Data Parallel Programming and the MASPAR

P.Y. Wang

Department of Computer Science 4A5

George Mason University

Fairfax VA 22030-4444 U.S.A.

A Data Parallel Programming Model

- Massive data level parallelism
  - Homogeneous, fine-grain processors
  - Local memory, arithmetic-logic unit
  - Interconnection network for data routing
- **Single** control sequence
  - Instructions are sequentially broadcast from a main control unit to all processors
  - PE's are masked in order for effective computation to the carried out
- Two sets programming instructions
  - Host instruction set (sequential)
  - PE instruction set (data parallel)

Major Features of High-Level Data Parallel Languages

- Data representation issues

  What are the basic language declarations that are used to declare single and parallel data structures?

- Parallel instructions

  How are parallel instructions coded?

- Processor selection

  How are processing elements masked out so that they do not participate in the execution of a parallel instruction?

©2004 P.Y. Wang

- Looping constructs

  What constructs in the languages can be used to repeat sections parallel code?

- Data aggregation and reduction

  What kinds of data aggregation and reduction operations are supported by data parallel languages?

  How are these operations coded?

- Data distribution and broadcasting

  What are the data distribution facilities provided by these languages?

  How are they implemented?

- Data routing

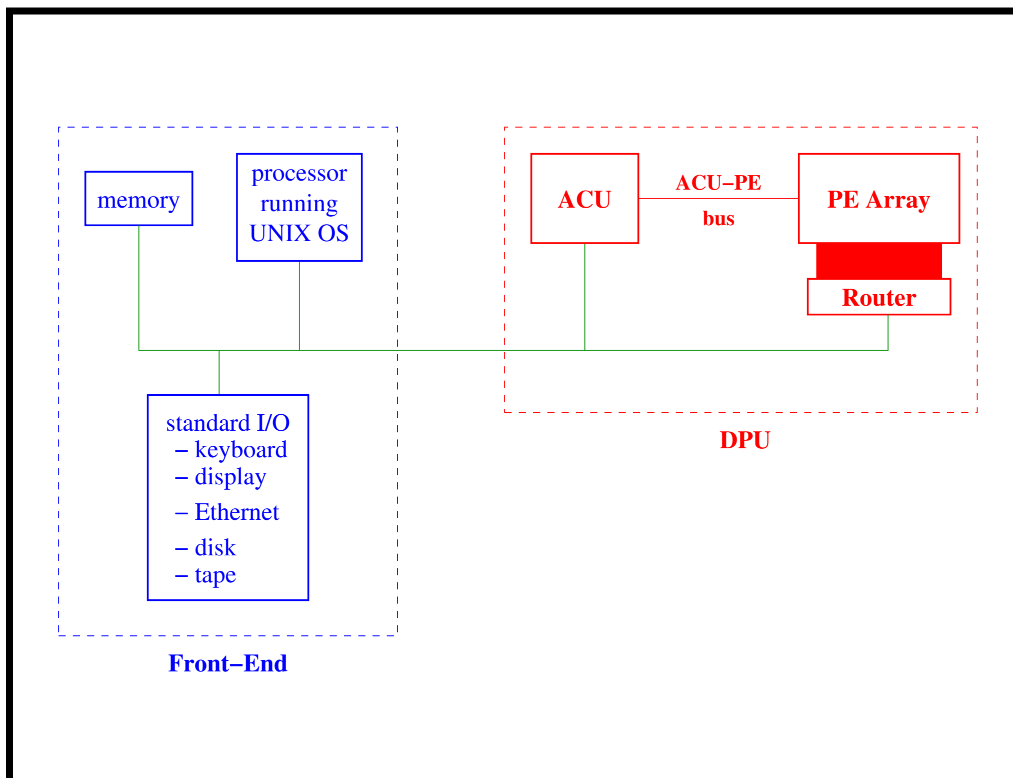
  What are the data routing facilities provided by these languages?

©2004 P.Y. Wang

Maspar MP-2 Architecture Overview

- Consists of a *front-end unit* and a Data Parallel Unit (**DPU**)
- The front-end is a Unix graphics workstation with windowing capability and standard I/O devices
- The DPU consists of an Array Control Unit (**ACU**) and a processor element (**PE**) array

©2004 P.Y. Wang



©2004 P.Y. Wang

- The **ACU** contains a processor with its own registers as well as data and instruction memory
  - Five 32-bit registers for register variables, 128 KB of data memory, 1 MB RAM that expands to 4 GB of virtual instruction memory
  - Controls the **PE array** and performs operations on *singular* data
  - Sends data and instructions to each PE simultaneously

©2004 P.Y. Wang

- The **PE** (processor element) **array**:
  - Each PE has sixteen 32-bit registers available for register variables
  - Each PE has 16 KB of RAM
  - PE's that are enabled (i.e. *active*) will execute the instructions on variables that reside on PE's
  - PE's are arranged in matrices size 1 K, 2 K, 4 K, 8 K or 16 K that are either square or such that the number of columns is twice the number of rows (fixed or toroidal wrap)

©2004 P.Y. Wang

- A PE *cluster* is a non-overlapping square matrix of 16 PE's in the **PE array**
  - A 1 K **PE array** = sixty-four 4×4 PE clusters
  - Clusters are important in global router communications

©2004 P.Y. Wang

- Communication types:
  - Between **ACU** and **PE array** – reflect broadcast and reduction
  - Between PE and PE – occur in the **PE array**
    - ∗ **x-net** communications

      Direct between and PE and any other PE that lies on a a straight line in the N/S/E/W or NE/NW/SE/SW directions
    - ∗ **router** communications

      Arbitrary permutations of data among the PE's

©2004 P.Y. Wang

MPL-Maspar Parallel Application Language

MPL keywords:

- all K & R keywords are supported, including **asm**, but not **entry** and **fortran**
- **all, globalor, plural, proc, router, visible, xnet[p,c]{N,S,E,W,NW,NE,SW,SE}**

plural type:

- Variable being declared is allocated identically to all PE's
- Declaration affects all PE 's, in contrast to **plural** statements and functions

©2004 P.Y. Wang

long long type:

- 64-bit integer data type
- unsigned or signed
- can be **singular** or **plural**

plural types

- can be char, short, int, long long, long, unsigned, float, double, enum, <struct-specifier>, <union-specifier>

©2004 P.Y. Wang

Pointers:

- **Singular** pointer to **singular** data
- **Singular** pointer to **plural** data
- **Plural** pointers to **plural** data

  ⇒ A PE can only point to its own memory

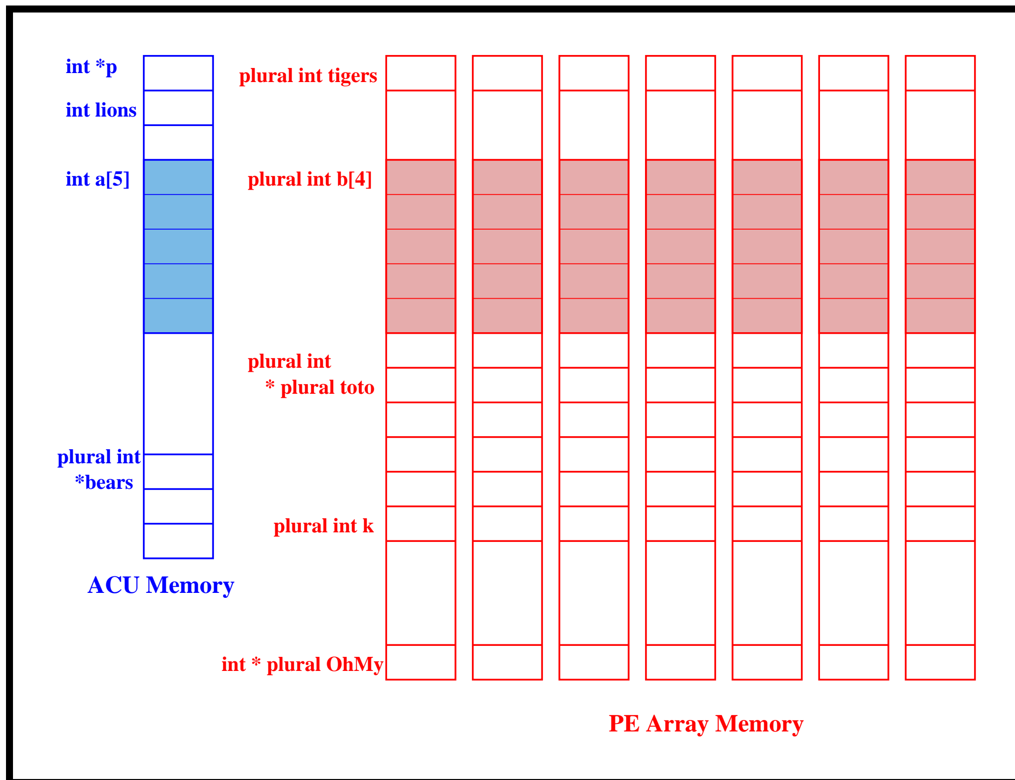
  ⇒ Communications constructs **xnet, router, proc** must be used to access memory of some other PE

  ⇒ The address of a **plural** variable is *singular*

Examples

```
int lions;
plural int tigers;
plural int * bears;
int *plural ohMy;
plural int k;
plural int *plural toto;
```

- lions is a **singular** variable **ACU** memory
- tigers is a **plural** variable in the PE's memory
- bears is a **singular** variable in the **ACU** memory and points to PE memory
- **all** values pointed to by bears have the *same* address



Note:

```
int *plural ohMy;
plural int k;

k = *ohMy;
*ohMy = k;
```

The first statement is well-defined.

The last statement is undefined when the value of ohMy is the same on two or more PE's.

register attribute:

use with the **plural** variable advises the compiler to allocate the variable in a PE register– cannot take the address of such a variable

Plural arrays:

array space is replicated across all PE's (we already saw this)

Structure and union specifiers:

- If a structure variable is declared it as **plural**, the entire structure is allocated on the PE's. Otherwise, the entire structure is allocated on the **ACU**.
- Structure and union members cannot be explicitly declared as **plural**

Example

```
struct gangster {
        int legal_member;
};
struct gangster bonnie;
plural struct gangster clyde;
```

bonnie and bonnie.legal_member are **singular**

clyde and cyde.legal_member are **plural**

struct members may be pointers to singular or plural data:

```
struct {
        int * sopwith;
        plural int * fokker;
} airplane;

plural struct {
        int * redstone;
        plural int * viking;
} rocket;

plural struct farkle {
        struct farkle * frob;
        plural struct farkle * gleep;
} x;
```

x is a **plural** variable with members frob and gleep

frob points to **ACU** memory while gleep points to PE memory

Function definitions:

```
plural int fun(i)
        int i;
{ . . . }
```

This function has a **singular** argument and produces **plural** results.

```
plural float fun(x)
        plural float x;
{ . . . }
```

This function has a **plural** arguments and produces **plural** results.

Front-End ⇔ DPU Data Transfer

The visible keyword

- makes the address of the identifiers associated with it known (visible) to both the front-end and the DPU
- Any identifiers that you pass by reference using

  callRequest(), copyIn(), copyOut(), blockIn(), blockOut()

  must be declared **visible**.
- Example: The front-end calling an MPL subroutine

Front-End	MPL Code
extern mpfun();	visible mpfun()
callRequest(mpfun,nnn,data);	{ . . . }

©2004 P.Y. Wang

Another Example

Front-End	MPL Code
int A[2048];	visible extern int A[2048];
extern int X;	visible plural int X;
blockOut(A,X, . . .);	blockIn(A,&X, . . .);
blockIn(X,A, . . .);	blockOut(&X,A, . . .);

©2004 P.Y. Wang

Writing Expressions

- The arithmetic operators of K & R C have been extended to operate on **plural** operands
- It looks more or less like what you would expect

```
int s1, s2;
plural int p1;
p1 = s1 + s2;
p1 = p1 + s1;
p1 = p1 + (plural int) s1;
```

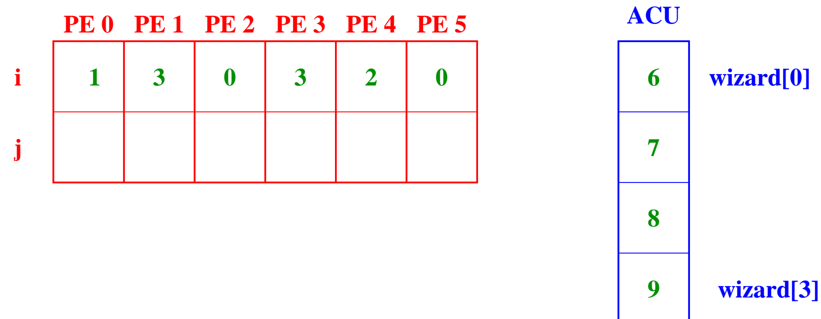
- The operation

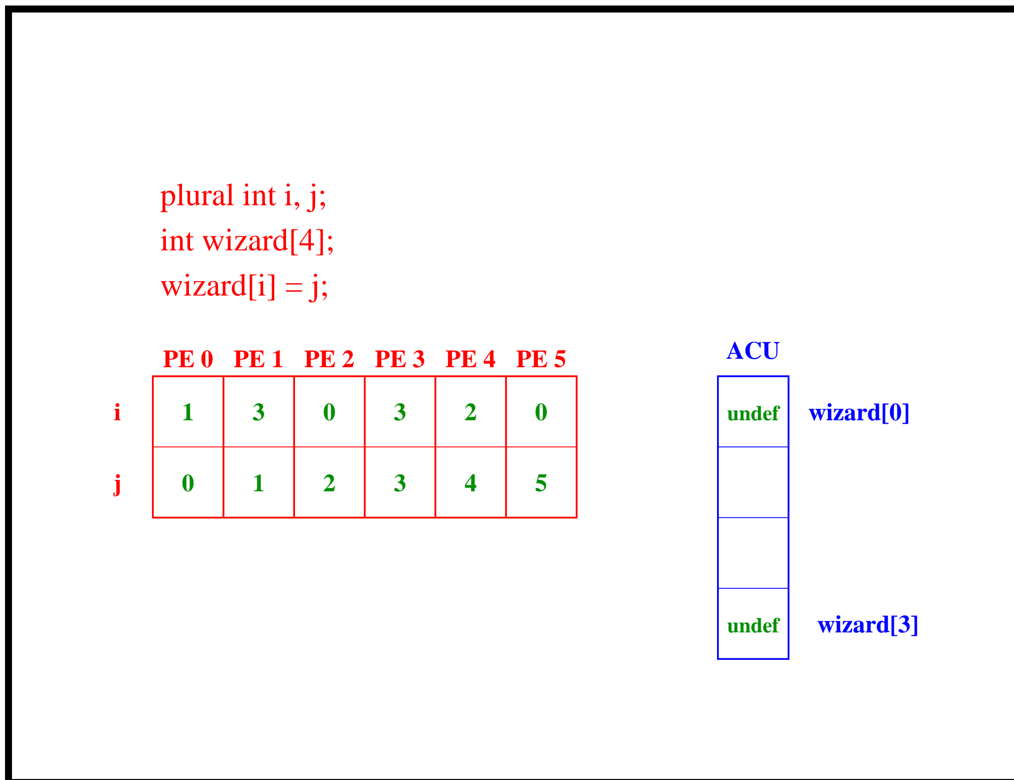
  singular object ← **plural** value

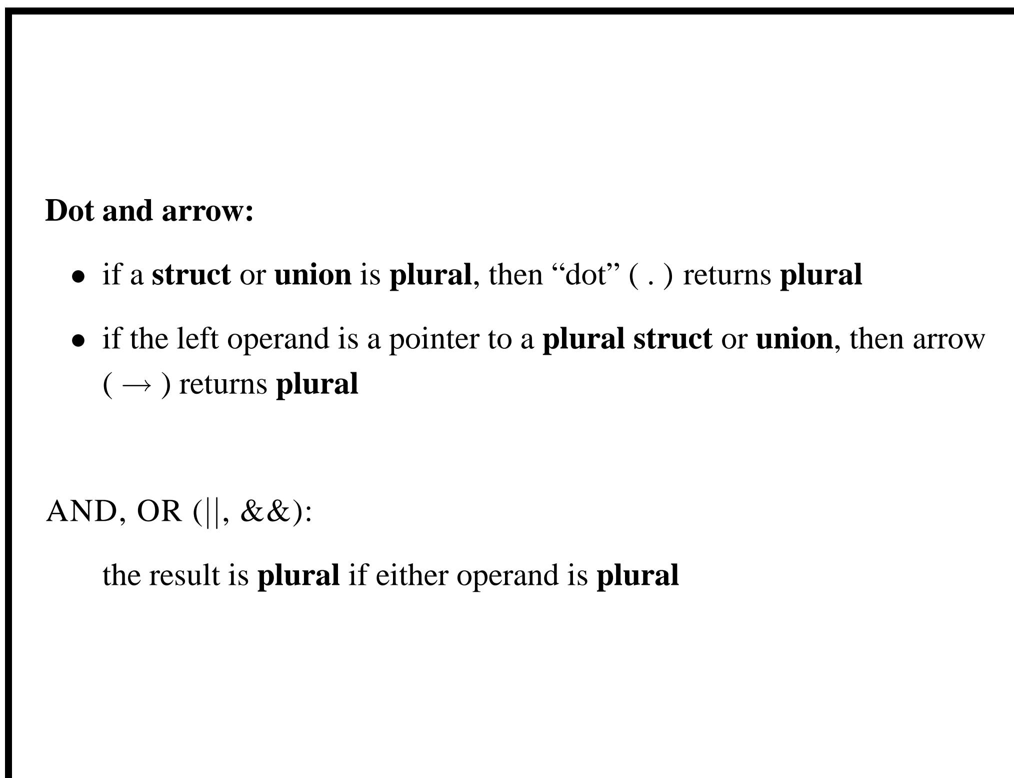
  is *illegal*.

Array subscripting

```
plural int i, j;
int wizard[4];
j = wizard[i];
```




plural int i, j;
int wizard[4];
wizard[i] = j;
PE 0 PE 1 PE 2 PE 3 PE 4 PE 5
i 1 3 0 3 2 0
j 0 1 2 3 4 5
ACU
undef wizard[0]
undef wizard[3]


Dot and arrow:
• if a struct or union is plural, then "dot" (.) returns plural
• if the left operand is a pointer to a plural struct or union, then arrow (→) returns plural
AND, OR (||, &&):
the result is plural if either operand is plural

Conditional expressions

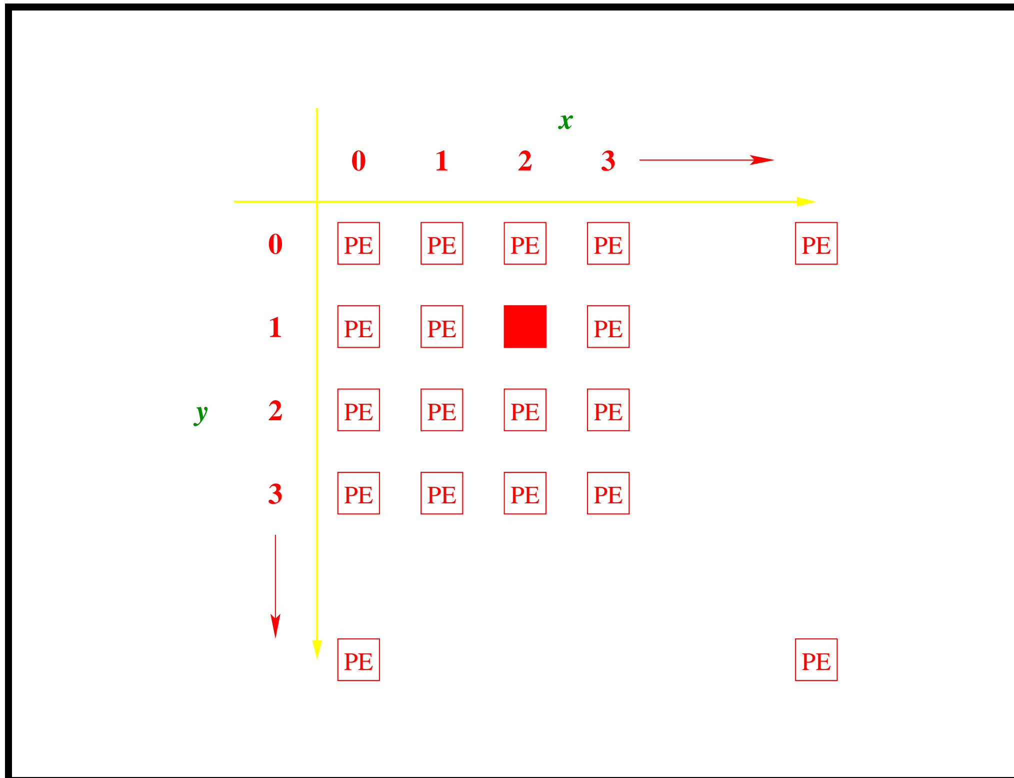
if any of the operands are **plural**, then the PE's for which the condition is true define the active set

globalor

performs a bitwise OR operation on all values of a **plural** expression in the active set and returns a **singular** result

Processing Element Addressing

- MPL Predefined variables are used to give the size and shape of the **PE array**.
- These permit each PE to know its relative position (self-address).



- Size (**singular**):
  - nproc = number of PE's
  - nxproc = number of PE's in the x-direction
  - nyproc = number of PE's in the y-direction
- Representation:
  - lnproc = number of bits needed to represent nproc
  - lnxproc = number of bits needed to represent nxproc
  - lnyproc = number of bits needed to represent nyproc
- Self-addresses (plural):
  - iproc = one-dimensional self-address: 0 . . . nproc-1 (row-wise)
  - ixproc = x self-address: 0 . . . nxproc-1
  - iyproc = y self-address: 0 . . . nyproc-1

Self-address Example

```
if ( ixproc = = 2)
        { /* active set */
        . . .
        }
```


Processing Element (PE) Communications

- **router** uses linear view of the **PE array**
- **proc:** to access the value on a *single* PE

```
int x;
plural int pixel;
x = proc[4].pixel;
x = proc[0][63].pixel;
```

- **xnet:** PE to PE communications – same direction and distance with toroidal wrap

XNET Example

```
plural int i, j;
. . .
i = xnetN[1].j;
i = xnetN[1].j + xnetS[2].k;
i = *xnetN[1].p;
xnetN[1].j = i;
xnetN[1].j = xnetSW[1].i;
```

Remember:

xnet on

left_hand_side ⇔ "send"

right_hand_side ⇔ "fetch"

XNETP

- **xnetp** like **xnet** except it is faster and pipelines its values through intermediate PE's
- Any PE along the pipeline that is active causes the pipelining to stop
- *Example:*

```
external plural int iyproc;
plural int coeff;
if (iyproc == 5) {
        coeff = coeff + xnetpS[8].coeff;
}
```

$\Rightarrow$ row 13 is added to row 5

XNETC

- **xnetc** like **xnetp** but it leaves copies in intermediate PE's along the way
- *Example:*

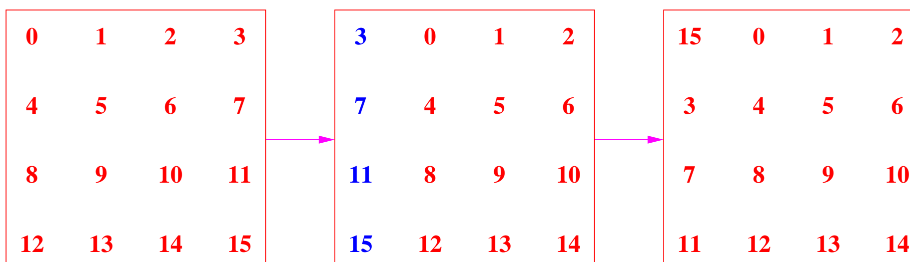
```
plural float coeff;
if (ixproc == 6) {
        xnetcE[3].coeff = coeff;
}
```

⇒ column 6 is copied into columns 7, 8, and 9

©2004 P.Y. Wang

XNET for Non-regular Topologies

```
plural int linear_shift (x)
plural int x;
{
  x = iproc;
  xnetE[1].x = x;
  if (ixproc == 0)
      xnetS[1].x. = x;
  return x;
}
```



©2004 P.Y. Wang

ROUTER Communication Statement

- Each active PE will participate in general communication
- If **router** construct is on the left_hand_side of an assignment, then you are doing a *send* from the active set to the connected set
- If **router** construct is on the right_hand_side of an assignment, then you are doing a *fetch* from the connected set to the active set

©2004 P.Y. Wang

ROUTER Examples

plural int i, j, k;

iproc	**0**	**1**	**2**	**3**	**...**
active	1	0	0	1	
i	**2**	**1**	**3**	**0**	
j	**17**	**23**	**31**	**41**	
k	**19**	**29**	**37**	**43**	

router[i].j = k;

j	**43**	**23**	___	**41**

k = router[i].j;

k	**31**	**29**	**37**

©2004 P.Y. Wang GEORGE MASON UNIVERSITY

MPL Special Parallel Primitives

- Purpose: to compare/order PE instances of a **plural** variable
- **(plural) i = enumerate ();**
  - depends on the active set unlike **iproc, ixproc, iyproc** which are static
- **i = rankN (plural_any_type);**
  - integer order of argument on each PE returned into **i** of that PE
  - **N** denotes data type (8, 8u, 16, 16u, 32, 32u, 64, 64u, f, d)
- **psortN(plural_any_type);**
  - arranges the plural argument, putting the smallest value on PE with lowest self-address
  - after doing **psort(x)**, a **rank(x)** returns the list of integers matching **enumerate**

A Parallel Primitives Example

PE	**0**	**1**	**2**	**3**	**4**	**5**
active	1	0	0	1	0	1

i = enumerate();

i	0			___		___

y = rank16(x);

x	**4**	**5**	**8**	**6**	**1**	**3**
y	___	___	___	___	**0**	**1**

MPL Reduction Functions

- Definition: *reduction* is the **singular** result of performing an operation cumulatively on all elements of an array
- MASPAR implementation details
  1. size of array is determined by the active set (traversed linearly)
  2. result is initialized to the operation's "identity element" (but always returns 0 if there are no active PE's)

©2004 P.Y. Wang

- Conventions for use:

```
#include < reduce.h >            /* needed for type defs */
plural double reduceMuld();      /* defined in reduce.h */
double y;
plural double x;
. . .
y = reduceMuld(x);
```

- Suffixes are 8, 8u, 16, 16u, 32, 32u, 64, 64u, f, d
- Reduction functions are **Mul**, **Add**, **And**, **Or**, **Max**, **Min**

©2004 P.Y. Wang

MPL SENDWITH Functions

- Motivation: with **router**, several PE's can send to the same destination
  - When multiple PE's send data to the **ACU**, we can "reduce" them to a single value (using globalOR, reduceADD, etc).
  - If a PE receives from many senders, the type of reduction should be specified by the programmer
  - **sendwithAdd8** is a hybridization of **reduceAdd8** and the **router** construct
  - Each receiving PE adds together all the data elements it gets during the **sendwithAdd8** procedure

©2004 P.Y. Wang

- Note: if a PE is in the active set but receives no messages, the value returned on that PE is 0, regardless of the type of operation specified as part of the "with" in **sendwith...**
- Same reductions of **Mul**, **Add**, **And**, **Or**, **Max**, **Min**
- Same suffixes of 8, 8u, 16, 16u, 32, 32u, 64, 64u, f, d

©2004 P.Y. Wang

Examples

- Sum from all active PE's into **ACU** receiver:

  sr = reduceAdd__(pv);

  Here pv is **plural** and sr is **singular**.

- Sum from all active PE's to the receivers who are active PE's matching some sender's destination:

  pr = sendwithAdd__(pv,dest);

  Here pv, dest, result are **plural**.

©2004 P.Y. Wang

MPL scan Functions

- **scan** performs an operation cumulatively on a list of PE's
- **segmented scan** performs an operation cumulatively on all PE's within a *segment*
  - Within each segment, it is similar to reduction
  - But unlike **reduceAddd**, for example, **scanAddd** returns a **plural** *double* result and intermediate sums are saved on intermediate PE's
- Currently supported functions are
  - **scanMul**, **scanAdd**, **scanAnd**, **scanOr**, **scanMax**, **scanMin**
  - suffixes are 8, 8u, 16, 16u, 32, 32u, 64, 64u, f, d

©2004 P.Y. Wang

- A *segment* is a series of PE's with a **plural char** variable containing 0, 0, 0, . . ., 0, 1 respectively
  - Cumulative result is returned in each PE (by ascending **iproc**)
  - Segment is confined to PE's in the active set
  - For an unsegmented scan across all active PE's, set **all** seg = 0;
  - Segment wraps from highest active PE to lowest active PE if (seg==1) on at least one active PE and (seg==0) on highest PE

©2004 P.Y. Wang

Two SCAN Examples

PE	0	1	2	3	4	5
x	3	4	8	1	2	6
seg	0	0	0	0	0	1

i = scanAdd16(x,seg);

i	3	7	15	16	___	___

- -

PE	0	1	2	3	4	5
x	3	4	8	1	2	6
seg	0	0	1	0	0	1

i = scanAdd16(x,seg);

i	3	7	15	1	3	

©2004 P.Y. Wang

MPL Statements (Summary)

- K & R used for **singular** statements/expressions
- The active set is the set of PE's enabled at the time
- **if**, **switch**, **while**, **do**, and **for** become **plural** when the controlling expressions is **plural**
- **if** statement

  if (plural_expression) statement

  if (plural_expression) statement1 else statement2

  ⇒ Both blocks may be executed by a different set of PE's

- **while** statement
  - the body is executed by the active PE's
  - at each iteration, the active set is either the same as the previous active set or smaller than the previous active set
  - can contain **return()**, **break**, and **continue** statements, but not **goto()**
- **do** statement: similar to **while**
- **for** statement: it's **plural** if expr2 is **plural**

  for (expr1; expr2; expr3) statement

NOTE: **while**, **do** and **for** restore to the original "active set" before execution, when they are finished

MasPar MPL Sample Program

- Find the average of all 8 neighbors using
  - Toroidal connections
  - Non-torioidal connections

The main.m Code

```
/* main program for the average example */

#include <stdio.h>
#include <stdlib.h>

extern plural int avg( plural int src );
extern plural int nowrapAvg( plural int src );

main ()
{

  plural int i;
  plural int avg_value;
  plural int nowrapAvg_value;
```

```
  i = 10*iproc;   /* Give i some data on each PE*/

  avg_value = avg( i );

  if( iproc == 0 )
    p_printf( " The average on processor 0",
              " with toroidal wrap : %d\n",
        avg_value );

  nowrapAvg_value = nowrapAvg( i );

  if( iproc == 0 )
    p_printf( " The average on processor 0",
              " with no toroidal wrap: %d\n",
        nowrapAvg_value );
  exit(0);
}
```


The average.m Code

```
#include <mpl.h>
#define nowrapXN(d,val) ((iyproc < d) ? 0 : xnetN[d].val)
#define nowrapXS(d,val) ((iyproc >= nyproc-d) ? 0 : xnetS[d].val)
#define nowrapXW(d,val) ((ixproc < d) ? 0 : xnetW[d].val)
#define nowrapXE(d,val) ((ixproc >= nxproc-d) ? 0 : xnetE[d].val)

/* nowrapAvg() averages each pixel with the values of it's 8
   neighbors but doesn't wrap around the PE Array.
   It turns out that only 4 Xnet's are necessary since the first
   line accumulates the NW and NE values into the N neighbor and
   the SW and SE values into the S neighbor.
*/

plural int
nowrapAvg( plural int src )
{
    src += nowrapXW(1,src) + nowrapXE(1,src);
    src += nowrapXN(1,src) + nowrapXS(1,src);
    return(src/9);
}
```

```
/* avg() averages each pixel with the values of it's 8 neighbors.
   It turns out that only 4 Xnet's are necessarY since the first line
   accumulates the NW and NE values into the N neighbor and the
   SW and SE values into the S neighbor.
*/

plural int
avg( plural int src )
{
    src += xnetW[1].src + xnetE[1].src;
    src += xnetN[1].src + xnetS[1].src;

    return(src/9);
}
```

©2004 P.Y. Wang