Chapter 11

The *Lisp Programming Language

The *Lisp language is a high-level programming language for the Connection Machine system. Based on the Common Lisp programming language, *Lisp allows you to write data parallel programs for the CM using the data types, programming constructs, and programming style of Lisp. Programs written in *Lisp make full use of CM hardware, yet at the same time retain the clarity, expressiveness, and flexibility of Lisp.

The *Lisp language extends the Common Lisp language by providing parallel equivalents for the basic operations of Common Lisp, along with operations that are unique to data parallel programming, such as processor selection, parallel prefix calculations, interprocessor communication, and data shape specification.

A *Lisp program is simply a Common Lisp program that includes calls to *Lisp operators. A call to a *Lisp operator causes all active CM processors to execute that operation in parallel. Thus, *Lisp is fully compatible with Common Lisp; programs written in Common Lisp will run unmodified in *Lisp.

*Lisp functions and macros are defined via **`defun`** and **`defmacro`**, just as in Common Lisp. *Lisp programs are compiled by the *Lisp compiler, which includes (and is invoked in the same ways as) the Common Lisp compiler. This means that programs in *Lisp and Common Lisp can be written, compiled, and tested with the same editors and debuggers.

Copyright © 1993 Thinking Machines Corporation

11.1 Structuring Parallel Data

Scalar and Parallel Data

*Lisp is an extension of Common Lisp and therefore includes all the standard Common Lisp data types. These data types are collectively referred to as *scalar* data. *Lisp also supports an additional parallel data type, called a *pvar*. A pvar is a *parallel* variable, that is, a single variable with a separate, modifiable value in each processor of the CM. Operations performed on a pvar are performed simultaneously by all active CM processors, with each processor modifying only its own value for the pvar. Many of the scalar data types in Common Lisp have corresponding pvar equivalents. The eight basic pvar data types are boolean, integer, floating-point, complex, character, array, structure, and front-end value.

Creating Pvars in *Lisp

There are three basic ways to create, or *allocate*, a pvar in *Lisp, each designed to serve a specific purpose, as shown in the examples below:

```
(!! 5)                    ;; Allocating a temporary pvar

(defpvar my-five-pvar 5) ;; Allocating a permanent pvar

(*let ((my-pi!! pi))      ;; Allocating a local pvar
    (*!! 2 my-pi!!))
```

As these examples show, *Lisp supports temporary, permanent, and local pvars.

- Temporary pvars are allocated by the **!!** (bang-bang) function, which takes a single scalar value as its argument and returns a temporary pvar with that value in every processor.
- Local pvars are allocated by the ***let** and ***let*** functions. They exist for the duration of a body of *Lisp code.
- Permanent pvars are allocated by the **defpvar** function. They remain in existence until specifically deallocated.

November 1993
Copyright © 1993 Thinking Machines Corporation

Defining the Shape of the Data

The shape of the data stored in a pvar is determined by a grid of processors that the CM is currently simulating. The defining property of a processor grid is its *geometry*: the rank of the simulated grid and the sizes of its dimensions.

The combination of a particular grid geometry and a set of pvars that share that geometry is called a *virtual processor* set (VP set). For example, the expression

```
(def-vp-set my-vp-set '(64 64)
 :*defvars    ((x 1 nil fixnum-pvar)
               (y 1.0 nil single-float-pvar)))
```

defines a VP set named **my-vp-set** with 64 x 64 processors and associates two permanent pvars with it: an integer pvar **x** and a single-precision floating-point pvar **y**.

Because the CM can simulate many grids within a single program, *Lisp uses the concept of a *current VP set* to determine which VP set is active. Unless otherwise specified, all pvar operations take place within the current VP set. If no VP set has been defined, all pvar operations occur within a *default VP set* that is automatically defined whenever *Lisp starts up.

Processor Addressing

An important feature of the simulated grids defined by VP sets is that they permit the assignment of *addresses* to processors. There are two basic methods used to assign addresses to processors on the CM: *send addressing* and *grid addressing*.

Each processor has a unique numeric *send address* based upon its location within the physical hardware, accessible via the *Lisp operation **(self-address!!)**.

Each processor also has a *grid address*, a sequence of coordinates that defines its position in the *n*-dimensional grid of processors the CM is currently simulating. The *Lisp operation **(self-address-grid!! n)** returns a pvar whose value in each processor is the coordinate of that processor along the **n**th dimension of the current grid.

Copyright © 1993 Thinking Machines Corporation

Accessing and Copying Parallel Data

*Lisp allows you to access pvar values on a per-processor basis, to copy the value of one pvar into another, and to display the elements of a pvar over a range of processors. For example:

- **(pref my-pvar 10)** returns the value of **my-pvar** in processor 10.
- **(*setf (pref my-pvar 10) 123)** stores the quantity 123 into processor 10 of **my-pvar**.
- **(*setf (pref my-pvar (cube-from-grid-address 5 7)) 111)** stores 111 into **my-pvar** at grid location (5,7).
- **(*set pvar1 pvar2)** copies the contents of **pvar2** into **pvar1** in all active processors.
- **(*set pvar1 5)** stores the value 5 into **pvar1** in all active processors.

The *Lisp operation **ppp** (short for **pretty-print-pvar**) displays the values of a pvar. For example, the expression

```
(ppp (self-address!!) :end 20)
```

displays the send addresses of the first 20 processors:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

11.2 Computing in Parallel

The parallel operations supplied by *Lisp are modeled very closely on the existing scalar operations of Common Lisp and include parallel equivalents for most Common Lisp functions and macros. These parallel operations typically have the same name as their scalar Common Lisp counterparts, with either the characters "**!!**" added to the end or an asterisk "*****" appended to the front. The characters "**!!**" are meant to resemble the mathematical symbol ||, which means *parallel*. The asterisk similarly denotes the concept of an operation taking place in parallel. For example, the parallel version of the Common Lisp **mod** function is **mod!!**, and the Common Lisp **if** operator has two *Lisp equivalents, **if!!** and ***if**.

November 1993
Copyright © 1993 Thinking Machines Corporation

Most *Lisp operators take pvars as arguments and return a pvar result. In general, if a Common Lisp operation takes arguments of a specific data type, the *Lisp equivalent for that operation takes pvars of that data type as arguments and returns an appropriately typed pvar result.

For example, the functions **+!!**, **-!!**, ***!!**, and **/!!** perform the same operations as the Common Lisp functions **+**, **-**, *****, and **/**, but take numeric pvars as arguments and perform the appropriate arithmetic operation in parallel. The *Lisp expression

```
(*set pvar2 (+!! pvar1 (*!! pvar1 pvar2)))
```

multiplies the values of **pvar1** and **pvar2** in all active processors, adds the value of **pvar1**, then stores the result in **pvar2**.

*Lisp includes parallel versions of Common Lisp functions for many data types, including operations for complex and character pvars. *Lisp also includes an extensive selection of operators for manipulating array, vector, string, sequence, and structure pvars. There are even operations that allow you to create pvars that reference front-end data structures (such as symbols and lists).

In addition, *Lisp redefines many Common Lisp operations so that they will accept pvar arguments and will call the appropriate *Lisp operations to compute the result. This means that the above ***set** example can be rewritten as:

```
(*set pvar2 (+ pvar1 (* pvar1 pvar2)))
```

Selection of Active Sets of Processors

Parallel computations can be performed in all processors simultaneously, or in a specific subset of *active* processors selected by the user. Pvar values in inactive processors are not changed. *Lisp provides several macros for selecting the current set of active processors (sometimes referred to as the *currently selected set*).

The most basic processor selection operators are ***when** and ***unless**. Similar to their Common Lisp counterparts, these operators conditionally evaluate a body of code based on the result of a test. The difference is that the test controls which processors will evaluate the code, not whether the code will be evaluated at all. In the following code sample, ***when** is used to select all processors with odd send addresses. The value of **my-pvar** in those processors is then negated.

Copyright © 1993 Thinking Machines Corporation

```
(*set my-pvar (self-address!!))

(*when (oddp!! (self-address!!))
    (*set my-pvar (-!! my-pvar)))

(ppp my-pvar :end 19)
0 -1 2 -3 4 -5 6 -7 8 -9 10 -11 12 -13 14 -15 16 -17 18
```

The ***all** construct unconditionally selects all processors for the duration of a body of *Lisp code. For example, evaluating the expression

```
(*all (*set my-pvar 10))
```

ensures that the value of **my-pvar** in all processors is 10, regardless of the state of the currently selected set.

11.3 Communicating in Parallel

Like all CM languages, *Lisp supports both regular and irregular communication. For example:

- **news!!** causes each active processor to get a value from another processor a fixed distance away on the grid.
- ***news** causes each active processor to send a value to another processor a fixed distance away on the grid.
- **pref!!** allows each active processor to get a value from any other processor in the grid.
- ***pset** allows each active processor to send a value to any other processor in the grid.

If two or more processors attempt to read the data of a single processor, they all receive the same correct data. If two or more processors attempt to write to the same address, the user can specify how they are to be combined (for instance, by summing the values).

November 1993
Copyright © 1993 Thinking Machines Corporation

11.4 Transforming Parallel Data

*Lisp contains many functions to help perform transformations on data. These include operators computing parallel prefixes (scanning) of data, spreading data across the processors of the CM, and sorting and enumeration of pvar values. Some examples:

- **`scan!!`** and **`segment-set-scan!!`** permit the selection of many kinds of scanning operations, such as addition/multiplication of values; taking the maximum and minimum of values; taking the logical/arithmetic AND, OR, and XOR of values; and even simply copying values across the processor grid.

  The **`scan!!`** operation accepts a segmentation argument for simple uses of this feature. The **`segment-set-scan!!`** operation uses a special type of pvar, a *segment set* pvar, to allow much finer control over the segmentation of processors than **`scan!!`** provides.

- **`spread!!`** replicates the value of a pvar at a given coordinate to all processors along a selected dimension of the currently selected grid. A related operation, **`reduce-and-spread!!`**, combines the operations of scanning and spreading.

- The **`sort!!`** operator reorders the values of a numeric pvar into ascending order.

- The **`enumerate!!`** operator assigns to each currently active processor a distinct integer between 0 (inclusive) and the number of active processors (exclusive).

Copyright © 1993 Thinking Machines Corporation