Parallel Programming: Principle and Practice

INTRODUCTION

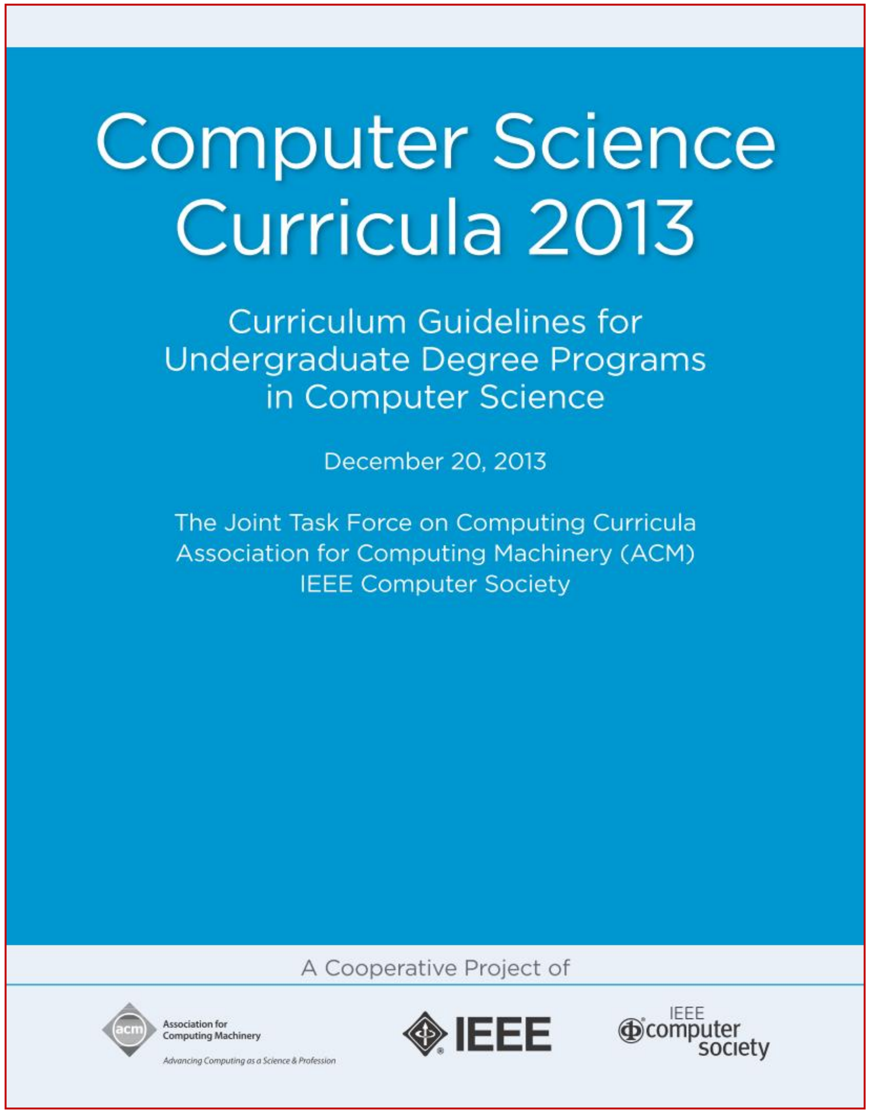
Course Goals

- The students will get the skills to use some of the best existing parallel programming tools, and be exposed to a number of open research questions
- This course will
  - provide an introduction to parallel computing including parallel computer architectures, analytical modeling of parallel programs, the principles of parallel algorithm design
  - include material on TBB, OpenMP, CUDA, OpenCL, MPI, MapReduce
- Course resources
  - http://grid.hust.edu.cn/courses/parallel/

Syllabus

- Part 1: Principles
  - Lec-1 Why Parallel Programming?
  - Lec-2 Parallel Architecture
  - Lec-3 Parallel Programming Models
  - Lec-4 Parallel Programming Methodology
  - Lec-5 Parallel Programming: Performance
- Part 2: Typical issues solved by parallel
  - Lec-6 Shared Memory Programming and OpenMP
  - Lec-7 Threads programming with TBB
  - Lec-8 Programming Using the Message Passing Paradigm
  - Lec-9 Introduction to GPGPUs and CUDA Programming Model
  - Lec-10 Parallel Computing with MapReduce
- Part 3: Parallel Programming Case Study and Assignments
  - Lec-11 Case Study
  - Assignment

IEEE/ACM Computer Science Curricula 2013



Computer Science Curricula 2013

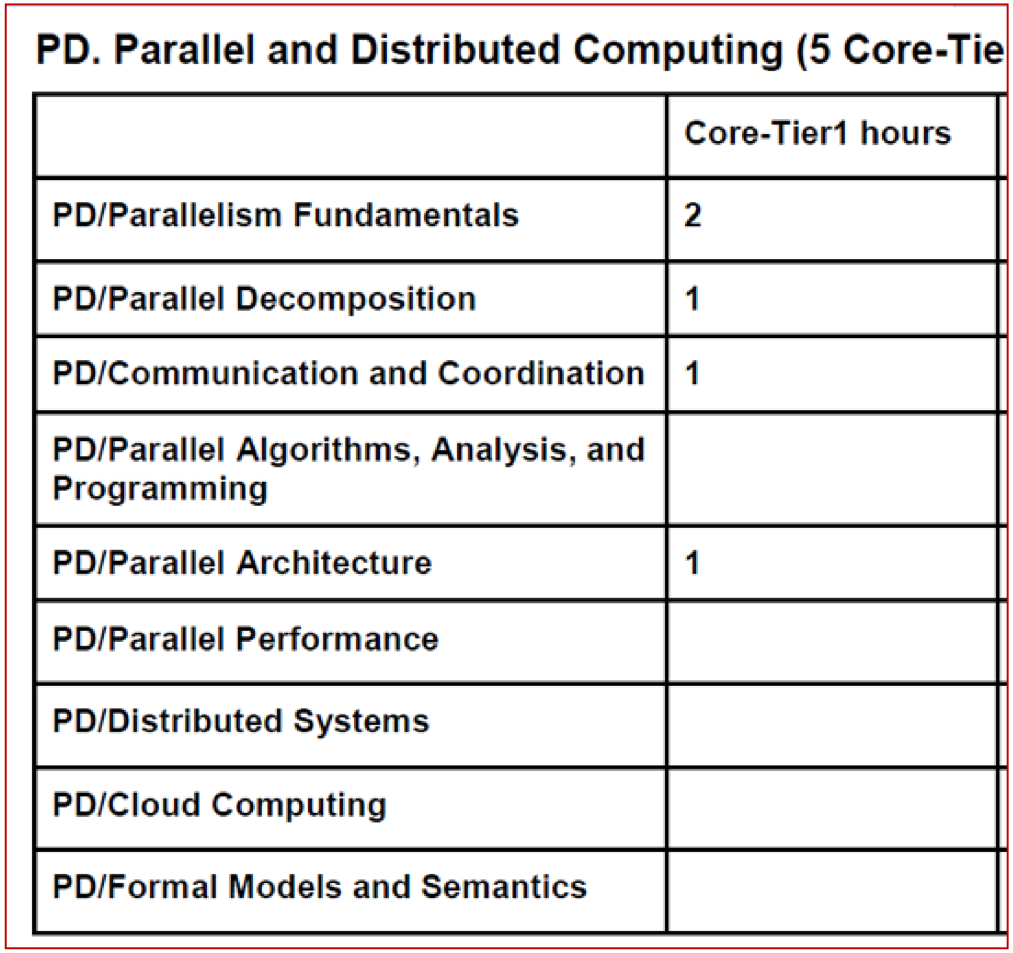
Curriculum Guidelines for Undergraduate Degree Programs in Computer Science

December 20, 2013

The Joint Task Force on Computing Curricula
Association for Computing Machinery (ACM)
IEEE Computer Society

A Cooperative Project of

Association for Computing Machinery
Advancing Computing as a Science & Profession
IEEE
IEEE computer society



PD. Parallel and Distributed Computing (5 Core-Tie

	Core-Tier1 hours
PD/Parallelism Fundamentals	2
PD/Parallel Decomposition	1
PD/Communication and Coordination	1
PD/Parallel Algorithms, Analysis, and Programming	
PD/Parallel Architecture	1
PD/Parallel Performance	
PD/Distributed Systems	
PD/Cloud Computing	
PD/Formal Models and Semantics	

Parallel Programming Principle and Practice

Lecture 1 — Why Parallel Programming?

Outline

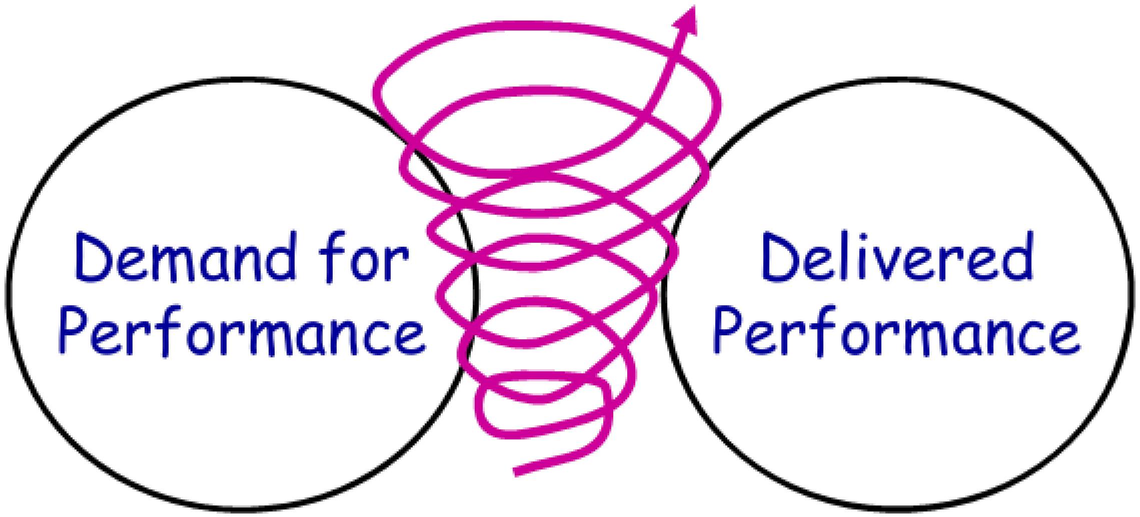
- Application demands
- Architectural trends
- What is parallel programming
- Why do we need parallel programming
- Distributed computing

Why parallel programming

APPLICATION DEMANDS

Application Trends

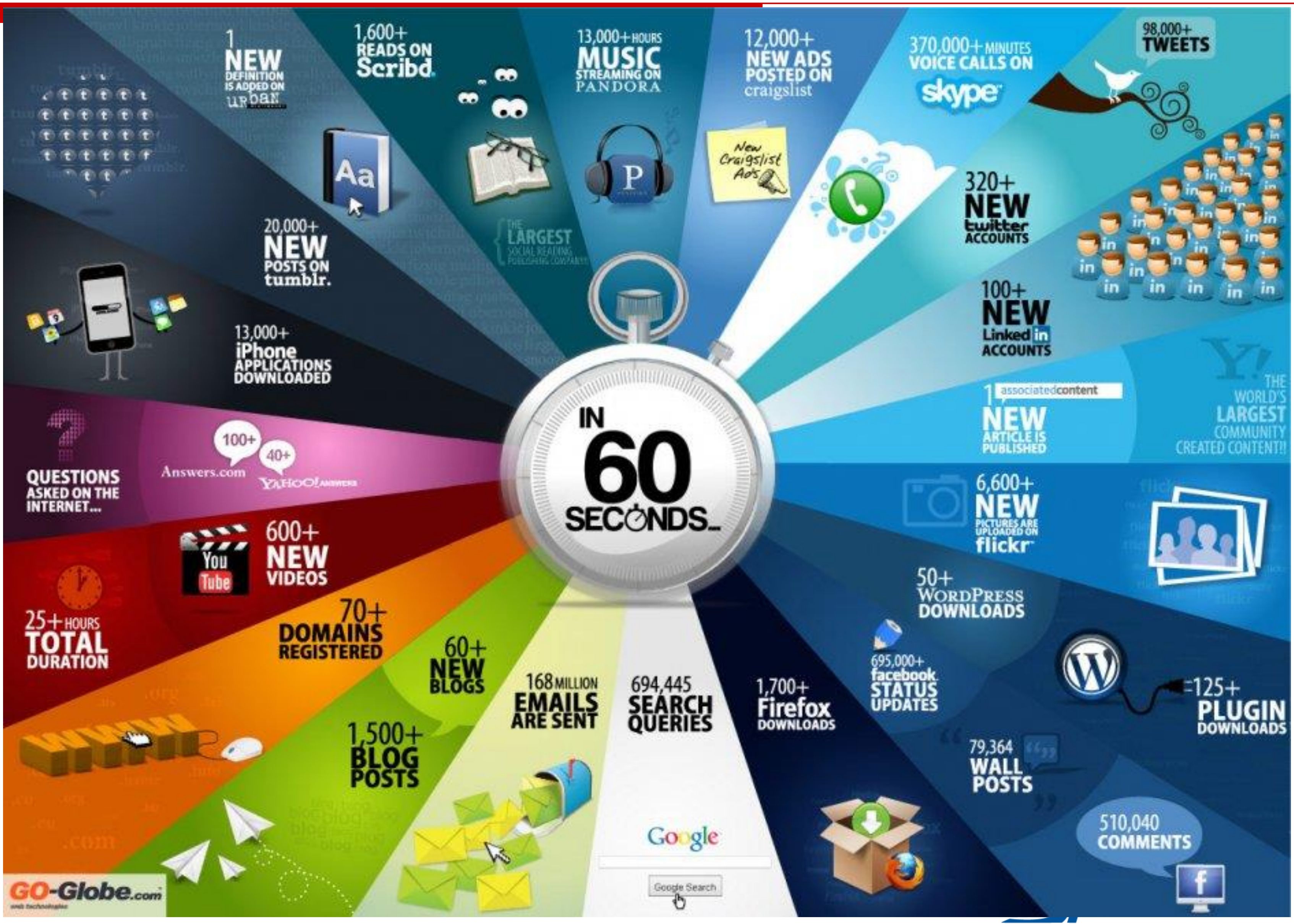
There is a positive feedback cycle between delivered performance and applications' demand for performance



Example application domains:

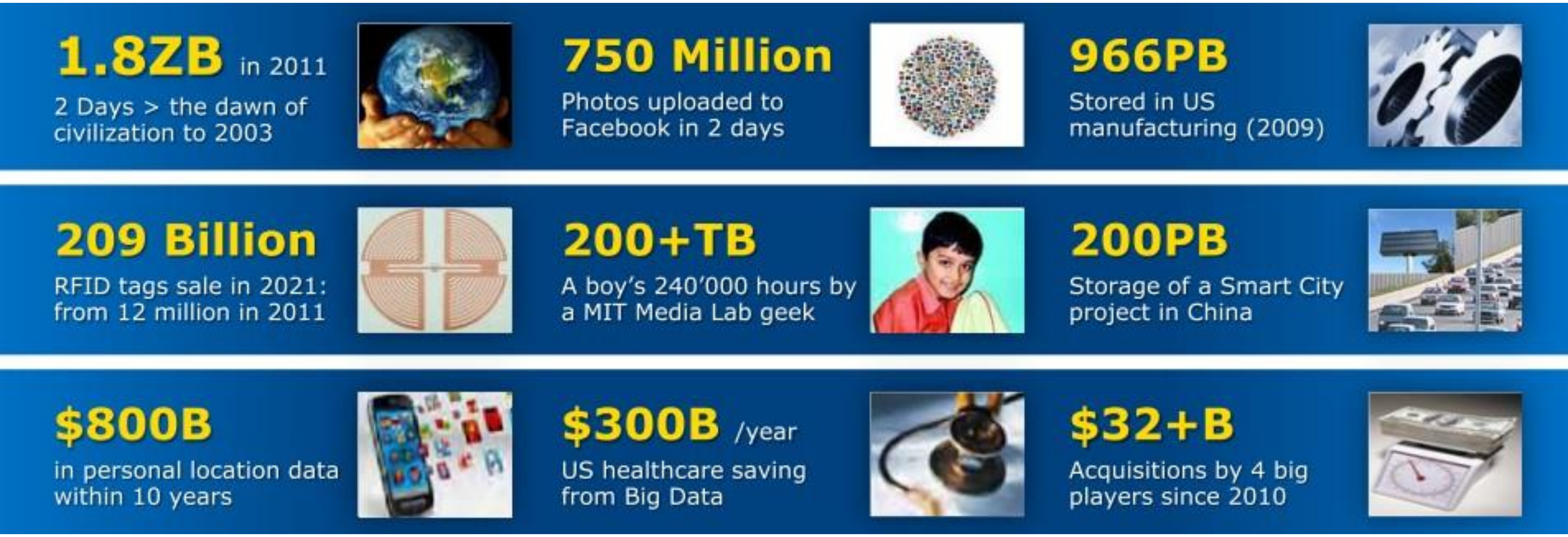
- *Scientific computing* : CFD, Biology, Chemistry, Physics, ...
- *General-purpose computing* : Video, Graphics, CAD, Databases, …

Incredible Things That Happen Every Minute On The Internet


IN 60 SECONDS
1 NEW DEFINITION IS ADDED ON urban
1,600+ READS ON Scribd.
THE LARGEST SOCIAL READING PUBLISHING COMPANY
13,000+ HOURS MUSIC STREAMING ON PANDORA
12,000+ NEW ADS POSTED ON craigslist
New Craigslist Ads
370,000+ MINUTES VOICE CALLS ON skype
98,000+ TWEETS
320+ NEW twitter ACCOUNTS
100+ NEW LinkedIn ACCOUNTS
1 NEW ARTICLE IS PUBLISHED associatedcontent
THE WORLD'S LARGEST COMMUNITY CREATED CONTENT!!
6,600+ NEW PICTURES ARE UPLOADED ON flickr
50+ WORDPRESS DOWNLOADS
125+ PLUGIN DOWNLOADS
695,000+ facebook STATUS UPDATES
79,364 WALL POSTS
510,040 COMMENTS
1,700+ Firefox DOWNLOADS
694,445 SEARCH QUERIES
Google
Google Search
168 MILLION EMAILS ARE SENT
1,500+ BLOG POSTS
60+ NEW BLOGS
70+ DOMAINS REGISTERED
25+ HOURS TOTAL DURATION
600+ NEW VIDEOS
You Tube
QUESTIONS ASKED ON THE INTERNET...
100+ Answers.com
40+ YAHOO! ANSWERS
13,000+ iPhone APPLICATIONS DOWNLOADED
20,000+ NEW POSTS ON tumblr.
GO-Globe.com

Big Data Phenomenon

- "Data are becoming the new raw material of business: an economic input almost on a par with capital and labor"
  —The Economist, 2010
- "Information will be the 'oil of the 21st century"
  —Gartner， 2010



1.8ZB in 2011
2 Days > the dawn of civilization to 2003

750 Million
Photos uploaded to Facebook in 2 days

966PB
Stored in US manufacturing (2009)

209 Billion
RFID tags sale in 2021: from 12 million in 2011

200+TB
A boy's 240'000 hours by a MIT Media Lab geek

200PB
Storage of a Smart City project in China

$800B
in personal location data within 10 years

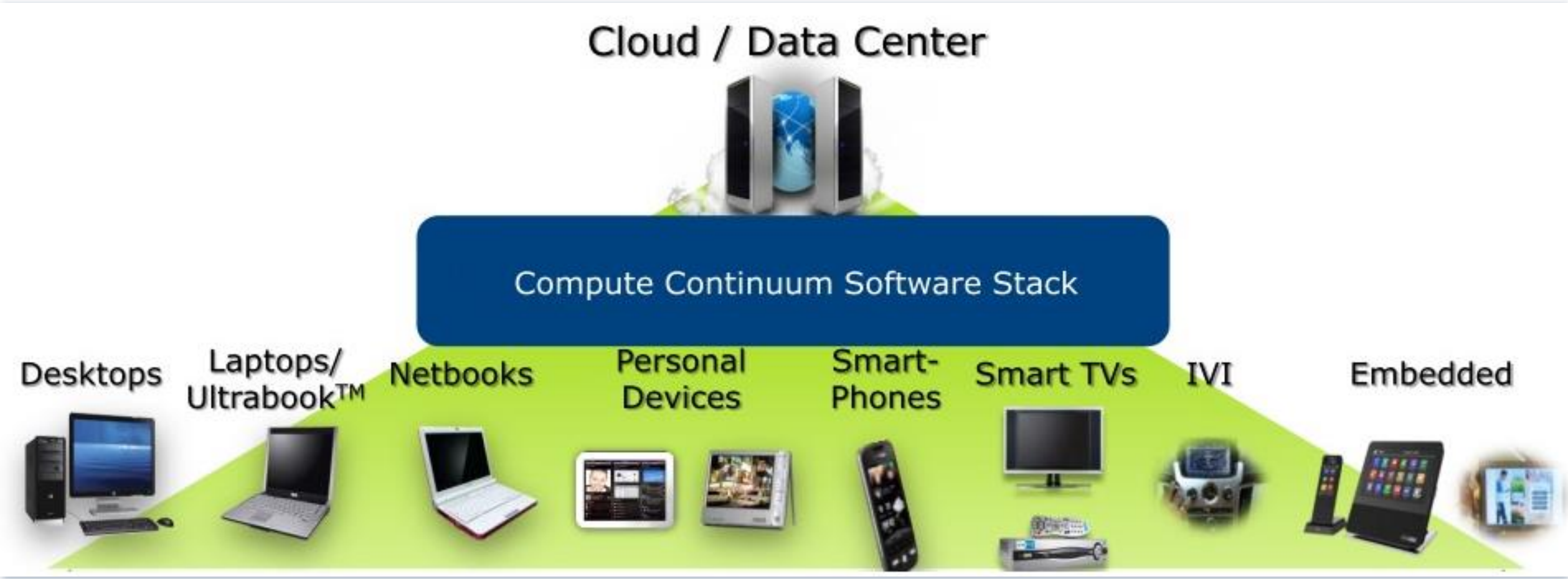
$300B /year
US healthcare saving from Big Data

$32+B
Acquisitions by 4 big players since 2010

Source：IDF2012

Cloud Vision

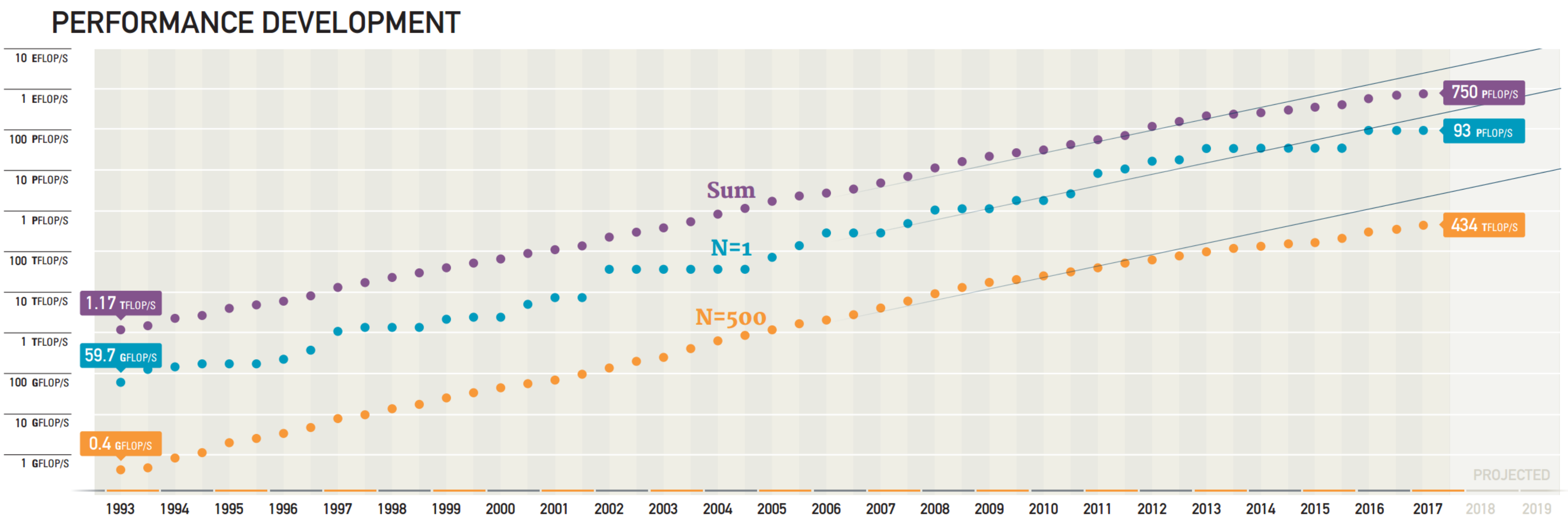
- Coexistence of Opportunities and Challenges



Source：IDF2012

Trends to Exascale Performance

- **Roughly 10x performance every 4 years, predicts that we'll hit Exascale performance in 2018-19**



Why parallel programming

ARCHITECTURAL TRENDS

Architectural Trends

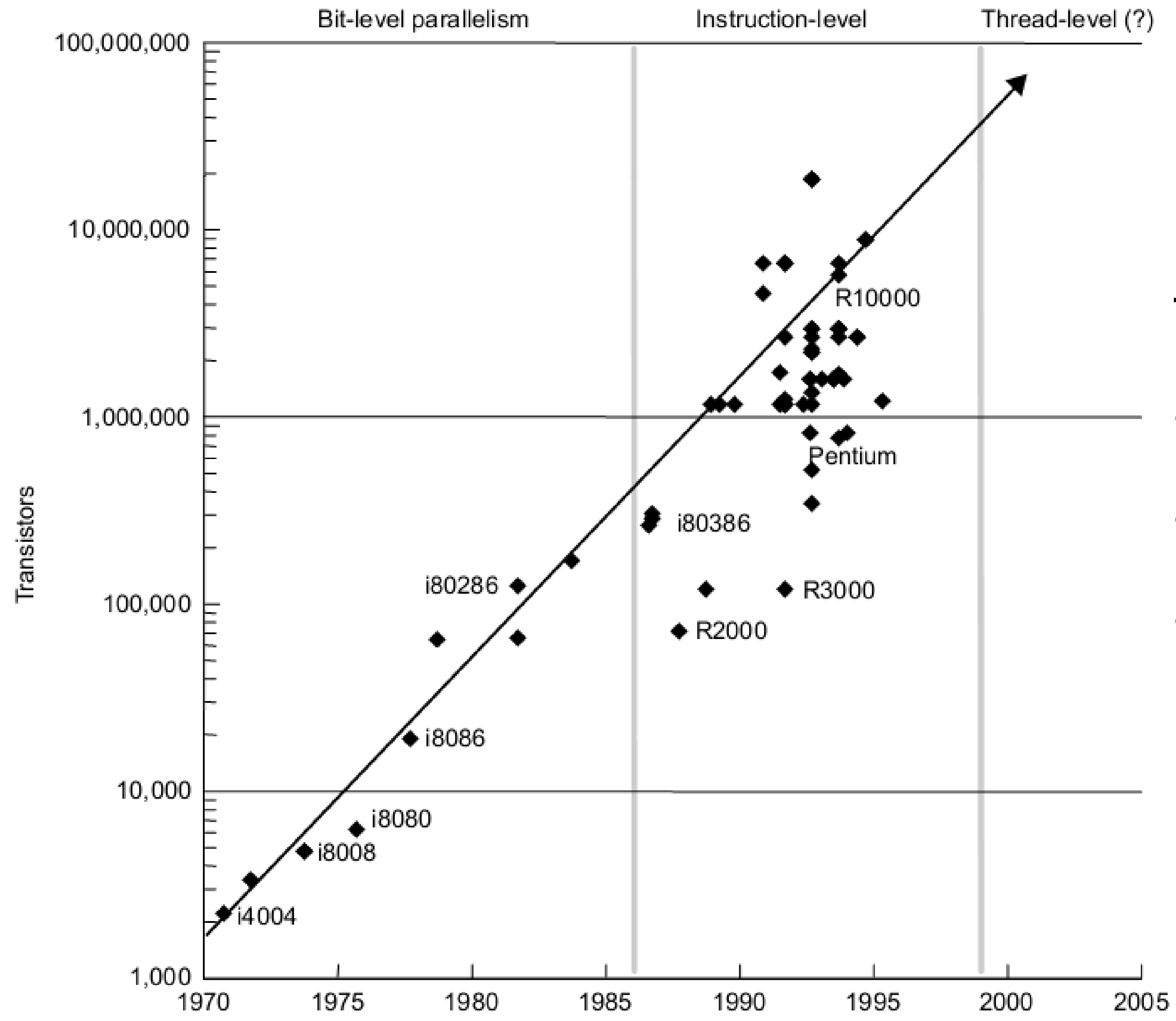
- Architecture translates technology's gifts to performance and capability
- Four generations of architectural history: tube, transistor, IC, VLSI
- Greatest delineation in VLSI has been in type of parallelism exploited

Arch. Trends: Exploiting Parallelism

Greatest trend in VLSI generation increases in parallelism

- Up to 1985: bit level parallelism: 4-bit -> 8 bit -> 16-bit
  - slows after 32 bit
  - adoption of 64-bit now under way, 128-bit far (not performance issue)
  - great inflection point when 32-bit micro and cache fit on a chip
- Mid 80s to mid 90s: instruction level parallelism
  - pipelining and simple instruction sets + compiler advances (RISC)
  - on-chip caches and functional units => superscalar execution
  - greater sophistication: out of order execution, speculation, prediction
    - to deal with control transfer and latency problems
- Now: thread level parallelism

Phases in VLSI Generation

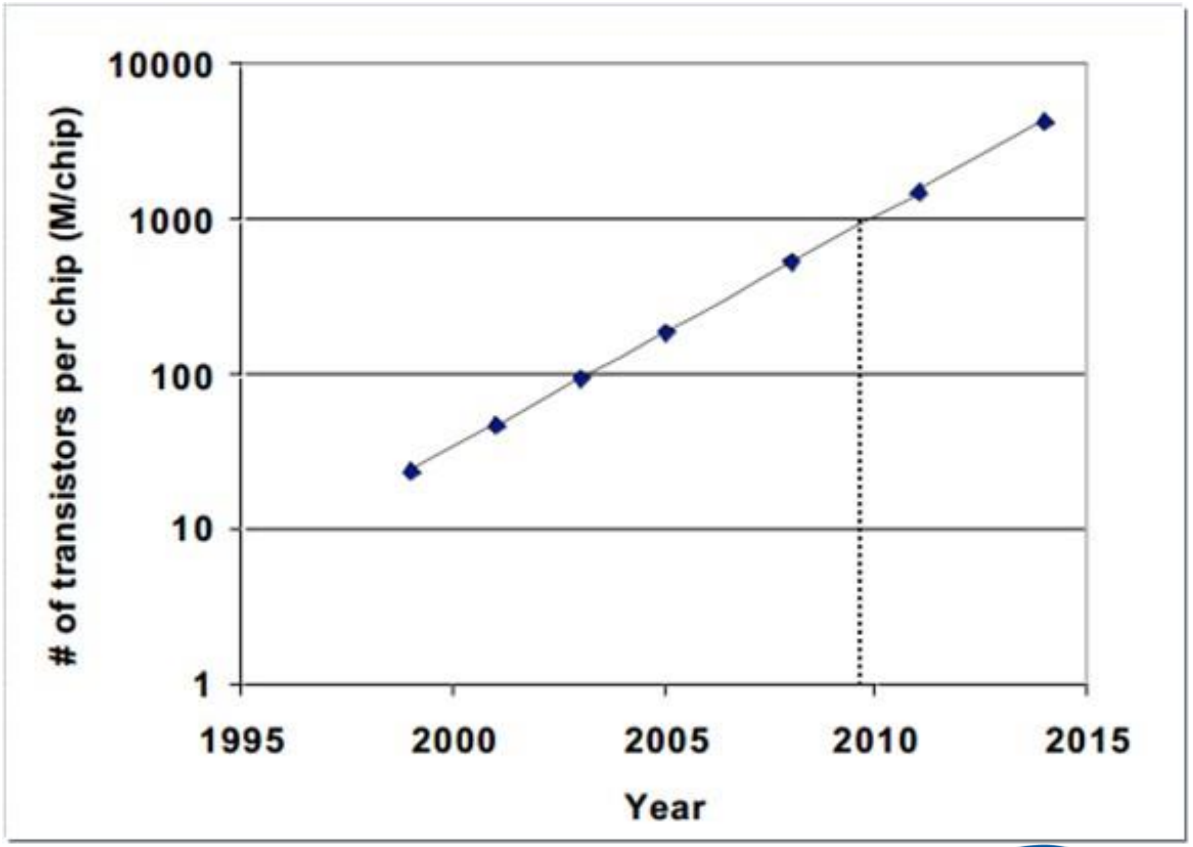


Three phases:

- •Bit-level
- •Instruction-level
- •Thread-level

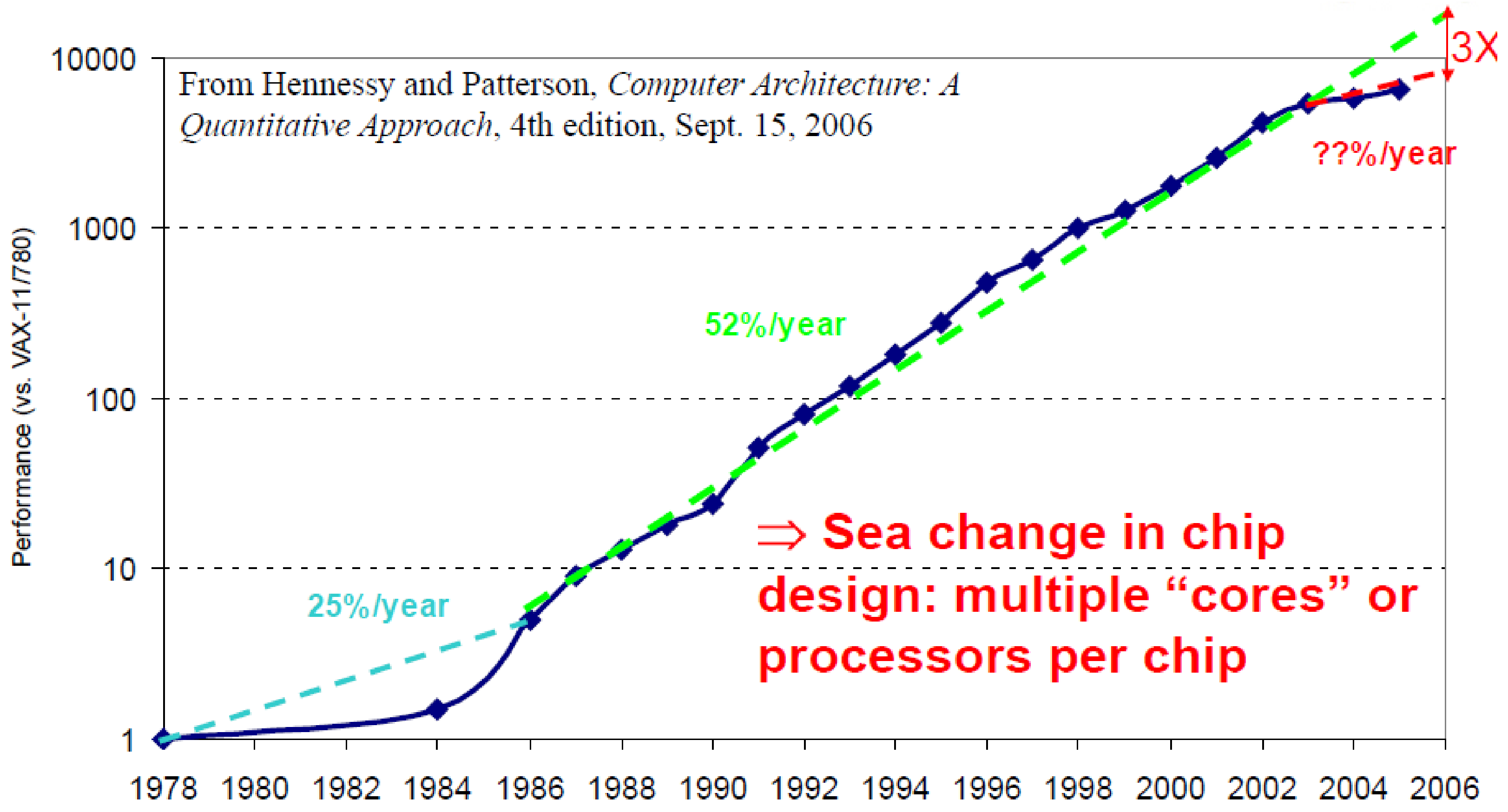
VLSI Technology Trends

- Intel announced that they have reached 1.7 billion with Itanium processor
- Gigascale Integration (GSI) = 1 billion transistors per chip



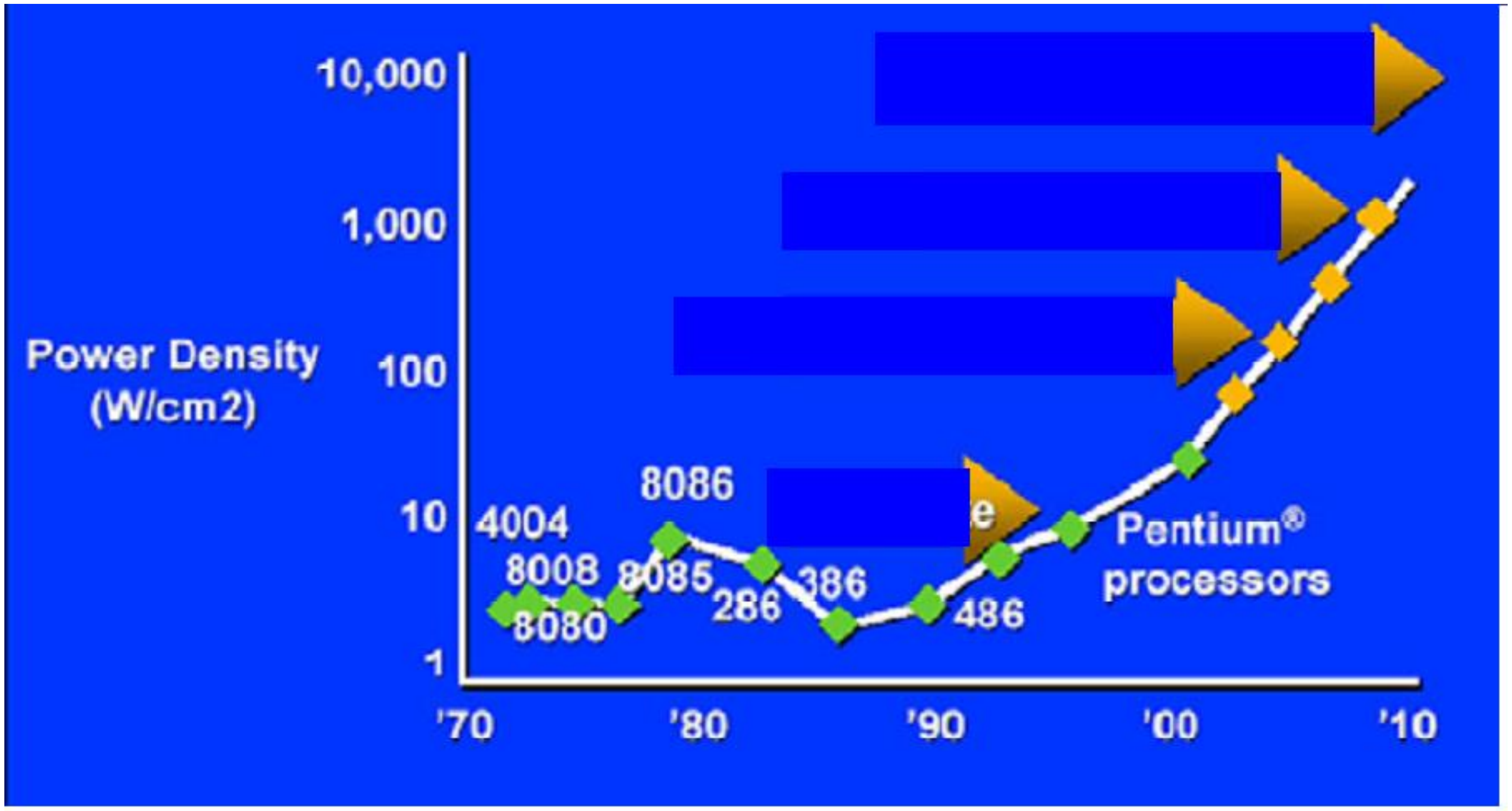
http://users.ece.gatech.edu/~jeff/ece4420/technology.pdf

The Rate of Single-Thread Performance Improvement has Decreased



- VAX: 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Impact of Power Density on the Microprocessor Industry



Pat Gelsinger, ISSCC 2001

The development tendency is not higher clock rates, but multiple cores per die

Recent Intel Processors

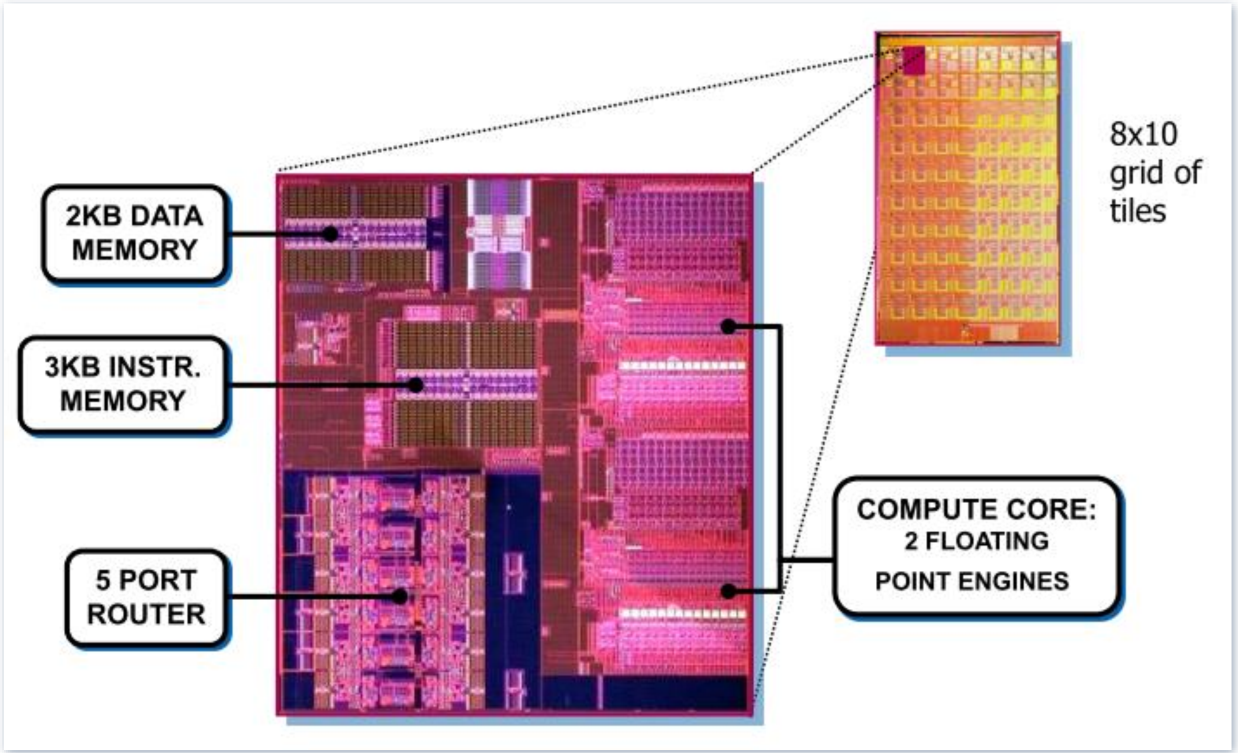
Processors	Year	Fabrication(nm)	Clock(GHz)	Power(W)
Pentium 4	2000	180	1.80-4.00	35-115
Pentium M	2003	90/130	1.00-2.26	5-27
Core 2 Duo	2006	65	2.60-2.90	10-65
Core 2 Quad	2006	65	2.60-2.90	45-105
Core i7(Quad)	2008	45	2.93-3.60	95-130
Core i5(Quad)	2009	45	3.20-3.60	73-95
Pentium Dual-Core	2010	45	2.80-3.33	65-130
Core i3(Duo)	2010	32	2.93-3.33	18-73
2nd Gen i3(Duo)	2011	32	2.50-3.40	35-65
2nd Gen i5(Quad)	2011	32	3.10-3.80	45-95
2nd Gen i7(Quad/Hexa)	2011	32	3.80-3.90	65-130
3rd Gen i3(Duo)	2012	22/32	2.80-3.40	35-55
3rd Gen i5(Quad)	2012	22/32	3.20-3.80	35-77
3rd Gen i7(Quad/Hexa)	2012	22/32	3.70-3.90	45-77
Xeon E5(8-cores)	2013	22	1.80-2.90	60-130
Xeon Phi(60-cores)	2013	22	1.10	300

- *"We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry."* Intel President Paul Otellini, IDF 2005

華中科技大學

Intel's Many Core and Multi-core

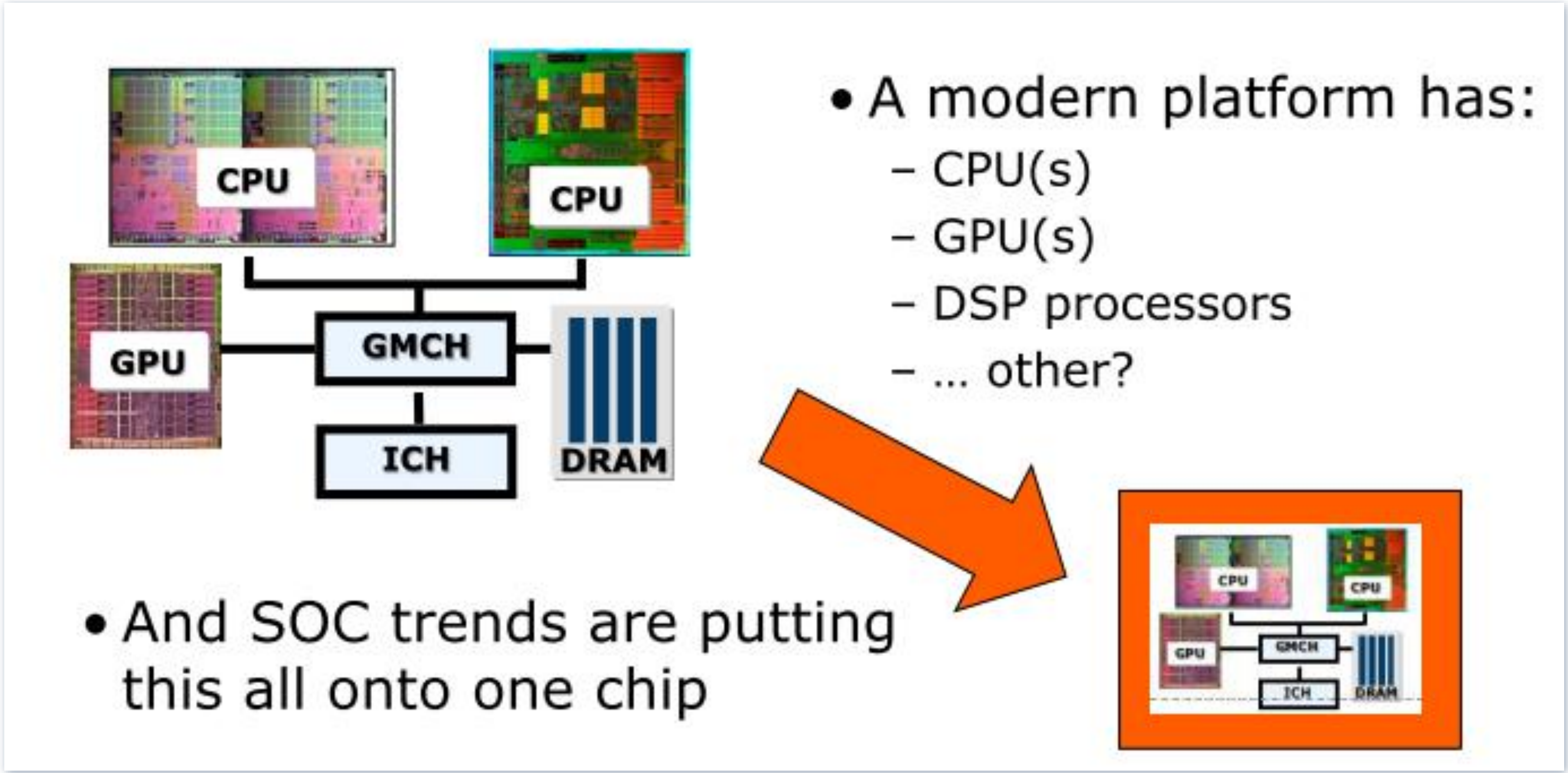
- Intel 80-core TeraScale Processor (Vangal et al. 2008)
  - developed a solver (single precision) for this chip that ran at 1 TFLOP with only 97 Watts



Source: Tim Mattson, Intel Labs

Trends are putting all onto one chip

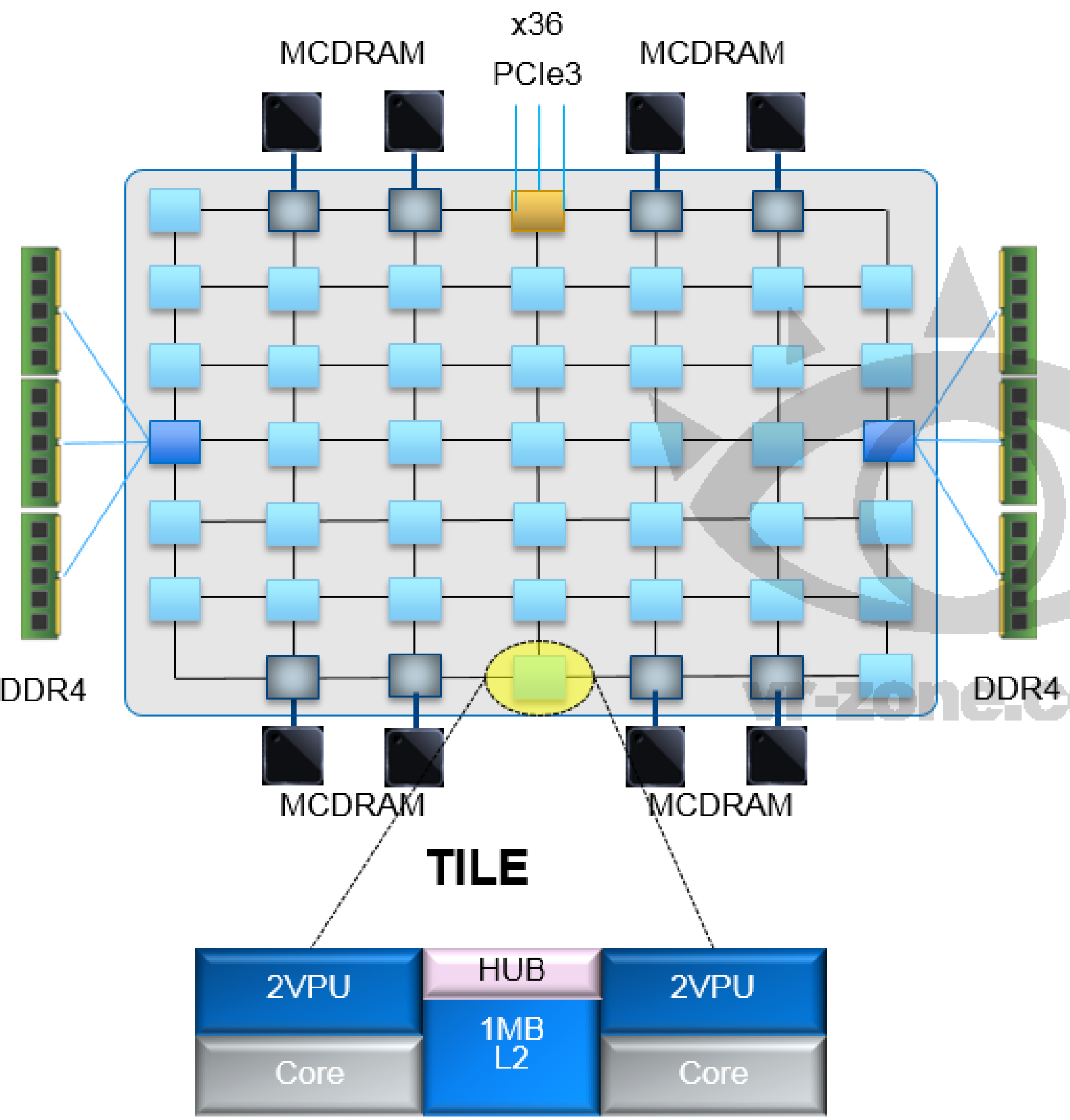
- The future belongs to heterogeneous, many core SOC as the standard building block of computing
- SOC = system on a chip



- A modern platform has:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?
- And SOC trends are putting this all onto one chip

Source: Tim Mattson, Intel Labs

Intel 72-core x86 Knights Landing CPU for exascale supercomputing



Up to 72 Intel Architecture cores based on Silvermont (Intel® Atom processor)

- Four threads/core
- Two 512b vector units/core
- Up to 3x single thread performance improvement over KNC generation

Full Intel® Xeon processor ISA compatibility through AVX-512 (except TSX)

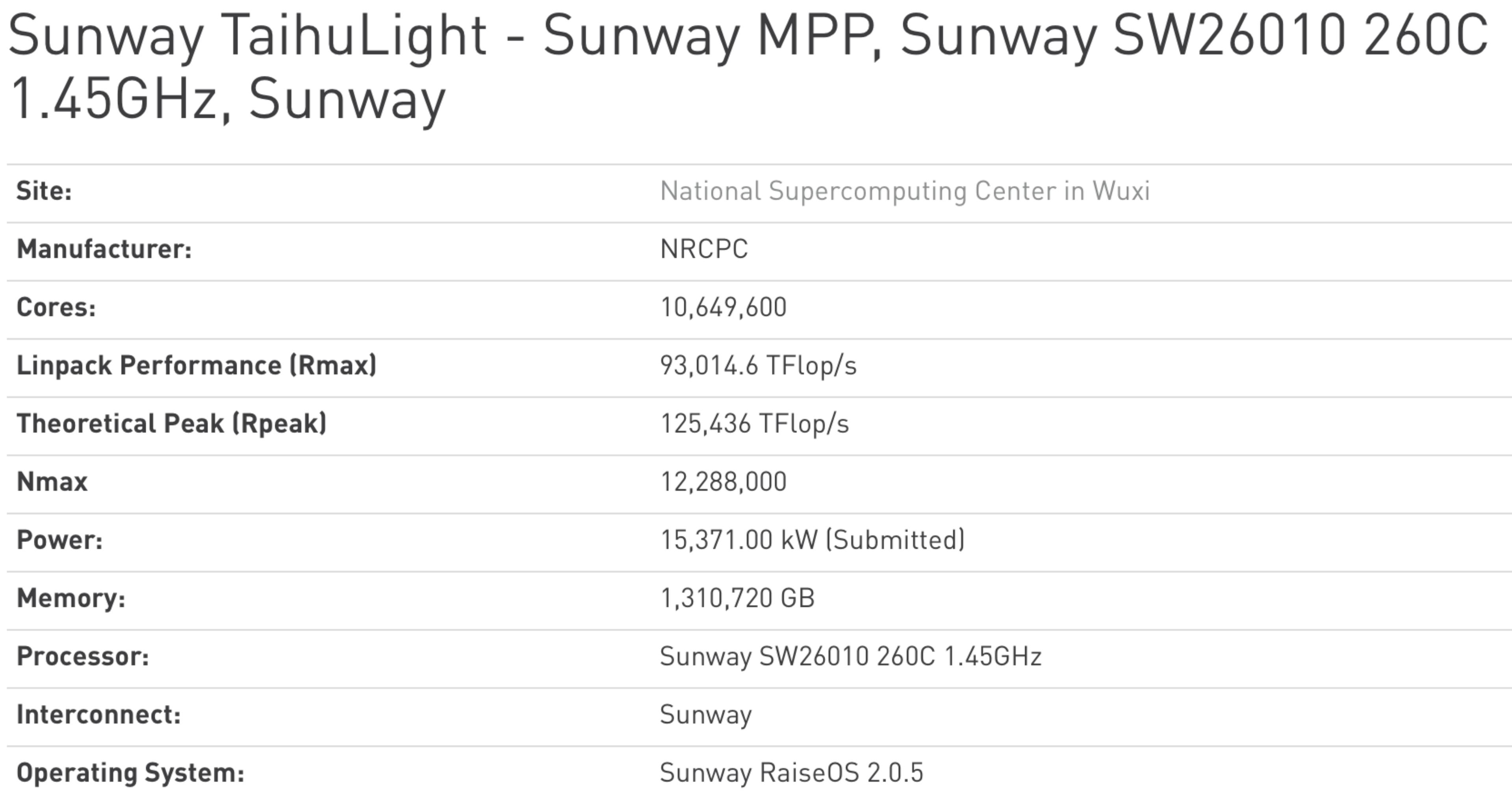
6 channels of DDR4 2400 MHz -up to 384GB

36 lanes PCI Express* Gen 3

8/16GB of high-bandwidth on-package MCDRAM memory >500GB/sec

200W TDP

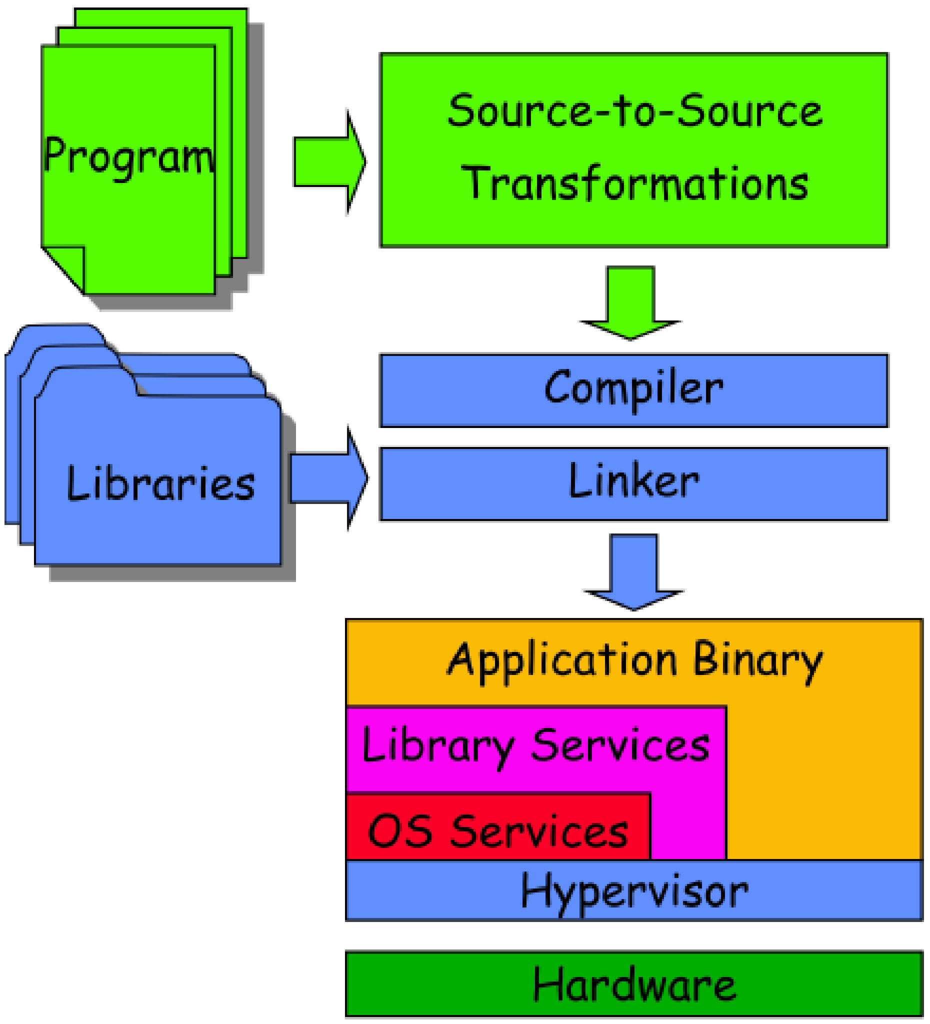
Large-Scale Computing Systems



Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway

Site:	National Supercomputing Center in Wuxi
Manufacturer:	NRCPC
Cores:	10,649,600
Linpack Performance (Rmax)	93,014.6 TFlop/s
Theoretical Peak (Rpeak)	125,436 TFlop/s
Nmax	12,288,000
Power:	15,371.00 kW (Submitted)
Memory:	1,310,720 GB
Processor:	Sunway SW26010 260C 1.45GHz
Interconnect:	Sunway
Operating System:	Sunway RaiseOS 2.0.5

Execution is *not* just about hardware



- The VAX fallacy
  - Produce one instruction for every high-level concept
  - Absurdity: polynomial multiply
    - Single hardware instruction
    - But Why? Is this really faster??
- RISC Philosophy
  - Full System Design
  - Hardware mechanisms viewed in context of complete system
  - Cross-boundary optimization
- Modern programmer does not see assembly language
  - Many do not even see “low-level” languages like “C”

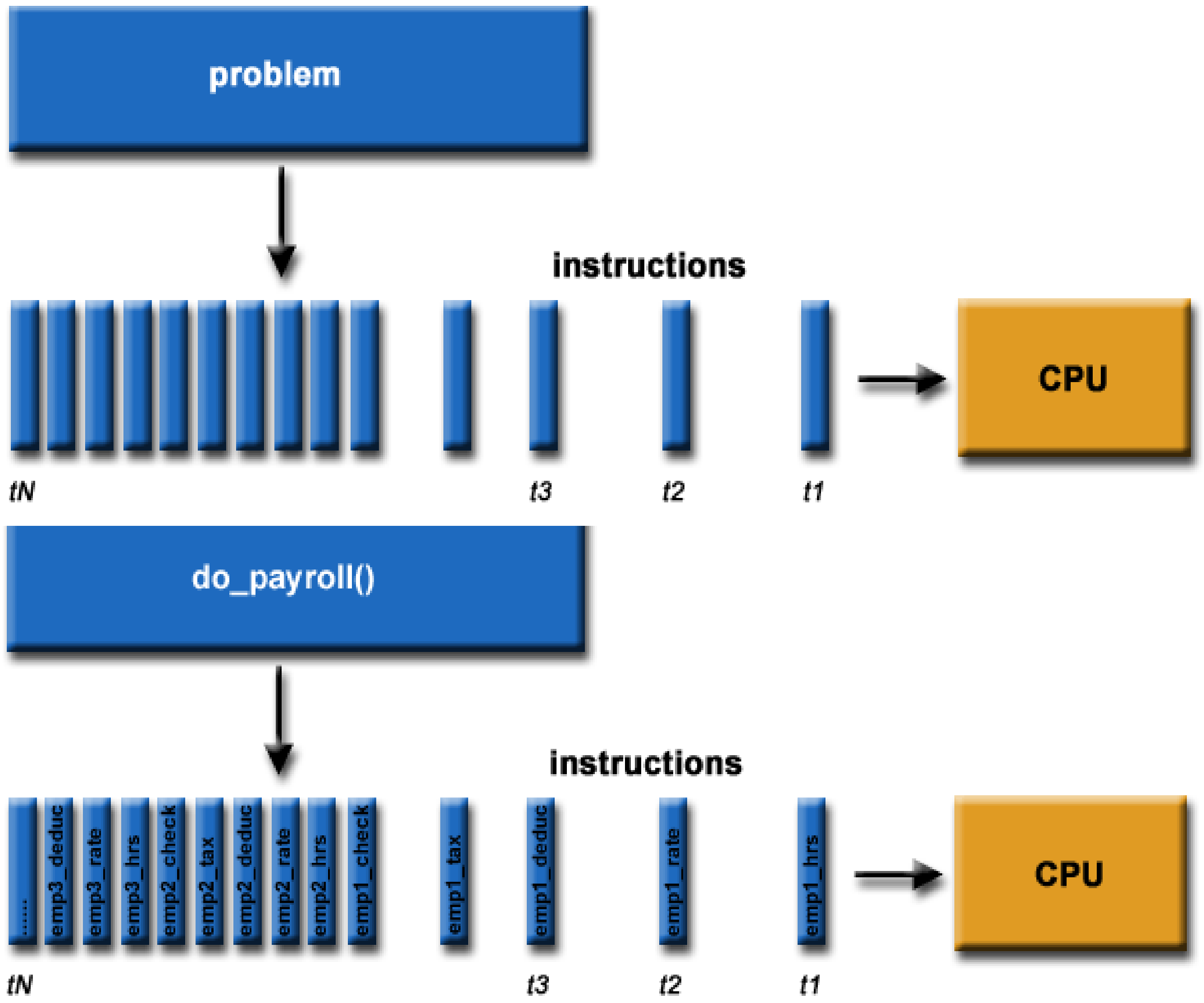
Why parallel programming

WHAT IS PARALLEL PROGRAMMING?

What is Parallel Computing?

- ***Traditionally***, software has been written for serial computation
  - To be run on a single computer having a single CPU
  - A problem is broken into a discrete series of instructions
  - Instructions are executed one after another
  - Only one instruction may execute at any moment in time

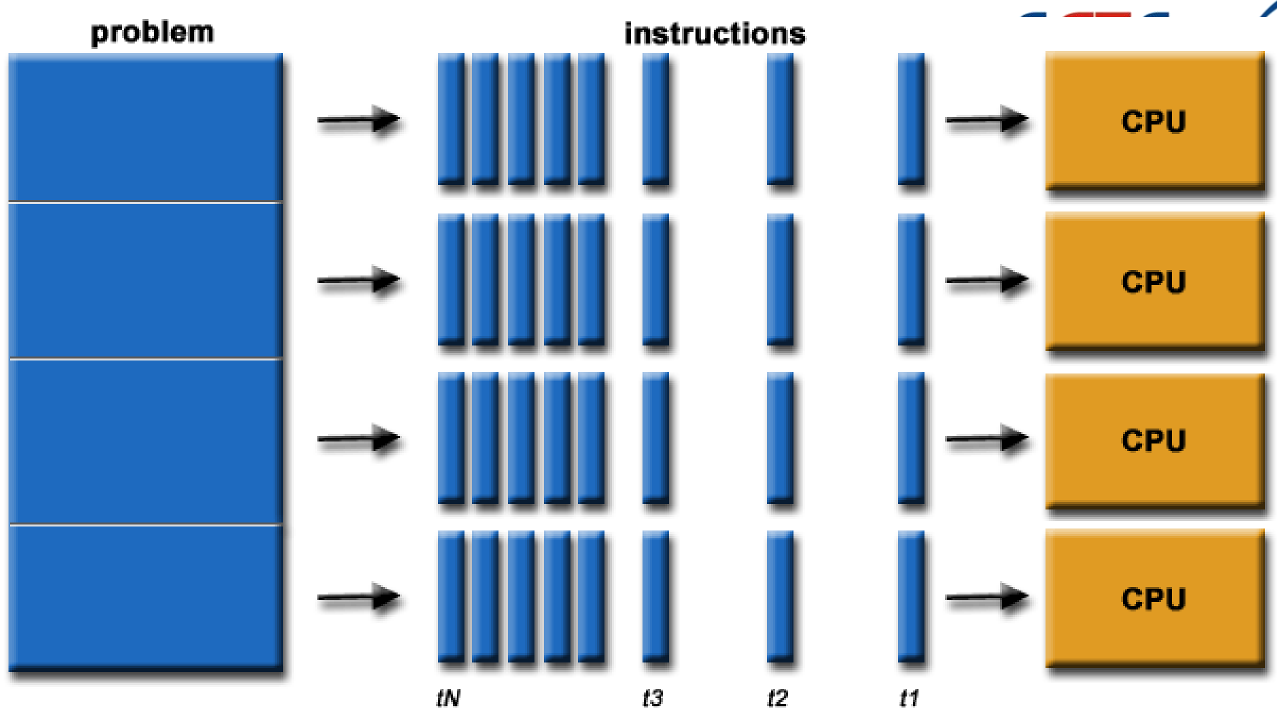
For example

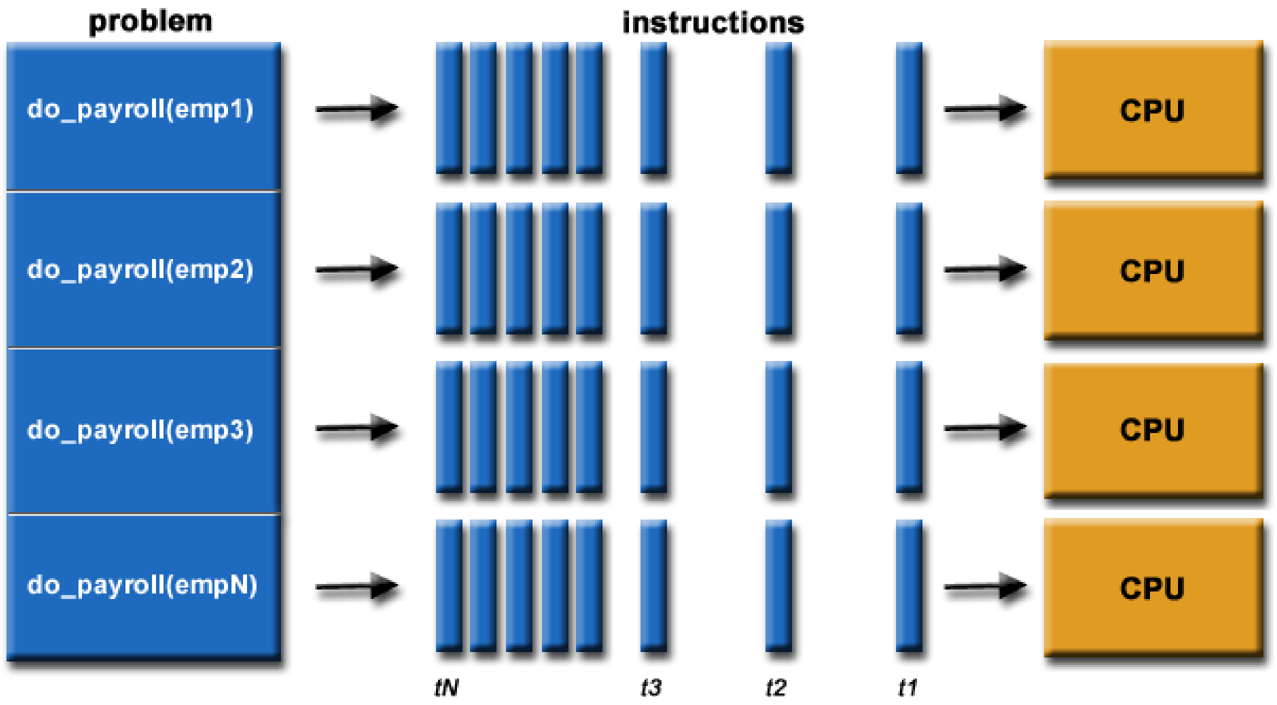

problem
instructions
CPU
tN
t3
t2
t1
do_payroll()
instructions
CPU
......
emp3_deduc
emp3_rate
emp3_hrs
emp2_check
emp2_tax
emp2_deduc
emp2_rate
emp2_hrs
emp1_check
emp1_tax
emp1_deduc
emp1_rate
emp1_hrs
tN
t3
t2
t1

Parallel Computing

- In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different CPUs

Example


problem
instructions
CPU
CPU
CPU
CPU
tN
t3
t2
t1


problem
instructions
do_payroll(emp1)
do_payroll(emp2)
do_payroll(emp3)
do_payroll(empN)
CPU
CPU
CPU
CPU
tN
t3
t2
t1

Example

- The compute resources might be
  - A single computer with multiple processors
  - An arbitrary number of computers connected by a network
  - A combination of both
- The computational problem should be able to
  - Be broken apart into discrete pieces of work that can be solved simultaneously
  - Execute multiple program instructions at any moment in time
  - Be solved in less time with multiple compute resources than with a single compute resource

Speedup

Goal of applications in using parallel machines: Speedup

$$\text{Speedup}(p\ \text{processors}) = \frac{\text{Performance}(p\ \text{processors})}{\text{Performance}(1\ \text{processor})}$$

For a fixed problem size (input data set), performance = 1/time

$$\text{Speedup}_{\text{fixed problem}}(p\ \text{processors}) = \frac{\text{Time}(1\ \text{processor})}{\text{Time}(p\ \text{processors})}$$

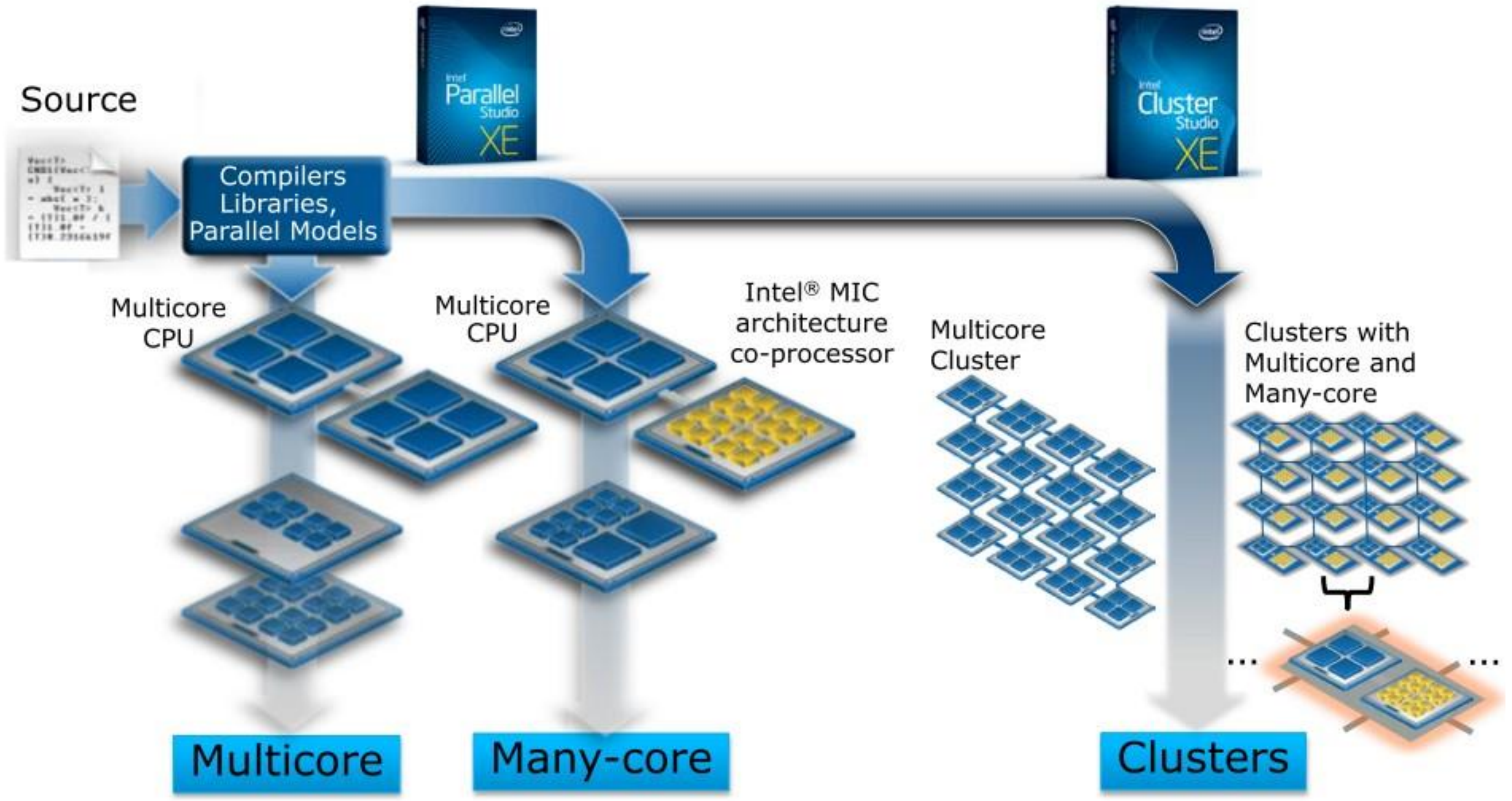
Commercial Computing

- Databases, online-transaction processing, decision support, data mining, data warehousing, machine learning, DNN ...
- Also relies on parallelism for high end
  - Scale not so large, but use much more wide-spread
  - Computational power determines scale of business that can be handled
- TPC benchmarks (TPC-C order entry, TPC-D decision support)
  - Explicit scaling criteria provided
  - Size of enterprise scales with size of system
  - Problem size no longer fixed as *p* increases, so throughput is used as a performance measure (transactions per minute or tpm)

Why parallel programming

WHY DO WE NEED PARALLEL PROGRAMMING?

Now we can get: single-source approach to multi- and many-core

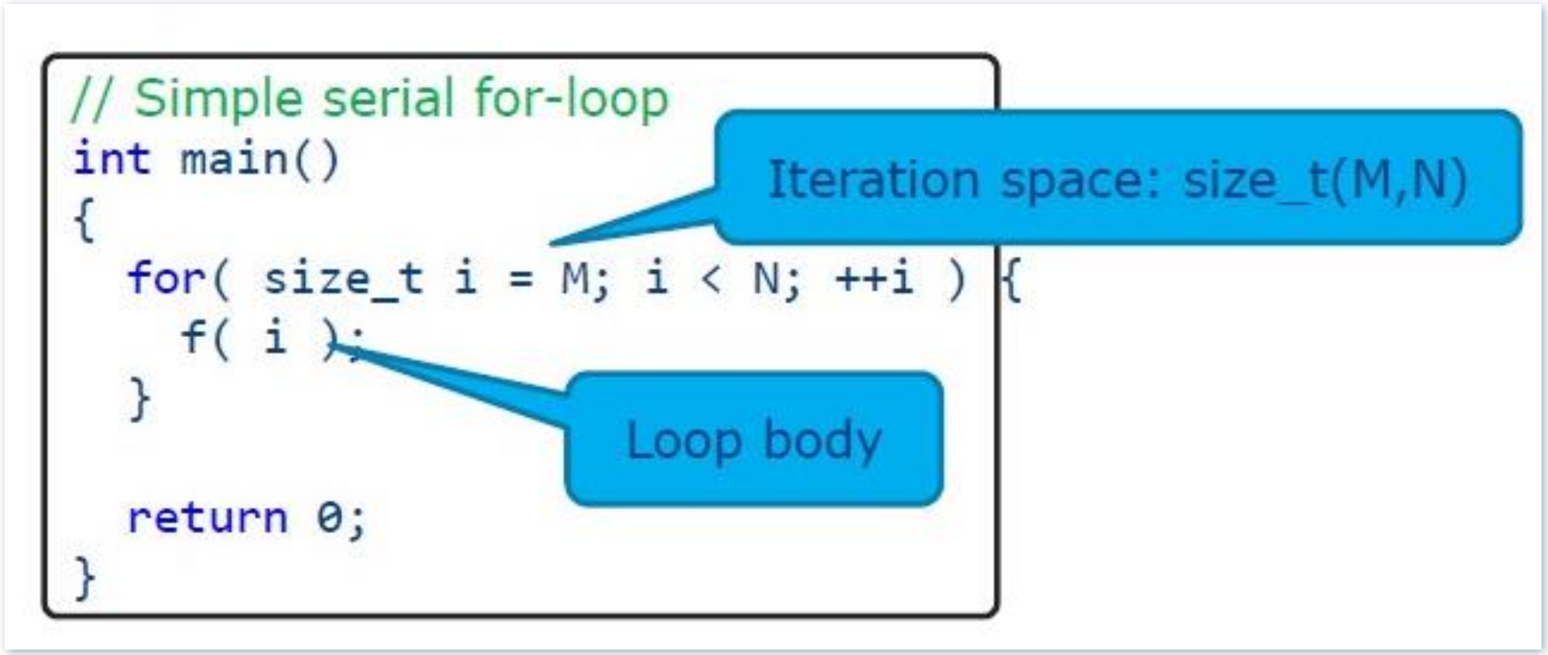


However, the Parallelizing Compilers

- After 30 years of intensive research
  - only limited success in parallelism detection and program transformations
    - instruction-level parallelism at the *basic-block level can be detected*
    - parallelism in *nested for-loops containing arrays with simple index expressions can be analyzed*
    - analysis techniques, such as data dependence analysis, pointer analysis, flow sensitive analysis, abstract interpretation, ... when applied across procedure boundaries often take far too long and tend to be fragile, i.e., can break down after small changes in the program
  - instead of training compilers to recognize parallelism, ***people have been trained to write programs that parallelize***

A simple example

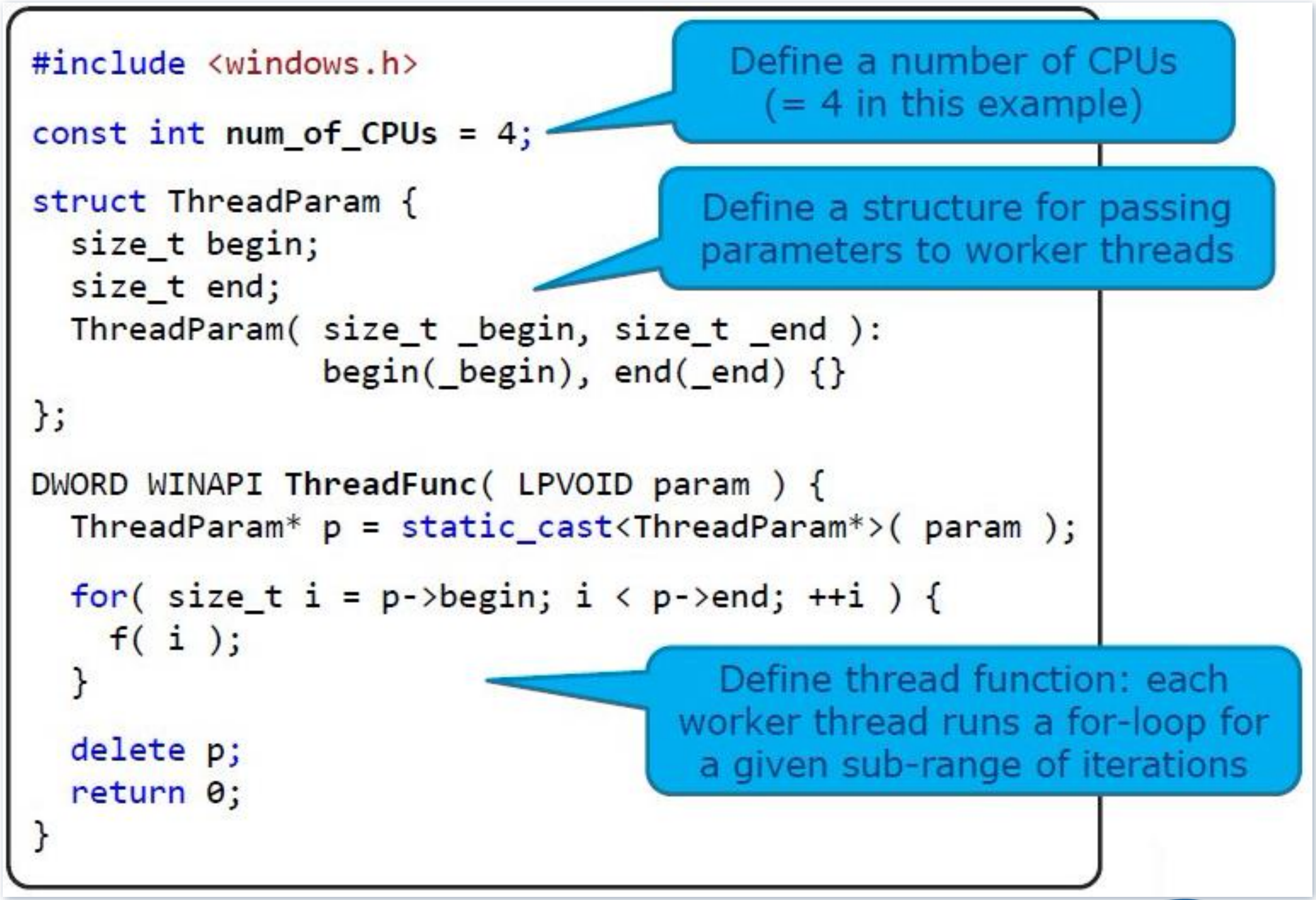
- Loop is a simple example of a code region that can benefit from parallelism
- Let’s look at one of the possible implementations of parallel for-loop



```
// Simple serial for-loop
int main()
{
  for( size_t i = M; i < N; ++i ) {
    f( i );
  }

  return 0;
}
```

Things to Consider in Creating a Parallelized "for-loop"

- Step 1

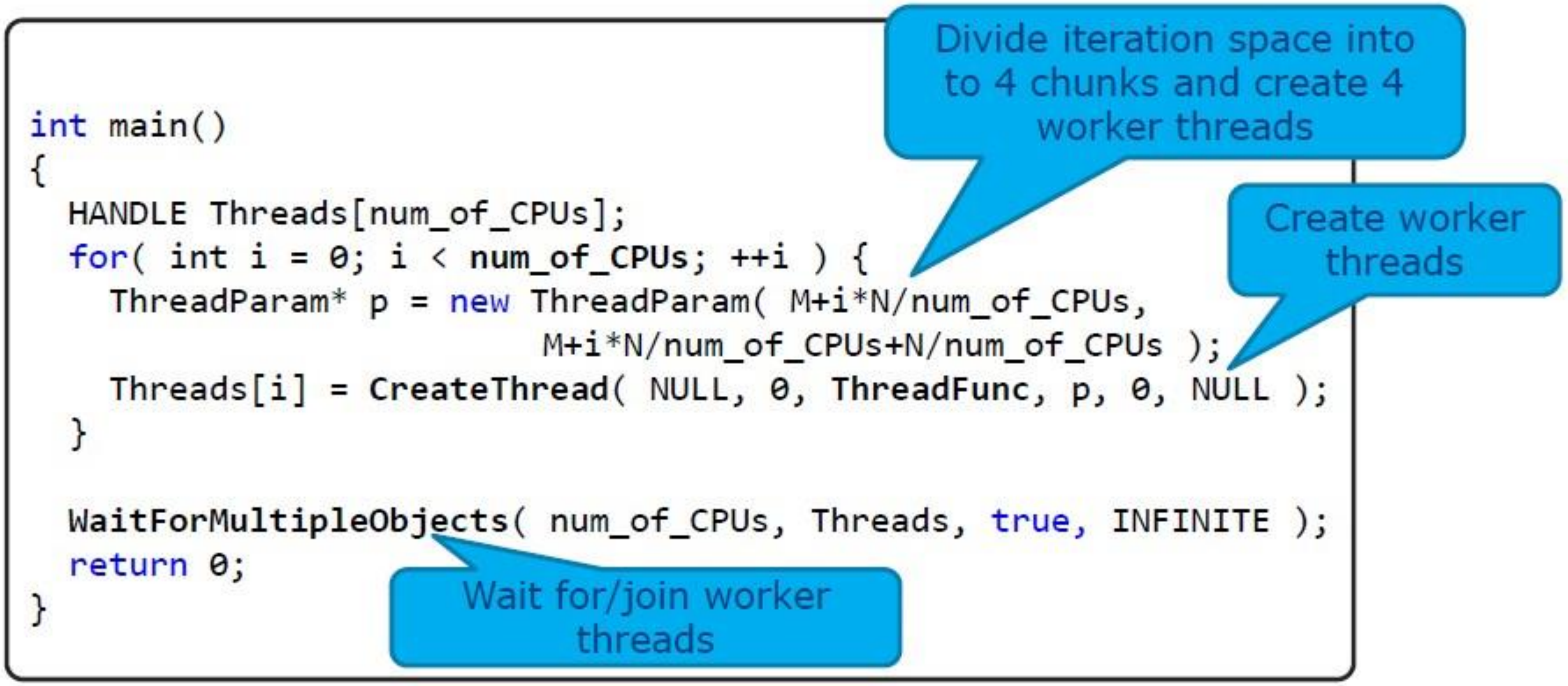


```
#include <windows.h>

const int num_of_CPUs = 4;

struct ThreadParam {
  size_t begin;
  size_t end;
  ThreadParam( size_t _begin, size_t _end ):
               begin(_begin), end(_end) {}
};

DWORD WINAPI ThreadFunc( LPVOID param ) {
  ThreadParam* p = static_cast<ThreadParam*>( param );

  for( size_t i = p->begin; i < p->end; ++i ) {
    f( i );
  }

  delete p;
  return 0;
}
```

Define a number of CPUs (= 4 in this example)
Define a structure for passing parameters to worker threads
Define thread function: each worker thread runs a for-loop for a given sub-range of iterations

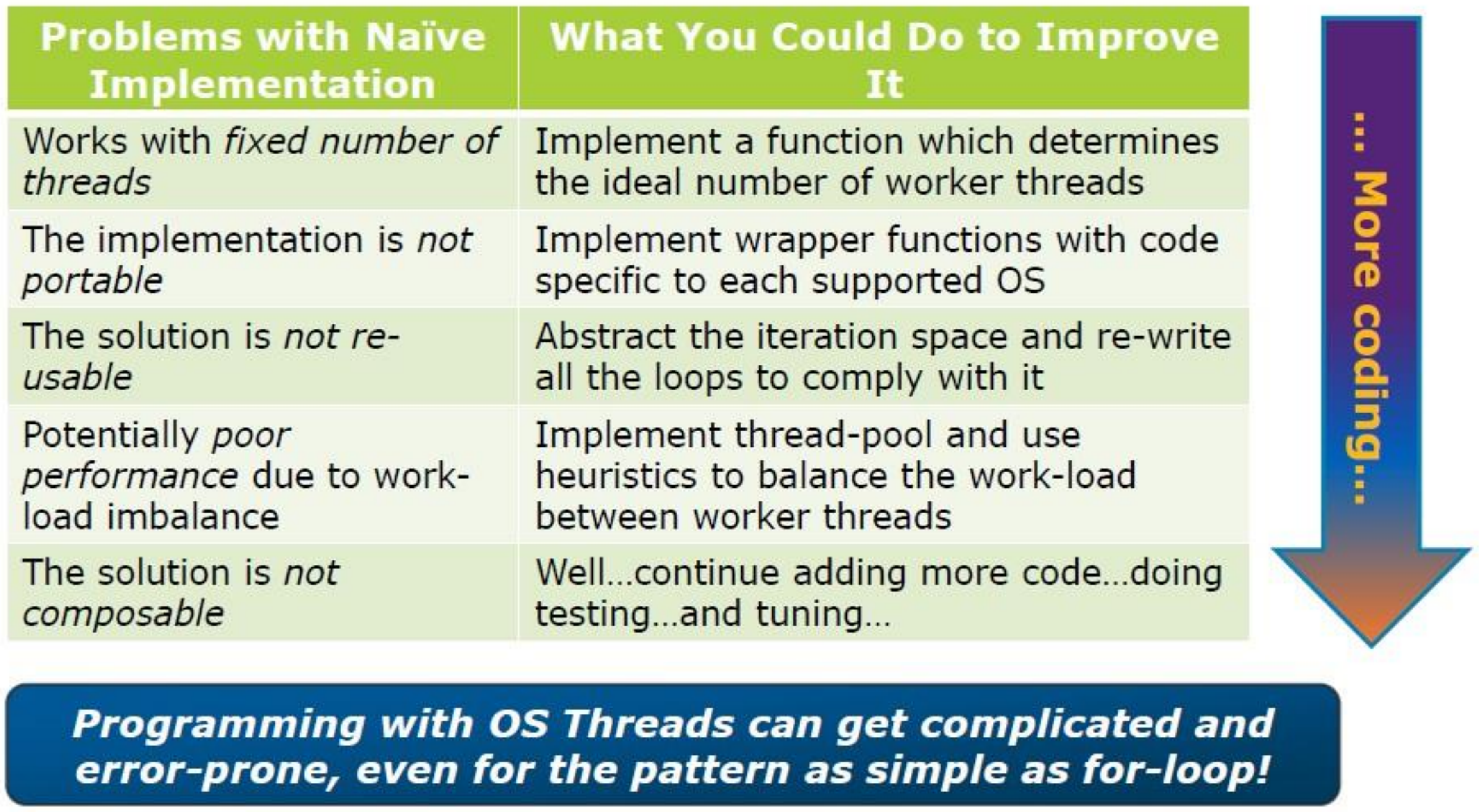
Things to Consider in Creating a Parallelized “for-loop”

- Step 2


Divide iteration space into to 4 chunks and create 4 worker threads
Create worker threads
Wait for/join worker threads

```
int main()
{
  HANDLE Threads[num_of_CPUs];
  for( int i = 0; i < num_of_CPUs; ++i ) {
    ThreadParam* p = new ThreadParam( M+i*N/num_of_CPUs,
                                      M+i*N/num_of_CPUs+N/num_of_CPUs );
    Threads[i] = CreateThread( NULL, 0, ThreadFunc, p, 0, NULL );
  }

  WaitForMultipleObjects( num_of_CPUs, Threads, true, INFINITE );
  return 0;
}
```

Many Ways to Improve Naïve Implementation


… More coding…

Problems with Naïve Implementation	What You Could Do to Improve It
Works with *fixed number of threads*	Implement a function which determines the ideal number of worker threads
The implementation is *not portable*	Implement wrapper functions with code specific to each supported OS
The solution is *not re-usable*	Abstract the iteration space and re-write all the loops to comply with it
Potentially *poor performance* due to work-load imbalance	Implement thread-pool and use heuristics to balance the work-load between worker threads
The solution is *not composable*	Well…continue adding more code…doing testing…and tuning…

Programming with OS Threads can get complicated and error-prone, even for the pattern as simple as for-loop!

Parallel Programming Complexity

- Enough parallelism? (Amdahl’s Law)
- Granularity
- Locality
- Load balance
- Coordination and Synchronization
- **All of these things makes parallel programming even harder than sequential programming**

Parallel Compared to Sequential Programming

- Has different costs, different advantages
- Requires different, unfamiliar algorithms
- Must use different abstractions
- More complex to understand a program’s behavior
- More difficult to control the interactions of the program’s components
- Knowledge/tools/understanding more primitive

Is it really harder to "think" in parallel?

- Some would argue it is more natural to think in parallel…
- … and many examples exist in daily life
  - House construction -- parallel tasks, wiring and plumbing performed at once *(independence)*, but framing must precede wiring *(dependence)*
    - ✓ Similarly, developing large software systems
  - Assembly line manufacture - pipelining, many instances in process at once
  - Call center - independent calls executed simultaneously (data parallel)
  - "Multi-tasking" – all sorts of variations

Example

- Compute *n* values and add them together
- Serial solution

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

Example (cont'd)

- We have *p* cores, *p* much smaller than *n*
- Each core performs a partial sum of approximately *n*/p values

```
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;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

Example (cont'd)

- After each core completes execution of the code, a private variable my_sum contains the sum of the values computed by its calls to Compute_next_value

- Ex., 8 cores, n = 24, then the calls to Compute_next_value return:

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

Example (cont'd)

- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated "master" core which adds the final result

```
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;
}
```

Example (cont'd)

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Global sum

$8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95$

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

But wait!

There's a much better way to compute the global sum.

Better parallel algorithm

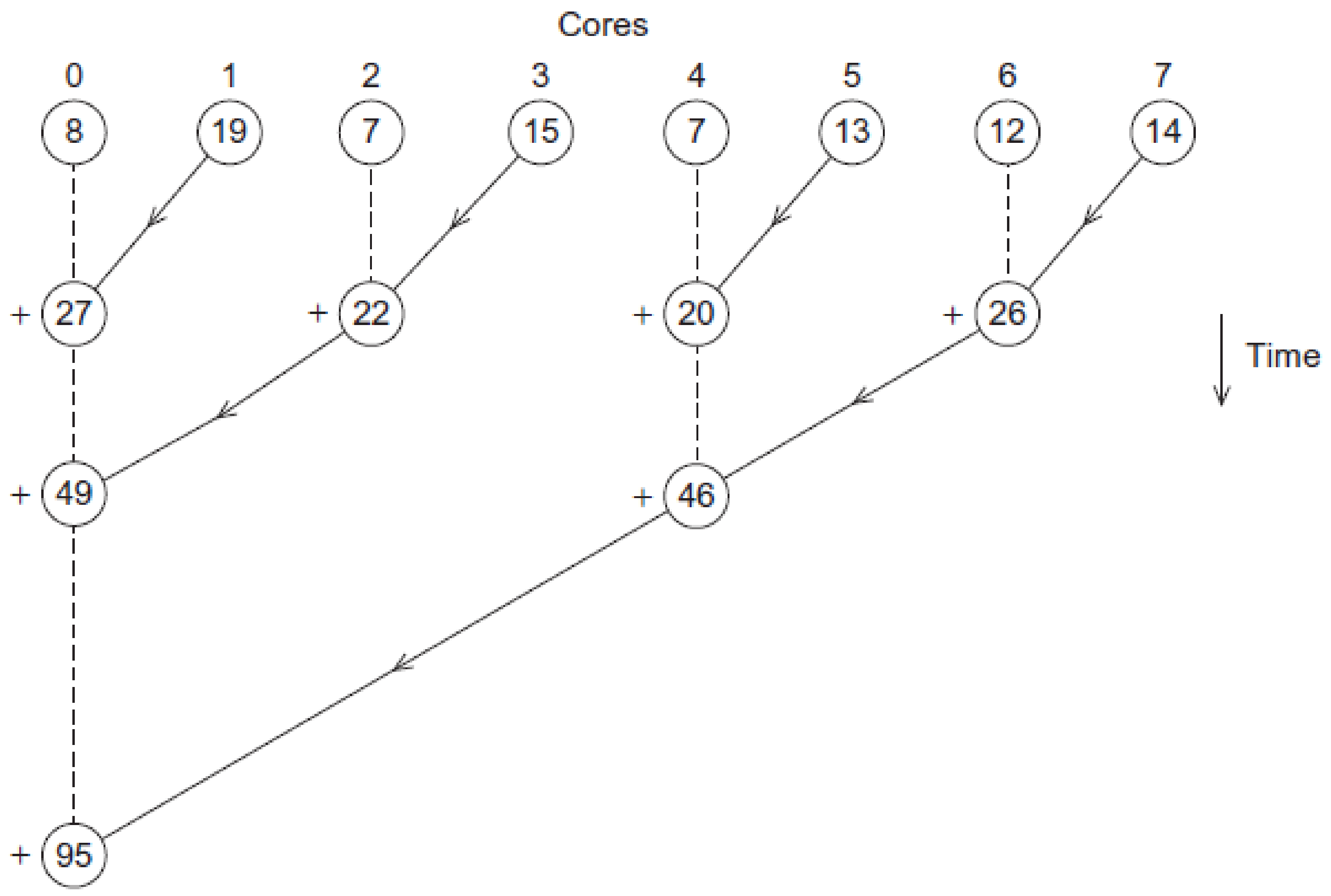
- Don’t make the master core do all the work
- Share it among the other cores
- Pair the cores so that core 0 adds its result with core 1’s result
- Core 2 adds its result with core 3’s result, etc
- Work with odd and even numbered pairs of cores

Better parallel algorithm (cont'd)

- Repeat the process now with only the evenly ranked cores
- Core 0 adds result from core 2
- Core 4 adds the result from core 6, etc.

- Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result

Multiple cores forming a global sum



Analysis

- In the first example, the master core performs 7 receives and 7 additions
- In the second example, the master core performs 3 receives and 3 additions
- The improvement is more than a factor of 2!

Analysis (cont'd)

- The difference is more dramatic with a larger number of cores
- If we have 1000 cores
  - The first example would require the master to perform 999 receives and 999 additions
  - The second example would only require 10 receives and 10 additions
- That's an improvement of almost a factor of 100!

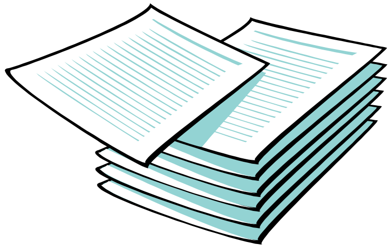
How do we write parallel programs?

- Task parallelism
  - Partition various tasks carried out solving the problem among the cores
- Data parallelism
  - Partition the data used in solving the problem among the cores
  - Each core carries out similar operations on its part of the data

Professor A

15 questions

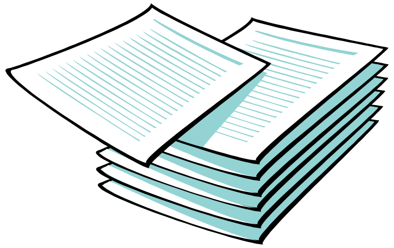
300 exams



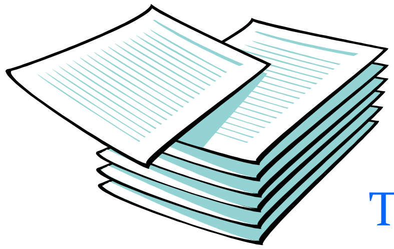
Professor A’s grading assistants

Division of work – data parallelism

TA#1

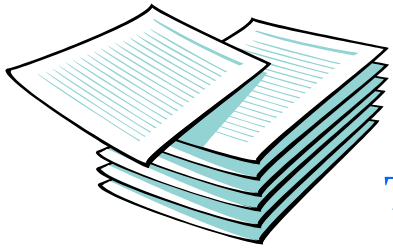


100 exams



TA#3

100 exams



TA#2

100 exams

Division of work – task parallelism

TA#1

Questions 1 - 5

Questions 6 - 10

TA#2

Questions 11 - 15

TA#3

Division of work – data parallelism

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

Division of work – task parallelism

```
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;
}
```

Tasks

1) Receiving
2) Addition

Coordination

- Cores usually need to coordinate their work
- Communication – one or more cores send their current partial sums to another core
- Load balancing – share the work evenly among the cores so that one is not heavily loaded
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest

Why parallel programming

DISTRIBUTED COMPUTING

Parallel vs Distributed Computing

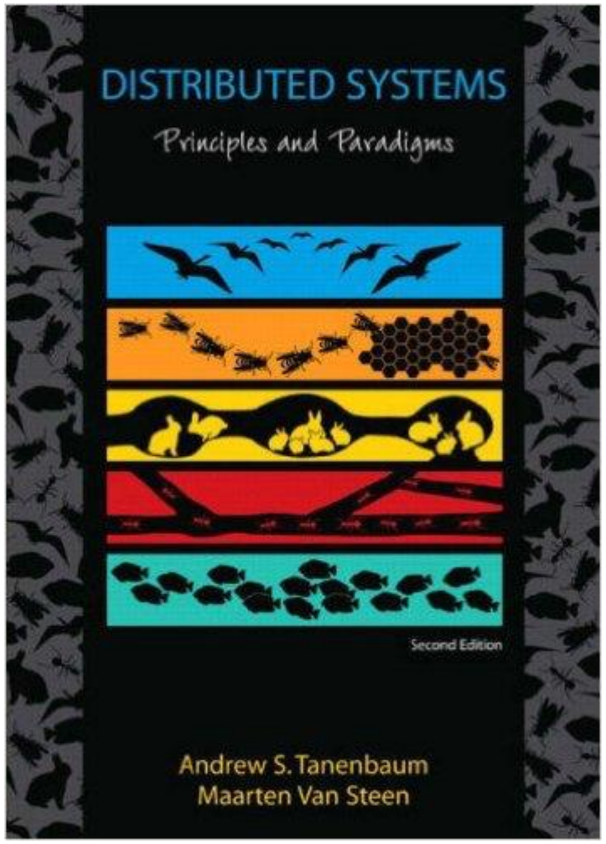
- **Parallel computing** splits a single application up into tasks that are executed at the same time and is more like a **top-down** approach
- Parallel computing is about decomposition
  - how we can perform a single application concurrently
  - how we can divide a computation into smaller parts which may potentially be executed in parallel
- Parallel computing consider how to reach a maximum degree of concurrency
  - Scientific computing

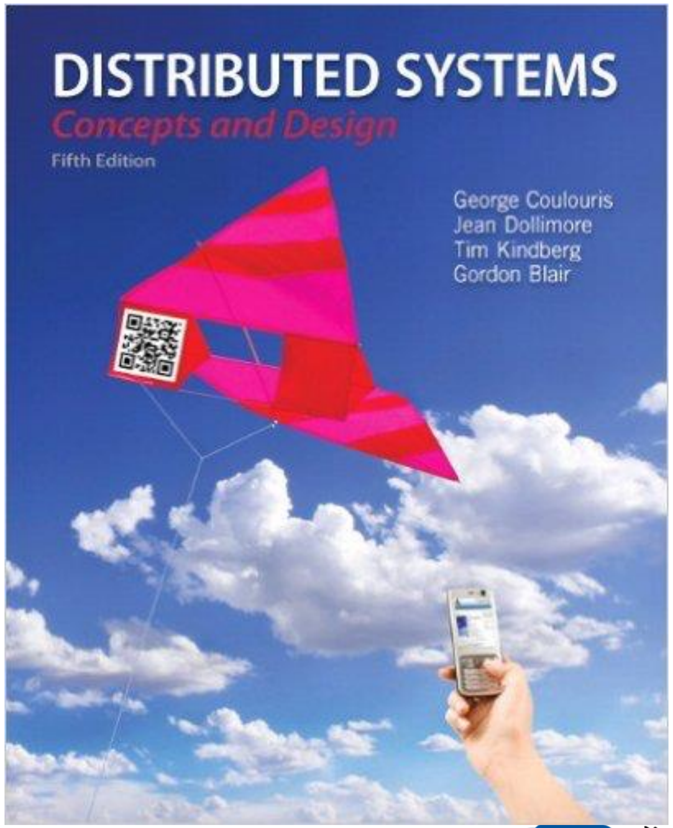
Parallel vs Distributed Computing

- **Distributed computing** considers a single application which is executed as a whole but at different locations and is more like a **bottom-up** approach
- Distributed computing is about composition
  - What happens if many distributed processes interact with each other
  - If a global function can be achieved although there is no global time or state
- Distributed computing considers reliability and availability
  - Information/resource sharing

Distributed Systems

A collection of independent computers that appears to its users as a single coherent system

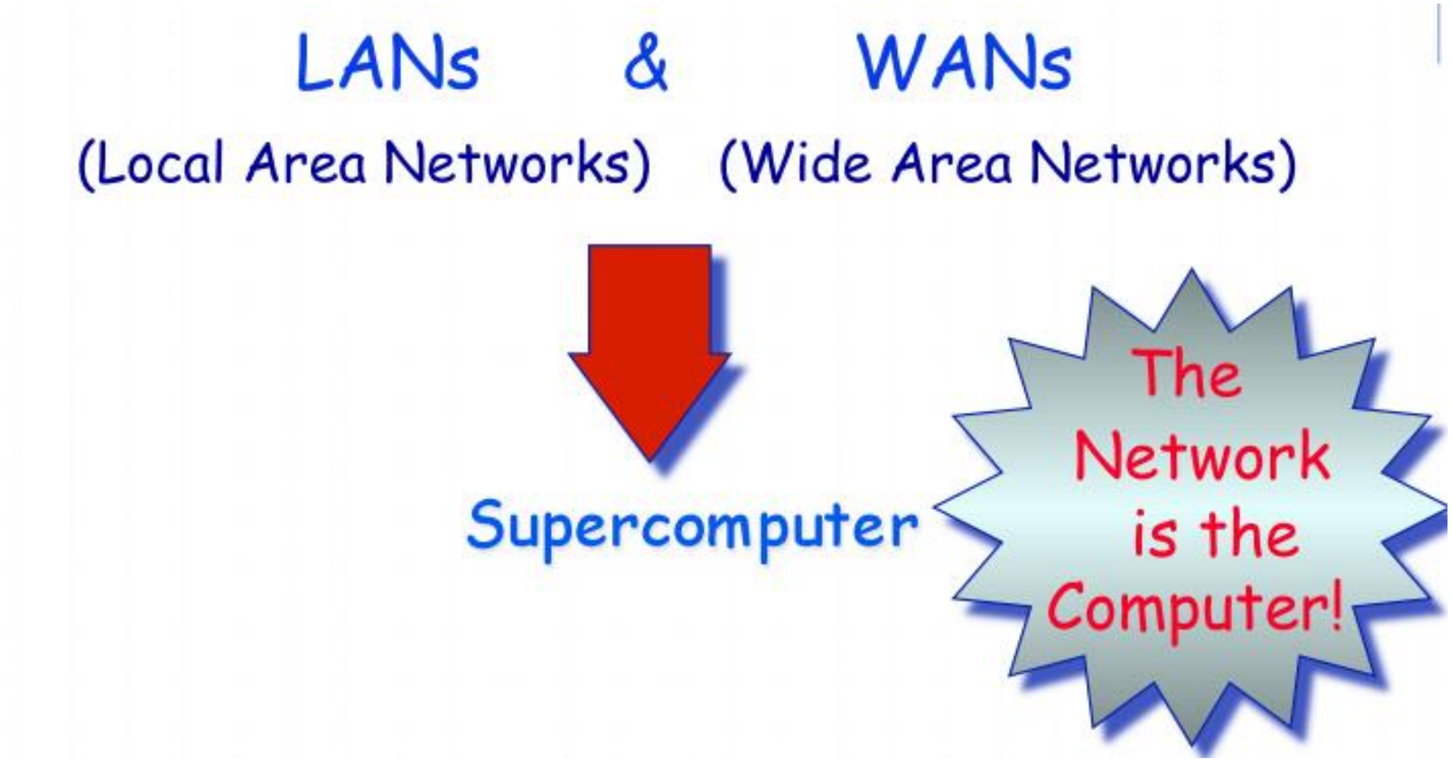




Parallel vs Distributed Computing

- The differences are now blurred, especially after the introduction of grid computing and cloud computing
- The two related fields have many things in common
  - Multiple processors
  - Networks connecting the processors
  - Multiple computing activities and processes
  - Input/output data distributed among processors

The Network is the Computer


LANs & WANs
(Local Area Networks) (Wide Area Networks)
Supercomputer
The Network is the Computer!

"when the network is as fast as the computer's internal links, the machine disintegrates across the net into a set of special purpose appliances"

Grid Computing

- **Grid computing** is the combination of computer resources from multiple administrative domains applied to a common task, usually to a scientific, technical or business problem that requires a great number of computer processing cycles or the need to process large amounts of data
- It is a form of distributed computing whereby a “super and virtual computer” is composed of a cluster of networked loosely coupled computers acting in concert to perform very large tasks
- This technology has been applied to computationally intensive scientific, mathematical, and academic problems, and used in commercial enterprise data intensive applications

Cloud Computing

- A style of computing where massively scalable IT-related capabilities are provided “as a service” using **Internet technologies** to multiple external customers
- Cloud computing describes a new supplement, consumption and delivery model for IT services based on the Internet, and it typically involves the provision of dynamically scalable and often **virtualized resources (storage, platform, infrastructure, and software) as a service** over the Internet

Conclusion

- Certainly, it is no longer sufficient for even basic programmers to acquire only the traditional, conventional sequential programming skills
- Need for imparting a broad-based skill set in PDC technology at various levels in the educational fabric woven by Computer Science (CS) and Computer Engineering (CE) programs as well as related computational disciplines

References

- The content expressed in this chapter comes from
  - UC Berkeley open course (http://parlab.eecs.berkeley.edu/2010bootcampagenda)
  - Carnegie Mellon University’s public course, Parallel Computer Architecture and Programming (CS 418) (http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/public/lectures/)
  - Livermore Computing Center’s training materials, Introduction to Parallel Computing (https://computing.llnl.gov/tutorials/parallel_comp/)