# Programming with the ANSI Data Parallel C Extensions

James L. Frankel
Frankel and Associates, Inc.
frankel@world.std.com

Philip J. Hatcher
University of New Hampshire
pjh@cs.unh.edu

Supercomputing `95

# Overview

- Introduction (15 minutes)
- Simple Example (10 minutes)
- Shape, Context, and Control Flow (25 minutes)
- Operators and Standard Functions (25 minutes)
- Communication I (15 minutes)
- —BREAK— (30 minutes)
- Communication II (15 minutes)
- Pointers (25 minutes)
- Elemental and Nodal Functions (25 minutes)
- Conclusions (15 minutes)
- Questionnaire (10 minutes)

# Introduction

- Motivation
- ANSI DPCE process/history
- Prior art
- C base — performance transparency
- Data-parallel programming model

# Motivation

- Support for data-parallel applications
- Execution on high-performance hardware
- Integration with ANSI C goals

# ANSI DPCE process/history

- Began in Array Syntax (ASX) subgroup of Numeric C Extensions Group (NCEG)
- NCEG became ANSI C subcommittee (X3J11.1)
- ASX became Data Parallel C Extensions under X3J11.1
- DPCE specification is a chapter in the X3J11.1 technical report
- X3J11.1 technical report was approved by ANSI C in Dec 1994

# ANSI X3J11.1 Technical Report

- Includes numerous chapters:
  - IEEE floating point support
  - complex numbers
  - variable length arrays
  - restricted pointers
  - DPCE
  - and others…
- Best viewed as a pre-standard.
- Provides a specification sufficient for test implementations.
- Goal is to evaluate the extensions for possible inclusion in ANSI C in the future.
- DPCE would be a standard extension, not a mandatory component.

# Prior Art

- DPCE process used the Thinking Machines C* Reference Manual as the “base document.”
- DPCE specification constructed by subtraction and addition to the base document.
- DPCE process drew upon a broad range of experiences with data-parallel extensions to C.
- Incomplete list:
  - MasPar MPL
  - Pomp C — Hyper C
  - SRC dbC
  - TMC “old” C*
  - OSU/UNH Dataparallel C

# Integration with ANSI C

- Performance transparency and efficiency:
  - Performance hot-spots should be obvious in source code.
  - Efficient C operations should remain efficient.
  - All-knowing compilers should not be required.
- Portability and machine independence:
  - Semantics based upon abstract data-parallel model.
  - Based upon implementation experience with model on a wide-variety of architectures.
  - Underlying hardware details are hidden:
    - processor architecture
    - memory architecture
    - number of processors

# Data-Parallel Programming Model

- Data parallelism is parallelism achieved through the simultaneous application of an operation to a data set.
- Simple example: addition of two vectors
- The data-parallel model is single-threaded.
- All processing elements execute synchronously.
- Non-parallel code executes on conventional “front-end” processor.
- Parallel code executes on “back-end” processor array.

# Multi-Threaded (MT) vs Data-Parallel (DP) Models

- MT is most general but many, many applications are DP
- MT requires programmer to deal with difficult timing issues:
  - race conditions
  - nondeterminism
  - deadlock
- Single-threaded execution of DP model is closer to conventional, serial, programming model.

# Other Abstract Features

- Virtual processors
  - programmer chooses number of processors to match algorithmic requirements, not hardware constraints.
- Shared memory
  - Non-uniform access, but grimy details of how to access memory hierarchy should be hidden.

# Simple Example

- A gentle introduction to:
  - shape
  - context
  - parallel variables
  - overloaded operators
  - grid communication
  - reduction operators

# A Simple DPCE Program

- Jacobi iteration:
  - compute the steady-state temperature distribution on an insulated two-dimensional plate, given constant boundary conditions.
- 3 slides:
  - declarations
  - initializations
  - main loop

```
                        /* Jacobi iteration: declarations */
#include <dpce.h>
#define SIZE            128
#define TEMP            50.0
#define EPSILON         0.1
shape [SIZE][SIZE]cell;
unsigned char:cell active;
double:cell change;
double:cell new;
double:cell old;
main () {
        double    maxerr;
        int       cool_cells;
```

```
                                /* Jacobi iteration: initialization */
    where ((!pcoord(cell,0)) || (!pcoord(cell,1)) ||
                (pcoord(cell,1) == (SIZE-1))) {
        active = 0;
        old = new = 0.0;
    }
    else where (pcoord(cell,0) == (SIZE-1)) {
        active = 0;
        old = new = 100.0;
    }
    else {
        active = 1;
        new = 50.0;
    }
```

```
                        /* Jacobi iteration: main loop */
    do {
        where (active) {
            old = new;
            new = ([.-1][.]old + [.+1][.]old +
                    [.][.+1]old + [.][.-1]old) / 4.0;
            change = fabs(old-new);
            maxerr = (>?= change);
        }
    } while (maxerr > EPSILON);
    cool_cells = (+= (new < TEMP));
}
```

# Shape and Context

- Declaring shape variables and creating shape objects
- Parallel types and parallel variables.
- Layout
- Context
- Context manipulation
- Context versus control flow

# Shape

- A *shape* is a descriptor or template for parallel objects and values.
- The shape specifies the logical structure of the parallel data.
- The shape specifies the physical location of the parallel data.
- An operator may not be applied to two parallel operands unless they have equivalent shapes.
- A shape can be thought of as specifying a virtual processor array.
- A shape has these components:
  - rank
  - dimensions
  - layout
  - context

# Shape Variables, Objects and Values

- Shape type can be used to declare variables.
- Shape objects can be created.
- Shape values can be assigned, queried, passed to functions, etc.
- Shape compatibility rules specify compile-time checking of shape assignment and parallel operand shape conformance.

```
shape S;          /* fully unspecified */

shape []S2;       /* partially specified */

shape [10]S3;     /* fully specified */

S = get_shape();

use_shape(S);

S2 = S3;
```

# Parallel Types and Variables

- A parallel type describes a structured collection of objects or values with a particular member type, called the element type.
- The structure of the collection is defined by an associated shape.
- Variables declared with a parallel type are called parallel variables.
- Shape must be “fully assigned” at run time when parallel variable is created.

```
shape [10]S;

int:S parallel_int;

double:S parallel_struct;

shape S2;

/* error because shape S2 not fully assigned */
int:S2 bogus;
```

# Parallel Structures and Unions

- An entire struct or union may be declared to be parallel.
- Only non-parallel 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 non-parallel struct or union.
- An array may be included in 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:S1 f(int:S0 i) {
        }
```

- Parallel arguments to functions and parallel return values from functions are allowed.
- 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:Sa a, double:void b, float:T *c);
```

# Layout

- Layout indicates how shape positions are mapped to nodes of the processing environment.
- Shape definitions can optionally include suggestions from the the programmer for the layout.
- The **block** specifier suggests that data is to be distributed in blocks of the indicated size.

```
/*    distribute blocks of 10 */
shape [100 block (10)]S;

/*    distribute 10x10 blocks */
shape [100 block (10)][100 block (10)]S2;

/*    distribute 100x10 blocks;
      ie first dimension is not distributed */
shape [100 block (100)][100 block (10)]S3;
```

- The **scale** specifier suggests that data be partitioned into maximal blocks with block dimensions proportional to the scale expressions in the corresponding dimensions.

  /* distribute 2N×N blocks */
  shape [100 scale (2)][100 scale (1)]S2;

- Implementations can use different layouts than what is suggested by the programmer.

# Context

- Positions of a shape are either active or inactive.
- All positions are initially active.
- If inactive, the position does not participate in operations performed on operands of the shape.

# Context Manipulation

where (*parallel-condition*)
    *statement*

where (*parallel-condition*)
    *statement*$_1$
else
    *statement*$_2$

- 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 context altered is the context of the shape of the controlling expression.
- 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
- Non-parallel code in *statement* is always executed

# Context Enlargement

everywhere (*shape-expression*)
*statement*

- The “everywhere” statement widens the context of the shape specified by *shape-expression* for the duration of its nested *statement*.
- Dynamically bound to call chain
- Can exit
- Can call functions
- Non-parallel code in *statement* is always executed

# Context versus Control Flow

- Context is not flow of control. Flow of control is affected by conditionals.
- The context only affects operations on operands of its shape!
- Non-parallel code inside a **where** or **everywhere** always executes.
- Use ANSI C to manipulate flow of control.
- On break, continue, goto, and return, context is correctly reestablished.

# Combining Conditionalization and Contextualization

if (|= (*parallel-condition* != 0))
    where (*parallel-condition*)
        *statement*

- The **where** statement will only be executed if at least one element of the *parallel-condition* is true
- Thus, non-parallel code in *statement* is only executed if at least one position of the current shape remains active
- If the *parallel-condition* contains side effects, the following template may be used:

    if (|= ((*parallel-temp* = *parallel-condition*) != 0))
        where (*parallel-temp*)
            *statement*

# Per-Position Iteration

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

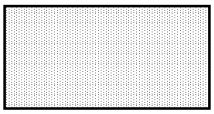
- The *statement* should cause code to be executed which will eventually decrease the positions in which the *parallel-condition* is true
- Thus, the *statement* is repeatedly executed with a gradually diminishing set of active positions
- When no more positions remain active, the **while** loop will terminate
- If the *parallel-condition* 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 | before first iteration |
| prod | 1 | 1 | 1 | 1 | |

| | | | | | |
|---|---|---|---|---|---|
| count | 2 | 0 | 1 | 0 | during first iteration at ♦ |
| prod | 2 | 1 | 2 | 2 | |

| | | | | | |
|---|---|---|---|---|---|
| count | 1 | 0 | 0 | 0 | during second iteration at ♦ |
| prod | 4 | 1 | 4 | 2 | |

| | | | | | |
|---|---|---|---|---|---|
| count | 0 | 0 | 0 | 0 | during third iteration at ♦ |
| prod | 8 | 1 | 4 | 2 | |

| | | | | | |
|---|---|---|---|---|---|
| count | 0 | 0 | 0 | 0 | after while loop completes |
| prod | 8 | 1 | 4 | 2 | |

# Operators and Standard Functions

- New operators
- Overloading and promotion for parallel types
- Functions declared in <dpce.h>

# 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

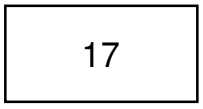
- Assignment version of the >? operator.

%% Real modulus operator

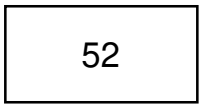
- The % operator is not uniquely defined when either argument is negative.
- The result has the same sign as the denominator.

# Examples of Min and Max Operators

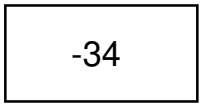
```
int i;
i = 52 <? 17;
```

i 


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

i 


```
i = –34;
i <?= 22;
```

i 

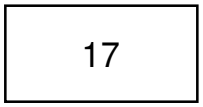

```
i = –34;
i >?= 22;
```

i

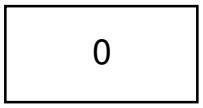
# Examples of Modulus Operator

```
int i;
```

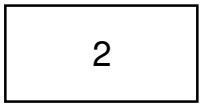
```
i = 17 %% 32;
```

i 

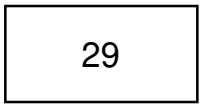

```
i = 32 %% 32;
```

i 


```
i = 34 %% 32;
```

i 


```
i = (–3) %% 32;
```

i 

i = (–3) %% (–5);

i

| -3 |
|---|

i = (–5) %% (–5);

i

| 0 |
|---|

i = (–7) %% (–5);

i

| -2 |
|---|

i = 4 %% (–5);

i

| -1 |
|---|

# Expression Syntax

ANSI C

with

overloaded operators for shapes and parallel variables

para1 = para2 (binary-op) para3;

para1 = (unary-op) para2;

para1 = para2 ? para3 : para4;

# Promotion

- In most cases, when a non-parallel variable is combined with a parallel variable, the non-parallel is promoted to parallel by replication.
- This “broadcast” operation is a form of communication.

# 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 non-parallel, the non-parallel 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 non-parallel overloadings of these operators perform conditionalization.

# Operations Involving Shapes

- Shapes and pointers to shapes may also be operands of ANSI C operators.
- Shapes and shape pointers may be assigned and compared.
- Assignment of a shape copies all the components of a shape object.
- Comparison of shapes causes all the components of the shape objects to be compared (as if they were structs).
- Application of the address-of operator, &, to a shape yields a pointer to a shape. Application of the dereference operator, unary *, to a pointer to a shape yields a shape.
- Pointers to shapes may also be assigned and compared.

# Additional Operator: shapeof

shapeof (*parallel-variable*)

- "shapeof" may be applied to parallel variables.
- It returns a pointer to the shape object associated with its operand.
- "shapeof" may be used anywhere a pointer to a shape may be used.

# Functions Declared in <dpce.h>

- Elemental versions of <stdlib.h> functions:

  abs atoi atol qsort

  calloc malloc realloc

- rankof(*shape*)
  - rankof takes one argument: a shape.
  - It returns the rank (number of dimensions) of its argument.

- dimof(*shape*, *axis*)
  - dimof takes two arguments: 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(*shape*)
  - positionsof takes one argument: a shape.
  - It returns the total number of positions (the product of all dimensions) of its argument.

# Functions Declared in <dpce.h>: Dynamic Behavior

- layoutof(*shape*, *axis*)
  - layoutof takes two arguments: a shape and an axis number.
  - It returns a layout specification of type dpce_layout_t.
- newshape(rank, *dimensions, *specs)
  - newshape takes three arguments: the rank, a pointer to a list of *rank* dimensions of type int, and a pointer to a list of *rank* layout specs of type dpce_layout_t.
  - It returns a new shape with the requested rank, dimensions, and layout. All positions of the new shape will initially be active.
- palloc(*shape, size_in_bytes)
- pfree(*parallel_variable)
  - “palloc” and “pfree” perform dynamic storage allocation and deallocation for parallel variables.

# Identifiers and Types Declared in <dpce.h>: **physical** and layout specification

- shape []physical;
  - **physical** shape has rank one.
  - Dimension of **physical** is equal to the number of nodes in the execution environment.
  - Layout of **physical** is implementation-defined.
- typedef enum { DPCE_BLOCK, DPCE_SCALE } dpce_layout_specs_t;
- typedef struct {
  size_t size;
  dpce_layout_specs_t spec;
  } dpce_layout_t;

# Example of Dynamic Parallel Variable Allocation

```
shape [50000]S;
main() {
        int:S * p0, * p1;
        p0 = palloc(S, sizeof(int:S));
        p1 = palloc(S, sizeof(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() {
        int dim = 4000;
        dpce_layout_t spec = { 0, DPCE_BLOCK }
        S = newshape(1, &dim, &spec);
        {
                int:S t0, t1;
                t0 = 23;
                t1 = 76;
                t0 += t1;
        }
}
```

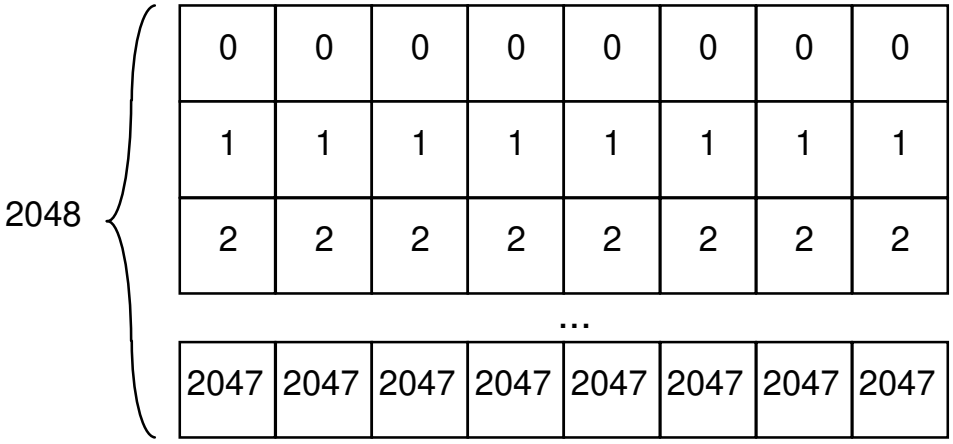
- The number of positions in shape S is defined at run time.
- Because shape S was declared to be a rank one shape, the assignment must maintain that rank.
- {0, DPCE_BLOCK} means that the shape uses the default distribution.
- If shape S were declared as a fully-unspecified shape, any rank shape could be allocated for it.
- palloc/pfree and newshape may be used together.

# Functions Declared in <dpce.h> (cont.)

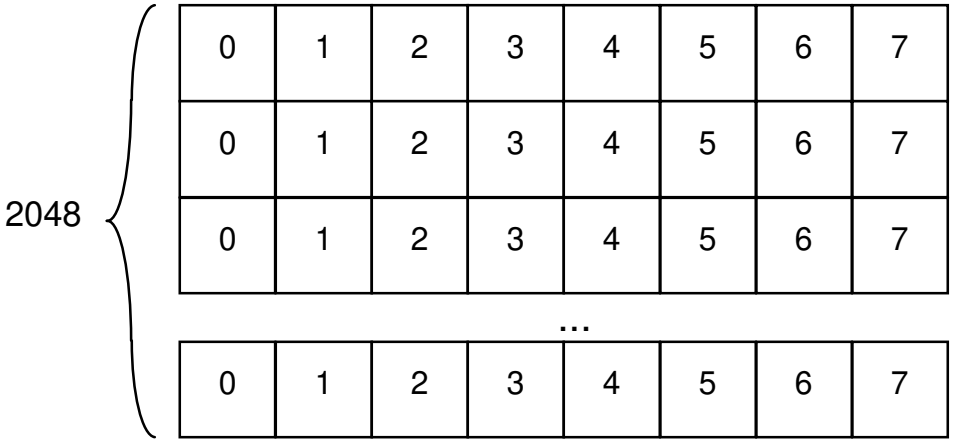
- nodeof(shape)
  - nodeof returns a parallel int of the same shape as the argument.
  - The return value is a node designator of where each position has been mapped.
- nodepositionsof(shape)
  - nodepositionsof returns a parallel int of physical shape.
  - The return value in each position is the number of positions of *shape* that have been mapped to that node.
- psrand(seed)
  - psrand sets the seed for future calls to prand just as does the srand function.
- prand(shape)
  - prand returns a parallel int of the designated *shape* just as does the rand function.

# Functions Declared in <dpce.h>: pcoord

pcoord(Sc, 0) = shape [2048][8]Sc;




pcoord(Sc, 1) = shape [2048][8]Sc;



# Functions Declared in <dpce.h> (cont.)

- Elemental versions of <math.h> functions:

  acos asin atan atan2

  cos sin tan

  cosh sinh tanh

  exp frexp ldexp log log10

  modf pow sqrt

  ceil fabs floor fmod

- Elemental versions of <string.h> functions:

  memcpy memmove

  memcmp memset

# Communication

- Reduction
- Parallel indexing
- Assignment operators
- Grid communication
- Parallel array indexing
- Slicing

# Parallel-to-Non-Parallel Reductions

non-parallel (assignment-op) parallel

| | | |
|---|---|---|
| += | −= | |
| *= | /= | |
| \|= | &= | ^= |
| <?= | >?= | |

# 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.
- Both the parallel binary overloadings and the unary overloadings involve communication.

non-parallel –= parallel;

is defined equivalent to

non-parallel += – += parallel;

or

non-parallel –= += parallel;

# 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 |
| ^= | 0 |
| <?= | largest representable number |
| >?= | smallest representable number |

# Parallel Indexing

[non-parallel]parallel yields a non-parallel

where “parallel” is a parallel variable of rank 1

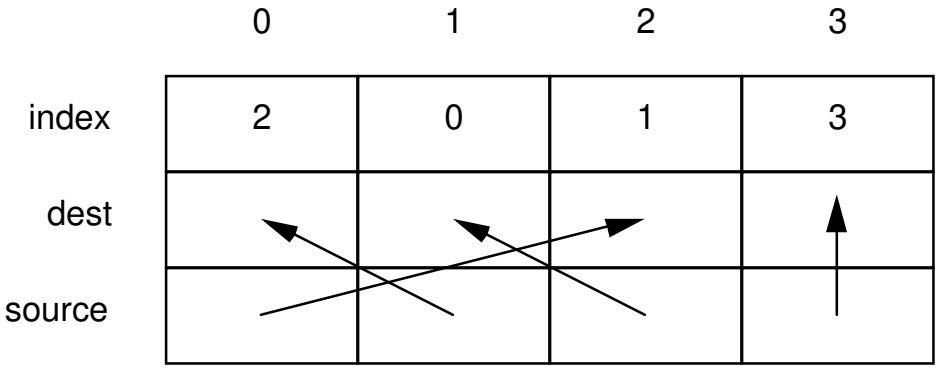
- Moves non-parallel data to or from an element of a parallel variable.
- In general, for a rank *n* parallel variable, all *n* parallel-indices must be non-parallel to achieve this effect.
- May be used in conjunction with array indices if the variable being indexed is an array of parallel variables.

# General Communication

- Parallel 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-indexed parallel variable is the shape of the index.
- If some parallel-indexes are parallel and others are non-parallel, the non-parallel parallel-indexes are promoted to parallel.
- For a rank *n* parallel variable, all *n* parallel-indices must be specified.

# Parallel Index on Left-Hand-Side

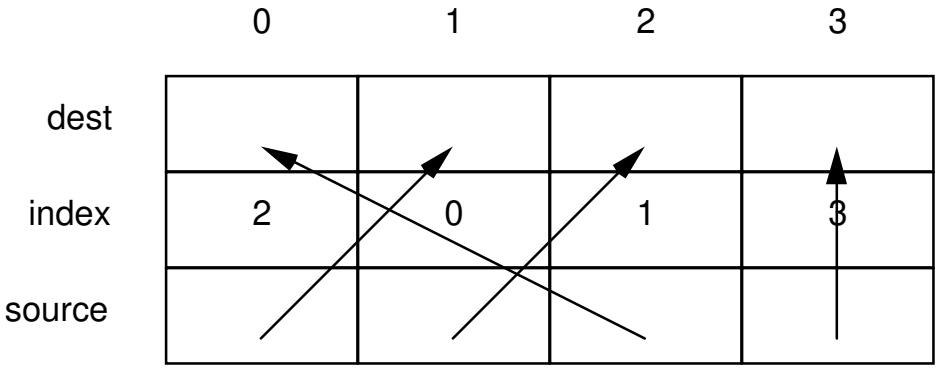
[index]dest = source;




- This operation is a “send.”

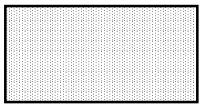
# Parallel Index on Right-Hand-Side

dest = [index]source;



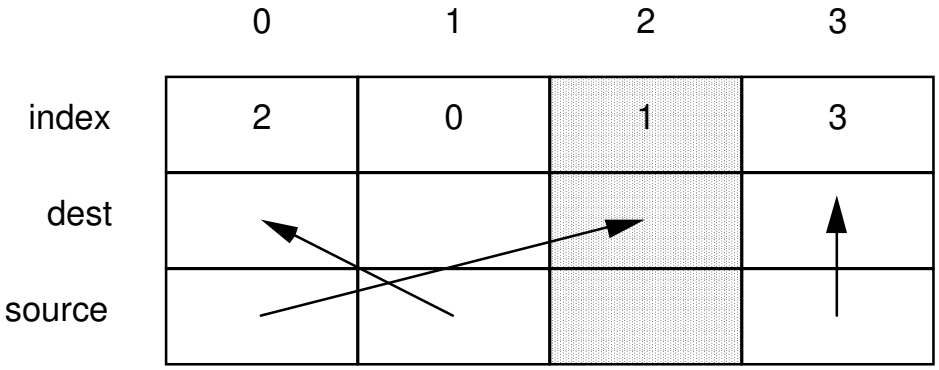

- This operation is a “get.”

# Parallel Index on LHS with Inactivity



inactive

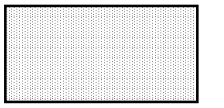
```
where(index != 1)
      [index]dest = source;
```




*inactivity affects sender*

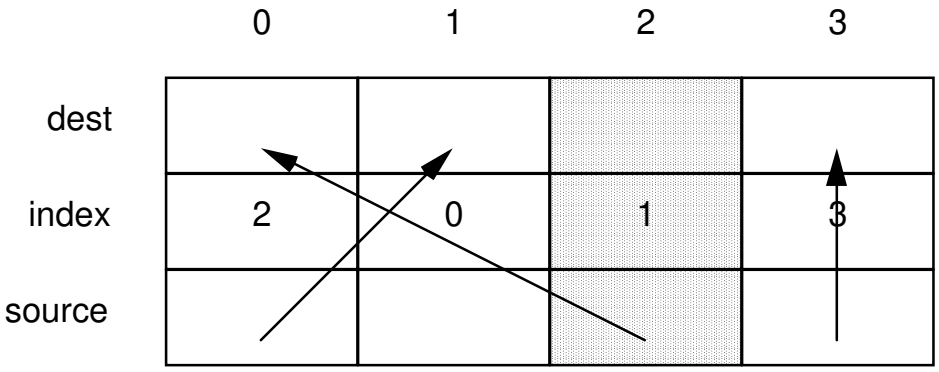
- An active position can send a datum to an inactive position.

# Parallel Index on RHS with Inactivity



inactive

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




*inactivity affects receiver*

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

# Advanced Parallel 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 |

[indexb]a = b;

indexb

| | 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.”

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.”

[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 |

b (after)

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

- This operation is a “send.”

b = [indexb]a;

indexb

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

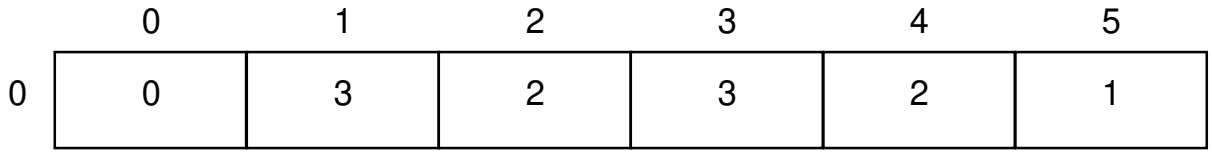
b (after)

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

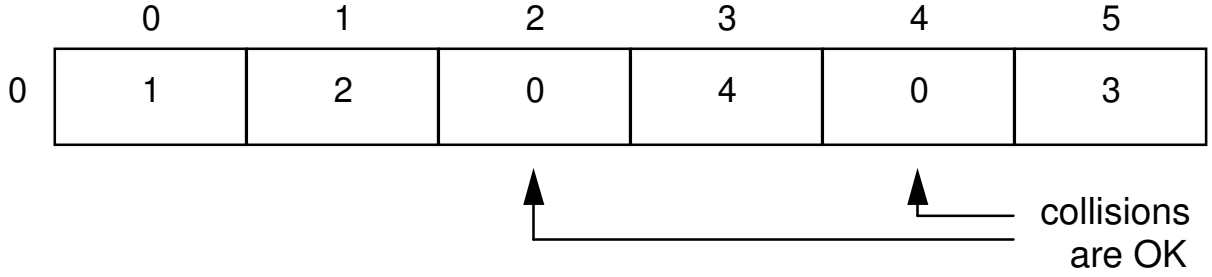
- This operation is a “get.”

a = [idxa0][idxa1]c;

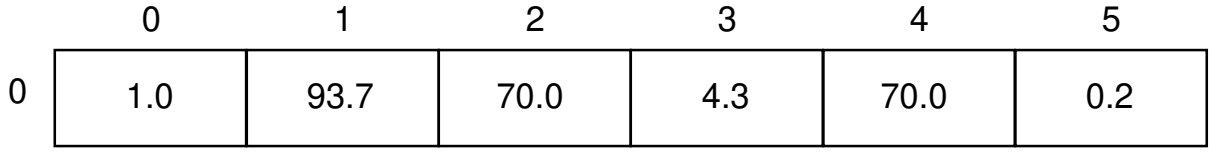
idxa0




idxa1




a (after)




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

[indexc]a = c;

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 | 1.0 | 50.0 | 2.1 | 8.3 | 60.0 | ● |

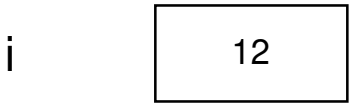
| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 6.1 | | 0.1 | 3.14 | 2.7 |
| 1 | | 7.2 | | 0.2 | 6.02 |
| 2 | 70.0 | | | 3.2 | 0.3 |
| 3 | 12.0 | 10.0 | 93.7 | 9.4 | 4.3 |

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.

# A Detailed Look at Assignment Operators

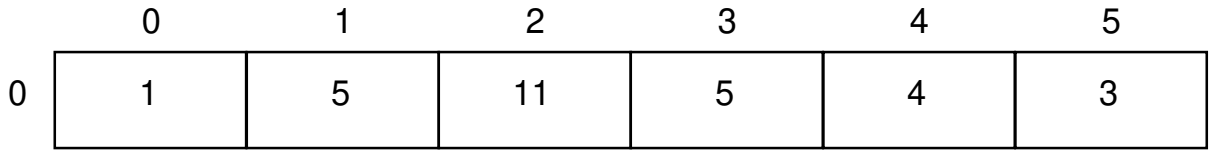
```
int i, j;
i = 5;
j = 7;
i += j;
```




- When both the *lhs* and *rhs* are non-parallel, normal ANSI C behavior results.

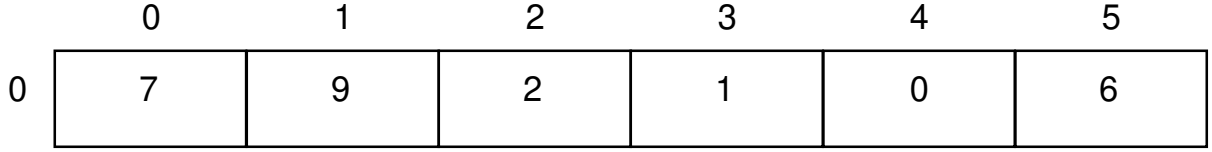
int:A a0, a1;

a0



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

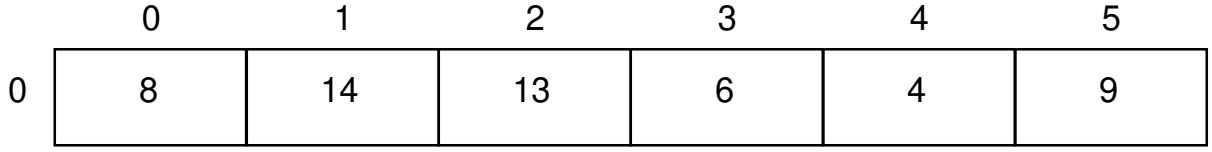
a1



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

a0 += a1;

a0

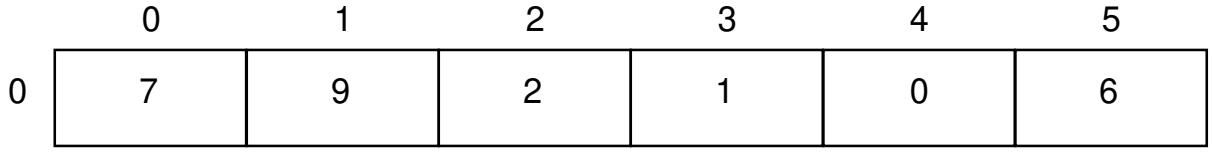


| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 8 | 14 | 13 | 6 | 4 | 9 |

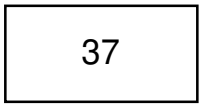
- When both the *lhs* and *rhs* are parallel, element-wise DPCE behavior results.

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

a1




i




- When the *lhs* is non-parallel and the *rhs* is parallel, each active element of the parallel variable is operated — in this case, added — into the non-parallel (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 non-parallel, usual DPCE rules apply. That is, the non-parallel variable is promoted to parallel by replication then element-wise DPCE behavior results.

[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 | |

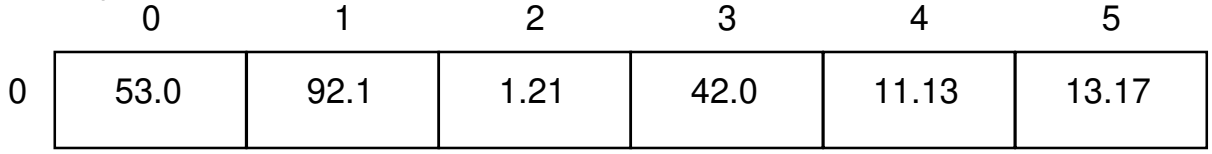
13.17+6.1+0.1+3.14+2.7+7.2+0.2+6.02+
70.0+3.2+0.3+12.0+10.0+93.7+9.4+4.3

- 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

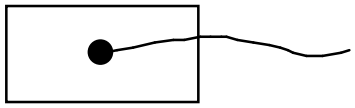
- An arbitrary element of a parallel variable may be selected by casting a parallel variable into a non-parallel type.

a (before)




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

x (after)



one of the active values in “a,” namely 53.0, 92.1, 1.21, 42.0, 11.13, or 13.17

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

# Explicitly Causing Promotion to Parallel

- A non-parallel expression may be cast to a parallel type to cause the explicit promotion to parallel.
- The non-parallel expression is made parallel by replicating its value across all positions.
- Using this paradigm, several programming idioms may be fabricated:
  - “Are any positions of shape S active?” maps into:

    |= (int:S) 1
  - “How many positions of shape S are active?” maps into:

    += (int:S) 1

# Grid Communication

dest = [pcoord(s, 0)+*k*]*source*;
dest = [pcoord(s, 0)–*k*]*source*;

dest = [.+*k*]*source*;
dest = [.–*k*]*source*;

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

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

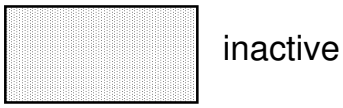
Where: *source* is a parallel expression,
*k* is a non-parallel integral expression,
*s* is the shape of source

# Grid Communication (continued)

- 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*.
- On most platforms Grid communication is faster than General communication

# Grid Communication on Right-Hand-Side

```
double:A newa;
newa = 0.0;
where(pcoord(A, 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 |

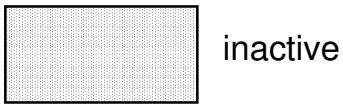
newa (after)

| | 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;
newa = 0.0;
where(pcoord(A, 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;
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 |

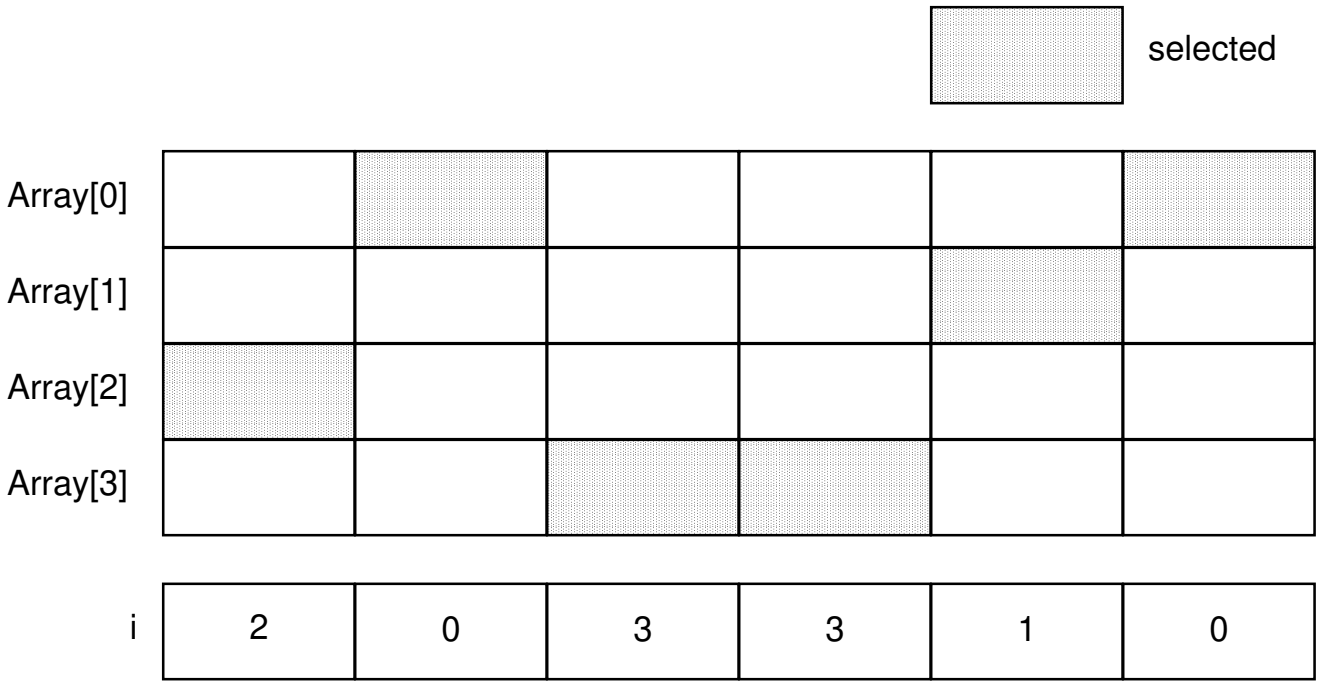
- This operation is a “torus get.”

# Parallel Array Indexing

- A parallel 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

# Slicing

[*first* : *last* : *stride*]*par-expr*

or

[*first* : *last*]*par-expr* — *stride* of 1 is assumed

- May be used on both the lhs and the rhs of assignments.

```
shape [100][100]S;
int:S x;
```

```
[1][0:99:10]x;
```

- This denotes the elements of “x” where the first axis has the sequence {1} and the second axis has the sequence {0, 10, 20, 30, 40, 50, 60, 70, 80, 90}. The resultant shape is “shape [1][10]”.

```
[9:5:-1][3:7:2]x;
```

- This denotes the elements of “x” where the first axis has the sequence {9, 8, 7, 6, 5} and the second axis has the sequence {3, 5, 7}. The resultant shape is “shape [5][3]”.

```
[0:99:2][0:99:2]x = 5;
```

- Odd elements are assigned 5.

```
[0:49][0:49]x = [50:99][50:99]x;
```

- Upper left corner gets lower right corner.

```
[99:0:-1][99:0:-1]x = x;
```

- x is transposed.

# Pointers

- Parallel pointer targets
- Parallel pointer handles

# 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 a prototyped function which takes no arguments and returns an int —ANSI C.

shape * p;

- “p” is a pointer to a shape.

int:S * p;

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

# Pointer Handles and Targets

int * p;

- A pointer itself has type information associated with it.
- The referent of a pointer also has type information associated with it.
- In this case, “p” is the handle and is of non-parallel type.
- The referent of “p” is the target and is of non-parallel int type.
- Because pointers are allowed to point to other pointers, the concepts of handle and target are relative.

int:S ** ptop;

- “ptop” is a non-parallel pointer to a non-parallel pointer to a parallel int of shape S. The type of the target of “ptop” is a non-parallel pointer to a parallel int of shape S

- The address-of operator, unary “&”, applied to a parallel variable returns a non-parallel pointer to that parallel variable.

- The dereference operator, unary “*”, applied to a non-parallel 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 int’s 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” operator. “shapeof” returns a pointer to the shape object.

```
shape [5000]S;
int:S x;
int:void * p;
p = &x;
assert(*shapeof(*p) == S);
```

# Parallel Pointer Handles

- In addition to non-parallel pointer handles, DPCE allows pointer handles to be parallel.

int:S *:S parPtrToParInt;

- The statement above defines a parallel pointer handle “parPtrToParInt” of shape “S” which points to parallel int targets of shape “S”.

double *:S parPtrToNonparDbl;

- This statement defines a parallel pointer handle “parPtrToNonparDbl” of shape “S” which points to non-parallel double targets.

- All declarators which are parallel and are used in the declaration of a pointer handle type must be of compatible shape types. That is, pointer types may not consist of more than one shape.

# Assigning to Parallel Pointers

- Each element of a parallel pointer handle is a pointer.
- Each element may be assigned separately.
- All elements may be assigned using promotion.

```
double *:S parPtrToNonparDbl;
double dbl0, dbl1, dbl2, dbl3;
parPtrToNonparDbl = &dbl0;
[5]parPtrToNonparDbl = &dbl1;
[31]parPtrToNonparDbl = &dbl2;
[17]parPtrToNonparDbl = &dbl3;
```

- Similarly for parallel pointer targets...

```
int:S *:S parPtrToParInt;
int:S parint0, parint1, parint2, parint3;
parPtrToParInt = &parint0;
[2]parPtrToParInt = &parint1;
[3]parPtrToParInt = &parint2;
[40]parPtrToParInt = [42]parPtrToParInt = &parint3;
```

# Dereferencing Parallel Pointers

- Dereferencing a parallel pointer of shape S to a non-parallel type T results in an lvalue of parallel type T and of shape S.
    - The value at each position of the result is the dereference of the pointer in the corresponding position.
- Dereferencing a parallel pointer of shape S to a parallel type T of shape S results in an lvalue of parallel type T and of shape S.
    - The value at each position of the result is the position-oriented dereference of the parallel pointer.
    - That is, each element of the result is formed by following the pointer handle for that element to a parallel target and then selecting only the corresponding element (the element with the same parallel index).
    - This operation *does not* involve any communication!

# Elemental and Nodal Functions

- Elemental functions
- Elemental pointers
- Nodal functions

# Elemental Functions

- Goals:
  - broaden the control flow model
  - facilitate reuse of C code in DPCE applications
- The term elemental is chosen to support two meanings:
  - the function is applied to each *element* of its parallel operands
  - the function implements an *elemental*, logically indivisible, operation
- Elemental functions can be used in both parallel and non-parallel settings.
- One application: many <math.h> functions are declared to be elemental in <dpce.h>.

# Elemental Pointers

- Pointers may be qualified as elemental.
- Elemental pointer qualifier only has meaning when determining the type of the enclosing declaration.
- The type “pointed to” becomes a parallel type if the qualified pointer is parallel.
- See following example…

# Example

- Illustrates building of non-parallel and parallel (per position) linked data structures
- 4 slides
  - declarations
  - elemental function to build data structure
  - elemental function to use data structure
  - main program

```
#include <dpce.h>

/* Linked list node for use either in parallel or
       non-parallel setting */

typedef struct node * elemental LINK;

struct node {
        int count;
        LINK next; /* Pointer to either parallel or
                              non-parallel struct node */
};

shape [100]S;

int n;
int values[10];

int:S many _n;
int:S many_values[10];

LINK head;          /* Non-parallel pointer to non-parallel
                           struct node */

LINK:S many_head; /* Parallel pointer to parallel struct
                           node */
```

```
/* Elemental function to build a linked list:
      n linked nodes are built
      count fields are initialized from values argument */

   struct node *build(int n, int values[]) elemental
   {
    /* Automatic "flow through" for elemental function
       locals: in elemental function everything is parallel
       or everything is nonparallel */

    struct node *last = NULL;
    struct node *new;
    int i;

    for (i = n-1; i >= 0; i--) {
      new = (struct node*) malloc(sizeof(struct node));
      new->count = values[i];
      new->next = last;
      last = new;
    }

    return last;
   }
```

```
/* Elemental function to sum "count" fields of a
   "struct node" list */

    int sum(struct node *list) elemental
    {
      int ret = 0;
      while (list != NULL) {
        ret += list->count;
        list = list->next;
      }
      return ret;
    }
```

```
main()
{
   /* Call to init values, many_values, many_n */

   init_stuff();

   head = build(n,values);              /* nonparallel */

   many_head = build(many_n,many_values); /* parallel */

   printf("sum of head is %d\n", sum(head));

   printf("sum of many_head is %d\n",
      +=(sum(many_head)));

   printf("sum of just [4]many_head is %d\n",
      [4](sum(many_head)));

   /* Or perhaps more efficiently */
   where (pcoord(S,0) == 4) {
      printf("sum of just [4]many_head is %d\n",
         +=(sum(many_head)));
   }

}
```

# Nodal Functions

- Provide an escape hatch to a SPMD programming model.
- Nodal function is invoked once on each node of the execution environment.
- A separate thread is spawned on each node to execute the function body.
- The threads are only required to synchronize upon return.
- Parallel arguments are converted from “global” to “local” view.
- Each node only “sees” the data that is stored on the node.

# Example

- Nodal function to implement Sieve of Eratosthenes
- 3 slides
  - main function (top half)
  - main function (rest)
  - nodal function

```
/* Using a nodal function to implement the
       Sieve of Eratosthenes */

#include <dpce.h>
#define N 4800000
typedef unsigned char BOOL;

shape [N scale (1)] natural;

main () {
/* Each position of 'prime' represents itself and is to be
    evaluated for primality.  The algorithm requires each
    node to have a single contiguous block of numbers. */

  int next_prime = 2;
  BOOL:natural active;
  BOOL:natural prime;

  where (pcoord(natural,0) < 2)
    prime = active = 0;
  else where (pcoord(natural,0) == 2) {
    prime = 1;
    active = 0;
  } else
    prime = active = 1;
```

```
while (|= active)
  where (active) {

/*  Could strike out nonprimes by testing all active
    elements using %, but nodal function can apply
    more efficient method exploiting contiguous
    distribution.  Casts are needed on first two
    parameters to cause pointers to each active
    prime to be generated. */

    StrikeOut(nodepositionsof(natural),
              (BOOL:natural *:natural) &prime,
              (BOOL:natural *:natural) &active,
              pcoord(natural,0),
              next_prime);

    where (prime)
      next_prime = (<?= pcoord(natural,0));

    where (next_prime == pcoord(natural,0))
      active = 0;

  }

  printf("There are %d primes less than %d\n",
    (+= prime), N);
}
```

```
void StrikeOut ( int local_dim;
                 BOOL *local_primes[],
                 BOOL *local_active[],
                 int local_coords[],
                 int current_prime
               ) nodal
{
   int lowbound = local_coords[0];
   int offset = lowbound % current_prime;
   int i;

/* Where is the first number that is a multiple of the
        current prime? */
   if (offset == 0) i = 0;
   else i = current_prime - offset;

/* Stride through to knock out multiples of current prime */
   for ( ; i < local_dim; i += current_prime) {
     *local_primes[i] = 0;
     *local_active[i] = 0;
   }

}
```

# Conclusions

- Differences from C*
- Relationship to HPF
- Relationship to C++
- Where to get more information
- Questions?

# Differences from C*

- no **with** statement and no **current** shape
- no **bool** type
- no function overloading
- less extensive compile-time shape checking
- shape objects have value semantics rather than pointer semantics
- no communication library (cscomm.h)
- added elemental and nodal functions added parallel pointer handles
- added layout suggestions
- added slicing operator

# Relationship to High Performance Fortran

- HPF is also based on the data-parallel model.
- HPF derives data parallelism largely from arrays.
  - Why did DPCE add the shape type?
  - Combination of C arrays and C pointers requires contiguous memory allocation for arrays.
  - Parallel variables may be distributed across processor memories. Distinction of arrays from parallel variables improves the transparency mandated by C culture.
  - In particular, the distinction of array indexing from parallel indexing improves transparency.
- Lack of transparency in HPF programs requires a rich set of tools to help programmers.
- DPCE syntax is not comment syntax!
- Bigger first step to go from serial to parallel program.

# Relationship to C++

- Many parallel C++ approaches exist.
- One camp: hide parallelism inside class libraries.
- Another camp: extend C++ for new parallel features.
- No consensus has formed?
- Efficiency concerns may limit “hidden parallelism” approach.
- C++ is still being standardized.

# Where To Get More Information

- World Wide Web:
  - http://www.cs.unh.edu/pjh/dpce/dpce.html
- Includes pointers to the specification, an informal errata sheet, and related projects.