Welcome!
Today's Agenda:

- Introduction
- Hardware
- Trust No One / An Efficient Pattern
- Experiments
- Final Assignment
A Brief History of Many Cores

Once upon a time...

Then, in 2005: Intel's Core 2 Duo (April 22).
(Also 2005: AMD Athlon 64 X2. April 21.)

2007: Intel Core 2 Quad

2010: AMD Phenom II X6
A Brief History of Many Cores

Once upon a time...

Then, in 2005: Intel's Core 2 Duo (April 22).
(Also 2005: AMD Athlon 64 X2. April 21.)

2007: Intel Core 2 Quad

2010: AMD Phenom II X6

Today...
Introduction

A Brief History of Many Cores

Once upon a time...

Then, in 2005: Intel's Core 2 Duo (April 22).
(Also 2005: AMD Athlon 64 X2. April 21.)

2007: Intel Core 2 Quad

2010: AMD Phenom II X6

2017: Threadripper 1920X

2018: Threadripper 2950X
Introduction

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Introduction

Threads / Scalability

Performance scaling in HWBOT Prime (Java)

Timed Apache Compilation v2.2.17

OpenSSL AES Decryption
Introduction

Optimizing for Multiple Cores

What we did before:

1. Profile.
2. Understand the hardware.
3. Trust No One.

Goal:

- It’s fast enough when it scales linearly with the number of cores.
- It’s fast enough when the parallelizable code scales linearly with the number of cores.
- It’s fast enough if there is no sequential code.
Today's Agenda:

- Introduction
- Hardware
- Trust No One / An Efficient Pattern
- Experiments
- Final Assignment
Hardware Review

We have:

- Four physical cores
- Each running two threads
- L1 cache: 32Kb, 4 cycles latency
- L2 cache: 256Kb, 10 cycles latency
- A large shared L3 cache.
Simultaneous Multi-Threading (SMT)

(Also known as hyperthreading)

Pipelines grow wider and deeper:

- Wider, to execute multiple instructions in parallel in a single cycle.
- Deeper, to reduce the complexity of each pipeline stage, which allows for a higher frequency.

However, parallel instructions must be independent, otherwise we get bubbles.

Observation: two independent threads provide twice as many independent instructions.
Simultaneous Multi-Threading (SMT)

```
fldz
xor ecx, ecx
fld dword ptr [4520h]
mov edx, 28929227h
fld dword ptr [452Ch]
push esi
mov esi, 0C350h
add ecx, edx
mov eax, 91D2A969h
xor edx, 17737352h
shr ecx, 1
mul eax, edx
fld st(1)
faddp st(3), st
mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1), st
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Eh
dec esi
jne tobetimed+1Fh
```
Simultaneous Multi-Threading (SMT)

Nehalem (i7): six wide.

- Three memory operations
- Three calculations (float, int, vector)

```plaintext
fldz
xor ecx, ecx
fld dword ptr [4520h]
mov edx, 28929227h
fld dword ptr [452Ch]
push esi
mov esi, 0C350h
add ecx, edx
mov eax, [91D2h]
xor edx, 17737352h
shr ecx, 1
mul eax, edx
fld st(1)
faddp st(3), st
mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1), st
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Eh
dec esi
jne tobetimed+1Fh
```
INFOMOV – Lecture 12 – “Multithreading”

Hardware

Simultaneous Multi-Threading (SMT)

Nehalem (i7): six wide*.

- Three memory operations
- Three calculations (float, int, vector)

SMT: feeding the pipe from two threads.

All it really takes is an extra set of registers.

Simultaneous Multi-Threading (SMT)

Hyperthreading does mean that now *two* threads are using the same L1 and L2 cache.

- For the average case, this will reduce data locality.
- If both threads use the same data, data locality remains the same.
- One thread can also be used to fetch data that the other thread will need *

Hardware

Multiple Processors: NUMA

Two physical processors on a single mainboard:

- Each CPU has its own memory
- Each CPU can access the memory of the other CPU.

The penalty for accessing ‘foreign’ memory is $\sim$50%.
INFOMOV – Lecture 12 – “Multithreading”

Hardware

Multiple Processors: NUMA

Do we care?

- Most boards host 1 CPU.
- A quadcore still talks to memory via a single interface.

However:

Threadripper is a NUMA device.

Threadripper = 2x Zeppelin, with for each Zeppelin:

- L1, L2, L3 cache
- A link to memory

This CPU behaves as two CPUs in a single socket.
Hardware

Multiple Processors: NUMA

Threadripper & Windows:

- Threadripper hides NUMA from the OS
- Most software is not NUMA-aware.
Today's Agenda:

- Introduction
- Hardware
- Trust No One / An Efficient Pattern
- Experiments
- Final Assignment
Windows

DWORD WINAPI myThread(LPVOID lpParameter)
{
    unsigned int& myCounter = *(*(unsigned int*)lpParameter);
    while(myCounter < 0xFFFFFFFF) ++myCounter;
    return 0;
}

int main(int argc, char* argv[])
{
    using namespace std;
    unsigned int myCounter = 0;
    DWORD myThreadId;
    HANDLE myHandle = CreateThread(0, 0, myThread, &myCounter, 0, &myThreadId);
    char myChar = ' ';
    while(myChar != 'q') {
        cout << myCounter << endl;
        myChar = getchar();
    }
    CloseHandle(myHandle);
    return 0;
}
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
    boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
    for (int i = 0; i < 5; ++i)
    {
        wait(1);
        std::cout << i << '
';
    }
}

int main()
{
    boost::thread t{thread};
t.join();
}
#pragma omp parallel for
for (int n = 0; n < 10; ++n) printf("%d", n);
printf(".\n");

float a[8], b[8];
#pragma omp simd
for (int n = 0; n < 8; ++n) a[n] += b[n];

struct node { node *left, *right; };
extern void process(node*);

void postorder_traverse(node* p)
{
    if (p->left)
        #pragma omp task
        postorder_traverse(p->left);

    if (p->right)
        #pragma omp task
        postorder_traverse(p->right);

    #pragma omp taskwait
    process(p);
}
#include "tbb/task_group.h"

using namespace tbb;

int Fib( int n )
{
    if (n<2)
    {
        return n;
    }
    else
    {
        int x, y;
        task_group g;
        g.run([&]{x=Fib(n-1);}); // spawn a task
        g.run([&]{y=Fib(n-2);}); // spawn another task
        g.wait(); // wait for both tasks to complete
        return x + y;
    }
}
Considerations

When using external tools to manage your threads, ask yourself:

- What is the overhead of creating / destroying a thread?
- Do I even know when threads are created?
- Do I know on which cores threads execute?

What if... we handled everything ourselves?
Worker threads never die
- Tasks are claimed by worker threads
- Execution of a task may depend on completion of other tasks
- Tasks can produce new tasks
Trust No One

worker thread 0
worker thread 1
worker thread 2
worker thread 3
worker thread 4
worker thread 5
worker thread 6
worker thread 7

Fibers:
- Light-weight threads, with a complete state: registers (incl. program counter), stack
- Available in Windows, PS4, ...
- Allows the task system to suspend a job, e.g. to wait for scheduled sub-tasks

Sub-tasks:
- Decrement a counter when done
- When counter reaches zero, linked task is resumed.

Naughty Dog’s “The Last of Us”:
- Tasks are executed as fibers
- A fiber stores a stack and a set of registers
- Tasks can be interrupted by storing the fiber in a waiting list
What does this mean for the jobs themselves?

- "Cooperative multithreading", no preemption
- Must be independent!
Today's Agenda:

- Introduction
- Hardware
- Trust No One / An Efficient Pattern
- Experiments
- Final Assignment
Experiments

1. False sharing
   - Setups:
     - 8 threads update a single counter
     - 8 threads update counters in a single cache line
     - 8 threads update counters in different cache lines

2. Locking to cores
   - Four rotating hedgehogs, core-locked and not core-locked

3. Calculating the Mandelbrot using worker threads

4. Hyperthreading
   - Setup:
     - 4 threads calculate the special Mandelbrot
     - 8 threads calculate the special Mandelbrot
     - Now with worker threads
     - Switch out Mandelbrot for blur, to test bandwidth-intensive app
Today's Agenda:

- Introduction
- Hardware
- Trust No One / An Efficient Pattern
- Experiments
- Final Assignment
END of “Multithreading”

next lecture: “Guest Lecture”