Ownership of a queue for practical lock-free scheduling

Lincoln Quirk

May 4, 2008

Abstract

We consider the problem of scheduling tasks in a multiprocessor. Tasks cannot always be scheduled independently – for example, only one task associated with a particular object can be run at once. How can a scheduler decide which tasks to run under this constraint? We describe a novel OwnerQueue data structure, a single-dequeuer lock-free queues which allow for the smooth handoff of dequeuing privilege from processor to processor. One OwnerQueue is allocated per object, and tasks on the object are enqueued to it. Only the owner of the queue may run tasks. The OwnerQueue itself ensures two safety properties: no more than one thread considers itself the owner at any one time, and at least one thread considers itself the owner if there are any items on the queue. With these guarantees we construct a scheduler satisfying the constraints.

1 Introduction

Most programming languages are based around the idea of a process, which comes into existence, does some work, and exits. However, this model does not work well for event-driven server systems: the process must instruct the computer to listen and respond to events, and the code to do this tends to be difficult to write and maintain if the programmer is using a process model. Furthermore, processes have a lot of state, and so when multiple threads of execution exist within a process (as is often created by asynchronous events in event-driven systems), synchronization of this state between threads is also difficult to write and maintain. Lastly, requiring the programmer to specify which processes run on which processors (or threads) requires the programmer to predict beforehand how much work a particular process will require.

A new programming paradigm can make steps towards solving these issues. We have designed a programming language based not around processes but around *units*, which are an aggregation of data (which must be consistent with itself) and a set of *operations* (code which operates on the data). Units communicate only by sending asynchronous messages to other units, telling them to execute operations.

Since the messages are asynchronous between units, the system is free to schedule non-conflicting operations in parallel. Due to this scheduling freedom, the system exhibits natural parallelism at the granularity of units. However, to preserve consistency within a unit, the system is constrained never to schedule two operations on a particular unit at once.

2 Units and Operations

The programming language is designed around asynchronous message passing between units. The runtime system must keep track of units and *messages* as they flow through the system. A message is addressed to an operation on a unit, may include arguments to control the behavior of the operation, and results in the asynchronous execution of that operation.

Units are somewhat similar to objects in object-oriented programming. Users define a set of shared state for each unit, and the system guarantees to the user that all operations on a unit are linearizable[3], and to make it easier to create programs using the system, the user need not think about synchronization in the presence of concurrent requests at all. The user simply defines an operation as a series of steps, possibly including sending other messages, as well as reads and writes to memory local to the unit.

There are a few ways to linearize all unit operations. In future work, we will examine automatic linearization of operations using software transactional memory[4] or other similar ideas. For now, however, we will require the scheduler to execute at most one operation on a particular unit at one time. This property is called *unit exclusion.*

Not all operations need to be part of a unit: if the operation doesn't read or write any shared state, then it is known as a *free operation* and may be executed in parallel with any other operation.

3 Multithreaded computation model

To provide a formal framework in which to specify this system, consider the model of multithreaded computations given by Blumofe and Leiserson [1]: computations are split into a series of *steps*, some of which may be executed in parallel. A *task* is a series of steps. Each step is equivalent to a machine language instruction. After it is executed, it may either *continue*, going to the next step in the sequence; *spawn*, starting a new asynchronous task; or *join*, wait for a step in another task to be completed. A program is a directed graph, where the steps are nodes, connected by continue, spawn, or join edges.

Figure 1 shows a typical multithreaded computation. The first task spawns at step v0 (using a dotted spawn edge) an asynchronous task, whose value it accepts in step v3 using a join edge. Figure 2 shows a non-strict computation for contrast.

This (Blumofe's) framework requires that all multithreaded computations are fully strict, meaning that all join edges from a task go to the task's parent.

Here, the design of the programming language suggests a different, incompatible constraint on computations: all join edges to a task go to the first step in the task. I call this property *forwardesque*, because with the additional condition of acyclicity in the graph, it means that all edges are forward – they go from an earlier step in the computation to a later one. For the purposes of the programming language, this means that tasks do not block once they have started execution.

This is desirable: if a task is a synchronized operation on a unit, because of unit exclusion, it can be thought of as holding a monitor or lock on the unit for its duration. In thread programming, it is discouraged to block while holding a lock, because no other thread can take the lock even while the thread holding it is waiting. In our system, since tasks have no way to block midway through, this is not a problem.

Figure 3 shows a typical forwardesque computation. The main task spawns two asynchronous tasks (v4-v7 and v8-v11). It then spawns a receiver task, which waits for the result of the values of the two asynchronous tasks (indicated by the two join edges) before it executes.

The user in our system creates forwardesque computations using Futures.

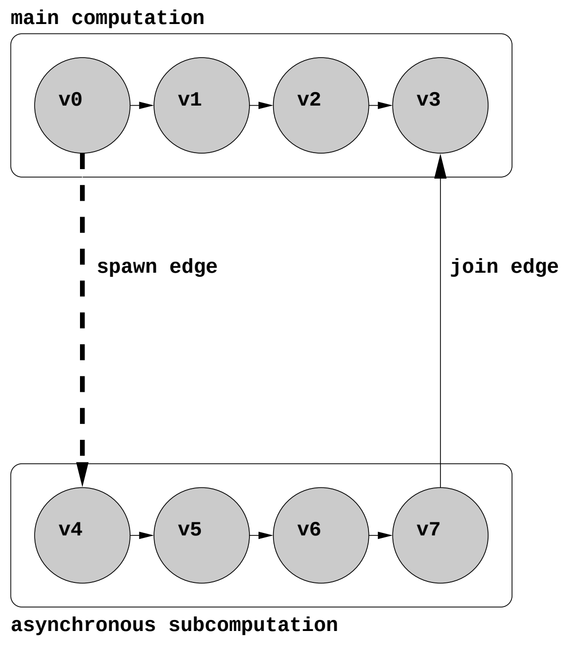

Figure 1: Fully strict computation.

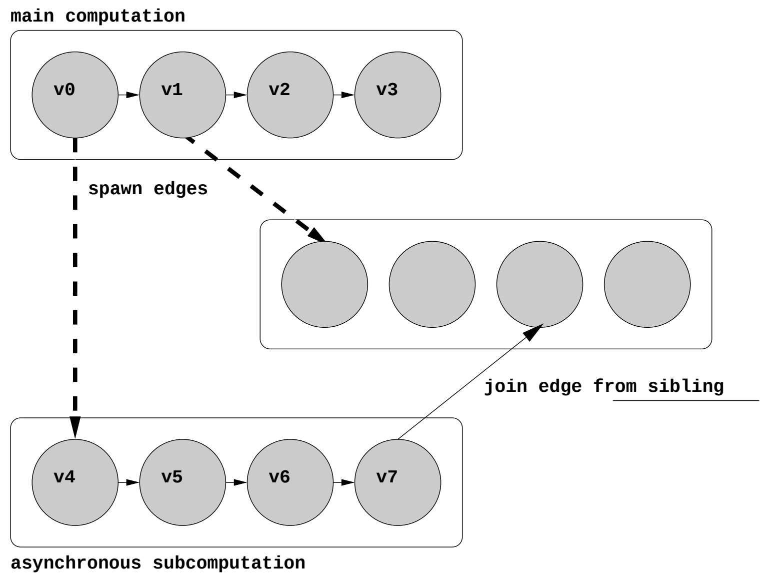

Figure 2: A non-strict multithreaded computation. Note the join edge from a sibling.

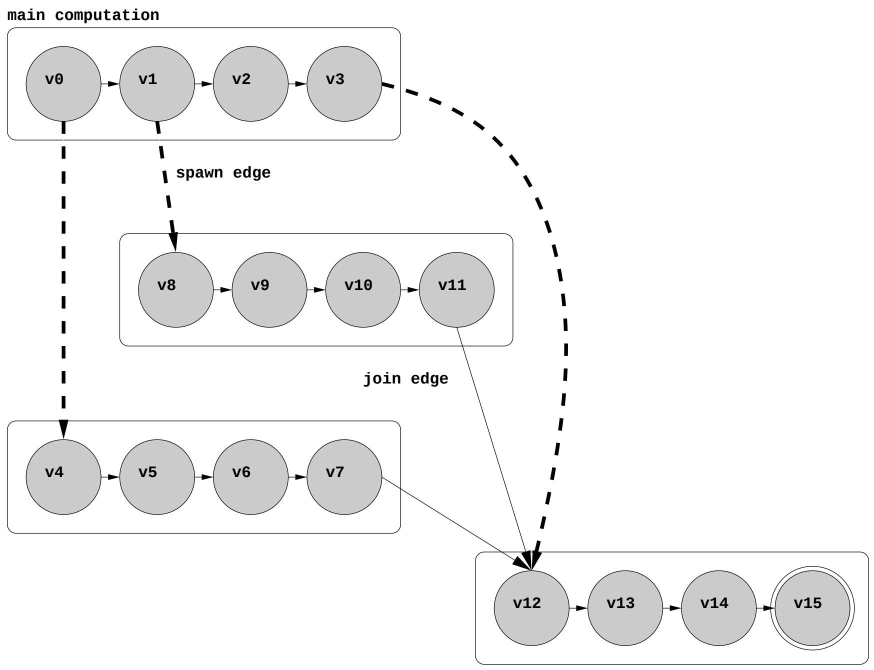

Figure 3: Forwardesque computation.

Our notion of Futures is similar to Halstead's in Multilisp[2]: an operation is started asynchronously, and returns a Future immediately, which is a promise for a later value when that asynchronous task terminates.

The user starts an asynchronous task (sends a message) using the `send` primitive. `Send` accepts a receiver operation and some number of arguments, returns a Future, and causes the receiver to be invoked asynchronously with the arguments. When the receiver returns a value, the Future is filled (resolved) with the value.

In Multilisp, the user may "force" a future during execution. In our system, on the other hand, there is no way to force a Future – no capability for blocking on Futures during execution of an operation. In order to get the value from a Future, the user must perform `send`, passing the Future he wishes to resolve to *another asynchronous task* as an argument. That task will not begin until the Future that was sent as an argument is resolved. In fact, all reachable Futures, e.g., embedded in data structures, returned from operations, etc., must be resolved before the task can begin.

In this system, Futures are the only way to create join edges. The only way to wait on a Future is to spawn a new task to receive its value. That task will have some number of join edge to its first node, but there are no other join edges, so the computation is forwardesque.

Theorem 1. *Computations in the system are acyclic and therefore deadlock-free, assuming that every task terminates and a task can never start with a reference to an unresolved Future (a join edge).*

Proof. If each task terminates, there are trivially no cycles among continue or spawn edges alone. Thus any cycle must include a join edge.

Assume there is a path from task T to itself, including a join edge to T. The source of the join edge cannot be T itself, because join edges to a task must be created from previously spawned tasks. Similarly, the source of the join edge cannot be a task spawned after T. So it must be a task spawned before T. However, this logic applies to each task T in the cycle, and they cannot all have been spawned before each other in time. Therefore, cycles do not exist. □

Note that this theorem stipulates that computations may not start with a reference to an unresolved Future. Therefore, provisions must be made against the possibility of a Future "escaping" a task unresolved. If this were possible, a task might obtain a reference to its own Future, attempt to return it, and thereby deadlock.

In practice, this restriction (no escaping Futures) may be relaxed without admitting deadlock in some situations. For instance, a Future F_S for task S could escape to task T if T is pure functional (no side effects, no shared state) and therefore does not save F_S anywhere that S could read it.

4 Scheduling specification

In order to execute a program in this system, we use a *scheduler* which selects steps to run among all the steps that are ready. Running a step means executing it on a *processor*. The machine architecture may include any number of processors, and the scheduler is expected to efficiently utilize all available processors in order to complete the computation as quickly as possible. Each processor executes one step per unit of time.

It is possible that there are more runnable steps than processors and yet not all processors can be busy, because some of those steps might be parts of tasks on the same unit, and the unit exclusion requirement prevents the scheduler from concurrently scheduling two tasks on the same unit.

Besides satisfying unit exclusion, the scheduler must also be efficient. It should scale as well as possible to highly concurrent systems with many processors. I believe that lock-free algorithms scale well because they minimize critical sections, so I implemented a lock-free multithreaded scheduler for this system. Other scheduler implementations are not considered here.

5 Scheduler implementation

My scheduler is non-preempting; tasks are executed to completion by following continue edges. A spawn edge leaving a node is considered a message and is placed in a queue. Messages are addressed to a particular operation, which is either a free operation or a unit operation. Messages are enqueued to be executed asynchronously. Join edges are handled by scanning the message arguments for Futures, including deeply traversing data structures. If the message contains unresolved Futures, the system saves the message for when the Futures are resolved rather than enqueuing it right away. (This data structure traversal is fairly expensive for large data structures, and the next system will probably have a slightly different mechanism.)

One approach to lock-free scheduling is to use a global runqueue from which each processor dequeues. This is subject to high contention due to cache misses and interconnect bandwidth on the runqueue, however, so Blumofe describes a model where each processor has its own private runqueue. When there are only free operations ready to run, the scheduler in this paper is very similar to a wait-free implementation of the work-stealing deques scheduler in Blumofe.

Unlike that of Blumofe, our scheduler must be careful to only run one operation per unit. To guarantee unit exclusion, we place unit-specific messages on a unit runqueue, and then place the unit on a processor's runqueue.

Here it is important to consider the scheduler in the presence of *work-stealing*. When a processor is not busy and other processors are, it is usually beneficial for that processor to try to find some work to do. This is accomplished through workstealing. When it would otherwise be idle, the processor scans other processor runqueues and attempts to steal a unit by dequeuing from the first runqueue it comes across with some units on it.

But in the presence of workstealing, a unit that appeared twice on a processor's runqueue might have one of its units stolen. Then it appears twice on two processors' runqueues, and unit exclusion could be broken. Thus any unit must appear at most once on a processor's runqueue. This is done with an abstract data structure called an OwnerQueue, which has a similar interface to a standard queue.

6 OwnerQueue specification

The OwnerQueue structure is an modification of a simple lock-free queue. The idea is that it can notify the user when the queue is empty, but this capability, along with a contract for how the OwnerQueue is used, allows the scheduler to safely transfer ownership of the queue from thread to thread during scheduling.

The OwnerQueue data structure defines two methods: `enq_and_was_empty` and `deq_and_is_empty`. While it approximates the behavior of a queue, the system does not guarantee that items are extracted in exactly the same order they appear to have been inserted.

The sequential specification for `enq_and_was_empty` is the following: Enter the given item into the set of items in the structure, and if the set was empty, return *true*. Otherwise, return *false*.

The sequential specification for `deq_and_is_empty` is the following: Nondeterministically remove an item i from the set of items in the structure. If the set is now empty, return $(i, true)$. Otherwise, return $(i, false)$.

The OwnerQueue provides the user with certain guarantees, provided its contract is met. Critical to the contract is the notion of *ownership* of an OwnerQueue. When a thread receives *true* from `enq_and_was_empty`, that thread becomes the owner of the OwnerQueue. That thread now has exclusive privilege to call `deq_and_is_empty` on the OwnerQueue, until `deq_and_is_empty` returns *true*, at which time the thread's privilege is revoked and it is no longer the owner.

The OwnerQueue guarantees that if the threads obey the OwnerQueue contract, then no two threads own the queue at the same time (Theorem 2).

7 OwnerQueue implementation

The OwnerQueue is implemented as a composition of two atomic objects: a lock-free queue and a shared counter.

The OwnerQueue's enqueue and dequeue methods are implemented as a simple non-synchronized two-step operation: perform the operation on the underlying queue, then atomically `fetch_and_increment` or `decrement_and_fetch` the counter, assigning the return value to c, then return whether $c == 0$.

For example, consider an empty queue with two threads t_1 and t_2 concurrently attempting to enqueue. Each performs the enqueue operation on the underlying queue, then attempts to increment the counter. Both increments will succeed, but only one gets the value 0 and thus returns *true*.

This implementation does not quite have the semantics of a proper queue, which is why the specification simply requires the implementation to keep track of a set of items in the structure. t_1's underlying-enq operation might succeed before that of t_2, but the subsequent `fetch_and_increment` operations were executed in the reverse order. t_2 would be somewhat surprised to find that although the queue appeared "empty" according to the return value of `enq_and_was_empty`, the first element subsequently dequeued was not the item that t_2 enqueued (it was instead that of t_1).

Thus, the contract's mention of "exclusive privilege to call `deq_and_is_empty`" also comes with a responsibility: the requirement that each thread that becomes the owner eventually exhausts the queue by dequeuing from it until `deq_and_is_empty` returns false. A thread must be prepared to deal with any number of other threads' items on the queue, in arbitrary order, if it becomes the owner.

8 Correctness

The correctness condition of linearizability[3] says that to be linearizable, any concurrent history of the object must be equivalent to a legal sequential history.

The correctness and linearizability of the underlying lock-free queue is assumed (e.g., as in [5]). Linearizability of shared counters using `fetch_and_add` and similar operations is also assumed. The OwnerQueue described here is probably linearizable, but not proved here.

A *history* is a sequence of events observed by clients of an object, as described by Herlihy et al. In these histories, the OwnerQueue is the object in question. A method call is indicated by the syntax *oq Method Thread* and a method return is indicated by the syntax *oq MethodOk Thread.* The first field ("oq") is the object, the second is the name of the method, and the third is the thread ID ("A" or "B") performing the invocation. Any text later on the line is a comment on the particular line.

OwnerQueue contract. *For all histories, for any thread* A *and OwnerQueue* oq*, while A is the owner (as in the following pattern in the history), no thread* B *calls* Deq *on* oq.

oq Enq(_) A	
oq EnqOk(true) A	A becomes the owner.
…	other entries, not including Deq() B
oq Deq() A	
oq DeqOk(_, false) A	A is no longer the owner.

In the above example, A became the owner when `enq_and_was_empty` returned *true*. During the period of time when A was the owner, no other thread called `deq_and_is_empty`.

Theorem 2. *If the OwnerQueue contract is satisfied, there are no overlapping calls to* `deq_and_is_empty`.

Proof. First I prove that no two threads consider themselves the owner at once.

Ownership transfer is associated with counter movement (from zero to one and one to zero). Proof by induction on steps – in the initial state, the queue is empty and quiescent, and no thread is the owner.

On each subsequent step, a thread may either increment the counter, decrement the counter, or take some unrelated action. If it increments the counter from zero, that one thread considers itself the owner and the counter is now greater than or equal to one. Other threads who increment the counter will not observe the value zero pre-increment. By the OwnerQueue contract, only the owner may dequeue and thereby decrement the counter. If the counter is decremented from one, that thread is no longer the owner, and any other thread would observe the base case. □

Note that a thread which starts but doesn't finish an enq operation could result in an OQ with one more item than is represented by the state of the counter. If this is the case, then when other threads perform enqueues and dequeues, they will handoff ownership with respect to the counter – i.e., the queue will always have one more item in it than the other threads realize.

It is also worthwhile to consider the property of *totality*. An operation is total if it is defined for every object value, otherwise it is *partial*.

Here I discuss totality of the OwnerQueue with respect to its contract. Normally in queues, enqueue operations are total because you can always add an element to the queue. Dequeue operations, however, are partial, because the queue might be empty when you tried to dequeue, and in this case the behavior is not defined. In the OwnerQueue contract, however, we prove that the dequeue operation is always defined as long as the caller respects the contract.

Theorem 3. *If the OwnerQueue contract is satisfied, both* `enq_and_was_empty` *and* `deq_and_is_empty` *are total.*

Proof. The underlying queue is unbounded, and the shared counter is effectively infinite[1], so `enq_and_was_empty` is total.

A thread can only become the owner after enqueuing. If the OwnerQueue contract holds, no thread will ever dequeue unless it is the owner.

[1]We are considering unbounded queues only. In practice, there exist both a memory limit for items on the queue and a limit on the size of integers for the counter.

A thread which starts but doesn't finish a deq operation could result in an OQ with one fewer item than is represented by the counter. But since only one thread can be calling dequeue at once, this disagreement can never result in a failure to dequeue. Therefore, `deq_and_is_empty` is total. □

The OwnerQueue is used in a fairly simple manner: a unit's runqueue is an OwnerQueue. If there are no items on the runqueue, there may or may not be an owner. But as soon as a thread enqueues onto an empty runqueue, that thread becomes the owner. That thread is constrained to ensure that messages on that queue will be executed as long as there are messages ready. In order for another thread to execute ready messages on that queue, it must obtain the first thread's consent to receive ownership of the queue (which, in our system, is obtained by taking it off the victim's runqueue atomically).

9 Efficiency

The system was tested on two benchmark programs, called `fib` and `ddb`.

`fib` is an implementation of tree-recursive Fibonacci: calculation of a particular value k in the Fibonacci sequence by creating two recursive free `fib` operations to calculate $k-1$ and $k-2$, and another free operation (`plus`) that waits until the futures from the previous operations are filled with values, then sums the values. This is inefficient for calculating the Fibonacci sequence (a dynamic programming algorithm reduces the exponential complexity of this algorithm to a linear one). However, the experiment is a good test that messages and futures are handled properly.

A modification of the experiment applies some arbitrary amount of extra work (for example, counting to ten thousand) at each `fib` and `plus` operation, in order to have a better idea of synchronization overhead (the extra work requires no synchronization).

`ddb` is a simple distributed database. Four database servers are implemented as units, and there are twenty-eight clients. The database servers map integer keys to values, and each server holds a unique set of keys; the server on which a key resides is determined by the hash of the key. The clients put heavy load on the servers, requesting and updating keys, and taking action based on the value of particular keys. Each server unit exposes three operations: put, get, and increment. This experiment is designed to test the unit-related aspects of the system: performance of units, unit exclusion, and workstealing.

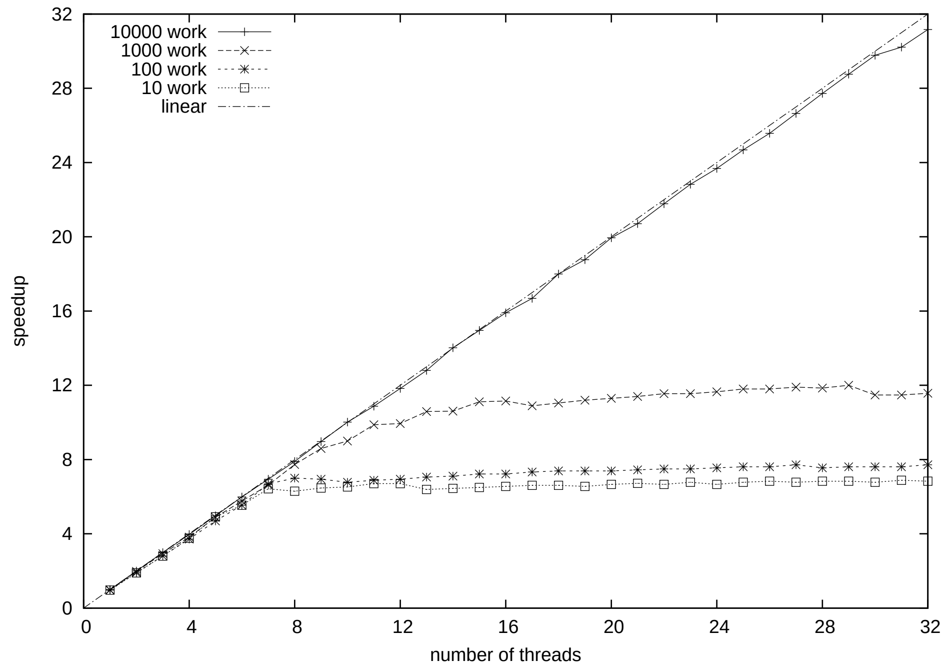

Figure 4: Speedup for the `fib` experiment: real time for the experiment divided by by the real time for the one-processor case.

The load on the database is unbalanced; in particular, one key is designed to have heavy load. Because of the characteristics of the database, this key is always assigned to one server, and because of unit exclusion, that server will be executed by at most one processor. In fact, if that server has enough requests to keep it constantly busy, then that server will only execute requests on that one unit. Therefore, units which process many messages should tend to have affinity to whatever processor they end up on. Since there are 32 units and 32 processors, the system should "settle" in the 32-processor case after a small amount of workstealing on having one unit per processor.

9.1 Results

The Fibonacci experiment demonstrated that the system scales fairly well to multiprocessor machines. The experiment is almost embarrassingly parallel, so it should exhibit linear speedup as processors are added, except for synchronization overhead. This appears to be the case, as Fig. 4 demonstrates;

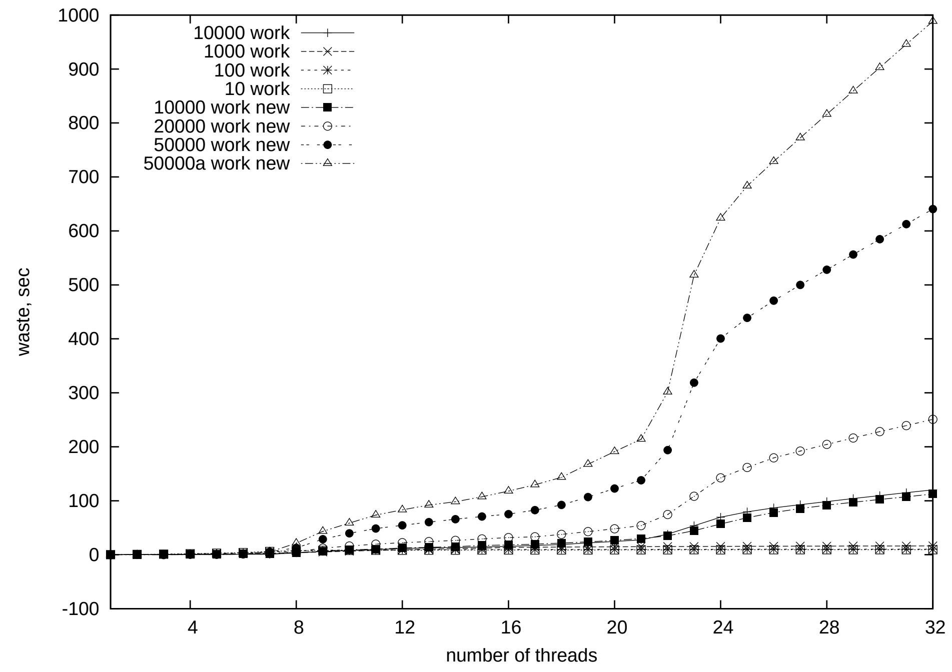

Figure 5: CPU time lost during `fib` due to synchronization. The lost time is calculated by subtracting the total CPU time from the CPU time of the one-thread execution.

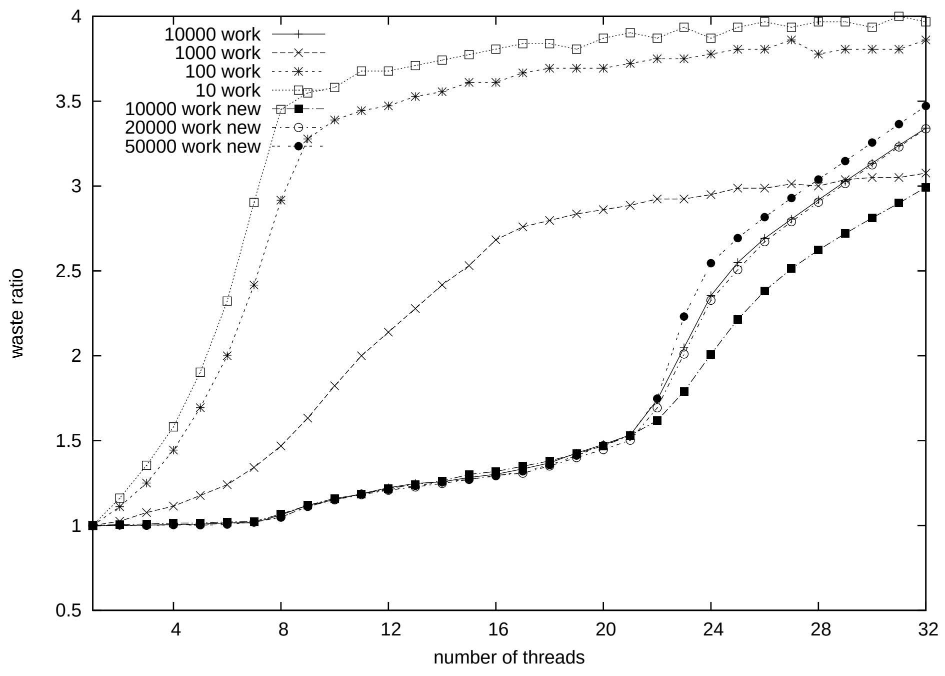

Figure 6: CPU waste ratio of `fib`: total CPU time divided by CPU time of one-thread execution.

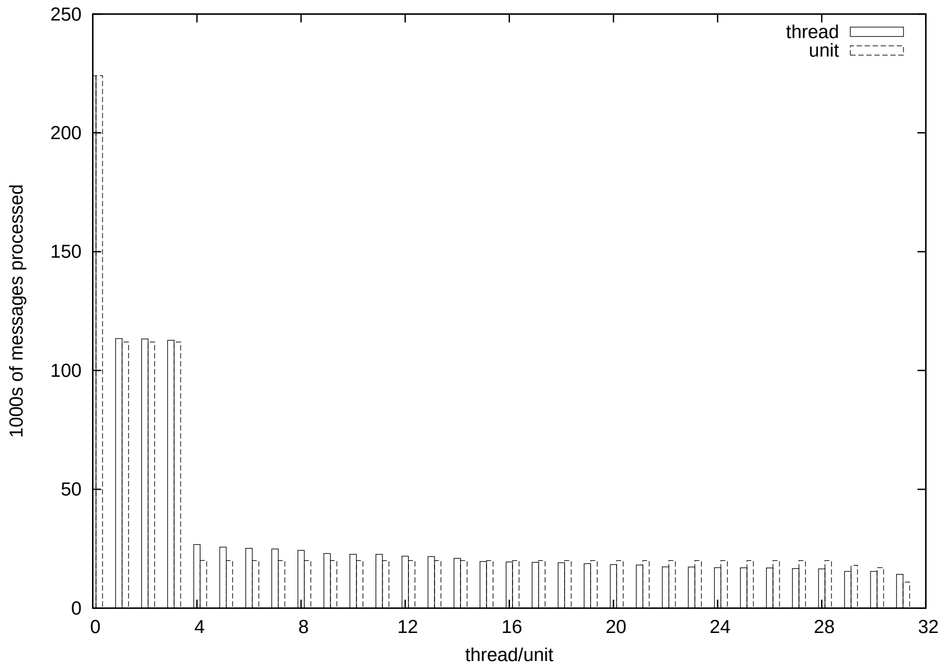

Figure 7: Number of messages processed in `ddb` by each processor.

when the amount of work to be done at each node is large, the system utilizes all the processors fairly effectively. Some synchronization overhead is observed, as the graph tails off at higher concurrencies. Another measurement of synchronization overhead is the idea of waste: the amount of extra CPU time that adding a processor costs. Figs. 5 and 6 demonstrate the amount of waste exhibited by the system. (I don't yet know what to make of these results, as it seems waste is a function of work, yet I don't believe it should be.)

The distributed database experiment demonstrated that the system is able to distribute work to processors, and that the workstealing algorithm we chose has the effect of placing each unit onto separate processors. Fig. 7 demonstrates that the workstealing was able to do this, since the distribution of work assigned to processors matches up very closely with the amount of work assigned to units by the experiment. Fig. 8 shows how much workstealing was taking place across the experiment; Fig. 9 shows which units were stolen during the 32-processor experiment. The results are as expected: the units with higher load were stolen less often, because the processors they

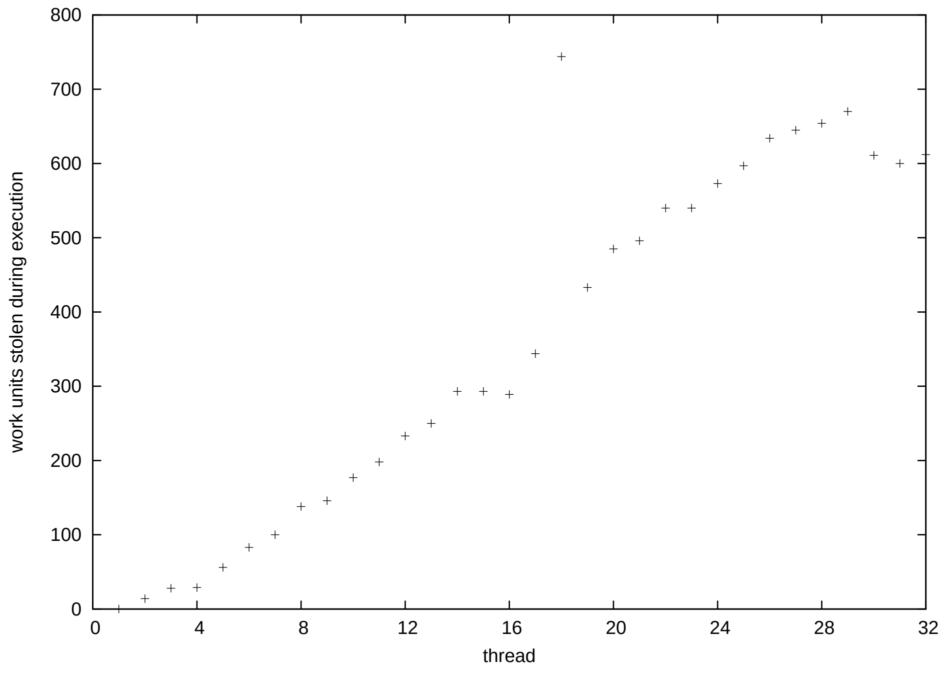

Figure 8: Number of units stolen during `ddb` experiment.

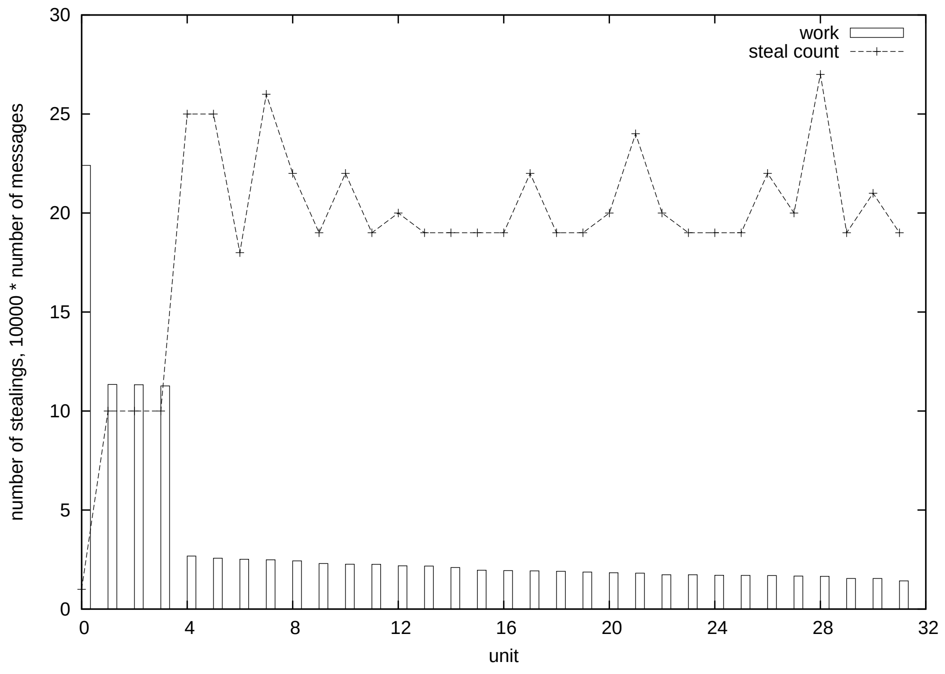

Figure 9: Units which were stolen during `ddb` experiment on 32 processors.

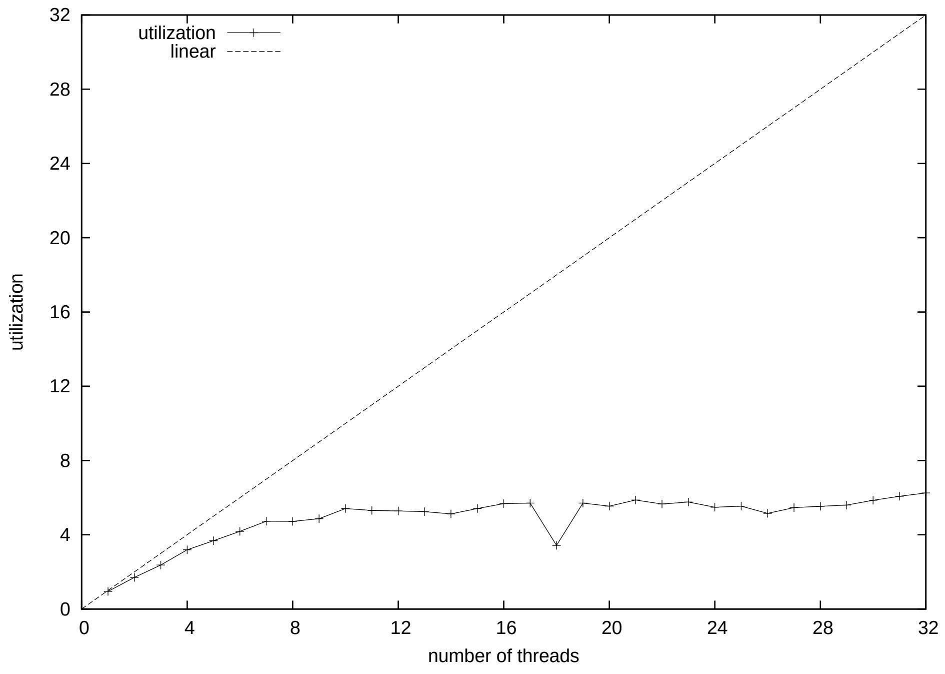

Figure 10: CPU Utilization for the `ddb` experiment.

were assigned to were busy working on them, and workstealing never steals a unit that another processor already owns.

CPU utilization in `ddb` was lower because of unit exclusion (see Fig. 10). Since most of the messages were concentrated onto a few units, it was often the case that some processors ran out of work.

10 Conclusions

A lock-free multithreaded scheduler that implements unit exclusion is feasible to implement. Performance in my implementation is not great, but is acceptable, and there are some notable ways to improve it (for example, using thread runqueues for free ops rather than scheduling them all off a main runqueue).

The OwnerQueue data structure seems very useful and more generally applicable as a design pattern for lock-free implementations. The OwnerQueue was found useful for another part of the scheduling system, not just unit exclusion: When a future is filled, it is important to wake up a list of messages that depend on that future. However, messages must not be woken up before the future is filled, and "lost wakeups" are not permissible, but messages could be added to the list asynchronously, even while the future is being filled.

Therefore, the list was implemented using an OwnerQueue. A thread which fills the future also takes ownership of the queue and wakes up all dependent messages. A thread which adds a message to the queue takes ownership only if it adds a message to an empty queue. Otherwise, it is notified by the real owner of the queue.

Thus, when it is desirable for a particular task to be handled by exactly one thread at a time, but you don't want to designate a separate thread to handle it, an "ownership" model is a useful abstraction. An OwnerQueue is a simple implementation of an ownership-based queue of tasks.

References

[1] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[2] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[3] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[4] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[5] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, 1994.