基于多核系统的并行算法和实例分析

免费的午餐结束了

- 并行时代已经到来
  - 频率提升受到限制
  - 多核 CPU 成为主流并向着更多的核发展
- 为什么我们需要并行计算？
  - 更快的完成计算（更低的延迟）
  - 处理更大规模的问题（更高的吞吐量）

为什么并行计算是一个难题

- 本质上说并行计算是一种优化的手段
- 糟糕的并行算法在并行化上所产生的额外开销常常大于计算本身
- 没有一种方法可以自动的把串行程序并行化
- 没有银弹——库、框架、语言、开发工具

如何应对？

- 算法是并行计算的灵魂
- 用并行的思想重新思考算法

并行算法的核心问题

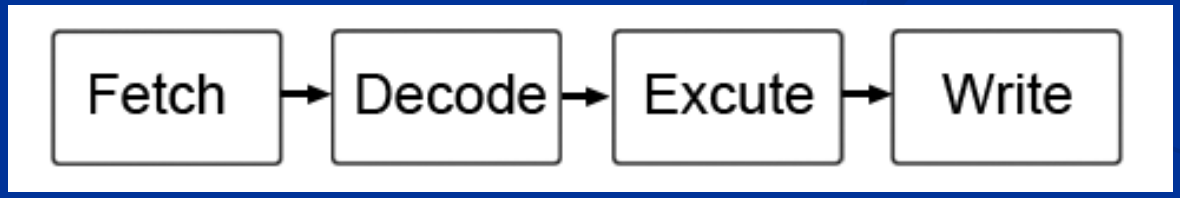
任务分解

基于时间的分解

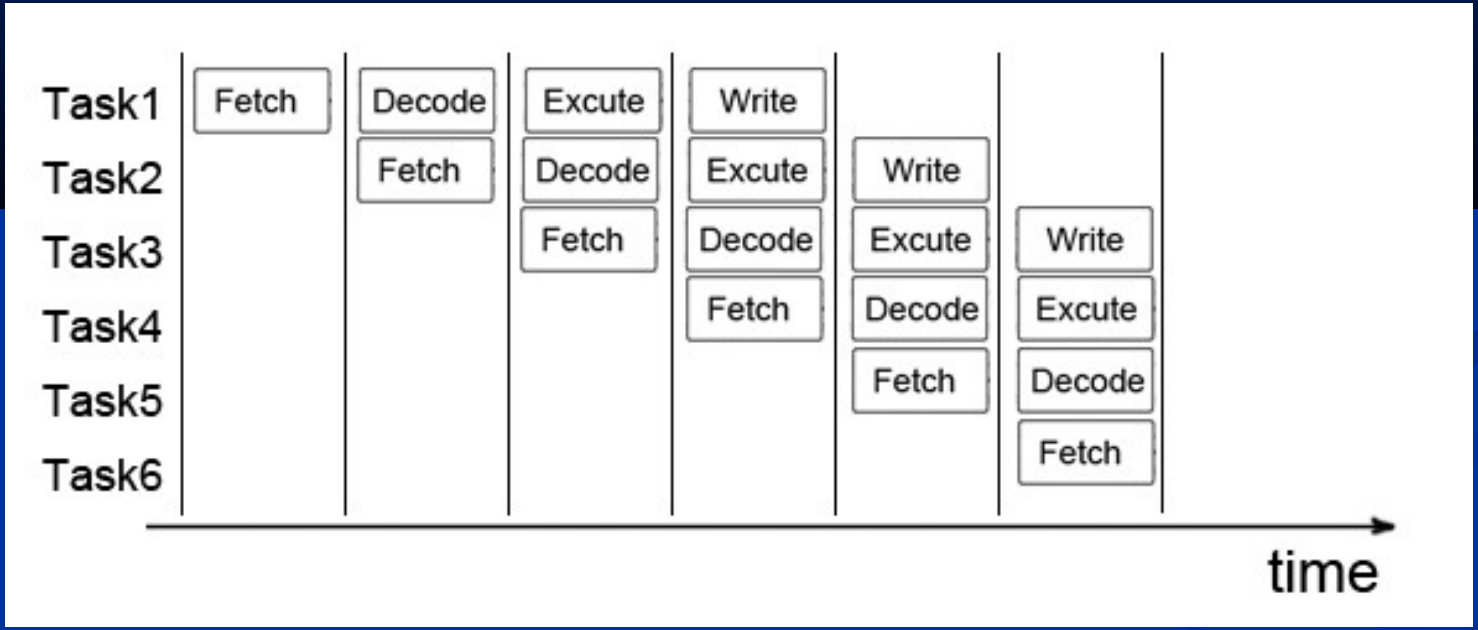
- Pipeline

  将任务在时间上分解为若干独立的步骤，为每个步骤都赋予一个执行单元，所有的执行单元一起协同工作就构成了流水线。

一个简单的 4 级指令流水线的例子：



流水线工作的时序图：


Task1
Task2
Task3
Task4
Task5
Task6
Fetch
Decode
Fetch
Excute
Decode
Fetch
Write
Excute
Decode
Fetch
Write
Excute
Decode
Fetch
Write
Excute
Decode
Fetch
time

Pipeline 的优缺点

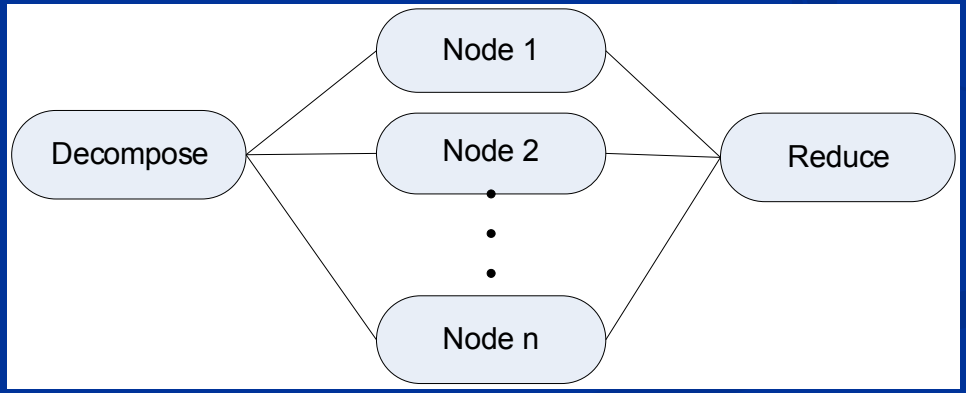
- 优点
  - 很容易将串行算法转换为流水线算法
  - 非常有效的提升任务吞吐量
  - 适合在硬件上实现
- 缺点
  - 不能降低单个任务的计算时间
  - 流水线级数是固定的，不具 Scalability
  - 一般情况下不适合于软件实现

基于空间的分解

- Map Reduce

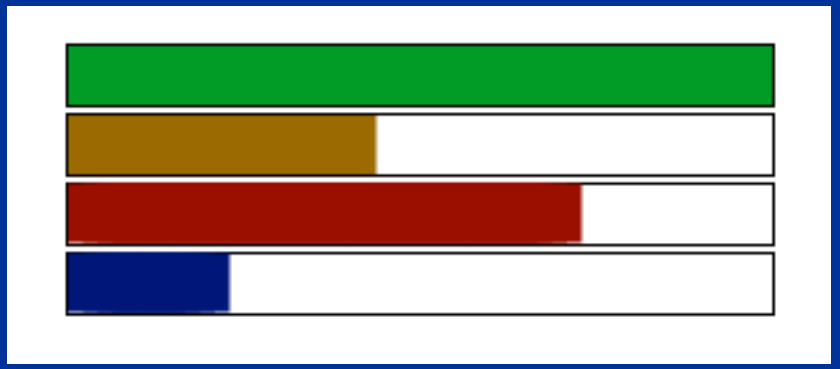
  将一个任务在空间上划分为若干独立的子任务，应用 Map 操作同时计算这些子任务，最后用 Reduce 操作将计算结果合并。

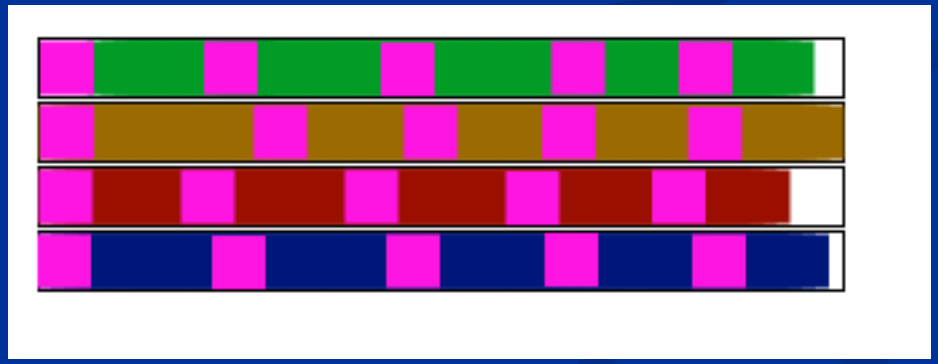
MapReduce 的基本框架：



任务分解需要考虑的问题

- 任务之间的独立性
- 任务的粒度
  - 粒度太大，容易出现负载不均衡
  - 粒度太小，并行化的 Overhead 会影响性能





并行环境下的内存问题

- 内存访问对处理器来说是一种 IO
  - 有 IO 就有时序，小心同步问题
- 避免使用锁
  - 锁的开销往往很大
  - 容易出错
- 定制并行环境下的内存分配器
- Cache 的影响
  - 任务的分解要考虑如何利用好 cache
  - 避免 false-sharing

共享内存 VS 消息传递

- 共享内存
  - 所有的核心可以访问在同一地址空间的内存
  - 高性能的数据共享
  - 需要小心的同步内存的访问
  - 适用于多线程模型
- 消息传递
  - 每个任务有独立的内存地址空间
  - 使用收发消息来实现任务之间的数据交换
  - 通讯的代价很大
  - 适用于分布式系统

抽象内存 IO 模型
Scatter vs Gather

- Scatter

```
for(i = 0; i < N; ++i)
    A[F(i)] = B[i];
```

- Gather

```
for(i = 0; i < N; ++i)
    A[i] = B[F(i)];
```

- Gather 是并行化友好的内存 IO，而 Scatter 不是
  - 在设计并行算法时，考虑用 Gather 代替 Scatter
  - 在用 Scatter 时，使用一些技巧来避免使用锁同步

常用的并行计算库介绍

- OpenMP
  - 语言级的并行扩展，使用简单
  - 对 parellel_for 提供了良好的支持
  - 缺乏对线程和内存的控制机制
- TBB
  - 基于 C++ 模板库
  - 功能强大，提供了丰富的并行化模式 (parallel_for, parallel_reduce, parallel_scan, parallel_pipline...)，并行化容器，轻量级锁等等...

实现一个最简单的并行计算框架

```
namespace Parallel
{
  struct Kernel
  {
    virtual void Execute(int index) = 0;
  };
  void Initialize(int maxThread = -1);
  void Destroy();
  void For(int from, int to, Kernel* kernel);
  int WorkerCount();
  int CurrentWorkerId();
};
```

一个简单的例子：数组求和

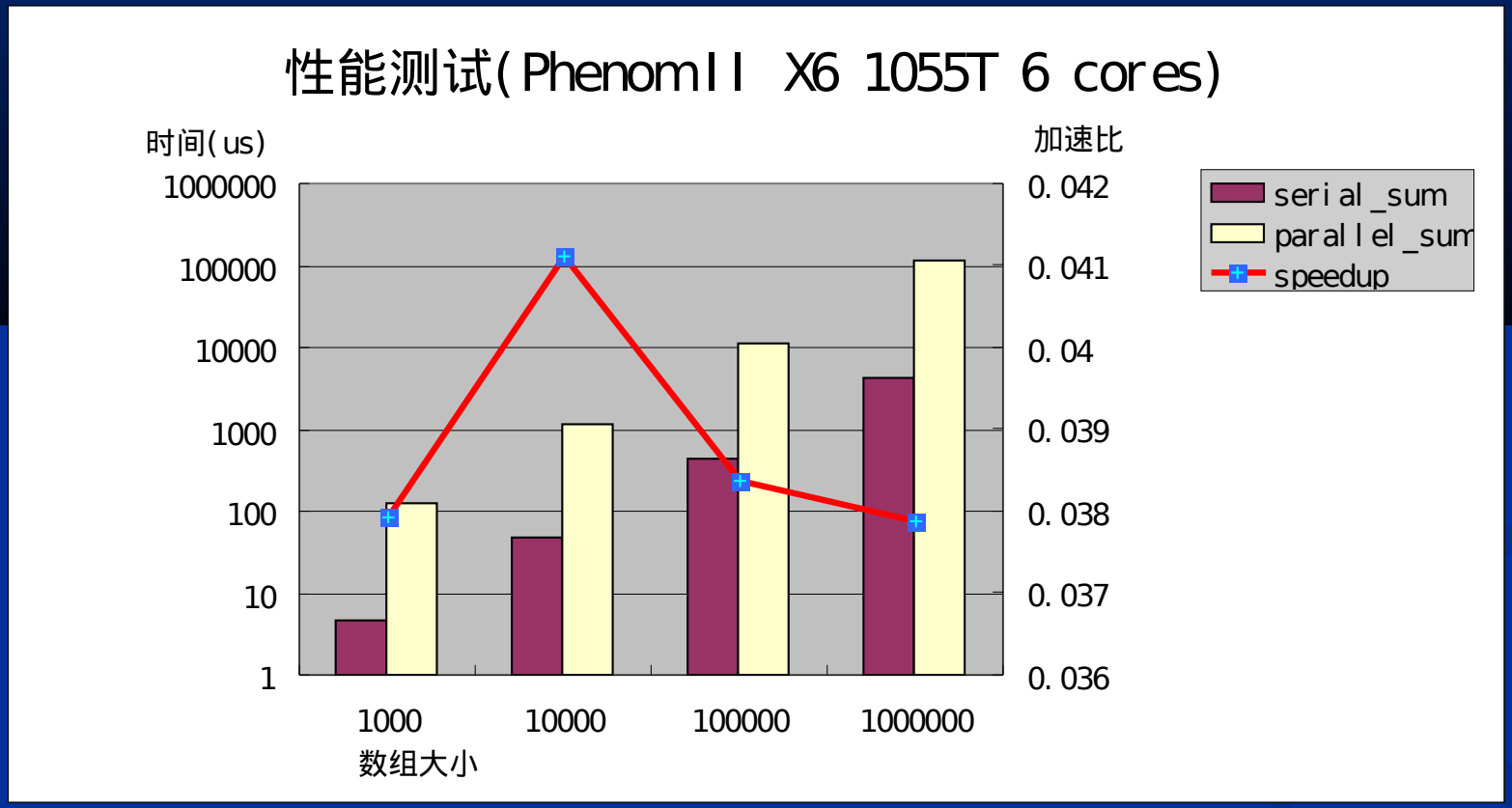
串行版本

```
float serial_sum(const float* numbers, int count)
{
  float sum = 0;
  for(int i = 0; i < count; ++i)
    sum += numbers[i];
  return sum;
}
```

一个幼稚的并行实现：

```
float parallel_sum(const float* numbers, int count)
{
  float sum = 0;
  struct SumKernel : Parallel::Kernel{
    float results[MAX_THREAD_COUNT];
    const float* numbers;

    SumKernel(const float* numbers){
      this->numbers = numbers;
      ZeroMemory(results, sizeof(results));
    }

    void Execute(int index){
      results[Parallel::CurrentWorkerId()] += numbers[index];
    }

    float Reduce(){
     float sum = 0;
     for(int i = 0; i < Parallel::WorkerCount(); ++i)
       sum += results[i];
     return sum;
   }
  };

  SumKernel kernel(numbers);
  Parallel::For(0, count, &kernel);
  return kernel.Reduce();
}
```

性能比较



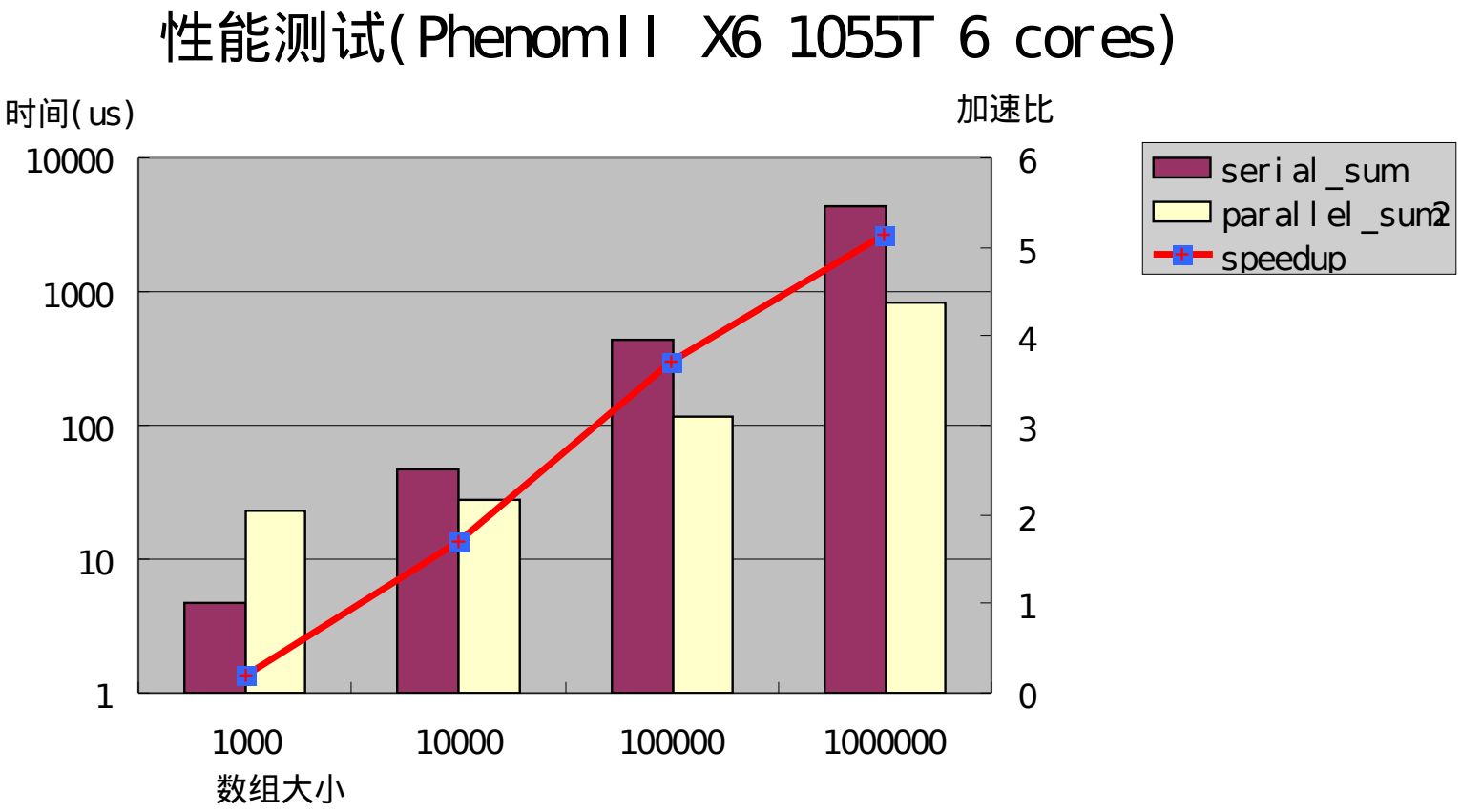
- 问题
  - 任务粒度太小
  - 存在 false-sharing

改进一下

```
float parallel_sum2(const float* numbers, int count)
{
  float sum = 0;
  struct SumKernel : Parallel::Kernel{
    float results[MAX_THREAD_COUNT];
    const float* numbers;
    int count;

    SumKernel(const float* numbers, int count){
      this->count = count;
      this->numbers = numbers;
      ZeroMemory(results, sizeof(results));
    }

    void Execute(int index){
      int c = count / Parallel::WorkerCount();
      int start = index * c;
      float result = 0;
      for(int i = start; i < start + c; ++i)
        result += numbers[i];
      results[index] = result;
    }
```

```
    float Reduce(){
      float sum = 0;
      for(int i = 0; i < Parallel::WorkerCount(); ++i)
        sum += results[i];
      for(int i = 0; i < count % Parallel::WorkerCount();
   ++i)
        sum += numbers[count - i - 1];
      return sum;
    }
  };

  SumKernel kernel(numbers);
  Parallel::For(0, Parallel::WorkerCount(), &kernel);
  return kernel.Reduce();
}
```

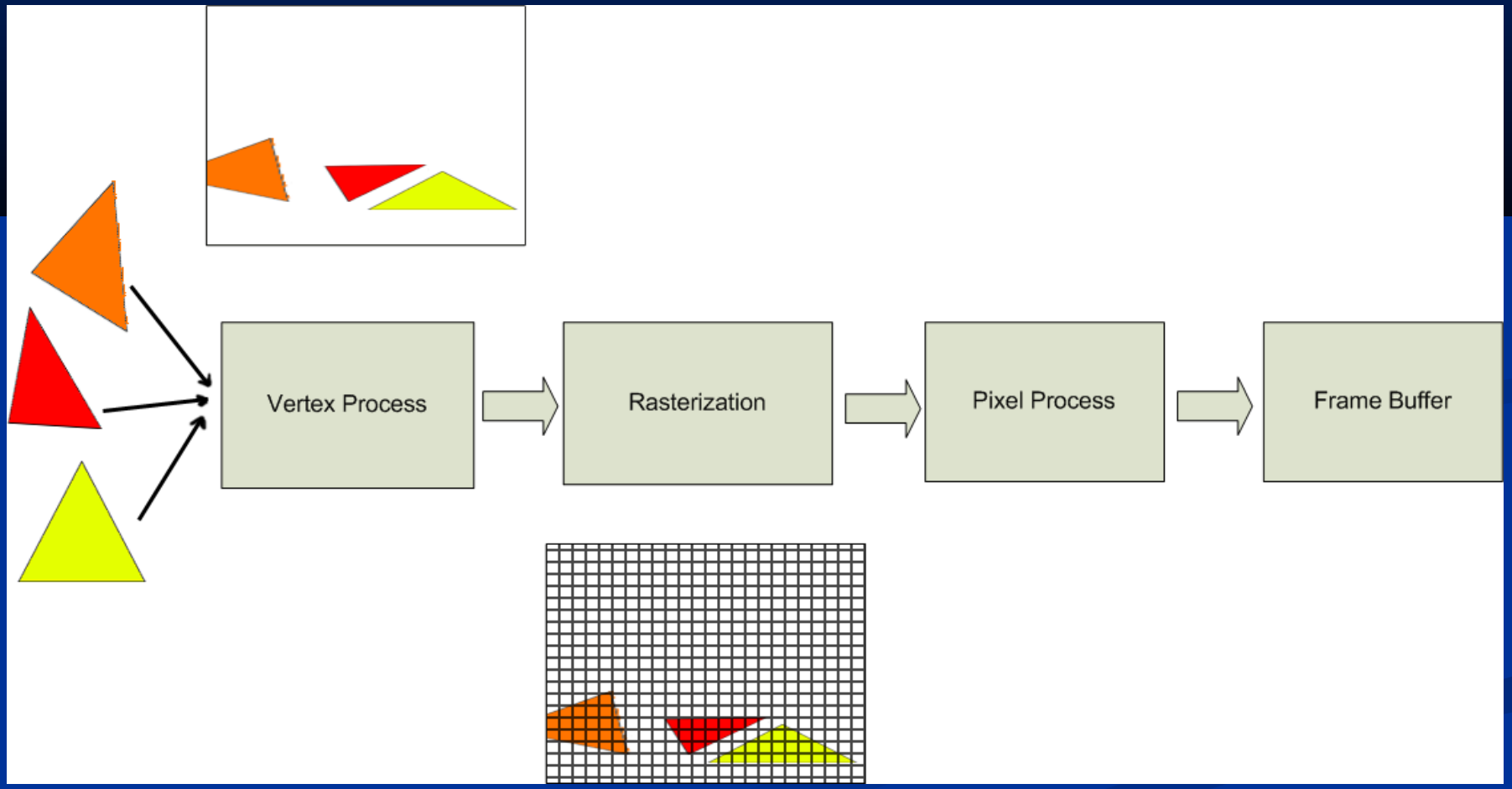
性能测试


性能测试(PhenomII X6 1055T 6 cores)
时间(us)
加速比
10000
1000
100
10
1
6
5
4
3
2
1
0
serial_sum
parallel_sum2
speedup
1000
10000
100000
1000000
数组大小

软件光栅化渲染器中的并行算法

- Hybird3D 简介
  - 一款基于软件并行算法的光栅化 / 光线跟踪混合渲染器
- 什么是光栅化？
  - 将几何图元转换成点阵图像的处理过程

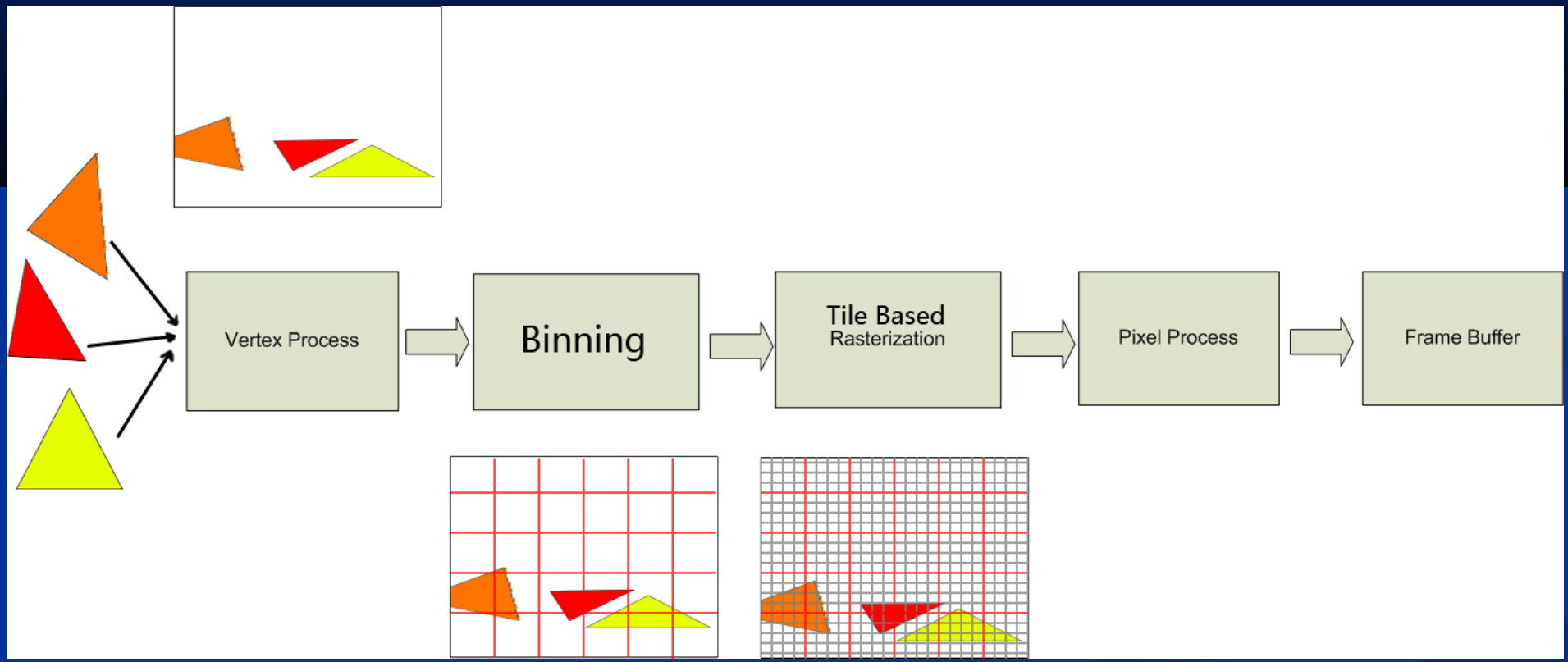
传统的光栅化硬件流水线



软件光栅化的并行算法

- 问题：
  - 光栅化是一个 Scatter 过程，难以直接并行
  - 传统的硬件光栅化依赖细粒度的硬件流水线来实现，不适用于软件算法
- 解决方案：
  - 基于 Tile 的任务划分，将屏幕划分为若干 Tiles
  - 利用 Binning 算法将图元分配到 Tile 中
  - 并行的光栅化这些 Tiles

基于 Tile 的光栅化流水线



伪代码实现

```
struct tile
{
  list<primitive> primitives[thread_count];
};
```

- Bin primitives to tiles

```
parallel for(primitive in primitives)
  for each(tile in tiles)
  if(primitive intersect tile)
  insert primitive to tile[curr_thread]
```

- Render the tiles

```
parallel for(primitives in tiles)
  merge the primitive lists into one list
  for each(primitive in primitive list)
  raster the primitive
```

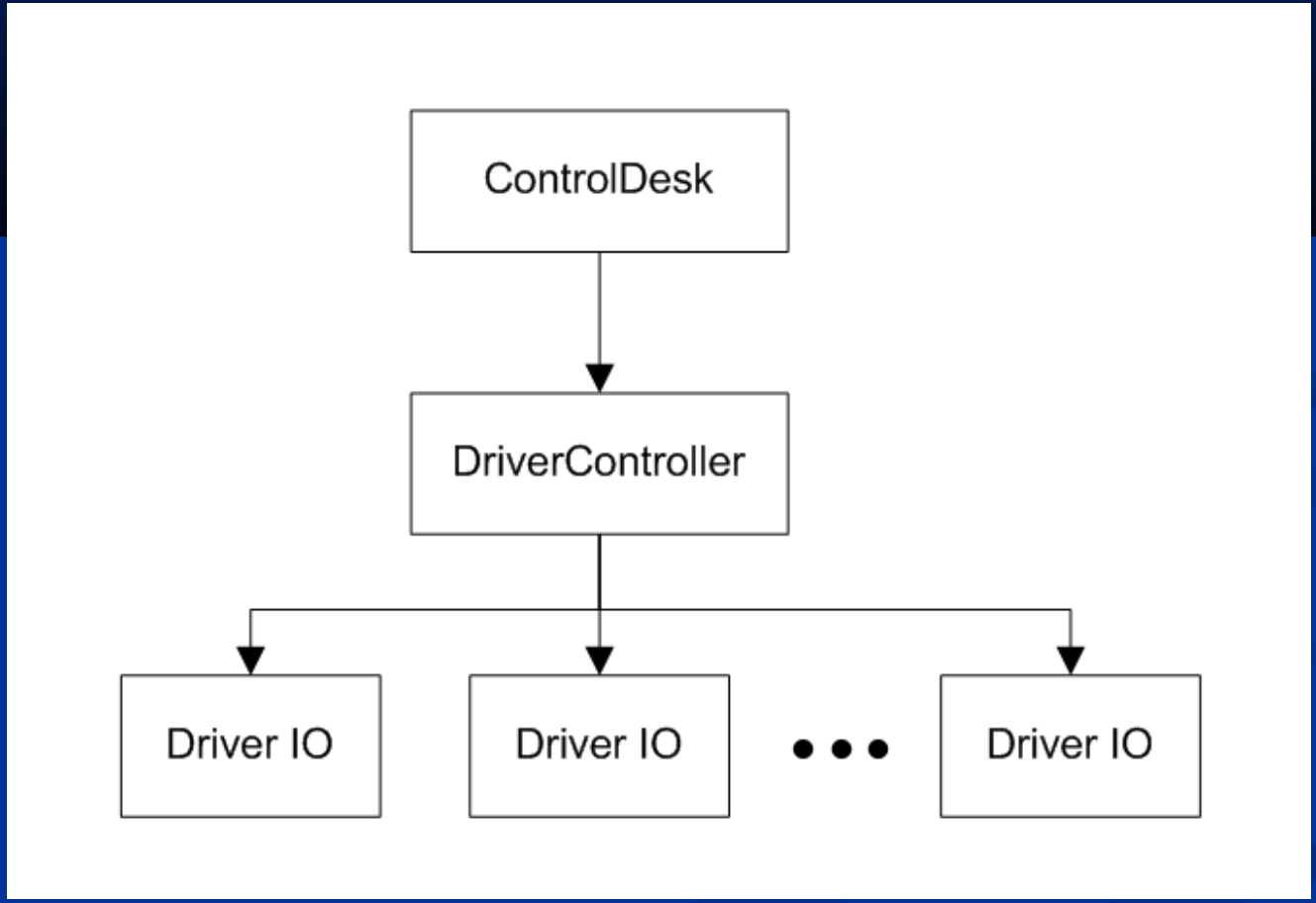
一些实现细节

- 利用 SIMD 实现更细粒度的并行计算
  - SIMD 光栅化算法（一条指令可同时计算 4 个像素）
- Tile 大小的选取
  - 保持任务的粒度适中
  - 与 Cache 的大小匹配
  - 兼顾 SIMD 宽度
- 内存分配器
  - 根据对象的生存期配备若干组对象分配器
  - 每线程独享一组对象分配器

舞台机械控制系统中的并行算法

- 系统简介
  - 舞台机械驱动是由一系列开关、传感器、制动器、变频器、马达等部件组成的，通过一个 IO 设备来实现控制系统对机械驱动装置的控制
  - 控制系统和 IO 设备通过以太网通讯（ UDP 协议）
  - 控制系统以恒定的时钟间隔向 IO 设备发送若干组控制信号，同时接收 IO 设备返回的若干组反馈信号，根据传感器反馈的当前状态计算下一时钟周期的控制信号
  - 一个任务由一组驱动装置协同运行，驱动之间需要保持同步控制
  - 一个大型的舞台需要控制系统同时操纵 200 多个驱动装置
  - 时钟间隔尽可能的短，以提高控制精度（ >500Hz ）

系统架构图


ControlDesk
DriverController
Driver IO
Driver IO
Driver IO

- 挑战
  - 高实时系统（延迟 <1ms ）
  - 高并发 IO ， 每秒 200*500=100,000 次并发 IO
  - 任务级并行
  - 高计算负载

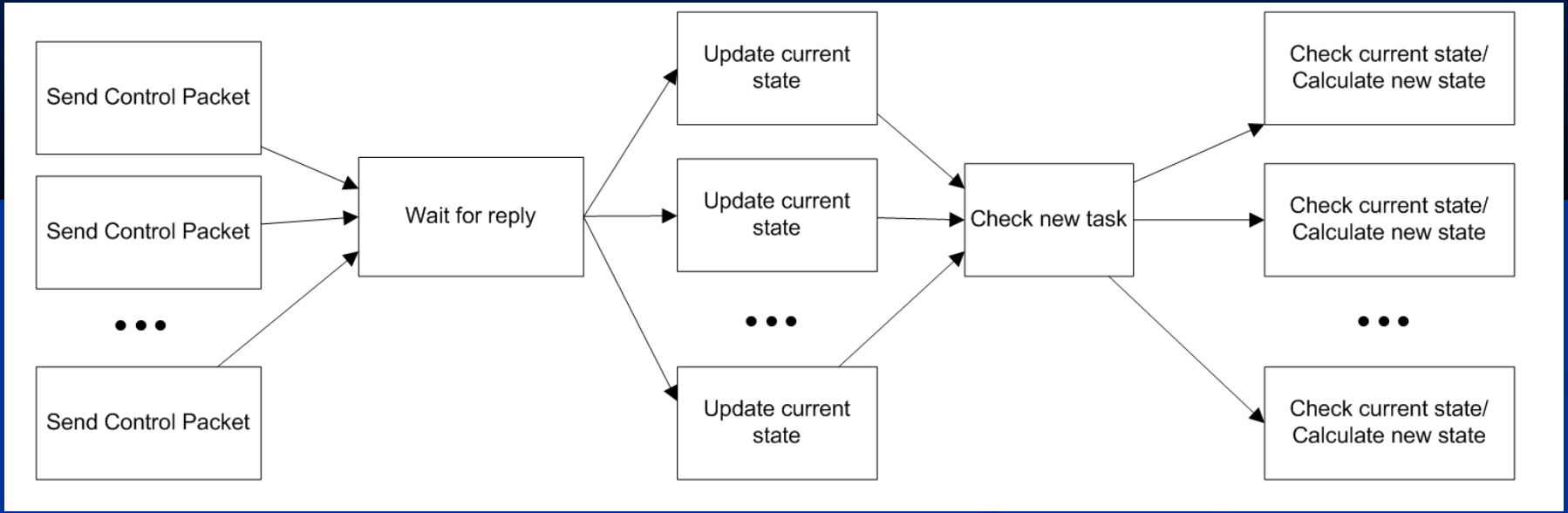
- 解决方案
  - 使用实时操作系统
  - 选用高端硬件提升系统整体的 IO 吞吐量
  - 用并行计算来缓解计算负载

任务级并行的实现

- 分析
  - 单个驱动的在一个循环内的任务流程图

- 解决方案的讨论
  - 流水线算法（不满足低延迟的要求，不可取）
  - 用一组线程池，每个线程处理一组 driver（不满足同步的需求）
  - 将任务流程划分为若干 Pass，对可并行处理的 Pass 进行并行计算

并行算法流程


Send Control Packet
Send Control Packet
Send Control Packet
Wait for reply
Update current state
Update current state
Update current state
Check new task
Check current state/ Calculate new state
Check current state/ Calculate new state
Check current state/ Calculate new state

- 伪代码实现

```
parallel for (driver in drivers)
   send control packet to driver
wait for 500us
parallel for (driver in drivers)
   receive data from driver
  update driver current state
Check new tasks
parallel for (driver in drivers)
   check current state
  calculate control packet
```

总结和展望

- 并行计算并不神秘
  - 用并行的思想来设计新的算法
  - 正确的任务分解是成功的一半
  - 耐心的调优
- 关注硬件的未来发展
  - Many-cores system
  - 异构系统
    - Cell
    - OpenCL/CUDA

提问