Tutorial Notes: WRF Software 2.1

John Michalakes, NCAR

WRF Software Architecture Working Group

Outline

- Introduction
- Computing Overview
- Software Overview
- Data Structures
- Registry
- I/O & Nesting
- Examples

Introduction

- Intended audience for this tutorial session:
  - Primarily scientific users and others who wish to:
    - Work with the code
    - Extend/modify the code to enable their work/research
    - Address problems as they arise
    - Adapt the code to take advantage of local computing resources
  - Also: developers, computer scientists and software engineers, computer vendors
    - Developing new functionality (e.g. moving nests, coupling)
    - Integration with frameworks and other community infrastructure
    - Porting and benchmarking new platforms

Resources

- WRF project home page
  - http://www.wrf-model.org
- WRF users page (linked from above)
  - http://www.mmm.ucar.edu/wrf/users
- On line documentation (also from above)
  - http://www.mmm.ucar.edu/wrf/WG2/software_v2
- WRF users help desk
  - wrfhelp@ucar.edu

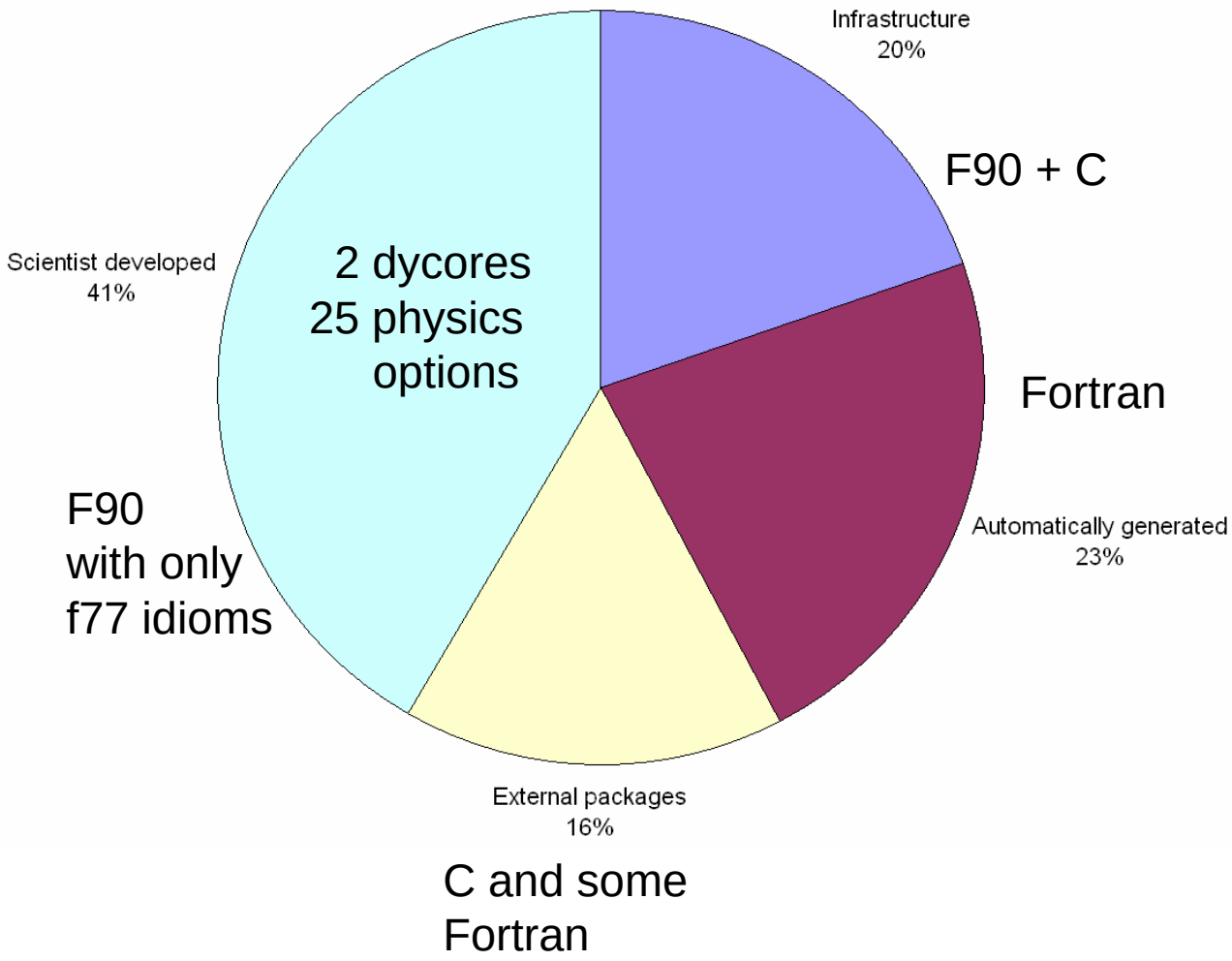
Introduction

- Characteristics of WRF Software
  - Developed from scratch beginning around 1998
  - Requirements emphasize flexibility over a range of platforms, applications, users; performance
  - WRF develops rapidly. First released Dec 2000; Current Release WRF v2.1
  - Current source code…

Introduction

WRF SOURCE CODE

- 250-thousand lines total
- 50-thousand lines of infrastructure leverages:
  - 100-thousand lines of contributed scientist-developed code; amount is increasing
  - At least 40-thousand lines of "external" software
- 60-thousand lines are automatically generated at compile time



Introduction

- Supported Platforms (alphabetical)

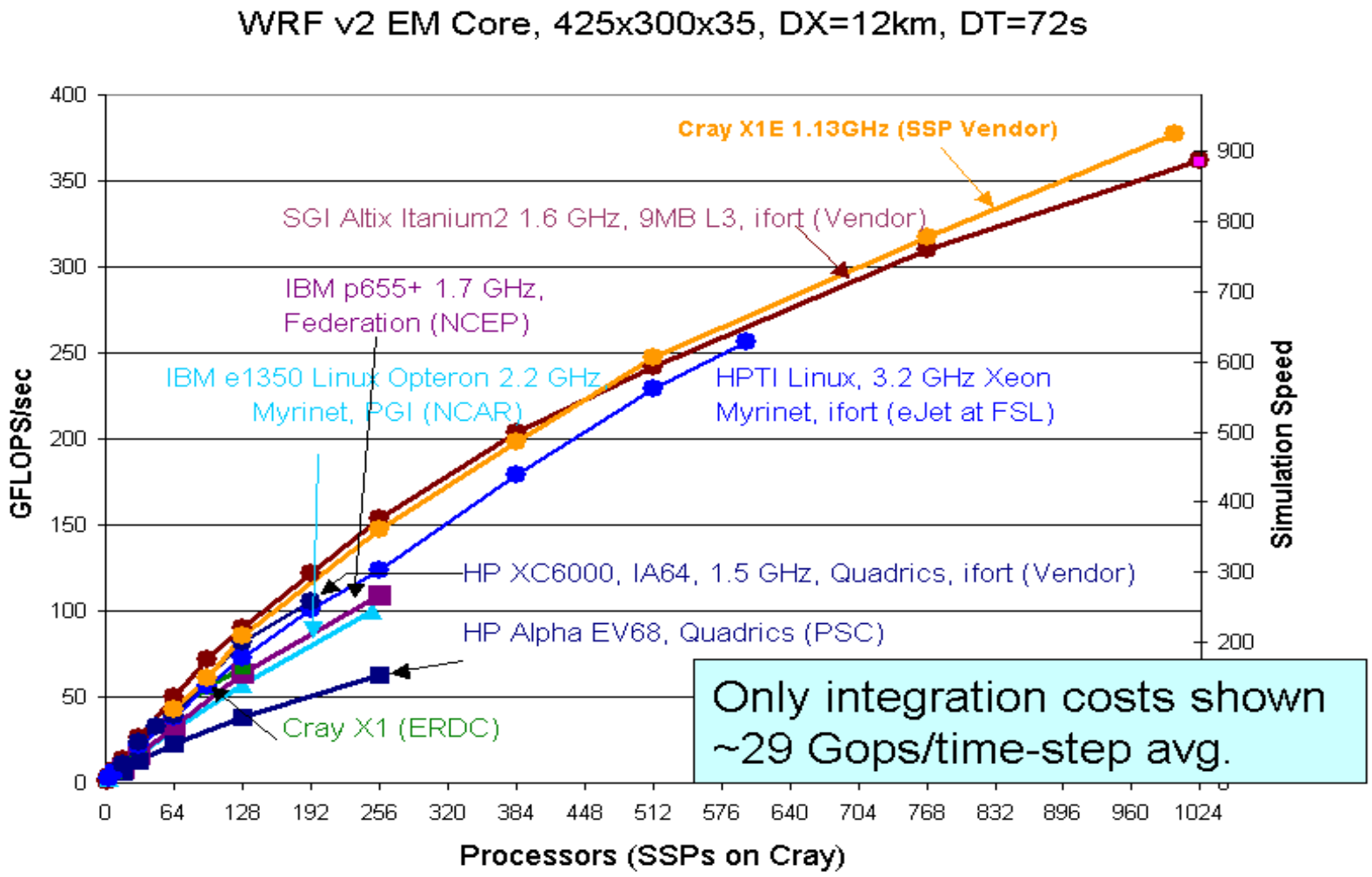
Vendor	Hardware	OS	Compiler
Apple (*)	G5	MacOS	IBM
Cray Inc.	X1	UNICOS	Cray
HP/Compaq	Alpha	Tru64	Compaq
	Itanium-2	Linux	Intel
		HPUX	HP
IBM	SP Power-3/4	AIX	IBM
SGI	Itanium-2	Linux	Intel
	MIPS	IRIX	SGI
Sun (*)	UltraSPARC	Solaris	Sun
various	Xeon and Athlon	Linux	Intel and Portland Group
	Itanium-2 and Opteron		

(*) dm-parallel not supported yet

WSF Enhancements for V2.1 (Highlights)

- Inclusion of NMM-Core in source distribution (with Tom Black, S. Gopal, NCEP)
- WRF DA-VAR and Model version synchronization
- GRIB 1 (Todd Hutchinson, WSI)
- Generalized physics interface (with Sue Chen et al. NRL-MRY)
- Nest init option similar to MM5's IOVERW=2 (for AFWA)
- Comprehensive regression testing for quality control
- Performance and Efficiency
- ESMF Integration
- Nesting and Moving Nests

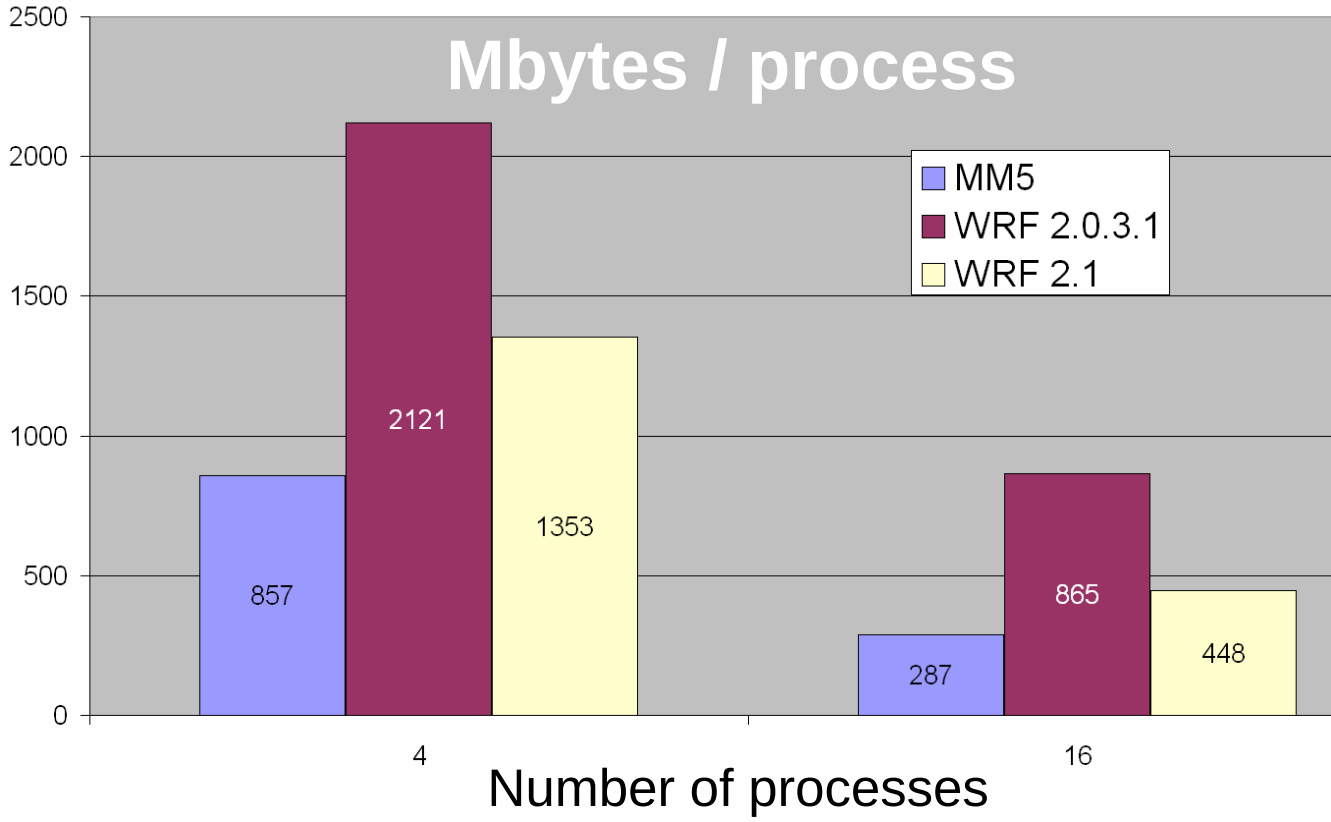
Performance (v2.0.x)



- New v2.1 based standardized benchmark cases will be released in coming weeks, with release of WRF v2.1

Improved Memory Utilization in v2.1

- Reduced temporary data for nesting
- Removed 2nd time-level for tracer arrays
- Smaller, lighter-weight comm package RSL LITE



2 domain configuration
425 x 300 x 35 each

ESMF Integration

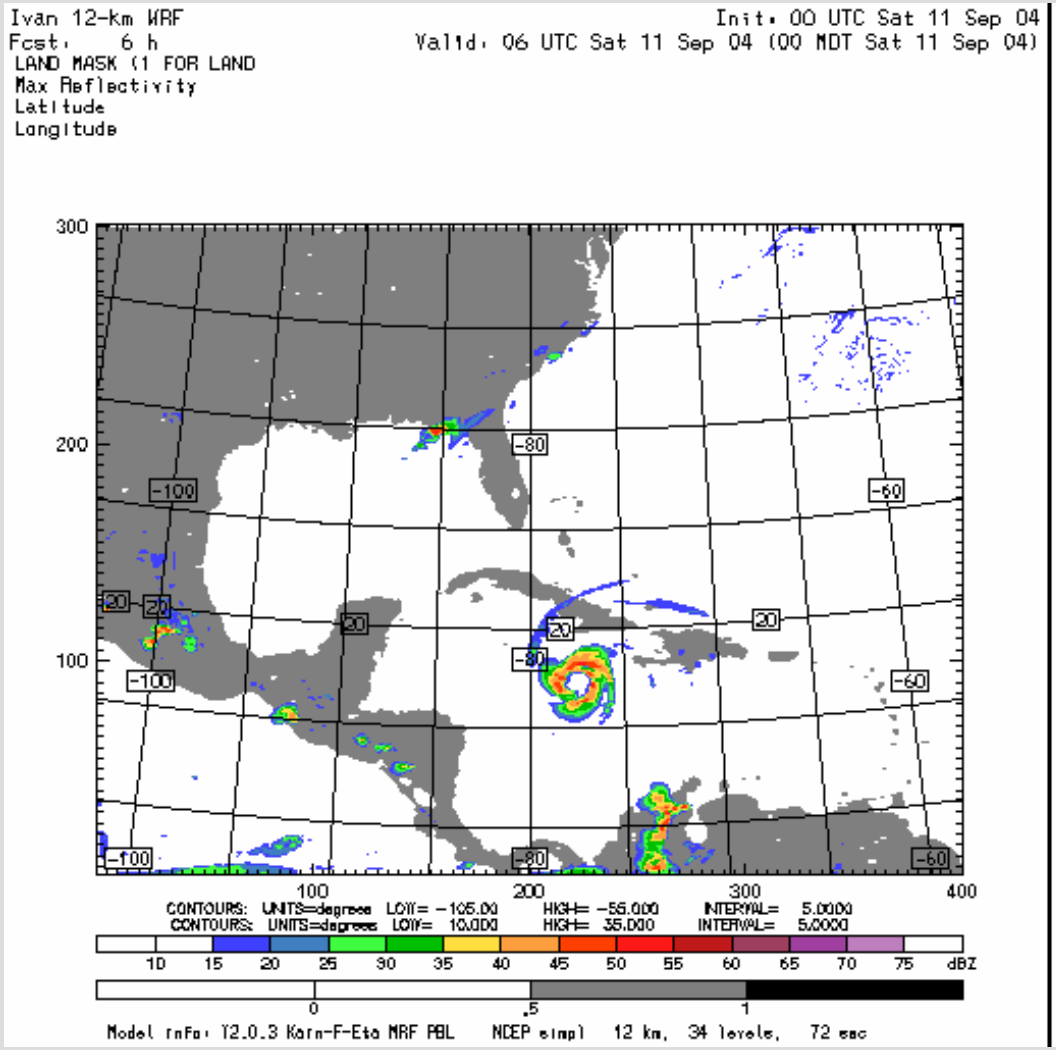
- WRF as an ESMF component model
  - WRF v2.1 can operate as an ESMF component
  - Full coupling functionality through ESMF in-progress (initial target: HyCOM)
- Also:
  - ESMF Time Management Utility
  - ESMF Error Logging Utility
  - WRF I/O has been adopted in ESMF
  - Participating in CF metadata convention standardization
- ESMF can be used with v2.1 (but is not required)

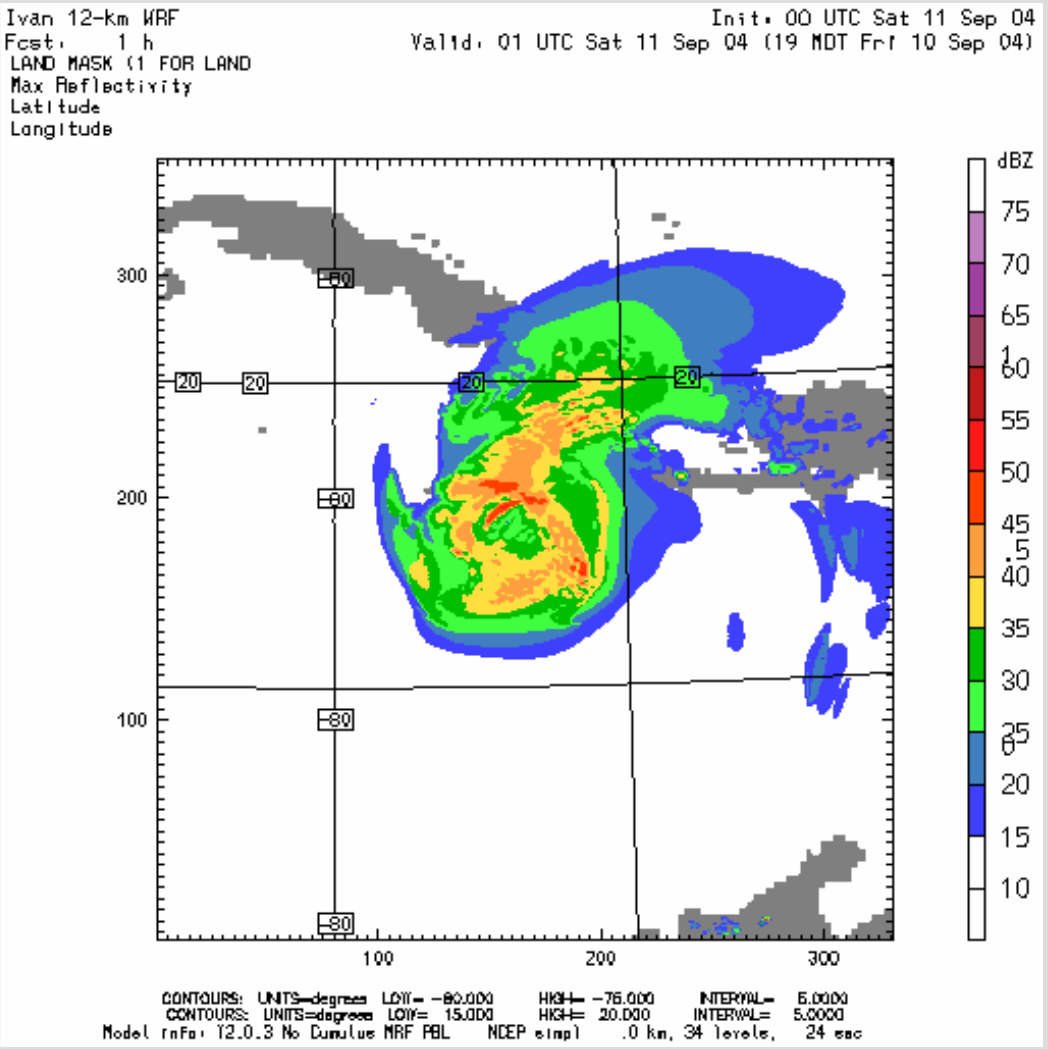
Five-day Hurricane Ivan 12km/4km Moving Nest

Two-way interacting nest with high-resolution terrain ingest at leading edge

400 x 301 x 35, dt = 72 sec

331 x 352 x 35, dt = 24 sec





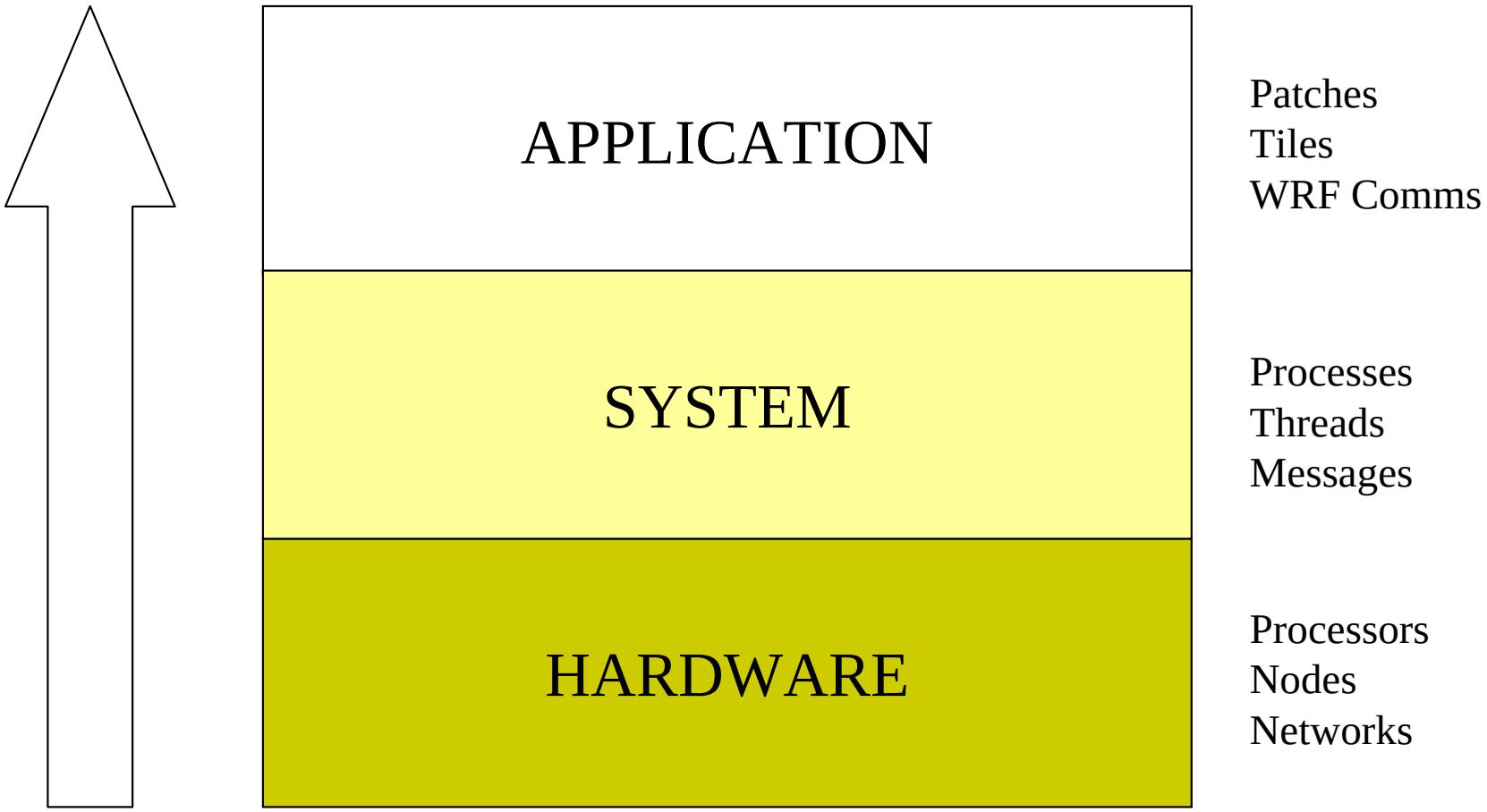
Run time: 8.6 hours on 64p IBM Power 4

Some terms

- WRF Architecture – scheme of software layers and interface definitions
- WRF Framework – the software infrastructure, also "driver layer" in the WRF architecture
- WRF Model Layer – the computational routines that are specifically WRF
- WRF Model – a realization of the WRF architecture comprising the WRF model layer with framework
- WRF – a set of WRF architecture-compliant applications, of which the WRF Model is one

Computing Overview

Computing 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
  - 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

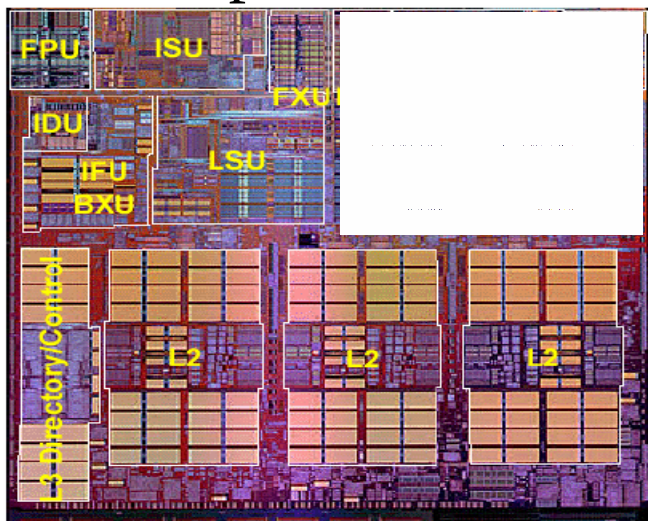
~144 Kbytes

~50,000 flop/s

48hr 12km WRF CONUS in 600 years

A computer in 2002

IBM p690



4-way superscalar

64-bit floating point precision

1.4 Mbytes (shown)

> 500 Mbytes (not shown)

~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
  - > 4,000x 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 $10^{11\text{-}12}$ flop/second
  - Aggregate available memories of $10^{11\text{-}12}$ bytes

~100,000,000,000 flop/s
48 12km WRF CONUS in under 15 minutes

Parallel computing terms -- hardware

- **Processor**:
  - A device that reads and executes instructions in sequence to produce perform operations on data that it gets from a memory device producing results that are stored back onto the memory device
- **Node**: One memory device connected to one or more processors.
  - Multiple processors in a node are said to share-memory and this is "shared memory parallelism"
  - They can work together because they can see each other's memory
  - The latency and bandwidth to memory affect performance
- **Cluster**: Multiple nodes connected by a network
  - The processors attached to the memory in one node can not see the memory for processors on another node
  - For processors on different nodes to work together they must send messages between the nodes. This is "distributed memory parallelism"
- **Network:**
  - Devices and wires for sending messages between nodes
  - Bandwidth – a measure of the number of bytes that can be moved in a second
  - Latency – the amount of time it takes before the first byte of a message arrives at its destination

Parallel Computing Terms -- Software

"The only thing one does directly with hardware is pay for it."

- Process:
  - A set of instructions to be executed on a processor
  - Enough state information to allow process execution to stop on a processor and be picked up again later, possibly by another processor
- Processes may be lightweight or heavyweight
  - Lightweight processes, e.g. shared-memory threads, store very little state; just enough to stop and then start the process
  - Heavyweight processes, e.g. UNIX processes, store a lot more (basically the memory image of the job)

Parallel Computing Terms -- Software

- Every job has at least one heavy-weight *process*.
  - A job with more than one process is a distributed-memory parallel job
  - Even on the same node, heavyweight processes do not share memory†
- Within a heavyweight process you may have some number of lightweight processes, called *threads.*
  - Threads are shared-memory parallel; only threads in the same memory space can work together.
  - A thread never exists by itself; it is always inside a heavy-weight process.
- Processes (heavy-weight) are the vehicles for distributed memory parallelism
- Threads are the vehicles for shared-memory parallelism

Jobs, Processes, and Hardware

- MPI is used to start up and pass messages between multiple heavyweight processes
  - The **mpirun** command controls the number of processes and how they are mapped onto nodes of the parallel machine
  - Calls to MPI routines send and receive messages and control other interactions between processes
  - http://www.mcs.anl.gov/mpi
- OpenMP is used to start up and control threads within each process
  - Directives specify which parts of the program are multi-threaded
  - **OpenMP** environment variables determine the number of threads in each process
  - http://www.openmp.org
- The number of processes (number of MPI processes times the number of threads in each process) usually corresponds to the number of processors

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?
  - 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 (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 4
mpirun –np 32
```

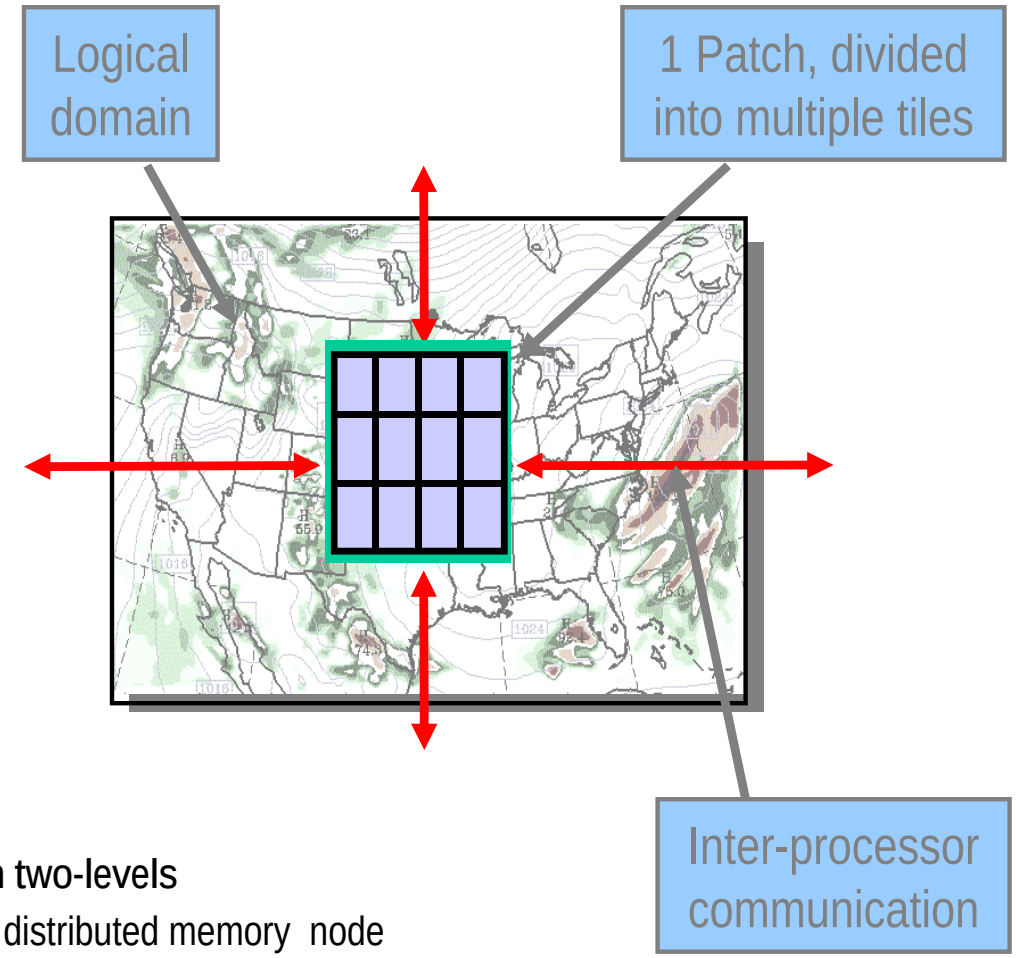
Other information about Parallel Processes

- Memory limits for heavy-weight processes
  - A process doesn't get all the memory and running out is ugly
    - Often appears as a segmentation violation in some otherwise correct-looking part of the program
    - Soft limits on per-process memory controlled by the **limit** and **unlimit** commands
    - Hard limits are set in the operating system; need administrator to change
  - Virtual memory
    - Even when you're not running out of memory, you may be running out of physical memory
    - Program will still run but it will be many times slower
  - Make sure that **mpirun** is distributing processes evenly over the nodes in your partition. You may need to use the –**machinefile** or other options
  - Some versions of MPI have buffer size limits
- Memory limits for light-weight processes
  - Thread-private stack size is usually limited and running out is uglier
  - May be enlarged; for example, the MPSTKZ environment variable with the Portland Group compilers

Application: WRF

- WRF uses *domain decomposition* to divide total amount of work over parallel processes
- Since the process model has two levels, the decomposition 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
  - 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

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

Example code fragment that requires communication between patches

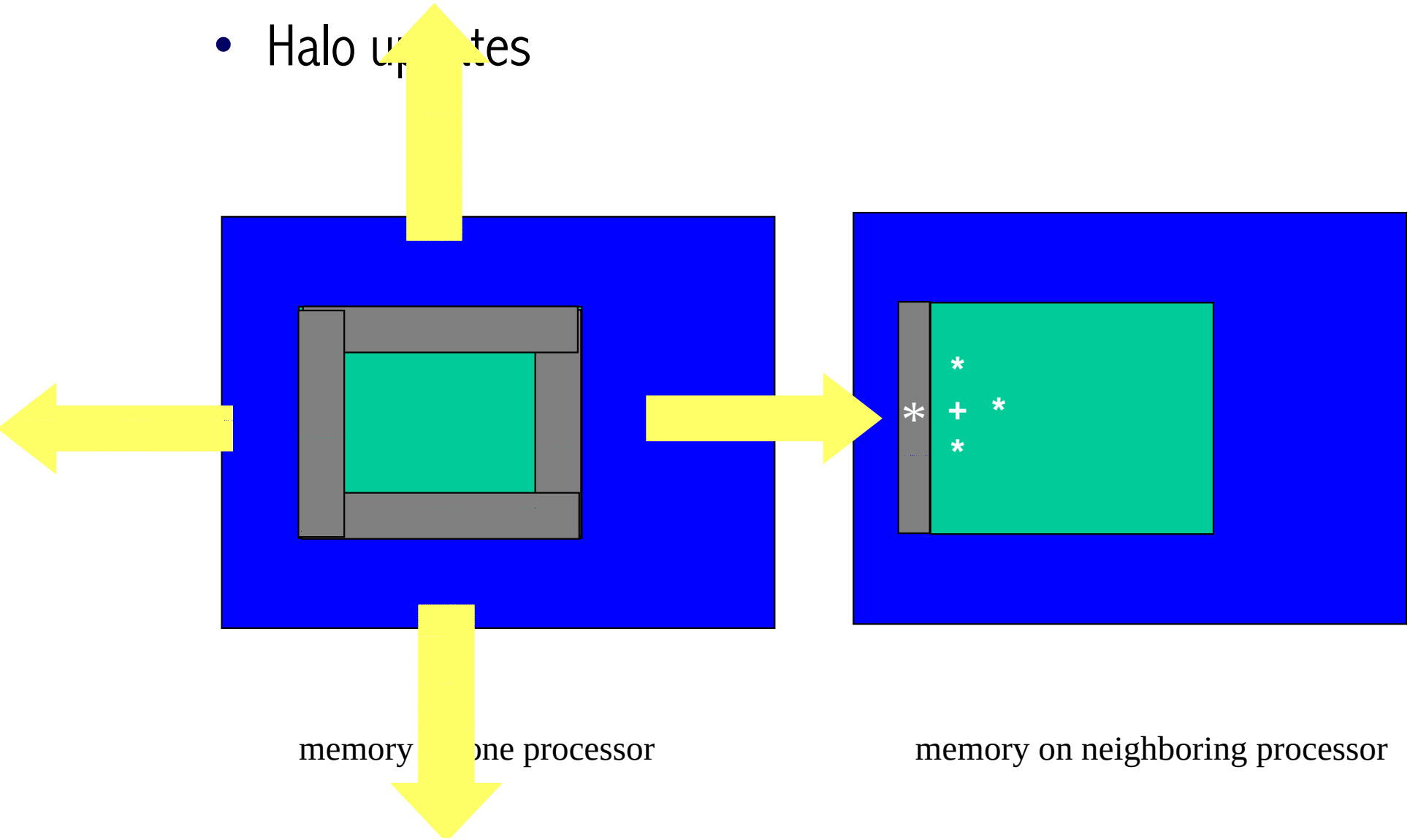
Note the tell-tale +1 and –1 expressions in indices for **rr**, **H1,** and **H2** arrays on right-hand side of assignment. These are *horizontal data dependencies* because the indexed operands may lie in the patch of a neighboring processor. That neighbor's updates to that element of the array won't be seen on this processor. We have to communicate.

(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
 . . .
```

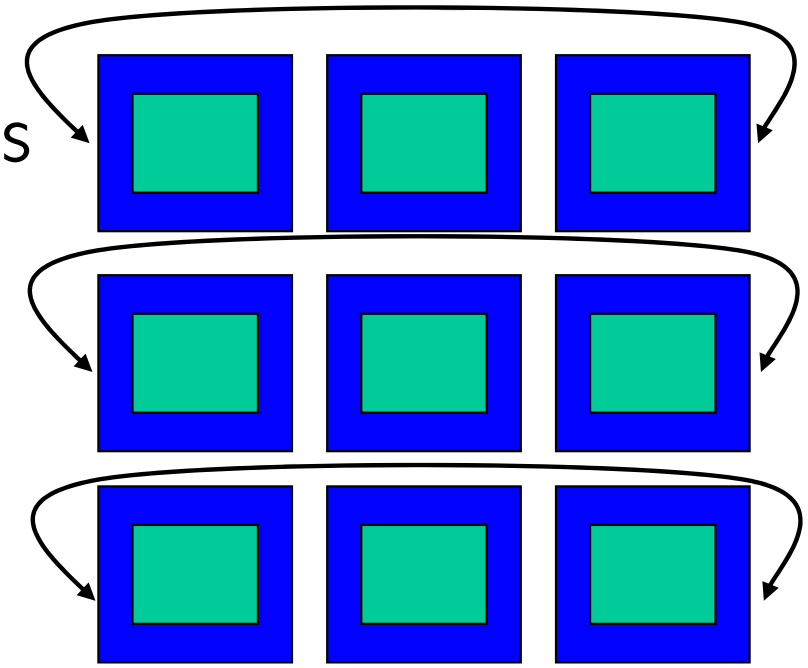
Distributed Memory MPI Communications

- Halo updates



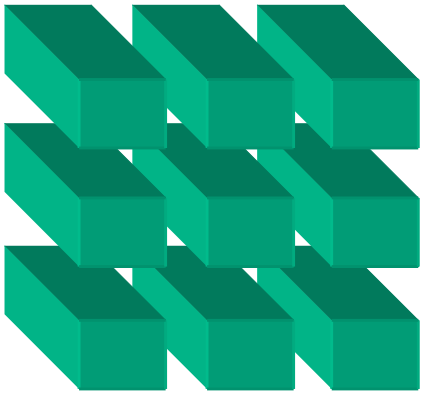
Distributed Memory (MPI) Communications

- Halo updates
- Periodic boundary updates

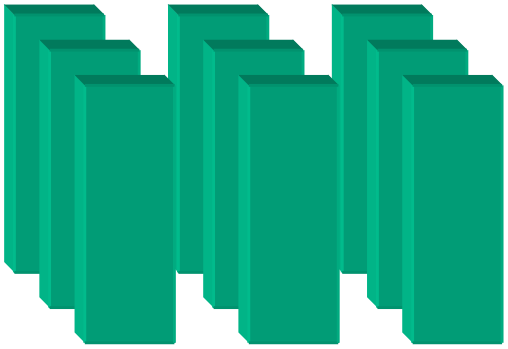


Distributed Memory (MPI) Communications

- Halo updates
- Periodic boundary updates
- Parallel transposes



all z on patch



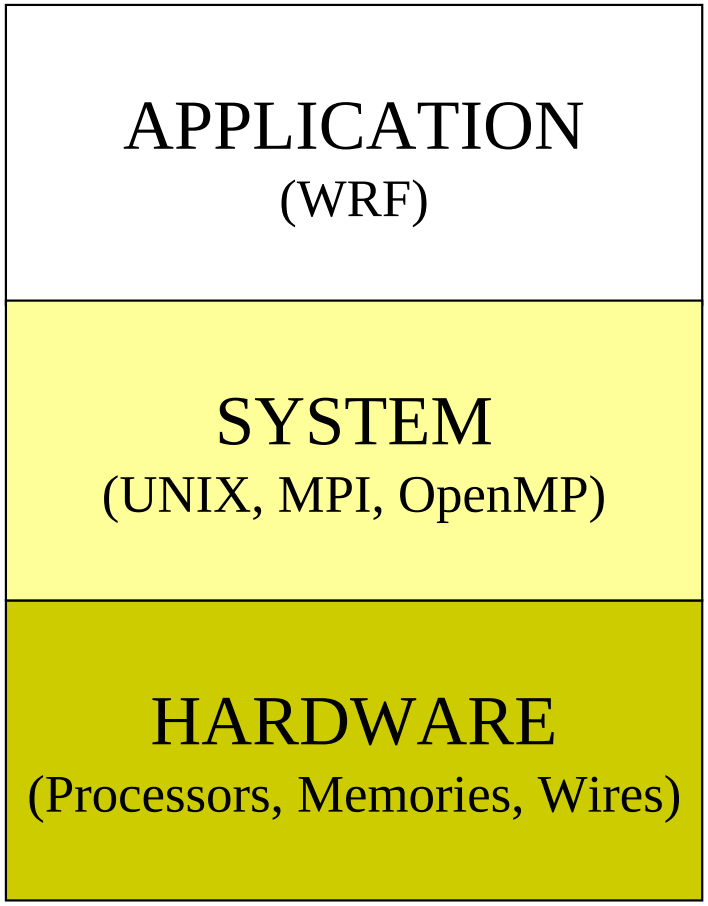
all y on patch

all x on patch

Distributed Memory (MPI) Communications

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

Review



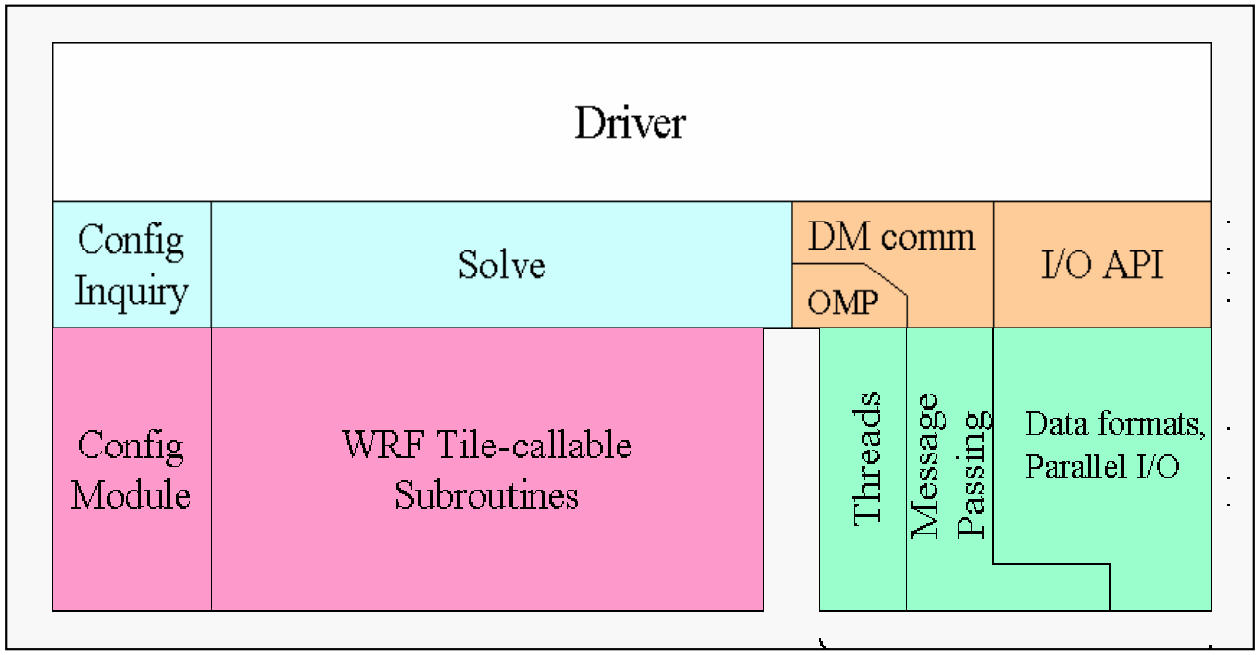
		Distributed Memory Parallel		Shared Memory Parallel
Domain	*contains*	Patches	*contain*	Tiles
Job	*contains*	Processes	*contain*	Threads
Cluster	*contains*	Nodes	*contain*	Processors

WRF Software Overview

WRF Software

- Architecture
- Directory structure
- Module Conventions and USE Association
- Model Layer Interface

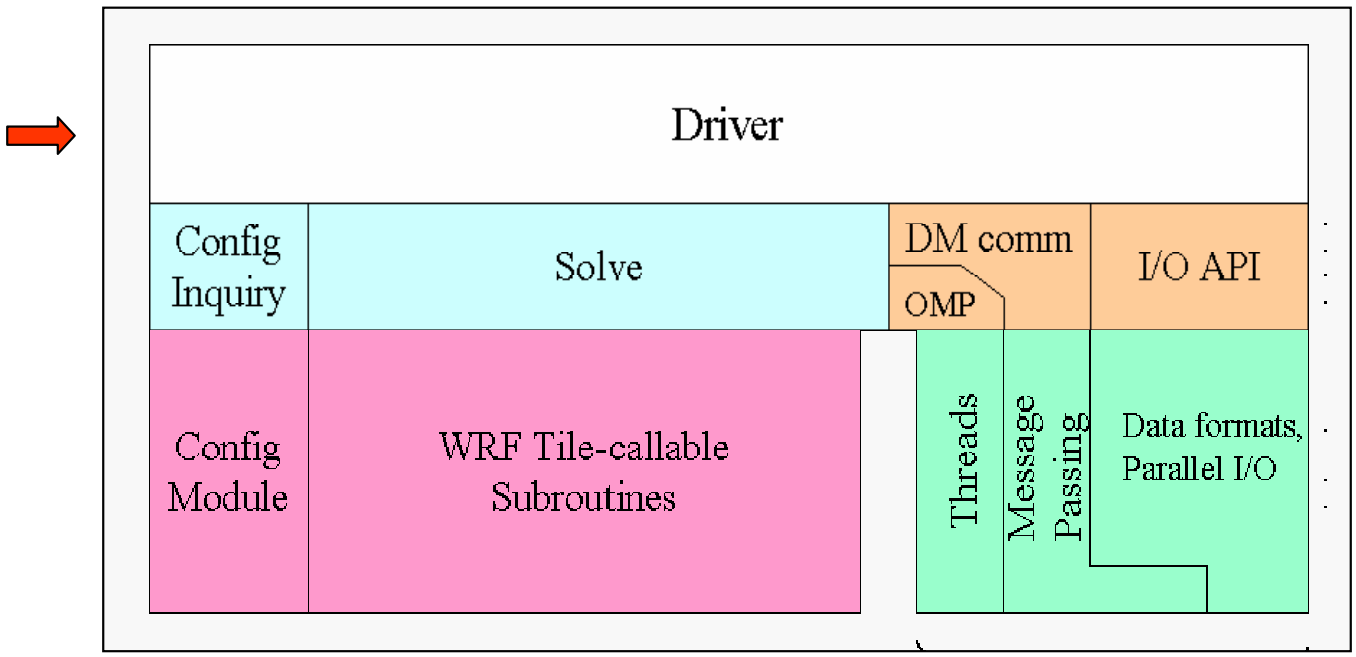
WRF Software Architecture



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.

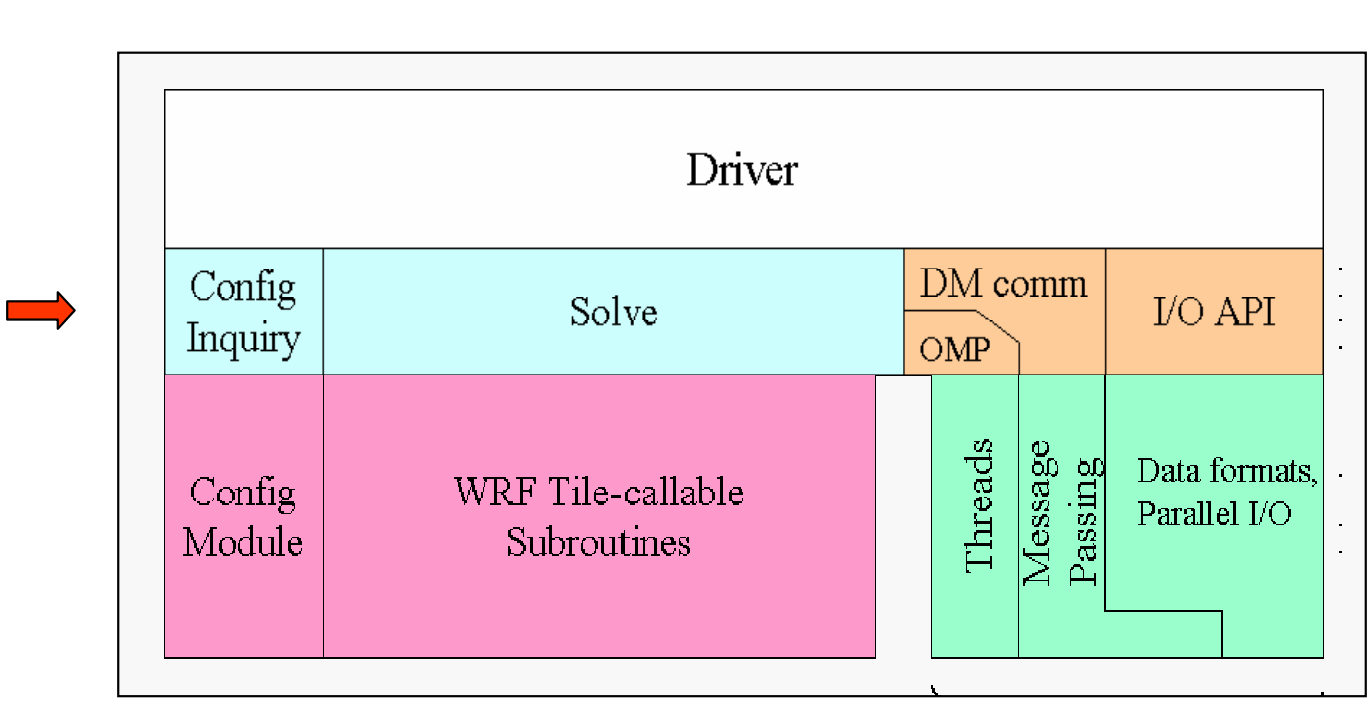
WRF Software Architecture



Registry

- Driver Layer
  - Allocates, stores, decomposes model domains, represented abstractly as single data objects
  - Contains top-level time loop and algorithms for integration over nest hierarchy
  - Contains the calls to I/O, nest forcing and feedback routines supplied by the Mediation Layer
  - Provides top-level, non package-specific access to communications, I/O, etc.
  - Provides some utilities, for example **module_wrf_error**, which is used for diagnostic prints and error stops

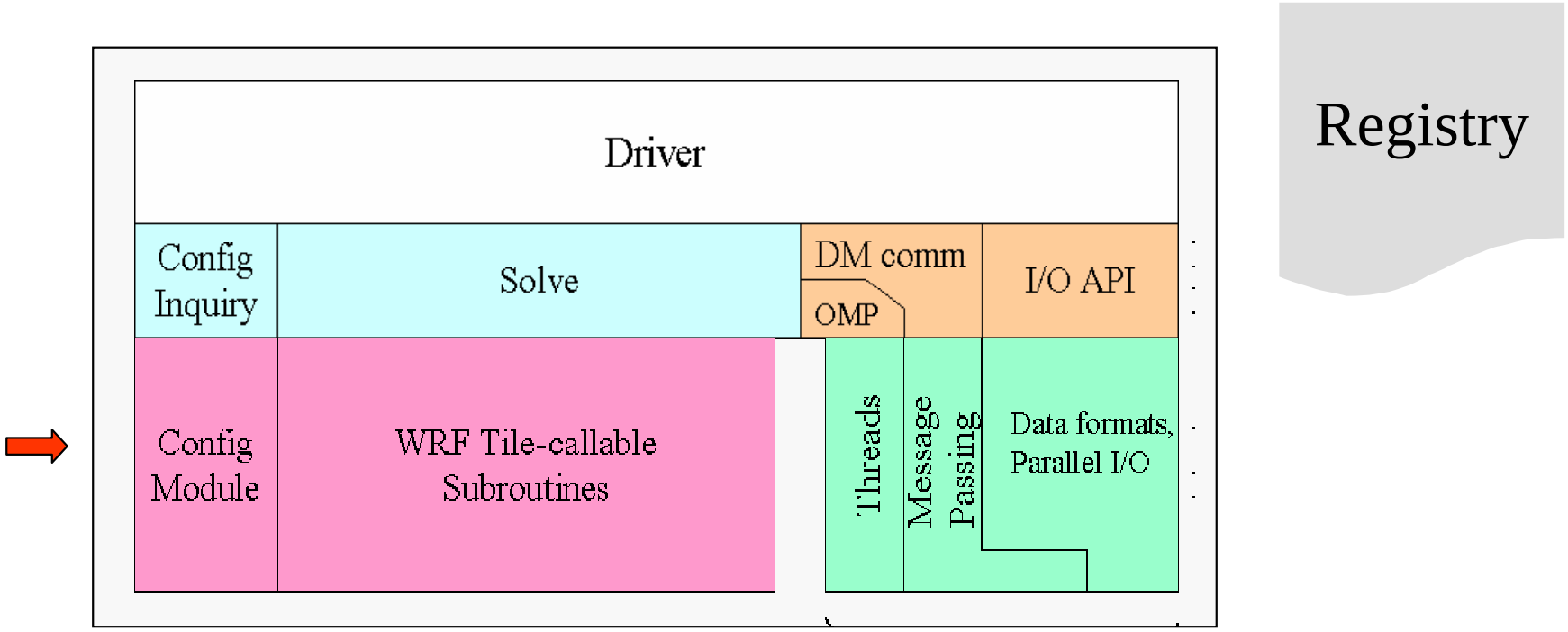
WRF Software Architecture



Registry

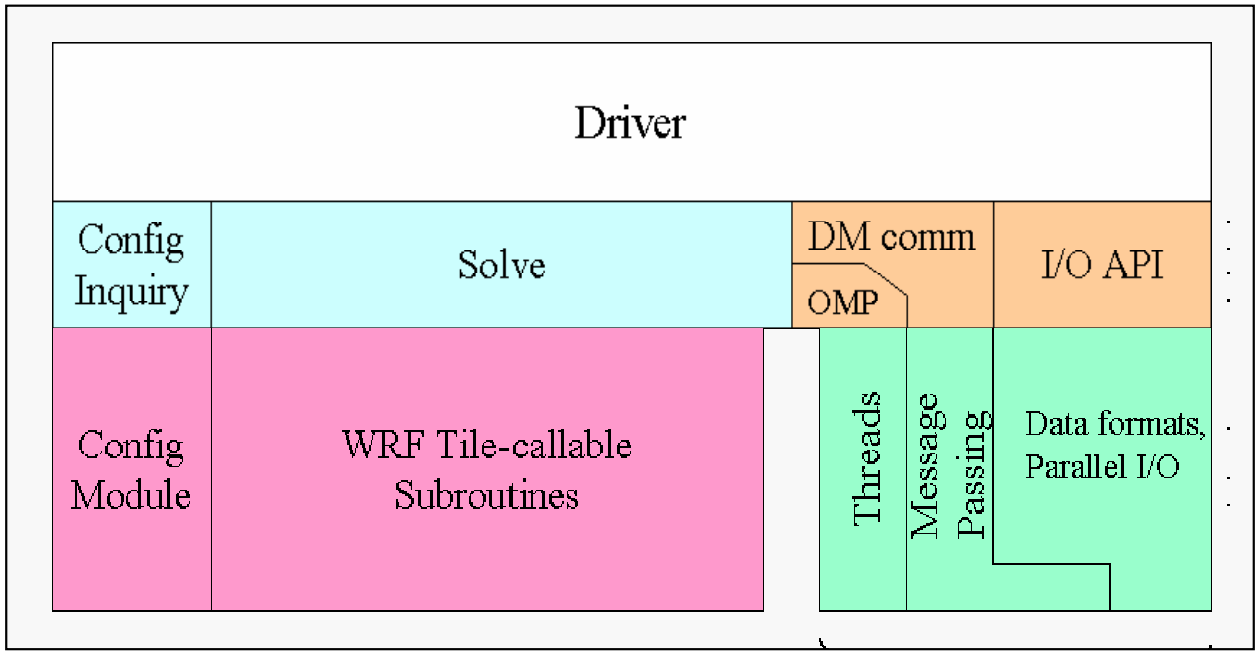
- Mediation Layer
  - Provides to the Driver layer
    - Solve solve routine, which takes a domain object and advances it one time step
    - I/O routines that Driver calls when it is time to do some input or output operation on a domain
    - Nest forcing and feedback routines
    - The Mediation Layer and not the Driver knows the specifics of what needs to be done
  - The sequence of calls to Model Layer routines for doing a time-step is known in Solve routine
  - Responsible for dereferencing driver layer data objects so that individual fields can be passed to Model layer Subroutines
  - Calls to message-passing are contained here as part of solve routine

WRF Software Architecture



- Model Layer
  - Contains the information about the model itself, with machine architecture and implementation aspects abstracted out and moved into layers above
  - Contains the actual WRF model routines that are written to perform some computation over an arbitrarily sized/shaped subdomain
  - All state data objects are simple types, passed in through argument list
  - Model Layer routines don't know anything about communication or I/O; and they are designed to be executed safely on **one thread** – they never contain a **PRINT**, **WRITE**, or **STOP** statement
  - These are written to conform to the Model Layer Subroutine Interface (more later) which makes them "tile-callable"

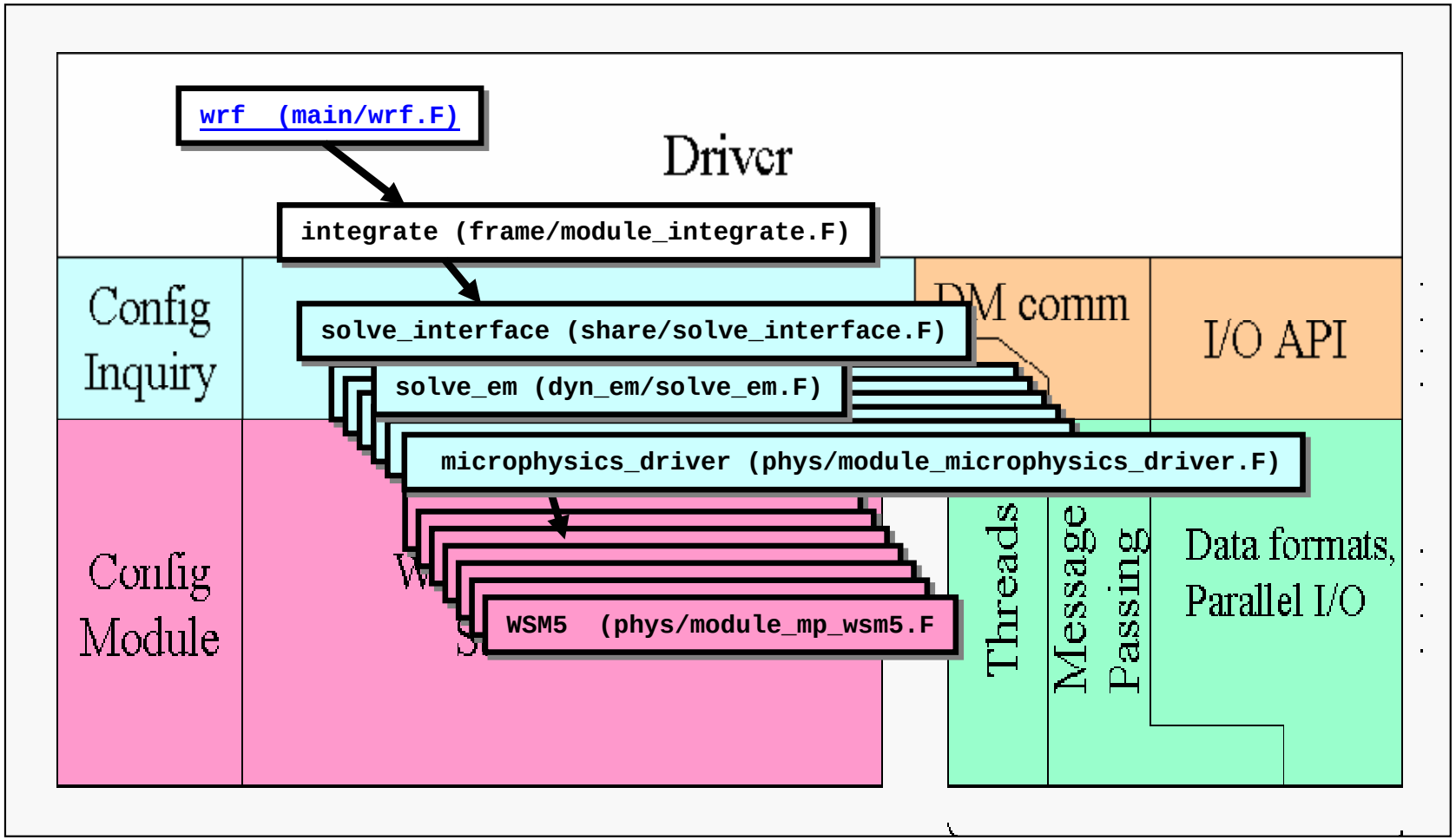
WRF Software Architecture



Registry

- 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

Call Structure superimposed on Architecture



WRF Model Directory Structure

```
maple% ls
CHANGES              README.ADDCORE      arch/       dyn_em/      external/
CVS/                 README.GRAPS        clean*      dyn_exp/     frame/
Makefile             README.NMM          compile*    dyn_graps/   inc/
NEST_SESSION         README_test_cases   configure*  dyn_nmm/     main/
README               Registry/           dyn_eh/     dyn_slt/     phys/
```

2.1. DIRECTORY STRUCTURE

The top-level WRFMODEL directory contains the following:

main -- directory containing Makefile and files containing main programs for the WRF model and initialization programs;

frame -- directory containing Makefile and source files specific to the WRF software framework;

dyn_*xx* -- directory containing Makefile and source files specific to a particular dynamical core *xx*;

phys -- directory containing Makefile and source files for physics;

share -- directory containing Makefile and source files for non-physics modules shared between dynamical cores;

external -- directory containing Makefile and subdirectories containing external packages for I/O, communications, etc.;

Registry -- directory containing the registry database;

clean, configure, and compile -- shell scripts (csh) for cleaning, configuring, and compiling the model;

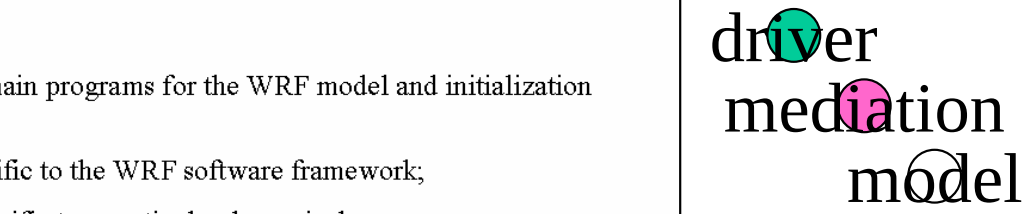
arch -- directory containing settings files and scripts for configuring the model on different platforms; the file containing the settings for all currently supported platforms is configure.defaults;

inc -- directory that holds registry-generated include files (essentially empty on initial distribution);

tools -- directory containing tools used to build the model; the Makefile and source files for the registry mechanism reside here;

run and test -- run directories for the model; run is the default run directory; test contains standardized idealized and real-data test cases for the model; and

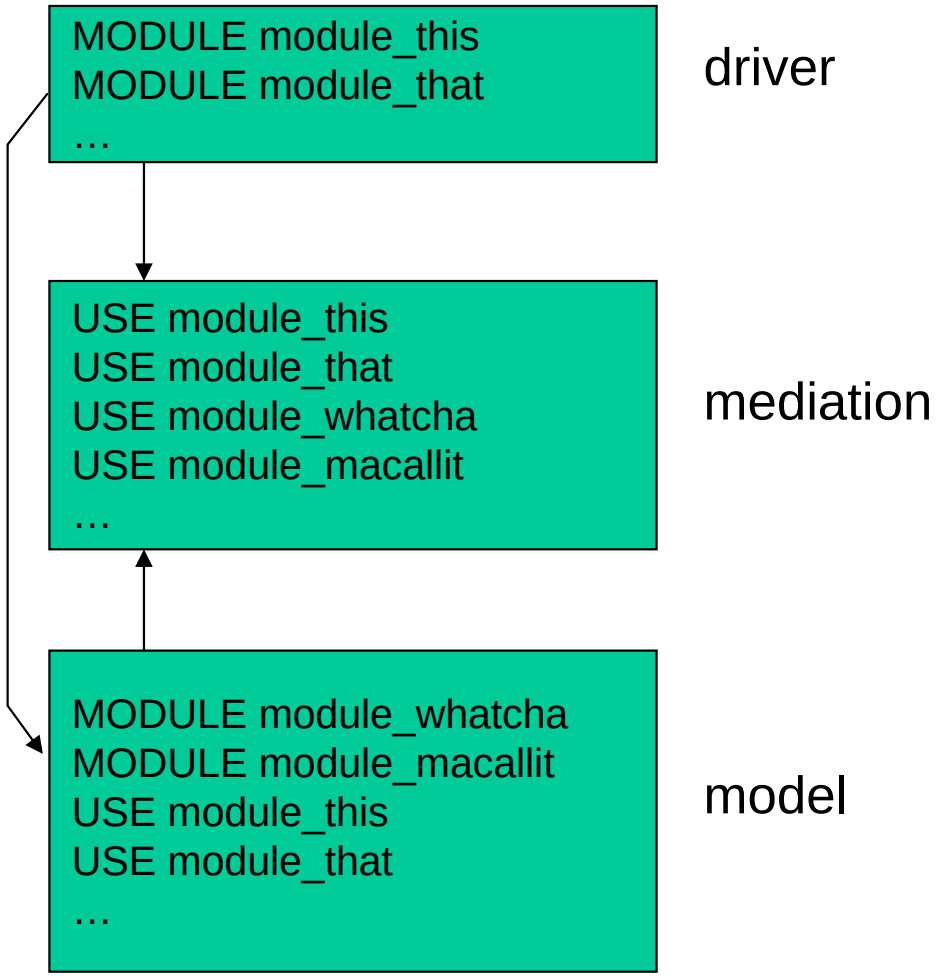
Makefile -- the top level (UNIX) make file for building WRF. This is not used directly; WRF is configured and built using the scripts mentioned above.



page 5,
WRF D&I Document

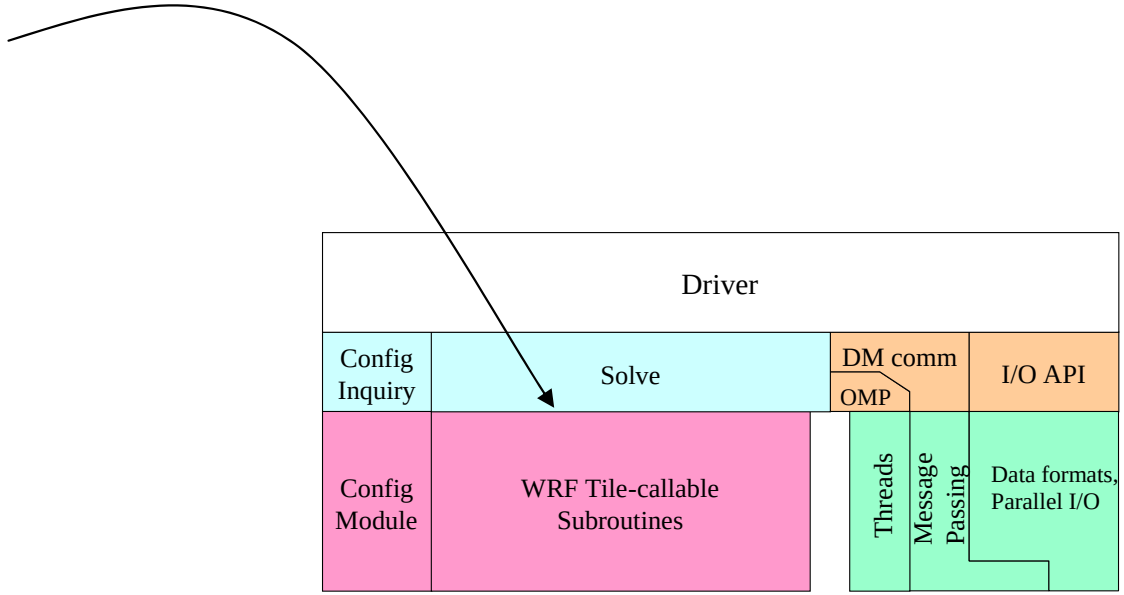
Module Conventions and USE Association

- Modules are named module_*something*
- Name of file containing module is module_*something*.F
- If a module includes an initialization routine, that routine should be named init_module_*something()*
- Typically:
  - Driver and model layers are made up of modules,
  - Mediation layer is bare subroutines, except for physics drivers in phys directory
  - Gives benefit of modules while avoiding cycles in the use association graph



WRF Model Layer Interface

- Mediation layer / Model Layer Interface



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
- 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

```
SUBROUTINE solve_xxx (
    . . .
!$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
```

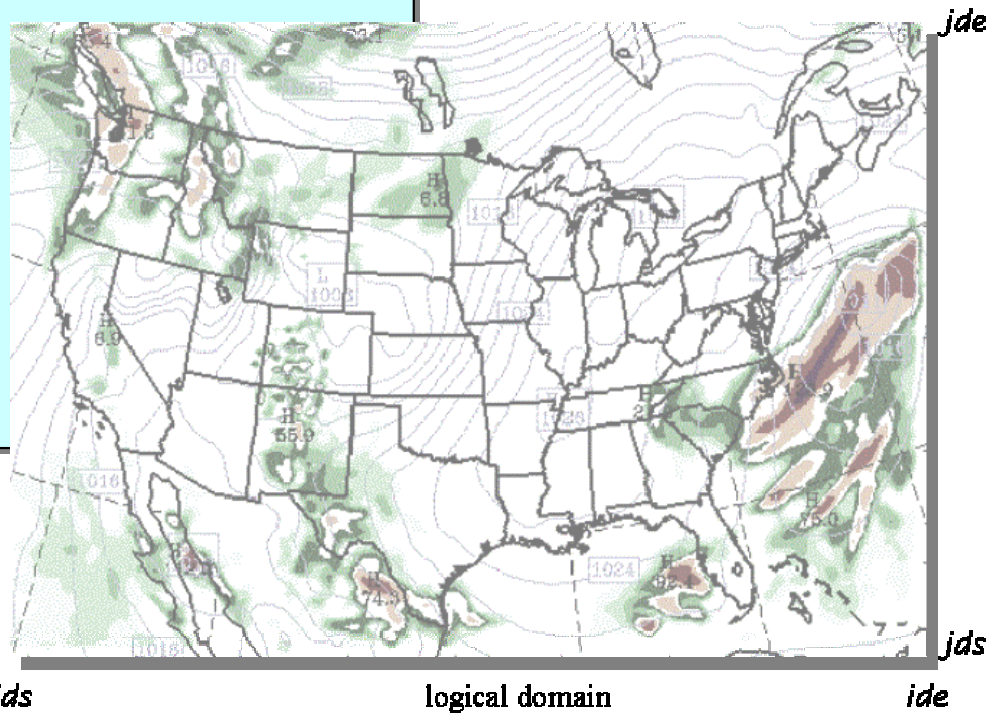
template for model layer subroutine

```
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 (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 = jts, jte
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide)
      loc(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 = jts, jte
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide)
      loc(i,k,j) = arg1(i,k,j) + …
    END DO
  END DO
END DO
```

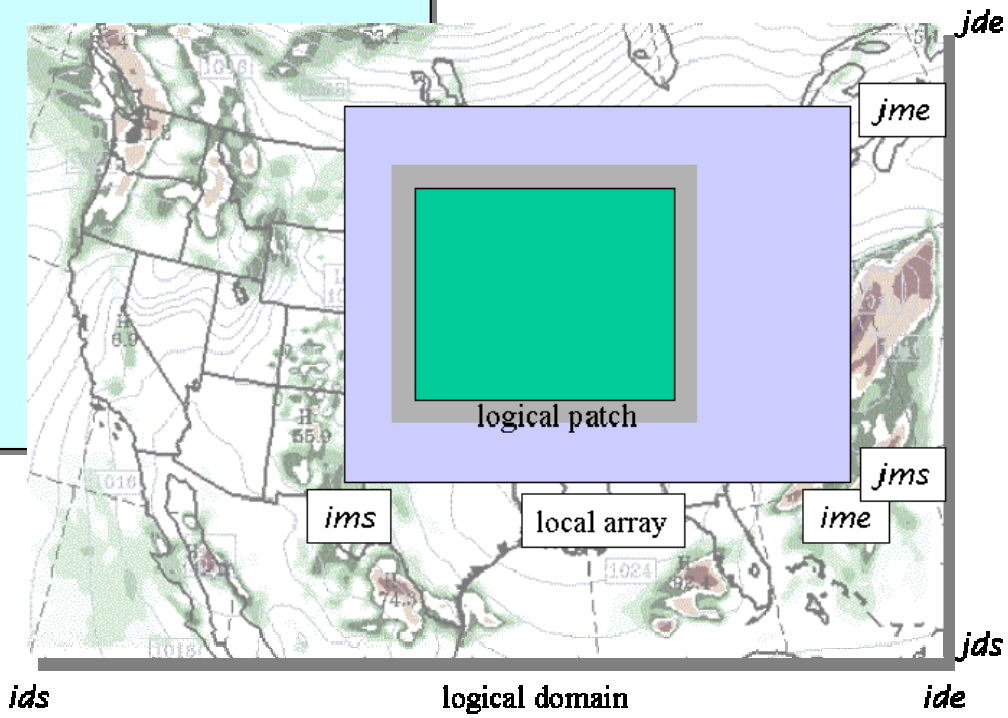
- Domain dimensions
  - Size of logical domain
  - Used for bdy tests, etc.



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 = jts, jte
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide)
      loc(i,k,j) = arg1(i,k,j) + …
    END DO
  END DO
END DO
```

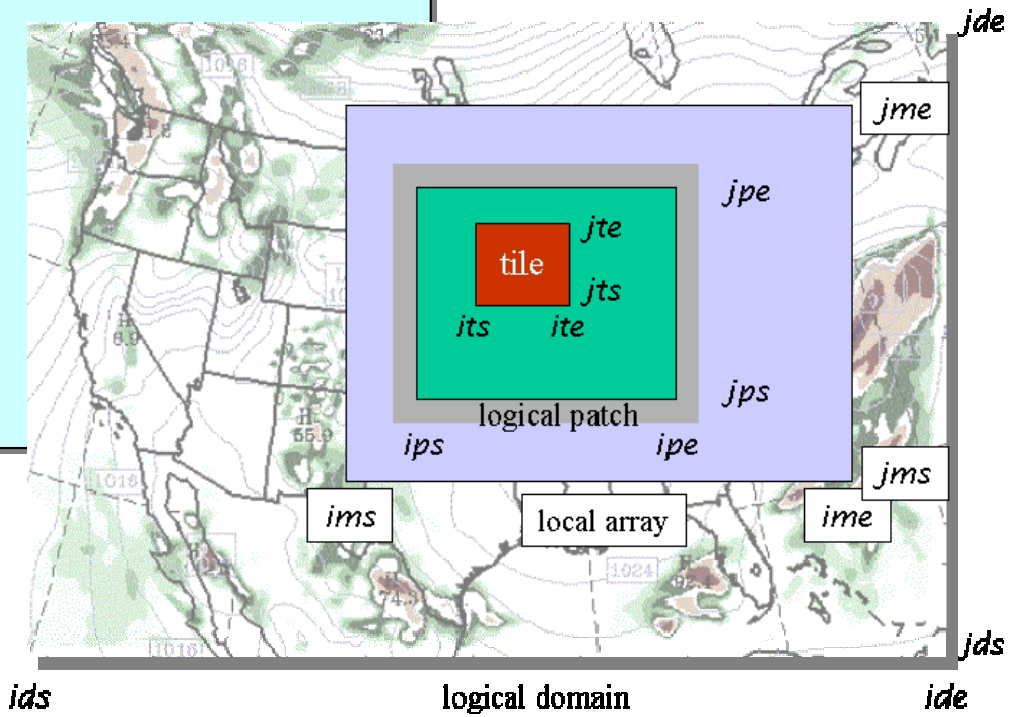
- Domain dimensions
  - Size of logical domain
  - Used for bdy tests, etc.
- Memory dimensions
  - Used to dimension dummy arguments
  - Do not use for local arrays



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 = jts, jte
  DO k = kts, kte
    DO i = MAX(its,ids), MIN(ite,ide)
      loc(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



Data Structures

Data Structures

- Data Taxonomy
- How data appears at different levels of architecture
- Grid representation in WRF arrays
- Lateral Boundary Condition arrays
- Four dimensional tracer arrays

Data Structures

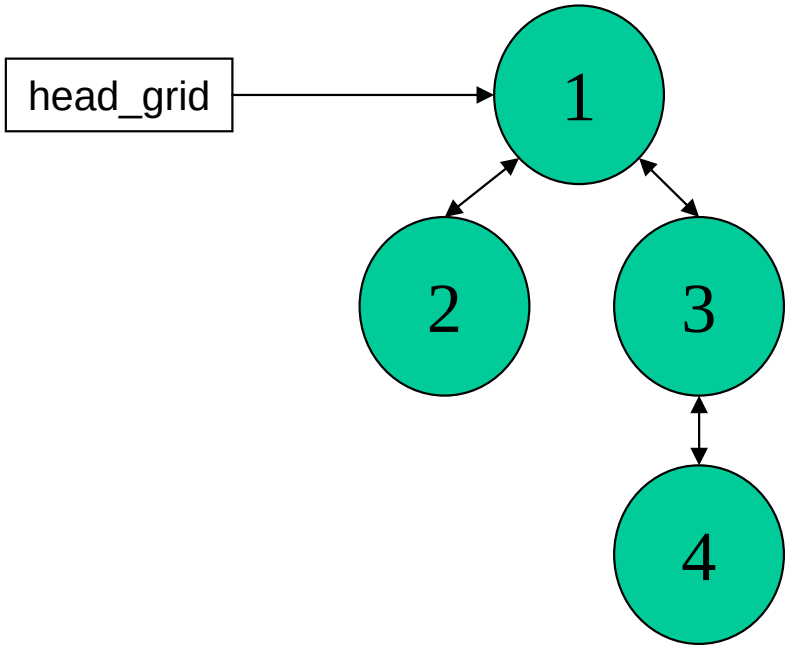
- WRF Data Taxonomy
  - State data
  - Intermediate data type 1 (I1)
  - Intermediate data type 2 (I2)
  - Heap storage (COMMON or Module data)

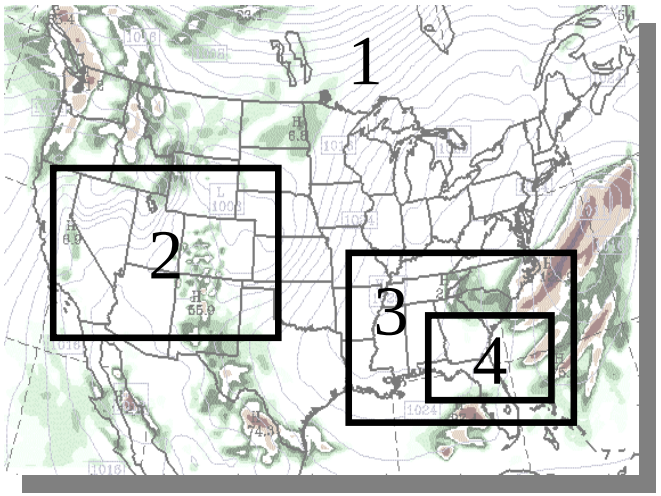
State Data

- Persist for the duration of a domain
- Represented as fields in domain data structure
- Arrays are represented as dynamically allocated pointer arrays in the domain data structure
- Declared in Registry using **state** keyword
- Always **memory** dimensioned; always **thread shared**
- Only state arrays can be subject to I/O and Interprocessor communication

Data Structures

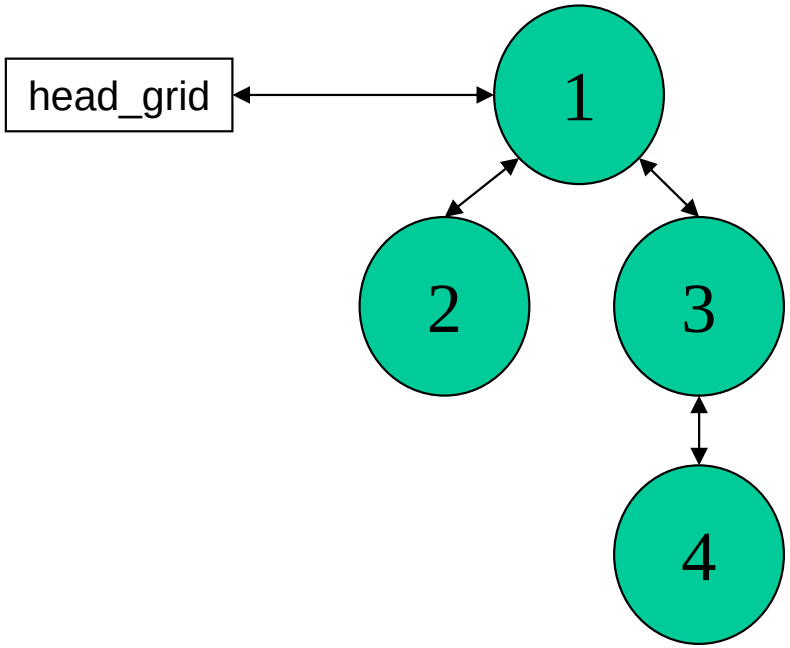
- What you see depends on where you are
  - Driver layer
    - All data for a domain is a single object, a domain derived data type (DDT)
    - The domain DDTs are dynamically allocated/deallocated
    - Linked together in a tree to represent nest hierarchy; root pointer is **head_grid**, defined in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





Data Structures

- What you see depends on where you are
  - Driver layer
    - All data for a domain is a single 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)



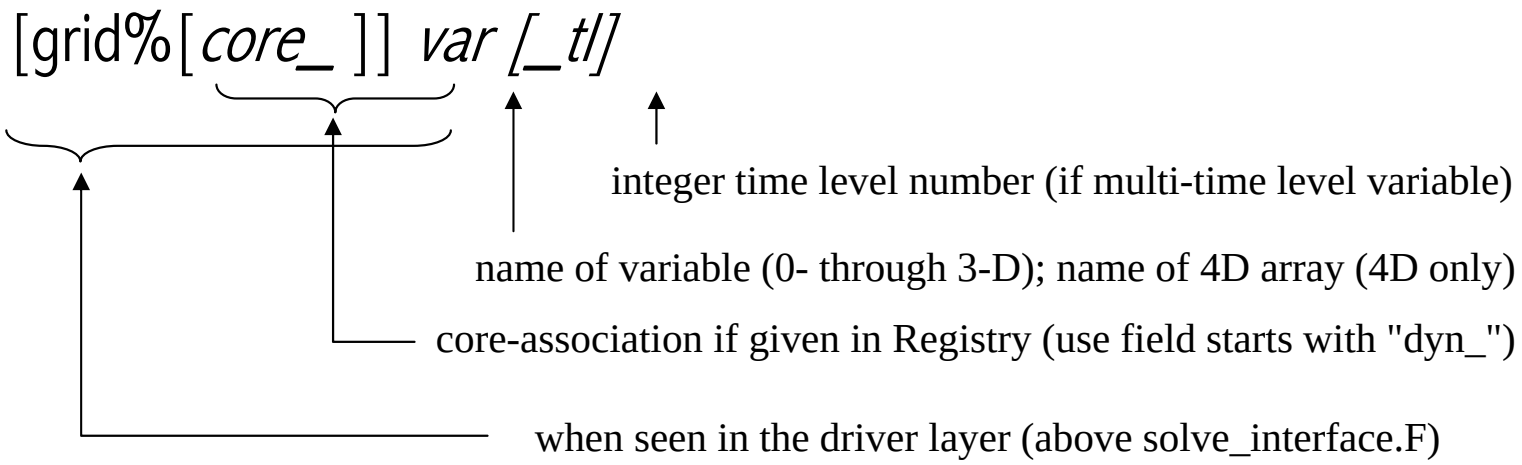

```
RECURSIVE SUBROUTINE integrate ( domain , stoptime )

  WHILE ( domain.time <= stoptime )
    CALL solve ( domain )   ! Advances domain.time
      FOR EACH CHILD OF DOMAIN
        CALL integrate ( domain.child , domain.time )
```

Data Structures

- What you see depends on where you are (Cont.)
  - Model layer
    - All data objects are scalars and arrays of simple types only
    - Virtually all passed in through subroutine argument lists
  - 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
  - The name of a data type and how it is referenced may differ depending on the level of the architecture

WRF State Variables

- May be 0d, 1d, 2d, 3d, or 4d
- What they look like in the code:



Example

The second time level of the u variable in the Eulerian Mass (EM) core can be accessed in the driver layer as:

```
grid%em_u_2
```

in the solve_em routine and below it is simply:

```
u_2
```

Data Structures

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

I1 Data

- Data that persists for the duration of 1 time step on a domain and then released
- Declared in Registry using **i1** keyword
- Typically automatic storage (program stack) in solve routine
- Typical usage is for tendency arrays in solver
- Always **memory** dimensioned and **thread shared**
- Typically **not** communicated or I/O

I2 Data

- I2 data are local arrays that exist only in model-layer subroutines and exist only for the duration of the call to the subroutine
- I2 data is not declared in Registry, never communicated and never input or output
- I2 data is **tile** dimensioned and **thread local**

Heap Storage

- Data stored on the process heap is not thread- safe and is generally forbidden anywhere in WRF
  - COMMON declarations
  - Module data
- Exception: If the data object is:
  - Completely contained and private within a Model Layer module, and
  - Set once and then read-only ever after, and
  - No decomposed dimensions.

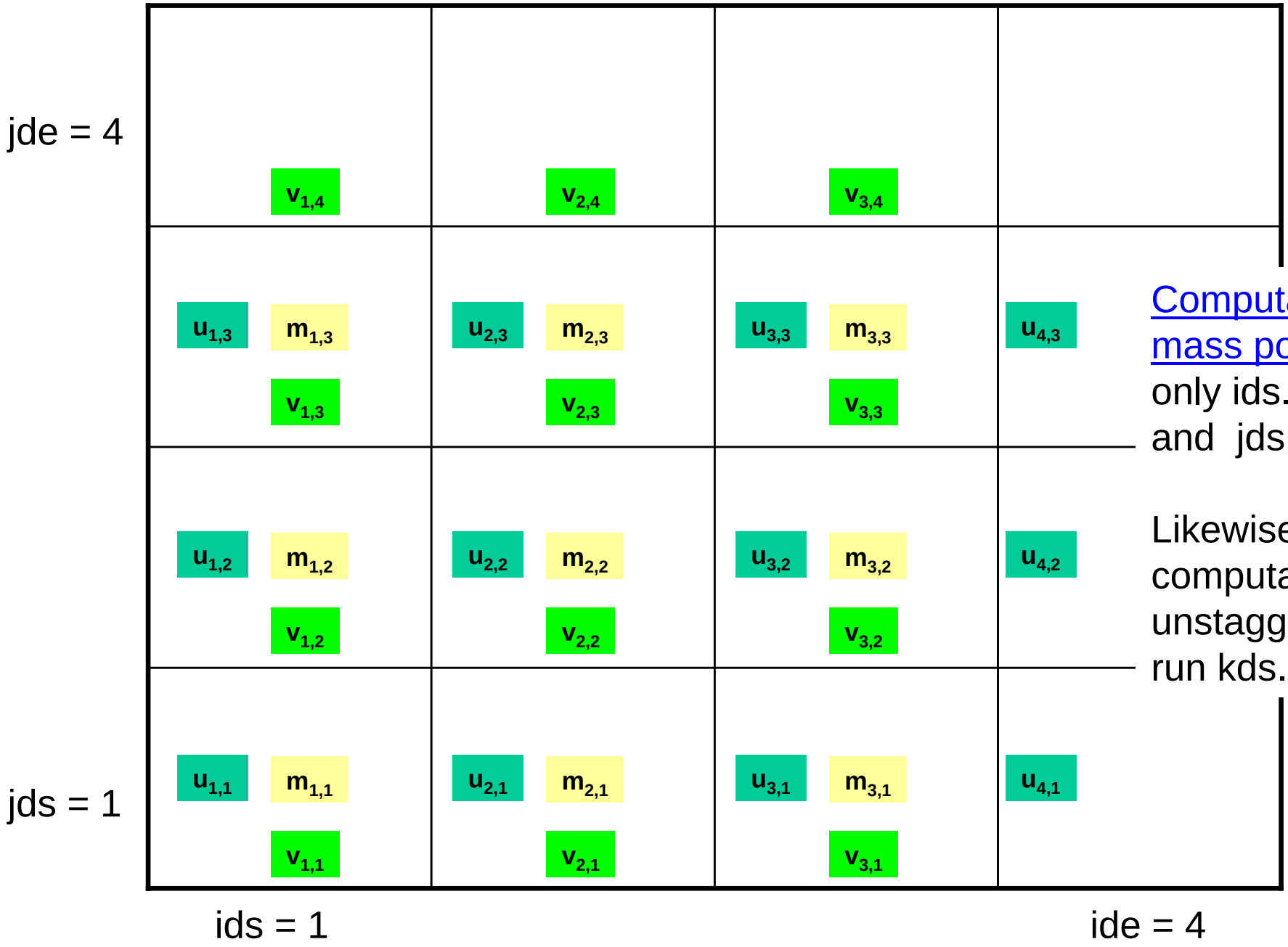
Grid Representation in Arrays

- 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

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.

Grid Indices Mapped onto Array Indices (C-grid example)



LBC Arrays

- Two additional arrays containing lateral boundary values and tendencies that are associated with a 2-, 3-, or 4-dimensional state arrays when the **b** modifier in the dimension field of the state array's entry of the Registry
- These store boundary forcing data, either from a file (mother domain), or interpolated from a parent (nested domain)
- All four boundaries are stored in the array; last index is over:

  P_XSB (western)
  P_XEB (eastern)
  P_YSB (southern)
  P_YEB (northern)

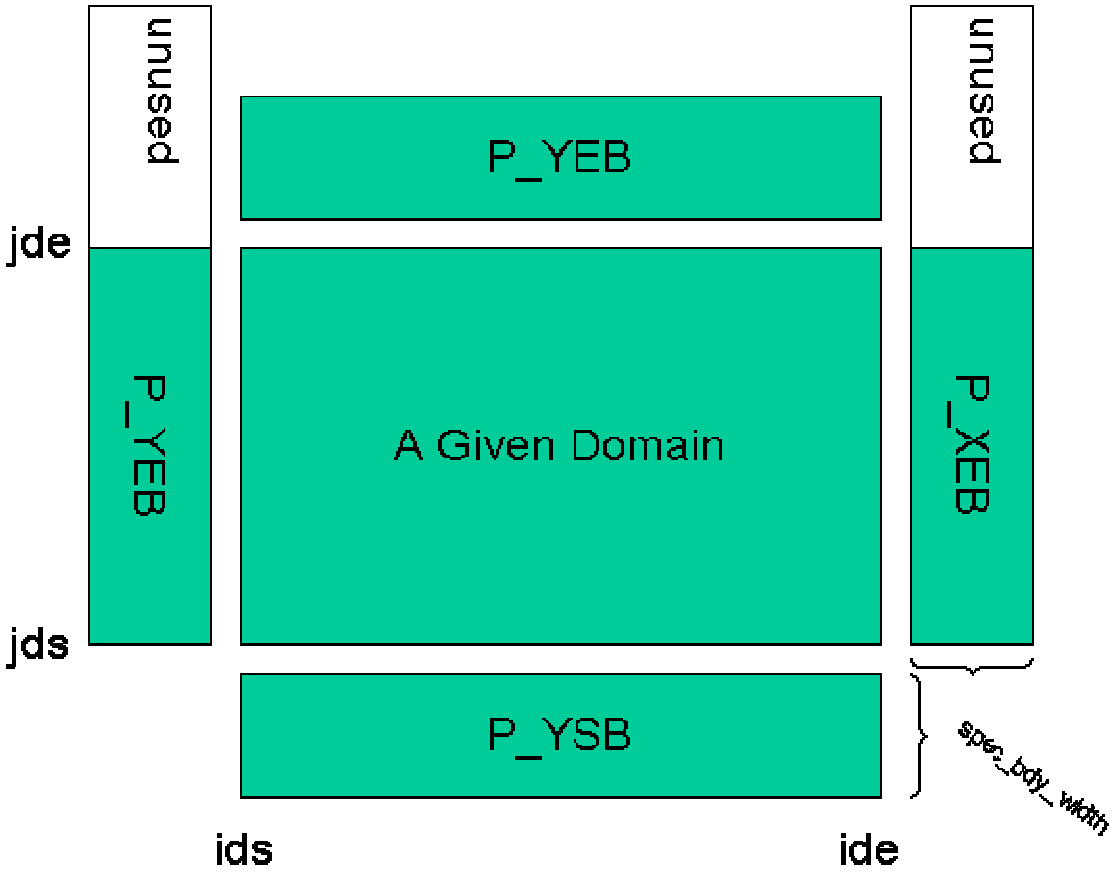
  These are defined in module_state_description.F

LBC Arrays

- LBC arrays are dimensioned as:

em_u_b(max(ide,jde),kde,spec_bdy_width,4)
em_u_bt(max(ide,jde),kde,spec_bdy_width,4)

  - Globally dimensioned in first index as the maximum of x and y dimensions
  - Second index is over vertical dimension
  - Third index is the width of the boundary (defined in the namelist)
  - Fourth index is which boundary



Four Dimensional Tracer Arrays

```
    --- File: Registry ---
# Moist Scalars
#       type   sym   dims     use    tl  stag  io    dname
state   real    qv  ikjftb   moist  1     -    irh  "QVAPOR"
state   real    qc  ikjftb   moist  1     -    irh  "QCLOUD"
state   real    qr  ikjftb   moist  1     -    irh  "QRAIN"
state   real    qi  ikjftb   moist  1     -    irh  "QICE"
state   real    qs  ikjftb   moist  1     -    irh  "QSNOW"
state   real    qg  ikjftb   moist  1     -    irh  "QGRAUP"
```

- State arrays, used to store arrays of 3D fields such as moisture tracers, chemical species, ensemble members, etc.
- First 3 indices are over grid dimensions; last dimension is the tracer index
- Each tracer is declared in the Registry as a separate **state** array but with **f** and optionally also **t** modifiers to the dimension field of the entry
- The field is then added to the 4D array whose name is given by the use field of the Registry entry

Four Dimensional Tracer Arrays

```
    --- File: Registry ---
# Moist Scalars
#       type   sym   dims     use    tl  stag  io    dname
state   real    qv  ikjftb   moist  1     -    irh  "QVAPOR"
state   real    qc  ikjftb   moist  1     -    irh  "QCLOUD"
state   real    qr  ikjftb   moist  1     -    irh  "QRAIN"
state   real    qi  ikjftb   moist  1     -    irh  "QICE"
state   real    qs  ikjftb   moist  1     -    irh  "QSNOW"
state   real    qg  ikjftb   moist  1     -    irh  "QGRAUP"
```

- Appears in Solver as:

  moist (ims:ime, kms:kme, jms:jme, num_moist)

  moist_tend (ims:ime, kms:kme, jms:jme, num_moist)

  moist_b(max(ide,jde), kms:kme, spec_bdy_width, 4, num_moist)

  moist_bt(max(ide,jde), kms:kme, spec_bdy_width, 4, num_moist)

- The extent of the last dimension of a tracer array is from PARAM_FIRST_SCALAR to num_*tracername* *(e.g.* moist*)*
  - PARAM_FIRST_SCALAR is a defined constant (value: 2)
  - Num_*tracername* is computed at run-time in set_scalar_indices_from_config (module_configure)
  - Calculation is based on which of the tracer arrays are associated with which specific packages in the Registry and on which of those packages is active at run time (namelist.input)

Four Dimensional Tracer Arrays

- Each tracer has an index (e.g. P_QV) that allows fields to be referenced individually from the 4D array where needed:

```
    --- File: dyn_em/solve_em.F ---

CALL microphysics_driver(                                                &
                    . . .
   &        , QV_CURR=moist(ims,kms,jms,P_QV),     F_QV=F_QV            &
   &        , QC_CURR=moist(ims,kms,jms,P_QC),     F_QC=F_QC            &
   &        , QR_CURR=moist(ims,kms,jms,P_QR),     F_QR=F_QR            &
   &        , QI_CURR=moist(ims,kms,jms,P_QI),     F_QI=F_QI            &
   &        , QS_CURR=moist(ims,kms,jms,P_QS),     F_QS=F_QS            &
   &        , QG_CURR=moist(ims,kms,jms,P_QG),     F_QG=F_QG            &
   &        , QNI_CURR=scalar(ims,kms,jms,P_QNI), F_QNI=F_QNI           &
                                                                         )
```

Four Dimensional Tracer Arrays

- Loops over tracer indices should always run from PARAM_FIRST_SCALAR to num_*tracername*

```
    --- File: dyn_em/solve_em.F ---

moist_variable_loop: DO im = PARAM_FIRST_SCALAR, num_moist

  CALL rk_update_scalar(                                  &
                          . . .
                        moist(ims,kms,jms,im),            &
                        moist_tend(ims,kms,jms,im),       &
                          . . .
                                                          )

ENDDO moist_variable_loop
```

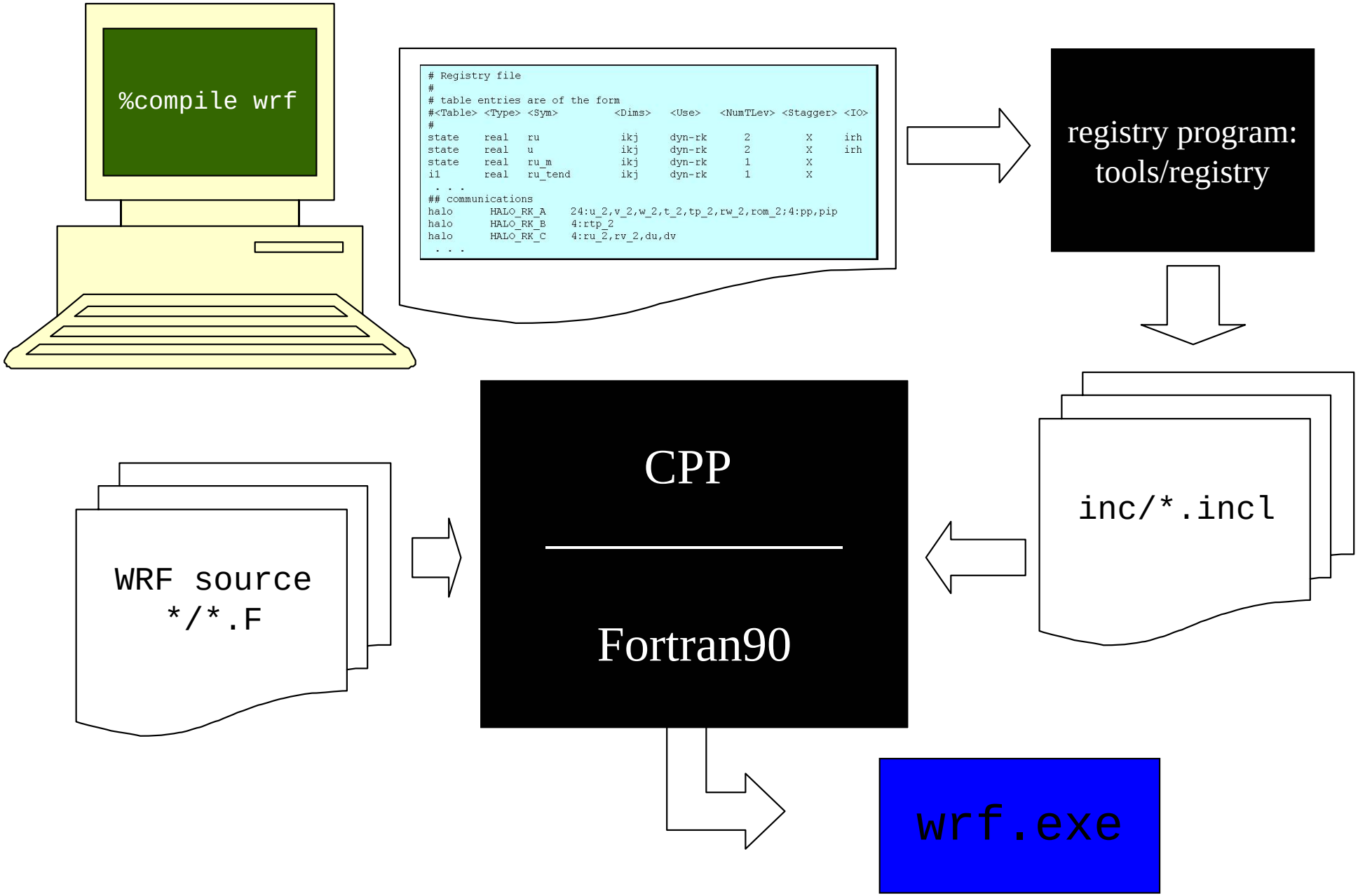
- Code should always test that a tracer index greater than or equal to PARAM_FIRST_SCALAR before referencing the tracer (inactive tracers have an index of 1)

The Registry

WRF Registry

- "Active data-dictionary" for managing WRF data structures
  - Database describing attributes of model state, intermediate, and configuration data
    - Dimensionality, number of time levels, staggering
    - Association with physics
    - I/O classification (history, initial, restart, boundary)
    - Communication points and patterns
    - Configuration lists (e.g. namelists)
  - Program for auto-generating sections of WRF from database:
    - 570 Registry entries $\Rightarrow$ 30-thousand lines of automatically generated WRF code
    - Allocation statements for state data, I1 data
    - Argument lists for driver layer/mediation layer interfaces
    - Interprocessor communications: Halo and periodic boundary updates, transposes
    - Code for defining and managing run-time configuration information
    - Code for forcing, feedback and interpolation of nest data
- Automates time consuming, repetitive, error-prone programming
- Insulates programmers and code from package dependencies
- Allow rapid development
- Documents the data
- Reference: **Description of WRF Registry,** http://www.mmm.ucar.edu/wrf/software_v2

Registry Mechanics



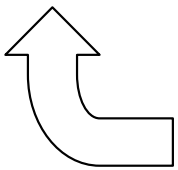
Registry Data Base

- Currently implemented as a text file: Registry/Registry.EM
- Types of entry:
  - *Dimspec* – Describes dimensions that are used to define arrays in the model
  - *State* – Describes state variables and arrays in the domain structure
  - *I1* – Describes local variables and arrays in solve
  - *Typedef* – Describes derived types that are subtypes of the domain structure
  - *Rconfig* – Describes a configuration (e.g. namelist) variable or array
  - *Package* – Describes attributes of a package (e.g. physics)
  - *Halo* – Describes halo update interprocessor communications
  - *Period* – Describes communications for periodic boundary updates
  - *Xpose* – Describes communications for parallel matrix transposes

Dimspec entry

- Elements
  - *Entry*: The keyword "dimspec"
  - *DimName*: The name of the dimension (single character)
  - *Order*: The order of the dimension in the WRF framework (1, 2, 3, or '-')
  - *HowDefined*: specification of how the range of the dimension is defined
  - *CoordAxis*: which axis the dimension corresponds to, if any (X, Y, Z, or C)
  - *DatName*: metadata name of dimension
- Example

```
#<Table>  <Dim>  <Order> <How defined>          <Coord-axis>  <DatName>
dimspec    i      1     standard_domain           x          west_east
dimspec    j      3     standard_domain           y          south_north
dimspec    k      2     standard_domain           z          bottom_top
dimspec    l      2     namelist=num_soil_layers  z          soil_layers
```



```
          --- namelist.input ---
   . . .
s_we                                = 1,
e_we                                = 91,
s_sn                                = 1,
e_sn                                = 82,
s_vert                              = 1,
e_vert                              = 28,
   . . .
num_soil_layers                     = 4,
```

Registry State Entry: ordinary State

- Elements
  - *Entry*: The keyword “state”
  - *Type*: The type of the state variable or array (real, double, integer, logical, character, or derived)
  - *Sym*: The symbolic name of the variable or array
  - *Dims*: A string denoting the dimensionality of the array or a hyphen (-)
  - *Use*: A string denoting association with a solver or 4D scalar array, or a hyphen
  - *NumTLev*: An integer indicating the number of time levels (for arrays) or hypen (for variables)
  - *Stagger*: String indicating staggered dimensions of variable (X, Y, Z, or hyphen)
  - *IO*: String indicating whether and how the variable is subject to I/O and Nesting
  - *DName*: Metadata name for the variable
  - *Units*: Metadata units of the variable
  - *Descrip*: Metadata description of the variable
- Example

```
#       Type Sym   Dims    Use      Tlev Stag IO         Dname       Descrip

state   real  u    ikjb    dyn_em    2    X    irhusdf    "U"     "X WIND COMPONENT“
```

Registry State Entry: ordinary State

```
#       Type Sym   Dims          Use      Tlev Stag IO     Dname       Descrip

state   real  u    ikjb    dyn_em    2    X     irhusdf    "U"     "X WIND COMPONENT“
```

- This single entry results in 130 lines automatically added to 43 different locations of the WRF code:
  - Declaration and dynamic allocation of arrays in TYPE(domain)
    - Two 3D state arrays corresponding to the 2 time levels of U
      u_1 (ims:ime , kms:kme , jms:jme)
      u_2 (ims:ime , kms:kme , jms:jme)
    - Two LBC arrays for boundary and boundary tendencies
      u_b (max(ide,jde), kms:kme, spec_bdy_width, 4)
      u_bt (max(ide,jde), kms:kme, spec_bdy_width, 4)
  - Add u_1, u_2, u_b, and u_2 to solver argument list
  - Nesting code to interpolate, force, feedback, and smooth **u**
  - Addition of **u** to the input, restart, history, and LBC I/O streams
  - Interprocessor communications involving u_1 and u_2 (with additional registry entries)

Registry State Entry: Four Dimensional Tracer Arrays

```
    --- File: Registry ---
# Moist Scalars
#       type   sym   dims     use    tl  stag  io    dname
state   real    qv  ikjftb   moist  1     -    irh  "QVAPOR"
state   real    qc  ikjftb   moist  1     -    irh  "QCLOUD"
state   real    qr  ikjftb   moist  1     -    irh  "QRAIN"
state   real    qi  ikjftb   moist  1     -    irh  "QICE"
state   real    qs  ikjftb   moist  1     -    irh  "QSNOW"
state   real    qg  ikjftb   moist  1     -    irh  "QGRAUP"
```

- Defines a 4D tracer variable named “moist” that contains up to 6 moisture species
  - Also a 4d tendency array named moist_tend (because **t** modifier appears in dims)
  - Also two 5d LBC arrays (because **b** modifier appears in dims)
  - The moisture variables are part of history, input, and restart I/O
- Appears in Solver as:

moist (ims:ime, kms:kme, jms:jme, num_moist)

moist_tend (ims:ime, kms:kme, jms:jme, num_moist)

moist_b(max(ide,jde), kms:kme, spec_bdy_width, 4, num_moist)

moist_bt(max(ide,jde), kms:kme, spec_bdy_width, 4, num_moist)

Rconfig entry

- This defines namelist entries
- Elements
  - *Entry*: the keyword “rconfig”
  - *Type*: the type of the namelist variable (integer, real, logical, string)
  - *Sym*: the name of the namelist variable or array
  - *How set*: indicates how the variable is set: e.g. namelist or derived, and if namelist, which block of the namelist it is set in
  - *Nentries*: specifies the dimensionality of the namelist variable or array. If 1 (one) it is a variable and applies to all domains; otherwise specify max_domains (which is an integer parameter defined in module_driver_constants.F).
  - *Default*: the default value of the variable to be used if none is specified in the namelist; hyphen (-) for no default
- Example

```
#         Type       Sym              How set          Nentries  Default
rconfig   integer spec_bdy_width  namelist,bdy_control    1          1
```

Rconfig entry

```
#          Type       Sym                How set           Nentries  Default
rconfig    integer spec_bdy_width  namelist,bdy_control     1           1
```

- Result of this Registry Entry:
  - Define an namelist variable “spec_bdy_width” in the bdy_control section of namelist.input
  - Type integer (others: real, logical, character)
  - If this is first entry in that section, define “bdy_control” as a new section in the namelist.input file
  - Specifies that bdy_control applies to all domains in the run
    - if Nentries is “max_domains” then the entry in the namelist.input file is a comma-separate list, each element of which applies to a separate domain
  - Specify a default value of “1” if nothing is specified in the namelist.input file
  - In the case of a multi-process run, generate code to read in the bdy_control section of the namelist.input file on one process and broadcast the value to all other processes

```
 --- File: namelist.input ---

&bdy_control
 spec_bdy_width       = 5,
 spec_zone            = 1,
 relax_zone           = 4,
    . . .
 /
```

Comm entries: halo and period

- Elements
  - *Entry*: keywords “halo” or “period”
  - *Commname*: name of comm operation
  - *Description*: defines the halo or period operation
    - For halo: *npts:f1,f2,...[;npts:f1,f2,...]**
    - For period: *width:f1,f2,...[;width:f1,f2,...]**
- Example

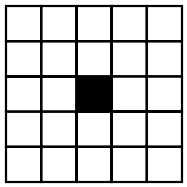
```
halo   HALO_EM_A  dyn_em 24:u_2,v_2,w_2,t_2;4:pp,pip
```

Comm entries: halo and period

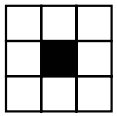
```
halo   HALO_EM_A  dyn_em 24:u_2,v_2,w_2,t_2;8:pp,pip
```

- Defines a halo exchange that, when invoked, will update:

  u_2, **v_2**, **w_2**, and **t_2** on a 24 point stencil:



  pp and **pip** on an 8 point stencil:



- Maximum allowed stencil is 168 pt
- Invoke by including in a mediation layer routine such as solve_em.F:

  #include <HALO_EM_A.inc>

I/O

WRF Model IO and Coupling

- WRF I/O and Coupling Streams
  - Streams: the logical data paths into and out of WRF
  - Available streams in WRF
    - Input, plus 5 auxiliary input streams
    - History, plus 5 auxiliary output streams
    - Dedicated output stream for Cycling 3DVAR
    - Boundary
    - Restart
  - Read from and written to in "variable-sets"
  - Variable-sets are defined at *compile-time* in the Registry
- Formats
  - The mechanism by which I/O is moved on a stream
  - Implemented using external packages and interfaced to the model through the WRF I/O and Model Coupling API (§7, WRF Design and Implementation Document)
  - Formats are specified at *run-time* in namelist.input
    - Native binary (Format 1)
    - NetCDF (Format 2)
    - Parallel HDF5 (Format 4), thanks Kent Yang, NCSA
    - Grib1 Output, thanks Todd Hutchinson, WSI
    - Experimental Model-Coupling interfaces through MCT, MCEL (Format 7)

I/O Software Stack

- Application-level I/O (Built-in)
  - Controlled through Registry and namelist
- Domain I/O
  - Subroutines in framework that perform I/O on a stream for an entire domain (based on variable sets defined in Registry)
- Low-Level I/O API
  - Subroutines provided by external packages
  - Operate on individual fields

Defining a variable-set for an I/O stream

- Fields are added to a variable-set on an I/O channel in the Registry

```
#       Type Sym  Dims   Use      Tlev Stag IO    Dname       Descrip

state   real  u   ikjb   dyn_em    2   X    irh   "U"   "X WIND COMPONENT“
```

IO is a string that specifies if the variable is to be subject to initial, restart, history, or boundary I/O. The string may consist of 'h' (subject to history I/O), 'i' (initial dataset), 'r' (restart dataset), or 'b' (lateral boundary dataset). The 'h', 'r', and 'i' specifiers may appear in any order or combination.

The ‘h’ and ‘i’ specifiers may be followed by an optional integer string consisting of ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, and/or ‘5’. Zero denotes that the variable is part of the principal input or history I/O stream. The characters ‘1’ through ‘5’ denote one of five auxiliary input or history I/O streams.

Defining Variable-set for an I/O stream

`irh` -- The state variable will be included in the input, restart, and history I/O streams

`irh13` -- The state variable has been added to the first and third auxiliary history output streams; it has been removed from the principal history output stream, because zero is not among the integers in the integer string that follows the character 'h'

`rh01` -- The state variable has been added to the first auxiliary history output stream; it is also retained in the principal history output

`i205hr` -- Now the state variable is included in the principal input stream as well as auxiliary inputs 2 and 5. Note that the order of the integers is unimportant. The variable is also in the principal history output stream

`ir12h` -- No effect; there is only 1 restart data stream and ru added to it.

Assigning I/O Streams to Formats

- Run-time: specified in namelist.input file

```
io_form_history    = 2,
io_form_restart    = 2,
io_form_input      = 2,
io_form_boundary   = 2,
```

Nest Initialization, Forcing, and Feedback

- Three built-in streams for exchange of data between nested domains
  - DOWN: data from a coarse domain state array to a nested domain state array
  - UP: data from a nested domain state array to a coarse-domain state array
  - FORCE: data from a coarse domain array to boundary arrays for a nested domain array
- Format is specialized, parallel and built-in to WRF
- Like I/O streams, variable-sets on nest streams defined in Registry
- User-specified interpolation functions can be added

```
#       Type Sym  Dims    Use      Tlev Stag              IO

state   real  u   ikjb    dyn_em    2    X    i01rhusdf=(bdy_interp:dt)
```

Outline

- Introduction
- Computing Overview
- Software Overview
- Data Structures
- Registry
- I/O & Nesting
- Examples

Example: Adding I/O to the Model

Example: Input periodic SSTs

- Problem: adapt WRF to input a time-varying lower boundary condition, e.g. SSTs, from an input file for a new surface scheme
- Given: Input file in WRF I/O format containing 12-hourly SST's
- Modify WRF model to read these into a new state array and make available to WRF surface physics

Example: Input periodic SSTs

- Steps
  - Add a new state variable and definition of a new surface layer package that will use the variable to the Registry
  - Add to variable stream for an unused Auxiliary Input stream
  - Adapt physics interface to pass new state variable to physics
  - Setup namelist to input the file at desired interval

Example: Input periodic SSTs

- Add a new state variable to Registry/Registry.EM and put it in the variable set for input on AuxInput #3

```
#     type  symbol dims use  tl stag  io     dname     description        units
state real  nsst   ij   misc  1  -    i3rh  "NEW_SST" "Time Varying SST" "K“
```

- Also added to History and Restart
- Result:
  - 2-D variable named **nsst** defined and available in solve_em
  - Dimensions: ims:ime, jms:jme
  - Input and output on the AuxInput #3 stream will include the variable under the name NEW_SST

Example: Input periodic SSTs

- Pass new state variable to surface physics

```
        --- File: dyn_em/solve_em.F ---

  CALL surface_driver(                                                &
            . . .
! Optional
     &        ,QG_CURR=moist(ims,kms,jms,P_QG), F_QG=F_QG                  &
     &        ,NSST=nsst                                                   & ! new
     &        ,CAPG=capg, EMISS=emiss, HOL=hol,MOL=mol                     &
     &        ,RAINBL=rainbl                                               &
     &        ,RAINNCV=rainncv,REGIME=regime,T2=t2,THC=thc                 &
     &        ,QSG=qsg,QVG=qvg,QCG=qcg,SOILT1=soilt1,TSNAV=tsnav           & ! ruc lsm
     &        ,SMFR3D=smfr3d,KEEPFR3DFLAG=keepfr3dflag                     & ! ruc lsm
     &                                                                  )
```

Example: Input periodic SSTs

- Add new variable **nsst** to Physics Driver in Mediation Layer

```
      --- File: phys/module_surface_driver.F ---

   SUBROUTINE surface_driver(                                          &
         . . .
            !  Other optionals (more or less em specific)
     &          ,nsst                                                  &
     &          ,capg,emiss,hol,mol                                    &
     &          ,rainncv,rainbl,regime,t2,thc                          &
     &          ,qsg,qvg,qcg,soilt1,tsnav                              &
     &          ,smfr3d,keepfr3dflag                                   &
            !  Other optionals (more or less nmm specific)
     &          ,potevp,snopcx,soiltb,sr                               &
                                                                       ))
         . . .
   REAL, DIMENSION( ims:ime, jms:jme ), OPTIONAL, INTENT(INOUT)::   nsst
```

- By making this an “Optional” argument, we preserve the driver’s compatibility with other cores and with versions of WRF where this variable hasn’t been added.

Example: Input periodic SSTs

- Add call to Model-Layer subroutine for new physics package to Surface Driver

```
        --- File: phys/module_surface_driver ---

!$OMP PARALLEL DO   &
!$OMP PRIVATE ( ij, i, j, k )
   DO ij = 1 , num_tiles
     sfclay_select: SELECT CASE(sf_sfclay_physics)

       CASE (SFCLAYSCHEME)
          . . .
       CASE (NEWSFCSCHEME)  ! <- This is defined by the Registry "package" entry

         IF (PRESENT(nsst))  THEN
            CALL NEWSFCCHEME(                                        &
                nsst,                                                &
                ids,ide, jds,jde, kds,kde,                           &
                ims,ime, jms,jme, kms,kme,                           &
                i_start(ij),i_end(ij), j_start(ij),j_end(ij), kts,kte    )
         ELSE
           CALL wrf_error_fatal('Missing argument for NEWSCHEME in surface driver')
         ENDIF
          . . .
     END SELECT sfclay_select
   ENDDO
!$OMP END PARALLEL DO
```

- Note the PRESENT test to make sure new optional variable **nsst** is available

Example: Input periodic SSTs

- Add definition for new physics package NEWSCHEME as setting 4 for namelist variable sf_sfclay_physics

```
rconfig   integer  sf_sfclay_physics   namelist,physics   max_domains    0

package   sfclayscheme   sf_sfclay_physics==1        -             -
package   myjsfcscheme   sf_sfclay_physics==2        -             -
package   gfssfcscheme   sf_sfclay_physics==3        -             -
package   newsfcscheme   sf_sfclay_physics==4        -             -
```

- This creates a defined constant NEWSFCSCHEME and represents selection of the new scheme when the namelist variable sf_sfclay_physics is set to '4' in the namelist.input file
- Clean –a and recompile so code and Registry changes take effect

Example: Input periodic SSTs

- Setup namelist to input SSTs from the file at desired interval

```
     --- File: namelist.input ---

&time_control
   . . .
 auxinput3_inname      = "sst_input"
 auxinput3_interval_mo = 0
 auxinput3_interval_d  = 0
 auxinput3_interval_h  = 12
 auxinput3_interval_m  = 0
 auxinput3_interval_s  = 0
   . . .
/

   . . .
&physics
 sf_sfclay_physics  = 4, 4, 4
   . . .
/
```

- Run code with sst_input file in run-directory

Example: Input periodic SSTs

- A few notes…
  - The read times and the time-stamps in the input file must match exactly
  - We haven't done anything about what happens if the file runs out of time periods (the last time period read will be used over and over again, though you'll see some error messages in the output if you set debug_level to be 1 or greater in namelist.input)
  - We haven't said anything about what generates sst_input

Example: Computing a Diagnostic

Example: Compute a Diagnostic

- Problem: Output global average and global maximum and lat/lon location of maximum for 10 meter wind speed in WRF
- Steps:
  - Modify solve to compute wind-speed and then compute the local sum and maxima at the end of each time step
  - Use reduction operations built-in to WRF software to compute the global qualitities
  - Output these on one process (process zero, the “monitor” process)

Example: Compute a Diagnostic

- Compute local sum and local max and the local indices of the local maximum

```
   --- File: dyn_em/solve_em.F  (near the end) ---

! Compute local maximum and sum of 10m wind-speed
   sum_ws = 0.
   max_ws = 0.
   DO j = jps, jpe
     DO i = ips, ipe
       wind_vel = sqrt( u10(i,j)*u10(i,j) + v10(i,j)*v10(i,j) )
       IF ( wind_vel .GT. max_ws ) THEN
          max_ws = wind_vel
          idex = i
          jdex = j
       ENDIF
       sum_ws = sum_ws + wind_vel
     ENDDO
   ENDDO
```

Example: Compute a Diagnostic

- Compute global sum, global max, and indices of the global max

```
! Compute global sum
   sum_ws = wrf_dm_sum_real ( sum_ws )

! Compute global maximum and associated i,j point
   CALL wrf_dm_maxval_real ( max_ws, idex, jdex )
```

Example: Compute a Diagnostic

- On the process that contains the maximum value, obtain the latitude and longitude of that point; on other processes set to an artificially low value.
- The use parallel reduction to store that result on every process

```
   IF ( ips .LE. idex .AND. idex .LE. ipe .AND.  &
        jps .LE. jdex .AND. jdex .LE. jpe ) THEN

      glat = xlat(idex,jdex)
      glon = xlong(idex,jdex)

   ELSE
      glat = -99999.
      glon = -99999.
   ENDIF

! Compute global maximum to find glat and glon
   glat = wrf_dm_max_real ( glat )
   glon = wrf_dm_max_real ( glon )
```

This had been *xlat*.
Fixed after tutorial presentation July 27, 2005. J.M.

Example: Compute a Diagnostic

- Output the value on process zero, the “monitor”

```
! Print out the result on the monitor process
   IF ( wrf_dm_on_monitor() ) THEN
      WRITE(outstring,*)'Avg. ',sum_ws/((ide-ids*1)*(jde-jds+1))
      CALL wrf_message ( TRIM(outstring) )
      WRITE(outstring,*)'Max. ',max_ws,' Lat. ',glat,' Lon. ',glon
      CALL wrf_message ( TRIM(outstring) )
   ENDIF
```

- Output from process zero of a 4 process run

Amended output after fix from previous slide. J.M.

```
     --- Output file: rsl.out.0000 ---
    . . .
  Avg.    5.159380
  Max.    15.09370     Lat.    37.25022     Lon.    -67.44571
Timing for main: time 2000-01-24_12:03:00 on domain   1:    8.96500 elapsed seconds.
  Avg.    5.166167
  Max.    14.97418     Lat.    37.25022     Lon.    -67.44571
Timing for main: time 2000-01-24_12:06:00 on domain   1:    4.89460 elapsed seconds.
  Avg.    5.205693
  Max.    14.92687     Lat.    37.25022     Lon.    -67.44571
Timing for main: time 2000-01-24_12:09:00 on domain   1:    4.83500 elapsed seconds.
    . . .
```