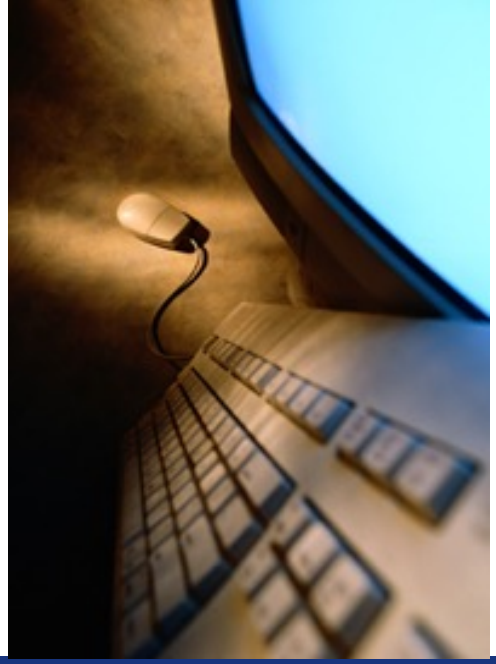
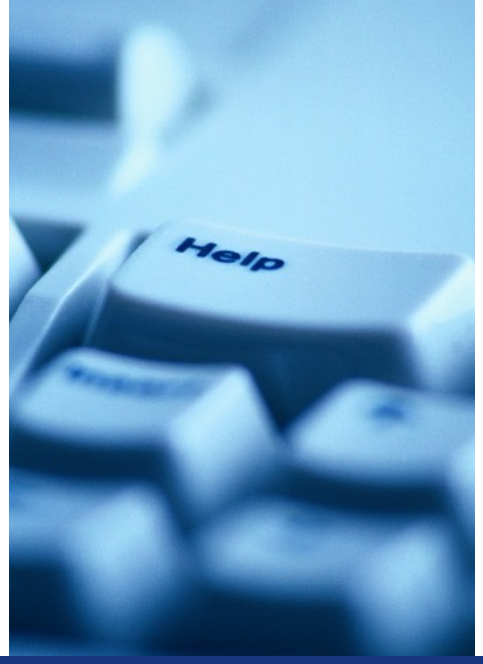
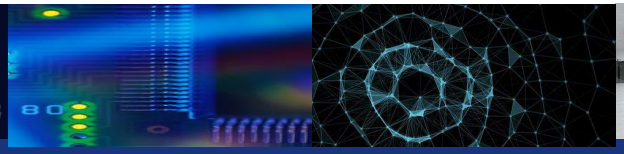



并行与分布式计算

Parallel & Distributed Computing

陈鹏飞
计算机学院
2021-09-03

第一讲：并行计算概览



1 个人简介

2 课程安排

3 课程考核

4 参考资料

5 并行计算简介

个人简介

陈鹏飞

副教授，博士生导师

任职于中山大学数据科学与计算机学院。2016年6月于西安交通大学计算机科学与技术系获博士学位。2012年7-2012年11月作为“明日之星”在微软亚洲研究院工作实习，2016年6月-2018年1月在IBM中国研究院云计算部分担任研究员，2017年2月-2017年4月作为访问研究员在IBM T.J. Watson研究中心工作。主要的研究方向为：大规模分布式系统、云计算、软件可靠性、智能运维分析等。近年来在国际会议和期刊共发表27篇论文，其中SCI论文6篇，中科院一区及CCF A类会议和期刊论文5篇，同时担任多个国际期刊和会议的审稿人。

http://inpluslab.com/member

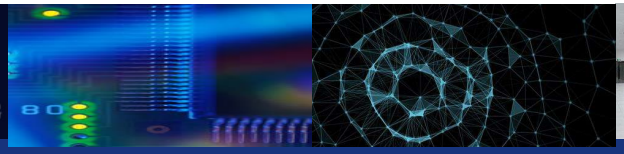
chen pengfei

13138623378

chenpf7@mail.sysu.edu.cn

超算五楼529d

游泳、羽毛球、徒步、爬山



课程安排

课程描述

本课程广泛讨论并行分布式系统相关话题包括：并行系统架构、并行编程模型、并行编程方法、并行系统性能评测、共性内存编程、MPI编程、CUDA编程、OpenCL编程、MapReduce编程、Ray编程、并行算法、大数据处理方法、云数据中心等，涵盖并行与分布式系统的基本原理与实践，同时会介绍学术界与工业界在这一领域的最新研究成果和最佳实践。课程以讲授为主、实际实践为辅，中间穿插学生对于相关领域文献的讨论。 为了强化学生对并行和分布式系统相关理论的理解，本课程设计了多个实践项目，供学生们选择。

课程目的

1、 了解并行与分布式系统的相关概念和原理；
2、 掌握并行与分布式系统的编程方法；
3、 了解并行与分布式系统领域的前沿技术；
4、 熟悉并行与分布式系统的架构设计；
5、 激发学生对并行与分布式系统的探索兴趣；

课程最终目标(可选做)

设计并实现一种并行编程语言，利用LLVM编译成中间代码，可转化成不同硬件指令，运行在Intel CPU、GPU、FPGA、或ARM上，并任选泽两种进行测试；

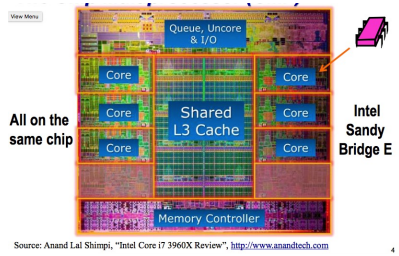
```
let add_edges context start_node exn_node exit_nodes method_body_nodes impl super_call =
  let pc_nb = Array.length method_body_nodes in
  let last_pc = pc_nb - 1 in
  let is_last pc = (pc = last_pc) in
  let rec get_body_nodes pc =
    let current_nodes = method_body_nodes.(pc) in
    match current_nodes with
    | JTrans.Skip when (is_last pc) && not (JContext.is_goto_jump context pc) ->
        exit_nodes
    | JTrans.Skip -> direct_successors pc
    | JTrans.Instr node -> [node]
    | JTrans.Prune (node_true, node_false) -> [node_true; node_false]
    | JTrans.Loop (join_node, _, _) -> [join_node]
  and direct_successors pc =
    if is_last pc && not (JContext.is_goto_jump context pc) then
      exit_nodes
    else
      match JContext.get_goto_jump context pc with
      | JContext.Next -> get_body_nodes (pc + 1)
      | JContext.Jump goto_pc ->
          if pc = goto_pc then [] (* loop in goto *)
```

Xlang

LLVM

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @factorial(i32 %val, i32 %total) #0 {
entry:
  %retval = alloca i32, align 4
  %val.addr = alloca i32, align 4
  %total.addr = alloca i32, align 4
  store i32 %val, i32* %val.addr, align 4
  store i32 %total, i32* %total.addr, align 4
  %0 = load i32, i32* %val.addr, align 4
  %cmp = icmp eq i32 %0, 1
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  %1 = load i32, i32* %total.addr, align 4
  store i32 %1, i32* %retval, align 4
  br label %return

if.end:                                           ; preds = %entry
  %2 = load i32, i32* %val.addr, align 4
  %sub = sub nsw i32 %2, 1
  %3 = load i32, i32* %val.addr, align 4
  %4 = load i32, i32* %total.addr, align 4
```

IR code



Source: Anand Lal Shimpi, "Intel Core i7 3960X Review", http://www.anandtech.com

CPU AVX

GPU PTX

https://github.com/duncantl/RLLVMCompile

https://www.llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html

课程最终目标(可选做)

设计并实现一种并行编程语言，利用LLVM编译成中间代码，可转化成不同硬件指令，运行在Intel CPU、GPU、FPGA、或ARM上，并任选泽两种进行测试；

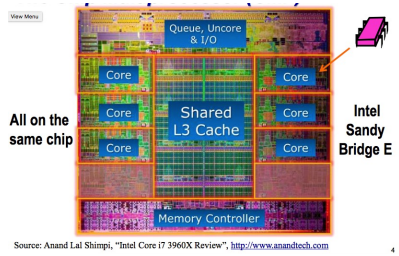
```
let add_edges context start_node exn_node exit_nodes method_body_nodes impl super_call =
  let pc_nb = Array.length method_body_nodes in
  let last_pc = pc_nb - 1 in
  let is_last pc = (pc = last_pc) in
  let rec get_body_nodes pc =
    let current_nodes = method_body_nodes.(pc) in
    match current_nodes with
    | JTrans.Skip when (is_last pc) && not (JContext.is_goto_jump context pc) ->
        exit_nodes
    | JTrans.Skip -> direct_successors pc
    | JTrans.Instr node -> [node]
    | JTrans.Prune (node_true, node_false) -> [node_true; node_false]
    | JTrans.Loop (join_node, _, _) -> [join_node]
  and direct_successors pc =
    if is_last pc && not (JContext.is_goto_jump context pc) then
      exit_nodes
    else
      match JContext.get_goto_jump context pc with
      | JContext.Next -> get_body_nodes (pc + 1)
      | JContext.Jump goto_pc ->
          if pc = goto_pc then [] (* loop in goto *)
```

Xlang

LLVM

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @factorial(i32 %val, i32 %total) #0 {
entry:
  %retval = alloca i32, align 4
  %val.addr = alloca i32, align 4
  %total.addr = alloca i32, align 4
  store i32 %val, i32* %val.addr, align 4
  store i32 %total, i32* %total.addr, align 4
  %0 = load i32, i32* %val.addr, align 4
  %cmp = icmp eq i32 %0, 1
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  %1 = load i32, i32* %total.addr, align 4
  store i32 %1, i32* %retval, align 4
  br label %return

if.end:                                           ; preds = %entry
  %2 = load i32, i32* %val.addr, align 4
  %sub = sub nsw i32 %2, 1
  %3 = load i32, i32* %val.addr, align 4
  %4 = load i32, i32* %total.addr, align 4
```

IR code



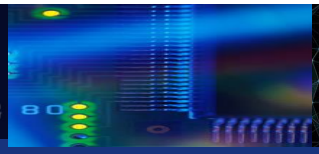
Source: Anand Lal Shimpi, "Intel Core i7 3960X Review", http://www.anandtech.com

CPU AVX

GPU PTX

https://github.com/duncantl/RLLVMCompile

https://www.llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html



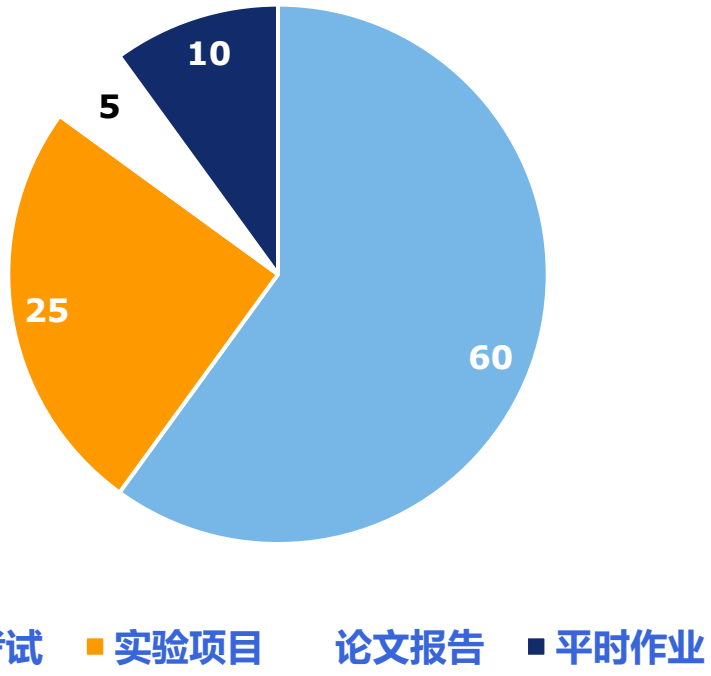
课程安排

课程内容

周次	课程内容	周次	课程内容
第1周	并行系统简介	第11周	GPU和CUDA并行编程
第2周	并行计算架构	第12周	OpenCL并行编程 OpenCL优化
第3周	并行编程模型	第13周	MapReduce编程 并行程序调试与测试
第4周	并行编程方法论	第14周	离散搜索和负载均衡 并行图算法
第5周	并行架构通信模式	第15周	论文演讲 Spark 编程
第6周	并行编程性能和扩展分析 性能分析工具	第16周	大数据分析技术 分布式系统简介
第7周	共享内存、一致性和同步	第17周	分布式模型与使能技术 分布式系统容错和复制
第8周	OpenMP编程	第18周	云平台与数据中心
第9周	MPI编程	第19周	分布式机器学习 课程答疑
第10周	停课	第20周	项目作业提交及考试

课程考核

本门课程是计算机科学和软件工程领域的重点课程，包括理论知识的学习和工程实践。为了巩固知识、锻炼学生的实操能力，本门课程考核主要包括四部分：**期末考试、实验项目（4次）、论文报告和平时作业（每两周一次）**。 加分项：课堂积极回答问题、按时上课等。

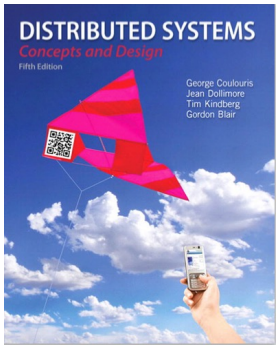


成绩权重分布

参考资料

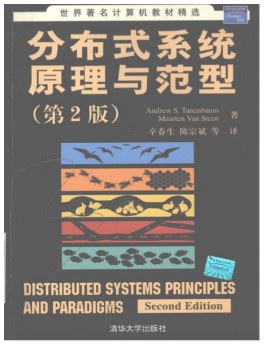
无固定教材！！！

- 参考教科书

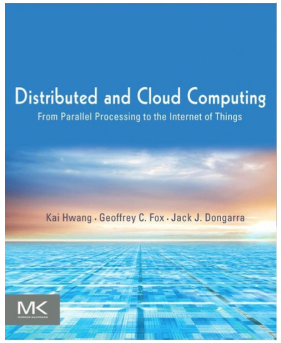


Distributed systems - Concepts and Design (5th)

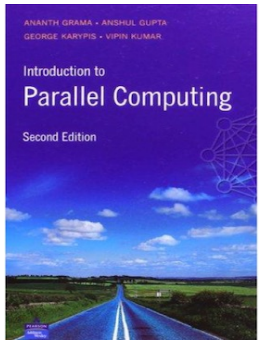
Parallel Computer Architecture



Distributed Systems Principles and Paradigms



Distributed and cloud computing



Introduction to parallel computing

- 参考网络资源

1. Swarthmore course, https://www.cs.swarthmore.edu/~newhall/cs87/s12/;
2. Brown course, http://cs.brown.edu/courses/csci1780/;
3. Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/;
4. CMU course, http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/www/;
5. Top 500, https://www.top500.org/;
6. MPI, http://www.mcs.anl.gov/research/projects/mpi/;
7. Some tips for presentation, https://www.cs.swarthmore.edu/~newhall/presentation.html;
8. 华中科技大学-金海：http://grid.hust.edu.cn/courses/parallel/;

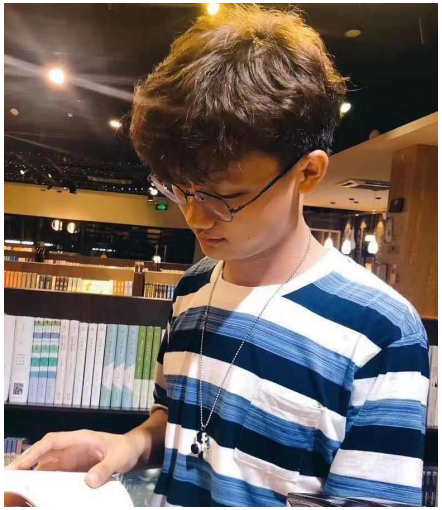
课程网站和助教

并行与分布式系统助教

该二维码7天内(9月8日前)有效，重新进入将更新

并行与分布式计算课程微信群

杨婉琪

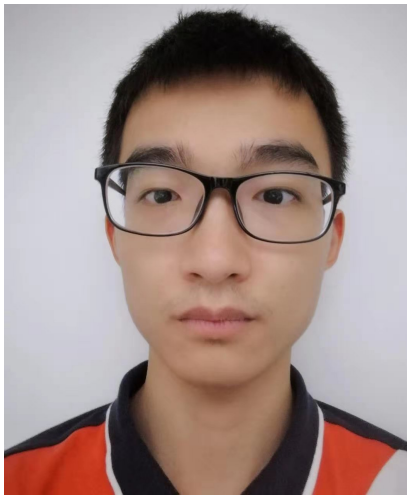


胡子俊

杨婉琪：yangwq23@mail2.sysu.edu.cn

胡子俊：809004963@qq.com

张景润：zhangjr35@mail2.sysu.edu.cn

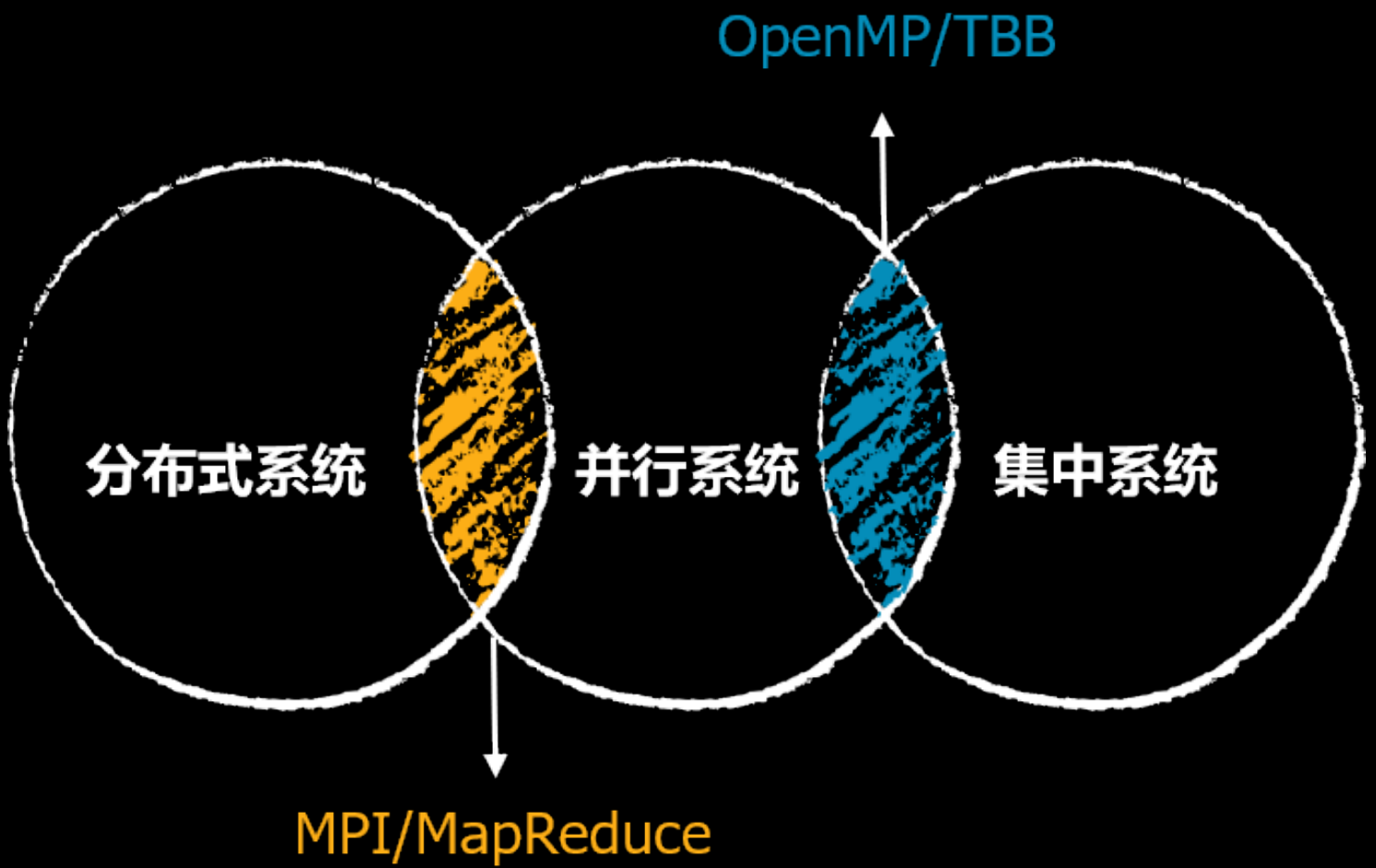


张景润

课程网站：http://course.dds-sysu.tech/

并行计算简介

并行与分布式系统



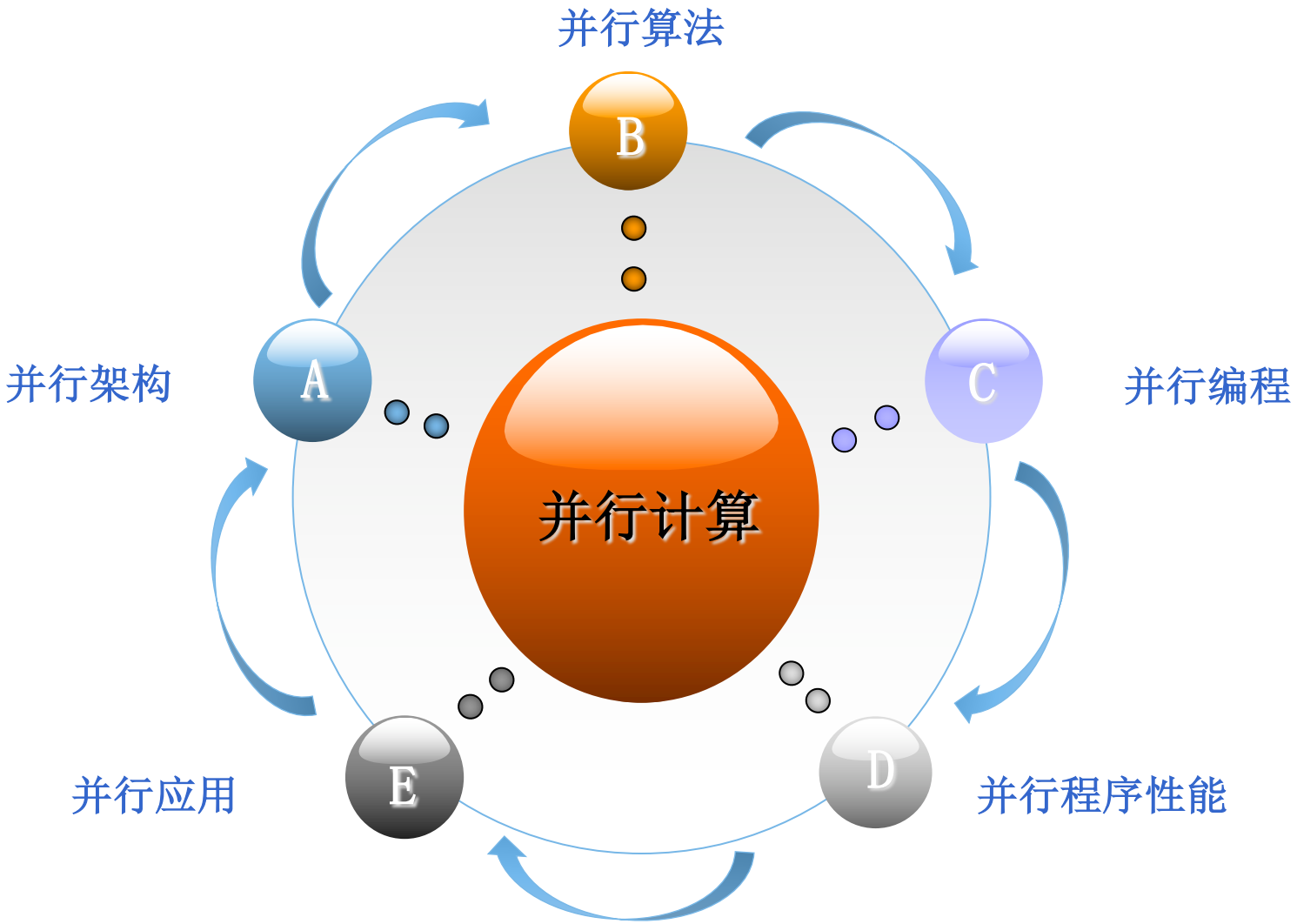
并行计算场景


Uncompromised viewing
Borderless, notch-free
HUAWEI Mate X
7.2"
Galaxy Fold

并行计算设备

手机　个人电脑　服务器　数据中心　超级计算机

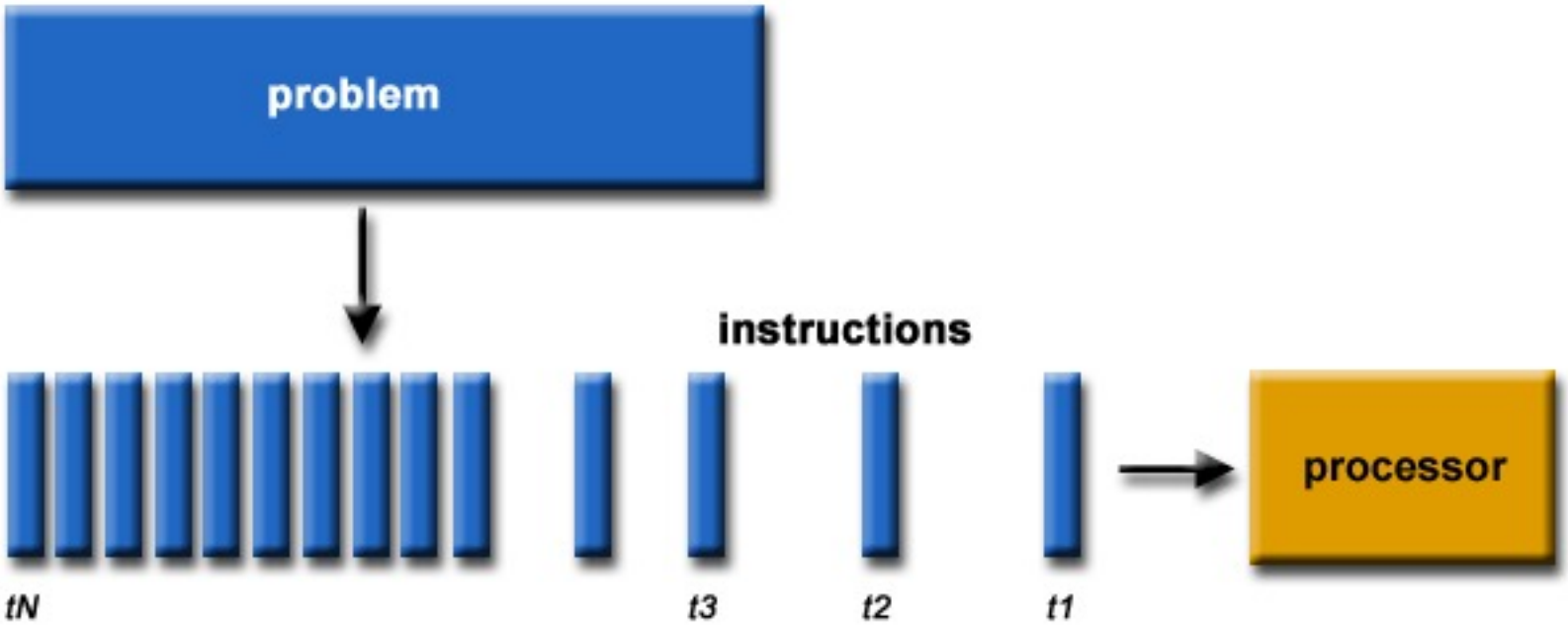
并行计算总览



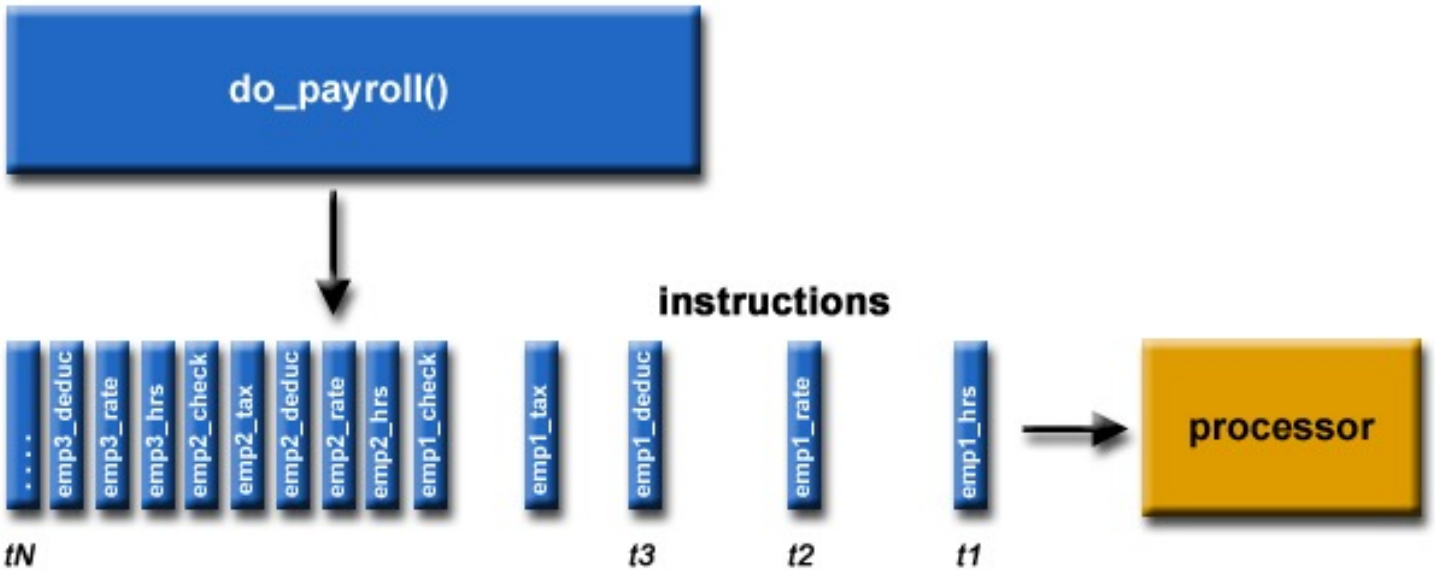
什么是并行计算？

✧ 早期传统软件为串行计算而设计

- 在仅有一个处理器的单机上运行；
- 一个问题分解成离散的串行指令序列求解；
- 指令是一条接一条执行；
- 在任意时间任意时刻，只有一条指令在执行。



什么是并行计算？

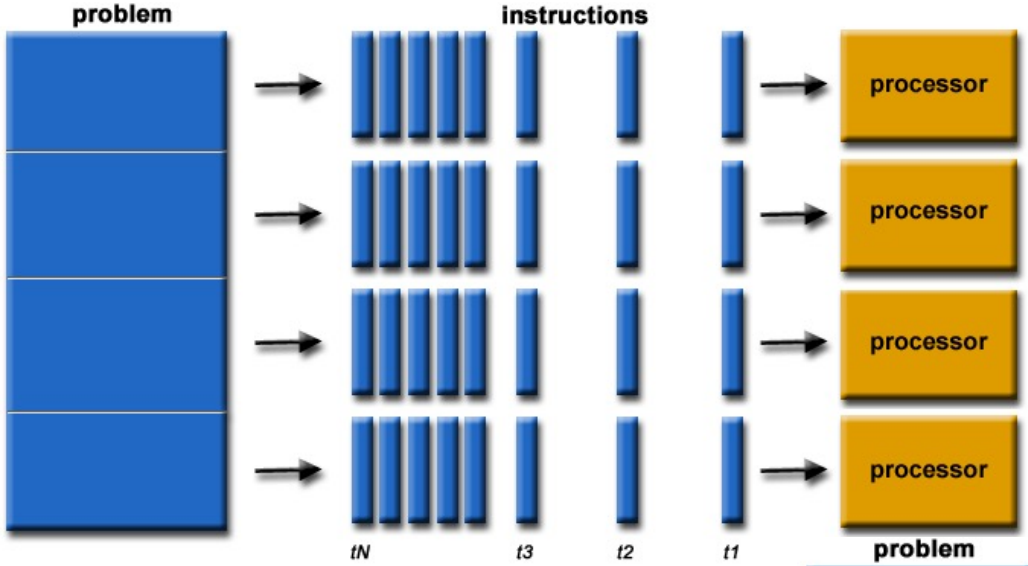


串行计算样例（接上页）

✧ 并行计算可以简单定义为同时利用多个计算资源解决一个计算问题

- 程序运行在多个CPU上；
- 一个问题被分解成离散可并发解决的小部分；
- 每一小部分被进一步分解成一组指令序列；
- 每一部分的指令在不同的CPU上同时执行；
- 需要一个全局的控制和协调机制；

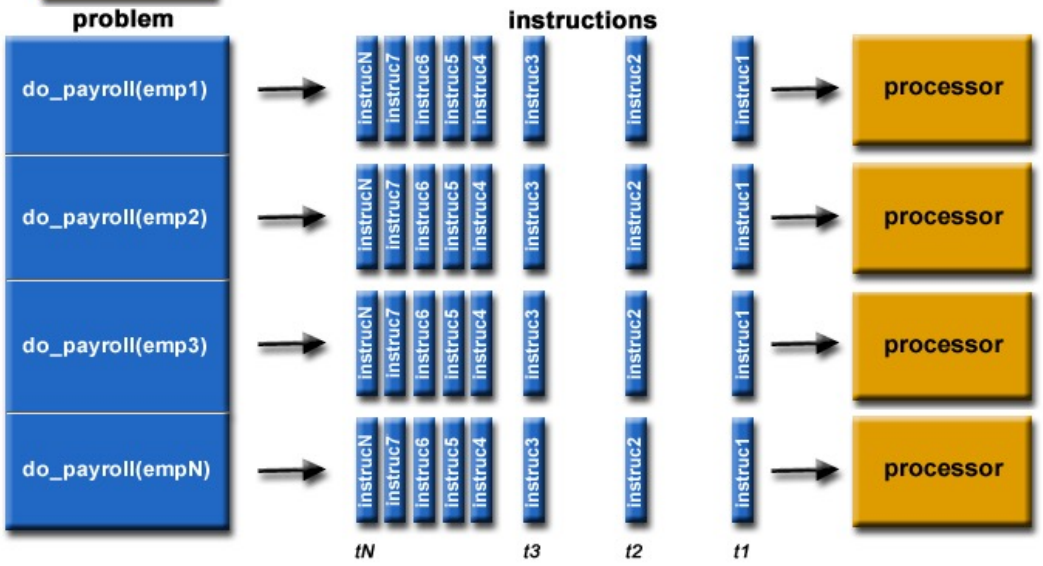
什么是并行计算？



并行计算原理

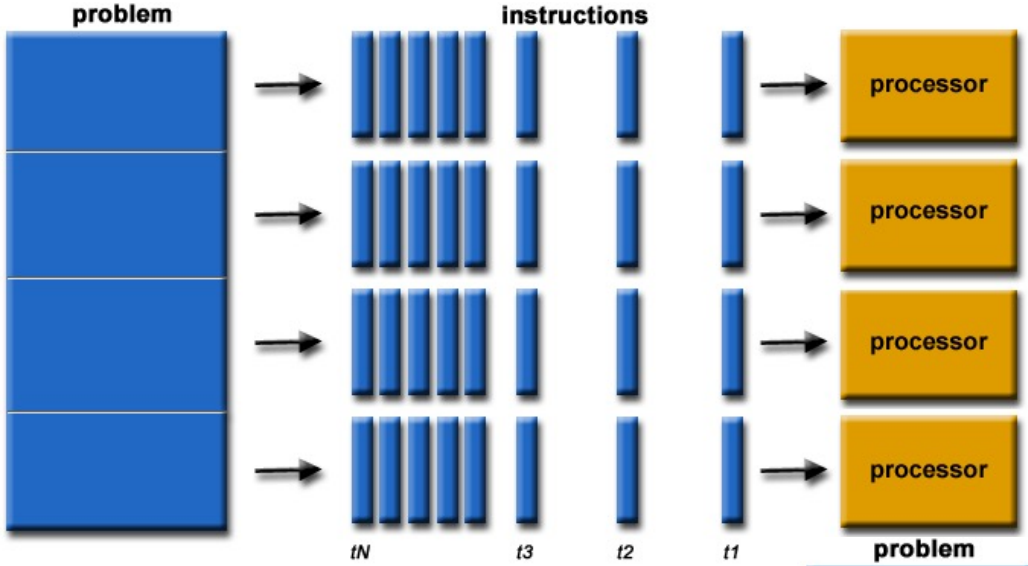
可并行的计算问题应该满足：
1、可分解成同时计算的几个离散片段；
2、在任意时刻可同时执行多条指令；
3、多计算资源所花的时间少于单计算资源；

计算资源一般为：
1、具有多处理器 / 多核的单台主机；
2、通过网络连接的若干数量的主机；



并行计算样例

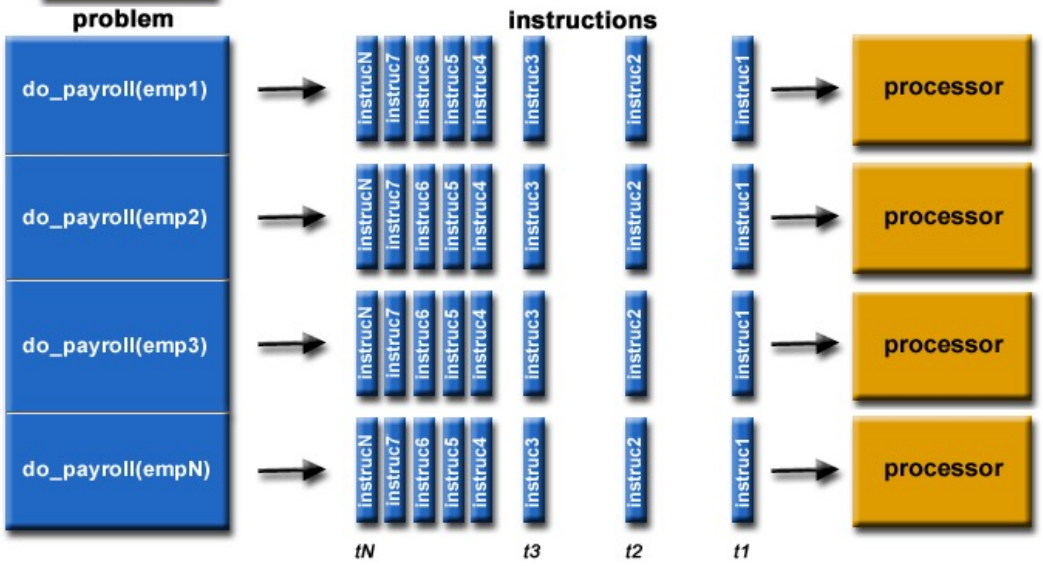
什么是并行计算？



并行计算原理

可并行的计算问题应该满足：
1、可分解成同时计算的几个离散片段；
2、在任意时刻可同时执行多条指令；
3、多计算资源所花的时间少于单计算资源；

计算资源一般为：
1、具有多处理器 / 多核的单台主机；
2、通过网络连接的若干数量的主机；



并行计算样例

并行计算的优势

1).在自然界，很多复杂的、交叉发生的事件是同时发生的，但是又在同一个时间序列中；

2).与串行计算相比，并行计算更擅长建模、模拟、理解真实复杂的现象；

Galaxy Formation

Planetary Movments

Climate Change

Rush Hour Traffic

Plate Tectonics

Weather

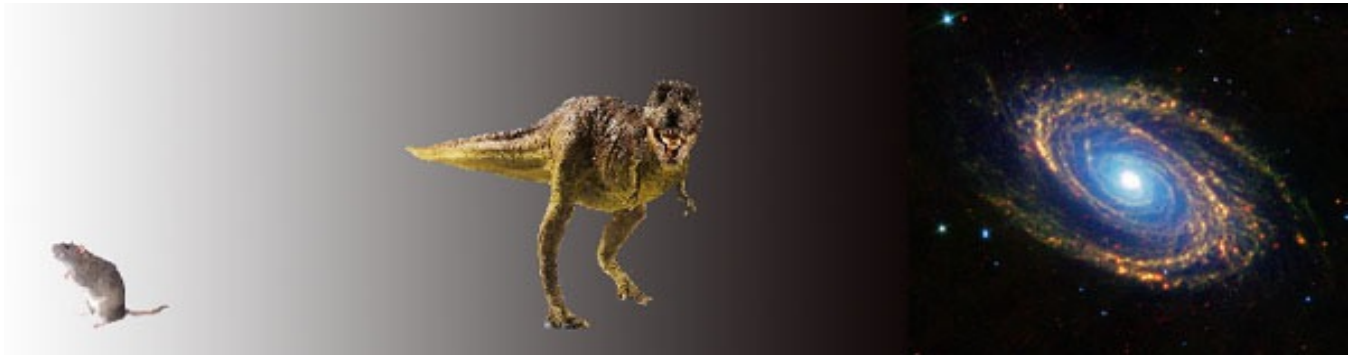
并行计算的优势

- **节省时间和花费**

  1).理论上，给一个任务投入更多的资源将缩短任务的完成时间，减少潜在的代价；
  2).并行计算机可以由多个便宜、通用计算资源构成；

- **解决更大／更复杂问题**

  很多问题很复杂，不实际也不可能在单台计算机上解决，例如：Grand Challenges

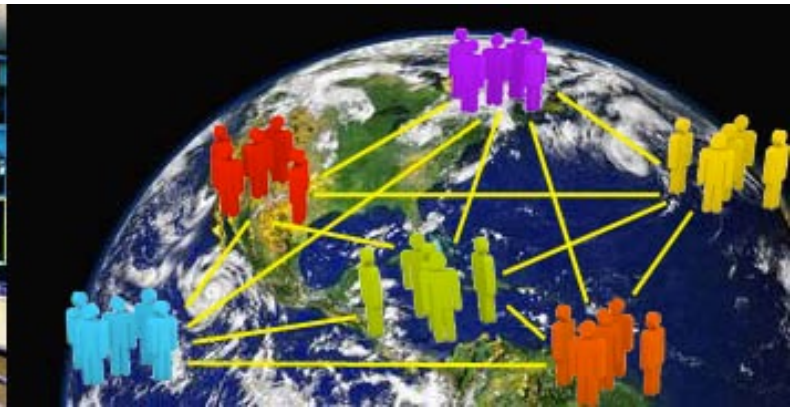


https://grand-challenge.org/

并行计算的优势

- **实现并发处理**

  1).单台计算机只能做一件事情，而多台计算机却可以同时做几件事情；
  2).例如协作网络，来自世界各地的人可以同时工作；



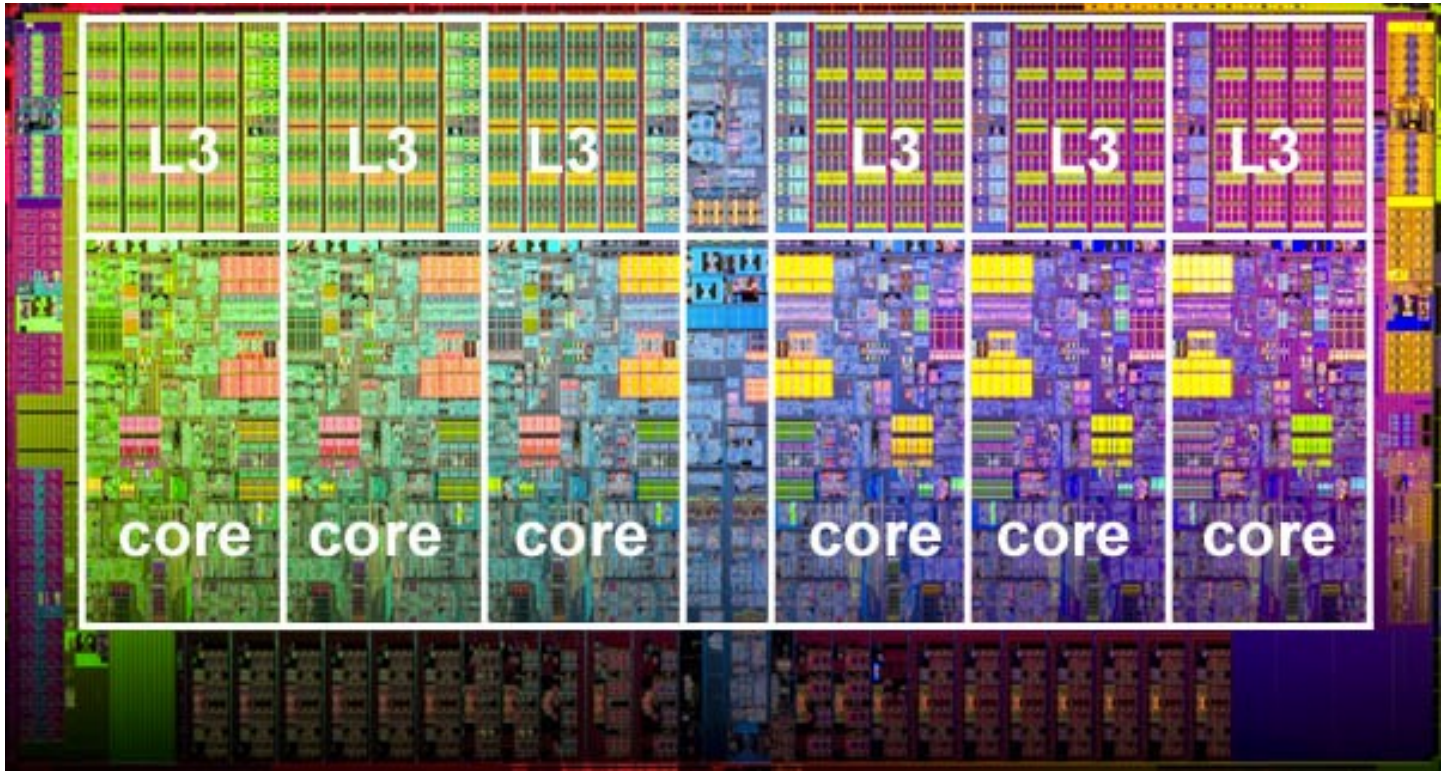
- **利用非本地资源**

  当本地计算资源稀缺或者不充足时，可以利用甚至是来自互联网的计算资源。
  1). SETI@home (setiathome.berkeley.edu);
  2). Folding@home (folding.stanford.edu)

并行计算的优势

- 更好地发挥底层并行硬件

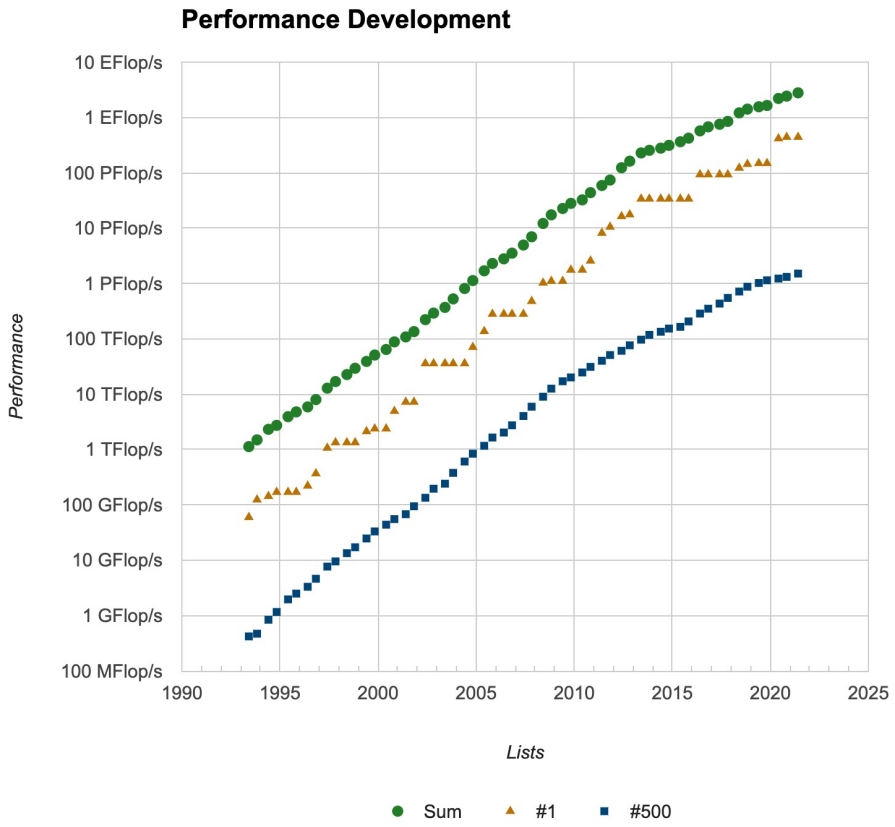
1).现代计算机甚至笔记本都具有多个处理器或者核心；
2).并行软件就是为了针对并行硬件架构出现的；
3).串行程序运行在现代计算机上会浪费计算资源；



具有6个核心和6个L3 cache的Intel Xeon处理器

并行计算的未来

- 过去20年，高速网络、分布式系统、多处理器计算机架构的发展趋势说明并行化是计算的未来；
- 在这20年间，超级计算机的性能提高了500，000，而且没有停止增长的迹象；
- 竞争延续到百亿亿（Exascale）次计算；

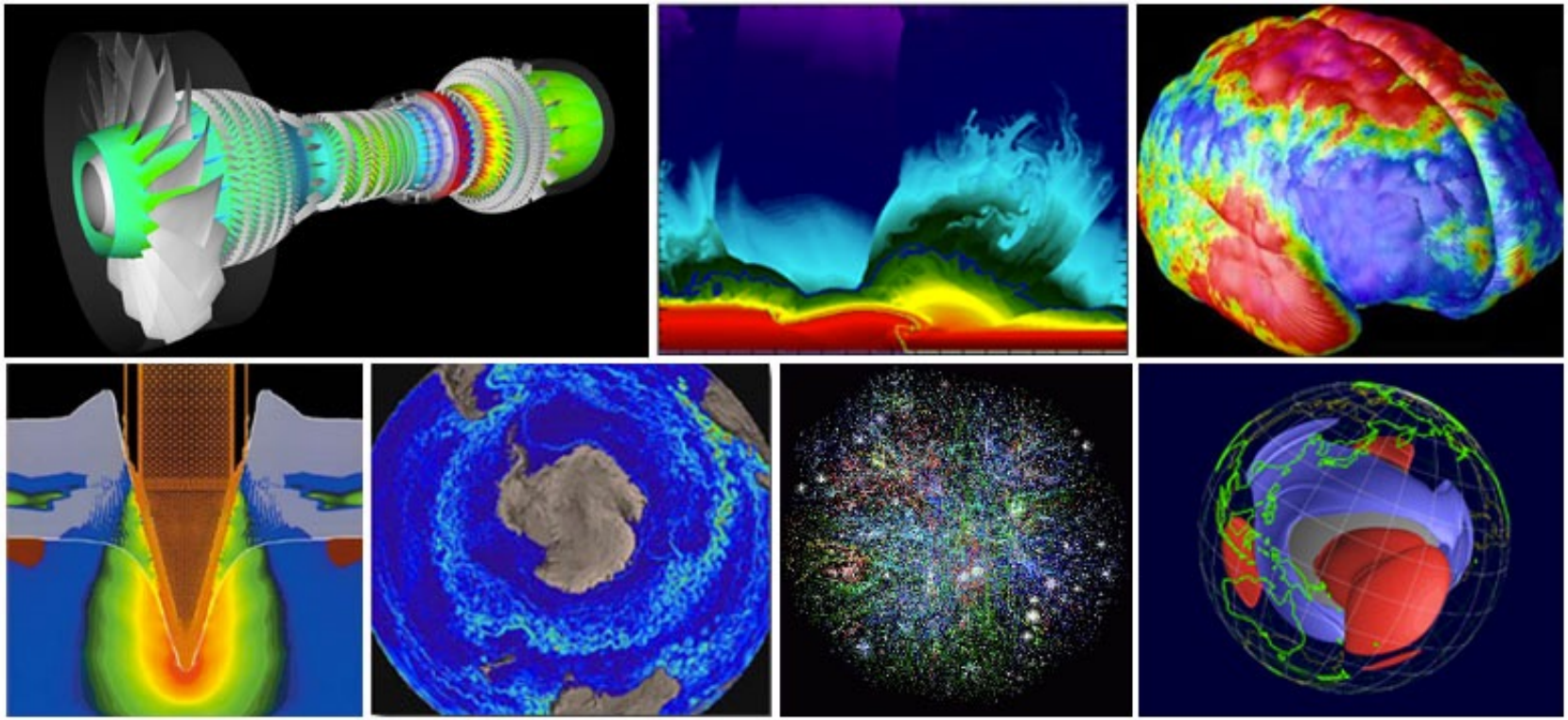


并行计算的用途

➢ 科学和工程计算

一直以来，并行计算被认为是“高端计算”，已经被用于科学和工程的很多领域：

- 大气、地球、环境；
- 物理学如核能、粒子模拟、高压、高分子等；
- 生物科技、遗传学；

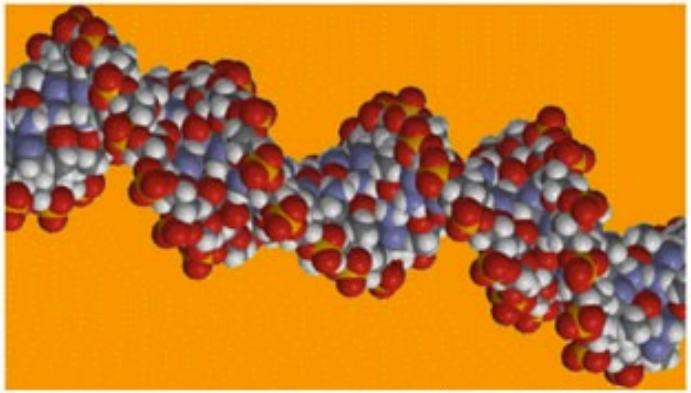
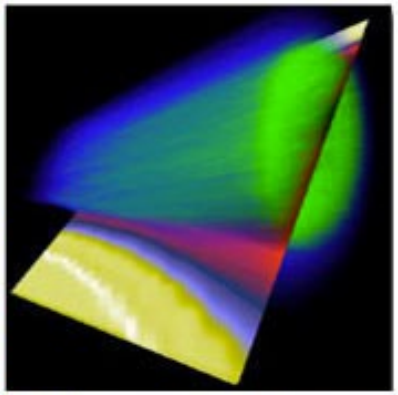
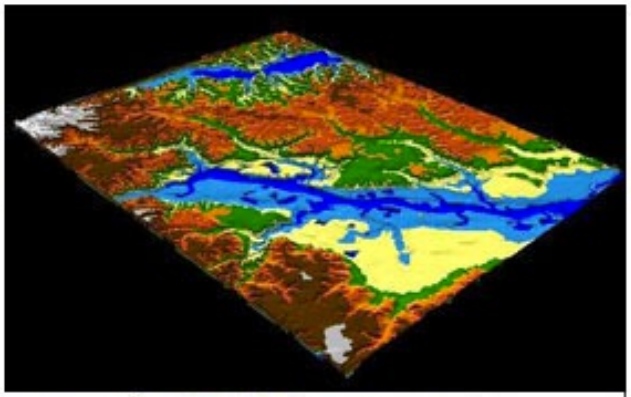
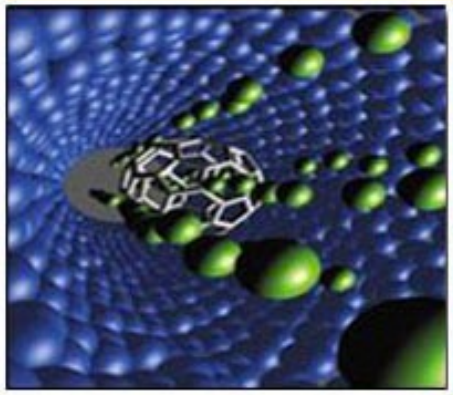


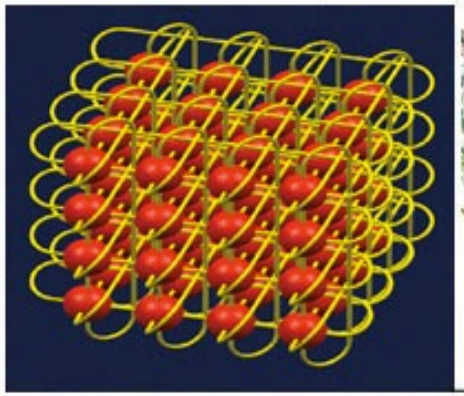
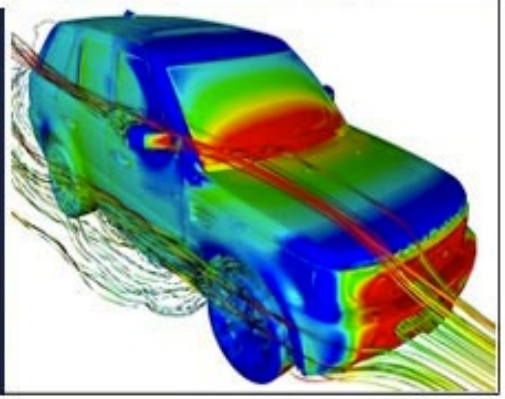
并行计算的用途

➢ 工业和商业应用

当前，商业应用对于高速计算机的开发起到了巨大的推动作用：

- “大数据”、数据库、数据挖掘；
- 石油勘探；
- Web搜索引擎，医学图像处理等等；

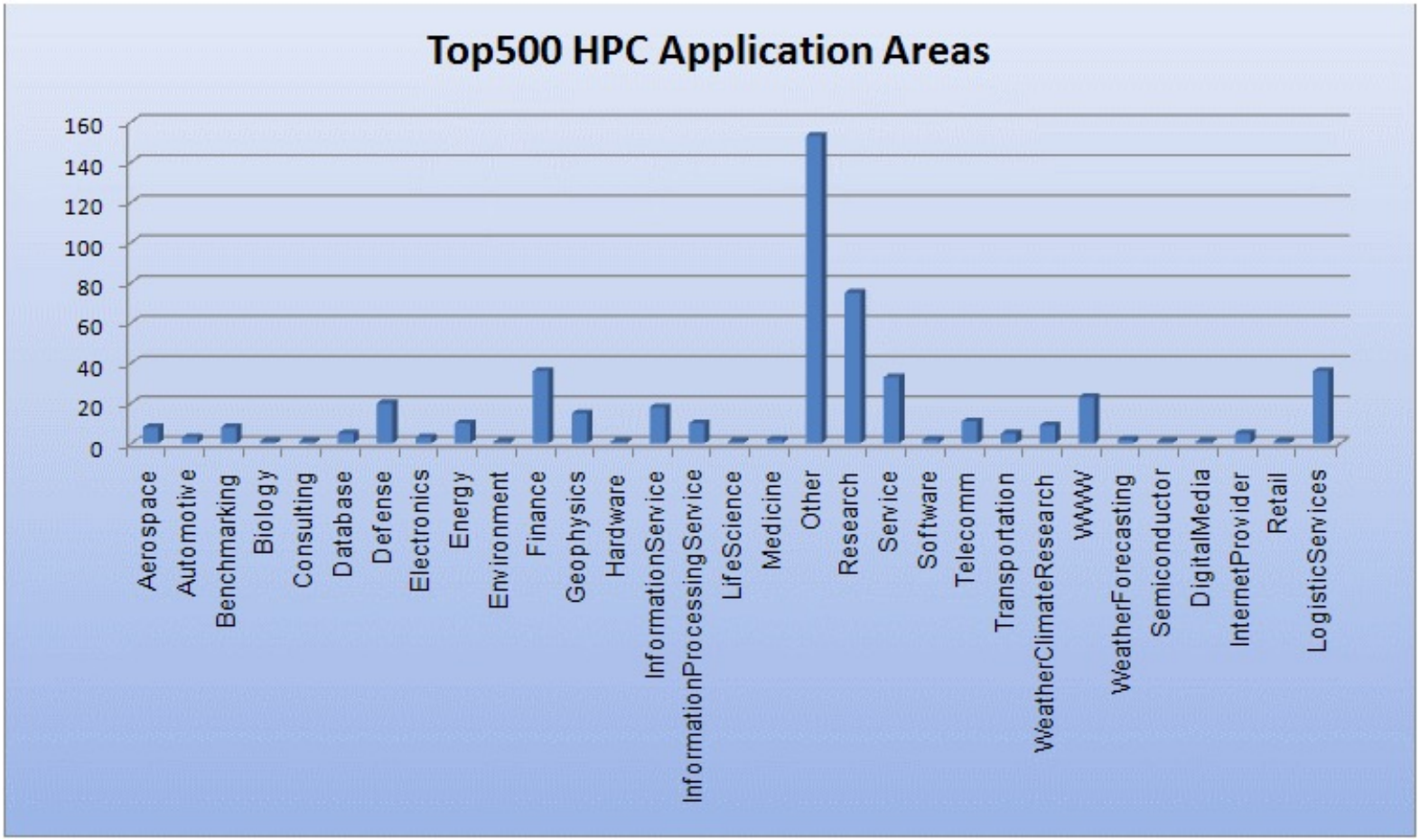





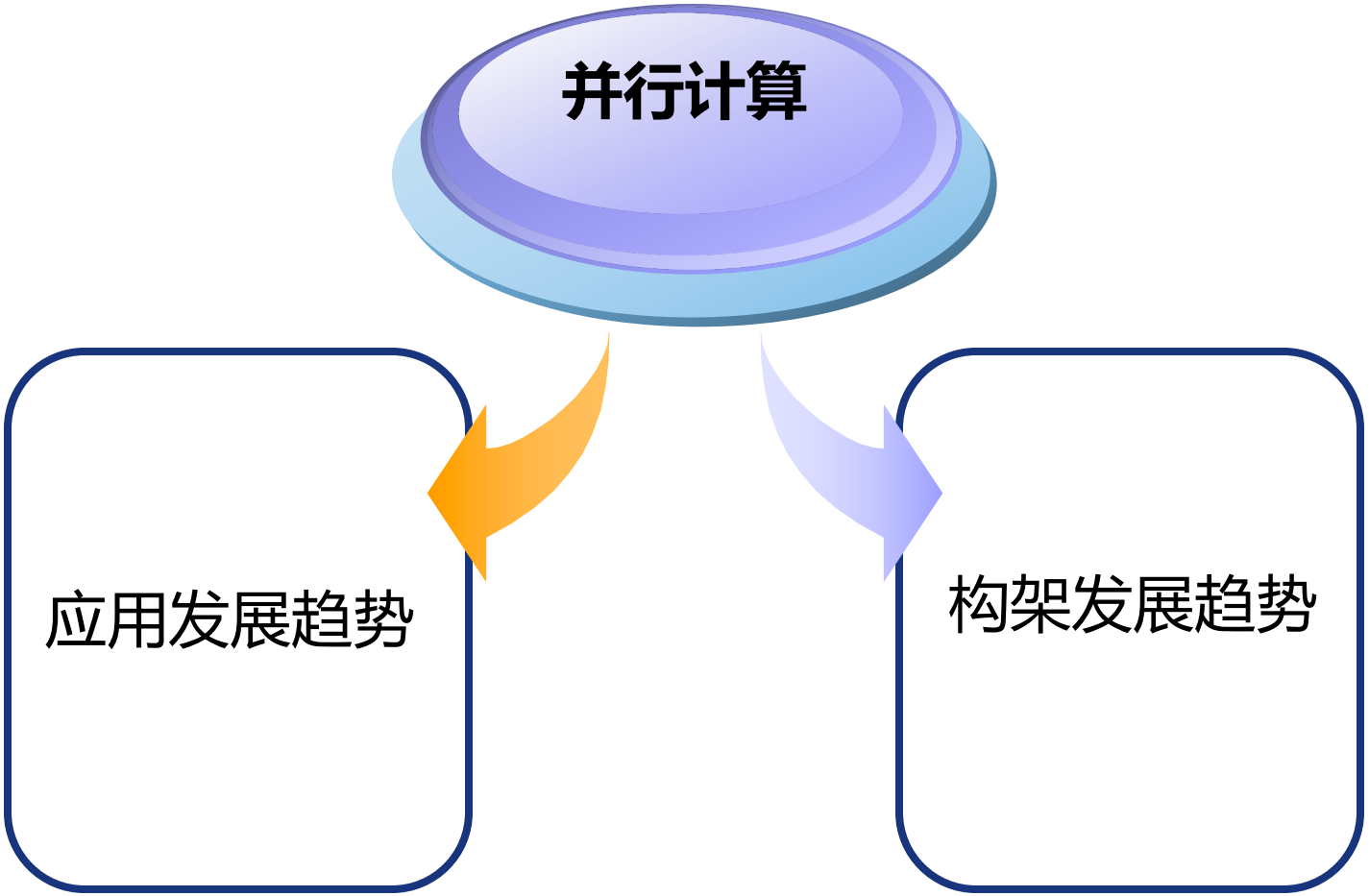
并行计算的用途

- 全球范围的应用

并行计算在全球范围内在各个领域得到了广泛的应用。



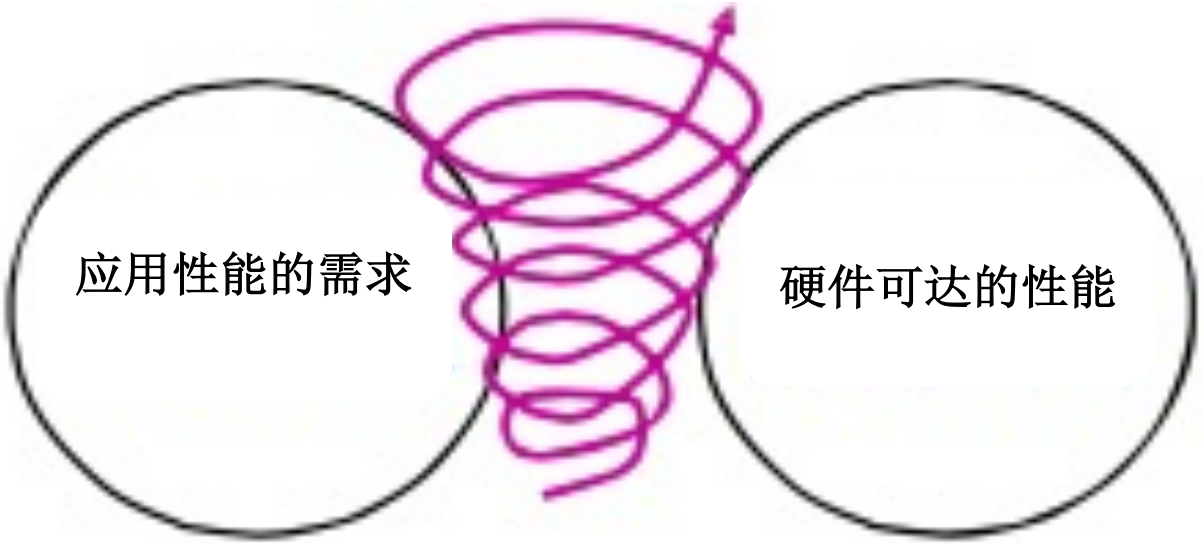
并行计算发展的驱动力



并行计算发展的驱动力

- 应用发展趋势

在硬件可达到的性能与应用对性能的需求之间存在正反馈（Positive Feedback Cycle）



例如应用领域：

1).科学计算：生物、化学、物理，……

2).通用计算：视频、图像、CAD、数据库，……

思考：现在的应用对于性能的需求？

并行计算发展的驱动力

- 应用发展趋势

大量设备、用户、内容涌现

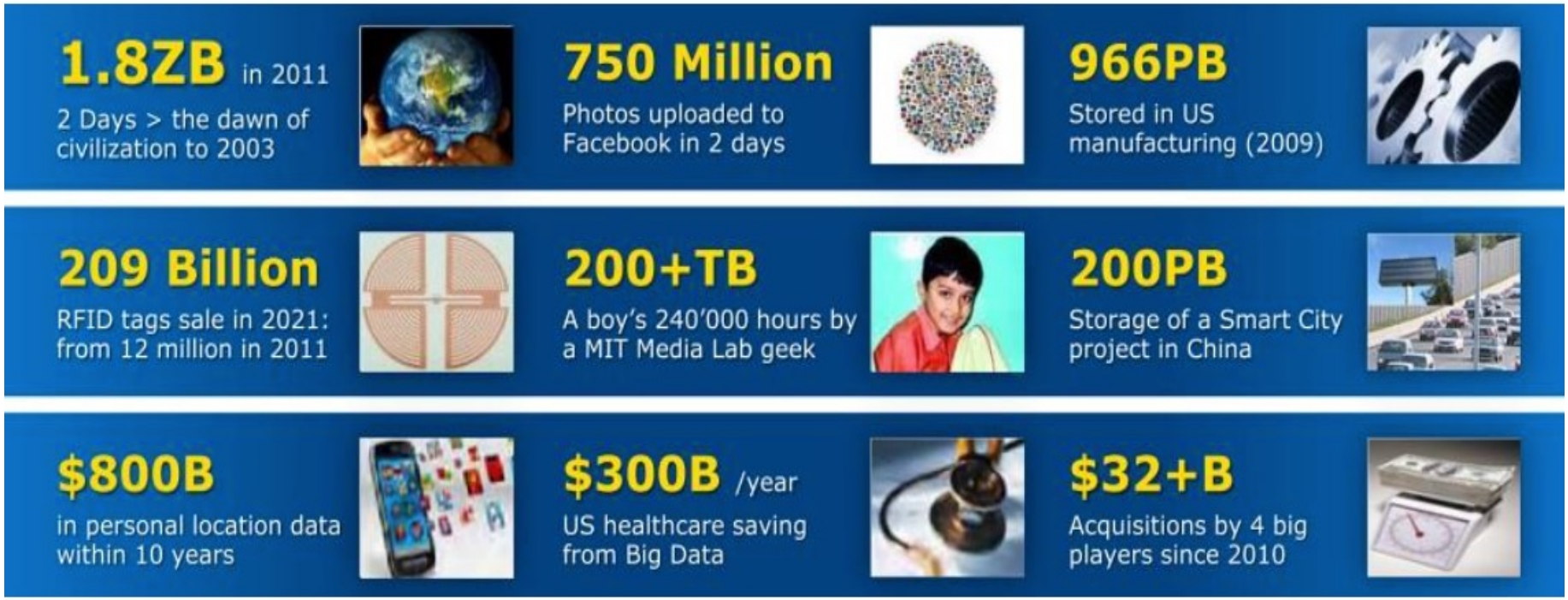
Source: IDF2012

并行计算发展的驱动力

- 应用发展趋势

大数据场景：

大数据正逐渐成为新的生产资料，大数据承载的“大”信息将成为21世界的动力来源



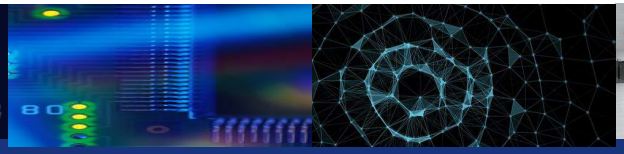
并行计算发展的驱动力

- 应用发展趋势

并行计算发展的驱动力

- 应用发展趋势

云计算的兴起：
廉价的硬件，应用弹性的扩展；
应用种类繁多，负载异构性增加；



并行计算发展的驱动力

- 架构发展趋势

迄今为止，CPU架构技术经历了四代即：电子管（Tube）、晶体管（Transistor）、集成电路（IC）和大规模集成电路（VLSI），这里只关注VLSI。

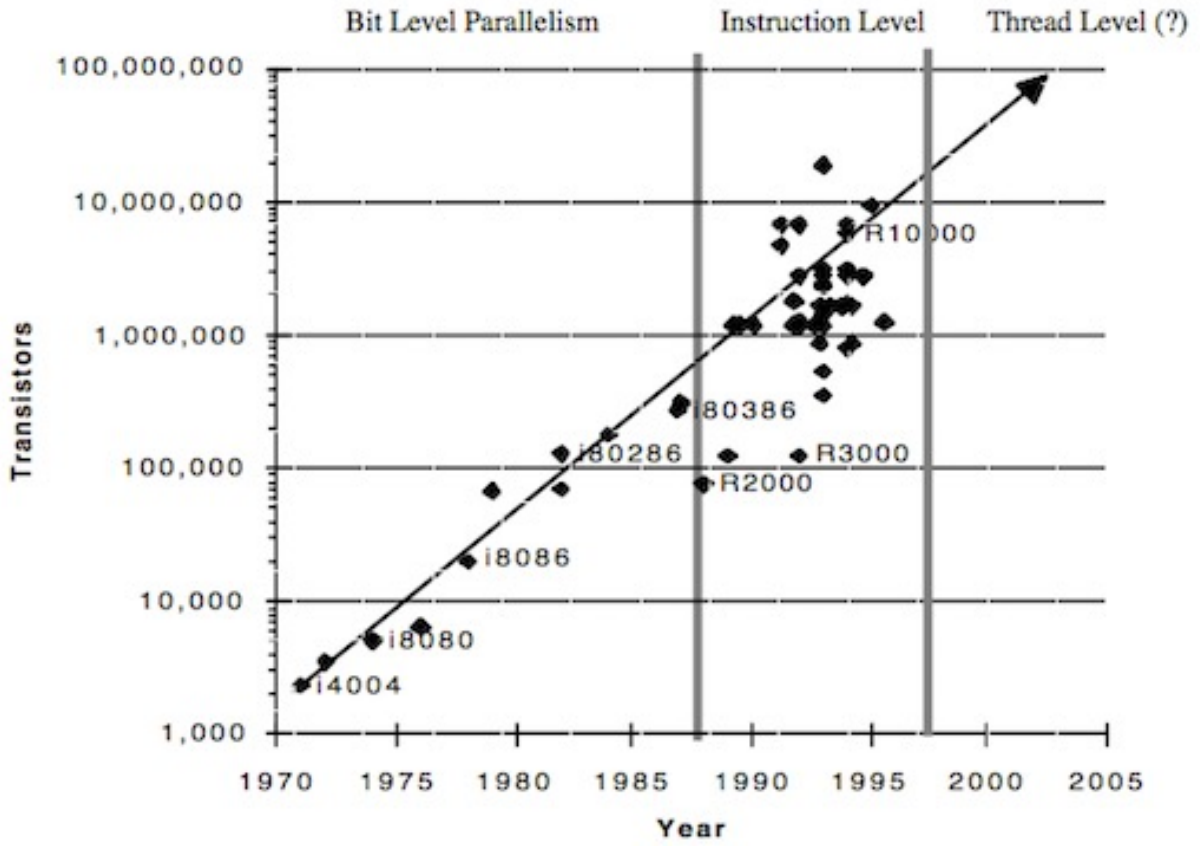
VLSI最的特色是在于对并行化的利用，不同的VLSI时代具有不同的并行粒度：

1). 1985年之前：bit水平的并行，从4bit->8 bit->16bit （数据通路）
   a. 32bit后并行化速度放慢?
   b. 最近几年64bit才被广泛使用，128bit还很遥远，（非性能限制）
   c. 32bit并行技术的应用导致了计算机性能的显著提高
2). 80年代到90年代：指令水平的并行
   a. 流水线、简单指令集、先进的编译技术（RISC）
   b. 片上缓存（caches）和功能部件 =>超变量执行
   c. 更复杂控制机制：乱序执行、推测、预测，用于控制转换和延迟问题；
3). 线程水平的并行
   现代并行计算的主要方式：线程并行。

并行计算发展的驱动力

- 架构发展趋势

VLSI不同代际划分



三个阶段：

Bit级并行；

指令级并行；

线程级并行；

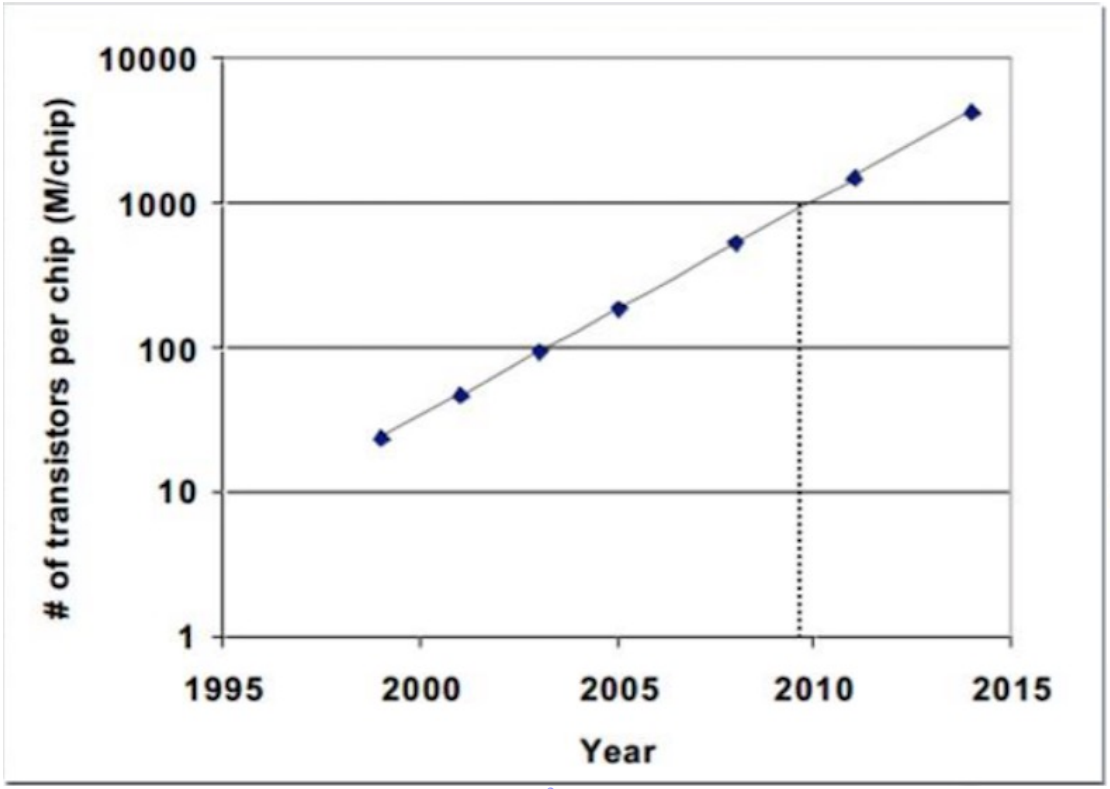
并行计算发展的驱动力

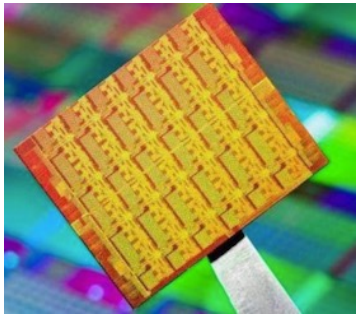
- 架构发展趋势

VLSI技术发展趋势：

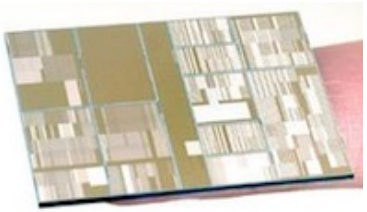
1).Intel声明在Itanium（安腾）处理器 晶体管的集成数量达到1.7B；

2).Gigscale集成（GSI）＝每个芯片集成晶体管数量达到1B；





Intel 14nm



IBM 7nm

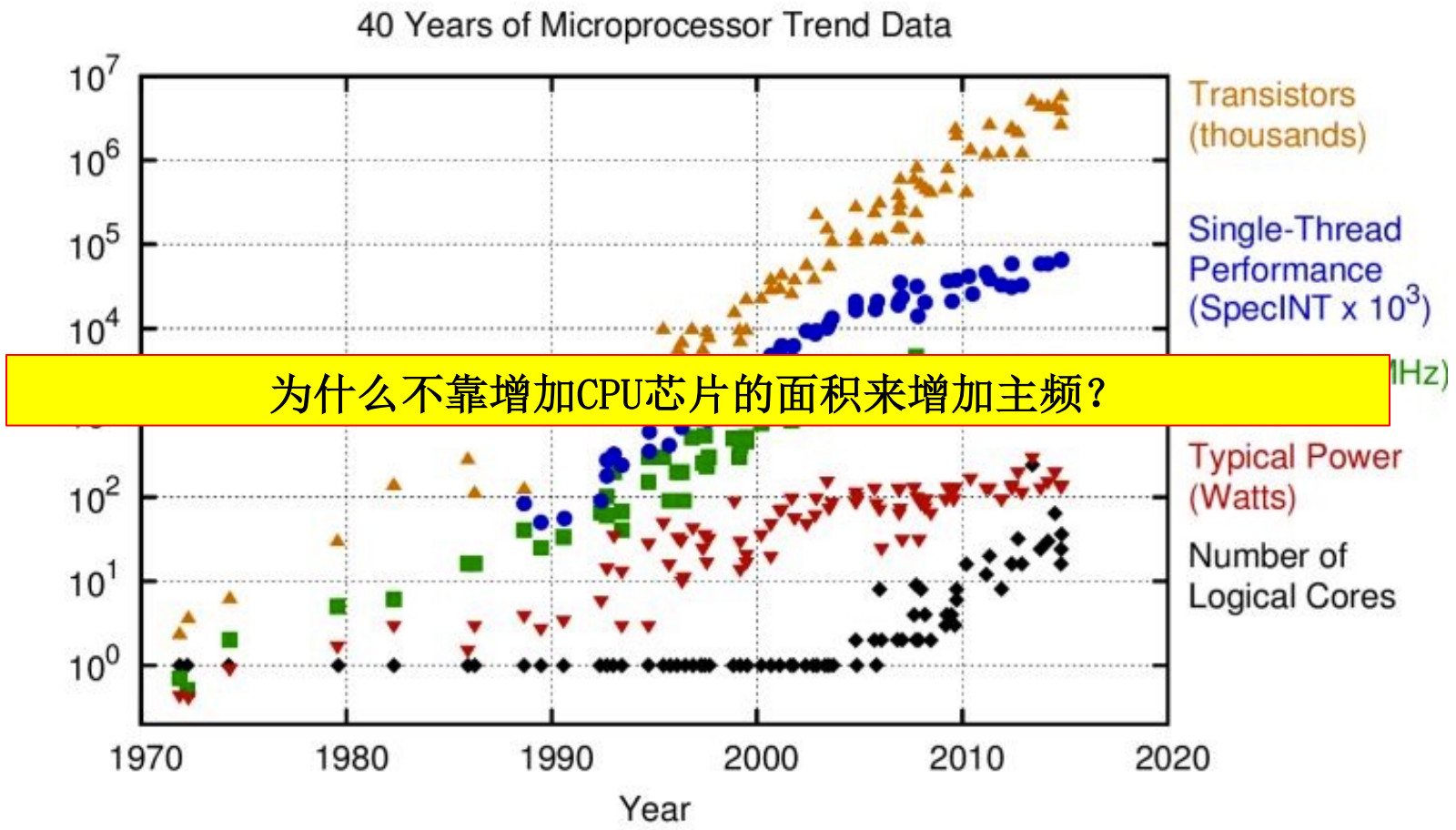
来源：http://users.ece.gatech.edu/~jeff/ece4420/technology.pdf

Semiconductor manufacturing processes

10 μm – 1971
6 μm – 1974
3 μm – 1977
1.5 μm – 1982
1 μm – 1985
800 nm – 1989
600 nm – 1994
350 nm – 1995
250 nm – 1997
180 nm – 1999
130 nm – 2001
90 nm – 2004
65 nm – 2006
45 nm – 2008
32 nm – 2010
22 nm – 2012
14 nm – 2014
10 nm – 2017
7 nm – ~2019
5 nm – ~2021

并行计算发展的驱动力

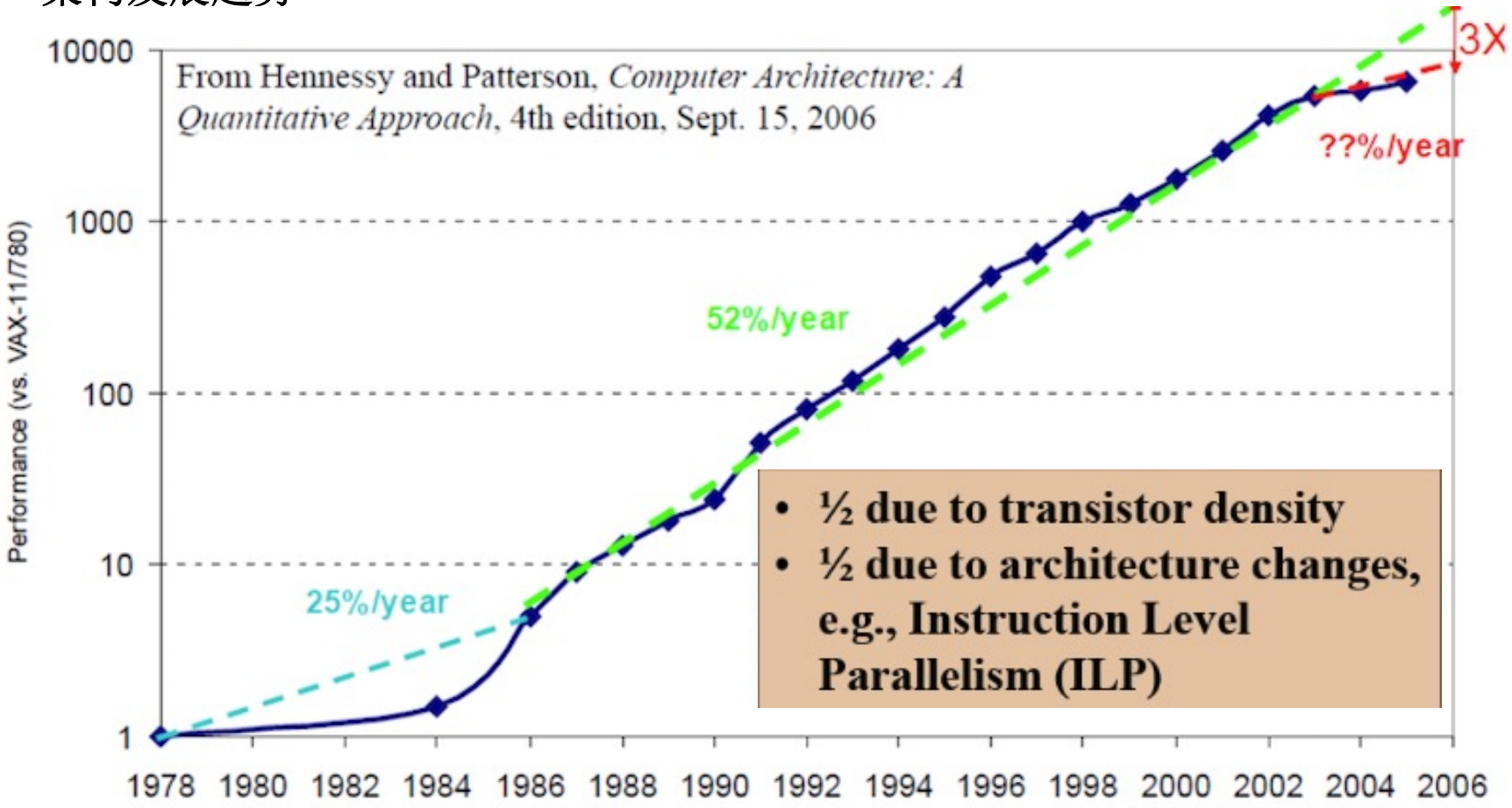
- 架构发展趋势



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

并行计算发展的驱动力

- 架构发展趋势



Vax：25%/年 从1978到1986；
RISC+X86：52%/年 从1986到2002；RISC+X86： 5%/年 从2002年到现在

并行计算发展的驱动力

- 架构发展趋势

1). Moore定律：

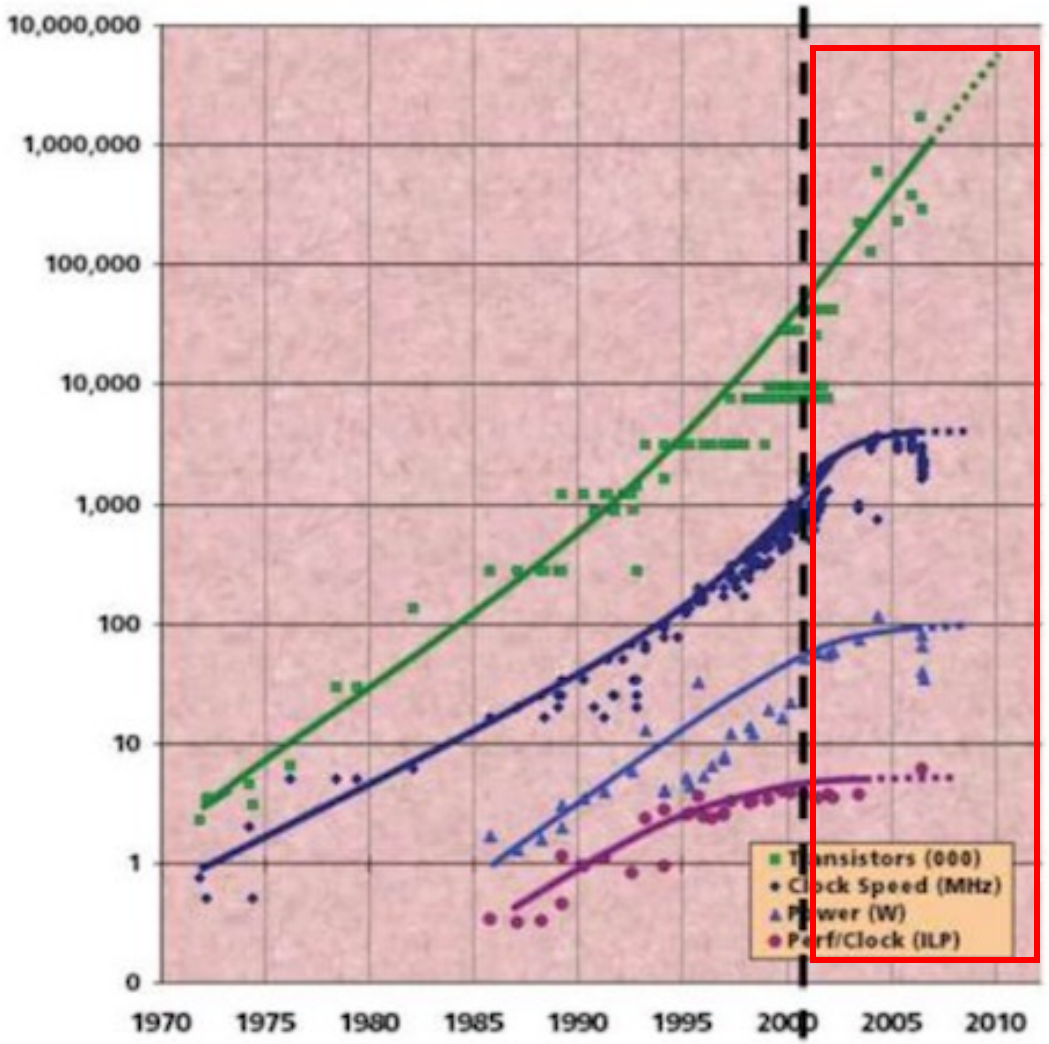
“芯片上的集成晶体管数量每18个月增加一倍”；

2). 2000年以前：

a. 单处理器的性能处于一直增加的状态；

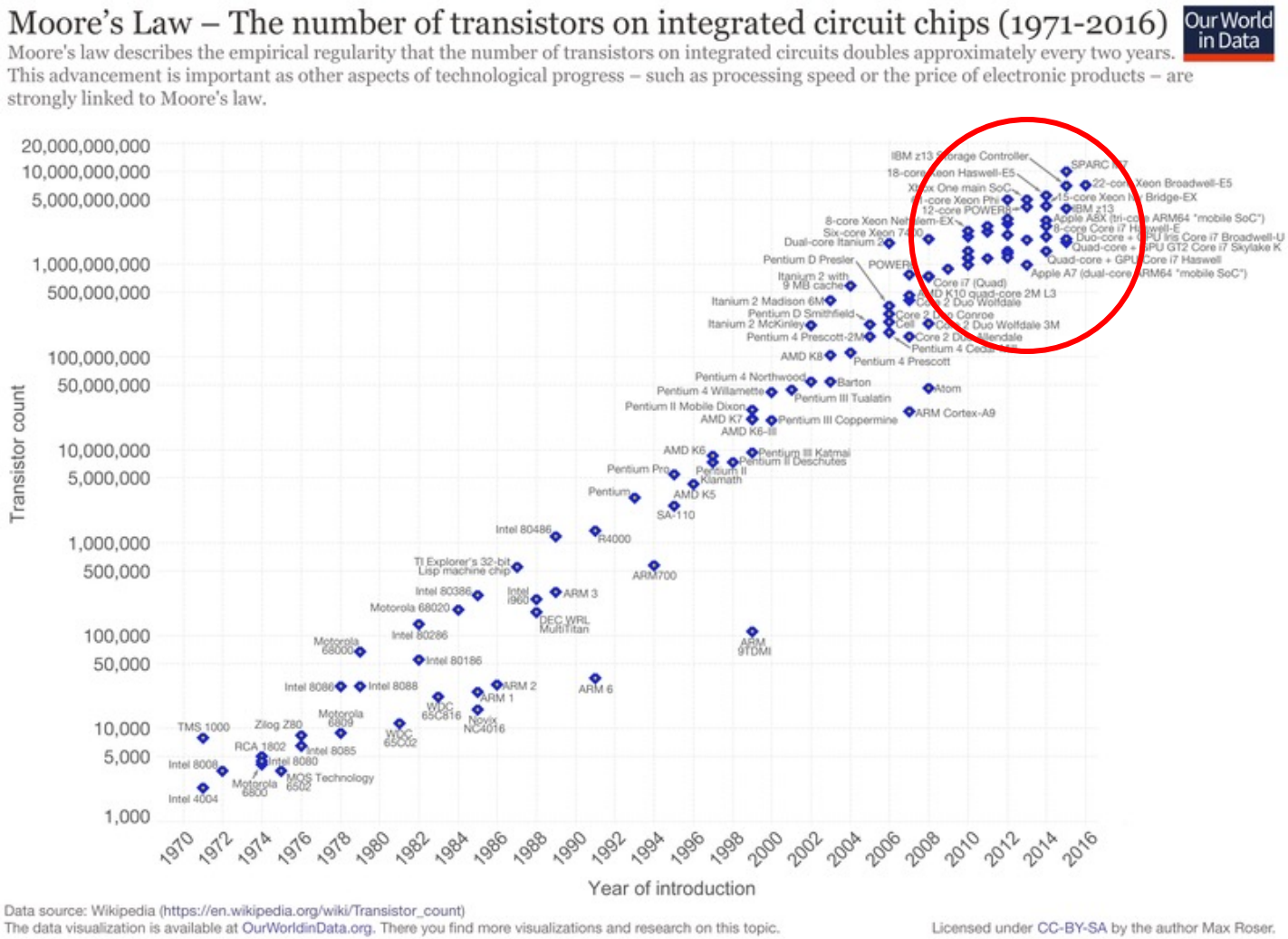
b. 同样的串行程序在新的硬件上的执行速度变化；

c. 计算机市场上提到最多的就是“MHZ/GHZ”



并行计算发展的驱动力

- 架构发展趋势

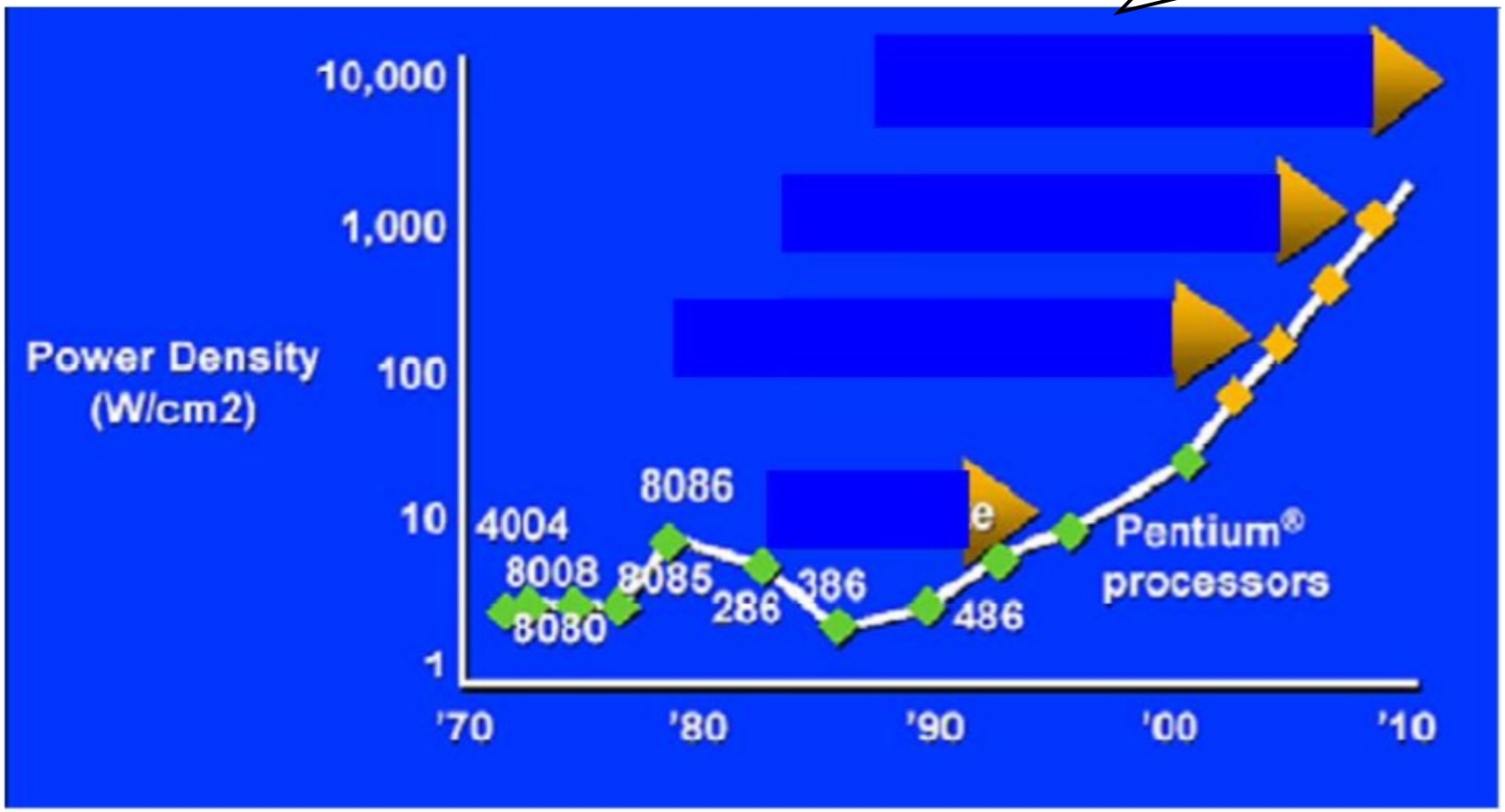


并行计算发展的驱动力

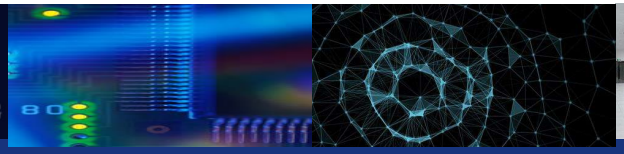
- 架构发展趋势

为什么CPU主频增长越来越缓慢，Moore定律会失效？

发展趋势不再是高速的CPU主频，而是“多核”



Pat Gelsinger, ISSCC 2001



并行计算发展的驱动力

- 架构发展趋势

如何提高CPU的处理速度？

1990年之前的解决方式：

1. 增加时钟频率（扩频）
   a. 深化流水线（采用更多 / 更短的流水阶段）
   b. 芯片的工作温度会过高
2. 推测超标量（Speculative Superscalar, SS）
   多条指令同时执行（指令级的并行，ILP）：
   a. 硬件自动找出串行程序中的能够同时执行的独立指令集合；
   b. 硬件预测分支指令；
   在分支指令实际发生之前先推测执行；

局限：最终出现“收益下降（diminishing returns）”

这种解决方法的优点：程序员并不需要知道这些过程的细节

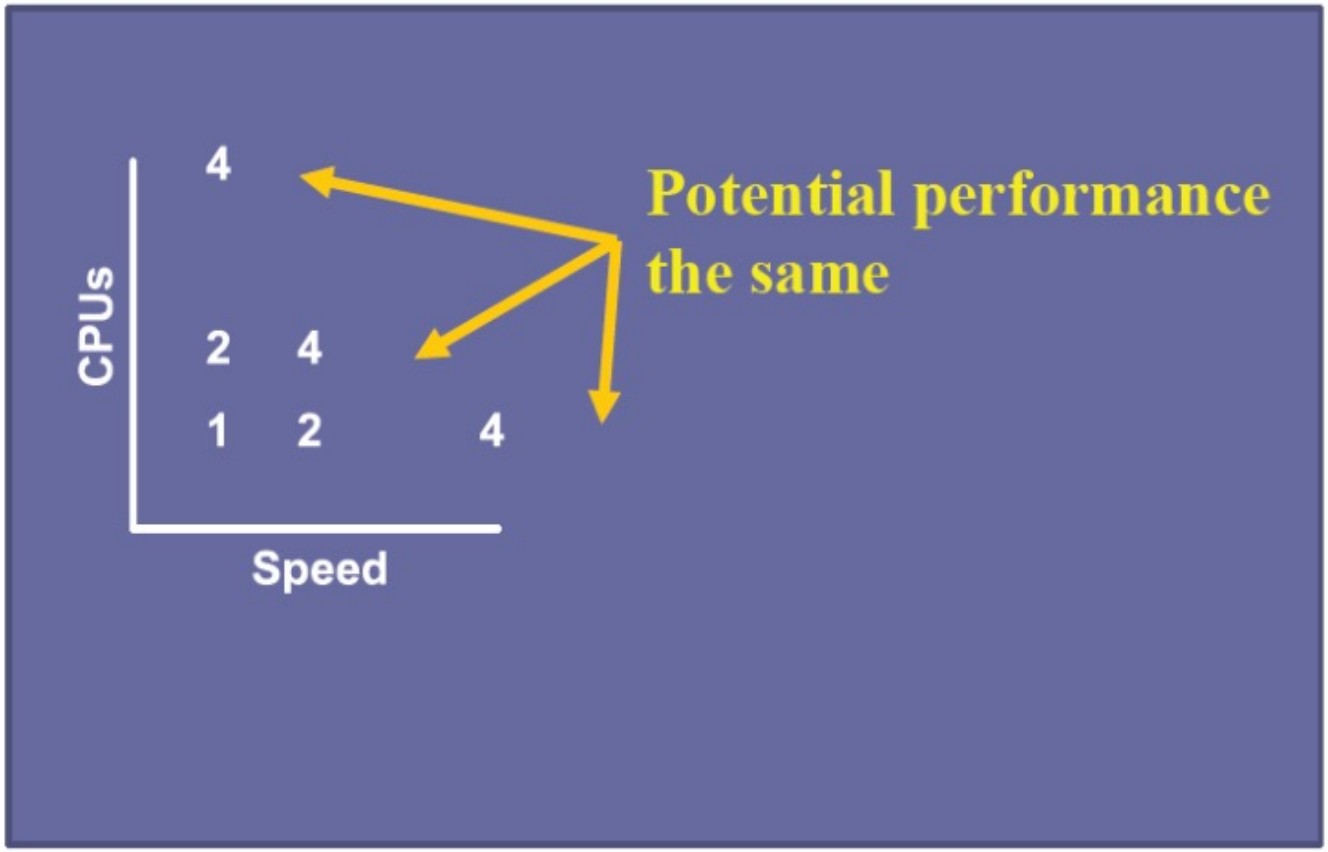
2000年之后的解决方式：

1. 时钟频率很难增加；
2. SS触到天花板出现“收益下降”；
3. **利用额外的额外的晶体管在芯片上构建更多 / 更简单的处理器；**

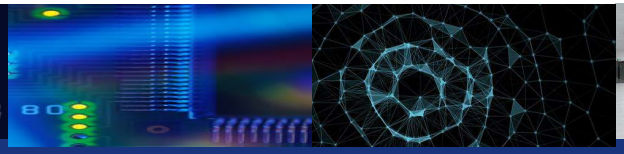
并行计算发展的驱动力

- 架构发展趋势

简化的处理器如何提高性能？



Source: Intel



并行计算发展的驱动力

- 架构发展趋势

近年来Intel处理器的发展

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

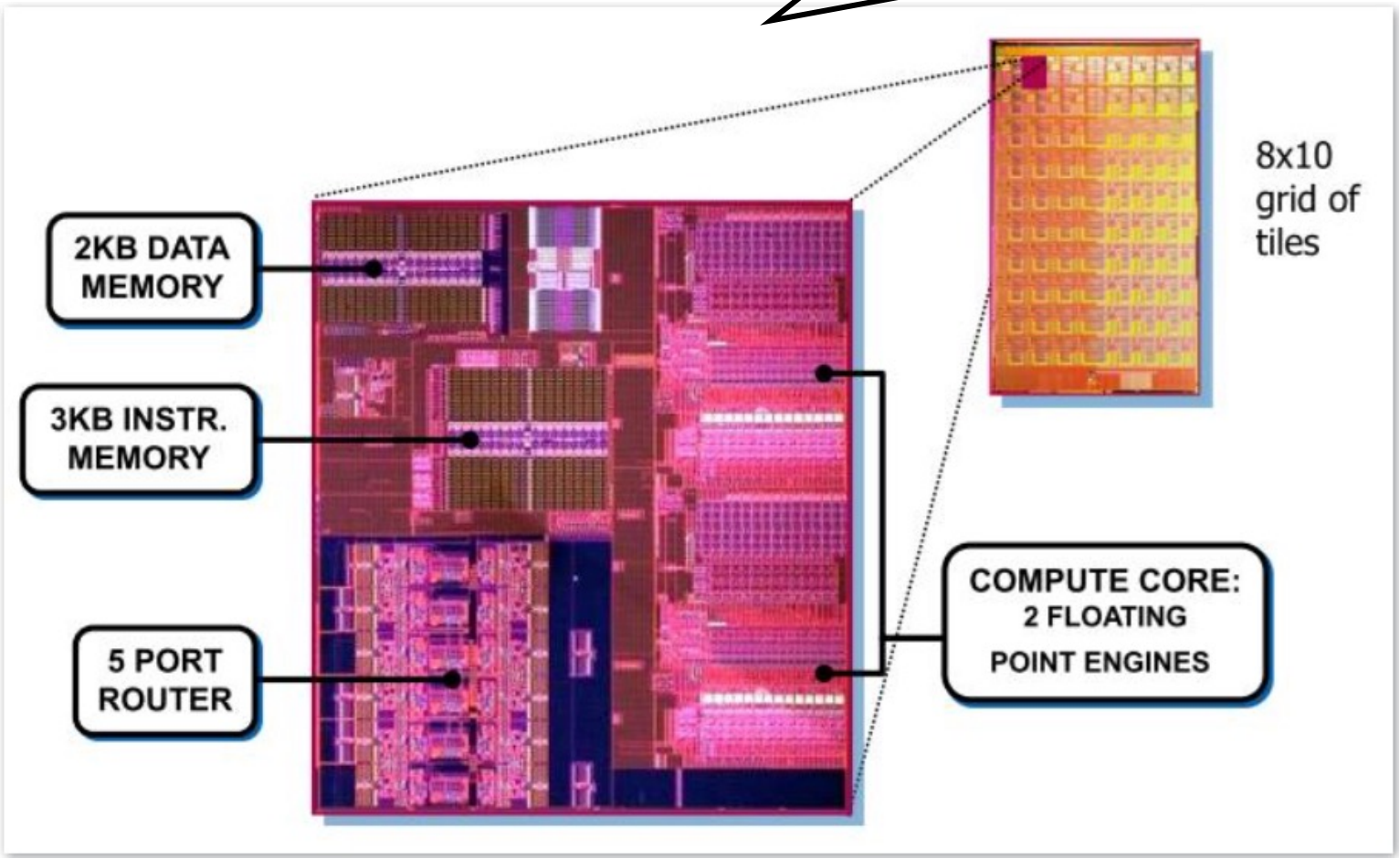
We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry." Intel President Paul Otellini, IDF 2005

并行计算发展的驱动力

- **架构发展趋势**

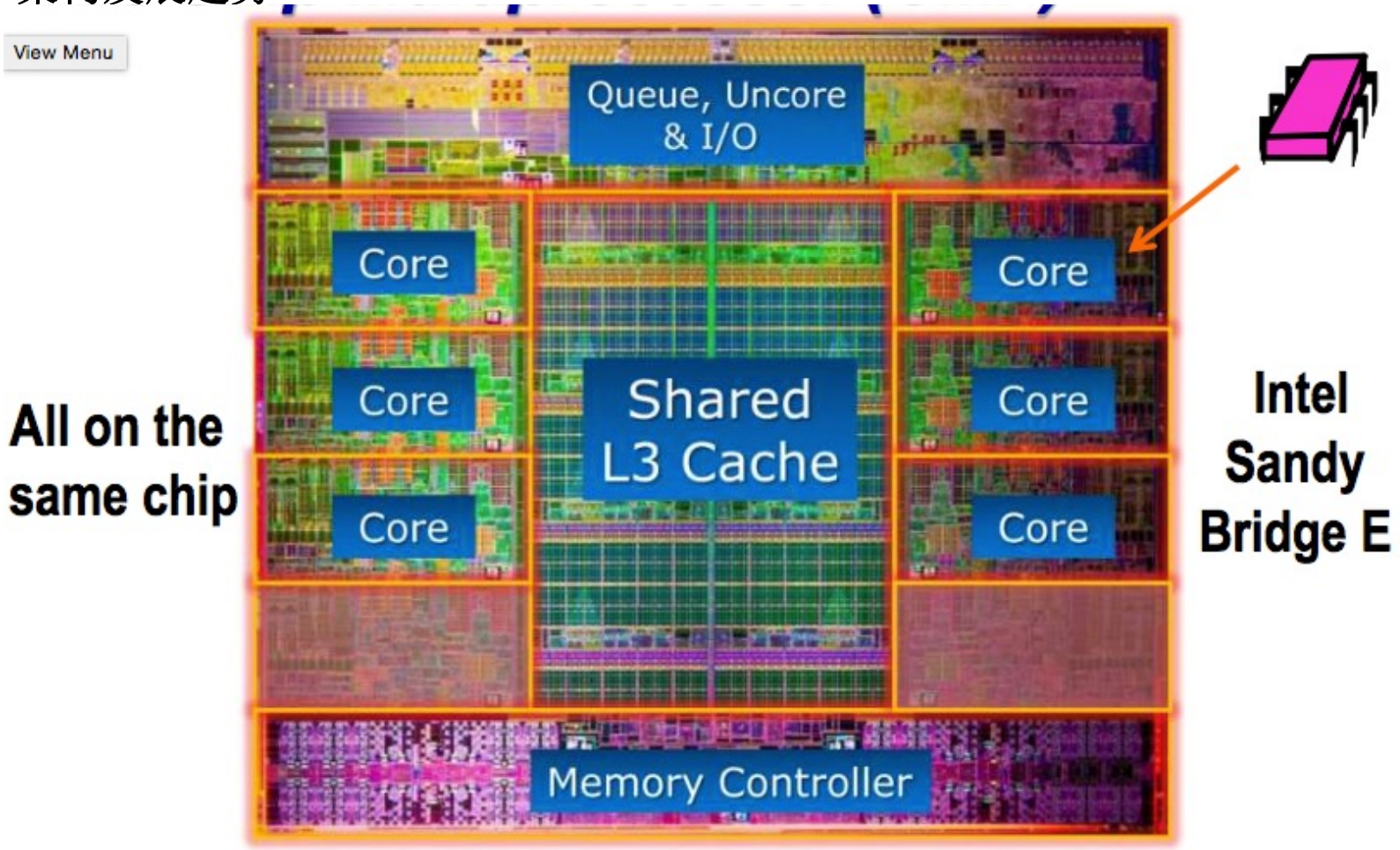
  Intel的多核和众核处理器

Intel80-core的TeraScale处理器，单精度求解达到1TFLOP只需要97Watts



并行计算发展的驱动力

- 架构发展趋势



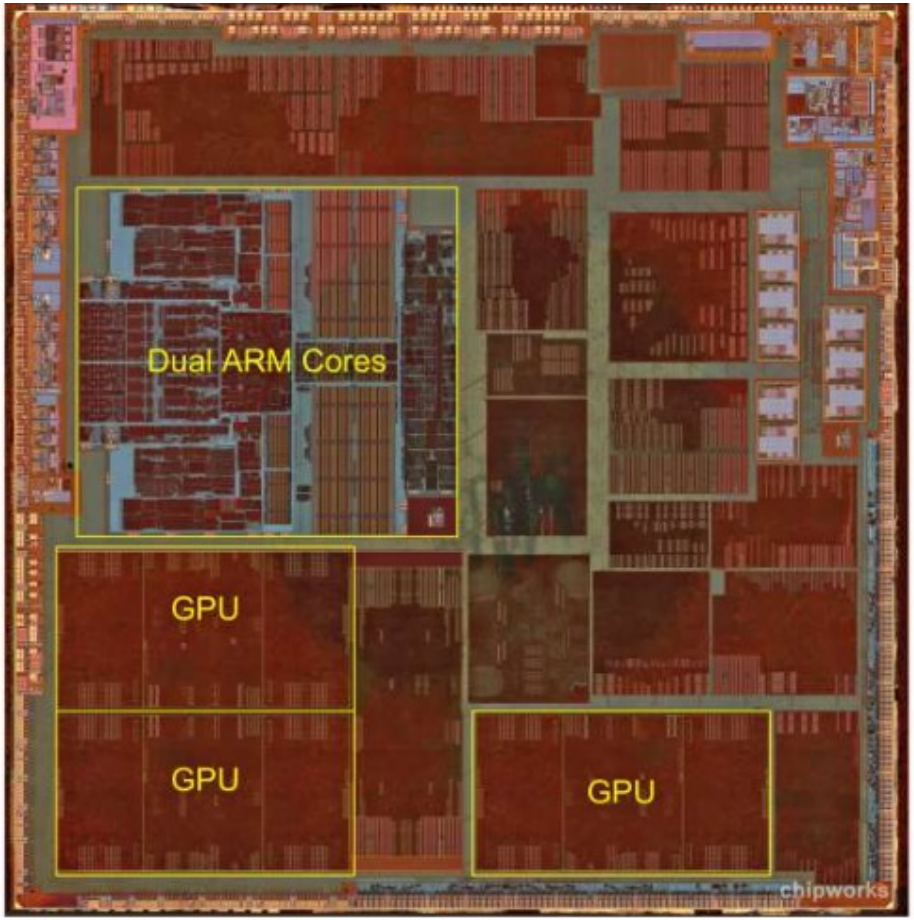
Source: Anand Lal Shimpi, "Intel Core i7 3960X Review", http://www.anandtech.com

4

并行计算发展的驱动力

- 架构发展趋势

苹果处理器架构



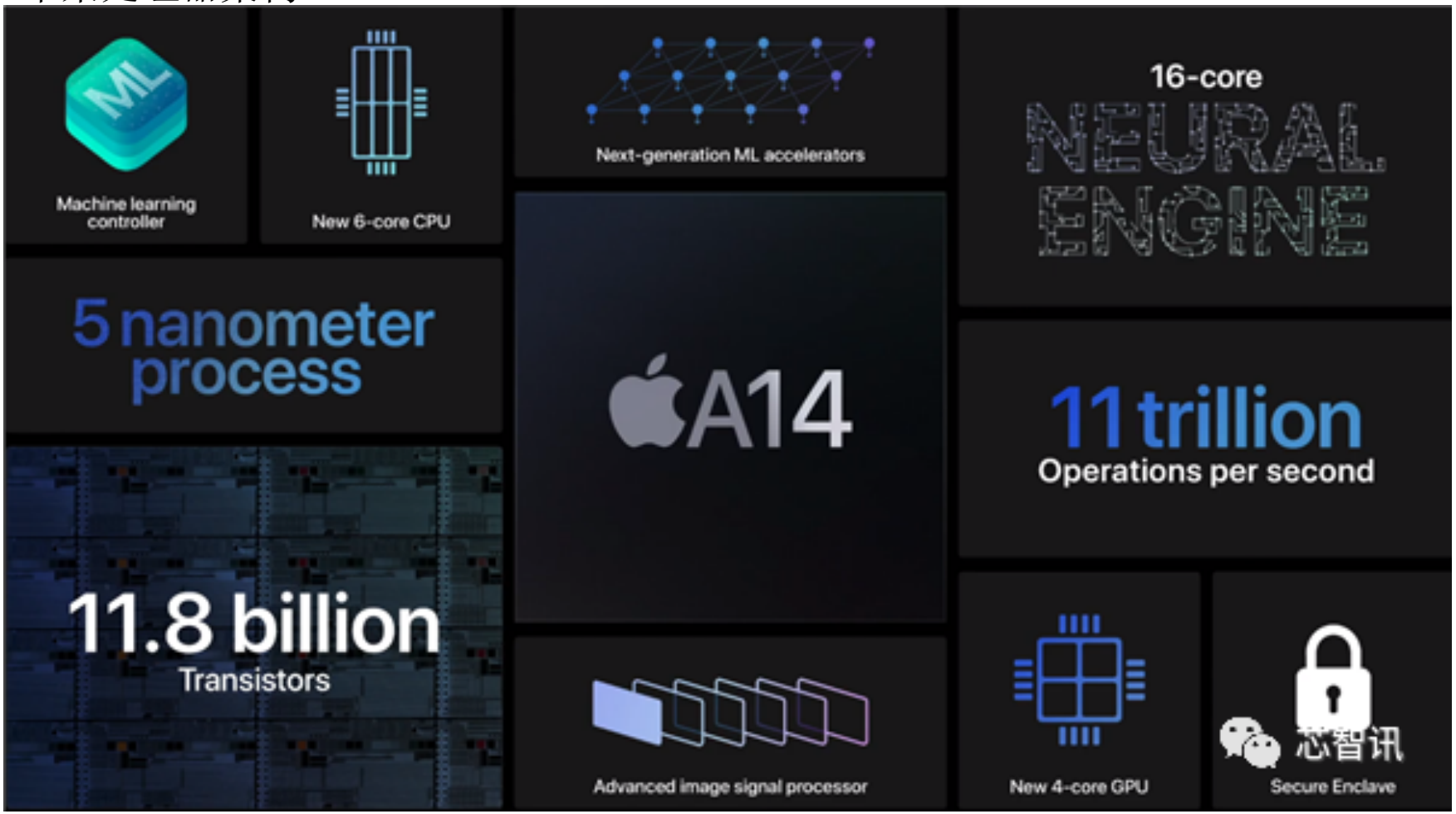
Apple A6

Source: Apple A6 Teardown, http://www.ifixit.com

并行计算发展的驱动力

- 架构发展趋势

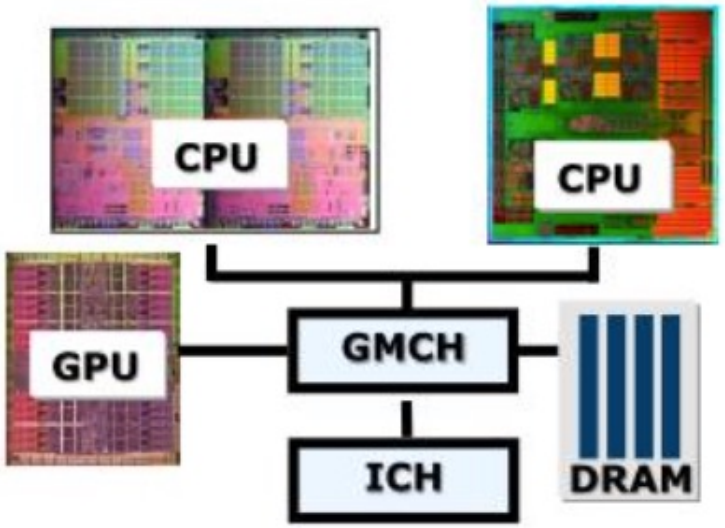
苹果处理器架构



并行计算发展的驱动力

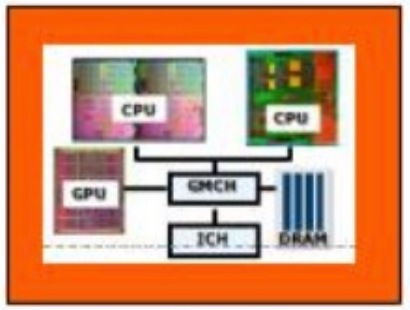
- 架构发展趋势

  未来属于异构、多核的SOC架构，SOC = System On Chip



- A modern platform has:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?

- And SOC trends are putting this all onto one chip

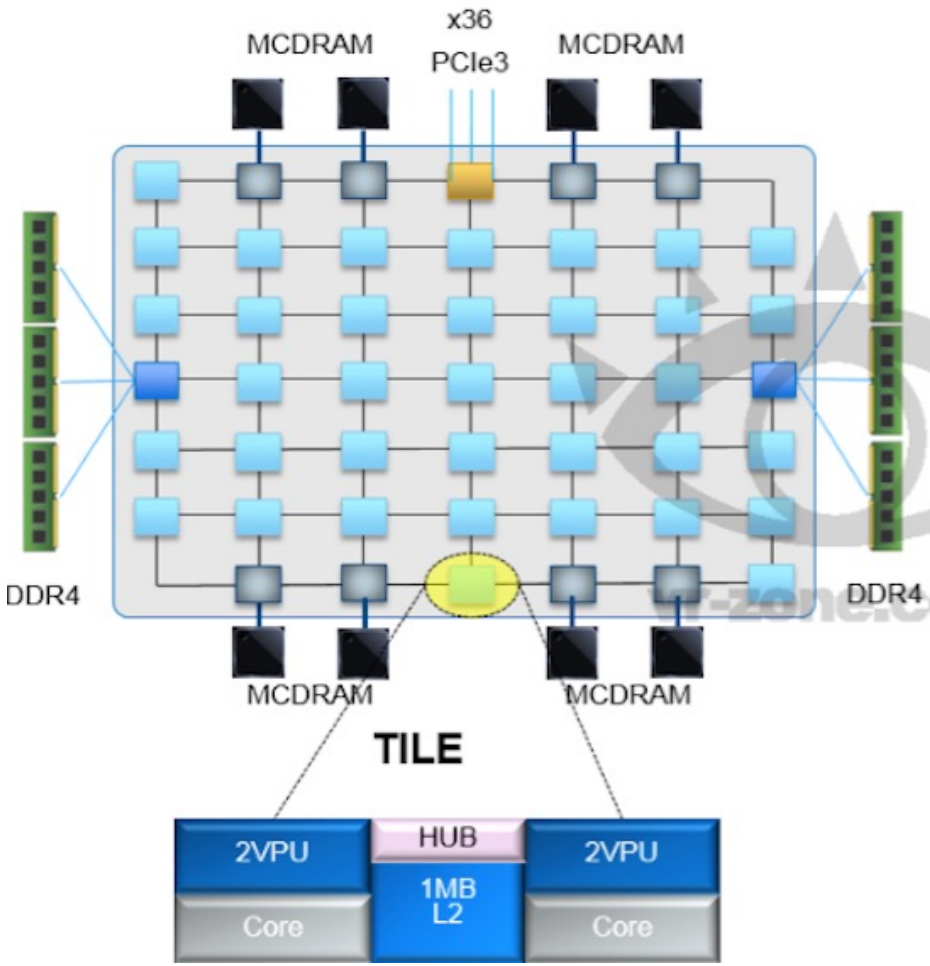


Source: Tim Mattson, Intel Labs

并行计算发展的驱动力

- 架构发展趋势

Intel 72-core X86 Knights Landing CPU专用于Exascale超算



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

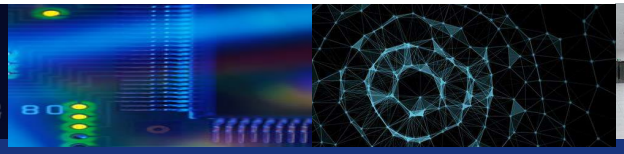
Full Intel® Xeon processor ISA compatibility through AVX-512 (except TSX)

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



并行计算发展的驱动力

- **架构发展趋势**
  所有的超级计算机都在使用众核处理器

2021年第57次Top500排名

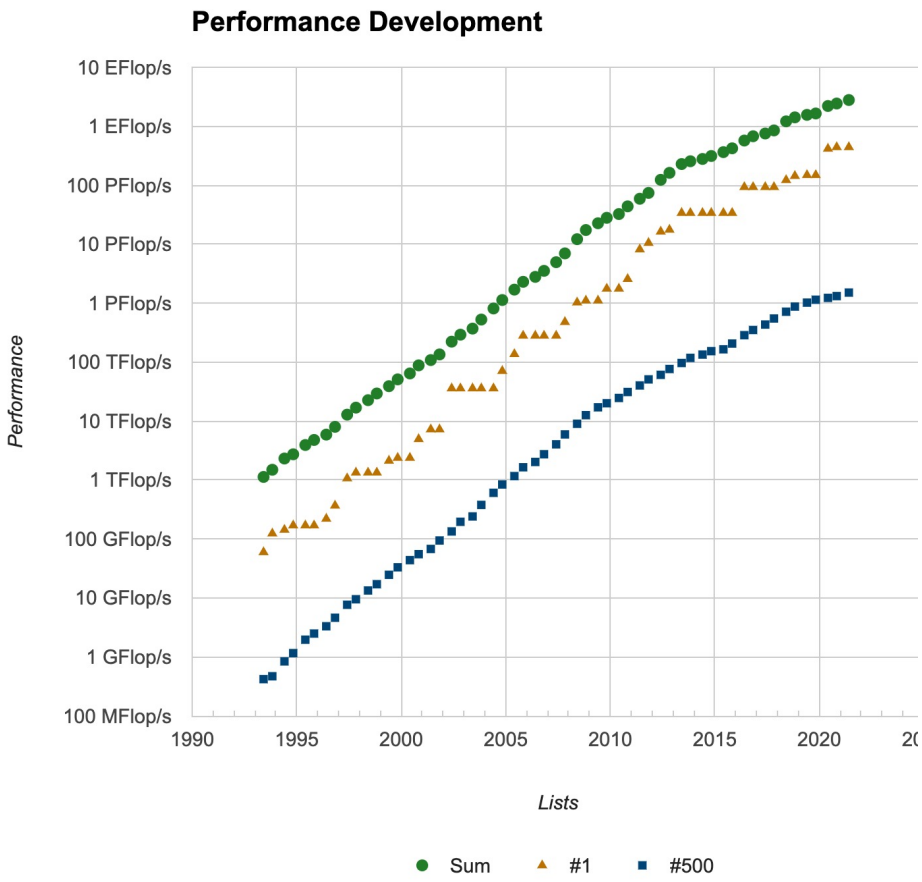
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, 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	**Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	706,304	64,590.0	89,794.5	2,528
6	**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

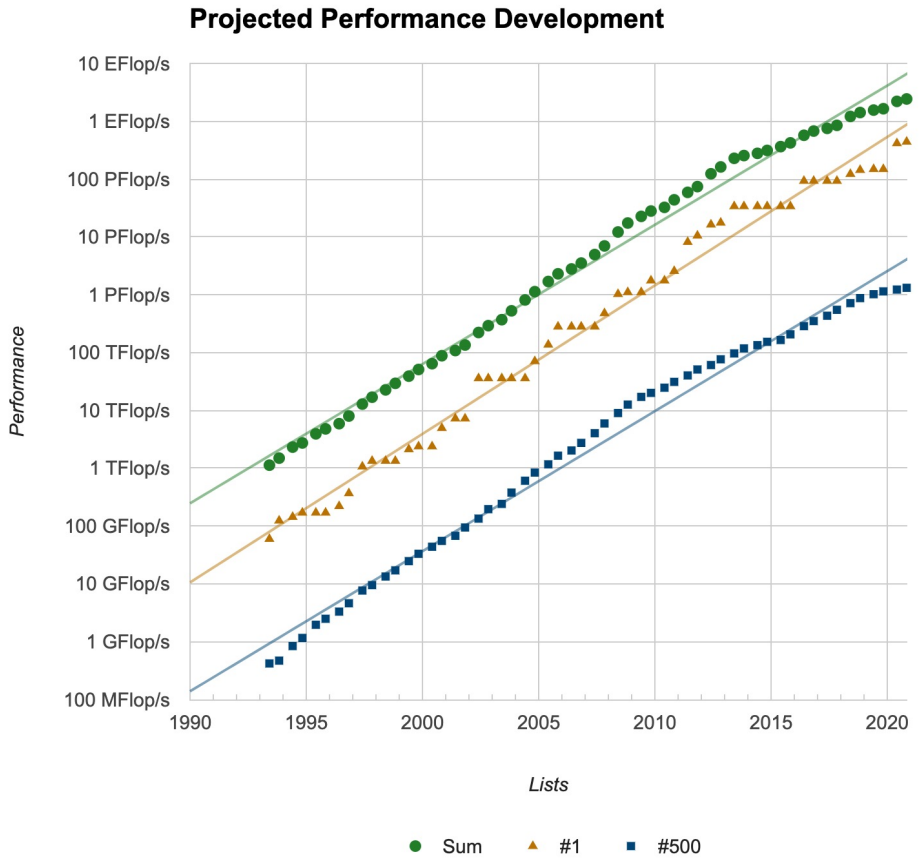
https://www.top500.org/

并行计算发展的驱动力

- 架构发展趋势

所有的超级计算机都在使用众核处理器

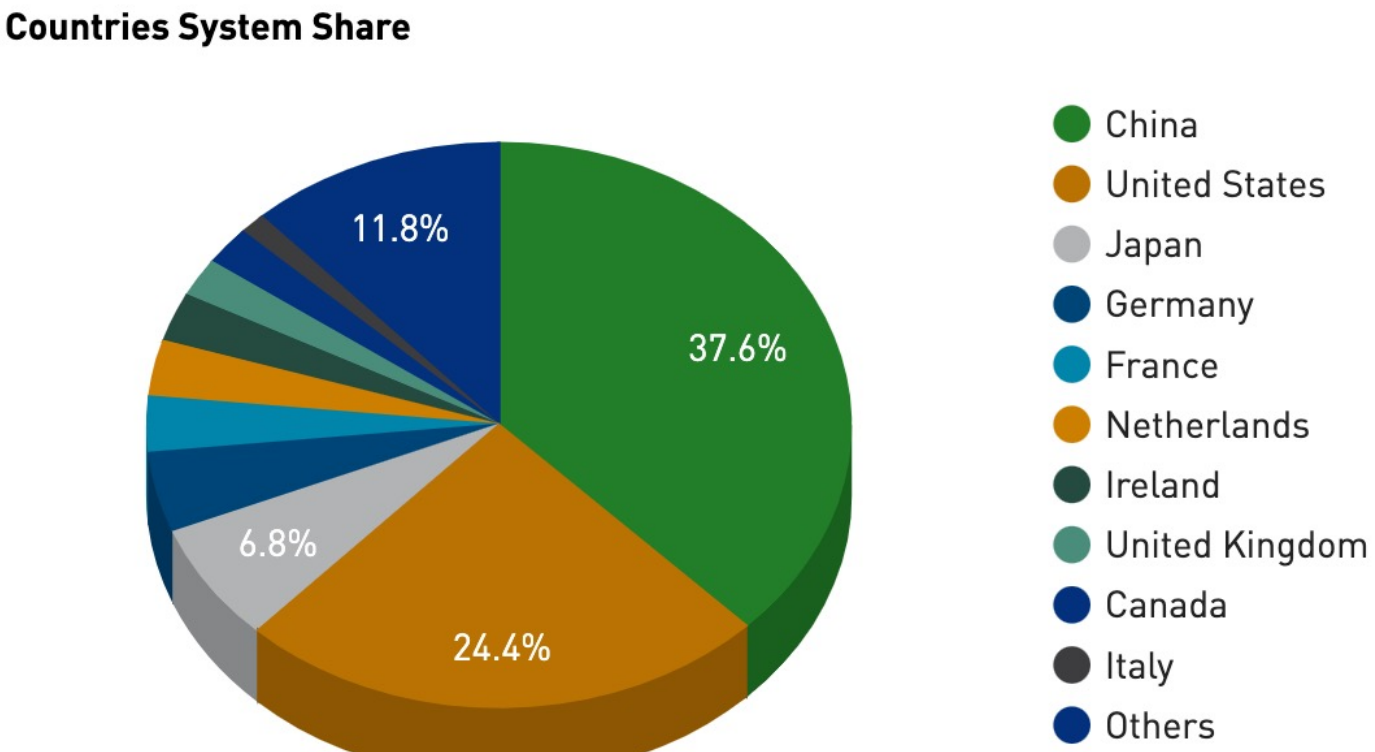




并行计算发展的驱动力

- 架构发展趋势

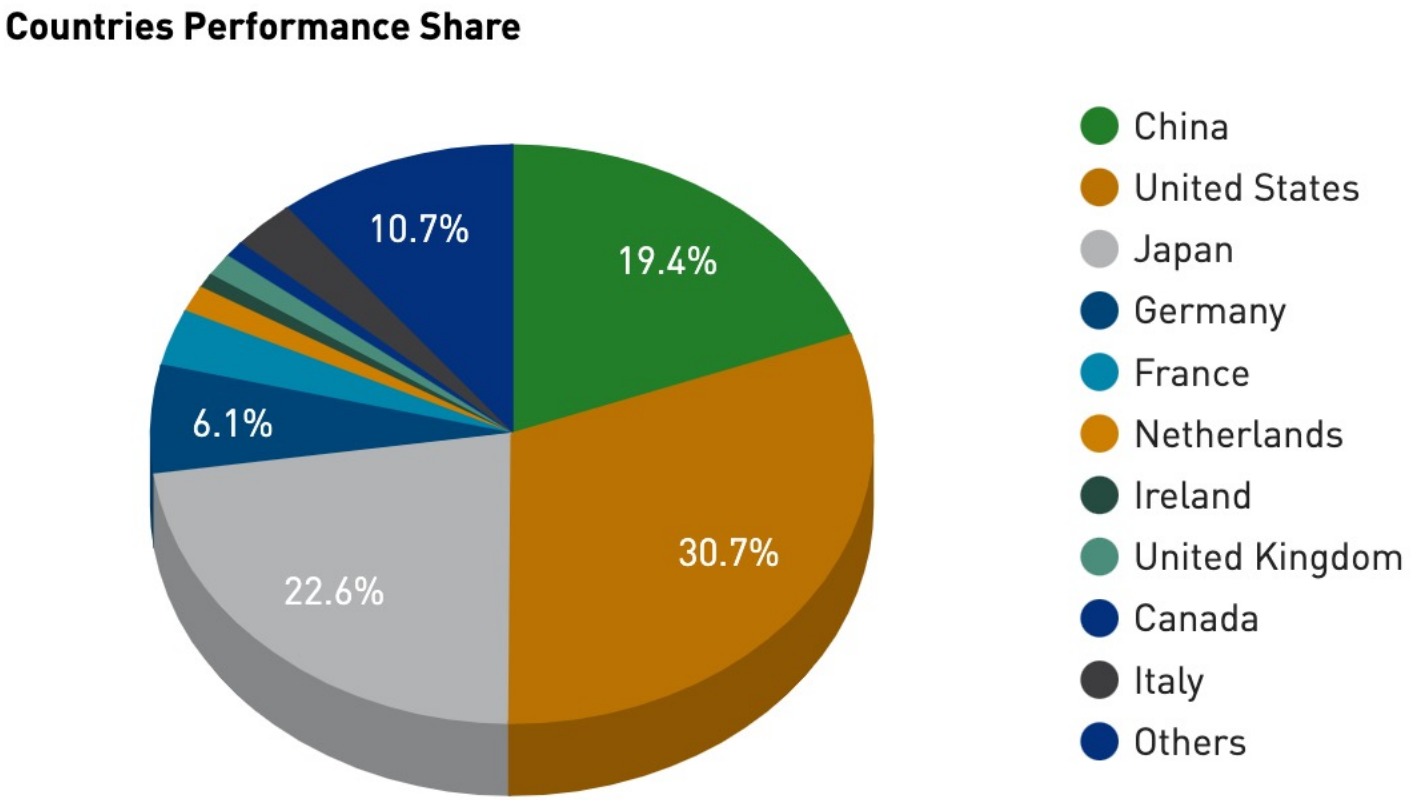
Top500在不同国家的分布

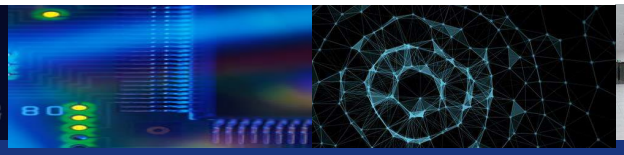


并行计算发展的驱动力

- 架构发展趋势

Top500在不同国家的分布





并行计算发展的驱动力

- 架构发展趋势

神威太湖之光

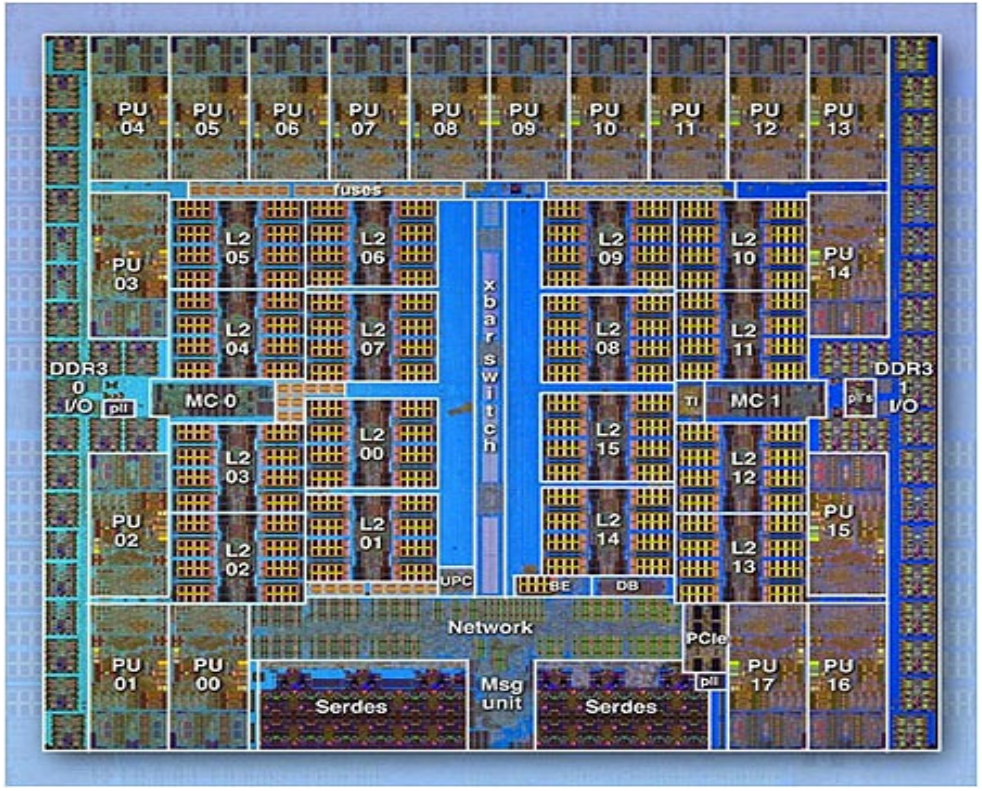
Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway

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

并行计算机

从硬件角度来讲，今天的单个计算机都是并行计算机，主要体现为：

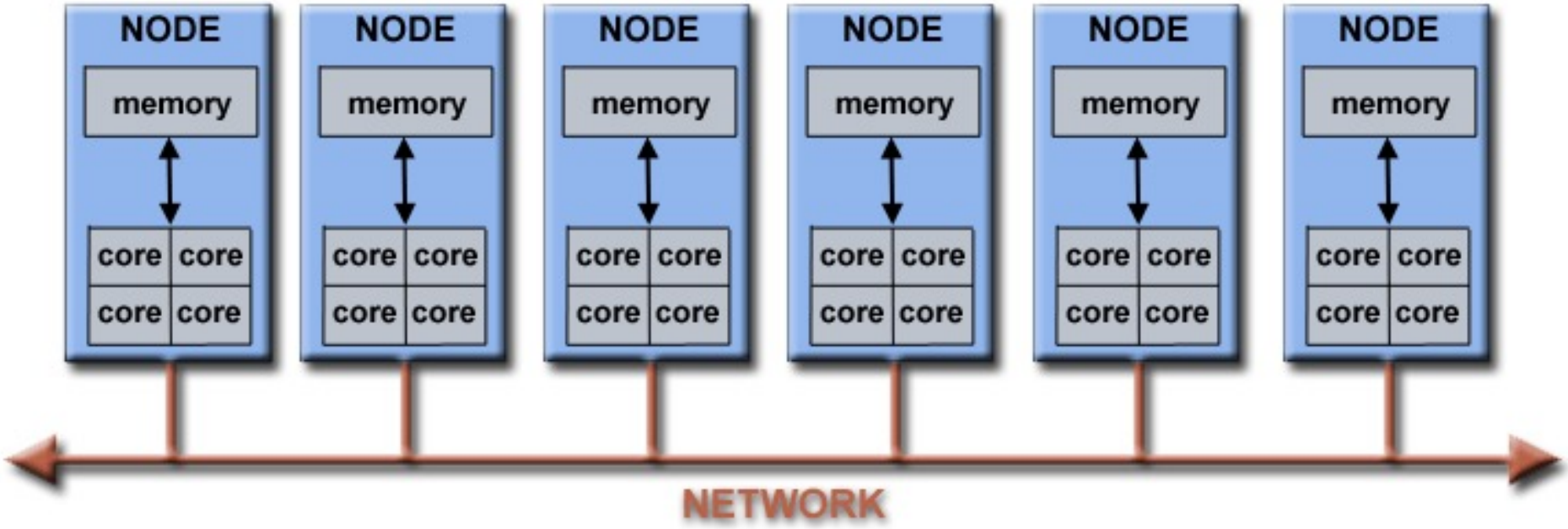
- 多个功能单元（L1 Cache、L2 Cache、Branch、Prefetch、GPU等）；
- 多个执行单元或者核心
- 多个硬件线程



IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

并行计算集群

多个单独的计算机通过网络连接起来形成计算集群



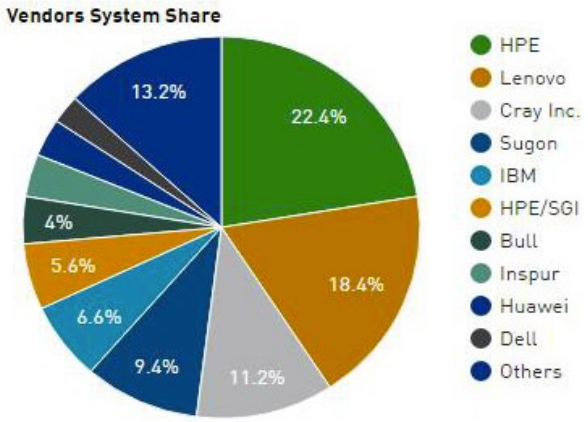
计算集群

并行计算集群

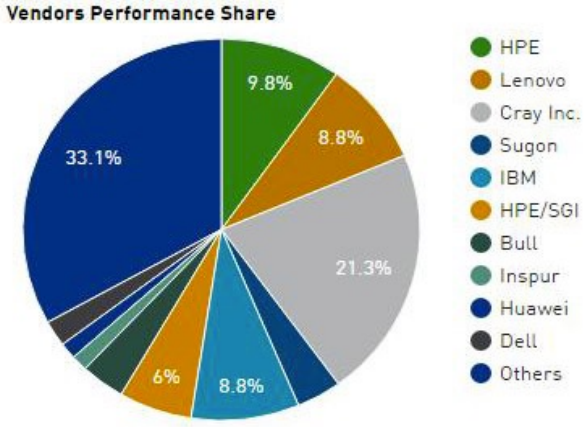
LLNL并行计算集群

- 每个节点都是一个多处理器并行机；
- 多个计算节点通过Infiniband网络连接；

并行计算集群



主要的超级计算机厂商



Vendors	Count	System Share (%)
HPE	112	22.4
Lenovo	92	18.4
Cray Inc.	56	11.2
Sugon	47	9.4
IBM	33	6.6
HPE/SGI	28	5.6
Bull	20	4
Inspur	18	3.6
Huawei	16	3.2
Dell	12	2.4
Fujitsu	11	2.2
Penguin Computing	9	1.8
NUDT	4	0.8
Lenovo/IBM	4	0.8
NEC	3	0.6
T-Platforms	3	0.6
Atipa	3	0.6
IBM/Lenovo	3	0.6
NRCPC	2	0.4
MEGWARE	2	0.4
RSC Group	2	0.4
Self-made	1	0.2
IPE, Nvidia, Tyan	1	0.2
DALCO	1	0.2
Adtech	1	0.2
ClusterVision	1	0.2

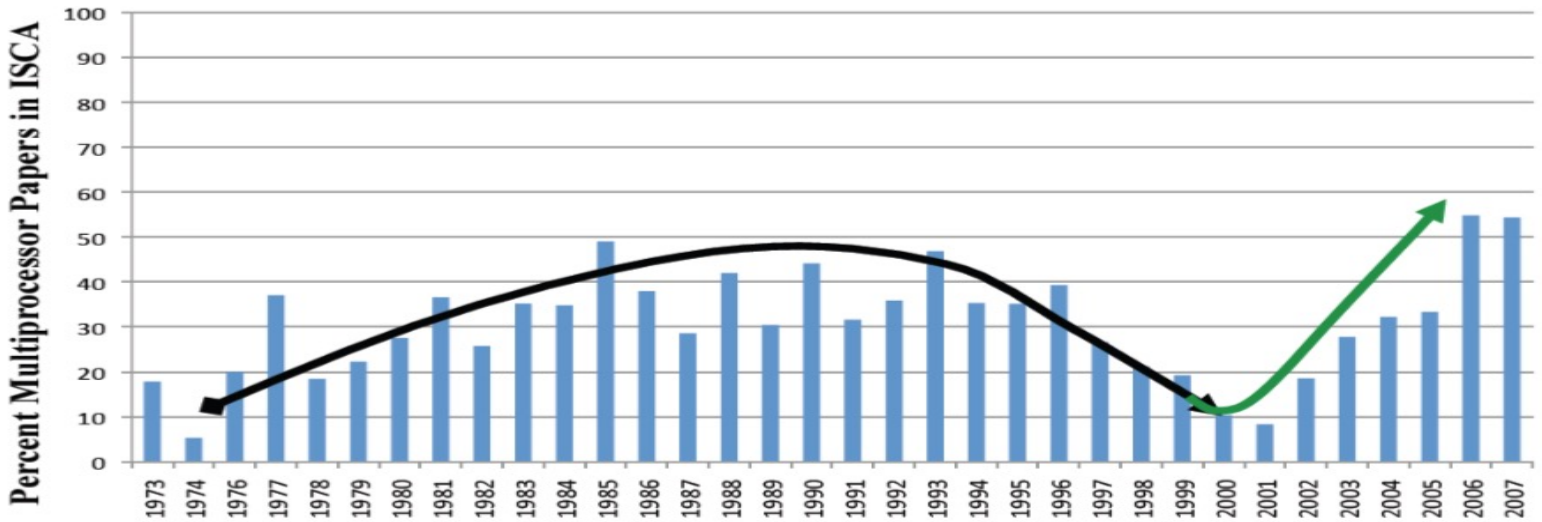
Source: Top500.org

并行计算发展的驱动力

Moore’s law新解：

1. 每两年芯片上的核心数目会翻倍；
2. 时钟频率不再增加，甚至是降低；
3. 需要处理具有很多并发线程的系统；
4. 需要处理芯片内并行和芯片之间的并行；
5. 需要处理异构和各种规范（不是所有的核都相同）；

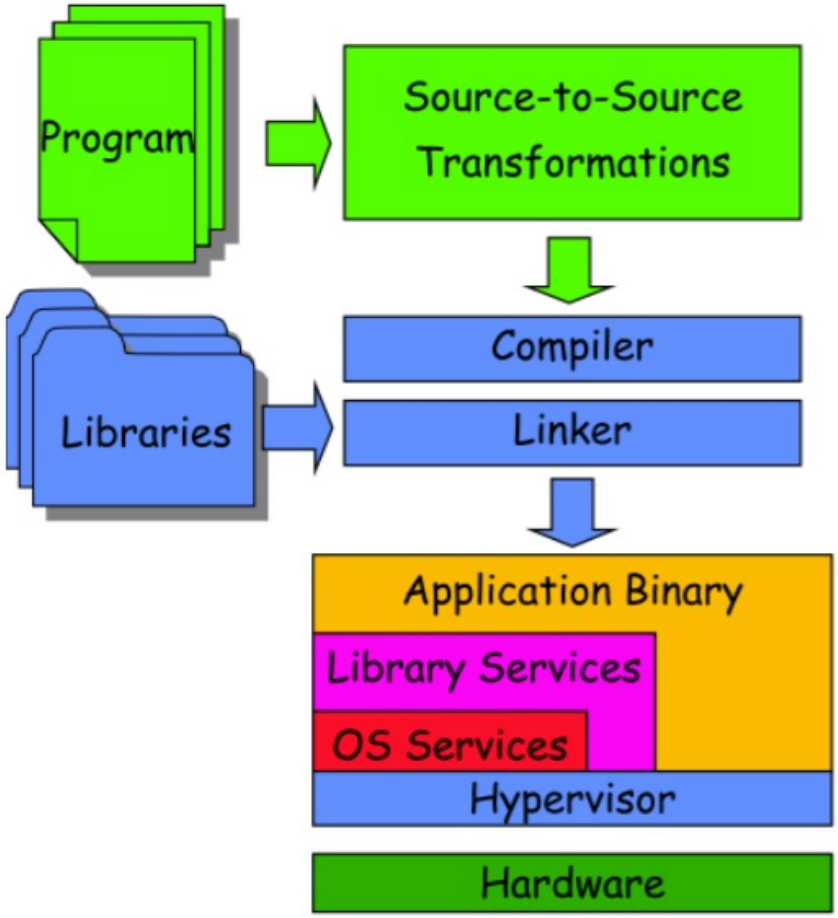
……



Source: Mark Hill, 2/2007

并行计算也不是一直都很流行

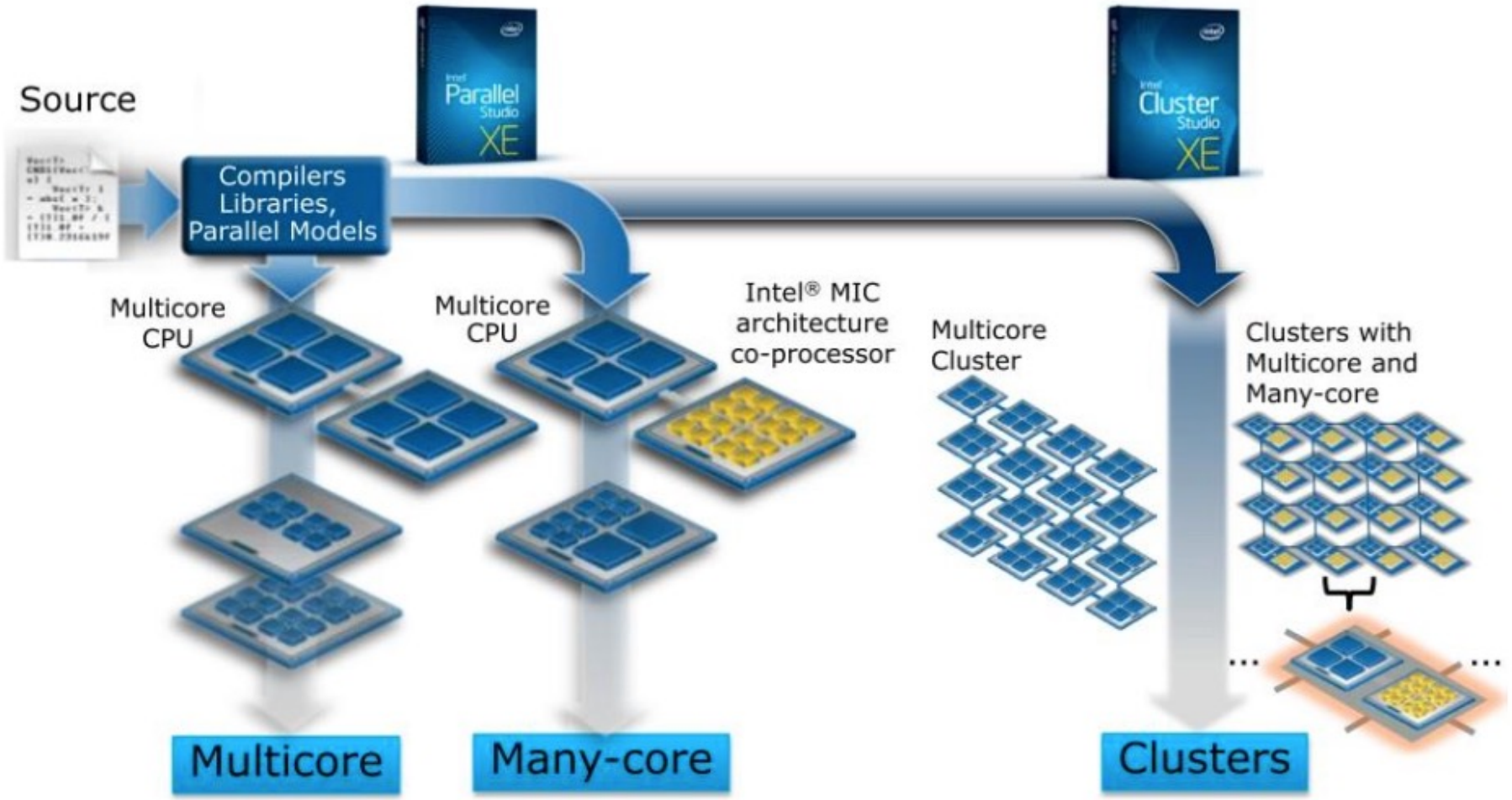
并行计算不仅仅是硬件

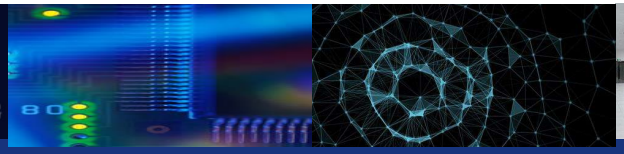


- VAX 指令的谬论：
  1. 对于高阶抽象只用一条指令表示；
  2. 单条硬件指令真的快吗？
- RISC指令的设计思想：
  1. 全系统设计；
  2. 从全局角度设计硬件机制；
  3. 跨组件优化
- 当前程序员看不到汇编语言，甚至看不到底层C语言。

并行编程！

从单节点串行计算到多核、众核的并行计算






并行编译器的局限

经过30多年的研究：

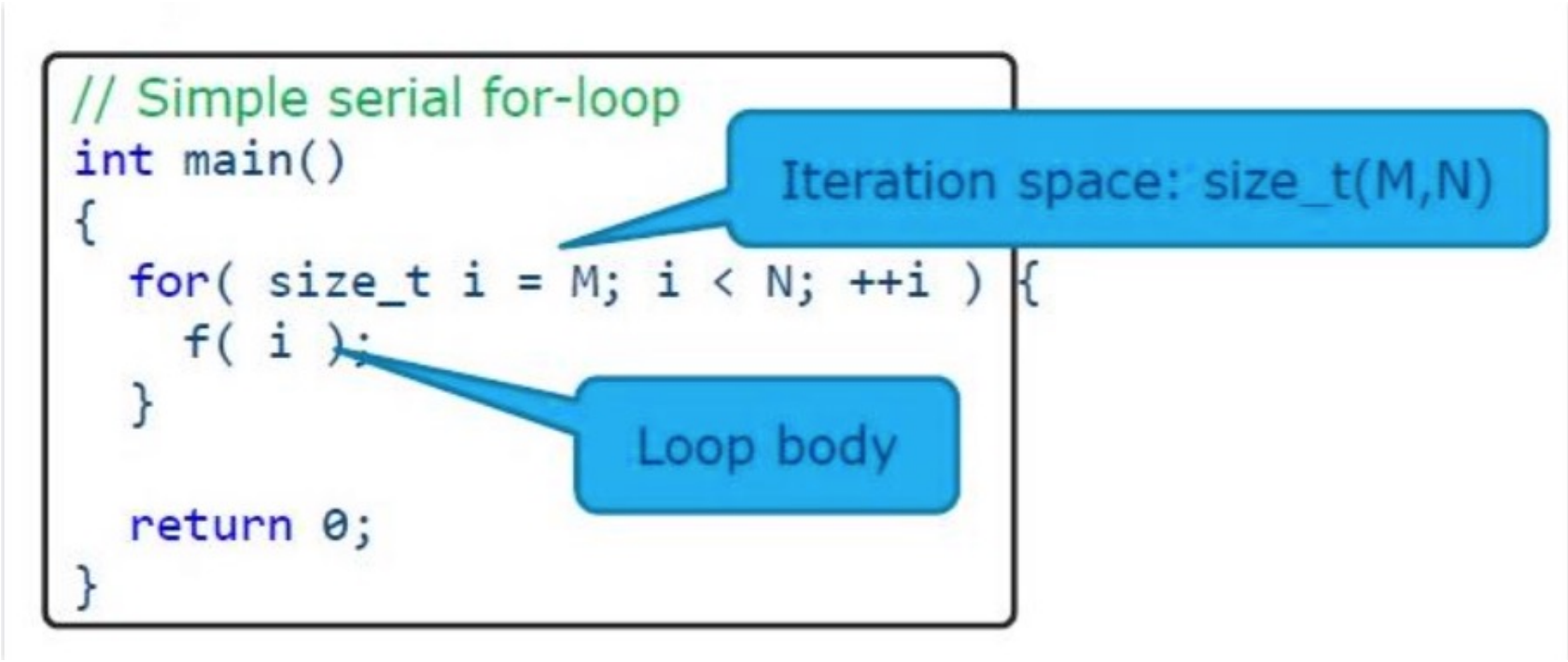
- 编译器仅做到了有限的并行检测核程序转换：

  1. 执行级的并行，只能检测到basic block；
  2. 少量的嵌入的 for-loops；
  3. 分析技术如数据依赖分析、指针分析、流敏感分析等，难以应用到大规模程序中；

- 比较成功的做法是让人去学习如何编写并行程序；

并行编程的例子

Loop 循环是一种能够并行化的简单代码段，看一个for-loop并行化的例子：



并行编程的例子

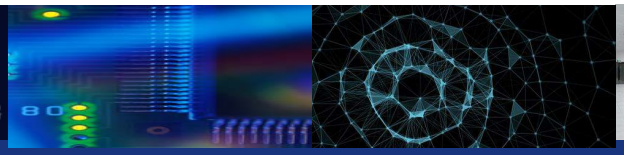
- Step 1

```
#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



并行编程的例子

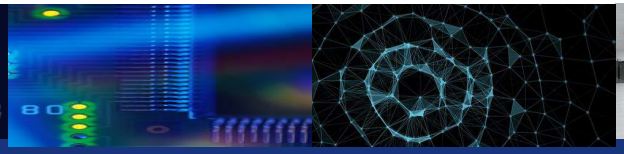
- Step 2

```
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;
}
```

Divide iteration space into to 4 chunks and create 4 worker threads

Create worker threads

Wait for/join worker threads



多种改进方法

Problems with Naïve Implementation	What You Could Do to Improve It
Works with *fixed number of threads*	
The implementation is *not portable*	
The solution is *not re-usable*	
Potentially *poor performance* due to work-load imbalance	
The solution is *not composable*	

... More coding...

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

并行编程的难点

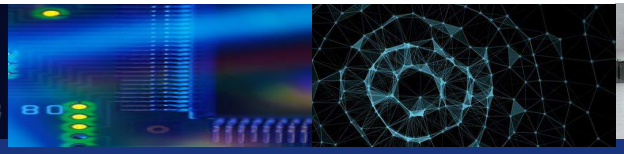
- 找到尽可能多的并行点（Amdahl’s Law）
- 粒度

  函数级并行、线程级并行还是进程级并行
- 局部性

  并行化后是否能够利用局部数据等
- 负载均衡

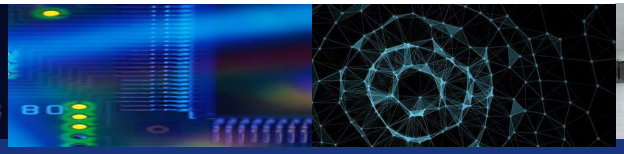
  不同线程、不同 core之间的负载分布
- 协作与同步
- 性能模型

所有这些难点让并行编程要比串行编程复杂得多



并行编程与串行编程的比较

- 编程代价不同、优势不同；
- 并行编程需要不同的，不熟悉的算法；
- 并行编程必须利用不同的问题抽象；
- 并行程序的行为更复杂；
- 更难以控制程序不同组件之间的交互；
- 需要掌握更多编程工具、更多知识等；



并行编程样例

计算 n 个值，并将这些值相加。

串行的方法：

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

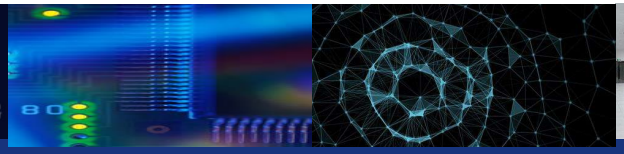
并行编程样例

计算 n 个值，并将这些值相加。

- 假定我们有 p 个 cores，p 远远小于 n;
- 每个核心执行全部加和的一部分，即约为 n/p;

```
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.

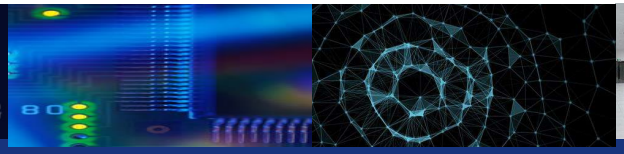


并行编程样例

计算 n 个值，并将这些值相加。

- 每个core计算完毕后，用一个私有变量my_sum保存计算结果。这些core 会把计算结果发送到 “master”core，将所有的my_sum加起来，形成最后的结果。

```
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;
}
```



并行编程样例

计算 n 个值，并将这些值相加。

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

Global sum

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

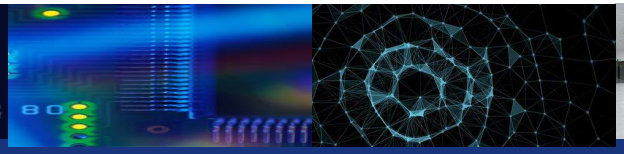
For master, there are 7 operations

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

并行编程样例

计算 n 个值，并将这些值相加。

等等！是否还有更好的方法计算全局的sum?



并行编程样例

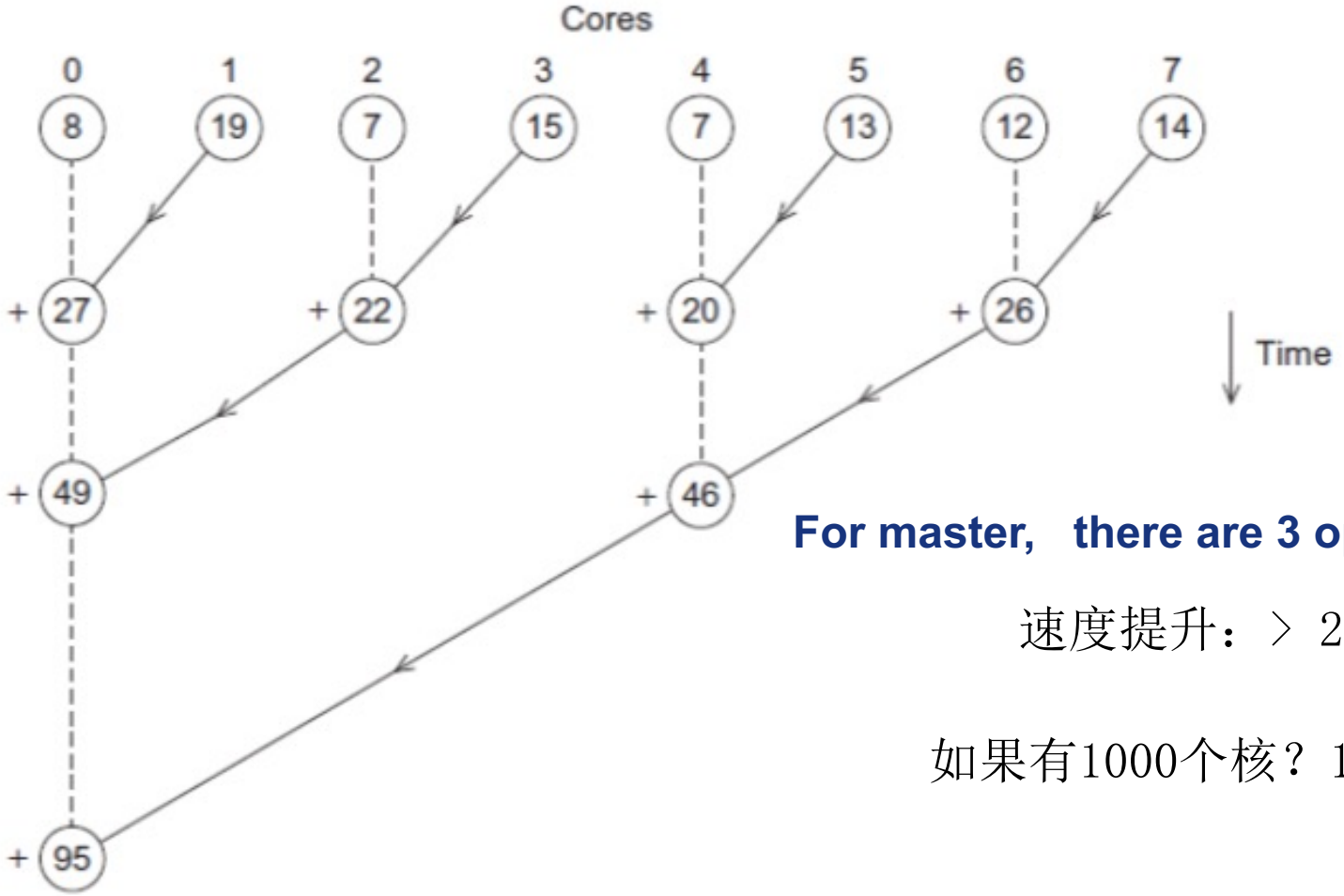
计算 n 个值，并将这些值相加。

- 不让 Master core 做所有的工作；
- 加和任务与其他core共享；
- 两个core成对加和即core 0负责将自己的结果与core 1的结果相加；
- Core 2负责将自己的结果与core 3的结果相加；
- 重复这个过程直到仅剩最后一个core有新的结果；

Core 0 加 core 2的结果；
Core 4 加 core 6的结果，等等；

并行编程样例

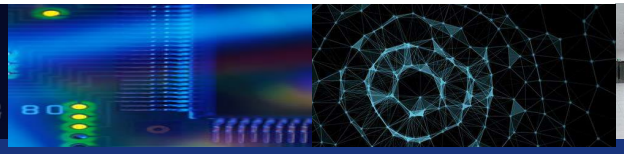
计算 n 个值，并将这些值相加。



For master, there are 3 operations

速度提升：> 2倍

如果有1000个核？10000？



如何编写并行程序？

计算密集型，可拆分成不同的子任务

- 任务并行：

  将一个问题划分成多个不同的子任务分别在不同的core上运行；

- 数据并行：

  1. 将问题所涉及的数据拆分到不同的core上执行；
  2. 每个core运行的是相同的程序；

数据密集型，大数据应用

如何编写并行程序？

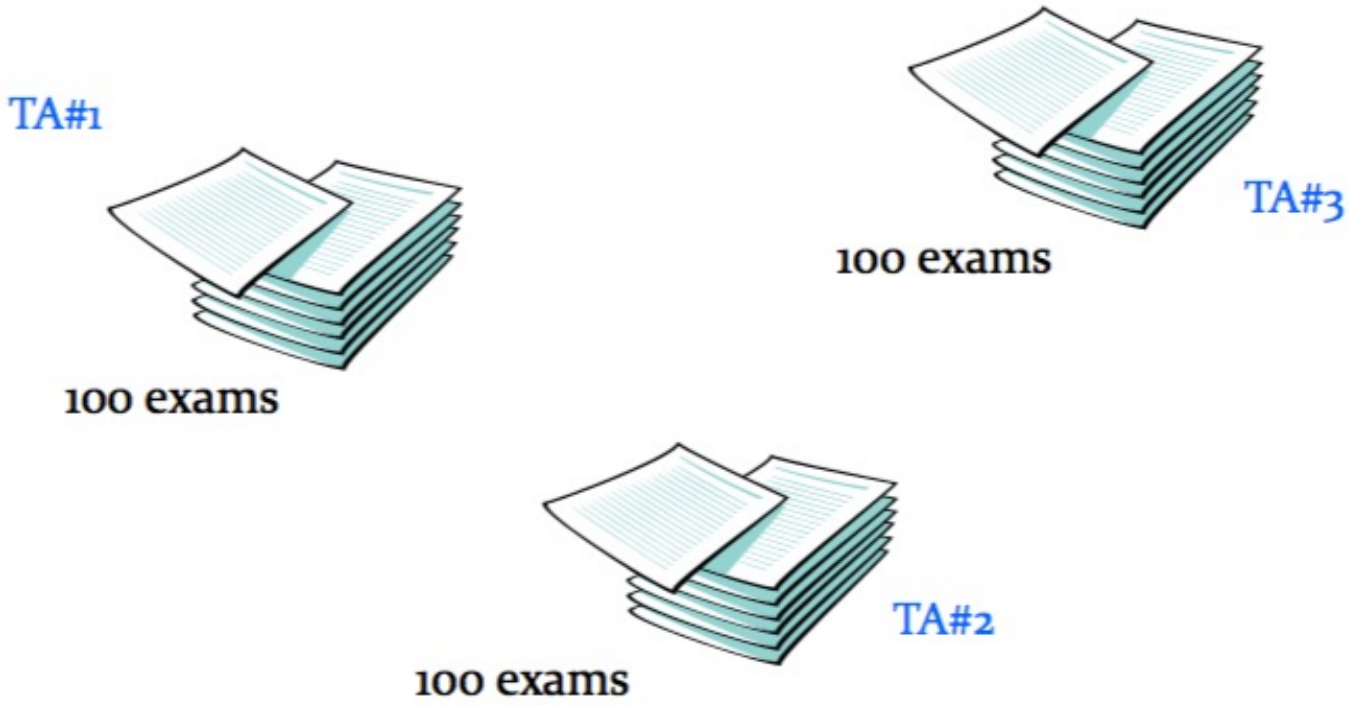
15 questions

300 exams

如何编写并行程序？

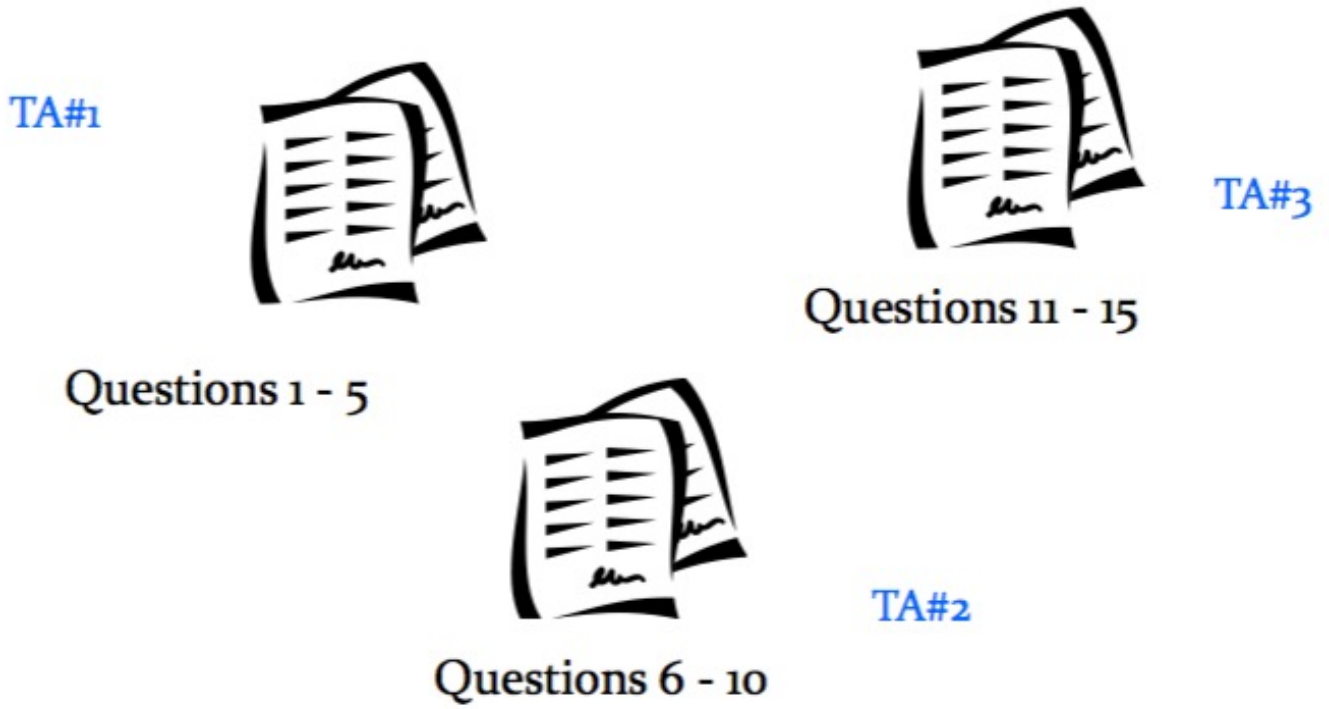
如何编写并行程序？

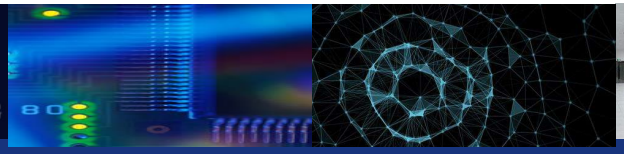
数据并行：



如何编写并行程序？

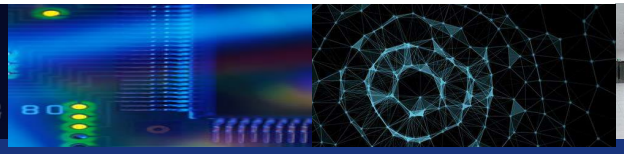
任务并行：






并行程序之间的协作

- Cores之间通常需要相互协作；
- 通信： 一个或者多个core会将他们当前的计算结果发送到其他core上；
- 负载均衡： 在cores 之间平衡分配任务，这样不至于出现某个core过载的现象，减少长尾；
- 同步：每个core独立运行，在某些点上，确保core之间的状态同步；



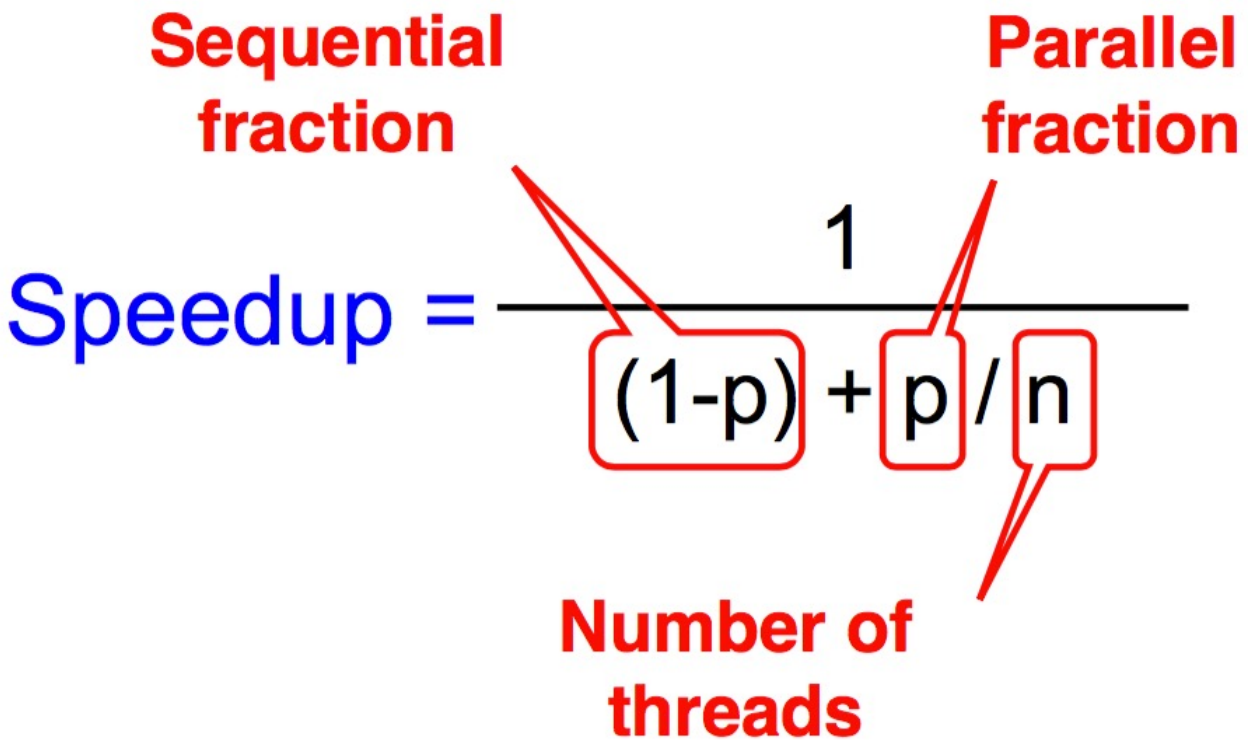
Amdahl’s Law

用于度量并行程序的加速效果。

$$\text{Speedup} = \frac{\text{1-thread execution time}}{\textit{n}\text{-thread execution time}}$$

理想的并行加速效果

Amdahl’s Law

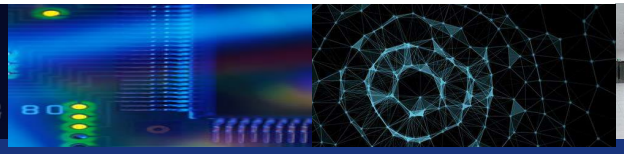


实际的加速效果

Amdahl’s Law

理想丰满，现实骨感的原因：

- 应用程序只有一部分可以并行；
- 大量的串行代码降低并行性能；

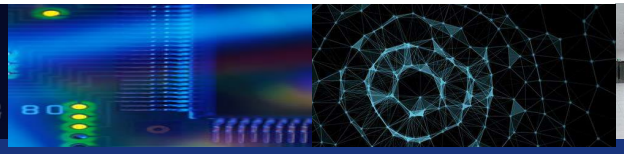


Amdahl’s Law

案例

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}$$

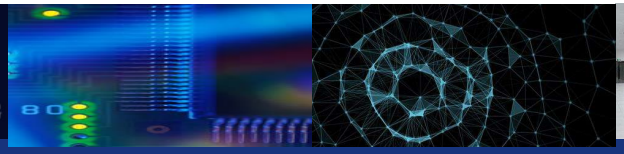


Amdahl’s Law

案例

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}$$

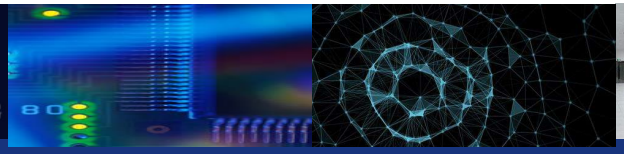


Amdahl’s Law

案例

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$



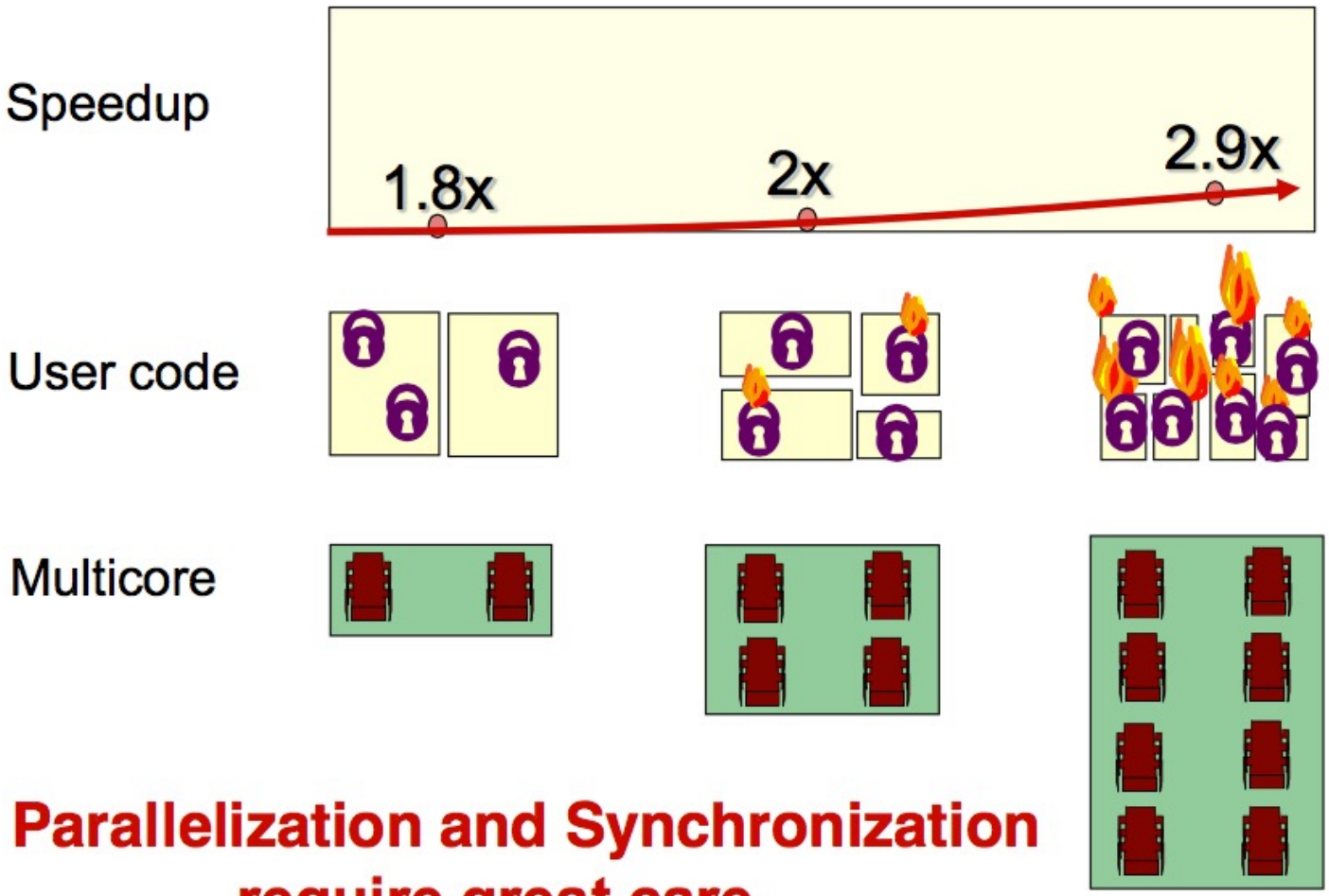
Amdahl’s Law

案例

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}$$

Amdahl's Law


Speedup
1.8x
2x
2.9x
User code
Multicore
Parallelization and Synchronization
require great care…

编程模型

- **Standardized parallel programming platforms**
  - **OpenMP**
  - **Message Passing Interface (MPI)**
  - **Posix Threads (pthreads)**
  - **Open Computing Language (OpenCL)**

- **Why do they help?**
  - **Longer life-cycle for parallel programs**
  - **Code works across platforms**
  - **Automatic scaling?**

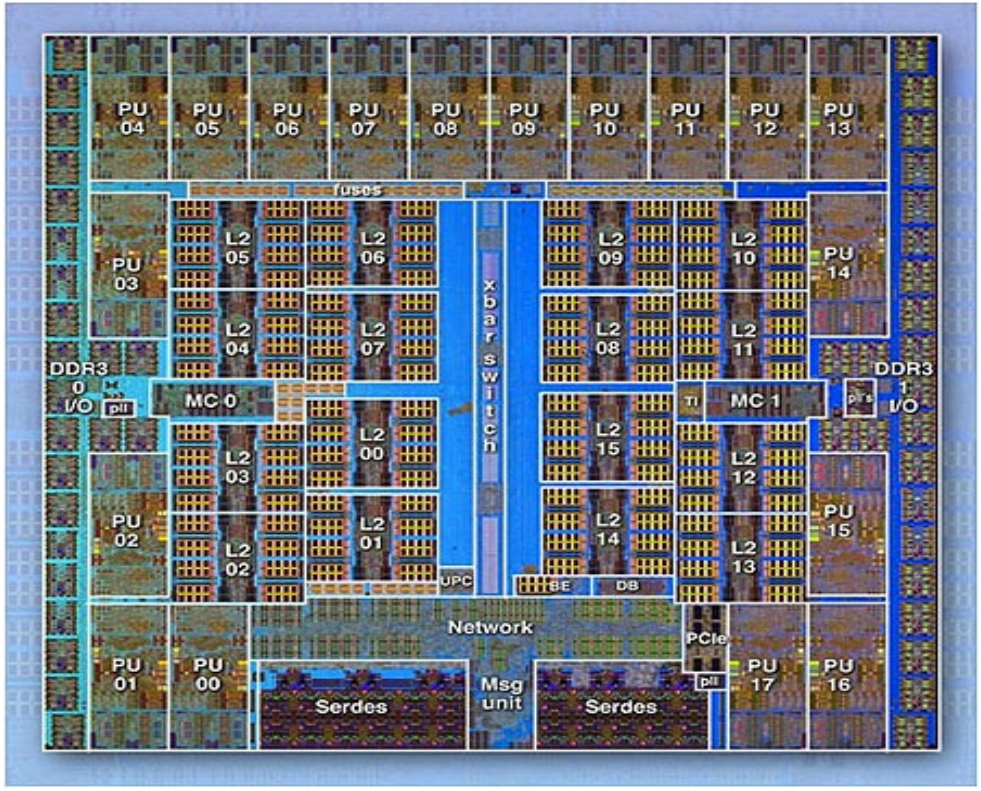
Amdahl’s Law

Backup

并行计算机

从硬件角度来讲，今天的单个计算机都是并行计算机，主要体现为：

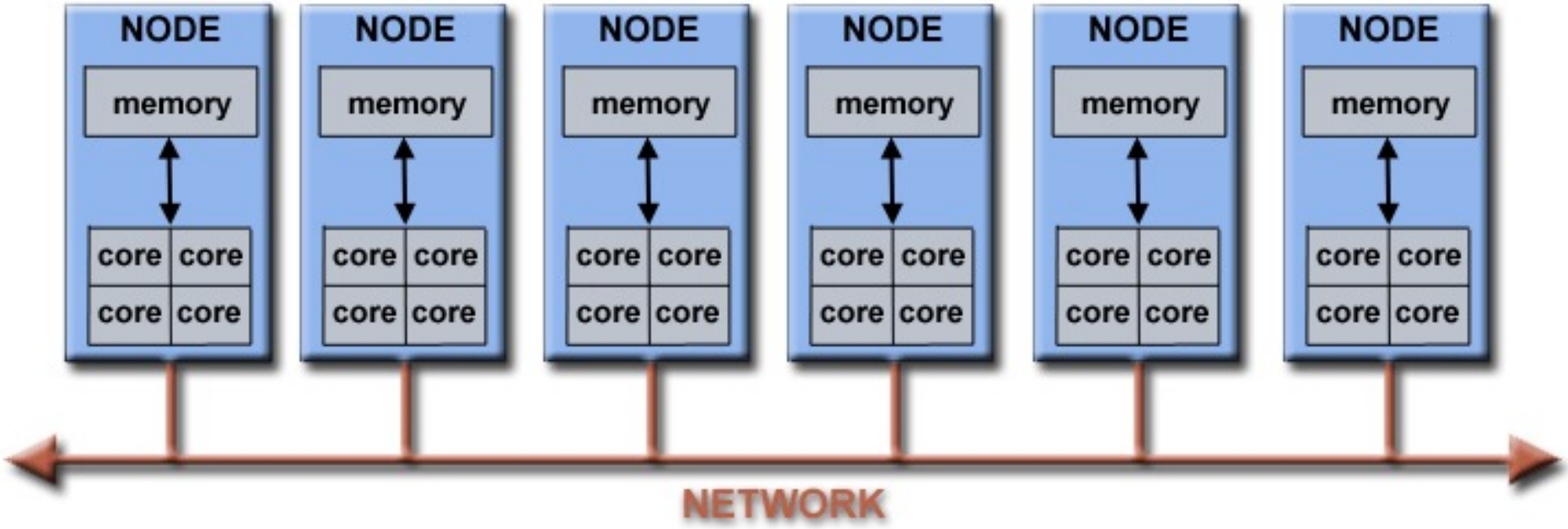
- 多个功能单元（L1 Cache、L2 Cache、Branch、Prefetch、GPU等）；
- 多个执行单元或者核心
- 多个硬件线程



IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

并行计算集群

多个单独的计算机通过网络连接起来形成计算集群



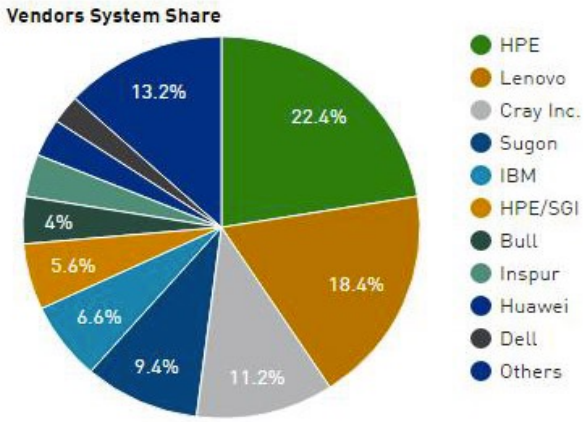
计算集群

并行计算集群

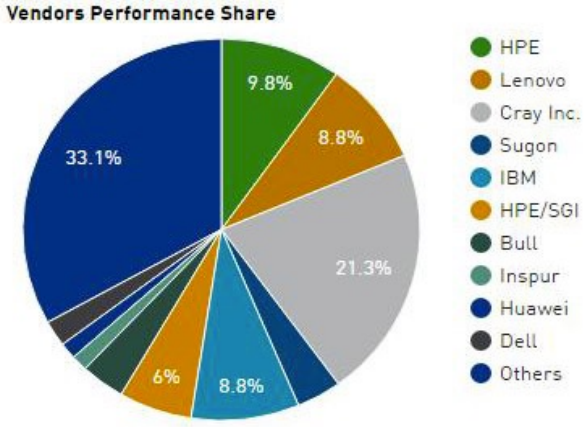
LLNL并行计算集群

- 每个节点都是一个多处理器并行机；
- 多个计算节点通过Infiniband网络连接；

并行计算集群

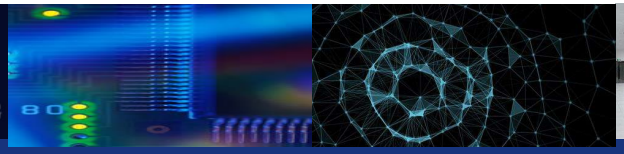


主要的超级计算机厂商



Vendors	Count	System Share (%)
HPE	112	22.4
Lenovo	92	18.4
Cray Inc.	56	11.2
Sugon	47	9.4
IBM	33	6.6
HPE/SGI	28	5.6
Bull	20	4
Inspur	18	3.6
Huawei	16	3.2
Dell	12	2.4
Fujitsu	11	2.2
Penguin Computing	9	1.8
NUDT	4	0.8
Lenovo/IBM	4	0.8
NEC	3	0.6
T-Platforms	3	0.6
Atipa	3	0.6
IBM/Lenovo	3	0.6
NRCPC	2	0.4
MEGWARE	2	0.4
RSC Group	2	0.4
Self-made	1	0.2
IPE, Nvidia, Tyan	1	0.2
DALCO	1	0.2
Adtech	1	0.2
ClusterVision	1	0.2

Source: Top500.org



❖参考书：

1、 Principles of Parallel programming By Yun Calvin Lin, Lawrence Snyder Pearson/Addison Wesley, 2008.

2、 Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edition 5, © Addison-Wesley 2012.

http://cs149.stanford.edu/fall19/