Notes on Executing Programs: What's Under the Hood

- Evolution of a dot product
  - Parse trees and interpretation
  - Computer architecture
  - Compilation
  - Performance
  - Java and incremental compilation
- Memory management is important too…
  - Memory resources
  - Heap issues
- Parallelism

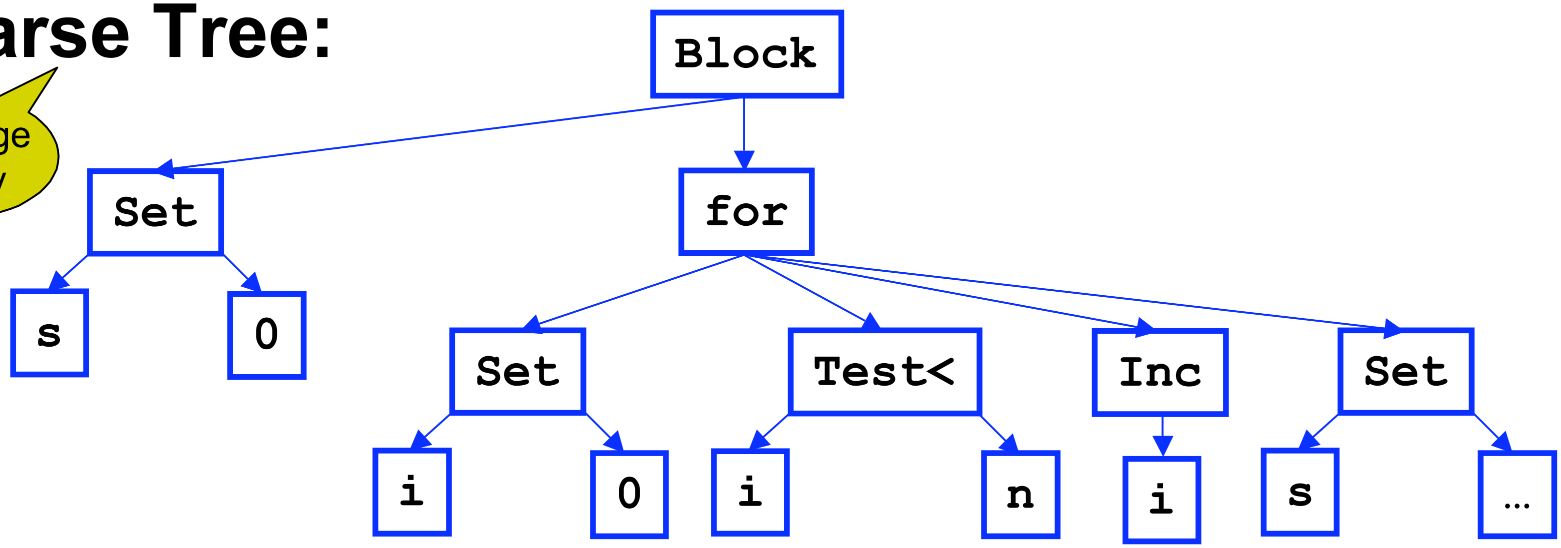
Parsing a Dot Product & Interpretation

Math: s = x.y

Code: {s = 0; for (i = 0; i < n; i++) s += x[i]*y[i];}

Parse Tree:

Language Theory



Interpreter traverses the tree to execute

Ref: en.wikipedia.org/wiki/Programming_language_theory

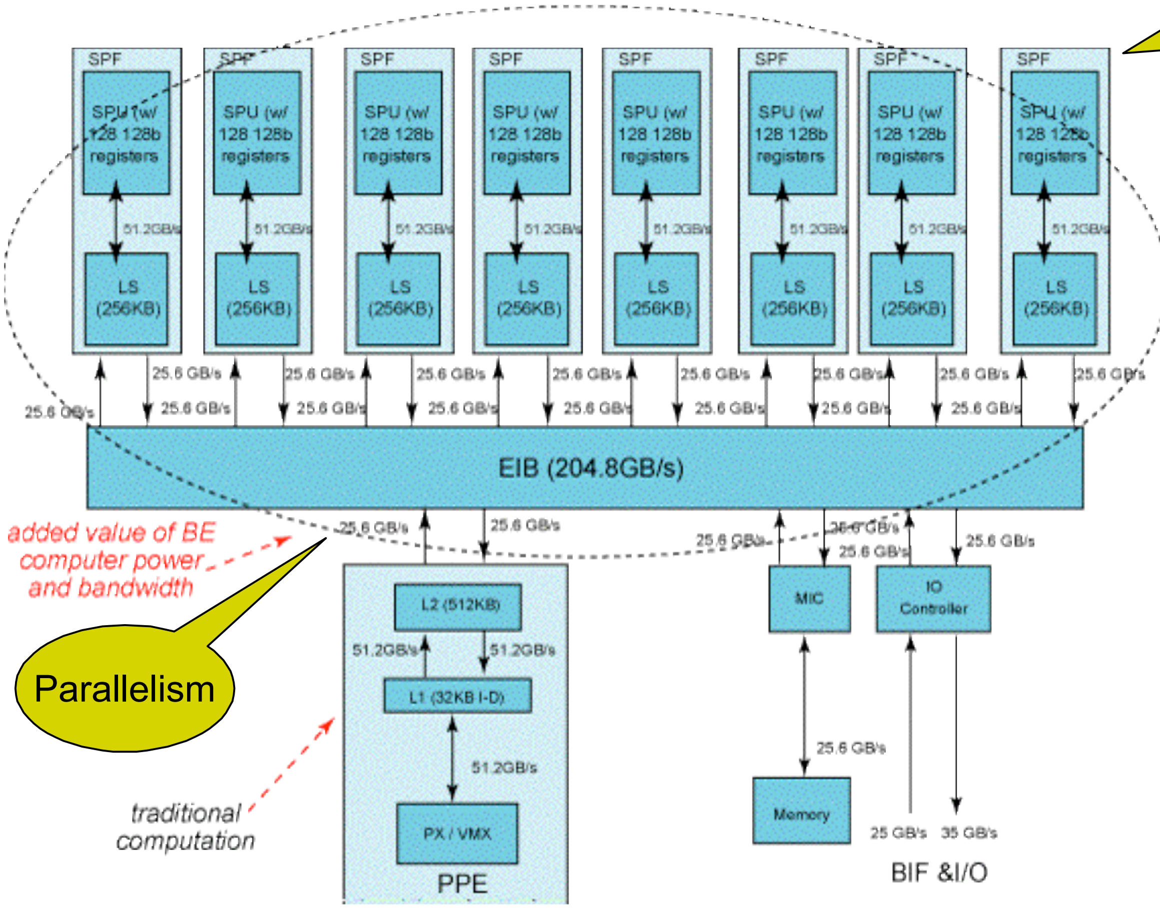
If your program runs fine interpreted on a single core: Great, you're done!

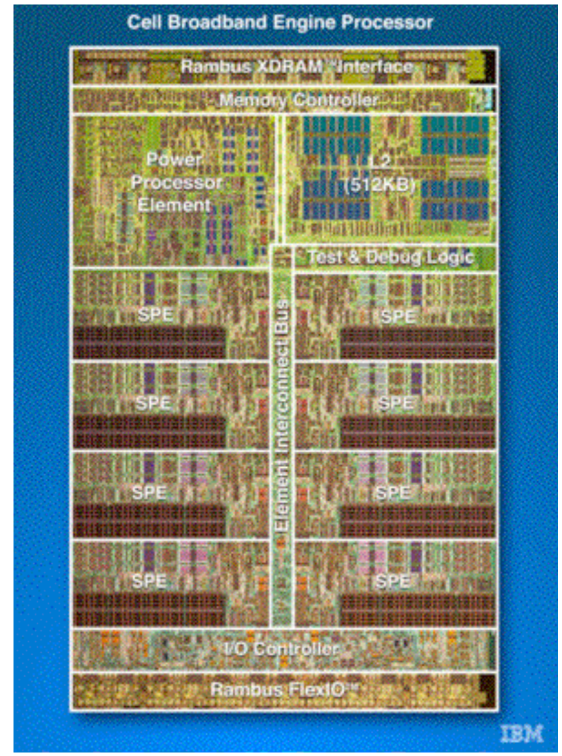
But what about supporting:

- **Analysis of 10^15 byte data sets,**
- **Realistic interactive visualization,**
- **Many large multi-level multi-physics simulations?**

Computer Architecture:
Cell Broadband Engine

Memories& Data Flows





Peter Hofstee

Refs: 1. Lectures by Ed Upchurch and Thomas Sterling
2. Cell Broadband Engine Architecture and its first implementation
www.ibm.com/developerworks/power/library/pa-cellperf/

pdp11

Compilation

CACR

Original Code: {s = 0; for (i = 0; i < n; i++) s += x[i]*y[i];}

Transformed Code: {s = 0; l = x + n; while(x < l) s += *x++ * *y++;}

Assembler Code:

```
; x in r0, y in r1, n in r2, destroys r0-2
; result s in ac0
dotd:
  clrd ac0                  ; s = 0
  tst r2                    ; machine code: 005702 (octal)
  beq L1
  ash #3, r2                ; r2<<3, doubles are 8 bytes
  add r0, r2                ; l = x + i
L2:                         ; start of inner loop
  ldd (r0)+, ac1
  muld (r1)+, ac1; PDP11/70 4680ns (13 cycles)
  addd ac1, ac0             ; 980ns (~3 cycles)
  cmp r0, r2
  blo L2            ; repeat if r0 < r2
L1: ; continue…
```

What the machine sees

Optimizing Compilers

C Syntax

"-O4: Performs aggressive optimizations and correctness not guaranteed"

Performance:
Cycles per Inner Loop

Double Exponential?

- **PDP11/70: ~20**
- **Modern procs.: 1-2**
  - **Pipeline**
  - **Multiple functional units**
- **Interpretation: 3-10x**
  - **generic compilers**
- **Supercomputers: 1/P**
  - **Linpack benchmark**
  - **Top500.org**

Why do many codes not scale?

Ref: For numerical libraries, see Linda Petzold's lecture

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz , Voltaire Infiniband / 2008 IBM	129600	1105.00	1456.70	2483.47
2	Oak Ridge National Laboratory United States	Jaguar - Cray XT5 QC 2.3 GHz / 2008 Cray Inc.	150152	1059.00	1381.40	6950.60
3	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz / 2008 SGI	51200	487.01	608.83	2090.00
4	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
5	Argonne National Laboratory United States	Blue Gene/P Solution / 2007 IBM	163840	450.30	557.06	1260.00
6	Texas Advanced Computing Center/Univ. of Texas United States	Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband / 2008 Sun Microsystems	62976	433.20	579.38	2000.00
7	NERSC/LBNL United States	Franklin - Cray XT4 QuadCore 2.3 GHz / 2008 Cray Inc.	38642	266.30	355.51	1150.00
8	Oak Ridge National Laboratory United States	Jaguar - Cray XT4 QuadCore 2.1 GHz / 2008 Cray Inc.	30976	205.00	260.20	1580.71
9	NNSA/Sandia National Laboratories United States	Red Storm - Sandia/ Cray Red Storm, XT3/4, 2.4/2.2 GHz dual/quad core / 2008 Cray Inc.	38208	204.20	284.00	2506.00
10	Shanghai Supercomputer Center China	Dawning 5000A - Dawning 5000A, QC Opteron 1.9 Ghz, Infiniband, Windows HPC 2008 / 2008 Dawning	30720	180.60	233.47	

Java and Virtual Machines

- ***Machine independent machine language (bytecodes)***
- **Source to .class, *transport* & execute .class (securely)**
- **HotSpot Java Virtual Machine (Java SE, Mac OS X, …)**
- **Incremental compilation**
  - **Can do some optimizations better than static compilers (inline virtual functions)**
  - **With multi-core, the interpreter-compiler distinction may go away. If cores are "free", why not use a few cores to optimize the execution of others? *Introspective execution.***

Research

- **Works for other languages, e.g. Python to .class**
  - **(Proviso: explicit eval)**
- **Ref: The Java HotSpot Perf. Engine Architecture**
  - **java.sun.com/products/hotspot/whitepaper.html**

Memory Resources

```
{
   double s;
   …
   x = new double[n];
   …
}
```

CACR

- Register
  - Zero cycle latency
  - But only 32-256, allocated by compiler
- Stack
  - Main memory, can have long latency*
  - Large, automatic FIFO allocation
- Heap
  - Main memory, long latency*
  - Large, *random allocation, manual or automatic management*
  - Expensive ("Cons-less" programming)
- *Cache: 1 cycle if local (e.g. recent stack frames)

"Was" NP Hard

Memory latency research

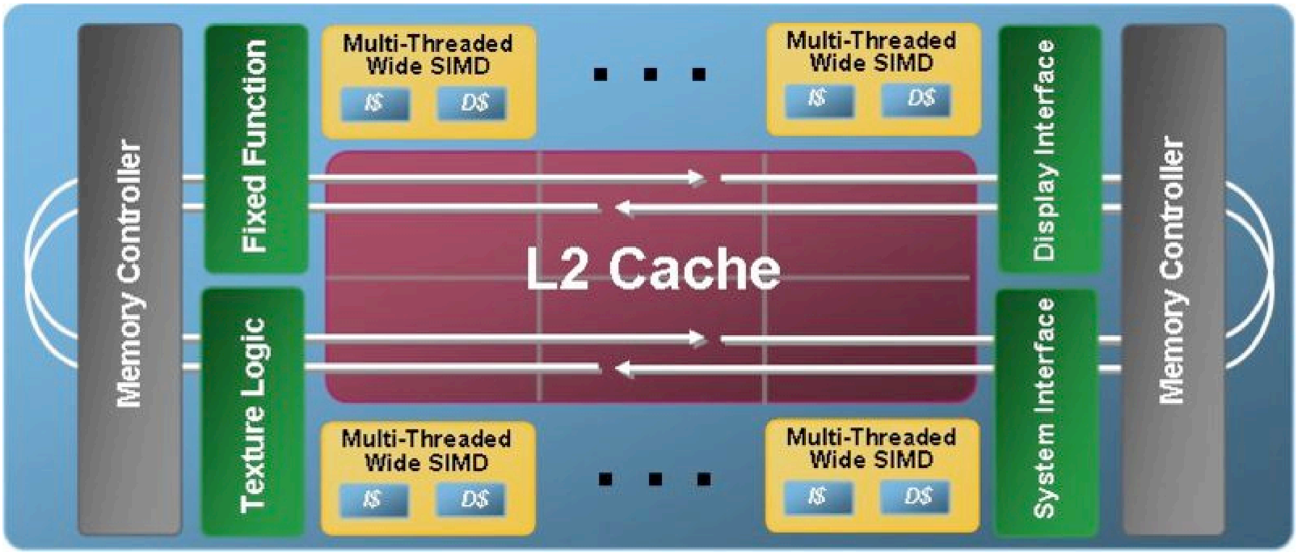
Heap Issues

```
{
   …
   x = new double[n];
   z = x;
   …
}
```

- ~50% of bugs in basically running codes are due to memory allocation/deallocation
  - Leaks & multiple de-allocations (C++)
  - FORTRAN 4 solution
  - Tools like Purify
  - Strong argument for auto allocation schemes (Lisp, Java)
- Garbage collection
  - Mark and sweep (cyclic structures)
  - Execution pauses, background scavenger thread
  - Fragmentation
  - See HotSpot reference

Parallelism:
Many Core

NVIDIA Tesla
128 SIMD cores



Intel Larrabee 2
Many IA32+ cores

Be prepared for P~100 on laptops in a few years
Two orders of magnitude *can be* qualitatively different

Ref: Ct: C for Thoughput Computing
techresearch.intel.com/articles/Tera-Scale/1514.htm

Takeaways

- Programs as transforms
  - Execution
  - Development
- Remember memory: 50% of latent bugs
- Think very parallel, even for laptops
- Optimize time to solution for given resources: cpu, memory, disk, network, software tools, and *people*

Notes on Executing Programs: What's Under the Hood

Ay/Bi 199 Winter 2009

January 13, 2009