OpenMP

OpenMP

- Portable programming of shared memory systems.
- It is a quasi-standard.
- OpenMP-Forum
- 1997 - 2002
- API for Fortran and C/C++
  - directives
  - runtime routines
  - environment variables
- www.openmp.org

Example

Program

```
main(){
   #pragma omp parallel
   {
    printf("Hello world");
   }
}
```

Compilation

```
> cc –O2 –parallel –omp openmp.c
```

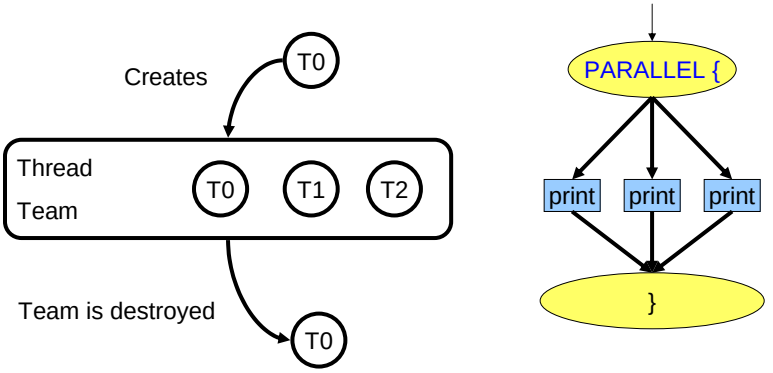
Execution

```
> export OMP_NUM_THREADS=2
> prun –p IAPAR –n 1 a.out
Hello world
Hello world
```

```
> export OMP_NUM_THREADS=3
> prun –p IAPAR –n 1 a.out
Hello World
Hello World
Hello World
```

Execution Model

```
#pragma omp parallel
{
 printf("Hello world %d\n", omp_get_thread_num());
}
```



Fork/Join Execution Model

1. An OpenMP-program starts as a single thread (*master thread*).
2. Additional threads (*Team*) are created when the master hits a parallel region.
3. When all threads finished the parallel region, the new threads are given back to the runtime or operating system.

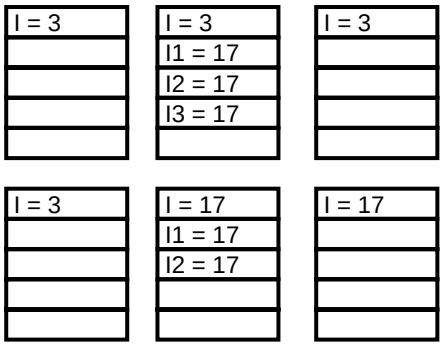
- A team consists of a fixed set of threads executing the parallel region redundantly.
- All threads in the team are synchronized at the end of a parallel region via a barrier.
- The master continues after the parallel region.

Shared and Private Data

- Shared data are accessible by all threads. A reference a[5] to a shared array accesses the same address in all threads.
- Private data are accessible only by a single thread. Each thread has its own copy.
- The default is shared.

Example: Private Data

```
I=3
#pragma omp parallel private(i)
{
 I=17
}
Printf("Value of I=%d\n", I);
```

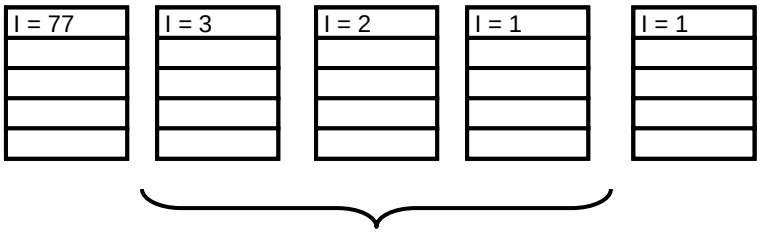


Private Data

- A new copy is created for each thread.
- One thread may reuse the global shared copy.
- The private copies are destroyed after the parallel region.
- The value of the shared copy is undefined.

Example: Shared Data

```
I=77
#pragma omp parallel shared(i)
{
 I=omp_get_thread_num();
}
Printf("Value of I=%d\n", I);
```



In Parallel Region

Syntax of Directives and Pragmas

Fortran

!$OMP *directive name* [*parameters*]

```
!$OMP PARALLEL DEFAULT(SHARED)
      write(*,*) ´Hello world´
!$OMP END PARALLEL
```

C / C++

#pragma omp *directive name* [*parameters*]

```
int main()  {
  #pragma omp parallel default(shared)
  {
    printf("hello world\n" );
  }
}
```

Directives

Directives can have continuation lines

- Fortran

```
!$OMP directive name first_part &
!$OMP continuation_part
```

- C

```
#pragma omp parallel private(i) \
    private(j)
```

Parallel Region

- The statements enclosed lexically within a region define the lexical extent of the region.
- The dynamic extent further includes the routines called from within the construct.

```
#pragma omp parallel [parameters]
{
  parallel region
}
```

Example

```
main (){
int iam, nthreads;

#pragma omp parallel private(iam,nthreads)
 {
  iam = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  printf("ThradID %d, out of %d threads\n", iam, nthreads);

  if (iam == 0) ! Different control flow
    printf("Here is the Master Thread.\n");
  else
    printf("Here is another thread.\n");
 }
}
```

Lexical and Dynamic Extend

```
main (){
int a[100];
#pragma omp parallel
 {
  …
 }
}
```

```
sub(int a[])
{
#pragma omp for
  for (int i= 1; i<n;i++)
     a(i) = i;
}
```

- Local variables of a subroutine called in a parallel region are by default private.

Work-Sharing Constructs

- Work-sharing constructs distribute the specified work to all threads within the current team.

- Types
  - Parallel loop
  - Parallel section
  - Master region
  - Single region
  - General work-sharing construct (only Fortran)

Parallel Loop

```
#pragma omp for [parameters]
    for ...
```

- The iterations of the do-loop are distributed to the threads.
- The scheduling of loop iterations is determined by one of the scheduling strategies *static*, *dynamic*, *guided*, and *runtime*.
- There is no synchronization at the beginning.
- All threads of the team synchronize at an implicit barrier if the parameter *nowait* is not specified.
- The loop variable is by default private. It must not be modified in the loop body.
- The expressions in the for-statement are very restricted.

Scheduling Strategies

- Schedule clause

  schedule (type [size])
- Scheduling types:
  - static: Chunks of the specified size are assigned in a round-robin fashion to the threads.
  - dynamic: The iterations are broken into chunks of the specified size. When a thread finishes the execution of a chunk, the next chunk is assigned to that thread.
  - guided: Similar to dynamic, but the size of the chunks is exponentially decreasing. The size parameter specifies the smallest chunk. The initial chunk is implementation dependent.
  - runtime: The scheduling type and the chunk size is determined via environment variables.

Example: Dynamic Scheduling

```
main(){
int i, a[1000];

#pragma omp parallel
 {
  #pragma omp for schedule(dynamic, 4)
  for (int i=0; i<1000;i++)
    a[i] = omp_get_thread_num();

  #pragma omp for schedule(guided)
  for (int i=0; i<1000;i++)
         a[i] = omp_get_thread_num();

 }
}
```

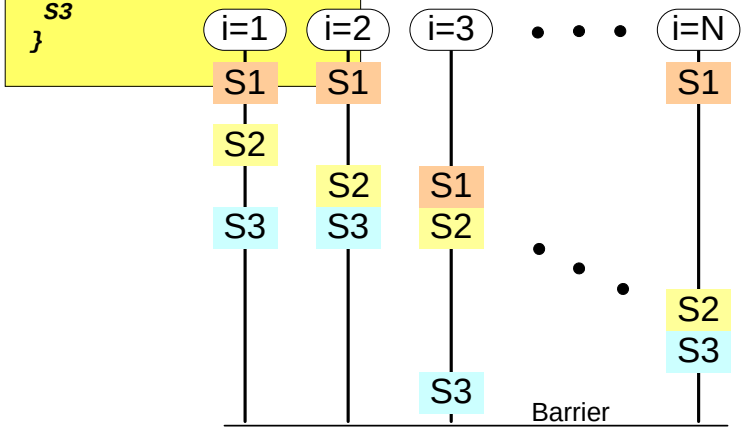
Ordered Construct

```
#pragma omp for ordered
    for (...)
     { ...
       #pragma omp ordered
       { ... }
       ...
     }
```

- Construct must be within the dynamic extent of an *omp for* construct with an ordered clause.
- Ordered constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.

Example with ordered clause

```
#pragma omp for ordered
for (...)
 { S1
  #pragma omp ordered
  { S2}
  S3
 }
```



Parallel Section

```
#pragma omp sections [parameters]
{
 [#pragma omp section]
      block
 [#pragma omp section
      block ]
}
```

- Each section of a parallel section is executed once by one thread of the team.
- Threads that finished their section wait at the implicit barrier at the end of the section construct.

Example: Parallel Section

```
main(){
int i, a[1000], b[1000]

#pragma omp parallel private(i)
 {
 #pragma omp sections
  {
  #pragma omp section
  for (int i=0; i<1000; i++)
    a[i] = 100;
  #pragma omp section
  for (int i=0; i<1000; i++)
    b[i] = 200;
  }

 }
}
```

OMP Workshare (Fortran only)

```
!$OMP WORKSHARE [parameters]
block
!$OMP END WORKSHARE [NOWAIT]
```

- The WORKSHARE directive divides the work of executing the enclosed code into separate units of work and distributes the units amongst the threads.
- An implementation of the WORKSHARE directive must insert any synchronization that is required to maintain standard Fortran semantics.
- There is an implicit barrier at the end of the workshare region.

Sharing Work in a Fortran 90 Array Statement

A(1:N)=B(2:N+1)+C(1:N)

- Each evaluation of an array expression for an individual index is a unit of work.
- The assignment to an individual array element is also a unit of work.

Master / Single Region

```
#pragma omp master
        block

#pragma omp single [parameters]
  block
```

- A master or single region enforces that only a single thread executes the enclosed code within a parallel region.
- Common
  - No synchronization at the beginning of region.
- Different
  - Master region is executed by master thread while the single region can be executed by any thread.
  - Master region is skipped by other threads while all threads are synchronized at the end of a single region.

Combined Work-Sharing and Parallel Constructs

- #pragma omp parallel for
- #pragma omp parallel sections
- !$OMP PARALLEL WORKSHARE

Critical Section

```
#pragma omp critical [(Name)]
{ ... }
```

- Mutual exclusion
  - A critical section is a block of code that can be executed by only one thread at a time.
- Critical section name
  - A thread waits at the beginning of a critical section until no other thread is executing a critical section with the same name.
  - All unnamed critical directives map to the same name.
  - Critical section names are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.

Barrier

```
#pragma omp barrier
```

- The barrier synchronizes all the threads in a team.
- When encountered, each thread waits until all of the other threads in that team have reached this point.

Example: Critical Section

```
main(){
int ia = 0
int ib = 0
int itotal = 0

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

```
#pragma omp parallel private(i)
{
#pragma omp sections
 {
  #pragma omp section
  {
  for (int i=0;i<N;i++)
    ia = ia + a[i];
  #pragma omp critical (c1)
   {
    itotal = itotal + ia;
   }}
  #pragma omp section
  {
  for (int i=0;i<N;i++)
    ib = ib + b[i]
  #pragma omp critical (c1)
   {
    itotal = itotal + ib;
   }}
}}
```

Atomic Statements

```
#pragma ATOMIC
     expression-stmt
```

- The ATOMIC directive ensures that a specific memory location is updated atomically
- Has to have the following form:
  - *x binop= expr*
  - *x++ or ++x*
  - *x-- or – x*
  - where x is an lvalue expression with scalar type and expr does not reference the object designated by x.
- All parallel assignments to the location must be protected with the atomic directive.

Translation of Atomic

•Only the load and store of x are protected.

```
#pragma omp atomic
   x += expr
```

can be rewritten as

```
xtmp = expr
!$OMP CRITICAL (name)
   x = x + xtmp
!$OMP END CRITICAL (name)
```

Flush

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

- The flush directive synchronizes copies in register or cache of the executing thread with main memory.
- It synchronizes those variable in the given list or, if no list is specified, all shared variable accessible in the region.
- It does not update implicit copies at other threads.
- Load/stores executed before the flush in program order have to be finished.
- Load/stores following the flush in program order are not allowed to be executed before the flush.
- A flush is executed implicitly for some constructs, e.g. begin and end of a parallel region, end of work-sharing constructs ...

Example: Flush

```
#define MAXTHREAD 100
int iam, neigh, isync[MAXTHREAD];

isync[0] = 1;
#pragma omp parallel private(iam, neigh)
{
 iam = omp_get_thread_num()+1;
 neigh = iam – 1;
 isync[iam] = 0;
 #pragma omp barrier
 work();
 isync[iam] = 1;  //I am done
 #pragma omp flush(isync)

 while (isync[neigh] == 0) {
   #pragma omp flush(isync)
 }              //Wait for neighbor
}
```

isync

1
0
0
1
0
1

Reductions

```
reduction(operator: list)
```

- This clause performs a reduction on the variables that appear in *list*, with the operator *operator*.
- Variables must be shared and can be scalar or arrays
- *operator* is one of the following:
  - +, *, -, &, ^, |, &&, ||
- Reduction variable might only appear in statements with the following form:
  - *x = x operator expr*
  - *x binop= expr*
  - *x++, ++x, x--, --x*

Example: Reduction

```
#pragma omp parallel for reduction(+: a)
for (i=0; i<n; i++) {
  a = a + b[i];

}
```

Classification of Variables

- private(var-list)
  - Variables in var-list are private.
- shared(var-list)
  - Variables in var-list are shared.
- default(private | shared | none)
  - Sets the default for all variables in this region.
- firstprivate(var-list)
  - Variables are private and are initialized with the value of the shared copy before the region.
- lastprivate(var-list)
  - Variables are private and the value of the thread executing the last iteration of a parallel loop in sequential order is copied to the variable outside of the region.

Scoping Variables with Private Clause

```
int i, j;
i = 1;
j = 2;

#pragma omp parallel private(i) firstprivate(j)
 {
  i = 3;
  j = j + 2;
  printf("%d %d\n", i, j);
 }
```

- The values of the shared copies of *i* and *j* are undefined on exit from the parallel region.
- The private copies of *j* are initialized in the parallel region to 2.

Lastprivate example

```
#pragma omp parallel
{
  #pragma omp for lastprivate(i)
  for (i=0; i<n-1; i++)
     a[i] = b[i] + b[i+1];
}
// The value of i is n-1
a[i]=b[i];
```

Copyprivate Example

```
#pragma omp parallel private(x)
{
  #pragma omp single copyprivate(x)
  {
    getValue(x);
  }
  useValue(x);
}
```

- Copyprivate
  - Clause only for single region.

Other Copyprivate Example

```
float read_next( ) {
float * tmp;
float return_val;
  #pragma omp single copyprivate(tmp)
  {
   tmp = (float *) malloc(sizeof(float));
  }
  #pragma omp master
  {
   get_float( tmp );
  }
  #pragma omp barrier
   return_val = *tmp;
  #pragma omp barrier
  #pragma omp single
  {
   free(tmp);
  }
  return return_val;
}
```

Runtime Routines for Threads (1)

- Determine the number of threads for parallel regions
  - omp_set_num_threads(count)
- Query the maximum number of threads for team creation
  - numthreads = omp_get_max_threads()
- Query number of threads in the current team
  - numthreads = omp_get_num_threads()
- Query own thread number (0..n-1)
  - iam = omp_get_thread_num()
- Query number of processors
  - numprocs = omp_get_num_procs()

Runtime Routines for Threads (2)

- Query state
  logicalvar = omp_in_parallel()
- Allow runtime system to determine the number of threads for team creation
  omp_set_dynamic(logicalexpr)
- Query whether runtime system can determine the number of threads
  logicalvar= omp_get_dynamic()
- Allow nesting of parallel regions
  omp_set_nested(logicalexpr)
- Query nesting of parallel regions
  logicalvar= omp_get_nested()

Simple Locks

- Locks can be hold by only one thread at a time.
- A lock is represented by a lock variable of type omp_lock_t.
- The thread that obtained a simple lock cannot set it again.

- Operations
  - omp_init_lock(&lockvar): initialize a lock
  - omp_destroy_lock(&lockvar): destroy a lock
  - omp_set_lock(&lockvar): set lock
  - omp_unset_lock(&lockvar): free lock
  - logicalvar = omp_test_lock(&lockvar): check lock and possibly set lock, returns true if lock was set by the executing thread.

Example: Simple Lock

```
#include <omp.h>
int id;
omp_lock_t lock;

omp_init_lock(lock);
#pragma omp parallel shared(lock) private(id)
{
  id = omp_get_thread_num();
  omp_set_lock(&lock); //Only a single thread writes
    printf("My Thread num is: %d", id);
  omp_unset_lock(&lock);

  WHILE (!omp_test_lock(&lock))
    other_work(id);     //Lock not obtained
  real_work(id);        //Lock obtained
  omp_unset_lock(&lock);//Lock freed
}
omp_destroy_lock(&lock);
```

locked

locked

Nestable Locks

- Unlike simple locks, nestable locks can be set multiple times by a single thread.
- Each set operation increments a lock counter.
- Each unset operation decrements the lock counter.
- If the lock counter is 0 after an unset operation, the lock can be set by another thread.

Environment Variables

- OMP_NUM_THREADS=4
  - Number of threads in a team of a parallel region
- OMP_SCHEDULE=”dynamic”, OMP_SCHEDULE=”GUIDED,4“#
  - Selects scheduling strategy to be applied at runtime
- OMP_DYNAMIC=TRUE
  - Allow runtime system to determine the number of threads.
- OMP_NESTED=TRUE
  - Allow nesting of parallel regions.

Summary

- OpenMP is quasi-standard for shared memory programming
- Based on Fork-Join Model
- Parallel region and work sharing constructs
  - Declaration of private or shared variables
  - Reduction variables
  - Scheduling strategies
- Synchronization via Barrier, Critical section, Atomic, locks, nestable locks
- Nested parallelism included in standard but not supported by compilers.