Παράλληλη Επεξεργασία

Εαρινό Εξάμηνο 2025-26
“Εισαγωγή”

Παναγιώτης Χατζηδούκας, Ευάγγελος Δερματάς

Teaching Quote

“Tell me and I forget, teach me and I may remember, involve me and I learn.”

Benjamin Franklin

“Those who can, do; those who can’t, teach”

From George Bernard Shaw's Man and Superman

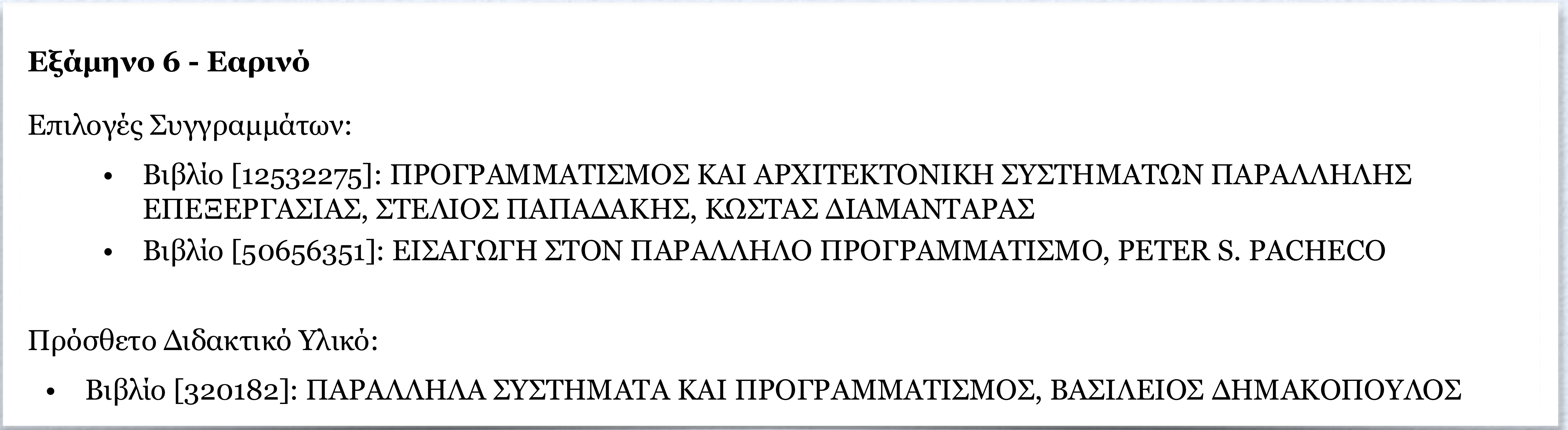
OUTLINE

- Class Information
- What and Why High Performance Computing (HPC) ?
- Overview of
  - Hardware for HPC
  - Performance
  - Applications
  - Programming models (what we will see)

Course schedule (tentative)

24.2 – Introduction + Recap
03.3 – Advanced Multithreading
10.3 – OpenMP (nested parallelism, tasks)
17.3 – MPI I (intro)
24.3 – MPI II (asynchronous communication)
31.3 – MPI III (hybrid model, parallel I/O)
07.4 –
14.4 –
21.4 – SIMD, Linear Algebra
28.4 – CUDA (intro)
05.5 – CUDA (streams)
12.5 – CUDA
19.5 – Tools + Applications
26.5 – Recap

Books



Εξάμηνο 6 - Εαρινό

Επιλογές Συγγραμμάτων:

- Βιβλίο [12532275]: ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΚΑΙ ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΣΥΣΤΗΜΑΤΩΝ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ, ΣΤΕΛΙΟΣ ΠΑΠΑΔΑΚΗΣ, ΚΩΣΤΑΣ ΔΙΑΜΑΝΤΑΡΑΣ
- Βιβλίο [50656351]: ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΠΑΡΑΛΛΗΛΟ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ, PETER S. PACHECO

Πρόσθετο Διδακτικό Υλικό:

- Βιβλίο [320182]: ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ ΚΑΙ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ, ΒΑΣΙΛΕΙΟΣ ΔΗΜΑΚΟΠΟΥΛΟΣ

READING MATERIAL

- See Class Website
- Parallel Programming for Science and Engineering, Victor Eijkhout:
  - https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/EijkhoutParallelProgramming.pdf
- Introduction to Parallel Programming:
  - http://www-users.cs.umn.edu/~karypis/parbook/

Class Website

https://eclass.upatras.gr/courses/CEID1057/

Exercises

- Goal of the exercises are to help you become familiar with the material discussed in the class
- Exercise session:
    - solution of the previous exercise: discussion and feedback
    - introduction of the new exercise
- The solution of the exercises will be available one week after their introduction

Exam

GRADING:

MAX(EXAM**, 0.8***EXAM + **0.2***PROJECT) *all* **(2007 -)**

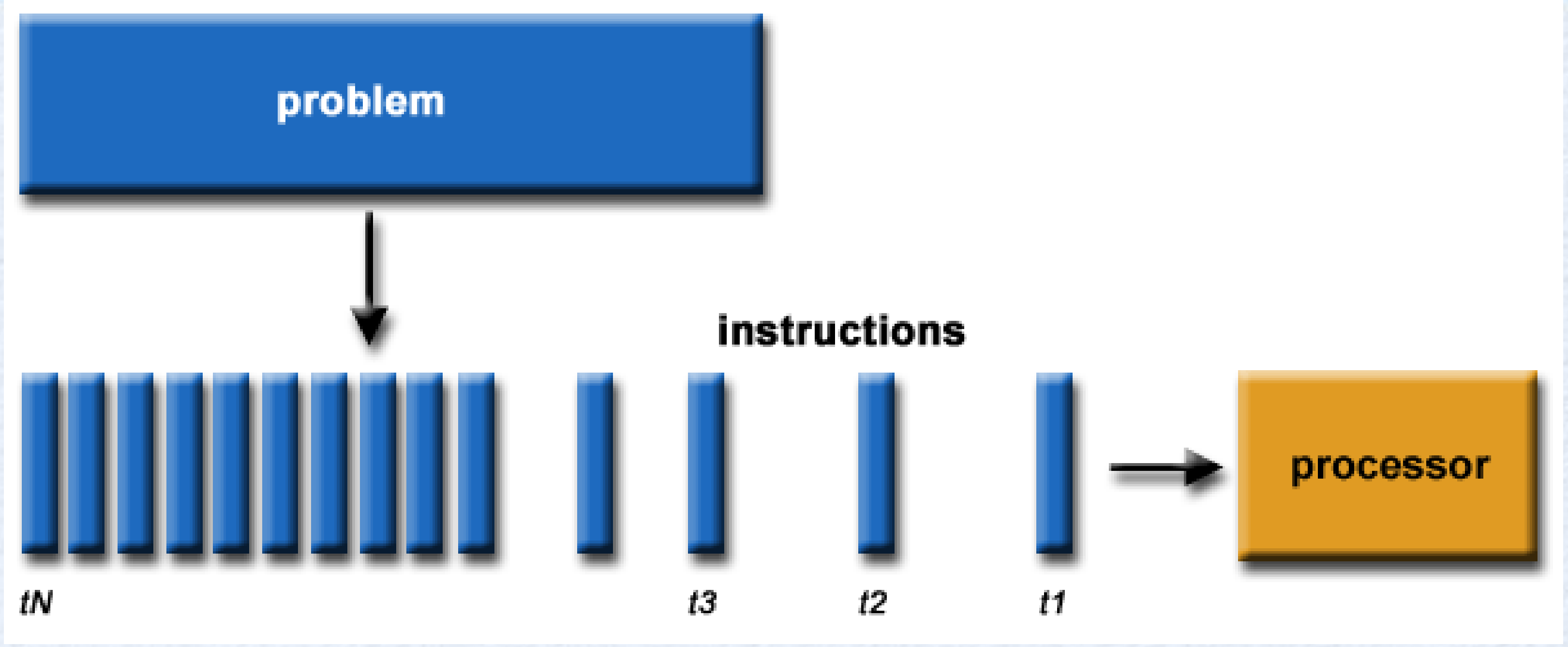
Why ~~powerful~~ computers are parallel

circa 1991-2006

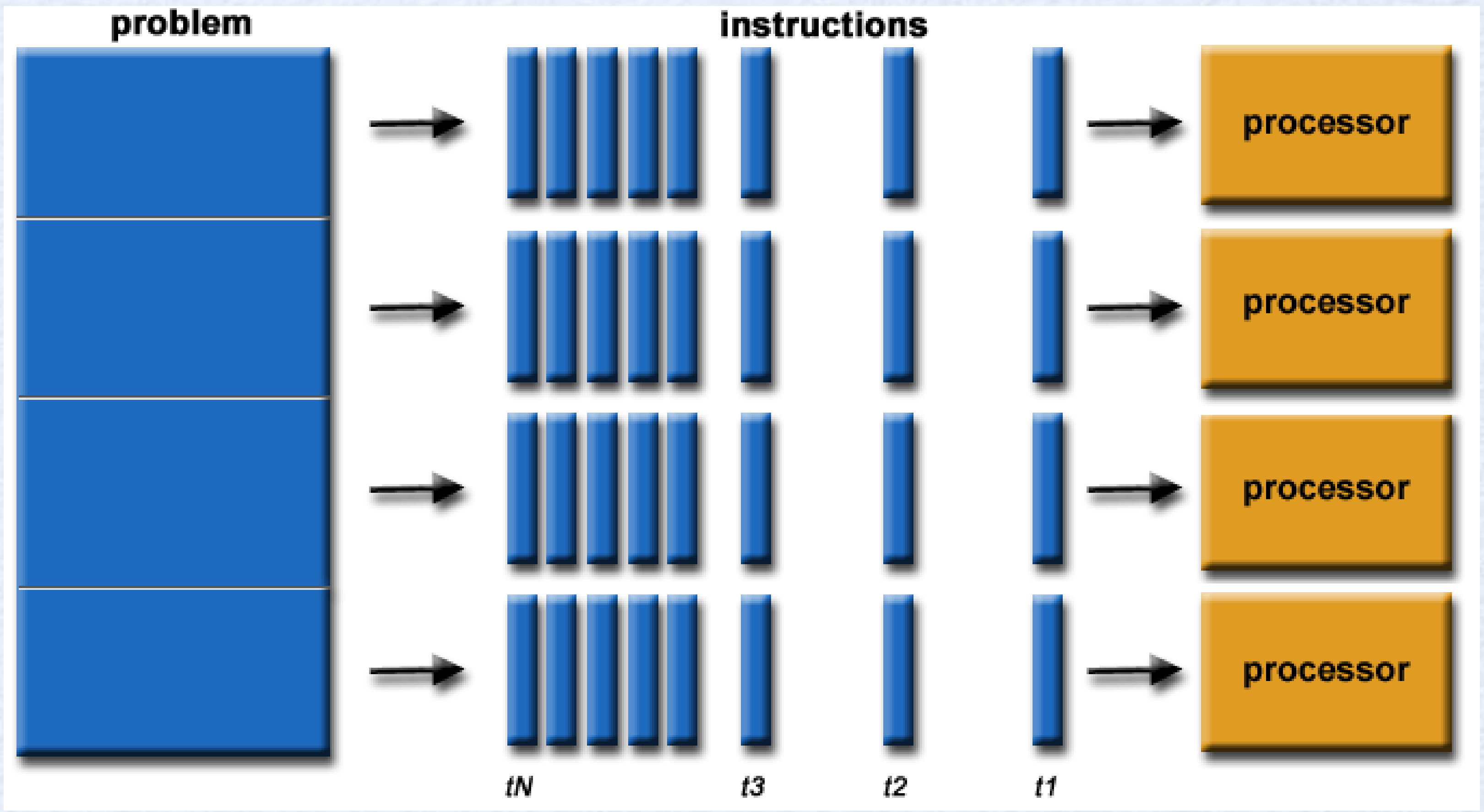
CREDIT: J. Demmel

Parallel Computing

- Serial Computing:
  - Problem has a series of instructions
  - Executed sequentially, one at a time



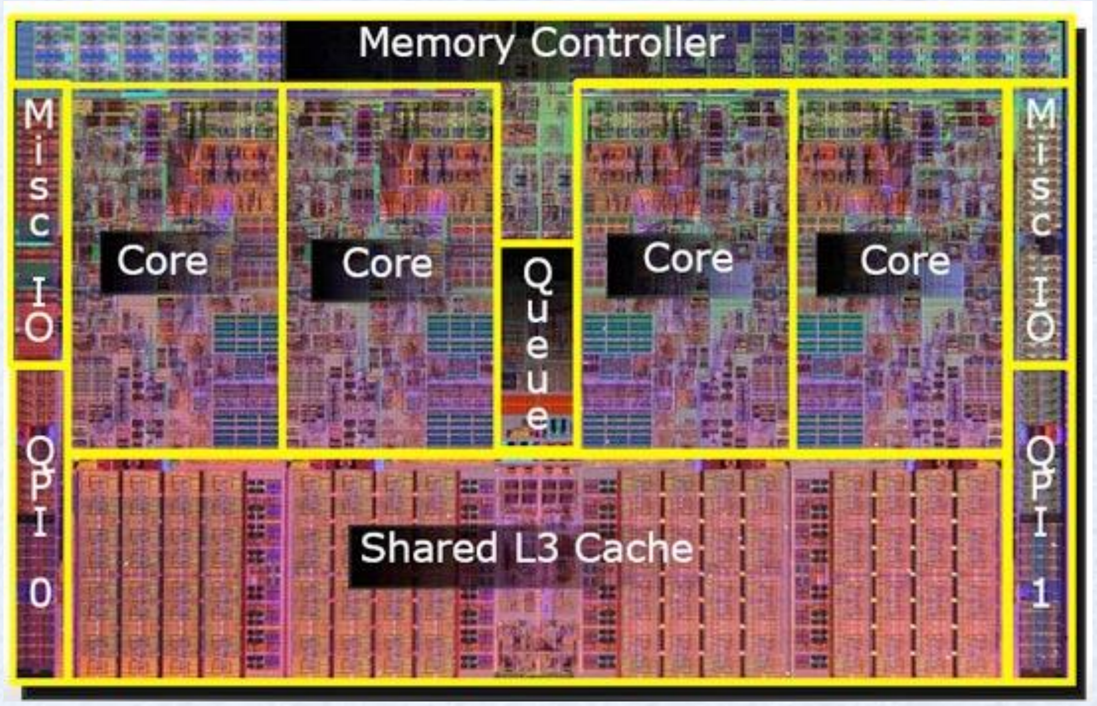
- Parallel Computing
  - Problem is split into sub-problems that can be solved **concurrently**
  - Each subproblem runs sequentially (as above) on a separate machine/processor
  - Some control/coordination mechanism needed



Computer layout (in a nutshell)

- CPU
  - does the computations
  - contains **multiple cores** (usually)
    - each core works mostly independently,
      copy of a single core with global coordination
  - contains several levels of caches to speed up reading/writing to memory (very relevant for high performance computing)
- Memory
  - stores data for computations
  - **shared** among the cores of the CPU (or multiple CPUs in a compute node)
- Network: connect compute nodes and connect to outer world
- Input/Output: displays, hard-drives, etc

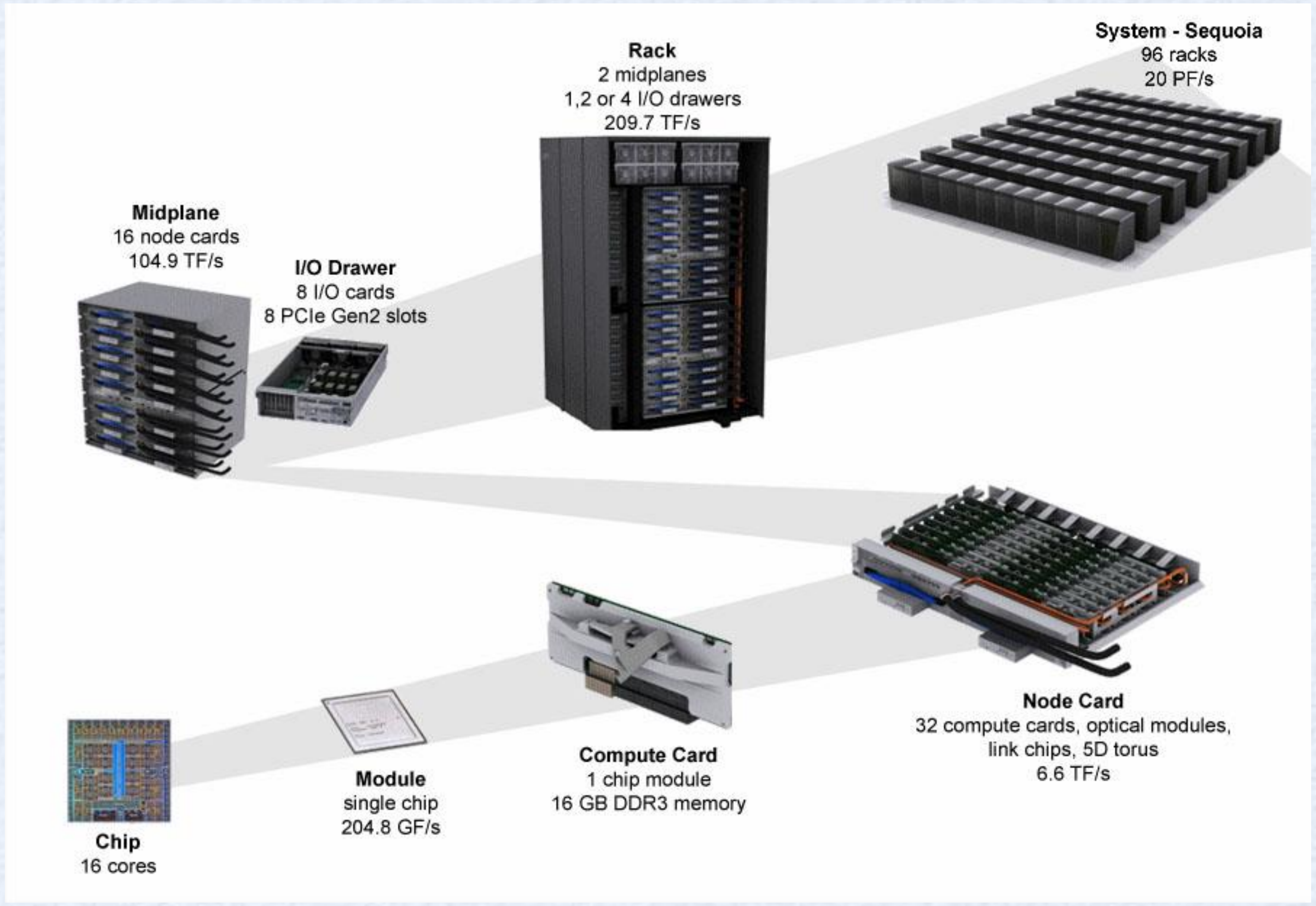
Intel Core i7 CPU



picture source: legitreviews

Massively Parallel Computing

Sequoia IBM BlueGene/Q supercomputer (at Lawrence Livermore National Laboratory)



source:computing.llnl.gov

Components of a Supercomputer (roughly)

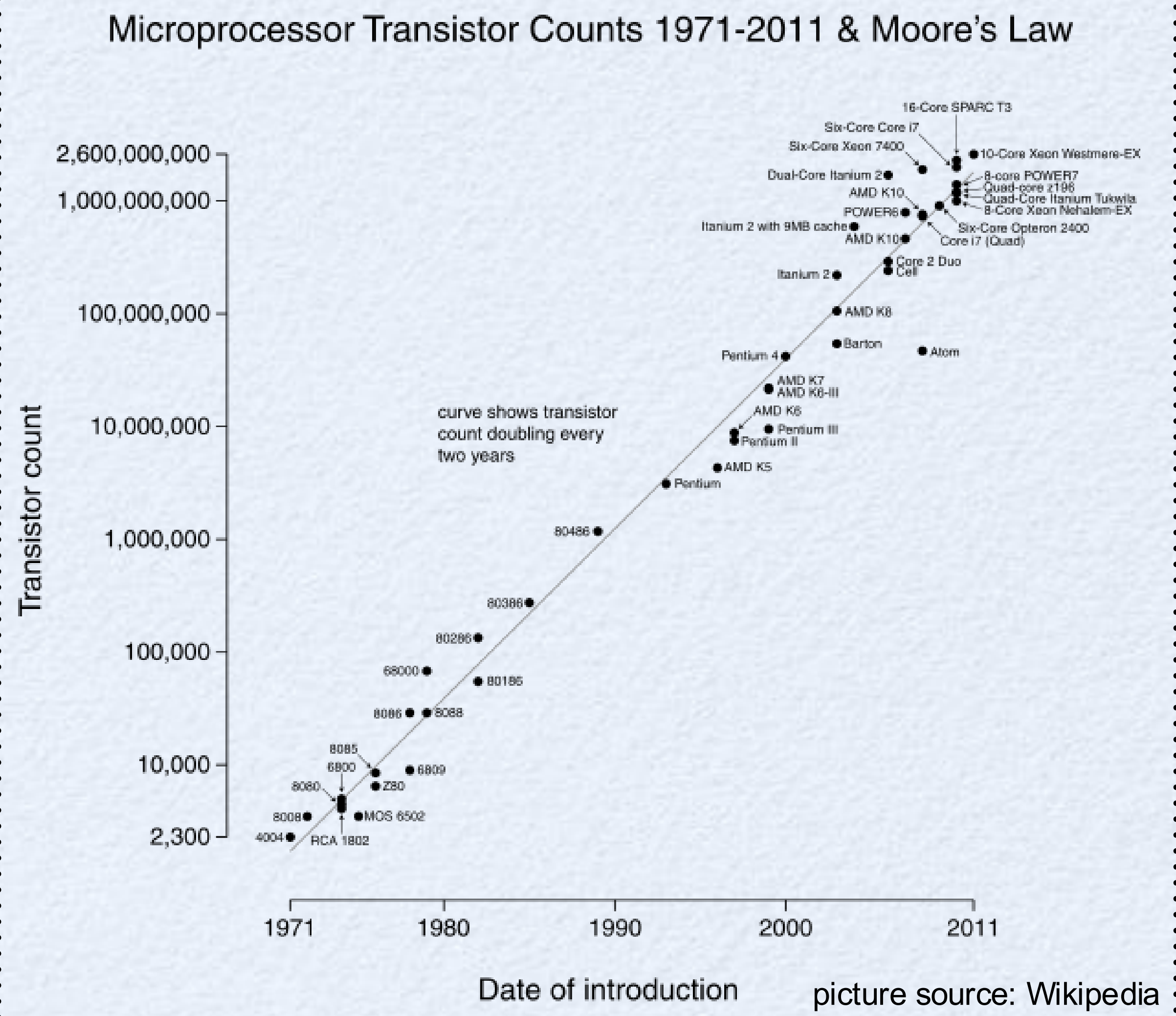
- Processors (CPUs) <= *note that those already contain multiple cores*
- Compute Node: collection of CPUs with a shared memory
  - nodes may also have “accelerators” like graphical processing units (GPU)
- Cluster: collection of nodes connected with a (very fast) network

Why Parallel Computing?

- PAST : Parallel Computing = High End Science
- TODAY : Parallel Computing = Everyday (industry, academia) Computing
- Advantages
  - Save time and money - shorten time to completion
  - Solve bigger problems and process more data
  - Exploit concurrency - many things at the same time
  - Use of non-local resources

Why Parallel Computing?

- Moore's law (1965)
  - **observation**: number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years
  - still ongoing...BUT: increase comes by having more and **more cores per CPU**
- Before (roughly) 2002:
  - more performance for free,
    clock rates increased,
    cores got faster
- Now:
  - observed gap between attained performance and possible one
  - **need to use parallel computing**
- Wirth's law (1995)
  - "software is getting slower more rapidly than hardware becomes faster"

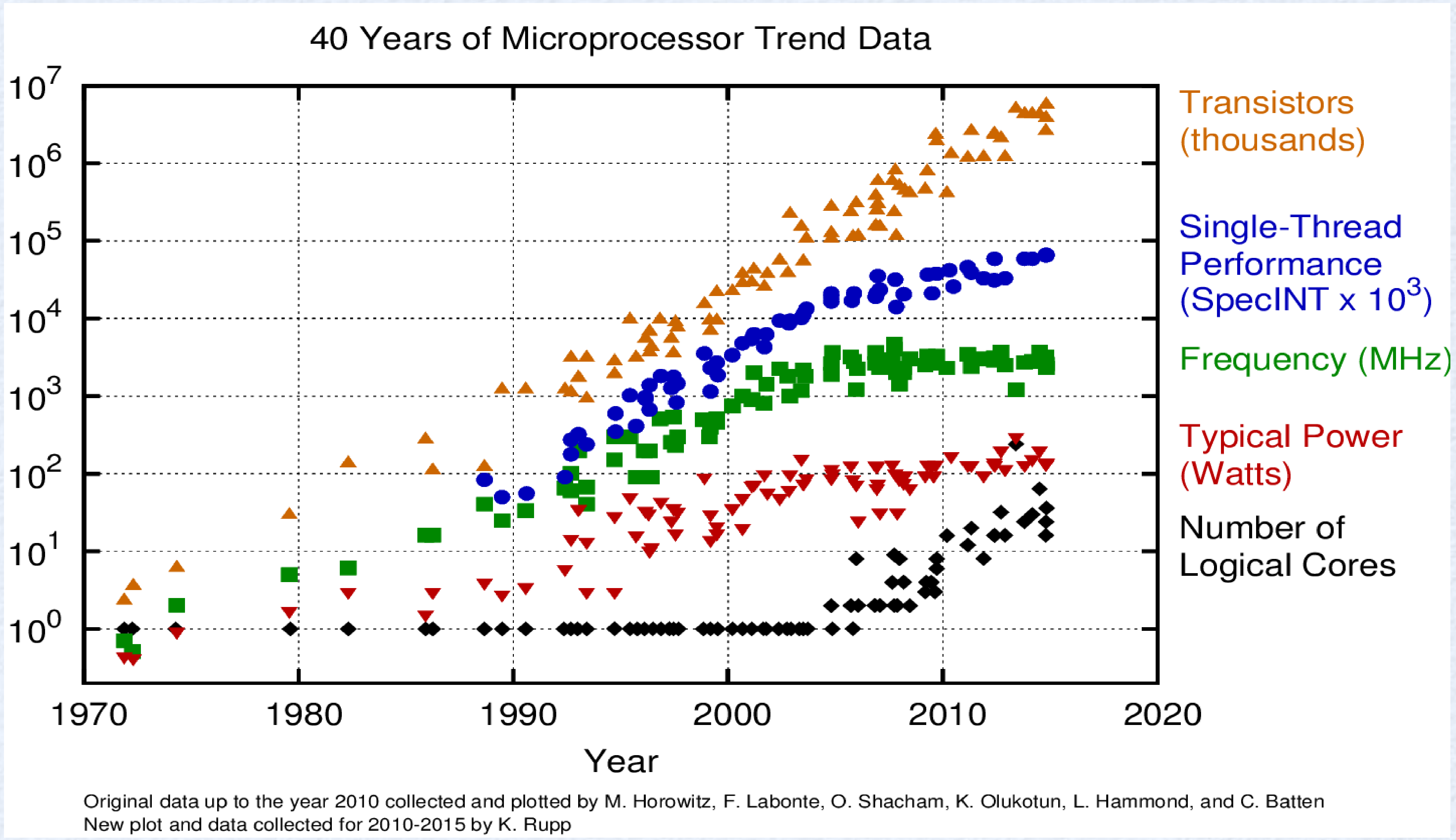


picture source: Wikipedia

Why Parallel Computing?

- **Physical/Practical constraints for even faster serial computers:**
  - **Transmission speeds** - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
  - **Limits to miniaturization** - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
  - **Economic limitations** - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.
  - **Energy Limits** - limits imposed by cooling needs for chips and supercomputers → hitting the power wall

Revolution in Processors



- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
- Number of processor cores may double instead
- Power is under control, no longer growing

Multicore Era

- The sea change: since 2006 all microprocessor companies are shipping computers with multiple cores per chip
- New Moore's Law: double the number of cores per microprocessor per semi-conductor technology generation every two years

$$\begin{tikzcd}
\text{OLD} \arrow[r] & \text{Hardware, Architecture, Compilers} \arrow[r] & \text{Speed} \\
\text{NEW} \arrow[r] & \text{Parallel programming} \arrow[r] & \text{Speed}
\end{tikzcd}$$

- Need to deal with systems with millions of concurrent threads
- Need to deal with inter-chip parallelism as well as intra-chip parallelism

Parallelism today ?

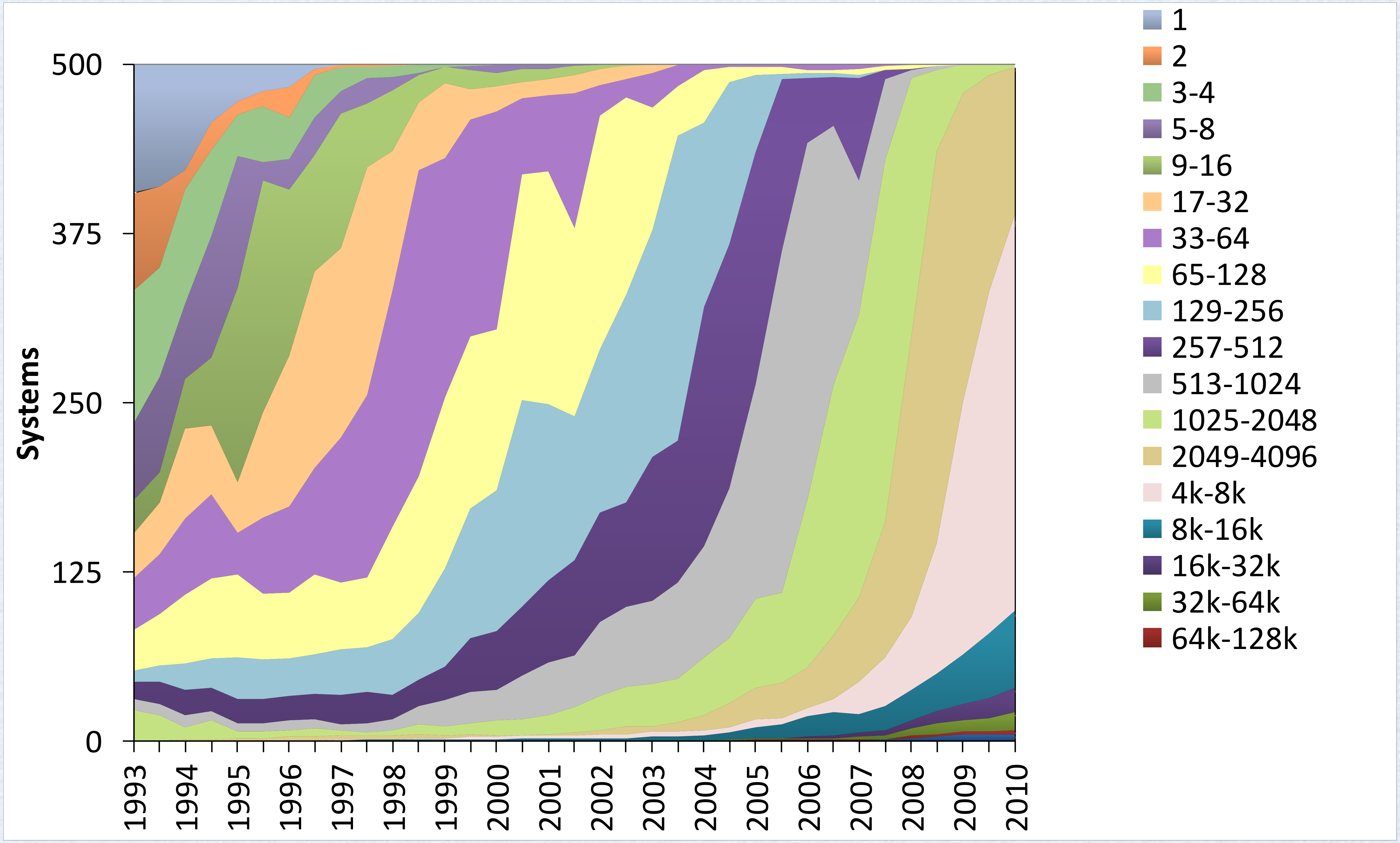
- All major processor vendors are producing *multicore* chips
  - Every machine is practically a parallel machine
  - To keep doubling performance, parallelism must double
- Which (commercial) applications can use this parallelism?
  - Do they have to be rewritten from scratch?
- Will all programmers have to be parallel programmers?
  - New software model needed
  - Try to hide complexity from most programmers – eventually
  - In the meantime, need to understand it
- Computer industry betting on this big change, but does not have all the answers

CREDIT: J. Demmel

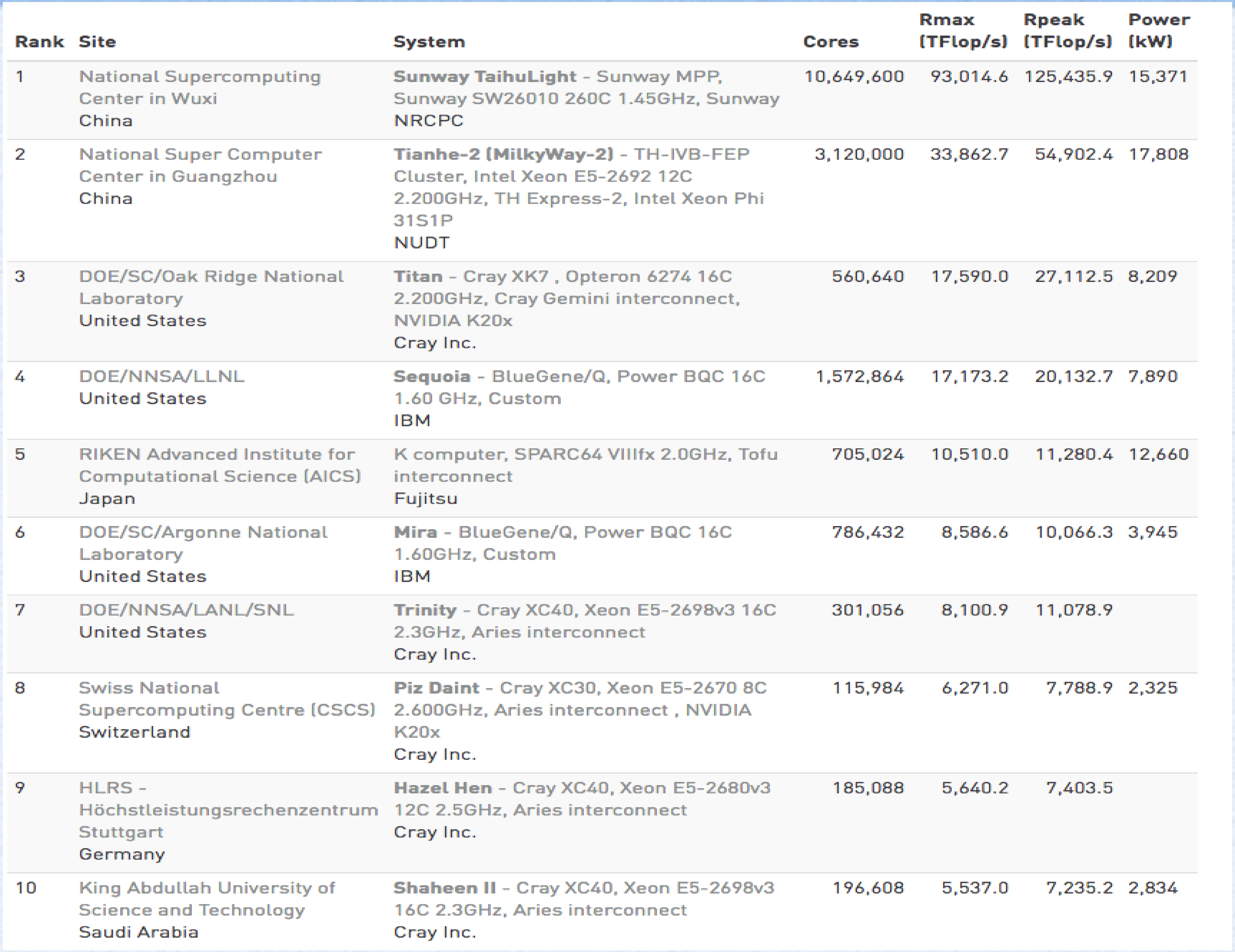
The TOP500 Project

- Listing the 500 most powerful computers in the world
- Yardstick: performance (Rmax) of Linpack
  - Solve Ax=b, dense problem, matrix is random
  - Dominated by dense matrix-matrix multiply
- Updated twice a year:
  - ISC'xy in June in Germany
  - SCxy in November in the U.S.
- TOP500 web site at: www.top500.org

Core Count

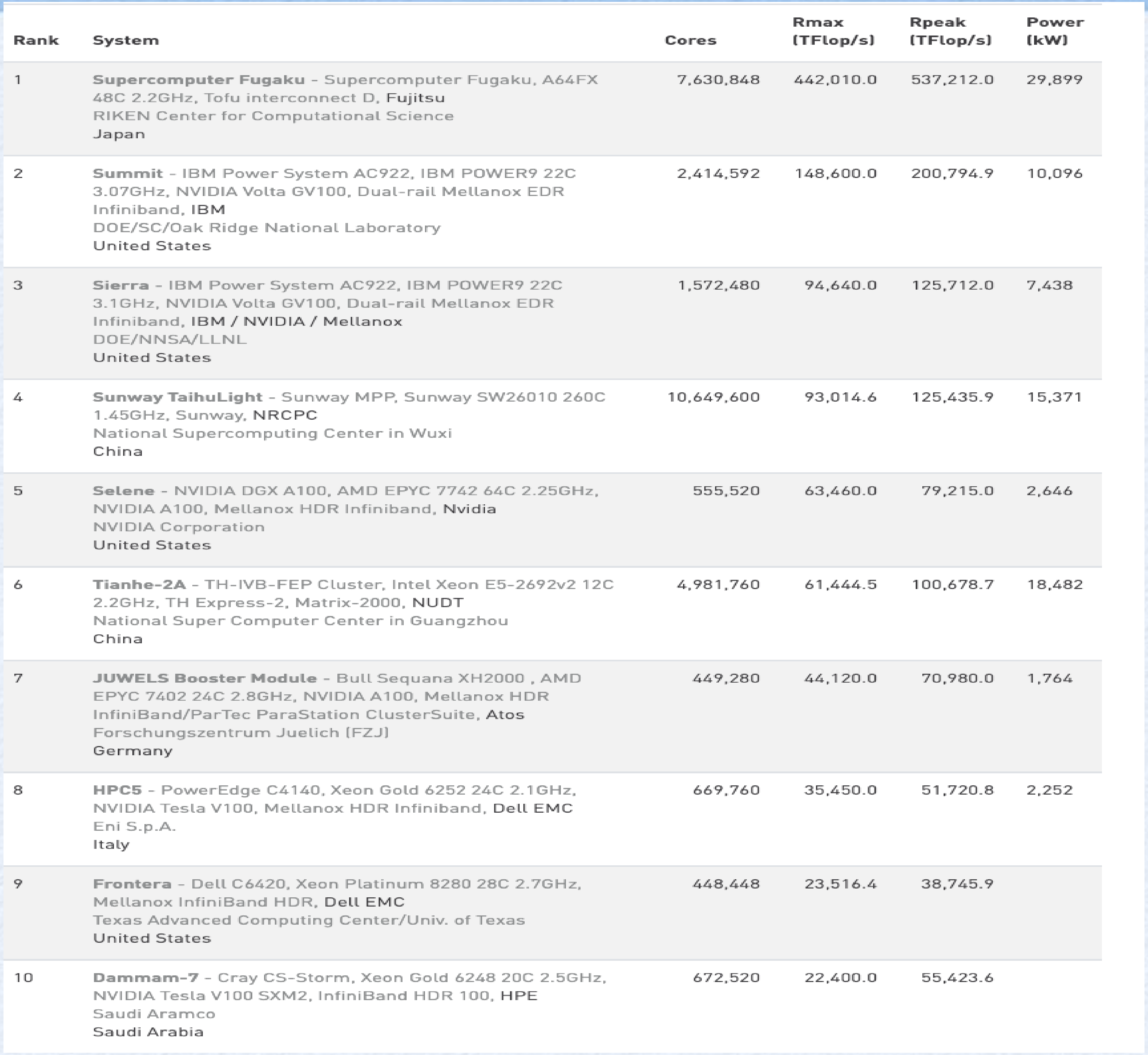

500
375
250
125
0
Systems
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
1
2
3-4
5-8
9-16
17-32
33-64
65-128
129-256
257-512
513-1024
1025-2048
2049-4096
4k-8k
8k-16k
16k-32k
32k-64k
64k-128k

TOP500 - June 2016



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
6	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
7	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	301,056	8,100.9	11,078.9	
8	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
9	HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	**Hazel Hen** - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.	185,088	5,640.2	7,403.5	
10	King Abdullah University of Science and Technology Saudi Arabia	**Shaheen II** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	196,608	5,537.0	7,235.2	2,834

TOP500 - June 2020



Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
10	**Dammam-7** - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	

Units of Measure

- High Performance Computing (HPC) units are:
  - **Flop:** floating point operation, usually double precision unless noted
  - **Flop/s:** floating point operations per second
  - **Bytes:** size of data (a double precision floating point number is 8 bytes)

- Typical sizes are millions, billions, trillions…

Mega	MFlop/s = 10^6 flop/sec	MByte = 2^20 ~ 10^6 bytes
Giga	GFlop/s = 10^9 flop/sec	GByte = 2^30 ~ 10^9 bytes
Tera	**TFlop/s = 10^12 flop/sec**	**TByte = 2^40 ~ 10^12 bytes**
Peta	**PFlop/s = 10^15 flop/sec**	**PByte = 2^50 ~ 10^15 bytes**
Exa	**EFlop/s = 10^18 flop/sec**	**EByte = 2^60 ~ 10^18 bytes**
Zetta	ZFlop/s = 10^21 flop/sec	ZByte = 2^70 ~ 10^21 bytes

- Current fastest (public) machine: ~ 1194/1679 PFlop/s, 8.7M cores

How Rpeak is computed

- Rpeak = Nominal Peak Performance (**PP**)
- **PP** [Flop/s] = **f** [Hz = cycles/s] x **c** [Flop/cycle] x **v** [-] x **n** [-]
  - **f** : core frequency in CPU cycles per second
  - **c** : how many Flops per cycle
  - **v** : SIMD width in number of doubles (or floats)
  - **n** : # cores

These features can be found at the hardware specifications

- Example: IBM BGQ chip (one compute node)
  - **f** = 1.6 GHz, **c** = 2 (supports FMA), **v** = 4, **n** = 16
  - **PP** = 1.6 * 2 * 4 * 16 GFlop/s = 204.8 GFlop/s

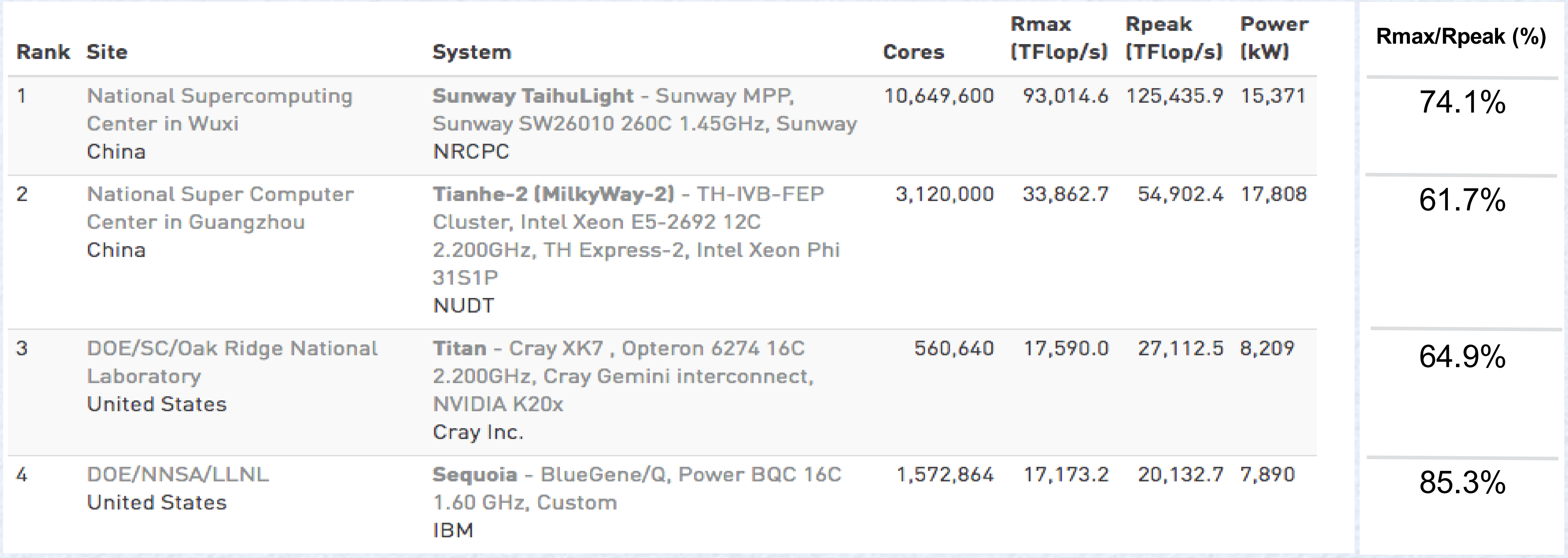
FMA (fused multiply-add): a*b+c in one step

- BGQ Rack (1024 nodes): 1024*204.8 = 209715.2 GFlop/s = 209.7 TFlop/s
- IBM Sequoia @ LLNL (96 racks): 96 * 209.7 TFlop/s = 20.13 PFlop/s

How Rmax is computed

- Application Performance
  - How many FP operations the application performs
  - Execution time (in seconds)
  - Fraction of the peak = Attained/Nominal performance

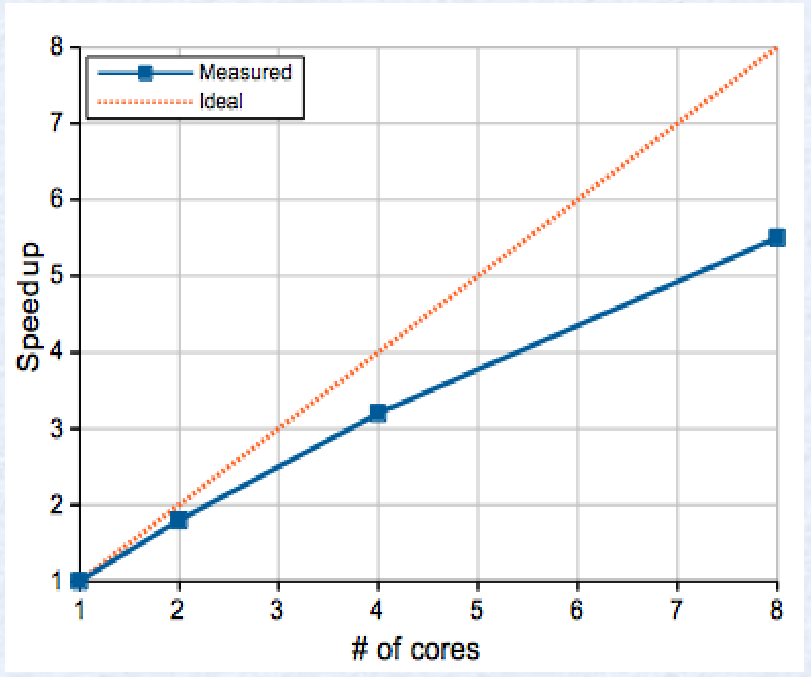
- In many cases, FP operations can be replaced with INT operations, interactions, transactions, etc. (per second)

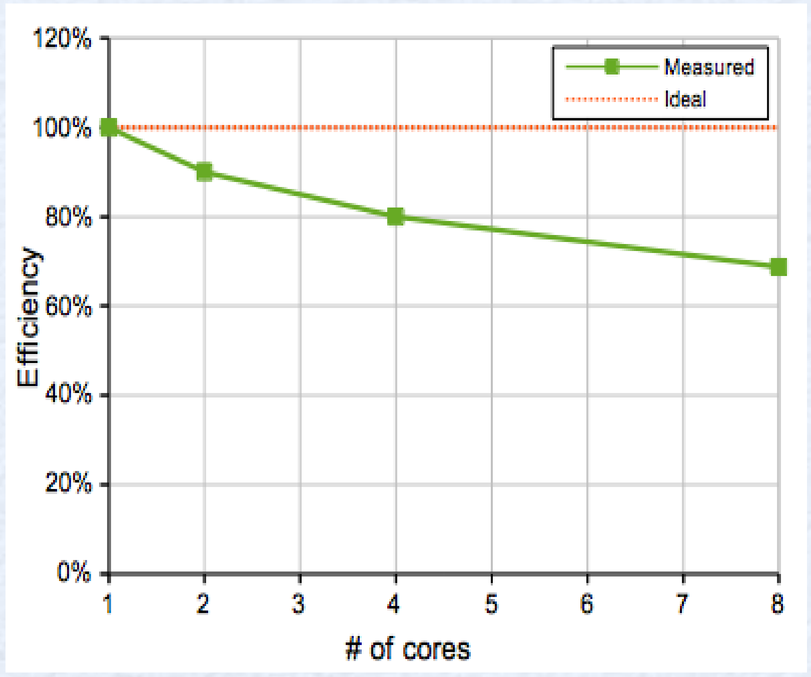


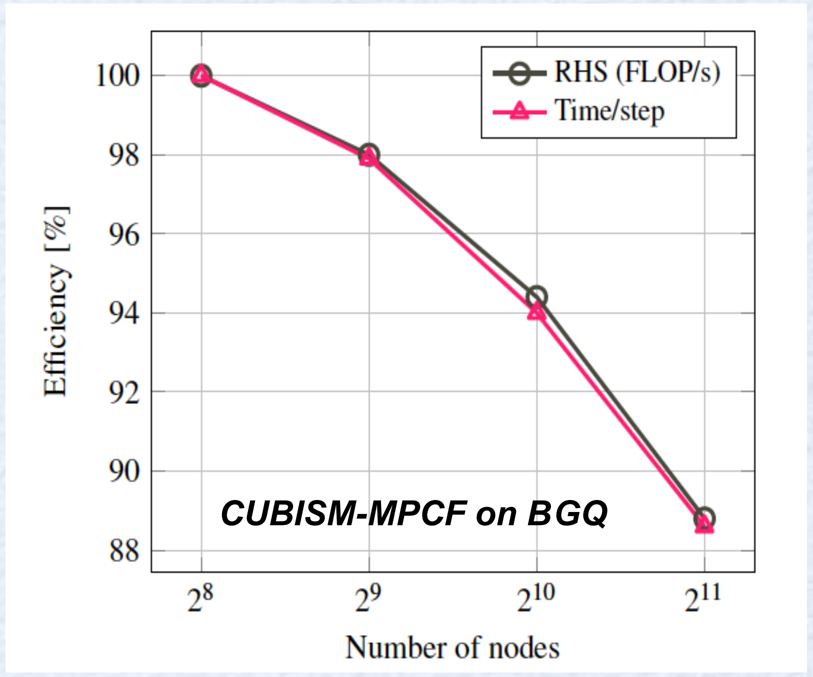
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)	Rmax/Rpeak (%)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371	74.1%
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808	61.7%
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209	64.9%
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890	85.3%

Performance Metrics

- Time to solution: solve a problem as fast as possible
- T(p): execution time on p processors
- Speedup(p) = T(1)/T(p)
- Strong scaling: keep the problem size constant as you increase the number of CPU cores p
- Strong scaling Efficiency(p) = Speedup(p)/p (x100%)

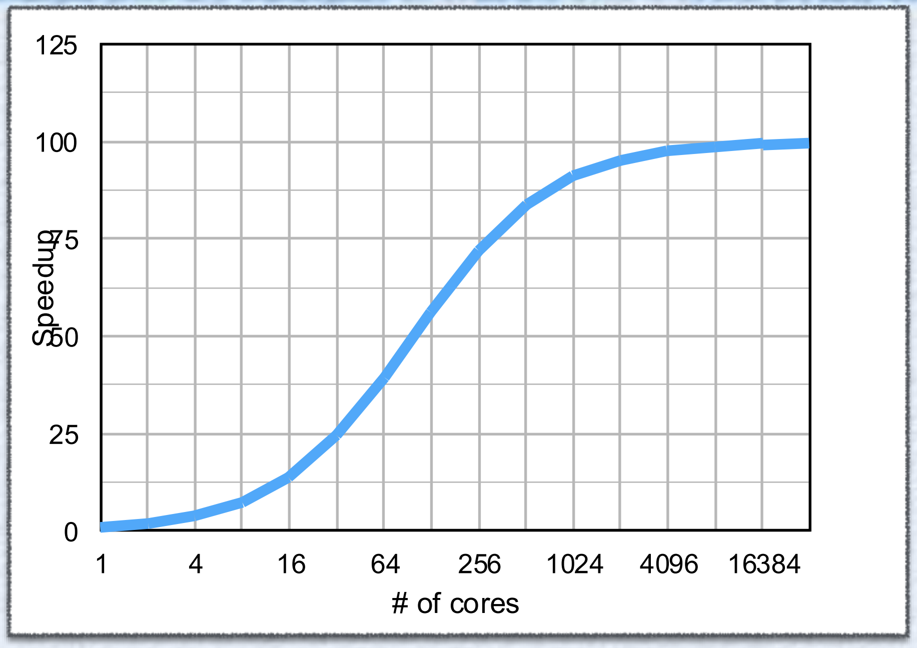




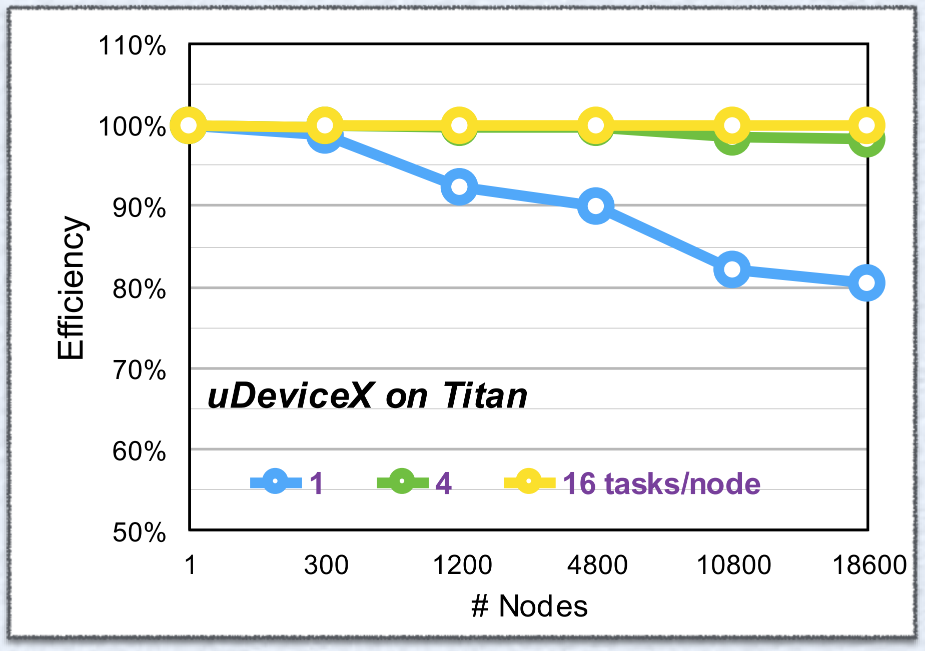


Performance Metrics

- Problem of strong scaling: speedup is limited by the serial fraction ***s*** of the code (Amdahl's Law)
  - s = 1% → max speedup = 100



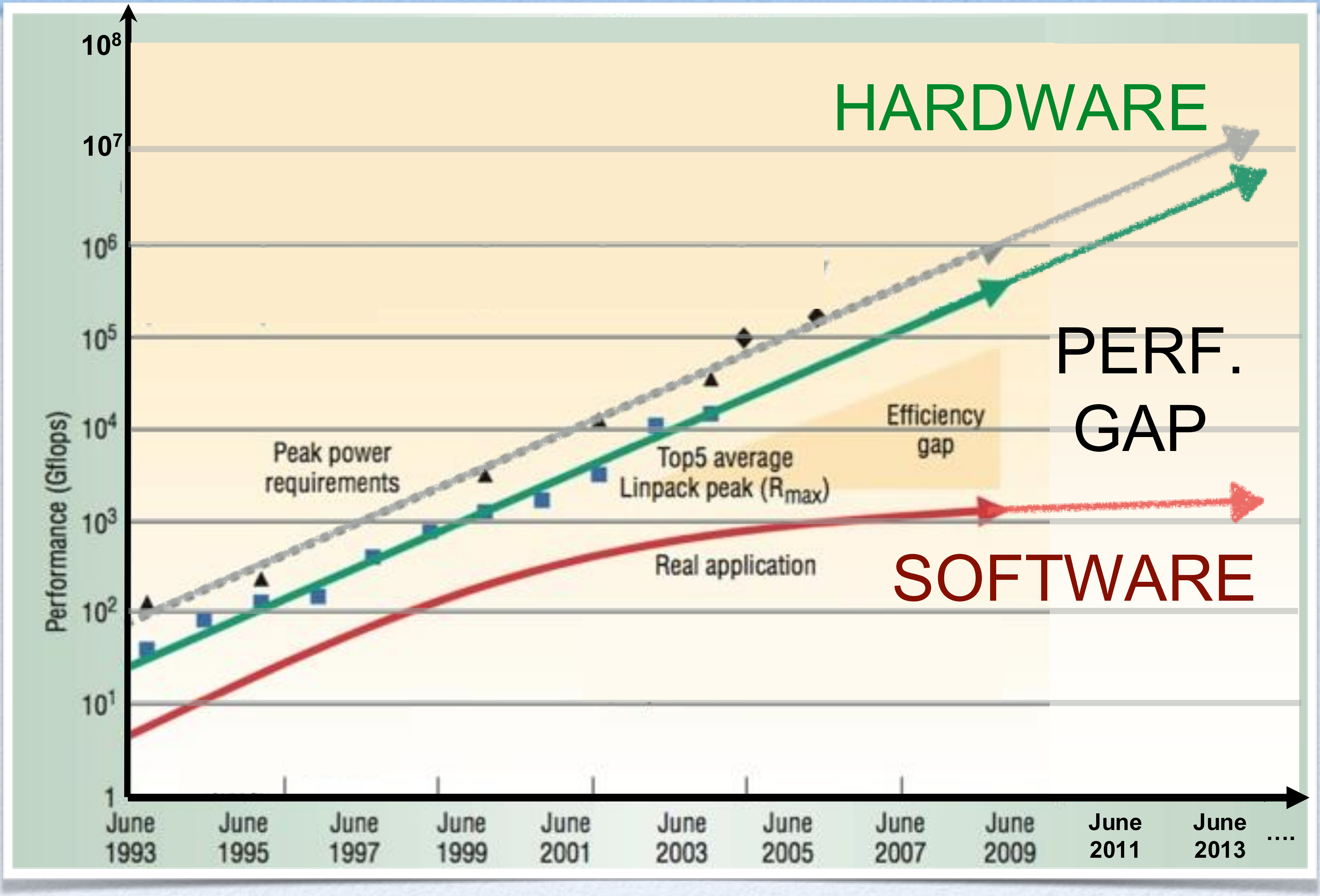
- Weak scaling: constant work per core
  - increase the problem size with the number of CPU cores p
- Efficiency(p) = T(p)/T(1) (x100%)
  - How well you can solve bigger problems



- ACM Gordon Bell Prize (every year at the SC conference): "awarded for **peak performance** or special achievements in **scalability** and **time-to-solution** on **important science and engineering problems**"

source: http://awards.acm.org/bell/nominations.cfm

Performance GAP



Cameron et al, IEEE Computer 2005

~99% of SOFTWARE uses < 10 % of HARDWARE

HPC across Science/Technology

Can we find widely used patterns ?

Common patterns of communication and computation

1. Embedded Computing (EEMBC benchmark)
2. Desktop/Server Computing (SPEC2006)
3. Database / Text Mining Software
4. Games/Graphics/Vision
5. Machine Learning
6. **High Performance Computing (Original “7 Dwarfs”)**

- Result: 13 scientific kernels or “Dwarfs”

CREDIT: J. Demmel

Common Patterns = “Dwarfs” (Collela)

“Dwarfs”	Embed	SPEC	DB	Games	ML	HPC
1. Finite State Mach.	🟥	🟥	🟥	🟨	🟨	🟦
2. Combinational	🟥	🟦	🟩	🟦	🟩	🟦
3. Graph Traversal	🟥	🟨	🟨	🟨	🟥	🟦
4. Structured Grid	🟥	🟥	🟦	🟨	🟦	🟥
5. Dense Matrix	🟥	🟥	🟨	🟥	🟥	🟥
6. Sparse Matrix	🟨	🟨	🟦	🟥	🟥	🟥
7. Spectral (FFT)	🟨	🟦	🟦	🟨	🟨	🟥
8. Dynamic Progr.	🟨	🟦	🟥	🟦	🟥	🟦
9. N-Body	🟦	🟨	🟦	🟨	🟦	🟥
10. MapReduce	🟦	🟩	🟥	🟦	🟥	🟥
11. Backtrack/ B&B	🟦	🟦	🟨	🟦	🟥	🟦
12. Graphical Models	🟦	🟦	🟨	🟦	🟥	🟦
13. Unstructured Grid	🟦	🟦	🟦	🟨	🟨	🟥

Dwarf Popularity (Red Hot → Blue Cool)

CREDIT: J. Demmel

Why writing fast parallel programs is hard

- Essential to know the hardware to get the best out of software

- KEY ISSUE: Understand in this context alternatives between algorithms

Principles of Parallel Computing

- Finding enough parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning

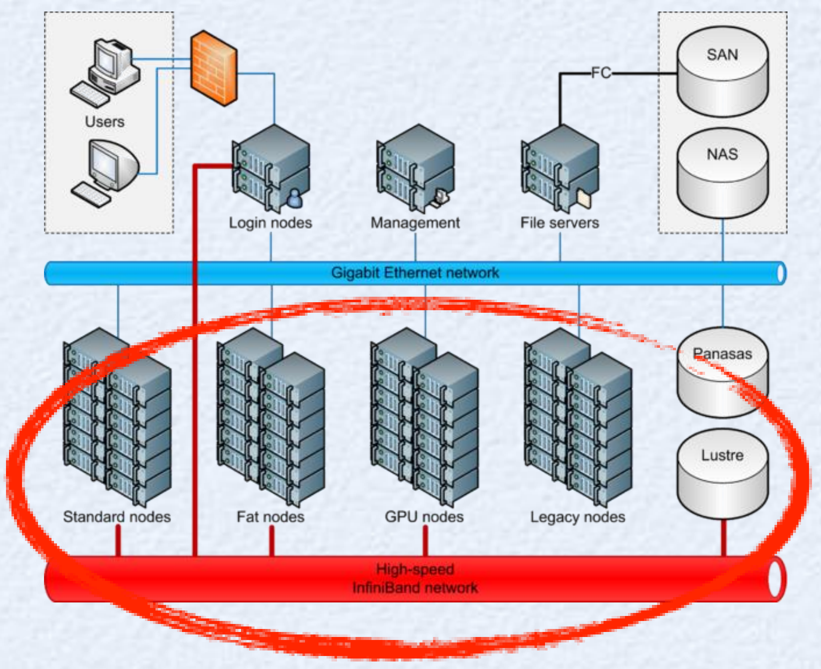
All of these things make parallel programming even harder than sequential programming.

Parallel Software Eventually

- 2 types of programmers ➔ 2 layers of software
- **Efficiency Layer** (10% of programmers)
  - Expert programmers build Libraries implementing kernels, “Frameworks”, OS, ….
  - Highest fraction of peak performance possible
- **Productivity Layer** (90% of programmers)
  - Domain experts / Non-expert programmers productively build parallel applications by composing frameworks & libraries
  - Hide as many details of machine, parallelism as possible
  - Willing to sacrifice some performance for productive programming

(credit: J. Demmel)

Anatomy of a Cluster



Cluster: network of nodes

Distributed memory

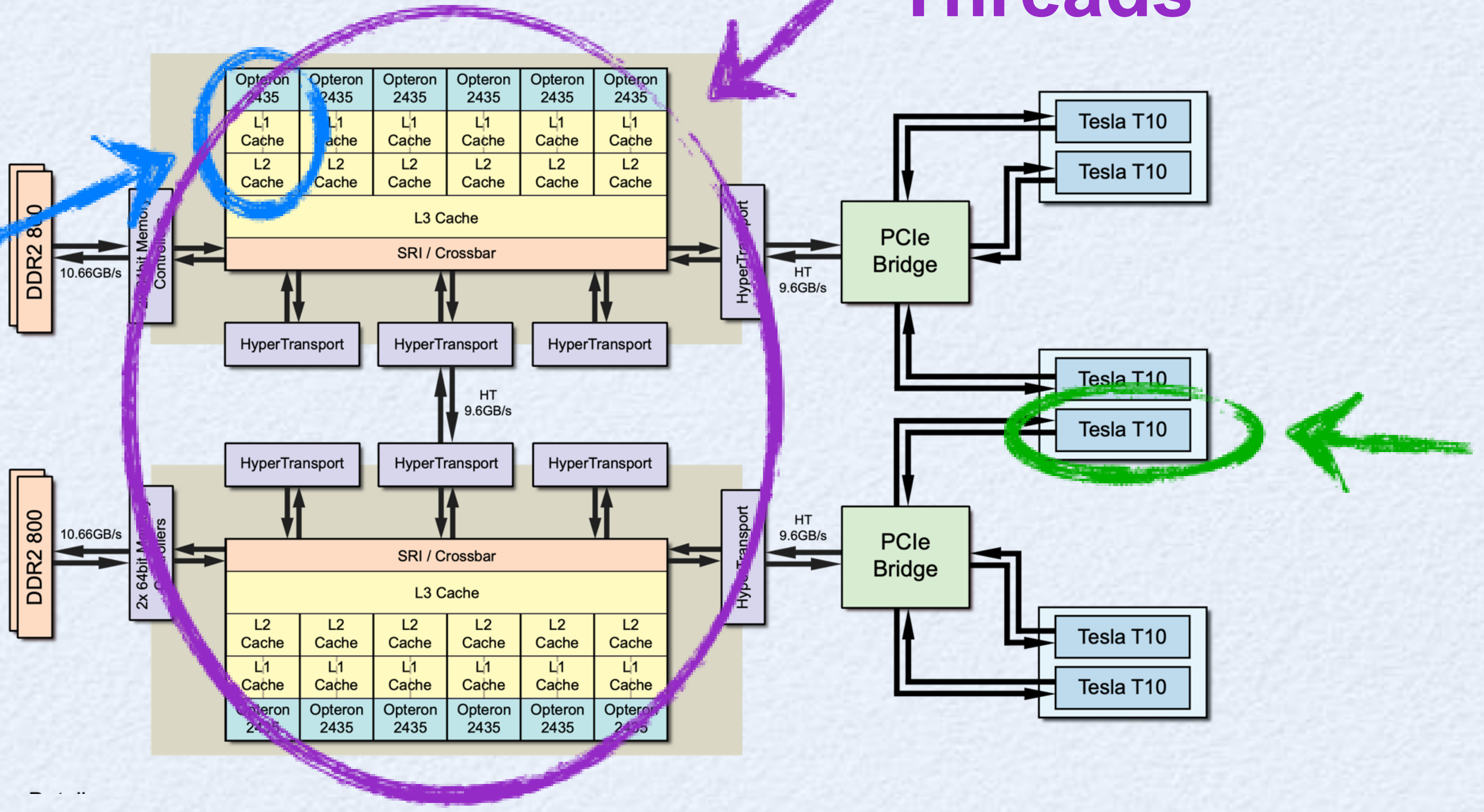
MPI

Node: multiple processors

Shared Memory

OpenMP, C++/POSIX

Threads



CPU core

C/C++,

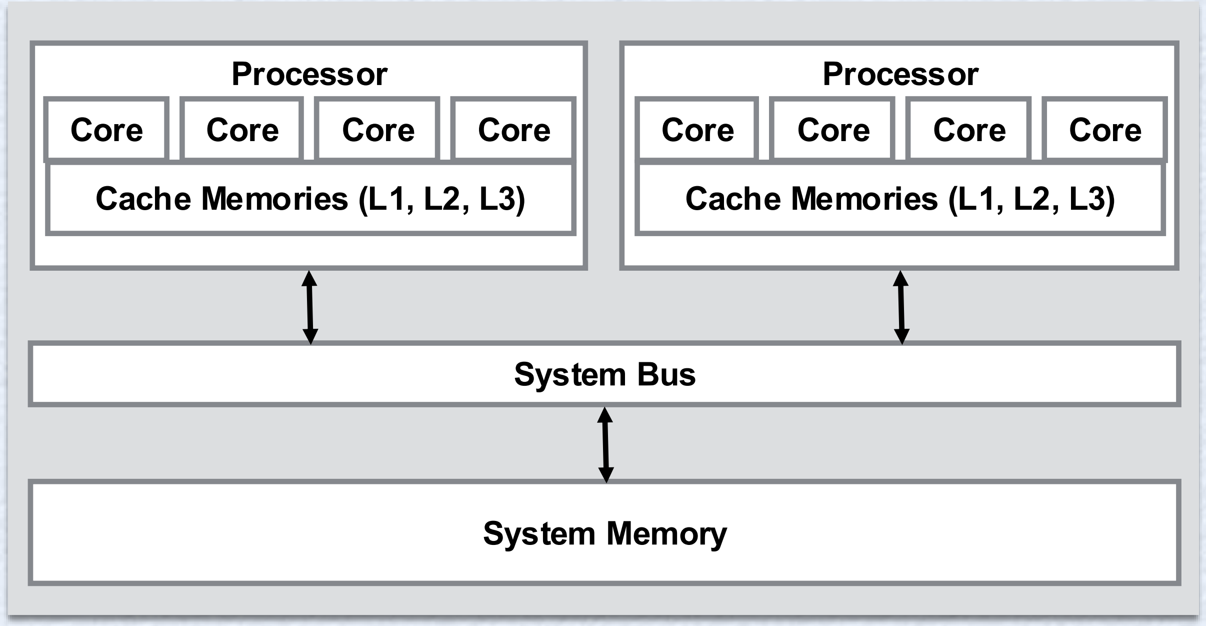
SIMD

GPUs

CUDA

Some terminology

- Parallelism in Hardware:
  - multiple **cores** and **memory**
- Parallelism in Software:
  - **process**: executed program (has its own memory space etc.), can contain multiple threads, can run in parallel, can communicate with other processes
  - **thread**: can run in parallel and all threads of the same process share the application data (memory)



```
int a[1000];

int main( int argc, char** argv )
{
  for(int i =   0; i <  500; i++ ) a[i] = 0;
  for(int i = 500; i < 1000; i++ ) a[i] = 1;

  return 0;
}
```

Sequential Code

```
int main(int argc, char** argv )
{
   // vector size
   const int N = 1600000;

   // initialize vectors
   std::vector<float> x(N,-1.2), y(N,3.4), z(N);

   // do the sum z = x + y
   for(int i = 0; i < N; i++) z[i] = x[i] + y[i];

   return 0;
}
```

SIMD

```
int main(int argc, char** argv)
{
   // vector size
   const int N = 1600000;

   // initialize vectors (assume correct memory alignment)
   std::vector<float> x(N,-1.2), y(N,3.4), z(N);

   // DO THE SUM z = x + y with SSE (width=4)
   for( int i = 0; i < N; i += 4 )
   {
      // z[i] = x[i] + y[i];
      __m128 xx = _mm_load_ps( &x[i] );
      __m128 yy = _mm_load_ps( &y[i] );
      __m128 zz = _mm_add_ps( xx, yy );
      _mm_store_ps( &z[i], zz );
   }

   return 0;
}
```

C++ Threads

```
int main(int argc, char** argv)
{
   // vector size
   const int N = 1600000;

   // initialize vectors
   std::vector<float> x(N,-1.2), y(N,3.4), z(N);


   // DO THE SUM z = x + y using 4 threads
   int num_threads = 4;
   int chunk = N / num_threads;
   std::vector<std::thread> threads;

   for (int t = 0; t < num_threads; t++)
   {
      threads.emplace_back( [&,t] {
         for( int i = t*chunk; i < (t+1)*chunk; i++)
            z[i] = x[i] + y[i];
         });
   }

   for (std::thread& t:threads)
      t.join();

   return 0;

}
```

POSIX Threads

```
struct arg_t
{
   float *x;
   float *y;
   float *z;
   int t;
   int chunk;
};


void *work(void *argument)
{
   struct arg_t *args = (struct arg_t *)argument;

   float *x = args->x;
   float *y = args->y;
   float *z = args->z;
   int t = args->t;
   int chunk = args->chunk;

   for (int i = t*chunk; i < (t+1)*chunk; i++)
         z[i] = x[i] + y[i];

   return NULL;
}
```

```
int main(int argc, char** argv)
{
   // vector size
   const int N = 1600000;

   // initialize vectors
   std::vector<float> x(N,-1.2), y(N,3.4), z(N);


   // DO THE SUM z = x + y using 4 threads
   int num_threads = 4;
   int chunk = N / num_threads;

   struct arg_t args[num_threads];
   pthread_t threads[num_threads];

   for (int t = 0; t < num_threads; t++)
   {
      args[t].x = &x[0];
      args[t].y = &y[0];
      args[t].z = &z[0];
      args[t].t = t;
      args[t].chunk = chunk;

      pthread_create(&threads[t], NULL, add, &args[t]);
   }

   for (int t = 0; t < num_threads; ++t)
      pthread_join(threads[t], NULL);

  return 0;

}
```

OpenMP

```
int main(int argc, char** argv)
{
   // vector size
   const int N = 1600000;

   // initialize vectors
   std::vector<float> x(N,-1.2), y(N,3.4), z(N);

   // do the sum z = x + y
   #pragma omp parallel for
   for (int i = 0; i < N; i++) z[i] = x[i] + y[i];

   return 0;
}
```

MPI

```
int main( int argc, char** argv )
{
   // vector size
   const int N = 1600000;

   // Initialize communication, determine num_processes and our rank
   int num_processes, rank;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD,&num_processes);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   // initialize local parts of the vectors and do the sum z = x + y
   int nlocal = N / num_processes;
   std::vector<float> x(nlocal,-1.2), y(nlocal,3.4), z(nlocal);

   for (int i = 0; i < nlocal; i++ ) z[i] = x[i] + y[i];

   if (rank == 0 )
   {
      std::vector<float> fullz(N);
      // collect all parts into fullz
      MPI_Gather(&z[0],nlocal,MPI_FLOAT,&fullz[0],nlocal,MPI_FLOAT, 0,MPI_COMM_WORLD);
   }
   else
      MPI_Gather(&z[0],nlocal,MPI_FLOAT,NULL,0,MPI_FLOAT,0,MPI_COMM_WORLD);

   MPI_Finalize();

   return 0;
}
```

MPI + OpenMP

```
int main( int argc, char** argv )
{
   // vector size
   const int N = 1600000;

   // Initialize communication, determine num_processes and our rank
   int num_processes, rank;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD,&num_processes);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   // initialize local parts of the vectors and do the sum z = x + y
   int nlocal = N / num_processes;
   std::vector<float> x(nlocal,-1.2), y(nlocal,3.4), z(nlocal);

   #pragma omp parallel for
   for(int i = 0; i < nlocal; i++ ) z[i] = x[i] + y[i];

   if (rank == 0)
   {
      std::vector<float> fullz(N);
      // collect all parts into fullz
      MPI_Gather(&z[0],nlocal,MPI_FLOAT,&fullz[0],nlocal,MPI_FLOAT, 0,MPI_COMM_WORLD);
   }
   else
      MPI_Gather(&z[0],nlocal,MPI_FLOAT,NULL,0,MPI_FLOAT,0,MPI_COMM_WORLD);

   MPI_Finalize();

   return 0;
}
```

When do I need Parallelism?

A program needs to:

- be correct
- solve an important problem
- provide a useful interface (to people and other programs)

OK
Sequential

- Fast
- Throughput
- Optimization

Only
Parallel

Programming Environments

Threads, OpenMP
- Work sharing
- Synchronization
- Performance and optimization
- Nested parallelism
- Tasking model

Message Passing (MPI)
- Point-to-point communication
- Collective operations
- Non-blocking communication
- Parallel I/O

MPI+OpenMP
- Motivation
- General concepts
- Hybrid programming on SMP clusters

Vectorization
- SIMD Instructions
- Data alignment

GPUs
- Architecture
- CUDA programming

What YOU should get out of the course

- Understanding of computer hardware options from the HPC perspective
- Overview of Multithreading/OpenMP, MPI, CUDA and experience using them
- Performance analysis and tuning
- Exposure to various open research questions