

15 性能优化（工作分配和调度）

- 高性能编程

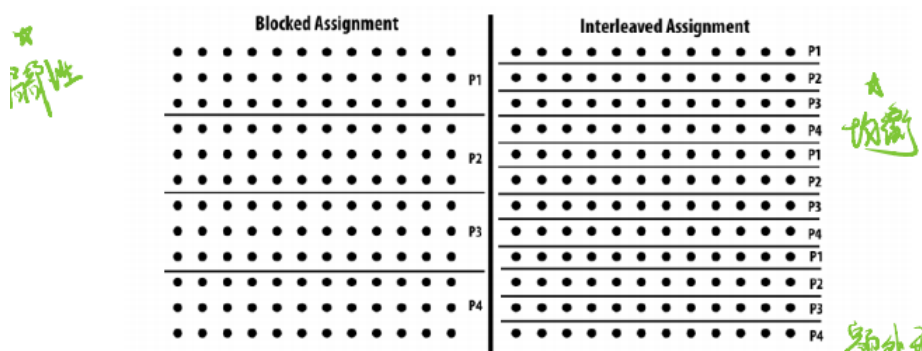
- 优化并行程序的性能是细化分解、分配和编排选择的迭代过程.....
- 关键目标（相互冲突的）
 - 将工作负载平衡到可用的执行资源上
 - 减少沟通（避免停顿）
 - 减少为增加并行性、管理分配、减少沟通等而执行的额外工作（开销）。
- 我们将谈论丰富的技术空间
- 提示 # 1：始终首先实施最简单的解决方案，然后测量性能以确定您是否需要做得更好。
- 如果您预计只运行低核数机器，则可能没有必要实施一种复杂的方法来创建数百或数千个独立工作

- 平衡工作负载

- 理想情况下：所有处理器在程序执行期间一直在计算（它们同时计算，并且同时完成它们的部分工作）

- 静态分派

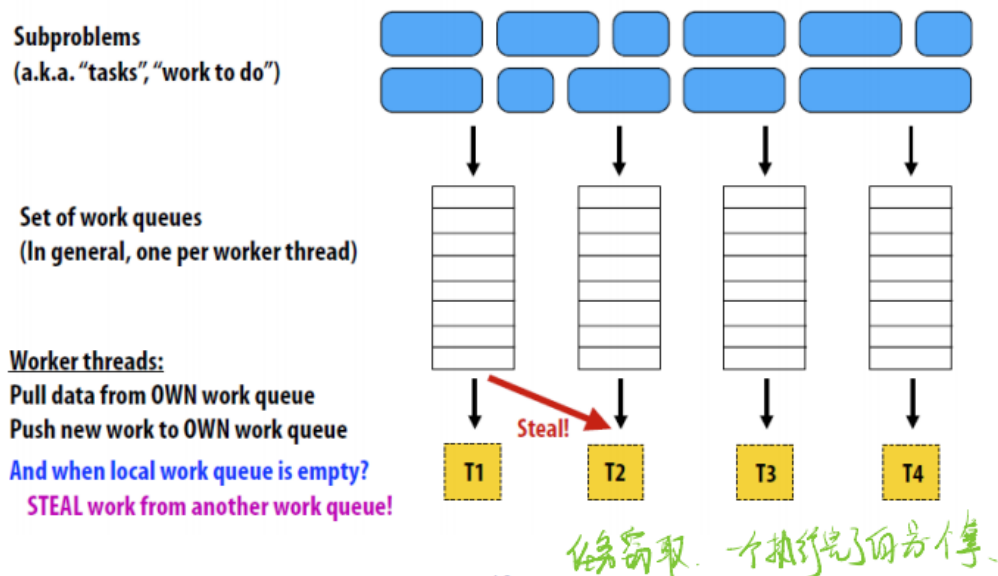
- 线程的工作分配是预先确定的
 - 不一定在编译时确定（分配算法可能取决于运行时参数，例如输入数据大小、线程数等）
- 回忆求解器示例：为每个线程（工作者）分配相等数量的网格单元（工作）
- 我们讨论了两种静态分配给工人的工作（阻塞和交错）



- 优点：简单，基本上零运行时开销（在这个例子中：实现赋值的额外工作是一点索引数学）
- 适用于
 - 当工作的成本（执行时间）和工作量是可以预测的（这样程序员可以提前制定出好的作业）最简单的例子：预先知道所有工作的成本相同
 - 当工作是可预测的，但并非所有工作都具有相同的成本
 - 当有关执行时间的统计信息已知时（例如，平均成本相同）

- 半静态分配（Semi-static）

- 近期的工作成本是可预测的
 - 思想：最近的过去很好的预测近期的未来
- 应用程序定期分析自身并重新调整分配
 - 对于重新调整之间的间隔，分配是“静态的”
- **动态分配**
 - 程序在运行时动态确定分配，以确保负载分布良好。（任务的执行时间，或者说任务的总数，是不可预测的。）
 - 使用工作队列
 - 单队列——瓶颈
 - 共享工作队列：待做工作的集合（假设工作之间独立）
 - 工作线程：从工作队列中取数据；将产生的新工作加入到工作队列中
 - **选择任务粒度**
 - 增大任务粒度——减小通信同步开销
 - 减小任务粒度——有利于负载均衡
 - 拥有比处理器更多的任务很有用（许多小任务通过动态分配实现良好的工作负载平衡）
 - 开始时的任务粒度小一点
 - 但希望尽可能少的任务，以尽量减少管理分配的开销
 - 合并成大粒度任务
 - 理想的粒度取决于许多因素（本课程的共同主题：必须了解您的工作量和您的机器）
- **工作调度**
 - 短作业优先——负载不均衡——“长尾现象”
 - 解决
 - 将任务分解为更大数量的小任务
 - 也许会增加同步开销
 - 可能没效果(如果长任务是连续的)
 - 先分派大任务执行——更聪明的调度
 - 执行大任务的线程相比其他线程可能执行的任务的数量更少.
 - 需要能预测任务的开销
 - 使用一组分布式队列减少同步开销。若本地队列为空——steal别的队列



- 在窃取过程中发生代价高昂的同步/通信
 - 但并非每次线程都进行新工作，仅在需要确保良好的负载平衡时才会进行窃取
- 导致局部性增加
 - 常见情况：线程处理它们创建的任务（生产者-消费者位置）
- 实现挑战
 - 从谁那里偷？
 - 偷多少钱？
 - 如何检测程序终止？
 - 确保本地队列访问快速（同时保持互斥）

• 总结

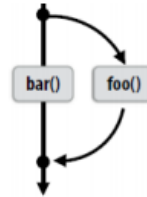
- 挑战：实现良好的工作负载平衡
 - 希望所有处理器一直工作（否则，资源处于空闲状态！）
 - 但想要实现这种平衡的低成本解决方案
 - 最小化计算开销（例如，调度/分配逻辑）
 - 最小化同步成本
- 静态分配与动态分配
 - 真的，这不是一个非此即彼的决定，而是一个连续（共存）的选择
 - 尽可能使用有关工作负载的前期知识，以减少负载不平衡和任务管理/同步成本（在极限情况下，如果系统知道一切，请使用完全静态分配）
- 今天讨论的问题涵盖分解、分配和编排

• 调度 fork-join 并行性

- 在分而治之的算法中表达独立工作的自然方式
- 本讲座的代码示例将在 Cilk Plus 中
 - C++ 语言扩展

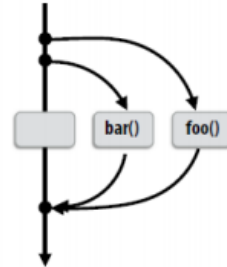
- 最初在麻省理工学院开发，被英特尔收购
- 但英特尔正在弃用它。最好坚持使用 MIT 版本
- **cilk_spawn** foo(args);
 - 语义：调用 foo，但与标准函数调用不同，调用者可以继续异步执行 foo。
- **cilk_sync**;
 - 语义：当当前函数产生的所有调用都完成时返回。（“sync up” with the spawned calls）
- 注意：在每个包含 cilk_spawn 的函数的末尾都有一个隐式的 cilk_sync（暗示：当 Cilk 函数返回时，与该函数相关的所有工作都已完成）

```
// foo() and bar() may run in parallel
cilk_spawn foo();
bar();
cilk_sync;
```

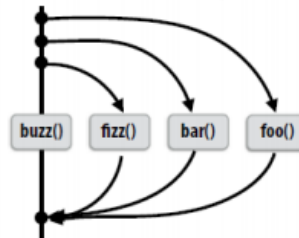


```
// foo() and bar() may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_sync;
```

Same amount of independent work first example, but potentially higher runtime overhead (due to two spawns vs. one)

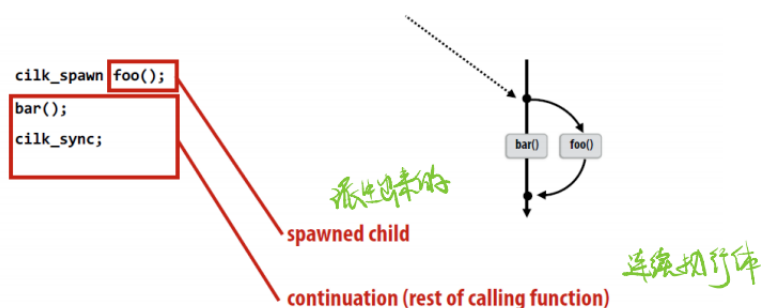


```
// foo, bar, fizz, buzz, may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_spawn fizz();
buzz();
cilk_sync;
```



- cilk_spawn 抽象不指定何时怎样调度执行——用户不可见，同时和调用者一起执行
- cilk_sync 有限制的调度：在他返回之前所有 spawned 调用必须完成
- 写程序
 - 主要思想：使用 cilk_spawn 向系统公开独立工作（潜在的并行性）
 - 回忆一下并行编程的经验法则
 - 至少需要与并行执行能力一样多的工作（例如，程序应该可能产生至少与内核一样多的工作）
 - 需要比执行能力更多的独立工作，以使所有工作在内核上实现良好的工作负载平衡
 - “parallel slack” = 独立工作与机器并行执行能力的比率（在实践中：~8 是一个很好的比率）
 - 但不要过多独立工作，以免工作粒度太小（过多的 slack 会导致管理精细的开销 - 细粒度的工作）
- 调度程序
 - 考虑非常简单的调度程序：

- 使用 `pthread_create` 为每个 `cilk_spawn` 启动 `pthread`
- 将 `cilk_sync` 转换为适当的 `pthread_join` 调用
- 潜在的性能问题？
 - `spawn`操作重载，代价高
 - 并发运行比内核多得多的线程
 - 上下文切换开销
 - 比需要更大的工作集，更少缓存局部性
- 工作线程池
 - Cilk Plus 运行时维护工作线程池
 - 思想：在应用程序启动时创建的所有线程 *
 - 与机器中的执行上下文一样多的工作线程
- spawned child & continuation



- 执行过程
 - 在到达`cilk_spawn foo()`之后，线程将`continuation`放到工作队列中，然后开始执行`foo()`
 - 此时若另一线程空闲，它可以`steal`，向忙碌线程请求工作，并将工作放到自己的工作队列中
 - 原空闲线程也开始执行
- 两种调度运行方式
 - run continuation first: 窃取的是spawned child ——“child stealing”
 - 先展开所有循环

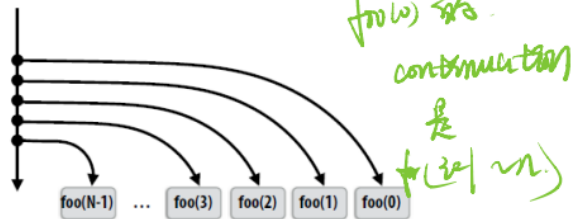
```

for (int i=0; i<N; i++) {
    cilk_spawn foo(i);
}
cilk_sync;

```

是先创建
并不断谁先

foo(i) : i < N : i++
foo(i)



■ Run **continuation first** ("child stealing")

- Caller thread spawns work for all iterations before executing any of it *所有任务不执行，不派生工作*
- Think: **breadth-first traversal of call graph**. $O(N)$ space for spawned work (maximum space)
- If no stealing, **execution order is very different than that of program with `cilk_spawn` removed**

⇒ 所以先展开循环

Thread 0 work queue



从 N-1 开始



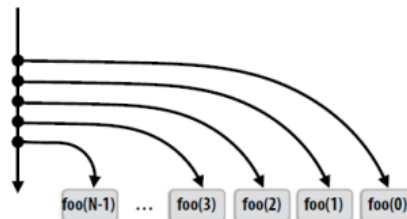
- run child first: 窃取的是continuation——"continuation stealing"

```

for (int i=0; i<N; i++) {
    cilk_spawn foo(i);
}
cilk_sync;

```

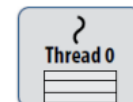
先执行后展开
在 stealing 的时候就是运行



■ Run **child first** ("continuation stealing")

- Caller thread only creates one item to steal (continuation that represents all remaining iterations)
- If no stealing occurs, thread continually pops continuation from work queue, enqueues new continuation (with updated value of `i`)
- **Order of execution is the same as for program with `spawn` removed.**
- Think: **depth-first traversal of call graph**

Thread 0 work queue



Executing foo(0)...

- stealing thread 执行下一个迭代


- 完善 work stealing

- 双端队列

- 工作队列实现为出队（双端队列）
 - 顶端存的都是大粒度的工作——先 steal 它，减少工作开销
- 本地线程从“尾部”（底部）推送/弹出
- 远程线程从“头部”窃取（顶部）
- 存在有效的无锁出队实现

- 随机选取

- 空闲线程随机选择一个线程来尝试窃取
- 从出队顶部窃取...
 - 减少与本地线程的争用：本地线程没有访问与窃取线程相同的出队部分！

- 在调用树开始时窃取工作：这是一项“更大”的工作，因此执行窃取的成本将在未来更长的计算中摊销（分摊）
- 最大化局部性：（结合 run-child-first 策略）本地线程在调用树的本地部分工作
- cilk_sync实现
 - 如果其他线程没有工作，sync的用处也体现不出来
 - stalling join
 - 如果有派生其他线程，哪一个线程初始化fork，哪一个线程调用sync。他要一直等所有线程执行完成。
 - block A的描述符被创建
 - 维护一个结构体，初始化如下，done表示完成的任务次数
- 
 - steal一次，spawn数动态增加一次
 - 谁最后将done更新到spawn，即该线程最后一个执行完工作，谁就告知初始化fork的线程，让他来返回cilk_sync
- greedy policy贪婪策略
 - 谁最后一次执行完，谁来负责cilk_sync
 - 所有线程无事做时都想要stealing
- 调整粒度
 - 细粒度，划分的任务数太多，spawn的调度开销太大
 - 粒度大，任务数量少，并行度不足
 - 需要程序员来决定

以上内容整理于 [幕布文档](#)