Quick Review

Fortran Directives Format

Format: (case insensitive)

sentinel	directive-name	[clause ...]
All Fortran OpenMP directives must begin with the sentinel: **!$OMP** C/C++ OpenMP directives start with **#pragma omp**	A valid OpenMP directive. Must appear after the sentinel and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.

Quick Review

General Rules

- Comments can not appear on the same line as a directive
- Only one directive-name may be specified per directive
- Fortran compilers which are OpenMP enabled generally include a command line option which instructs the compiler to activate and interpret all OpenMP directives.
- Several Fortran OpenMP directives come in pairs:

  ```
  !$OMP directive
  …
  …
  !$OMP end directive.     (The "end" directive is optional but advised for readability.)
  ```
- In C, Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.
- OpenMP code is running on a single node, ibrun is only required for jobs running over multiple nodes. For our current purposes, run it as you would a normal compiled program.

The Parallel Construct

The first OpenMP construct that appears in your program is the *parallel* construct. It is when the*parallel* construct is encountered that new threads are started. These threads will continue in existence until the parallel region for which they were started comes to an end. A parallel region is delimited by a block of code enclosed in `{}` in C/C++ and by an **`end parallel`** construct in Fortran.

It is illegal to branch into or out of a parallel region. If a thread should happen to terminate in a parallel region, all the rest of the threads will be killed.

The Parallel Construct

The behavior of the parallel construct can be influenced by three control variables that an OpenMP runtime maintains. These variables are accessible only through utility functions and environment variables. The environment variables are listed in the table below in ALL CAPS.

Control variable	Ways to modify	Way to query*	Default
nthreads	OMP_NUM_THREADS **`omp_set_num_threads()`**	**`omp_get_max_threads()`**	vendor-defined
dynamic	OMP_DYNAMIC **`omp_set_dynamic()`**	**`omp_get_dynamic()`**	vendor-defined
nesting	OMP_NESTED **`omp_set_nested()`**	**`omp_get_nested()`**	false

*Note that omp_get_max_threads() is the appropriate query for finding the value of the *nthreads* control variable; omp_get_num_threads() returns the number of threads in the currently executing parallel region, which may be less

The Parallel Construct

The *parallel* construct can be combined with one of the work-sharing constructs that will be described later. For now, we will focus on the parallel construct itself. The parallel construct begins with the OpenMP sentinel and can be followed on the same line (or continuation of that line) by a number of modifiers in *clauses*.

The Parallel Construct, Examples

Examples:

```
!$OMP PARALLEL [clause[, clause]]

....

!$OMP END PARALLEL

or

#pragma omp parallel [clause[, clause]]

{}
```

combined with a work-sharing construct:

```
c$OMP PARALLEL DO [clause[, clause]]

DO I=1,1000000

....

END DO

c$OMP END PARALLEL DO [nowait]
```

or a C example:

```
#pragma omp parallel for [clause[, clause]]

for (i=0; i<1000000; i++)

{}
```

The Parallel Construct - Clauses

The available clauses are:

- if(*scalar expression*)
- private(*list*)
- firstprivate(*list*)
- default(*private*|*shared*|*none*)
- shared(*list*)
- copyin(*list*)
- reduction(*operator*:*list*)
- num_threads(*integer expression*)

The Parallel Construct - Clauses

- The *if* clause, helps to determine whether the parallel construct will be used. There are two situations where the parallel construct would not be used: if the scalar expression in this clause evaluates to *false* or if the parallel construct appears while another parallel construct is active (the OpenMP nesting flag is off, Note: that the standard specifies that nesting is off by default.)
- The *private* clause is followed by a list of variables that will be instantiated separately for each thread in the parallel region. The type of each private variable is determined by its type in the enclosing context, but its value is undefined at the beginning of the parallel region. Likewise, the value in the enclosing context of each variable that was declared private in a parallel region is undefined when the parallel region ends. Certain global variables cannot be listed as private because their status can't be changed to undefined.

The Parallel Construct - Clauses

- The *firstprivate* clause contains a list of variables that, like the *private* variables, are instantiated separately for each thread in the parallel region. The one difference is that instead of being undefined at the beginning of the parallel region, the value of each variable is initialized to the value it had when the parallel region was entered.
- The *default* clause sets the sharing status of any variables whose sharing status is not explicitly determined.
    - The *private* option is available only in Fortran.
    - The *none* option requires that every variable referenced in the parallel region have a sharing attribute.

The Parallel Construct - Clauses

- The *shared* clause can be used to list variables that are shared among all threads. This isn't usually necessary because most variables are shared by default.
- The *copyin* clause causes the listed variables to be copied from the master thread to all other threads in the team immediately after the threads have been created and before they do any other work. The variables in the list must also be threadprivate because if they were shared, copying would be meaningless.

The Parallel Construct - Reduction Clause

- The *reduction* clause lists variables upon which a reduction operation will be done at the end of the parallel region. The clause also specifies the operator for the reduction. A private copy of each reduction variable is created for each thread and is initialized as described in the following table. The private copies of the variables are updated as the threads execute and then the private copies from all threads are combined at the end of the parallel region. Any variable defined outside the parallel region that is involved in a reduction operation is undefined while the reduction calculation is in progress.

The Parallel Construct - Reduction Clause Operations

C op	Initial value	F op	Initial value
+	0	+	0
*	1	*	1
-	0	-	0
&	~0	.and.	.true.
|	0	.or.	.false.
^	0	.eqv.	.true.
&&	1	.neqv.	.false.
||	0	max	largest neg #
		min	largest pos #
		iand	all bits on
		ior	0
		ieor	0

*The *num_threads* expression, if present, overrides any other specification of the number of threads to start for this parallel region.

The Parallel Construct - Exercise 1

Write a program that performs vector addition.

- Create two single dimensioned arrays, populate them with random numbers.
- Create a loop that sums the two arrays (vectors) together and stores them in a third array (vector).
- add a time function, and record start time and end time of your loop and how long the loop took to process.
- run this for 10, 100, 1000, 10000, and 100000 elements

How were your running times?

Now, put a simple parallel directive around your loop, and rerun. Has this affected the processing time?

Now, rewrite your code so each thread has a balanced number of elements added within its loop. Run this with 8, 16, 32, and 64 threads. How has this affected the processing time.

Finally, for the fun of it, initialize and populate your table in parallel as well, and rerun with 8, 16, 32, and 64 threads.

Note: Do this with shared memory.

The Parallel Construct - Exercise 2 - Homework

Write a program that performs matrix multiplication.

- Create two double dimensioned arrays, populate them with random numbers.
- Create a set of nested loops that multiplies the two arrays (your matrices) together and stores them in a third double dimensioned array.
- add a time function, and record start time and end time of your loop and how long the loop took to process.
- run this for 10, 100, 1000, and 10000 elements

How were your running times?

Now, put a simple parallel directive around your nested loops, and rerun. Has this affected the processing time?

Now, rewrite your code so each thread has a balanced number of elements added within its loop. Run this with 8, 16, 32, and 64 threads. How has this affected the processing time.

Finally, for the fun of it, initialize and populate your table in parallel as well, and rerun with 8, 16, 32, and 64 threads.

Note: Do this with shared memory.