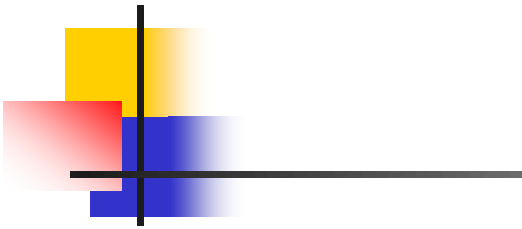
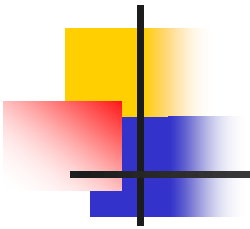
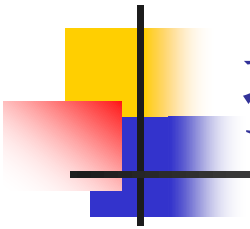


第5讲 OpenMP编程



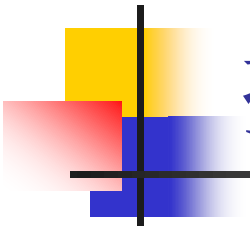
阅读内容

- 第五章 OpenMP共享内存编程
  - 5.1 开始准备
  - 5.2 梯度法
  - 5.3 作用域
  - 5.4 归约
  - 5.5 parallel for
  - 5.7 循环调度
  - 5.8 生产者和消费者



提纲

- OpenMP并行模型
- 并行循环
- 数据依赖
- 循环调度
- 局部性
- 任务并行



提纲

- OpenMP并行模型
- 并行循环
- 数据依赖
- 循环调度
- 局部性
- 任务并行

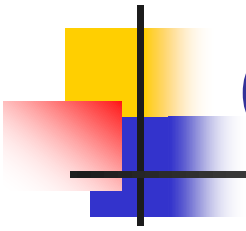
OpenMP编程模型

- 是Pthread的常见替代，更简单，但限制也更多
  - 通过少量编译指示指出并行部分和数据共享，即可实现很多串行程序的并行化
- 可移植：不同共享内存架构
- 可扩展
- 增量并行化：程序不同部分逐步并行化
- 依赖编译器生成线程创建和管理代码
  - 语言扩展：C、C++、Fortran
    主要是编译指示、少量库函数

官网 http://www.openmp.org

OpenMP：程序员视角

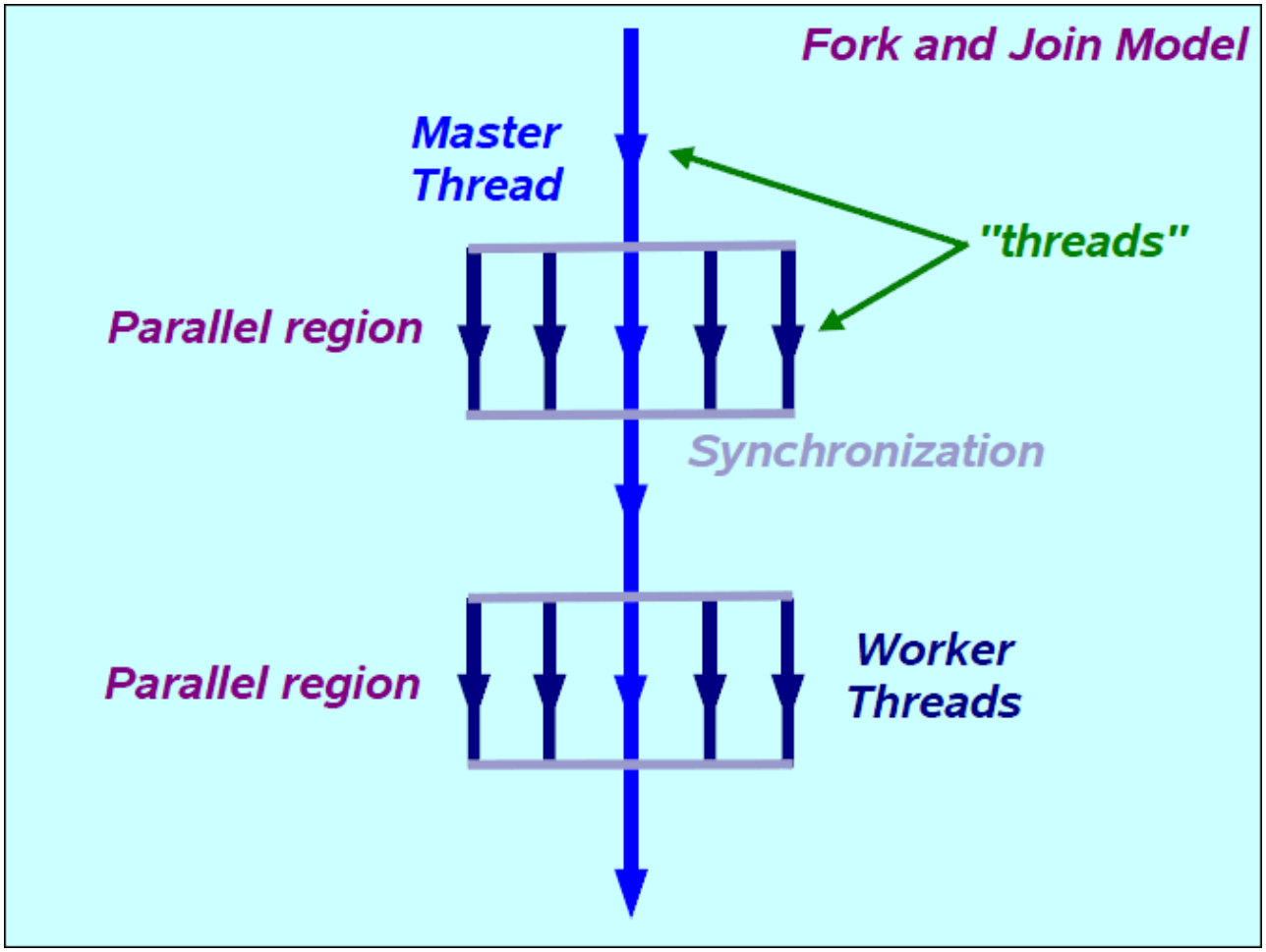
- 一种可移植共享内存多线程编程规范，“轻量”语法
  - 准确行为依赖于具体实现
  - 需要编译器支持（C、C++、Fortran）
- OpenMP能
  - 程序员只需将程序分为串行和并行区域，而不是构建并发执行的多线程
  - 隐藏栈管理
  - 提供同步机制
- OpenMP不能
  - 自动并行化
  - 确保加速
  - 避免数据竞争

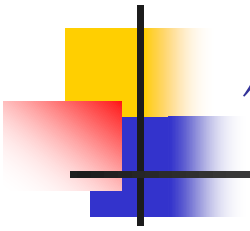


OpenMP执行模型

- Fork-join并行执行模型
- 执行伊始是单进程（主线程）
- 并行结构开始
  - 主线程创建一组线程（工作线程）
- 并行结构结束
  - 线程组同步——隐式barrier
- 只有主线程继续执行
- 实现优化
  - 工作线程等待下一次fork

OpenMP执行模型（续）





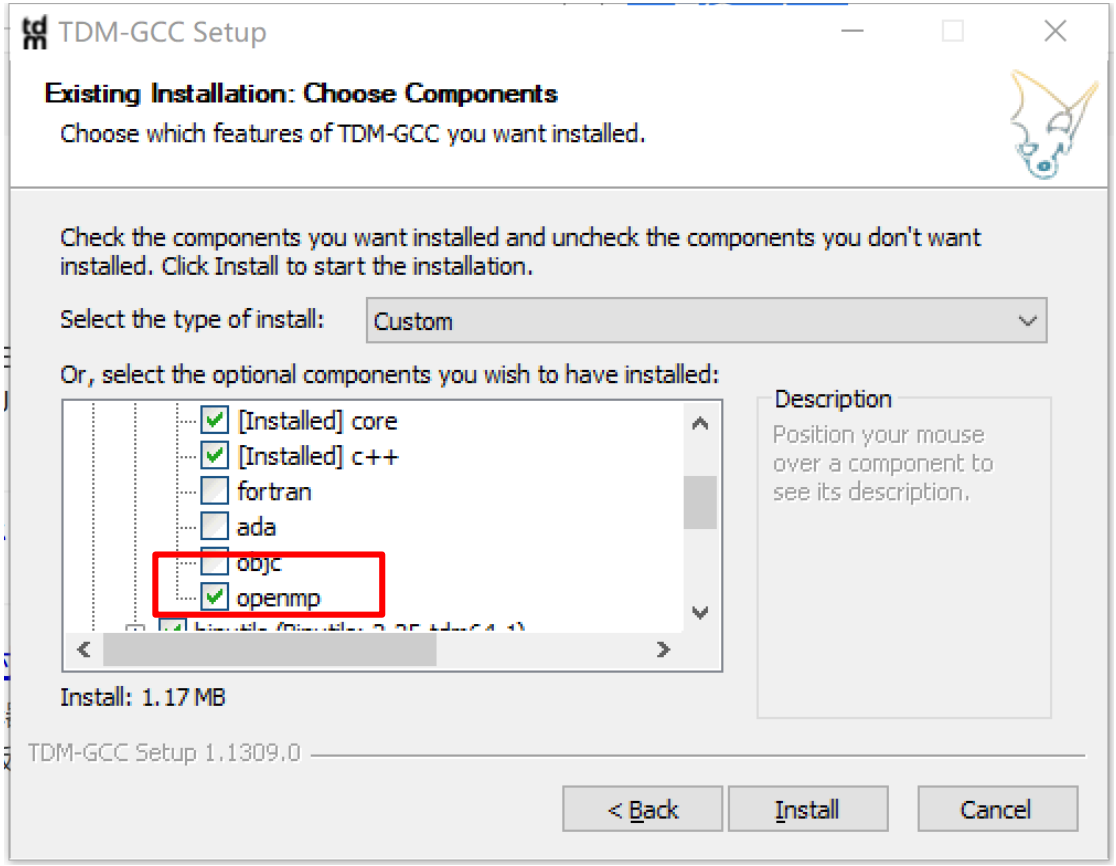
使用编译指示

- 编译指示（pragma）是一些特殊的预处理指令
- 为了提供标准C规范之外的功能
- 编译器会忽略不支持的编译指示
- OpenMP编译指示的解释
  - 修改紧跟其后的语句——可能是循环这样的复合语句

#pragma omp ...

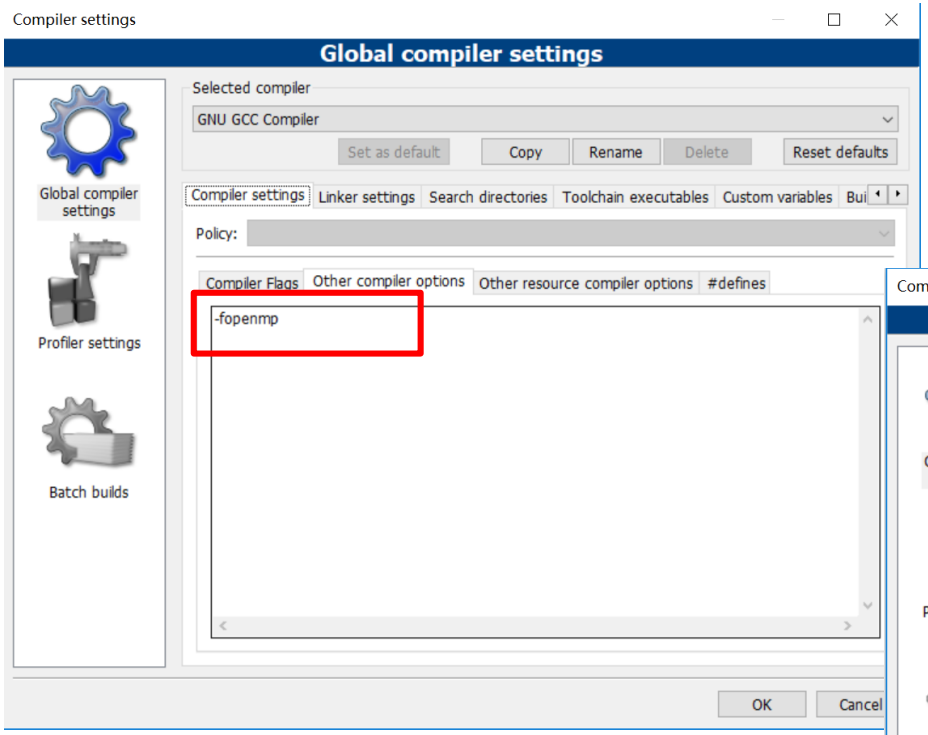
Code::Blocks+TDM-GCC环境配置

- 一、 TDM-GCC安装需要勾选openmp

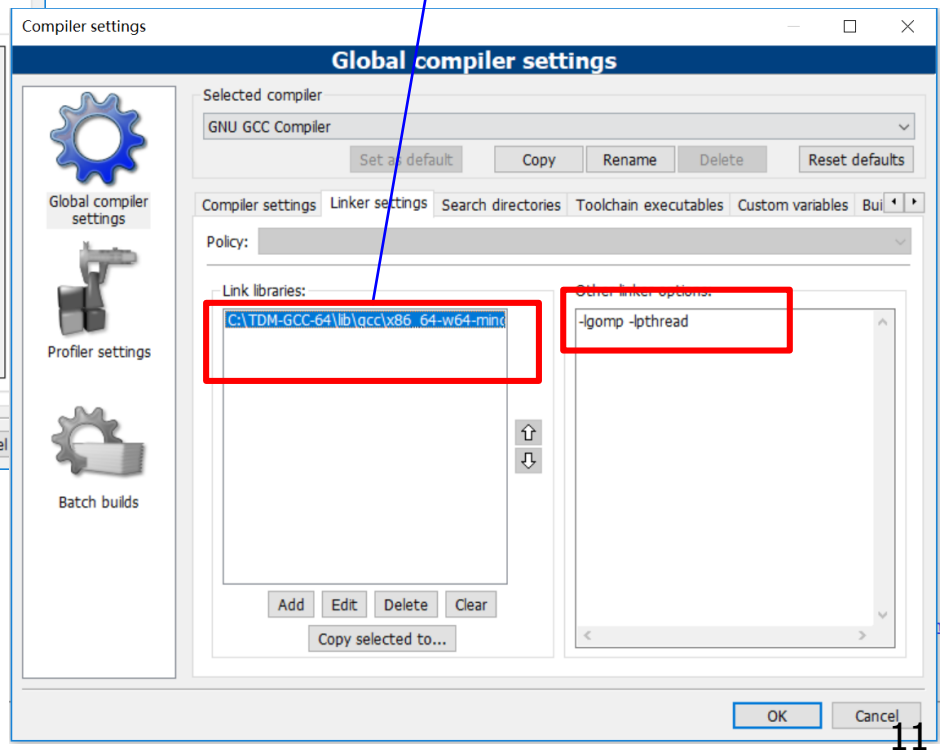


Code::Blocks+TDM-GCC环境配置

- 二、 Code ::Blocks编译器设置



C:\TDM-GCC-64\lib\gcc\x86_64-w64-mingw32\5.1.0\libgomp.dll.a



编程模型 – 数据共享

- 并行程序通常使用两种数据
  - 共享数据，所有线程可见，通常是具名的
  - 私有数据，单线程可见，通常在栈中分配
- PThread
  - 全局作用域变量是共享的
  - 栈中分配的变量是私有的
- OpenMP
  - **shared**变量是共享的
  - **private**变量是私有的
  - 默认是**shared**
  - 循环变量是**private**

```
// 共享全局变量
int bigdata[1024];


void* foo(void* bar) {
  int tid;


  #pragma omp parallel \
  shared ( bigdata ) \
  private ( tid )
  {
    /* Calc. here */
  }
}
```

编译指示格式

- 编译指示格式

```
#pragma omp directive_name [ clause [ clause ] ... ]
```

- 条件编译

```
#ifdef _OPENMP
    …
  printf("%d avail.processors\n",omp_get_num_procs());
#endif
```

- 大小写敏感
- 使用库函数需要包含头文件

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

查询函数

```
int omp_get_num_threads(void);
```

- 返回执行当前并行区域的线程组中的线程数

```
int omp_get_thread_num(void);
```

- 返回当前线程在线程组中的编号，值在0和omp_get_num_threads()-1之间。主线程的编号为0

并行区域结构

- 多线程并行执行的代码块
- 每个线程执行相同的代码（SPMD）
  - 线程组中线程分担工作，任务被分配给它们
- C/C++语法示例

```
#pragma omp directive_name [ clause [ clause ] ...
  ]
  语句块
```

- 子句可为下面情况

```
private (list)
shared (list)
```

Hello World

- 我们从一个简单的并行区域结构开始
- 需要考虑的问题
  - 从命令行读取线程数
  - 没有pragma和库调用的代码应该是正确的
- 与Pthread代码的区别
  - 更多必要的代码是由编译器和运行时环境管理的
  - 有隐含的线程标识

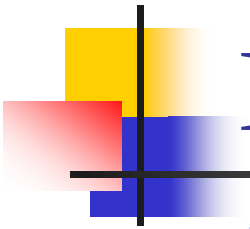
gcc -fopenmp …

Hello World

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
  /* Get number of threads from command line */
  int thread_count = strtol(argv[1], NULL, 10);

# pragma omp parallel num_threads(thread_count)
  Hello();

  return 0;
}  /* main */
```

Hello World(2)

```
void Hello(void) {
  int my_rank = omp_get_thread_num();
  int thread_count = omp_get_num_threads();

  printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

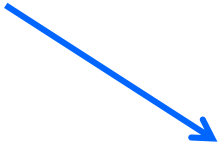


Hello World(3)

Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4

为防止编译器不支持OpenMP

```
# include <omp.h>
```

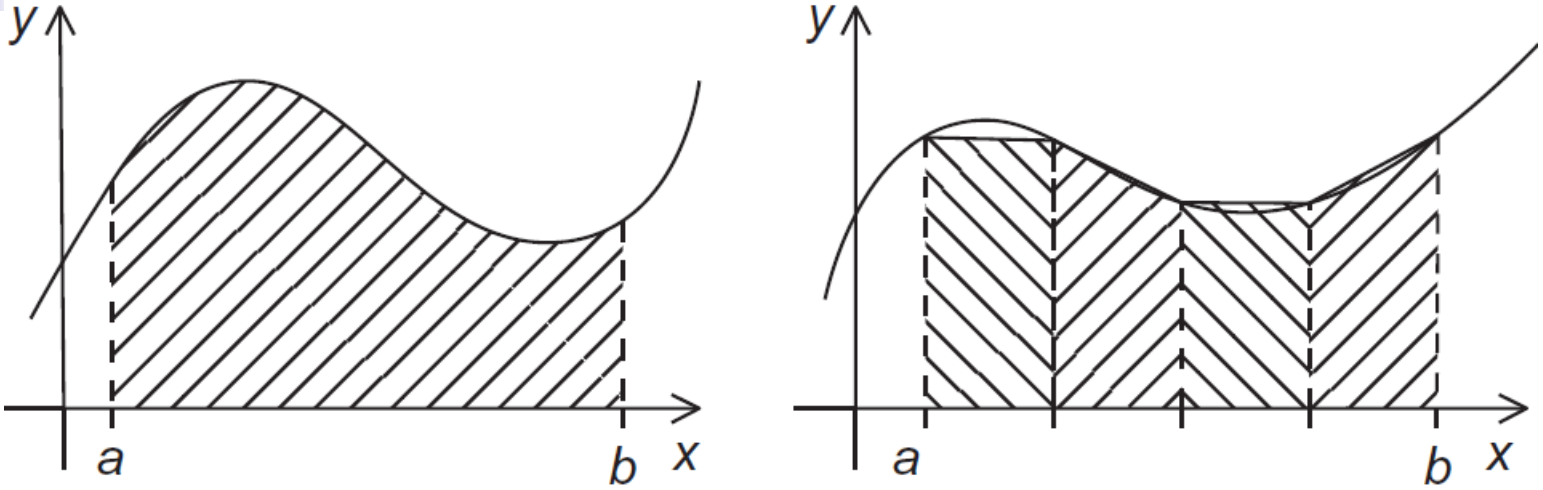


```
#ifdef _OPENMP
# include <omp.h>
#endif
```

为防止编译器不支持OpenMP

```
# ifdef _OPENMP
   int my_rank = omp_get_thread_num ( );
   int thread_count = omp_get_num_threads ( );
# else
   int my_rank = 0;
   int thread_count = 1;
# endif
```

梯形积分法：函数f(x) [a,b]间积分



- 间隔$h=x_{i+1}-x_i=(b-a)/n$
- 每个梯形面积 $\frac{h}{2}[f(x_i)+f(x_{i+1})]$

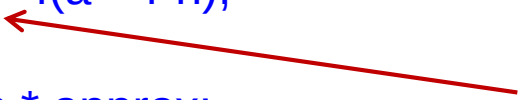
$$x_0 = a,\ x_1 = a+h,\ x_2 = a+2h, \ldots,\ x_{n-1} = a+(n-1)h,\ x_n = b$$

- 梯形面积之和

$$h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

串行版本

```
/* Input: a, b, n */

h = (b - a)/n;
approx = (f(a) + f(b))/2.0;

for (i = 1; i <= n-1; i++)
  approx += f(a + i*h);

approx = h * approx;
```



多线程求全局和
怎么保证正确性？

临界区指令

- 被临界区包围的代码
  - 所有线程都执行，但
  - 每个时刻限制只有一个线程执行

```
#pragma omp critical [ ( name ) ]
  语句块
```

- 线程在临界区开始位置等待，直至组中没有其他线程在同名临界区中执行
- 所有未命名临界区指示都映射到相同的未指定名字

第一个OpenMP版本

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Trap( double a, double b, int n, double* global_result_p);
int main(int argc, char* argv[]) {
  double global_result = 0.0;
  double a, b;
  int n;
  int thread_count ;

  thread_count = strtol(argv[1], NULL, 10) ;
  printf("Enter a, b, and n\n");
  scanf("%lf %lf %d", &a, &b, &n);
# pragma omp parallel num_threads(thread_count)
  Trap(a, b, n, &global_result);
  printf("With n = %d trapezoids, our estimate\n", n);
  printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
  return 0;
} /* main*/
```

第一个OpenMP版本(2)

```
void Trap( double a, double b, int n, double* global_result_p ) {
  double h, x, my_result ;
  double local_a , local_b ;
  int i, local_n ;
  int my_rank = omp_get_thread_num();
  int thread_count = omp_get_num_threads();
  h = (b-a)/n;
  local_n = n/thread_count ;
  local_a = a + my_rank*local_n*h;
  local_b = local_a + local_n*h;
  my_result = (f(local_a) + f(local_b))/2.0;
  for (i = 1; i <= local_n 1; i++) {
    x = local_a + i*h;
    my_result += f(x);
  }
  my_result = my_result*h;
# pragma omp critical
  *global_result_p += my_result ;
} /* Trap*/
```

先局部求和
避免过多通信

用临界区保证
全局求和正确性

改得更简洁

```
 global_result = 0.0;
 …
# pragma omp parallel num_threads(thread_count)
 Trap(a, b, n, &global_result);
 …
```

有什么问题？

```
  global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
  {
#   pragma omp critical
    global_result += Local_trap(a, b, n);
  }
```

不同线程的全部计算被串行化了

避免所有计算串行化

```
 global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
 {
#   pragma omp critical
    global_result += Local_trap(a, b, n);
 }
```

```
 global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
 {
    double my_result = 0.0; /* private */
    my_result += Local_trap(a, b, n);
#   pragma omp critical
    global_result += my_result;
 }
```

不同线程并行计算

只有全局求和串行

OpenMP归约

- OpenMP支持归约操作
  - 归约就是将相同的规约操作符重复地应用到操作数序列来得到一个结果的计算。

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 100; i++)      {
   sum += array[i];
}
```

- 支持的归约运算及初值：

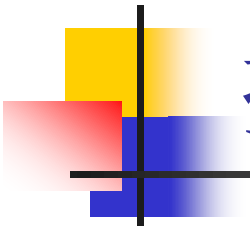
+	0	位与 &	~0	逻辑与 &	1
-	0	位或 |	0	逻辑或 |	0
*	1	位异或 ^	0		

改为使用归约

```
  global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
  {
     double my_result = 0.0; /* private */
     my_result += Local_trap(a, b, n);
#    pragma omp critical
     global_result += my_result;
  }
```

```
  global_result = 0.0;
# pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
  global_result += Local_trap(a, b, n);
```

私有变量等繁琐工作编译器代劳
全局求和采用递归算法



提纲

- OpenMP并行模型
- 并行循环
- 数据依赖
- 循环调度
- 局部性
- 任务并行

OpenMP数据并行：并行循环

- 所有编译指示都以#pragma开始
- 编译器为每个线程计算负责的循环范围——根据串行源码（计算分解）
- 编译器还管理数据划分
- 同步也是自动的（barrier）

Serial Program:

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

Parallel Program:

```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

局限和语义

```
                                                  index++
                                                  ++index
                          index < end             index--
                          index <= end            --index
for ( index = start   ;   index >= end    ;     index += incr
                          index > end             index -= incr
                                                  index = index + incr
                                                  index = incr + index
                                                  index = index - incr )
```

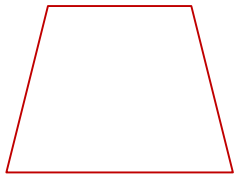
- 并非所有“元素级”循环都能并行化

```
#pragma omp parallel for
for (i=0; i < numPixels; i++) {}
```

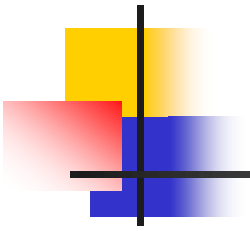
  - 循环变量：带符号整数
  - 终止检测：<, <=, >, >=与**循环不变量**
  - 每步迭代递增/递减一个循环不变量
  - 对<, <=向上计数；对>, >=向下计数
  - 循环体：无进/出控制流
- 创建线程，在线程间分配循环步；**要求迭代次数可预测。**
  - 不检查依赖性！
  - 不支持while 、do-while 、循环体包含break等

简单并行化循环的版本

```
/* Input: a, b, n */
h = (b - a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
  approx += f(a + i*h);
approx = h * approx;
```



```
/* Input: a, b, n */
h = (b - a)/n;
approx = (f(a) + f(b))/2.0;
#pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
for (i = 1; i <= n-1; i++)
  approx += f(a + i*h);
approx = h * approx;
```



同步

- 隐式barrier
  - 并行结构开始和结束位置
  - 其他控制结构的结束位置
  - 用nowait子句可去除隐式同步
- 显式同步
  - critical
  - atomic(单一语句)
  - barrier

并行for指示的各种形式

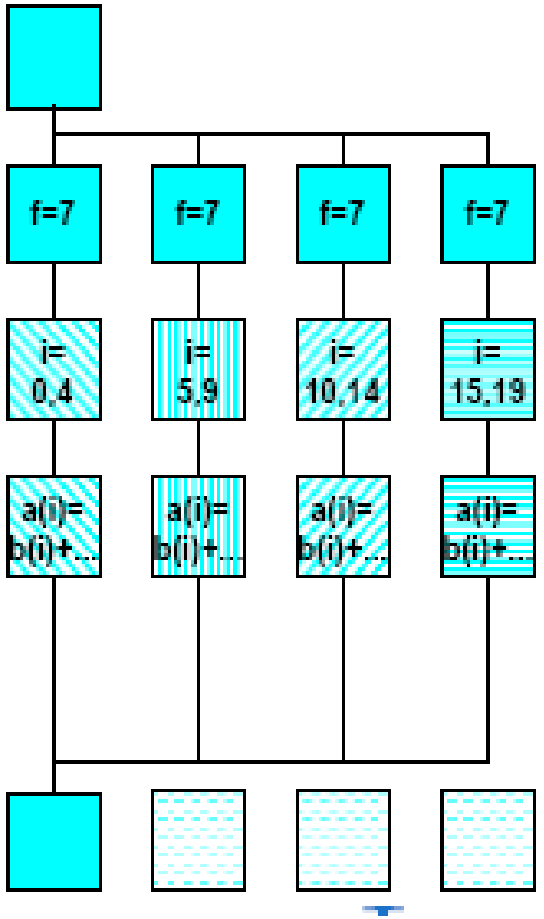
- 语法

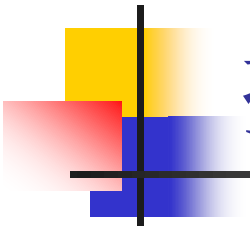
#pragma omp for [clause [clause] ...]
    for循环

- 子句形式

  shared(list)
  private(list)
  reduction(operator:list)
  schedule(type[, chunk])
  nowait（C/C++：用于#pragma omp for）

```
#pragma omp parallel private(f) {
        f=7;
#pragma omp for
        for (i=0; i<20; i++)
          a[i] = b[i] + f * (i+1);
} /* omp end parallel */
```





提纲

- OpenMP并行模型
- 并行循环
- 数据依赖
- 循环调度
- 局部性
- 任务并行

竞争条件与数据依赖

- 执行结果依赖于两个或更多事件的**时序**，则存在**竞争条件**（race condition）
- **数据依赖**（data dependence）就是两个内存操作的序，为了保证结果的正确性，必须保持这个序
  - 如果两个内存访问指向相同的内存位置且其中一个是写操作，则它们产生数据依赖

一些定义、定理（Allen & Kennedy）

- 一些定义
  - 两个计算等价 ← 在相同的输入上
    - 它们产生相同的输出
    - 输出按相同的顺序生成
  - 一个重排转换
    - 改变语句执行的顺序
    - 不增加或删除任何语句的执行
  - 一个重排转换保持依赖关系←
    - 它保持了依赖源和目的语句的相对执行顺序
- 依赖关系基本定理
  - 任何重排转换，只要保持了程序中所有依赖关系，它就保持了程序的含义。

考虑两种重排转换

- 并行化
  - 同步点之间并行执行的计算可能重排执行顺序。这种重排是否安全？根据我们的定义，如果它能保持代码中的依赖关系，则它是安全的
- 局部性优化
  - 假定我们希望修改访问顺序，以便更好地利用cache。这也是一种重排转换，同样，若它保持了代码中的依赖关系，则它是安全的
- 归约计算
  - 对于使用满足交换律和结合律的运算的归约操作，对其重排是安全的

数组的数据依赖

```
for (i=2; i<5; i++)
      A[i] = A[i-2]+1;
```

循环进位依赖关系

```
for (i=1; i<=3; i++)
      A[i] = i+1;
      B[i] = A[i]*3
```

循环独立关系

- 识别并行循环（直观地）
  - 寻找循环中的数据依赖
  - 没有依赖关系跨越迭代步边界→并行化是安全的

估算π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
  sum += factor/(2*k+1);
  factor = -factor;
}
pi_approx = 4.0*sum;
```

尝试第一个OpenMP版本

```
double factor = 1.0;
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
        reduction(+:sum)
for (k = 0; k < n; k++) {
  sum += factor/(2*k+1);
  factor = -factor;
}
pi_approx = 4.0*sum;
```

有什么问题？

一条语句由多个线程
并行执行产生数据依赖

根据k的奇偶性计算factor

- k为奇数，factor为-1；k为偶数，factor为+1

```
double factor = 1.0;
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
        reduction(+:sum)
for (k = 0; k < n; k++) {
  factor = (k % 2 == 0) ? 1.0 : -1.0;
  sum += factor/(2*k+1);
}
pi_approx = 4.0*sum;
```

仍是错误的！

factor应是私有的！

factor变为私有

```
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
        reduction(+:sum) private(factor)
for (k = 0; k < n; k++) {
  factor = (k % 2 == 0) ? 1.0 : -1.0;
  sum += factor/(2*k+1);
}
pi_approx = 4.0*sum;
```

起泡排序

```
for (list_length= n; list_length >= 2; list_length--)
  for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
        tmp = a[i];
        a[i] = a[i+1];
        a[i+1] = tmp;
}
```

外层循环：依赖关系跨越迭代步边界

内层循环迭代步之间也有依赖

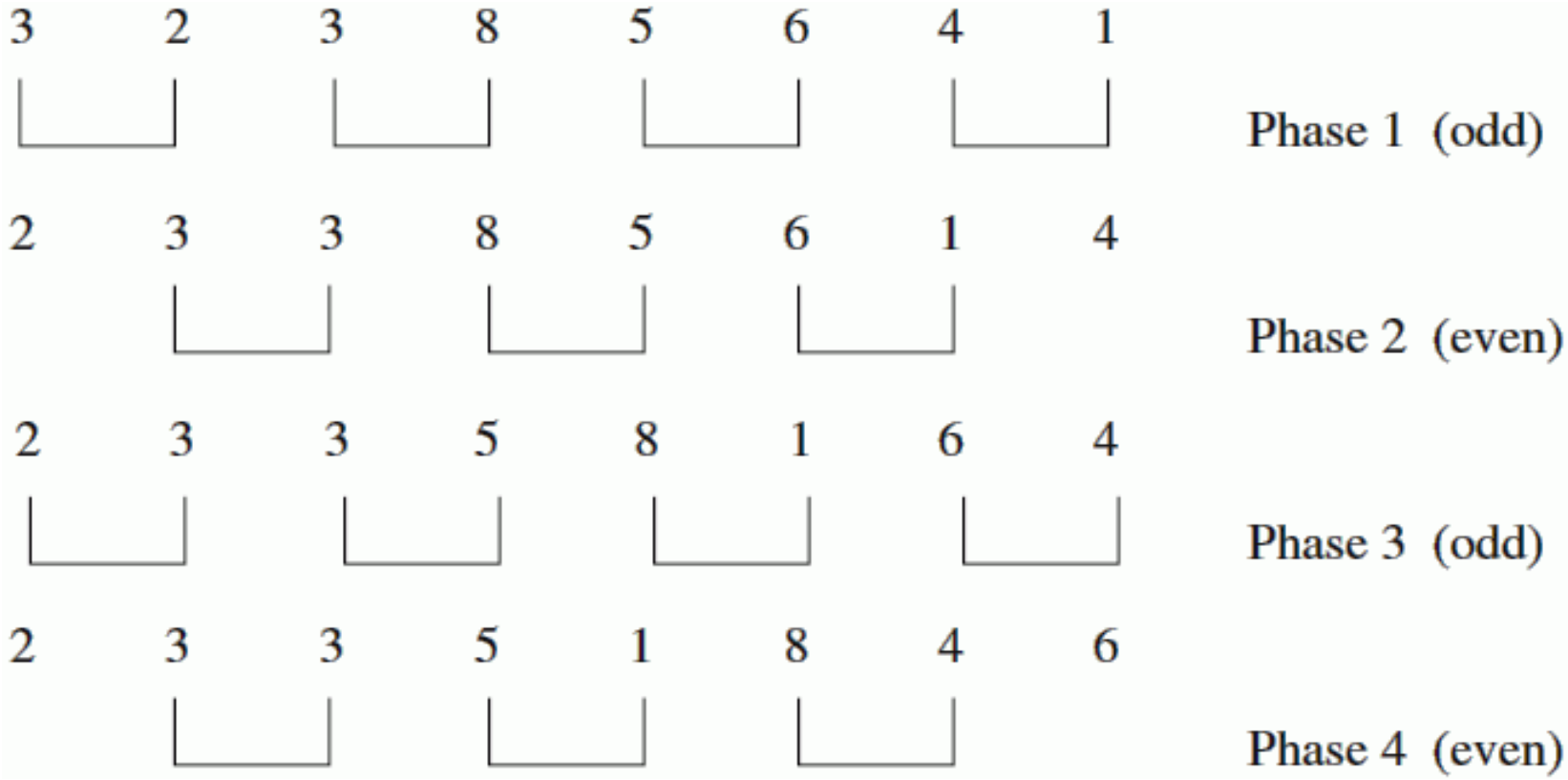
能否直接并行化？

- 例：a=[3,4,1,2]
  - 外循环，每次循环“下沉”最大数。
    第一步后 [3,1,2,4]
  - 内循环：每次用到a[i]和a[i+1]，明显存在循环依赖

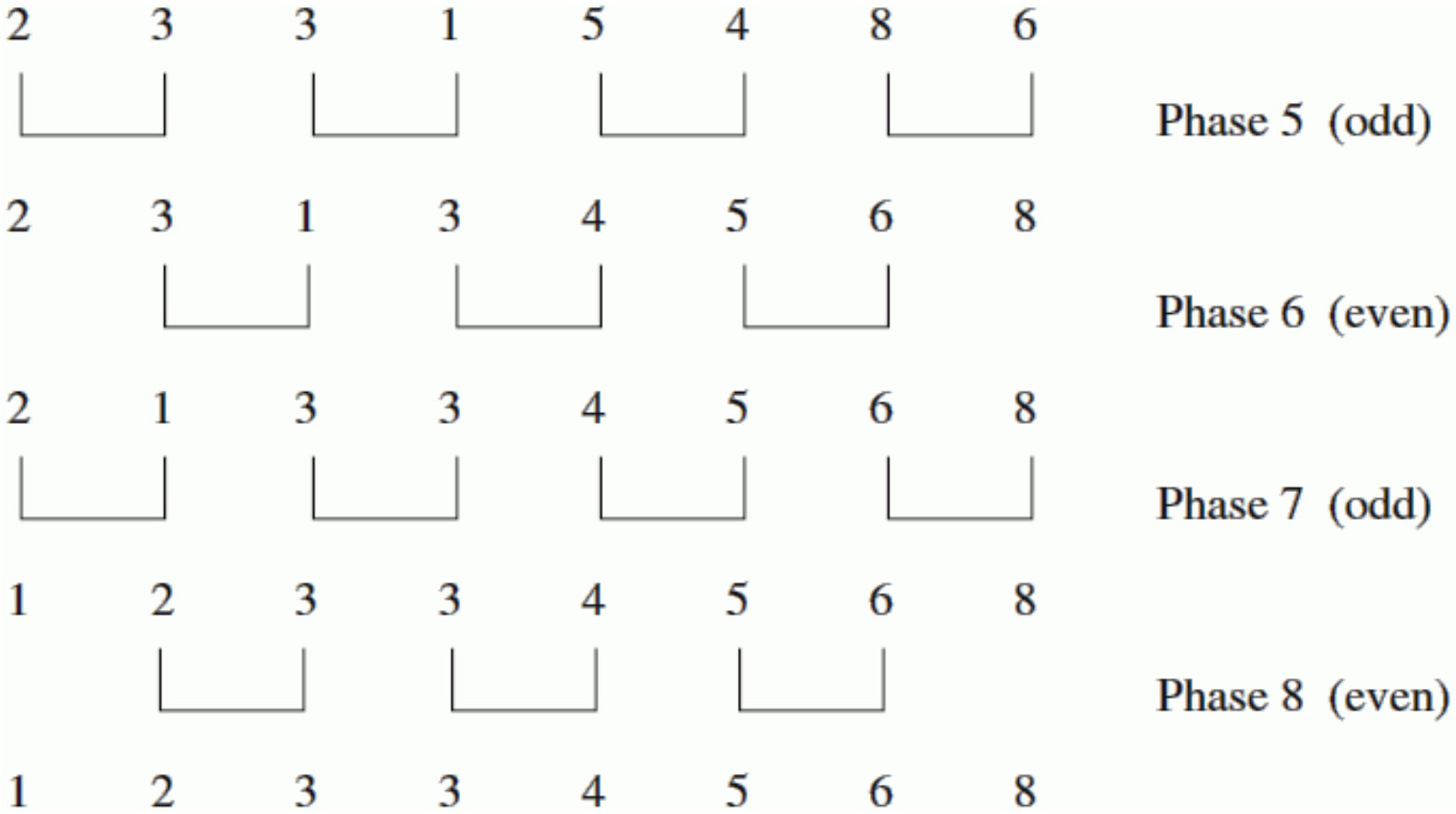
Odd-even转置排序

```
for (phase = 0; phase < n; phase++)
  if (phase % 2 == 0)
    for (i = 1; i < n; i += 2)
      if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
  else
    for (i = 1; i < n-1; i += 2)
      if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

Odd-even转置排序示例



Odd-even转置排序示例



Odd-even OpenMP版本

```
for (phase = 0; phase < n; phase++) {
  if (phase % 2 == 0)
#   pragma omp parallel for num_threads(thread_count)  \
          default(none) shared(a, n) private(i, tmp)
     for (i = 1; i < n; i += 2) {
       if (a[i-1] > a[i]) {
         tmp = a[i-1];
         a[i-1] = a[i];
         a[i] = tmp;
       }
     }
```

Odd-even OpenMP版本(2)

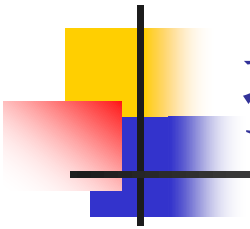
```
  else
#  pragma omp parallel for num_threads(thread_count)  \
        default(none) shared(a, n) private(i, tmp)
    for (i = 1; i < n - 1; i += 2) {
      if (a[i] > a[i+1]) {
        tmp = a[i+1];
        a[i+1] = a[i];
        a[i] = tmp;
      }
    }
}
```

这个程序存在什么问题？

频繁创建、销毁线程。

避免线程创建、销毁开销

```
#  pragma omp parallel num_threads(thread_count)  \
        default(none) shared(a, n) private(i, tmp, phase)
  for (phase = 0; phase < n; phase++) {
    if (phase % 2 == 0)
#     pragma omp for
      for (i = 1; i < n; i += 2) {
…
      }
    else
#     pragma omp for
      for (i = 1; i < n - 1; i += 2) {
…
    }
  }
```



提纲

- OpenMP并行模型
- 并行循环
- 数据依赖
- 循环调度
- 局部性
- 任务并行

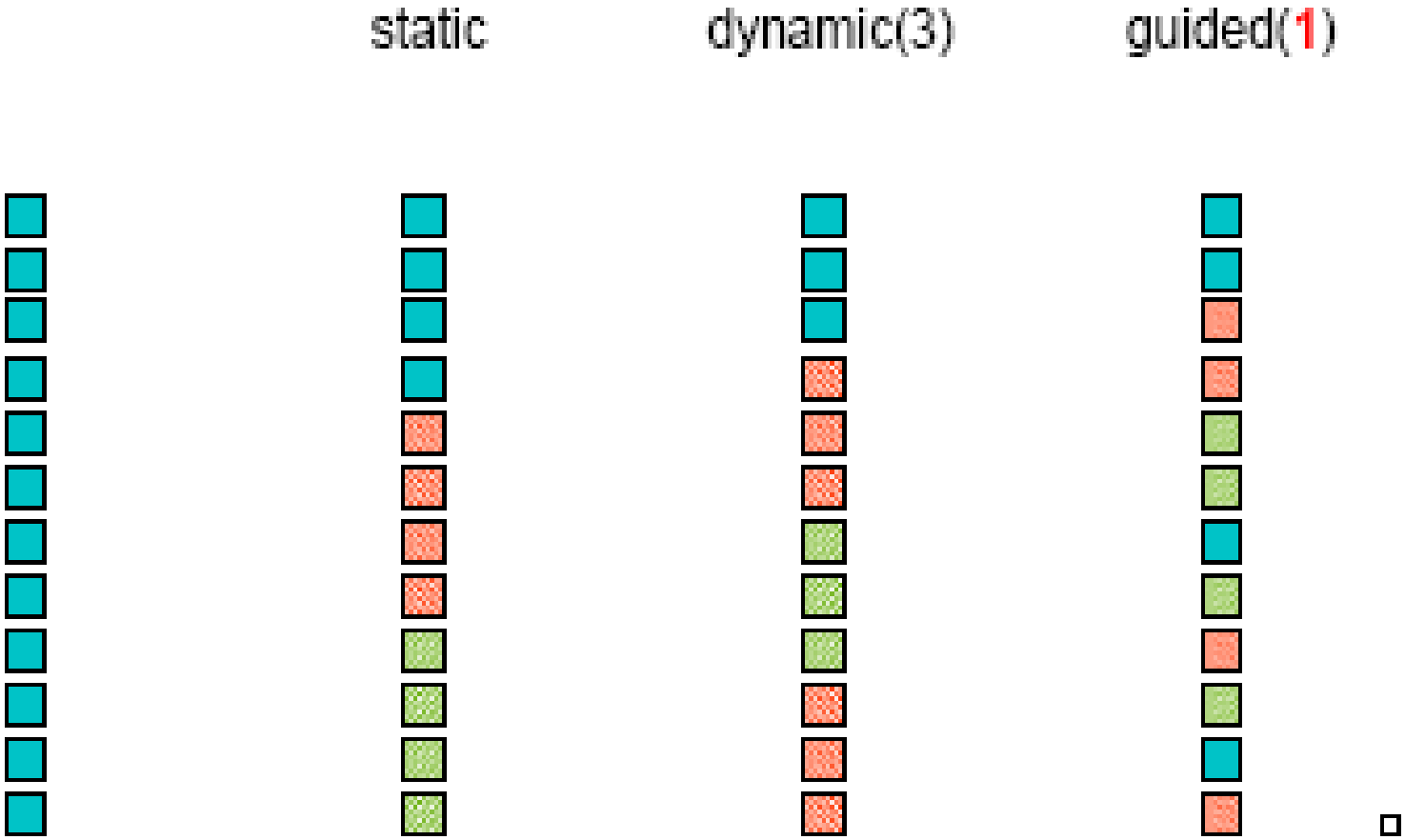
编程模型——循环调度

- schedule子句确定如何在线程间划分循环
  - static([chunk])静态划分
    - 分配给每个线程 [chunk]步迭代，所有线程都分配完后继续循环分配，直至所有迭代步分配完毕
    - 默认[chunk]为ceil(#iterations/#threads)
  - dynmaic([chunk])动态划分
    - 分给每个线程[chunk]步迭代，一个线程完成任务后再为其分配[chunk]步迭代
    - 逻辑上形成一个任务池，包含所有迭代步
    - 默认[chunk]为1
  - guided([chunk])动态划分，但划分过程中[chunk]指数减小
    - 类似于DYNAMIC调度，但分块开始大，随着迭代分块越来越少，循环区间的划分是基于类似下列公式完成的（不同的编译系统可能不同）：

$$S_k = \left\lceil \frac{R_k}{2N} \right\rceil$$

其中N是线程个数，Sk表示第k块的大小，Rk是剩余下未被调度的循环迭代次数。

循环调度



更多循环调度属性

- RUNTIME 调度决策推迟到运行时由环境变量指定
  需提前设定环境变量OMP_SCHEDULE
- AUTO 委托编译器和/或运行时系统做出调度决策
- NO WAIT/nowait：线程在并行循环结束时不进行同步
- ORDERED：循环步必须串行执行
- COLLAPSE：指出嵌套循环如何收缩为一个大的循环并根据子句进行划分（收缩顺序依据原串行执行顺序）

OpenMP环境变量

- OMP_NUM_THREADS
  - 设置执行期间使用的线程数
  - 若允许动态调整线程数，则此环境变量值为最大线程数
  - 例如

```
setenv OMP_NUM_THREADS 16 [csh, tcsh]
export OMP_NUM_THREADS=16 [sh, ksh, bash]
```

- OMP_SCHEDULE
  - 应用于调度类型为RUNTIME的do/for和parallel do/for指示
  - 为所有这些循环设置调度类型和块大小
  - 例如

```
setenv OMP_SCHEDULE GUIDED,4 [csh, tcsh]
export OMP_SCHEDULE=GUIDED,4 [sh, ksh, bash]
```

调度子句

- 默认调度

```
 sum = 0.0;
# pragma omp parallel for num_threads(thread_count) reduction(+:sum)
  for (i = 0; i <= n; i++)
     sum += f(i);
```

等价于

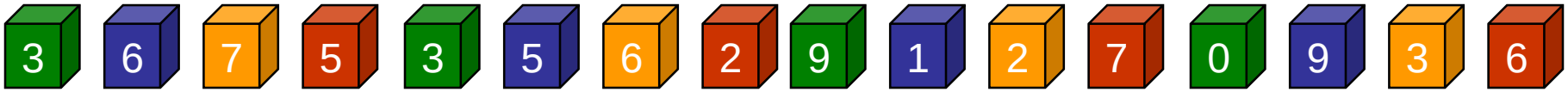
```
 sum = 0.0;
# pragma omp parallel for num_threads(thread_count) reduction(+:sum)
                          schedule(static, n/thread_count)
  for (i = 0; i <= n; i++)
     sum += f(i);
```

数据分布

- 全局数据如何划分到不同处理器

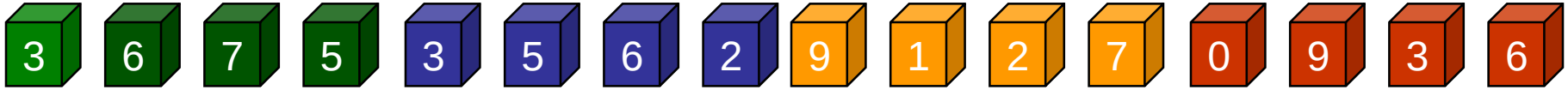
CYCLIC (chunk = 1): 循环划分

```
for (i = 0; i<blocksize; i++)
        … in [i*blocksize + tid];
```



BLOCK (chunk = 4): 块划分

```
for (i=tid*blocksize; i<(tid+1) *blocksize; i++)
    … in[i];
```



BLOCK-CYCLIC (chunk = 2): 块循环划分

调度例：多个不均衡数组排序

```
for (int i = 0; i < ARR_NUM; i++)
  stable_sort(arr[i].begin(), arr[i].end());
```

串行：**1969ms**

```
#pragma omp parallel for num_threads(THREAD_NUM)
  for (int i = 0; i < ARR_NUM; i++)
    stable_sort(arr[i].begin(), arr[i].end());
```

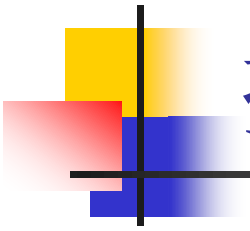
静态块划分：**618ms**

```
#pragma omp parallel for num_threads(THREAD_NUM) \
        schedule(dynamic, 50)
  for (int i = 0; i < ARR_NUM; i++)
    stable_sort(arr[i].begin(), arr[i].end());
```

动态划分：**516ms**

调度决策对性能的影响

- 负载均衡
  - 每步迭代工作量一致？
  - 处理器计算速度一致？
- 调度开销
  - 静态决策代价很低，因为不需要运行时协调
  - 动态决策：依赖于决策方法的复杂性和频率
- 数据局部性
  - 小chunk下，cache line内的局部性特别要考虑
  - 也影响同一个处理器上的数据重用

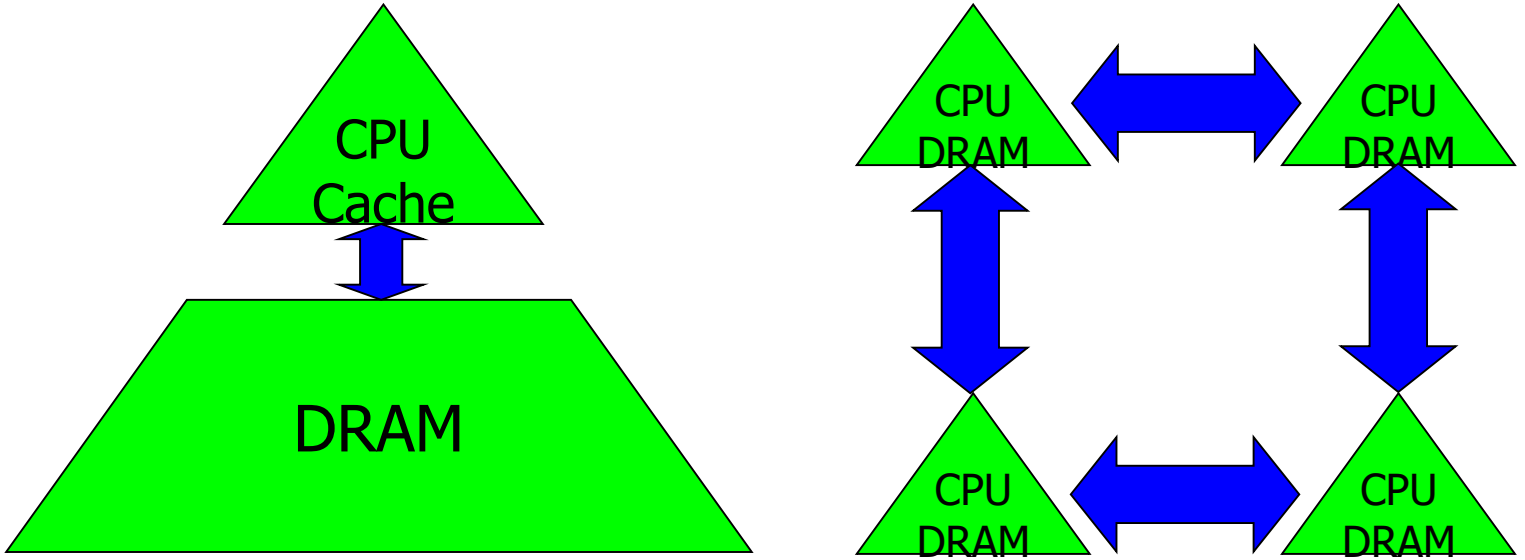


提纲

- OpenMP并行模型
- 并行循环
- 数据依赖
- 循环调度
- 局部性
- 任务并行

再次讨论局部性

- 算法代价分为两部分
  - 算术运算（FLOPS）
  - 通信：移动数据 ← 算法设计尽量减少通信
    - 内存的不同层次之间（串行算法）
    - 网络中的不同处理器之间（并行算法）



减少通信的重要性

- 算法运行时间可描述为3项：
  - #flops * time_per_flop
  - #words moved / bandwidth
  - #messages *latency

  } communication

- Time_per_flop << 1/bandwidth << latency
  - 它们的发展速度也有指数差距

Annual improvements			
Time_per_flop		Bandwidth	Latency
59%	Network	26%	15%
	DRAM	23%	5%

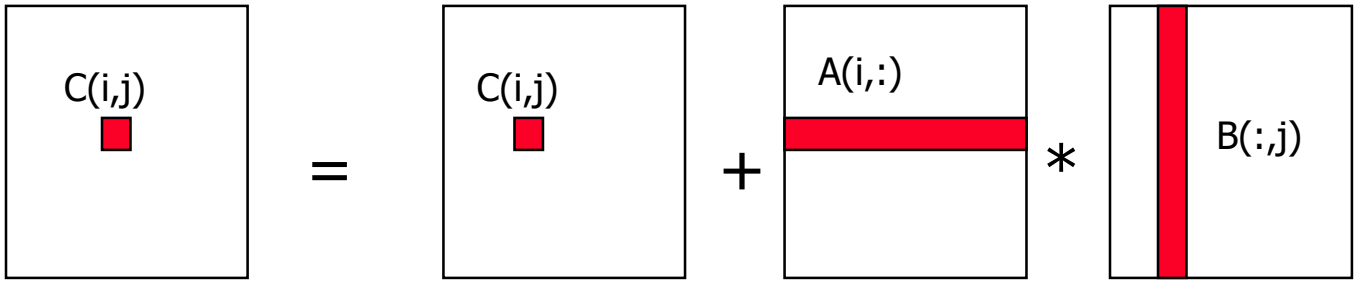
- 目标：重组计算流程，减少通信
  - 所有内存层次间： L1 ⟷ L2 ⟷ DRAM ⟷ network
  - 不只是*隐藏*通信（与计算重叠），可能达到很高加速比

Slide source: Jim Demmel, CS267

串行矩阵乘法C=C+A*B

```
for i = 1 to n
 {将A的第i行读入缓存，n² 次读}
  for j = 1 to n
     {将C(i,j)读入缓存，n² 次读}
     {将B的第j列读入缓存，n³ 次读}
     for k = 1 to n
        C(i,j) = C(i,j) + A(i,k) * B(k,j)
     {将C(i,j)写回内存，n² 次写}
```

共$n^3 + O(n^2)$ 次读/写



Slide source: Jim Demmel, CS267

分块矩阵乘法减少通信

- 分块矩阵乘法C=A·B，元素乘/加→子矩阵乘/加，子矩阵规模依赖于cache大小
  - A^{nxn}, B^{nxn}, C^{nxn} 分解为bxb的子矩阵，标记为A(i,j)…
  - b的选择——3个bxb的子矩阵可放入cache中

for i = 1 to n/b for j=1 to n/b for k=1 to n/b

C(i,j) = C(i,j) + A(i,k)·B(k,j) … b x b的矩阵乘法，$4b^2$ 次读/写

  - 共$(n/b)^3 \cdot 4b^2 = 4n^3/b$次读/写
  - 最小化条件：b尽量大，但子矩阵能放入cache $3b^2$=cache大小M，$O(n^3/M^{1/2})$
  - 多个内存层次（L1、L2、…）时会怎样？
    - 每个层次可能都需要多出3层循环

分块 vs Cache-Oblivious算法

- 分块矩阵乘法C=A·B，元素乘/加
  →子矩阵乘/加，子矩阵规模依赖于cache大小
  - A^{nxn}, B^{nxn}, C^{nxn} 分解为bxb的子矩阵，标记为A(i,j)…
  - b的选择——3个bxb的子矩阵可放入cache中

```
for i = 1 to n/b   for j=1 to n/b   for k=1 to n/b
    C(i,j) = C(i,j) + A(i,k)·B(k,j)     … b x b的矩阵乘法，4b² 次读/写
    多一个内存层次就需要多出3层循环
```

- Cache-Oblivious方法：C=A·B与cache无关

```
Function C = RMM(A,B)   … “R”表示递归
If A and B are 1x1
    C = A · B
else   … 将A^nxn, B^nxn, C^nxn 分解为(n/2)x(n/2) 子矩阵，标记为A(i,j)…
    for  i = 1 to 2,    for  j = 1 to 2,    for  k = 1 to 2
        C(i,j) = C(i,j) + RMM( A(i,k), B(k,j) )     … n/2 x n/2 矩阵乘法
```

通信开销下界

- 假定n^3次算术运算
- 串行算法，需快速存储空间大小为M
  - #words moved下界为$\Omega\,(n^3 / M^{1/2})$ [Hong, Kung, 81]
  - 用分块算法或Cache-Oblivious算法可达到
- P个处理器上的并行算法
  - 假设每个处理器内存为M，算法负载均衡
  - #words moved下界为$\Omega\,(n^3 /(p \cdot M^{1/2}))$ [Irony, Tiskin, Toledo, 04]
  - 若$M = 3n^2/p$（每个矩阵一份副本，均匀散布在p个处理器），则下界为$\Omega\,(n^3 /p^{1/2})$
  - SUMMA算法、Cannon算法可以达到

Slide source: Jim Demmel, CS267

适用于所有“直接”线性代数运算的新下界

令M=每个处理器中“快速”存储大小

=cache大小（串行）或$O(n^2/p)$（并行）

#flops=每个处理器完成的浮点运算数，则

每处理器#words moved= $\Omega(\#flops / M^{1/2})$

每处理器#messages sent= $\Omega(\#flops / M^{3/2})$

- 矩阵乘、BLAS、LU、QR、eig、SVD、张量约缩、…
- 若干这些操作组成的程序，如计算A^k
- 稠密及稀疏矩阵（#flops << n^3）
- 串行和并行算法
- 某些图论算法（如Floyd-Warshall算法）

适用于所有“直接”线性代数运算的新下界

令M=每个处理器中“快速”存储大小

=cache大小（串行）或$O(n^2/p)$（并行）

#flops=每个处理器完成的浮点运算数，则

每处理器#words moved= $\Omega(\#flops / M^{1/2})$

每处理器#messages sent= $\Omega(\#flops / M^{3/2})$

- 串行情况，稠密n x n矩阵，因此$O(n^3)$ flops
  - #words moved= $\Omega(n^3 / M^{1/2})$
  - #messages_sent = $\Omega(n^3 / M^{3/2})$
- 并行情况，稠密n x n矩阵
  - 负载均衡，因此每个处理器$O(n^3/p)$
  - 数据一份拷贝，均衡分布，因此每个处理器$M=O(n^2/p)$
  - #words moved= $\Omega(n^3 / p^{1/2})$
  - #messages_sent = $\Omega(p^{1/2})$

Slide source: Jim Demmel, CS267

可否达到下界？

- LAPACK和ScaLPACK等传统实现是否达到？
  - 大部分没有达到
- 是否有其他算法实现能达到？有
- 算法的目标
  - 最小化#words moved
  - 最小化#message sent
    - 需要新的数据结构
  - 对多级内存层次进行优化： Cache-Oblivious算法最简单
  - 当矩阵能放入最快内存层次时尽量减少flops
    - Cache-Oblivious算法并不总是能达到
- 几乎对所有稠密线性代数计算都能达到
  - 目前为止只有少部分实现达到了
  - 只有少数稀疏算法达到了（如Cholesky）

重用和局部性

- 考虑数据如何访问
  - 数据重用
    - 多次使用相同或邻近数据
    - 计算中的固有特征
  - 数据局部性
    - 数据重用，且从“快速存储”访问
    - 使用相同数据或相同的数据阐述
  - 如果计算存在重用，我们如何获得局部性？
    - 恰当的数据布局
    - 代码的重排转换

利用重用：局部性优化

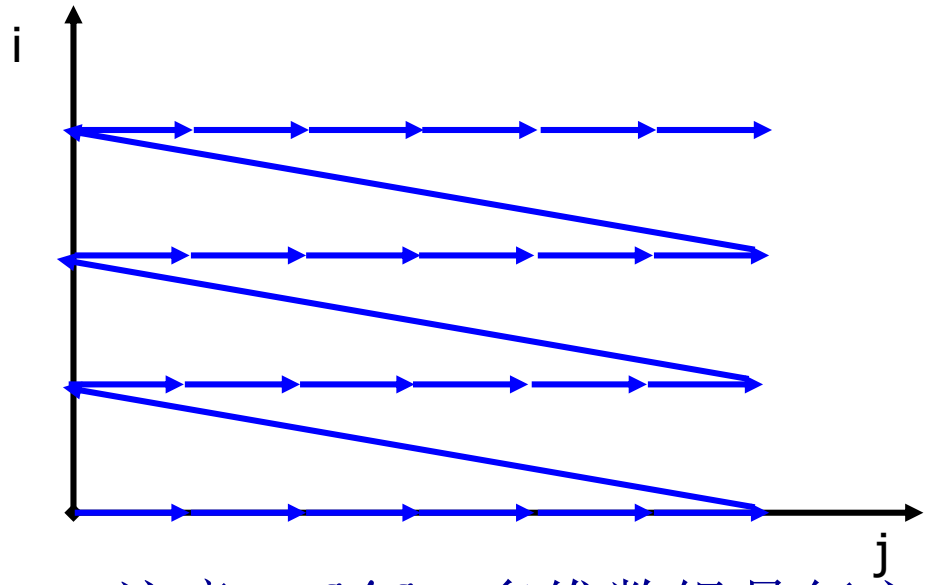
- 循环转换：重排内存访问，提高局部性
- 这些转换方法也适合于并行化
- 两个关键问题
  - 安全性
    - 转换保持了依赖关系？
  - 收益
    - 转换是否能带来收益？
    - 收益是否大于转换开销？

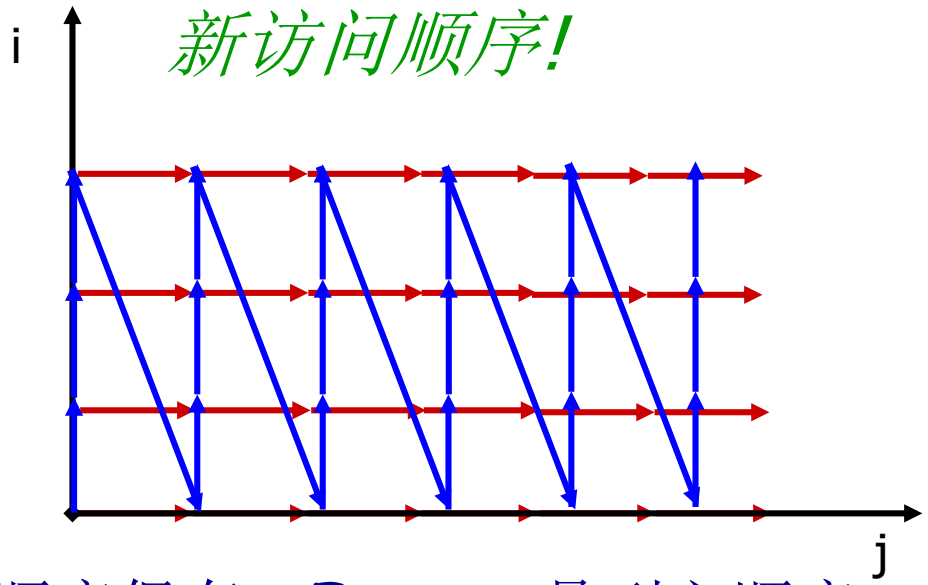
循环转换：重排

重排循环顺序以改变访存顺序

```
for (i= 0; i<3; i++)
  for (j=0; j<6; j++)
    A[i][j+1]=A[i][j]+B[j];
```

```
for (j=0; j<6; j++)
  for (i= 0; i<3; i++)
    A[i][j+1]=A[i][j]+B[j];
```

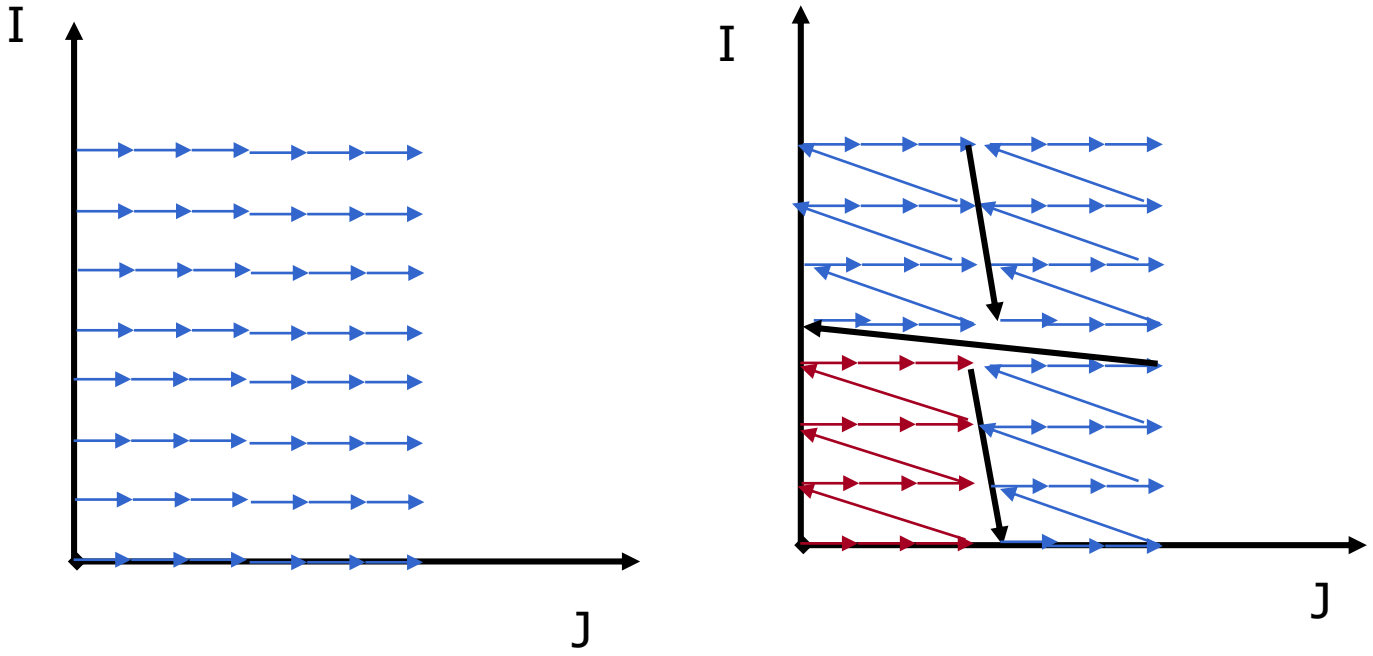




注意：C/C++多维数组是行主顺序保存，Fortran是列主顺序

分片（分块）

- 令重用数据的迭代步在时间上更靠近
- 目标是让数据在重用间隔内驻留在高速存储中



分片示例

```
for (j=1; j<M; j++)
   for (i=1; i<N; i++)
      D[i] = D[i] +B[j,i]
```

Strip mine

```
for (j=1; j<M; j++)
   for (ii=1; ii<N; ii+=s)
      for (i=ii; i<min(ii+s-1,N); i++)
         D[i] = D[i] +B[j,i]
```

Permute

```
for (ii=1; ii<N; ii+=s)
   for (j=1; j<M; j++)
      for (i=ii; i<min(ii+s-1,N); i++)
         D[i] = D[i] +B[j,i]
```

Unroll——循环展开

- 简单重复语句，重复次数——展开因子
- 只要重复次数不超过迭代数，就安全
- Unroll-and-jam展开外层循环，并重复内存循环的语句（不保证安全）
- 最有效的优化，但过分展开有风险

Original:

```
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i];
```

Unroll j

```
for (i=0; i<4; i++)
 for (j=0; j<8; j+=2)
  A[i][j] = B[j+1][i];
  A[i][j+1] = B[j+2][i];
```

Unroll-and-jam i

```
for (i= 0; i<4; i+=2)
 for (j=0; j<8; j++)
   A[i][j] = B[j+1][i];
   A[i+1][j] = B[j+1][i+1];
```

Unroll-and-Jam如何有利局部性

Original:

```
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i] + B[j+1][i+1];
```

Unroll-and-jam i and j loops

```
for (i=0; i<4; i+=2)
 for (j=0; j<8; j+=2) {
  A[i][j]   = B[j+1][i] + B[j+1][i+1];
  A[i+1][j] = B[j+1][i+1] + B[j+1][i+2];
  A[i][j+1] = B[j+2][i] + B[j+2][i+1];
  A[i+1][j+1] = B[j+2][i+1] + B[j+2][i+2];
}
```

- 短时间内复用B的元素
- 外层循环展开越多效果越明显

Unroll-and-Jam的其他优点

Original:

```
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i] + B[j+1][i+1];
```

- 更少的循环控制操作
- 无关的计算→指令级并行

Unroll-and-jam i and j loops

```
for (i=0; i<4; i+=2)
 for (j=0; j<8; j+=2) {
  A[i][j]   = B[j+1][i] + B[j+1][i+1];
  A[i+1][j] = B[j+1][i+1] + B[j+1][i+2];
  A[i][j+1] = B[j+2][i] + B[j+2][i+1];
  A[i+1][j+1] B[j+2][i+1] + B[j+2][i+2];
}
```

分片的安全性

- 分片= strip-mine和permutation
  - Strip-mine不重排迭代步
  - Permutation必须合法
    或
    条纹长度小于依赖距离

Unroll-and-Jam的安全性

- Unroll-and-Jam=分片+展开
  - Permutation必须合法
    或
    展开大小小于依赖距离

Unroll-and-jam = tile + unroll?

Original:

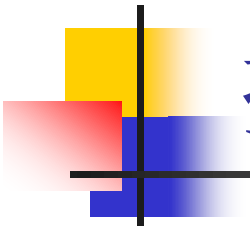
```
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i];
```

Tile i loop:

```
for (ii=0; ii<4; ii+=2)
 for (j=0; j<8; j++)
   for (i=ii; i<ii+2; i++)
      A[i][j] = B[j+1][i];
```

Unroll i tile:

```
for (ii= 0; ii<4; ii+=2)
 for (j=0; j<8; j++)
   A[i][j] = B[j+1][i];
   A[i+1][j] = B[j+1][i+1];
```



提纲

- OpenMP并行模型
- 并行循环
- 数据依赖
- 循环调度
- 局部性
- 任务并行

条件并行

if (scalar expression)

✓*仅当表达式求值为真时才并行执行*
✓*否则串行执行*

```
#pragma omp parallel if (n > threshold) \
shared(n,x,y) private(i)   {
#pragma omp for
for (i=0; i<n; i++)
    x[i] += y[i];
} /*-- End of parallel region --*/
```

Review:
- parallel
- for
- private
- shared

Single和Master结构

- 线程组中只有一个线程执行代码
- 用于I/O或初始化等任务
- 入口或出口无隐式barrier

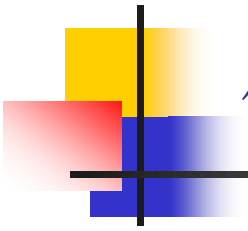
```
#pragma omp single {
   <code-block>
}
```

- 类似的，只有主线程执行代码

```
#pragma omp master {
   <code-block>
}
```

更多的控制机制

```
#pragma omp parallel shared(A,B,C)
{
   tid = omp_get_thread_num();
   A[tid] = big_calc1(tid);
  #pragma omp barrier
  #pragma omp for
    for (i=0; i<N; i++) C[i] = big_calc2(tid);
  #pragma omp for nowait
    for (i=0; i<N; i++) B[i] = big_calc3(tid);
  A[tid] = big_calc4(tid);
}
```

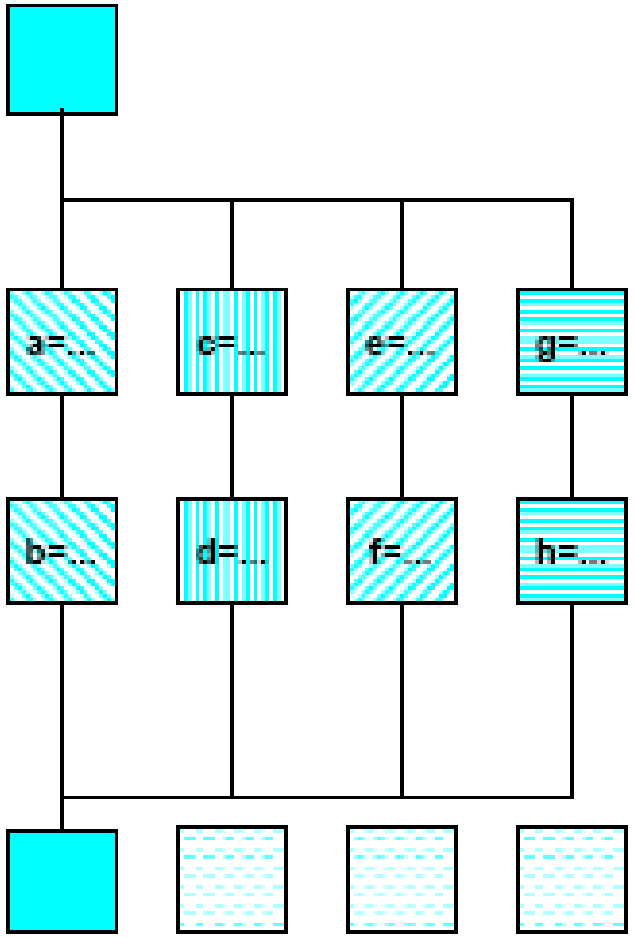


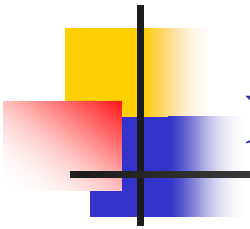
任务并行

- 定义
  - 同时执行不同计算任务。由于任务数固定，因此是不可扩展的
- OpenMP支持任务并行
  - 并行区域：不同线程执行相同代码
  - 任务机制：不同时间创建和执行任务
- 任务并行的常见用途：生产者/消费者
  - 生产者创建工作，消费者去处理
  - 可理解为一种流水线，类似组装线
  - “生产者”写FIFO队列，“消费者”读取

简单方式：OpenMP区域指示

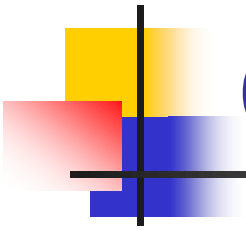
```
#pragma omp parallel
{
#pragma omp sections
#pragma omp section
   {{ a=...;
   b=...; }
#pragma omp section
   { c=...;
   d=...; }
#pragma omp section
   { e=...;
   f=...; }
#pragma omp section
   { g=...;
   h=...; }
} /*omp end sections*/
} /*omp end parallel*/
```





并行区域例子

```
#pragma omp parallel shared(n,a,b,c,d) private(i)
{
  #pragma omp sections nowait    {
    #pragma omp section
      for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
    #pragma omp section
      for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
  } /*-- End of sections --*/
} /*-- End of parallel region
```



OpenMP 3.0中的任务

- 一个任务具有
  - 要执行的代码
  - 数据环境（共享的、私有的、归约的）
  - 使用数据执行代码的线程
- 两个动作：打包和执行
  - 每个线程打包一个新的任务
  - 线程组中某个线程随后执行任务

简单的生产者-消费者例子

```
// PRODUCER: initialize A with random data
void fill_rand(int nval, double *A) {
  for (i=0; i<nval; i++) A[i] = (double) rand()/1111111111;
}

// CONSUMER: Sum the data in A
double Sum_array(int nval, double *A) {
  double sum = 0.0;
  for (i=0; i<nval; i++)  sum = sum + A[i];
  return sum;
}
```

生产者—消费者模型关键问题

- 生产者需通知消费者数据已就绪
- 消费者需要等待直至数据就绪
- 生产者和消费者需要一种方式交互数据
  - 生产者的输出是消费者的输入
- 通常通过先进先出（FIFO）队列交互

FIFO队列读写方法

- FIFO放在全局内存中，被线程们共享
- 如何确保数据更新？
- 需要一种结构保证内存的一致视图
  - 刷新：确保数据写回全局内存

Example:

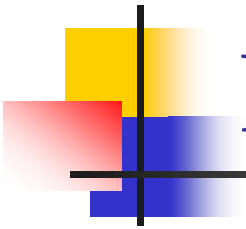
```
Double A;
A = compute();
Flush(A);
```

解决方案

```
flag = 0;
#pragma omp parallel
{
   #pragma omp section
   {
      fillrand(N,A);
      #pragma omp flush
      flag = 1;
      #pragma omp flush(flag)
   }

   #pragma omp section
   {
       while (!flag)
          #pragma omp flush(flag)
      #pragma omp flush
      sum = sum_array(N,A);
   }
}
```

生产者：生成随机数数组

消费者：数组求和



Flush指示

- Flush
  - 指出所有线程对所有共享对象都有一致的内存视图
- Flush(var)
  - 只刷新共享变量“var”

另一个简单的例子

```
for (j=0; j<M; j++) {
    sum[j] = 0;
    for(i = 0; i < size; i++) {
      // 任务1：缩放结果
      out[i] = _iplist[j][i]*(2+i*j);
      // 任务2：求和
      sum[j] += out[i];
    }
    // 计算平均值，并求最大平均值
    res = sum[j] / size;
    if (res > maxavg) maxavg = res;
}
return maxavg;
```

实现消息传递

- 在共享内存系统实现消息传递
- 用一个FIFO队列保存消息
- 线程显式发送/接收消息
  - 发送消息：队列入队
  - 接收消息：出队
  - 确保安全访问

消息传递

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
  Send_msg();
  Try_receive();
}

while (!Done())
  Try_receive();
```

发送消息

```
    mesg = random();
    dest = random() % thread_count;
#   pragma omp critical
    Enqueue(queue, dest, my_rank, mesg);
```

使用同步机制更新FIFO队列

接收消息

```
    if (queue_size == 0) return;
    else if (queue_size == 1)
#     pragma omp critical
      Dequeue(queue, &src, &mesg);
    else
      Dequeue(queue, &src, &mesg);
Print_message(src, mesg);
```

只有这个线程从队首取消息

其他线程向队尾添加消息

因此，尽在尾元素处才需同步

终止检测

```
queue size = enqueued  - dequeued;
if (queue_size == 0 && done_sending == thread_count)
    return TRUE;
else
    return FALSE;
```

每个线程在自己的工作完成后
都递增此变量

对done_sending需要更多同步操作

任务例：链表遍历

```
.....
while (my_pointer) {
    (void) do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
.....
```

- 如何用并行for表达？
  - 迭代次数必须固定
  - 循环不变的循环条件，不能提前退出

用OpenMP 3.0实现！

```
my_pointer = listhead;
#pragma omp parallel    {
   #pragma omp single nowait    {
      while (my_pointer) {
        #pragma omp task firstprivate(my_pointer)          {
          (void) do_independent_work (my_pointer);
        }
        my_pointer = my_pointer->next ;
      }
   } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier here
```

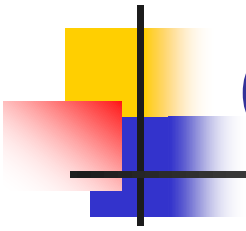
while循环由单一线程串行执行

不等待任务完成就遍历下一个结点

创建出的任务由其他线程并发执行

firstprivate: 从全局变量拷贝初始值到私有变量

lastprivate: 将私有变量值拷贝回全局变量



OpenMP小结

- 优点
  - 简单修改串行程序即可得到并行程序
  - 不必关心低层映射细节
  - 可移植、可扩展、单处理器也正确
- 缺点
  - 从头写并行程序的话不是很自然
  - 表达常见并行结构并不总是可行
  - 局部性处理
  - 性能控制