Lecture 8: Parallel Computers

So far, we have only considered single-processor systems. To the user, it appears that only one instruction at a time is being executed, although we know that the microarchitecture can exploit some instruction-level parallelism to speed things up.

However, there is another model of computation that has been around for almost as long as computing itself. Parallel computers allow the user to explicitly manage data and instructions on multiple processing elements. Many workloads exhibit large amounts of parallelism at larger granularities than the instruction-level.

Today, we will see:

- Basics of parallel computers
  - Diminishing returns from ILP
- Shared memory versus message passing
  - Parallel software models
  - Communication and synchronization
  - Message-passing and cluster computing
  - Shared-memory multiprocessors
  - Cache coherence
  - Granularity and cost effectiveness
- Future of computer architecture
  - What to do with a billion transistors and slow wires
  - Single-chip multiprocessors
  - Exploiting fine-grain threads
  - Processor with DRAM (PIM)
  - Reconfigurable processors

Parallel Computing

Why should we consider parallel computing as opposed to building better single-processor systems?

- Diminishing returns of ILP
  - Programs have only so much instruction-level parallelism.
  - It's very expensive to extract the last bit of ILP.
  - We are limited by various bottlenecks like fetch bandwidth, memory hierarchy, etc. that are hard to parallelize in a single-threaded system.
- Peak performance increases linearly with more processors. If a problem is nearly perfectly parallelizable, a parallel system with N processors can run the program N times faster. In the single-processor world, we're very happy if we can get 2 IPC. But with parallel computers, some programs can get N IPC where N is as large as you like, if you can afford N processors.
- Adding processors is easier. Rather than changing a single-processor through expensive re-design of the microarchitecture, we can simply add many of the same old processors together. We still have to design the logic to glue them together, but that is easier. We can generalize this to cluster computers that are connected by commodity communication technologies.
- Adding processors is cheap. I can buy a dual-processor Athlon MP 1500 with a motherboard for $390.00, vs. $439 for the top-of-the-line Athlon XP 2700. For parallelizable workloads, the cheaper dual-processor system will be significantly faster.
- *But* if the workload has a large serial component, the extra parallelism will not give us a linear speedup.

Parallel Software Models

There are two main parallel software models, i.e., two ways the multiple processing elements (*nodes*) are represented to the programmer. They differ in how they communicate data and how they synchronize. Synchronization is essentially how processors communicate control flow among nodes.

- Message passing.
  - Processors fork processes, typically one per node.
  - Processes communicate only by passing messages.
    - Message-passing code is inserted by the programmer or by high-level compiler optimizations. Some systems use message passing to implement parallel languages.
    - Send a message to some specific process using a unique tag to name the message. `send(pid, tag, message);`
    - `receive(tag, message);` Wait until a message with a given tag is received.
  - Processes can synchronize by blocking on messages. The processor waits until a message arrives.
- Shared memory.
  - Processors fork threads.
    - Threads all share the same address space.
    - Typically one thread per node.
    - Threads are either explicitly programmed by the programmer or found through high-level analysis by the compiler.
  - Threads communicate through the shared address space.
    - Loads and stores.
    - Cache contents are an issue.
  - Threads synchronize by atomic memory operations, e.g. test-and-set.

Message Passing Multicomputers and Cluster Computers

This kind of architecture is a collection of computers (nodes) connected by a network. The processors may all be on the same motherboard, or each on different motherboards connected by some communication technology, or some of each.

- Computers are augmented with fast network interface
  - send, receive, barrier
  - user-level, memory mapped
- Computer is otherwise indistinguishable from a conventional PC or workstation.
- One approach is to network many workstations with a very fast network. This idea is called "cluster computers." Off-the-shelf APIs like MPI can be used over fast Ethernet or ATM, or other fast communication technology.

Shared-Memory Multiprocessors

These systems have many processors but present a single, coherent address space to the threads.

- Several processors share one address space.
  - Conceptually like a shared memory.
  - Often implemented just like a message-passing multicomputer.
  - E.g. Address space distributed over private memories.
- Communication is *implicit*.
  - Read and write accesses to shared memory locations.
  - Simply by doing a load instruction, one processor may communicate with another.
- Synchronization
  - Via shared memory locations.
  - E.g. spin waiting for non-zero.

- Or e.g. special synchronization instructions

Cache Coherence

Cache coherence is an issue with shared memory multiprocessors. Although the system conceptually uses a large shared memory, we know what is really going on behind the scenes: each processor has its own cache.

- With caches, action is required to prevent access to *stale* data.
  - Some processor may read old data from its cache instead of new data from memory.
  - Some processor may read old data from memory instead of new data in some other processor's cache.
- Solutions
  - No caching of shared data.
  - Cache coherence protocol.
    - Keep track of copies
    - Notify (udpate or invalidate) on writes
    - Ignore the problem and leave it up to the programmer.

Granularity and Cost-Effectiveness of Parallel Computers

- Parallel computers built for:
  - Capability - run problems that are too big or take too long to solve any other way. Absolute performance at any cost. Predicting the weather, simulating nuclear bombs, code-breaking, etc.
  - Capacity - get throughput on lots of small problems. Transaction processing, web-serving, parameter searching.
- A parallel computer built from workstation size nodes will always have a lower performance/cost than a workstation.
  - Sublinear speedup
  - Economies of scale
- A parallel computer with less memory per node can have better performance/cost than a workstation

Future of Multiprocessing

Moore's Law will soon give us billions of transistors on a die, but with relatively slow wires. How can we build a computer out of these components?

- Technology changes the cost and performance of computer elements in a non-uniform way.
  - Logic and arithmetic is becoming plentiful and cheap.
  - Wires are becoming slow and scarce.
- This changes the trade-offs between alternative architectures.
  - Superscalar doesn't scale well.
  - Global control and data lead aren't good when we have slow wires.
- So what will the architectures of the future look like?

Single-chip Multiprocessors

A quote from Andy Glew, a noted microarchitect:
"It seems to me that CPU groups fall back to explicit parallelism when they have run out of ideas for improving uniprocessor performance. If your workload has parallelism, great; even if it doesn't currently have parallelism, sometimes occasionally it is easy to write multithreaded code than single threaded code. But, if your workload doesn't have enough natural parallelism, it is far too easy to persuade yourself that software should be rewritten to expose more parallelism... because explicit parallelism is easy to microarchitect for."

- Build a multiprocessor on a single chip.
- Linear increase in *peak* performance.
- Advantage of fast communication between processors, relative to going off-chip.
- But memory bandwidth problem is multiplied; each processor will be making demands on the memory system.
- Exploiting CMPs. Where will the parallelism come from to keep all of these processors busy?
  - ILP - limited to about 5.
  - Outer-loop parallelism, e.g., domain decomposition. Requires big problems to get lots of parallelism.
  - TLP (thread-level parallelism). Fine threads:
    - Make communication and syncronization very fast (1 cycle)
    - Break the problem into smaller pieces
    - More parallelism
- Examples of CMPs:
  - Cell. This processor developed by IBM has one Power Processor Element (PPE) and eight Synergistic Processing Elements (SPE) linked by a high-speed bus on chip. The PPE is a POWER-compatible processor that acts as a controller for the SPEs and performs the single-threaded component of the program. The SPEs are 128-bit SIMD RISC processors, i.e., RISC processors with vector instructions. Each SPE has its own local memory that can also be accessed by the PPE. Its peak computing power for single-precision floating point should far outrun other platforms such as Intel. However, it is hard to program.
  - Niagara. This processor developed by Sun Microsystems has 4, 6, or 8 processor cores on a single chip. Each processor can run 4 independent threads. It's kind of like a supercomputer on a chip. Each core is simple compared with contemporary single-core CPUs. They are inorder, have small caches, and no branch prediction.
  - Athlon 64 X2. This line of processors developed by AMD has two processor cores on a single chip. Each processor has a private cache. They are connected by a HyperTransport bus.
  - POWER4 and POWER5 from IBM are dual core.
  - Intel's Pentium D, Pentium 4 Extreme Edition, and Core Duo are dual-core processors.
  - There are others.

Processors with DRAM (PIM)

- Main idea - put the processor and main memory onto a single chip.
- Much lower memory latency.
- Much higher memory bandwidth.
- But it's kind of weird.
- Graphics processors (GPUs) make use of this kind of idea.

Reconfigurable Processors

- Adapt the processor to the application.
  - Special functional units.
  - Special wiring between functional units.
- Builds on FPGA technology (field programmable gate arrays).
  - FPGAs are inefficient.
  - A multiplier built from an FPGA is about 100x larger and 10x slower than a custom multiplier.
  - Need to raise the granularity from the gate-level. Configure ALUs, or whole processors.
- Memory and communication are usually the bottleneck. Not addressed by configuring a lot of ALUs.