# Supercomputing in Plain English

## Part VII: Multicore Madness


**Henry Neeman, Director**

**OU Supercomputing Center for Education & Research**

**University of Oklahoma**


**Wednesday October 17 2007**

# This is an experiment!

It’s the nature of these kinds of videoconferences that **failures are guaranteed to happen!**

**NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

# Access Grid/VRVS

If you're connecting via the Access Grid or VRVS, the venue is:

NCSA Venue Monte Carlo

It's available Wed Oct 17 2007 1:00-4:30pm Central Time, but the workshop starts at 3:00pm Central Time.

Many thanks to John Chapman of U Arkansas for setting this up for us.

# iLinc

- We only have about 40-45 simultaneous iLinc connections available.
- Therefore, each institution has at most one iLinc person designated.
- If you're the iLinc person for your institution, you've already gotten e-mail about it, so please follow the instructions.
- If you aren't your institution's iLinc person, then you can't become it, because we're completely out of iLinc connections.
- Many thanks to Katherine Kantardjieff of California State U Fullerton for setting this up for us.

# QuickTime Broadcast

If you don’t have iLinc, you can connect via QuickTime:

`rtsp://129.15.254.141/neeman_02.sdp`

We strongly recommend using QuickTime player, since we’ve seen it work.

When you run it, traverse the menus

File -> Open URL

Then paste in the rstp URL the Movie URL space, and click OK.

Many thanks to Kevin Blake of OU for setting this up.

# Phone Bridge

If all else fails, you can call into our phone bridge:

1-866-285-7778, access code 6483137#

Please mute yourself and use the phone to listen.

Don't worry, I'll call out slide numbers as we go.

To ask questions, please use Google Talk or Gmail.

Many thanks to Amy Apon of U Arkansas for setting this up for us, and to U Arkansas for absorbing the costs.

# Google Talk

To ask questions, please use our Google Talk group chat session (text only).

You need to have (or create) a gmail.com account to use Google Talk.

Once you've logged in to your gmail.com account, go to:

http://www.google.com/talk/

and then contact the user named:

oscer.sipe

Alternatively, you can send your questions by e-mail to oscer.sipe@gmail.com.

# This is an experiment!

**REMINDER**:

It's the nature of these kinds of videoconferences that **failures are guaranteed to happen!**

**NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

# Outline

- The March of Progress
- Multicore/Many-core Basics
- Software Strategies for Multicore/Many-core
- A Concrete Example: Weather Forecasting

# The March of Progress

# OU’s TeraFLOP Cluster, 2002

10 racks @ 1000 lbs per rack

270 Pentium4 Xeon CPUs,
 2.0 GHz, 512 KB L2 cache

270 GB RAM, 400 MHz FSB

8 TB disk

Myrinet2000 Interconnect

100 Mbps Ethernet Interconnect

OS: Red Hat Linux

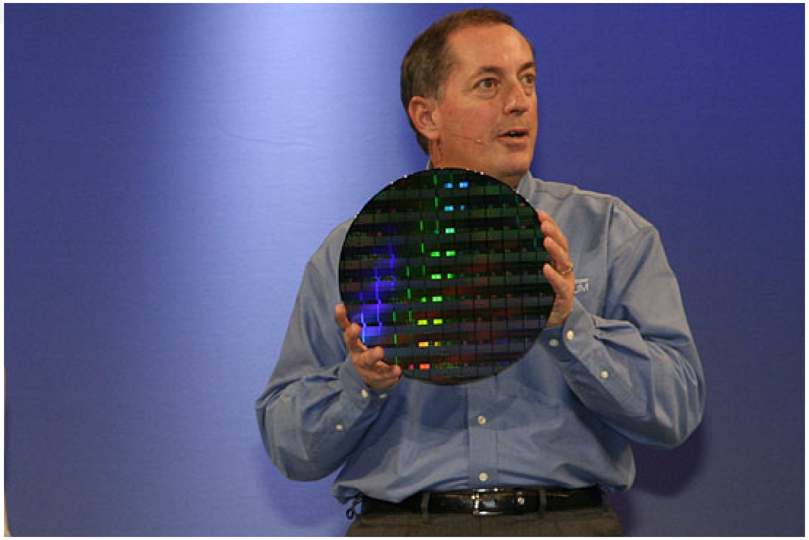
Peak speed: 1.08 TFLOP/s
 (1.08 trillion calculations per second)

One of the first Pentium4 clusters!

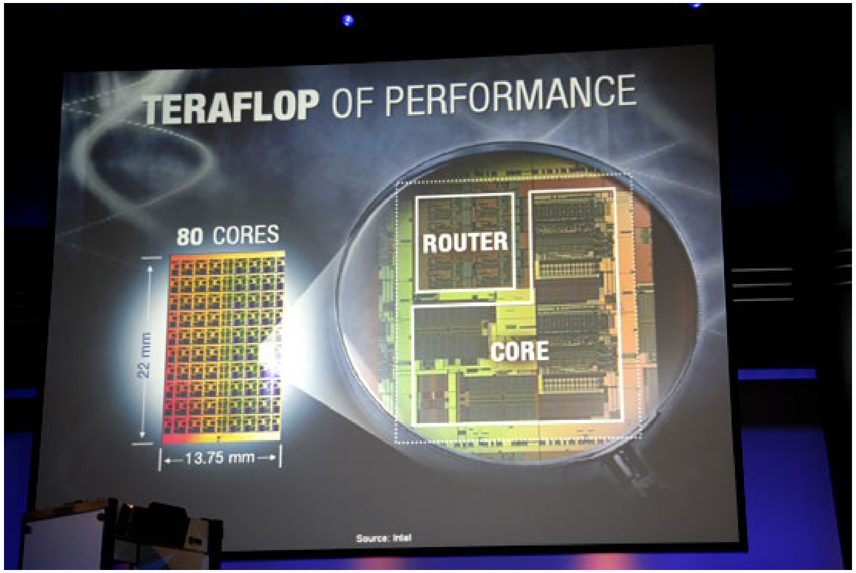
Aspen Systems Inc.


**`boomer.oscer.ou.edu`**

# TeraFLOP, Prototype 2006, Sale 2011



**9 years from room to chip!**




http://news.com/2300-1006_3-6119652.html

# Moore's Law

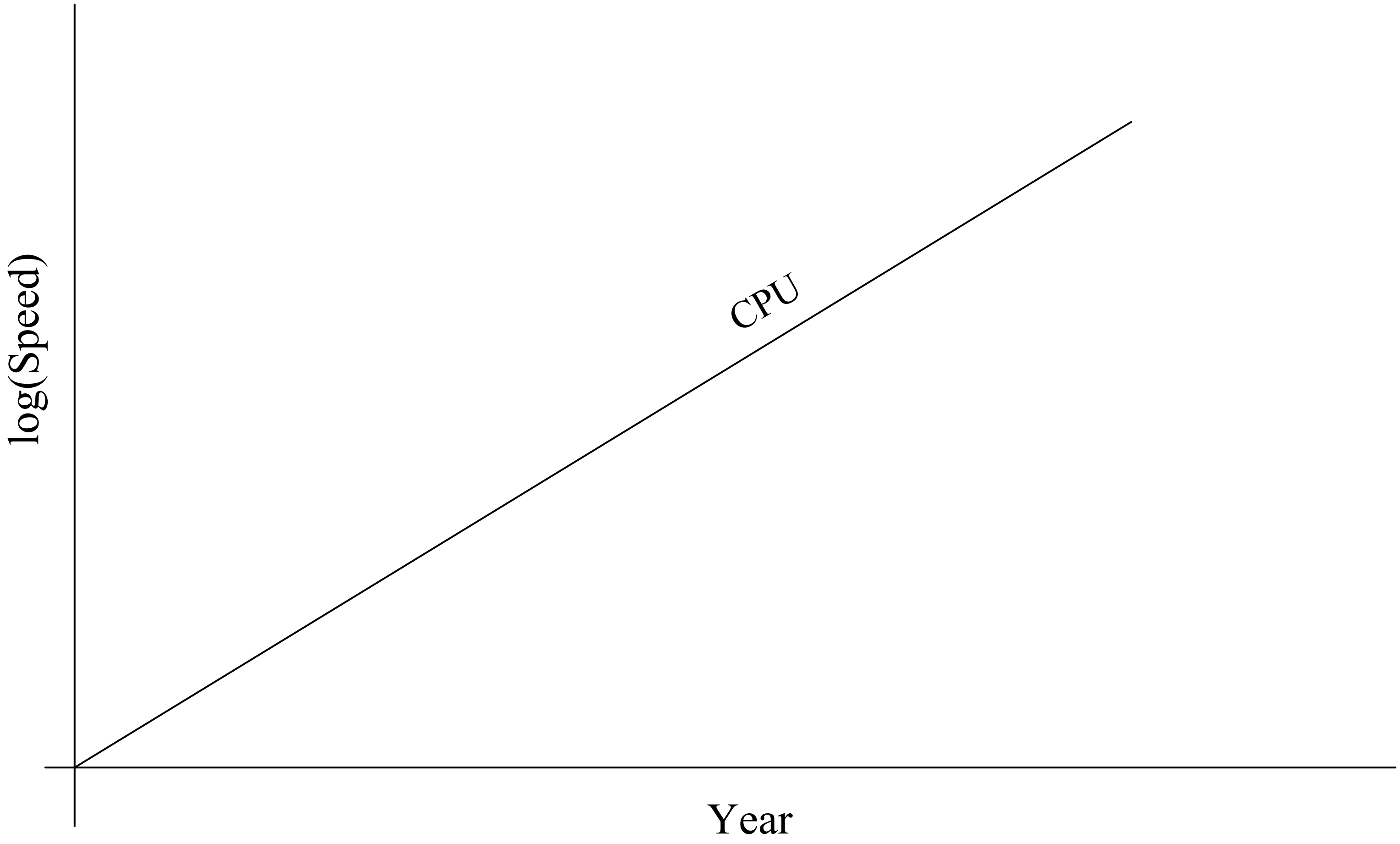
In 1965, Gordon Moore was an engineer at Fairchild Semiconductor.

He noticed that the number of transistors that could be squeezed onto a chip was doubling about every 18 months.

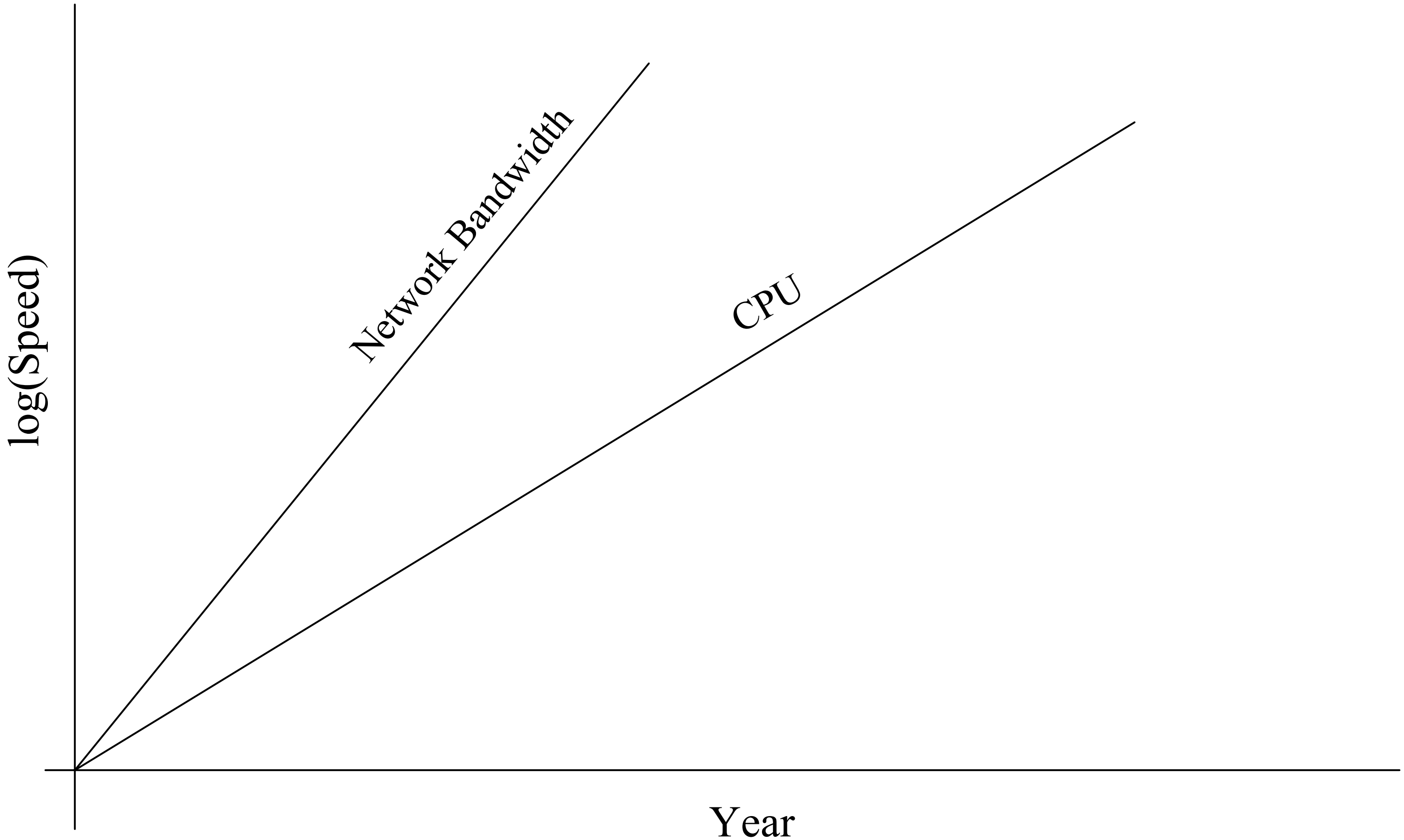
It turns out that computer speed is roughly proportional to the number of transistors per unit area.

Moore wrote a paper about this concept, which became known as ***"Moore's Law."***

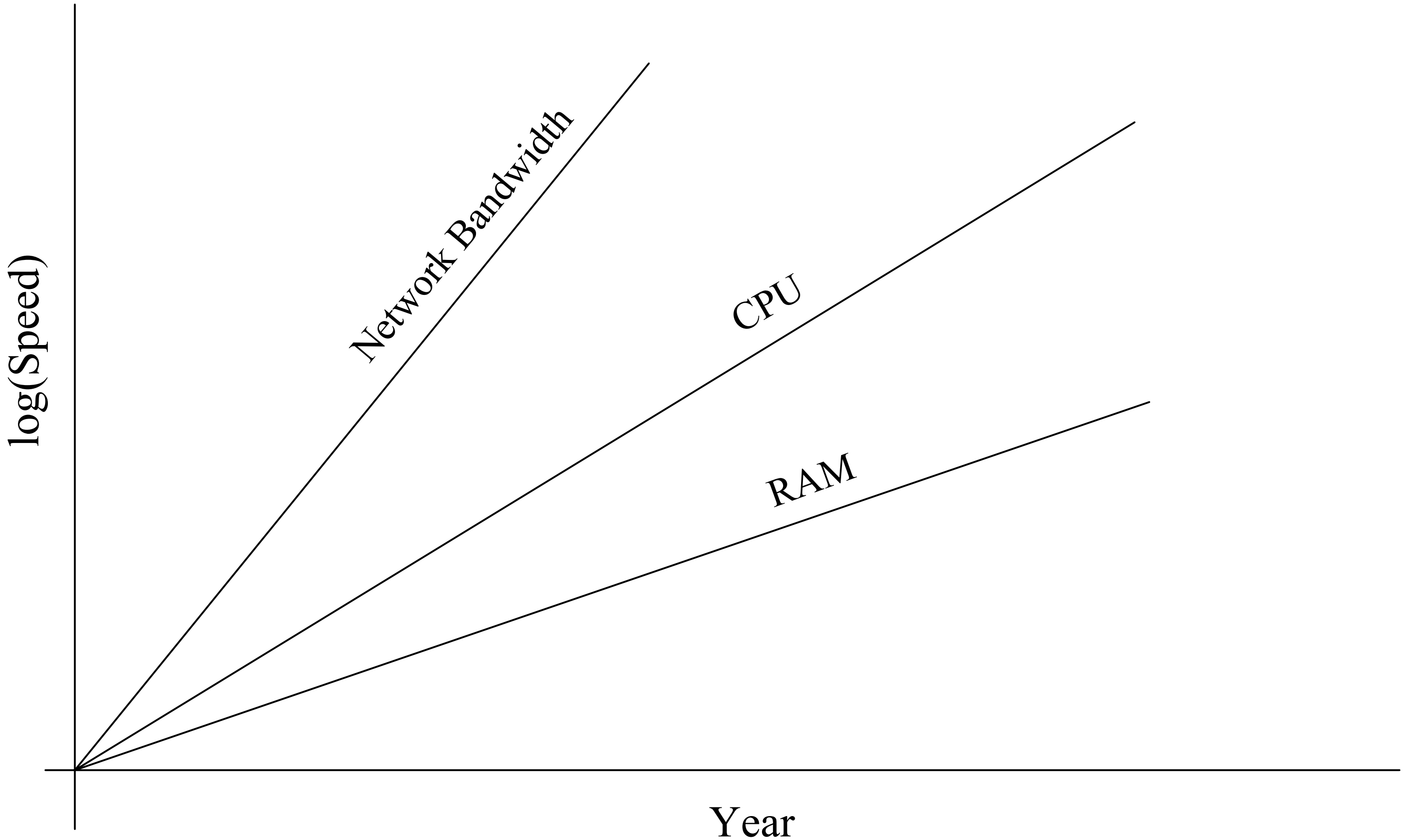
# Moore’s Law in Practice



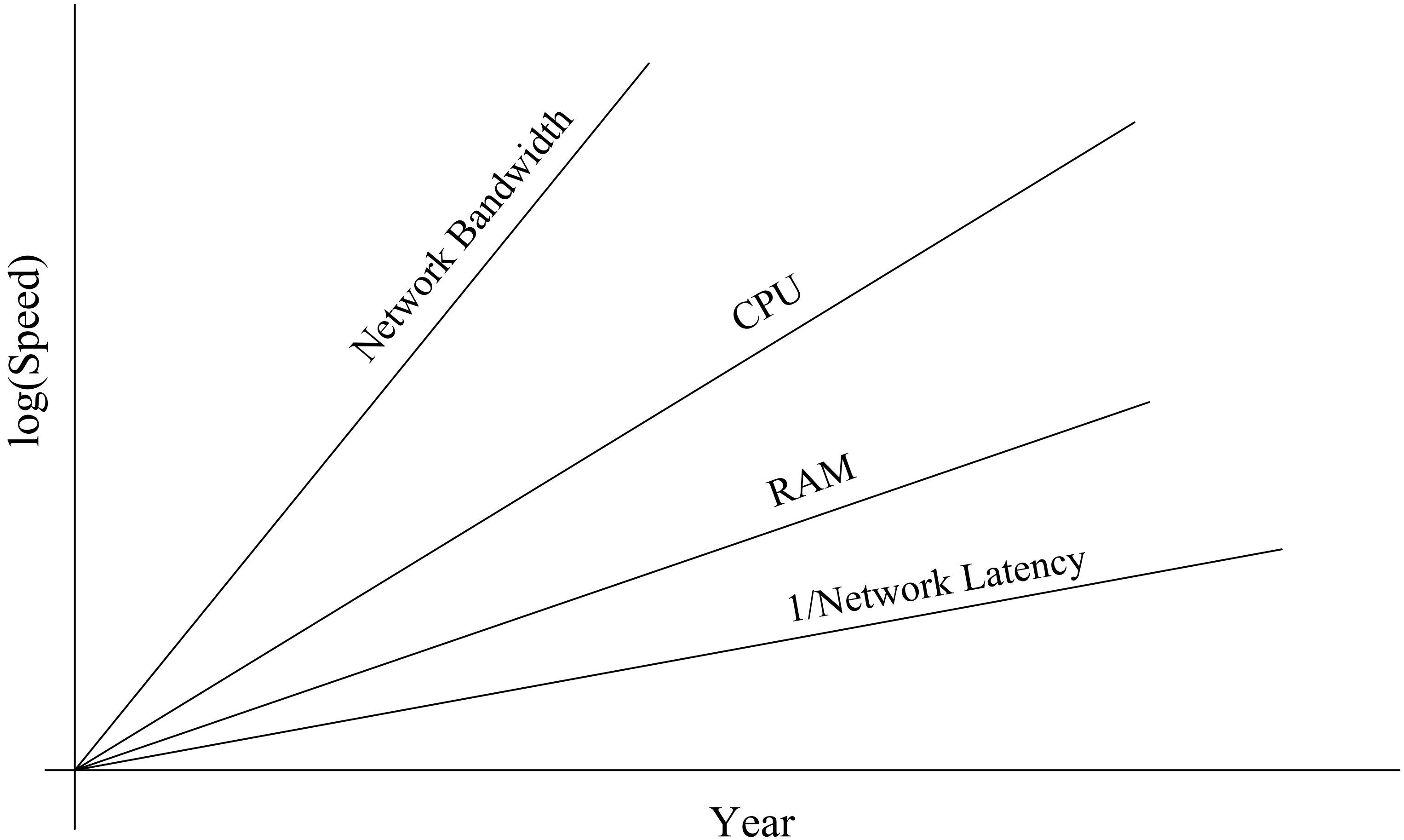
# Moore's Law in Practice



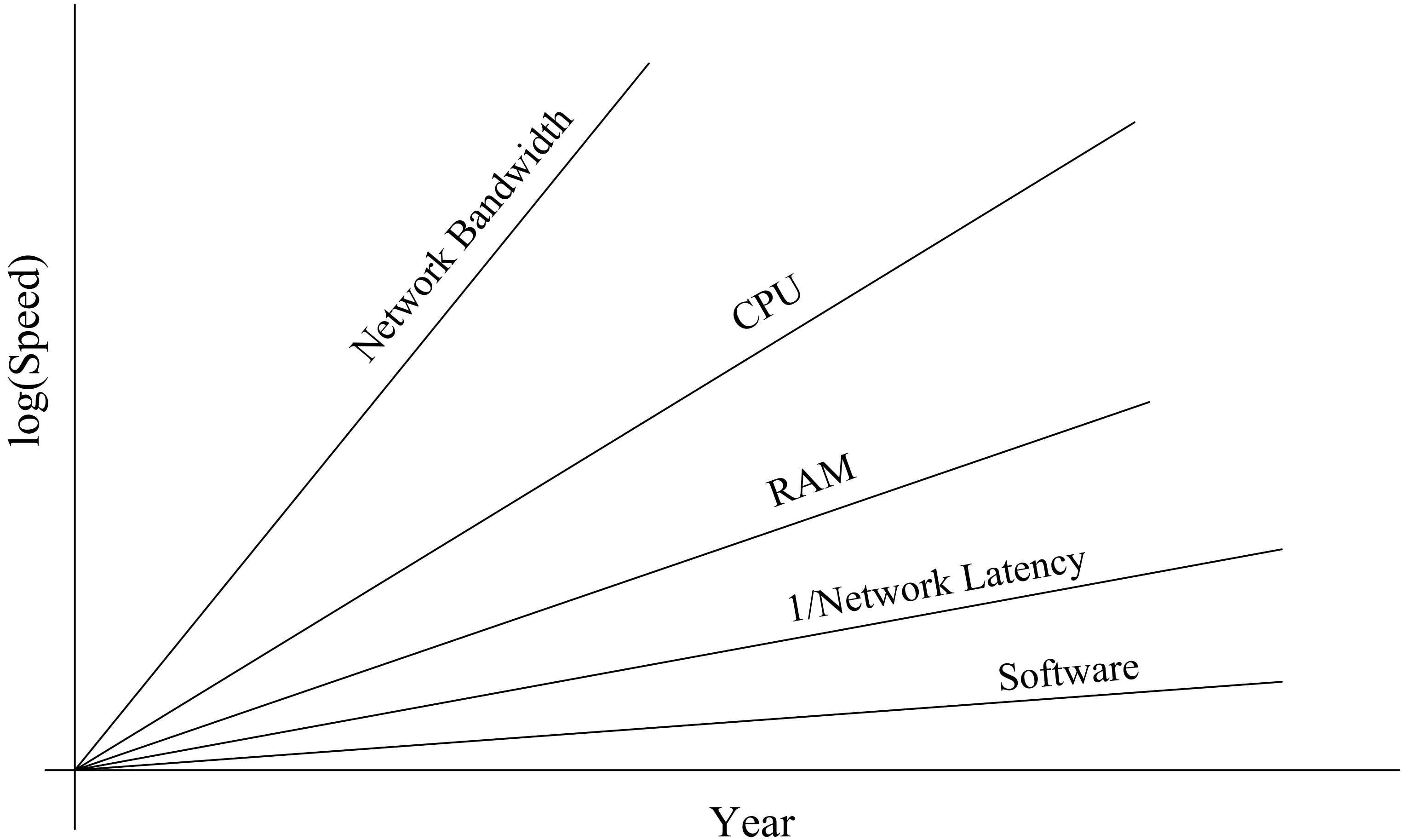
# Moore’s Law in Practice



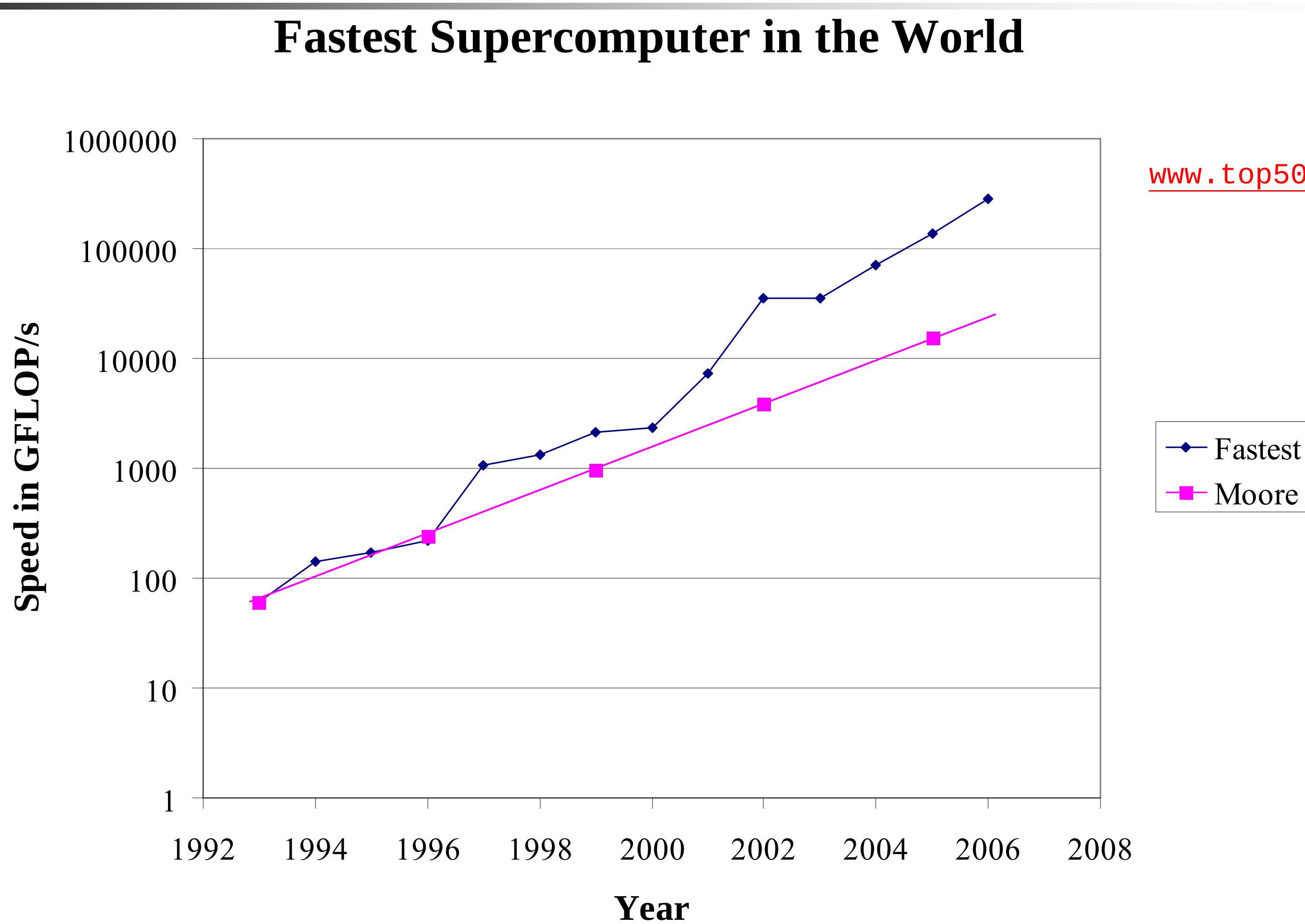
# Moore's Law in Practice



# Moore’s Law in Practice



# Fastest Supercomputer vs. Moore




www.top500.org

# The Tyranny of the Storage Hierarchy

# Henry's Laptop

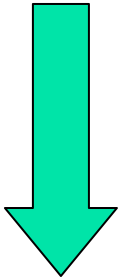
**Dell Latitude D620[4]**

- Pentium 4 Core Duo T2400 1.83 GHz w/2 MB L2 Cache
- 2 GB (2048 MB) 667 MHz DDR2 SDRAM
- 100 GB 7200 RPM SATA Hard Drive
- DVD±RW/CD-RW Drive (8x)
- 1 Gbps Ethernet Adapter
- 56 Kbps Phone Modem

# The Storage Hierarchy

[5]

**Fast, expensive, few**



**Slow, cheap, a lot**

- Registers
- Cache memory
- Main memory (RAM)
- Hard disk
- Removable media (e.g., DVD)
- Internet

[6]

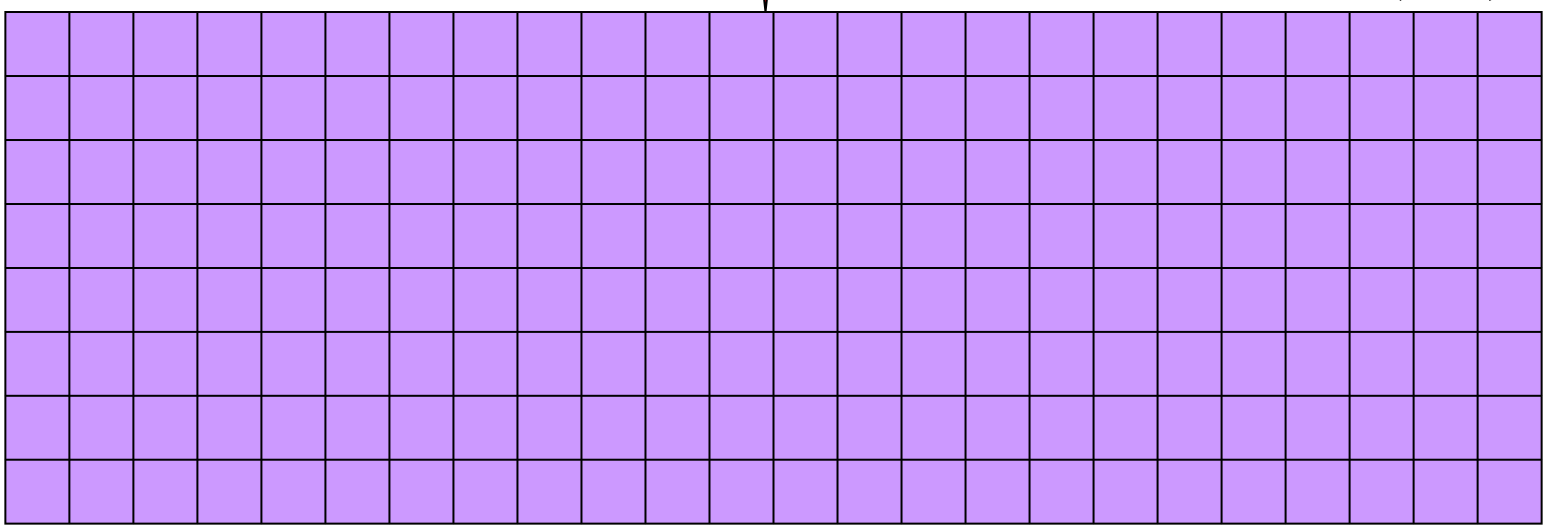
# RAM is Slow

The speed of data transfer between Main Memory and the CPU is much slower than the speed of calculating, so the CPU spends most of its time waiting for data to come in or go out.

**CPU** 351 GB/sec[7]

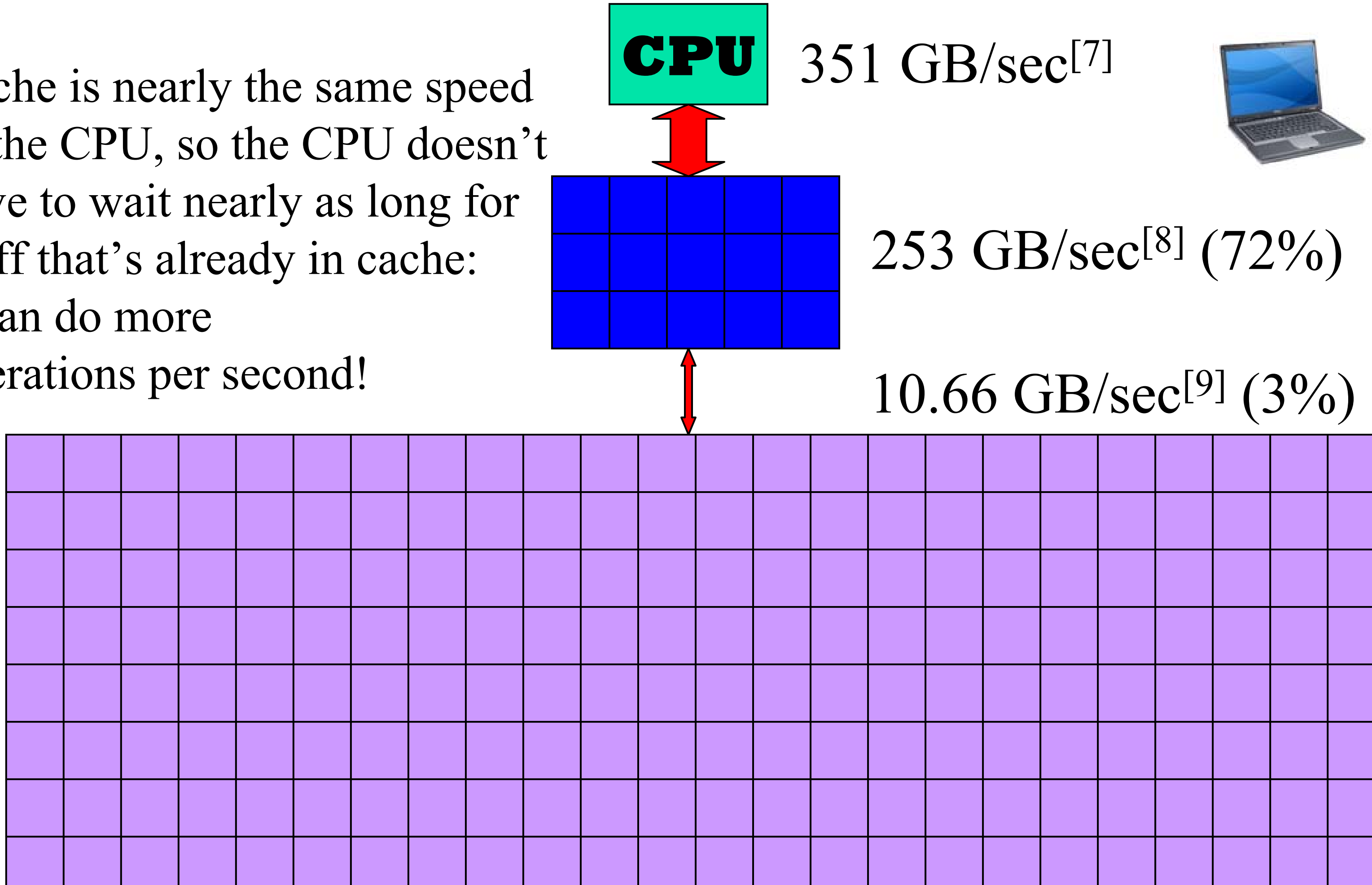
***Bottleneck***

10.66 GB/sec[9] (3%)



# Why Have Cache?

Cache is nearly the same speed as the CPU, so the CPU doesn't have to wait nearly as long for stuff that's already in cache: it can do more operations per second!



# Henry's Laptop, Again

**Dell Latitude D620[4]**

- Pentium 4 Core Duo T2400 1.83 GHz w/2 MB L2 Cache
- 2 GB (2048 MB) 667 MHz DDR2 SDRAM
- 100 GB 7200 RPM SATA Hard Drive
- DVD±RW/CD-RW Drive (8x)
- 1 Gbps Ethernet Adapter
- 56 Kbps Phone Modem

# Storage Speed, Size, Cost

| **Henry's Laptop** | Registers (Pentium 4 Core Duo 1.83 GHz) | Cache Memory (L2) | Main Memory (667 MHz DDR2 SDRAM) | Hard Drive (SATA 7200 RPM) | Ethernet (1000 Mbps) | DVD±RW (8x) | Phone Modem (56 Kbps) |
|---|---|---|---|---|---|---|---|
| Speed (MB/sec) [peak] | 359,792[7] (14,640 MFLOP/s*) | 258,785 [8] | 10,920 [9] | 100 [10] | 125 | 10.8 [11] | 0.007 |
| Size (MB) | 304 bytes** [12] | 2 | 2048 | 100,000 | unlimited | unlimited | unlimited |
| Cost ($/MB) | – | $21 [13] | $0.12 [13] | $0.002 [13] | charged per month (typically) | $0.0003 [13] | charged per month (typically) |

* MFLOP/s: millions of floating point operations per second

** 8 32-bit integer registers, 8 80-bit floating point registers, 8 64-bit MMX integer registers, 8 128-bit floating point XMM registers

# Storage Use Strategies

- ***Register reuse***: Do a lot of work on the same data before working on new data.
- ***Cache reuse***: The program is much more efficient if all of the data and instructions fit in cache; if not, try to use what's in cache a lot before using anything that isn't in cache.
- ***Data locality***: Try to access data that are near each other in memory before data that are far.
- **I/O efficiency**: Do a bunch of I/O all at once rather than a little bit at a time; don't mix calculations and I/O.

# A Concrete Example

- OSCER's big cluster, topdawg, has Irwindale CPUs: single core, 3.2 GHz, 800 MHz Front Side Bus.
- The theoretical peak CPU speed is 6.4 GFLOPs (double precision) per CPU, and in practice we've gotten as high as 94% of that.
- So, in theory each CPU could consume 143 GB/sec.
- The theoretical peak RAM bandwidth is 6.4 GB/sec, but in practice we get about half that.
- So, any code that does less than 45 calculations **per byte** transferred between RAM and cache has speed limited by RAM bandwidth.

# Good Cache Reuse Example

# A Sample Application

**Matrix-Matrix Multiply**

Let A, B and C be matrices of sizes
$nr \times nc$, $nr \times nk$ and $nk \times nc$, respectively:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,nc} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,nc} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,nc} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{nr,1} & a_{nr,2} & a_{nr,3} & \cdots & a_{nr,nc} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,nk} \\ b_{2,1} & b_{2,2} & b_{2,3} & \cdots & b_{2,nk} \\ b_{3,1} & b_{3,2} & b_{3,3} & \cdots & b_{3,nk} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{nr,1} & b_{nr,2} & b_{nr,3} & \cdots & b_{nr,nk} \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & \cdots & c_{1,nc} \\ c_{2,1} & c_{2,2} & c_{2,3} & \cdots & c_{2,nc} \\ c_{3,1} & c_{3,2} & c_{3,3} & \cdots & c_{3,nc} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{nk,1} & c_{nk,2} & c_{nk,3} & \cdots & c_{nk,nc} \end{bmatrix}$$

The definition of $A = B \bullet C$ is

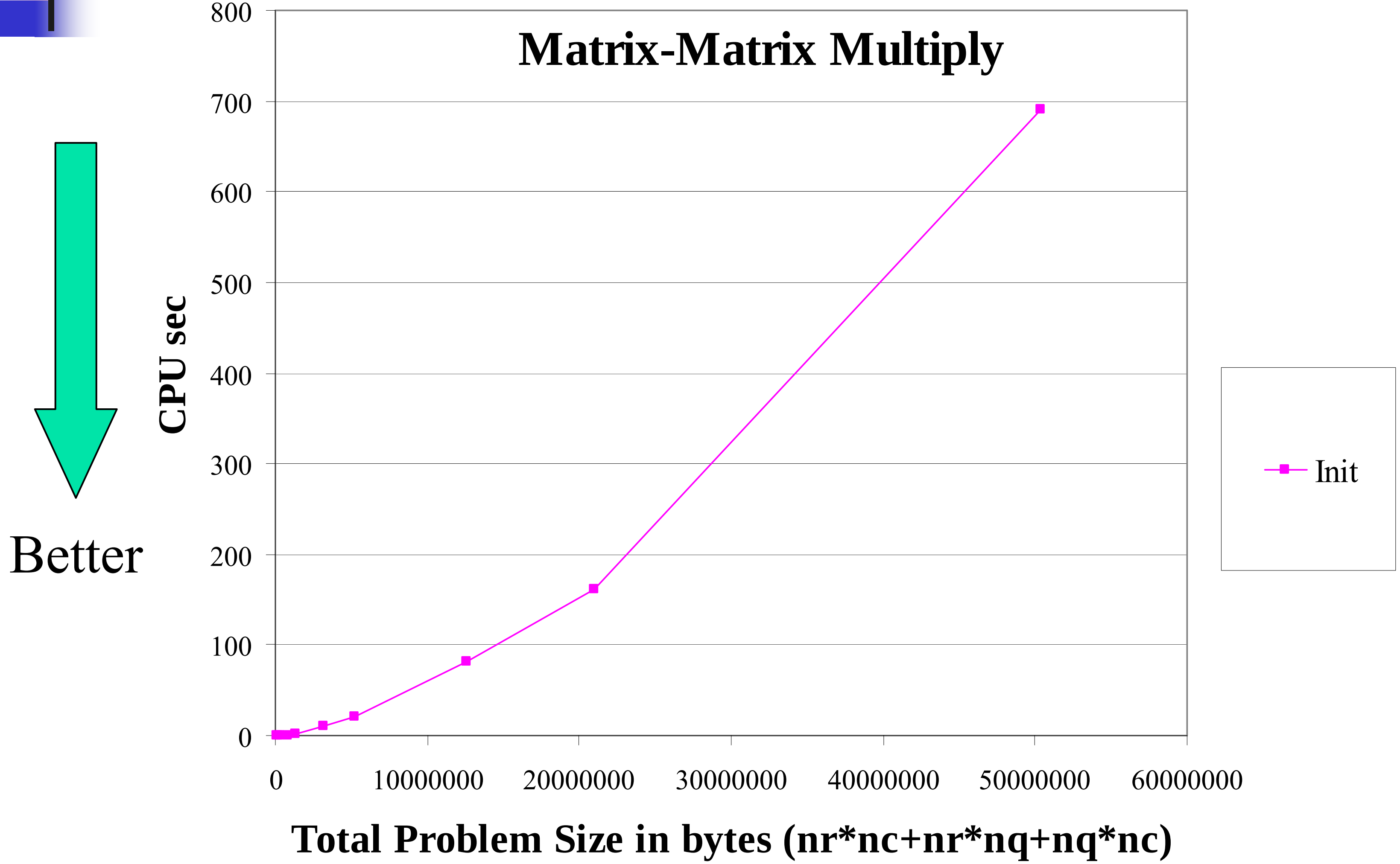
$$a_{r,c} = \sum_{k=1}^{nk} b_{r,k} \cdot c_{k,c} = b_{r,1} \cdot c_{1,c} + b_{r,2} \cdot c_{2,c} + b_{r,3} \cdot c_{3,c} + \ldots + b_{r,nk} \cdot c_{nk,c}$$

for $r \in \{1, nr\}$, $c \in \{1, nc\}$.

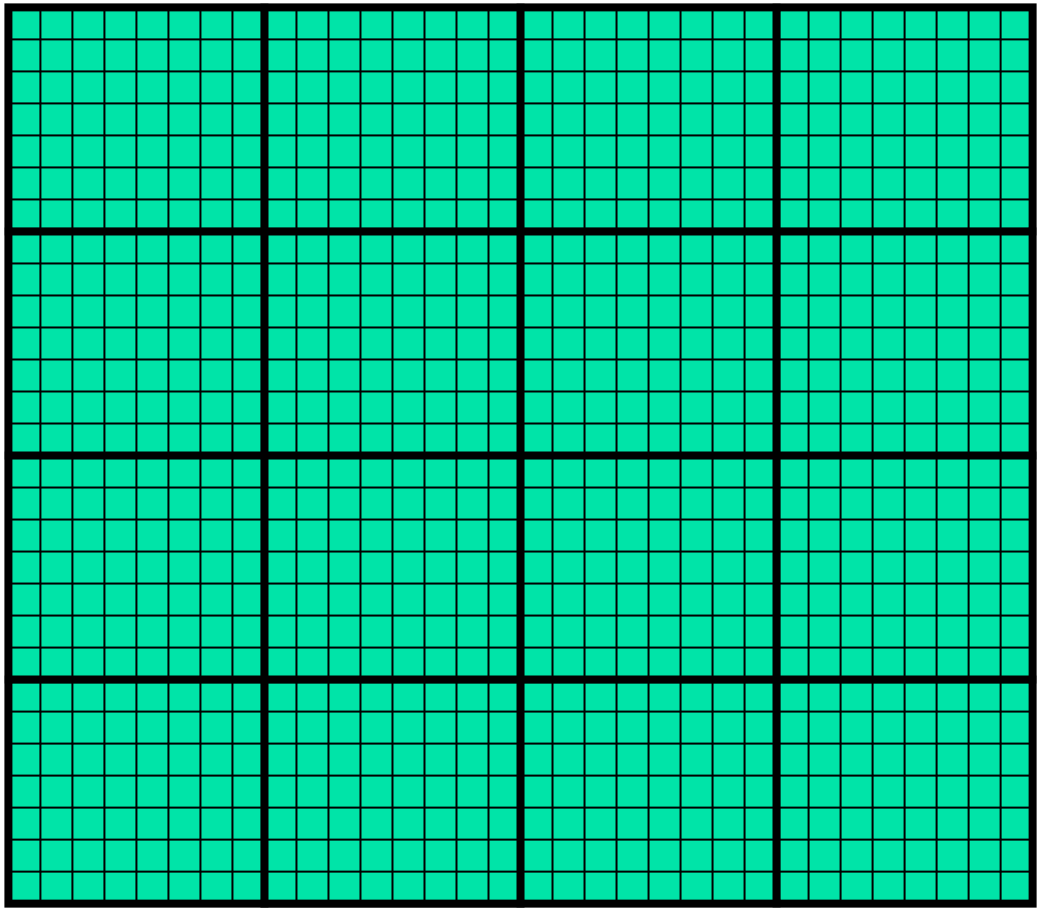
# Matrix Multiply: Naïve Version

```
SUBROUTINE matrix_matrix_mult_naive (dst, src1, src2, &
 &                                   nr, nc, nq)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: nr, nc, nq
  REAL,DIMENSION(nr,nc),INTENT(OUT) :: dst
  REAL,DIMENSION(nr,nq),INTENT(IN)  :: src1
  REAL,DIMENSION(nq,nc),INTENT(IN)  :: src2

  INTEGER :: r, c, q

  DO c = 1, nc
    DO r = 1, nr
      dst(r,c) = 0.0
      DO q = 1, nq
        dst(r,c) = dst(r,c) + src1(r,q) * src2(q,c)
      END DO
    END DO
  END DO
END SUBROUTINE matrix_matrix_mult_naive
```

# Performance of Matrix Multiply



# Tiling



# Tiling

- ***Tile***: A small rectangular subdomain of a problem domain. Sometimes called a ***block*** or a ***chunk***.
- ***Tiling***: Breaking the domain into tiles.
- **Tiling strategy**: Operate on each tile to completion, then move to the next tile.
- **Tile size** can be set at runtime, according to what's best for the machine that you're running on.

# Tiling Code

```
SUBROUTINE matrix_matrix_mult_by_tiling (dst, src1, src2, nr, nc, nq, &
 &           rtilesize, ctilesize, qtilesize)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: nr, nc, nq
  REAL,DIMENSION(nr,nc),INTENT(OUT) :: dst
  REAL,DIMENSION(nr,nq),INTENT(IN)  :: src1
  REAL,DIMENSION(nq,nc),INTENT(IN)  :: src2
  INTEGER,INTENT(IN) :: rtilesize, ctilesize, qtilesize

  INTEGER :: rstart, rend, cstart, cend, qstart, qend

  DO cstart = 1, nc, ctilesize
    cend = cstart + ctilesize - 1
    IF (cend > nc) cend = nc
    DO rstart = 1, nr, rtilesize
      rend = rstart + rtilesize - 1
      IF (rend > nr) rend = nr
      DO qstart = 1, nq, qtilesize
        qend = qstart + qtilesize - 1
        IF (qend > nq) qend = nq
        CALL matrix_matrix_mult_tile(dst, src1, src2, nr, nc, nq, &
 &                                   rstart, rend, cstart, cend, qstart, qend)
      END DO
    END DO
  END DO
END SUBROUTINE matrix_matrix_mult_by_tiling
```

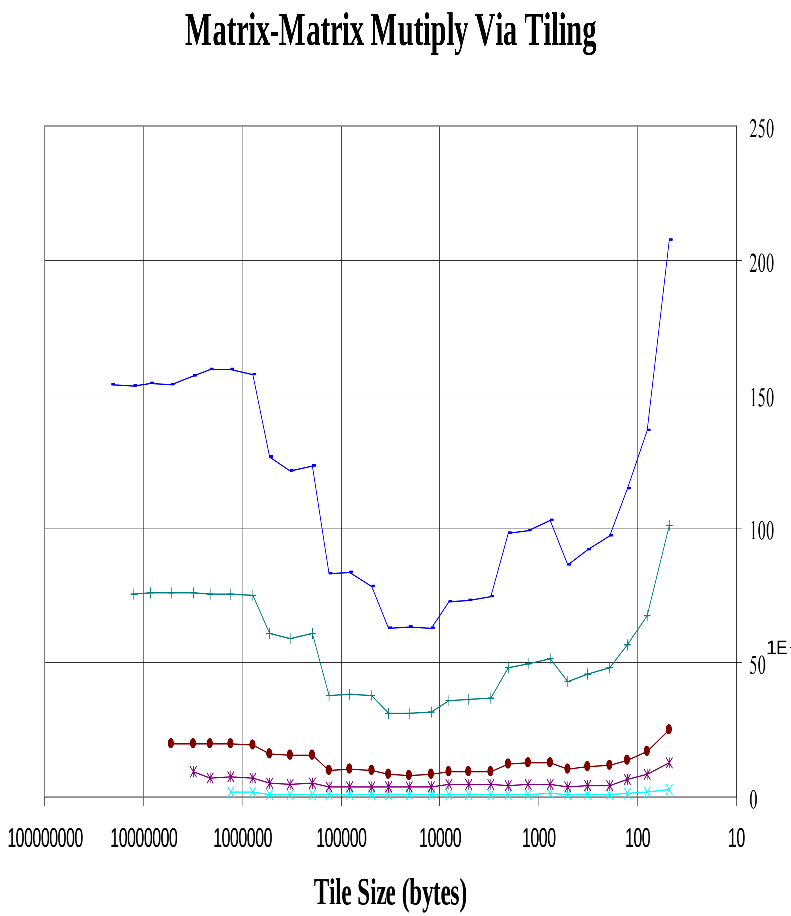
# Multiplying Within a Tile

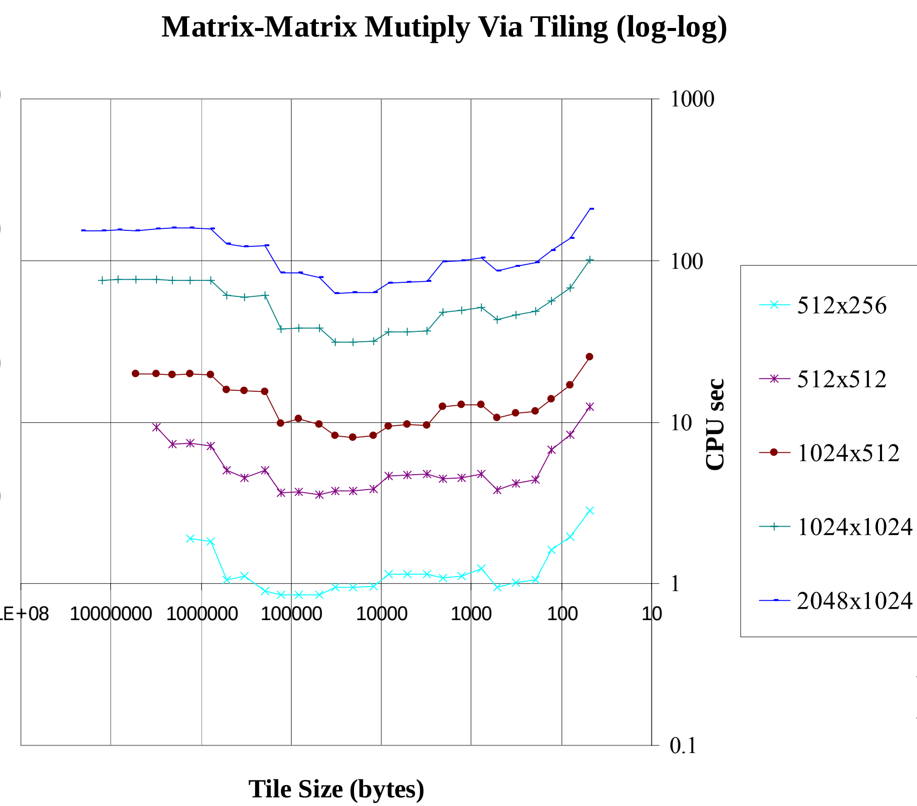
```
SUBROUTINE matrix_matrix_mult_tile (dst, src1, src2, nr, nc, nq, &
 &             rstart, rend, cstart, cend, qstart, qend)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: nr, nc, nq
  REAL,DIMENSION(nr,nc),INTENT(OUT) :: dst
  REAL,DIMENSION(nr,nq),INTENT(IN)  :: src1
  REAL,DIMENSION(nq,nc),INTENT(IN)  :: src2
  INTEGER,INTENT(IN) :: rstart, rend, cstart, cend, qstart, qend

  INTEGER :: r, c, q

  DO c = cstart, cend
    DO r = rstart, rend
      IF (qstart == 1) dst(r,c) = 0.0
      DO q = qstart, qend
        dst(r,c) = dst(r,c) + src1(r,q) * src2(q,c)
      END DO
    END DO
  END DO
END SUBROUTINE matrix_matrix_mult_tile
```

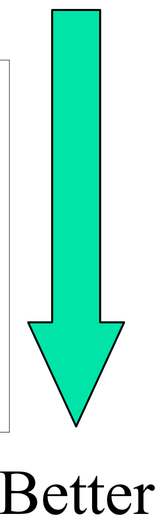
# Reminder: Naïve Version, Again

```
SUBROUTINE matrix_matrix_mult_naive (dst, src1, src2, &
 &                                   nr, nc, nq)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: nr, nc, nq
  REAL,DIMENSION(nr,nc),INTENT(OUT) :: dst
  REAL,DIMENSION(nr,nq),INTENT(IN)  :: src1
  REAL,DIMENSION(nq,nc),INTENT(IN)  :: src2

  INTEGER :: r, c, q

  DO c = 1, nc
    DO r = 1, nr
      dst(r,c) = 0.0
      DO q = 1, nq
        dst(r,c) = dst(r,c) + src1(r,q) * src2(q,c)
      END DO
    END DO
  END DO
END SUBROUTINE matrix_matrix_mult_naive
```

# Performance with Tiling



# The Advantages of Tiling

- It allows your code to **exploit data locality** better, to get much more cache reuse: your code runs faster!
- It's a relatively **modest amount of extra coding** (typically a few wrapper functions and some changes to loop bounds).
- **If you don't need** tiling – because of the hardware, the compiler or the problem size – then you can **turn it off by simply** setting the tile size equal to the problem size.

# Why Does Tiling Work Here?

Cache optimization works best when the number of calculations per byte is large.

For example, with matrix-matrix multiply on an $n \times n$ matrix, there are $\mathbf{O}(n^3)$ calculations (on the order of $n^3$), but only $\mathbf{O}(n^2)$ bytes of data.

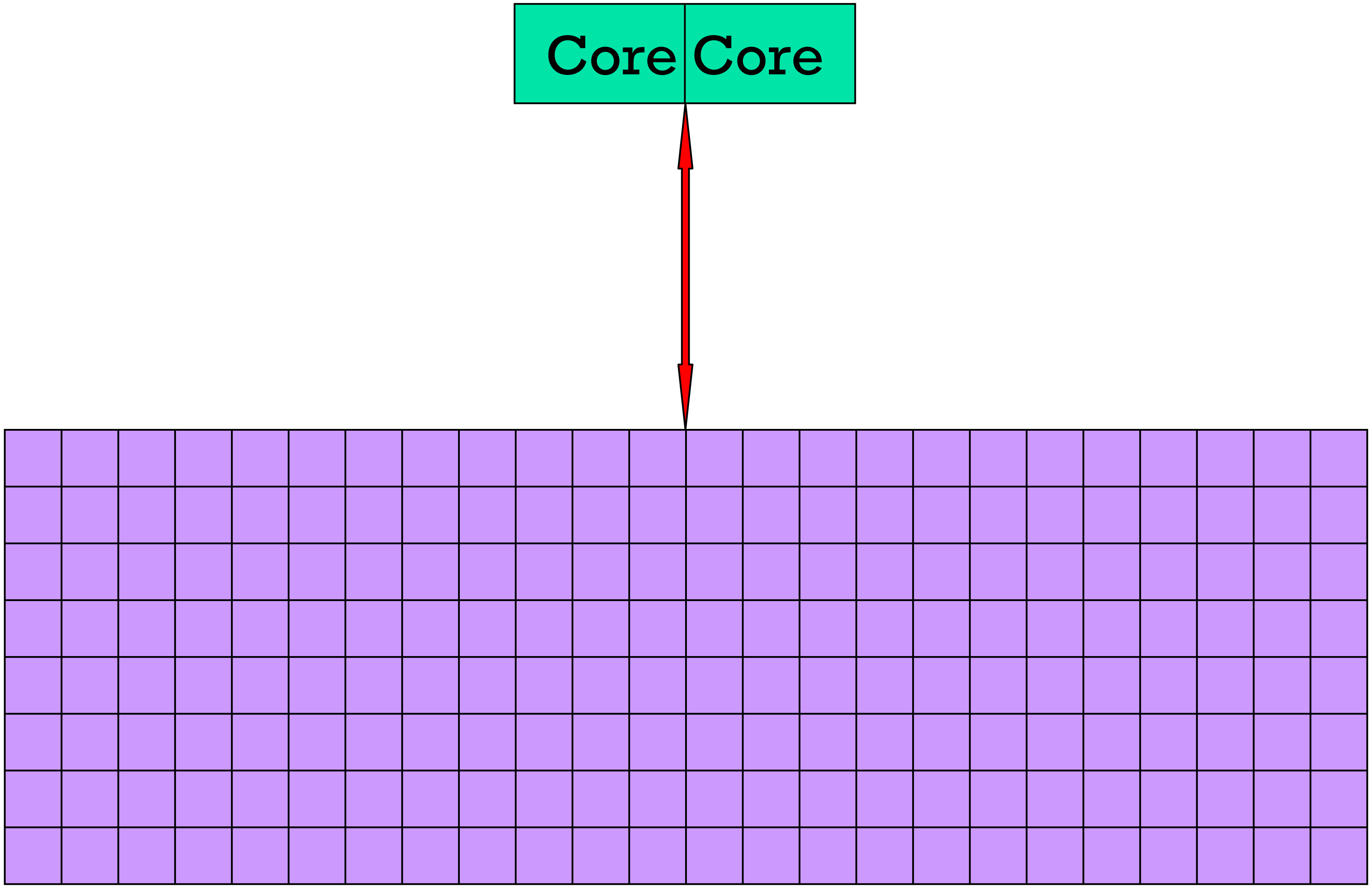
So, for large $n$, there are a huge number of calculations per byte transferred between RAM and cache.

# Multicore/Many-core Basics

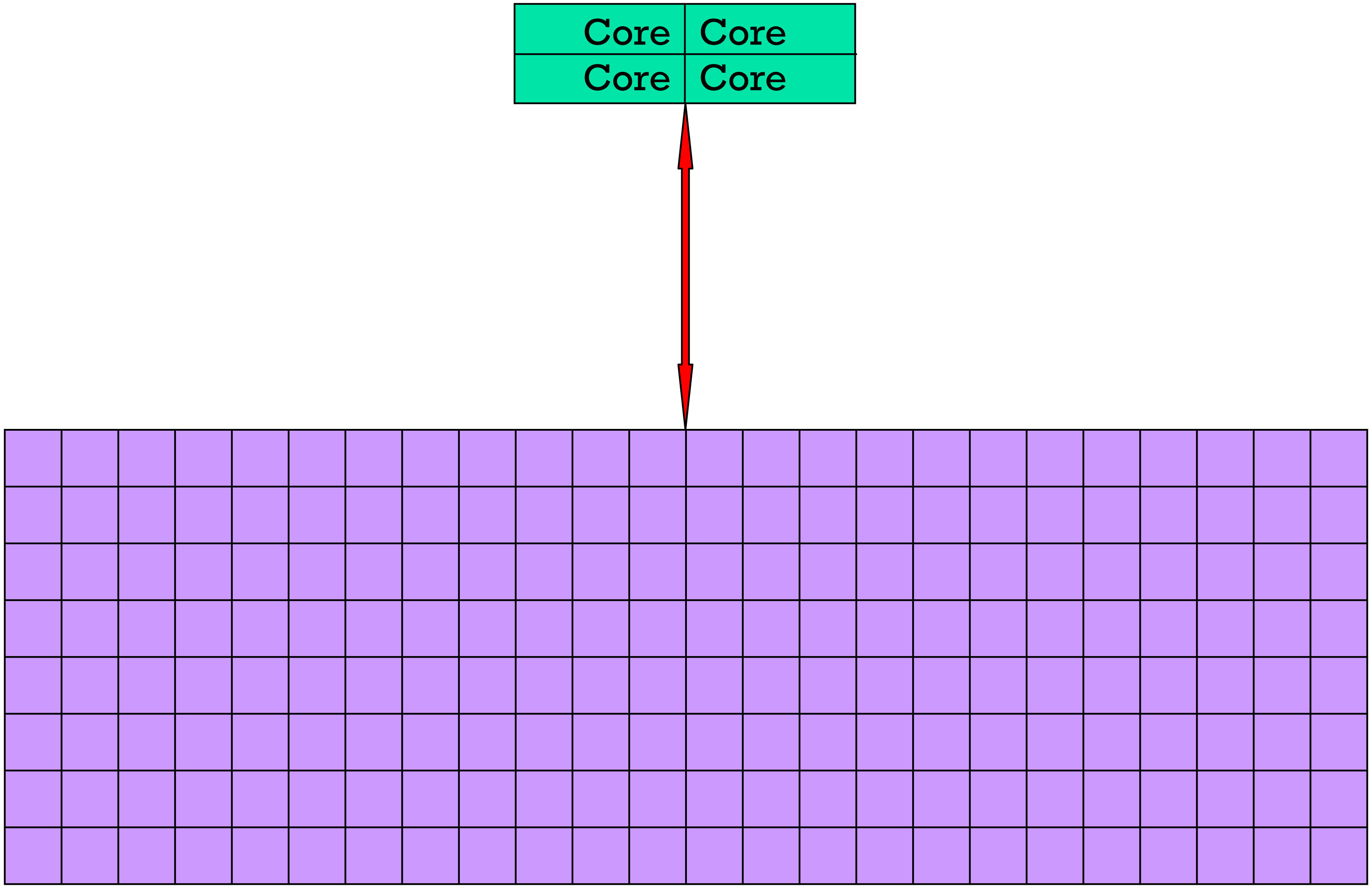
# What is Multicore?

- In the olden days (i.e., the first half of 2005), each CPU chip had one "brain" in it.
- More recently, each CPU chip has 2 ***cores*** (brains), and, starting in late 2006, 4 cores.
- **Jargon**: Each CPU chip plugs into a ***socket***, so these days, to avoid confusion, people refer to **sockets** and **cores**, rather than CPUs or processors.
- Each core is just like a full blown CPU, except that it shares its socket with one or more other cores – and therefore shares its bandwidth to RAM.

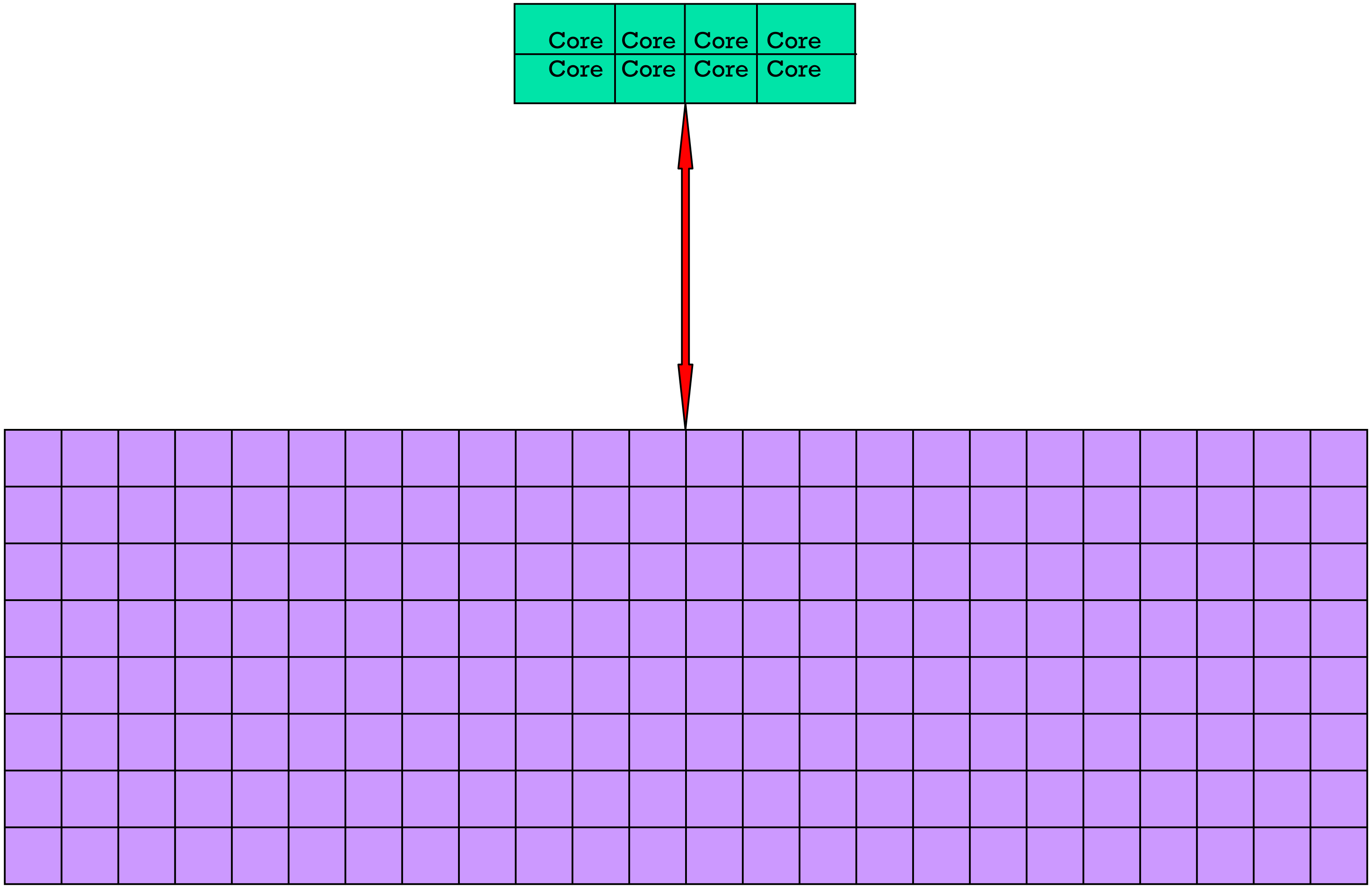
# Dual Core



# Quad Core



# Oct Core



# The Challenge of Multicore: RAM

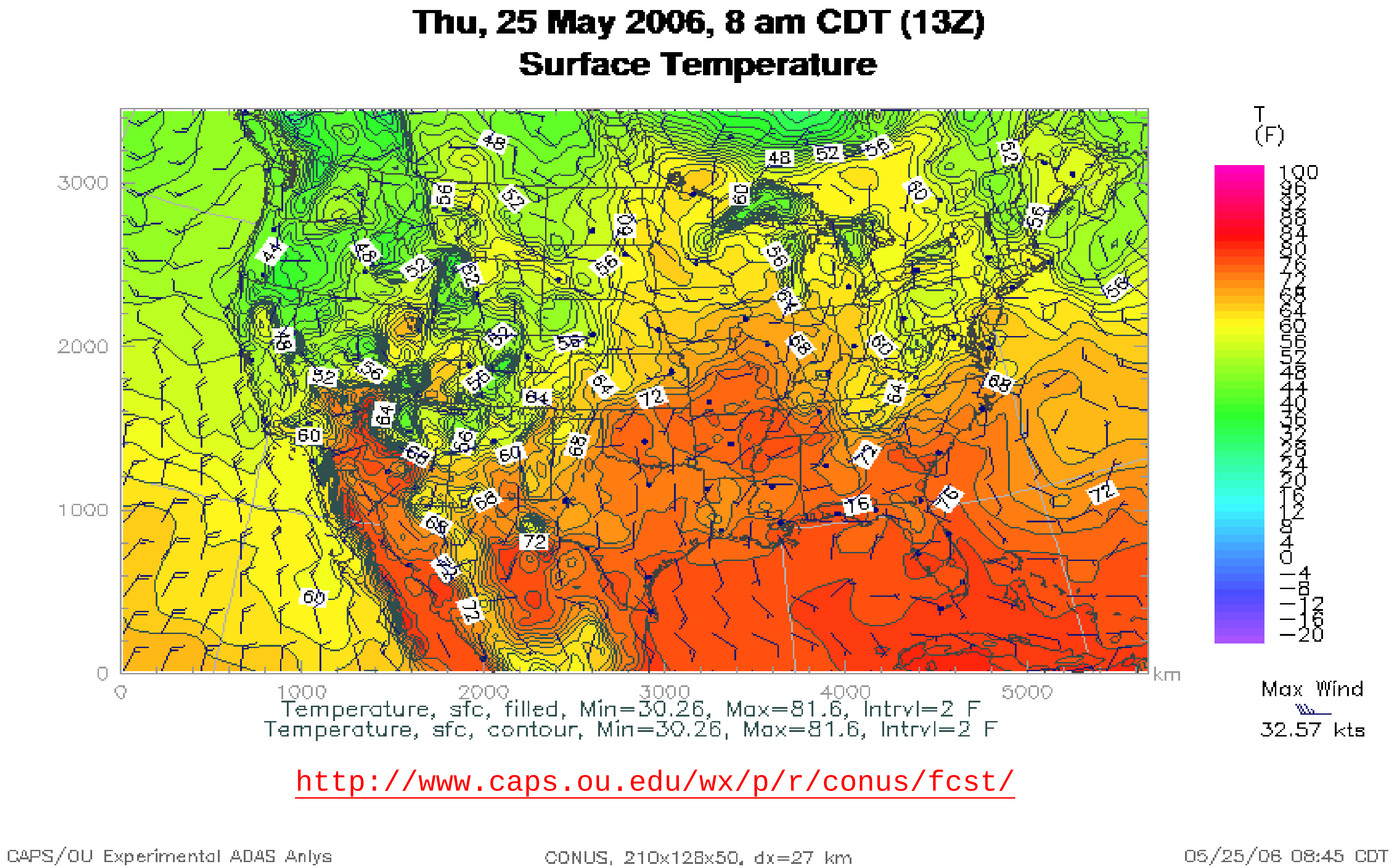
- Each socket has access to a certain amount of RAM, at a **fixed RAM bandwidth per SOCKET**.
- As the number of cores per socket increases, the **contention for RAM bandwidth increases** too.
- At 2 cores in a socket, this problem isn't too bad. But at 16 or 32 or 80 cores, it's **a huge problem**.
- So, applications that **are cache optimized** will get **big speedups**.
- But, applications whose performance is **limited by RAM bandwidth** are going to speed up only as fast as RAM bandwidth speeds up.
- RAM bandwidth **speeds up much slower** than CPU speeds up.

# The Challenge of Multicore: Network

- Each node has access to a certain number of network ports, at a **fixed number of network ports per NODE**.
- As the number of cores per node increases, the **contention for network ports increases** too.
- At 2 cores in a socket, this problem isn't too bad. But at 16 or 32 or 80 cores, it's **a huge problem**.
- So, applications that **do minimal communication** will get **big speedups**.
- But, applications whose performance is **limited by the number of MPI messages** are going to speed up very very little – and may even crash the node.

# A Concrete Example: Weather Forecasting

# Weather Forecasting

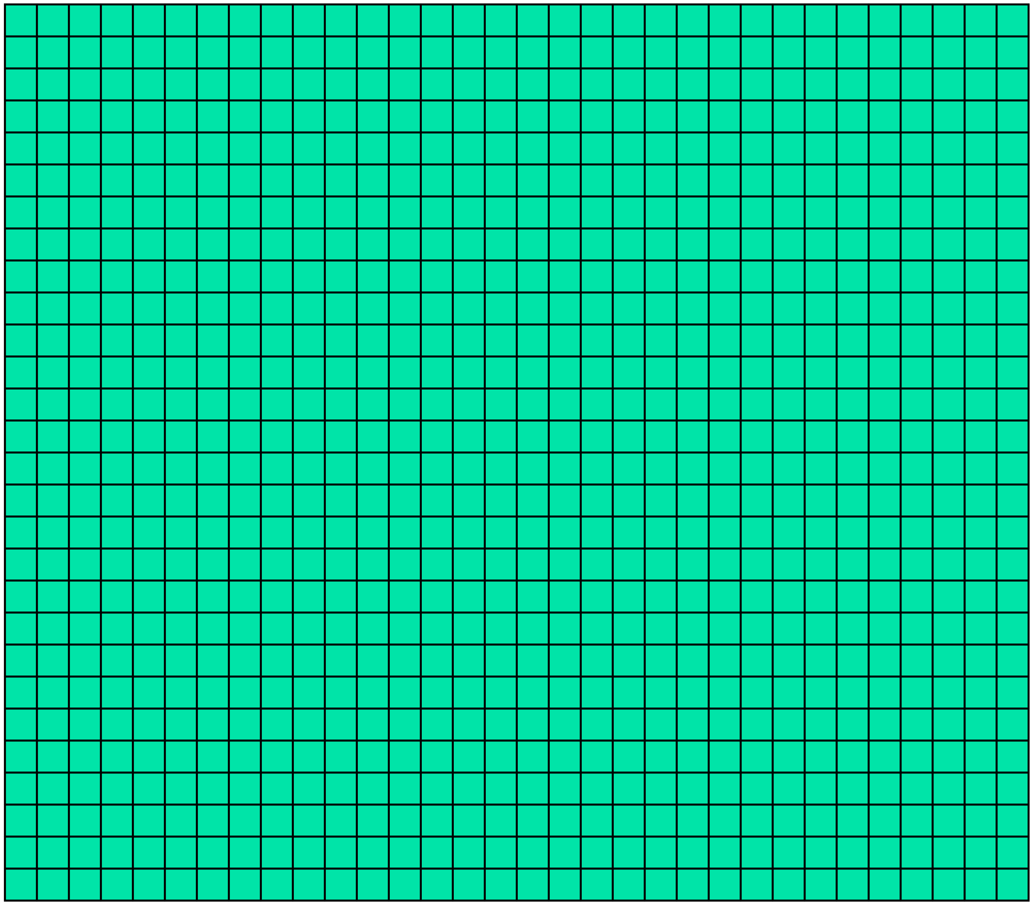



http://www.caps.ou.edu/wx/p/r/conus/fcst/

# Weather Forecasting

- Weather forecasting is one of many ***transport*** problems.
- The goal is to predict future weather conditions by simulating the movement of fluids in Earth's atmosphere.
- The physics is the Navier-Stokes Equations.
- The numerical method is Finite Difference.

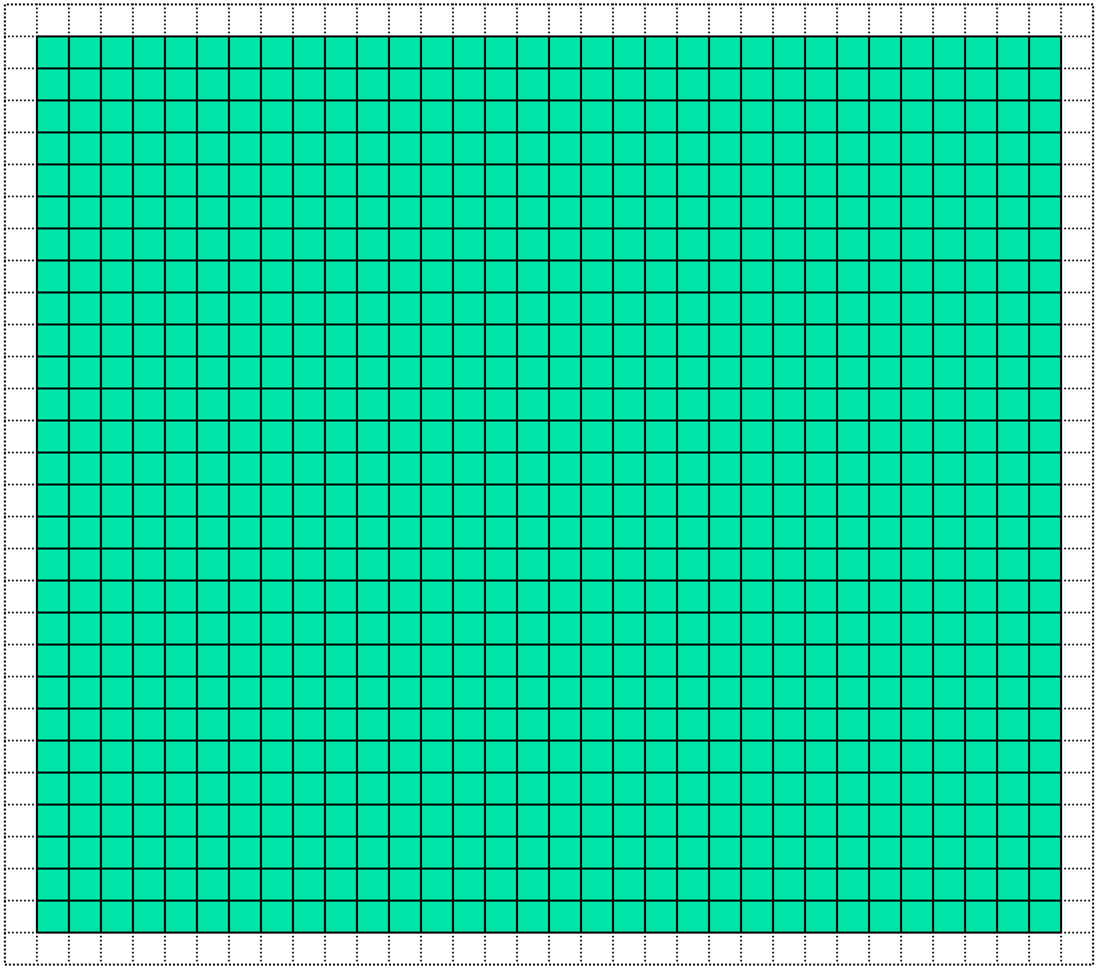
# Cartesian Mesh



# Finite Difference

$$
\begin{aligned}
unew(i,j,k) = \mathrm{F}(uold,\ i,\ j,\ k,\ \Delta t) = \\
& \mathrm{F}(uold(i,j,k), \\
& \quad uold(i-1,j,k),\ uold(i+1,j,k), \\
& \quad uold(i,j-1,k),\ uold(i,j+1,k), \\
& \quad uold(i,j,k-1),\ uold(i,j,k+1),\ \Delta t)
\end{aligned}
$$

# Ghost Boundary Zones



# Software Strategies for Weather Forecasting on Multicore/Many-core

# Tiling NOT Good for Weather Codes

- Weather codes typically have on the order of 150 3D arrays used in each timestep (some transferred multiple times in the same timestep, but let's ignore that for simplicity).
- These arrays typically are single precision (4 bytes per floating point value).
- Thus, a typical weather code uses about 600 bytes per mesh zone per timestep.
- Weather codes typically do 5,000 to 10,000 calculations per mesh zone per timestep.
- So, the ratio of calculations to data is less than 20 to 1 – much less than the 45 to 1 needed (on mid-2005 hardware).

# Weather Forecasting and Cache

- On current weather codes, data decomposition is by process. That is, each process gets one subdomain.
- As CPUs speed up and RAM sizes grow, the size of each processor's subdomain grows too.
- However, given RAM bandwidth limitations, this means that performance can only grow with RAM speed – which increases slower than CPU speed.
- If the codes were optimized for cache, would they speed up more?
- First: How to optimize for cache?

# How to Get Good Cache Reuse?

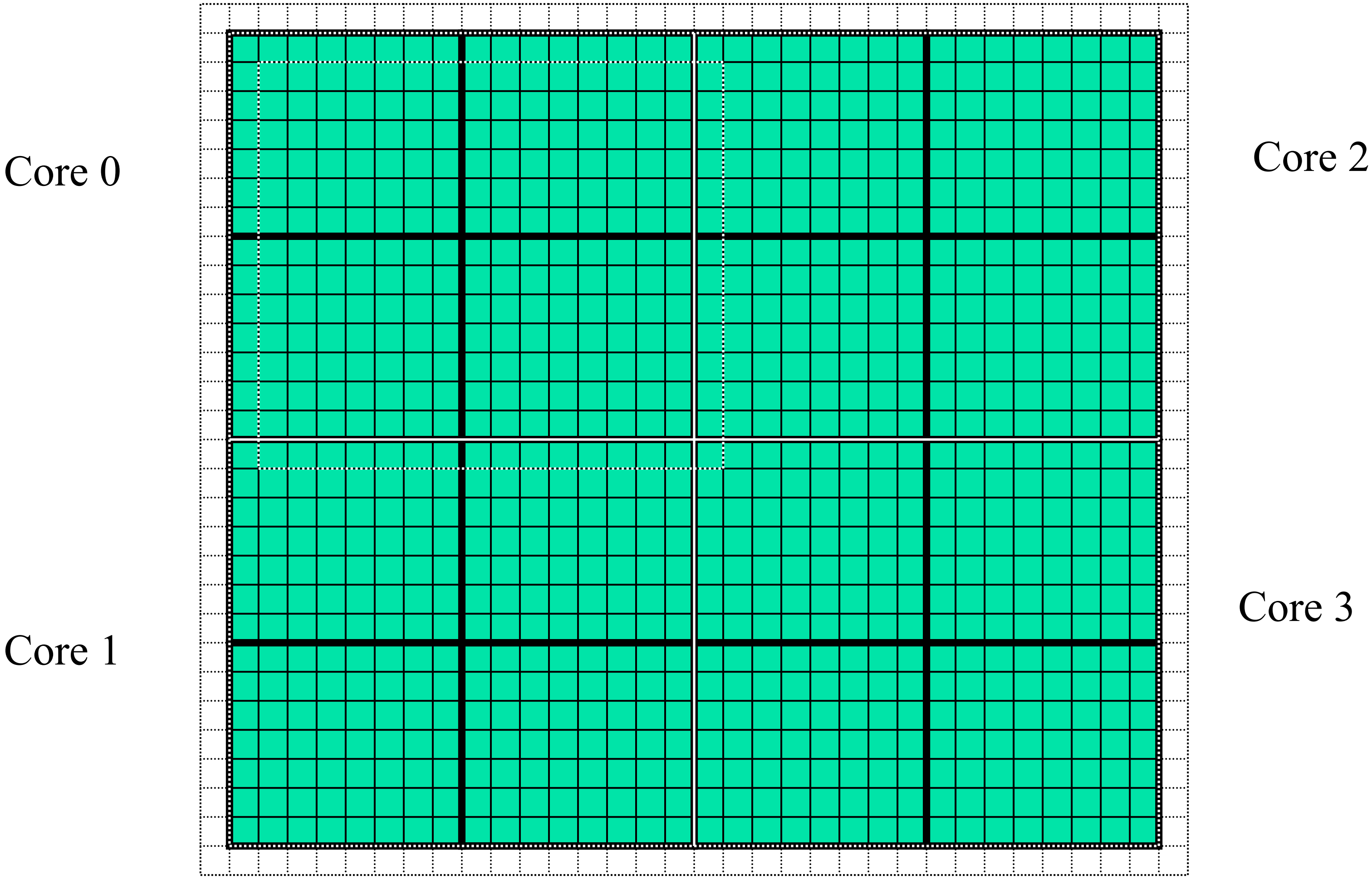
1. Multiple independent subdomains per processor.
2. Each subdomain fits entirely in L2 cache.
3. Each subdomain's page table entries fit entirely in the TLB.
4. Expanded ghost zone stencil allows multiple timesteps before communicating with neighboring subdomains.
5. Parallelize along the Z-axis as well as X and Y.
6. Use higher order numerical schemes.
7. Reduce the memory footprint as much as possible.

Coincidentally, this also reduces communication cost.

# Cache Optimization Strategy: Tiling?

Would tiling work as a cache optimization strategy for weather forecasting codes?

# Multiple Subdomains Per Core



# Why Multiple Subdomains?

- If each subdomain fits in cache, then the CPU can bring all the data of a subdomain into cache, chew on it for a while, then move on to the next subdomain: lots of cache reuse!
- Oh, wait, what about the TLB? Better make the subdomains smaller! (So more of them.)
- But, doesn't tiling have the same effect?

# Why Independent Subdomains?

- Originally, the point of this strategy was to hide the cost of communication.
- When you finish chewing up a subdomain, send its data to its neighbors non-blocking (**`MPI_Isend`**).
- While the subdomain's data is flying through the interconnect, work on other subdomains, which hides the communication cost.
- When it's time to work on this subdomain again, collect its data (**`MPI_Waitall`**).
- If you've done enough work, then the communication cost is zero.

# Expand the Array Stencil

- If you expand the array stencil of each subdomain beyond the numerical stencil, then you don't have to communicate as often.
- When you communicate, instead of sending a slice along each face, send a slab, with extra stencil levels.
- In the first timestep after communicating, do extra calculations out to just inside the numerical stencil.
- In subsequent timesteps, calculate fewer and fewer stencil levels, until it's time to communicate again – less total communication, and more calculations to hide the communication cost underneath!

# An Extra Win!

- If you do all this, there's an amazing side effect: you get better cache reuse, because you stick with the same subdomain for a longer period of time.
- So, instead of doing, say, 5000 calculations per zone per timestep, you can do 15000 or 20000.
- So, you can better amortize the cost of transferring the data between RAM and cache.

# New Algorithm

```
DO timestep = 1, number_of_timesteps, extra_stencil_levels
    DO subdomain = 1, number_of_local_subdomains
        CALL receive_messages_nonblocking(subdomain,
                 timestep)
        DO extra_stencil_level=0, extra_stencil_levels - 1
            CALL calculate_entire_timestep(subdomain,
                     timestep + extra_stencil_level)
        END DO
        CALL send_messages_nonblocking(subdomain,
                 timestep + extra_stencil_levels)
    END DO
END DO
```

# Higher Order Numerical Schemes

- Higher order numerical schemes are great, because they require more calculations per zone per timestep, which you need to amortize the cost of transferring data between RAM and cache. Might as well!
- Plus, they allow you to use a larger time interval per timestep (**`dt`**), so you can do fewer total timesteps for the same accuracy – or you can get higher accuracy for the same number of timesteps.

# Parallelize in Z

- Most weather forecast codes parallelize in X and Y, but not in Z, because gravity makes the calculations along Z more complicated than X and Y.
- But, that means that each subdomain has a high number of zones in Z, compared to X and Y.
- For example, a 1 km CONUS run will probably have 100 zones in Z (25 km at 0.25 km resolution).

# Multicore/Many-core Problem

- Most multicore chip families have relatively small cache per core (e.g., 2 MB) – and this problem seems likely to remain.
- Small TLBs make the problem worse: 512 KB per core rather than 2 MB.
- So, to get good cache reuse, you need subdomains of no more than 512 KB.
- If you have 150 3D variables at single precision, and 100 zones in Z, then your horizontal size will be 3 x 3 zones – just enough for your stencil!

# What Do We Need?

- We need much bigger caches!
  - 16 MB cache → 16 x 16 horizontal including stencil
  - 32 MB cache → 23 x 23 horizontal including stencil
- TLB must be big enough to cover the entire cache.
- It'd be nice to have RAM speed increase as fast as core counts increase, but let's not kid ourselves.

# Next Time

**Part VII:**

**High Throughput Computing**

# To Learn More Supercomputing

http://www.oscer.ou.edu/education.php

# Thanks for your attention!

# Questions?

# Virtual Memory

- Every CPU family today uses ***virtual memory***, in which disk pretends to be a bigger RAM.
- Virtual memory capability **can't be turned off**.
- RAM is split up into ***pages***, typically **4 KB** each.
- Each page is either **in RAM** or **out on disk**.
- To keep track of the pages, a ***page table*** notes whether each table is in RAM, where it is in RAM (that is, physical address and virtual address are different), and some other information.
- So, a 4 GB physical RAM would need over a million ***page table entries***.

# Why Virtual Memory is Slow

- When you want to access a byte of memory, you have to find out whether it's in physical memory (RAM) or virtual disk (disk) – and the page table is in RAM!
- A page table of a million entries can't fit in a 2 MB cache.
- So, each memory access (load or store) is actually 2 memory accesses: the first for the page table entry, and the second for the data itself.
- This is slow!
- And notice, this is assuming that you don't need more memory than your physical RAM.

# The Notorious T.L.B.

- To speed up memory accesses, CPUs today have a special cache just for page table entries, known as the ***Translation Lookaside Buffer*** (TLB).
- The size of TLBs varies from 64 entries to 1024 entries, depending on chip families.
- At 4 KB pages, this means that the size of cache covered by the TLB varies from 256 KB to 4 MB.

# The T.L.B. on a Current Chip

On Intel Core Duo ("Yonah"):

- Cache size is 2 MB per core.
- Page size is 4 KB.
- A core's data TLB size is 128 page table entries.
- Therefore, D-TLB only covers 512 KB of cache.

# The T.L.B. on a Current Chip

On Intel Core Duo ("Yonah"):

- Cache size is 2 MB per core.
- Page size is 4 KB.
- A core's data TLB size is 128 page table entries.
- Therefore, D-TLB only covers 512 KB of cache.
- Mesh: At 100 vertical levels of 150 single precision variables, 512 KB is a 3 x 3 vertical domain – **nothing but ghost zones**!
- The cost of a TLB miss is 49 cycles, equivalent to as many as **196 calculations**! (4 FLOPs per cycle)

http://www.digit-life.com/articles2/cpu/rmma-via-c7.html