OpenMP

Application Program Interface

Introduction

Shared-memory parallelism in C, C++ and Fortran

- compiler directives
- library routines
- environment variables

Directives

- single program multiple data (SPMD) constructs
- tasking constructs
- worksharing constructs
- synchronization constructs
- with support for sharing and privatizing data

Compiler Support

Command line option to the compiler that activates and allows interpretation of OpenMP directives (-fopenmp for gcc)

```
#include <stdio.h>

int main()
{
  printf("ciao\n");

  return 0;
}
```

Conditional Compilation

In implementations that support a preprocessor, the _OPENMP macro name is defined to have the decimal value yyyymm where yyyy and mm are the year and month designations of the version of the OpenMP API that the implementation supports

```
int main()
{
#ifdef _OPENMP
  printf("OpenMP-compliant implementation.\n");
#endif

  return 0;
}
```

Conditional Compilation

Fortran

- Sentinels recognized in Fixed Form sources:

  `!$ | c$ | *$`

- Sentinels recognized in Free Form sources:

  `!$`

Threading Concepts

- thread: An execution entity with a stack and associated static memory, called threadprivate memory
- thread-safe routine: A routine that performs the intended function even when executed concurrently (by more than one thread)

Directives

C/C++

- OpenMP directives are specified by using the #pragma mechanism provided by the C and C++ standards

Fortran

- OpenMP directives are specified by using special comments that are identified by unique sentinels. Also, a special comment form is available for conditional compilation

- Compilers can therefore ignore OpenMP directives and conditionally compiled code if support of the OpenMP API is not provided or enabled

Directive Format

C/C++

- OpenMP directives for C/C++ are specified with the pragma preprocessing directive

```
#pragma omp directive-name [clause[[,] clause]...]
```

- Directives are case-sensitive
- An OpenMP executable directive applies to at most one succeeding statement, which must be a structured block

Directive Format

Fortran

- OpenMP directives for Fortran are specified as follows:

```
sentinel directive-name [clause[ [,] clause]...]
```

- Directives are case insensitive
- Directives cannot be embedded between statements
- Sentinels recognized in Fixed Form sources:

```
!$omp | c$omp | *$omp
```

- Sentinels recognized in Free Form sources:

```
!$omp
```

parallel Construct

- starts parallel execution
- the syntax of the parallel construct is as follows:

C/C++

```
#pragma omp parallel
structured-block
```

Fortran

```
!$omp parallel
structured-block
!$omp end parallel
```

parallel Construct

- When a thread encounters a parallel construct, a team of threads is created to execute the parallel region
- The thread that encountered the parallel construct becomes the master thread of the new team
- All threads in the new team, including the master thread, execute the region
- There is an implied barrier at the end of a parallel region
- After the end of a parallel region, only the master thread of the team resumes execution of the enclosing task region

```
int main()
{
#pragma omp parallel
  printf("ciao\n");

  return 0;
}
```

WRONG

```
int main()
{
  int i;

#pragma omp parallel
{
  for(i = 0; i < 10; ++i)
    printf("iteration %d\n", i);
}

  return 0;
}
```

Worksharing Constructs

- distribute the execution of the associated region among the members of the team that encounters it
- have no barrier on entry
- an implied barrier exists at the end of the worksharing region, unless a nowait clause is specified
- If a nowait clause is present threads that finish early may proceed straight to the instructions following the worksharing region without waiting for the other members of the team to finish the worksharing region

The following restrictions apply to worksharing constructs:

- Each worksharing region must be encountered by all threads in a team or by none at all
- The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team

Worksharing Constructs

The OpenMP API defines the following worksharing constructs:

- loop construct
- sections construct
- single construct
- workshare construct

loop Construct

- The loop construct specifies that the iterations of the associated loop will be executed in parallel by threads in the team
- The iterations are distributed across threads that already exist in the team executing the parallel region

```
#pragma omp for
for(init-expr; test-expr; incr-expr)
structured-block
```

loop Construct

- Keep init-expr, test-expr, incr-expr as simple as possible to avoid surprices!
- The iteration variable:
  - if shared, is implicitly made private in the loop construct
  - must not be modified during the execution of the for-loop other than in incr-expr
  - its value after the loop is unspecified

```
int main()
{
  int i;

#pragma omp parallel
{
#pragma omp for
  for(i = 0; i < 10; ++i)
    printf("%d\n", i);
}

  return 0;
}
```

loop Construct

Fortran

- The syntax of the loop construct is as follows:

```
!$omp do
do-loops
!$omp end do
```

WRONG

```
SUBROUTINE DO_WRONG
  INTEGER I, J

  DO 100 I = 1,10
    !$OMP DO
    DO 100 J = 1,10
      CALL WORK(I,J)
      100 CONTINUE
    !$OMP ENDDO
END SUBROUTINE DO_WRONG
```

loop Construct

- There is an implicit barrier at the end of a loop construct
- The only loop that is associated with the loop construct is the one that immediately follows the loop directive
- The schedule clause specifies how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team

schedule *kind*

```
#pragma omp directive-name [clause [ [,] clause]...]

#pragma omp for [clause [ [,] clause]...]

#pragma omp for schedule(kind)
```

The schedule kind can be one of the following:

- schedule(static, chunk_size)
- schedule(dynamic, chunk_size)
- schedule(guided, chunk_size)
- schedule(auto)
- schedule(runtime)

schedule static

- schedule(static, chunk_size)
  - iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number
  - When no chunk_size is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread

schedule dynamic

- schedule(dynamic, chunk_size)
  - the iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed
  - Each chunk contains chunk_size iterations
  - when no chunk_size is specified, it defaults to 1

schedule guided

- schedule(guided, chunk_size)
  - the iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned
  - for a chunk_size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1
  - for a chunk_size with value k, the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations
  - When no chunk_size is specified, it defaults to 1

loop Construct

- Different loop regions with the same schedule and iteration count, even if they occur in the same parallel region, can distribute iterations among threads differently
- Programs that depend on which thread executes a particular iteration under any other circumstances are non-conforming

sections Construct

- The sections construct is a noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team
- Each structured block is executed once by one of the threads in the team

The syntax of the sections construct is as follows:

C/C++

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

sections Construct

Fortran

```
!$omp sections [clause[[,] clause]...]
   [!$omp section]
      structured-block
   [!$omp section
      structured-block]
   ...
!$omp end sections [nowait]
```

sections Construct

```
#pragma omp parallel
{
#pragma omp sections
  {
#pragma omp section
    printf("section 1\n");
#pragma omp section
    printf("section 2\n");
  }
}
```

single Construct

The single construct specifies that the associated structured block is executed by only one of the threads in the team

The syntax of the single construct is as follows:

C/C++

```
#pragma omp single
structured-block
```

Fortran

```
!$omp single
structured-block
!$omp end single
```

```
#pragma omp parallel
{
#pragma omp single
  printf("ciao\n");
}
```

Combined Parallel Worksharing

Shortcuts for specifying a worksharing construct nested immediately inside a parallel construct

```
#pragma omp parallel for
```

```
#pragma omp parallel sections
```

```
!$omp parallel workshare
structured-block
!$omp end parallel workshare
```

Master and Synchronization

- master

- critical
- barrier
- atomic
- ordered

master Construct

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

```
#pragma omp parallel
{
#pragma omp master
  printf("ciao\n");
}
```

critical Construct

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

- The critical construct restricts execution of the associated structured block to a single thread at a time
- Region execution is restricted to a single thread at a time among all the threads in the program, without regard to the team(s) to which the threads belong
- An optional name may be used to identify the critical construct. All critical constructs without a name are considered to have the same unspecified name

critical Construct

```
#pragma omp parallel
{

#pragma omp
critical(long_and_strange_critical_name)
  doSomeCriticalWork_1();

#pragma omp critical
  doSomeCriticalWork_2();

#pragma omp critical
  doSomeCriticalWork_3();

}
```

barrier Construct

Specifies an explicit barrier at the point at which the construct appears

```
int counter = 0;

#pragma omp parallel
{
#pragma omp master
  counter = 1;

#pragma omp barrier

  printf("%d\n", counter);
}
```

atomic Construct

The atomic construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values

```
#pragma omp atomic
expression-stmt
```

where expression-stmt is an expression statement with one of the following forms:

x++;

x--;

++x;

--x;

x binop= expr;

x = x binop expr;

where binop is one of +, *, -, /, &, !^, |, <<, or >>

atomic Construct

- atomic regions enforce exclusive access with respect to other atomic regions that access the same storage location x among all the threads in the program without regard to the teams to which the threads belong
- Only the read and write of the location designated by x are performed mutually atomically

```
int counter = 0;

#pragma omp parallel
{
#pragma omp atomic
  ++counter;

#pragma omp barrier

  printf("%d\n", counter);
}
```

ordered Construct

- 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
- The loop region to which an ordered region binds must have an ordered clause specified on the corresponding loop (or parallel loop) construct

ordered Construct

```
void work(int k)
{
#pragma omp ordered
  printf(" %d\n", k);
}

void ordered_example(int lb, int ub, int stride)
{
  int i;

#pragma omp parallel for ordered schedule(dynamic)
  for(i = lb; i < ub; i += stride)
    work(i);
}

int main()
{
  ordered_example(0, 100, 5);

  return 0;
}
```

Data-Sharing Attribute Rules

C/C++

- Variables with automatic storage duration that are declared in a scope inside the construct are private
- Objects with dynamic storage duration are shared
- Variables with static storage duration that are declared in a scope inside the construct are shared
- Formal arguments of called routines in the region that are passed by reference inherit the data-sharing attributes of the associated actual argument
- Other variables declared in called routines in the region are private
- The loop iteration variable in the associated for-loop of a for or parallel for construct is private

Data-Sharing Attribute Rules

```
int h;

int main()
{
  int i;
  int c;
  void *d;

#pragma omp parallel
{
#pragma omp single
  d = malloc(10);

  work(1, d, &c);

#pragma omp for
  for(i = 0; i < 9; ++i);
}

  free(d);

  return 0;
}
```

```
void work(int a, void *p, int *g)
{
  static int f;
  int b;
  void *e = malloc(2);

  free(e);
}
```

Data-Sharing Attribute Rules

Fortran

- Variables and common blocks appearing in threadprivate directives are threadprivate
- The loop iteration variable(s) in the associated do-loop(s) of a do or parallel do construct is(are) private
- A loop iteration variable for a sequential loop in a parallel construct is private in the innermost such construct that encloses the loop
- Assumed-size arrays are shared
- Local variables declared in called routines in the region and that have the save attribute, or that are data initialized, are shared unless they appear in a threadprivate directive
- Variables belonging to common blocks, or declared in modules, and referenced in called routines in the region are shared unless they appear in a threadprivate directive
- Dummy arguments of called routines in the region that are passed by reference inherit the data-sharing attributes of the associated actual argument
- Implied-do indices and other local variables declared in called routines in the region are private.

Data-Sharing Attribute Clauses

```
#pragma omp directive-name [clause[[,] clause]...]
```

- Several constructs accept clauses that allow a user to control the data-sharing attributes of variables referenced in the construct
- Not all of the clauses listed in this section are valid on all directives
- Most of the clauses accept a comma-separated list of list items

default/shared/private Clause

default(none)

- The default(none) clause requires that each variable that is referenced in the construct must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause

valid on: parallel

shared(list)

- The shared clause declares one or more list items to be shared

valid on: parallel

private(list)

- The private clause declares one or more list items to be private
- A new list item of the same type is allocated for the construct
- The new list item has an undefined initial value

valid on: parallel, for, sections, single

default/shared/private Clause

```
int q;
int w;

#pragma omp parallel default(none) private(q) shared(w)
{
  q = 0;

#pragma omp single
  w = 0;

#pragma omp critical(stupid_application_stdout_critical)
  printf("%d %d\n", q, w);
}
```

firstprivate Clause

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

valid on: parallel, for, sections, single

```
int q = 3;
int w;

#pragma omp parallel default(none) firstprivate(q) shared(w)
{
#pragma omp single
  w = 0;

#pragma omp critical(stupid_application_stdout)
  printf("%d %d\n", q, w);
}
```

lastprivate Clauses

When a lastprivate clause appears on the directive that identifies a worksharing construct, the value of each new list item from the sequentially last iteration of the associated loops, or the lexically last section construct, is assigned to the original list item

valid on: for, sections

```
void lastpriv(int n, float *a, float *b)
{
  int i;

#pragma omp parallel
{
#pragma omp for lastprivate(i)
  for(i = 0; i < (n-1); ++i)
    a[i] = b[i] + b[i + 1];
}

  a[i] = b[i];
}
```

reduction Clause

- Specifies an operator and one or more list items
- For each list item, a private copy is created
- Each list item is initialized appropriately for the operator
- After the end of the region, the original list item is updated with the values of the private copies using the specified operator

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

valid on: parallel, for, sections

reduction Clause

```
int i;
int a = 5;

#pragma omp parallel
{
#pragma omp for reduction(+:a)
  for(i = 0; i < 10; ++i)
    ++a;
}

printf("%d\n", a);
```

reduction Clause

Fortran

```
reduction({operator | intrinsic_procedure_name}:list)
```

Example:

```
SUBROUTINE REDUCTION(A, B, C, D, X, Y, N)
  REAL :: X(*), A, D
  INTEGER :: Y(*), N, B, C
  INTEGER :: I
  A = 0
  B = 0
  C = Y(1)
  D = X(1)
  !$OMP PARALLEL DO PRIVATE(I) SHARED(X, Y, N) REDUCTION(+:A) &
  !$OMP& REDUCTION(IEOR:B) REDUCTION(MIN:C) REDUCTION(MAX:D)
  DO I=1,N
    A = A + X(I)
    B = IEOR(B, Y(I))
    C = MIN(C, Y(I))
    IF (D < X(I)) D = X(I)
  END DO
END SUBROUTINE REDUCTION
```

reduction Clause

Be afraid of Fortran features:

```
PROGRAM REDUCTION_WRONG
  MAX = HUGE(0)
  M = 0
  !$OMP PARALLEL DO REDUCTION(MAX: M)
  ! MAX is no longer the intrinsic so this is non-conforming
  DO I = 1, 100
    CALL SUB(M,I)
  END DO
END PROGRAM REDUCTION_WRONG

SUBROUTINE SUB(M,I)
  M = MAX(M,I)
END SUBROUTINE SUB
```

copyprivate Clause

Provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region

valid on: single

copyprivate Clause

```
float read_next()
{
  float * tmp;
  float return_val;

#pragma omp single copyprivate(tmp)
{
  tmp = (float *) malloc(sizeof(float));
} /* copies the pointer only */

#pragma omp master
{
  scanf("%f", tmp);
}

#pragma omp barrier
  return_val = *tmp;
#pragma omp barrier

#pragma omp single nowait
{
  free(tmp);
}
  return return_val;
}
```

Runtime Library Routines

- Prototypes for the C/C++ runtime library routines are provided in a header file named omp.h
- Interface declarations for the OpenMP Fortran runtime library routines are provided in the form of a Fortran include file named omp_lib.h or a Fortran 90 module named omp_lib

- `int omp_get_num_threads(void);`

  returns the number of threads in the current team

- `int omp_get_thread_num(void);`

  returns the thread number, within the current team, of the calling thread

- `double omp_get_wtime(void);`

  returns a value equal to the elapsed wall clock time in seconds since some "time in the past"

Environment Variables

- OMP_SCHEDULE: controls the schedule type and chunk size of all loop directives that have the schedule type runtime
- OMP_NUM_THREADS: sets the number of threads to use for parallel regions
- OMP_DYNAMIC: controls dynamic adjustment of the number of threads to use for executing parallel regions
- OMP_NESTED: controls nested parallelism
- OMP_STACKSIZE: controls the size of the stack for threads created by the OpenMP implementation

- bash

```
export OMP_SCHEDULE="dynamic"
```

- csh

```
setenv OMP_SCHEDULE "dynamic"
```

- DOS

```
set OMP_SCHEDULE=dynamic
```