# Parallel Processing Thread Model

This topic explains the processing of the parallelized program and adds more definitions of the terms used in the parallel programming.

## The Execution Flow

A program containing OpenMP C++ API compiler directives begins execution as a single process, called the master thread of execution. The master thread executes sequentially until the first parallel construct is encountered.

In the OpenMP C++ API, the #pragma omp parallel directive defines the parallel construct. When the master thread encounters a parallel construct, it creates a team of threads, with the master thread becoming the master of the team. The program statements enclosed by the parallel construct are executed in parallel by each thread in the team. These statements include routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent includes the static extent as well as the routines called from within the construct. When the #pragma omp parallel directive is encountered, the threads in the team synchronize at that point, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state. You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

### General Performance Guidelines

For applications where the workload depends on application input that can vary widely, delay the decision about the number of threads to employ until runtime when the input sizes can be examined. Examples of workload input parameters that affect the thread count include things like matrix size, database size, image/video size and resolution, depth/breadth/bushiness of tree based structures, and size of list based structures. Similarly, for applications designed to run on systems where the processor count can vary widely, defer the number of threads to employ decision till application run-time when the machine size can be examined.

For applications where the amount of work is unpredictable from the input data, consider using a calibration step to understand the workload and system characteristics to aid in choosing an appropriate number of threads. If the calibration step is expensive, the calibration results can be made persistent by storing the results in a permanent place like the file system. Avoid creating more threads than the number of processors on the system, when all the threads can be active simultaneously; this situation causes the operating system to multiplex the processors and typically yields sub-optimal performance.

When developing a library as opposed to an entire application, provide a mechanism whereby the user of the library can conveniently select the number of threads used by the library, because it is possible that the user has higher-level parallelism that renders the parallelism in the library unnecessary or even disruptive.

Finally, for OpenMP, use the num_threads clause on parallel regions to control the number of threads employed and use the if clause on parallel regions to decide whether to employ multiple threads at all. The omp_set_num_threads function can also be used but it is not recommended except in specialized well-understood situations because its affect is global and persists even after the current function ends, possibly affecting parents in the call tree. The num_threads clause is local in its effect and so does not impact the calling environment.

## Using Orphaned Directives

In routines called from within parallel constructs, you can also use directives. Directives that are not in the lexical extent of the parallel construct, but are in the dynamic extent, are called orphaned directives. Orphaned directives allow you to execute major portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines. For example:

### Example 1

```
int main(void)
{
  ...
  #pragma omp parallel
  {
    phase1();
  }
}

void phase1(void)
{
  ...
  #pragma omp for private(i) shared(n)
  for(i=0; i < n; i++)
  {
    some_work(i);
  }
}
```

This is an orphaned directive because the parallel region is not lexically present.

### Data Environment Directive

A data environment directive controls the data environment during the execution of parallel constructs.

You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can:

- Privatize named common blocks by using THREADPRIVATE directive
- Control data scope attributes by using the THREADPRIVATE directive's clauses.
- The data scope attribute clauses are:

- COPYIN
- DEFAULT
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- SHARED

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute clause on a directive, the default is SHARED for those variables affected by the directive.

## Example 2: Pseudo Code of the Parallel Processing Model

```
main() {                            // Begin serial execution
...                                 // Only the master thread executes

#pragma omp parallel                // Begin a Parallel Construct, form
{                                   // a team. This is Replicated Code
  ...                               // (each team member executes
  ...                               // the same code)

  #pragma omp sections              // Begin a Worksharing Construct
  {
    #pragma omp section             // One unit of work
    {...}
    #pragma omp section             // Another unit of work
    {...}
  }                                 // Wait until both units of work complete
  ...                               // More Replicated Code

  #pragma omp for nowait            // Begin a Worksharing Construct
  for(...)
  {                                 // Each iteration is unit of work
    ...                             // Work is distributed among the team members
  }                                 // End of Worksharing Construct
                                    // nowait was specified, so threads proceed

  #pragma omp critical              // Begin a Critical Section
  {
    ...                             // Replicated Code, but only one
                                    // thread can execute it at a
  }                                 // given time

  ... // More Replicated Code

  #pragma omp barrier               // Wait for all team members to arrive

  ... // More Replicated Code

} // End of Parallel Construct
// disband team and continue serial execution
... // Possibly more Parallel constructs
} // End serial execution
```