CME 213

SPRING 2012-2013

Eric Darve

Stanford University

SYLLABUS

People

Instructors:

- **Eric Darve, M.E.**
- **Erich Elsen, Royal Caliber**
- **NVIDIA engineers:**
  - Steve Rennich
  - Justin Luitjens
  - David Goodwin
  - Sean Baxter

Teaching assistants:

- **Sammy El Ghazzal, head T.A.**
- **Henry Wang**
- **Qiyuan Tian**

Web sites

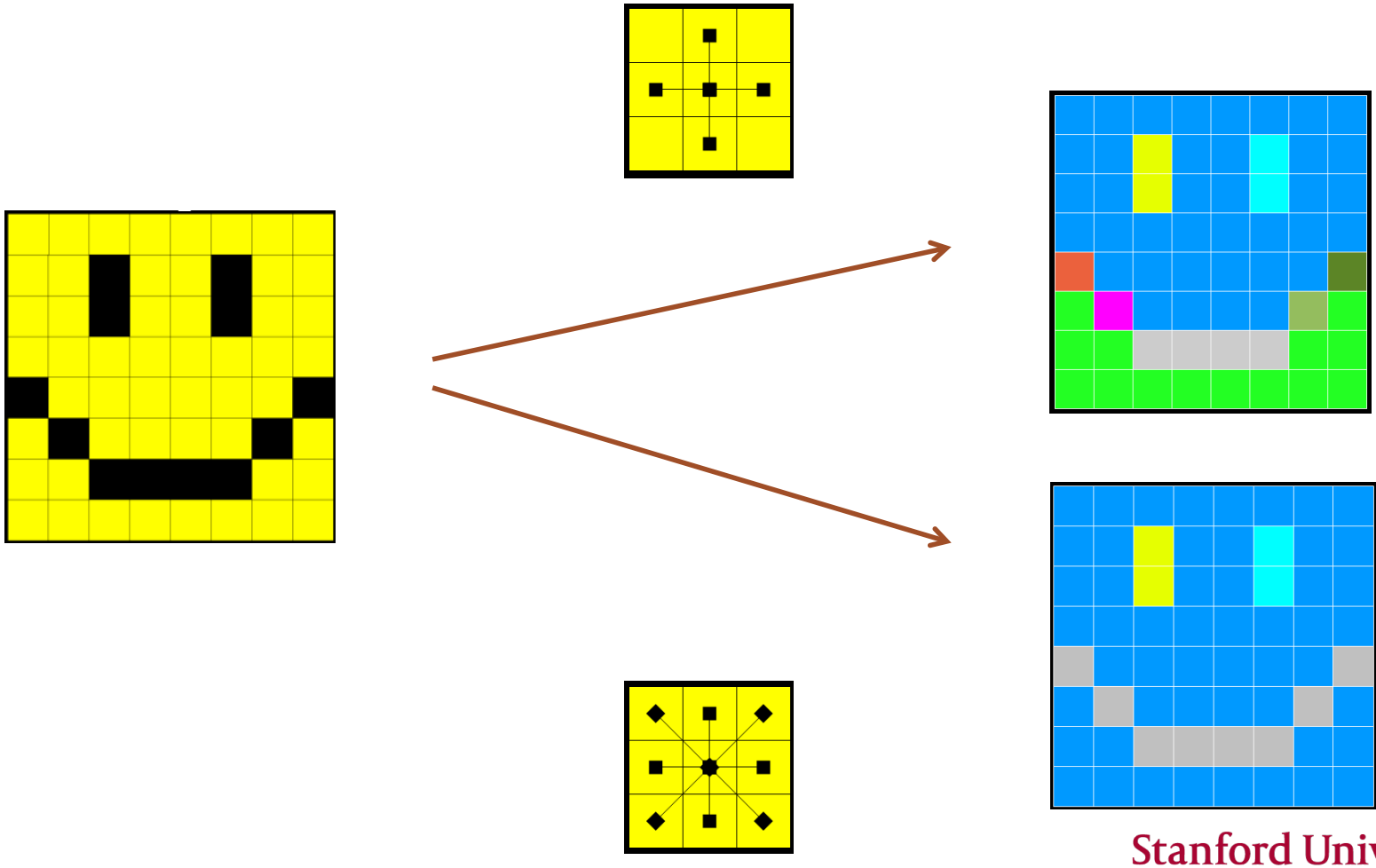
- Class grades and material: coursework
- Class forum: piazza (you may need to register manually)

GRADING, HOMEWORK, PROJECT

- **5 homework assignments: 70% of grade**
- **One final project: 30% of grade**
- **Curving algorithm. This may change at any time!**
  - A: 55%
  - B: 40%
  - C: 5%
- **Honor code:**
  - You cannot copy someone's computer code.
  - The best way to make sure there is no honor code violation is to never show your code or paper to other students, and to not look at the code or paper of other students.
  - It's very tempting to want to copy code that works from another student, but this is a violation.
  - You are still allowed to discuss solutions and interpretations but the answers in the homework must be your own.

THE FINAL PROJECT

The final project will be an implementation of the connected component algorithm using CUDA and GPUs.



Books

Good news: all books are available electronically from the Stanford Library. Just go to:

http://searchworks.stanford.edu/

OpenMP, MPI, Parallel programming

- Parallel Programming for Multicore and Cluster Systems, Rauber and Rünger. Applications focus mostly on linear algebra.
- Introduction to Parallel Computing, Grama, Gupta, Karypis, Kumar. Wide range of applications from sort to FFT, linear algebra and tree search.
- An introduction to parallel programming, Pacheco. More examples and less theoretical.

OpenMP and Multicore Books

- Using OpenMP: portable shared memory parallel programming, Chapman, Jost, van der Pas. Advanced coverage of OpenMP.
- Parallel Programming in OpenMP, Chandra, Menon, Dagum, Kohr, Maydan, McDonald; a bit outdated.
- The art of multiprocessor programming, Herlihy, Shavit. Specializes on advanced multicore programming.

CUDA BOOKS

- CUDA by Example: An Introduction to General-Purpose GPU Programming, Sanders, Kandrot
- CUDA Handbook: A Comprehensive Guide to GPU Programming, Wilt
- ~~CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, Cook~~

What this class is about

- **We will focus on how to program:**
  - Multicore processors, e.g., desktop processors: Pthread, OpenMP.
  - NVIDIA graphics processors using CUDA.
  - Computer clusters using MPI.
- **We will cover some numerical algorithms for illustration: sort, linear algebra, basic parallel primitives.**

What this class is not about

- **Parallel computer architecture**
- **Parallel design patterns and programming models**
- **Parallel numerical algorithms. See for example *CME 342: Parallel Methods in Numerical Analysis***

What this class requires

- Some basic knowledge of UNIX (ssh, makefile, etc)
- Good knowledge of C and C++ (including pointers, templates)
- Proficiency in scientific programming, including debugging and testing

SCHEDULE

1. Pthread, Eric Darve, 2 lectures
2. OpenMP, Eric Darve, 2 lectures
3. CUDA, Erich Elsen, 6 lectures
4. CUDA, NVIDIA, 4 lectures
5. MPI, Eric Darve, 5 lectures

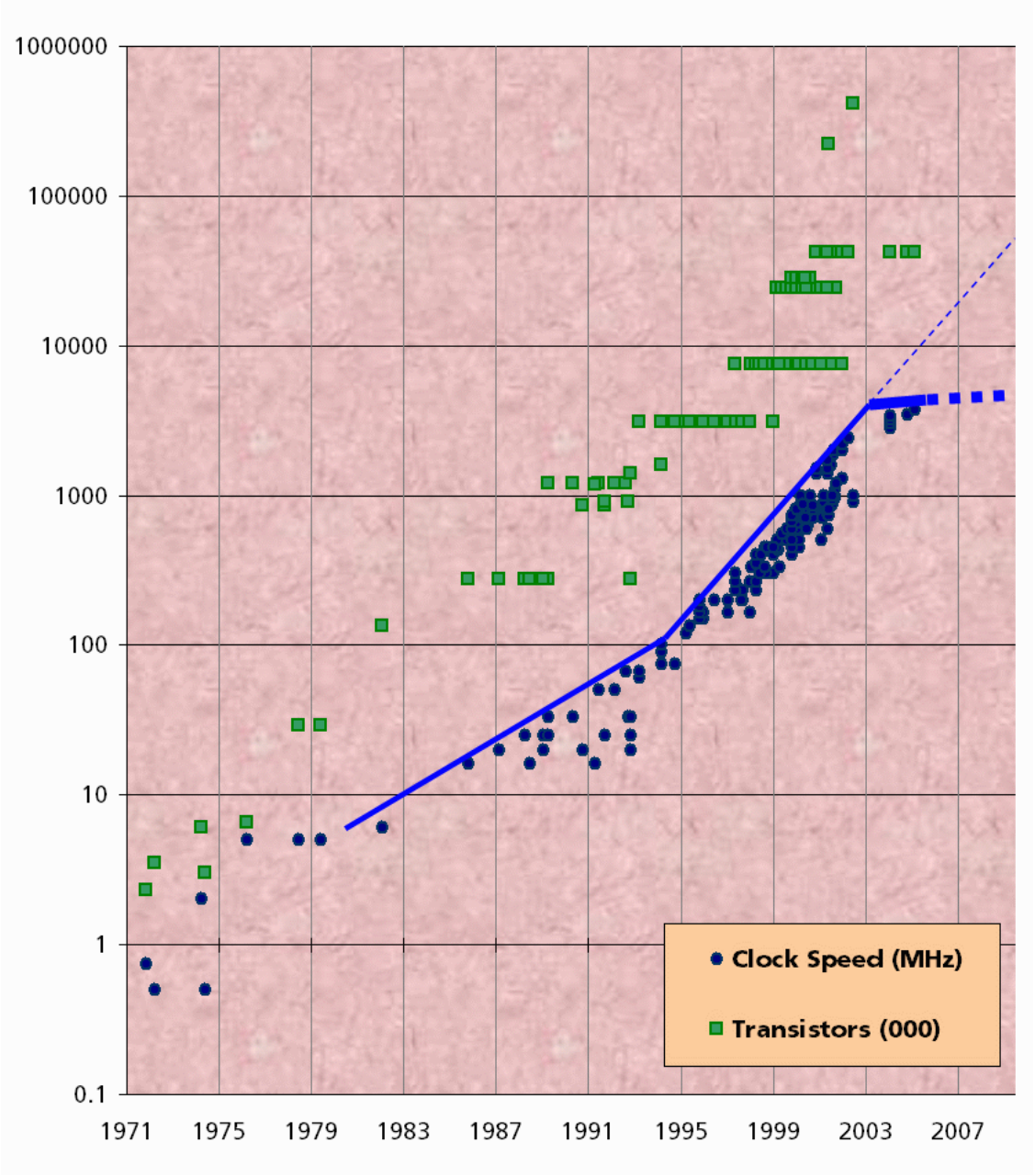
WHY PARALLEL COMPUTING?

WHY PARALLEL COMPUTING?

- **Parallel computing has existed for a long time but until recently it was a specialized area that concerned only a small fraction of engineers.**
- **Nowadays, parallel computing is a dominant player in scientific and large scale computing.**
- **What happened?**

Why parallel computing?

- Gordon Moore 1965: the number of transistors on a chip shall double every 18-24 months.
- This has been valid for more than 40 years.
- This increase in the number of transistors has been accompanied by an increase in clock speed.

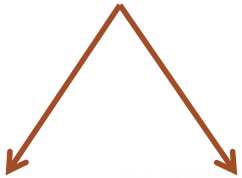

1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
Clock Speed (MHz)
Transistors (000)

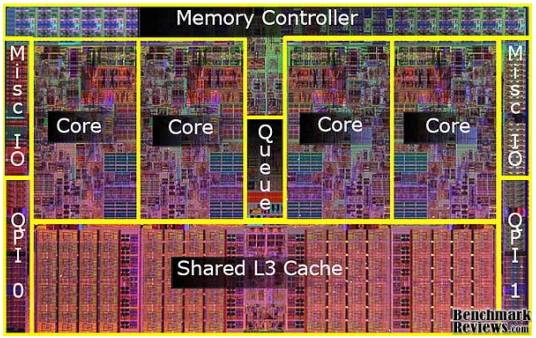
Performance increase

- **Increase in transistor density is limited by:**
  - Leakage current increases
  - Power consumption increases
  - Heat generated increases
- **In addition, the memory access time has not been reduced at a rate comparable with processing speed (processor clock period).**
- **New ways need to be found.**
- **The most promising approach is to have multiple cores on a single processor.**

PARALLEL COMPUTING EVERYWHERE: MOBILE DEVICES

Laptops and desktops

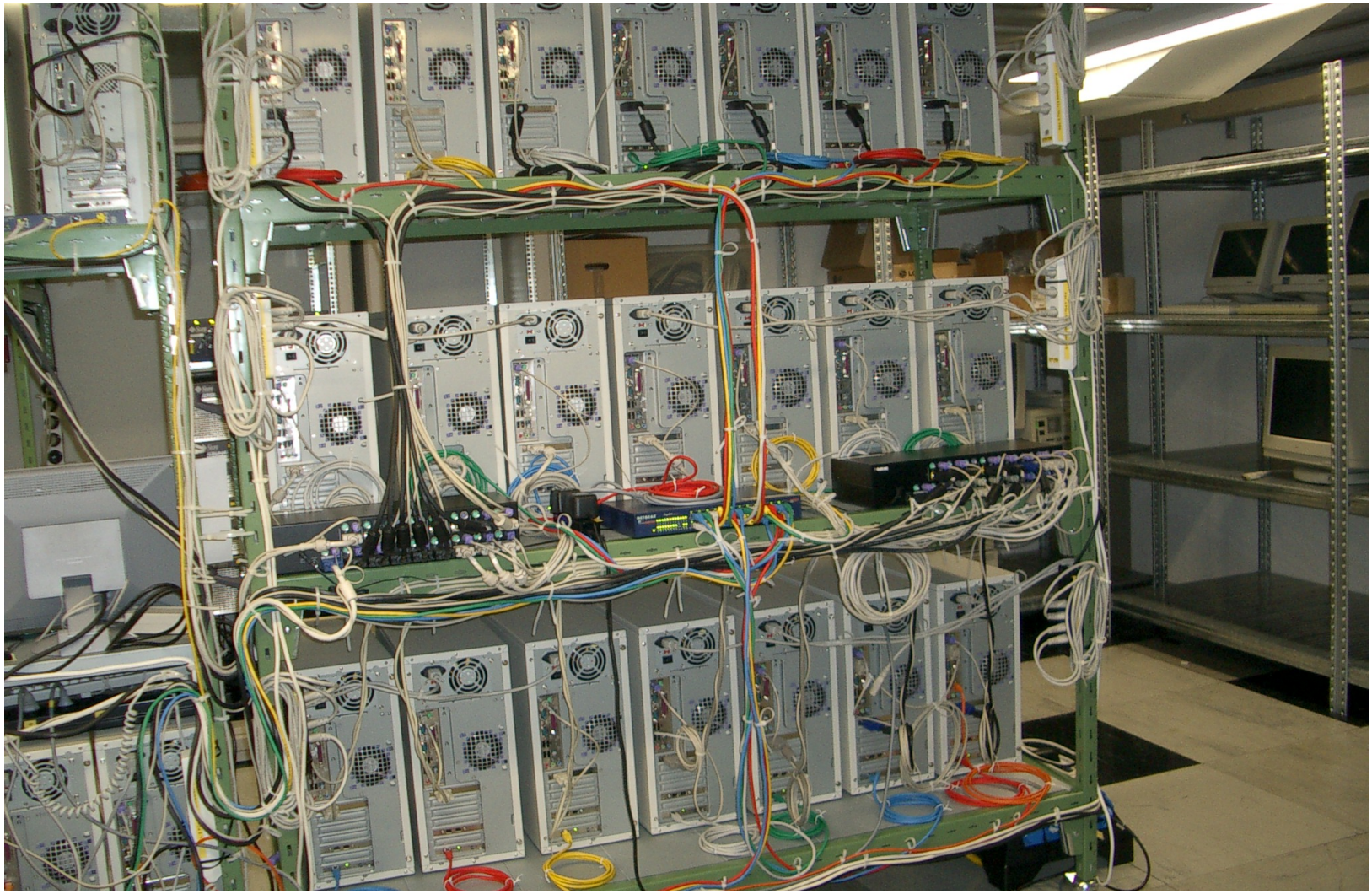




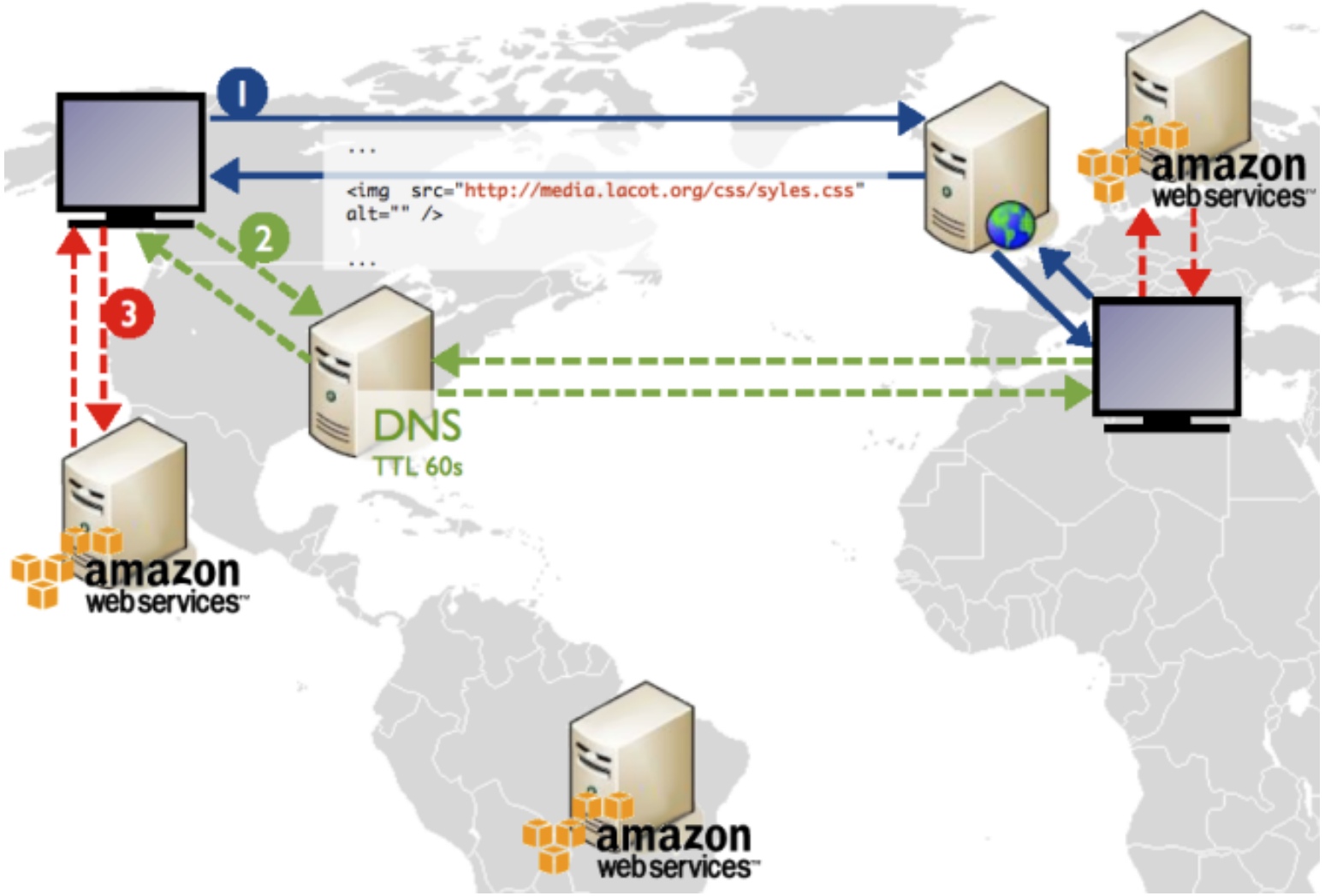
multicore

GPU

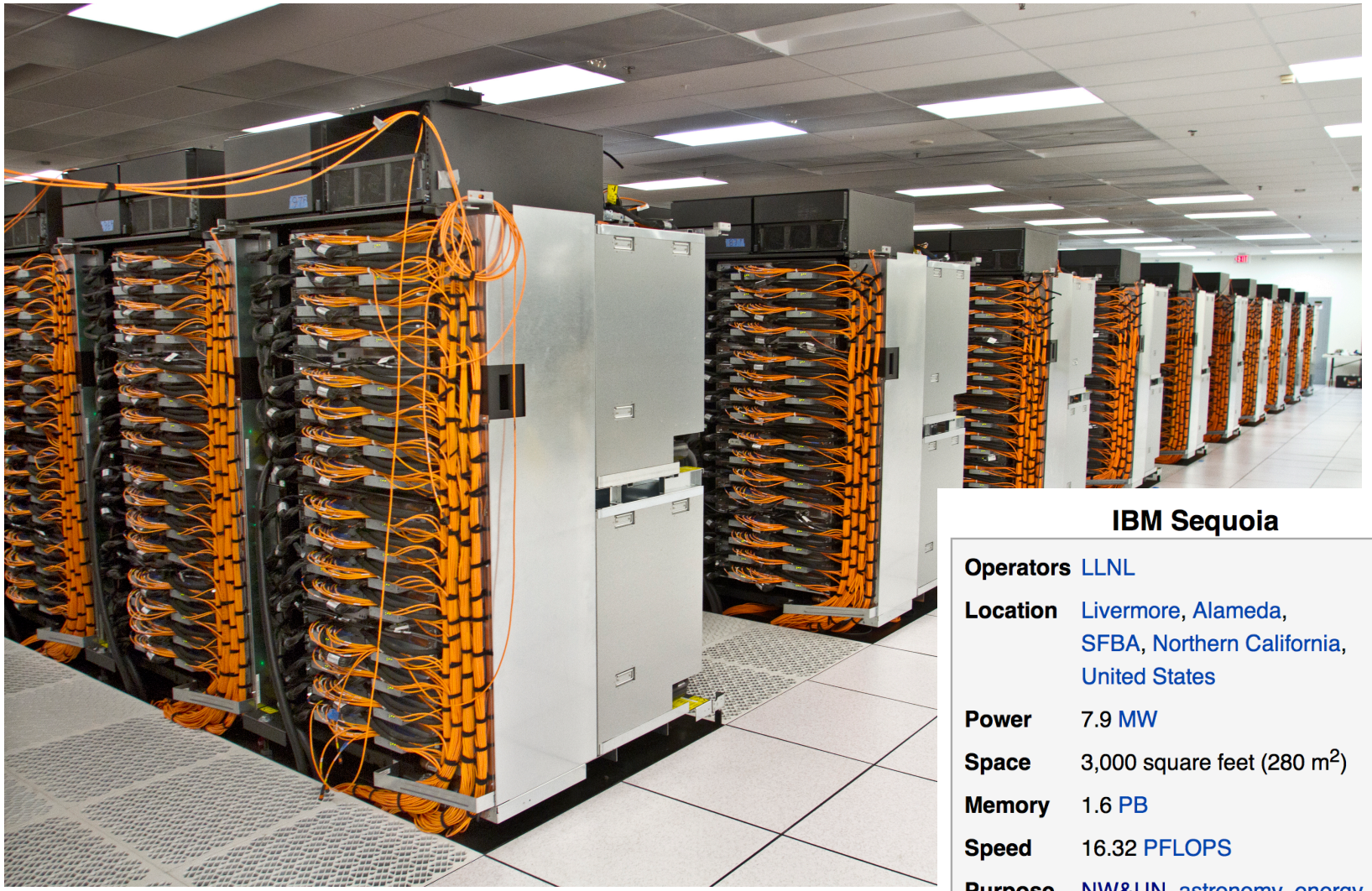
CLUSTERS



CLOUD COMPUTING



SUPERCOMPUTERS



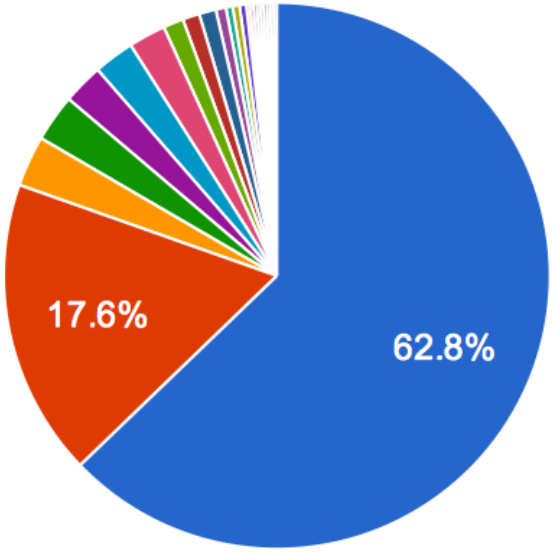
IBM Sequoia

Operators	LLNL
Location	Livermore, Alameda, SFBA, Northern California, United States
Power	7.9 MW
Space	3,000 square feet (280 m^2)
Memory	1.6 PB
Speed	16.32 PFLOPS
Purpose	NW&UN, astronomy, energy, human genome, and climate change

Statistics about the top 500 Supercomputers

Go to www.top500.org

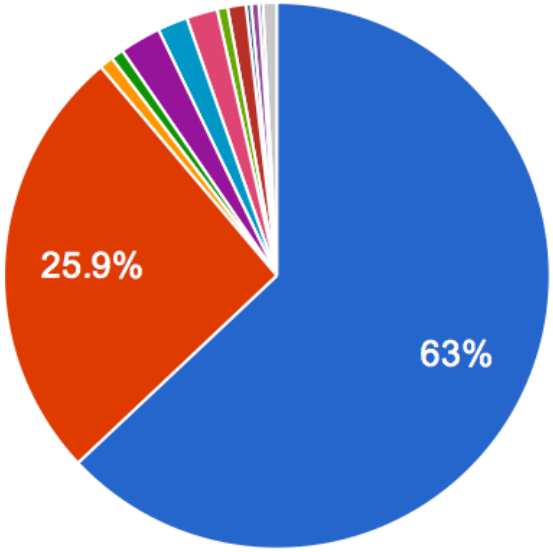
Application Area System Share



- Not Specified
- Research
- Web Services
- Geophysics
- Weather and Climate ...
- Energy
- Defense
- Benchmarking

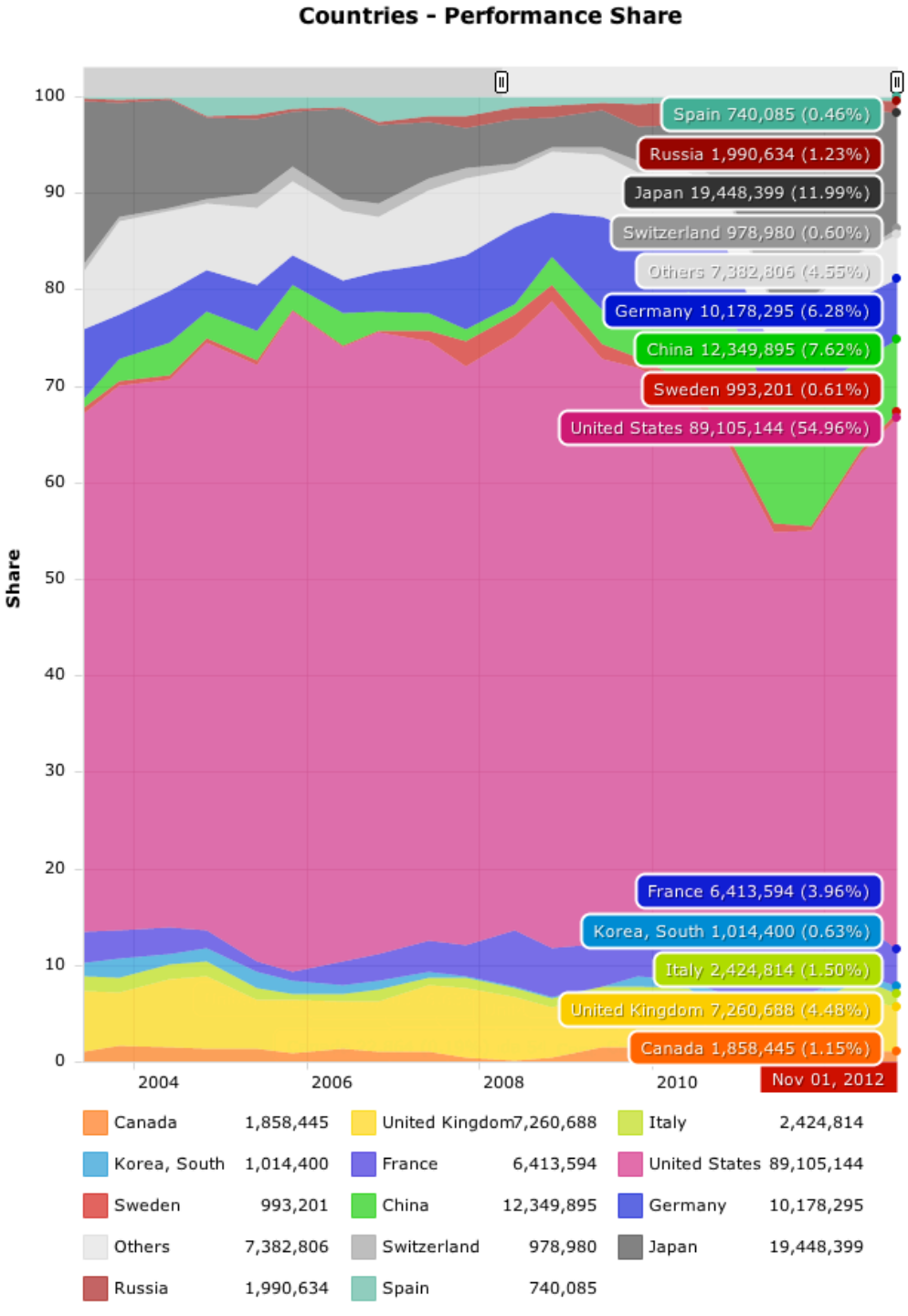
1/3

Application Area Performance Share

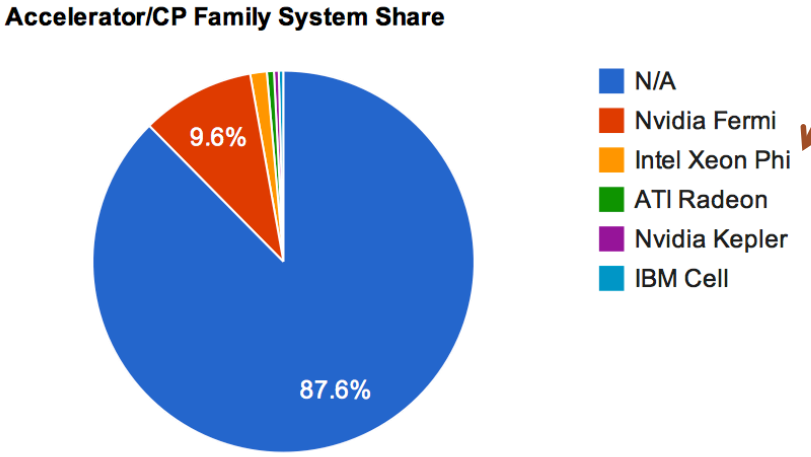


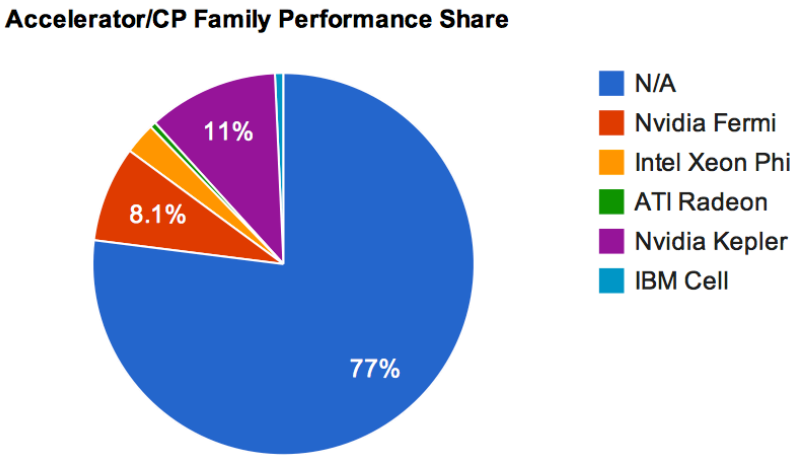
- Not Specified
- Research
- Web Services
- Geophysics
- Weather and Climate ...
- Energy
- Defense
- Benchmarking

1/2


Countries - Performance Share
Share
100
90
80
70
60
50
40
30
20
10
0
2004
2006
2008
2010
Nov 01, 2012
Spain 740,085 (0.46%)
Russia 1,990,634 (1.23%)
Japan 19,448,399 (11.99%)
Switzerland 978,980 (0.60%)
Others 7,382,806 (4.55%)
Germany 10,178,295 (6.28%)
China 12,349,895 (7.62%)
Sweden 993,201 (0.61%)
United States 89,105,144 (54.96%)
France 6,413,594 (3.96%)
Korea, South 1,014,400 (0.63%)
Italy 2,424,814 (1.50%)
United Kingdom 7,260,688 (4.48%)
Canada 1,858,445 (1.15%)
Canada 1,858,445
United Kingdom 7,260,688
Italy 2,424,814
Korea, South 1,014,400
France 6,413,594
United States 89,105,144
Sweden 993,201
China 12,349,895
Germany 10,178,295
Others 7,382,806
Switzerland 978,980
Japan 19,448,399
Russia 1,990,634
Spain 740,085

Intel Many Integrated Core Architecture or MIC, 60 cores





Accelerator/CP Family	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
N/A	438	87.6	124844008	165167770	12795391
Nvidia Fermi	48	9.6	13133114	26922764	984676
Intel Xeon Phi	7	1.4	4302764	6309356	337301
ATI Radeon	3	0.6	827300	1832963	62832
Nvidia Kepler	2	0.4	17863700	27505427	568800
IBM Cell	2	0.4	1168500	1537632	136800

Example of Parallel Computation

Why we need to write parallel programs

- Most programs you have written so far are (probably) sequential.
- Unfortunately parallel programs often look very different.
- An efficient parallel implementation of a serial program may not be obtained by simply parallelizing each step.
- Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm.

LET'S CALCULATE THE SUM OF N NUMBERS

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(…);
    sum += x;
}
```

Our first parallel program

- Assume we have p cores that can compute and exchange data.
- Then we could accelerate the previous calculation by splitting the work among all these cores.

```
my_sum = 0;
my_first_i = … ;
my_last_i = … ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
    my_x = Compute_next_value(…);
    my_sum += my_x;
}
```

BUT IT'S NOT THAT SIMPLE

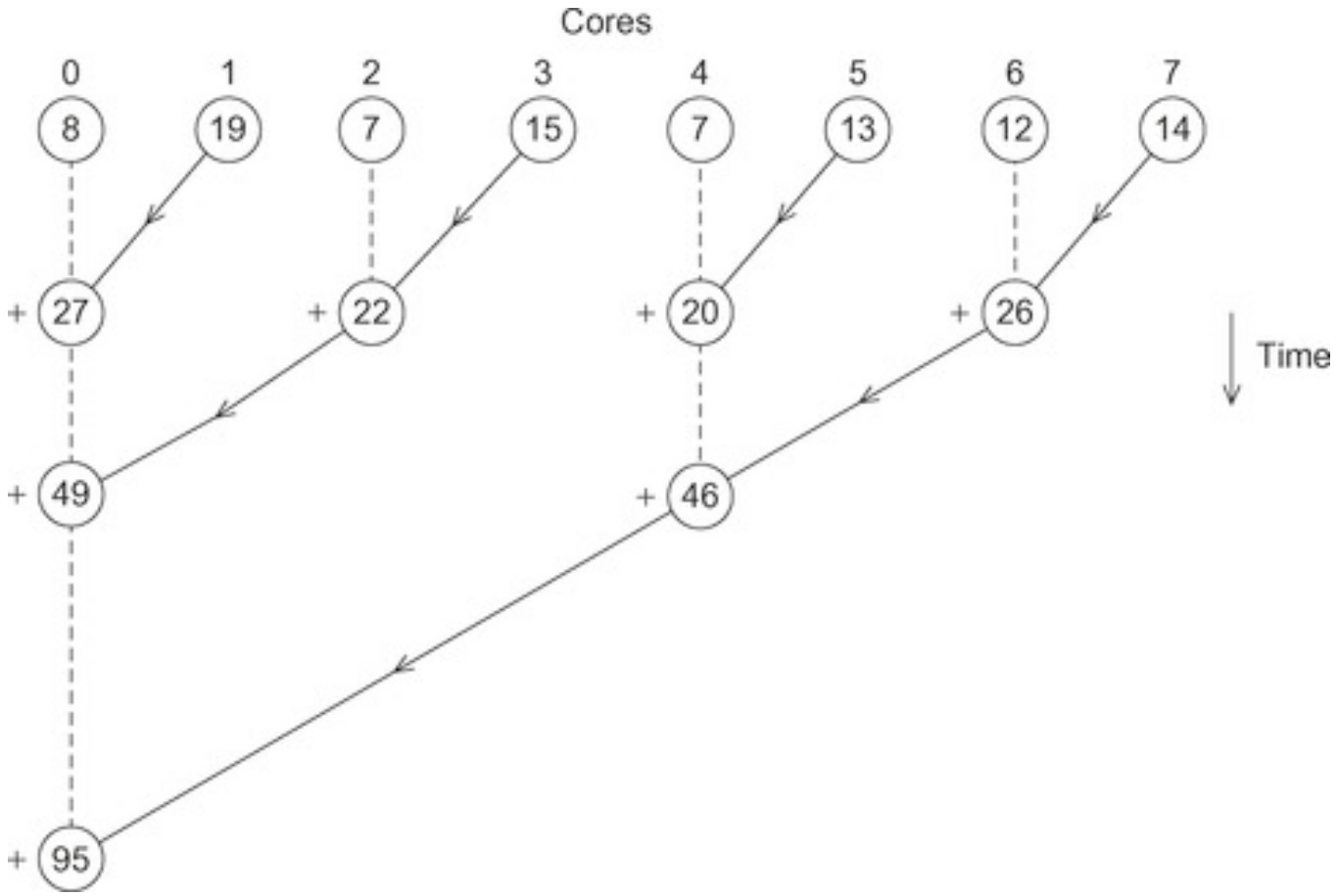
- Each core has computed a partial sum.
- All these partial sums need to summed up together.
- The simplest approach is to have one "master" core do all the work:

In pseudo-code:

```
if (I'm the master core) {
    sum = my_x;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_x to the master;
}
```

THAT MAY NOT BE ENOUGH

- If we have many cores, this final sum may in fact take a lot of time.
- A better implementation would follow the following steps:



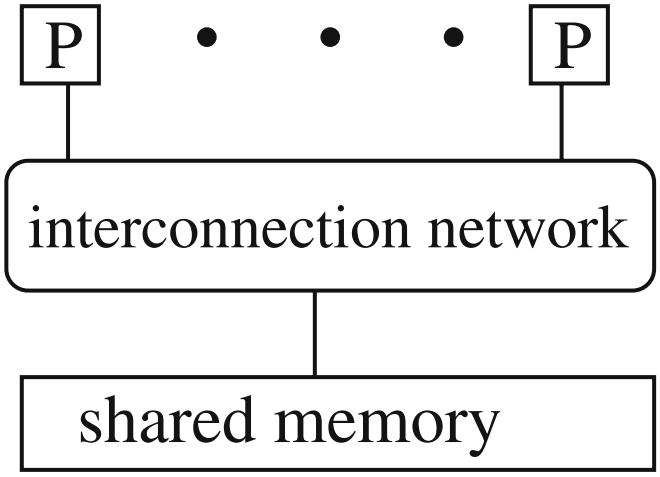
AUTOMATIC PARALLELIZATION

- This simple example illustrates the fact that it is difficult for a compiler to parallelize a program.
- Instead the programmer must re-write his code from scratch having in mind that multiple cores will be computing in parallel.
- The purpose of this class is to teach you the most common parallel languages used in science and engineering.

SHARED MEMORY PROCESSOR

Schematic of a multicore processor

- **Model for shared memory machines**
- **Comprised of:**
  - A number of processors or cores
  - A shared physical memory (global memory)
  - An interconnection network to connect the processors with the memory.



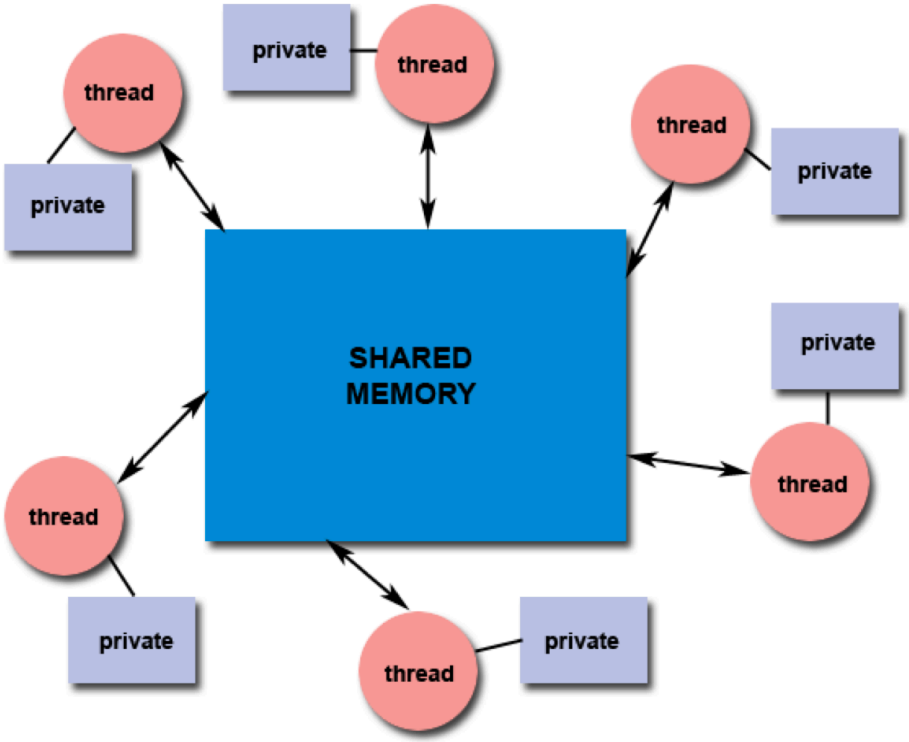
Processes and threads

Definition:

- **Process:**
  - Program in execution.
  - Comprises: the executable program along with all information that is necessary for the execution of the program.
- **Thread: an extension of the process model. Can be viewed as a "lightweight" process.**
- **In this model, each process may consist of multiple independent control flows that are called threads.**
- **A thread may be described as a "procedure" that runs independently from the main program.**
- **Imagine a program that contains a number of procedures. Then imagine these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. This describes a "multi-threaded" program.**

SHARED ADDRESS SPACE

- All the threads of one process share the address space of the process, i.e., they have a common address space.
- When a thread stores a value in the shared address space, another thread of the same process can access this value afterwards.



PTHREADS


LELAND STANFORD JUNIOR UNIVERSITY
DIE LUFT DER FREIHEIT WEHT
1891

Programming using threads

- This is the most basic approach for programming in parallel.
- Pthreads: POSIX threads. This is a standard to implement threads on UNIX systems.
- The other approach is based on OpenMP.

THE BASICS

- Include the header file:

```
<pthread.h>
```

- Compile using:

```
gcc -o pth_hello pth_hello.c -lpthread
```

LET'S DIVE IN

```
…
#include <pthread.h>
#define NUM_THREADS    5

void *PrintHello(void *threadid){
   long tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]){
   pthread_t threads[NUM_THREADS];
   int rc; long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Output

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

THREAD CREATION

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*routine)(void*),
    void *arg)
```

- **`thread`** thread identifier
- **`routine`** function that will be executed by the generated thread
- **`arg`** pointer to the argument value with which the thread function **`routine()`** will be executed
- **`attr`** use **`NULL`** for the time being