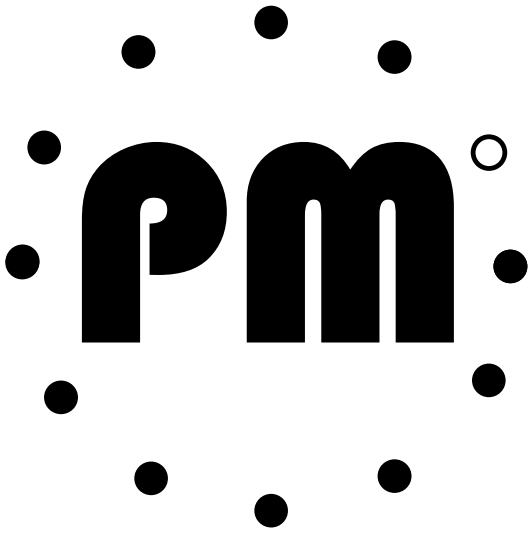

PM – Programming Parallel Models

Version 0.1

Language Reference (incomplete)

Contents

Background

PM (Parallel Models) is a new programming language designed for implementing environmental models, particularly in a research context where both ease of coding and performance of the implementation are both essential but frequently conflicting requirements. PM is designed to allow a natural coding style, including familiar forms such as loops over array elements. Language elements have been included (and more importantly excluded) from the language in a careful combination that enables the language implementation to both vectorise and parallelise the code. The following goals guided the development of the PM language:

- Language structures facilitate the generation of vectorised and parallelised code
- Vectorisation and parallelisation should be explicit and configurable
- Programmers should be able to concentrate on coding the model
- The language should concentrate on parallel structures required for numerical computation
- PM code should be as readable as possible by those not familiar with the language.
- PM should be formally specified – not defined by an implementation.

PM draws inspiration from many sources. Notable influences include: ZPL (http://research.cs.washington.edu/zpl/overview/overview.html), Parasail (parasail-lang.org), Go (golang.org) , NESL (http://www.cs.cmu.edu/~scandal/nesl.html) and llc (Reyes *et al*.,2009, 16th European PVM/MPI Users Group Meeting). Acknowledgement must also be made to two recent languages tackling similar problems: Chapel (chapel.cray.com) and Julia (julialang.org).

Conventions used to define language syntax

PM syntax will be described using the following extended BNF notation:

name ::= *list*	Define a non-terminal element in terms of other elements
name	Non terminal element
dim	Keyword
'**>=**'	Character combination
[*elements*]	Elements are optional – may appear zero or one times
{ *elements* }	Elements may appear zero, one or more times
el1 | *el2*	Exactly one of the listed element sequences must be present
(el1 | el2)	Brackets may be used to group selections that are not enclosed by {} or []

Lexical Elements

Comments start with a '!' and continue to the end of the line

```
! Comments are ignored
```

PM uses upper and lower case letters, digits and a number of special symbols:

letter::= '**a**' |'**b**' |'**c**' |'**d**' |'**e**' |'**f**' |'**g**' |'**h**' |'**i**' |'**j**' |'**k**' |'**l**' |'**m**' |'**n**' |'**o**' |'**p**' |'**q**' |'**r**' |'**s**' |'**t**' |'**u**' |'**v**' |'**w**' |'**x**' |'**y**' | '**z**'
'**A**' |'**B**' |'**C**' |'**D**' |'**E**' |'**F**' |'**G**' |'**H**' |'**I**' |'**J**' |'**K**' |'**L**' |'**M**' |'**N**' |'**O**' |'**P**' |'**Q**' |'**R**' |'**S**' |'**T**' |'**U**' |'**V**' |'**W**' |'**X**' |'**Y**' | '**Z**'
digit::= '**0**' | '**1**' | '**2**' |'**3**' | '**4**' | '**5**' | '**6**' | '**7**' | '**8**' | '**9**'

The following single characters and character combinations are used as ***delimiters***:

'**$**' '**&**' '**(**' '**)**' '******' '*****' '**,**' '**//**' '**/**' '**=>**' '**:**' '**||**' '**;**' '**@**' '**[**'
'**]**' '**_**' '**{**' '**|**' '**}**' '**-**' '**""**' '**+**' '**<**' '**<=**' '**=**' '**==**' '**>**' '**>=**' '**%**'
'**..**' '**.**' '**->**' '**::**'

To cater for environments with restricted character sets, the following substitutions are always permitted:
'**(/**' for '**[**' '**/)**' for '**]**' '**(:**' for '**{**' '**:)**' for '**}**' '**%%**' for '**@**' '**:/**' for '**|**'
'**://**' for '**||**'

White space characters (space, newline, form feed and horizontal tab) may appear between lexical elements (delimiters, names, keywords, numeric constants) but not within them. Names, keywords and numbers must be separated from each other by white space. Otherwise white space is optional.

New lines are not generally significant, but in most contexts where a semicolon '**;**' is expected/allowed by the syntax, the semicolon may be omitted if the following lexical (non-white-space) symbol starts on a new line. The exception is the use of a semicolon to separate loop and non-loop arguments in a procedure call.

Names have a maximum length of 100 characters. They are comprised of upper and lower case letters, decimal digits, and the underscore character '**_**'. They may not start with a digit. Letter case is significant.

name ::= ['**_**']*letter*{'**_**'|*letter*|*digit*}

Names that start with '**_**' are unique to the module within which they are defined. A module entity defined using such a name may not be referred to from another module.

The following are ***keywords*** and may not be used as names:

also	**and**	**arg**	**argc**	**build**	**by**
case	**check**	**conc**	**const**	**debug**	
default	**dim**	**do**	**else**	**elseif**	**enddebug**
enddo	**endfor**	**endif**	**endproc**	**endselect**	**endtype**
endwhile	**false**	**find**	**for**	**high**	**in**
include	**includes**	**invar**	**is**	**key**	**low**
if	**not**	**null**	**of**	**opt**	
otherwise	**over**	**param**	**proc**	**rec**	**or**
reduce	**render**	**repeat**	**requires**	**result**	**select**
seq	**step**	**struct**	**then**	**true**	**type**
until	**use**	**using**	**when**	**with**	**where**
while					

Numeric constants take the following forms*:*

number ::= [*integer_constant* | *real_constant* | *imaginary_constant*] [*bits*]
integer_constant ::= [{ *digit* } '**r**'] { [*digit* | *letter*] } ['**l**']
real_constant ::= { *digit* } ['**.**'*digit* { *digit* }] ['**e**'['**+**' | '**-**'] digit { *digit* }]['**d**']
imaginary_constant ::= [*integer_constant* | *real_constant*] '**i**'
bits ::= '_' { *digit* }

String constants are enclosed by double quotes'"'and may include any character other than '"'. They may not span more than one line.

```
"Hello World"
```

Modular Structure

A PM program consists of one or more ***modules***. A module defines a collection of ***procedures***, ***types*** and ***parameters*** (module level constants). A ***program module*** additionally contains executable statements. A ***library module*** may optionally contain a debug statement containing module testing code but may not contain any other executable statements. Module names are constructed from a sequence of standard PM names optionally joined by '**.**' delimiters. They are linked to source file names in an implementation-dependant manner. Module names starting with '**lib.**' refer to standard libraries.

module **::=** *program_module* | *library_module*
program_module::=
    [*decls*] *statements*
library_module::=
    decls [**debug** *statements* **enddebug**]
modname::=
    name { '**.**' *name* }

Names may optionally start with an underscore character. Such names are local to the module in which they are used and do not match names in any other module into which they are imported.

decls::=
    { *import* '**;**' }(*import* | *decl*) { '**;**'*decl* }
import::=
    use *modname* [*modifiers*] { '**,**' *modname* [*modifiers*] }

Modules may include other modules using a **use** statement. In its simplest form, **use** *module_name* , the statement imports all parameters, default values, types and procedures defined in the named module into the current module. The importing process does not indirectly import definitions accessed through **use** statements in the imported module.

A **use** statement must not result in two type or parameter definitions being accessible using the same name. Similarly, it must not result in procedure definitions accessed through the same procedure name having conflicting signatures. Types, procedures and parameters occupy separate name spaces – they may have the same name as each other. Name clashes may be avoided by using a qualifying clause in the **use** statement. This lists specific elements to import and optionally renames them:

modifiers::=
    '**{**' *modifier* { '**,**' *modifier* } '**}**'
modifier::=
    [**type** | **param** | **proc** | **default**] *name* '**=>**' *name*

```
use model4 {
      type model_params
      param theta =>model_theta
      test =>test_model
}
```

In this context, the => operator causes definitions accessed via a given name in the imported module to be made available using a different name in the current module. If the renaming operation is not qualified as **type**, **param** or **proc** then it is assumed to be a procedure (**proc**).

Type, procedure and parameter declarations follow any **include** statements in the module. The simplest of these is the **param** declaration which simply defines a named constant:

paramdec::=
    param *name* '**=**' *xexpr*

For example:

```
param pi = 3.14159265
```

param statements may refer to each other, irrespective of the order in which they are defined. However, such references must not be recursive or mutually recursive. There is no explicit restriction on the procedures employed within a **param** declaration expression – they may be interpreted as zero-parameter procedures with their own separate name space.

Type and procedure declarations are described later in this document.

In a program module, the optional declarations are followed by executable statements. Such a module may be run as a program. It may not, however, be used by other modules. Library modules may be used by other modules. They may not contain executable statements after the declarations except for an optional single **debug** statement that should contain module testing code.

Types, values and objects

PM programs operate on *values* stored in *objects*. A *type* consists of a (possibly infinite) set of values and/or other types and is defined using a *type constraint expression*. C*onformance* relationships are defined between types and between values and types:

- A value V conforms to an type T if V∈T or V∈U and U conforms to T
- Type T conforms to type U if T∈U or T∈V and V conforms to U

While the PM type system provides flexibility similar to some dynamically typed languages, it is designed for static analysis. Run time type inference and procedure selection should only be necessary in situations where specifically polymorphic values are generated (using ***polymorphic types*** or recursive procedure invocation). This may require type analysis between and across modules.

The universal type, which includes all other types, is denoted using the keyword **any**. Explicit use of the universal type is usually not required; the absence of a type constraint serves a similar purpose in most cases where a type expression is expected.

User defined types

User defined types associate a name with a given set of type constraint expressions, defining the new type as a set of the listed types. A type is defined in a type definition:

typedecl::=
    type *name* ['**{**' *typeparams* '**}**'] [**in** *namelist*] (**is** *typelist* | **includes** *typelist* | **also includes** *typelist*)
typeparams::=
    name ['**:**' *type*] { '**,**' *name* ['**:**' *type*] }

The simplest type definition simply associates a name with a set of types:

```
type x is int, struct{x:int,y:int}
```

A ***parameterised type*** declaration defines a template that may be parameterised by one of more types in order to generate an ***actual type*** that may be used in constraint expression. An actual type is derived from a type template by providing the correct number of type constraint expressions as type arguments.

```
type point{t:num} is struct{ x:t, y:t}
type integer_point is point{int}
type num_point is point{}
```

If ***parameter type constraints*** are present, then the type arguments used to create an actual type must conform, parameter by parameter, to the corresponding type parameter type constraints, or be missing. Missing type arguments are assumed to be equal to the corresponding type parameter constraint or to **any** if no such constraint is present.

Type templates are characterised by their name and number of parameters, but not their respective type parameter constraints. Thus **R{t}**, **T{t}** and **T{t,u}** each refer to different type templates while **T{t,u}**, **T{r,s}** and **T{,}** refer to the same type template . Two type template definitions may not both refer to the same template name if they have the same number of parameters.

An ***open type declaration***, using the **includes** syntax provides an incomplete definition for a type which may be added to by other definitions. The type definition may be augmented using an **also includes** type definition or by using by declaring another expression to be **in** that type:

```
type a includes int, real
type a also includes string
type b in a is cpx    ! Type a is int, real, string, b
```

Augmenting type declarations do not have to be in the same module as the original **includes** declaration and do not have to precede the **includes** declaration in the module or program text.

Open type declarations may also be parameterised:

```
type point{T:num} includes rec point{ x:T,y:T }
type point{T} also includes rec point{ x:T,y:T,z:T}
```

If an **also includes** type definition is parameterised, then the type constraints associated with its parameters must either be absent or must conform, parameter by parameter, with the type constraints in the **includes** definition to which it refers. If type constraints are present, the **also includes** definition only augments the type template when the type arguments in the actual type expression conform to the types parameters in the **also includes** definition. For example:

```
type point{T:int16} also includes rec point{ combined_index: int32 }

proc f (  t : point{int},    ! Matches rec point{x:int,y:int}
          u : point{int16)   ! Matches both rec point{x:int,y:int} and
                             !  rec point{ combined_index:int32 }
     )  ...
```

A type definition may not refer to itself recursively unless that recursive reference occurs within the definition of an ***optional type***.

The PM system defines a number of ***intrinsic types***. These are in effect user defined types, declared in a system module implicitly imported into every other module.

Numeric types

PM supports a range of integer types, defined in the following table. The definitions of these types are flexible to enable portability of code. Code requiring a strict bit size should use inquiry functions to check that a given type meets the expected requirements.

int	*short integer*	*System defined standard integer*	
long	*long integer*	*Integer capable of counting elements in largest possible array*	
int8	*8-bit integer*	*OR smallest integer holding -127..+127*	
int16	*16-bit integer*	*OR smallest integer holding -32767..+32767*	
int32	*32-bit integer*	*OR smallest integer holding -2147483647..+ 2147483647*	*OR same as* **int**
int64	*64-bit integer*	*OR smallest integer holding -9,223,372,036,854,775,807.. +9,223,372,036,854,775,807*	*OR same as* **int32**
int128	*128-bit integer*	*OR smallest integer holding -170,141,183,460,469,231,731,687,303,715,884,105,727.. +170,141,183,460,469,231,731,687,303,715,884,105,727*	*OR same as* **int64**

Integer constants may be defined with respect to any base between 2 and 62 by specifying the base followed by '**r**' and then the digits, with '**a**'..'**z**' and '**A**'..'**Z**' representing 10 to 36 respectfully (case insensitive).

```
123        2r101011101   16rdeadbeef
```

By default integer constants yield values conforming to **int**. Long integer constants should append an **l** to the value. For other integer types, append an underscore and the corresponding (requested) number of bits:

```
889874324897284746                      ! long
255_8                                   ! int8
2r10101010101010101010101010101010_32   ! int32
```

PM also defines a set of real (floating point) types:

real	*Standard (system defined) floating point value*		
double	*Standard (system defined) double precision floating point value*		
real32	*32-bit floating point*	*OR real number with ≥1 decimal digits in mantissa*	*OR same as* **real**
real64	*64-bit floating point*	*OR real number with ≥15 decimal digits in mantissa*	*OR same as* **real32**
real128	*128- bit floating point*	*OR real number with ≥24 decimal digits in mantissa*	*OR same as* **real64**

Real constants must contain a decimal point and may contain an exponent preceded by '**e**'.

```
-5.2       2e3          4.2e-20
```

By default real constants yield values conforming to type **real**. To specify a **double** constant, append a **d**. For other real types append and underscore and the assumed number of bits:

```
-5.2d      ! double precision
3.2e-3     ! real
1.2e-2_32  ! real32
```

Complex types are defined as follows:

cpx	*Complex number formed from **real** components*	
double_cpx	*Complex number formed from **double** components*	
cpx64	*64-bit complex number*	*OR complex number based on* **real32** *components*
cpx128	*128-bit complex number*	*OR complex number based on* **real64** *components*
cpx256	*256-bit complex number*	*OR complex number based on* **real128** *components*

Imaginary constants terminate with a letter '**i**' or '**j**'.

```
3.0i          ! cpx
1.0i_64       ! cpx64
-2.2-30di     ! double_cpx
```

The following intrinsic types define groups of numeric types:

any_int is int, long, int8, int16, int32, int128
any_real is real, double, real32, real64, real128
any_cpx is cpx, double_cpx, cpx64, cpx128, cpx256
int_num includes any_int
real_num includes any_int, any_real
cpx_num includes any_int, any_real, any_cpx
num includes cpx_num
std_int is int, long
std_real is real,double
std_cpx is cpx, double_cpx
std_num is std_int, std_real, std_cpx

Numeric type balancing

Values of different numeric types may appear together in numeric expressions. The rules more mixed types are:

The most general type is selected for the results and the least general value converted to that type. Generality is defined as follows (least..most):

int long int8 int16 int32 int64 int128 real double real32 real64 real128 cpx double_cpx cpx64 cpx128 cpx256

Non-numeric types

The **bool** type contains the logical values **true** and **false**

The **string** type contains values that are arbitrary length character strings. ***String constants*** are enclosed by double quotes '"' and may include any character other than '"'. They may not span more than one line.

```
"Hello World"
```

The **name** type contains values that are valid PM names. Name constants are formed by preceding a name by the dollar symbol:

```
$x
```

The **proc** type contains values that are valid PM procedure names (including operator symbol). Procedure constants are formed by preceding a name by the **proc** keyword:

```
proc cell_model
proc +
```

The **proc** keyword may be omitted for procedure names which are PM names (but not operator symbols or reserved words) which are not the same as the names of local variables or constants and not the same as any visible parameter definition.

The type **type** contains types used as first class values. Type values are created using:

type *type*

Structures and records

Structures and records provide a mechanism to create aggregate values. They are very similar, except that it is not possible to alter a single component of a record, while this is possible for a structure.

A structure or record values is created using a generating expression:

[**struct** | **rec**] [*name*] '**{**' *name* '**=**'*exp* { '**;**' *name* '**=**'*exp* } '**}**'

For example:

```
struct { name=”John Smith”, age=42 }
rec point { x=2, y=4 }
```

A structure or record value associates each component with a name. The structure of record value may optionally tagged with a further name. Component names are local to the structure or record. The name tagging the whole structure or record resides in a name space that is global to the program (unless preceded by an '_' in which case it is local to the given module)

A structure or record type is constructed using:

[**struct** | **rec**] [*name*] '**{**'*name* ['**:**' *type*] { '**,**' *name* ['**:**' *type*] } '**}**'

For example:

```
struct { name:string, age:int }
rec point { x:int, y:int }
```

A structure or record value is a member of a given structure or record type if all of the following apply:

1. The value is a record and the type is defined using **rec** or the value is a structure and the type **struct**
2. The name tagging the value matches the corresponding name tag given in the **struct**/**rec** type, or both are absent
3. The same component names are present (not necessarily in the same order)
4. If a type constraint is associated with a component name in the type constraint expression then the corresponding component of the structure or record value, associated with the same name, conforms to that type constraint.

A structure or record type *T* conforms to a structure or record type *U* if:

1. Either both *T* and *U* are record types or both are structure types.
2. The name tag for *T* is the same as the name tag for *U*, or both are absent.
3. *T* and *U* have the same number of components with the same component names (not necessarily in the same order).

4. If for any given component name, *U* specifies a type constraint for that name, then *T* must also specify a constraint type conforming to the constraint type specified by *U*.

A given component of a structure or record may be accessed using the '**.**' operator. Structure components may be modified using the same operator.

```
person:=struct { name="John Smith", age=42 }
location:=rec point { x=2, y=4 }
person.age=person.age+1
print(person.age)
print(location.x)
```

In order to maintain the no-modification rule, a '**.**' operation by not be applied to a record on the left hand side of an assignment, on the right hand side of a reference connection for a non-constant reference or in the expression defining a reference parameter in a procedure call ('**&**')

Polymorphic types

A polymorphic value has an internal representation that is capable of representing any value conforming to a given type constraint. A value of a given polymorphic type is created using a generating expression:

'**<**' *type'* **>**' *subterm*

If a type is not given then it is set equal to the concrete type of the expression (see section on variables below). The value contained within a polymorphic value may be obtained using the unary '*****' operator.
For example:

```
Type int_or_string is int,string
a:= <int_or_string> 1
a= <int_or_string>'Hello'
print(*a)
```

A polymorphic type expression has the following form:

'**<**'[*type*]'**>**'

A polymorphic value conforms to type **<***U***>** if it was created using expression <T>*value* and T conforms to U. A polymorphic type **<***T***>** conforms to **<***U***>** if *T* conforms to *U*.

Optional types

An optional type is an extended form of polymorphic type which can hold any value conforming to a given type constraint and can additionally hold a null value (designated by **null**)

<opt *type*>

A value of an optional type may be created in a similar manner to that of a polymorphic type:

```
a:= <opt int> 1
print(*a) ! Will print the number one
a= <opt int> null
print(*a) ! Will generate an error – null values cannot be accessed
```

Domains

A domain defines a set of indices over which a (possibly parallel) iteration may be performed and over which arrays may be defined. The intrinsic domain type domain is defined as:

- **type dom includes int, long, range{std_int}, seq{}, grid, vect_dom, mat_dom**

All valid domain values *D* support the **num_elem(***D***)** procedure which returns the number of elements in the domain.

Simple Domains

A simple domain is specified using an integer or long integer value. It represents the set of integers from zero to the given number minus one.

Ranges

Range types represent open or closed intervals. Range values are created using the '**..**', '**<..**','**..<**' or '**<..<**' operators. For numerical values mixed type promotion rules apply. E.g:

```
closed_range:= 0..10
part_open_range:= 0<..10          ! Contains 1..10
part_open_range2:= 0..<10         ! Contains 0..9
real_open_range:=0<..<10.0        ! Contains { x | x>0.0 and x<10.0 }
```

The two bounds of a range value *R* may be obtained using the procedures **low(***R***)** and **high(***R***)**.

The following intrinsic range type are defined:

```
range_base includes any_real
range  { t: range_base }
open_start_range { t: range_base }
open_finish_range { t: range_base }
open_range { t: range_base }
```

Sequences

A sequence type represents a sequence of values. Sequence values are created by applying the **by** operator to a range value. The second argument to **by** gives the step between sequence elements. Numeric balancing will apply between the start/finish values of the range and the step argument.

```
integer_sequence := 1..6 by 2
real_sequence := 3.2 .. 5.4 by 0.7
```

The step of a sequence *S* may be found using the procedure **step(***S***)**. The two bounds of the underlying range value may be obtained using the procedures **low(***S***)** and **high(***S***)**.

Sequence types are defined as:

```
seq  { t: range_base }
```

Block Sequences

A block sequence combines two other sequences. Values are created using the **block** function. Values in a block sequence are defined by the following code equivalence:

```
for i in block(M..N by S,P..Q by R) do print(i) endfor
! is equivalent to
for i in M..N by S do for j in P..Q by R do print(i+j) endfor endfor
```

Note that for the above example to be valid, S must be greater than N-M. Block sequence types are defined as:

block{ t: range_base }

Grids

A grid type defines an *N*-dimensional (2<*N*≤7) grid of points. Each dimension of the grid may be defined elements of a domain, which may themselves be grids. There are six specific grid types of varying number of dimensions, and a generalised grid type encompassing all of these:

grid2d is grid{grid_base,grid_base}
grid3d is grid{grid_base,grid_base,grid_base }
grid4d is grid{grid_base,grid_base,grid_base ,grid_base }
grid5d is grid{grid_base,grid_base,grid_base ,grid_base ,grid_base }
grid6d is grid{grid_base,grid_base,grid_base,grid_base,grid_base,grid_base}
grid7d is grid{grid_base,grid_base,grid_base ,grid_base ,grid_base ,grid_base ,grid_base}
grid is grid2d, grid3d, grid4d, grid5d, grid6d, grid7d

Grid values are created by calling the **grid** procedure:

```
model_grid := grid (0..num_cols, 0..num_rows)
```

The value corresponding to each element of a grid value *G* may be obtained using: *G* **.d1**, *G* **.d2**, … , *G* **.d7**.

Domain shapes

All domains *D* have a corresponding ***domain shape***, which may be determined using the **shape(*D*)** procedure. This is defined as follows:

- The domain shape of a vector or matrix domain is equivalent to the original domain
- The domain shape of a grid domain **grid{***T1,T2,…***}** is given by
  proc shape(g: grid{*T1,T2,…***})= grid(num_elem(g.d1),1..num_elem(g.d2),** … **)**
- The domain shape of any other domain type *D* is given by **num_elem(*D*)**
- The programmer may provide specialised domain shape definitions for given types by providing additional definitions for the shape procedure. There is no restriction on the valid value types for a domain shape.

Arrays

An array type is a generic type indicating the storage of a set of values over corresponding to elements of a given domain.

One and two dimensional array values, with domains **int** and **grid{int,int}** respectively, may be defined using the following syntax:

'**{**' *list2d* '**}**'
list2d::=
exprlist {'**;**' *exprlist* } ['**;**']

For example:

```
array_1d := [ 1, 2, 3 ]                      ! domain is 1..3
array_2d := [ 1, 2, 3 ; 4, 5, 6; 7, 8, 9]    ! domain is grid(1..3,1..3)
```

For array values over other domains, or with elements which can store more generalised sets of values, the following extended syntax is available:

[' *list2d* '**]**'

If the array definition contains a single value inside the square brackets, then this is replicated over all elements of the array. For a one dimensional list, the number of elements in the list must equal the number of elements in the specified domain. For a two-dimensional list of size *M* by *N*, the domain must be a two dimensional grid with shape **grid(***M*,*N***)**.

Array assignment requires that the two participating arrays have the same domain shape. Arrays cannot change size – flexible size arrays must be implemented using polymorphic types. An array of type **int[int]** cannot change size. A value of type **<int[int]>** can refer to arrays of different sizes.

Array values may also be created using the **dim** operator:

expr **dim** *expr*

e.g.: `0.0 dim grid(0..3,0..3).`

As with other PM operators, the **dim** operator may be defined for other or more specific types (see later).

Array types are defined using:

[*type*] '**[**' *opttypelist* '**]**'
opttypelist::=
[*type*] { '**,**'[*type*] }

An array value conforms to array type T[U] if its element type conform to T and its domain conforms to U. If more than one type is present within the square brackets then the domain constraint is equal to a grid type parameterised by the supplied type list. For example:

```
int[int]     ! Integer valued array over integer domain
int[]        ! Integer valued array over any domain
[]           ! Any array over any domain – equivalent to
int[int,int] ! Integer array over gridded domain
             !- equivalent to int[grid{int,int}]
int[,,]      ! Integer array over any 3d gridded domain –
             !  -equivalent to int[grid{grid_base,grid_base,grid_base}]
```

Vectors and matrices

A vector is a numeric array defined over the **vect_dom** domain, which is a special one dimensional domain with a long integer dimension. A **vect_dom** value is created using the **vect_dom** procedure:

```
zero_vector = 0.0 dim vect_dom(1l..3l)
```

A matrix is a numeric array defined over the **mat_dom** domain, which is a special two dimensional domain with two long integer dimensions. A **mat_dom** value is created using the **mat_dom** procedure:

```
zero_matrix = 0.0 dim mat_dom(1l..3l,1l..3l)
```

Vector and matrix values may be created using a generating expression:

'**[**'*list2d* '**]**'
list2d::=

exprlist {**;**' *exprlist* } [**;**']

For example:

```
V :=  [1, 3, 1]
a : = [ 3, 0, 0
        0, 2, 1
        0, 0, 1 ]
```

As with array generators, all elements must have the same type (numeric type balancing is not invoked.) Note the use of the optional semicolon rule in the above example.

Matrix and vector values follow the usual rules for matrix multiplication. Vectors act as row or column matrices as appropriate in the context of matrix multiplication.

The vector and matrix intrinsic types are defined as:

- **vect is [vect_dom]**
- **mat is [mat_dom]**

Partitions

A ***partition*** of domain D over domain G associates a subdomain of D with each point in G. Conceptually, a partition acts as an array of sub-domain values although it is not defined as such.

The corresponding intrinsic type is **part{d:dom,g:dom}**

Partitions are generated using intrinsic procedures:

For example:

```
domain:= grid(16,16)
part_domain:= grid(4,4)
partition:= block(domain,part_domain)
block:=partition[2,2]                    ! yields grid(5..8,5..8)
```

Variables and constants

Objects are created and associated with names using a declaration statement:

statement::=
    const *assignlist*
    definition
    assignment
definition::=
    (*name*|'**_**') { '**,**' (*name*|'**_**') } '**:=**'*xexpr*
assignlist::=
    assignment { ',' *assignment* } *subexp*
assignment ::=
    lhs '**=**' *expr*
    lhs'**,**' *lhs*{ '**,**' *lhs* } '**=**' *call*

For example

```
x:= 4
const message = "Hello World"
```

An object has a ***concrete type*** that defines the set of values it is capable of storing. The concrete type of an object is determined by as follows:

1. If the initial value conforms to any of: **int long int8 int16 int32 int64 int128 real double real32 real64 real128 cpx double_cpx cpx64 cpx128 cpx256 bool string name proc** then that type is the concrete type of the object.
2. If the initial value is a structure or record, then the concrete type is a structure or record type with component type constraints given by the concrete types of the corresponding components, determined by applying these rules recursively.
3. If the initial value is a polymorphic value created using **poly** *T* **{***expr***}** then the concrete type is **poly{***T***}**
4. If the initial value is an array, then the concrete type is an array type with element and domain types equal to concrete types obtained recursively from corresponding components of the array value.

If the name is declared using '**:=**', values in the object may be changed later in the program. If it is declared using **const** then no changes to its value are permitted. The restriction for **const** names is enforced by forbidding them from appearing on the left hand side of an assignment statement or as a reference argument to a procedure. A name is defined until the end of the block of statements in which it is defined. Local (variable, constant and reference - defined later) names may not local shadow names defined in enclosing blocks or clash with procedure parameter names.

The values stored in variables may change over time. Values are changed in one of two ways: (i) using an assignment statement and (ii) by passing the variable as a reference argument to a procedure.

```
proc double(&x)=x*2
x := 4       ! Value stored in x is 4
x = x + 1    ! Value changed to 5
double(&x)   ! Value changed to 10
```

Some procedures return more than one value. In this case, multiple left hand sides are permitted:

assignment ::=
    lhs '**=**' *expr*
    lhs '**,**' *lhs* { '**,**' *lhs* } '**=**' *call*
lhs::=
    name qualifier |'**_**'
qualifier::=
    { '**.**' *name*| '**[**' *exprlist*'**]**' | '**{**' *exprlist*'**}**' }

For example:

```
x,y,_, z = returns_four_args(a)
```

An underscore may be used as a place filler, allowing a given returned value to be ignored. It is also possible for the left hand side of an assignment to refer to a component of an object.

```
a.f = b
a[3,2]=c
```

Procedures

A procedure defines an operation on a set of objects, which may change some of their stored values. Procedures may also return one of more values. A procedure declaration defines the name of the procedure, parameters on which it operates and their associated type constraints and any values returned.

```
proc square(x) = x**2

proc calc_stats(data: float dim any) do
     ……
     result=mean,std_dev
endproc
```

Procedure names may be simple names (with or without a leading underscore) or may be one of the following operators:

+ - * / ** == /= > >= and or // => dim = [] {} []= {}=
.. <.. ..< <..< by

Note the absence of **<** or **<=** from this list – these operators are always defined relative to > or >= respectfully.

Each procedure is associated with a **signature** consisting of the following:

1. The name of the procedure
2. The number of values the procedure will return.
3. The number of arguments the procedure will accept
4. The type constrains on the procedure's parameters
5. Which parameters are declared to be reference parameters
6. The number of channel variable parameters
7. Which parameters are required to be loop invariant

Items 6-7 are defined in the section on communicating procedures.

A procedure signature *P* is said to be **strictly more specific** than another procedure signature *Q* if:

1. The procedure names are the same.
2. The numbers of parameters are the same (allowing for variable argument definitions in either procedure).
3. All reference parameters occur in the same position.
4. The type constraint for each parameter of *P* conforms to the type constraint for the equivalent parameter of *Q*
5. Type constrains are not strictly equal to one another for at least one parameter position or for one position the parameter for P is **invar** and that for Q is not.

A procedure call invokes a procedure, supplying it with a list of values and/or objects to operate on. If the call is part of an assignment statement (discussed in detail below) then the call also provides a list of objects to receive any values generated.

```
u, v = myproc(x,&y,z*2)
```

If a procedure call is nested inside another procedure call, then the nested call is assumed to return a single value. Thus the following are equivalent:

```
y = f(g(x))

const _anon = g(x)
y = f( _anon )
```

Reference parameters, denoted by **&**, indicate that the procedure may change the value stored in corresponding object. Any procedure call must precede the corresponding argument with an **&**.

When encountering a procedure call, PM finds all procedures with signatures that ***match*** the call. A call matches a procedure signature if :

1. The procedure name is the same.
2. The number of arguments equals the number of parameters defined by the signature, or is greater than or equal to the number of parameters for a variable arguments signature.
3. All reference parameters are associated with a reference argument, starting with an &, in the same position.
4. The object supplied for each argument conforms to the type constraint for the corresponding parameter.
5. The number of channel variable parameters matches the number of channel variable arguments
6. Parameters constrained to be loop-invariant are associated with loop-invariant arguments

Items 5-6 are described in the section on communicating procedures. If more than one procedure matches a given call then PM will try to find a procedure that is strictly more specific than all the other candidate procedures. If no such procedure exists, then the call is ***ambiguous*** and the procedure selection is arbitrary. This condition may be checked for by the PM system and warnings issued.

A procedure may accept a variable number of arguments:

```
proc set_numbers(&dest:int[],arg:int...) do  ….
```

From the body of the procedure, the excess arguments are accessed using **arg[]** and **argc**. **argc** returns the number of arguments in positions corresponding to and following that of **arg…** as an **int** value. **arg[]** must be supplied an **int** value in the range **1..argc**, otherwise it will raise an error condition. It returns the value of the corresponding argument**.**

```
for i in 1..argc do
    dest[i] = arg[i]
endfor
```

It is also possible to pass optional arguments *en masse* to a procedure call:

```
proc process(a) do
    print("Value is"//a)
enproc

proc process(a,arg...) do
    process(a)
    process(arg...)
endproc

process_values(1,2,3,4)
```

Arguments passed in this way do form part of the signature of the call – they are used in the matching process and do not necessarily have to correspond to the variable argument portion of the called procedures parameter list.

Expressions

An expression consists of a set of nested procedure calls. The usual infix notation is provided for common mathematical operations, but this is *syntactic sugar* for procedure calls.

Operation	Priority	Equivalent call	Description
`x[a,b]`	1	`proc [] (x,a,b)`	Array subscript
`x{a,b}`	1	`proc {} (x,a,b)`	Array subscript (based on shape)
`x[a,b]=v`	1	`proc []=(x,v,a,b)`	Set array subscript
`x{a,b}=v`	1	`proc {}=(x,v,a,b)`	Set array subscript (based on shape)
`*x`	2	`proc * (x)`	Obtain value
`-x`	2	`proc - (x)`	Minus
`x**y`	3	`proc ** (x,y)`	Power
`x*y`	4	`proc * (x,y)`	Multiplication (including matrix multiplication)
`x/y`	4	`proc / (x,y)`	Division
`x + y`	6	`proc + (x,y)`	Addition
`x - y`	6	`proc - (x,y)`	Subtraction
`x == y`	7	`proc == (x,y)`	Equals
`x /= y`	7	`proc /= (x,y)`	Not equal
`x > y`	7	`proc > (x,y)`	Greater than
`x >= y`	7	`proc >= (x,y)`	Greater than or equal
`x < y`	7	`proc > (y,x)`	Less than
`x <= y`	7	`proc >= (y,x)`	Less than or equal
`x and y`	8	`proc and (x,y)`	Logical and
`x or y`	9	`proc or (x,y)`	Logical or
`x // y`	10	`proc // (x,y)`	String concatenation
`x .. y`	11	`proc .. (x,y)`	Range creation
`x <.. y`	11	`proc <.. (x,y)`	Range creation
`x ..< y`	11	`proc ..< (x,y)`	Range creation
`x <..< y`	11	`proc <..< (x,y)`	Range creation
`x by y`	11	`proc by (x,y)`	Sequence creation
`x => y`	12	`proc => (x,y)`	If x is true then y else null optional value
`x || y`	13	`proc || (x,y)`	If x is not null (or null optional value) then x else y
`x => y || z`	13	`proc||(proc=>(x,y), z)`	If x is true then y else z
`x dim y`	14	`proc dim(x,y)`	Array creation – spread copies of x over domain y

Sub-expressions

Sub-expressions may be separated from the main expression using **where**.

```
s = a * -exp (b/a) where a= c/sqrt(b)
```

There may be multiple values defined after a **where** keyword. These clauses may not refer to each other, but it is possible to follow with a second where statement and associated clauses:

```
a = x **2 / y**3 where x = s/p, y=s/q where s = sqrt(p**2 + q**2)
```

It is also possible to include a check clause after an expression. This raises an error if the associated logical expression does not yield a **true** value:

```
x= my_sqrt_fn(x) check x**2==old_x where old_x=x
```

Check expressions may optionally not be compiled or executed.

Subscripts
Subscript expressions have the following syntax:

'**[**' *sexpr*{ '**,** ' *sexpr* }'**]**' | '**{**' *sexpr* { '**,** ' *sexpr* } '**}**' }
sexpr::=
expr| [*expr*] ('..' | '<..' | '..<' | '<..<') [*expr*] [**by** [*expr*]]

Subscript expressions using square brackets **[]** index an element or slice with reference to the domain of the array value being subscripted. Subscript expressions using braces **{}** index elements or slices with respect to the shape of the domain of the array.

Subscript expressions may contain the keywords **high**, **low**, or **step**. If this occurs in a given index position of the subscript expression list, then these are converted to calls to the equivalent procedures on the corresponding dimensions of the array's domain.

```
x[low..high/2,low..high by step*2 ]
```

is equivalent to:

```
x[low(d1(x))..high(d1(x))/2,low(d1(x))..high(d1(x))/2 by step(d2(x))*2]
```

If the subscript expression consists of a range generator or a direct combination of range and sequence generators (**x..y by z**) then the start, end or step expressions may be omitted, in which case they are set to **low**, **high** or **step** respectfully. Thus the above expression could be written:

```
x [ .. high/2 , .. by step*2 ]
```

Statements
PM provides a range of conventional control statements:

if *xexpr* **then** *statements* { **elseif** *xexpr* **then** *statements* } [**else** *statements*] **endif**
select *xexpr* **case** *xexprlist* **do** *statements* {**case** *xexprlist* **do** *statements* } [**otherwise** *statements*] **endselect**
while *xexpr* **do** *statements* **endwhile**
repeat *statements* **until** *xexpr*

The **if** statement conditionally executes a list of statements:

```
if x > y then
    print("X is greater")
endif

if x>y then
    print("X is greater")
else
    print("Y is greater or equal")
endif
```

The **select** statement tests an expression against lists of values and executes the first statement list to be associated with a value matching the expression. The **select** statement builds sets of values (which do not have to be constants) and uses the set **in** operator to check the applicability of each case.

```
select digit
     case '1','3','5','7','9' do print("Odd")
     case '2','4','6','8' do print("Even")
     otherwise do print("?")
endselect
```

The **repeat** statement sequentially repeats a list of statements until an expression yields a true value (test at the end).

```
repeat
     x = x/2
     y = y+1
until x == 0
```

The **while** statement executes a statement zero or more time while a given expression yields a true value (test at the start)

```
nbits=0
while x>0 do
     x = x/2
     nbits = nbits + 1
endwhile
```

In each of these cases, conditional expressions must return a **bool** value. An error will result if they do not.

Two statements allow for debugging and may optionally not be compiled and executed. The **check** statement confirms that an expression yields a **true** value and raises an error if this is not the case. The **debug** statement executes a block of code only if debugging is enabled.

check *xexpr*
debug *statements* **enddebug**

For example:

```
check value /= null

debug print("Entering main loop")enddebug
```

Parallel execution

The **for** statement executes its body in parallel for every element of one or more domains or arrays:

[**using** *assignlist*] **for** *iter subexp* [**seq** | **conc**] *loopbody* **endfor**
iter::=
name { **,**' *name* } **in** *exp* { **,**'*exp* }
loopbody::=
do *statements* [**build** *assignlist*]
build *assignlist*

For example:

```
for pixel in image do
     pixel = min(pixel,threshold)
endfor
```

It is possible to iterate over more than one domain and/or array, providing they all share the same domain shape:

```
for pixel,pixel2 in image1, image2 do
     if pixel+pixel2>threshold then
          pixel=pixel-threshold/2
          pixel2=pixel2-threshold/2
     endif
endfor
```

Statements in the body of a parallel loop are not normally allowed to modify variables defined outside of the scope of the **for** statement. It is possible to require that the loop body is executed sequentially, in which case these restrictions do not apply.

```
for s in array seq do
     sum = sum + f(s)
endfor
```

Communicating operators

'**@**'*name*
'**@**' *name* '**(**'*arglist*'**)**'
name '**@**' '**{**' *sexpr* { '**,**' *sexpr* } '**}**'
name '::' *name* ['**@**' '**{**' *sexpr* { '**,**' *sexpr* } '**}**']

Numerical models will usually require interaction between loop invocations. PM enables this using the '@' operator. Used as a unary operator (@x) this provides an array view of a given expression across all invocations. Used as a subscripting operator (x@{ndb_desc}) it returns values from a local neighbourhood of the calling invocation. For example:

```
for cell in model_array do
   advection=advection_model(cell)
   advected_cell=cell@{advection}
   cell=model(advected_cell,parameters)
endfor
```

In the latter case, the result will be an array with an optional element type and off-edge values undefined.

```
for pixel in image do
     pixel = median(pixel@{-1..1,-1..1})
endfor
```

The neighbourhood is defined with respect to the common shape of the domains/arrays being iterated over, shifted so that the zero point lies at the location of the point associated with the current invocation.

Reducing '**::**' operators compute a single value from values taken across the loop:

```
for cell in model_grid do
     cell=model(cell,parameters)
     cell=cell/sum::(cell)     ! Normalise
enddo
```

A reduction operation must be defined using a specialised procedure definition:

It is possible to confine a reduction operator to a neighbourhood:

```
local_mean = sum::A@{nbd}/count:: A@{nbd}
```

Communicating procedures

It is possible to define ***communicating procedures*** incorporating '**@**' operators that can only be called inside non-sequential for statements:

```
proc mean_filter%(x;n)=sum::x@{nbd}/count::x@{nbd}
     where nbd=grid(-n..n,-n..n)
proc image_mean::(x)=sum::x/count::x

for pixel in image do
     pixel = mean_filter(pixel;1)
     norm_pixel = pixel / image_mean(pixel)
endfor
```

Communicating procedures that return a different value to each invocation use the '**%**' notation. Procedures that return the same value to every invocation use the '**::**' notation.

Arguments to communicating procedures fall into two categories, depending on whether they are positioned before or after the semicolon in the parameter list. Parameters listed before the semicolon are *channel variable* parameters. When the procedure is called, these parameters must be associated with the name of a variable defined within the **for** statement – not a more general expression. Channel parameters may be used by '**@**' operators and passed as channel arguments to other communicating procedures with regular arguments to a communicating procedure may not. They are additionally associated with synchronisation rules defined below. This restriction does not apply to parameters positioned after the semicolon, which may be associated with general expressions and are not subject to synchronisation rules. It is possible to require a non-channel parameter to be loop-invariant using the **invar** keyword. An **invar** parameter is strictly more specific than a parameter that does not have this constraint. This facility enables special cases where one or more parameters are loop-invariant to employ specific algorithms. Keyword arguments to a communicating procedure must be loop invariant.

There are a number of specialised forms of communicating procedure definition. Reduction procedures combine values though the application of a binary operator:

```
proc sum:: (x) reduce(y) = x + y
```

The binary operation is applied with arbitrary ordering and bracketing. This will only give a deterministic result if the operation is both commutative and associative. A reduction procedure definition must specify the same number of channel parameters as the number of values it returns. Moreover, it must specify the same number of reduction variables (in the **reduce** clause) and the number of values it returns.

A second special form of communicating procedure is the processor-local procedure. Each argument to a **local** procedure is passed as an array over the subdomain being processed on the current processor node (**invar** parameters are passed unmodified.) Local '**%**' procedures must return arrays over the same domain as their arguments. Local '**::**' simply return values, however the results are undefined if the same value is not returned on all processor nodes.

Synchronisation requirements – structured parallelism

The channel variables associated with PM parallel communication operators must be operated on in the same sequence for each invocation of the enclosing parallel statement. The communication sequence along each possible path through the body of a for statement must be the same. This sequence is defined as follows:

1. The sequence associated with a communication operator is defined to be the variable to which the operator is applied (in terms of the object accessed – not just the variable name)
2. The sequence associated with a call to a loop procedure is defined to be the channel variables associated with the call
3. The sequence associated with a sync statement is defined to be the variables listed by the statment
4. The sequence associated with a statement list is defined to be an ordered set of the sequences associated with each statement in the statement list which contains communicating operators
5. For a conditional statement (**if**, **select**) the sequences associated with each branch of the conditional must be the same.
6. For a sequential loop (**while**, **repeat**, **for**..**seq**) the sequence is defined as the sequence of the loop body, grouped as a ***repeated*** sequence. The form of loop is not relevant; all sequential loops apply the same repeated grouping. In terms of sequence matching, a repeated group must match another repeated group containing the same sequence.

In terms of matching sequences, communicating operators or procedures may only be matched with **sync** statements, not with each other. The **sync** statement supplies the required values to the communicating procedure to which it is matched.

If a conditional expression (**if**, **select**) contains communicating operations then each conditional branch of the statement must contain the same sequence of operations.

If communicating operations appear within non-parallel loops (**while**, **repeat**, **for**..**seq**). These loops are not forced to repeat the same number of times for each invocation of the enclosing parallel **for**. If a communicating operation executed in a longer-running loop tries to access a value from an invocation where the loop has already terminated, the operation will be supplied with the value that the named variable possessed immediately prior to the termination of the shorter-running loop. There is an implicit synchronisation at the end of sequential loops – a communicating operation executing after the end of the loop in one parallel invocation cannot match an operation executed from within the loop in a different parallel invocation.

If a sequential loop containing communicating operations occurs in a conditional statement, then a loop containing a matching sequence of communicating operators must occur at the same position (in terms of sequence of communicating operations) on each branch of the conditional statement. Matching loops do not have to employ the same statement forms – a **while** loop can match a **repeat** loop and both can match a **for**.. **seq** loop. Moreover, a nested set of sequential loops in which only the innermost loop body contains communicating operations may be considered to be a single loop when matching parallel communication operations.

Returning values using build

A **for** statement may return one or more loop invariant expressions using the build clause:

```
sum_f := for point in domain do value:= f(point) build sum::val endfor
```

Parallel Search

The **for** .. **do** .. **find** .. **endfor** version of the **for** statement executes arbitrary invocations of the loop body until at least one such body is associated with a true value for the search criterion of until all invocations have been completed.

[*lhs* { '**,**' *lhs* } ('='|'**:=**')] [**using** *keyargs*]
[**for** *iter*] [**do** *statements*]
find *exprlist* **when** *exprlist* [**otherwise** *exprlist*] **endfor**

For example:

```
high_pixel:=for pixel in image
find pixel when pixel>threshold
otherwise -1
endfor
```

The expression(s) following **find** are computed for an arbitrary loop invocation for which the **when** expression evaluates to **true**. The expressions following the **otherwise** keyword are returned if no loop invocation yield a true value for the **when** expression. The body of a **for** .. **find** loop may not use communicating operators, except within nested parallel **for** statements.

Task parallelism

Task parallelism is supported by the **do** .. **enddo** construct.

[**using** *keyargs*] [**with** *statements*] **do** *statements* { **also do** *statements* } **enddo**

The semantics of a **do** statement are defined according to an equivalent **for** statement.

using *opts* **with** $statements_0$ **do** $statements_1$ **also do** $statements_2$ … **also do** $statements_N$ **enddo**

is semantically equivalent to:

using *opts* **for** *_index* **in 1..***N* **do**
$statements_0$
select *_index*
case 1 do $statements_1$
case 2 do $statements_2$
…
case *N* **do** $statements_N$
endselect
endfor

Tasks and processors

PM programs are assumed to be executed on an N-dimensional array of ***processors***. Processors are capable of efficiently executing a set of ***tasks***. The PM specification does not define how the abstract concept of a processor into hardware: a processor can relate to anything from a single core to a sub-cluster. The primary requirement is that a processor is capable of keeping its own hardware busy running available tasks which effectively requires efficient mechanisms for task migration within the processor. Task migration between processors, however, is assumed to either introduce a degree of overhead which the programmer has to take into account, or not to be possible at all in any acceptably efficient manner.

PM assumes that the number of processors in the processor grid is fixed by the implementation. When a PM program proceeds, this processor set is divided among available processors according to the following:

On start-up, the PM program is assumed to be running a single task. This task owns all processors in the grid.

When a parallel statement (**for** or **do**.. **also do**) is encountered. Then processor allocation is governed by comparing its iteration space (which is implicit in **do**..**also do**) to the processor grid associated with the currently running task

1. If the number of elements in the processor grid is greater than the number of elements in the iteration space, then the processor grid is partitioned over the iteration space using the ***default processor partition function***. A new task is created to process each element in the iteration space. This task owns the processors listed in the element of the processor grid partition corresponding to the given element in the iteration space.

2. If the number of elements in the processor grid is less than or equal to the number of elements in the iteration space, then a distribution of the iteration space over the processor grid is calculated using the ***default iteration space distribution function***. The distribution is conceptually a nested array **D[p][i]** of subdomains of the iteration space, indexed by both an element **p** of the processor grid and a second, arbitrary, index **i**. On each processor, a new task is created for each element **D[p][i]** of **D[p]**. Each task is responsible for processing the iteration space elements contained in **D[p][i]** and shares ownership of the single-element processor grid **p.** The default distribution function applies a block cyclic distribution for matrix domains and a nested block distribution for grid domains.

The PM definition does not specify how a task owning multiple processors utilises those processors to perform its computations. Two possible implementation approaches are:

1. Every processor in the processor set owned by a given task redundantly executes the same code.

2. A single processor in the processor set executes the code, the remaining processors are held in a standby state until needed by nested parallel operations.

There is clearly a trade-off between ease of implementation, inter-processor message overhead and resource utilisation between these two approaches. A PM implementation is not required to employ either of these extremes. The only requirement, as with defining processors, is that controlling implementation overheads should not require intervention from the programmer.

The above process may be modified by providing keyword arguments in the using clause of a **for** or **do** statement. The following options are defined.

distr=*Distribution*	Use the given distribution in place of the output of the default distribution procedure
proc_distr=*Distribution*	*Use the given distribution in place of the output of the default processor distribution procedure*
work=*Array*	*If this option is specified, the default processor distribution will unevenly distribute processors according to a measure of the relative amount of work associated with each invocation. The given array should have numeric elements and a domain shape equal to the common shape of the inputs to the parallel statement*
float=*BoolValue*	*If this option is true, then the parallel statement should allow tasks to move between processors, assuming the PM implementation supports this option. The default is false.*

block=*Domain* *This option sets the block size used by the default distribution procedure.*

It is also possible to specify that a **for** statement is concurrent only:

for … **conc do** …. **endfor**

This form of the **for** statement does not partition its invocations among available processors. Moreover, it set the number of available processors within the statement body to one – thus all dynamically nested for statements are forced to be concurrent (if not specifically sequential).

Syntax

module **::=** *program_module* | *library_module*

program_module::=

[*decls*] *statements*

library_module::=

decls [**debug** *statements* **enddebug**]

decls::=

{ *import* '**;**' } (*import*| *decl*) { '**;**'*decl* }

import::=

use *modname* [*modifiers*] { '**,**' *modname* [*modifiers*] }

modname::=

name { '**.**' *name* }

modifiers::=

'**{**' *modifier* { '**,**' *modifier* } '**}**'

modifier::=

[**type** | **param** | **proc**] *name* '**=>**' *name*

decl::=

procdecl | *typedecl* | *paramdecl* | *defaultdecl*

signature::=

procsig | *typesig* | *paramsig*

procdecl::=

proc *procname params* [*special*] '**=**' *xexprlist* [**do** *statements* **endproc**]

proc *procname params* [*special*] [**check** *exprlist* [*whereclause*]] **do** *statements* [**result** '**=**'*xexprlist*] **endproc**

special::=

local | **reduce** '**(**' *namelist* '**)**'

procname::=

name | *op* | *name* '**::**' | **low** | **high** | **step** | **null** | **opt**

op::=

'**+**' | '**-**' |'*****' | '**/**' | '******'| '**==**' | '**/=**' | '**>**'| '**>=**' | **and** | **or** | '**//**' | '**=>**' | '**||**' | **dim**| '**=**'| '**[]**' | '**{}**'|

'**..**' | '**<..**' | '**..<**' | '**<..<**' | **by**

params::=

'**(**' [cpars '**;**'] { *param* '**,**' } [*param* ['**,**' *keyparams*] | **arg** '**...**' ['**,**' *keyparams*] | *keyparams*] '**)**'

cpars::=

name['**:**' *type*] { '**,**' *name*['**:**' *type*] }

param::=

['**&**'] *name*['**:**' [**invar**] *type*]

keyparams::=

{ *name* '**=**' *expr*'**,**'} (*name* '**=**' *expr* |**key** '**...**')

namelist::=

name { '**,**' *name* }

typedecl::=

type *name* ['**{**' *typeparams* '**}**'][**in** *namelist*] (**is** *typelist* | **includes** *typelist* | **also includes** *typelist*)

typeparams::=

name '**:**' *type* { '**,**' *name* '**:**' *type* }

paramdecl::=

param *name* '**=**' *xexpr*

defaultdecl::=

default *name* '**=**' *xexpr*

typelist::=

type { '**,**' *type* }

opttypelist::=

[*type*] { '**,**' [*type*] }

type::=

name ['**{**' *opttypelist* '**}**']

[*type*] '**[**' *opttypelist* '**]**'

'**<**' *type* '**>**'
'**<**' **opt** *type* '**>**'
[**struct**| **rec**] [*name*] '**{**'*name* ['**:**' *type*] { '**,**' *name* ['**:**' *type*] } '**}**'

statements::=
statement { '**;**' *statement* } ['**;**']

statement::=
if *xexpr* **then** *statements*{ **elseif** *xexpr* **then** *statements* } [**else** *statements*] **endif**
select *xexpr* { **case** *xexprlist* **do** *statements* } [**otherwise** *statements*] **endselect**
while *xexpr* **do** *statements* **endwhile**
repeat *statements* **until** *xexpr*
[*lhs* { '**,**' *lhs* } ('**=**'|'**:=**')] [**using** *keyargs*] **for** *iter subexp* [**seq** | **conc**] *loopbody* **endfor**
[**using** *keyargs*] [**with** *definition* {'**;**' *definition* }] **do** *statements* { **also do** *statements* } **enddo**
sync *namelist*
check *expr* [*whereclause*]
debug *statements* **enddebug**
const *assignlist*
assignment
call
iter::=
name { '**,**' *name* } **in** *exprlist*
loopbody::=
build *assignlist*
do *statements* [**build** *assignlist*]
[**do** *statements*] **find** *xexprlist* **when** *xexprlist* [**otherwise** *xexprlist*]
definition::=
(*name*|'**_**') { '**,**' (*name*|'**_**') } '**:=**'*xexpr*
assignlist::=
assignment { ',' *assignment* } *subexp*
assignment ::=
lhs '**=**' *expr*
lhs'**,**' *lhs* { '**,**' *lhs* } '**=**' *call*
lhs::=
name qualifier |'**_**'
expr::=
<u>*lowest to highest precedence*</u>
expr **dim** *expr*
expr '**||**' *expr*
expr '**=>**' *expr*
expr ['**..**' | '**<..**' | '**..<**' | '**<..<**'| **by**] *expr*
expr '**//**' *expr*
expr **or** *expr*
expr **and** *expr*
not *expr*
expr ['**==**' | '**/=**' | '**>**'| '**<**' | '**>=**' | '**<=**' | **in**] *expr*
expr ['**+**' | '**-**'] *expr*
expr ['*****' | '**/**'] *expr*
expr '******' *expr*
['**-**' | '*****'] *term*
term qualifier
subexp::=
[**check** *exprlist*] *whereclause*
whereclause::=
{ **where** *namelist* '='*expr* { '**,**' *namelist* '='*expr* } }

xexpr::=
    expr subexp
xexprlist::=
    exprlist subexp
exprlist ::=
    expr { '**,**' *expr* }
call::=
    [*name* | *name* '**%**' | name '**::**'| **low** | **high** | **step** | **opt** | **null**] '**(**' *arglist* '**)**'
    proc *procname* '**(**'*arglist*'**)**'
term::=
    subterm
    value
    '**@**'*name*
    name '**@**' '**(**' *arglist*'**)**'
    name '**@**' '**{**' *sexpr* { '**,**' *sexpr* } '**}**'
    name '::' *name* ['**@**' '**{**' *sexpr* { '**,**' *sexpr* } '**}**']
subterm::=
    '**<**' [**opt**] [*type*] *'>' subterm*
    name
    '**(**'*exprlist* '**)**'
    call
    array
    [**struct** | **rec**] [*name*] '**{**' *name* '**=**'*exp* { '**;**' *name* '**=**'*exp* } '**}**'
value::=
    constant
    arg '**[**'*expr*'**]**' | **argc** | **low** | **high** |**true** | **false** | **null**
    proc *procname* | **type** *type*
constant::=
    number | *string* |'**$**' *name*
arglist ::=
    [*namelist ';'*] { *arg* '**,**' } (*arg* | **arg** '**...**')['**,**' *keyargs*]
    [*keyargs*]
arg::=
    '**&**'*name qualifier* | *expr*
keyargs::=
    name '**=**' *expr* { '**,**' *name* '**=**' *expr* }
qualifier::=
    { '**.**'*name*| '**[**' *sexpr* { '**,** ' *sexpr* } '**]**' | '**{**' *sexpr* { '**,** ' *sexpr* } '**}**' }
sexpr::=
    expr| [*expr*] ('**..**' | '**<..**' | '**..<**' | '**<..<**') [*expr*] [**by** [*expr*]]
array::=
    '**{**' (*list2d* | *generator*) '**}**'
    '**[**' (*list2d* | *generator*) '**]**'
list2d::=
    exprlist {'**;**' *exprlist*} ['**;**']
generator::=
    expr '**:**' *iter*

Intrinsic procedures

FORTHCOMING