

WRF Software Architecture

John Michalakes, Head WRF Software Architecture

Michael Duda

Dave Gill

Outline

- Introduction
- Computing Overview
- WRF Software Overview

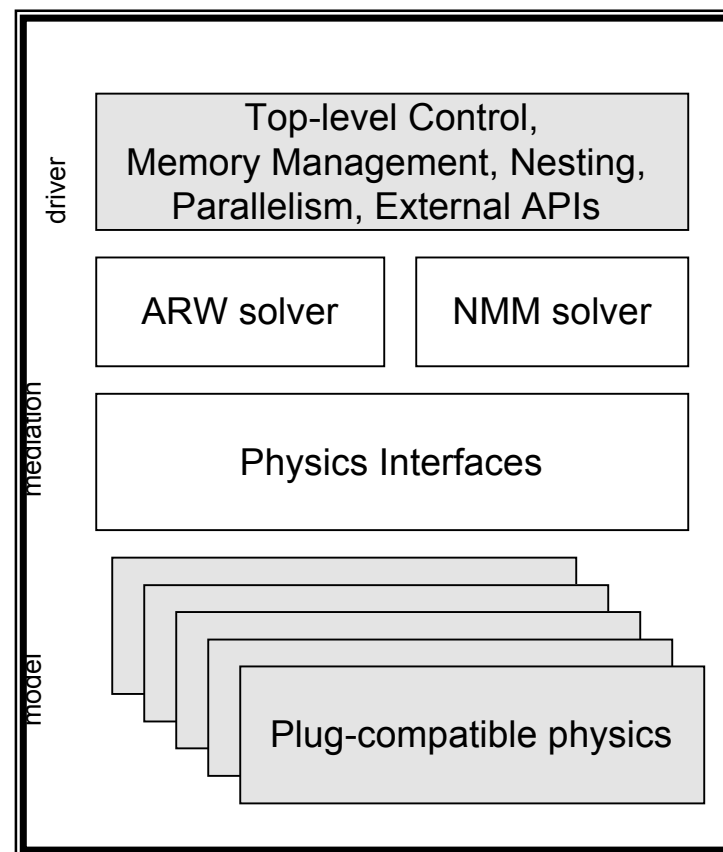
Introduction – WRF Software Characteristics

- Developed from scratch beginning around 1998, primarily Fortran and C
- Requirements emphasize flexibility over a range of platforms, applications, users, performance
- WRF develops rapidly. First released Dec 2000; current release WRF 3.0 (April 2008); next release WRF v3.1 (March 2009)
- Supported by flexible efficient architecture and implementation called the WRF Software Framework

Introduction - WRF Software Framework Overview

- Implementation of WRF Architecture
 - Hierarchical organization
 - Multiple dynamical cores
 - Plug compatible physics
 - Abstract interfaces (APIs) to external packages
 - Performance-portable
- Designed from beginning to be adaptable to today's computing environment for NWP

<http://box.mmm.ucar.edu/wrf/WG2/bench/>

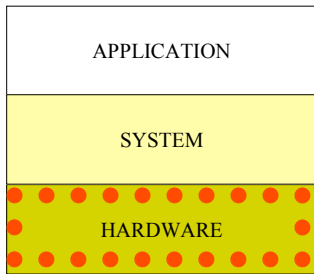


Introduction - WRF Supported Platforms

Vendor	Hardware	OS	Compiler
Apple	G4/G5 + Intel	MacOS	IBM, g95, PGI, Intel
Cray Inc.	X1, X1e	UNICOS	Cray
	Opteron	Linux	PGI, PathScale
HP/Compaq	Alpha	Tru64	Compaq
	Itanium-2	Linux	Intel
		HPUX	HP
IBM	Power-3/4/5/6	AIX	IBM
SGI	Opteron, Itanium-2	Linux	Intel
	MIPS	IRIX	SGI
Sun	UltraSPARC	Solaris	Sun
	Xeon and Athlon	Linux	PGI, Intel, g95, Pathscale
	Itanium-2 and Opteron		
ENIAC	Bunch of tubes	Mostly people	Not invented

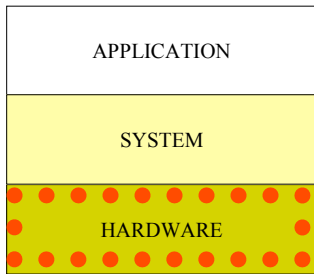
Outline

- Introduction
- Computing Overview
- WRF Software Overview



Hardware: The Computer

- The 'N' in NWP
- Components
 - Processor
 - A program counter
 - Arithmetic unit(s)
 - Some scratch space (registers)
 - Circuitry to store/retrieve from memory device
 - Cache
 - Memory
 - Secondary storage
 - Peripherals
- The implementation has been continually refined, but the basic idea hasn't changed much



Hardware has not changed much...

A computer in 1960

IBM 7090



6-way superscalar

36-bit floating point precision

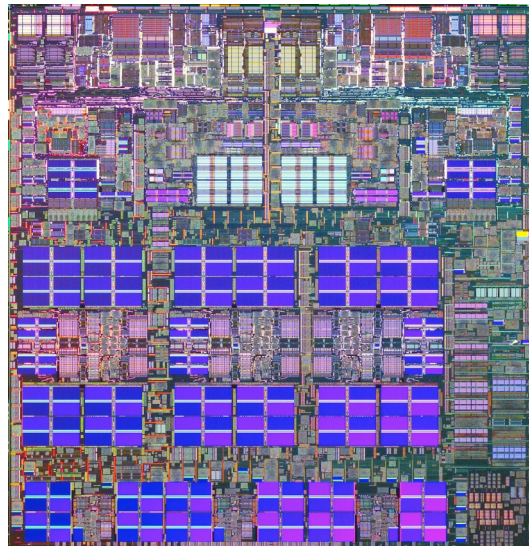
~144 Kbytes

~50,000 flop/s

48hr 12km WRF CONUS in 600 years

A computer in 2008

IBM P6



Dual core, 4.7 GHz chip

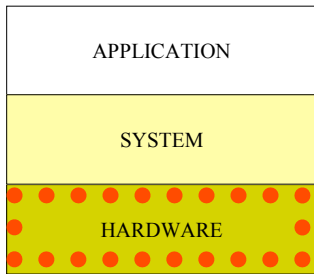
64-bit floating point precision

1.9 MB L2, 36 MB L3

Upto 16 GB per processor

~5,000,000,000 flop/s

48 12km WRF CONUS in 52 Hours



...how we use it has

- Fundamentally, processors haven't changed much since 1960
- Quantitatively, they haven't improved nearly enough
 - 100,000x increase in peak speed
 - 100,000x increase in memory size
 - These are too slow and too small for even a moderately large NWP run today
- We make up the difference with parallelism
 - Ganging multiple processors together to achieve 10^{11-12} flop/second
 - Aggregate available memories of 10^{11-12} bytes

*$\sim 1,000,000,000,000$ flop/s ~ 250 procs
48-h, 12-km WRF CONUS in under 15 minutes*

Examples

- If the machine consists of 4 nodes, each with 4 processors, how many different ways can you run a job to use all 16 processors?

- 4 MPI processes, each with 4 threads

```
setenv OMP_NUM_THREADS 4  
mpirun -np 4 wrf.exe
```

1 MPI

4 threads

1 MPI

4 threads

- 8 MPI processes, each with 2 threads

```
setenv OMP_NUM_THREADS 2  
mpirun -np 8 wrf.exe
```

1 MPI

4 threads

1 MPI

4 threads

- 16 MPI processes, each with 1 thread

```
setenv OMP_NUM_THREADS 1  
mpirun -np 16 wrf.exe
```

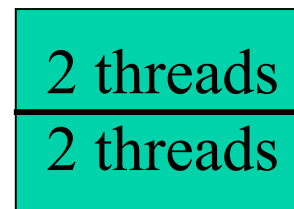
Examples

- If the machine consists of 4 nodes, each with 4 processors, how many different ways can you run a job to use all 16 processors?

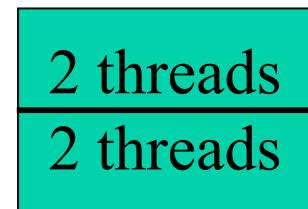
- 4 MPI processes, each with 4 threads

```
setenv OMP_NUM_THREADS 4  
mpirun -np 4 wrf.exe
```

2 MPI



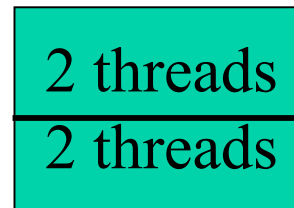
2 MPI



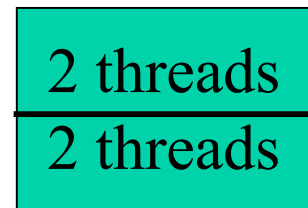
- 8 MPI processes, each with 2 threads

```
setenv OMP_NUM_THREADS 2  
mpirun -np 8 wrf.exe
```

2 MPI



2 MPI



- 16 MPI processes, each with 1 thread

```
setenv OMP_NUM_THREADS 1  
mpirun -np 16 wrf.exe
```

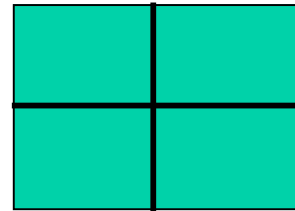
Examples

- If the machine consists of 4 nodes, each with 4 processors, how many different ways can you run a job to use all 16 processors?

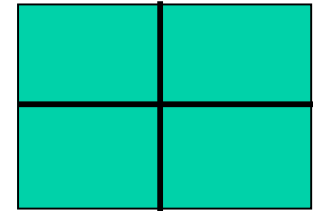
- 4 MPI processes, each with 4 threads

```
setenv OMP_NUM_THREADS 4  
mpirun -np 4 wrf.exe
```

4 MPI



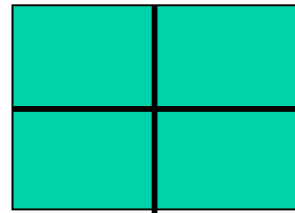
4 MPI



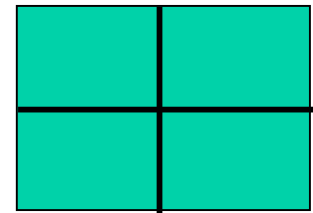
- 8 MPI processes, each with 2 threads

```
setenv OMP_NUM_THREADS 2  
mpirun -np 8 wrf.exe
```

4 MPI



4 MPI



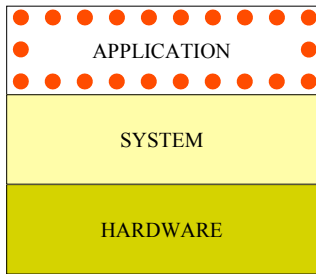
- 16 MPI processes, each with 1 thread

```
setenv OMP_NUM_THREADS 1  
mpirun -np 16 wrf.exe
```

Examples (cont.)

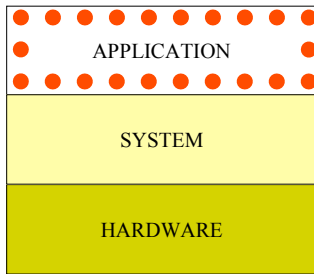
- Note, since there are 4 nodes, we can never have fewer than 4 MPI processes because nodes do not share memory
- What happens on this same machine for the following?

```
setenv OMP_NUM_THREADS 8  
mpirun -np 32
```



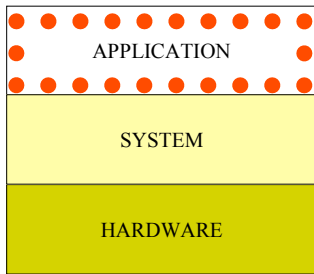
Application: WRF

- WRF can be run serially or as a parallel job
- WRF uses *domain decomposition* to divide total amount of work over parallel processes



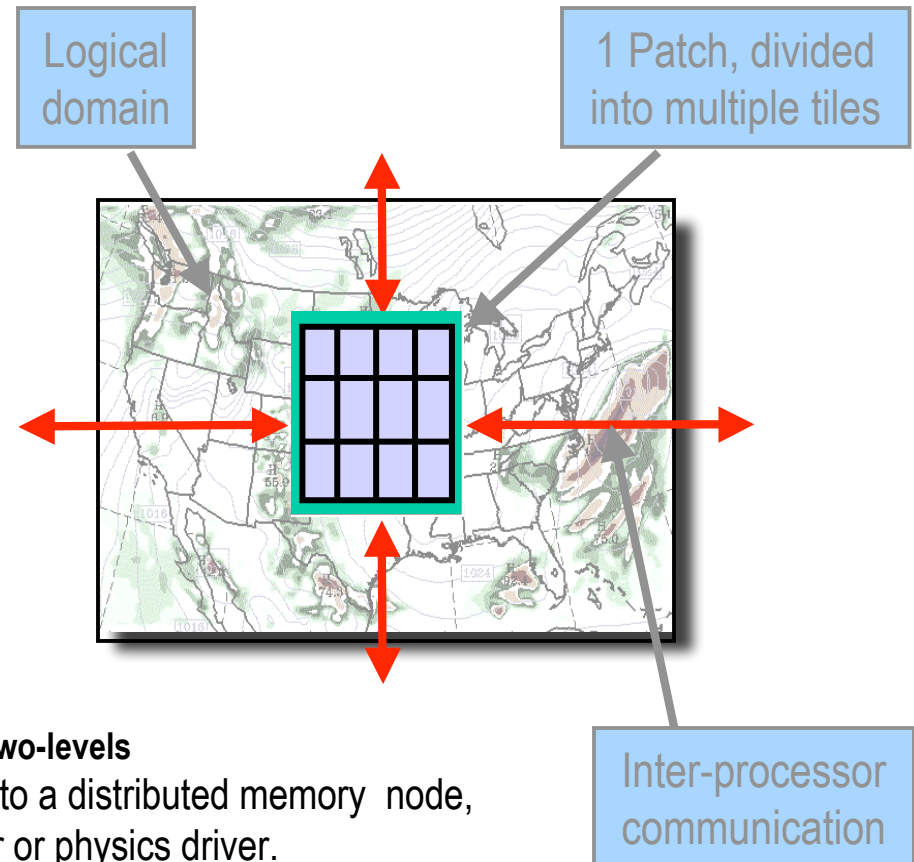
Application: WRF

- Since the process model has two levels (heavy-weight and light-weight = MPI and OpenMP), the decomposition of the application over processes has two levels:
 - The *domain* is first broken up into rectangular pieces that are assigned to heavy-weight processes. These pieces are called *patches*
 - The *patches* may be further subdivided into smaller rectangular pieces that are called *tiles*, and these are assigned to *threads* within the process.



Parallelism in WRF: Multi-level Decomposition

- Single version of code for efficient execution on:
 - Distributed-memory
 - Shared-memory (SMP)
 - Clusters of SMPs
 - Vector and microprocessors



Model domains are decomposed for parallelism on two-levels

Patch: section of model domain allocated to a distributed memory node, this is the scope of a mediation layer solver or physics driver.

Tile: section of a patch allocated to a shared-memory processor within a node; this is also the scope of a model layer subroutine.

Distributed memory parallelism is over patches; shared memory parallelism is over tiles within patches

Distributed Memory Communications

When
Needed?

Communication is required between patches when a horizontal index is incremented or decremented on the right-hand-side of an assignment.

Why?

On a patch boundary, the index may refer to a value that is on a different patch.

Following is an example code fragment that requires communication between patches

Note the tell-tale **+1** and **-1** expressions in indices for **rr**, **H1**, and **H2** arrays on right-hand side of assignment.

These are ***horizontal data dependencies*** because the indexed operands may lie in the patch of a neighboring processor. That neighbor's updates to that element of the array won't be seen on this processor.

Dr Phil

We have to communicate.

Distributed Memory Communications

```
(module_diffusion.F )
```

```
SUBROUTINE horizontal_diffusion_s (tendency, rr, var, . . .  
. . .  
  DO j = jts,jte  
  DO k = kts,ktf  
  DO i = its,ite  
    mrdx=msft(i,j)*rdx  
    mrdy=msft(i,j)*rdy  
    tendency(i,k,j)=tendency(i,k,j) -  
      (mrdx*0.5*((rr(i+1,k,j)+rr(i,k,j))*H1(i+1,k,j) -  
        (rr(i-1,k,j)+rr(i,k,j))*H1(i,k,j)) +  
      mrdy*0.5*((rr(i,k,j+1)+rr(i,k,j))*H2(i,k,j+1) -  
        (rr(i,k,j-1)+rr(i,k,j))*H2(i,k,j-1)) -  
      msft(i,j)*(H1avg(i,k+1,j)-H1avg(i,k,j) +  
        H2avg(i,k+1,j)-H2avg(i,k,j)  
        )/dzetaw(k)  
  )  
  ENDDO  
  ENDDO  
  ENDDO  
. . .
```

Distributed Memory Communications

(module_diffusion.F)

SUBROUTINE horizontal_diffusion_s (tendency, rr, var, . . .

. . .

DO j = jts,jte

DO k = kts,ktf

DO i = its,ite

mrndx=msft(i,j)*rdx

mrddy=msft(i,j)*rdy

```
tendency(i,k,j)=tendency(i,k,j) - &
    (mrndx*0.5* ((rr(i+1,k,j)+rr(i,k,j))*H1(i+1,k,j) - &
    (rr(i-1,k,j)+rr(i,k,j))*H1(i,k,j)) + &
    mrddy*0.5* ((rr(i,k,j+1)+rr(i,k,j))*H2(i,k,j+1) - &
    (rr(i,k,j-1)+rr(i,k,j))*H2(i,k,j)) - &
    msft(i,j)*(H1avg(i,k+1,j)-H1avg(i,k,j) + &
    H2avg(i,k+1,j)-H2avg(i,k,j) &
    )/dzetaw(k) &
    )
```

ENDDO

ENDDO

ENDDO

. . .

Distributed Memory Communications

```
(module_diffusion.F )
```

```
SUBROUTINE horizontal_diffusion_s (tendency, rr, var, . . .
```

```
. . .
```

```
DO j = jts,jte
```

```
DO k = kts,ktf
```

```
DO i = its,ite
```

```
  mrdx=msft(i,j)*rdx
```

```
  mrdy=msft(i,j)*rdy
```

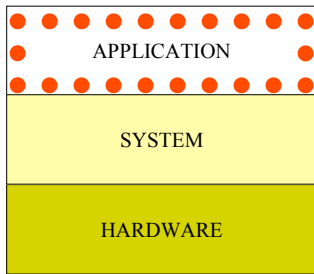
```
  tendency(i,k,j)=tendency(i,k,j) -
    (mrdx*0.5* ((rr(i+1,k,j)+rr(i,k,j))*H1(i+1,k,j) -
      (rr(i-1,k,j)+rr(i,k,j))*H1(i,k,j)) +
    mrdy*0.5* ((rr(i,k,j+1)+rr(i,k,j))*H2(i,k,j+1) -
      (rr(i,k,j-1)+rr(i,k,j))*H2(i,k,j)) -
    msft(i,j)*(H1avg(i,k+1,j)-H1avg(i,k,j)+
      H2avg(i,k+1,j)-H2avg(i,k,j)
      )/dzetaw(k)
  )
```

```
ENDDO
```

```
ENDDO
```

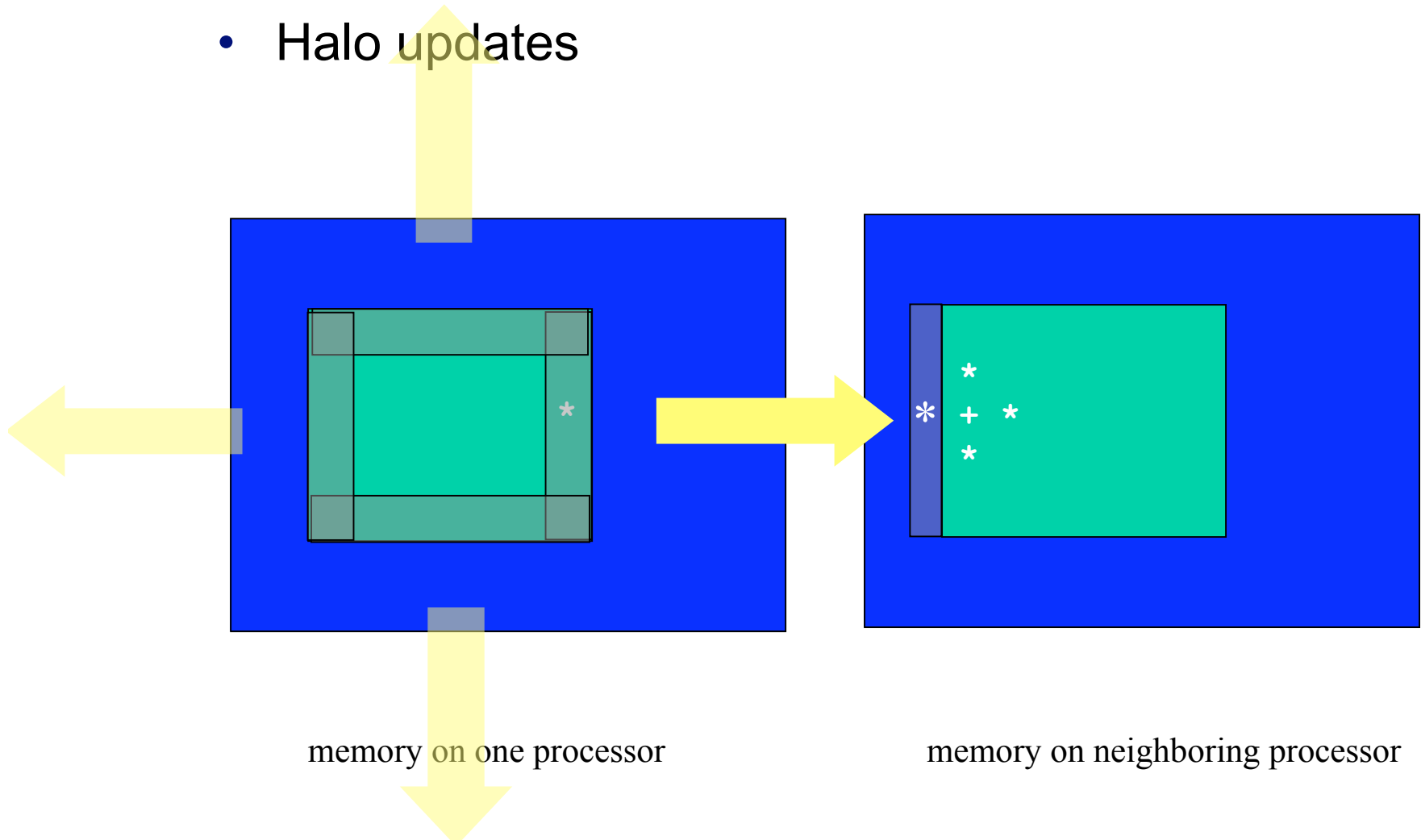
```
ENDDO
```

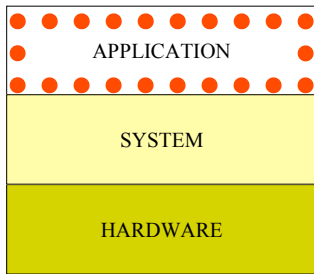
```
. . .
```



Distributed Memory MPI Communications

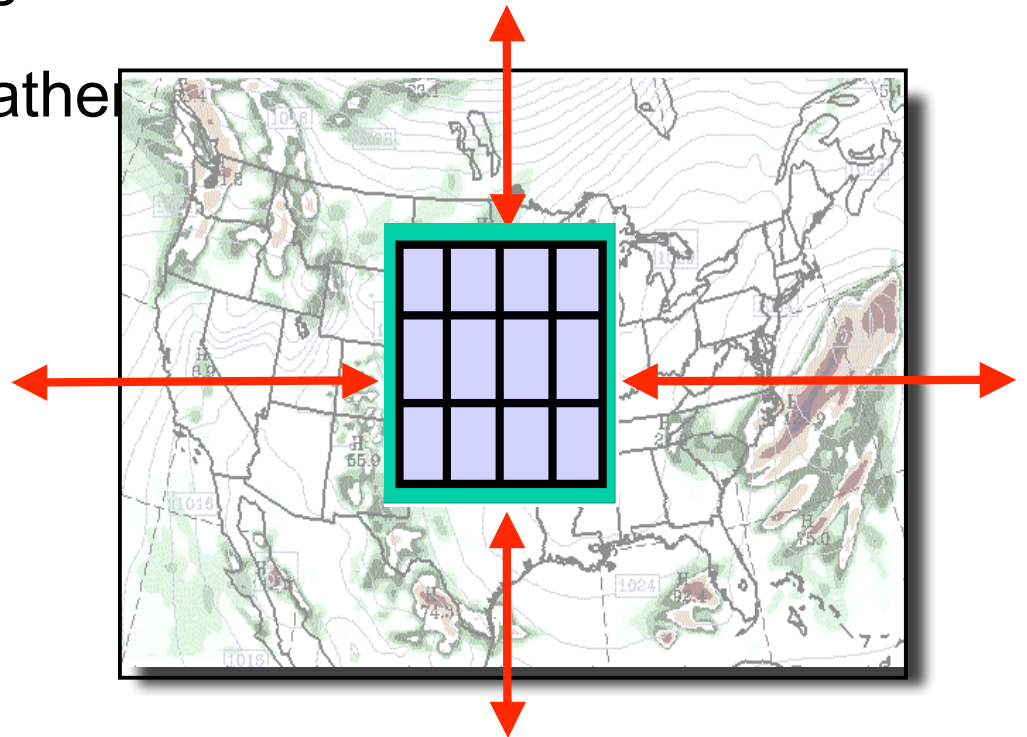
- Halo updates

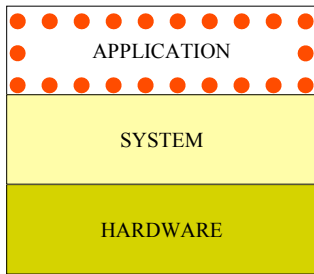




Distributed Memory (MPI) Communications

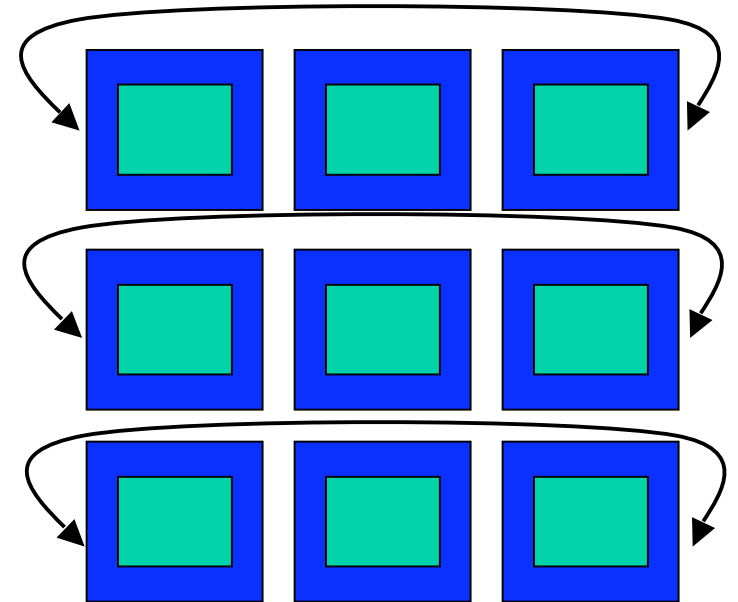
- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gather



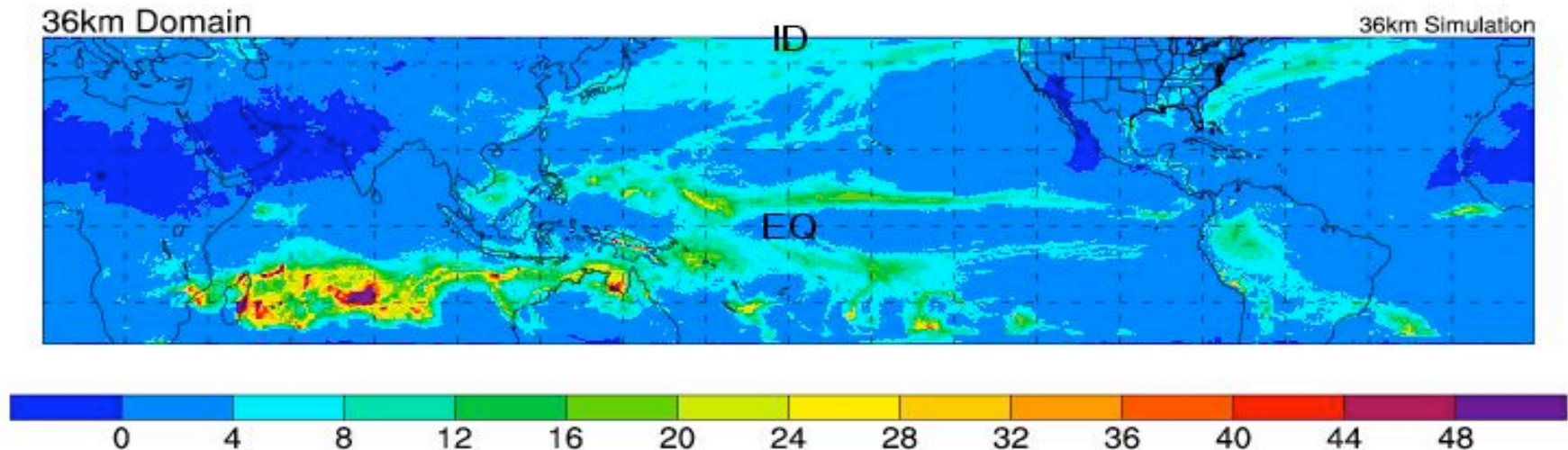


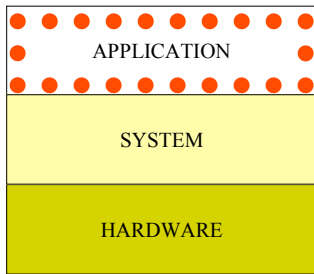
Distributed Memory (MPI) Communications

- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



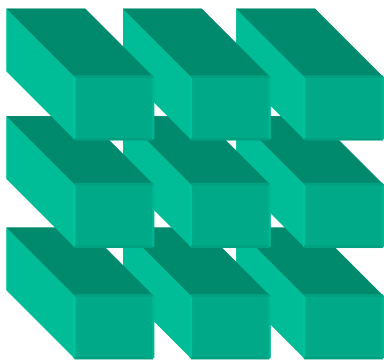
Average Daily Total rainfall (mm) - March 1997



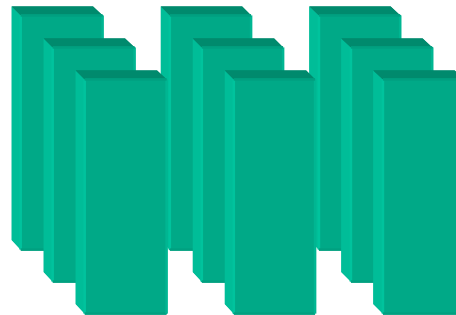


Distributed Memory (MPI) Communications

- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



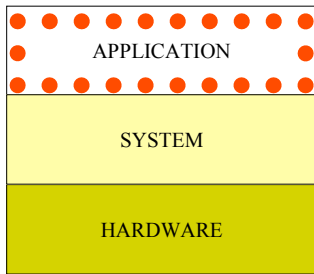
all y on
patch



all z on
patch

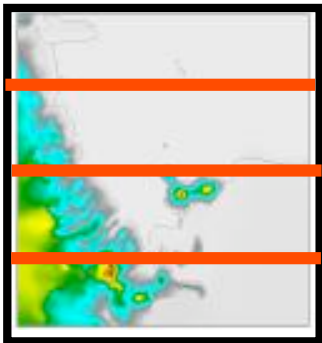


all x on
patch

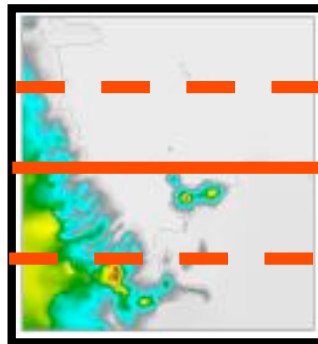


Distributed Memory (MPI) Communications

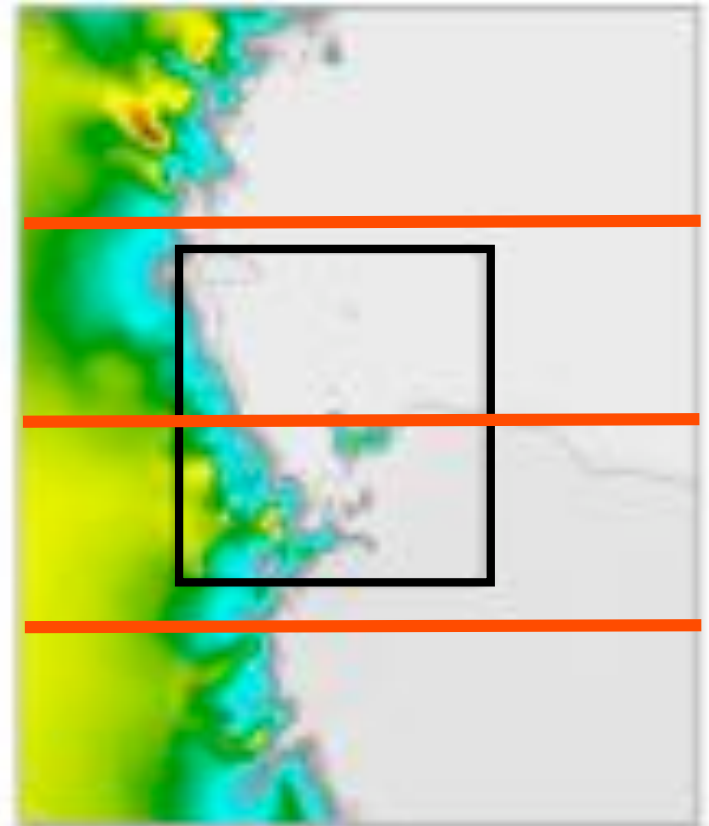
- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



NEST: 2.22 km

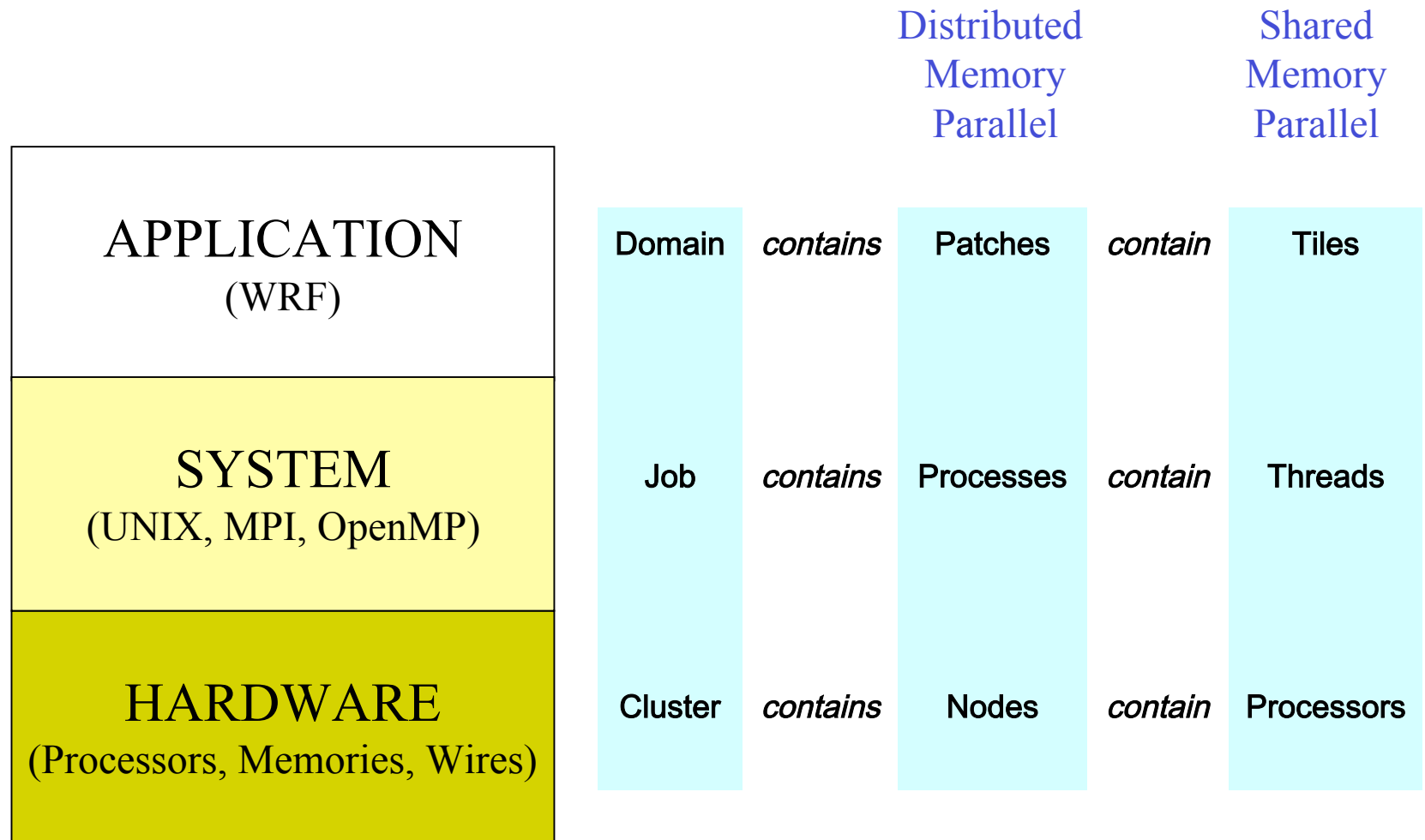


INTERMEDIATE: 6.66 km



COARSE
Ross Island
6.66 km

Review – Computing Overview



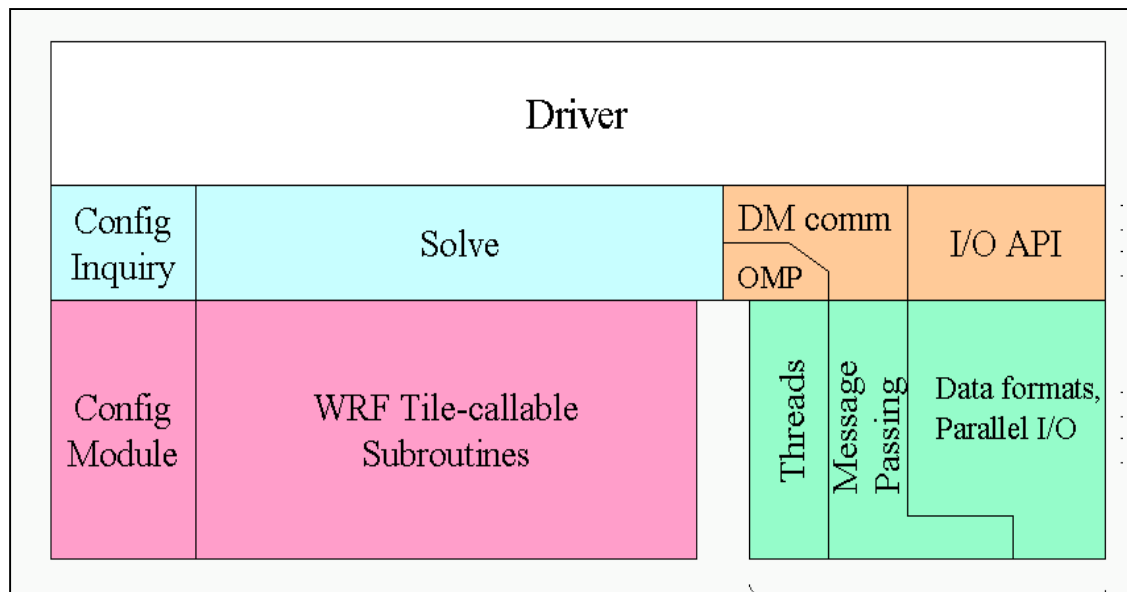
Outline

- Introduction
- Computing Overview
- WRF Software Overview

WRF Software Overview

- Architecture
- Directory structure
- Model Layer Interface
- Data Structures
- I/O

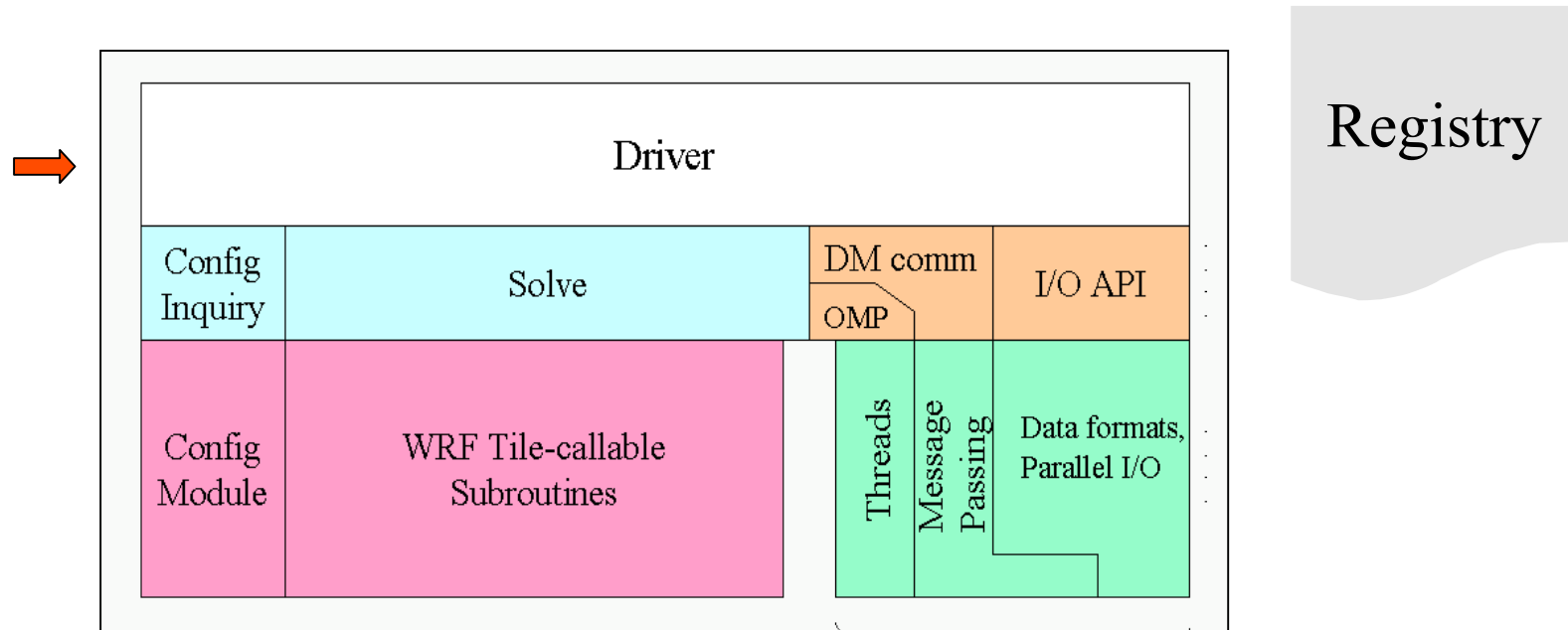
WRF Software Architecture



Registry

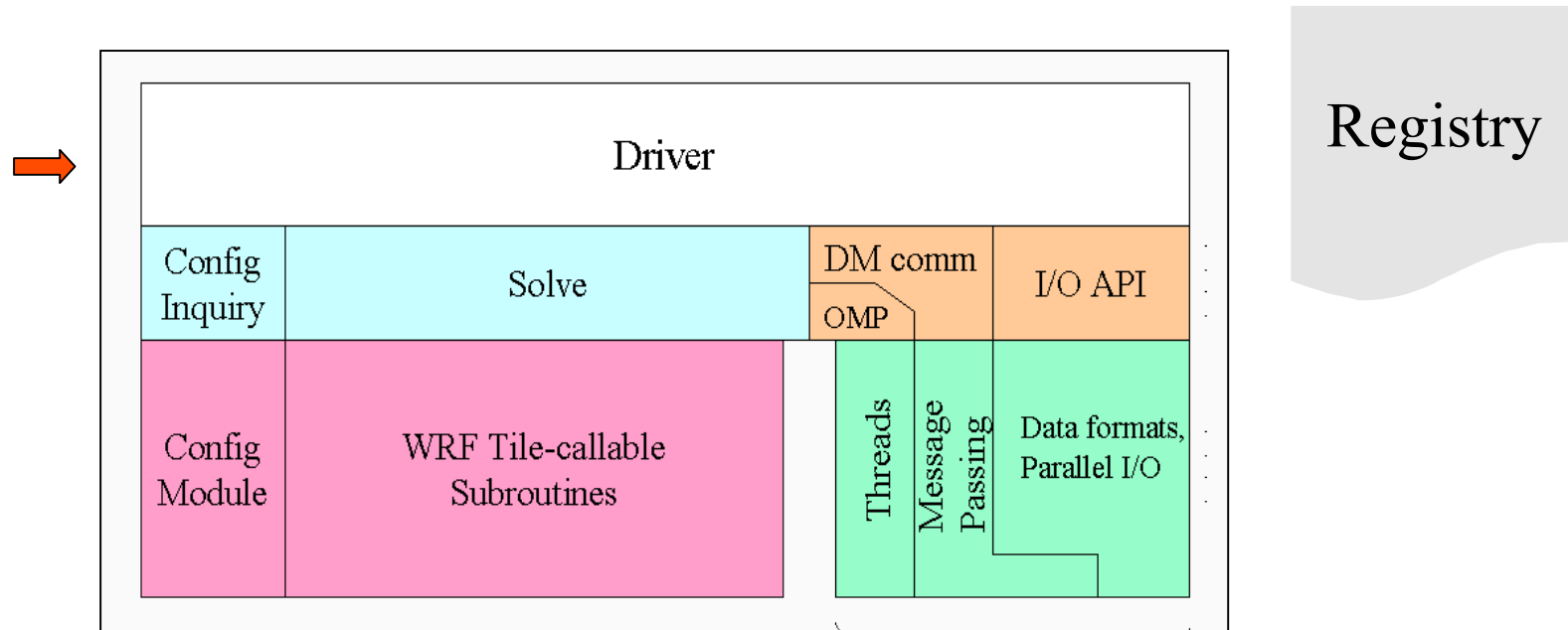
- Hierarchical software architecture
 - Insulate scientists' code from parallelism and other architecture/implementation-specific details
 - Well-defined interfaces between layers, and external packages for communications, I/O, and model coupling facilitates code reuse and exploiting of community infrastructure, e.g. ESMF.

WRF Software Architecture



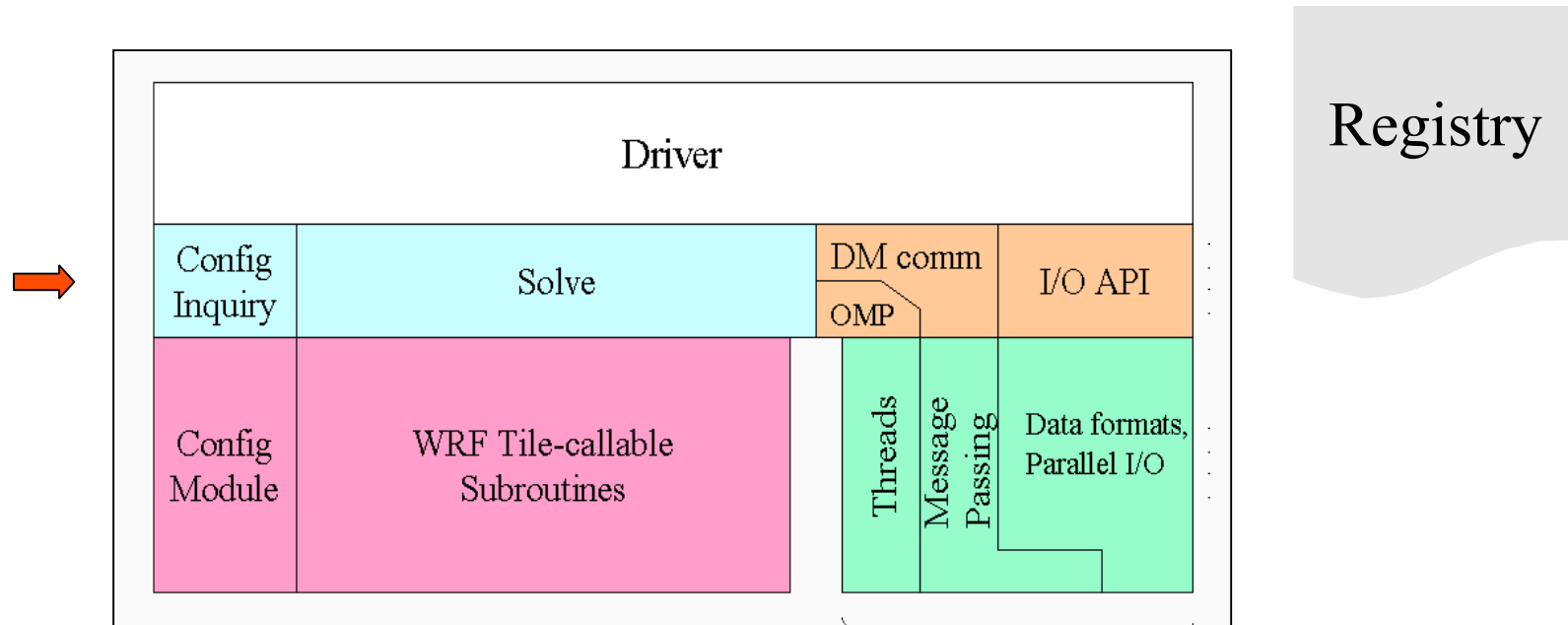
- Driver Layer
 - **Domains:** Allocates, stores, decomposes, represents abstractly as single data objects
 - **Time loop:** top level, algorithms for integration over nest hierarchy
 - **Mediation Layer calls:** nest forcing and feedback

WRF Software Architecture



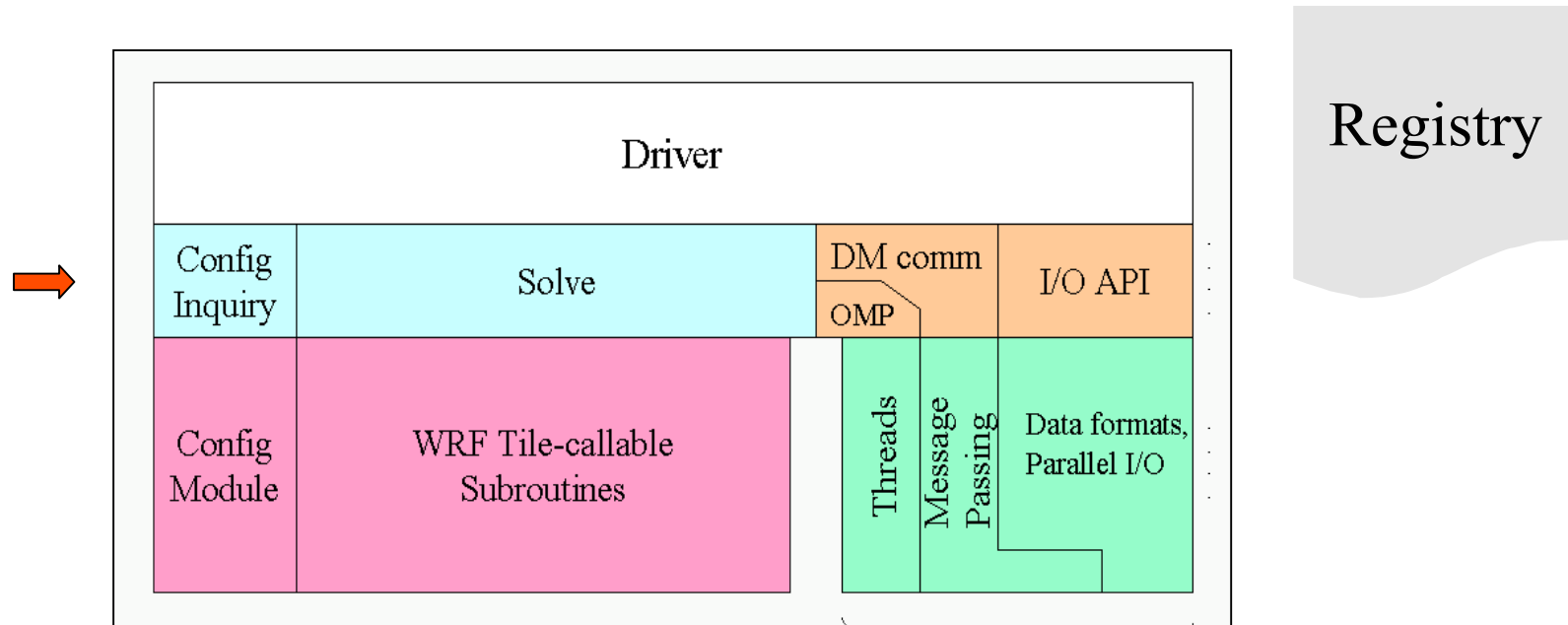
- Driver Layer
 - **Non package-specific access:** communications and I/O
 - **Utilities:** for example module_wrf_error, which is used for diagnostic prints and error stops, accessibility to run-time options

WRF Software Architecture



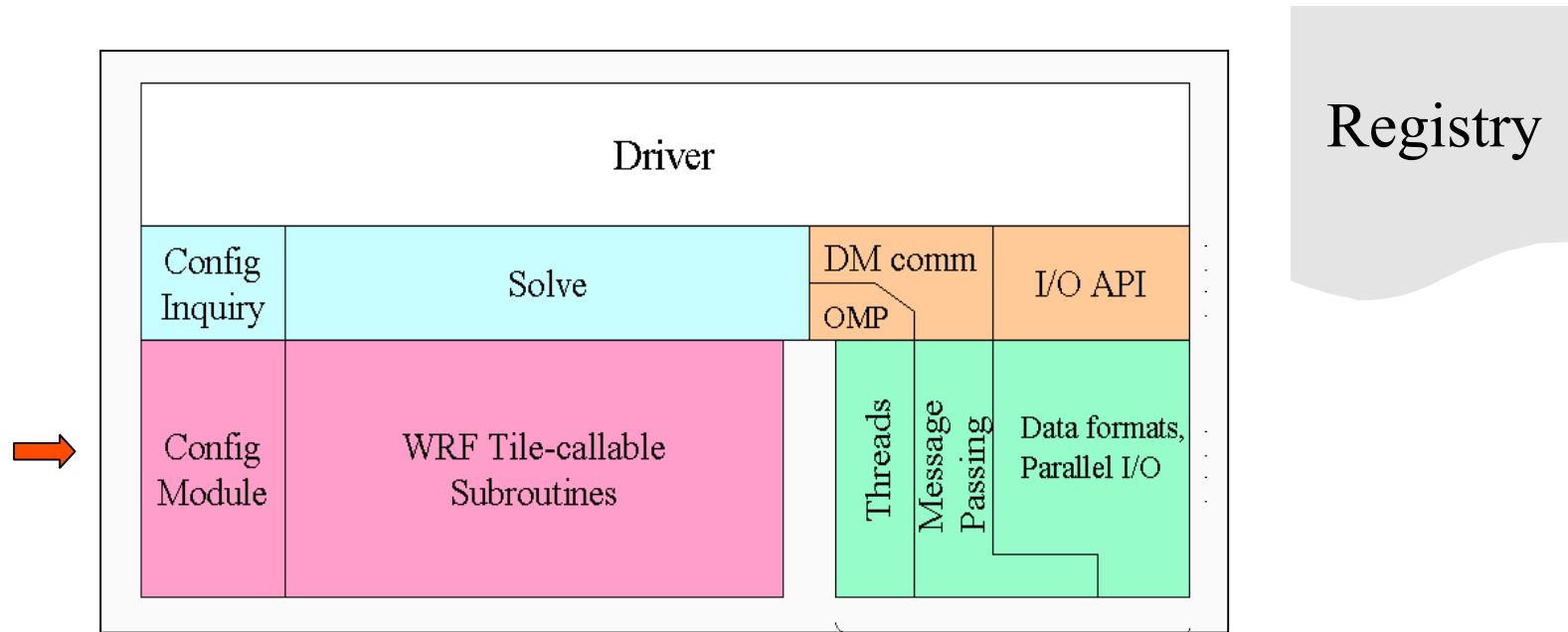
- Mediation Layer
 - Provides to the Driver Layer
 - Solve routine, which takes a domain object and advances it one time step
 - I/O routines that Driver calls when it is time to do some input or output operation on a domain
 - Nest forcing, interpolation, and feedback routines

WRF Software Architecture



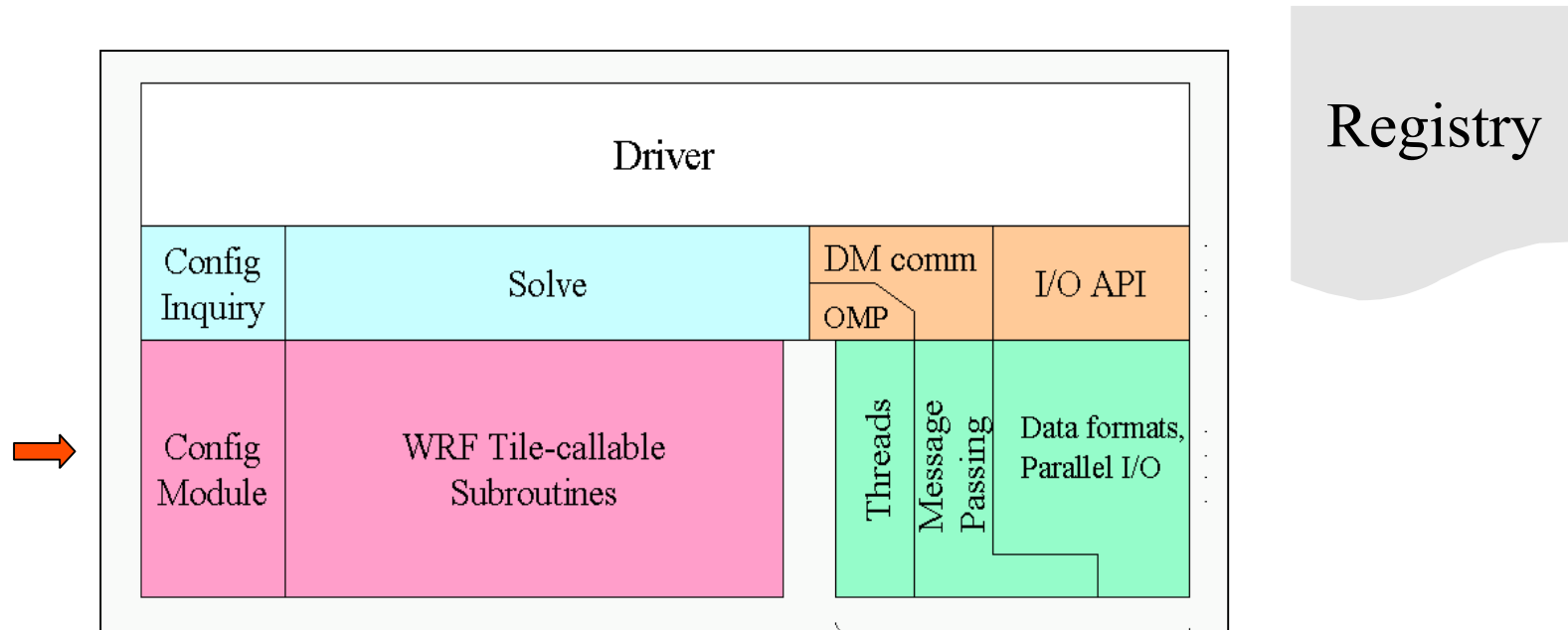
- Mediation Layer
 - Provides to Model Layer
 - The sequence of calls for doing a time-step for one domain is known in Solve routine
 - Dereferences fields in calls to physics drivers and dynamics code
 - Calls to message-passing are contained here as part of Solve routine

WRF Software Architecture



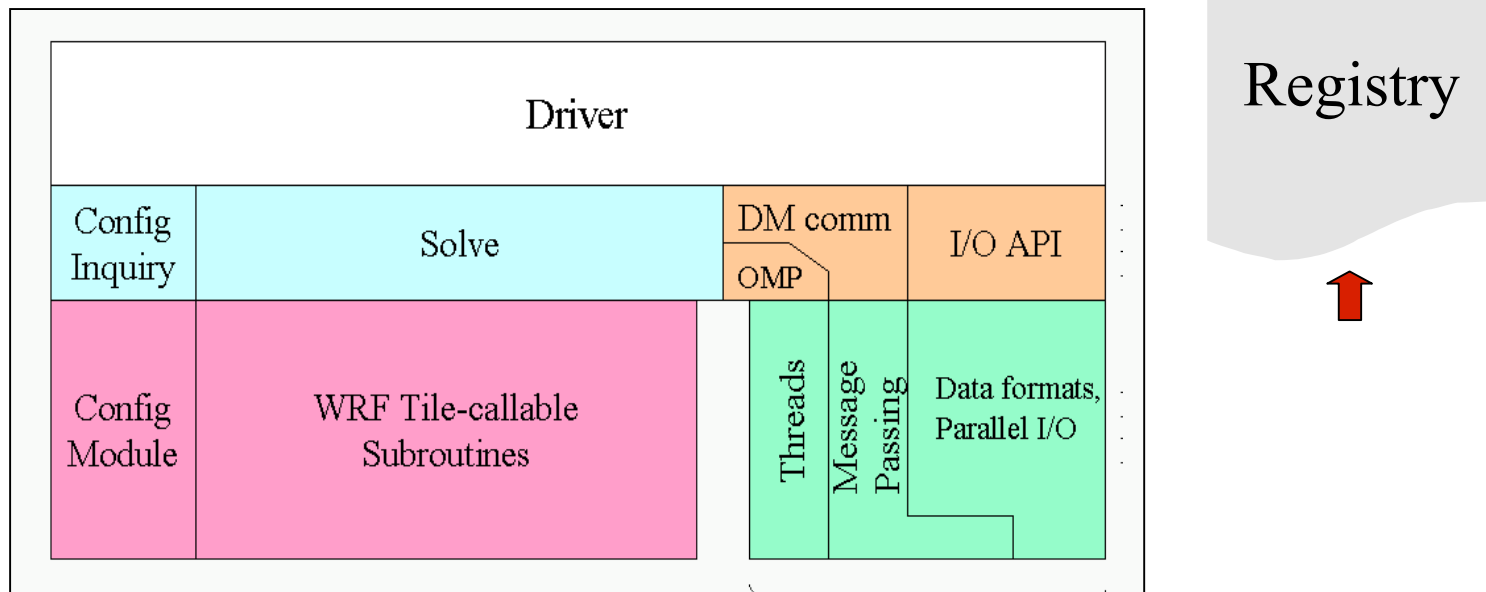
- Model Layer
 - **Information about the model itself:** machine architecture and implementation aspects abstracted out and moved into layers above
 - **Physics and Dynamics:** contains the actual WRF model routines are written to perform some computation over an arbitrarily sized/shaped subdomain

WRF Software Architecture



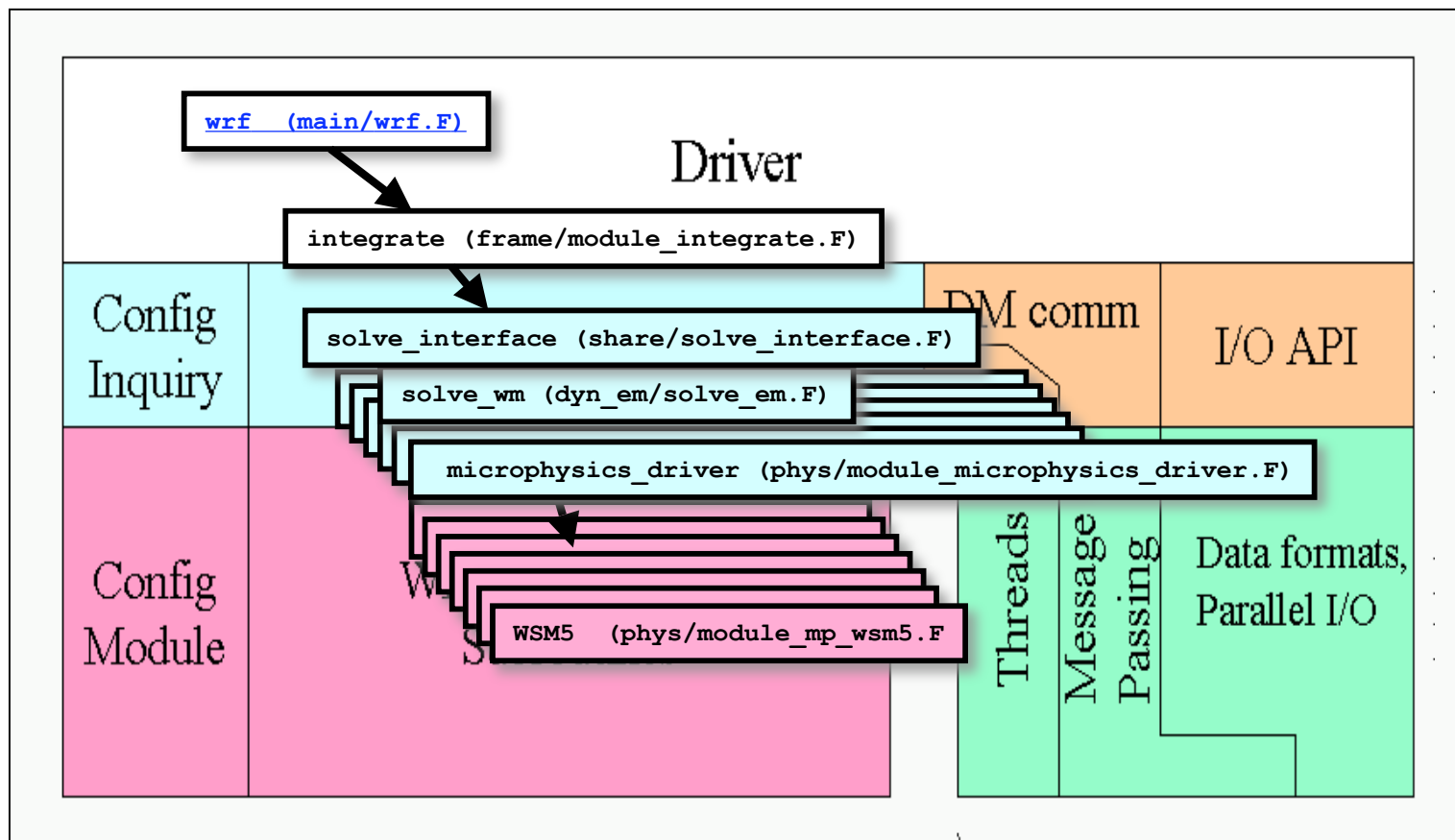
- Model Layer
 - **F77-esque**: all state data objects are simple types, passed in through argument list from physics drivers
 - **No I/O, comms, control**: Model Layer routines don't know anything about communication or I/O, executed on **one thread** – they never contain a **PRINT**, **WRITE**, or **STOP** statement
 - **Model Layer Subroutine Interface**: “tile-callable”, no external COMMON, no decomposed heap data

WRF Software Architecture



- Registry: an “Active” data dictionary
 - Tabular listing of model state and attributes
 - Large sections of interface code generated automatically
 - Scientists manipulate model state simply by modifying Registry, without further knowledge of code mechanics
 - Special “cases” exist: chemistry, SST coupling

Call Structure Superimposed on Architecture



WRF Software Overview

- Architecture
- Directory structure
- Model Layer Interface
- Data Structures
- I/O

WRF Model Top-Level Directory Structure

[WRF Design
and
Implementation](#)

Doc, p 5

DRIVER ●
MEDIATION ●
MODEL ●

Makefile

README

README_test_cases

clean

compile

configure

Registry/
arch/

● dyn_em/
● dyn_nnm/
external/

● frame/
inc/

● main/

● phys/

● share/
tools/

run/

test/

build

scripts

CASE input files
machine build rules

source

code

directories

execution

directories

WRF Software Overview

- Architecture
- Directory structure
- Model Layer Interface
- Data Structures
- I/O

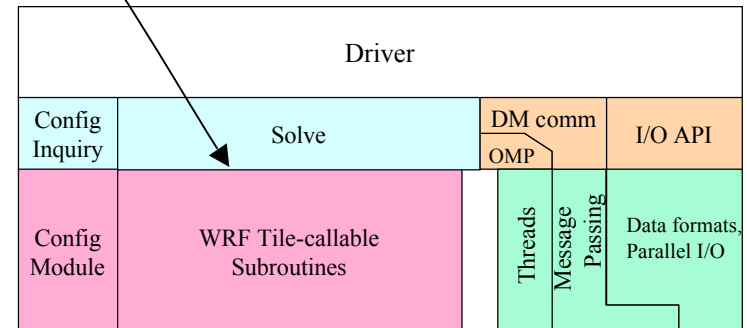
WRF Model Layer Interface

Mediation Layer / Model Layer Interface

All state arrays passed through argument list as simple (not derived) data types

Domain, memory, and run dimensions passed unambiguously in three physical dimensions

Model layer routines are called from mediation layer (physics drivers) in loops over tiles, which are multi-threaded

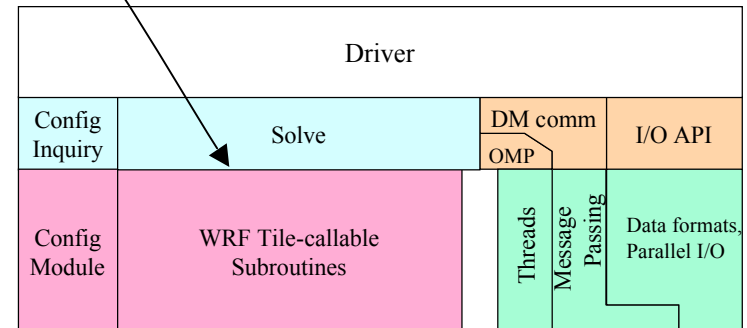


WRF Model Layer Interface

Mediation Layer / Model Layer Interface

Restrictions on Model Layer subroutines:

No I/O, communication, no stops or aborts
(use `wrf_error_fatal` in
`frame/module_wrf_error.F`)



No common/module storage of decomposed data (exception allowed for set-once/read-only tables)

Spatial scope of a Model Layer call is one “tile”

WRF Model Layer Interface

- Mediation layer / Model Layer Interface
- Model layer routines are called from mediation layer in loops over tiles, which are multi-threaded
- All state arrays passed through argument list as simple data types

WRF Model Layer Interface

- Domain, memory, and run dimensions passed unambiguously in three physical dimensions
- Restrictions on model layer subroutines
 - No I/O, communication, no stops or aborts (use `wrf_error_fatal` in `frame/module_wrf_error.F`)
 - No common/module storage of decomposed data (exception allowed for set-once/read-only tables)
 - Spatial scope of a Model Layer call is one “tile”
 - Temporal scope of a call is limited by coherency

WRF Model Layer Interface

```
SUBROUTINE driver_for_some_physics_suite (  
    . . .  
    !$OMP DO PARALLEL  
        DO ij = 1, numtiles  
            its = i_start(ij) ; ite = i_end(ij)  
            jts = j_start(ij) ; jte = j_end(ij)  
            CALL model_subroutine( arg1, arg2, . . .  
                ids , ide , jds , jde , kds , kde ,  
                ims , ime , jms , jme , kms , kme ,  
                its , ite , jts , jte , kts , kte )  
        END DO  
    . . .  
END SUBROUTINE
```

WRF Model Layer Interface

template for model layer subroutine

```
SUBROUTINE model_subroutine ( &
    arg1, arg2, arg3, ... , argn,    &
    ids, ide, jds, jde, kds, kde, & ! Domain dims
    ims, ime, jms, jme, kms, kme, & ! Memory dims
    its, ite, jts, jte, kts, kte ) ! Tile dims

IMPLICIT NONE

! Define Arguments (State and I1) data
REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .
REAL, DIMENSION (ims:ime,jms:jme)          :: arg7, . . .
. . .
! Define Local Data (I2)
REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .
. . .
```

WRF Model Layer Interface

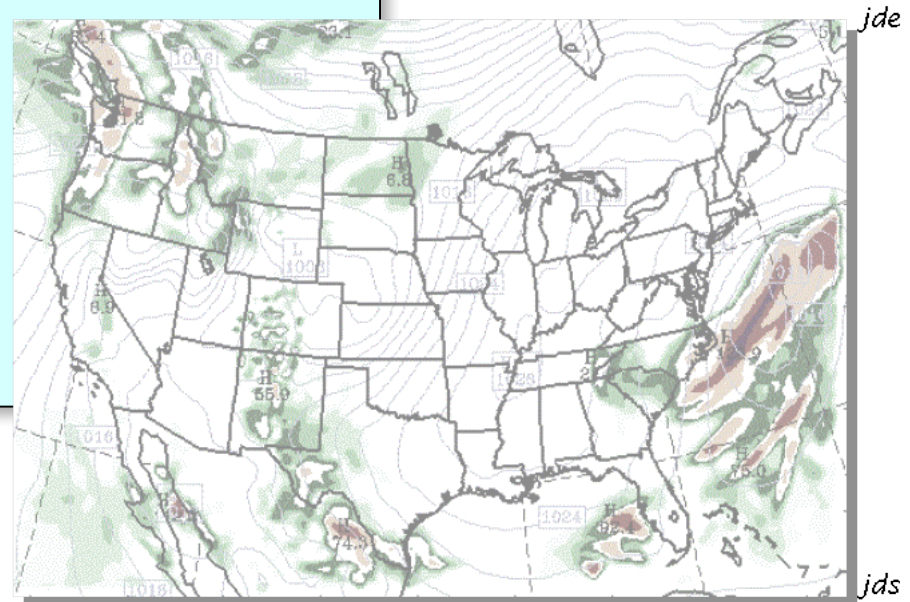
template for model layer subroutine

```
. . .  
! Executable code; loops run over tile  
! dimensions  
DO j = MAX(jts,jds), MIN(jte,jde-1)  
  DO k = kts, kte  
    DO i = MAX(its,ids), MIN(ite,ide-1)  
      loc1(i,k,j) = arg1(i,k,j) + ...  
    END DO  
  END DO  
END DO
```

template for model layer subroutine

```
SUBROUTINE model ( &  
  arg1, arg2, arg3, ..., argn, &  
  ids, ide, jds, jde, kds, kde, & ! Domain dims  
  ims, ime, jms, jme, kms, kme, & ! Memory dims  
  its, ite, jts, jte, kts, kte ) ! Tile dims  
  
  IMPLICIT NONE  
  
  ! Define Arguments (S and I1) data  
  REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .  
  REAL, DIMENSION (ims:ime,jms:jme)          :: arg7, . . .  
  . . .  
  ! Define Local Data (I2)  
  REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .  
  . . .  
  ! Executable code; loops run over tile  
  ! dimensions  
  DO j = MAX(jts,jds), MIN(jte,jde-1)  
    DO k = kts, kte  
      DO i = MAX(its,ids), MIN(ite,ide-1)  
        loc1(i,k,j) = arg1(i,k,j) + ...  
      END DO  
    END DO  
  END DO
```

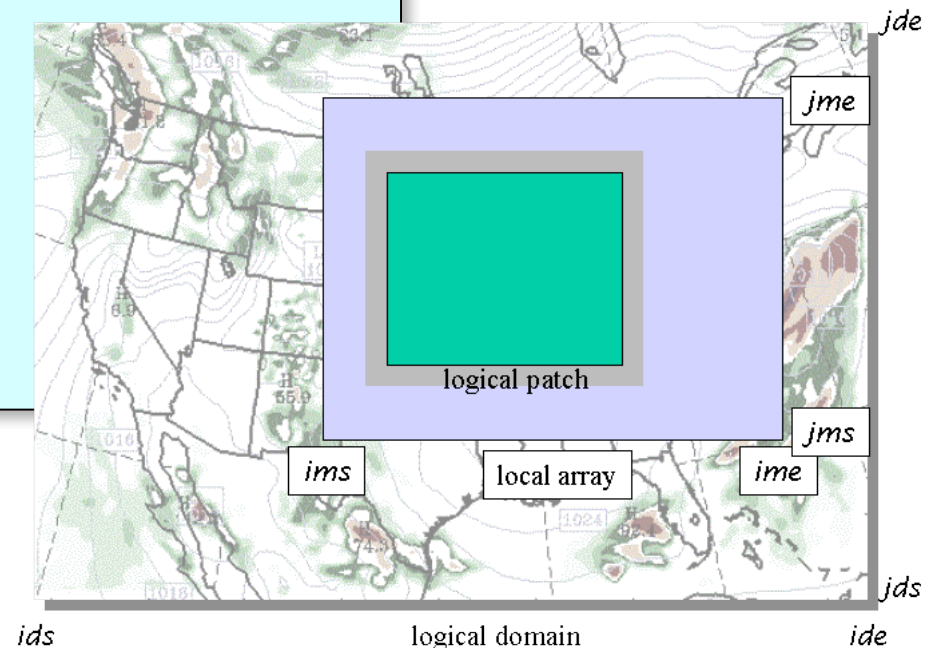
- Domain dimensions
 - Size of logical domain
 - Used for bdy tests, etc.



template for model layer subroutine

```
SUBROUTINE model ( &  
  arg1, arg2, arg3, ... , argn, &  
  ids, ide, jds, jde, kds, kde, & ! Domain dims  
  ims, ime, jms, jme, kms, kme, & ! Memory dims  
  its, ite, jts, jte, kts, kte ) ! Tile dims  
  
  IMPLICIT NONE  
  
  ! Define Arguments (S and I1) data  
  REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .  
  REAL, DIMENSION (ims:ime,jms:jme) :: arg7, . . .  
  . . .  
  ! Define Local Data (I2)  
  REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .  
  . . .  
  ! Executable code; loops run over tile  
  ! dimensions  
  DO j = MAX(jts,jds), MIN(jte,jde-1)  
    DO k = kts, kte  
      DO i = MAX(its,ids), MIN(ite,ide-1)  
        loc1(i,k,j) = arg1(i,k,j) + ...  
      END DO  
    END DO  
  END DO
```

- Domain dimensions
 - Size of logical domain
 - Used for bdy tests, etc.
- Memory dimensions
 - Used to dimension dummy arguments
 - Do not use for local arrays



template for model layer subroutine

```

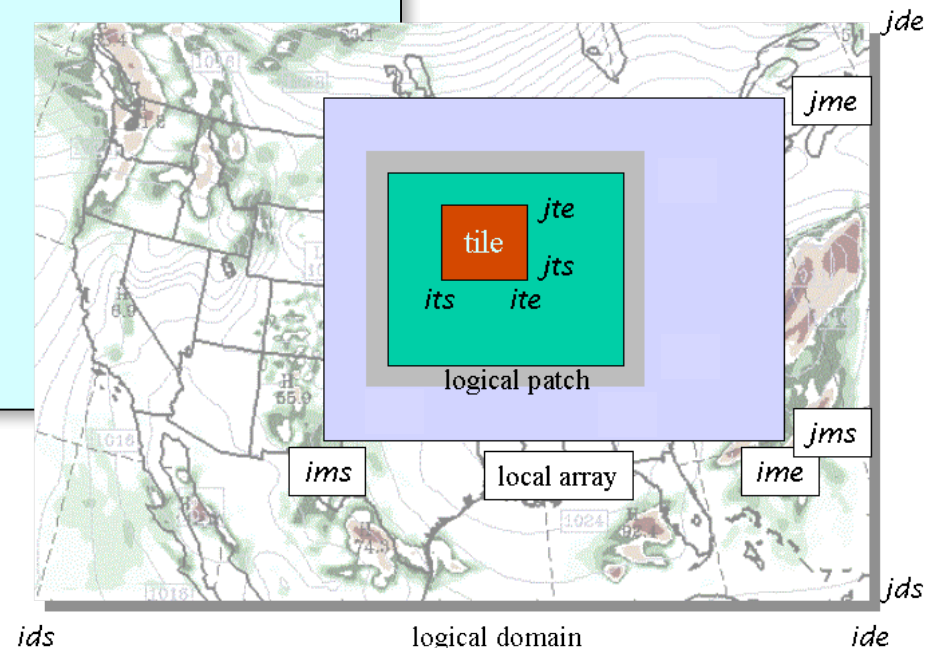
SUBROUTINE model ( &
  arg1, arg2, arg3, ... , argn,    &
  ids, ide, jds, jde, kds, kde, & ! Domain dims
  ims, ime, jms, jme, kms, kme, & ! Memory dims
  its, ite, jts, jte, kts, kte ) ! Tile dims

IMPLICIT NONE

! Define Arguments (S and I1) data
REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .
REAL, DIMENSION (ims:ime,jms:jme)          :: arg7, . . .
. . .
! Define Local Data (I2).....
REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .
. . .
! Executable code; loops run over tile
! dimensions
DO j = MAX(jts,jds), MIN(jte,jde-1)
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide-1)
      loc1(i,k,j) = arg1(i,k,j) + ...
    END DO
  END DO
END DO
END DO

```

- Domain dimensions
 - Size of logical domain
 - Used for bdy tests, etc.
- Memory dimensions
 - Used to dimension dummy arguments
 - Do not use for local arrays
- Tile dimensions
 - Local loop ranges
 - Local array dimensions



template for model layer subroutine

```

SUBROUTINE model ( &
  arg1, arg2, arg3, ... , argn,    &
  ids, ide, jds, jde, kds, kde, & ! Domain dims
  ims, ime, jms, jme, kms, kme, & ! Memory dims
  its, ite, jts, jte, kts, kte ) ! Tile dims

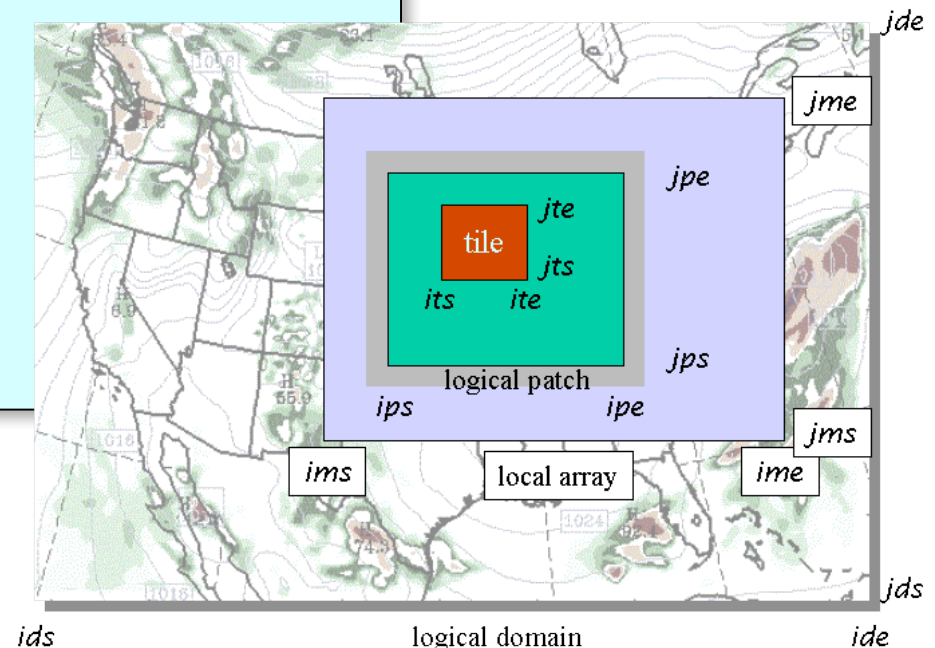
IMPLICIT NONE

! Define Arguments (S and I1) data
REAL, DIMENSION (ims:ime,kms:kme,jms:jme) :: arg1, . . .
REAL, DIMENSION (ims:ime,jms:jme)          :: arg7, . . .
. . .
! Define Local Data (I2).....
REAL, DIMENSION (its:ite,kts:kte,jts:jte) :: loc1, . . .
. . .
! Executable code; loops run over tile
! dimensions
DO j = MAX(jt,jds), MIN(jte,jde-1)
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide-1)
      loc1(i,k,j) = arg1(i,k,j) + ...
    END DO
  END DO
END DO

```

- Domain dimensions
 - Size of logical domain
 - Used for bdy tests, etc.
- Memory dimensions
 - Used to dimension dummy arguments
 - Do not use for local arrays
- Tile dimensions
 - Local loop ranges
 - Local array dimensions

- Patch dimensions
 - Start and end indices of local distributed memory subdomain
 - Available from mediation layer (solve) and driver layer; not usually needed or used at model layer

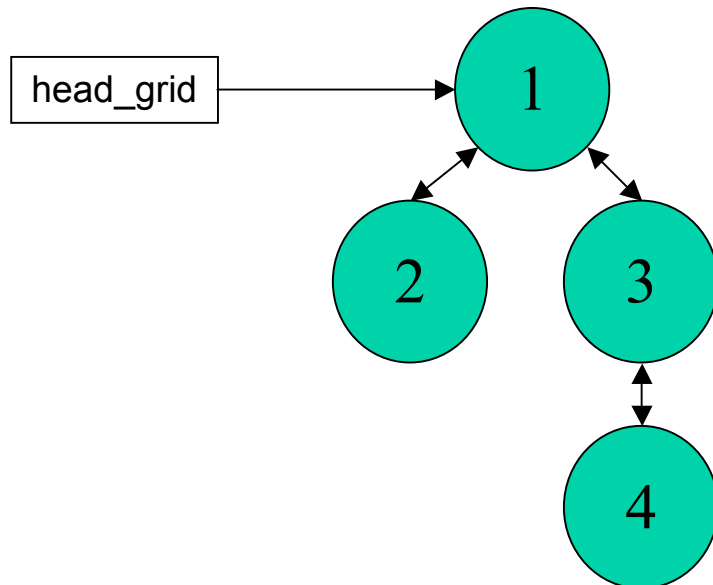


WRF Software Overview

- Architecture
- Directory structure
- Model Layer Interface
- Data Structures
- I/O

Driver Layer Data Structures: Domain Objects

- Driver layer
 - All data for a domain is an object, a domain **derived data type** (DDT)
 - The domain DDTs are dynamically allocated/deallocated
 - Linked together in a tree to represent nest hierarchy; root pointer is **head_grid**, defined in frame/module_domain.F
 - Supports recursive depth-first traversal algorithm (frame/module_integrate.F)



Model Layer Data Structures: F77

- Model layer
 - All data objects are scalars and arrays of simple types only
 - Virtually all passed in through subroutine argument lists
 - Non-decomposed arrays and “local to a module” storage are permitted with an initialization at the model start

Mediation Layer Data Structures: Objects + F77

- Mediation layer
 - One task of mediation layer is to dereference fields from DDTs
 - Therefore, sees domain data in both forms, as DDT and as individual fields which are components of the DDTs
- The name of a data type and how it is referenced differs depending on the level of the architecture

Data Structures

- WRF Data Taxonomy
 - State data
 - Intermediate data type 1 (I1)
 - Intermediate data type 2 (I2)
 - Heap storage (COMMON or Module data)

Data Structures

- WRF Data Taxonomy

- State data
- Intermediate data type 1 (I1)
- Intermediate data type 2 (I2)
- Heap storage (COMMON or Module data)

Defined in the
Registry

Data Structures

- WRF Data Taxonomy

- State data

- Intermediate data type 1 (I1)

- Intermediate data type 2 (I2)

- Heap storage (COMMON or Module data)

Defined in
the
subroutine

Data Structures

- WRF Data Taxonomy
 - State data
 - Intermediate data type 1 (I1)
 - Intermediate data type 2 (I2)
 - Heap storage (COMMON or Module)

Defined in the
module top,
typically
look-up tables
and routine
constants,
NO HORIZ
DECOMPOSED
DATA!

Mediation/Model Layer Data Structures: State Data

- Duration: Persist between start and stop of a domain
- Represented as fields in domain data structure
 - Memory for state arrays are dynamically allocated, only big enough to hold the local subdomain's (ie. patch's) set of array elements
 - Always **memory** dimensioned
 - Declared in Registry using **state** keyword
- Only state arrays can be subject to I/O and Interprocessor communication

Mediation/Model Layer Data Structures:

I1 Data

- Persist for the duration of a single time step in solve
- Represented as fields in domain data structure
 - Memory for I1 arrays are dynamically allocated, only big enough to hold the local subdomain's (ie. patch's) set of array elements
 - Always **memory** dimensioned
 - Declared in Registry using I1 keyword
 - Typically tendency fields computed, used, and discarded in a single time step

Model Layer Data Structures:

I2 Data

- Persist for the duration of a call of the physics routine
- NOT contained within the DDT structure
 - Memory for I2 arrays are dynamically allocated on subroutine entry, and automatically deallocated on exit
 - Always **tile** dimensioned
 - Not declared in the Registry, not communicated, no IO, not passed back to the solver

Grid Representation in Arrays

- Increasing indices in WRF arrays run
 - West to East (X, or I-dimension)
 - South to North (Y, or J-dimension)
 - Bottom to Top (Z, or K-dimension)
- Storage order in WRF is IKJ but this is a WRF Model convention, not a restriction of the WRF Software Framework (provides cache coherency, but long vectors possible)
- Output data has grid ordering independent of the ordering inside the WRF model

Grid Representation in Arrays

- The extent of the logical or *domain* dimensions is always the "staggered" grid dimension. That is, from the point of view of a non-staggered dimension, there is always an extra cell on the end of the domain dimension

WRF Software Overview

- Architecture
- Directory structure
- Model Layer Interface
- Data Structures
- I/O

WRF I/O

- Streams: pathways into and out of model
 - History + 11 auxiliary output streams (10 and 11 are reserved for nudging)
 - Input + 11 auxiliary input streams (10 and 11 are reserved for nudging)
 - Restart, boundary, and a special Var stream

WRF I/O

- Attributes of streams
 - Variable set
 - The set of WRF state variables that comprise one read or write on a stream
 - Defined for a stream at compile time in Registry
 - Format
 - The format of the data outside the program (e.g. NetCDF), split
 - Specified for a stream at run time in the namelist

WRF I/O

- Attributes of streams
 - Additional namelist-controlled attributes of streams
 - Dataset name
 - Time interval between I/O operations on stream
 - Starting, ending times for I/O (**specified as intervals from start of run**)

Outline - Review

- Introduction
 - WRF started 1998, clean slate, Fortran + C
 - Targeted for research and operations
- WRF Software Overview
 - Hierarchical software layers
 - Patches (MPI) and Tiles (OpenMP)
 - Strict interfaces between layers
 - Contract with developers
 - Data Structures
 - I/O