2 并行架构

- 回顾 架构趋势
  - VLSI 的最大趋势是利用的并行性增加
    - 直到 1985 年：位级并行度：4 位 -> 8 位 -> 16 位
      - 32 位后变慢，正在采用 64 位
    - 80 年代中期到 90 年代中期：指令级并行
      - 流水线和简单指令集 (RISC)
      - 片上缓存和功能单元 => 超标量执行
      - 更复杂：乱序执行
    - 现在：超线程，多核
- **Flynn's Taxonomy**（分类方法）
  - 基于计算机体系结构的分类指令和数据流的数量
  - SISD架构、SIMD、MIMD：现代并行系统、MISD：无

AVX指令集是Sandy Bridge和Larrabee架构下的新指令集。AVX是在之前的128位扩展到和256位的单指令多数据流。而Sandy Bridge的单指令多数据流演算单元扩展到256位的同时数据传输也获得了提升，所以从理论上看CPU内核浮点运算性能提升到了2倍。

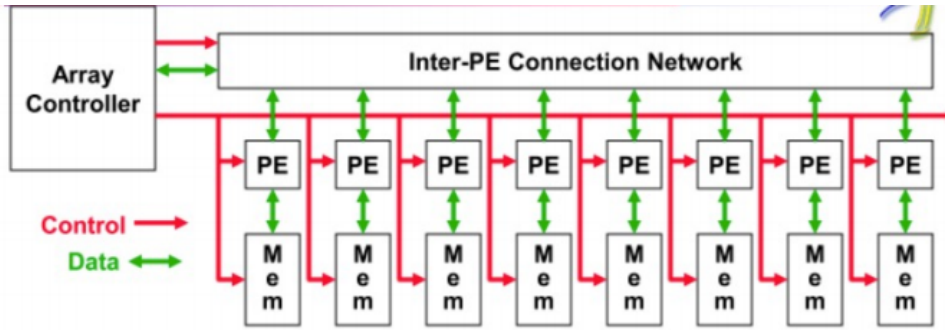
Intel AVX指令集，在单指令多数据流计算性能增强的同时也沿用了的MMX/SSE指令集。不过和MMX/SSE的不同点在于增强的AVX指令，从指令的格式上就发生了很大的变化。x86(IA-32/Intel 64)架构的基础上增加了prefix(Prefix)，所以实现了新的命令，也使更加复杂的指令得以实现，从而提升了x86 CPU的性能。

```
Gcc  -mavx -mavx2 -mfma -msse -msse2 -msse3 -Wall -O
```

- 单处理器并行
  - 并行化无处不在
    - 现代处理器芯片有大约 10 亿个晶体管
      - 显然必须让它们并行工作
      - 问题：程序员必须了解多少这种并行性?
    - 单处理器计算机架构如何提取并行性?
      - 通过在指令流中找到并行性
      - 称为“指令级并行”（ILP）
      - 理论：对程序员隐藏并行性
    - 计算机架构师的目标直到 2002 年左右：
      - 向所有人隐藏潜在的并行性：操作系统、编译器、程序员
    - **ILP 技术示例**
      - **流水线：重叠指令的各个部分**
      - 超标量执行：同时做多件事
      - VLIW：让编译器指定哪些操作可以并行运行

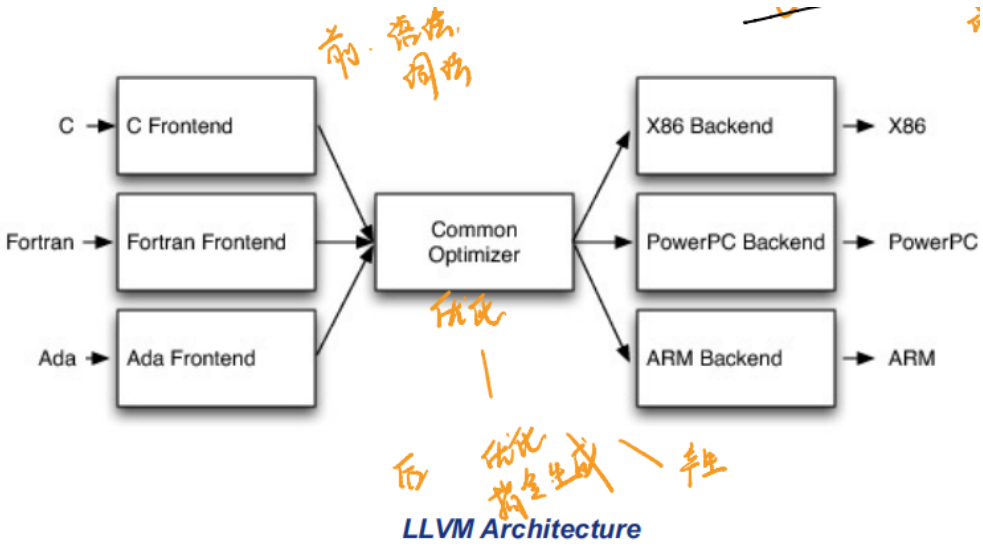
        - 向量处理：指定类似（独立）操作的组
        - 乱序执行（OOO）：允许长时间的操作发生
- **流水线**
    - 带宽：loads/hour
    - 流水线有助于带宽但无延迟
        - 带宽受限于最慢的流水线阶段
        - 潜在加速 = pipe的阶段数
    - 理想情况下：CPI（cycles/instruction）=1
        - 平均而言，将一条指令放入流水线，取出一条
        - 超标量：启动多个指令/周期
    - 流水线的限制
        - 防止任意划分的开销
            - 锁存器的成本（阶段之间）限制了阶段内可以做的事情
            - 设置最小工作量/阶段
        - 冒险会阻止下一条指令在其指定的时钟周期内执行
            - 结构冒险：尝试使用相同的硬件同时做两件不同的事情
            - 数据冒险：指令依赖于仍在流水线中的先前指令的结果
                - 必须等待（“停顿”）才能完成结果（不存在“回到过去”！）
                - 最终结果是 CPI > 1
                - 超标量增加了危险的频率
            - 控制冒险：由获取指令和决定控制流变化（分支和跳转）之间的延迟引起
        - 超标量增加了危险的发生——更多冲突指令/周期
- 乱序执行（out of order）
    - 核心思想：允许停顿后的指令继续进行
    - OOO 调度中的动态调度问题
        - 必须将结果与说明的消费者相匹配
        - 精确中断
- 现代指令级并行
    - 动态调度，乱序执行
        - 当前的微处理器每个周期获取 6-8 条指令
        - 流水线的深度为 10 个周期 → 许多重叠的指令同时执行，尽管工作经常被丢弃
    - 会发生什么
        - 抓取一堆指令，确定它们的所有依赖关系，尽可能消除dep，将它们全部放入执行单元，让每个指令在其依赖关系解决后继续前进

        - 看起来好像是按顺序执行的
    - 处理冒险：可能需要猜测！
        - 称为“推测执行”
        - 推测分支结果、依赖关系，甚至值！
            - 如果正确，则不需要为结果停止→产生性能
            - 如果不正确，浪费时间和精力
        - 如果猜测错误，必须能够撤消结果
        - 问题：猜测的准确性随着流水线中同时指令的数量而降低
    - 巨大的复杂性
        - 许多组件的复杂度为n^2（问题宽度）
        - 耗电大问题
    - 限制力量：时钟速度和 ILP
        - 结论：并行化必须拓展到软件上
- 向量处理——SIMD
    - 要求程序员（或编译器）识别并行性，硬件不需要重新提取并行度
    - 许多多媒体/HPC 应用程序是向量处理的自然消费者
    - SIMD架构



        - 中央控制器向多个处理元件 (PE) 广播指令
            - 整个阵列只需要一个控制器
            - 只需要存储一份程序代码副本
            - 所有计算完全同步
        - 最近回归人气
            - GPU（图形处理单元）具有 SIMD 属性
            - 然而，多核行为也是如此，所以 SIMD 和 MIMD 的混合
        - Vector和SIMD执行之间的双重
    - GP-GPU
        - 2006 年，Nvidia 推出了 GeForce 8800 GPU，支持一种新的编程语言：CUDA
            - Compute Unified Device Architecture
            - OpenCL 是相同想法的供应商中立版本
        - 思路：利用 GPU 计算性能和内存带宽，为通用计算加速一些内核

        - 附加处理器模型：主机 CPU 向 GP-GPU 发出数据并行内核执行
    - CPU上的向量处理——Intel SPMD编译器（ISPC）
        - ISPC是一个在Intel SIMD架构上支持SPMD编程模型基于LLVM的语言和编译器
            - 目标
                - 高性能编码
                - 使用额外资源进行扩展：核心数和 SIMD 向量宽度
                - 易于采用和集成
                - 使用来自程序员的最少输入来胜过基于循环的自动向量化编译器
        - low level virtual machine(LLVM)
            - LLVM 编译器基础架构
                - 为构建编译器提供可重用的组件
                - 减少构建新编译器的时间/成本
                - 构建静态编译器、JIT、基于跟踪的优化器，...
            - LLVM 编译器框架



LLVM Architecture

                - 使用 LLVM 基础架构的端到端编译器
                - C 和 C++ 是健壮和激进的：
                - Java、Scheme 等正在开发中
                - 为 X86、Sparc、PowerPC 发出 C 代码或本机代码
            - SPMD编程抽象
                - 调用 ISPC 函数会导致 ISPC“程序实例”的“帮派”
                - 所有实例同时运行 ISPC
                - 返回时，所有实例都已完成
- 多线程：包括pthreads
    - 线程级并行（TLP）
        - ILP 利用循环或直线代码段中的隐式并行操作
        - TLP 通过使用本质上是并行的多个执行线程来明确表示

        - 线程可以在单个处理器上，或者，在多个处理器上
    - 并发与并行
        - 并发是指两个任务可以在重叠的【时间段】内启动、运行和完成。 这并不一定意味着它们会同时运行。例如，单线程机器上的多任务处理
        - 并行性是指任务实际上同时运行，例如。 在多核处理器上
    - 目标：使用多个指令流来提升（MI）
        - 运行许多程序的计算机的吞吐量
        - 多线程程序的执行时间
- 线程创建的常见概念
    - cobegin/coend
        - 块中的语句并行执行
        - 嵌套的，所以cobegin和coend一一对应
    - fork/join
        - fork使得并行执行
        - 如果没有完成，等待join
    - 未来线程交互，为使用返回值而等待
    - 代码中表达的线程可能不会变成独立的计算，仅在处理器空闲时创建线程
        - 示例：线程窃取运行时
- **POSIX threads概览**
    - POSIX：用于 UNIX 的便携式操作系统接口
        - 操作系统实用程序的接口
    - Pthreads：POSIX 线程接口
        - 创建和同步线程的系统调用
        - 在类 UNIX 操作系统平台上应该是相对统一的
        - 最初是 IEEE POSIX 1003.1c
    - Pthreads 包含支持
        - 创建并行性
        - 同步
        - 没有显式支持通信，因为共享内存是隐式的； 一个指向共享数据的指针被传递给一个线程
            - 仅适用于堆！ 栈的变量是私有的
- 共享数据和线程
    - 在 main 之外声明的变量是共享的
    - 在堆上分配的对象可能是共享的（如果传递了指针）
    - 栈上的变量是私有的：将指向这些变量的指针传递给其他线程可能会导致问题

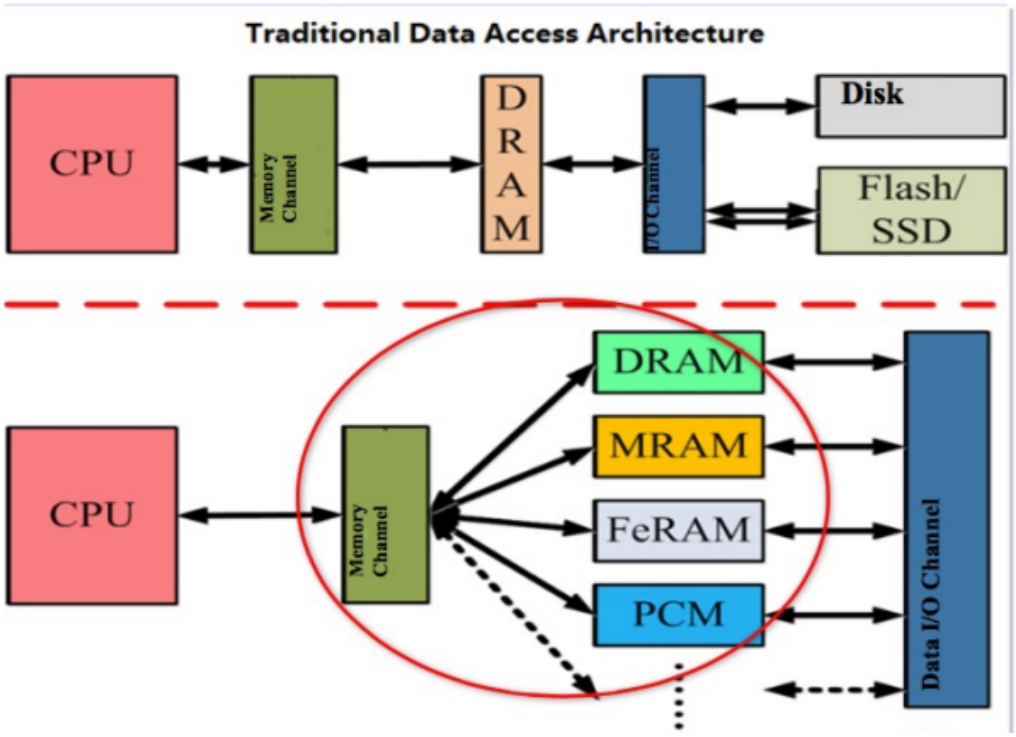
- 通常通过创建一个大型“线程数据”结构来完成，该结构作为参数传递给所有线程——参数多于1，创建结构体来传参数
  - char *message = "Hello World!\n";
  - pthread_create(&thread1, NULL, print_fun,(void*) message);
- 循环级并行
  - 许多应用程序在循环中具有并行性
  - 但是线程创建的开销是不小的
    - update_stuff 应该有大量的工作
  - 常见的性能陷阱：线程过多
    - 在现代架构上，创建线程的成本是几千个周期的几十万
    - 解决方案：线程阻塞：使用少量线程，通常等于内核/处理器或硬件线程的数量
- 线程调度
  - 创建后，该线程何时运行：这取决于操作系统或硬件，但它最终会运行，即使您的线程数多于内核数，但是调度可能不适合您的应用程序
  - 程序员在某些情况下可以提供提示或亲和力
    - 例如，精确创建 P 个线程并分配给 P 个核心。将线程绑定到某个core上可以提升性能：防止再次调度时，去到另外的一个core上产生大量的 cache miss
  - 可以为某些系统提供用户级调度
  - 基于编程模型的特定于应用程序的调整
- 多线程执行
  - 多任务操作系统
    - 给人一种“错觉”，即多件事同时发生
    - 以粗粒度时间切换（例如：10ms）
  - 硬件多线程：多个线程同时共享处理器（几乎没有操作系统帮助）
    - 硬件进行切换
      - 用于在少量循环中进行快速线程切换的硬件
      - 比操作系统切换快得多
    - 处理器复制每个线程的独立状态
      - 例如，单独的寄存器文件副本、单独的 PC，以及运行独立程序的单独页表
    - 通过已经支持多进程的虚拟内存机制共享内存
  - 何时在线程之间切换?
    - 每个线程的交替指令（细粒度）
    - 当一个线程停止时，可能是因为缓存未命中，可以执行另一个线程（粗粒度）

        - 指令级并行和线程级并行结合
            - TLP 和 ILP 在程序中利用两种不同的并行结构
            - 面向 ILP 的处理器能否从利用 TLP 中受益?
                - 由于代码中的停顿或依赖性，功能单元在为 ILP 设计的数据路径中经常处于空闲状态
                - TLP 用作独立指令的来源，可能会保持处理器在停顿期间忙碌
                - TLP 用于占用 ILP 不足时原本空闲的功能单元
            - 称为“同时多线程”
                - 英特尔将此重命名为“超线程”
- **单处理器存储系统**
    - 存储系统性能的限制
        - 内存系统，而不是处理器速度，通常是许多应用程序的瓶颈。
        - 内存系统性能主要由延迟和带宽这两个参数来衡量。
        - **延迟**是从发出内存请求到处理器上数据可用的时间。
        - **带宽**是内存系统将数据泵送到处理器的速率。
        - 内存墙——内存性能严重限制CPU性能发挥的现象

        - 架构师如何解决这个差距?
            - **在 CPU 和 DRAM（动态随机存取存储器）之间放置小型、快速的“高速缓存”存储器。**
            - 创建“内存层次结构”
    - **局部性原则**——程序在任何时刻访问相对较小部分的地址空间
        - 两种不同类型的局部性
            - 时间局部性（Locality in Time）：如果一个项目被引用，它很快就会被再次引用（例如，循环，重用）
            - 空间局部性（Locality in Space）：如果一个项目被引用，地址在附近的项目往往会很快被引用（例如，直线代码，数组访问）
        - 过去 25 年，硬件速度依赖局部性
    - **内存层次结构**

    - 利用局部性原则
        - 提供与最便宜的技术一样多的内存
        - 以最快的技术提供的速度提供访问
    - 寄存器->片上缓存->| L1 L2 L3 cache（SRAM）->主存（DRAM）->辅存（磁盘）->三级存储（磁带）-----虚拟内存
- 内存访问性能
    - 命中：数据出现在上层的某些块中（例如：块 X）
        - 命中率：在上层发现的内存访问比例
        - Hit Time：访问上层的时间，包括 RAM 访问时间 + 确定命中/未命中的时间
    - Miss：需要从较低级别（Block Y）的块中检索数据
        - 未命中率 = 1 - (命中率)
        - Miss Penalty：替换上层块的时间 + 将块交付给处理器的时间
    - 命中时间 << 未命中
- **使用缓存有效提高内存延迟**
    - 高速缓存是处理器和 DRAM 之间的小而快的内存元素。
        - 充当低延迟高带宽存储。
        - 如果一条数据被重复使用，这个内存系统的有效延迟可以通过缓存来降低。
    - 缓存满足的数据引用比例称为缓存命中率。
    - 内存系统上的代码实现的缓存命中率通常决定了它的性能
- 内存带宽的影响
    - 内存带宽由内存总线和内存单元的带宽决定。
        - 内存带宽可以通过增加内存块的大小来提高。
        - 底层系统需要 l 个时间单位（其中 l 是系统的延迟）来传递 b 个单位的数据（其中 b 是块大小）。
    - 带宽比延迟每年提高 23%，而每年提高 7%
    - 请注意，增加块大小不会改变系统的延迟。
    - 在物理上，它可以被视为连接到多个存储体的宽数据总线（4 个字或 128 位）。
    - 实际上，如此宽的总线的成本很高。
    - 在更实用的系统中，在检索到第一个字之后，在随后的总线周期上在存储器总线上发送连续的字。
    - 为了利用全带宽
        - 假设数据布局是存储器中的连续数据字被连续指令使用
        - 参考空间位置
    - 如果空间局部性较差

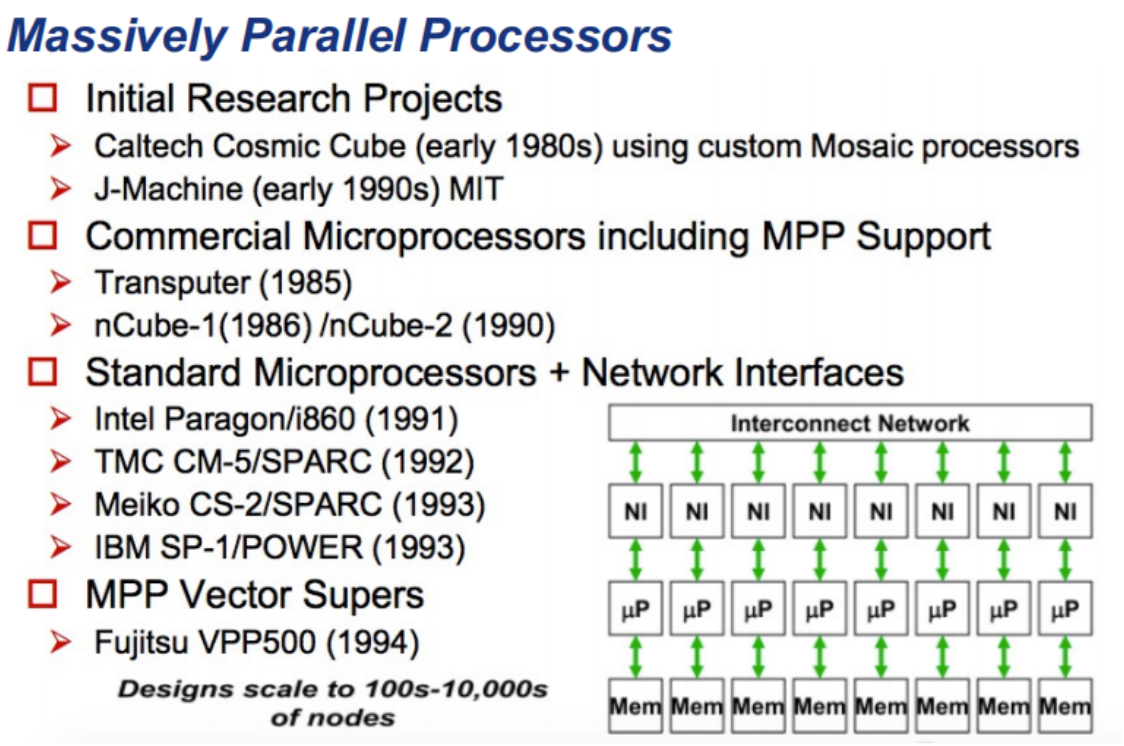
        - 计算通常需要重新排序以增强参考的空间局部性。
    - 内存层次结构教训
        - 缓存极大地影响性能——不考虑内存层次就不能考虑性能
        - 一个简单程序的实际性能可能是架构的一个复杂功能
            - 架构或程序的微小变化会显着改变性能
            - 要编写快速程序，需要考虑架构，在顺序或并行处理器上同样适用
            - 我们想要简单的模型来帮助我们设计高效的算法
        - 提高缓存性能的常用技术，称为阻塞或平铺
            - 思路：用分治法来定义适合寄存器/L1-cache/L2-cache的问题
        - 自动调整：通过实验处理复杂性
            - 生成几个不同版本的代码——不同的算法、阻塞因子、循环排序等
            - 对于每个架构，运行不同的版本，看看哪个是最快的
            - 可以（原则上）导航复杂的设计选项以获得最佳效果
    - **新的内存计算架构** SCM



- 多核芯片
    - 并行芯片级处理器
    - *Sun’s T1 (“Niagara”) *
    - 嵌入式并行处理器
        - 通常体现旧架构的风格和思想的混合
        - 暴露的内存层次结构和互连网络
            - 程序员对“金属”进行编码以获得最佳效果
            - 成本/功耗/性能

        - 跨平台的可移植性不太重要
        - 自定义同步机制
            - 互锁的通信通道（如果数据未准备好，则读取处理器块）
            - barrier信号
            - 专门的原子操作单元
        - 更多、更简单的内核
    - IBM 单元处理器 (Playstation-3)
    - GPU
        - GPU（图形处理器单元）可用于许多台式机
        - 示例：16 个内核，类似于具有 8 个通道的矢量处理器（总共 128 个流处理器）
            - 处理 32 个 SIMD 组中的线程（“warp”）
            - 一些在硬件中完成的条带化
        - 线程可以分支，但与所有线程运行相同代码时相比会损失性能
        - 完整的并行编程环境 (CUDA)
            - 许多并行代码已移植到这些 GPU
            - 对于某些数据并行应用程序，GPU 提供最快的实现
    - *Nvidia Tesla GPU *
- 并行计算架构
    - **什么是并行架构**
        - 具有多个处理器的机器
        - 并行计算机是协作以快速解决大型问题的处理元素的集合 一些广泛的问题
        - 资源分配：有多大的集合？元素有多强大？多少内存?
        - 数据访问、通信和同步：元素如何合作和交流？处理器之间如何传输数据？合作的抽象和原语是什么?
        - 性能和可扩展性：这一切如何转化为性能？ 它是如何扩展的?
        - 并行结构一般是指并行体系结构和软件架构采取并行编程。主要目的是使更多任务或数据同时运行。并行体系结构是指许多指令能同时进行的体系结构；并行编程一般有以下模式：共享内存模式；消息传递模式；数据并行模式。
        - 并行计算机是一组处理元素的集合，它们协同快速地解决一些大问题。
    - MIMD 机器
        - 多个独立的指令流、程序计数器等，称为“多处理”而不是“多线程”，虽然，多个处理器中的每一个都可以是多线程的，当独立指令流局限于单芯片时，就变成了“多核”处理器
        - 共享内存：通过内存进行通信
            - 选项 1：没有硬件全局缓存一致性
            - 选项 2：硬件全局缓存一致性
        - 消息传递：通过消息进行通信

        - 应用程序在节点之间发送显式消息以进行通信
    - 对于大多数机器来说，共享内存建立在消息传递网络之上
    - 基于总线的机器是“例外”
- **MIMD 的并行架构包括哪些实现类型**
- 对称多处理器：内置多个处理器，共享内存通信，每个处理器运行操作系统的拷贝，例如现在的多核芯片。
- 通过主机的独立 I/O 的非统一共享内存：多处理器，每个都有本地内存，通用可扩展网络，节点上非常轻的 OS 提供简单的服务，调度/同步，用于 I/O 的网络可访问主机。
- 集群：多台独立机接入通用网络，通过消息沟通。
- **MPP**



以上内容整理于 幕布文档