7 OpenMP的性能优化

- 大纲
  - 定义加速和效率：使用阿姆达尔定律预测最大加速
  - 并行程序性能优化技术
    - 经验法则
      - 从最佳串行算法开始
      - 最大化局部性
    - 调度
    - 循环变换
      - 循环裂变
      - 循环合并
      - 循环交换
- 加速比和效率
  - 加速比
    - 核心利用率的衡量标准
    - 加速比：Speedup = 串行执行时间/并行执行时间
  - **效率**
    - 衡量core利用率
    - 加速比除以core数
  - Amdahl’s Law之后：程序串行部分不变，随着核心数增加，效率降低
    - 过于乐观
      - 阿姆达尔定律忽略了并行处理开销
        - 这种开销的示例包括创建和终止线程所花费的时间
        - 并行处理开销通常是内核（线程）数量的递增函数
      - 阿姆达尔定律假设计算在核心之间平均分配
        - 实际上，工作量在核心之间分配不均
        - 核心等待时间是另一种形式的开销
  - **更通用的加速公式**

n	problem size
p	number of cores
$\psi(n,p)$	Speedup for problem (of size n on p cores)
$\sigma(n)$	Time spent in sequential portion of code
$\varphi(n)$	Time spent in parallel portion of code
$\kappa(n,p)$	Parallel overhead

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

    - n趋于无穷时，加速比趋向于p
- 并行程序性能优化技术
    - 经验法则
        - 从最佳串行算法开始、
            - 不要将“加速”与“速度”混淆
            - 加速比：程序在 1 个核上的执行时间与其在 p 个核上的执行时间之比
            - 如果从劣质顺序算法开始怎么办？——有可能最佳串行比并行之后还快
        - 最大化局部性
            - 时间局部性：如果处理器访问一个内存位置，它很有可能很快会重新访问该内存位置
            - 数据局部性：如果处理器访问内存位置，它很有可能很快会访问附近的位置
            - 程序倾向于表现出局部性，因为它们倾向于通过数组进行循环索引
            - 局部性原则使高速缓存有效
            - 并行处理和局部性
                - 多核⇒多个缓存
                - 当一个core写入一个值时，系统必须确保没有core试图引用一个过时的值（缓存一致性问题）
                - 一个核心的写入会导致另一个核心的缓存行副本失效，从而导致缓存未命中
                - 经验法则：最好让不同的内核处理完全不同的数组块
                - 如果内核的内存写入往往不会干扰其他内核正在完成的工作，我们就说并行程序具有良好的局部性
    - **循环调度**
        - 如何将循环迭代分配给线程
        - 静态调度：在执行循环之前分配给线程的迭代
        - 动态调度：在循环执行期间分配给线程的迭代

- OpenMP 调度子句 schedule 影响循环迭代映射到线程的方式
- schedule(static [, chunk])
  - 大小为“chunk”的迭代块到线程
  - 轮询分配的
  - 低开销，可能导致负载不平衡
  - 最适合用于可预测和类似的工作迭代
- schedule(dynamic [, chunk])
  - 线程抓取“块”迭代
  - 完成迭代后，线程请求下一组
  - 更高的线程开销，可以减少负载不平衡
  - 最适用于不可预测或高度可变的工作
- schedule(guided[, chunk])
  - 从大块开始的动态调度
  - 块的大小缩小； 不小于“块”
  - 初始块=number_of_iterations / number_of_threads
  - 后续块=number_of_iterations_remaining / number_of_threads
  - 当计算变得越来越耗时时，最好被作为动态的特殊情况以减少调度开销

- **循环转换**
  - 循环裂变**fission**
    - 从具有循环携带依赖的单循环开始，将循环拆分为两个或多个循环，新循环可以并行执行
    - 例子

```
float *a, *b;
for (int i = 1; i < N; i++) {
    // perfectly parallel
    if (b[i] > 0.0) a[i] = 2.0;
    else a[i] = 2.0 * fabs(b[i]);
    // loop-carried dependence
    b[i] = a[i-1];
}
```

```
float *a, *b;
#pragma omp parallel
{
#pragma omp for
    for (int i = 1; i < N; i++) {
        if (b[i] > 0.0) a[i] = 2.0;
        else a[i] = 2.0 * fabs(b[i]);
    }
#pragma omp for
    for (int i = 1; i < N; i++) b[i] = a[i-1];
}
```

- 循环合并**fusion**
  - 循环裂变的反义词，合并循环增加粒度尺寸
  - 例子

```
for (int i = 0; i < N; i++) a[i] = foo(i);
x = a[N-1] – a[0];
for (int i = 0; i < N; i++) b[i] = bar(a[i]);
y = x * b[0] / b[N-1];
```

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    a[i] = foo(i);
    b[i] = bar(a[i]);
}
x = a[N-1] – a[0];
y = x * b[0] / b[N-1];
```

  - 复制工作
    - 每个线程迭代都有成本，例如，barrier同步
    - 有时线程复制工作比通过barrier同步更快
    - 例子1：对下面的代码并行化有两种可能

```
for (int i = 0; i < N; i++) a[i] = foo(i);
x = a[0] / a[N-1];
for (int i = 0; i < N; i++) b[i] = x * a[i];
```

- 使用隐式屏障

```
#pragma omp parallel
{
#pragma omp for
    for (int i = 0; i < N; i++) a[i] = foo(i);
#pragma omp single
    x = a[0] / a[N-1]; // implicit barrier
#pragma omp for
    for (int i = 0; i < N; i++) b[i] = x * a[i];
}
```

- 使用复制工作

```
#pragma omp parallel private (x)
{
    x = foo(0) / foo(N-1);
#pragma omp for
    for (int i = 0; i < N; i++) {
        a[i] = foo(i);
        b[i] = x * a[i];
    }
}
```

- 例子2：外层是质数次迭代，内层并行化不足以抵消并行开销

```
#define N 23
#define M 1000
…
for (int k = 0; k < N; k++)
    for (int j = 0; j < M; j++)
        w_new[k][j] = DoSomeWork(w[k][j], k, j);
```

- 合并，扩大迭代次数

```
for (int kj = 0; kj < N*M; kj++) {
    k = kj / M;
    j = kj % M;
    w_new[k][j] = DoSomeWork(w[k][j], k, j);
}
```

- 循环交换**exchange**
  - 嵌套的 for 循环可能具有阻止并行化的数据依赖性
  - 交换 for 循环的嵌套可能
    - 公开一个可并行化的循环
    - 增加粒度
    - 提高并行程序的局部性

以上内容整理于 幕布文档