

The C*[®] Language

James L. Frankel
Thinking Machines Corporation

Goals

- C tradition: efficient, fairly low-level, follow Standard C
- Tailored for data parallel computing
- Dynamic behavior
- Simple pointer syntax and semantics
- Allow lower-level machine access
- Allow layered object-oriented extensions

Crucial New Concepts

- Shape
- Left-indexing
- Parallel variable
- Overloading
- Context

Shapes

shape Sa, [4096]Sb, [2048][8]Sc;
shape [32][64]Sd, []Se, [][]Sf;

- fully specified
- partially specified
- fully unspecified
- left-indexing
- On the CM-2, there is a restriction that the dimension of each axis must be a power-of-two and that the total number of positions must be a multiple of the number of processors in the attached portion of the Connection Machine.

Arrays of shapes:

shape Sarray1[40];
shape []Sarray2[20];
shape [8][1024]Sarray3[4];

Parallel Variables

```
int:Sa ai1, ai2;  
int:Sb bi1, bi2;  
double ad1:Sa, bd1:Sb, bd2:Sb, bd3:Sb;
```

- The shape must be fully-specified before a parallel variable is declared of that shape.
- Unlike most C implementations, the C* compiler allocates storage for parallel variables on entry to a block. Storage for scalar variables is still allocated on entry to a function.

Shape Selection

```
with(Sb) {  
    bd1 = bd2 + bd3;  
}
```

- can nest
- dynamically bound to call chain
- can exit

Special Shapes

current

- “current” is bound at execution time to the currently selected shape.

physical

- “physical” is a one-dimensional shape whose number of positions is equal to the number of processors in the attached portion of a Connection Machine.

void

- “void” as a shape is for use with pointers.

New Operators

<? Minimum operator

- Similar to the usual macro:
`#define min(x, y) ((x) < (y)) ? (x) : (y)`
except each argument is evaluated only once.

>? Maximum operator

- Similar to the usual macro:
`#define max(x, y) ((x) > (y)) ? (x) : (y)`
except each argument is evaluated only once.

<?=
Minimum assignment operator

- Assignment version of the <? operator.

>?=
Maximum assignment operator

- Assignment version of the >? operator.

%% Real modulus operator

- The % operator is not uniquely defined when either argument is negative.

- The result has the same sign as the denominator.

Examples of Min and Max Operators

```
int i;  
i = 52 <? 17;
```

i 17

```
i = 52 >? 17;
```

i 52

```
i = -34;  
i <?= 22;
```

i -34

```
i = -34;  
i >?= 22;
```

i 22

Examples of Modulus Operator

int i;

i = 17 %% 32;

i 17

i = 32 %% 32;

i 0

i = 34 %% 32;

i 2

i = (-3) %% 32;

i 29

i = (-3) %% (-5);

i -3

i = (-5) %% (-5);

i 0

i = (-7) %% (-5);

i -2

i = 4 %% (-5);

i -1

Expression Syntax

Standard (ANSI/ISO) C

with

overloaded operators for shapes and parallel variables

para1 = para2 $\text{\textcircled{binary-op}}$ para3;

para1 = $\text{\textcircled{unary-op}}$ para2;

para1 = para2 ? para3 : para4;

Promotion

- In most cases, when a scalar variable is combined with a parallel variable, the scalar is promoted to parallel by replication.

Parallel-to-Scalar Reductions

scalar assignment-op parallel

`+=`

`--`

`|=`

`&=`

`^=`

`<?=`

`>?=`

And, in the CM-5 version only:

`*=`

`/=`

Unary Overloading of Assignment Operators

- All of the operators above may be used as unary operators if their operand is parallel.
- This unary overloading simply returns the reduction.

scalar $\dashv\equiv$ parallel;

is defined equivalent to

scalar $\dashv\equiv$ $- \dashv\equiv$ parallel; or scalar $\dashv\equiv$ $\dashv\equiv$ parallel;

Context Manipulation

“where” statement

where (*parallel-condition*)
 statement

where (*parallel-condition*)
 statement

else
 statement

- The context is maintained on a by-shape basis. Therefore, when the context is altered, operations on all parallel variables of that shape are affected.
- The “where” statement narrows the context for the duration of its nested *statement(s)*.
- Can nest
- Dynamically bound to call chain
- Can exit
- Can call functions

- Scalar code in *statement* is always executed

Context Enlargement

“everywhere” statement

everywhere
statement

- The “everywhere” statement widens the context for the duration of its nested *statement*.
 - Dynamically bound to call chain
 - Can exit
 - Can call functions
 - Scalar code in *statement* is always executed
-
- Context is not flow of control. Flow of control is affected by conditionals.
 - Use Standard C to manipulate flow of control.
 - On break, continue, goto, and return, context and shape are correctly reestablished.

Combining Conditionalization and Contextualization

if (\neq (*parallel-condition* \neq 0))
 where (*parallel-condition*)
 statement

- The **where** statement will only be executed if at least one element of the *parallel-condition* is true
- Thus, scalar code in *statement* is only executed if at least one position of the current shape remains active
- If the *parallel-condition* is known to be 0- or 1-valued, the not-equal comparison to 0 is not needed, as follows:

if (\neq *parallel-condition*)
 where (*parallel-condition*)
 statement

- If the *parallel-condition* contains side effects, the following template may be used:

if ($\neq ((\textit{parallel-temp} = \textit{parallel-condition}) \neq 0)$)
 where (*parallel-temp*)
 statement

Per-Position Iteration

```
while (|= (parallel-condition != 0))  
  where (parallel-condition)  
    statement
```

- The *statement* should cause code to be executed which will eventually decrease the positions in which the *parallel-condition* is true
- Thus, the *statement* is repeatedly executed with a gradually diminishing set of active positions
- When no more positions remain active, the **while** loop will terminate
- If the *parallel-condition* is known to be 0- or 1-valued, the not-equal comparison to 0 is not needed, as follows:

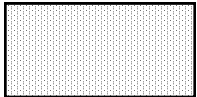
```
while (|= parallel-condition)  
  where (parallel-condition)  
    statement
```

- If the *parallel-condition* contains side effects, the following template may be used:

```
while (|= ((parallel-temp = parallel-condition) != 0))
    where (parallel-temp)
        statement
```

- This technique is similar for the other iteration statements in C: **do-while** and **for**
- Here's an example of using this technique:

```
shape [4]S;
int:S count, prod;
[0]count = 3; [1]count = 0; [2]count = 2; [3]count = 1;
prod = 1;
while (|= (count>0))
    where (count>0) {
        count--;
        prod *= 2;    ◆
    }
```

 inactive

| | 0 | 1 | 2 | 3 |
|-------|---|---|---|---|
| count | 3 | 0 | 2 | 1 |
| prod | 1 | 1 | 1 | 1 |

before first iteration

| | | | | |
|-------|---|---|---|---|
| count | 2 | 0 | 1 | 0 |
| prod | 2 | 1 | 2 | 2 |

during first iteration at ♦

| | | | | |
|-------|---|---|---|---|
| count | 1 | 0 | 0 | 0 |
| prod | 4 | 1 | 4 | 2 |

during second iteration at ♦

| | | | | |
|-------|---|---|---|---|
| count | 0 | 0 | 0 | 0 |
| prod | 8 | 1 | 4 | 2 |

during third iteration at ♦

| | | | | |
|-------|---|---|---|---|
| count | 0 | 0 | 0 | 0 |
| prod | 8 | 1 | 4 | 2 |

after while loop completes

Unary Reductions with No Positions Active

- If a unary reduction operator is called when no positions are active, the result is the identity for that operator.
- The following table lists each reduction operator and its identity.

| Operator | Identity |
|-----------|-------------------------------|
| $+=$ | 0 |
| $-=$ | 0 |
| $*=$ | 1 |
| $/=$ | 1 |
| $ =$ | 0 |
| $\&=$ | ~ 0 |
| $\wedge=$ | 0 |
| $<?=$ | largest representable number |
| $>?=$ | smallest representable number |

Short Circuit Operators

- Several operators in C have so-called short circuit behavior.
- These are the `&&`, `||`, and `?:` operators.
- The `&&` and `||` operators don't evaluate their second argument unless necessary. The `?:` operator evaluates either its second or third argument, as appropriate.
- If either of the operands to `&&` and `||` is parallel, the other operand is promoted to parallel and the parallel overloading of the operator is applied. If the first operand to `?:` is parallel, both the second and third operands are promoted to parallel and the parallel overloading of the `?:` operator is applied. If the first operand to `?:` is scalar, the scalar overloading of the `?:` operator is applied and, in addition, if either of the second and third operands are parallel, the other is promoted to parallel.
- The parallel overloadings of these operators perform contextualization in the same way that the scalar overloadings of these operators perform conditionalization.

Intrinsic Functions

- Intrinsic functions have function-like syntax, but knowledge of the intrinsic is required by the compiler.
 - For example, an intrinsic might be able to be called where a function normally could not appear.
- `rankof`
 - `rankof` takes one argument: a parallel variable or a shape.
 - It returns the rank (number of dimensions) of its argument.
- `dimof`
 - `dimof` takes two arguments: a parallel variable or a shape and an axis number.
 - Axes are numbered from left to right starting at zero.
 - It returns the dimension of the specified axis.
- `positionsof`

- `positions` takes one argument: a parallel variable or a shape.
- It returns the total number of positions (the product of all dimensions) of its argument.

“pcoord”

with shape Sc current:

pcoord(0) =

shape [2048][8]Sc;

| | | | | | | | |
|------|------|------|------|------|------|------|------|
| 2048 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | ... | | | | | | |
| 2047 | 2047 | 2047 | 2047 | 2047 | 2047 | 2047 | 2047 |

pcoord(1) =

shape [2048][8]Sc;

| | | | | | | | | |
|------|-----|---|---|---|---|---|---|---|
| 2048 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | ... | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

Left-indexing

[scalar]parallel yields a scalar

where “parallel” is a parallel variable of rank 1

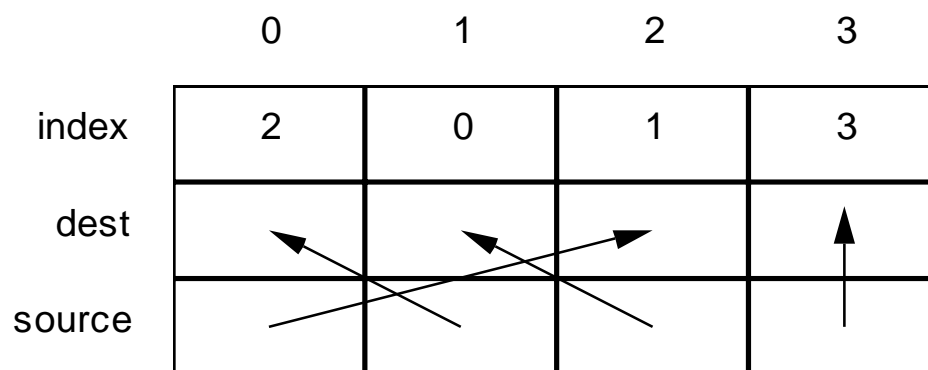
- Moves scalar data to or from an element of a parallel variable.
- In general, for a rank n parallel variable, all n left-indices must be scalar to achieve this effect.
- A parallel variable may be scalar left-indexed without requiring that the shape of the parallel variable is the current shape. Furthermore, no shape need be current to perform scalar left-indexing.
- May be used in conjunction with right indices if the variable being indexed is an array of parallel variables.

General Communication

- Extend left index syntax to accept parallel indices.
- Parallel left indexes act like vector-valued subscripts.
- Subscripts give a mapping from the shape of the parallel variable being indexed to the shape of the index.
- The shape of a parallel-left-indexed parallel variable is the shape of the index.
- The parallel left-indexes must be of the current shape, but the parallel variable being indexed may be of any shape.
- If some left-indexes are parallel and others are scalar, the scalar left-indexes are promoted to parallel.

Parallel Left-index on Left-Hand-Side

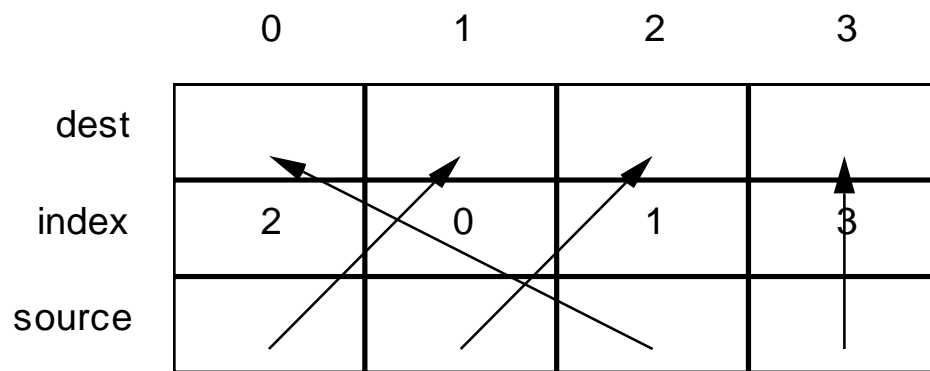
`[index]dest = source;`



- This operation is a “send.”

Parallel Left-index on Right-Hand-Side

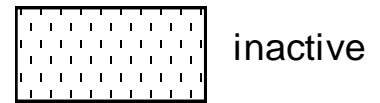
`dest = [index]source;`









- This operation is a “get.”
- A “get” is roughly twice as expensive in time as a “send.”
- A “get” requires much more storage than does “send.” The “get” needs to store a backward routing path.

Parallel Left-index on LHS with Inactivity

where(index != 1)
[index]dest = source;



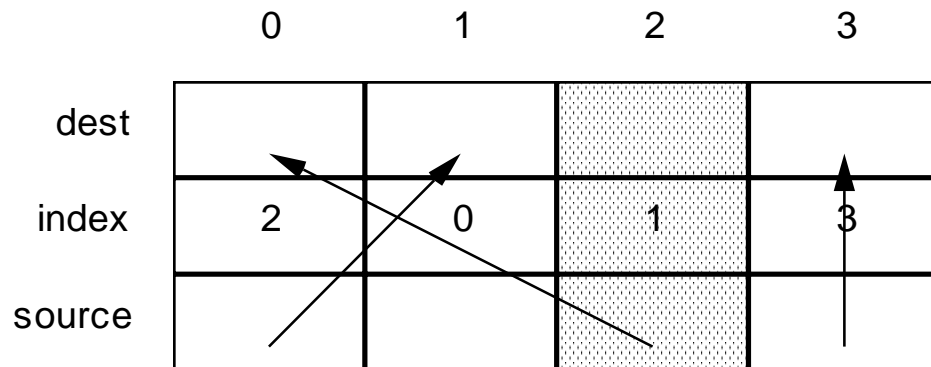
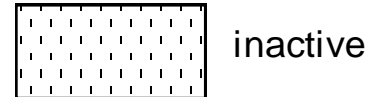
| | 0 | 1 | 2 | 3 |
|--------|---|---|---|---|
| index | 2 | 0 | 1 | 3 |
| dest |  | |  |  |
| source |  |  | |  |

inactivity affects sender

- An active position can send a datum to an inactive position.

Parallel Left-index on RHS with Inactivity

```
where(index != 1)  
  dest = [index]source;
```



inactivity affects receiver

- An active position can get a datum from an inactive position.

Advanced Parallel Left-index Examples

```
shape [6]A, [2][3]B, [4][5]C;
```

```
double a:A, b:B, c:C;
```

```
int:A indexa0, indexa1, idxa0, idxa1;
```

```
int:B indexb;
```

```
int:C indexc;
```

a

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|------|-------|-------|
| 0 | 53.0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 |

b

| | 0 | 1 | 2 |
|---|-----|------|-----|
| 0 | 7.1 | 12.1 | 3.1 |
| 1 | 2.7 | 13.2 | 9.9 |

c

| | 0 | 1 | 2 | 3 | 4 |
|---|------|------|------|------|------|
| 0 | 6.1 | 1.0 | 0.1 | 3.14 | 2.7 |
| 1 | 50.0 | 7.2 | 2.1 | 0.2 | 6.02 |
| 2 | 70.0 | 60.0 | 8.3 | 3.2 | 0.3 |
| 3 | 12.0 | 10.0 | 93.7 | 9.4 | 4.3 |

with(B)

[indexb]a = b;

| | | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| 0 | 5 | 2 | 0 |
| 1 | 4 | 1 | 3 |

a (after)

| | | | | | | |
|---|-----|------|------|-----|-----|-----|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 3.1 | 13.2 | 12.1 | 9.9 | 2.7 | 7.1 |

- This operation is a “send.”

with(A)

a = [indexa0][indexa1]b;

indexa0

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 0 |

indexa1

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 2 | 1 | 1 | 2 | 0 | 0 |

a (after)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|-----|------|------|-----|-----|-----|
| 0 | 3.1 | 13.2 | 12.1 | 9.9 | 2.7 | 7.1 |

- This operation is a “get.”

with(A)
[indexa0][indexa1]b = a;

indexa0

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 0 |

indexa1

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 2 | 1 | 1 | 2 | 0 | 0 |

| | 0 | 1 | 2 |
|---|-------|------|------|
| 0 | 13.17 | 1.21 | 53.0 |
| 1 | 11.13 | 92.1 | 42.0 |

- This operation is a “send.”

with(B)

b = [indexb]a;

| | 0 | 1 | 2 |
|--------|---|---|---|
| indexb | 5 | 2 | 0 |
| | 4 | 1 | 3 |

| | 0 | 1 | 2 |
|-----------|-------|------|------|
| b (after) | 13.17 | 1.21 | 53.0 |
| | 11.13 | 92.1 | 42.0 |

- This operation is a “get.”

with(A)

a = [idxa0][idxa1]c;

idxa0

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 3 | 2 | 3 | 2 | 1 |

idxa1

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 0 | 4 | 0 | 3 |

↑ collisions are OK

a (after)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|-----|------|------|-----|------|-----|
| 0 | 1.0 | 93.7 | 70.0 | 4.3 | 70.0 | 0.2 |

- This operation is a “get.”
- Collisions in a “get” are acceptable — they just cause several elements to “get” values from one source.

with(C)

[indexc]a = c;

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 5 | 0 | 5 | 5 | 5 |
| 1 | 1 | 5 | 2 | 5 | 5 |
| 2 | 5 | 4 | 3 | 5 | 5 |
| 3 | 5 | 5 | 5 | 5 | 5 |

a (after)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|-----|------|-----|-----|------|---|
| 0 | 1.0 | 50.0 | 2.1 | 8.3 | 60.0 | ● |

| | 0 | 1 | 2 | 3 | 4 |
|---|------|------|------|------|------|
| 0 | 6.1 | | 0.1 | 3.14 | 2.7 |
| 1 | | 7.2 | | 0.2 | 6.02 |
| 2 | 70.0 | | | 3.2 | 0.3 |
| 3 | 12.0 | 10.0 | 93.7 | 9.4 | 4.3 |

one of the
other values
in "c"

- This operation is a "send."

- Collisions in a “send” are acceptable — an arbitrarily chosen source will be stored into the destination.

with(A)

[indexa0][indexa1]b = [idxa0][idxa1]c;

could also be expressed as:

with(A) {

temp = [idxa0][idxa1]c;

[indexa0][indexa1]b = *temp*

}

idxa0

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 3 | 2 | 3 | 2 | 1 |

idxa1

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 0 | 4 | 0 | 3 |

↑ ↑ collisions are OK

temp (after "*temp* = [idxa0][idxa1]c")

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|-----|------|------|-----|------|-----|
| 0 | 1.0 | 93.7 | 70.0 | 4.3 | 70.0 | 0.2 |

indexa0

| | | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 |

indexa1

| | | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 2 | 1 | 1 | 2 | 0 | 0 |

b (after “[indexa0][indexa1]b = temp”)

| | | | |
|---|------|------|-----|
| | 0 | 1 | 2 |
| 0 | 0.2 | 70.0 | 1.0 |
| 1 | 70.0 | 93.7 | 4.3 |

- The original statement causes both a “get” and a “send” to occur.

A Detailed Look at Assignment Operators

```
int i, j;  
i = 5;  
j = 7;  
i += j;
```

i

| |
|----|
| 12 |
|----|

- When both the *lhs* and *rhs* are scalar, normal Standard C behavior results.

int:A a0, a1;

a0

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|----|---|---|---|
| 0 | 1 | 5 | 11 | 5 | 4 | 3 |

a1

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 7 | 9 | 2 | 1 | 0 | 6 |

a0 += a1;

a0

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|----|----|---|---|---|
| 0 | 8 | 14 | 13 | 6 | 4 | 9 |

- When both the *lhs* and *rhs* are parallel, element-wise C* behavior results.

```
i = 12;  
i += a1;
```

a1

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 7 | 9 | 2 | 1 | 0 | 6 |

i

| |
|----|
| 37 |
|----|

- When the *lhs* is scalar and the *rhs* is parallel, each active element of the parallel variable is operated — in this case, added — into the scalar (as if in some serial order).

```
i = 5;  
a1 += i;
```

a1 (before)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 7 | 9 | 2 | 1 | 0 | 6 |

a1 (after)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|----|----|---|---|---|----|
| 0 | 12 | 14 | 7 | 6 | 5 | 11 |

- When the *lhs* is parallel and the *rhs* is scalar, usual C* rules apply. That is, the scalar variable is promoted to parallel by replication then element-wise C* behavior results.

with(C)
 [indexc]a += c;

a (before)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|------|-------|-------|
| 0 | 53.0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 |

indexc

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 5 | 0 | 5 | 5 | 5 |
| 1 | 1 | 5 | 2 | 5 | 5 |
| 2 | 5 | 4 | 3 | 5 | 5 |
| 3 | 5 | 5 | 5 | 5 | 5 |

a (after)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|-------|------|------|-------|---|
| 0 | 54.0 | 142.1 | 3.31 | 50.3 | 71.13 | ● |

$$13.17+6.1+0.1+3.14+2.7+7.2+0.2+6.02+70.0+3.2+0.3+12.0+10.0+93.7+9.4+4.3$$

- This operation is a “send with add.”
- Collisions in a “send with combiner” are acceptable — all sources will be combined into the destination by performing the specified operation.

Selecting an Arbitrary Representative

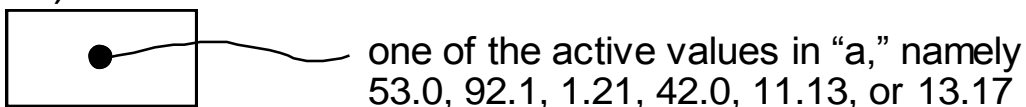
- An arbitrary element of a parallel variable may be selected by casting a parallel variable into a scalar type.

a (before)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|------|-------|-------|
| 0 | 53.0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 |

```
double x;  
with(A)  
  x = (double) a;
```

x (after)



- If no element of "a" is active, the result is undefined.

Explicitly Causing Promotion to Parallel

- A scalar expression may be cast to a parallel type to cause the explicit promotion to parallel.
- The scalar expression is made parallel by replicating its value across all positions.
- Using this paradigm, several programming idioms may be fabricated:
 - “Are any positions active?” maps into
`|= (int:current) 1`
 - “How many positions are active?” maps into
`+= (int:current) 1`

Grid Communication

$dest = [pcoord(0)+k]source;$

$dest = [pcoord(0)-k]source;$

$dest = [.+k]source;$

$dest = [.-k]source;$

$dest = [(pcoord(0)+k) \% \% dimof(s, 0)]source;$

$dest = [(pcoord(0)-k) \% \% dimof(s, 0)]source;$

$dest = [(.+k) \% \% dimof(s, 0)]source;$

$dest = [.-k) \% \% dimof(s, 0)]source;$

Where: *source* is a parallel expression of the current
shape,

k is a scalar integral expression,

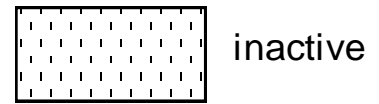
s is the shape of source (or “current”)

Grid Communication (continued)

- $\text{dimof}(s, 0)$ may be substituted by a constant expression with the same value, if this is for a compile-time fully-specified shape.
- Each subscript of a multidimensional parallel variable must be of one of the above forms.
- All subscripts must be within range. Subscripts which would be outside the shape may be disabled through use of the “where” statement.
- The indexed expression may appear on either the *lhs* or the *rhs*.
- Grid communication is faster than General communication
- Grid sends and gets are of equal cost.
- If the offset expressions, k , are not constant expressions, a run-time function is called to perform the appropriate number of Grid operations.
- On the CM-2, both nearest neighbor and power-of-two neighbor communication with both positive and negative offsets are performed to minimize execution time.

Grid Communication on Right-Hand-Side

```
double:A newa;  
with(A) {  
    newa = 0.0;  
    where(pcoord(0) < (dimof(A, 0) - 1))  
        newa = [.+1]a;  
}
```



a

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|------|-------|-------|
| 0 | 53.0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 |

newa (before)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|-----|-----|-----|-----|-----|-----|
| 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

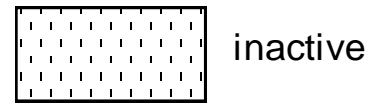
newa (after)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|-------|-------|-----|
| 0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 | 0.0 |

- This operation is a “grid get.”

Grid Communication on Left-Hand-Side

```
double:A newa;  
with(A) {  
    newa = 0.0;  
    where(pcoord(0) > 0)  
        [.-1]newa = a;  
}
```



a

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|------|-------|-------|
| 0 | 53.0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 |

newa (before)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|-----|-----|-----|-----|-----|-----|
| 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

newa (after)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|-------|-------|-----|
| 0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 | 0.0 |

- This operation is a “grid send.”

```
double:A newa;
with(A)
    newa = [(.+1) %% dimof(A, 0)]a;
```

a

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|------|-------|-------|
| 0 | 53.0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 |

newa (after)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|------|------|------|-------|-------|------|
| 0 | 92.1 | 1.21 | 42.0 | 11.13 | 13.17 | 53.0 |

- This operation is a “torus get.”
- Torus communication has roughly the same performance as does usual grid communication. (On the CM-2, for an axis whose dimension is a power-of-two, the performance is identical. Otherwise, some additional cost is required to perform the torus wrapping.)







Parallel Right Indexing

- A parallel right index may be used as a subscript for a parallel array when both the index and the array are of the same shape.
- This enables the programmer to express indirect addressing within a position.
- For example, given the following declaration:

```
int:A i, Array[4];
```

it is permissible to write the expression **Array[i]**. This means that in each position, **i** is used as an index in choosing an element of **Array**. For the following data, the selected elements are shaded:

 selected

| | | | | | | |
|----------|---|---|---|--|---|---|
| Array[0] | |  | | | |  |
| Array[1] | | | | |  | |
| Array[2] |  | | | | | |
| Array[3] | | |  |  | | |
| i | 2 | 0 | 3 | 3 | 1 | 0 |

Built-in Communications Functions

- Grid communication
 - Grid and torus versions
 - Single axis and multiple axes offsets
 - Scan, reduce, copy_reduce, global reduction, spread, copy_spread, enumerate, rank, multispread functions are available
- General communication
 - Manipulation of send addresses
- Reading and writing complete parallel variable to and from the front end

New Boolean Data Type

bool

- A bool is an integral type (i.e., it follows the standard integral promotions specified in Standard C). When used as an rvalue, it is promoted to an int.
- Storing into a bool causes a logical test to occur. That is, if zero is assigned, a zero is stored into the bool; if a non-zero value is assigned, a one is stored into the bool.
- A bool is at least one bit in size.
- In C*, a parallel data type and its corresponding scalar data type need not have identical representations, alignment, or sizes. For example on the CM-2 with a VAX front end, a scalar float is represented in VAX floating-point format, but a parallel float is represented in IEEE floating-point format; with any front end, a scalar bool is stored as a char, but a parallel bool is stored as a bit. On the CM-5, both scalar and parallel bools are stored as chars.
- A related operator, “boolsizeof,” returns the size of its operand in units of bools. For example, in the CM-2

implementation, `boolsizeof(int) == 4` and
`boolsizeof(int:current) == 32`.

Pointers

```
int * p;
```

- “p” is a pointer to an int — just plain old C.

```
double (* p)();
```

- “p” is a pointer to an unprototyped function which returns a double — just plain old C.

```
int (* p)(void);
```

- “p” is a pointer to an prototyped function which takes no arguments and returns an int — Standard C.

```
shape * p;
```

- “p” is a pointer to a shape.

```
int:current * p;
```

- “p” is a pointer to a parallel int of the current shape.

- The address-of operator, unary “&”, applied to a parallel variable returns a pointer to that parallel variable. The address-of operator may be applied to a parallel variable which need not be of the current shape. Furthermore, no shape need be current.
- The dereference operator, unary “*”, applied to a pointer to a parallel variable returns the parallel variable.

```
int: void * p;
```

- The target of a pointer may be declared of “void” shape.
- Pointers to parallel ints of any shape may be assigned to such a pointer.
- A pointer to a parallel variable includes sufficient information to know the actual shape of the parallel variable to which it points.
- The shape may be retrieved by using the “shapeof” intrinsic function.

```
shape [8192]S;
int: S x;
```

```
int: void * p;  
p = &x;  
assert(sizeof(*p) == S);
```

Parallel Structures and Unions

- An entire struct or union may be declared to be parallel.
- Only scalar variables are allowed within a struct or union.
- Shapes are not allowed within structs or unions; however, a pointer to a shape is allowed within a scalar struct or union.
- An array may be included in a parallel struct or union.
- Pointers are not allowed within a parallel struct or union.

```
struct complex {  
    double real;  
    double imaginary;  
};  
struct complex:A cmplxpvar;
```

- **cmplxpvar.real** and **cmplxpvar.imaginary** are parallel doubles of shape **A**.

Functions

```
int:current f(int:current i) {  
    }  
}
```

- Parallel arguments to functions and parallel return values from functions must be of the current shape (the keyword “current” need not be used, but the specified shape does need to be the current shape when the function is called).
- Functions may deal with parallel arguments and parallel return values of any shape by passing and returning their addresses (i.e., use a pointer to a parallel variable).
- As with assignment, a parallel variable passed by-value is only passed in the active positions. To cause all positions of a parallel variable to be accessible from within a function, pass a pointer to the variable (or insure that all positions are active by using “everywhere”).
- Shapes may be passed to and returned from functions.

float:C * f(char:current a, double:void * b, float:T * c);

Overloading Functions

- User functions may be overloaded — that is, several functions may share the same name so long as the functions differ in the type of at least one of their arguments or have a different number of arguments.
- Before overloading is allowed (specifically, before the second declaration of a same-named function), the function must be declared with the **overload** keyword, as follows:

overload function;

- This overload declaration must occur in the same order relative to the function's prototype declarations in all compilation units which declare this function.
- The overloaded declarations might appear as follows:

float function(float x);

double function(double x);

float:current function(float:current x);

double:current function(double:current x);

- Based on the type and number of arguments in a call to an overloaded function, the compiler will select the appropriate function at compile time.

To Prototype or Not To Prototype

- If a function is prototyped or does not return an int, it must be declared before it is called. The form of the declaration must be the same as the definition.

Declaration:

```
int:current incr();
```

Definition:

```
int:current incr(i)
int:current i;
{
    return i+1;
}
```

- The above is a non-prototyped function.
- When an integral type smaller than an int is passed to a non-prototyped function, it is promoted to an int by the caller.
- When a floating-point type smaller than a double is passed to a non-prototyped function, it is promoted to a double by the caller.

- If the function declares a formal parameter which is an integral type smaller than an int, the int passed to the function is demoted to the type of the formal parameter by the function.
- If the function declares a formal parameter which is a floating-point type smaller than a double, the double passed to the function is demoted to the type of the formal parameter by the function.
- These rules are appropriately extended for parallel arguments.
- When these promotions and demotions are needless, they may be omitted by the compiler *if the functions are declared and defined with prototypes*.

Declaration:

```
int:current incr(int:current i);
```

Definition:

```
int:current incr(int:current i) {  
    return i+1;  
}
```

- The above is a prototyped function.

- Promotions and demotions of arguments and return value will occur as if by assignment.

Dynamic Behavior

- Dynamic storage allocation and deallocation for parallel variables is provided via the following functions:

palloc
pfree

- Dynamic allocation and deallocation of shapes is provided via the following intrinsic functions:

allocate_shape
allocate_detailed_shape
deallocate_shape

Example of Dynamic Parallel Variable Allocation

```
shape [16384]S;  
main() {  
    int:S * p0, * p1;  
    p0 = palloc(S, boolesizeof(int:S));  
    p1 = palloc(S, boolesizeof(int:S));  
    f(p0);  
    pfree(p0);  
    pfree(p1);  
}
```

- “palloc” and “pfree” allocate and free storage for parallel variables.
- The storage is heap managed (i.e., it may be allocated and freed in any order).

Example of Dynamic Shape Allocation

```
shape []S;
main() {
    allocate_shape(&S, 1, 4096);
    {
        int:S t0, t1;
        with(S) {
            t0 = 23;
            t1 = 76;
            t0 += t1;
        }
    }
    deallocate_shape(&S);
}
```

- The number of positions in shape S is defined at run time.
- Because shape S was declared to be a rank one shape, `allocate_shape` must maintain that rank.
- If shape S were declared as a fully-unspecified shape, any rank shape could be allocated for it.
- `palloc/pfree` and `allocate_shape/deallocate_shape` may be used together.

- On the CM-2, a function, `allocate_detailed_shape`, allows layout to be specified.

Invoking the Compiler

- The filename extension for C* source files is “.cs”.
- The compiler is named **cs** and is invoked by entering the command string:

`cs filename.cs`

- An executable load module named “a.out” is produced for the compilation.
- Multiple C* source files may be specified on the command line. Each source file is compiled, then the object files are linked into a single executable load module.
- In addition to C* source files, **cs** accepts **.c** source files, **.o** output files, **.obj** JBL files (VAX only), and **.a** library files.
- A number of switches may also be specified on the command line.

Switches In Common with **cc**

- c Compile only.
- D*name*[=*def*] Define a symbol name to the preprocessor.
- g Produce additional symbol table information for debugging; required for C* debugging functions.
- I*dir* Search the specified directory for **#include** files.
- L*dir* Add *dir* to the list of directories in the object library search path.
- llib Link with the specified library.
- o *output* Change the name of the final output file to *output*.
- pg Link with profiling libraries for use with gprof.
- U*name* Undefine the C preprocessor symbol *name*.

Basic Switches

- cm2 Use the CM-2 C* compiler. This is the default unless the environment variable **CS_DEFAULT_MACHINE** is set to **cm5**.
- cm5 Use the CM-5 C* compiler.
- help Give information about **cs** without compiling.
- O Enable additional optimization. A program compiled with this level of optimization is too highly optimized to debug.
- O0 Disable optimization.
- version Print the version number of the compiler prior to compilation. If no source file is specified on the command line, compilation is not attempted.

The CM-2 Compilation Process

- Compilation of C* programs progresses in three stages:
 - A preprocessing phase
 - A C* compilation phase
 - A C compilation phase (including preprocessing and linking)
- The C* compilation phase produces a C/Paris file with the extension **..c**.
- By default, **.cs** files progress through all three stages whereas **.c**, **.o**, and **.a** files bypass the first two stages.

Advanced Switches

- cc *compiler* Use the specified C compiler.
- dryrun Show, but do not execute, the compilation steps.
- force Force **.c** files through the C* compilation phase.
- keep c CM-2 only. Keep the intermediate **..c** file.
- noline CM-2 only. Suppress **#line** directives in the output C file.
- verbose Display informational messages during compilation. (Must use “-v” on the CM-5.)
- warn Suppress warnings from the C* compilation phase.
- Z*comp switch* Pass option *switch* to component *comp*, where *comp* is **cpp** or **cc**. Use this switch to specify options for **cpp** or **cc** that **cs** does not recognize. For example,

```
cs -Zcc -w prog.cs
```

will suppress **cc** warning messages.

Issues

- Object-oriented programming in C
 - C++ as emerging object-oriented C standard
 - C* was designed to allow C++ object-oriented extensions to be layered on top of it
 - Object-oriented C* will not be available in the foreseeable future
- Slicewise code generation from C*
 - There are no plans to build a C* compiler which generates slicewise code for the CM-2
- Complex numbers are not yet available in C* (or in Standard C)
 - Complex numbers may be used through the struct mechanism; however, they may not be directly manipulated in expressions
- On the CM-2, the dimension of each axis of a shape must be a power of two; product of all dimensions must be a multiple of the attached machine size
 - This is a Paris restriction which applies only to the CM-2 version of C*

- Allow the programmer to choose the appropriate language for their task
 - One of C*'s goals is to provide most capabilities other languages provide
- Arbitrary bit length integers
 - C* provides bit-fields to minimize storage requirements for smaller integers
 - C* does not provide arbitrary bit length integers through their manipulation in expressions
- Standardization of C*
 - We support the standardization of data parallel programming languages
 - We are currently involved in working with other computer system vendors to standardize C*