Chapter 14

General Communication

The C* communications functions we have discussed so far have required that the source and destination parallel variables be of the current shape (except for **global**, where the destination is a scalar variable), and that the communication be in regular patterns — that is, all elements transfer their values the same number of positions in the same direction. In this chapter, we introduce functions that allow communication in which:

- One of the parallel variables need not be of the current shape, and
- The communication need not be in a regular pattern.

The **get** and **send** functions described in this chapter provide communication comparable to that offered by parallel left indexing; see Chapter 10.

The **read_from_position** function described in this chapter provide communication comparable to that offered by assigning a scalar-indexed parallel variable to a scalar variable; **write_to_position** is comparable to assigning a scalar variable to a scalar-indexed parallel variable. The **read_from_pvar** function reads data from a parallel variable into a scalar array; **write_to_pvar** writes data from an array to a parallel variable.

Include the header file **<cscomm.h>** when calling any of the functions discussed in this chapter.

14.1 The make_send_address Function

Grid communication requires knowing the coordinates of parallel variable elements in the shape. More information is required for general communication.

Copyright © 1990–1993 Thinking Machines Corporation

Specifically, you need to supply a *send address* for a parallel variable element's position. This send address, along with a position's shape, uniquely identifies a position among all positions in all shapes; thus, you can use this address when an element of the current shape is communicating with an element that is of a different shape.

Use the **make_send_address** function to obtain a send address for one or more positions. **make_send_address** is an overloaded function that has different versions depending on these conditions:

- *Whether you want to return a single address or multiple addresses.* Multiple addresses are returned as a parallel variable of the current shape.
- *Whether you specify axis coordinates for the position in a stdargs list or in an array.* The choice is the same as that for the **allocate_shape** function, which we discussed in Section 9.3. If you know the rank of the position's shape, it is easier to use the stdargs version. If the rank will not be known until run time, you must use an array.

14.1.1 Obtaining a Single Send Address

To obtain a send address for a single position, use **make_send_address** with one of these formats:

```
CMC_sendaddr_t make_send_address (
    shape s,
    int axis_0_coord, ...);
```

or:

```
CMC_sendaddr_t make_send_address (
    shape s,
    int axes[]);
```

where:

s is the shape to which the position whose address you are obtaining belongs.

axis_0_coord
(in the first version) specifies the position's coordinate along axis 0. Specify as many coordinates as there are axes in the shape.

Copyright © 1990–1993 Thinking Machines Corporation

axes[ ] (in the second version) is an array that contains the position's coordinates.

The function returns a scalar value (of type **CMC_sendaddr_t**) that is the send address of the position. This address is returned even if the position is inactive.

Note that the shape you specify in the parameter list need not be the current shape.

An Example

The code below calculates the send address of position [77][44] in shape **image** and assigns this address to the scalar variable **addr**:

```
CMC_sendaddr_t addr;
addr = make_send_address(image, 77, 44);
```

14.1.2 Obtaining Multiple Send Addresses

To obtain send addresses for more than one position, use **make_send_address** with one of these formats:

```
CMC_sendaddr_t:current make_send_address(
    shape s,
    int:current axis_0_coord, ...);
```

or:

```
CMC_sendaddr_t:current make_send_address (
    shape s,
    int:current axes[]);
```

These formats are the same as the ones shown in Section 14.1.1, except that the **axis_*n*_coord** arguments take parallel **int**s of the current shape, and the function returns a parallel variable of the current shape.

The value in each element of the parallel variable you specify for an axis of shape **s** represents a coordinate along that axis. The corresponding elements of the parallel variables that represent all the axes of the shape therefore fully specify a position in shape **s**. The function returns the send address for each position specified in this way. These send addresses are returned as the values of elements of a parallel variable that is of the current shape.

Copyright © 1990–1993 Thinking Machines Corporation

For example, if you specify **p1** as the **axis** argument for a 1-dimensional shape **s**, and **[0]p1** contains the value 4, then the send address of position [4] of shape **s** is returned in element [0] of a parallel variable of the current shape.

You cannot mix scalar values and parallel values in the argument list. If you want to use a scalar value (for example, because you only want the send addresses of positions whose coordinate for axis 1 is 3), either:

- Use a separate assignment statement to assign 3 to a parallel variable; or
- Use a cast in the argument list to explicitly promote 3 to a parallel value.

When Positions Are Inactive

If a position in the current shape is inactive, that position does not participate in the operation. In other words, the function does not return the send address specified by that position's parallel variable elements.

If elements specify a position in shape **s** that is inactive, the send address for that position *is* returned.

An Example

Figure 90 shows an example of **make_send_address**, using parallel variables of the 1-dimensional shape **t** to map parallel variables of the 2-dimensional shape **s**.

Copyright © 1990–1993 Thinking Machines Corporation

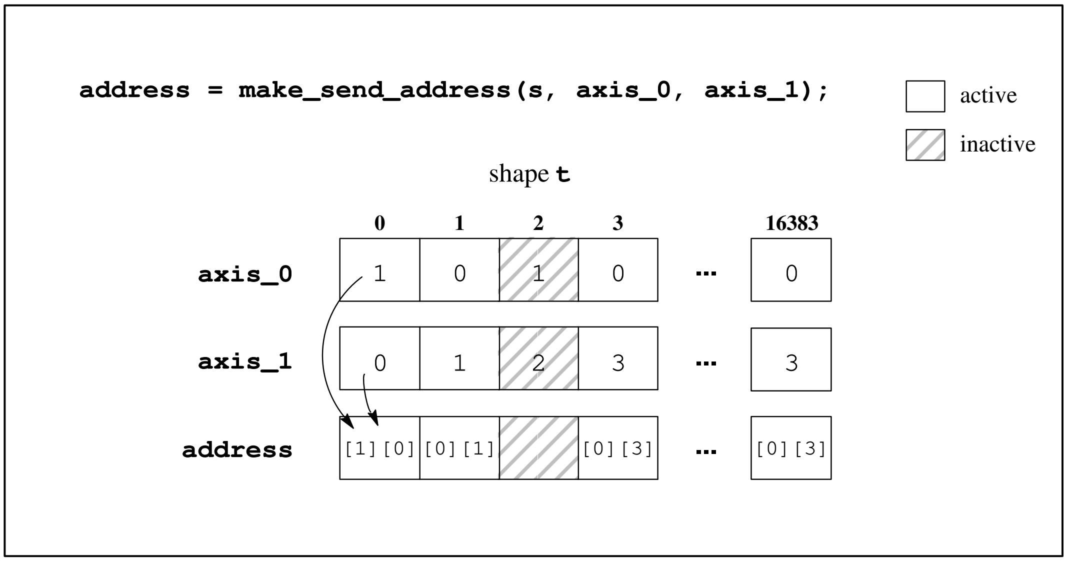

Figure 90. An example of the `make_send_address` function.

Note these points in Figure 90:

- Two elements contain the same send address; this is legal.
- Position [2] is inactive; therefore, element [2] of **`address`** does not obtain the send address specified by the values in **`[2]axis_0`** and **`[2]axis_1`**.

The values of the elements that specify coordinates for an axis must be within the range of these coordinates. If, for example, shape **`s`** has 256 positions along axis 0, an element of **`axis_0`** cannot have a value greater than 255.

14.2 Getting Parallel Data: The get Function

Use the **`get`** function to get values from a parallel variable when grid communication is not possible — that is, when communicating between shapes, or when the communication is not in a regular pattern. The **`get`** function is overloaded for both arithmetic and aggregate types.

Copyright © 1990–1993 Thinking Machines Corporation

14.2.1 Getting Parallel Variables

The **get** function has this definition when used with arithmetic types:

```
type:current get (
    CMC_sendaddr_t:current send_address,
    type:void *sourcep,
    CMC_collision_mode_t collision_mode);
```

where:

send_address
is a parallel variable of the current shape. The parallel variable contains send addresses for positions in a shape that need not be the current shape; see Section 14.1. They must, however, be of the same shape as the parallel variable pointed to by **sourcep**.

sourcep
is a scalar pointer to a parallel variable (of any shape) from which values are to be returned. The parallel variable pointed to by **send_address** specifies which values are to be returned and where they are to be assigned.

collision_mode
specifies the behavior if more than one destination parallel variable element tries to get from the same element of the source parallel variable. Possible values are **CMC_collisions**, **CMC_no_collisions**, **CMC_few_collisions**, and **CMC_many_collisions**. See "Collisions in Get Operations," below.

The **get** function returns a parallel variable of the current shape. It has the same arithmetic type as the parallel variable pointed to by **sourcep**, and it contains the values of the parallel variable pointed to by **sourcep** in the positions specified by **send_address**.

The **get** function works like a get operation using a parallel left index; see Chapter 10. A destination parallel variable obtains values of the source parallel variable, using the parallel variable **send_address** as an index. Thus, given this code:

```
#include <cscomm.h>

shape [65536]ShapeA;
shape [512][128]ShapeB;
```


Copyright © 1990–1993 Thinking Machines Corporation

```
int:ShapeA axis_0, axis_1, dest;
int:ShapeB source;
```

These two code fragments have the same results:

```
with (ShapeA) {
    CMC_sendaddr_t:ShapeA address;
    address = make_send_address(ShapeB, axis_0, axis_1);
    dest = get(address, &source, CMC_collisions);
}
```

and:

```
with (ShapeA)
    dest = [axis_0][axis_1]source;
```

The **get** function is more general, however:

- You can use **get** even if the rank of the shape from which you want to get values is not known until run time. Parallel left indexing requires that you know the rank of the shape when you write the program.
- The **get** function lets you control how collisions are handled; see below.
- The **get** function also lets you get parallel arrays. See Section 14.2.2, below.

If there are inactive positions in **ShapeA** in the first example above, elements of **dest** at these positions do not get values from **source**. The status of the positions in **ShapeB** does not matter; the active elements of **dest** get the values from the positions for which **address** has send addresses, whether or not these positions are active. Once again, this behavior is the same as that for get operations with parallel left indexing.

Collisions in Get Operations

The collisions we have talked about previously occur when two elements try to send to the same element at the same time. Get operations also have collisions, however; these occur when more than one parallel variable element tries to get a value from the same element at the same time. Unlike send collisions, get collisions are permitted in C*; they are handled automatically by get operations in the language. The **get** function and its **collision_mode** argument, however, gives you some control over how collisions are handled.

Copyright © 1990–1993 Thinking Machines Corporation

We recommend using the **CMC_collisions** option of **collision_mode** for most applications. This is the method used by get operations in the language itself. The other options may be useful in special circumstances:

- If there is no possibility of collisions, you can specify **CMC_no_collisions**; currently, this option uses the same code as **CMC_collisions**. However, future implementations of the **get** function may increase the performance of **CMC_no_collisions**.

- **CMC_many_collisions** and **CMC_few_collisions** can be useful if your application is memory-intensive and risks running out of storage. (You can determine this if, for example, your program doesn't run with a certain number of physical processors, but does run with a larger number of processors.) **CMC_collisions** requires memory for two aspects of its operation: to store the paths it takes in doing gets for each position, and to store colliding addresses. If it runs out of memory, it switches over and tries the algorithm used by **CMC_many_collisions**, which is slower but requires less memory. Under these circumstances, the operation would be faster if you specified **CMC_many_collisions** to begin with, thus avoiding the time spent trying the **CMC_collisions** algorithm.

If **CMC_collisions** takes a long time due to memory limitations and the get has few collisions, **CMC_few_collisions** may be faster. In this case, the get operation iterates separately over each collision, saving the memory required to store the colliding addresses.

14.2.2 Getting Parallel Data of Any Length

You can also use the **get** function to obtain values from parallel locations of any length — typically, parallel structures or parallel arrays.

This version of the **get** function has this definition:

```
void  get (
    void:current *destp,
    CMC_sendaddr_t:current *send_addressp,
    void:void *sourcep,
    CMC_collision_mode_t collision_mode,
    int length);
```

where:

Copyright © 1990–1993 Thinking Machines Corporation

destp is a scalar pointer to a parallel location of the current shape. This location obtains values from **sourcep**, based on the index in the parallel variable pointed to by **send_addressp**.

send_addressp
is a scalar pointer to a parallel variable of the current shape. The parallel variable contains send addresses for positions in a shape that need not be the current shape. See Section 14.1.

sourcep is a scalar pointer to a parallel location; it need not be of the current shape. The parallel variable pointed to by **send_addressp** specifies positions of this location. Data is to be gotten from these positions.

collision_mode
specifies what to do if more than one destination parallel variable element tries to get from the same element of the source parallel variable. Possible values are **CMC_collisions**, **CMC_no_collisions**, **CMC_few_collisions**, and **CMC_many_collisions**. See "Collisions in Get Operations," above.

length specifies the length in **bool**s of the parallel location pointed to by **sourcep**.

This version of the **get** function lets you obtain data that is larger than the standard data types; typically, this data would be in a parallel structure or parallel array. For example:

```
#include <cscomm.h>

shape [65536]ShapeA;
shape [512][128]ShapeB;
struct S {
    int a;
    int b;
};
int:ShapeA axis_0, axis_1;
struct S:ShapeA dest_struct;
struct S:ShapeB source_struct;

main()
{
    with (ShapeA) {
```


Copyright © 1990–1993 Thinking Machines Corporation

```
            CMC_sendaddr_t:ShapeA address;
            address = make_send_address(ShapeB, axis_0, axis_1);
            get(&dest_struct, &address, &source_struct,
                 CMC_collisions, boolsizeof(source_struct));
        }
    }
```

dest_struct, of shape **ShapeA**, gets data from individual positions of the structure **source_struct**, of shape **ShapeB**, based on the send addresses stored in **address**. Note the use of the intrinsic function **boolsizeof** to obtain the length, in **bool**s, of **source_struct**.

14.3 Sending Parallel Data: The send Function

Use the **send** function to send parallel data when grid communication is not possible — that is, when communicating between shapes, or when the communication is not in a regular pattern. The **send** function is overloaded for both arithmetic and aggregate types.

14.3.1 Sending Parallel Variables

The **send** function has this definition when used with arithmetic types:

```
type:current send (
    type:void *destp,
    CMC_sendaddr_t:current send_address,
    type:current source,
    CMC_combiner_t combiner,
    bool:void *notifyp);
```

where:

destp is a scalar pointer to a parallel variable to which values are to be sent. It can be of any arithmetic type and any shape.

send_address
is a parallel variable of the current shape. The parallel variable contains send addresses for positions in the shape

Copyright © 1990–1993 Thinking Machines Corporation

of the parallel variable pointed to by **destp**. This shape need not be the current shape; see Section 14.1.

source is a parallel variable from which values are to be sent. It must be of the current shape, and it must have the same type as the parallel variable pointed to by **destp**.

combiner specifies how **send** is to handle collisions. Possible values are **CMC_combiner_max**, **CMC_combiner_min**, **CMC_combiner_add**, **CMC_combiner_logior**, **CMC_combiner_logxor**, **CMC_combiner_logand**, and **CMC_combiner_overwrite**. All of these are defined in Section 13.1 except **CMC_combiner_overwrite**. If you specify **CMC_combiner_overwrite** and more than one value is sent to a parallel variable element, one of the values is chosen arbitrarily and stored in the element, and the rest of the values are discarded.

notifyp is a scalar pointer to a **bool**-size parallel variable of the same shape as the parallel variable pointed to by **destp**. When an element of the **destp** parallel variable receives a value, the corresponding element of the parallel variable pointed to by **notifyp** is set to 1; other elements are set to 0. If you do not want to use a notify bit, specify **CMC_no_field** for this argument.

send returns the source.

Using the **send** function is roughly equivalent to performing a send operation with parallel left indexing; see Chapter 10. The **source** parallel variable sends values to the **destp** parallel variable, using **send_address** as an index. The combiners are equivalent to reduction assignment operators. **CMC_combiner_overwrite** has the same effect as the **=** operator, when the parallel right-hand side is cast to the type of the scalar left-hand side.

There are some differences, however, between the **send** function and send operations with parallel left indexing:

- The **send** function can be used when the rank of the shape of the destination parallel variable is not known until run time.
- The **send** function lets you include a notify bit, which provides notification that a value has been received by an element of the destination parallel variable.

Copyright © 1990–1993 Thinking Machines Corporation

- There is not a complete correspondence between the combiners and the reduction assignment operators. For example, there is no combiner that is equivalent to the **-=** reduction assignment operator.
- The **send** function has an overloaded version that lets you send parallel arrays; see Section 14.3.2, below.

Inactive Positions

Inactive positions are treated in the same way they are treated by send operations with parallel left indexes:

- An element in an inactive position in the current shape does not send a value.
- Destination parallel variable elements receive values even if they are in inactive positions.

In addition, the notify bit can be set even in an inactive position.

An Example

This code sends values from elements of **source** to elements of **dest**:

```
#include <cscomm.h>

shape [16384]ShapeA;
shape [2][16384]ShapeB;
int:ShapeA axis_0, axis_1, source;
int:ShapeB dest;

/* Code to initialize parallel variables omitted. */

main()
{
    with (ShapeA) {
         CMC_sendaddr_t:ShapeA address;
         address = make_send_address(ShapeB, axis_0, axis_1);

         where (source < 9)
              send(&dest, address, source, CMC_combiner_min,
                        &notify_bit);
    }
}
```


Copyright © 1990–1993 Thinking Machines Corporation

Some sample results are shown in Figure 91. The arrows show what happens to the value at **[3]source**, based on the send address in **[3]address**.

Note these points in the results:

- Position [2] of **ShapeA** is inactive; therefore, **[2]source** does not send its value.
- The **CMC_combiner_min** combiner causes the 3 from **[0]source**, rather than the 5 from [**1]source**, to be sent to **[1][0]dest**.
- The notify bit is set in the two positions that receive values.

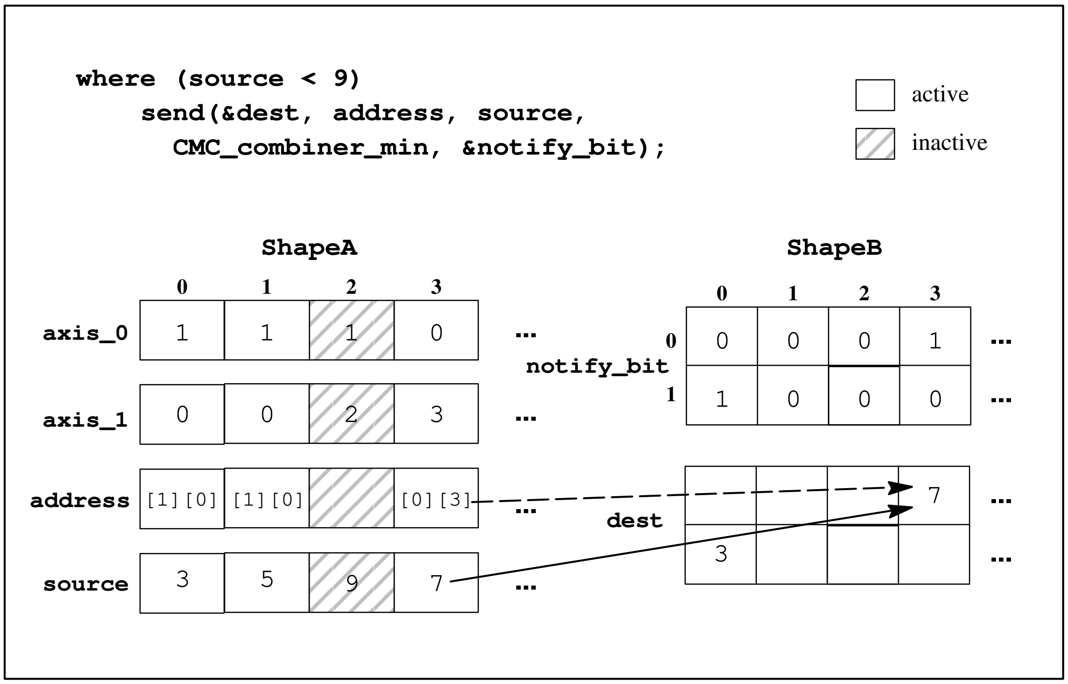

Figure 91. An example of the send function.

14.3.2 Sending Parallel Data of Any Length

You can also use the **send** function to send parallel data of any length — typically a parallel structure or parallel array.

This version of the **send** function is defined as follows:

Copyright © 1990–1993 Thinking Machines Corporation

```
void:current * send (
    void:void *destp,
    CMC_sendaddr_t:current *send_addressp,
    void:current *sourcep,
    int length,
    bool:void *notifyp);
```

where:

destp is a scalar pointer to a parallel location to which data is to be sent. **void:void** specifies that **destp** points to a location that can be of any type and of any shape.

send_addressp
is a scalar pointer to a parallel variable of the current shape. The parallel variable contains send addresses for positions in the shape of the parallel variable pointed to by **destp**.

sourcep is a scalar pointer to a parallel location from which data is to be sent. It must be of the current shape.

length specifies the length in **bool**s of the location whose beginning is pointed to by **sourcep**.

notifyp is a scalar pointer to a **bool**-sized parallel variable of the same shape as the location pointed to by **destp**. When data is written to a position pointed to by **destp**, the corresponding element of the parallel variable pointed to by **notifyp** is set to 1. If you do not want to use a notify bit, specify **CMC_no_field** for this argument.

send returns a pointer to the source.

This version of the **send** function lets you send data that is larger than the standard data types; typically, this data would be in a parallel structure or parallel array. The data is sent from the source location to the destination location, using the parallel variable pointed to by **send_addressp** as an index to determine the destination.

Note that this version of **send** does not include a **combiner** argument. This version uses the **CMC_combiner_overwrite** option, and arbitrarily chooses a position of the array or structure if there would otherwise be a collision.

For example:

Copyright © 1990–1993 Thinking Machines Corporation

```
#include <cscomm.h>

shape [65536]ShapeA;
shape [512][128]ShapeB;
struct S {
    int a;
    int b;
};
int:ShapeA axis_0, axis_1;
struct S source_struct:ShapeA, dest_struct:Shape_B;

main()
{
    with (ShapeA) {
         CMC_sendaddr_t:ShapeA address;
         address = make_send_address(ShapeB, axis_0, axis_1);
         send(&dest_struct, &address, &source_struct,
              boolsizeof(source_struct), &notify_bit);
    }
}
```

The values of individual positions of the parallel structure **source_struct**, of shape **ShapeA**, are sent to **dest_struct**, of shape **ShapeB**, based on the send addresses stored in **address**. Note the use of the intrinsic function **boolsizeof** to obtain the length, in **bool**s, of **source_struct**.

14.3.3 Sorting Elements by Their Ranks

You can use **send**, along with the **make_send_address** and **rank** functions, to reorder elements of a parallel variable by the ranks of their values. Note that this is also possible with parallel left indexing, as described in Section 13.7.1.

In the example below, we rearrange salary data for employees:

```
#include <cscomm.h>

shape [5]employees;
struct employee {
    int id;
    int salary;
    };
struct employee:employees staff;

main()
{
```


Copyright © 1990–1993 Thinking Machines Corporation

```
/* Code to initialize salaries and ids omitted. */

    with (employees) {
         int:employees order;
         CMC_sendaddr_t:employees address;

         /* Determine ranks of salary values. */

         order = rank(staff.salary, 0, CMC_upward, CMC_none,
                        CMC_no_field);

         /* Create send addresses, using salary ranks as
              the index. */

         address = make_send_address(employees, order);

         /* Send employee data for each employee to new
              positions, based on the salary ranks. */

         send(&staff, &address, &staff, boolsizeof(staff),
                   CMC_no_field);
    }
}
```

The code proceeds as follows:

1. It declares the shape, and declares and initializes the parallel structure. (The initialization of **staff.salary** and **staff.id** is omitted.)

2. It calls **rank** to return the ranks of the elements of **staff.salary**. The results are shown in Figure 92.

3. It calls **make_send_address** to return send addresses, using the salary ranks as the index. Upon return, **[0]address** contains the send address of position [1] of shape **employees**, **[1]address** contains the send address of position [0] of **employees**, and so on.

4. It then calls **send** to send the variables in the parallel structure to new positions, based on the send addresses. The result is that the values are rearranged as shown in Figure 93.

Copyright © 1990–1993 Thinking Machines Corporation

```
order = rank(staff.salary, 0, CMC_upward, CMC_none,
  CMC_no_field);
```

shape **employees**

	0	1	2	3	4
staff.id	50	51	52	53	54
staff.salary	530	230	616	614	800
order	1	0	3	2	4

Figure 92. Using the `rank` function to rank elements of a parallel variable.

```
address = make_send_address(employees, order);
send(&staff, &address, &staff, boolsizeof(staff),
    CMC_no_field);
```

shape **employees**

	0	1	2	3	4
staff.id	51	50	53	52	54
staff.salary	220	530	614	616	800
order	1	0	3	2	4

Figure 93. Using `make_send_address` and `send` to reorder the elements of parallel variables by rank.

Copyright © 1990–1993 Thinking Machines Corporation

14.4 Communicating between Scalar and Parallel Variables

This section discusses C* communication functions that provide general communication between the scalar and parallel variables.

14.4.1 From a Parallel Variable to a Scalar Variable

The read_from_position Function

Use the **read_from_position** function to read a value from a parallel variable element (not necessarily of the current shape) and assign it to a scalar variable. This function is overloaded for any arithmetic type.

The **read_from_position** function has this definition:

```
type read_from_position (
    CMC_sendaddr_t send_address,
    type:void *sourcep);
```

where:

send_address
is the send address of a position from which a value is to be read.

sourcep
is a scalar pointer to the parallel variable from which a value is to be read; the parallel variable can be of any shape and any arithmetic type.

Before calling **read_from_position** (or as part of the **read_from_position** call), you must use the single-address version of **make_send_address** to obtain a send address; see Section 14.1. The **read_from_position** function uses this send address to specify the position, and it uses **sourcep** to specify the parallel variable. It returns the value obtained from the parallel variable element at that position. The value is returned even if the position is inactive.

Since **read_from_position** deals with a scalar value, it does not have to be called within the scope of a **with** statement, and the source parallel variable does not have to be of the current shape.

Copyright © 1990–1993 Thinking Machines Corporation

This function, in combination with **make_send_address**, produces the same result as assigning a scalar-indexed parallel variable to a scalar variable. For example:

```
scalar = [7]p1;
```

You can use **read_from_position** even when the rank of the shape is not known until run time, however.

The example below reads the value from element [16][4] of parallel variable **p1**, which is of shape **image**. It assigns the value to the scalar variable **s1**.

```
#include <cscomm.h>

shape [256][256]image;
float:image p1;
CMC_sendaddr_t address;
float s1;

main()
{
    address = make_send_address(image, 16, 4);
    s1 = read_from_position(address, &p1);
}
```

Note that the call to **make_send_address** can also be made from within **read_from_position**'s argument list:

```
s1 = read_from_position(make_send_address(image, 16, 4),
                        &p1);
```

The read_from_pvar Function

Use the **read_from_pvar** function to read the values of active elements of a parallel variable and assign them to a scalar array. This function is overloaded for any arithmetic type. It has this definition:

```
void read_from_pvar (
    type *destp,
    type:current source)
```

where:

destp is a pointer to the buffer to which values are to be written.

Copyright © 1990–1993 Thinking Machines Corporation

source is a parallel variable of the current shape from which values are to be read. Both **source** and the array pointed to by **destp** must have the same arithmetic type.

The values in **source** are written into the specified scalar array. Values in inactive elements are not copied; array elements that correspond to inactive positions receive undefined values. Typically, the scalar array will have the same number of elements and dimensions as the source parallel variable. It cannot have fewer elements than the source parallel variable.

This example copies the values in **p1** to the scalar array **scalar_array**:

```
#include <cscomm.h>

shape [16384]ShapeA;
int:ShapeA p1;
int scalar_array[16384];

main()
{
    /* Initialization of p1 omitted */

    with (ShapeA)
         read_from_pvar(scalar_array, p1);
}
```

Note, however, that if the scalar array has more than one dimension, you must cast it to be a pointer to the type of the array, so that the function knows where to put the data. For example:

```
#include <cscomm.h>

shape [128][256]ShapeB;
float:ShapeB q1;
float scalar_array2[128][256];

main()
{
    /* Initialization of q1 omitted */

    with (ShapeB)
         read_from_pvar((float *)scalar_array2, q1);
}
```


Copyright © 1990–1993 Thinking Machines Corporation

Also, when there is more than one dimension involved, the data is transferred so that the highest-numbered parallel dimension is contiguous in scalar memory. In other words, the left indexes of the parallel variable match up with the right indexes of the scalar array.

Note for users of CM-5 C*: The CM-5 implementation also has a version of this function for parallel data of any length. It has this definition:

```
void read_from_pvar (
    void *destp,
    void:current *sourcep,
    int length);
```

where **`destp`** is a pointer to the scalar array to which the values are to be written, **`sourcep`** is a pointer to the parallel data, and **`length`** is the length, in units of **`bool`**s, of each data element pointed to by **`sourcep`**.

Note that using this version of **`read_from_pvar`** with aggregate data may improve performance, but it will also make your program nonportable (because of its reliance on size, alignment, and structure field padding).

14.4.2 From a Scalar Variable to a Parallel Variable

The write_to_position Function

Use the **`write_to_position`** function to write a value from a scalar variable to a parallel variable element (not necessarily of the current shape). The **`write_to_position`** function has this definition:

```
type write_to_position (
    CMC_sendaddr_t send_address,
    type:void *destp,
    type source);
```

where:

`send_address`
is the send address of the position to which a value is to be written.

`destp` is a scalar pointer to the parallel variable to which a value is to be written; the parallel variable can be of any shape and any arithmetic type.

Copyright © 1990–1993 Thinking Machines Corporation

source is the scalar variable whose value is to be sent to the destination parallel variable element. Both **source** and the parallel variable pointed to by **destp** must have the same arithmetic type.

The function returns the value of **source**.

As with **read_from_position**, you must use the single-address version of **make_send_address** to obtain a send address; see Section 14.1. **write_to_position** uses this send address to specify the position, and it uses **destp** to specify the parallel variable. It sends the value in **source** to the element specified by these arguments. The value is written into this element even if the element's position is inactive.

write_to_position does not have to be called within the scope of a **with** statement, and the destination parallel variable does not have to be of the current shape.

This function, when used along with **make_send_address**, produces the same result as assigning a scalar variable to a scalar-indexed parallel variable. For example:

```
[7]p1 = scalar;
```

You can use **write_to_position** even when the rank of the shape is not known until run time, however.

The example below reverses the example for **read_from_position** in the previous section. It assigns the value of the scalar variable **s1** to element [16][4] of parallel variable **p1**, which is of shape **image**.

```
#include <cscomm.h>

shape [256][256]image;
float:image p1;
CMC_sendaddr_t address;
float s1;

main()
{
    address = make_send_address(image, 16, 4);
    write_to_position(address, &p1, s1);
}
```


Copyright © 1990–1993 Thinking Machines Corporation

The write_to_pvar Function

Use the **write_to_pvar** function to write data from a scalar array to a parallel variable of the current shape. The function is overloaded for any arithmetic type. It has this definition:

```
type:current write_to_pvar (
    type *sourcep)
```

where **sourcep** is a pointer to a scalar array from which data is to be written.

The function returns a parallel variable of the current shape containing the values in the scalar array. If there are inactive positions in the shape at the time the function is called, the values in these inactive positions are not overwritten. The scalar array typically has the same number of elements and dimensions as the current shape; it cannot have fewer elements.

The example below reverses the example for **read_from_pvar** shown in the previous section. The array **scalar_array** writes its values to the parallel variable **p1**:

```
#include <cscomm.h>

shape [16384]ShapeA;
int:ShapeA p1;
int scalar_array[16384];

main()
{
    /* Initialization of scalar_array omitted */

    with (ShapeA)
         p1 = write_to_pvar(scalar_array);
}
```

Note once again, however, that if the scalar array has more than one dimension, you must cast it to be a pointer to the type of the array, so that the function knows where to put the data. For example:

```
#include <cscomm.h>

shape [128][256]ShapeB;
float:ShapeB q1;
float scalar_array2[128][256];

main()
```


Copyright © 1990–1993 Thinking Machines Corporation

```
{
    /* Initialization of scalar_array2 omitted */

    with (ShapeB)
         q1 = write_to_pvar((float *) scalar_array2);
}
```

Also, when there is more than one dimension involved, the data is transferred so that values that are contiguous in scalar memory become the highest-numbered dimension of the parallel variable. In other words, the right indexes of the scalar array match up with the left indexes of the parallel variable.

Note for users of CM-5 C*: The CM-5 implementation also has a version of this function for parallel data of any length. It has this definition:

```
void write_to_pvar (
    void:current *destp,
    void *sourcep,
    int length);
```

where **destp** is a pointer to the parallel data in which the values are to be written, **sourcep** is a pointer to the scalar array, and **length** is the length, in units of **bool**s, of the data pointed to by **destp**.

Note that using this version of **write_to_pvar** with aggregate data may improve performance, but it will make your program nonportable (because of its reliance on size, alignment, and structure field padding).

14.5 The make_multi_coord and copy_multispread Functions

As we mentioned in Section 13.8, the **copy_multispread** function is comparable to the **copy_spread** function, except that you use it on hyperplanes instead of scan classes.

copy_multispread takes as one of its arguments a *multicoordinate*. The multicoordinate specifies which position of the parallel variable is to be spread through each hyperplane. For example, in the discussion of **multispread** in Section 13.8, we saw that, if we allowed positions to differ along axes 0 and 1 while keeping axis 2 fixed, we created these two hyperplanes (for a 2-by-2-by-2 shape):

Copyright © 1990–1993 Thinking Machines Corporation

```
[0][0][0]
[1][0][0]
[0][1][0]
[1][1][0]
```

and:

```
[0][0][1]
[1][0][1]
[0][1][1]
[1][1][1]
```

Choosing an individual element in these hyperplanes requires that you specify only two of the three coordinates, since the third (the coordinate for axis 2) is fixed (it is [0] in the first hyperplane, [1] in the second). The multicoordinate specifies what the coordinates are along the axes that are not fixed. If the multicoordinate specifies [0] for axis 0 and [0] for axis 1, for example, then position [0][0][0] is chosen for the first hyperplane, and [0][0][1] is chosen for the second hyperplane.

To obtain this multicoordinate for a position, use the **make_multi_coord** function. You can then use the multicoordinate in the call to **copy_multispread**. The multicoordinate specifies the desired position in each hyperplane.

make_multi_coord is an overloaded function. It provides three different ways of specifying a position:

- By including the position's coordinates as arguments to the function.
- By specifying an array that contains these coordinates. Use this version if the shape's rank will not be known until run time.
- By specifying the position's send address.

The three versions of **make_multi_coord** have these definitions:

```
CMC_multicoord_t make_multi_coord (
   shape s,
    unsigned int axis_mask,
    int axis_0_coord, ... );
```

or:

```
CMC_multicoord_t make_multi_coord (
    shape s,
```


Copyright © 1990–1993 Thinking Machines Corporation

```
      unsigned int axis_mask,
      int axes[]);
```

or:

```
  CMC_multicoord_t make_multi_coord (
      shape s,
      unsigned int axis_mask,
      CMC_sendaddr_t send_address);
```

where:

s specifies the shape for which the multicoordinate is to be obtained.

axis_mask is a bit mask that specifies the axis or axes along which positions in a hyperplane are allowed to differ. Bit 1 corresponds to axis 0, bit 2 to axis 1, and so on. For example, use a bit mask of 3 to specify axes 0 and 1; use 6 to specify axes 1 and 2; use 5 to specify axes 0 and 2.

axis_0_coord
(in the first version) specifies the coordinates of a position in shape **s** along axis 0. Specify as many coordinates as there are axes in the shape.

axes[] (in the second version) is an array that contains the position's coordinates. Specify as many coordinates as there are axes in the shape.

send_address
(in the third version) is the send address for a position in shape **s**. Any position will do.

In all versions, the function returns the multicoordinate for the specified position with the specified axis mask.

The definition of **copy_multispread** is:

```
  type:current copy_multispread (
      type:current *sourcep,
      unsigned int axis_mask,
      CMC_multicoord_t multi_coord);
```

where:

Copyright © 1990–1993 Thinking Machines Corporation

sourcep is a scalar pointer to a parallel variable from which values are to be copied. The parallel variable can be of any arithmetic type; it must be of the current shape.

axis_mask is a bit mask that specifies the axis or axes along which positions in a hyperplane are allowed to differ.

multi_coord specifies the coordinates that determine the elements of the source parallel variable from which values are to be copied.

The function copies the value from each specified element to each active position in that element's hyperplane. It returns a parallel variable containing these values; the parallel variable is of the current shape and has the same arithmetic type as **source**. Values of inactive elements are copied.

14.5.1 An Example

For example, given these declarations:

```
#include <cscomm.h>

CMC_sendaddr_t address;
CMC_multicoord_t multi_coord;
shape [128][128][128]ShapeA;
int:ShapeA source, dest;
```

then:

```
address = make_send_address(ShapeA, 0, 0, 1);
```

obtains the send address for position [0][0][1] in shape **ShapeA** and assigns it to the scalar **int address**.

```
multi_coord = make_multi_coord(ShapeA, 3, address);
```

obtains the multicoordinate for this position along axes 0 and 1 (specified by the value 3 for the **axis_mask** argument) and assigns it to the **multi_coord**.

```
with (ShapeA)
  dest = copy_multispread(&source, 3, multi_coord);
```


Copyright © 1990–1993 Thinking Machines Corporation

takes each element of parallel variable **source** specified by the axis mask (3) and the multicoordinate (**multi_coord**) and copies its value into the elements of parallel variable **dest** in the same hyperplane. In other words (for a 2-by-2-by-2 shape):

- The value in **[0][0][0]source** is assigned to **[0][0][0]dest**, **[1][0][0]dest**, **[0][1][0]dest**, and **[1][1][0]dest**.
- The value in **[0][0][1]source** is assigned to **[0][0][1]dest**, **[1][0][1]dest**, **[0[1][1]dest**, and **[1][1][1]dest**.

Copyright © 1990–1993 Thinking Machines Corporation