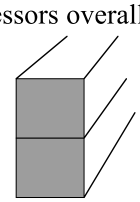
The MultiC Language

- MultiC is primary language on the WaveTracer and the Zephyr SIMD computers.
- The Zephyr is a second generation WaveTracer, but was never commercially available.
- Both MultiC and a parallel language designed for the MasPar are similar to an earlier parallel language called C*.
  - C* was designed by Guy Steele for the Connection Machine.
  - All are data parallel and extensions of the C language
- An assembler was also written for the WaveTracer (and probably the Zephyr).
  - It was intended for use only by company technicians.
  - Information about assembler were released to WaveTracer customers on a "need to know" basis.
  - No manual was written but some details were recorded in a short writeup/report.
  - Professor Potter has a reasonable amount of information about assembler to use in putting the ASC language on the WaveTracer

- MultiC is an extension to ANSI C, as documented by the following book:
  - The C Programming Language, Second Edition, 1988, Kernighan & Richie.
- The manual for the MultiC language is a spiral bound book titled "The MultiC Programming Language" by WaveTracer, 1991.
- The WaveTracer computer is called a Data Transport Computer (DTC) in manual
  - a large amount of data can be moved in parallel using interprocessor communications.
- Primary expected uses for WaveTracer were scientific modeling and scientific computation
  - Accoustic waves
  - heat flow
  - fluid flow
  - medical imaging
  - molecular modeling
  - neural networks
- The 3-D applications are supported by a 3D mesh on the WaveTracer
  - Done by sampling a finite set of points (nodes) in space.

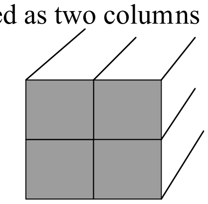
WaveTracer Architecture Background

- Architecture for Zephyr is fairly similar
  - Exceptions will be mentioned whenever known
- Each board has 4096 bit-serial processors, which can be connected in any of the following ways:
  - 16x16x16 cube in 3D space
  - 64x64 square in 2D space
  - 4096 array in 1D space
- The 3D architecture is native on the WT and the other networks are supported in hardware using primarily the 3D hardware
  - The Zephyr probably has a 2D network and only simulates the more expensive 3D network using system software.
- WaveTracer was available in 1, 2, or 4 boards, arranged as follows:
  - 2 boards were arranged as a 16x32x16 cube
    - one cube stacked on the top of another cube
    - 8192 processors overall



WaveTracer Architecture (Cont)

- Four boards are arranged as a 32x32x16 cube
  - 16,384 procesors
  - Arranged as two columns of stacked cubes



- Computer supports automatic creation of virtual processors and network connections to connect these virtual processors.
  - If each processor supports k nodes, this slows down execution speed by a factor of k
    - Each processor performs each operation k times.
    - Limited by the amount of memory required for each virtual node
    - In practice, slowdown is usually less than k
- The set of virtual processors supported by a physical processor is called its *territory.*

Specifiers for MultiC Variables

- Any datatype in C except pointers can be declared *multi*
- This replicates the data object for each processor, to produce a 1,2, or 3 dimensional data object
- In a parallel execution, all *multi* objects must have the same dimension.
- The *multi* declaration follows the same format as ANSC C, e.g

  multi int imag, buffer;
- The *uni* operation is used to declare a scalar variable
  - Is the default and need not be shown.
  - The following are equivalent:

    uni int ptr;

    int ptr;
- Bit Length Variables
- can be of type *uni* or *multi*
  - Allows user to save memory
  - All operations can be performed on these bit-length values
  - Example: A 2 color image can be declared by

    multi unsigned int image :1;

    and an 8 color image by

    multi unsigned int picture:3;

Some Control Flow Commands

- For uni type data structures, control flow in MultiC is identical to that in ANSI C.
- IF-ELSE Statement
  - As in ASC, both the IF and ELSE portions of the code is executed.
  - As with ASC, the IF is a mask-setting operation rather than a branching command
  - FORMAT: Same as for C
  - WARNING: Both sets of statements are executed.
    - Even if no responders are active in one part, the sequential commands in that part are executed.
      - Differs from ASC here
      - Example: count := count + 1;
- WHILE statement
  - The format used is

    while(expression)
  - The repetition continues as long as expression is satisfied by one or more responders.
  - While does not change scope (i.e., the mask).
  - Commands are executed by all processors that were active upon initially reaching the WHILE

Other Commands

- Jump Statements
  - goto, return, continue, break
  - These commands are in conflict with structured programming and should be used with restraint.
- Parallel Reduction Operators

*=	Accumulative Product
/=	Recripocal Accumulative Product
+=	Accumulative Sum
-=	Negate & then Accumulative Sum
&=	Accumulative bitwise AND
|=	Accumulative bitwise OR
>?=	Accumulative Maximum
<?=	Accumulative Minimum

- Each of the above reduction operations return a uni value and provide a powerful arithmetic operation.
  - Each accumulative operation would otherwise require one or more ANSI C loop constructs.
  - Example: If A is a multi data type

    largest_value = >?= A

    smallest_value = <?= A

- Data Replication
  - Example:

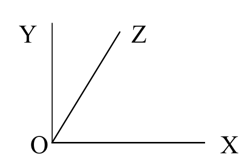
    multi int A = 0;

    .

    .

    A = 2;
  - First statement stores 0 in every cell in A field
  - Last statement stores 2 in every cell in A field
- Interprocessor Communications
  - Operators have the form

    [dx; dy; dz]m
  - This operator can shift the components of the multi variable m of all active processors along one or more coordinate dimensions.
  - Example: A = [-1; 2; 1]B
    - Causes each active processor to move the data in its B field to the A field of the processor at the following location:
      - one unit in the negative X direction
      - one unit in the positive Y direction
      - two units in the positive Z direction
  - Coordinate Axes



- Conventions:
  - If value of dz operator is not specified, it is assumed to be 0
  - If the values of dy and dz operators are not specified, both are assumed to be 0
  - Example: [x; y]V is the same as [x; y; 0]V
- Inactive processor actions
  - Does not send its data to another processor
  - Participates in moving the data from other processors along.
- Transmission of data occurs in lock step (SIMD fashion) without conjestion or buffering.

• Coordinate Functions
- Used to return a coordinate for each active virtual processor.
- Format: multi_x(), multi_y(), and multi_z()
- Example:

```
If(multi_x() == 0 && multi_y == 2 && multi_z == 1)
                u =   += A;
```

  - Note that all processors except one at (0,2,1) are inactive with the body of the IF.
  - The accumulated sum of the active components of the multivariable A is just the value of the component of A at processor (0,2,1)
  - Effect of this example is to store the value in A at (0,2,1) in the uni variable u.

  - If the second command in the example is changed to
    A = u;
    the effect is to store the contents of the uni variable u into multi variable A at location (0,2,1).
  - (see manual pg 11-13,14 for more details)

• Arrays
- Multi-pointers are not supported.
  - Can not have a parallel variable containing a pointer to each component of the array.
- uni pointers to multi-variables are allowed.
- Array Examples:

```
int array_1 [10];
int array_2 [5][5];
multi int array_3 [5];
```

  - array_1 is a 1 dimensional standard C array
  - array_2 is a 2 dimensional standard C array
  - array_3 is a 1-dimensional array of multi variables

• MULTI_PERFORM Command
- Command gives the size of each dimension of all multi-values prior to calling for a parallel execution.
- Format:

```
multi_perform(func, xsize, ysize, zsize)
```

  - Here, "func" is the function being executed.
  - "xsize", "ysize", "zsize" are positive integers specifying the DTC network configuration.
  - If "zsize" is 1, then multi_perform creates a 2D grid of size "xsize ¥ ysize"

- multi_perform is normally called within the main program.
  - Usually calls a subroutine that includes all of the
    - parallel work
    - parallel I/O
- The main program usually includes
  - Opening and closing of files
  - Some of the scalar I/O
  - define and include statements
- When multi_perform is called, it initializes any *extern* and *static multi* objects
- In the previous example, multi_perform calls *func.* After *func* returns, the multi space created for it becomes undefined.
- The *perror* function is extended to print error messages corresponding to *errno* numbers resulting from the execution of multiC extensions.
  - Has the following format

```
if(multi_perform(func,x,y,z)) perror(argv[0]);
```

  - See usage in the examples in Appendix A
  - More information on page 11-2 of manual

• Examples in Manual
- Many examples in the manual
- 17 in appendices alone
- Also stored under *exname.mc* in the MultiC package
  - They can be compiled and executed.

The AnyResponder Function

• Code Segment for Tallying Responders

```
unsigned int short, tall;
multi float height;
load_height;    /* assigns values to height */
if(height >= 6)
        tall =   += (multi int)1;
else
        short =   += (multi int)1;
printf("There are %d tall people  \n", tall);
```

• Comments on Code Segment
- Note that the construct
  += (multi int)1
  counts the active PE (i.e., responders).
- This technique avoids setting up a bit field to use to tally active PEs.
  - Instead sets up a temporary multi variable.
- Can be used to see there is at least one responder at a given step.
  - Check to see if resulting sum is positive
- Provides technique to define the AnyResponder function needed for associative programming

Accessing Components from Multi Variables

- Code from page 11-14 of MultiC manual

```
#include <multi.h>    /* includes multi library */
#include <stdio.h>
#include <stdio.h>
void work (void)
{  uni int  a, b, c, u;
    multi int n;
    /*  Code goes here to assign values to n */
    /* Code goes here to assign values to a, b, c  */
    if (mult_x() == a  &&  multi_y() == b
                    &&  multi_z() == c)
        u  =   += n;   /* Assigns value of n at PE(a,b,c) */
}
 int main (int argc, char, *argv[])
{   if( multi_perform(work, 7 , 7, 7))
        perror = argv{0};
    exit(exit_success);
}
```

- To place a value of 5 into the selected location, replace the line "u = +=n" with the line

  n = 5;
- The capability to read or place a value in a parallel variable at a selected position is essential for multiC to execute associative programs.

The *oneof* and *next* Functions

- Function *oneof* provides a way of selecting one out of several active processors
  - Defined in Multi Struct program (A.15) in manual
  - Procedure is essential for associative programming.
- Code for *oneof*:

```
multi unsigned oneof(void):1
{   /* Stores coordinate values in multi
   variables x and y  */
   multi unsigned x  =  multi_x(),
                  y  =  multi_y(),
                  uno:1 = 0;
   /*  Next select processor with highest
   coordinate value */
   if( x == >? x)
      if( y == >? y)
        uno = 1;

   return uno;
}
```

- Note that multi variable *uno* stores a 1 for exactly one processor and all the other coordinates of *uno* stores a 0.
- The function *oneof* can be used by another procedure which is called by *multi_perform*.
  - An example of *oneof* being called by another procedure is given on pages A47-50 of the manual.
  - Should be useable in the form

    if(oneof())   /* Check to see if an active responder exists */
- Following preceding code, we can assign

  a = >? x; b = >? y; c = >? z

  Then (a,b,c) stores the location of the PE selected by *oneof*

- Preceding procedure assumed a 2D configuration of processors with z=1.
  - If configuration is 3D, the process selecting the coordinates can be continued by selecting the highest z-coordinate.
- **Stepping through the active PEs (i.e., *next*)**
  - Provides the MultiC equivalent of the ASC *next* command
  - An additional one-bit multi int called *bi* (for "busy-idle") is needed.
  - First set *bi* to zero
  - Activate the PEs you wish to step through.
  - Next, have the active PEs to write a 1 into *bi.*
  - Use

    if(oneof())

    to restrict the mask to one of the active PEs.
  - Perform all desired operations with active PE.
  - Have active PE set its *bi* value to 0 and then exit the above *if* statement.
  - Use the **+=** (accumulative sum) operator to see if any PEs remain to be processed.
    - If so, return to step above calling *oneof*
    - This step can be implemented using a *while* loop.

Printing values of a Multi Variable

- Example: Print a block of the 2D bit array called *image*.
  - A function *select_int* is used which will return the value of *image* at the specified (x,y,z) coordinate.
  - The printing occurs in two loops which
    - increments the value of x from 0 to some specified constant.
    - increments the value of y from 0 to some specified constant.
  - This example is from page 8-1 of the manual and is part of a larger example on pgs A16-18.
  - *select_int* Function

```
select_int (multi *mptr, int x, int y, int z)
/* Here, *mptr is a uni pointer to type multi */
{  int r
   if( multi_x == x &&
          multi_y == y &&
          multi_z == z)
   /* Restricts scope to the one PE at (x,y,z) */
          r = 1 = *mptr;
   return r;
   /* Transfers binary value of multi variable at location
   (x,y,z) to the uni variable. */ }
```

– The two loops to print a block of values of the *image* multi variable.

```
for( y = 0; y < ysize; y++)
{ for (x =0; x < xsize; x++)
      printf( "% d", select_int(&image,x,y,z)
      printf( "\n");
}
```

- Above technique can be adapted to print or read multi variables or part of multi variables.
  – Efficient as long as the number of locations accessed is small.
- If I/O operations involving large multi variables are needed, more efficient data transfer functions described in manual (Chapter 8 and Sections 11.2.2 and 11.13.6) should be used.
- The functions *multi_fread* and *multi_fwrite* are analogous to *fwrite* and *fread* in C. Information about them is given on pages 11-1 to 11-4 of the manual.
- The functions

  multi_from_uni ...

  multi_to_uni ...

  (where "..." is replaced with char, short, int, long, float, etc.) are described on pages 11-17 to 11-22.
  – Functions are also used in several examples.

- **Loading and Unloading**
- Allows the user to transfer whole arrays from "uni" to/from "multi".
  – `multi_from_uni_int( mptr *, uniptr *, x, y, z );`
  – `multi_to_uni_int( mptr *, uniptr *, x, y, z );`
  – Also for:
    - char
    - short
    - int
    - long
    - float
    - double
    - cfloat
    - cdouble
  – Example:
    - `multi_from_uni_int( &mtemp, &utarget[ 0][ 0][ 0] , TSIZEX, TSIZEY, TSIZEZ );`

Compiling and Executing Programs on the WaveTracer

- MultiC on WaveTracer
  – login on intrepid
  – Location of WaveTracer Software is in /local/opt/wt
    - Put that subdirectory in your PATH environment variable.
  – Command to compile (note extension)
    - mcc filename.mc
    - mcc -o executable_name filename.mc
- Executing ASC on the WaveTracer
  – This is not presently installed on intrepid!!!!!
  – login on intrepid
  – cd /usr/local/ASC/ASC
  – Command to compile

    asc -wt file.asc [< file2.asc]
  – Command to execute

    ???????????????????

- **Recursion**
- It is possible to write recursive "multi" functions in multiC, but you have to test if there are active PEs still working.
- Consider the following multiC function

```
multi int factorial( multi int n )
{
   multi int r;
   if( n != 1 )
      r = (factorial(n-1)*n);
   else
      r = 1;
   return( r );
}
```

- What happens?

- Recursion

```
multi int factorial( multi int n )
{
   multi int r;

   /* stop calculating if every component has
   been computed */
   if( ! |= (multi int) 1 )
        return(( multi int ) 0 );

   /* otherwise, continue calculating */
   if( n > 1 )
        r = factorial( n-1 ) * n;
   else
        r = 1;

   return( r );
}
```