Martin Berzins
School of Computing

CS6230 PARALLEL HIGH PERFORMANCE COMPUTING

Professor Martin Berzins, Rm 4803 WEB email: mb@cs.utah.edu

Lectures Monday Wednesday 8.05-9.25 WEB 122
Also some use of Friday 8.05– 9.25 WEB 122
Practical Classes - Use CHPC (Telluride Cluster) , CADE LAB, perhaps Raven cluster or National Supercomputer Resources
Assessments (4) 60% Coursework+ 40% Mid Term and Final Exam
Extra credit option – more later.
Web site **http://www.sci.utah.edu/~mb/Teaching**

Office hours – by arrangement by mailing me

Center for high Performance Computing
Telluride Cluster Overview

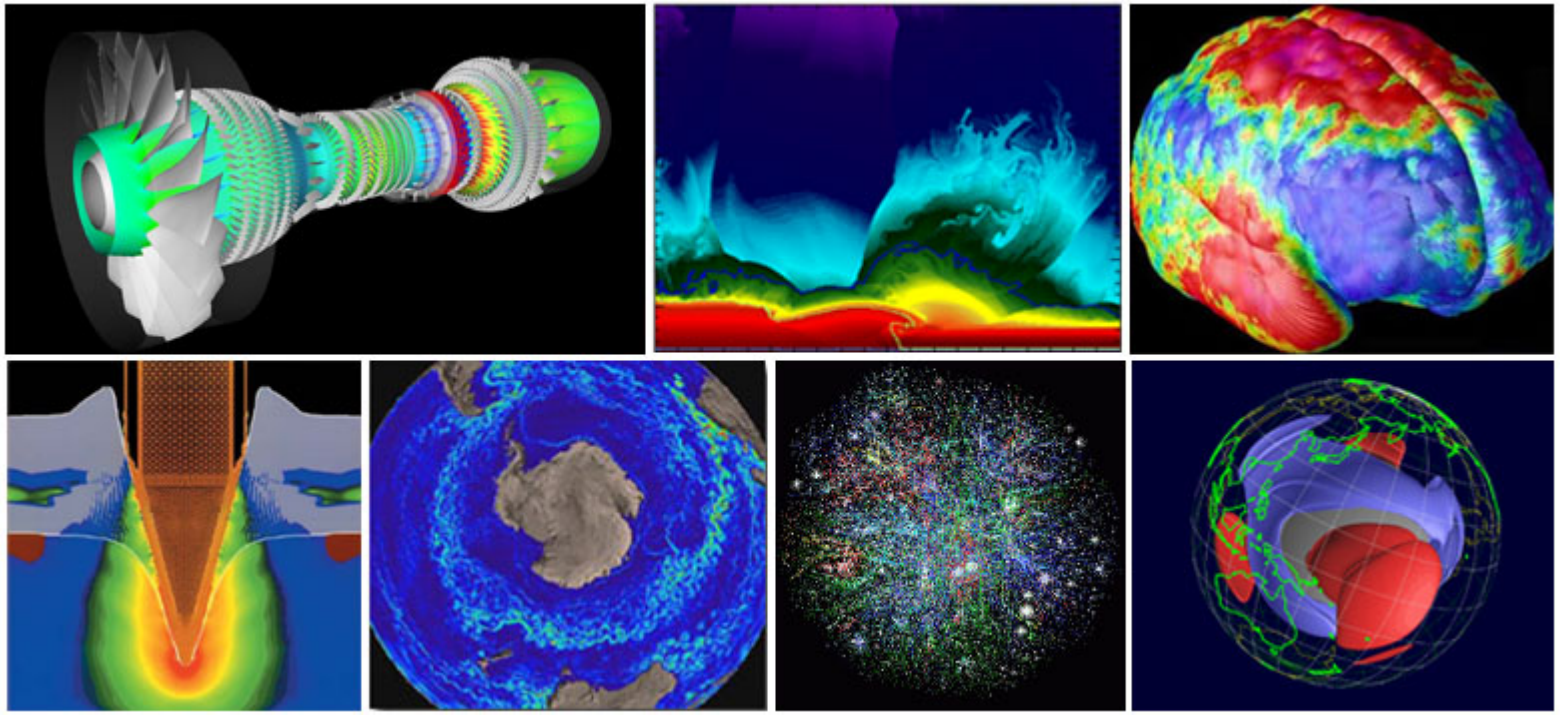
- For dedicated use by one research group
- 72 nodes, 576 processors
- 2.333 (48 nodes) and 2.66 (24 nodes) GHz processors
- 16 Gbytes memory per node
- Inifiniband interconnect

Other machines available for development

Turretarch 10-13 and 18-21

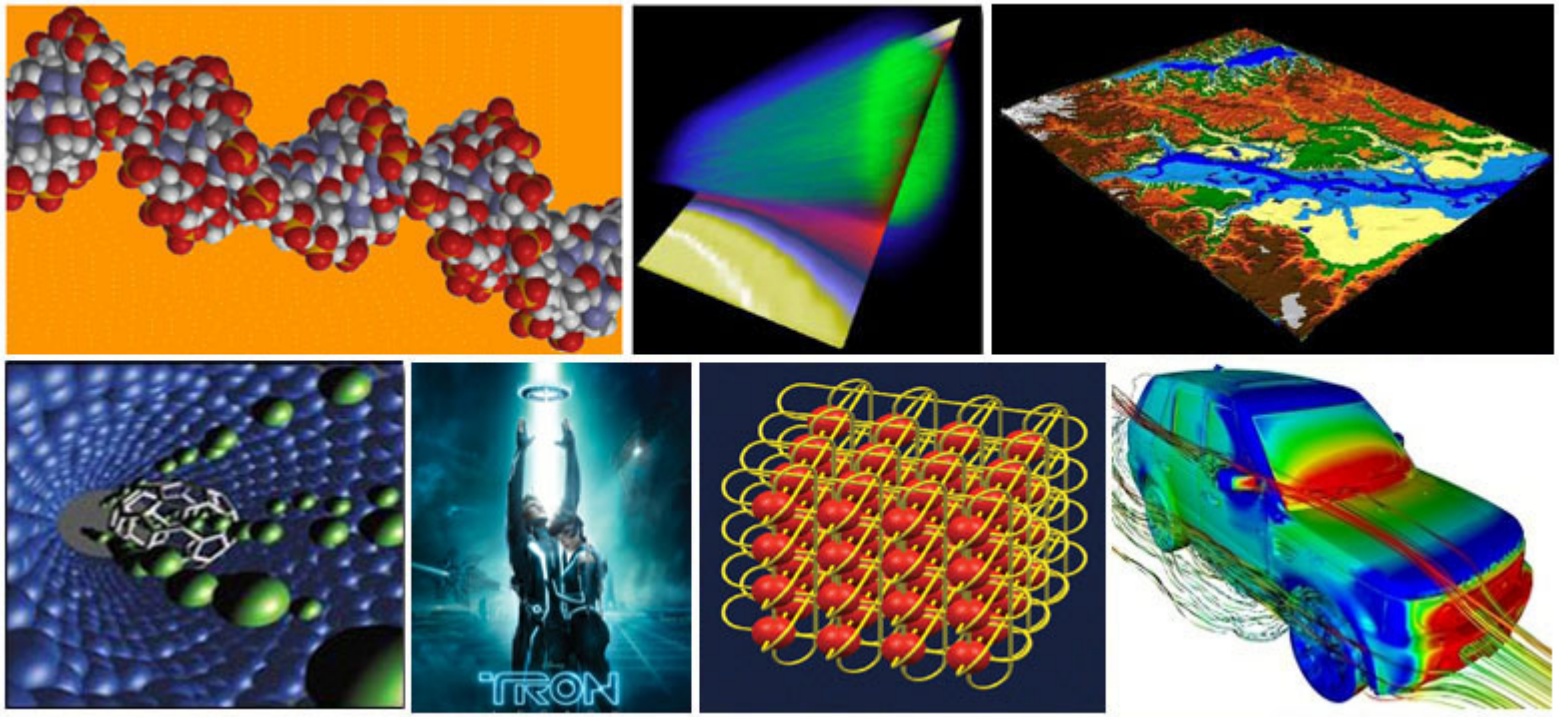
Why Parallel **Computing**?

Science and Engineering: Historically, parallel computing has been considered to be "the high end of computing", and has been used to model difficult problems in many areas of science and engineering:



- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical Engineering - from prosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics

Industrial and Commercial: Today, commercial applications provide an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. For example:

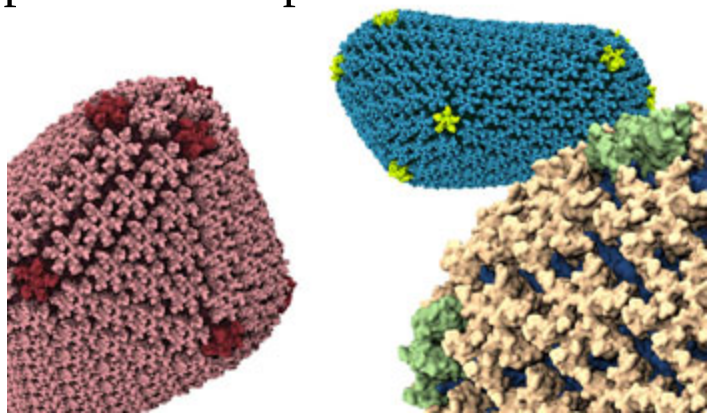


- Databases, data mining
- Oil exploration
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Financial and economic modeling
- Management of national and multi-national corporations
- Advanced graphics and virtual reality, particularly in the entertainment industry
- Networked video/ multi-media technologies

Motivation: Determining The Structure of the HIV Capsid using Blue Waters

- NSF Researchers have determined the precise chemical structure of the HIV capsid, a protein shell that protects the virus's genetic material and is a key to its ability to infect and debilitate the human body's defense mechanism.
- The Capsid is a target for the development of new antiretroviral drugs that suppress the HIV virus and stop the progression of AIDS [Schulten et al. *Nature* (5/30/13).
- This required a 64M atom simulation on NSF's Blue Waters machine, one of the world's most powerful computers, without which scientists were unable to decipher in atomic-level detail the entire HIV capsid--an assemblage of more than 1,300 identical proteins forming a cone-shaped structure. The simulations that added the missing pieces to the puzzle.

Three different renderings of the HIV capsid, with multiple colors.



Spanish Fork Accident 8/10/05

Speeding truck with 8000 explosive boosters each with

2.5-5.5 lbs of explosive overturned and caught fire

Experimental evidence suggests that a transition from deflagration

to detonation took place. Why?

Deflagration wave moves at ~400m/s not all explosive consumed. Detonation wave moves 8500m/s all explosive consumed.

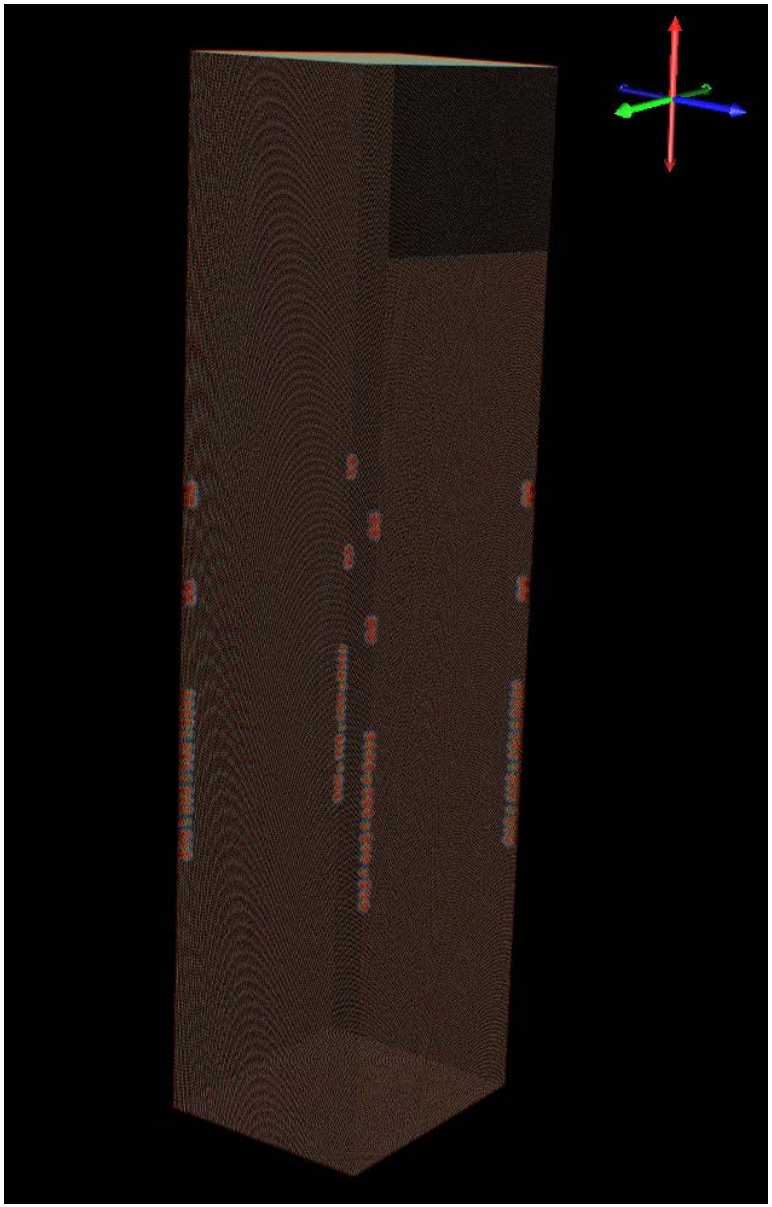
Jaqueline Beckvermit (chem grad student) has 200M CPU hours to solve this problem in 2014

540K T in corner

5Gpa P in corner

ProblemReducing Explosive Hazards

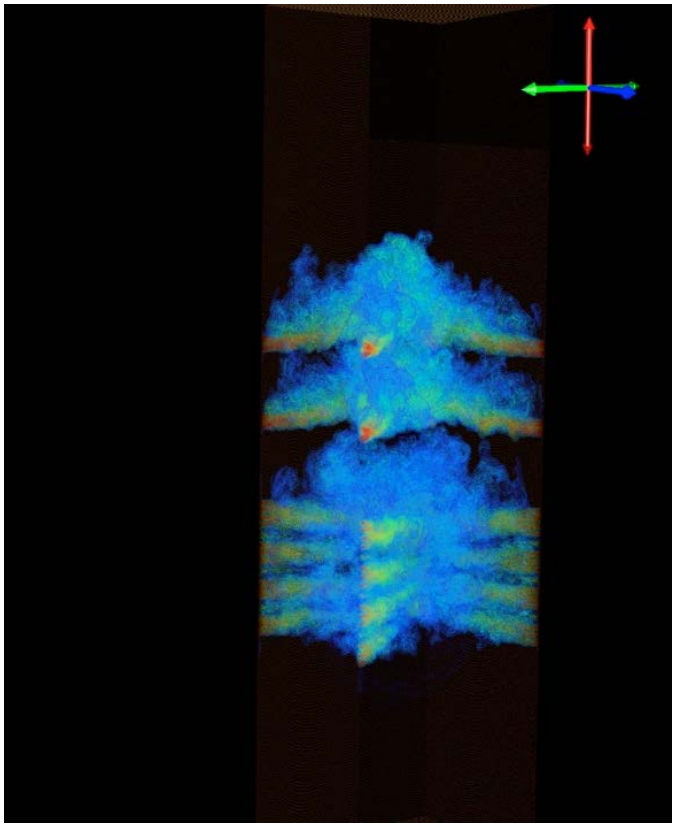
Design of Alstom Clean coal Boilers



For 350MWe boiler problem. LES resolution needed: 1mm per side for each computational volume = 9x 10^{12} cells
Based on initial runs - to run in 48 hours of wall clock time requires 50-100M fast cores.

Prof. Phil Smith and Marti Berzins lead One of 3 PSAAP II centers in the US $20M 5 years

Problem 2. Design of Alstom Clean coal Boilers



Temperature field

For 350MWe boiler problem. LES resolution needed: 1mm per side for each computational volume = 9x 10^{12} cells Based on initial runs - to run in 48 hours of wall clock time requires 50-100M fast cores.

Prof. Phil Smith and Marti Berzins lead One of 3 PSAAP II centers in the US $20M 5 years

What is the course about?

Designing algorithms and writing programs that use multiple cores (processors) or multiple cores/accelerators to solve large and medium scale computational problems, .

How do you make best use of a multicore architecture or 100s, 1000s or 100,000s of processors to ?

Solve larger problems efficiently **WEAK SCALABILITY**

(e.g. a problem twice as large is solved in the same time on twice as many cores)

Solve the same (larger) problem more quickly

STRONG SCALABILITY

(e.g. solve the same problem twice as quickly on a computer twice as large)

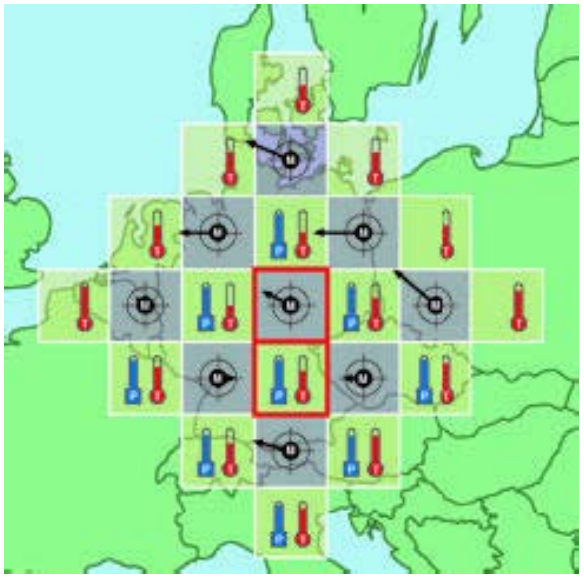
These approaches are being used to address the computational challenges such as those shown above.

Students who took this class are now using some of the largest parallel machines

With the Uintah framework (www.uintah.utah.edu)

The idea isn't new: Weather Modeling

Louis Fry Richardson's Computation, 1917

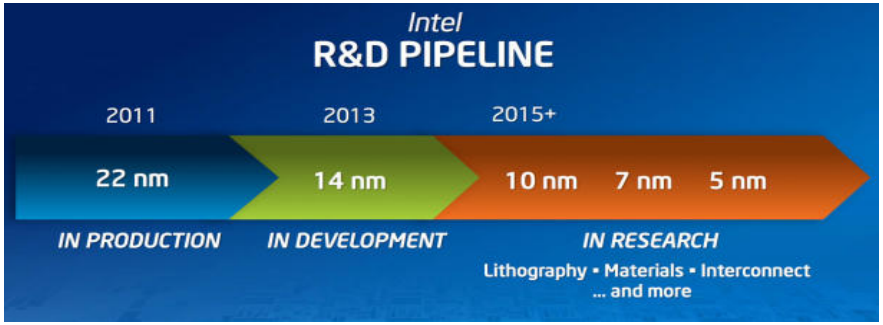


Courtesy John Burkhardt, Virginia Tech

Richardson's computers were people who each had a portion of the domain and who passed information to one an other. He proposed doing global weather modeling by seating his "computers" in the Albert Hall each of them computing a section of the globe.

- **If we can we will solve more complex and larger computational problems**
- The move to multicore architectures means that parallel computing techniques are needed to exploit existing and future hardware
- Energy problems mean that clock speeds can't easily increase.
- Improved processes mean that chips with feature sizes of 45nm 32nm , 22nm and soon 17nm are both here and possible
- More (but simpler cores) can be placed on a chip. 2, 4 and 8 . Intel accelerators have 60 cores.
- At the same time larger and larger parallel machines are being built with many 100,000s of cores .
- **The combination of multicores accelerators and large scale parallelism makes understanding parallel computing and algorithms more important than ever before.**

Why Parallel Computing?

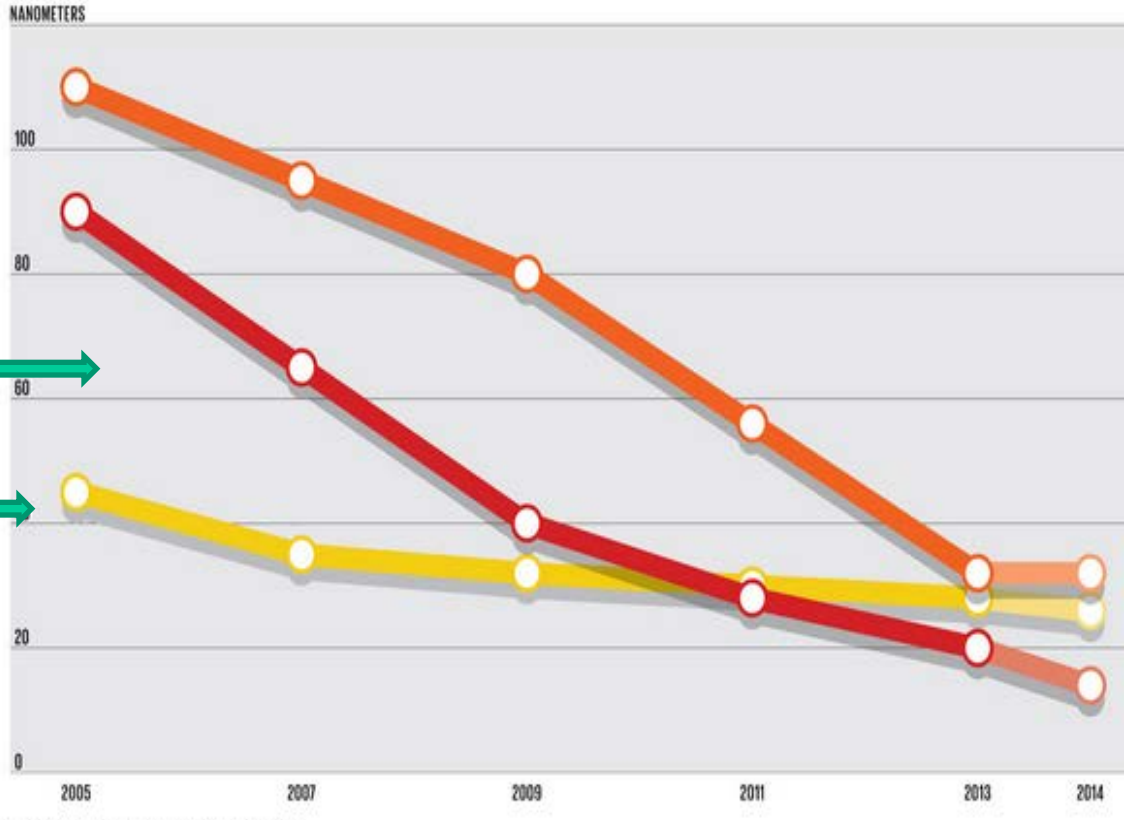


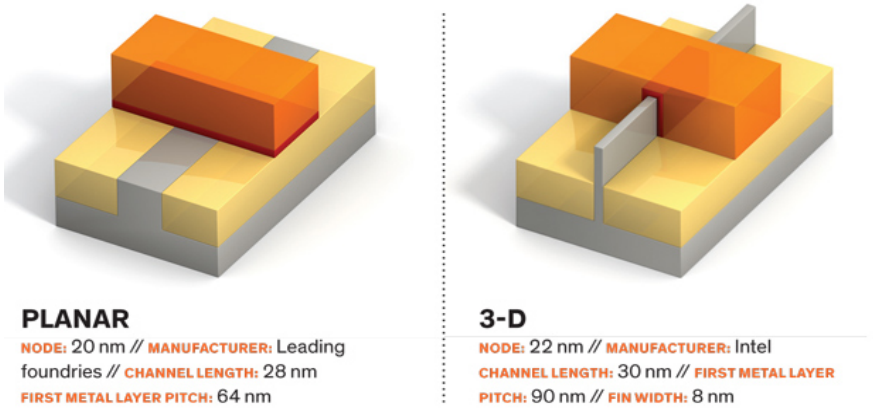
Status of Moore's Law - Not what you think

Metal one half pitch
(half dist. Spanned by wire width and space to next one on first metal chip layer)

Node name

Transistor gate length



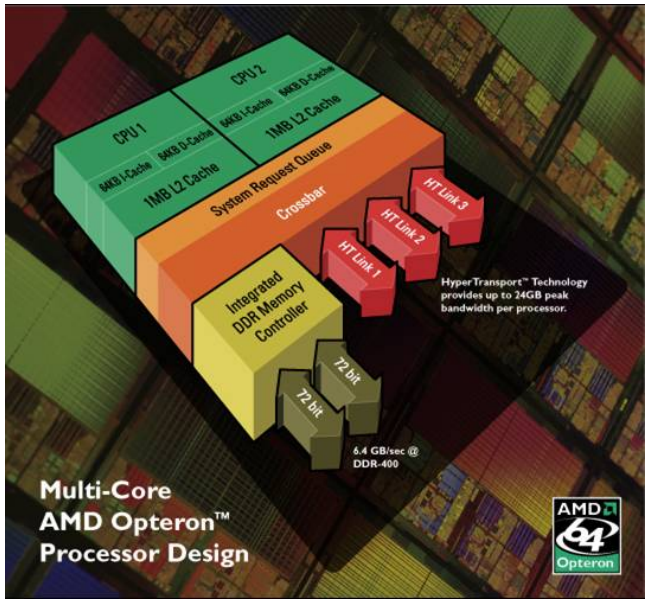


Fundamental changes in technology are taking place, 3D transistors memory stacking

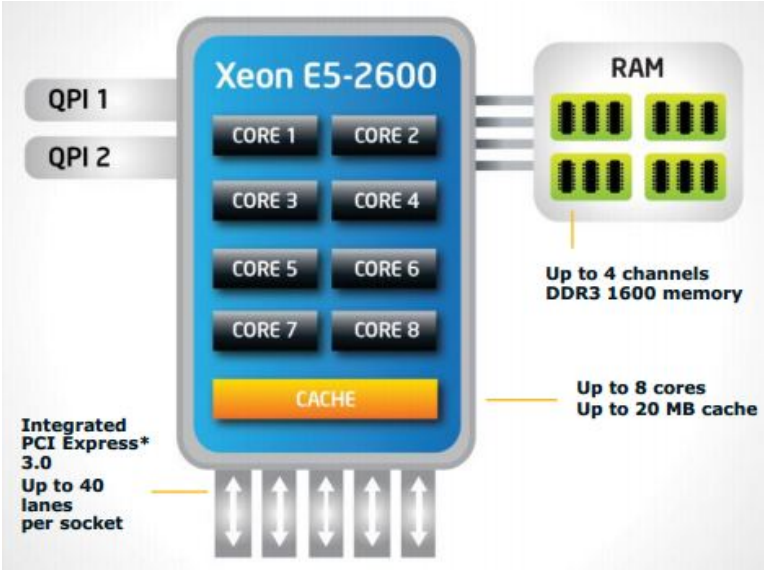
What is a Core/Socket/Node?

- A core is a single chip package that fits in a socket
- ≥1 core (not much point in <1 core…)
  - Cores perform arithmetic and can have functional units,
  - Cores can be fast or slow, just as today
- Shared resources
  - More cache
  - Other integration: AMD Northbridge on-chip crossbar switch, memory controllers, high-speed serial links, etc.
- One system interface no matter how many cores
  - Number of signal pins doesn't scale with number of cores
- Nodes have sockets each of which has multiple cores and now often have accelerators such as GPUs

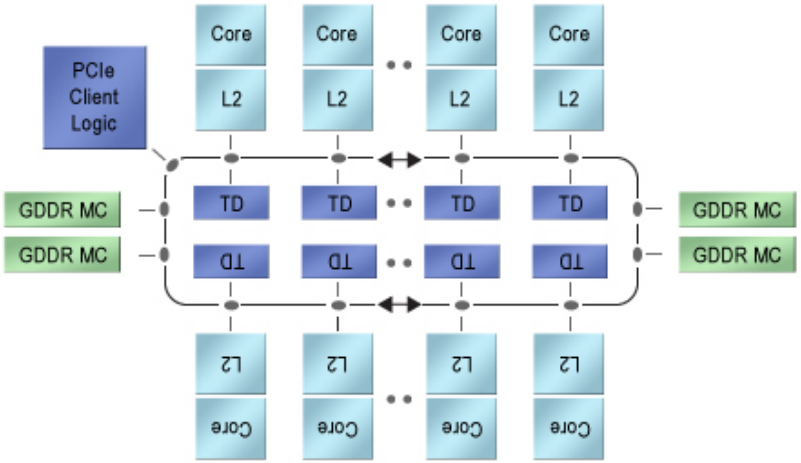
AMD and Intel Multi-Core Processor 2012

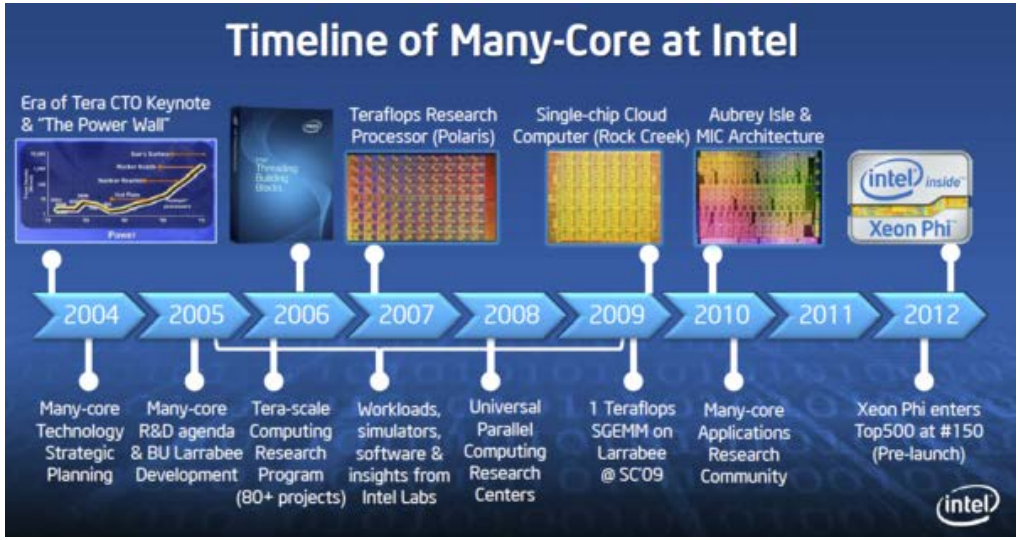


8-core Sandybridge processor with shared level 3 cache



Intel Xeon Phi Accelerator 2012

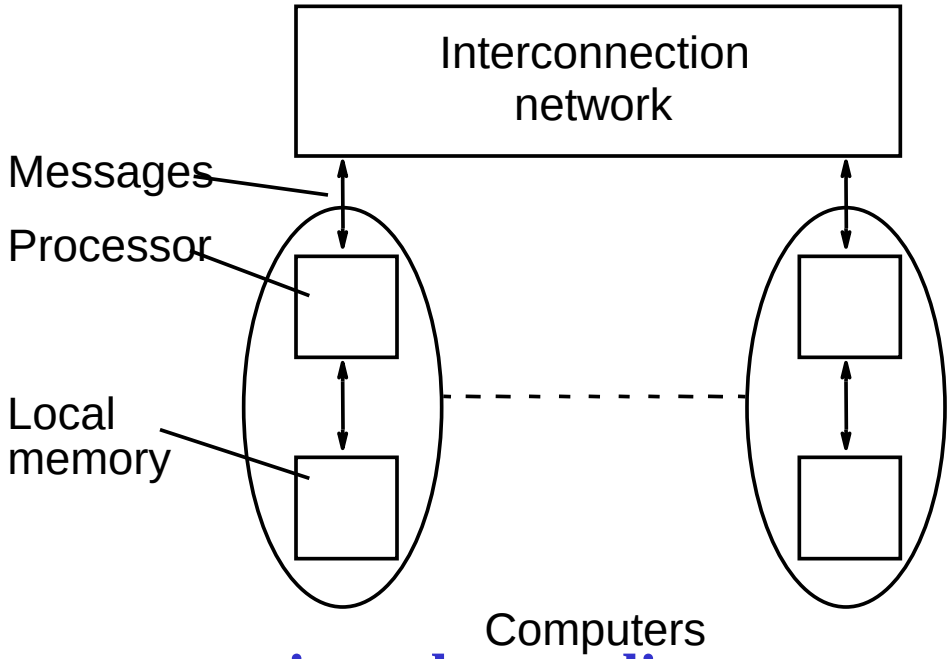




Example of a Distributed Memory Parallel Computer

Single source program written and each processor executes its personal copy of this program, although independently and with synchronization at certain points.

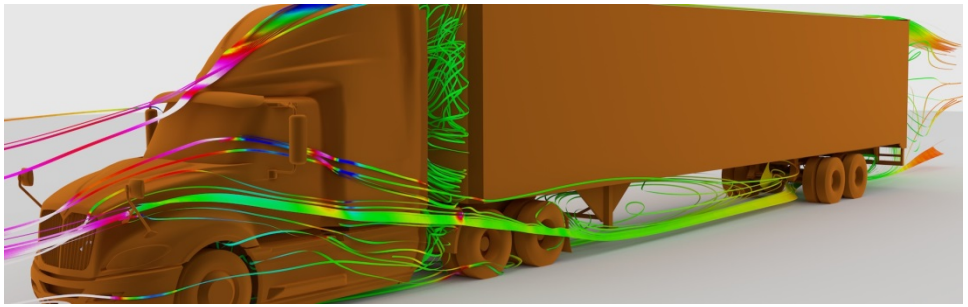
Processor can now be multicore cpu with or without GPU and/or Intel Xeon Phi accelerator

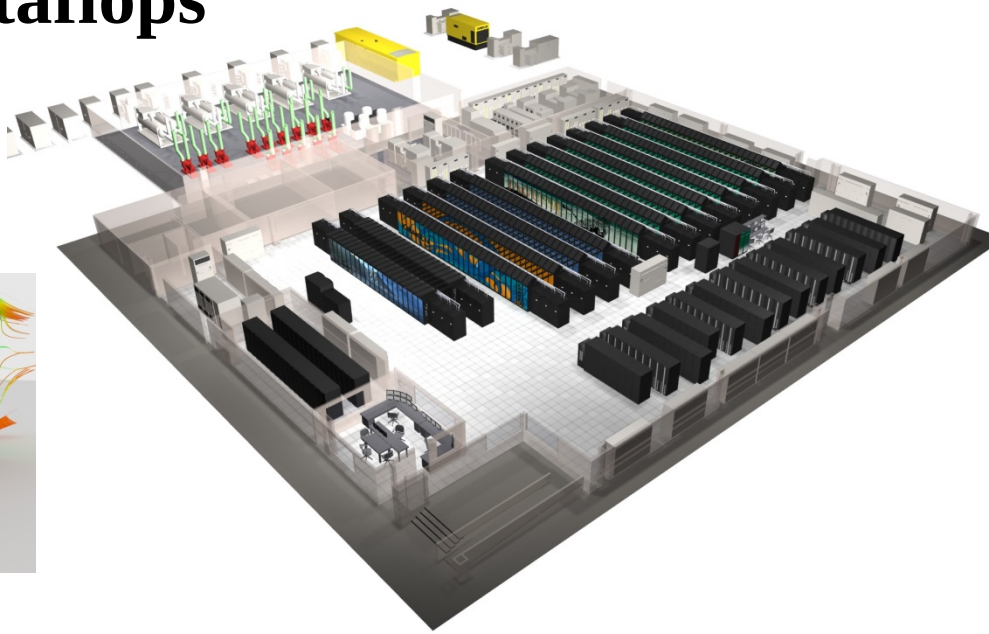


All computers communicate by sending messages through the interconnection network: e.g. MPI standard

DoE ORNL's Jaguar XT5 224K cores 1.75 petaflops

NSF NICs (Oak Ridge) Kraken
Cray XT5 112K cores 0.8 petaflops

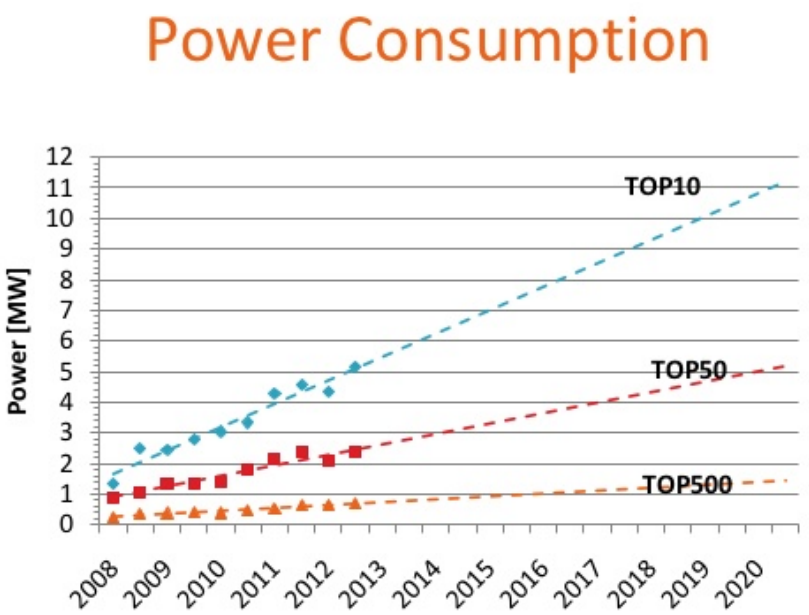




Power is an Industry Wide Problem

The New York Times

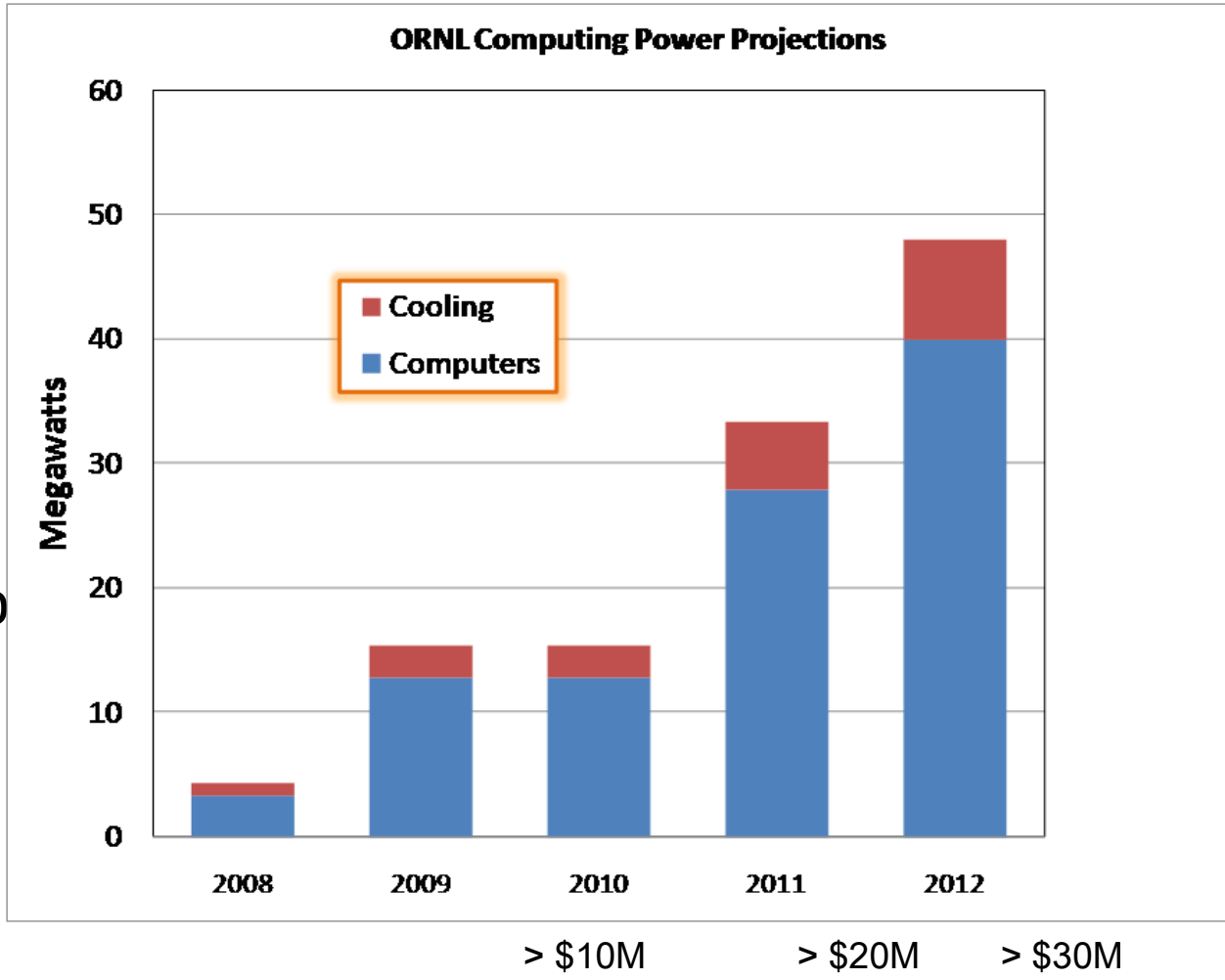
30 billion watts and rising: balancing the internet's energy and infrastructure needs



Google Plant in The Dalles, Oregon, from NYT, June 14, 2006

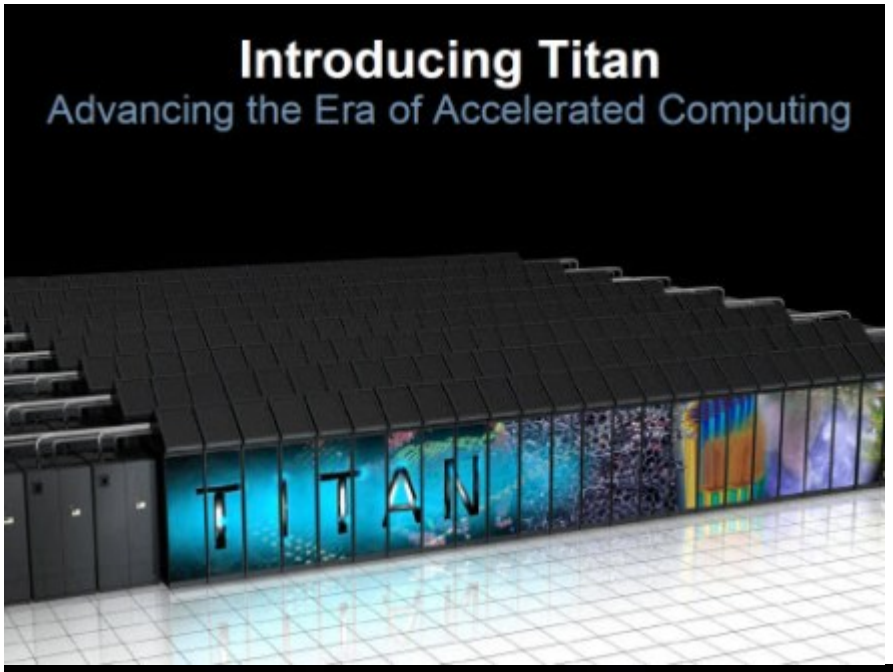
ORNL/UTK Computer Power Cost Projections 2008-2012

- Over the next 5 years ORNL/UTK will deploy 2 large Petascale systems
- Used15 MW 2008 $10M
- By 2012 close to 50MW?
- Power costs close to $10M today.
- Cost estimates based on $0.07 per KwH



Power becomes the architectural driver for future large systems

Did this happen?



Increasing Performance Does not always lead to a Corresponding Power Increase

Breakthrough Titan Performance

Jaguar Specs (2011)	
Compute Nodes	18,688
Login & I/O Nodes	256
Memory per node	16 GB
# of Opteron cores	224,256
# of NVIDIA K20 "Kepler" accelerators (2013)	N/A
Total System Memory	300 TB
Total System Peak Performance	2.3 Petaflops

Titan Specs (2012)	
Compute Nodes	18,688
Login & I/O Nodes	512
Memory per node	32 GB + 6 GB
# of Opteron cores	299,008
# of NVIDIA K20 "Kepler" accelerators (2013)	18,688
Total System Memory	710 TB
Total System Peak Performance	20+ Petaflops

DOEs Titan draws 12.7 MW, 2 MW more than Jaguar did, but it is almost ten times as fast in terms of floating point calculations by using both cpus and GPUs.

Programming Parallel Computers.

Parallel Programming environments since 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

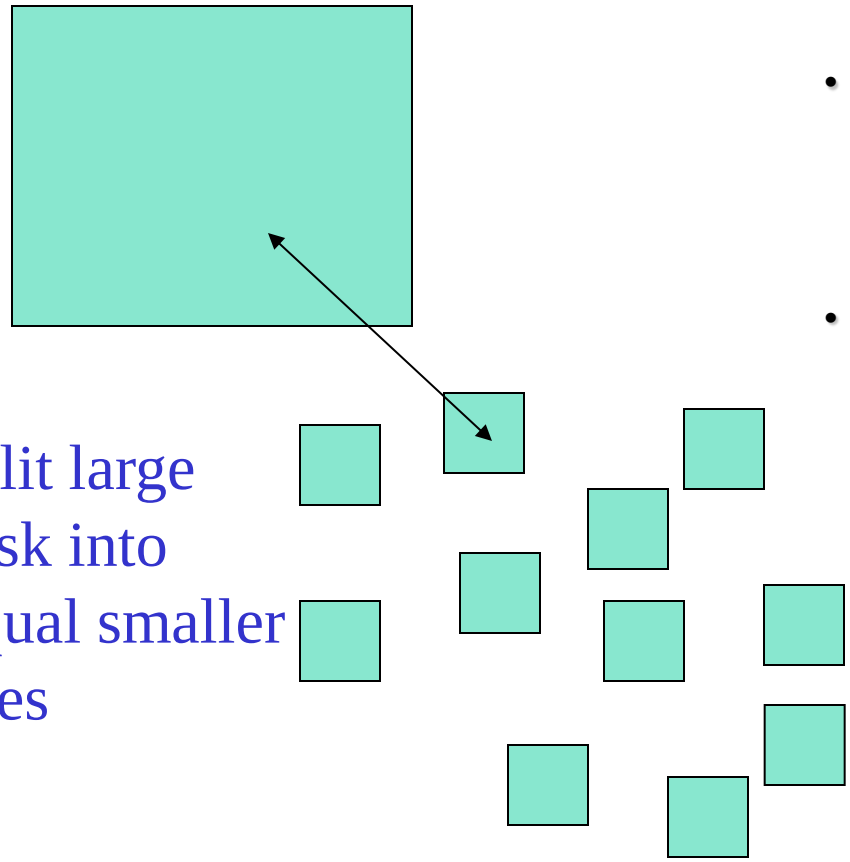
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

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

Language acceptance is not a technical issue; it is a social and commercial issue too

Parallel Programming Summary

Domain Decomposition Done Well: Load Balanced



- You have to spread *something* out.
- These can theoretically be many types of abstractions: work, threads, tasks, processes, data,…
- But what they *will* be is your data. And then you will use MPI, and possibly OpenMP, to operate on that data.
- A parallel algorithm can only be as fast as the slowest chunk.
  - Balance the number crunching
  - Might be dynamic
- Communication will take time
  - Usually orders of magnitude difference between registers, cache, memory, network/remote memory, disk
  - Data locality and “neighborly-ness” matters very much.

How do we program Parallel Computers today ?

- **Message Passing** – write programs that exist on each processor and pass messages to communicate data between the processors.
- **Threads** – write programs that involve setting up interacting execution streams of instructions that share data .
- **Open MP, OPENACC** write serial programs and then modify the programs by inserting directives to tell the compiler how to parallelize that part of the code.
- Use a **specialist parallel programming language** such as UPC (Unified Parallel C) , CUDA
- Maybe use a **dataflow language** like CnC: Concurrent collections

Typically message passing is used to communicate between processors while Open MP or threads are used when processors (or cores ?) share memory
New languages are attractive - if only they would take off and persist!

Example - Dot Product $$Sum = \sum_{i=1}^{n} a_i b_i$$

Sequential Dot Product

```
int main(argc,argv)
int argc;
char *argv[];
{double sum;  double a[256], b[256];
 int n;
 n =256;
 for (i =0; i < n; i++) {
    a[i] = i*0.5;
    b[i] = i*2.0;    }
 sum = 0.0;
 for (i = 1; i<n; i++) {
    sum =sum + a[i]*b[i];
 }
 printf (" sum= %f",sum);
 }
```

Dot Product OPENMP version

```
int main(argc,argv)
int argc;
char *argv[];
{double sum;  double a[256], b[256];
 int n;      n =256;
#pragma omp parallel for private(i) shared(a,b)
   for (i =0; i < n; i++) {
      a[i] = i*0.5;
      b[i] = i*2.0;
   }
   sum = 0.0;
 #pragma omp for reduction (+:sum)
   for (i = 1; i<n; i++) {
      sum =sum + a[i]*b[i];
   }
   printf (" sum= %f",sum);
   }
```

MPI VERSION

```
int main(argc,argv)
int argc;    char *argv[];
{double sum, sum_local;  double a[256], b[256];
 int n; numprocs, myid, my_first, my_last;
 n = 256;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 my_first = myid*n/numprocs;
 my_last  = (myid+1) * n/numprocs;

 for (i =0; i < n; i++) {
    a[i] = i*0.5;    b[i] = i*2.0;          }
 sum = 0.0;
 for (i = my_first; i< my_last; i++) {
    sum_local =sum_local  + a[i]*b[i];  }
 MPI_Allreduce(&sum_local,&sum, 1,MPI_DOUBLE, MPI_SUM,
                        MPI_COMM_WORLD);
 if(myid == 0)printf (" sum= %f",sum);
 }
```

UPC demo program for the dot product of vectors x and y.

We use upc_forall to compute partial sums and then use a upc_lock to protect the part where we reduce the individual partial sums to the total dot product. The other part of the demo is to show difference between the blocked and cyclic distribution. The arrays x_cyc and y_cyc are declared with block size 1 and arrays x_blk and y_blk are declared with [*]] block size. This can't change the results, but it does change the partial sums collected by each thread.

<dotproduct.c>=

```
//dotproduct.c -- simple dot product //Intro: upc_forall, locks, cyclic vs blocked
#include <stdio.h> #include <upc.h>
#define NperTHREAD 100
#define SIZE (NperTHREAD * THREADS)
#define BLOCK NperTHREAD

shared float dot_cyc, dot_blk;
shared float x_cyc[SIZE], y_cyc[SIZE];
shared [*] float x_blk[SIZE], y_blk[SIZE];
upc_lock_t *dotlock;
```

```
main ()

{ int i; float mydot;
dotlock = upc_all_lock_alloc();
upc_lock_init( dotlock );
if(MYTHREAD == 0) dot_blk = dot_cyc = 0.0; upc_barrier(0);
 // "affinity" is an int so it is (i mod THREADS)
upc_forall( i=0; i< SIZE; i++; i )
   { x_cyc[i] = (float) i; y_cyc[i] = x_cyc[i];

      x_blk[i] = (float) i; y_blk[i] =  x_blk[i]; }
upc_barrier(1);
mydot = 0.0;
// "affinity" is found from affinity of x_cyc[i]
upc_forall( i=0; i< SIZE; i++; &x_cyc[i] )
       mydot += x_cyc[i] * y_cyc[i];
 printf ("Process %2d holds %g (cyclic)\n", MYTHREAD, mydot);
upc_lock(dotlock);
    dot_cyc = dot_cyc + mydot;
upc_unlock(dotlock);

upc_barrier(2);
```

```
if( MYTHREAD == 0 )
    printf("Total (cyclic) is %g\n", dot_cyc);
upc_barrier(3);
mydot = 0.0;
// "affinity" is found from affinity of x[i]

upc_forall( i=0; i< SIZE; i++; &x_blk[i] )
    mydot += x_blk[i] * y_blk[i];

printf ("Process %2d holds %g (blocked)\n", MYTHREAD, mydot);
upc_lock(dotlock);
     dot_blk = dot_blk + mydot;
upc_unlock(dotlock);
upc_barrier(2);
if( MYTHREAD == 0 )
   printf("Total (blocked) is %g\n", dot_blk);

}
```

Message Passing

- Pros
  - Flexible and very widely used, low level
  - Efficient what the machine does anyway
  - Implementations Solid excpet perhaps on latest machines
  - Algorithmic Support – much existing knowledge
  - Debugging Support – except on very large machines
- Cons
  - Lower level means more detail for the coder
  - Debugging requires more attention to detail
  - Development usually requires a “start from scratch” approach
  - Domain decomposition and memory management must be explicit

MPI

Has been around a longtime (~20 years inc. PVM)
Dominant standard
Will be around a longtime (on all new platforms/roadmaps)
Lots of libraries
Lots of algorithms
Very scalable (500K+ cores right now)
Portable
Works with hybrid models accelerators GPUs

Data Parallel

Only one executable.

Do computation on arrays of data using array operators.

Do communications using array shift or rearrangement operators.

Good for problems with static load balancing that are array-oriented SIMD machines.

Variants:

- FORTRAN 90
- CM FORTRAN
- HPF
- C*
- GPU Langauges (CUDA)

Pros:

1. Scales transparently to different size machines
2. Easy debugging, as there I sonly one copy of code executing in highly synchronized fashion

Cons:

1. Much wasted synchronization
2. Sometimes difficult to balance load

Threads

Splits up tasks (as opposed to arrays in data parallel) such as loops amongst separate processors.

Do communication as a side effect of data loop distribution. Not an big issue on shared memory machines. Impossible on distributed memory.

Common Implementations:

pthreads (Unix standard)

OpenMP

Strengths:

1. Doesn't perturb data structures, so can be incrementally added to existing serial codes.

Weaknesses:

1. Serial code left behind will be problematic
2. Can only be used at socket or shared memory machine.

OpenMP Pros and Cons

- Simple additions to existing code
- Standard and widely available (supported at *compiler* level) e.g. gcc intel PGI IBM
- Compiler directives are generally simple and easier to use than thread API's
- In general, only moderate speedups can be achieved.as OpenMP codes tend to have serial-only portions,
- Can only really be run on a socket or in shared memory environments
- High Startup costs

Partitioned Global Address Space: (PGAS)

Multiple threads share at least a part of a global address space.

Can access local and remote data with same mechanisms.

Can distinguish between local and remote data with some sort of typing.

Variants:

Co-Array Fortran (CAF)

Unified Parallel C (UPC)

STILL EVOLVING AND NOT WIDELY USED but can do well =>

Strengths:

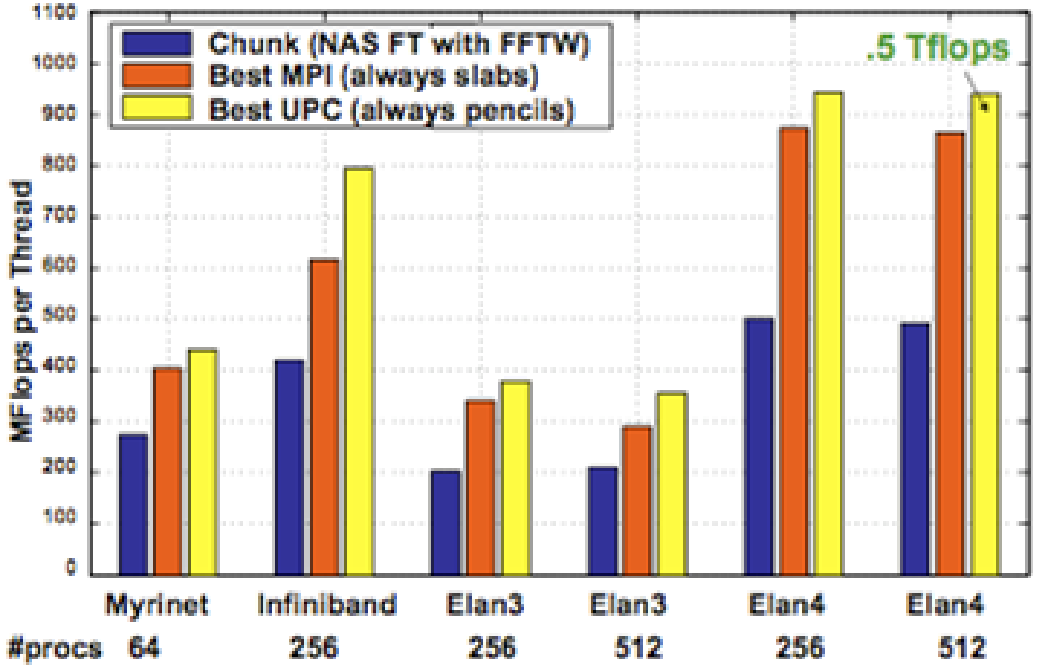
Looks like SMP on a distributed memory machine.

* Currently translates code into an underlying message passing version for efficiency.

Weaknesses:

Immature and depends on * to be efficient.

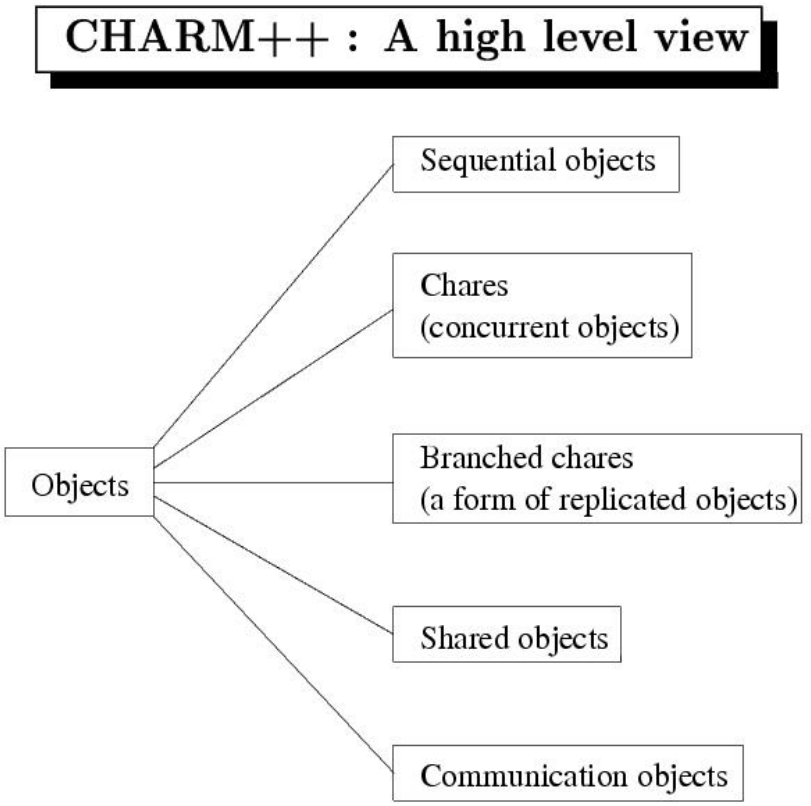
Can easily write lots of expensive remote memory access without paying attention.



Frameworks

One of the more experimental approaches that is gaining some traction is to use a parallel framework that handles the load balancing and messaging while you “fill in” the science. Charm++ is a popular example:

- Charm++
  - Object-oriented parallel extension to C++
  - Run-time engine allows work to be “scheduled” on the computer.
  - Highly-dynamic, extreme load-balancing capabilities.
  - Completely asynchronous.
  - NAMD, a very popular MD simulation engine is written in Charm++



Hybrid Coding

- Problem: given the engineering constraint of a machine made up of a large collection of multi-core processors, how do we use message passing at the wide level while still taking advantage of the local shared memory?

- Solution (at least one): Hybrid Coding.

- As the most useful MP library is MPI, and the most useful SM library is OpenMP, the obvious mix is MPI and OpenMP.

- But, *one must design the MPI layer first, and them apply the OpenMP code at the node level*. The reverse is rarely a viable option.

A Few Coding Hints

- Minimize Eliminate serial sections of code

- Minimize communication overhead
  - Choose algorithms that emphasize nearest neighbor communication
  - Overlap computation and communication with asynchronous communication models if possible
- Dynamic load balancing (at least be aware of issue)
- Minimize I/O and learn how to use parallel I/O
  - Very expensive time wise, so use sparingly (and always binary)
- Choose the right language for the job!
- *Plan out your code beforehand.*
  - *Because the above won't just happen late in development*
  - *Transforming a serial code to parallel is rarely the best strategy*
  - *Consider **stateless functions** as a coding model*

Weak and Strong Scalability:

Problem size n on p cores takes time T(n,p)

Strong Scalability $T(n, p) = T(n,1) / p$

Try to solve the same problem p times more quickly on p cores

Weak Scalability $T(np, p) = T(n,1)$

Solve a problem that is p times as large in the same time on p cores

Theorem

Both weak and strong scalability only if **linear complexity**

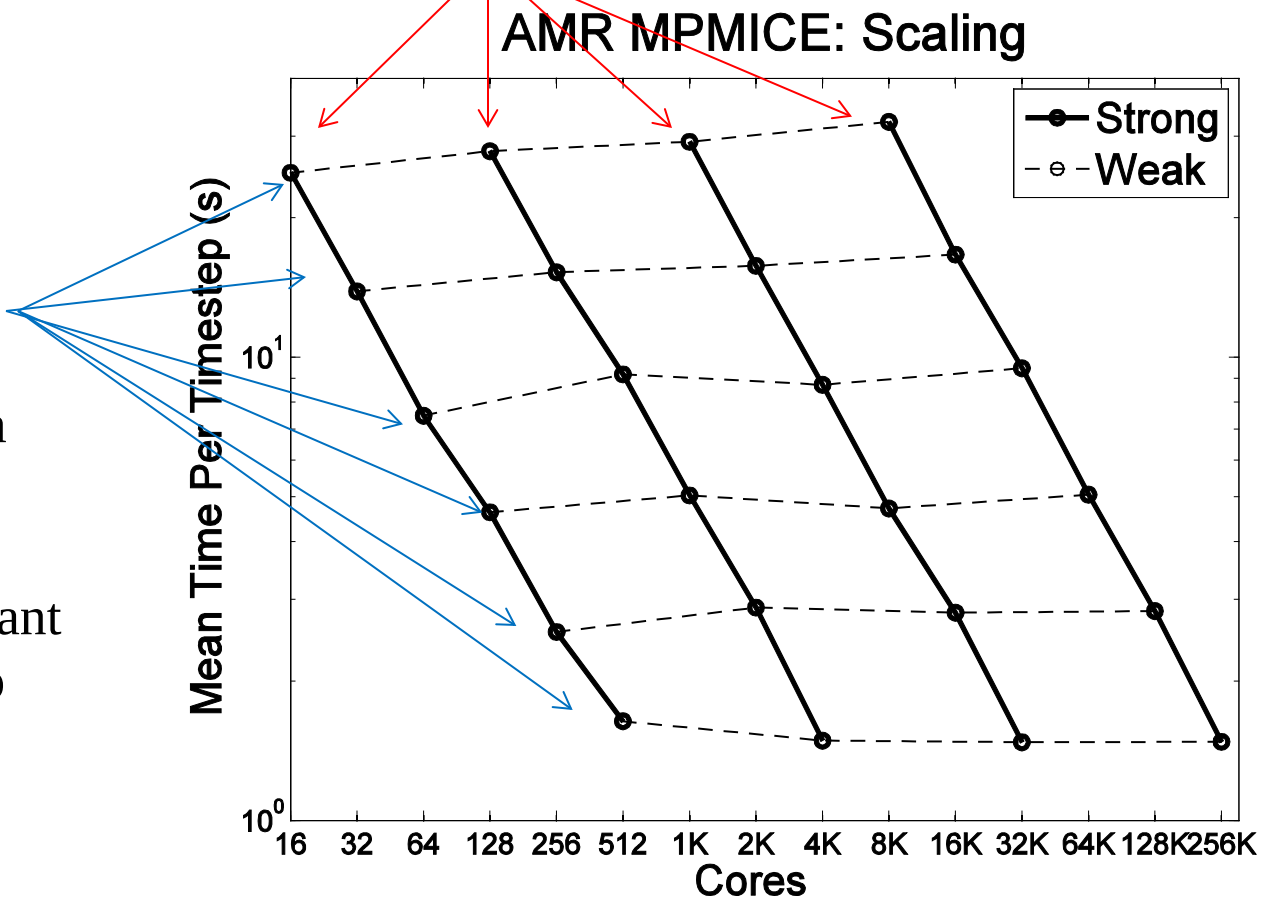
[Tirado + Martin] 1998

$$T(n,1) = \alpha n$$

Weak and Strong Scalability?

Four Strong scaling runs ____ Fixed problem size time should be half when no of cores is doubled

Weak Scaling
6 runs with
Constant Problem
Size per Core
---
should give constant
time per time step



Runs with Uintah on DOEs Titan Machine by Qingyu Meng 2012

WHAT DO WE RUN OUR PARALLEL PROGRAMS ON?

CHPC has a number of parallel machines including the Updraft and Ember systems system

The class will use the Telluride cluster and some of the Turretarch nodes for development

National Resources from NSFs XSEDE network may be available

Cade lab for shared memory

For Research we use the largest NSF and DOE
(soon) DoD machines currently use 3 of the top 8 fastest
Machines in the world

Course Topics and Structure

- Introduction to parallel computing: machines, MPI, performance, etc
- Embarrassingly parallel and synchronous parallel computing
- Partitioning and efficiency
- Sorting linear algebra and image processing in parallel
- OpenMP OPENACC and threads
- Advanced load balancing
- Future of HPC architectures and software

Questions?

COURSE TEXT(s)

- Wilkinson B and Allen M,Parallel Programming: techniques and applications using networked workstations and parallel computers, Prentice Hall, 2005 (Essential but low level)

- B.Chapman G.Jost R Van der Pas. Using OpenMP.MIT Press. 2008 (Useful)

- MPI Parallel Programming with MPI by Peter Pacheco Morgan Kaufmann Publishers Inc.C and Fortran programs available from **http://fawlty.cs.usfca.edu/mpi** (Useful)

- Designing and Building Parallel Programs by Ian Foster Addison-Wesley (1995); A Good online general text.

Message Passing Interface MPI tutorials

- Message passing system MPICH - see /usr/local/mpich portable version of MPI
- http://www-unix.mcs.anl.gov/mpi/mpich/
- See course webpage - includes a guide on how to use the machines
- http://www-unix.mcs.anl.gov/mpi/tutorial/

http://www.mpi.nd.edu/mpi/tutorials/current/

http://www.nas.nasa.gov/Groups/SciCon/Tutorials/MPIintro/

http://www.epcc.ed.ac.uk/epcc-tec/documents/mpi-course/mpi-course.book_2.html

Summary

- Parallel computing makes it possible to solve problems of a size that was previously impossible in times that were hitherto impossible
- Machines are both growing in size in terms of the numbers of cores and numbers of sockets. Power consumption is a real issue.
- Scalability of such systems is challenging at both algorithmic and programming levels.
- Programming for the near future will probably still consist of MPI but coupled to Openmp or some other multi-core programming or accelerator approach at socket level
- Understanding scalable algorithms and programs is a key part of this class