OpenMP Application Programming Interface

Examples

Version 4.5.0 – November 2016

Source codes for OpenMP 4.5.0 Examples can be downloaded from github.

This page intentionally left blank

Contents

Introduction

This collection of programming examples supplements the OpenMP API for Shared Memory Parallelization specifications, and is not part of the formal specifications. It assumes familiarity with the OpenMP specifications, and shares the typographical conventions used in that document.

Note – This first release of the OpenMP Examples reflects the OpenMP Version 4.5 specifications. Additional examples are being developed and will be published in future releases of this document.

The OpenMP API specification provides a model for parallel programming that is portable across shared memory architectures from different vendors. Compilers from numerous vendors support the OpenMP API.

The directives, library routines, and environment variables demonstrated in this document allow users to create and manage parallel programs while permitting portability. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing and privatizing data. The functionality to control the runtime environment is provided by library routines and environment variables. Compilers that support the OpenMP API often include a command line option to the compiler that activates and allows interpretation of all OpenMP directives.

The latest source codes for OpenMP Examples can be downloaded from the **`sources`** directory at https://github.com/OpenMP/Examples. The codes for this OpenMP 4.5.0 Examples document have the tag *v4.5.0*.

Complete information about the OpenMP API and a list of the compilers that support the OpenMP API can be found at the OpenMP.org web site

`http://www.openmp.org`

Examples

The following are examples of the OpenMP API directives, constructs, and routines.

C / C++

A statement following a directive is compound only when necessary, and a non-compound statement is indented with respect to a directive preceding it.

C / C++

Each example is labeled as *ename.seqno.ext*, where *ename* is the example name, *seqno* is the sequence number in a section, and *ext* is the source file extension to indicate the code type and source form. *ext* is one of the following:

- *c* – C code,
- *cpp* – C++ code,
- *f* – Fortran code in fixed form, and
- *f90* – Fortran code in free form.

CHAPTER 1

Parallel Execution

A single thread, the *initial thread*, begins sequential execution of an OpenMP enabled program, as if the whole program is in an implicit parallel region consisting of an implicit task executed by the *initial thread*.

A **parallel** construct encloses code, forming a parallel region. An *initial thread* encountering a **parallel** region forks (creates) a team of threads at the beginning of the **parallel** region, and joins them (removes from execution) at the end of the region. The initial thread becomes the master thread of the team in a **parallel** region with a *thread* number equal to zero, the other threads are numbered from 1 to number of threads minus 1. A team may be comprised of just a single thread.

Each thread of a team is assigned an implicit task consisting of code within the parallel region. The task that creates a parallel region is suspended while the tasks of the team are executed. A thread is tied to its task; that is, only the thread assigned to the task can execute that task. After completion of the **parallel** region, the master thread resumes execution of the generating task.

Any task within a **parallel** region is allowed to encounter another **parallel** region to form a nested **parallel** region. The parallelism of a nested **parallel** region (whether it forks additional threads, or is executed serially by the encountering task) can be controlled by the **OMP_NESTED** environment variable or the **omp_set_nested()** API routine with arguments indicating true or false.

The number of threads of a **parallel** region can be set by the **OMP_NUM_THREADS** environment variable, the **omp_set_num_threads()** routine, or on the **parallel** directive with the **num_threads** clause. The routine overrides the environment variable, and the clause overrides all. Use the **OMP_DYNAMIC** or the **omp_set_dynamic()** function to specify that the OpenMP implementation dynamically adjust the number of threads for **parallel** regions. The default setting for dynamic adjustment is implementation defined. When dynamic adjustment is on and the number of threads is specified, the number of threads becomes an upper limit for the number of threads to be provided by the OpenMP runtime.

WORKSHARING CONSTRUCTS

A worksharing construct distributes the execution of the associated region among the members of the team that encounter it. There is an implied barrier at the end of the worksharing region (there is no barrier at the beginning). The worksharing constructs are:

- loop constructs: **for** and **do**
- **sections**
- **single**
- **workshare**

The **for** and **do** constructs (loop constructs) create a region consisting of a loop. A loop controlled by a loop construct is called an *associated* loop. Nested loops can form a single region when the **collapse** clause (with an integer argument) designates the number of *associated* loops to be executed in parallel, by forming a "single iteration space" for the specified number of nested loops. The **ordered** clause can also control multiple associated loops.

An associated loop must adhere to a "canonical form" (specified in the *Canonical Loop Form* of the OpenMP Specifications document) which allows the iteration count (of all associated loops) to be computed before the (outermost) loop is executed. Most common loops comply with the canonical form, including C++ iterators.

A **single** construct forms a region in which only one thread (any one of the team) executes the region. The other threads wait at the implied barrier at the end, unless the **nowait** clause is specified.

The **sections** construct forms a region that contains one or more structured blocks. Each block of a **sections** directive is constructed with a **section** construct, and executed once by one of the threads (any one) in the team. (If only one block is formed in the region, the **section** construct, which is used to separate blocks, is not required.) The other threads wait at the implied barrier at the end, unless the **nowait** clause is specified.

The **workshare** construct is a Fortran feature that consists of a region with a single structure block (section of code). Statements in the **workshare** region are divided into units of work, and executed (once) by threads of the team.

MASTER CONSTRUCT

The **master** construct is not a worksharing construct. The master region is is executed only by the master thread. There is no implicit barrier (and flush) at the end of the **master** region; hence the other threads of the team continue execution beyond code statements beyond the **master** region.

1.1 A Simple Parallel Loop

The following example demonstrates how to parallelize a simple loop using the parallel loop construct. The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a **`private`** clause.

C / C++

Example ploop.1.c

```
S-1  void simple(int n, float *a, float *b)
S-2  {
S-3      int i;
S-4
S-5  #pragma omp parallel for
S-6      for (i=1; i<n; i++) /* i is private by default */
S-7          b[i] = (a[i] + a[i-1]) / 2.0;
S-8  }
```

C / C++

1.2 The `parallel` Construct

The **parallel** construct can be used in coarse-grain parallel programs. In the following example, each thread in the **parallel** region decides what part of the global array *x* to work on, based on the thread number:

C / C++

Example parallel.1.c

```
S-1   #include <omp.h>
S-2
S-3   void subdomain(float *x, int istart, int ipoints)
S-4   {
S-5     int i;
S-6
S-7     for (i = 0; i < ipoints; i++)
S-8         x[istart+i] = 123.456;
S-9   }
S-10
S-11  void sub(float *x, int npoints)
S-12  {
S-13      int iam, nt, ipoints, istart;
S-14
S-15  #pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
S-16      {
S-17          iam = omp_get_thread_num();
S-18          nt =  omp_get_num_threads();
S-19          ipoints = npoints / nt;     /* size of partition */
S-20          istart = iam * ipoints;  /* starting array index */
S-21          if (iam == nt-1)      /* last thread may do more */
S-22            ipoints = npoints - istart;
S-23          subdomain(x, istart, ipoints);
S-24      }
S-25  }
S-26
S-27  int main()
S-28  {
S-29      float array[10000];
S-30
S-31      sub(array, 10000);
S-32
S-33      return 0;
S-34  }
```

C / C++

1.3 Controlling the Number of Threads on Multiple Nesting Levels

The following examples demonstrate how to use the **OMP_NUM_THREADS** environment variable to control the number of threads on multiple nesting levels:

C / C++

Example nthrs_nesting.1.c

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3   int main (void)
S-4   {
S-5      omp_set_nested(1);
S-6      omp_set_dynamic(0);
S-7      #pragma omp parallel
S-8      {
S-9         #pragma omp parallel
S-10        {
S-11           #pragma omp single
S-12           {
S-13           /*
S-14           * If OMP_NUM_THREADS=2,3 was set, the following should print:
S-15           * Inner: num_thds=3
S-16           * Inner: num_thds=3
S-17           *
S-18           * If nesting is not supported, the following should print:
S-19           * Inner: num_thds=1
S-20           * Inner: num_thds=1
S-21           */
S-22              printf ("Inner: num_thds=%d\n", omp_get_num_threads());
S-23           }
S-24        }
S-25        #pragma omp barrier
S-26        omp_set_nested(0);
S-27        #pragma omp parallel
S-28        {
S-29           #pragma omp single
S-30           {
S-31           /*
S-32           * Even if OMP_NUM_THREADS=2,3 was set, the following should
S-33           * print, because nesting is disabled:
S-34           * Inner: num_thds=1
S-35           * Inner: num_thds=1
S-36           */
S-37              printf ("Inner: num_thds=%d\n", omp_get_num_threads());
S-38           }
```

```
S-39          }
S-40          #pragma omp barrier
S-41          #pragma omp single
S-42          {
S-43             /*
S-44             * If OMP_NUM_THREADS=2,3 was set, the following should print:
S-45             * Outer: num_thds=2
S-46             */
S-47             printf ("Outer: num_thds=%d\n", omp_get_num_threads());
S-48          }
S-49       }
S-50       return 0;
S-51    }
```

C / C++

1.4 Interaction Between the `num_threads` Clause and `omp_set_dynamic`

The following example demonstrates the **num_threads** clause and the effect of the **omp_set_dynamic** routine on it.

The call to the **omp_set_dynamic** routine with argument **0** in C/C++, or **.FALSE.** in Fortran, disables the dynamic adjustment of the number of threads in OpenMP implementations that support it. In this case, 10 threads are provided. Note that in case of an error the OpenMP implementation is free to abort the program or to supply any number of threads available.

C / C++

Example nthrs_dynamic.1.c

```
S-1   #include <omp.h>
S-2   int main()
S-3   {
S-4     omp_set_dynamic(0);
S-5     #pragma omp parallel num_threads(10)
S-6     {
S-7       /* do work here */
S-8     }
S-9     return 0;
S-10  }
```

C / C++

The call to the **omp_set_dynamic** routine with a non-zero argument in C/C++, or **.TRUE.** in Fortran, allows the OpenMP implementation to choose any number of threads between 1 and 10.

C / C++

Example nthrs_dynamic.2.c

```
S-1   #include <omp.h>
S-2   int main()
S-3   {
S-4     omp_set_dynamic(1);
S-5     #pragma omp parallel num_threads(10)
S-6     {
S-7       /* do work here */
S-8     }
S-9     return 0;
S-10  }
```

C / C++

It is good practice to set the *dyn-var* ICV explicitly by calling the **omp_set_dynamic** routine, as its default setting is implementation defined.

1.5 Fortran Restrictions on the `do` Construct

Fortran

If an **`end do`** directive follows a *do-construct* in which several **`DO`** statements share a **`DO`** termination statement, then a **`do`** directive can only be specified for the outermost of these **`DO`** statements. The following example contains correct usages of loop constructs:

The following example is non-conforming because the matching **`do`** directive for the **`end do`** does not precede the outermost loop:

Fortran

1.6 The `nowait` Clause

If there are multiple independent loops within a **parallel** region, you can use the **nowait** clause to avoid the implied barrier at the end of the loop construct, as follows:

C / C++

Example nowait.1.c

```
S-1   #include <math.h>
S-2
S-3   void nowait_example(int n, int m, float *a, float *b, float *y, float *z)
S-4   {
S-5     int i;
S-6     #pragma omp parallel
S-7     {
S-8       #pragma omp for nowait
S-9         for (i=1; i<n; i++)
S-10          b[i] = (a[i] + a[i-1]) / 2.0;
S-11
S-12      #pragma omp for nowait
S-13        for (i=0; i<m; i++)
S-14          y[i] = sqrt(z[i]);
S-15    }
S-16  }
```

C / C++

In the following example, static scheduling distributes the same logical iteration numbers to the threads that execute the three loop regions. This allows the **nowait** clause to be used, even though there is a data dependence between the loops. The dependence is satisfied as long the same thread executes the same logical iteration numbers in each loop.

Note that the iteration count of the loops must be the same. The example satisfies this requirement, since the iteration space of the first two loops is from **0** to **n-1** (from **1** to **N** in the Fortran version), while the iteration space of the last loop is from **1** to **n** (**2** to **N+1** in the Fortran version).

C / C++

Example nowait.2.c

```
S-1
S-2   #include <math.h>
S-3   void nowait_example2(int n, float *a, float *b, float *c, float *y, float
S-4   *z)
S-5   {
S-6      int i;
S-7   #pragma omp parallel
S-8      {
S-9   #pragma omp for schedule(static) nowait
S-10     for (i=0; i<n; i++)
S-11        c[i] = (a[i] + b[i]) / 2.0f;
S-12  #pragma omp for schedule(static) nowait
S-13     for (i=0; i<n; i++)
S-14        z[i] = sqrtf(c[i]);
S-15  #pragma omp for schedule(static) nowait
S-16     for (i=1; i<=n; i++)
S-17        y[i] = z[i-1] + a[i];
S-18     }
S-19  }
```

C / C++

1.7 The `collapse` Clause

In the following example, the **k** and **j** loops are associated with the loop construct. So the iterations of the **k** and **j** loops are collapsed into one loop with a larger iteration space, and that loop is then divided among the threads in the current team. Since the **i** loop is not associated with the loop construct, it is not collapsed, and the **i** loop is executed sequentially in its entirety in every iteration of the collapsed **k** and **j** loop.

The variable **j** can be omitted from the **private** clause when the **collapse** clause is used since it is implicitly private. However, if the **collapse** clause is omitted then **j** will be shared if it is omitted from the **private** clause. In either case, **k** is implicitly private and could be omitted from the **private** clause.

C / C++

Example collapse.1.c

```
S-1   void bar(float *a, int i, int j, int k);
S-2   int kl, ku, ks, jl, ju, js, il, iu,is;
S-3   void sub(float *a)
S-4   {
S-5       int i, j, k;
S-6       #pragma omp for collapse(2) private(i, k, j)
S-7       for (k=kl; k<=ku; k+=ks)
S-8          for (j=jl; j<=ju; j+=js)
S-9             for (i=il; i<=iu; i+=is)
S-10               bar(a,i,j,k);
S-11  }
```

C / C++

In the next example, the **k** and **j** loops are associated with the loop construct. So the iterations of the **k** and **j** loops are collapsed into one loop with a larger iteration space, and that loop is then divided among the threads in the current team.

The sequential execution of the iterations in the **k** and **j** loops determines the order of the iterations in the collapsed iteration space. This implies that in the sequentially last iteration of the collapsed iteration space, **k** will have the value **2** and **j** will have the value **3**. Since **klast** and **jlast** are **lastprivate**, their values are assigned by the sequentially last iteration of the collapsed **k** and **j** loop. This example prints: **2 3**.

C / C++

Example collapse.2.c

```
S-1   #include <stdio.h>
S-2   void test()
S-3   {
S-4      int j, k, jlast, klast;
S-5      #pragma omp parallel
S-6      {
S-7         #pragma omp for collapse(2) lastprivate(jlast, klast)
S-8         for (k=1; k<=2; k++)
S-9            for (j=1; j<=3; j++)
S-10           {
S-11              jlast=j;
S-12              klast=k;
S-13           }
S-14        #pragma omp single
S-15        printf("%d %d\n", klast, jlast);
S-16     }
S-17  }
```

C / C++

The next example illustrates the interaction of the **collapse** and **ordered** clauses.

In the example, the loop construct has both a **collapse** clause and an **ordered** clause. The **collapse** clause causes the iterations of the **k** and **j** loops to be collapsed into one loop with a larger iteration space, and that loop is divided among the threads in the current team. An **ordered** clause is added to the loop construct, because an ordered region binds to the loop region arising from the loop construct.

According to Section 2.12.8 of the OpenMP 4.0 specification, a thread must not execute more than one ordered region that binds to the same loop region. So the **collapse** clause is required for the example to be conforming. With the **collapse** clause, the iterations of the **k** and **j** loops are collapsed into one loop, and therefore only one ordered region will bind to the collapsed **k** and **j** loop. Without the **collapse** clause, there would be two ordered regions that bind to each iteration of the **k** loop (one arising from the first iteration of the **j** loop, and the other arising from the second iteration of the **j** loop).

The code prints

```
0 1 1
0 1 2
0 2 1
1 2 2
1 3 1
1 3 2
```

C / C++

Example collapse.3.c

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3   void work(int a, int j, int k);
S-4   void sub()
S-5   {
S-6      int j, k, a;
S-7      #pragma omp parallel num_threads(2)
S-8      {
S-9         #pragma omp for collapse(2) ordered private(j,k) schedule(static,3)
S-10        for (k=1; k<=3; k++)
S-11           for (j=1; j<=2; j++)
S-12           {
S-13              #pragma omp ordered
S-14              printf("%d %d %d\n", omp_get_thread_num(), k, j);
S-15              /* end ordered */
S-16              work(a,j,k);
S-17           }
S-18     }
S-19  }
```

C / C++

1.8 `linear` Clause in Loop Constructs

The following example shows the use of the **`linear`** clause in a loop construct to allow the proper parallelization of a loop that contains an induction variable (*j*). At the end of the execution of the loop construct, the original variable *j* is updated with the value *N/2* from the last iteration of the loop.

C / C++

Example linear_in_loop.1.c

```
S-1   #include <stdio.h>
S-2
S-3   #define N 100
S-4   int main(void)
S-5   {
S-6      float a[N], b[N/2];
S-7      int i, j;
S-8
S-9      for ( i = 0; i < N; i++ )
S-10        a[i] = i + 1;
S-11
S-12     j = 0;
S-13     #pragma omp parallel
S-14     #pragma omp for linear(j:1)
S-15     for ( i = 0; i < N; i += 2 ) {
S-16        b[j] = a[i] * 2.0f;
S-17        j++;
S-18     }
S-19
S-20     printf( "%d %f %f\n", j, b[0], b[j-1] );
S-21     /* print out: 50 2.0 198.0 */
S-22
S-23     return 0;
S-24  }
```

C / C++

1.9 The `parallel sections` Construct

In the following example routines **XAXIS**, **YAXIS**, and **ZAXIS** can be executed concurrently. The first **section** directive is optional. Note that all **section** directives need to appear in the **parallel sections** construct.

C / C++

Example psections.1.c

```
S-1   void XAXIS();
S-2   void YAXIS();
S-3   void ZAXIS();
S-4
S-5   void sect_example()
S-6   {
S-7     #pragma omp parallel sections
S-8     {
S-9       #pragma omp section
S-10        XAXIS();
S-11
S-12      #pragma omp section
S-13        YAXIS();
S-14
S-15      #pragma omp section
S-16        ZAXIS();
S-17    }
S-18  }
```

C / C++

1.10 The `firstprivate` Clause and the `sections` Construct

In the following example of the **`sections`** construct the **`firstprivate`** clause is used to initialize the private copy of **`section_count`** of each thread. The problem is that the **`section`** constructs modify **`section_count`**, which breaks the independence of the **`section`** constructs. When different threads execute each section, both sections will print the value 1. When the same thread executes the two sections, one section will print the value 1 and the other will print the value 2. Since the order of execution of the two sections in this case is unspecified, it is unspecified which section prints which value.

C / C++

Example fpriv_sections.1.c

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3   #define NT 4
S-4   int main( ) {
S-5       int section_count = 0;
S-6       omp_set_dynamic(0);
S-7       omp_set_num_threads(NT);
S-8   #pragma omp parallel
S-9   #pragma omp sections firstprivate( section_count )
S-10  {
S-11  #pragma omp section
S-12      {
S-13          section_count++;
S-14          /* may print the number one or two */
S-15          printf( "section_count %d\n", section_count );
S-16      }
S-17  #pragma omp section
S-18      {
S-19          section_count++;
S-20          /* may print the number one or two */
S-21          printf( "section_count %d\n", section_count );
S-22      }
S-23  }
S-24      return 0;
S-25  }
```

C / C++

1.11 The `single` Construct

The following example demonstrates the **`single`** construct. In the example, only one thread prints each of the progress messages. All other threads will skip the **`single`** region and stop at the barrier at the end of the **`single`** construct until all threads in the team have reached the barrier. If other threads can proceed without waiting for the thread executing the **`single`** region, a **`nowait`** clause can be specified, as is done in the third **`single`** construct in this example. The user must not make any assumptions as to which thread will execute a **`single`** region.

C / C++

Example single.1.c

```
S-1   #include <stdio.h>
S-2
S-3   void work1() {}
S-4   void work2() {}
S-5
S-6   void single_example()
S-7   {
S-8     #pragma omp parallel
S-9     {
S-10      #pragma omp single
S-11        printf("Beginning work1.\n");
S-12
S-13      work1();
S-14
S-15      #pragma omp single
S-16        printf("Finishing work1.\n");
S-17
S-18      #pragma omp single nowait
S-19        printf("Finished work1 and beginning work2.\n");
S-20
S-21      work2();
S-22    }
S-23  }
```

C / C++

Fortran (cont.)

Fortran (cont.)

1.12 The `workshare` Construct

Fortran

The following are examples of the **workshare** construct.

In the following example, **workshare** spreads work across the threads executing the **parallel** region, and there is a barrier after the last statement. Implementations must enforce Fortran execution rules inside of the **workshare** block.

In the following example, the barrier at the end of the first **workshare** region is eliminated with a **nowait** clause. Threads doing **CC = DD** immediately begin work on **EE = FF** when they are done with **CC = DD**.

The following example shows the use of an **atomic** directive inside a **workshare** construct. The computation of **SUM(AA)** is workshared, but the update to **R** is atomic.

Fortran **WHERE** and **FORALL** statements are *compound statements*, made up of a *control* part and a *statement* part. When **workshare** is applied to one of these compound statements, both the control and the statement parts are workshared. The following example shows the use of a **WHERE** statement in a **workshare** construct.

Each task gets worked on in order by the threads:

AA = BB then
CC = DD then
EE .ne. 0 then
FF = 1 / EE then
GG = HH

In the following example, an assignment to a shared scalar variable is performed by one thread in a **workshare** while all other threads in the team wait.

The following example contains an assignment to a private scalar variable, which is performed by one thread in a **workshare** while all other threads wait. It is non-conforming because the private scalar variable is undefined after the assignment statement.

Fortran execution rules must be enforced inside a **workshare** construct. In the following example, the same result is produced in the following program fragment regardless of whether the code is executed sequentially or inside an OpenMP program with multiple threads:

Fortran

1.13 The `master` Construct

The following example demonstrates the master construct . In the example, the master keeps track of how many iterations have been executed and prints out a progress report. The other threads skip the master region without waiting.

C / C++

Example master.1.c

```
S-1   #include <stdio.h>
S-2
S-3   extern float average(float,float,float);
S-4
S-5   void master_example( float* x, float* xold, int n, float tol )
S-6   {
S-7     int c, i, toobig;
S-8     float error, y;
S-9     c = 0;
S-10    #pragma omp parallel
S-11    {
S-12      do{
S-13        #pragma omp for private(i)
S-14        for( i = 1; i < n-1; ++i ){
S-15          xold[i] = x[i];
S-16        }
S-17        #pragma omp single
S-18        {
S-19          toobig = 0;
S-20        }
S-21        #pragma omp for private(i,y,error) reduction(+:toobig)
S-22        for( i = 1; i < n-1; ++i ){
S-23          y = x[i];
S-24          x[i] = average( xold[i-1], x[i], xold[i+1] );
S-25          error = y - x[i];
S-26          if( error > tol || error < -tol ) ++toobig;
S-27        }
S-28        #pragma omp master
S-29        {
S-30          ++c;
S-31          printf( "iteration %d, toobig=%d\n", c, toobig );
S-32        }
S-33      }while( toobig > 0 );
S-34    }
S-35  }
```

C / C++

1.14 Parallel Random Access Iterator Loop

C++

The following example shows a parallel random access iterator loop.

Example pra_iterator.1.cpp

```
S-1   #include <vector>
S-2   void iterator_example()
S-3   {
S-4     std::vector<int> vec(23);
S-5     std::vector<int>::iterator it;
S-6   #pragma omp parallel for default(none) shared(vec)
S-7     for (it = vec.begin(); it < vec.end(); it++)
S-8     {
S-9       // do work with *it //
S-10    }
S-11  }
```

C++

1.15 The `omp_set_dynamic` and `omp_set_num_threads` Routines

Some programs rely on a fixed, prespecified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using **`omp_set_dynamic`**, and **`omp_set_num_threads`**.

In this example, the program executes correctly only if it is executed by 16 threads. If the implementation is not capable of supporting 16 threads, the behavior of this example is implementation defined. Note that the number of threads executing a **`parallel`** region remains constant during the region, regardless of the dynamic threads setting. The dynamic threads mechanism determines the number of threads to use at the start of the **`parallel`** region and keeps it constant for the duration of the region.

C / C++

Example set_dynamic_nthrs.1.c

```
S-1   #include <omp.h>
S-2   #include <stdlib.h>
S-3
S-4   void do_by_16(float *x, int iam, int ipoints) {}
S-5
S-6   void dynthreads(float *x, int npoints)
S-7   {
S-8     int iam, ipoints;
S-9
S-10    omp_set_dynamic(0);
S-11    omp_set_num_threads(16);
S-12
S-13    #pragma omp parallel shared(x, npoints) private(iam, ipoints)
S-14    {
S-15      if (omp_get_num_threads() != 16)
S-16        abort();
S-17
S-18      iam = omp_get_thread_num();
S-19      ipoints = npoints/16;
S-20      do_by_16(x, iam, ipoints);
S-21    }
S-22  }
```

C / C++

1.16 The `omp_get_num_threads` Routine

In the following example, the **omp_get_num_threads** call returns 1 in the sequential part of the code, so **np** will always be equal to 1. To determine the number of threads that will be deployed for the **parallel** region, the call should be inside the **parallel** region.

C / C++

Example get_nthrs.1.c

```
S-1   #include <omp.h>
S-2   void work(int i);
S-3
S-4   void incorrect()
S-5   {
S-6     int np, i;
S-7
S-8     np = omp_get_num_threads();  /* misplaced */
S-9
S-10    #pragma omp parallel for schedule(static)
S-11    for (i=0; i < np; i++)
S-12      work(i);
S-13  }
```

C / C++

The following example shows how to rewrite this program without including a query for the number of threads:

C / C++

Example get_nthrs.2.c

```
S-1   #include <omp.h>
S-2   void work(int i);
S-3
S-4   void correct()
S-5   {
S-6     int i;
S-7
S-8     #pragma omp parallel private(i)
S-9     {
S-10      i = omp_get_thread_num();
S-11      work(i);
S-12    }
S-13  }
```

C / C++

CHAPTER 2

OpenMP Affinity

OpenMP Affinity consists of a **proc_bind** policy (thread affinity policy) and a specification of places ("location units" or *processors* that may be cores, hardware threads, sockets, etc.). OpenMP Affinity enables users to bind computations on specific places. The placement will hold for the duration of the parallel region. However, the runtime is free to migrate the OpenMP threads to different cores (hardware threads, sockets, etc.) prescribed within a given place, if two or more cores (hardware threads, sockets, etc.) have been assigned to a given place.

Often the binding can be managed without resorting to explicitly setting places. Without the specification of places in the **OMP_PLACES** variable, the OpenMP runtime will distribute and bind threads using the entire range of processors for the OpenMP program, according to the **OMP_PROC_BIND** environment variable or the **proc_bind** clause. When places are specified, the OMP runtime binds threads to the places according to a default distribution policy, or those specified in the **OMP_PROC_BIND** environment variable or the **proc_bind** clause.

In the OpenMP Specifications document a processor refers to an execution unit that is enabled for an OpenMP thread to use. A processor is a core when there is no SMT (Simultaneous Multi-Threading) support or SMT is disabled. When SMT is enabled, a processor is a hardware thread (HW-thread). (This is the usual case; but actually, the execution unit is implementation defined.) Processor numbers are numbered sequentially from 0 to the number of cores less one (without SMT), or 0 to the number HW-threads less one (with SMT). OpenMP places use the processor number to designate binding locations (unless an "abstract name" is used.)

The processors available to a process may be a subset of the system's processors. This restriction may be the result of a wrapper process controlling the execution (such as **numactl** on Linux systems), compiler options, library-specific environment variables, or default kernel settings. For instance, the execution of multiple MPI processes, launched on a single compute node, will each have a subset of processors as determined by the MPI launcher or set by MPI affinity environment variables for the MPI library.

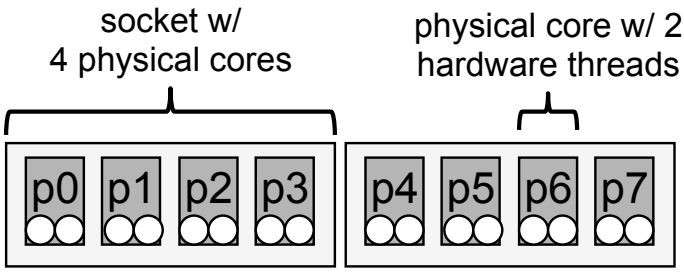
Threads of a team are positioned onto places in a compact manner, a scattered distribution, or onto the master's place, by setting the **OMP_PROC_BIND** environment variable or the **proc_bind**

clause to *close*, *spread*, or *master*, respectively. When **OMP_PROC_BIND** is set to FALSE no binding is enforced; and when the value is TRUE, the binding is implementation defined to a set of places in the **OMP_PLACES** variable or to places defined by the implementation if the **OMP_PLACES** variable is not set.

The **OMP_PLACES** variable can also be set to an abstract name (*threads*, *cores*, *sockets*) to specify that a place is either a single hardware thread, a core, or a socket, respectively. This description of the **OMP_PLACES** is most useful when the number of threads is equal to the number of hardware thread, cores or sockets. It can also be used with a *close* or *spread* distribution policy when the equality doesn't hold.

2.1 The `proc_bind` Clause

The following examples demonstrate how to use the **`proc_bind`** clause to control the thread binding for a team of threads in a **`parallel`** region. The machine architecture is depicted in the figure below. It consists of two sockets, each equipped with a quad-core processor and configured to execute two hardware threads simultaneously on each core. These examples assume a contiguous core numbering starting from 0, such that the hardware threads 0,1 form the first physical core.



The following equivalent place list declarations consist of eight places (which we designate as p0 to p7):

```
OMP_PLACES="{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"
```

or

```
OMP_PLACES="{0:2}:8:2"
```

2.1.1 Spread Affinity Policy

The following example shows the result of the **`spread`** affinity policy on the partition list when the number of threads is less than or equal to the number of places in the parent's place partition, for the machine architecture depicted above. Note that the threads are bound to the first place of each subpartition.

C / C++

Example affinity.1.c

```
S-1  void work();
S-2  int main()
S-3  {
S-4  #pragma omp parallel proc_bind(spread) num_threads(4)
S-5     {
S-6        work();
S-7     }
S-8     return 0;
S-9  }
```

C / C++

It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the parallel region:

- thread 0 executes on p0 with the place partition p0,p1
- thread 1 executes on p2 with the place partition p2,p3
- thread 2 executes on p4 with the place partition p4,p5
- thread 3 executes on p6 with the place partition p6,p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- thread 0 executes on p2 with the place partition p2,p3
- thread 1 executes on p4 with the place partition p4,p5
- thread 2 executes on p6 with the place partition p6,p7
- thread 3 executes on p0 with the place partition p0,p1

The following example illustrates the **spread** thread affinity policy when the number of threads is greater than the number of places in the parent's place partition.

Let *T* be the number of threads in the team, and *P* be the number of places in the parent's place partition. The first *T/P* threads of the team (including the master thread) execute on the parent's place. The next *T/P* threads execute on the next place in the place partition, and so on, with wrap around.

C / C++

Example affinity.2.c

```
S-1  void work();
S-2  void foo()
S-3  {
S-4    #pragma omp parallel num_threads(16) proc_bind(spread)
S-5    {
S-6      work();
S-7    }
S-8  }
```

C / C++

It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the parallel region:

- threads 0,1 execute on p0 with the place partition p0
- threads 2,3 execute on p1 with the place partition p1
- threads 4,5 execute on p2 with the place partition p2
- threads 6,7 execute on p3 with the place partition p3
- threads 8,9 execute on p4 with the place partition p4
- threads 10,11 execute on p5 with the place partition p5

- threads 12,13 execute on p6 with the place partition p6
- threads 14,15 execute on p7 with the place partition p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0,1 execute on p2 with the place partition p2
- threads 2,3 execute on p3 with the place partition p3
- threads 4,5 execute on p4 with the place partition p4
- threads 6,7 execute on p5 with the place partition p5
- threads 8,9 execute on p6 with the place partition p6
- threads 10,11 execute on p7 with the place partition p7
- threads 12,13 execute on p0 with the place partition p0
- threads 14,15 execute on p1 with the place partition p1

2.1.2 Close Affinity Policy

The following example shows the result of the **close** affinity policy on the partition list when the number of threads is less than or equal to the number of places in parent's place partition, for the machine architecture depicted above. The place partition is not changed by the **close** policy.

C / C++

Example affinity.3.c

```
S-1  void work();
S-2  int main()
S-3  {
S-4  #pragma omp parallel proc_bind(close) num_threads(4)
S-5     {
S-6        work();
S-7     }
S-8     return 0;
S-9  }
```

C / C++

It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the **parallel** region:

- thread 0 executes on p0 with the place partition p0-p7
- thread 1 executes on p1 with the place partition p0-p7
- thread 2 executes on p2 with the place partition p0-p7
- thread 3 executes on p3 with the place partition p0-p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- thread 0 executes on p2 with the place partition p0-p7
- thread 1 executes on p3 with the place partition p0-p7
- thread 2 executes on p4 with the place partition p0-p7
- thread 3 executes on p5 with the place partition p0-p7

The following example illustrates the **close** thread affinity policy when the number of threads is greater than the number of places in the parent's place partition.

Let *T* be the number of threads in the team, and *P* be the number of places in the parent's place partition. The first *T/P* threads of the team (including the master thread) execute on the parent's place. The next *T/P* threads execute on the next place in the place partition, and so on, with wrap around. The place partition is not changed by the **close** policy.

C / C++

Example affinity.4.c

```
S-1  void work();
S-2  void foo()
S-3  {
S-4    #pragma omp parallel num_threads(16) proc_bind(close)
S-5    {
S-6      work();
S-7    }
S-8  }
```

C / C++

It is unspecified on which place the master thread is initially started. If the master thread is initially running on p0, the following placement of threads will be applied in the parallel region:

- threads 0,1 execute on p0 with the place partition p0-p7
- threads 2,3 execute on p1 with the place partition p0-p7
- threads 4,5 execute on p2 with the place partition p0-p7
- threads 6,7 execute on p3 with the place partition p0-p7
- threads 8,9 execute on p4 with the place partition p0-p7
- threads 10,11 execute on p5 with the place partition p0-p7
- threads 12,13 execute on p6 with the place partition p0-p7
- threads 14,15 execute on p7 with the place partition p0-p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0,1 execute on p2 with the place partition p0-p7
- threads 2,3 execute on p3 with the place partition p0-p7
- threads 4,5 execute on p4 with the place partition p0-p7

- threads 6,7 execute on p5 with the place partition p0-p7
- threads 8,9 execute on p6 with the place partition p0-p7
- threads 10,11 execute on p7 with the place partition p0-p7
- threads 12,13 execute on p0 with the place partition p0-p7
- threads 14,15 execute on p1 with the place partition p0-p7

2.1.3 Master Affinity Policy

The following example shows the result of the **master** affinity policy on the partition list for the machine architecture depicted above. The place partition is not changed by the master policy.

C / C++

Example affinity.5.c

```
S-1  void work();
S-2  int main()
S-3  {
S-4  #pragma omp parallel proc_bind(master) num_threads(4)
S-5     {
S-6        work();
S-7     }
S-8     return 0;
S-9  }
```

C / C++

It is unspecified on which place the master thread is initially started. If the master thread is initially running on p0, the following placement of threads will be applied in the parallel region:

- threads 0-3 execute on p0 with the place partition p0-p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0-3 execute on p2 with the place partition p0-p7

2.2 Affinity Query Functions

In the example below a team of threads is generated on each socket of the system, using nested parallelism. Several query functions are used to gather information to support the creation of the teams and to obtain socket and thread numbers.

For proper execution of the code, the user must create a place partition, such that each place is a listing of the core numbers for a socket. For example, in a 2 socket system with 8 cores in each socket, and sequential numbering in the socket for the core numbers, the **OMP_PLACES** variable would be set to "{0:8},{8:8}", using the place syntax {*lower_bound*:*length*:*stride*}, and the default stride of 1.

The code determines the number of sockets (*n_sockets*) using the **omp_get_num_places()** query function. In this example each place is constructed with a list of each socket's core numbers, hence the number of places is equal to the number of sockets.

The outer parallel region forms a team of threads, and each thread executes on a socket (place) because the **proc_bind** clause uses **spread** in the outer **parallel** construct. Next, in the *socket_init* function, an inner parallel region creates a team of threads equal to the number of elements (core numbers) from the place of the parent thread. Because the outer **parallel** construct uses a **spread** affinity policy, each of its threads inherits a subpartition of the original partition. Hence, the **omp_get_place_num_procs** query function returns the number of elements (here procs = cores) in the subpartition of the thread. After each parent thread creates its nested parallel region on the section, the socket number and thread number are reported.

Note: Portable tools like hwloc (Portable HardWare LOCality package), which support many common operating systems, can be used to determine the configuration of a system. On some systems there are utilities, files or user guides that provide configuration information. For instance, the socket number and proc_id's for a socket can be found in the /proc/cpuinfo text file on Linux systems.

C / C++

Example affinity.6.c

```
S-1
S-2     #include <stdio.h>
S-3     #include <omp.h>
S-4
S-5     void socket_init(int socket_num)
S-6     {
S-7        int n_procs;
S-8
S-9        n_procs = omp_get_place_num_procs(socket_num);
S-10       #pragma omp parallel num_threads(n_procs) proc_bind(close)
S-11       {
S-12          printf("Reporting in from socket num, thread num:  %d %d\n",
S-13                                      socket_num,omp_get_thread_num() );
```

```
S-14        }
S-15     }
S-16
S-17     int main()
S-18     {
S-19        int n_sockets, socket_num;
S-20
S-21        omp_set_nested(1);              // or export OMP_NESTED=true
S-22        omp_set_max_active_levels(2);   // or export OMP_MAX_ACTIVE_LEVELS=2
S-23
S-24        n_sockets = omp_get_num_places();
S-25        #pragma omp parallel num_threads(n_sockets) private(socket_num) \
S-26                             proc_bind(spread)
S-27        {
S-28           socket_num = omp_get_place_num();
S-29           socket_init(socket_num);
S-30        }
S-31     }
```

C / C++

CHAPTER 3

Tasking

Tasking constructs provide units of work to a thread for execution. Worksharing constructs do this, too (e.g. **for**, **do**, **sections**, and **singles** constructs); but the work units are tightly controlled by an iteration limit and limited scheduling, or a limited number of **sections** or **single** regions. Worksharing was designed with "data parallel" computing in mind. Tasking was designed for "task parallel" computing and often involves non-locality or irregularity in memory access.

The **task** construct can be used to execute work chunks: in a while loop; while traversing nodes in a list; at nodes in a tree graph; or in a normal loop (with a **taskloop** construct). Unlike the statically scheduled loop iterations of worksharing, a task is often enqueued, and then dequeued for execution by any of the threads of the team within a parallel region. The generation of tasks can be from a single generating thread (creating sibling tasks), or from multiple generators in a recursive graph tree traversals. A **taskloop** construct bundles iterations of an associated loop into tasks, and provides similar controls found in the **task** construct.

Sibling tasks are synchronized by the **taskwait** construct, and tasks and their descendent tasks can be synchronized by containing them in a **taskgroup** region. Ordered execution is accomplished by specifying dependences with a **depend** clause. Also, priorities can be specified as hints to the scheduler through a **priority** clause.

Various clauses can be used to manage and optimize task generation, as well as reduce the overhead of execution and to relinquish control of threads for work balance and forward progress.

Once a thread starts executing a task, it is the designated thread for executing the task to completion, even though it may leave the execution at a scheduling point and return later. The thread is tied to the task. Scheduling points can be introduced with the **taskyield** construct. With an **untied** clause any other thread is allowed to continue the task. An **if** clause with a *true* expression allows the generating thread to immediately execute the task as an undeferred task. By including the data environment of the generating task into the generated task with the **mergeable** and **final** clauses, task generation overhead can be reduced.

A complete list of the tasking constructs and details of their clauses can be found in the *Tasking Constructs* chapter of the OpenMP Specifications, in the *OpenMP Application Programming Interface* section.

3.1 The `task` and `taskwait` Constructs

The following example shows how to traverse a tree-like structure using explicit tasks. Note that the **`traverse`** function should be called from within a parallel region for the different specified tasks to be executed in parallel. Also note that the tasks will be executed in no specified order because there are no synchronization directives. Thus, assuming that the traversal will be done in post order, as in the sequential code, is wrong.

C / C++

Example tasking.1.c

```
S-1   struct node {
S-2     struct node *left;
S-3     struct node *right;
S-4   };
S-5   extern void process(struct node *);
S-6   void traverse( struct node *p ) {
S-7     if (p->left)
S-8   #pragma omp task   // p is firstprivate by default
S-9         traverse(p->left);
S-10    if (p->right)
S-11  #pragma omp task    // p is firstprivate by default
S-12        traverse(p->right);
S-13    process(p);
S-14  }
```

C / C++

In the next example, we force a postorder traversal of the tree by adding a **`taskwait`** directive. Now, we can safely assume that the left and right sons have been executed before we process the current node.

C / C++

Example tasking.2.c

```
S-1   struct node {
S-2     struct node *left;
S-3     struct node *right;
S-4   };
S-5   extern void process(struct node *);
S-6   void postorder_traverse( struct node *p ) {
S-7       if (p->left)
S-8          #pragma omp task    // p is firstprivate by default
S-9              postorder_traverse(p->left);
S-10      if (p->right)
S-11         #pragma omp task   // p is firstprivate by default
S-12             postorder_traverse(p->right);
S-13      #pragma omp taskwait
```

```
    process(p);
}
```

C / C++

The following example demonstrates how to use the **task** construct to process elements of a linked list in parallel. The thread executing the **single** region generates all of the explicit tasks, which are then executed by the threads in the current team. The pointer *p* is **firstprivate** by default on the **task** construct so it is not necessary to specify it in a **firstprivate** clause.

C / C++

Example tasking.3.c

```
typedef struct node node;
struct node {
      int data;
      node * next;
};

void process(node * p)
{
    /* do work here */
}
void increment_list_items(node * head)
{
    #pragma omp parallel
    {
        #pragma omp single
            {
               node * p = head;
               while (p) {
                    #pragma omp task
                     // p is firstprivate by default
                           process(p);
                     p = p->next;
                  }
            }
     }
}
```

C / C++

The **fib()** function should be called from within a **parallel** region for the different specified tasks to be executed in parallel. Also, only one thread of the **parallel** region should call **fib()** unless multiple concurrent Fibonacci computations are desired.

C / C++

Example tasking.4.c

```
S-1     int fib(int n) {
S-2        int i, j;
S-3        if (n<2)
S-4          return n;
S-5        else {
S-6           #pragma omp task shared(i)
S-7              i=fib(n-1);
S-8           #pragma omp task shared(j)
S-9              j=fib(n-2);
S-10          #pragma omp taskwait
S-11             return i+j;
S-12       }
S-13    }
```

C / C++

Note: There are more efficient algorithms for computing Fibonacci numbers. This classic recursion algorithm is for illustrative purposes.

The following example demonstrates a way to generate a large number of tasks with one thread and execute them with the threads in the team. While generating these tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the **`task`** directive, and start executing unassigned tasks. Once the number of unassigned tasks is sufficiently low, the thread may resume execution of the task generating loop.

C / C++

Example tasking.5.c

```
S-1   #define LARGE_NUMBER 10000000
S-2   double item[LARGE_NUMBER];
S-3   extern void process(double);
S-4
S-5   int main() {
S-6   #pragma omp parallel
S-7     {
S-8       #pragma omp single
S-9       {
S-10        int i;
S-11        for (i=0; i<LARGE_NUMBER; i++)
S-12              #pragma omp task    // i is firstprivate, item is shared
S-13                   process(item[i]);
S-14      }
S-15    }
S-16  }
```

The following example is the same as the previous one, except that the tasks are generated in an untied task. While generating the tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the **task** directive, and start executing unassigned tasks. If that thread begins execution of a task that takes a long time to complete, the other threads may complete all the other tasks before it is finished.

In this case, since the loop is in an untied task, any other thread is eligible to resume the task generating loop. In the previous examples, the other threads would be forced to idle until the generating thread finishes its long task, since the task generating loop was in a tied task.

C / C++

Example tasking.6.c

```
S-1   #define LARGE_NUMBER 10000000
S-2   double item[LARGE_NUMBER];
S-3   extern void process(double);
S-4   int main() {
S-5   #pragma omp parallel
S-6     {
S-7       #pragma omp single
S-8       {
S-9         int i;
S-10        #pragma omp task untied
S-11        // i is firstprivate, item is shared
S-12        {
S-13           for (i=0; i<LARGE_NUMBER; i++)
S-14               #pragma omp task
S-15                    process(item[i]);
S-16        }
S-17      }
S-18    }
S-19    return 0;
S-20  }
```

C / C++

The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of the OpenMP 4.0 specification affect the usage of **threadprivate** variables in tasks. A **threadprivate** variable can be modified by another task that is executed by the same thread. Thus, the value of a **threadprivate** variable cannot be assumed to be unchanged across a task scheduling point. In untied tasks, task scheduling points may be added in any place by the implementation.

A task switch may occur at a task scheduling point. A single thread may execute both of the task regions that modify **tp**. The parts of these task regions in which **tp** is modified may be executed in any order so the resulting value of **var** can be either 1 or 2.

C / C++

Example tasking.7.c

```
S-1
S-2     int tp;
S-3     #pragma omp threadprivate(tp)
S-4     int var;
S-5     void work()
S-6     {
S-7     #pragma omp task
S-8         {
S-9             /* do work here */
S-10    #pragma omp task
S-11            {
S-12                tp = 1;
S-13                /* do work here */
S-14    #pragma omp task
S-15                {
S-16                    /* no modification of tp */
S-17                }
S-18                var = tp; //value of tp can be 1 or 2
S-19            }
S-20            tp = 2;
S-21        }
S-22    }
```

C / C++

In this example, scheduling constraints prohibit a thread in the team from executing a new task that modifies **tp** while another such task region tied to the same thread is suspended. Therefore, the value written will persist across the task scheduling point.

C / C++

Example tasking.8.c

```
S-1
S-2     int tp;
S-3     #pragma omp threadprivate(tp)
S-4     int var;
S-5     void work()
S-6     {
S-7     #pragma omp parallel
S-8         {
S-9             /* do work here */
S-10    #pragma omp task
S-11            {
S-12                tp++;
S-13                /* do work here */
```

```
#pragma omp task
            {
                /* do work here but don't modify tp */
            }
            var = tp; //Value does not change after write above
        }
    }
}
```

C / C++

The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of the OpenMP 4.0 specification affect the usage of locks and critical sections in tasks. If a lock is held across a task scheduling point, no attempt should be made to acquire the same lock in any code that may be interleaved. Otherwise, a deadlock is possible.

In the example below, suppose the thread executing task 1 defers task 2. When it encounters the task scheduling point at task 3, it could suspend task 1 and begin task 2 which will result in a deadlock when it tries to enter critical region 1.

C / C++

Example tasking.9.c

```
void work()
{
   #pragma omp task
   { //Task 1
       #pragma omp task
       { //Task 2
            #pragma omp critical //Critical region 1
            {/*do work here */ }
       }
       #pragma omp critical //Critical Region 2
       {
           //Capture data for the following task
           #pragma omp task
           { /* do work here */ } //Task 3
       }
   }
}
```

C / C++

In the following example, **lock** is held across a task scheduling point. However, according to the scheduling restrictions, the executing thread can't begin executing one of the non-descendant tasks that also acquires **lock** before the task region is complete. Therefore, no deadlock is possible.

C / C++

Example tasking.10.c

```
S-1   #include <omp.h>
S-2   void work() {
S-3       omp_lock_t lock;
S-4       omp_init_lock(&lock);
S-5   #pragma omp parallel
S-6       {
S-7           int i;
S-8   #pragma omp for
S-9           for (i = 0; i < 100; i++) {
S-10  #pragma omp task
S-11              {
S-12          // lock is shared by default in the task
S-13          omp_set_lock(&lock);
S-14                  // Capture data for the following task
S-15  #pragma omp task
S-16          // Task Scheduling Point 1
S-17                  { /* do work here */ }
S-18                  omp_unset_lock(&lock);
S-19              }
S-20          }
S-21      }
S-22      omp_destroy_lock(&lock);
S-23  }
```

C / C++

The following examples illustrate the use of the **mergeable** clause in the **task** construct. In this first example, the **task** construct has been annotated with the **mergeable** clause. The addition of this clause allows the implementation to reuse the data environment (including the ICVs) of the parent task for the task inside **foo** if the task is included or undeferred. Thus, the result of the execution may differ depending on whether the task is merged or not. Therefore the mergeable clause needs to be used with caution. In this example, the use of the mergeable clause is safe. As **x** is a shared variable the outcome does not depend on whether or not the task is merged (that is, the task will always increment the same variable and will always compute the same value for **x**).

C / C++

Example tasking.11.c

```
S-1   #include <stdio.h>
S-2   void foo ( )
S-3   {
S-4      int x = 2;
S-5      #pragma omp task shared(x) mergeable
S-6      {
S-7         x++;
S-8      }
S-9      #pragma omp taskwait
S-10     printf("%d\n",x);  // prints 3
S-11  }
```

C / C++

This second example shows an incorrect use of the **mergeable** clause. In this example, the created task will access different instances of the variable **x** if the task is not merged, as **x** is **firstprivate**, but it will access the same variable **x** if the task is merged. As a result, the behavior of the program is unspecified and it can print two different values for **x** depending on the decisions taken by the implementation.

C / C++

Example tasking.12.c

```
S-1   #include <stdio.h>
S-2   void foo ( )
S-3   {
S-4      int x = 2;
S-5      #pragma omp task mergeable
S-6      {
S-7         x++;
S-8      }
S-9      #pragma omp taskwait
S-10     printf("%d\n",x);  // prints 2 or 3
S-11  }
```

C / C++

The following example shows the use of the **final** clause and the **omp_in_final** API call in a recursive binary search program. To reduce overhead, once a certain depth of recursion is reached the program uses the **final** clause to create only included tasks, which allow additional optimizations.

The use of the **omp_in_final** API call allows programmers to optimize their code by specifying which parts of the program are not necessary when a task can create only included tasks (that is, the code is inside a **final** task). In this example, the use of a different state variable is not necessary so once the program reaches the part of the computation that is finalized and copying from the parent state to the new state is eliminated. The allocation of **new_state** in the stack could also be avoided but it would make this example less clear. The **final** clause is most effective when used in conjunction with the **mergeable** clause since all tasks created in a **final** task region are included tasks that can be merged if the **mergeable** clause is present.

C / C++

Example tasking.13.c

```
#include <string.h>
#include <omp.h>
#define LIMIT  3 /* arbitrary limit on recursion depth */
void check_solution(char *);
void bin_search (int pos, int n, char *state)
{
   if ( pos == n ) {
      check_solution(state);
      return;
   }
   #pragma omp task final( pos > LIMIT ) mergeable
   {
      char new_state[n];
      if (!omp_in_final() ) {
        memcpy(new_state, state, pos );
        state = new_state;
      }
      state[pos] = 0;
      bin_search(pos+1, n, state );
   }
   #pragma omp task final( pos > LIMIT ) mergeable
   {
      char new_state[n];
      if (! omp_in_final() ) {
        memcpy(new_state, state, pos );
        state = new_state;
      }
      state[pos] = 1;
```

```
S-29          bin_search(pos+1, n, state );
S-30       }
S-31       #pragma omp taskwait
S-32    }
```

C / C++

The following example illustrates the difference between the **if** and the **final** clauses. The **if** clause has a local effect. In the first nest of tasks, the one that has the **if** clause will be undeferred but the task nested inside that task will not be affected by the **if** clause and will be created as usual. Alternatively, the **final** clause affects all **task** constructs in the **final** task region but not the **final** task itself. In the second nest of tasks, the nested tasks will be created as included tasks. Note also that the conditions for the **if** and **final** clauses are usually the opposite.

C / C++

Example tasking.14.c

```
S-1     void bar(void);
S-2
S-3     void foo ( )
S-4     {
S-5        int i;
S-6        #pragma omp task if(0)  // This task is undeferred
S-7        {
S-8           #pragma omp task     // This task is a regular task
S-9           for (i = 0; i < 3; i++) {
S-10              #pragma omp task     // This task is a regular task
S-11              bar();
S-12          }
S-13       }
S-14       #pragma omp task final(1) // This task is a regular task
S-15       {
S-16          #pragma omp task  // This task is included
S-17          for (i = 0; i < 3; i++) {
S-18              #pragma omp task     // This task is also included
S-19              bar();
S-20          }
S-21       }
S-22    }
```

C / C++

3.2 Task Priority

In this example we compute arrays in a matrix through a *compute_array* routine. Each task has a priority value equal to the value of the loop variable *i* at the moment of its creation. A higher priority on a task means that a task is a candidate to run sooner.

The creation of tasks occurs in ascending order (according to the iteration space of the loop) but a hint, by means of the **priority** clause, is provided to reverse the execution order.

C / C++

Example task_priority.1.c

```
S-1   void compute_array (float *node, int M);
S-2
S-3   void compute_matrix (float *array, int N, int M)
S-4   {
S-5      int i;
S-6      #pragma omp parallel private(i)
S-7      #pragma omp single
S-8      {
S-9         for (i=0;i<N; i++) {
S-10           #pragma omp task priority(i)
S-11           compute_array(&array[i*M], M);
S-12        }
S-13     }
S-14  }
```

C / C++

3.3 Task Dependences

3.3.1 Flow Dependence

In this example we show a simple flow dependence expressed using the **depend** clause on the **task** construct.

C / C++

Example task_dep.1.c

```
S-1   #include <stdio.h>
S-2   int main()
S-3   {
S-4      int x = 1;
S-5      #pragma omp parallel
S-6      #pragma omp single
S-7      {
S-8         #pragma omp task shared(x) depend(out: x)
S-9            x = 2;
S-10        #pragma omp task shared(x) depend(in: x)
S-11           printf("x = %d\n", x);
S-12     }
S-13     return 0;
S-14  }
```

C / C++

The program will always print `"x = 2"`, because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program and the program would have a race condition.

3.3.2 Anti-dependence

In this example we show an anti-dependence expressed using the **depend** clause on the **task** construct.

C / C++

Example task_dep.2.c

```
S-1   #include <stdio.h>
S-2   int main()
S-3   {
S-4      int x = 1;
S-5      #pragma omp parallel
S-6      #pragma omp single
S-7      {
S-8         #pragma omp task shared(x) depend(in: x)
S-9            printf("x = %d\n", x);
S-10        #pragma omp task shared(x) depend(out: x)
S-11           x = 2;
S-12     }
S-13     return 0;
S-14  }
```

C / C++

The program will always print `"x = 1"`, because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

3.3.3 Output Dependence

In this example we show an output dependence expressed using the **depend** clause on the **task** construct.

C / C++

Example task_dep.3.c

```
S-1   #include <stdio.h>
S-2   int main()
S-3   {
S-4      int x;
S-5      #pragma omp parallel
S-6      #pragma omp single
S-7      {
S-8         #pragma omp task shared(x) depend(out: x)
S-9            x = 1;
S-10        #pragma omp task shared(x) depend(out: x)
S-11           x = 2;
S-12        #pragma omp taskwait
S-13        printf("x = %d\n", x);
```

```
S-14        }
S-15        return 0;
S-16     }
```

C / C++

The program will always print "x = 2", because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

3.3.4 Concurrent Execution with Dependences

In this example we show potentially concurrent execution of tasks using multiple flow dependences expressed using the **depend** clause on the **task** construct.

C / C++

Example task_dep.4.c

```
S-1      #include <stdio.h>
S-2      int main()
S-3      {
S-4         int x = 1;
S-5         #pragma omp parallel
S-6         #pragma omp single
S-7         {
S-8            #pragma omp task shared(x) depend(out: x)
S-9               x = 2;
S-10           #pragma omp task shared(x) depend(in: x)
S-11              printf("x + 1 = %d. ", x+1);
S-12           #pragma omp task shared(x) depend(in: x)
S-13              printf("x + 2 = %d\n", x+2);
S-14        }
S-15        return 0;
S-16     }
```

C / C++

The last two tasks are dependent on the first task. However there is no dependence between the last two tasks, which may execute in any order (or concurrently if more than one thread is available). Thus, the possible outputs are "x + 1 = 3. x + 2 = 4. " and "x + 2 = 4. x + 1 = 3. ". If the **depend** clauses had been omitted, then all of the tasks could execute in any order and the program would have a race condition.

3.3.5 Matrix multiplication

This example shows a task-based blocked matrix multiplication. Matrices are of NxN elements, and the multiplication is implemented using blocks of BSxBS elements.

C / C++

Example task_dep.5.c

```
S-1   // Assume BS divides N perfectly
S-2   void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float
S-3   C[N][N] )
S-4   {
S-5      int i, j, k, ii, jj, kk;
S-6      for (i = 0; i < N; i+=BS) {
S-7         for (j = 0; j < N; j+=BS) {
S-8            for (k = 0; k < N; k+=BS) {
S-9   // Note 1: i, j, k, A, B, C are firstprivate by default
S-10  // Note 2: A, B and C are just pointers
S-11  #pragma omp task private(ii, jj, kk) \
S-12              depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
S-13              depend ( inout: C[i:BS][j:BS] )
S-14              for (ii = i; ii < i+BS; ii++ )
S-15                 for (jj = j; jj < j+BS; jj++ )
S-16                    for (kk = k; kk < k+BS; kk++ )
S-17                       C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
S-18           }
S-19        }
S-20     }
S-21  }
```

C / C++

3.4 The `taskgroup` Construct

In this example, tasks are grouped and synchronized using the **`taskgroup`** construct.

Initially, one task (the task executing the **`start_background_work()`** call) is created in the **`parallel`** region, and later a parallel tree traversal is started (the task executing the root of the recursive **`compute_tree()`** calls). While synchronizing tasks at the end of each tree traversal, using the **`taskgroup`** construct ensures that the formerly started background task does not participate in the synchronization, and is left free to execute in parallel. This is opposed to the behaviour of the **`taskwait`** construct, which would include the background tasks in the synchronization.

C / C++

Example taskgroup.1.c

```
S-1   extern void start_background_work(void);
S-2   extern void check_step(void);
S-3   extern void print_results(void);
S-4   struct tree_node
S-5   {
S-6      struct tree_node *left;
S-7      struct tree_node *right;
S-8   };
S-9   typedef struct tree_node* tree_type;
S-10  extern void init_tree(tree_type);
S-11  #define max_steps 100
S-12  void compute_something(tree_type tree)
S-13  {
S-14     // some computation
S-15  }
S-16  void compute_tree(tree_type tree)
S-17  {
S-18     if (tree->left)
S-19     {
S-20       #pragma omp task
S-21         compute_tree(tree->left);
S-22     }
S-23     if (tree->right)
S-24     {
S-25       #pragma omp task
S-26         compute_tree(tree->right);
S-27     }
S-28     #pragma omp task
S-29     compute_something(tree);
S-30  }
S-31  int main()
S-32  {
```

```
S-33      int i;
S-34      tree_type tree;
S-35      init_tree(tree);
S-36      #pragma omp parallel
S-37      #pragma omp single
S-38      {
S-39        #pragma omp task
S-40          start_background_work();
S-41        for (i = 0; i < max_steps; i++)
S-42        {
S-43            #pragma omp taskgroup
S-44            {
S-45               #pragma omp task
S-46                 compute_tree(tree);
S-47            } // wait on tree traversal in this step
S-48            check_step();
S-49        }
S-50      } // only now is background work required to be complete
S-51      print_results();
S-52      return 0;
S-53    }
```

C / C++

3.5 The `taskyield` Construct

The following example illustrates the use of the **`taskyield`** directive. The tasks in the example compute something useful and then do some computation that must be done in a critical region. By using **`taskyield`** when a task cannot get access to the **`critical`** region the implementation can suspend the current task and schedule some other task that can do something useful.

C / C++

Example taskyield.1.c

```
S-1   #include <omp.h>
S-2
S-3   void something_useful ( void );
S-4   void something_critical ( void );
S-5   void foo ( omp_lock_t * lock, int n )
S-6   {
S-7      int i;
S-8
S-9      for ( i = 0; i < n; i++ )
S-10        #pragma omp task
S-11        {
S-12            something_useful();
S-13            while ( !omp_test_lock(lock) ) {
S-14               #pragma omp taskyield
S-15            }
S-16            something_critical();
S-17            omp_unset_lock(lock);
S-18        }
S-19  }
```

C / C++

3.6 The `taskloop` Construct

The following example illustrates how to execute a long running task concurrently with tasks created with a **taskloop** directive for a loop having unbalanced amounts of work for its iterations.

The **grainsize** clause specifies that each task is to execute at least 500 iterations of the loop.

The **nogroup** clause removes the implicit taskgroup of the **taskloop** construct; the explicit **taskgroup** construct in the example ensures that the function is not exited before the long-running task and the loops have finished execution.

C / C++

Example taskloop.1.c

```
S-1   void long_running_task(void);
S-2   void loop_body(int i, int j);
S-3
S-4   void parallel_work(void) {
S-5      int i, j;
S-6   #pragma omp taskgroup
S-7      {
S-8   #pragma omp task
S-9         long_running_task(); // can execute concurrently
S-10
S-11  #pragma omp taskloop private(j) grainsize(500) nogroup
S-12        for (i = 0; i < 10000; i++) { // can execute concurrently
S-13           for (j = 0; j < i; j++) {
S-14              loop_body(i, j);
S-15           }
S-16        }
S-17     }
S-18  }
```

C / C++

CHAPTER 4

Devices

The **target** construct consists of a **target** directive and an execution region. The **target** region is executed on the default device or the device specified in the **device** clause.

In OpenMP version 4.0, by default, all variables within the lexical scope of the construct are copied *to* and *from* the device, unless the device is the host, or the data exists on the device from a previously executed data-type construct that has created space on the device and possibly copied host data to the device storage.

The constructs that explicitly create storage, transfer data, and free storage on the device are catagorized as structured and unstructured. The **target data** construct is structured. It creates a data region around **target** constructs, and is convenient for providing persistent data throughout multiple **target** regions. The **target enter data** and **target exit data** constructs are unstructured, because they can occur anywhere and do not support a "structure" (a region) for enclosing **target** constructs, as does the **target data** construct.

The **map** clause is used on **target** constructs and the data-type constructs to map host data. It specifies the device storage and data movement **to** and **from** the device, and controls on the storage duration.

There is an important change in the OpenMP 4.5 specification that alters the data model for scalar variables and C/C++ pointer variables. The default behavior for scalar variables and C/C++ pointer variables in an 4.5 compliant code is **firstprivate**. Example codes that have been updated to reflect this new behavior are annotated with a description that describes changes required for correct execution. Often it is a simple matter of mapping the variable as **tofrom** to obtain the intended 4.0 behavior.

In OpenMP version 4.5 the mechanism for target execution is specified as occuring through a *target task*. When the **target** construct is encountered a new *target task* is generated. The *target task* completes after the **target** region has executed and all data transfers have finished.

This new specification does not affect the execution of pre-4.5 code; it is a necessary element for asynchronous execution of the **target** region when using the new **nowait** clause introduced in OpenMP 4.5.

4.1 `target` Construct

4.1.1 `target` Construct on `parallel` Construct

This following example shows how the **`target`** construct offloads a code region to a target device. The variables *p*, *v1*, *v2*, and *N* are implicitly mapped to the target device.

C / C++

Example target.1.c

```
S-1   extern void init(float*, float*, int);
S-2   extern void output(float*, int);
S-3   void vec_mult(int N)
S-4   {
S-5      int i;
S-6      float p[N], v1[N], v2[N];
S-7      init(v1, v2, N);
S-8      #pragma omp target
S-9      #pragma omp parallel for private(i)
S-10     for (i=0; i<N; i++)
S-11       p[i] = v1[i] * v2[i];
S-12     output(p, N);
S-13  }
```

C / C++

4.1.2 `target` Construct with `map` Clause

This following example shows how the **`target`** construct offloads a code region to a target device. The variables *p*, *v1* and *v2* are explicitly mapped to the target device using the **`map`** clause. The variable *N* is implicitly mapped to the target device.

C / C++

Example target.2.c

```
S-1   extern void init(float*, float*, int);
S-2   extern void output(float*, int);
S-3   void vec_mult(int N)
S-4   {
S-5      int i;
S-6      float p[N], v1[N], v2[N];
S-7      init(v1, v2, N);
S-8      #pragma omp target map(v1, v2, p)
S-9      #pragma omp parallel for
S-10     for (i=0; i<N; i++)
S-11       p[i] = v1[i] * v2[i];
S-12     output(p, N);
S-13  }
```

C / C++

4.1.3 `map` Clause with `to`/`from` map-types

The following example shows how the **`target`** construct offloads a code region to a target device. In the **`map`** clause, the **`to`** and **`from`** map-types define the mapping between the original (host) data and the target (device) data. The **`to`** map-type specifies that the data will only be read on the device, and the **`from`** map-type specifies that the data will only be written to on the device. By specifying a guaranteed access on the device, data transfers can be reduced for the **`target`** region.

The **`to`** map-type indicates that at the start of the **`target`** region the variables *v1* and *v2* are initialized with the values of the corresponding variables on the host device, and at the end of the **`target`** region the variables *v1* and *v2* are not assigned to their corresponding variables on the host device.

The **`from`** map-type indicates that at the start of the **`target`** region the variable *p* is not initialized with the value of the corresponding variable on the host device, and at the end of the **`target`** region the variable *p* is assigned to the corresponding variable on the host device.

C / C++

Example target.3.c

```
S-1   extern void init(float*, float*, int);
S-2   extern void output(float*, int);
S-3   void vec_mult(int N)
S-4   {
S-5      int i;
S-6      float p[N], v1[N], v2[N];
S-7      init(v1, v2, N);
S-8      #pragma omp target map(to: v1, v2) map(from: p)
S-9      #pragma omp parallel for
S-10     for (i=0; i<N; i++)
S-11       p[i] = v1[i] * v2[i];
S-12     output(p, N);
S-13  }
```

C / C++

The **to** and **from** map-types allow programmers to optimize data motion. Since data for the *v* arrays are not returned, and data for the *p* array are not transferred to the device, only one-half of the data is moved, compared to the default behavior of an implicit mapping.

4.1.4 `map` Clause with Array Sections

The following example shows how the **target** construct offloads a code region to a target device. In the **map** clause, map-types are used to optimize the mapping of variables to the target device. Because variables *p*, *v1* and *v2* are pointers, array section notation must be used to map the arrays. The notation **:N** is equivalent to **0:N**.

C / C++

Example target.4.c

```
S-1   extern void init(float*, float*, int);
S-2   extern void output(float*, int);
S-3   void vec_mult(float *p, float *v1, float *v2, int N)
S-4   {
S-5      int i;
S-6      init(v1, v2, N);
S-7      #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8      #pragma omp parallel for
S-9      for (i=0; i<N; i++)
S-10       p[i] = v1[i] * v2[i];
S-11     output(p, N);
S-12  }
```

C / C++

In C, the length of the pointed-to array must be specified. In Fortran the extent of the array is known and the length need not be specified. A section of the array can be specified with the usual Fortran syntax, as shown in the following example. The value 1 is assumed for the lower bound for array section *v2(:N)*.

A more realistic situation in which an assumed-size array is passed to **vec_mult** requires that the length of the arrays be specified, because the compiler does not know the size of the storage. A section of the array must be specified with the usual Fortran syntax, as shown in the following example. The value 1 is assumed for the lower bound for array section *v2(:N)*.

4.1.5 `target` Construct with `if` Clause

The following example shows how the **target** construct offloads a code region to a target device.

The **if** clause on the **target** construct indicates that if the variable *N* is smaller than a given threshold, then the **target** region will be executed by the host device.

The **if** clause on the **parallel** construct indicates that if the variable *N* is smaller than a second threshold then the **parallel** region is inactive.

C / C++

Example target.5.c

```
S-1   #define THRESHOLD1 1000000
S-2   #define THRESHOLD2 1000
S-3   extern void init(float*, float*, int);
S-4   extern void output(float*, int);
S-5   void vec_mult(float *p, float *v1, float *v2, int N)
S-6   {
S-7      int i;
S-8      init(v1, v2, N);
S-9      #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])\
S-10          map(from: p[0:N])
S-11     #pragma omp parallel for if(N>THRESHOLD2)
S-12     for (i=0; i<N; i++)
S-13       p[i] = v1[i] * v2[i];
S-14     output(p, N);
S-15  }
```

C / C++

The following example is a modification of the above *target.5* code to show the combined **target** and parallel loop directives. It uses the *directive-name* modifier in multiple **if** clauses to specify the component directive to which it applies.

The **if** clause with the **target** modifier applies to the **target** component of the combined directive, and the **if** clause with the **parallel** modifier applies to the **parallel** component of the combined directive.

C / C++

Example target.6.c

```
S-1   #define THRESHOLD1 1000000
S-2   #define THRESHOLD2 1000
S-3   extern void init(float*, float*, int);
S-4   extern void output(float*, int);
S-5   void vec_mult(float *p, float *v1, float *v2, int N)
S-6   {
S-7      int i;
S-8      init(v1, v2, N);
S-9      #pragma omp target parallel for \
S-10          if(target: N>THRESHOLD1) if(parallel: N>THRESHOLD2) \
S-11          map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-12     for (i=0; i<N; i++)
S-13       p[i] = v1[i] * v2[i];
S-14     output(p, N);
S-15  }
```

C / C++

4.2 `target data` Construct

4.2.1 Simple `target data` Construct

This example shows how the **`target data`** construct maps variables to a device data environment. The **`target data`** construct creates a new device data environment and maps the variables *v1*, *v2*, and *p* to the new device data environment. The **`target`** construct enclosed in the **`target data`** region creates a new device data environment, which inherits the variables *v1*, *v2*, and *p* from the enclosing device data environment. The variable *N* is mapped into the new device data environment from the encountering task's data environment.

C / C++

Example target_data.1.c

```
S-1   extern void init(float*, float*, int);
S-2   extern void output(float*, int);
S-3   void vec_mult(float *p, float *v1, float *v2, int N)
S-4   {
S-5      int i;
S-6      init(v1, v2, N);
S-7      #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8      {
S-9         #pragma omp target
S-10        #pragma omp parallel for
S-11        for (i=0; i<N; i++)
S-12          p[i] = v1[i] * v2[i];
S-13     }
S-14     output(p, N);
S-15  }
```

C / C++

The Fortran code passes a reference and specifies the extent of the arrays in the declaration. No length information is necessary in the map clause, as is required with C/C++ pointers.

4.2.2 `target data` Region Enclosing Multiple `target` Regions

The following examples show how the **`target data`** construct maps variables to a device data environment of a **`target`** region. The **`target data`** construct creates a device data environment and encloses **`target`** regions, which have their own device data environments. The device data environment of the **`target data`** region is inherited by the device data environment of an enclosed **`target`** region. The **`target data`** construct is used to create variables that will persist throughout the **`target data`** region.

In the following example the variables *v1* and *v2* are mapped at each **`target`** construct. Instead of mapping the variable *p* twice, once at each **`target`** construct, *p* is mapped once by the **`target data`** construct.

C / C++

Example target_data.2.c

```
S-1   extern void init(float*, float*, int);
S-2   extern void init_again(float*, float*, int);
S-3   extern void output(float*, int);
S-4   void vec_mult(float *p, float *v1, float *v2, int N)
S-5   {
S-6      int i;
S-7      init(v1, v2, N);
S-8      #pragma omp target data map(from: p[0:N])
S-9      {
S-10        #pragma omp target map(to: v1[:N], v2[:N])
S-11        #pragma omp parallel for
S-12        for (i=0; i<N; i++)
S-13          p[i] = v1[i] * v2[i];
S-14        init_again(v1, v2, N);
S-15        #pragma omp target map(to: v1[:N], v2[:N])
S-16        #pragma omp parallel for
S-17        for (i=0; i<N; i++)
S-18          p[i] = p[i] + (v1[i] * v2[i]);
S-19     }
S-20     output(p, N);
S-21  }
```

C / C++

The Fortran code uses reference and specifies the extent of the *p*, *v1* and *v2* arrays. No length information is necessary in the **`map`** clause, as is required with C/C++ pointers. The arrays *v1* and *v2* are mapped at each **`target`** construct. Instead of mapping the array *p* twice, once at each target construct, *p* is mapped once by the **`target data`** construct.

In the following example, the variable tmp defaults to **tofrom** map-type and is mapped at each **target** construct. The array *Q* is mapped once at the enclosing **target data** region instead of at each **target** construct.

C / C++

Example target_data.3.c

```
S-1   #include <math.h>
S-2   #define COLS 100
S-3   void gramSchmidt(float Q[][COLS], const int rows)
S-4   {
S-5       int cols = COLS;
S-6       #pragma omp target data map(Q[0:rows][0:cols])
S-7       for(int k=0; k < cols; k++)
S-8       {
S-9           double tmp = 0.0;
S-10          #pragma omp target map(tofrom: tmp)
S-11          #pragma omp parallel for reduction(+:tmp)
S-12          for(int i=0; i < rows; i++)
S-13              tmp += (Q[i][k] * Q[i][k]);
S-14
S-15          tmp = 1/sqrt(tmp);
S-16
S-17          #pragma omp target
S-18          #pragma omp parallel for
S-19          for(int i=0; i < rows; i++)
S-20              Q[i][k] *= tmp;
S-21      }
S-22  }
S-23
S-24  /* Note:  The variable tmp is now mapped with tofrom, for correct
S-25     execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-26   */
```

C / C++

In the following example the arrays *v1* and *v2* are mapped at each **target** construct. Instead of mapping the array *Q* twice at each **target** construct, *Q* is mapped once by the **target data** construct. Note, the *tmp* variable is implicitly remapped for each **target** region, mapping the value from the device to the host at the end of the first **target** region, and from the host to the device for the second **target** region.

4.2.3 `target data` Construct with Orphaned Call

The following two examples show how the **`target data`** construct maps variables to a device data environment. The **`target data`** construct's device data environment encloses the **`target`** construct's device data environment in the function **`vec_mult()`**.

When the type of the variable appearing in an array section is pointer, the pointer variable and the storage location of the corresponding array section are mapped to the device data environment. The pointer variable is treated as if it had appeared in a **`map`** clause with a map-type of **`alloc`**. The array section's storage location is mapped according to the map-type in the **`map`** clause (the default map-type is **`tofrom`**).

The **`target`** construct's device data environment inherits the storage locations of the array sections *v1[0:N]*, *v2[:n]*, and *p0[0:N]* from the enclosing target data construct's device data environment. Neither initialization nor assignment is performed for the array sections in the new device data environment.

The pointer variables *p1*, *v3*, and *v4* are mapped into the target construct's device data environment with an implicit map-type of alloc and they are assigned the address of the storage location associated with their corresponding array sections. Note that the following pairs of array section storage locations are equivalent (*p0[:N]*, *p1[:N]*), (*v1[:N]*,*v3[:N]*), and (*v2[:N]*,*v4[:N]*).

C / C++

Example target_data.4.c

```
S-1   void vec_mult(float*, float*, float*, int);
S-2   extern void init(float*, float*, int);
S-3   extern void output(float*, int);
S-4   void foo(float *p0, float *v1, float *v2, int N)
S-5   {
S-6      init(v1, v2, N);
S-7      #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p0[0:N])
S-8      {
S-9         vec_mult(p0, v1, v2, N);
S-10     }
S-11     output(p0, N);
S-12  }
S-13  void vec_mult(float *p1, float *v3, float *v4, int N)
S-14  {
S-15     int i;
S-16     #pragma omp target map(to: v3[0:N], v4[:N]) map(from: p1[0:N])
S-17     #pragma omp parallel for
S-18     for (i=0; i<N; i++)
S-19       p1[i] = v3[i] * v4[i];
S-20  }
```

C / C++

The Fortran code maps the pointers and storage in an identical manner (same extent, but uses indices from 1 to *N*).

The **target** construct's device data environment inherits the storage locations of the arrays *v1*, *v2* and *p0* from the enclosing **target data** constructs's device data environment. However, in Fortran the associated data of the pointer is known, and the shape is not required.

The pointer variables *p1*, *v3*, and *v4* are mapped into the **target** construct's device data environment with an implicit map-type of **alloc** and they are assigned the address of the storage location associated with their corresponding array sections. Note that the following pair of array storage locations are equivalent (*p0*,*p1*), (*v1*,*v3*), and (*v2*,*v4*).

In the following example, the variables *p1*, *v3*, and *v4* are references to the pointer variables *p0*, *v1* and *v2* respectively. The **target** construct's device data environment inherits the pointer variables *p0*, *v1*, and *v2* from the enclosing **target data** construct's device data environment. Thus, *p1*, *v3*, and *v4* are already present in the device data environment.

C++

Example target_data.5.cpp

```
S-1   void vec_mult(float* &, float* &, float* &, int &);
S-2   extern void init(float*, float*, int);
S-3   extern void output(float*, int);
S-4   void foo(float *p0, float *v1, float *v2, int N)
S-5   {
S-6      init(v1, v2, N);
S-7      #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p0[0:N])
S-8      {
S-9         vec_mult(p0, v1, v2, N);
S-10     }
S-11     output(p0, N);
S-12  }
S-13  void vec_mult(float* &p1, float* &v3, float* &v4, int &N)
S-14  {
S-15     int i;
S-16     #pragma omp target map(to: v3[0:N], v4[:N]) map(from: p1[0:N])
S-17     #pragma omp parallel for
S-18     for (i=0; i<N; i++)
S-19       p1[i] = v3[i] * v4[i];
S-20  }
```

C++

In the following example, the usual Fortran approach is used for dynamic memory. The *p0*, *v1*, and *v2* arrays are allocated in the main program and passed as references from one routine to another. In **vec_mult**, *p1*, *v3* and *v4* are references to the *p0*, *v1*, and *v2* arrays, respectively. The **target**

construct's device data environment inherits the arrays *p0*, *v1*, and *v2* from the enclosing target data construct's device data environment. Thus, *p1*, *v3*, and *v4* are already present in the device data environment.

4.2.4 `target data` Construct with `if` Clause

The following two examples show how the **`target data`** construct maps variables to a device data environment.

In the following example, the if clause on the **`target data`** construct indicates that if the variable *N* is smaller than a given threshold, then the **`target data`** construct will not create a device data environment.

The **`target`** constructs enclosed in the **`target data`** region must also use an **`if`** clause on the same condition, otherwise the pointer variable *p* is implicitly mapped with a map-type of **`tofrom`**, but the storage location for the array section *p[0:N]* will not be mapped in the device data environments of the **`target`** constructs.

C / C++

Example target_data.6.c

```
S-1   #define THRESHOLD 1000000
S-2   extern void init(float*, float*, int);
S-3   extern void init_again(float*, float*, int);
S-4   extern void output(float*, int);
S-5   void vec_mult(float *p, float *v1, float *v2, int N)
S-6   {
S-7      int i;
S-8      init(v1, v2, N);
S-9      #pragma omp target data if(N>THRESHOLD) map(from: p[0:N])
S-10     {
S-11        #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-12        #pragma omp parallel for
S-13        for (i=0; i<N; i++)
S-14          p[i] = v1[i] * v2[i];
S-15        init_again(v1, v2, N);
S-16        #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-17        #pragma omp parallel for
S-18        for (i=0; i<N; i++)
S-19          p[i] = p[i] + (v1[i] * v2[i]);
S-20     }
S-21     output(p, N);
S-22  }
```

C / C++

The **if** clauses work the same way for the following Fortran code. The **target** constructs enclosed in the **target data** region should also use an **if** clause with the same condition, so that the **target data** region and the **target** region are either both created for the device, or are both ignored.

In the following example, when the **if** clause conditional expression on the **target** construct evaluates to *false*, the target region will execute on the host device. However, the **target data** construct created an enclosing device data environment that mapped *p[0:N]* to a device data environment on the default device. At the end of the **target data** region the array section *p[0:N]* will be assigned from the device data environment to the corresponding variable in the data environment of the task that encountered the **target data** construct, resulting in undefined values in *p[0:N]*.

C / C++

Example target_data.7.c

```
S-1   #define THRESHOLD 1000000
S-2   extern void init(float*, float*, int);
S-3   extern void output(float*, int);
S-4   void vec_mult(float *p, float *v1, float *v2, int N)
S-5   {
S-6      int i;
S-7      init(v1, v2, N);
S-8      #pragma omp target data map(from: p[0:N])
S-9      {
S-10        #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-11        #pragma omp parallel for
S-12        for (i=0; i<N; i++)
S-13          p[i] = v1[i] * v2[i];
S-14     } /* UNDEFINED behavior if N<=THRESHOLD */
S-15     output(p, N);
S-16  }
```

C / C++

The **if** clauses work the same way for the following Fortran code. When the **if** clause conditional expression on the **target** construct evaluates to *false*, the **target** region will execute on the host device. However, the **target data** construct created an enclosing device data environment that mapped the *p* array (and *v1* and *v2*) to a device data environment on the default target device. At the end of the **target data** region the *p* array will be assigned from the device data environment to the corresponding variable in the data environment of the task that encountered the **target data** construct, resulting in undefined values in *p*.

4.3 `target enter data` and `target exit data` Constructs

The structured data construct (**`target data`**) provides persistent data on a device for subsequent **`target`** constructs as shown in the **`target data`** examples above. This is accomplished by creating a single **`target data`** region containing **`target`** constructs.

The unstructured data constructs allow the creation and deletion of data on the device at any appropriate point within the host code, as shown below with the **`target enter data`** and **`target exit data`** constructs.

The following C++ code creates/deletes a vector in a constructor/destructor of a class. The constructor creates a vector with **`target enter data`** and uses an **`alloc`** modifier in the **`map`** clause to avoid copying values to the device. The destructor deletes the data (**`target exit data`**) and uses the **`delete`** modifier in the **`map`** clause to avoid copying data back to the host. Note, the stand-alone **`target enter data`** occurs after the host vector is created, and the **`target exit data`** construct occurs before the host data is deleted.

C++

Example target_unstructured_data.1.cpp

```
S-1   class Matrix
S-2   {
S-3
S-4     Matrix(int n) {
S-5       len = n;
S-6       v = new double[len];
S-7       #pragma omp target enter data map(alloc:v[0:len])
S-8     }
S-9
S-10    ~Matrix() {
S-11      // NOTE: delete map type should be used, since the corresponding
S-12      // host data will cease to exist after the deconstructor is called.
S-13
S-14      #pragma omp target exit data map(delete:v[0:len])
S-15      delete[] v;
S-16    }
S-17
S-18    private:
S-19    double* v;
S-20    int len;
S-21
S-22  };
```

C++

The following C code allocates and frees the data member of a Matrix structure. The

init_matrix function allocates the memory used in the structure and uses the **target enter data** directive to map it to the target device. The **free_matrix** function removes the mapped array from the target device and then frees the memory on the host. Note, the stand-alone **target enter data** occurs after the host memory is allocated, and the **target exit data** construct occurs before the host data is freed.

C / C++

Example target_unstructured_data.1.c

```
S-1   #include <stdlib.h>
S-2   typedef struct {
S-3     double *A;
S-4     int N;
S-5   } Matrix;
S-6
S-7   void init_matrix(Matrix *mat, int n)
S-8   {
S-9     mat->A = (double *)malloc(n*sizeof(double));
S-10    mat->N = n;
S-11    #pragma omp target enter data map(alloc:mat->A[:n])
S-12  }
S-13
S-14  void free_matrix(Matrix *mat)
S-15  {
S-16    #pragma omp target exit data map(delete:mat->A[:mat->N])
S-17    mat->N = 0;
S-18    free(mat->A);
S-19    mat->A = NULL;
S-20  }
```

C / C++

The following Fortran code allocates and deallocates a module array. The **initialize** subroutine allocates the module array and uses the **target enter data** directive to map it to the target device. The **finalize** subroutine removes the mapped array from the target device and then deallocates the array on the host. Note, the stand-alone **target enter data** occurs after the host memory is allocated, and the **target exit data** construct occurs before the host data is deallocated.

4.4 `target update` Construct

4.4.1 Simple `target data` and `target update` Constructs

The following example shows how the **`target update`** construct updates variables in a device data environment.

The **`target data`** construct maps array sections *v1[:N]* and *v2[:N]* (arrays *v1* and *v2* in the Fortran code) into a device data environment.

The task executing on the host device encounters the first **`target`** region and waits for the completion of the region.

After the execution of the first **`target`** region, the task executing on the host device then assigns new values to *v1[:N]* and *v2[:N]* (*v1* and *v2* arrays in Fortran code) in the task's data environment by calling the function **`init_again()`**.

The **`target update`** construct assigns the new values of *v1* and *v2* from the task's data environment to the corresponding mapped array sections in the device data environment of the **`target data`** construct.

The task executing on the host device then encounters the second **`target`** region and waits for the completion of the region.

The second **`target`** region uses the updated values of *v1[:N]* and *v2[:N]*.

C / C++

Example target_update.1.c

```
S-1   extern void init(float *, float *, int);
S-2   extern void init_again(float *, float *, int);
S-3   extern void output(float *, int);
S-4   void vec_mult(float *p, float *v1, float *v2, int N)
S-5   {
S-6      int i;
S-7      init(v1, v2, N);
S-8      #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
S-9      {
S-10        #pragma omp target
S-11        #pragma omp parallel for
S-12        for (i=0; i<N; i++)
S-13          p[i] = v1[i] * v2[i];
S-14        init_again(v1, v2, N);
S-15        #pragma omp target update to(v1[:N], v2[:N])
S-16        #pragma omp target
S-17        #pragma omp parallel for
S-18        for (i=0; i<N; i++)
S-19          p[i] = p[i] + (v1[i] * v2[i]);
S-20     }
```

```
S-21      output(p, N);
S-22  }
```

C / C++

4.4.2 `target update` Construct with `if` Clause

The following example shows how the **`target update`** construct updates variables in a device data environment.

The **`target data`** construct maps array sections *v1[:N]* and *v2[:N]* (arrays *v1* and *v2* in the Fortran code) into a device data environment. In between the two **`target`** regions, the task executing on the host device conditionally assigns new values to *v1* and *v2* in the task's data environment. The function **`maybe_init_again()`** returns *true* if new data is written.

When the conditional expression (the return value of **`maybe_init_again()`**) in the **`if`** clause is *true*, the **`target update`** construct assigns the new values of *v1* and *v2* from the task's data environment to the corresponding mapped array sections in the **`target data`** construct's device data environment.

C / C++

Example target_update.2.c

```
S-1   extern void init(float *, float *, int);
S-2   extern int maybe_init_again(float *, int);
S-3   extern void output(float *, int);
S-4   void vec_mult(float *p, float *v1, float *v2, int N)
S-5   {
S-6      int i;
S-7      init(v1, v2, N);
S-8      #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
S-9      {
S-10        int changed;
S-11        #pragma omp target
S-12        #pragma omp parallel for
S-13        for (i=0; i<N; i++)
S-14          p[i] = v1[i] * v2[i];
S-15        changed = maybe_init_again(v1,  N);
S-16        #pragma omp target update if (changed) to(v1[:N])
S-17        changed = maybe_init_again(v2,  N);
S-18        #pragma omp target update if (changed) to(v2[:N])
S-19        #pragma omp target
S-20        #pragma omp parallel for
S-21        for (i=0; i<N; i++)
S-22          p[i] = p[i] + (v1[i] * v2[i]);
S-23     }
S-24     output(p, N);
S-25  }
```

4.5 `declare target` Construct

4.5.1 `declare target` and `end declare target` for a Function

The following example shows how the **`declare target`** directive is used to indicate that the corresponding call inside a **`target`** region is to a **`fib`** function that can execute on the default target device.

A version of the function is also available on the host device. When the **`if`** clause conditional expression on the **`target`** construct evaluates to *false*, the **`target`** region (thus **`fib`**) will execute on the host device.

For C/C++ codes the declaration of the function **`fib`** appears between the **`declare target`** and **`end declare target`** directives.

C / C++

Example declare_target.1.c

```
S-1   #pragma omp declare target
S-2   extern void fib(int N);
S-3   #pragma omp end declare target
S-4   #define THRESHOLD 1000000
S-5   void fib_wrapper(int n)
S-6   {
S-7      #pragma omp target if(n > THRESHOLD)
S-8      {
S-9         fib(n);
S-10     }
S-11  }
```

C / C++

The Fortran **`fib`** subroutine contains a **`declare target`** declaration to indicate to the compiler to create an device executable version of the procedure. The subroutine name has not been included on the **`declare target`** directive and is, therefore, implicitly assumed.

The program uses the **`module_fib`** module, which presents an explicit interface to the compiler with the **`declare target`** declarations for processing the **`fib`** call.

The next Fortran example shows the use of an external subroutine. Without an explicit interface (through module use or an interface block) the **`declare target`** declarations within a external subroutine are unknown to the main program unit; therefore, a **`declare target`** must be provided within the program scope for the compiler to determine that a target binary should be available.

4.5.2 `declare target` Construct for Class Type

C++

The following example shows how the **`declare target`** and **`end declare target`** directives are used to enclose the declaration of a variable *varY* with a class type **`typeY`**. The member function **`typeY::foo()`** cannot be accessed on a target device because its declaration did not appear between **`declare target`** and **`end declare target`** directives.

Example declare_target.2.cpp

```
S-1   struct typeX
S-2   {
S-3      int a;
S-4   };
S-5   class typeY
S-6   {
S-7      int a;
S-8     public:
S-9      int foo() { return a^0x01;}
S-10  };
S-11  #pragma omp declare target
S-12  struct typeX varX;  // ok
S-13  class typeY varY; // ok if varY.foo() not called on target device
S-14  #pragma omp end declare target
S-15  void foo()
S-16  {
S-17     #pragma omp target
S-18     {
S-19        varX.a = 100; // ok
S-20        varY.foo(); // error foo() is not available on a target device
S-21     }
S-22  }
```

C++

4.5.3 `declare target` and `end declare target` for Variables

The following examples show how the **`declare target`** and **`end declare target`** directives are used to indicate that global variables are mapped to the implicit device data environment of each target device.

In the following example, the declarations of the variables *p*, *v1*, and *v2* appear between **`declare target`** and **`end declare target`** directives indicating that the variables are mapped to the implicit device data environment of each target device. The **`target update`** directive is then

used to manage the consistency of the variables *p*, *v1*, and *v2* between the data environment of the encountering host device task and the implicit device data environment of the default target device.

C / C++

Example declare_target.3.c

```
S-1   #define N 1000
S-2   #pragma omp declare target
S-3   float p[N], v1[N], v2[N];
S-4   #pragma omp end declare target
S-5   extern void init(float *, float *, int);
S-6   extern void output(float *, int);
S-7   void vec_mult()
S-8   {
S-9      int i;
S-10     init(v1, v2, N);
S-11     #pragma omp target update to(v1, v2)
S-12     #pragma omp target
S-13     #pragma omp parallel for
S-14     for (i=0; i<N; i++)
S-15       p[i] = v1[i] * v2[i];
S-16     #pragma omp target update from(p)
S-17     output(p, N);
S-18  }
```

C / C++

The Fortran version of the above C code uses a different syntax. Fortran modules use a list syntax on the **declare target** directive to declare mapped variables.

The following example also indicates that the function **Pfun()** is available on the target device, as well as the variable *Q*, which is mapped to the implicit device data environment of each target device. The **target update** directive is then used to manage the consistency of the variable *Q* between the data environment of the encountering host device task and the implicit device data environment of the default target device.

In the following example, the function and variable declarations appear between the **declare target** and **end declare target** directives.

C / C++

Example declare_target.4.c

```
S-1   #define N 10000
S-2   #pragma omp declare target
S-3   float Q[N][N];
S-4   float Pfun(const int i, const int k)
S-5   { return Q[i][k] * Q[k][i]; }
S-6   #pragma omp end declare target
S-7   float accum(int k)
S-8   {
S-9       float tmp = 0.0;
S-10      #pragma omp target update to(Q)
S-11      #pragma omp target map(tofrom: tmp)
S-12      #pragma omp parallel for reduction(+:tmp)
S-13      for(int i=0; i < N; i++)
S-14          tmp += Pfun(i,k);
S-15      return tmp;
S-16  }
S-17
S-18  /* Note:  The variable tmp is now mapped with tofrom, for correct
S-19     execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-20   */
```

C / C++

The Fortran version of the above C code uses a different syntax. In Fortran modules a list syntax on the **declare target** directive is used to declare mapped variables and procedures. The *N* and *Q* variables are declared as a comma separated list. When the **declare target** directive is used to declare just the procedure, the procedure name need not be listed – it is implicitly assumed, as illustrated in the **Pfun()** function.

4.5.4 declare target and end declare target with declare simd

The following example shows how the **declare target** and **end declare target** directives are used to indicate that a function is available on a target device. The **declare simd** directive indicates that there is a SIMD version of the function **P()** that is available on the target device as well as one that is available on the host device.

C / C++

Example declare_target.5.c

```
S-1   #define N 10000
S-2   #define M 1024
S-3   #pragma omp declare target
S-4   float Q[N][N];
S-5   #pragma omp declare simd uniform(i) linear(k) notinbranch
S-6   float P(const int i, const int k)
S-7   {
S-8     return Q[i][k] * Q[k][i];
S-9   }
S-10  #pragma omp end declare target
S-11
S-12  float accum(void)
S-13  {
S-14    float tmp = 0.0;
S-15    int i, k;
S-16  #pragma omp target map(tofrom: tmp)
S-17  #pragma omp parallel for reduction(+:tmp)
S-18    for (i=0; i < N; i++) {
S-19      float tmp1 = 0.0;
S-20  #pragma omp simd reduction(+:tmp1)
S-21      for (k=0; k < M; k++) {
S-22        tmp1 += P(i,k);
S-23      }
S-24      tmp += tmp1;
S-25    }
S-26    return tmp;
S-27  }
S-28
S-29  /* Note:  The variable tmp is now mapped with tofrom, for correct
S-30     execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-31   */
```

C / C++

The Fortran version of the above C code uses a different syntax. Fortran modules use a list syntax of the **declare target** declaration for the mapping. Here the *N* and *Q* variables are declared in the list form as a comma separated list. The function declaration does not use a list and implicitly assumes the function name. In this Fortran example row and column indices are reversed relative to the C/C++ example, as is usual for codes optimized for memory access.

4.5.5 `declare target` Directive with `link` Clause

In the OpenMP 4.5 standard the **`declare target`** directive was extended to allow static data to be mapped, *when needed*, through a **`link`** clause.

Data storage for items listed in the **`link`** clause becomes available on the device when it is mapped implicitly or explicitly in a **`map`** clause, and it persists for the scope of the mapping (as specified by a **`target`** construct, a **`target data`** construct, or **`target enter/exit data`** constructs).

Tip: When all the global data items will not fit on a device and are not needed simultaneously, use the **`link`** clause and map the data only when it is needed.

The following C and Fortran examples show two sets of data (single precision and double precision) that are global on the host for the entire execution on the host; but are only used globally on the device for part of the program execution. The single precision data are allocated and persist only for the first **`target`** region. Similarly, the double precision data are in scope on the device only for the second **`target`** region.

C / C++

Example declare_target.6.c

```
S-1   #define N 100000000
S-2
S-3   #pragma omp declare target link(sp,sv1,sv2) \
S-4                              link(dp,dv1,dv2)
S-5   float  sp[N], sv1[N], sv2[N];
S-6   double dp[N], dv1[N], dv2[N];
S-7
S-8   void s_init(float *, float *, int);
S-9   void d_init(double *, double *, int);
S-10  void s_output(float *, int);
S-11  void d_output(double *, int);
S-12
S-13  #pragma omp declare target
S-14  void s_vec_mult_accum()
S-15  {
S-16     int i;
S-17
S-18     #pragma omp parallel for
S-19     for (i=0; i<N; i++)
S-20       sp[i] = sv1[i] * sv2[i];
S-21  }
S-22
S-23  void d_vec_mult_accum()
S-24  {
S-25     int i;
S-26
S-27     #pragma omp parallel for
```

```
S-28        for (i=0; i<N; i++)
S-29          dp[i] = dv1[i] * dv2[i];
S-30      }
S-31      #pragma omp end declare target
S-32
S-33      int main()
S-34      {
S-35         s_init(sv1, sv2, N);
S-36         #pragma omp target map(to:sv1,sv2) map(from:sp)
S-37            s_vec_mult_accum();
S-38         s_output(sp, N);
S-39
S-40         d_init(dv1, dv2, N);
S-41         #pragma omp target map(to:dv1,dv2) map(from:dp)
S-42            d_vec_mult_accum();
S-43         d_output(dp, N);
S-44
S-45        return 0;
S-46      }
```

C / C++

4.6 `teams` Constructs

4.6.1 `target` and `teams` Constructs with `omp_get_num_teams` and `omp_get_team_num` Routines

The following example shows how the **target** and **teams** constructs are used to create a league of thread teams that execute a region. The **teams** construct creates a league of at most two teams where the master thread of each team executes the **teams** region.

The **omp_get_num_teams** routine returns the number of teams executing in a **teams** region. The **omp_get_team_num** routine returns the team number, which is an integer between 0 and one less than the value returned by **omp_get_num_teams**. The following example manually distributes a loop across two teams.

C / C++

Example teams.1.c

```
S-1   #include <stdlib.h>
S-2   #include <omp.h>
S-3   float dotprod(float B[], float C[], int N)
S-4   {
S-5      float sum0 = 0.0;
S-6      float sum1 = 0.0;
S-7      #pragma omp target map(to: B[:N], C[:N]) map(tofrom: sum0, sum1)
S-8      #pragma omp teams num_teams(2)
S-9      {
S-10        int i;
S-11        if (omp_get_num_teams() != 2)
S-12           abort();
S-13        if (omp_get_team_num() == 0)
S-14        {
S-15           #pragma omp parallel for reduction(+:sum0)
S-16           for (i=0; i<N/2; i++)
S-17              sum0 += B[i] * C[i];
S-18        }
S-19        else if (omp_get_team_num() == 1)
S-20        {
S-21           #pragma omp parallel for reduction(+:sum1)
S-22           for (i=N/2; i<N; i++)
S-23              sum1 += B[i] * C[i];
S-24        }
S-25     }
S-26     return sum0 + sum1;
S-27  }
S-28
S-29  /* Note:  The variables sum0,sum1 are now mapped with tofrom, for correct
```

```
S-30      execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-31   */
```

C / C++

4.6.2 `target`, `teams`, and `distribute` Constructs

The following example shows how the **`target`**, **`teams`**, and **`distribute`** constructs are used to execute a loop nest in a **`target`** region. The **`teams`** construct creates a league and the master thread of each team executes the **`teams`** region. The **`distribute`** construct schedules the subsequent loop iterations across the master threads of each team.

The number of teams in the league is less than or equal to the variable *num_blocks*. Each team in the league has a number of threads less than or equal to the variable *block_threads*. The iterations in the outer loop are distributed among the master threads of each team.

When a team's master thread encounters the parallel loop construct before the inner loop, the other threads in its team are activated. The team executes the **`parallel`** region and then workshares the execution of the loop.

Each master thread executing the **`teams`** region has a private copy of the variable *sum* that is created by the **`reduction`** clause on the **`teams`** construct. The master thread and all threads in its team have a private copy of the variable *sum* that is created by the **`reduction`** clause on the parallel loop construct. The second private *sum* is reduced into the master thread's private copy of *sum* created by the **`teams`** construct. At the end of the **`teams`** region, each master thread's private copy of *sum* is reduced into the final *sum* that is implicitly mapped into the **`target`** region.

C / C++

Example teams.2.c

```
S-1   #define min(x, y) (((x) < (y)) ? (x) : (y))
S-2
S-3   float dotprod(float B[], float C[], int N, int block_size,
S-4     int num_teams, int block_threads)
S-5   {
S-6       float sum = 0.0;
S-7       int i, i0;
S-8       #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
S-9       #pragma omp teams num_teams(num_teams) thread_limit(block_threads) \
S-10        reduction(+:sum)
S-11      #pragma omp distribute
S-12      for (i0=0; i0<N; i0 += block_size)
S-13         #pragma omp parallel for reduction(+:sum)
S-14         for (i=i0; i< min(i0+block_size,N); i++)
S-15             sum += B[i] * C[i];
S-16      return sum;
```

```
}

/* Note:  The variable sum is now mapped with tofrom, for correct
   execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
 */
```

C / C++

4.6.3 `target teams`, and Distribute Parallel Loop Constructs

The following example shows how the **`target teams`** and distribute parallel loop constructs are used to execute a **`target`** region. The **`target teams`** construct creates a league of teams where the master thread of each team executes the **`teams`** region.

The distribute parallel loop construct schedules the loop iterations across the master threads of each team and then across the threads of each team.

C / C++

Example teams.3.c

```
float dotprod(float B[], float C[], int N)
{
   float sum = 0;
   int i;
   #pragma omp target teams map(to: B[0:N], C[0:N]) \
                            defaultmap(tofrom:scalar) reduction(+:sum)
   #pragma omp distribute parallel for reduction(+:sum)
   for (i=0; i<N; i++)
      sum += B[i] * C[i];
   return sum;
}

/* Note:  The variable sum is now mapped with tofrom from the defaultmap
   clause on the combined target teams construct, for correct
   execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
 */
```

C / C++

4.6.4 `target teams` and Distribute Parallel Loop Constructs with Scheduling Clauses

The following example shows how the **`target teams`** and distribute parallel loop constructs are used to execute a **`target`** region. The **`teams`** construct creates a league of at most eight teams where the master thread of each team executes the **`teams`** region. The number of threads in each team is less than or equal to 16.

The **`distribute`** parallel loop construct schedules the subsequent loop iterations across the master threads of each team and then across the threads of each team.

The **`dist_schedule`** clause on the distribute parallel loop construct indicates that loop iterations are distributed to the master thread of each team in chunks of 1024 iterations.

The **`schedule`** clause indicates that the 1024 iterations distributed to a master thread are then assigned to the threads in its associated team in chunks of 64 iterations.

C / C++

Example teams.4.c

```
S-1   #define N 1024*1024
S-2   float dotprod(float B[], float C[])
S-3   {
S-4       float sum = 0.0;
S-5       int i;
S-6       #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
S-7       #pragma omp teams num_teams(8) thread_limit(16) reduction(+:sum)
S-8       #pragma omp distribute parallel for reduction(+:sum) \
S-9                   dist_schedule(static, 1024) schedule(static, 64)
S-10      for (i=0; i<N; i++)
S-11          sum += B[i] * C[i];
S-12      return sum;
S-13  }
S-14
S-15  /* Note:  The variable sum is now mapped with tofrom, for correct
S-16     execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-17   */
```

C / C++

4.6.5 `target teams` and `distribute simd` Constructs

The following example shows how the **`target teams`** and **`distribute simd`** constructs are used to execute a loop in a **`target`** region. The **`target teams`** construct creates a league of teams where the master thread of each team executes the **`teams`** region.

The **`distribute simd`** construct schedules the loop iterations across the master thread of each team and then uses SIMD parallelism to execute the iterations.

C / C++

Example teams.5.c

```
S-1   extern void init(float *, float *, int);
S-2   extern void output(float *, int);
S-3   void vec_mult(float *p, float *v1, float *v2, int N)
S-4   {
S-5      int i;
S-6      init(v1, v2, N);
S-7      #pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8      #pragma omp distribute simd
S-9      for (i=0; i<N; i++)
S-10       p[i] = v1[i] * v2[i];
S-11     output(p, N);
S-12  }
```

C / C++

4.6.6 `target teams` and Distribute Parallel Loop SIMD Constructs

The following example shows how the **`target teams`** and the distribute parallel loop SIMD constructs are used to execute a loop in a **`target teams`** region. The **`target teams`** construct creates a league of teams where the master thread of each team executes the **`teams`** region.

The distribute parallel loop SIMD construct schedules the loop iterations across the master thread of each team and then across the threads of each team where each thread uses SIMD parallelism.

C / C++

Example teams.6.c

```
S-1   extern void init(float *, float *, int);
S-2   extern void output(float *, int);
S-3   void vec_mult(float *p, float *v1, float *v2, int N)
S-4   {
S-5      int i;
S-6      init(v1, v2, N);
S-7      #pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8      #pragma omp distribute parallel for simd
S-9      for (i=0; i<N; i++)
S-10       p[i] = v1[i] * v2[i];
S-11     output(p, N);
S-12  }
```

C / C++

4.7 Asynchronous `target` Execution and Dependences

Asynchronous execution of a **`target`** region can be accomplished by creating an explicit task around the **`target`** region. Examples with explicit tasks are shown at the beginning of this section.

As of OpenMP 4.5 and beyond the **`nowait`** clause can be used on the **`target`** directive for asynchronous execution. Examples with **`nowait`** clauses follow the explicit **`task`** examples.

This section also shows the use of **`depend`** clauses to order executions through dependences.

4.7.1 Asynchronous `target` with Tasks

The following example shows how the **`task`** and **`target`** constructs are used to execute multiple **`target`** regions asynchronously. The task that encounters the **`task`** construct generates an explicit task that contains a **`target`** region. The thread executing the explicit task encounters a task scheduling point while waiting for the execution of the **`target`** region to complete, allowing the thread to switch back to the execution of the encountering task or one of the previously generated explicit tasks.

C / C++

Example async_target.1.c

```
S-1   #pragma omp declare target
S-2   float F(float);
S-3   #pragma omp end declare target
S-4   #define N 1000000000
S-5   #define CHUNKSZ 1000000
S-6   void init(float *, int);
S-7   float Z[N];
S-8   void pipedF()
S-9   {
S-10     int C, i;
S-11     init(Z, N);
S-12     for (C=0; C<N; C+=CHUNKSZ)
S-13     {
S-14        #pragma omp task shared(Z)
S-15        #pragma omp target map(Z[C:CHUNKSZ])
S-16        #pragma omp parallel for
S-17        for (i=0; i<CHUNKSZ; i++)
S-18           Z[i] = F(Z[i]);
S-19     }
S-20     #pragma omp taskwait
S-21  }
```

C / C++

The Fortran version has an interface block that contains the **declare target**. An identical statement exists in the function declaration (not shown here).

The following example shows how the **task** and **target** constructs are used to execute multiple **target** regions asynchronously. The task dependence ensures that the storage is allocated and initialized on the device before it is accessed.

C / C++

Example async_target.2.c

```
S-1   #include <stdlib.h>
S-2   #include <omp.h>
S-3   #pragma omp declare target
S-4   extern void init(float *, float *, int);
S-5   #pragma omp end declare target
S-6   extern void foo();
S-7   extern void output(float *, int);
S-8   void vec_mult(float *p, int N, int dev)
S-9   {
S-10     float *v1, *v2;
S-11     int i;
S-12     #pragma omp task shared(v1, v2) depend(out: v1, v2)
S-13     #pragma omp target device(dev) map(v1, v2)
S-14     {
S-15         // check whether on device dev
S-16         if (omp_is_initial_device())
S-17            abort();
S-18         v1 = malloc(N*sizeof(float));
S-19         v2 = malloc(N*sizeof(float));
S-20         init(v1, v2, N);
S-21     }
S-22     foo(); // execute other work asychronously
S-23     #pragma omp task shared(v1, v2, p) depend(in: v1, v2)
S-24     #pragma omp target device(dev) map(to: v1, v2) map(from: p[0:N])
S-25     {
S-26         // check whether on device dev
S-27         if (omp_is_initial_device())
S-28            abort();
S-29         #pragma omp parallel for
S-30         for (i=0; i<N; i++)
S-31           p[i] = v1[i] * v2[i];
S-32         free(v1);
S-33         free(v2);
S-34     }
S-35     #pragma omp taskwait
```

```
S-36        output(p, N);
S-37    }
```

C / C++

The Fortran example below is similar to the C version above. Instead of pointers, though, it uses the convenience of Fortran allocatable arrays on the device. In order to preserve the arrays allocated on the device across multiple **target** regions, a **target data** region is used in this case.

If there is no shape specified for an allocatable array in a **map** clause, only the array descriptor (also called a dope vector) is mapped. That is, device space is created for the descriptor, and it is initially populated with host values. In this case, the *v1* and *v2* arrays will be in a non-associated state on the device. When space for *v1* and *v2* is allocated on the device in the first **target** region the addresses to the space will be included in their descriptors.

At the end of the first **target** region, the arrays *v1* and *v2* are preserved on the device for access in the second **target** region. At the end of the second **target** region, the data in array *p* is copied back, the arrays *v1* and *v2* are not.

A **depend** clause is used in the **task** directive to provide a wait at the beginning of the second **target** region, to insure that there is no race condition with *v1* and *v2* in the two tasks. It would be noncompliant to use *v1* and/or *v2* in lieu of *N* in the **depend** clauses, because the use of non-allocated allocatable arrays as list items in a **depend** clause would lead to unspecified behavior.

Note – This example is not strictly compliant with the OpenMP 4.5 specification since the allocation status of allocatable arrays *v1* and *v2* is changed inside the **target** region, which is not allowed. (See the restrictions for the **map** clause in the *Data-mapping Attribute Rules and Clauses* section of the specification.) However, the intention is to relax the restrictions on mapping of allocatable variables in the next release of the specification so that the example will be compliant.

4.7.2 `nowait` Clause on `target` Construct

The following example shows how to execute code asynchronously on a device without an explicit task. The **nowait** clause on a **target** construct allows the thread of the *target task* to perform other work while waiting for the **target** region execution to complete. Hence, the the **target** region can execute asynchronously on the device (without requiring a host thread to idle while waiting for the *target task* execution to complete).

In this example the product of two vectors (arrays), *v1* and *v2*, is formed. One half of the operations is performed on the device, and the last half on the host, concurrently.

After a team of threads is formed the master thread generates the *target task* while the other threads can continue on, without a barrier, to the execution of the host portion of the vector product. The completion of the *target task* (asynchronous target execution) is guaranteed by the synchronization

in the implicit barrier at the end of the host vector-product worksharing loop region. See the **barrier** glossary entry in the OpenMP specification for details.

The host loop scheduling is **dynamic**, to balance the host thread executions, since one thread is being used for offload generation. In the situation where little time is spent by the *target task* in setting up and tearing down the the target execution, **static** scheduling may be desired.

C / C++

Example async_target.3.c

```
S-1
S-2   #include <stdio.h>
S-3
S-4   #define N 1000000      //N must be even
S-5   void init(int n, float *v1, float *v2);
S-6
S-7   int main(){
S-8      int   i, n=N;
S-9      int   chunk=1000;
S-10     float v1[N],v2[N],vxv[N];
S-11
S-12     init(n, v1,v2);
S-13
S-14     #pragma omp parallel
S-15     {
S-16
S-17        #pragma omp master
S-18        #pragma omp target teams distribute parallel for nowait \
S-19                                map(to: v1[0:n/2]) \
S-20                                map(to: v2[0:n/2]) \
S-21                                map(from: vxv[0:n/2])
S-22        for(i=0; i<n/2; i++){ vxv[i] = v1[i]*v2[i]; }
S-23
S-24        #pragma omp for schedule(dynamic,chunk)
S-25        for(i=n/2; i<n; i++){ vxv[i] = v1[i]*v2[i]; }
S-26
S-27     }
S-28     printf(" vxv[0] vxv[n-1] %f %f\n", vxv[0], vxv[n-1]);
S-29     return 0;
S-30  }
```

C / C++

4.7.3 Asynchronous `target` with `nowait` and `depend` Clauses

More details on dependences can be found in Section 3.3 on page 47, Task Dependences. In this example, there are three flow dependences. In the first two dependences the target task does not execute until the preceding explicit tasks have finished. These dependences are produced by arrays *v1* and *v2* with the **`out`** dependence type in the first two tasks, and the **`in`** dependence type in the target task.

The last dependence is produced by array *p* with the **`out`** dependence type in the target task, and the **`in`** dependence type in the last task. The last task does not execute until the target task finishes.

The **`nowait`** clause on the **`target`** construct creates a deferrable *target task*, allowing the encountering task to continue execution without waiting for the completion of the *target task*.

C / C++

Example async_target.4.c

```
S-1
S-2   extern void init(  float*, int);
S-3   extern void output(float*, int);
S-4
S-5   void vec_mult(int N)
S-6   {
S-7      int i;
S-8      float p[N], v1[N], v2[N];
S-9
S-10     #pragma omp parallel num_threads(2)
S-11     {
S-12        #pragma omp single
S-13        {
S-14           #pragma omp task depend(out:v1)
S-15           init(v1, N);
S-16
S-17           #pragma omp task depend(out:v2)
S-18           init(v2, N);
S-19
S-20           #pragma omp target nowait depend(in:v1,v2) depend(out:p) \
S-21                                    map(to:v1,v2) map( from: p)
S-22           #pragma omp parallel for private(i)
S-23           for (i=0; i<N; i++)
S-24              p[i] = v1[i] * v2[i];
S-25
S-26           #pragma omp task depend(in:p)
S-27           output(p, N);
S-28        }
S-29     }
S-30  }
```

4.8 Array Sections in Device Constructs

The following examples show the usage of array sections in **map** clauses on **target** and **target data** constructs.

This example shows the invalid usage of two seperate sections of the same array inside of a **target** construct.

C / C++

Example array_sections.1.c

```
S-1   void foo ()
S-2   {
S-3      int A[30];
S-4   #pragma omp target data map( A[0:4] )
S-5   {
S-6      /* Cannot map distinct parts of the same array */
S-7      #pragma omp target map( A[7:20] )
S-8      {
S-9         A[2] = 0;
S-10     }
S-11  }
S-12  }
```

C / C++

This example shows the invalid usage of two separate sections of the same array inside of a **target** construct.

C / C++

Example array_sections.2.c

```
S-1   void foo ()
S-2   {
S-3      int A[30], *p;
S-4   #pragma omp target data map( A[0:4] )
S-5   {
S-6      p = &A[0];
S-7      /* invalid because p[3] and A[3] are the same
S-8       * location on the host but the array section
S-9       * specified via p[...] is not a subset of A[0:4] */
S-10     #pragma omp target map( p[3:20] )
S-11     {
S-12        A[2] = 0;
S-13        p[8] = 0;
S-14     }
S-15  }
S-16  }
```

C / C++

This example shows the valid usage of two separate sections of the same array inside of a **target** construct.

C / C++

Example array_sections.3.c

```
S-1   void foo ()
S-2   {
S-3      int A[30], *p;
S-4   #pragma omp target data map( A[0:4] )
S-5   {
S-6      p = &A[0];
S-7      #pragma omp target map( p[7:20] )
S-8      {
S-9         A[2] = 0;
S-10        p[8] = 0;
S-11     }
S-12  }
S-13  }
```

C / C++

This example shows the valid usage of a wholly contained array section of an already mapped array section inside of a **target** construct.

C / C++

Example array_sections.4.c

```
S-1   void foo ()
S-2   {
S-3      int A[30], *p;
S-4   #pragma omp target data map( A[0:10] )
S-5   {
S-6      p = &A[0];
S-7      #pragma omp target map( p[3:7] )
S-8      {
S-9         A[2] = 0;
S-10        p[8] = 0;
S-11        A[8] = 1;
S-12     }
S-13  }
S-14  }
```

C / C++

4.9 Device Routines

4.9.1 `omp_is_initial_device` Routine

The following example shows how the **`omp_is_initial_device`** runtime library routine can be used to query if a code is executing on the initial host device or on a target device. The example then sets the number of threads in the **`parallel`** region based on where the code is executing.

C / C++

Example device.1.c

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3   #pragma omp declare target
S-4   void vec_mult(float *p, float *v1, float *v2, int N);
S-5   extern float *p, *v1, *v2;
S-6   extern int N;
S-7   #pragma omp end declare target
S-8   extern void init_vars(float *, float *, int);
S-9   extern void output(float *, int);
S-10  void foo()
S-11  {
S-12     init_vars(v1, v2, N);
S-13     #pragma omp target device(42) map(p[:N], v1[:N], v2[:N])
S-14     {
S-15        vec_mult(p, v1, v2, N);
S-16     }
S-17     output(p, N);
S-18  }
S-19  void vec_mult(float *p, float *v1, float *v2, int N)
S-20  {
S-21     int i;
S-22     int nthreads;
S-23     if (!omp_is_initial_device())
S-24     {
S-25        printf("1024 threads on target device\n");
S-26        nthreads = 1024;
S-27     }
S-28     else
S-29     {
S-30        printf("8 threads on initial device\n");
S-31        nthreads = 8;
S-32     }
S-33     #pragma omp parallel for private(i) num_threads(nthreads)
S-34     for (i=0; i<N; i++)
S-35       p[i] = v1[i] * v2[i];
S-36  }
```

C / C++

4.9.2 `omp_get_num_devices` Routine

The following example shows how the **`omp_get_num_devices`** runtime library routine can be used to determine the number of devices.

C / C++

Example device.2.c

```
S-1   #include <omp.h>
S-2   extern void init(float *, float *, int);
S-3   extern void output(float *, int);
S-4   void vec_mult(float *p, float *v1, float *v2, int N)
S-5   {
S-6      int i;
S-7      init(v1, v2, N);
S-8      int ndev = omp_get_num_devices();
S-9      int do_offload = (ndev>0 && N>1000000);
S-10     #pragma omp target if(do_offload) map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-11     #pragma omp parallel for if(N>1000) private(i)
S-12     for (i=0; i<N; i++)
S-13       p[i] = v1[i] * v2[i];
S-14     output(p, N);
S-15  }
```

C / C++

4.9.3 `omp_set_default_device` and `omp_get_default_device` Routines

The following example shows how the **`omp_set_default_device`** and **`omp_get_default_device`** runtime library routines can be used to set the default device and determine the default device respectively.

C / C++

Example device.3.c

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3   void foo(void)
S-4   {
S-5      int default_device = omp_get_default_device();
S-6      printf("Default device = %d\n", default_device);
S-7      omp_set_default_device(default_device+1);
S-8      if (omp_get_default_device() != default_device+1)
S-9         printf("Default device is still = %d\n", default_device);
S-10  }
```

C / C++

4.9.4 Target Memory and Device Pointers Routines

The following example shows how to create space on a device, transfer data to and from that space, and free the space, using API calls. The API calls directly execute allocation, copy and free operations on the device, without invoking any mapping through a **target** directive. The **omp_target_alloc** routine allocates space and returns a device pointer for referencing the space in the **omp_target_memcpy** API routine on the host. The **omp_target_free** routine frees the space on the device.

The example also illustrates how to access that space in a **target** region by exposing the device pointer in an **is_device_ptr** clause.

The example creates an array of cosine values on the default device, to be used on the host device. The function fails if a default device is not available.

C / C++

Example device.4.c

```
S-1   #include <stdio.h>
S-2   #include <math.h>
S-3   #include <stdlib.h>
S-4   #include <omp.h>
S-5
S-6   void get_dev_cos(double *mem, size_t s)
S-7   {
S-8      int h, t, i;
S-9      double * mem_dev_cpy;
S-10     h = omp_get_initial_device();
S-11     t = omp_get_default_device();
```

```
S-12
S-13        if (omp_get_num_devices() < 1 || t < 0){
S-14           printf(" ERROR: No device found.\n");
S-15           exit(1);
S-16        }
S-17
S-18        mem_dev_cpy = omp_target_alloc( sizeof(double) * s, t);
S-19        if(mem_dev_cpy == NULL){
S-20           printf(" ERROR: No space left on device.\n");
S-21           exit(1);
S-22        }
S-23
S-24                                /* dst  src */
S-25        omp_target_memcpy(mem_dev_cpy, mem, sizeof(double)*s,
S-26                                   0,    0,
S-27                                   t,   h);
S-28
S-29        #pragma omp target is_device_ptr(mem_dev_cpy) device(t)
S-30        #pragma omp teams distribute parallel for
S-31          for(i=0;i<s;i++){ mem_dev_cpy[i] = cos((double)i); } /* init data */
S-32
S-33                        /* dst  src */
S-34         omp_target_memcpy(mem, mem_dev_cpy, sizeof(double)*s,
S-35                           0,             0,
S-36                           h,             t);
S-37
S-38         omp_target_free(mem_dev_cpy, t);
S-39    }
```

C / C++

CHAPTER 5

SIMD

Single instruction, multiple data (SIMD) is a form of parallel execution in which the same operation is performed on multiple data elements independently in hardware vector processing units (VPU), also called SIMD units. The addition of two vectors to form a third vector is a SIMD operation. Many processors have SIMD (vector) units that can perform simultaneously 2, 4, 8 or more executions of the same operation (by a single SIMD unit).

Loops without loop-carried backward dependency (or with dependency preserved using ordered simd) are candidates for vectorization by the compiler for execution with SIMD units. In addition, with state-of-the-art vectorization technology and **`declare simd`** construct extensions for function vectorization in the OpenMP 4.5 specification, loops with function calls can be vectorized as well. The basic idea is that a scalar function call in a loop can be replaced by a vector version of the function, and the loop can be vectorized simultaneously by combining a loop vectorization (**`simd`** directive on the loop) and a function vectorization (**`declare simd`** directive on the function).

A **`simd`** construct states that SIMD operations be performed on the data within the loop. A number of clauses are available to provide data-sharing attributes (**`private`**, **`linear`**, **`reduction`** and **`lastprivate`**). Other clauses provide vector length preference/restrictions (**`simdlen`** / **`safelen`**), loop fusion (**`collapse`**), and data alignment (**`aligned`**).

The **`declare simd`** directive designates that a vector version of the function should also be constructed for execution within loops that contain the function and have a **`simd`** directive. Clauses provide argument specifications (**`linear`**, **`uniform`**, and **`aligned`**), a requested vector length (**`simdlen`**), and designate whether the function is always/never called conditionally in a loop (**`branch`**/**`inbranch`**). The latter is for optimizing peformance.

Also, the **`simd`** construct has been combined with the worksharing loop constructs (**`for simd`** and **`do simd`**) to enable simultaneous thread execution in different SIMD units.

5.1 `simd` and `declare simd` Constructs

The following example illustrates the basic use of the **`simd`** construct to assure the compiler that the loop can be vectorized.

C / C++

Example SIMD.1.c

```
S-1  void star( double *a, double *b, double *c, int n, int *ioff )
S-2  {
S-3     int i;
S-4     #pragma omp simd
S-5     for ( i = 0; i < n; i++ )
S-6        a[i] *= b[i] * c[i+ *ioff];
S-7  }
```

C / C++

When a function can be inlined within a loop the compiler has an opportunity to vectorize the loop. By guaranteeing SIMD behavior of a function's operations, characterizing the arguments of the function and privatizing temporary variables of the loop, the compiler can often create faster, vector code for the loop. In the examples below the **`declare simd`** construct is used on the *add1* and *add2* functions to enable creation of their corresponding SIMD function versions for execution within the associated SIMD loop. The functions characterize two different approaches of accessing data within the function: by a single variable and as an element in a data array, respectively. The *add3* C function uses dereferencing.

The **`declare simd`** constructs also illustrate the use of **`uniform`** and **`linear`** clauses. The **`uniform(fact)`** clause indicates that the variable *fact* is invariant across the SIMD lanes. In the *add2* function *a* and *b* are included in the **`unform`** list because the C pointer and the Fortran array references are constant. The *i* index used in the *add2* function is included in a **`linear`** clause with a constant-linear-step of 1, to guarantee a unity increment of the associated loop. In the **`declare simd`** construct for the *add3* C function the **`linear(a,b:1)`** clause instructs the compiler to generate unit-stride loads across the SIMD lanes; otherwise, costly *gather* instructions would be generated for the unknown sequence of access of the pointer dereferences.

In the **`simd`** constructs for the loops the **`private(tmp)`** clause is necessary to assure that the each vector operation has its own *tmp* variable.

Example SIMD.2.c

```
S-1   #include <stdio.h>
S-2
S-3   #pragma omp declare simd uniform(fact)
S-4   double add1(double a, double b, double fact)
S-5   {
S-6      double c;
S-7      c = a + b + fact;
S-8      return c;
S-9   }
S-10
S-11  #pragma omp declare simd uniform(a,b,fact) linear(i:1)
S-12  double add2(double *a, double *b, int i, double fact)
S-13  {
S-14     double c;
S-15     c = a[i] + b[i] + fact;
S-16     return c;
S-17  }
S-18
S-19  #pragma omp declare simd uniform(fact) linear(a,b:1)
S-20  double add3(double *a, double *b, double fact)
S-21  {
S-22     double c;
S-23     c = *a + *b + fact;
S-24     return c;
S-25  }
S-26
S-27  void work( double *a, double *b, int n )
S-28  {
S-29     int i;
S-30     double tmp;
S-31     #pragma omp simd private(tmp)
S-32     for ( i = 0; i < n; i++ ) {
S-33        tmp  = add1( a[i],  b[i], 1.0);
S-34        a[i] = add2( a,     b, i, 1.0) + tmp;
S-35        a[i] = add3(&a[i], &b[i], 1.0);
S-36     }
S-37  }
S-38
S-39  int main(){
S-40     int i;
S-41     const int N=32;
S-42     double a[N], b[N];
S-43
S-44     for ( i=0; i<N; i++ ) {
```

```
S-45          a[i] = i; b[i] = N-i;
S-46       }
S-47
S-48       work(a, b, N );
S-49
S-50       for ( i=0; i<N; i++ ) {
S-51          printf("%d %f\n", i, a[i]);
S-52       }
S-53
S-54       return 0;
S-55    }
```

C / C++

A thread that encounters a SIMD construct executes a vectorized code of the iterations. Similar to the concerns of a worksharing loop a loop vectorized with a SIMD construct must assure that temporary and reduction variables are privatized and declared as reductions with clauses. The example below illustrates the use of **`private`** and **`reduction`** clauses in a SIMD construct.

C / C++

Example SIMD.3.c

```
S-1     double work( double *a, double *b, int n )
S-2     {
S-3        int i;
S-4        double tmp, sum;
S-5        sum = 0.0;
S-6        #pragma omp simd private(tmp) reduction(+:sum)
S-7        for (i = 0; i < n; i++) {
S-8           tmp = a[i] + b[i];
S-9           sum += tmp;
S-10       }
S-11       return sum;
S-12    }
```

C / C++

A **`safelen(N)`** clause in a **`simd`** construct assures the compiler that there are no loop-carried dependencies for vectors of size *N* or below. If the **`safelen`** clause is not specified, then the default safelen value is the number of loop iterations.

The **`safelen(16)`** clause in the example below guarantees that the vector code is safe for vectors up to and including size 16. In the loop, *m* can be 16 or greater, for correct code execution. If the value of *m* is less than 16, the behavior is undefined.

C / C++

Example SIMD.4.c

```
S-1   void work( float *b, int n, int m )
S-2   {
S-3      int i;
S-4      #pragma omp simd safelen(16)
S-5      for (i = m; i < n; i++)
S-6         b[i] = b[i-m] - 1.0f;
S-7   }
```

C / C++

The following SIMD construct instructs the compiler to collapse the *i* and *j* loops into a single SIMD loop in which SIMD chunks are executed by threads of the team. Within the workshared loop chunks of a thread, the SIMD chunks are executed in the lanes of the vector units.

C / C++

Example SIMD.5.c

```
S-1   void work( double **a, double **b, double **c, int n )
S-2   {
S-3      int i, j;
S-4      double tmp;
S-5      #pragma omp for simd collapse(2) private(tmp)
S-6      for (i = 0; i < n; i++) {
S-7         for (j = 0; j < n; j++) {
S-8            tmp = a[i][j] + b[i][j];
S-9            c[i][j] = tmp;
S-10        }
S-11     }
S-12  }
```

C / C++

5.2 `inbranch` and `notinbranch` Clauses

The following examples illustrate the use of the **`declare simd`** construct with the **`inbranch`** and **`notinbranch`** clauses. The **`notinbranch`** clause informs the compiler that the function *foo* is never called conditionally in the SIMD loop of the function *myaddint*. On the other hand, the **`inbranch`** clause for the function goo indicates that the function is always called conditionally in the SIMD loop inside the function *myaddfloat*.

C / C++

Example SIMD.6.c

```
S-1   #pragma omp declare simd linear(p:1) notinbranch
S-2   int foo(int *p){
S-3     *p = *p + 10;
S-4     return *p;
S-5   }
S-6
S-7   int myaddint(int *a, int *b, int n)
S-8   {
S-9   #pragma omp simd
S-10    for (int i=0; i<n; i++){
S-11        a[i]  = foo(&b[i]);  /* foo is not called under a condition */
S-12    }
S-13    return a[n-1];
S-14  }
S-15
S-16  #pragma omp declare simd linear(p:1) inbranch
S-17  float goo(float *p){
S-18    *p = *p + 18.5f;
S-19    return *p;
S-20  }
S-21
S-22  int myaddfloat(float *x, float *y, int n)
S-23  {
S-24  #pragma omp simd
S-25    for (int i=0; i<n; i++){
S-26       x[i] = (x[i] > y[i]) ? goo(&y[i]) : y[i];
S-27         /* goo is called under the condition (or within a branch) */
S-28    }
S-29    return x[n-1];
S-30  }
```

C / C++

In the code below, the function *fib()* is called in the main program and also recursively called in the function *fib()* within an **`if`** condition. The compiler creates a masked vector version and a non-masked vector version for the function *fib()* while retaining the original scalar version of the *fib()* function.

C / C++

Example SIMD.7.c

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3
S-4   #define N 45
S-5   int a[N], b[N], c[N];
S-6
S-7   #pragma omp declare simd inbranch
S-8   int fib( int n )
S-9   {
S-10     if (n <= 1)
S-11        return n;
S-12     else {
S-13        return fib(n-1) + fib(n-2);
S-14     }
S-15  }
S-16
S-17  int main(void)
S-18  {
S-19     int i;
S-20
S-21     #pragma omp simd
S-22     for (i=0; i < N; i++) b[i] = i;
S-23
S-24     #pragma omp simd
S-25     for (i=0; i < N; i++) {
S-26        a[i] = fib(b[i]);
S-27     }
S-28     printf("Done a[%d] = %d\n", N-1, a[N-1]);
S-29     return 0;
S-30  }
```

C / C++

5.3 Loop-Carried Lexical Forward Dependence

The following example tests the restriction on an SIMD loop with the loop-carried lexical forward-dependence. This dependence must be preserved for the correct execution of SIMD loops.

A loop can be vectorized even though the iterations are not completely independent when it has loop-carried dependences that are forward lexical dependences, indicated in the code below by the

read of *A[j+1]* and the write to *A[j]* in C/C++ code (or *A(j+1)* and *A(j)* in Fortran). That is, the read of *A[j+1]* (or *A(j+1)* in Fortran) before the write to *A[j]* (or *A(j)* in Fortran) ordering must be preserved for each iteration in *j* for valid SIMD code generation.

This test assures that the compiler preserves the loop carried lexical forward-dependence for generating a correct SIMD code.

C / C++

Example SIMD.8.c

```
S-1   #include <stdio.h>
S-2   #include <math.h>
S-3
S-4   int   P[1000];
S-5   float A[1000];
S-6
S-7   float do_work(float *arr)
S-8   {
S-9     float pri;
S-10    int i;
S-11  #pragma omp simd lastprivate(pri)
S-12    for (i = 0; i < 999; ++i) {
S-13      int j = P[i];
S-14
S-15      pri = 0.5f;
S-16      if (j % 2 == 0) {
S-17        pri = A[j+1] + arr[i];
S-18      }
S-19      A[j] = pri * 1.5f;
S-20      pri = pri + A[j];
S-21    }
S-22    return pri;
S-23  }
S-24
S-25  int main(void)
S-26  {
S-27    float pri, arr[1000];
S-28    int i;
S-29
S-30    for (i = 0; i < 1000; ++i) {
S-31       P[i]   = i;
S-32       A[i]   = i * 1.5f;
S-33       arr[i] = i * 1.8f;
S-34    }
S-35    pri = do_work(&arr[0]);
S-36    if (pri == 8237.25) {
S-37      printf("passed: result pri = %7.2f (8237.25) \n", pri);
```

```
S-38      }
S-39      else {
S-40        printf("failed: result pri = %7.2f (8237.25) \n", pri);
S-41      }
S-42      return 0;
S-43    }
```

C / C++

CHAPTER 6

Synchronization

The **barrier** construct is a stand-alone directive that requires all threads of a team (within a contention group) to execute the barrier and complete execution of all tasks within the region, before continuing past the barrier.

The **critical** construct is a directive that contains a structured block. The construct allows only a single thread at a time to execute the structured block (region). Multiple critical regions may exist in a parallel region, and may act cooperatively (only one thread at a time in all **critical** regions), or separately (only one thread at a time in each **critical** regions when a unique name is supplied on each **critical** construct). An optional (lock) **hint** clause may be specified on a named **critical** construct to provide the OpenMP runtime guidance in selection a locking mechanism.

On a finer scale the **atomic** construct allows only a single thread at a time to have atomic access to a storage location involving a single read, write, update or capture statement, and a limited number of combinations when specifying the **capture** *atomic-clause* clause. The *atomic-clause* clause is required for some expression statements, but are not required for **update** statements. Please see the details in the *atomic Construct* subsection of the *Directives* chapter in the OpenMP Specifications document.

The **ordered** construct either specifies a structured block in a loop, simd, or loop SIMD region that will be executed in the order of the loop iterations. The ordered construct sequentializes and orders the execution of ordered regions while allowing code outside the region to run in parallel.

Since OpenMP 4.5 the **ordered** construct can also be a stand-alone directive that specifies cross-iteration dependences in a doacross loop nest. The **depend** clause uses a **sink** *dependence-type*, along with a iteration vector argument (vec) to indicate the iteration that satisfies the dependence. The **depend** clause with a **source** *dependence-type* specifies dependence satisfaction.

The **flush** directive is a stand-alone construct that forces a thread's temporal local storage (view) of a variable to memory where a consistent view of the variable storage can be accesses. When the construct is used without a variable list, all the locally thread-visible data as defined by the base language are flushed. A construct with a list applies the flush operation only to the items in the list.

The **flush** construct also effectively insures that no memory (load or store) operation for the variable set (list items, or default set) may be reordered across the **flush** directive.

General-purpose routines provide mutual exclusion semantics through locks, represented by lock variables. The semantics allows a task to *set*, and hence *own* a lock, until it is *unset* by the task that set it. A *nestable* lock can be set multiple times by a task, and is used when in code requires nested control of locks. A *simple lock* can only be set once by the owning task. There are specific calls for the two types of locks, and the variable of a specific lock type cannot be used by the other lock type.

Any explicit task will observe the synchronization prescribed in a **barrier** construct and an implied barrier. Also, additional synchronizations are available for tasks. All children of a task will wait at a **taskwait** (for their siblings to complete). A **taskgroup** construct creates a region in which the current task is suspended at the end of the region until all sibling tasks, and their descendants, have completed. Scheduling constraints on task execution can be prescribed by the **depend** clause to enforce dependence on previously generated tasks. More details on controlling task executions can be found in the *Tasking* Chapter in the OpenMP Specifications document.

6.1 The `critical` Construct

The following example includes several **`critical`** constructs. The example illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a **`critical`** region. Because the two queues in this example are independent, they are protected by **`critical`** constructs with different names, *xaxis* and *yaxis*.

C / C++

Example critical.1.c

```
S-1   int dequeue(float *a);
S-2   void work(int i, float *a);
S-3
S-4   void critical_example(float *x, float *y)
S-5   {
S-6     int ix_next, iy_next;
S-7
S-8     #pragma omp parallel shared(x, y) private(ix_next, iy_next)
S-9     {
S-10      #pragma omp critical (xaxis)
S-11        ix_next = dequeue(x);
S-12      work(ix_next, x);
S-13
S-14      #pragma omp critical (yaxis)
S-15        iy_next = dequeue(y);
S-16      work(iy_next, y);
S-17    }
S-18
S-19  }
```

C / C++

The following example extends the previous example by adding the **`hint`** clause to the **`critical`** constructs.

C / C++

Example critical.2.c

```
S-1   #include <omp.h>
S-2
S-3   int dequeue(float *a);
S-4   void work(int i, float *a);
S-5
S-6   void critical_example(float *x, float *y)
S-7   {
S-8     int ix_next, iy_next;
S-9
S-10    #pragma omp parallel shared(x, y) private(ix_next, iy_next)
S-11    {
S-12      #pragma omp critical (xaxis) hint(omp_lock_hint_contended)
S-13        ix_next = dequeue(x);
S-14      work(ix_next, x);
S-15
S-16      #pragma omp critical (yaxis) hint(omp_lock_hint_contended)
S-17        iy_next = dequeue(y);
S-18      work(iy_next, y);
S-19    }
S-20
S-21  }
```

C / C++

6.2 Worksharing Constructs Inside a `critical` Construct

The following example demonstrates using a worksharing construct inside a **`critical`** construct. This example is conforming because the worksharing **`single`** region is not closely nested inside the **`critical`** region. A single thread executes the one and only section in the **`sections`** region, and executes the **`critical`** region. The same thread encounters the nested **`parallel`** region, creates a new team of threads, and becomes the master of the new team. One of the threads in the new team enters the **`single`** region and increments **`i`** by **`1`**. At the end of this example **`i`** is equal to **`2`**.

C / C++

Example worksharing_critical.1.c

```
S-1   void critical_work()
S-2   {
S-3     int i = 1;
S-4     #pragma omp parallel sections
S-5     {
S-6       #pragma omp section
S-7       {
S-8         #pragma omp critical (name)
S-9         {
S-10          #pragma omp parallel
S-11          {
S-12            #pragma omp single
S-13            {
S-14              i++;
S-15            }
S-16          }
S-17        }
S-18      }
S-19    }
S-20  }
```

C / C++

6.3 Binding of `barrier` Regions

The binding rules call for a **`barrier`** region to bind to the closest enclosing **`parallel`** region.

In the following example, the call from the main program to *sub2* is conforming because the **`barrier`** region (in *sub3*) binds to the **`parallel`** region in *sub2*. The call from the main program to *sub1* is conforming because the **`barrier`** region binds to the **`parallel`** region in subroutine *sub2*.

The call from the main program to *sub3* is conforming because the **`barrier`** region binds to the implicit inactive **`parallel`** region enclosing the sequential part. Also note that the **`barrier`** region in *sub3* when called from *sub2* only synchronizes the team of threads in the enclosing **`parallel`** region and not all the threads created in *sub1*.

C / C++

Example barrier_regions.1.c

```
S-1   void work(int n) {}
S-2
S-3   void sub3(int n)
S-4   {
S-5     work(n);
S-6     #pragma omp barrier
S-7     work(n);
S-8   }
S-9
S-10  void sub2(int k)
S-11  {
S-12    #pragma omp parallel shared(k)
S-13      sub3(k);
S-14  }
S-15
S-16  void sub1(int n)
S-17  {
S-18    int i;
S-19    #pragma omp parallel private(i) shared(n)
S-20    {
S-21      #pragma omp for
S-22      for (i=0; i<n; i++)
S-23        sub2(i);
S-24    }
S-25  }
S-26
S-27  int main()
S-28  {
S-29    sub1(2);
S-30    sub2(2);
```

```
S-31        sub3(2);
S-32        return 0;
S-33    }
```

6.4 The `atomic` Construct

The following example avoids race conditions (simultaneous updates of an element of *x* by multiple threads) by using the **`atomic`** construct .

The advantage of using the **`atomic`** construct in this example is that it allows updates of two different elements of *x* to occur in parallel. If a **`critical`** construct were used instead, then all updates to elements of *x* would be executed serially (though not in any guaranteed order).

Note that the **`atomic`** directive applies only to the statement immediately following it. As a result, elements of *y* are not updated atomically in this example.

C / C++

Example atomic.1.c

```
S-1   float work1(int i)
S-2   {
S-3     return 1.0 * i;
S-4   }
S-5
S-6   float work2(int i)
S-7   {
S-8      return 2.0 * i;
S-9   }
S-10
S-11  void atomic_example(float *x, float *y, int *index, int n)
S-12  {
S-13    int i;
S-14
S-15    #pragma omp parallel for shared(x, y, index, n)
S-16      for (i=0; i<n; i++) {
S-17        #pragma omp atomic update
S-18        x[index[i]] += work1(i);
S-19        y[i] += work2(i);
S-20       }
S-21  }
S-22
S-23  int main()
S-24  {
S-25    float x[1000];
S-26    float y[10000];
S-27    int index[10000];
S-28    int i;
S-29
S-30    for (i = 0; i < 10000; i++) {
S-31      index[i] = i % 1000;
S-32      y[i]=0.0;
S-33    }
```

```
S-34      for (i = 0; i < 1000; i++)
S-35        x[i] = 0.0;
S-36      atomic_example(x, y, index, 10000);
S-37      return 0;
S-38    }
```

C / C++

The following example illustrates the **read** and **write** clauses for the **atomic** directive. These clauses ensure that the given variable is read or written, respectively, as a whole. Otherwise, some other thread might read or write part of the variable while the current thread was reading or writing another part of the variable. Note that most hardware provides atomic reads and writes for some set of properly aligned variables of specific sizes, but not necessarily for all the variable types supported by the OpenMP API.

C / C++

Example atomic.2.c

```
S-1     int atomic_read(const int *p)
S-2     {
S-3         int value;
S-4     /* Guarantee that the entire value of *p is read atomically. No part of
S-5      * *p can change during the read operation.
S-6      */
S-7     #pragma omp atomic read
S-8          value = *p;
S-9          return value;
S-10    }
S-11
S-12    void atomic_write(int *p, int value)
S-13    {
S-14    /* Guarantee that value is stored atomically into *p. No part of *p can
S-15    change
S-16     * until after the entire write operation is completed.
S-17     */
S-18    #pragma omp atomic write
S-19        *p = value;
S-20    }
```

C / C++

The following example illustrates the **capture** clause for the **atomic** directive. In this case the value of a variable is captured, and then the variable is incremented. These operations occur atomically. This particular example could be implemented using the fetch-and-add instruction available on many kinds of hardware. The example also shows a way to implement a spin lock using the **capture** and **read** clauses.

C / C++

Example atomic.3.c

```
int fetch_and_add(int *p)
{
/* Atomically read the value of *p and then increment it. The previous value
is
 * returned. This can be used to implement a simple lock as shown below.
 */
    int old;
#pragma omp atomic capture
    { old = *p; (*p)++; }
    return old;
}

/*
 * Use fetch_and_add to implement a lock
 */
struct locktype {
    int ticketnumber;
    int turn;
};
void do_locked_work(struct locktype *lock)
{
    int atomic_read(const int *p);
    void work();

    // Obtain the lock
    int myturn = fetch_and_add(&lock->ticketnumber);
    while (atomic_read(&lock->turn) != myturn)
       ;
    // Do some work. The flush is needed to ensure visibility of
    // variables not involved in atomic directives

#pragma omp flush
    work();
#pragma omp flush
    // Release the lock
    fetch_and_add(&lock->turn);
}
```

C / C++

6.5 Restrictions on the `atomic` Construct

The following non-conforming examples illustrate the restrictions on the **`atomic`** construct.

C / C++

Example atomic_restrict.1.c

```
S-1   void atomic_wrong ()
S-2   {
S-3    union {int n; float x;} u;
S-4
S-5   #pragma omp parallel
S-6     {
S-7   #pragma omp atomic update
S-8       u.n++;
S-9
S-10  #pragma omp atomic update
S-11      u.x += 1.0;
S-12
S-13  /* Incorrect because the atomic constructs reference the same location
S-14     through incompatible types */
S-15    }
S-16  }
```

C / C++

C / C++

Example atomic_restrict.2.c

```
S-1   void atomic_wrong2 ()
S-2   {
S-3    int  x;
S-4    int *i;
S-5    float   *r;
S-6
S-7    i = &x;
S-8    r = (float *)&x;
S-9
S-10  #pragma omp parallel
S-11    {
S-12  #pragma omp atomic update
S-13      *i += 1;
S-14
S-15  #pragma omp atomic update
S-16      *r += 1.0;
S-17
S-18  /* Incorrect because the atomic constructs reference the same location
S-19     through incompatible types */
```

```
S-20
S-21        }
S-22    }
```

C / C++

Fortran

The following example is non-conforming because `I` and `R` reference the same location but have different types.

Although the following example might work on some implementations, this is also non-conforming:

Fortran

6.6 The `flush` Construct without a List

The following example distinguishes the shared variables affected by a **flush** construct with no list from the shared objects that are not affected:

C / C++

Example flush_nolist.1.c

```
S-1   int x, *p = &x;
S-2
S-3   void f1(int *q)
S-4   {
S-5     *q = 1;
S-6     #pragma omp flush
S-7     /* x, p, and *q are flushed */
S-8     /* because they are shared and accessible */
S-9     /* q is not flushed because it is not shared. */
S-10  }
S-11
S-12  void f2(int *q)
S-13  {
S-14    #pragma omp barrier
S-15    *q = 2;
S-16    #pragma omp barrier
S-17
S-18    /* a barrier implies a flush */
S-19    /* x, p, and *q are flushed */
S-20    /* because they are shared and accessible */
S-21    /* q is not flushed because it is not shared. */
S-22  }
S-23
S-24  int g(int n)
S-25  {
S-26    int i = 1, j, sum = 0;
S-27    *p = 1;
S-28    #pragma omp parallel reduction(+: sum) num_threads(10)
S-29    {
S-30      f1(&j);
S-31
S-32      /* i, n and sum were not flushed */
S-33      /* because they were not accessible in f1 */
S-34      /* j was flushed because it was accessible */
S-35      sum += j;
S-36
S-37      f2(&j);
S-38
S-39      /* i, n, and sum were not flushed */
```

```
S-40        /* because they were not accessible in f2 */
S-41        /* j was flushed because it was accessible */
S-42        sum += i + j + *p + n;
S-43      }
S-44      return sum;
S-45    }
S-46
S-47    int main()
S-48    {
S-49      int result = g(7);
S-50      return result;
S-51    }
```

C / C++

6.7 The `ordered` Clause and the `ordered` Construct

Ordered constructs are useful for sequentially ordering the output from work that is done in parallel. The following program prints out the indices in sequential order:

C / C++

Example ordered.1.c

```
S-1   #include <stdio.h>
S-2
S-3   void work(int k)
S-4   {
S-5     #pragma omp ordered
S-6       printf(" %d\n", k);
S-7   }
S-8
S-9   void ordered_example(int lb, int ub, int stride)
S-10  {
S-11    int i;
S-12
S-13    #pragma omp parallel for ordered schedule(dynamic)
S-14    for (i=lb; i<ub; i+=stride)
S-15      work(i);
S-16  }
S-17
S-18  int main()
S-19  {
S-20    ordered_example(0, 100, 5);
S-21    return 0;
S-22  }
```

C / C++

It is possible to have multiple **`ordered`** constructs within a loop region with the **`ordered`** clause specified. The first example is non-conforming because all iterations execute two **`ordered`** regions. An iteration of a loop must not execute more than one **`ordered`** region:

C / C++

Example ordered.2.c

```
S-1   void work(int i) {}
S-2
S-3   void ordered_wrong(int n)
S-4   {
S-5     int i;
S-6     #pragma omp for ordered
S-7     for (i=0; i<n; i++) {
S-8   /* incorrect because an iteration may not execute more than one
S-9      ordered region */
S-10      #pragma omp ordered
S-11        work(i);
S-12      #pragma omp ordered
S-13        work(i+1);
S-14    }
S-15  }
```

C / C++

The following is a conforming example with more than one **ordered** construct. Each iteration will execute only one **ordered** region:

C / C++

Example ordered.3.c

```
S-1   void work(int i) {}
S-2   void ordered_good(int n)
S-3   {
S-4     int i;
S-5   #pragma omp for ordered
S-6     for (i=0; i<n; i++) {
S-7       if (i <= 10) {
S-8         #pragma omp ordered
S-9            work(i);
S-10      }
S-11      if (i > 10) {
S-12        #pragma omp ordered
S-13          work(i+1);
S-14      }
S-15    }
S-16  }
```

C / C++

6.8 Doacross Loop Nest

An **ordered** clause can be used on a loop construct with an integer parameter argument to define the number of associated loops within a *doacross loop nest* where cross-iteration dependences exist. A **depend** clause on an **ordered** construct within an ordered loop describes the dependences of the *doacross* loops.

In the code below, the **depend(sink:i-1)** clause defines an *i-1* to *i* cross-iteration dependence that specifies a wait point for the completion of computation from iteration *i-1* before proceeding to the subsequent statements. The **depend(source)** clause indicates the completion of computation from the current iteration (*i*) to satisfy the cross-iteration dependence that arises from the iteration. For this example the same sequential ordering could have been achieved with an **ordered** clause without a parameter, on the loop directive, and a single **ordered** directive without the **depend** clause specified for the statement executing the *bar* function.

C / C++

Example doacross.1.c

```
S-1   float foo(int i);
S-2   float bar(float a, float b);
S-3   float baz(float b);
S-4
S-5   void work( int N, float *A, float *B, float *C )
S-6   {
S-7     int i;
S-8
S-9     #pragma omp for ordered(1)
S-10    for (i=1; i<N; i++)
S-11    {
S-12      A[i] = foo(i);
S-13
S-14    #pragma omp ordered depend(sink: i-1)
S-15      B[i] = bar(A[i], B[i-1]);
S-16    #pragma omp ordered depend(source)
S-17
S-18      C[i] = baz(B[i]);
S-19    }
S-20  }
```

C / C++

The following code is similar to the previous example but with *doacross loop nest* extended to two nested loops, *i* and *j*, as specified by the **ordered(2)** clause on the loop directive. In the C/C++ code, the *i* and *j* loops are the first and second associated loops, respectively, whereas in the Fortran code, the *j* and *i* loops are the first and second associated loops, respectively. The **depend(sink:i-1,j)** and **depend(sink:i,j-1)** clauses in the C/C++ code define cross-iteration dependences in two dimensions from iterations (*i-1, j*) and (*i, j-1*) to iteration (*i, j*).

Likewise, the **depend(sink:j-1,i)** and **depend(sink:j,i-1)** clauses in the Fortran code define cross-iteration dependences from iterations (*j-1, i*) and (*j, i-1*) to iteration (*j, i*).

C / C++

Example doacross.2.c

```
S-1   float foo(int i, int j);
S-2   float bar(float a, float b, float c);
S-3   float baz(float b);
S-4
S-5   void work( int N, int M, float **A, float **B, float **C )
S-6   {
S-7     int i, j;
S-8
S-9     #pragma omp for ordered(2)
S-10    for (i=1; i<N; i++)
S-11    {
S-12      for (j=1; j<M; j++)
S-13      {
S-14        A[i][j] = foo(i, j);
S-15
S-16    #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
S-17        B[i][j] = bar(A[i][j], B[i-1][j], B[i][j-1]);
S-18    #pragma omp ordered depend(source)
S-19
S-20        C[i][j] = baz(B[i][j]);
S-21      }
S-22    }
S-23  }
```

C / C++

The following example shows the incorrect use of the **ordered** directive with a **depend** clause. There are two issues with the code. The first issue is a missing **ordered depend(source)** directive, which could cause a deadlock. The second issue is the **depend(sink:i+1,j)** and **depend(sink:i,j+1)** clauses define dependences on lexicographically later source iterations (*i+1, j*) and (*i, j+1*), which could cause a deadlock as well since they may not start to execute until the current iteration completes.

C / C++

Example doacross.3.c

```
S-1   #define N 100
S-2
S-3   void work_wrong(double p[][N][N])
S-4   {
S-5     int i, j, k;
S-6
S-7   #pragma omp parallel for ordered(2) private(i,j,k)
S-8     for (i=1; i<N-1; i++)
S-9     {
S-10      for (j=1; j<N-1; j++)
S-11      {
S-12  #pragma omp ordered depend(sink: i-1,j) depend(sink: i+1,j) \
S-13                     depend(sink: i,j-1) depend(sink: i,j+1)
S-14        for (k=1; k<N-1; k++)
S-15        {
S-16          double tmp1 = p[i-1][j][k] + p[i+1][j][k];
S-17          double tmp2 = p[i][j-1][k] + p[i][j+1][k];
S-18          double tmp3 = p[i][j][k-1] + p[i][j][k+1];
S-19          p[i][j][k] = (tmp1 + tmp2 + tmp3) / 6.0;
S-20        }
S-21  /* missing #pragma omp ordered depend(source) */
S-22      }
S-23    }
S-24  }
```

C / C++

The following example illustrates the use of the **collapse** clause for a *doacross loop nest*. The *i* and *j* loops are the associated loops for the collapsed loop as well as for the *doacross loop nest*. The example also shows a compliant usage of the dependence source directive placed before the corresponding sink directive. Checking the completion of computation from previous iterations at the sink point can occur after the source statement.

C / C++

Example doacross.4.c

```
S-1   double foo(int i, int j);
S-2
S-3   void work( int N, int M, double **A, double **B, double **C )
S-4   {
S-5     int i, j;
S-6     double alpha = 1.2;
S-7
S-8     #pragma omp for collapse(2) ordered(2)
```

```
S-9       for (i = 1; i < N-1; i++)
S-10      {
S-11        for (j = 1; j < M-1; j++)
S-12        {
S-13          A[i][j] = foo(i, j);
S-14      #pragma omp ordered depend(source)
S-15
S-16          B[i][j] = alpha * A[i][j];
S-17
S-18      #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
S-19          C[i][j] = 0.2 * (A[i-1][j] + A[i+1][j] +
S-20                           A[i][j-1] + A[i][j+1] + A[i][j]);
S-21        }
S-22      }
S-23    }
```

C / C++

6.9 Lock Routines

This section is about the use of lock routines for synchronization.

6.9.1 The `omp_init_lock` Routine

The following example demonstrates how to initialize an array of locks in a **parallel** region by using **omp_init_lock**.

C++

Example init_lock.1.cpp

```
S-1   #include <omp.h>
S-2
S-3   omp_lock_t *new_locks()
S-4   {
S-5     int i;
S-6     omp_lock_t *lock = new omp_lock_t[1000];
S-7
S-8     #pragma omp parallel for private(i)
S-9       for (i=0; i<1000; i++)
S-10      {
S-11        omp_init_lock(&lock[i]);
S-12      }
S-13      return lock;
S-14  }
```

C++

6.9.2 The `omp_init_lock_with_hint` Routine

The following example demonstrates how to initialize an array of locks in a **parallel** region by using **omp_init_lock_with_hint**. Note, hints are combined with an **|** or **+** operator in C/C++ and a **+** operator in Fortran.

C++

Example init_lock_with_hint.1.cpp

```
S-1   #include <omp.h>
S-2
S-3   omp_lock_t *new_locks()
S-4   {
S-5     int i;
S-6     omp_lock_t *lock = new omp_lock_t[1000];
S-7
S-8     #pragma omp parallel for private(i)
S-9       for (i=0; i<1000; i++)
S-10      {
S-11        omp_init_lock_with_hint(&lock[i],
S-12          omp_lock_hint_contended | omp_lock_hint_speculative);
S-13      }
S-14      return lock;
S-15  }
```

C++

6.9.3 Ownership of Locks

Ownership of locks has changed since OpenMP 2.5. In OpenMP 2.5, locks are owned by threads; so a lock released by the **omp_unset_lock** routine must be owned by the same thread executing the routine. Beginning with OpenMP 3.0, locks are owned by task regions; so a lock released by the **omp_unset_lock** routine in a task region must be owned by the same task region.

This change in ownership requires extra care when using locks. The following program is conforming in OpenMP 2.5 because the thread that releases the lock **lck** in the parallel region is the same thread that acquired the lock in the sequential part of the program (master thread of parallel region and the initial thread are the same). However, it is not conforming beginning with OpenMP 3.0, because the task region that releases the lock **lck** is different from the task region that acquires the lock.

C / C++

Example lock_owner.1.c

```
S-1   #include <stdlib.h>
S-2   #include <stdio.h>
S-3   #include <omp.h>
S-4
S-5   int main()
S-6   {
```

```
  int x;
  omp_lock_t lck;

  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;

#pragma omp parallel shared (x)
  {
    #pragma omp master
      {
        x = x + 1;
        omp_unset_lock (&lck);
      }

    /* Some more stuff. */
  }
  omp_destroy_lock (&lck);
  return 0;
}
```

C / C++

6.9.4 Simple Lock Routines

In the following example, the lock routines cause the threads to be idle while waiting for entry to the first critical section, but to do other work while waiting for entry to the second. The **omp_set_lock** function blocks, but the **omp_test_lock** function does not, allowing the work in **skip** to be done.

Note that the argument to the lock routines should have type **omp_lock_t**, and that there is no need to flush it.

C / C++

Example simple_lock.1.c

```
#include <stdio.h>
#include <omp.h>
void skip(int i) {}
void work(int i) {}
int main()
{
  omp_lock_t lck;
  int id;
  omp_init_lock(&lck);

```

```
S-11      #pragma omp parallel shared(lck) private(id)
S-12      {
S-13        id = omp_get_thread_num();
S-14
S-15        omp_set_lock(&lck);
S-16        /*  only one thread at a time can execute this printf */
S-17        printf("My thread id is %d.\n", id);
S-18        omp_unset_lock(&lck);
S-19
S-20        while (! omp_test_lock(&lck)) {
S-21          skip(id);   /* we do not yet have the lock,
S-22                         so we must do something else */
S-23        }
S-24
S-25        work(id);       /* we now have the lock
S-26                           and can do the work */
S-27
S-28        omp_unset_lock(&lck);
S-29      }
S-30      omp_destroy_lock(&lck);
S-31
S-32      return 0;
S-33    }
```

C / C++

Note that there is no need to flush the lock variable.

6.9.5 Nestable Lock Routines

The following example demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

C / C++

Example nestable_lock.1.c

```
S-1     #include <omp.h>
S-2     typedef struct {
S-3           int a,b;
S-4           omp_nest_lock_t lck; } pair;
S-5
S-6     int work1();
S-7     int work2();
S-8     int work3();
S-9     void incr_a(pair *p, int a)
S-10    {
```

```
S-11      /* Called only from incr_pair, no need to lock. */
S-12      p->a += a;
S-13    }
S-14    void incr_b(pair *p, int b)
S-15    {
S-16      /* Called both from incr_pair and elsewhere, */
S-17      /* so need a nestable lock. */
S-18
S-19      omp_set_nest_lock(&p->lck);
S-20      p->b += b;
S-21      omp_unset_nest_lock(&p->lck);
S-22    }
S-23    void incr_pair(pair *p, int a, int b)
S-24    {
S-25      omp_set_nest_lock(&p->lck);
S-26      incr_a(p, a);
S-27      incr_b(p, b);
S-28      omp_unset_nest_lock(&p->lck);
S-29    }
S-30    void nestlock(pair *p)
S-31    {
S-32      #pragma omp parallel sections
S-33      {
S-34        #pragma omp section
S-35          incr_pair(p, work1(), work2());
S-36        #pragma omp section
S-37          incr_b(p, work3());
S-38      }
S-39    }
```

C / C++

CHAPTER 7

Data Environment

The OpenMP *data environment* contains data attributes of variables and objects. Many constructs (such as **parallel**, **simd**, **task**) accept clauses to control *data-sharing* attributes of referenced variables in the construct, where *data-sharing* applies to whether the attribute of the variable is *shared*, is *private* storage, or has special operational characteristics (as found in the **firstprivate**, **lastprivate**, **linear**, or **reduction** clause).

The data environment for a device (distinguished as a *device data environment*) is controlled on the host by *data-mapping* attributes, which determine the relationship of the data on the host, the *original* data, and the data on the device, the *corresponding* data.

DATA-SHARING ATTRIBUTES

Data-sharing attributes of variables can be classified as being *predetermined*, *explicitly determined* or *implicitly determined*.

Certain variables and objects have predetermined attributes. A commonly found case is the loop iteration variable in associated loops of a **for** or **do** construct. It has a private data-sharing attribute. Variables with predetermined data-sharing attributes can not be listed in a data-sharing clause; but there are some exceptions (mainly concerning loop iteration variables).

Variables with explicitly determined data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct. Some of the common data-sharing clauses are: **shared**, **private**, **firstprivate**, **lastprivate**, **linear**, and **reduction**.

Variables with implicitly determined data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause of an enclosing construct. For a complete list of variables and objects with predetermined and implicitly determined attributes, please refer to the *Data-sharing Attribute Rules for Variables Referenced in a Construct* subsection of the OpenMP Specifications document.

DATA-MAPPING ATTRIBUTES

The **map** clause on a device construct explictly specifies how the list items in the clause are mapped from the encountering task's data environment (on the host) to the corresponding item in the device data environment (on the device). The common *list items* are arrays, array sections, scalars, pointers, and structure elements (members).

Procedures and global variables have predetermined data mapping if they appear within the list or block of a **declare target** directive. Also, a C/C++ pointer is mapped as a zero-length array section, as is a C++ variable that is a reference to a pointer.

Without explict mapping, non-scalar and non-pointer variables within the scope of the **target** construct are implicitly mapped with a *map-type* of **tofrom**. Without explicit mapping, scalar variables within the scope of the **target** construct are not mapped, but have an implicit firstprivate data-sharing attribute. (That is, the value of the original variable is given to a private variable of the same name on the device.) This behavior can be changed with the **defaultmap** clause.

The **map** clause can appear on **target**, **target data** and **target enter/exit data** constructs. The operations of creation and removal of device storage as well as assignment of the original list item values to the corresponding list items may be complicated when the list item appears on multiple constructs or when the host and device storage is shared. In these cases the item's reference count, the number of times it has been referenced (+1 on entry and -1 on exited) in nested (structured) map regions and/or accumulative (unstructured) mappings, determines the operation. Details of the **map** clause and reference count operation are specified in the *map Clause* subsection of the OpenMP Specifications document.

7.1 The `threadprivate` Directive

The following examples demonstrate how to use the **threadprivate** directive to give each thread a separate counter.

C / C++

Example threadprivate.1.c

```
S-1   int counter = 0;
S-2   #pragma omp threadprivate(counter)
S-3
S-4   int increment_counter()
S-5   {
S-6     counter++;
S-7     return(counter);
S-8   }
```

C / C++

C / C++

The following example uses **threadprivate** on a static variable:

Example threadprivate.2.c

```
S-1   int increment_counter_2()
S-2   {
S-3     static int counter = 0;
S-4     #pragma omp threadprivate(counter)
S-5     counter++;
S-6     return(counter);
S-7   }
```

The following example demonstrates unspecified behavior for the initialization of a **threadprivate** variable. A **threadprivate** variable is initialized once at an unspecified point before its first reference. Because **a** is constructed using the value of **x** (which is modified by the statement **x++**), the value of **a.val** at the start of the **parallel** region could be either 1 or 2. This problem is avoided for **b**, which uses an auxiliary **const** variable and a copy-constructor.

Example threadprivate.3.cpp

```
S-1   class T {
S-2     public:
S-3       int val;
S-4       T (int);
S-5       T (const T&);
S-6   };
S-7
S-8   T :: T (int v){
S-9      val = v;
S-10  }
S-11
S-12  T :: T (const T& t) {
S-13     val = t.val;
S-14  }
S-15
S-16  void g(T a, T b){
S-17     a.val += b.val;
S-18  }
S-19
S-20  int x = 1;
S-21  T a(x);
S-22  const T b_aux(x); /* Capture value of x = 1 */
S-23  T b(b_aux);
S-24  #pragma omp threadprivate(a, b)
S-25
S-26  void f(int n) {
S-27     x++;
S-28     #pragma omp parallel for
S-29     /* In each thread:
S-30      * a is constructed from x (with value 1 or 2?)
S-31      * b is copy-constructed from b_aux
S-32      */
S-33
S-34     for (int i=0; i<n; i++) {
S-35         g(a, b); /* Value of a is unspecified. */
S-36     }
S-37  }
```

C / C++

1 The following examples show non-conforming uses and correct uses of the **threadprivate**
2 directive.

Fortran (cont.)

Fortran (cont.)

Fortran

The following example is non-conforming because the common block is not declared local to the subroutine that refers to it:

The following example is also non-conforming because the common block is not declared local to the subroutine that refers to it:

The following example is a correct rewrite of the previous example:

The following is an example of the use of **threadprivate** for local variables:

The above program, if executed by two threads, will print one of the following two sets of output:

```
a = 11 12 13
ptr = 4
i = 15
```

```
A is not allocated
ptr = 4
i = 5
```

or

```
A is not allocated
ptr = 4
i = 15
```

```
a = 1 2 3
ptr = 4
i = 5
```

The following is an example of the use of **threadprivate** for module variables:

Fortran

C++

The following example illustrates initialization of **threadprivate** variables for class-type **T**. **t1** is default constructed, **t2** is constructed taking a constructor accepting one argument of integer type, **t3** is copy constructed with argument **f()**:

Example threadprivate.4.cpp

```
S-1  static T t1;
S-2  #pragma omp threadprivate(t1)
S-3  static T t2( 23 );
S-4  #pragma omp threadprivate(t2)
S-5  static T t3 = f();
S-6  #pragma omp threadprivate(t3)
```

The following example illustrates the use of **threadprivate** for static class members. The **threadprivate** directive for a static class member must be placed inside the class definition.

Example threadprivate.5.cpp

```
S-1  class T {
S-2   public:
S-3     static int i;
S-4  #pragma omp threadprivate(i)
S-5  };
```

C++

7.2 The `default(none)` Clause

The following example distinguishes the variables that are affected by the **`default(none)`** clause from those that are not.

C / C++

Beginning with OpenMP 4.0, variables with **`const`**-qualified type and no mutable member are no longer predetermined shared. Thus, these variables (variable *c* in the example) need to be explicitly listed in data-sharing attribute clauses when the **`default(none)`** clause is specified.

Example default_none.1.c

```
#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)

void default_none(int a) {
  const int c = 1;
  int i = 0;

  #pragma omp parallel default(none) private(a) shared(z, c)
  {
     int j = omp_get_num_threads();
          /* O.K.  - j is declared within parallel region */
     a = z[j];   /* O.K.  - a is listed in private clause */
                 /*       - z is listed in shared clause */
     x = c;      /* O.K.  - x is threadprivate */
                 /*       - c has const-qualified type and
                                 is listed in shared clause */
     z[i] = y;   /* Error - cannot reference i or y here */

  #pragma omp for firstprivate(y)
          /* Error - Cannot reference y in the firstprivate clause */
     for (i=0; i<10 ; i++) {
        z[i] = i; /* O.K. - i is the loop iteration variable */
     }

     z[i] = y;   /* Error - cannot reference i or y here */
  }
}
```

C / C++

7.3 The `private` Clause

In the following example, the values of original list items *i* and *j* are retained on exit from the **`parallel`** region, while the private list items *i* and *j* are modified within the **`parallel`** construct.

C / C++

Example private.1.c

```
S-1   #include <stdio.h>
S-2   #include <assert.h>
S-3
S-4   int main()
S-5   {
S-6     int i, j;
S-7     int *ptr_i, *ptr_j;
S-8
S-9     i = 1;
S-10    j = 2;
S-11
S-12    ptr_i = &i;
S-13    ptr_j = &j;
S-14
S-15    #pragma omp parallel private(i) firstprivate(j)
S-16    {
S-17      i = 3;
S-18      j = j + 2;
S-19      assert (*ptr_i == 1 && *ptr_j == 2);
S-20    }
S-21
S-22    assert(i == 1 && j == 2);
S-23
S-24    return 0;
S-25  }
```

C / C++

In the following example, all uses of the variable *a* within the loop construct in the routine *f* refer to a private list item *a*, while it is unspecified whether references to *a* in the routine *g* are to a private list item or the original list item.

C / C++

Example private.2.c

```
int a;

void g(int k) {
  a = k; /* Accessed in the region but outside of the construct;
          * therefore unspecified whether original or private list
          * item is modified. */
}


void f(int n) {
  int a = 0;

  #pragma omp parallel for private(a)
   for (int i=1; i<n; i++) {
       a = i;
       g(a*2);     /* Private copy of "a" */
    }
}
```

C / C++

The following example demonstrates that a list item that appears in a **private** clause in a **parallel** construct may also appear in a **private** clause in an enclosed worksharing construct, which results in an additional private copy.

C / C++

Example private.3.c

```
#include <assert.h>
void priv_example3()
{
  int i, a;

  #pragma omp parallel private(a)
  {
     a = 1;
    #pragma omp parallel for private(a)
      for (i=0; i<10; i++)
     {
       a = 2;
     }
    assert(a == 1);
  }
}
```

C / C++

7.4 Fortran Private Loop Iteration Variables

Fortran

In general loop iteration variables will be private, when used in the *do-loop* of a **do** and **parallel do** construct or in sequential loops in a **parallel** construct (see Section 2.7.1 and Section 2.14.1 of the OpenMP 4.0 specification). In the following example of a sequential loop in a **parallel** construct the loop iteration variable *I* will be private.

Example fort_loopvar.1.f90

```
S-1     SUBROUTINE PLOOP_1(A,N)
S-2     INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-3
S-4     REAL A(*)
S-5     INTEGER I, MYOFFSET, N
S-6
S-7     !$OMP PARALLEL PRIVATE(MYOFFSET)
S-8           MYOFFSET = OMP_GET_THREAD_NUM()*N
S-9           DO I = 1, N
S-10            A(MYOFFSET+I) = FLOAT(I)
S-11          ENDDO
S-12    !$OMP END PARALLEL
S-13
S-14    END SUBROUTINE PLOOP_1
```

In exceptional cases, loop iteration variables can be made shared, as in the following example:

Example fort_loopvar.2.f90

```
S-1     SUBROUTINE PLOOP_2(A,B,N,I1,I2)
S-2     REAL A(*), B(*)
S-3     INTEGER I1, I2, N
S-4
S-5     !$OMP PARALLEL SHARED(A,B,I1,I2)
S-6     !$OMP SECTIONS
S-7     !$OMP SECTION
S-8          DO I1 = I1, N
S-9            IF (A(I1).NE.0.0) EXIT
S-10         ENDDO
S-11    !$OMP SECTION
S-12         DO I2 = I2, N
S-13           IF (B(I2).NE.0.0) EXIT
S-14         ENDDO
S-15    !$OMP END SECTIONS
S-16    !$OMP SINGLE
S-17        IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, 'ARE ALL ZERO.'
S-18        IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, 'ARE ALL ZERO.'
S-19    !$OMP END SINGLE
```

```
S-20     !$OMP END PARALLEL
S-21
S-22     END SUBROUTINE PLOOP_2
```

Note however that the use of shared loop iteration variables can easily lead to race conditions.

Fortran

Fortran (cont.)

7.5 Fortran Restrictions on `shared` and `private` Clauses with Common Blocks

Fortran

When a named common block is specified in a **`private`**, **`firstprivate`**, or **`lastprivate`** clause of a construct, none of its members may be declared in another data-sharing attribute clause on that construct. The following examples illustrate this point.

The following example is conforming:

The following example is also conforming:

The following example is conforming:

The following example is non-conforming because **`x`** is a constituent element of **`c`**:

The following example is non-conforming because a common block may not be declared both shared and private:

Fortran

Fortran (cont.)

7.6 Fortran Restrictions on Storage Association with the `private` Clause

Fortran

The following non-conforming examples illustrate the implications of the **`private`** clause rules with regard to storage association.

Fortran

7.7 C/C++ Arrays in a `firstprivate` Clause

C / C++

The following example illustrates the size and value of list items of array or pointer type in a **firstprivate** clause . The size of new list items is based on the type of the corresponding original list item, as determined by the base language.

In this example:

- The type of **A** is array of two arrays of two ints.
- The type of **B** is adjusted to pointer to array of **n** ints, because it is a function parameter.
- The type of **C** is adjusted to pointer to int, because it is a function parameter.
- The type of **D** is array of two arrays of two ints.
- The type of **E** is array of **n** arrays of **n** ints.

Note that **B** and **E** involve variable length array types.

The new items of array type are initialized as if each integer element of the original array is assigned to the corresponding element of the new array. Those of pointer type are initialized as if by assignment from the original item to the new item.

Example carrays_fpriv.1.c

```
S-1   #include <assert.h>
S-2
S-3   int A[2][2] = {1, 2, 3, 4};
S-4
S-5   void f(int n, int B[n][n], int C[])
S-6   {
S-7     int D[2][2] = {1, 2, 3, 4};
S-8     int E[n][n];
S-9
S-10    assert(n >= 2);
S-11    E[1][1] = 4;
S-12
S-13    #pragma omp parallel firstprivate(B, C, D, E)
S-14    {
S-15      assert(sizeof(B) == sizeof(int (*)[n]));
S-16      assert(sizeof(C) == sizeof(int*));
S-17      assert(sizeof(D) == 4 * sizeof(int));
S-18      assert(sizeof(E) == n * n * sizeof(int));
S-19
S-20      /* Private B and C have values of original B and C. */
S-21      assert(&B[1][1] == &A[1][1]);
S-22      assert(&C[3] == &A[1][1]);
S-23      assert(D[1][1] == 4);
S-24      assert(E[1][1] == 4);
S-25    }
```

```
S-26    }
S-27
S-28    int main() {
S-29      f(2, A, A[0]);
S-30      return 0;
S-31    }
```

C / C++

7.8 The `lastprivate` Clause

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables in a **`lastprivate`** clause so that the values of the variables are the same as when the loop is executed sequentially.

C / C++

Example lastprivate.1.c

```
S-1   void lastpriv (int n, float *a, float *b)
S-2   {
S-3     int i;
S-4
S-5     #pragma omp parallel
S-6     {
S-7       #pragma omp for lastprivate(i)
S-8       for (i=0; i<n-1; i++)
S-9         a[i] = b[i] + b[i+1];
S-10    }
S-11
S-12    a[i]=b[i];       /* i == n-1 here */
S-13  }
```

C / C++

7.9 The `reduction` Clause

The following example demonstrates the **`reduction`** clause ; note that some reductions can be expressed in the loop in several ways, as shown for the **`max`** and **`min`** reductions below:

C / C++

Example reduction.1.c

```
S-1   #include <math.h>
S-2   void reduction1(float *x, int *y, int n)
S-3   {
S-4     int i, b, c;
S-5     float a, d;
S-6     a = 0.0;
S-7     b = 0;
S-8     c = y[0];
S-9     d = x[0];
S-10    #pragma omp parallel for private(i) shared(x, y, n) \
S-11                             reduction(+:a) reduction(^:b) \
S-12                             reduction(min:c) reduction(max:d)
S-13      for (i=0; i<n; i++) {
S-14        a += x[i];
S-15        b ^= y[i];
S-16        if (c > y[i]) c = y[i];
S-17        d = fmaxf(d,x[i]);
S-18      }
S-19  }
```

C / C++

A common implementation of the preceding example is to treat it as if it had been written as follows:

C / C++

Example reduction.2.c

```
S-1   #include <limits.h>
S-2   #include <math.h>
S-3   void reduction2(float *x, int *y, int n)
S-4   {
S-5     int i, b, b_p, c, c_p;
S-6     float a, a_p, d, d_p;
S-7     a = 0.0f;
S-8     b = 0;
S-9     c = y[0];
S-10    d = x[0];
S-11    #pragma omp parallel shared(a, b, c, d, x, y, n) \
S-12                             private(a_p, b_p, c_p, d_p)
```

```
S-13      {
S-14        a_p = 0.0f;
S-15        b_p = 0;
S-16        c_p = INT_MAX;
S-17        d_p = -HUGE_VALF;
S-18        #pragma omp for private(i)
S-19        for (i=0; i<n; i++) {
S-20          a_p += x[i];
S-21          b_p ^= y[i];
S-22          if (c_p > y[i]) c_p = y[i];
S-23          d_p = fmaxf(d_p,x[i]);
S-24        }
S-25        #pragma omp critical
S-26        {
S-27          a += a_p;
S-28          b ^= b_p;
S-29          if( c > c_p ) c = c_p;
S-30          d = fmaxf(d,d_p);
S-31        }
S-32      }
S-33    }
```

C / C++

Fortran

1 *Example reduction.2.f90*

```
S-1       SUBROUTINE REDUCTION2(A, B, C, D, X, Y, N)
S-2         REAL :: X(*), A, D
S-3         INTEGER :: Y(*), N, B, C
S-4         REAL :: A_P, D_P
S-5         INTEGER :: I, B_P, C_P
S-6         A = 0
S-7         B = 0
S-8         C = Y(1)
S-9         D = X(1)
S-10        !$OMP PARALLEL SHARED(X, Y, A, B, C, D, N) &
S-11        !$OMP&         PRIVATE(A_P, B_P, C_P, D_P)
S-12          A_P = 0.0
S-13          B_P = 0
S-14          C_P = HUGE(C_P)
S-15          D_P = -HUGE(D_P)
S-16          !$OMP DO PRIVATE(I)
S-17          DO I=1,N
S-18            A_P = A_P + X(I)
S-19            B_P = IEOR(B_P, Y(I))
S-20            C_P = MIN(C_P, Y(I))
S-21            IF (D_P < X(I)) D_P = X(I)
```

```
      END DO
      !$OMP CRITICAL
        A = A + A_P
        B = IEOR(B, B_P)
        C = MIN(C, C_P)
        D = MAX(D, D_P)
      !$OMP END CRITICAL
    !$OMP END PARALLEL
  END SUBROUTINE REDUCTION2
```

The following program is non-conforming because the reduction is on the *intrinsic procedure name* **MAX** but that name has been redefined to be the variable named **MAX**.

Example reduction.3.f90

```
 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
```

The following conforming program performs the reduction using the *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **REN**.

Example reduction.4.f90

```
MODULE M
   INTRINSIC MAX
END MODULE M

PROGRAM REDUCTION3
   USE M, REN => MAX
   N = 0
!$OMP PARALLEL DO REDUCTION(REN: N)     ! still does MAX
   DO I = 1, 100
      N = MAX(N,I)
```

```
   END DO
END PROGRAM REDUCTION3
```

The following conforming program performs the reduction using *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **MIN**.

Example reduction.5.f90

```
MODULE MOD
   INTRINSIC MAX, MIN
END MODULE MOD

PROGRAM REDUCTION4
   USE MOD, MIN=>MAX, MAX=>MIN
   REAL :: R
   R = -HUGE(0.0)

!$OMP PARALLEL DO REDUCTION(MIN: R)     ! still does MAX
   DO I = 1, 1000
      R = MIN(R, SIN(REAL(I)))
   END DO
   PRINT *, R
END PROGRAM REDUCTION4
```

Fortran

The following example is non-conforming because the initialization (**a = 0**) of the original list item **a** is not synchronized with the update of **a** as a result of the reduction computation in the **for** loop. Therefore, the example may print an incorrect value for **a**.

To avoid this problem, the initialization of the original list item **a** should complete before any update of **a** as a result of the **reduction** clause. This can be achieved by adding an explicit barrier after the assignment **a = 0**, or by enclosing the assignment **a = 0** in a **single** directive (which has an implied barrier), or by initializing **a** before the start of the **parallel** region.

C / C++

Example reduction.6.c

```
#include <stdio.h>

int main (void)
{
  int a, i;

  #pragma omp parallel shared(a) private(i)
  {
    #pragma omp master
    a = 0;

```

```
S-12        // To avoid race conditions, add a barrier here.
S-13
S-14        #pragma omp for reduction(+:a)
S-15        for (i = 0; i < 10; i++) {
S-16            a += i;
S-17        }
S-18
S-19        #pragma omp single
S-20        printf ("Sum is %d\n", a);
S-21      }
S-22      return 0;
S-23    }
```

C / C++

The following example demonstrates the reduction of array *a*. In C/C++ this is illustrated by the explicit use of an array section *a[0:N]* in the **reduction** clause. The corresponding Fortran example uses array syntax supported in the base language. As of the OpenMP 4.5 specification the explicit use of array section in the **reduction** clause in Fortran is not permitted. But this oversight will be fixed in the next release of the specification.

C / C++

Example reduction.7.c

```
S-1     #include <stdio.h>
S-2
S-3     #define N 100
S-4     void init(int n, float (*b)[N]);
S-5
S-6     int main(){
S-7
S-8       int i,j;
S-9       float a[N], b[N][N];
S-10
S-11      init(N,b);
S-12
S-13      for(i=0; i<N; i++) a[i]=0.0e0;
S-14
S-15      #pragma omp parallel for reduction(+:a[0:N]) private(j)
S-16      for(i=0; i<N; i++){
S-17        for(j=0; j<N; j++){
S-18           a[j] +=  b[i][j];
S-19        }
S-20      }
S-21      printf(" a[0] a[N-1]: %f %f\n", a[0], a[N-1]);
S-22
S-23      return 0;
S-24    }
```

7.10 The `copyin` Clause

The **copyin** clause is used to initialize threadprivate data upon entry to a **parallel** region. The value of the threadprivate variable in the master thread is copied to the threadprivate variable of each other team member.

C / C++

Example copyin.1.c

```
S-1   #include <stdlib.h>
S-2
S-3   float* work;
S-4   int size;
S-5   float tol;
S-6
S-7   #pragma omp threadprivate(work,size,tol)
S-8
S-9   void build()
S-10  {
S-11    int i;
S-12    work = (float*)malloc( sizeof(float)*size );
S-13    for( i = 0; i < size; ++i ) work[i] = tol;
S-14  }
S-15
S-16  void copyin_example( float t, int n )
S-17  {
S-18    tol = t;
S-19    size = n;
S-20    #pragma omp parallel copyin(tol,size)
S-21    {
S-22      build();
S-23    }
S-24  }
```

C / C++

7.11 The `copyprivate` Clause

The **copyprivate** clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads. In this example, if the routine is called from the sequential part, its behavior is not affected by the presence of the directives. If it is called from a **parallel** region, then the actual arguments with which **a** and **b** are associated must be private.

The thread that executes the structured block associated with the **single** construct broadcasts the values of the private variables **a**, **b**, **x**, and **y** from its implicit task's data environment to the data environments of the other implicit tasks in the thread team. The broadcast completes before any of the threads have left the barrier at the end of the construct.

C / C++

Example copyprivate.1.c

```
S-1   #include <stdio.h>
S-2   float x, y;
S-3   #pragma omp threadprivate(x, y)
S-4
S-5   void init(float a, float b ) {
S-6       #pragma omp single copyprivate(a,b,x,y)
S-7       {
S-8           scanf("%f %f %f %f", &a, &b, &x, &y);
S-9       }
S-10  }
```

C / C++

In this example, assume that the input must be performed by the master thread. Since the **master** construct does not support the **copyprivate** clause, it cannot broadcast the input value that is read. However, **copyprivate** is used to broadcast an address where the input value is stored.

C / C++

Example copyprivate.2.c

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3
S-4   float read_next( ) {
S-5     float * tmp;
S-6     float return_val;
S-7
S-8     #pragma omp single copyprivate(tmp)
S-9     {
S-10      tmp = (float *) malloc(sizeof(float));
S-11    }  /* copies the pointer only */
S-12
```

```
S-13
S-14      #pragma omp master
S-15      {
S-16        scanf("%f", tmp);
S-17      }
S-18
S-19      #pragma omp barrier
S-20      return_val = *tmp;
S-21      #pragma omp barrier
S-22
S-23      #pragma omp single nowait
S-24      {
S-25        free(tmp);
S-26      }
S-27
S-28      return return_val;
S-29    }
```

C / C++

Suppose that the number of lock variables required within a **parallel** region cannot easily be determined prior to entering it. The **copyprivate** clause can be used to provide access to shared lock variables that are allocated within that **parallel** region.

C / C++

Example copyprivate.3.c

```
S-1     #include <stdio.h>
S-2     #include <stdlib.h>
S-3     #include <omp.h>
S-4
S-5     omp_lock_t *new_lock()
S-6     {
S-7       omp_lock_t *lock_ptr;
S-8
S-9       #pragma omp single copyprivate(lock_ptr)
S-10      {
S-11        lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
S-12        omp_init_lock( lock_ptr );
S-13      }
S-14
S-15      return lock_ptr;
S-16    }
```

C / C++

Fortran

Note that the effect of the **copyprivate** clause on a variable with the **allocatable** attribute is different than on a variable with the **pointer** attribute. The value of **A** is copied (as if by intrinsic assignment) and the pointer **B** is copied (as if by pointer assignment) to the corresponding list items in the other implicit tasks belonging to the **parallel** region.

Fortran

7.12 C++ Reference in Data-Sharing Clauses

C++

C++ reference types are allowed in data-sharing attribute clauses as of OpenMP 4.5, except for the **threadprivate**, **copyin** and **copyprivate** clauses. (See the Data-Sharing Attribute Clauses Section of the 4.5 OpenMP specification.) When a variable with C++ reference type is privatized, the object the reference refers to is privatized in addition to the reference itself. The following example shows the use of reference types in data-sharing clauses in the usual way. Additionally it shows how the data-sharing of formal arguments with a C++ reference type on an orphaned task generating construct is determined implicitly. (See the Data-sharing Attribute Rules for Variables Referenced in a Construct Section of the 4.5 OpenMP specification.)

Example cpp_reference.1.cpp

```
S-1
S-2   void task_body (int &);
S-3   void gen_task (int &x) { // on orphaned task construct reference argument
S-4     #pragma omp task // x is implicitly determined firstprivate(x)
S-5     task_body (x);
S-6   }
S-7   void test (int &y, int &z) {
S-8     #pragma omp parallel private(y)
S-9     {
S-10      y = z + 2;
S-11      gen_task (y); // no matter if the argument is determined private
S-12      gen_task (z); // or shared in the enclosing context.
S-13
S-14      y++;          // each thread has its own int object y refers to
S-15      gen_task (y);
S-16    }
S-17  }
S-18
```

C++

7.13 Fortran ASSOCIATE Construct

Fortran

The following is an invalid example of specifying an associate name on a data-sharing attribute clause. The constraint in the Data Sharing Attribute Rules section in the OpenMP 4.0 API Specifications states that an associate name preserves the association with the selector established at the **ASSOCIATE** statement. The associate name *b* is associated with the shared variable *a*. With the predetermined data-sharing attribute rule, the associate name *b* is not allowed to be specified on the **private** clause.

In next example, within the **parallel** construct, the association name *thread_id* is associated with the private copy of *i*. The print statement should output the unique thread number.

The following example illustrates the effect of specifying a selector name on a data-sharing attribute clause. The associate name *u* is associated with *v* and the variable *v* is specified on the **private** clause of the **parallel** construct. The construct association is established prior to the **parallel** region. The association between *u* and the original *v* is retained (see the Data Sharing Attribute Rules section in the OpenMP 4.0 API Specifications). Inside the **parallel** region, *v* has the value of -1 and *u* has the value of the original *v*.

Example associate.3.f90

```
S-1   program example
S-2     integer :: v
S-3     v = 15
S-4   associate(u => v)
S-5   !$omp parallel private(v)
S-6     v = -1
S-7     print *, v                ! private v=-1
S-8     print *, u                ! original v=15
S-9   !$omp end parallel
S-10  end associate
S-11  end program
```

Fortran

CHAPTER 8

Memory Model

In this chapter, examples illustrate race conditions on access to variables with shared data-sharing attributes. A race condition can exist when two or more threads are involved in accessing a variable in which not all of the accesses are reads; that is, a WaR, RaW or WaW condition exists (R=read, a=after, W=write). A RaR does not produce a race condition. Ensuring thread execution order at the processor level is not enough to avoid race conditions, because the local storage at the processor level (registers, caches, etc.) must be synchronized so that a consistent view of the variable in the memory hierarchy can be seen by the threads accessing the variable.

OpenMP provides a shared-memory model which allows all threads access to *memory* (shared data). Each thread also has exclusive access to *threadprivate memory* (private data). A private variable referenced in an OpenMP directive's structured block is a new version of the original variable (with the same name) for each task (or SIMD lane) within the code block. A private variable is initially undefined (except for variables in **`firstprivate`** and **`linear`** clauses), and the original variable value is unaltered by assignments to the private variable, (except for **`reduction`**, **`lastprivate`** and **`linear`** clauses).

Private variables in an outer **`parallel`** region can be shared by implicit tasks of an inner **`parallel`** region (with a **`share`** clause on the inner **`parallel`** directive). Likewise, a private variable may be shared in the region of an explicit **`task`** (through a **`shared`** clause).

The **`flush`** directive forces a consistent view of local variables of the thread executing the **`flush`**. When a list is supplied on the directive, only the items (variables) in the list are guaranteed to be flushed.

Implied flushes exist at prescribed locations of certain constructs. For the complete list of these locations and associated constructs, please refer to the *flush Construct* section of the OpenMP Specifications document.

Examples 1-3 show the difficulty of synchronizing threads through **`flush`** and **`atomic`** directives.

8.1 The OpenMP Memory Model

In the following example, at Print 1, the value of x could be either 2 or 5, depending on the timing of the threads, and the implementation of the assignment to x. There are two reasons that the value at Print 1 might not be 5. First, Print 1 might be executed before the assignment to x is executed. Second, even if Print 1 is executed after the assignment, the value 5 is not guaranteed to be seen by thread 1 because a flush may not have been executed by thread 0 since the assignment.

The barrier after Print 1 contains implicit flushes on all threads, as well as a thread synchronization, so the programmer is guaranteed that the value 5 will be printed by both Print 2 and Print 3.

C / C++

Example mem_model.1.c

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   int main(){
S-5     int x;
S-6
S-7     x = 2;
S-8     #pragma omp parallel num_threads(2) shared(x)
S-9     {
S-10
S-11      if (omp_get_thread_num() == 0) {
S-12         x = 5;
S-13      } else {
S-14      /* Print 1: the following read of x has a race */
S-15        printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
S-16      }
S-17
S-18      #pragma omp barrier
S-19
S-20      if (omp_get_thread_num() == 0) {
S-21      /* Print 2 */
S-22        printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
S-23      } else {
S-24      /* Print 3 */
S-25        printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
S-26      }
S-27    }
S-28    return 0;
S-29  }
```

C / C++

The following example demonstrates why synchronization is difficult to perform correctly through variables. The value of flag is undefined in both prints on thread 1 and the value of data is only well-defined in the second print.

C / C++

Example mem_model.2.c

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3   int main()
S-4   {
S-5       int data;
S-6       int flag=0;
S-7       #pragma omp parallel num_threads(2)
S-8        {
S-9          if (omp_get_thread_num()==0)
S-10          {
S-11              /* Write to the data buffer that will be
S-12              read by thread */
S-13              data = 42;
S-14              /* Flush data to thread 1 and strictly order
S-15              the write to data
S-16              relative to the write to the flag */
S-17              #pragma omp flush(flag, data)
S-18              /* Set flag to release thread 1 */
S-19              flag = 1;
S-20              /* Flush flag to ensure that thread 1 sees
S-21              the change */
S-22              #pragma omp flush(flag)
S-23          }
S-24          else if(omp_get_thread_num()==1)
S-25          {
S-26              /* Loop until we see the update to the flag */
S-27              #pragma omp flush(flag, data)
S-28              while (flag < 1)
S-29                {
S-30                  #pragma omp flush(flag, data)
S-31                }
S-32              /* Values of flag and data are undefined */
S-33              printf("flag=%d data=%d\n", flag, data);
S-34              #pragma omp flush(flag, data)
S-35              /* Values data will be 42, value of flag
S-36              still undefined */
S-37              printf("flag=%d data=%d\n", flag, data);
S-38          }
```

```
    }
    return 0;
}
```

C / C++

The next example demonstrates why synchronization is difficult to perform correctly through variables. Because the *write*(1)-*flush*(1)-*flush*(2)-*read*(2) sequence cannot be guaranteed in the example, the statements on thread 0 and thread 1 may execute in either order.

C / C++

Example mem_model.3.c

```
#include <omp.h>
#include <stdio.h>
int main()
{
    int flag=0;

    #pragma omp parallel num_threads(3)
    {
        if(omp_get_thread_num()==0)
        {
            /* Set flag to release thread 1 */
            #pragma omp atomic update
            flag++;
            /* Flush of flag is implied by the atomic directive */
        }
        else if(omp_get_thread_num()==1)
        {
            /* Loop until we see that flag reaches 1*/
            #pragma omp flush(flag)
            while(flag < 1)
            {
                #pragma omp flush(flag)
            }
            printf("Thread 1 awoken\n");

            /* Set flag to release thread 2 */
            #pragma omp atomic update
            flag++;
            /* Flush of flag is implied by the atomic directive */
        }
        else if(omp_get_thread_num()==2)
        {
            /* Loop until we see that flag reaches 2 */
            #pragma omp flush(flag)
            while(flag < 2)
```

```
S-36                {
S-37                    #pragma omp flush(flag)
S-38                }
S-39                printf("Thread 2 awoken\n");
S-40            }
S-41        }
S-42        return 0;
S-43    }
```

C / C++

8.2 Race Conditions Caused by Implied Copies of Shared Variables in Fortran

Fortran

The following example contains a race condition, because the shared variable, which is an array section, is passed as an actual argument to a routine that has an assumed-size array as its dummy argument. The subroutine call passing an array section argument may cause the compiler to copy the argument into a temporary location prior to the call and copy from the temporary location into the original variable when the subroutine returns. This copying would cause races in the **parallel** region.

Example fort_race.1.f90

```
S-1     SUBROUTINE SHARED_RACE
S-2
S-3       INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-4
S-5       REAL A(20)
S-6       INTEGER MYTHREAD
S-7
S-8     !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)
S-9
S-10      MYTHREAD = OMP_GET_THREAD_NUM()
S-11      IF (MYTHREAD .EQ. 0) THEN
S-12         CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
S-13      ELSE
S-14         A(6:10) = 12
S-15      ENDIF
S-16
S-17    !$OMP END PARALLEL
S-18
S-19    END SUBROUTINE SHARED_RACE
S-20
S-21    SUBROUTINE SUB(X)
S-22      REAL X(*)
S-23      X(1:5) = 4
S-24    END SUBROUTINE SUB
```

Fortran

CHAPTER 9

Program Control

Some specific and elementary concepts of controlling program execution are illustrated in the examples of this chapter. Control can be directly managed with conditional control code (ifdef's with the **_OPENMP** macro, and the Fortran sentinel (**!$**) for conditionally compiling). The **if** clause on some constructs can direct the runtime to ignore or alter the behavior of the construct. Of course, the base-language **if** statements can be used to control the "execution" of stand-alone directives (such as **flush**, **barrier**, **taskwait**, and **taskyield**). However, the directives must appear in a block structure, and not as a substatement as shown in examples 1 and 2 of this chapter.

CANCELLATION

Cancellation (termination) of the normal sequence of execution for the threads in an OpenMP region can be accomplished with the **cancel** construct. The construct uses a *construct-type-clause* to set the region-type to activate for the cancellation. That is, inclusion of one of the *construct-type-clause* names **parallel**, **for**, **do**, **sections** or **taskgroup** on the directive line activates the corresponding region. The **cancel** construct is activated by the first encountering thread, and it continues execution at the end of the named region. The **cancel** construct is also a concellation point for any other thread of the team to also continue execution at the end of the named region.

Also, once the specified region has been activated for cancellation any thread that encounnters a **cancellation point** construct with the same named region (*construct-type-clause*), continues execution at the end of the region.

For an activated **cancel taskgroup** construct, the tasks that belong to the taskgroup set of the innermost enclosing taskgroup region will be canceled.

A task that encounters the cancel taskgroup construct continues execution at the end of its task region. Any task of the taskgroup that has already begun execution will run to completion, unless it encounters a **cancellation point**; tasks that have not begun execution "may" be discarded as completed tasks.

CONTROL VARIABLES

Internal control variables (ICV) are used by implementations to hold values which control the execution of OpenMP regions. Control (and hence the ICVs) may be set as implementation defaults, or set and adjusted through environment variables, clauses, and API functions. Many of the ICV control values are accessible through API function calls. Also, initial ICV values are reported by the runtime if the **OMP_DISPLAY_ENV** environment variable has been set to **TRUE**.

NESTED CONSTRUCTS

Certain combinations of nested constructs are permitted, giving rise to a *combined* construct consisting of two or more constructs. These can be used when the two (or several) constructs would be used immediately in succession (closely nested). A combined construct can use the clauses of the component constructs without restrictions. A *composite* construct is a combined construct which has one or more clauses with (an often obviously) modified or restricted meaning, relative to when the constructs are uncombined.

Certain nestings are forbidden, and often the reasoning is obvious. Worksharing constructs cannot be nested, and the **barrier** construct cannot be nested inside a worksharing construct, or a **critical** construct. Also, **target** constructs cannot be nested.

The **parallel** construct can be nested, as well as the **task** construct. The parallel execution in the nested **parallel** construct(s) is control by the **OMP_NESTED** and **OMP_MAX_ACTIVE_LEVELS** environment variables, and the **omp_set_nested()** and **omp_set_max_active_levels()** functions.

More details on nesting can be found in the *Nesting of Regions* of the *Directives* chapter in the OpenMP Specifications document.

9.1 Conditional Compilation

C / C++

The following example illustrates the use of conditional compilation using the OpenMP macro **_OPENMP**. With OpenMP compilation, the **_OPENMP** macro becomes defined.

Example cond_comp.1.c

```
S-1   #include <stdio.h>
S-2
S-3   int main()
S-4   {
S-5
S-6   # ifdef _OPENMP
S-7       printf("Compiled by an OpenMP-compliant implementation.\n");
S-8   # endif
S-9
S-10      return 0;
S-11  }
```

C / C++

Fortran

The following example illustrates the use of the conditional compilation sentinel. With OpenMP compilation, the conditional compilation sentinel **!$** is recognized and treated as two spaces. In fixed form source, statements guarded by the sentinel must start after column 6.

Fortran

9.2 Internal Control Variables (ICVs)

According to Section 2.3 of the OpenMP 4.0 specification, an OpenMP implementation must act as if there are ICVs that control the behavior of the program. This example illustrates two ICVs, *nthreads-var* and *max-active-levels-var*. The *nthreads-var* ICV controls the number of threads requested for encountered parallel regions; there is one copy of this ICV per task. The *max-active-levels-var* ICV controls the maximum number of nested active parallel regions; there is one copy of this ICV for the whole program.

In the following example, the *nest-var*, *max-active-levels-var*, *dyn-var*, and *nthreads-var* ICVs are modified through calls to the runtime library routines **omp_set_nested**, **omp_set_max_active_levels**, **omp_set_dynamic**, and **omp_set_num_threads** respectively. These ICVs affect the operation of **parallel** regions. Each implicit task generated by a **parallel** region has its own copy of the *nest-var, dyn-var*, and *nthreads-var* ICVs.

In the following example, the new value of *nthreads-var* applies only to the implicit tasks that execute the call to **omp_set_num_threads**. There is one copy of the *max-active-levels-var* ICV for the whole program and its value is the same for all tasks. This example assumes that nested parallelism is supported.

The outer **parallel** region creates a team of two threads; each of the threads will execute one of the two implicit tasks generated by the outer **parallel** region.

Each implicit task generated by the outer **parallel** region calls **omp_set_num_threads(3)**, assigning the value 3 to its respective copy of *nthreads-var*. Then each implicit task encounters an inner **parallel** region that creates a team of three threads; each of the threads will execute one of the three implicit tasks generated by that inner **parallel** region.

Since the outer **parallel** region is executed by 2 threads, and the inner by 3, there will be a total of 6 implicit tasks generated by the two inner **parallel** regions.

Each implicit task generated by an inner **parallel** region will execute the call to **omp_set_num_threads(4)**, assigning the value 4 to its respective copy of *nthreads-var*.

The print statement in the outer **parallel** region is executed by only one of the threads in the team. So it will be executed only once.

The print statement in an inner **parallel** region is also executed by only one of the threads in the team. Since we have a total of two inner **parallel** regions, the print statement will be executed twice – once per inner **parallel** region.

C / C++

Example icv.1.c

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   int main (void)
```

```
S-5   {
S-6     omp_set_nested(1);
S-7     omp_set_max_active_levels(8);
S-8     omp_set_dynamic(0);
S-9     omp_set_num_threads(2);
S-10    #pragma omp parallel
S-11      {
S-12        omp_set_num_threads(3);
S-13
S-14        #pragma omp parallel
S-15          {
S-16            omp_set_num_threads(4);
S-17            #pragma omp single
S-18              {
S-19                /*
S-20                 * The following should print:
S-21                 * Inner: max_act_lev=8, num_thds=3, max_thds=4
S-22                 * Inner: max_act_lev=8, num_thds=3, max_thds=4
S-23                 */
S-24                printf ("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
S-25                omp_get_max_active_levels(), omp_get_num_threads(),
S-26                omp_get_max_threads());
S-27              }
S-28          }
S-29
S-30        #pragma omp barrier
S-31        #pragma omp single
S-32          {
S-33            /*
S-34             * The following should print:
S-35             * Outer: max_act_lev=8, num_thds=2, max_thds=3
S-36             */
S-37            printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
S-38                    omp_get_max_active_levels(), omp_get_num_threads(),
S-39                    omp_get_max_threads());
S-40          }
S-41      }
S-42      return 0;
S-43  }
```

C / C++

9.3 Placement of `flush`, `barrier`, `taskwait` and `taskyield` Directives

The following example is non-conforming, because the **`flush`**, **`barrier`**, **`taskwait`**, and **`taskyield`** directives are stand-alone directives and cannot be the immediate substatement of an **`if`** statement.

C / C++

Example standalone.1.c

```
S-1
S-2   void standalone_wrong()
S-3   {
S-4     int a = 1;
S-5
S-6           if (a != 0)
S-7     #pragma omp flush(a)
S-8   /* incorrect as flush cannot be immediate substatement
S-9      of if statement */
S-10
S-11          if (a != 0)
S-12    #pragma omp barrier
S-13  /* incorrect as barrier cannot be immediate substatement
S-14     of if statement */
S-15
S-16          if (a!=0)
S-17    #pragma omp taskyield
S-18  /* incorrect as taskyield cannot be immediate substatement of if statement
S-19  */
S-20
S-21          if (a != 0)
S-22    #pragma omp taskwait
S-23  /* incorrect as taskwait cannot be immediate substatement
S-24     of if statement */
S-25
S-26  }
```

C / C++

The following example is non-conforming, because the **`flush`**, **`barrier`**, **`taskwait`**, and **`taskyield`** directives are stand-alone directives and cannot be the action statement of an **`if`** statement or a labeled branch target.

The following version of the above example is conforming because the **`flush`**, **`barrier`**, **`taskwait`**, and **`taskyield`** directives are enclosed in a compound statement.

C / C++

Example standalone.2.c

```
S-1   void standalone_ok()
S-2   {
S-3     int a = 1;
S-4
S-5     #pragma omp parallel
S-6     {
S-7        if (a != 0) {
S-8     #pragma omp flush(a)
S-9        }
S-10       if (a != 0) {
S-11    #pragma omp barrier
S-12       }
S-13       if (a != 0) {
S-14    #pragma omp taskwait
S-15       }
S-16          if (a != 0) {
S-17    #pragma omp taskyield
S-18          }
S-19    }
S-20  }
```

C / C++

The following example is conforming because the **flush**, **barrier**, **taskwait**, and **taskyield** directives are enclosed in an **if** construct or follow the labeled branch target.

9.4 Cancellation Constructs

The following example shows how the **cancel** directive can be used to terminate an OpenMP region. Although the **cancel** construct terminates the OpenMP worksharing region, programmers must still track the exception through the pointer ex and issue a cancellation for the **parallel** region if an exception has been raised. The master thread checks the exception pointer to make sure that the exception is properly handled in the sequential part. If cancellation of the **parallel** region has been requested, some threads might have executed **phase_1()**. However, it is guaranteed that none of the threads executed **phase_2()**.

C++

Example cancellation.1.cpp

```
S-1   #include <iostream>
S-2   #include <exception>
S-3   #include <cstddef>
S-4
S-5   #define N 10000
S-6
S-7   extern void causes_an_exception();
S-8   extern void phase_1();
S-9   extern void phase_2();
S-10
S-11  void example() {
S-12      std::exception *ex = NULL;
S-13  #pragma omp parallel shared(ex)
S-14      {
S-15  #pragma omp for
S-16          for (int i = 0; i < N; i++) {
S-17              // no 'if' that prevents compiler optimizations
S-18              try {
S-19                  causes_an_exception();
S-20              }
S-21              catch (std::exception *e) {
S-22                  // still must remember exception for later handling
S-23  #pragma omp atomic write
S-24                  ex = e;
S-25                                  // cancel worksharing construct
S-26  #pragma omp cancel for
S-27              }
S-28          }
S-29     // if an exception has been raised, cancel parallel region
S-30          if (ex) {
S-31  #pragma omp cancel parallel
S-32          }
S-33          phase_1();
S-34  #pragma omp barrier
```

```
S-35            phase_2();
S-36        }
S-37        // continue here if an exception has been thrown in the worksharing loop
S-38        if (ex) {
S-39            // handle exception stored in ex
S-40        }
S-41    }
```

C++

The following example illustrates the use of the **cancel** construct in error handling. If there is an error condition from the **allocate** statement, the cancellation is activated. The encountering thread sets the shared variable **err** and other threads of the binding thread set proceed to the end of the worksharing construct after the cancellation has been activated.

The following example shows how to cancel a parallel search on a binary tree as soon as the search value has been detected. The code creates a task to descend into the child nodes of the current tree node. If the search value has been found, the code remembers the tree node with the found value through an **atomic** write to the result variable and then cancels execution of all search tasks. The function **search_tree_parallel** groups all search tasks into a single task group to control the effect of the **cancel taskgroup** directive. The *level* argument is used to create undeferred tasks after the first ten levels of the tree.

C / C++

Example cancellation.2.c

```
S-1     #include <stddef.h>
S-2
S-3     typedef struct binary_tree_s {
S-4        int value;
S-5        struct binary_tree_s *left, *right;
S-6     } binary_tree_t;
S-7
S-8     binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
S-9         binary_tree_t *found = NULL;
S-10        if (tree) {
S-11            if (tree->value == value) {
S-12                found = tree;
S-13            }
S-14            else {
S-15    #pragma omp task shared(found) if(level < 10)
S-16                {
S-17                    binary_tree_t *found_left = NULL;
S-18                    found_left = search_tree(tree->left, value, level + 1);
S-19                    if (found_left) {
S-20    #pragma omp atomic write
S-21                        found = found_left;
```

```
#pragma omp cancel taskgroup
                    }
                }
#pragma omp task shared(found) if(level < 10)
                {
                    binary_tree_t *found_right = NULL;
                    found_right = search_tree(tree->right, value, level + 1);
                    if (found_right) {
#pragma omp atomic write
                        found = found_right;
#pragma omp cancel taskgroup
                    }
                }
#pragma omp taskwait
            }
        }
        return found;
}
binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
    binary_tree_t *found = NULL;
#pragma omp parallel shared(found, tree, value)
    {
#pragma omp master
        {
#pragma omp taskgroup
            {
                found = search_tree(tree, value, 0);
            }
        }
    }
    return found;
}
```

C / C++

The following is the equivalent parallel search example in Fortran.

9.5 Nested Loop Constructs

The following example of loop construct nesting is conforming because the inner and outer loop regions bind to different **parallel** regions:

C / C++

Example nested_loop.1.c

```
void work(int i, int j) {}

void good_nesting(int n)
{
  int i, j;
  #pragma omp parallel default(shared)
  {
    #pragma omp for
    for (i=0; i<n; i++) {
      #pragma omp parallel shared(i, n)
      {
        #pragma omp for
        for (j=0; j < n; j++)
          work(i, j);
      }
    }
  }
}
```

C / C++

The following variation of the preceding example is also conforming:

C / C++

Example nested_loop.2.c

```
void work(int i, int j) {}


void work1(int i, int n)
{
  int j;
  #pragma omp parallel default(shared)
  {
    #pragma omp for
    for (j=0; j<n; j++)
      work(i, j);
  }
}
```

```
S-15
S-16    void good_nesting2(int n)
S-17    {
S-18      int i;
S-19      #pragma omp parallel default(shared)
S-20      {
S-21        #pragma omp for
S-22        for (i=0; i<n; i++)
S-23          work1(i, n);
S-24      }
S-25    }
```

C / C++

9.6 Restrictions on Nesting of Regions

The examples in this section illustrate the region nesting rules.

The following example is non-conforming because the inner and outer loop regions are closely nested:

C / C++

Example nesting_restrict.1.c

```
S-1   void work(int i, int j) {}
S-2   void wrong1(int n)
S-3   {
S-4     #pragma omp parallel default(shared)
S-5     {
S-6       int i, j;
S-7       #pragma omp for
S-8       for (i=0; i<n; i++) {
S-9          /* incorrect nesting of loop regions */
S-10         #pragma omp for
S-11           for (j=0; j<n; j++)
S-12             work(i, j);
S-13      }
S-14    }
S-15  }
```

C / C++

The following orphaned version of the preceding example is also non-conforming:

C / C++

Example nesting_restrict.2.c

```
S-1   void work(int i, int j) {}
S-2   void work1(int i, int n)
S-3   {
S-4     int j;
S-5   /* incorrect nesting of loop regions */
S-6     #pragma omp for
S-7       for (j=0; j<n; j++)
S-8         work(i, j);
S-9   }
S-10
S-11  void wrong2(int n)
S-12  {
S-13    #pragma omp parallel default(shared)
S-14    {
S-15      int i;
S-16      #pragma omp for
```

```
S-17          for (i=0; i<n; i++)
S-18             work1(i, n);
S-19       }
S-20    }
```

C / C++

The following example is non-conforming because the loop and **single** regions are closely nested:

C / C++

Example nesting_restrict.3.c

```
S-1     void work(int i, int j) {}
S-2     void wrong3(int n)
S-3     {
S-4       #pragma omp parallel default(shared)
S-5       {
S-6         int i;
S-7         #pragma omp for
S-8           for (i=0; i<n; i++) {
S-9     /* incorrect nesting of regions */
S-10            #pragma omp single
S-11              work(i, 0);
S-12          }
S-13      }
S-14    }
```

C / C++

The following example is non-conforming because a **barrier** region cannot be closely nested inside a loop region:

C / C++

Example nesting_restrict.4.c

```
S-1   void work(int i, int j) {}
S-2   void wrong4(int n)
S-3   {
S-4
S-5     #pragma omp parallel default(shared)
S-6     {
S-7       int i;
S-8       #pragma omp for
S-9         for (i=0; i<n; i++) {
S-10          work(i, 0);
S-11  /* incorrect nesting of barrier region in a loop region */
S-12          #pragma omp barrier
S-13          work(i, 1);
S-14        }
S-15    }
S-16  }
```

C / C++

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **critical** region. If this were permitted, it would result in deadlock due to the fact that only one thread at a time can enter the **critical** region:

C / C++

Example nesting_restrict.5.c

```
S-1   void work(int i, int j) {}
S-2   void wrong5(int n)
S-3   {
S-4     #pragma omp parallel
S-5     {
S-6       #pragma omp critical
S-7       {
S-8          work(n, 0);
S-9   /* incorrect nesting of barrier region in a critical region */
S-10         #pragma omp barrier
S-11         work(n, 1);
S-12      }
S-13    }
S-14  }
```

C / C++

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **single** region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the **single** region:

C / C++

Example nesting_restrict.6.c

```
S-1   void work(int i, int j) {}
S-2   void wrong6(int n)
S-3   {
S-4     #pragma omp parallel
S-5     {
S-6       #pragma omp single
S-7       {
S-8         work(n, 0);
S-9   /* incorrect nesting of barrier region in a single region */
S-10        #pragma omp barrier
S-11        work(n, 1);
S-12      }
S-13    }
S-14  }
```

C / C++

APPENDIX A

Document Revision History

A.1 Changes from 4.0.2 to 4.5.0

- Reorganized into chapters of major topics
- Included file extensions in example labels to indicate source type
- Applied the explicit **`map(tofrom)`** for scalar variables in a number of examples to comply with the change of the default behavior for scalar variables from **`map(tofrom)`** to **`firstprivate`** in the 4.5 specification
- Added the following new examples:
  - **`linear`** clause in loop constructs (Section 1.8 on page 15)
  - task priority (Section 3.2 on page 46)
  - **`taskloop`** construct (Section 3.6 on page 54)
  - *directive-name* modifier in multiple **`if`** clauses on a combined construct (Section 4.1.5 on page 59)
  - unstructured data mapping (Section 4.3 on page 68)
  - **`link`** clause for **`declare target`** directive (Section 4.5.5 on page 78)
  - asynchronous target execution with **`nowait`** clause (Section 4.7 on page 86)
  - device memory routines and device pointers (Section 4.9.4 on page 96)
  - doacross loop nest (Section 6.8 on page 123)
  - locks with hints (Section 6.9 on page 127)
  - C/C++ array reduction (Section 7.9 on page 148)
  - C++ reference types in data sharing clauses (Section 7.12 on page 157)

A.2 Changes from 4.0.1 to 4.0.2

- Names of examples were changed from numbers to mnemonics
- Added SIMD examples (Section 5.1 on page 99)
- Applied miscellaneous fixes in several source codes
- Added the revision history

A.3 Changes from 4.0 to 4.0.1

Added the following new examples:

- the **proc_bind** clause (Section 2.1 on page 27)
- the **taskgroup** construct (Section 3.4 on page 51)

A.4 Changes from 3.1 to 4.0

Beginning with OpenMP 4.0, examples were placed in a separate document from the specification document.

Version 4.0 added the following new examples:

- task dependences (Section 3.3 on page 47)
- **target** construct (Section 4.1 on page 56)
- **target data** construct (Section 4.2 on page 61)
- **target update** construct (Section 4.4 on page 70)
- **declare target** construct (Section 4.5 on page 73)
- **teams** constructs (Section 4.6 on page 80)
- asynchronous execution of a **target** region using tasks (Section 4.7.1 on page 86)
- array sections in device constructs (Section 4.8 on page 92)
- device runtime routines (Section 4.9 on page 94)
- Fortran ASSOCIATE construct (Section 7.13 on page 158)
- cancellation constructs (Section 9.4 on page 172)