# Introduction to OpenMP

## Part3. *Working with OpenMP*

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

# Working with OpenMP

- Synchronization 线程同步
- Data environment 数据属性
- Linked lists traversal 链表遍历

# Synchronization: Barrier

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

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

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

# Master Construct

- The master construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    { exchange_boundaries(); }
    #pragma omp barrier
    do_many_other_things();
}
```

master默认
没有barrier

# Single Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

与master区别：
默认有barrier

```
#pragma omp parallel
{
     do_many_things();
     #pragma omp single
     { exchange_boundaries(); }
     do_many_other_things();
}
```

# Sections Construct

- The *Sections* worksharing construct gives a different structured block to each thread.
- By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            x_calculation();
        #pragma omp section
            y_calculation();
        #pragma omp section
            z_calculation();
    }
}
```

# Synchronization: Lock routines

- A lock implies a memory fence (a “flush”) of all thread visible variables.

比critical等更底层

- Simple Lock:
  - A simple lock is available if it is unset.
  - –omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- Nested Locks:

# Synchronization: Lock routines

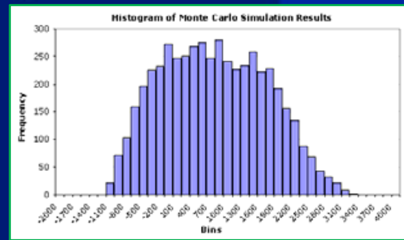
- A lock implies a memory fence (a "flush") of all thread visible variables.
- Simple Lock:
- Nested Locks:
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
  - –omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

# Synchronization: Simple Lock

```
#pragma omp parallel for
 for(i=0;i<NBUCKETS; i++){
      omp_init_lock(&hist_locks[i]);    hist[i] = 0;
 }
 #pragma omp parallel for
 for(i=0;i<NVALS;i++){
    ival = (int)  sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
       hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
  }

for(i=0;i<NBUCKETS; i++)
  omp_destroy_lock(&hist_locks[i]);
```

One lock per element of hist




Enforce mutual exclusion on update to hist array

Free-up storage when done.

# Runtime Library routines

- Runtime environment routines – 1 :
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Are we in an active parallel region?
    - omp_in_parallel()

# Runtime Library routines

- Runtime environment routines – 2 :
  - Do you want the system to dynamically vary the number of threads from one parallel construct to another?
    - omp_set_dynamic(), omp_get_dynamic()
  - How many processors in the system?
    - omp_num_procs()

# Runtime Library routines

- **Use fixed number of threads in a program:**

```
#include <omp.h>
void main()
{     int num_threads;
      omp_set_dynamic( 0 );
      omp_set_num_threads( omp_num_procs() );
      #pragma omp parallel
      {     int id=omp_get_thread_num();
      #pragma omp single
                num_threads = omp_get_num_threads();
            do_lots_of_stuff(id);
      }
}
```

# Environment Variables

- Set the default number of threads to use.
  - –**OMP_NUM_THREADS *int_literal***
- Control the size of child threads' stack
  - **–OMP_STACKSIZE**
- Hint to runtime how to treat idle threads
  - **–OMP_WAIT_POLICY**
    - – **ACTIVE** **keep threads alive at barriers/locks**
    - – **PASSIVE** **try to release processor at barriers/locks**
- Bind threads to processor
  - **–OMP_PROC_BIND**
    - – TRUE / FALSE **turn on for better use of caches**

# Working with OpenMP

- Synchronization 线程同步
- Data environment 数据属性
- Linked lists traversal 链表遍历

# Data environment

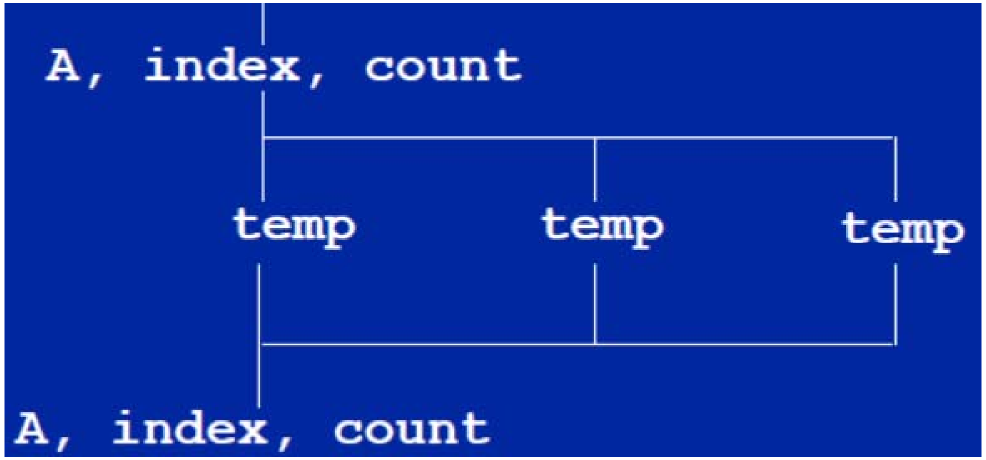
- **Default storage attributes:**
  - Shared Memory programming model:
    - Most variables are shared by default (heap)
  - Global variables are SHARED among threads:
    - File scope variables, static
    - dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in functions called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

# Data environment: example

```
double A[10];
int main() {
      int index[10];
      #pragma omp parallel
            work(index);
      printf("%d\n", index[0]);;
}
```

```
extern double A[10];
void work(int *index) {
      double temp[10];
      static int count;
      ...
}
```

- **A, index and count** are shared by all threads.(**heap**)
- **temp** is local to each thread.(**stack**)



# Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses:
  - – SHARED
  - – PRIVATE
  - – FIRSTPRIVATE
- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - – LASTPRIVATE
- The default attributes can be overridden with:
  - – DEFAULT (SHARED | NONE)

## Data Sharing: Private Clause

- private(var)
  - creates a new local copy of var for each thread.
  - – The value of the private copies is uninitialized
  - – The value of the original variable is unchanged after the region

```
void wrong() {
      int tmp = 0;
#pragma omp parallel for private(tmp)
     for (int j = 0; j < 1000; ++j)
         tmp += j;
     printf("%d\n", tmp);
}
```

# Data Sharing: Firstprivate Clause

- firstprivate(var)
  - Variables initialized from shared variable
  - C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
    if ((i%2)==0) incr++;
    A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0

# Data Sharing: Lastprivate Clause

- lastprivate(var)
  - Variables update shared variable using value from last iteration
  - C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm) {
    double x; int i;
    #pragma omp parallel for lastprivate(x)
        for (i = 0; i < n; i++){
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
        }
    *lastterm = x;
}
```

"x" has the value it held for the "last sequential" iteration (i.e., for i=(n-1))

# Data Sharing: A Test

```
variables:  A = 1,B = 1, C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- **Are A,B,C local to each thread or shared inside the parallel region?**
- **What are their initial values inside and values after the parallel region?**

# Data Sharing: A Test

variables: A = 1,B = 1, C = 1

```
#pragma omp parallel private(B)  firstprivate(C)
```

**Inside this parallel region ...**

- **"A" is shared by all threads; equals 1**
- **"B" and "C" are local to each thread.**
  - **B's initial value is undefined**
  - **C's initial value equals 1**

**Following the parallel region ...**

- **B and C revert to their original values of 1**
- **A is either 1 or the value it was set to inside the parallel region**

# Data Sharing: Default Clause

- **DEFAULT(SHARED)**
  - Note this is the default storage attribute
  - Exception: #pragma omp task
- **DEFAULT(PRIVATE)**
  - To change default to PRIVATE
  - *each* variable in the construct is made private as if specified in a private clause
- **DEFAULT(NONE):**
  - *no* default for variables in static extent. Must list storage attribute for each variable in static extent.
  - Good programming practice!

# Example: 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;
}
```

# Example: PI Program minimal changes

```
#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 for private(x) 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;
}
```

# Working with OpenMP

- Synchronization 线程同步
- Data environment 数据属性
- Linked lists traversal 链表遍历

# Review

- To create a team of threads
  - #pragma omp parallel
- To share work between threads:
  - #pragma omp for
  - #pragma omp single
- To prevent conflicts (prevent races)
  - #pragma omp critical
  - #pragma omp atomic
  - #pragma omp barrier
  - #pragma omp master

# Review

- Data environment clauses
  - private (variable_list)
  - firstprivate (variable_list)
  - lastprivate (variable_list)
  - reduction(+:variable_list)

- Where variable_list is a comma separated list of variables.

# Simple list traversal

- Given what we've covered about OpenMP, how would you process this loop in Parallel?

```
p=head;
while (p) {
    process(p);
    p = p->next;
}
```

- the loop worksharing construct only works with loops **for**
- The **while** loops are not covered

# Test2： linked lists the hard way

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using constructs described so far.
- Once you have a correct program, optimize it.

- Code: test2_Fibonacci/linked.c

# list traversal

- When OpenMP was first created, it focused on common use cases in HPC … Fortran arrays processed over “regular” loops.
- Recursion and “pointer chasing” were removed from the Fortan focus.
- Hence, even a simple list traversal is exceedingly difficult with the original versions of OpenMP.

# Test2-1: Linked lists omp25

```
while (p != NULL) {
      p = p->next;
      count++;
}
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
}
#pragma omp parallel
{
      #pragma omp for
      schedule(static,1)
      for(i=0; i<count; i++)
         processwork(parr[i]);

}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

Code: test2_Fibonacci/solution/Linked_omp25.c

# Linked lists

- Using C++ STL

```
std::vector<node *> nodelist;
for (p = head; p != NULL; p = p->next)
    nodelist.push_back(p);

int j = (int)nodelist.size();

#pragma omp parallel for schedule(static,1)
    for (int i = 0; i < j; ++i)
        processwork(nodelist[i]);
```

Count number of items in the linked list

Count number of items in the linked list

Process nodes in parallel with a for loop

# Conclusion

- We were able to parallelize the linked list traversal …
  - but it was ugly and required multiple passes over the data.

- To move beyond its roots in the array based world of scientific computing, we needed to:
  - support more general data structures
  - support loops beyond basic ‘for’ loops.

- To do this, we have the OpenMP 3.0

# 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。