Parallel New World

Jonathan Worthington

French Perl Workshop 2007

Parallel New World

3.0×10^8 m/s

3.0×10^8 m/s

The fastest we can move information

Parallel New World

3.0×10^9 hz

Parallel New World

3.0×10^9 hz

The number of cycles a 3GHz CPU does per second

Parallel New World

10 cm

Parallel New World

10 cm

The distance we can move data per cycle.*

10 cm

The distance we can move data per cycle.*

* If the CPU were a vacuum.

Parallel New World

10 cm

The distance we can move data per cycle.*

* If the CPU were a vacuum.
(By the way, it's not. ☺)

Current leakage
Capacitance effects
Quantum effects

We can't just keep clocking up the cycles.

The Answer:
Go parallel.

Parallel New World

Task Parallelism

Task Parallelism

- Take a range of different tasks
- Run them in parallel
- Consider a game…
  - Rendering beautiful graphics
  - Doing AI for intelligent opponents
- Can do these two tasks in parallel
- Tasks will often need to communicate somehow

Parallel New World

Data Parallelism

Data Parallelism

- Performing the same task on many items
- For example, calling the same method on every element of an array
- If we know it is safe, we can call the method over the array in parallel
    - Different execution units work on different elements of the array

Parallel New World

Parallelism Primitives

Threads

- A thread has its own program counter and stack
- Lives in the same memory space as other threads in the process => communication by shared memory
- Usually, we create it and point it at a bit of code to start executing

Mutexes

- Short for Mutual Exclusion
- Only one thread can hold the lock at a time
- If thread 1 has the mutex and thread 2 wants it, thread 2 must wait until thread 1 is finished with it
- Easy to implement directly on top of hardware primitives

Semaphores

- An integer value
- Initialize it to the number of resources available
- P operation acquires a resource:

```
sub P($s) { wait until $s > 0 then $s-- }
```

- V releases it:

```
sub V($s) { $s++ }
```

- Blame the Dutch for the naming. ☺

Parallel New World

Ways To Screw It Up

The Ten Thousand Threads Myth

- "If I start ten thousand threads my application will run about ten thousand times as fast!!!"

The Ten Thousand Threads Myth

- "If I start ten thousand threads my application will run about ten thousand times as fast!!!"
- Well, sure, if...
  - You've got 10,000 CPUs/cores...
  - And no scheduling overhead...
  - And virtually no inter-thread communication.

How many threads?

- If your application is CPU-bound there is no point having more threads than you have CPUs/cores.
- Having more…
  - Means the scheduler has more work so you get to run less code
  - Introduces more contention on locks with no gains

Lack of concurrency control

- If two threads are accessing the same bit of data, you need to control access to it

Thread 1	Thread 2
`$acc_a.debit(100);` `$acc_b.credit(100);`	`$total += $acc_a.bal` `$total += $acc_b.bal`

Lack of concurrency control

- If two threads are accessing the same bit of data, you need to control access to it

Thread 1	Thread 2
`$acc_a.debit(100);`	`$total += $acc_a.bal`
`$acc_b.credit(100);`	`$total += $acc_b.bal`

Lack of concurrency control

- If two threads are accessing the same bit of data, you need to control access to it

Thread 1	Thread 2
`$acc_a.debit(100);`	`$total += $acc_a.bal`
`$acc_b.credit(100);`	`$total += $acc_b.bal`

Lack of concurrency control

- If two threads are accessing the same bit of data, you need to control access to it

Thread 1	Thread 2
`$acc_a.debit(100);`	`$total += $acc_a.bal`
`$acc_b.credit(100);`	`$total += $acc_b.bal`

Lack of concurrency control

- If two threads are accessing the same bit of data, you need to control access to it

Thread 1	Thread 2
`$acc_a.debit(100);`	`$total += $acc_a.bal`
`$acc_b.credit(100);`	`$total += $acc_b.bal`

Lack of concurrency control

- If two threads are accessing the same bit of data, you need to control access to it

Thread 1	Thread 2
`$acc_a.debit(100);`	`$total += $acc_a.bal`
`$acc_b.credit(100);`	`$total += $acc_b.bal`

- Our program to get the balance across the accounts gets the wrong answer, because it sees an inconsistent state

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

`lock $acc_a {`
`    lock $acc_b {`
`        ...`
`    }`
`}`
	`lock $acc_b {`
	`    lock $acc_a {`
	`        ...`
	`    }`
	`}`

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

`lock $acc_a {`
`    lock $acc_b {`
`        ...`
`    }`
`}`
	`lock $acc_b {`
	`    lock $acc_a {`
	`        ...`
	`    }`
	`}`

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

`lock $acc_a {`
`    lock $acc_b {`
`        ...`
`    }`
`}`
	`lock $acc_b {`
	`    lock $acc_a {`
	`        ...`
	`    }`
	`}`

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

`lock $acc_a {`
`    lock $acc_b {`
`        ...`
`    }`
`}`
	`lock $acc_b {`
	`    lock $acc_a {`
	`        ...`
	`    }`
	`}`

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

`lock $acc_a {`
`    lock $acc_b {`
`        ...`
`    }`
`}`
	`lock $acc_b {`
	`    lock $acc_a {`
	`        ...`
	`    }`
	`}`

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

`lock $acc_a {`
`    lock $acc_b {`
`        ...`
`    }`
`}`
	`lock $acc_b {`
	`    lock $acc_a {`
	`        ...`
	`    }`
	`}`

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

<pre>lock $acc_a {
    lock $acc_b {
        ...
    }
}</pre>
	<pre>lock $acc_b {
	    lock $acc_a {
	        ...
	    }
	}</pre>

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

`lock $acc_a {`
`    lock $acc_b {`
`        ...`
`    }`
`}`
	`lock $acc_b {`
	`    lock $acc_a {`
	`        ...`
	`    }`
	`}`

Deadlock

- Deadlock occurs when you get circular waiting on a lock

Thread 1	Thread 2

`lock $acc_a {`
`    lock $acc_b {`
`        ...`
`    }`
`}`
	`lock $acc_b {`
	`    lock $acc_a {`
	`        ...`
	`    }`
	`}`

- Thread 1 requires lock A, Thread 2 requires lock B. Neither can proceed.

Lock Granularity Issues

- You have a hash table data structure
- You can have a lock over the whole data structure or a lock on every bucket in the hash (the place where the data is stored)
- Which to do depends on the kind of operations that will be performed most often on the hash

Lock Granularity Issues

- If you have a single lock for the entire hash...
  - Take one lock for any operation(s)
  - Only one access to the hash can proceed at a time
  - If you're mostly reading from the hash or updating existing values, you lose a lot of potential concurrency

Lock Granularity Issues

- If you have a lock for every element...
  - More concurrency on reads/updates
  - A write may need to resize the hash table => requires locking of every bucket (a LOT of locking overhead)!
  - Compound operations or iterating over the hash needs many locks
  - Can waste a lot of time juggling locks

Parallel New World

Cache Issues

Why CPUs have caches

- A CPU cache stores copies of sections of main memory
- Accessing main memory is slow (on the order of magnitude of 100s of CPU cycles)
  - You have to go off-chip
  - DRAM read speeds haven't kept up with CPU speed gains

What A Cache Looks Like

- Contains a number of Cache Lines, perhaps of length 128 bits (holds 4 32-bit words), but it varies by CPU
- A cache line holds a copy of an area of main memory
- When the cache is full, have to choose a line to evict; random but not last used is a common policy

Reads And Writes With A Cache

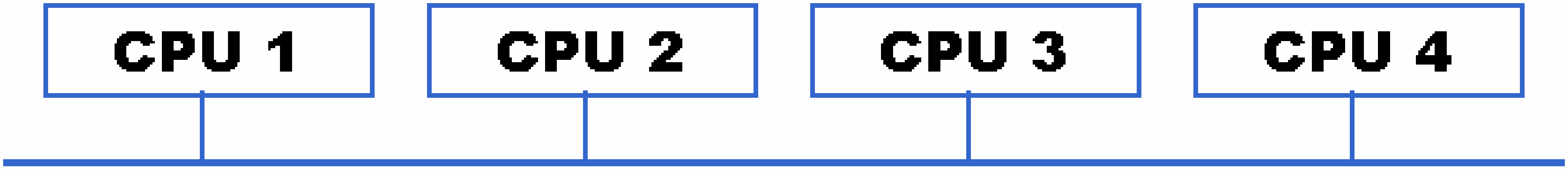
- Reads are done through the cache; if the data isn't in the cache, fetched from main memory
- Write-through: when making a change, you bypass the cache and write directly to memory
- Write-back: you make changes to the line in the cache, and when that line is evicted the data is written to memory

Cache Line States For One CPU

- With a single CPU, a cache line can be in one of three states:
  - Invalid, when it's not holding a copy of any memory location
  - Valid, when it holds a copy of a memory location
  - Dirty, when the data it holds has been changed (update memory on eviction)

SMP Architecture

- The CPUs or cores are connected by a shared bus



- They can all see each others memory requests
- Note: SMP doesn't scale, as the more CPUs you get the more contention you get on the bus

Caches And Multiple CPUs

- Reading data through the cache is fine
  - Every CPU can have a copy of the data in its cache
  - Multiple threads reading shared memory is fine
- Note that all CPUs can see what other CPUs are reading

Caches And Multiple CPUs

- Writes are more complex...
  - We need other caches to toss their copies of the data, as it is about to become out of date
  - We start by doing an exclusive read
  - This gives us the data
  - All other CPUs see it and their caches evict those cache lines

Caches And Multiple CPUs

- The Modified state
  - When we modify data held in a cache line, its state changes to Modified
- If we see somebody else try to read a memory address and we have it cached and marked modified, we:
  - Jam their read
  - Write the modification to memory

Consequences for application design

- If multiple threads are reading from and writing to memory on the same cache line, the cache line is "pinging" between the CPUs
- Therefore…
  - Maximize thread local storage
  - Minimize writes to memory shared between threads

Consequences for locking

- A lock is just a bit of memory set to a value saying whether the lock is taken or not (or maybe saying who holds the lock, if anyone)
- Cache line for that memory location will ping between CPUs too
- Therefore, taking locks is expensive in terms of bus activity (and thus time)

False Sharing

- You have four threads numbered 1 to 4
- You have an array with four elements, all 32-bit words
- Thread 1 will read/write the first element, thread 2 will read/write the second element, etc.
- So we've got no sharing and we're nice and efficient, right?

False Sharing

- No. ☹
- Cache lines hold many words
- Our array may lie all on a single cache line or at best be spread over two cache lines
- At the program level, there is no sharing of memory. At the hardware level, there is, due to cache line size.

Summary of cache issues

- If you want to write concurrent programs that perform well, you must think about its cache characteristics
- Use as much thread-local storage as possible
- Minimize lock taking (but larger locks limit concurrency – difficult trade-off)
- Beware of false sharing

Parallel New World

Parallelism In Perl 6

The Big Picture

- Syntax and details are yet to be finalized, so anything I say may change
- Much more declarative than Perl 5
  - We say what we want and leave the computer to work out how to make it happen
- Support for both task based parallelism and data based parallelism

`async` blocks

- To say that a block of code should be executed asynchronously (e.g. in a thread of its own), prefix the block with `async`.

```
async {
    # Code in here runs in another thread
}
# Code here is still in original thread
```

async blocks

- It returns a thread object, which we can keep and use to control the thread if needed.

```
my $thread = async {
    # Code in here runs in another thread
}

# Numifies to the thread ID.
say "The thread has ID " ~ +$thread;

# Can do the usual thread operations...
$thread.join();
```

Software Transactional Memory

- A transaction is a sequence of operations that are guaranteed to:
  - Either complete in full or have no effect on the system at all
  - Take place atomically
- STM implements very lightweight and cache-sensitive transactions with these characteristics

How STM Works, Roughly

- We don't take any locks
- Whenever we make a change to some state (e.g. assign to a variable), we don't change the original, but instead note the change that took place
- Reads inside our transaction see the changes
- Reads outside of it don't see them

How STM Works, Roughly

- If we have run all of the instructions successfully, we try to commit the transaction
  - Check none of the values that we read or modified have changed
  - If none did, apply our changes
  - Otherwise, run our transaction from the start again, with updated values

How STM Works, Roughly

- If an error occurs inside of our transaction...
  - We didn't actually change anything, just noted our changes
  - So just discard the transaction information, and it's as if we never ran it

STM in Perl 6

- Write an `atomic` block to state that the code inside it is to run as a transaction.

```
atomic {
    $acc_a.debit(100);
    $acc_b.credit(100);
}
```

- Note that you can not do any I/O inside a transaction, as we can't roll that back; trying to do so is a fatal error.

Advantages Of STM

- Declarative: frees the programmer from worrying about locking issues
- A good implementation of STM will result in little locking taking place under the hood, making the application much more scalable
- Your program always makes progress: if a transaction fails at commit time, it's because another one succeeded

Data Parallelism

- Possible when we are performing the same operation over all elements of an aggregate, for example an array
- Perl 6 provides ways to perform a range of operations over all elements of an array:

```
# Element-wise addition of two arrays
my @c = @a >>+<< @b;
# Call a method on every element
@dogs>>.wag_tail();
```

Data Parallelism

- When you use the hyper, cross or reduction operators, it's not just cool syntax
- You are declaring that there are no data dependencies between operations on elements of the array
- From this, the compiler can infer that it is safe to parallelize the operations

Data Parallelism

- You will probably use a pragma to indicate that you want such operations to be parallelized.

```
use Parallel::Hypers; # I made this up :-)
```

- These pragmas will almost certainly be written as modules – probably in Perl
- Powerfully allows us to separate concerns – how to parallelize an operation from the operation itself

Parallel New World

Research And Future Ideas

Concurrency Annotations

- Another idea being worked on is annotating variables with what kind of concurrency control they require

```
my mutex $foo; # Mutex
my mrsw $bar;  # Multiple Reader Single
               # Writer lock
my $baz;       # No locking required
```

- Then the compiler is responsible for enforcing these locking regimes on accesses to the variables

Lock Free Data Structures

- We have seen that taking locks limits performance and scalability
- What if we could build data structures that…
  - Allowed concurrent operation
  - Don't require taking of locks
  - Will still guarantee that we don't end up with an inconsistent state

Atomic Compare And Swap

- Many CPUs provide a CAS (Atomic Compare And Swap) operation

```
seen = CAS(addr, expected, replacement)
```

- `addr` is a memory address
- `expected` is what it should contain
- `replacement` is what we store there if it contains the expected value
- `seen` is the value that was there

Using CAS

- CAS is the primitive concurrency operation upon which we can build all others (the theorists proved it)
- For example, a bad implementation of a mutex using CAS is:

```
sub take {
    while (CAS($mutex, 0, 1) != 0) { }
}
sub release {
    CAS($mutex, 1, 0) || die "WTF?!";
}
```

Making Any Data Structure Lock Free

- If you add a level of indirection, you end up with a memory location containing a pointer (reference) to the data structure
- You make a copy of the data structure at that pointer
- You modify the copy
- You then try and use CAS to swap the original pointer with that of your copy

Making Any Data Structure Lock Free

- This has the (good) property that if your change fails to be made, it's because another thread made one => program made progress ☺
- For certain data structures, you can do MUCH more efficient things built on top of CAS.
- It's really hard to get right => need smart people to write libraries of these

The End

Questions?

Merci!