A really Brief (and rambling) Introduction to Parallel Computing

Tim Mattson

Intel Corp.

timothy.g.mattson@ intel.com

1

Introduction

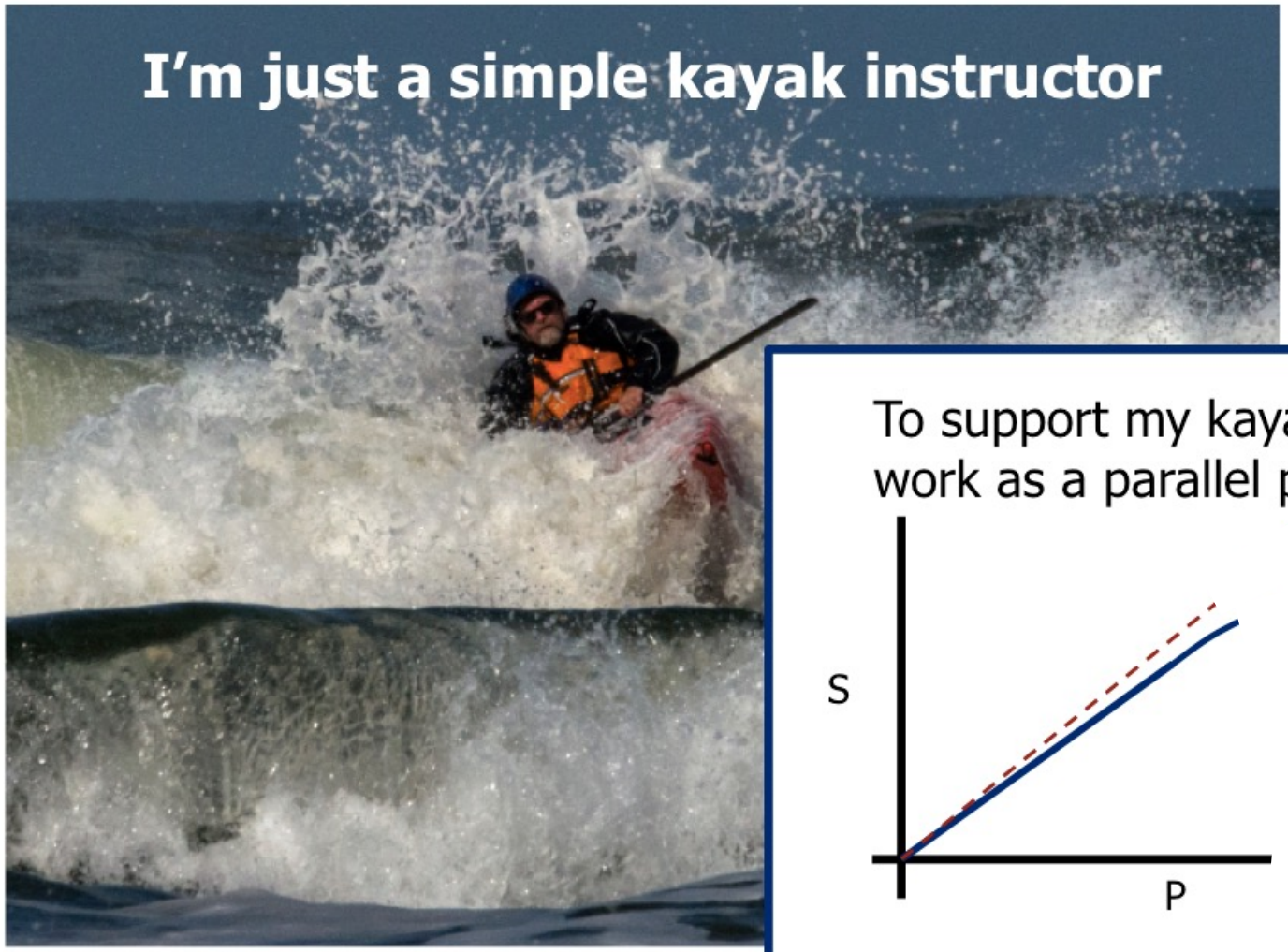


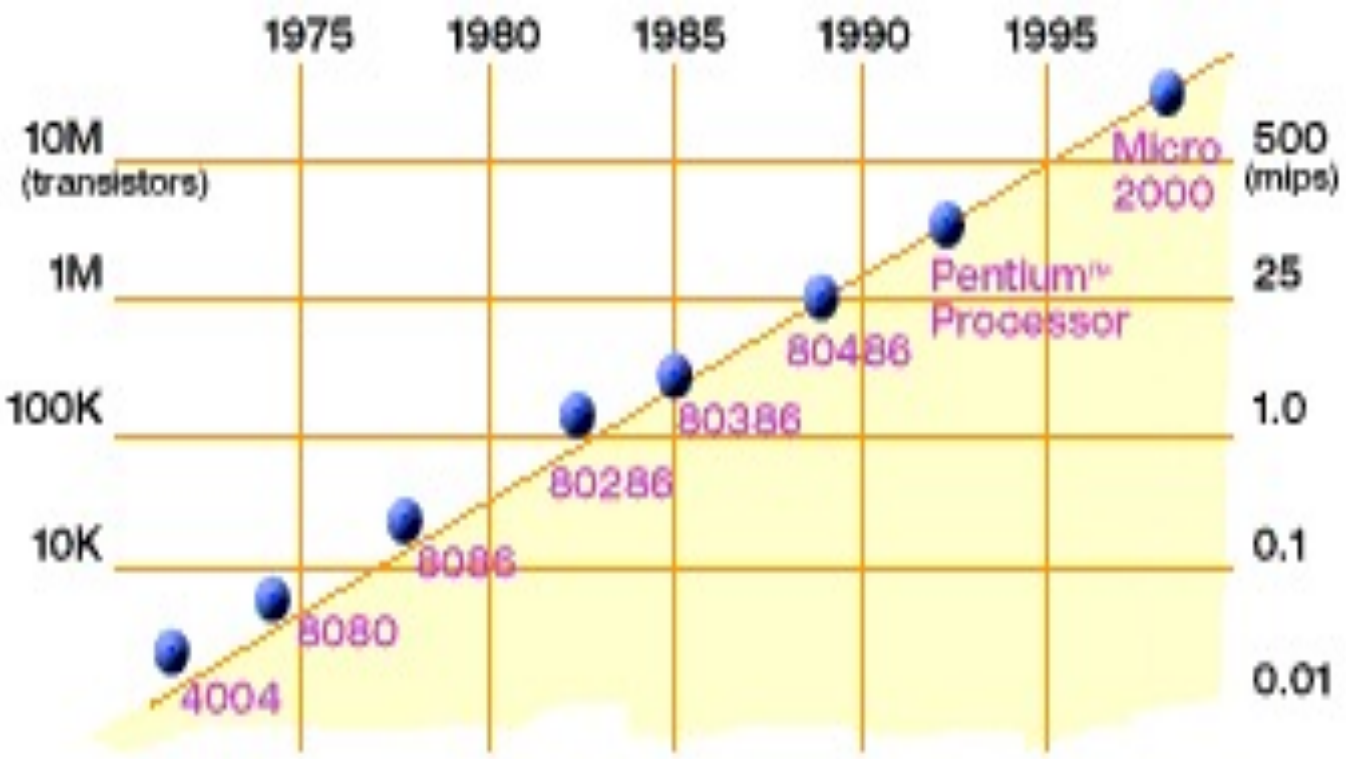
Photo © by Greg Clopton, 2014

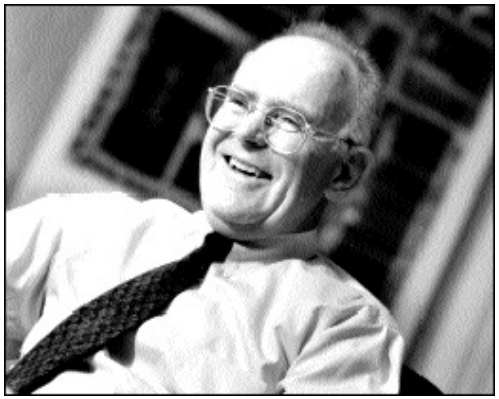
To support my kayaking habit I work as a parallel programmer

Which means I know how to turn math into lines on a speedup plot

Why should you care about parallel programming?

Moore's Law

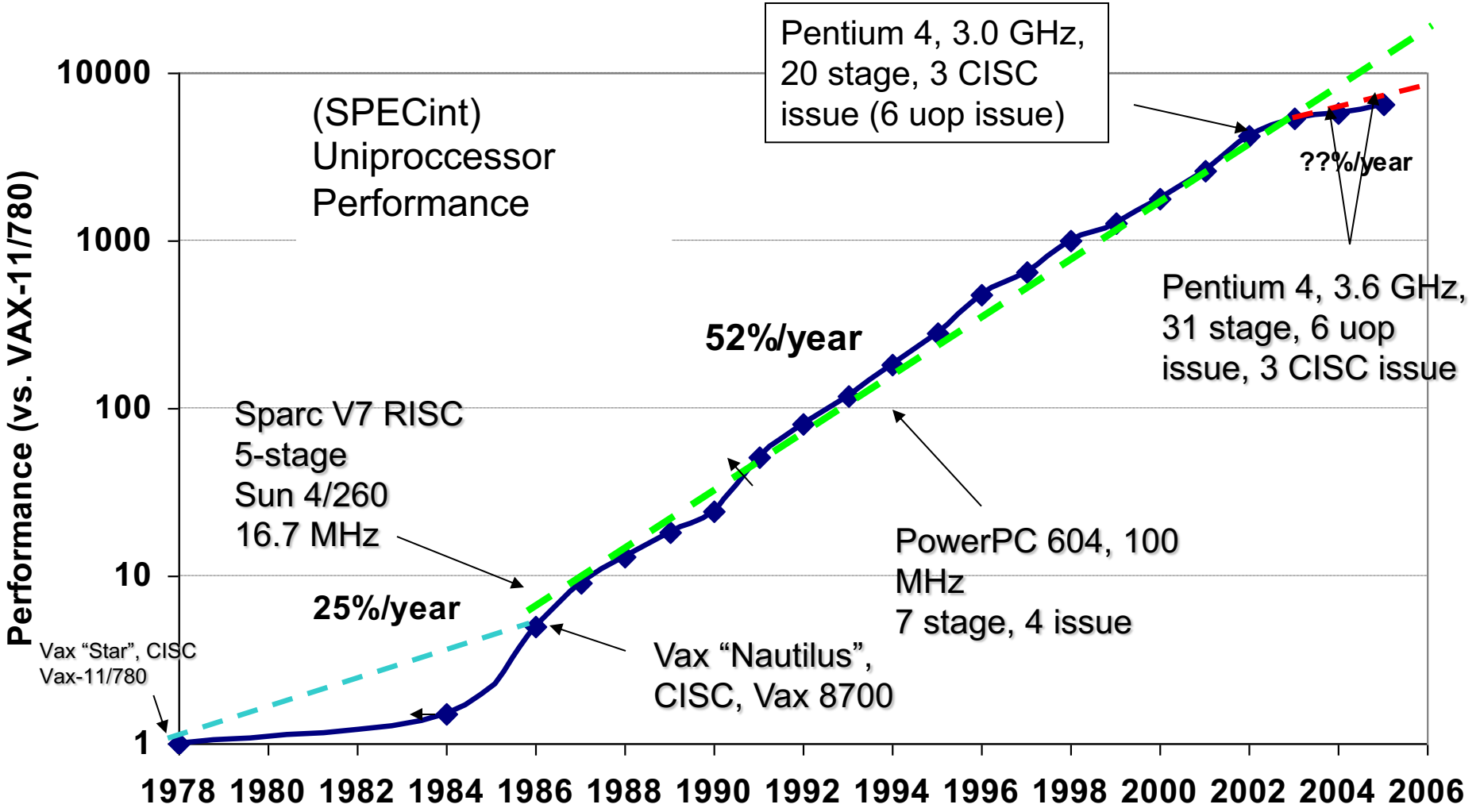




- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Over the last 50 years, transistor densities have increased as he predicted.

Slide source: UCB CS 194 Fall'2010

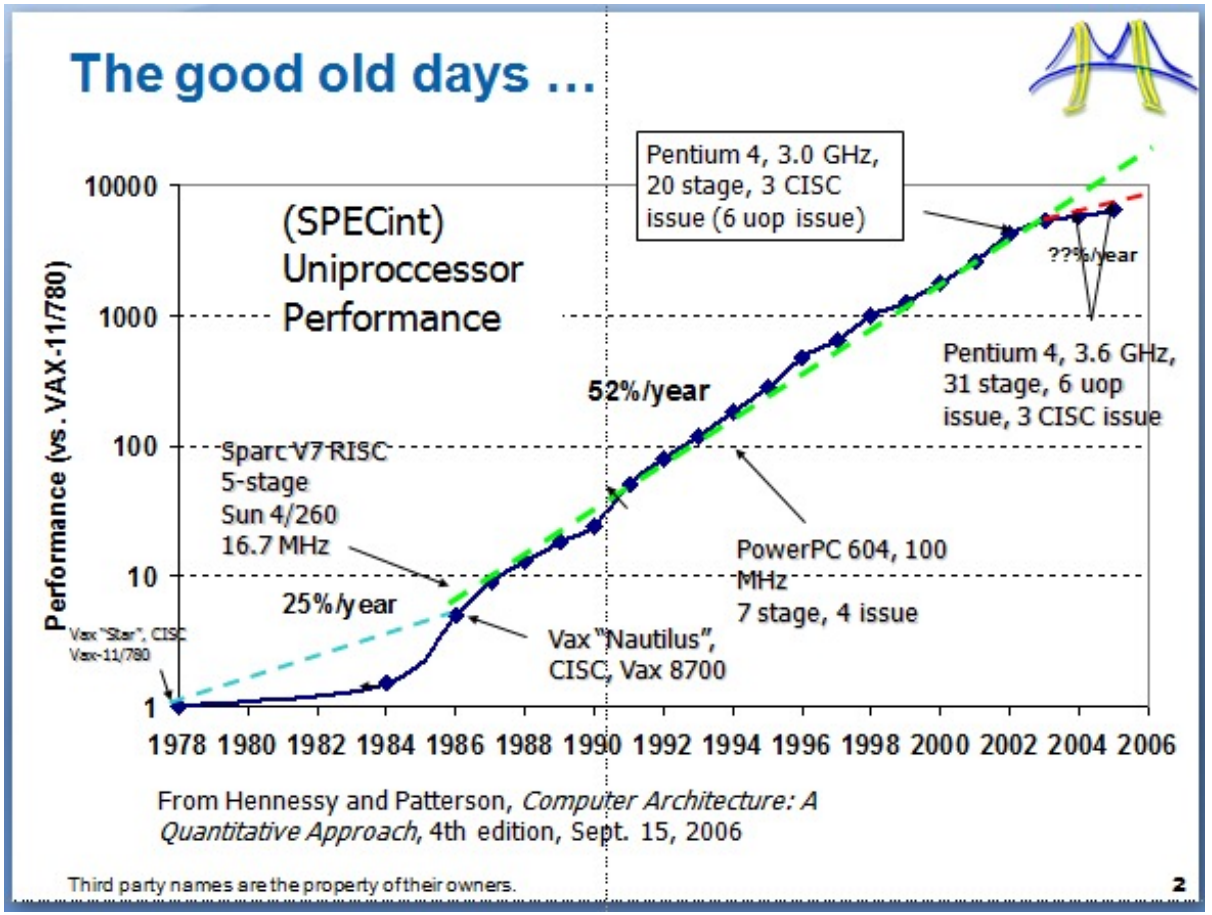
The good old days …



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

The Hardware/Software contract

- Write your software as you choose and the HW-geniuses will take care of performance.



- The result: Generations of performance ignorant software engineers using performance-handicapped languages (such as Python) … which was OK since performance was a HW job.

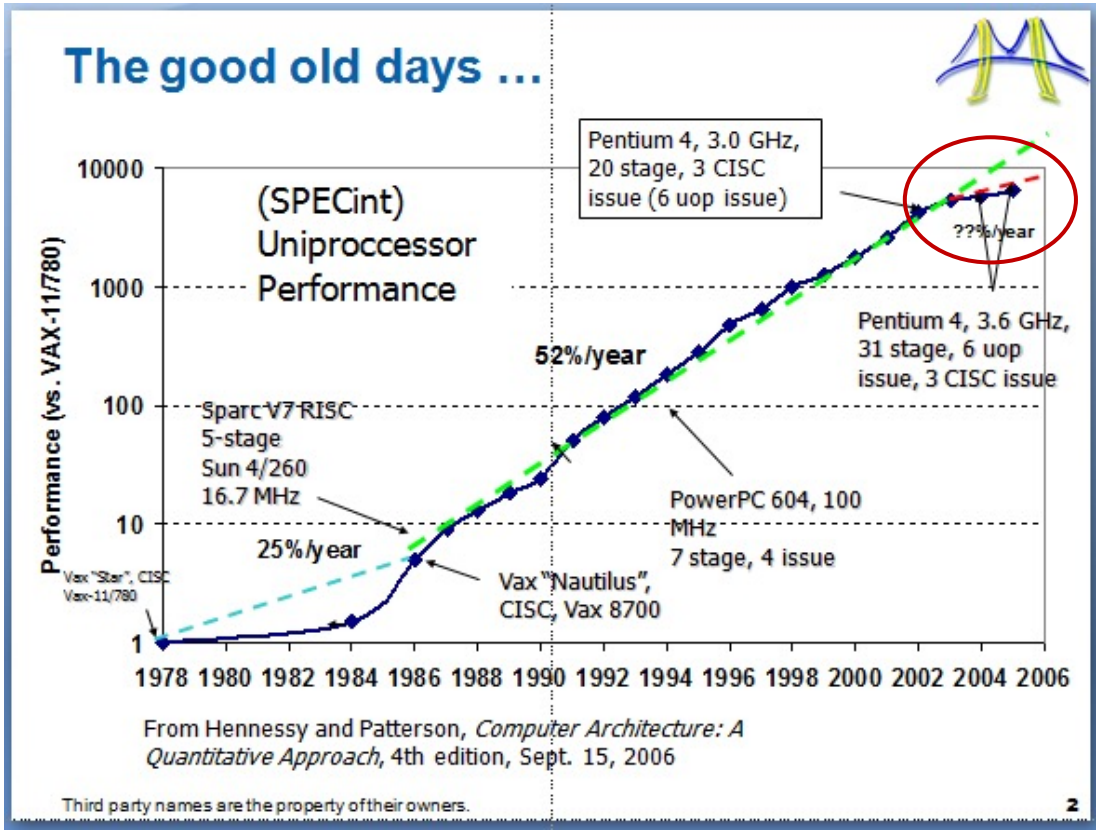
Why the drop off in performance?

Dennard Scaling:

- Transistors shrink, circuit delays go down, frequency goes up, power per transistor goes down.
- Result … if transistor density doubles, circuit is 40% faster for twice as many transistors for fixed power.

Dennard scaling considers "dynamic effects" driven by frequency. It assumes static effects such as leakage are negligible.

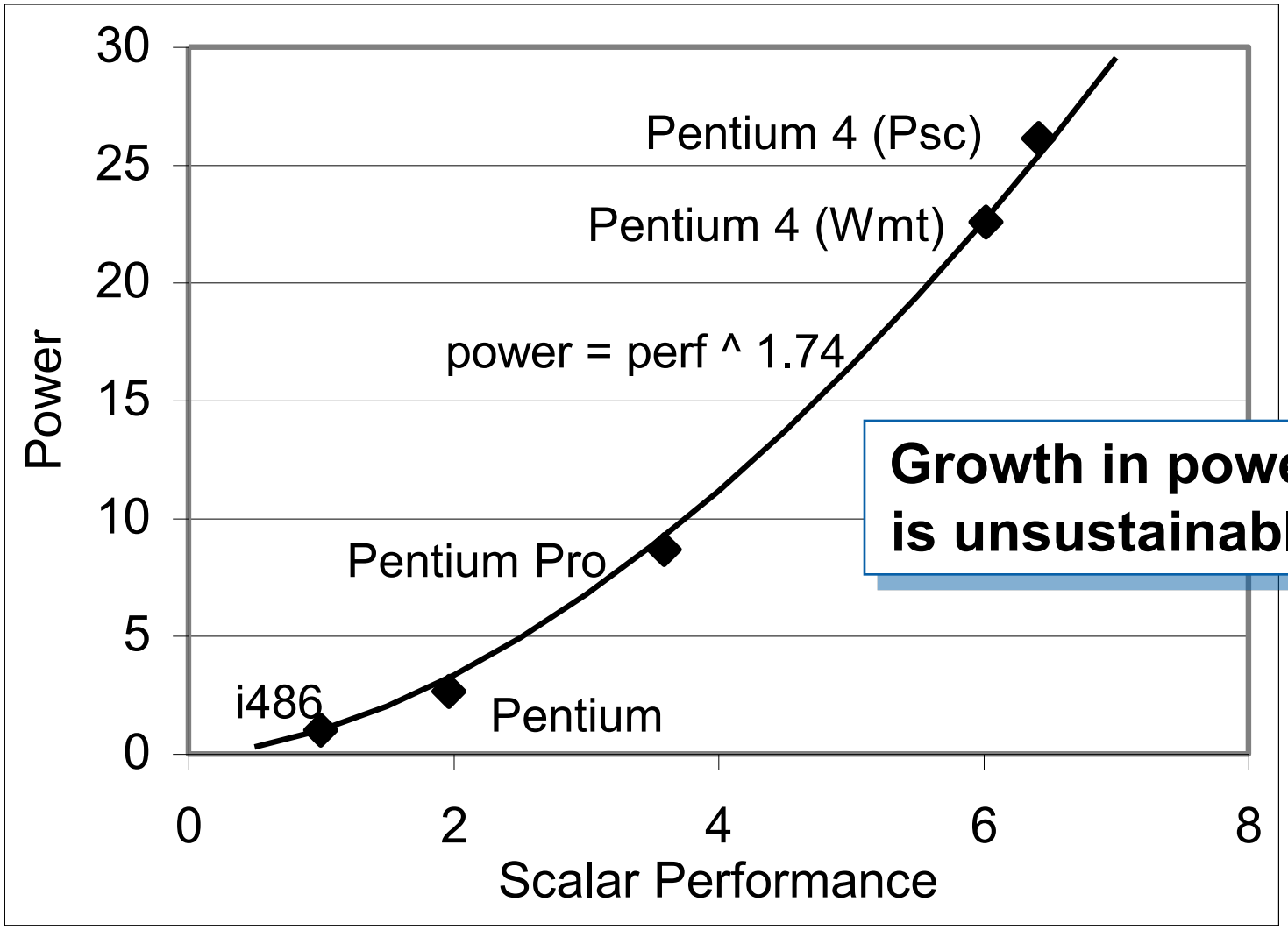
- As transister densities climb, these static effects DO NOT scale and eventually dominate.



- The problem isn't the end of Moore's law. The problem is the end of Dennard scaling. This means with new generations of process technology, chip frequency no longer improves. The free lunch is over.

Modern design emphasizes Power consumption.

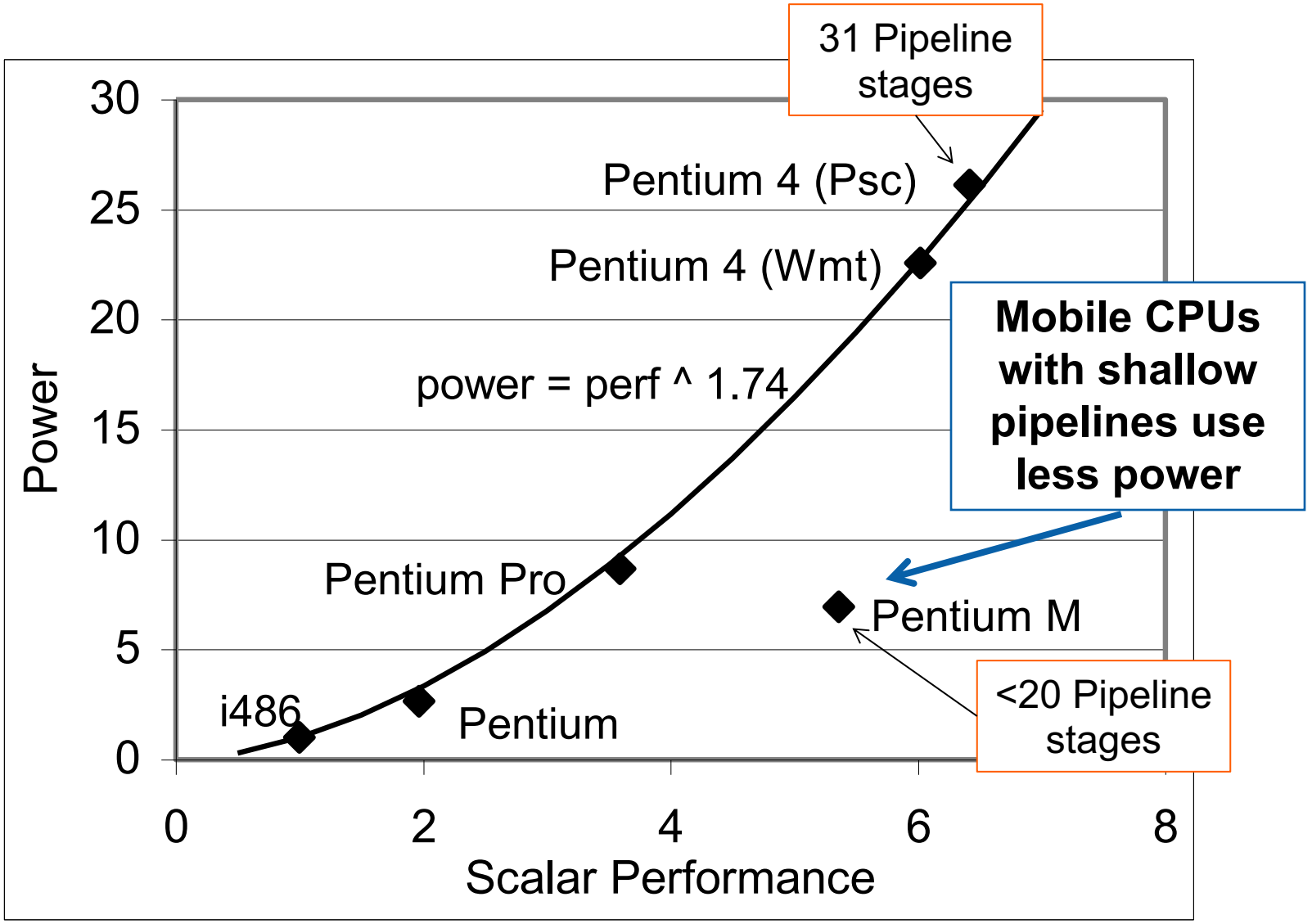
Power vs Performance (normalized to i486 process tech.)



Source: Ed Grochowski, Intel

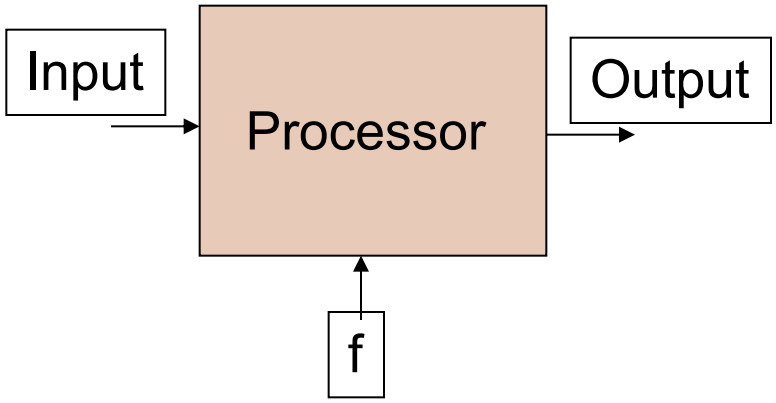
Modern design emphasizes Power consumption.

Power vs Performance (normalized to i486 process tech.)



Source: Ed Grochowski, Intel

Consider power in a chip …



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

C = capacitance … it measures the ability of a circuit to store energy:

$C = q/V \rightarrow \quad q = CV$

Work is pushing something (charge or q) across a "distance" … in electrostatic terms pushing q from 0 to V:

$V * q = W.$

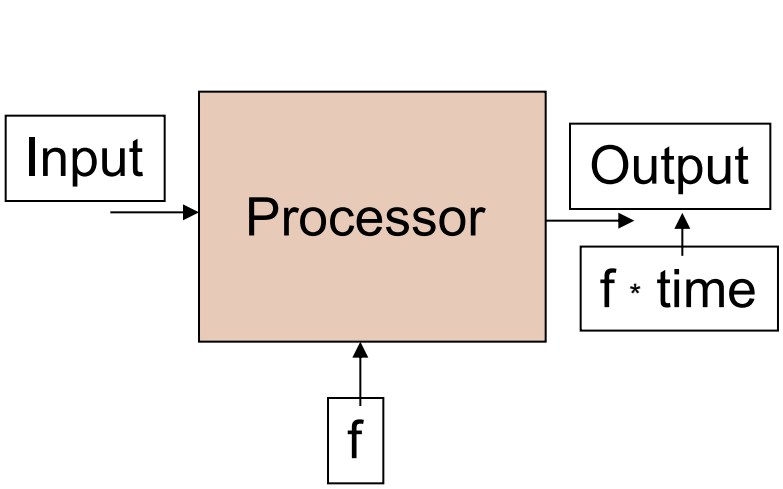
But for a circuit $q = CV$ so

$W = CV^2$

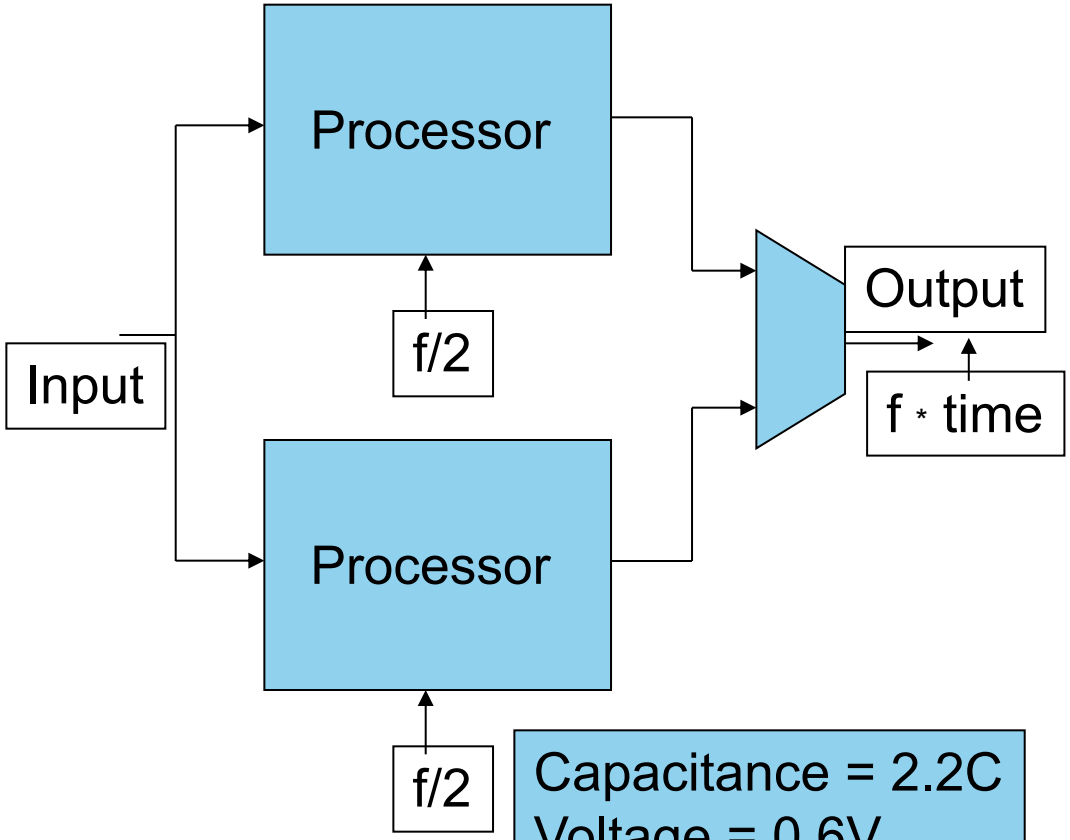
power is work over time … or how many times in a second we oscillate the circuit

Power = W* F → Power = CV^2f

... Reduce power by adding cores



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

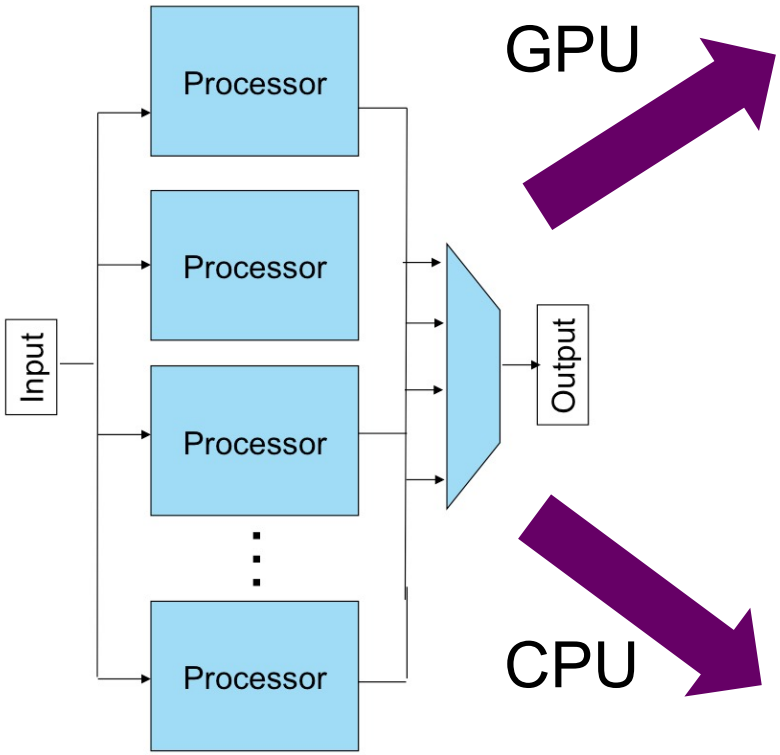


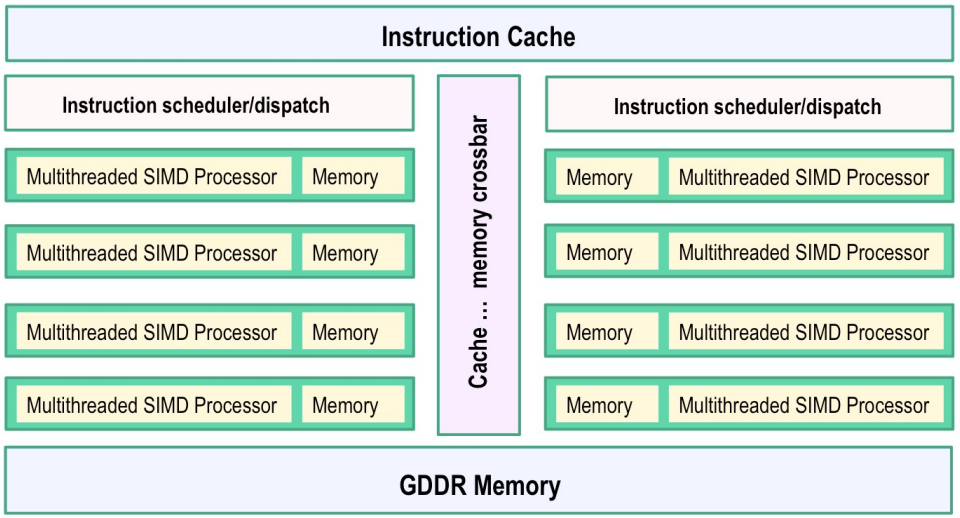
Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

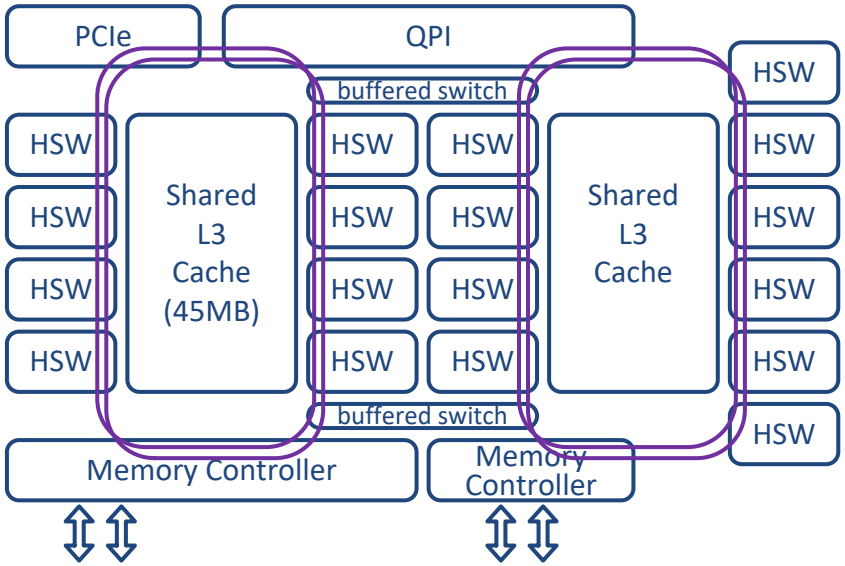
Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

Source:
Vishwani Agrawal

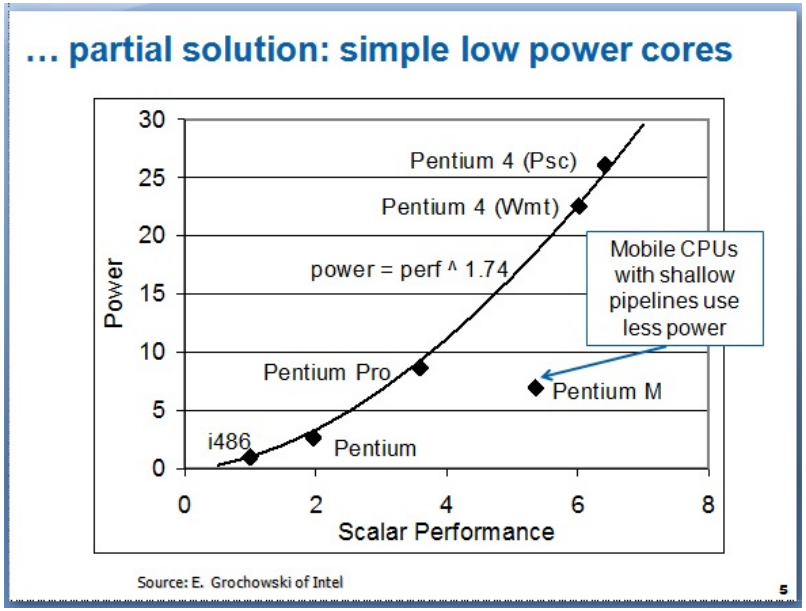
... Many core: we are all doing it



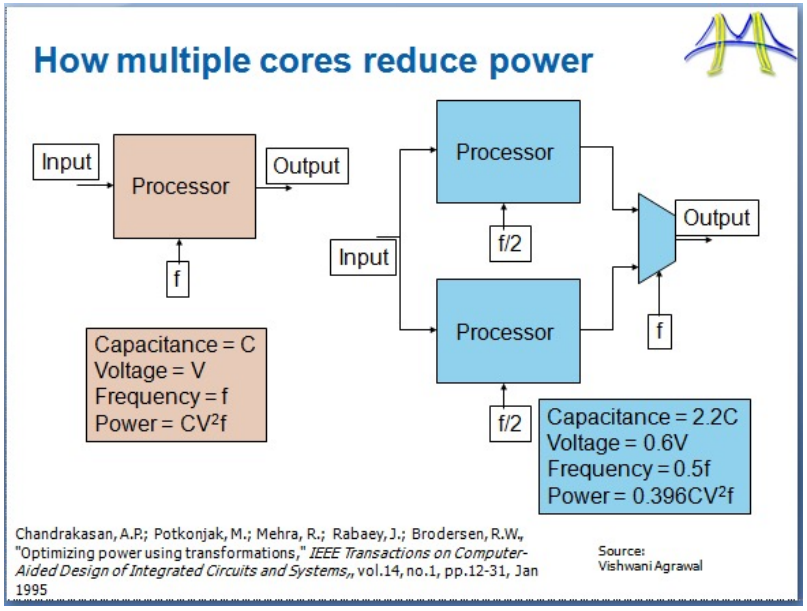




The result…



+



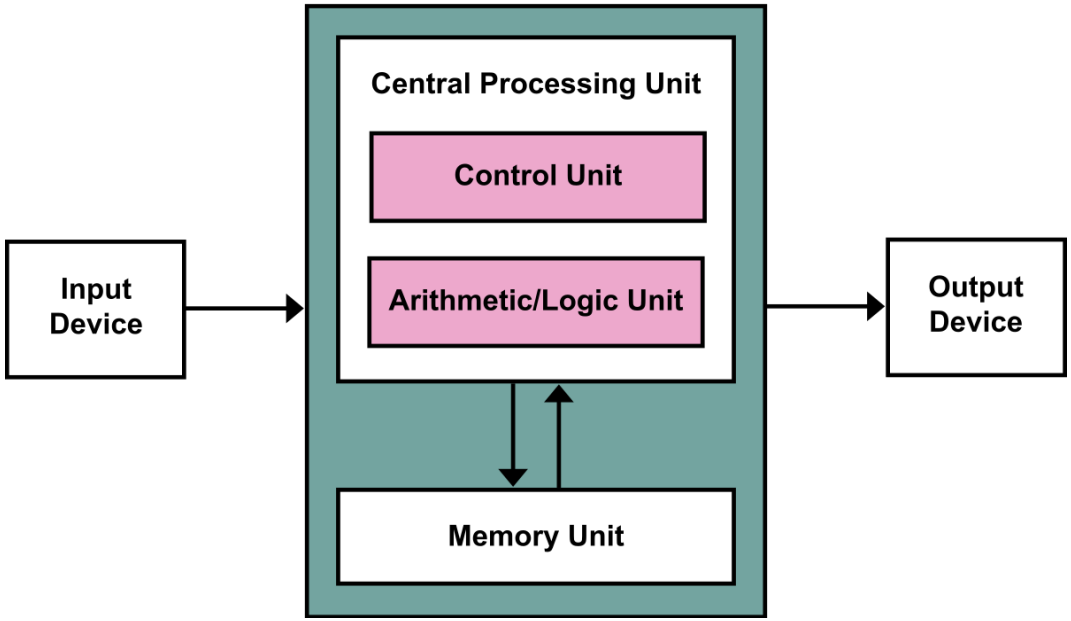
=

A new HW/SW contract … HW people will do what's natural for them (lots of cores) and optimization is up to SW people who will have to adapt (rewrite everything)

Core concepts in parallel programming

Let's agree on a few definitions:

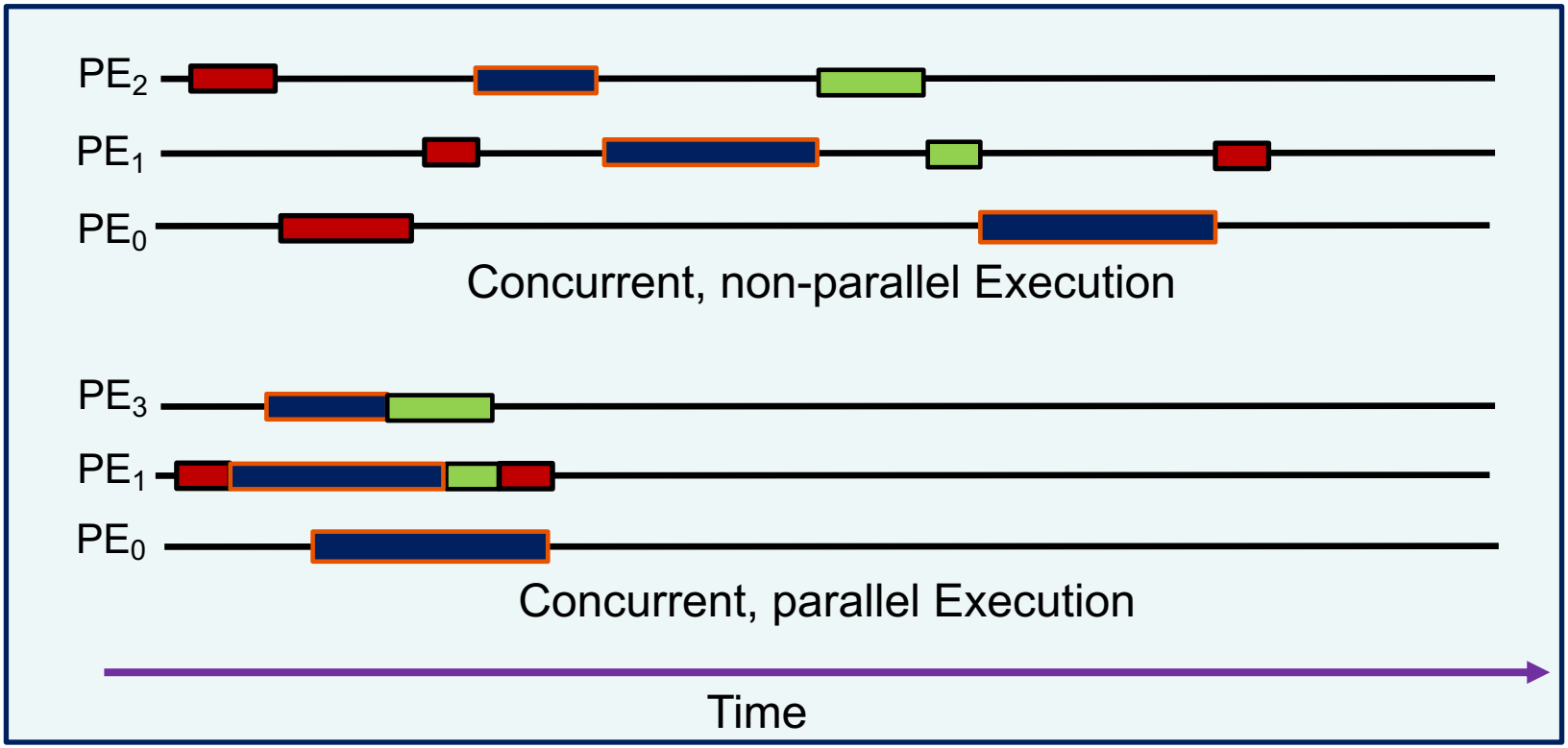
- **Computer:**
  - A machine that transforms *input data* into *output data*.
  - Typically a computer consists of Control, Arithmetic/Logic, and Memory units.
  - The transformation is defined by a stored **program** (von Neumann architecture).



- **Task:**
  - A specific sequence of instructions plus a data environment. A program is composed of one or more tasks.
- **Active task:**
  - A task that is available to be scheduled for execution. When the task is moving through its sequence of instructions, we say it is making **forward progress**
- **Fair scheduling:**
  - When a scheduler gives each active task an equal *opportunity* for execution.

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.

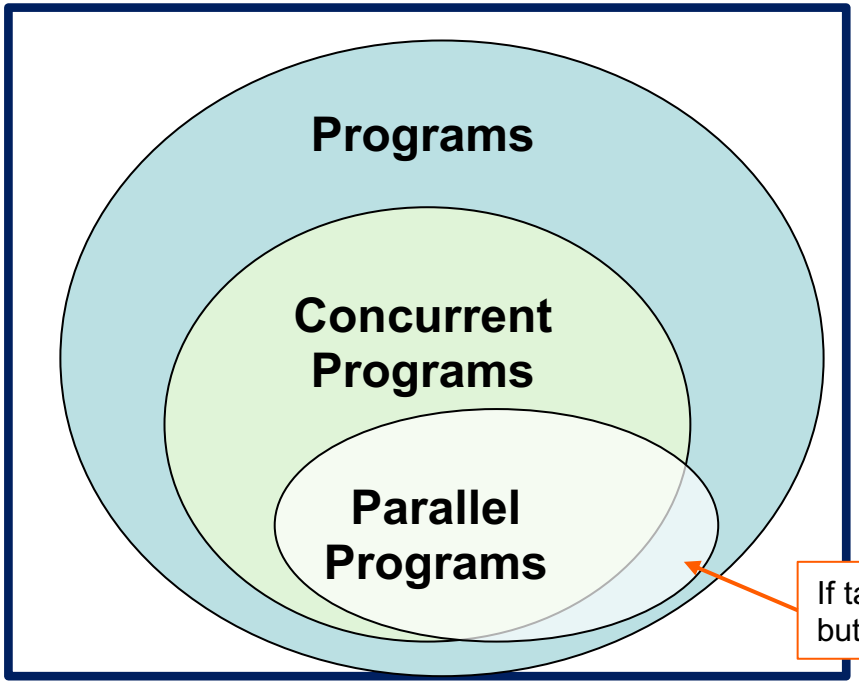


PE = Processing Element

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at one time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at one time.



In most cases, parallel programs exploit concurrency in a problem to run tasks on multiple processing elements

We use Parallelism to:
- Do more work in less time
- Work with larger problems

If tasks execute in "lock step" they are not concurrent, but they are still parallel. Example … a SIMD unit.

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Consider performance of parallel programs

Compute N independent tasks on one processor

Load Data | Compute T_1 | … | Compute T_N | Consume Results

$$Time_{seq}(1) = T_{load} + N*T_{task} + T_{consume}$$

Compute N independent tasks with P processors

Load Data

Compute T_1

…

Compute T_N

Consume Results

Ideally Cut runtime by ~1/P

(Note: Parallelism only speeds-up the concurrent part)

$$Time_{par}(P) = T_{load} + (N/P)*T_{task} + T_{consume}$$

Talking about performance

- Speedup: the increased performance from running on P processors.

$$S(P) = \frac{Time_{seq}(1)}{Time_{par}(P)}$$

- Perfect Linear Speedup: happens when no parallel overhead and algorithm is 100% parallel.

$$S(P) = P$$

- Super-linear Speedup: typically due to cache effects … i.e. as P grows, aggregate cache size grows so more of the problem fits in cache

$$S(P) > P$$

So now you should understand my silly introduction slide.

Introduction

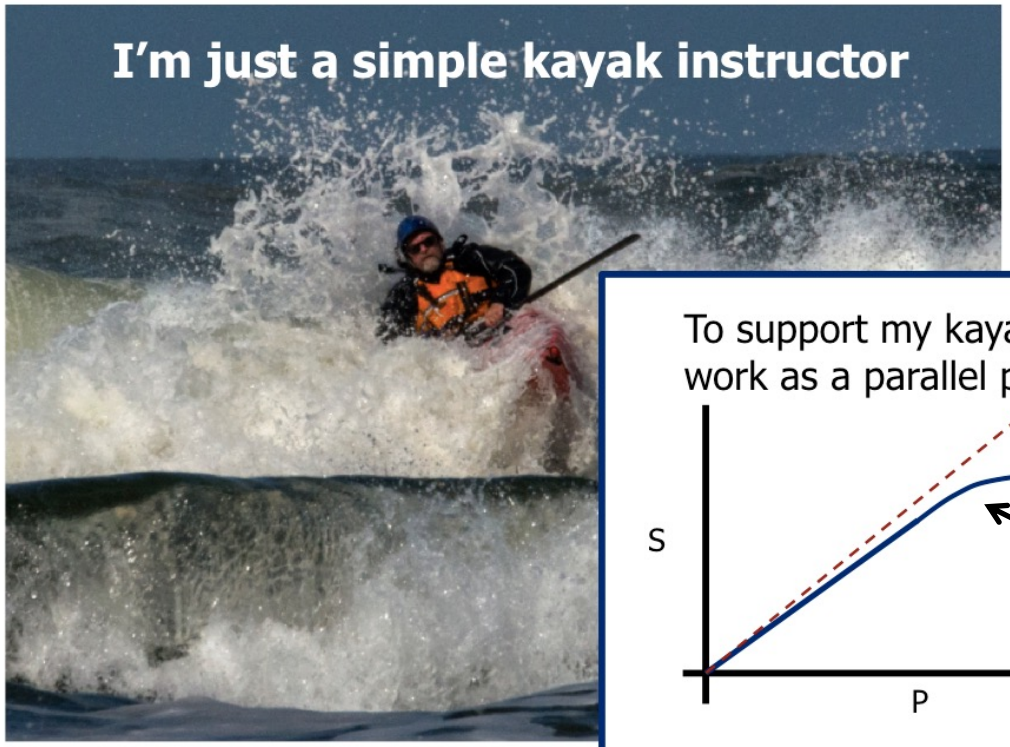


Photo © by Greg Clopton, 2014

To support my kayaking habit I work as a parallel programmer

Which means I know how to turn math into lines on a speedup plot

4

We measure our success as parallel programmers by how close we come to ideal linear speedup.

A good parallel programmer always figures out when you fall off the linear speedup curve and why that has occurred.

Amdahl's Law

- What is the maximum speedup you can expect from a parallel program?
- Approximate the runtime as a part that can be sped up with additional processors and a part that is fundamentally serial.

$$Time_{par}(P) = (serial\_fraction + \frac{parallel\_fraction}{P}) * Time_{seq}$$

- If serial_fraction is α and parallel_fraction is (1- α) then the speedup is:

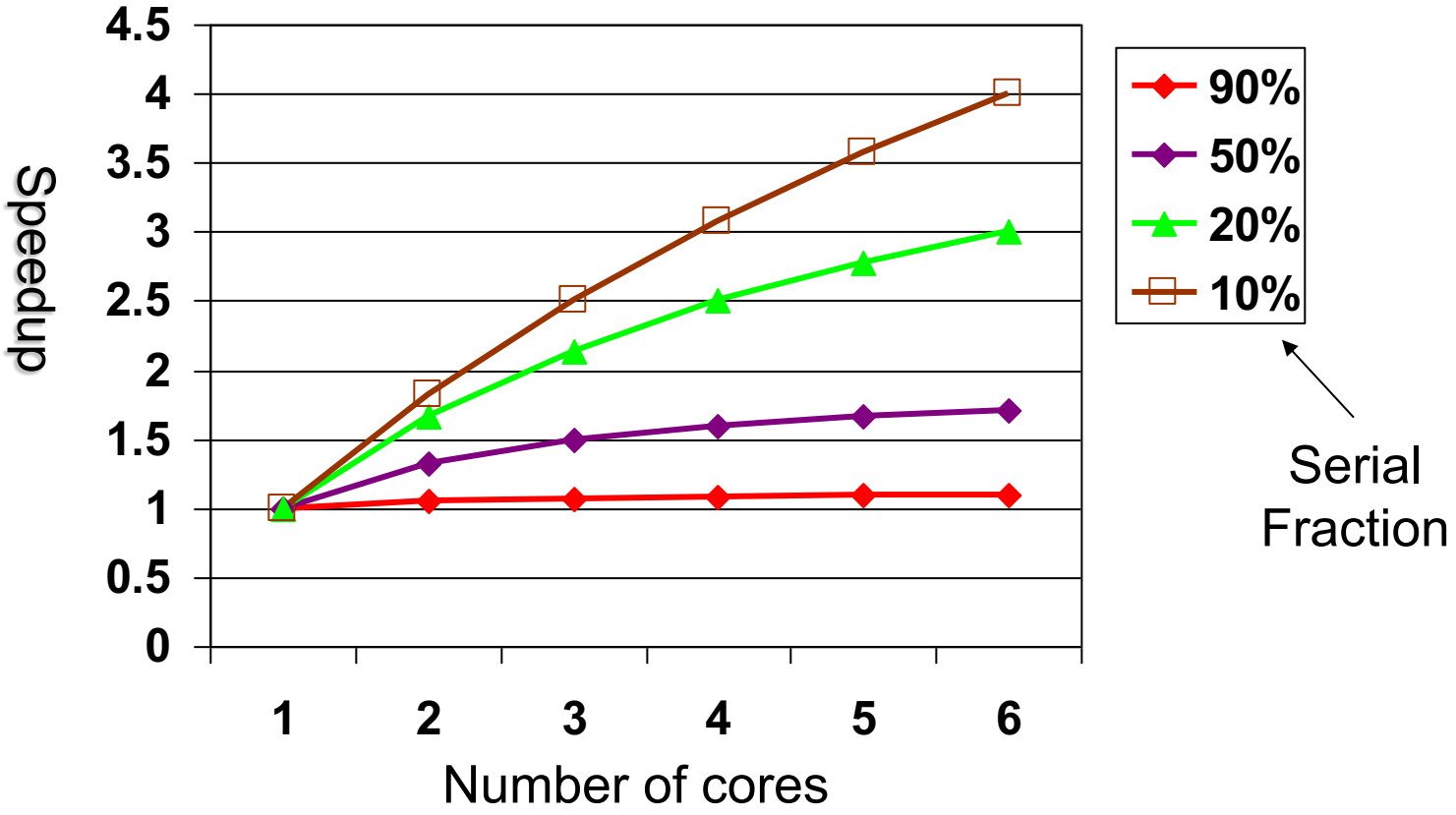
$$S(P) = \frac{Time_{seq}}{Time_{par}(P)} = \frac{Time_{seq}}{(\alpha + \frac{1-\alpha}{P}) * Time_{seq}} = \frac{1}{\alpha + \frac{1-\alpha}{P}}$$

- If you had an unlimited number of processors: $P \rightarrow \infty$

- The maximum possible speedup is: $S = \frac{1}{\alpha}$ ← Amdahl's Law

Implications of Amdahl's Law

- Consider benefits of adding processors to your parallel program for different serial fractions.
- Note: getting a serial fraction under 10% is challenging for the typical application



Acknowledgement: , 2009 DAC Tutorial – Tom Spyrou

Weak Scaling: a response to Amdhal

- **Gustafson's Observation:** For many problems, as the size of the problem (N) grows, the serial fraction ($\alpha(N)$) decreases. What does this imply for the speedup (S(P,N))?

$$S(P,N) = \frac{T_{seq}(1)}{(\alpha(N) + \frac{1-\alpha(N)}{P}) * T_{seq}(1)}$$

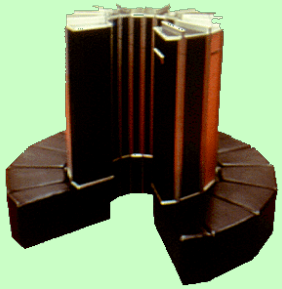
$$\lim_{N \to N_{large}} \alpha(N) = 0$$

$$S(P, N_{large}) \to P$$

- In other words … if parallelizable computations asymptotically dominate the runtime, then solving a larger problem will increase your Amdahl-limited speedup.
- **Weak Scaling:** Performance of an application when the problem size increases with the number of processors (fixed size problem per node)

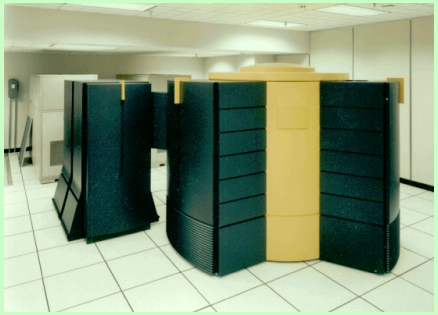
History of parallel computing systems

Parallel computing: It's old



Cray 1 (1976)

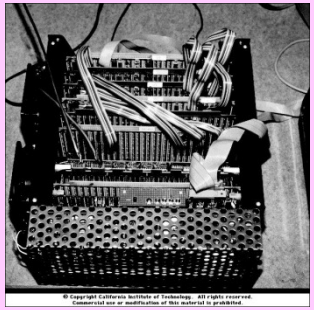
Cray 2 (1985)



Cray C-90 (1991)

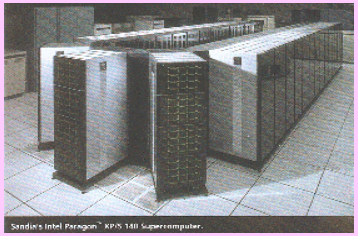
Vector Computers

SMP computers

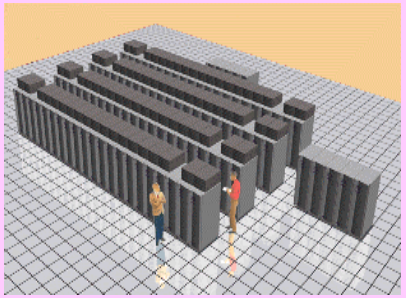


Cosmic cube (1983)

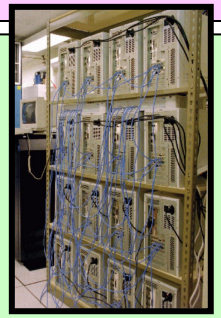
Massively Parallel Processors (MPP)



Paragon (1993)



ASCI Red (1997)



Clusters (late 80's)

Cluster Computers

Linux PC Clusters
(~1995)

Late 70's

Late 80's

Late 90's

The birth of Supercomputing

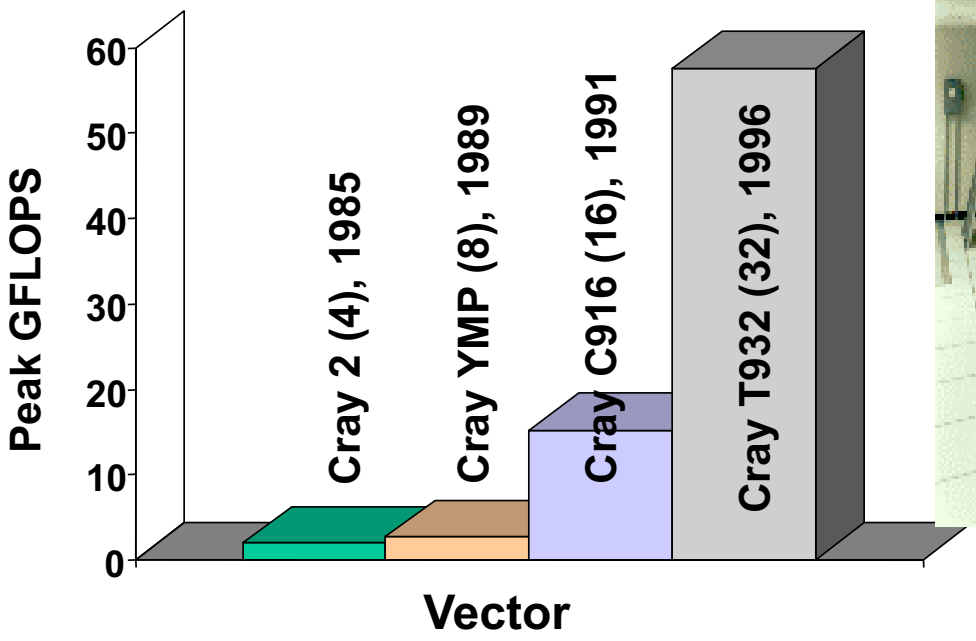
- On July 11, 1977, the CRAY-1A, serial number 3, was delivered to NCAR. The system cost was $8.86 million ($7.9 million plus $1 million for the disks).

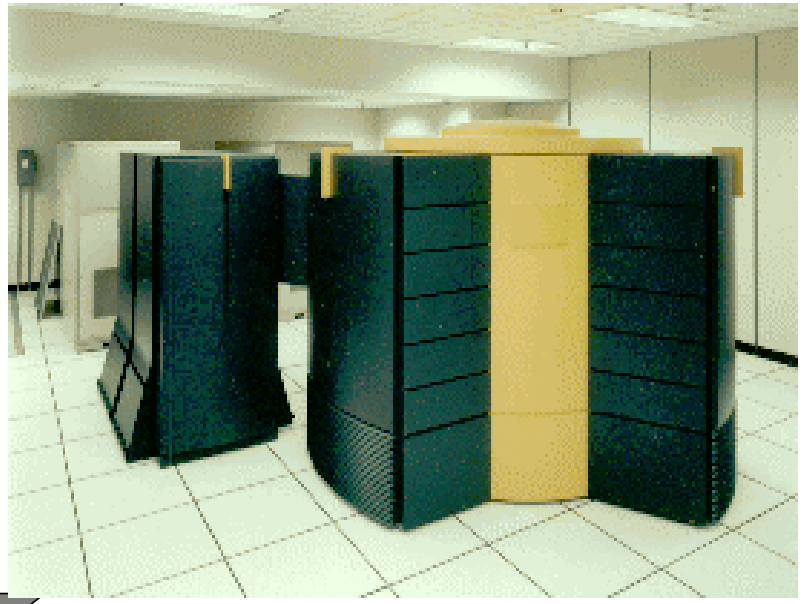
- The CRAY-1A:
  - 2.5-nanosecond clock,
  - 64 vector registers,
  - 1 million 64-bit words of high-speed memory.
  - Peak speed:
    - 80 MFLOPS scalar.
    - 250 MFLOPS vector (but this was VERY hard to achieve)
- Cray software … by 1978
  - Cray Operating System (COS),
  - the first automatically vectorizing Fortran compiler (CFT),
  - Cray Assembler Language (CAL) were introduced.

http://www.cisl.ucar.edu/computers/gallery/cray/cray1.jsp

History of Supercomputing: The Era of the Vector Supercomputer

- Large mainframes that operated on vectors of data
- Custom built, highly specialized hardware and software
- Multiple processors in an shared memory configuration
- Required modest changes to software (vectorization)





The Cray C916/512 at the Pittsburgh Supercomputer Center

The attack of the killer micros

- The **Caltech Cosmic Cube** developed by Charles Seitz and Geoffrey Fox in1981
- 64 Intel 8086/8087 processors
- 128kB of memory per processor
- 6-dimensional hypercube network

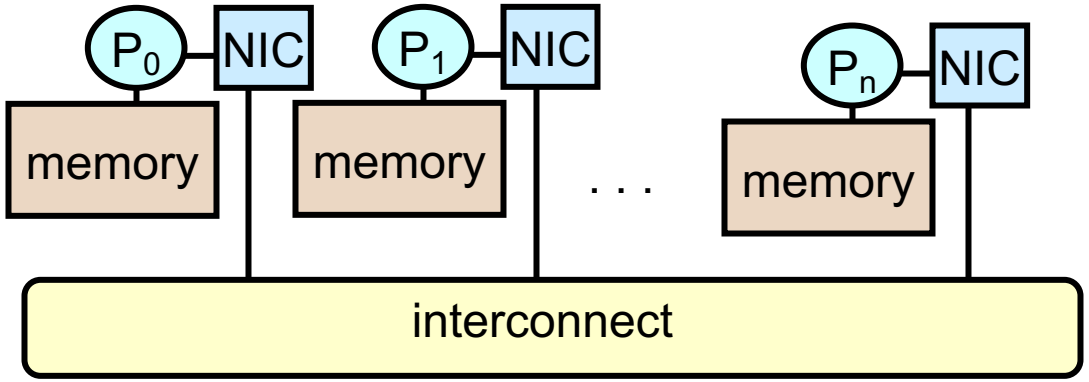
The cosmic cube, Charles Seitz
Communications of the ACM, Vol 28, number 1 January 1985, p. 22

Launched the “attack of the killer micros”
Eugene Brooks, SC’90

http://calteches.library.caltech.edu/3419/1/Cubism.pdf

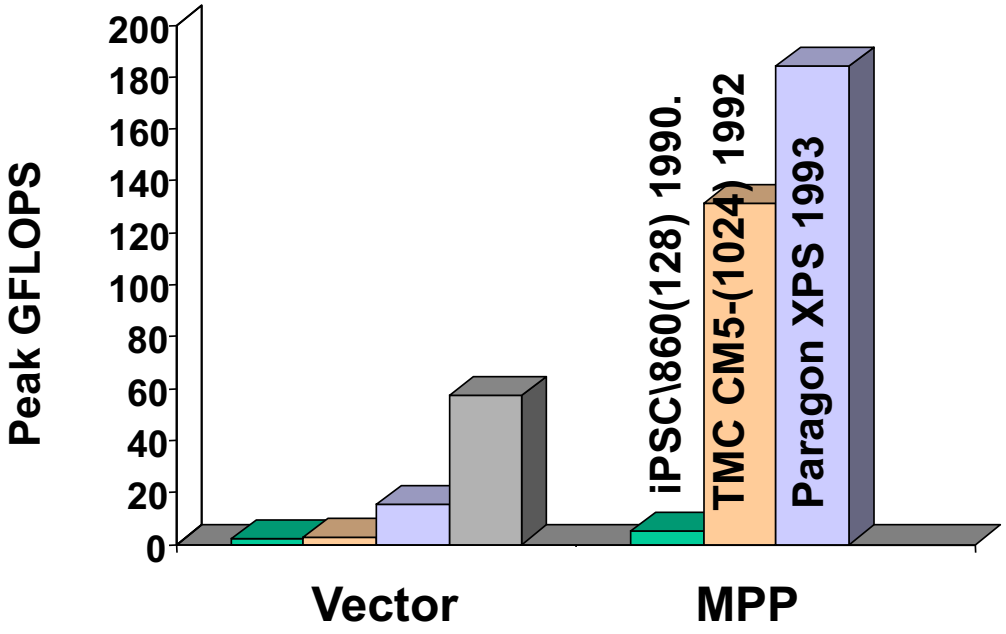
Machine Model: Distributed Memory

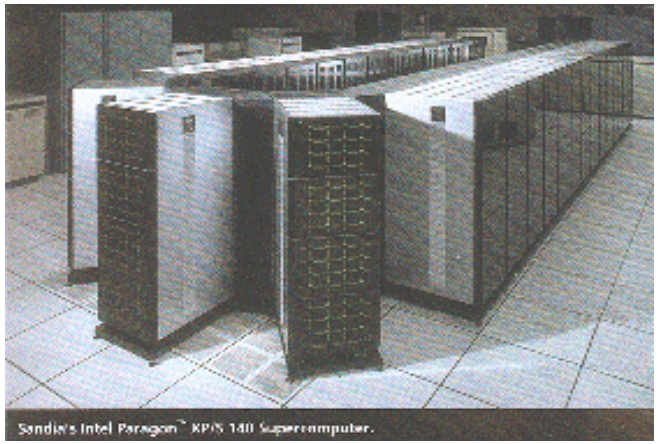
- The new microprocessor based parallel computers were all distributed memory machines.
- Each node has its own processors, memory and caches but cannot directly access another node's memory.
- Each "node" has a Network Interface component (NIC) for all communication and synchronization.
- Fundamentally more scalable than shared memory machines … especially cache coherent shared memory.



It took a while, but MPPs came to dominate supercomputing

- Parallel computers with large numbers of microprocessors
- High speed, low latency, scalable interconnection networks
- Lots of custom hardware to support scalability
- Required massive changes to software (parallelization)

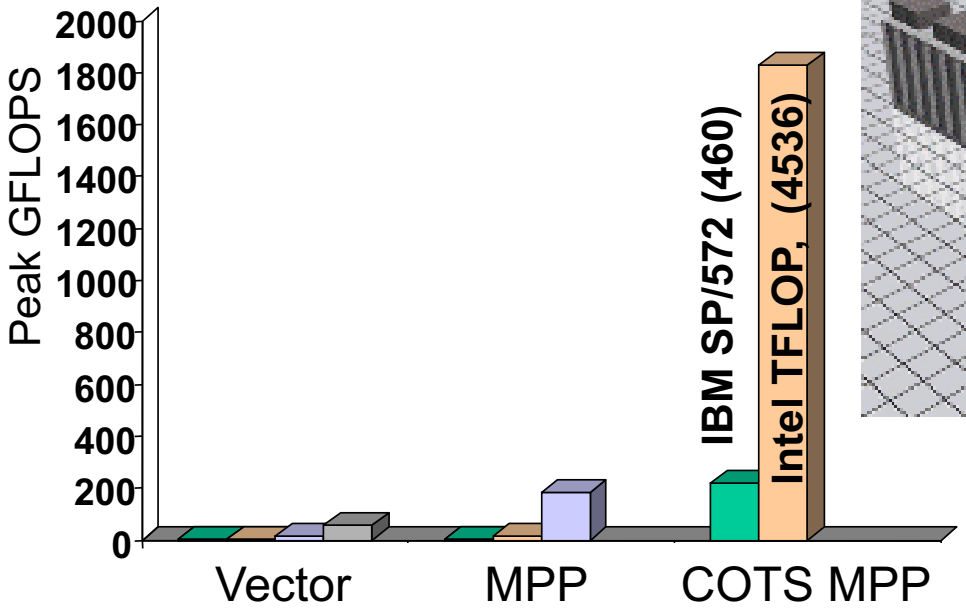


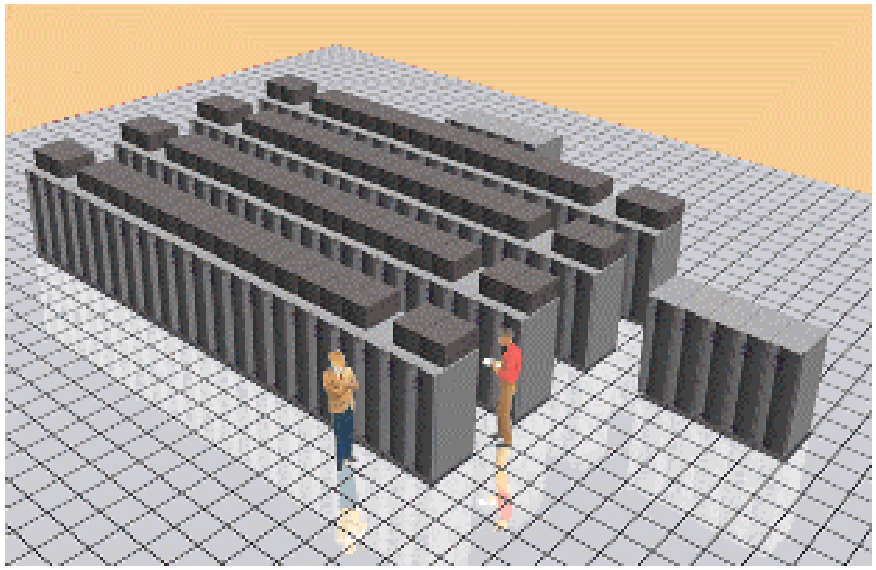


Paragon XPS-140 at Sandia National labs in Albuquerque NM

The cost advantage of mass market COTS

- MPPs using Mass market Commercial off the shelf (COTS) microprocessors and standard memory and I/O components
- Decreased hardware and software costs makes huge systems affordable





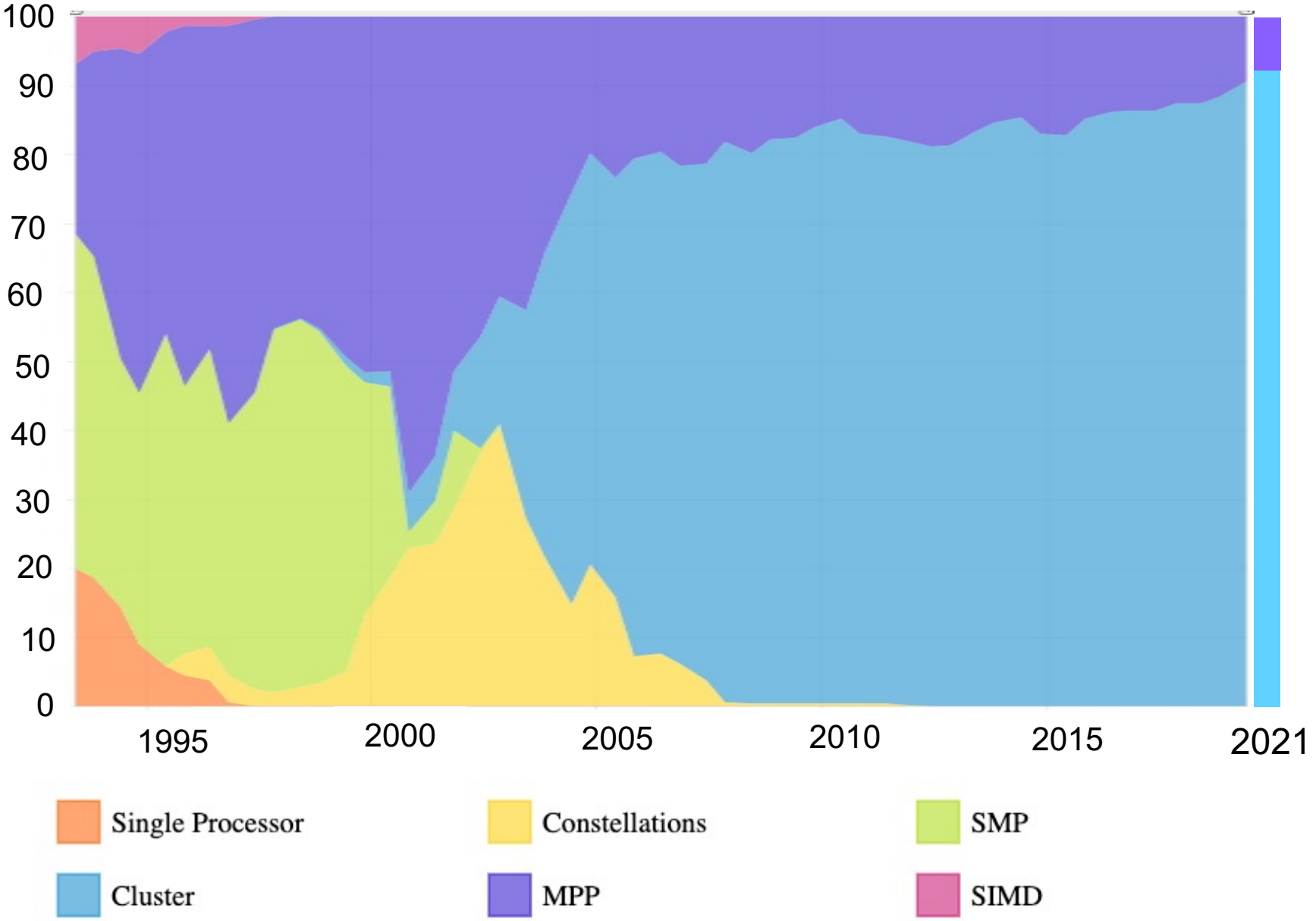
ASCI Red TFLOP Supercomputer

The MPP future looked bright … but then clusters took over

- A cluster is a collection of connected, independent computers that work in unison to solve a problem.
- Nothing is custom … motivated users could build cluster on their own

- First clusters appeared in the late 80's
- The Intel Pentium Pro in 1995 coupled with Linux made them competitive.
  - NASA Goddard's Beowulf cluster demonstrated publically that high visibility science could be done on clusters.
- Clusters made it easier to bring the benefits due to Moores's law into working supercomputers

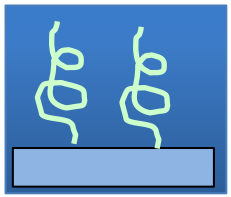
Top 500 list: System Architecture



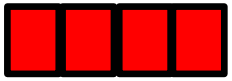
Constellation: A cluster where the number of cores on a node is greater than the number of nodes … a term only used by top500.

Clusters rule but what about parallelism on each node of the cluster?

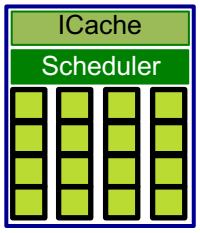
Hardware Diversity: Basic Building Blocks



CPU Core: one or more hardware threads sharing an address space. Optimized for low latencies.

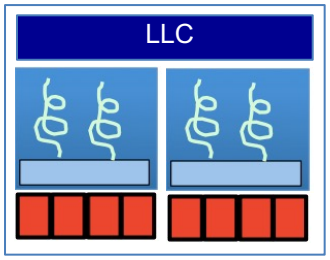


SIMD: Single Instruction Multiple Data.
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.

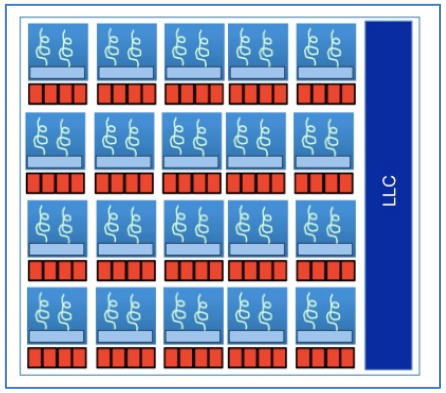


SIMT: Single Instruction Multiple Threads.
A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

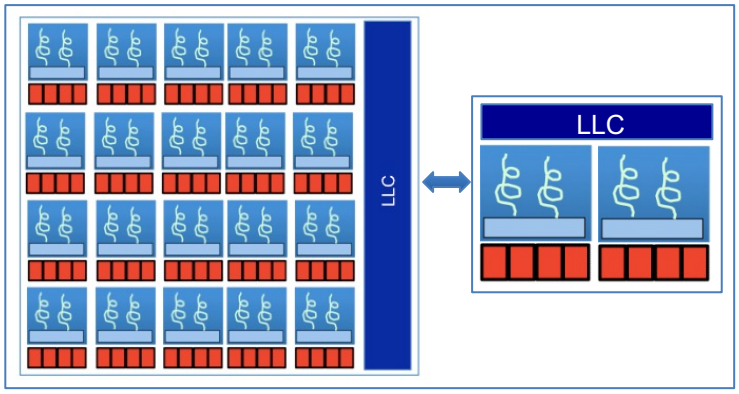
Hardware Diversity: Combining building blocks to construct nodes



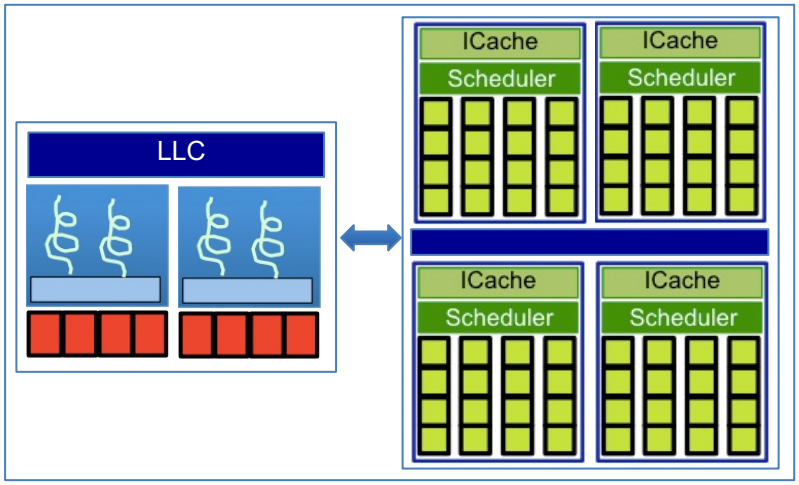
Multicore CPU



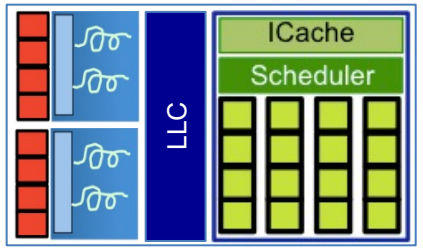
Manycore CPU



Heterogeneous:
CPU + manycore coprocessor

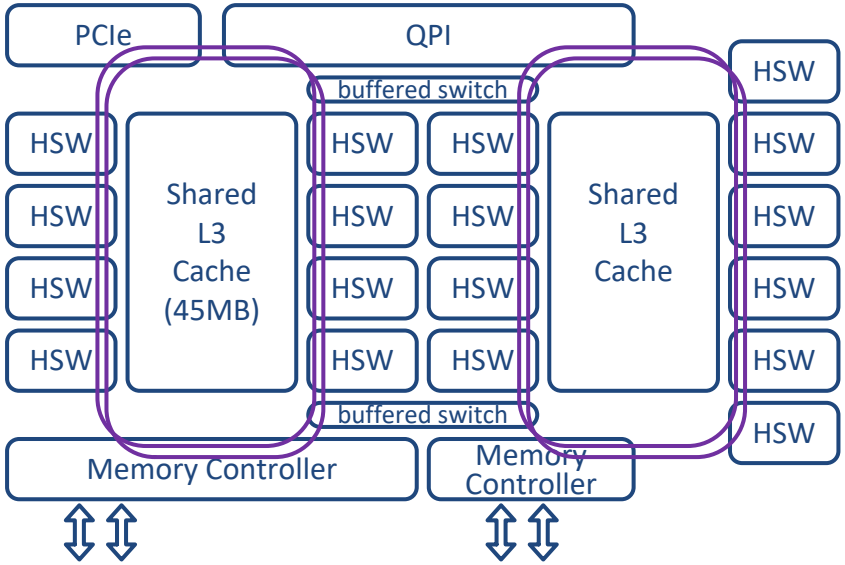


Heterogeneous: CPU+GPU



Heterogeneous:
Integrated CPU+GPU

Hardware diversity: CPUs



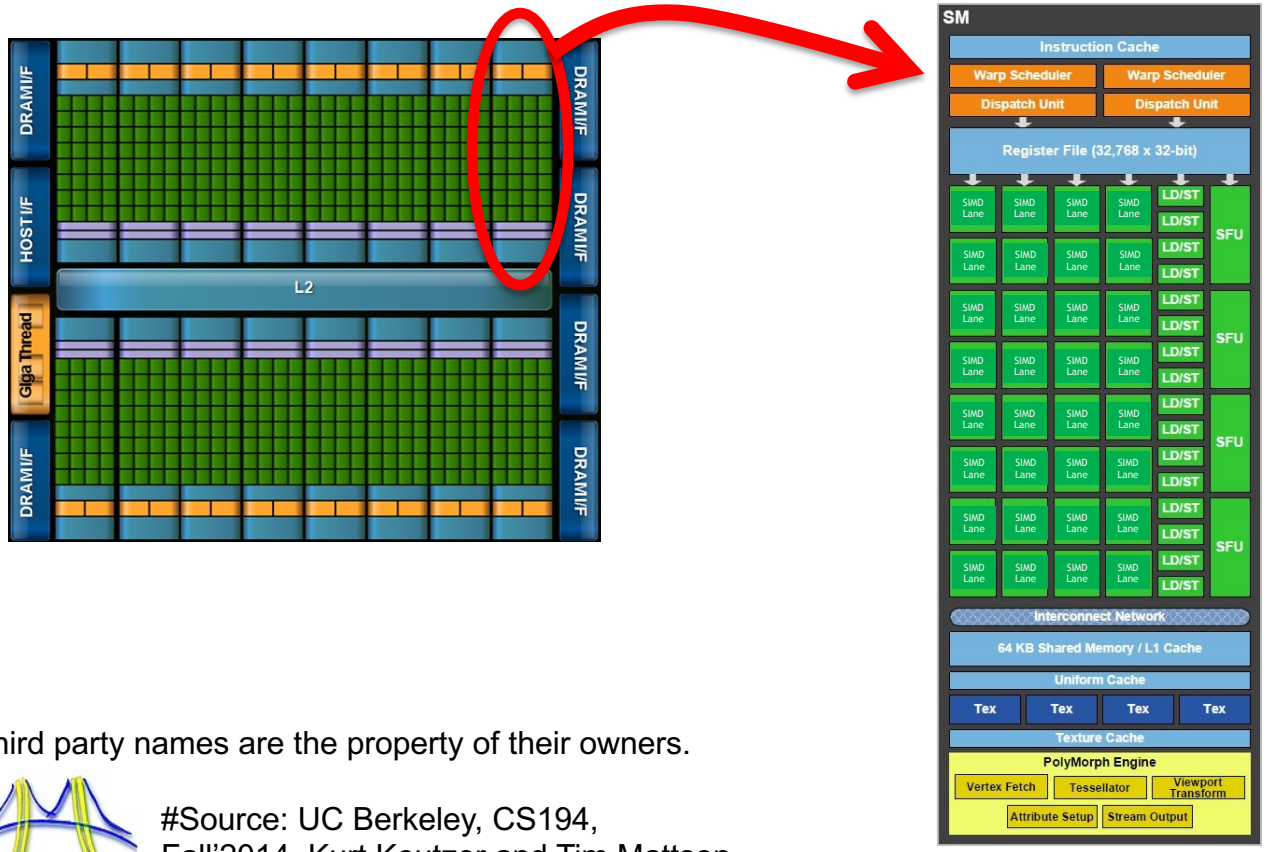
Intel® Xeon® processor: multicore E7 v3 series (Haswell or HSW)

- 18 cores
- 36 Hardware threads
- 256 bit wide vector units

In both cases … Cache hierarchy to create a low latency, coherent view of a shared address space.

Hardware diversity: GPUs

- Nvidia® GPUs are a collection of “Streaming Multiprocessors” (SM)
  - Each SM is analogous to a core of a Multi-Core CPU
- Each SM is a collection of SIMD execution pipelines that share control logic, register file, and L1 Cache#

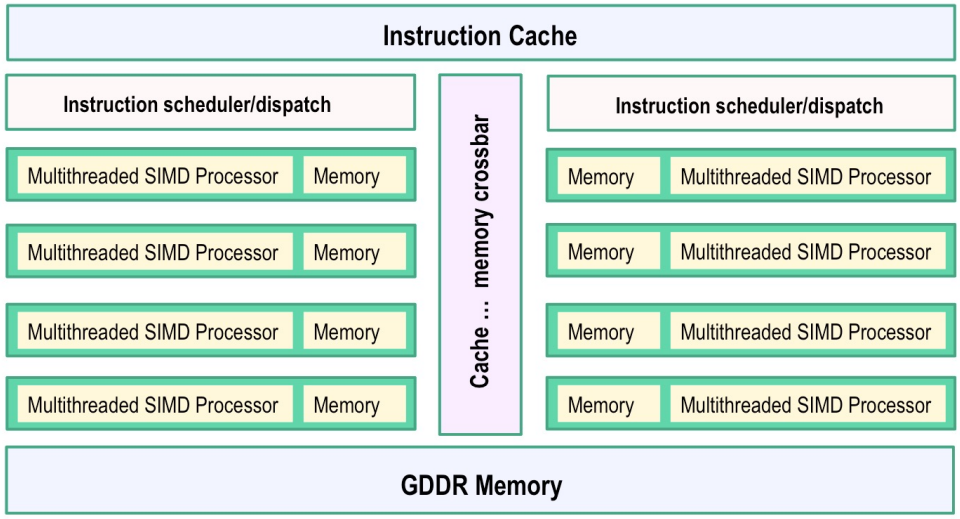


For example: an NVIDIA Tesla C2050 (Fermi) GPU with 3GB of memory and 14 streaming multiprocessor cores*.

#Source: UC Berkeley, CS194, Fall’2014, Kurt Keutzer and Tim Mattson

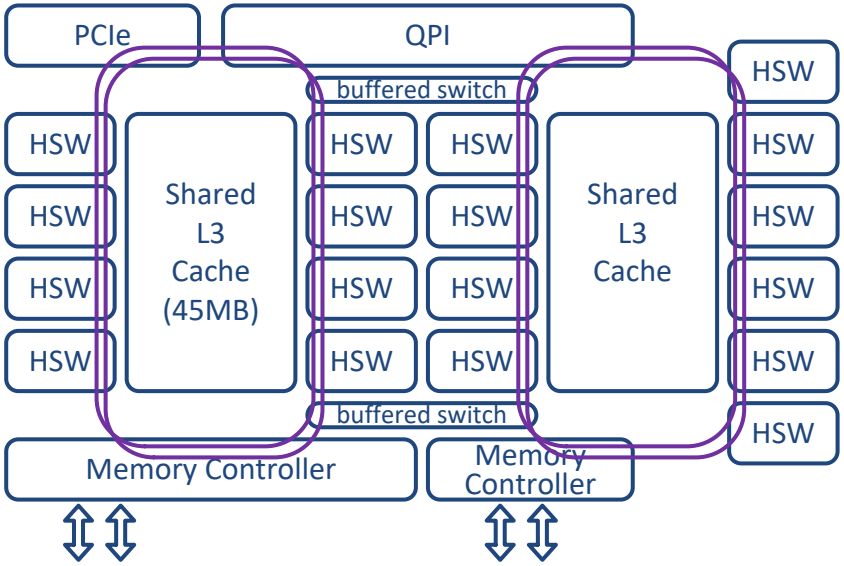
*Source: http://www.nersc.gov/users/computational-systems/dirac/node-and-gpu-configuration/

... Many core: we are all doing it



GPU

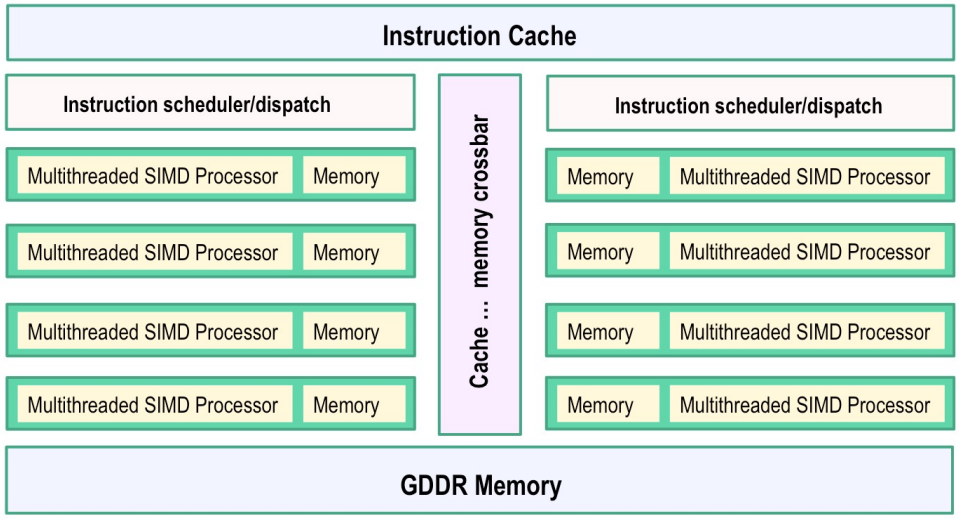
- Hundreds of Cores Driven by a single stream of instruction
- Each core has many (32, 64, …) SIMD lanes
- Optimized for throughput … oversubscription to hide memory latency



CPU

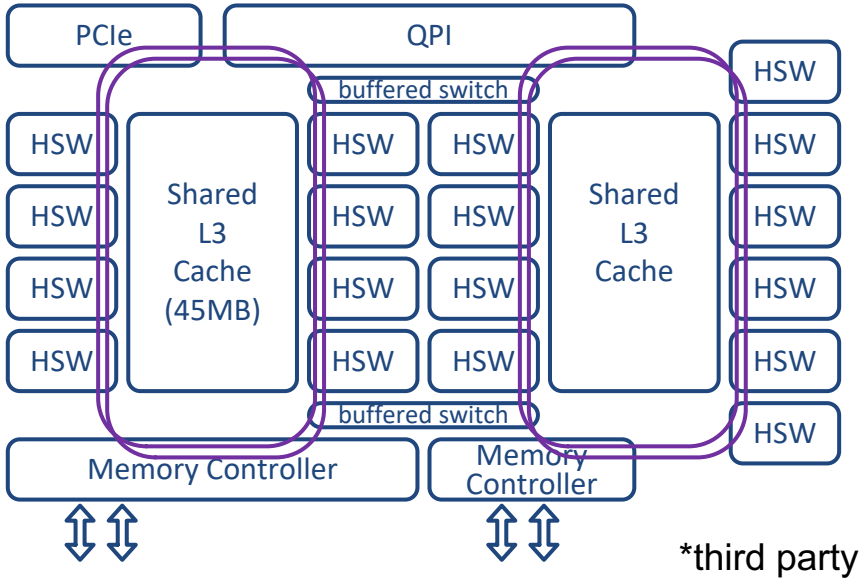
- A few to dozens of Cores with independent streams of instructions.
- Cores typically have complex logic (e.g. out of order) to make individual threads run fast.
- Optimized for latency … complex memory hierarchy to run fast out of cache

It's really about competing software platforms



GPU

- **Single Instruction multiple threads.**
  - turn loop bodies into kernels.
  - HW intelligently schedules kernels to hide latencies.
- *Dogma*: a natural way to express huge amounts of data parallelism
- Examples: CUDA, OpenCL, OpenACC



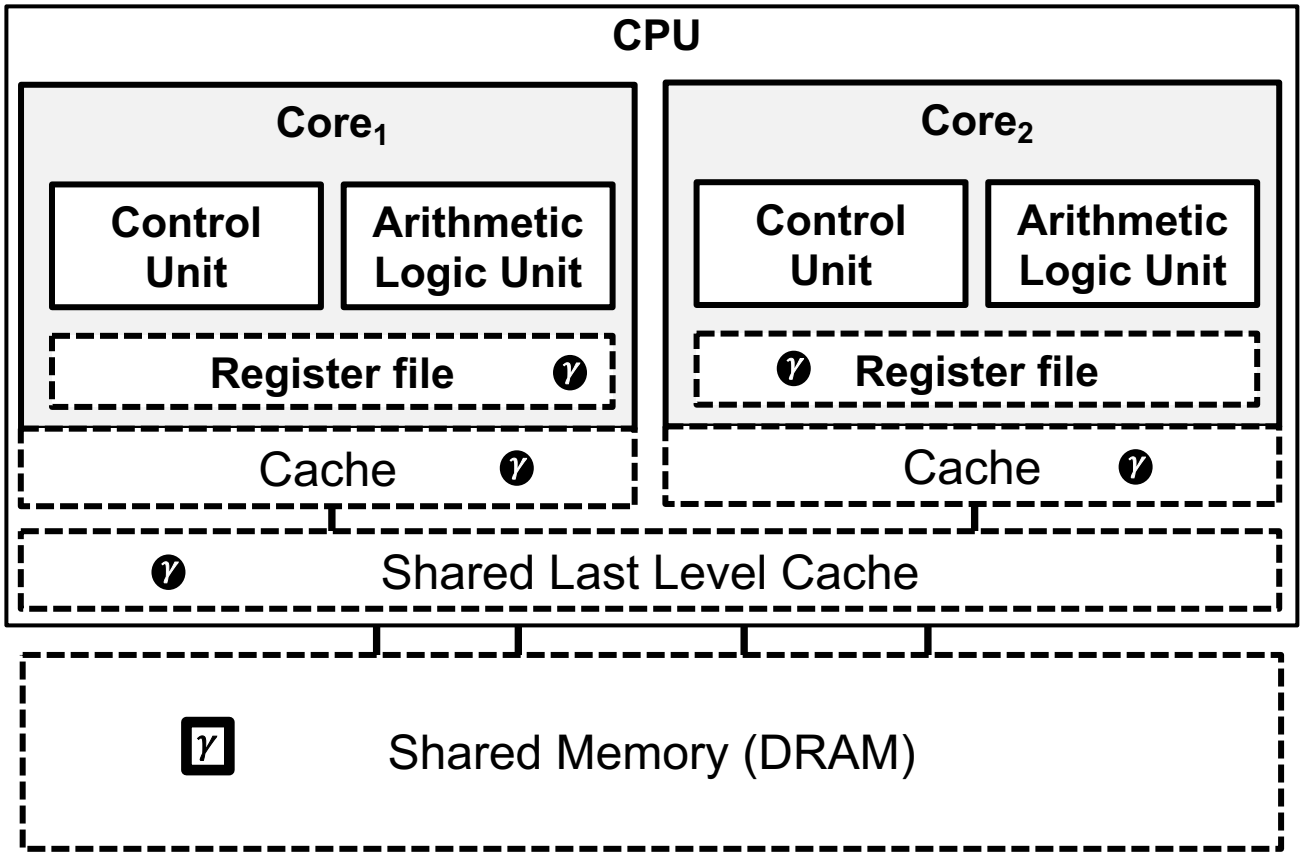
CPU

- **Shared Address space, multi-threading.**
  - Many threads executing with coherent shared memory.
- *Dogma*: The legacy programming model people already knows. Easier than alternatives.
- Examples: OpenMP, Pthreads, C++11

*third party names are the property of their owners

Back-up

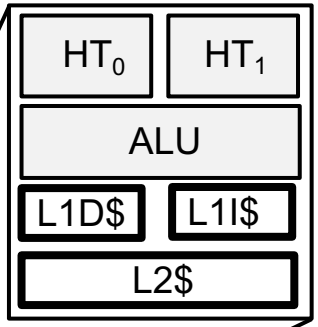
- Memory Hierarchy
- Vector instructions
- Computer networks
- Design Patterns



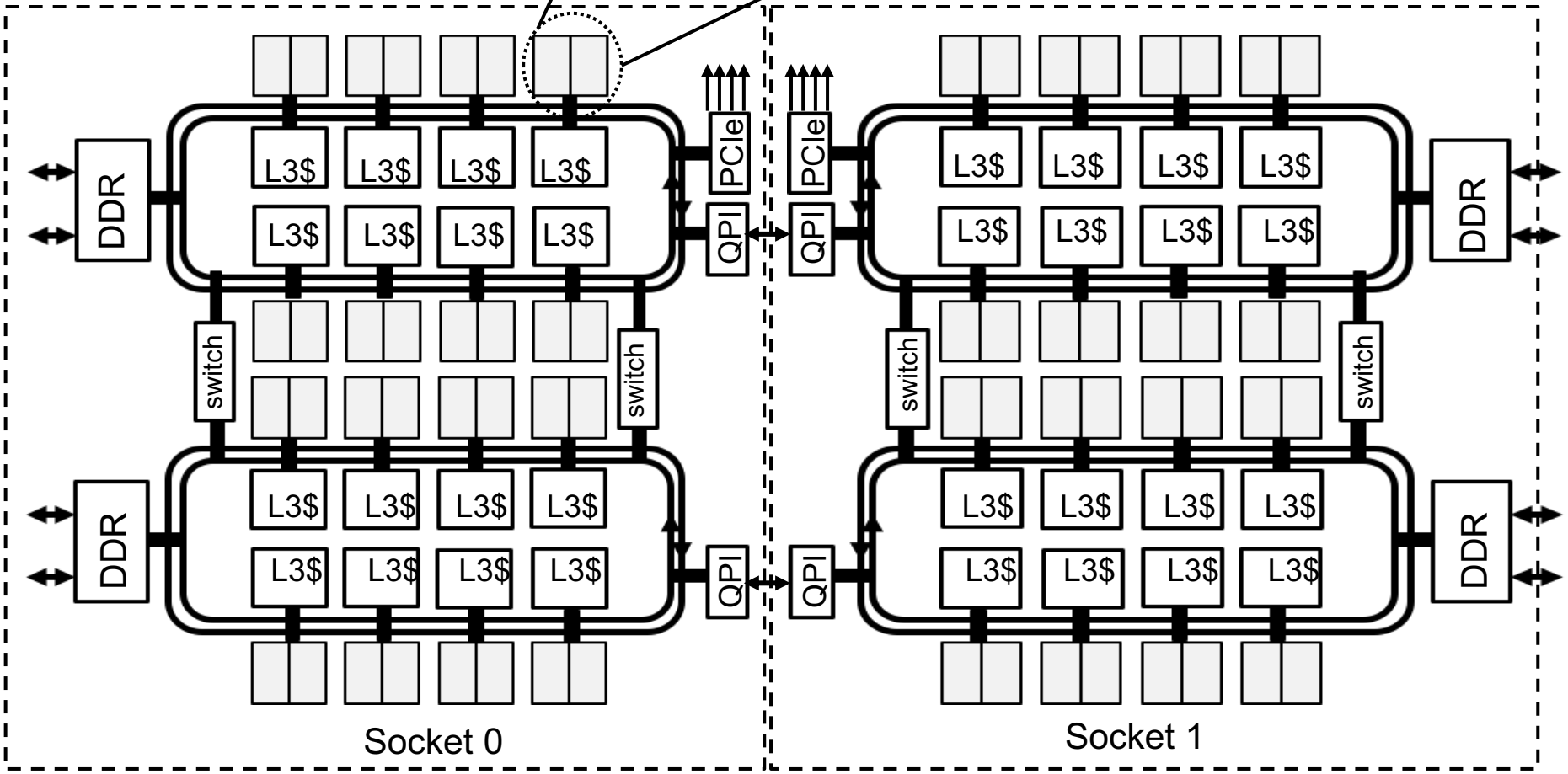
A variable is a name for a location in memory (DRAM).

Temporary copies of that variable may exist across the memory hierarchy … and these temporary values may be different.

A NUMA node



A node in Cori … A CPU based supercomputer at NERSC based on two sockets of Xeon processors with 32 cores per socket

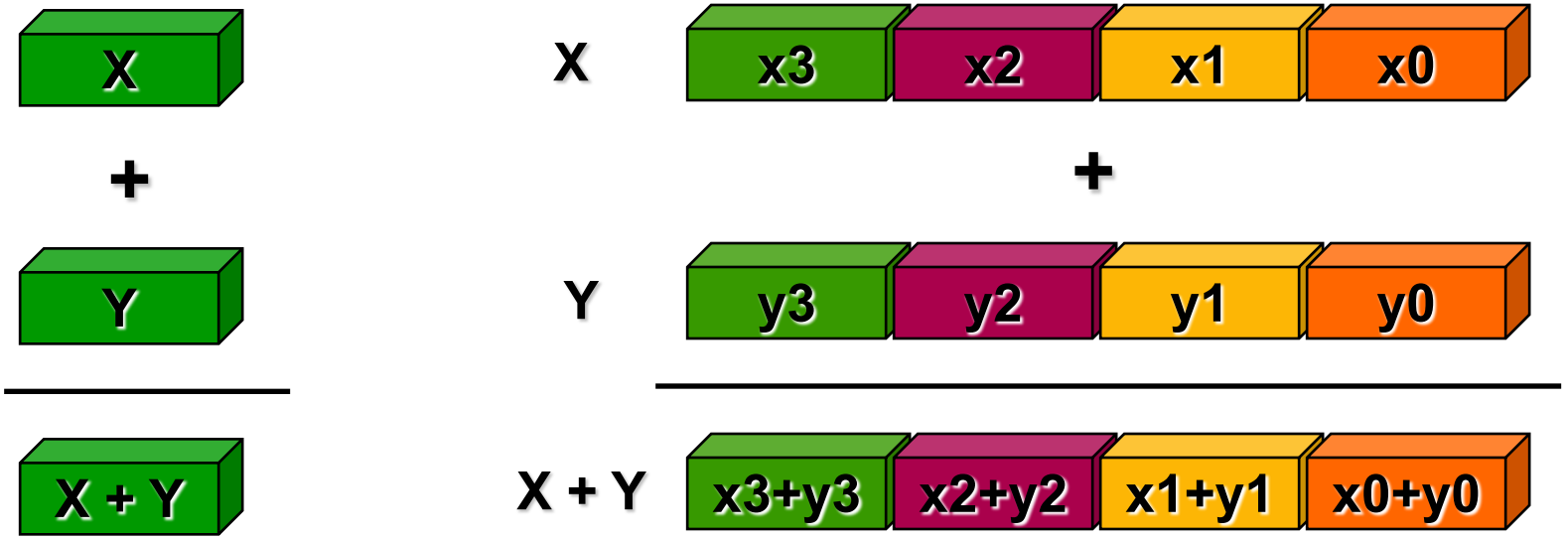


Back-up

- Memory Hierarchy
- Vector instructions
- Computer networks
- Design Patterns

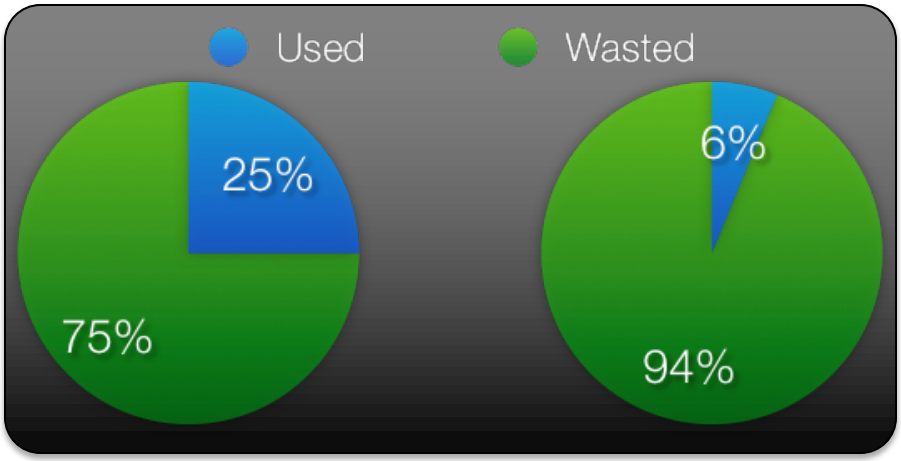
Vector SIMD

- A functional unit typically associated with a CPU core takes a single stream of instructions that are applied in parallel to the elements of values in special vector registers.
- Vector instructions usually generated by the compiler “automatically” from loops (often with help from programmer inserted directives).
- Best performance may require explicit coding with vector intrinsics.



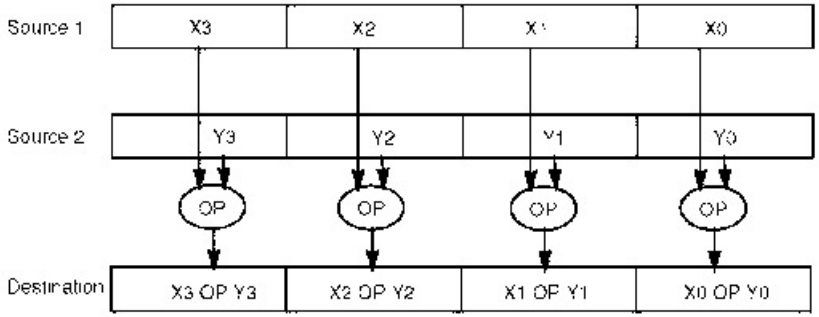
Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

Vector (SIMD) Programming



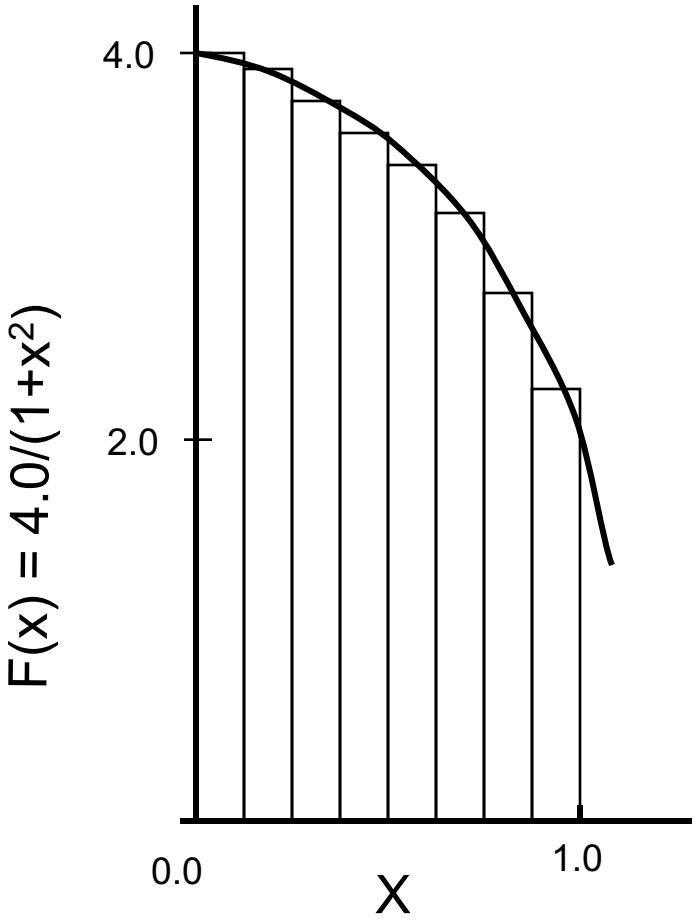
4 way SIMD (SSE)

16 way SIMD (Xeon™ PHI)



- Architects love vector units, since they permit space- and energy- efficient parallel implementations.
- However, standard SIMD instructions on CPUs are inflexible, and can be difficult to use.
- Options:
  - Let the compiler do the job
  - Assist the compiler with language level constructs for explicit vectoriztion.
  - Use intrinsics … an assembly level approach.

Example Problem: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

Compile as (O3 no-vec), 0.012 secs

```
static long num_steps = 8388608;
float step;
int main ()
{           int i;      float x, pi, sum = 0.0;

            step = 1.0/(float) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Normally, I'd use double types throughout to minimize roundoff errors especially on the accumulation into sum. But to maximize impact of vectorization for these exercise, we'll use float types.

Explicit Vectorization PI program

Compile as (O3 no-vec), 0.012 secs
Compile as (O3), 0.012 secs

```
static long num_steps = 8388608;
float step;
int main ()
{       int i;      float x, pi, sum = 0.0;

        step = 1.0/(float) num_steps;
        #pragma omp simd reduction(+:sum)
        for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

No vectorization benefit . Why?

Note that literals (such as 4.0, 1.0 and 0.5) are not explicitly declared with the desired type. The C language treats these as “double” and that impacts compiler optimizations. We call this the default case.

Explicit Vectorization PI program

Literals as double (O3 no-vec), 0.012 secs
Literals as Float (O3 no-vec), 0.0042 secs
Literals as Float (O3), 0.0024 secs

```
static long num_steps = 8388608;
float step;
int main ()
{           int i;      float x, pi, sum = 0.0;

            step = 1.0f/(float) num_steps;
            #pragma omp simd reduction(+:sum)
            for (i=0;i< num_steps; i++){
                        x = (i+0.5f)*step;
                        sum = sum + 4.0f/(1.0f+x*x);
            }
            pi = step * sum;
}
```

Note that literals (such as 4.0, 1.0 and 0.5) are explicitly declared as type float (to match the types of the variables in this code. This greatly enhances vectorization and compiler optimization.

Pi Program: Vectorization with intriniscs (SSE)

```
 float pi_sse(int  num_steps)
{  float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
   step = 1.0/(float) num_steps;

   __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
   __m128 one    = _mm_load1_ps(&scalar_one);
   __m128 four   = _mm_load1_ps(&scalar_four);
   __m128 vstep  = _mm_load1_ps(&step);
   __m128 sum    = _mm_load1_ps(&scalar_zero);
   __m128 xvec;   __m128 denom;  __m128 eye;

  for (int i=0;i< num_steps; i=i+4){         // unroll loop 4 times
     ival       = (float)i;                  // and assume num_steps%4 = 0
     eye        = _mm_load1_ps(&ival);
     xvec       = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
     denom  = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
     sum      = _mm_add_ps(_mm_div_ps(four,denom),sum);
   }
   _mm_store_ps(&vsum[0],sum);
   pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  return pi;
}
```

Pi Program: Vector intriniscs plus OpenMP

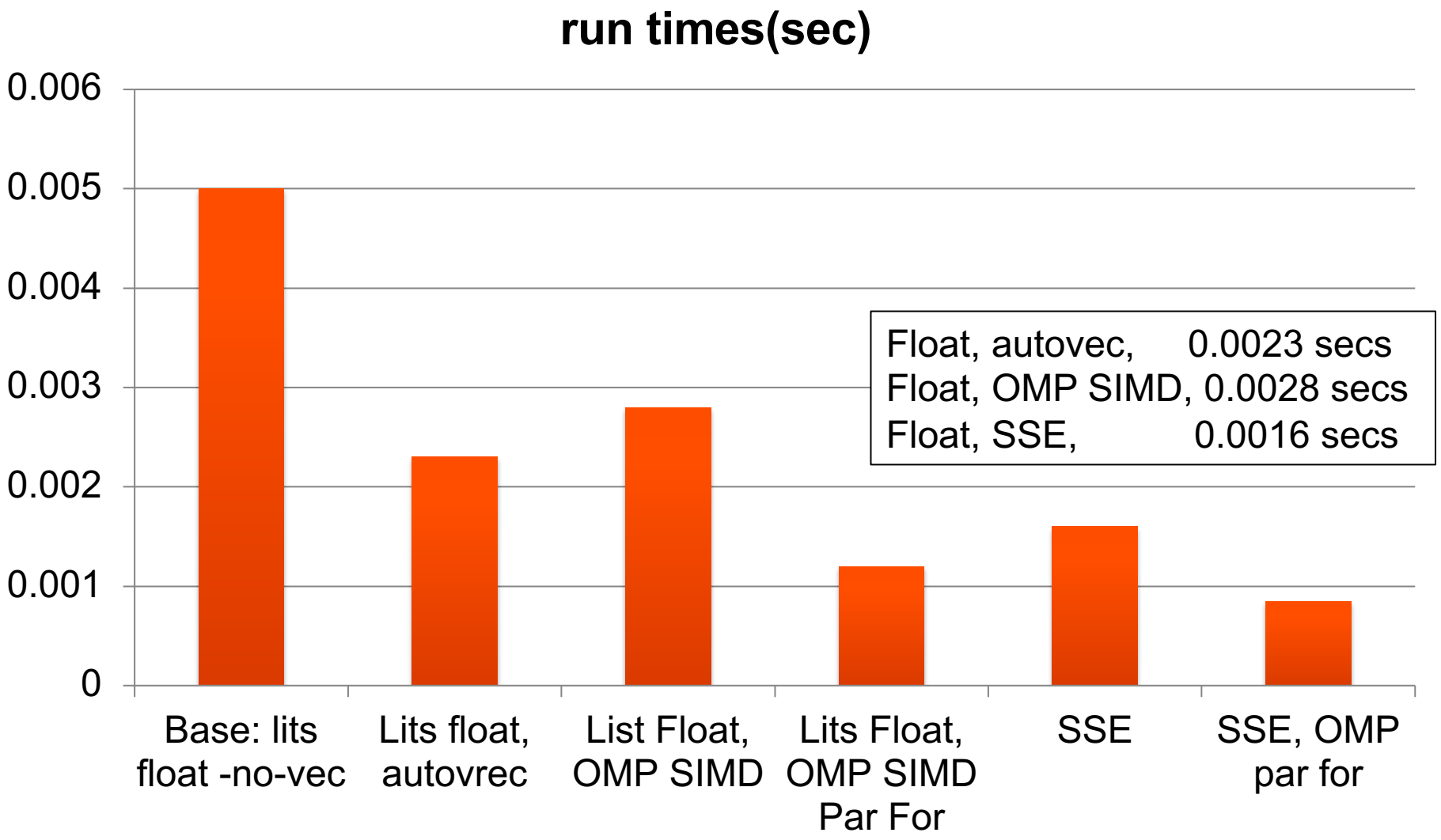
```
 float pi_sse(int  num_steps)
{  float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
   float local_sum[NTHREADS];   // set NTHREADS elsewhere, often to num of cores
   step = 1.0/(float) num_steps;  pi = 0.0;
  #pragma omp parallel
  {    int i, ID=omp_get_thread_num();
       __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
       __m128 one    = _mm_load1_ps(&scalar_one);
       __m128 four   = _mm_load1_ps(&scalar_four);
       __m128 vstep  = _mm_load1_ps(&step);
       __m128 sum    = _mm_load1_ps(&scalar_zero);
       __m128 xvec;   __m128 denom;  __m128 eye;
      #pragma omp for
      for (int i=0;i< num_steps; i=i+4){
         ival      = (float)i;
         eye       = _mm_load1_ps(&ival);
         xvec      = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
         denom     = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
         sum       = _mm_add_ps(_mm_div_ps(four,denom),sum);
      }
     _mm_store_ps(&vsum[0],sum);
     local_sum[ID] = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  }
   for(int k = 0; k<NUM_THREADS;k++) pi+=local_sum[k];
  return pi;
}
```

To parallelize with OpenMP:

1. Promote local_sum to an array to there is a variable private to each thread but available after the parallel region
2. Add parallel region and declare vector registers inside the parallel region so each thread has their own copy.
3. Add workshop loop (for) construct
4. Add local sums after the parallel region to create the final value for pi

PI program Results: The details

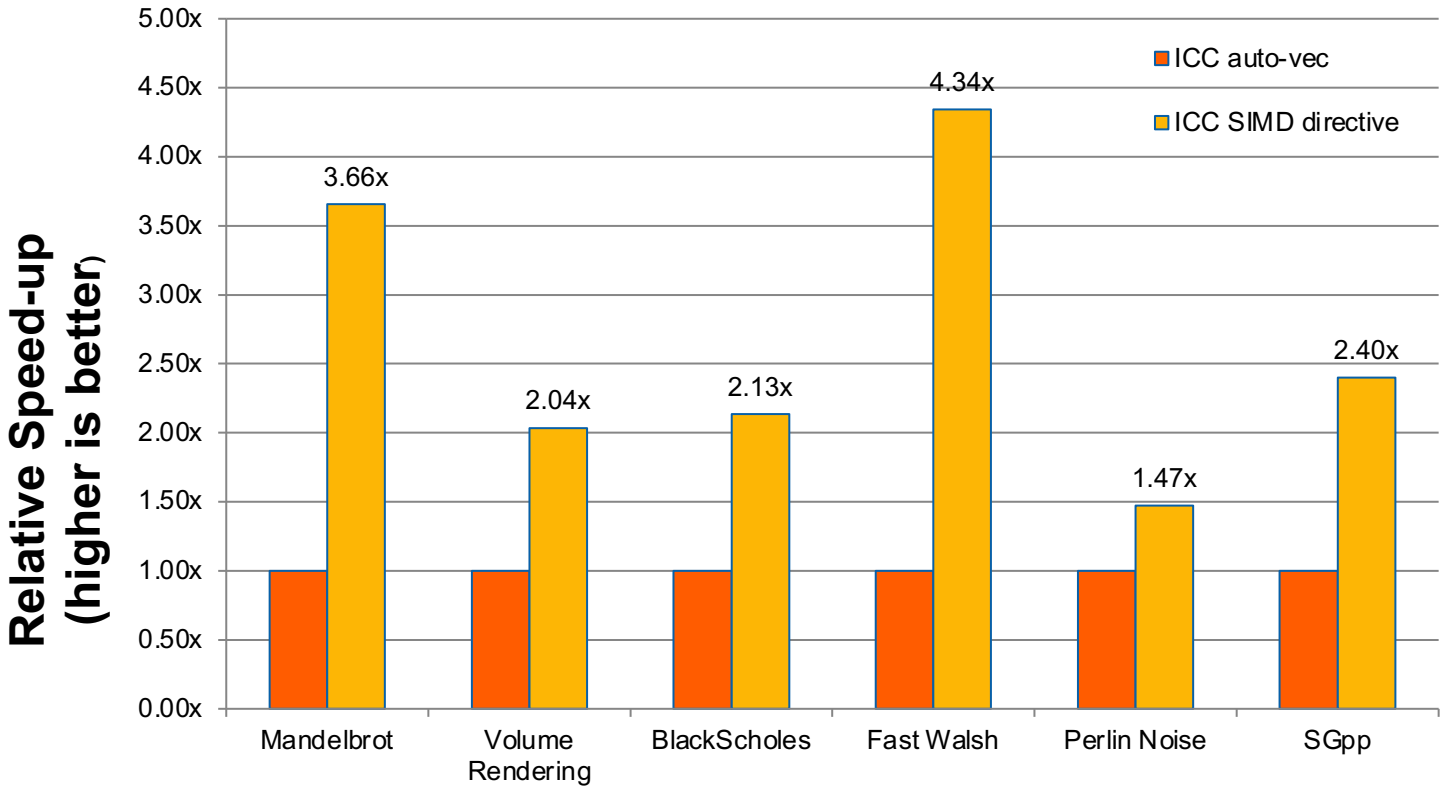
4194304 steps Times in Seconds (50 runs, min time reported)



- Intel Core i7, 2.2 Ghz, 8 GM 1600 MHz DDR3, Apple MacBook Air OS X 10.10.5.
- Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.3.187 Build 20150408

Explicit Vectorization – Performance Impact

Explicit Vectorization looks better when you move to more complex problems.



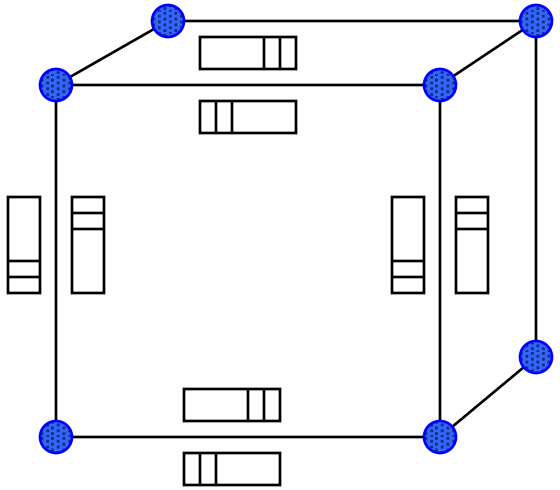
Source: M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, “Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP”, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

Back-up

- Memory Hierarchy
- Vector instructions
- Computer networks
- Design Patterns

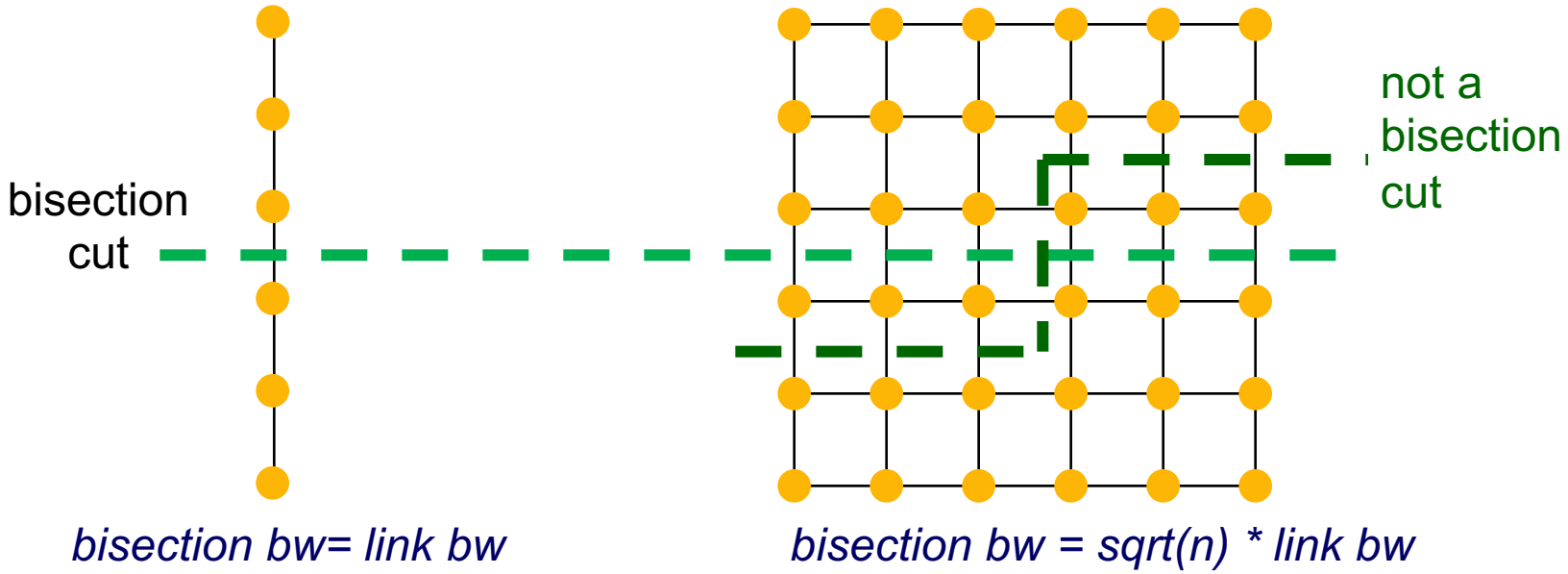
An early focus on networks

- Early machines were:
  - Collection of microprocessors.
  - Communication was performed using bi-directional queues between nearest neighbors.
- Messages were forwarded by processors on path.
  - “Store and forward” networking
- There was a strong emphasis on topology in algorithms, in order to minimize the number of hops.



Properties of a Network: Bisection Bandwidth:

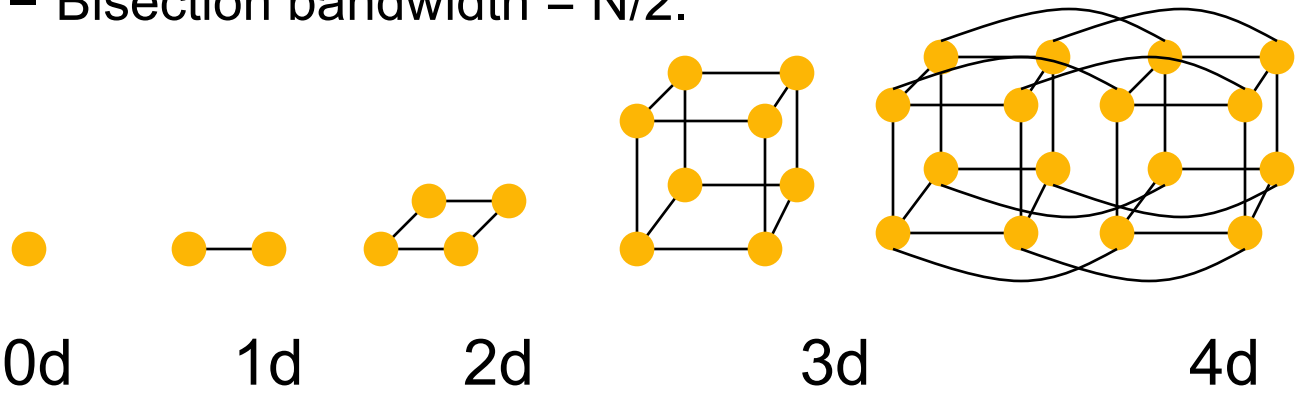
- Bisection bandwidth: bandwidth across smallest cut that divides network into two equal halves
- Bandwidth across “narrowest” part of the network



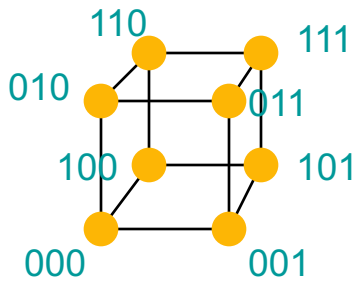
- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Early Networks for connecting processors

- Hpercube: Number of nodes N = 2^d for dimension d.
  - Diameter = d.
  - Bisection bandwidth = N/2.



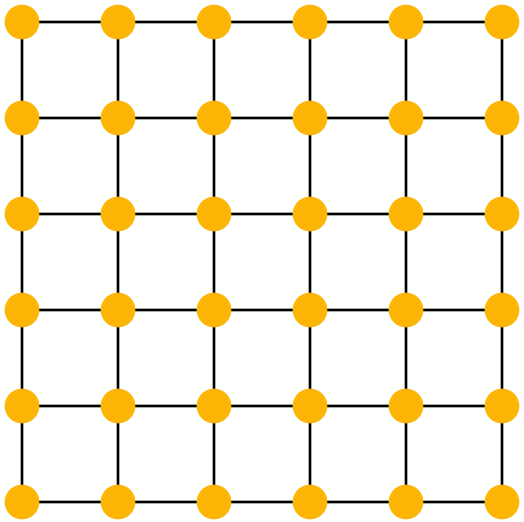
- Popular in early machines
  - Lots of clever algorithms.
- Greycode addressing:
  - Each node connected to d others with 1 bit different.



Meshes and Tori

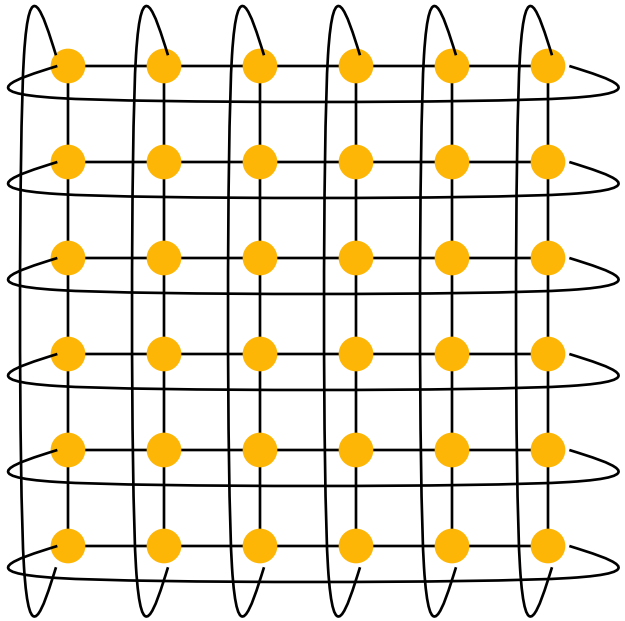
Two dimensional mesh

- Diameter = 2 * (sqrt(N) – 1)
- Bisection bandwidth = sqrt(N)



Two dimensional torus

- Diameter = sqrt(N)
- Bisection bandwidth = 2* sqrt(N)

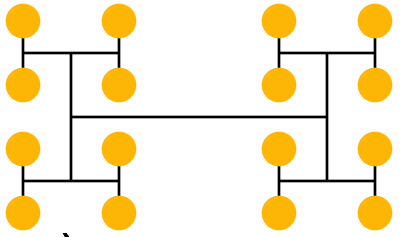


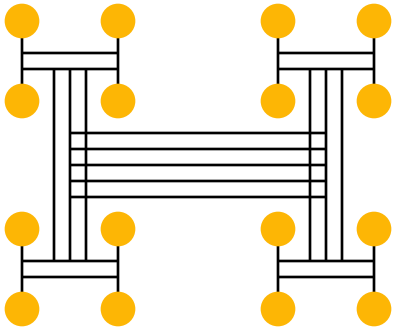
- Intel MPPs from Delta on used a 2D mesh.
- Cray T3D used 3D Torus.
- Mesh and Torus networks are natural fits for algorithms that work with 2D and/or 3D arrays.

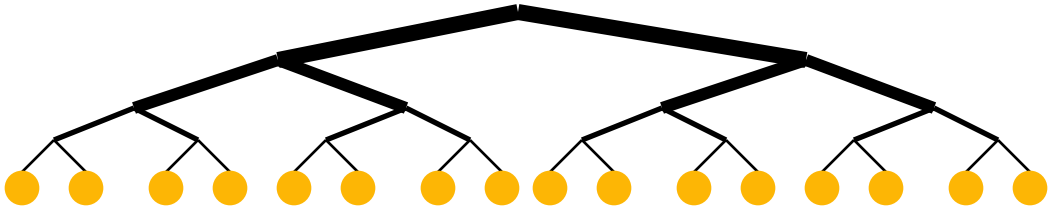
N = the number of nodes in the network

Trees

- Diameter = log n.
- Bisection bandwidth = 1.
- Easy layout as planar graph.
- Many tree algorithms (e.g., summation).
- Fat trees avoid bisection bandwidth problem:
  - More (or wider) links near top.
  - Example: Thinking Machines CM-5. IBM SP.





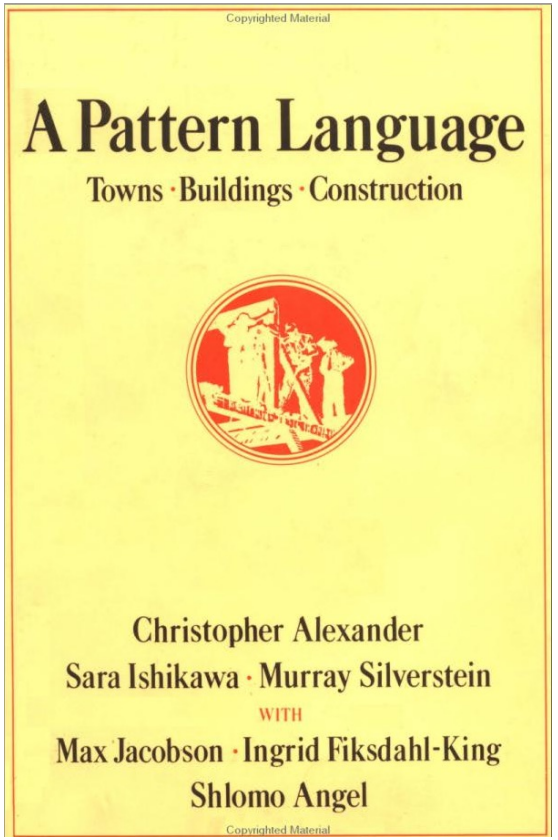


Back-up

- Memory Hierarchy
- Vector instructions
- Computer networks
- Design Patterns

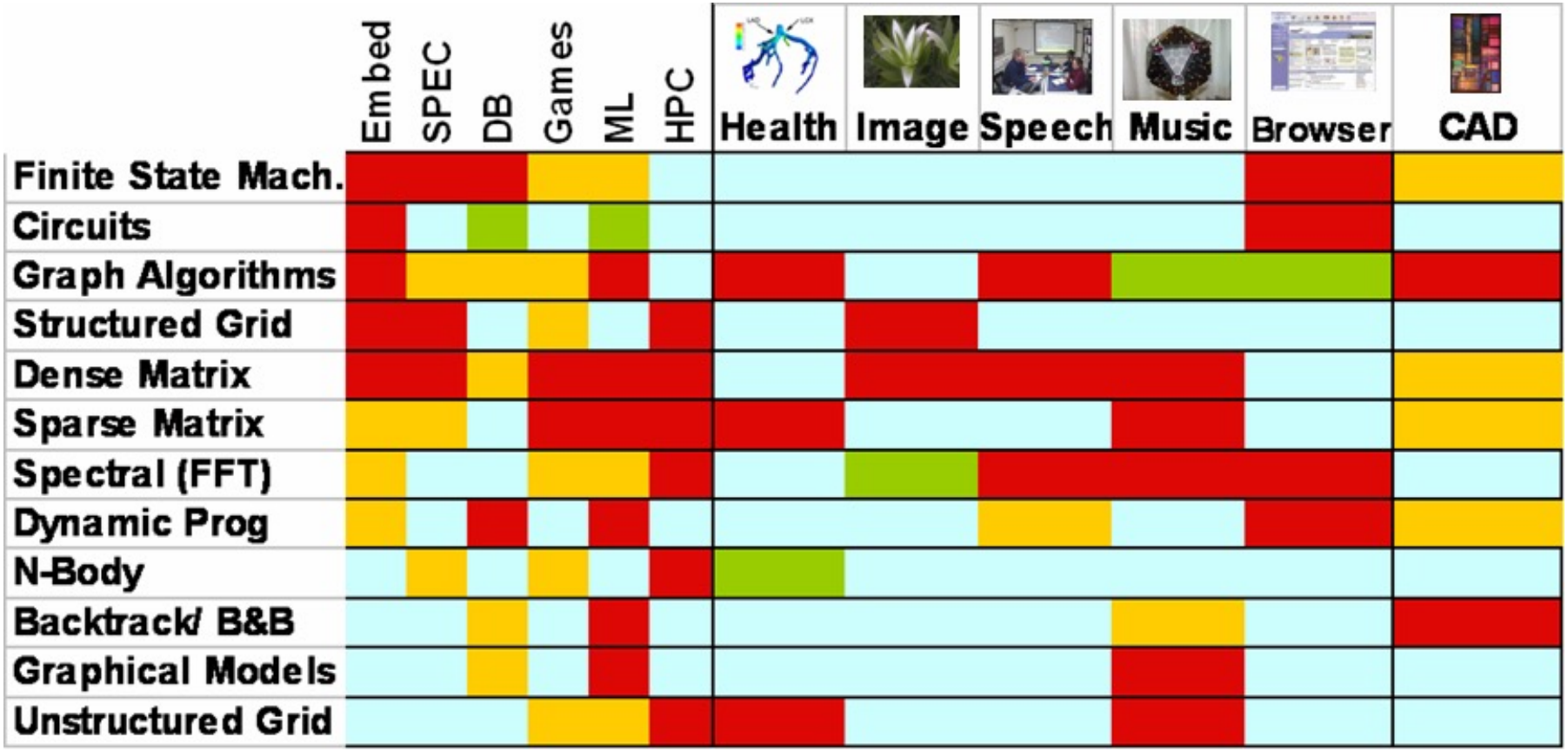
Alexander's Pattern Language

- Christopher Alexander's approach to (civil) architecture:
  - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." *Page x*, *A Pattern Language,* Christopher Alexander
- Alexander's 253 (civil) architectural patterns range from the creation of cities (2. distribution of towns) to particular building problems (232. roof cap)
- A pattern language is an organized way of tackling an architectural problem using patterns
- Main limitation:
  - It's about civil not software architecture!!!



Computational Patterns

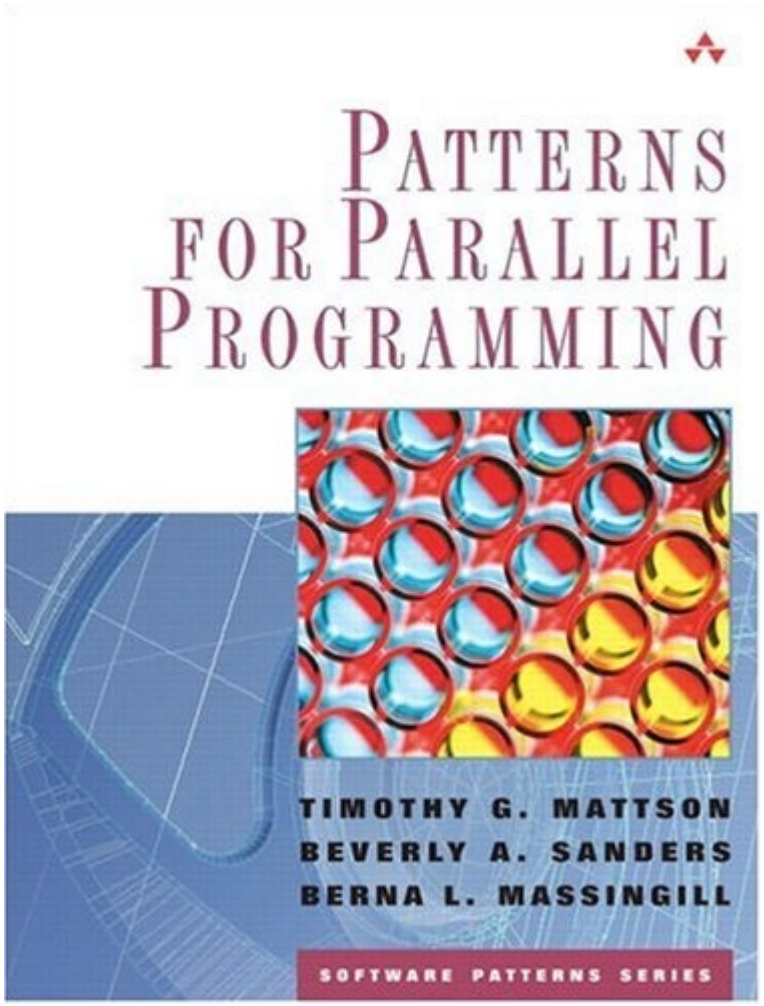
The Dwarfs from "The Berkeley View" (Asanovic et al.)
Dwarfs form our key computational patterns



Patterns for Parallel Programming (PLPP)

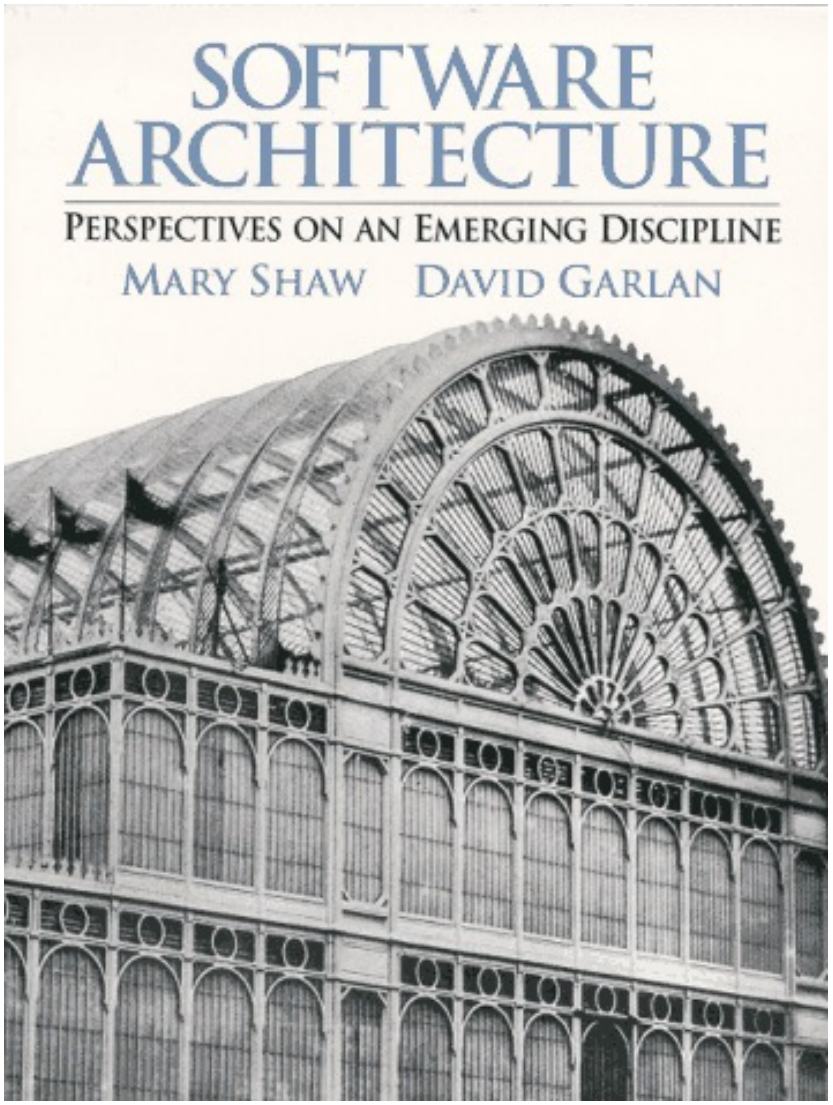
- PLPP is the first attempt to develop a complete *pattern language* for parallel software development.
- PLPP is a great model for a pattern language for parallel software
- PLPP mined scientific applications that utilize a monolithic application style
- PLPP doesn't help us much with horizontal composition

- Much more useful to us than: *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson & Vlissides, Addison-Wesley, 1995.

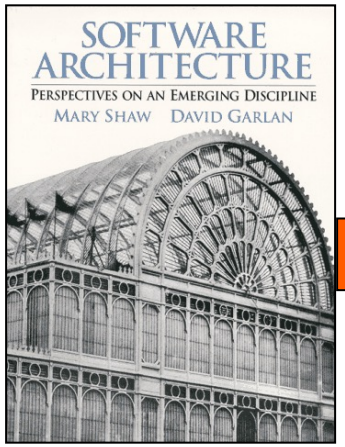


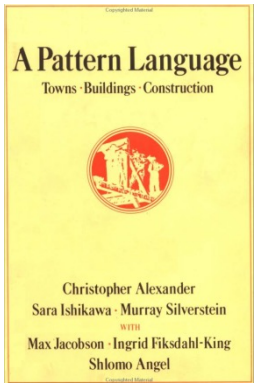
Structural programming patterns

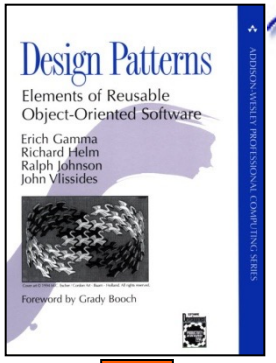
- In order to create more complex software it is necessary to compose programming patterns
- For this purpose, it has been useful to induct a set of patterns known as “architectural styles”
- Examples:
  - pipe and filter
  - event based/event driven
  - layered
  - Agent and repository/blackboard
  - process control
  - Model-view-controller

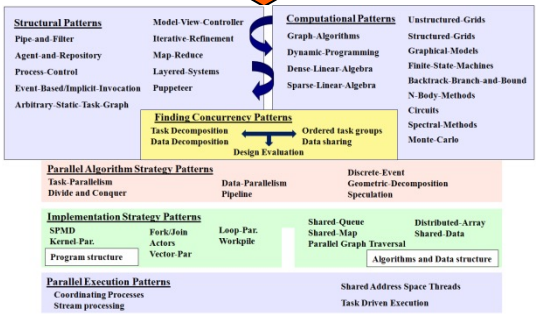


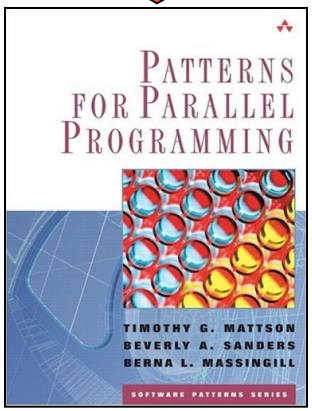
To get frameworks right … start with an understanding of software architecture



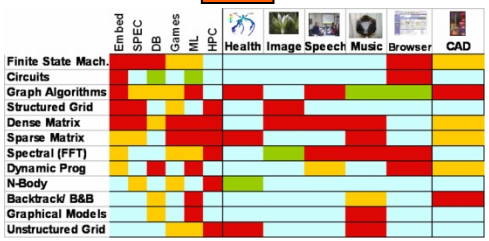






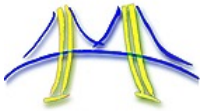


PLPP: Pattern language of Parallel Programming



13 dwarves

OPL/PLPP 2012



Applications

Structural Patterns

- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Arbitrary-Static-Task-Graph
- Model-View-Controller
- Iterative-Refinement
- Map-Reduce
- Layered-Systems
- Puppeteer

Computational Patterns

- Graph-Algorithms
- Dynamic-Programming
- Dense-Linear-Algebra
- Sparse-Linear-Algebra
- Unstructured-Grids
- Structured-Grids
- Graphical-Models
- Finite-State-Machines
- Backtrack-Branch-and-Bound
- N-Body-Methods
- Circuits
- Spectral-Methods
- Monte-Carlo

Finding Concurrency Patterns

Task Decomposition
Data Decomposition
Ordered task groups
Data sharing
Design Evaluation

Parallel Algorithm Strategy Patterns

- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline
- Discrete-Event
- Geometric-Decomposition
- Speculation

Implementation Strategy Patterns

- SPMD
- Kernel-Par.
- Fork/Join
- Actors
- Vector-Par
- Loop-Par.
- Task-queue

Program structure

- Shared-Queue
- Shared-Map
- Parallel Graph Traversal
- Distributed-Array
- Shared-Data

Algorithms and Data structure

Parallel Execution Patterns

- Coordinating Processes
- Stream processing
- Shared Address Space Threads
- Task Driven Execution

Concurrency Foundation constructs (not expressed as patterns)

Thread/proc management | Communication | Synchronization

OPL/PLPP 2012

Applications

Structural Patterns

Pipe-and-Filter

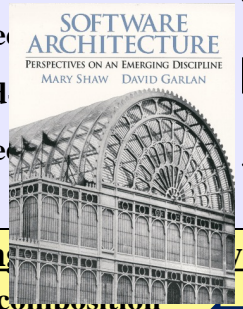
Agent-and-Repository

Model-View-Controller

Iterative-Refinement

Map-Re

Garlan and Shaw Architectural Styles



Computational Patterns

Graph-Algorithms

Dynamic-Programming

Dense-Linear-Algebra

Spa

Unstructured-Grids

Berkeley View 13 dwarfs

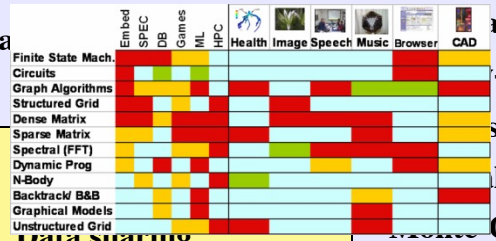
Finite-State-Machines

ack-Branch-and-Bound

-Methods

l-Methods

Carlo



Finding Patterns

Task Decomposition

Data Decomposition

Data Sharing

Design Evaluation

Parallel Algorithm Strategy Patterns

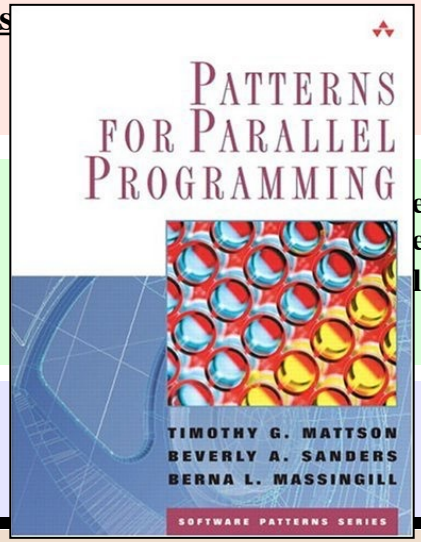
Task-Parallelism

Divide and Conquer

Discrete-Event

Geometric-Decomposition

Speculation



Implementation Strategy Patterns

SPMD

Kernel-Par.

Fork/Join

Actors

Vector-Par

Program structure

ed-Queue

ed-Map

llel Graph Traversal

Distributed-Array

Shared-Data

Algorithms and Data structure

Parallel Execution Patterns

Coordinating Processes

Stream processing

Shared Address Space Threads

Task Driven Execution

Concurrency Foundation constructs (not expressed as patterns)

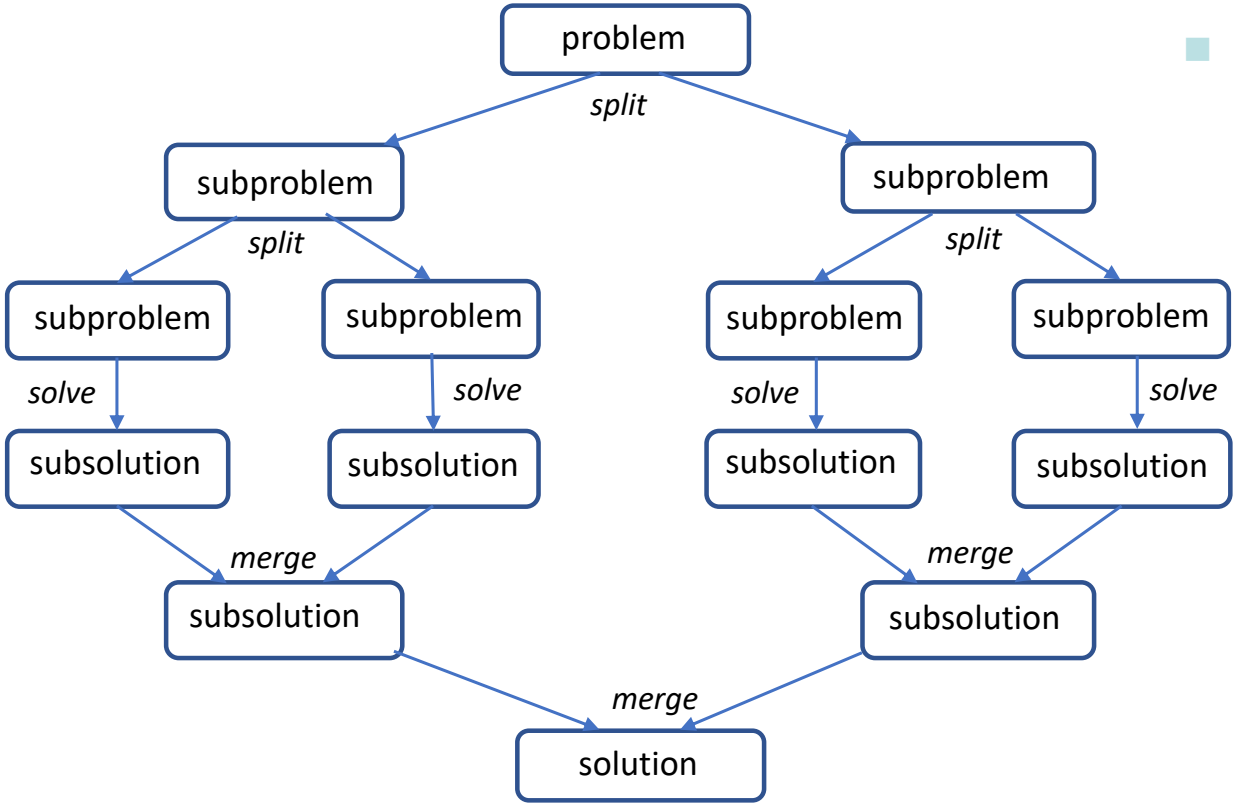
Thread/proc management

Communication

Synchronization

Divide and Conquer

- Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly



- 3 Options:
  - Do work as you split into sub-problems
  - Do work only at the leaves
  - Do work as you recombine