中国科学院大学
University of Chinese Academy of Sciences

Introduction to OpenMP
Part2. *Core Features*

人工智能学院
缪青海
miaoqh@ucas.ac.cn

Content

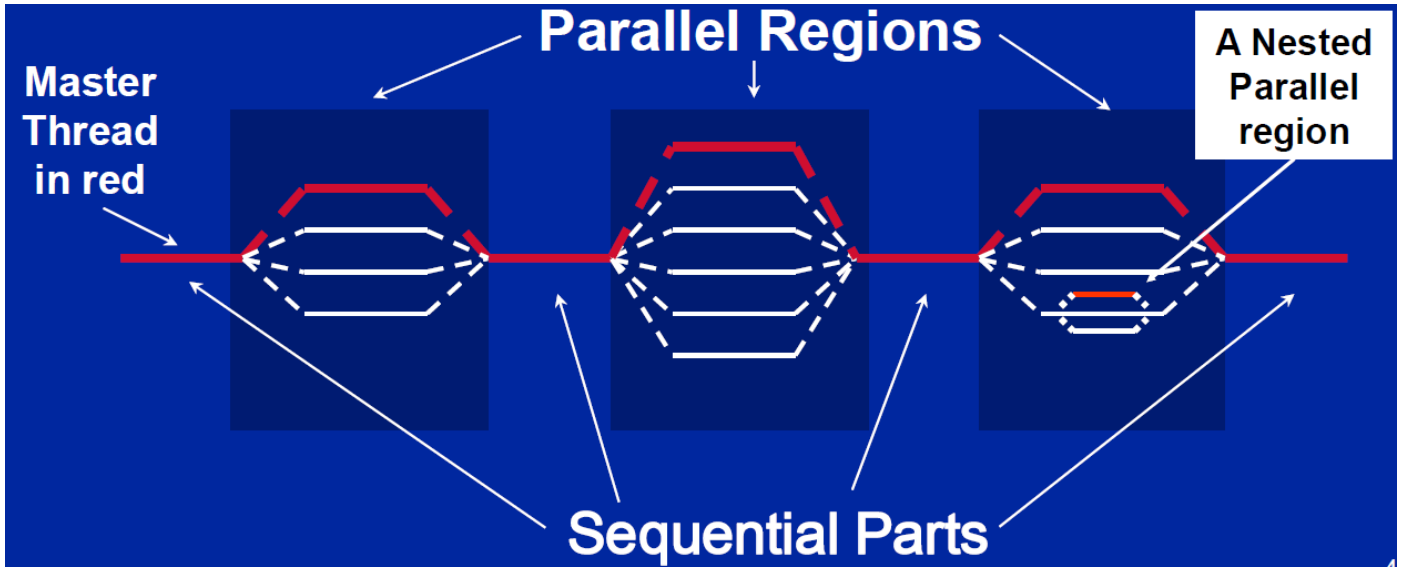
- Getting started with OpenMP
- **Core features of OpenMP**
- Work with OpenMP
- Advanced topics on OpenMP

Core features of OpenMP

- **Creating threads**
  - the Pi program
- **Synchronization**
  - Pi program revisited
- **Parallel Loops**
  - making the Pi program simple

OpenMP Programming Model

- **Fork-Join Parallelism**:
  - Master thread spawns a team of threads as needed.
  - Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



Thread Creation: Parallel Regions

- Create threads in OpenMP with the parallel construct.
- For example, create 4 thread parallel region:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
      int ID = omp_get_thread_num();
      pooh(ID,A);
}
```

- Each thread calls pooh(ID,A) for ID = 0 to 3

Thread Creation : Parallel Regions

- Create threads in OpenMP with the parallel construct.
- For example, create 4 thread parallel region:

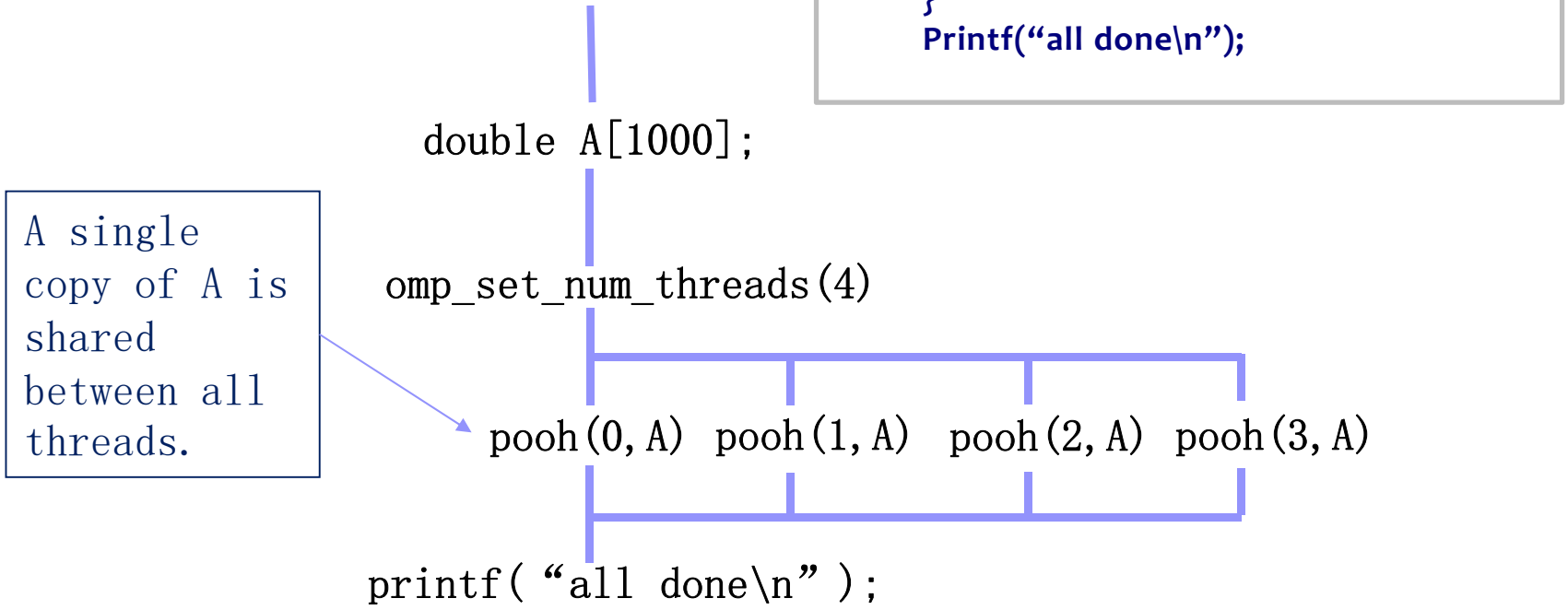
```
double A[1000];

#pragma omp parallel num_threads(4)
{
      int ID = omp_get_thread_num();
      pooh(ID,A);
}
```

- Each thread calls pooh(ID,A) for ID = 0 to 3

Thread Creation : Parallel Regions

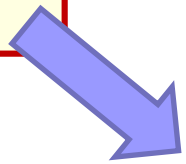
- Each thread executes the same code redundantly.

```
double A[1000];
#pragma omp parallel num_threads(4)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
Printf("all done\n");
```



what the compiler does

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```



```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
        &tid[i],0,thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region.

Only three threads are created because the last parallel section will be invoked from the parent thread.

Test1: PI by Numerical Integration

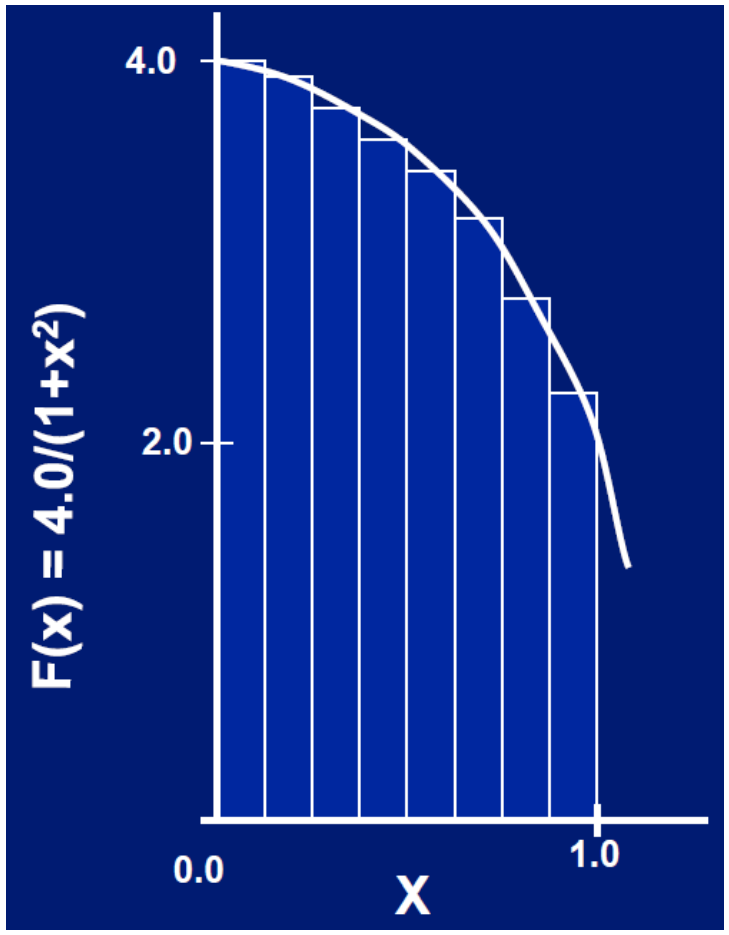
Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

Approximate as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.



Test 1: Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Code: test1_Pi/pi.c

Test1 aim

- Create a parallel version of the pi program using a parallel construct. Pay close attention to :
  - **Shared versus private variables.**
- **Runtime library routines:**
  - **int omp_get_num_threads();** — Number of threads in the team
  - **int omp_get_thread_num();** — Thread ID or rank
  - **double omp_get_wtime();** — Time in Seconds since a fixed point in the past

Test1-1: A simple parallel PI Program

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
int main ()
{
   int i, nthreads; double pi, sum[NUM_THREADS];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
        int i, id,nthrds; double x;
        id = omp_get_thread_num();
        sum[id]=0.0
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id; i< num_steps; i= i+nthrds){
             x = (i+0.5)*step;
             sum[id] += 4.0/(1.0+x*x);
        }
   }
   for(i=0, pi=0.0; i<nthreads; i++)  pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread copy value, avoid writing conflicts to a address.

trick in SPMD to create a cyclic distribution of loop iterations.

Test1-1 : Algorithm strategy

- **The SPMD (Single Program Multiple Data) design pattern:**
  - Run the same program on P processing elements where P can be arbitrarily large.
  - Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.
- **The most commonly used pattern in the history of parallel programming.**
- **Code:**
  - **test1_Pi/solutions/ pi_spmd_simple.c**

Test1-1 : Results

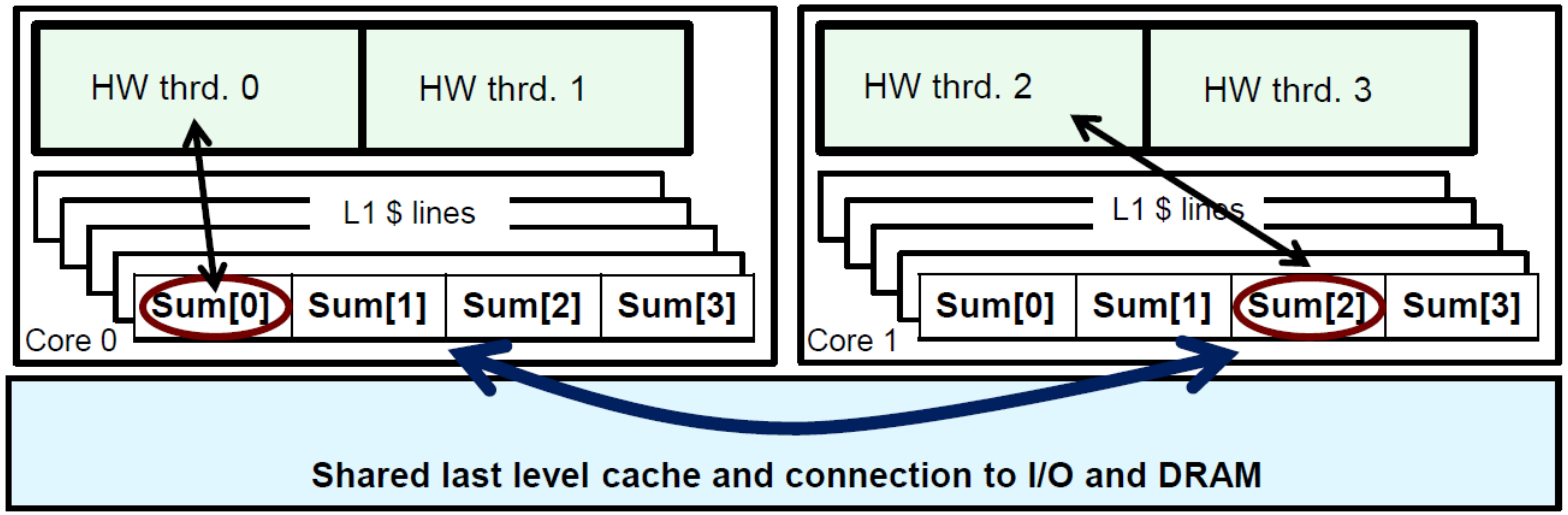
- **CPU:**
  - Intel Core i5 1.7 Ghz
- **Memory:**
  - 4G DDR3 1.333 Ghz
- **OS:**
  - Apple OS X 10.7.3
- **Complier:**
  - Intel icpc

作业示例

threads	Time(s)
1	1.86
2	1.03
3	1.08
4	0.97

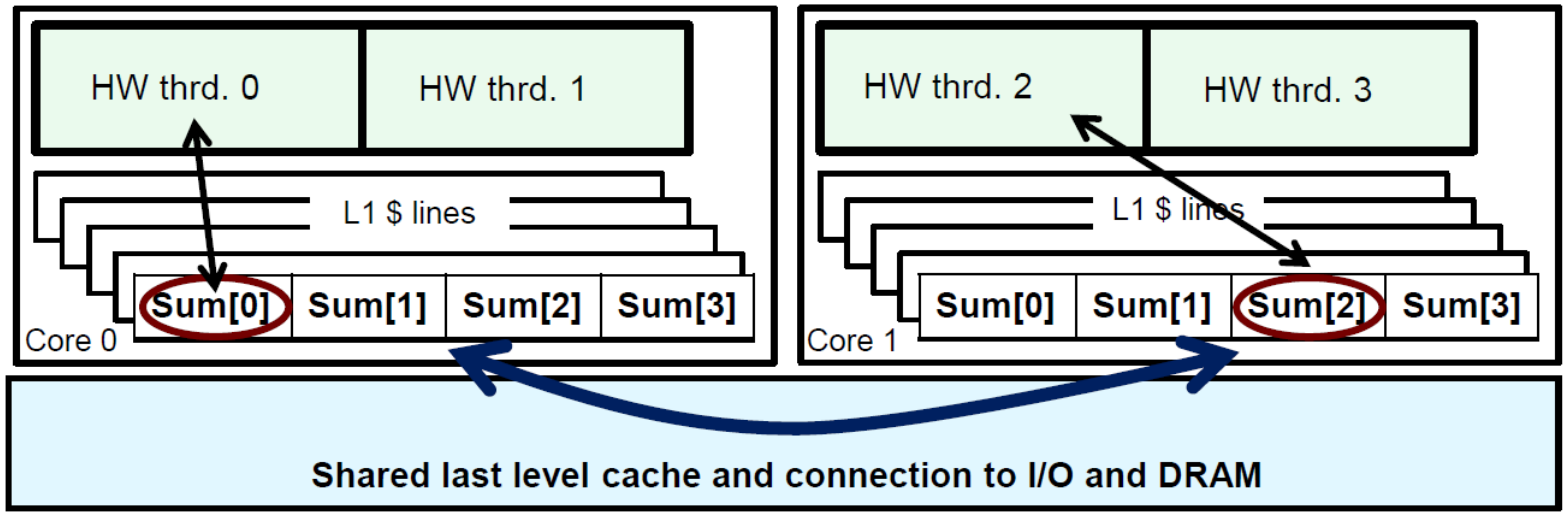
Test1-1 : False sharing

- **Why such poor scaling? False sharing!**
  - If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads …
  - This is called **"false sharing"**.



Test1-1 : False sharing

- **Why such poor scaling? False sharing!**
  - If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
  - Solution: Pad arrays so elements you use are on distinct cache lines.



Test1-1 : A simple parallel PI Program

```
#include <omp.h>
static long num_steps = 100000; double step;
#define PAD 8 // assume 64 byte L1 cache line size
#define NUM_THREADS 2
int main ()
{
   int i, nthreads; double pi, sum[NUM_THREADS][PAD];
   step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
        int i, id,nthrds; double x;
        id = omp_get_thread_num();
       nthrds = omp_get_num_threads();
       if (id == 0) nthreads = nthrds;
      for (i=id, sum[id]=0.0; i< num_steps; i= i+nthrds){
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++)  pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line.

Test1-1 : is padding necessary?

- Padding arrays requires deep knowledge of the cache architecture.
- Move to a machine with different sized cache lines and your software performance falls apart.
- There has got to be a better way to deal with false sharing.

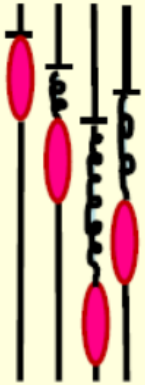
Core features of OpenMP

- **Creating threads**
  - the Pi program
- **Synchronization**
  - Pi program revisited
- **Parallel Loops**
  - making the Pi program simple

Synchronization

- Bringing one or more threads to a well defined and known point in their execution.

Barrier: each thread wait at the barrier until all threads arrive.



Mutual exclusion: Define a block of code that only one thread at a time can execute.

Synchronization

- High level synchronization:
  - – critical
  - – atomic
  - – barrier
  - – ordered
- Low level synchronization
  - – flush
  - – locks (both simple and nested)

Synchronization is used to impose order constraints and to protect access to shared data.

Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```
#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier

        B[id] = big_calc2(id, A);
}
```

Synchronization: Critical

- **Mutual exclusion:** Only one thread at a time can enter a critical region.

Threads wait their turn – only one at a time calls consume()

```
float  res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     for(i=id;i<niters;i+=nthrds){
          B =  big_job(i);
#pragma omp critical
          res += consume (B);
     }
}
```

Synchronization: Atomic (*basic form*)

- **Atomic** provides mutual exclusion but only applies to the update of a memory location.

```
#pragma omp parallel
{
        double tmp, B;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
        X +=  tmp;
}
```

The statement inside the atomic must be one of the following forms:

- x binop= expr
- x++
- ++x
- x—
- --x

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

Test1-2

- Test1-1 *used an array* to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
  - – Non-shared data in the same cache line so each update invalidates the cache line … in essence "sloshing independent data" back and forth between threads.
- Modify your "pi program" from test1-1 to avoid false sharing due to the sum array.
- Code: test1_Pi/solutions/ pi_spmd_final.c

Test1-2: A simple parallel PI Program

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
int main ()
{
   int i, nthreads; double pi, sum[NUM_THREADS];
   step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id,nthrds; double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0; i< num_steps; i= i+nthrds){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++)  pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Recall

Only one thread copy value, avoid writing conflicts to a address.

trick in SPMD to create a cyclic distribution of loop iterations.

Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;   double step;
#define NUM_THREADS 2
void main ()
{           double pi; step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
            int i, id,nthrds; double x, sum;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0) nthreads = nthrds;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
            }
            #pragma omp critical
            pi += sum * step;
    }
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Must protect summation into pi in a critical region so updates don't conflict.

Test1-2: results

```
#include <omp.h>
static long num_steps = 100000;   double step;
#define NUM_THREADS 2
void main ()
{          double pi; step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
           int i, id,nthrds; double x, sum;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0) nthreads = nthrds;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                      x = (i+0.5)*step;
                      sum += 4.0/(1.0+x*x);
           }
           #pragma omp critical
           pi += sum * step;
    }
}
```

Create a sc
each thread
accumulate

No array, so no false sharing.

Must protect summation into pi in a critical region so updates don't conflict.

threads	1st SPMD	1st SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

Using an atomic to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;   double step;
#define NUM_THREADS 2
void main ()
{           double pi;  step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
            int i, id,nthrds; double x, sum;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0) nthreads = nthrds;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
            }
            sum = sum*step;
            #pragma atomic
            pi += sum;
    }
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Must protect summation into pi so updates don't conflict.

Core features of OpenMP

- **Creating threads**
  - the Pi program
- **Synchronization**
  - Pi program revisited
- **Parallel Loops**
  - making the Pi program simple

SPMD vs. Worksharing

- A parallel construct by itself creates an SPMD program …
  i.e., each thread redundantly executes the same code.
- How do we split up pathways through the code between threads within a team?
  - This is called worksharing
    - – Loop construct
    - – Sections/section constructs
    - – Single construct
    - – Task construct

Loop Worksharing constructs

- The loop worksharing construct splits up loop iterations among the threads in a team

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

The variable i is made "private" to each thread by default. You could do this explicitly with a "private(i)" clause

Loop Worksharing constructs

Serial

```
for(i=0;i<N;i++)  { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)  { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++)  { a[i] = a[i] + b[i];}
```

Loop Worksharing constructs

- **The schedule clause: affects how loop iterations are mapped onto threads.**
  - schedule(**static** [,chunk])
    - – Deal-out blocks of iterations of size "chunk" to each thread.
  - schedule(**dynamic**[,chunk])
    - – Each thread grabs "chunk" iterations off a queue until all iterations have been handled.

Loop Worksharing constructs

- **The schedule clause**:
  - schedule(**guided**[,chunk])
    - – Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - schedule(**runtime**)
    - – Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
  - schedule(**auto**)
    - – Schedule is left up to the runtime to choose (does not have to be any of the above).

Loop Worksharing constructs

- The schedule clause: when to use?

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can "learn" from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Loop Worksharing constructs

- OpenMP shortcut:
  - Combined parallel/worksharing construct

```
double res[MAX]; int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```

=

```
double res[MAX]; int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
```

Nested loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
          .....
  }
}
```

Number of loops to be parallelized, counting from the outside

- Will form a single loop of length NxM and then parallelize that.

Reduction

- How do we handle this case?

```
double  ave=0.0, A[MAX];   int i;
 for (i=0;i< MAX; i++) {
      ave + = A[i];
 }
 ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed.
- This is a very common situation … called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:
  - reduction (op : list)
- Inside a parallel or a work-sharing construct:
  - – A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+").
  - – Updates occur on the local copy.
  - – Local copies are reduced into a single value and combined with the original global value.

Reduction

- OpenMP reduction clause:
  - reduction (op : list)
- The variables in "list" must be shared in the enclosing parallel region.

```
double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
        ave + = A[i];
 }
 ave = ave/MAX;
```

Reduction: operands/initial-values

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Test1-3: Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct.
- Your goal is to minimize the number of changes made to the serial program.

Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Test1-3 : Pi with loop and reduction

```
#include <omp.h>
static long num_steps = 100000;double step;
int main ()
{
      int i; double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
          double x;
          #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
              x = (i+0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
          }
      }
      pi = step * sum;
}
```

Create a team of threads

a scalar local to each thread

setting up a reduction into sum.

Code: test1_Pi/solution/pi_loop.c

references

- Tim Mattson, A “Hands-on” Introduction to OpenMP, Intel Corp.
- Mark Bull, Parallel Programming with OpenMP, EPCC, University of Edinburgh, UK.
- 迟学斌等，并行计算与实现技术，科学出版社。