5 OpenMP 并行编程模型

- OpenMP概览
  - 共享内存模型架构PPT5有图
    - 统一内存访问
    - 非同一内存访问（一组CPU共享内存）
  - 线程基于并行度
    - OpenMP 程序仅通过使用线程来实现并行性
    - 线程执行是操作系统可以调度的最小处理单元
    - 线程存在于单个进程的资源中，没有这个进程，它们将不复存在
    - 通常，线程的数量与机器处理器/内核的数量相匹配。但是，线程的实际使用取决于应用程序
  - 显式并行度
    - OpenMP 是一种显式（非自动）编程模型，为程序员提供对并行化的完全控制
    - 并行化可以像使用串行程序并插入编译器指令一样简单.....
    - 或者像插入子程序来设置多级并行、锁甚至嵌套锁一样复杂
  - 什么是**OpenMP**
    - Open Multi-Processing的缩写
    - 通过来自硬件和软件行业、政府和学术界的相关方之间的协作，开放多处理规范
    - OpenMP 是一种显式（非自动）编程模型，为程序员提供对并行化的完全控制
    - 提供针对共享式内存并行编程的 API，简化了并行编程在 fortran、C、C++等环境下
  - 线程之间交流
    - OpenMP 是一种多线程、共享地址模型，线程通过共享变量进行通信
    - 意外共享数据导致竞争条件
      - 竞争条件：当程序的结果因线程调度不同而发生变化时
    - 控制竞争条件：使用同步来保护数据冲突
    - 同步很昂贵：更改访问数据的方式以最大程度地减少同步需求
  - fork-join 模型
    - fork：创建线程，主线程创建（或唤醒）一组并行线程。
    - join：当团队线程完成并行区域构造中的语句时，它们会同步并终止，只留下主线程
  - 三个 API 组件
    - 编译器指令
    - 运行时库例程
    - 环境变量

- 编译器指令的语法
  - #pragma：一个 C/C++ 编译器指令
    - 代表“附注信息”
    - 程序员与编译器通信的一种方式
    - 编译指示由预处理器处理
    - 编译器可以随意忽略编译指示
  - 所有 OpenMP 编译指示都具有以下语法：

    ```
    #pragma omp <directive-name> [clause, ...]
    ```

  - 编译指示出现在相关构造（并行控制语句）之前
- **常用指令，执行过程**
  - #pragma omp parallel // 表明之后的结构化代码块被多个线程处理
  - #pragma omp parallel num_threads(thread_count) // 可自定义线程数量
  - #pragma omp critical // 只有一个线程能够执行对应代码块,并且第一个线程完成操作前，没有其他的线程能够开始执行这段代码
  - #pragma omp parallel for // parallel for 指令生成一组线程来执行后面的结构化代码块（必须是for循环）。
    - 循环迭代的次数必须在循环之前的运行时可以被计算
    - 不能并行化 while 等其他循环。goto
    - 循环变量必须在循环开始执行前就已经明确，不能为无限循环。
    - 不能存在其他循环出口，即不能存在 break、exit、return 等中途跳出循环的语句。
      - 在程序执行到该指令时，系统分配一定数量的线程，每个线程拥有自己的循环变量，互不影响，即使在代码中循环变量被声明为共享变量，在该指令中，编译过程仍然会为每一个线程创建一个私有副本这样防止循环变量出现数据依赖，跟在这条指令之后的代码块的末尾存在**隐式路障。**
- 常用函数
  - omp_get_num_procs：返回物理处理器可用的并行程序数
  - omp_set_num_threads(int t)：设置并行块上可用的线程数
  - omp_get_thread_num：返回线程号
  - omp_get_num_threads：返回线程数
- PPT23有一个判断循环哪个for，第一个有数据依赖，第三个线程开销大
  - 最小化线程开销
    - 由于 fork/join 是开销的来源，我们希望最大化每个 fork/join 完成的工作量；即粒度
    - 因此我们选择使中间循环并行
- private子句（子句——编译指示的可选附加组件）

    - 子句可以解决私有变量的问题，一些共享变量在循环中，如果每个线程都可以访问的话，可能会出错，该子句就是为每个线程创建一个共享变量的私有副本解决循环依赖。注 private 的括号中可以放置多个变量，逗号分隔
    - 更多私有变量
        - 每个线程都有自己的私有变量副本
        - 如果 j 被声明为私有，则在 for 循环内没有线程可以访问“其他”j（共享内存中的 j）
        - 没有线程可以使用先前定义的 j 的值
        - 没有线程可以为共享的 j 分配新值
        - 私有变量在循环进入和循环退出时未定义（要在开始时定义复制，否则就是随机一个值），减少了执行时间
- **firstprivate子句**——firstprivate(x)
    - firstprivate 子句告诉编译器私有变量应该在**循环进入时继承共享变量的值**，并行化代码结束之后还是原来共享变量的值，没有改变
    - 每个线程分配一次值，而不是每次循环迭代一次
    - x是一个基本数据类型——私有 x 直接从共享 x 复制而来
    - x是一个数组——将 sizeof(x) 的数据复制到私有内存中
    - x是一个指针——私有 x 指向与共享 x 相同的位置
    - x是一个类实例——调用拷贝构造函数来创建私有 x
- **lastprivate子句**
    - lastprivate 子句告诉编译器在循环退出时应将最后一次循环迭代后的私有变量的值分配给共享变量（进入线程时**赋值为0**，是所有循环的最后一次）
    - 换句话说，当负责顺序最后一个循环迭代的线程退出循环时，它的私有变量副本被复制回共享变量
- parallel子句
    - 为了增加粒度，有时应该并行执行的代码超出了单个 for 循环
    - 当一个代码块应该并行执行时，使用parallel pragma
        - SPMD 风格的编程
        - 单程序，多数据
    - for子句
        - 需要在已经被标记parrallel的代码块中使用
        - 将迭代分发给有效线程
        - 最后有隐式barrier
- single子句
    - single编译指示在并行代码块中使用
    - 它告诉编译器只有一个线程应该执行紧随其后的语句或代码块
    - 在处理非线程安全的代码部分（例如 I/O）时可能很有用

    - 团队中不执行单个指令的线程在封闭代码块的末尾等待，除非指定了 nowait 子句。
- master
    - 该子句仅有主线程执行后面的代码块，其他线程直接跳过该子句后面的代码块，且不存在隐式 barriers
    - master == single nowait
- sections 和 section
    - sections指定封闭的代码部分将在团队中的线程之间划分。
    - 独立的section指令嵌套在sections指令中。
    - 每个section由团队中的一个线程执行一次。
    - 不同的section可以由不同的线程执行。
    - 如果线程足够快并且实现允许，则线程可以执行多个部分。
    - parallel sections——各个section并行处理，只有sections的时候是串行处理的
- reduction子句——归并

    #pragma omp … reduction (op : list)

    - opreation +、-、*、&、|、&&、||、^ 将每个线程中的变量进行规约操作
    - 根据操作创建和初始化每个list变量的私有副本。
    - 这些副本由线程在本地更新
    - 在构造结束时，本地副本通过 op 组合成单个值，并组合原始共享变量中的值。
    - 共享变量初始值

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

- OpenMP的优势和劣势
    - 优势
        - 增量并行化和串行等价
        - 非常适合数据分解
        - 在 *nix 和 Windows 上可用
    - 劣势
        - 不适用于功能分解
        - 编译器不检查死锁和竞争条件等错误

以上内容整理于 幕布文档