OpenMP

Parallel Computing Introduction

Tim Mattson

Intel Corp.

timothy.g.mattson@intel.com

1

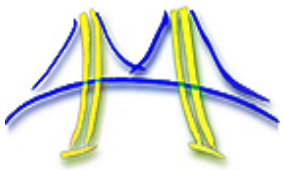
Disclaimer

- The views expressed in this talk are those of the speaker and not his employer.
- If I say something stupid, blame me … not the smart people I work with.

I work in Intel's research labs. I don't build products. Instead, I get to poke into dark corners and think silly thoughts… just to make sure we don't miss any great ideas.

Hence, my views are by design far "off the roadmap".

I have a great job!



Slides marked with this symbol come from a course at UC Berkeley that I teach with Professor Kurt Keutzer.

2

Introduction

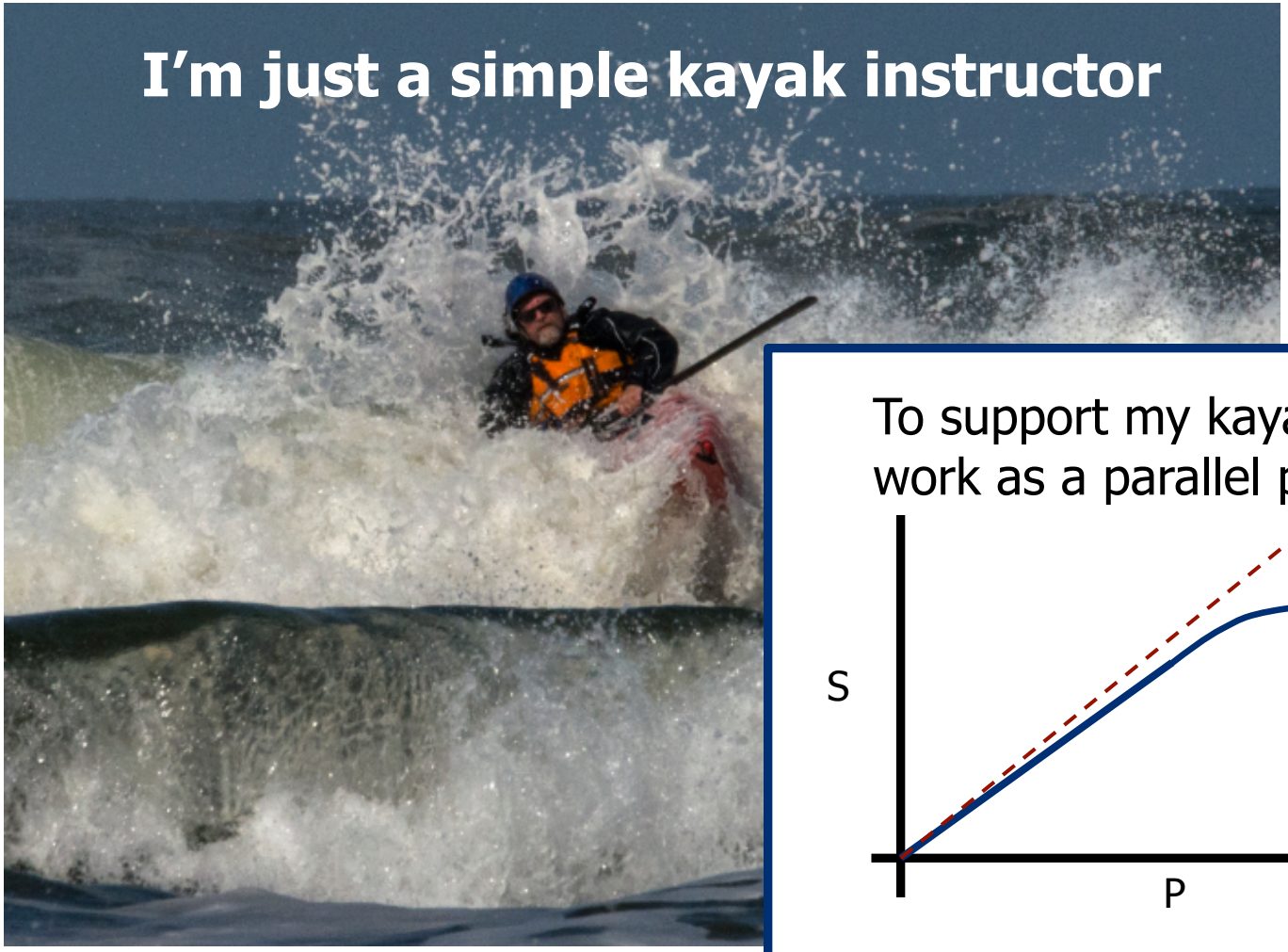


Photo © by Greg Clopton, 2014

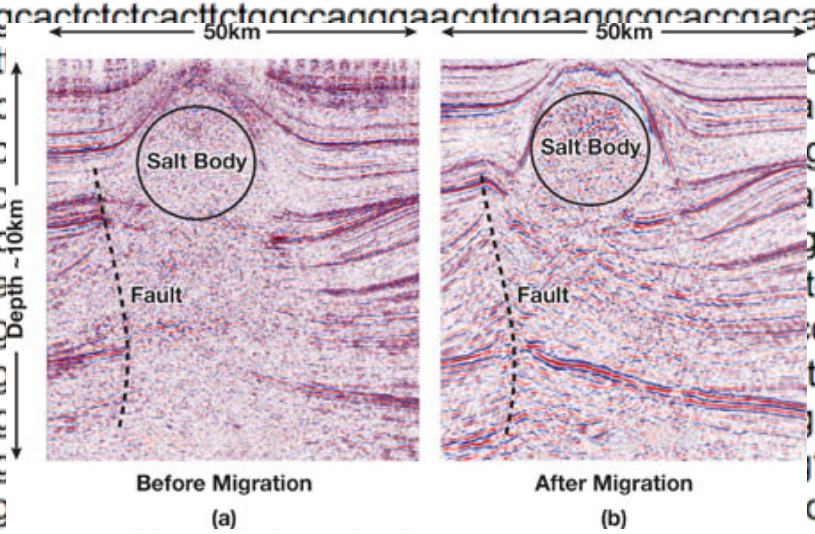
To support my kayaking habit I work as a parallel programmer

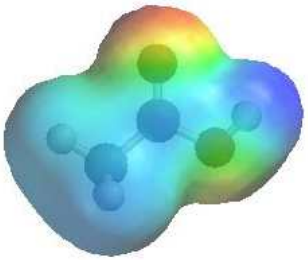
Which means I know how to turn math into lines on a speedup plot

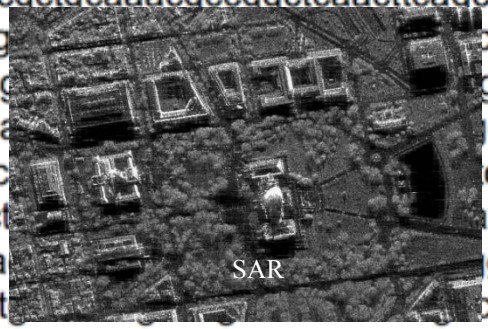
Outline

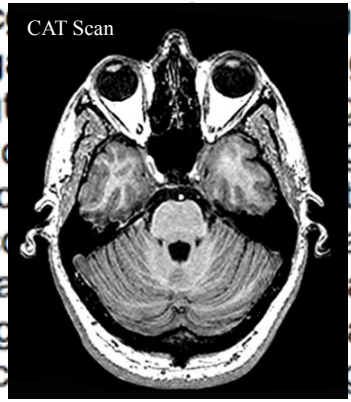
- High Performance computing: A hardware system view
- The processors in HPC systems
- Parallel Computing: Basic Concepts
- The Fundamental patterns of parallel Computing

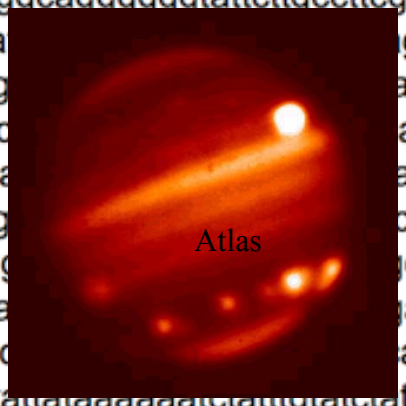
High performance computing is addictive

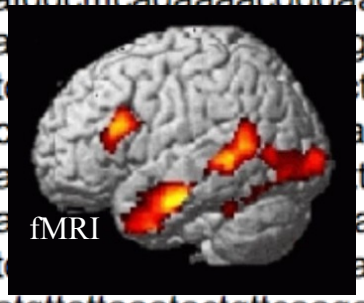


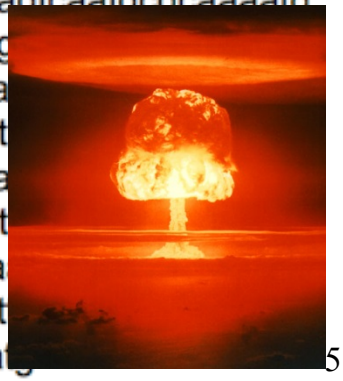












The birth of Supercomputing

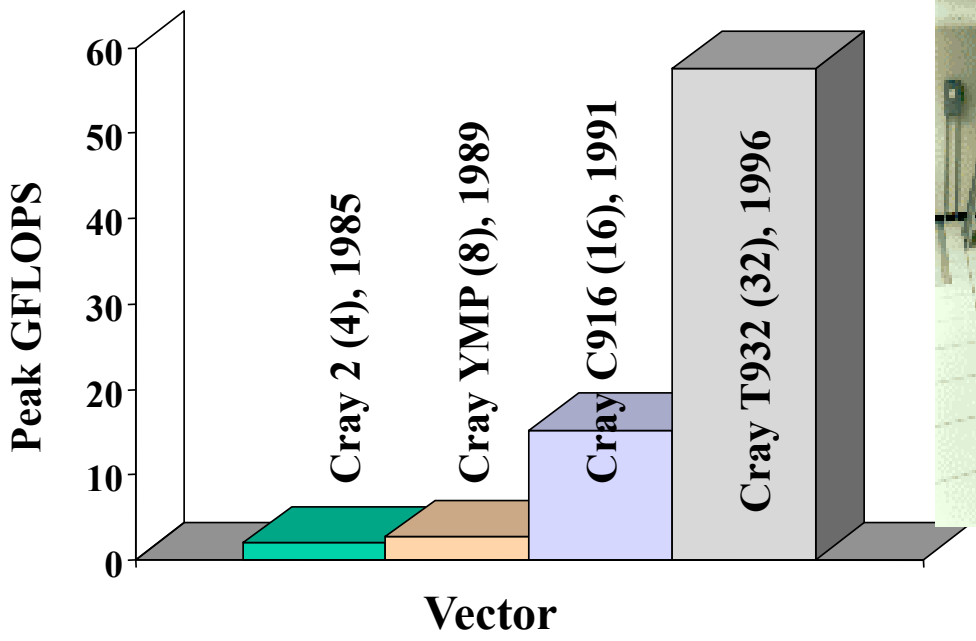
- On July 11, 1977, the CRAY-1A, serial number 3, was delivered to NCAR. The system cost was $8.86 million ($7.9 million plus $1 million for the disks).

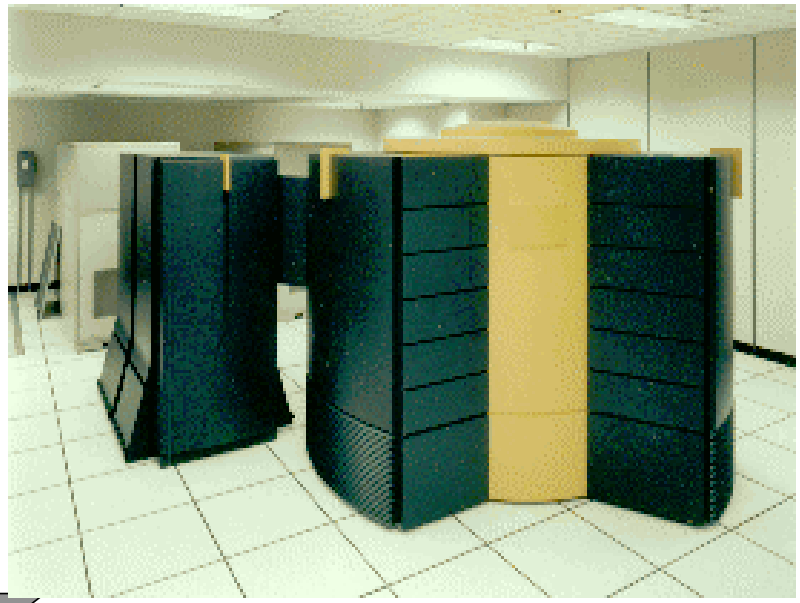
- The CRAY-1A:
  - 12.5-nanosecond clock,
  - 64 vector registers,
  - 1 million 64-bit words of high-speed memory.
  - Speed:
    - 160 MFLOPS vector peak speed
    - 110 MFLOPS Linpack 1000 (best effort)
- Cray software … by 1978
  - Cray Operating System (COS),
  - the first automatically vectorizing Fortran compiler (CFT),
  - Cray Assembler Language (CAL) were introduced.

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

The original Supercomputers
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

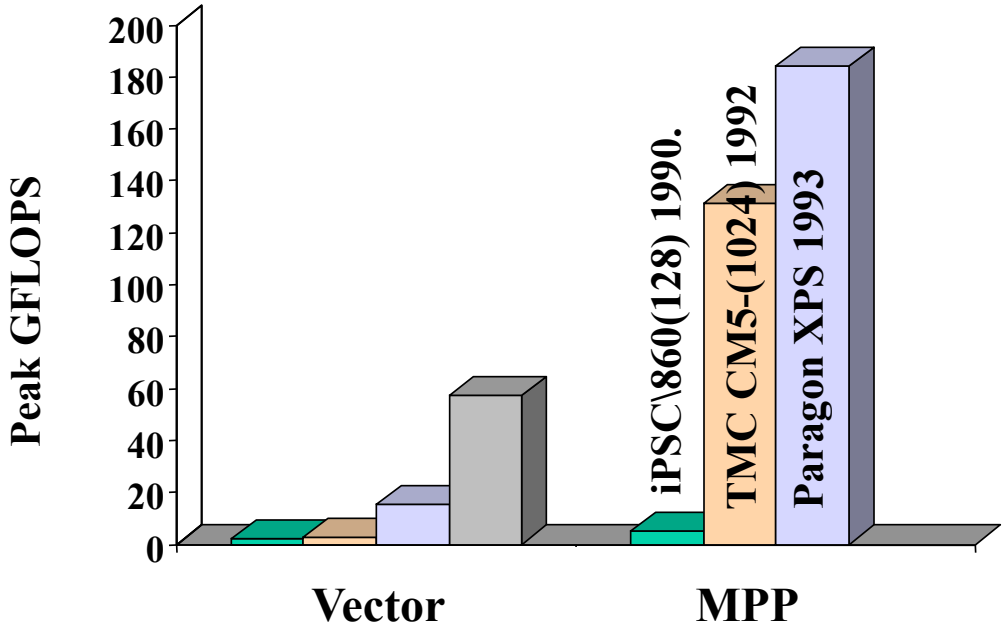
The cosmic cube, Charles Seitz
Communications of the ACM, Vol 28, number 1 January 1985, p. 22

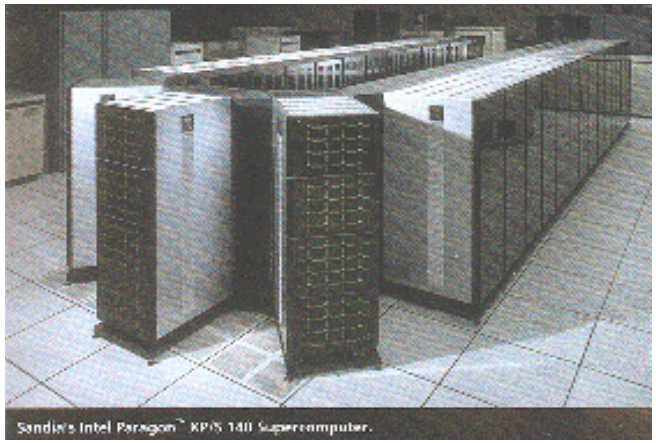
Launched the “attack of the killer micros”
Eugene Brooks, SC’90

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

Improving CPU performance and weak scaling helped MPPs dominate supercomputing

- Parallel computers with large numbers of commercial off the shelf 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

SIMD computers … the other MPP supercomputer

"… we want to build a computer that will be proud of us", Danny Hillis

Thinking machines CM-2: The Classic Symmetric SIMD supercomputer (mid-80's):

Description: Up to 64K bit-serial processing elements.

Strength: Supports deterministic programming models … single thread of control for ease of understanding.

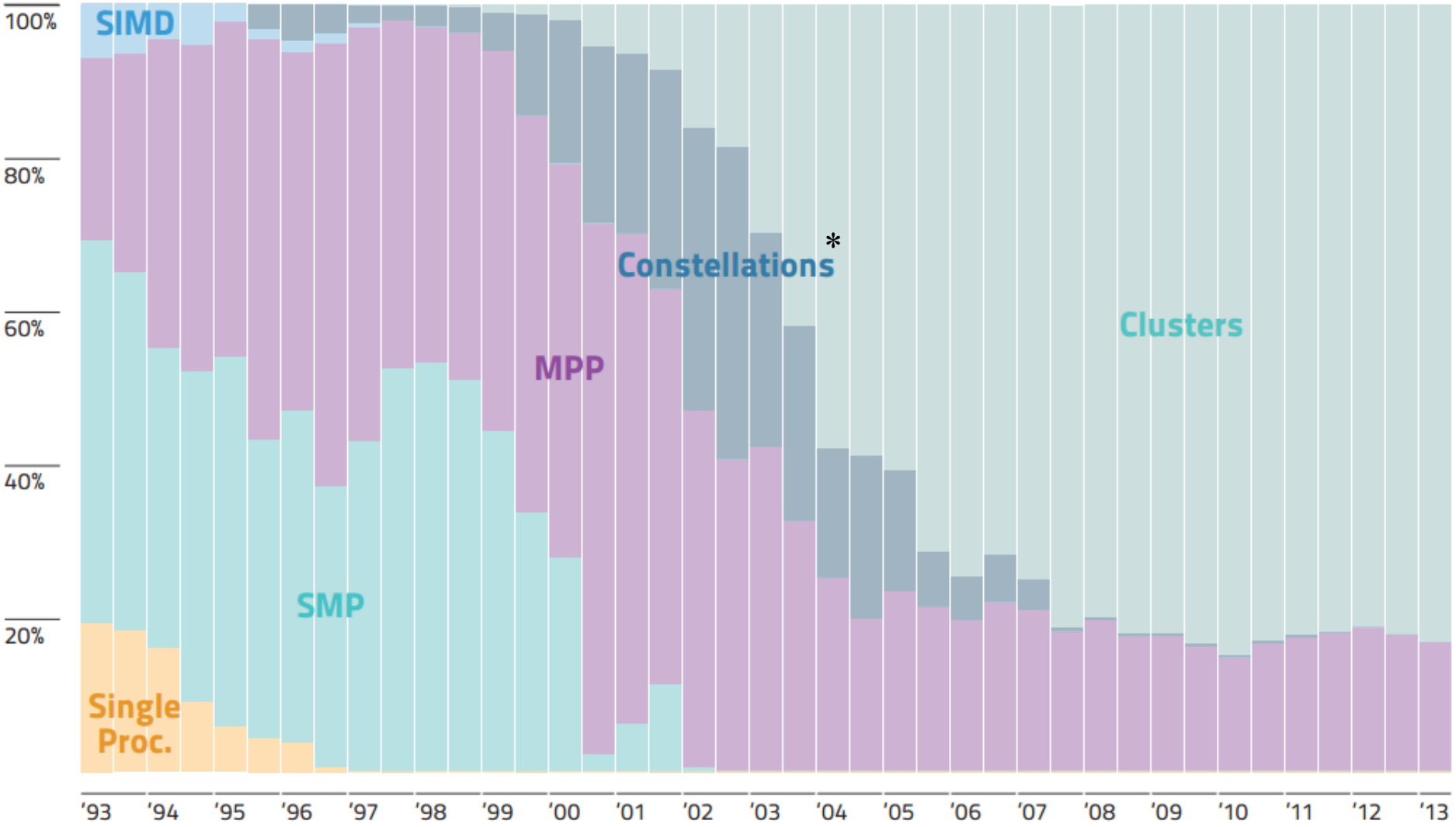
Weakness: Poor floating point performance. Programming model was not general enough. TMC struggled throughout the 90's and filed for bankruptcy in 1994.

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 a cluster on their own

- First clusters appeared in the late 80's (Stacks of "SPARC pizza boxes")
- 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

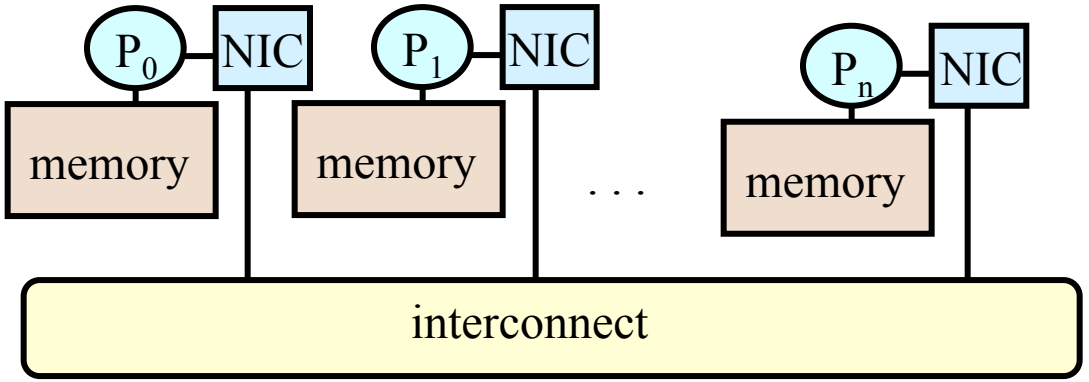


Source: http://s.top500.org/static/lists/2013/06/TOP500_201306_Poster.pdf

*Constellation: A cluster for which the number of processors on a node is greater than the number of nodes in the cluster. I've never seen anyone use this term outside of the top500 list.

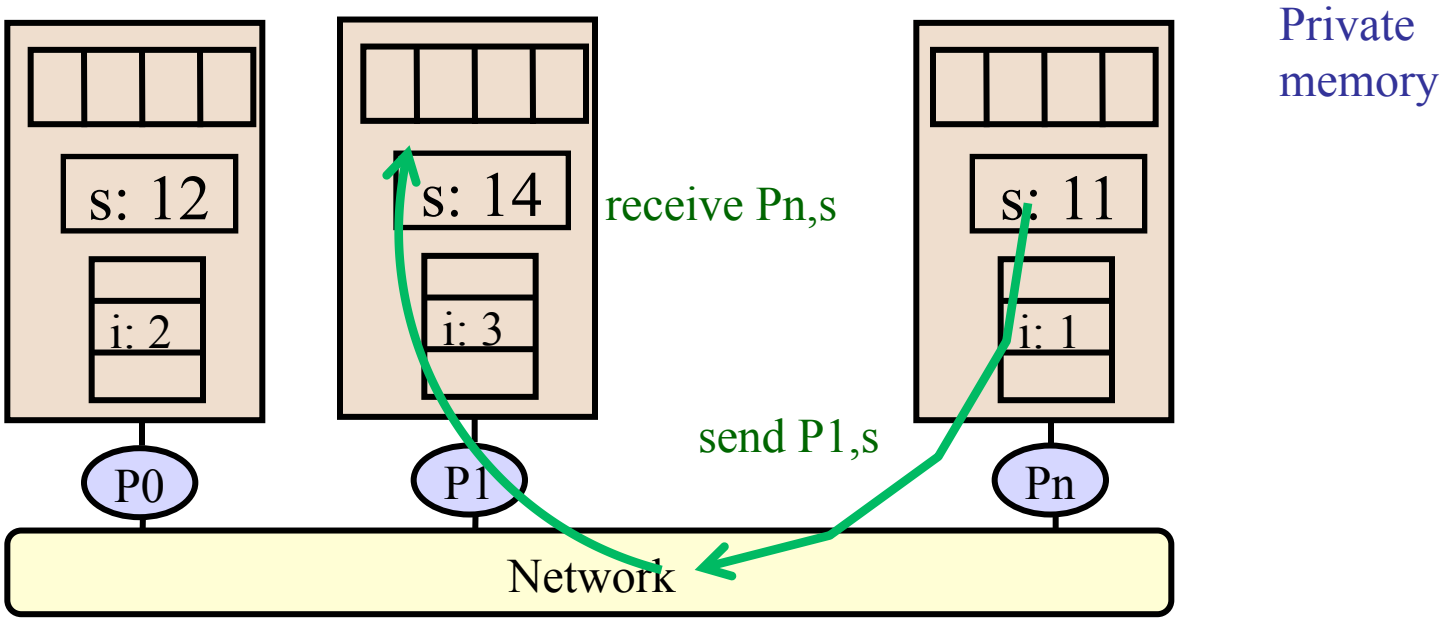
Execution model: Distributed memory MIMD

- Cluster or MPP … the future is clear. Distributed memory scales and is more energy efficient (cache coherence moves lots of electrons around and that consumes lots of power).
- 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.



Programming Model: Message Passing

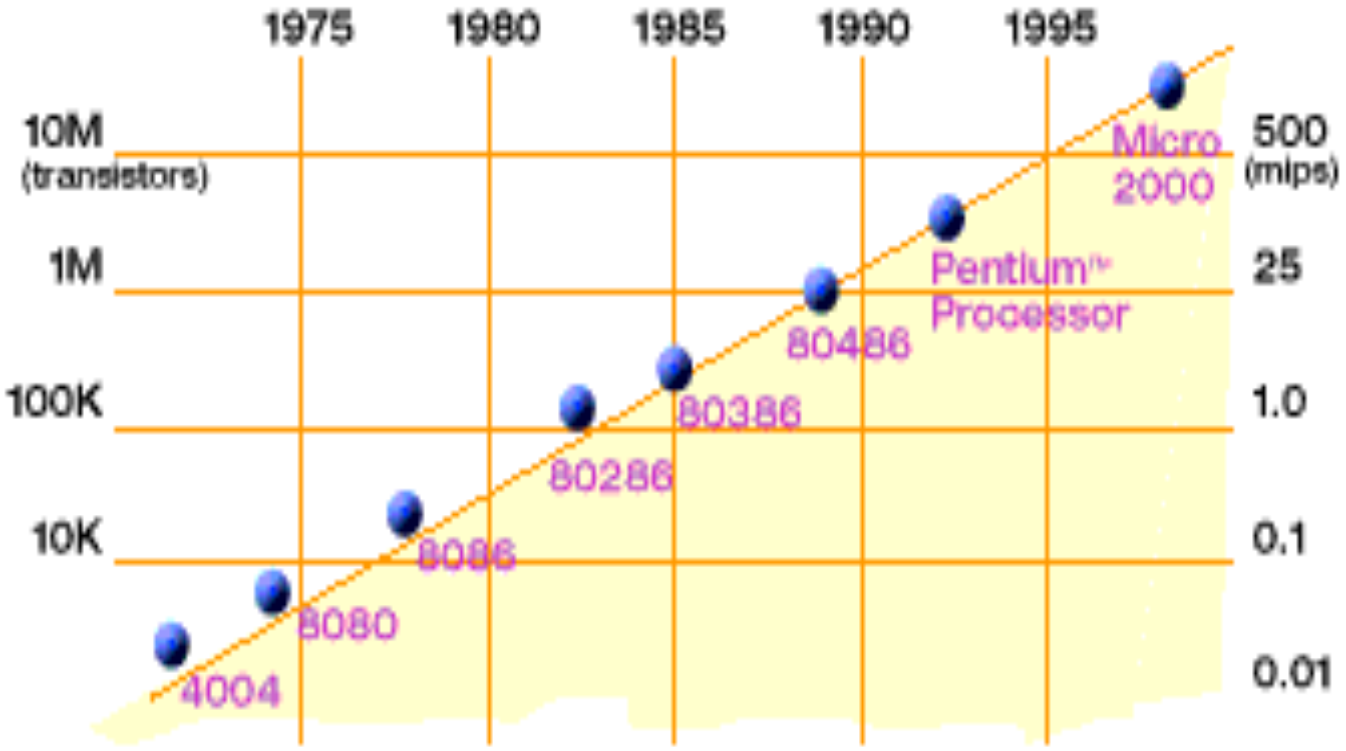
- Program consists of a collection of **named** processes.
  - **Number of processes usually fixed at program startup time**
  - **Local address space per node -- NO physically shared memory.**
- Processes communicate by explicit send/receive pairs
  - **Coordination is implicit in every communication event.**
  - **MPI (Message Passing Interface) is the most commonly used SW**

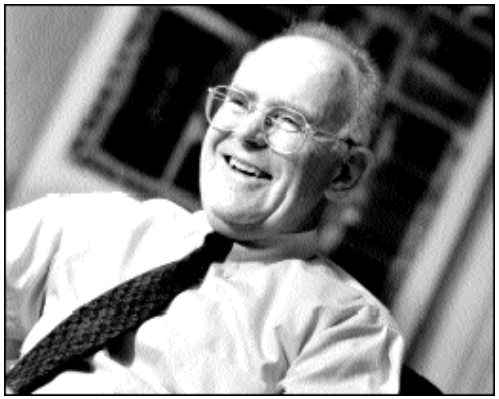


Outline

- High Performance computing: A hardware system view
- The processors in HPC systems
- Parallel Computing: Basic Concepts
- The Fundamental patterns of parallel Computing

Moore's Law

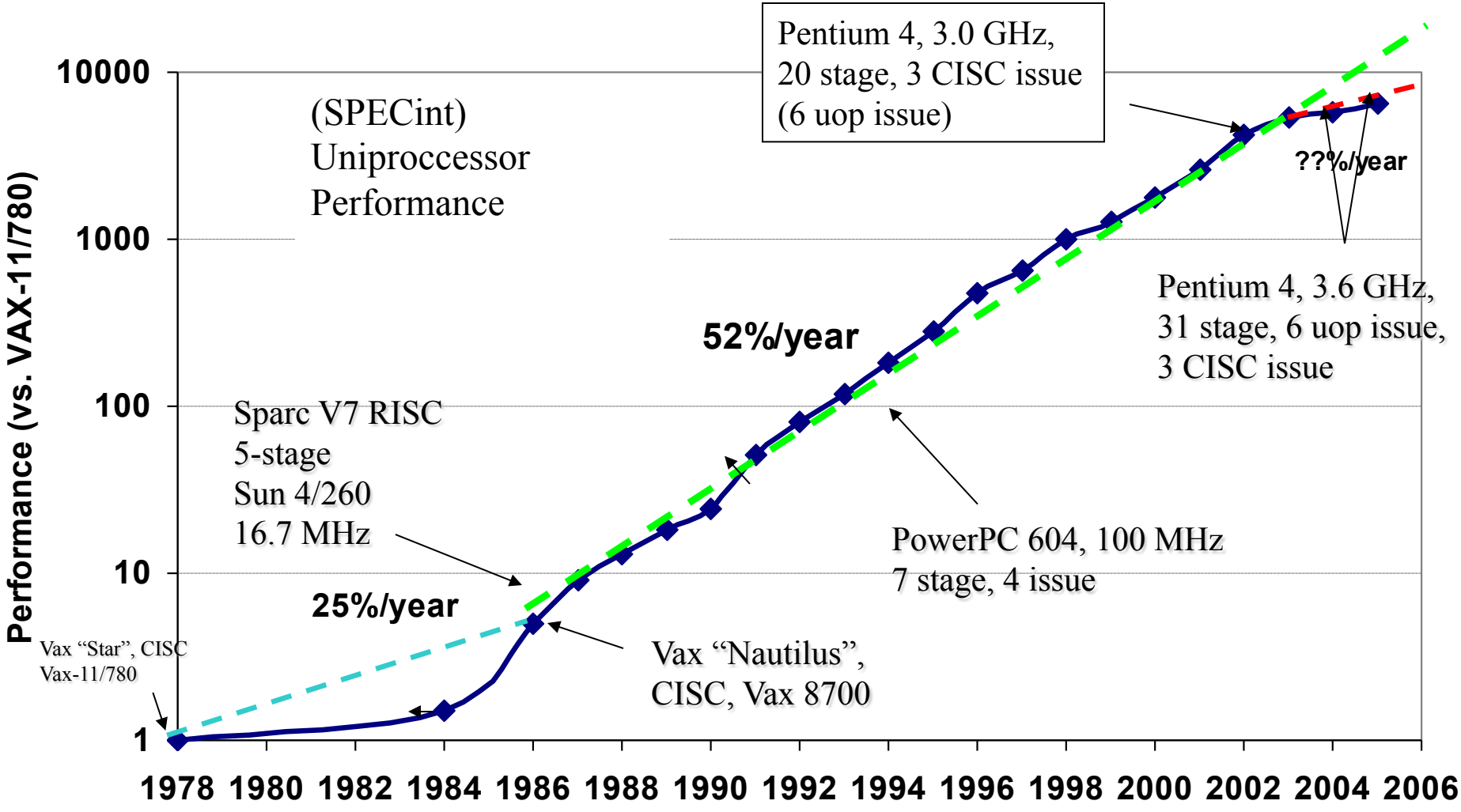




- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Over the last 50 years, transistor densities have increased as he predicted.

Slide source: UCB CS 194 Fall'2010

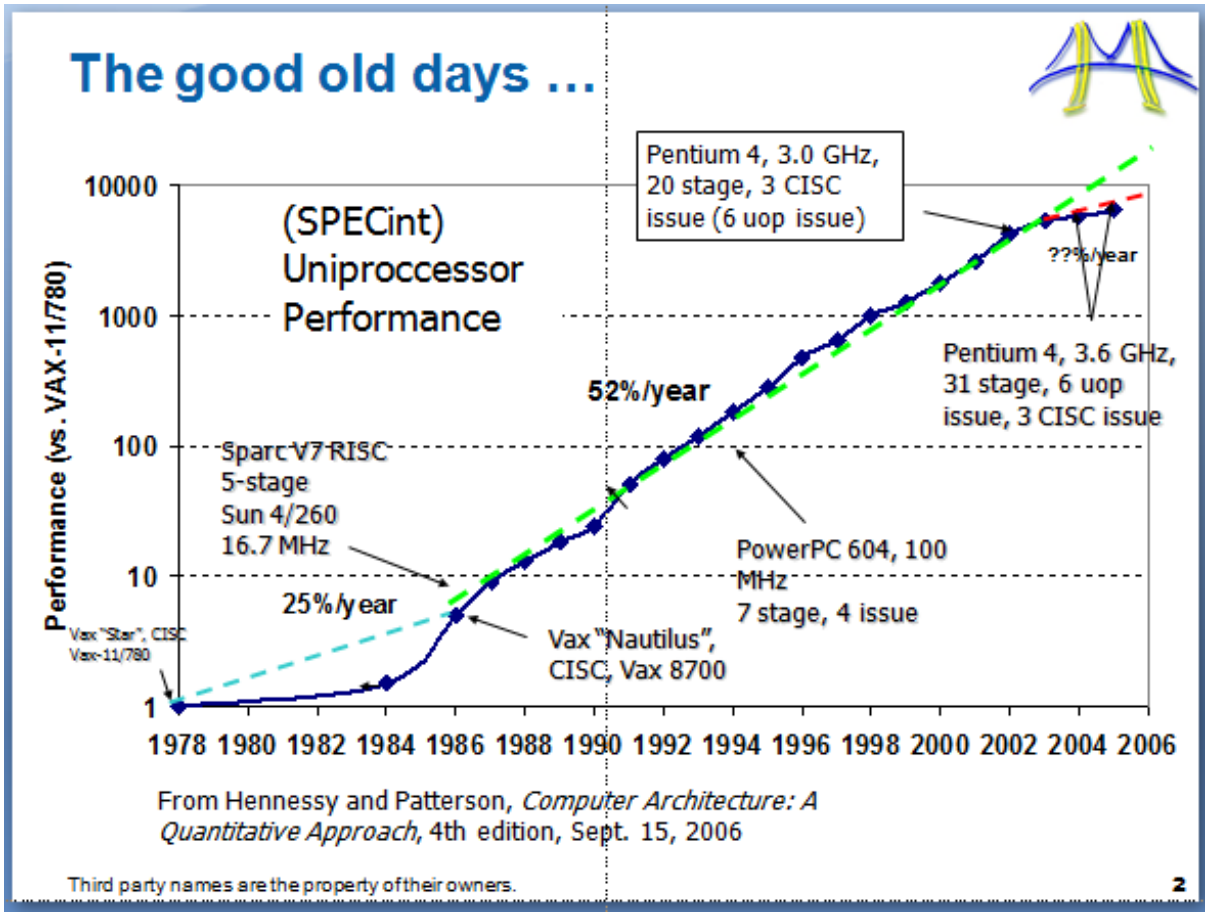
The good old days …



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

The Hardware/Software contract

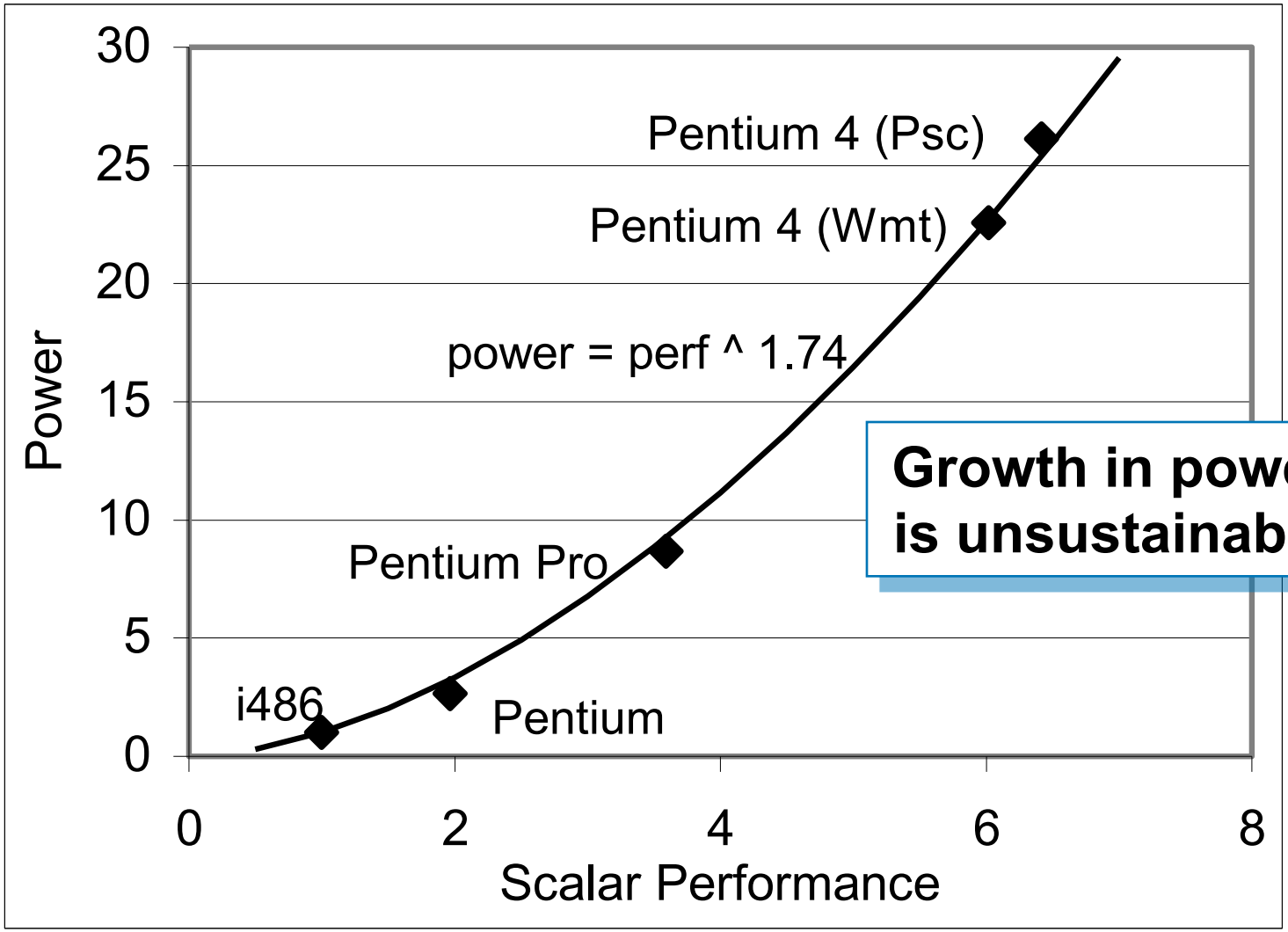
- Write your software as you choose and the HW-geniuses will take care of performance.



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

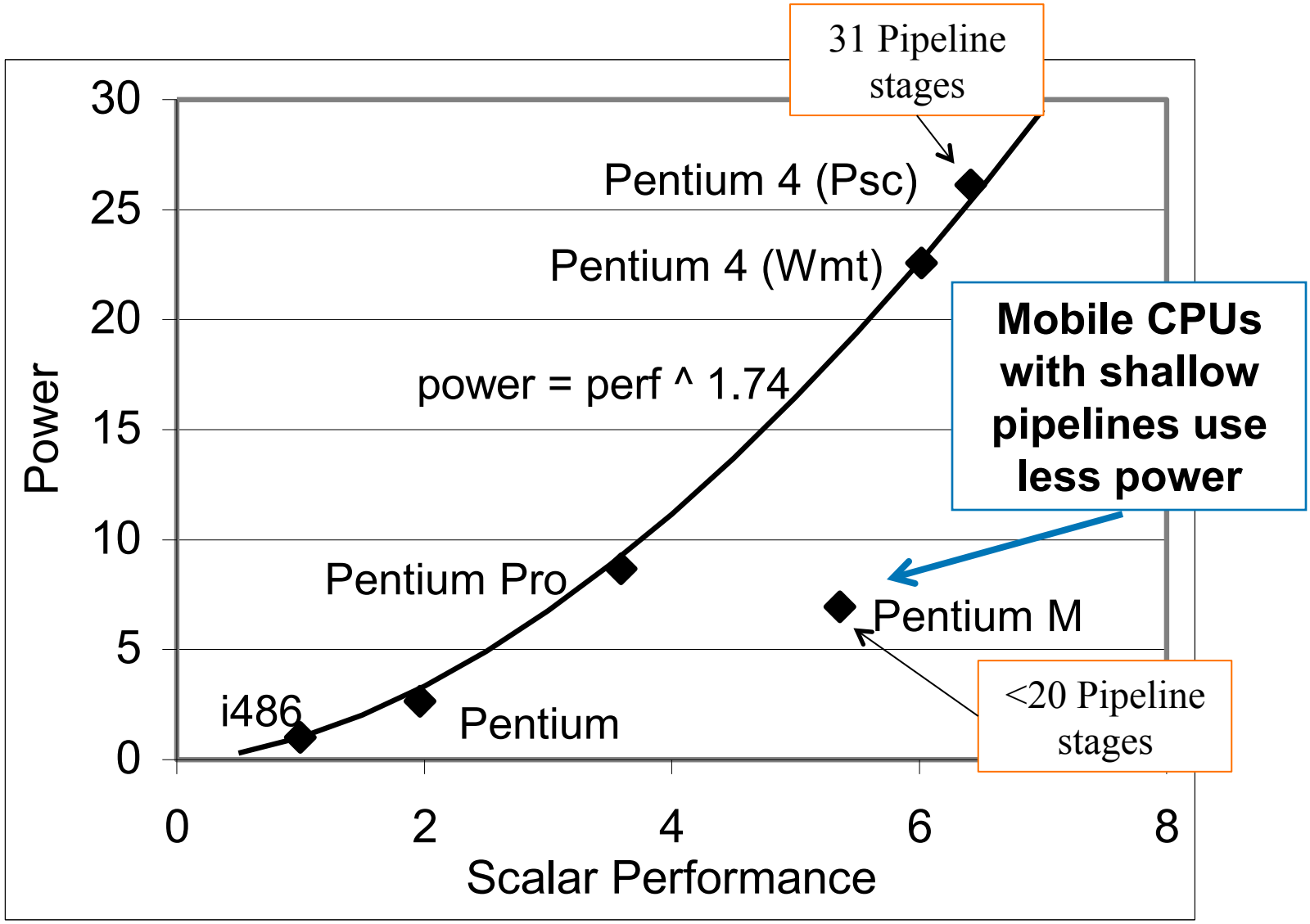
Why many core? Its all about Power.

Power vs Performance (normalized to i486 process tech.)



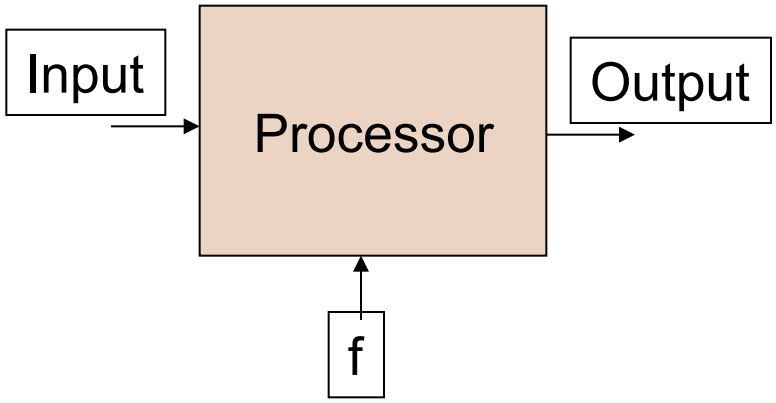
Source: Ed Grochowski, Intel

Design with Power in mind



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:

$V * q = W.$

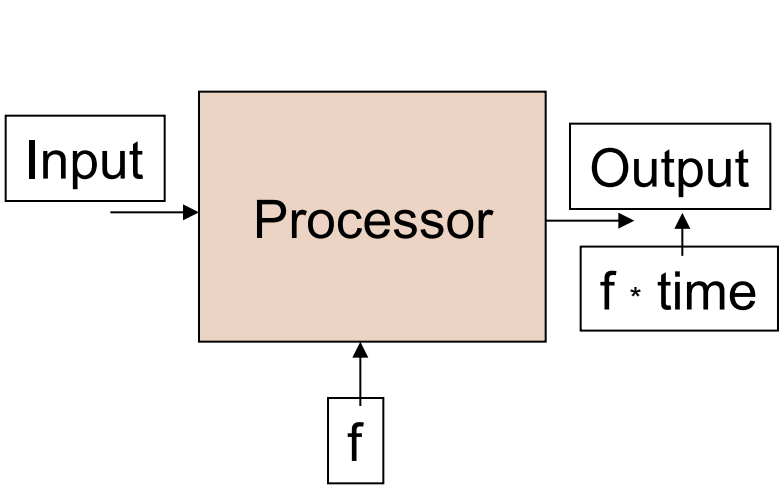
But for a circuit $q = CV$ so

$W = CV^2$

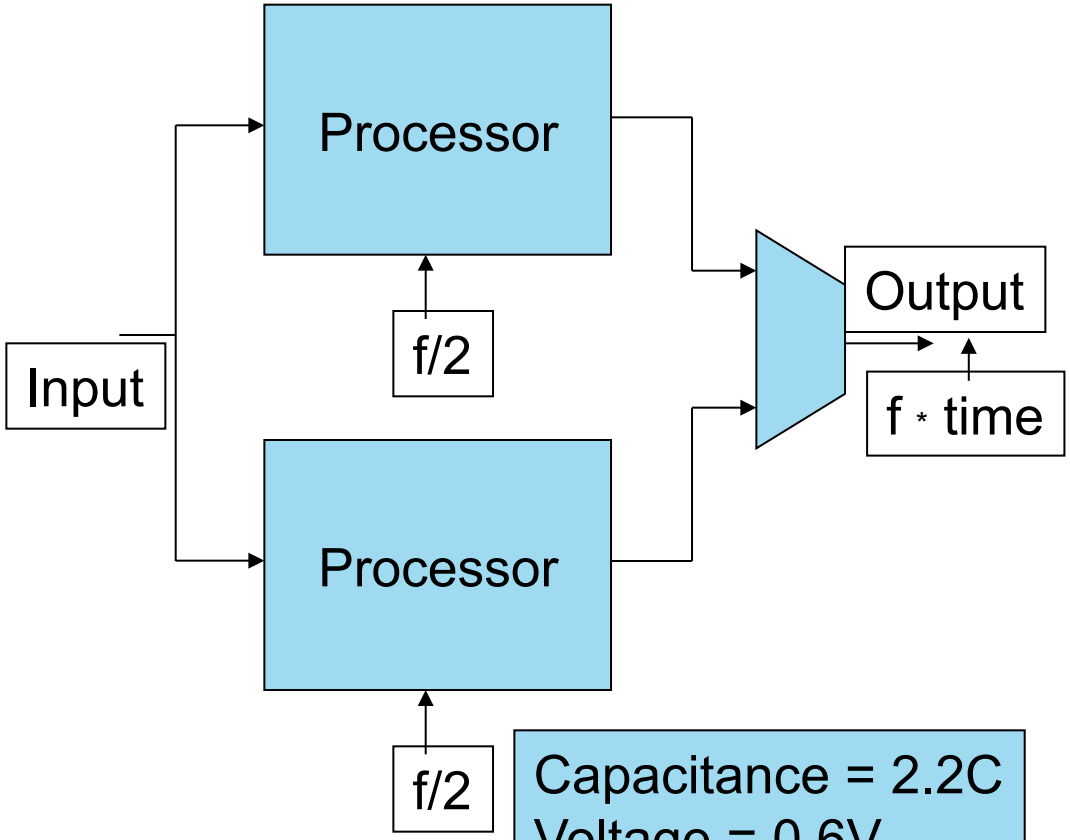
power is work over time … or how many times in a second we oscillate the circuit

Power = W* F → Power = CV^2f

... Reduce power by adding cores



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

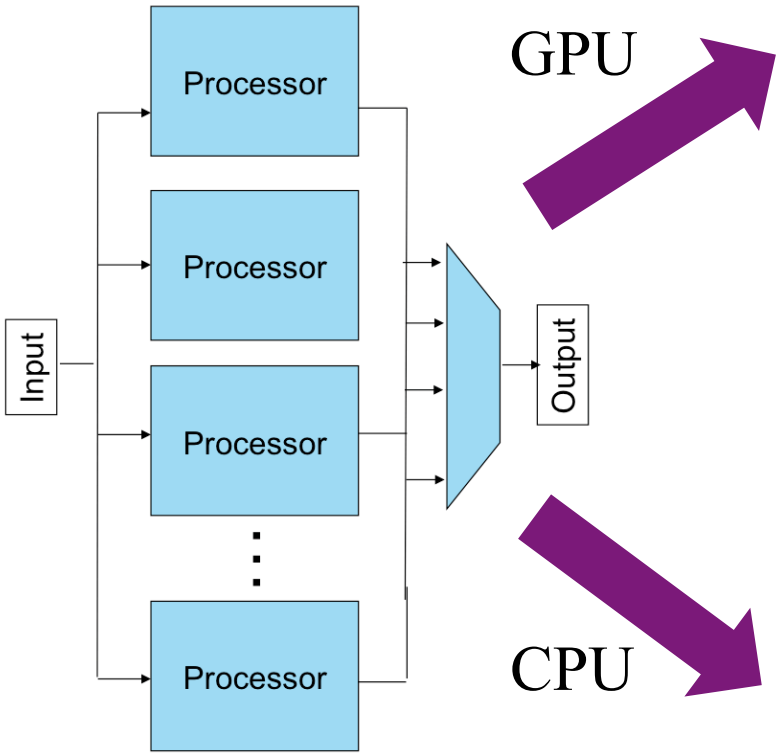


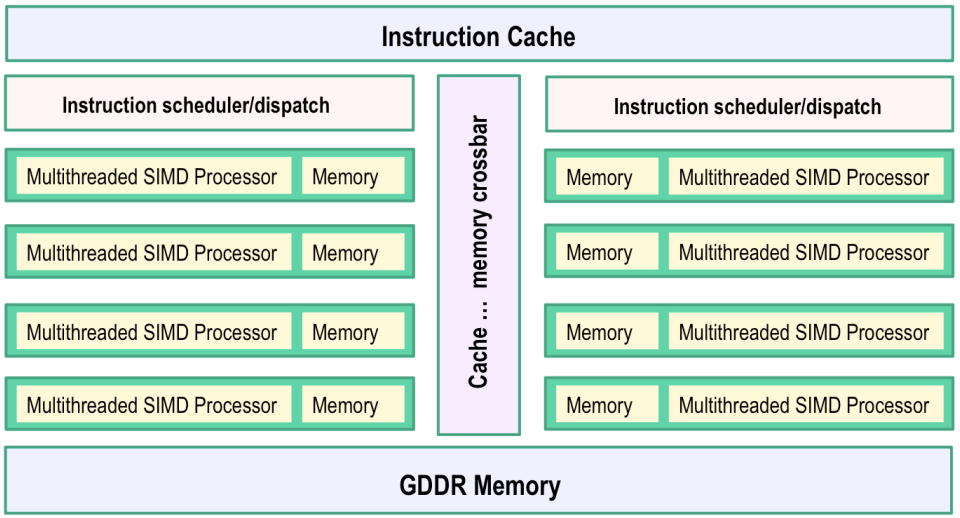
Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

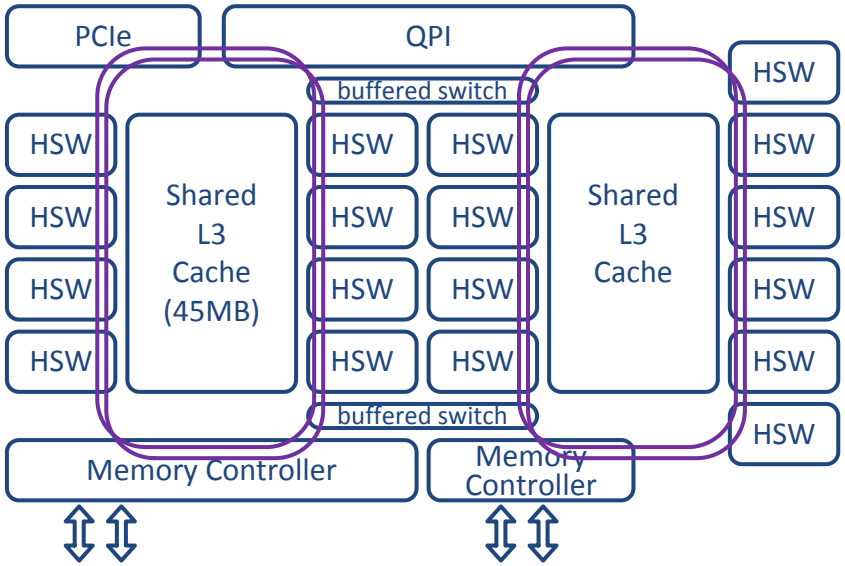
Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

Source:
Vishwani Agrawal

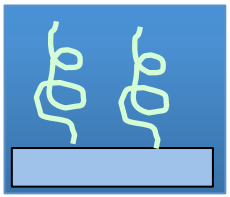
... Many core: we are all doing it



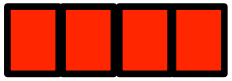




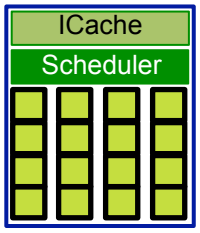
Hardware Diversity: Basic Building Blocks



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

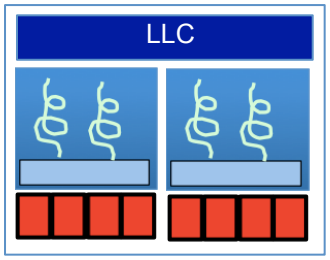


SIMD: Single Instruction Multiple Data.
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.

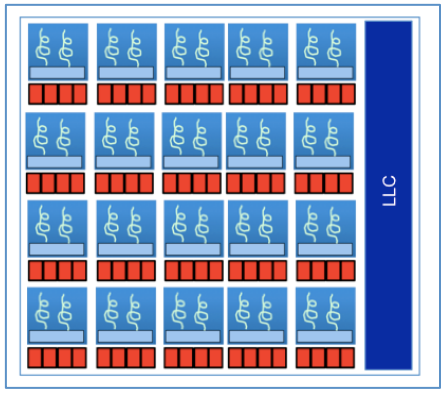


SIMT: Single Instruction Multiple Threads.
A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

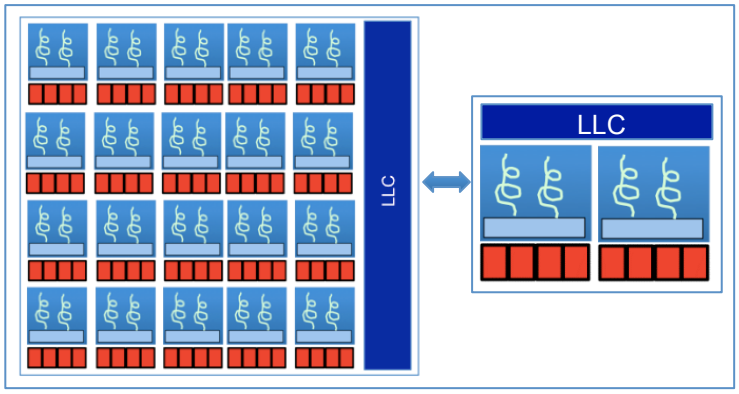
Hardware Diversity: Combining building blocks to construct nodes



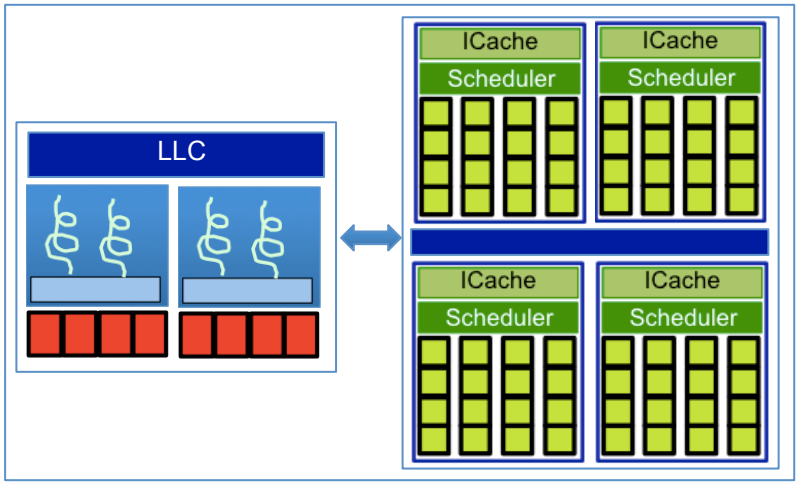
Multicore CPU



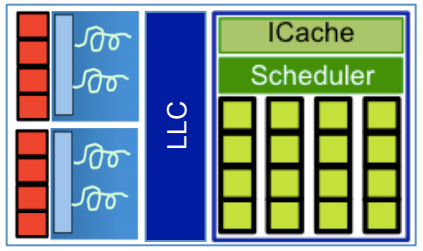
Manycore CPU



Heterogeneous:
CPU + manycore coprocessor

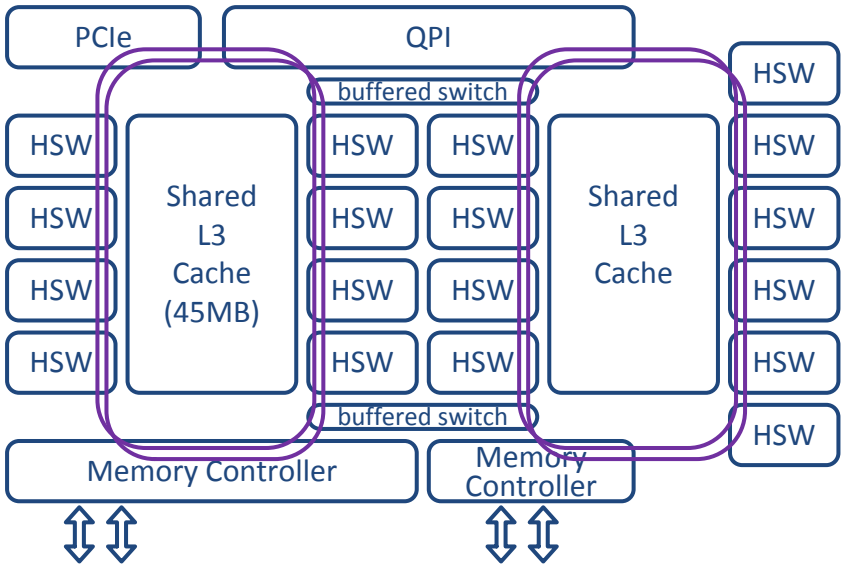


Heterogeneous: CPU+GPU



Heterogeneous:
Integrated CPU+GPU

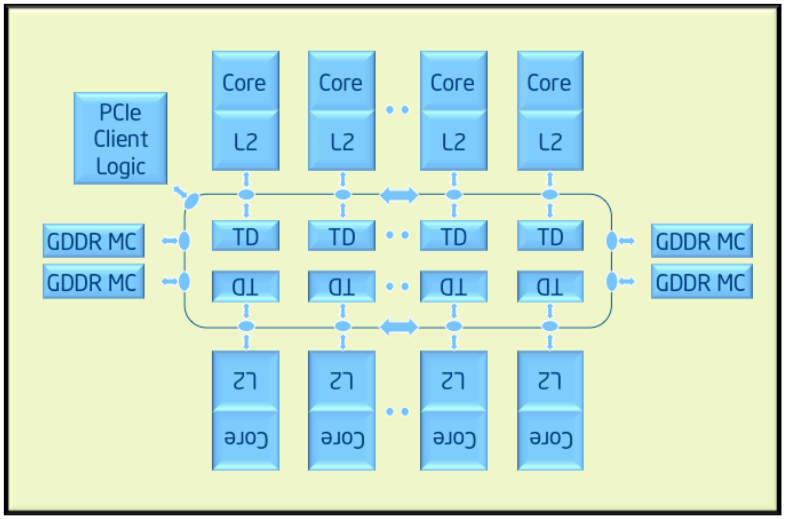
Hardware diversity: CPUs



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.

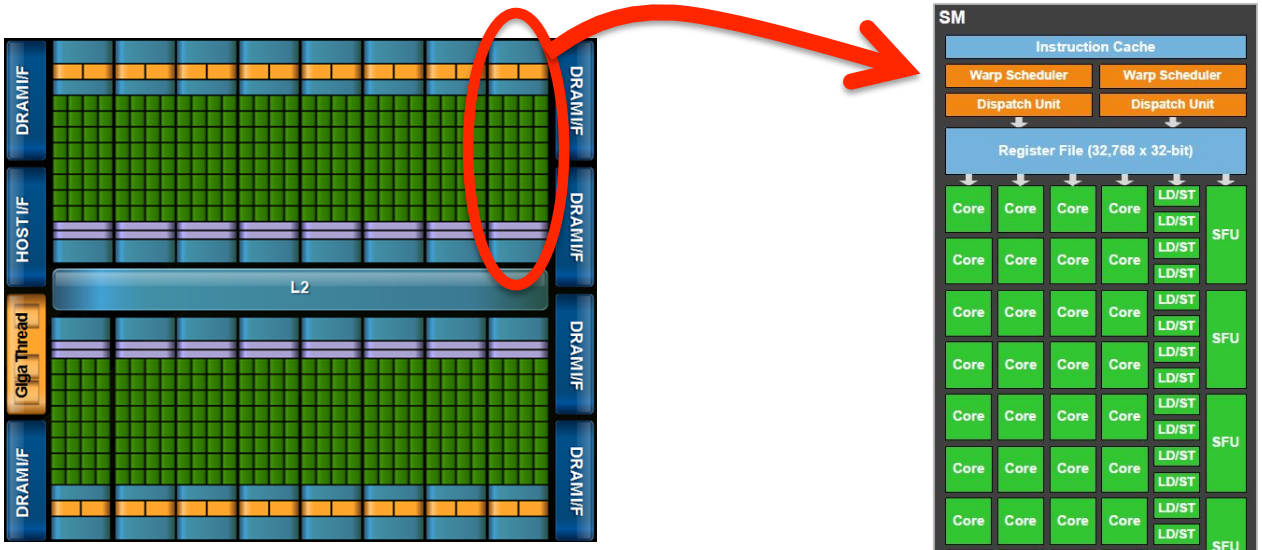


Intel® Xeon Phi™ coprocessor
(Knights Corner): Many core

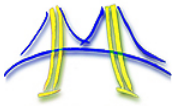
- 61 cores
- 244 Hardware threads
- 512 bit wide vector units

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#



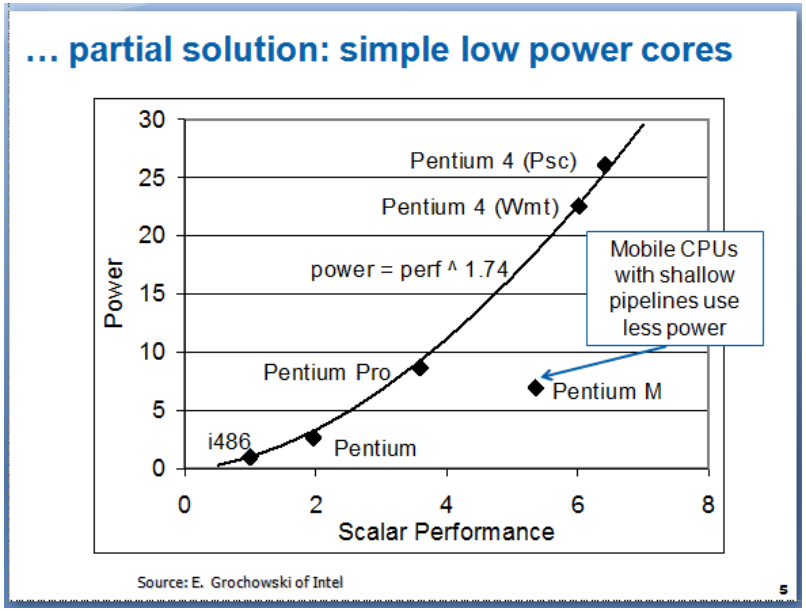
For example: an NVIDIA Tesla C2050 (Fermi) GPU with 3GB of memory and 14 streaming multiprocessor cores*.



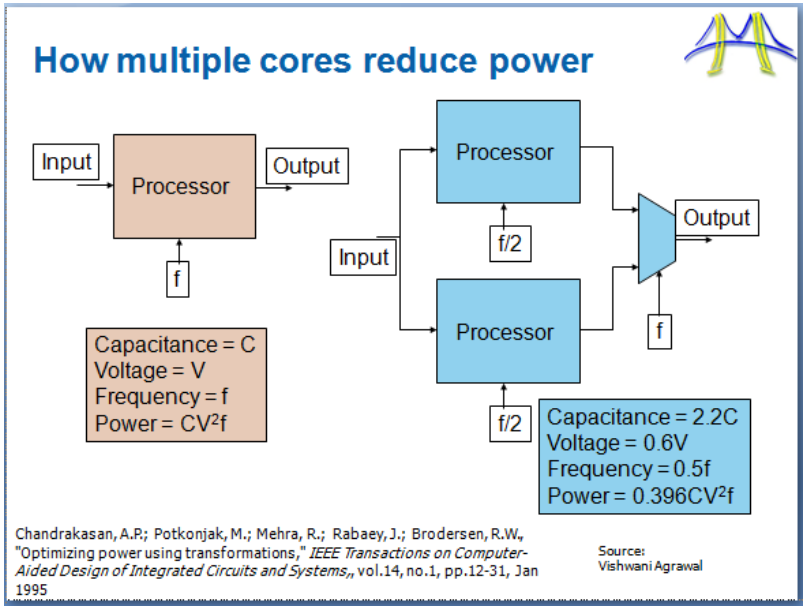
#Source: UC Berkeley, CS194, Fall’2014, Kurt Keutzer and Tim Mattson

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

The result…



+

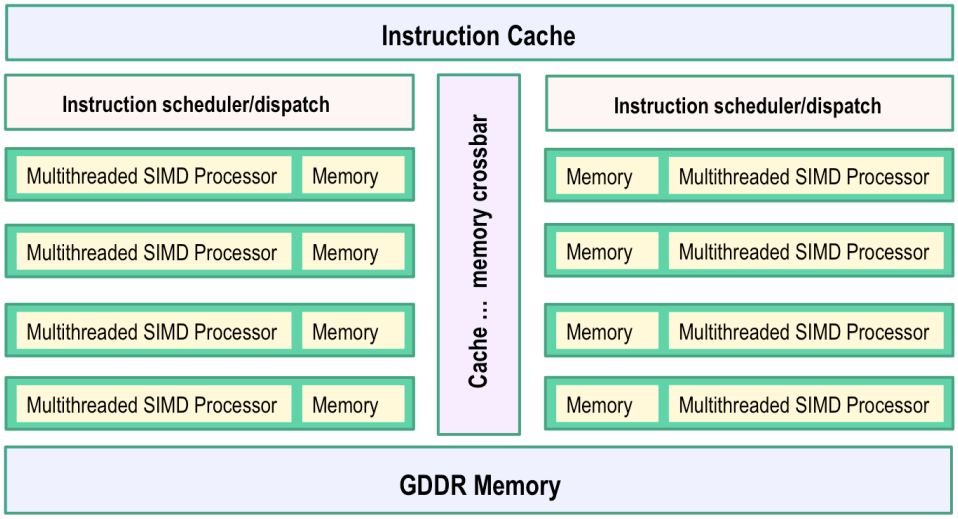


=

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)

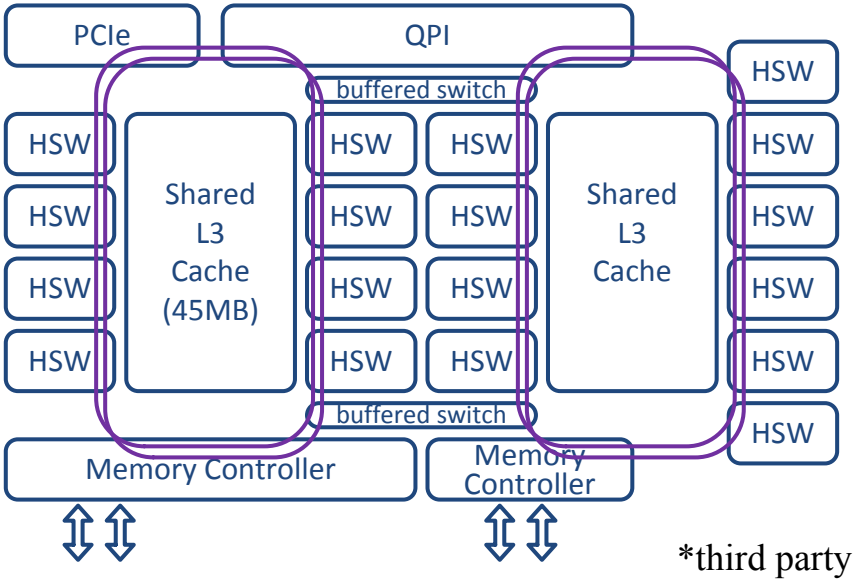
The problem is this was presented as an ultimatum … nobody asked us if we were OK with this new contract … which is kind of rude.

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



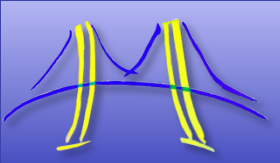
CPU

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

*third party names are the property of their owners

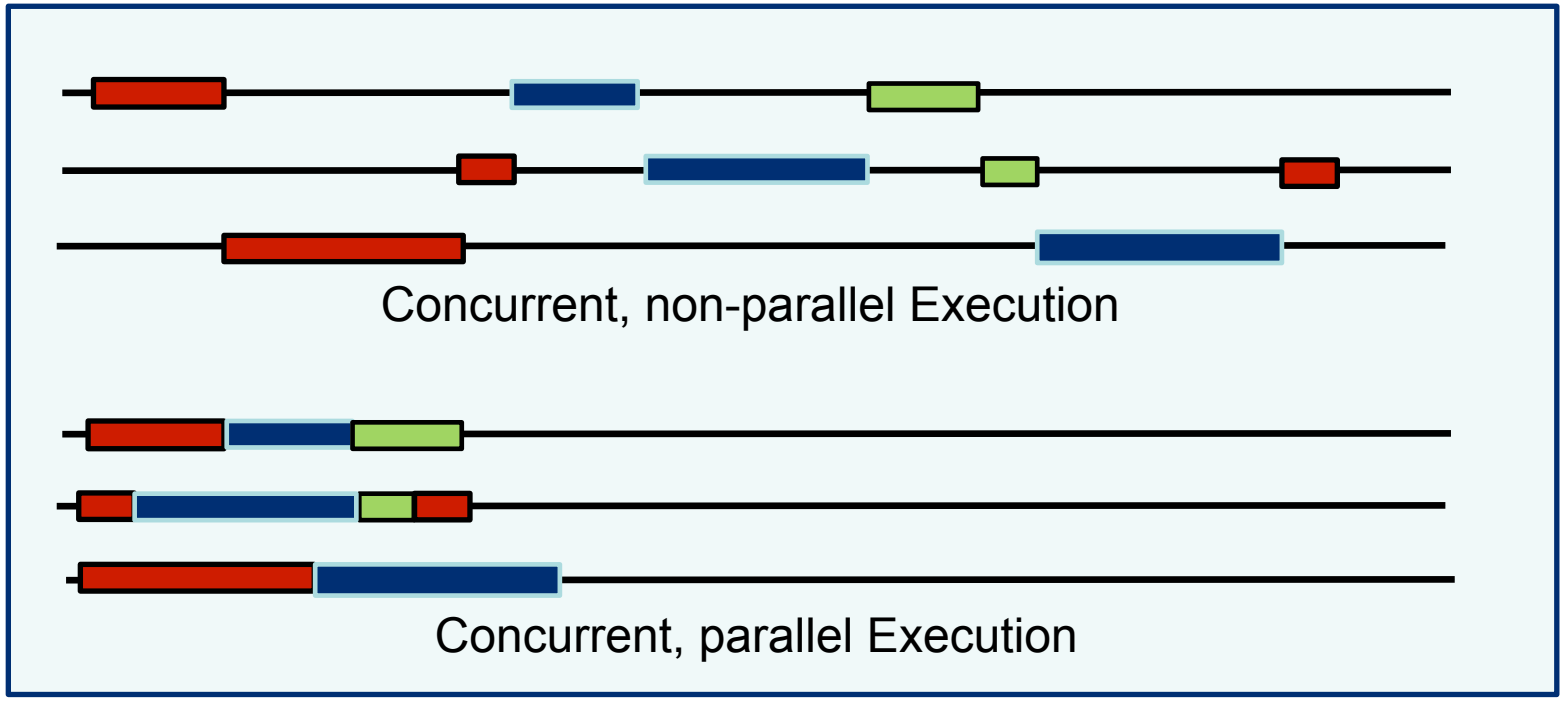
Outline

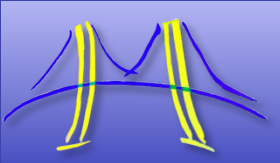
- High Performance computing: A hardware system view
- The processors in HPC systems
- Parallel Computing: Basic Concepts
- The Fundamental patterns of parallel Computing



Concurrency vs. Parallelism

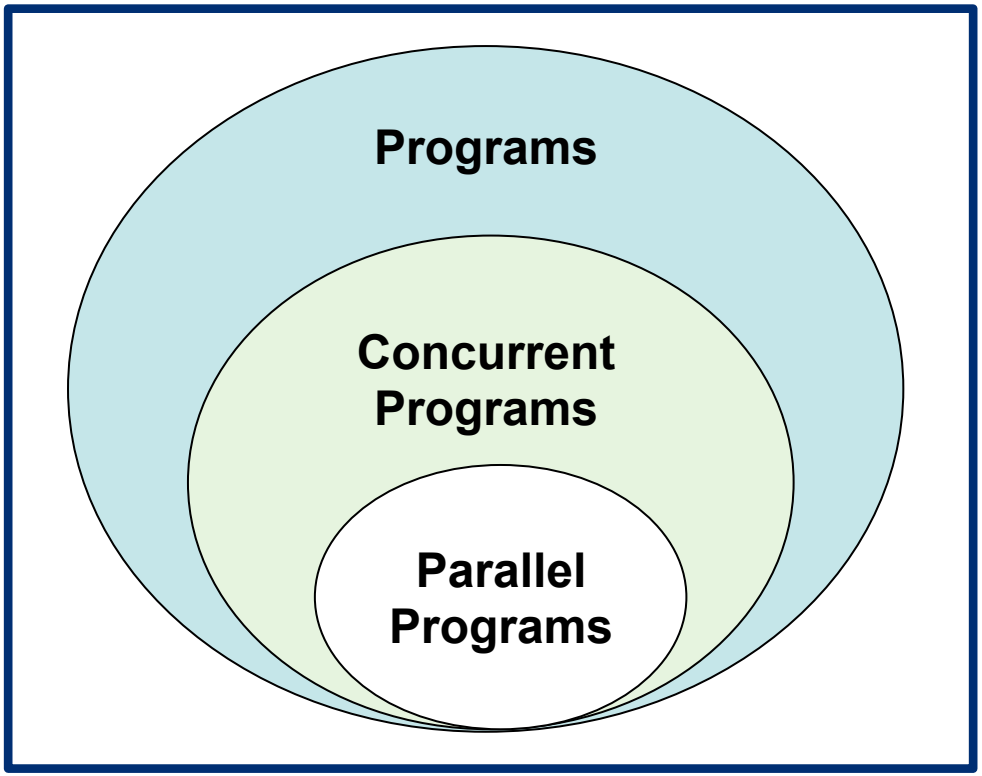
- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are ***logically*** active at one time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** active at one time.

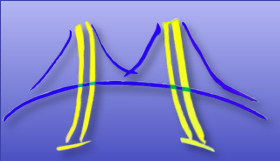




Concurrency vs. Parallelism

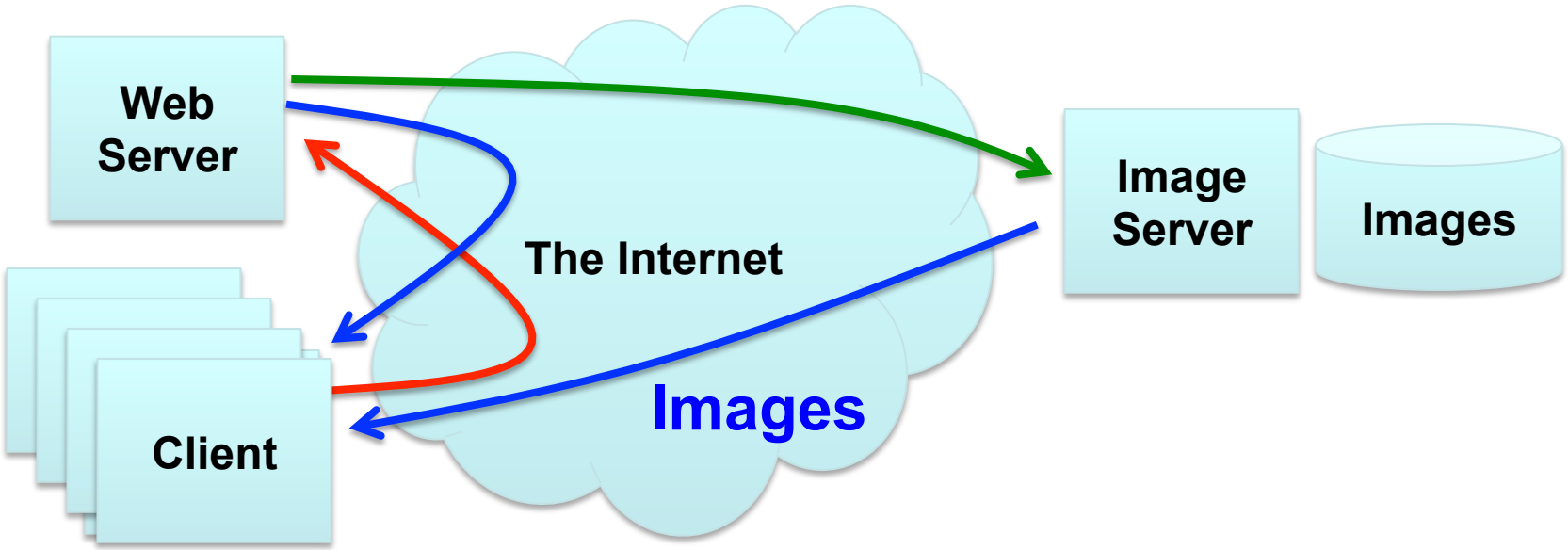
- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are ***logically*** active at one time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** active at one time.

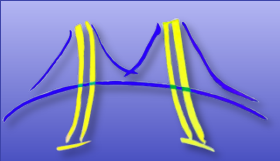




Concurrency in Action: a web server

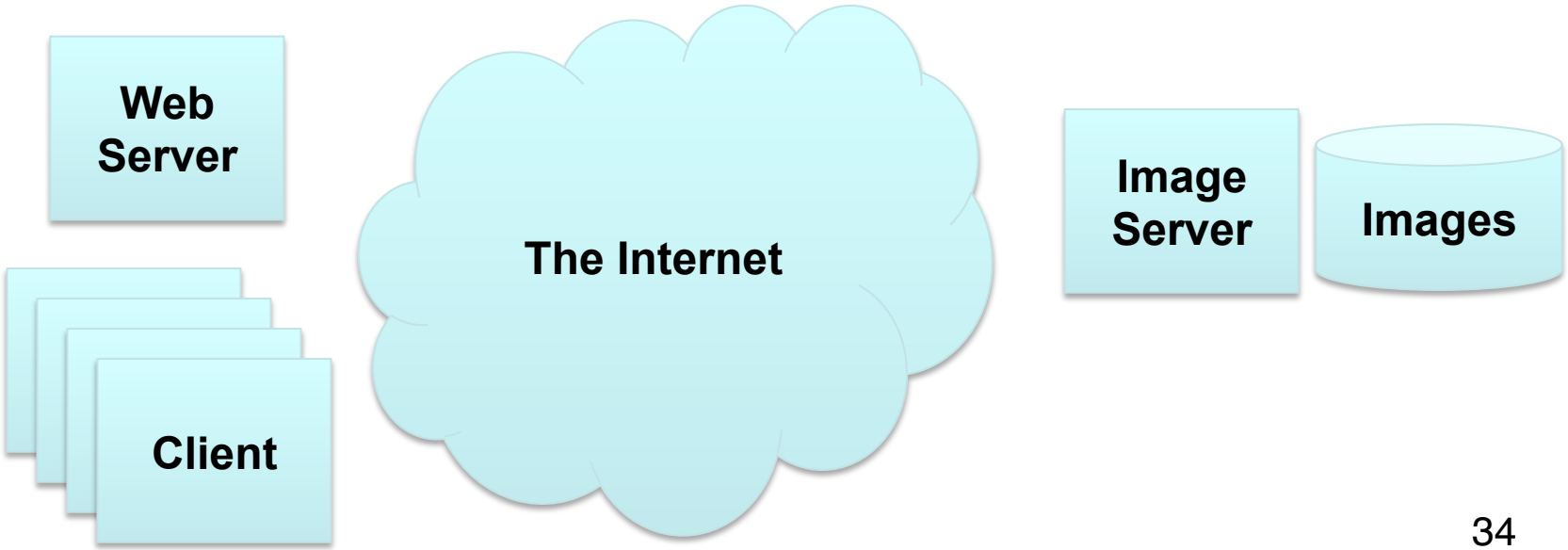
- A **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):
  - An arbitrary, large number of clients make requests which reference per-client persistent state
- Consider an **Image Server**, which relieves load on primary web servers by storing, processing, and serving only images

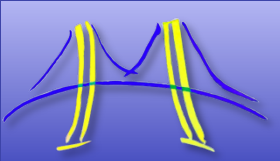




Concurrency in Action: a web server

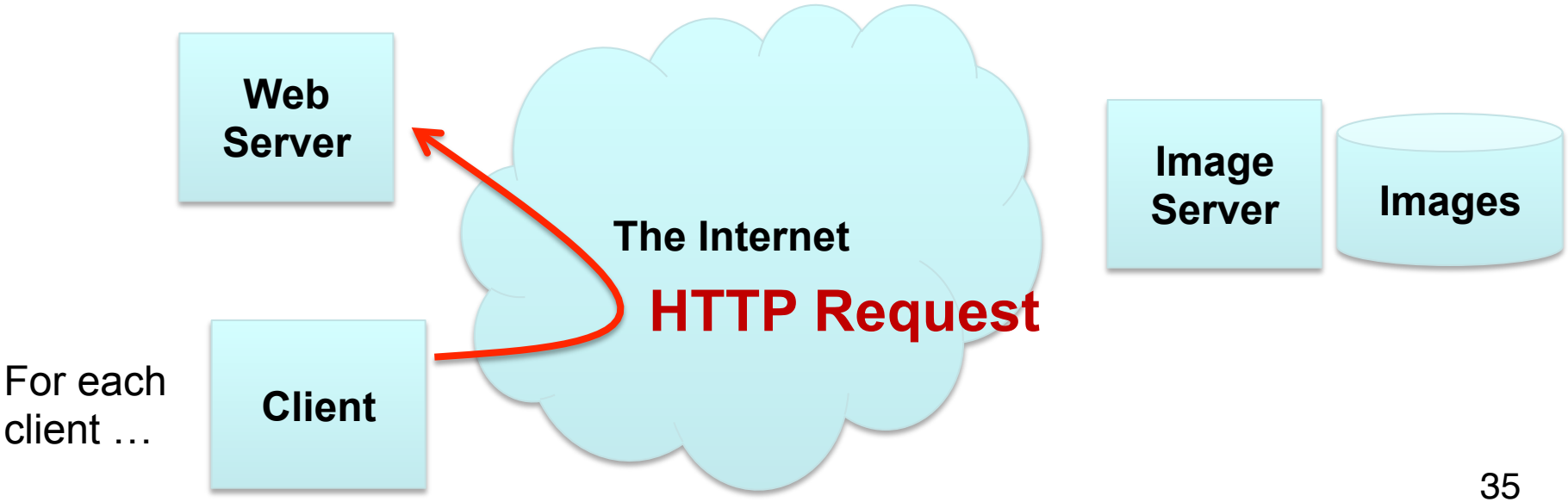
- An **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):
  - An arbitrary, large number of clients make requests which reference per-client persistent state
- Consider an **Image Server**, which relieves load on primary web servers by storing, processing, and serving only images

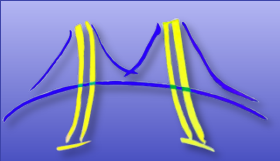




Concurrency in Action: a web server

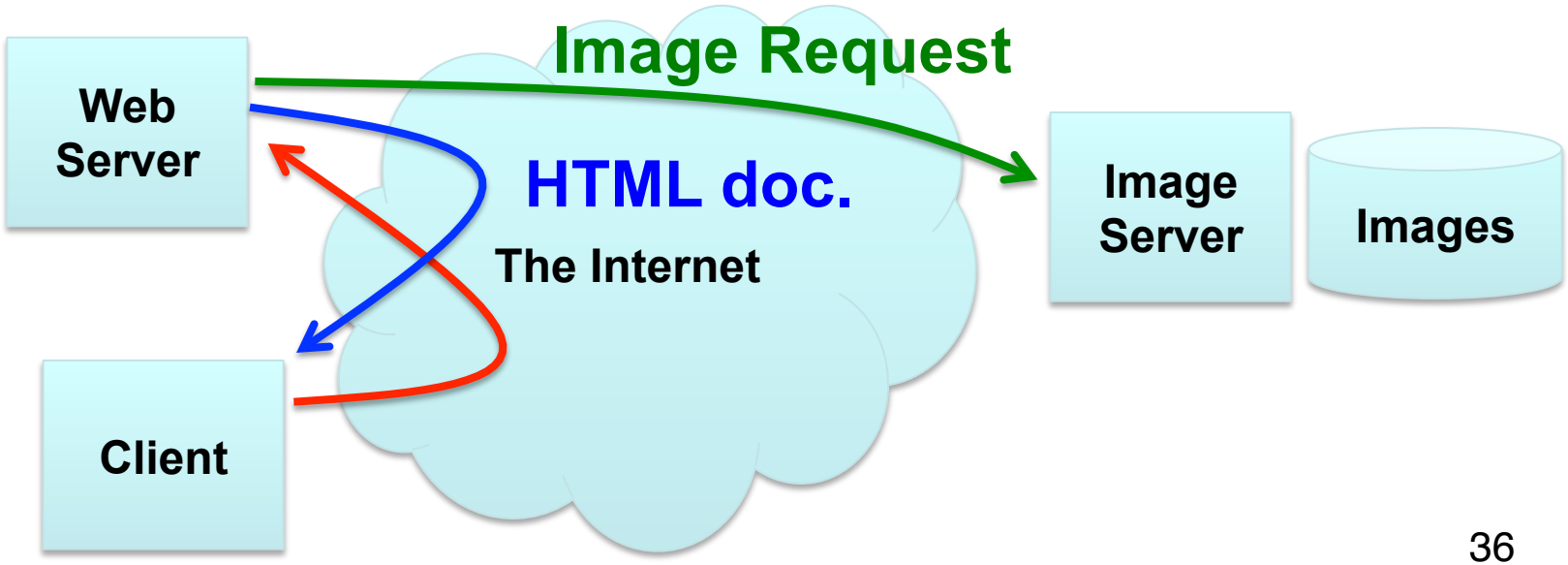
- A **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):
  - An arbitrary, large number of clients make requests which reference per-client persistent state
- Consider an **Image Server**, which relieves load on primary web servers by storing, processing, and serving only images

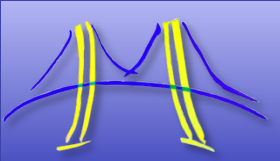




Concurrency in Action: a web server

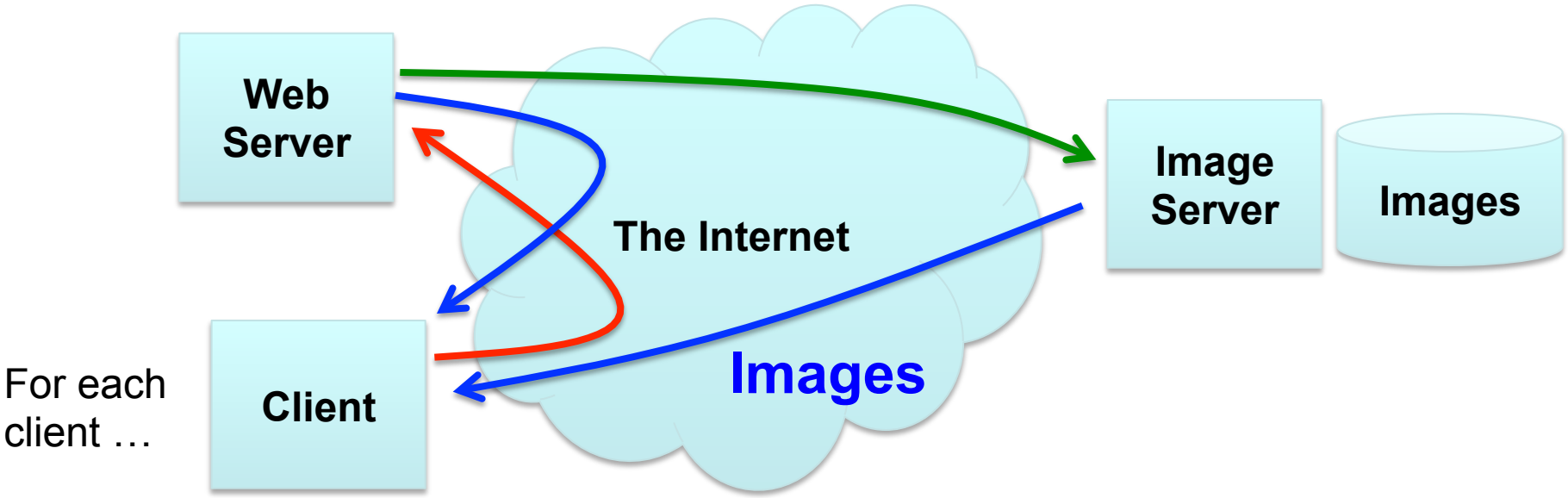
- A **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):
  - An arbitrary, large number of clients make requests which reference per-client persistent state
- Consider an **Image Server**, which relieves load on primary web servers by storing, processing, and serving only images

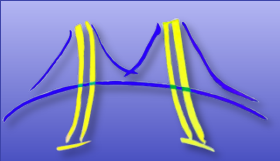




Concurrency in Action: a web server

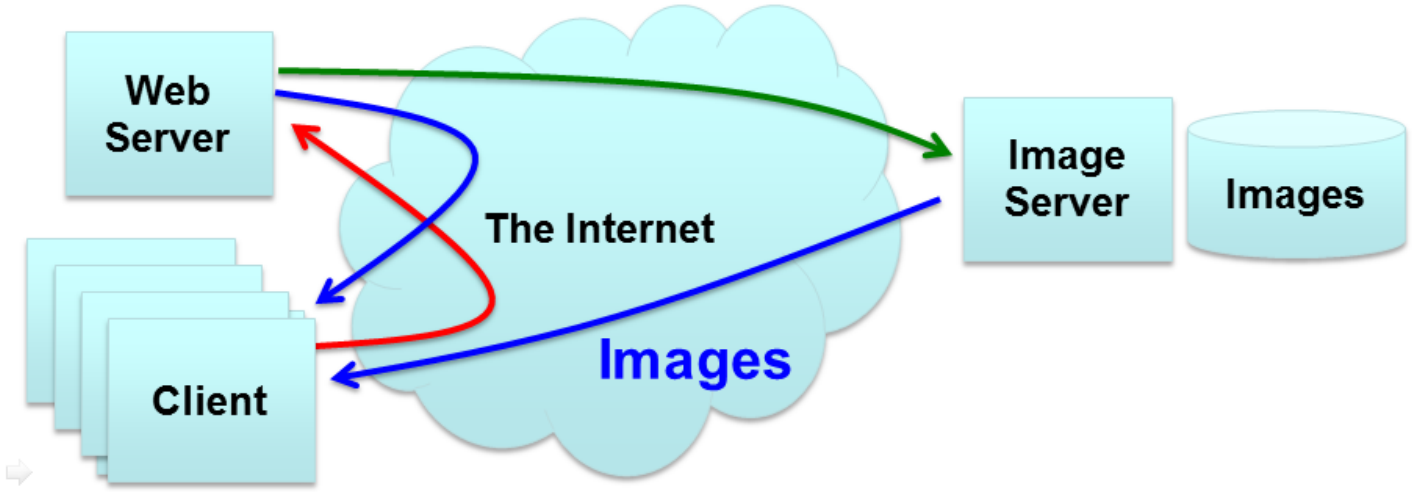
- A **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):
  - An arbitrary, large number of clients make requests which reference per-client persistent state
- Consider an **Image Server**, which relieves load on primary web servers by storing, processing, and serving only images





Concurrency in Action: a web server

- The Web server, image server, and clients (you have to plan on having many clients) all execute at the same time



- The problem of one or more clients interacting with a web server not only contains concurrency, the problem is fundamentally current. It doesn't exist as a serial problem.

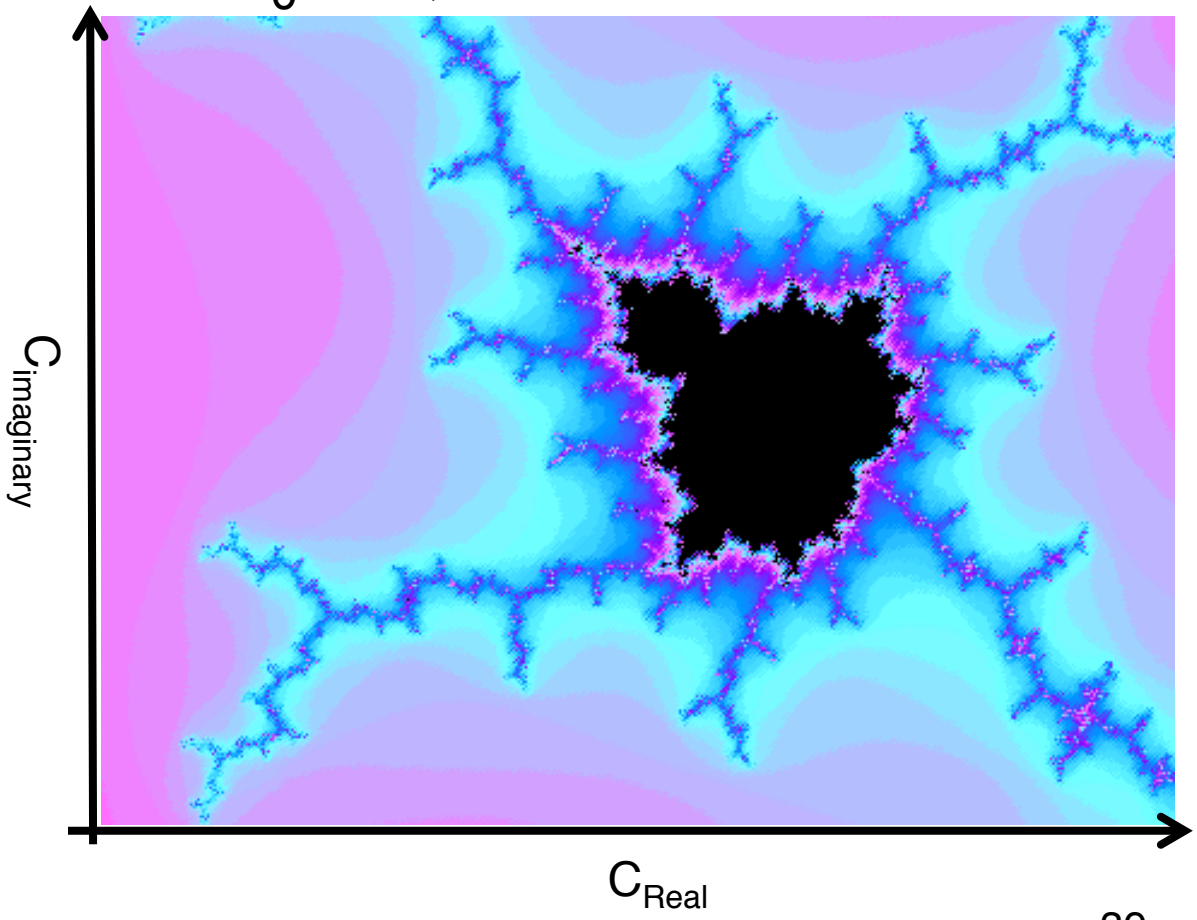
Concurrent application: An application for which the problem definition is fundamentally concurrent.

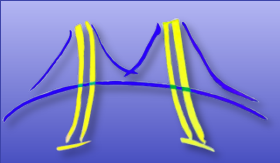
Concurrency in action: Mandelbrot Set

- The Mandelbrot set: An iterative map in the complex plane

$$z_{n+1} = z_n^2 + c \qquad z_0 = 0, \qquad c \text{ is constant}$$

- Color each point in the complex plain of C values based on convergence or divergence of the iterative map.





Concurrency in action: Mandelbrot Set

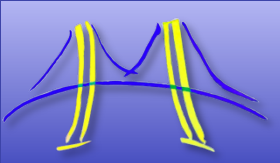
```
int mandel ( complex C) {
   int n;
   double a = C.real();
   double b = C.imag();
   double zr = 0.0 , zi = 0.0;
   double tzr , tzi ;
   n = 0;
   while (n < max_iters && sqrt (zr*zr + zi*zi) < t) {
      tzr = (zr*zr - zi*zi) + a;
      tzi = (zr*zi + zr*zi) + b;
      zr = tzr ;
      zi = tzi ;
      n = n+1;
   }
   return n;
}
```

Function to compute the iterative map for a single point C where

C = a + b * i

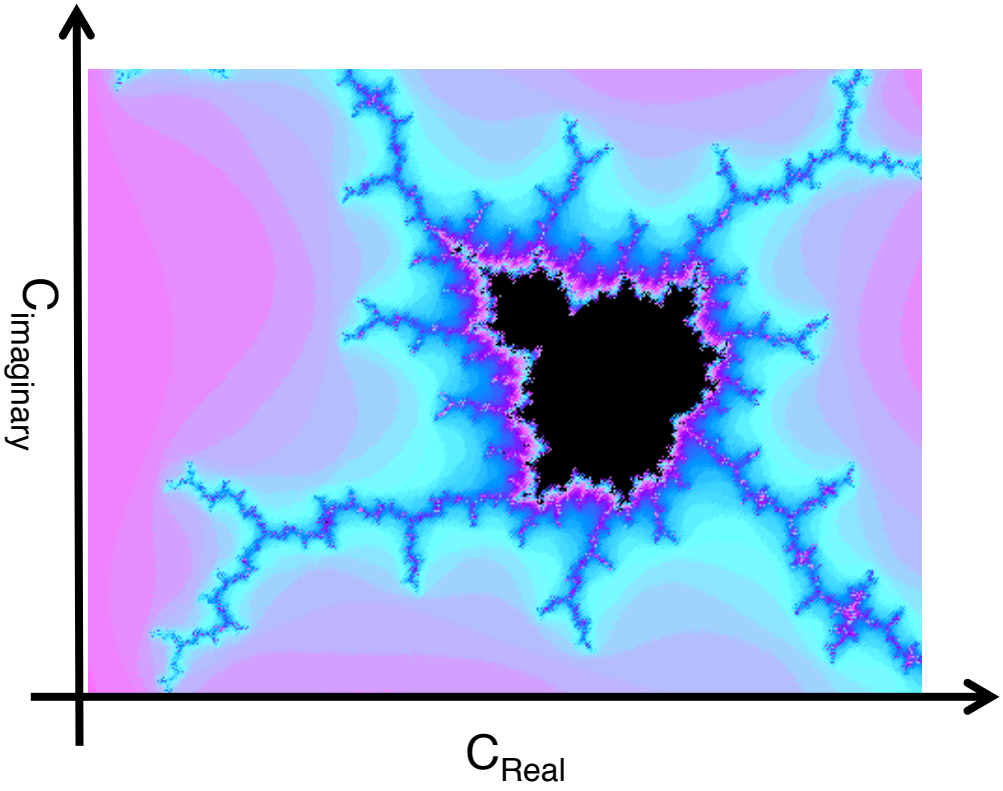
Where i is the square root of (-1)

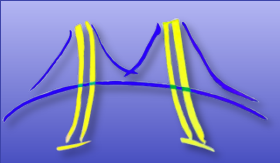
"t" is a constant that defines a threshold beyond which we consider the iterative map to diverge.



Concurrency in action: Mandelbrot Set

- To generate the famous Mandelbrot set image, we use the function mandel(C) where C comes from the points in the complex plane.
- At each point C, use n=mandel(C) to determine if:
  - The map converges (n=max_iters), assign the color black
  - The map diverges (n<max_iters), assign the color based on the value of n
- The computation for each point is independent of all the other points … a so-called *embarrassingly parallel* problem .

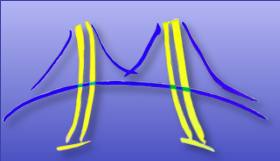




Concurrency in action: Mandelbrot Set

- The following is simplified code for the serial Mandelbrot program.

```
for (i=0; i<N; i++){
   for (j=0; j<N; j++) {
       complex c = get_const_at_pixel(i,j);
       complex image[i][j] = mandel( c);
   }
}
```

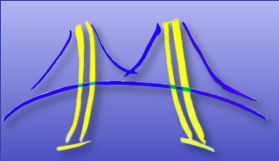


Concurrency in action: Mandelbrot Set

- The following is simplified code for the serial Mandelbrot program.
- Loop iterations are independent, so we can create a parallel version of this program as follows …

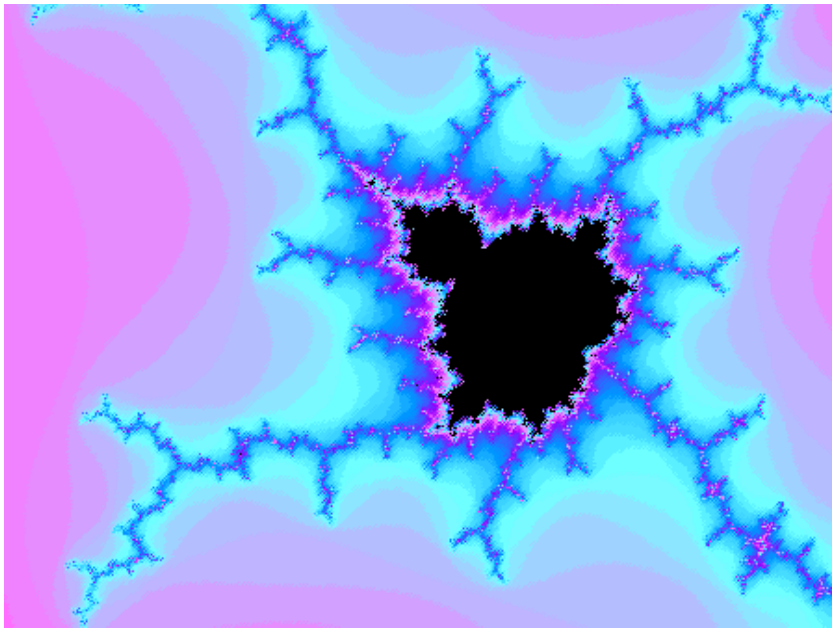
```
for (i=0; i<N; i++){
  for (j=0; j<N; j++) {
      complex c = get_const_at_pixel(i,j);
      complex image[i][j] = mandel( c);
   }
}
```

- Combine the two loops into one big loop and execute them in parallel

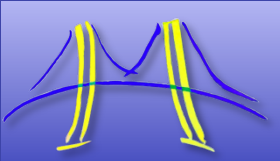


Concurrency in action: Mandelbrot Set

- The problem of generating an image of the Mandelbrot set *can be* viewed serially.
- We choose to exploit the concurrency contained in this problem so we can generate the image in less time

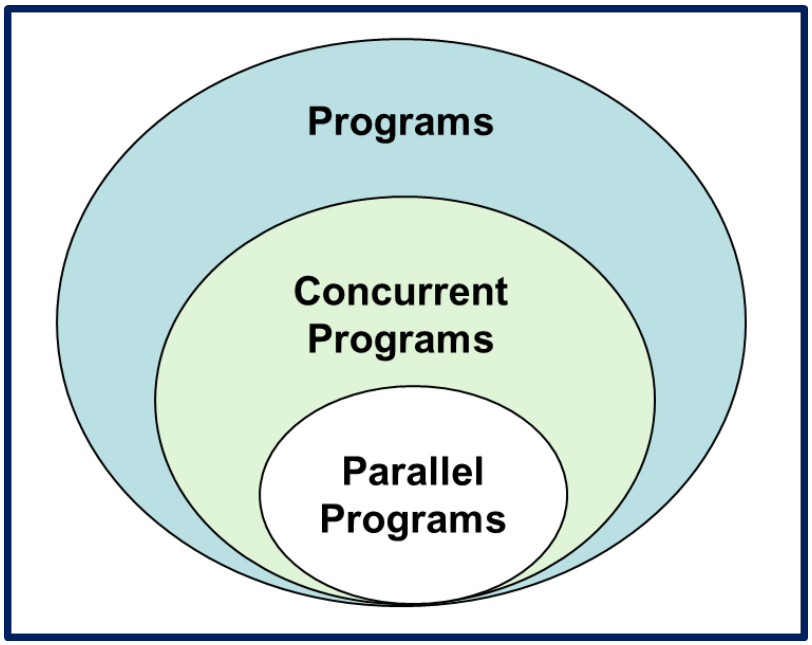


Parallel application: An application composed of tasks that actually execute concurrently in order to (1) consider larger problems in fixed time or (2) complete in less time for a fixed size problem.



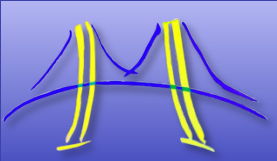
Concurrency vs. Parallelism: wrap up

- Key points:
  - A web server had concurrency in its problem definition … it doesn't make sense to even think of writing a "serial web server".
  - The Mandelbrot program didn't have concurrency in its problem definition. It would take a long time, but it could be serial

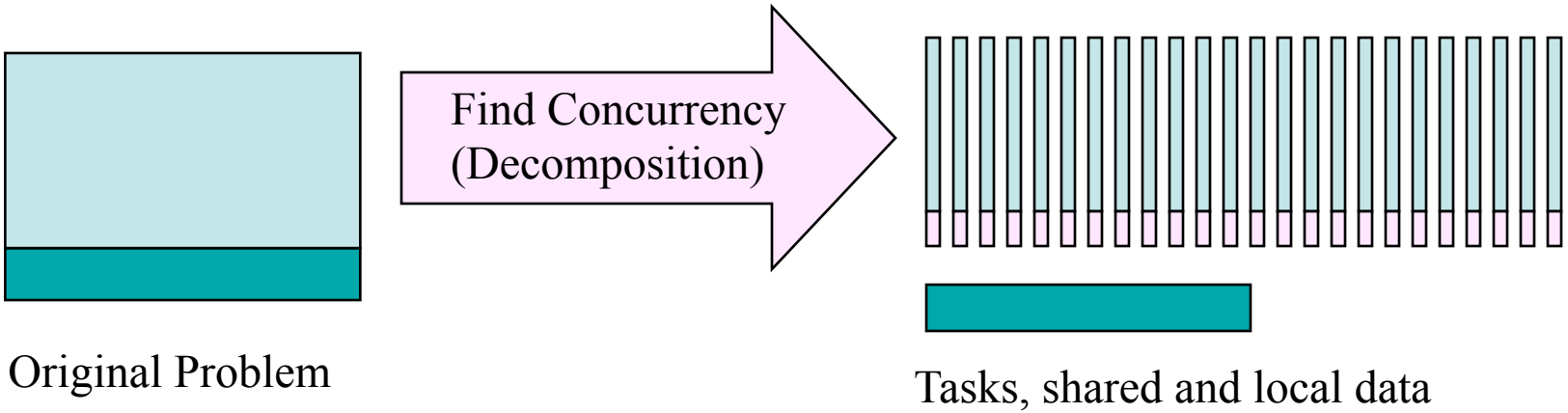


- Both cases use concurrency:
  - A concurrent application is concurrent by definition.
  - A parallel application solves a problem that could be serial, but it is run in parallel by …
    1. find concurrency in the problem
    2. expose the concurrency in the source code.
    3. exploit the exposed concurrency to complete a job in less time.

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

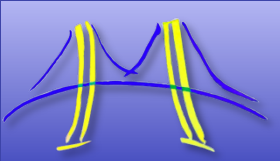


The Parallel programming process:



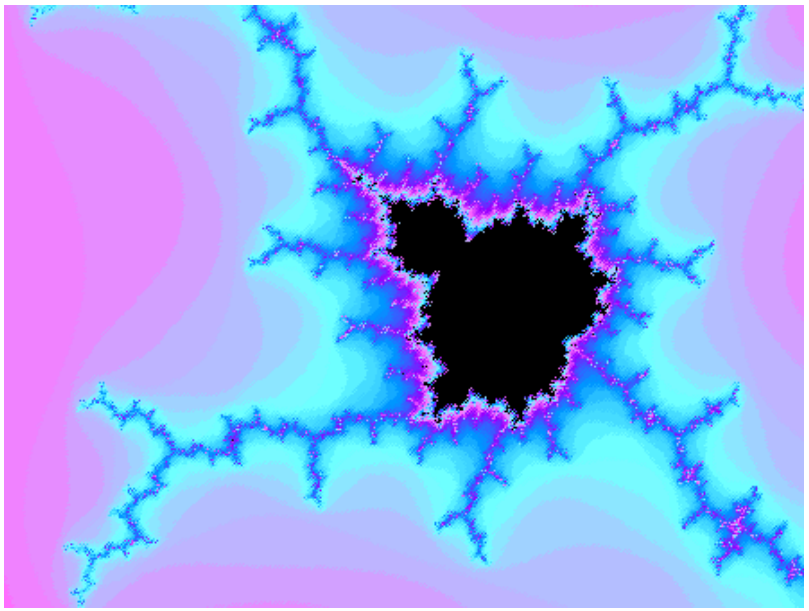
Original Problem

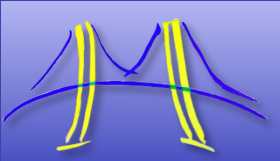
Tasks, shared and local data



Decomposition in parallel programs

- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..
  - **Data decomposition**: How must the data be broken down into chunks and associated with threads/processes to make the parallel program run efficiently.

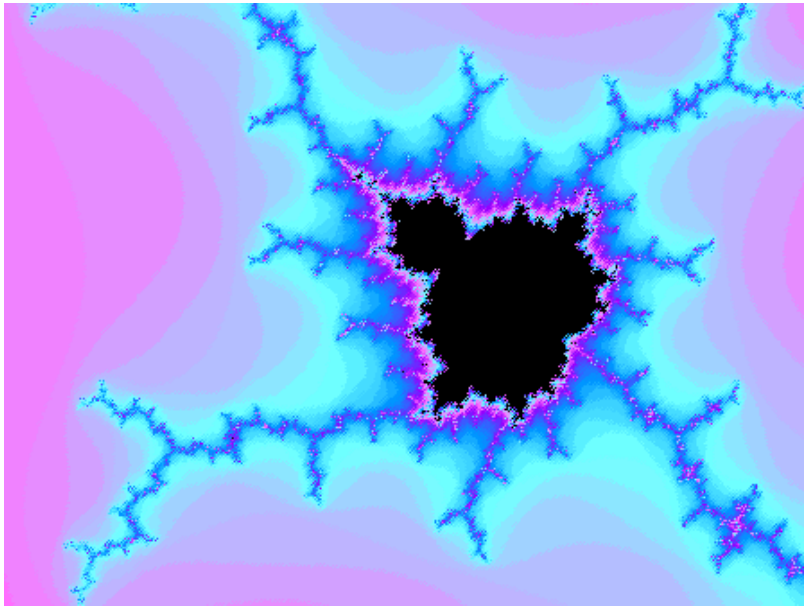


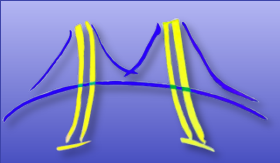


Decomposition in parallel programs

- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..
  - **Data decomposition**: How must the data be broken down into chunks and associated with threads/processes to make the parallel program run efficiently.

What's a task decomposition for this problem?

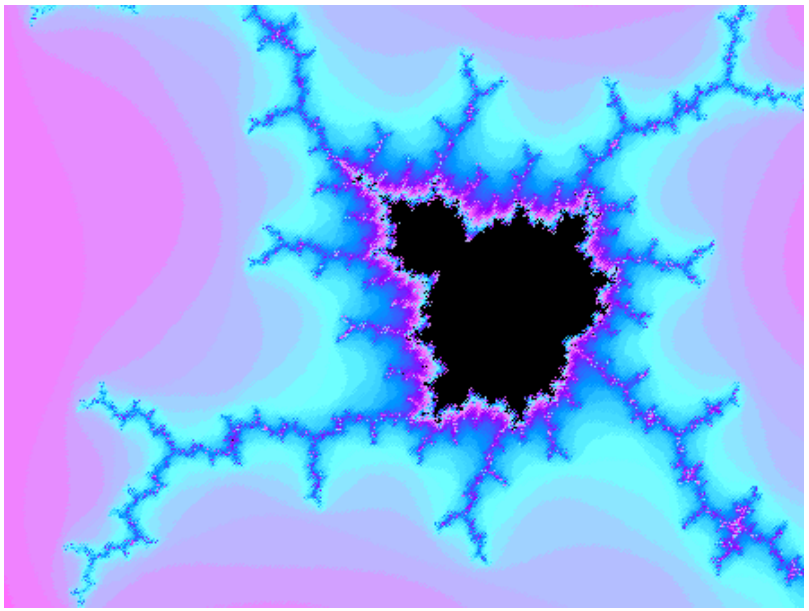




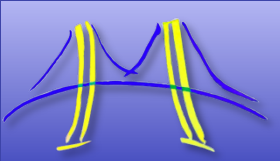
Decomposition in parallel programs

- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute

Hint: Think of the source code and work that is compute-intensive that can execute independently



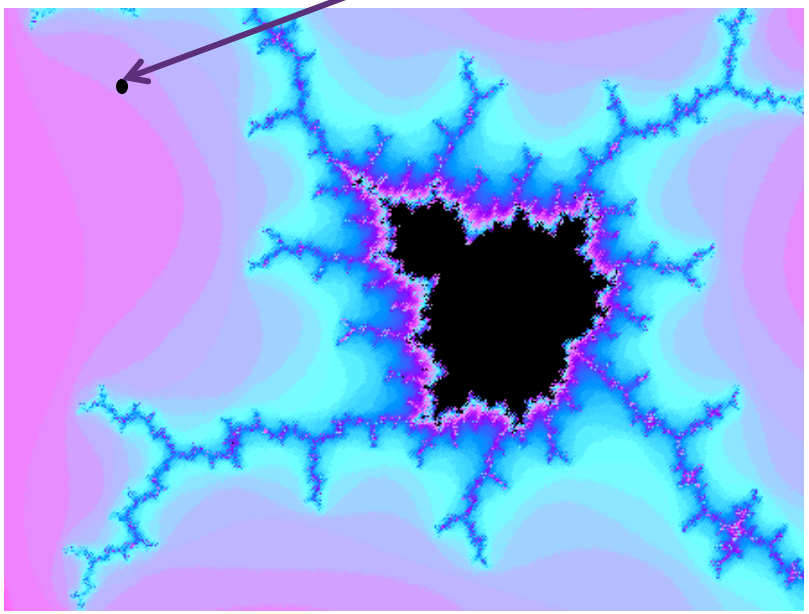
```
for (i=0; i<N; i++){
   for (j=0; j<N; j++) {
      complex c = get_const_at_pixel(i,j);
      complex image[i][j] = mandel( c);
   }
}
```



Decomposition in parallel programs

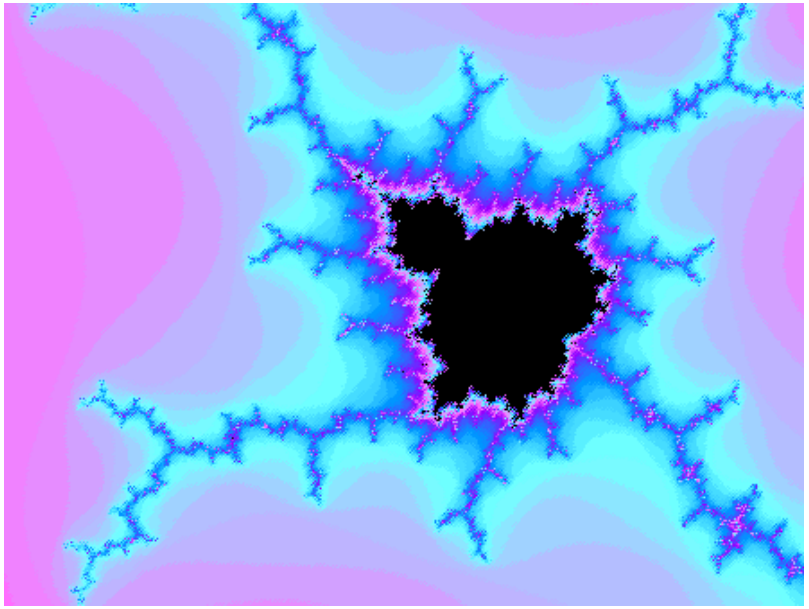
- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..
  - **Data decomposition**: How must the data be broken down into chunks and associated with threads/ processes to make the parallel program run efficiently.

Task: the computation required for each pixel … the body of the loop for a pair (i,j).

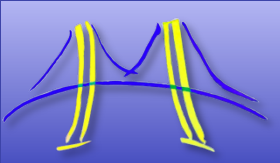


Decomposition in parallel programs

- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..
  - **Data decomposition**: How must the data (the complex plain, C) be broken down into chunks and associated with threads/ processes to make the parallel program run efficiently.



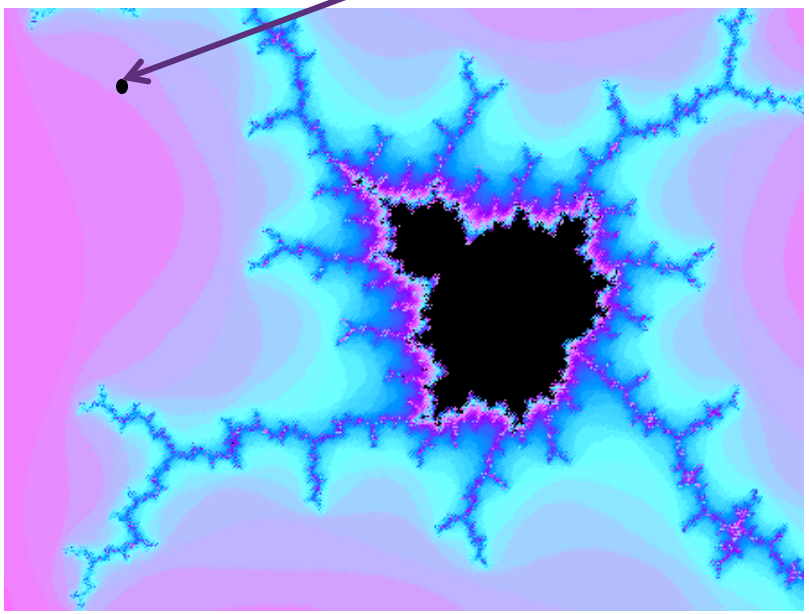
Suggest a data decomposition for this problem … assume a quad core shared memory PC.



Decomposition in parallel programs

- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..
  - **Data decomposition**: How must the data (the complex plain, C) be broken down into chunks and associated with threads/ processes to make the parallel program run efficiently.
  - .

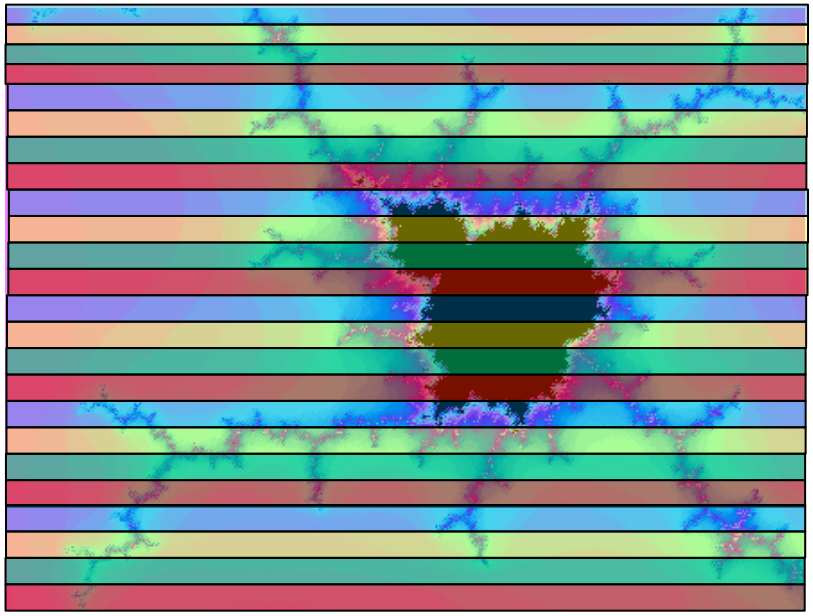
Task: the computation required for each pixel … the body of the loop for a pair (i,j).



Hint: you can define the data decomposition to match the task, but would that be efficient in this case?

Decomposition in parallel programs

- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..
  - **Data decomposition**: How must the data (the complex plain, C) be broken down into chunks and associated with threads/processes to make the parallel program run efficiently.

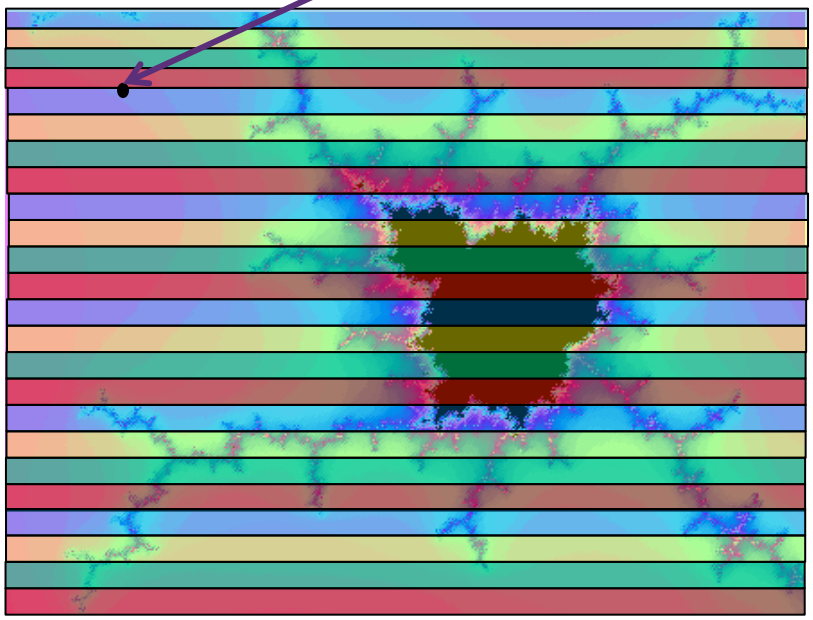


Map the pixels into row blocks and deal them out to the cores. This will give each core a memory efficient block to work on.

Decomposition in parallel programs

- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..
  - **Data decomposition**: How must the data (the complex plain, C) be broken down into chunks and associated with threads/processes to make the parallel program run efficiently.

But given this data decomposition, it is effective to think of a task as the update to a pixel? Should we update our task definition given the data decomposition?

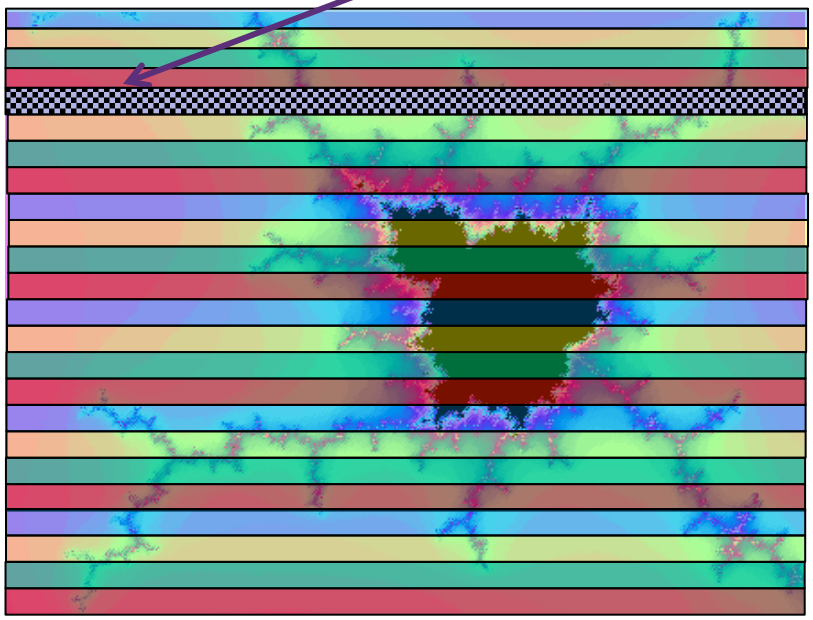


Map the pixels into row blocks and deal them out to the cores. This will give each core a memory efficient block to work on.

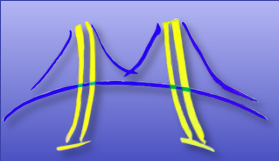
Decomposition in parallel programs

- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..
  - **Data decomposition**: How must the data (the complex plain, C) be broken down into chunks and associated with threads/processes to make the parallel program run efficiently.

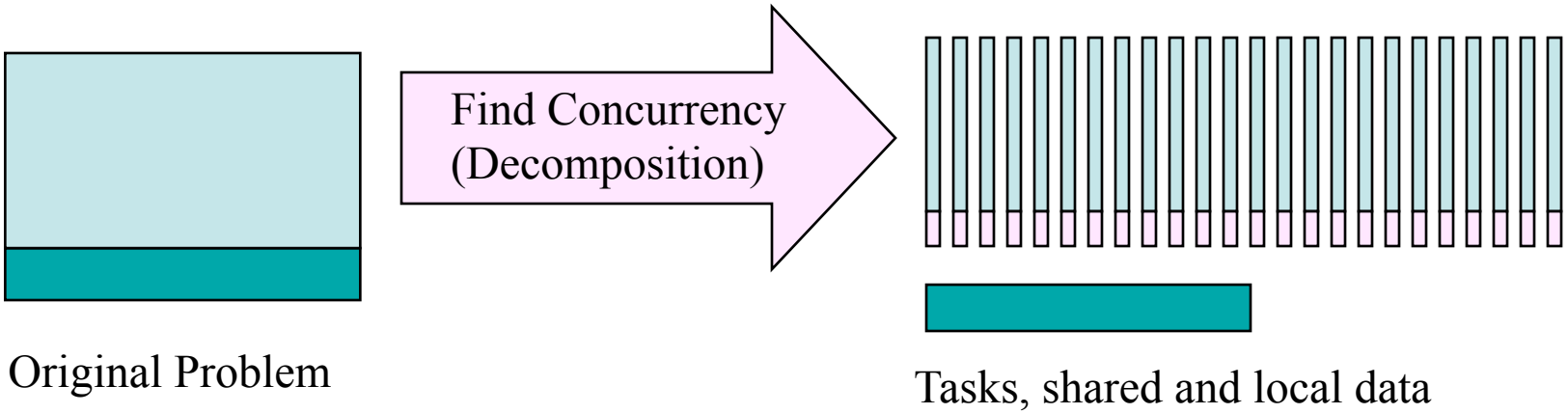
Yes. You go back and forth between task and data decomposition until you have a pair that work well together. In this case, let's define a task as the update to a row-block



Map the pixels into row blocks and deal them out to the cores. This will give each core a memory efficient block to work on.



Recap:



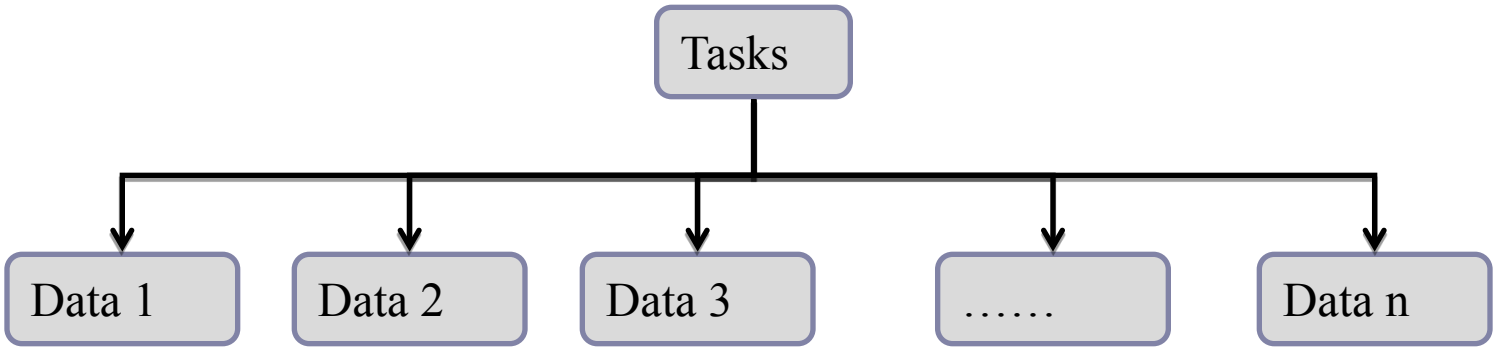
- To expose concurrency in a problem, we need to understand how the problem is decomposed into tasks AND how the problem's data is decomposed to support efficient computation.
- ***YOU ALWAYS NEED BOTH.***

Outline

- High Performance computing: A hardware system view
- The processors in HPC systems
- Parallel Computing: Basic Concepts
- The Fundamental patterns of parallel Computing

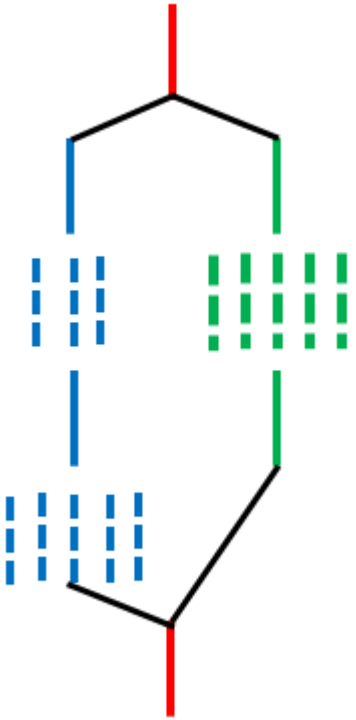
Data Parallelism Pattern

- Use when:
  - Your problem is defined in terms of *independent* collections of data elements operated on by a similar (if not identical) sequence of instructions; i.e. the concurrency is in the data.
    - *Hint: when the data decomposition dominates your design, this is probably the pattern to use!*
- Solution
  - Define collections of data elements that can be updated in parallel.
  - Define computation as a sequence of collective operations applied together to each data element.



Task Parallelism Pattern

- Use when:
  - The problem naturally decomposes into a distinct collection of tasks
    - Hint: when the task decomposition dominates you design, this is probably the pattern to use.



- Solution
  - Define the set of tasks and a way to detect when the computation is done.
  - Manage (or “remove”) dependencies so the correct answer is produced regardless of the details of how the tasks execute.
  - Schedule the tasks for execution in a way that keeps the work balanced between the processing elements of the parallel computer and

Task Parallelism in practice

- Embarrassingly parallel:
  - The tasks are independent, so the parallelism is “so easy to exploit it’s embarrassing”.
- Separable dependencies:
  - Turn a problem with dependent tasks into an “embarrassingly parallel” by “replicating data between tasks, doing the work, then recombining data (often a reduction) to restore global state.
- Functional Decomposition
  - A task is associated with a functional decomposition of the problem to produce a coarse grained parallel program

Its becoming common to associate this case as the prototypical “task parallel” approach … but to us old-timers, the previous two cases are overwhelming more common.

Fundamental Design Patterns:

- Data Parallelism:
  - Kernel Parallelism
  - Geometric Decomposition
  - Loop parallel
- Task Parallelism
  - Task queue
  - Divide and Conquer
  - Loop parallel
- Implementation Patterns (used to support the above)
  - SPMD (Any MIMD machine, but typically distributed memory)
  - Fork Join (Multithreading, shared address space MIMD)
  - Kernel Parallelism (GPGPU)

SPMD: Single Program Mulitple Data

- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

Loop parallelism

- Collections of tasks are defined as iterations of one or more loops.
- Transform loop body to remove loop carried dependencies and assure that the right answer is produced regardless of the order iterations are carried out.
- Loop iterations are divided between a collection of processing elements to compute tasks concurrently.

```
#pragma  omp parallel for shared(Results) schedule(dynamic)
For(i=0;i<N;i++){
        Do_work(i, Results);
}
```

This design pattern is also heavily used with data parallel design patterns. OpenMP programmers commonly use this pattern.

Kernel Parallelism

- Kernel Parallelism:
  - Implement data parallel problems:
    - Define an abstract index space that spans the problem domain.
    - Data structures in the problem are aligned to this index space.
    - Tasks (e.g. work-items in OpenCL or “threads” in CUDA) operate on these data structures for each point in the index space.
- This approach was popularized for graphics applications where the index space mapped onto the pixels in an image. Since 2006, It’s been extended to General Purpose GPU (GPGPU) programming.

Note: This is basically a fine grained extreme form of the SPMD pattern.

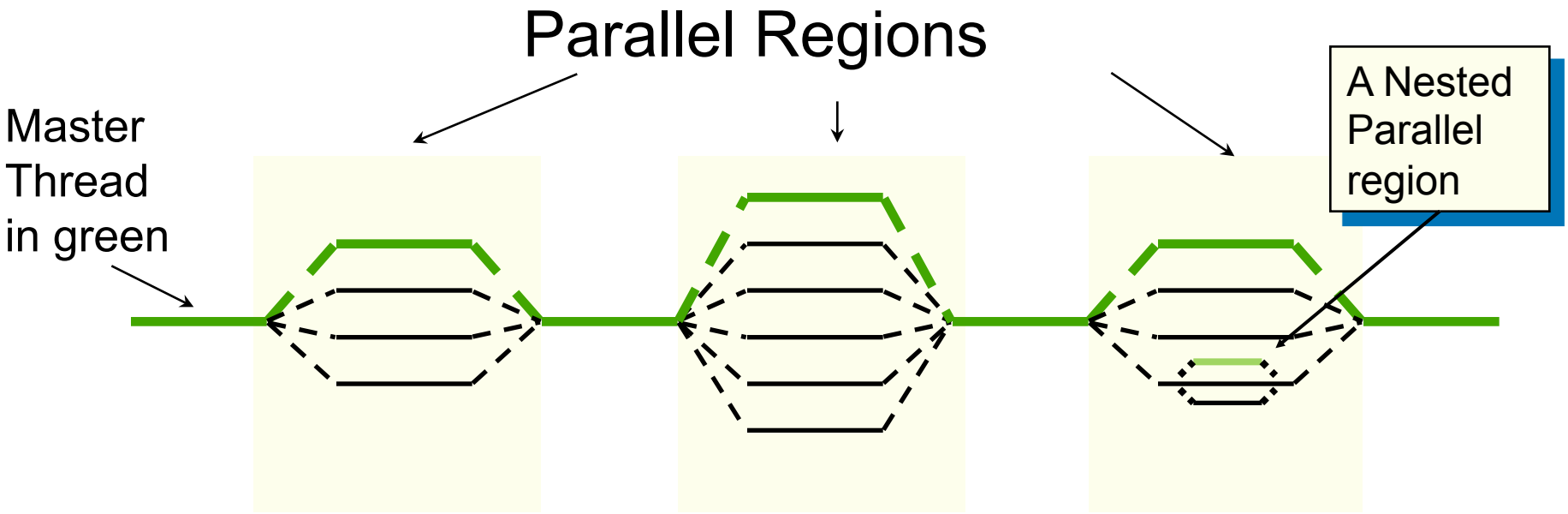
Fork-join

- Use when:
  - Target platform has a shared address space
  - Dynamic task parallelism
- Particularly useful when you have a serial program to transform incrementally into a parallel program
- Solution:
  1. A computation begins and ends as a single thread.
  2. When concurrent tasks are desired, additional threads are forked.
  3. The thread carries out the indicated task,
  4. The set of threads recombine (join)

Cilk and OpenMP 3.0 with explicit tasks make heavy use of this pattern.

Fork-join Design Pattern

- Use when:
  - Target platform has a shared address space
  - Dynamic task parallelism
- Particularly useful when incrementally parallelizing a serial program.
- Solution:
  1. A computation begins and ends as a single thread.
  2. When concurrent tasks are desired, additional threads are forked.
  3. The thread carries out the indicated task,
  4. The set of threads recombine (join)

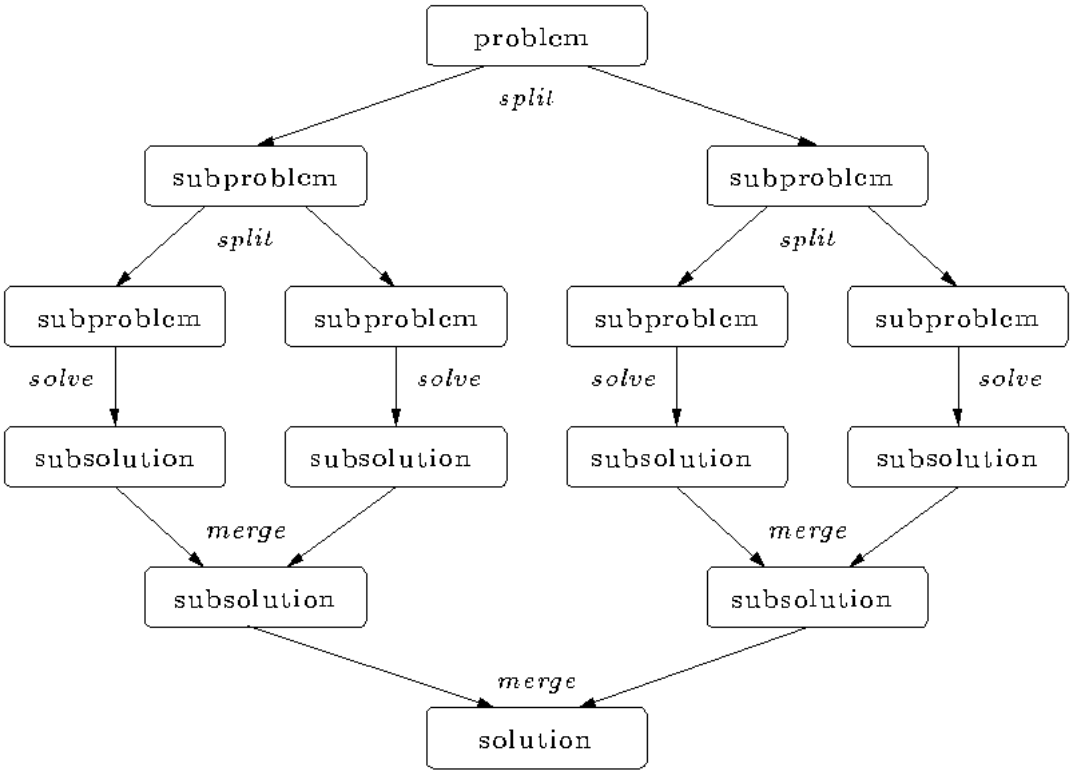


Divide and Conquer Pattern

- Use when:
  - A problem includes a method to divide into subproblems and a way to recombine solutions of subproblems into a global solution.
- Solution
  - Define a split operation
  - Continue to split the problem until subproblems are small enough to solve directly.
  - Recombine solutions to subproblems to solve original global problem.
- Note:
  - Computing may occur at each phase (split, leaves, recombine).

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.

Task Queue

- Use When
  - The computation is defined as a collection of independent tasks.
  - The key challenge is how to schedule them such that the computational load is evenly distributed.
- Solution:
  - Set up collection of workers to carry out the work.
  - Put tasks into a shared queue
  - Workers grab tasks, carry out computations, then return to queue for more work.
  - There must be some way to detect when the work is done and then shut down the workers.

This is an easy way to implement automatic load balancing … so easy that for years we've been telling people to use this pattern whenever they can.

Geometric Decomposition

- Use when:
  - The problem is organized around a central data structure that can be decomposed into smaller segments (chunks) that can be updated concurrently.
- Solution
  - Typically, the data structure is updated iteratively where a new value for one chunk depends on neighboring chunks.
  - The computation breaks down into three components: (1) exchange boundary data, (2) update the interiors or each chunk, and (3) update boundary regions. The optimal size of the chunks is dictated by the properties of the memory hierarchy.
- Note:
  - This pattern is often used with the Structured Mesh and linear algebra computational strategy pattern.

Summary

Processors with lots of cores/vector-units/SIMT connected into clusters are here to stay. You have no choice … embrace parallel computing!

Outline

- High Performance computing: A hardware system view
- The processors in HPC systems
- Parallel Computing: Basic Concepts
- The Fundamental patterns of parallel Computing

Summary

Processors with lots of cores/vector-units/SIMT connected into clusters are here to stay. You have no choice … embrace parallel computing!

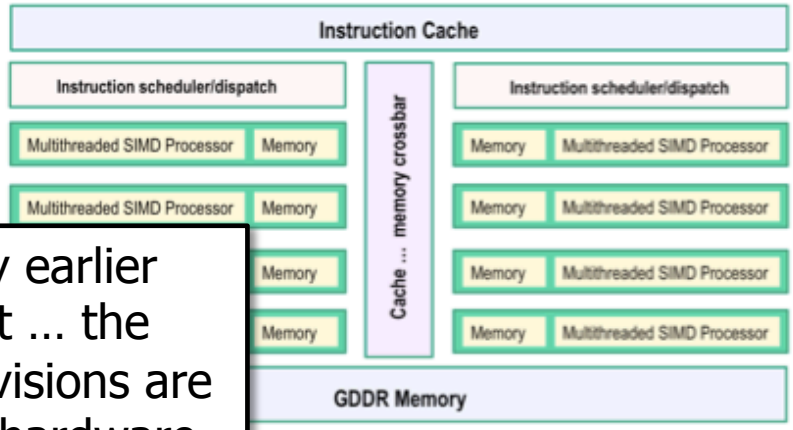
Outline

- High Performance computing: A hardware system view
- The processors in HPC systems
- Parallel Computing: Basic Concepts
- The Fundamental patterns of parallel Computing

Protect your software investment … refuse to use any programming model that locks you to a vendors platform. Open Standards are the ONLY rational approach in the long run.

The software platform debate!

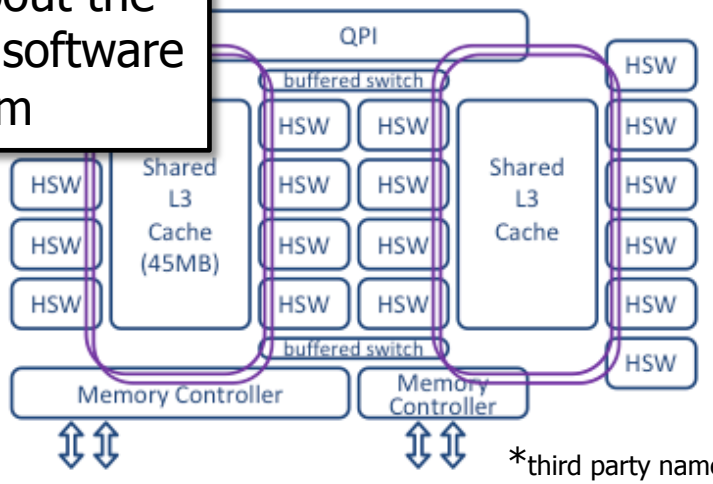
It's really about competing software platforms



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

Recall my earlier comment … the competing visions are less about hardware and more about the programmers software platform

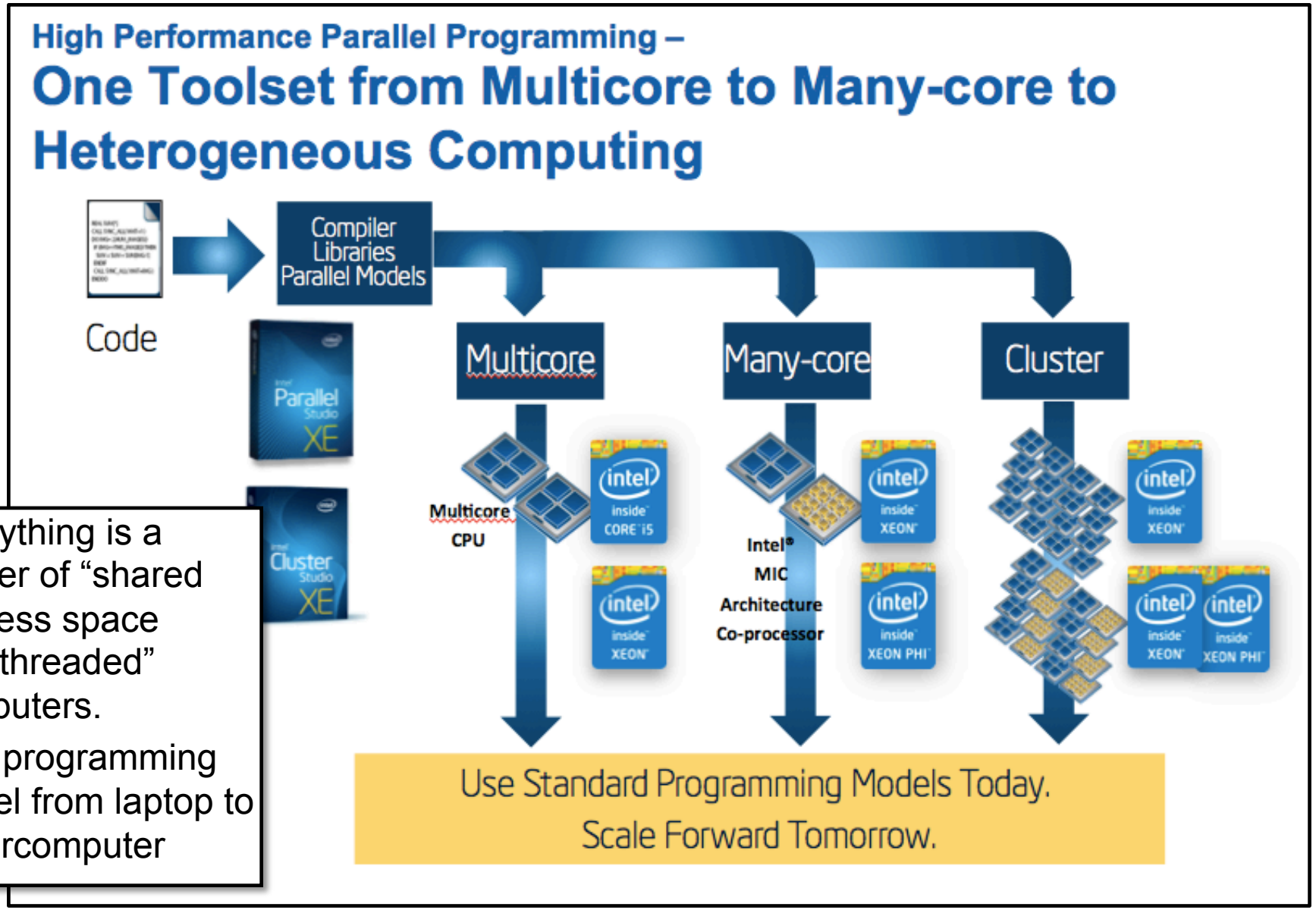


CPU

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

*third party names are the property of their owners

The official strategy at Intel.



- Everything is a cluster of "shared address space multithreaded" computers.
- One programming model from laptop to supercomputer

What about the competing platform:
Single Instruction multiple thread (SIMT)?

- Dominant as a proprietary solution based on CUDA and OpenACC.
- But there is an Open Standard response (supported to varying degrees by all major vendors)

OpenCL

SIMT programming for CPUs, GPUs, DSPs, and FPGAs. Basically, an Open Standard that generalizes the SIMT platform pioneered by our friends at NVIDIA®

OpenMP™

OpenMP 4.0 added target and device directives … Based on the same work that was used to create OpenACC. Therefore, just like OpenACC, you can program a GPU with OpenMP!!!

The long term viability of the SIMT platform depends on the user community demanding (and using) the Open Standard alternatives!

*third party names are the property of their owners

Summary

Processors with lots of cores/vector-units/SIMT connected into clusters are here to stay. You have no choice … embrace parallel computing!

Outline

- High Performance computing: A hardware system view
- The processors in HPC systems
- Parallel Computing: Basic Concepts
- The Fundamental patterns of parallel Computing

Protect your software investment … refuse to use any programming model that locks you to a vendors platform. Open Standards are the ONLY rational approach in the long run.

OpenMP, MPI, and OpenCL are in most cases, the way to go

Summary

Processors with lots of cores/vector-units/SIMT connected into clusters are here to stay. You have no choice … embrace parallel computing!

Outline

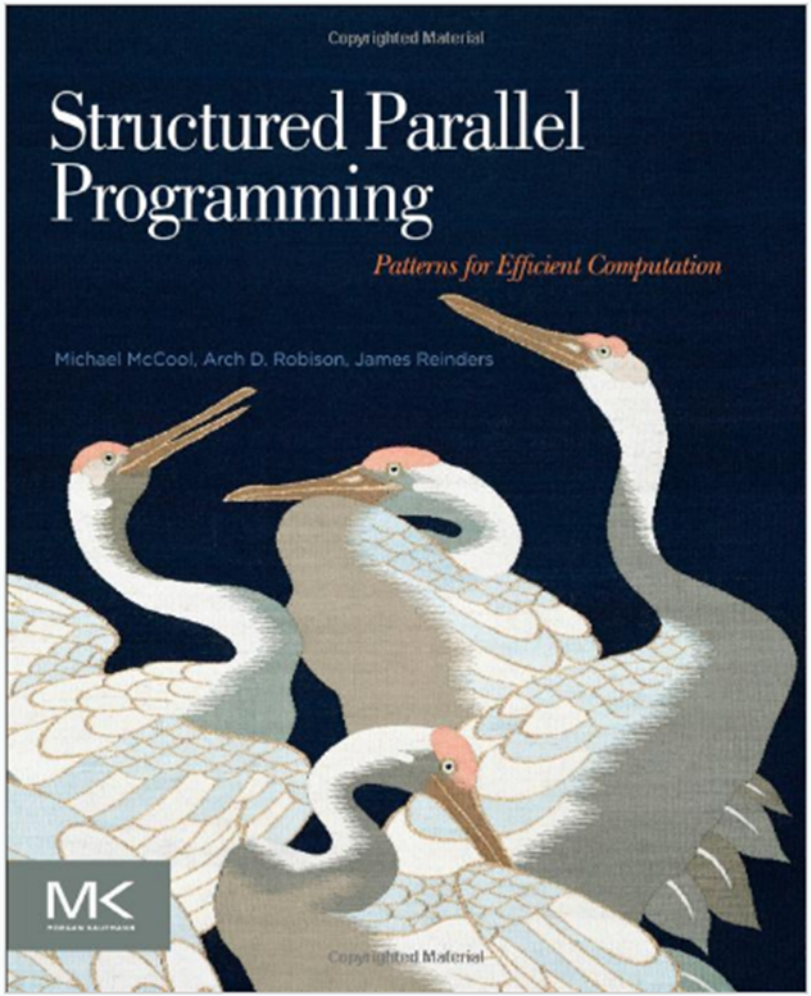
- High Performance computing: A hardware system view
- The processors in HPC systems
- Parallel Computing: Basic Concepts
- The Fundamental patterns of parallel Computing

Protect your software investment … refuse to use any programming model that locks you to a vendors platform. Open Standards are the ONLY rational approach in the long run.

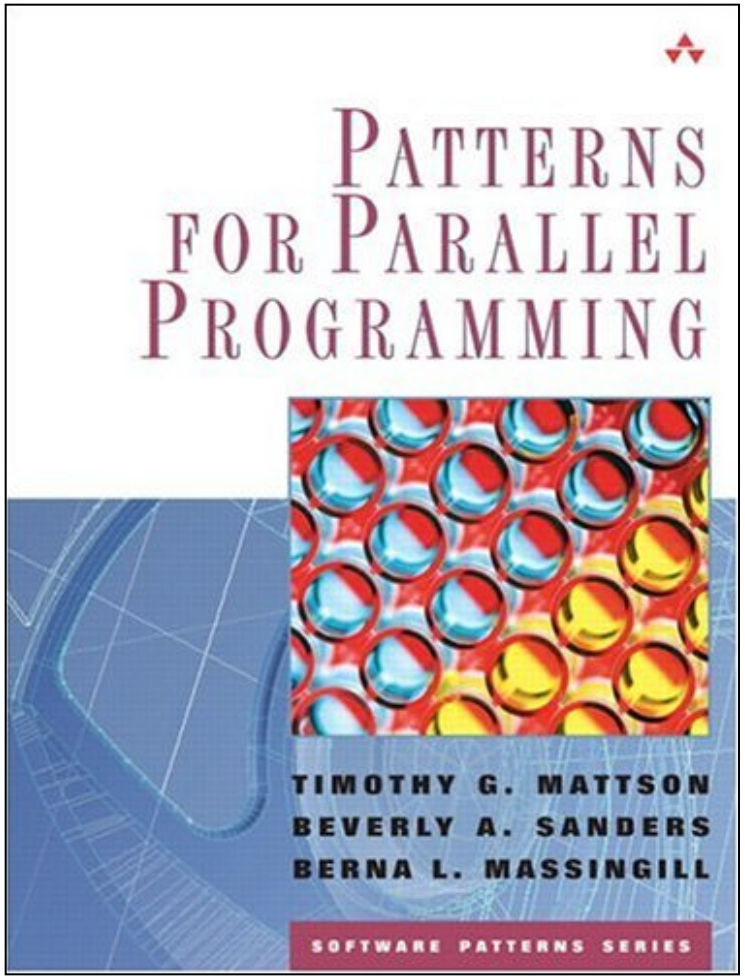
OpenMP, MPI, and OpenCL are in most cases, the way to go

There are countless parallel algorithms … but they make use of a small number of design patterns. Focus on the patterns and how to apply them in your programming models of choice and you'll be OK.

Background references



A great book that explores key patterns with Cilk, TBB, OpenCL, and OpenMP (by McCool, Robison, and Reinders)



- A book about how to "think parallel" with examples in OpenMP, MPI and java