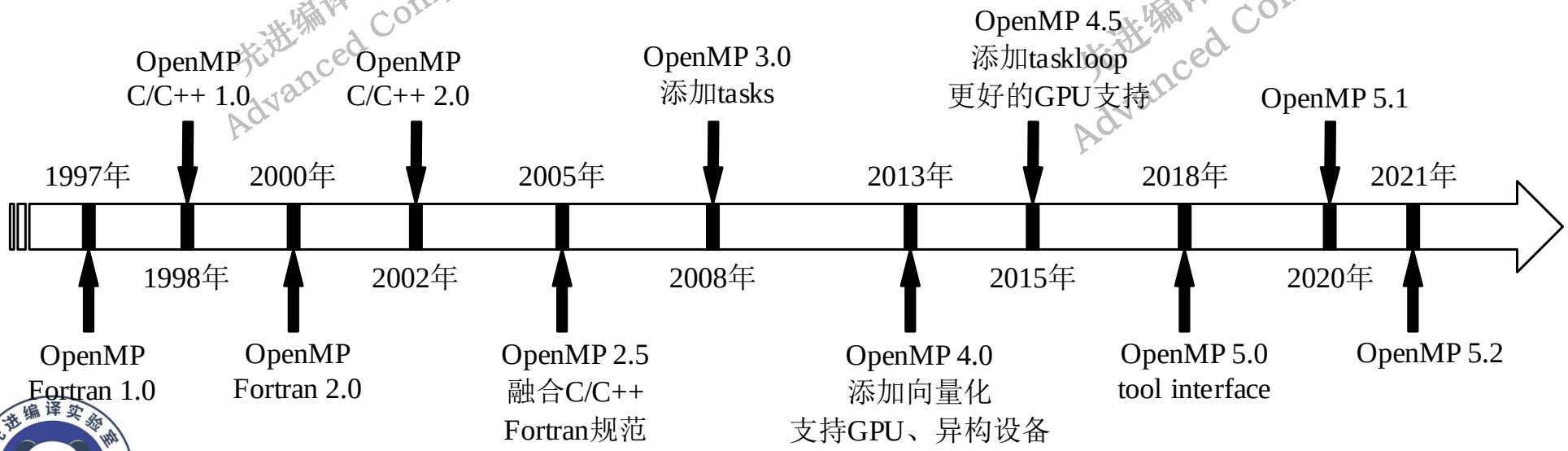
OpenMP编程简介

嘉宾：王磊

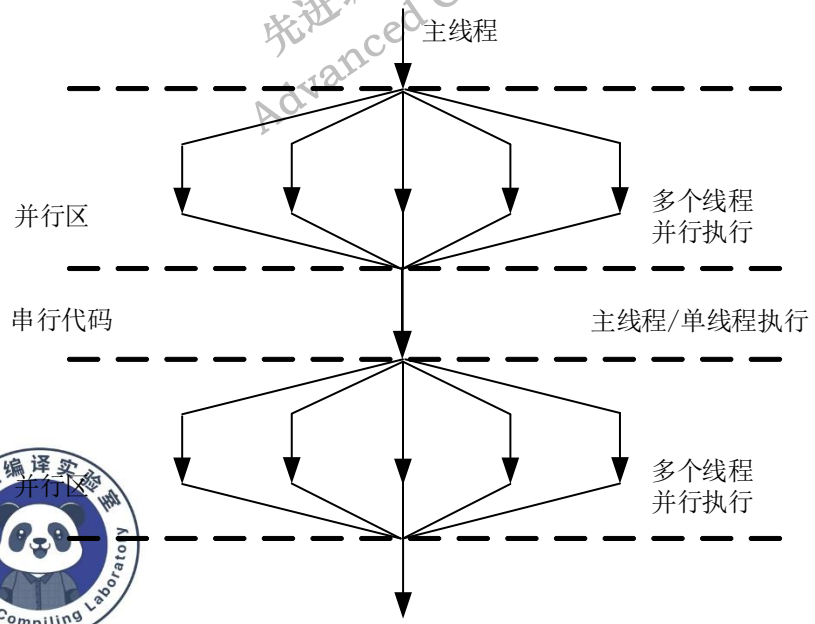
OpenMP是什么

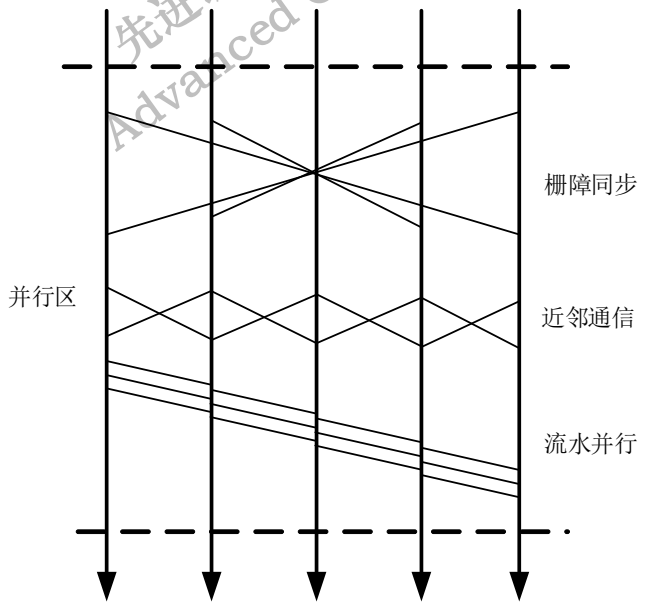
OpenMP是一种用于共享内存并行编程的多线程程序设计方案，适合在共享内存编程下的多核系统上进行并行程序设计。OpenMP的使用可以降低多核并行编程的难度，优化人员可以更多地考虑算法本身，而非具体的并行实现细节。



OpenMP是什么

- Fork-Join模式：在并行区的开始位置建立多个线程，在并行区的结束位置自动合并线程。
- SPMD模式：在单个并行程序中按照线程号来匹配程序分支，不同线程分配不同的计算任务。

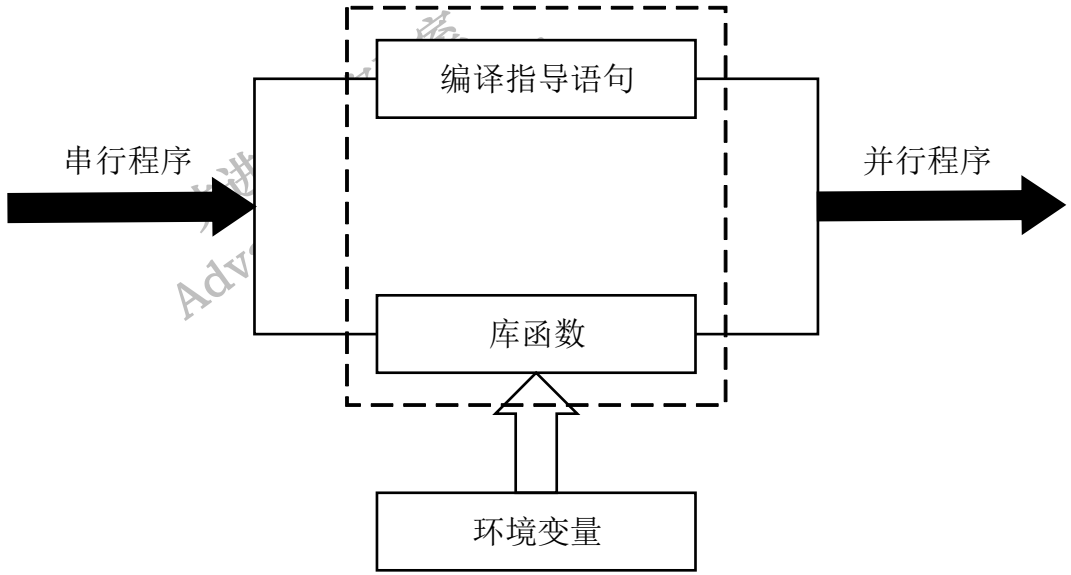




OpenMP指导语句

OpenMP程序由指导语句、库函数和环境变量三部分组成。

- 指导语句是串行程序实现并行化的桥梁，是编写OpenMP程序的关键。
- 库函数的作用是在程序运行阶段改变和优化并行环境从而控制程序的运行。
- 环境变量是库函数中控制函数运行的一些具体参数，



OpenMP指导语句

指导标识符	指导命令	子句列表	续行符
#pragma omp	parallel for	num_threads(4) private(tid,mcpu) shared(sum)	\ 换行符

```
for(int i=0;i<N;i++){
//并行执行代码
}
```

OpenMP指导语句

并行控制类	parallel、simd
工作共享类	for、sections、task、single、target
线程同步类	barrier、master、critical、atomic、flush、ordered
复合指导命令类	parallel for、for simd、parallel sections

OpenMP指导语句

parallel 指导语句

#pragma omp parallel [clause[[,]clause]……]
  structured-block
clause :
  allocate([allocator :] list)
  copyin(list)
  default(shared | firstprivate | private | none)
  firstprivate(list)
  if([parallel :]omp-logical-expression)
  num_threads(nthreads)
  private(list)
  proc_bind(close | primary | spread)
  reduction([reduction-modifier ,] reduction-identifier : list)
  shared(list)

for 指导语句

#pragma omp for[clause[[,]clause]……]
  loop-nest
clause:
  allocate([allocator :] list)
  collapse(n)
  firstprivate(list)
  lastprivate([lastprivate-modifier:] list)
  linear(list[:linear-step])
  nowait
  order([order-modifier:]concurrent)
  ordered[(n)]
  private(list)
  reduction([reduction-modifier ,] reduction-identifier : list)
  schedule ([modifier [, modifier] :] kind[, chunk_size])

先进编译实验室
Advanced Compiler

OpenMP版矩阵乘


```
#include<stdio.h>
#include<omp.h>
#define N 2000
float A[N][N],B[N][N];
float C[N][N];
int main(){
    int i,j,k;
    float sum=0.0;
    double start_time,end_time,used_time;
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            A[i][j]=i+1.0;
            B[i][j]=1.0;
            C[i][j]=0.0;
        }
    }
```

并行化改写

```
start_time=omp_get_wtime();
#                                                    hreads(4)
  for(i=0;i<N;i++)
      for(j=0;j<N;j++)
          for(k=0;k<N;k++)
              C[i][j]+=A[i][k]*B[k][j];
                                                    num_threads(4)
                                                    ,B,C) num_threads(4)
end_time=omp_get_wtime();
used_time=end_time-start_time;
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        sum+=C[i][j];
printf("sum=%lf,used_time=%lf s\n",sum,used_time);
```

热点代码段

参考资料

[1] 雷洪，胡许冰编著.多核并行高性能计算 OpenMP[M].北京：冶金工业出版社,2016.

AdvancedCompiler

Tel: 13839830713

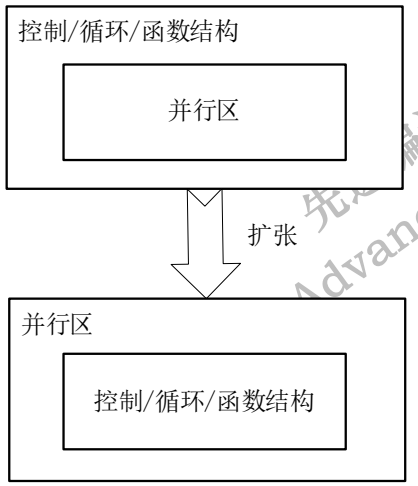
并行区重构

嘉宾：王磊

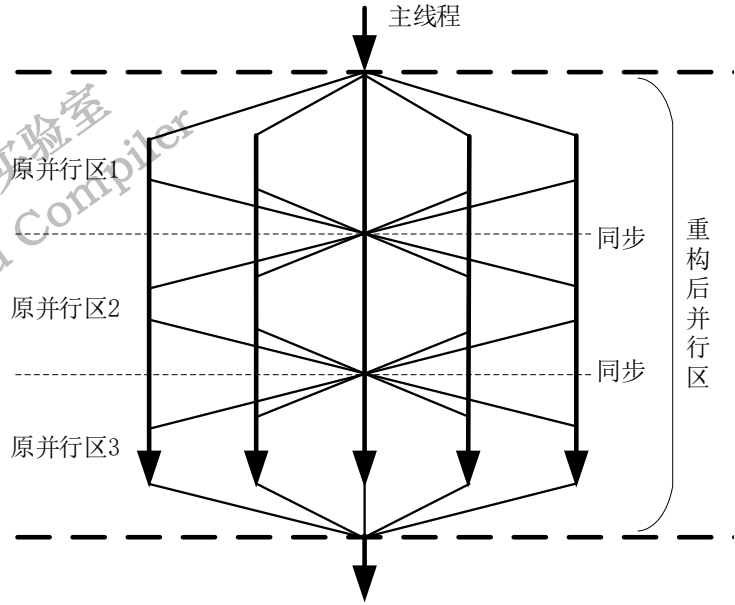
并行区重构

OpenMP Fork-Join模式程序运行过程中线程的创建和合并比较频繁，在并行性表达上处于一种低效状态。

并行区重构是结合数据和计算划分等信息，通过改变原并行区的结构，降低串并行程序之间切换以及其他开销。并行区重构包括并行区扩张和并行区合并两种方式。



(a)并行区扩张



(b)并行区合并

并行区扩张

循环结构并行区扩张针对的是包含整个循环结构的并行区，将此并行区扩张到循环之外，若该循环迭代次数为N，则并行区扩张后线程的创建和合并次数将减少N-1次。

√
```
#pragma omp parallel for private(j,k) shared(A,B,C) num_threads(4)
for(i=0;i<N;i++)
```

√
```
 #pragma omp parallel for private(j,k) shared(A,B,C) num_threads(4)
    for(j=0;j<N;j++)
        for(k=0;k<N;k++)
            C[i][j]+=A[i][k]*B[k][j];
```

并行区扩张

函数结构并行区扩张是指将函数结构内部的并行区扩张到整个函数结构的外部，并行区扩张后能够将函数结构内部的全部语句包含在并行区中，进一步获得更多的并行区合并机会。

```
void init_array(int* a){
#pragma omp parallel for
for(int i=0;i<N;i++)
      a[i]=I;
}
int main(){
       init_array(A);
       init_array(B);
       init_array(C);
}
```

√

```
void init_array(int* a){
 #pragma omp for {
  for(int i=0;i<N;i++)
      a[i]=i;
}
int main(){
#pragma omp parallel {
       init_array(A);
       init_array(B);
       init_array(C);
  }
}
```

并行区合并

并行区合并不仅仅是将原有程序的多个并行区改写至一个#pragma omp parallel区域，往往还需要考虑到并行区合并是否会修改原程序语义等问题。

为了确保并行区合并**不修改原程序语义**，就需要保证合并后各个子线程间**数据更新顺序**和**执行顺序与原程序保持一致性**，前者可以通过指导语句flush来实现，后者可以通过指导语句barrier进行同步来实现。

除此之外，还需要考虑合并过程中的**变量数据属性冲突**以及**并行区之间串行语句的处理**等问题。

并行区合并

变量属性冲突处理

```
int main(){
   int k=2,
   sum1=0,sum2=0;
 //并行区1
     #pragma omp parallel shared(k)
     # pragma omp for reduction(+: sum1)
     for (int i=0;i<10000;i++)
        sum1 += (k+i); }
 //并行区2
     #pragma omp parallel firstprivate(k)
     # pragma omp for reduction(+: sum2)
     for (int j=0;j<10000;j++)
        sum2 += (2*k+j);
}
```

√

```
int main(){
   int k=2,
   sum1=0,sum2=0;
     //合并后并行区
      #pragma omp parallel firstprivate(k){
       # pragma omp for reduction(+: sum1)
       for (int i=0;i<10000;i++)
          sum1 += (k+i);
       # pragma omp for reduction(+: sum2)
       for (int j=0;j<10000;j++)
          sum2 += (k*2+j);
     }
}
```

并行区合并


```
int main(){
int k=2, sum1=0,sum2=0;
//并行区1
    #pragma omp parallel shared(k){
      # pragma omp for reduction(+: sum1)
      for (int i=0;i<10000;i++)
          sum1 += (k+i);
    }
    sum2 = sum1;
    k++;
  //并行区2
      #pragma omp parallel firstprivate(k){
      # pragma omp for reduction(+: sum2)
      for (int j=0;j<10000;j++)
          sum2 += (2*k+j);
    }
printf("sum1=%d, sum2=%d\n", sum1, sum2);
}
```


串行语句处理

```
int main(){
    int k=2, sum1=0,sum2=0;
    #pragma omp parallel firstprivate(k) shared(sum1,sum2){
      # pragma omp for reduction(+: sum1)
      for (int i=0;i<10000;i++)
          sum1 += (k+i);
√     #pragma omp single
      sum2 = sum1;
      k++;
      # pragma omp for reduction(+: sum2)
      for (int j=0;j<10000;j++)
          sum2 += (2*k+j);
√     #pragma omp master
      printf("sum1=%d, sum2=%d\n", sum1, sum2);
    }
}
```

参考资料

[1]周雍浩,徐金龙,李斌等.面向神威高性能多核处理器的并行编译优化方法[J].计算机工程,2022,48(09):130-138.DOI:10.19678/j.issn.1000-3428.0062139.

AdvancedCompiler

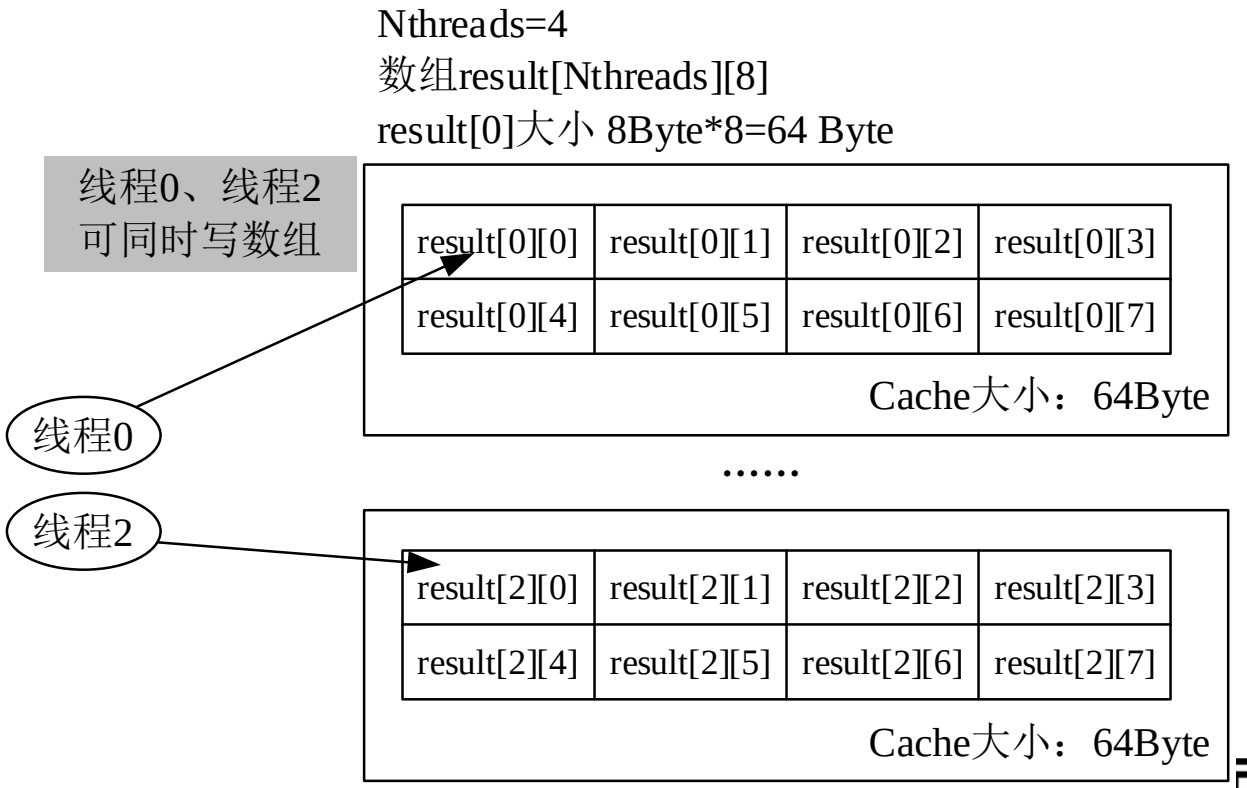
Tel: 13839830713

避免伪共享

嘉宾：王磊

分析伪共享

OpenMP在多核处理器间进行同步时常常需要共享一些变量，如用多线程同时对一个数组初始化时，多个线程对同一个数组进行修改，即使线程间从算法上并不需要共享变量，但是在实际执行时，若不同线程所需要赋值的地址处于同一个缓存行中，就会引起缓存冲突，严重降低程序性能，这就是伪共享。



数据填充避免伪共享

$\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx$ 数值计算求π

```
static long num_steps = 1000000;
double step;
int main(){
    int i;
    double temp,pi,result;
    step = 1.0/(double)num_steps;
    for(i=0;i<num_steps;i++){
        temp = (i+0.5)*step;
        result+=4.0/(1.0+temp*temp);
    }
    pi=step*result;
}
```

并行改写

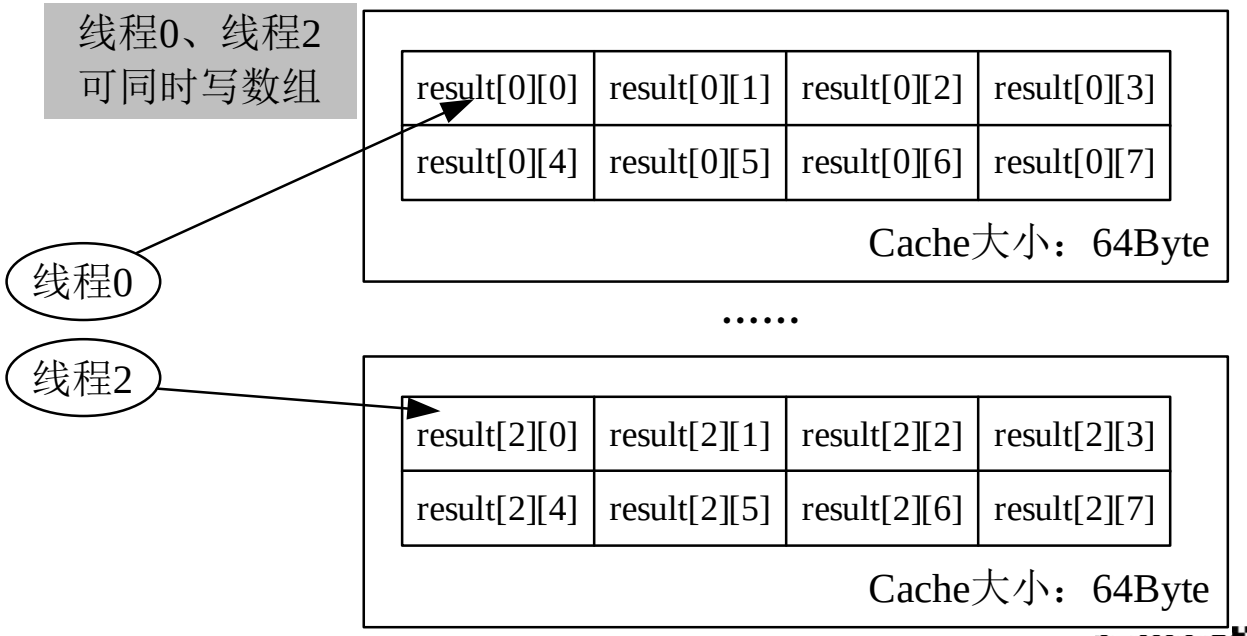
```
double result[Nthreads]={0.0};
#pragma omp parallel num_threads(Nthreads)
    {
        int i;
        int id = omp_get_thread_num();
        double temp;
        #pragma omp for
        for(i=0;i<num_steps;i++){
            temp = (i+0.5)*step;
            result[id] += 4.0/(1.0+temp*temp);
        }
    }
for(i=0;i<Nthreads;i++)
        pi += result[i];
    pi = step*pi;
```

数据填充避免伪共享


```
static long num_steps = 1000000;
double step;
int main(){
    int i;
    double pi = 0.0;
    double result[Nthreads][8]={0.0};
    step = 1.0/(double) num_steps;
    #pragma omp parallel num_threads(Nthreads){
        int i;
        int id = omp_get_thread_num();
        double temp;
        #pragma omp for
        for(i=0;i<num_steps;i++){
            temp = (i+0.5)*step;
            result[id][0] += 4.0/(1.0+temp*temp);
        }
    }
    for(i=0;i<Nthreads;i++)
        pi += result[i][0];
    pi = step * pi;
}
```

Nthreads=4
数组result[Nthreads][8]
result[0]大小 8Byte*8=64 Byte

线程0、线程2
可同时写数组



数据私有避免伪共享


```
static long num_steps = 1000000;
double step;
int main(){
    int i;
    double pi = 0.0;
    double result=0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel num_threads(Nthreads){
        int i;

        double temp;
        #pragma omp for reduction(+:result)
        for(i=0;i<num_steps;i++){
            temp = (i+0.5)*step;
            result+= 4.0/(1.0+temp*temp);
        }
    }

    pi = step*result;

}
```

归约操作是指反复地将运算符作用在一个变量或一个值上，并把结果保存在原变量中。归约子句reduction就是对前后有依赖的循环进行归约操作的并行化，即对一个或多个变量指定一个操作符，每个线程将创建变量列表中变量的一个私有副本，并将各线程变量的私有副本进行初始化。

参考资料

[1] 雷洪，胡许冰编著.多核并行高性能计算 OpenMP[M].北京：冶金工业出版社,2016.

[2]（德）海格，（德）韦雷因著.高性能科学与工程计算[M].北京：机械工业出版社,2014.

AdvancedCompiler

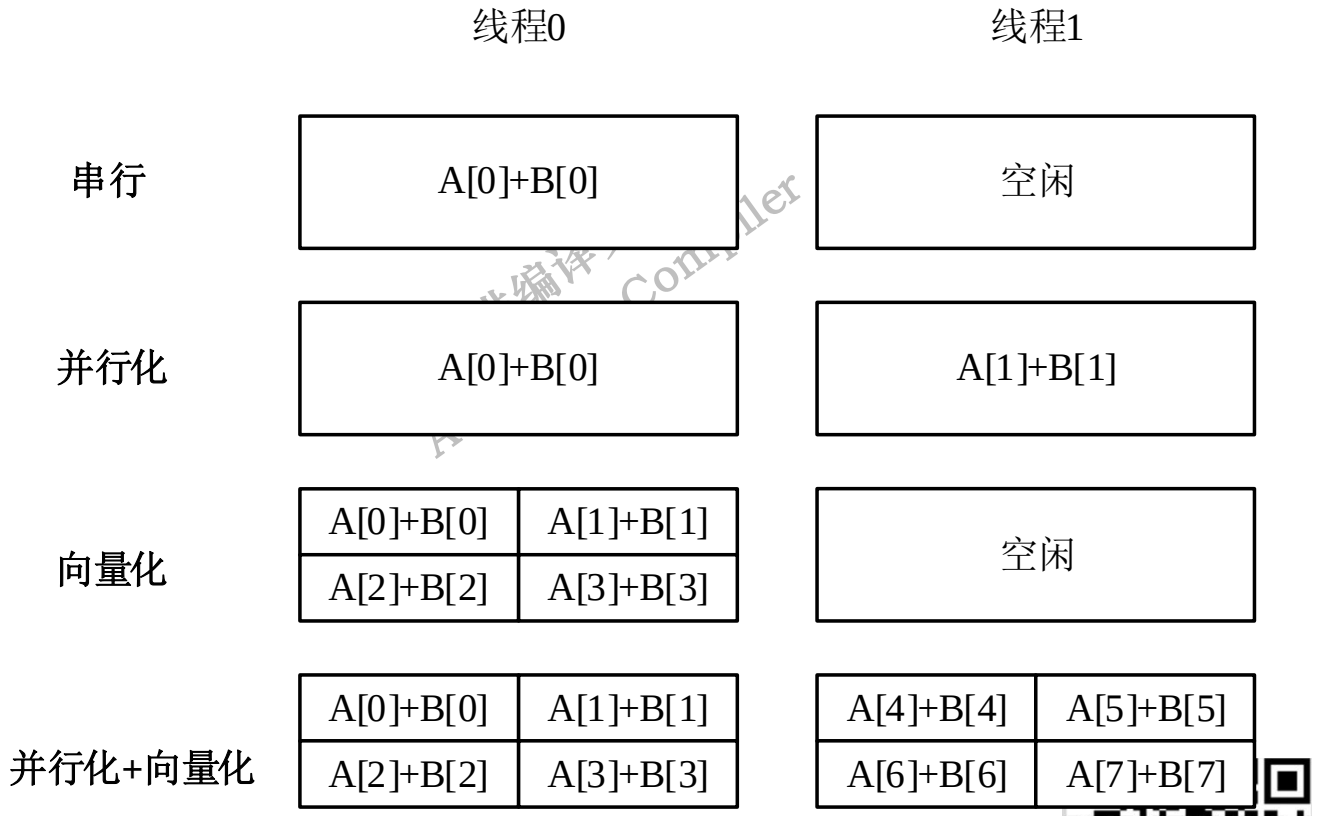
Tel: 13839830713

向量化指导命令

嘉宾：王磊

向量化指导命令

OpenMP指导语句中有两种支持向量化语句，分别为#pragma omp simd和#pragma omp for simd。其中#pragma omp simd指导语句对循环进行向量化是单线程的数据级并行，而#pragma omp for simd结合了for和simd两个指导命令同时进行并行化和向量化，具体是既将所有迭代的计算任务分配给各个线程，又进一步将每个线程执行的计算任务进行向量化。



向量化指导命令

simd 指导语句

#pragma omp simd [clause[[,]clause]……]
  loop-nest

clause :
  aligned(list[: alignment])
  collapse(n)
  if([simd:]omp-logical-expression)
  lastprivate([lastprivate-modifier :] list)
  linear(list[: linear-step])
  nontemporal(list)
  order([order-modifier :]concurrent)
  private(list)
  reduction([reduction-modifier ,] reduction-identifier : list)
  safelen(length)
  simdlen(length)

for simd 指导语句

#pragma omp for simd [clause[[,]clause]……]
  loop-nest

clause :
  aligned(list[: alignment])
  collapse(n)
  firstprivate(list)
  if([simd:]omp-logical-expression)
  lastprivate([lastprivate-modifier :] list)
  linear(list[: linear-step])
  nontemporal(list)
  nowait
  order([order-modifier :]concurrent)
  ordered[(n)]
  private(list)
  reduction([reduction-modifier ,] reduction-identifier : list)
  safelen(length)
  schedule ([modifier [, modifier] :] kind[, chunk_size])
  simdlen(length)

先进编译实验室
Advanced Compiler

向量化指导命令

simd 指导语句

```
#pragma omp simd
    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)
            for(int k=0;k<N;k++)
                C[i][j] += A[i][k]*B[k][j];
```

for simd 指导语句

```
#pragma omp parallel num_threads(4)
    #pragma omp for simd simdlen(8)
        for(int i=0;i<N;i++)
            for(int j=0;j<N;j++)
                for(int k=0;k<N;k++)
                    C[i][j] += A[i][k]*B[k][j];
```

参考资料

[1] 雷洪，胡许冰编著.多核并行高性能计算 OpenMP[M].北京：冶金工业出版社,2016.

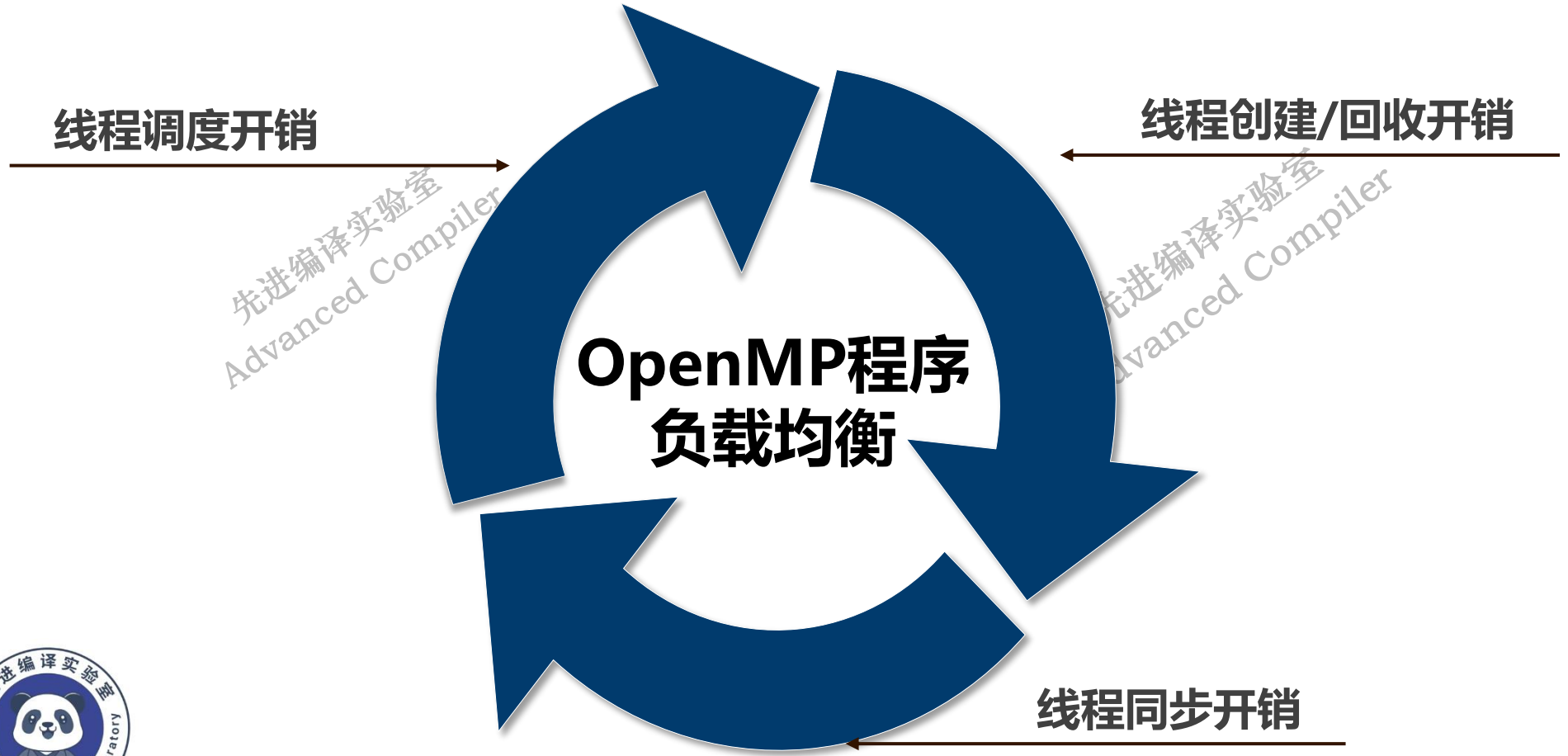
AdvancedCompiler

Tel: 13839830713

负载均衡优化

嘉宾：王磊

负载均衡优化



循环嵌套合并调度


```
#pragma omp parallel for private(i,j,k) shared(A,B,C) num_threads(4) collapse(2)
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      for(k=0;k<N;k++)
        C[i][j]+=A[i][k]*B[k][j];
```


子句collapse只能用于循环嵌套，它的具体语法格式为collapse(n)，其中参数n是一个整数，是指将与collapse子句最相邻的n层循环的迭代压缩合并在一起组成更大的任务调度空间，从而增加线程组调度空间中的循环总数，可调度迭代次数的增加有助于解决负载不均衡问题

线程调度配置策略

选择合适的线程调度策略，即对循环迭代采取静态或动态的方式分配到各个线程上并行执行，使得各个线程的工作量相当，以提升程序的性能。

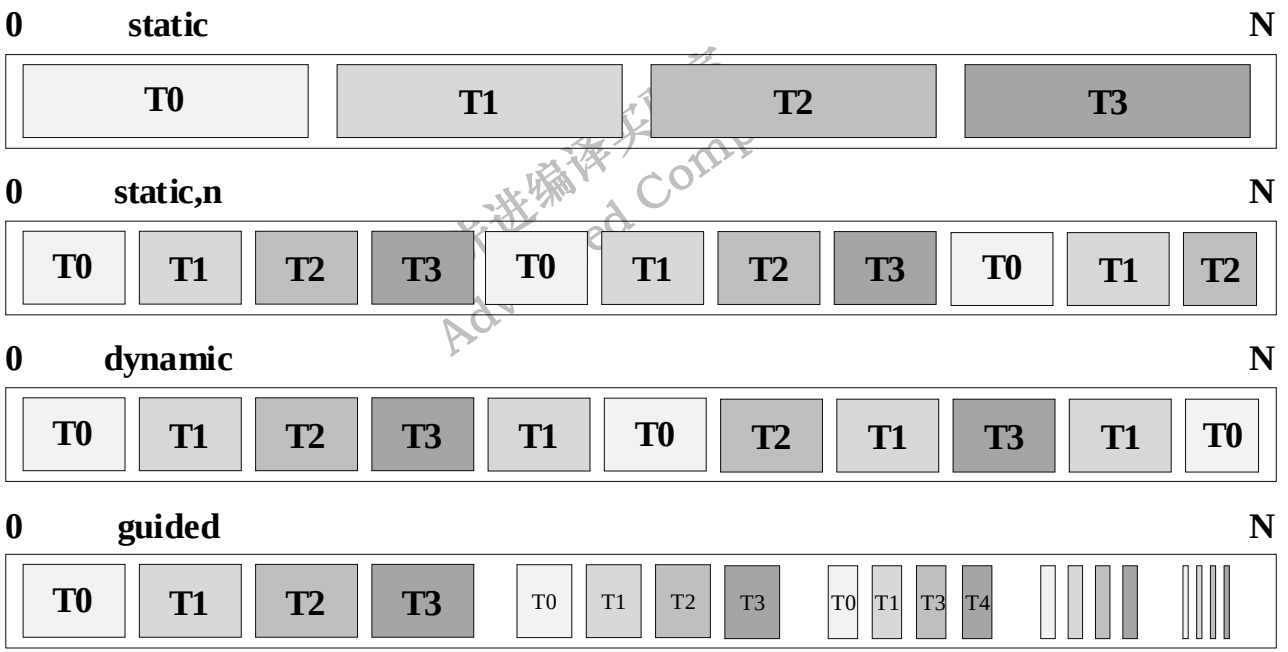
#pragma omp for schedule(schedule_name , chunk_size)

- **静态调度 static**：将所有的循环迭代划分为大小相等的调度块，使得迭代次数在线程上尽可能地均分。
- **动态调度 dynamic**：使用“先来先服务的排队申请策略”，当某个线程执行完当前调度块的任务时，就为其从调度块队列中分配一个新的调度块。

- **指导调度 guided**：强调的是任务分块动态变化，调度块的大小开始比较大，但会随程序的执行会按指数关系逐渐变小。
- **运行时调度 runtime**：指在运行时使用环境变量OMP_SCHDULE来确定上述三种调度策略的某一种。

线程调度配置策略

- **规则循环结构**：如矩阵乘法程序，建议使用带参数的静态调度以取得较好的性能。
- **递增型循环结构**：建议使用带参数的静态调度策略可在一定程度上缓解循环各迭代之间的计算量差距，以获得较好的性能。
- **递减循环结构**：首先使用指导调度，判断在开始时是否会因调度块较大会导致负载极为不均衡，若是则推荐使用动态调度，使得各线程调度的迭代块大小相当，从而实现有效的负载均衡。
- **随机循环结构**：推荐优先使用动态调度。



参考资料

[1] 雷洪，胡许冰编著.多核并行高性能计算 OpenMP[M].北京：冶金工业出版社,2016.

[2]刘胜飞,张云泉,孙相征.一种改进的OpenMP指导调度策略研究[J].计算机研究与发展,2010,47(04):687-694.

先进编译实验室
Advanced Compiler

王磊

2023年06月

AdvancedCompiler

Tel: 13839830713

线程数设置优化

嘉宾：王磊

串并行切换


```
#pragma omp parallel for private(i,j,k) shared(A,B,C) num_threads(8)  if(N>86)
  for(i=0;i<N;i++)
     for(j=0;j<N;j++)
        for(k=0;k<N;k++)
           C[i][j]+=A[i][k]*B[k][j];
```


串并行切换：OpenMP提供有if子句来自动切换程序的并行和串行，即在指导命令后增加if（N>Number）子句，其中当程序的计算量N大于阈值Number时，并行执行，否则串行执行。在本例中，阈值Number需要根据运行环境和线程设置数量等，不断调整矩阵规模N测试进行寻找。

选择合适的线程数

- **静态模式**：由优化人员确定并行区中线程的数量
  - omp_set_num_threads()
  - 添加子句num_threads
  - 使用环境变量OMP_NUM_THREADS
- **动态模式**
  - 采用默认模式（不指定线程数）
  - omp_set_dynamic() 设定并行区内线程的数目上限
- **嵌套模式**：函数omp_set_nested()启动或禁用嵌套并行，即在并行区中构建另一个并行区
- **条件模式**：利用子句if切换程序串行并行

先进编译实验室
Advanced Compiler

OpenMP程序设置的线程数量一定程度上影响了程序的性能，增加线程可以在特定时间段内完成更多的任务，但不断增加线程数量可能会使线程间同步等开销增加反而导致程序的性能降低，OpenMP提供的线程设置模式，帮助开发人员选择程序的并行线程数量。

```
omp_set_dynamic(1);
omp_set_num_threads(8);
#pragma omp parallel for private(i,j,k) shared(A,B,C) if(N>86)
 for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        for(k=0;k<N;k++)
            C[i][j]+=A[i][k]*B[k][j];
```

选择合适的线程数


```
#pragma omp parallel for private(i,j,k) shared(A,B,C) if(N>86) num_threads(12)
for (int ii = 0; ii < N; ii += BLOCK_SIZE)
        for (int jj = 0; jj < N; jj += BLOCK_SIZE)
                for (int kk = 0; kk < N; kk+=BLOCK_SIZE)
                        for (i = ii; i < min(ii+BLOCK_SIZE,N); i++)
                                for (k = kk; k < min(kk+BLOCK_SIZE,N); k++){
                                        int s=A[i][k];
                                        for ( j = jj; j < min(jj+BLOCK_SIZE,N); j++){
                                                C[i][j]+= s*B[k][j];
                                        }
                                }
```

在实际应用中，并行程序可能会进行程序优化调整或发生运行环境改变，由于不同程序的最优性能对应的线程数是不同的，需要重新设置线程数以达到性能最优，因此根据程序特征和运行环境来设置合适的线程数十分重要。

参考资料

[1] 雷洪，胡许冰编著.多核并行高性能计算 OpenMP[M].北京：冶金工业出版社,2016.

AdvancedCompiler

Tel: 13839830713

避免隐式同步

嘉宾：王磊

分析隐式同步


```
int i,A[N];
   #pragma omp parallel private(i) num_threads(4){
         #pragma omp for
         for(i=0;i<N;i++){
            A[i] = i;
            printf("thread id = %d : A[%d] = %d\n",omp_get_thread_num(),i,A[i]);
         }
         #pragma omp master
         printf("thread id = %d : All work done\n",omp_get_thread_num());
   }
}
pi = step*pi;
}
```

打印结果

```
thread id = 0 : A[0] = 0
thread id = 0 : A[1] = 1
thread id = 0 : A[2] = 2
……
thread id = 0 : All work done
```

OpenMP中有显式和隐式两种同步方式，使用指导语句#pragma omp barrier是进行显式同步的一种方式，其不需要对任何代码进行修饰，而是要求并行区内所有线程都执行到该指导语句处，才能继续执行后续代码。

而隐式同步是指执行parallel、for、sections等指导语句时，并行代码的结束处会自动添加一个同步点以进行隐式的线程同步，以保证程序执行结果的正确性。

消除隐式同步


```
#pragma omp parallel default(none) shared(N,x,y,z,scale) private(f,i,j) num_threads(8){
    f= 1.0;
    #pragma omp for nowait
    for(i=0;i<N;i++){
        z[i] = x[i] + y[i];
        complexcompute(omp_get_thread_num());
    }
    #pragma omp single
    complexcompute(200);
    #pragma omp single
    scale = sum(z,0,N) + f;
    }
```

并行程序并不总是需要同步，有时后续的任务不需要等待前面任务的完成，此时如果存在隐式同步则会浪费线程资源。为了让先完成计算任务的线程继续工作，可使用nowait子句消除隐式同步

消除隐式同步


```
#pragma omp parallel private(i,j,k) shared(A,B,C) num_threads(4){
    #pragma omp for
    for(i=0;i<N;i++){
        printf("线程%d执行C[%d][x]计算\n",omp_get_thread_num(),i);
        for(j=0;j<N;j++)
            for(k=0;k<N;k++)
                C[i][j]+=A[i][k]*B[k][j];
    }
    #pragma omp for reduction(+:sum) nowait
    for(i=0;i<N;i++){
        printf("线程%d执行C[%d][x]求和\n",omp_get_thread_num(),i);
        for(j=0;j<N;j++)
            sum+=C[i][j];
    }
}
```

先进编译实验室
Advanced Compiler

nowait子句的使用不是无条件的，需要在保证程序正确性的前提下才能使用。

Question: 左边的程序是否能保证结果正确?

结果正确√，原因?

1、使用默认的静态调度策略

2、计算和求和这两个循环有相同的迭代次数

3、循环绑定到同一个并行区

参考资料

[1] 雷洪，胡许冰编著.多核并行高性能计算 OpenMP[M].北京：冶金工业出版社,2016.

AdvancedCompiler

Tel: 13839830713

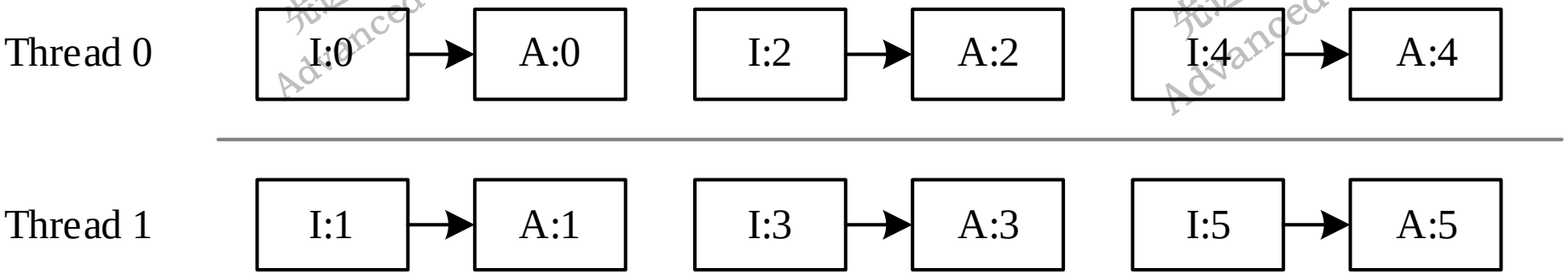
流水并行优化

嘉宾：王磊

流水并行简介

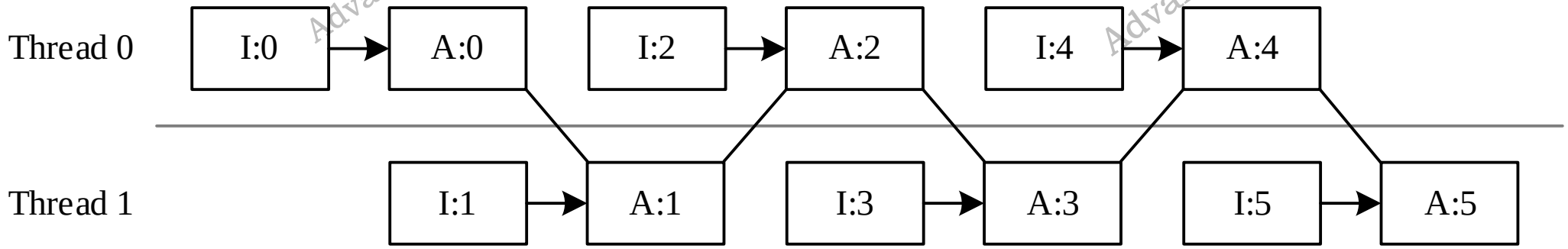
根据循环所蕴含并行性的不同，可以将循环分为DOALL循环和DOACROSS循环，不同类型循环蕴含的并行性和发掘其并行性的难度也完全不同。

DOALL循环中的各次迭代之间不存在依赖关系，迭代间不需要同步是一种完全并行。



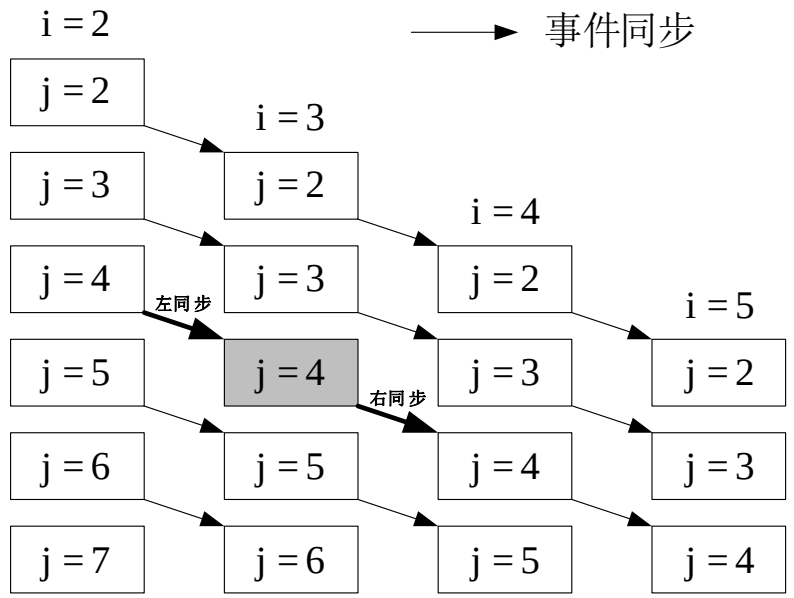
流水并行简介

DOACROSS循环中存在跨迭代的依赖关系，对于无法消除迭代间依赖的循环，仍然有可能通过迭代间的同步实现循环的并行执行。DOACROSS循环可根据跨迭代依赖距离可以细分为规则DOACROSS循环和不规则DOACROSS循环。规则DOACROSS循环中依赖关系的规则性，使得其更容易通过流水并行的方式进行并行。



流水并行示例

```
for(int j=1;j<N2-1;j++){
        sync_left();
        #pragma omp for schedule(static) nowait
        for(int i=1;i<Ni-1;i++){
                a[i][j]=0.25*(a[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1]);
        }
        sync_right();
}
```



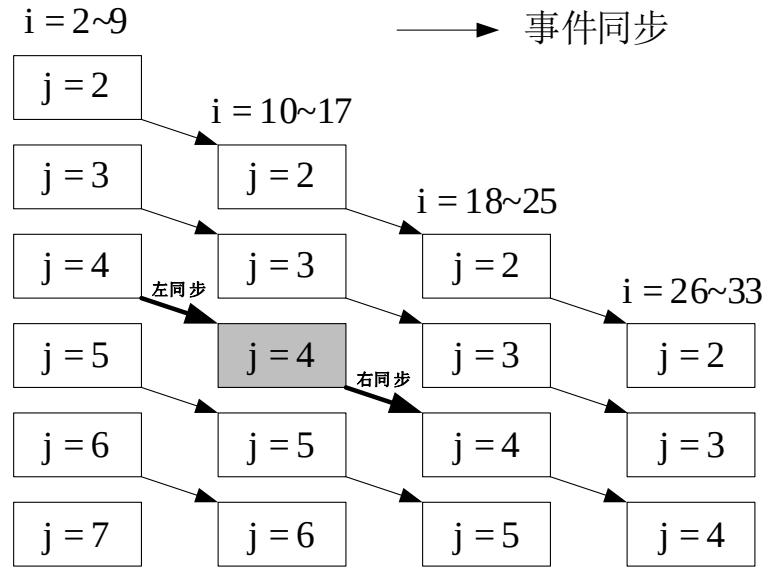
针对有限差分松弛法FDR循环示例进行流水并行的步骤如下：

(1)判断目的循环是否为DOACROSS循环且循环层数是否大于2。

(2)选择计算划分层和循环分块层。　选择j层为计算划分层，i层为循环

(3)实现线程同步。　使用计数信号量机制来实现

(4)线程负载优化。　考虑到实际情况中线程数有限进行负载优化



流水并行示例

左同步

```
int isync[256],mthreadnum,iam;
#pragma omp threadprivate(mthreadnum,iam)
void sync_left(){
        int neighbour;
        if(iam>0&&iam<=mthreadnum){
                neighbour=iam-1;
                while(isync[neighbour]==0) {
                        #pragma omp flush(isync)
                }
                isync[neighbour]=0;
                #pragma omp flush(isync,a)
        }
}
```

右同步

```
void sync_right(){
        if(iam<mthreadnum){
                while(isync[iam]==1) {
                        #pragma omp flush(isync)
                }
                #pragma omp flush(isync,a)
                isync[iam%(mthreadnum-1)]=1;
                #pragma omp flush(isync)
        }
}
```

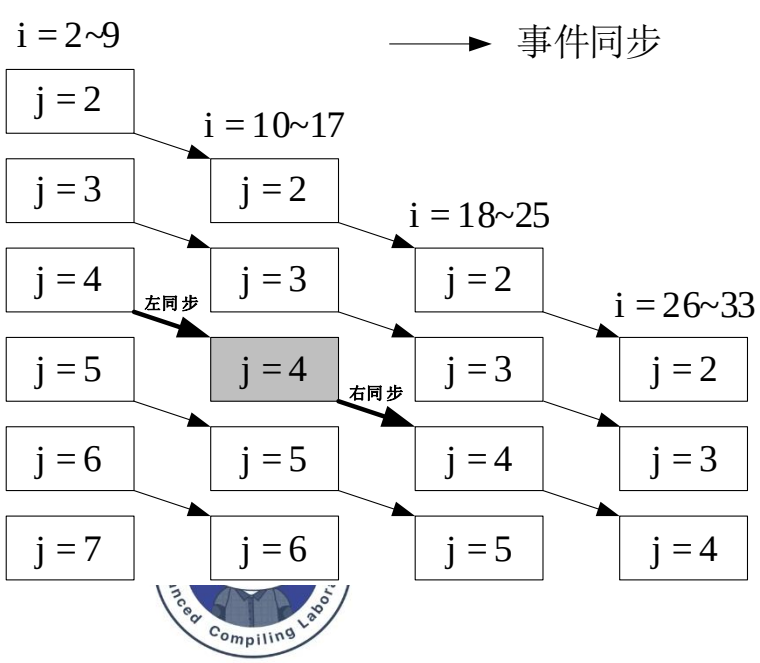

先进编译实验室
Advanced Compiler

流水并行粒度

流水计算粒度：同一线程两次同步之间的计算工作量大小，通常可以使用循环交换和循环分块这两种循环变换方式来完成流水粒度的调节，以平衡并行粒度和同步代价两者的关系。

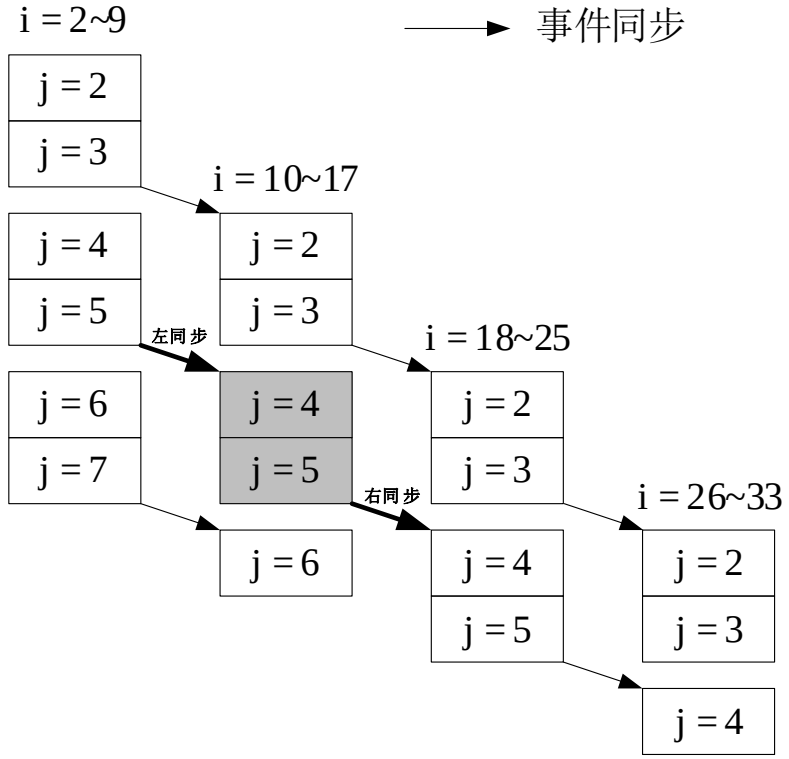
细粒度流水是指将被划分的循环层放置在循环嵌套的较内层，较高的同步代价。

粗粒度流水是指将计算划分的循环层放到循环嵌套的较外层，同步代价相对较小。



循环分块

增大粒度



流水并行粒度

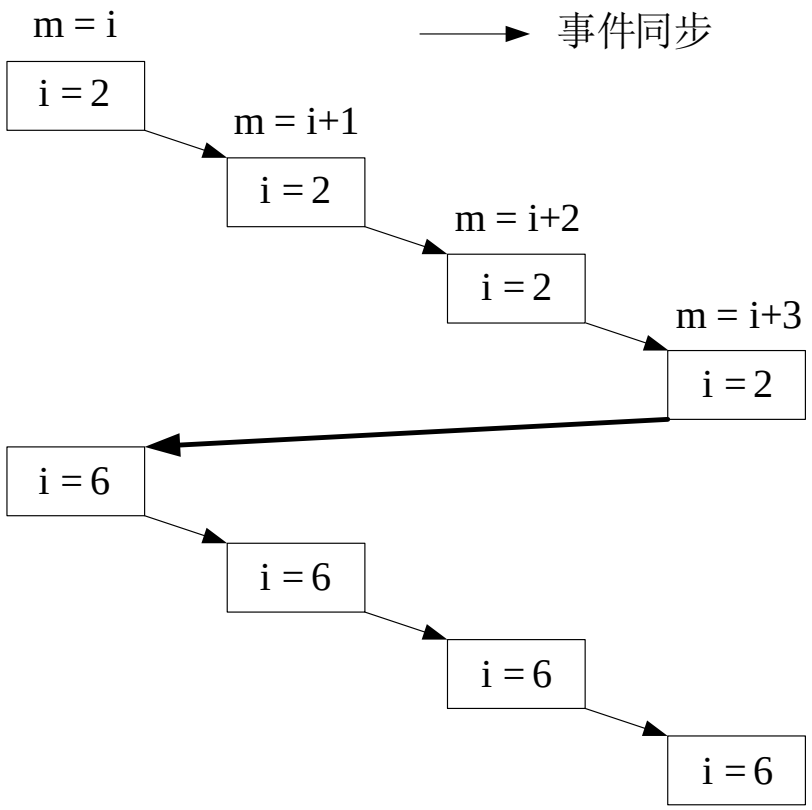
循环分块增大流水粒度

```
int b=9;//分块大小
for(int i=1;i<N1-1;i=i+b){
        sync_left();
        #pragma omp for schedule(static) nowait
        for(int j=1;j<N2-1;j++){
                for(int m=i;m<min(i+b,N1-1);m++){
                        a[m][j]=0.25*(a[m-1][j]+a[m][j-1]+a[m+1][j]+a[m][j+1]);
                }
        }
        sync_right();
}
```

流水并行粒度


```
int b=4;//分块大小
for(int j=1;j<N2-1;j++){
    sync_left();
    for(int i=1;i<N1-1;i+=b){
        for(int m=i;m<min(i+b,N1-1);m++){
            a[m][j]=0.25*(a[m-1][j]+a[m][j-1]
                    +a[m+1][j]+a[m][j+1]);
        }
    }
    sync_right();
}
```

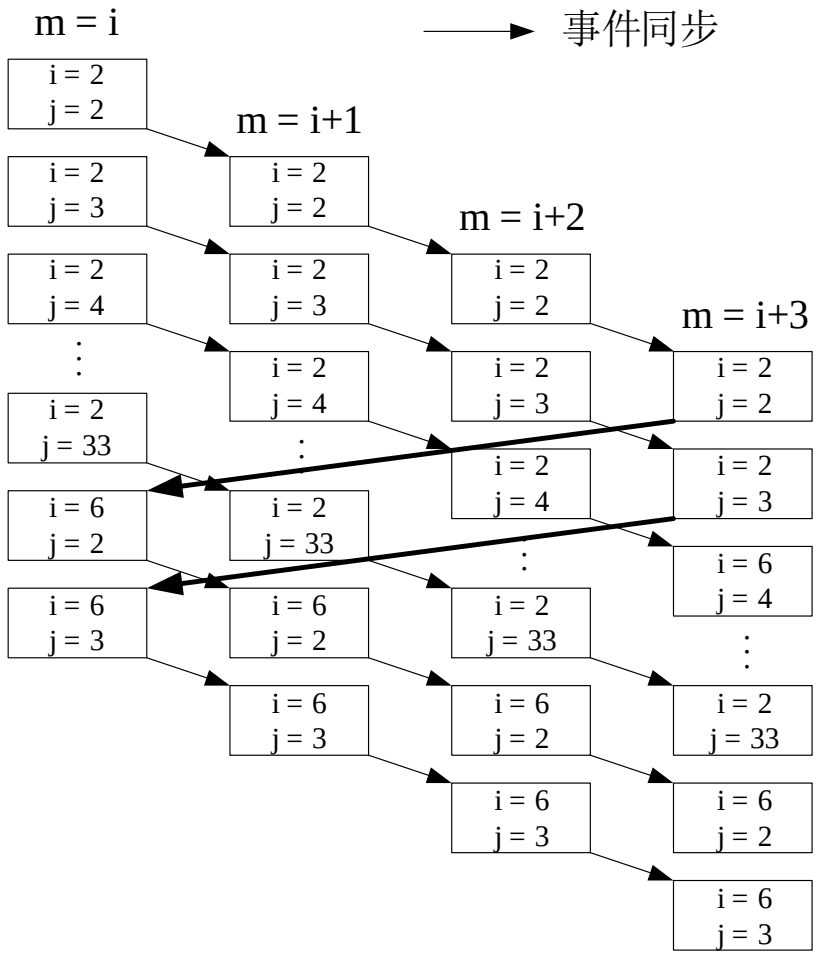
循环分段减小流水粒度



流水并行粒度

循环分段+循环交换减小流水粒度

```
int b=9;//分块大小
for(int j=1;j<N2-1;j=j+b){
        sync_left();
        #pragma omp for schedule(static) nowait
        for(int i=1;i<N1-1;i++){
                for(int m=j;m<min(j+b,N2-1);m++){
                        a[i][m]=0.25*(a[i-1][m]+a[i][m-1]+a[i+1][m]+a[i][m+1]);
                }
        }
        sync_right();
}
```



流水并行粒度

除了选择增大粒度和减小粒度之外，分块大小的选择也很关键，通常优化人员会建立相应的代价模型来计算最优的分块大小。在DOACROSS流水并行时需要综合考虑的代价因素包括，计算划分层和循环分块层的迭代数、线程数目、单个分块的执行时间、同步开销时间以及线程执行的分块数等，因此可以对这些因素建立代价模型，最后通过测试评估后得到并行优化后的最优分块大小。

对于本示例通过建立代价模型得到结论，通常当单个分块的同步开销时间与执行时间之比大于1时，采用增大粒度的策略，分块大小设置为两者比值取整时，往往能够对流水并行予以一定程度的优化，否则需要采用减小粒度的策略。

参考资料

[1] 刘晓娴.面向共享存储结构的并行编译优化技术研究[D].解放军信息工程大学,2013.

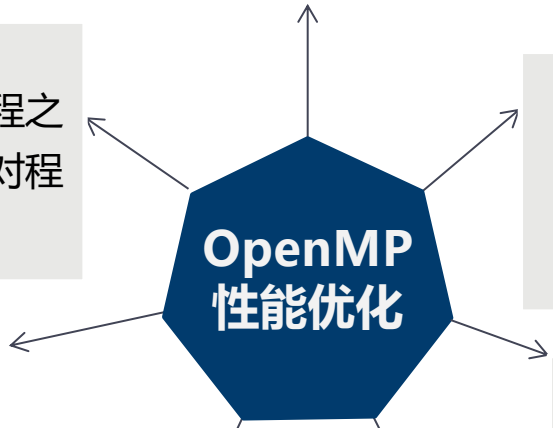
AdvancedCompiler

Tel: 13839830713

OpenMP程序优化

嘉宾：王磊

OpenMP程序优化



并行区重构

包括并行区扩张以及合并，对OpenMP程序中循环中的并行区进行重构，可以减少并行区产生的开销，达到提升程序性能的目的。

流水并行优化

将循环的各次迭代分配给不同的线程后，线程之间通过流水执行来获得更好的并行性，从而对程序进行优化。

避免伪共享

针对程序中存在的伪共享问题，通过将变量数据在存储器或缓存行中保持边界对齐，或者数据线程私有化的方法来进行优化。

避免隐式同步

针对程序中部分指导语句结构结束时不必要的隐式同步，通过优化手段消除以达到提升程序性能的目的。

循环向量化

将多线程并行与程序语句向量执行结合，以达到更好的加速效果。

线程数设置优化

说明了如何选择设置和选择并行程序的线程数使得程序的性能达到最好

负载均衡优化

合理利用调度策略、循环转换以及线程数设置等方式使得OpenMP程序负载尽可能地均衡，提升程序的性能。

参考资料

[1] 雷洪，胡许冰编著.多核并行高性能计算 OpenMP[M].北京：冶金工业出版社,2016.

[2]（德）海格，（德）韦雷因著.高性能科学与工程计算[M].北京：机械工业出版社,2014.

[3]周雍浩,徐金龙,李斌等.面向神威高性能多核处理器的并行编译优化方法[J].计算机工程,2022,48(09):130-138.DOI:10.19678/j.issn.1000-3428.0062139.

[4]刘胜飞,张云泉,孙相征.一种改进的OpenMP指导调度策略研究[J].计算机研究与发展,2010,47(04):687-694.

[5] 刘晓娴.面向共享存储结构的并行编译优化技术研究[D].解放军信息工程大学,2013.

先进编译实验室
王磊
2023年06月

AdvancedCompiler

Tel: 13839830713