Chapter 2. Automatic Parallelization #pragma Directives

Table 2, lists the #pragma directives discussed in this chapter, along with a brief description of each and the compiler versions in which the directive is supported.

Table 2. IRIS Power C Analyzer #pragma Directives

#pragma	Short Description	Compiler Versions
#pragma concurrent	Tells the compiler to ignore assumed dependences in the next loop.	7.2 and lat
#pragma concurrent call	Tells the compiler that the function calls in the next loop are safe to execute in parallel.	7.2 and lat
#pragma concurrentize	Tells the compiler to parallelize the next loop, overriding any #pragma no concurrentize directive that may apply to that loop.	7.2 and lat
#pragma no concurrentize	Varies with placement. Tells the compiler to not parallelize any loops in a function or file.	7.2 and lat
#pragma permutation	The specified array is a permutation array.	7.2 and lat
#pragma prefer concurrent	Tells the compiler to parallelize the next loop if it is safe.	7.2 and lat
#pragma prefer serial	Tells the compiler to not parallelize the next loop.	7.2 and lat
#pragma serial	Forces the loop immediately following it to be serial, and restricts optimization by forcing all enclosing loops to be serial also.	7.2 and lat

2.1 #pragma concurrent

The #pragma concurrent directive instructs the compiler, when analyzing the loop immediately following this assertion, to ignore all dependences between two references to the same array.

The syntax of #pragma concurrent is as follows:

```
#pragma concurrent
```

When using this directive, be aware of the following:

- If multiple loops in a nest can be parallelized, #pragma concurrent instructs the compiler to parallelize the loop immediately following the directive.
- Applying this directive to an inner loop may cause the loop to be made outermost by the compiler's loop interchange operations.
- #pragma concurrent does not affect how the compiler analyzes function calls. See Section 2.2.
- #pragma concurrent does not affect how the compiler analyzes dependences between two potentially aliased pointers.
- If there are real dependences between array references, #pragma concurrent may cause the compiler to generate incorrect code.

2.2 `#pragma concurrent call`

The `#pragma concurrent call` directive instructs the compiler to ignore the dependences of any function calls contained in the loop that follows the directive.

The syntax for `#pragma concurrent call` is as follows:

```
#pragma concurrent call
```

This directive applies to the loop that immediately follows it and to all loops nested inside that loop.

To prevent incorrect parallelization, make sure the following conditions are met when using `#pragma concurrent call`:

- A function inside the loop cannot read from a location that is written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.
- A function inside the loop cannot write to a location that is read from or written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.

2.2.1 Examples of #pragma concurrent call

2.2.1.1 Example 1

In this example the compiler ignores the dependences in the function `fred()` when it analyzes the following loop:

```
#pragma concurrent call  
for (i = 0; i < N; i++0  
{  
fred(...)  
...  
}  
  
void fred (...)  
{  
...  
}
```

2.2.1.2 Example 2

The following code shows an illegal use of the assertion. Function `fred()` writes to variable *T,* which is also read by `wilma()` during other iterations.

```
float A[M], B[M];  
int i, T;  
#pragma concurrent call  
for (i = 0; i < M; i++)  
{  
fred(B, i, &T);  
wilma(A, i, &T);  
}  
  
void fred(float B[], int i, int* T)  
{  
*T = B[i];  
}  
  
void wilma(float A[], int i, int* T)  
{  
A[i] = *T;  
}
```

By localizing the variable *T*, you can manually parallelize the preceding example safely. But the compiler is not instructed to localize *T*, and the loop is illegally parallelized because of the assertion.

2.3 `#pragma concurrentize`

The `#pragma concurrentize` directive instructs the compiler to parallelize an entire file or function.

The syntax of `#pragma concurrentize` is as follows:

```
#pragma concurrentize
```

Placing the `#pragma concurrentize` directive inside a function overrides a `#pragma no concurrentize` directive placed outside of it. In other words, this directive allows you to selectively parallelize functions in a file that has been made sequential with `#pragma no concurrentize`.

This directive works only with the MIPSpro Automatic Parallelizer.

2.4 `#pragma no concurrentize`

The `#pragma no concurrentize` directive prevents parallelization of a file or function.

The syntax of `#pragma no concurrentize` is as follows:

```
#pragma no concurrentize
```

The effect of `#pragma no concurrentize` depends on its placement:

- When placed inside a function, the directive prevent its parallelization.
- When placed outside of a function, `#pragma no concurrentize` prevents the parallelization of all functions in the file, even those that appear ahead of it in the file.

This directive works only with the MIPSpro Automatic Parallelizer.

2.5 `#pragma permutation`

When placed inside a function, the `#pragma permutation` directive instructs the compiler that the specified array is a permutation array.

The syntax of `#pragma permutation` is as follows:

`#pragma permutation (`*`array`*`)`

array is the name of a permutation array. Every element of *array* has a distinct value. The directive does not require the permutation array to be dense. In other words, while every *array[1]* must have a distinct value, there can be gaps between those values, such as *array[1]* = 1, *array[2]* = 4, *array[3]* = 9, and so on.

You can use this assertion to parallelize loops that use arrays for indirect addressing. Without this directive, the compiler cannot determine that the array elements used as indexes are distinct.

The `#pragma permutation` directive affects every loop in a function, even those that precede it.

2.6 `#pragma prefer concurrent`

The `#pragma prefer concurrent` directive instructs the compiler to parallelize the loop immediately following the directive, if it is safe to do so.

The syntax of the `#pragma prefer concurrent` directive is as follows:

```
#pragma prefer concurrent
```

This pragma is always safe to use. The compiler parallelizes the loop only when it can determine that it is safe to do so.

When dealing with nested loops, the compiler follows these guidelines:

- If the loop specified by this directive is safe to parallelize, the compiler chooses it to parallelize, even if other loops are also candidates for parallelization.
- If the specified loop is not safe to parallelize, the compiler uses its heuristics to choose among loops that are safe.
- If this directive is applied to an inner loop, the compiler may make it the outermost loop.
- If this assertion is applied to more than one loop in a nest, the compiler uses its heuristics to choose one of the specified loops.

This directive works only with the MIPSpro Automatic Parallelizer.

2.7 `#pragma prefer serial`

The `#pragma prefer serial` directive instructs the compiler to not parallelize the loop that immediately follows it. It performs in the same way as the `#pragma serial` directive.

The syntax of `#pragma prefer serial` is as follows:

```
#pragma prefer serial
```

This directive works only with the MIPSpro Automatic Parallelizer.

2.8 `#pragma serial`

The `#pragma serial` directive instructs the compiler to not parallelize the loop following the assertion. However, the compiler may parallelize another loop in the same nest. The parallelized loop may be either inside or outside the designated sequential loop.

The syntax for this directive is as follows:

```
#pragma serial
```

This directive works only with the MIPSpro Automatic Parallelizer.