# OpenMP

## *Objectives*

- Overview of OpenMP
- Structured blocks
- Variable scope, work-sharing
- Scheduling, synchronization

## *Overview of OpenMP*

OpenMP is a collection of *compiler directives* and *library functions* that are used to create parallel programs for *shared-memory* computers.

The “MP” in OpenMP stands for “multi-processing”, another term for shared-memory parallel computing.

OpenMP is combined with C, C++, or Fortran to create a *multithreading programming language*, in which all processes are assumed to share a single *address space*.

OpenMP is based on the *fork / join* programming model: all programs start as a single (*master*) thread, fork additional threads where parallelism is desired (the *parallel region*), then join back together.

This sequence of events is repeated until all the parallel regions have been executed.

**Note:** The threads must synchronize before joining.

## *Overview of OpenMP*

The philosophy of OpenMP is to not sacrifice ease of coding and maintenance in the name of performance.

Accordingly, OpenMP was designed based on two principles: *sequential equivalence* and *incremental parallelism*.

A program is said to be sequentially equivalent if it returns the same results whether it executes on one thread or many threads.

Such programs are generally easier to understand, write, and hence maintain.

Incremental parallelism is the process of taking working serial code and converting pieces to execute in parallel.

At each increment, the code can be re-tested to ensure its correctness, thus enhancing the likelihood of success for the overall project.

Note that although this process sounds appealing, it is not universally applicable.

## *Core concepts*

Recall the “hello, world!” program in Fortran90:

```
PROGRAM helloWorld

  PRINT *, "hello, world"

END PROGRAM helloWorld
```

We have hinted that OpenMP is *explicitly parallel*:

Any parallelism in the code has to be put there explicitly by the programmer.

The good news is that the low-level details of how the parallelism is executed is done automatically.

In Fortran, it is easy to denote that the `PRINT` statement should be executed on each thread by enclosing it in a block governed by a *compiler directive*.

## *Core concepts*

```
PROGRAM HelloWorldOpenMP

  !$OMP PARALLEL
  PRINT*, "hello, world"
  !$OMP END PARALLEL

END PROGRAM HelloWorldOpenMP
```

The program can be compiled on a shared-memory machine (like `moneta.usask.ca`) via

```
gfortran -fopenmp helloWorldOMP.f90 -o helloWorldOMP
```

and executed with

```
./helloWorldOMP
```

# *Core concepts*

On `moneta` this produces the output

```
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
```

## *Core concepts*

**Notes:**

1. We can conclude that the default number of threads on `moneta` is 16.

2. It is possible to specify the number of threads (e.g., 4) by setting an *environment variable* via

   ```
   setenv OMP_NUM_THREADS 4
   ```

   or from within the program via the statement

   ```
   CALL OMP_SET_NUM_THREADS(4)
   ```

3. OpenMP requires that I/O be *thread safe*; i.e., output from one thread is handled without interference from any other threads.

   However, as usual, the threads can print out in any order, depending on the order in which they reach the print command.

4. This program can be compiled and run on a serial machine (with one thread) using a serial compiler because the compiler directives are treated as comments.

5. Compiler directives using a fixed format (as per Fortran 77) can be specified as

   ```
   !$OMP
   *$OMP
   ```

   They must start in column 1; continuation lines must have a non-blank or non-zero character in column 6; comments may appear after column 6 starting with `!`.

   Only `!$OMP` is available for free format. The directive must start the line, but it may start at any column; & is the continuation marker at the end of the line; comments may appear after column 6 starting with `!`.

## *Core concepts*

Things are slightly different in C.

The compiler directives are called `pragmas`, with syntax

```
# pragma
```

where the # appears in column 1 and the remainder of the directive is aligned with the rest of the code.

`pragmas` are only allowed to be one line long; so if one happens to require more than one line, the line can be continued using $\backslash$ at the end of intermediate lines.

## *Core concepts*

The code looks like:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

/*--------------------------------------------------------------------*/
int main(int argc, char* argv[])
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
  /* main */

/*--------------------------------------------------------------------
 * Function:    Hello
 * Purpose:     Thread function that prints message
 */
void Hello(void)
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

  /* Hello */
```

## *Core concepts*

This code can be compiled and run using

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

To run, we specify the number of threads on the command line; e.g., to run with $4$ threads, we use

```
./omp_hello 4
```

Output might look like

```
Hello from thread 3 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
```

Note that for this program if you forget to specify the number of threads, you will see the dreaded

```
Segmentation fault: 11
```

To be able to run without specifying the number of threads (as we did in Fortran), all code pertaining to `strtol` should be removed.

## ***Core concepts***

Things to note from the code:

- OpenMP is a library of functions and macros, so we need to include a header file with prototypes and macro definitions.

- The `strtol` function from `stdlib.h` gets the number of threads from the command line.

  The syntax is

  ```
  long strtol(
        const char* number p  /* in  */
        char**      end p     /* out */
        int         base      /* in  */);
  ```

  The first argument is the command-line argument; the last is the numeric base in which the string is represented — in this case $10$. We do not make use of the second argument so we pass `NULL`.

## *Core concepts*

The `pragma` then says the program should start a number of threads equal to what was passed in via the command line.

Each thread then executes the function `Hello`.

The threads rejoin the main thread when they return from `Hello`, at which point they are terminated.

The main thread is then itself terminated.

That is quite a bit of action for relatively little code!

OpenMP `pragma`s start with

```
# pragma omp
```

The directive

```
# pragma omp parallel
```

specifies that the structured block of code that follows is to be executed by multiple threads.

## *Structured Blocks*

An OpenMP *construct* is defined to be a *compiler directive* plus a *block of code*.

The block of code must be *structured* in the sense that it must have a single point of entry at the top and a single point of exit at the bottom; i.e., **branching in or out of a structured block is not allowed!**

This usually leads to a (fatal) compile-time error.

Similarly, a structured block cannot contain a `RETURN` statement.

Only `STOP` statements (in Fortran) or calls to the function `exit` (in C) prevent the program from exiting a valid structured block.

The `END` directive in Fortran is not necessary if the structured block only contains one statement.

## *Structured Blocks*

In C, we prefer to specify the number of threads via the command line, so we modify our `parallel` directive with a `num_threads` *clause*; i.e., in OpenMP, a clause is something that modifies a directive.

There may be system limitations that preclude running a given number of threads.

In fact, OpenMP does not guarantee it will start `num_threads` threads in case, but the limit in practice on available threads can be thousands or even millions, so it is uncommon to not get all the threads requested.

## *Structured Blocks*

Prior to the `parallel` directive, the program used a single thread.

Upon reaching the `parallel` directive, the original thread continues and (`num_threads`$-1$) new threads are spawned.

In OpenMP parlance, the original thread is called the *master*, the new threads are called the *slaves*, and collectively the threads are called a *team*.

Each thread in the team executes the block following the directive; in this case, the call to function `Hello`.

When the block is completed, there is an *implicit barrier*, at which threads wait for all to reach before completing the block.

When the block is completed, all the slave threads are terminated and only the master thread continues; in our case, to execute the `return` statement and terminate the program.

## *Structured Blocks*

Each thread has its own stack, so it will have its own private (local) variables.

In particular, each thread gets its own rank from calling

`omp_get_thread_num`

with syntax

```
int omp_get_thread_num(void);
```

and the number of threads in the team from calling

`omp_get_num_threads`

with syntax

```
int omp_get_num_threads(void);
```

In OpenMP, `stdout` is shared among the threads, so each thread can execute the `printf` statement.

There is no scheduling of access to `stdout`, so output is non-deterministic.

## *Error checking*

In the name of compactness, we rarely include any error checking; however, in general it is an exceptionally good idea to check for errors while writing code.

The obvious one for the example is to check for the presence of a command-line argument (and whether it is positive!) and handle it more elegantly than allowing a segmentation fault.

Another source of potential problems is the compiler.

OpenMP is designed so that if the compiler does not support it, the `parallel` directive is ignored; however, the attempts to include `omp.h` and call `omp_get_thread_num` and `omp_get_num_threads` will cause errors.

To handle these issues, we can check for the definition of the preprocessor macro `_OPENMP`.

If it is defined, we can include `omp.h` and make calls to OpenMP functions.

## *Error checking*

Code to check for including `omp.h` might look like

```
# ifdef _OPENMP
#    include <omp.h>
# endif
```

Similarly, code to check for using the other OpenMP functions used in the example might look like

```
# ifdef _OPENMP
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
# else
    int my_rank = 0;
    int thread_count = 1;
# endif
```

In other words, we specify explicitly that if OpenMP is not available, then the code will execute with one thread having rank 0.

## ***Example: Trapezoidal Rule***

We consider again the example of computing a definite integral by means of the trapezoidal rule.

To refresh our memories of the pertinent facts, we are computing

$$\int_a^b f(x)\,dx \approx \Delta x \left[ f(x_0)/2 + \sum_{i=1}^{n-1} f(x_i) + f(x_n)/2 \right]$$

by means of $n$ trapezoids of width $\Delta x$.

Serial code to accomplish this might look like

```
/* input a, b, n */
dx = (b-a)/n;
approx = (f(a)+f(b))/2;
for (i = 1; i <= n-1; i++){
  x_i = a + i*dx;
  approx += f(x_i);
}
approx = dx*approx;
```

## ***Example: Trapezoidal Rule***

Foster's parallel program design methodology yields

1. find the area of a single trapezoid; add areas up

2. no communication between tasks computing single trapezoid areas, but all answers are communicated

3. assuming more trapezoids than threads, split $[a, b]$ into `num_threads` subintervals

4. map tasks to threads (subintervals to threads; results back to thread 0)

## ***Example: Trapezoidal Rule***

Care must be exercised in accumulating the results in thread 0.

If we simply use a shared variable to accumulate the individual results, the end result will be unpredictable (and hence likely erroneous).

For example, a statement such as

```
total_area += local_area;
```

will lead to an erroneous result if more than one thread attempts to simultaneously execute it; i.e., threads may be adding to an old (incorrect) value of `total_area` instead of the current (correct) one.

Recall this is called a *race condition*.

We need a way to ensure that once one thread begins executing the statement, other threads must wait until it is finished before they can execute the statement.

## *Example: Trapezoidal Rule*

In OpenMP, we define a *critical section* by means of the `critical` directive:

```
# pragma omp critical
  total_area += local_area;
```

This directive explicitly says that only one thread can execute this block of code at a time; i.e., each thread has *mutually exclusive* access to the block of code.

The following is a listing of a code that accomplishes this as well as some basic error checking (albeit with a cryptic error message) that a command-line argument must be supplied to the code in order for it to run and that the number of trapezoids must be evenly divisible by the number of threads assigned.

The code also assumes the compiler can handle OpenMP programs and hence does not check for the existence of `_OPENMP`.

## *Example: Trapezoidal Rule*

```
/* File:    omp_trap1.c
 * Purpose: Estimate definite integral (or area under curve)
 *          using the trapezoidal rule.
 *
 * Input:   a, b, n
 * Output:  estimate of integral from a to b of f(x)
 *          using n trapezoids.
 *
 * Compile: gcc -g -Wall -fopenmp -o omp_trap1 omp_trap1.c
 * Usage:   ./omp_trap1 <number of threads>
 *
 * Notes:
 *   1.  The function f(x) is hardwired.
 *   2.  In this version, each thread explicitly computes the integral
 *       over its assigned subinterval, a critical directive is used
 *       for the global sum.
 *   3.  This version assumes that n is evenly divisible by the
 *       number of threads
 *
 * IPP:  Section 5.2.1 (pp. 216 and ff.)
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

void Usage(char* prog_name);
double f(double x);    /* Function we’re integrating */
void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[])
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
   if (n % thread_count != 0) Usage(argv[0]);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
  /* main */

/*--------------------------------------------------------------------
 * Function:    Usage
 * Purpose:     Print command line for function and terminate
 * In arg:      prog_name
 */
void Usage(char* prog_name)

   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   fprintf(stderr, "   number of trapezoids must be evenly divisible by\n");
   fprintf(stderr, "   number of threads\n");
   exit(0);
  /* Usage */

/*------------------------------------------------------------------
 * Function:    f
 * Purpose:     Compute value of function to be integrated
 * Input arg:   x
 * Return val:  f(x)
 */
double f(double x)
```

```
   double return_val;

   return_val = x*x;
   return return_val;
  /* f */

/*------------------------------------------------------------------
 * Function:    Trap
 * Purpose:     Use trapezoidal rule to estimate definite integral
 * Input args:
 *    a: left endpoint
 *    b: right endpoint
 *    n: number of trapezoids
 * Output arg:
 *    integral:  estimate of integral from a to b of f(x)
 */
void Trap(double a, double b, int n, double* global_result_p)
   double  dx, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   dx = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*dx;
   local_b = local_a + local_n*dx;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++)
     x = local_a + i*dx;
     my_result += f(x);

   my_result = my_result*dx;

#  pragma omp critical
   *global_result_p += my_result;
  /* Trap */
```

## *Variable Scope*

The *scope* of a variable refers to the parts of a program in which the variable can be "seen" (accessed).

For example, in C, variables declared at the beginning of a function can only be seen within the function; variables declared at the beginning of a file can be seen by any function in the file that declares the variable.

In OpenMP, scope refers to the set of threads that can see a variable in a `parallel` block.

When a variable can be seen by all threads in a team, it is said to have *shared* scope; a variable that can be seen by only one thread is said to have *private* scope.

OpenMP is a shared-memory programming model.

A general rule is that *any variable declared outside of a parallel region has a shared scope*.

In some sense, the "default" variable scope is shared.

## *Variable Scope*

So the output from

```
PROGRAM HelloWorld2OpenMP

  INTEGER :: I=57
  !$OMP PARALLEL
  PRINT*, "hello, world", I
  !$OMP END PARALLEL

END PROGRAM HelloWorld2OpenMP
```

is not surprisingly

```
 hello, world          57
 hello, world          57
       ⋮
 hello, world          57
 hello, world          57
```

(The variable `I` has shared scope.)

## *Variable Scope*

If a variable is declared inside a parallel region, it has private (or *local*) scope to that thread.

Because in Fortran, variable declarations can only occur at the beginning of a (sub)program, we need to specify which variables are local to a given block:

```
PROGRAM HelloWorld2OpenMP

  INTEGER :: threadNUM, OMP_GET_THREAD_NUM

  !$OMP PARALLEL PRIVATE(threadNUM)
  threadNUM = OMP_GET_THREAD_NUM()
  PRINT*, "hello, world!", threadNUM
  !$OMP END PARALLEL

END PROGRAM HelloWorld2OpenMP
```

```
 hello, world!           1
 hello, world!           3
     ⋮
 hello, world!           6
```

## *Variable Scope*

Similarly in `omp_hello.c`, the variables `my_rank` and `thread_count` were declared in the function `Hello`, which is itself called from inside the `parallel` block.

These variables are all allocated from each thread's (private) stack and hence have private scope.

In `omp_trap1.c`, the variables declared in `main` (`a`, `b`, `n`, `global_result`, and `thread_count`) are all shared (and the code makes implicit use of this in `Trap`).

The general rule is that variables declared before a `parallel` directive have a shared scope and start off on all threads with the values they have at the beginning of the parallel block.

After completion of the block, the variables retain whatever last values they have after the last thread has updated them.

Variables declared within the `parallel` block have private scope.

## *Reduction Clauses*

OpenMP also has *reduction clauses*, i.e., operations that combine a set of values into a single value by means of a binary, associative operator.

The Fortran syntax is

```
REDUCTION(operator|intrinsic: list)
```

where `operator` can be +, −, ∗, `.AND.`, `.OR.`, etc.; `intrinsic` can be `MAX`, `MIN`, etc; – can be problematic.

For each name in `list`, a private variable is created and initialized with the identity element of the operator (e.g., for +, it would be 0).

Each thread carries out the reduction into its copy of the local variable associated with the name in `list`.

At the end of the construct with the `REDUCTION` clause, the local values are combined to define a single value.

This value is assigned to the variable with the same name in the region after the construct.

## *Example: Dot Product*

```
! OpenMP program to compute dot product of vectors X and Y
PROGRAM dotProduct

  INTEGER, PARAMETER :: N=100, CHUNKSIZE=7
  REAL :: X(N), Y(N), XdotY = 0.0
  INTEGER I

  ! initialize the vectors; parallelize just for fun
  !OMP PARALLEL DO LASTPRIVATE(X,Y)
  DO I=1,N
     X(I) = I
     Y(I) = I**2
  END DO
  !OMP END PARALLEL DO

  !OMP PARALLEL DO &
  !OMP SCHEDULE(STATIC,CHUNKSIZE) &
  !OMP REDUCTION(+:XdotY)
  DO I=1,N
     XdotY = XdotY + X(I)*Y(I)
  END DO
  !OMP END PARALLEL DO

  PRINT*, 'X.Y = ', XdotY
  PRINT*, 'Exact answer = ', (N*(N+1)/2)**2

END PROGRAM dotProduct
```

## ***Reduction Clauses***

Defining a critical section allowed the trapezoidal rule program to get the correct answer, but it serialized the code in order to do so.

Arguably, this is not a serious drawback for the trapezoidal rule program, but in general such serialization could be, especially if it is avoidable (as it is in this case).

We can use a reduction operator to improve the performance of the trapezoidal rule program.

The syntax in C for the `reduction` clause is

```
reduction(<operator>: <variable list>)
```

where `operator` can be `+`, `*`, `&`, `|`, etc.

## ***Reduction Clauses***

To allow the trapezoidal rule code to take advantage of a `reduction` clause, it can be modified as follows.

In the function `Trap`,

```
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

where the function `Local_trap` has been created to return local integrations that can then be reduced into the shared variable `global_result`.

Note that the `parallel` directive is spread over two lines using the \ character.

## *Reduction Clauses*

Variables included in a `reduction` clause are shared.

However, a private variable is created for each thread.

This private variable is updated in the `parallel` block; when the `parallel` block ends, the private variables are reduced into the shared variable.

Finally, the private variables are initialized to the *identity* member of the reduction operator.

For example, if the reduction operator is `+`, the private variables are initialized to 0; if the reduction operator is `*`, they are initialized to 1.

## PARALLEL DO / parallel for

Because the construct `PARALLEL` followed by a loop is so common, OpenMP has equivalent shortcut syntax.

In Fortran, it is

```
!$OMP PARALLEL DO
...
!$OMP END PARALLEL DO
```

In C, it is invoked as in

```
# pragma omp parallel for num_threads(thread_count)
```

It is very similar to the `parallel` directive; i.e.,

A team of threads is forked to execute the subsequent structured block; however, this structured block must be a `DO`/`for` loop.

## PARALLEL DO / parallel for

The loop is parallelized by dividing the iterations among the threads.

Thus, `parallel for` is actually quite different from `parallel` because the work in a `parallel` block is divided among the threads by the threads themselves.

The actual partitioning of a `parallel for` loop is determined by the system, but the default is usually a rough block partition; i.e., if there are `m` iterations and `p` threads, then roughly the first `m/p` iterations are assigned to thread 0, etc.

The trapezoidal rule code can be modified to

```
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i=1; i<= n-1; i++)
      approx += f(a+i*dx);
   approx = dx*approx;
```

## `PARALLEL DO` / `parallel for`

Notes on scope:

- It is essential that `approx` be a reduction variable. Otherwise, it would default to being a shared variable and hence the loop body would be an unprotected critical section.

- Similarly, the scope of loop variable `i` defaults to private; it is clear that having a shared loop variable would create a race condition (and likely a huge mess in the code).

  In other words, each thread has its own local copy of `i`.

## `PARALLEL DO` / `parallel for`

Although it may seem idyllic that `for` loops can be parallelized by adding a single `parallel for` directive in front of them, there are some caveats associated with its use.

The first one is that `while` or `do-while` loops cannot be parallelized in this way.

In principle, it is possible to convert such loops to `for` loops anyway, so this may not appear to be much of a restriction in practice.

However, in the spirit of this, OpenMP will only parallelize `for` loops for which the number of iterations can be determined

- from the `for` statement itself, or
- prior to execution of the loop.

## PARALLEL DO / parallel for

So, we cannot parallelize the infinite loop

```
for ( ; ; ) {
    ...
}
```

nor can we parallelize the loop

```
for (i=0; i<n; i++) {
    if ( ... ) break;
    ...
}
```

because the precise number of iterations cannot be determined from the `for` statement alone.

Indeed, this `for` loop is not a structured block because the `break` statement adds another point of exit from the loop.

## `PARALLEL DO` / `parallel for`

In fact, OpenMP will only parallelize `for` loops that are in *canonical* form:

```
for (index = start; \
     index <,<=,>=,> end; \
     ±±index±±, index ±= incr )
```

subject to the usual restrictions

- `index` must have type integer or pointer (not float)
- `start`, `end`, `incr` must have compatible types
- `start`, `end`, `incr` must not change during the loop
- `index` can only change via `incr` during the loop

All these things ensure the number of iterations can be determined before the loop is executed.

The only allowable exception is that a call to `exit` may be made in the body of the loop.

## *Data dependencies*

Programs that do not satisfy the preceding rules will not compile.

For example, an attempt with gcc to compile the code

```
int Linear search(int key, int A[], int n) {
   int i;
   /* thread count is global */
#  pragma omp parallel for num_threads(thread _count)
   for (i = 0; i < n; i++)
      if (A[i] == key) return i;
   return 1; /* key not in list */
}
```

should return with an error like

```
Line 6: error: invalid exit from OpenMP structured block
```

Errors like this are most usefully interpreted as good news in the sense that they are obvious, and the compiler will not allow you to execute erroneous code.

## ***Data dependencies***

A more subtle problem occurs in loops where the computation in one iteration depends on the results of previous iterations.

Recall the Fibonacci sequence $\{F_n\}_{n=0}^{\infty}$ defined

$$F_0 = 1,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2},\ n = 2, 3, \ldots.$$

Consider the following attempt to parallelize a loop to calculate the first `n` Fibonacci numbers.

```
   fibo[0] = fibo[1] = 1;
#  pragma omp parallel for num_threads(thread_count)
   for (i = 2; i < n; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];
```

This code will compile, but if it is run with more than one thread, the result is (at best) unpredictable.

## *Data dependencies*

For example, with one thread (and mostly even two threads) we obtain the correct answer

```
1 1 2 3 5 8 13 21 34 55
```

but with three threads, we obtain

```
1 1 2 3 5 0 0 0 0 0
```

It appears that the computation of `fibo[2]`, `fibo[3]`, and `fibo[4]` went to one thread whereas the rest went to the others.

Then the other threads started before the first thread finished and were working with initialized values of $0$.

Hence, the remaining values were computed to be $0$.

Note that the correct output can be obtained using more than one thread if the threads complete in order; however the chances of this happening grow increasingly unlikely as the number of threads increases.

## ***Data dependencies***

The two take-away messages from this exercise are:

1. It is the programmer's responsibility to check for dependencies within iterations in a `parallel for` loop; OpenMP compilers will not do it.

2. It is unlikely that OpenMP will correctly parallelize loops for which iterations are not independent.

The dependence of `fibo[i]` on `fibo[i-1]`, etc., is called a *data dependency*; in the context of this example, it may be called a *loop-carried dependency*.

Fortunately, we need only worry about loop-carried dependencies (and not more general ones) in a `parallel for` loop.

## *Data dependencies*

For example, strictly speaking, in the loop

```
for (i=0; i < n; i++) {
   x[i] = a + i*dx;
   y[i] = exp(x[i]);
}
```

there is a data dependency between `y[i]` and `x[i]`.

However, there is no problem parallelizing it as a `parallel for` loop because the dependency remains *within an iteration*.

The key to recognizing (and hence avoiding) loop-carried dependencies is to look for variables that are read (or written) in one iteration but written (or re-written) in another.

## *Data dependencies*

For example, consider the following (poor) method for approximating $\pi$:

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \ldots\right].$$

A naive parallel implementation of this method might look like

```
   double factor = 1.0; sum = 0.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+:sum)
   for (k=0; k<n; k++) {
      sum += factor/(2*k+1);
      factor = -factor;
   }
   pi_approx = 4.0*sum;
```

It should be clear that `factor` is updated in a manner that introduces a loop-carried dependency.

## *Data dependencies*

It is straightforward to define `factor` only in terms of the iteration counter `k` by using

```
factor = (k % 2 == 0) ? 1.0 : -1.0;
```

*before* the line that increments `sum`.

Unfortunately, that change alone does not lead to correct parallel code.

The problem is that (by default) `factor` is still shared among the threads; it was declared outside of the `parallel for` directive.

Thus, `factor` can be modified by one thread between the time another thread computes it and uses it!

So, we need to ensure `factor` has private scope.

That is easily achieved via

```
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+:sum) private(factor)
```

## ***Data dependencies***

A final fact to keep in mind is that *the value of a variable with private scope is unspecified before the beginning and after the completion of the* `parallel` *(or* `parallel for`*) block.*

Put differently, a private variable only has scope within the `parallel` block, even if a variable with the same name is declared outside the block (and the scope of which is hence is shared), *they are distinct copies!*

So, e.g., if we declare and initialize the variable `factor` outside the `parallel` block, when we print it, we will see its value *as if the* `parallel` *block did not exist*.

Similarly, if we print its value from within the `parallel` block, we will see its value *independent of whether it existed outside the block*.

## *Data dependencies*

Rather than let OpenMP decide on the scope of a variable, it is better programming practice to specify variable scope explicitly.

This is accomplished via the `default` clause.

We would modify our `pragma` to read

```
#  pragma omp parallel for num_threads(thread_count) \
      default(none) reduction(+:sum) private(k,factor) \
      shared(n)
```

We note that `sum` is a reduction variable, so it has properties of both private and shared scope.

We have discussed the reasons to make `factor` private.

The loop counter `k` would have defaulted to private as well, but now we have to declare it explicitly.

Variables such as `n` that are not updated by the loop are safely shared.

Such variables keep their final values upon exiting the block, so changes made within the block matter.

## ***Data dependencies***

As a final example (in Fortran), we consider the following program, in which an expensive function `BIG_COMP` must be called many times:

```
PROGRAM bigComp
      DOUBLE PRECISION :: ANSWER = 0.0D0, RES, BIG_COMP
      INTEGER :: I, N = 1000

      DO I=1,N
         RES = BIG_COMP(I)
         CALL COMBINE(ANSWER, RES)
      END DO
      PRINT*, ANSWER

END PROGRAM bigComp
```

**We assume the `COMBINE` subroutine must be called in sequential order.**

## *Data dependencies*

To make the iterations independent, we put the call to `COMBINE` into its own loop.

This should be acceptable if it is not expensive compared with `BIG_COMP`.

The cost is that `RES` is now an array of size `N`.

```
PROGRAM bigComp2
  INTEGER, PARAMETER :: N = 1000
  DOUBLE PRECISION :: ANSWER = 0.0D0, RES(N), BIG_COMP
  INTEGER :: I

  DO I=1,N
     RES(I) = BIG_COMP(I)
  END DO

  DO I=1,N
     CALL COMBINE(ANSWER, RES(I))
  END DO
  PRINT*, ANSWER

END PROGRAM bigComp2
```

## *Data dependencies*

Now, no two threads update the same variable and thus can be safely executed in parallel.

The OpenMP work-sharing construct `PARALLEL` now assigns loop iterations to multiple threads.

```
PROGRAM bigComp3
  INTEGER, PARAMETER :: N = 1000
  DOUBLE PRECISION :: ANSWER = 0.0D0, RES(N), BIG_COMP
  INTEGER :: I

  !$OMP PARALLEL
  !$OMP DO
  DO I=1,N
     RES(I) = BIG_COMP(I)
  END DO
  !$OMP END DO
  !$OMP END PARALLEL

  DO I=1,N
     CALL COMBINE(ANSWER, RES(I))
  END DO
  PRINT*, ANSWER

END PROGRAM bigComp3
```

## *Work-sharing constructs*

The `parallel for` is known as a *work-sharing* construct because the work involved in executing a loop is distributed (shared) among the threads.

Different threads are doing different things.

This is in contrast to the `parallel` construct, where (ostensibly) each thread executes the same statements.

By default, there is an implicit *barrier* at the end of any OpenMP work-sharing construct.

This barrier can be removed via the `NOWAIT` clause:

```
!$OMP END DO NOWAIT
```

You should be careful when using `NOWAIT` because in most cases these barriers are needed to prevent *race conditions*, i.e., the situation in which an error in a parallel program occurs because the result depends on the relative scheduling / execution of the threads.

## *Work-sharing constructs*

There are two other common types of work-sharing constructs in OpenMP:

- `SINGLE`, which defines a block of code that only the first thread executes (all others skip to the end and wait at the implicit barrier). This may be useful for sections of code that are not thread safe (e.g., I/O).

- `SECTIONS`, which sets up blocks of code specifically for different threads; it is a *non-iterative* work-sharing construct.

There is also a `PARALLEL` variant of the `SECTIONS`.

Its behaviour is pretty much as expected, allowing threads to execute their sections in parallel and giving the programmer finer control over how threads are scheduled to do so.

## Work-sharing constructs

```
PROGRAM VEC_ADD_SECTIONS

  INTEGER N, I
  PARAMETER (N=1000)
  REAL A(N), B(N), C(N), D(N)

  !     Some initializations
  DO I = 1, N
     A(I) = I * 1.5
     B(I) = I + 22.35
  ENDDO

  !$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)
  !$OMP SECTIONS
  !$OMP SECTION
  DO I = 1, N
     C(I) = A(I) + B(I)
  ENDDO

  !$OMP SECTION
  DO I = 1, N
     D(I) = A(I) * B(I)
  ENDDO

  !$OMP END SECTIONS NOWAIT
  !$OMP END PARALLEL

END PROGRAM VEC_ADD_SECTIONS
```

## *More on variable scope*

OpenMP has other clauses that can change how variables are shared between threads.

The most common ones are:

- `FIRSTPRIVATE(LIST)`, in which new private variables for each thread are created for each name in `LIST`, but instead of being undefined, the newly created variables take on the values that were bound to the name in the region before this clause.

- `LASTPRIVATE(LIST)`, in which new private variables for each thread are created for each name in `LIST`, but instead of being undefined, the value of the private variables from the *sequentially last* loop iteration are copied into the variable bound to the name in the region after this clause.

Both the `FIRSTPRIVATE` and `LASTPRIVATE` clauses can be used in one construct.

## ***Scheduling***

One key to achieving maximum performance from a parallel program is to ensure the load is balanced between the threads as much as possible.

OpenMP tries to do this for the programmer, but the best results are achieved only when the programmer gets dirty hands by telling the compiler precisely how to divide loop iterations among threads.

This is accomplished via a `SCHEDULE` clause added to the `DO` work-sharing construct.

The Fortran syntax is

```
!$OMP DO SCHEDULE(SCHED[, chunk])
```

where `SCHED` is one of `STATIC`, `DYNAMIC`, `GUIDED`, or `RUNTIME`, and `chunk` is an optional integer argument.

## ***Scheduling***

When we first introduced the `parallel for` directive, we noted that the precise assignment (or *scheduling*) of loop iterations to threads was system dependent; typically it was (approximately) a block partition.

We have also noted that it is not hard to imagine when such a schedule would be decidedly sub-optimal; e.g., if iterations became significantly more expensive with iteration number.

An alternative method of scheduling threads is *cyclic*, i.e., in a round-robin fashion.

A good schedule can have a dramatic effect on performance!

In OpenMP, finer control over how loop iterations are assigned to threads in a `parallel for` or `for` directive is achieved via the `schedule` clause.

## *Scheduling*

If we do nothing special, the default schedule is used:

```
   sum = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: sum)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

To get a cyclic schedule, we add a `schedule` clause to the `parallel for` directive:

```
   sum = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: sum) schedule(static,1)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

## *Scheduling*

The general syntax of the `schedule` clause is

```
schedule(<type> [, <chunksize>])
```

where `type` can be any one of

- `static`. Iterations assigned to threads *before* loop is executed.
- `dynamic` or `guided`. Iterations assigned to threads *while* loop is executed; e.g., after thread completes its current assignment, it can request another.
- `auto`. Schedule determined by compiler and/or run-time system.
- `runtime`. Schedule determined at run time.

## ***Scheduling***

The (optional) variable `chunksize` is a positive integer.

In OpenMP parlance, a *chunk* of iterations is a block of iterations that would be executed consecutively in the corresponding serial loop.

The number of iterations in such a block is known as the `chunksize`.

Only `static`, `dynamic`, and `guided` schedules accept a `chunksize`.

This specifies the details of schedule, but the exact interpretation depends on the `type` of schedule.

## *Scheduling*

The `static` schedule type assigns `chunksize` iterations to each thread in a round-robin fashion.

For example, if we have `total_iterations = 12` and `thread_count = 3`, then `schedule(static,1)` produces the assignment

```
Thread 0: Iterations 0, 3, 6,  9
Thread 1: Iterations 1, 4, 7, 10
Thread 2: Iterations 2, 5, 8, 11
```

and `schedule(static,2)` produces the assignment

```
Thread 0: Iterations 0, 1,  6,  7
Thread 1: Iterations 2, 3,  8,  9
Thread 2: Iterations 4, 5, 10, 11
```

etc.

When `chunksize` is not specified, it (approximately) defaults to `total_iterations/thread_count`.

## *Scheduling*

In a `dynamic` schedule, each thread executes a chunk `chunksize` (consecutive) iterations, but when a thread finishes with a given chunk, it requests another from the run-time system.

This thread repeats until all iterations are completed.

When `chunksize` is not specified, it defaults to 1.

In a `guided` schedule, chunks are executed and requests for new ones are made as with a `dynamic` schedule; however, the `chunksize` *decreases* as chunks are completed, roughly according to the formula (unassigned iterations) / (number of threads).

When `chunksize` is specified, chunk sizes decrease to it (except for possibly the last one, of course).

When `chunksize` is not specified, the chunk sizes decrease to 1.

## *Scheduling*

The `runtime` schedule uses the environment variable `OMP_SCHEDULE` to determine how to schedule the iterations in a loop.

It can be set to any allowable value for a `static`, `dynamic`, or `guided` schedule; e.g., if we execute

```
export OMP_SCHEDULE="static,1"
```

then a `parallel for` directive modified by `schedule(runtime)` will be equivalent to the modification `schedule(static,1)`.

## ***Scheduling: Which one to choose?***

There is overhead associated with the various `schedule` clauses, so some care must be taken in choosing one.

Generally, the overhead for the `static` schedule is the least, followed by `dynamic`, and then `guided`.

A good rule of thumb is not to alter the default schedule if you are content with the performance.

If you suspect substantial improvements can be made by altering it, some experimentation may uncover a better schedule.

The optimal schedule may depend on the specific number of both the threads and the total iterations.

It may also be the case that the loop simply does not parallelize very well.

## ***Scheduling: Which one to choose?***

Some guidelines when exploring for optimal schedules are as follows.

- If each iteration has roughly the same computational cost, the default schedule will likely be hard to beat.
- If the cost of each iteration changes linearly, then a `static` schedule with small chunk sizes should perform well.
- If the cost of a given iteration is unknown or highly variable, a good dose of broad experimentation might be necessary. This process can be facilitated through the use of the `runtime` schedule so that different schedules can be implemented without having to alter (and recompile, etc.) the code.

## *Runtime Library Functions*

The syntax of OpenMP is expressed through compiler directives as much as possible.

Despite this, some features are handled via runtime library functions (accessed via `#include <omp.h>`).

- `OMP_SET_NUM_THREADS()`, which takes an `INTEGER` and requests that number of threads in subsequent parallel regions. It can only be called from serial portions of the code, and it has precedence over the `OMP_NUM_THREADS` environment variable.

- `OMP_GET_NUM_THREADS()`, which returns an `INTEGER` equal to the actual number of threads in the current team of threads.

- `OMP_GET_THREAD_NUM()`, which returns an `INTEGER` equal to the ID of the current thread. The master thread has an ID of 0. Threads are numbered from 0 to `OMP_GET_NUM_THREADS()`$-1$.

## *Runtime Library Functions*

- `OMP_GET_MAX_THREADS()` returns the maximum value that can be returned by `OMP_GET_NUM_THREADS()`.

- `OMP_GET_THREAD_LIMIT()` returns the maximum number of OpenMP threads available to a program.

- `OMP_GET_NUM_THREADS()` returns the number of threads that are available to the program.

- `OMP_GET_WTIME()` returns a double-precision value equal to the number of elapsed seconds since some point in the past; usually used in "pairs" with differences in values used to obtain the elapsed time for a block of code; designed to be *per thread* times, and so may be different across threads.

- `OMP_GET_WTICK()` returns a double-precision value equal to the number of seconds between successive clock ticks (the timer resolution).

## *Runtime Library Functions*

```
PROGRAM whoAmI

  IMPLICIT NONE
  INTEGER, PARAMETER :: P=3
  INTEGER :: RANK, THREADS
  INTEGER :: OMP_GET_THREAD_NUM, OMP_GET_NUM_THREADS

  CALL OMP_SET_NUM_THREADS(P)

  !$OMP PARALLEL PRIVATE (RANK, THREADS)
  RANK    = OMP_GET_THREAD_NUM()
  THREADS = OMP_GET_NUM_THREADS()
  PRINT*, "I am thread", RANK, "out of", THREADS
  !$OMP END PARALLEL

END PROGRAM whoAmI
```

Output:

```
 I am thread           0 out of           3
 I am thread           1 out of           3
 I am thread           2 out of           3
```

Note that any valid interleaving of the output records can occur, so just because this one happened to come out "in order" does not imply they always will.

## *Synchronization*

Many OpenMP programs can be written using only the `PARALLEL` and `PARALLEL DO` constructs.

However, sometimes the programmer needs finer control over how variables are shared.

The programmer must ensure that threads do not interfere with each other so that the output does not depend on how the individual threads are scheduled.

In particular, the programmer must manage threads so that they read the correct values of a variable and that multiple threads do not try to write to a variable at the same time.

The major synchronization constructs in OpenMP include the following.

## *Synchronization*

- `MASTER` specifies a region to be executed only by the master thread. All other threads skip this region. There is no implied barrier. The Fortran syntax is

  ```
  !$OMP MASTER
  ...
  !$OMP END MASTER
  ```

- `CRITICAL` defines a section for mutual exclusion. The Fortran syntax is

  ```
  !$OMP CRITICAL [name]
  ...
  !$OMP END CRITICAL [name]
  ```

  where `[name]` is an optional identifier that enables different `CRITICAL` regions to exist. The identifiers are global; different `CRITICAL` regions with the same `name` are treated as the same section, as are `CRITICAL` sections with no name.

## *Synchronization*

- `FLUSH` defines a point at which memory consistency is enforced. This can be a tricky concept. Basically, values for variables can be held in registers or buffers before they are written to main memory, and hence different threads may see different values of a given variable. The `FLUSH` command ensures all threads see the same value. The Fortran syntax is

  ```
  !$OMP FLUSH [(list)]
  ```

  where `(list)` is a comma-separated list of variables to be flushed. In the absence of a `list`, all variables visible to the calling thread are flushed. It is rare that programmers need to explicitly call `FLUSH` because OpenMP does it automatically when it makes sense, e.g., upon entry to and exit from `parallel` and `critical` sections, etc.

## *Synchronization*

- `BARRIER` defines a synchronization point at which all threads wait for each thread in the team to arrive before continuing. The Fortran syntax is

  ```
  !$OMP BARRIER
  ...
  !$OMP END BARRIER
  ```

  A `BARRIER` can be added explicitly, but sometimes one is implied, e.g., at the end of work-sharing constructs. A `BARRIER` implies a `FLUSH`.

- `ORDERED` specifies that iterations of the `DO` / `for` loop enclosed are to be executed in the same order as if they were executed in serial. Threads wait before executing their chunk of iterations until previous iterations have completed.

## *Synchronization*

In the `bigComp` program, it was necessary to store the results from `BIG_COMP` in an array `RES` because it mattered in which order `RES` was calculated.

However, this is not usually the case in practice.

We can safely keep them in one loop if we enforce mutual exclusion, e.g., using the `CRITICAL` construct.

```
PROGRAM bigComp4
  INTEGER, PARAMETER :: N = 1000
  DOUBLE PRECISION :: ANSWER = 0.0D0, BIG_COMP
  INTEGER :: I

  !$OMP PARALLEL DO PRIVATE(RES)
  DO I=1,N
     RES = BIG_COMP(I)
     !$OMP CRITICAL
     COMBINE(ANSWER, RES)
     !$OMP END CRITICAL
  END DO
  !$OMP END PARALLEL DO

  PRINT*, ANSWER

END PROGRAM bigComp4
```

Note the declaration of `RES` as a `PRIVATE` variable.

## ***Summary***

- Philosophy: sequential equivalence, incremental parallelism
- Compiler directives
- Data scope, data dependencies, work-sharing
- Scheduling, load-balancing, synchronization