

Efficient Multi-core Application Architectures

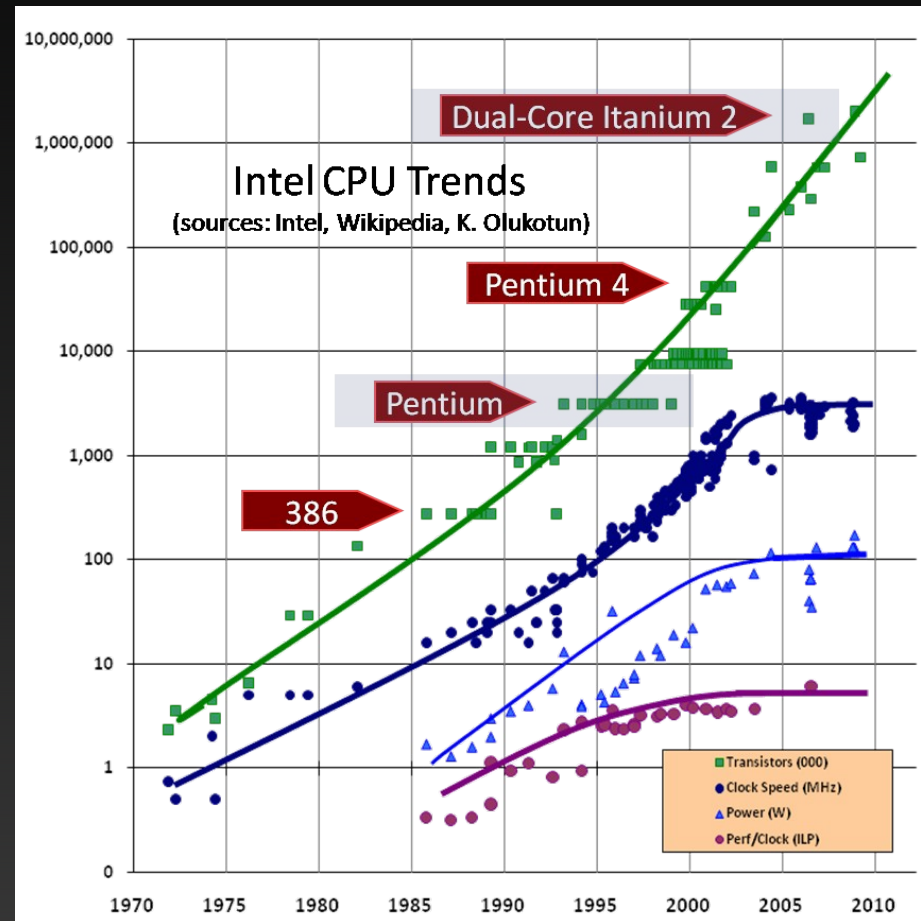
Open Source Bridge 2010

Eric Day

<http://oddmments.org/>

Senior Software Architect @ Rackspace

Moore failed us in 2003 (sort of)



<http://www.gotw.ca/publications/concurrency-ddj.htm>

The world is now multi-core



<http://www.freakingnews.com/Two-Headed-Llama-Pictures-39232.asp>

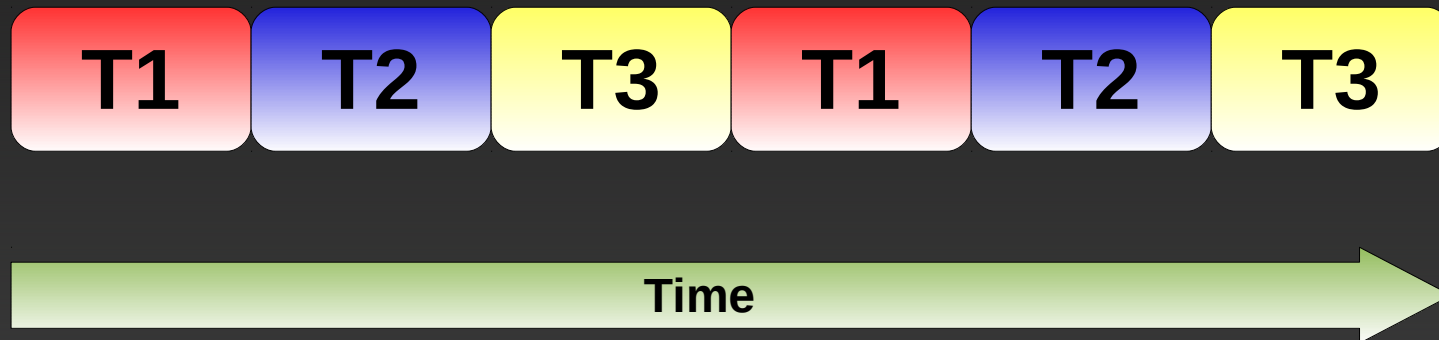
Most applications are not

Very few handle it efficiently

Concurrency \neq Parallelism

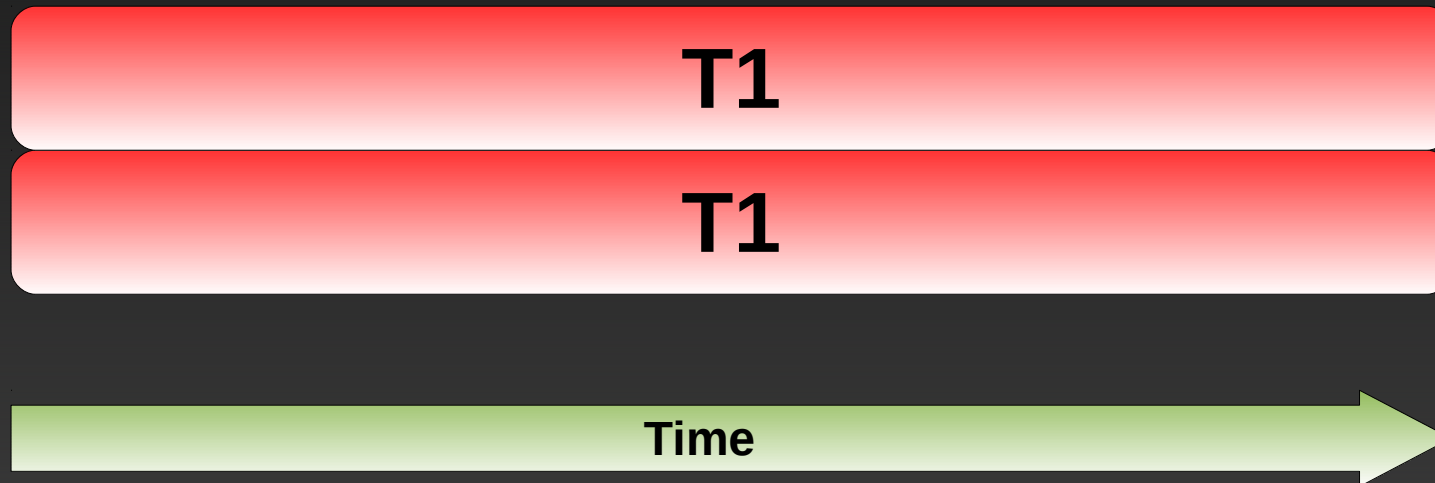
Concurrency

- Multiple tasks
- Execution of tasks interleaved
- Doesn't need to be parallel



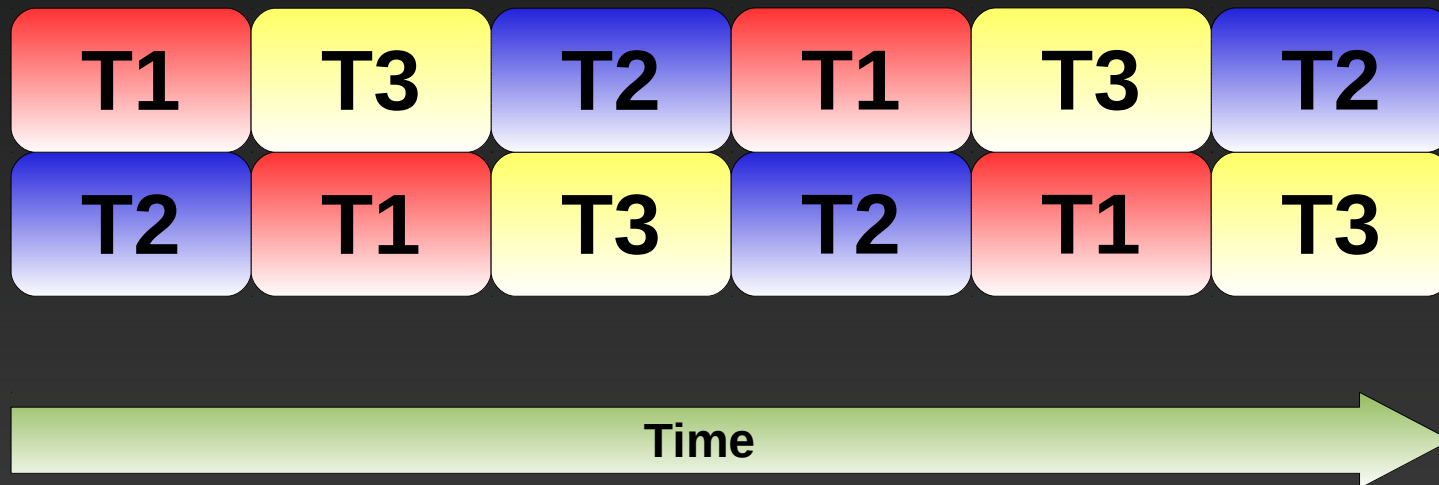
Parallelism

- Multi-CPU or Multi-core
- Operations happening in parallel
- Doesn't need to be concurrent



Concurrency + Parallelism

- 2+ tasks executing at same time
- Get more done in same time (usually)



Many solutions in modern operating systems

Processes

- Heavyweight
- Managed by kernel
- No shared resources except System V IPC
- Preemptive

Kernel threads

- Medium-weight
- Managed by kernel
- Shares process resources
- Preemptive

User threads

- Lightweight
- Managed by user-space libraries
- Shares process resources
- Preemptive and cooperative
- Some implementations can have starvation

Fibers

- Ultra lightweight
- Managed by user-space libraries
- Shares process resources
- Cooperative
- Not very common

Other

- Coroutines
- Go - “goroutines”
- Erlang – Lightweight processes
- Apple OSX – Grand Central Dispatch

But, most language
implementations hate you

Some have no
concept of threads

Some have threads
without parallelism
(useless on multi-core)

Use case: web server

C10K? Hah!

C50K

One process per client

- Apache 1.x
- Waste of memory (especially without CoW)
- Harder to share resources like caches
- Kernel slows with high process counts

One thread per client

- Apache 2.x (sometimes)
- Smaller memory footprint, just a stack
- Share resources with other clients
- Kernel slows with high thread counts
- User threads can be better

One thread for all clients

- Lighttpd, Python Twisted, node.js
- Concurrency through I/O events
- Non-blocking system calls
- Very small memory footprint
- Concurrent, but not parallel

Non-blocking I/O Events

- Calls like `read()` and `write()` block by default
- `fcntl(socket, F_SETFL, O_NONBLOCK);`
- Return `EAGAIN` instead of blocking
- Multiplex with `poll()`, `select()`, `epoll`, `kqueue`, `/dev/poll`, event ports, ...

So, which is the best?

Events + Threads

Why don't all servers do this?

- We are taught blocking I/O
- “It's too complicated”
- Most libraries don't support it
- No good frameworks to start with

I'm an efficiency freak

I'm a modularity freak

Scale Stack



<http://scalestack.org/>

Scale Stack

- Like Python Twisted and node.js, but threaded
- C++
- Open Source (BSD)
- Extremely modular
- Write your own C10k+ servers quickly

Echo Server

```
class stream: public network::stream
{
public:

    stream(network::stream_service& creator);

    ~stream();

    size_t read(uint8_t* buffer, size_t size);

    void flush_write(void);

private:

    size_t _echo(void);

    size_t _size;
    uint8_t* _buffer;
};
```

Echo Server

```
size_t stream::read(uint8_t* buffer, size_t size)
{
    _size = size;
    _buffer = buffer;
    return _echo();
}

void stream::flush_write(void)
{
    consume(_echo());
}

size_t stream::_echo(void)
{
    size_t written = write(_buffer, _size, true);
    _size -= written;
    _buffer += written;
    return written;
}
```

Echo Server

- Another “glue” class for TCP binding
- Module definition file
- Use existing event and network/tcp modules!

Echo Server

```
shell$ scalestack echo::server::tcp.ports=12345 \  
      event::service.threads=4 v  
NOTICE [event] Loaded  
NOTICE [event::libevent] Loaded  
NOTICE [network] Loaded  
NOTICE [echo::server] Loaded  
NOTICE [network::ip] Loaded  
NOTICE [network::ip::tcp] Loaded  
NOTICE [echo::server::tcp] Loaded  
NOTICE [echo::server::tcp] Listening on 0.0.0.0:12345  
NOTICE [kernel] Core now running  
NOTICE [echo::server::tcp] Accepted 127.0.0.1:50840  
^CNOTICE [kernel] Core received shutdown request  
NOTICE [echo::server::tcp] Unloaded  
NOTICE [echo::server] Unloaded  
NOTICE [network::ip::tcp] Unloaded  
NOTICE [network::ip] Unloaded  
NOTICE [network] Unloaded  
NOTICE [event::libevent] Unloaded  
NOTICE [event] Unloaded
```

Echo Server

- C10k+ server in minutes
- Fairly low level for overriding functionality
- Minimal socket buffer copies
- For details, see:
 - [scalestack/echo/server/*](#)
 - [scalestack/echo/server/tcp/*](#)
- UDP and Unix sockets also supported

Yay, my sockets don't block!
What about files on disk?

Disk I/O

- Will block, unlike sockets
- Create pool of threads to handle disk I/O
- Asynchronous queue for disk I/O threads
- Can use this technique for other resources too

Great! But what about
task coordination?

Synchronizing Shared Resources

- Semaphores
- Mutexes
- Atomic operations
- Spinning/busy-wait

Techniques

- Coarse vs fine grained control
 - Watch out for starvation, deadlock, ...
- Asynchronous queues between threads
 - Minimal locking
- Lock-less data structures with atomic ops

Get involved!

- Me: <http://oddmments.org/>
- Project: <http://scalestack.org/>
 - Mailing list
 - Links to Launchpad
 - #scalestack on irc.freenode.net