Tutorial Notes: WRF Software 2.1

John Michalakes, Head Software Architecture WG, NCAR

Dave Gill, NCAR

WRF Software Architecture Working Group

Outline

- Introduction
- Computing Overview
- WRF Software Overview

- - - - - - - - - - - - -

- Examples

Introduction – Intended Audience

- Intended audience for this tutorial session: scientific users and others who wish to:
  - Understand overall design concepts and motivations
  - 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

Introduction – WRF 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

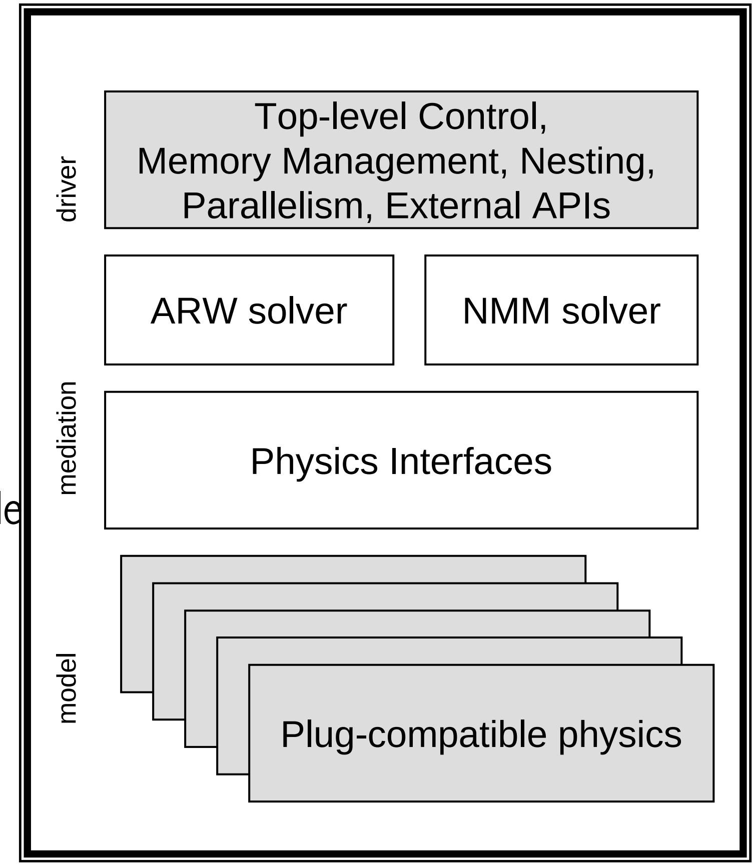
Introduction – WRF Software Characteristics

- 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 v2.1.2 (January 2006)
- Supported by flexible efficient architecture and implementation called the WRF Software Framework

Introduction - WRF Software Framework Overview

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

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



Introduction - WRF Supported Platforms

Vendor	Hardware	OS	Compiler
Apple (*)	G4/G5	MacOS	IBM
Cray Inc.	X1, X1e	UNICOS	Cray
	Opteron	Linux	PGI
HP/Compaq	Alpha	Tru64	Compaq
	Itanium-2	Linux	Intel
		HPUX	HP
IBM	Power-3/4/5; BG/L (**)	AIX	IBM
SGI	Itanium-2	Linux	Intel
	MIPS	IRIX	SGI
Sun (*)	UltraSPARC	Solaris	Sun
various	Xeon and Athlon	Linux	PGI, Intel, Pathscale
	Itanium-2 and Opteron		

(*) dm-parallel not supported yet; (**) Experimental, not released

Outline

- Introduction
- Computing Overview
- WRF Software Overview

  - - - - - - - - - - - - -

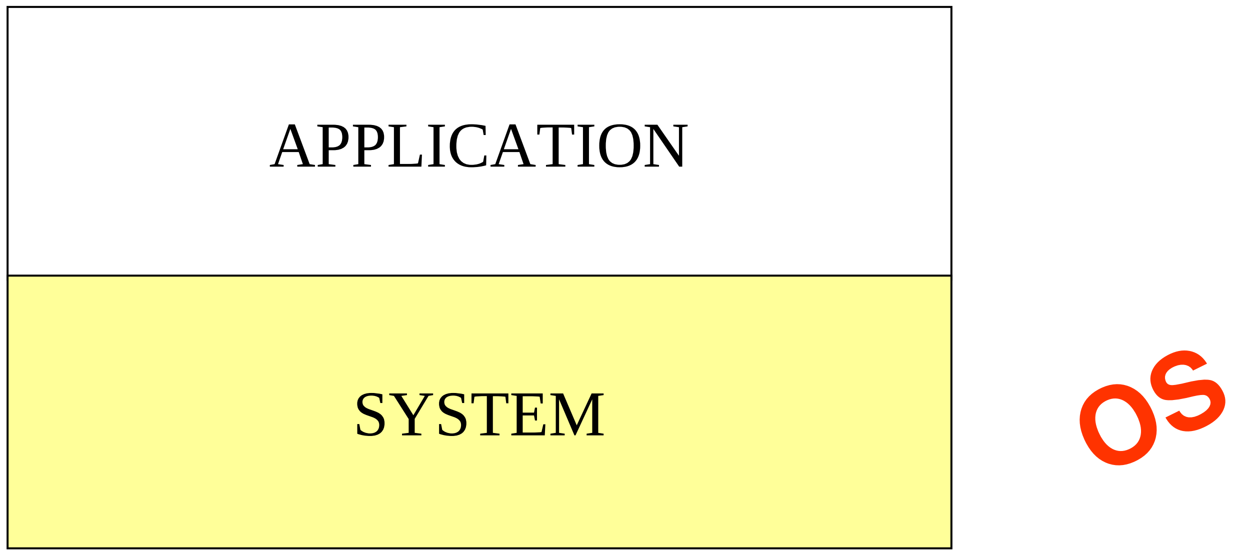
- Examples

Computing Overview

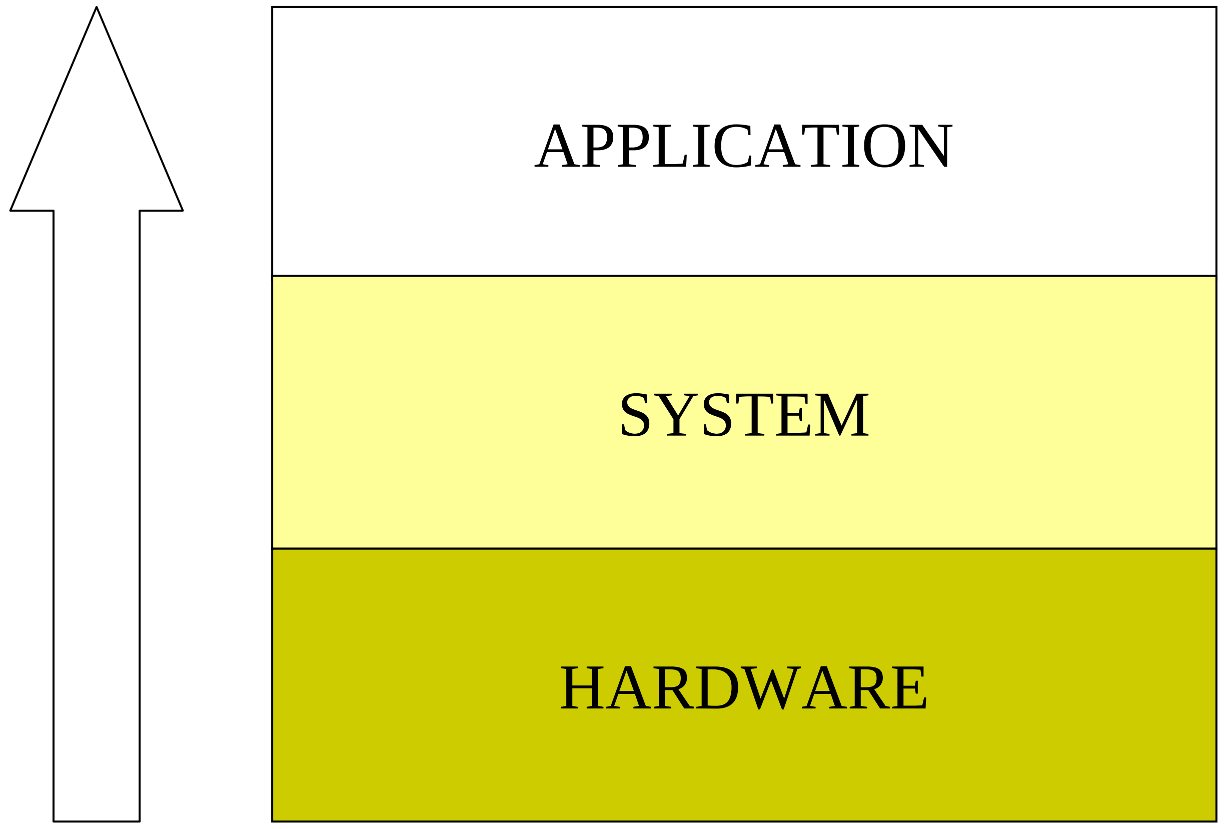
APPLICATION

WRF

Computing Overview



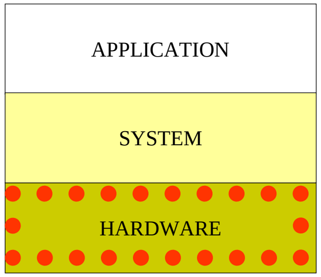
Computing Overview



Patches
Tiles
WRF Comms

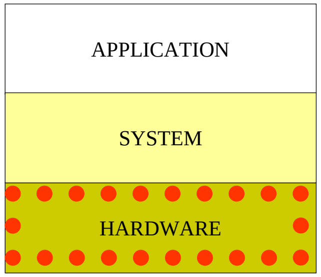
Processes
Threads
Messages

Processors
Nodes
Networks



Hardware: The Computer

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



Hardware has not changed much…

A computer in 1960

IBM 7090

6-way superscalar

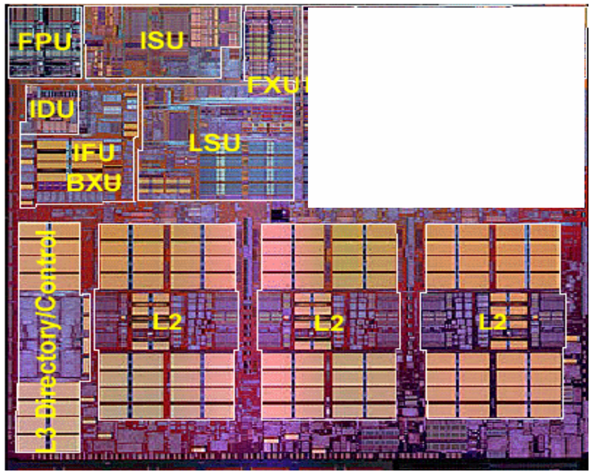
36-bit floating point precision

~144 Kbytes

~50,000 flop/s
48hr 12km WRF CONUS in 600 years

A computer in 2002

IBM p690



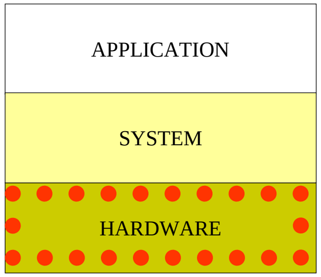
4-way superscalar

64-bit floating point precision

1.4 Mbytes (shown)

> 500 Mbytes (not shown)

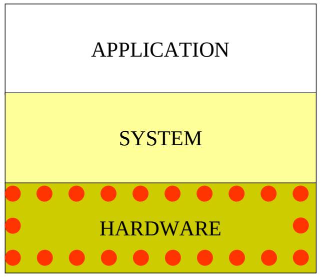
~5,000,000,000 flop/s
48 12km WRF CONUS in 52 Hours



…how we use it has

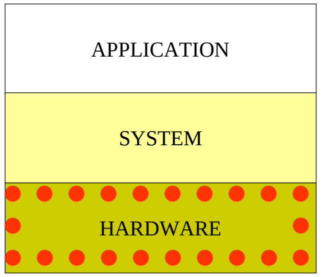
- Fundamentally, processors haven't changed much since 1960
- Quantitatively, they haven't improved nearly enough
  - 100,000x increase in peak speed
  - > 4,000x increase in memory size
  - These are too slow and too small for even a moderately large NWP run today
- We make up the difference with <u>parallelism</u>
  - 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



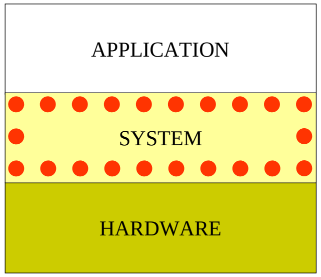
Parallel Computing Terms -- Hardware

- **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



Parallel Computing Terms -- Hardware

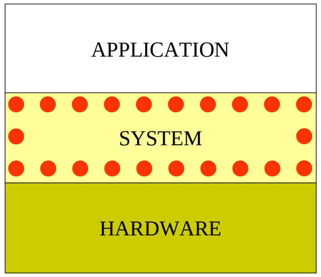
- **Cluster**: Multiple nodes connected by a network
  - The processors attached to the memory in one node can not see the memory for processors on another node
  - For processors on different nodes to work together they must send messages between the nodes. This is "distributed memory parallelism"
- **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 – System Software

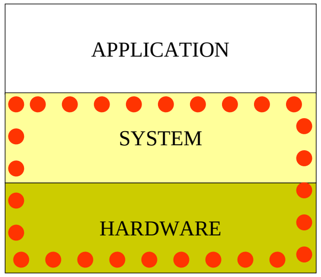
"The only thing one does directly with hardware is pay for it."

- 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 – System 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.
- Heavy-weight processes are the vehicles for distributed memory parallelism
- Threads (light-weight processes) are the vehicles for shared-memory parallelism



Jobs, Processes, and Hardware

- Message Passing Interface – MPI, referred to as the communication layer
- 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
- OpenMP is usually activated via a compiler option, MPI is usually activated via the compiler name
- 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 processors?
  - 4 MPI processes, each with 4 threads

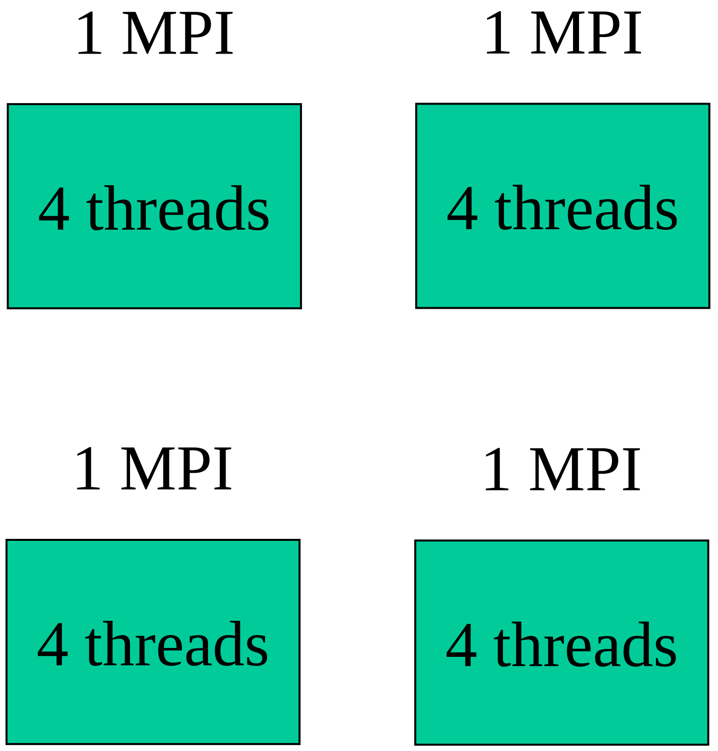
```
setenv OMP_NUM_THREADS 4
mpirun –np 4 wrf.exe
```

  - 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

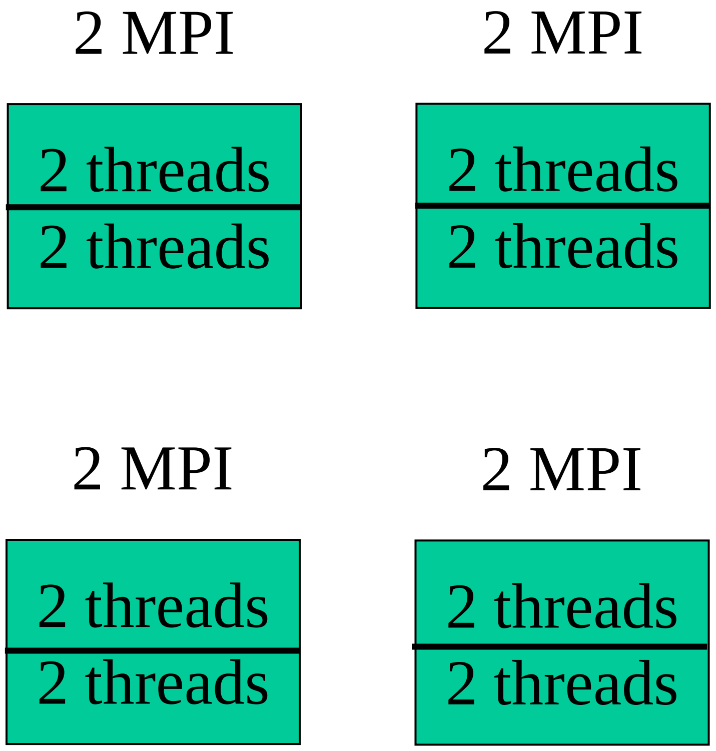
```
setenv OMP_NUM_THREADS 4
mpirun –np 4 wrf.exe
```

  - 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

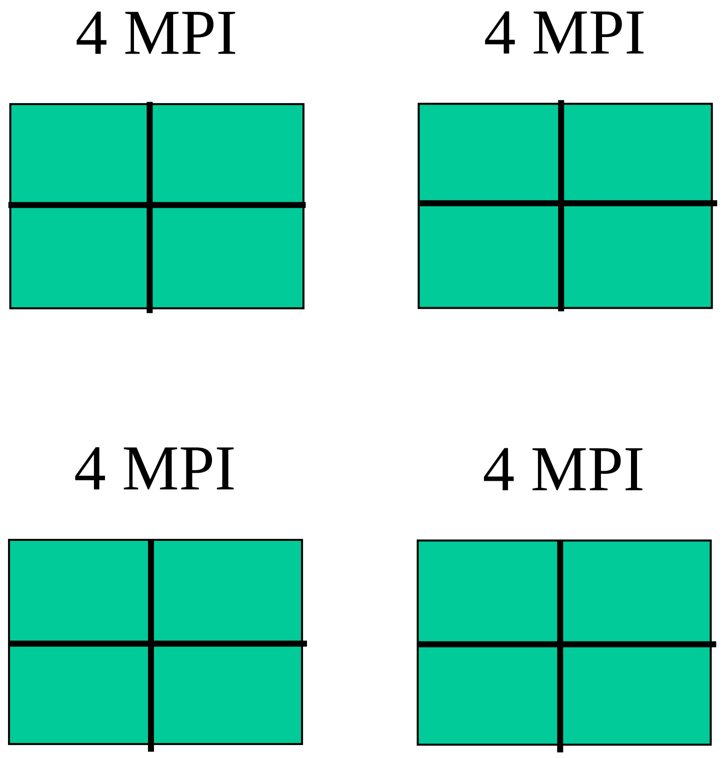
```
setenv OMP_NUM_THREADS 4
mpirun –np 4 wrf.exe
```

  - 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
```

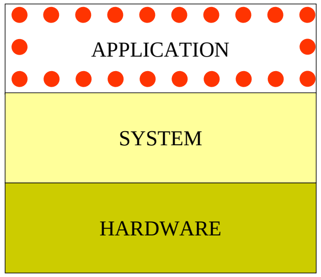


Examples (cont.)

- Note, since there are 4 nodes, we can never have fewer than 4 MPI processes because nodes do not share memory

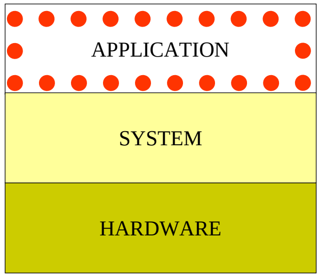
- What happens on this same machine for the following?

```
setenv OMP_NUM_THREADS 4
mpirun –np 32
```

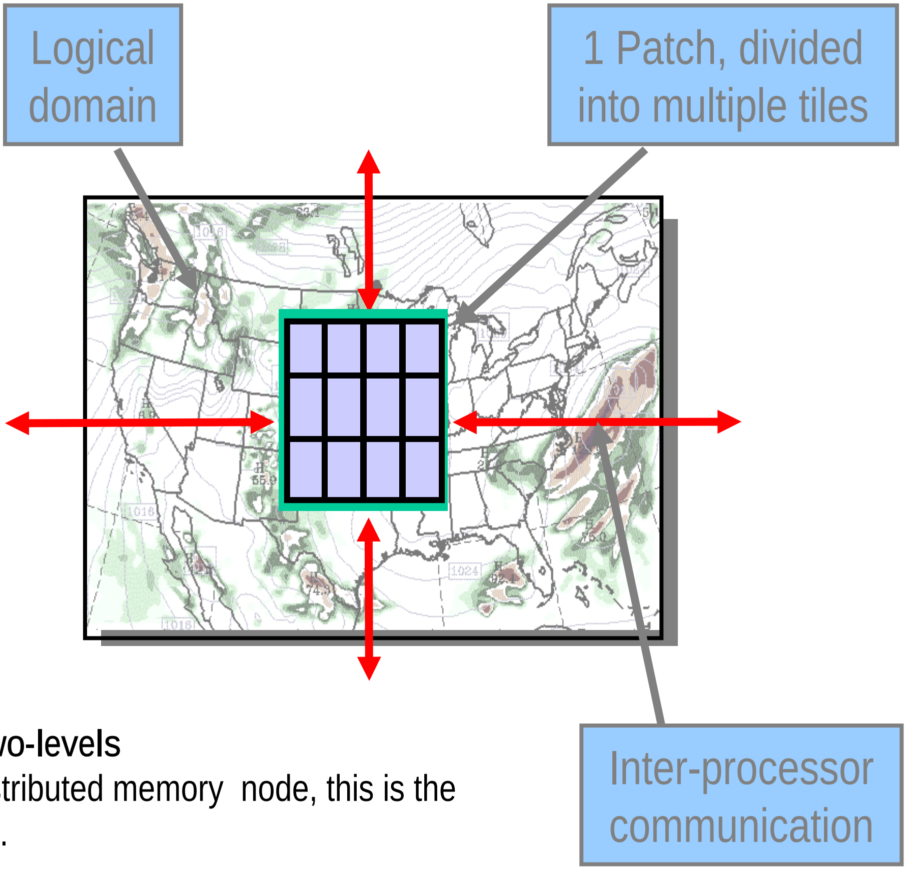


Application: WRF

- WRF can be run serially or as a parallel job
- WRF uses *domain decomposition* to divide total amount of work over parallel processes
- Since the process model has two levels (heavy-weight and light-weight = MPI and OpenMP), the decomposition of the application over processes has two levels:
  - The *domain* is first broken up into rectangular pieces that are assigned to heavy-weight processes. These pieces are called *patches*
  - The *patches* may be further subdivided into smaller rectangular pieces that are called *tiles,* and these are assigned to *threads* within the process*.*



Parallelism in WRF: Multi-level Decomposition


Logical domain
1 Patch, divided into multiple tiles
Inter-processor communication

- 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

Communication is required between patches when a horizontal index is incremented or decremented on the right-hand-side of an assignment. On a patch boundary, the index may refer to a value that is on a different patch.

Following is an example code fragment that requires communication between patches

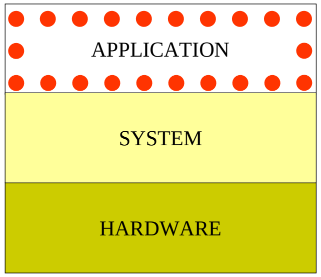
Note the tell-tale +1 and –1 expressions in indices for **rr**, **H1,** and **H2** arrays on right-hand side of assignment.

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

Distributed Memory Communications

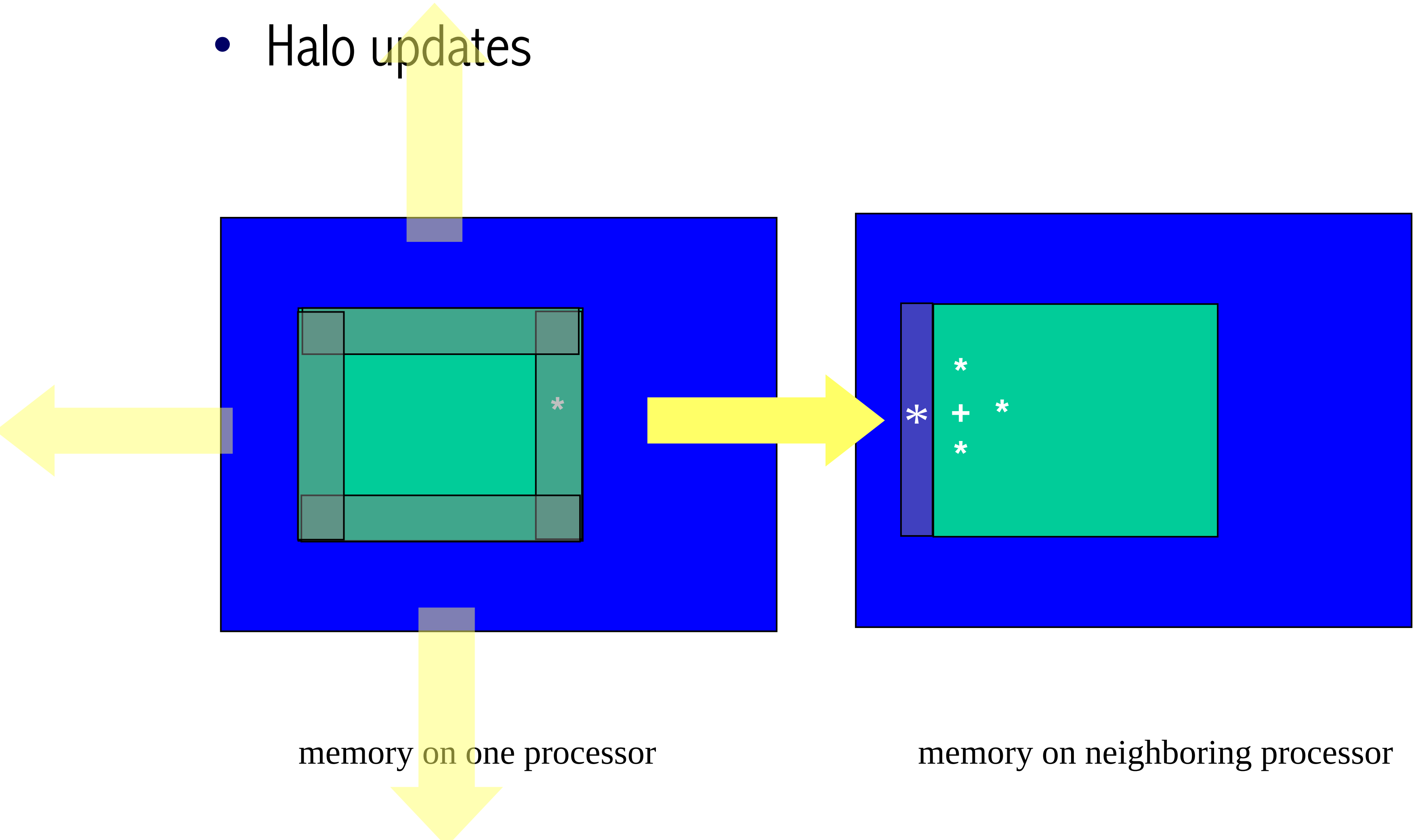
```
                       (module_diffusion.F )

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



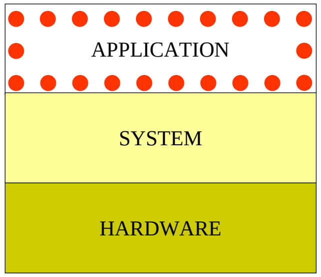
Distributed Memory MPI Communications

- Halo updates



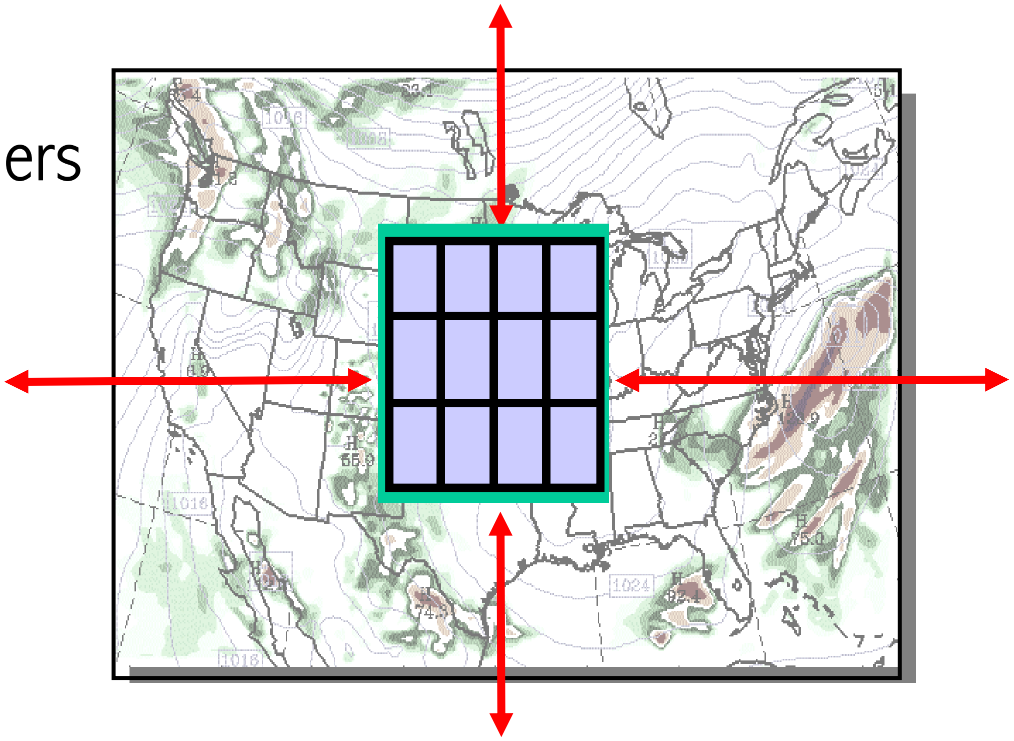
memory on one processor

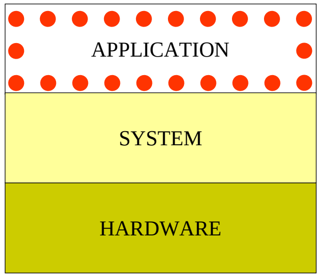
memory on neighboring processor



Distributed Memory (MPI) Communications

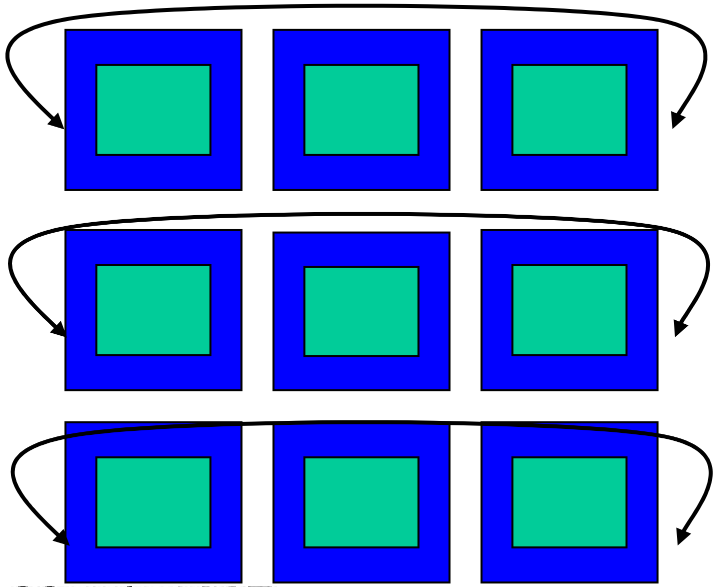
- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



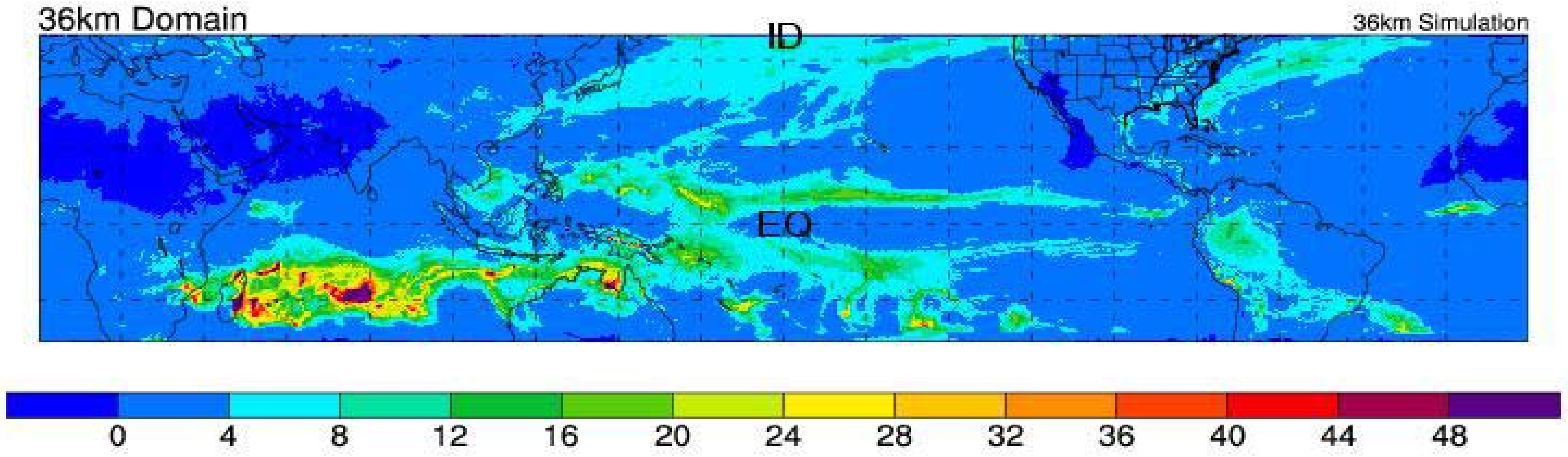


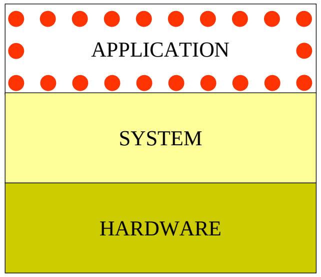
Distributed Memory (MPI) Communications

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



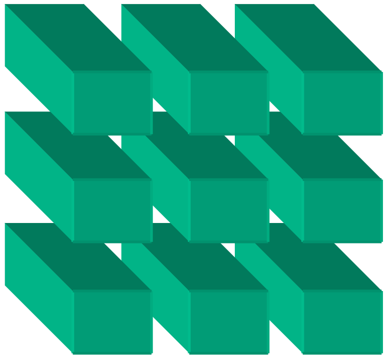
Average Daily Total rainfall (mm) - March 1997



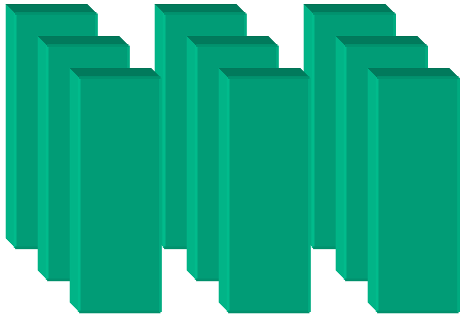


Distributed Memory (MPI) Communications

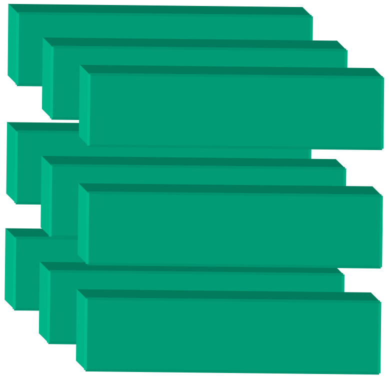
- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



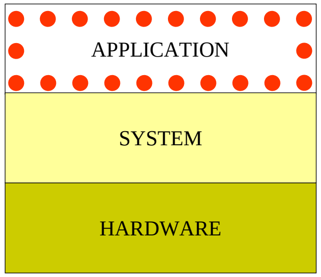
all y on patch



all z on patch

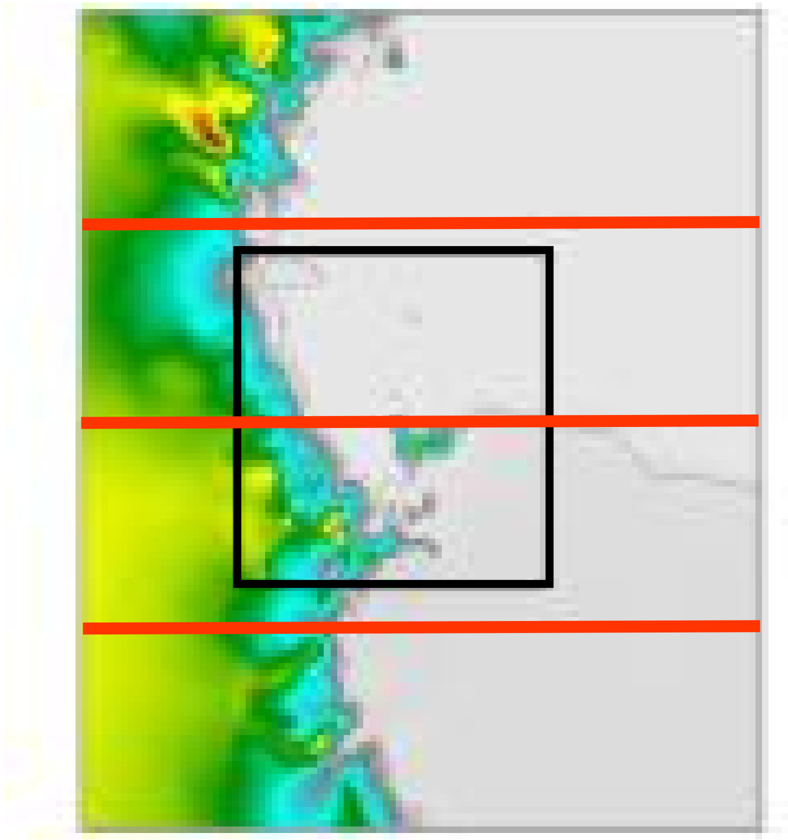


all x on patch

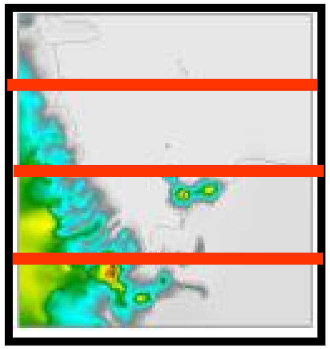


Distributed Memory (MPI) Communications

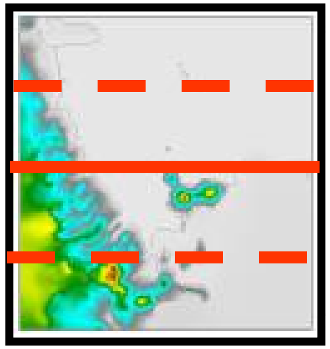
- Halo updates
- Periodic boundary updates
- Parallel transposes
- Nesting scatters/gathers



COARSE
Ross Island
6.66 km



NEST:2.22 km



INTERMEDIATE: 6.66 km

Review – Computing Overview

			Distributed Memory Parallel		Shared Memory Parallel
APPLICATION (WRF)	Domain	*contains*	Patches	*contain*	Tiles
SYSTEM (UNIX, MPI, OpenMP)	Job	*contains*	Processes	*contain*	Threads
HARDWARE (Processors, Memories, Wires)	Cluster	*contains*	Nodes	*contain*	Processors

Outline

- Introduction
- Computing Overview
- WRF Software Overview

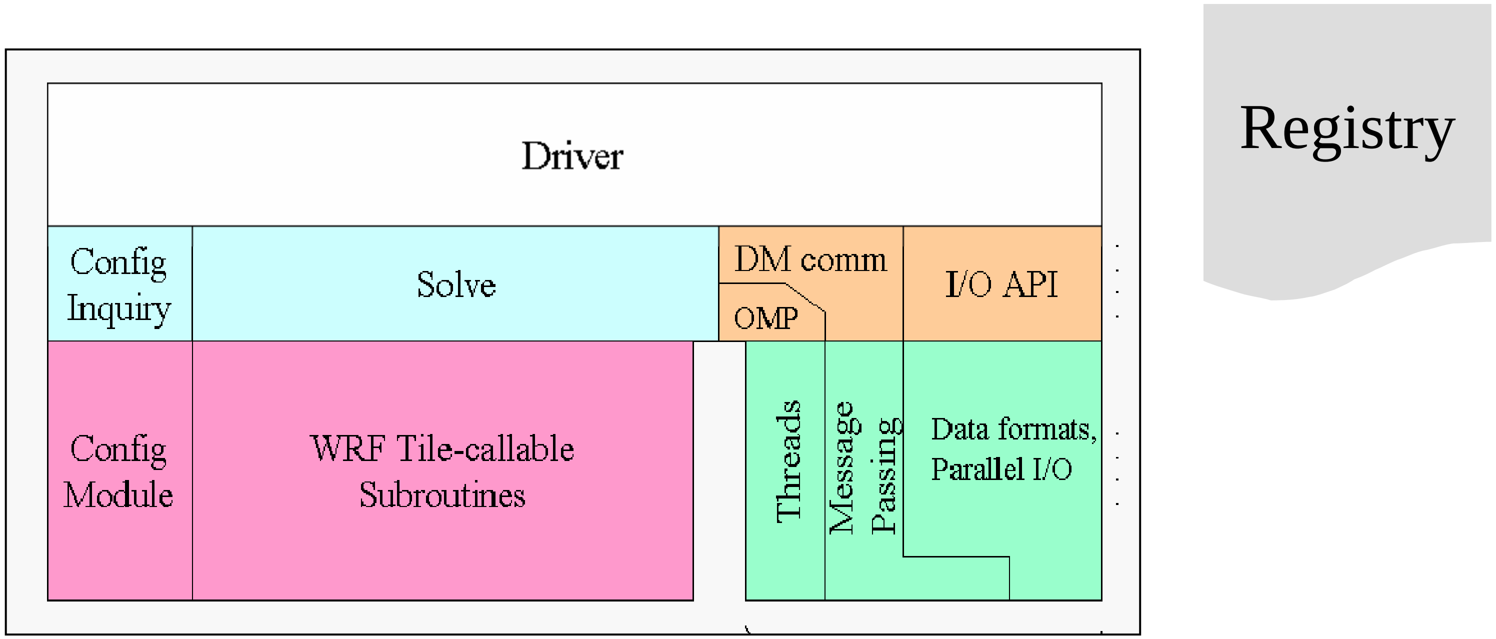
  - - - - - - - - - - - - -

- Examples

WRF Software Overview

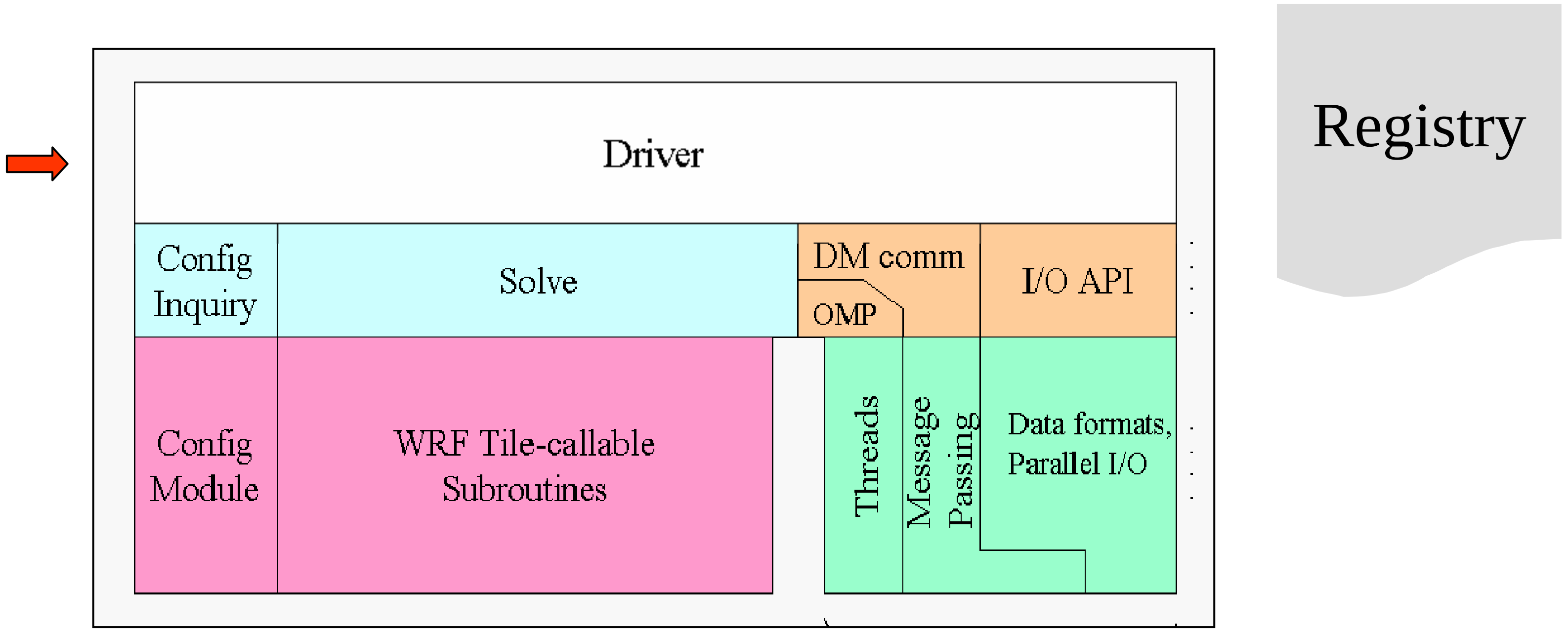
- Architecture
- Directory structure
- Model Layer Interface
- Data Structures
- I/O
- Registry

WRF Software Architecture



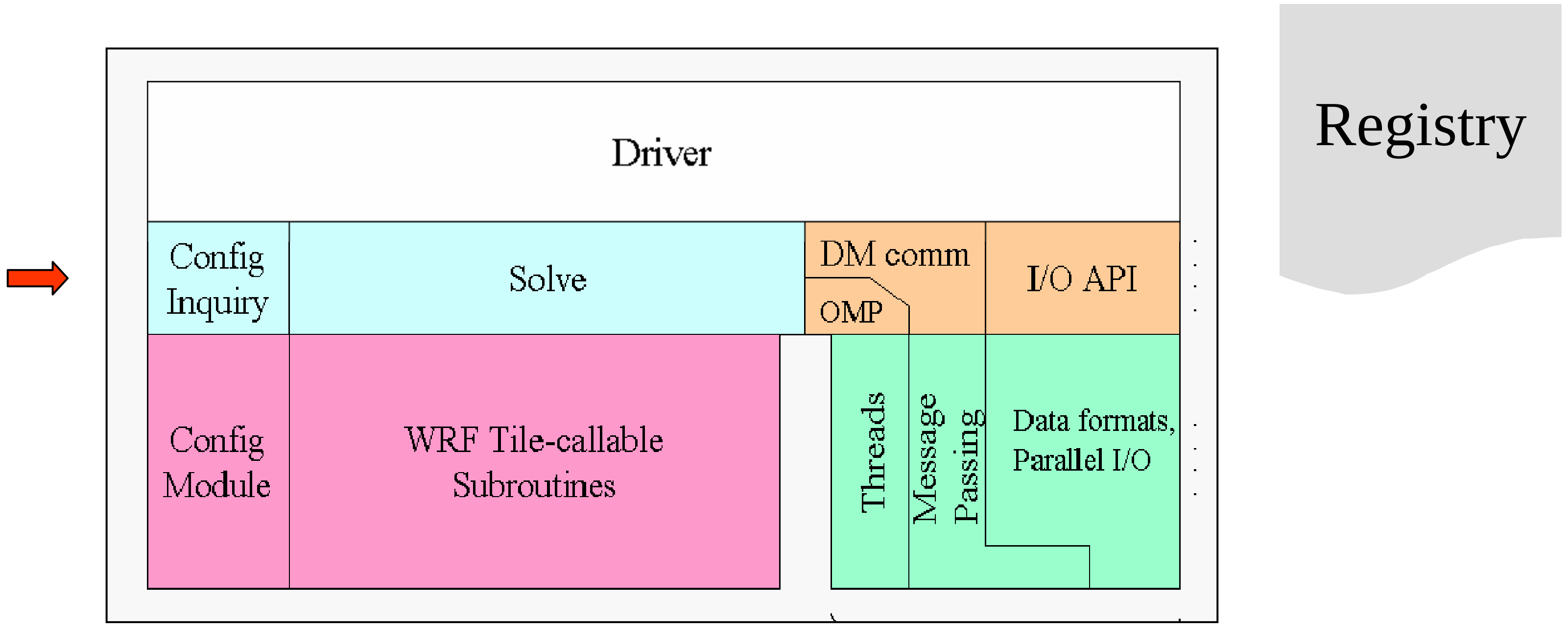
- Hierarchical software architecture
  - Insulate scientists' code from parallelism and other architecture/implementation-specific details
  - Well-defined interfaces between layers, and external packages for communications, I/O, and model coupling facilitates code reuse and exploiting of community infrastructure,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

WRF Software Architecture



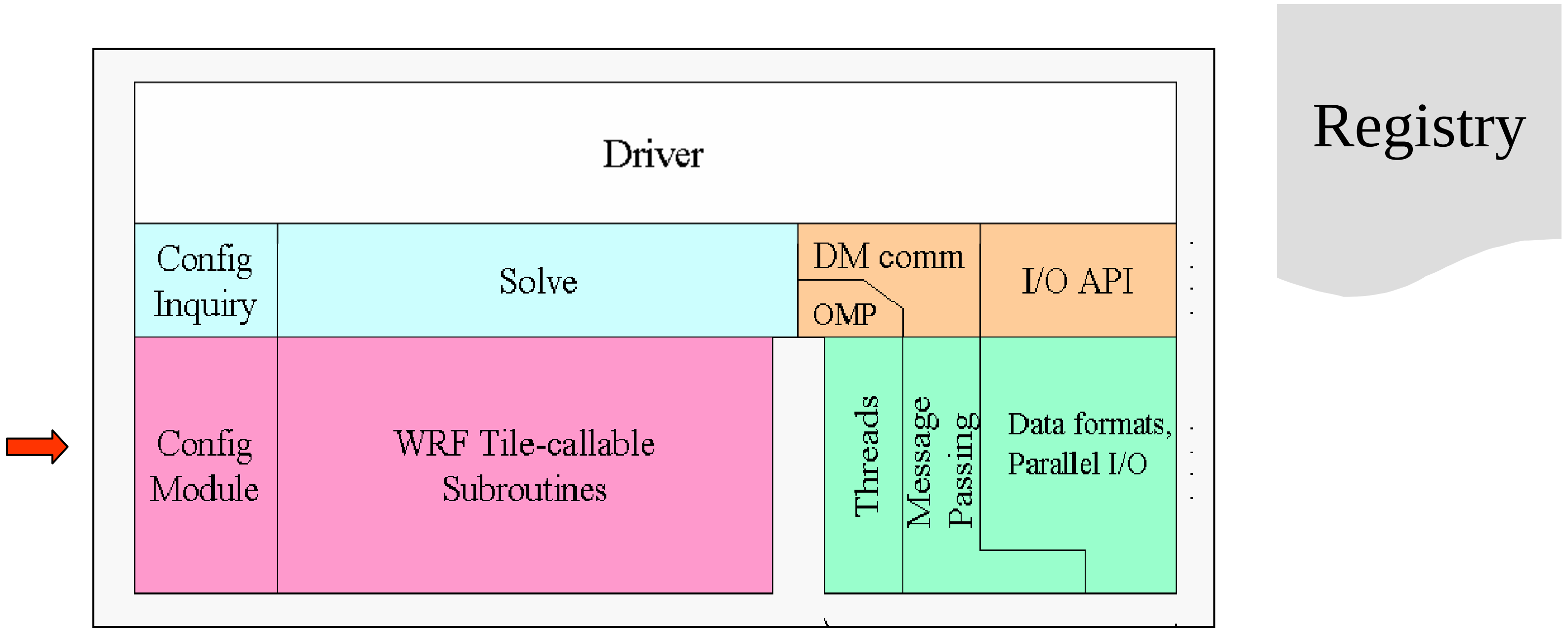
- Driver Layer
  - Allocates, stores, decomposes model domains, represented abstractly as single data objects
  - Contains top-level time loop and algorithms for integration over nest hierarchy
  - Contains the calls to I/O, nest forcing and feedback routines supplied by the Mediation Layer
  - Provides 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



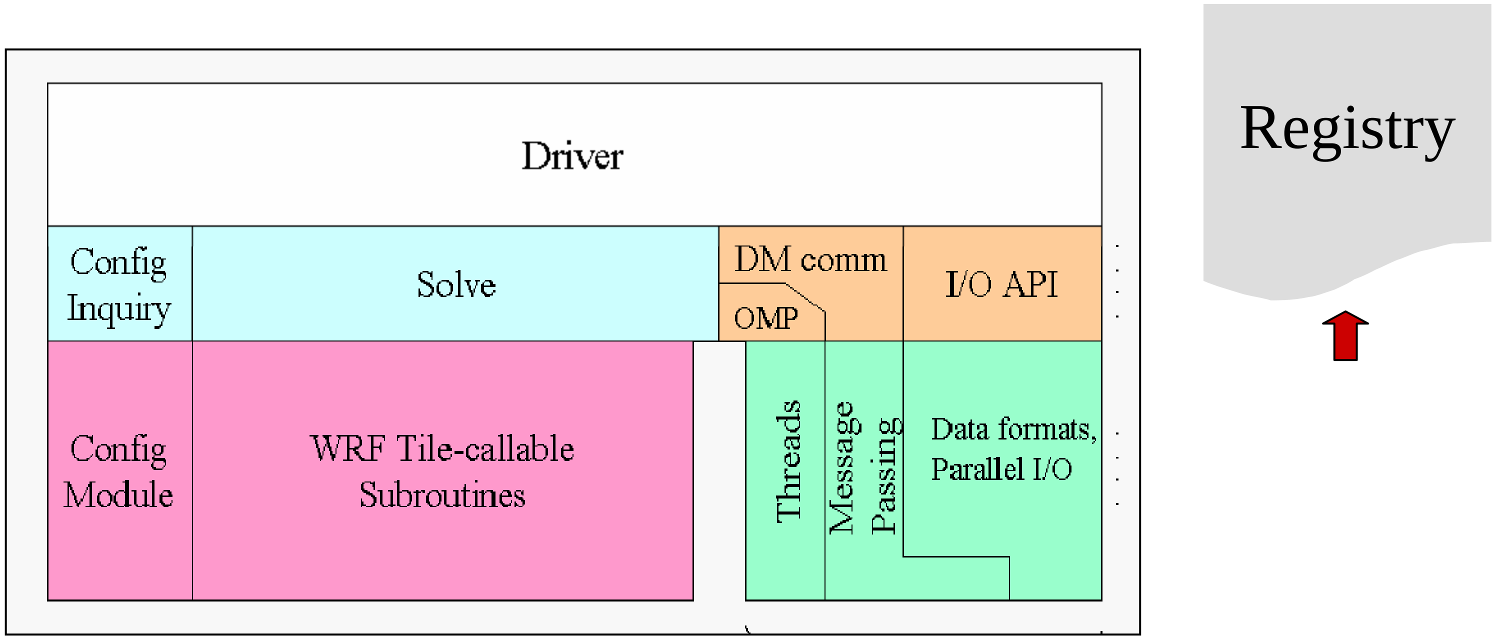
- Mediation Layer
  - Provides to the Driver layer
    - Solve routine, which takes a domain object and advances it one time step
    - I/O routines that Driver calls when it is time to do some input or output operation on a domain
    - Nest forcing, interpolation, and feedback routines
    - The Mediation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



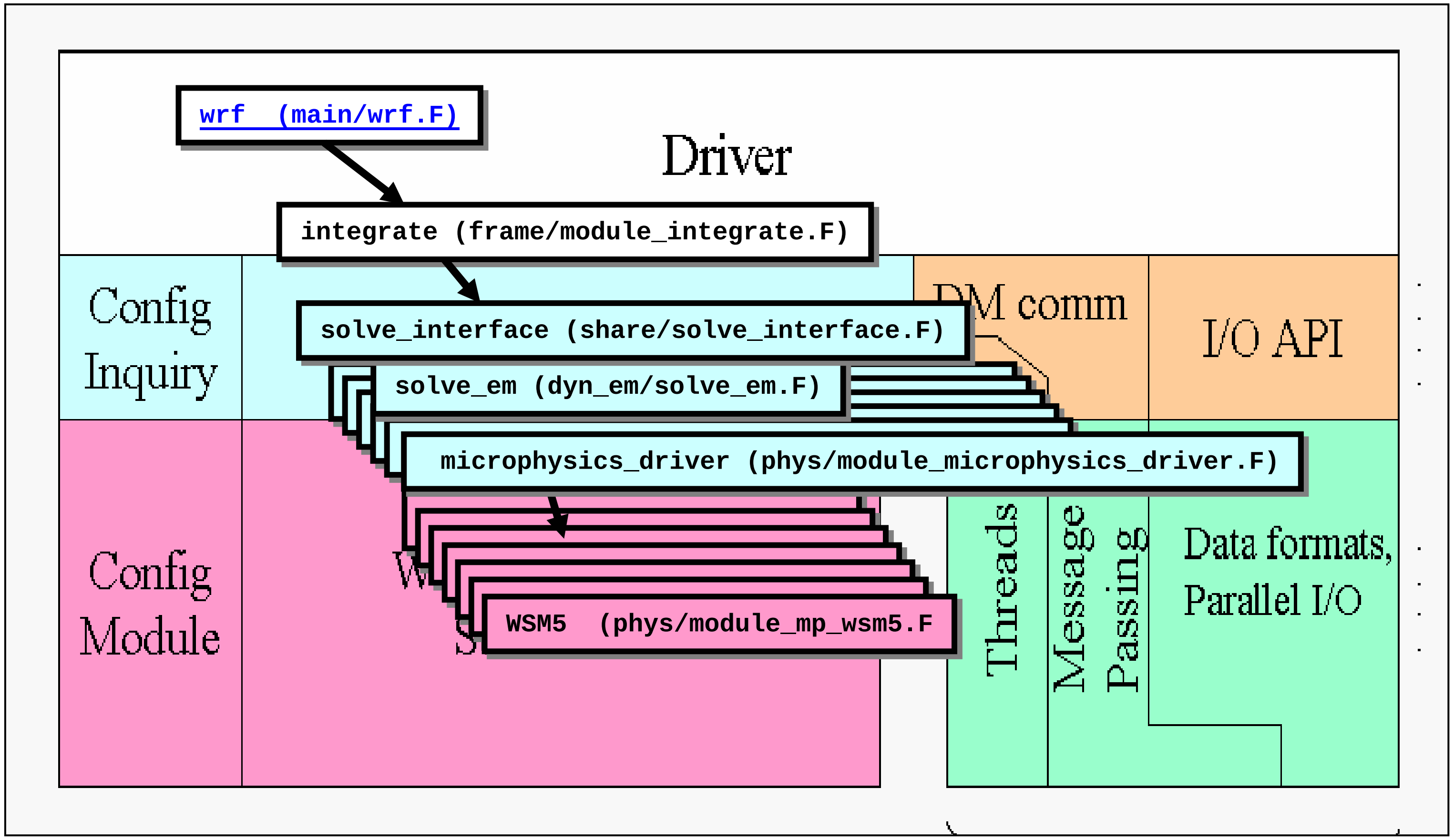
- Model Layer
  - Contains the information about the model itself, with machine architecture and implementation aspects abstracted out and moved into layers above
  - Contains the actual WRF model routines are written to perform some computation over an arbitrarily sized/shaped subdomain
  - All state 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 <u>never</u> 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: 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

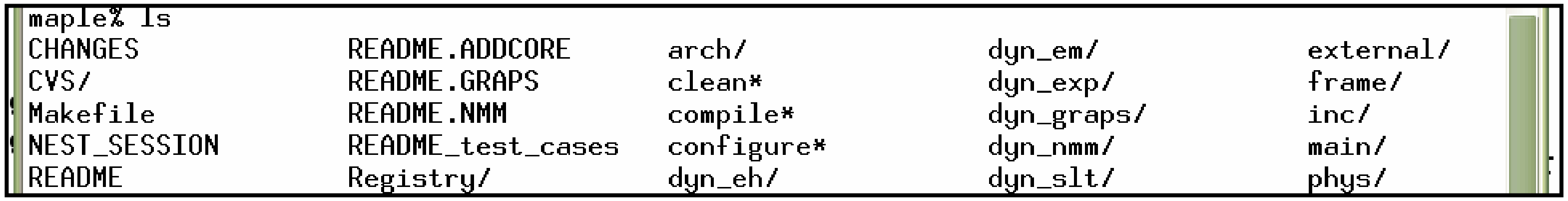
Call Structure superimposed on Architecture


wrf (main/wrf.F)
Driver
integrate (frame/module_integrate.F)
Config Inquiry
DM comm
I/O API
solve_interface (share/solve_interface.F)
solve_em (dyn_em/solve_em.F)
microphysics_driver (phys/module_microphysics_driver.F)
Config Module
WSM5 (phys/module_mp_wsm5.F
Threads
Message Passing
Data formats, Parallel I/O

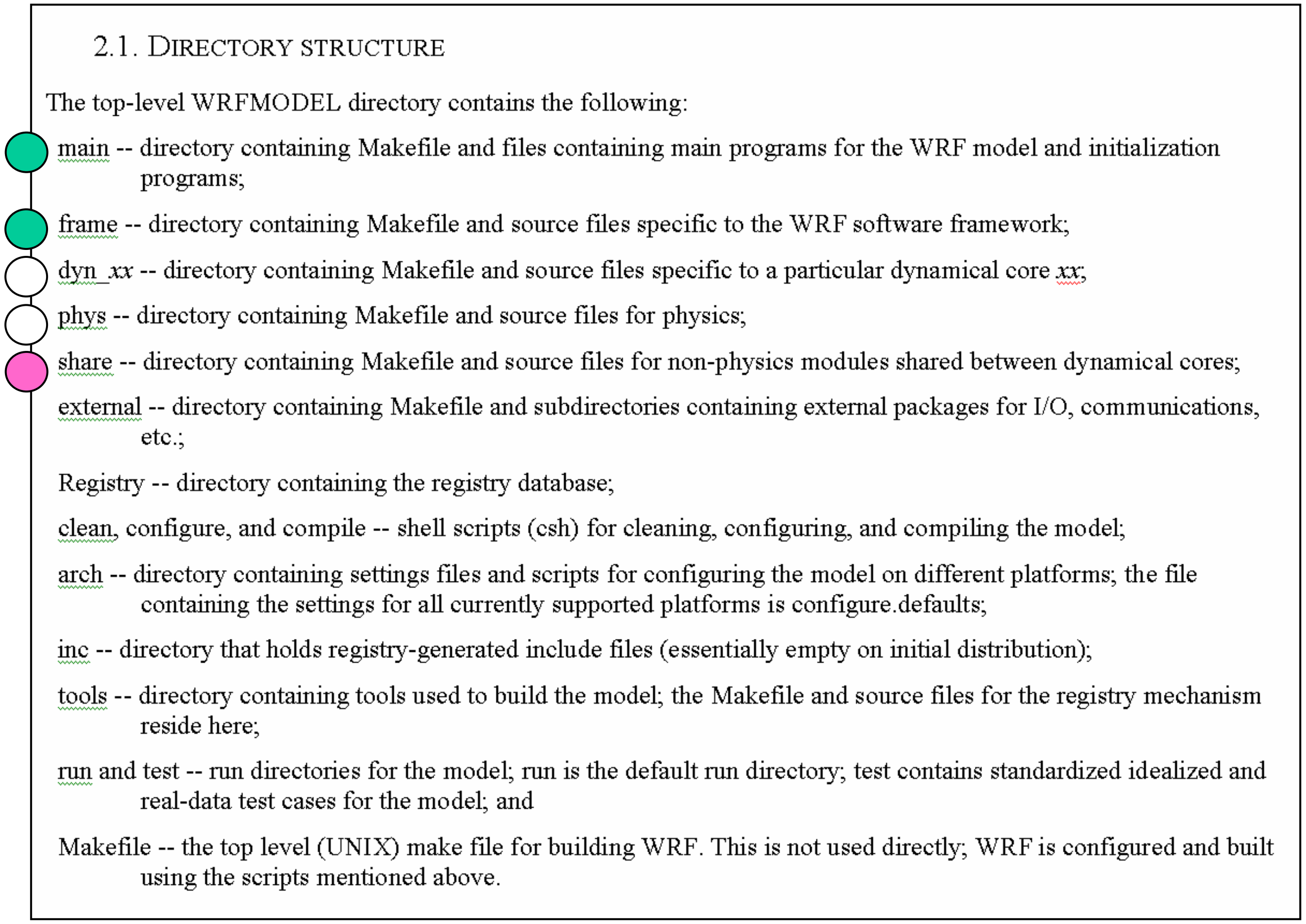
WRF Software Overview

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

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.

● driver
● mediation
○ model

page 5,
WRF D&I Document

WRF Software Overview

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

WRF Model Layer Interface

- Mediation layer / Model Layer Interface

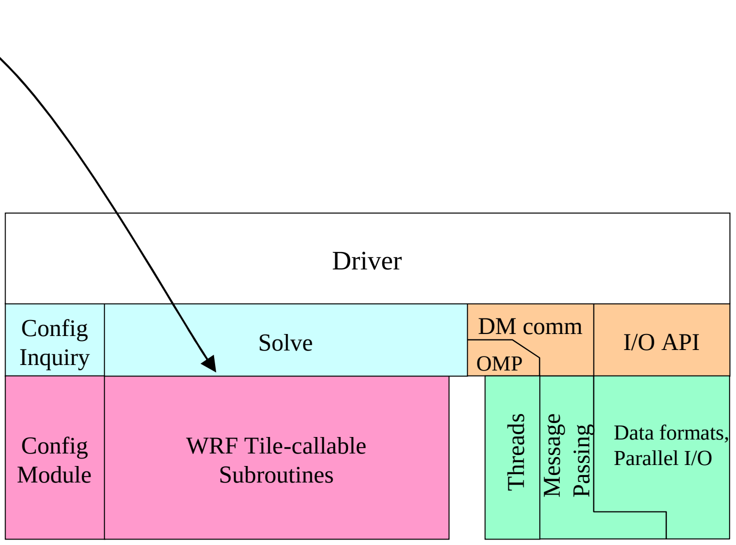
  All state arrays passed through argument list as simple (not derived) data types

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

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

  Restrictions on model layer subroutines

  - No I/O, communication, no stops or aborts (use wrf_error_fatal in frame/module_wrf_error.F)
  - No common/module storage of decomposed data (exception allowed for set-once/read-only tables)
  - Spatial scope of a Model Layer call is one “tile”
  - Temporal scope of a call is limited by coherency



WRF Model Layer Interface

- 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
```

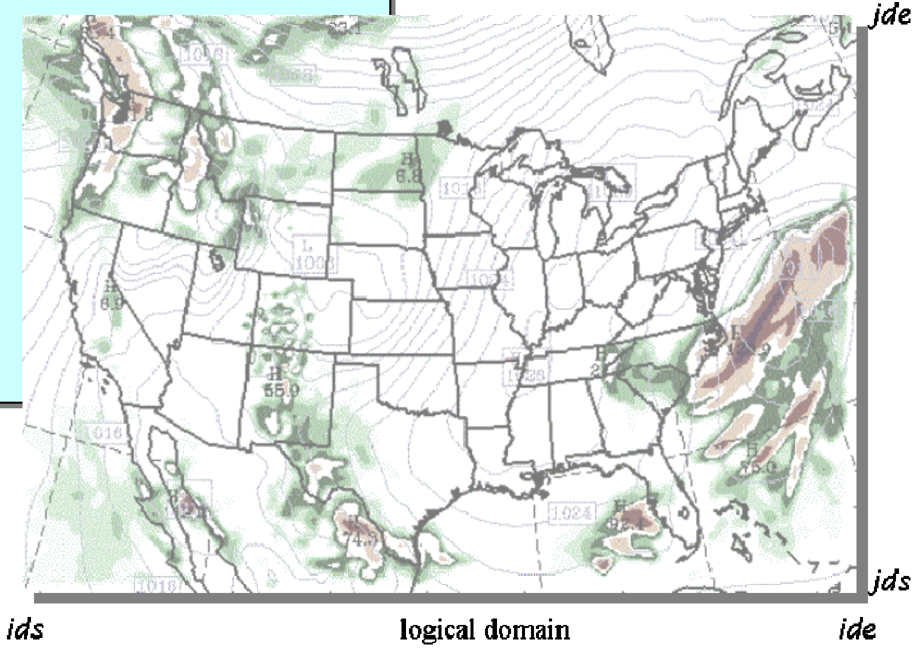
template for model layer subroutine

```
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.



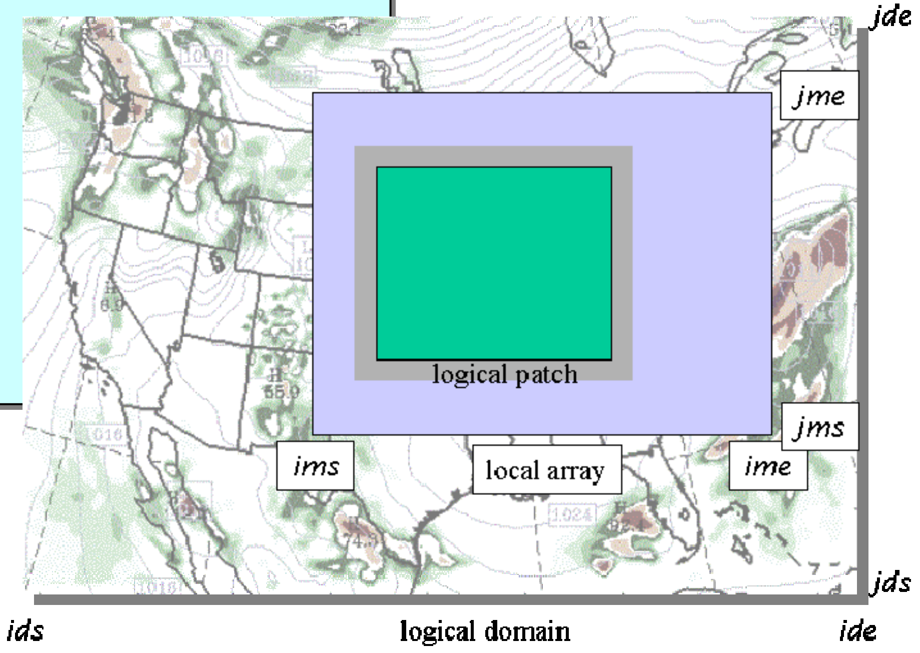
template for model layer subroutine

```
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



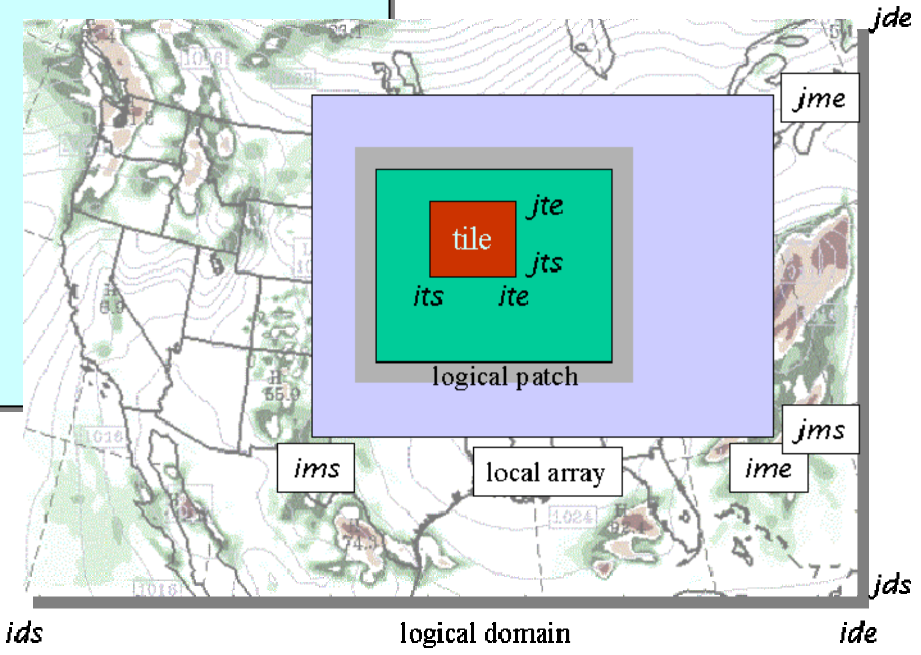
template for model layer subroutine

```
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



template for model layer subroutine

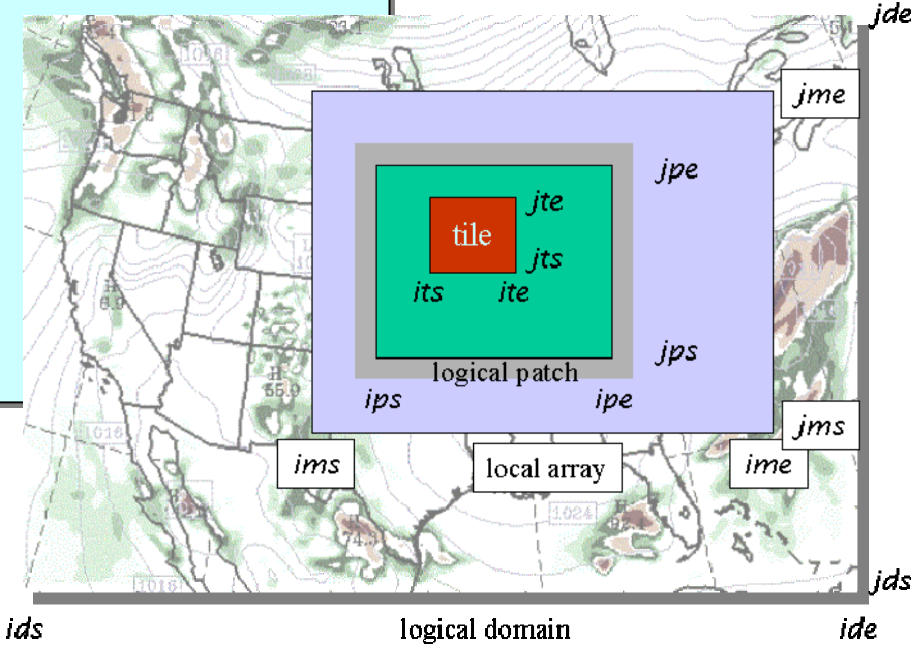
```
SUBROUTINE model ( &
  arg1, arg2, arg3, … , argn,   &
  ids, ide, jds, jde, kds, kde, &  ! Domain dims
  ims, ime, jms, jme, kms, kme, &  ! Memory dims
  its, ite, jts, jte, kts, kte  )  ! Tile dims

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

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

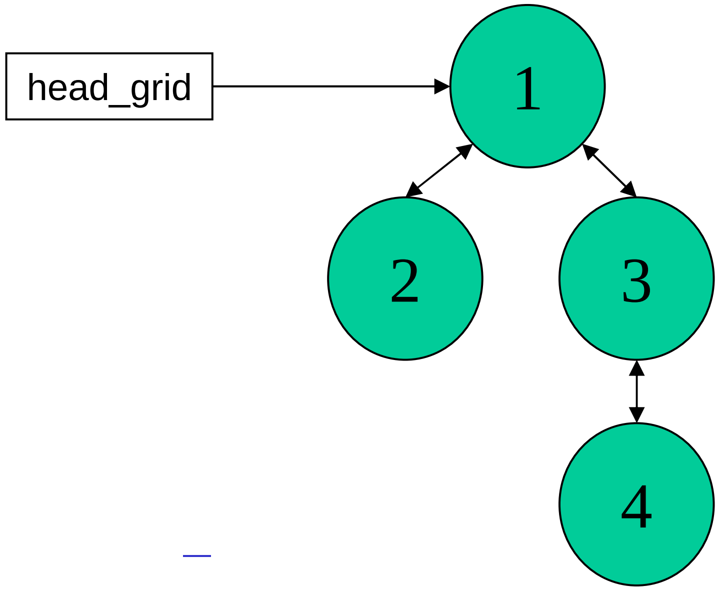


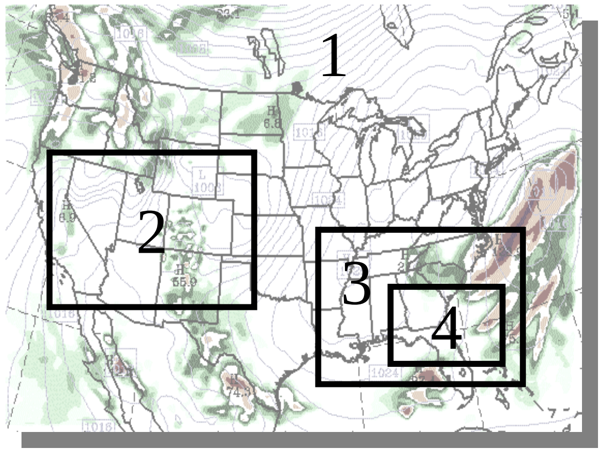
WRF Software Overview

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

Driver Layer Data Structures: Domain Objects

- 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

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

Mediation/Model Layer Data Structures: State Data

- Persist for the duration 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

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
- In the case of the NMM dynamics (E-grid) neither the IDEth nor JDEth index is ever used – logically all computations run from JDS..JDE-1 and IDS..IDE-1 or IDS..IDE-2 (depending on value of J index)

WRF Software Overview

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

WRF I/O

- Streams: pathways into and out of model
  - History + 5 auxiliary output streams
  - Input + 5 auxiliary input streams
  - Restart and boundary
- 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)
    - Specified for a stream at run time in the namelist
  - 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 Software Overview

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

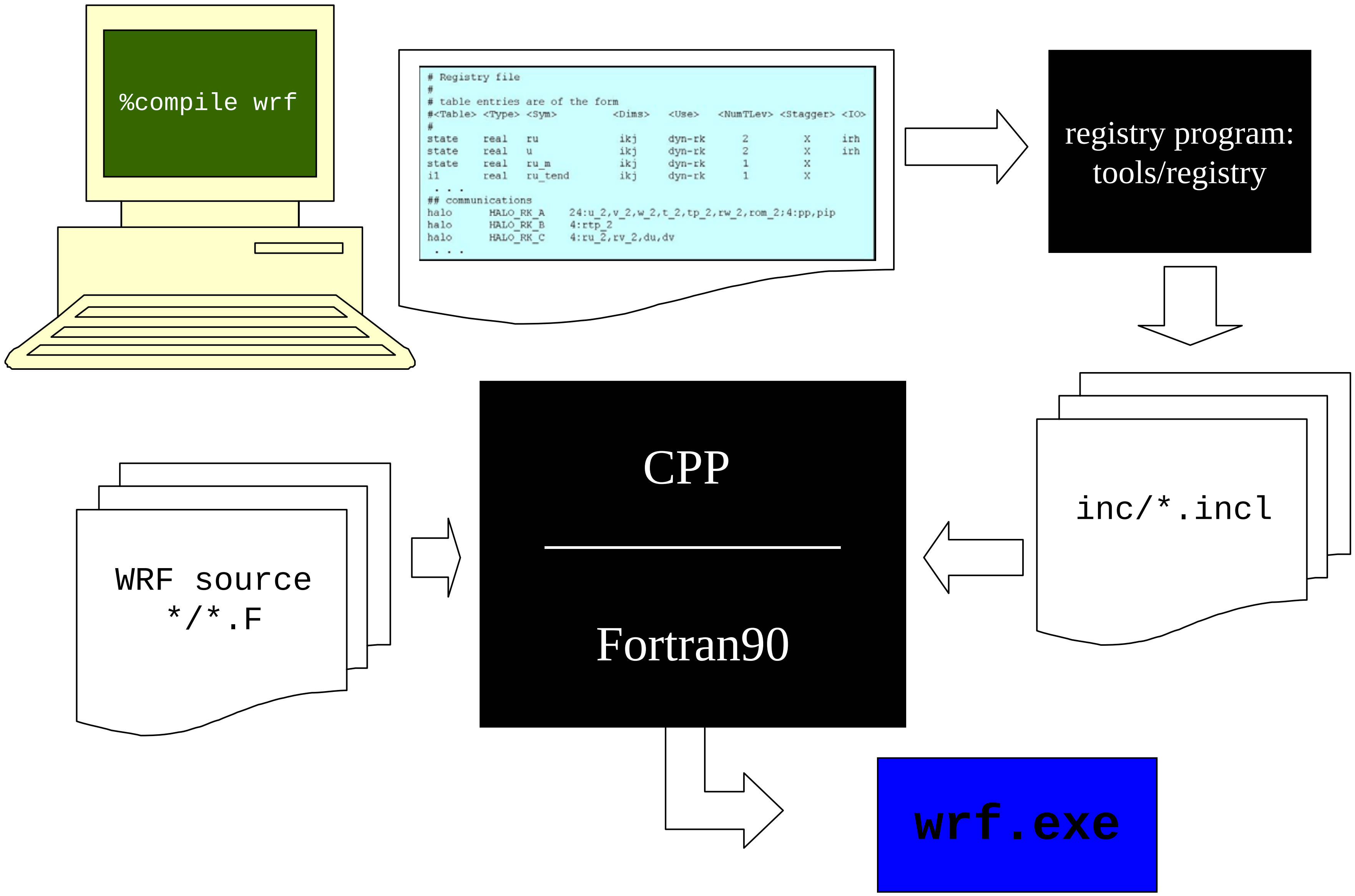
WRF Registry

- "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:
    - <u>570</u> Registry entries ⇒ <u>30-thousand</u> 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

WRF Registry

- Why?
  - 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 file
#
# table entries are of the form
#<Table> <Type> <Sym>          <Dims>   <Use>    <NumTLev> <Stagger> <IO>
#
state    real   ru              ikj     dyn-rk       2          X     irh
state    real   u               ikj     dyn-rk       2          X     irh
state    real   ru_m            ikj     dyn-rk       1          X
i1       real   ru_tend         ikj     dyn-rk       1          X
 . . .
## communications
halo      HALO_RK_A    24:u_2,v_2,w_2,t_2,tp_2,rw_2,rom_2;4:pp,pip
halo      HALO_RK_B    4:rtp_2
halo      HALO_RK_C    4:ru_2,rv_2,du,dv
 . . .
```

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

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

```
#      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

State Entry: Defining a variable-set for an I/O stream

- Fields are added to a variable-set on an I/O stream 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.

State Entry: 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.

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

```
#          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

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

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

Rconfig entry

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

- Result of this Registry Entry:
  - 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,
    . . .
 /
```

Package Entry

- Elements
  - *Entry*: the keyword “package”,
  - *Package name*: the name of the package: e.g. “kesslerscheme”
  - *Associated rconfig choice*: the name of a rconfig variable and the value of that variable that choses this package
  - *Package state vars*: unused at present; specify hyphen (-)
  - *Associated* 4D *scalars*: the names of 4D scalar arrays and the fields within those arrays this package uses

```
# specification of microphysics options
package   passiveqv     mp_physics==0    -       moist:qv
package   kesslerscheme mp_physics==1    -       moist:qv,qc,qr
package   linscheme     mp_physics==2    -       moist:qv,qc,qr,qi,qs,qg
package   ncepcloud3    mp_physics==3    -       moist:qv,qc,qr
package   ncepcloud5    mp_physics==4    -       moist:qv,qc,qr,qi,qs

# namelist entry that controls microphysics option
rconfig   integer     mp_physics   namelist,namelist_04     max_domains    0
```

Examples: working with WRF software

Add a new physics package with time varying input source 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        &
     &        ,SMFR3D=smfr3d,KEEPFR3DFLAG=keepfr3dflag                  &
     &                                                                )
```

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: Working with WRF Software

- Computing and outputting 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 )
```

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

```
      --- 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.
     . . .
```

Example: Compute a Diagnostic (complete)

```
   INTEGER        :: idex, jdex
   REAL           :: sum_ws, max_ws, glat, glon, wind_vel
   LOGICAL, EXTERNAL :: wrf_dm_on_monitor
   CHARACTER*256  :: outstring
    . . .
! 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
! 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 )
! Deterimine if i,j point of maximum is on this process
! and if so, set the lat and lon of that point, otherwise
! set to an absolute minimum
   IF ( ips .LE. idex .AND. idex .LE. ipe .AND.  &
        jps .LE. jdex .AND. jdex .LE. jpe ) THEN
      glat = xlat(idex,jdex)
      glon = xlat(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 )
! 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
```