Performance evaluation of inter-thread communication mechanisms on multicore/multithreaded architectures

Massimiliano Meneghin
IBM Research Lab
Dublin, Ireland
massimin@ie.ibm.com

Davide Pasetto
IBM Research Lab
Dublin, Ireland
pasetto_davide@ie.ibm.com

Hubertus Franke
IBM TJ Watson
Yorktown Heights, NY
frankeh@us.ibm.com

Fabrizio Petrini
IBM TJ Watson
Yorktown Heights, NY
fpetrin@us.ibm.com

Jimi Xenidis
IBM Research Lab
Austin, TX
jimix@us.ibm.com

ABSTRACT

The three major solutions for increasing the nominal performance of a CPU are: multiplying the number of cores per socket, expanding the embedded cache memories and use multi-threading to reduce the impact of the deep memory hierarchy. System with tens or hundreds of hardware threads, all sharing a cache coherent UMA or NUMA memory space, are today the de-facto standard. While these solutions can easily provide benefits in a multi-program environment, they require recoding of applications to leverage the available parallelism. Application threads must synchronize and exchange data, and the overall performance is heavily influenced by the overhead added by these mechanisms, especially as developers try to exploit finer grain parallelism to be able to use all available resources.

This paper examines two fundamental synchronization mechanisms - locks and queues - in the context of multi and many cores systems with tens of hardware threads. Locks are typically used in non streaming environments to synchronize access to shared data structures, while queues are mainly used as a support for streaming computational models. The analysis examines how the algorithmic aspect of the implementation, the interaction with the operating system and the availability of supporting machine language mechanism contribute to the overall performance. Experiments are run on Intel X86^{TM} and IBM PowerEN^{TM}, a novel highly multi-threaded user-space oriented solution, and focus on fine grain parallelism - where the work performed on each data item requires only a handful of microseconds. The results presented here constitute both a selection tool for software developer and a datapoint for CPU architects.

General Terms

Performance,Experimentation

1. INTRODUCTION

In the past, advances in transistor count and fabrication technology have led to increased performance, typically proportional to improvements in clock rates. However, this trend has slowed due to limitations arising from power consumption, design complexity, and wire delays. In response, designers have turned to multi-core and multi-thread configurations that incorporate several cores on one or more dies. While multiple cores can readily support throughput applications, such as web servers or map-reduce searches that are embarrassingly parallel, threaded applications that operate on a shared address space to complete a unique task demand efficient synchronization and communication mechanisms. Efficient and low overhead core-to-core communication is critical for many solution domains with countless examples in network computing, business analytics, financial markets, biology, and high-performance computing in general. As the number of cores increases, the *desired grain of parallelism becomes smaller* and understanding the overhead and tradeoff of the core-to-core communication mechanism is becoming increasingly important.

Most threaded applications have concurrent needs to access resources that can only be shared with a logical sequential consistency model. The way these contentions are resolved directly affects the system's timeliness properties. Several mechanisms are available today, and they can broadly be classified into: (1) lock-based schemes and (2) non-blocking schemes including wait-free protocols [16] and lock-free protocols [12] [2]. Lock-based protocols, typically used in multi-threaded applications that do not follow a stream computing model, serialize accesses to shared objects by using mutual exclusion, resulting in reduced concurrency [4]. Many lock-based protocols typically incur additional run-time overhead due to scheduler activations that occur when activities request locked objects.

Concurrent lock-free queues for inter-thread communication have been widely studied in literature since they are the basic building block of stream computing solutions. These algorithms are normally based on atomic operations and modern processors provide all the necessary hardware primitives such as atomic compare-and-set (CAS) and load-linked store-conditional (LL/SC). All these primitives implicitly in-

troduce synchronization at the hardware level that are often an order of magnitude slower, even for uncontested cache aligned and resident words, than primitive stores. With the exception of Lamport's queue [13], the focus of prior art has been on multiple producer and/or multiple-consumer (MP/MC) queue variants. These general purpose queues to date have been limited in performance due to high overheads of their implementations. Additionally, general purpose MP/MC variants often use linked lists which require indirection, exhibit poor cache-locality, and require additional synchronization under weak consistency models [15].

While an extensive amount of work has been performed on locks and queues, being these fundamental building blocks for threaded applications, a comprehensive comparison of their runtime performance characteristics and scalability on modern architectures is still missing. This paper wants to address the following open questions:

- Can modern CPU architectures effectively execute *fine grain* parallel programs that utilize all available hardware resources in a coordinated way?
- What is the overhead and the scalability of the supporting synchronization mechanisms?
- Which synchronization algorithm and Instruction Set Architecture level support is the best for fine grain parallel programs?

The paper contains an evaluation of different types of locks and queues on large multi-core multi-threaded systems and focuses on fine grain parallelism, where the amount of work performed on each "data item" is in the order of microseconds. The implementations cover a range of different solutions, from Operating System to full user-space based, and look at their behaviour as the load increases. Moreover a new Instruction Set Architecture solution for low overhead user-space memory waiting is presented and its usage is evaluated both from a performance and overhead reduction point of view. The next section describes the two test systems and briefly details the hardware synchronization mechanisms available in their CPU cores. Section 3 briefly describes the locking algorithms considered while section 4 examines their runtime behavior; the following section 5 details the queueing strategies implemented; these are then evaluated in section 6. Section 7 contains some concluding remarks.

System	Nehalem	PowerEN
Sockets	2	1
Cores	6	16
Threads per core	2	4
Total threads	24	64

Table 1: Systems under test.

2. TARGET PLATFORMS

This paper examines the performance of synchronization mechanisms on two very different computer architectures: Intel Nehalem-EP^{TM} and IBM PowerEN^{TM}. Their general characteristics are summarized in Table 1 and both systems run Linux Operating System.

The Intel Xeon^{TM} 5570 (Nehalem-EP) is a 45nm quad core processor whose high level overview is shown in figure 1. Designed for general purpose processing, each core has a private L1 and L2 cache, while the L3 cache is shared across the cores on a socket. Each core supports Simultaneous Multi Threading (SMT), allowing two threads to share processing resources in parallel on a single core. Nehalem EP has a 32KB L1, 256KB L2 and a 8MB L3 cache. As opposed to older processor configurations, this architecture implements an inclusive last level cache. Each cache line contains 'core valid bits' that specify the state of the cache line across the processor. A set bit corresponding to a core indicates that the core may contain a copy of this line. When a core requests for a cache line that is contained in the L3 cache, the bits specify which cores to snoop for the latest copy of this line, thus reducing the snoop traffic. The MESIF [10] cache-coherence protocol extends the native MESI [7] protocol to include 'forwarding'. This feature enables forwarding of unmodified data that is shared by two cores to a third one.

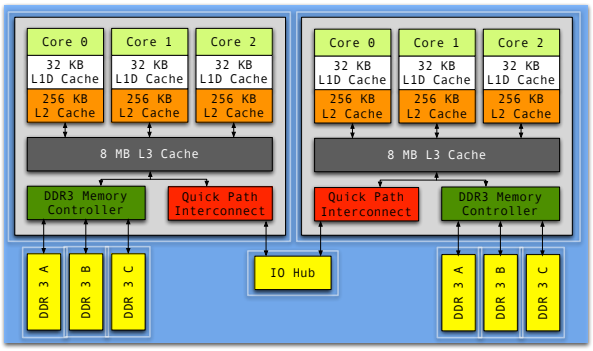

Figure 1: High-Level Overview of Intel Nehalem^{TM}.

Atomic operations on the Intel architecture are implemented using a LOCK prefix: the `lock` instruction can be prefixed to a number of operations and has the effect to lock the system bus (sometimes only the local cache in recent architectures) to ensure exclusive access to the shared resource. In 2003, Intel first introduced, as part of the SSE3 instruction set extension, the MONITOR and MWAIT instructions. The MONITOR operation sets up an address range that is monitored by the hardware for special events to occur, and MWAIT waits for that event or for a general interrupt to happen. One possible use is the monitoring of store events: a spin lock's wait operation can be implement by arming the MONITOR facility and executing the MWAIT instruction, which puts the hardware thread into an "implementation optimized state", which generally implies this hardware thread does not dispatch further instructions. When the thread holding the critical section stores to the lock variable releasing the lock, the waiting hardware thread will be woken up. The drawback of the MONITOR/MWAIT instructions is that they are privileged in the X86 Instruction Set Architecture and thus cannot be executed by application software. As a result an application writer still would have to utilize costly system calls to perform these operations. Finally the

Intel X86 architecture provides a specific instruction (called `pause`) meant to be used inside busy waiting loop to "wait for a CPU specific amount of time" to reduce memory and instruction scheduling pressure. The amount of time actually waited, and how this is implemented, depends on the processor family and it is not disclosed.

The IBM PowerEN^{TM} "Edge of Network" processor (also known as WireSpeedProcessor) [9][11] was recently introduced by IBM and integrates technologies from both network and server processors to optimise network facing applications where latency and throughput are the primary design targets. PowerEN, whose high level view is shown in figure 2, is a system on a chip (SoC) consisting of 16 embedded 2.3GHz 64-bit PowerPC cores and a set of acceleration units. Each CPU core consists of four concurrent hardware threads that feed a dual issue in-order pipeline. Each core includes a 16KB L1 data cache and a 16KB L1 instruction cache with 64 byte cache lines. Each group of four cores shares a sliced 2MB L2 cache for total of 8MB L2 cache inside the chip. Two DDR3 DRAM controllers support a memory bandwidth of up to 64GB/s and four chips can be joined together to form a 64 core (256 threads) cache coherent system.

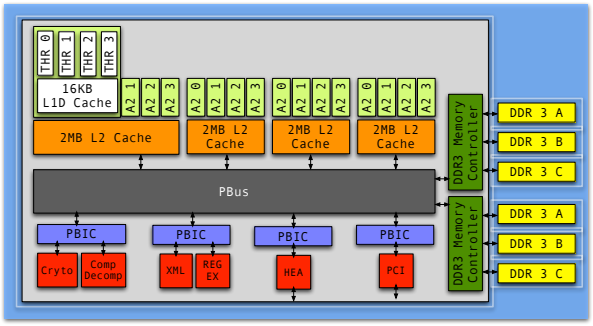

Figure 2: High-Level Overview of PowerEN^{TM}.

In contrast to the X86, PowerPC architectures utilize the load reservation / store conditional mechanism to support atomic operations: the read-modify-write semantic is realized by loading the involved memory address and simultaneously requesting a reservation on that loaded address (`lwarx` - load word with reservation). Multiple threads can do this simultaneously, but only one reservation can be active per hardware thread and it is cleared upon any machine exception, such as a context switch or interrupt. The thread performs register updates on the loaded data and, to complete the atomic operation, it conditionally stores the modified item (`stwx`). The store succeeds only if the thread is still holding the reservation. Architecturally this mechanism is implemented by the cache coherency protocol [8] and uses reservation thread information attached to each cache line. When issuing a `lwarx` the reservation for the current thread is simultaneously set with the load operation. In order to have the right to perform a store operation to a cacheline, the issuing cache must contain the data exclusively, that is no other cache contains the data. If the cacheline is not exclusively held, a request is issued on the bus that all caches snoop on which leads to the supply of cacheline from other caches if present and the invalidation of the cachelines (including the reservation) across all caches. Finally atomic operations are realized by looping until the `stwx` succeeds.

The PowerEN architecture introduces the `wrlos` instruction that allows for the thread to wait, i.e. not being dispatched, until a reservation is lost. This new element in Power ISA is in theory particularly useful in multi-threaded cores: the instruction units inside the core are shared across the various threads and execution typically proceeds in a round robin fashion. When a software thread needs to wait for a location to change value, or to obtain a specific value, it is normally forced to poll that location repeatedly (or to use operating system semaphores) consuming valuable CPU cycles. The `wrlos` instruction allows user-space and kernel-space threads to wait for this asynchronous event without consuming execution unit resources. Finally the instruction dispatching priority of the four hardware threads per core can be controlled by software, for example reducing it before a busy waiting loop and increasing it once the condition is satisfied. PowerEN provides these mechanisms at all privilege levels and hence application writers can take full advantage of them.

3. LOCKS

Two threads in the same address space (or two processes with shared memory segments) accessing common resources must synchronize their behaviour in order to avoid wrong or unpredicted behavior. The period of exclusive access is referred to as a *critical section*, which is enforced through *mutual exclusion* implying that only one thread at a time can be in this critical section of code. This paper examines 3 distinct mechanisms for mutual exclusion: binary semaphores, spinlocks and ticket locks.

Binary semaphores implement a blocking semantics: if a lock can not be acquired the thread is descheduled by the Operating System. The drawback of this approach is the overhead associated with the system call required to implement the semantics, regardless of the state of the lock. Modern systems provide hybrid versions; for instance `futexes` [3] utilize user level atomic operations to attempt to lock a mutex and only upon failure enter the kernel to implement the wait operation.

In spinlocks, competing threads will try to acquire the lock, until they succeed, entirely in userspace. Their implementation is very simple: there's a single shared variable and acquiring the lock is done by using an atomic instruction, such as `test_and_set` or `compare_and_swap`, looping until it succeeds. Freeing the lock requires a single store operation; best performance can be achieved when the shared variable has a proper CPU specific alignment and sits in a cache line of its own to avoid false shearing. The drawback of this approach is that a thread attempting to acquire a lock will run in a tight loop while the lock is held by another thread, consuming valuable CPU cycles. Traditionally, spinlocks with small hold periods were acceptable due to the low number of application threads. As the number of threads in modern systems increases, this approach might not scale when lock contention increases. Furthermore, as the number of hardware threads per core increases, the spinning threads will consume cycles of the core's instruction pipeline and potentially slow down the dispatch of other threads presumably

executing useful work. On PowerEN the `wrlos` version of the spinlock uses the hardware supported wait to be notified when the lock has been released, suspending instruction issue in the current hardware thread; its pseudo code description is shown in Figure 3.

```
void sp_init(ulong *lock)
{
  *lock = 0;
}
void sr_lock(volatile ulong *lock)
{
  int success = 0;
  success  = atomic_cas(lock, 0, 1);
  while(!success) {
    wreslos(*lock,0) ;
    success  = atomic_cas(lock, 0, 1);
  }
}
void sp_unlock(ulong *lock)
{
  *lock = 0;
  mem_sync();
}
```

Figure 3: PowerEN version of spinlock that uses wrlos instruction.

Under high contention spinlocks and Binary Semaphores are usually not "fair": it might happen that some threads very often win access to the shared resource while other threads will never be able to access it. Ticket locks[1], also known as fair locks, ensure that all threads that want to access the critical section will gain it in the exact order of arrival. This is implemented in userspace by using two counters: a serving id and an access id, both initialized to 0. To acquire the lock a thread fetches the access id, atomically incrementing it at the same time, by an atomic `fetch_and_add` instruction. The thread must then wait for the serving id to be equal to its private copy of the access id. Unlocking is done by simply incrementing the serving id. Better performance can be achieved when the shared counters have a proper CPU specific alignment and they sit in two distinct cache lines, removing false sharing. Moreover the serving id counter can be replicated and distributed, ensuring that only one thread is polling on a specific cache line, thus eliminating cache line contention in the waiting phase. A pseudocode implementation for this version of ticket lock implementation is shown in Figure 4.

The busy waiting loop can be optimized for the specific architecture by inserting a `pause` instruction on X86 to halt instruction dispatching for some time, or by lowering the thread priority or issuing the `wrlos` on IBM PowerEN.

4. LOCK PERFORMANCE

The lock performance test measures the average service time of a typical application loop: perform some computation on private data, lock, update shared data structure, unlock. Figure 5 shows the normalized average service time spent to complete these operations when only a single thread is working, thus reporting the minimum overhead encountered by a thread. The locking strategies considered on Intel X86 architecture are:

```
struct lock {
  ulong serv,acc;
};
void fp_init(lock *l)
{
  l->serv = 0;
  l->acc = 0;
}
void fp_lock(volatile lock *l)
{
  ulong myid = atomic_add(&l->acc,1);
  while(l->serv[myid % nthr] != myid) ;
}
void fp_unlock(lock *l)
{
  l->serv[myid % nthr] = myid+1;
  mem_sync();
}
```

Figure 4: Intel version of ticket lock that uses atomic add.

- pthread_lock - these are available on any architecture and Operating System. They are normally realized using futexes and the lock operation (when the lock is free) does not require any OS interaction while unlock triggers the scheduler to wake up other waiting threads.
- spinlock - using the algorithm described in section 3 with a tight busy waiting loop.
- spinlock with `pause` - like the previous one where a `pause` is inserted inside the busy waiting loop.
- ticket lock - using the algorithm described in section 3 with a tight busy waiting loop.
- ticket lock with `pause` - like the previous one where a `pause` is inserted inside the busy waiting loop.

On IBM PowerEN the examined strategies are:

- pthread_lock - the same Operating System based locks as in the Intel X86 case.
- spinlock with low thread priority polling - using the algorithm described in section 3 and lowering the thread instruction priority before the busy waiting loop and resetting after it.
- spinlock with `wrlos` - like the previous one but replacing the busy waiting loop with `wrlos`.
- ticket lock with low thread priority polling - using the algorithm described in section 3 and lowering the thread instruction priority before the busy waiting loop and resetting after it.
- ticket lock with `wrlos` - like the previous one but replacing the busy waiting loop with `wrlos`.

The computation time on each data item is around one microsecond and the ratio between the time spent computing

on private data and updating the data structure is one order of magnitude. This is consistent with a fine grain parallelism scenario: a simple computation and the update of a complex data structure (like a hash table). The results highlight that, on both architectures, spinlocks have the lowest overhead and pthread locks, that require OS coordination, the highest.

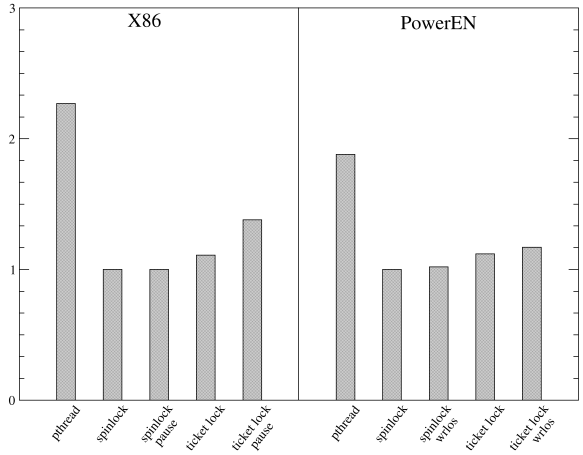

Figure 5: Normalized service time - no contention.

Figure 6 examines what happens to the average service time under contention on the Intel X86 architecture. The test creates a number of threads, each mapped on a different hardware context, a single global lock and a single global shared data structure. Each thread executes the same operations as the first test (compute on private data, lock, update shared data structure, unlock). The test stops when all threads have executed a predefined number of iteration. The results provide a good insights of the benefits and drawbacks of the lock prefix mechanism for hardware implementation of atomic operations and its interaction with hyperthreading. In the single socket scenario - see sub-graph - spinlocks (based on atomic CAS) achieve a slightly lower overhead than ticket locks (based on atomic increment). When the application starts using the second socket, and the QPI interface among processors, the atomic increment performance degrades drastically. Spinlocks still achieve a lower overhead compared to pthreads. Once the application starts using the second thread of each core spinlock performance degrades unless the `pause` instruction is used in the tight busy waiting loop; this reduces the CPU resources used by the spinning thread allowing the second thread to better access the core.

Figure 7 examines the average service time under contention on the IBM PowerEN architecture. This solution is user-space oriented, providing mechanisms to control the instruction dispatching priority and for waiting for memory location change. This design target and the available solutions clearly reflects in the performance results: pthread performance is extremely limited, since it requires interaction with the Operating System but, on the other hand, user-space performance is remarkable - with ticket locks clearly being the best. It is also interesting to notice that the memory reservation mechanism, used in the PowerPC architecture for implementing atomic operations, in this specific situation

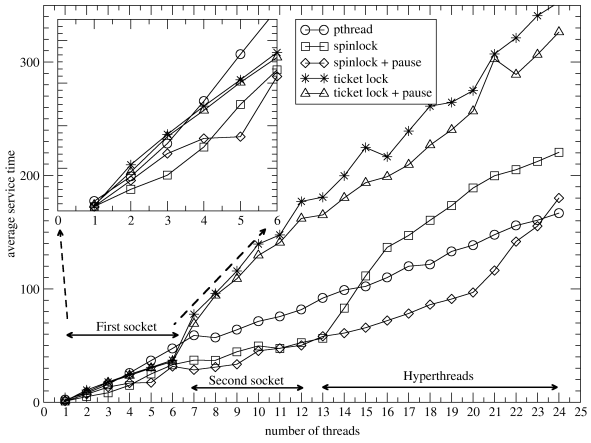

Figure 6: Normalized Intel X86 service time under load.

seems to make atomic increment more efficient than atomic compare and swap - the opposite of what is happening with the lock prefix mechanism used on Intel X86.

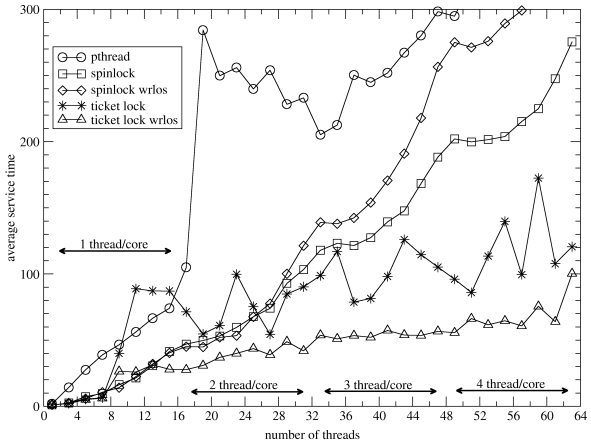

Figure 7: Normalized PowerEN service time under load.

The most important characteristic of multi-threaded solutions is that the hardware execution units are shared inside the CPU core. The reason for this choice is that "most of the time" the core is waiting for some data coming from lower (and slower) levels of the memory hierarchy, thus it is not performing any useful work. This situation is called "memory bound" computation. By multiplexing the use of hardware resources across multiple logical threads it should be possible to hide these delays and increase the number of instructions executed per unit of time. In any case the instruction per cycle performance will depend on the mix of operations executed by all dispatching threads at the same time. Every user-space lock implementation, that does not rely on the kernel to be notified of an event, works by looping over "a try to acquire lock" operation normally called *busy waiting*. These tight loops have the potential to monopolize CPU resources over more important work. The effect is analysed in Figure 8: a computation job is run on one CPU thread and the lock test job is run in all other

CPU threads and cores, including the co-located hardware thread (on PowerEN the test uses one, two or three of the co-located hardware threads). It is easy to see that an architecture such as PowerEN, which provides several user-space oriented features, can cope very well with this issue: the performance reduction is extremely limited and always below 5%. A more traditional solution like Intel X86 absolutely requires the use of `pause` instruction in busy waiting loops to avoid starving the computation thread.

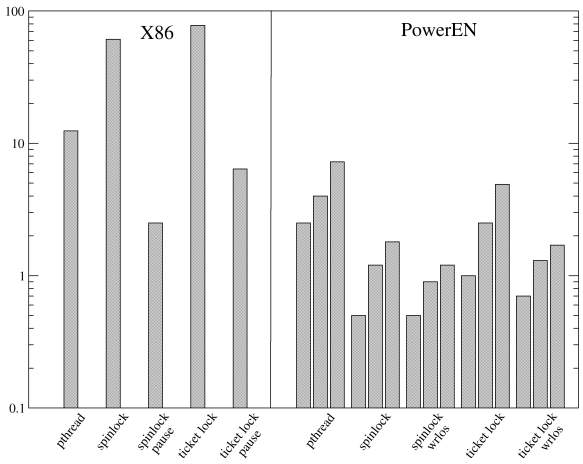

Figure 8: Normalized locking overhead over computation.

5. QUEUES

A common multi-threaded programming model is stream computing [6], where work is represented by a sequence of items pipelined through a graph of modules each executing on a different thread. Due to variances in the execution time of each sub work item, queueing provides a mean both of connecting modules and of adapting to workload fluctuation. To take advantage of highly multi-threaded processors it is to be expected that work items become smaller in nature and the network of threads deeper to exploit the large number of resources available. As a result, the impact of inefficient or inappropriate queue implementation will become more and more evident.

The semantics of queues expect a set of producers and a set of consumers to properly coordinate access to a limited or unlimited buffer of elements (data). Proper access implies that a producer can not enter a new element when the queue buffer is full, nor can a consumer retrieve an element if the queue is empty. Open is to what a thread (producer or consumer) should do when one of these boundary conditions exists. Similar to locking, the thread could either poll, go to sleep using OS services or use other wait mechanisms such as PowerEN `wrlos`. Efficient queues have received considerable attention in the research community. For instance, the non-blocking concurrent queue algorithm as described in [14] has demonstrated excellent performance even under high contention. Basic operations on queues are enqueue() and dequeue() - the former queues an element while the later removes the oldest (in a FIFO queue) element. Queues can be classified as:

- *Bounded* (when there's a maximum number of queued elements) or *UnBounded*;
- *Blocking* (when enqueue and dequeue operation are serialized using locks) or *NonBlocking*;
- *Partial* (when enqueue and dequeue operation wait for the queue to be not full and not empty) or *Total*;
- *Single Producer* (when only one thread is allowed to call enqueue operation) or *Multiple Producer*;
- *Single Consumer* (when only one thread is allowed to call dequeue operation) or *Multiple Consumer*;

Bounded queues are normally implemented using circular buffers, statically sized and not requiring any allocation at runtime, and can usually achieve higher performance. Unbounded queues must use some kind of dynamic data structure and allocators to implement their dynamic behavior. The implementation of a thread safe queue that supports multiple producers and/or consumers must always use locks or atomic operations. This paper focuses on fine grain parallelism, thus only "Bounded" queues are considered: the overhead of managing element allocation and deallocation as well as flow control at runtime would be overkill. Moreover the tests look only at the "Partial" version of enqueue() and dequeue() since the focus is understanding their performance and overhead independently of the actual behavior of the computation. Finally, given the lack of space and that Single Producer Single Consumer (SPSC) queues have been widely studied, the experiment consider only Multiple Producer Multiple Consumer (MPMC) semantic that is often used for "fan out" and "fan in" streaming configuration such as in a worker farm structure. Two different queue implementation are examined: a (1) Blocking and a (2) NonBlocking.

Algorithm 1 BNPVB enqueue() [CAS Implementation]

Require: q queue, $qlen$ max queue len, d data item

```
 1: loop
 2:   myEQC ⇐ q.eqc
 3:   myI ⇐ myEQC mod qlen
 4:   status ⇐ q.v[myI].s
 5:   if status ≡ myEQC then
 6:     if CAS(q.eqc, myEQC, myEQC + 1) then
 7:       q.v[myI].data = d
 8:       mem_sync()
 9:       q.v[myI].s = q.v[myI].s + 1
10:       return true
11:     else
12:       continue
13:     end if
14:   else
15:     if myEQC ≠ q.eqc then
16:       continue
17:     end if
18:     wait_for_change_meml(q.v[myI].s, status)
19:   end if
20: end loop
```

The first queue is built upon the Lamport[13] lock free single producer single consumer algorithm. This solution employs

Algorithm 2 BNPVB enqueue() [AttAdd Implementation]

Require: q queue, $qlen$ max queue len, d data item
1: $myEQC = Att\_ADD(q.eqc, +1)$
2: $myI \Leftarrow myEQC \bmod qlen$
3: $wait\_for\_val\_meml(q.v[myI].s, myEQC)$
4: $q.v[myI].data = d$
5: $mem\_sync()$
6: $q.v[myI].s = q.v[myI].s + 1$
7: **return** true

Algorithm 4 BNPVB dequeue() [AttAdd Implementation]

Require: q queue, $qlen$ max queue len, d data item
1: $myDQC = Att\_ADD(q.eqc, +1)$
2: $myI \Leftarrow myDQC \bmod qlen$
3: $wait\_for\_val\_meml(q.v[myI].s, myDQC + 1)$
4: $retv = q.v[myI].data$
5: $mem\_sync()$
6: $q.v[myI].s = q.v[myI].s + qlen - 1$
7: **return** true

a pre-allocated buffer B of size P and two indexes: the insertion index (ms) and the extraction index (mr); both indexes are initialized to 0. Insert first checks if the queue is not full by verifying that $ms < mr + P$; if insertion is allowed the new message is stored at position $B[ms\%P]$. Extract checks if the queue is empty by verifying $mr > ms$ and, if possible, retrieves the message at position $B[mr\%P]$. Concurrent access is synchronized since the single producer can write only the insertion index and read the extraction index; the single consumer write only the extraction index and read the input one. The extension[5] for handling multiple producers and consumers is simple and requires just two locks: the first is acquired when inserting an item, thus is protecting the head pointer; the second is used when removing an element, thus protecting the tail pointer. The "Partial" queue version considered in this paper waits (while holding the lock) for the error condition to disappear; this wait is implemented using either polling or `wrlos` (on PowerEN).

Algorithm 3 BNPVB queue dequeue()

Require: q queue, $qlen$ max queue len
1: **loop**
2: $myDQC \Leftarrow q.dqc$
3: $myI \Leftarrow myDQC \bmod qlen$
4: $status \Leftarrow q.v[myI].s$
5: **if** $status \equiv myDQC + 1$ **then**
6: **if** $CAS(q.dqc, myDQC, myDQC + 1)$ **then**
7: $retv = q.v[myI].data$
8: $mem\_sync()$
9: $q.v[myI].s = q.v[myI].s + qlen - 1$
10: **return** $retv$
11: **else**
12: **continue**
13: **end if**
14: **else**
15: **if** $myDQC \neq q.dqc$ **then**
16: **continue**
17: **end if**
18: $wait\_for\_change\_meml(q.v[myI].s, status)$
19: **end if**
20: **end loop**

The second queue version is implemented using a novel algorithm here introduced. Data items are stored inside a shared circular vector accessed using a shared enqueue and a dequeue offset; no locks are required, making this implementation "Non Blocking". Each entry of the the support vector contains a variable to store pushed data and a status variable which exactly identifies the data variable either readable or writable. At initialization, all the status variable are set to the index of their entry in the vector. The mechanism of the status is based on some properties of toroidal spaces and it requires the cardinality of the numbers representable by the status variable to be a multiple of the queue length. In the presented implementation a 64 bit variable was chosen, therefore the length of the queue can be set to any power of two smaller than 2^{63}. The enqueue() Compare And Swap based implementation ("partial" version) is shown in Algorithm 1 and its Atomic Add version in Algorithm 4. It is based on a single compare and swap (line 6) that atomically updates the enqueue index reserving one slot for the producer. In the "partial" implementation line 20 is used to wait for the status of the target index to change value; this can be realised using polling or `wrlos`. The dequeue() behaviour is shown in Algorithm 3 for the Compare And Swap and in Algorithm4 for atomic add. Again, this is based on a single compare and swap (line 6) that updates the dequeue index when an element is present and ready to be extracted, and line 18 shows the wait for a status change implemented with `wrlos`.

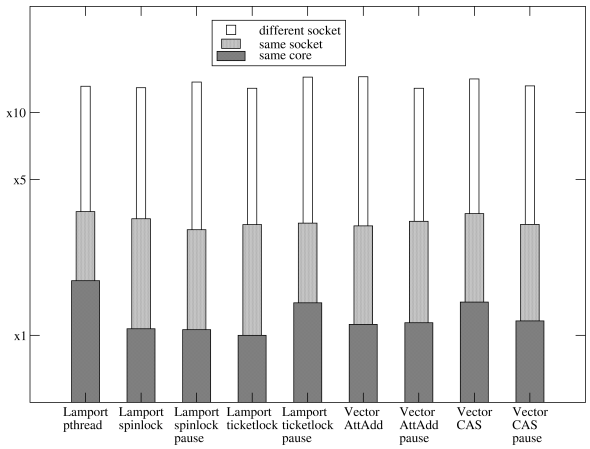

Figure 9: Normalized queue latency on Intel X86.

6. QUEUE PERFORMANCE

This paper focuses on queues used as communication support for fine grain streaming applications, where work items are received as an (infinite) sequence and "flow" across a network of lightweight threads that perform some computation and forward the item to other threads. Note that, since the programming environments we are interested in are shared memory, these queues contain references to data items to avoid the overhead of copying the actual data. The most common communication patterns used in this scenario are: point-to-point, fan-out and fan-in. Fan out and fan in structures are normally used for implementing thread farms,

where several workers are used to increase the bandwidth of a specific computation; in this case the most important parameter is scalability: how many workers can be added to the farm before saturation. Point to point links are normally used to implement other network topologies of threads and in this case the most important parameter is latency: how much time it takes for data items to flow across the connection.

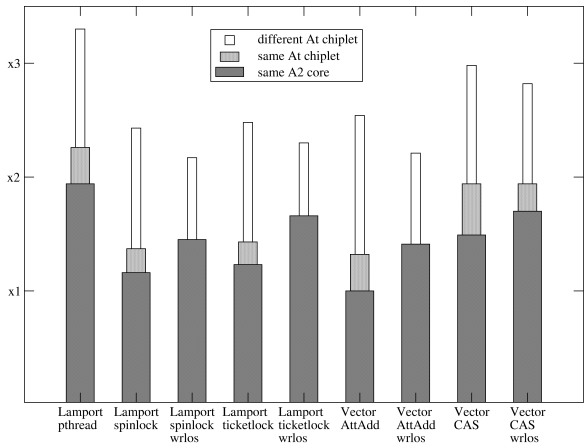

Figure 10: Normalized queue latency on IBM PowerEN.

The first experiment measures the basic point-to-point (PtP) queue latency under the optimal condition of an empty queue. This is the minimum overhead involved in handling the communication between two peers on the architecture under exam. The basic queue latency is heavily influenced by how the producer and consumer are mapped on the underlying physical architecture. Figure 9 shows the latency data (normalized to the minimum value) on the Intel X86 architecture under test for the two queues considered (Lamport lock based and Vector-based lock free) and the three possible mappings (same core, same socket and different sockets). When producer and consumer are mapped inside the same core, using two hyper-threads, the spinlock based Lamport queue seems to have an edge, being twice as fast as the pthread based Lamport implementation and showing a tiny increase over the vector based lock free implementation. When the peers move outside the single core, but still inside the same socket, the situation does not change too much, with the spinlock version still showing a small edge over the other implementations. From this results it is possible to see that the L2 cache, used when exchanging data between two cores, is about 3 times slower than the L1 cache that is used when exchanging data between hyper-threads inside the same core. When mapping the two threads on different sockets, the performance of every queue implementation degrades more than 10 times and the vector based queues, with atomic add or atomic CAS, become the best performing, but with a small edge (about 10% latency) over the others.

Figure 10 examines the same scenarios on IBM PowerEN. Since this hardware solution is a system on a chip the performance difference between the various mappings is very limited and always below 3 times, thus ensuring a very low overhead for exchanging data between lightweight threads. An interesting side effect of how `wrlos` works is that, being implemented inside the L2 controller, queueing solutions that employ the instruction perform exactly the same in the "same A2" and "same At chiplet" case. Looking at the bar chart it is possible to see that the vector based implementation with atomic add, both with and without `wrlos`, provide very good performance across all mappings.

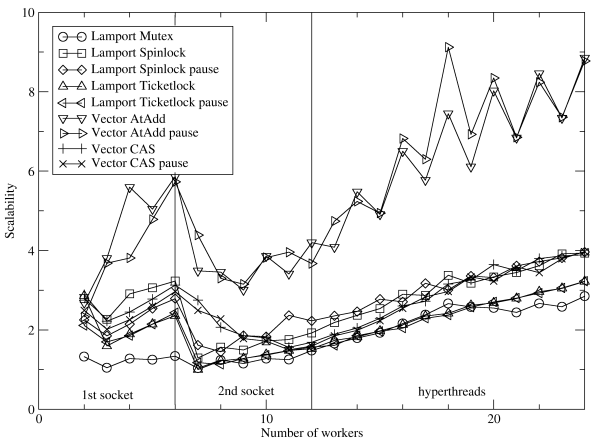

Figure 11: Fan-out scalability on Intel X86.

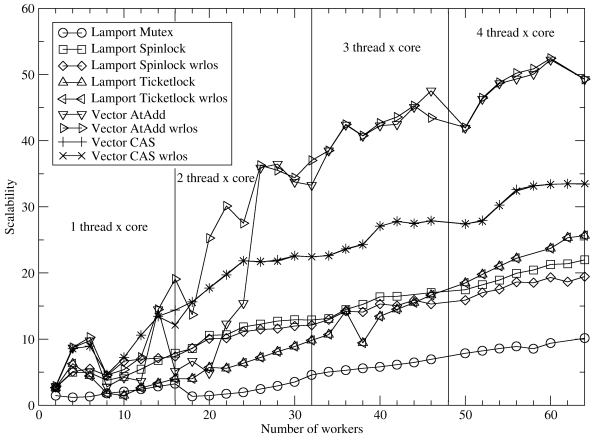

Figure 12: Fan-out scalability on IBM PowerEN.

The next test explores the behaviour in the fan-out case, which is often used to distribute work items on a farm of threads to increase application bandwidth. Since this paper focuses on fine grain parallelism, the task performed by farm test workers takes about one microsecond. The single queue producer is enqueueing work items as fast as possible and the overall bandwidth is measured as the number of workers increases. The graph reports scalability: the farm bandwidth is normalized over the slowest configuration. Intel X86 figures are reported in Figure 11; it is easy to see that there's a clear winner on this platform: the vector based queue using atomic add reaches a maximum scalability of 9 with about 20 workers.

It is interesting to observe what happens when we start using two sockets for coordinated fine grain parallelism using a

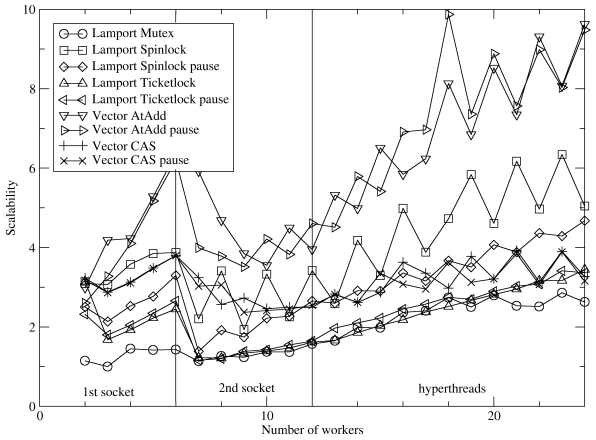

Figure 13: Normalized Fin bandwidth on Intel X86.

streaming farm paradigm: the overall performance degrades sharply *using any tested synchronization mechanism*; this is due to the overhead involved at running atomic operations over the QPI inter-chip link. The only way to "recover" the lost performance it to use almost all the available threads; this is a strong signal that hiding the memory cost, including the atomic operation cost, is essential to achieve scalability for fine grain parallelism. Multi-threading seems to be a very good strategy to achieve the desired results.

This can be confirmed by looking at IBM PowerEN figures shown in Figure 12. This hardware solution is heavily multi-threaded, having 4 threads per core, and the performance results show how scaling the number of workers is able to multiply performance by over 50 times using 64 workers! The vector based lock free implementations are much better than the lock based Lamport versions, regardless of the kind of lock employed. Again the atomic add version of the queue achieved the best scalability.

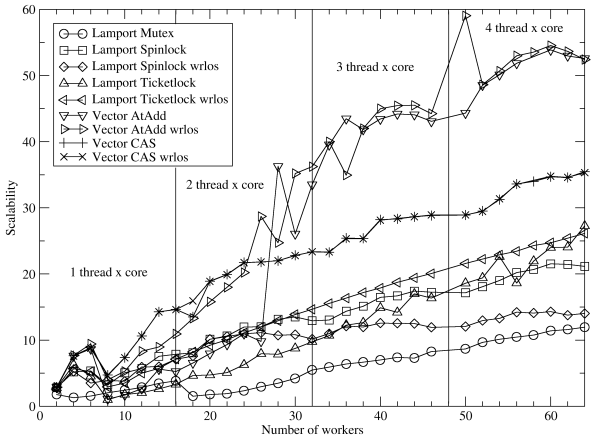

Figure 14: Normalized Fin bandwidth on IBM PowerEN.

The fan-in test examines the multiple producer single consumer case, often used to collect worker farm results for further processing in a streaming environment. Also for these tests the amount of computation performed by farm workers is in the microsecond range and the results are reported as scalability by normalizing the overall bandwidth over the slowest case. Intel X86 figures are shown in Figure 13 and are very similar to the fan-in case: the vector based lock free queue that uses atomic add achieves the best scalability, which in any case it is not too good, reaching a maximum of 10 with 24 workers. Also in this case we can observe the "second socket performance penalty"; this is consistent with the fact that all these queues are symmetrical: the complexity of element insertion and extraction is identical.

IBM PowerEN figures are shown in Figure 14 and again are very similar to the fan-out case. Interestingly enough the scalability in the single producer multiple consumer case is slightly higher than the multiple producer single consumer but the lock free schema have always an edge over the lock based solution.

7. CONCLUSION AND FUTURE WORK

It is well known that the selection of the proper synchronization mechanism is fundamental in any multi-threaded program. When these applications are executed on a large multi-core multi-thread processor this becomes (if possible) even more important. It is also common knowledge that "Operating Systems calls must be avoided as much as possible": the idea is that their cost is huge when compared to their user-space counterparts and anyway scheduler involvement is not required when the number of available hardware threads is high.

This naive analysis states that user-space operations should be used whenever possible to increase program performance and scalability, but their specific implementation mechanism (such as busy waiting or atomic operations) must be carefully evaluated and it is tied to the desired program synchronization semantic. An in depth analysis of the benefits and drawbacks of Operating System and userspace based solutions was still missing and this work desires to close this gap at least in the fine grain parallelism case, where this issues can become predominant over the application algorithmic aspects.

This paper examines the performance of two fundamental inter-thread synchronization mechanisms, locks and queues, in the scenario of fine grain parallelism. The tests are designed to highlight the architectural differences between two hardware solutions whose design targets are completely different:

- The first system examined here is Intel X86 Nehalem, a Complex Instruction Set Computer architecture with powerful cores, very large on chip cache and a low degree of Symmetric Multi Threading. Intel X86 systems usually embed 4 to 6 cores on a chip and employ multiple sockets, connected via Quick Path Interconnect links, to achieve more concurrency. The primary design target for this processor is heavyweight processing of complex serial workloads, thus the units employ very advanced out of order execution and branch prediction strategies.
- The second design point here analysed is IBM Pow-

erEN, a novel solution that sits half way between a network processor and a server processor. This design is targeted at throughput oriented workloads and couples a Reduced Instruction Set Computer architecture, with an high degree of Symmetric Multi Threading, many cores with simple "in order" execution engines, large caches and several novel userspace oriented features. Everything is embedded in a single chip to provide a powerful foundation for exploiting the available fine grain parallelism.

The first basic synchronization mechanism examined is locks, often used in non streaming applications to protect and serialize the access to shared resources. The common interaction scenario is: compute on private data, lock, modify shared data structure and unlock. This paper examines fine grain parallelism, where the computation cost is about one microsecond and the time spend in updating the shared resource is one order of magnitude less than the computation. Three types of locks are evaluated: one operating system based (pthreads) and two userspace based locks (spinlocks and ticket locks). Results show how userspace based solutions are mandatory for fine grain parallelism. The more advanced mechanisms available on PowerEN allow much better scalability by reducing the overhead experienced at very high contention.

The second mechanism here evaluated are queues, often used as basic support of streaming programming environments. Again the focus is on fine grain parallelism, with very small computational task of about one microsecond for each data token. The tests cover only bounded multiple producer multiple consumer types of queues, which are the most used in shared memory streaming environments. Two distinct implementations are examined: a blocking one based on the extension of Lamport algorithm and a non blocking one based on a circular buffer and a lock free protocol that uses atomic operations. The test results show that multi-threading is a good mechanism for hiding both memory access latency and atomic operation costs; moreover the availability of specific userspace mechanisms to control CPU instruction dispatch priority and to wait for memory-based events allow for unmatched scalability.

The results here outlined support the concept that *fine grain parallelism, with unit of work in the order of few microseconds, is indeed possible on modern architectures.* Program scalability can be helped by Symmetric Multi Threading and it is extremely sensitive to the use of architecture specific optimizations that start from the proper selection of algorithms. The results highlight also the fact that some novel mechanism can indeed help program performance and scalability: providing in userspace the ability to control hardware thread instruction scheduling priority and the possibility of passively waiting on a memory location reduces the synchronization overhead by an order of magnitude.

Future work will evaluate how the Non Uniform Memory Access characteristics of most systems can influence the performance and synchronization scalability figures. More work must also be done on architectural support for user-space synchronization by leveraging the cache coherence protocol: which other mechanism (if any) would provide the best benefit to the implementation of locks and queues.

8. REFERENCES

[1] R. Bianchini, E. V. Carrera, and L. Kontothanassis. The interaction of parallel programming constructs and coherence protocols. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '97, pages 69–79, New York, NY, USA, 1997. ACM.

[2] G. Cong and D. A. Bader. Designing irregular parallel algorithms with mutual exclusion and lock-free protocols. *J. Parallel Distrib. Comput.*, 66:854–866, June 2006.

[3] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[4] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25, May 2007.

[5] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 43–52, New York, NY, USA, 2008. ACM.

[6] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[7] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. *Proceedings of the 42nd Annual IEEEACM International Symposium on Microarchitecture Micro42*, 978(1):413, 2009.

[8] J. Hennessy, M. Heinrich, and A. Gupta. Cache-coherent distributed shared memory: Perspectives on its development and future challenges. *Proceedings of the IEEE*, 87:418–429, March 1999.

[9] H.Franke, J.Xenidis, B.Bass, C.Basso, S.Woodward, J.D.Brown, and C.L.Johnson. Introduction to the Wirespeed Architecture and Processor. *IBM Journal of Research and Development*, 2010.

[10] . N. T. P. P. O. . U. HUM, Herbert and A. . T. R. M. W. . U. GOODMAN, James. Forward state for use in cache coherency in a multiprocessor system, 07 2004.

[11] C. Johnson, D. H. Allen, J. Brown, S. Vanderwiel, R. Hoover, H. Achilles, C.-Y. Cher, G. A. May, H. Franke, J. Xenedis, and C. Basso. A Wire-Speed PowerTM Processor: 2.3GHz 45nm SOI with 16 Cores and 64 Threads. In *Proceedings of the IEEE Int. Solid-State Circuits Conf*, pages 104–105, 2010.

[12] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC '94, pages 130–140, New York, NY, USA, 1994. ACM.

[13] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5:190–222, April

1983.

[14] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51:1–26, May 1998.

[15] A. Naeem, X. Chen, Z. Lu, and A. Jantsch. Scalability of relaxed consistency models in noc based multicore architectures. *SIGARCH Comput. Archit. News*, 37:8–15, April 2010.

[16] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 232–248, New York, NY, USA, 1987. ACM.