The PM Programming Language

IMPLEMENTING NUMERICAL MODELS ON DISTRIBUTED HARDWARE

Tim Bellerby
School of Environmental Sciences
University of Hull, UK

HULL VIPER HPC

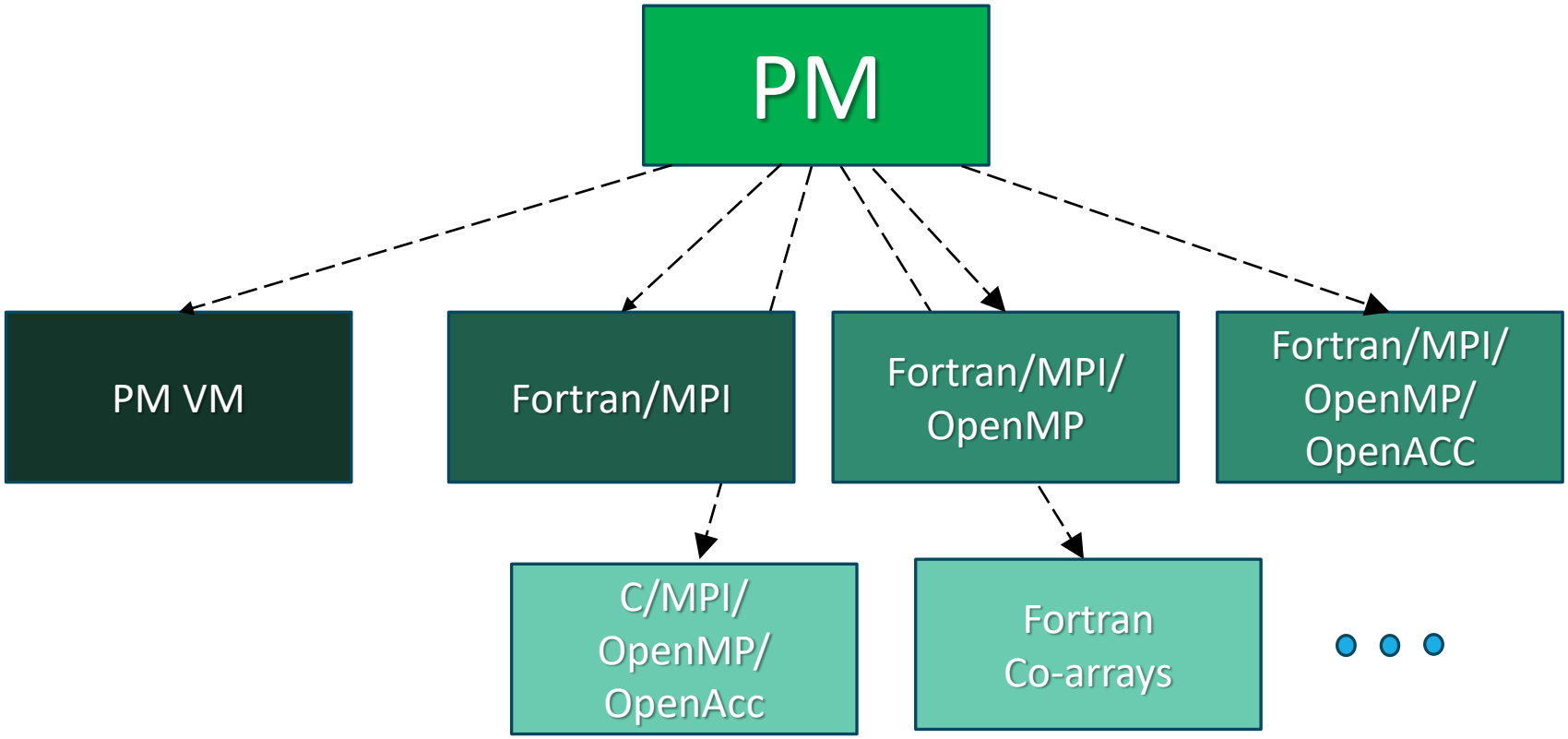
Research/HPC Software Divide

Python / Matlab / …

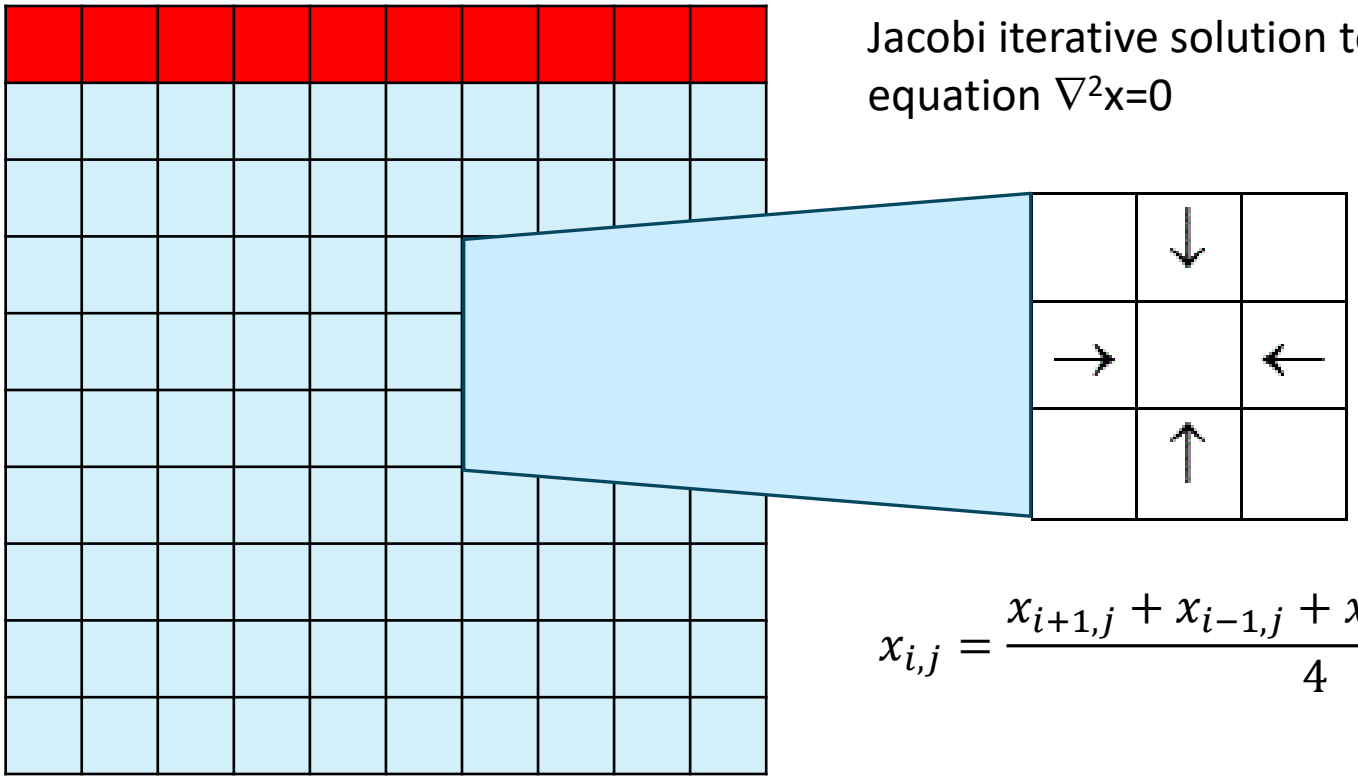
FORTRAN/C/C++
+ MPI + OpenMP
+ OpenAcc/CUDA/…

PM Programming Language

New programming language and programming language implementation designed for numerical modelling on distributed systems



An example model

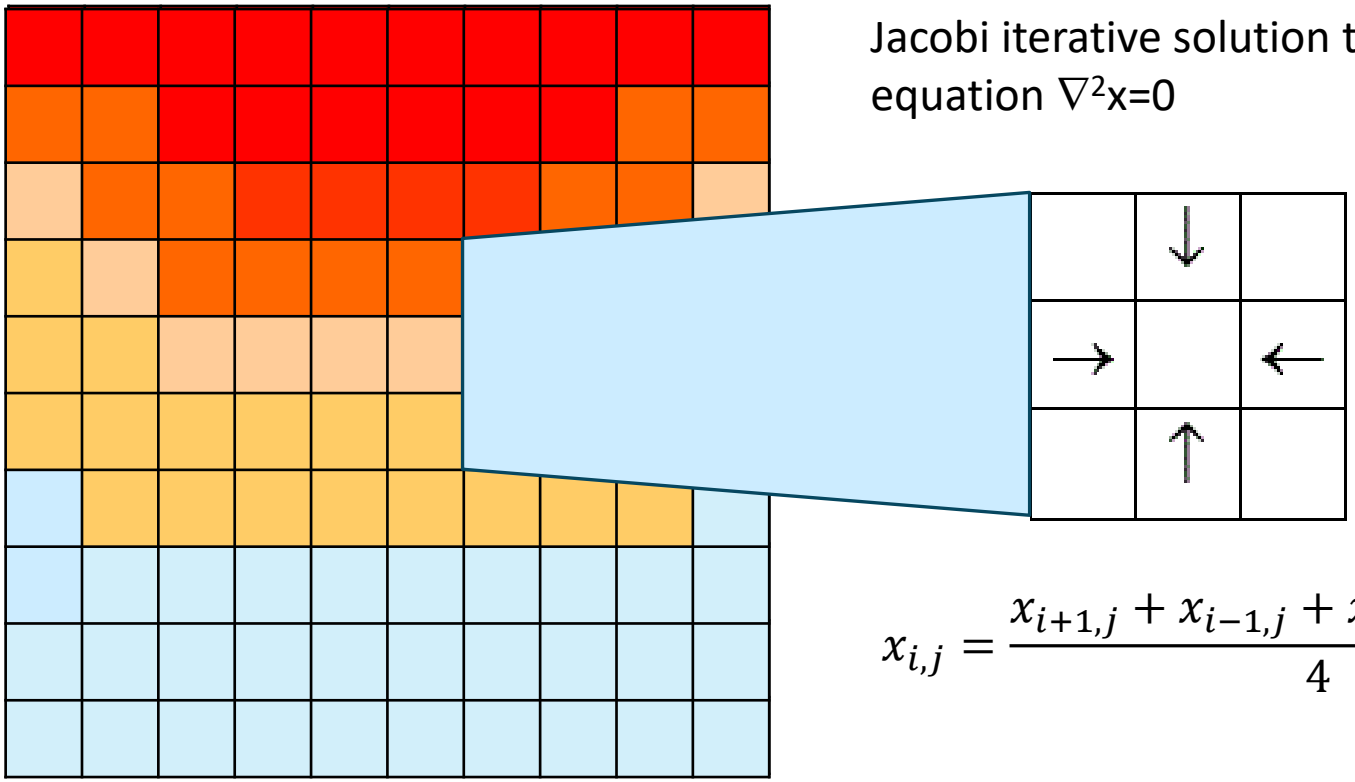


Jacobi iterative solution to 2D heat equation $\nabla^2 x=0$

$$x_{i,j} = \frac{x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1}}{4}$$

Repeat until no further change

An example model

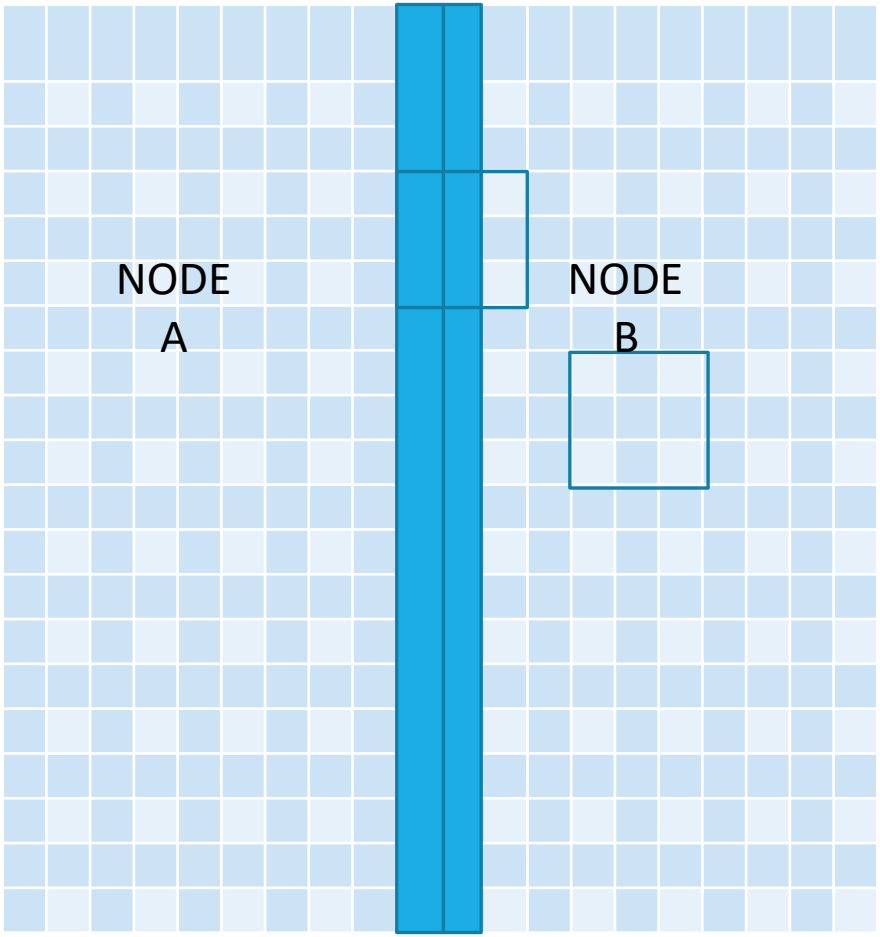


Jacobi iterative solution to 2D heat equation $\nabla^2 x=0$

$$x_{i,j} = \frac{x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1}}{4}$$

Repeat until no further change

Distribution



Coding models in PM

```
var out_array=darray(0.0,[0..MODEL_COLS+1,0..MODEL_ROWS+1])
for x in out_array {
   over [0,]: x=1.0
   until invar totdiff<TOL {
     var diff=0.0
     nhd [-1..1,-1..1] dx of x bounds EXCLUDED {
      cell=(dx[-1,0]+dx[1,0]+dx[0,-1]+dx[0,1])/4.0
      diff=(x-dx[0,0])**2
     }
     totdiff=sqrt(sum%(diff)/size(out_array))
   }
}
```

Coding models in PM

```
var out_array=darray(0.0,[0..MODEL_COLS+1,0..MODEL_ROWS+1])
for x in out_array {
    over [0,]: x=1.0
    until invar totdiff<TOL
      var diff=0.0
      nhd [-1..1,-1..1] dx
       cell=(dx[-1,0]+dx[1,0]+dx[0,-1]+dx[0,1])/4.0
       diff=(x-dx[0,0])**2
      }
      totdiff=sqrt(sum%(diff)/size(out_array))
    }
}
```

Extensive, optional, compile-time type inference

Coding models in PM

```
var out_array=darray(0.0,[0..MODEL_COLS+1,0..MODEL_ROWS+1])
for x in out_array {
  over [0,]: x=1.0
  until invar totdiff<TOL {
    [] dx of x bounds EXCLUDED {
      dx=(dx[1,0]+dx[0,-1]+dx[0,1])/4.0
      diff=(x-dx[0,0])**2
    }
    totdiff=sqrt(sum%(diff)/size(out_array))
  }
}
```

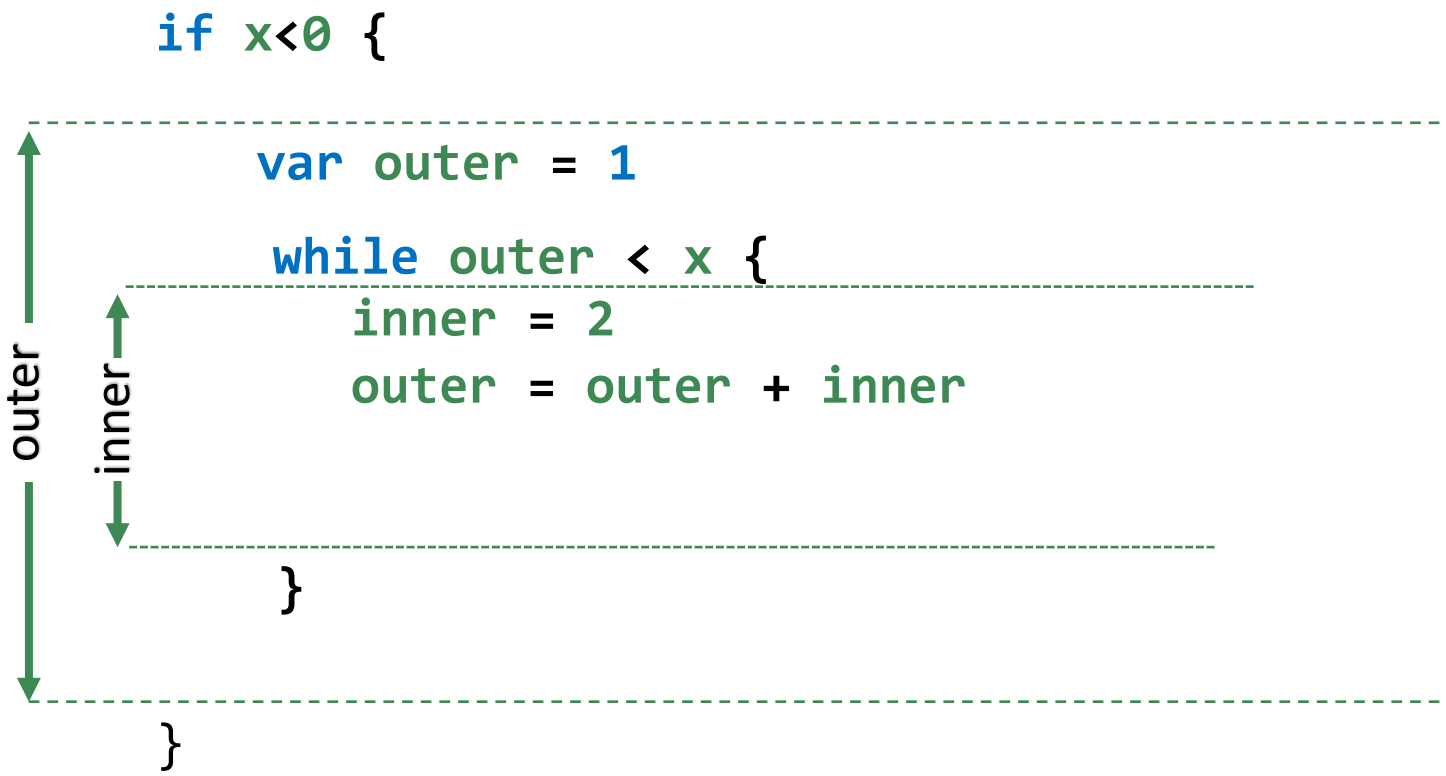
Explicitly parallel statements

Coding models in PM

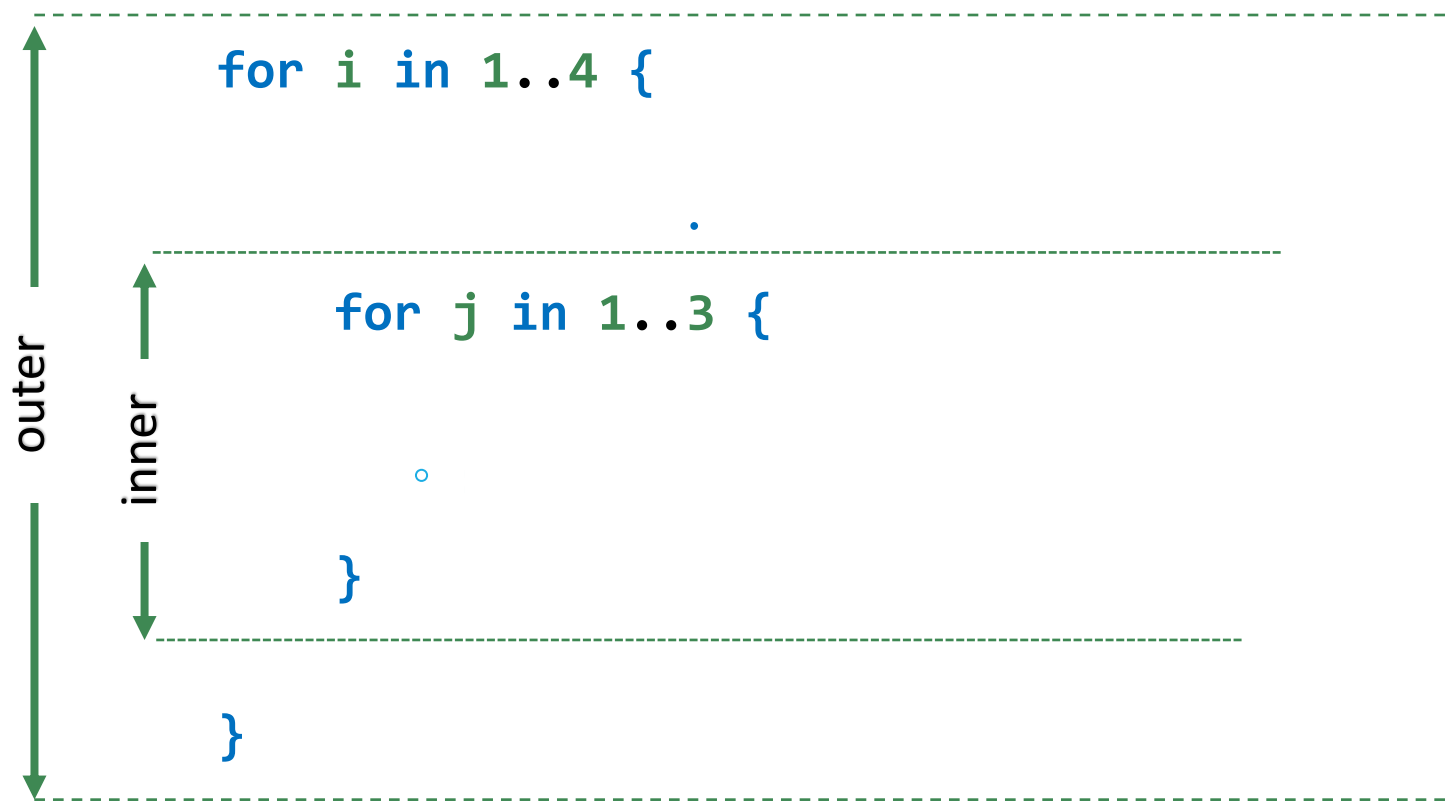
```
var out_array=darray(0.0,[0..MODEL_COLS+1,0..MODEL_ROWS+1])
for x in out_array {
   over [0,]: x=1.0
   until invar totdiff<TOL {
     var diff=0.0
     nhd [-1..1,-1..1] dx of x bounds EXCLUDED {
      cell=(dx[-1,0]+dx[1,0]+dx[0,-1]+dx[0,1])/4.0
      diff=(x-dx[0,0])**2
     }
     totdiff=sqrt(sum%(diff)/size(out_array))
   }
}
```

Operations that communicate between parallel strands

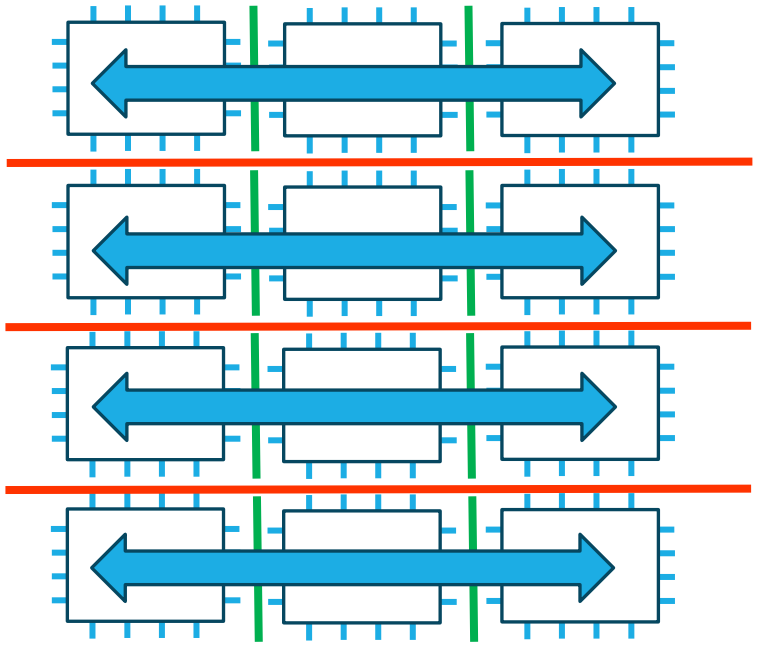
Scope


if x<0 {
var outer = 1
while outer < x {
inner = 2
outer = outer + inner
}
}
outer
inner

Parallel Scope

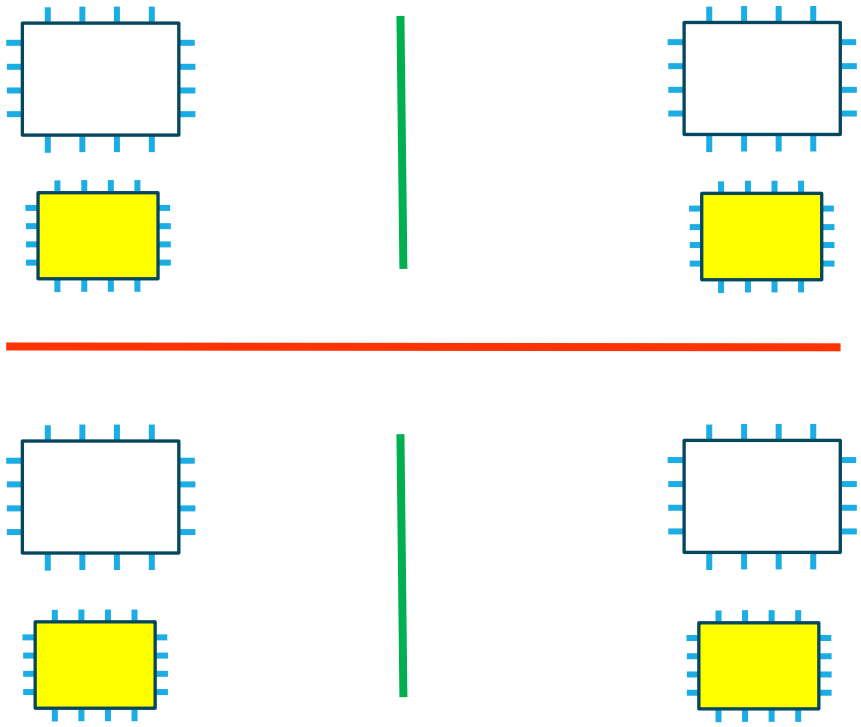


Parallel Scope



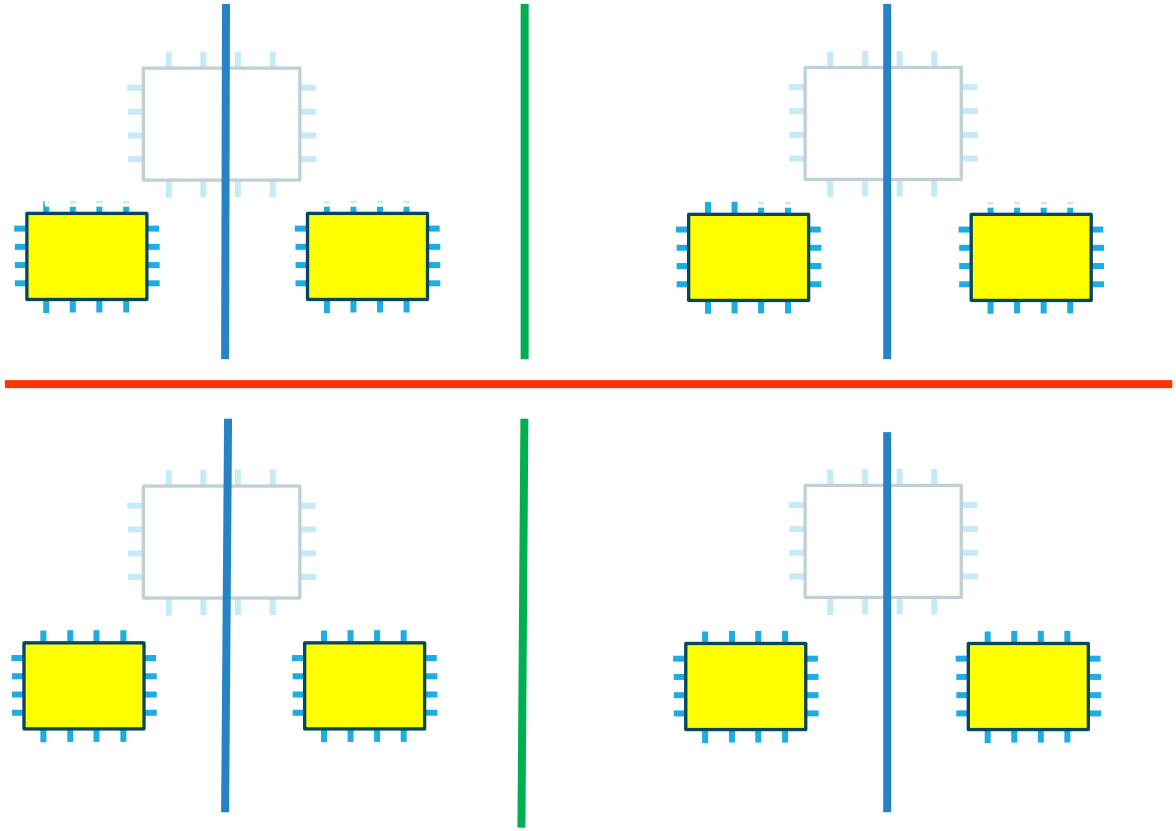
```
for i in 1..4 {

    for j in 1..3 {

    }

}
```

Adding accelerators*

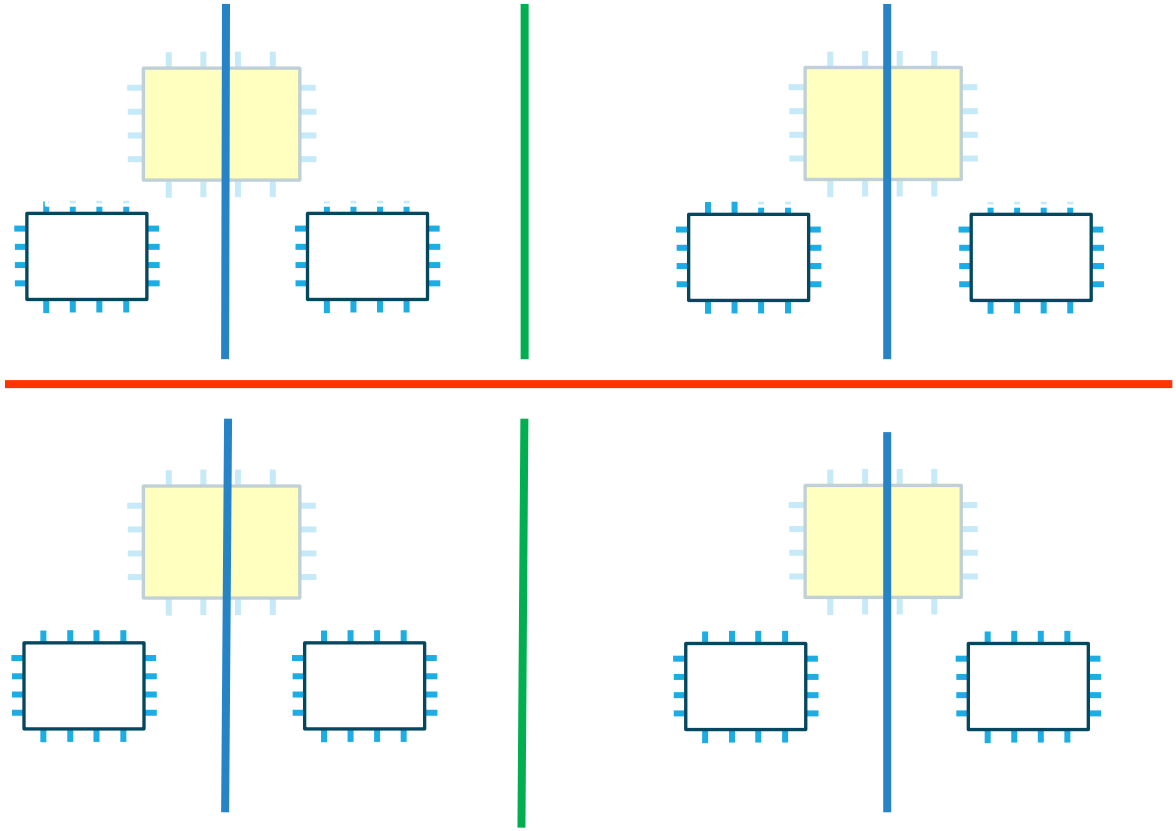


* Planned

Adding accelerators



Adding accelerators



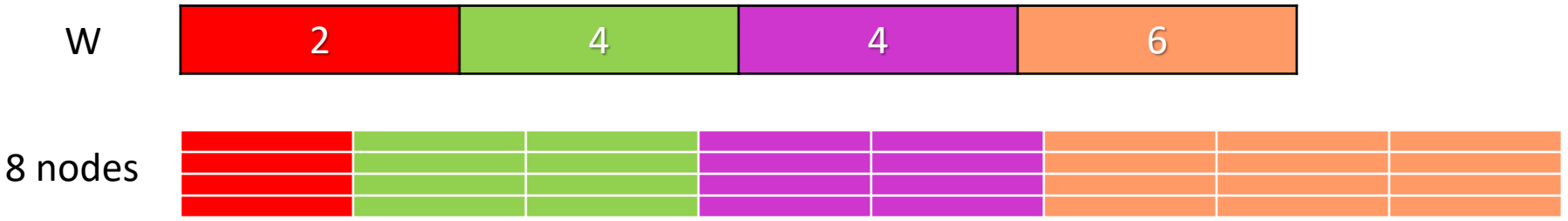
Assigning nodes

```
for i in 1..4 {
    for j = 1..8 {
        ...
    }
}
```

8 nodes

Assigning nodes

```
W={2,4,4,6}
for i in 1..4 <<work=W>> {
    for j in 1..W[i]*2 {
        ...
    }
}
```



Assigning nodes

```
proc process_node(node) {
   …  // Check for and process leaves
   par {
   task LEFT <<work=node.left.nchildren>>:
        process_node(node.left)
   task RIGHT <<work=node.right.nchildren>>:
        process_node(node.right)
   }
}
```

Distributions

10 array elements on 3 processors

Variable Block

Fixed Block (block=4)

Cyclic

Block cyclic (block=2)

Distributions

2D block cyclic execution with 2x2 block size over 7x8 grid

1,1	2,1	3,1	4,1	5,1	6,1	7,1
1,2	2,2	3,2	4,2	5,2	6,2	7,2
1,3	2,3	3,3`	4,3	5,3	6,3	7,3
1,4	2,4	3,4	4,4	5,4	6,4	7,4
1,5	2,5	3,5	4,5	5,5	6,5	7,5
1,6	2,6	3,6	4,6	5,6	6,6	7,6
1,7	2,7	3,7	4,7	5,7	6,7	7,7
1,8	2,8	3,8	4,8	5,8	6,8	7,8

```
for j in [1..7,1..8] <<dist = BLOCK_CYCLIC(2)>> {
    …
}
```

Distributions

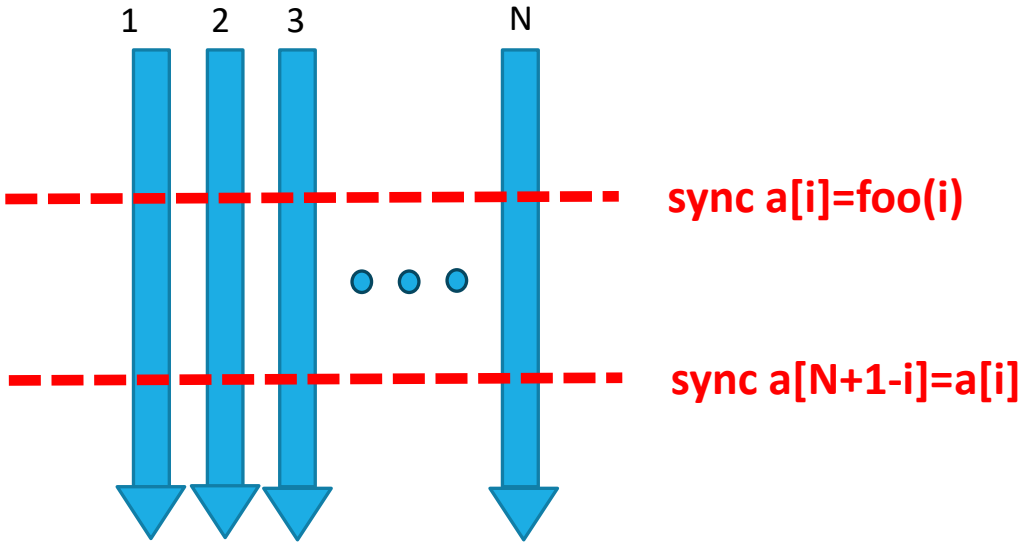
2D block cyclic array with block size of two on first dimension only over 7x8 grid

1,1	2,1	3,1	4,1	5,1	6,1	7,1
1,2	2,2	3,2	4,2	5,2	6,2	7,2
1,3	2,3	3,3`	4,3	5,3	6,3	7,3
1,4	2,4	3,4	4,4	5,4	6,4	7,4
1,5	2,5	3,5	4,5	5,5	6,5	7,5
1,6	2,6	3,6	4,6	5,6	6,6	7,6
1,7	2,7	3,7	4,7	5,7	6,7	7,7
1,8	2,8	3,8	4,8	5,8	6,8	7,8

```
A = array (0.0,[1..7,1..8],[BLOCK_CYCLIC(2),])
```

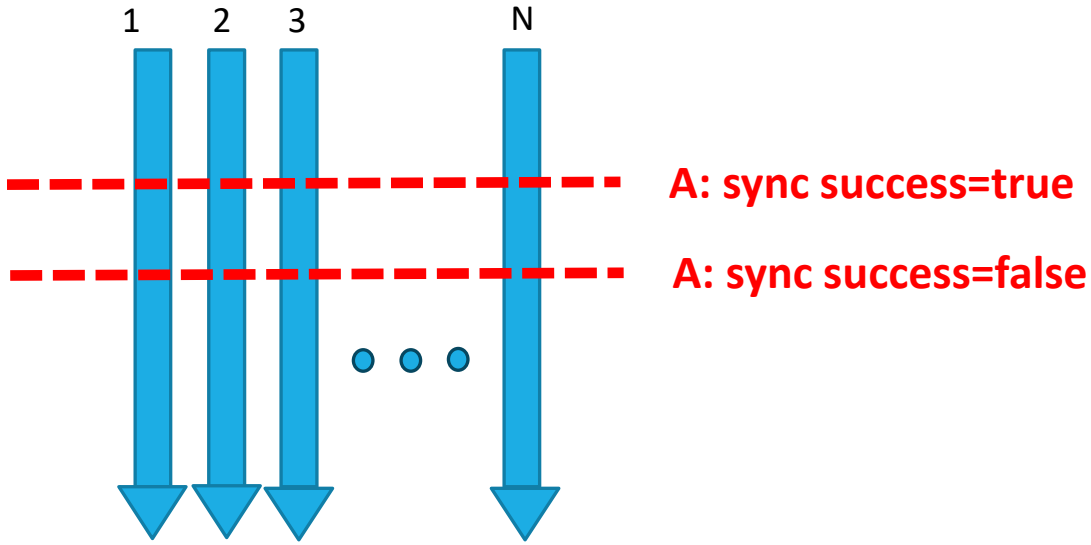
Synchronisation

```
var a = array(0.0,[1..N])
for i in 1..N {
    sync a[i]=foo(i)
     …
    sync a[N+1-i]=a[i]
     …
}
```



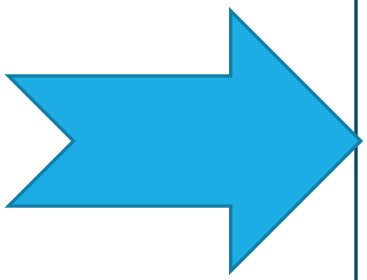
Synchronisation

```
var success = false
for i in 1..N {
   if criterion_met(i) {
        A:  sync success=true
   } elseif blocking_criterion_met(i) {
        A:  sync success=false
    }
}
```



Implementation

```
for ... {
    if E1 {
        S1
        L: sync a= ...
        S2
    } else {
        S3
        L: sync b = ...
        S4
    }
}
```



```
DIMENSION ...
!OMP PARALLEL
DO ...
    IF E1 THEN
        S1
    ELSE
        S2
    ENDIF
 ENDDO
!OMP END PARALLEL
 A= -> MPI
 B= -> MPI
!OMP PARALLEL
 DO ...
     IF E1 THEN
         S3
     ELSE
         S4
     ENDIF
 ENDDO
!OMP END PARALLEL
```

Scheduling

Reorder instructions to:

- Minimise loop start/stops
- Minimise storage requirement
- Interleave computation and communication
- Merge synchronisation points

iSend A
iRecv B
Wait
Process B
iSend C
iRecv D
Wait
Process D
Process E

iRecv B
iRecv D
iSend A
iSend C
Process E
Wait
Process B
Process D

PM Version 0.4

Formal language definition released under Creative Commons Attribution 4.0 International License.

Language Implementation:

- PM to FORTRAN/MPI compiler
- Vector-virtual machine (intended for development/debugging)
- Both available under MIT Licence

PM Version 0.5

A small number of language additions

- Closures
- Sparse Arrays
- Interoperability with C/FORTRAN

Added backends

- FORTRAN/MPI/OpenMP
- FORTRAN/MPI/OpenMP/OpenAcc

Planned summer 2024.

Thank you for your attention

Questions? T.J.Bellerby@hull.ac.uk

Follow progress: www.pm-lang.org

𝕏 @pmlanguage