

The OpenMP Common Core
The Fortran Supplement (Version 1.2.1)

Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges

The MIT Press
Cambridge, Massachusetts
London, England

© 2019-2022 Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges

All rights reserved.

This book was set in L^AT_EX by the authors.

Contents

	Preface to the Fortran supplement	v
1	Parallel Computing	1
2	The Language of Performance	5
3	What is OpenMP?	7
4	Threads and the OpenMP Programming Model	9
5	Parallel Loops	23
6	OpenMP Data Environment	33
7	Tasks in OpenMP	47
8	OpenMP Memory Model	67
9	Common Core Recap	73
10	Multithreading beyond the Common Core	75
11	Synchronization and the OpenMP Memory Model	87
12	Beyond OpenMP Common Core Hardware	95
13	Your Continuing Education in OpenMP	111

Preface to the Fortran supplement

OpenMP is defined for C, C++, and Fortran. Ideally, when you write a book about OpenMP, everything is covered in triplicate; once for each of these programming languages. If that was done, however, the resulting book would be cluttered and painful to read. What’s a poor author to do?

After struggling with the language problem for *many moons* we came upon what we hope is an effective compromise. Our book on the *OpenMP Common Core* covers C and Fortran. Every time we present an item from the OpenMP API, we define it in both C and Fortran. The code discussed in the book, however, only addresses C. The result is a book tightly woven around a set of C examples; free from the clutter of replicated content from other languages.

For C++ programmers, this solution works quite well. While offensive to a “proper” C++ programmer, you can think of C++ as a superset of C. With few exceptions, if you move OpenMP for C into C++, things just work. We thought this was an adequate solution for Fortran programmers as well since surely modern Fortran programmers understand C. Based on informal surveys at numerous OpenMP tutorials, however, we’ve learned that this assumption is not universally true. Many Fortran programmers are not comfortable with C. The authors of the *OpenMP Common Core* started life as Fortran programmers. We love Fortran and would hate to leave our fellow “Formula Translation” buddies out in the cold without the benefit of our excellent book.

We came up with a simple solution to this problem. We produced a supplement to our book that presents every example from the book implemented in Fortran. Our Fortran friends would buy the *OpenMP Common Core* book and download this free supplement. Having the two side by side, Fortran programmers could easily absorb the contents of our book and apply what they have learned to their own Fortran programs.

There is one technical complication to this solution. When presenting OpenMP to C and C++ programmers, we delay introduction of clauses that manipulate the data environment. There is so much to grapple with when learning multithreaded programming. It greatly simplifies the discussion if we don’t move beyond the default rules for data sharing until much later.

For Fortran programmers however, this is not possible. A C programmer can declare a new variable almost anywhere. Fortran, on the other hand, requires that all variables are declared before any executable code. Therefore, to discuss OpenMP with Fortran programmers, we need to present one of the clauses from Chapter 6 (OpenMP Data Environment) right from the beginning. This clause is the **private** clause.

The `private(list)` clause takes a comma separated list of variables as an argument. Each of the variables in the list are declared earlier (in the declarative section of the program). We call those the *original variables*. When the `private` clause is used with a construct that creates threads, each thread allocates a variable for each of the original variables. These new variables are local or “private” to each thread. These variables are uninitialized, so inside the code executed by the OpenMP threads, you must initialize all private variables before using them.

There are numerous details with the `private` clause, but we leave those to chapter 6. To get started with the OpenMP Common Core, you only need to use the private clause with simple scalar variables. Once you know about this clause, its use is almost self evident. You’ll see when you get to its first use in Figure 4.2.

With that small technical detail out of the way, we return to our discussion of the *Fortran Supplement* to our book on the *OpenMP Common Core*. Production of the *Fortran Supplement* took far more time than we expected. As is often the case when writing a book, we were so fixated on the text that we did not organize all the code we used in one place. So one of us (Helen) extracted each and every fragment of C code from the book. She then converted each of them to Fortran. While she tested the resulting code, another member of the team (Tim) independently verified all of her code. He then made a copy of the Latex source code for the *OpenMP Common Core* book and modified it to include just the figures-with-source-code and code-embedded-in-text replacing the C code with Helen’s Fortran code.

The result is this document. With Fortran versions for all the code from our *OpenMP Common Core* book, it should make our book useful to Fortran programmers. We have mirrored the chapter structure of the Common Core book with section headings that call out:

- The figure number for programs presented in the *OpenMP Common Core* book
- The page number from the *OpenMP Common Core* book where code-embedded-in-text is found.

We hope you find our solution to the multi-language problem in OpenMP useful. We really want our Fortran readers to benefit from our wonderful little book on the *OpenMP Common Core*.

Acknowledgments

When we first started teaching OpenMP, Scientific Computing was almost exclusively centered on Fortran. Even if programmers used a different language, they were generally comfortable reading Fortran. Over the years, the dominance of Fortran has slipped only to be replaced by C. In response, our materials for teaching OpenMP shifted taking us to place where we largely ignored Fortran.

This situation is changing. We are moving Fortran back into the core materials we use for teaching OpenMP. We can only do this, however, with help from the OpenMP Fortran community. We all still use Fortran and are comfortable with the language, but we need an expert who is up to date with the latest developments in Fortran to check our work and make sure it is correct. Henry Jin (NASA Ames Research Center) played the “expert Fortran reviewer” role for us. He found numerous errors, both large and small, in our code. We are grateful to Henry for the many hours he spent reviewing our code and making sure that this *Fortran Supplement* is of the highest quality.

1 Parallel Computing

Figure 1.1: Our first OpenMP Program

A simple “Hello World” program to demonstrate concurrent execution.

```
1 program helloworld
2 use omp_lib
3 !$omp parallel
4     write(*, '(a)', advance='no') 'Hello '
5     write(*, '(a)') 'World '
6 !$omp end parallel
7 end program helloworld
```

Figure 1.5: A pthreads “Hello World” program

A “Hello World” where a C function using Pthreads is called from a Fortran Program.

```

1
2 ! This is a Fortran wrapper to call C Pthreads function
3 ! Save the contents in 2 files as below.
4
5 ! To compile:
6 !     gfortran -c Fig_1.5_PosixHello.f90
7 !     gcc -c Fig_1.5_PosixHello_external.c
8 !     gfortran Fig_1.5_PosixHello.o Fig_1.5_PosixHello_external.o
9
10 # File 1: "Fig_1.5_PosixHello.f90"
11
12 program main
13     implicit none
14     external :: pthreads_c
15     call pthreads_c()
16 end program main
17
18 # File 2: "Fig_1.5_PosixHello_external.c"
19
20 #include <pthread.h>
21 #include <stdio.h>
22 #include <stdlib.h>
23 #define NUMTHREADS 4
24
25 void *PrintHelloWorld(void *InputArg)
26 {
27     printf(" Hello ");
28     printf(" World \n");
29 }
30
31 void pthreads_c_()
32 {
33     pthread_t threads[NUMTHREADS];
34     int id;

```

```
35     pthread_attr_t attr;
36     pthread_attr_init(&attr);
37     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
38
39     for (id = 0; id < NUMTHREADS; id++) {
40         pthread_create(&threads[id], &attr, PrintHelloWorld, NULL);
41     }
42
43     for (id = 0; id < NUMTHREADS; id++){
44         pthread_join(threads[id], NULL);
45     }
46
47     pthread_attr_destroy(&attr);
48     pthread_exit(NULL);
49 }
```


2 The Language of Performance

In this chapter, we focus on the words we use when talking about the performance of a parallel program. These concepts are language independent. Hence, there is no C, C++, or Fortran code in this chapter.

3 What is OpenMP?

Table 3.1: The contents of the OpenMP Common Core

A structured block is implied between any directive and its “end directive” form. Square brackets ([]) denote optional items.

Directives, subprograms, clauses, and an environment variable	Description
!\$OMP parallel !\$OMP end parallel	Create a team of threads to execute a structured block of code
integer function omp_get_num_threads() integer function omp_get_thread_num()	Number of threads in a team (N) Thread ID (from 0 to (N-1))
subroutine omp_set_num_threads(numthrds) integer numthrds	Set default number of threads to request when creating a team of threads
double precision function omp_get_wtime()	Wall clock time
export OMP_NUM_THREADS=N	Environment Variable: number of threads
!\$OMP barrier	Wait for all threads in the team
!\$OMP critical !\$OMP end critical	Mutual exclusion synchronization
!\$OMP do [!\$OMP end do]	Divide a loop’s work between the team
!\$OMP parallel do [!\$OMP end parallel do]	Create a team then share work of a loop across the team
reduction(op: list)	Reduction across a team
schedule(static [, chunk]) schedule(dynamic [, chunk])	Fixed loop-distribution at compile time Loop-distribution varies at runtime
private(list) firstprivate (list) shared(list)	Create variable local to each thread/task Create and initialize a private variable Share a variable between threads/tasks
default (none)	Force explicit storage attribute definitions
nowait	Disable implied barriers
!\$OMP single !\$OMP end single	Workshare work so it is done by a single thread
!\$OMP task !\$OMP end task	Create an explicit task
!\$OMP taskwait	Wait for tasks to complete

4 Threads and the OpenMP Programming Model

Figure 4.2: Shared data, private data, and parallel regions

Data movement and parallel regions – This simple program sets the default number of threads to request for a parallel region to 4. A parallel region is defined with a `private` clause so each thread has its own copy of `ID`. The thread ID is set and a simple subroutine is called. Key points from this program: (1) all the threads independently execute the same block of code in this parallel region, (2) all threads have access to the array declared prior to the parallel region, and (3) each thread has its own, private copy of the integer `ID`.

```
1 Program parReg
2   use omp_lib
3   implicit none
4
5   real :: A(10)
6   integer :: ID
7
8   A = 0
9   call omp_set_num_threads(4)
10
11   !$omp parallel private(ID)
12     ID = omp_get_thread_num() + 1
13     call pooh (ID, A)
14     write(*,100)ID, A(ID)
15 100 format("A of ID(", I3, ")=", f10.4)
16   !$omp end parallel
17
18   contains
19
20   subroutine pooh (ID, A)
21     integer :: ID
22     real, dimension(:) :: A
23     A(ID) = ID
24   end subroutine pooh
25
26 end Program parReg
```

Figure 4.3: Thread counts and IDs

Library routines to manage threads – This program shows how to set the default number of threads to request in parallel regions, query the number of threads in a team, and set a unique thread ID. Notice the care taken to avoid a data race when assigning to `size_of_team`.

```
1 Program parReg1
2   use omp_lib
3   implicit none
4
5   integer :: ID, size_of_team, NThrs
6   call omp_set_num_threads(4)
7   !$omp parallel private(ID,NThrs)
8     ID = omp_get_thread_num()
9     NThrs = omp_get_num_threads()
10    if (ID == 0) size_of_team = NThrs
11  !$omp end parallel
12  print *, "We just did the join on a team of size ", size_of_team
13 end Program parReg1
```

Figure 4.5: The Pi program

Serial program to numerically estimate a definite integral using the mid-point rule – The loop iterations are independent other than the summation into `sum`.

```
1          PROGRAM MAIN
2
3          USE OMP_LIB
4          IMPLICIT NONE
5
6          INTEGER :: i
7          INTEGER, PARAMETER :: num_steps = 100000000
8          REAL*8 :: x, pi, sum, step
9          REAL*8 :: start_time, run_time
10
11         sum = 0.0
12
13         step = 1.0 / num_steps
14         start_time = OMP_GET_WTIME()
15
16         DO i = 1, num_steps
17             x = (i - 0.5) * step
18             sum = sum + 4.0 / (1.0 + x * x)
19         ENDDO
20
21         pi = step * sum
22         run_time = OMP_GET_WTIME() - start_time
23
24         WRITE(*,100) pi, num_steps, run_time
25 100    FORMAT('pi = ', f15.8, ', ', i14, ' steps, ', f8.3, ' secs ')
26
27         END PROGRAM MAIN
```

Page 59: Loop level parallelism with the SPMD pattern

```
integer :: i
ID = omp_get_thread_num()
numthreads = omp_get_num_threads()

do i = ID + 1, num_steps, numthreads
  ! body of the loop
end do
```

Figure 4.6: SPMD parallel Pi program

SPMD parallel numerical integration with cyclic distribution of the loop iterations – The program computes the area of the curve defined by the integrand by filling the area under a curve with rectangles and summing up their areas. This version of the program promotes the accumulation variable `sum` to an array and uses a cyclic distribution of loop iterations between threads.

```

1 PROGRAM MAIN
2     USE OMP_LIB
3     IMPLICIT NONE
4
5     INTEGER, PARAMETER :: MAX_THREADS = 4
6     INTEGER :: i, j, id, numthreads, nthreads
7     INTEGER, PARAMETER :: num_steps = 100000000
8     REAL*8 :: pi, real_sum, step, x
9     REAL*8 :: start_time, run_time
10    REAL*8 :: sum(0:MAX_THREADS-1)
11
12    step = 1.0 / num_steps
13
14    CALL OMP_SET_NUM_THREADS(MAX_THREADS)
15    start_time = omp_get_wtime()
16
17    !$OMP PARALLEL PRIVATE(id, x, numthreads)
18        id = omp_get_thread_num()
19        numthreads = OMP_GET_NUM_THREADS()
20        sum(id) = 0.0
21
22        IF (id == 0) THEN
23            nthreads = numthreads
24        ENDIF
25
26        DO i = id, num_steps - 1, numthreads
27            x = (i + 0.5) * step
28            sum(id) = sum(id) + 4.0 / (1.0 + x * x)
29        ENDDO
30    !$OMP END PARALLEL
31
```

```
32      pi = 0.0
33      DO i = 0, nthreads-1
34          pi = pi + sum(i)
35      ENDDO
36
37      pi = step * pi
38      run_time = OMP_GET_WTIME() - start_time
39      WRITE(*,100) pi, num_steps, run_time
40 100    FORMAT('pi = ', f15.8, ', ', i14, ' steps ', f8.3, ' secs ')
41
42      END PROGRAM MAIN
```

Figure 4.7: SPMD Pi program with block-distribution of loop iterations

SPMD parallel numerical integration with block decomposition of the loop iterations – The parallel region from the code in Figure 4.6, but replacing the cyclic distribution of loop iterations with a block distribution.

```
1         step = 1.0 / num_steps
2
3  !$OMP PARALLEL PRIVATE(id,x,numthreads, istart,iend)
4         id = omp_get_thread_num()
5         numthreads = OMP_GET_NUM_THREADS()
6         sum(id) = 0.0
7
8         IF (id == 0) THEN
9             nthreads = numthreads
10        ENDIF
11
12        istart = id * num_steps / numthreads + 1
13        iend = (id+1) * num_steps / numthreads
14        if (id == (numthreads -1)) iend = num_steps
15
16        DO i = istart, iend
17            x = (i - 0.5) * step
18            sum(id) = sum(id) + 4.0 / (1.0 + x * x)
19        ENDDO
20  !$OMP END PARALLEL
```

Figure 4.9: Remove false sharing with a padded array

Padded sum array numerical integration – The `sum` array padded to fill an L1 cache line with the extra dimension and put subsequent rows of `sum`, i.e., each `sum(0,id)`, on different cache lines.

```

1      PROGRAM MAIN
2      USE OMP_LIB
3      IMPLICIT NONE
4
5      INTEGER :: i, j, id, numthreads, nthreads
6      INTEGER, PARAMETER :: num_steps=100000000
7      INTEGER, PARAMETER :: MAX_THREADS=4
8      INTEGER, PARAMETER :: CBLK=8
9      REAL*8 :: pi, step, x
10     REAL*8 :: start_time, run_time
11     REAL*8 :: sum(CBLK,0:MAX_THREADS-1)
12
13     step = 1.0 / num_steps
14
15     CALL OMP_SET_NUM_THREADS(MAX_THREADS)
16     start_time = omp_get_wtime()
17
18     !$OMP PARALLEL PRIVATE(id, x, numthreads)
19         id = omp_get_thread_num()
20         numthreads = OMP_GET_NUM_THREADS()
21         sum(1,id) = 0.0
22
23         IF (id == 0) THEN
24             nthreads = numthreads
25         ENDIF
26
27         DO i = id, num_steps-1, numthreads
28             x = (i + 0.5) * step
29             sum(1,id) = sum(1,id) + 4.0 / (1.0 + x * x)
30         ENDDO
31     !$OMP END PARALLEL
32
33     pi = 0.0

```



```
34      DO i = 0, nthreads - 1
35          pi = pi + sum(1,i)
36      ENDDO
37
38      pi = step * pi
39      run_time = OMP_GET_WTIME() - start_time
40      WRITE(*,100) pi, run_time, nthreads
41 100    FORMAT('pi is ',f15.8,' in ',f8.3,' secs and ',i3,' threads')
42
43      END PROGRAM MAIN
```

Figure 4.10: Mutual exclusion synchronization with critical

A critical section – `consume()` must be called by one thread at a time.

```

1 ! sample compile command to generate the *.o object file :
2 !           gfortran -fopenmp -c Fig_4.10_crit.f90
3
4 program crit
5     use omp_lib
6     implicit none
7     real :: res = 0.0
8     integer :: niters = 1000
9     real :: B
10    integer :: i, id, nthrds
11
12    interface
13        function big_job(i)
14            real :: big_job
15            integer, intent(in) :: i
16        end function big_job
17
18        function consume(a)
19            real :: consume
20            real, intent(in) :: a
21        end function consume
22    end interface
23
24    !$omp parallel private (id, nthrds, B)
25        id = omp_get_thread_num()
26        nthrds = omp_get_num_threads()
27        do i = id, niters - 1, nthrds
28            B = big_job(i)
29            !$omp critical
30                res = res + consume(B)
31            !$omp end critical
32        end do
33    !$omp end parallel
34 end program crit

```

Figure 4.11: Pi program with a Critical section

Numerical integration with a critical section – The partial sums go into a private variable allocated by each thread. These private variables are extremely unlikely to reside on the same L1 cache lines and therefore, there will be no false sharing. The partial sums are combined inside a critical section so there is no data race.

```

1      PROGRAM MAIN
2      USE OMP_LIB
3      IMPLICIT NONE
4
5      INTEGER :: i, j, id, numthreads, nthreads
6      INTEGER, PARAMETER :: num_steps=100000000
7      INTEGER, PARAMETER :: MAX_THREADS=4
8      REAL*8 :: pi, real_sum, step, full_sum, x
9      REAL*8 :: start_time, run_time
10     REAL*8 :: partial_sum
11
12     full_sum = 0.0
13
14     step = 1.0 / num_steps
15
16     CALL OMP_SET_NUM_THREADS(MAX_THREADS)
17     full_sum = 0.0
18     start_time = OMP_GET_WTIME()
19
20     !$OMP PARALLEL PRIVATE(i, id, numthreads, partial_sum, x)
21         id = OMP_GET_THREAD_NUM()
22         numthreads = OMP_GET_NUM_THREADS()
23         partial_sum = 0.0
24
25         if (id == 0) nthreads = numthreads
26
27         DO i = id, num_steps-1, numthreads
28             x = (i+0.5)*step
29             partial_sum = partial_sum + 4.0/(1.0+x*x)
30         ENDDO
31
32     !$OMP CRITICAL

```

```
33         full_sum = full_sum + partial_sum
34 !$OMP END CRITICAL
35
36 !$OMP END PARALLEL
37
38         pi = step * full_sum
39         run_time = OMP_GET_WTIME() - start_time
40         WRITE(*,100) pi, run_time, nthreads
41 100     FORMAT('pi is ',f15.8,' in ',f8.3,'secs and ',i3,' threads ')
42
43         END PROGRAM MAIN
```

Figure 4.12: Barrier synchronization

Example of an explicit barrier – An explicit barrier is used to assure that all threads complete filling the array `Arr` before using it to compute `Brr`. We assume the SPMD pattern so we pass the thread id and the number of threads to all the functions. Notice that only one thread saves the number of threads to a shared variable should it be needed after the parallel region.

```
1 real*8 :: Arr(8), Brr(8)
2 integer :: numthrs
3 integer :: id, nthrds
4 real*8, external :: lots_of_work, needs_all_of_Arr
5
6 call omp_set_num_threads(8)
7 !$omp parallel private (id, nthrds)
8     id = omp_get_thread_num() + 1
9     nthrds = omp_get_num_threads()
10    if (id == 1) numthrs = nthrds
11    Arr(id) = lots_of_work(id, nthrds)
12 !$omp barrier
13    Brr(id) = needs_all_of_Arr(id, nthrds, Arr)
14 !$omp end parallel
```


5 Parallel Loops

Page 75: Basic vector addition loop

```
do i = 1, N
    a(i) = a(i) + b(i)
end do
```

Figure 5.1: SPMD pattern for loop-level parallelism

SPMD parallel vector add program – Create a team of threads and assign one chunk of loop iterations to each thread.

```
1 ! OpenMP parallel region and SPMD pattern
2
3 integer :: id, i, Nthrds, istart, iend
4
5 !$omp parallel private(id,i,istart,iend,Nthrds)
6     id = omp_get_thread_num()
7     Nthrds = omp_get_num_threads()
8     istart = id * N / Nthrds + 1
9     iend = (id + 1) * N / Nthrds
10    if (id == Nthrds - 1) iend = N
11    do i = istart, iend
12        a(i) = a(i) + b(i)
13    end do
14 !$omp end parallel
```

Page 76: Worksharing-loop directives

```
!$omp do
    ...
!$omp end do
```

Figure 5.2: The worksharing-loop construct

Loop-level parallelism for the vector add program – We create a team of threads and then add a single directive to split up loop iterations among threads.

```

1 ! OpenMP parallel region and a worksharing-loop construct
2
3 !$omp parallel
4     !$omp do
5         do i = 1, N
6             a(i) = a(i) + b(i)
7         end do
8     !$omp end do
9 !$omp end parallel

```

Page 77: Standard do loop format to use with a worksharing-loop construct

```

do i = init, end, incr
    structured block
end do

```

Figure 5.3: Parallel and worksharing-loop constructs

An example of a parallel worksharing-loop construct – Create multiple threads, then split the loop iterations among multiple threads to share the work.

```

1 !$omp parallel
2     !$omp do
3         do i = N, 0, -2
4             call NEAT_STUFF(i)
5         enddo
6     !$omp end do
7 !$omp end parallel

```


Page 79: Combined parallel worksharing-loop construct

The following pattern with a pair of OpenMP constructs, one to create the team of threads and the other to split up loop iterations among threads, is very common:

```
!$omp parallel
    !$omp do
        do-loop
    !$omp end do
!$omp end parallel
```

As a convenience, these two directives can be combined into a single directive:

```
!$omp parallel do
    do-loop
!$omp end parallel do
```

Figure 5.4: A simple program with a reduction

A serial reduction example – This loop has a loop-carried dependence through the variable *ave* and therefore, the loop cannot be parallelized with a worksharing-loop directive without completely changing the body of the loop.

```
1      integer :: i
2      real*8  :: ave, A(N)
3
4      call Init(A,N)
5      ave = 0.0
6
7      do i = 1, N
8          ave = ave + A(i)
9      enddo
10     ave = ave/N
```

Figure 5.5: Worksharing-loop with a reduction

An OpenMP reduction –Each thread has a private copy of the variable `ave` to use for its loop iterations. At the end of the loop, these values are combined to create the final value of the reduction which is then combined with the globally visible, shared copy of the variable `ave`.

```
1      integer :: i
2      real*8  :: ave, A(N)
3
4      call Init(A,N)
5      ave = 0.0
6
7      !$omp parallel do reduction(+:ave)
8      do i = 1, N
9          ave = ave + A(i)
10     enddo
11     !$omp end parallel do
12
13     ave = ave/N
```

Figure 5.6: Loop schedules specified “at compile time”

A worksharing-loop with a static schedule – In this example, the work is predictable and balanced for each loop index. Using the static schedule is expected to work best in this case.

```

1 program main
2   use omp_lib
3   implicit none
4
5   ! Use a smaller value of ITER if available memory is too small
6   integer, parameter :: ITER = 100000000
7   integer :: i, id
8   real*8 :: A(iter)
9   real*8 :: tdata
10  real :: x
11
12  do i = 1, ITER
13    A(i) = 2.0 * i
14  enddo
15
16  !$omp parallel private (id, tdata, x)
17
18    id = omp_get_thread_num()
19    tdata = omp_get_wtime()
20
21    !$omp do schedule(static)
22    do i = 1, ITER
23      x = i * 1.0
24      A(i) = A(i) * sqrt(x) / (sin(x) ** tan(x))
25    enddo
26
27    tdata = omp_get_wtime() - tdata
28
29    if (id == 0) print *, "Time spent is ", tdata, " sec"
30
31  !$omp end parallel
32 end program main

```

Figure 5.7: Dynamic loop schedules that vary at runtime

A worksharing-loop with a dynamic schedule – In this program, the work per iteration is highly variable. The dynamic schedule should be much better at balancing the load across the team of threads.

```

1  program main
2
3      use omp_lib
4      implicit none
5
6  ! Use a smaller value of ITER if available memory is too small
7      integer, parameter :: ITER = 50000000
8      integer :: i, id
9      real*8 :: tdata
10     integer :: sum = 0
11
12     !$omp parallel private (i, id, tdata)
13         id = omp_get_thread_num()
14         tdata = omp_get_wtime()
15
16     !$omp do reduction (+:sum) schedule(dynamic)
17         do i = 2, ITER
18             if (check_prime(i) == 1) sum = sum + 1
19         enddo
20     !$omp end do
21
22         tdata = omp_get_wtime() - tdata
23
24         if (id == 0) print *, "Number of prime numbers is ", &
25             & sum, "in ", tdata, " sec"
26     !$omp end parallel
27
28     contains
29         integer function check_prime (num)
30             implicit none
31             integer, intent (in) :: num
32             integer :: i, iend
33

```

```
34         iend = int (sqrt(num*1.0))
35     do i = 2, iend
36         if (mod(num,i) == 0) then
37             check_prime = 0
38             return
39         endif
40     enddo
41
42     check_prime = 1
43
44     end function check_prime
45
46 end program main
```

Figure 5.8: Using `nowait` to disable implied barriers

Using a `nowait` clause with worksharing-loops – In this example, we explore the need for barriers and cases where they can be disabled with a `nowait` clause.

```
1 real*8 :: A(big), B(big), C(big)
2 integer :: id
3
4 !$omp parallel private(id)
5     id = omp_get_thread_num() + 1
6     A(id) = big_calc1(id)
7
8     !$omp barrier
9
10    !$omp do
11        do i = 1, N
12            B(i) = big_calc2(C,i)
13        end do
14    !$omp enddo nowait
15
16    A(id) = big_calc4(id)
17 !$omp end parallel
```

Figure 5.9: A particularly simple parallel Pi program

Pi program with a worksharing-loop and a reduction – The program computes the integral of a function by filling the area under a curve with rectangles and summing their areas. Loop iterations are divided among threads by the compiler under direction of the worksharing-loop construct. The reduction creates a private copy of `sum` for each thread, initializes it to zero, accumulates partial sums into the `sum` variable, and then combines partial sums to generate the global sum.

```

1          PROGRAM MAIN
2          USE OMP_LIB
3          IMPLICIT NONE
4
5          INTEGER :: i, id
6          INTEGER, PARAMETER :: num_steps=100000000
7          INTEGER :: NTHREADS = 4
8          REAL*8 :: x, pi, sum, step
9          REAL*8 :: start_time, run_time
10
11         sum = 0.0
12         step = 1.0 / num_steps
13         start_time = OMP_GET_WTIME()
14
15         CALL OMP_SET_NUM_THREADS(NTHREADS)
16
17         !$OMP PARALLEL PRIVATE(i,x)
18         !$OMP DO REDUCTION(+:sum)
19             DO i = 1, num_steps
20                 x = (i - 0.5) * step
21                 sum = sum + 4.0 / (1.0 + x * x)
22             ENDDO
23         !$OMP END DO
24         !$OMP END PARALLEL
25
26         pi = step * sum
27         run_time = OMP_GET_WTIME() - start_time
28         WRITE(*,100) pi, run_time
29 100     FORMAT('pi is ',f15.8,' in ',f8.3,' secs')
30     END PROGRAM MAIN

```

Figure 5.10: Making loop iterations independent

Loop dependence example –The first loop is sequential and contains a loop-carried dependence. The value of j for a loop index is dependent on the value of j for the previous loop index. In the parallel code in the second loop, the loop-carried dependence has been removed by calculating j from the loop control index.

```
1 ! Sequential code with loop dependence
2     integer :: i, j, A(MAX)
3     j = 5
4     do i = 1, MAX
5         j = j + 2
6         A(i) = big(j)
7     end do
8
9 ! parallel code with loop dependence removed
10    integer :: i, j, A(MAX)
11    !$omp parallel do private(j)
12    do i = 1, MAX
13        do j = 5 + 2*(i+1)
14            A(i) = big(j)
15        end do
16    end do
17 !$omp end parallel do
```


6 OpenMP Data Environment

Figure 6.1: How data is stored by default

An example of default storage attributes – `A`, `index`, `count` are shared variables since `A` is a global variable defined in a module, `index` is defined prior to the parallel region, and `count` is a saved variable. `temp` is a private variable since it is declared inside the parallel region.

```
1 ! File #1:
2 module data_mod
3     real*8 :: A(10)
4 end module data_mod
5
6 program main
7     use data_mod
8     implicit none
9     integer :: index(10)
10    !$omp parallel
11        call work(index)
12    !$omp end parallel
13    print *, index(1)
14 end program main
15
16
17 ! File #2:
18 subroutine work(index)
19     use data_mod
20     implicit none
21     integer :: index
22     real*8 :: temp(10)
23     integer, save :: count
24     ...
25 end subroutine work
```

Figure 6.3: The shared and private clauses

The shared clause – An example of a shared clause on a parallel construct. Strictly this clause is not needed. It is included in this case to remind the programmer that of the three variables A, B, and C, only B and C are shared. A copy of the variable A is created for each thread by the clause `private(A)`.

```

1  ! sample compile command to generate *.o object file :
2  !   gfortran -fopenmp -c Fig_6.3_sharedEx.f90
3
4  program sharedEx
5     use omp_lib
6     implicit none
7     integer :: A, B, C
8
9     interface
10        subroutine initialize(A, B, C)
11           integer, intent(out) :: A, B, C
12        end subroutine
13    end interface
14
15    call initialize(A, B, C)
16
17    ! remember the value of A before the parallel region
18    print *, 'A before =', A
19
20    !$omp parallel shared (B,C) private(A)
21       A = omp_get_thread_num()
22       !$omp critical
23          C = B + A
24       !$omp end critical
25    !$omp end parallel
26
27    ! A in the parallel region goes out of scope, we revert
28    ! to the original variable for A
29    print *, 'A after = ', A, ' and C = ', C
30
31 end program sharedEx

```

Figure 6.4: The Private clause (note: this program is not correct)

An example of a private clause – The original variable `tmp` is masked by the private copy of the variable inside the parallel do region. This program is incorrect since a private variable is not initialized.

```
1 ! sample compile command to generate *.o object file :
2 ! gfortran -fopenmp -c Fig_6.4_wrongPrivate.f90
3
4 program wrong
5     integer :: tmp
6     tmp = 0
7
8 !$omp parallel do private(tmp)
9     do j = 1, 1000
10         tmp = tmp +j
11     enddo
12 !$omp end parallel do
13
14     print *, tmp    ! tmp is 0 here
15 end program wrong
```

Figure 6.5: OK to use a local variable of same name outside of a module

A second example of the private clause – This Fortran program works (unlike the corresponding C code which has a subtle bug). `tmp` is a local variable in subroutine `OK`, not the one from the module file as updated in subroutine `work`, hence the value printed in line 13 should be the same as the original value 0 as defined in line 9 before the parallel region.

```
1 ! File #1
2 module data_mod
3     integer :: tmp
4 end module data_mod
5
6 subroutine OK()
7     implicit none
8     integer :: tmp
9     tmp = 0
10    !$omp parallel private(tmp)
11        call work()
12    !$omp end parallel
13    print *, tmp      ! tmp is 0, same as the original local value
14                    ! defined before the parallel region
15 end subroutine OK
16
17 ! File #2
18 subroutine work()
19     use data_mod
20     implicit none
21     tmp = 5
22 end subroutine work
```

Figure 6.6: Creating private variables that are initialized

Example of using the firstprivate clause – `incr` is a firstprivate variable so it is private to each thread and has an initial value (zero).

```
1  incr = 0
2  !$omp parallel do firstprivate(incr)
3      do i = 1, MAX
4          if (mod(i,2) == 0) incr = incr + 1
5          A(i) = incr
6      end do
7  !$omp end parallel do
```

Figure 6.7: Data environment quiz

An OpenMP data environment quiz – Consider the storage attributes and values for A, B and C.

```
1      A = 1
2      B = 1
3      C = 1
4  !$omp parallel private(B) firstprivate(C)
```

Figure 6.8 and 6.9: Find the area of the Mandelbrot set

Mandelbrot set area: original code with errors – This version of the program has multiple bugs. Your job is to inspect the code and find the bugs.

```

!C PROGRAM: Mandelbrot area
!C
!C PURPOSE: Program to compute the area of a Mandelbrot set.
!C          Correct answer should be around 1.510659.
!C          WARNING: this program may contain errors
!C
!C USAGE:   Program runs without input ... just run the executable
!C

      MODULE mandel_module
      implicit none

      INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(14)

      REAL(KIND = DP) :: r

      INTEGER, PARAMETER :: NPOINTS=1000
      INTEGER, PARAMETER :: MAXITER=1000
      INTEGER :: numoutside=0

      TYPE d_complex
         REAL(KIND = DP) :: r
         REAL(KIND = DP) :: i
      END TYPE d_complex

      TYPE(d_complex) :: c

      contains

      SUBROUTINE testpoint()

!C iterate over  $z=z*z+c$ .  $|z| > 2$  means the point is outside set

```

```

!C If loop count reaches MAXITER, point is inside the set

    implicit none
    TYPE(d_complex) :: z
    INTEGER :: iter
    REAL(KIND = DP) :: temp

    z = c

    DO iter = 1, MAXITER
        temp = (z%r*z%r) - (z%i*z%i) + c%r
        z%i = z%r*z%i*2 + c%i
        z%r = temp

        IF ((z%r*z%r + z%i*z%i) > 4.0) THEN
            numoutside = numoutside + 1
            EXIT
        ENDIF
    ENDDO

    END SUBROUTINE

END MODULE mandel_module

PROGRAM mandel_wrong
USE OMP_LIB
USE mandel_module
IMPLICIT NONE

INTEGER :: i, j
REAL(KIND = DP) :: area, error
REAL(KIND = DP) :: eps = 1.0e-5

!C Loop over grid of complex points in the domain of the
!C Mandelbrot set, testing each point to see whether it is
!C inside or outside the set.

```

```
!$OMP PARALLEL DO DEFAULT(shared) PRIVATE(c,eps)

    DO i = 1, NPOINTS
    DO j = 1, NPOINTS
        c%r = -2.0 + 2.5 * DBLE(i-1) / DBLE(NPOINTS) + eps
        c%i = 1.125 * DBLE(j-1) / DBLE(NPOINTS) + eps
        CALL testpoint()
    ENDDO
    ENDDO
!$OMP END PARALLEL DO

!C Calculate area of set and error estimate and output the results

    area = 2.0 * 2.5 * 1.125 * DBLE(NPOINTS*NPOINTS - numoutside) &
&        /DBLE(NPOINTS*NPOINTS)
    error = area / DBLE(NPOINTS)

    PRINT *, "numoutside=", numoutside
    WRITE(*,100) area, error
100  FORMAT("Area of Mandelbrot set = ", f12.8, " +/-", f12.8)
    PRINT *, "Correct answer should be around 1.510659"

    END PROGRAM mandel_wrong
```


Figures 6.10 and 6.11: Debugging the Mandelbrot set program

Mandelbrot set area solution – c is passed as an argument to subroutine `testpoint`. `eps` is declared as `firstprivate` and the inner loop index `j` is declared as `private`.

```

!C PROGRAM: Mandelbrot area
!C
!C PURPOSE: Program to compute the area of a Mandelbrot set.
!C           Correct answer should be around 1.510659.
!C
!C USAGE:   Program runs without input ... just run the executable
!C

MODULE mandel_par_module
  implicit none

  INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(14)

  INTEGER, PARAMETER :: NPOINTS=1000
  INTEGER, PARAMETER :: MAXITER=1000
  INTEGER :: numoutside=0

  TYPE d_complex
    REAL(KIND = DP) :: r
    REAL(KIND = DP) :: i
  END TYPE d_complex

  contains

  SUBROUTINE testpoint(c)

!C iterate over  $z=z*z+c$ .  $|z| > 2$  means the point is outside set
!C If loop count reaches MAXITER, point is inside the set

    implicit none
    TYPE(d_complex) :: z,c
    INTEGER :: iter

```

```

REAL(KIND = DP) :: temp

z = c

DO iter = 1, MAXITER
  temp = (z%r*z%r) - (z%i*z%i) + c%r
  z%i = z%r*z%i*2 + c%i
  z%r = temp

  IF ((z%r*z%r + z%i*z%i) > 4.0) THEN
    !$OMP CRITICAL
      numoutside = numoutside + 1
    !$OMP END CRITICAL
    EXIT
  ENDIF
ENDDO

END SUBROUTINE

END MODULE mandel_par_module

PROGRAM mandel_par

USE OMP_LIB
USE mandel_par_module
IMPLICIT NONE

INTEGER :: i, j
REAL(KIND = DP) :: area, error
REAL(KIND = DP) :: eps = 1.0e-5

TYPE(d_complex) :: c

!   CALL OMP_SET_NUM_THREADS(4)

!C Loop over grid of complex points in the domain of the

```

```
!C Mandelbrot set, testing each point to see whether it is
!C inside or outside the set.

!$OMP PARALLEL DO DEFAULT(shared) FIRSTPRIVATE(eps) PRIVATE(c,j)

    DO i = 1, NPOINTS
    DO j = 1, NPOINTS
        c%r = -2.0 + 2.5 * DBLE(i-1) / DBLE(NPOINTS) + eps
        c%i = 1.125 * DBLE(j-1) / DBLE(NPOINTS) + eps
        CALL testpoint(c)
    ENDDO
    ENDDO
!$OMP END PARALLEL DO

!C Calculate area of set and error estimate and output the results
    write(*,*)"numoutside=", numoutside

    area = 2.0*2.5*1.125 * DBLE(NPOINTS*NPOINTS - numoutside) &
&        / DBLE(NPOINTS*NPOINTS)
    error = area / DBLE(NPOINTS)

    WRITE(*,100) area, error
100  FORMAT("Area of Mandelbrot set = ", f12.8, f12.8)
    WRITE(*,*)"Correct answer should be around 1.510659"

    END PROGRAM mandel_par
```

Figure 6.12: Worksharing-loops and data environment clauses help us write particularly simple parallel Pi programs

Pi Program with combined parallel worksharing-loop and reduction –
 Each thread accumulates its local sum that is later combined into the global sum with the reduction operation. Variable `x` is declared as private with a data environment clause.

```

1          PROGRAM MAIN
2          USE OMP_LIB
3          IMPLICIT NONE
4
5          INTEGER :: i, id
6          INTEGER, PARAMETER :: num_steps=100000000
7          INTEGER :: NTHREADS = 4
8          REAL*8 x, pi, sum, step
9          REAL*8 start_time, run_time
10
11         sum = 0.0
12         step = 1.0 / num_steps
13
14         CALL OMP_SET_NUM_THREADS(NTHREADS)
15         start_time = OMP_GET_WTIME()
16
17  !$OMP PARALLEL DO PRIVATE(i,x) REDUCTION(+:sum)
18         DO i = 1, num_steps
19             x = (i - 0.5) * step
20             sum = sum + 4.0 / (1.0 + x * x)
21         ENDDO
22  !$OMP END PARALLEL DO
23
24         pi = step * sum
25         run_time = OMP_GET_WTIME() - start_time
26         WRITE(*,100) pi, run_time
27 100     FORMAT('pi is ',f15.8,' in ',f8.3,' secs')
28
29         END PROGRAM MAIN

```

Figure 6.13: Static arrays in data environment clauses

Static arrays in data environment clauses – The compiler creates a private array with 1000 values of type `int` on the stack for each thread.

```
int varray(1000)
call initv(1000, varray)    ! function to initialize the array

!$omp parallel private(varray)
    ! body of parallel region not shown
!$omp end parallel
```

Figure 6.14: Data environment clauses with dynamic arrays and pointers

Dynamic arrays and pointers in data environment clauses – The compiler gives each thread its own pointer pointing to the same block of memory.

```
real, pointer :: vptr(:)

allocate (vptr(1000))
call initv(1000, vptr)    ! function to initialize the array

!$omp parallel firstprivate(vptr)
    ! body of parallel region not shown
!$omp end parallel
```

Page 119: Array sections

You define an array section in terms of the `lower-bound` and the `upper-bound` of the section.

```
(lower-bound:upper-bound)
(:upper-bound)          ! lower-bound implied as one
```

We can use an array section in a reduction where we copy a section of an array into a private copy of that array for each thread. That private copy of the array section is used in the reduction.

```
!$omp parallel for reduction(+:vptr(1:1000))
```

7 Tasks in OpenMP

Figure 7.1: Traversing a linked list

Serial linked list program– Traverse the linked list and do a block of work (`processwork(p)`) for each node in the list where we assume `processwork(p)` for any node is independent of the other nodes.

```
1 p => head
2 do
3     call processwork(p)
4     p => p%next
5     if (.not. associated(p)) exit
6 end do
```

Figure 7.2: Traversing a linked list in parallel using worksharing-loop constructs

Parallel linked list program without using tasks – Three passes through the data to count the length of the list, collect values into an array, and process the array in parallel. This is an example of the inspector-executor design pattern.

```

1 ! sample compile command to generate *.o object file :
2 !   gfortran -fopenmp -c Fig_7.2_linkedListNoTasks.f90
3
4 module list_mod
5     integer , parameter :: NMAX = 10
6     type :: node
7         integer :: data
8         integer :: procResult
9         type(node), pointer :: next
10    end type node
11 end module list_mod
12
13 program main
14     use list_mod
15     implicit none
16
17     type(node), pointer :: p => null()
18     type(node), pointer :: temp => null()
19     type(node), pointer :: head => null()
20     type(node), dimension(:), allocatable , target :: parr
21
22     interface
23     ! initialize the list (not shown)
24     subroutine initList(p)
25         use list_mod
26         implicit none
27         type(node), pointer :: p
28     end subroutine initList
29
30     ! a long computation (not shown)
31     integer function work(data)
32         implicit none

```



```
33         integer :: data
34     end function work
35 end interface
36
37 integer :: i, count
38
39 call initList(p)
40
41 ! save head of the list
42 head => p
43
44 count = 0
45 do
46     p = p%next
47     count = count + 1
48     if (.not. associated(p)) exit
49 end do
50
51 allocate(parr(count))
52
53 p => head
54 do i = 1, count
55     parr(i)%data = p%data
56     p => p%next
57 end do
58
59 !$omp parallel do schedule(static,1)
60 do i = 1, count
61     call procWork(parr(i))
62 end do
63 !$omp end parallel do
64
65 contains
66
67 subroutine procWork (a_node)
68     use list_mod
69     implicit none
70     type(node) :: a_node
71     integer :: n
```

```
72     integer , external :: work
73     n = a_node%data
74     a_node%procResult = work(n)
75 end subroutine procWork
76
77 end program main
```

Figure 7.4: A simple program with OpenMP tasks

Schrödinger’s Program – Two threads each generates two tasks. They wait a random bit of time and then set a shared variable to true or false. Whichever task executes last determines the final value of the variable and whether the cat is “dead” or “alive”.

```

1  ! Schrodingers racy program ... is the cat dead or alive?
2  !
3  ! You can use atomics and make the program race free , or comment out
4  ! the atomics and run with a race condition . It works in both cases
5  !
6  ! History: Written by Tim Mattson , Feb 2019
7  !           Converted to Fortran by Helen He, Nov 2019
8
9
10 program main
11     use omp_lib
12     implicit none
13
14     ! random number generator parameters
15     ! (from numerical recipies)
16     integer , parameter :: MULT = 4096
17     integer , parameter :: ADD = 150889
18     integer , parameter :: MOD_val = 714025
19
20     real*8 :: wait_val , val
21     integer*8 :: rand , i , dcount , lcount , coin
22     logical :: dead_or_alive , HorT
23     integer , parameter :: NTRIALS = 10
24
25     dcount = 0
26     lcount = 0
27
28     do i = 1, NTRIALS
29         !$omp parallel num_threads(2) shared(dead_or_alive) private(val)
30             if(omp_get_thread_num() == 0) then
31                 print *, " with ", omp_get_num_threads(), " threads."
32                 write(*, '(a)', advance='no') " Schrodingers program says the cat is "
33             endif

```

```

34
35     !$omp single
36         ! "flip a coin" to choose which task is for the dead
37         ! cat and which for the living cat.
38         call seedIt(coin)
39         HorT = flip(coin)
40
41         ! without the atomics, these tasks are participating in a data race
42     !$omp task
43         val = waitAbit()
44         ! a store of a single machine word (bool)
45     !$omp atomic write
46         dead_or_alive = HorT
47     !$omp end atomic
48 !$omp end task
49 !$omp task
50     val = waitAbit()
51     ! a store of a single machine word (bool)
52 !$omp atomic write
53     dead_or_alive = .not. HorT
54 !$omp end atomic
55 !$omp end task
56 !$omp end single
57 !$omp end parallel
58
59 if (dead_or_alive) then
60     print *, " alive."
61     lcount = lcount + 1
62 else
63     print *, " dead."
64     dcount = dcount + 1
65 endif
66 end do    ! end loop over trials (for testing only)
67
68 print *, " dead ", dcount, " times", " and alive ", lcount, " times."
69
70 contains
71
72 ! seed the pseudo random sequence with time of day

```

```
73     subroutine seedIt(val)
74         implicit none
75         integer*8 :: val
76         val = int (omp_get_wtime())
77     end subroutine seedIt
78
79 ! Linear congruential random number generator
80     integer*8 function nextRan(last) result(next)
81         implicit none
82         integer*8, intent(in) :: last
83         next = mod(MULT*last+ADD, MOD_val)
84     end function nextRan
85
86
87 ! flip a coin ... heads (true) or tails (false)
88     logical function flip(coin)
89         implicit none
90         integer*8 :: coin
91         coin = nextRan(coin)
92         if (coin > MOD_val/2) then
93             flip = .true.
94         else
95             flip = .false.
96         endif
97     end function flip
98
99 ! wait a short random amount of time
100     real*8 function waitAbit() result(val)
101         implicit none
102         integer*8 :: i, count, rand
103         val = 0.0
104         call seedIt(rand)
105         count = nextRan(rand)
106
107         ! do some math to make us wait a while
108         do i = 1, count
109             rand = nextRan(rand)
110             val = val + dble(rand)/dble(MULT)
111         end do
```

```
112     end function waitAbit
113
114 end program main
```

Figure 7.5: Single worksharing construct

An OpenMP single construct example – All threads execute *do_many_things* and *do_many_other_things*, but only one thread executes *exchange_boundaries*.

```
!$omp parallel
  call do_many_things()
  !$omp single
    call exchange_boundaries()
  !$omp end single
  call do_many_other_things()
!$omp end parallel
```

Figure 7.6: Creating explicit tasks

A basic task example – Inside a parallel region, 3 tasks are created by a single thread.

```
1 !$omp parallel
2   !$omp single
3     !$omp task
4       call fred()
5     !$omp end task
6     !$omp task
7       call daisy()
8     !$omp end task
9     !$omp task
10      call billy()
11     !$omp end task
12   !$omp end single
13 !$omp end parallel
```

Figure 7.7: waiting for tasks to finish with taskwait

A taskwait example – Tasks *fred* and *daisy* must complete before task *billy* starts.

```

1 !$omp parallel
2     !$omp single
3         !$omp task
4             call fred()
5         !$omp end task
6         !$omp task
7             call daisy()
8         !$omp end task
9         !$omp taskwait
10        !$omp task
11            call billy()
12        !$omp end task
13    !$omp end single
14 !$omp end parallel

```

Figure 7.8: How data moves between tasks

Tasks data environment example – *A* is shared, *B* is firstprivate, and *C* is private.

```

1 integer :: C
2 !$omp parallel shared(A) private(B)
3     ...
4     !$omp task private(C)
5         call compute(A, B, C)
6     !$omp end task
7 !$omp end parallel

```


Figure 7.9: Tasks make parallel linked list traversal really simple

Linked list with tasks – The implementation with OpenMP tasks is much more elegant than the three-pass solution in Figure 7.2.

```
1 !$omp parallel
2     !$omp single
3         p => head
4         do
5             !$omp task firstprivate(p)
6                 call processwork(p)
7             !$omp end task
8             p => p%next
9             if (.not. associated(p)) exit
10        end do
11    !$omp end single
12 !$omp end parallel
```

Figure 7.10: A really inefficient way to compute Fibonacci numbers

Fibonacci example – This is the serial recursive implementation.

```

1 recursive integer function fib (n) result(res)
2     implicit none
3     integer, intent(in) :: n
4     integer :: x, y
5     if (n < 2) then
6         res = n
7     else
8         x = fib(n-1)
9         y = fib(n-2)
10        res = x + y
11    endif
12 end function fib
13
14 program main
15     implicit none
16
17     interface
18         function fib (n)
19             integer :: fib
20             integer, intent(in) :: n
21         end function fib
22     end interface
23
24     integer :: NW, result
25     NW = 30
26     result = fib(NW)
27     print *, "fib(",NW,")=", result
28 end program main

```

Figure 7.11: Parallel Fibonacci program using the divide and conquer pattern**Parallel implementation of the Fibonacci program using OpenMP tasks**

– Two tasks create child tasks recursively. `taskwait` ensures the direct child tasks complete before the merge. The base case to exit the recursion is defined for when `n < 2`.
code:fibonacciTasks

```
1 recursive integer function fib (n) result(res)
2   use omp_lib
3   implicit none
4
5   integer, intent(in) :: n
6   integer :: x, y
7   if (n < 2) then
8     res = n
9   else
10    !$omp task shared (x)
11     x = fib(n-1)
12    !$omp end task
13    !$omp task shared (y)
14     y = fib(n-2)
15    !$omp end task
16    !$omp taskwait
17    res = x + y
18  endif
19 end function fib
20
21 program main
22   use omp_lib
23   implicit none
24
25   interface
26     function fib (n)
27       integer :: fib
28       integer, intent(in) :: n
29     end function fib
30   end interface
31
```

```
32     integer :: NW, result
33     NW = 30
34     !$omp parallel
35         !$omp single
36             result = fib(NW)
37         !$omp end single
38     !$omp end parallel
39     print *, "fib(",NW,")=", result
40 end program main
```

Figure 7.13: The Pi program (from Figure 4.5)

Serial Pi program to numerically estimate a definite integral using the midpoint rule – The loop iterations are independent other than the summation into sum.

```

1          PROGRAM MAIN
2
3          USE OMP_LIB
4          IMPLICIT NONE
5
6          INTEGER :: i
7          INTEGER, PARAMETER :: num_steps = 100000000
8          REAL*8 :: x, pi, sum, step
9          REAL*8 :: start_time, run_time
10
11         sum = 0.0
12
13         step = 1.0/num_steps
14         start_time = OMP_GET_WTIME()
15
16         DO i = 1, num_steps
17             x = (i - 0.5) * step
18             sum = sum + 4.0 / (1.0 + x * x)
19         ENDDO
20
21         pi = step * sum
22         run_time = OMP_GET_WTIME() - start_time
23
24         WRITE(*,100) pi, num_steps, run_time
25 100    FORMAT('pi = ', f15.8, ', ', i14, ' steps ', f8.3, ' secs ')
26
27         END PROGRAM MAIN

```

Figure 7.14: Serial recursive Pi program

Serial Pi program using the divide and conquer pattern –Just to make the code simpler, we pick a number of steps that is a power of 2. This way we can split the number of steps in half repeatedly and always create intervals that are divisible by 2.

```

1 module data_mod
2     integer , parameter :: num_steps = 1024*1024*1024
3     integer , parameter :: MIN_BLK = 1024*256
4
5     contains
6         real*8 recursive function pi_comp(Nstart , Nfinish , step) result(sum)
7             implicit none
8
9             integer , intent(in) :: Nstart , Nfinish
10            real*8 :: x, sum1, sum2, step
11            integer :: i, iblk
12
13            sum = 0.0
14
15            if (Nfinish - Nstart < MIN_BLK) then
16                do i = Nstart , Nfinish - 1
17                    x = (i + 0.5) * step
18                    sum = sum + 4.0 / (1.0 + x * x)
19                enddo
20            else
21                iblk = Nfinish - Nstart
22                sum1 = pi_comp(Nstart , Nfinish - iblk/2, step)
23                sum2 = pi_comp(Nfinish - iblk/2, Nfinish , step)
24                sum = sum1 + sum2
25            endif
26
27        end function
28
29 end module data_mod
30
31 program main
32     use data_mod
33     implicit none

```

```
34
35     integer :: i
36     real*8 :: pi, sum, step
37
38     step = 1.0 / num_steps
39     sum = pi_comp(0, num_steps, step)
40     pi = step * sum
41
42     WRITE(*,100) pi, num_steps
43 100     FORMAT('pi = ', f15.8, ', ', i14, ' steps ')
44
45 end program main
```

Figure 7.15: Parallel recursive Pi program

Parallel Pi program using tasks – It is accomplished with the divide and conquer pattern by splitting the problem into two subtasks to calculate *sum1* and *sum2*, recursively solving each task, and then combining the results.

```

1 module data_mod
2     integer , parameter :: num_steps = 1024*1024*1024
3     integer , parameter :: MIN_BLK = 1024*256
4
5     contains
6         real*8 recursive function pi_comp(Nstart , Nfinish , step) result(sum)
7             use omp_lib
8             implicit none
9
10            integer , intent(in) :: Nstart , Nfinish
11            real*8 :: x, sum1, sum2, step
12            integer :: i, iblk
13
14            sum = 0.0
15
16            if (Nfinish - Nstart < MIN_BLK) then
17                do i = Nstart , Nfinish - 1
18                    x = (i + 0.5) * step
19                    sum = sum + 4.0 / (1.0 + x * x)
20                enddo
21            else
22                iblk = Nfinish - Nstart
23                !$omp task shared(sum1)
24                sum1 = pi_comp(Nstart , Nfinish - iblk/2, step)
25                !$omp end task
26                !$omp task shared(sum2)
27                sum2 = pi_comp(Nfinish - iblk/2, Nfinish , step)
28                !$omp end task
29                !$omp taskwait
30                sum = sum1 + sum2
31            endif
32
33        end function

```



```
34
35 end module data_mod
36
37 program main
38     use omp_lib
39     use data_mod
40     implicit none
41
42     integer :: i
43     real*8 :: pi, sum, step
44
45     step = 1.0 / num_steps
46
47     !$omp parallel
48         !$omp single
49         sum = pi_comp(0, num_steps, step);
50         !$omp end single
51     !$omp end parallel
52
53     pi = step * sum
54
55     WRITE(*,100) pi, num_steps
56 100     FORMAT('pi = ', f15.8, ', ', i14, ' steps ')
57
58 end program main
```


8 OpenMP Memory Model

Figure 8.2: Relaxed memory models and race conditions

A program with a race condition – A relaxed memory model permits the assertion to fail; i.e., the thread with `id == 1` can observe values in memory such that `r == 1` while `x` is still 0.

```
1 program main
2   use omp_lib
3   implicit none
4
5   integer :: x, y, r
6   integer :: id, nthrds
7   x = 0
8   y = 0
9   r = 0
10  call omp_set_num_threads(2) ! request two threads
11  !$omp parallel private(id, nthrds)
12     id = omp_get_thread_num()
13     !$omp single
14         nthrds = omp_get_num_threads()
15         ! verify that we have at least two threads
16         if (nthrds < 2) stop 1
17     !$omp end single
18
19     if (id == 0) then
20         x = 1
21         r = x
22     else if (id == 1) then
23         if (r == 1) then
24             y = x;
25             if (y /= 1) then
26                 stop "fails y==1" ! Assertion will occasionally fail;
27                                     ! i.e., r = 1 while x = 0
28             endif
29         endif
30     endif
31  !$omp end parallel
32 end program main
```

Figure 8.3: Updates may not be fully shared

An erroneous program where updates may not be fully shared – This program carries out an iterative computation over the elements of an array *A*. Assume the function `doit()` carries out a computation that is embarrassingly parallel with a fixed subset of the array *A* selected by the thread ID. This program could fall into an infinite loop if the value of `conv` does not issue the break from the while loop and the shared variable `iter` is not propagated across all the threads allowing it to trigger the loop exit condition (`iter < MAX`).

```

1
2 ! sample compile command:
3 !       gfortran -fopenmp -c Fig_8.3_regPromote.f90
4 ! to generate *.o object file
5
6 program main
7     use omp_lib
8     implicit none
9
10    interface
11        function doit(A, N, id)
12            integer :: N, id
13            real*8 :: A(N)
14            real*8 :: doit
15        end function
16    end interface
17
18    integer, parameter :: MAX = 10000
19    integer, parameter :: NMAX = 1000
20    real, parameter :: TOL = 0.0001
21
22    integer :: iter, N
23    real*8 :: A(NMAX)
24    real*8 :: conv
25    integer :: id, nthrd
26
27    iter = 0
28    N = 1000
29    A = 0.0

```

```
30     conv = 0.0
31
32     !$omp parallel shared(A,N,iter) firstprivate(conv) private(id,nthrd)
33         id = omp_get_thread_num()
34         nthrd = omp_get_num_threads()
35
36         do while (iter < MAX)
37             conv = doit(A, N, id)
38             if (conv < TOL) exit
39             if (id == 0) iter = iter + 1
40         end do
41
42     !$omp end parallel
43
44 end program main
```

Figure 8.4: Races due to nowait

Reductions need a barrier – This program carries out a computation inside a parallel loop and accumulates the result with a reduction. The function called after the loop uses the SPMD pattern and does not use any of the values computed in the loop, hence the programmer used a `nowait` clause. The last function uses the reduction variable which may not be available for all threads since the reduction is only guaranteed to complete at the next barrier following the loop. As a result, this is an incorrect program.

```
1 integer :: id, nthrds, i
2 !$omp parallel shared(A, B, sum) private(id, nthrds)
3     id = omp_get_thread_num()
4     nthrds = omp_get_num_threads()
5
6     !$omp do reduction(+:sum)
7         do i = 1, N
8             sum = sum + big_job(A,N)
9         end do
10    !$omp end do nowait
11
12    bigger_job(B, id)      ! a function that does not use A
13    another_job(sum, id)  ! sum may not be available
14 !$omp end parallel
```

Figure 8.5: Synchronization in producer-consumer programs

Pairwise synchronization – A producer-consumer pattern with one thread producing a result that another thread will consume. This program uses a spin-lock to make the consumer wait for the producer to finish. *Note: This program is not properly synchronized and as written will not work.*

```
1 integer :: flag    ! flag to communicate when consumer can start
2 integer :: id , nthrds
3 flag = 0
4
5 !$omp parallel shared(A, flag) private(id , nthrds)
6     id = omp_get_thread_num()
7     nthrds = omp_get_num_threads()
8
9     ! we need two or more threads for this program
10    if ((id == 0) .and. (nthrds < 2)) stop 1
11
12    if (id == 0) then
13        call produce(A)
14        flag = 1
15    endif
16    if (id == 1) then
17        do while (flag == 0)
18            ! spin through the loop waiting for flag to change
19        enddo
20        call consume (A)
21    endif
22 !$omp end parallel
```


9 Common Core Recap

This chapter provides a summary of the items from OpenMP that make up the Common Core. Hence, there are not any example programs in this chapter.

10 Multithreading beyond the Common Core

Page 176: Additional clauses for the Parallel construct

```
integer :: nthreads
nthreads = omp_get_num_threads()

!$omp parallel if (nthreads < maxthreads/4) num_threads(4)
    ...
!$omp end parallel
```

Figure 10.1: Clauses on parallel constructs

Examples of clauses on the Parallel construct – The matrix **A** is transformed by a transformation which is assumed to be a unitary transform (i.e., a trace preserving transform). Notice how continuation of a directive onto an additional line is indicated by an ampersand (&). We do not show code for `initMats()` and `transform()` as their function bodies are not relevant for this example.

```

1 ! sample command to compile to object file:
2 !      gfortran -fopenmp -c Fig_10.1_parClaw.f90
3
4 program main
5     use omp_lib
6
7     interface
8         ! initialization function
9         subroutine initMats(N, A, T)
10            integer :: N
11            real, dimension(:, :), allocatable :: A, T
12        end subroutine
13        ! transform function
14        subroutine transform(N, id, Nthrds, A, T)
15            integer :: N, id, Nthrds
16            real, dimension(:, :), allocatable :: A, T
17        end subroutine
18    end interface
19
20    real :: trace = 0
21    integer :: i, id, N, Nthrds
22    real, dimension(:, :), allocatable :: A, T
23
24    integer :: narg      ! number of Arg
25    character(len=10) :: name ! Arg name
26
27    narg = command_argument_count()
28    if (narg == 1) then
29        call get_command_argument(1, name)
30        read(name, *) N
31    else

```

```

32     N = 10
33   endif
34   print *, "N=", N
35
36   ! allocate space for two N x N matrices and initialize them
37   allocate (T(N,N))
38   allocate (A(N,N))
39   call initMats(N, A, T)
40
41   !$omp parallel if(N>100) num_threads(4) default(none) &
42   !$omp&          shared(A,T,N) private (i,id,Nthrds) reduction(+:trace)
43     id = omp_get_thread_num()
44     Nthrds = omp_get_num_threads()
45     call transform(N, id, Nthrds, A, T)
46
47     ! compute trace of A matrix
48     ! i.e., the sum of diagonal elements
49     !$omp do
50       do i = 1, N
51         trace = trace + A(i,i)
52       enddo
53     !$omp end do
54   !$omp end parallel
55   print *, " transform complete with trace = ", trace
56 end program main

```

Page 180: The lastprivate clause

```

integer :: i

!$omp do lastprivate(ierr)
  do i = 1, N
    ierr = work(i)
  enddo
!$omp end do

```

Figure 10.3: Manipulating schedules for worksharing-loops at runtime

Use of runtime schedules – Function computes forces in a simple molecular dynamics program. Prints information about the runtime schedule when enabled by the DEBUG variable. Notice how we do line continuation for an OpenMP compiler directive in our parallel construct.

```

1 ! sample compile command to generate *.o object file
2 !   gfortran -fopenmp -c Fig_10.3_runtimeEx.f90
3
4 subroutine forces(npart, x, f, side, rcoeff)
5   use omp_lib
6   implicit none
7
8   interface
9     ! external function for potential energy term
10    function pot (dist) result(res)
11      real*8 :: dist
12      real*8 :: res
13    end function pot
14  end interface
15
16  integer(kind=omp_sched_kind) :: kind
17  integer :: chunk_size
18  logical :: DEBUG
19  integer :: npart, i, j
20  real*8 :: x(0:npart*3+2), f(0:npart*3+2)
21  real*8 :: side, rcoeff
22  real*8 :: fxi, fyi, fzi
23  real*8 :: xx, yy, zz, rd, fcomp
24
25  character (len=:), allocatable :: schdKind(:)
26  allocate (character(8) :: schdKind(0:4))
27  ! map schedule kind enum values to strings for printing
28
29  schdKind(0) = "ERR"
30  schdKind(1) = "static"
31  schdKind(2) = "dynamic"

```

```

32     schdKind(3) = "guided"
33     schdKind(4) = "auto"
34     DEBUG = .true.
35
36     !$omp parallel do schedule(runtime) &
37     !$omp private (fxi , fyi , fzi , j , xx , yy , zz , rd , fcomp)
38
39         do i = 0, npart*3-1, 3
40             ! zero force components on particle i
41             fxi = 0.0
42             fyi = 0.0
43             fzi = 0.0
44
45             ! loop over all particles with index > i
46             do j = i+3, npart*3-1, 3
47
48                 ! compute distance between i and j with wraparound
49                 xx = x(i) - x(j)
50                 yy = x(i+1) - x(j+1)
51                 zz = x(i+2) - x(j+2)
52
53                 if (xx < (-0.5*side)) xx = xx + side
54                 if (xx > (0.5*side)) xx = xx - side
55                 if (yy < (-0.5*side)) yy = yy + side
56                 if (yy > (0.5*side)) yy = yy - side
57                 if (zz < (-0.5*side)) zz = zz + side
58                 if (zz > (0.5*side)) zz = zz - side
59                 rd = xx * xx + yy * yy + zz * zz
60
61                 ! if distance is inside cutoff radius , compute forces
62                 if (rd <= rcoff*rcoff) then
63                     fcomp = pot(rd)
64                     fxi = fxi + xx*fcomp
65                     fyi = fyi + yy*fcomp
66                     fzi = fzi + zz*fcomp
67                     !$OMP critical
68                         f(j) = f(j) - xx*fcomp
69                         f(j+1) = f(j+1) - yy*fcomp
70                         f(j+2) = f(j+2) - zz*fcomp

```

```
71             !$OMP end critical
72         endif
73     enddo
74     ! update forces on particle i
75     f(i) = f(i) + fxi
76     f(i+1) = f(i+1) + fyi
77     f(i+2) = f(i+2) + fzi
78     enddo
79 !$omp end parallel do
80
81     if (DEBUG) then
82         call omp_get_schedule(kind, chunk_size)
83         print *, "schedule ", schdKind(kind), " chunk_size=", chunk_size
84     endif
85 end subroutine forces
```


Figure 10.4: Combining loops to generate one large worksharing-loop

The collapse clause on a worksharing-loop construct – The `Apply` function applies a function input as a function pointer to each element of an N by M array, `A`. Note that the pointer expression `(A+i*M+j)` points to the (i,j) element of the array `A`.

```

1 ! sample compile command to generate *.o object file
2 !           gfortran -fopenmp -c Fig_10.4_loopCollapse.f90
3
4  subroutine Apply(N, M, A, MFUNC)
5      use omp_lib
6      implicit none
7
8      integer :: N, M
9      real :: A(N,M)
10     integer :: i, j
11
12     interface
13         subroutine MFUNC (i,j,x)
14             integer, intent(in) :: i, j
15             real :: x
16         end subroutine MFUNC
17     end interface
18
19     ! apply a function MFUNC to each element of an N by M array
20
21     !$omp parallel do num_threads(4) collapse(2) if(N*M>100)
22         do i = 1, N
23             do j = 1, M
24                 call MFUNC(i, j, A(i,j))
25             enddo
26         enddo
27     !$omp end parallel do
28 end subroutine

```

Figure 10.6: Creating a DAG of tasks

Task Dependencies – This program implements the DAG (Directed Acyclic Graph) shown in Figure 10.5. The functions represent the nodes and the edges of the DAG are captured by the patterns of `depend` clauses.

```

1  ! sample compile command to generate *.o object file
2  !           gfortran -fopenmp -c Fig_10.6_taskDep.f90
3
4  program main
5     use omp_lib
6     implicit none
7     external :: AWork, BWork, Cwork, Dwork, Ework
8     real :: A, B, C, D, E
9
10    !$omp parallel shared(A, B, C, D, E)
11       !$omp single
12          !$omp task depend(out:A)
13             call Awork(A)
14          !$omp end task
15          !$omp task depend(out:E)
16             call Ework(E)
17          !$omp end task
18          !$omp task depend(in:A) depend(out:B)
19             call Bwork(B)
20          !$omp end task
21          !$omp task depend(in:A) depend(out:C)
22             call Cwork(C)
23          !$omp end task
24          !$omp task depend(in:B,C,E)
25             call Dwork(E)
26          !$omp end task
27       !$omp end single
28    !$omp end parallel
29 end program main

```

Figure 10.7: Threadprivate variables to make variables private to a thread but global inside a thread

Counting task executions with a threadprivate counter – This program traverses a linked list in parallel with tasks doing a random amount of work for each node in the list. A threadprivate variable is used to keep track of how many tasks were executed by each thread. Note: we do not provide the functions used for the list nor the list processing.

```

1 ! sample compile command to generate *.o object file
2 !      gfortran -fopenmp -c Fig_10.7_threadpriv.f90
3
4 module data_mod
5     implicit none
6     integer :: counter
7     !$omp threadprivate(counter)
8     type node
9         integer :: data
10        type(node), pointer :: next
11    end type node
12    contains
13        subroutine init_list(p)
14            type (node), pointer :: p
15            ! init list here
16        end subroutine
17        subroutine processwork(p)
18            type (node), pointer :: p
19            ! proces work here
20        end subroutine
21        subroutine freeList(p)
22            type (node), pointer :: p
23            ! free list here
24        end subroutine
25        subroutine inc_count()
26            counter = counter + 1
27        end subroutine
28 end module data_mod
29
30 program main

```

```
31 use omp_lib
32 use data_mod
33 implicit none
34
35 type(node), pointer :: head
36 type(node), pointer :: p
37 counter = 0
38 call init_list(p)
39 head => p
40
41 !$omp parallel private(p) copyin(counter)
42     !$omp single
43         p => head
44         do
45             !$omp task firstprivate(p)
46                 call inc_count()
47                 call processwork(p)
48             !$omp end task
49             p => p%next
50             if (.not. associated(p)) exit
51         end do
52     !$omp end single
53 !$omp end parallel
54
55 call freeList(p)
56 end program main
```

Figure 10.8: Threadprivate variables and Fortran common blocks

A counter with threadprivate in Fortran – This code come from the OpenMP 4.5 Examples document (threadprivate.1.f). This Fortran function creates a global scope variable in Fortran through common blocks. Hence, the counter is placed in a named common block and that block is made threadprivate.

```
1      INTEGER FUNCTION INCREMENT.COUNTER()  
2      COMMON/INC.COMMON/COUNTER  
3  !$OMP THREADPRIVATE(/INC.COMMON/)  
4      COUNTER = COUNTER +1  
5      INCREMENT.COUNTER = COUNTER  
6      RETURN  
7      END FUNCTION INCREMENT.COUNTER
```


11 Synchronization and the OpenMP Memory Model

Figure 11.1: Sequence points from C expressed in Fortran

Examples of sequence points – This code shows the most common sequence points and the relations sequenced-before, indeterminately sequenced, and unsequenced.

```
1 ! sample compile command to generate *.o object file :
2 !           gfortran -fopenmp -c Fig_11.1_seqPts.f90
3 program main
4     implicit none
5     integer :: a , b, c, d, e
6     integer :: i, N = 100
7     integer , external :: func1 , func2 , func3
8
9     ! Each semicolon defines a sequence point ...
10    ! all ordered by sequenced-before relations.
11
12    a = 1;  b = 2; c = 0
13
14    ! 3 sequence points: the full statement plus the 2 function calls.
15    ! The + operator is not a sequence point so the function calls
16    ! are unordered and therefore , indeterminately sequenced.
17
18    d = func2(a) + func3(b)
19
20    ! each expression in the for statement is a sequence point.
21    ! they occur in a sequenced-before relation.
22
23    do i = 1, N
24        ! function invocations are each a sequence point. Argument
25        ! evaluations are unordered or indeterminately sequenced.
26        e = func1(func2(a), func3(b))
27    enddo
28
29    ! There is no Fortran increment syntax such as a++ in C.
30    ! a = a + 1 evaluates a + 1 first , then store the new value in a.
31
32    a = a + 1
33 end program main
```

Figure 11.3: Producer-consumer program with data races

Pairwise synchronization with incorrect synchronization – A producer consumer pattern with one thread producing a result that another thread will consume. This program uses a spin-lock to make the consumer wait for the producer to finish. Note: While the logic in this program is correct, it contains a data race. Hence it is not a valid OpenMP program and as written will not work.

```
1 integer :: flag    ! flag to signal when the consumer can start
2 integer :: id , nthrds
3 flag = 0
4 call omp_set_num_threads(2)
5
6 !$omp parallel shared(A, flag) private(id , nthrds)
7     id = omp_get_thread_num()
8     nthrds = omp_get_num_threads()
9
10    ! we need two or more threads for this program
11    if ((id == 0) .and. (nthrds < 2)) stop 1
12
13    if (id == 0) then
14        call produce(A)
15        flag = 1
16    endif
17    if (id == 1) then
18        do while (flag == 0)
19            ! spin through the loop waiting for flag to change
20        enddo
21        call consume(A)
22    endif
23 !$omp end parallel
```


Figure 11.4: Spin locks and flushes

Pairwise synchronization with flushes – A producer consumer program with a spin lock and explicit flushes. **This code is incorrect** since the operations on the flag define a data race.

```
1 integer :: flag      ! flag to signal when the consumer can start
2 integer :: id, nthrds
3 flag = 0
4 call omp_set_num_threads(2)
5
6 !$omp parallel shared(A, flag) private(id, nthrds)
7     id = omp_get_thread_num()
8     nthrds = omp_get_num_threads()
9
10    ! we need two or more threads for this program
11    if ((id == 0) .and. (nthrds < 2)) stop 1
12
13    if (id == 0) then
14        call produce(A)
15        !$omp flush
16        flag = 1
17        !$omp flush (flag)
18    endif
19    if (id == 1) then
20        !$omp flush (flag)
21        do while (flag == 0)
22            !$omp flush (flag)
23        enddo
24        !$omp flush
25        call consume(A)
26    endif
27 !$omp end parallel
```

Figure 11.5: Spin-lock and flushes with atomics

Pairwise synchronization with flushes and atomics – A producer consumer program with a spin lock and explicit flushes. With the use of atomics to update and then read `flag`, this program is race free on any processor.

```

1 integer :: flag, flag_temp    ! flag to signal when the consumer can start
2 integer :: id, nthrds
3 flag = 0
4 call omp_set_num_threads(2)
5
6 !$omp parallel shared(A, B, flag) private(id, nthrds, flag_temp)
7     id = omp_get_thread_num()
8     nthrds = omp_get_num_threads()
9
10    ! we need two or more threads for this program
11    if ((id == 0) .and. (nthrds < 2)) stop 1
12
13    if (id == 0) then
14        call produce(A)
15        !$omp flush
16        !$omp atomic write
17            flag = 1
18        !$omp end atomic
19    endif
20    if (id == 1) then
21        do while (flag_temp /= 0)
22            !$omp atomic read
23                flag_temp = flag
24            !$omp end atomic
25        enddo
26        !$omp flush
27        call consume(A)
28    endif
29 !$omp end parallel

```

Figure 11.6: Synchronization mapped onto the elements of a data structure

Locks to protect updates to a histogram – Generate a sequence of pseudorandom numbers and assigns them to a histogram.

```

1  ! sample compile command to generate *.o object file
2  !           gfortran -fopenmp -c Fig_11.6_hist.f90
3
4  program main
5      use omp_lib
6      implicit none
7
8      integer, parameter :: num_trials = 1000000 ! number of x values
9      integer, parameter :: num_bins = 100      ! number of bins in histogram
10     real*8, save :: xlow = 0.0; ! low end of x range
11     real*8, save :: xhi = 10.0; ! high end of x range
12
13     real*8 :: x
14     integer*8 :: hist(num_bins) ! the histogram
15     integer*8 :: ival, i
16     real*8 :: bin_width ! the width of each bin in the histogram
17     real*8 :: sumh, sumhsq, ave, std_dev
18     ! hist_lcks is an array of locks, one per bucket
19     integer(kind=omp_lock_kind) :: hist_lcks(num_bins)
20
21
22     interface
23         function drandom() result(val)
24             real*8 :: val
25         end function
26         subroutine seed(low_in, hi_in)
27             real*8, intent(in) :: low_in, hi_in
28         end subroutine
29     end interface
30
31     call seed(xlow, xhi) ! seed random generator over range of x
32     bin_width = (xhi - xlow) / dble(num_bins)
33

```

```

34     ! initialize the histogram and the array of locks
35     !$omp parallel do schedule(static)
36         do i = 1, num_bins
37             hist(i) = 0
38             call omp_init_lock(hist_lcks(i))
39         enddo
40     !$omp end parallel do
41     ! test uniform pseudorandom sequence by assigning values
42     ! to the right histogram bin
43     !$omp parallel do schedule(static) private(x,ival)
44         do i = 1, num_trials
45             x = drandom()
46             ival = int8((x - xlow)/bin_width)
47             ! protect histogram bins.
48             ! Low overhead due to uncontended locks
49             call omp_set_lock(hist_lcks(ival))
50             hist(ival) = hist(ival) + 1
51             call omp_unset_lock(hist_lcks(ival))
52         enddo
53     !$omp end parallel do
54
55     sumh = 0.0
56     sumhsq = 0.0
57     ! compute statistics (ave, std_dev) and destroy locks
58     !$omp parallel do schedule(static) reduction(+:sumh,sumhsq)
59         do i = 1, num_bins
60             sumh = sumh + hist(i)
61             sumhsq = sumhsq + hist(i)*hist(i)
62             call omp_destroy_lock(hist_lcks(i))
63         enddo
64     !$omp end parallel do
65
66     ave = sumh / dble(num_bins)
67     std_dev = sqrt(sumhsq /dble(num_bins) - ave * ave)
68 end program main

```

Figure 11.7: Atomics make our producer-consumer program much simpler

Pairwise synchronization with sequentially consistent atomics – A producer consumer program but now the form of atomic construct used implies all the flushes we need.

```

1 integer :: flag, temp_flag ! flag to signal when the consumer can start
2 integer :: id, nthrd
3 flag = 0
4
5 call omp_set_num_threads(2)
6
7 !$omp parallel shared(A, flag) private(id, nthrd, temp_flag)
8     id = omp_get_thread_num()
9     nthrds = omp_get_num_threads()
10
11     ! we need two or more threads for this program
12     if ((id == 0) .and. (nthrds < 2)) stop -1
13
14     if (id == 0) then
15         call produce(A)
16         !$omp atomic write seq_cst
17             flag = 1
18         !$omp end atomic
19     endif
20
21     if (id == 1) then
22         do while(1)
23             !$omp atomic read seq_cst
24                 flag_temp = flag
25             if (flag_temp /= 0) exit
26         enddo
27         call consume(A)
28     endif
29 !$omp end parallel

```


12 Beyond OpenMP Common Core Hardware

Figure 12.6: First touch and reducing memory movement costs

STREAM initialization with and without first touch – Without first touch:
step 1.a + step 2. With first touch: step 1.b + step 2.

```
1 ! Step 1.a Initialization by initial thread only
2   do j = 1, VectorSize
3     a(j) = 1.0
4     b(j) = 2.0
5     c(j) = 0.0
6   enddo
7
8 ! Step 1.b Initialization by all threads (first touch)
9   call omp_set_dynamic(0)
10  !$omp parallel do schedule(static)
11    do j = 1, VectorSize
12      a(j) = 1.0
13      b(j) = 2.0
14      c(j) = 0.0
15    enddo
16  !$omp end parallel do
17
18 ! Step 2 Compute
19 !$omp parallel do schedule(static)
20   do j = 1, VectorSize
21     a(j) = b(j) + d * c(j)
22   enddo
23 !$omp end parallel do
```

Figure 12.9: Nested parallelism

Nested OpenMP parallel constructs – There are 3 levels of nested OpenMP parallel regions, 2 threads in each level. The `num_threads` clause is used to specify the number of threads desired for each parallel region.

```

1  subroutine report_num_threads(level)
2      use omp_lib
3      implicit none
4
5      integer :: level
6      !$omp single
7          write(*,100) level, omp_get_num_threads()
8 100  format("Level ", I3, ": number of threads in the team is ", I6)
9      !$omp end single
10 end subroutine report_num_threads
11
12 program main
13     use omp_lib
14     implicit none
15     external :: report_num_threads
16
17     call omp_set_dynamic(.false.)
18     !$omp parallel num_threads(2)
19         call report_num_threads(1)
20     !$omp parallel num_threads(2)
21         call report_num_threads(2)
22     !$omp parallel num_threads(2)
23         call report_num_threads(3)
24     !$omp end parallel
25     !$omp end parallel
26     !$omp end parallel
27 end program main

```


Figure 12.12: Controlling thread affinity

Affinity format example – We set the thread affinity format string and then ran the STREAM benchmark on the server-node with logical CPU numbering from Figure 12.4. We show two different executions of the STREAM benchmark: one with OMP_PROC_BIND set to spread and the other with OMP_PROC_BIND set to close.

```
$ ifort -qopenmp -DNTIMES=20 -DSTREAM_ARRAY_SIZE=64000000 -c stream.f
$ ifort -qopenmp -o stream stream.o
$ export OMP_DISPLAY_AFFINITY=true
$ export OMP_AFFINITY_FORMAT="Thrd Lev=%3L, thrd_num=%5n, thrd_aff=%15A"
$ export OMP_PLACES=threads
$ export OMP_NUM_THREADS=8
$ export OMP_PROC_BIND=spread
```

```
$ ./stream | sort -k3
<stream results omitted ...>
Thrd Lev=1 , thrd_num=0 , thrd_aff=0
Thrd Lev=1 , thrd_num=1 , thrd_aff=8
Thrd Lev=1 , thrd_num=2 , thrd_aff=16
Thrd Lev=1 , thrd_num=3 , thrd_aff=24
Thrd Lev=1 , thrd_num=4 , thrd_aff=1
Thrd Lev=1 , thrd_num=5 , thrd_aff=9
Thrd Lev=1 , thrd_num=6 , thrd_aff=17
Thrd Lev=1 , thrd_num=7 , thrd_aff=25
```

```
$ export OMP_PROC_BIND=close
$ ./stream |sort -k3
<stream results omitted ...>
Thrd Lev=1 , thrd_num=0 , thread_aff=0
Thrd Lev=1 , thrd_num=1 , thread_aff=32
Thrd Lev=1 , thrd_num=2 , thread_aff=2
Thrd Lev=1 , thrd_num=3 , thread_aff=34
Thrd Lev=1 , thrd_num=4 , thread_aff=4
Thrd Lev=1 , thrd_num=5 , thread_aff=36
Thrd Lev=1 , thrd_num=6 , thread_aff=6
Thrd Lev=1 , thrd_num=7 , thread_af=38
```

Figure 12.13: Serial Pi program (from Figure 4.5)

Serial Pi program –This program approximates a definite integral using the midpoint rule. The loop iterations are independent other than the summation into `sum`. Note that we must explicitly represent all constants as floats to prevent internal operations from using double precision.

```
1          PROGRAM MAIN
2          IMPLICIT NONE
3
4          INTEGER :: i
5          INTEGER, PARAMETER :: num_steps = 1000000
6          REAL :: x, pi, sum, step
7
8          sum = 0.0
9
10         step = 1.0/num_steps
11
12         DO i = 1, num_steps
13             x = (i - 0.5) * step
14             sum = sum + 4.0 / (1.0 + x * x)
15         ENDDO
16
17         pi = step * sum
18         print *, "pi=", pi
19
20     END PROGRAM MAIN
```

Figure 12.14: Unrolled loop in the Pi program

Serial Pi program with loops unrolled by 4 – Numerical integration to estimate Pi. We assume the number of steps is evenly divided by 4 just to keep the program simpler.

```

1          PROGRAM MAIN
2          IMPLICIT NONE
3
4          INTEGER :: i
5          INTEGER, PARAMETER :: num_steps = 100000
6          REAL :: x0, x1, x2, x3, pi, sum
7          REAL :: step
8
9          sum = 0.0
10         step = 1.0/num_steps
11
12         DO i = 1, num_steps, 4
13             x0 = (i - 0.5) * step
14             x1 = (i + 0.5) * step
15             x2 = (i + 1.5) * step
16             x3 = (i + 2.5) * step
17             sum = sum + 4.0 * (1.0 / (1.0 + x0 * x0) &
18                 & + 1.0 / (1.0 + x1 * x1) &
19                 & + 1.0 / (1.0 + x2 * x2) &
20                 & + 1.0 / (1.0 + x3 * x3))
21         ENDDO
22
23         pi = step * sum
24
25         WRITE(*,100) pi, num_steps
26 100    FORMAT('pi = ', f15.8, ', ', i14, ' steps ')
27
28         END PROGRAM MAIN

```

Figure 12.15: Calling a C function with SSE code from Fortran

Pi program using SSE vector intrinsics – Numerical integration to estimate Pi. We assume the number of steps is evenly divided by 4 just to keep the program simpler.

```

Save the contents in 2 files as below:
! To build and run:
! % gcc -c get_pi_vec.c
! % gfortran get_pi_vec.o Fig_12.15_explicitVecPi.f90
! % ./a.out

# File 1: "Fig_12.15_explicitVecPi.f90"

program main
  interface
    function get_pi_vec () result(r) bind(C, name="get_pi_vec")
      use, intrinsic :: iso_c_binding, only : c_float
      real(c_float) :: r
    end function get_pi_vec
  end interface

  real :: pi
  pi = get_pi_vec()
  print *, "in Fortran: pi=", pi
end program

#File 2: "get_pi_vec.c"

#include <x86intrin.h>
static long num_steps = 100000;
float scalar_four = 4.0f, scalar_zero = 0.0f, scalar_one = 1.0f;
float step;
float get_pi_vec ()
{
  int i;
  float pi;
  float vsum[4], ival;

```

```
step = 1.0f/(double) num_steps;

__m128 ramp = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
__m128 one = _mm_load1_ps(&scalar_one);
__m128 four = _mm_load1_ps(&scalar_four);
__m128 vstep = _mm_load1_ps(&step);
__m128 sum = _mm_load1_ps(&scalar_zero);
__m128 xvec;
__m128 denom;
__m128 eye;

for (i = 0; i < num_steps; i = i + 4){
    ival = (float) i;
    eye = _mm_load1_ps(&ival);
    xvec = _mm_mul_ps(_mm_add_ps(eye,ramp), vstep);
    denom = _mm_add_ps(_mm_mul_ps(xvec,xvec), one);
    sum = _mm_add_ps(_mm_div_ps(four,denom), sum);
}
_mm_store_ps(&vsum[0], sum);

pi = step * (vsum[0] + vsum[1] + vsum[2] + vsum[3]);
return pi;
}
```

Figure 12.16: Multithreading with SSE vectorization

A multithreaded and vectorized Pi program – This program carries out a numerical integration to estimate Pi. We assume the number of steps is evenly divisible by 4 and that we got 4 threads just to keep the program simple.

```

1  Save the contents in 2 files as below:
2  ! To build and run:
3  ! % gcc -c -fopenmp get_pi_par_vec.c
4  ! % gfortran -fopenmp get_pi_par_vec.o Fig_12.16_parVecPi.f90
5  ! % ./a.out
6
7  ! Save the contents in 2 files as below:
8  ! To build and run:
9  ! % gcc -c -fopenmp get_pi_par_vec.c
10 ! % gfortran -fopenmp get_pi_par_vec.o Fig_12.16_parVecPi.f90
11 ! % ./a.out
12
13 #File 1: "Fig_12.16_parVecPi.f90"
14
15 program main
16     use omp_lib
17     interface
18         function get_pi_vec (r) result(r) bind(C, name="get_pi_vec")
19             use, intrinsic :: iso_c_binding, only : c_float
20             real(c_float) :: r
21         end function get_pi_vec
22     end interface
23
24     real :: pi
25     pi = get_pi_vec()
26     print *, "in Fortran: pi=", pi
27 end program
28
29 #File 2: "get_pi_par_vec.c"
30
31 #include <x86intrin.h>
32 static long num_steps = 100000;
33 #define MAX_THREADS 4

```

```

34 float scalar_four = 4.0f, scalar_zero = 0.0f, scalar_one = 1.0f;
35 float step;
36 float get_pi_par_vec ()
37 {
38     int i, k;
39     float local_sum[MAX_THREADS];
40     float pi, sum = 0.0;
41     step = 1.0f/(double) num_steps;
42
43     for (k = 0; k < MAX_THREADS; k++) local_sum[k] = 0.0;
44
45     #pragma omp parallel num_threads(4) private(i)
46     {
47         int ID = omp_get_thread_num();
48         float vsum[4], ival, scalar_four = 4.0;
49
50         __m128 ramp = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
51         __m128 one = _mm_load1_ps(&scalar_one);
52         __m128 four = _mm_load1_ps(&scalar_four);
53         __m128 vstep = _mm_load1_ps(&step);
54         __m128 sum = _mm_load1_ps(&scalar_zero);
55         __m128 xvec;
56         __m128 denom;
57         __m128 eye;
58
59         // unroll loop 4 times ... assume num_steps\%4 =0
60         #pragma omp for schedule(static)
61         for (i = 0; i < num_steps; i = i + 4){
62             ival = (float) i;
63             eye = _mm_load1_ps(&ival);
64             xvec = _mm_mul_ps(_mm_add_ps(eye,ramp), vstep);
65             denom = _mm_add_ps(_mm_mul_ps(xvec,xvec), one);
66             sum = _mm_add_ps(_mm_div_ps(four,denom), sum);
67         }
68         _mm_store_ps(&vsum[0], sum);
69         local_sum[ID] = step * (vsum[0] + vsum[1] + vsum[2] + vsum[3]);
70     }
71     for (k = 0; k < MAX_THREADS; k++) pi += local_sum[k];
72     return pi;

```

73 }

Figure 12.17: Vectorization with the OpenMP SIMD construct

OpenMP program to vectorize the Pi program – The `simd` clause directs the compiler to explicitly vectorize the program. As with many OpenMP features, this clause asserts to the compiler that it is safe to vectorize the code and it will do so, even if there are loop-carried dependencies that should prevent vectorization.

```
1          PROGRAM MAIN
2          IMPLICIT NONE
3
4          INTEGER :: i
5          INTEGER, PARAMETER :: num_steps = 100000
6          REAL :: x, pi, sum, step
7
8          sum = 0.0
9
10         step = 1.0/num_steps
11
12  !$OMP parallel SIMD private(x) reduction(+:sum)
13         DO i = 1, num_steps
14             x = (i - 0.5) * step
15             sum = sum + 4.0 / (1.0 + x * x)
16         ENDDO
17  !$OMP end parallel SIMD
18
19         pi = step * sum
20         print *, "pi=", pi
21
22         END PROGRAM MAIN
```

Figure 12.18: Combining multithreading and vectorization

OpenMP program to multithread and vectorize the Pi program – This is a familiar “parallel do” approach to solving the problem but we have added one additional clause: a `simd` clause for explicit vectorization.

```
1          PROGRAM MAIN
2
3          USE OMP_LIB
4          IMPLICIT NONE
5
6          INTEGER :: i
7
8          INTEGER, PARAMETER :: num_steps = 100000
9          REAL :: x, pi, sum, step
10
11         sum = 0.0
12
13         step = 1.0/num_steps
14
15         !$omp parallel do simd private(x) reduction(+:sum)
16             DO i = 1, num_steps
17                 x = (i - 0.5) * step
18                 sum = sum + 4.0 / (1.0 + x * x)
19             ENDDO
20         !$omp end parallel do simd
21
22         pi = step * sum
23
24         print *, "pi=", pi
25         END PROGRAM MAIN
```

Figure 12.19: Target construct with default data movement**OpenMP program for elementwise multiplication of vectors on a GPU –**

Default data movement moves the vectors **a**, **b**, and **c** onto the device before the computation starts and back onto the host (the CPU) when the computation has completed.

```
1 program main
2     use omp_lib
3     implicit none
4
5     integer , parameter :: N = 1024
6     real*8 :: a(N), b(N), c(N)
7     integer :: i
8
9     ! initialize a, b, and c (code not shown)
10
11     !$omp target
12     !$omp teams distribute parallel do simd
13         do i = 1, N
14             c(i) = c(i) + a(i) * b(i)
15         enddo
16     !$omp end teams distribute parallel do simd
17     !$omp end target
18
19 end program main
```

Figure 12.20: Explicit data movement between the host and a device

Explicit data movement with the target directive – The map clause controls movement of data from the host **to** a device or **from** the device onto the host. When working with pointers to arrays, you need to use array sections to define precisely which data to move.

```
1 program main
2   use omp_lib
3
4   integer , parameter :: N = 1024
5   real*8, allocatable , dimension (:) :: a, b, c
6   integer :: i
7
8   allocate (a(N))
9   allocate (b(N))
10  allocate (c(N))
11
12 ! initialize a, b, and c (code not shown)
13
14 !$omp target map(to:a(1:N),b(1:N)) map(tofrom:c(1:N))
15 !$omp teams distribute parallel do simd
16   do i = 1, N
17     c(i) = c(i) + a(i) * b(i)
18   enddo
19 !$omp end teams distribute parallel do simd
20 !$omp end target
21
22 end program main
```

Figure 12.21: Data movement between multiple target regions

Multiple target regions – The map clause controls movement of data from the host **to** a device or **from** the device onto the host. When working with pointers to arrays, you need to use array sections to define precisely which data to move.

```
1 program main
2   use omp_lib
3
4   integer , parameter :: N = 1024
5   real*8, allocatable , dimension (:) :: a, b, c, d
6   integer :: i
7
8   allocate (a(N))
9   allocate (b(N))
10  allocate (c(N))
11  allocate (d(N))
12
13 ! initialize a, b, and c (code not shown)
14
15 !$omp target map(to:a(1:N),b(1:N)) map(tofrom:c(1:N))
16 !$omp teams distribute parallel do simd
17   do i = 1, N
18     c(i) = c(i) + a(i) * b(i)
19   enddo
20 !$omp end teams distribute parallel do simd
21 !$omp end target
22
23 !$omp target map(to:a(1:N),b(1:N)) map(tofrom:d(1:N))
24 !$omp teams distribute parallel do simd
25   do i = 1, N
26     d(i) = d(i) + a(i) * c(i)
27   enddo
28 !$omp end teams distribute parallel do simd
29 !$omp end target
30
31 end program main
```

Figure 12.22: Managing data movement across multiple target regions

Target Data Region – A single target data region manages data at the level of a device. It persists and is used between multiple target constructs. code:ompTargDat

```

1 program main
2   use omp_lib
3   integer , parameter :: N = 1024
4   real*8, allocatable , dimension (:) :: a, b, c, d
5   integer :: i
6   allocate (a(N))
7   allocate (b(N))
8   allocate (c(N))
9   allocate (d(N))
10
11 ! initialize a, b, and c (code not shown)
12
13 !$omp target data map(to:a(1:N),b(1:N),c(1:N)) map(tofrom:d(1:N))
14
15 !$omp target
16 !$omp teams distribute parallel do simd
17   do i = 1, N
18     c(i) = c(i) + a(i) * b(i)
19   enddo
20 !$omp end teams distribute parallel do simd
21 !$omp end target
22
23 !$omp target
24 !$omp teams distribute parallel do simd
25   do i = 1, N
26     d(i) = d(i) + a(i) * c(i)
27   enddo
28 !$omp end teams distribute parallel do simd
29 !$omp end target
30
31 !$omp end target data
32
33 end program main

```

13 Your Continuing Education in OpenMP

Figure 13.1: Task with critical constructs can deadlock

A subtle deadlock with tasks: – This is the tasking.9.c example from the OpenMP 4.5 Examples document. This function can deadlock if the thread suspends task 1 to begin work on task 2.

```
1 ! sample compile command to generate *.o object file :
2 !   gfortranc -fopenmp -c Fig_13.1_taskBug.f90
3
4 subroutine  work()
5     use omp_lib
6     implicit none
7
8     !$omp task    ! task 1
9         !$omp task    ! task 2
10            !$omp critical    ! Critical region 1
11                ! do work here
12            !$omp end critical    ! end Critical Region 1
13        !$omp end task    ! end task2
14        !$omp critical    ! Critical Region 2
15            !$omp task    ! task 3
16                ! do work here }
17            !$omp end task    ! end task3
18        !$omp end critical    ! end Critical Region 2
19    !$omp end task    ! end task 1
20 end subroutine
```