HPC Runtime Support for Fast and Power Efficient Locking and Synchronization

Conference Paper · September 2013

DOI: 10.1109/CLUSTER.2013.6702659

HPC Runtime Support for Fast and Power Efficient Locking and Synchronization

Hakan Akkan*, Michael Lang†, Latchesar Ionkov†
*New Mexico Consortium, †Los Alamos National Laboratory

***Abstract*—As compute nodes increase in parallelism, existing intra-node locking and synchronization primitives need to be scalable, fast, and power efficient. Most parallel runtime systems try to find a balance between these properties during synchronization by fine-tuned spin-waiting and processor yielding to the OS. Unfortunately, the code path followed by the OS to put the processor into a lower power state for idling almost always includes the interrupt processing path. This introduces an unnecessary overhead for both the waiting tasks and the task waking them up. In this work we investigate a pair of x86 specific instructions, `MONITOR` and `MWAIT` , that can be used to build these primitives with the desired performance and power efficiency properties. This pair of instructions allow a processor to quickly pause execution until another one wakes it up with single memory store avoiding the overhead of switching to the idle thread of the OS for the waiting task, and sending IPIs for the waking task. We implement a locking primitive using these instructions and evaluate its effectiveness in OpenMP on low to high scales. In these tests we have seen very good scaling and performance improvements of up to 23x and 6x power reduction at 64 cores. With these results as a motivation we propose that other high-core count processors include these type of instructions and make them available to user-space applications.**

I. Introduction

Efficient intra-node parallelism is becoming more important with supercomputers getting close to exascale, and nodes expected to contain thousands of cores[1]. Increased number of execution threads makes the concurrent access to shared resources crucial to the overall application performance. Support for fine-grained management of these, with minimal overhead and in a energy efficient manner is a requirement for moving forward. Many of the mechanisms for ensuring the correct access to shared resources are implemented on top of few primitives, like locks, or barriers. The implementations that are used currently don't scale very well and using them with thousands of cores will cause degraded application performance. Also, power is one of the most important constraints at exascale. As most of the synchronization implementations leverage short-time spinning before they yield the execution, with thousands of cores, a lot of energy will be wasted during the synchronization stages.

The environment in which the HPC applications run is very different than general computing environments. In most cases, HPC applications utilize one execution thread per available core. The threads are usually pinned to the cores to minimize the penalties of cache misses, NUMA domain crossings, etc. The nodes are reserved for the application, there are few to none system services or other processes using the node while the application is running. For that reason, when a thread waits on a lock, there is nothing else available to run on the core, and the core is idle until the lock is acquired.

The goal of our project is to evaluate the available mechanisms for synchronization and try to find what is the implementation that best optimizes both the performance and power consumption for HPC environments.

978-1-4799-0898-1/13/$31 © 2013 IEEE

We focused on the x86 architecture, because it is the most widespread architecture for HPC. We compared the performance and power usage of the most popular implementations for locks, both in the kernel and user space, and compared it with some less popular implementations, as well as our own algorithms. We focused the `MWAIT/MONITOR` pair instructions, available as part of the SSE3 extension of the x86 instruction set, because of their energy saving potential. The Linux kernel already uses these instructions to idle a core when nothing is available to run on it, but because of the different environment HPC applications run in, we believe it is feasible to use `MWAIT/MONITOR` in user space and save a context switch, required currently for contended locks and to reduce the energy wasted in spin-waits.

There are two popular schemes to implement locks in user-space. In the first one, the thread tries to acquire the lock, if it fails it yields its execution to the OS until the lock becomes available. The drawback of this approach is the penalty involved with system call execution – context switching, and complex logic run in kernel mode in addition to the overhead of sending interprocessor interrupts (IPI) to wake up the sleeping core. In the second scheme, known as spinlock, the threads loop in user-space, trying to acquire the lock until they succeed. The drawback of this approach is that the CPU cores use a lot of energy while spinning to acquire the lock. Many of the locks implement hybrid approach, spinning for some time before yielding the execution to the OS. Barriers as well employ a similar logic where tasks spin for a short period of time and then yield the execution to the OS while waiting for other tasks to arrive at the barrier.

The main contributions of our work are as follows; 1) optimizing performance (speed and energy) of intra-node synchronization for the HPC environment. We show that using the `MWAIT/MONITOR` instructions for synchronization improves drastically both speed and power usage as the number of cores per node increases. We show this with our 1spinner implementation, modification of the MCS lock, and a 2-level barrier implementation which will be described in detail later in this paper. 2) Our results can be used as motivation to make these instructions available in user-space for all x86 processors (they are privileged instructions on Intel processors) and to make these type of instructions available on other non-x86 architectures. Finally, 3) replacing the standard synchronization primitives in OpenMP, shows how this work can transparently improve the performance of parallel tasks, making it more feasible for developers to implement finer-grained tasks in their applications.

The remainder of the paper is arranged as follows; section II describes the related work, section III explains the current state of art in the implementations of synchronization primitives and our versions using `MWAIT/MONITOR` ; section IV reports the performance and energy efficiency of these primitives under microbenchmarks and as they perform when integrated into OpenMP; and finally in section V we have the conclusions and future work.

II. Related Work

In [10] the authors present a survey of common locking mechanisms and compare their performance. Boyd-Wiziker et al. [2] compare Linux ticket lock implementation to some more scalable alternatives, like MCS lock, CLH lock, etc.

A futex [4] is a common locking mechanism used in Linux, which leverages shared memory regions and atomic operations to manage locks in users space without context switches in the uncontented case. A thread usually spins on a futex hoping for an easy acquisition, then times out and sleeps before trying again. Signaling this sleeping task is achieved by sending IPIs across CPUs that introduces the overhead of the complex interrupt processing machinery in the kernel. Performance of the futex (used by the pthread and OpenMP locks) is reported in the investigation.

Comparison of the energy usage of different spinlock implementations in relation to ARM processors in a system-on-a-chip platform is presented in [5]. Here the authors suggest that a hardware spin lock implementation provides the best energy and performance, but their investigation is focused on small scale parallelism as with most SOC investigations.

In [12] the authors investigate locks for multi-threaded multi-core architectures, they compare methods on Power architecture and Intel processors. Much like our investigation they are looking at scalability and performance. They look at Intel's `MWAIT/MONITOR` and Power architecture's *wrlos* – allows threads to wait asynchronously without polling, but do not investigate the energy aspect of locks.

There is some research, especially in for embedded systems, for optimizing synchronization mechanisms for saving energy. Most of them, for example C-Locks [7] and thrifty barriers [8], require additional hardware.

RCL [9] uses a reserved core to act as a server for a single lock, the lock data is then local to the core, hopefully in cache. This lock is then treated as an remote procedure call (RPC), this requires changes to the source code of the applications. The authors of that work are also aware of /mm instruction pointing out that it will save energy, but they do not provide results.

Another method for synchronizing tasks on shared memory machines involves transactional memory (TM). TM is an optimistic method, where the tasks use the data and then roll back if there are conflicts detected. Using the transactional model requires keeping track of information to support the rollbacks which can be open ended. Some studies [11] show that TM may reduce the energy consumption compared to traditional locks.

Of the investigations, none cover the scale of parallelism or the HPC workload which is the focus of our work.

III. Synchronization Primitives with `MWAIT/MONITOR`

In this section we describe techniques used to leverage the `MWAIT/MONITOR` pair of x86 instructions for implementing mutual exclusion and barrier primitives. All of the different synchronization primitives we investigate in this work depend on availability of one or more of atomic operations in the processor, which are supported by most of the modern processors. On x86 platforms, atomic operations work by using the `LOCK` prefix with individual instructions that modify a memory word, which instantaneously locks the memory bus so that no other CPU can access the same memory word during the execution of that instruction. Consequently, atomic operations are much more expensive compared to their non-atomic counterparts. Besides, synchronization among the tasks on NUMA machines requires remote loads/stores, which adds another layer of overhead to the operation of the primitive. Thus, synchronization algorithms strive for reducing the number of atomic operations and cross NUMA-node accesses.

The `MWAIT/MONITOR` instructions leveraged in this work allow a core to go into a sleep state and wait on an address. The core is then to be woken up when another core modifies the address that this core is waiting on. This enables energy savings while the core sleeps rather then busy spinning and wasting power and overwhelming memory buses. In addition, hyper-threaded cores release their shared resources to sibling cores while sleeping, which increases the efficiency of the sibling core.

A. Mutual exclusion primitives

There are two general categories of locks: queuing (fair) and non-queuing (non-fair) locks. In non-queuing locks, no order is kept of the tasks acquiring the lock. Once the lock becomes available (released by the current owner), all tasks requesting the lock make an attempt to acquire the lock at the same time with one randomly succeeding. This method causes significant stress on memory buses as the cache coherency protocol is required to keep the same memory region consistent across all cores of all the tasks participating in the lock. This does not scale and additionally it frequently results in starvation. Therefore, this method does not have wide usage. In queuing locks, tasks are queued in the order they attempt to acquire the lock and are woken up only by their predecessors once the lock is ready to be acquired again, i.e. the predecessor hands the lock to its successor. This results in a fair rotation of the lock among tasks which eliminates starvation.

1) Linux FIFO ticket spinlocks: The FIFO ticket spinlock was introduced into Linux kernel in 2008 when it was discovered that the non-queuing lock implementation being used at that time caused significant unfairness and starvation resulting in the kernel exhibiting non-deterministic latency into time-sensitive operations [13]. The ticket lock shown in Algorithm 1[1] is a two-part data structure that represents the head and tail of the lock. Tasks atomically increment and read the tail value, which denotes their position in the queue. The lock is acquired by a task when the head of the lock becomes equal to the tail value that that task read. The head value is incremented by the task that is releasing the lock.

Algorithm 1 Linux ticket lock

```
procedure INITIALIZE($lock$)
    $lock.head \leftarrow 0$
    $lock.tail \leftarrow 0$
end procedure
procedure ACQUIRE($lock$)
    $position \leftarrow atomic\_fetch\_and\_inc(\&lock.tail)$
    repeat
        $head \leftarrow lock.head$
    until $head = position$
end procedure
procedure RELEASE($lock$)
    $lock.head \leftarrow lock.head + 1$
end procedure
```

[1]Actual implementation uses a double-word and reads and modifies it atomically with a single instruction. We present the underlying algorithm without this detail for brevity.

This algorithm allows tasks to be released in the FIFO order and prevents starvation. However, all tasks are required to poll the same word and this overwhelms the memory buses under high contention. Thus, this lock implementation does not scale well. On the other hand it still is a suitable implementation to be used in the Linux kernel as spinlocks are almost always held for very short periods and generally not highly contented, which does not warrant the use of more complicated locks such as an MCS lock or our 1SPINNER lock.

2) MCS lock: The MCS lock [10] maintains a queue of tasks. The queue is implemented by chaining together a list of the lock requestors together. When a new task wants to join the queue, it atomically adds itself to its predecessor. If the task is not at the head of the queue it spins waiting for the predecessor to trigger it, thereby eventually moving it to the front of the queue.

3) 1spinner lock: The motivation behind the design of our novel 1spinner lock is preventing all tasks from polling on the same memory object, which will reduce the amount of cache thrashing, provide better scalability on high core count nodes and increase energy efficiency. Our design makes use of *thread-local* or *per-cpu* data regions to indicate the status of each thread/CPU. The lock object itself is a 3-part data structure denoting 1) the current holder of the lock, 2) current tail of the waiting queue, and 3) a generation number set by the current tail of the lock. The pseudo-code of the 1-spinner lock is shown in Algorithm 2.

In the 1spinner algorithm each task brings the lock data into its cache when they do the exchange and then never access it again until their predecessor releases them, thus reducing the amount of cache thrashing. There can only be one task that is polling the `lock.head` at any one given time.

Unlike MCS, 1spinner doesn't require the current holder to know, or wait for, its successor. Thus, releasing the lock is as simple as changing the head of the lock to the sentinel value. One other advantage of 1spinner over MCS is that the lock data structure itself is much smaller (4-8 bytes vs 8-16 bytes).

We assume threads/CPUs can access each others local data regions, which is necessary for them to check their predecessor's state. We use a global array called `monitors` to store the pointers thread-local data regions indexed by the thread/CPU number.

The power efficient version of this lock replaces the first loop with `MWAIT/MONITOR` pair waiting on the predecessor's monitor. Whenever a task acquires the lock, it wakes up its successor by modifying its own monitor and the successor starts polling on the `lock.head`. This allows tasks to avoid the overhead of `MWAIT/MONITOR` logic when the lock becomes available.

B. Barriers with `MWAIT/MONITOR`

Barrier primitive is used when all of the tasks in a parallel program are expected to come to a common state before any of them executes further. A simple algorithm to implement a barrier is to allow all tasks in the parallel program to atomically decrement a shared counter and wait until its value is equal to zero [6], at which point the counter is reinitialized with the number of tasks in the program by the last task that entered the barrier and tasks continue execution. GNU OpenMP implements the same barrier and leverages the futexes in the Linux kernel to idle the core after a short period of polling. However, as we pointed out before, waking up from futex involves IPIs and introduces significant overhead for both the waiting task and signaling task. Besides, since there is only one shared counter, the atomic modification of it by all of the tasks is a bottleneck overwhelming the memory buses in the machine.

Algorithm 2 1spinner lock

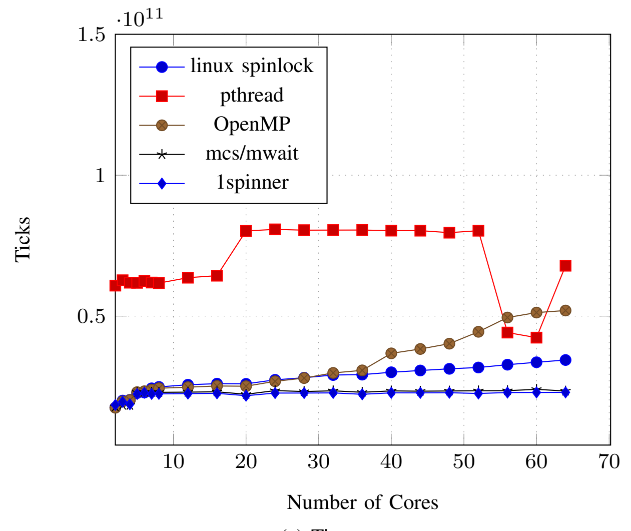
```
struct lock {
    u16 $head$
    struct {
        u8 $task$
        u8 $generation$
    } $tail$
}
procedure INITIALIZE($lock$)
    $lock.head \leftarrow \emptyset$
    $lock.tail \leftarrow \emptyset$
end procedure
procedure ACQUIRE($lock$)
    $id \leftarrow per\_cpu.id$
    $gen \leftarrow monitors[id]$
    $\{prev,\ prev\_gen\} \leftarrow xchg(\&lock.tail, \{id, gen\})$
    if $prev \neq \emptyset$ and $prev \neq id$ then
        while $monitors[prev] = prev\_gen$ do
        end while                                  ▷ Empty loop body
        while $lock.head = prev$ do
        end while                                  ▷ Empty loop body
    end if
    $lock.head \leftarrow id$
    $monitors[id] \leftarrow monitors[id] + 1$
end procedure
procedure RELEASE($lock$)
    $lock.head \leftarrow \emptyset$
end procedure
```

In our implementation, there is a barrier per NUMA node for the tasks running on the same node as well as a global barrier that is used by the final tasks of each per-node barrier. Each task enters the NUMA-local barrier first and, if it is not the final task, starts waiting on the global barrier's wait monitor. The final tasks in per-node barriers reset their local counters first and enter the global barrier. The last task to enter the global barrier resets the counter of the global barrier and notifies every other task by updating the global wait monitor. Since tasks use the `MONITOR` instruction to sleep rather than polling the global wait variable, this implementation eliminates the bottleneck of high counts of simultaneous reads on the same cache line. Besides, the atomic operations are performed on per-node variables rather than a single global variable, which reduces the bottleneck of high counts of simultaneous atomic decrements and scales better.

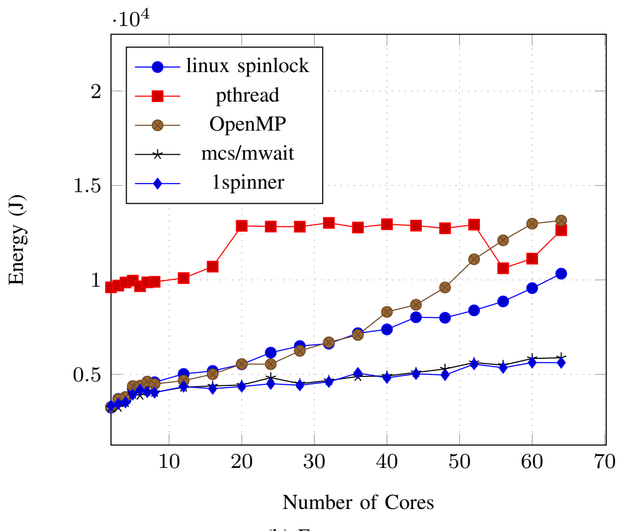
Implementing this 2-level barrier without `MWAIT/MONITOR` is not beneficial because all tasks would eventually start polling on the global wait variable, which is not an improvement over the traditional one-level barrier. In other words, the advantage of our barrier is that if the tasks were polling rather than using `MWAIT` , they would be overwhelming the buses as they will all be polling on the same cache line. With `MWAIT` , they still wait on a single line but in an idle state rather than polling. When the value of the variable changes with a single store, all tasks are woken up and the barrier is complete.

IV. Results

In this section we present the results of our modified lock and barrier implementation with various micro-benchmarks and compare them to the performance and energy use of existing implementations. The focus is on the scaling of

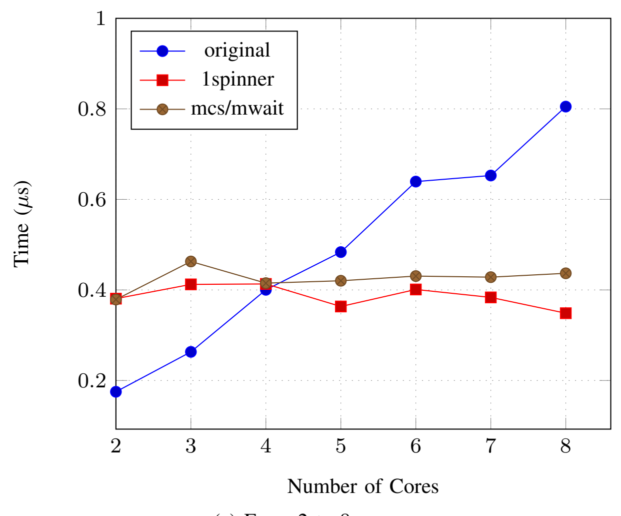


(a) Time

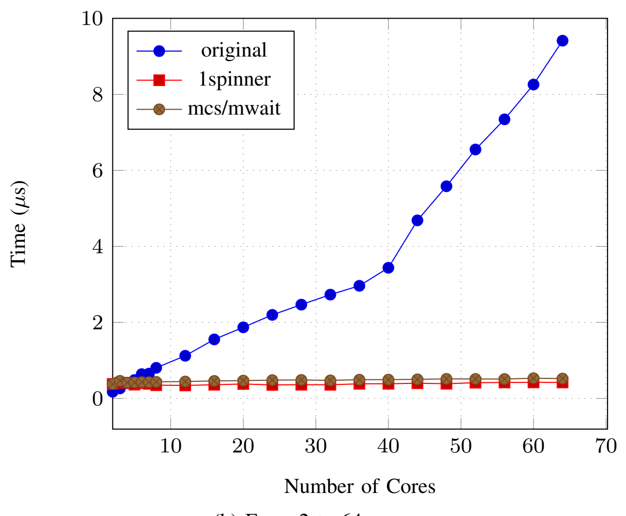


(b) Energy

Fig. 1: Lock/Unlock Performance



(a) From 2 to 8 cores



(b) From 2 to 64 cores

Fig. 2: Syncbench Lock/Unlock Results

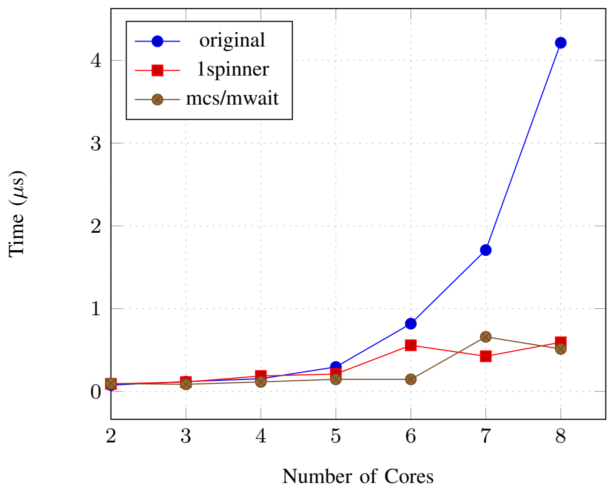
number of cores and energy used, as we expect high core counts on future processors where energy efficiency is a high priority. One limitation of the solution is that due to the `MWAIT/MONITOR` instructions affecting all threads and processes of a particular core, these locks are only available at a core granularity rather than a thread granularity. Therefore the experiments are conducted with one thread per core participating in the locks and barriers.

The experimental setup included a 64-core server node with 4 AMD Opteron 6272 CPUs. Power was measured with a Yokogawa 210 power meter attached to the main power supply of the node. The power meter was set up to collect average power values when triggered, these readings were combined with the measured time to produce energy values. The node was running a standard 3.5.0 Linux kernel and all processes were pinned to a core so they don't move to another core while running the tests.

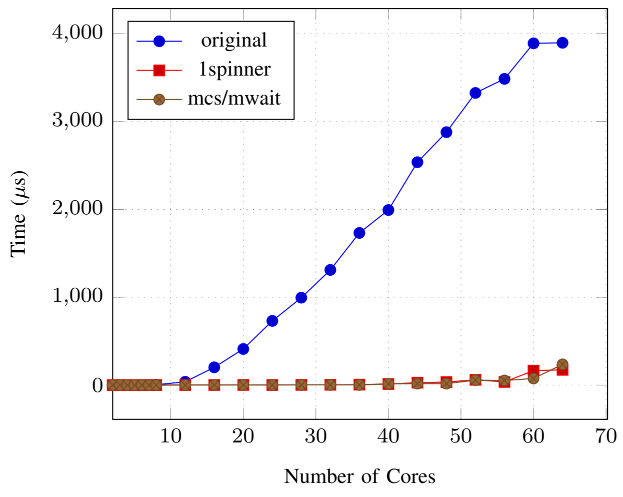
A. *Locking Primitives*

In the first set of experiments, we compare the performance of three standard lock implementations (pthread mutex, OpenMP, and linux spinlock) to our lock implementations that use `MWAIT/MONITOR` (mcs and 1spinner). We used a custom micro-benchmark that performs equal amount of work inside and outside of a critical section, protected by a lock. Figure 1-a shows the number of cores participating in the locks increasing on the x-axis and the time on the y-axis. Figure 1-b shows the cores increasing on the x-axis with the energy used on the y-axis. Our implementations scale much better than all standard implementations, with as much as 4 times less overhead. With this increase in performance we also reduce energy by up to 3.5 times.

To be transparently applicable to many applications we need to integrate our locks into application run-times. We chose to start with OpenMP as it is very mature standard and is widely used. We integrated our MCS/MWAIT and

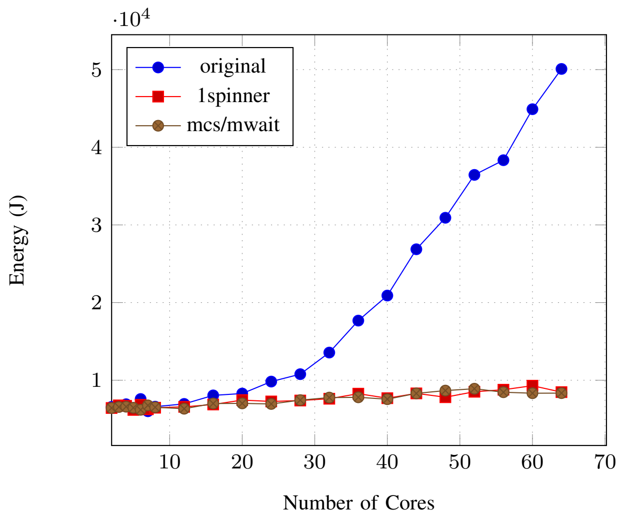


(a) From 2 to 8 cores

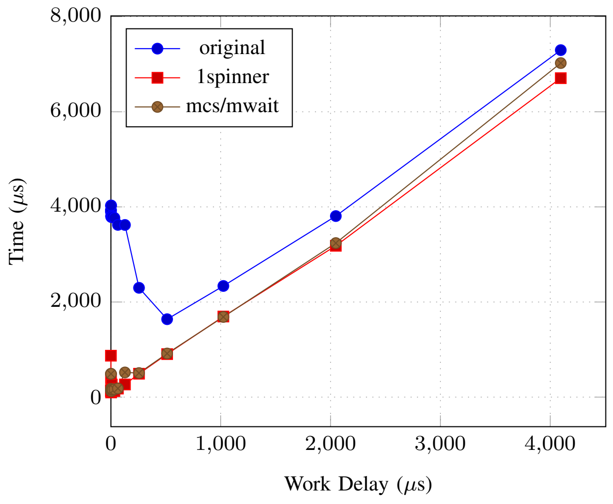


(b) From 2 to 64 cores

Fig. 3: Taskbench Parallel Task Results



(a) Taskbench Energy



(b) Taskbench Parallel Tasks Scaling Amount of Work

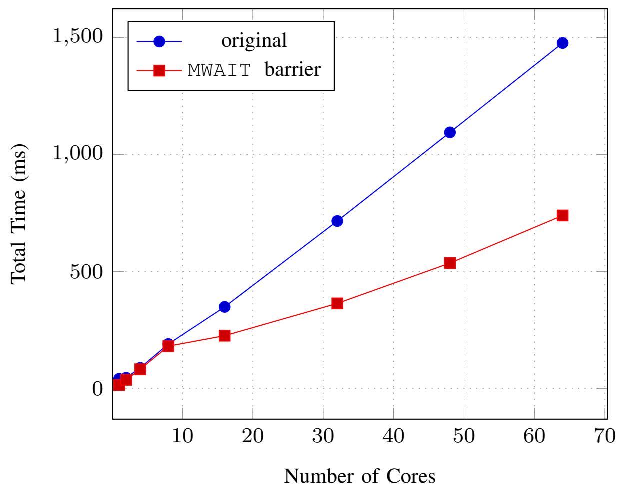
Fig. 4: Taskbench – Energy and Workscaling

1SPINNER implementations into OpenMP (GNU libgomp implementation included in gcc 4.7.3) and compared it with the standard OpenMP lock using version 3 of the EPCC's OpenMP Microbenchmarks suite[3]. Integration of our implementation into OpenMP was not invasive and required only a small amount of changes. We enabled the selection of the locking mechanism via environment variables for ease of testing.

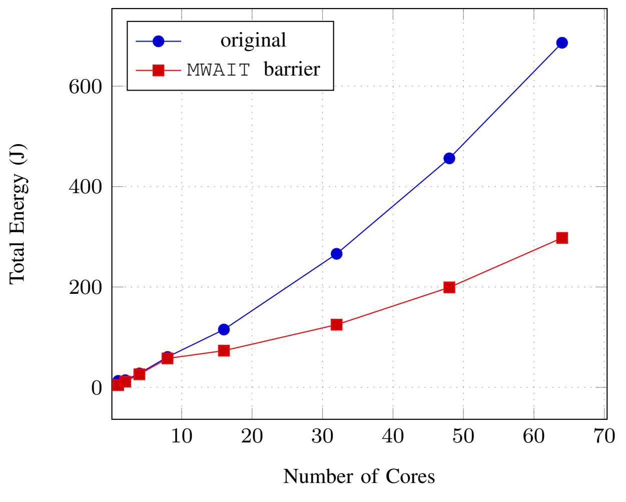
As in the first set of tests, we investigated how the implementations scale as the number of cores used by the benchmark increases. Figure 2 shows the LOCK/UNLOCK results from the syncbench benchmark where each parallel block acquired a lock, does work for specified time (the default for openmpbench 0.1 μs), then releases the lock. For low core counts, the original OpenMP implementation performs better, but as the number of cores increases, it is outperformed by both MCS/MWAIT and 1SPINNER, with 1SPINNER being 22 times faster for 64 cores. Figure 3 shows the results for the TASK PARALLEL part of the taskbench test. In this test the benchmark spawns parallel tasks using the *omp parallel* pragma, which performs some work in parallel. With comparable performance for low core count, the MCS/MWAIT and 1SPINNER implementations strongly outperform the original implementation as the number of cores increases, up to 23 times for 64 cores.

Figure 4a shows the energy consumed by the computer while running the taskbench test. The 1SPINNER implementation uses 6 times less energy than the original implementation.

With the third set of experiments we tested the relationship between the performance gain compared to the amount work done in a parallel task. In this test we always use all 64 cores and vary the amount of work from 0.1 μs to 4000 μs. The total test time we used was 10000 μs, so the benchmark split the execution in 100000 to 3 parallel tasks. Figure 4b shows that even though the improvements using the `MWAIT/MONITOR` locks drops with the increase of the amount of work, our implementations still outperform the original lock. Because our implementations overhead is so

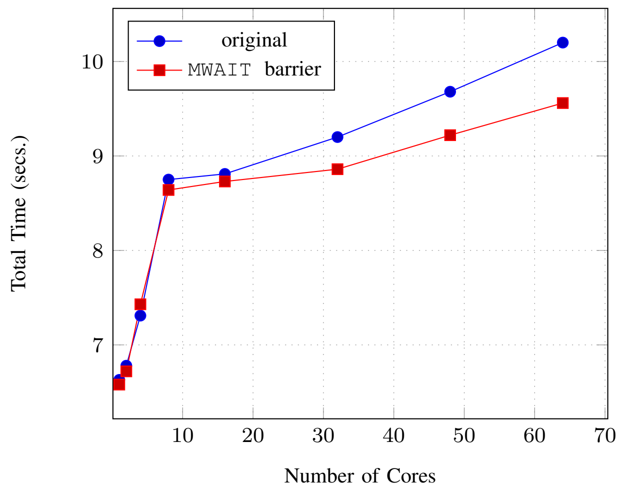


(a) Time

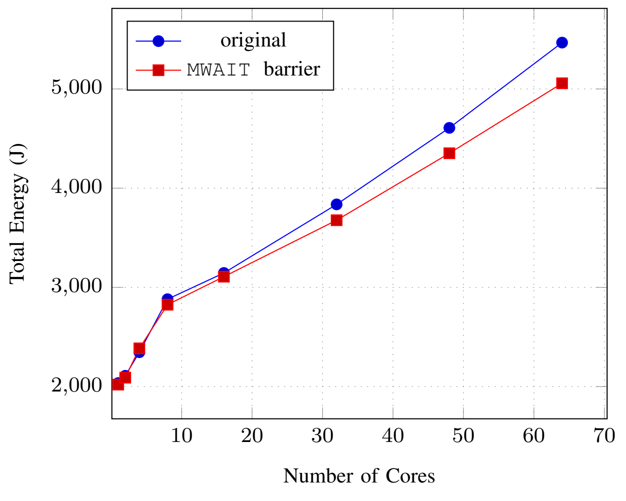


(b) Energy

Fig. 5: Completion of 100000 barriers with 1μs unit of work



(a) Time



(b) Energy

Fig. 6: Completion of 100000 barriers with 250μs unit of work

much lower when the amount of work is small, it may allow developers to split tasks that they keep large because of the high overhead of the standard implementation and improve the application performance by using more independent tasks.

B. Barrier Primitives

We compared the performance and power efficiency of the 2-level barrier with `MWAIT/MONITOR` to the stock barriers in the GNU OpenMP implementation using microbenchmarks. In our tests, all tasks perform a short amount of work varying between 1 μs to 250 μs and enter the barrier. We measure and report the total time it took for task #0 to complete execution of 100000 iterations of this work&barrier sequence.

Figure 5 shows that the 2-level barriers with `MWAIT/MONITOR` scale much better than the single-level barriers with futex. In this test, the work performed by the tasks between iterations is very short, which emphasizes the performance of the barrier primitive. As can be seen in the time figure, the overhead of the 2-level barriers is almost the half of the stock barriers in GNU OpenMP implementation. The lower energy metric is implied by the earlier completion of the entire program.

When the amount of work is increased up to 250μs, the overhead of the barrier primitive becomes obscured (6) but the 2-level barrier with `MWAIT/MONITOR` still outperforms the stock barriers.

Note that the two implementations have the identical performance and power metrics in low scales as the 2-level barrier implementation is only enabled when the participating tasks occupy more than one NUMA domain.

V. Conclusions

Due to the increasing parallelism that is unavoidable in today's processors, applications will have to exploit all available parallelism to efficiently use the compute resources. To enable this shared memory locks will have to be highly tuned and power efficient for large numbers of lock participants. In this work we have show the utility of the `MWAIT` and `MONITOR` instructions for reducing power usage on highly contended locks. Our solution modifies the run-times such that applications get the performance and power efficiency without extra work or modifications. A variety of locks have been evaluated for both performance and energy requirements as the number of processes or threads contending for the locks increases. Our results show substantial benefits in scaling, up to 23x and benefits in energy efficiency, up to 6x reduction at 64 cores. Our implementations' overhead is so much lower than traditional locking when the amount of work is small, it may allow developers to split tasks that they have previously kept together, and improve the application performance by using more independent tasks. Even with the restriction allowing only one thread per core to participate in locks, we feel that the performance and energy savings are evident. Furthermore, if the number of cores increases as is expected to upwards of 1000 cores per processor this work will become increasingly important.

Future Work: We would like to broaden the types of locks and synchronization primitives available to run-times, because `MWAIT` and `MONITOR` enables more options where spin-waiting is more expensive. Since performance was not as good at very low core counts we could combine the locking methods choosing the original method at low cores counts and transition to our method at higher core counts. We would also like to try to extend this work by investigating the effects of different C-state waits, which are configurable on some Intel processors. Further we plan to integrate these primitives into other run-times, such as some of the adaptive run-times that are becoming available.

VI. Acknowledgments

This work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract DE-FC02-06ER25750. The publication has been assigned the LANL identifier LA-UR-13-26575.

References

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor and study lead, 2008.

[2] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2012.

[3] J. M. Bull and D. O'Neill. A microbenchmark suite for openmp 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48, Dec. 2001.

[4] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, page 85. AUUG, Inc., 2002.

[5] O. Golubeva, M. Loghi, and M. Poncino. On the energy efficiency of synchronization primitives for shared-memory single-chip multiprocessors. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, GLSVLSI '07, pages 489–492, New York, NY, USA, 2007. ACM.

[6] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, Feb. 1988.

[7] S. Kim, S. Lee, M. Jun, B. Lee, W. Ro, E. Chung, and J. Gaudiot. C-lock: Energy efficient synchronization for embedded multicore systems. *Computers, IEEE Transactions on*, PP(99):1–1, 2013.

[8] J. Li, J. Martinez, and M. Huang. The thrifty barrier: energy-aware synchronization in shared-memory multiprocessors. In *Software, IEE Proceedings-*, pages 14–23, 2004.

[9] J.-P. Lozi, F. David, G. Thomas, J. Lawall, G. Muller, et al. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proc. Usenix Annual Technical Conf*, pages 65–76, 2012.

[10] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[11] T. Moreshet, R. Bahar, and M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, pages 331–334, 2005.

[12] D. Pasetto, M. Meneghin, H. Franke, F. Petrini, and J. Xenidis. Performance evaluation of interthread communicationmechanisms on multicore/multithreaded architectures. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 131–132, New York, NY, USA, 2012. ACM.

[13] N. Piggin. x86: Fifo ticket spinlocks, 2008.