COMP4007: 并行处理和体系结构

第三章: 并行计算范式与性能评估

授课老师：王强、施少怀
助　　教：刘虎成、田超

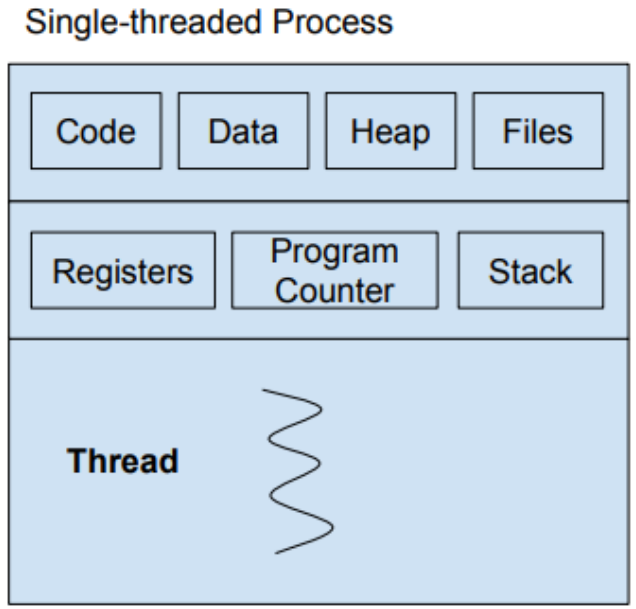
哈尔滨工业大学（深圳）

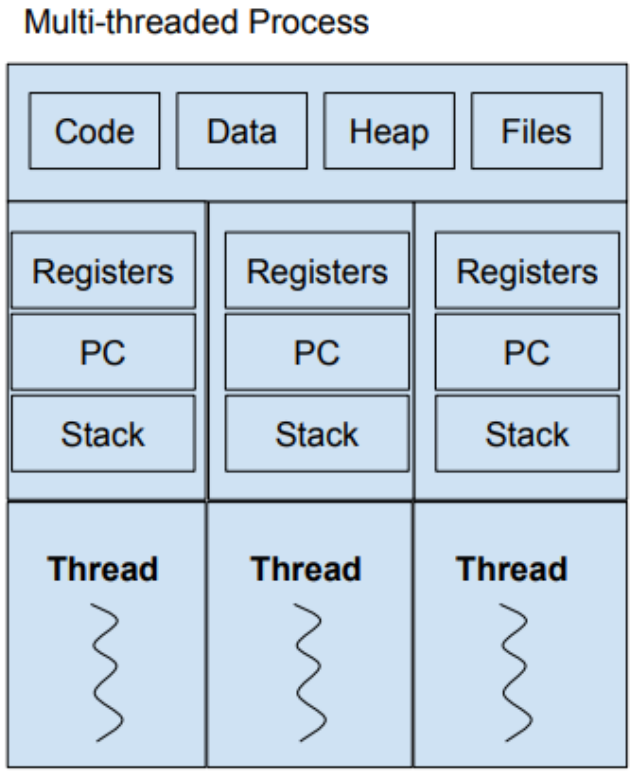
大纲

- 并行编程模型(Parallel Programming Models)
  - 共享内存模型(Shared Memory Model)
  - 分布式内存模型(Distributed Memory) / 消息传递模型(Message Passing)
  - 数据并行模型(Data Parallel)
  - 混合模型(Hybrid)
  - 单程序多数据模型(Single Program Multiple Data, SPMD)
  - 多程序多数据模型(Multiple Program Multiple Data, MPMD)
- 局部性原理(Locality)
- 并行程序设计
- 并行程序性能评估

程序(Program) vs. 进程(Process) vs. 线程(Thread)

- 程序
  - 存储在计算机中的代码
    - 磁盘(disk)
    - 或非易失性内存(non-volatile memory)
  - 可被计算机执行
- 进程
  - 执行程序的实例是**进程**，由**地址空间**和**一个或多个控制线程**组成
- 线程
  - 进程中的最小执行单位。一个进程可能只有一个线程，也可能有多个线程
  - 通常由程序计数器、寄存器和堆栈表示



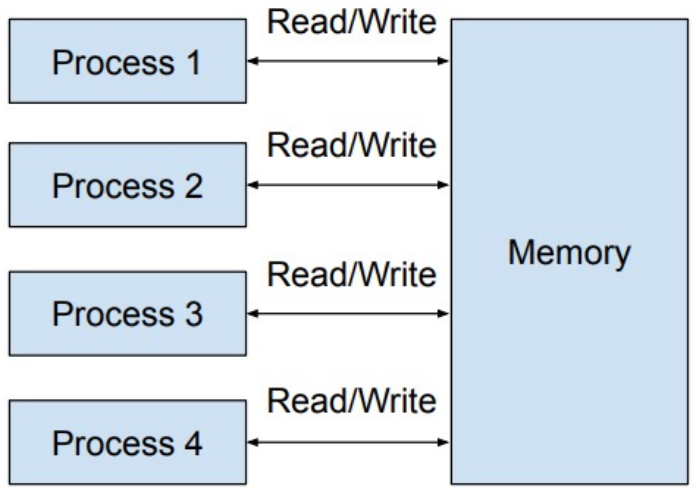


并行硬件上的并行编程模型

- 并行编程模型并不针对特定类型的机器或内存架构
- 分布式机器上的共享内存模型
  - Kendall Square Research (KSR) ALLCACHE 方法
  - 又称“虚拟共享内存”
- 共享内存机器上的分布式内存模型
  - 如SGI Origin 2000服务器上的消息传递接口(Message Passing Interface, MPI)

共享内存模型(无线程)

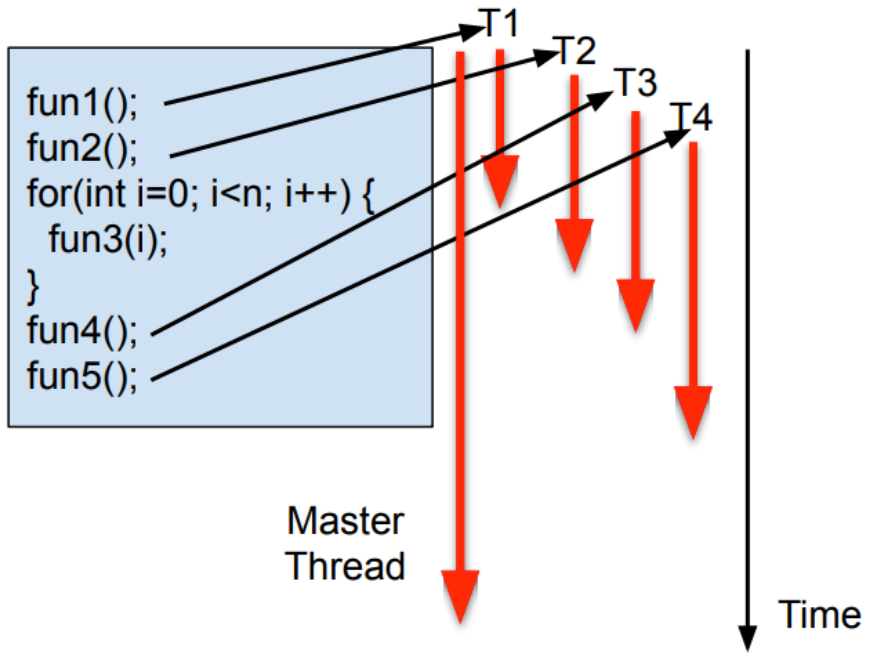
- 进程或任务共用一个地址空间
  - 异步读取或写入
  - 锁/信号量用于控制访问
    - 解决竞争
    - 防止出现竞争条件和死锁
  - 优点
    - 简单
    - 无需明确指定任务之间的通信
  - 缺点
    - 难以理解和管理数据**局部性**



- 实现方案
  - 针对独立共享内存机器
    - POSIX标准 (shm_overview: https://www.man7.org/linux/man-pages/man7/shm_overview.7.html)
    - 直接操作共享内存段(e.g., shmget, shmat, shmctl, etc).
  - 针对分布式内存机器
    - SHMEM (from Cray Research's "shared memory" library)

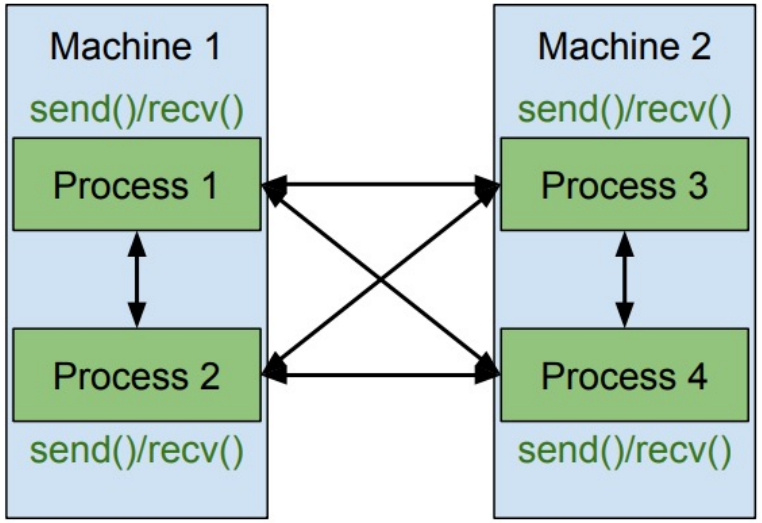
共享内存模型(线程)

- 一个“重量级(任务量大)”的**进程**可划分成可并行执行的多个 “轻量级(任务量小)”线程
  - 主线程(master thread)创建多个子线程
    - 所有线程均属于该进程
    - 所有线程共享该进程的内存地址
  - 这些线程由操作系统调度执行
  - 任何线程都可以与其他线程同时执行任何子程序
    - T1: fun1()
    - T2: fun2()
    - T3: fun4()
    - T4: fun5()
  - 线程之间通过全局内存进行通信
  - 线程可动态创建和销毁
- 实现方案
  - 通过子程序库: 如POSIX threads (pthreads)
    - 明确的并行性
    - 程序员需高度了解算法的并行细节
  - 通过编译指令: 如**OpenMP**
    - 使用简单方便
    - 可移植/跨多平台，包括Unix和Windows平台



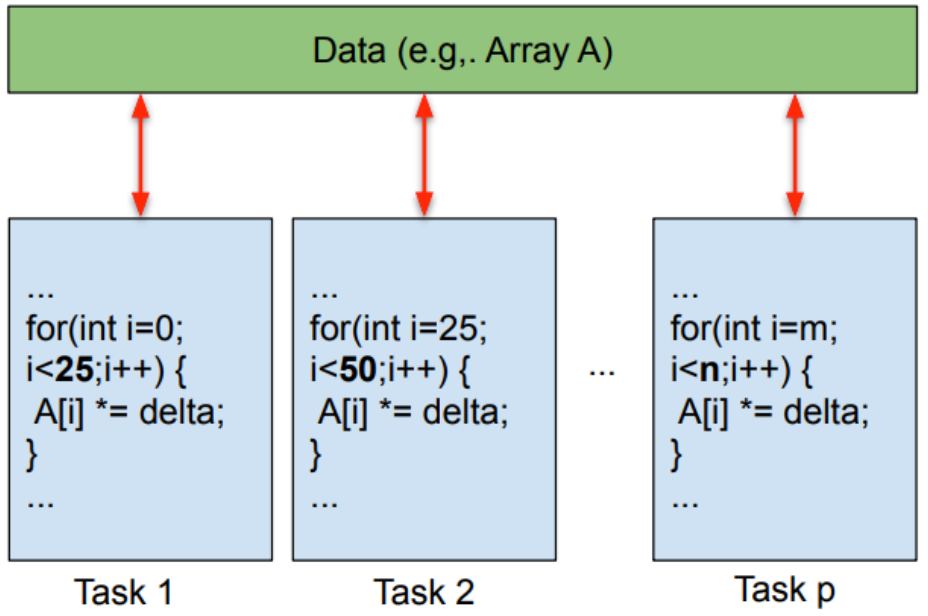
分布式内存模型/消息传递模型

- 计算过程中使用任务(进程)所在机器本地内存的一组任务(进程)
  - 该组任务可分配到同一台物理机
  - 该组任务可分配到任意数量的不同物理机
- 任务间通过收发信息进行数据交换
  - 数据传输通常需要任务间协同进行
  - 例, 发送操作必须有对应的接收操作
- 实现方案
  - TCP/UDP套接字编程
    - 应用程序开发难度大
  - 始于1994年的消息传送接口(Message Passing Interface, MPI)
    - 事实上的行业标准



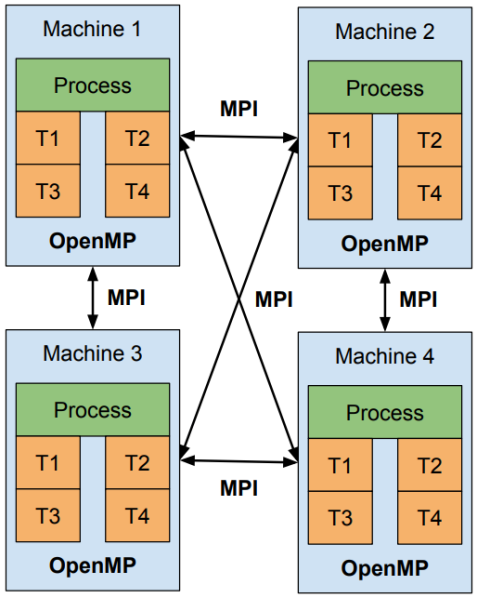
数据并行模型

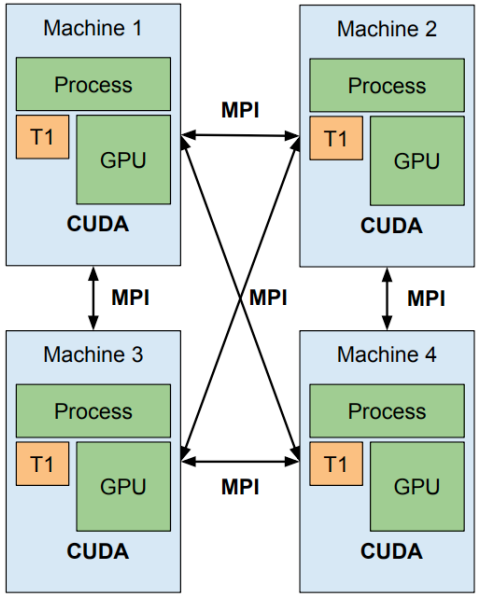
- 又称分区全局地址空间(Partitioned Global Address Space, PGAS)
- 大多数并行工作都侧重于对数据集执行操作
  - 数据集通常被组织成一种通用结构，如数组或立方体。
- 一组任务共同处理同一个数据结构
  - 每个任务处理同一个数据结构上的不同部分(partition, 分区)
  - 对分区执行相同的操作
- 数据分区
  - 在共享内存架构中，所有任务都可以通过全局内存访问数据结构
  - 在分布式内存架构中，全局数据结构可以在逻辑上和/或物理上在不同任务之间拆分。
- 实现方案
  - Coarray Fortran: Fortran 95的小型扩展集
  - Unified Parallel C (UPC): 编程语言的扩展，用于 SPMD 并行编程
  - Global Arrays: 提供共享内存式编程环境
  - X10: 基于 PGAS 的并行编程语言
  - Chapel: 开放源码并行编程语言
  - CUDA on GPUs



混合模型

- 混合模型结合了前面介绍的一种以上的编程模型
  - MPI+OpenMP
    - 线程使用节点上的本地数据执行计算密集型任务
    - 不同节点上的进程之间通过网络使用 MPI 进行通信
  - MPI+CUDA
    - 使用节点上的GPU执行计算密集型任务
    - 使用CUDA在节点本地内存与 GPU 之间交换数据
    - 使用CPU和本地内存运行MPI任务，MPI任务间通过网络相互通信
  - MPI+CUDA+OpenMP
    - 某些计算密集型任务由多个 CPU 或 CPU 内核执行
    - 部分计算密集型任务由节点上的 GPU 执行
    - 使用CPU和本地内存运行MPI任务，MPI任务间通过网络相互通信





单程序多数据模型(SPMD)

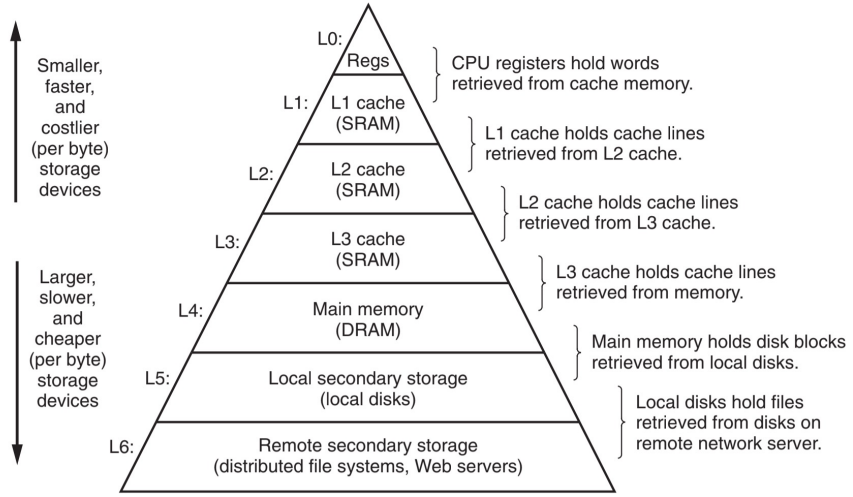
- SPMD
  - 所有任务同时执行**同一程序**的副本
  - 每个任务可能使用**不同的数据**
- SPMD是一种“高级”编程模型
  - 可以建立在前面提到的并行编程模型的任何组合上
- 与SIMD的不同
  - 是 MIMD 的一个子类别
  - SIMD 要求在不同数据上执行相同的指令，而 SPMD 可以在不同数据上执行不同的指令（如不同的子程序）。

多程序多数据模型(MPMD)

- MPMD
  - 任务可同时执行不同的程序
  - 每个任务可能使用不同的数据
- MPMD也是一种“高级”编程模型
  - 可以建立在前面提到的并行编程模型的任何组合上
- 不像 SPMD 那么常见
  - 但可能更适合某些类型的问题

局部性原理(Locality)

- 目标: 让处理器从快速存储中访问数据
  - 数据移动成本高昂，尤其是在速度较慢的内存中移动数据
- 时间局部性(Temporal locality)
  - 如果在某一时刻使用了某一内存位置的数据，那么在**不久的将来**很可能会再次使用该数据
- 空间局部性(Spatial locality)
  - 如果某个存储位置的数据在特定时间被使用，那么**附近存储位置的数据**很可能在不久的将来也会被使用
  - 缓存**以缓存行为单位**对数据元素进行缓存



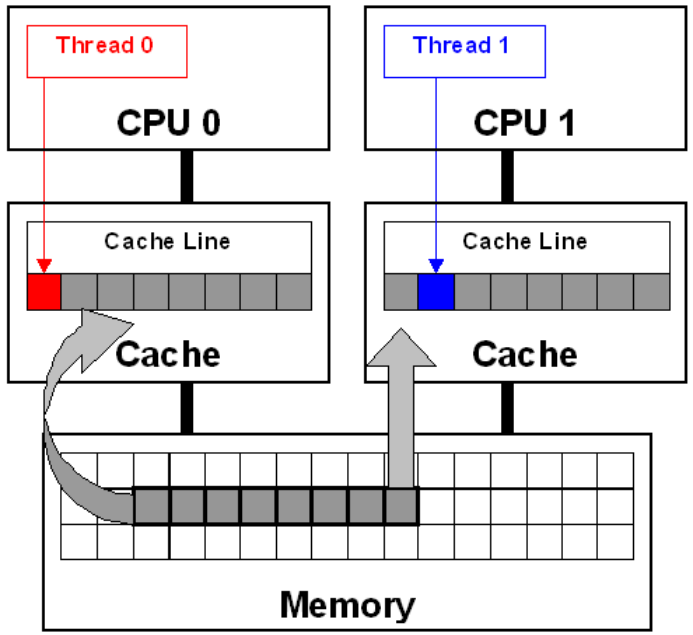
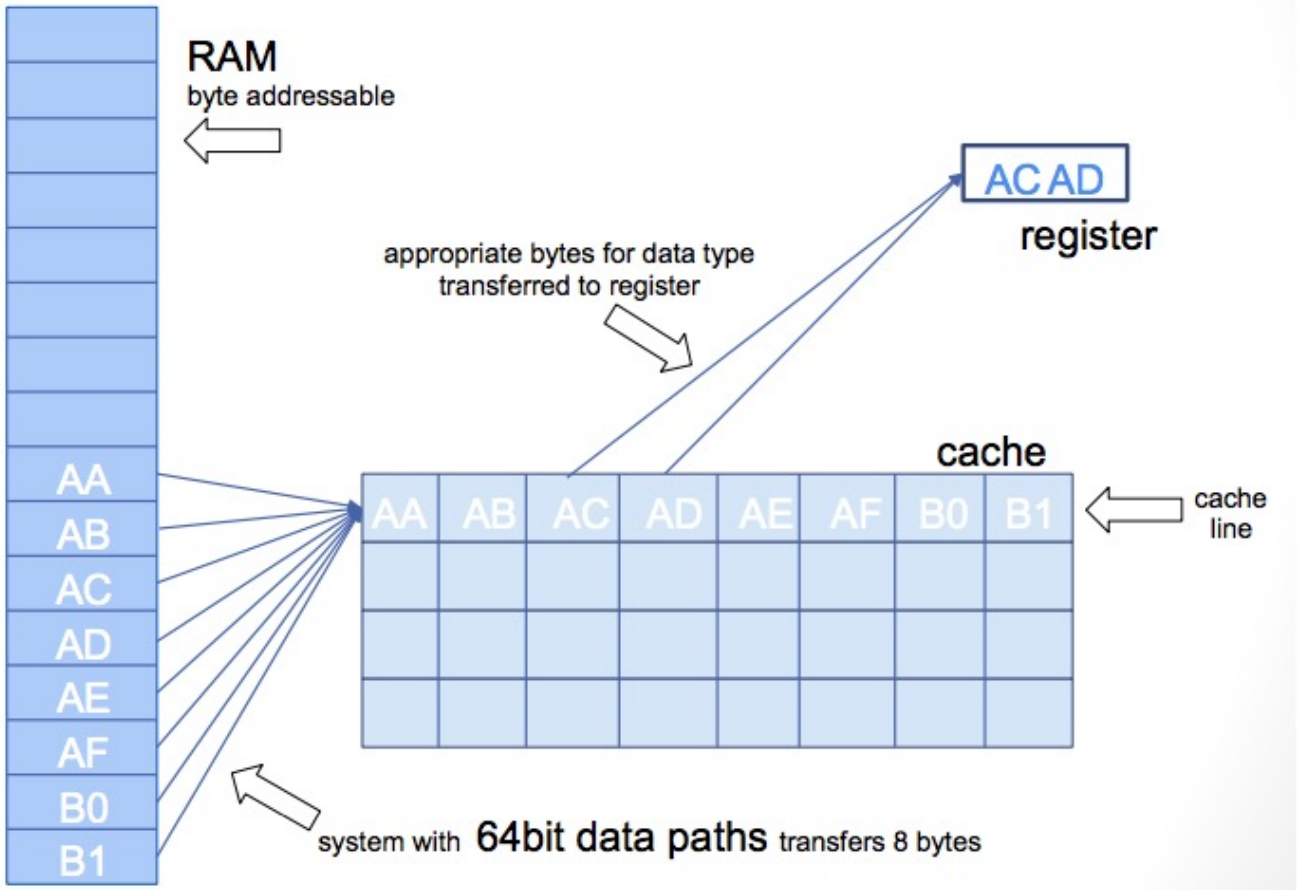


Image credit: http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

局部性原理 – 缓存行(Cache Line)

- 直接映射
- 缓存性能指标
  - **不命中率(Miss rate)**
    - 数据在缓存中找不到的比率
  - 命中时间(Hit time)
    - 读取缓存行数据到处理器寄存器的时间
    - 典型数据
      - 访问L1缓存需1个时钟周期
      - 访问L2缓存需3-8个时钟周期
  - 不命中代价(Miss penalty)
    - 因缓存不命中而需要的额外的时间
    - 访问主存通常需要 25-100 个周期



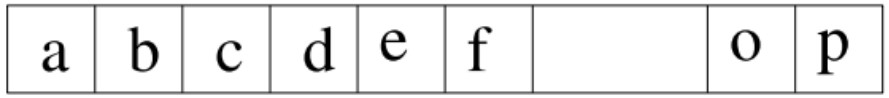
- 高速缓存整体性能
  - AMAT (Avg. Mem. Access Time) = t_hit + prob_miss * penalty_miss

Image credit: http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

C/C++数组在内存中的布局

- C/C++ 数组按行主序分配
  - 每行数据在内存中连续存储
  - 在一行中跨列进行读取
    - for(int i=0; i<N; i++) sum += A[0][i];
    - 被访问元素在内存中是连续的
    - 如果数据B的大小超过4字节, 则可利用空间局部性
  - 在一列中跨行进行读取
    - for(int i=0; i<N; i++) sum += A[i][0];
    - 被访问元素在内存中不连续
    - 没有空间局部性！

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p



Row-major order

局部性原理示例1

- 示例: 求和二维数组的所有元素
  - 假设: 一个字(word)的大小为4 字节、一个缓存行的大小为4个字

```
int sum_array_rows(int A[N][N]) {
  int i, j, sum = 0;
  for(i=0; i<N; i++) {
    for(j=0; j<N; j++) {
      sum += A[i][j];
    }
  }
}
```

```
int sum_array_cols(int A[N][N]) {
  int i, j, sum = 0;
  for(j=0; j<N; j++) {
    for(i=0; i<N; i++) {
      sum += A[i][j];
    }
  }
}
```

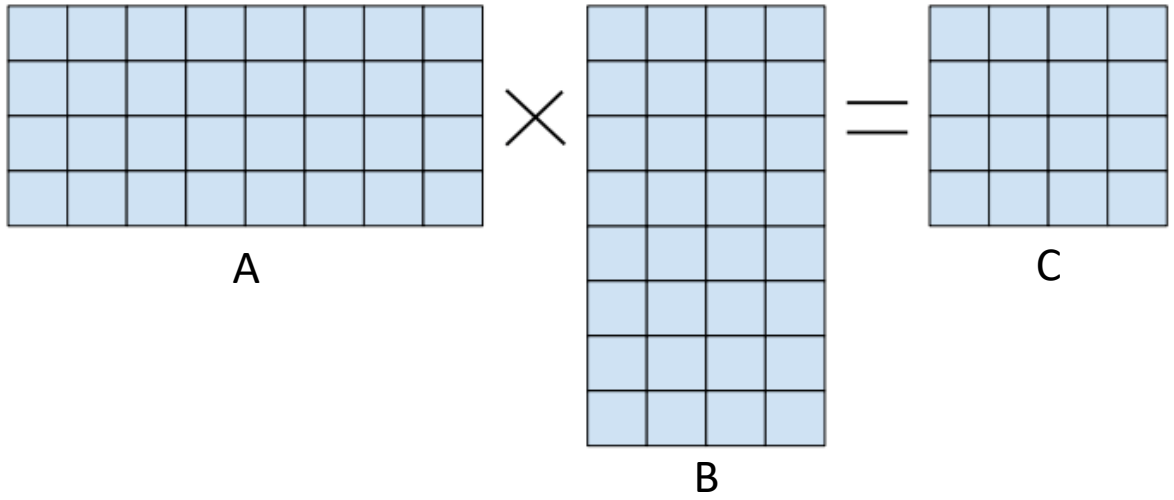
a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a	b	c	d	e	f		o	p

Row-major order

局部性原理示例2


```
const int m=4096;
const int n=4096;
const int k=4096;
float C[m][n];
float B[k][n];
float A[m][k];
```




```
for(int i=0; i<m; i++) {
  for(int j=0; j<n; j++) {
    for(int p=0; p<k; p++) {
      C[i][j] += A[i][p]*B[p][j];
    }
  }
}
```

```
for(int i=0; i<m; i++) {
  for(int p=0; p<k; p++) {
    for(int j=0; j<n; j++) {
      C[i][j] += A[i][p]*B[p][j];
    }
  }
}
```

并行程序设计

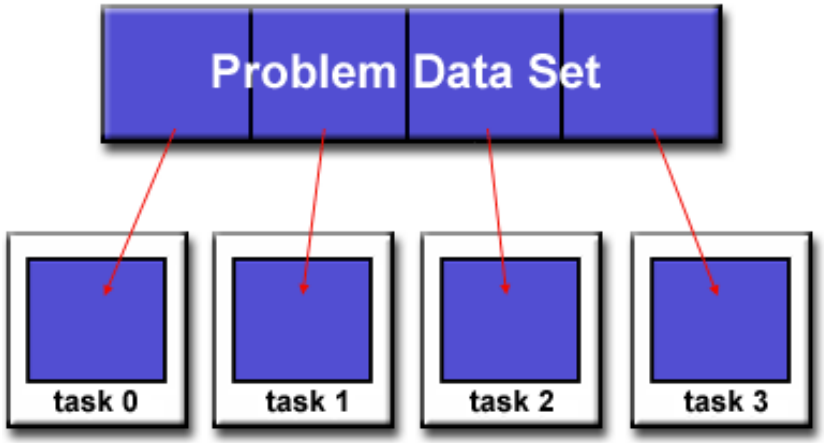
- 自动并行化 vs. 人工并行化
  - 设计和开发并行程序通常需要大量人工
  - 程序员负责确定算法的并行性和实际执行时并行性
- 人工开发并行程序
  - 耗时、复杂、易出错
- 使用编译器进行自动并行化的两种方式
  - 全自动
    - 编译器分析源代码，找出并行化的机会
    - 循环（do、for）是最常见的自动并行化目标
  - 程序员指导
    - 使用 "编译器指令 "或可能的编译器标志
- 我们需要的是人工设计和自动工具！
  - 人工设计: 了解问题和程序
  - 自动工具: 利用并行编程框架（如 OpenMP、CUDA 和 MPI）

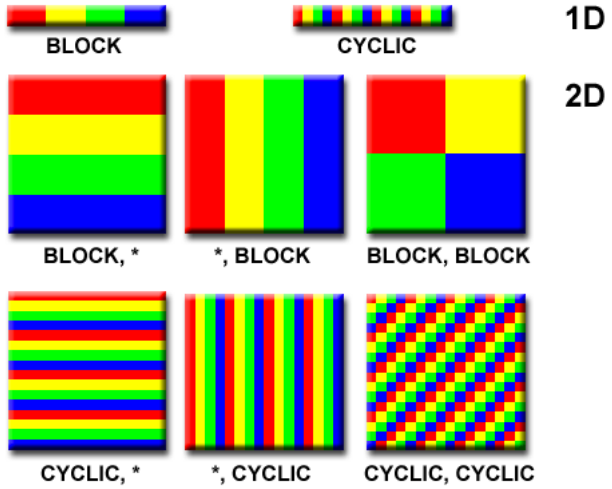
了解问题和程序

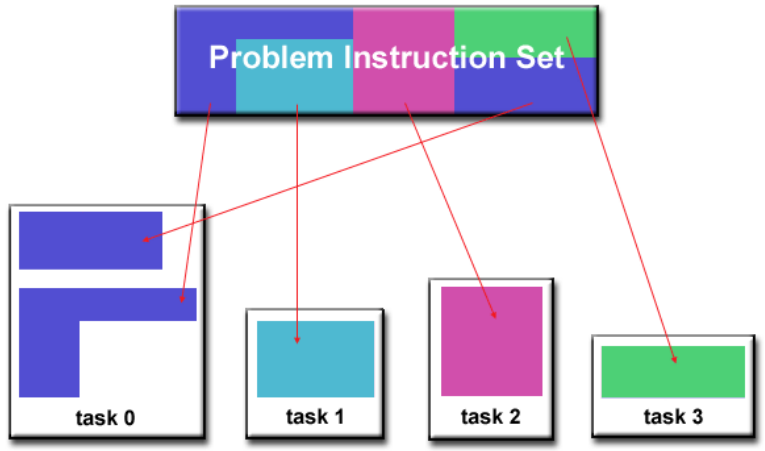
- 开发并行软件的第一步: 首先了解要并行解决的问题
  - 确定问题是否可以并行化
    - 了解现有的串行代码
  - 确定程序的热点代码
  - 找出程序中的瓶颈
  - 确定影响并行的因素
  - 如果可能，研究其他算法
  - 利用供应商提供的优化好的第三方并行软件和高度优化的数学库

分区(Partitioning)

- 将问题分解成可分配给多个任务的相对独立的"块"
- 域分解和功能分解
  - 分解：对与问题相关的数据进行分解
  - 功能分解：根据必须完成的工作对问题进行分解







了解问题 - 示例

- 示例 1: 计算数千种**独立分子构象**的势能。完成后，找出能量最小的构象
  - 每个**分子构象**都可独立确定
  - 计算**最小能量构象**也是一个可并行处理的问题

- Example 2: 利用公式: F(n) = F(n-1) + F(n-2), F(0)=0, and F(1)=1计算斐波那契数列(0,1,1,2,3,5,8,13,21,...)
  - 在计算 F(n) 值时，必须先计算 F(n-1) 和 F(n-2) 的值。
  - 递归方式很难并行！
  - 如果我们使用F(n)的解析形式呢？

并行程序设计- 通信

- 谁需要通信？
  - 任务之间的通信需求取决于需解决的问题
- 无通信问题
  - 某些类型的问题可以被分解多个并行执行且不需要共享数据的子任务
  - 这类可拆分成多个相互之间几乎不需要通信的子任务的问题通常被称为**易并行问题**
- 通信难题
  - 大多数并行应用程序并不那么简单，它们确实需要在子任务之间共享数据
- 需要考虑的因素
  - 通信开销
    - 延迟(Latency) vs. 带宽(bandwidth)
  - 通信的可见性
  - 同步(Synchronous) vs. 异步(asynchronous)

并行程序设计的要素

假设你接到一个问题，需要将一段串行代码改成并行代码，不考虑特定的体系结构和编程语言，需要考虑一下几个要素：

- 任务划分
- 通信方案
- 同步方案
- 负载平衡

三个臭皮匠，顶个诸葛亮

- 如何快速完成以下计算题：

$452\times432\times3$	同学1
$+5763\div3$	同学2
$+54\times332$	同学3
$+43\times552$	同学4
$+567\times12$	同学5

老师1负责加和

性能度量

- 性能定义为 (Effective FLOP/S) = FLOPS/Time

- 在任务J上，处理器X的性能是处理器Y的n倍

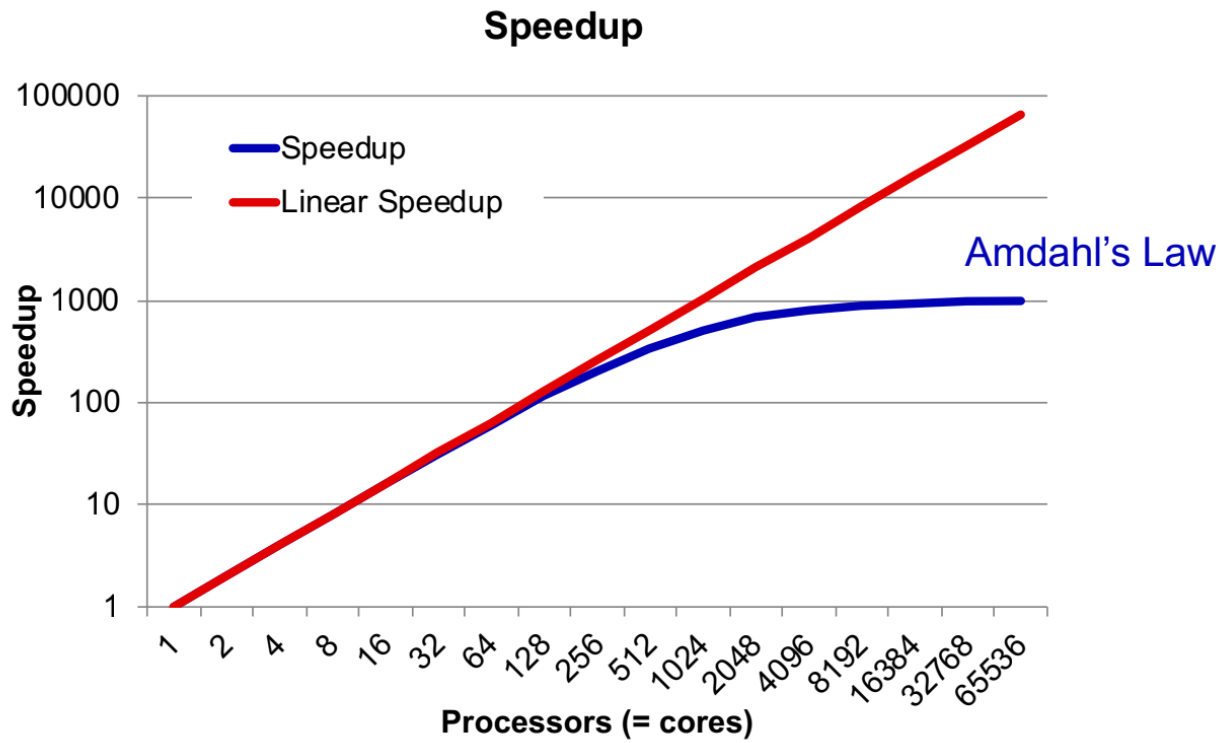
$$\frac{P_X}{P_Y} = \frac{T_Y}{T_X} = n$$

其中P一般指性能，T一般指时间

- 例如，它们任务J上的运行时间分别是：
  - 处理器 X ->10s，处理器 Y -> 15s
  - $\frac{T_Y}{T_X} = \frac{15}{10} = 1.5$

Amdahl定律

- Amdahl定律：
  - ✓ 计算任务包含可以进行并行计算的部分（T_p）以及不可并行的部分（T_s）；
  - ✓ 理论加速比受 T_s 限制。
- 对于一个串行程序：
  - ✓ $T_{serial} = T_p + T_s = (1 - s) + s$
  - ✓ s称为Amdahl片段
- 假设有p个处理器并行
  - ✓ $T_{parallel} \geq \frac{T_p}{p} + T_s = \frac{1-s}{p} + s$
- 加速比
  - ✓ $\frac{T_{serial}}{T_{parallel}} \leq \frac{1}{\frac{1-s}{p}+s} \leq 1/s$



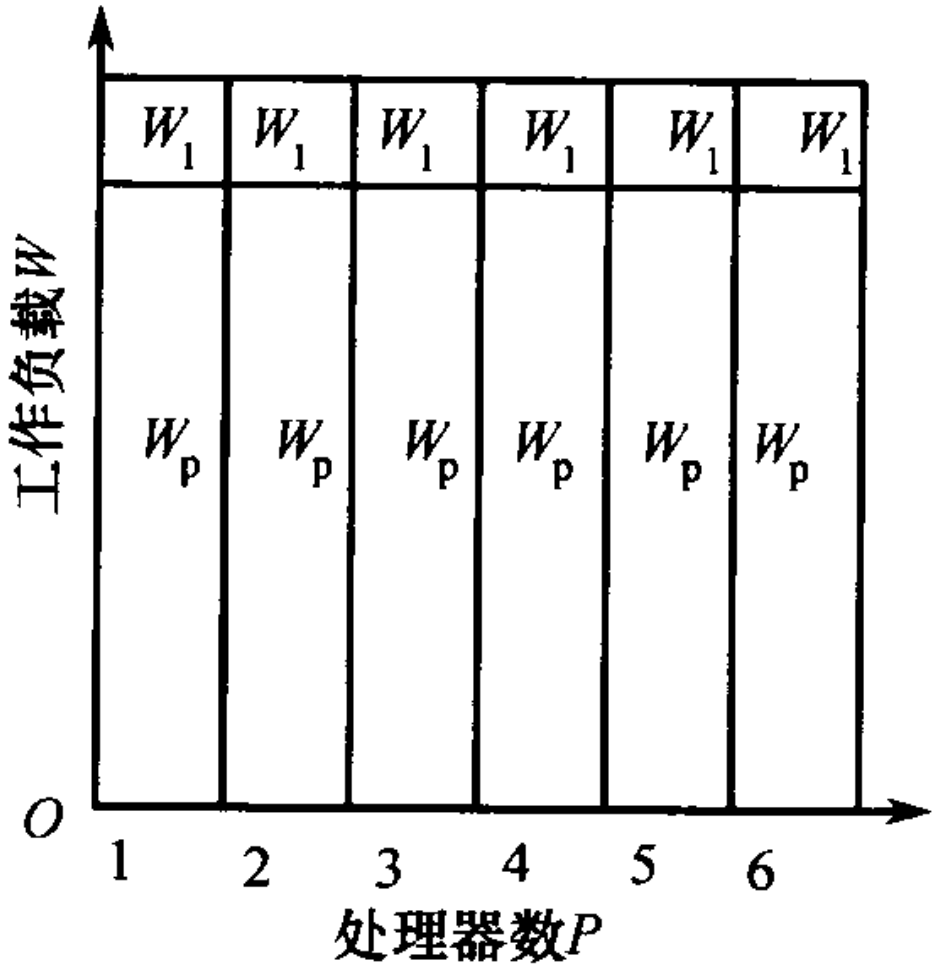
Amdahl定律

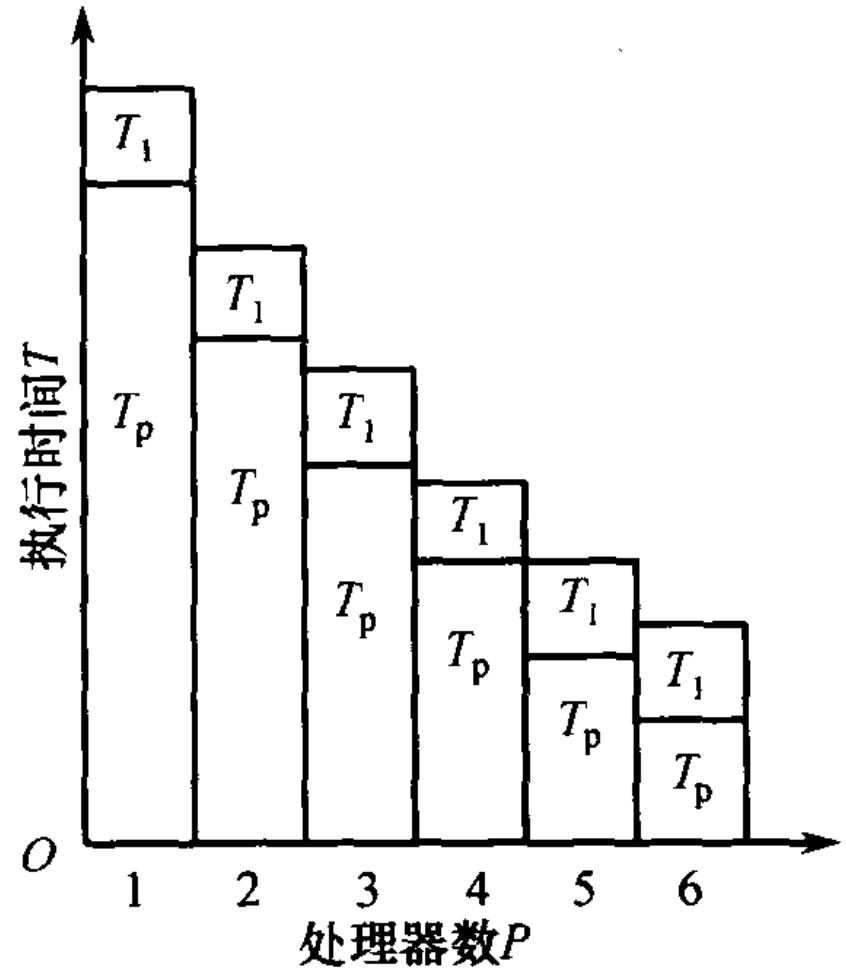
Amdahl定律(1967)：

$$\text{系统加速比} = \frac{1}{(1-\text{可加速部分比例}) + \frac{\text{可加速部分比例}}{\text{理论加速比}}}$$

并行处理面临着**两个重要的挑战**

- ✓ 程序中的并行性有限
- ✓ 相对较大的通信开销


工作负载W
W_1 W_1 W_1 W_1 W_1 W_1
W_p W_p W_p W_p W_p W_p
O
1 2 3 4 5 6
处理器数P


执行时间T
T_1 T_1 T_1 T_1 T_1 T_1
T_p T_p T_p T_p T_p T_p
O
1 2 3 4 5 6
处理器数P

练习

假设想用100个处理器达到80的加速比，求原计算程序中串行部分最多可占多大的比例？

根据**Amdahl**定律：

$$\text{加速比}=\frac{1}{\frac{\text{可加速部分比例}}{\text{理论加速比}}+(1-\text{可加速部分比例})}$$

$$80=\frac{1}{\frac{\text{并行比例}}{100}+(1-\text{并行比例})}$$

由上式可得：并行比例＝0.9975

Gustafson定律

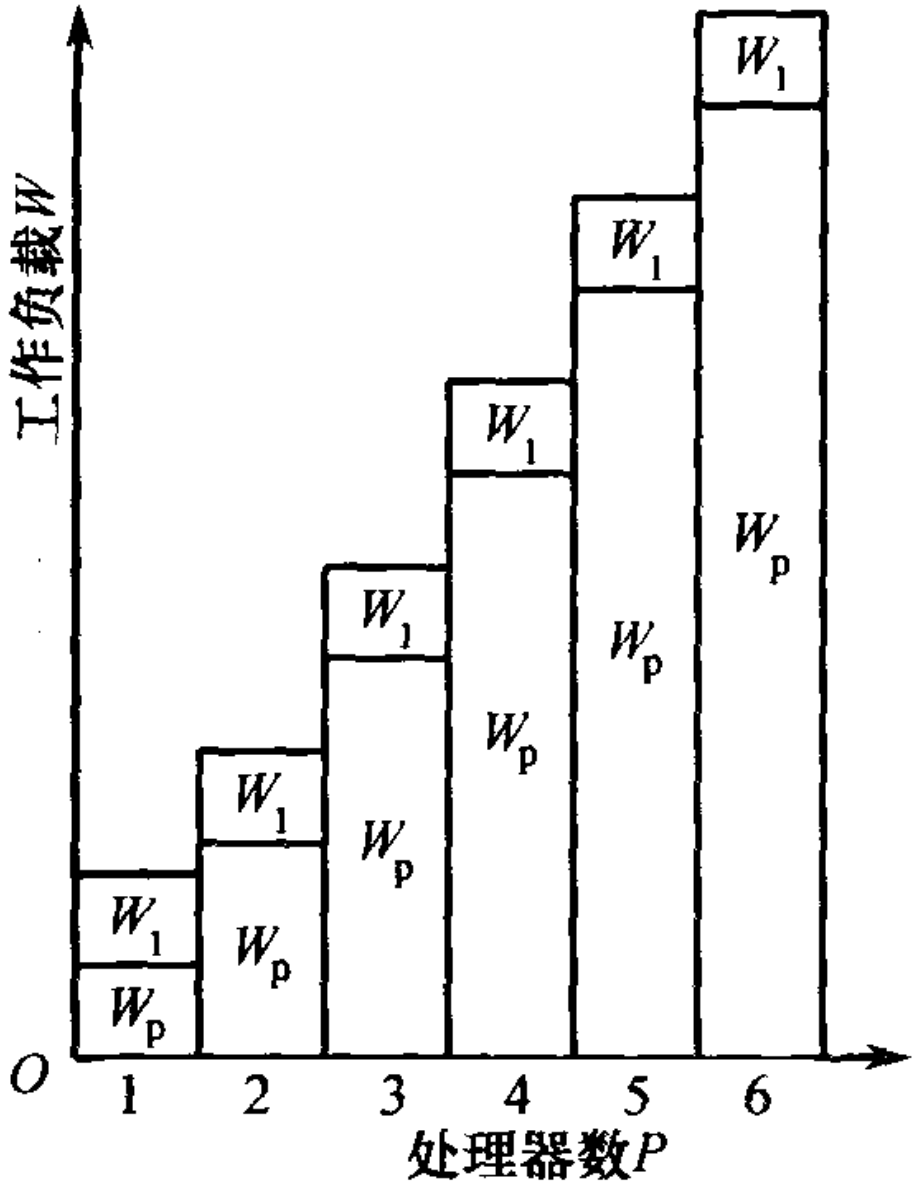
- John Gustafson (1988年) 提出了固定时间的概念。
  - ✓ Amdahl定律的局限：假设程序的计算任务是固定的。
  - ✓ 在增加计算量的情况下，为了维持相同的时间，并行进程数应如何变化？
  - ✓ **举例：**为了提高精度，加大计算量，需加多处理器数，才能维持时间不变。

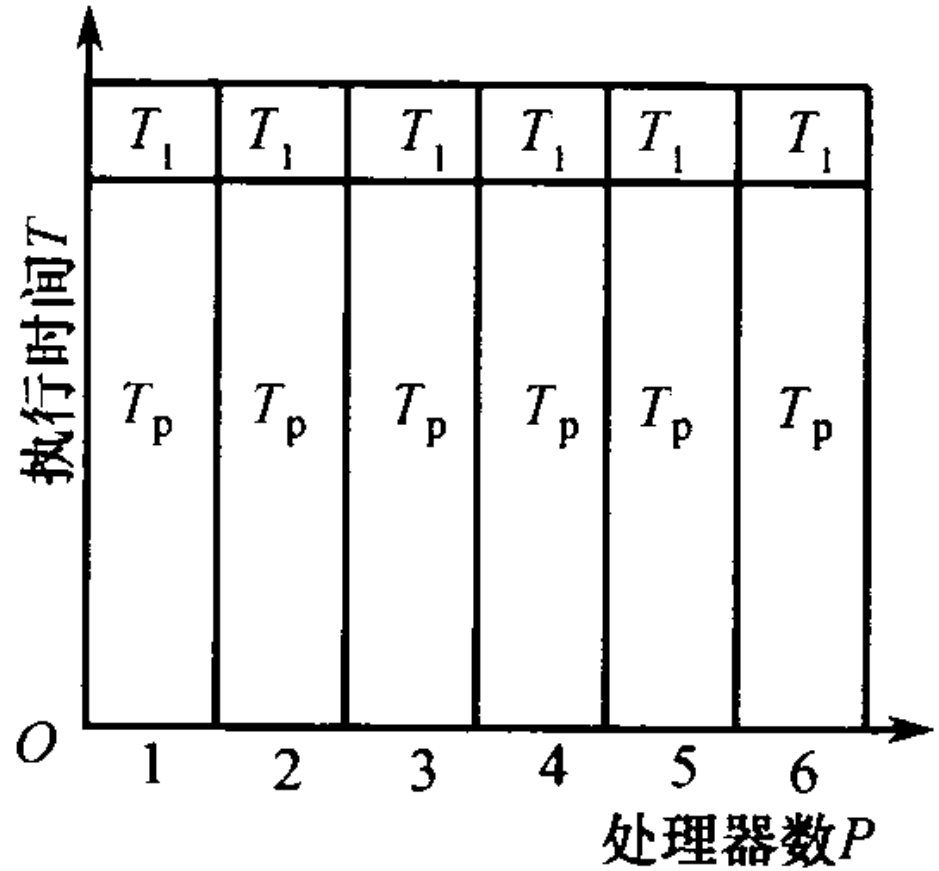
- 在实际应用中，没有必要固定工作负载，而计算程序运行在不同数目的处理器上，增多处理器必须相应地增大问题规模，才有实际意义。

$$S_n = \frac{\text{执行扩大工作负载的顺序执行时间}}{\text{执行扩大工作负载的并行执行时间}} = \frac{sW + (1-s)nW}{W}$$

$$S_n = s + (1-s)n$$

s：固定负载的比例/不可并行的比例


工作负载W
O
1
2
3
4
5
6
处理器数P
W1
Wp


执行时间T
O
1
2
3
4
5
6
处理器数P
T1
Tp

Sun-Ni定理

- Xian-He Sun(孙贤和)和Lionel Ni于1993年提出
  - ✓ 提出了存储受限的加速定律。
  - ✓ 将Amdahl定律和Gustafson定律一般化

基本思想：只要存储空间许可，应尽量增大问题规模，以产生更好和更精确的解(此时可能使执行时间略有增加)。

Sun-Ni定理

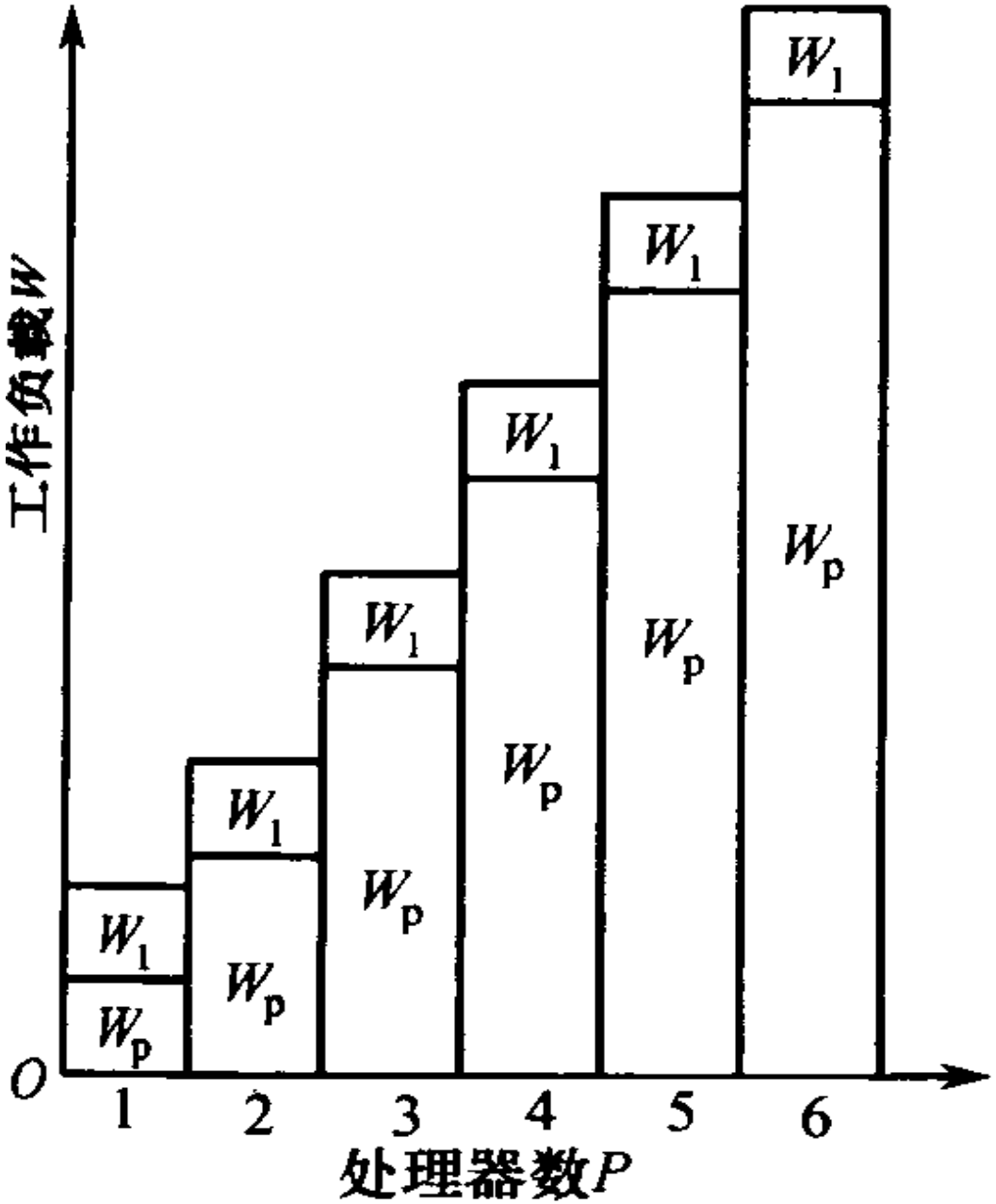
- 在n个节点的并行系统上，能够求解较大规模的问题是因为存储容量可增加nM。
- 令因子$G(n)$反应存储容量增加到P倍时工作负载的增加量，扩大后的工作负载:

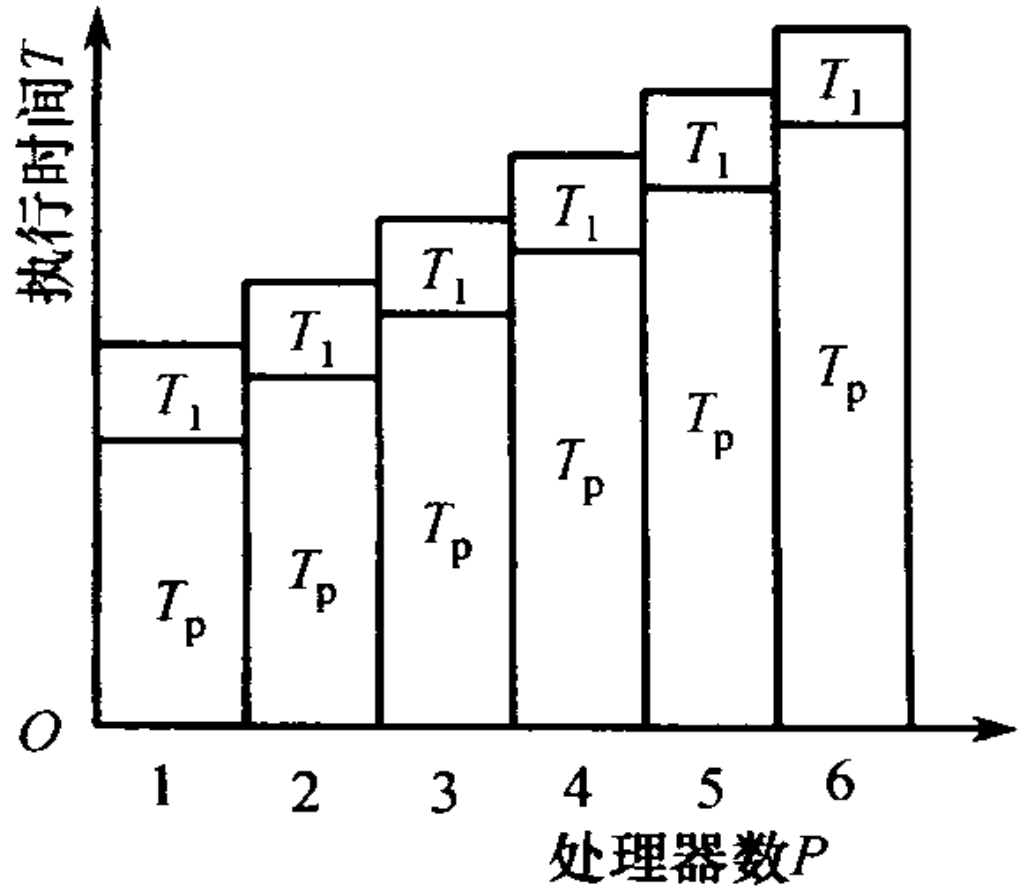
$$W_{serial} = sW + (1-s)G(n)W$$

$$W_{parallel} = sW + \frac{(1-s)G(n)W}{n}$$

$$S_n = \frac{sW + (1-s)G(n)W}{sW + \frac{(1-s)G(n)W}{n}}$$

$$S_n = \frac{s + (1-s)G(n)}{s + \frac{(1-s)G(n)}{n}}$$


工作负载W
O
1
2
3
4
5
6
处理器数P
W_1
W_p


执行时间T
O
1
2
3
4
5
6
处理器数P
T_1
T_p

三大定理之间的关系

Sun-Ni定理

$$S_n = \frac{s + (1-s)G(n)}{s + \frac{(1-s)G(n)}{n}}$$

$G(n) = 1$

$G(n) = n$

Amdahl定理

$$S_n = \frac{1}{s + \frac{1-s}{n}}$$

Gustafson定理

$$S_n = s + (1-s)n$$

并行扩展性分析

➢ Amdahl定律-强扩展性

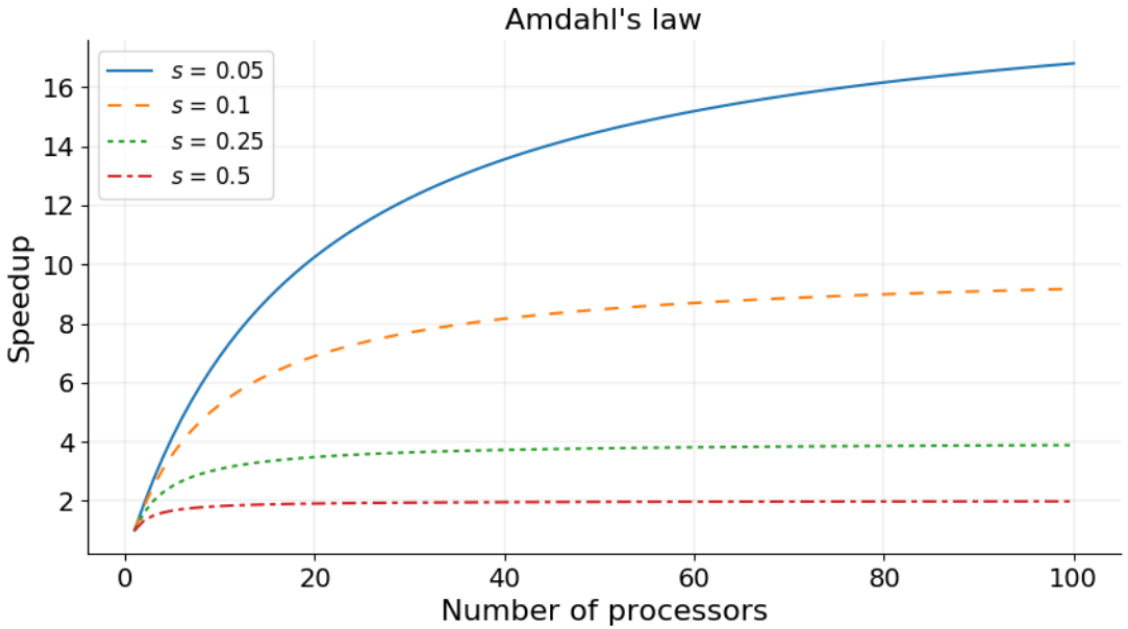
假设负载大小是W，处理器核数是p，则每个核分到的负载是 W/p 。

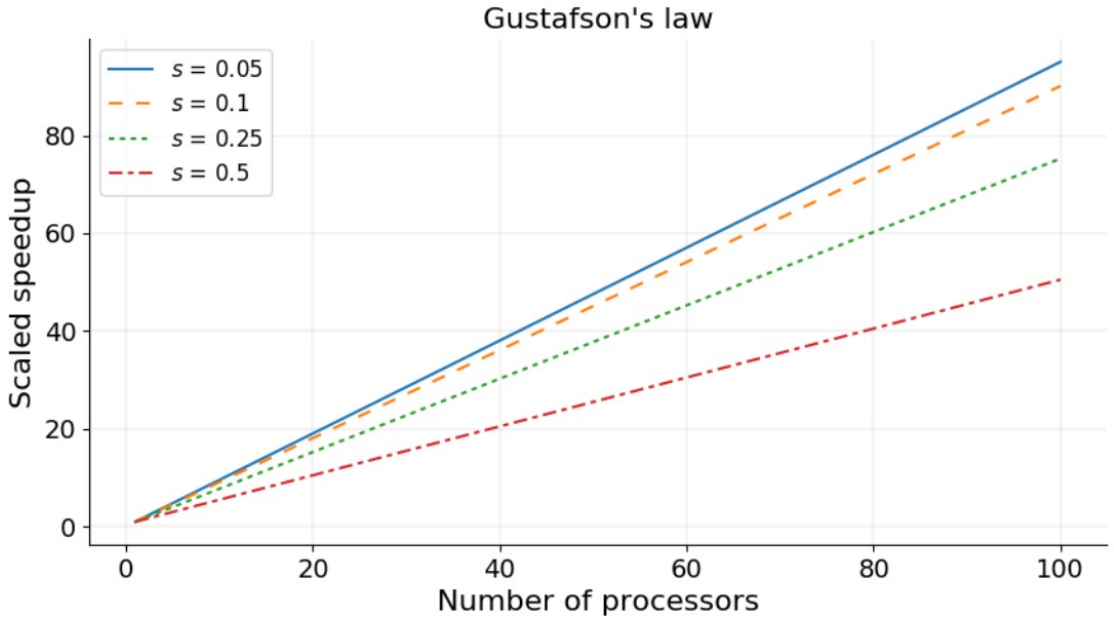
$$speedup \leq 1/s$$

➢ Gustafson定律-弱扩展性

假设每个核处理的负载大小是w，处理器核数是p，总负载是$w \times p$。

$$speedup \leq s + (1-s) \times p$$





s：串行代码比例

https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

扩展性分析

- 假设我们采用一个 p 核处理器来加速一个应用程序，串行实现的时间开销是 T_s，并行实现的时间开销是T_p，那么加速比是：

$$S = T_s/T_p$$

- 并行效率的定义：$E = S/p$，衡量了并行计算的资源利用效率。

- 例题：假设在一个4核CPU的处理器上对程序进行加速，程序从串行实现的1小时缩减到20分钟，请问该并行实现的效率是多少？

$$E = \frac{(\frac{60}{20})}{4} = 75\%$$

小结与扩展：提升并行效率

- 三大并行性能定律
  - ✓ 并行性不足：减少串行代码比例、增多处理器
  - ✓ 三者的转换关系
- 在并行处理中，影响扩展性的关键因素
  - ✓ 数据的分配；
  - ✓ 并行算法的结构；
  - ✓ 在空间和时间上对数据的访问模式。
- 并行程序的计算 / 通信比率
  - ✓ 反映并行程序性能的一个重要的度量；
  - ✓ 随着处理数据规模的增大而增加；随着处理器数目的增加而减少。

总结

- 不同类型的并行编程模型
  - Shared Memory Model共享内存模型(Shared Memory Model)
  - 分布式内存模型(Distributed Memory) / 消息传递模型(Message Passing)
  - 数据并行模型(Data Parallel)
  - 混合模型(Hybrid)
  - 单程序多数据模型(Single Program Multiple Data, SPMD)
  - 多程序多数据模型(Multiple Program Multiple Data, MPMD)
- Locality局部性原理(Locality)
  - 内存层次结构
  - 缓存行
- 并行程序性能评估
  - 三大定律
  - 并行扩展性

阅读清单

- Pacheco, Peter. An introduction to parallel programming. Elsevier, 2011. **Chapter 2.4-2.8.** [PDF: http://www.e-tahtam.com/~turgaybilgin/2013-2014-guz/ParalelProgramlama/ParallelProg.pdf]