Timing Properties and Correctness for Structured Parallel Programs on *x86-64* Multicores

Kevin Hammond, Christopher Brown, and Susmit Sarkar

School of Computer Science, University of St Andrews, Scotland, UK.
{susmit.sarkar,kevin.hammond,cmb21}@st-andrews.ac.uk

Abstract. This paper determines correctness and timing properties for *structured* parallel programs on *x86-64* multicores. Multicore architectures are increasingly common, but real architectures have unpredictable timing properties, and even correctness is not obvious when systems are built on the relaxed-memory concurrency models that are enforced by commonly-used hardware. This paper takes a rigorous approach to correctness and timing properties, examining common locking protocols from first principles, and extending this through queues to structured parallel constructs. We prove functional correctness and derive simple timing models, extending these for the first time from low-level machine operations to high-level parallel patterns. Our derived high-level timing models for structured parallel programs allow us to *accurately predict* upper bounds on program execution times on *x86-64* multicores.

1 Introduction

Multicore architectures are increasingly common, providing excellent tradeoffs between performance and energy consumption. However, the actual execution behaviour of parallel programs on real multicores is still often difficult to understand. Arguing about the correctness of parallel programs is non-trivial in the presence of real-world relaxed-memory consistency models. Moreover, predicting the execution time of parallel programs is hard. Both issues derive from the same problem: *there is no well-understood correspondence between the high-level parallel primitives that programmers use and the low-level implementations of those primitives that are actually executed.* We address this problem by directly considering correctness and timing properties from basic machine-level operations all the way to high-level parallelism structures. This paper exploits *structured parallel programming*, in the form of *algorithmic skeletons* [1,2]. Such approaches bring similar abstraction advantages to parallel program design that standard structured programming brings to sequential programming, abstracting over basic parallelism primitives, such as process creation, communication and synchronisation. The main advantages to us are:

1. The programmer can think in terms of *patterns* of parallelism, rather than e.g. low-level memory operations;

```
int x = 0, y = 0;                 extern int x, y;
...                               ...
{ // thread 1                     { // thread 2
  x = 1;                            y = 2;
  return y;                         return x;
}                                 }
```

Fig. 1. Under the *x86-TSO* relaxed-memory consistency model, both threads could return 0. This is not possible under SC.

2. Certain complex parallel conditions, such as race conditions and deadlocks, are eliminated *by design*, so dramatically simplifying design, implementation and testing of parallel programs;
3. Simple, but effective, performance models are possible.

This paper exploits these advantages to produce strong and principled cost models for two fundamental parallel programming structures: *task farms* and *parallel pipelines*. Danelutto *et al.* [3] have shown that many common parallelism patterns can be expressed in terms of these two primitives. Our work thus extends straightforwardly to many other high-level patterns of parallelism.

Memory Consistency. Standard correctness proofs of parallel programs usually assume memory accesses to be interleaved, so-called *Sequential Consistency* (SC) [4]. However, *x86-64* multicores, and many other recent architectures, do not comply with this assumption, making such reasoning invalid. Consider, e.g., the program fragment in Figure 1. Under the *Total Store Order* relaxed-memory consistency model that is used by *x86-64* multicores (*x86-TSO*), both threads could return 0, a result that is impossible using the simpler SC model.

Novel Contributions. While there has been significant previous work on algorithmic skeletons, this paper represents the first attempt, of which we are aware, to establish cost models from first principles for widely-used multicore hardware. In addition, this paper makes the following specific novel contributions:

1. It describes an operational semantics for structured parallel programs, that is used to derive, from first principles, a compositional cost model for any combination of the farm and pipeline algorithmic skeletons;
2. It gives a simple operational proof of the (partial) correctness of a widely-used *spin-lock* protocol using the actual relaxed-memory concurrency semantics used by *x86-64* multicore machines; the proof composes in a straightforward fashion to show that the farm and pipeline algorithmic skeletons are free from deadlock, both singly and in any combination, when implemented by queues using that locking protocol;
3. It validates the cost model against some example parallel programs, giving accurate predictions of lower-bound speedups. Our predictions are typically within 7% and in all cases within 30% of the actual speedup.

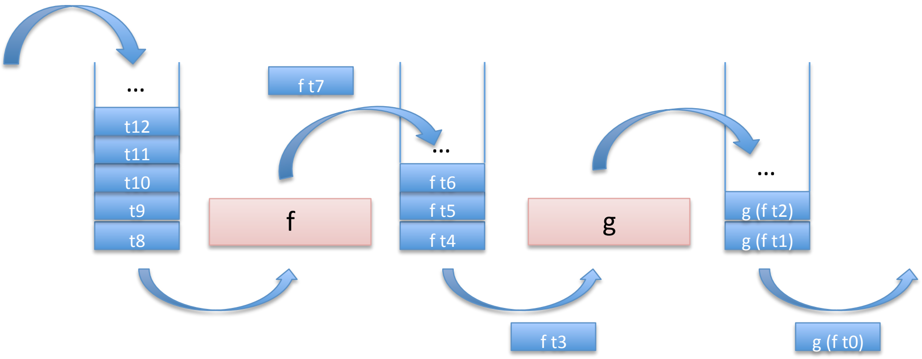

Fig. 2. Two-Stage Parallel Pipeline: $f \mid g$

A key aspect of our approach is that we model the actual relaxed-memory model used by real *x86-64* multicore systems, exploiting recent work on the *x86-TSO* model [5,6], which gives a precise account of the observable behaviour of *x86-64* multiprocessors in terms of an idealised abstract machine. We are thus able, for the first time, to provide realistic cost models for programs that are executed in parallel on such architectures. The spin-lock protocol that we use has been shown to be the most efficient protocol for implementing simple locks on *x86-64* architectures [5]. It is widely used to implement real parallel programs. Our correctness proof is the first direct operational proof that this protocol is correctly implemented by the underlying hardware through its relaxed-memory access protocol. Finally, the queue protocol shown here has many uses beyond algorithmic skeletons, and is representative of many widely-used higher-level synchronisation protocols. The correctness proof shown here is likewise the first proof that such a protocol is correctly implemented by the underlying multicore hardware.

2 Pipeline and Farm Skeletons

In this paper, we consider two fundamental skeletons: 2-stage *pipelines* (Figure 2) and *task farms* (Figure 3). We use a streaming implementation that links skeletons using (unbounded) queues. This allows skeletons to be easily composed, and also allows arbitrarily nested parallel structures to be built from the basic skeletons. Parallel pipelines ($f \mid g$) represent the composition of two operations f and g, streamed over a sequence of inputs $t_0, t_1, \ldots$, with f and g possibly executed in parallel. The result of the pipeline is the stream $g\ (f\ t_0), g\ (f\ t_1), \ldots$ Multi-stage pipelines can be built by composing multiple two-stage pipelines and then merging the input/output queues. *Task farms* ($\Phi(f)$) apply the same operation f to each of the inputs in a stream. A fixed number of worker instances is created, which each apply f in parallel to a subset of the inputs. The result of applying a farm to a stream of inputs $t_0, t_1, \ldots$ is then the *bag* $\{f\ t_0, f\ t_1, \ldots\}$, where the results may be produced in an arbitrary order. This definition of a

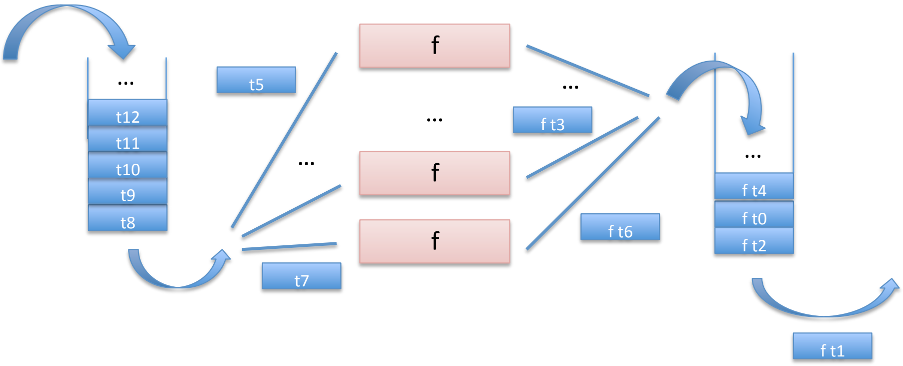

Fig. 3. Task Farm: $\Phi(f)$

```
Value qget(Queue q)
{
  Value v;
  do {
    lock(&q.lock);
    if (!qempty(q))
      break;
    unlock(&q.lock);
  } while (1);

  /* lock is held */
  v = front(q);
  unlock(&q.lock);
  return(v);
}
```

```
void qput(Queue q, Value v)
{
  lock(&q.lock);
  addtoq(q,v);
  unlock(&q.lock);
}
```

Fig. 4. Simple queue implementation in C.

task farm allows an efficient parallel implementation, where each of the workers is mapped to a different processing agent. As each worker produces its result, it is placed in the output queue, and the worker takes the next input (if any) from the input queue. Farms can be nested to an arbitrary depth by replacing the operation f by a farm and linking the corresponding input and output queues. It is also possible to embed pipelines within farms or *vice-versa*, so yielding arbitrarily complex parallel systems.

2.1 Simple Queue Implementation using Locks

The pipeline and farm skeletons use queues to manage the input/output streams. Figure 4 shows how queues can be implemented using `lock` and `unlock` primitives, plus operations to remove the first element from the queue (`front`), add a new element to the queue (`addtoq`), and check whether the queue is empty (`qempty`). Queues are implemented using the abstract type `Queue`, containing values of type `Value`. The `qget` operation returns the first element from the queue, spinning if the queue is empty. This implements a blocking read operation. The corresponding `qput` operation adds a value to the end of the queue, locking and unlocking as necessary. When used as a pair, `qget`/`qput` implement a synchronisation operation between two parallel threads. Figure 5 gives a spin-lock implementation of `lock`/`unlock` in C and *x86-64* assembler, assuming an atomic exchange primitive, `exchange`. Spin-locks are widely used in parallel systems where there are low contention rates, since they are simple to implement and the costs of acquiring and releasing the locks are very low. In fact, until recently, the Linux kernel used almost identical code. For fairness reasons, it now uses a slightly modified version (*ticketed spinlocks*). Here, `lockcell` is a variable that contains the lock. If it has the value `1`, then the lock has been acquired by some thread; if it has the value `0`, then no thread has acquired the lock. Acquiring the lock using the `lock` routine involves reading the value of `lockcell`, and exchanging it with the locked value, `1`. If the lock has already been acquired by another thread, the process is repeated until the lock can be acquired (i.e. the previous value of `lockcell` was `0`). Releasing the lock using the `unlock` routine simply involves setting `lockcell` to `0`. Note that the unlock need not be an atomic memory operation.

3 Key Hardware Characteristics

In what follows, we will consider memory accesses to be classified into *Read*s from and *Write*s to specific memory locations, plus *Fence*s and atomic *Exchange*s. Intel's *x86-64* instruction set also provides some refined versions of these basic operations [7]. *Sequential consistency* (SC) memory models [4] ensure that memory accesses from multiple threads are carried out in an order that is consistent with some valid set of memory accesses by a fully sequential processor. That is, memory accesses from different threads are interleaved so that there is effectively a single thread of memory accesses. Recall the simple example from Figure 1. Here, depending on the exact timing of reads and writes, thread 1 could return either 1 or 0 and thread 2 could return either 2 or 0. It is not possible for both threads to return 0, however, since one of the writes must happen last. While SC is effective on uni-processor systems, enforcing an SC memory model on a multicore system can carry significant performance penalties. For example, all caches and other memory hardware must be synchronised in order to avoid inconsistent results. Since such a strong model is not always required, multicore hardware vendors generally support weaker consistency models that offer higher performance. Modern *x86-64*-class microprocessors use Total Store Ordering (*x86-TSO*) [6,5]. *x86-TSO* guarantees that the order in which *Write* instructions for a given processor appear in (shared) memory is identical to the sequence in which the processor issued the *Writes*. *x86-TSO* can be implemented by providing each hardware thread of execution with a private FIFO write-buffer. This is an abstract machine implementation: a more realistic hardware implementation is discussed further below. *Writes* are stored temporarily

```
void lock( volatile char *lockcell ) {
   char old_value ;
   do {
      old_value = exchange(lockcell , 1);
   } while ( 1 == old_value ) ;
}

void unlock( volatile char *lockcell ) {
   *lockcell = 0 ;
}
```

```
; Assume EBX contains address of lock cell
_lock:
  mov  eax, 1      ; Set EAX register to 1 (locked)
  xchg eax, [ebx]  ; Exchange EAX and lock cell
  test eax, eax    ; Test if cell is locked
  jnz  _lock       ; Retry the lock if so
  ...              ; Lock held here


_unlock:
  mov  eax, 0      ; Set EAX register to 0 (unlocked)
  mov  [ebx], eax  ; Release the lock
```

Fig. 5. Above, simple spin-lock implementation in pseudo-C using an atomic `exchange` operation on the `lockcell` memory location; `1` indicates the lock is acquired; 0 is used to release the lock. Below, corresponding *x86-64* instructions (ignoring function prelude/postlude). The `xchg` instruction in `_lock` acquires the lock, which is released in in `_unlock` using a normal memory write. Note that on *x86-64*, a `xchg` implicitly behaves as a full memory barrier.

in this buffer prior to being actioned by the main memory. *Reads* from the local processor (only) can access this write buffer, if necessary, as an intermediate step between reading from the private and shared caches. *Writes* recorded in the write buffer will be used in preference to any corresponding values in the shared cache. After an unpredictable, but finite, time, each *Write* is flushed from the buffer, and so becomes visible to all processors. In this way, SC is enforced for a single processor, *but not for all processors in a multi-processor system.* In the *x86-TSO* model, explicit *memory fence* (`mfence`) or *atomic exchange* (`xchg`) instructions are needed to enforce consistency across multiple processors. When a Fence/Exchange instruction is encountered by a processor, all of its outstanding *Reads* and *Writes* are executed immediately. This provides strong local temporal guarantees: all local accesses that appeared in the instruction stream prior to a *Fence/Exchange* will be executed *before* any accesses that appeared after the *Fence/Exchange*. It also provides strong global memory guarantees: all memory locations (and any cached copies) will be consistent with the memory state immediately following the *Fence/Exchange*.

Definition 1. *Thread. A* Thread *is an ordered sequence of memory accesses, where memory accesses comprise* Reads*,* Writes*,* Fences *and* Exchanges *to specific memory locations. A* Read *takes a memory location and returns a value. A* Write *takes a memory location and a value and has no result. A* Fence *has no parameters or result. An* Exchange *is treated as a simultaneous and indivisible* Read*,* Write *and* Fence*. It takes a memory location and a value and returns a (possibly different) value.*

Definition 2. *Write buffer. A* Write buffer *is associated with each thread, and contains an ordered sequence of* Writes*, each associating a value with an address. Each* Write *is recorded in the corresponding write buffer immediately it is executed by any* Thread*.*

Definition 3. *Memory. A* Memory *maps addresses to values.*

Definition 4. *Core. A* Core *comprises a set of* Threads*.*

Definition 5. *Multicore. A* Multicore *comprises a set of* Cores *plus a single shared* Memory[1]*.*

Definition 6. *Execution Order. The* execution order *is a linear trace of transitions made by the labelled transition system called the x86-TSO machine [6].*

Hardware Correspondence The abstract model above talks about "threads" of execution. In current multiprocessor implementations, these are grouped together in various ways, each of which has varying implications for scheduling and timing. Firstly, *user-level threads* are mapped by the *runtime system* to *hardware-level threads*. In this paper, we ignore this scheduling cost. That is, we consider only a non-preemptive model with no explicit thread *yields* to other threads or the runtime system. Taking such yields into account would introduce interference with other processes and threads running on the system. The model above speaks of such hardware threads when mentioning a thread of execution. Secondly, each hardware thread is mapped to specific hardware resources on a core. The mapping may be one-to-one, or many-to-one (*hyperthreading*, also known as *simultaneous multithreading*). Each hyperthread generally has exclusive use of registers and a load-store reordering buffer. The load-store reordering buffer maintains metadata to ensure observable FIFO buffering for the stores, and also to ensure that loads appear ordered to the programmer, even though aggressive implementations can and do perform out-of-order operations, e.g. satisfaction of read requests. Third, a set of cores (typically 2 to 4) are collected in one CPU. Typically, the first levels of cache (L1 and L2 on the Intel Core-i7) are private to the core, while higher levels (L3 on the Intel Core-i7) are shared between all the cores on one CPU. Finally, multiple CPUs may be connected in one system, all sharing a common memory. The caches communicate with each

[1] We will ignore shared (L2) cache here, since it does not have a significant impact on the proofs.

other and with memory to maintain "cache coherence", that is, a clear notion of order of update operations (stores) to a location. This is generally managed at the cache-line granularity (64 on the Intel Core-i7). To be clear, the functional model above speaks of store buffers. These are implemented by so-called Memory Ordering Buffers (MOBs), which internally do out-of-order actions. The flushing of buffers then corresponds to the point when non-local threads (on the same core or otherwise) can see those stores. Further buffering occurs between various levels of the cache. However, in all cases the cache maintains a coherent view, so that if the store is visible to at least one hardware thread other than the one executing the store, then it is visible to all of them.

x86-64 Cache Protocol. Both the Intel and AMD implementations of the cache protocol are variations of the classic MESI protocol[8]. Conceptually, there can be four states for each location in a cache (managed on a cache-line granularity): **M**odified (this cache holds an exclusive copy and memory has a stale copy), **E**xclusive (this cache holds an exclusive copy, the memory copy is also valid), **S**hared (this cache holds a copy, but other caches possibly hold copies as well), and **I**nvalid (this cache does not hold a current copy for this location). While the cache behaviour does not directly impact the functional correctness proof, it does have implications for the cost models we develop.

4 Correctness and Progress Properties

This section considers the functional correctness of the spin-lock, queue and pipeline/task farm implementations described above, working from first principles in terms of the basic *x86-64* memory operations and *x86-TSO* consistency model. We first consider a key ordering relation, *coherence order*, then sketch soundness proofs for the spin-lock and queue implementations, and finally build on these to sketch soundness proofs for the skeleton implementations.

4.1 Coherence Order

A derived relation called *coherence order* naturally emerges from the *x86-TSO* memory consistency model described above. *Coherence order* is a total, linear ordering of *Writes* to a given memory location, organised in the order that the *Writes* affect the memory location (are *flushed* from their local buffers).

Definition 7. *Coherence Order. Given a set of* write buffers*, WB_i containing tuples $\langle t, m, v\rangle$, where t is the time that the* Write *is flushed from WB_i, m is the memory location, and v is the new value to be written to that location, then the* coherence order *for some memory location m is defined to be the sequence*

$$CO(m) \text{ linear order over } \{\langle t_j, m, v_j\rangle \mid \langle t_j, m, v_j\rangle \in \bigcup_{i=1}^{n} WB_i\}$$

$$s.t. \quad \forall j, k. \langle t_j, m_j, v_j\rangle <_{CO(m)} \langle t_k, m_k, v_k\rangle \implies t_k > t_j.$$

Note that under this definition, it is not possible for two *Writes* to the same memory location to occur at the same time. *Writes* to different memory locations may, however, occur at the same time. This is consistent with the restrictions of physical memory. It is easy to show from this definition that no *Thread* can read values out of coherence order.

Lemma 1. *Read Coherence Order. No Thread can read values out of coherence order.*

Proof Sketch: Suppose that two *Reads* r_1 and r_2 in the same *Thread*, T, from the same memory location, m, occur in order, but that they return different values. Let us call the corresponding *Writes* w_1 and w_2 ($w_1 \neq w_2$). We do a case analysis depending on whether w_2 is read by r_2 from the local write buffer or from memory. Suppose that w_2 is in the *write buffer*. Then w_2 cannot have been the last write in the write buffer at the time of the read r_1, and thus w_2 must be flushed at some later time, and definitely later than w_1. Suppose that, instead, w_2 is taken from memory. Then the local write buffer must be empty for that location. Now, either r_1 read w_1 from the write buffer, and therefore w_1 must have been flushed (before w_2), or it read it from memory, and again w_2 is in *Coherence order* before w_1. □

4.2 Functional Correctness of the Spin-Lock Implementation

The spin-lock implementation needs to enforce two key properties: i) that at most one thread at a time possesses the lock; and ii) that all *Writes* that are made while the lock is held are always visible to any subsequent thread that acquires the lock. We show this by using the memory properties defined in the previous section, together with the code for the spin-lock implementation of `lock` and `unlock` that was given in Figure 5.

Lock acquisition and release. We assume that the only *Reads*/*Writes* to `lockcell` are made by the `lock` and `unlock` functions. We also assume that `unlock` is called only when safe, i.e. when the calling *Thread* possesses the lock, and that `lock` is called only when the calling *Thread* does not possess the lock. It follows that the only values that `lockcell` can contain are `0` (the unlocked value) and `1` (the locked value).

Theorem 1. *Lock acquisition and release. Under the assumptions of the previous paragraph,*

- *following a call to `lock`, the calling* Thread *will possess the lock, and no other Thread will possess the lock; and*
- *following a call to `unlock`, the calling* Thread *will no longer possess the lock until it has successfully called `lock` again.*

Proof Sketch: Both of the above parts can be proved simultaneously by induction on the `lock` and `unlock` calls in an execution trace, and case analysis of the write buffer state at every lock. The key operation is the `lock` function, which is called when acquiring the lock. We return from this function exactly when the internal loop exits, that is when `old_value` is not `1`. This means that `old_value` is `0`, or in other words, that `lockcell` held `0` in the final iteration of the loop and is now `1`. Since the only way to update `lockcell` with a locked (`1`) value is via an Exchange, this will always appear directly in memory, and never stay in the write buffer. In contrast, an unlocked (`0`) value can stay in the buffers, since this is done by a simple *Write*. Since only the *Thread* that has successfully acquired the lock ever calls the `unlock` function, and since it does this precisely once for each lock acquisition, there will only ever be at most one unlock value (`0`) for `lockcell` in any of the write buffers. There are now two cases to consider.

Case 1: Consider first the case when no write buffer contains an unlock value. If `lockcell` holds `1`, then the lock is already held. No lock acquisition can succeed until `lockcell` becomes `0`. Conversely, if `lockcell` holds `0`, the lock is free. Any isolated lock acquisition will successfully acquire the lock, but because of the atomic nature of the *Exchange*, if there are multiple simultaneous acquisition attempts then only one of these can succeed. Furthermore, because of coherence on `lockcell`, no subset of threads can disagree on which lock acquisition succeeded.

Case 2: The second case occurs when some *Thread* T's write buffer contains an unlock value. It follows that `lockcell` must still be locked, and therefore no *Thread* can acquire the lock. However, thread T has released the lock. Now either the unlock value must eventually flush by itself, and we will then be in Case 1 above, or T can attempt to re-acquire the lock, in which case the first action of the *Exchange* will be to flush the write buffer. □

We thus obtain the first key property of the lock implementation, that the lock, from an initially unlocked position, flips between the locked and unlocked states, and because of coherence, no set of threads can disagree about which thread was responsible for each lock and unlock operation in that sequence. In other words, two distinct threads never think that they hold the lock at the same time.

Synchronisation. We now turn to the second key property, that the lock acquisition and release operations collectively provide synchronisation. More precisely, following the acquisition of a lock, a thread should have an identical view of the shared memory to that of the unlocking thread at the immediately preceding lock release. This is not an immediate result, since the two threads may be different, and the view of the shared memory is mediated by the write buffers of the cores that are involved.

Theorem 2. *Synchronisation. Assuming shared-memory accesses are only performed by threads holding a lock, all* Writes *made by a* Thread *between a successful call to `lock` and the following call to `unlock` are visible to the next* Thread *that acquires the lock.*

Proof Sketch: We observe that each write buffer is emptied in a FIFO manner, and that each *Thread* has its own write buffer. *Writes* to shared memory locations within the critical section controlled by the lock are initially buffered in the write buffer for the *Thread* that acquired the lock. The lock release is itself a *Write*, that is placed in the buffer *after* all these *Writes*. We proved above that a new lock is only acquired when the unlocking *Write* is flushed from the buffer. Since the write buffer is FIFO, this implies that all the preceding *Writes* in the critical section must have already been flushed to the *Memory*. Now turning to the lock acquisition, since a lock acquisition involves an *Exchange*, which completely flushes a *Thread*'s write buffer, any *Read* to the shared locations (in case of multiple accesses to one location, the first such) by the acquiring *Thread* must read its value from memory. It follows that synchronisation is obtained. □

4.3 Progress of the Spin-Lock Implementation

When acquiring a lock, the *Thread* calling the `lock` function will loop until it succeeds. Conversely, the `unlock` function will always succeed. Progress of the system depends on two assumptions about the memory system. Firstly, if there are multiple contending *Exchanges*, exactly one will succeed. Secondly, all write buffers will eventually flush their contents to *Memory*. The first assumption is required to allow progress in the case of contending lock-acquires when the lock is free. The second assumption is required to allow progress when one thread has released a lock and other threads are waiting to acquire the lock. Note that if the same thread later wants to acquire the lock, the *Exchange* within `lock` will automatically flush the buffers. This is the progress condition for hardware exchanges that was discussed above.

4.4 Functional Correctness of the Queue Implementation

Recall that the pseudo-code for the queue implementation was shown in Figure 4. We will assume that the queue is only touched by the `qget` and `qput` functions that are defined here, and the underlying lock location is not directly accessed by other code. We first verify that the queue implementation validates the assumptions made in the proof of the spin-lock implementation. First, since the only code that affects `lockcell` are the calls to the `lock` and `unlock` functions, it is easy to see no other code touches `lockcell`. Furthermore, on every control flow path, the `unlock` function is only called after having acquired the lock, it is not called more than once, and it is definitely called before either `qget` or `qput` returns. Moreover, any access to the queue and its fields occurs in program-order between a `lock` and an `unlock` from the same thread. Now we argue that the queue properties are ensured by the operations. There are three key properties:

1. The `qget` operation returns with a value that was previously placed in the queue.
2. A value that is placed once in the queue is never removed twice.
3. **FIFO** order is maintained for the queue.

For 1), the loop in the `qget` function can only be exited by the **break** statement. This means that when the loop exits, we know that the queue is non-empty, and furthermore, that the *Thread* that called `qget` holds the lock. Thus no other thread can remove elements from the queue, and it follows that the `front` operation will be able to find and remove at least one value. For 2), since the `front` operation completes before the lock is released, a value cannot be removed more than once. For 3), since only one thread can execute either `qget` or `qput` at a time, the queue follows sequential semantics. Therefore values are removed in the order that they were put into the queue.

4.5 Progress of the Queue Implementation

Since there is only one lock per queue, there is no possibility of deadlock. Progress depends on the progress properties of the underlying spin-lock implementation, in that if there are multiple contending `qget` and `qput` calls, at least one must succeed. Progress also depend on the fairness of the spin-lock implementation, which is an additional assumption. To see why, consider the case when the queue is empty, and there are one or more `qget` operations contending with a `qput` operation. An unfair implementation could allow a `qget` operation to succeed in acquiring the lock, notice that the queue is empty, release the lock, and immediately acquire the lock again without allowing the `qput` operation to acquire the lock. In this case, the `qput` would be starved, and the system as a whole would be prevented from making progress.

4.6 Functional Correctness of the Farm/Pipeline Skeletons

As defined here, the pipeline and farm skeletons are both streaming operations, with workers applying functional operations to inputs to produce some output. The requirement is that all inputs that are placed in the input streams are processed to produce a corresponding output. Moreover, each input in a stream is processed precisely once. Each worker is associated with an input queue $Q_i = x_1, x_2, \ldots, x_n$ and an output queue Q_o, containing tasks and results, respectively. Each queue is guarded by its own lock, and is shared with one or more other workers. A worker obtains an input task from its input queue, applies its functional operation, and then places the result of the task on the output queue. It is easy to see that the assumptions that we need to ensure the correctness of the queue implementation (the queue is touched only by the `qget` and `qput` methods) are valid for both of the queues. Furthermore, any communication between workers occurs only through these queues.

Let us consider a parallel pipeline $f|g$. From the queue properties, by induction on the initial sequence in the input queue $x_1, x_2, \ldots, x_n$, and assuming that f is finite, we can see that worker f will produce results in the sequence $f(x_1), f(x_2), \ldots, f(x_n)$, and that all inputs will produce a corresponding output. Again, by the queue properties and by induction on this sequence, the next stage will produce the sequence $g(f(x_1)), g(f(x_2)), \ldots, g(f(x_n))$. That is,

$\forall j, 0 < j \leq n.Q_o[j] = g(fQ_i[j])$, and so the output queues Q_o represents a map of $g.f$ over the elements in the input queues Q_i.

Now let us consider a task farm $\Phi(f)$. Suppose that the initial sequence in the input queue is $x_1, x_2, \ldots, x_n$. By the queue properties, each element is removed exactly once, and assuming that f is finite, all workers will complete and put results $f(x_i)$ on the output queue. However, without further limitations, the order of the results in the output queue will now depend on the scheduling of each of the worker threads. Unlike the strong ordering for the pipeline, here we can only say that the output queue will contain $f(x_1), f(x_2), \ldots, f(x_n)$ in some permutation. It is not possible to make a strong statement about result ordering without further knowledge of the thread scheduler.

4.7 Progress of the Farm and Pipeline Skeletons

Since both the farm and the pipeline skeletons do not contain any high-level cycles, they satisfy progress. The only possible violations of progress are then when the queue operations are called, with either blocking or deadlocks. We have proved that the queue operations satisfy progress above (assuming a fair spin-lock implementation), and thus proving progress under the same assumption here is almost trivial. The ease of this argument emphasises the advantages of taking a structured view.

5 Timing Models for *x86-64* Multicores

Our overall objective is to obtain good timing predictions for parallel code running on *x86-64* multicores, based on a rigorous understanding of the underlying relaxed memory model. In order to construct our timing cost models, we will use *traces* to describe the operation of each thread in the system, and then abstract over these to determine the overall timing behaviour of the system. A *trace* describes an actual execution of a thread in terms of the observable primitive operations that it performs. Since we are not interested in the values that our system produces, but only in the time that it takes to execute, we abstract over the actual computations that a thread performs between each memory access, using the abstract *Compute* operation to capture the time taken by actual computations. This paper considers only the costs associated with memory accesses. In the actual hardware, memory access costs depend on a variety of factors such as the presence or absence of a location in cache, the interconnect topology, the relative speeds of the various cache levels of the cache, etc. There are, thus, varying levels of realism that can be included in the cost model. We begin with a relatively naïve model, which just considers buffer sizes. As we will see, even a simple model like this already captures most of the important timing effects that we are interested in.

5.1 Simple Average Timing Model

Our first timing model approximates execution costs by assigning an average time to each kind of access. Our parameters are:

T_{Read}	the (average) time to read a location
T_{Write}	the (average) time to write a location
T_{Exchange}	the (average) time to exchange a location

We build up our model in stages, starting with the spin-lock implementation from Figure 5, then considering the queue implementation from Figure 4, and finally extending our model to the high-level farm and pipeline skeletons.

Spin-lock Timings. The cost of an `unlock` operation is just T_{Write}, while that of a `lock` operation is $N \cdot T_{\mathit{Exchange}}$, where N is the number of times that a thread spins before acquiring the lock. Since each lock can only be held by one thread at a time, if a thread attempts to acquire a lock while it is held by another thread, it will spin uselessly. Once the lock is released, if t threads are all trying to acquire the lock, only one will succeed. Assuming that the hardware allows precisely one thread to exchange successfully, it follows that it will take $t \cdot T_{\mathit{Exchange}}$ time before the lock is acquired by some thread.

Queue Timings. Suppose that there are n threads accessing the queue. In the worst case, they will all contend with each other. By the argument above, a `qput` operation therefore succeeds in time: $T_{\texttt{qput}} = n \cdot T_{\mathit{Exchange}} + T_{\mathit{Write}} + T_{\mathit{Write}}$, i.e., $n \cdot T_{\mathit{Exchange}}$ for the lock; one T_{Write} when adding to the queue; and another T_{Write} for the unlock. Similarly, *provided that the queue is not empty*, a `qget` operation succeeds in time $T_{\texttt{qget}} = n \cdot T_{\mathit{Exchange}} + T_{\mathit{Read}} + 2T_{\mathit{Write}}$ i.e., $n \cdot T_{\mathit{Exchange}}$ for the lock; one T_{Read} when reading the head of the queue; and two T_{Write}, one for writing the queue head and one for the unlock.

Farm and Pipeline Skeleton Timings. For the *farm* skeleton, suppose there are n worker threads. The work done by each thread is a `qget` followed by the actual computation and finally a `qput` operation. Each of the queue operations has contention $n + 1$. Thus, each thread takes time $T_{\texttt{qget}} + T_f + T_{\texttt{qput}}$ which simplifies to $2 \cdot (n + 1) \cdot T_{\mathit{Exchange}} + 4 \cdot T_{\mathit{Write}} + T_{\mathit{Read}} + T_f$. For the two-stage *pipeline* skeleton, there are two cases depending on which of the two stages dominates the time taken (the other being idle). If the first stage dominates (qualitatively, the second stage has insufficient work), then the first queue has contention $|f|+1$, the middle queue has contention $|f|+1$, and the second queue has contention 1. The first stage thus takes time $T_{\texttt{qget}} + T_f + T_{\texttt{qput}}$ which simplifies to $2 \cdot (|f| + 1) \cdot T_{\mathit{Exchange}} + 5 \cdot T_{\mathit{Write}} + T_f$. while the second stage takes time T_g. The total time is the sum of the times for the two stages $2 \cdot (|f| + 1) \cdot T_{\mathit{Exchange}} + 5 \cdot T_{\mathit{Write}} + T_f + T_g$. Conversely, if the second stage dominates (qualitatively, the second stage is saturated), then the first queue has contention $|f|+1$, the middle queue has contention $|f| + |g| + 1$, and the second queue has contention 1. The first stage takes time T_f, and the second takes $T_{\texttt{qget}} + T_g + T_{\texttt{qput}}$ which simplifies to $(2 \cdot |g| + |f| + 1) \cdot T_{\boldsymbol{Exchange}} + 5 \cdot T_{\boldsymbol{Write}} + T_g$ Again, the total time is the sum of the times for the two stages $T_f + (2 \cdot |g| + |f| + 1) \cdot T_{\boldsymbol{Exchange}} + 5 \cdot T_{\boldsymbol{Write}} + T_g$.

5.2 Including Store Buffer Flushing

A more realistic model would take into account that an exchange operation has two components: i) flushing the local store buffer, $T_{\boldsymbol{Fl}}$; and ii) the actual exchange operation $T_{\boldsymbol{JustX}}$. Naturally, the first component depends on the number of entries in the local buffer that are waiting to be flushed, b. It follows that $T_{\boldsymbol{Exchange}} = b \cdot T_{\boldsymbol{Fl}} + T_{\boldsymbol{JustX}}$.

Spin-lock Timings. Notice that the exchange operation is only used in the context of a tight loop in the lock operation. The first time, the store buffer must be flushed, but each successive times, the store buffer is already empty. The price of a flush is thus paid only once. Assuming that t threads contend for a lock as before, and with the same assumption as before that a single exchange succeeds, the cost of a `lock` operation is $b \cdot T_{\boldsymbol{Fl}} + t \cdot T_{\boldsymbol{JustX}}$. Here, b is the (average) number of writes waiting in the buffers, which is between 0 and the size of the store buffer, B. The `unlock` operation has the same cost as in our first model.

Queue Timings. The `qput` operation now takes time: $b \cdot n \cdot T_{\boldsymbol{Fl}} + n \cdot T_{\boldsymbol{JustX}} + T_{\boldsymbol{Write}} + T_{\boldsymbol{Write}}$, while the `qget` operation takes time: $b \cdot n \cdot T_{\boldsymbol{Fl}} + n \cdot T_{\boldsymbol{JustX}} + T_{\boldsymbol{Read}} + 2 \cdot T_{\boldsymbol{Write}}$.

Farm and Pipeline Skeleton Timings. A one-stage farm takes: $2 \cdot (n+1) \cdot b \cdot T_{\boldsymbol{Fl}} + (n+1) \cdot T_{\boldsymbol{JustX}} + 4T_{\boldsymbol{Write}} + T_{\boldsymbol{Read}} + T_f$. For a two-stage pipeline, the same analysis as before applies. If the first-stage dominates, the time is: $2 \cdot (|f| + 1) \cdot b \cdot T_{\boldsymbol{Fl}} + (2 \cdot |f| + |g| + 1) \cdot T_{\boldsymbol{JustX}} + 5 \cdot T_{\boldsymbol{Write}} + T_f$ and if the second-stage dominates, the time is: $(2 \cdot |g| + |f| + 1) \cdot b \cdot T_{\boldsymbol{Fl}} + (2 \cdot |f| + |g| + 1) \cdot T_{\boldsymbol{JustX}} + 5 \cdot T_{\boldsymbol{Write}} + T_g$.

5.3 Predicted Speedups

Speedup for Farms. We now use the cost model above to predict execution speedups for some example skeletons. First consider a simple one-stage farm with n threads uniformly doing T_f work. The computation takes time $n \cdot T_f$, which we can consider to be constant when calculating speedup. The total cost is as calculated for farms in §5.2. Assuming that T_f is sufficiently large to dominate the time of a single *Read*, *Write*, or pure *Exchange*, the predicted speedup is approximately: $\text{Speedup}_{\Phi} \simeq \frac{n \cdot W}{1 + c \cdot n^2}$, where W is a constant that depends on the total cost of computation, and c is a constant that depends on the number of times that the lock is taken.

Speedup for Two-Stage Pipelines. Now consider a two-stage pipeline with f first-stage workers and g second-stage workers, each uniformly doing work T_f and T_g, respectively. Since the total work is constant, if the first stage is fully occupied then the predicted speedup will be: $\text{Speedup}_{f|g} \simeq \frac{f \cdot W_1}{1 + c_1 \cdot f}$ (*eqn.* 1), where W_1

is a constant that depends on the total cost of computation done in stage one (and can be approximated by the cost of a single thread in the first and a single thread in the second stages), and c_1 is a constant that depends on the number of times that the lock on the first queue is taken. If the second stage is saturated, then the predicted speedup will be: $\text{Speedup}_{f|g} \simeq \frac{f \cdot W_2}{1+c_2 \cdot (f+g)}$ (*eqn.* 2), where again W_2 is a constant depending on the total cost of computation, and c_2 is a constant depending on the number of times the lock on the second queue is taken. Constant W_2 will be T_g/T_f times the constant W_1 above (and thus constant for a particular application). We will switch between which stage dominates precisely depending on which of the values from eqn. (1) or (2) is lower, i.e. precisely when the second stage becomes saturated. The final speedup is thus: $\text{Speedup}_{f|g} \simeq \min\left(\frac{f \cdot W_1}{1+c_1 \cdot f}, \frac{f \cdot W_2}{1+c_2 \cdot (f+g)}\right)$.

6 Experimental Validation

We evaluate our cost models against real execution costs using a number of different benchmarks, running on three different *x86-64* multicores. The main system that we use (*titanic*) is a 2.4GHz 24-core, AMD Opteron 6176 architecture, running Centos Linux 2.6.18-274.e15, and gcc 4.4.6. We also use *ladybank*, a 2.93GHz 8-core (plus hyperthreading) Intel Xeon X5570 architecture, running GNU/Linux 2.6.32-358.6.2.el6, and *lovelace*, a 2.3GHz 64-core, AMD Opteron 6376 architecture, running GNU/Linux 2.6.32-279.22.1.el6, which we used to test scalability for a limited set of experiments. All the results reported here are averages of 10 runs on an idle machine (but not in single-user mode). All speedups are absolute results against the original sequential versions.

6.1 Image Convolution

Image convolution is widely used in image processing applications such as blurring, smoothing or edge detection. The convolution algorithm is a composition of two functions $r \circ p$, where r reads in an image from a file and p processes the image by applying a filter. This process is typically applied to a stream of input images, and produces a stream of output images. For each pixel in the input image, the filtering stage consists of computing the scalar product of the filter and the window surrounding the pixel:

$$output\_pixel(i,j) = \sum_m \sum_n input\_pixel(i-n, j-m) \times filter\_weight(n,m)$$

For our benchmark tests, we parallelise the convolution using a two-stage pipeline, where each stage is a farm: $\Phi_1(r) \mid \Phi_2(p)$. Speedup results for two different image sizes on the 24-core *titanic* machine are shown on the left and centre of Figure 6, with each worker in either farm allocated to its own core. Speedups are recorded against the sequential version of the algorithm. The solid lines show the actual execution speedups and the dashed lines show the speedup predictions. The x

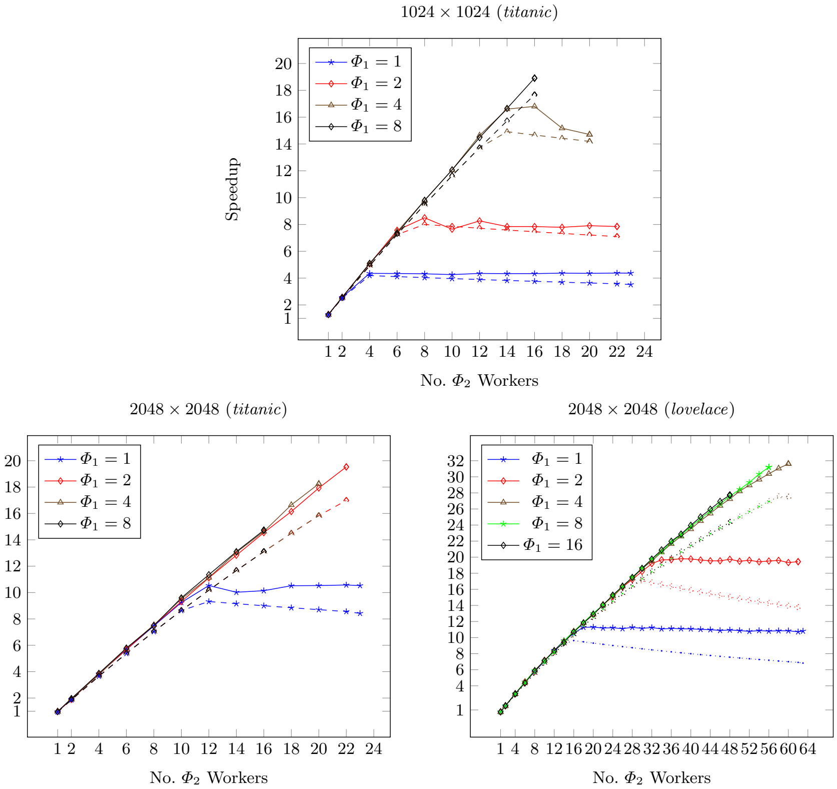

Fig. 6. Image Convolution ($\Phi_1(r) \mid \Phi_2(p)$) on *titanic* and *lovelace*. Dashed/lighter lines are predictions. Note that the number of $\Phi_1 + \Phi_2$ workers $\leq$ total hardware threads (24 for *titanic*/64 for *lovelace*).

axis is the number of Φ_2 workers and each line corresponds to a fixed number of Φ_1 workers. In all cases, the predicted speedups closely match the actual results, giving a correct lower bound prediction. For the 1024 × 1024 images, the best speedup is 18.90 for $\Phi_1 = 8$ and $\Phi_2 = 16$ workers. Our cost model predicts a speedup of 17.65 (within 7% of the actual value). We observe three *knees* in the graph, where the speedups flatten out: for $\Phi_1 = 1$ and $\Phi_2 = 4$; or $\Phi_1 = 2$ and $\Phi_2 = 6$; and at $\Phi_1 = 8$ and $\Phi_2 = 14$. In all cases, our cost models correctly predict both the knee and the speedup. For the 2048 × 2048 images, the best speedup is 19.53 for $\Phi_2 = 22$ and $\Phi_1 = 2$, where our cost model predicts a speedup of 16.98 (within 14% of the actual value). We observe one *knee*, at $\Phi_1 = 1$ and $\Phi_2 = 12$, which our cost model also correctly predicts. The righthand graph in Figure 6 investigates the scalability of our approach, showing speedup results for 2048 × 2048 images on the 64-core *lovelace* machine. The best speedup that we obtain, at $\Phi_1 = 4$ and $\Phi_2 = 60$, is 31.6, versus a predicted speedup of 27.48 (that

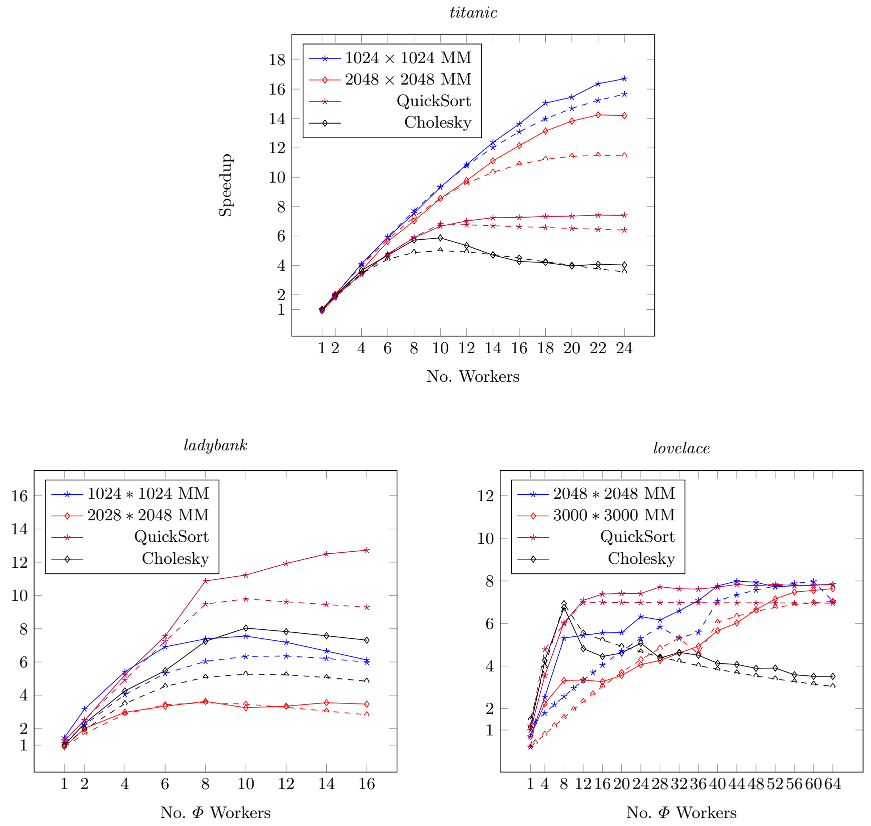

Fig. 7. Speedups for Cholesky Decomposition, Matrix Multiplication and Quicksort, using single task farms, on *titanic*, *ladybank* and *lovelace*. Dashed lines are predictions.

is, within 14% of the actual value). Speedup predictions for the remaining number of Φ_1 workers (4,8 and 10) are consistent with the actual speedups, showing that the cost model correctly predicts identical speedup in all three cases.

6.2 Cholesky Decomposition

Cholesky Decomposition is used in linear algebra, comprising the decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. Our implementation uses a task farm, Φ_1, to model the decomposition stage. Speedup results for a 1024 × 1024 matrix are shown in Figure 7. The top set of results shows speedups for the 24-core *titanic* machine (an AMD architecture) and the bottom set of results shows the corresponding speedups on the 8-core (*hyperthreaded*) *ladybank* machine (an Intel Xeon architecture). For *titanic*, the best actual speedup is 5.86 for 10 workers, versus a best predicted speedup of 5.0 (also for 10 workers). The cost model also gives an almost perfect prediction, giving a lower bound between 6-12 workers. For *ladybank*, the best actual speedup is 8.03 for 10 workers, versus a predicted speedup of 5.26 for 10 workers. This is less accurate than for the AMD architecture, perhaps showing some additional complexities in the memory model that would repay further investigation. The cost model does, however, correctly predict a lower bound on the speedup in all cases.

6.3 Matrix Multiplication

The product of two matrices, A and B is defined as:

$$(AB)_{i,j} = \sum_{k=1}^{m} A_{ik}.B_{kj} \tag{1}$$

As for Cholesky decompostion, this can also be implemented using a task farm. with Figure 7 giving speedup results for 1024 × 1024 and 2048 × 2048 matrices on both *titanic* and *ladybank*. For *ladybank*, the 1024 × 1024 example gives a speedup of 7.55 on 10 cores, versus a prediction of 6.33. Once again, the cost model correctly predicts lower bounds on speedup, predicting, for example, a speedup of 5.98 on 16 cores versus an actual speedup of 6.12. For the 2048 × 2048 execution, the best speedup is 3.54 on 14 cores, versus a predicted speedup of 3.04. For *titanic*, the 1024 × 1024 example gives a best speedup of 16.7 on 24 cores versus a predicted speedup of 15.65. Likewise, for the 2048 × 2048 example, we obtain a best speedup of 14.19 versus a predicted speedup of 11.46. In both cases, overall speedups are better for smaller matrices. This is due to lower communication costs. As with the Cholesky example, in all cases our cost models correctly predict a lower bound on speedup. In the best case, the prediction is within 3% of the actual speedup.

6.4 QuickSort

Figure 7 shows speedup results for a *divide-and-conquer* implementation of the classical QuickSort algorithm mapped to a task farm. All instances sort the same 10^9 element list of randomly generated integers, with a threshold size of 10^5. The best actual speedup on *ladybank* was 12.72 versus a predicted speedup of 9.29. For 8 and 10 workers, we observe a super-linear speedup of 10.87 for 8 workers and 11.2 speedup for 10 workers. While we do not have a precise explanation for this, our experiments showed that the effect is repeatable, and is presumably therefore some hardware effect. What is important is that our cost model predicts a lower bound correctly, showing a similar curve with a knee at 8 workers, as in the actual executions. For *titanic*, speedup increases well from 1 to 10 workers, with an actual speedup of 6.6 on 10 workers versus a predicted speedup of 6.8 – one of the few cases where the cost model yields a (slight) over-estimate of the speedup. Beyond this point, speedup still improves, but only slightly, up to a maximum of 7.4 speedup on 24 cores (versus a prediction of 6.4). As with the other examples, the cost model closely predicts the actual speedup.

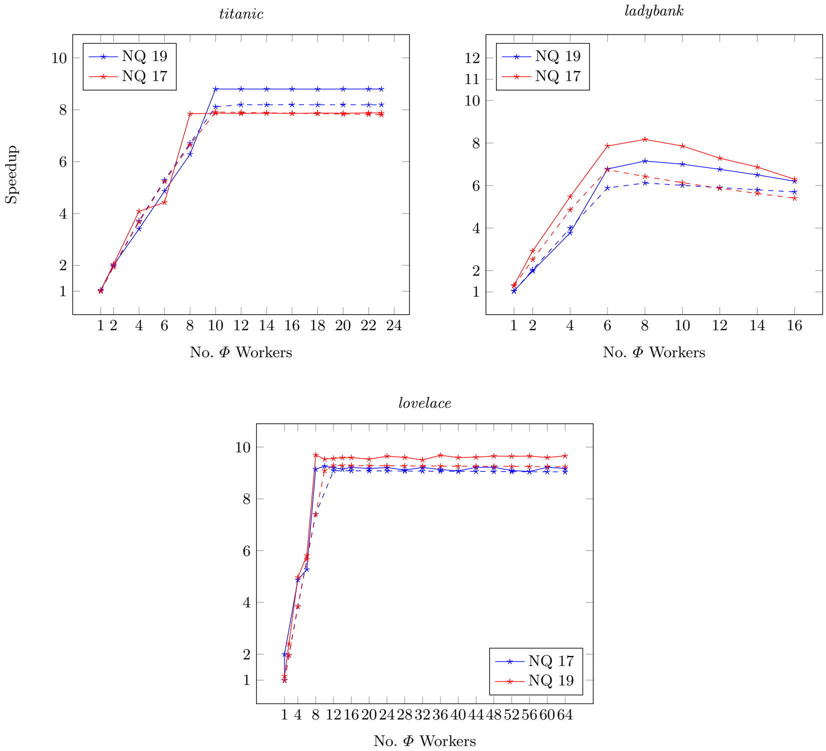

Fig. 8. Speedups for NQueens ($g \mid \Phi(s)$) on *ladybank* and *titanic*. Dashed lines are predictions.

6.5 NQueens

NQueens involves placing n queens on an $n * n$ chessboard, so that no two queens may attack each other, according to the usual rules of chess. The solution requires that no two queens occupy the same row, column or diagonal. In our implementation, we have modelled the NQueens problem as a two-stage pipeline, $g \mid \Phi(s)$, where g is the stage that generates all positions of the queens, and s is the stage that solves the position. Speedup results for this implementation of NQueens are shown in Figure 8 for a 15×15 board, which produces 15 possible positions in the first stage, and for a 19×19 board, which produces 17 possible queens in the first stage. As with the other examples, the predicted speedups closely model the actual speedups. There is one over-prediction: for 6 workers on the 19×19 board, where we observe a discontinuity in the actual speedup. Otherwise, the predictions closely mimic the actual results that we obtain. The best speedups are 8.8 for 10 workers on a 19×19 board and 7.87 for 10 workers on a 17×17 board.

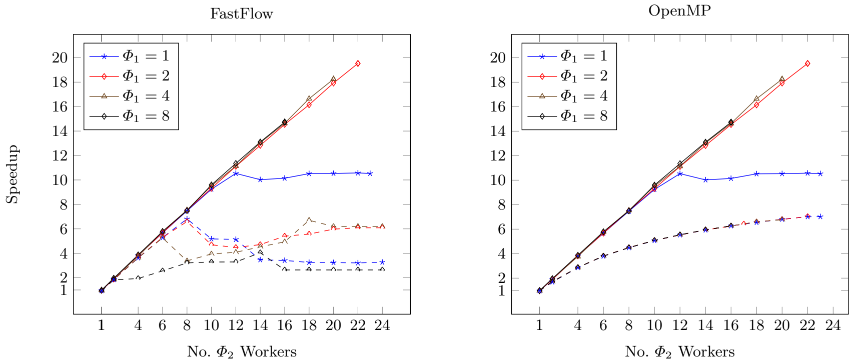

Fig. 9. Speedups for our implementation (solid lines) versus FastFlow (above, dashed lines), and OpenMP (below, dashed lines); Convolution of 500 images, size 2048 × 2048 (*titanic*). The baseline sequential performance is identical in each case.

6.6 Comparison with other Techniques

In order to demonstrate the efficiency of our queue and locking implementations, we have compared speedups for the Image Convolution example against those for two other state-of-the-art parallel implementations: OpenMP [9] and FastFlow [10]. Figure 9 compares all three implementations using 2048 × 2048 images on *titanic*. The top graph compares our implementation against FastFlow, and the bottom against OpenMP. In both cases, the speedup from our implementation is shown using solid lines, and the dashed lines show the speedups for FastFlow/OpenMP. All speedups were measured against the same sequential implementation (which took 768.75 seconds). In order to correctly compare the parallelism structures, we implemented the same two-stage pipeline in all three systems, using equivalent *farm* and *pipeline* structures in FastFlow, and two dynamic **for**-loops separated by a barrier synchronisation in OpenMP. The barrier is necessary to avoid the second stage processing images that have not yet been generated by the first stage, and is a natural translation of the parallelism structure that is used in the other implementations. As Figure 9 shows, the FastFlow implementation is comparable to ours up to 8 workers for Φ_2 and for all versions of Φ_1, but speedup reduces drastically after this point, whereas our implementation scales well up to 23 cores. The FastFlow implementation also carries further overhead in the form of two additional dedicated cores per task farm, which is not needed in our implementation. The OpenMP implementation scales better than FastFlow, but begins to flatten out at about 6 Φ_2 workers, where our implementation continues to scale well to 23 cores.

7 Related Work

Timing issues are critically important for parallel and concurrent execution, and they have therefore been widely studied in the literature. However, despite their prevalence in real hardware, very little work considers relaxed memory models. This paper represents the first attempt of which we are aware to consider the precise impact of relaxed memory models on functional correctness, deadlock and timing. Correctness of the spin-lock protocol on x86-TSO has been previously proved using a semantic criterion (*triangular-race freedom* [11]) derived from the *x86*-TSO model used here [6,5]. In contrast, our new proof proves functional correctness and deadlock freedom directly over the operational model of the actual *x86-64* instructions [6,5]. Our proof has been composed with a proof for queues and thence skeletons, and furthermore clearly isolates necessary hardware assumptions for progress. We also consider, for the first time, the crucial issue of execution time. Burckhardt *et al* [12] verify spinlocks on TSO by adapting a modified version of linearisability, and thus implicitly obtain compositionality. However, they ignore the progress and fairness constraints that we uncover, and also do not treat execution time.

Specific algorithmic skeletons are frequently associated with timing cost models [13]. However, these are obtained through measurement rather than being systematically derived from machine-level models, as here. Much of the work on developing cost models for parallel execution has focused on data parallelism. The Parallel Random-Access Machine (PRAM) execution model [14] acts as a theory of complexity for parallel algorithms on idealised shared-memory SIMD machines. In the basic PRAM execution model, basic computations and shared-memory accesses are both assumed to take unit time. Unfortunately, PRAM costs underestimate actual machine execution costs, but in an unpredictable way [15]. The Bulk Synchronous Parallel (BSP) model [16] extends the PRAM model in a more realistic way, introducing a synchronising communication step after each set of computation steps. Lisper [17] has investigated the use of a Bulk Synchronous Parallel skeleton for determining worst-case execution times, but only informally, and not in the context of real processor models. Skillicorn and Cai have likewise developed a high-level cost calculus for data parallel computations [15], based on the *shape* of data structures, and known properties of primitive parallel operations, but have not based this on a strong machine-level semantics. Blelloch and Greiner have demonstrated provable time and space bounds for nested data parallel computations in NESL [18]. To date, none of these models have therefore been derived from first principles for real multi-core architectures with relaxed-memory models. The model we give here thus represents a significant step in determining accurate cost models for algorithmic skeletons and other structured parallel forms on modern processor architectures.

The C/C++11 standards provide atomic operations that support various kinds of weak memory models [19,20]. They could be used to implement the *Fence* and *Exchange* operations, but do not support higher-level parallel structures, such as the structured algorithmic skeletons that we have used here.

Boehm and Adve [21] consider the foundations of this concurrency model. Adve and Boehm [22] give a useful survey of weak memory models as of 2010.

8 Conclusions

This paper has developed new proofs of functional correctness for *x86-64* multicores and used these to derive cost models for structured parallel programs. For the first time, we have direct operational proofs of functional correctness and deadlock freedom for standard locking and queueing algorithms under the relaxed memory model used by common *x86-64* multicores. We have used these to build accurate cost models for parallel programs that are structured using common algorithmic skeletons. Our predictions match very closely to actual results: in most cases predicting a lower-bound to within 7% accuracy.

8.1 Limitations/Further Work

There are a number of obvious limitations to the work described here that would repay further work. Firstly, the queues that we have used here are unbounded. Extending our work to consider bounded queues should not be technically difficult, but would add some complexity to the queue definitions and would also require us to consider how "back-pressure" [23] from the demand on one queue impacts the production of values that feed that queue. Secondly, the locking mechanism that we have used here carries some cost. By definition, when a processor is executing a spin-lock, it is wasting energy and not doing any useful work. More efficient notification or "lock-free"[2] techniques would obviate this at the cost of a significantly more complex proof and rather more complex cost model. We have therefore chosen not to do this here. Thirdly, and more seriously, as with most work on algorithmic skeletons, we have not considered any form of feedback, as in FastFlow's "farm/pipeline-with-feedback" skeletons [10]. Incorporating skeletons with feedback would allow the construction of more complex parallel programs, but would require us to determine fixed-points in the skeletons and to solve the resulting timing recurrence relations. Fourthly, we have ignored some hardware effects. In particular, while we have accounted for the behaviour of the cache and associated hardware when determining functional correctness and deadlock properties, we have not attempted to precisely model memory access behaviour in determining execution times, but have assumed that the sequential cost model properly accounts for such costs. We have also not modelled the processor instruction pipeline. Neither issue will affect functional correctness, but could, obviously, impact low-level timing accuracy.

References

1. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge, MA, USA (1991)

[2] In the sense that locks are not visible to the programmer, rather than they are not used by the hardware.

2. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw., Pract. Exper. **40**(12) (2010) 1135–1160
3. Danelutto, M., Torquati, M.: A RISC Building Block Set for Structured Parallel Programming. In: 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '13
4. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers **C-28**(9) (September 1979) 690–691
5. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.O.: x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. CACM **53**(7) (July 2010) 89–97
6. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: Proc. TPHOLs '09. (August 2009) 391–407
7. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1, §8.2.2. Intel (2013)
8. Papamarcos, M.S., Patel, J.H.: A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories. In: Proc 11th Annual International Symposium on Comp Arch. ISCA '84, New York, NY, USA, ACM (1984) 348–354
9. Chapman, B., Jost, G., Pas, R.v.d.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press (2007)
10. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-level and Efficient Streaming on Multi-core. In: Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana. (2012) 13
11. Owens, S.: Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: Proc. ECOOP '10, Springer (June 2010) 478–503
12. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent Library Correctness on the TSO Memory Model. In: Proc. ESOP '12. (2012) 87–107
13. Hamdan, M.M.: A Survey of Cost Models for Algorithmic Skeletons. Technical report, Heriot-Watt University (1999)
14. Fortune, S., Wyllie, J.: Parallelism in Random Access Machines. In: Proc. STOC '78. STOC '78, ACM (1978) 114–118
15. Skillicorn, D.B.: A Cost Calculus for Parallel Functional Programming. Journal of Parallel and Distributed Computing **28** (1995)
16. Valiant, L.G.: A Bridging Model for Parallel Computation. Commun. ACM **33**(8) (August 1990) 103–111
17. Lisper, B.: Towards Parallel Programming Models for Predictability. In: WCET '12. Volume 23 of OpenAccess Series in Informatics (OASIcs). (2012) 48–58
18. Blelloch, G.E., Greiner, J.: A Provable Time and Space Efficient Implementation of NESL. In: Proc. ICFP '96. (1996) 213–225
19. Becker, P., ed.: Programming Languages — C++. ISO/IEC (2011)
20. Williams, A.: C++ Concurrency in Action: Practical Multithreading. Manning Publications (2012)
21. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. SIGPLAN Not. **43**(6) (June 2008) 68–78
22. Adve, S.V., Boehm, H.J.: Memory Models: A Case for Rethinking Parallel Languages and Hardware. CACM **53**(8) (2010) 90–101
23. Collins, R.L., Carloni, L.P.: Flexible filters: load balancing through backpressure for stream programs. In: EMSOFT. (2009) 205–214