

3 并行编程模型

- 简介


- 计算 π

- threads

```
void* thread_sum(void* rank) {
    int my_rank = (int) rank;
    double my_sum = 0.0;
    // domain decomposition
    for (int i = my_rank; i < n; i += thread_count) {
        double x = (i + 0.5) / n;
        my_sum += 1.0 / (1.0 + x * x);
    }
    local_sum[my_rank] = my_sum;
    return NULL;
}

// global variables
int thread_count;
int n;
double* local_sum;

// multithreaded version
double compute_pi () {
    thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
    local_sum = (double*) malloc (thread_count*sizeof(double));
    // parallel part
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, thread_sum, (void*)thread);
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
    // sequential part
    double sum = 0.0;
    for (thread = 0; thread < thread_count; thread++)
        sum += local_sum[thread];
    double pi = 4.0 * sum / n;
    return pi;
}
```



- openMP

```
double compute_pi(int n) {
    int i;
    double sum = 0.0;
    #pragma omp parallel for reduction(+: sum) schedule(static)
    for (i = 0; i < n; ++i) {
        double x = (i + 0.5) / n;
        sum += 1.0 / (1.0 + x * x);
    }
    double pi = 4.0 * sum / n;
    return pi;
}
```

$$= 4.0 / (1 + x^2)$$

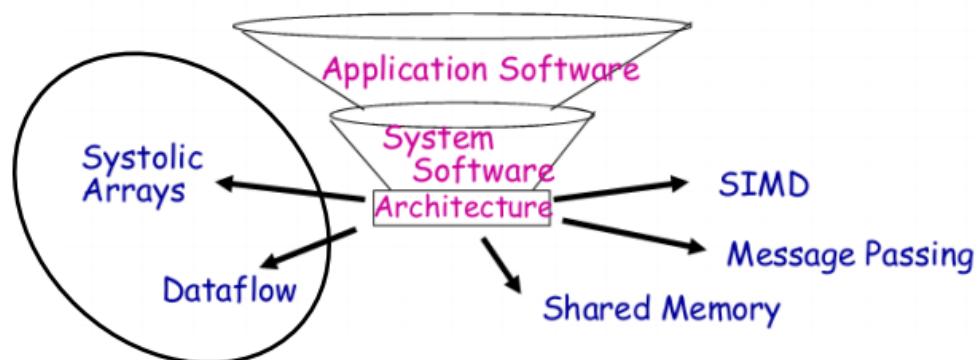
- MPI

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] ) {
    int n, my_rank, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    while (1) {
        if (my_rank == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if (n == 0) break;
        else {
            sum = 0.0;
            for (i = my_rank; i < n; i += numprocs) {
                x = (i + 0.5) / n;
                sum += 4.0 / (1.0 + x*x);
            }
            mypi = sum / n;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
            if (my_rank == 0) printf("pi is approximately %.16f\n", pi);
        }
    }
    MPI_Finalize();
    return 0;
}
```

- 不同的并行粒度：Opencl线程级的
- 不同的并行实现
 - 隐式：OpenMP, OpenCL
 - 显式：pthreads, MPI
- 不同代码规模
- 不同的硬件架构——OpenCL需要专门的硬件设备
- 历史

- 1970s – early 1990s, 并行机由它的并行模型和语言唯一决定。
- Historically (1970s – early 1990s), each parallel machine was unique, along with its programming model and language.
 - 架构 (Architecture) = 编程模型 (prog. model) + 通信抽象 (comm. abstraction) + 机器 (machine) ;
 - 并行架构依附于编程模型 (parallel architectures tied to programming models) ;



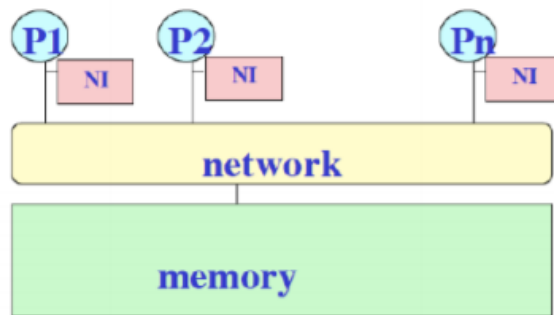
Uncertainty of direction paralyzed parallel software development!

- 收缩阵列 (Systolic Array)
 - 专用计算机的种类:
 - 不灵活且高度专用的结构
 - 结构, 实现一些可编程性和重新配置。
 - 数据将以本地状态确定的定期“心跳”通过系统。
- 如今
 - 现在我们将编程模型与底层并行机架构分开
 - 主导: 共享地址空间、消息传递、数据并行
 - 其他: 数据流、脉动阵列
 - “计算机体系结构”的扩展以支持通信和合作
 - OLD: 指令集架构; 新: 通信架构
 - 定义
 - 关键抽象、边界和原语 (接口)
 - 实现接口 (硬件或软件) 的组织结构
 - 编译器、库和操作系统是重要的桥梁
- 编程模型
 - 程序员在编码应用程序中使用什么
 - **指定通信和同步**
 - 多道程序: 没有通信或同步——独立
 - 共享地址空间: 像公告板
 - 消息传递: 如信件或电话, 明确的点对点

- 数据并行：对数据进行更严格的全局操作，使用共享地址空间或消息传递实现
- 冯诺依曼模型
 - 执行指令流（机器代码）
 - 指令可以指定：算术运算、数据地址、下一条要执行的指令
 - 复杂度高
 - 跟踪数十亿个数据位置和数百万条指令
 - 管理：模块化设计、高级编程语言（同构）
- 并行编程模型
 - 什么是并行编程模型（感觉 ppt 上没有明确定义）
 - 程序员在编码应用程序中所使用的（What programmer uses in coding applications）
 - 具体化的通信和同步机制（Specifies communication and synchronization）
 - 并行编程模型是作为对硬件和内存架构的抽象而存在的。
 - 消息传递
 - 有本地数据的独立任务
 - 任务通过交换消息进行通信
 - 共享内存
 - 任务共享一个公共地址空间
 - 任务通过异步读写这个空间进行交互
 - 数据并行化
 - 任务执行一系列独立的操作
 - 数据通常均匀分布在任务之间
 - 也被称为“尴尬的平行”
- 架构模型的演变
 - 从历史上看，为编程模型量身定制的机器
 - 编程模型、通信抽象和机器组织统称为“架构”
 - 进化有助于理解收敛：确定core概念
 - 最常见的模型：共享内存模型、线程模型、分布式内存模型、GPGPU编程模型、数据密集型计算模型
 - 其他模型：数据流，Systolic arrays
 - 检查编程模型、动机、预期应用和对融合的贡献
- 并行编程模型的几个部分
 - 控制：并行性是如何产生的？操作应该按什么顺序进行？不同的控制线程如何同步？
 - 变量名称：哪些数据是私有的还是共享的？如何访问共享数据？

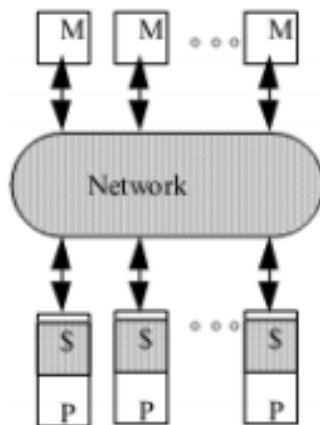
- 操作：哪些操作是原子的？
 - 开销：我们如何计算运营成本？
- 共享内存模型
 - 任何处理器都可以直接引用任何内存位置：由于加载和存储，通信隐式地发生
 - 便捷性
 - 位置透明度
 - 类似于单处理器分时的编程模型
 - 除了在不同处理器上运行的进程
 - 多道程序工作负载的良好吞吐量
 - 俗称共享内存机器或模型
 - 不明确：内存可能物理分布在处理器之间
 - 进程：虚拟地址空间上有一个或多个控制线程
 - 进程的部分地址空间是共享的
 - 写入对其他线程、进程可见的共享地址
 - 单处理器模型的自然扩展：用于通信的常规内存操作；用于同步的特殊原子操作
 - 线程通过读/写共享变量进行通信
 - 同步原语也是共享变量
 - 在这种编程模型中，任务共享一个公共地址空间，它们异步读写
 - 可以使用锁/信号量等各种机制来控制对共享内存的访问
 - 共享内存的优缺点
 - 从程序员的角度来看，该模型的一个优点是缺少数据“所有权”的概念，因此无需明确指定任务之间的数据通信，程序开发通常可以简化
 - 性能方面的一个重要缺点是更难以理解和管理数据局部性
 - 将数据保持在处理其上的处理器的本地可以节省内存访问、缓存刷新和多个处理器使用相同数据时发生的总线流量，不幸的是，控制数据局部性很难理解并且超出了普通用户的控制范围
 - **共享内存编程模型实现**
 - 本机编译器和/或硬件将用户程序变量转换为全局的实际内存地址（在独立的 SMP 机器上，这很简单）
 - 在分布式共享内存机器上，例如 SGI Origin，内存在物理上分布在机器网络中，但通过专门的硬件和软件实现全局化
 - 共享内存编程模型实现的典型架构举例
 - SAS Machine Architecture
 - 其中的典型架构：对称多处理器 One representative architecture: SMP
 - 下面是实现原理
 - Used to mean Symmetric MultiProcessor:

- All CPUs had equal capabilities in every area, e.g., in terms of I/O as well as memory access
- Evolved(发展) to mean Shared Memory Processor:
- Non-message-passing machines (included crossbar as well as bus based systems 系统总线)
- Now it tends to refer to bus-based shared memory machines:
- Small scale: < 32 processors typically
-

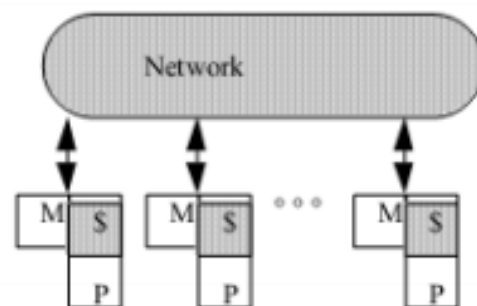


29

- 上面的架构在规模一大会出现问题（左图是有问题的初始架构，右边是改进的）
-



"Dance hall"



Distributed memory

- 左边的问题：
 - 互连的总线等成本问题（Problem is interconnect）：
 - cost (crossbar) or bandwidth (bus)
 - 带宽不会提升（Dance-hall: bandwidth is not scalable, but lower cost than crossbar）
 - Latencies to memory uniform, but uniformly large
- 改进成右边：
 - Distributed memory or non-uniform memory access (NUMA)：
 - Construct shared address space out of simple message transactions across a general-purpose network (e.g. read-request, read-response)。

- 线程模型

- 这种编程模型是一种共享内存编程
- 在并行编程的线程模型中，单个进程可以有多个并发执行路径
- 也许可以用来描述线程的最简单的类比是包含许多子例程的单个程序的概念
- 程序是控制线程的集合：在某些语言中，可以动态创建
- 每个线程都有一组私有变量，例如，本地栈变量
- 还有一组共享变量，例如静态变量、共享公共块或全局堆
 - 线程通过写入和读取共享变量进行隐式通信
 - 线程通过在共享变量上同步来协调

- **造成并行编程模型不能达到理想加速比的原因？**

- 利用 Amdahl's Law 定律，分析如下：有的部分不能并行
- 还有其他的问题：
 - 线程创建需要开销。
 - 数据划分会出现不均衡（负载不均衡）。
 - 共享数据会出现 竞争条件问题、死锁问题：加锁、解锁影响性能。

- 解耦

- 数据解耦
 - 将整个数据集分解成更小的离散部分，然后并行处理它们
- 任务解耦
 - 基于自然的独立子任务集划分整个任务
- 注意事项
 - 导致更少或没有共享数据
 - 避免子任务之间的依赖，否则会变成流水线

- **任务和线程**

- 任务包含数据和它的进程，任务被调度到线程上进行执行
- 任务比线程更轻量级
- 线程之间可以通过任务的 steal 达到负载均衡的目的
- 任务适合多种数据结构：适合列表、树、地图数据结构
- 注意事项
 - 任务比线程多得多
 - 更灵活地安排任务、轻松平衡工作量
 - 任务中的计算量必须足够大，以抵消管理任务和线程的开销
 - 静态调度
 - 任务是单独的、独立的函数调用的集合，或者是循环迭代
 - 动态调度
 - 任务执行长度可变且不可预测

- 可能需要一个额外的线程来管理一个共享结构来保存所有任务
- **线程之间竞争条件**
 - 线程之间竞争资源：假定执行顺序，但不能保证
 - 存储冲突最常见：多个线程并发访问同一内存位置，至少有一个线程正在写入
 - **确定性竞争和数据竞争**（有锁一定没有数据竞争，但确定性竞争都可以）
 - 当两个并行链访问相同的内存位置并且至少一个链执行写入操作时，就会发生确定性竞争。程序结果取决于哪个链“赢得比赛”并首先访问内存。
 - 数据竞争是确定性竞争的一种特殊情况。数据竞争是一种竞争条件，当两个并行链（没有共同的锁）访问相同的内存位置并且至少一个链执行写操作时发生。程序结果取决于哪个链“赢得比赛”并首先访问内存。
 - 如果并行访问受锁保护，则不存在数据竞争。但是，使用锁的程序可能不会产生确定的结果。锁可以通过在更新期间保护数据结构在中间状态不可见来确保一致性，但不能保证确定性结果
 - 可能不会在任何时候都很明显
 - **解决**
 - 设置临界区
 - 控制临界区的共享访问：互斥与同步、critical session、原子操作
 - 将变量作用域变为线程本地
 - 拥有共享数据的本地副本
 - 在线程堆栈上分配变量
- **死锁**（持久占用，互斥，不可抢占，循环等待）
 - 2个或更多线程互相等待释放资源
 - 一个线程等待一个永远不会发生的事件，比如挂起的锁
 - 最常见的原因是锁定层次结构
 - **解决**
 - 始终以相同的顺序锁定和解锁，并尽可能避免层次结构
 - 使用原子
- **线程安全**
 - 它在多个线程同时执行期间正常运行
 - **非线程安全指标**
 - 访问全局/静态变量/堆
 - 分配/重新分配/释放 具有全局作用域的资源（文件）
 - 通过句柄和指针间接访问
 - **解决**
 - 任何更改的变量都必须是每个线程的本地变量
 - 例程可以使用互斥来避免与其他线程发生冲突
 - <使例程可重入比添加同步更好>

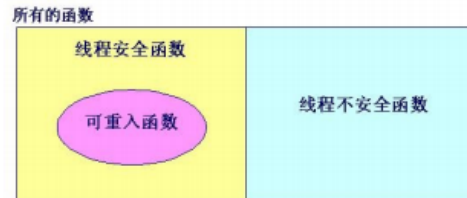
• 可重入函数

✧ **可重入**：多个执行流反复执行一个代码，其结果不会发生改变，通常访问的都是各自的私有栈资源；

➤ **可重入函数**：当一个执行流因为异常或者被内核切换而中断正在执行的函数而转为另外一个执行流时，当后者的执行流对同一个函数的操作并不影响前一个执行流恢复后执行函数产生的结果；

➤ **可重入函数满足的条件**：

- 不使用全局变量或静态变量；
- 不使用malloc或者new开辟出的空间；
- 不调用不可重入函数；
- 不返回静态或全局数据，所有数据都由函数的调用者提供；
- 使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据；
- 不调用标准I/O；



53

• 不平衡的工作负载

- 所有线程以相同的方式处理数据，但一个线程分配了更多的工作，因此需要更多的时间来完成它并影响整体性能
- 解决
 - 并行化内循环
 - 更倾向于细粒度
 - 选择合适的算法
 - 分而治之，master and worker, work-stealing

• 任务粒度

- 一个更大的实体被细分的程度
- 粗粒度意味着更少和更大的组件
- 细粒度意味着更多更小的组件
- 考虑
 - 细粒度会增加任务调度器的工作量
 - 粗粒度可能会导致工作负载不平衡
 - 设置适当粒度的基准

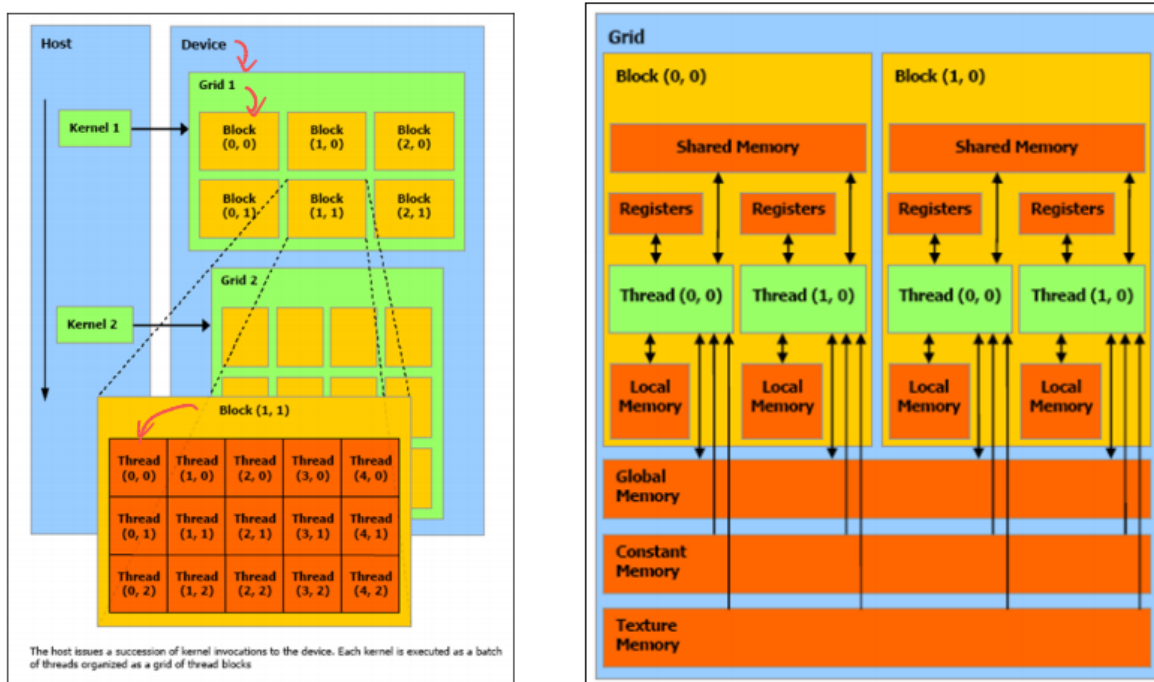
• 锁&等待

- 保护共享数据并确保任务按正确顺序执行
- 使用不当会产生副作用
- 解决
 - 选择适当的同步原语
 - tbb::atomic、InterlockedIncrement、EnterCriticalSection.....

- 使用非阻塞锁——拿不到锁时做其他事，不浪费CPU时间
 - TryEnterCriticalSection、pthread_mutex_try_lock Spin_lock
 - 减少锁粒度
 - 不要做一个锁中心——加锁之后别的线程再也获取不了锁
 - 为共享数据引入并发容器（管程）
- 并行算法PPT58
- 通用开发周期PPT62
 - 线程应用程序需要设计、调试和性能调整步骤的多次迭代
 - 使用工具提高生产力
 - 释放双核和多核处理器的力量
- 消息传递模型
 - 完整的计算机作为构建块，包括 I/O：通过显式 I/O 操作进行通信
 - 编程模型
 - 直接访问私有地址空间（本地内存）
 - 通过显式消息（发送/接收）进行通信
 - 类似于分布式内存 SAS 的高级框图
 - 但通信集成在 IO 级别，无需放入内存系统
 - 比可扩展的 SAS 更容易构建
 - 远离基本硬件操作的编程模型：库或操作系统干预
 - 消息传递抽象
 - 发送指定要传输的缓冲区和发送过程
 - Recv 指定接收进程和要接收的应用程序存储
 - 内存到内存复制，但需要命名进程
 - 发送时的可选标签和接收时的匹配规则
 - 许多开销：复制、缓冲区管理、保护
 - 演变
 - 早期机器：每个链路上的 FIFO
 - 硬件接近编程模型：同步操作
 - 被 DMA 取代，启用非阻塞操作：由系统在目的地缓冲，直到接收
 - 拓扑的递减作用
 - 存储转发路由：拓扑很重要
 - 引入流水线路由使其不那么重要
 - 成本在节点网络接口中
 - 简化编程
 - 迈向架构融合
 - 软件的演进与角色界限模糊

- 通过缓冲区在 SAS 机器上支持发送/接收
- 可以使用散列在 MP 上构建全局地址空间
- 基于页面（或细粒度）的共享虚拟内存
- 编程模型不同，但组织趋同
 - 通过通用网络和通信辅助连接的节点
- 实现也趋同，至少在高端机器中
- 实现
 - 从编程的角度
 - 消息传递实现通常包含一个子程序库
 - 对这些子例程的调用嵌入在源代码中
 - 程序员负责确定所有并行度
 - 从历史上看，自 1980 年代以来，已经有各种消息传递库可用。这些实现彼此之间存在很大差异，使程序员难以开发可移植的应用程序
 - 1992 年，MPI 论坛成立，其主要目标是为消息传递实现建立标准接口
- GPGPU编程模型

CUDA Programming Model



- CUDA 目标：SIMD 编程
 - 硬件架构师喜欢 SIMD，因为它允许非常节省空间和节能的实施
 - 但是，CPU 上的标准 SIMD 指令不灵活，难以使用，编译器难以定位
 - CUDA 线程抽象将以增加硬件为代价提供可编程性
- OpenCL 编程模型
 - OpenCL 是一个框架，用于编写跨异构平台执行的程序，包括 CPU、GPU、DSP、FPGA 和其他处理器或硬件加速器

- 数据并行 - SPMD
 - 工作组中的工作项运行相同的程序
 - 使用工作项 ID 并行更新数据结构以选择数据并指导执行
- 任务并行
 - 每个工作组一个工作项.....用于粗粒度任务级并行性
 - 本机函数接口：从 OpenCL 命令队列运行任意代码的陷阱门
- OpenCL 的两种数据并行方式
 - 显式 SIMD 数据并行性
 - 内核定义了一个指令流
 - 使用宽向量类型的并行性
 - 大小矢量类型以匹配本机硬件宽度
 - 结合任务并行性以利用多个内核
 - 隐式 SIMD 数据并行性（即着色器样式）
 - 将内核编写为“标量程序”
 - 使用算法自然大小的矢量数据类型
 - 内核由编译器/运行时/硬件自动映射到 SIMD 计算资源和内核
- 数据并行系统
 - 编程模型
 - 对数据结构的每个元素并行执行的操作
 - 逻辑上单线程控制，执行顺序或并行步骤
 - 从概念上讲，与每个数据元素相关联的处理器
 - 架构模型
 - 由许多简单、廉价的处理器组成的阵列，每个处理器的内存都很少：处理器不按指令排序
 - 连接到发出指令的控制处理器
 - 专业通用通信，廉价全球同步
 - 原始动机
 - 匹配简单的微分方程求解器
 - 集中高成本的指令获取和排序
 - 数据流架构
 - 计算、架构和语言的非冯诺依曼模型
 - 程序未附加到程序计数器
 - 指令的可执行性和执行完全取决于指令输入参数的可用性
 - 指令执行的顺序是不可预测的：i. e. 行为是不确定的
 - 静态和动态数据流机器
 - 静态数据流机器：使用常规内存地址作为数据依赖标签

- 动态数据流机器：使用内容可寻址内存 (CAM)

以上内容整理于 [幕布文档](#)