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

Introduction to OpenMP

Part4. *Advanced Topics*

人工智能学院

缪青海

miaoqh@ucas.ac.cn

Content

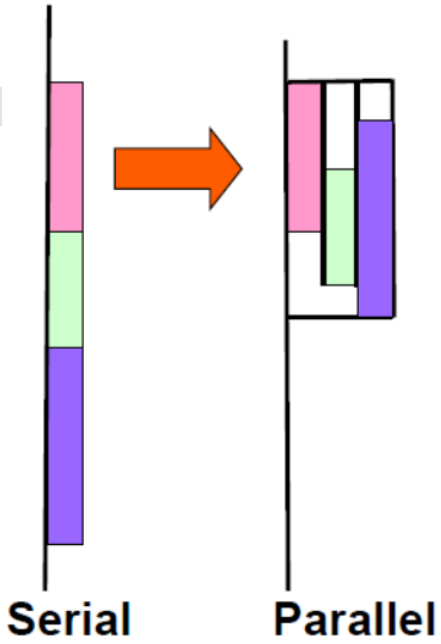
- Getting started with OpenMP
- Core features of OpenMP
- Work with OpenMP
- **Advanced topics of OpenMP**

Advances of OpenMP

- Tasks for linked list
- Monte Carlo Calculations
- Parallel Patterns

OpenMP Tasks

- Tasks are independent units of work.
- Tasks are composed of:
  - – **code** to execute
  - – **data** environment
  - – **internal control variables** (ICV)

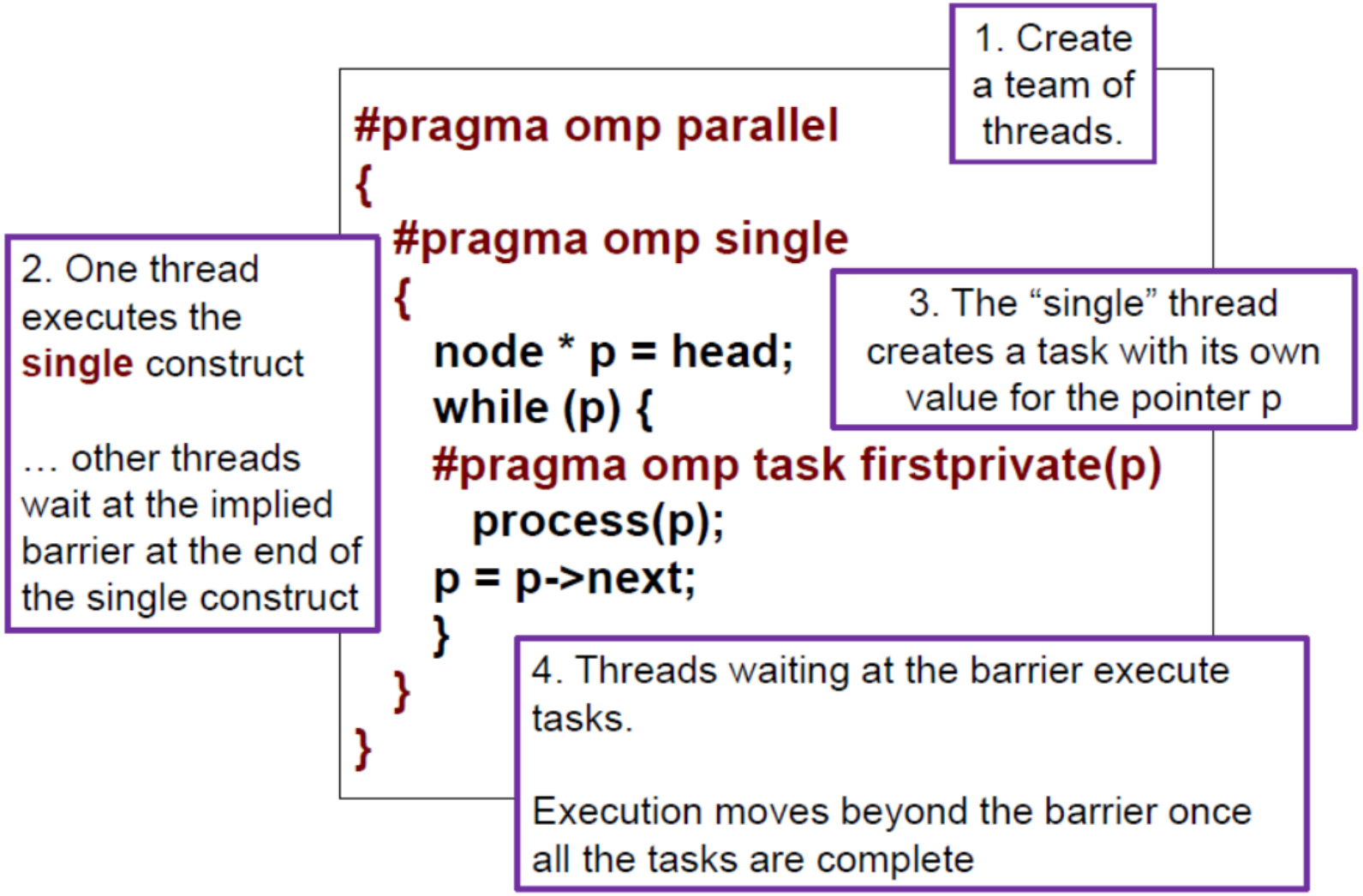


- Threads perform the work of each task.
- The runtime system decides when tasks are executed
  - – Tasks may be deferred
  - – Tasks may be executed immediately

Task Definitions

- *Task construct*
  - – task directive plus structured block
- *Task*
  - – the package of code and instructions for allocating data created when a thread encounters a task construct
- *Task region*
  - – the dynamic sequence of instructions produced by the execution of a task by a thread

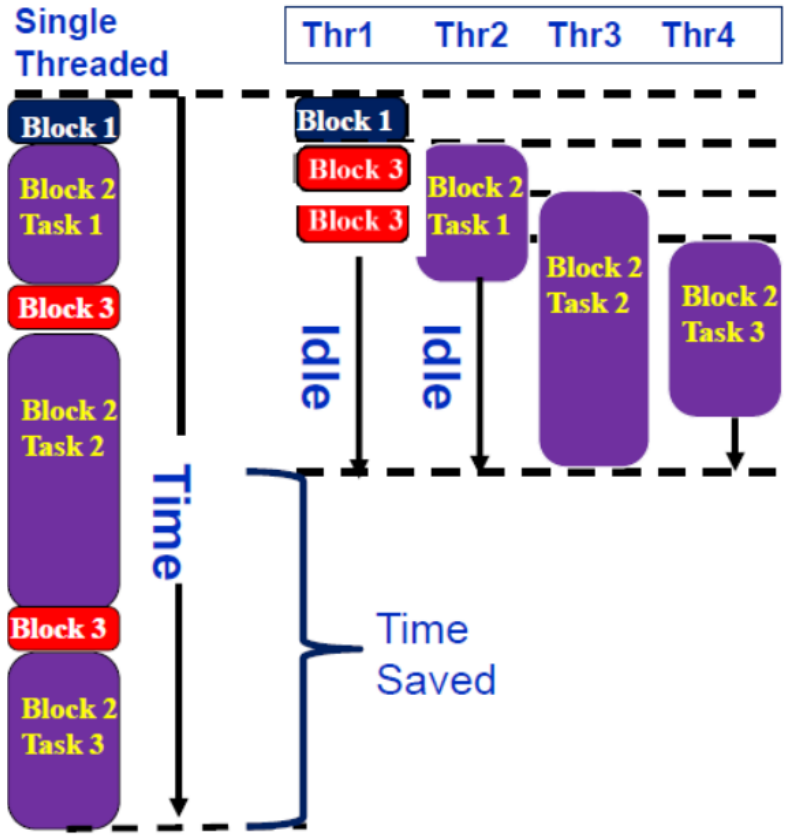
Task Construct – Explicit Tasks



Execution of tasks

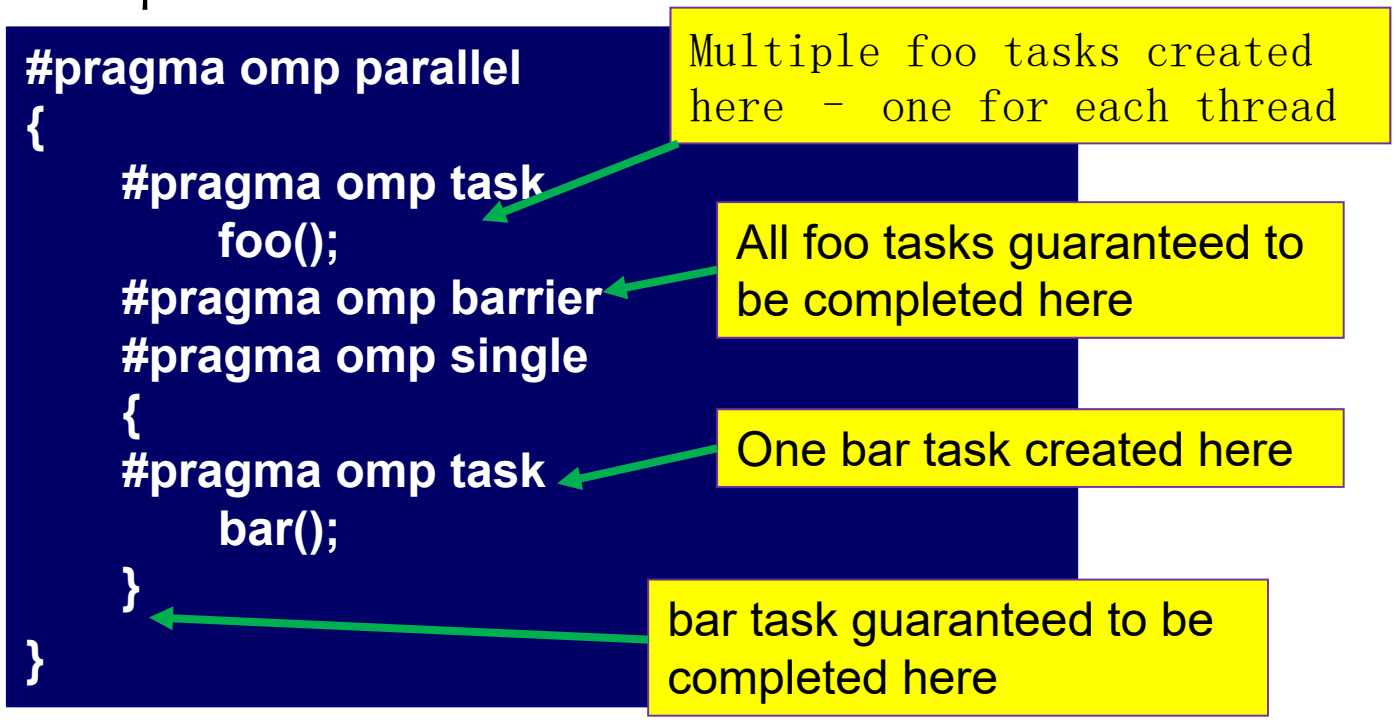
- Have potential to parallelize **irregular patterns** and **recursive function calls**

```
#pragma omp parallel
{
    #pragma omp single
    {   //block 1
        node * p = head;
        while (p) { // block 2
        #pragma omp task
            process(p);
        p = p->next; //block 3
        }
    }
}
```



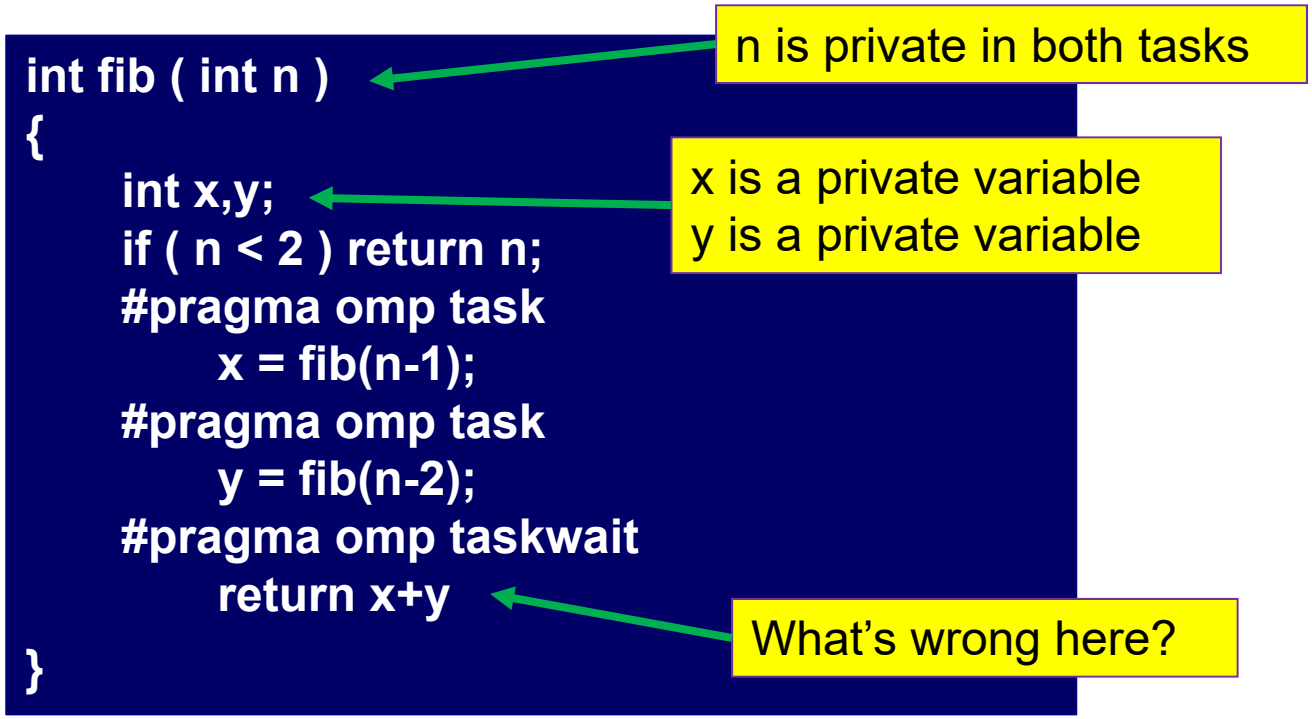
When tasks completed?

- Tasks are guaranteed to be complete at thread barriers:
  - #pragma omp barrier
- or task barriers
  - #pragma omp taskwait



Data Scoping with tasks

- Fibonacci example:
  - The divide and conquer design pattern



A task's private variables are undefined outside the task

Data Scoping with tasks

- Fibonacci example:
  - The divide and conquer design pattern

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
    #pragma omp task shared (x)
        x = fib(n-1);
    #pragma omp task shared (y)
        y = fib(n-2);
    #pragma omp taskwait
        return x+y
}
```

n is private in both tasks

x & y are shared
Good solution
we need both values
to compute the sum

Data Scoping with tasks

- List Traversal example:

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
      for(e=ml->first;e;e=e->next)
      #pragma omp task
      process(e);
}
```

What's wrong here?

Possible data race ! Shared variable e updated by multiple tasks

Data Scoping with tasks

- List Traversal example:

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
      for(e=ml->first;e;e=e->next)
      #pragma omp task firstprivate(e)
      process(e);
}
```

What's wrong here?

Good solution
– e is firstprivate

Test2-2: tasks in OpenMP

- **Consider the program linked.c**
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- **Parallelize this program using tasks.**
  - Compare new solution's complexity to the approach without tasks.

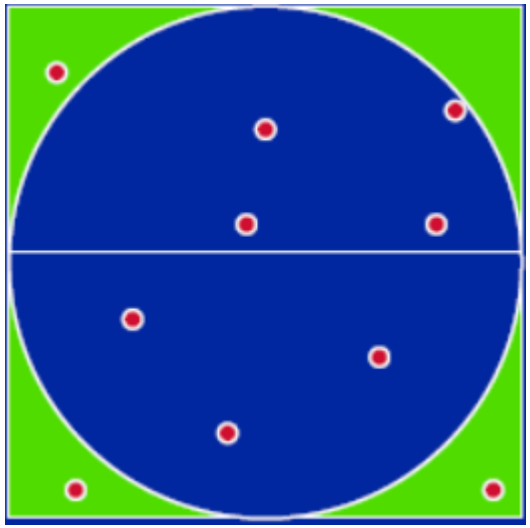
- **Code:**
  - **test2_Fibonacci/solutions/linked_omp3_tasks.c**

Advances of OpenMP

- Tasks for linked list
- Monte Carlo Calculations
- Parallel Patterns

Monte Carlo Calculations

- **Monte Carlo Method:**
  - Using Random numbers to solve tough problems
  - Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- **Example: Computing π with a digital dart board:**



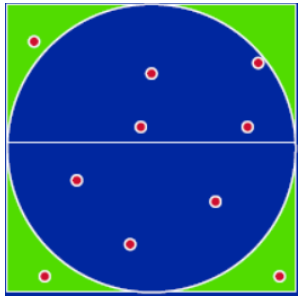
N= 10 π = 2.8

N=100 π = 3.16

N= 1000 π = 3.148

Monte Carlo Calculations:

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:
  - $Ac = \pi * r^2$
  - $As = (2*r) * (2*r) = 4 * r^2$
  - $P = Ac/As = \pi /4$



- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

- Code: test3_Pi_mc/pi_mc.c 串行代码

Monte Carlo Calculations:

- Parallel Programmers love Monte Carlo algorithms.
- Embarrassingly parallel:
  - the parallelism is so easy its embarrassing.
- Add two lines and you have a parallel program.

- Code: test3_Pi_mc/solutions/pi_mc_par.c 并行代码

Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality.

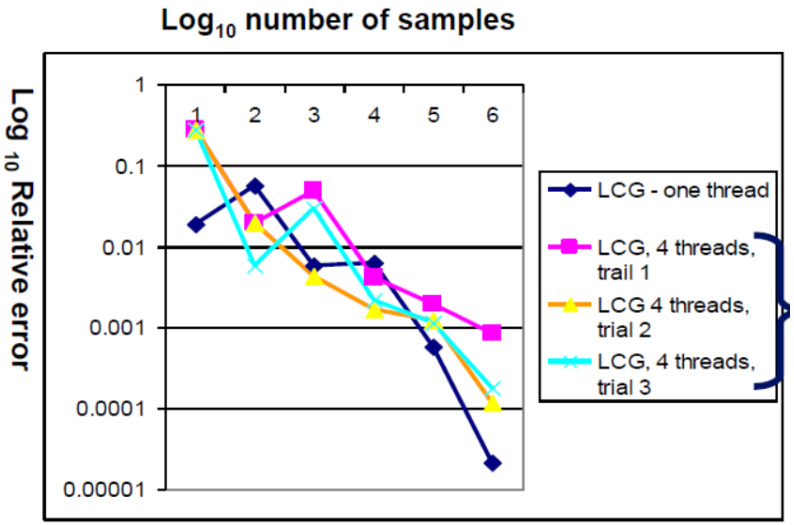
```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up :
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025
- Code: test3_Pi_mc/solutions/random_seq_lcg.c

  compile: gcc –fopenmp pi_mc_par.c random_seq_lcg.c

PI_MC with LCG generator

- **Problem:**
  - Run the same program the same way and get different answers!
  - That is not acceptable!



- **Issue:**
  - the LCG generator is not threadsafe.

LCG code: threadsafe version

```
static long MULTIPLIER = 1366;
static long ADDEND = 150889;
static long PMOD = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
   long random_next;
   random_next = (MULTIPLIER * random_last + ADDEND)% PMOD;
   random_last = random_next;
   return ((double)random_next/(double)PMOD);
}
```

Construct (指令)

threadprivate不是针对某一个并行块，而是作用于整个程序。

LCG code: threadsafe version

- #pragma omp threadprivate(random_last)
- random_last carries state between random number computations
- To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.
- Code:
  - test3_Pi_mc/solutions/random_par.c

compile: gcc –fopenmp pi_mc_par.c random_par.c

Data sharing: Threadprivate

- Makes global data private to a thread
  - C: File scope and static variables, static class members
- Different from making them PRIVATE 区别
  - with PRIVATE global variables are masked in a parallel region.
  - THREADPRIVATE preserves global scope within each Thread.
- Threadprivate variables can be initialized using COPYIN or at time of definition (using language defined initialization capabilities).

A threadprivate example

- Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)
int increment_counter()
{
    counter++;
    return (counter);
}
```

A threadprivate example

```
#pragma omp parallel
{
  int    count;
#pragma omp single copyprivate(counter)
  {
     counter = 50;
  }
  count = increment_counter();
  printf("ThreadId: %ld, count = %ld/n", mp_get_thread_num(), count);
}
```

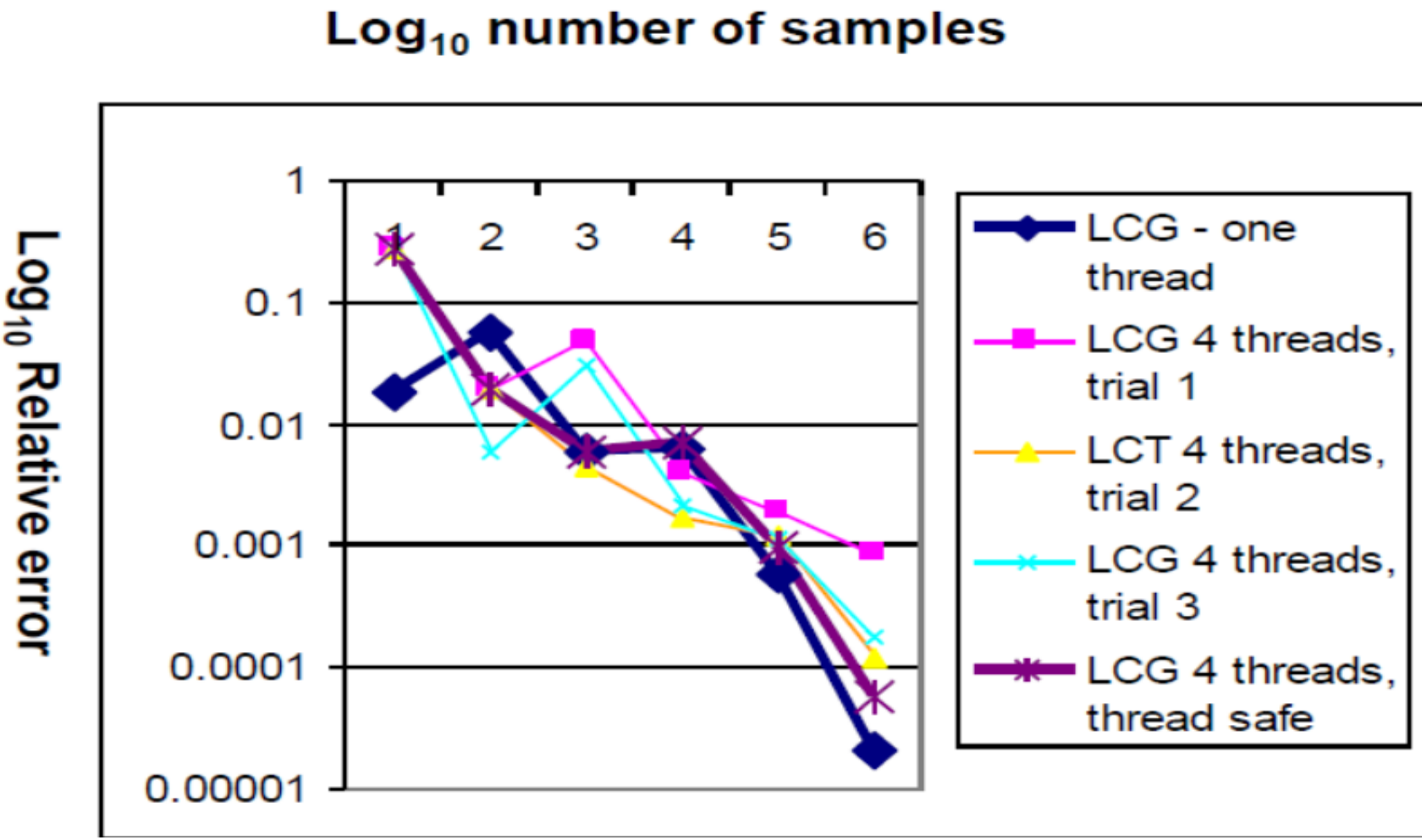
- Copyprivate():
  - Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.

Thread safe random number generators

- test3_Pi_mc:
  - Thread safe version gives the same answer each time you run the program.

- But for large number of samples, its quality is lower than the one thread result!
- Why?

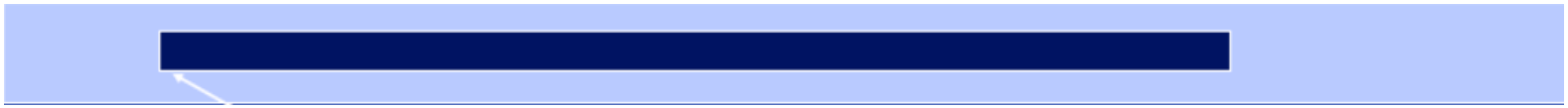
Thread safe random number generators



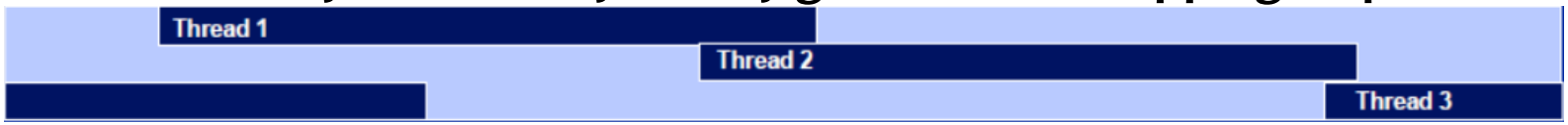
Pseudo Random Sequences

- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

- In a typical problem, you grab a subsequence of the RNG range



- Grab arbitrary seeds and you may generate overlapping sequences



- Overlapping sequences = over-sampling and bad statistics … lower quality or even wrong answers!

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
  - Give each thread a separate, independent generator
  - Have one thread generate all the numbers.
  - Leapfrog … deal out sequence values "round robin" as if dealing a deck of cards.
  - Block method … pick your seed so each threads gets a distinct contiguous block.

- OR… use/buy a math library that does it right.

Advances of OpenMP

- Tasks for linked list
- Monte Carlo Calculations
- Parallel Patterns

Summary of OpenMP

- **Parallel Patterns:**
  - SPMD: Single Program Multiple Data
  - Loop parallelism
  - Divide and Conquer
  - ……

Parallel Patterns

- **SPMD:**
  - Run the same program on P processing elements;
  - Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks.
  - most commonly used pattern in the history of parallel programming
  - Example : computing Pi

SPMD

```
#include <omp.h>
void main (int argc, char *argv[])
{
      int i, pi=0.0, step, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel firstprivate(sum) private(x, i)
      {
        int id = omp_get_thread_num();
        int numprocs= omp_get_num_threads();
        int step1 = id *num_steps/numprocs;
        int stepN = (id+1)*num_steps/numprocs;
        if (stepN! = num_steps) stepN= num_steps;
        for (i=step1;  i<stepN; i++){
          x = (i+0.5)*step;
          sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
            pi += sum *step ;
      }
}
```

Parallel Patterns

- **Loop parallelism:**
  - Collections of tasks are defined as iterations of loops;
  - Loop iterations are divided between a collection of processing elements.
  - heavily used with data parallel design patterns;
  - commonly used by OpenMP programmers.
  - Example : computing Pi

LOOP

```
#include <omp.h>
static long num_steps= 100000;
double step;
#define NUM_THREADS 2
void main ()
{
      Int i; double x, pi, sum =0.0;
      step = 1.0/(double) num_steps;
      omp_set_num_threads(NUM_THREADS);
      #pragma omp parallel for private(x) reduction (+:sum)
            for (i=0; I < num_steps;  i++){
               x = (i+0.5)*step;
               sum += 4.0/(1.0+x*x);
            }
      pi = sum * step;
}
```

Parallel Patterns

- **Divide and Conquer:**
  - Divide a problem into sub-problems;
  - Computing in parallel;
  - and recombine solutions of sub-problems into a global solution.
  - Example : computing Pi

Divide and Conquer

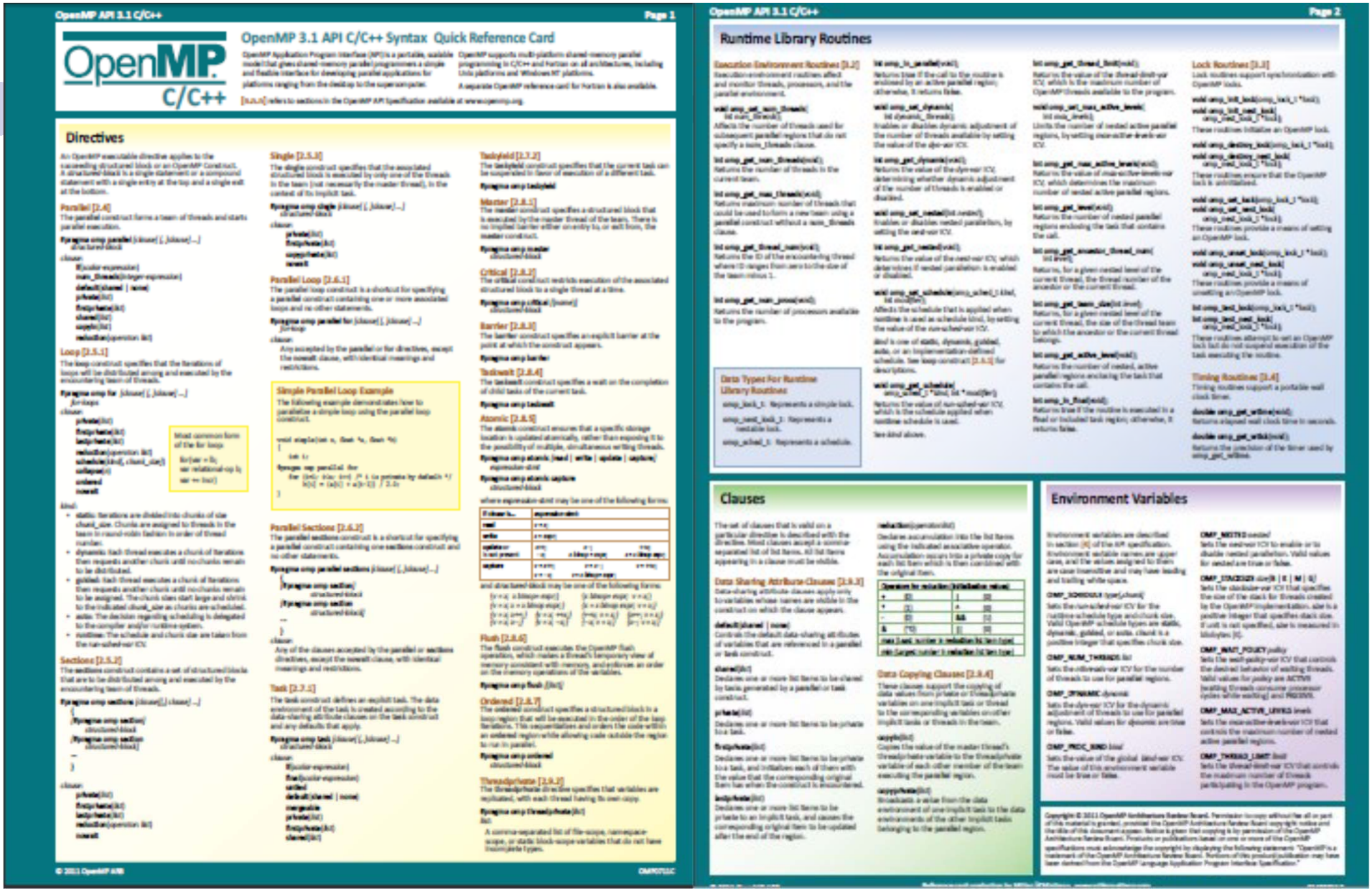
```
#include <omp.h>
static long num_steps= 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart, int Nfinish, double step){
      Int i, iblk; double x, sum = 0.0, sum1, sum2;
      if (Nfinish-Nstart< MIN_BLK){
            for (i=Nstart; i< Nfinish; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
            }
      }
      else{
            iblk= Nfinish-Nstart;
            #pragma omp task shared(sum1)
                  sum1 = pi_comp(Nstart, Nfinish-iblk/2, step);
            #pragma omp task shared(sum2)
                  sum2 = pi_comp(Nfinish-iblk/2, Nfinish,  step);
            #pragma omp taskwait
                  sum = sum1 + sum2;
      }
      return sum;
}
```

```
int main (){
   inti;double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel {
      #pragma omp single
      sum=pi_comp(0,num_steps,step);
   }
   pi = step * sum;
}
```

Advances of OpenMP

- More topics
  - Memory model
  - Teams
  - Distributed SIMD
  - Pairwise Synchronization
  - ……

https://www.openmp.org/specifications/



OpenMP API 3.1 C/C++ Page 1

OpenMP 3.1 API C/C++ Syntax Quick Reference Card

OpenMP Application Program Interface (API) is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.

A separate OpenMP reference card for Fortran is also available.

[3.2.1] refers to sections in the OpenMP API Specification available at www.openmp.org.

Directives

An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A structured-block is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

Parallel [2.4]

The parallel construct forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[ [,]clause]...]
  structured-block
```

clause:
- if(scalar-expression)
- num_threads(integer-expression)
- default(shared | none)
- private(list)
- firstprivate(list)
- shared(list)
- copyin(list)
- reduction(operator: list)

Loop [2.5.1]

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

```
#pragma omp for [clause[ [,]clause] ...]
  for-loops
```

clause:
- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction(operator: list)
- schedule(kind[, chunk_size])
- collapse(n)
- ordered
- nowait

Most common form of the for loops:

```
for(var = lb;
    var relational-op b;
    var += incr)
```

kind:
- static: Iterations are divided into chunks of size chunk_size. Chunks are assigned to threads in the team in round-robin fashion in order of thread number.
- dynamic: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be distributed.
- guided: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned. The chunk sizes start large and shrink to the indicated chunk_size as chunks are scheduled.
- auto: The decision regarding scheduling is delegated to the compiler and/or runtime system.
- runtime: The schedule and chunk size are taken from the run-sched-var ICV.

Sections [2.5.2]

The sections construct contains a set of structured blocks that are to be distributed among and executed by the encountering team of threads.

```
#pragma omp sections [clause[ [,]clause] ...]
{
[#pragma omp section]
  structured-block
[#pragma omp section
  structured-block]
...
}
```

clause:
- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction(operator: list)
- nowait

Single [2.5.3]

The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.

```
#pragma omp single [clause[ [,]clause]...]
  structured-block
```

clause:
- private(list)
- firstprivate(list)
- copyprivate(list)
- nowait

Parallel Loop [2.6.1]

The parallel loop construct is a shortcut for specifying a parallel construct containing one or more associated loops and no other statements.

```
#pragma omp parallel for [clause[ [,]clause] ...]
  for-loop
```

clause:
Any accepted by the parallel or for directives, except the nowait clause, with identical meanings and restrictions.

Simple Parallel Loop Example

The following example demonstrates how to parallelize a simple loop using the parallel loop construct.

```
void simple(int n, float *a, float *b)
{
  int i;
#pragma omp parallel for
  for (i=1; i<n; i++) /* i is private by default */
    b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Parallel Sections [2.6.2]

The parallel sections construct is a shortcut for specifying a parallel construct containing one sections construct and no other statements.

```
#pragma omp parallel sections [clause[ [,]clause] ...]
{
[#pragma omp section]
  structured-block
[#pragma omp section
  structured-block]
...
}
```

clause:
Any of the clauses accepted by the parallel or sections directives, except the nowait clause, with identical meanings and restrictions.

Task [2.7.1]

The task construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

```
#pragma omp task [clause[ [,]clause] ...]
  structured-block
```

clause:
- if(scalar-expression)
- final(scalar-expression)
- untied
- default(shared | none)
- mergeable
- private(list)
- firstprivate(list)
- shared(list)

Taskyield [2.7.2]

The taskyield construct specifies that the current task can be suspended in favor of execution of a different task.

```
#pragma omp taskyield
```

Master [2.8.1]

The master construct specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

```
#pragma omp master
  structured-block
```

Critical [2.8.2]

The critical construct restricts execution of the associated structured block to a single thread at a time.

```
#pragma omp critical [(name)]
  structured-block
```

Barrier [2.8.3]

The barrier construct specifies an explicit barrier at the point at which the construct appears.

```
#pragma omp barrier
```

Taskwait [2.8.4]

The taskwait construct specifies a wait on the completion of child tasks of the current task.

```
#pragma omp taskwait
```

Atomic [2.8.5]

The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

```
#pragma omp atomic [read | write | update | capture]
  expression-stmt
#pragma omp atomic capture
  structured-block
```

where expression-stmt may be one of the following forms:

If clause is...	expression-stmt:
read	v = x;
write	x = expr;
update or (not present)	x++; / x--; / ++x; / --x; / x binop= expr; / x = x binop expr; / x = expr binop x;
capture	v = x++; / v = x--; / v = ++x; / v = --x; / v = x binop= expr; / v = x = x binop expr; / v = x = expr binop x;

and structured-block may be one of the following forms:

{v = x; x binop= expr;} {x binop= expr; v = x;}
{v = x; x = x binop expr;} {x = expr binop x; v = x;}
{v = x; x++;} {v = x; ++x;} {++x; v = x;} {x++; v = x;}
{v = x; x--;} {v = x; --x;} {--x; v = x;} {x--; v = x;}

Flush [2.8.6]

The flush construct executes the OpenMP flush operation, which makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

```
#pragma omp flush [(list)]
```

Ordered [2.8.7]

The ordered construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

```
#pragma omp ordered
  structured-block
```

Threadprivate [2.9.2]

The threadprivate directive specifies that variables are replicated, with each thread having its own copy.

```
#pragma omp threadprivate (list)
```

list:
A comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.

OpenMP API 3.1 C/C++ Page 2

Runtime Library Routines

Execution Environment Routines [3.2]

Execution environment routines affect and monitor threads, processors, and the parallel environment.

void omp_set_num_threads(int num_threads);
Affects the number of threads used for subsequent parallel regions that do not specify a num_threads clause.

int omp_get_num_threads(void);
Returns the number of threads in the current team.

int omp_get_max_threads(void);
Returns maximum number of threads that could be used to form a new team using a parallel construct without a num_threads clause.

int omp_get_thread_num(void);
Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

int omp_get_num_procs(void);
Returns the number of processors available to the program.

int omp_in_parallel(void);
Returns true if the call to the routine is enclosed by an active parallel region; otherwise, it returns false.

void omp_set_dynamic(int dynamic_threads);
Enables or disables dynamic adjustment of the number of threads available by setting the value of the dyn-var ICV.

int omp_get_dynamic(void);
Returns the value of the dyn-var ICV, determining whether dynamic adjustment of the number of threads is enabled or disabled.

void omp_set_nested(int nested);
Enables or disables nested parallelism, by setting the nest-var ICV.

int omp_get_nested(void);
Returns the value of the nest-var ICV, which determines if nested parallelism is enabled or disabled.

void omp_set_schedule(omp_sched_t kind, int modifier);
Affects the schedule that is applied when runtime is used as schedule kind, by setting the value of the run-sched-var ICV.

kind is one of static, dynamic, guided, auto, or an implementation-defined schedule. See loop construct [2.5.1] for descriptions.

void omp_get_schedule(omp_sched_t *kind, int * modifier);
Returns the value of run-sched-var ICV, which is the schedule applied when runtime schedule is used.

See kind above.

int omp_get_thread_limit(void);
Returns the value of the thread-limit-var ICV, which is the maximum number of OpenMP threads available to the program.

void omp_set_max_active_levels(int max_levels);
Limits the number of nested active parallel regions, by setting max-active-levels-var ICV.

int omp_get_max_active_levels(void);
Returns the value of max-active-levels-var ICV, which determines the maximum number of nested active parallel regions.

int omp_get_level(void);
Returns the number of nested parallel regions enclosing the task that contains the call.

int omp_get_ancestor_thread_num(int level);
Returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread.

int omp_get_team_size(int level);
Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

int omp_get_active_level(void);
Returns the number of nested, active parallel regions enclosing the task that contains the call.

int omp_in_final(void);
Returns true if the routine is executed in a final or included task region; otherwise, it returns false.

Lock Routines [3.3]

Lock routines support synchronization with OpenMP locks.

void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
These routines initialize an OpenMP lock.

void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
These routines ensure that the OpenMP lock is uninitialized.

void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
These routines provide a means of setting an OpenMP lock.

void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
These routines provide a means of unsetting an OpenMP lock.

int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

Timing Routines [3.4]

Timing routines support a portable wall clock timer.

double omp_get_wtime(void);
Returns elapsed wall clock time in seconds.

double omp_get_wtick(void);
Returns the precision of the timer used by omp_get_wtime.

Data Types For Runtime Library Routines

omp_lock_t: Represents a simple lock.

omp_nest_lock_t: Represents a nestable lock.

omp_sched_t: Represents a schedule.

Clauses

The set of clauses that is valid on a particular directive is described with the directive. Most clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.

Data-Sharing Attribute Clauses [2.9.3]

Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears.

default(shared | none)
Controls the default data-sharing attributes of variables that are referenced in a parallel or task construct.

shared(list)
Declares one or more list items to be shared by tasks generated by a parallel or task construct.

private(list)
Declares one or more list items to be private to a task.

firstprivate(list)
Declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

lastprivate(list)
Declares one or more list items to be private to an implicit task, and causes the corresponding original item to be updated after the end of the region.

reduction(operator:list)
Declares accumulation into the list items using the indicated associative operator. Accumulation occurs into a private copy for each list item which is then combined with the original item.

Operator for reduction	Initializer value
+	0
*	1
-	0
&	~0
|	0
&&	1
||	0
max	least number in reduction list item type
min	largest number in reduction list item type

Data Copying Clauses [2.9.4]

These clauses support the copying of data values from private or threadprivate variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.

copyin(list)
Copies the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region.

copyprivate(list)
Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.

Environment Variables

Environment variables are described in section [4] of the API specification. Environment variable names are upper case, and the values assigned to them are case insensitive and may have leading and trailing white space.

OMP_SCHEDULE type[,chunk]
Sets the run-sched-var ICV for the runtime schedule type and chunk size. Valid OpenMP schedule types are static, dynamic, guided, or auto. chunk is a positive integer that specifies chunk size.

OMP_NUM_THREADS list
Sets the nthreads-var ICV for the number of threads to use for parallel regions.

OMP_DYNAMIC dynamic
Sets the dyn-var ICV for the dynamic adjustment of threads to use for parallel regions. Valid values for dynamic are true or false.

OMP_PROC_BIND bind
Sets the value of the global bind-var ICV. The value of this environment variable must be true or false.

OMP_NESTED nested
Sets the nest-var ICV to enable or to disable nested parallelism. Valid values for nested are true or false.

OMP_STACKSIZE size[B | K | M | G]
Sets the stacksize-var ICV that specifies the size of the stack for threads created by the OpenMP implementation. size is a positive integer that specifies stack size. If unit is not specified, size is measured in kilobytes [K].

OMP_WAIT_POLICY policy
Sets the wait-policy-var ICV that controls the desired behavior of waiting threads. Valid values for policy are ACTIVE (waiting threads consume processor cycles while waiting) and PASSIVE.

OMP_MAX_ACTIVE_LEVELS levels
Sets the max-active-levels-var ICV that controls the maximum number of nested active parallel regions.

OMP_THREAD_LIMIT limit
Sets the thread-limit-var ICV that controls the maximum number of threads participating in the OpenMP program.

OpenMP Application Programming Interface

Examples

- Supplementary Source Code – GitHub Repository
- OpenMP 5.0 Examples – Nov 2019
- OpenMP 5.0 Examples Discussion Forum

references

- https://www.openmp.org

- 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。