

# WRF Software: Code and Parallel Computing

John Michalakes, Head WRF Software Architecture

Michael Duda

Dave Gill

# Outline

- Introduction
- Computing Overview
- WRF Software Overview

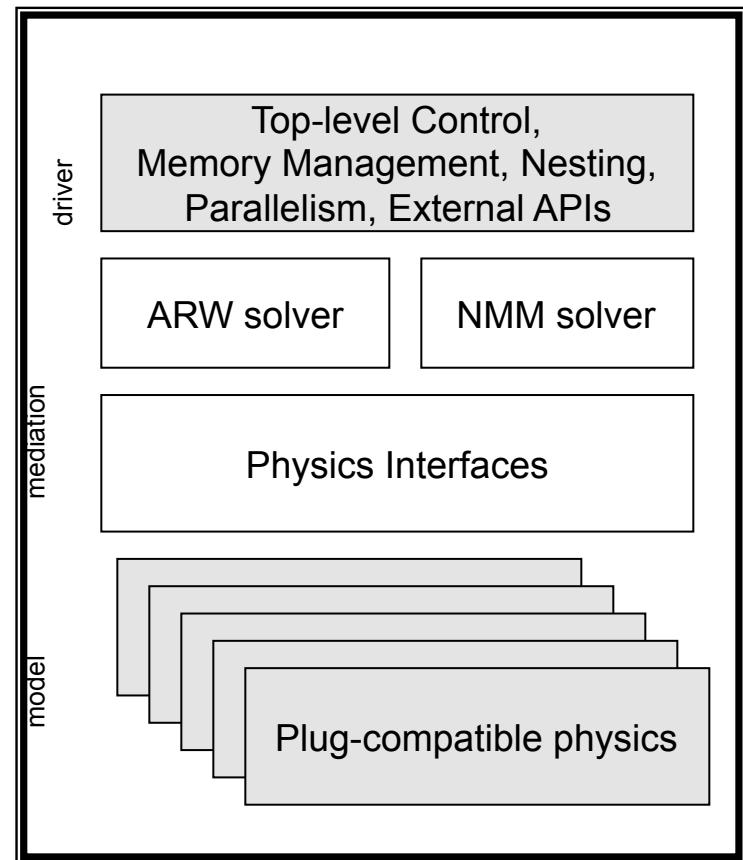
# Introduction – WRF Software Characteristics

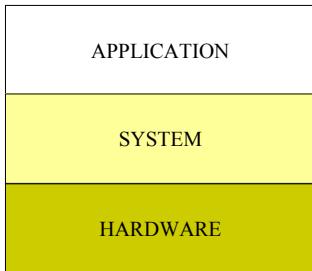
- Developed from scratch beginning around 1998, primarily Fortran and C
- Requirements emphasize flexibility over a range of platforms, applications, users, performance
- WRF develops rapidly. First released Dec 2000; current release WRF v3.4.1 (Aug 2012); next release WRF v3.5 (April 2013)
- Supported 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/>





# 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

APPLICATION

SYSTEM

HARDWARE

# Hardware has not changed much...

IBM 7090



A computer in 1960

6-way superscalar

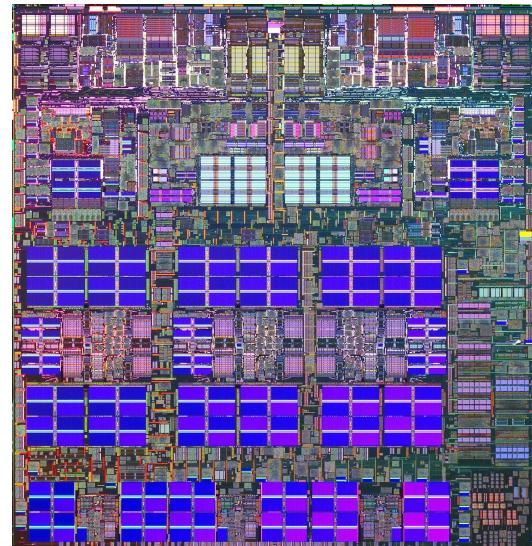
36-bit floating point precision

~144 Kbytes

*~50,000 flop/s*

*48hr 12km WRF CONUS in 600 years*

IBM P6



Dual core, 4.7 GHz chip

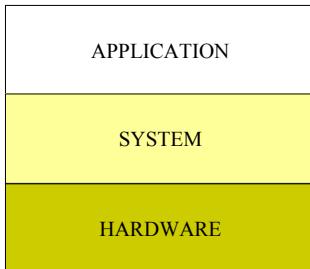
64-bit floating point precision

1.9 MB L2, 36 MB L3

Up to 16 GB per processor

*~5,000,000,000 flop/s*

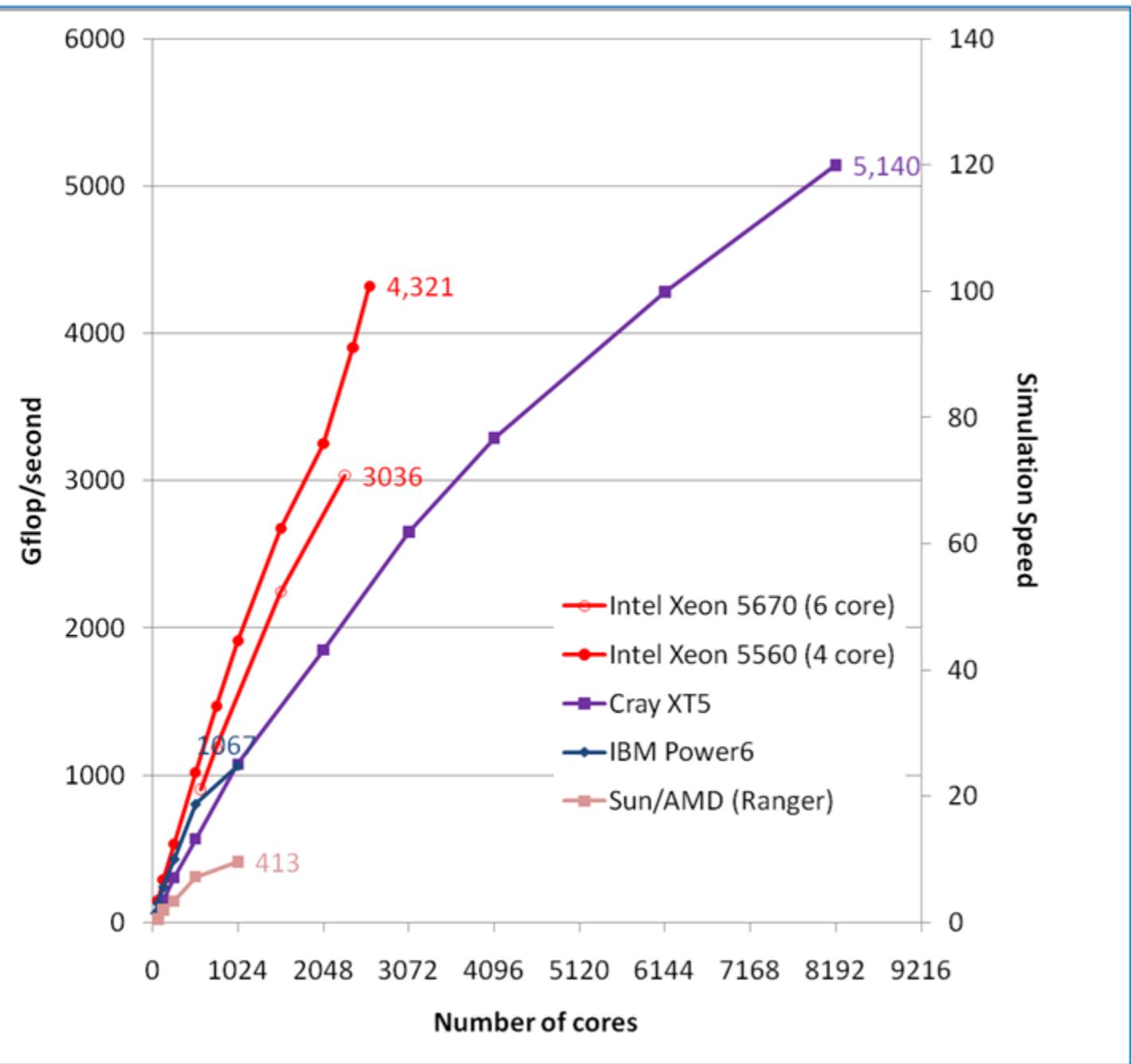
*48 12km WRF CONUS in 52 Hours*



## ...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
- 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  $\sim 250$  procs  
48-h, 12-km WRF CONUS in under 15 minutes*



## January 2000 Benchmark

- 74x61 grid cells
- 1 hour forecast, 3 minute time step, 20 time step average
- I0 excluded

Decomposed domain sizes      proc count: I-dim x J-dim

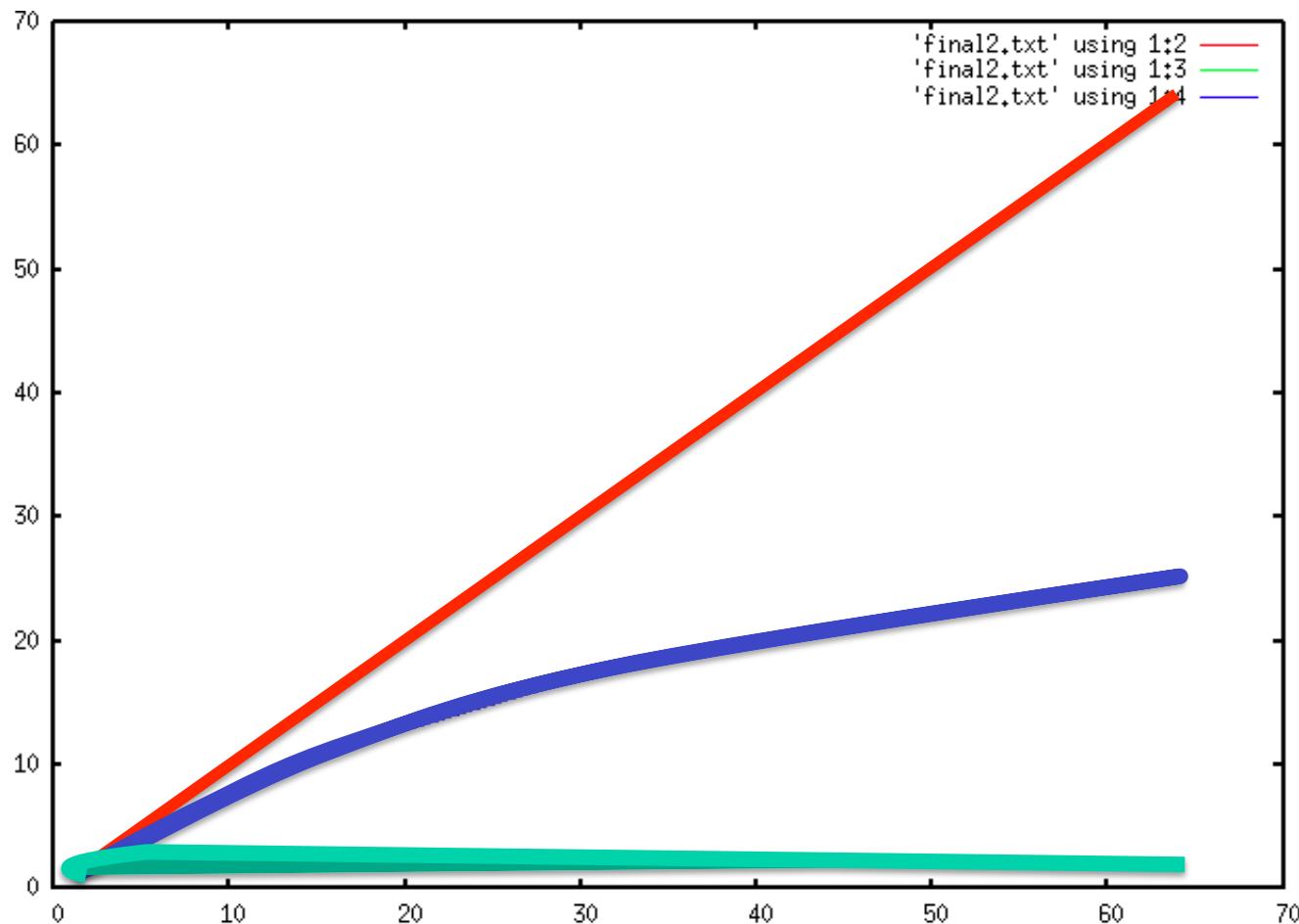
1: 74x61      2: 74x31      4: 37x31      8: 37x16

16: 19x16      32: 19x8      64: 10x8

# January 2000 Benchmark

Processor Count	SM – OpenMP % Efficiency	DM – MPI % Efficiency
1 74x61	100	100
2 74x31	72	98
4 37x31	65	91
8 37x16	31	83
16 19x16	16	70
32 19x8	8	56
64 10x8	3	40

# January 2000 Benchmark



Processor Count

1 2 4 8 16

32

64

# 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

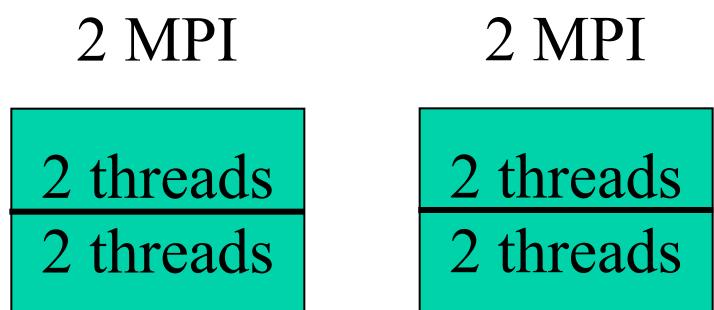
```
setenv OMP_NUM_THREADS 1  
mpirun -np 16 wrf.exe
```

# 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?

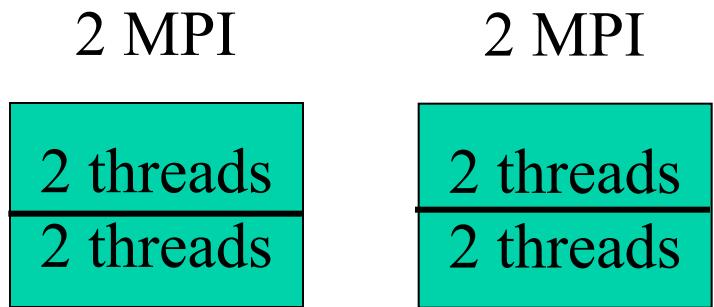
- 4 MPI processes, each with 4 threads

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



- 8 MPI processes, each with 2 threads

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



- 16 MPI processes, each with 1 thread

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

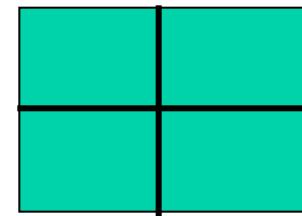
# Examples

- 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?

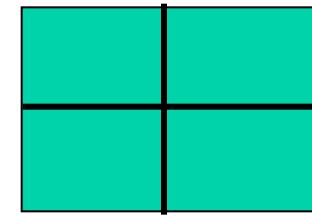
- 4 MPI processes, each with 4 threads

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

4 MPI



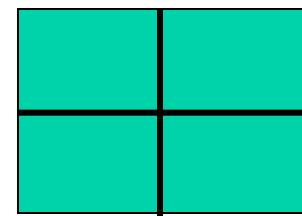
4 MPI



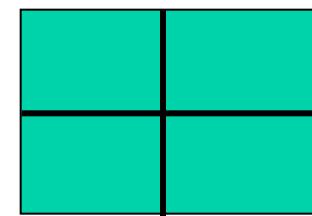
- 8 MPI processes, each with 2 threads

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

4 MPI

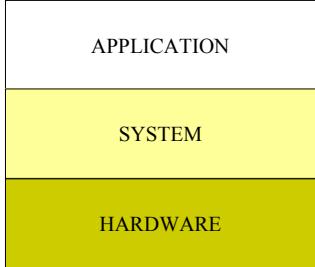


4 MPI



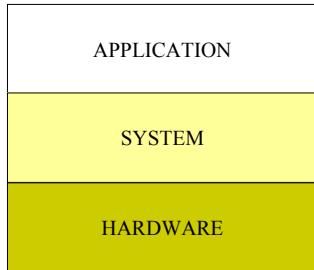
- 16 MPI processes, each with 1 thread

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



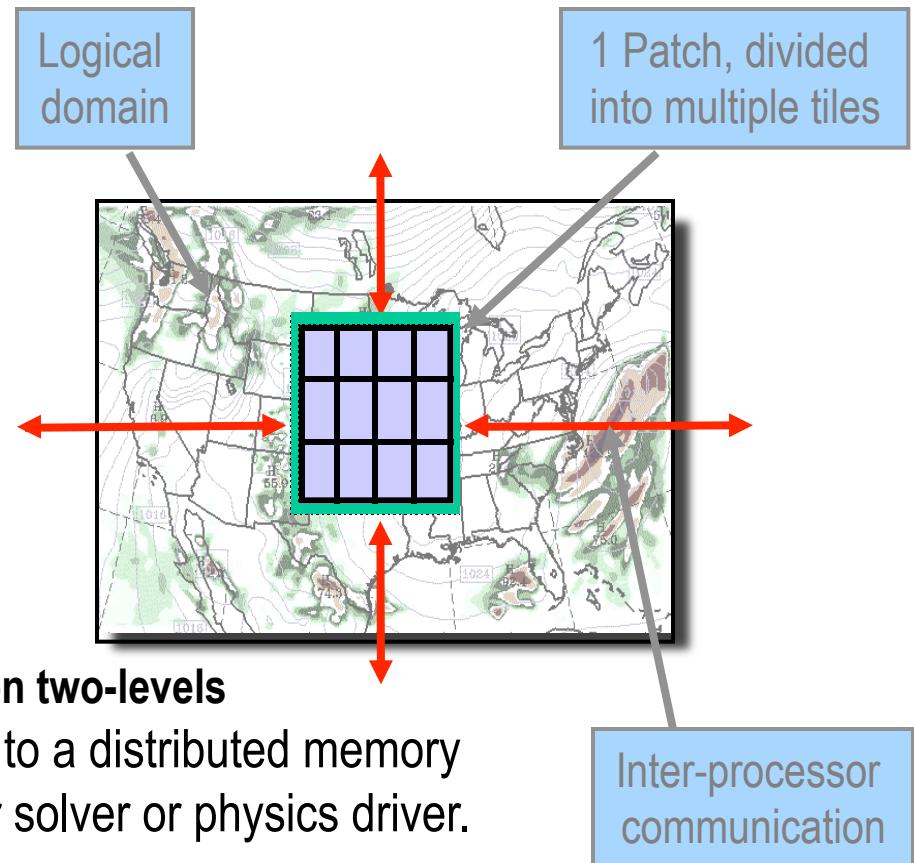
## Application: WRF

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



## 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.

Signs in  
code

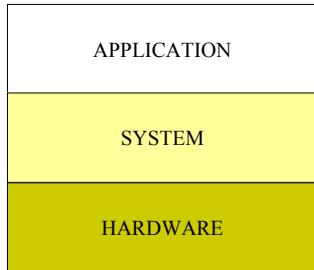
Following is an example code fragment that requires communication between patches

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.

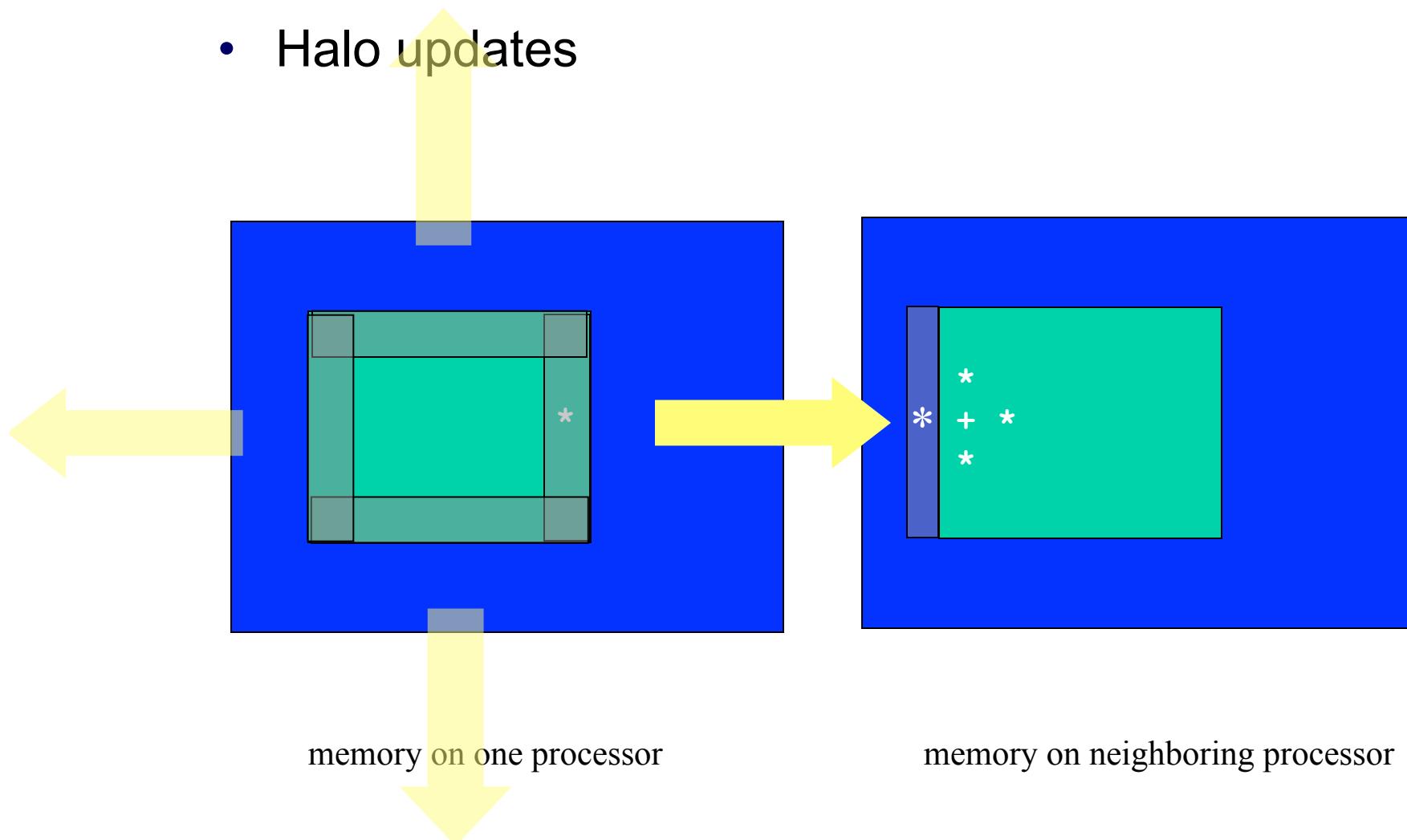
# Distributed Memory Communications

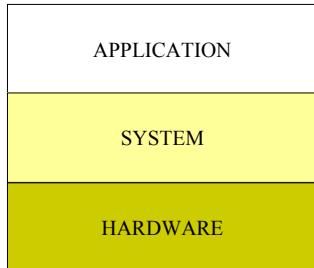
```
(module_diffusion.F )  
  
SUBROUTINE horizontal_diffusion_s (tendency, rr, var, . . .  
. . .  
    DO j = jts,jte  
    DO k = kts,ktf  
    DO i = its,ite  
        mrdx=msft(i,j)*rdx  
        mrdy=msft(i,j)*rdy  
        tendency(i,k,j)=tendency(i,k,j) -  
            (mrdx*0.5*((rr(i+1,k,j)+rr(i,k,j))*H1(i+1,k,j)-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

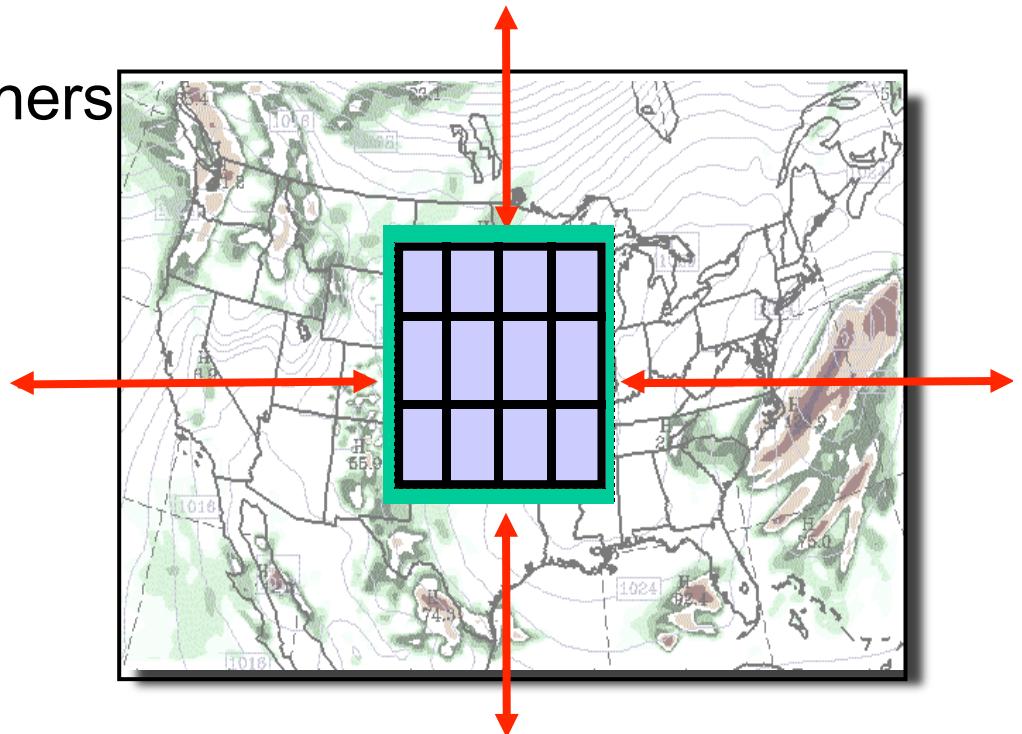
- Halo updates

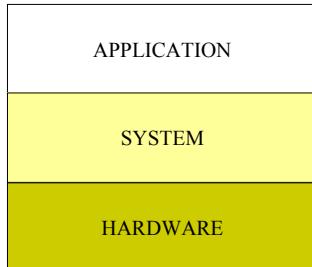




# Distributed Memory (MPI) Communications

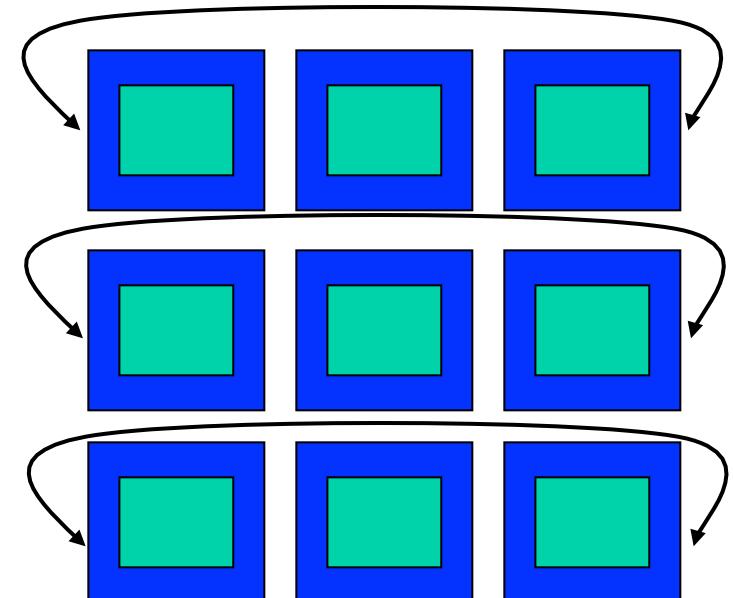
- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



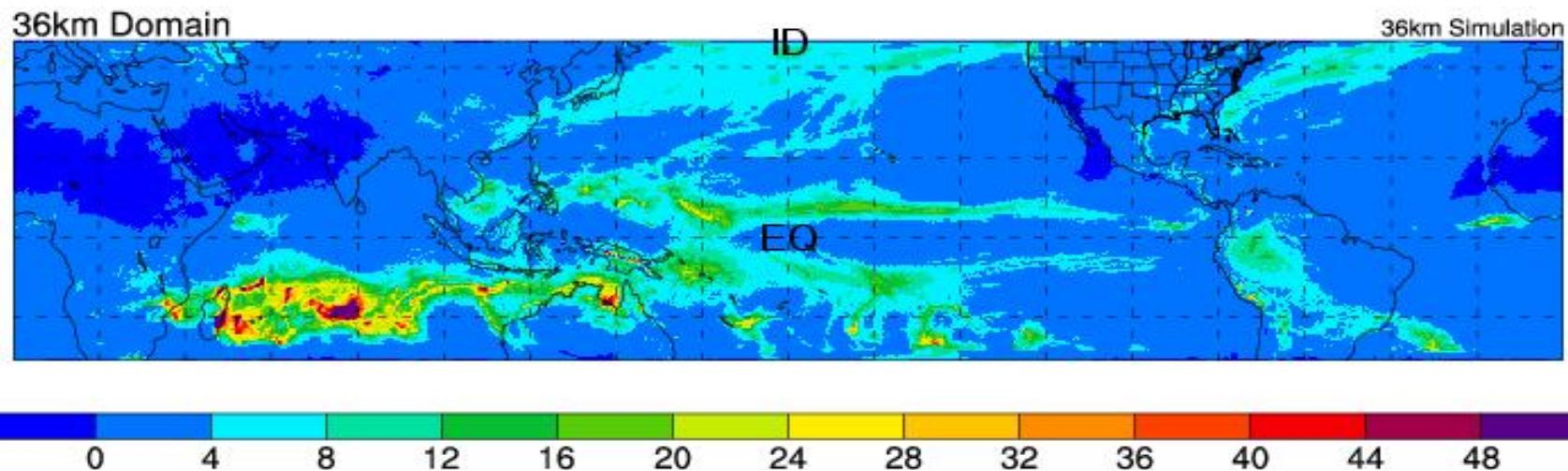


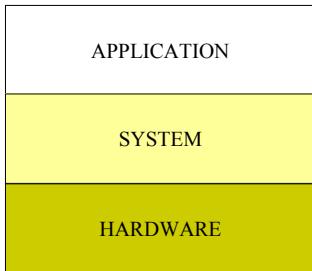
# Distributed Memory (MPI) Communications

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



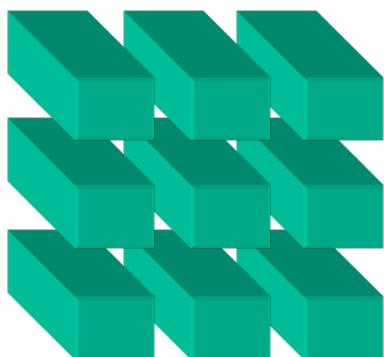
Average Daily Total rainfall (mm) - March 1997



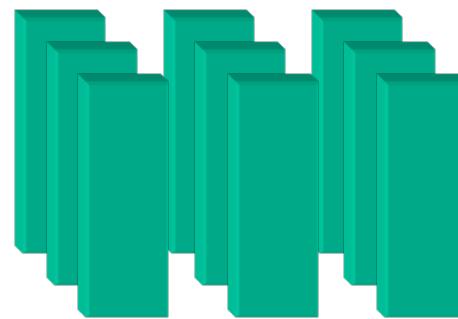


# Distributed Memory (MPI) Communications

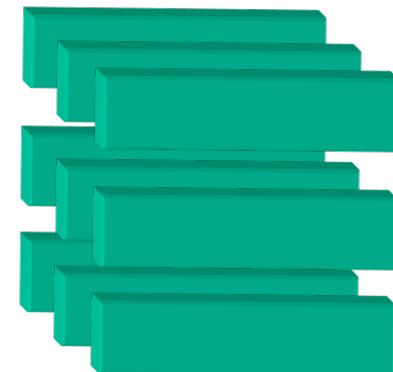
- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



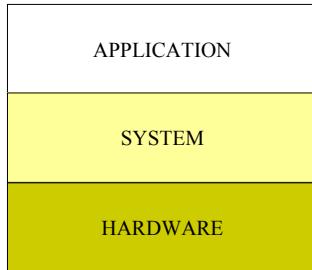
all y on  
patch



all z on  
patch

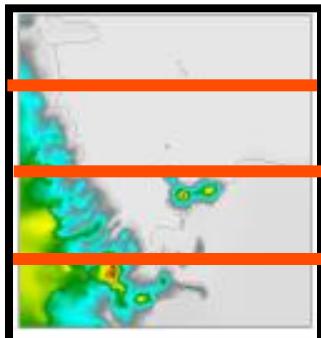


all x on  
patch

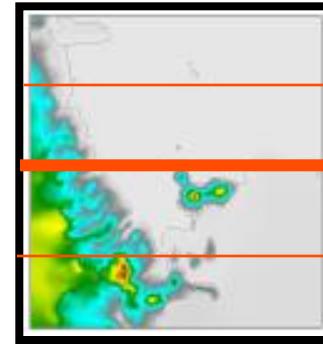


# Distributed Memory (MPI) Communications

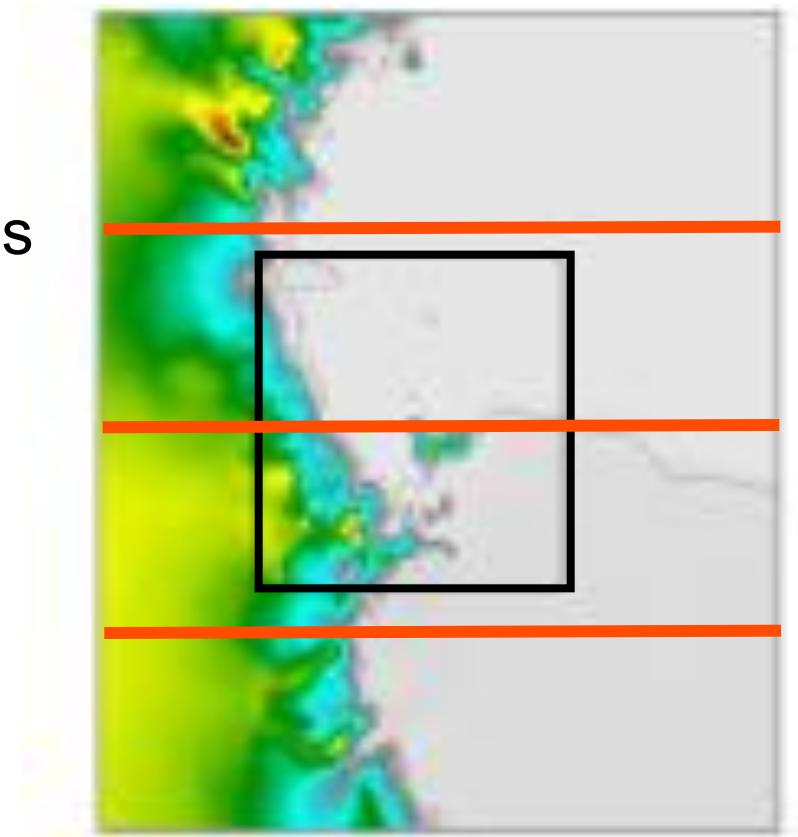
- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



NEST: 2.22 km

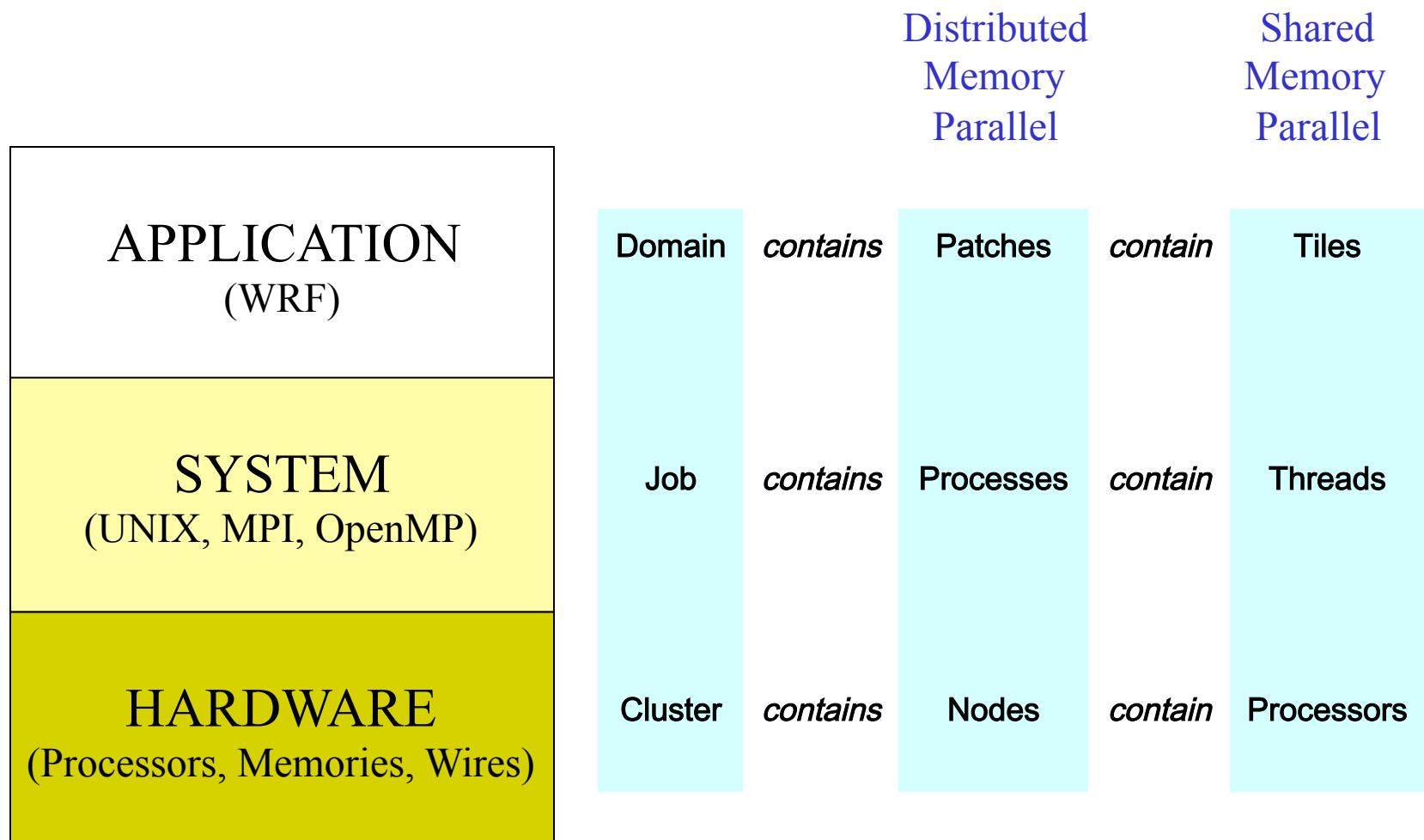


INTERMEDIATE: 6.66 km



COARSE  
Ross Island  
6.66 km

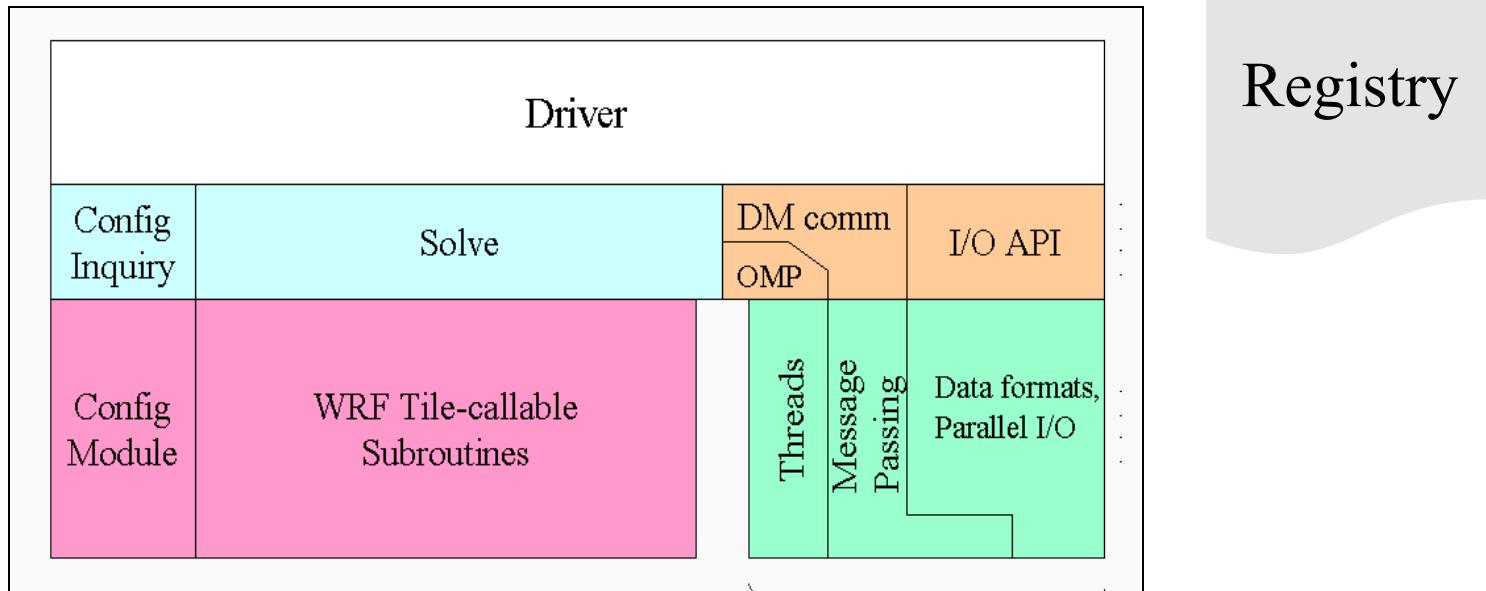
# Review – Computing Overview



# Outline

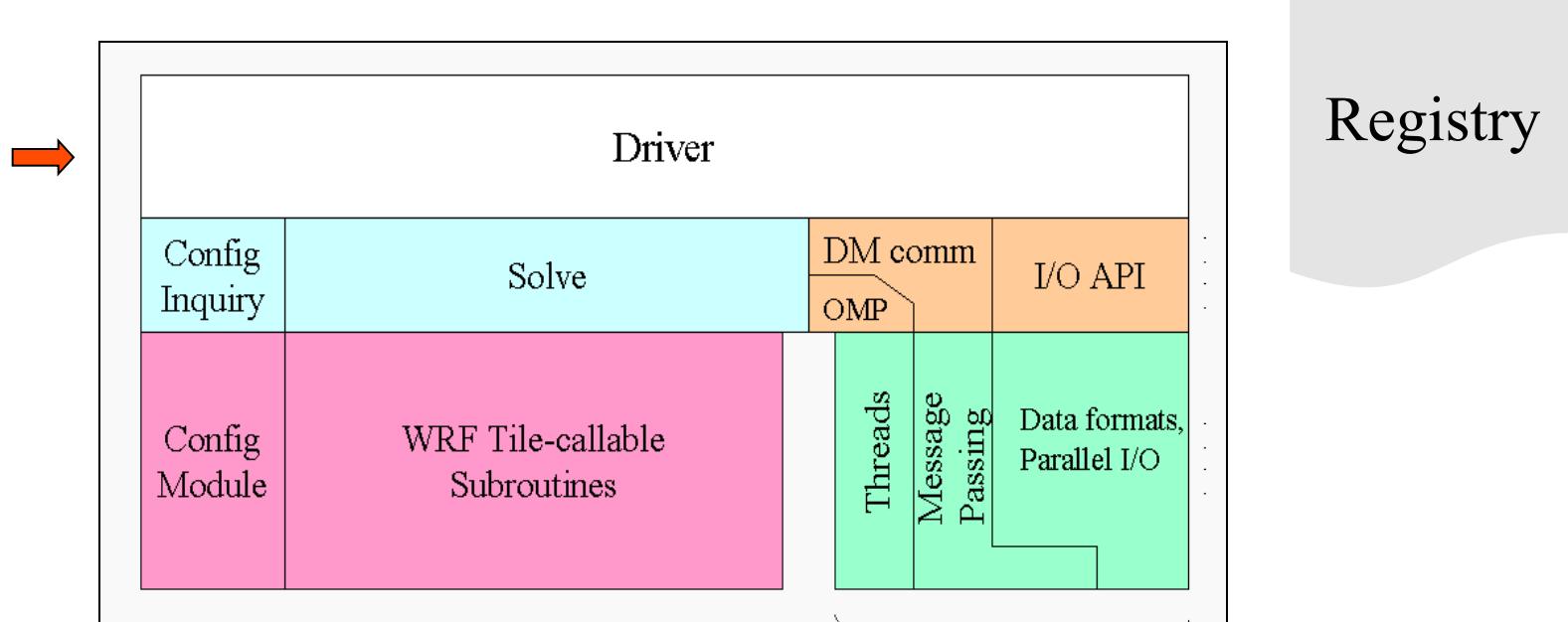
- Introduction
- Computing Overview
- WRF Software Overview

# WRF Software Architecture



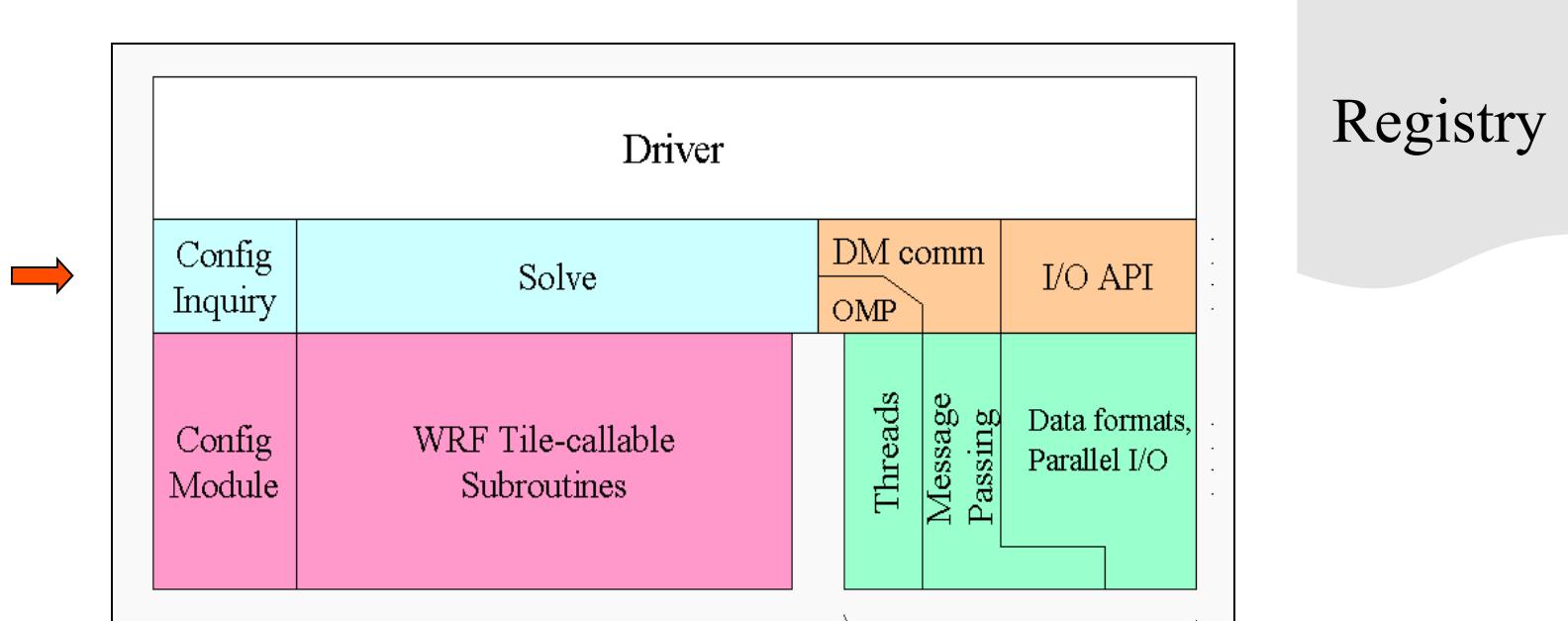
- Hierarchical software architecture
  - Insulate scientists' code from parallelism and other architecture/implementation-specific details
  - Well-defined interfaces between layers, and external packages for communications, I/O, and model coupling facilitates code reuse and exploiting of community infrastructure,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

# WRF Software Architecture



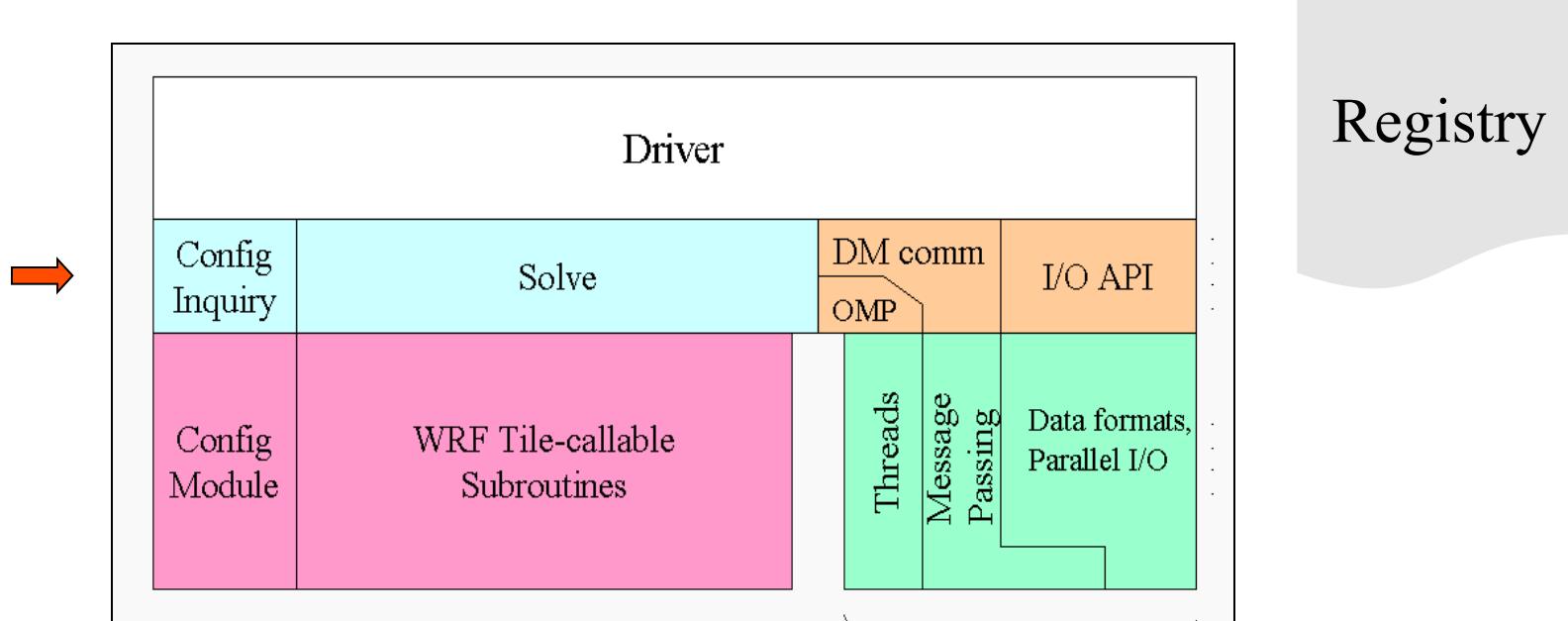
- **Driver** Layer
  - **Domains**: Allocates, stores, decomposes, represents abstractly as **single data objects**
  - **Time loop**: top level, algorithms for **integration over nest hierarchy**

# WRF Software Architecture



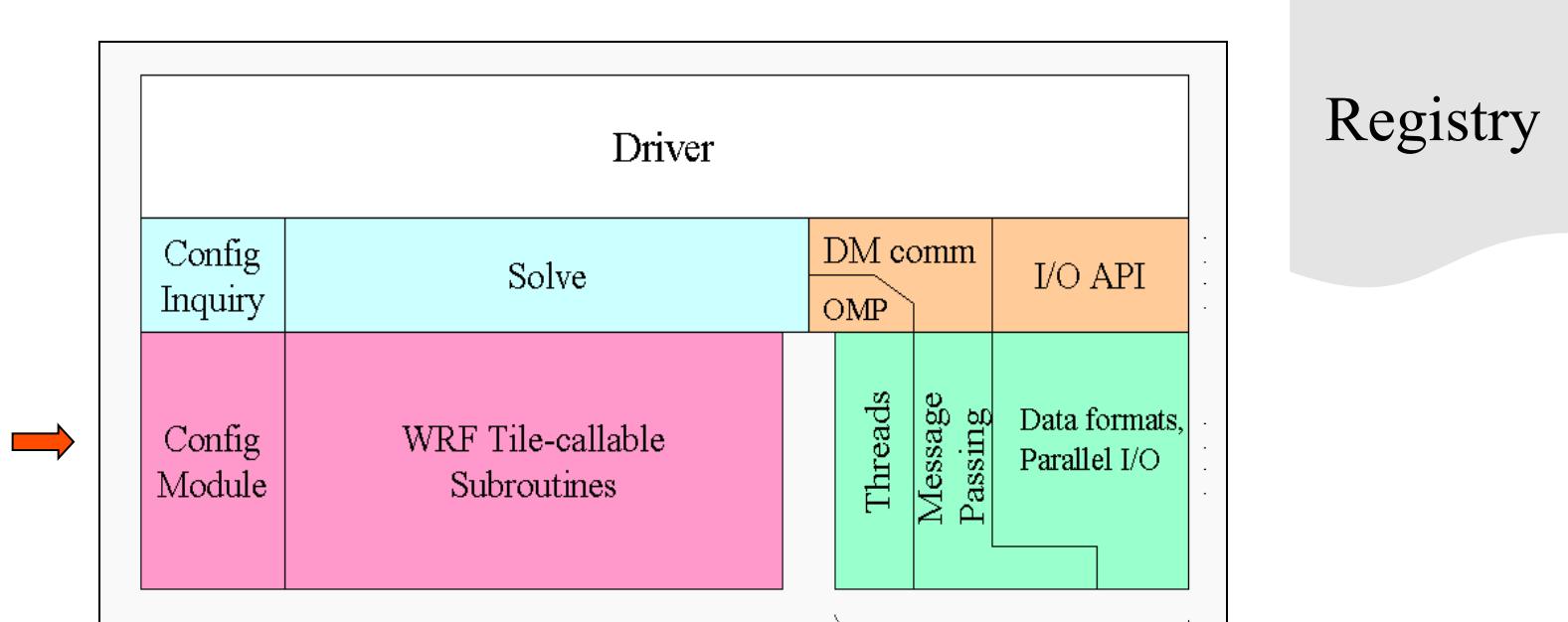
- Mediation Layer
  - Solve routine, takes a **domain object** and advances it **one time step**
  - Nest forcing, interpolation, and feedback routines

# WRF Software Architecture



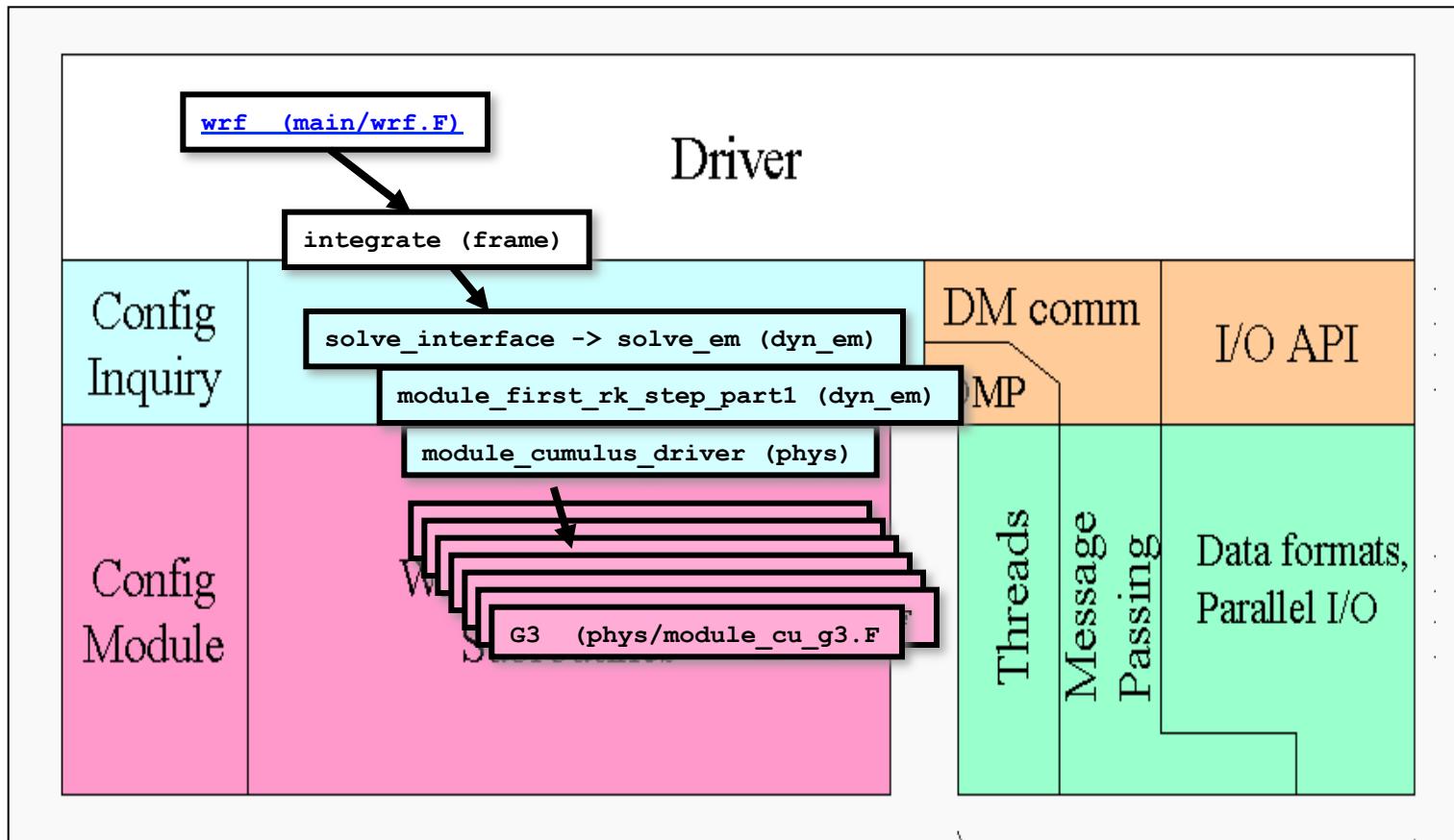
- Mediation Layer
  - The **sequence of calls** for doing a time-step for one domain is known in Solve routine
  - **Dereferences fields** in calls to physics drivers and dynamics code
  - Calls to **message-passing** are contained here as part of Solve routine

# WRF Software Architecture



- Model Layer
  - **Physics and Dynamics:** contains the actual WRF model routines are written to **perform some computation** over an arbitrarily sized/ shaped, 3d, rectangular subdomain

# Call Structure Superimposed on Architecture



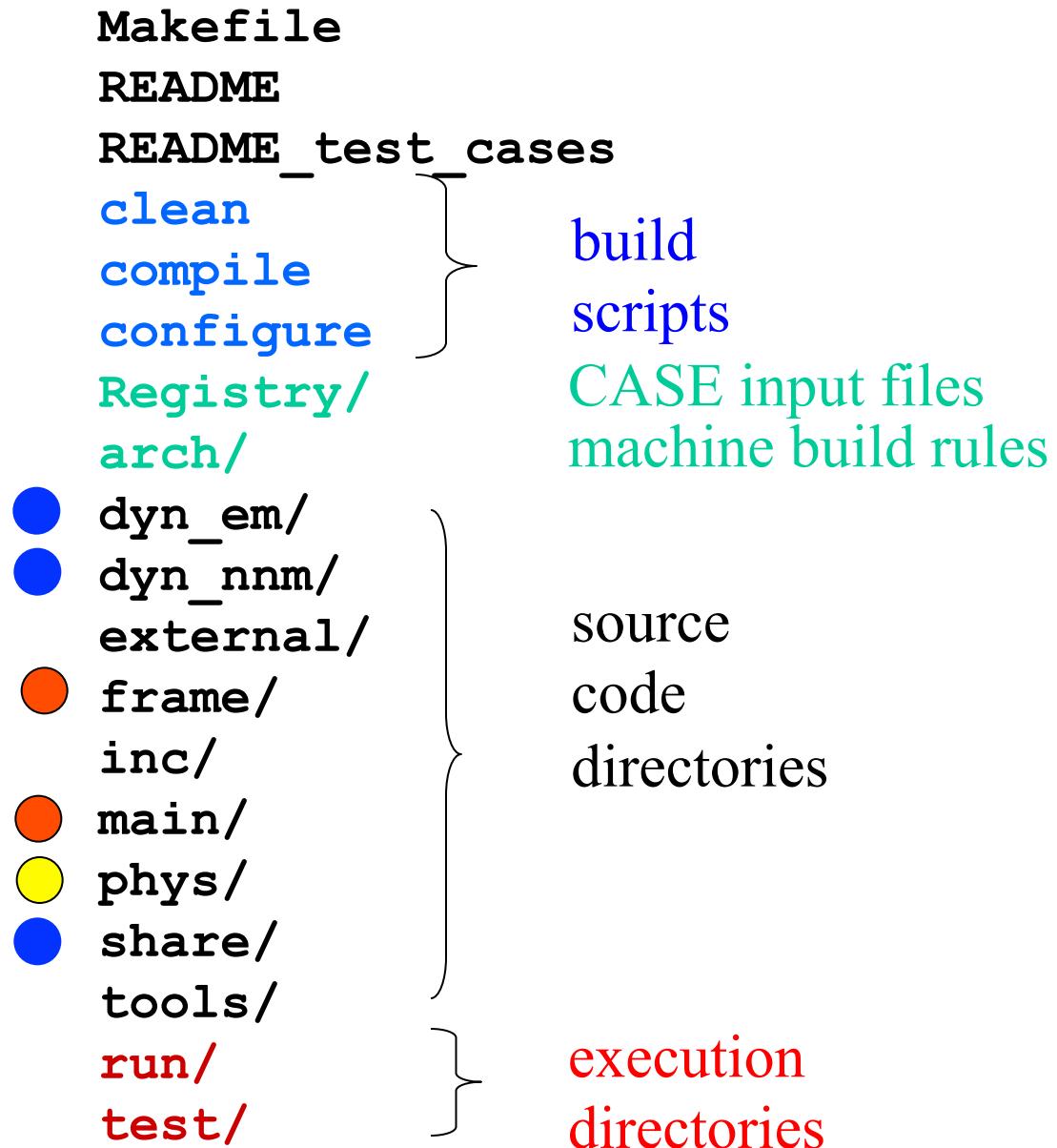
# WRF Software Overview

- 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        ●



# Where are WRF source code files located?

```
$ (RM) $@  
$(CPP) -I$(WRF_SRC_ROOT_DIR)/inc $(CPPFLAGS) $(OMPCPP) $*.F > $*.bb  
$(SED_FTN) $*.bb | $(CPP) > $*.f90  
$(RM) $*.b $*.bb  
@ if echo $(ARCHFLAGS) | $(FGREP) 'DVAR4D'; then \  
    echo COMPILING $*.F for 4DVAR ; \  
    $(WRF_SRC_ROOT_DIR)/var/build/da_name_space.pl $*.f90 > $*.f90.tmp ; \  
    mv $*.f90.tmp $*.f90 ; \  
fi  
  
if [ -n "$(OMP)" ] ; then echo COMPILING $*.F WITHOUT OMP ; fi ; \  
$(FC) -o $@ -c $(FCFLAGS) $(MODULE_DIRS) $(PROMOTION) $(FCSUFFIX) $*.f90 ; \  
$(RM) $*.bb
```

Where are WRF source code files located?

```
cpp -C -P file.F > file.f90
```

```
gfortran -c file.f90
```

# Where are WRF source code files located?

- The most important command is the “find” command. If there is an error in the model output, you can find that location in the source code with the **find** command.

```
cd WRFV3
```

```
find . -name *.F -exec grep -i "Flerchinger" {} \; -print
```

## Where are WRF source code files located?

- All of the differences between the .F and .f90 files are due to the included pieces that are manufactured by the Registry.
- These additional pieces are all located in the WRFV3/inc directory.
- For a serial build, almost 450 files are manufactured.
- Usually, most developers spend their time working with physics schemes.

# Where are WRF source code files located?

- The “main” routine that handles the calls to all of the physics and dynamics:
  - ARW: WRFV3/dyn\_em/solve\_em.F
  - NMM: WRFV3/dyn\_nmm/solve\_nmm.F
- This “solver” is where the tendencies are initialized to zero, some pre-physics terms are computed, and the time stepping occurs
- The calls to the physics schemes are made from a further call down the call tree
  - ARW: dyn\_em/module\_first\_rk\_step\_part1.F
  - NMM: dyn\_nmm/module\_PHYSICS\_CALLS.F

# Where are WRF source code files located?

- Inside of solve\_em and first\_rk\_step\_part1, all of the data is located in the “grid” structure: grid%ht.
- The dimensions in solve\_em and first\_rk\_step\_part1 are “d” (domain), and “m” (memory):
  - ids, ide, jds, jde, kds, kde
  - ims, ime, jms, jme, kms, kme
- The “t” (tile) dimensions are computed in first\_rk\_step\_part1 and passed to all drivers.
- WRF uses global indexing

# Where are WRF source code files located?

- If you are interested in looking at physics, the WRF system has organized the files in the WRFV3/phys directory.
- In WRFV3/phys, each type of physics has a driver:

module_cumulus_driver.F	cu
module_microphysics_driver.F	mp
module_pbl_driver.F	bl
module_radiation_driver.F	ra
module_surface_driver.F	sf

# Where are WRF source code files located?

- The subgrid-scale precipitation (\*\_cu\_\*.F)

module\_cu\_bmj.F

module\_cu\_camzm.F

module\_cu\_g3.F

module\_cu\_gd.F

module\_cu\_kf.F

module\_cu\_kfeta.F

module\_cu\_nsas.F

module\_cu\_osas.F

module\_cu\_sas.F

module\_cu\_tiedtke.F

# Where are WRF source code files located?

- Advection  
WRFV3/dyn\_em/module\_advect\_em.F
- Lateral boundary conditions  
WRFV3/dyn\_em/module\_bc\_em.F

## Where are WRF source code files located?

- Compute various RHS terms, pressure gradient, buoyancy, w damping, horizontal and vertical diffusion, Coriolis, curvature, Rayleigh damping  
WRFV3/dyn\_em/module\_big\_step\_utilities\_em.F
- All of the sound step utilities to advance u, v, mu, t, w within the small time-step loop  
WRFV3/dyn\_em/module\_small\_step\_em.F

# WRF Software Overview

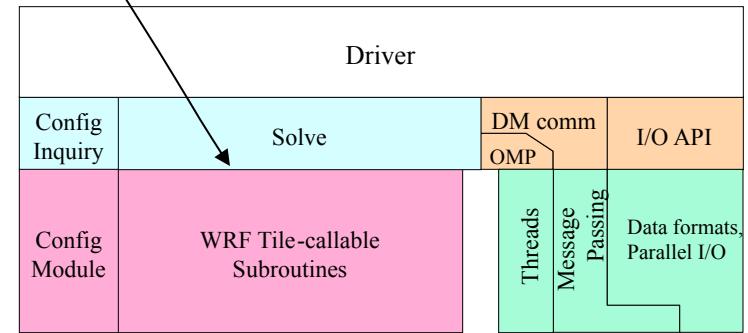
- Architecture
- Directory structure
- Model Layer Interface
- Data Structures
- I/O

## WRF Model Layer Interface – The Contract with Users

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

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

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



## WRF Model Layer Interface – The Contract with Users

### Restrictions on Model Layer subroutines:

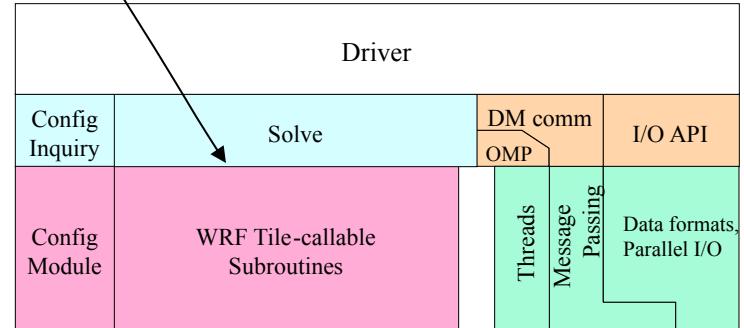
No I/O, communication

No stops or aborts

Use `wrf_error_fatal`

No common/module storage of  
decomposed data

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



## 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 = jts, MIN(jte,jde-1)
  DO k = kts, kte
    DO i = its, MIN(ite,idc-1)
      loc1(i,k,j) = arg1(i,k,j) + ...
    END DO
  END DO
END DO
```

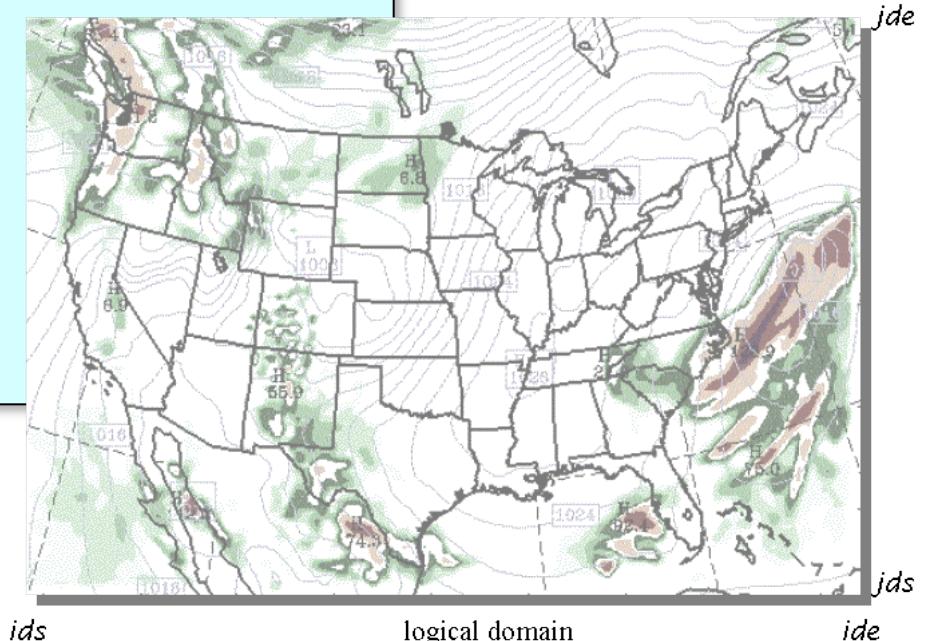
*template for model layer subroutine*

```
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
END DO
```

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



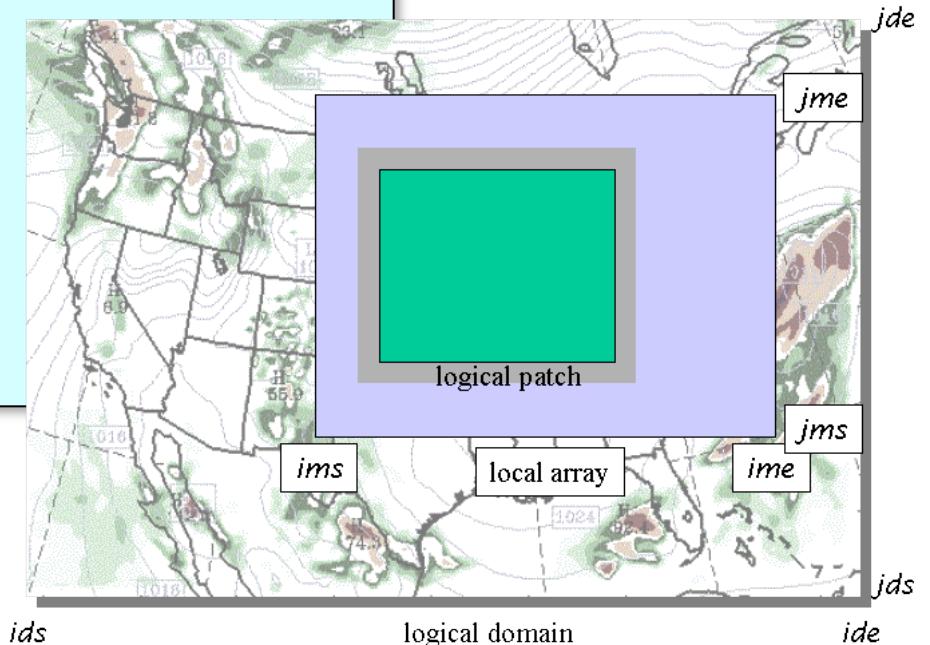
template for model layer subroutine

```
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
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



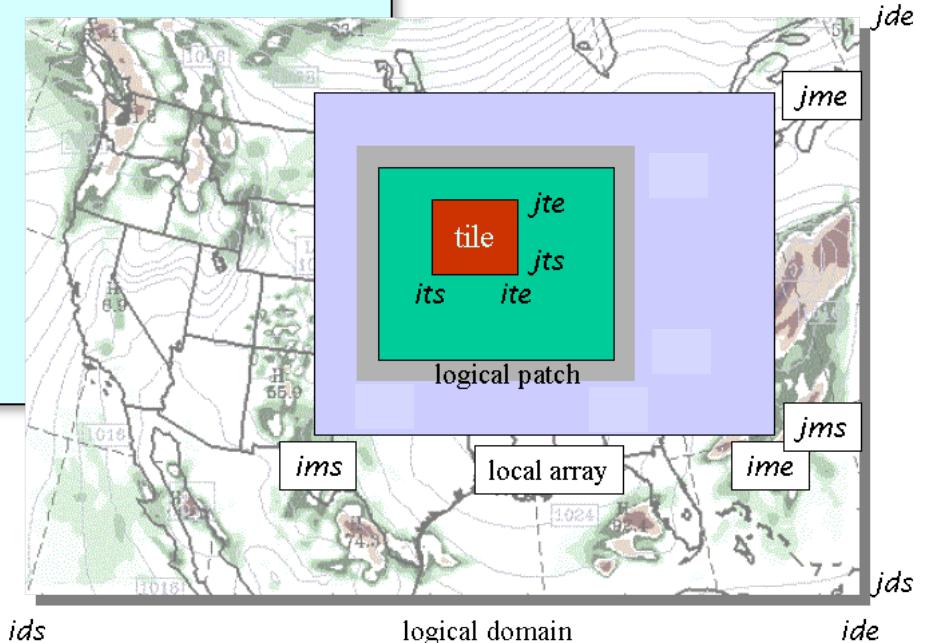
*template for model layer subroutine*

```
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

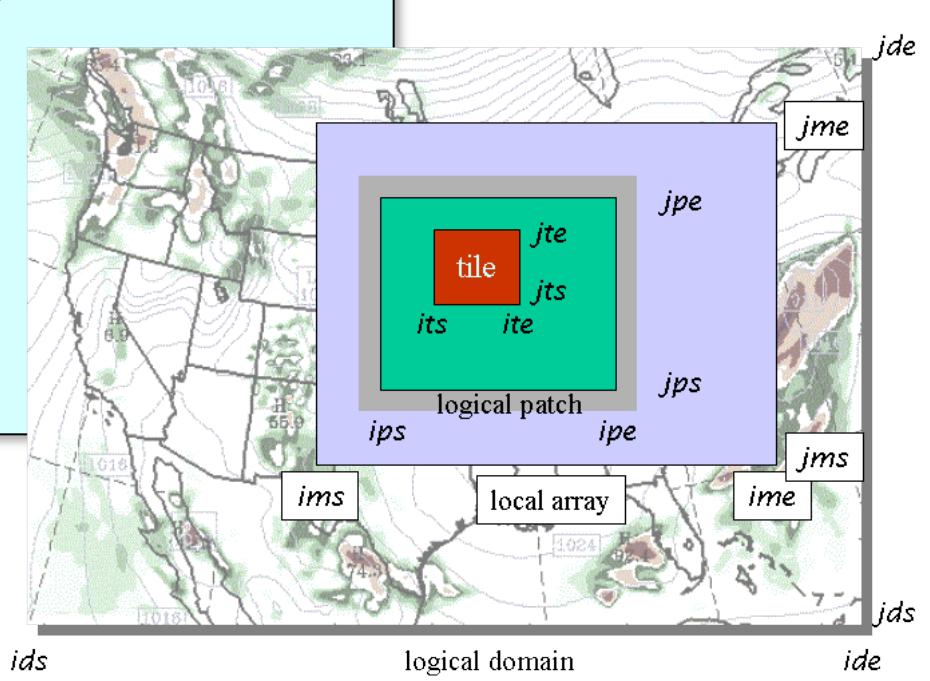
```
SUBROUTINE model ( &
    arg1, arg2, arg3, ... , argn,   &
    ids, ide, jds, jde, kds, kde, & ! Domain dims
    ims, ime, jms, jme, kms, kme, & ! Memory dims
    its, ite, jts, jte, kts, kte ) ! Tile dims

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
END DO
```

- 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

- 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

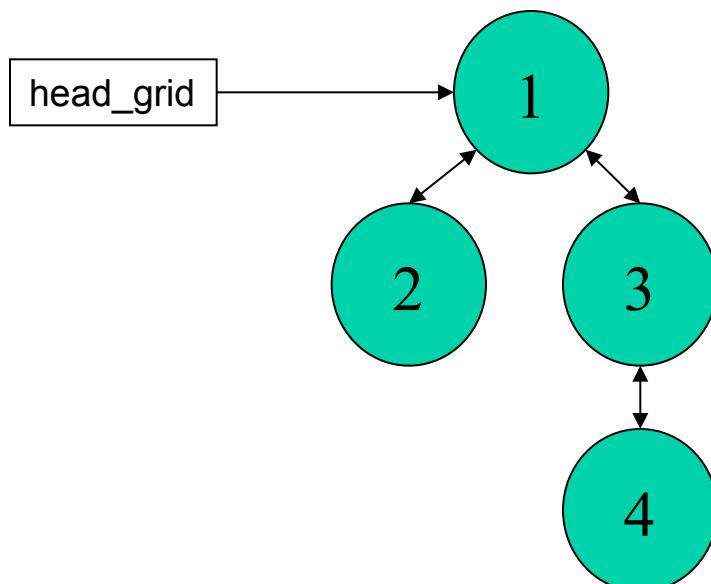


# WRF Software Overview

- 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)



- Every Registry defined **state**, **I1**, and **namelist** variable is contained inside the DDT (locally known as a **grid** of type **domain**), where each node in the tree represents a separate and complete 3D model domain/nest.

## 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)

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)on the stack
- Defined in  
the physics  
subroutines  
on the  
stack

# 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!  
Common blocks must not leave the Module.

## 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 at the end of every time step
  - Are not used to impact I1 variables on a child domain

## Model Layer Data Structures: I2 Data

- Persist for the duration of a call of the physics routine
- NOT contained within the DDT structure (no declarations in the Registry)
  - Memory for I2 arrays are dynamically allocated on subroutine entry, and automatically deallocated on exit
  - Local, intermediate dummy variables required for physics computations
  - If I2 arrays, then they are always **tile** dimensioned
  - Not declared in the Registry, not communicated, no IO, not passed back to the solver, do not exist (retain their previous value) between successive physics calls

# 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 (ARW) and IJK (NMM) but these are 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 (also referred to as the ARW “mass points”), 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
- Can be thought of as files, though that is a restriction
  - History + auxiliary output streams (10 and 11 are reserved for nudging)
  - Input + auxiliary input streams (10 and 11 are reserved for nudging)
  - Restart, boundary, and a special DA in-out stream
  - Currently, 24 total streams
  - Use the large values and work down to stay away from “used”

# 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**)

# WRF I/O

- Attributes of streams
  - Mandatory for stream to be used:
    - Time interval between I/O operations on stream
    - Format: `io_form`

# 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
  - I/O