Chapter 5.

DSM Optimization #pragma Directives

Table 5, lists the #pragma directives discussed in this chapter, along with a short description of each and the compiler versions in which the directive is supported. These directives are useful primarily on systems with distributed shared memory, such as Origin servers.

Table 5. Distributed Shared Memory #pragma Directives

#pragma	Short Description	Compiler Versions
#pragma distribute	Specifies data distribution.	7.2 and la
#pragma distribute_reshape	Specifies data distribution with reshaping.	7.2 and la
#pragma dynamic	Tells the compiler that the specified array may be redistributed in the program.	7.2 and la
#pragma page_place	Allows the explicit placement of data.	7.1 and la
#pragma pfor (Discussed in Chapter 9)	affinity clause allows data−affinity or thread−affinity scheduling; nest clause exploits nested concurrency. See Section 9.11	6.0 and la
#pragma redistribute	Specifies dynamic redistribution of data.	7.2 and la

5.1 #pragma distribute

The #pragma distribute directive specifies the distribution of data across the processors. It functions by influencing the mapping of virtual addresses to physical pages without affecting the layout of the data structure. Because the granularity of data allocation is a physical page (at least 16 KB), the achieved distribution is limited by the underlying page granularity. However, the advantages to using this directive are that it can be added to an existing program without any restrictions, and can be used for affinity scheduling. See Section 9.11.4, for more information about data affinity.

The syntax of the #pragma distribute directive is as follows:

```
#pragma distribute array[dst1][[dst2]...][onto (dim1, dim2[, dim3 ...])]
```

array is the name of the array you want to have distributed.

dst is the distribution specification for each dimension of the array. It can be any one of the following:

dst is the distribution specification for each dimension of the array. It can be any one of the following:

Value	Effect
*	Not distributed.
block	Partitions the elements of an array dimension into blocks equal to the size of the dimension (*N*) divided by the number of processors (*P*). The size of each block will be equal to *N*/*P*, rounded up to the nearest integer value (ceiling (N/P)).
cyclic [(*size_expr*)]	Partitions the elements of an array dimension into chunks and distributes the chunks sequentially across the processors. The

size of the pieces is equal to the value of *size_expr*. If *size_expr* is not specified, the chunk size defaults to 1. A `cyclic` distribution with a chunk size that is either greater than 1 or is determined at run time is sometimes also called `block-cyclic`.

dim is the specification for partitioning the processors across the distributed dimensions (see Section 5.5.1, for more information).

The following is some additional information about `#pragma distribute`:

- You must specify the `#pragma distribute` directive in the declaration part of the program, along with the array declaration.
- You can specify a data distribution directive for any local or global array.
- Each dimension of a multi−dimensional array can be independently distributed.
- A distributed array is distributed across all of the processors being used in that particular execution of the program, as determined by the environment variable `MP_SET_NUMTHREADS`.

5.1.1 Example of `#pragma distribute`

The following code fragment demonstrates the use of `#pragma distribute`:

```
float A[200][300];  
...  
#pragma distribute A[cyclic][block];  
...
```

On a machine with eight processors, the first dimension of array *A* is distributed across the processors in chunks of 1, and the second dimension is distributed in chunks of 25 for each processor.

5.1.2 `onto` Clause

If an array is distributed in more than one dimension, then by default the processors are apportioned as equally as possible across each distributed dimension. For instance, if an array has two distributed dimensions, then an execution with 16 processors assigns 4 processors to each dimension ($4 \times 4 = 16$), whereas an execution with 8 processors assigns 4 processors to the first dimension and 2 processors to the second dimension.

You can override this default and explicitly control the number of processors in each dimension by using the `onto` clause. The `onto` clause allows you to specify the processor topology when an array is being distributed in more than one dimension. For instance, if an array is distributed in two dimensions, and you want to assign more processors to the second dimension than to the first dimension, you can use the `onto` clause as in the following code fragment:

```
float A[100][200];  
  
/* Assign to the second dimension twice as many processors as to  
the first dimension. */  
  
#pragma distribute A[block][block] onto (1, 2)
```

5.2 `#pragma distribute_reshape`

The `#pragma distribute_reshape` directive, like `#pragma distribute`, specifies the desired distribution of an array. In addition, however, the `#pragma distribute_reshape` directive declares that the program makes no assumptions about the storage layout of that array. The compiler performs aggressive optimizations for reshaped arrays that violate standard layout assumptions but guarantee the desired data distribution for that array.

For information about using data affinity with `#pragma redistribute-reshape`, see Section 9.11.4.

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

```
#pragma distribute_reshape array[dst1][[dst2]...]
```

The `#pragma distribute_reshape` directive accepts the same distributions as the `#pragma distribute` directive:

- *array* is the name of the array you want to have distributed.
- *dst* is the distribution specification for each dimension of the array. It can be any one of the following:

Value	Effect
`*`	Not distributed.
`block`	Partitions the elements of an array dimension into blocks equal to the size of the dimension (*N*) divided by the number of processors (*P*). The size of each block will be equal to *N*/*P*, rounded up to the nearest integer value (`ceiling (N/P)`).
`cyclic` (*size_expr*)	Partitions the elements of an array dimension into chunks and distributes the chunks sequentially across the processors. The size of the pieces is equal to the value of *size_expr*. If *size_expr* is not specified, the chunk size defaults to 1. A `cyclic` distribution with a chunk size that is either greater than 1 or is determined at run time is sometimes also called `block-cyclic`.

The following is some additional information about `#pragma distribute_reshape`:

- You must specify the `#pragma distribute_reshape` directive in the declaration part of the program, along with the array declaration.
- You can specify a data distribution directive for any local or global array.
- Each dimension of a multi−dimensional array can be independently distributed.
- A distributed array is distributed across all of the processors being used in that particular execution of the program, as determined by the environment variable `MP_SET_NUMTHREADS`.
- A reshaped array is passed as an actual parameter to a subroutine, in which case two possible scenarios exist:

– The array is passed in its entirety (`func(A)` passes the entire array `A`, whereas `func(A([i][j])` passes a portion of `A`). The C compiler automatically clones a copy of the called function and compiles it for the incoming distribution. The actual and formal parameters must match in the number of dimensions, and the size of each dimension.

  The C++ compiler does not perform this cloning automatically, due to interactions in the compiler with the C++ template instantiation mechanism. For C++, therefore, the user has the following two options:

1. The first option is to specify `#pragma distribute_reshape` directly on the formal parameter of the called function.
2. The second option is to compile with `-MP:clone=on` to enable automatic cloning in C++.

   Caution: This option may not work for some programs that use templates.

– You can restrict a function to accept a particular reshaped distribution on a parameter by specifying a `#pragma distribute_reshape` directive on the formal parameter within the function. All calls to this function with a mismatched distribution will lead to compile− or link−time errors.

– A portion of the array can be passed as a parameter, but the callee must access only a single processor's portion. If the callee exceeds a single processor's portion, then the results are undefined. You can use intrinsics to access details about the array distribution (see the "Parallel Programming on Origin Servers" chapter in the *C Language Reference Manual* for more details).

  Caution: Because the `#pragma distribute_reshape` directive specifies that the program does not depend on the storage layout of the reshaped array, restrictions on reshaping arrays include the following (for more details on reshaping arrays, see the *C Language Reference Manual*):

- The distribution of a reshaped array cannot be changed dynamically (that is, there is no `#pragma redistribute_reshape` directive).
- Initialized data cannot be reshaped.
- Arrays that are explicitly allocated through `alloca/malloc` and accessed through pointers cannot be reshaped. Use variable length arrays instead.
- An array that is equivalenced to another array cannot be reshaped.
- A global reshaped array cannot be linked `-Xlocal`. This user error is not caught by the compiler or linker.

5.2.1 Example of `#pragma distribute_reshape`

The following code fragment demonstrates the use of `#pragma distribute_reshape`:

```
float A[400][300];  
...  
#pragma distribute_reshape A[block][cyclic(3]);  
...
```

On a machine with eight processors, the first dimension of array A is distributed in chunks of 50 for each processor, and the second dimension is distributed across the processors in chunks of 3.

5.3 `#pragma dynamic`

By default, the compiler assumes that a distributed array is not dynamically redistributed, and directly schedules a parallel loop for the specified data affinity. In contrast, a redistributed array can have multiple possible distributions, and data affinity for a redistributed array must be implemented in the run−time system based on the particular distribution.

The `#pragma dynamic` directive notifies the compiler that the named array may be dynamically redistributed at some point in the run. This tells the compiler that any data affinity for that array must be implemented at run time. For information about using data affinity with `#pragma dynamic`, see Section 9.11.4.

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

`#pragma dynamic` *`array`*

array is the name of the array in question.

The `#pragma dynamic` directive informs the compiler that *array* may be dynamically redistributed. Data affinity for such arrays is implemented through a run−time lookup. Implementing data affinity in this manner incurs some extra overhead compared to a direct compile−time implementation, so you should use the `#pragma dynamic` directive only if it is actually necessary.

You must explicitly specify the `#pragma dynamic` declaration for a redistributed array under the following conditions:

- The function contains a `pfor` loop that specifies data affinity for the array.
- The distribution for the array is not known.

Under the following conditions, you can omit the `#pragma dynamic` directive and just supply the `#pragma distribute` directive with the particular distribution:

- The function contains data affinity for the redistributed array.
- The array has a specified distribution throughout the duration of the function.

Because reshaped arrays cannot be dynamically redistributed, this is an issue only for regular data distribution.

5.4 `#pragma page_place`

The `#pragma page_place` directive is useful for dealing with irregular data structures. It allows you to explicitly place data in the physical memory of a particular processor. This directive is often used in conjunction with thread affinity (see Section 9.11.4, for more information).

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

`#pragma page_place (`*`object`*`, `*`size`*`, `*`threadnum`*`)`

1. *object* is the object you want to place
2. *size* is the size in bytes
3. *threadnum* is the number of the destination processor

On a system with physically distributed shared memory, you can explicitly place all data pages spanned by the virtual address range `[&object, &object+ size-1]` in the physical memory of the processor corresponding to the specified thread. This directive is an executable statement; therefore, you can use it to place either statically or dynamically allocated data.

The function `getpagesize()` can be invoked to determine the page size. On the Origin2000™ server, the minimum page size is 16384 bytes.

5.4.1 Example of `#pragma page_place`

The following is an example of the use of `#pragma page_place`:

```
double A[8192];  
#pragma page_place (A[0], 32768, 0)  
#pragma page_place (A[4096], 16384, 1)
```

The first `#pragma page_place` directive causes the first half of the array to be placed in the physical memory associated with thread 0. The second causes the next quarter of the array to be placed in the physical memory associated with thread 1. The remaining portion of *A* is allocated based on the operating system's allocation policy (default is "first–touch").

5.5 `#pragma redistribute`

The `#pragma redistribute` directive allows you to dynamically redistribute previously distributed arrays. For information about using data affinity with `#pragma redistribute`, see Section 9.11.4.

The syntax of the `redistribute` pragma is as follows:

`#pragma redistribute `*`array`*`[`*`dst1`*`][[`*`dst2`*`]...][onto (`*`dim1`*`, `*`dim2`*`[, `*`dim3`*` ...])]`

· *array* is the name of the array you wish to have distributed.

· *dst* is the distribution specification for each dimension of the array. It can be any one of the following:

Value	Effect
`*`	Not distributed.
`block`	Partitions the elements of an array dimension into blocks equal to the size of the dimension (*N*) divided by the number of processors (*P*). The size of each block will be equal to *N*/*P*, rounded up to the nearest integer value (`ceiling (N/P)`).
`cyclic` (*size_expr*)	Partitions the elements of an array dimension into chunks and distributes the chunks sequentially across the processors. The

size of the pieces is equal to the value of *size_expr*. If *size_expr* is not specified, the chunk size defaults to 1. A `cyclic` distribution with a chunk size that is either greater than 1 or is determined at run time is sometimes also called `block-cyclic`.

- *dim* is the specification for partitioning the processors across the distributed dimensions (see Section 5.5.1, for more information).

The following is some additional information about `#pragma redistribute`:

- It is an executable statement and can appear in any executable portion of the program.
- It changes the distribution permanently (or until another `redistribute` statement).
- It also affects subsequent affinity scheduling.

5.5.1 `onto` Clause

If an array is distributed in more than one dimension, then by default the processors are apportioned as equally as possible across each distributed dimension. For instance, if an array has two distributed dimensions, then an execution with 16 processors assigns 4 processors to each dimension ($4 \times 4 = 16$), whereas an execution with 8 processors assigns 4 processors to the first dimension and 2 processors to the second dimension.

You can override this default and explicitly control the number of processors in each dimension by using the `onto` clause. The `onto` clause allows you to specify the processor topology when an array is being distributed in more than one dimension. For instance, if an array is distributed in two dimensions, and you want to assign more processors to the second dimension than to the first dimension, you can use the `onto` clause as in the following code fragment:

```
float A[100][200];  
  
/* Assign to the second dimension twice as many processors as to  
the first dimension. */  
  
#pragma redistribute A[block][block] onto (1, 2)
```

5.5.2 Example of `#pragma redistribute`

The following code fragment demonstrates the use of `#pragma redistribute`:

```
float A[500][300];  
...  
#pragma redistribute A[cyclic(1)][cyclic (5)];  
...
```

After the `#pragma redistribute` directive, the first dimension of array *A* is distributed across the processors in chunks of 1, the second dimension in chunks of 5.