Machines & parallelism
Moore’s Law

- Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

This roughly means we have more and more transistors. What can we do with them?
Traditional use of transistors

- Build a more complex, more sophisticated, and faster core
  - Applications will be faster by default
  - Hide complexity from users, leave it in the hardware
- Hit the walls
  - ILP: not enough parallelism to keep it busy
  - Power: high density of transistors $\Rightarrow$ heat
  - Memory: much slower than processors $\Rightarrow$ stalls for data
New ways to use transistors

- Build multi-cores
  - Simpler, less sophisticated cores
  - More cores per die
  - Share complexity with users, not only hardware
  - Applications NOT faster by default

- “Multi-core revolution”
  - Every machine will soon be a parallel machine
  - Can applications use this parallelism?
    - Do they have to be rewritten from scratch?
More exotic solutions

- **Accelerators**
  - Graphics Processing Units (GPUs), e.g., NVIDIA and ATI/AMD
  - Game processors, e.g., Cell for PS3
  - Parallel processor attached to main processor, e.g., APU
  - Originally special purpose, getting more general
  - Programming model not yet mature

- **FPGAs – Field Programmable Gate Arrays**
  - More efficient than multicore for some domains
  - Programming challenge now includes hardware design, e.g., layout
Currently ...

- Popular processors (nodes)
  - Multi-core CPUs: Intel, AMD, IBM
  - Accelerators
    - GPUs: NVIDIA, AMD/ATI, Intel
    - MIC: Intel
    - FPGAs: Altera, Xilinx

- Popular machines = combinations of nodes
  - Clusters/grids/clouds
  - Super-computers: Cray, IBM, NVIDIA, Convey
Classify Parallel Architectures

- **Shared Memory**
  - Homogeneous compute nodes
  - Shared address space

- **Distributed Memory**
  - Homogeneous compute nodes
  - Local (disjoint) address spaces

- **Hybrids**
  - Heterogeneous compute nodes
  - Mixed address space(s)
Shared memory machines

- All processors have access to all memory locations
  - Uniform access time: UMA
  - Non-uniform access times: NUMA

- Interconnections (networks)
  - Bus/buses/ring(s)
  - Meshes (all-to-all)
  - Cross-bars
Distributed memory machines

- Every processor has its own local address space
- Non-local data can only be accessed through a request to the owning processor
- Access times (may) depend on distance => non-uniform
Virtual Shared Memory

- Virtual global address space
  - Memories are physically distributed
  - Local access remains faster than remote access
    - NUMA
  - All processors can access all memory spaces
  - Hardware (or software) hide memory distribution
Hybrids

- Various combinations between shared and distributed memory spaces.
  - More flexible in terms of coherence and consistency
  - More complex to program
Major issues

- **Shared Memory model**
  - Scalability problems (interconnect)
  - Programming challenge: RD/WR Conflicts

- **Distributed Memory model**
  - Data distribution is mandatory
  - Programming challenge: remote accesses, consistency

- **Virtual Shared Memory model**
  - Significant virtualization overhead
  - Easier programming

- **Hybrid models**
  - Local/remote data more difficult to trace
Examples

- Multi-core CPUs?
  - Shared memory with respect to system memory
  - Hybrid when taking caches into account

- Clusters?
  - Distributed memory
  - Could be shared if middleware for virtual shared space is provided

- Supercomputers?
  - Usually hybrid

- Accelerated machines?
  - Distributed for traditional, off-chip GPUs
  - Shared for new APUs
Core-level parallelism
Intra-Processor parallelism

- Low-level parallelism
  - Hidden from programmers
    - Should be automatically addressed by compilers
    - Low-level languages do expose it, if needed
  - Example 1: Pipelining
  - Example 2: Vector / SIMD operation
  - Various other parallelism sources might exist
    - but are machine- and/or vendor-specific

- Memory hierarchies
  - Caches and non-caches alike
Pipelining

David Patterson’s Laundry example:
- 1 load = wash (30mins) + dry (40mins) + fold (20mins)

1 laundry room, 4 people, 1 load each

Quiz:
- What is the speed-up using pipelining?
- What is the actual latency of the system? What is the peak latency?
- What is the actual bandwidth (loads/h)? What is the main limitation of the bandwidth? What is the peak bandwidth?
Pipelining examples

- **Theory**
  - Fetch = instruction read from memory to register
  - Decode = understand instruction and operands
  - Execute = employ execution units to do the op.
  - Write = write the results
In practice, pipelines are a lot more complicated:

- **AMD Bobcat**
- **Intel Core i7**
This complexity is hidden by the hardware and the compiler.

BUT: writing better code in terms of the instruction mix can impacts the performance of the compiler.
SIMD (vector operations)

- **Scalar processing**
  - Traditional
  - One operation produces one result

- **Vector processing**
  - With SSE, SSE2 … , AVX
  - One operation produces 4 results

\[
\begin{align*}
A & + B = C \\
A1 & + B1 = C1 \\
A2 & + B2 = C2 \\
A3 & + B3 = C3 \\
A4 & + B4 = C4
\end{align*}
\]
Assembly instructions
- 16 (or more) registers
- C/C++ intrinsics = “macro’s” to work on variables

```c
float data[N];
for (i=0; i<N; i++) data[i]=(float)i;
//vector: load first 4 elements, then next 4
__m128 myVector0=_mm_load_ps(data);
__m128 myVector1=_mm_load_ps(data+4);

//vector: add => 4 FLOPs
__m128 myVector2=_mm_add_ps(myVector0,myVector1);

// vector: _MM_SHUFFLE(x,y,z,t)=x,y from 1,z&t from 2
__m128 myVector3=_mm_shuffle_ps(myVector0, myVector1,
    _MM_SHUFFLE(2,3,0,1));
```
Vector addition with SSE

float A[N], B[N], C[N];
for (i=0; i<N; i++) C[i] = A[i]+B[i]

- **Step 1: loop unrolling**

float A[N], B[N], C[N];
for (i=0; i<N; i+=4) {
    C[i] = A[i]+B[i];
    C[i+1] = A[i+1]+B[i+1];
    C[i+3] = A[i+3]+B[i+1];
}

- **Step 2: use of intrinsics**

float A[N], B[N], C[N];
for (i=0; i<N; i+=4) {
    __m128 vecA = __mm_load_ps(A+i);
    __m128 vecB = __mm_load_ps(B+i);
    __m128 vecC = __mm_add_ps(vecA, vecB);
    __mm_store_ps(C+i, vecC); }
SSE vs. AVX

- SSE/SSEn is Intel-specific
- AVX in an extension of SSE
  - To be supported by more vendors
  - Currently: AMD and Intel
- AVX fact sheet:
  - Larger registers: 256b instead of 128b
  - Supports legacy instruction (lower 128b)
  - Adds support for 3-operand instructions
Memory performance

- Flat memory model
  - All accesses = same latency
  - Memory latency slower to improve than processor speed

... which means we wait longer for any access to the (DRAM) memory!

The gap grows ~50% per year
Solutions

- Improve latency
  - Technology
  - Memory hierarchies

- Make better use of bandwidth
  - Bandwidth increases 3x faster than latency!
  - Collect/gather/coalesce multiple memory accesses

- Overlap computation and communication
  - Prefetching
  - Default/automated (low-level)
  - Software-managed (aka, programmer implemented …)
Hierarchical memory model

- Several memory spaces
  - Large size, low cost, high latency – main memory
  - Small size, high cost, low latency – caches / registers

Main idea: Bring some of the data closer to the processor

- Smaller latency => faster access
- Smaller capacity => not all data fits!

Who can benefit?

- Applications with locality in their data accesses
  - Spatial locality
  - Temporal locality
Limitations
- Size: no space for every memory address
- Organization: what gets loaded & where?
- Policies: who’s in, who’s out, when, why?

Performance
- Hit = access found data in fast memory => low latency
- Miss = data not in fast memory => high latency + penalty
- Metric: hit ratio (H) = the fraction of accesses that hit
- Performance gain: ?
  - $T_{nocache} = N \times T_{main\_memory}$
  - $T_{cache} = (N-H)(T_{main\_memory}+penalty) + H \times T_{cache}$
Why do memory hierarchies work?

- **Temporal locality**
  - RD(x), RD(x), RD(x):
    - Main_RD, Cache_hit, Cache_hit
    - Gain: Cache latency is better!

- **Spatial locality**
  - RD(x), RD(x+1), RD(x+2):
    - Main_RD, Cache_hit, Cache_hit
    - Gain: Cache latency is better + better bandwidth for coalesced operations RD(x,x+1,x+2, ... )
Memory performance in practice

- Theoretical:
  - Peak latency & bandwidth are given
  - … but in theoretical optimal conditions

- Practice:
  - Microbenchmarking of the memory system
    - E.g.: Membench
    - … or build your own benchmark
Inter-core parallelism
Multi-core processors

- **Architecture**
  - Few fat cores
  - Homogeneous
  - Stand-alone

- **Memory**
  - Shared, cached
  - Per-core cache
  - Multiple caching layers

- **Parallelism**
  - Multiple threads of execution
  - SPMD or MPMD

- **Performance gain**
  - Concurrency, coarse-grain parallelism
  - Latency-oriented
Intel “traditional”

Nehalem-EX
- 2.26 GHz Nehalem Core 0
- 32KB L1D Cache
- 256KB L2 Cache
- 24MB L3 Cache
- DDR3 Memory Controllers
- Quick Path Interconnect
- 4x2B @ 6.4GT/s
- 8x20b @ 6.4GT/s

Westmere
- 3.33 GHz Westmere Core 0
- 32KB L1D Cache
- 256KB L2 Cache
- 12MB L3 Cache
- DDR3 Memory Controllers
- Quick Path Interconnect
- 3x8B @ 1.33GT/s
- 4x20b @ 6.4GT/s
AMD “traditional”
Intel’s current generation

Sandy Bridge Client

- 3.4 GHz Sandy Br Core 0 (Est)
- 32KB L1D Cache
- 256KB L2 Cache
- 8MB L3 Cache
- Direct Media Interface
- PCI-E 2.0 Controller
- DDR3 Memory Controllers

- 0.85 GHz Gen 6 GPU (Est)
- 2x8B @ 2.13GT/s (Est)
- 2x4b @ 2.5GT/s
- 2x16b @ 5GT/s
AMD Fusion

Llano

- 2.9 GHz Llano Core 0
- 1MB L2 Cache
- ... Coherent Request Queues (IFQ)
- DDR3 Memory Controllers

- 2.9 GHz Llano Core 3
- 1MB L2 Cache

- 0.6 GHz Cypress 5 Cores

- 2x16B Onion
- 4x32B Garlic

- 2x8B @ 1.86GT/s
Multi-core CPUs programming

- It’s all about multi-threading & SIMD!
- All shared memory models + low-level models
- Typical:
  - Pthreads (or other threading libraries)
  - TBB, ArBB
  - OpenMP
  - OpenCL
- Vectorization/SIMD-ization
  - Significant performance gain
  - Compilers are constantly improving
- Memory accesses
  - Caches play a huge role in performance
Graphical processing units

- **Architecture**
  - Hundreds/thousands of slim cores
  - Homogeneous
  - Accelerator(s)

- **Memory**
  - Complex system, less strict hierarchy
  - Shared per-core, per-processor and available per-unit

- **Programming**
  - Off-load model
  - (Many) Symmetrical threads
  - Hardware scheduler

- **Performance gain:**
  - Fine grain parallelism, bulk data-parallelism
  - Throughput oriented
- SM = streaming multiprocessor
- 1 SM = 8 SP (streaming processors/CUDA cores)
- 1 TPC = 2 x SM / 3 x SM = thread processing clusters
NVIDIA Fermi: under the hood
Cell/B.E.

- **Architecture**
  - Heterogeneous
  - 8 vector-processors (SPEs) + 1 trimmed PowerPC (PPE)
  - Accelerator or stand-alone

- **Memory**
  - Per-core only

- **Programming**
  - Asymmetrical multi-threading
  - User-controlled scheduling
  - 6 levels of parallelism, all under user control
Cell/B.E.

1 x PPE 64-bit PowerPC
- L1: 32 KB I$ + 32 KB D$
- L2: 512 KB

8 x SPE cores:
- Local mem (LS): 256 KB
- 128 x 128 bit vector registers

Main memory access:
- PPE: Rd/Wr
- SPEs: Async DMA
Intel Single-chip Cloud Computer

- **Architecture**
  - Tile-based many-core (48 cores)
  - A tile is a dual-core
  - Stand-alone / cluster

- **Memory**
  - Per-core and per-tile
  - Shared off-chip

- **Programming**
  - Multi-processing with message passing
  - User-controlled mapping/scheduling

- **Gain performance ...**
  - Coarse-grain parallelism (MPMD, SPMD)
  - Multi-application workloads (cluster-like)
Intel SCC
Intel MIC

1st Gen of Intel® Xeon® Phi™ Product Family: Coprocessor codenamed Knights Corner

- In Production in 2012
- >50 cores
- Based on 22nm 3-D TriGate Transistors
- PCIe form factor
- 8GB+ GDDR5 memory
- Intel standard programming model & software tools
- 1 TFLOPS measured performance, Double Precision
- Capable of running complete applications

[Under Embargo until June 18th, 6am PDT]
Intel MIC
Inter-node parallelism
Typical examples

- Clusters
  - See DAS4

- Super-computers
  - Half of top500

- Grids & clouds
  - See Grid5000
  - See Amazon EC2
IBM’s BlueGene/Q

1. Chip: 16+2 µP cores

2. Single Chip Module
   - 16 GB DDR3 Memory
   - Heat Spreader for H₂O Cooling

3. Compute card:
   - One chip module
   - 16 GB DDR3 Memory
   - Heat Spreader for H₂O Cooling

4. Node Card:
   - 32 Compute Cards
   - Optical Modules, Link Chips
   - 5D Torus

5a. Midplane:
   - 16 Node Cards
   - 8 PCIe Gen2 x8 slots
   - 3D I/O torus

5b. IO drawer:
   - 8 IO cards w/16 GB

6. Rack: 2 Midplanes

7. System:
   - 96 racks, 20PF/s

- Sustained single node perf: 10x P, 20x L
- MF/Watt: (6x) P, (10x) L (~2GF/W, Green 500 criteria)
- Software and hardware support for programming models for exploitation of node hardware concurrency

© 2011 IBM Corporation
Machine = Multiple layers of parallelism
  ◦ Intra-core
    • Hidden
      • ILP, pipelining, multiple functional units
    • Exposed / explicit
      • Vectorization/SIMD-ization
    • Memory
      • Caching
  ◦ Inter-core
    • Multiple threads per core
  ◦ Inter-node
    • Multiple processes/threads/jobs/applications