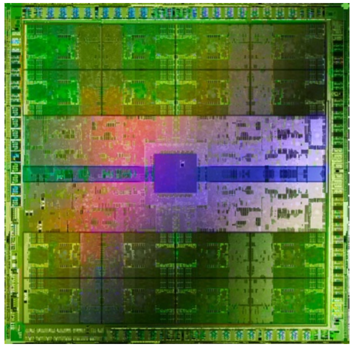
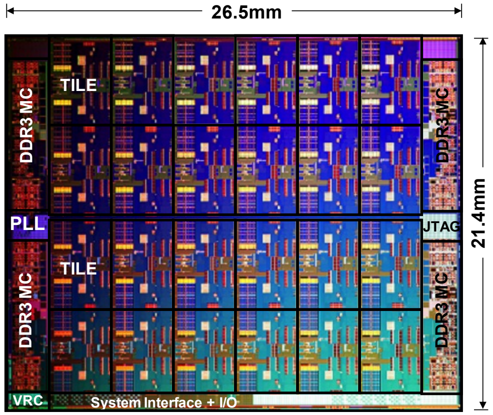
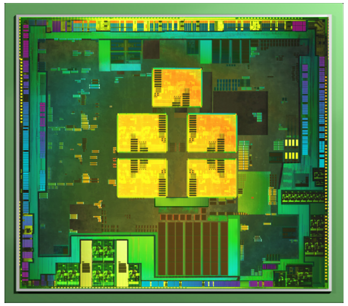


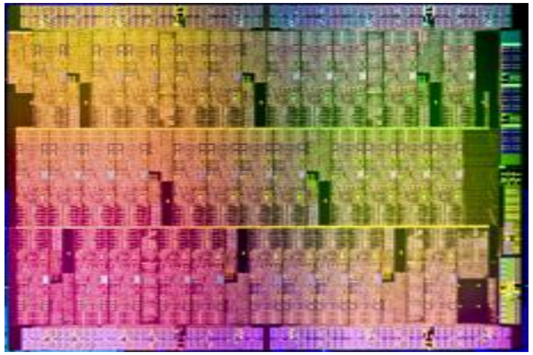
NVIDIA GTX 480 processor



Intel labs 48 core SCC processor



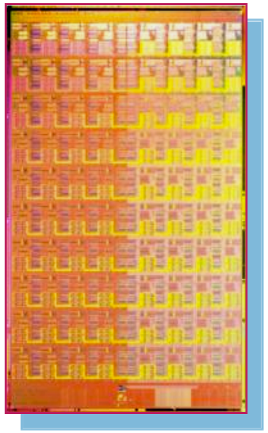
NVIDIA Tegra 3 (quad Arm Corex A9 cores + GPU)



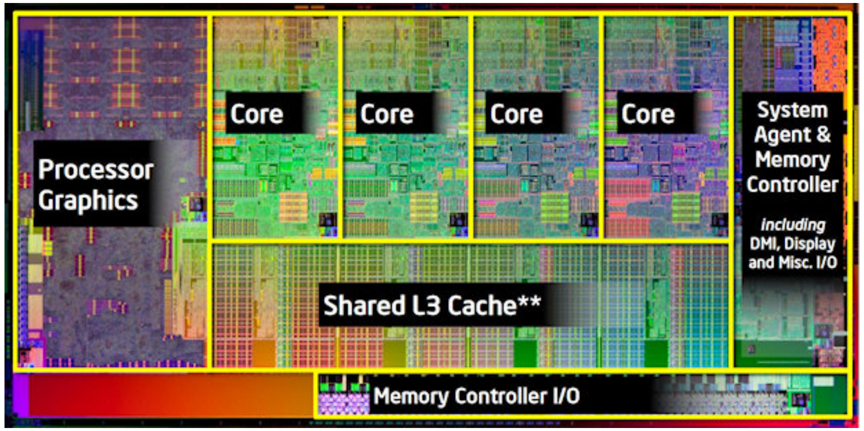
An Intel MIC processor

Introduction to Parallel Computing

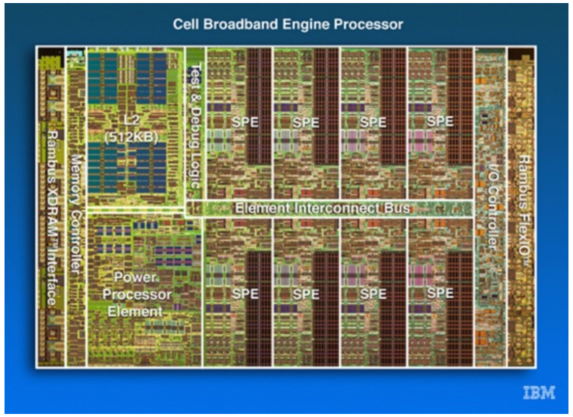
Tim Mattson (Intel Labs)



Intel Labs 80 core Research processor



Intel "Sandybridge" processor



IBM Cell Broadband engine processor

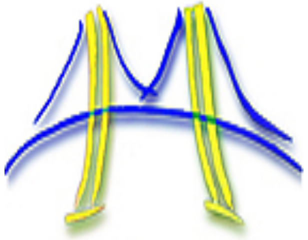
Other than the Intel lab's research processors. Die photos from UC Berkeley CS194 lecture notes

Third party names are the property of their owners

Disclaimer

READ THIS … its very important

- The views expressed in this talk are those of the speakers and not their employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if I say anything really stupid, it's my fault … don't blame my collaborators.

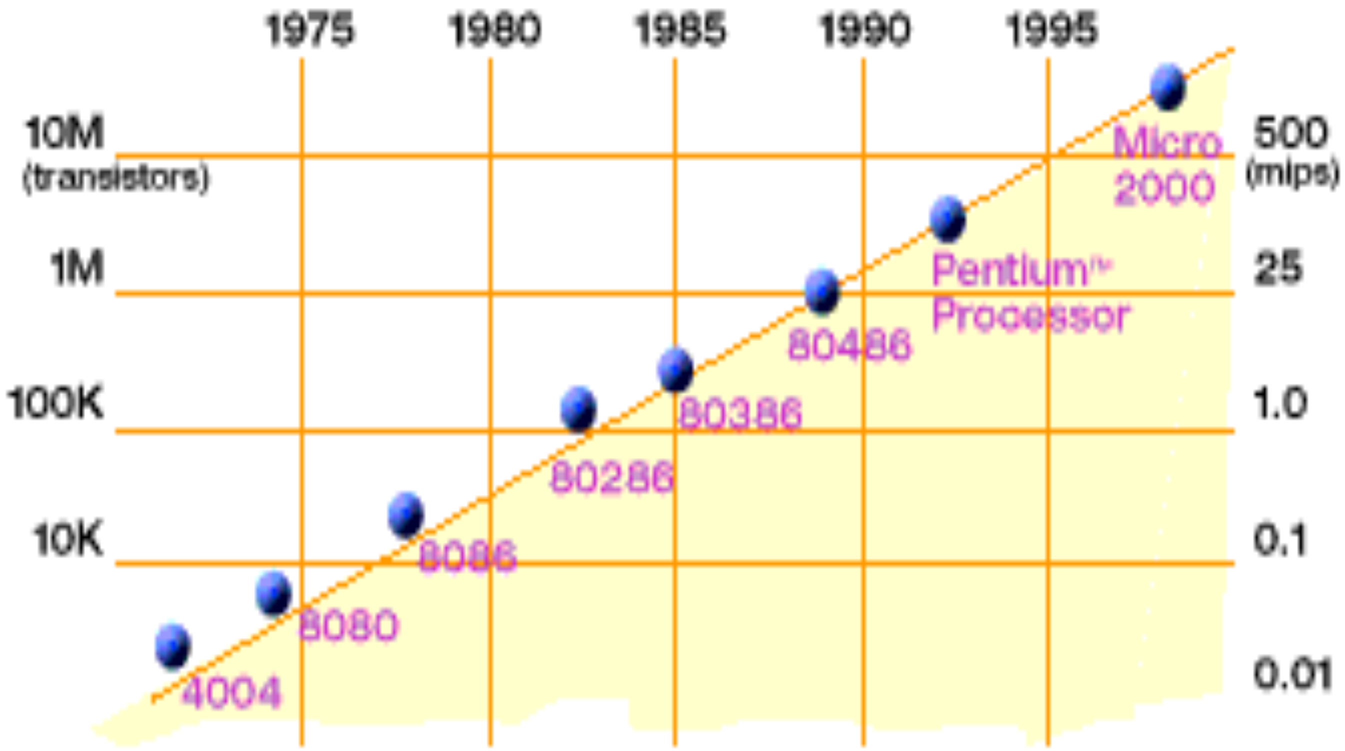


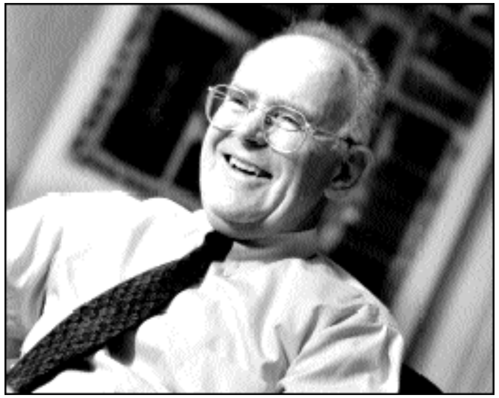
Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 … A UC Berkeley course on Architecting parallel applications with Design Patterns.

Outline

- ➡ Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
- An introduction to parallel hardware
- Software for parallel systems: key design patterns
- Closing comments

Moore's Law

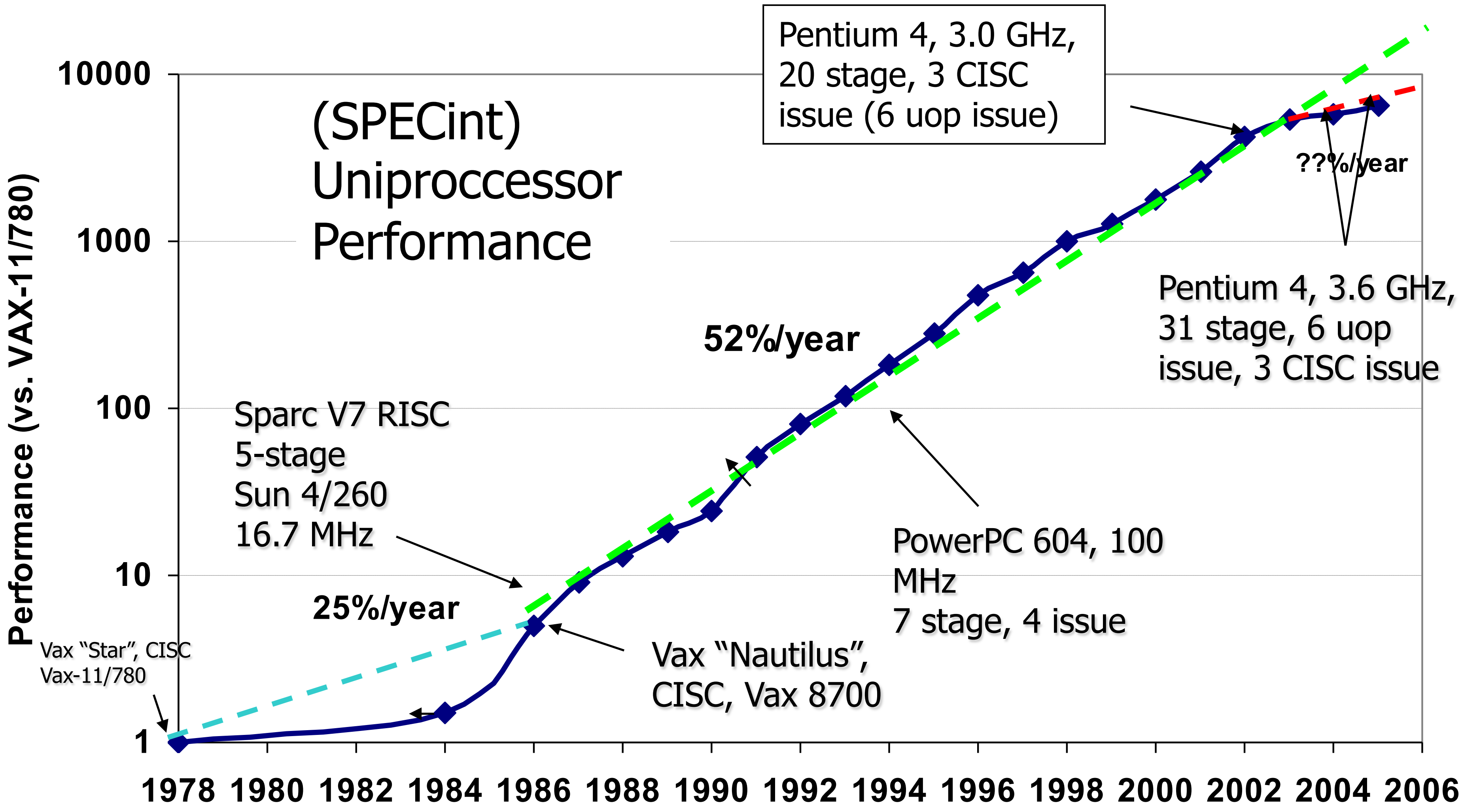




- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Transistors are *still* shrinking at the same rate

Slide source: UCB CS 194 Fall'2010

The good old days …

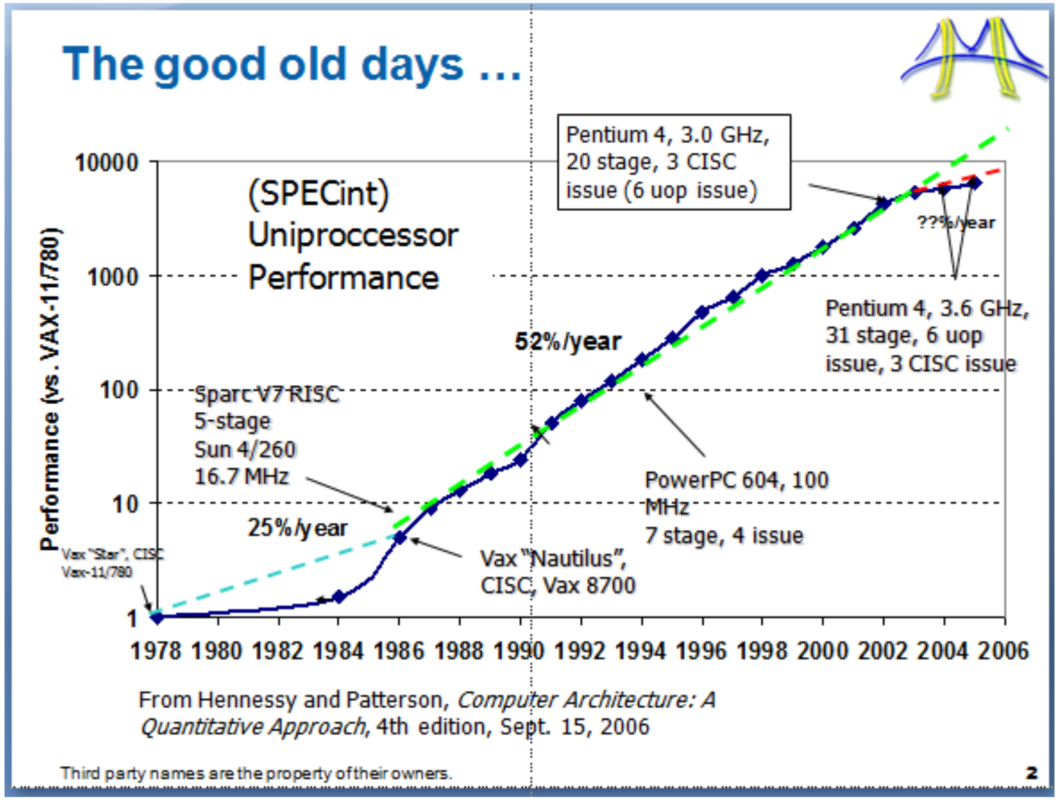


From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

Third party names are the property of their owners.

The Hardware/Software contract

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



The good old days …

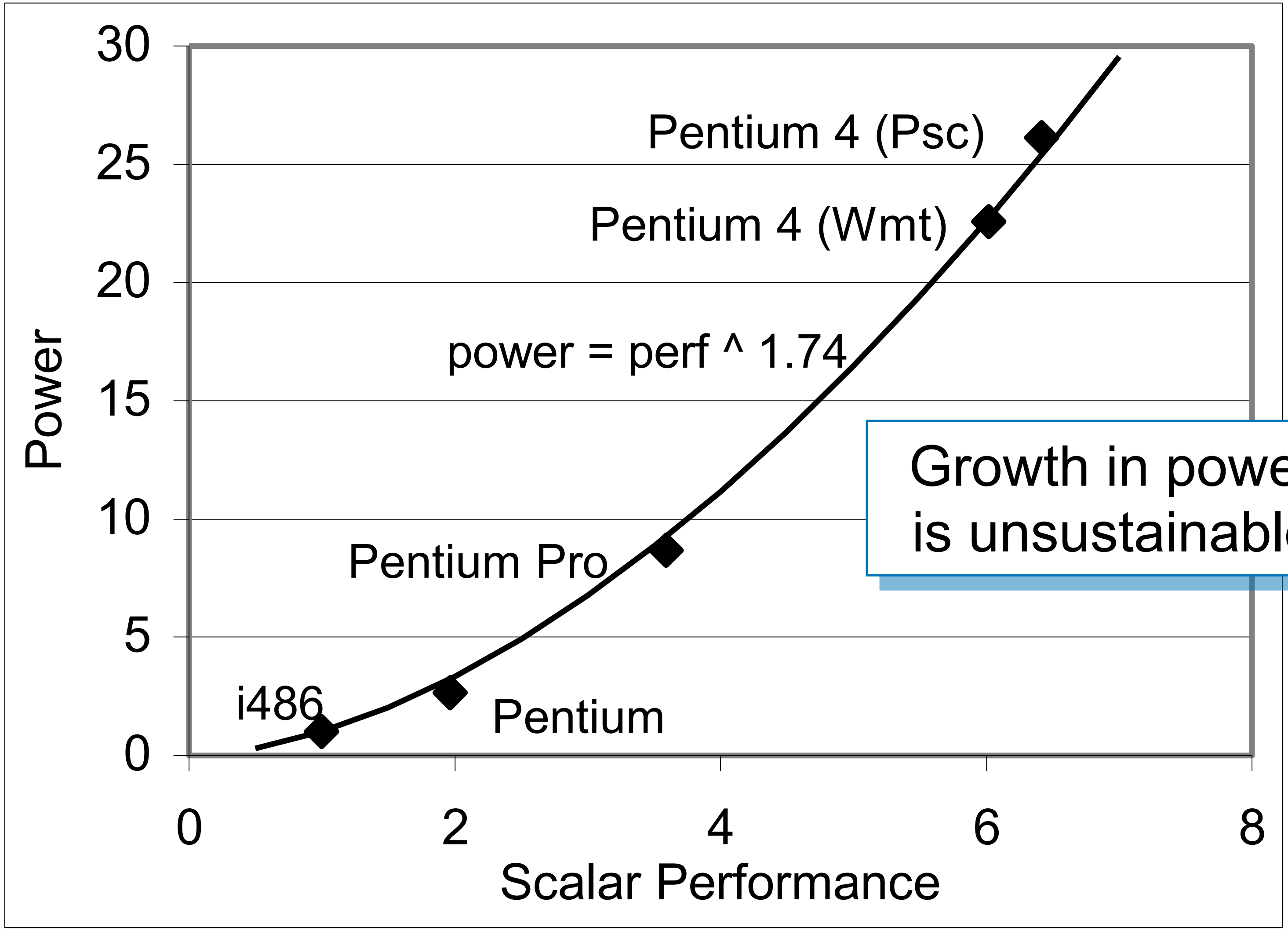
From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

Third party names are the property of their owners.

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

Third party names are the property of their owners.

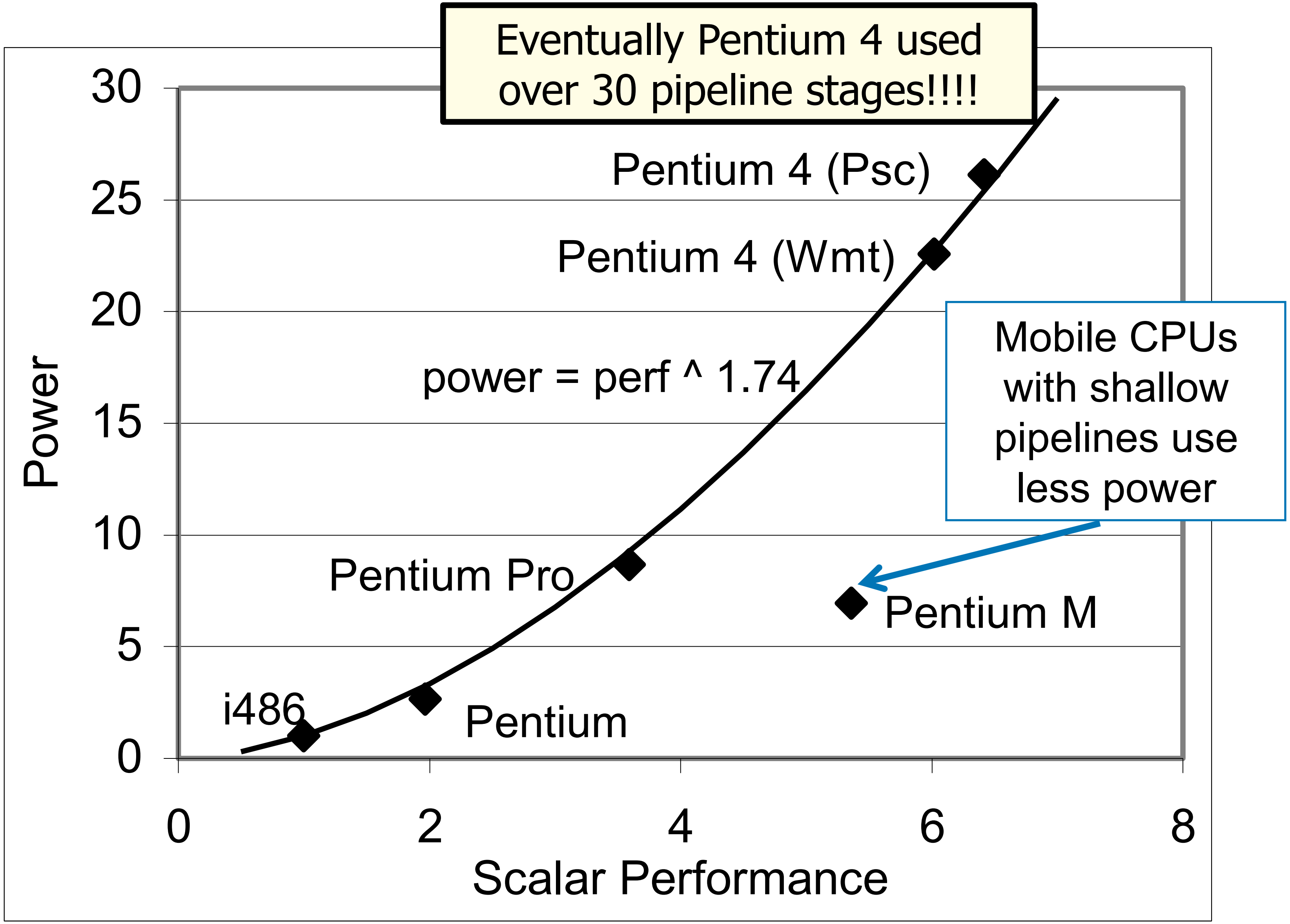
… Computer architecture and the power wall



Growth in power is unsustainable

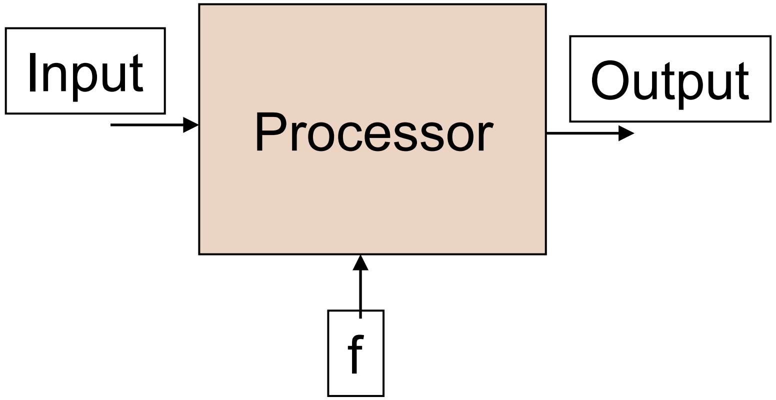
Source: E. Grochowski of Intel

… partial solution: simple low power cores



Source: E. Grochowski of Intel

For the rest of the solution 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:

$$V * q = W.$$

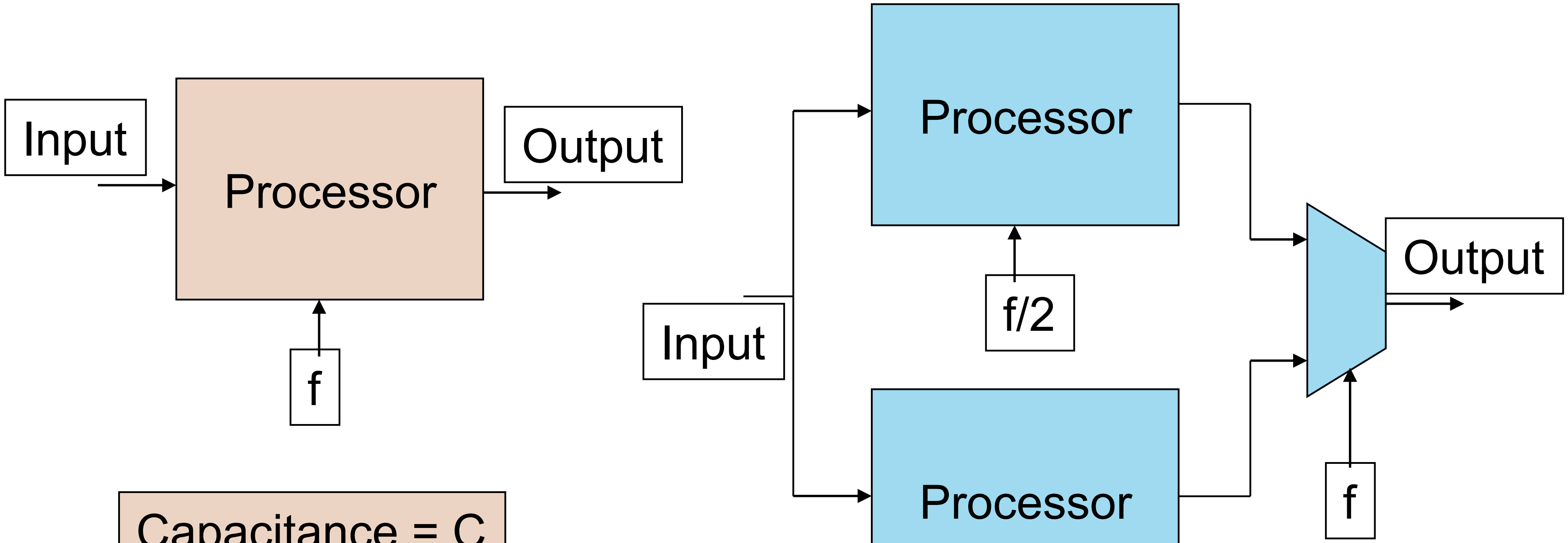
But for a circuit $q = CV$ so

$$W = CV^2$$

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

$$\text{Power} = W* F \quad \rightarrow \quad \text{Power} = CV^2f$$

... The rest of the solution add cores



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

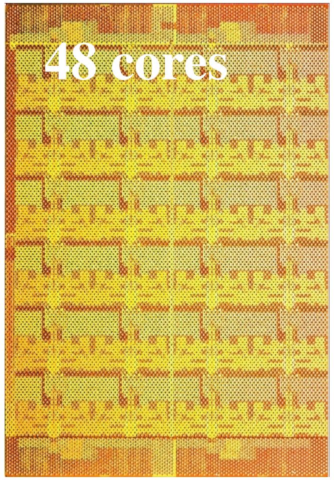
Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

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

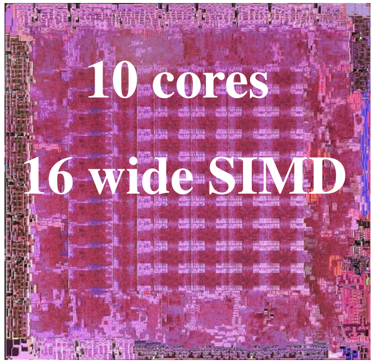
Source:
Vishwani Agrawal

Microprocessor trends

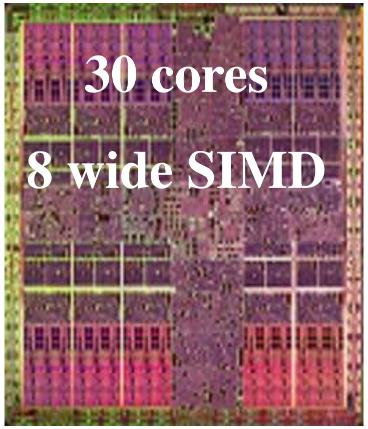
Individual processors are many core (and often heterogeneous) processors.



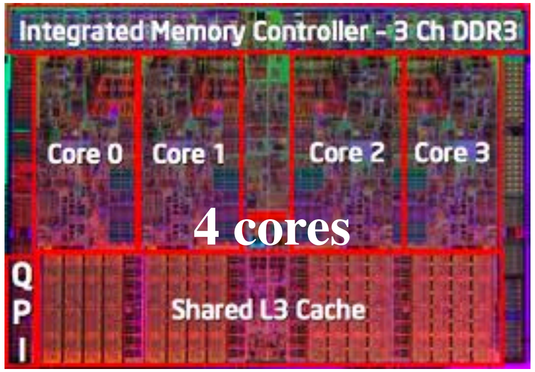
Intel SCC Processor



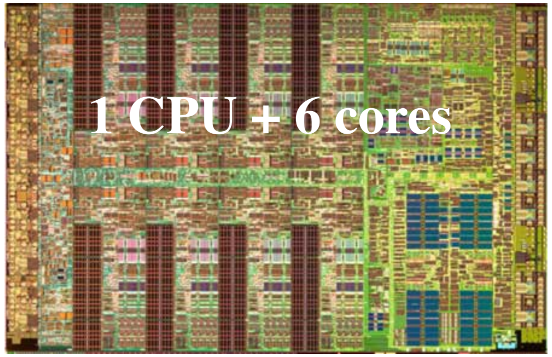
ATI RV770



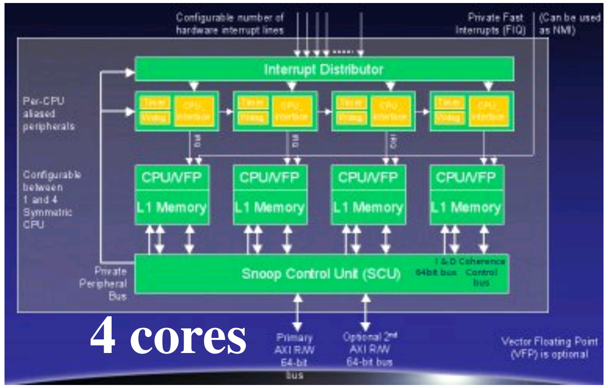
NVIDIA Tesla C1060



Intel Nehalem



IBM Cell

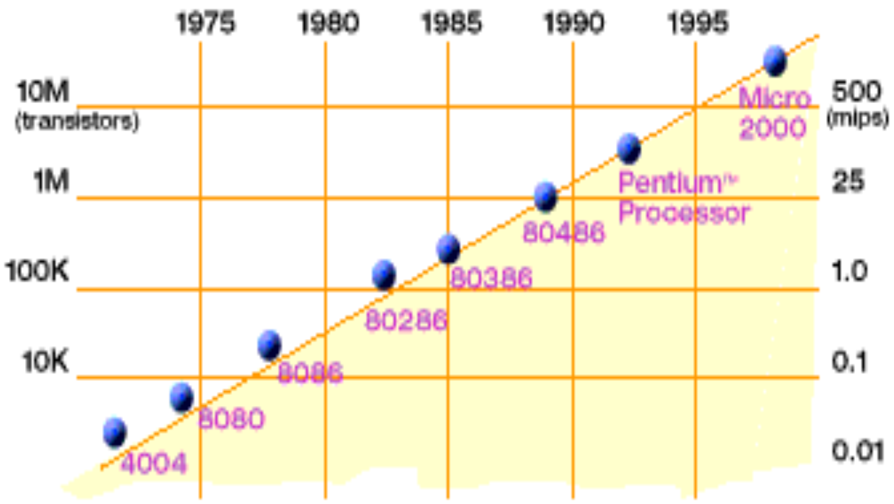

Configurable number of hardware interrupt lines
Private Fast Interrupts (FIQ)
(Can be used as NMI)
Interrupt Distributor
Per-CPU aliased peripherals
Configurable between 1 and 4 Symmetric CPU
CPU/VFP
L1 Memory
CPU/VFP
L1 Memory
CPU/VFP
L1 Memory
CPU/VFP
L1 Memory
Private Peripheral Bus
Snoop Control Unit (SCU)
I & D Coherence 64bit bus
Control bus
4 cores
Primary AXI R/W 64-bit bus
Optional 2nd AXI R/W 64-bit bus
Vector Floating Point (VFP) is optional

ARM MPCORE

3^{rd} party names are the property of their owners.

Source: OpenCL tutorial, Gaster, Howes, Mattson, and Lokhmotov, HiPEAC 2011

So how many cores?



2X transistors/Chip Every 1.5 years

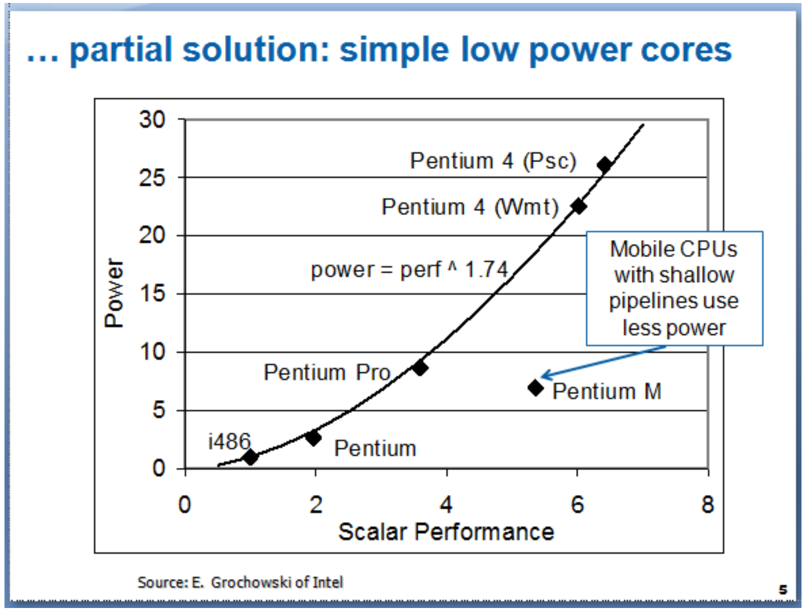
Called "Moore's Law"

Microprocessors have become smaller, denser, and more powerful.

- Let's assume Moore's law transistor doubling results in a doubling of the number of cores.
  - 50 cores in 2010
  - 100 cores in 2012
  - 200 cores in 2014
  - 400 cores in 2016
  - 800 cores in 2018
  - 1600 cores in 2020
- So 1000 cores in 10 years is not far fetched.

Market forces, not technology, will drive core counts

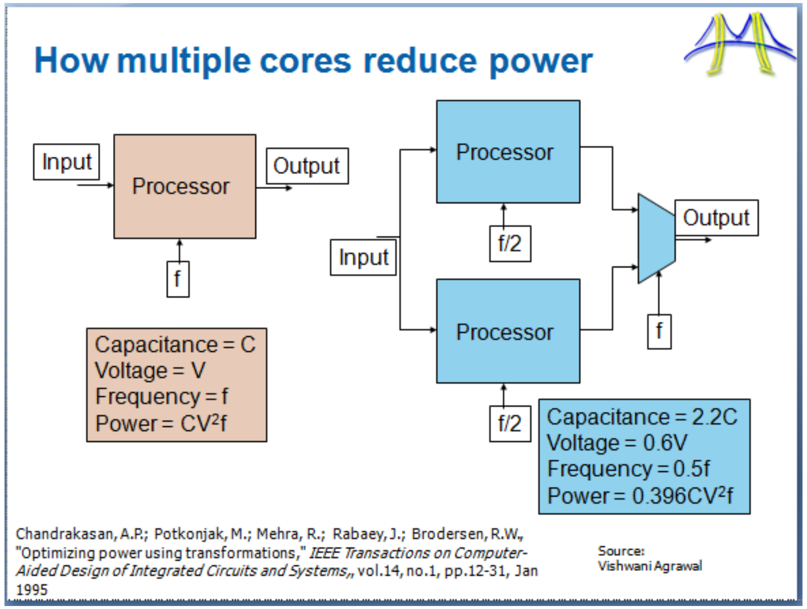
The result…



… partial solution: simple low power cores

Source: E. Grochowski of Intel

+



How multiple cores reduce power

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

=

A new contract … HW people will do what's natural for them (lots of simple cores) and SW people 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.

The many core challenge

- A harsh assessment …
  - We have turned to multi-core chips not because of the success of our parallel software but because of our failure to continually increase CPU frequency.

- Result: a fundamental and dangerous (for the computer industry) mismatch
  - □ Parallel hardware is ubiquitous.
  - □ Parallel software is rare

- The Many Core challenge …
  - Parallel software must become as common as parallel hardware

**Fortunately, we don't have to start over "from scratch".
We can draw from past experience with parallelism from high performance computing**

Outline

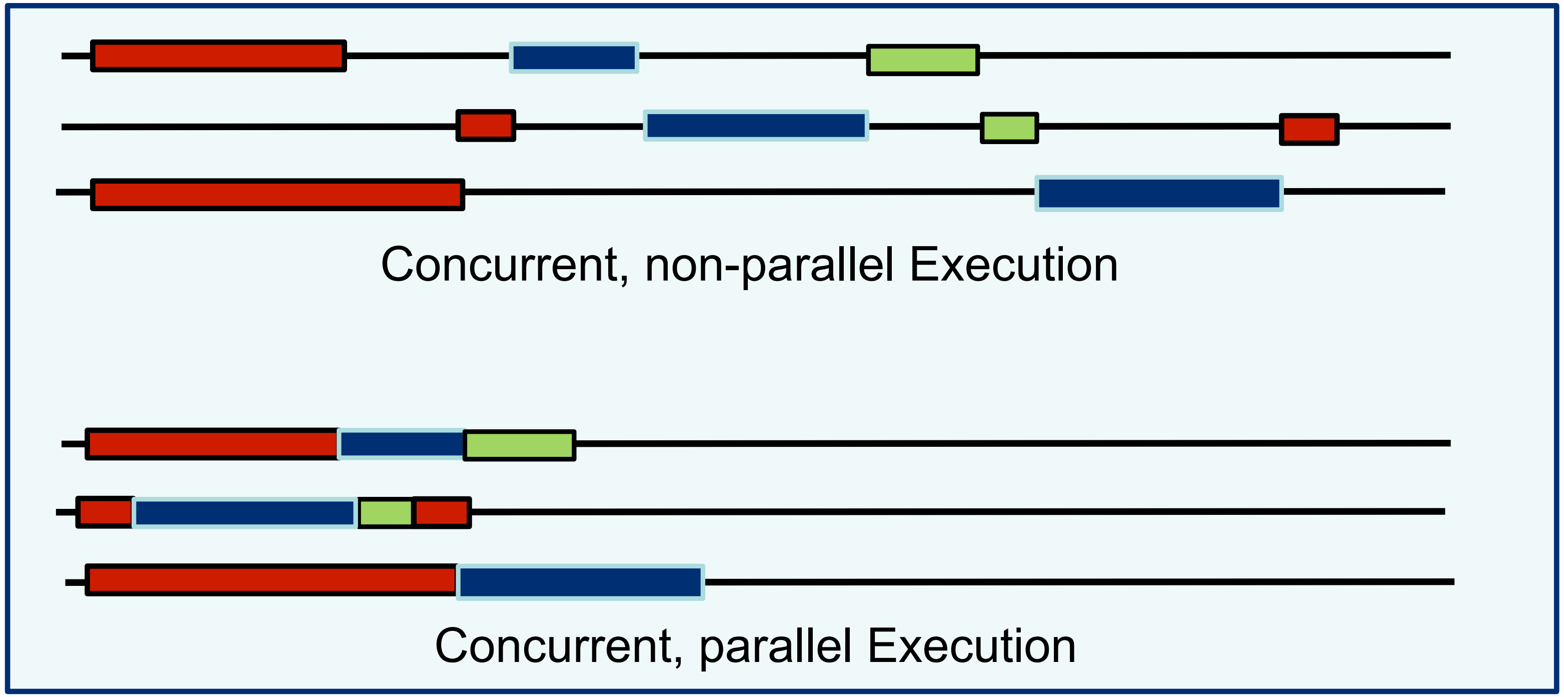
- Motivation: We all must be parallel programmers
- ➡ Key concepts in parallel Computing
- An introduction to parallel hardware
- Software for parallel systems: key design patterns
- Closing comments

Outline

- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
  - ➡ Basic definitions: Parallelism and Concurrency
  - Notions of parallel performance
  - The limits of scalability
  - Sources of parallel overhead
- An introduction to parallel hardware
- Software for parallel systems: key design patterns
- Closing comments

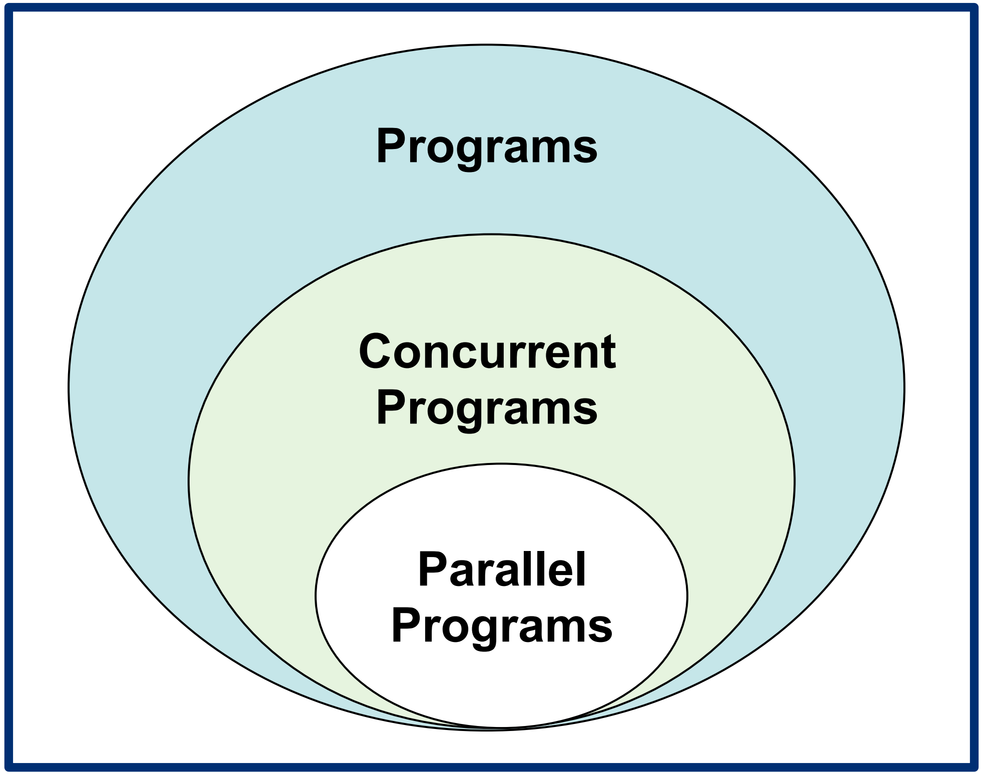
Concurrency vs. Parallelism

- 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.


Concurrent, non-parallel Execution
Concurrent, parallel Execution

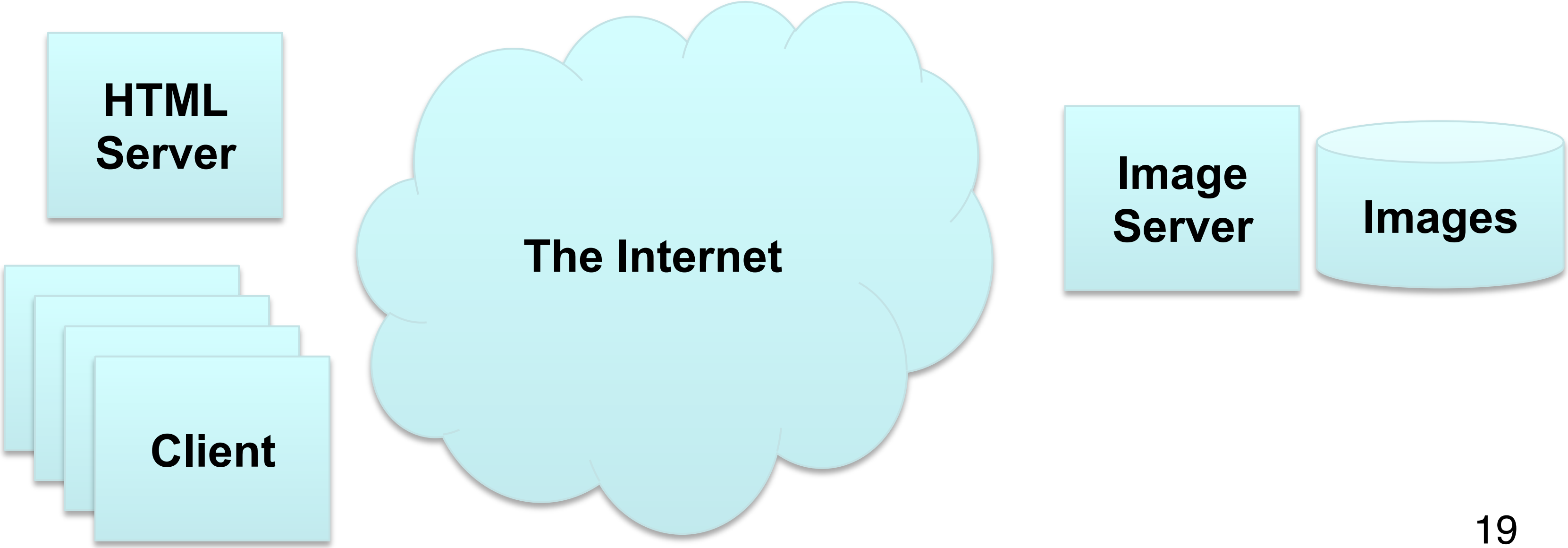
Concurrency vs. Parallelism

- 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.



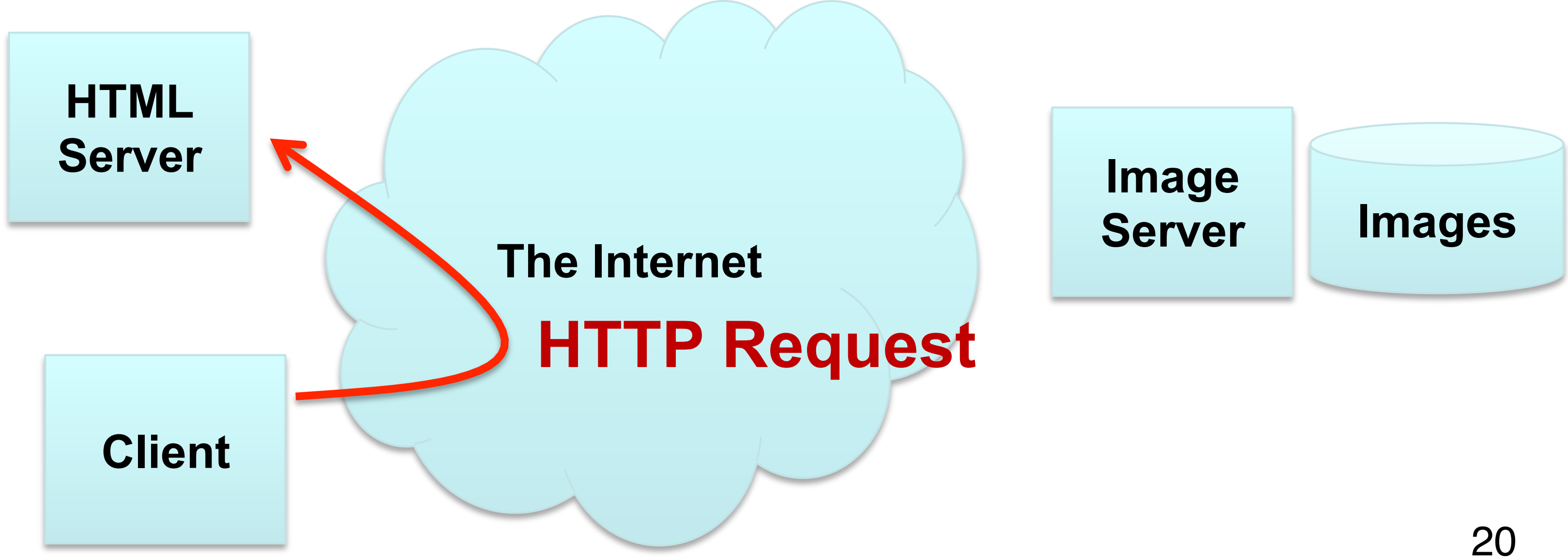
Concurrency in Action: a web server

- 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

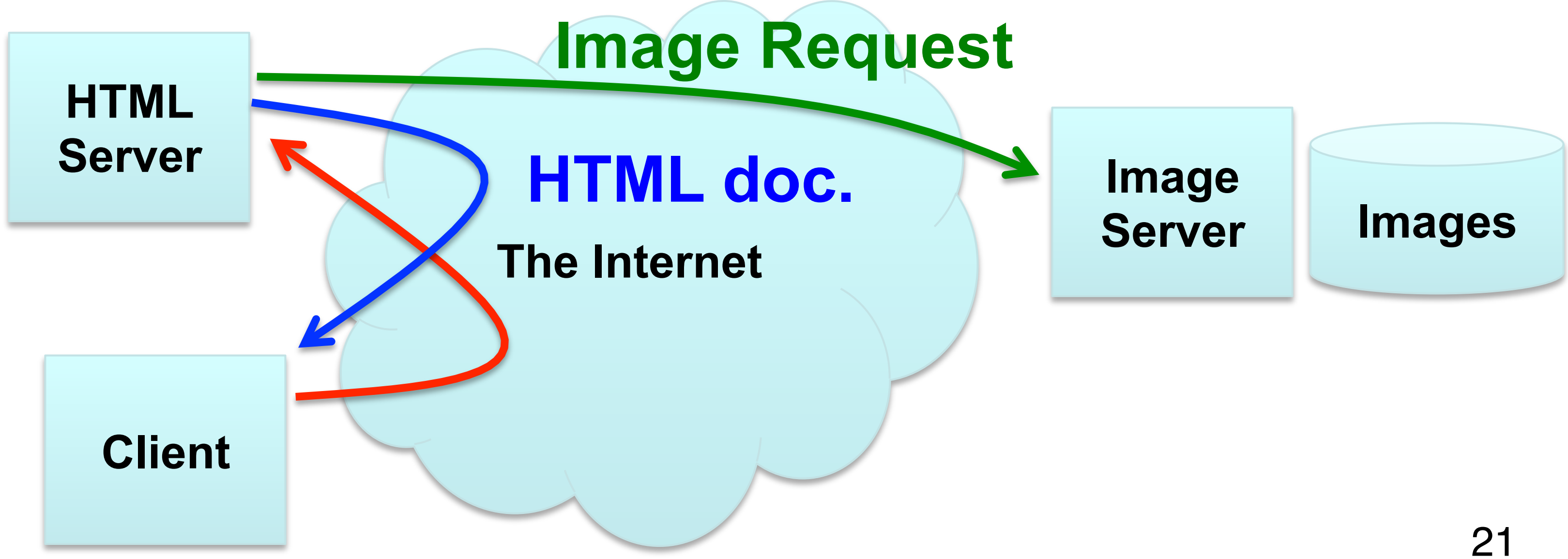
- A **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):
  - An arbitrary, large number of clients make requests which reference per-client persistent state
- Consider an **Image Server**, which relieves load on primary web servers by storing,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



For each client …

Concurrency in Action: a web server

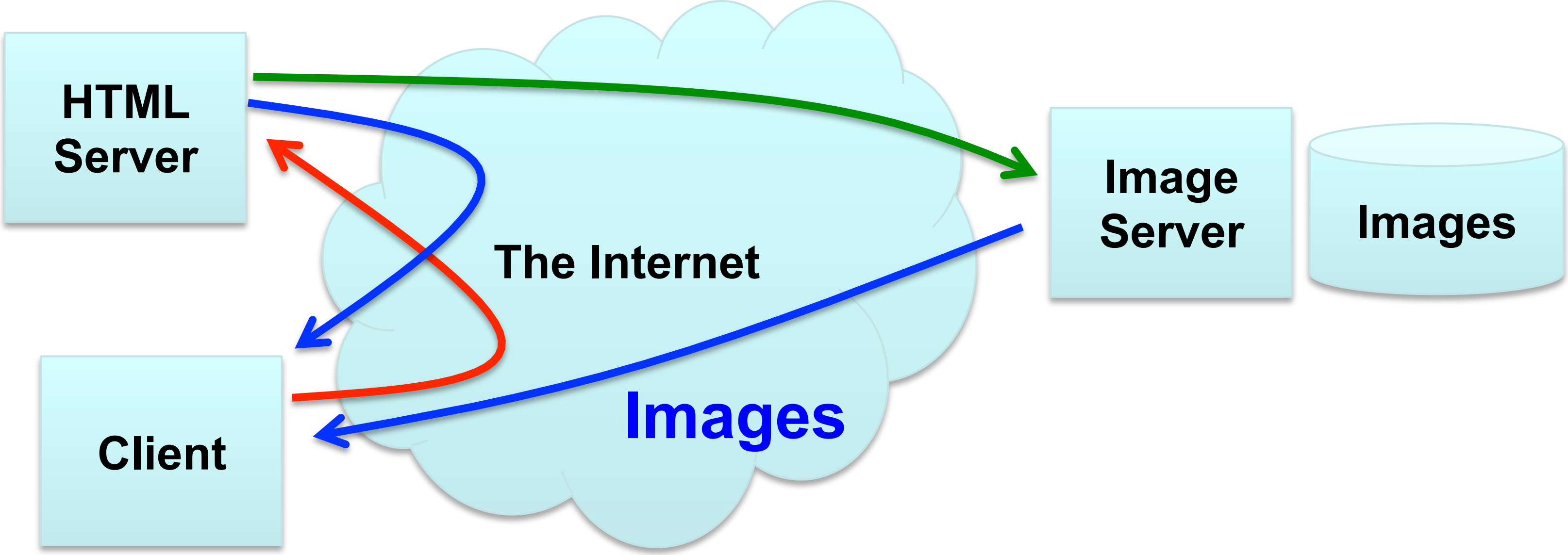
- A **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):
  - An arbitrary, large number of clients make requests which reference per-client persistent state
- Consider an **Image Server**, which relieves load on primary web servers by storing,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



For each client …

Concurrency in Action: a web server

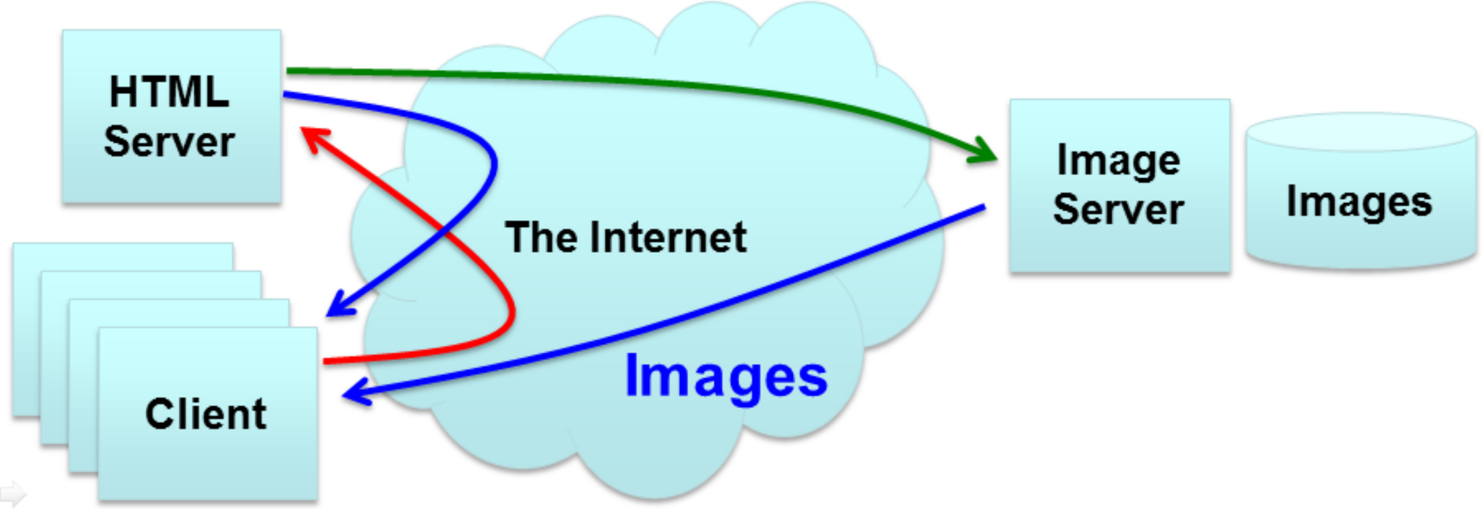
- A **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):
    - An arbitrary, large number of clients make requests which reference per-client persistent state
- Consider an **Image Server**, which relieves load on primary web servers by storing,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



For each client …

Concurrency in Action: a web server

- The HTML 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.

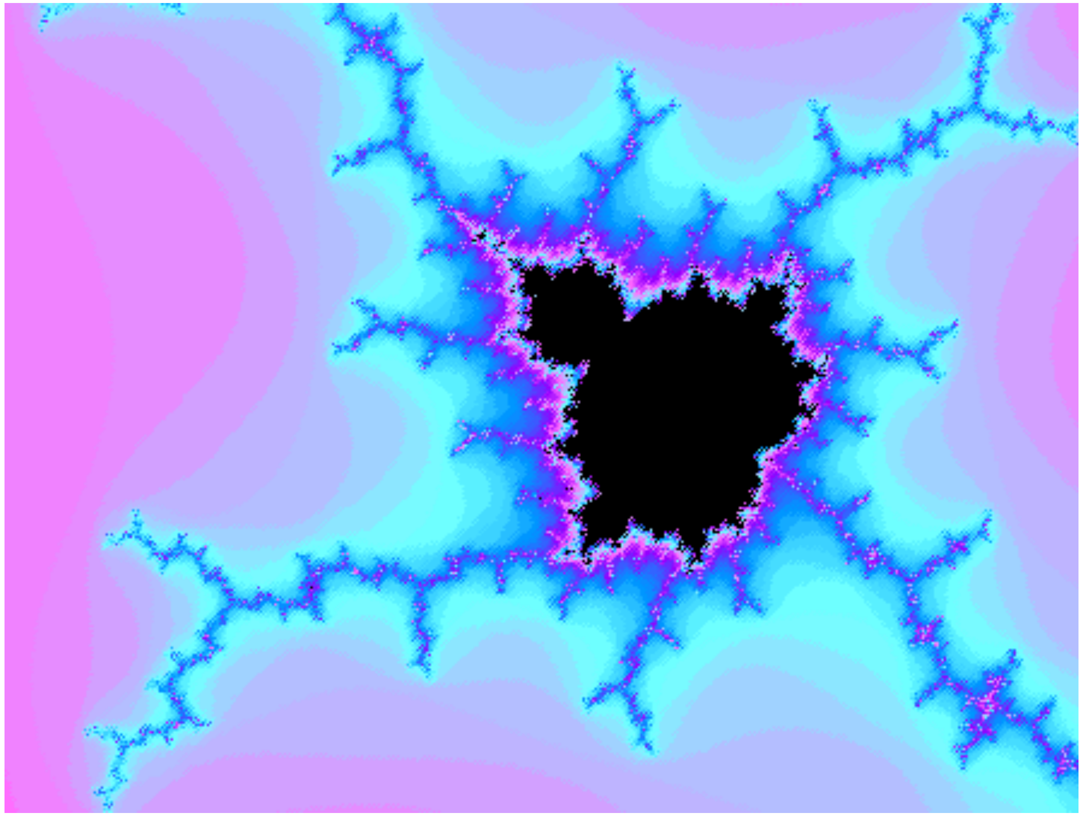
Concurrent application: An application for which the problem definition is fundamentally concurrent.

Concurrency in action: Mandelbrot Set

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

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

- Plot rate of divergence for different values of C.



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

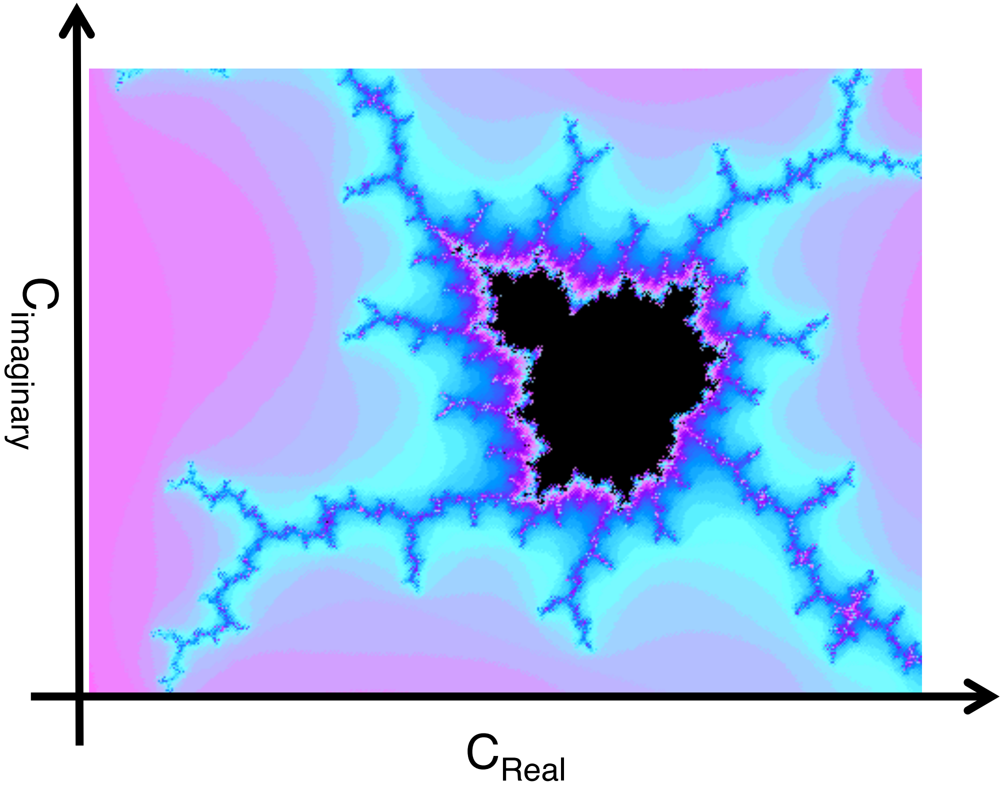
Where i is the square root of (-1)

"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

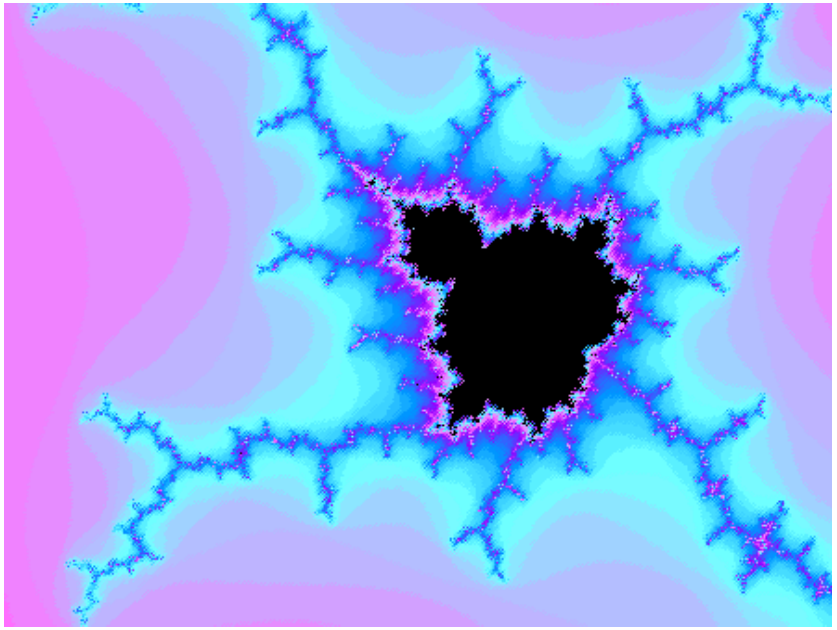
- The following is simplified code for the serial Mandelbrot program.…
- Loop iterations are independent, so we can create a parallel version of this program as follows …

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

```
#pragma omp parallel for collapse (2)
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

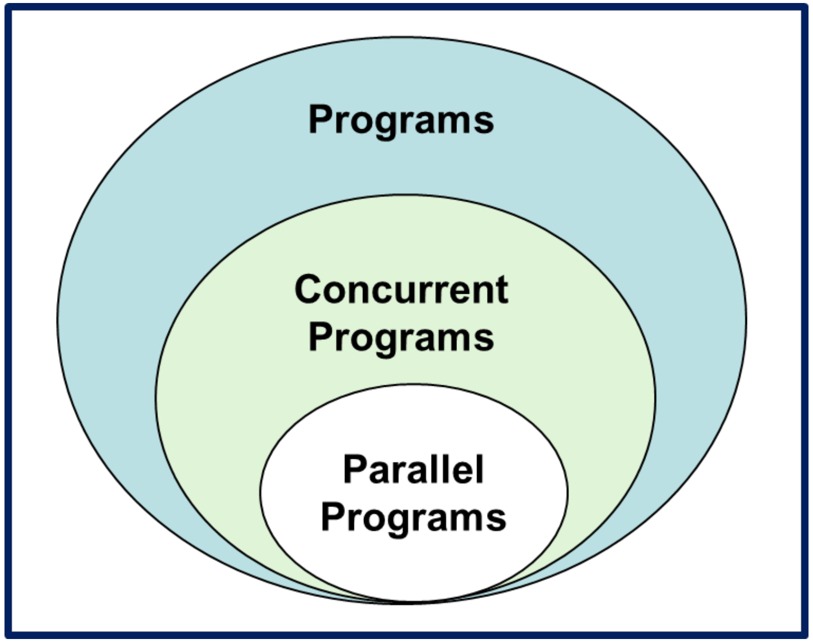
- The problem of generating an image of the Mandelbrot set *can be* viewed serially.
- We choose to exploit the concurrency contained in this problem so we can generate the image in less time



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

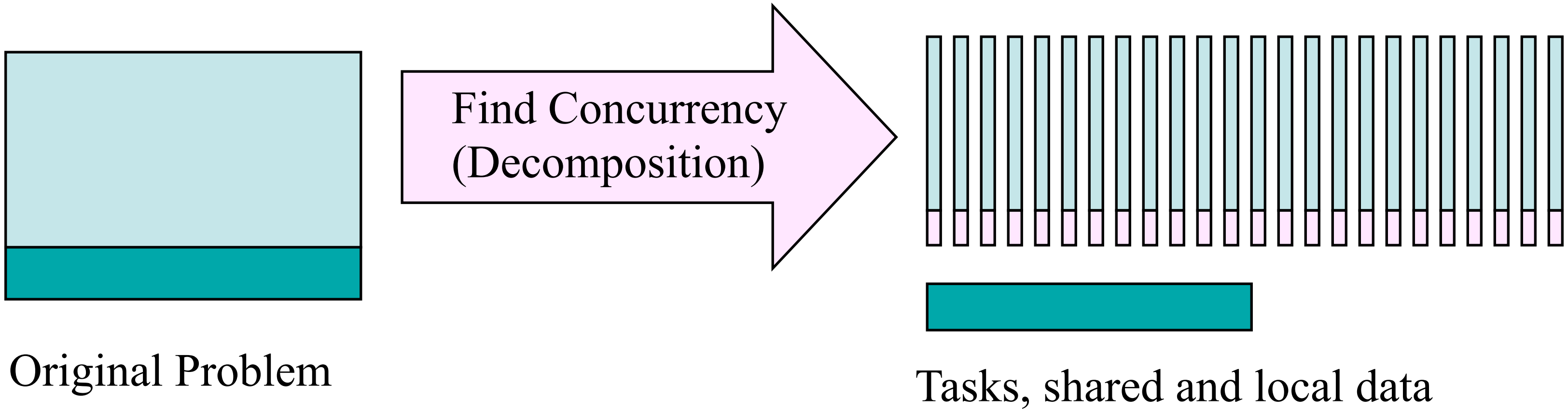
- Key points:
  - A web server had concurrency in its problem definition … it doesn't make sense to even think of writing a "serial web server".
  - The Mandelbrot program didn't have concurrency in its problem definition. It would take a long time, but it could be serial



- 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.

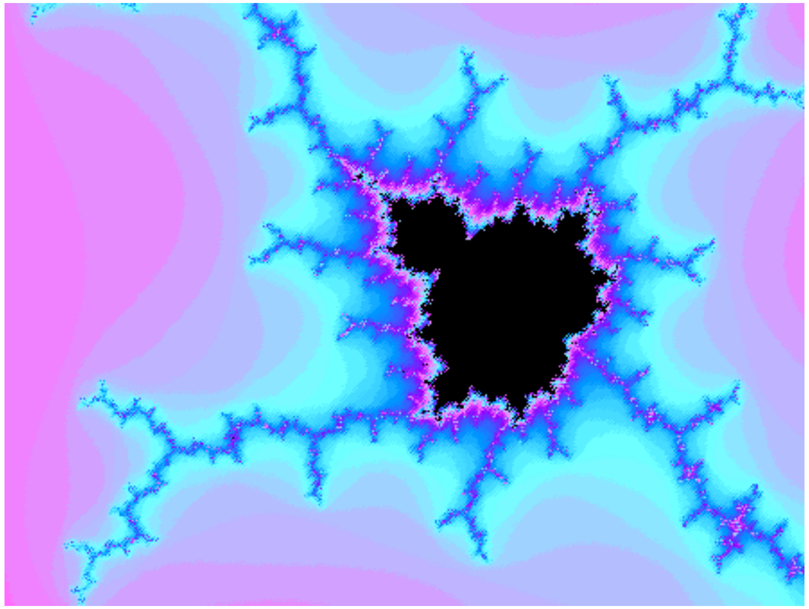
Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

The Parallel programming process:



Decomposition in parallel programs

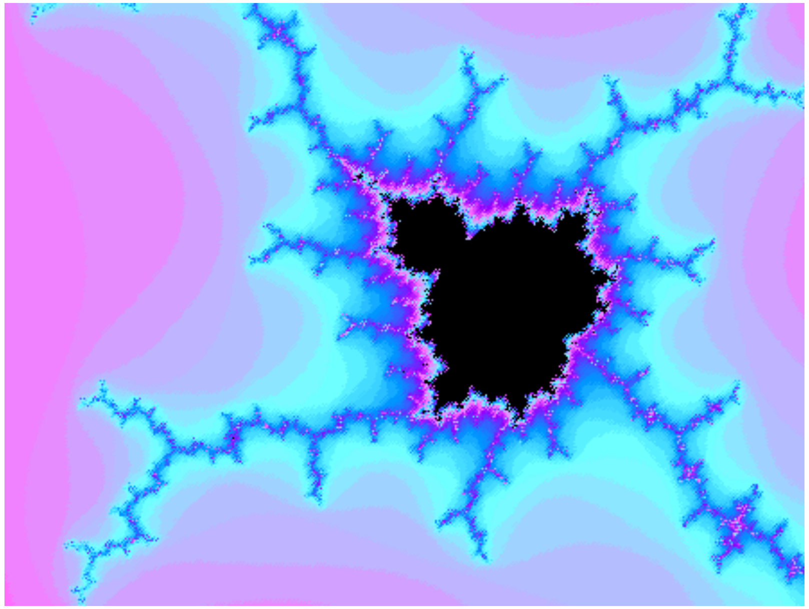
- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..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

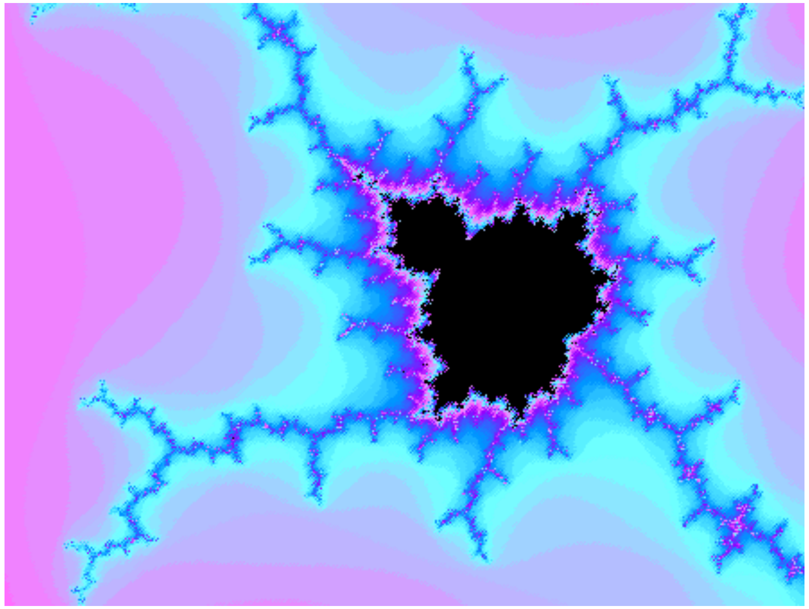
What's a task decomposition for this problem?



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

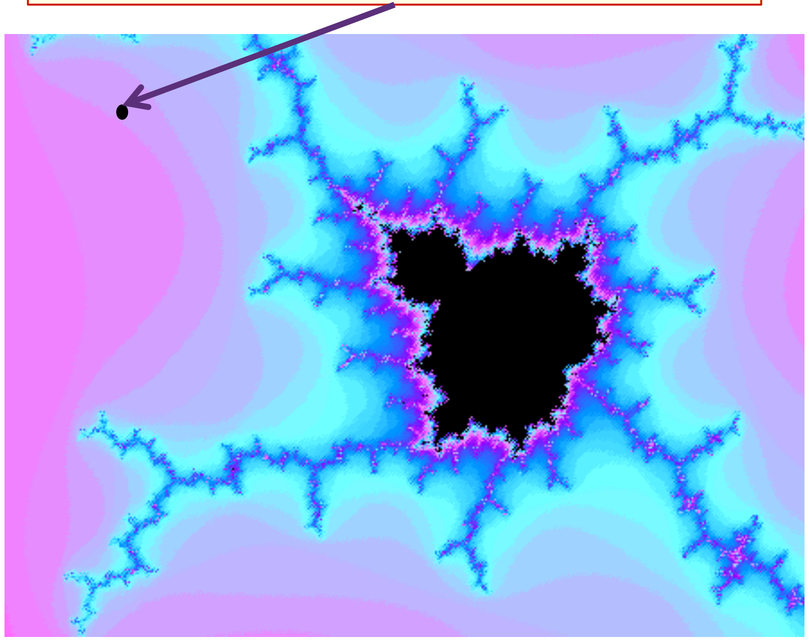


```
for (i=0; i<N; i++){
   for (j=0; j<N; j++) {
       complex c = get_const_at_pixel(i,j);
       complex image[i][j] = mandel( c);
   }
}
```

Decomposition in parallel programs

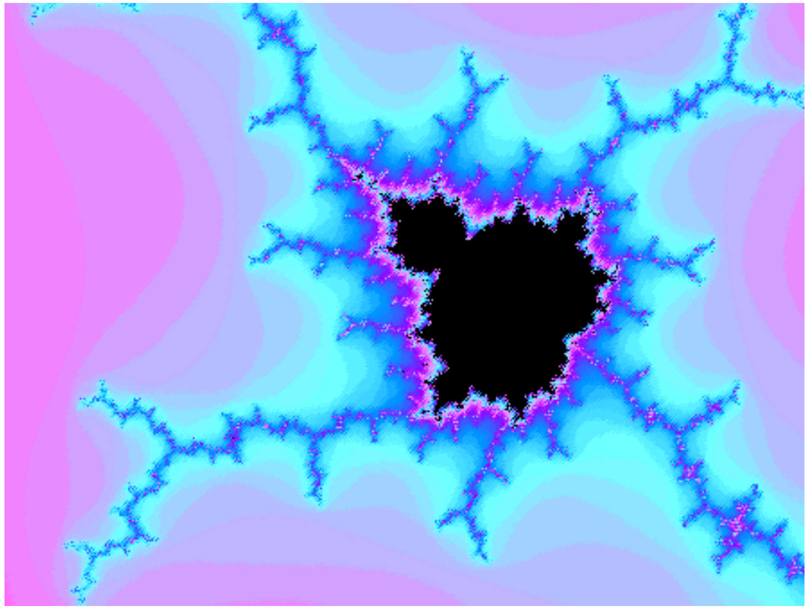
- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..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

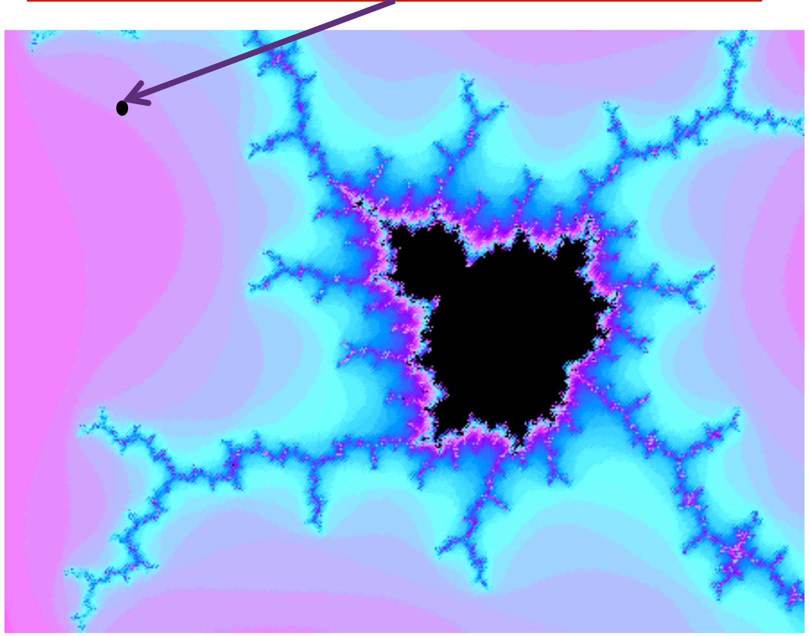


Suggest a data decomposition for this problem … assume a quad core shared memory PC.

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 be broken down into chunks and associated with threads/processes to make the parallel program run efficiently.

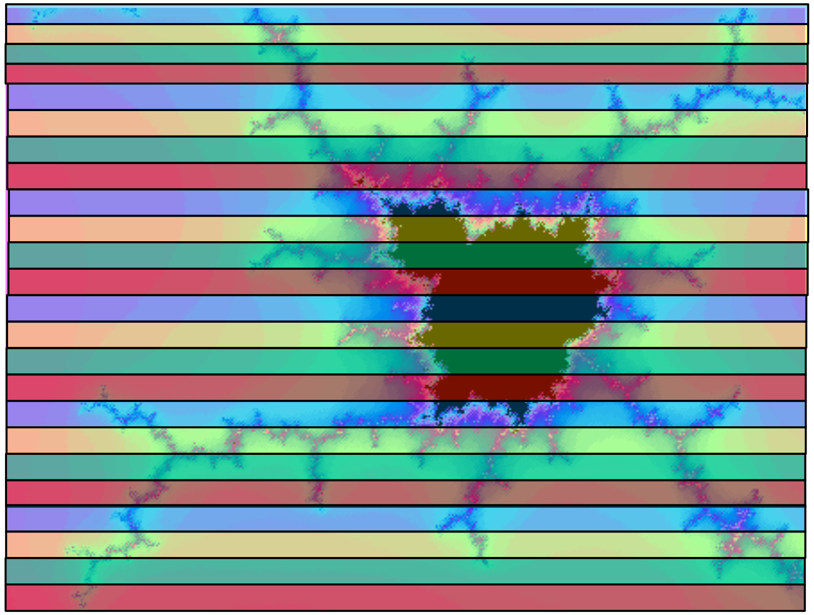
Task: the computation required for each pixel … the body of the loop for a pair (i,j).



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 be broken down into chunks and associated with threads/ processes to make the parallel program run efficiently.



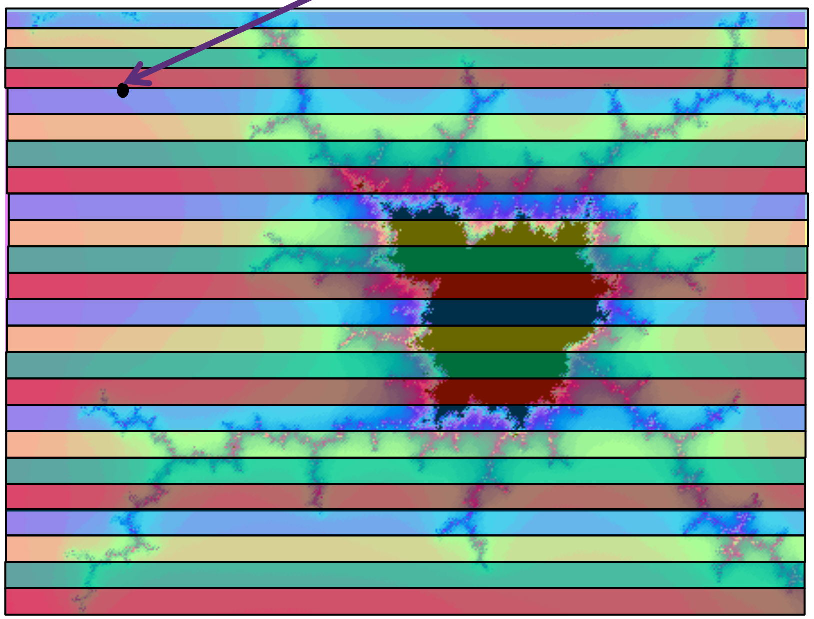
Map the pixels into row blocks and deal them out to the cores. This will give each core a memory efficient block to work on.

🟦 🟥
🟨 🟩

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 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?

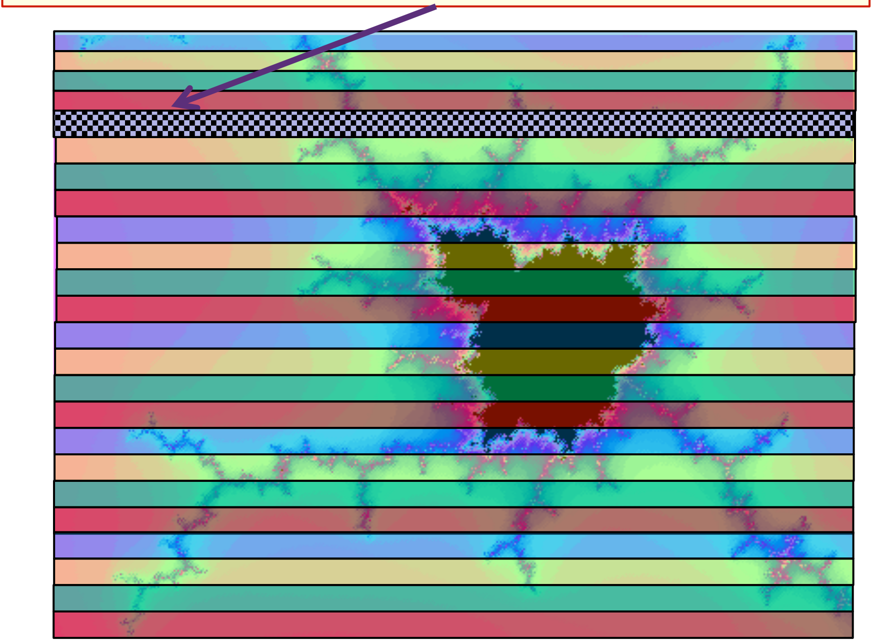


Map the pixels into row blocks and deal them out to the cores. This will give each core a memory efficient block to work on.

Decomposition in parallel programs

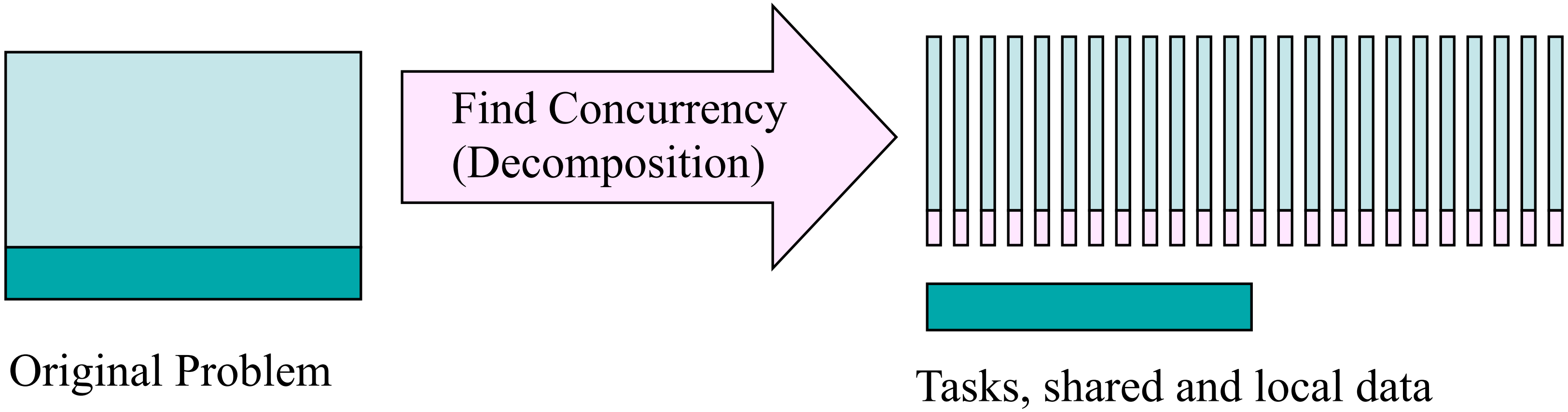
- Every parallel program is based on concurrency … i.e. tasks defined by an application that can run at the same time.
- **EVERY parallel program** requires a task decomposition and a data decomposition:
  - **Task decomposition**: break the application down into a set of tasks that can execute concurrently..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

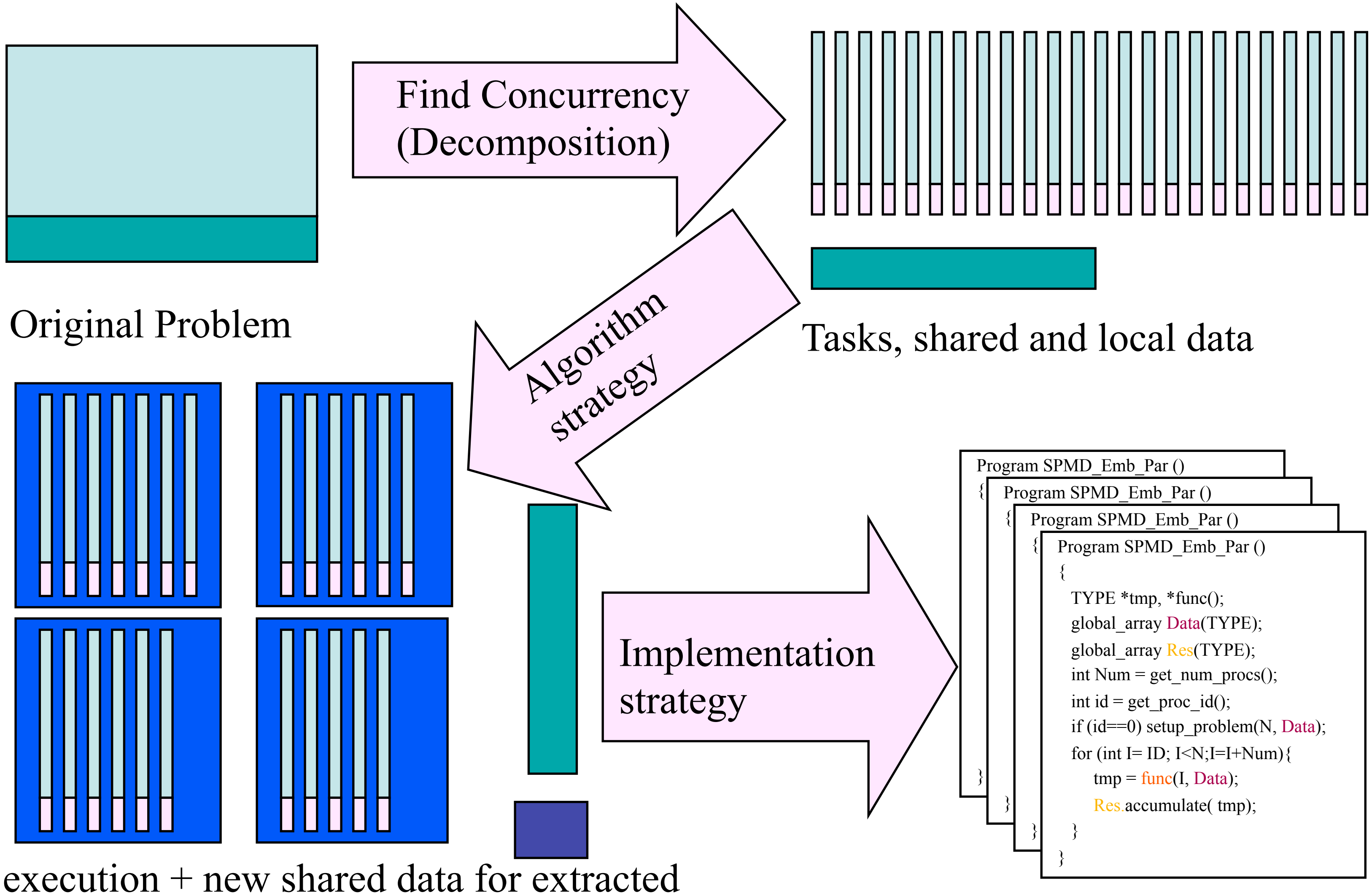


Map the pixels into row blocks and deal them out to the cores. This will give each core a memory efficient block to work on.

The Parallel programming process:



The Parallel programming process:



Find Concurrency (Decomposition)

Algorithm strategy

Implementation strategy

Original Problem

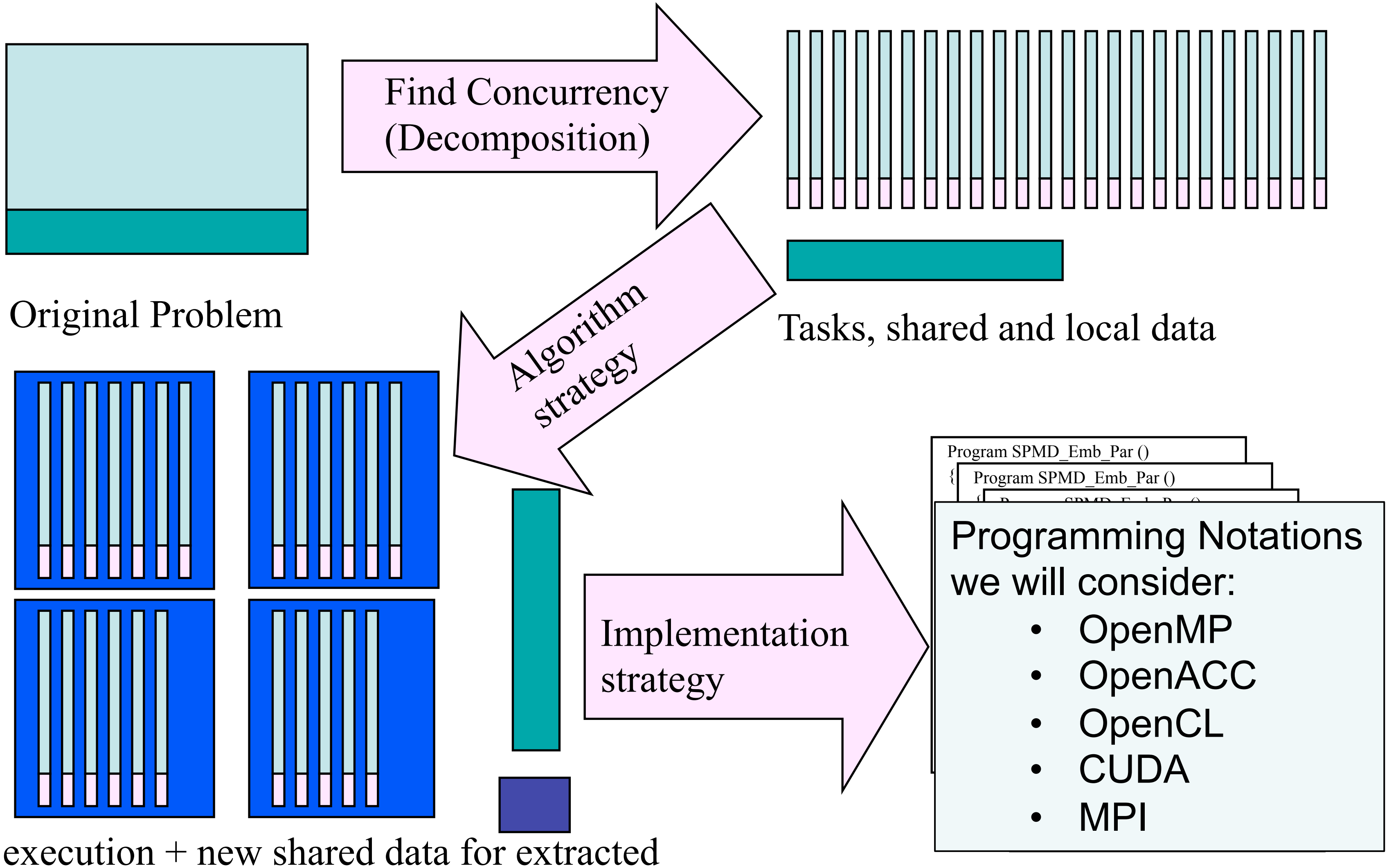
Tasks, shared and local data

Units of execution + new shared data for extracted dependencies

```
Program SPMD_Emb_Par ()
{
  TYPE *tmp, *func();
  global_array Data(TYPE);
  global_array Res(TYPE);
  int Num = get_num_procs();
  int id = get_proc_id();
  if (id==0) setup_problem(N, Data);
  for (int I= ID; I<N;I=I+Num){
      tmp = func(I, Data);
      Res.accumulate( tmp);
  }
}
```

Corresponding source code

The Parallel programming process:



Find Concurrency (Decomposition)

Original Problem

Tasks, shared and local data

Algorithm strategy

Program SPMD_Emb_Par ()

{

Program SPMD_Emb_Par ()

Implementation strategy

Programming Notations we will consider:

- OpenMP
- OpenACC
- OpenCL
- CUDA
- MPI

Units of execution + new shared data for extracted dependencies

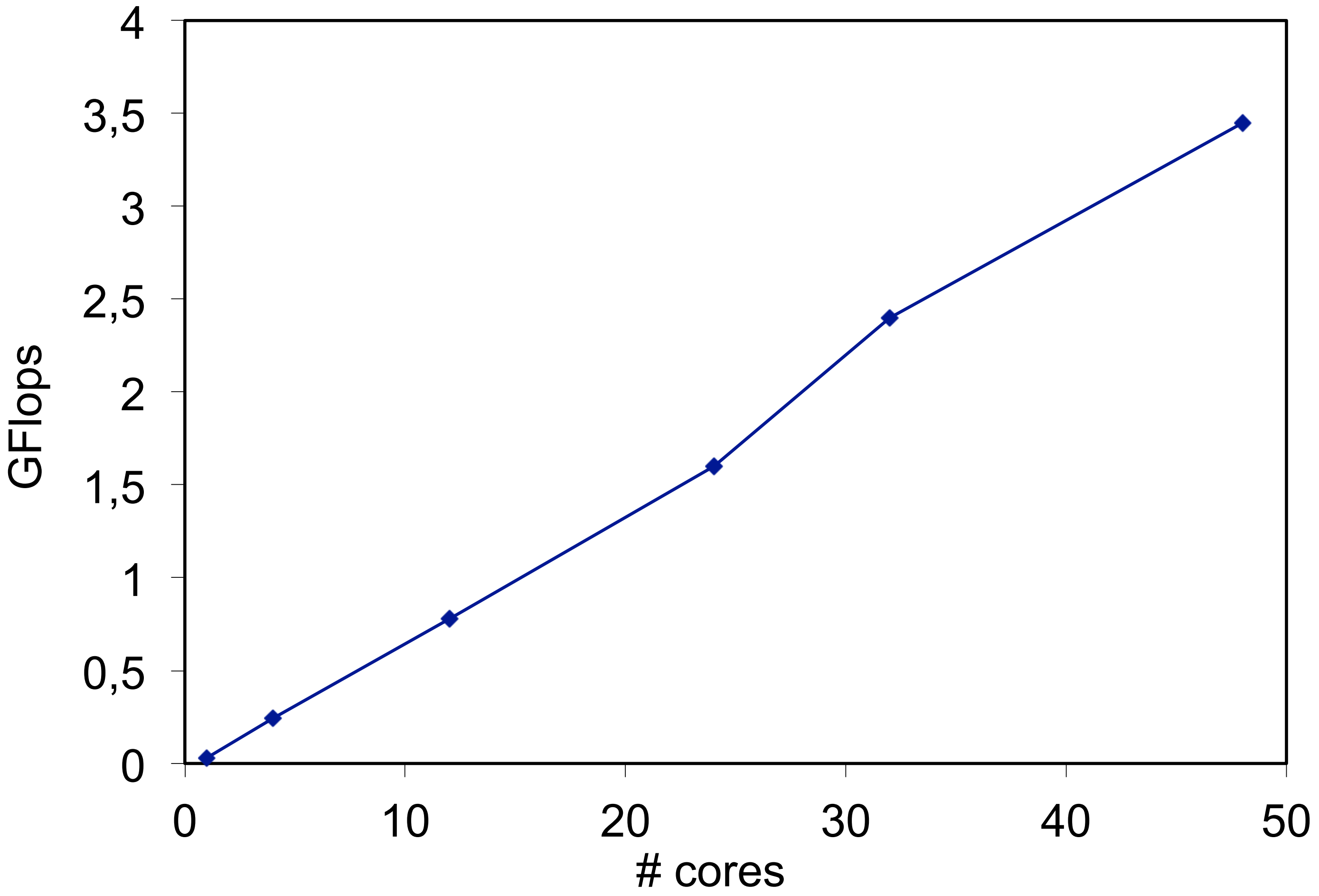
Corresponding source code

Outline

- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
  - Basic definitions: Parallelism and Concurrency
  - ➡ Notions of parallel performance
  - The limits of scalability
  - Sources of parallel overhead
- An introduction to parallel hardware
- Software for parallel systems: key design patterns
- Closing comments

Parallel Performance

- MP Linpack benchmark, order 1000 matrix (solve a dense system of linear equations … the dense linear algebra computational pattern).



- Intel SCC 48 processor, 500 MHz core, 1 GHz router, DDR3 at 800 MHz.

Talking about performance

- Speedup: the increased performance from running on P processors

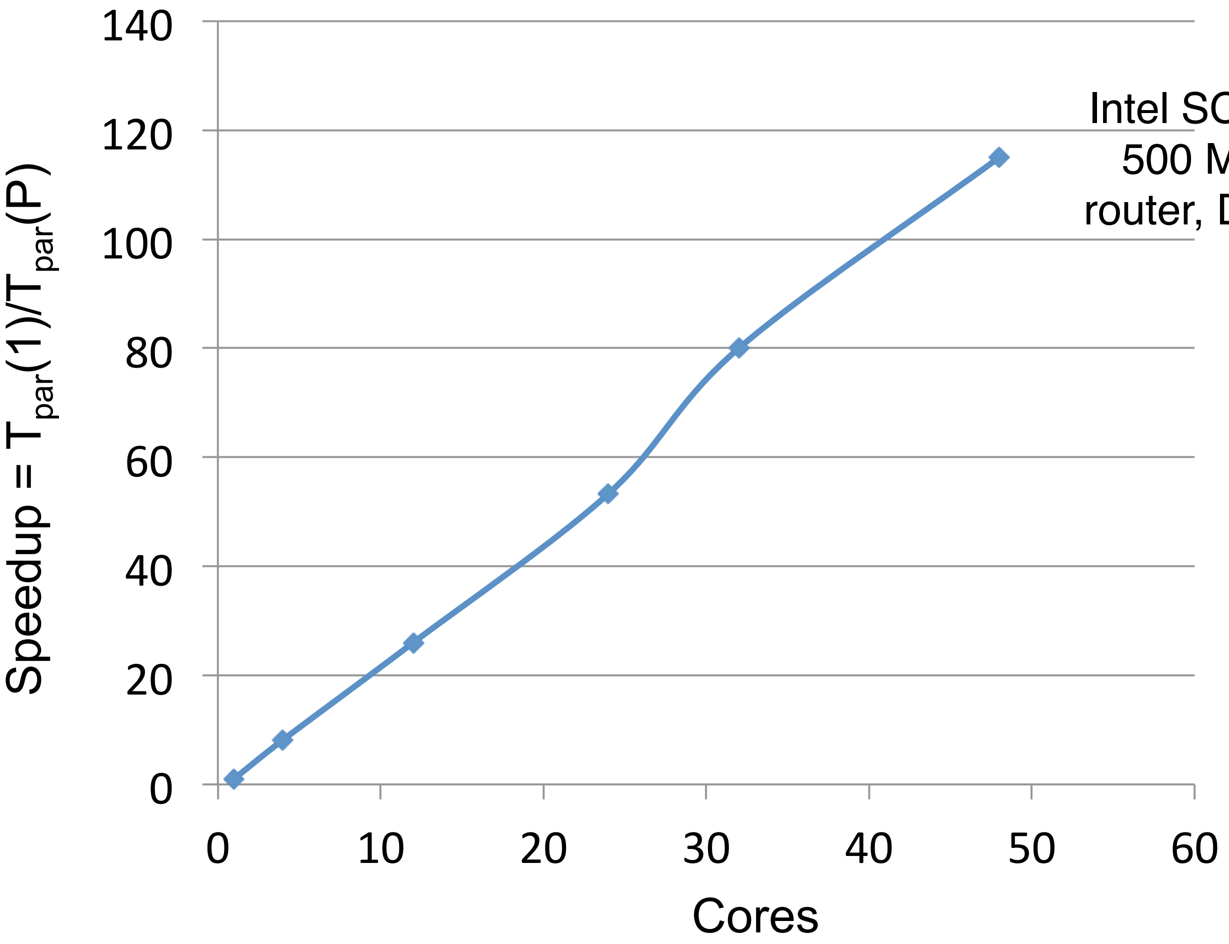
$$S(P) = \frac{Time_{seq}(1)}{Time_{par}(P)}$$

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

$$S(P) = P$$

Performance scalability

- HP Linpack benchmark, order 1000 matrix (solve a dense system of linear equations … the dense linear algebra computational pattern).

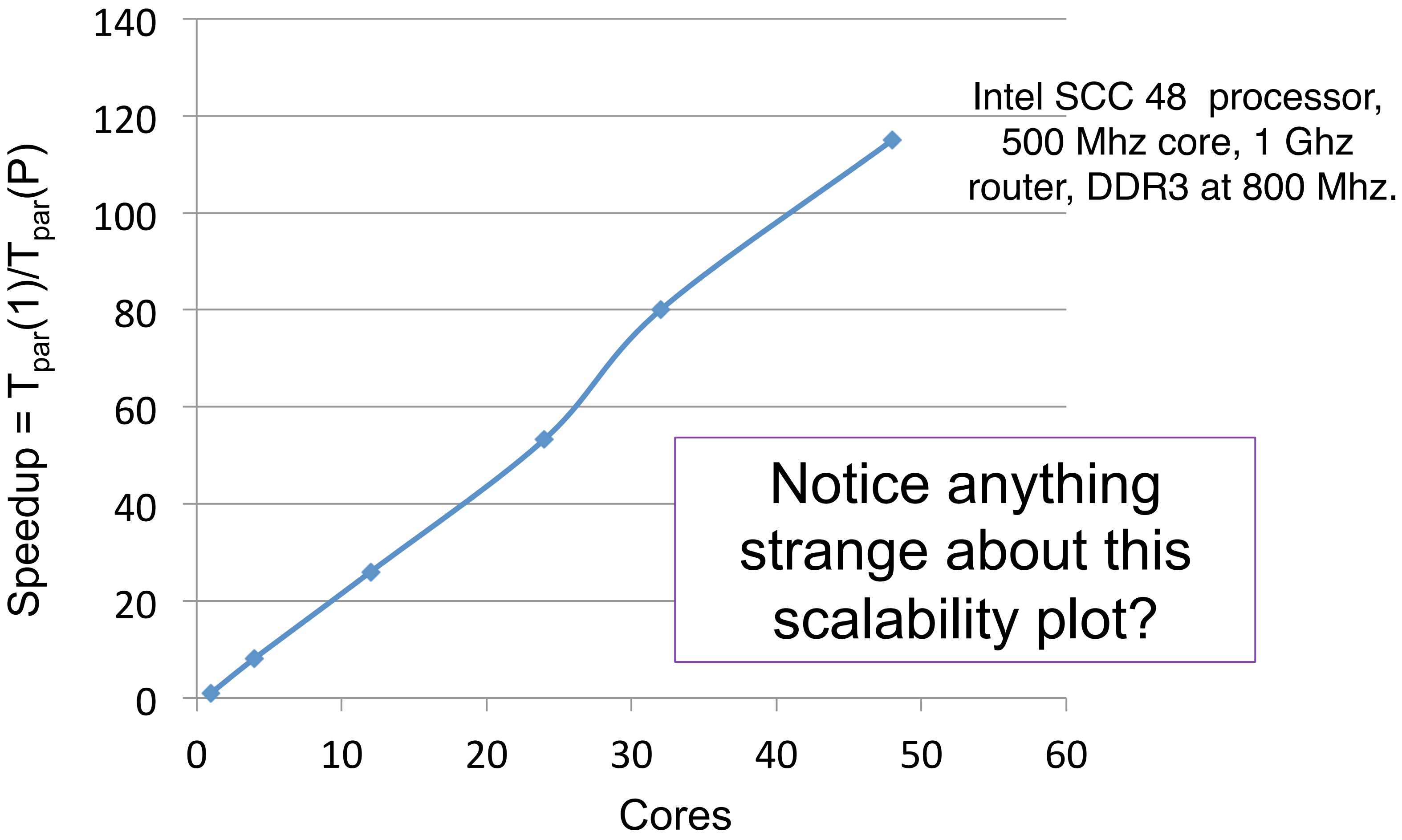


Intel SCC 48 processor, 500 Mhz core, 1 Ghz router, DDR3 at 800 Mhz.

The 48-core SCC processor: the programmer's view, T, G. Mattson, R. F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, Proceedings SC10, New Orleans 2010

Performance scalability

- HP Linpack benchmark, order 1000 matrix (solve a dense system of linear equations … the dense linear algebra computational pattern).



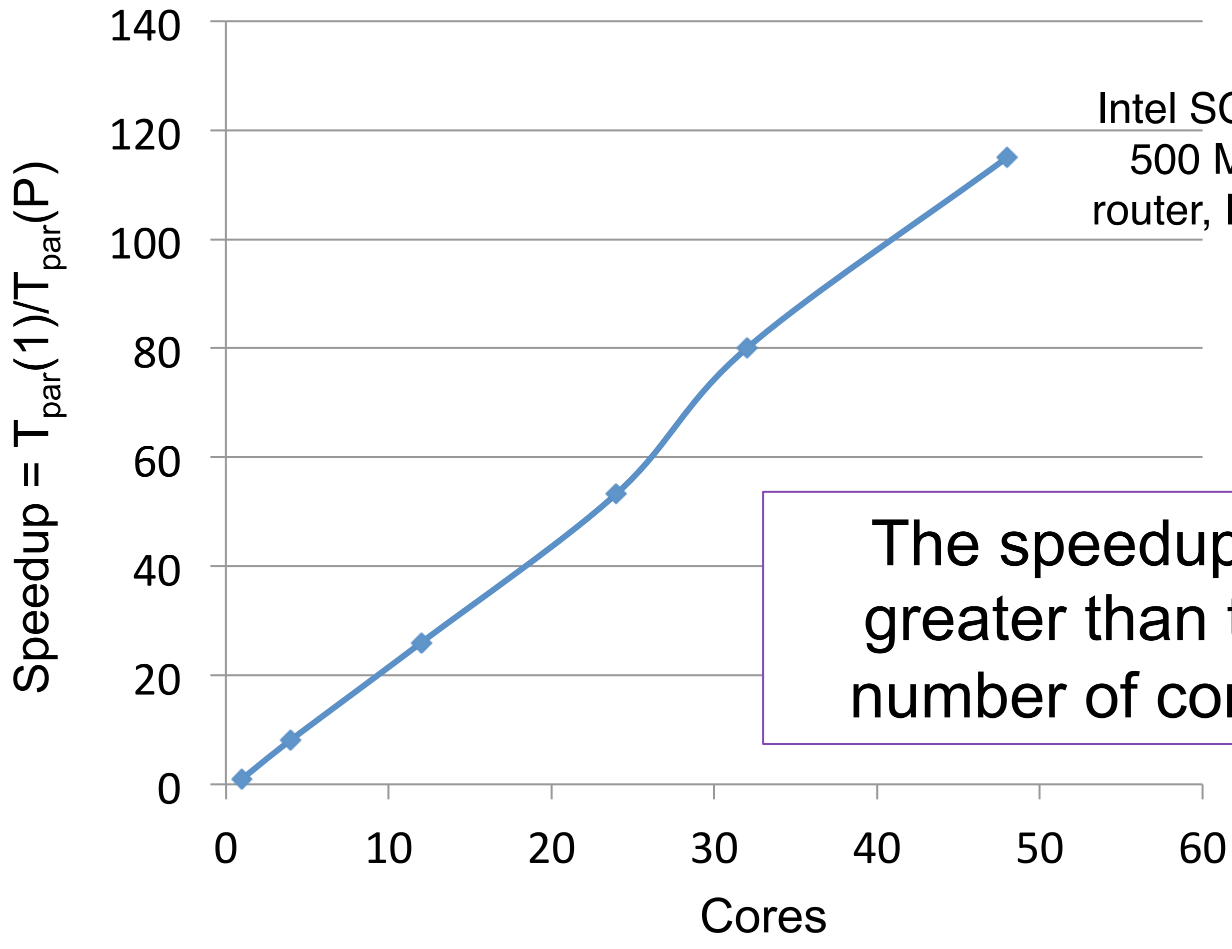
Intel SCC 48 processor, 500 Mhz core, 1 Ghz router, DDR3 at 800 Mhz.

Notice anything strange about this scalability plot?

The 48-core SCC processor: the programmer's view, T, G. Mattson, R. F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, Proceedings SC10, New Orleans 2010

Performance scalability

- HP Linpack benchmark, order 1000 matrix (solve a dense system of linear equations … the dense linear algebra computational pattern).



Intel SCC 48 processor, 500 Mhz core, 1 Ghz router, DDR3 at 800 Mhz.

The speedup is greater than the number of cores!

The 48-core SCC processor: the programmer’s view, T, G. Mattson, R. F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, Proceedings SC10, New Orleans 2010

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.

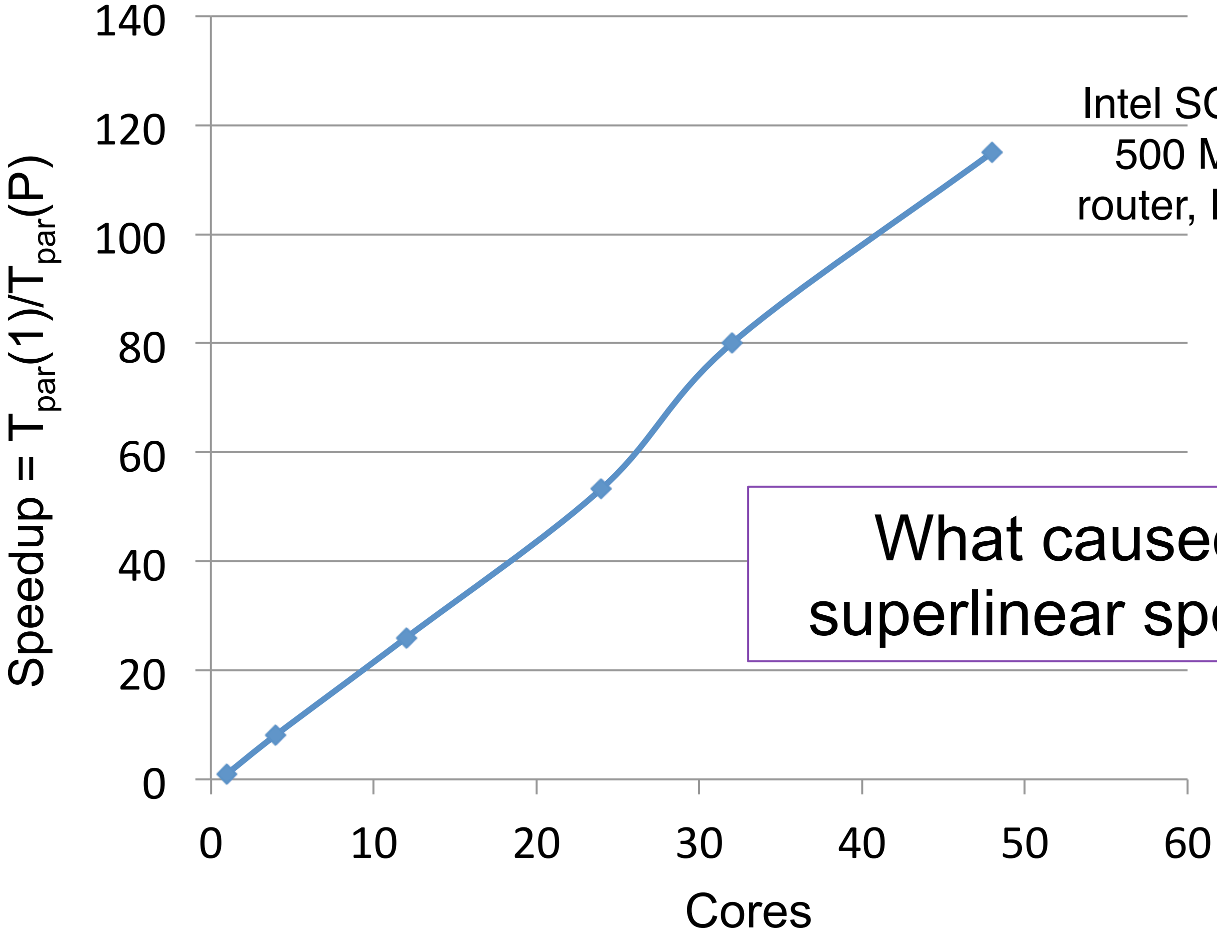
$$S(P) = P$$

- Super-linear Speedup: Speed grows faster than the number of processing elements

$$S(P) > P$$

Performance scalability

- HP Linpack benchmark, order 1000 matrix (solve a dense system of linear equations … the dense linear algebra computational pattern).



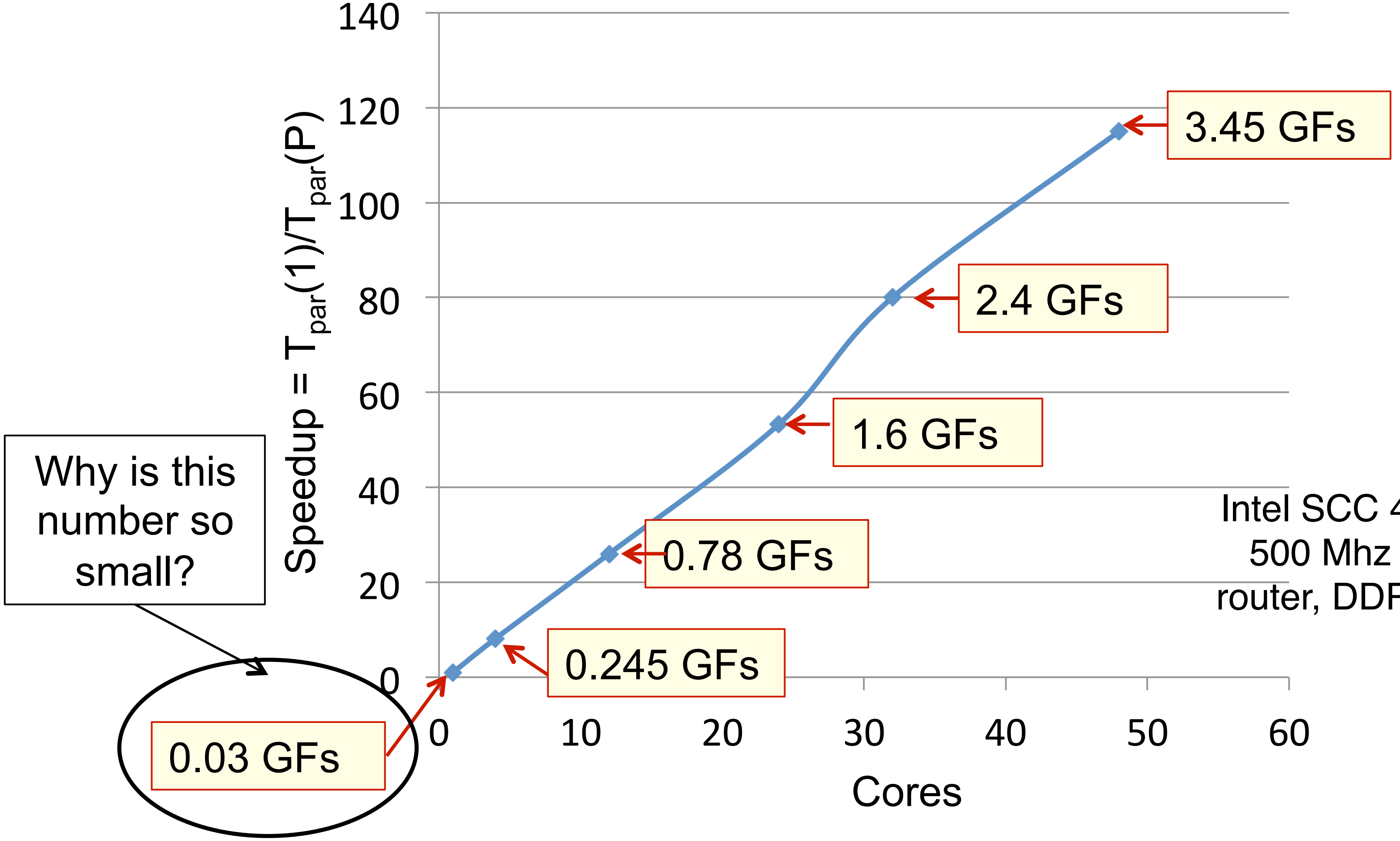
Intel SCC 48 processor, 500 Mhz core, 1 Ghz router, DDR3 at 800 Mhz.

What caused our superlinear speedup?

The 48-core SCC processor: the programmer’s view, T, G. Mattson, R. F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, Proceedings SC10, New Orleans 2010

SuperLInear Speedup

- HP Linpack benchmark, order 1000 matrix (solve a dense system of linear equations … the dense linear algebra computational pattern).

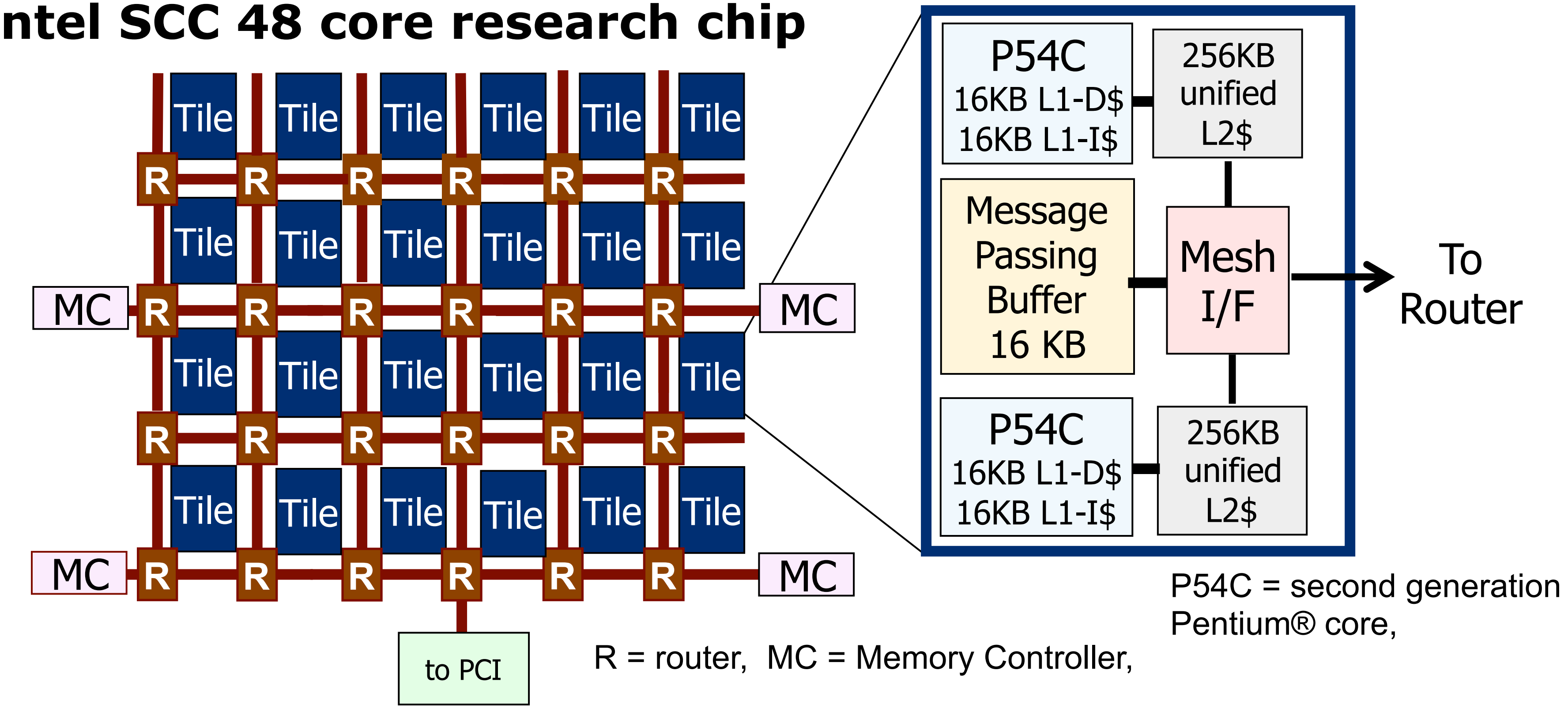


Why is this number so small?

Intel SCC 48 processor, 500 Mhz core, 1 Ghz router, DDR3 at 800 Mhz.

Why the Superlinear speedup?

- **Intel SCC 48 core research chip**



R = router, MC = Memory Controller,

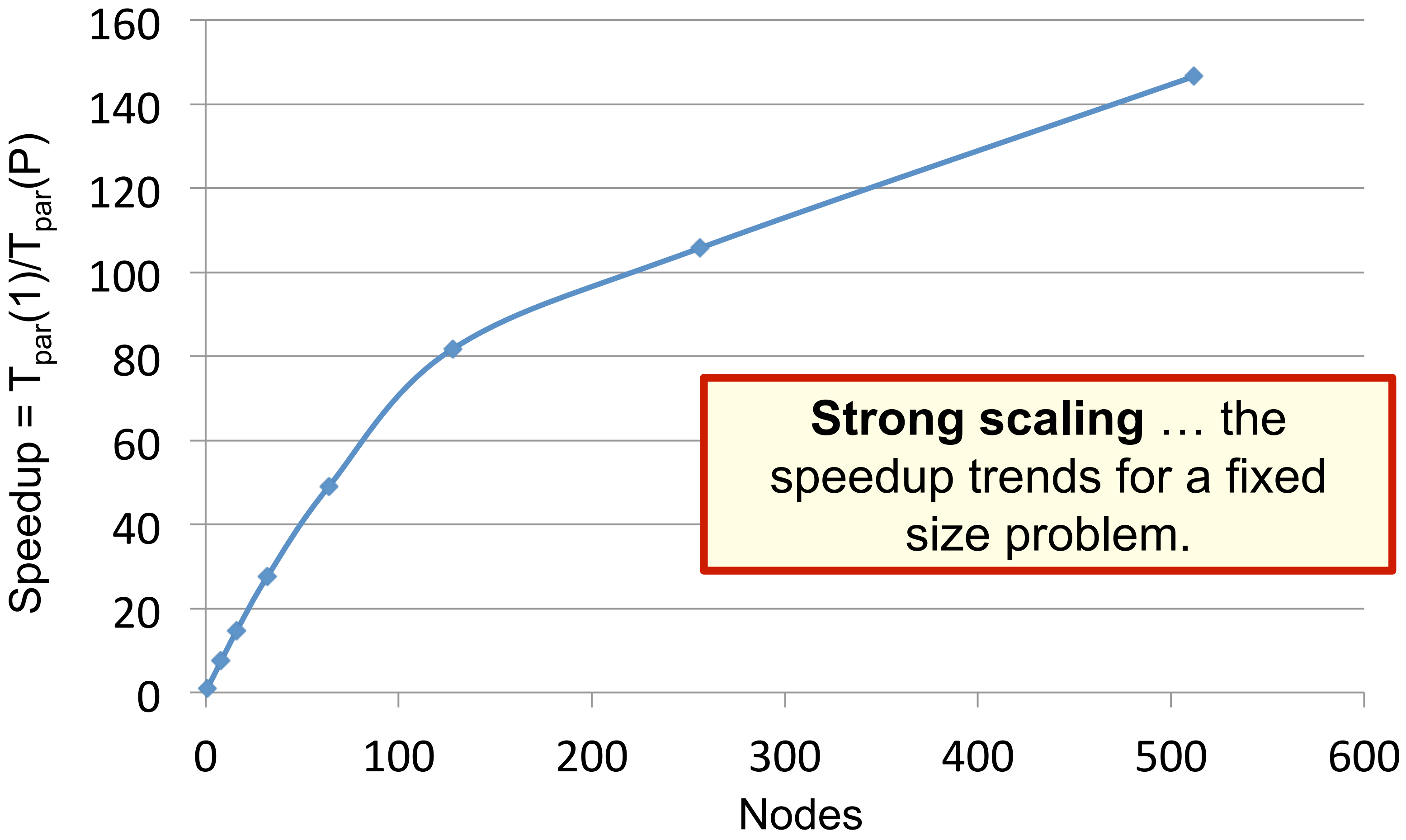
P54C = second generation Pentium® core,

- SCC caches are so small, even a small portion of our O(1000) matrices won't fit.
  - ➢ Hence the single node performance measures memory overhead.
- As you add more cores, the aggregate cache size grows.
  - ➢ Eventually the tiles of the matrices being processed fits in the caches and performance sharply increases → **superlinear speedup**.

The 48-core SCC processor: the programmer's view, T, G. Mattson, R. F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, Proceedings SC10, New Orleans 2010

A more typical speedup plot

- CHARMM molecular dynamics program running the myoglobin benchmark on an Intel Paragon XP/S supercomputer with 32 Mbyte nodes running OSF R 1.2. (The nbody computational pattern). Speedup relative to running the parallel program on one node.



Strong scaling … the speedup trends for a fixed size problem.

Porting Applications to the MP-Paragon Supercomputer: The CHARMM Molecular Dynamics program, T.G. Mattson, Intel Supercomputers User's Group meeting, 1995.

Efficiency

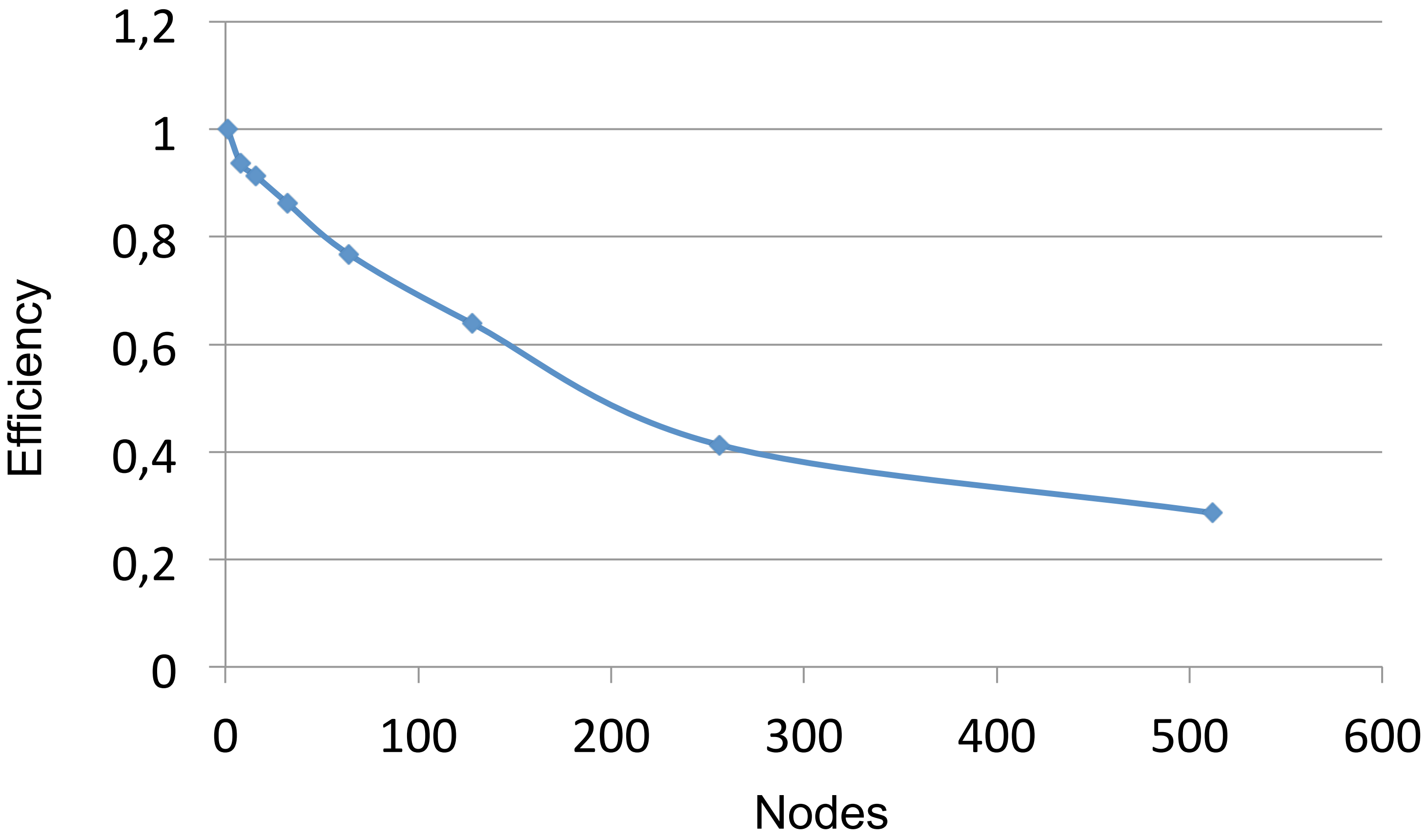
- Efficiency measures how well the parallel system's resources are being utilized.

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

- Where P is the number of nodes and T is the elapsed runtime.

Efficiency

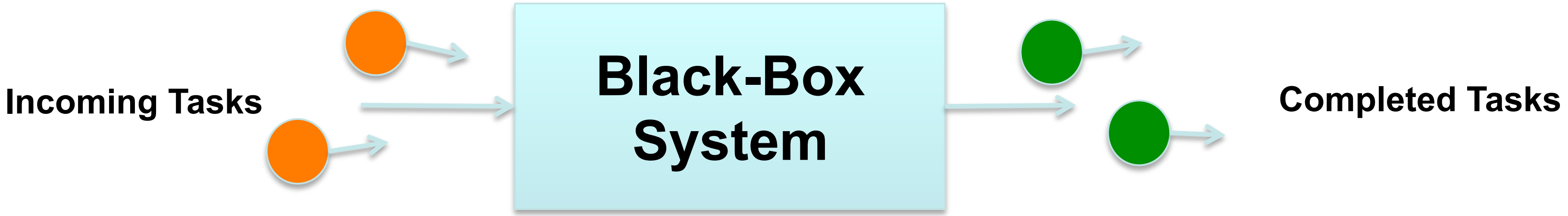
- CHARMM molecular dynamics program running the myoglobin benchmark on an Intel Paragon XP/S supercomputer with 32 Mbyte nodes running OSF R 1.2. (The nbody computational pattern). Speedup relative to running the parallel program on one node.



Porting Applications to the MP-Paragon Supercomputer: The CHARMM Molecular Dynamics program, T.G. Mattson, Intel Supercomputers User's Group meeting, 1995.

Little's Law

- Consider a system where tasks arrive periodically. The system takes some finite amount of time to execute each job.



- Suppose that the system is in Equilibrium: the average rate at which tasks arrive is equal to the average rate at which they are completed. Then, the average over time:

# tasks in the system = response time * arrival rate

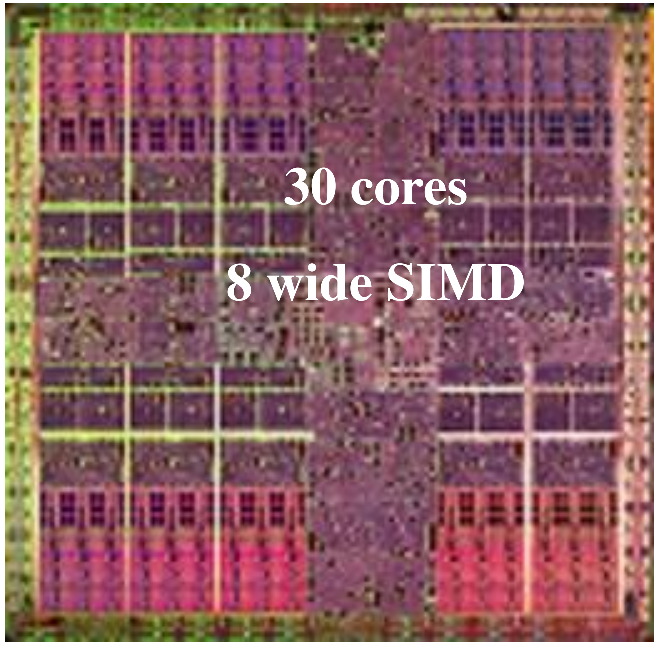
Little's Law

# tasks in the system = response time * arrival rate

- Tells us the number of "in flight" tasks we must have to keep our system busy, once we know how long tasks take to execute and the rate at which we can execute them.
- Applies in many situations:
  - # Outstanding load instrs = DRAM latency * DRAM bandwidth
  - Pipeline Depth = Instruction Latency * Pipeline Width
  - **Concurrency = latency * bandwidth**

Little’s law example ...

- Consider an NVIDIA GTX285 GPU.
  - Bandwidth to DRAM, 128 byte/cycle
  - Latency to DRAM, 500 cycles
  - An OpenCL work-item on a GTX285 issues 4 byte memory requests
- How many outstanding memory requests must be sustained to fully utilize the chip.
- What does this suggest concerning how many ***work-items*** you need in your program to keep this utilized at peak clock-rate?



NVIDIA GTX285 (Tesla C1060)

- Little’s law … concurrency = latency * bandwidth
  - Key … pay attention to units. Requests per clock cycle is what I need.
  - (128 bytes/cycle)*(1 request/4 bytes) = 32 requests/cycle
  - Concurrency = 500 cycles * 32 requests/cycle = 16000 requests
- In other words, you need 16 K threads to fully saturate this GPU.

Source: GPU Computing Gems jade edition, Wen Mei Hwu

Granularity

- Granularity is the ratio of compute time to communication time
  - Hardware: compute rate vs. communication rate … also expressed as flops relative to memory latency
  - Software: How much computation you need to compensate for parallel overhead.

Key rule: Granularity demanded by software must be met or bettered by hardware. Fine grained applications do not run well on coarse grained systems.

Parallel Architecture Granularity

- An NVIDIA GTX285 GPU.
  - 30 1.3 GHz Nvidia Streaming SIMD cores each with 8-wide SIMD (240 “CUDA cores”)
  - 2.5 DP GFLOPS per **ONE core**
  - Communication through shared memory
  - Latency to DRAM, 500 cycles
- A Linux Cluster
  - Many Linux PC’s.
    - Intel Core 2 Q6600 Kensfield, **4 core**, 2.4 GHz. 38 GFLOP DP peak
  - Communication over 1 gigbit ethernet
  - Communication latency ~ 40 microseconds (96 thousand cycles)
- A multiprocessor PC
  - 2 sockets each with a CPU
    - Intel Core 2 Q6600 Kensfield, **4 core**, 2.4 GHz. 38 GFLOP DP peak
  - Communication through shared memory
  - Latency to DRAM, 200 .. to L3 40 cycles

Consider how many FLOPS your algorithm needs to balance a single communication

~1000 DP FLOPS

1.5 * 10^6 DP FLOPS

3200 DP FLOPS
~150 DP FLOPS
between cores
sharing an L3

Outline

- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
    - Basic definitions: Parallelism and Concurrency
    - Notions of parallel performance
    - ➡ The limits of scalability
    - Sources of parallel overhead
- An introduction to parallel hardware
- Software for parallel systems: key design patterns
- Closing comments

Amdahl's Law: History

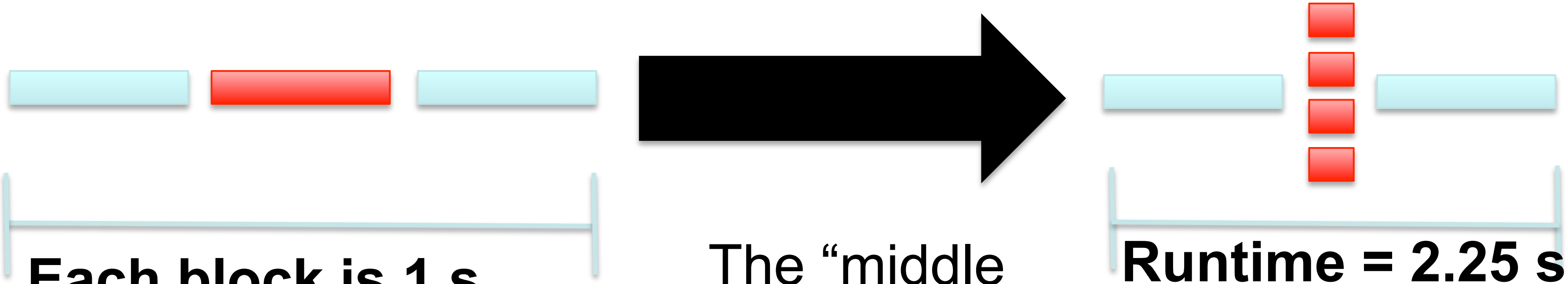
- Gene Amdahl was a computer architect in the 1960's at IBM
- In 1967, refuted the idea that parallel computing was a practical path to improving program performance.
- Example: Compare these two systems

IBM System 360, ca. 1964

- The IBM System 360:
  - A single-processor machine, running at 16 MHz.
  - 1 FP addition per 60 ns cycle, and 1 FP mul in ~10 60 ns cycles, and execute multiple instructions simultaneously
- ILLIAC IV:
  - “The first Supercomputer” … installed at NASA Ames in 1975.
  - 256 processors … could perform 256 FP adds in 240 ns.

Amdahl's Law

- Clearly, the ILLIAC will run programs much faster than the S/360: It has 60x higher instruction throughput!
  - ... if you always have 256 independent instructions
- Amdahl argued that large portions of many programs are ***not*** parallelizable. Parallel hardware does not help serial code:



Each block is 1 s …
Runtime = 3 s

The "middle second" runs perfectly parallel on 4 threads

Runtime = 2.25 s

Amdahl's Law

- What is the maximum speedup you can expect from a parallel program?
- Consider a sequential program with runtime:

$$Time\downarrow Seq \downarrow$$

- We can think of this program as consisting of two parts … one that can benefit from multiple processing elements (parallel) and a second part that is fundamentally serial.
- The runtime is therefore:

$$Timeseq = Time\downarrow Serial + Time\downarrow parallelizable$$

- We can express this in terms of a fraction of the program that is serial and a fraction of the program that is parallel or

$$Timeseq = serial\_fraction * Timeseq + parallel\_fraction * Time_{seq}$$

Amdahl's Law

- ❖ If we run the program on P processing elements and assume linear speedup, then our time for the parallel program becomes:

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

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

$$S(P)= Time{\downarrow}seq\ \ /Time{\downarrow}par\,(P)\ =Time{\downarrow}seq\,/(\alpha{+}1{-}\alpha/P\ )\overset{0}{*}$$

$$Time{\downarrow}seq\ \ =1/(\alpha{+}1{-}\alpha/P\,)$$

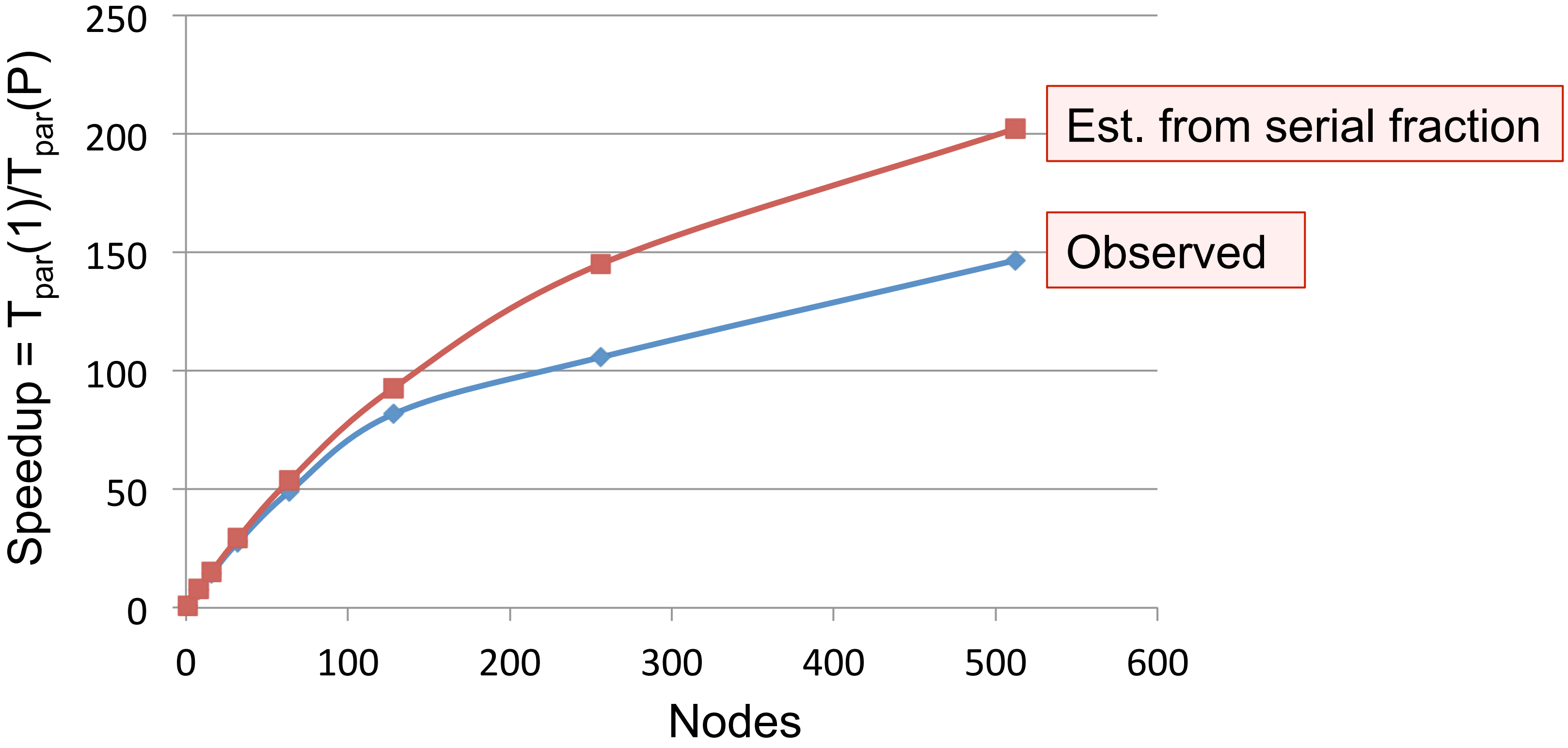
- If you had an unlimited number of processors: $\lim{┬}P{\to}\infty$ $1{-}\alpha/P\ =0$

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

Amdahl's Law and the CHARMM MD program

- We Profiled CHARMM running on the Paragon XPS to find the time spent in code that was not parallelized … concluded that CHARMM has a serial fraction of ~0.003.

- The maximum possible speedup is: S= 1/0.003 = 333



What if the problem size grows

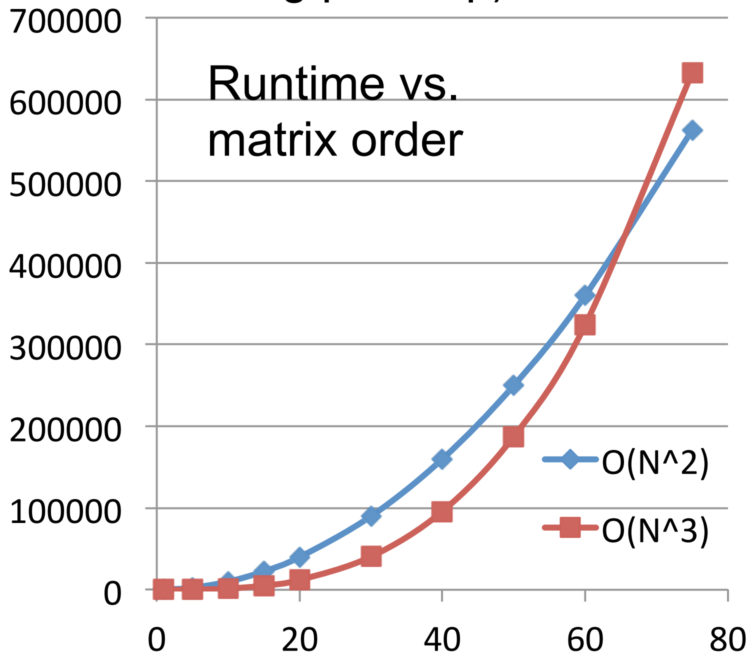
- Consider the dense linear algebra computational pattern (which we will cover in much more detail later).
- A key feature is that operations between matrices (such as LU factorization or matrix multiplication) scale as the cube of the order of the matrix.
- Assume we can parallelize the linear algebra operation ($O(N^3)$) but not the loading of the matrices from memory ($O(N^2)$). How does the serial fraction vary with matrix order (assume loading from memory is much slower than a floating point op).

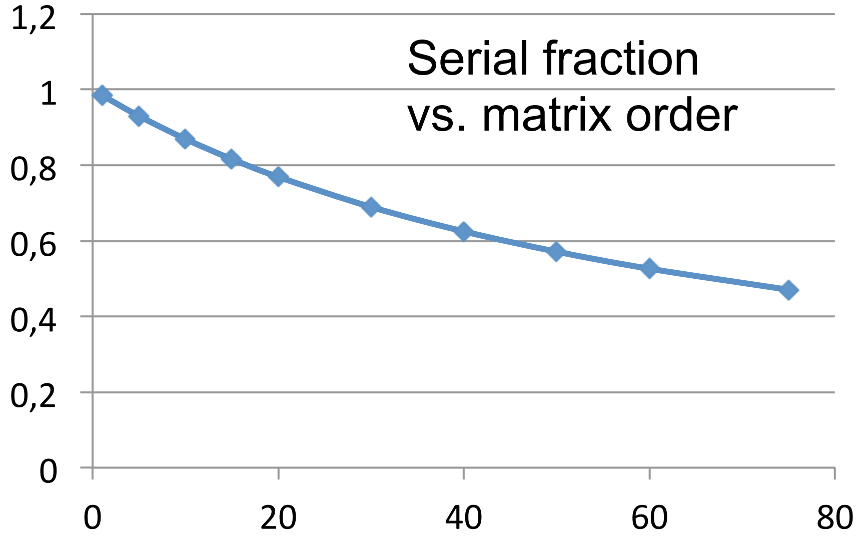
What would plots of runtime vs. problem size look like for the N squared and N cubed terms?

What would plots of serial fraction vs. problem size look like for the N squared and N cubed terms?

What if the problem size grows

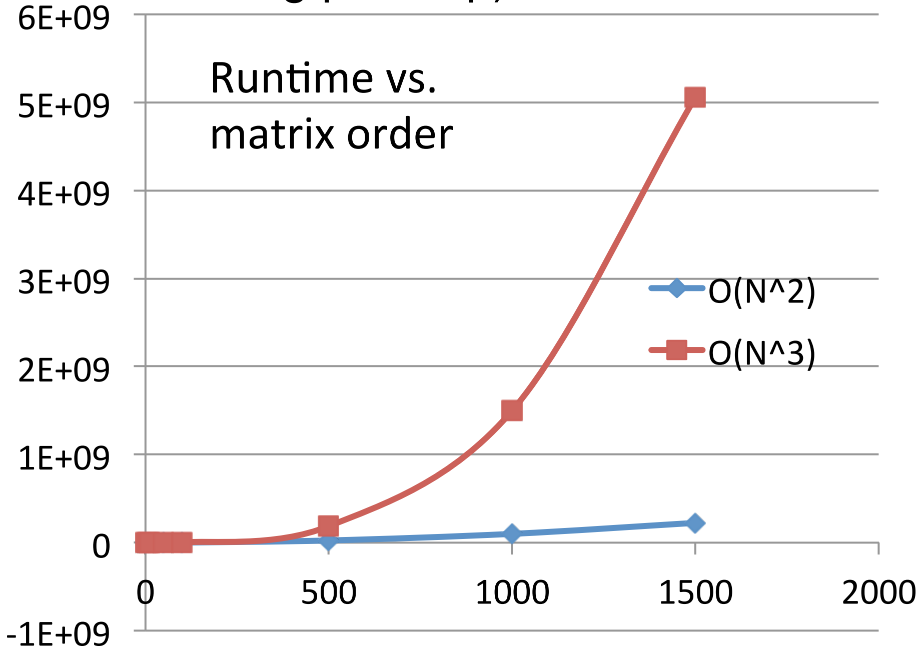
- Consider the dense linear algebra design pattern (which we will cover in much more detail later).
- A key feature is that operations between matrices (such as LU factorization or matrix multiplication) scale as the cube of the order of the matrix.
- Assume we can parallelize the linear algebra operation ($O(N^3)$) but not the loading of the matrices from memory ($O(N^2)$). How does the serial fraction vary with matrix order (assume loading from memory is much slower than a floating point op).

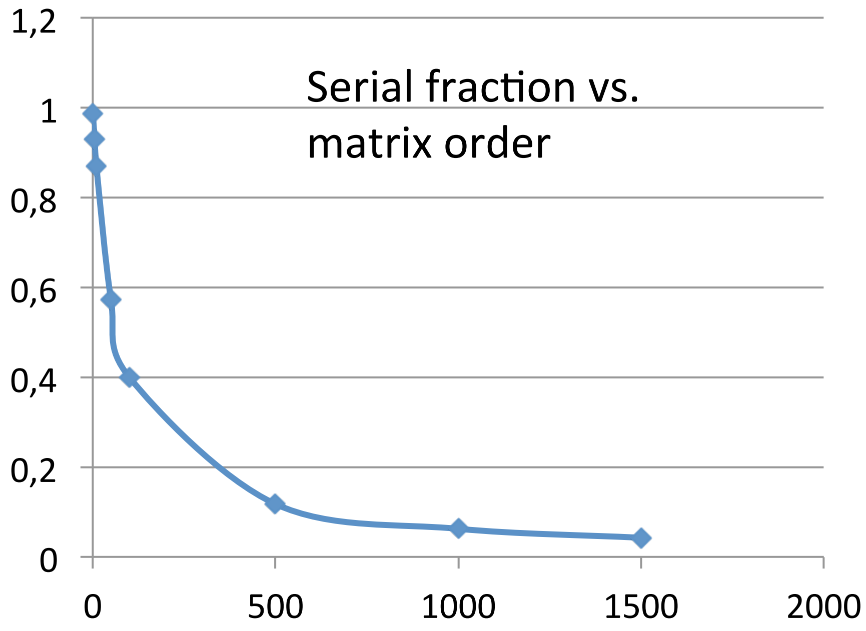




What if the problem size grows

- Consider the dense linear algebra design pattern (which we will cover in much more detail later).
- A key feature is that operations between matrices (such as LU factorization or matrix multiplication) scale as the cube of the order of the matrix.
- Assume we can parallelize the linear algebra operation ($O(N^3)$) but not the loading of the matrices from memory ($O(N^2)$). How does the serial fraction vary with matrix order (assume loading from memory is much slower than a floating point op).





For much larger Matrix orders …

Weak Scaling: a response to Amdhal

- Gary Montry and John Gustafson (1988, Sandia National Laboratories) observed that for many problems the serial fraction of a function of the problem size (N) decreases:

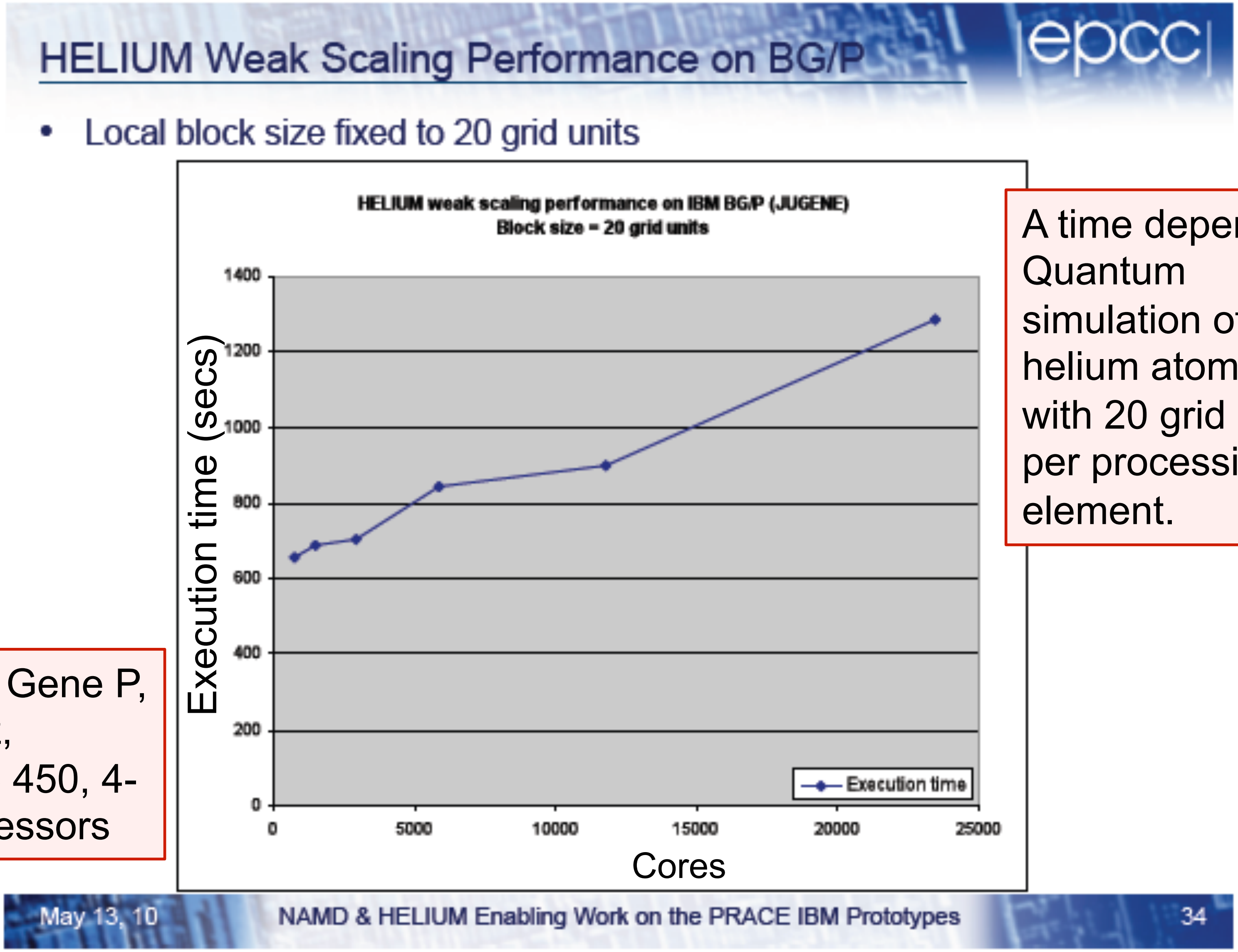
$$S(P,N) = \frac{T_{seq}(1)}{(\alpha(N) + \frac{1-\alpha(N)}{P}) * T_{seq}(1)}$$

$$\lim_{N \to N_{l\arg e}} \alpha(N) = 0$$

$$S(P, N_{l\arg e}) \to P$$

- In other words … if parallelizable computations asymptotically dominate the runtime, then you can increase a problem size until limitations due to Amdahl's law can be ignored. This is an easier form of scalability for a programmer to meet … so its called "weak scaling":
  - **Weak Scaling:** Performance of an application when the problem size increases with the number of processors (fixed size problem per node)

Example of weak scaling



HELIUM Weak Scaling Performance on BG/P

- Local block size fixed to 20 grid units

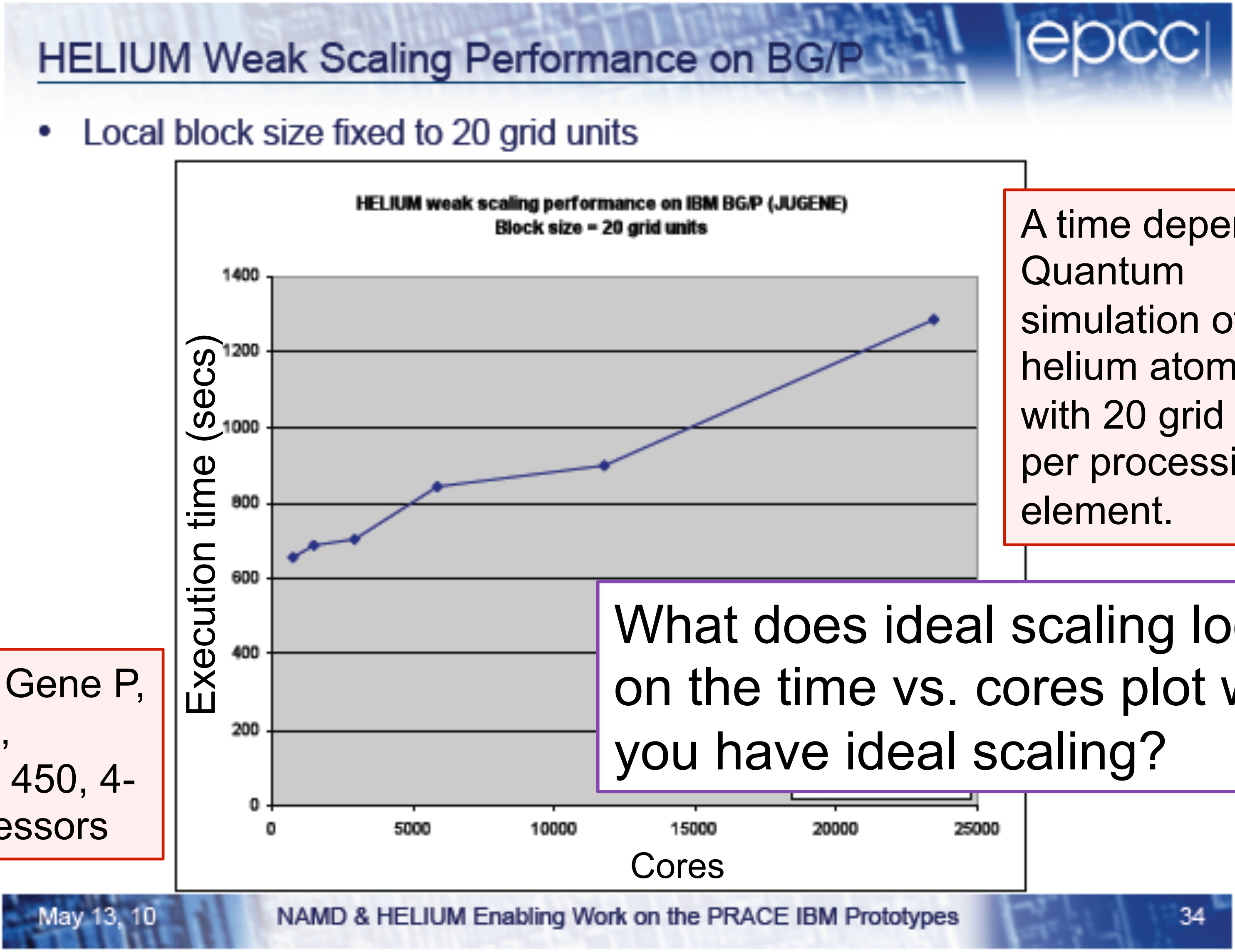
May 13, 10 NAMD & HELIUM Enabling Work on the PRACE IBM Prototypes 34

A time dependent Quantum simulation of helium atoms with 20 grid units per processing element.

IBM Blue Gene P, 0.85 GHz, PowerPC 450, 4-way processors

http://www.spscicomp.org/ScicomP16/presentations/PRACE_ScicomP.pdf

Example of weak scaling



HELIUM Weak Scaling Performance on BG/P

epcc

- Local block size fixed to 20 grid units

May 13, 10 NAMD & HELIUM Enabling Work on the PRACE IBM Prototypes 34

A time dependent Quantum simulation of helium atoms with 20 grid units per processing element.

What does ideal scaling look on the time vs. cores plot when you have ideal scaling?

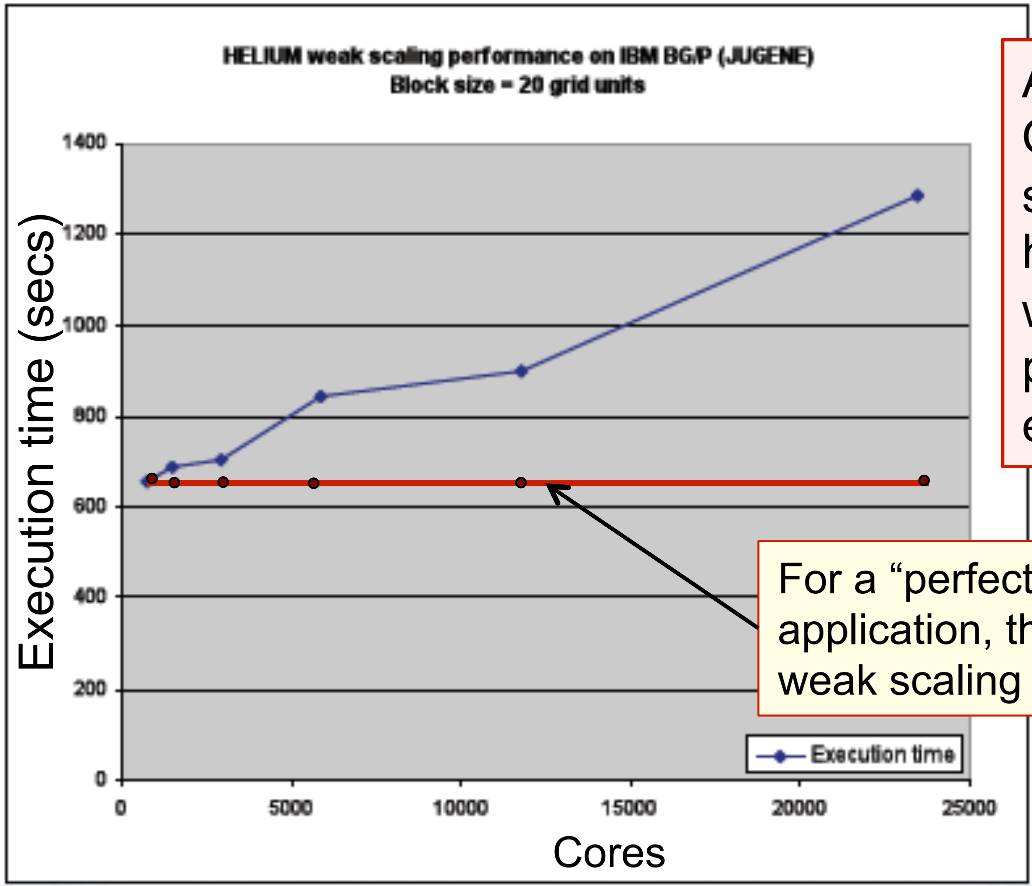
IBM Blue Gene P, 0.85 GHz, PowerPC 450, 4-way processors

http://www.spscicomp.org/ScicomP16/presentations/PRACE_ScicomP.pdf

Example of weak scaling

HELIUM Weak Scaling Performance on BG/P

- Local block size fixed to 20 grid units



A time dependent Quantum simulation of helium atoms with 20 grid units per processing element.

For a "perfectly scalable" application, the trend line for weak scaling should be flat.

IBM Blue Gene P, 0.85 GHz, PowerPC 450, 4-way processors

May 13, 10 NAMD & HELIUM Enabling Work on the PRACE IBM Prototypes 34

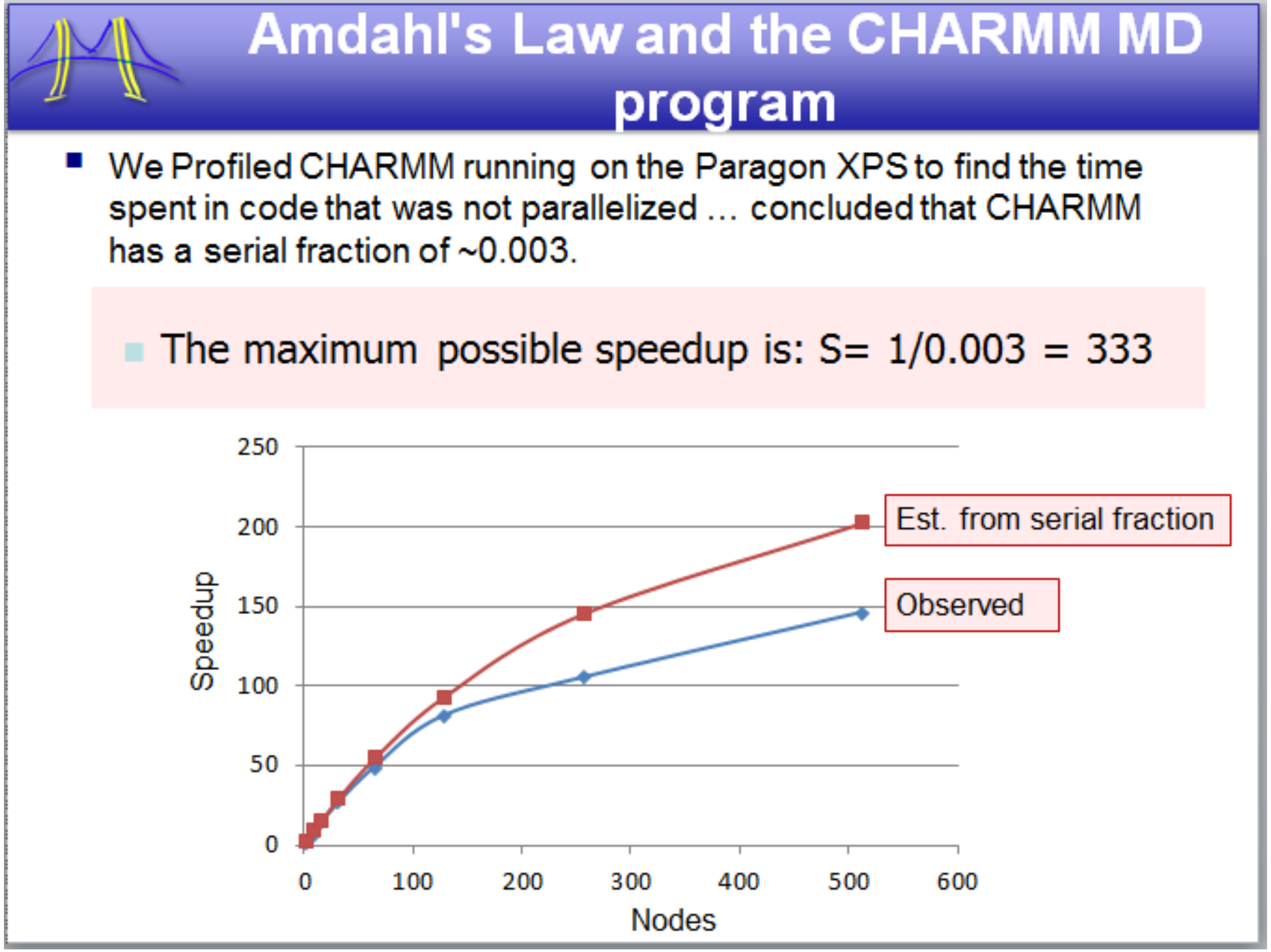
http://www.spscicomp.org/ScicomP16/presentations/PRACE_ScicomP.pdf

Outline

- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
  - Basic definitions: Parallelism and Concurrency
  - Notions of parallel performance
  - The limits of scalability
  - ➡ Sources of parallel overhead
- An introduction to parallel hardware
- Software for parallel systems: key design patterns
- Closing comments

Limitations to scalability

- Remember the speedup plot we discussed earilier?



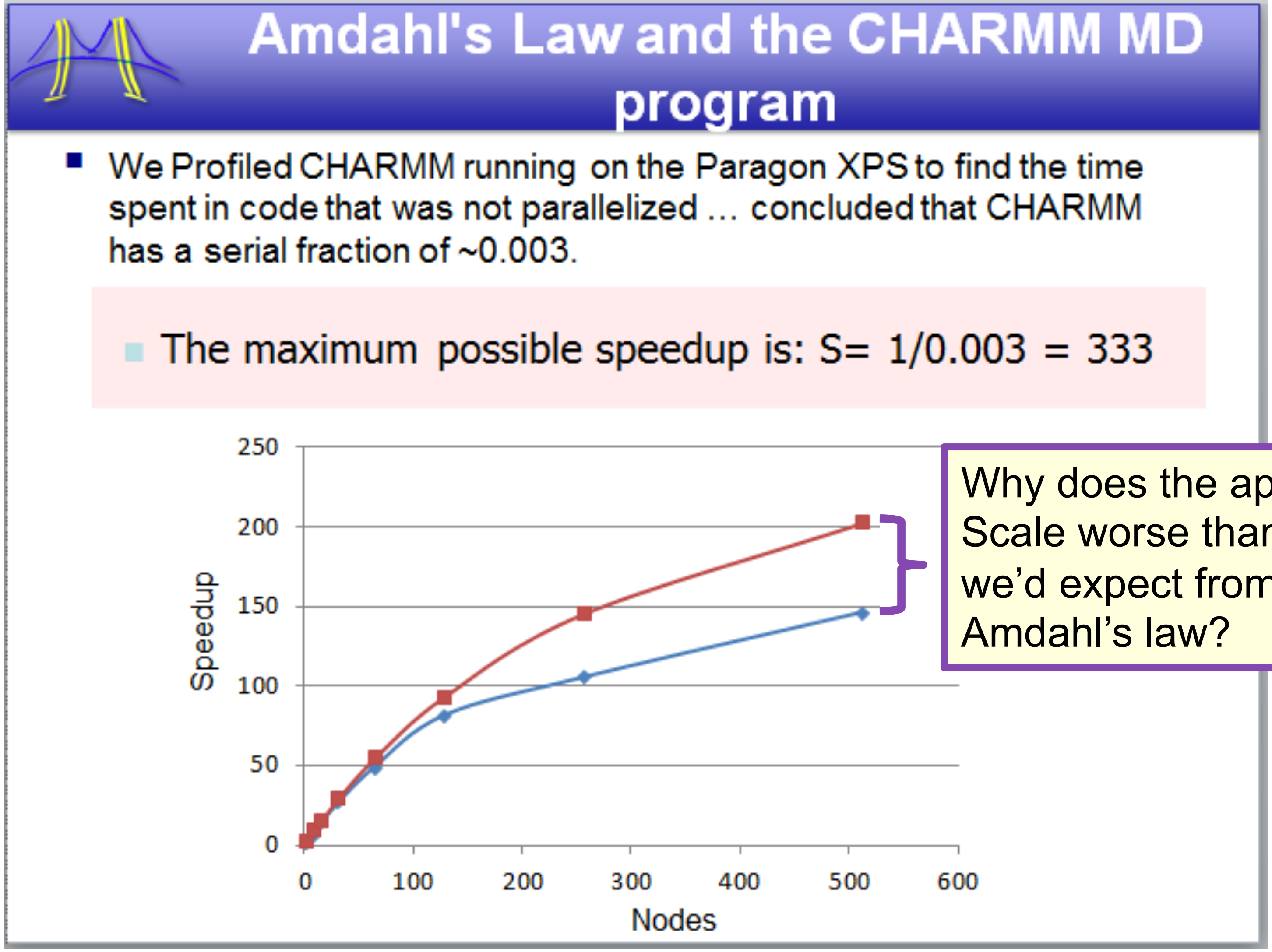
Amdahl's Law and the CHARMM MD program

- We Profiled CHARMM running on the Paragon XPS to find the time spent in code that was not parallelized … concluded that CHARMM has a serial fraction of ~0.003.

- The maximum possible speedup is: S= 1/0.003 = 333

Limitations to scalability

- Remember the speedup plot we discussed from last time?



Amdahl's Law and the CHARMM MD program

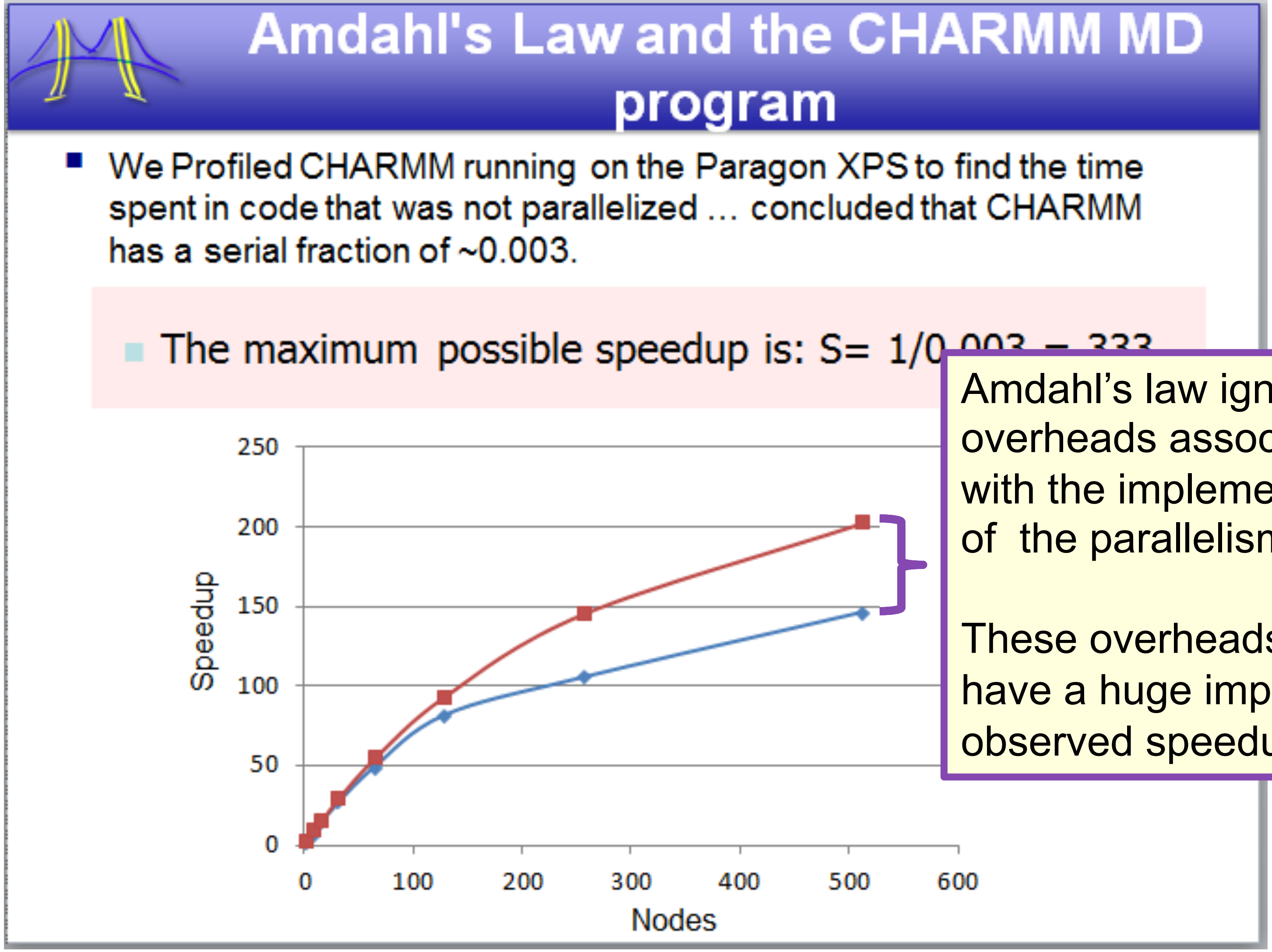
- We Profiled CHARMM running on the Paragon XPS to find the time spent in code that was not parallelized … concluded that CHARMM has a serial fraction of ~0.003.

  - The maximum possible speedup is: S= 1/0.003 = 333

Why does the app. Scale worse than we'd expect from Amdahl's law?

Limitations to scalability

- Remember the speedup plot we discussed from last time?



Amdahl's Law and the CHARMM MD program

- We Profiled CHARMM running on the Paragon XPS to find the time spent in code that was not parallelized … concluded that CHARMM has a serial fraction of ~0.003.

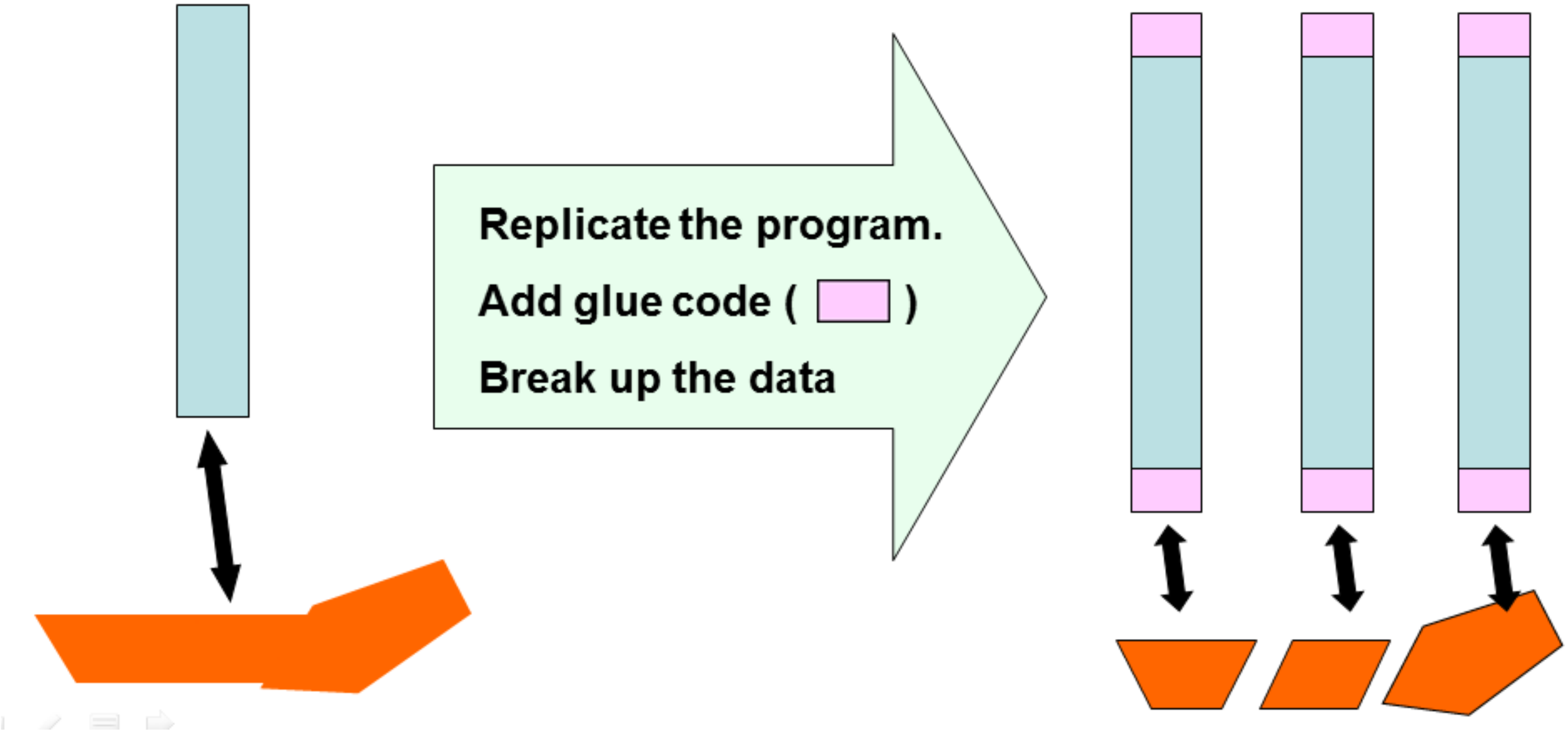
- The maximum possible speedup is: S= 1/0.003 = 333

Amdahl’s law ignores overheads associated with the implementation of the parallelism.

These overheads may have a huge impact on observed speedups.

Parallel overheads: The algorithmic structure of many HPC codes (part 1)

- A large fraction of HPC applications (such as CHARMM) use a message passing notation with the Single Program Multiple Data or SPMD design pattern.



Replicate the program.

Add glue code (▭)

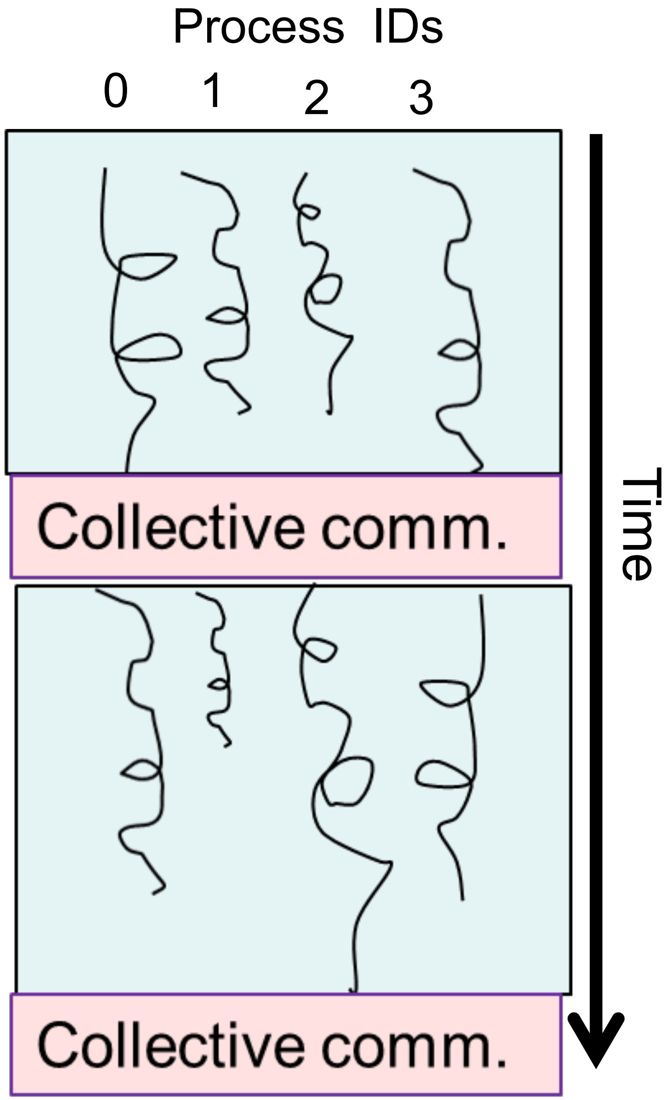
Break up the data

Original program

Parallel program

Parallel overheads: The algorithmic structure of many HPC codes (part 2)

- And many SPMD programs use an additional simplification … “**Bulk Synchronous Processing**”.
  - Each process maintains a local view of the global data
  - A problem is broken down into phases each composed of two subphases:
    - Compute on local view of data (the “squiggles” in the figure)
    - Communicate to update global view on all processes (collective communication).
  - Continue phases until complete

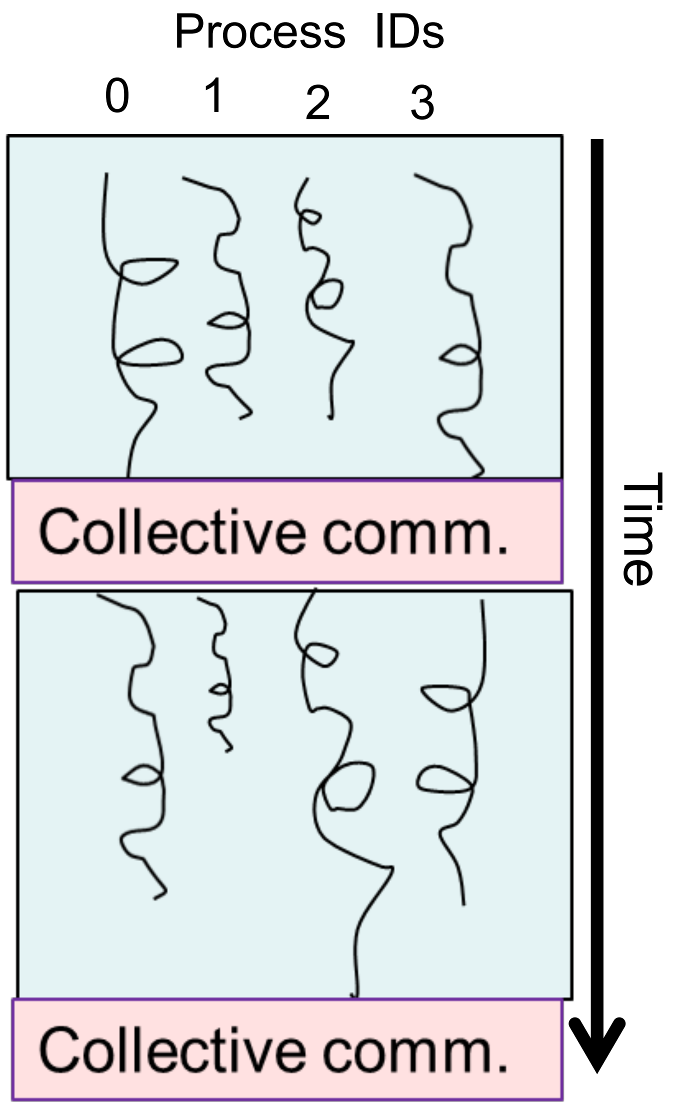


Parallel overheads with the Bulk Synchronous Processing pattern

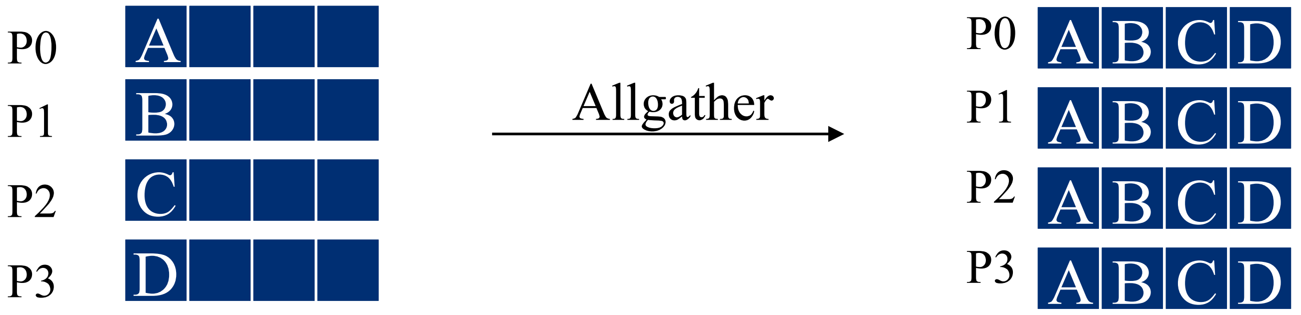
- Two major sources of parallel overhead:

1. Load imbalance: the slowest process determines when everyone is done. Time waiting for other processes to finish (i.e. unequal lengths of the "squiggles" in the figure) is time wasted.

2. Communication overhead: A cost only incurred by the parallel program. Grows with the number of processes for collective comm.



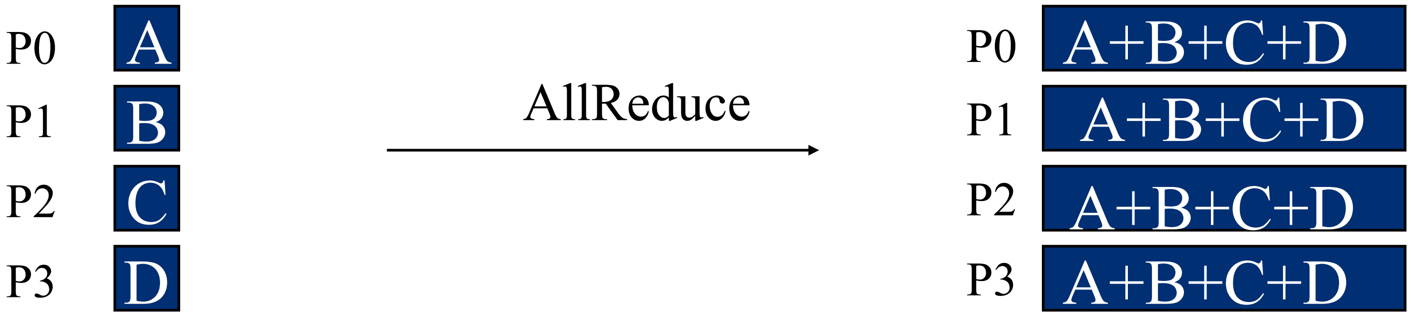
More Collective Data Movement



P0 A
P1 B
P2 C
P3 D

Allgather

P0 A B C D
P1 A B C D
P2 A B C D
P3 A B C D



P0 A
P1 B
P2 C
P3 D

AllReduce

P0 A+B+C+D
P1 A+B+C+D
P2 A+B+C+D
P3 A+B+C+D

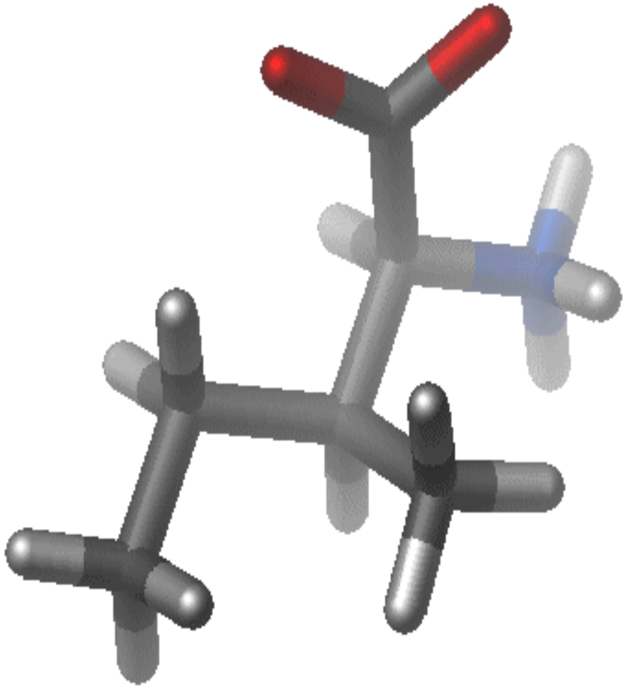
Source: CS267 Lecture 7

Molecular dynamics

- Models motion of atoms in molecular systems by solving Newton's equations of motion:

$$\vec{M}\frac{d^2\vec{r}}{dt^2} = -\nabla U(\vec{r}) = F(\vec{r})$$

- The potential energy, U(r), is divided into two parts:
  - Bonded terms – Groups of atoms connected by chemical bonds.
  - Non-bonded terms – longer range forces (e.g. electrostatic).
    - An **N-body problem** … i.e. every atom depends on every other atom, an $O(N^2)$ problem.



Bonds, angles and torsions

Source: Izaguirre, Ma and Skeel, SAC'03 slides, March 10 2003

Molecular dynamics simulation

We used a cutoff method … the potential energy drops off quickly so atoms beyond a neighborhood can be ignored in the nonbonded force calc.

```
real atoms(3,N)
real force(3,N)
int neighbors(MX,N)
// Every PE has a copy of atoms and force
loop over time steps
    parallel loop over atoms
      Compute neighbor list (for my atoms)
      Compute nonbonded forces (my atoms and neighbors)
      Barrier
      All reduce (Sum force arrays, each PE gets a copy)
      Compute bonded forces (for my atoms)
      Integrate to Update position (for my atoms)
      All_gather(update atoms array)
    end loop over atoms
end loop
```

Molecular dynamics simulation

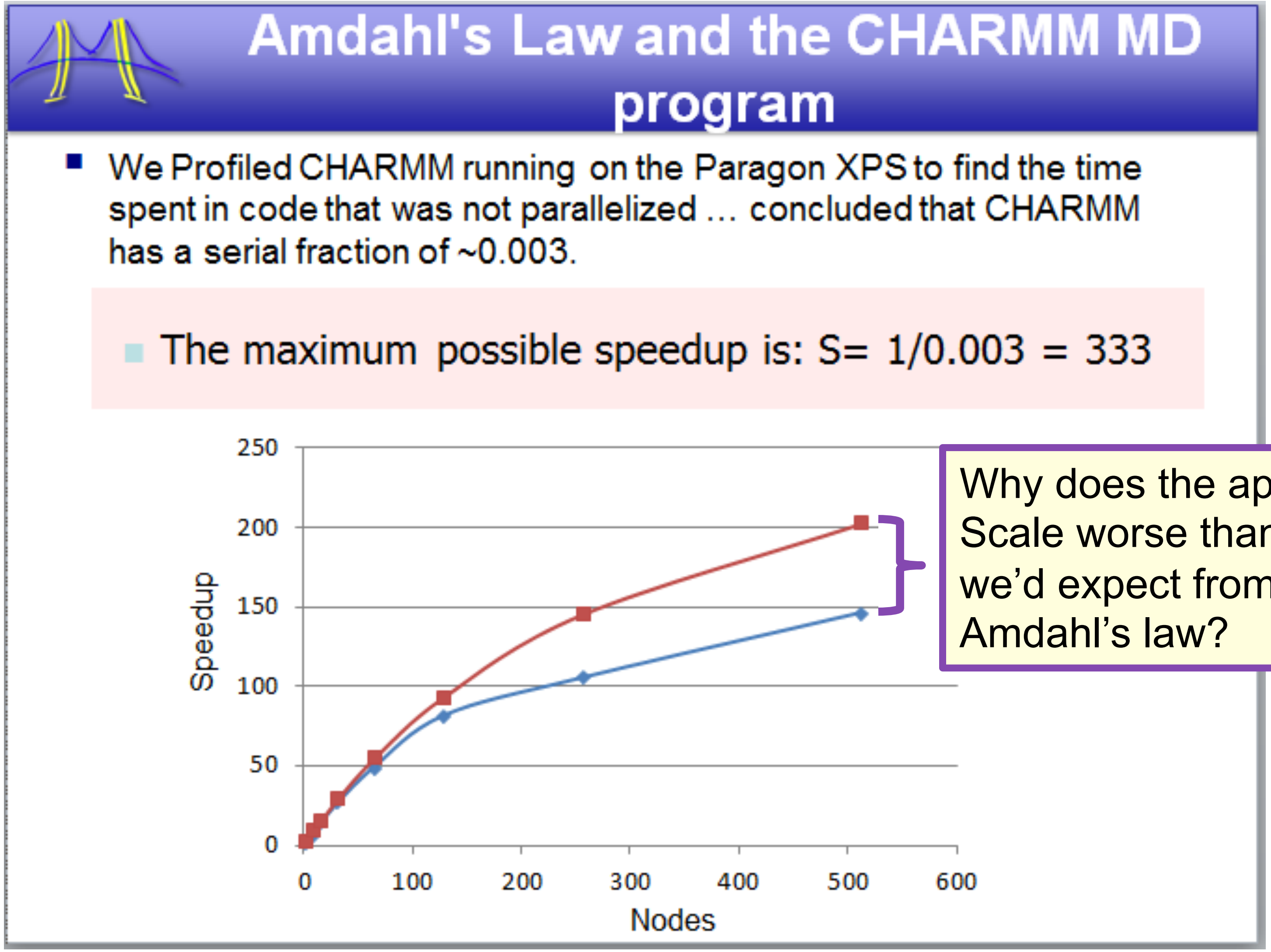
```
real atoms(3,N)
real force(3,N)
int neighbors(MX,N)   //MX = max neighbors an atom may have

// Every PE has a copy of atoms and force
loop over time steps
    parallel loop over atoms
      Compute neighbor list (for my atoms)
      Compute long range fo                 nd neighbors)
      Barrier
      All reduce (Sum force arrays,          copy)
      Compute bonded forces (for my
      Integrate to Update position
      All_gather(update atoms array)
    end loop over atoms
end loop
```

synchronization

Collective Communication

Limitations to scalability

- Remember the speedup plot we discussed from last time?



Amdahl's Law and the CHARMM MD program

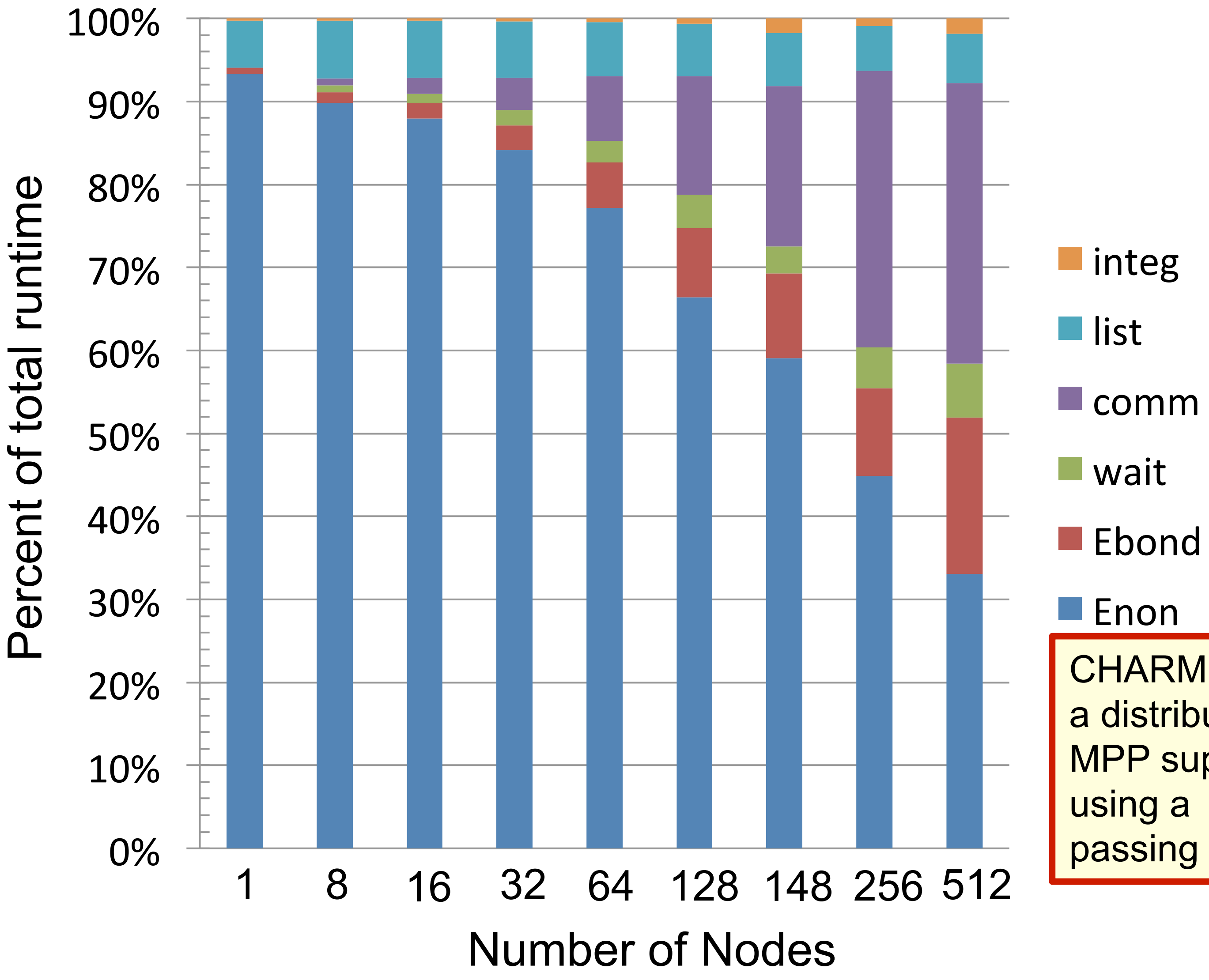
- We Profiled CHARMM running on the Paragon XPS to find the time spent in code that was not parallelized … concluded that CHARMM has a serial fraction of ~0.003.

- The maximum possible speedup is: S= 1/0.003 = 333

Why does the app. Scale worse than we'd expect from Amdahl's law?

CHARMM Myoglobin Benchmark

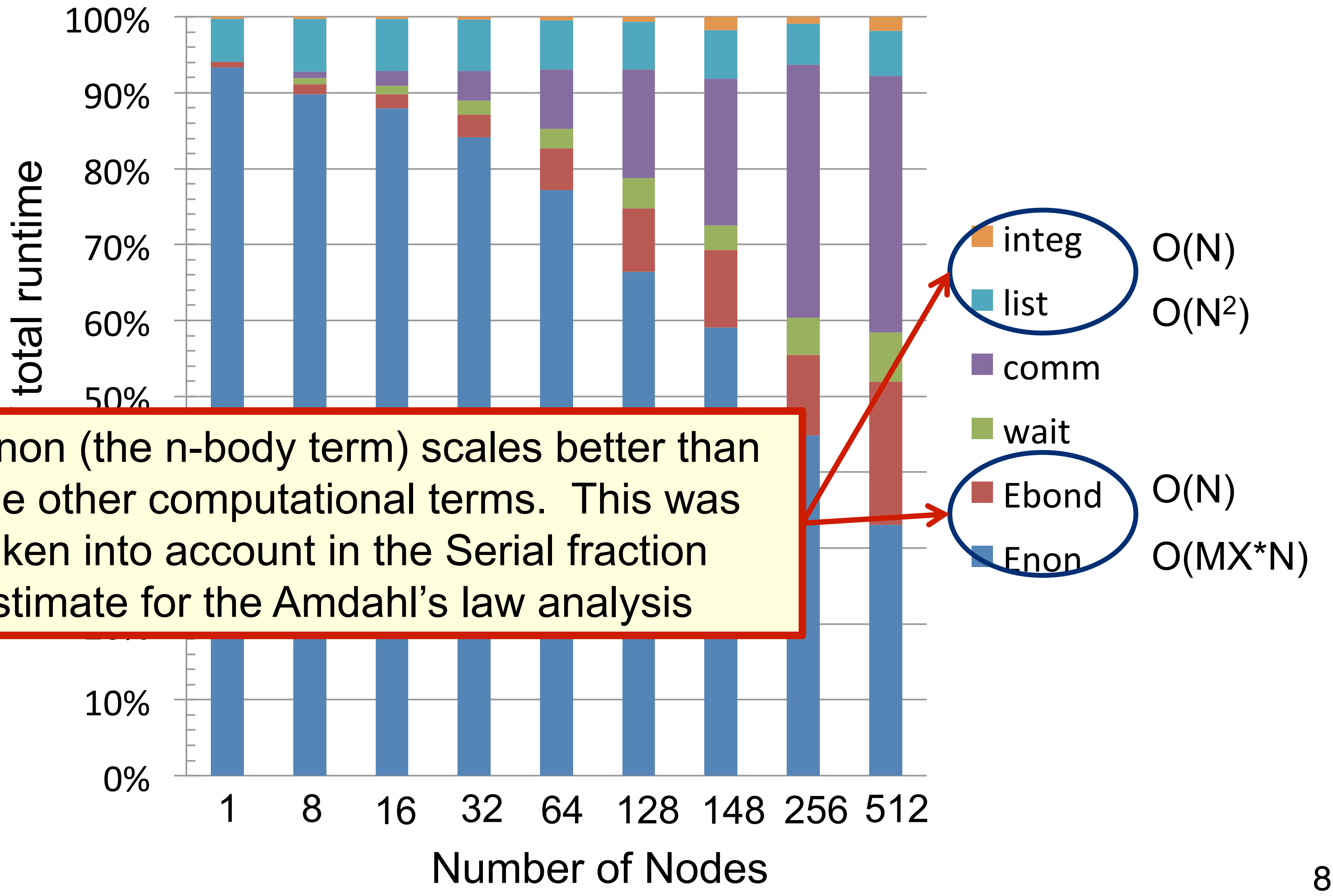
- Percent of runtime for the different phases of the computation



CHARMM running on a distributed memory, MPP supercomputer using a message passing library (NX)

Charm Myoglobin Benchmark

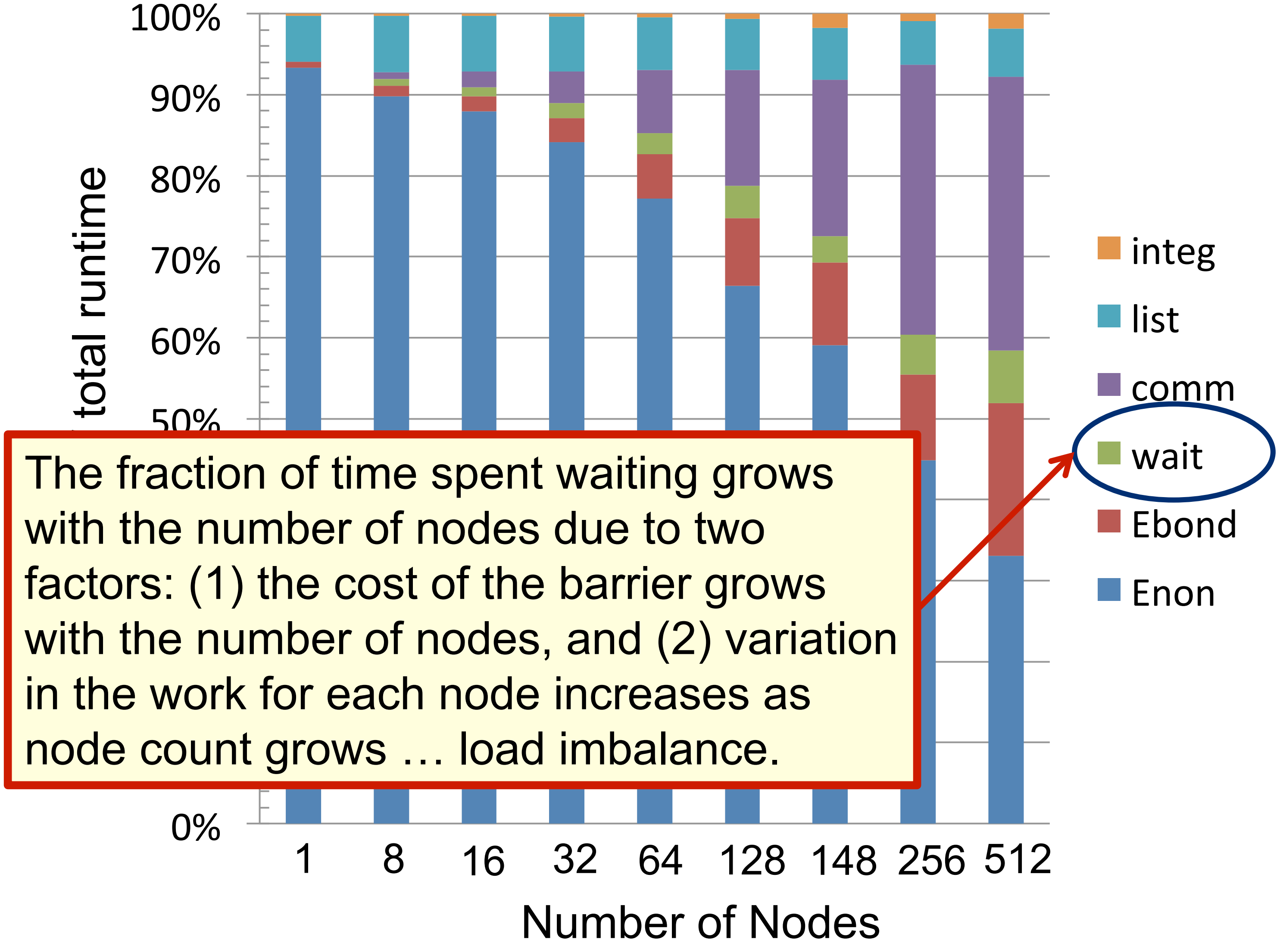
- Percent of runtime for the different phases of the computation



Enon (the n-body term) scales better than the other computational terms. This was taken into account in the Serial fraction estimate for the Amdahl's law analysis

Charm Myoglobin Benchmark

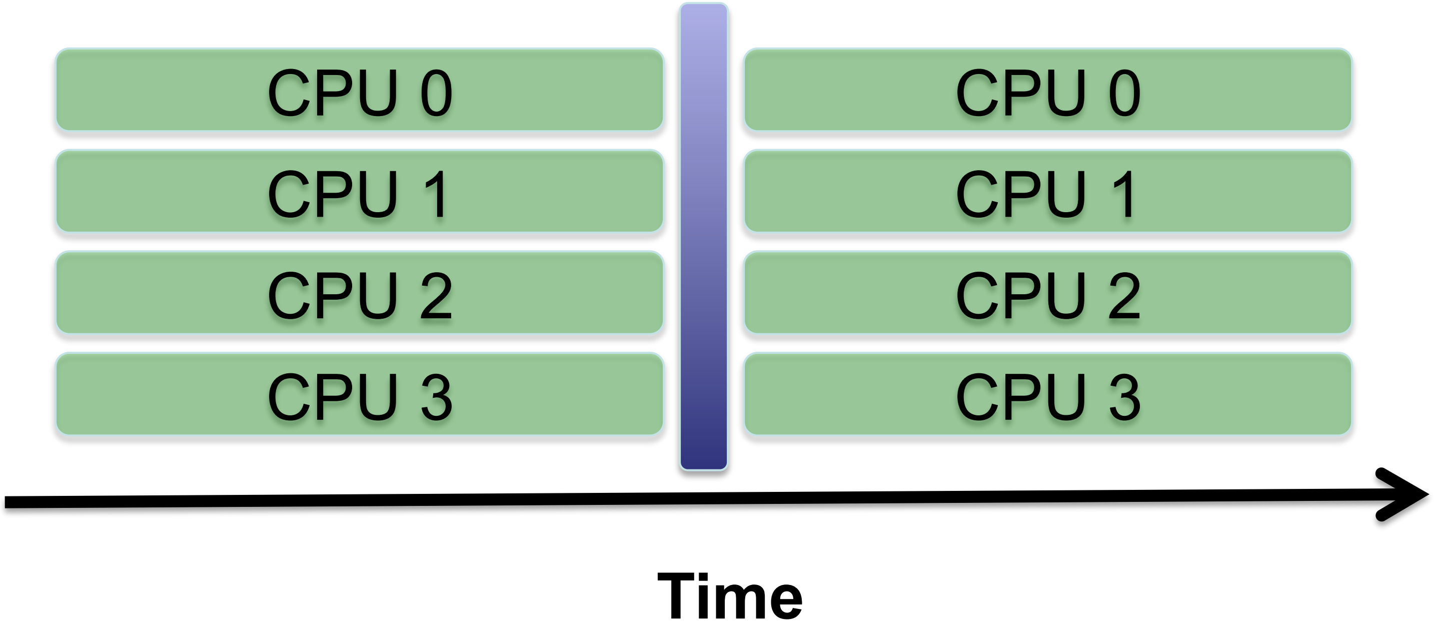
- Percent of runtime for the different phases of the computation



The fraction of time spent waiting grows with the number of nodes due to two factors: (1) the cost of the barrier grows with the number of nodes, and (2) variation in the work for each node increases as node count grows … load imbalance.

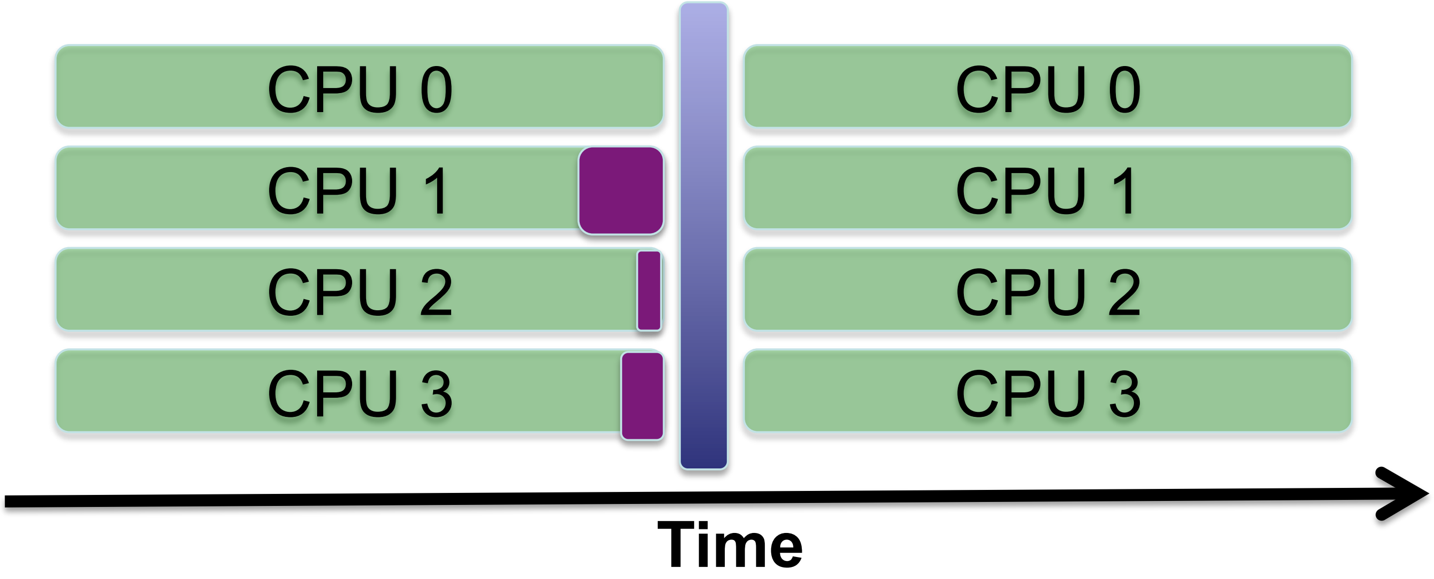
Synchronization overhead

- Processes finish their work and must assure that all processes are finished before the results are combined into the global force array.
  - This is parallel overhead since this doesn't occur in a serial program.
  - The synchronization construct itself takes time and in some cases (such as a barrier) the cost grows with the number of nodes.



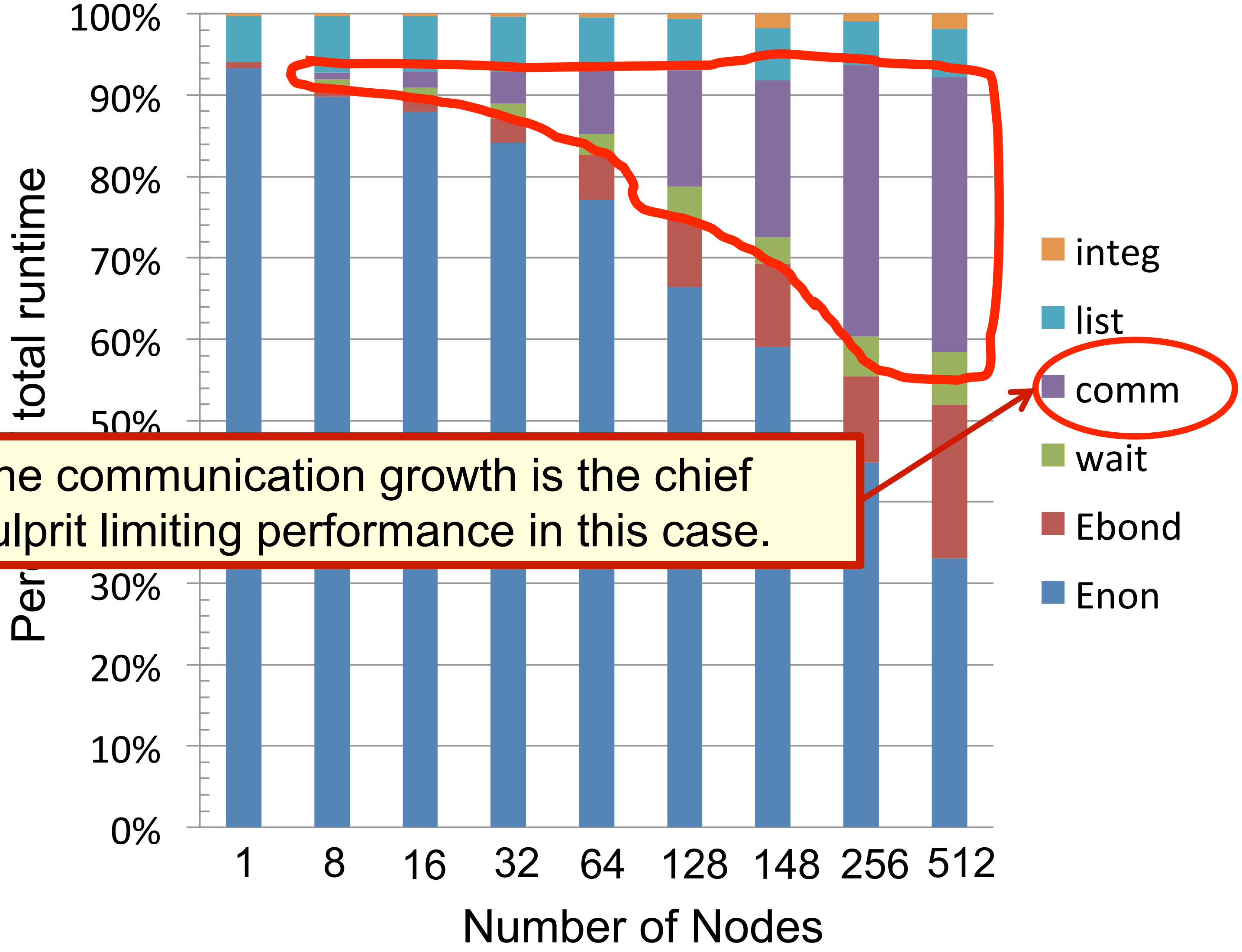
Load imbalance

- If some processes finish their share of the computation early, the time spent waiting for other processors is wasted.
  - This is an example of ***Load Imbalance***



Charm Myoglobin Benchmark

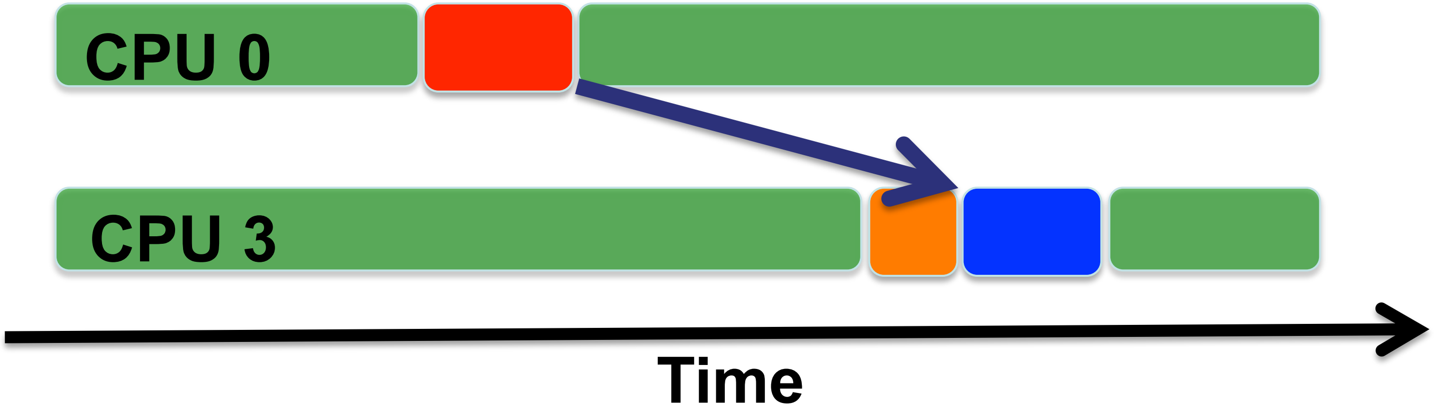
- Percent of runtime for the different phases of the computation



The communication growth is the chief culprit limiting performance in this case.

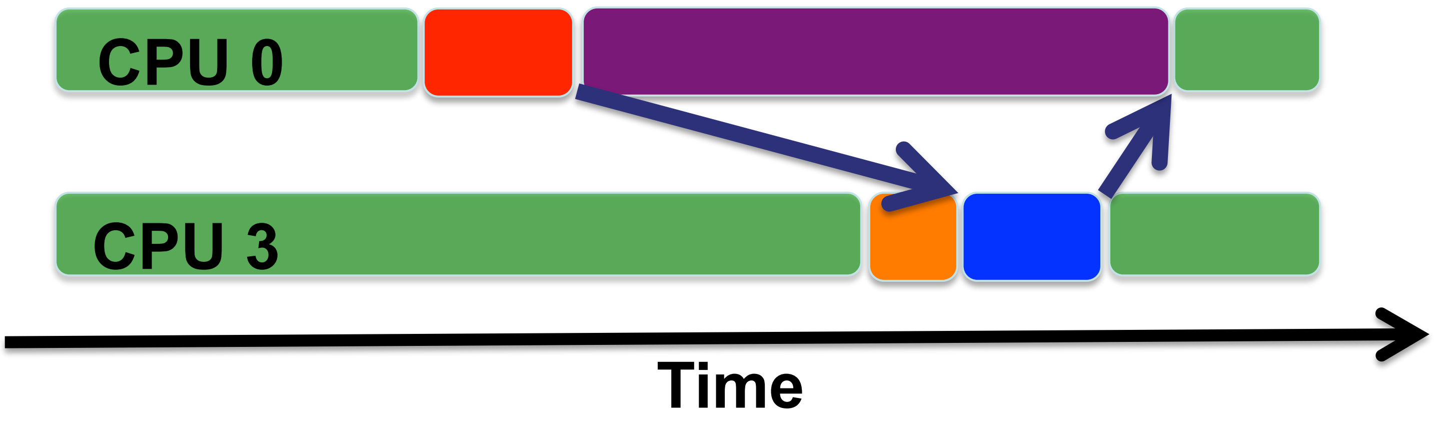
Communication

- On distributed-memory machines (e.g. a cluster), communication can only occur by sending discrete messages over a network
  - The sending processor **marshals the shared data** from the application's data structures into a message buffer
  - The receiving processor must **wait for the message** to arrive ...
  - ... and **un-pack the data** back into data structures



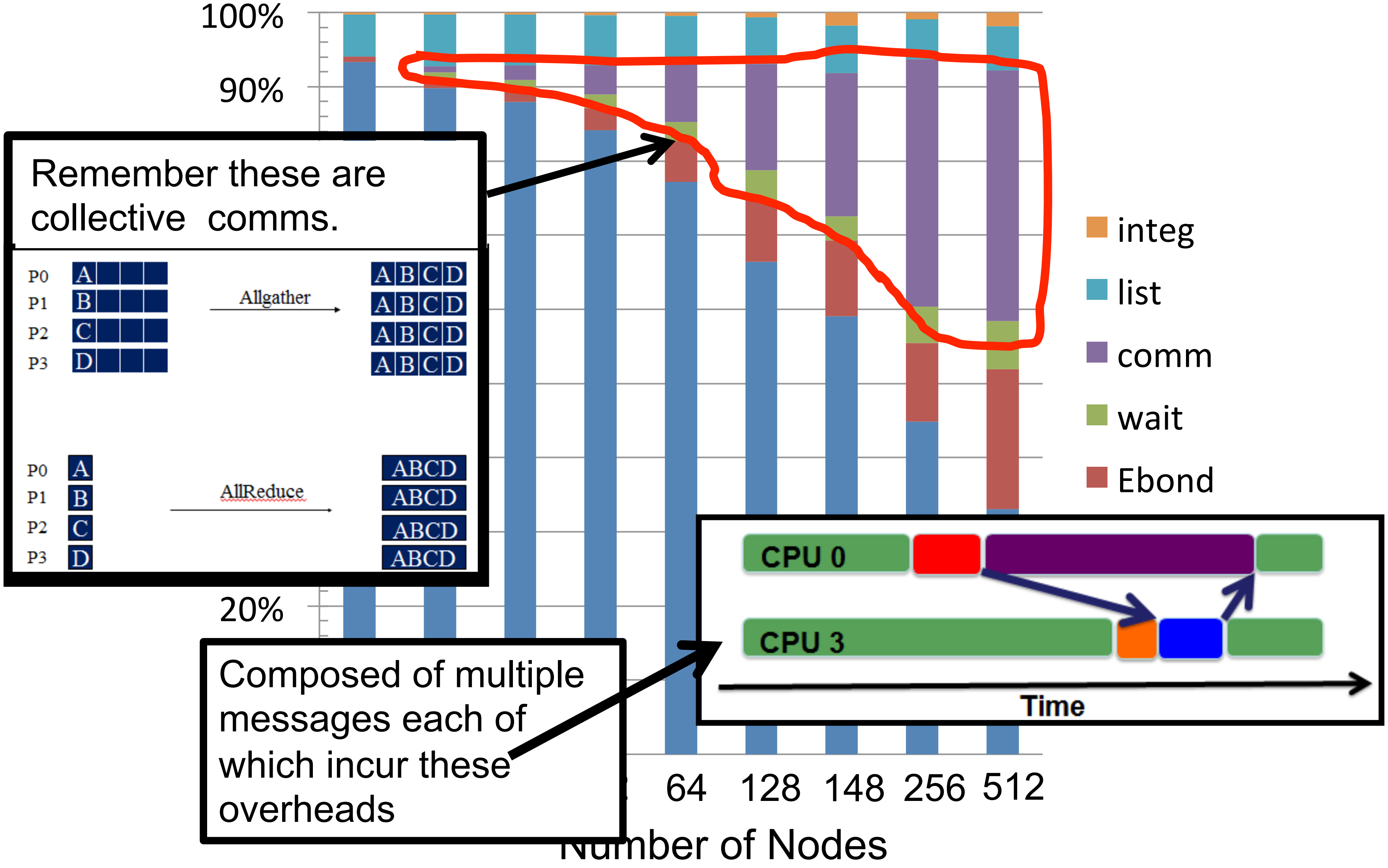
Communication

- On distributed-memory machines (e.g. a cluster), communication can only occur by sending discrete messages over a network
  - The sending processor **marshals the shared data** from the application's data structures into a message buffer
  - The receiving processor must **wait for the message** to arrive ...
  - ... and **un-pack the data** back into data structures
- If the communication protocol is ***synchronous***, then the sending processor must **wait for acknowledgement** that the message was received



Charm Myoglobin Benchmark

- Percent of runtime for the different phases of the computation

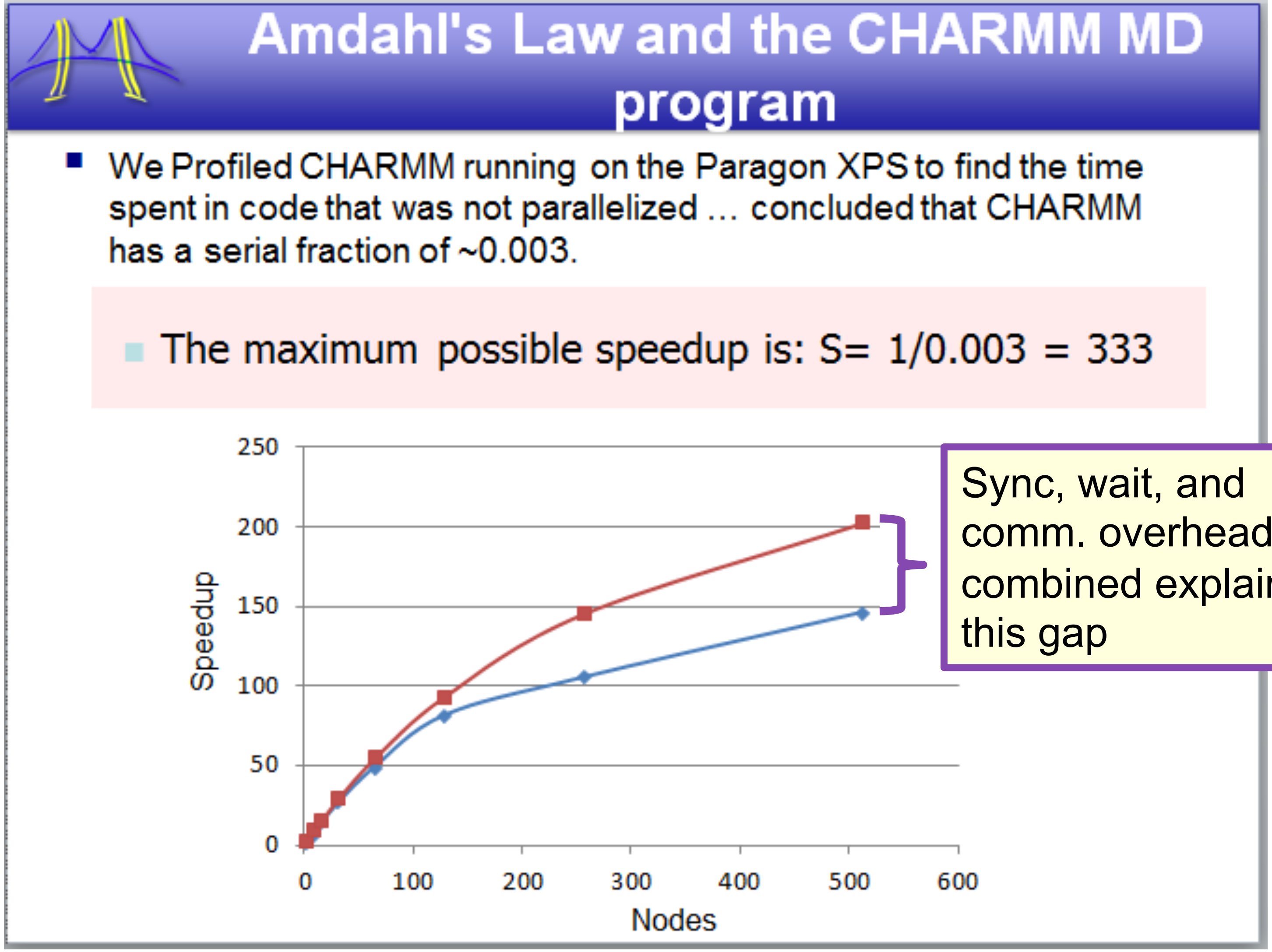


Remember these are collective comms.

Composed of multiple messages each of which incur these overheads

Limitations to scalability

- Remember the speedup plot we discussed from last time?



Amdahl's Law and the CHARMM MD program

- We Profiled CHARMM running on the Paragon XPS to find the time spent in code that was not parallelized … concluded that CHARMM has a serial fraction of ~0.003.

- The maximum possible speedup is: S= 1/0.003 = 333

Sync, wait, and comm. overheads combined explain this gap

Outline

- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
- ➡ An introduction to parallel hardware
- Software for parallel systems: key design patterns
- Closing Comments

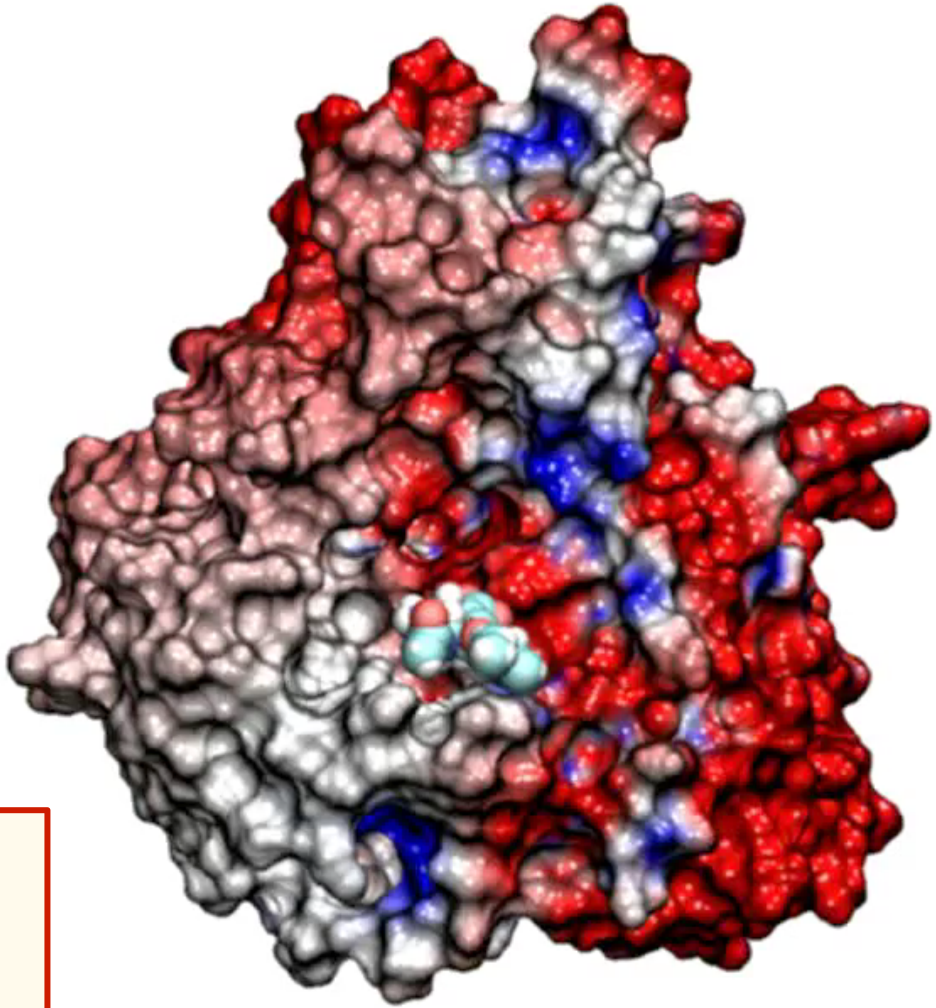
Outline

- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
- An introduction to parallel hardware
  - ➡ History of parallel hardware
  - The major building blocks of modern parallel systems
    - Multicore processors
    - The GPU
- Software for parallel systems: key design patterns
- Closing Comments

The Essence of supercomputing:
Amazing science

A study of the H1N1 virus and how mutations render anti-virus drugs ineffective.

The video shows the electrostatic surface potential around the drug binding site of the H1N1 neuraminidase enzyme… with unbinding and rebinding of Tamiflu into the active site on the protein.



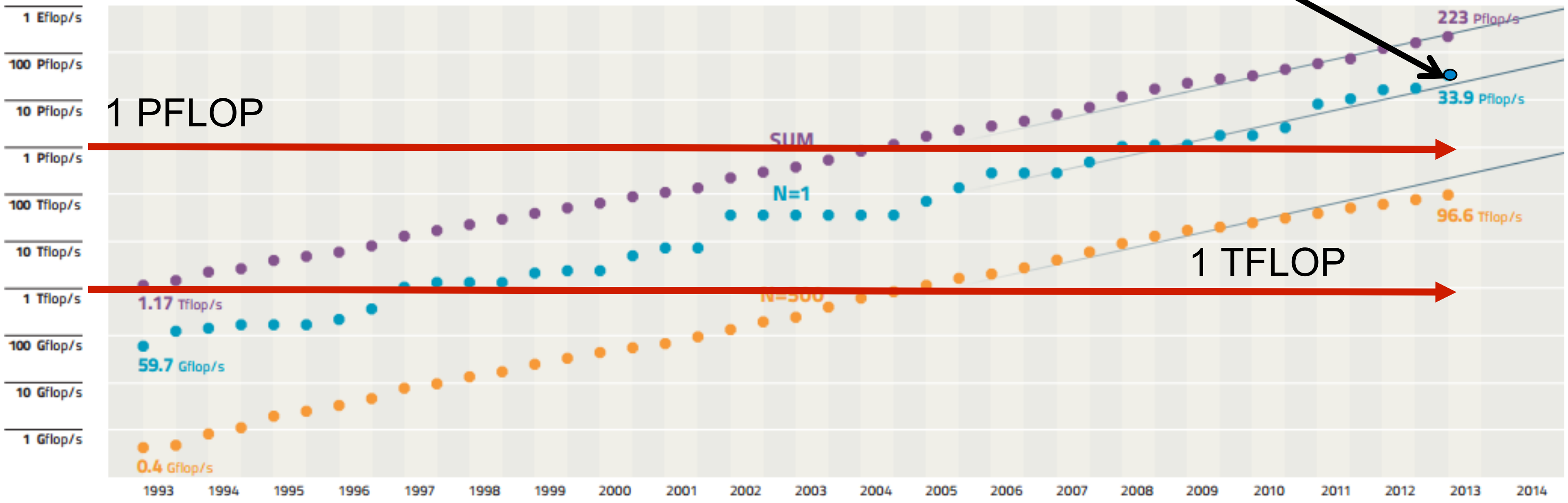
Scientific supercomputing is addictive. Once you wrap your brain around these sorts of problems, there is no going back.

Source: http://www.ks.uiuc.edu/Research/influenza, Klaus Schulten's biophysics group at UIUC using their NAMD program running on clusters (Ranger at TACC) and Nvidia GPUs

Tracking Supercomputers: Top500

- Top500: a list of the 500 fastest computers in the world (www.top500.org)
- Computers ranked by solution to the MPLinpack benchmark:
  - Solve Ax=b problem for any order of A
- List released twice per year: in June and November

Current number 1 (June 2013) R_{max}=33.9 PFLOPS
Tianhe-2, NUDT, Intel Ivy Bridge + Xeon Phi cluster
17.8 megawatts,   >3million cores

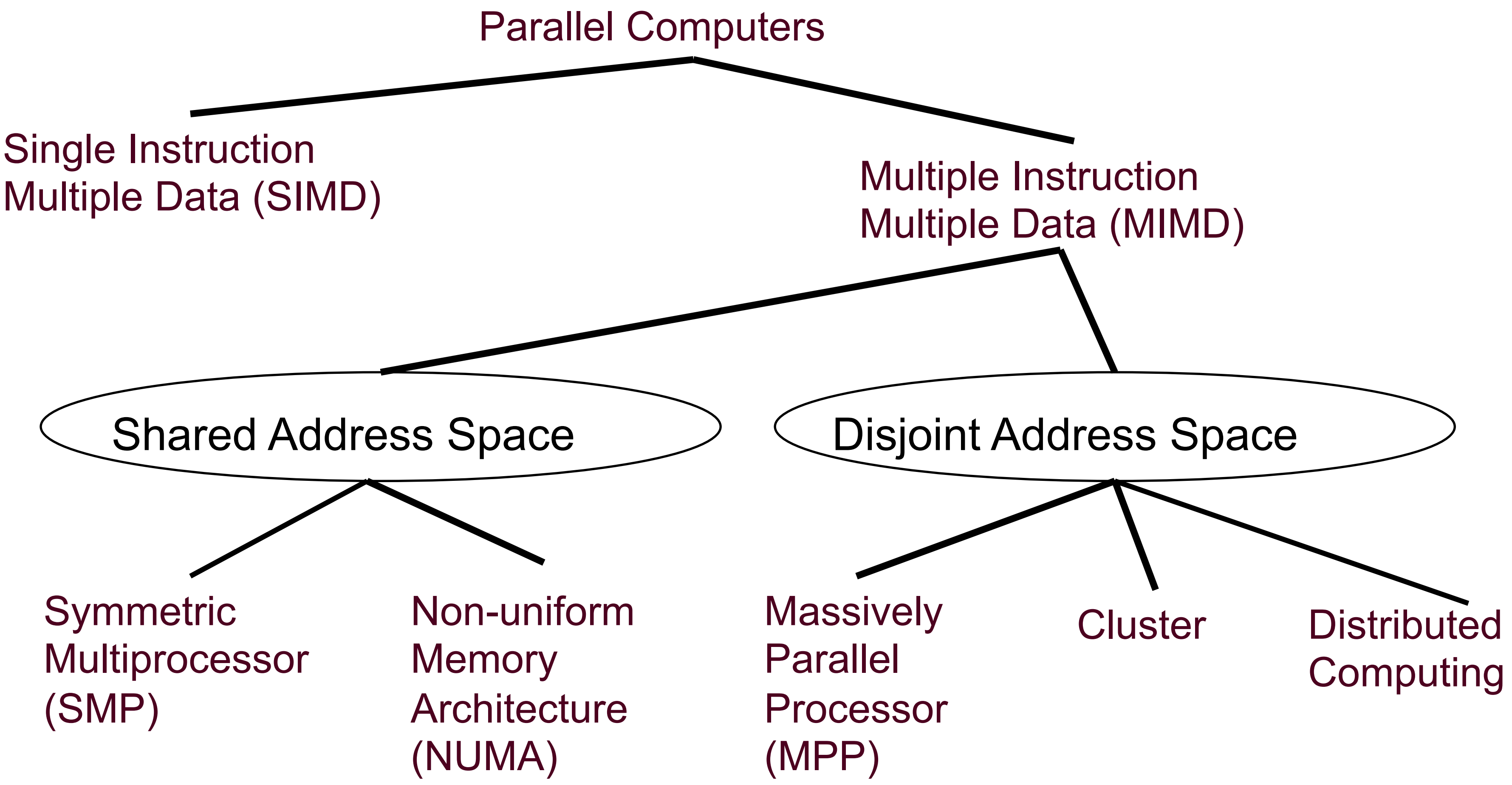


1 PFLOP

1 TFLOP

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

Hardware Architectures for High Performance Computing (HPC)



Parallel Computers

Single Instruction Multiple Data (SIMD)

Multiple Instruction Multiple Data (MIMD)

Shared Address Space

Disjoint Address Space

Symmetric Multiprocessor (SMP)

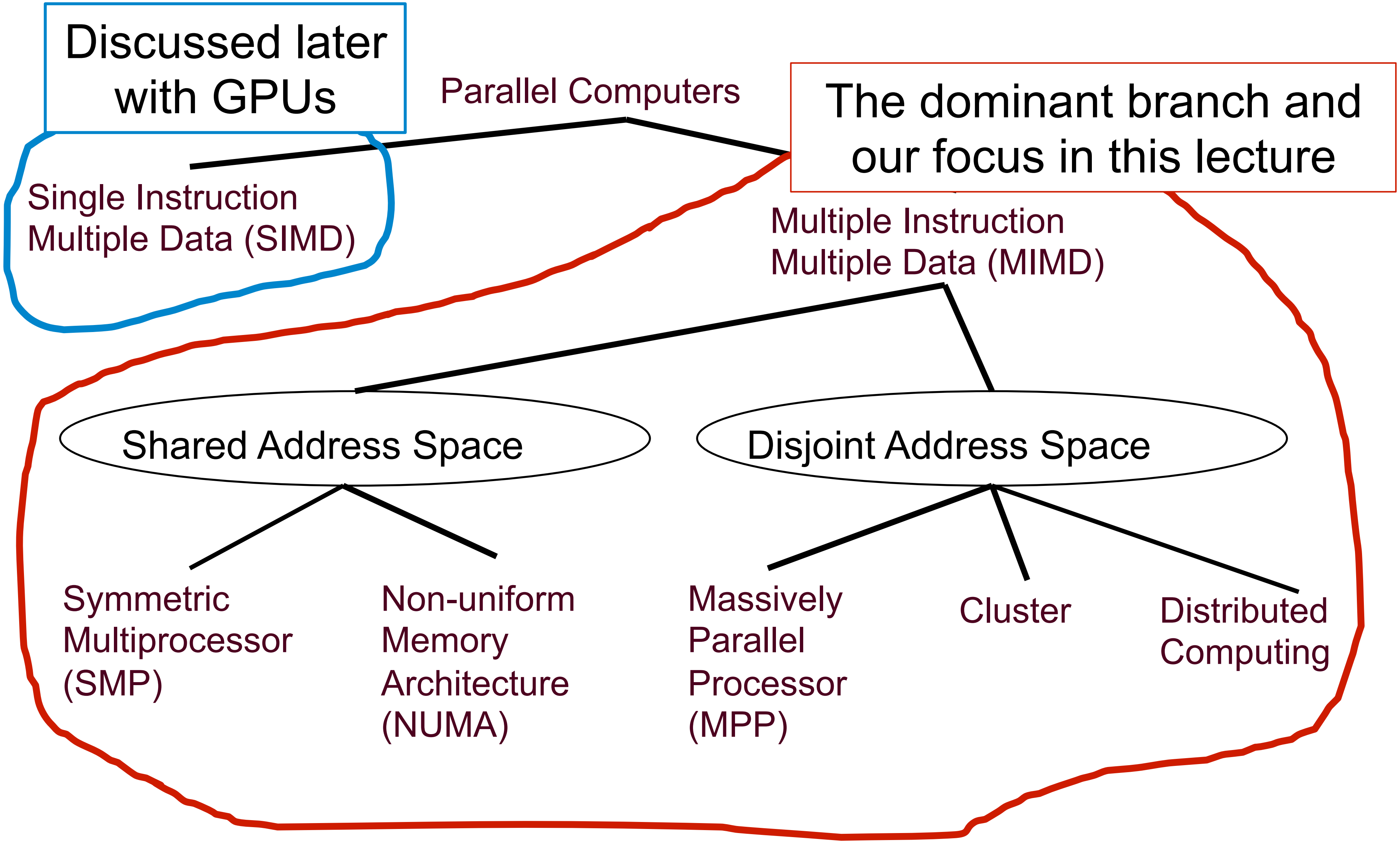
Non-uniform Memory Architecture (NUMA)

Massively Parallel Processor (MPP)

Cluster

Distributed Computing

Hardware Architectures for High Performance Computing (HPC)



Discussed later with GPUs

Parallel Computers

The dominant branch and our focus in this lecture

Single Instruction Multiple Data (SIMD)

Multiple Instruction Multiple Data (MIMD)

Shared Address Space

Disjoint Address Space

Symmetric Multiprocessor (SMP)

Non-uniform Memory Architecture (NUMA)

Massively Parallel Processor (MPP)

Cluster

Distributed Computing

The birth of Supercomputing

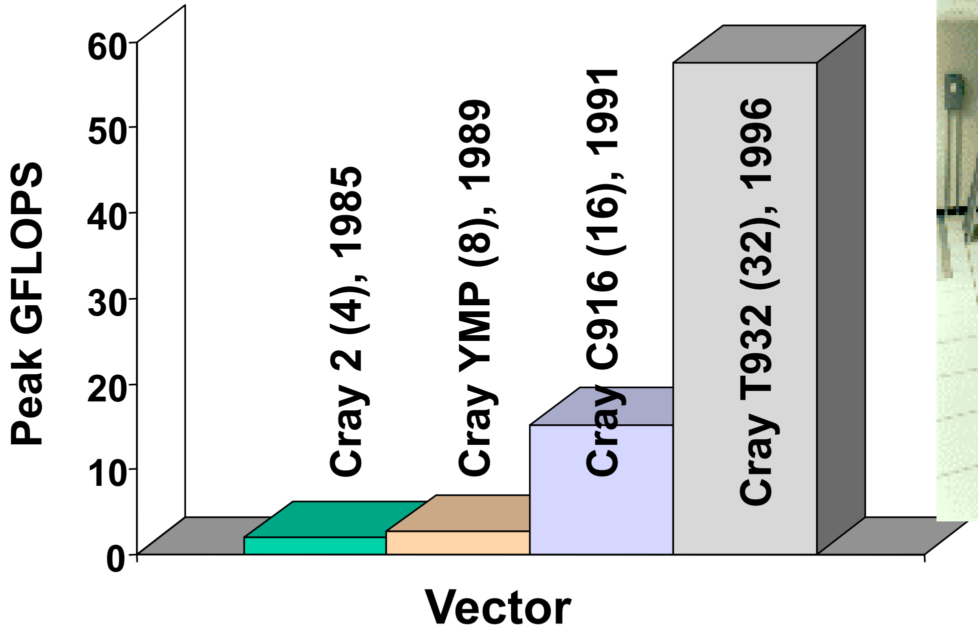
- On July 11, 1977, the CRAY-1A, serial number 3, was delivered to NCAR. The system cost was $8.86 million ($7.9 million plus $1 million for the disks).

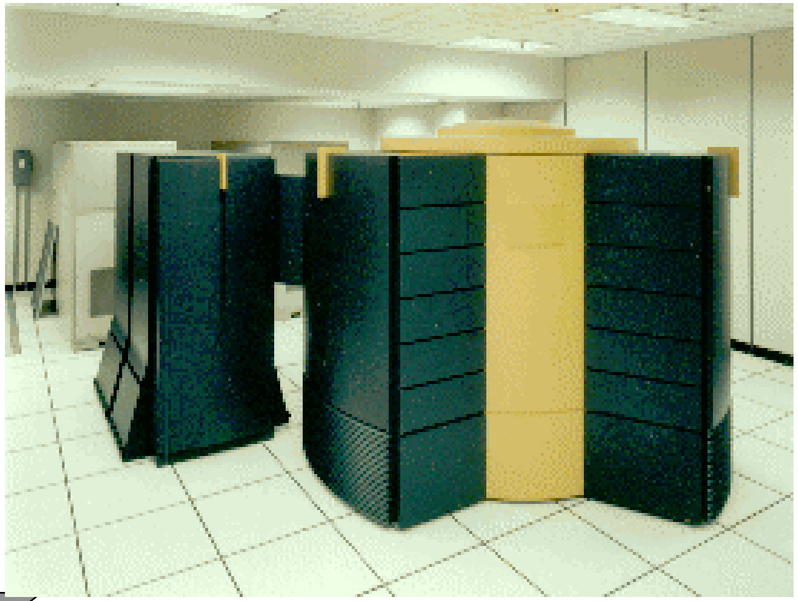
http://www.cisl.ucar.edu/computers/gallery/cray/cray1.jsp

- 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.

History of Supercomputing:

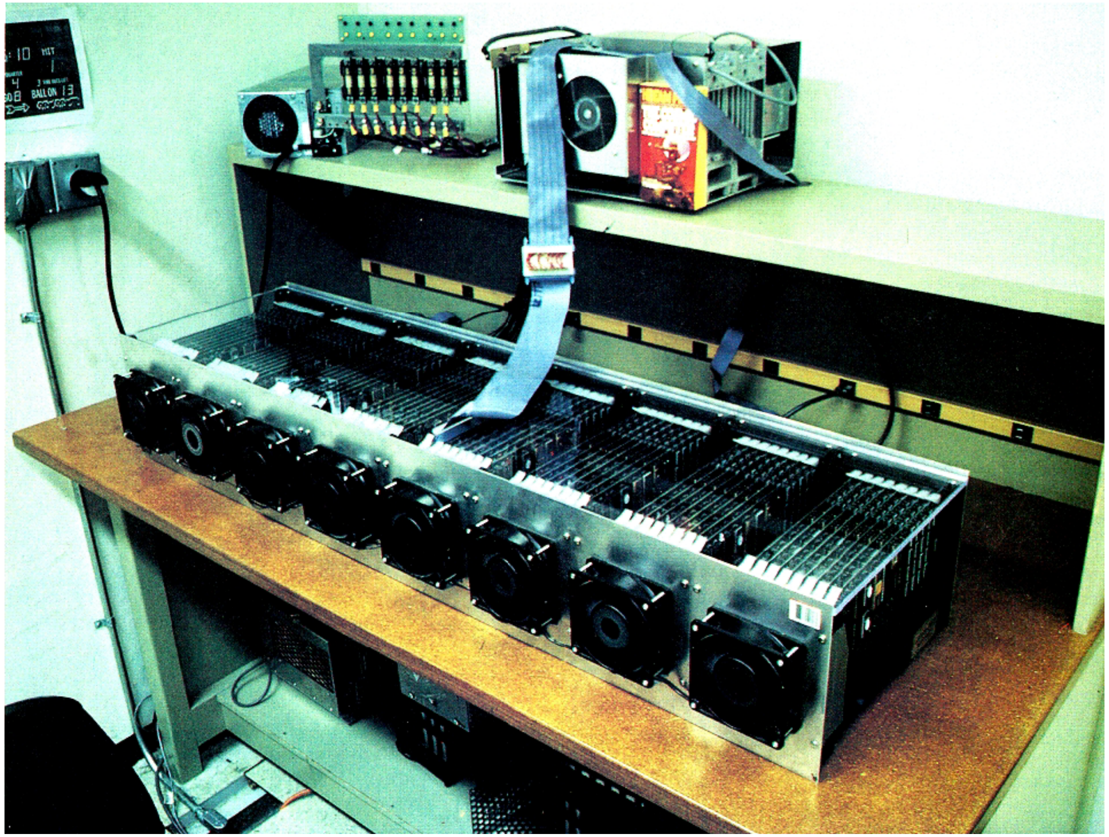
- Large mainframes that operated on vectors of data
- Custom built, highly specialized hardware and software
- Multiple processors in an shared memory configuration
- Required modest changes to software (vectorization)





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

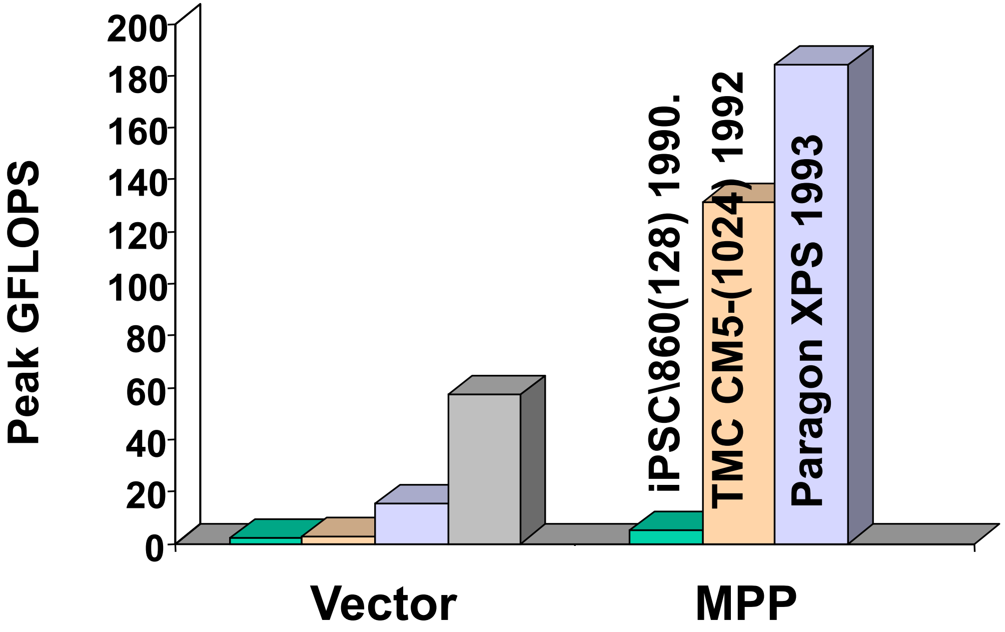
The cosmic cube, Charles Seitz
Communications of the ACM, Vol 28, number 1 January 1985, p. 22

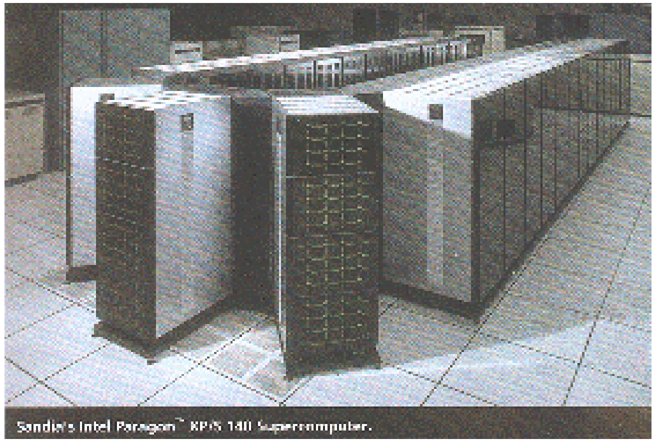
Launched the "attack of the killer micros"
Eugene Brooks, SC'90

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

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)

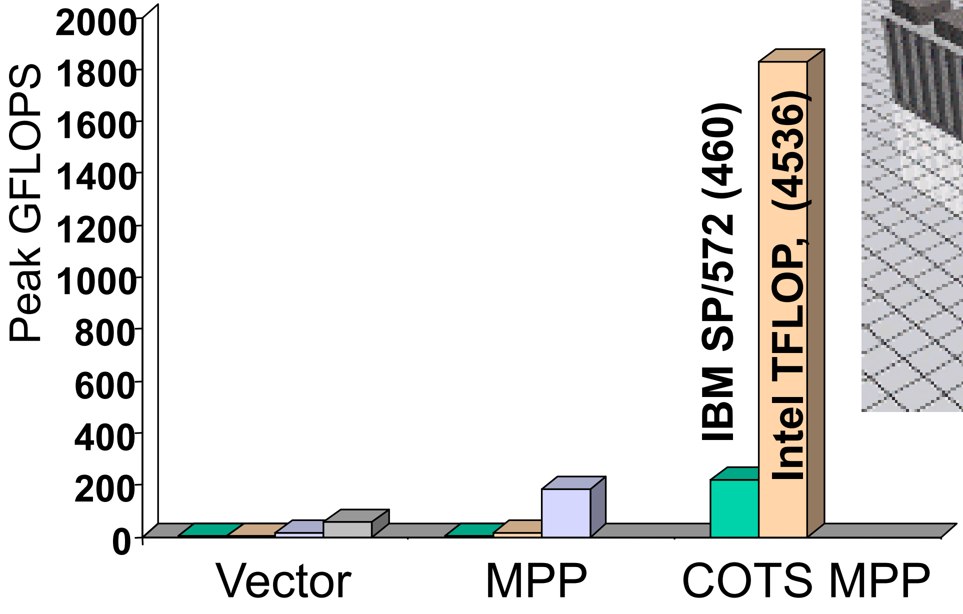


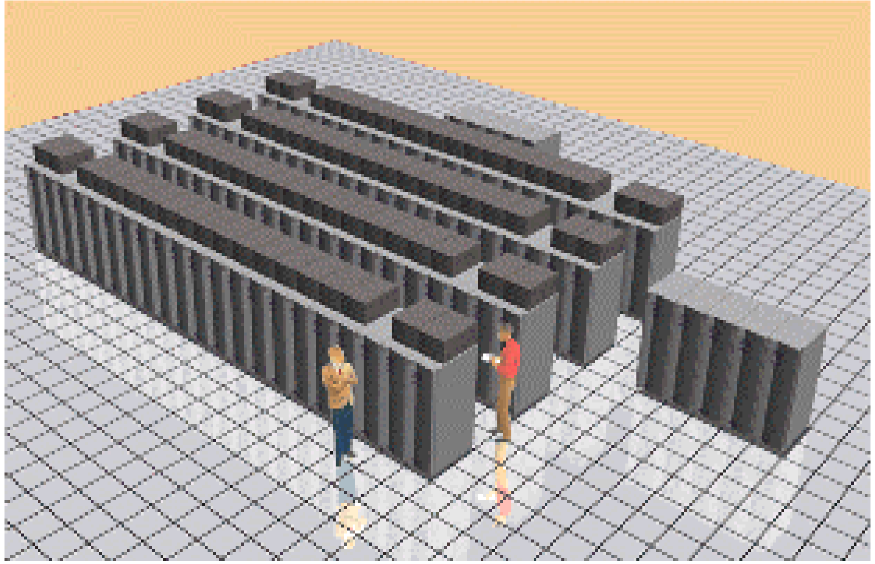


Paragon XPS-140 at Sandia National labs in Albuquerque NM

The cost advantage of mass market COTS

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



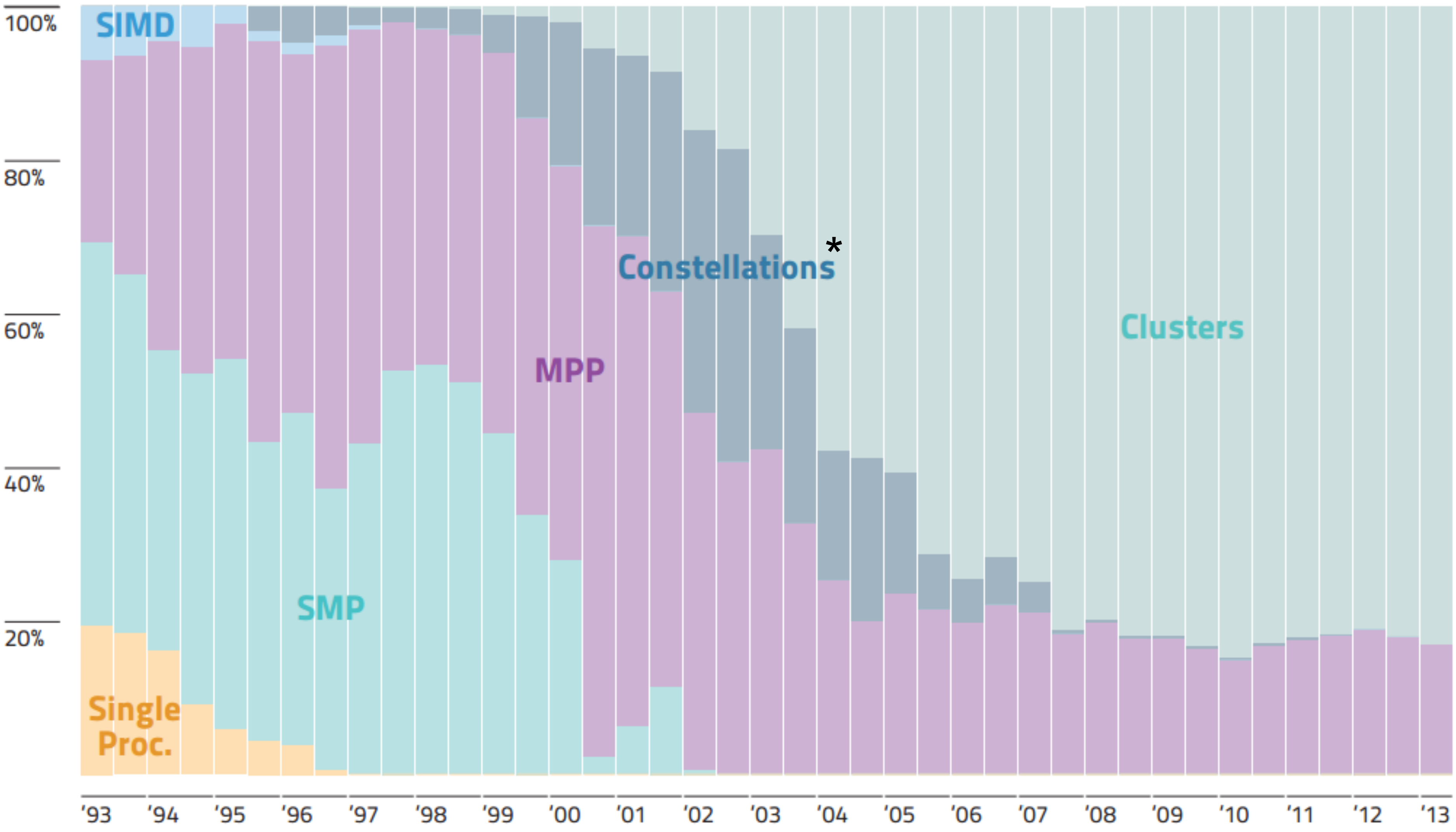


ASCI Red TFLOP Supercomputer

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 (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

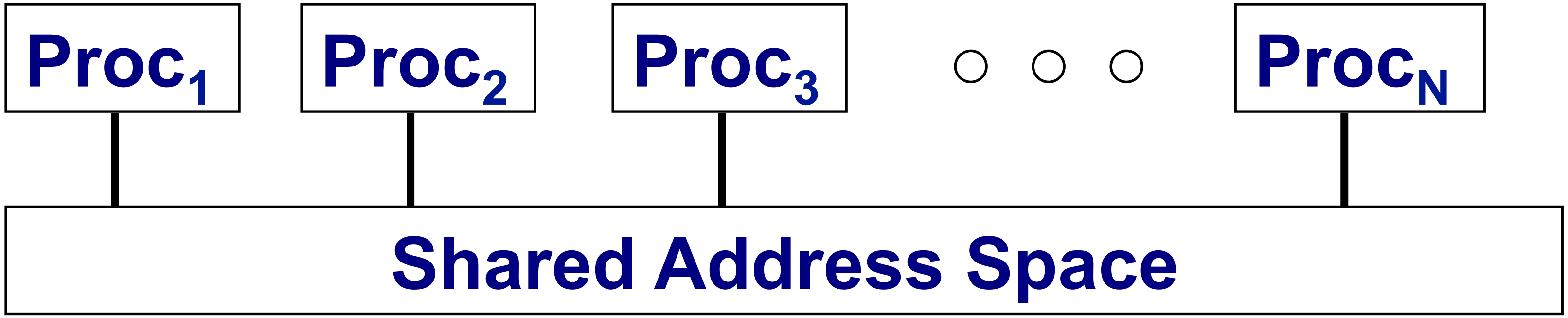
*Constellation: A cluster for which the number of processors on a node is greater than the number of nodes in the cluster. I've never seen anyone use this term outside of the top500 list.

Outline

- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
- An introduction to parallel hardware
  - History of parallel hardware
  - The major building blocks of modern parallel systems
    - ➡ Multicore processors
    - The GPU
- Software for parallel systems: key design patterns

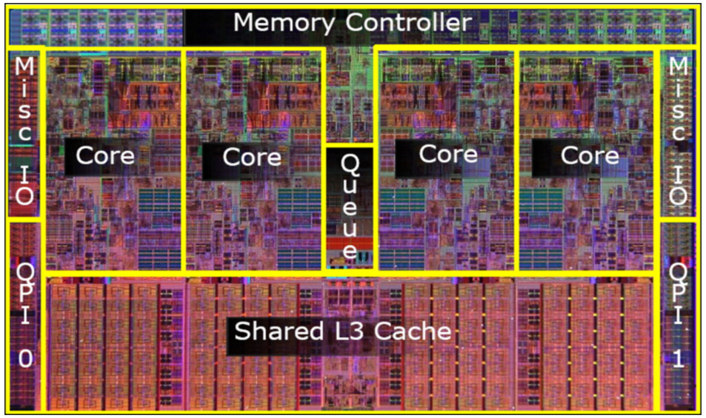
How do we connect cores together?

- A symmetric multiprocessor (SMP) consists of a collection of processors that share a single address space:
  - Multiple processing elements.
  - A shared address space with "equal-time" access for each processor.
  - The OS treats every processor the same



How realistic is this model?

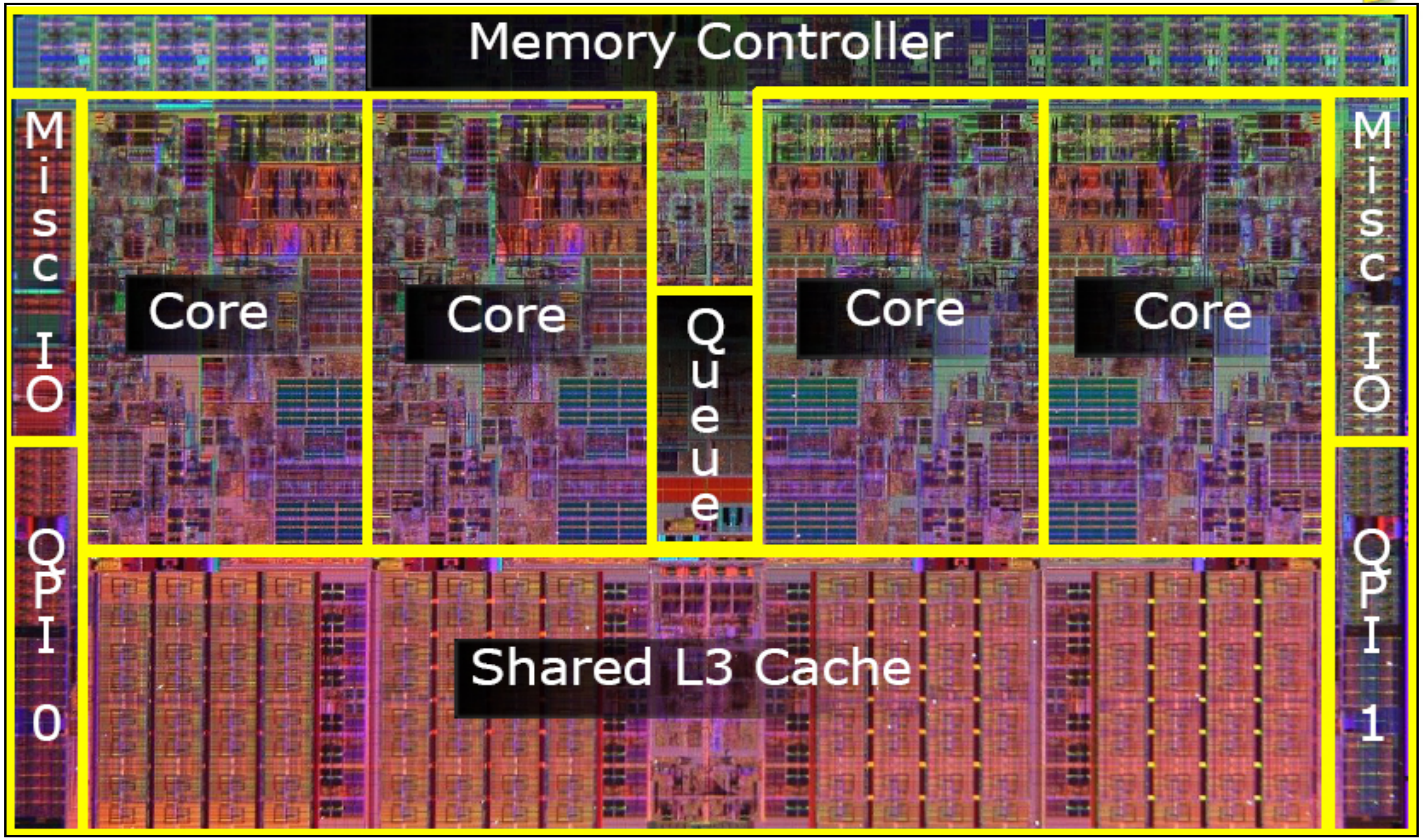
- Some of the old supercomputer mainframes followed this model,



A CPU with lots of cache …

- But as soon as we added caches to CPUs, the SMP model fell apart.
  - Caches … all memory is equal, but some memory is more equal than others.

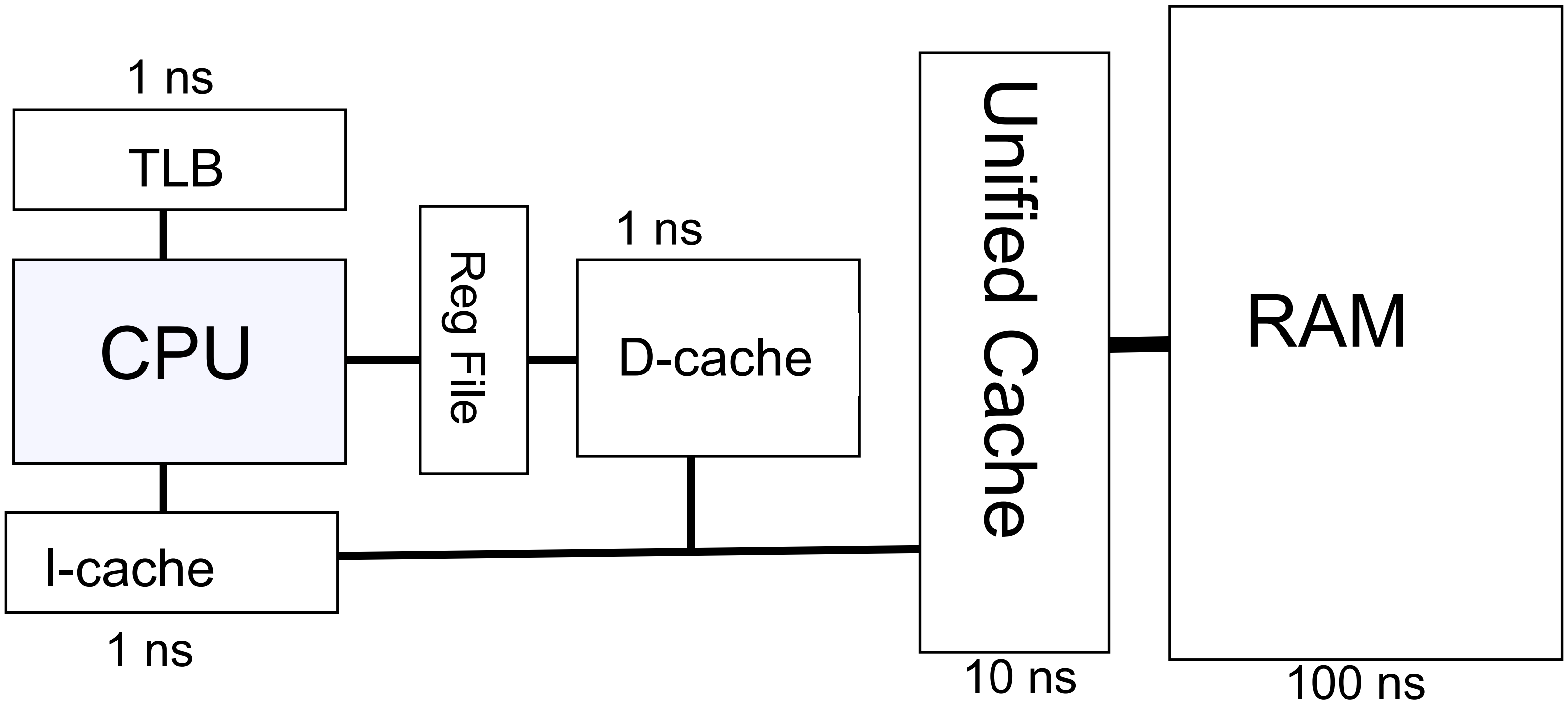
Example of modern core: Nahalem



- ON-chip cache resources:
  - For each core: L1: 32K instruction and 32K data cache, L2: 1MB
  - L3: 8MB shared among all 4 cores
- Integrated, on-chip memory controller (DDR3)

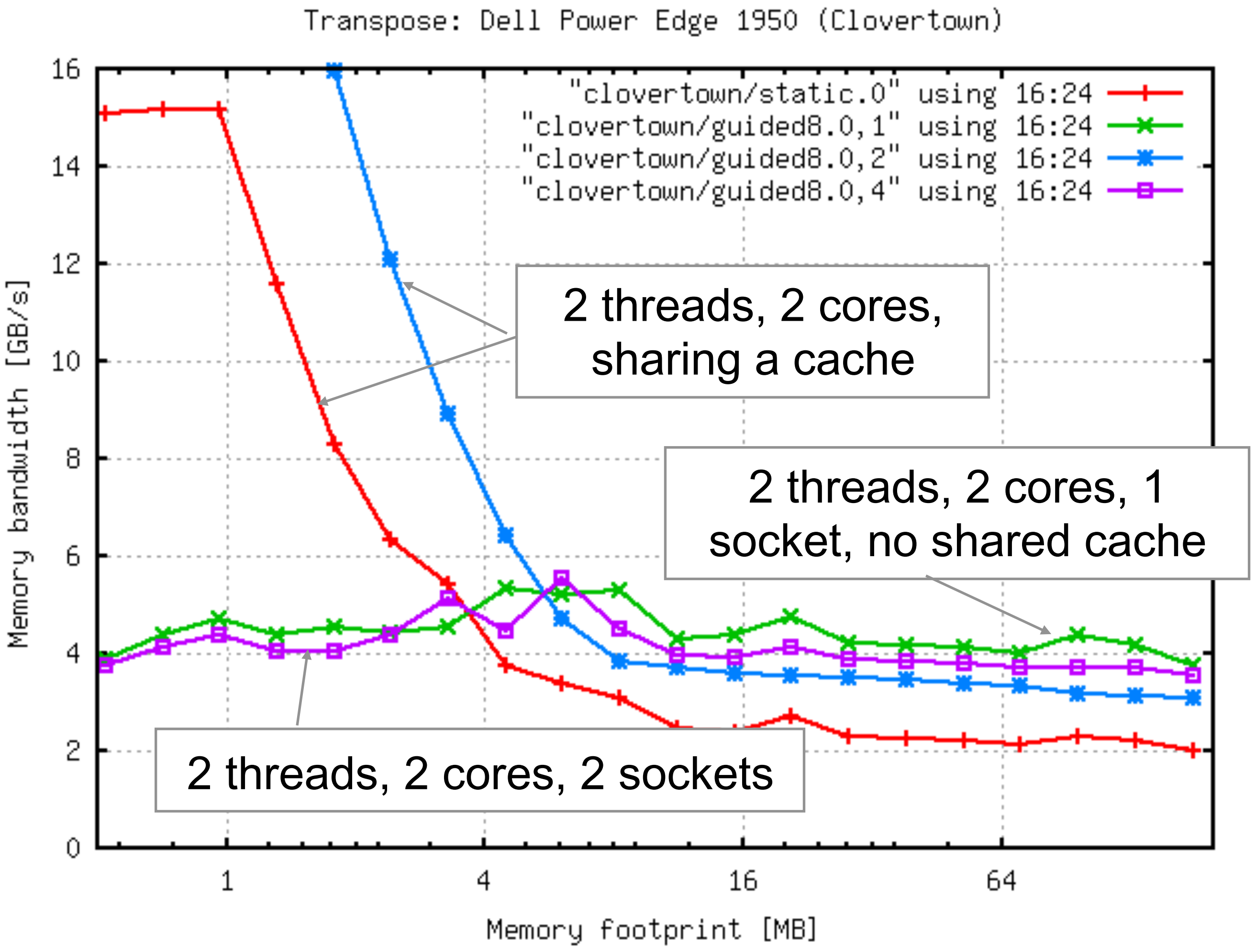
Memory Hierarchies

- A typical microprocessor memory hierarchy



- Instruction cache and data cache pull data from a unified cache that maps onto RAM.
- TLB implements virtual memory and brings in pages to support large memory foot prints.

NUMA* issues on a Multicore Machine
2-socket Clovertown Dell PE1950

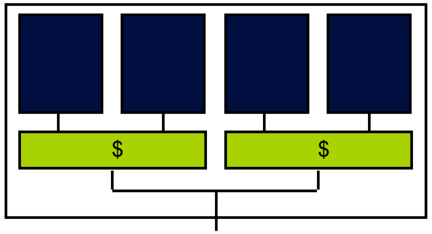


2 threads, 2 cores, sharing a cache

2 threads, 2 cores, 1 socket, no shared cache

2 threads, 2 cores, 2 sockets

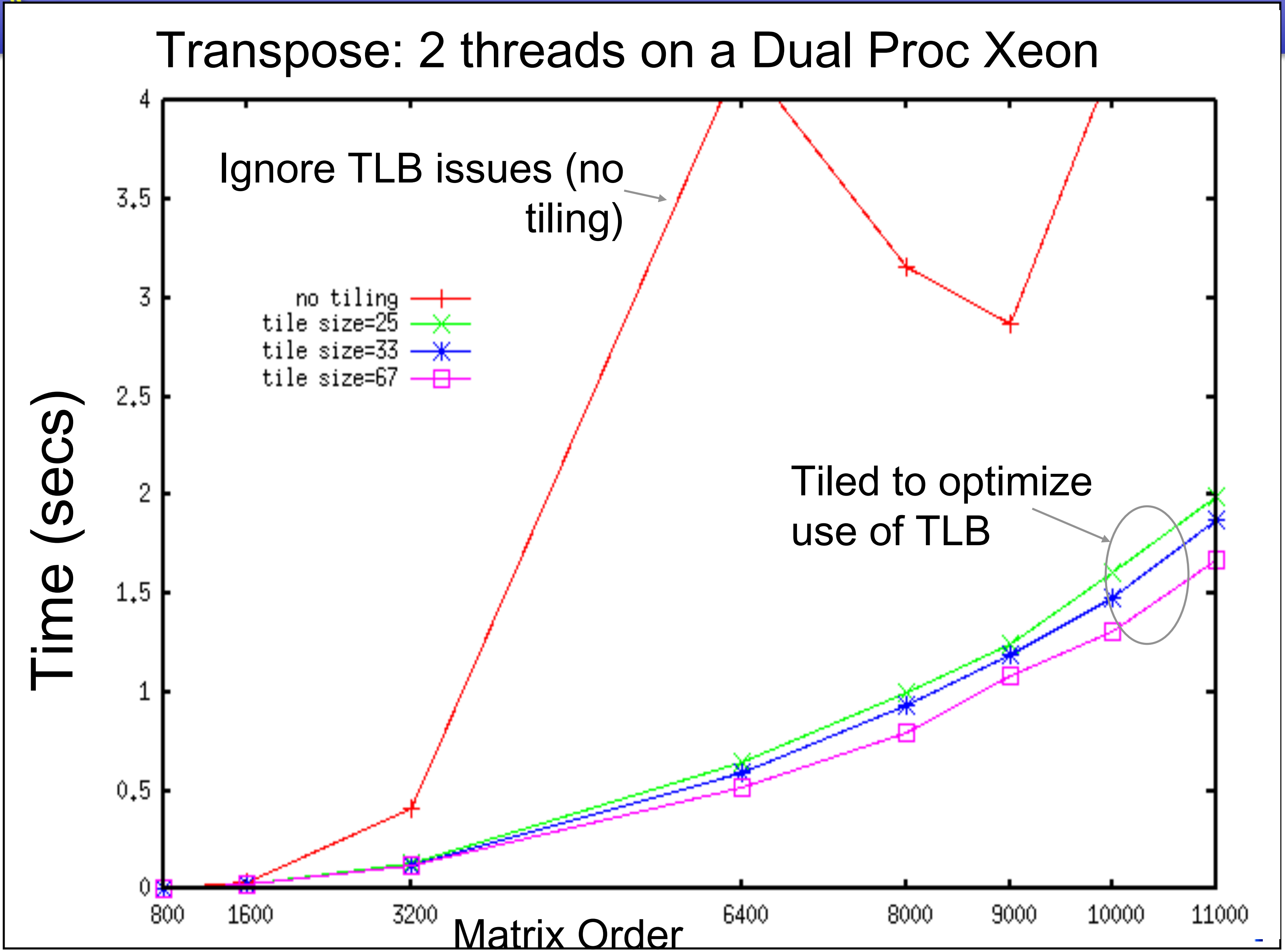
A single quad-core chip is a NUMA machine!



Xeon® 5300 Processor block diagram

*NUMA == Non Uniform Memory architecture … memory is shared but access times vary.

Do you need to worry about the TLB?

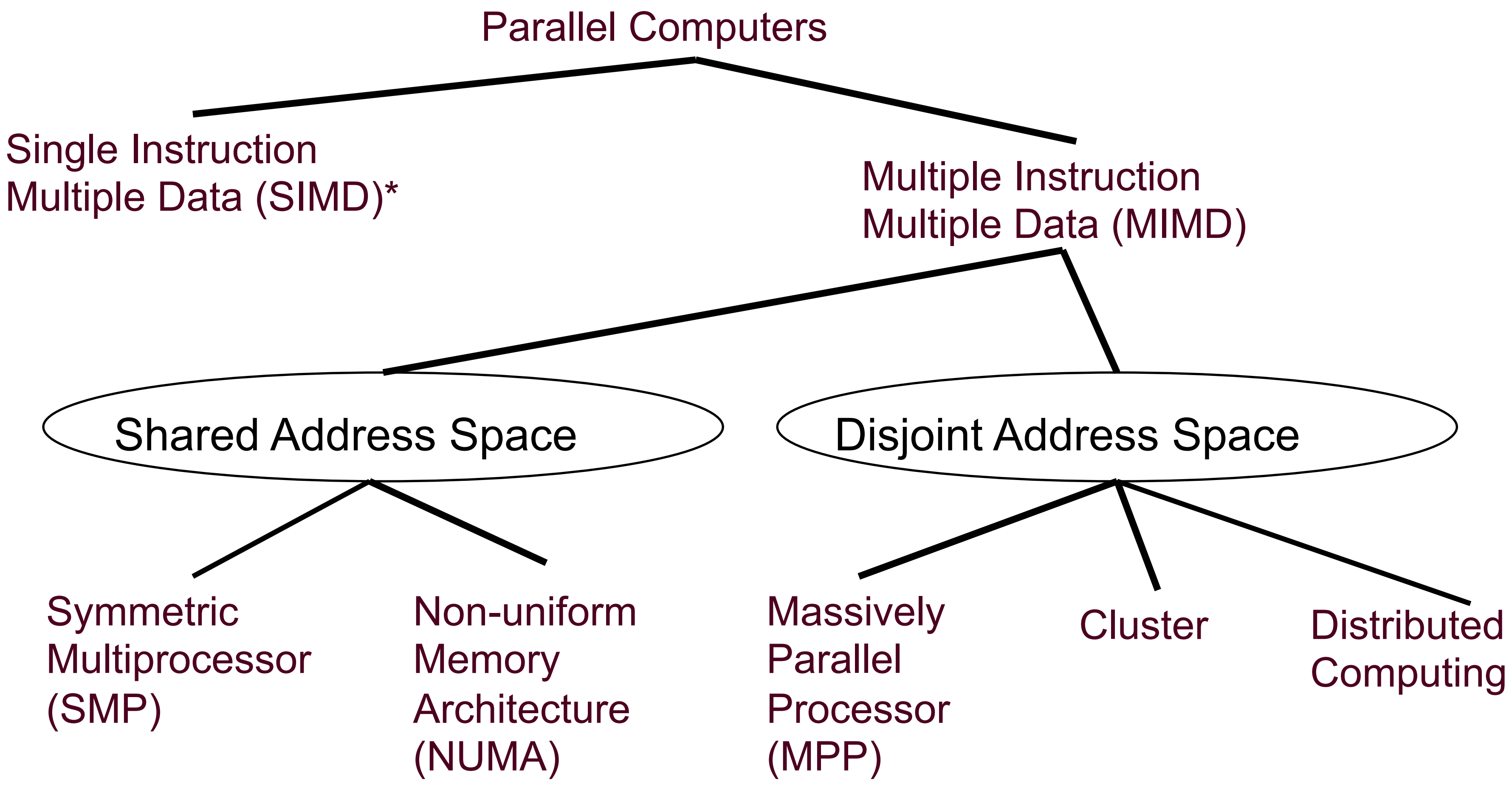


Source: M Frumkin, R. van de Wijngaart, T. G. Mattson, Intel

Outline

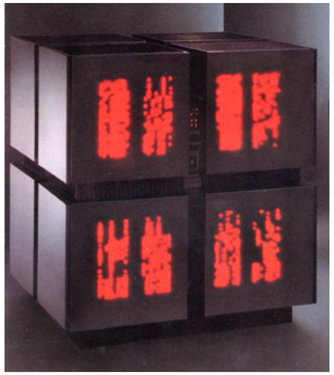
- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
- An introduction to parallel hardware
  - History of parallel hardware
  - The major building blocks of modern parallel systems
    - Multicore processors
    - ➡ The GPU
- Software for parallel systems: key design patterns
- Closing comments

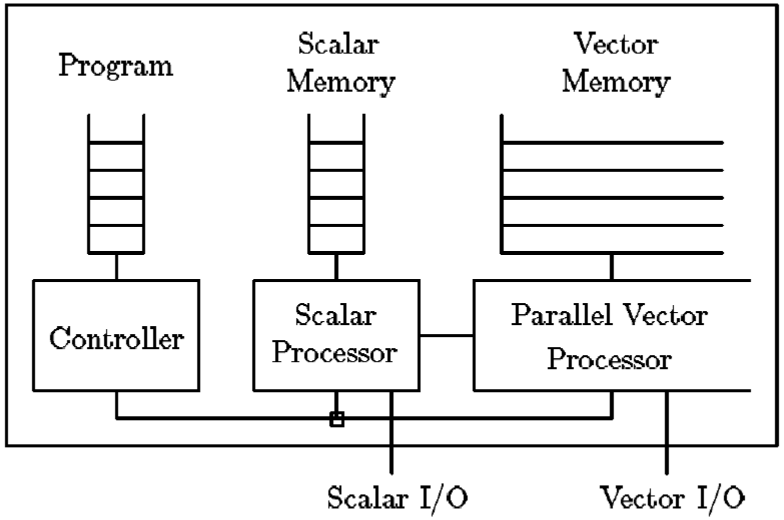
What happened to SIMD?


Parallel Computers
Single Instruction Multiple Data (SIMD)*
Multiple Instruction Multiple Data (MIMD)
Shared Address Space
Disjoint Address Space
Symmetric Multiprocessor (SMP)
Non-uniform Memory Architecture (NUMA)
Massively Parallel Processor (MPP)
Cluster
Distributed Computing

Data-Parallelism in HW Architecture

- Notions of "Data-Parallelism" in HW architecture were originally developed in the context of the strict-SIMD machines of the 1980's
  - Some of the first massively parallel systems: e.g. the Connection Machine with up to 64K processors
  - Have recently become relevant again (after a decade of dormancy) due to the wide availability of wide SIMD
- Called "Data-Parallel" because the source of parallelism is simultaneous operations across large sets of data, rather than from multiple threads of control
- The semantics of "pure" Data-Parallel languages are sequential, and parallelization is *implicit*
  - The program produces *"equivalent"* results if executed serially
  - Much easier to reason about correctness!

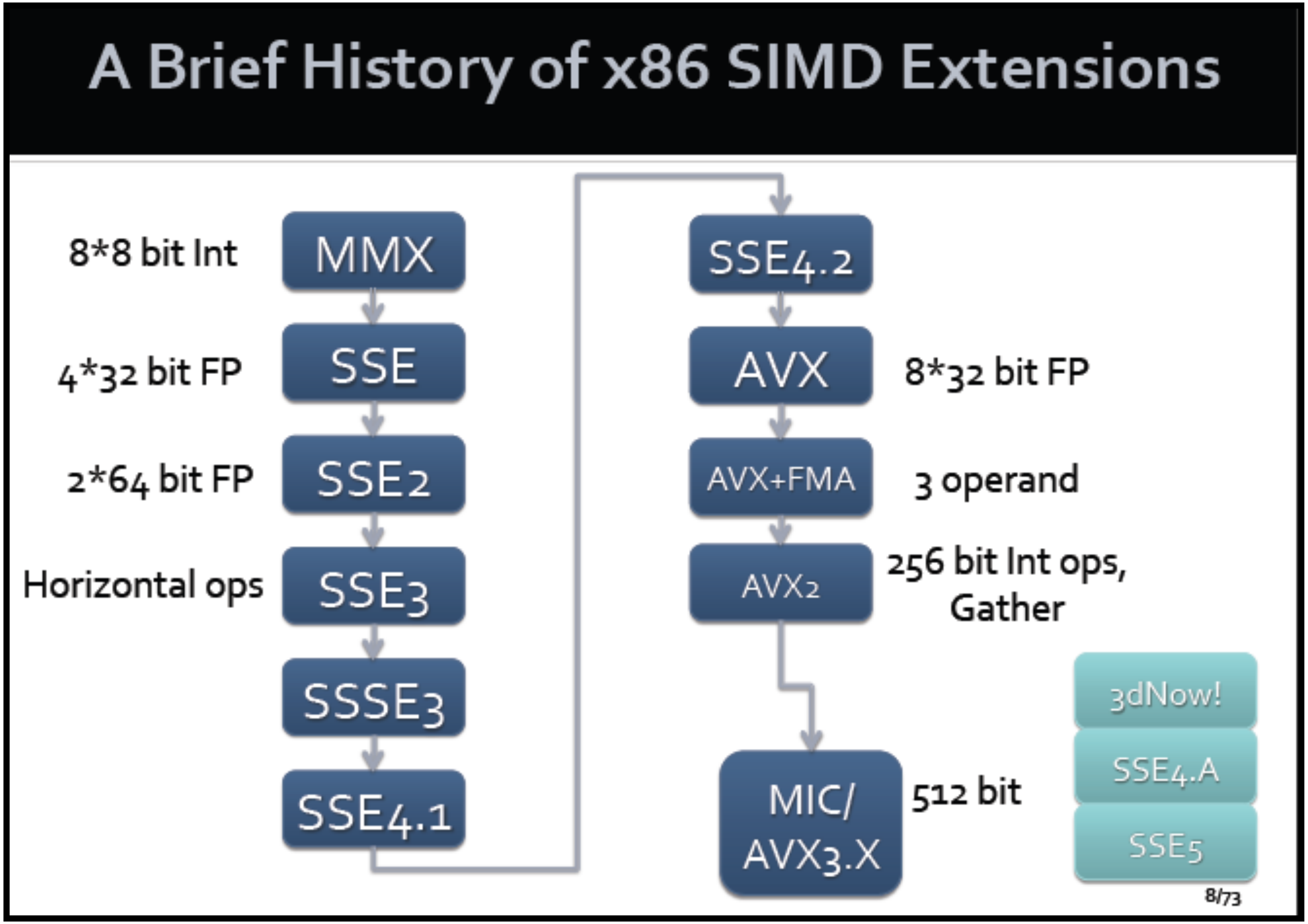




“Vector Models for Data-Parallel Computing”, Guy E. Blelloch

"Data Parallel Algorithms", Hillis and Steele, CACM 1986. Vol. 29, no. 12

SIMD and sx86 multimedia extensions.



A Brief History of x86 SIMD Extensions

8*8 bit Int — MMX

4*32 bit FP — SSE

2*64 bit FP — SSE2

Horizontal ops — SSE3

SSSE3

SSE4.1

SSE4.2

AVX — 8*32 bit FP

AVX+FMA — 3 operand

AVX2 — 256 bit Int ops, Gather

MIC/ AVX3.X — 512 bit

3dNow!

SSE4.A

SSE5

8/73

Source: Bryan Catanzaro, NVIDIA, UCB Parlab Bootcamp, 2013

A brief history of the GPU:

Coprocessors to support Graphics (and more)

1st generation:
Voodoo 3dfx (1996)

2nd Generation:
GeForce 256/Radeon 7500
(1998)

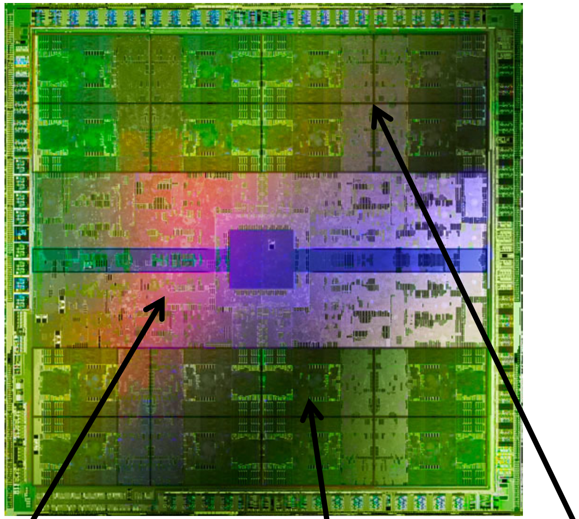
3rd Generation: GeForce3/Radeon 8500 (2001). The first GPU to allow a limited programmability in the vertex pipeline.

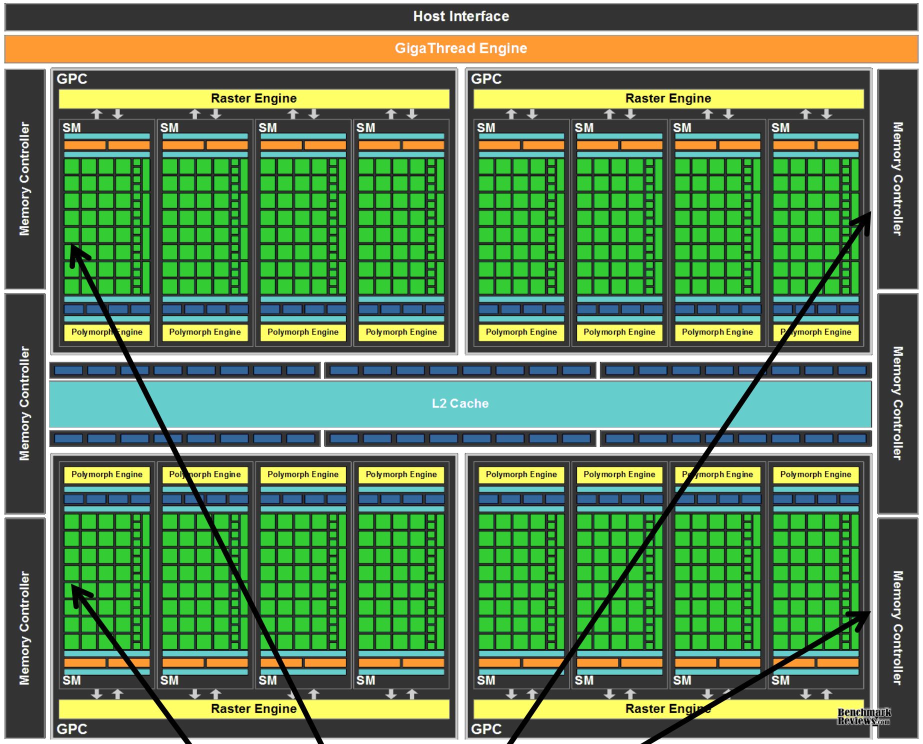
4th Generation: Radeon 9700/GeForce FX (2002): The first generation of “fully-programmable” graphics cards.

5th Generation: GeForce 8800/HD2900 (2006) and the birth of CUDA

Third party names are the property of their owners

NVIDIA GTX 480





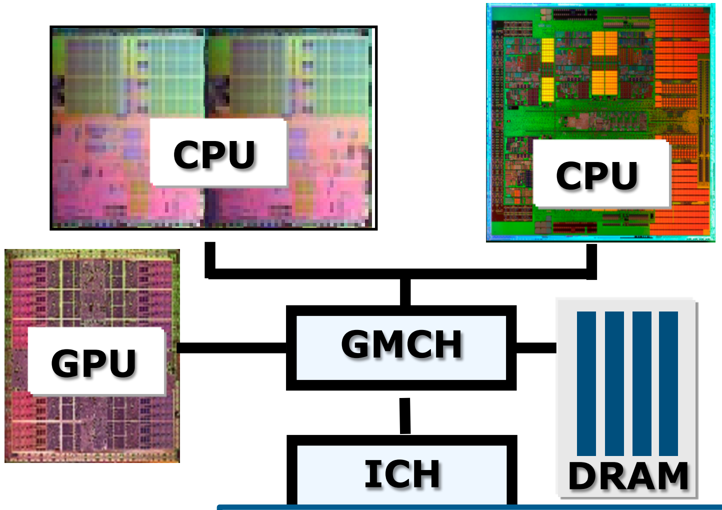
Graphics only
i.e. texture cache,
interpolation hardware

Memory Controllers

General compute + graphics
16 “Streaming multiprocessors”

500 Double-precision GFLOPs
16 Multiprocessors
32 ALUs/processor

The end of the discrete GPU

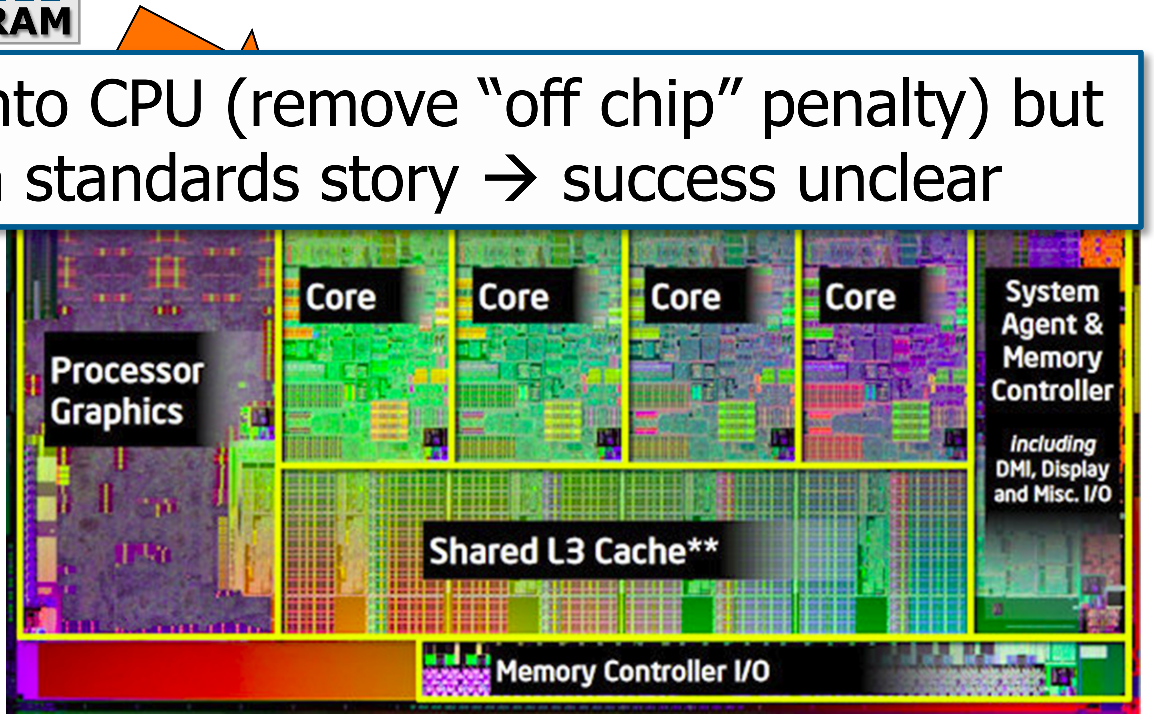


- A modern platform has:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?

Absorption into CPU (remove "off chip" penalty) but uncertain standards story → success unclear

- Curren

this functionality onto a single chip … mitigates the PCIe bottleneck in GPGPU computing!



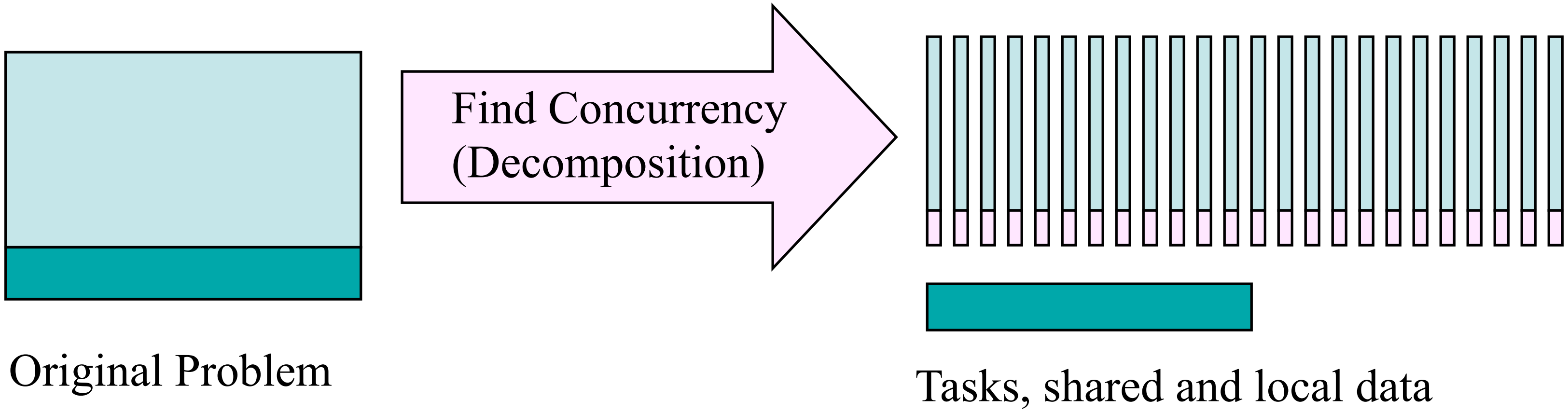
Intel® Core™ i5-2500K Desktop Processor (Sandy Bridge) Intel HD Graphics 3000 *(2011)*

GMCH = graphics memory control hub,
ICH = Input/output control hub

Outline

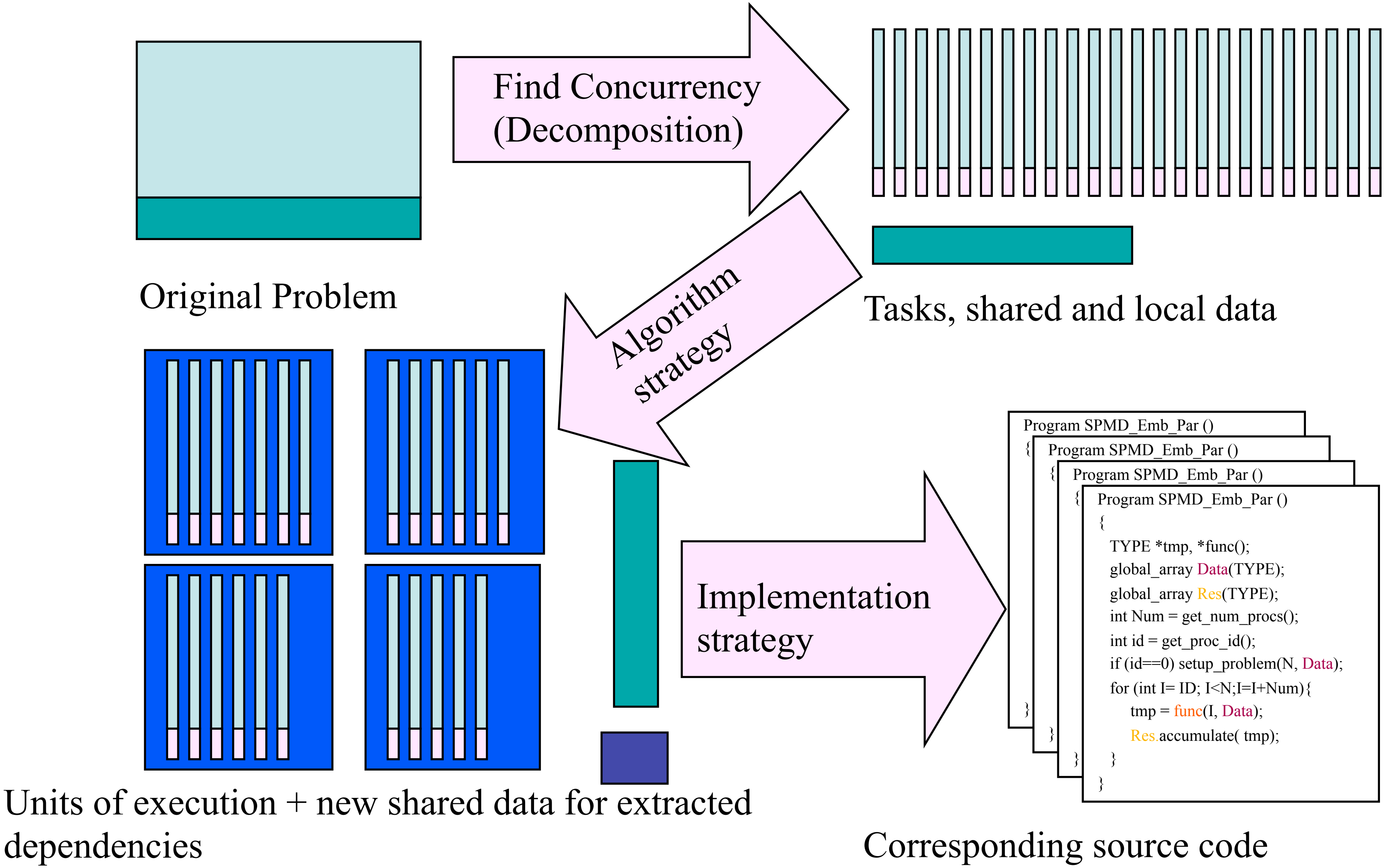
- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
- An introduction to parallel hardware
- ➡ Software for parallel systems: key design patterns
- Closing comments

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.
- Consider the following two problems. Can you come up with a task and data decomposition for these problems?
  - Graphics rendering pipeline.
  - Finding the best route between two cities on a map.

The Parallel programming process:



Original Problem

Tasks, shared and local data

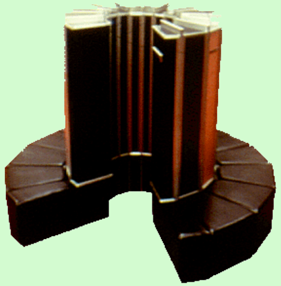
Units of execution + new shared data for extracted dependencies

```
Program SPMD_Emb_Par ()
{
  TYPE *tmp, *func();
  global_array Data(TYPE);
  global_array Res(TYPE);
  int Num = get_num_procs();
  int id = get_proc_id();
  if (id==0) setup_problem(N, Data);
  for (int I= ID; I<N;I=I+Num){
     tmp = func(I, Data);
     Res.accumulate( tmp);
  }
}
```

Corresponding source code

Parallel computing: It’s old

Vector Computers

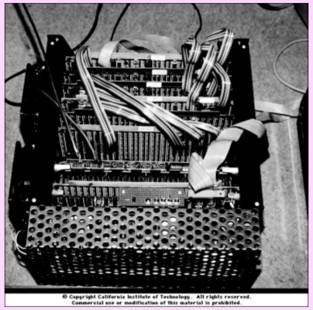


Cray 1 (1976)

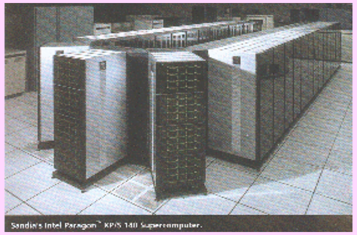
Cray 2 (1985)

Cray C-90 (1991)

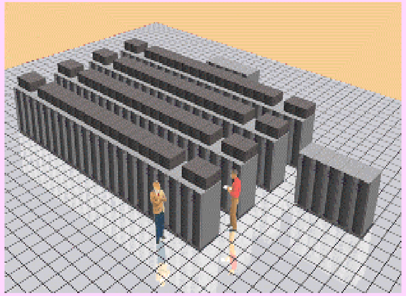
Massively Parallel Processors (MPP)


© Copyright California Institute of Technology. All rights reserved. Commercial use or modification of this material is prohibited.

Cosmic cube (1983)



Paragon (1993)



ASCI Red (1997)

Cluster Computers

Clusters (late 80’s)

Linux PC Clusters
(~1995)

Late 70’s

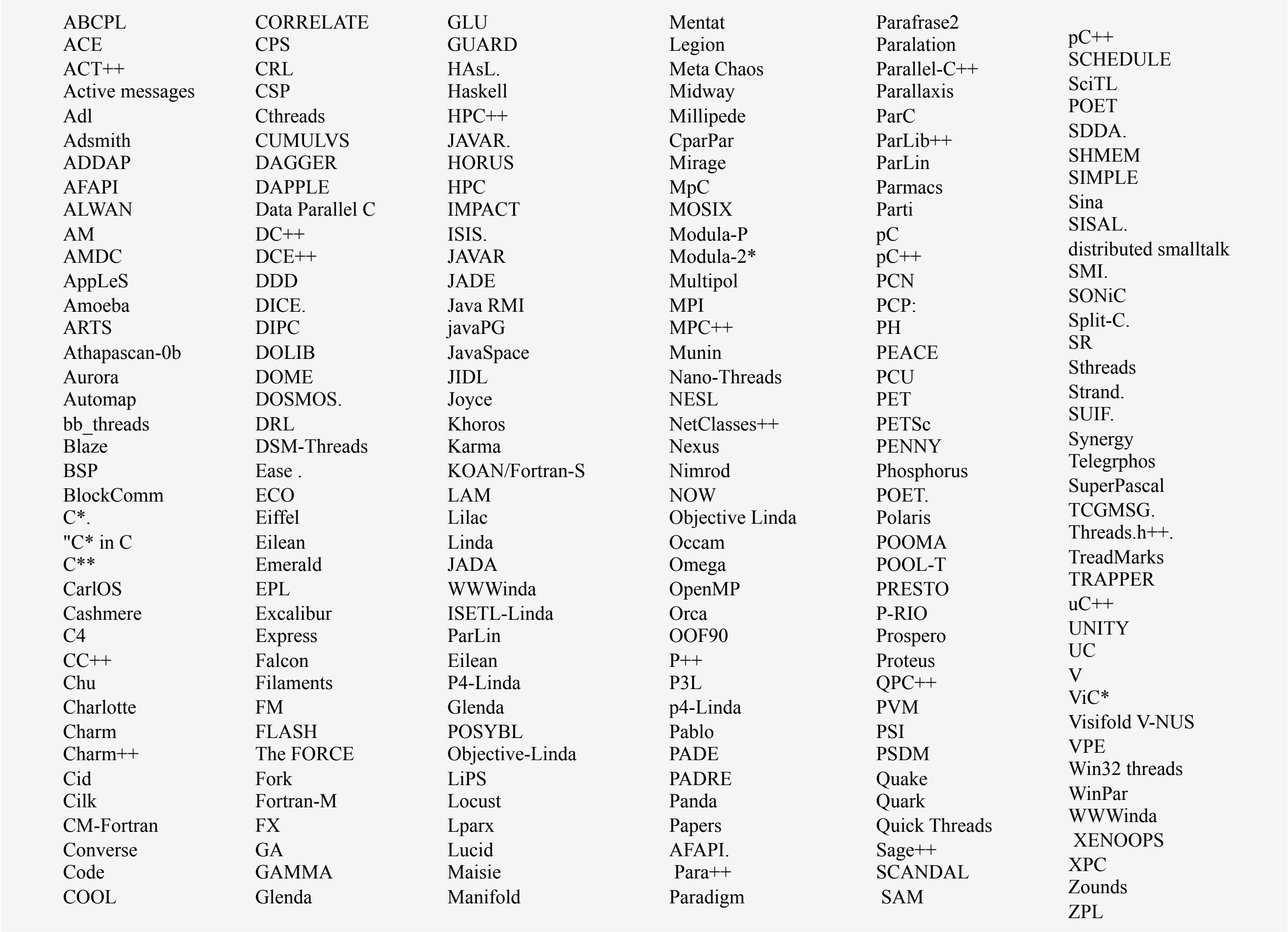
Late 80’s

Late 90’s

Third party names are the property of their owners.

We tried to solve the parallel programming problem by searching for the right programming environment

Parallel programming environments in the 90's



ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

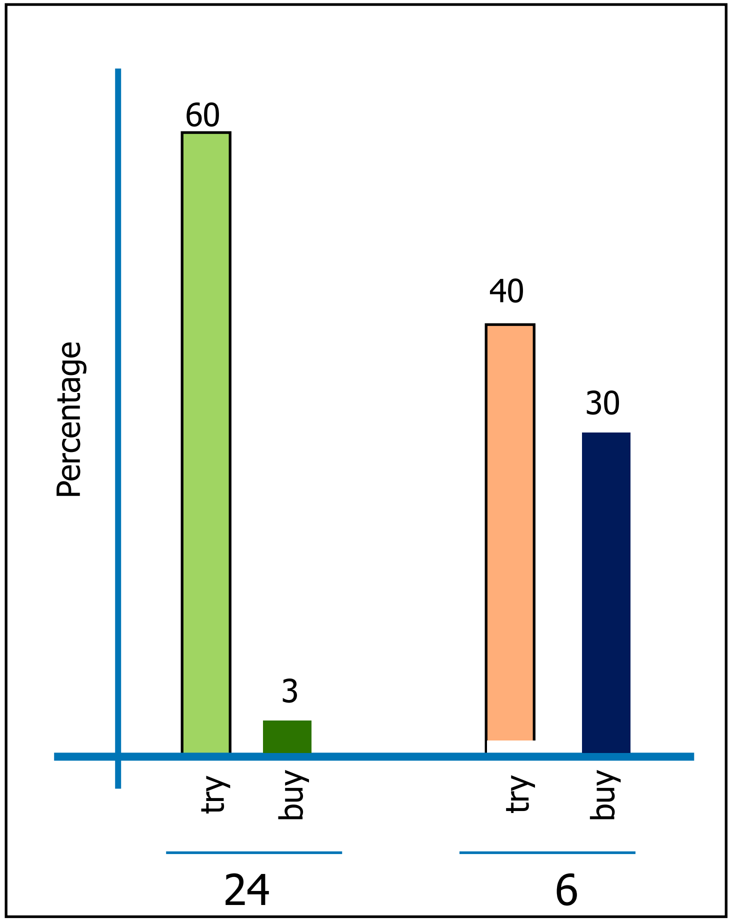
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Third party names are the property of their owners.

Language obsessions: More isn’t always better

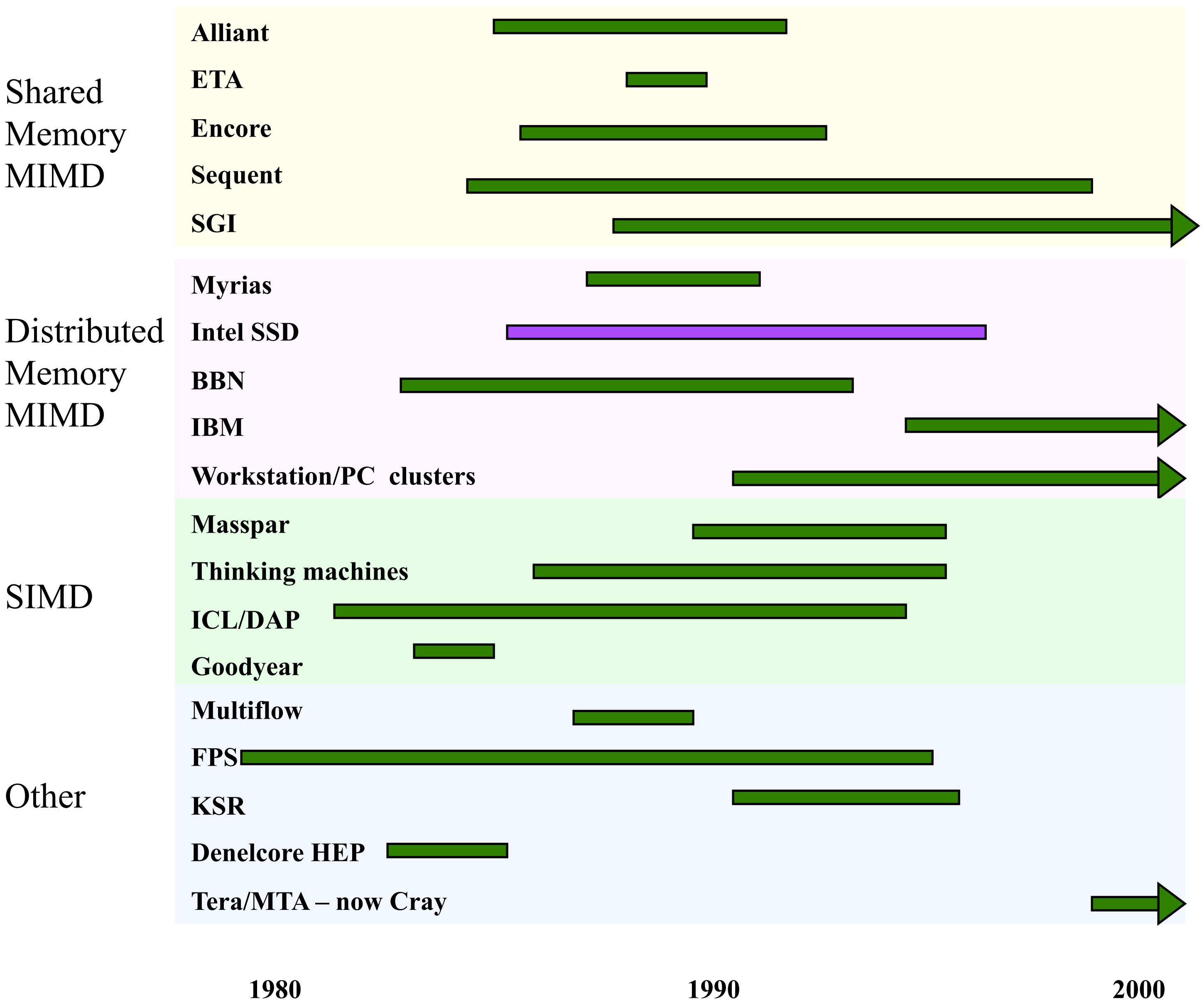
- The Draeger Grocery Store experiment consumer choice :
  - Two Jam-displays with coupon’s for purchase discount.
    - 24 different Jam’s
    - 6 different Jam’s
  - How many stopped by to try samples at the display?
  - Of those who “tried”, how many bought jam?



The findings from this study show that an extensive array of options can at first seem highly appealing to consumers, yet can reduce their subsequent motivation to purchase the product.

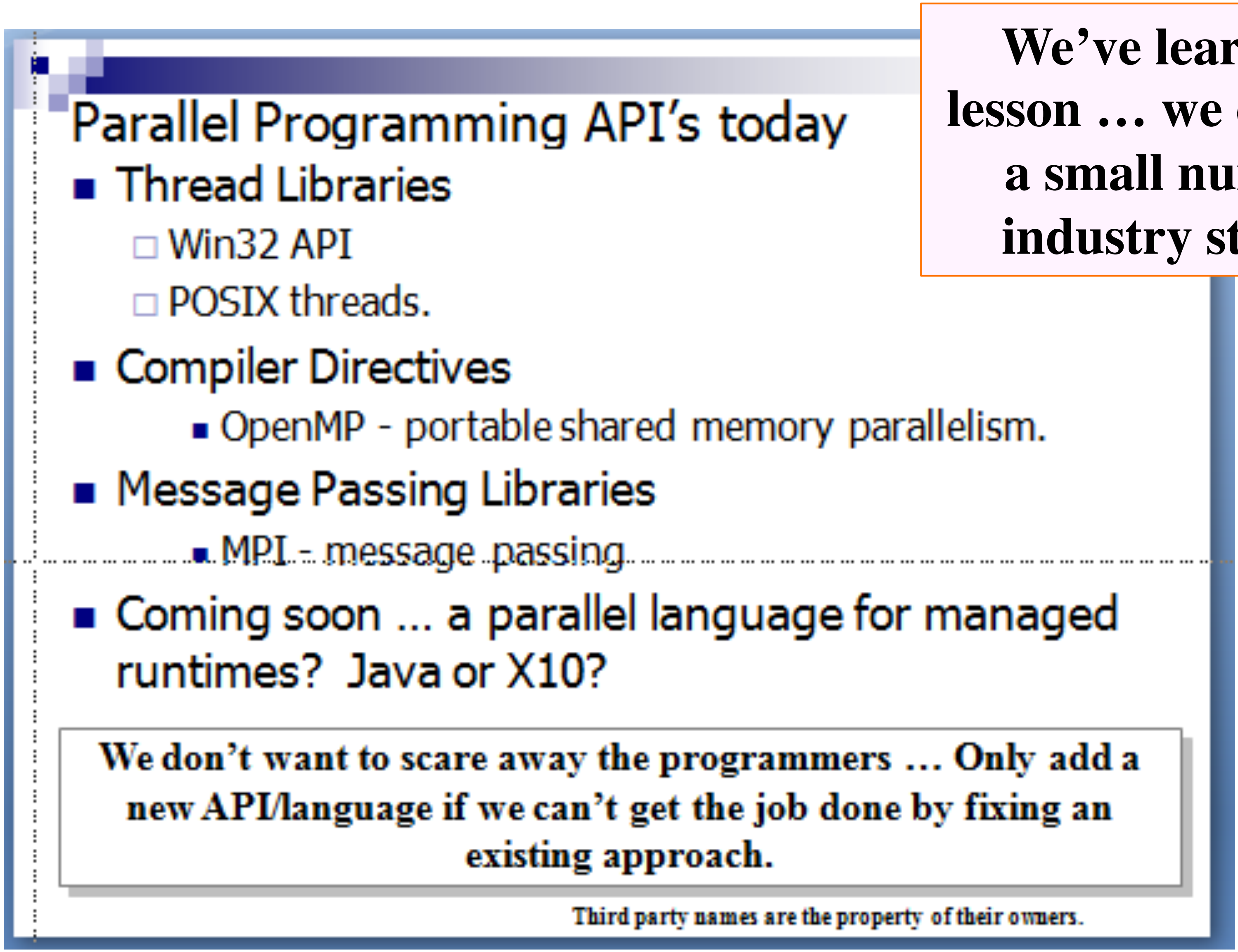
Iyengar, Sheena S., & Lepper, Mark (2000). When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 76, 995-1006.

Throwing new languages at the problem didn't work:
the "Dead Architecture Society"



Any product names on this slide are the property of their owners.

My optimistic view from 2005 …



Parallel Programming API's today

- Thread Libraries
  - □ Win32 API
  - □ POSIX threads.
- Compiler Directives
  - OpenMP - portable shared memory parallelism.
- Message Passing Libraries
  - MPI - message passing
- Coming soon … a parallel language for managed runtimes? Java or X10?

We don't want to scare away the programmers … Only add a new API/language if we can't get the job done by fixing an existing approach.

Third party names are the property of their owners.

We've learned our lesson … we emphasize a small number of industry standards

But we didn’t learn our lesson

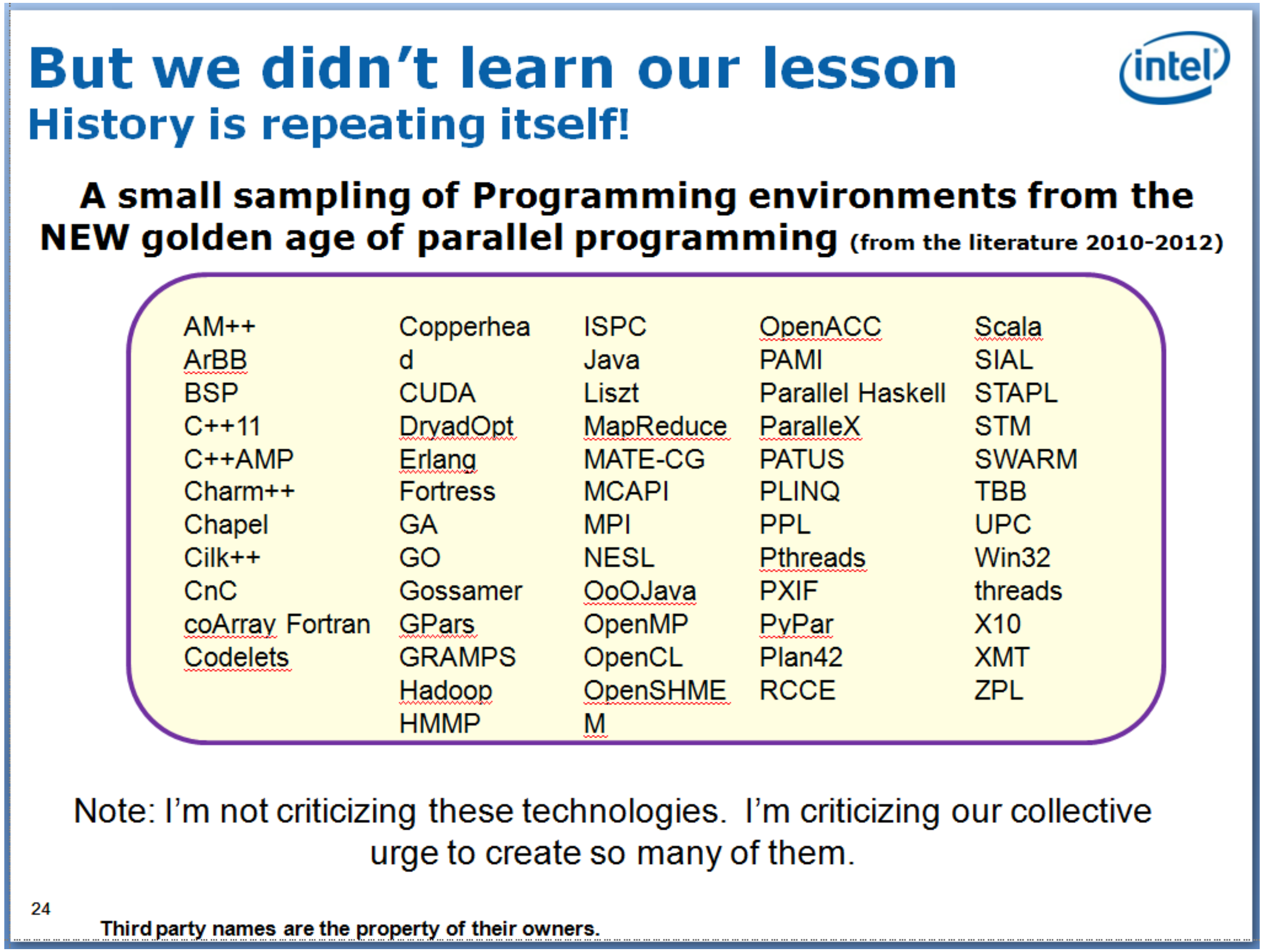
History is repeating itself!

A small sampling of Programming environments from the NEW golden age of parallel programming (from the literature 2010-2012)

- AM++
- ArBB
- BSP
- C++11
- C++AMP
- Charm++
- Chapel
- Cilk++
- CnC
- coArray Fortran
- Codelets
- Copperhead
- CUDA
- DryadOpt
- Erlang
- Fortress
- GA
- GO
- Gossamer
- GPars
- GRAMPS
- Hadoop
- HMPP
- ISPC
- Java
- Liszt
- MapReduce
- MATE-CG
- MCAPI
- MPI
- NESL
- OoOJava
- OpenMP
- OpenCL
- OpenSHMEM
- OpenACC
- PAMI
- Parallel Haskell
- ParalleX
- PATUS
- PLINQ
- PPL
- Pthreads
- PXIF
- PyPar
- Plan42
- RCCE
- Scala
- SIAL
- STAPL
- STM
- SWARM
- TBB
- UPC
- Win32 threads
- X10
- XMT
- ZPL

Note: I’m not criticizing these technologies. I’m criticizing our collective urge to create so many of them.

Maybe its time to try something different?



But we didn’t learn our lesson

History is repeating itself!

A small sampling of Programming environments from the NEW golden age of parallel programming **(from the literature 2010-2012)**

AM++
ArBB
BSP
C++11
C++AMP
Charm++
Chapel
Cilk++
CnC
coArray Fortran
Codelets

Copperhea
d
CUDA
DryadOpt
Erlang
Fortress
GA
GO
Gossamer
GPars
GRAMPS
Hadoop
HMMP

ISPC
Java
Liszt
MapReduce
MATE-CG
MCAPI
MPI
NESL
OoOJava
OpenMP
OpenCL
OpenSHME
M

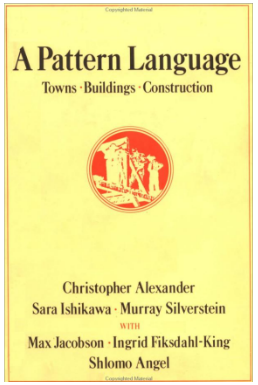
OpenACC
PAMI
Parallel Haskell
ParalleX
PATUS
PLINQ
PPL
Pthreads
PXIF
PyPar
Plan42
RCCE

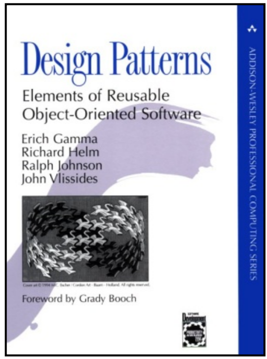
Scala
SIAL
STAPL
STM
SWARM
TBB
UPC
Win32
threads
X10
XMT
ZPL

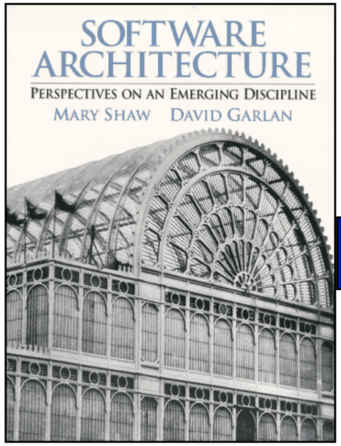
Note: I’m not criticizing these technologies. I’m criticizing our collective urge to create so many of them.

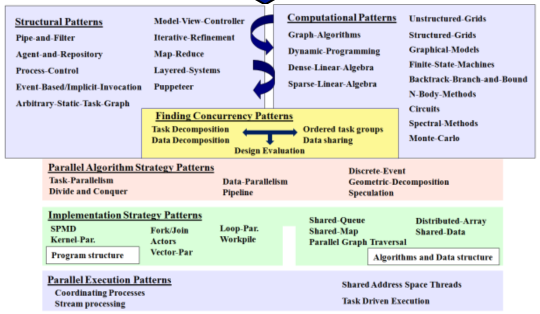
24

Third party names are the property of their owners.


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES





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

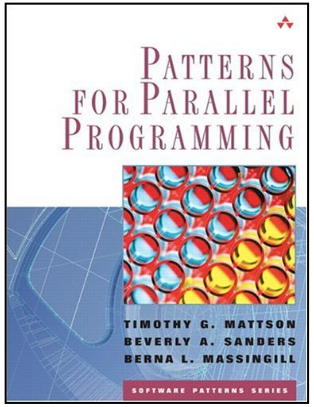
Parallel Algorithm Strategy Patterns
- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline
- Discrete-Event
- Geometric-Decomposition
- Speculation

Implementation Strategy Patterns
- SPMD
- Kernel-Par.
- Fork/Join
- Actors
- Vector-Par
- Loop-Par.
- Workpile
- Shared-Queue
- Shared-Map
- Parallel Graph Traversal
- Distributed-Array
- Shared-Data

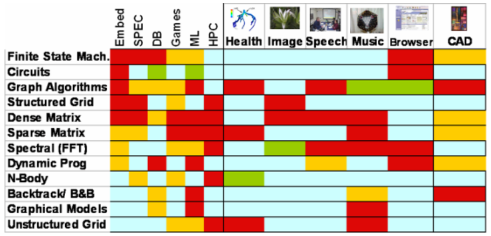
Program structure

Algorithms and Data structure

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



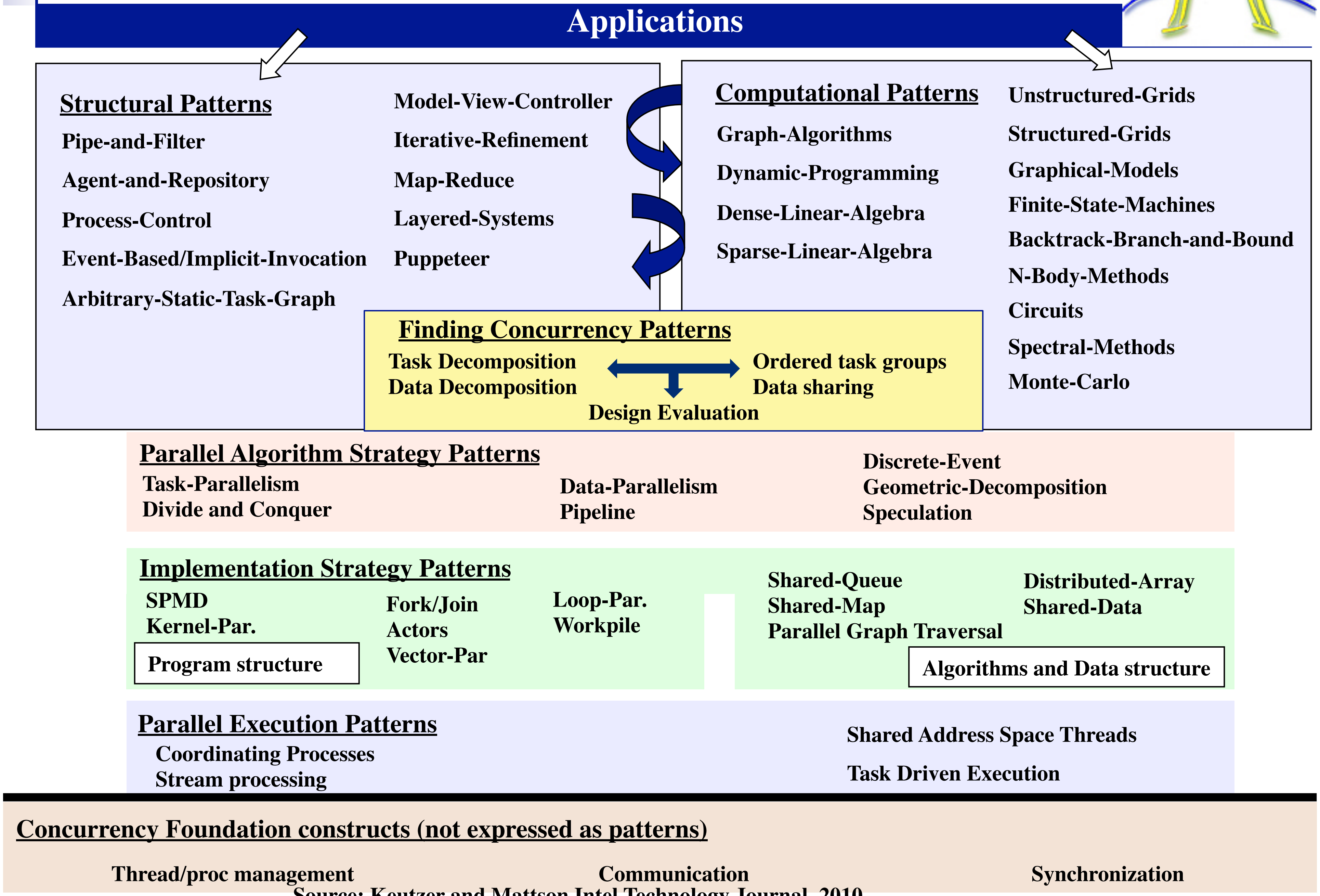
PLPP: Pattern language of Parallel Programming



	Embed	SPEC	DB	Games	ML	HPC	Health	Image	Speech	Music	Browser	CAD
Finite State Mach.	🟥	🟥	🟥	🟨	🟨	🟦	🟦	🟦	🟦	🟦	🟥	🟨
Circuits	🟥	🟦	🟩	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟥	🟦
Graph Algorithms	🟥	🟨	🟨	🟨	🟥	🟦	🟥	🟦	🟥	🟩	🟩	🟥
Structured Grid	🟥	🟥	🟦	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟦
Dense Matrix	🟥	🟥	🟨	🟥	🟥	🟥	🟦	🟥	🟥	🟥	🟦	🟨
Sparse Matrix	🟨	🟨	🟦	🟥	🟥	🟥	🟥	🟦	🟦	🟥	🟦	🟨
Spectral (FFT)	🟨	🟦	🟦	🟨	🟨	🟥	🟦	🟩	🟥	🟥	🟥	🟦
Dynamic Prog	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟨	🟦	🟥	🟨
N-Body	🟦	🟨	🟦	🟨	🟦	🟥	🟩	🟦	🟦	🟦	🟦	🟦
Backtrack/ B&B	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟨	🟦	🟥
Graphical Models	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟥	🟦	🟦
Unstructured Grid	🟦	🟦	🟦	🟨	🟨	🟥	🟥	🟦	🟦	🟥	🟦	🟦

13 dwarves

OPL Pattern Language (Keutzer & Mattson 2010)



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

Program structure

- SPMD
- Kernel-Par.
- Fork/Join
- Actors
- Vector-Par
- Loop-Par.
- Workpile

Algorithms and Data structure

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

Parallel Execution Patterns

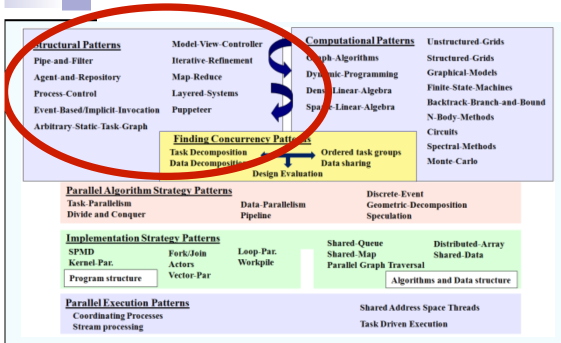
- Coordinating Processes
- Stream processing
- Shared Address Space Threads
- Task Driven Execution

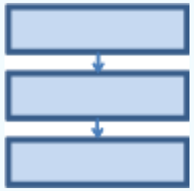
Concurrency Foundation constructs (not expressed as patterns)

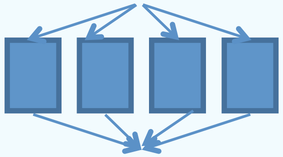
- Thread/proc management
- Communication
- Synchronization

Source: Keutzer and Mattson Intel Technology Journal, 2010

Pattern examples

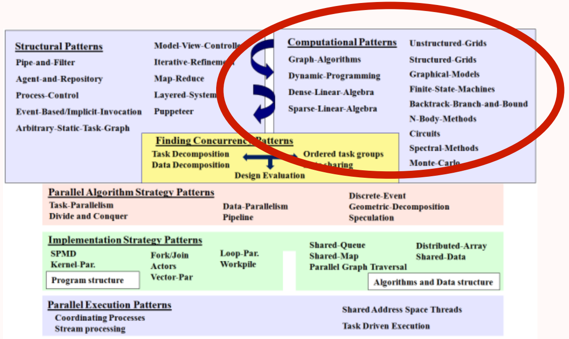


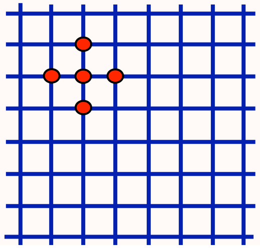


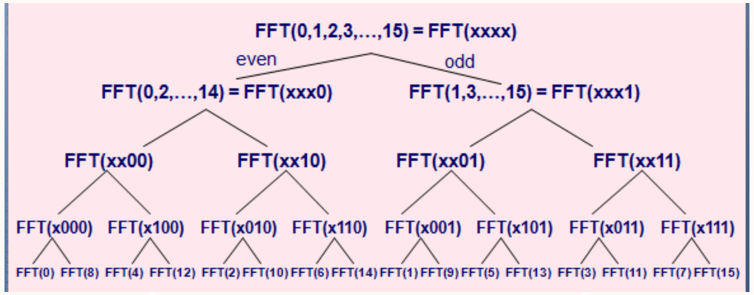


- Pipe-and-Filter
- Iterative refinement
- MapReduce

Structural Patterns: Define the software structure .. *Not* what is computed

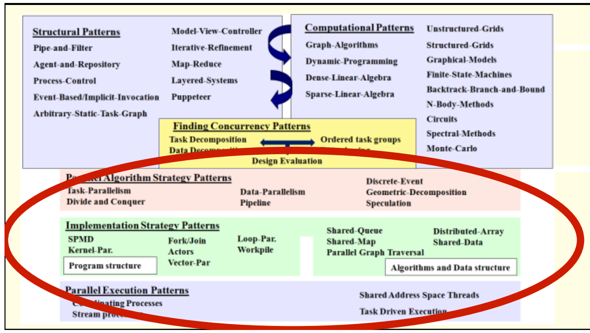






- Structured mesh
- Spectral methods

Computational Patterns: Define the computations “inside the boxes”

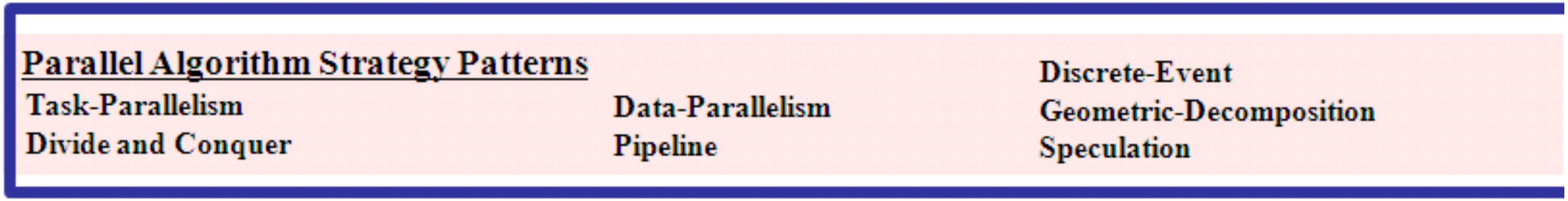


- Fork-join
- SPMD
- Data parallel

Parallel Patterns: Defines parallel algorithms

Seven strategies for parallelizing software

- These seven strategies for parallelizing software give us:
  - Names: so we can communicate better
  - Categories: so we can gather and share information
  - A palette (like an artist's palette) of approaches that is:
    - Necessary: we should consider them all and
    - Sufficient: once we have considered them all then we don't' have to worry that we forgot something



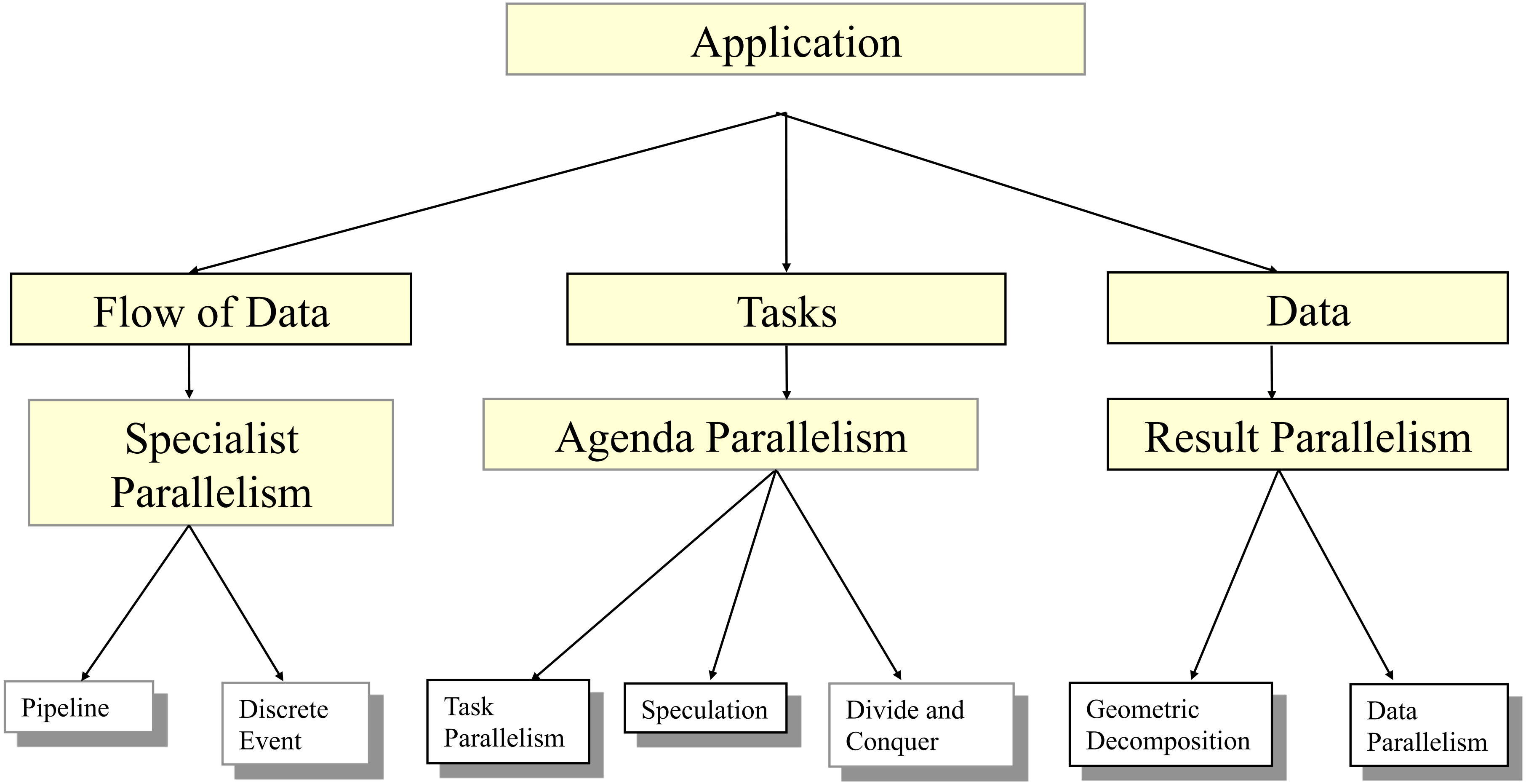
Parallel Algorithm Strategy Patterns

Task-Parallelism
Divide and Conquer

Data-Parallelism
Pipeline

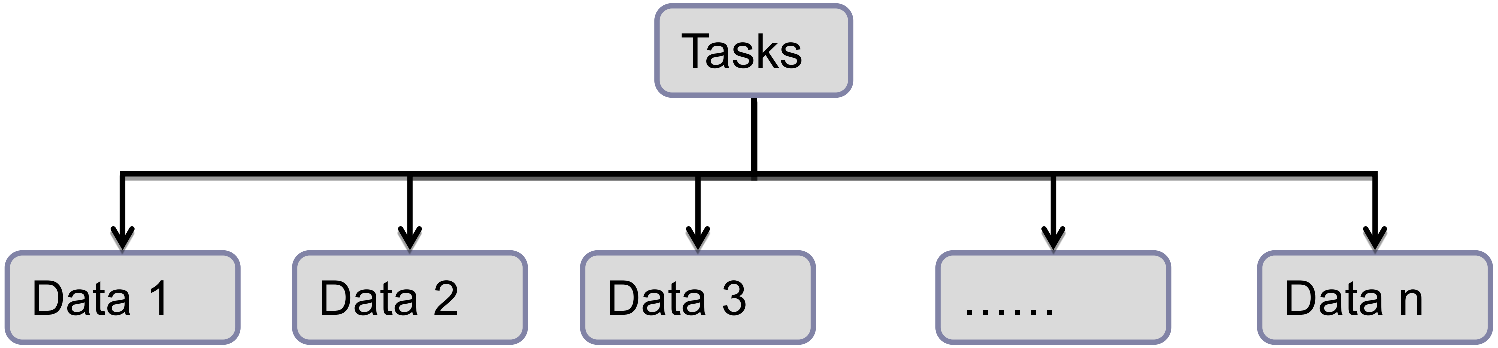
Discrete-Event
Geometric-Decomposition
Speculation

Parallel Algorithmic Strategies



Data Parallelism Pattern

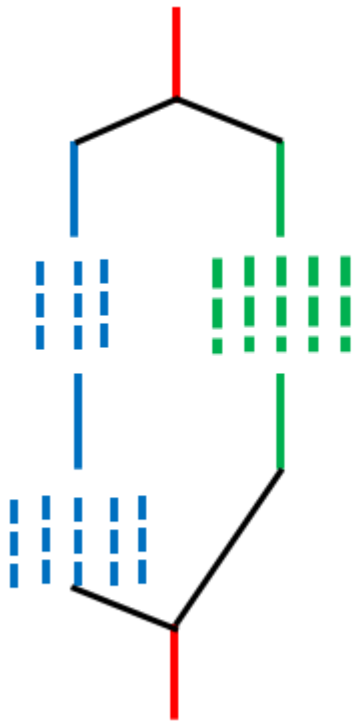
- Use when:
  - Your problem is defined in terms of collections of data elements operated on by a similar (if not identical) sequence of instructions; i.e. the concurrency is in the data.
- Solution
  - Define collections of data elements that can be updated in parallel.
  - Define computation as a sequence of collective operations applied together to each data element.



Task Parallelism Pattern

- Use when:
  - The problem naturally decomposes into a distinct collection of tasks

- 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

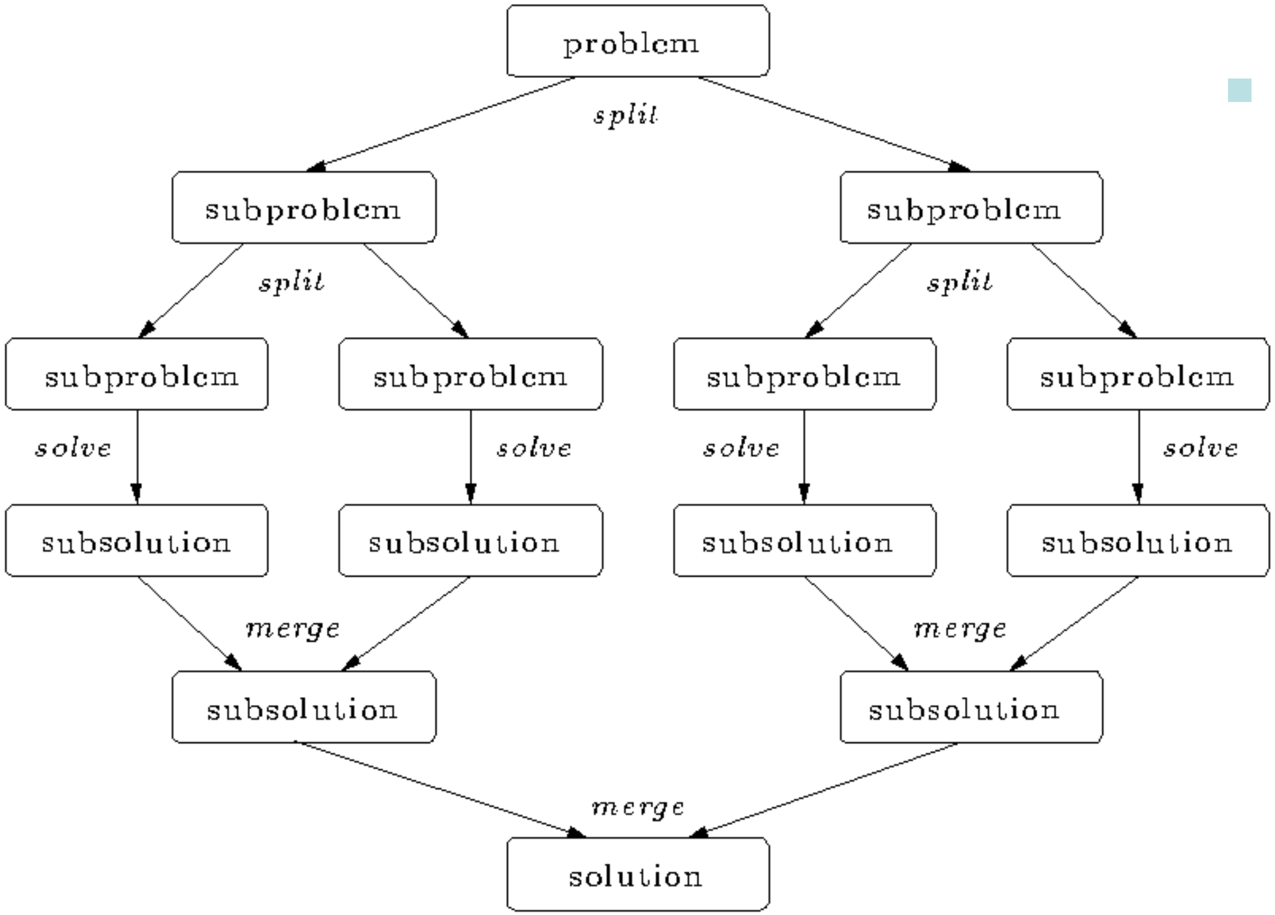
Its becoming common to associate this case as the prototypical "task parallel" approach … but to us old-timers, the previous two cases are overwhelming more common.

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

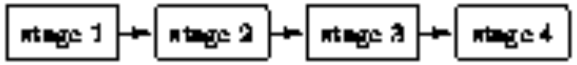
- Split the problem into smaller sub-problems. Continue until the sub-problems can be solve directly.



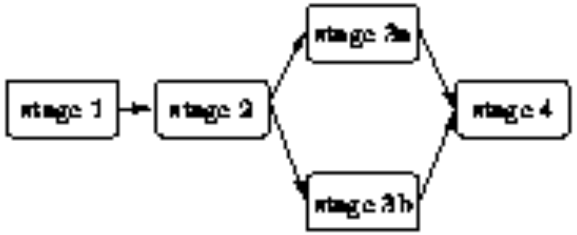
- 3 Options:
  - □ Do work as you split into sub-problems.
  - □ Do work only at the leaves.
  - □ Do work as you recombine.

Pipeline Pattern

- Use when:
  - Your problem can be described as data flowing through a sequence of computational stages
- Solution
  - □ Define a set of stages setup with data-flow connections between them.
  - □ Set up input/output channels to support data driven execution.
  - □ Parallelism comes from multiple stages acrive at one time.



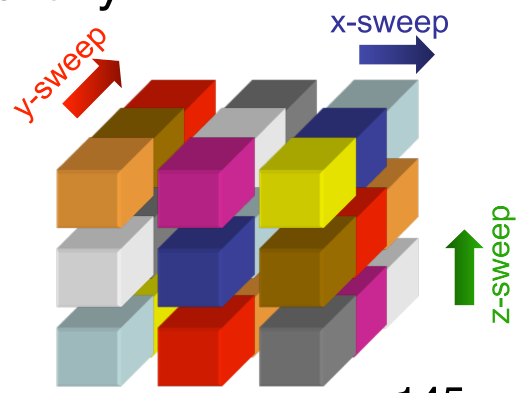
linear pipeline



nonlinear pipeline

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 grid and linear algebra computational strategy pattern.

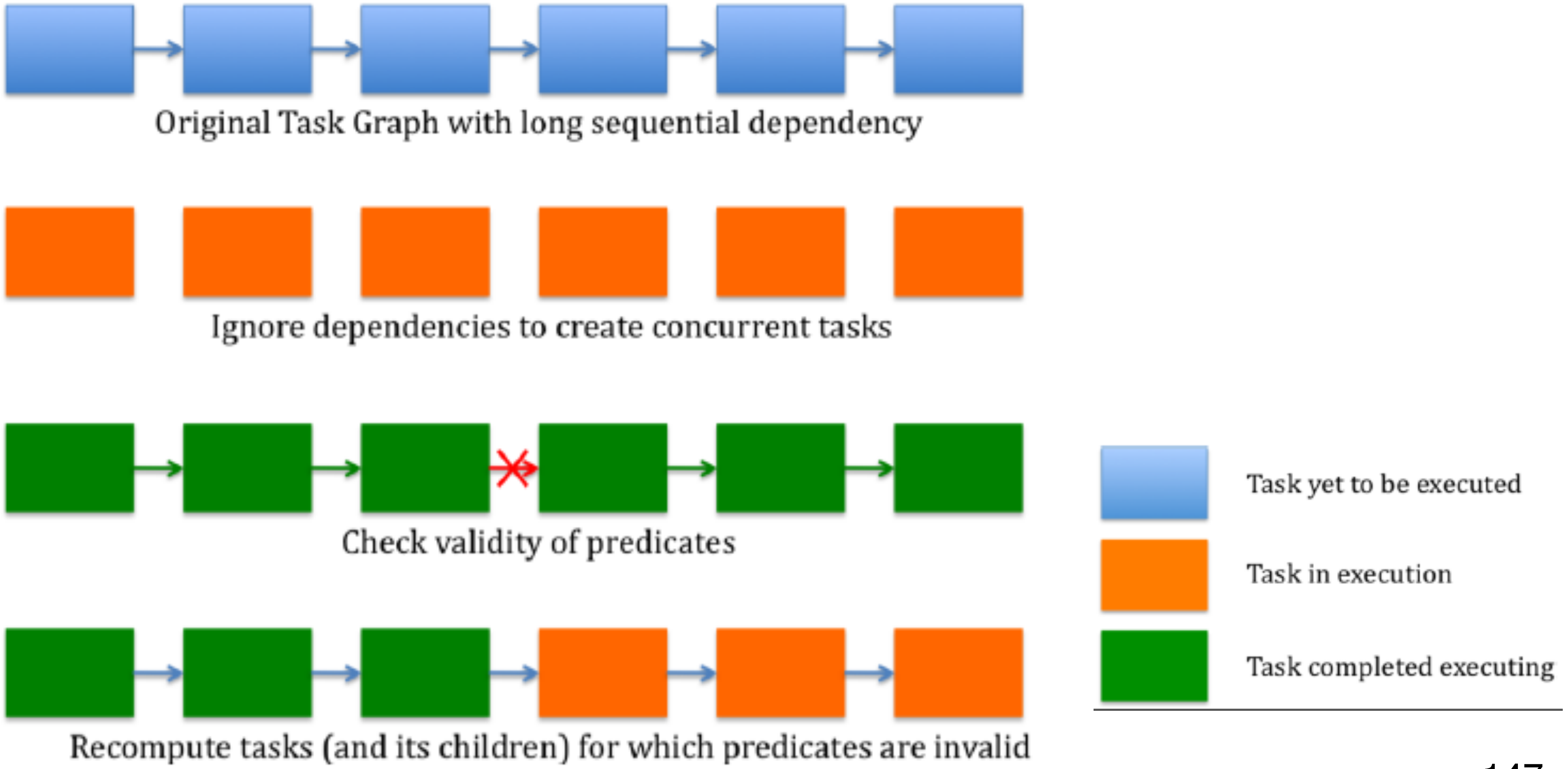


Speculation

- Use when:
  - Suppose that the computation has been decomposed into a number of tasks that are not completely independent, but where conflicts are expected to only infrequently occur when the computation is actually executed. Solution
- Solution:
  - An effective solution may be to just run the tasks independently, that is speculate that no conflicts will occur, and then clean up after the fact and retry in the rare situations where a conflict does occur. Two essential element of this solution are:
    1. Have an easily identifiable safety check to determine whether the computation ran without conflicts and can thus be committed
    2. The ability to rollback and re-compute the cases where conflicts occur.

Speculative Parallelism

- Speculative Parallelism:
  - Speculate on state of dependencies
  - Check validities of speculations
  - Recompute as needed to correct any mis-speculations

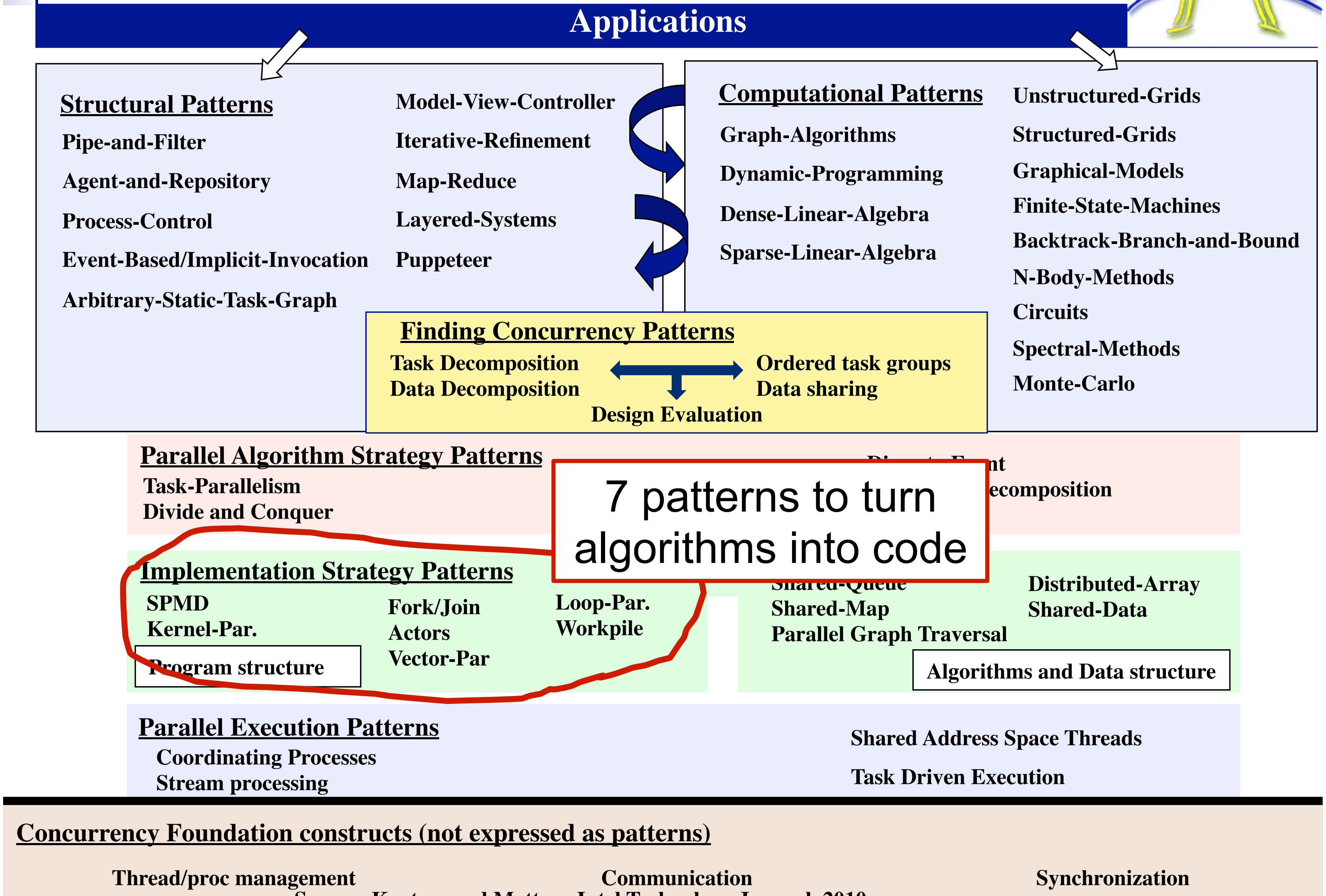

Original Task Graph with long sequential dependency
Ignore dependencies to create concurrent tasks
Check validity of predicates
Recompute tasks (and its children) for which predicates are invalid
Task yet to be executed
Task in execution
Task completed executing

Source: Narayanan Sundaram of UC Berkeley

Discrete-Event

- Use when:
  - The computation has been structured as loosely connected sequence of tasks that interact at unpredictable points in time.
- Solution
  - Setup an event handler infrastructure
  - Launch a collection of tasks whose interaction is handled through the event handler. The handler is an intermediary between tasks, and in many cases the tasks do not need to know the source or destination for the events.
- Note:
  - Discrete event is often used with problems, such as GUIs and discrete event simulations, that are handled with the Event-based implicit invocation, model-view-controller, or process control patterns.

OPL Pattern Language (Keutzer & Mattson 2010)



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
- Discrete-Event
- ecomposition

Implementation Strategy Patterns

- SPMD
- Kernel-Par.
- Fork/Join
- Actors
- Vector-Par
- Loop-Par.
- Workpile
- Program structure
- Shared-Queue
- Shared-Map
- Parallel Graph Traversal
- Distributed-Array
- Shared-Data
- Algorithms and Data structure

7 patterns to turn algorithms into code

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

Source: Keutzer and Mattson Intel Technology Journal, 2010

Seven strategies for implementing our algorithms as software

- These seven strategies for implementing our parallel algorithms give us:
  - Names: so we can communicate better
  - Categories: so we can gather and share information
  - A palette (like an artist’s palette) of approaches that is:
    - Necessary: we should consider them all and
    - Sufficient: once we have considered them all then we don’t’ have to worry that we forgot something

Implementation Strategy Patterns

SPMD
Actors

Fork/Join
Workpile

Loop-Parallel
Kernel-Parallel
Vector-Parallel

Program structure

Implementation Strategy patterns

- The most commonly used implementation strategy patterns:

SPMD	One program replicated, specialized by ID and NumProcs
Fork-Join	Single thread forks a team as needed and later joins
Work-pile	Create a pile of tasks for a set of workers to process
Loop-Parallel	Make expensive loops independent and use a “parallel for”
Vector-Parallel	Unroll loops to expose blocks, vector ops process blocks
Kernel-Parallel	Fine-Grained SPMD kernels . Large numbers to address little’s law.

- Programming models are often optimized around the needs of these patterns. For “our” programming models:
  - MPI: SPMD, work-pile
  - OpenMP: Loop-parallel, fork-join … SPMD on large NUMA systems.
  - OpenCL and CUDA: Kernel-parallelism
  - OpenACC: Loop-parallel and Kernel Parallel

Turning patterns into code: High level frameworks and the future of software development

Computer Games: one of the few (only?) consumer SW industries that have successfully embraced many-core industry-wide

- Divide Software group into two teams:
  - **Productivity programmers**:
    - 90% of the SW group.
    - Responsible for game content seen by a user (story line, characters, art, etc).
  - **Efficiency programmers**:
    - 10% of the software group
    - optimize the game software for specific platforms (C, assembly, etc)

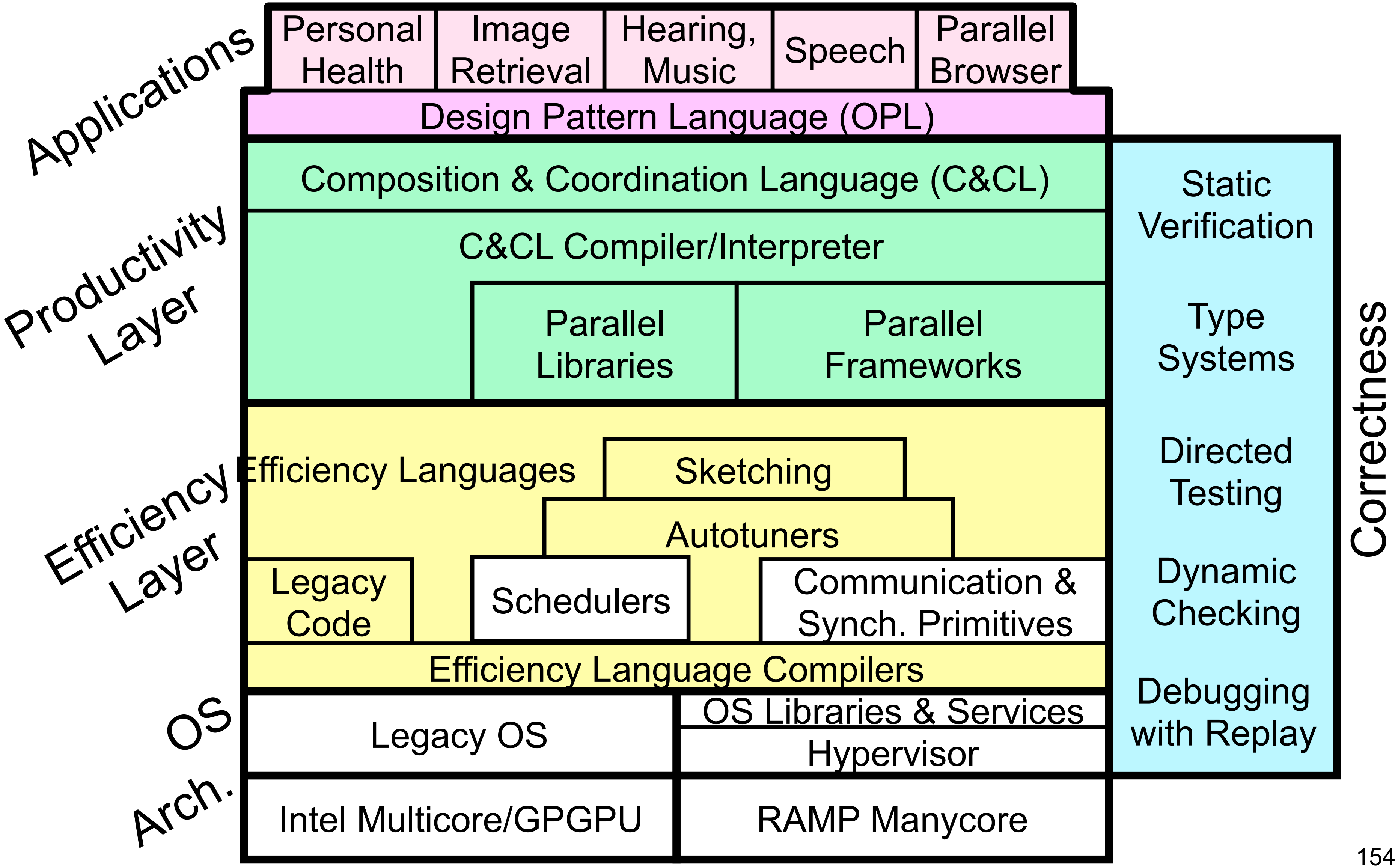
Army of Two (Electronic Arts)

Source: Tim Sweeney, Epic Games

- The full group needs to grapple with concurrency (tools cannot discover it automatically) … but only a small group (efficiency programmers) must understand the details of how to exploit concurrency in an efficient parallel program.

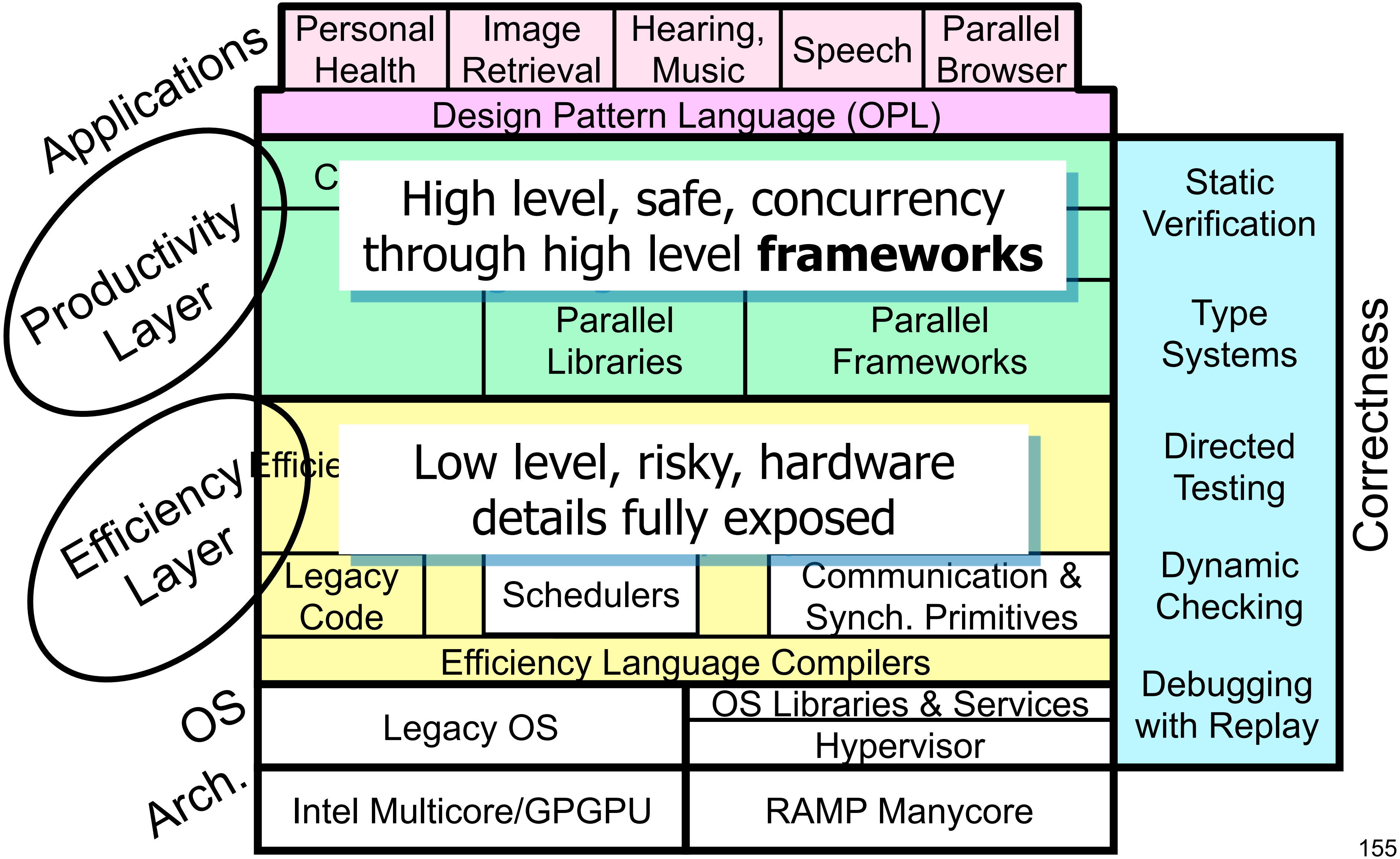
Par Lab (UC Berkeley) Overview

Easy to write correct software that runs efficiently on manycore



Par Lab (UC Berkeley) Overview

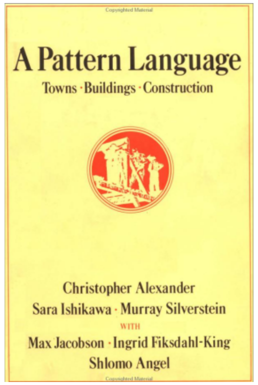
Easy to write correct software that runs efficiently on manycore

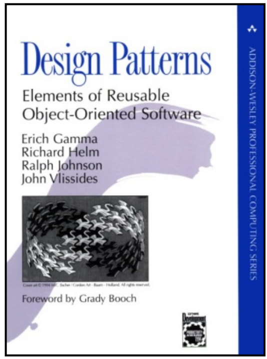


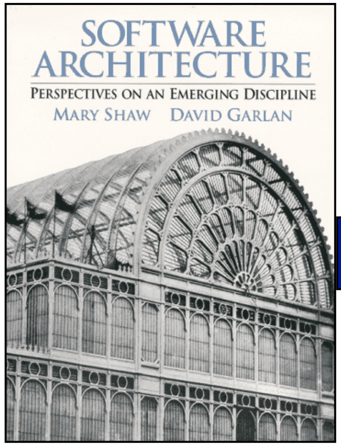
High level, safe, concurrency through high level **frameworks**

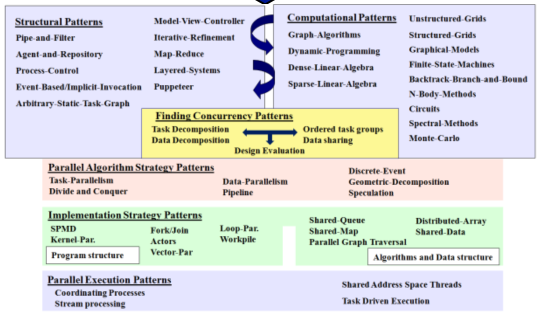
Low level, risky, hardware details fully exposed

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


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES





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

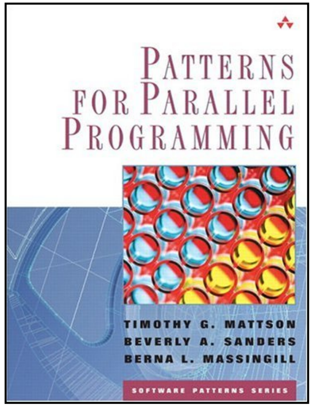
Implementation Strategy Patterns

SPMD
Kernel-Par.
Fork/Join
Actors
Vector-Par
Loop-Par.
Workpile
Program structure

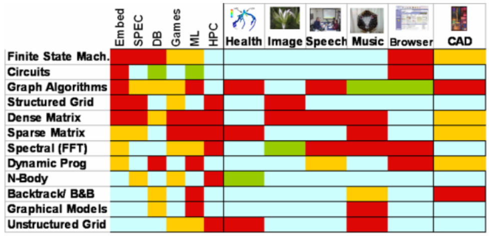
Shared-Queue
Shared-Map
Parallel Graph Traversal
Distributed-Array
Shared-Data
Algorithms and Data structure

Parallel Execution Patterns

Coordinating Processes
Stream processing
Shared Address Space Threads
Task Driven Execution



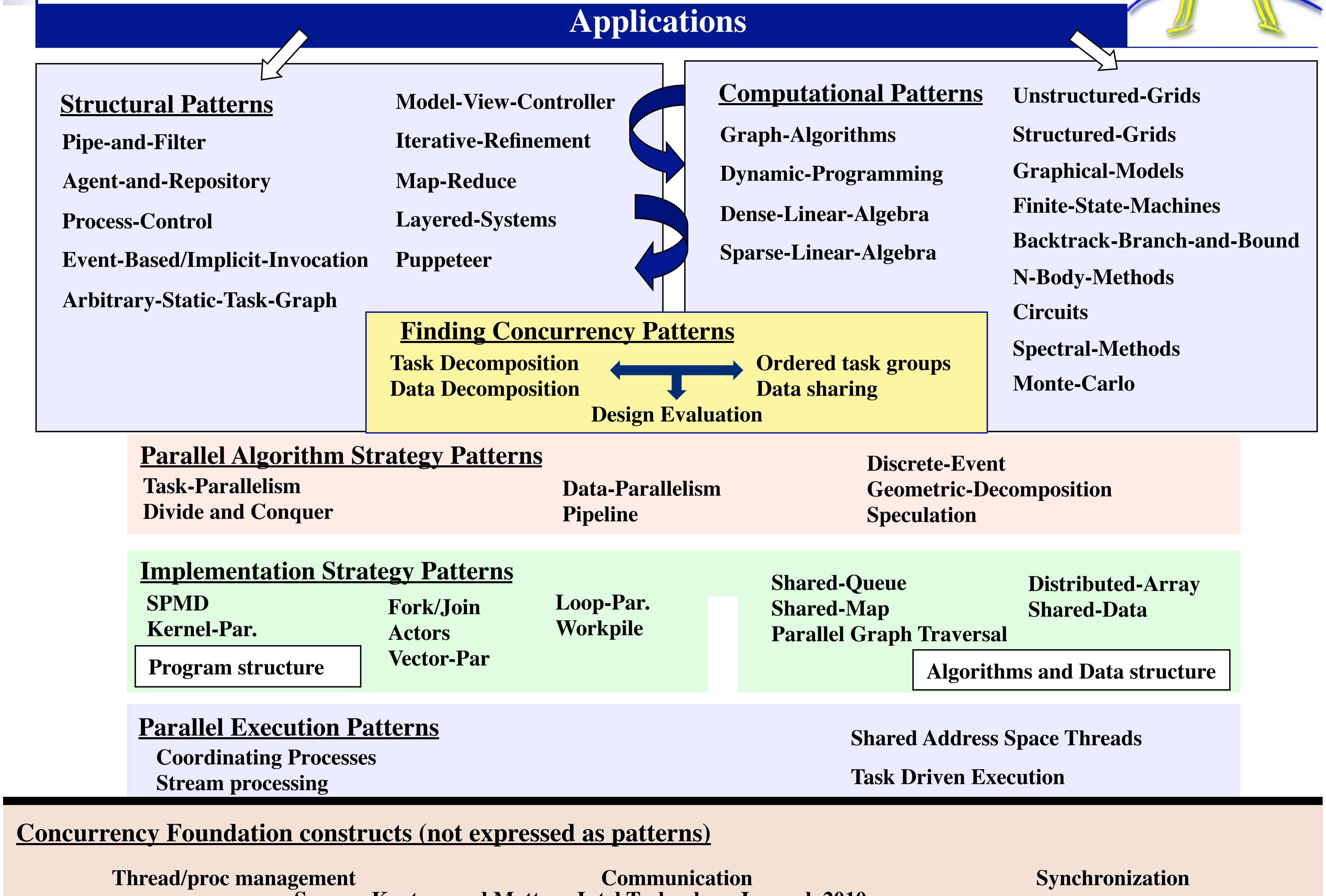
PLPP: Pattern language of Parallel Programming



	Embed	SPEC	DB	Games	ML	HPC	Health	Image	Speech	Music	Browser	CAD
Finite State Mach.	🟥	🟥	🟥	🟨	🟨	🟦	🟦	🟦	🟦	🟦	🟥	🟨
Circuits	🟥	🟦	🟩	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟥	🟦
Graph Algorithms	🟥	🟨	🟨	🟨	🟥	🟦	🟥	🟦	🟥	🟩	🟩	🟥
Structured Grid	🟥	🟥	🟦	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟦
Dense Matrix	🟥	🟥	🟨	🟥	🟥	🟥	🟦	🟥	🟥	🟥	🟦	🟨
Sparse Matrix	🟨	🟨	🟦	🟥	🟥	🟥	🟥	🟦	🟦	🟥	🟦	🟨
Spectral (FFT)	🟨	🟦	🟦	🟨	🟨	🟥	🟦	🟩	🟥	🟥	🟥	🟦
Dynamic Prog	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟨	🟦	🟥	🟨
N-Body	🟦	🟨	🟦	🟨	🟦	🟥	🟩	🟦	🟦	🟦	🟦	🟦
Backtrack/ B&B	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟨	🟦	🟥
Graphical Models	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟥	🟦	🟦
Unstructured Grid	🟦	🟦	🟦	🟨	🟨	🟥	🟥	🟦	🟦	🟥	🟦	🟦

13 dwarves

OPL Pattern Language (Keutzer & Mattson 2010)



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

Program structure

- SPMD
- Kernel-Par.
- Fork/Join
- Actors
- Vector-Par
- Loop-Par.
- Workpile

Algorithms and Data structure

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

Parallel Execution Patterns

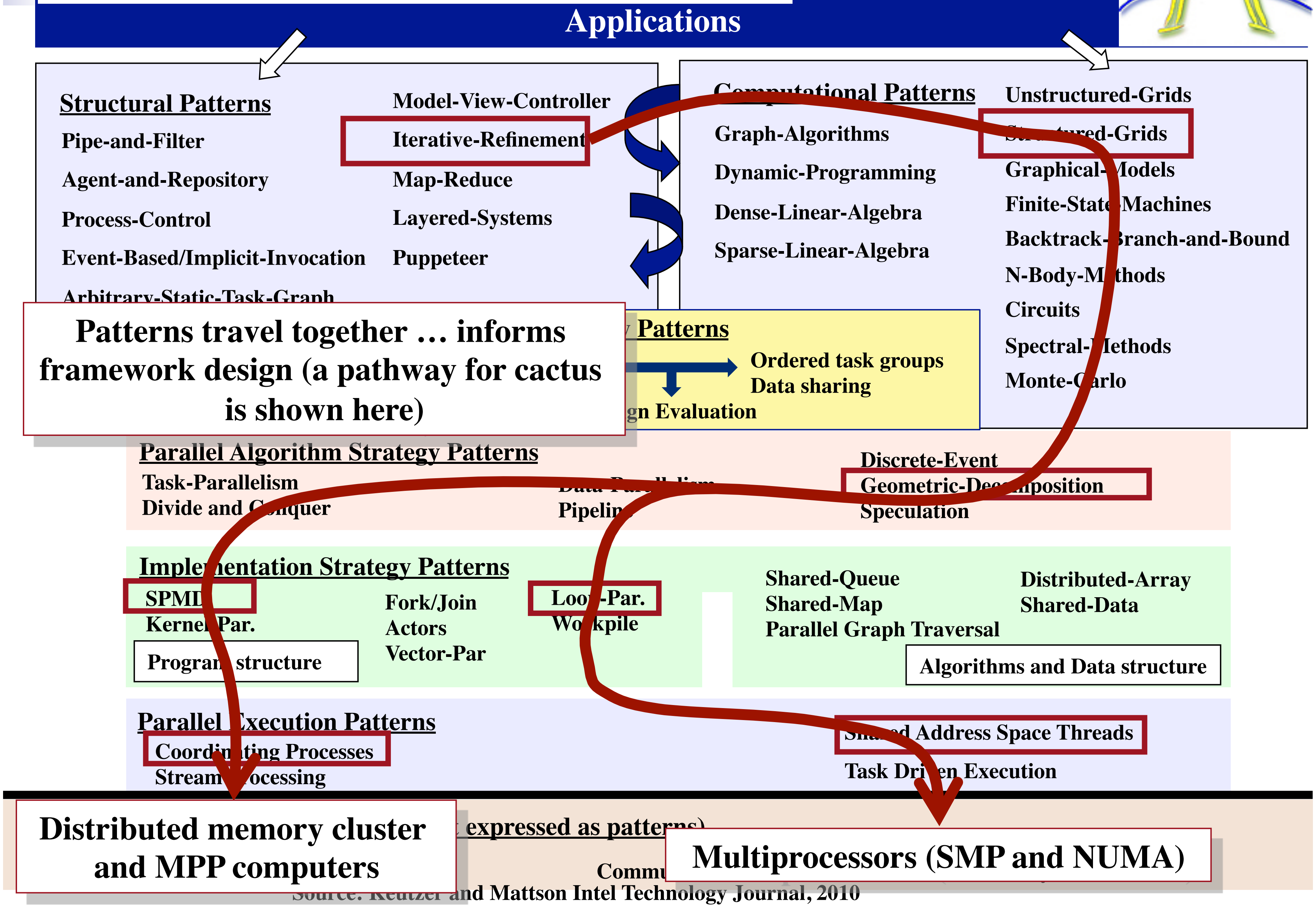
- Coordinating Processes
- Stream processing
- Shared Address Space Threads
- Task Driven Execution

Concurrency Foundation constructs (not expressed as patterns)

- Thread/proc management
- Communication
- Synchronization

Source: Keutzer and Mattson Intel Technology Journal, 2010

OPL Pattern Language



Patterns travel together … informs framework design (a pathway for cactus is shown here)

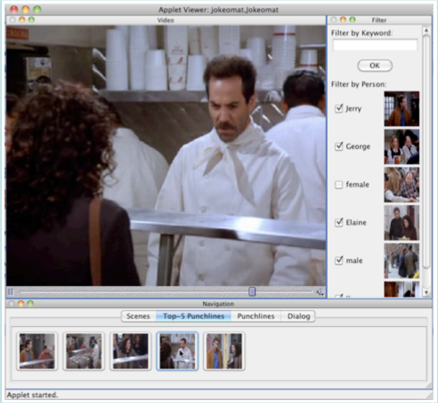
Distributed memory cluster and MPP computers

Multiprocessors (SMP and NUMA)

Source: Keutzer and Mattson Intel Technology Journal, 2010

Application driven Framework development

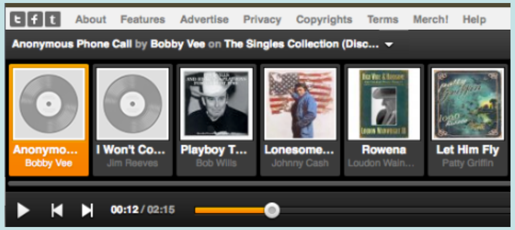
Speaker Diarization



- Who spoke when?
- 20 – 60 min meeting recordings

corpus.amiproject.org/

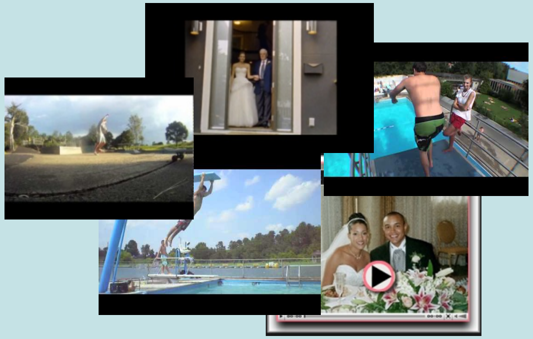
Music Recommendation



- Recommend songs most similar to a query

1 Million Song Dataset
labrosa.ee.columbia.edu/millionsong/

Video Event Detection



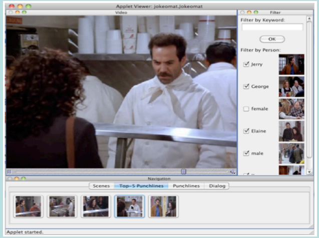
- Detect events in videos based on the soundtrack
- 1-50K video files

www-nlpir.nist.gov/projects/tv2011/

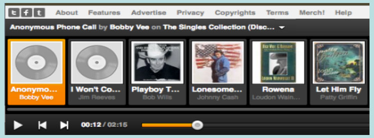
Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

Mining Patterns from Multi media Content Analysis

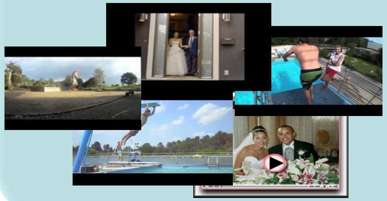
Speaker Diarization

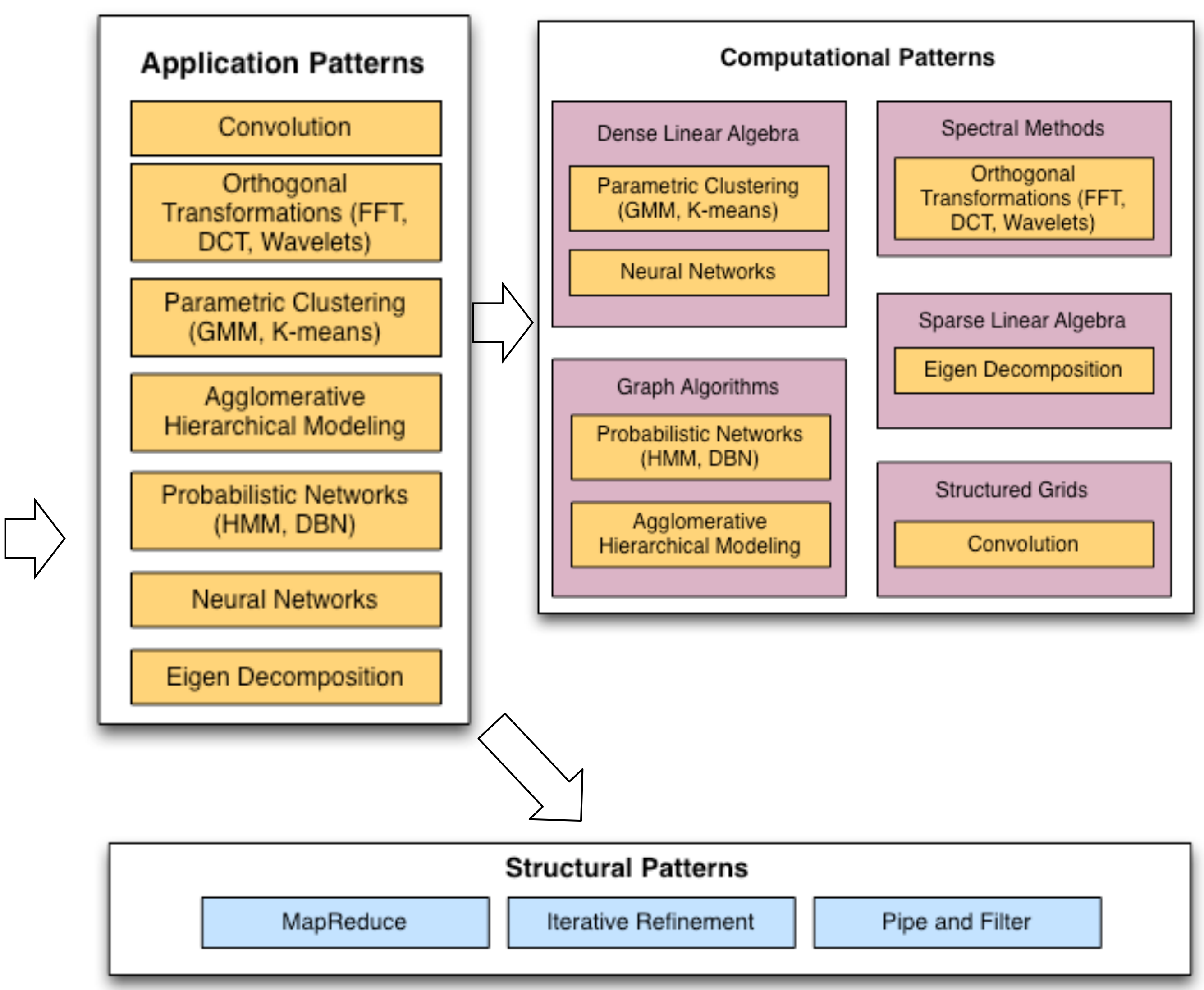


Music Recommendation



Video Event Detection





Application Patterns

- Convolution
- Orthogonal Transformations (FFT, DCT, Wavelets)
- Parametric Clustering (GMM, K-means)
- Agglomerative Hierarchical Modeling
- Probabilistic Networks (HMM, DBN)
- Neural Networks
- Eigen Decomposition

Computational Patterns

Dense Linear Algebra
- Parametric Clustering (GMM, K-means)
- Neural Networks

Spectral Methods
- Orthogonal Transformations (FFT, DCT, Wavelets)

Sparse Linear Algebra
- Eigen Decomposition

Graph Algorithms
- Probabilistic Networks (HMM, DBN)
- Agglomerative Hierarchical Modeling

Structured Grids
- Convolution

Structural Patterns

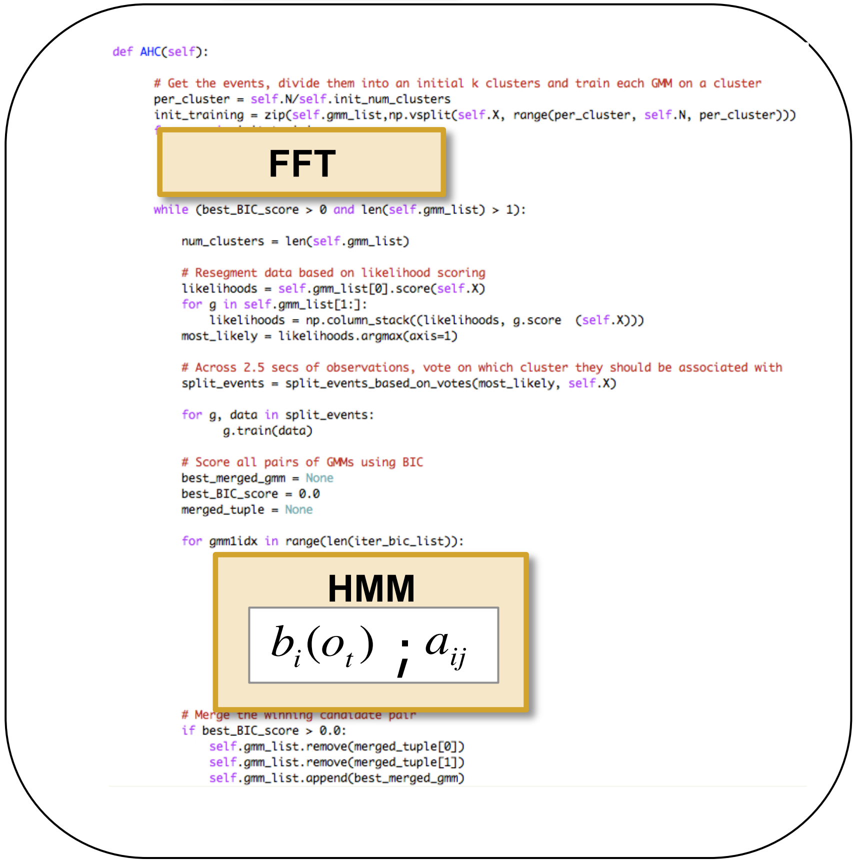
- MapReduce
- Iterative Refinement
- Pipe and Filter

Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

What the Framework Will Look Like

Library Components

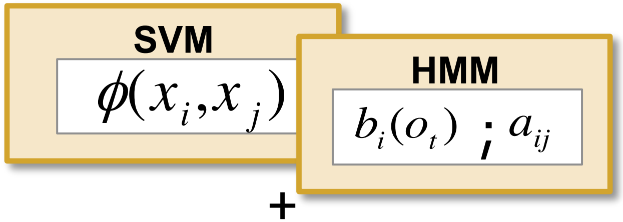
FFT
Wiener Filter
SVM
GMM training



```
def AHC(self):

    # Get the events, divide them into an initial k clusters and train each GMM on a cluster
    per_cluster = self.N/self.init_num_clusters
    init_training = zip(self.gmm_list,np.vsplit(self.X, range(per_cluster, self.N, per_cluster)))

    while (best_BIC_score > 0 and len(self.gmm_list) > 1):

        num_clusters = len(self.gmm_list)

        # Resegment data based on likelihood scoring
        likelihoods = self.gmm_list[0].score(self.X)
        for g in self.gmm_list[1:]:
            likelihoods = np.column_stack((likelihoods, g.score  (self.X)))
        most_likely = likelihoods.argmax(axis=1)

        # Across 2.5 secs of observations, vote on which cluster they should be associated with
        split_events = split_events_based_on_votes(most_likely, self.X)

        for g, data in split_events:
              g.train(data)

        # Score all pairs of GMMs using BIC
        best_merged_gmm = None
        best_BIC_score = 0.0
        merged_tuple = None

        for gmm1idx in range(len(iter_bic_list)):

        # Merge the winning candidate pair
        if best_BIC_score > 0.0:
            self.gmm_list.remove(merged_tuple[0])
            self.gmm_list.remove(merged_tuple[1])
            self.gmm_list.append(best_merged_gmm)
```

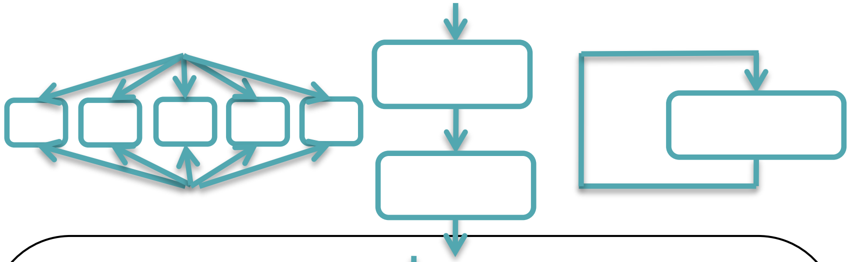
+

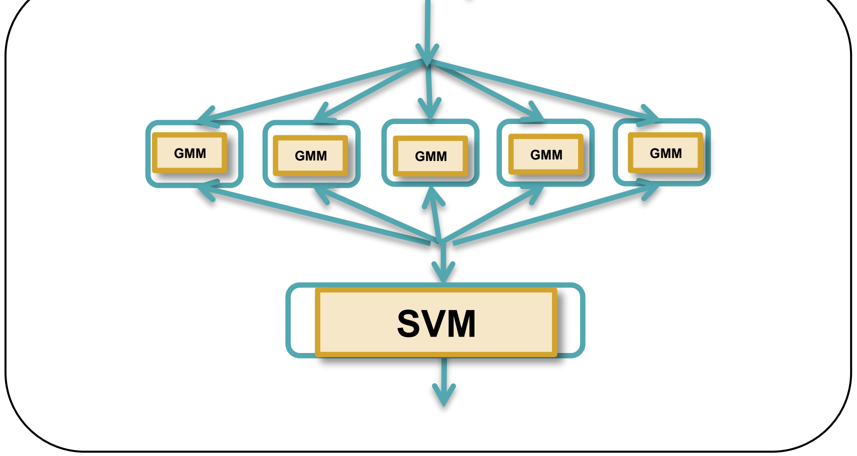
Customizable Components



+

Structural Patterns





Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

Library Component Example: GMM EM Training

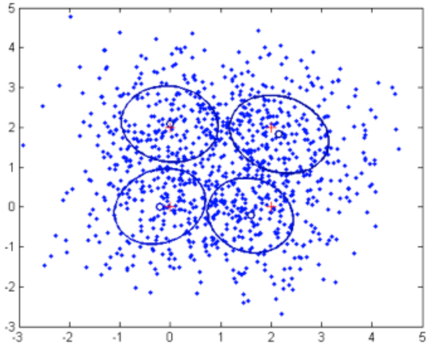
- GMM = probabilistic model for clustering data

GMM

$$p(x \mid \theta) = \sum_i \pi_i \frac{1}{(2\pi)^{\frac{m}{2}} \mid \Sigma_i \mid^{\frac{1}{2}}} exp\{-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1}(x-\mu_i)\}$$

where $\theta_i = (\pi_i, \mu_i, \Sigma_i)$

- Expectation Maximization (EM) Algorithm for training GMMs (find mean, covariance and weights)
  - Multiple parallelization strategies based on problem size and hardware platform characteristics
  - Written in C/CUDA/Cilk+ templates
  - Select best-performing strategy at runtime

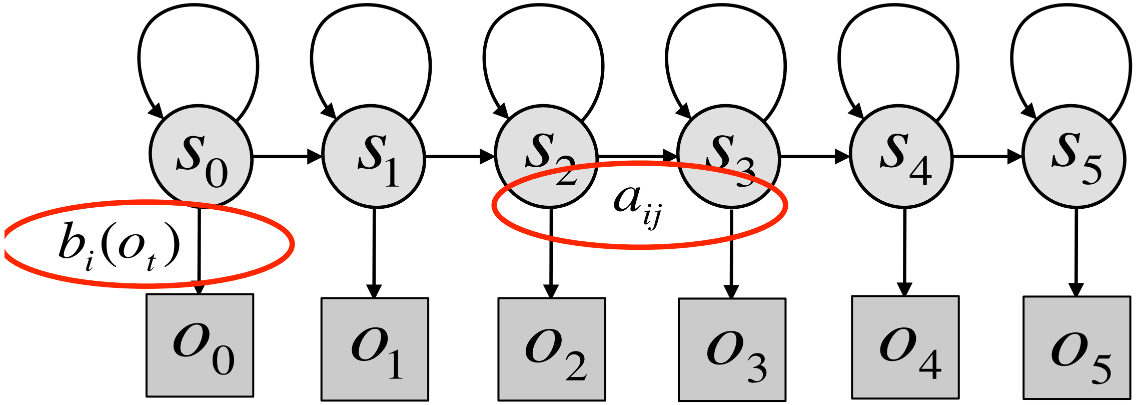


Example GMM in two dimensions
(Source: www.mathworks.com)

"CUDA-level Performance with Python-level Productivity for Gaussian Mixture Model Applications" Henry Cook, Ekaterina Gonina, Shoaib Kamil, Gerald Friedland, David Patterson, Armando Fox. In Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar'11). USENIX Association, Berkeley, CA, USA.

Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

Customizable Component Example: HMM EM Training



s_i - hidden state *i*

o_t - observation at time t

a_{ij} - Transition probability from state *i* to state *j*

$b_i(o_t)$ - observation probability of obs t given state i

- Model temporal sequences

HMM

$b_i(o_t)$ **;** a_{ij}

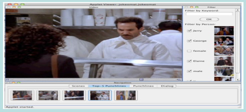
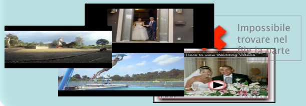
- Training – find parameters A and B given observation sequence O using the Baum-Welsh algorithm (generalized EM)
- Decoding – find the state sequence S that best matches an observation sequence O (Viterbi algorithm)

○ - customizable element

Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

PyCASP Productivity

- Create a tractable framework scope by using patterns
- Applications written in Python
  - Glue language

Application	Lines of Python Code	Approximated LOC Reduction (vs. C/C++)
Speaker Diarization 	50	60X
Music Recommendation	500	10-50X
Video Event Detection 	50 + 1	60X + 20X

Specializer	LOC
GMM	1500 C/ CUDA 800 Python
Map-Reduce	80 Python

Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

Efficiency

- Speaker Diarization
  - Average faster-than-real-time factor &error rate
  - Averaged across 12 meetings (AMI corpus) [1]

Implementation	Diarization Error Rate	Faster-than-real-time factor
State-of-the-art C++	~22%	1X
PyCASP	24.7%	115X

[1] E. Gonina, G. Friedland, H. Cook and K. Keutzer. "Fast Speaker Diarization Using a High-Level Scripting Language" In Proceedings of IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), Dec 11-15, 2011, Waikoloa, Hawaii

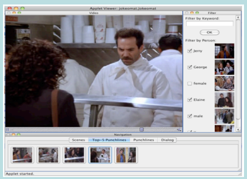
Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

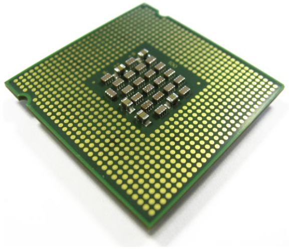
PyCASP Portability

- Speaker Diarization
  - Average faster-than-real-time factor
  - Intel Westmere and two CUDA GPUs
  - Averaged over 12 meetings (AMI corpus)
    - (Augmented Multi-party Interaction corpus)
    - 100 hours of meetings captured using many syn recording devices

Platform	Faster-than-real-time factor
Intel Westmere	56x
Nvidia GTX285	101x
Nvidia GTX480	115x

Speaker Diarization



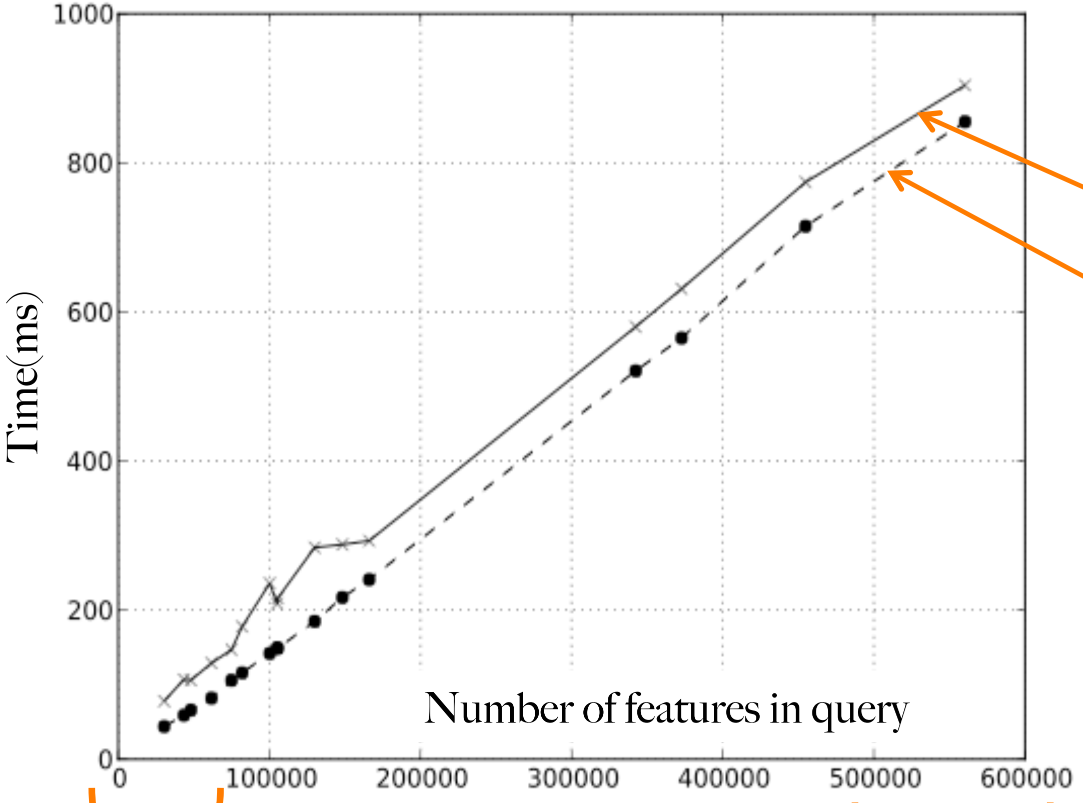


Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

PyCASP Scalability

- Music Recommendation

Recommendation Time vs. Query Size – 1 Million Songs



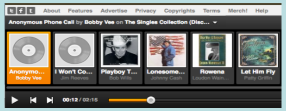
Total recommendation time

Query GMM training time

30 songs
“Elton John”

400-500 songs
“Elton John or Eric Clapton or Lady Gaga or Britney Spears”

Music Recommendation



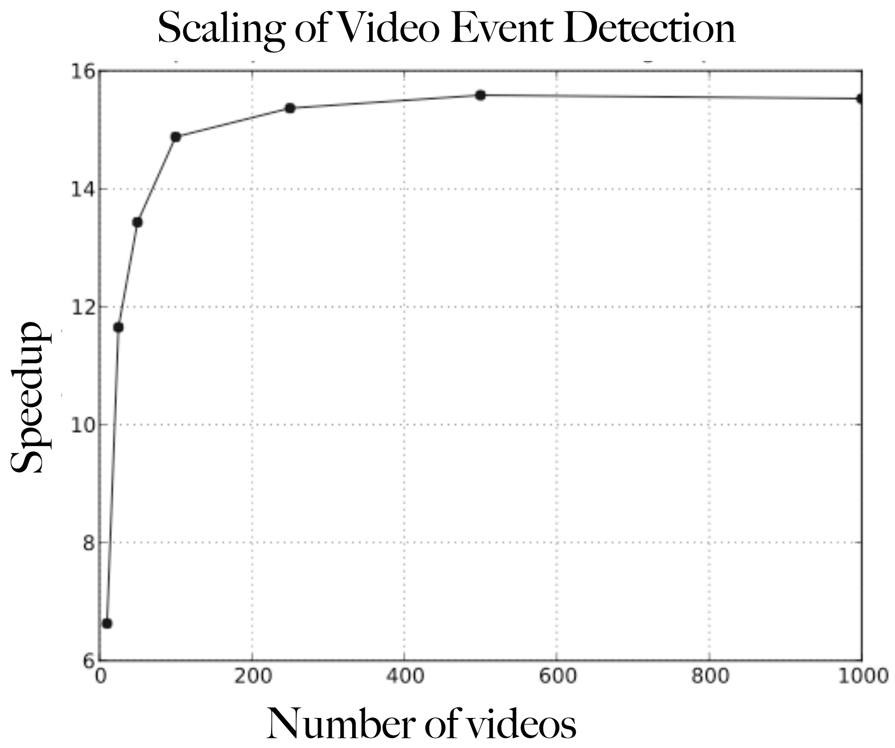
Under 1 second recommendation time for all queries!

Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

PyCASP Scalability

- Video Event Detection

- Nearly-optimal scaling on a cluster of GPUs:
  - 15.5x on 16-node cluster for 500 and 1000 videos



Source: Keutzer and Gonina, non-numeric computing workshop, Summer 2012.

Outline

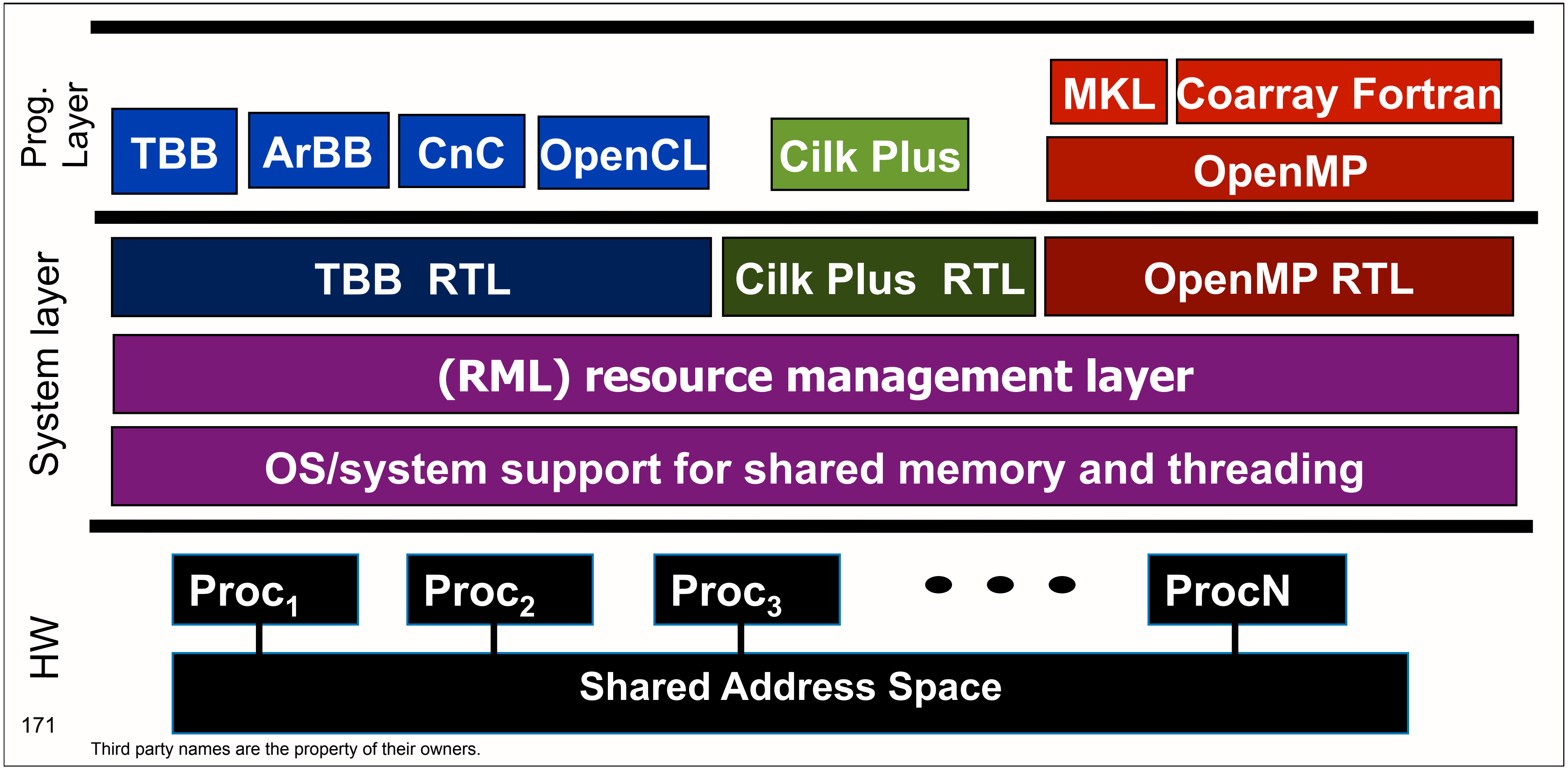
- Motivation: We all must be parallel programmers
- Key concepts in parallel Computing
- An introduction to parallel hardware
- Software for parallel systems: key design patterns
- ➡ Closing comments

Writing Parallel software isn’t enough

- Modern applications are built from multiple modules and libraries.
- We can parallelize them all … but ultimately they need to run together and “do the right thing” when put together.
- This is the parallel composition problem.
  - How do you manage resources between different modules?
  - How do you maintain isolation between modules to keep them from colliding?
  - How do you optimize resource allocation to produce the best results?
- We do not have a good solution to this problem. The starting point is a common runtime to support multiple programming models.

For example … consider what's happening at Intel?

Intel has developed a whole series of programming models that map onto three different runtime libraries (RTL) that all sit on top of a common RML. This gives us a foundation to work on as we attack the composability problem

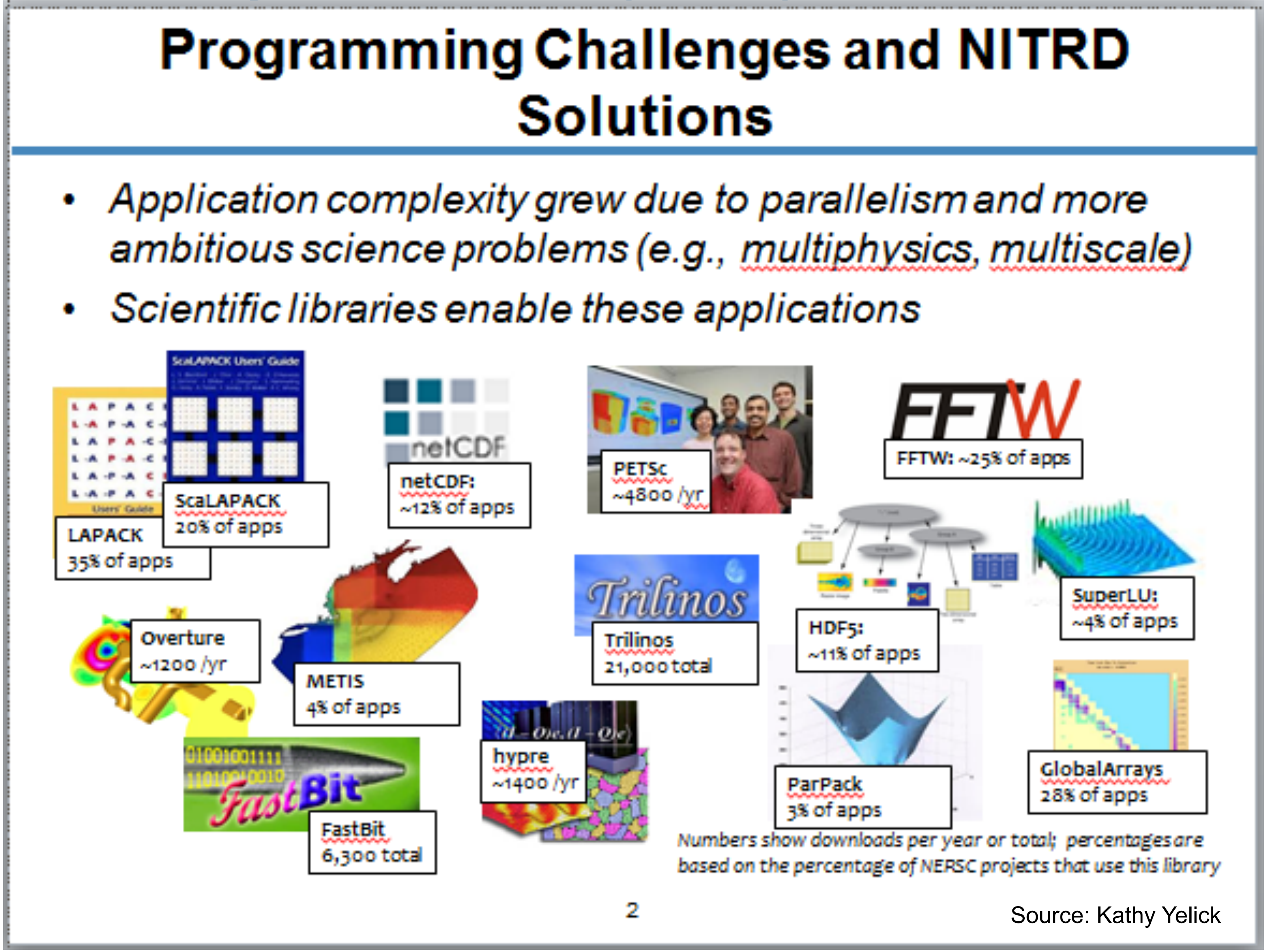


Third party names are the property of their owners.

Parallel programming is really hard

- Programming is hard whether you write serial or parallel code.
  - Parallel programming is just a new wrinkle added to the already tough problem of writing high quality, robust and efficient code.
- Why does Parallel programming seems so complex?
  - The literature overwhelms with hundreds of languages/APIs and a countless assortment of algorithms.
  - Experienced parallel programmers love to tell “war stories” of Herculean efforts to make applications scale … which can scare people away.
  - It’s new: synchronization, scalable algorithms, distributed data structures, concurrency bugs, memory models … hard or not it’s a bunch of new stuff to learn.

But it’s really not that bad (part 1): parallel libraries



Programming Challenges and NITRD Solutions

- *Application complexity grew due to parallelism and more ambitious science problems (e.g., multiphysics, multiscale)*
- *Scientific libraries enable these applications*

LAPACK
35% of apps

ScaLAPACK
20% of apps

netCDF:
~12% of apps

PETSc
~4800 /yr

FFTW: ~25% of apps

Overture
~1200 /yr

METIS
4% of apps

Trilinos
21,000 total

HDF5:
~11% of apps

SuperLU:
~4% of apps

FastBit
6,300 total

hypre
~1400 /yr

ParPack
3% of apps

GlobalArrays
28% of apps

Numbers show downloads per year or total; percentages are based on the percentage of NERSC projects that use this library

2

Source: Kathy Yelick

Third party names are the property of their owners.

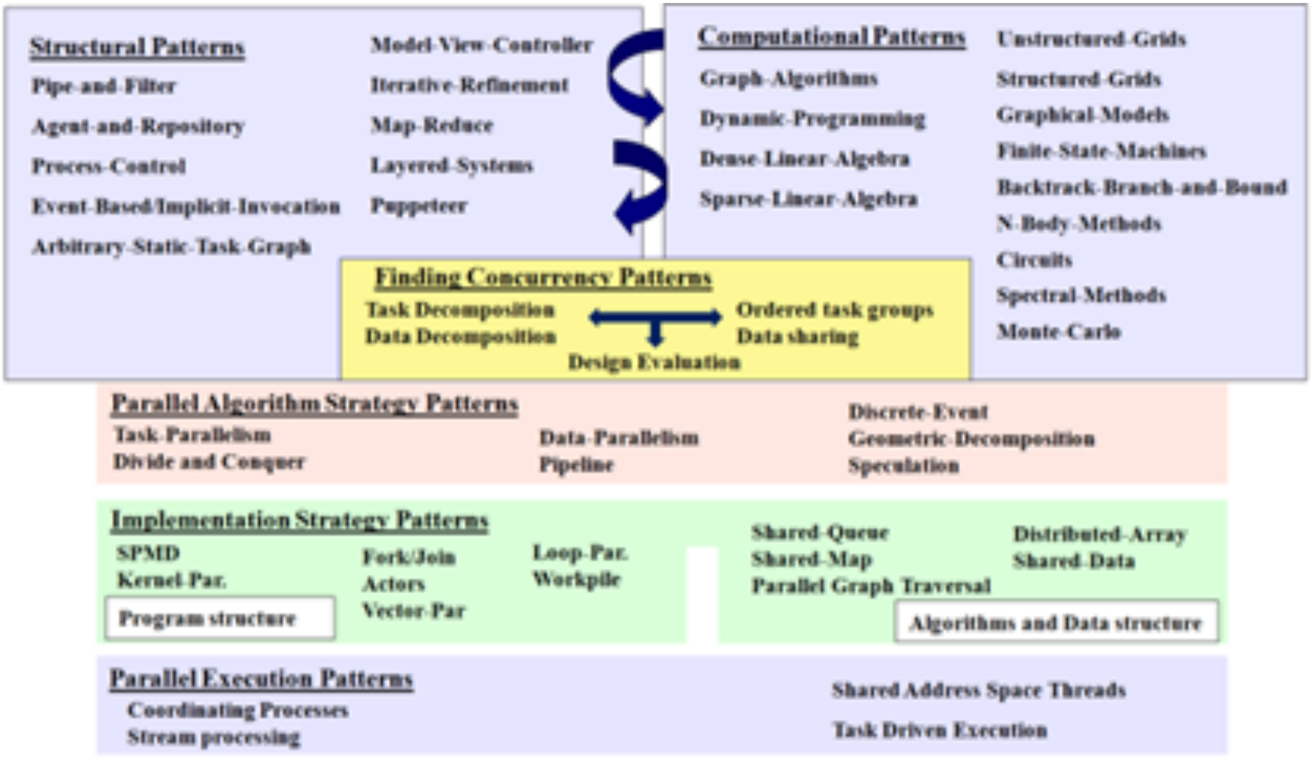
But its really not that bad: part 2

- Don’t let the glut of parallel programming languages confuse you.
- Leave research languages to C.S. researchers and stick to the small number of broadly used languages/APIs:
  - Industry standards:
    - Pthreads and OpenMP
    - MPI
    - OpenCL
    - TBB (… and maybe Cilk?)
  - or a broadly deployed solutions tied to your platform of choice
    - CUDA and OpenACC (for NVIDIA platforms and PGI compilers)
    - .NET and C++ AMP (Microsoft)
  - For HPC programmers dreaming of Exascale … maybe a PGAS language/API?
    - UPC
    - GA

Third party names are the property of their owners.

But its really not that bad : part 3

- Most algorithms are based on a modest number of recurring patterns (see Kurt Kreutzer's lecture tomorrow).



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.
Workpile
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

- Almost every parallel program is written in terms of just 7 basic patterns:
  - SPMD
  - Kernel Parallelism
  - Fork/join
  - Actors
  - Vector Parallelism
  - Loop Parallelism
  - Work Pile

Parallel programming is easy

- So all you need to do is:
  - **Pick** your language.
    - I suggest sticking to industry standards and open source so you can move around between hardware platforms:
  - pthreads
  - OpenMP
  - OpenCL
  - MPI
  - TBB
  - **Learn** the key 7 patterns
    - SPMD
    - Kernel Parallelism
    - Fork/join
    - Actors
    - Vector Parallelism
    - Loop Parallelism
    - Work Pile
  - **Master** the few patterns common to your platform and application domain … for example, most application programmers just use these three patterns
  - SPMD
  - Kernel Parallelism
  - Loop Parallelism

Third party names are the property of their owners.

Comparing parallel programming languages/APIs

- To compare programming languages and APIs at a high level, we can think in terms of four key elements

Units of Execution	A distinct executable agent that carries out the work of a program. Examples include the threads managed by an OS, processes running on the node of a cluster, or work-items in an OpenCL program
Tasks/mapping	Tasks are a logically related set of operations used to organize the computations in a program. A key aspect of a parallel program is how these tasks are associated (or mapped) onto the units of execution.
Coordination	Mechanisms to manage units of execution (e.g. create, destroy, suspend) and how they interact (e.g. synchronization and communication).
Hardware targets	Most programming models were designed with a particular class of parallel hardware in mind.

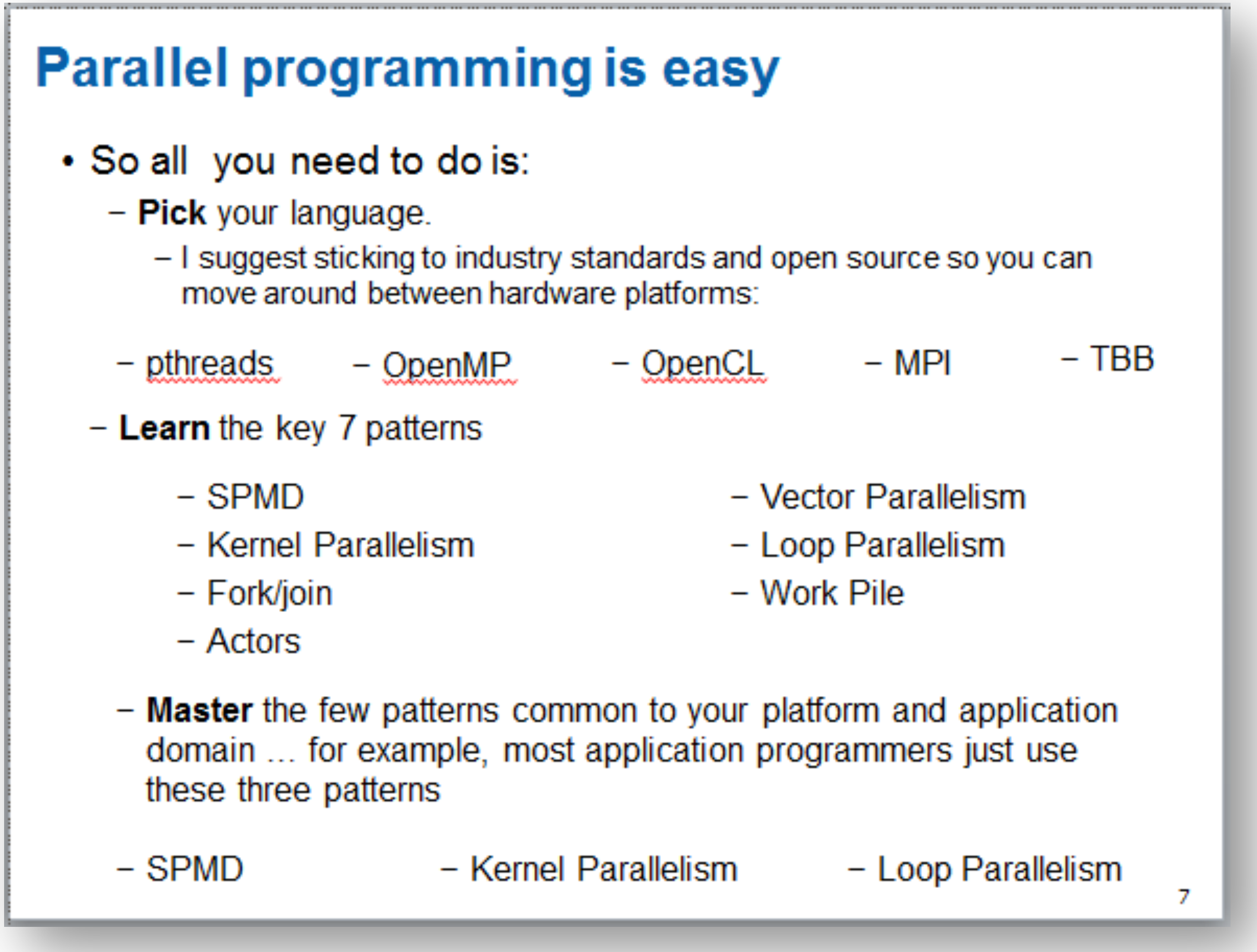
Comparing parallel programming languages/APIs

	Units of execution	**Tasks/mapping**	**Coordination**	**Hardware targets**
Pthreads	threads	Fork join	Shared variables and **explicit** synchronization constructs	Shared address space computers
OpenMP	threads	Teams of threads with worksharing (loops and tasks)	Shared variables and synchronization constructs	Shared address space computers
MPI	processes	SPMD*	Message passing	Any MIMD* computer
OpenCL	Work-items	Kernel parallelism*		Heterogeneous computers*
CUDA	CUDA-threads	Kernel parallelism*		NVIDIA GPUs

* MIMD (multiple instruction multiple data) and heterogeneous computers will be covered in a latter lecture on parallel hardware. The SPMD (single Program Multiple Data) and kernel parallelism patterns will be covered in our parallel design patterns lecture.

If you become overwhelmed during this course …

- Come back to this slide and remind yourself … things are not as bad as they seem



Parallel programming is easy

- So all you need to do is:
  - **Pick** your language.
    - I suggest sticking to industry standards and open source so you can move around between hardware platforms:
  - pthreads
  - OpenMP
  - OpenCL
  - MPI
  - TBB
  - **Learn** the key 7 patterns
    - SPMD
    - Kernel Parallelism
    - Fork/join
    - Actors
    - Vector Parallelism
    - Loop Parallelism
    - Work Pile
  - **Master** the few patterns common to your platform and application domain … for example, most application programmers just use these three patterns
  - SPMD
  - Kernel Parallelism
  - Loop Parallelism

7