Data-Parallel Programming Language:  
\textit{C*}

Presented by Yeh-Ching Chung

Outline
- Introduction
- A Simple C* Program
- Programming in C*
- C* Communication Functions
- Conclusions
Introduction

- C* is an extension of the C programming language designed to help users program massively parallel distributed-memory computers.
  - A method for describing the size and shape of parallel data and for creating parallel variables.
  - New operators and expressions for parallel data, and new meanings for standard operators that allow them to work with parallel data.
  - Methods for choosing the parallel variables, and the specific data points within parallel variables, upon which C* code is to act.

- New kinds of points that point to parallel data and to shapes.
- Changes to the way functions work so that, for example, a parallel variable can be used an argument.
- Library function that also allow communication among parallel variable.

Thinking Machines Corporation
- May, 1993
A Simple C* Program

```c
#include <stdio.h>
shape [2][32768]ShapeA; /* step 1 */
int SHAPEA p1, p2, p3;
int sum = 0;
main ( ) {
    with (ShapeA) { /* step 2 */
        p1=1; /* step 3 */
        p2=2;
        p3=p1+p2; /* step 4 */
        printf("The sum in one element is %d.\n", [0][1]p3); /* step 5 */
        sum += p3; /* step 6 */
        printf("The sum in all elements is %d.\n", sum);
    }
}
```

Step 1: Declaring Shapes Parallel Variables

- **Shapes**

```
shape [2][32768]ShapeA;
```

![Diagram of ShapeA with positions 0, 1, 2, and 32767]
Step 1: Declaring Shapes Parallel Variables (Cont.)

- Parallel Variables

```java
int:ShapeA p1, p2, p3;
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>32767</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Elements

Step 2: Select a Sheap

- With Statement
  - `current shape`

```java
with (ShapeA)
```

Back
Step 3: Assigning Values to Parallel Variables

\[
P1 = 1; \\
P2 = 2;
\]

Step 4: Performing Computations Using Parallel Variables

\[
P3 = p1 + p2;
\]
Step 5: Choosing an Individual Element of a Parallel Variables

![Image of a grid with elements 0, 1, 2, 32767]

Step 6: Performing a Reduction Assignment of a Parallel Variables

```
sum += p3;
```

![Image of a grid with elements 0, 1, 2, 32767]
Programming in C*

- Using Shapes and Parallel Variables
- Choosing a Shape
- Using C* Operators and Data Types
- Setting the Context
- Pointers
- Functions
- More on Shapes and Parallel Variables
- Communication

What is a Shape?

- A shape is a template for parallel data, a way of logically configuring data. You must define the shape of data before you operate on it.
  - The number of its dimensions: rank, axis
  - The number of positions in each of its dimensions

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
& 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & & & & & & & & \\
1 & & & & & & & & \\
2 & & & & & & & & \\
3 & & & & & & & & \\
\hline
\end{array}
\]

axes \[ \begin{array}{c|c}
0 & 1 \\
\hline
\end{array} \]

rank = 2

position
Declaring a Shape

- `shape [16384]employees;`
  - `shape` is a new keyword that C* adds to standard C.
  - `[16384]` specifies the number of positions in the shape.
  - `employees` is the number of the shape. Shape names follow standard C naming rules.

```
shape employees
```

- Declaring more than one shape
  - `shape [16384]employees, [256][512]images;`

Obtaining Information about a Shape

- C* intrinsic function
  - `positionsof` takes a shape as an argument and returns the total number of positions in the shape.
  - `rankof` takes a shape as an argument and returns the shape’s rank
  - `dimof` takes two arguments: a shape and an axis number. It returns the number of positions along that axis.

  - Example: `shape[16384]employees, [256][512]image;`
    - `rankof(employees)=1, positionsof(employees)=16384`
    - `rankof(image)=2, positionsof(image)=131072`
    - `dimof(image, 0)=256, dimof(image, 1)=512`
What is a Parallel Variable?

- Once a program has declared a shape, it can declare variables of that shape. These variables are called parallel variables.

- Parallel and Scalar Variables
  - Scalar variable
    - a type, along with its modifiers and qualifiers
    - a storage class that defines the manner in which the memory is to be allocated.
  - Parallel variable
    - Assign a constant to all elements of a parallel variable at the same time.
    - Perform an arithmetic operation on all elements of a parallel variable at the same time.
    - Do reduction assignments of data

Declaring a Parallel Variable

- unsigned int employee_id:employees;

- Declaring more than one parallel variable
  - unsigned int employee_id:employees, age:employees;
  - unsigned int employee_id:employees, field1:image;

- A shortcut for declaring more than one parallel variable
  - unsigned int: employees employee_id, age, salary;
  - int:shapeA p1, p2, p3;
Positions and Elements

- **corresponding elements**

```
shape employees

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>16383</th>
</tr>
</thead>
<tbody>
<tr>
<td>employee_id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>age</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>salary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Declaring a Parallel Structure

- **Example:**

```c
shape [16384]employees;
struct date {
  int month, day, year;
};
struct date: employees birthday;
```

- **Declaring more than one parallel structure**
  - struct date:employees birthday, date_of_hire;
  - struct date birthday:employee, date_of_purch: equipment
Declaring a Parallel Array

- Example: `int:employees ratings[3];`

Another syntax:
- `int ratings[3]:employees;`

Initializing Parallel Variables

- The initializer must be a single scalar value.

Examples
- `shape [65536]ShapeA;`
  `int:ShapeA p1=6;`
- `main() {
  int i=12;
  shape [65536]ShapeA;
  int:ShapeA p1=(6+i);
} `
Obtaining Information about Parallel Variables

- C* intrinsic function
  - \texttt{positionsof}, \texttt{rankof}, \texttt{dimof}
  - \texttt{shapeof} takes a parallel variable as an argument and returns the shape of the parallel variable. It is a \textit{shape-valued expression}.
  - Examples:
    \begin{verbatim}
    shape [16384]employees;  unsigned int:employees age;
    shapeof(age) returns the shape employees.
    unsigned int:employees salary => unsigned int:shapeof(age) salary
    struct date: employees birthday => struct date:shapeof(age) birthday
    \end{verbatim}

Choosing a Shape

- The \texttt{with} Statement
  - A program can operate on parallel data from only one shape at a time. (current shape)
  - We designate a shape to be the current shape by using the with statement.
    \begin{verbatim}
    shape [16384]employees;
    unsigned int:employees employee_id, age, salary, new_salary;
    with (employees) {
      unsigned int sample_salary;
      age = 0;
      new_salary = salary * 2;
      sample_salary = [0]salary;
    }
    \end{verbatim}
Choosing a Shape  (Cont.)

```c
shape [13864]employees, [8192]equipment;
unsigned int employee_id: employees, date_of_purchase: equipment;
main ( ) {
    with (employees) {
        date_of_purchase = 0;
        [6] date_of_purchase = 0;
    }
}
```

Using a Shape-Valued Expression

```c
shape [13864]employees;
unsigned int: employees age, salary;
main ( ) {
    with (shapeof(age)) {
        salary = 200;
    }
}
```
Nesting with Statement

Ex1: with (employees) add_salaries( );

Ex2: shape [16384]ShapeA, [32768]ShapeB;
    int:ShapeA p1, p2;  int:ShapeB q1;
    main( ) {
      with (ShapeA) {
        p1 = 6;
        with (ShapeB) q1 = 12;
        p2 = 18;
      }
    }

Nesting with Statement (Cont.)

Ex3: with (ShapeA) {
    loop:
      with (ShapeB) {
        goto loop;
      }
    }

Ex4: goto loop;
    with (ShapeA) {
      loop:
    }

The break, goto, continue, and return statements also reset the current shape when they branch to an outer level.
Initializing a Variable at Block Scope

Example:
shape [16384] ShapeA;
int:ShapeA p1=6;
main( ) {
    with (ShapeA) {
        int:ShapeA p2=p1;
    }
}

Using C* Operators and Data Types

- C* uses all the Standard C operators, plus a few new operators of its own.
  - C* operators
  - C* a new datatype: bool
  - Parallel unions.
Assignment with a Parallel LHS and a Scalar RHS

- \( p_1 = s_1 \)
- \( p_1 + s_1 \)
- \( p_1 == s_1 \)
- \( p_1 << s_1 \)
- \( (p_1 > 2) \&\& (s_1 == 4) \)

Assignment with a Scalar LHS and a parallel RHS

- \( s_1 = p_1 \)

- \( s_1 = [2]p_1 \)
With Two Parallel Operands

- Corresponding element
- Ex1: $p_2=p_1$
  
\[
\begin{array}{cccccc}
& 0 & 1 & 2 & 3 & \cdots & n \\
p_1 & 18 & 47 & 12 & 95 & 64 & \ldots \\
p_2 & 18 & 47 & 12 & 95 & 64 & \ldots \\
\end{array}
\]

- Ex2: $p_1 \times p_2$
- Ex3: $p_1 \geq p_2$
- Ex4: $(p_1>2) || (p_2<4)$

The Conditional Expression

- The ternary conditional expression `?` operates in slightly different ways depending on mix of parallel and scalar variables in the expression.
  - Ex1: $p_1 = (s_1<5) ? p_2:p_3$
  - Ex2: $p_1 = (s_1<5) ? p_2:s_2$
  - Ex3: $p_1 = (p_2<5) ? p_3:p_4$

\[
\begin{array}{cccccc}
& 0 & 1 & 2 & 3 & \cdots & n \\
p_2 & 3 & 4 & 5 & 6 & 7 & \ldots \\
p_3 & 9 & 10 & 11 & 12 & 13 & \ldots \\
p_4 & 21 & 22 & 23 & 24 & 25 & \ldots \\
p_1 & 9 & 10 & 23 & 24 & 25 & \ldots \\
\end{array}
\]
New C* Operators

- The `<? And `? Operators
  - a `< b => (a < b) ? a : b
  - Ex1: s1 `=? s2;
  - Ex2: p1 `<? p2;

- The `%% Operator
  - The operator provides the modulus of his operands.
  - Ex1: (17 `%% 4) == 1
  - Ex2: (17 `%% -4) == -3
  - Ex3: (-17 `%% 4) == 3
  - Ex4: (-17 `%% -4) == -1

Reduction Operators

- C* reduction operators provide a quick way of performing operations on all elements of a parallel variable.
  - Example: s1 += p1;

- Unary Reduction
  - Example: printf("Total is %d\n", +=salary);

- Parallel-to-Parallel Reduction Assignment
  - Example: [0]payroll += salary;
Reduction Operators (Cont.)

- **List of Reduction Operators**
  - += Sum of values of parallel variable elements
  - -= Negative of the sum of values
  - *= Product of values
  - /= Reciprocal of the product of value
  - &= Bitwise AND of values
  - ^= Bitwise XOR of values
  - |= Bitwise OR of values
  - <= Minimum of values
  - >= Maximum of values
- %=, <<=, >>= can't be used as C* reduction operators.

Reduction Operators (Cont.)

- **Minimum and Maximum Reduction Operators**
  - Example 1:
    ```
    printf("The lowest salary is %d\n", <?=salary);
    printf("The highest salary is %d\n", >>=salary);
    ```
  - Example 2:
    ```
    s1 <?= p1;
    s1 = <?=p1;
    ```
The bool Data Type

- The bool is a new unsigned integral data type in C*. The actual size and alignment of a bool are implementation-dependent.
  - bool is used to test conditions.
  - Example 1:
    ```c
    int i=0, j=4;
    printf("%d\n", (bool)i);
    printf("%d\n", (bool)j);
    ```
  - Example 2:
    ```c
    int i, j=1, k=1;
    bool: current b;
    i = j + k;
    b = j + k;
    ```

The bool Data Type (Cont.)

- The boolsizeof Operator
  - To obtain the exact size of a variable or data type in units of bools, use the new C* operator boolsizeof. boolsizeof has the same precedence and associativity as sizeof.
  - With a Parallel Variable or Data Type
    ```c
    boolsizeof(int:ShapeA);
    ```
  - With a Scalar Variable or Data Type
    ```c
    boolsizeof(int);
    ```
Parallel Unions

Union ptype {
   int i;
   float f;
};
union ptype:ShapeA p1;

p1.i = 50;
p1.f = 89.7;

Setting the Context

- What if we want an operation to be performed only on certain elements of a parallel variable?
- To do this, specify which positions are active by using a where statement. Using where to specify active positions it known as setting the context.
The where Statement

A where statement selects a subset of these positions to remain active.

- Example:
  ```c
  with (population) {
    where (weight > 150.0) {
      
    }
  }
  ```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>32767</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>148</td>
<td>109</td>
<td>170</td>
<td>212</td>
<td>116</td>
<td>212</td>
</tr>
</tbody>
</table>

The where Statement (Cont.)

```c
with (population) {
  float avg_weight;
  count = 1;
  where (weight > 150.0)
    avg_height = (+=height / +=count);
  avg_weight = ( +=weight / +=count);
}
```
The else Statement

shape [32768] population;
float: population weight, height;
unsigned int: population count;
float avg_height_heavy, avg_height_light;
main () {
    with (population) {
        count = 1;
        where (weight > 150.0) {
            avg_height_heavy = (+=height / +=count);
        } else {
            avg_height_light = (+=height / +=count);
        }
    }
}

The where Statement and Scalar Code

shape [32768] population;
float: population weight, height;
unsigned int: population count;
main () {
    with (population) {
        count = 1;
        where (weight > 150.0) {
            [0]weight = 225;
            [0]weight = [1]weight;
            [0]count += count;
        }
    }
}
The everywhere Statement

```c
shape [32768] population;
float: population weight, height;
unsigned int: population count, sex;
float avg_male_height, avg_female_height, avg_height;
main ( ) {
    with (population) {
        count = 1;
        where (weight > 150.0) {
            where (sex) avg_male_height = (+=height / +=count);
            else avg_female_height = (+=height / +=count);
            everywhere avg_height = (+=height / +=count);
        }
    }
}
```

Value of Unary Reduction Operators When There are No Active Positions

- `+= 0`
- `- = 0`
- `* = 1`
- `/ = 1`
- `& = ~0`
- `^ = 0`
- `|= 0`
- `<?= maximum value representable`
- `>?= minimum value representable`
Preventing Code from Executing

```c
if (weight > 150.0)
    where (weight > 150.0) {
        float avg_height = 0;
    }

with (population) {
    unsigned int population temporary = 0;
    if (temporary = (++weight > 150.0)) {
        where (temporary) {
            float avg_height = 0;
        }
    }
}
```

Context and the ||, &&, ?: Operators

- p3 = (p1 > 5) && (p2++);

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>32767</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>-2</td>
<td>13</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Context and the ||, &&, ?: Operators

- \[ p3 = (p1 > 5) \| (p2++) ; \]

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>0 1 2 3 4 ... 32767</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>p1</td>
<td>1 2 3 4 5 ... 8</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>p2</td>
<td>1 2 3 4 5 ... 0</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>after</td>
<td>1 1 1 1 1 ... 1</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>p3</td>
<td>2 2 2 2 2 ... 0</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Pointers

- C* has three kinds of pointers
  - the Standard C pointer
  - a scalar pointer to a shape
  - a scalar pointer to a parallel variable
- Scalar-to-Scalar Pointers
  - int *ptr (declares ptr to be a scalar pointer to an int)
  - ptr = &s1 (puts the address of s1 in ptr)
  - s2 = *ptr (puts the value of s1 into s2)
Scalar Pointers to Shapes

- Point to a shape
  - `shape *ptr;` (declares the scalar variable `ptr` to be a pointer to a shape)
  - `ptr = &ShapeA` (makes `ptr` point to `ShapeA`)
  - with `(*ptr)` (makes `ShapeA` the current shape)

Scalar Pointers to Parallel Variables

- Point to a parallel variable
  - `int:ShapeA *ptr` (declares a scalar pointer `ptr` that points to a parallel int of shape `ShapeA`)
  - A pointer to a parallel variable in C* does not store a physical address, but a value that uniquely identifies the entire set of elements of the parallel variable.
  - Ex1: `ptr = &p1;`

```c
&p1
```

```
p1 18 46 55 15 45 65 51...
```
Scalar Pointers to Parallel Variables

- Ex2: (*ptr)++;

\[ \text{ptr} \]

- Ex3: s1 += *ptr;

Alternative Declaration Syntax

\[ \text{int:ShapeA } p1 \Rightarrow \text{int:ShapeA } \ast \text{ptr}; \checkmark \]

\[ \text{int } p1: \text{ShapeA } \Rightarrow \text{int } \ast \text{ptr:ShapeA}; \xmark \]

- In this case, the compiler interprets the shape name as applying to the pointer, and parallel-to-scalar pointers do not exist in the language.
Pointer Arithmetic

Shape[65536]ShapeA;
int*ShapeA A1[40], *ptr1, *ptr2, p2, p3;

ptr1=&A1[7];
ptr2=ptr1 + 2;
printf("%d\n", ptr2 - ptr1);

p2 = *(A1+9);
p3 = A1[9];

ptr1 = &A1[7];
ptr2 = ptr1 + p2;  => *ptr2 = *ptr1 + p2;
p3 = *(ptr1 / p2);  => (*ptr1 / p2);

Parallel Indexes into Parallel Arrays

- Parallel Right Indexing

array A

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

i

A[i]
C* Pointers’ Limitations

shape [8192]S;
int S A[4], i, p1, p2, *ptr;
int s1;

p1 = A[i];
A[i] ++;
p1 = *(A+i);
p1 = *(ptr -i);

s1 = &(A[i] ); /* Can’t take the address */
s1 = (A + i ); /* Creates invalid pointer type */
p1 = ptr + p2; /* Can’t perform an operation without dereferencing */
p1 = *(ptr / i); /* Can only add or subtract */

Functions

- C* add support for parallel variables and shapes to standard C functions
  - C* functions can take parallel variables and shapes as arguments.
  - C* functions can return parallel variables and shapes.
  - C* adds a new keyword current, which you can use to specify that a variable is of the current shape.
  - C* includes a void predeclared shape name so that you can declare an argument to be a pointer to a parallel variable of any shape.
  - C* supports overloading of functions, so that functions operating on scalar and on parallel data can have the same name.
Passing a Parallel Variable as an Argument

```c
void print_num(int:ShapeA x) {
    printf("The sum is \%d\n", +=x);
}
```

If `p1` is a parallel variable of type `int` and shape `ShapeA`

```
print_sum(p1);
```

If the Parallel Variable is not of the Current Shape

```c
void print_num(int:ShapeA *x) {
    with (ShapeA)
    printf("The sum is \%d\n", +=*x);
}
```

If `p1` is a parallel variable of type `int` and shape `ShapeA`

```
print_sum(&p1);
```

Returning a Parallel Variable

```c
float:ShapeA increment(float:ShapeA x) {
    return (x + 1.);
}
```

Assume that `p1` and `p2` are parallel floats of shape `ShapeA`, and `ShapeA` is the current shape.

```
p2 = increment(p1);
```

The header of the function `increment`

```c
float increment(float:ShapeA x):ShapeA
```
In a Nested Context

```c
float ShapeA increment_if_over_5(float:Shape A x, float:Shape A y) {
    where ( y > 5. ) return (x + 1.);
}
with (ShapeA) p3 = increment_if_over_5( p1, p2);
```

<table>
<thead>
<tr>
<th>shape</th>
<th>ShapeA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p1</td>
</tr>
<tr>
<td></td>
<td>p2</td>
</tr>
<tr>
<td></td>
<td>p3</td>
</tr>
</tbody>
</table>

Passing a Shape as an Argument

```c
int number_of_active_positions(shape x) {
    with (x) {
        int:x local = 1;
        return ( += local);
    }
    The shape that you pass need not be the current shape.
}
float raise(shape employees, float:employees salary):employees {
    return (1.1 * salary);
} employees must be the current shape.
You pass more than one shape, and data is passing between the shapes.
```
Returning a Shape

Shape choose_shape(shape ShapeA, shape ShapeB, int n) {
    if (n) return ShapeA;
    else return ShapeB;
}

This function returns ShapeA or ShapeB, depending on the value of n.

with(choose_shape(shape1, shape2, s1))

The current Predeclared Shape Name

int:current variable1;

if employees is the current shape when this statement is executed, variable1 is of shape employees; if image is the current shape, variable1 is of the shape image.

void print_sum(int:current x) {
    printf(“The sum is %d\n”, +=x);
}

The void Predeclared Shape Name

```c
int sum(int,void *x) {
    with (shapeof(*x))
    return (+= *x);
}
```

Using shapeof with the void Shape

```c
int sum_of_two_vars(int,void *x, int:shapeof(*x) *y) {
    with (shapeof(*x))
    return (+= (*x + *y));
}
```

For parameters declared locally within the function, use current:

```c
float average(int,void *x) {
    with(shapeof(*x)) {
        int:current y =1;
        return ( +=y / +=y);
    }
}
```
Overloading Function

- It maybe convenient for you to have more than one version of a function with the same name.

```c
void f(int x);
void f(int x, int y);
void f(int:current x);
```

overload f;

More on Shapes and Parallel Variable

- Partially specifying a shape
- Creating copies of shapes
- Dynamically allocating and deallocating a shape.
- Using the C* library function `palloc` to explicitly allocate storage for parallel variable
- Casting to shape, and casting to or from a parallel data type.
Partially Specifying a Shape

- To declare a shape without fully specifying its rank and dimensions.
  - shape ShapeA;
  - shape [ ]ShapeB;
- \textbf{rankof} intrinsic function
  - shape s, [ ][ ]t, [8092]u;
  \text{rankof}(s) == 0;
  \text{rankof}(t) == 2;
  \text{rankof}(u) == 1;

Partially Specifying a Shape (Cont.)

- Partially Specifying an Array of Shapes
  - shape ShapeC[10];
  - shape [ ][ ]ShapeD[10];
- Arrays and Pointers
  - shape *ptr; ptr= Sarray;
  Sarray[3] == *(ptr +3) == *(Sarray + 3)
- Limitations
  - shape [ ][4]ShapeE; \text{\times}
  - shape [ ]ShapeF; \text{\checkmark}
Creating Copies

- One way to fully specify a shape is by using the assignment operator to copy a fully specified shape to a partially specified one.

  ```
  Shape ShapeA;
  Shape [256][256]ShapeB;
  ShapeA = ShapeB;
  ```

Assigning a Local Shape to a Global Shape

```java
shape ShapeA;
void f(void) {
    shape [1024][512]ShapeB;
    shapeA = ShapeB;
}
main {
    f();
    {
        int:ShapeA p1;
    }
}
```

Don't Assign a Local Shape to a Global Shape
Dynamically Allocating a Shape

- C* intrinsic function - `allocate_shape`
  - `shape []ShapeB;`
    `shapeB = allocate_shape(&ShapeB, 1, 65536);`
  - `allocate_shape(&new_shape, 3, 2, 2, 4096);`
  - `ShapeD[0] = allocate_shape(&ShapeD[0], 2, 4, 16384);`

Deallocation of a Shape

- C* intrinsic function - `deallocate_shape`

```c
#include <stdlib.h>
shape []S;
int positions = 4096;
main( ) {
    while(positions <= 65536) {
        S = allocate_shape(&S, 1, positions);
        int S p1, p2, p3;
        deallocate_shape(&S);
        positions *= 2;
    }
}
```
Dynamically Allocating a Parallel Variable

- C* intrinsic function - `palloc, pfree`

```c
#include <stdlib.h>
shape [16384]ShapeA;
int:ShapeA *ptr;
main( ) {
    ptr= palloc(ShapeA, boolsizeof(int:ShapeA));
    pfree(ptr);
}
```

---

Dynamically Allocating a Parallel Variable (Cont.)

```c
#include <stdio.h>
shape S;
double:S *p;
main( ) {
    S = allocate_shape(&S, 2, 4, 8192);
    p = palloc(S, boolsizeof(double:S));
    /* … */
    pfree(p);
    deallocate_shape(&S);
}
```
Casting with Shape and Parallel Variable

- Use the C* cast operator to cast an expression to a particular shape and type.

- **Scalar-to-Parallel Casts**
  - `s1 += (int:current)1;`

- **Parallel-to-Parallel Casts**
  - Casts to different type
    - `int:ShapeA p1;  sqrt((double:ShapeA)p1);`

---

Casting with Shape and Parallel Variable (Cont.)

- **Parallel-to-Parallel Casts** (Cont.)
  - Casts to a different shape
    - `shape [22256][256]ShapeB, ShapeA;`
    - `main ( ) {`
      - `ShapeA = ShapeB;`
      - `int a:ShapeA, b:ShapeB;`
      - `with(ShapeB) {`
        - `b = a;`
        - `b = (int:ShapeB)a;`
      }
    }`
The physical Shape

- **physical** is a new keyword that C* adds to C. The shape **physical** is always of rank 1, its number of positions is the number of physical processors in which your program is running.
  - (int:physical) p1;

Communication

- Use to perform communication among parallel data.
  - Sending values of parallel variable elements to other element of the same or a different shape.
  - Getting values of parallel variable elements that are of the same or a different shape.
- C* provides two methods of communication
  - General Communication
  - Grid Communication
Using a Parallel Left Index for a Parallel Variable

- A Get Operation
  - dest = [index]source;

<table>
<thead>
<tr>
<th>source</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>dest</td>
<td>10</td>
<td>30</td>
<td>0</td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>

Using a Parallel Left Index for a Parallel Variable (Cont.)

- A Send Operation
  - [index]dest = source;

<table>
<thead>
<tr>
<th>source</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>dest</td>
<td>20</td>
<td>0</td>
<td>40</td>
<td>10</td>
<td>30</td>
</tr>
</tbody>
</table>
When There Are Potential Collisions

For a Get Operation
- dest = [index]source;

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
source | 0 | 10 | 20 | 30 | 40 |
index | 1 | 1 | 1 | 1 | 1 |
dest | 10 | 10 | 10 | 10 | 10 |
```

When There Are Potential Collisions

For a Send Operation
- [index]dest += source;;

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
source | 0 | 10 | 20 | 30 | 40 |
index | 1 | 1 | 1 | 1 | 1 |
dest |   |   |   |   | 100 |
```
When There Are Inactive Position

- For a Get Operation
  - where (source < 30) dest=[index]source;

```
<table>
<thead>
<tr>
<th>source</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>dest</td>
<td>10</td>
<td>30</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

When There Are Inactive Position

- For a Send Operation
  - where (source < 30) [index]dest=source;

```
<table>
<thead>
<tr>
<th>source</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>dest</td>
<td>20</td>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
When There Are Inactive Position

- Send and Get Operation in Function Call
  ```c
  int:current get_op(int:current source, int:current index)
  {  return ([index]source); }
  ```

```
source
0 10 20 30 40
```  
```
index
1 3 0 4 2
```  
```
dest
10 0
```  

Mapping a Parallel Variable to Another Shape

- For a Get Operation
  ```c
  dest = [index]source;
  ```
An Example: Add Diagonals in a Matrix

shape [4][4] ShapeA;
shape [7] ShapeB;
int: ShapeA source, index;
int: ShapeB dest = 0;
main ( ) {
    with(ShapeA)
        {index} dest += source;
    }

An Example: Add Diagonals in a Matrix (Cont.)

With (ShapeA) [index] dest += source;

<table>
<thead>
<tr>
<th>Source</th>
<th>Index</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>3 4 5 6</td>
<td>12 21 27 30</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>2 3 4 5</td>
<td>18 9 3</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>12 13 14 15</td>
<td>0 1 2 3</td>
<td></td>
</tr>
</tbody>
</table>
Using the `pcoord` Function

- Using `pcoord` to create a parallel variable in the current shape; each element in the variable is initialized to its coordinate along the axis you specify as the argument to `pcoord`.

  ```
  p1 = pcoord(0);
  ```

```

0 1 2 3 4 5 32767
```

```

Using the `pcoord` Function (Cont.)

```c
shape [4][4096] ShapeB;
int:ShapeB p2;
main () {
  with (ShapeB) p2 = pcoord(1);
}
```

```

0 1 2 3 4 5 4095
```

```

0 1 2 3 4 5 ...
```

```

0 1 2 3 4 5 ...
```

```

0 1 2 3 4 5 ...
```

```

4095
```

```

4095
```

```

4095
```

```

4095
```
Using the pcoord Function (Cont.)

```c
shape [4][4096]ShapeB;
int:ShapeB p2;
main ( ) {
    with (ShapeB) p2 = pcoord(1);
}
```

An Example

- **Matrix Transposition**

```c
shape [3][3]ShapeA;
int:ShapeA matrix, mew_matrix;
main( ) {
    with (ShapeA) [pcoord(1)][pcoord(0)]new_matrix = matrix;
}
```
The pcoord Function and Grid Communication

- Get Communication
  - dest = [pcoord(0)+1]source;

- Send Communication
  - [pcoord(0)+1]dest = source;

- Multi-dimension Communication
  - dest = [pcoord(0)-2][pcoord(1)+1]source;
  - dest = [ . -2][ . +1]source;

The pcoord Function and Grid Communication (Cont.)

- Grid Communication without Wrapping
  where (pcoord(0) > 1)
  dest = [. -2]source;

  where (pcoord(0) < (dimof(ShapeA, 0) - 2))
  dest = [. +2]source;
The pcoord Function and Grid Communication (Cont.)

- Grid Communication with Wrapping
  
  ```
  dest = [( .+ 2) %% dimof(ShapeA, 0)]source;
  ```

C* Communication Functions

- Communication Functions allow you to
  - Send values of parallel variable elements to other elements of the same shape.
  - Send values of parallel variable elements of one shape to elements of another shape.
  - Perform different kinds of computation on values while sending them to elements of the same or different shape.
  - Send data from parallel variable elements to a scalar variable, and from a scalar variable to a parallel variable element.
  - Send data from a parallel variable to a scalar array, or from an array to a parallel variable.
Two Kinds of Communication

- Grad Communication
  - from_grad, from_grad_dim
  - from_torus, from_torus_dim
  - to_grad, to_grad_dim
  - to_torus, to_torus_dim
  - Pcoord

- General Communication
  - make_send_address
  - send, get
  - read_from_position, read_from_pvar
  - write_to_position, write_to_pvar
  - make_multi_coord

Communication and Computation

- scan
- spread, copy_spread
- multispread, copy_multispread
- enumerate
- rank
- reduce, copy_reduce
- global
Grad Communication

- Aspects of Grad Communication
  - axis, direction, distance, border behavior, behavior of inactive positions

<table>
<thead>
<tr>
<th>Function</th>
<th>Multiple Axes?</th>
<th>Wrapping?</th>
<th>Get or Send?</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_grad</td>
<td>✓</td>
<td>×</td>
<td>Get</td>
</tr>
<tr>
<td>from_grad_dim</td>
<td>×</td>
<td>×</td>
<td>Get</td>
</tr>
<tr>
<td>from_torus</td>
<td>✓</td>
<td>✓</td>
<td>Get</td>
</tr>
<tr>
<td>from_torus_dim</td>
<td>×</td>
<td>✓</td>
<td>Get</td>
</tr>
<tr>
<td>to_grad</td>
<td>✓</td>
<td>×</td>
<td>Send</td>
</tr>
<tr>
<td>to_grad_dim</td>
<td>×</td>
<td>×</td>
<td>Send</td>
</tr>
<tr>
<td>to_torus</td>
<td>✓</td>
<td>✓</td>
<td>Send</td>
</tr>
<tr>
<td>to_torus_dim</td>
<td>×</td>
<td>✓</td>
<td>Send</td>
</tr>
</tbody>
</table>

The from_grad_dim Function

```
type: current from_grad_dim (  
  type: current *sourcep, type: current value, int axis, int distance);  
);  
dest = from_grad_dim(&source, fill, 0, -1);  
```
The from_grad_dim Function (Cont.)

dest = from_grad_dim(&dest, fill, 1, 2);

---

The from_grad Function

type:current from_grad (  
type:current *sourcep, type:current value,  
int distance_along_axis_0,...);  
);  
dest = from_grad_dim(&source, fill, -1, 2);
The to_grad_dim Function

```c
void to_grad_dim (  
    type:current *destp, type:current source, type:current *valuep,  
    int axis, int distance);
);
to_grad_dim(&dest, source, &fill, 0, 1);
```

The to_grad Function

```c
void to_grad (  
    type:current *destp, type:current source, type:current *valuep,  
    int distance_along_axis_0,...);
);
to_grad (&dest, source, fill, 1, -2);
```
The from_torus_dim Function

type:current from_torus_dim(
    type:current *sourcep, int axis, int distance);
);
dest = from_torus_dim(&source, 0, -1);

The from_torus Function

type:current from_torus ( 
    type:current *sourcep, int distance_along_axis_0,...);
);
dest = from_torus_dim(&source, -1, 2);
The to_torus_dim Function

```c
void to_torus_dim(
    type:current *destp, type:current source, int axis, int distance);
);
```
to_torus_dim(&dest, source, 0, -1);

```
source                        dest

20 22 22 23
30 31 32 33
40 41 42 43
```

The to_torus Function

```c
void to_torus (
    type:current *destp, type:current source, int distance_along_axis_0,...);
);
```
to_torus (&dest, source, -1, 2);

```
source                  dest(middle)                      dest

10 11 12 13
20 22 22 23
30 31 32 33
40 41 42 43
22 23 20 22
32 33 30 31
42 43 40 41
12 13 10 11
```

2002/09/13
The scan Function

type:current scan (  
    type:current source, int axis, CMC_combiner_t combiner,  
    CMC_communication_direction_t direction,  
    CMC_segment_mode_t smode, bool:current *sbitp,  
    CMC_scan_inclusion_t inclusion);  
)

running_total = scan(data, 0, CMC_combiner_add, CMC_upward,  
    CMC_none, CMC_no_field, CMC_inclusive);  

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>

data

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>20</td>
<td>25</td>
<td>28</td>
<td>33</td>
<td>42</td>
<td>48</td>
</tr>
</tbody>
</table>

running_total

The reduce Function

void reduce (  
    type:current *destp, type:current source, int axis,  
    CMC_combiner_t combiner,  
    int to_coord);  
)

reduce(&max, data, 0, CMC_combiner_max, 0);  

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>

data

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The copy_reduce Function

```c
void copy_reduce (  
    type:current *destp, type:current source, int axis,  
    int to_coord, int from_coord);
);
    copy_reduce(&copy, data, 0, 3, 0);
```

<table>
<thead>
<tr>
<th>00</th>
<th>10</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>22</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
</tr>
</tbody>
</table>

```
data          copy
```

The spread Function

```c
type:current spread (  
    type:current source, int axis, CMC_combiner_t combiner);
);
    total = spread(data, 1, CMC_combiner_add);
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>06</td>
<td>66</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>86</td>
<td>86</td>
<td>86</td>
<td>86</td>
</tr>
</tbody>
</table>

```
data          total
```
The copy_spread Function

type:current copy_spread (  
  type:current source, int axis, int coordinate);  
);
  copy = spread(data, 1, 1);

```
  data                            copy

  0 1 2 3                        1 1 1 1
  10 11 12 13                   11 11 11 11
  20 21 22 23                   21 21 21 21
```

The enumerate Function

unsigned int:current enumerate (  
  int axis, CMC_communication_direction_t direction,  
  CMC_scan_inclusion_t inclusion, CMC_segment_mode_t smode,  
  bool:current *sbitp);
);
  number = enumerate(0, CMC_upward, CMCC_exclusive, CMC_none,  
  CMC_no_field);

```
  number

  0 1 2 3 4 5 6 7

  0 1 2 3 4 5 6 7
```
The rank Function

```c
unsigned int:current rank (  
type:current source, int axis,  
CMC_communication_direction_t direction,  
CMC_segment_mode_t smode,  
bool:current *sbitp
);

data_rank = rank(data, 0, CMC_upward, CMCC_none, CMC_no_field);
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 5 6 2 0 3 7 4</td>
</tr>
</tbody>
</table>