Chip MultiProcessors

Past, Present, and Future

Based in large parts on the technical report
"The Landscape of Parallel Computing Research: A View from Berkeley"
and references therein

http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

Introduction

- We are accustomed to Moores Law-like scaling in single threaded performance
- Huge heat sinks and massive power supplies common
- Exploding laptops

- The single threaded performance has recently hit a wall, and all eyes are focused towards parallelism
- We reason about why this shift is taking place and what we might expect to see in the near and distant future

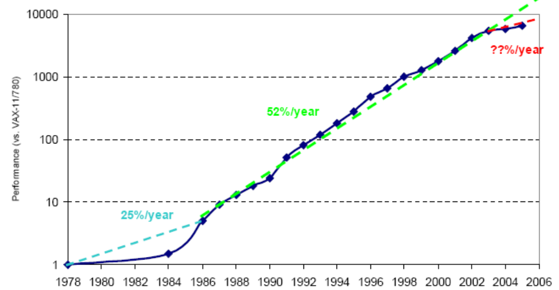
Old/New Common Wisdoms

- Old CW: Power is free, transistors are expensive.
- **New CW: The "Power Wall", power is expensive.**

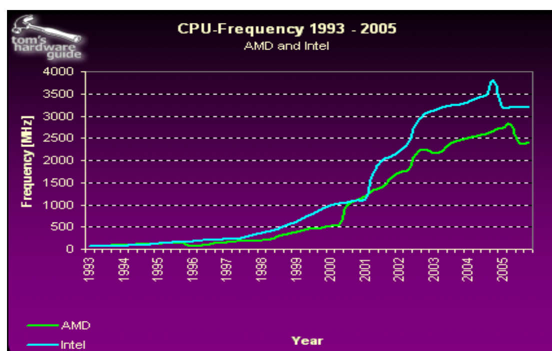
- Old CW: Multiply is slow, memory access is fast.
- **New CW: The "Memory Wall", memory access on modern processors can take 200 clocks but floating point operations take just a few.**

- Old CW: Uniprocessor performance doubles every 18 months.
- **New CW: Since 2002 there is a gap between "Moores law-performance" and actual performance, in 2006 lagging a factor of three.**

"Moores Gap"



Clock Frequency Limit Reached



Our Parallel Future

- Conclusion: Increasing clock frequency, instruction level parallelism, CPU tricks such as branch-prediction, etc not enough to keep up with Moores law-like performance increases.

- Remedy:
  - Go parallel
  - Put several processing cores on the same chip
  - We already see this:
    - AMD and Intel markets 2 and 4 core processors
    - Sun sells an 8 core UltraSPARC T1
    - NVIDIA GeForce 8800 GTX ships with 128 cores
    - Cisco has a router-chip with 188 cores
  - We've seen it before:
    - Execube (1993), early multi-core chip

Multicore/Manycore

- A few large cores or many small cores?
- **Multi**-core
  - Common cores squeezed together
  - SMP-like
  - Will probably scale poorly
- **Many**-core
  - Simpler (smaller) cores
  - Can fit many more cores on the same area
  - Massive parallelism
  - Custom core interconnection likely
  - More distributed memory-like

1000s of cores?

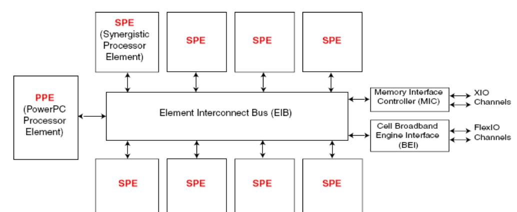
- Cisco ships a 188-core network processor with a 130nm process
- Scaling down to 30nm and we have 1504 cores
- Although we are not there yet, there is strong empirical evidence that we might arrive there soon
- Big complex cores will not reach this high anytime soon
  - Processor core area increase results in modest performance improvements while consuming large amounts of electricity
    - Increase in performance mainly from increased clock frequency
    - By simplifying the cores, some performance is lost, but much more could be gained by adding more cores

Heterogeneous Multicores

- Assume we fit 100 small cores on a given die area
- Assume we have a core which is
  - 2 times faster, but
  - Uses 10 times the die area
- We can consider
  - A) 100 cores (all small)
  - B) 91 cores (90 small, 1 large)
- Using Amdahl's law...
- If a parallel application has a serial fraction of 10%, then
  - A) MaxSpeedup = 1/(0.1 + 0.9/100) = 9.17 times faster
  - B) MaxSpeedup = 1/(0.1/2 + 0.9/90) = 16.67 times faster
- Yes, heterogeneous architectures might pay off

Heterogeneous Multicores

- Cell Broadband Engine architecture overview
- 1 PPE
- 8 SPEs



Yields

- Yield – percentage of processors that are functional
  - With reduced component sizes yields go down
  - STI Cell BE initially reported to have a yield of only 10-20%
- Multicores may be useful even if not all cores are working
  - Sony ships PS3 with only 7 of the 8 SPEs in the Cell BE working
  - Sun markets 4, 6, and 8 processor versions of UltraSPARC T1 based on the yields of a single 8 processor design

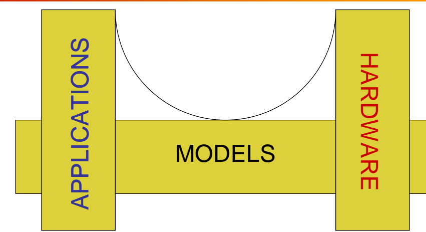
Memory Bandwidth

- *Simple fact: The core can not work faster than the memory is able to supply it with the data it needs*
- If we increase the number of cores without increasing memory bandwidth we will eventually hit a wall (the Memory Wall)
- Memory bandwidth is already a limitation for many applications, such as
  - Some sparse matrix operations
  - Level 1 and 2 BLAS
  - CRC problems
- The memory wall will get more aggravated by multicore architectures

Memory Latency

- Memory latency: time from memory request to memory available at the core
- In a naïve program memory is referenced when it is needed, and the memory latency hurts performance
- Three solutions to memory latency hiding
  - Caches, by storing small memory regions in low-latency memory the effective latency goes down
  - Prefetching, by requesting memory ahead of the instruction that needs it, thereby overlapping communication and communication
  - Simultaneous multithreading (SMT), by switching to another thread the processor can keep working while waiting for data to arrive

Applications/Hardware/Models



1) What are the applications?
2) What are the common kernels of the applications?
3) What are the hardware building blocks?
4) How to connect them?
5) How to describe applications and kernels?
6) How to program the hardware?

Benchmarks and Dwarfs

- Benchmark
  - Code or task supposed to be representative of real-world apps
    - SPEC
    - NAS
    - Intel RMS
    - Linpack
    - HPCC
    - etc
  - Algorithms or given code often favours some type of architecture
    - In particular: a serial benchmark on a multicore architecture...
- Dwarfs
  - Algorithmic or application pattern or kernel
    - Captures functionality and does not overspecify implementation
    - A set of dwarfs that capture most application work as base for benchmarking

A Set of Dwarfs

- The Berkeley View report defines 13 dwarfs:
  - **Dense linear algebra**
  - **Sparse linear algebra**
  - **Spectral methods**
  - **N-body methods**
  - **Structured grids**
  - **Unstructured grids**
  - **MapReduce**
  - Combinatorial logic
  - Graph traversal
  - Dynamic programming
  - Back-track and Branch-and-bound
  - Graphical models
  - Finite state machine

Software Controlled Memory Hierarchy

- Most of the die area currently used for caches
- Cold and conflict misses a consequence of the implicit operation of typical caches
  - Consequence: more capacity needed for effective operation
- It is known that cold and conflict misses can be reduced by using software controlled memory hierarchies
  - Programmer/compiler responsible for memory hierarchy transfers
  - Algorithmic prefetching to hide memory latency
  - Capacity can be reduced, more room for functional units

Core Interconnection Network

- Cores and caches usually connected via crossbars or buses
  - Not scalable to 1000s of cores
    - Too expensive (crossbars)
    - Too inefficient (buses)
- On-chip latency and BW often excellent
- Core-to-core communication could give performance boost and reduce memory traffic
- Scalable interconnection networks needed
  - Cell BE uses 4 ring networks to connect its 9 processors
- More than one network:
- Low-latency network
  - Useful for collectives, which are often (very) short and latency-bound
- High-bandwidth network
  - Useful for point-to-point messages, which are usually quite large and bandwidth-bound
- IBM BlueGene/L follows this principle: a low-latency tree for collectives and a high-bandwidth 3D-torus for point-to-point

Types of Parallelism

- 1) Thread level parallelism (TLP)
  - Coarse-grained
  - Tasks
  - Dependencies
  - Typically explicit by programmer
  - Implemented by loosely coupled cores
  - Multicore

Types of Parallelism

- 2) Data parallelism (SIMD/DLP)
  - Single Instruction Multiple Data
  - SIMD extensions (3DNow!, MMX, SSE)
  - Graphics applications
  - Explicit by intrinsics or assembler coding
  - Implicit by compiler "SIMDization"
  - Typically implemented by tightly coupled cores or functional units

Types of Parallelism

- 3) Instruction level parallelism (ILP)
  - Pipelining
  - Independent instructions
  - Out-of-order execution
  - Hardware optimizations (register renaming)
  - Superscalar architectures
  - Fine-grained
  - Explicit by assembler programming
  - Compilers are good at this
  - Implemented by complex cores and replicated functional units

Automatic Parallelization

- Old and still active research area
- Successful application to vectorization ("SIMDization")
- Experimental proposals like speculative multi-threading
- Can be problematic without user support because of language features such as aliasing
  - A(i) = B(i) + C(i)
    - Easy to vectorize if A, B, C refer to distinct memory regions
    - What if they are aliased (refering to overlapped regions)?

Automatic Parallelization

- *Fundamental limit:*

  The best parallel algorithm could be substantially different from the best serial algorithm

- Consider the AllPrefix Sums problem:
  - Given A(1:N), compute A(1), A(1)+A(2), ... , A(1)+...+A(N)
- (Arguably) the best serial algorithm, O(N) operations:

```
– for( i = 2; i <= N; i++ )
    A[i] += A[i-1];
```

- Does it parallelize well?
  - Loop carried dependency!
- Efficient parallel algorithms employ divide and conquer
  - More work, could be O(N log(N))
  - Less time because of higher level of parallelism
    - With N processors it takes time O(log(N)), a substantial improvement

Why is Parallel Programming Difficult?

- Multithreaded programs
  - Nondeterministic
  - Race conditions
  - Debugging by assuming race condition
  - The root cause for a race often remain unidentified
  - Memory is changing unless read-only, thread local or....
  - ...protected by locks
  - Deadlock is possible when there are multiple, unordered locks
  - Load balancing critical for performance
- Serial programs
  - Deterministic
  - Code coverage testing OK
  - Debugging by tracing execution
  - Finding the bug is harder than fixing the bug
  - Memory is stable
  - ...no, still stable w/o locks
  - No deadlocks
  - The load is perfectly balanced since only one thread

Synchronization

- Parallel tasks need to be synchronized in order to
  - Avoid race conditions (mutual exclusion)
  - Enforce a partial ordering of operations (task dependencies)
- In some applications synchronization is a dominant operation whose performance is critical
  - Fine-grained parallelism
- In order to enhance productivity by reducing the burden on programmers, the ubiquitous lock-based synchronization could possibly be replaced by other models

Lock-based Synchronization

- Race conditions can be avoided by carefully protecting shared memory with locks
- Mutual exclusion enforces deterministic behaviour of locked memory regions
- Different locks for different regions of memory
- **Deadlocks**

```
localSum = 0.0;
for( int i = 0; i < localN; i++) {
   localSum += localArray[i];
}
globalSum += localSum;
```

Unsafe (race condition on globalSum)

```
localSum = 0.0;
for( int i = 0; i < localN; i++) {
   localSum += localArray[i];
}
pthread_mutex_lock(&globalSumLock);
globalSum += localSum;
pthread_mutex_unlock(&globalSumLock);
```

Safe (protected by a lock)

Transactional Memory

- Transactional memory is based on the concept of transactions and in particular:
  - Atomicity, a transaction is either fully completed or it appears to never have happened
  - Serializability, a series of transactions have a serial semantics in that the result corresponds to some serial ordering of the transactions
- Intuitive semantics similar to mutual exclusion

```
localSum = 0.0;
for( int i = 0; i < localN; i++) {
   localSum += localArray[i];
}
atomic {
   globalSum += localSum;
}
```


Transactional Memory

- A lock pessimistically serializes execution
- Transactional memory optimistically allow concurrent execution while maintaining a familiar semantics
- The challenge is to implement this in hardware, but it can be done with some restrictions (see TLR)
- The programmer/compiler still responsible for annotating the code correctly

Programming Models

- A programming model bridges the gap between a developer's natural model of an application and an implementation on actual hardware
  - Ex: message passing, shared memory, pGAS, streaming...
- Obvious tension between:
  - Productivity, and
    - Ease-of-use
  - Efficiency
    - Operations per second
- Tradeoff:
  - Opacity (implicit), versus
    - Abstraction of key HW features (e.g. mapping threads to cores)
  - Visibility (explicit)
    - Key HW features visible to programmer (e.g. data distribution)

Programming Models

- Many models, none of which have proven to be "best"
- General correlations
  - High level of Implicitness: High productivity
  - High level of Explicitness: High efficiency
- Example: Linear Algebra on Shared memory machines
  - Can use OpenMP, high implicitness, generally easier than message passing, but
  - Message passing can sometimes deliver better performance
    - Even though passing messages means overhead

Autotuners

- In some applications there is considerable implementation freedom. Examples:
  - Matrix Multiply (order of loops, blocking, compiler switches)
  - Broadcast (linear, tree-based, fan-out, splitting of messages)
  - ...
- Large space to search for optimal solution
- Not going to be handled well by compilers
  - Complex code, optimizations difficult to get right
  - Hard to keep up with new architectural quirks
- Automatic tuning one solution
  - ATLAS, PHiPAC, FFTW, SpMV
- Research into how to generalize such efforts needed

Conclusion

- Every machine will be a parallel machine
- Moores law continues, we have 8 cores today, soon we might have over 100
- The memory wall will increasingly be a problem
  - Locality, locality, locality...
- Programming models (still) desperately needed
  - While waiting (might be a long wait) we use threads and message passing
  - Pessimism: Caches have been around for a long time, but we still have to design algorithms that explicitly targets caches
  - Optimism: Problems will affect a large portion of developers, so high potential for innovation

Resterande Delen av Kursen

- Fred Gustavson, IBM T.J. Watson Research Center, kommer hit
  - Måndag 7:e maj till fredag 25:e maj
  - Kommer ge föreläsningar om bl.a. linjär algebra, parallelisering, och multicore
- Artikel-seminarium om multicore-arkitekturer
  - Varje grupp får en arkitektur att studera
  - Presentationer inför de andra
  - Möjlighet för både er och oss att lära om CMP-arkitekturer
- Mer information ges löpande
  - Hela kursbokens material behandlas på tentan (även delar som inte tas upp på föreläsningar)