk() system call. We discuss the issues in detail in Section V.

The mmap() system call is used to read in input data. Once the user passes the pointer to the mmap()ed region as a runtime input, multiple threads will concurrently fault in the input data while invoking their map functions. Therefore, as we increased the number of worker threads, we observed the mmap() scalability being crucial to some of the workloads. More importantly, however, mmap() was also being used to implement thread stacks. Using the SunStudio [12] profiler, we found that thread join was a major bottleneck at large thread counts. Further analysis revealed that the increased thread join time was due to the calls to munmap(), which was used to deallocate thread stacks. To address the issue, we implemented a thread pool that reused threads across different MapReduce phases and iterations (multiple sets of map and reduce functions), reducing the total number of calls to munmap() to a strict minimum.

IV. Evaluation

A. *Performance Improvements Summary*

We implemented the optimizations in Section III on Phoenix and measured its performance by executing the applications included in the original release. Table II describes the workloads and their input datasets. We omitted the reverse_index workload because it was I/O bound. Compared to the original release [5], we significantly increased the input datasets to stress all the hardware contexts available on our system. In all the experiments we bound threads so that we filled up a chip (64 threads) first before providing for the other chips.

Figure 4(a) summarizes the scalability results for the optimized runtime. Specifically, workloads matrix_multiply, kmeans, pca, and string_match scaled up to 256 threads.

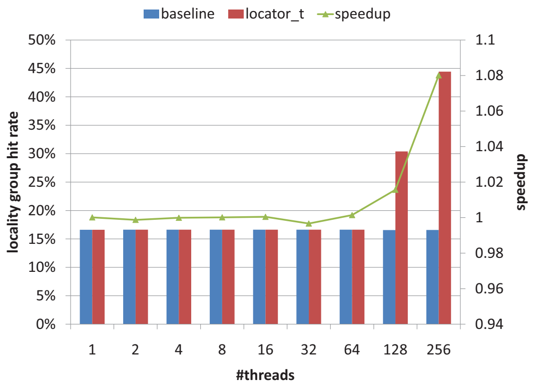


Fig. 5. Locality group hit rate improvement on string_match.

workload	populated bkts.	keys per bkt.	vals per key
histogram	256	2	256
kmeans	100	1	78
linear_regression	5	1	905
matrix_multiply	0	0	0 (bypasses reduce)
pca	21 256	2 274	1 (phase 1) 1 (phase 2)
string_match	0	0	0 (bypasses reduce)
word_count	256 244	156 31	34 (phase 1) 1 (phase 2)

TABLE III
Hash table key distribution.

However, some of the workloads still did not scale particularly well. We discuss their bottlenecks in detail in Section V, but in the remainder of this section we first focus on the optimizations that turned out to be successful.

Figure 4(b) measures the relative speedup of the new runtime over the original. It essentially compares Figure 4(a) and Figure 1 using the same dataset. In the figure, the top and bottom of each vertical bar denotes the maximum and minimum performance improvement achieved at a particular thread count, respectively; the horizontal line that connects those bars represent the harmonic means of speedups obtained from the entire workload. The optimized runtime led to improvements across all thread counts. For less than 64 threads (single chip), the average improvement was 1.5×, and the variation across applications was rather small (maximum of 2.8×). For large-scale, NUMA configurations with 128 or 256 threads, the optimizations were significantly more effective, reaching 19× in maximum and 2.53× on average. We saw similar differences between the two runtimes on the 8-socket, 32-core Opteron system as well.

Since the various optimizations had interrelated effects on the overall performance, e.g., iterators interacting with memory allocation pressure, we could not fully isolate the contributions of each class of optimization for all the workloads. Qualitatively speaking, OS interaction optimizations had the most impact, implementation optimizations next, and algorithmic optimizations the least. We provide insights into the impact of each optimization class in the following subsections.

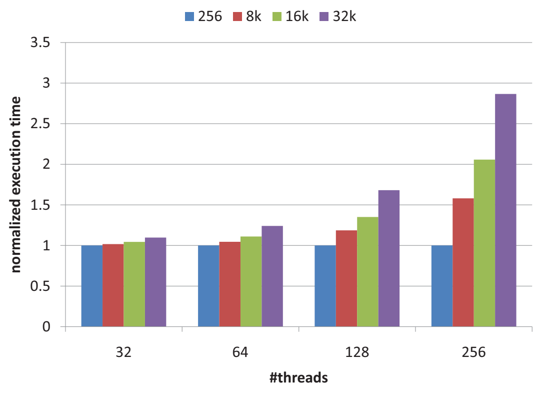


Fig. 6. kmeans sensitivity to hash bucket count.

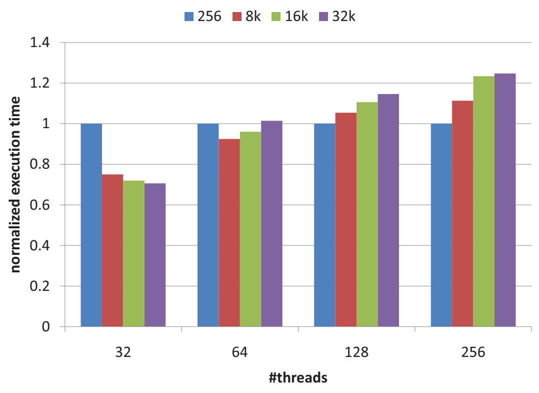


Fig. 7. word_count sensitivity to hash bucket count.

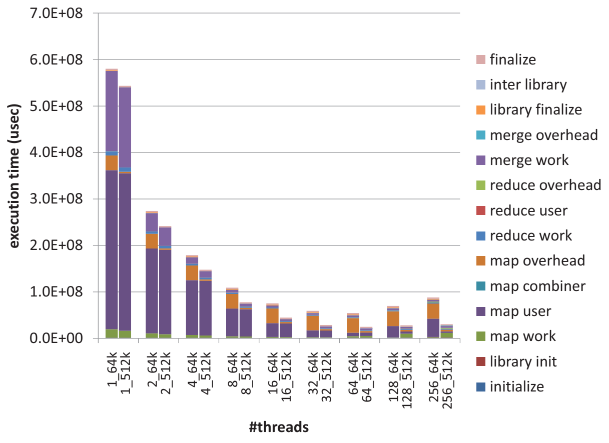


Fig. 8. pca execution time breakdown.

B. Impact of Algorithmic Optimizations

We implemented the per locality group task queue and task distribution as described in Section III-A. We utilized the meminfo() interface in Solaris [11], which returns the locality group of the physical memory that backs the virtual address being queried. According to the locality group information for an input data chunk, a task was queued to the pertaining locality group task queue. Note that the locality aware task distribution is orthogonal to the use of an application specific splitter on the input data; once the input is split in whatever means necessary, map tasks are distributed in a locality aware manner.

Figure 5 compares the *locality group hit rate* of the optimized runtime against that of the baseline for the string_match application. The hit rate represents the percentage of map task data that was served by a memory channel attached to the same chip (low latency memory access). When threads were confined to one chip, they were forced to take locality group misses when accessing data on remote memory. Hence, both schemes exhibited about 17% locality group hit rate for this case. However, when threads were created across multiple chips, careful placement of tasks could readily improve locality group hit rate. The proposed optimization effectively utilized this opportunity, which led to 30% and 44% locality group hit rate at 128 and 256 threads, respectively. On the average, this optimization led to 8% speedup improvement over the baseline at 256 threads.

C. Impact of Implementation Optimizations

1) Hash Table Improvements: In Section III-B, we discussed how large datasets led to performance issues with the core Phoenix data structure that stored the intermediate key / value pairs. We addressed this issue by increasing the number of hash buckets and by implementing the iterator interface. Especially, the increased hash bucket count avoided buffer reallocation and copying. However, not all the workloads benefited from the optimization. Table III shows the distribution of keys in the hash table for 64 threads, when the number of hash buckets was set to 256. Note that these are the averages taken over all the threads; the actual number of keys or values for each thread can be significantly higher. From the table, it can be seen that although workloads such as pca and word_count generated significant amount of unique keys, other workloads, e.g., kmeans and linear_regression, did not.

The problematic workloads generated a fixed number of unique keys regardless of the input size. Therefore, increasing the hash table size ended up in a net slowdown for such applications, and this slowdown actually became worse with more threads. Figure 6 shows the normalized execution time of kmeans increasing with higher bucket counts (time normalized to that with 256 buckets). Our analysis showed that when a reduce worker traversed down a row of the 2-D array (see Section III-B), there was a fixed cost just to find a remote keys array structure empty. Due to NUMA effects, this cost increased when more chips were used. In Figure 6, this effect manifests as worse performance with increasing number of threads. So, for workloads that did not generate a large number of unique keys, the hash bucket count had to be kept small.

Even for the workloads that did benefit from the increased bucket count, the improvements were not uniform. Figure 7 shows the normalized execution time of word_count when the number of hash buckets was increased. Again, the execution time is normalized to the 256 buckets case. At low thread counts (∼32 threads), increasing the hash table size resulted in speedup. However, as more threads were added, the benefit

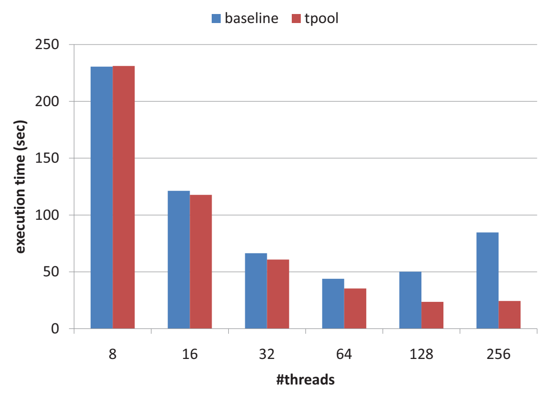


Fig. 9. kmeans performance improvement due to thread pool.

Without Thread Pool

♯ threads	time (sec)	♯ calls	time / call (msec)
8	0	20	0
16	0.31	1947	0.16
32	0.689	4499	0.15
64	1.695	9956	0.17
128	4.548	14661	0.31
256	8.219	14697	0.56

With Thread Pool

♯ threads	time (sec)	♯ calls	time / call (msec)
8	0	10	0
16	0.002	13	0.15
32	0.002	18	0.11
64	0.008	33	0.24
128	0.016	44	0.36
256	0.065	102	0.64

TABLE IV
THE EFFECT OF THE munmap() SYSTEM CALL ON kmeans.

was reversed due to the increased time spent in the merge phase. In Phoenix, worker threads output one result structure for each reduce task, which were merged by a tree-based parallel merge sort at the end. Since each hash bucket amounted to one reduce task, having an excessive amount of hash buckets resulted in a ‘wider’ merge tree. At the same time, increasing the thread count made the merge tree ‘deeper.’ When the overhead in the merge phase outweighed the savings from the reduced memory reallocation, increasing the number of hash buckets resulted in a net slowdown.

To summarize, no single hash table size worked best for all the workloads. For applications that generated small number of keys, the hash table size had to be kept to a minimum. On the other hand, for workloads with large amount of keys, the hash bucket count could only be increased as long as it did not negatively affect the merge phase execution time. In the optimized version of Phoenix, we made the hash table size user tunable, while providing the default values for efficient execution on each workload. As a rule of thumb, increasing the number of hash buckets linearly to the number of unique keys was sufficient.

2) Task Generation Time: We also made an interesting observation concerning the effect of map task chunk size on the overall execution time. As reported in [5], it was generally understood that varying the chunk size relative to the cache size did not have a significant impact on locality, due to the streaming nature of MapReduce applications. However, the chunk size had a direct impact on the map phase task generation time, which started to consume a noticeable amount of execution as the input datasets became larger. Figure 8 shows the breakdown of the pca execution time as we varied the input chunk size from 64 KB to 512 KB. At the default value of 64 KB, pca generated millions of map tasks, which is captured as the prominent map overhead portion in the figure. At higher number of threads, this overhead actually dominated the entire execution time.

In our case, increasing the chunk size up to 512 KB was sufficient to reduce the task generation time to a minimum (right columns in Figure 8). However, for systems of larger scale, it might be necessary to parallelize the task generation phase altogether.

D. *Impact of OS Interaction Optimizations*

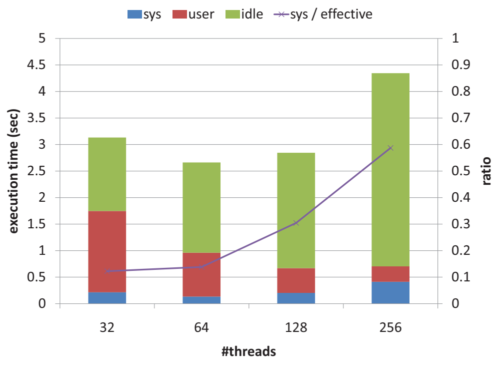
As described in Section III-C, the mmap() and munmap() system calls used to implement and destroy thread stack introduced high overhead at large thread counts. Especially, kmeans was affected the most since it iterated over multiple MapReduce instances, which resulted in repeatedly creating / destroying a large number of threads. The upper half of Table IV shows the effect of munmap() on kmeans measured with truss. It was expected that the number of calls to munmap() increased with increasing number of threads; but the problem was that each call to munmap() took longer as we used more threads.

To alleviate this problem, we implemented a thread pool. The lower half of Table IV presents the same data when the thread pool was enabled. The thread pool reduced the number of calls to munmap(), which made the time spent inside munmap() negligible. In result, the workload showed notably improved scalability (Figure 9). However, notice that the time spent for each call to munmap() stayed more or less the same. We discuss the issue in detail in Section V-B.

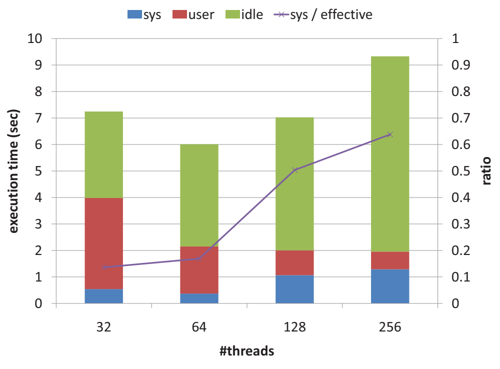
V. CHALLENGES AND LIMITATIONS

Although we were able to significantly improve the scalability of Phoenix, workloads histogram, linear_regression, and word_count still did not scale up to 256 threads. We used /usr/bin/time to assess where the execution time was being spent. Figure 10 shows the result with *effective* time defined as user + sys time. It was clear that the 3 non-scaling workloads shared two common trends. First, the idle time increased with increasing number of threads, dominating the total execution time at high thread count. Second, the portion of actual computation time assumed by kernel code (sys / effective) significantly increased as we went beyond the single chip boundary.

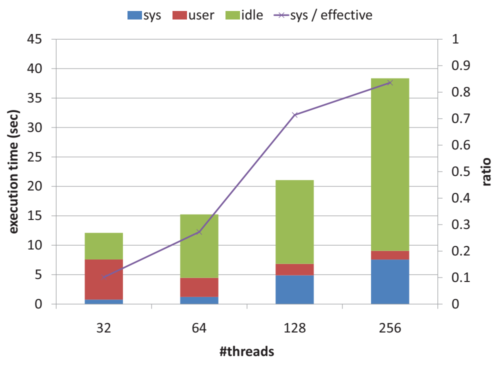
From the profiler analysis we also found that at 256 threads, histogram and linear_regression spent 64% and 63% of their execution time idling for data page fault, and that word_count spent 28% of its execution time in sbrk() called inside the memory allocator. word_count spent an additional 27% of



(a) Execution time breakdown on histogram.



(b) Execution time breakdown on linear_regression.



(c) Execution time breakdown on word_count.

Fig. 10. Execution time breakdown for non-scalable workloads.

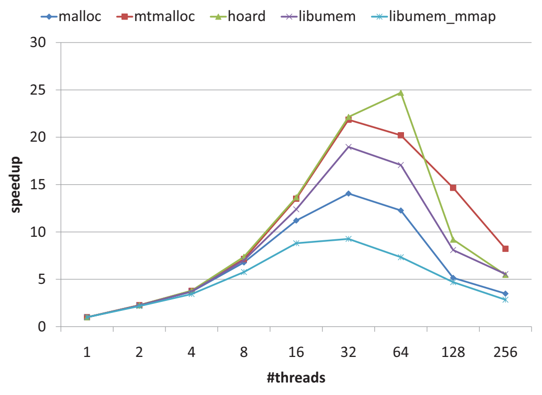


Fig. 11. Memory allocator scalability comparison on word_count.

execution time idling for data pages as well. In short, scalability on these three workloads were hampered by two OS services: memory allocation and I/O.

We would like to be clear that these issues are not related to the physical I/O performance. We observed the same issue even when we warmed up the OS file system cache by running the same workload with the same input multiple times. Measurements were taken only when the workload execution time had been stabilized. Hence, all the scalability issues we report here are due to the serialization introduced by the memory management portion of the operating system. It is also important to point out that simply substituting mmap() with read() resulted in unacceptable performance for those workloads that depended heavily on mmap(): namely, histogram and linear_regression.

A. *Memory Allocator Scalability*

Compared to the libc sequential allocator, the Solaris concurrent allocator mtmalloc provided improved performance. However, when it frequently called sbrk(), the allocator exhibited scalability problems. Inside sbrk(), a single user-level lock kept other threads from expanding the process's data segment, while per-address space locks protected in-kernel virtual memory object. As more threads relied on sbrk() to satisfy allocation requests, sbrk() became the point of contention.

We experimented with various allocators[1]. Figure 11 compares the scalability of different memory allocators on word_count. Solaris alone provided (at least) 5 different memory allocator implementations: malloc, bsdmalloc, mtmalloc, libumem, and mapmalloc. Among those, mtmalloc and libumem had concurrency support. Especially, libumem could be paired with a backend using mmap() instead of sbrk(); in Figure 11, libumem_mmap denotes the performance of libumem with mmap() backend. Also, the hoard trend line denotes the performance of the Hoard [14] memory allocator.

In summary, no allocator successfully scaled up to 256 threads; once the sbrk() became the bottleneck, no allocator was able to scale. We believe this issue opens up a new research opportunity for concurrent allocators and virtual memory subsystems on large-scale shared-memory systems. Comparing the results of libumem and libumem_mmap allowed us to arrive at the surprising conclusion that the mmap() backend performed even worse than the sbrk() backend; despite having no user-level lock like sbrk(), the mmap() backend managed to scale worse than the sequential malloc.

B. mmap() *Scalability*

The scalability of the mmap() system call is critical to shared-memory runtimes like Phoenix since workloads typically mmap() user input data and use multiple threads to fault them in. To quantify the problem, we created a microbenchmark

[1]We could not experiment with TCMalloc, an open-source memory allocator with NUMA extensions [13], since the library was not ported to SPARC.

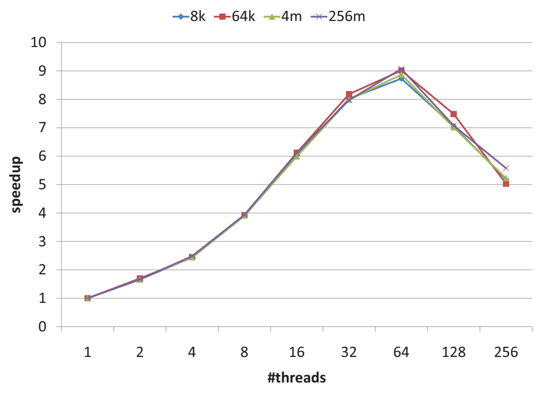


Fig. 12. mmap() microbenchmark scalability results.

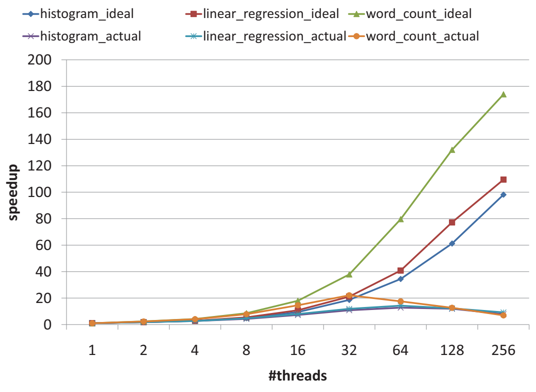


Fig. 13. Ideal vs. actual speedup for non-scalable workloads.

that assessed mmap() scalability. It simply mmap()ed a user input file and statically assigned the data chunks across threads. Then, the threads computed the total sum of all the data by streaming through the chunks assigned to them. No user level synchronization primitives were called, and memory allocation was only performed in the sequential portion of the program (an array to hold pthread_ts).

Figure 12 shows the results for the microbenchmark over varying page sizes. In short, mmap() exhibited serialization as we crossed the chip boundary. The fact that the performance was identical regardless of the page size also suggested that the problem was not related to TLB pressure. The issue was not limited to Solaris, either; executing the same microbenchmark on Linux produced similar results.

C. Importance of OS Scalability

To investigate to which degree these workloads were affected by the poor scalability of OS primitives, we tried to predict the ideal scalability of each one. Figure 13 shows the achievable speedup for each workload, when only the time spent executing the user code was considered (including the Phoenix runtime). It was clear that the operating system scalability was a significant issue, as the user time on these applications scaled well.

To better support our claim, we also created a synthetic benchmark that tried to get rid of sbrk() and mmap() effects on

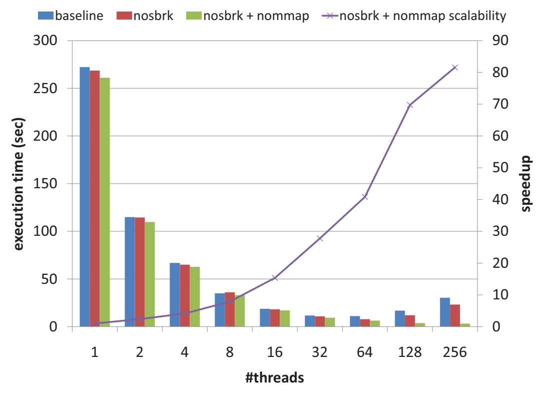


Fig. 14. word_count performance without sbrk() and mmap() effects.

word_count application by, first, pre-allocating all the memory it needed, and second, by reading in all the data into the user address space before it started executing. In Figure 14, we can see that removing each OS bottleneck gradually improved the application performance. The workload scaled well up to 256 threads when both the bottlenecks had been removed.

D. Discussions

Scaling operating systems for large-scale, shared-memory systems is an active field of research [15], [16]. Nonetheless, we have shown that the problems are more severe and readily encountered than expected, even with a highly optimized operating system like Solaris. Clearly, further research is warranted.

It is also important to point out that these issues are specific to shared-memory MapReduce. Unlike the cluster implementation, in shared-memory MapReduce, worker threads share a single address space. Therefore, how one thread allocates memory and performs I/O has a direct impact on the overall system performance. In an effort to localize OS interactions as much as possible, we also tried MapReducing-MapReduce: instantiating one MapReduce instance per locality group and crossing the chip boundaries only at the final phase to merge the results. However, since those MapReduce instances essentially constituted a single process, they still suffered from similar OS scalability issues.

One other way to overcome the problems we faced on the NUMA system would be to have each MapReduce worker implemented as a separate process running on a dedicated OS instance (virtual machine) but on the same physical machine. Unlike the cluster environment where workers communicate over the network infrastructure, we could build virtual machine mechanisms that would allow communications across different virtual machines to take place through shared memory, without even invoking the hypervisor every time. We leave this topic as future work.

VI. Related Work

Previous efforts in scaling runtimes for large-scale shared-memory systems include optimizations performed on the OpenMP [17]–[19] and Java [20] systems. However, none of these efforts focused on the unique position of the runtime in a software stack, to propose a comprehensive, multi-layered approach. Moreover, compared to our system, the NUMA systems utilized in these projects were typically small in scale or density. Hence, the OS scalability issues were not as profound.

A variety of production operating systems offer interfaces to reason about and enforce NUMA placement [11], [21]. However, we have shown that the problems are more severe and imminent, even when these mechanisms are used. For the future CMP systems and NUMA machines to be successful, significant research should be dedicated to OS scalability. Current investigations into OS scalability are a step in this direction [15], [16].

Kunz [22] observed similar locality and scalability issues with respect to OS in his study of a 64-chip FLASH machine that spanned across multiple racks. We concluded that we were facing the same issues within smaller form factors (as small as a single unit) due to the high density made available by modern multi-core systems. Unfortunately, the reduced latencies due to smaller distances do not change the picture significantly. These facts all the more stress the importance of a systematic approach to optimizing a runtime system on a large-scale NUMA machine.

VII. Conclusion

Dynamic runtime systems are a promising approach to solve parallel programming issues for parallel systems. However, optimizing a runtime for a large-scale, shared-memory environment with NUMA characteristics can be non-trivial, because of the complex interactions among the runtime, the user application, and the operating system. Such interactions can reduce locality and introduce scalability issues.

In this paper we point out that optimizing a runtime on a large-scale NUMA system calls for a comprehensive approach that considers all the layers of algorithm, implementation, and OS interaction. We demonstrated this approach by optimizing Phoenix, a shared-memory MapReduce system, and measured its performance on a 256-thread NUMA system. We significantly improved the scalability over the original version, achieving an average speedup improvement of 2.5×, and a peak speedup improvement of 19×. Moreover, we documented scalability limitations that are inherently tied to operating system services and can motivate further work on this topic.

The optimized version of Phoenix is publicly available at http://mapreduce.stanford.edu.

Acknowledgments

This work was supported by the Stanford Pervasive Parallelism Laboratory and the Gigascale Systems Research Center. The T5440 system was generously provided by Sun Microsystems. We thank Nathan Bronson, Jared Casper, and Megan Wachs for their help with setting up and maintaining the system. We also thank Woongki Baek, David Lo, and Daniel Sanchez for their input on the initial version of the paper.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *PPoPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1995, pp. 207–216.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005, pp. 519–538.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, Jr., and S. Tobin-Hochstadt, "The Fortress language specification, version 1.0," 2008. [Online]. Available: http://research.sun.com/projects/plrg/fortress.pdf

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI '04: Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, 2004, pp. 137–149.

[5] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 13–24.

[6] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 260–269.

[7] The Apache Software Foundation, "Hadoop," *http://hadoop.apache.org*.

[8] M. de Kruijf and K. Sankaralingam, "MapReduce for the Cell B.E. architecture," *University of Wisconsin Computer Sciences Technical Report CS-TR-2007-1625*, October 2007.

[9] Sun Microsystems, "Sun SPARC enterprise T5440 server architecture," *White Paper*, October 2008.

[10] S. Phillips, "VictoriaFalls: Scaling highly-threaded processor cores," in *Hot Chips 19*, 2007.

[11] Sun Microsystems, "Solaris: Memory and thread placement optimization developer's guide," 2007.

[12] ——, "Sun Studio 12: Performance analyzer," September 2007.

[13] Advanced Micro Devices, "NUMA aware heap memory manager," *White Paper*, 2009.

[14] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, 2000, pp. 117–128.

[15] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 43–57.

[16] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harrisa, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system," in *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, June 2008.

[17] S. Thibault, F. Broquedis, B. Goglin, R. Namyst, and P.-A. Wacrenier, "An efficient OpenMP runtime system for hierarchical architectures," *A Practical Programming Model for the Multi-Core Era*, pp. 161–172, 2008.

[18] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner, "Extending OpenMP for NUMA machines," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000, p. 48.

[19] D. S. Nikolopoulos, E. Ayguadé, and C. D. Polychronopoulos, "Runtime vs. manual data distribution for architecture-agnostic shared-memory programming models," *International Journal of Parallel Programming*, vol. 30, no. 4, pp. 225–255, 2002.

[20] M. M. Tikir and J. K. Hollingsworth, "NUMA-aware Java heaps for server applications," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, vol. 1, 2005, p. 108b.

[21] Silicon Graphics, "Local and remote memory: Memory in a Linux/NUMA system," *White Paper*, 2006. [Online]. Available: http://www.kernel.org/pub/linux/kernel/people/christoph/pmig/numamemory.pdf

[22] R. C. Kunz, "Performance bottlenecks on large-scale shared-memory multiprocessors," *Ph.D. Thesis, Stanford University*, December 2004.