Using the Distributed-Memory Parallel Version of ARPS

Yunheng Wang, Daniel Weber, Ming Xue

February 15, 2006

Version 1.0

Table of Content

Tables

1 Introduction

The distributed-memory parallel (DMP) version of the ARPS model is based on the MPI (Message Passing Interface) standard. Although the details of MPI implementation are vendor specific, the ARPS source is the same as long as the MPI library follows the MPI-1 standard or later. The DMP version of ARPS uses domain decomposition in the horizontal (not in the vertical) with each processor assigned one subdomain or patch, and all processors perform similar tasks on individual subdomains. The method belongs to the SIMD (Same Instructions Multiple Data) paradigm. The DMP ARPS has been tested successfully on many platforms that support MPI, including CRAY, NEC, SGI 2000, IBM SP, and Alpha based servers as well as Linux clusters with 32 bit and 64 bit Intel and AMD processors.

This document details the structure of the ARPS MPI implementation, the procedures of code compilation and job submission. The ARPS package also provides additional tools for splitting and combining the ARPS history and other data files. Besides the ARPS simulation model, the parallel capability is also implemented in several pre-processing and post-processing programs found in the ARPS package, such as ADAS, *ext2arps, arps2wrf*, *wrf2arps*, *arpsextsnd*, *arpsverif* etc. This document also outlines the specific feature in each of those DMP programs. At last, Simple queue scripts are provided for submitting ARPS parallel jobs on several platforms found locally at the University of Oklahoma and other national supercomputing centers such as the Pittsburgh Supercomputing Center (PSC), San Diego Supercomputing Center (SDSC) and National Center for Supercomputing Applications (NCSA).

2 ARPS MPI API

Message passing within the supported ARPS software is accomplished via calls to the ARPS MPI wrappers developed locally and contained in source file *src/arps/mpisubs.f90*. To wrap MPI subroutines has several benefits. The ARPS MPI interface simplifies the message passing processes within the model and avoids direct exposure of the code developer to the MPI libraries. The ARPS software contains both the serial and parallel capabilities without pre-processing, providing a single source code for development and diagnosing purposes. The code sharing between sequential and parallel models is realized by introducing several dummy subroutine declarations in file *src/arps/nompsubs.f90*. The Unix script, ***makearps***, provided with the ARPS package selects the appropriate source file for compilation, linking and executable generation.

The MPI wrappers provided in the ARPS package can be categorized as followings:

- MPI environment and variable initialization, finalization and miscellany: *mpinit_proc, mpinit_var, mpexit, inctag, mpbarrier, mpsendr, mprecvr, mpsendi, mprecvi, mpbcastr*

- Global reducing operations:
  mpupdater, mpupdatei, mpupdatec, mpupdatel, mptotal, mptotali, mpmax0, mpmax, mpmaxi, mpsumr, mpsumdp

- Inner boundary data exchange:
  mpsendrecv2dew, mpsendrecv2dns, mpsendrecv1dew, mpsendrecv1dns, mpsendrecv1diew, mpsendrecv1dins, mpsendrecvextew, mpsendrecvextns

- Model I/O Operations
  mpimerge1dx, mpimerge1dy, mpimerge2d, mpimerge2di, mpimerge3d, mpimerge2di, mpimerge4d, mpisplit1dx, mpisplit1dy, mpisplit2d, mpisplit2di, mpisplit3d, mpisplit3di, mpisplit4d, mpimerge2dns, mpimerge2dew, mpisplit2dns, mpisplit2dew

The first category is used to initialize the MPI specific variables and environment at beginning of the code execution and to exit the MPI environment upon a stop command. In addition, they provide several subroutines which are direct interfaces to the primary MPI subroutines, such as *mpsendr, mpsendi, mpbcastr*.

The second category includes subroutines to perform global reduction and broadcasting operations, such as those to compute the global maximum/minimum values or to obtain a global sum.

The third category contains the primary data exchange operations for the ARPS MPI enabled software. At every time step or small time step and at locations in which intermediate data are required, the inner processor boundary values (only North, East, West and South) are exchanged with the neighboring processors to update the "halo" or "fake" zone grid points.

The fourth category (available in ARPS Version 5.1.0 or later) provides I/O support for joining and/or splitting ARPS data files using a single process (process 0 is set as the root processor) to finalize the split or joined file. Note that data is sent or received from the other processors to the root process which performs the final I/O operation.

The ARPS software defines a set of global variables to support distributed memory applications and these variables are described and declared in file *include/mp.inc*. The most commonly used MPI related parameters are *mp_opt* & *myproc*. The *mp_opt* defines the parallel or sequential status and the parameter *myproc* indicates the rank of the local process starting from 0 to *nprocs*-1 for *nprocs* processes. The ARPS parallel software specifies the processor with rank 0 as the root process, *i.e. myproc* = 0.

Content of file "include/mp.inc"

```
!-----------------------------------------------------------------
!  Message passing variables
!-----------------------------------------------------------------

  INTEGER :: mp_opt     ! Message passing option
                        ! = 0, no message passing
                        ! = 1, use message passing option.
  INTEGER :: nprocs     ! Number of processors.
  INTEGER :: nproc_x    ! Number of processors in x-direction.
  INTEGER :: nproc_y    ! Number of processors in y-direction.
  INTEGER :: nproc_x_in ! The nproc_x specified in the input file.
  INTEGER :: nproc_y_in ! The nproc_y specified in the input file.
  INTEGER :: loc_x      ! Processor x-location (1 to nproc_x).
  INTEGER :: loc_y      ! Processor y-location (1 to nproc_y).

  INTEGER :: myproc     ! Processor number (0 to nprocs-1).
  INTEGER :: max_proc   ! Maximum number of processors.
  PARAMETER (max_proc=20000)
  INTEGER :: proc(max_proc)  ! Processor numbers.

  INTEGER :: max_fopen  ! Maximum number of files allowed open.
  INTEGER :: gentag     ! message tag number
  INTEGER :: joindmp    ! History dump format
                        ! = 0, dump file for each processor
                        ! = 1, dump one joined file.
  INTEGER :: readsplit  ! External data file read option
                        ! = 0, each processor do its own read
                        ! = 1, do split on-the-fly
  INTEGER :: readstride ! = nprocs     if readsplit == 1
                        ! = max_fopen  otherwise
  INTEGER :: dumpstride ! = nprocs     if joindmp == 1
                        ! = max_fopen  otherwise

  COMMON/arpsc005/mp_opt,nprocs,nproc_x,nproc_y,loc_x,loc_y, &
            myproc,proc,max_fopen,gentag,joindmp,readsplit,  &
            nproc_x_in,nproc_y_in, readstride, dumpstride

  INTEGER :: tag_w,   tag_e,   tag_n,   tag_s
  PARAMETER (tag_w=11,tag_e=12,tag_n=13,tag_s=14)
  INTEGER :: tag_sw,   tag_se,   tag_nw,   tag_ne
  PARAMETER (tag_sw=15,tag_se=16,tag_nw=17,tag_ne=18)
```

The parameters in COMMON block *arpsc005* are global parameters and they will be set by the program automatically via calls to subroutines *mpinit_var* & *mpinit_proc* and based on the domain decomposition configuration specified when running the model (see the section about runtime settings below). The variables specified below the *arpsc005* COMMON block are constants used to create MPI message tags and they can only used when the file *mp.inc* is included explicitly in the source code.

3 Compiling the Parallel Model

A Unix script, **makearps**, provides the user with a common interface to perform the compile and link operations across all ARPS supported platforms. The script can be used to compile the forecast model (ARPS) or several other programs including EXT2ARPS, ADAS and ARPSPLT etc. The **makearps** script can be augmented by user supplied options which specify the compile and link options to external libraries. The **makearps script** will search for MPI library and header files. Since the location of those files are platform dependent, **makearps** accepts two options, "-mp_inc *mpi_include_path*" & "-mp_lib *mpi_library_path*", to specify the locations for the include file and MPI library files, respectively. The default include file and library paths are "*/usr/include*" and "*/usr/library*", respectively. Those paths have already been configured correctly in file ***makearps*** to work on several local platforms at the University of Oklahoma and other distributed memory systems at several national supercomputing centers. The current supported platforms (at the time of this document is writing) are listed in Table 1.

Table 1. ARPS MPI-Supported Computing Platforms

Platform	Center	Cluster Name	Architecture	Note
modi*	NCSA		SGI Origin 2000	
tun*	NCSA	tungsten	Xeon Linux Cluster	
co-login*	NCSA	cobalt	SGI Altix Itanium2 Cluster	
ds*	SDSC	datastar	IBM P655/P690	64 bit mode
tg-login*	PSC	bigben	Cray XT3 MPP machines	
iam*	PSC	lemieux	HP Alphaserver Cluster	
tcsini	PSC		Compaq Tru64 Unix Cluster	
schooner	OSCER	schooner	Itanium2 Linux Cluster	
boomer	OSCER	boomer	Pentium4 Xeon Linux Cluster	
sooner	OSCER	sooner	IBM p690 Regatta	
paige	CAPS	paige	SGI Origin 2000	
weather /regatta1 /regatta2	WDT	williams		
chaos	WDT		chaos Linux cluster	
squall	WDT		1450 4 proc Linux cluster	
mac	Va Tech System		Mac OS X cluster	
tasc	TASC		Compaq Alpha server	LAM 6.5.6 MPI
ecas	ECAS	ecas		
cray			Cray Y-MP Cray C90	(obsolete)
nec			NEC SX-5 Vector machines	(obsolete)

The Unix/Linux command to prepare, compile and link a parallel ARPS application is:

```
$> ./makearps [options] arps_mpi
```

A parallel capable executable is created in subdirectory *bin/* within ARPS root directory. The MPI-related options for script **makearps** are (please refer to the ARPS User's Guide for other options):

```
-mp_inc mpi_include_path
-mp_lib mpi_library_path
-m      machine_type
-io     bin|net|hdf|nohdf
```

The machine type is usually extracted from the C shell environment variable "$HOSTTYPE" as a first attempt to matching the platform type. If the environment variable "$HOSTTYPE" is not set in the current shell, output from command "*/bin/uname*" is used to guess your machine type. **makearps** also accepts option "-m *mach*". You can specify the machine type explicitly with this option if the script does not properly identify the machine architecture. The currently supported ARPS MPI machine types are:

Table 2. **ARPS Supported Computer Architectures**

rs600	IBM RS/6000 machines
mac	Mac OS X machines
cray	Cray class machines (Cray Y-MP and Cray C90)
t3e	T3E class machines (NOT T3D)
hi-ux	Hitachi HI-UX machines
sun4	Sun4 machines
alpha	DEC Alpha machines
iris4d	SGI (iris4d) machines
linux	Linux machines
nec	NEC SX-5 Vector machines

To modify the compile options of a particular machine type, you can make changes directly in file *makearps*. You can first locate the option block for a particular machine type by searching string "$*mach* == *type*", where "*type*" is a string listed above, and change the definitions in that block. A new block for an unsupported machine may also be added. You can base your work on any available option blocks.

To create the utilities for splitting input data files into patches for individual processors, use command

$> ./**makearps** [options] *splitfiles*

To create the utilities for joining the output history dumps, use command

$> ./**makearps** [options] *joinfiles*

and/or

$> ./**makearps** [options] *joinfile*

(see below).

4 Runtime Settings

Both the sequential and the parallel version of ARPS read the same runtime configuration via a standard input (see chapter 4 in the ARPS User's Guide). In addition to the parameters used for model execution, the NAMELIST input file *arps.input* also contains a number of parameters that are used only by the parallel version. These parameters can be set at run time without modifying or recompiling the model code. One important feature of the ARPS parallel implementation is that only one processor, the root processor, reads the standard input and broadcasts the input file data to the other processors.

This section first introduces the horizontal domain decomposition feature of the ARPS parallel model, as well as the staggered grid structure of the Arakawa C grid used by the ARPS model. Then hints for setting those parallel specific parameters are provided. It also highlights the relationship between the model dimension parameters when running parallel version of the model.

4.1 Domain Decomposition and ARPS Grid implementation

The ARPS model employs the mode-splitting time integration technique introduced by Klemp and Wilhelmson (1978), *i.e.* the large time step is used to compute advection and mixing terms and the small time step used to compute the new wind velocity and pressure due to sound waves. The large time step integration uses the leap-frog time differencing scheme. The small time step integration uses a Crank-Nicholson scheme which solves the vertical velocity (w) and pressure (p) equations either implicitly, using a tri-diagonal solver, or explicitly in the vertical direction. The vertical implicit solution technique used to solve w and p also is an option for updating variables associated with the vertical subgrid scale mixing. Furthermore, many of the physical processes such as radiation and cumulus parameterization require column-wise computations that are non-local in the vertical direction. Therefore, the domain decomposition strategy used in the ARPS model is carried out efficiently in the horizontal direction. With the fourth-order advection and/or numerical diffusion, five grid points are involved in a single time step in each horizontal direction. However, we chose to implement these calculations in two steps, each involving only three grid points, so that only one "fake" zone is needed at subdomain boundaries. Of course, data in the single "fake" zone has to be updated after each of these two steps, implying communication between processors.

The ARPS domain decomposition scheme is illustrated in Figure 1. Figure 1a contains 4 processes with a 2x2 processor configuration. Figure 1b shows the inner processor relationship between 3 processors in x-direction and the same inner processor relationship holds for multiple processors in y-direction.

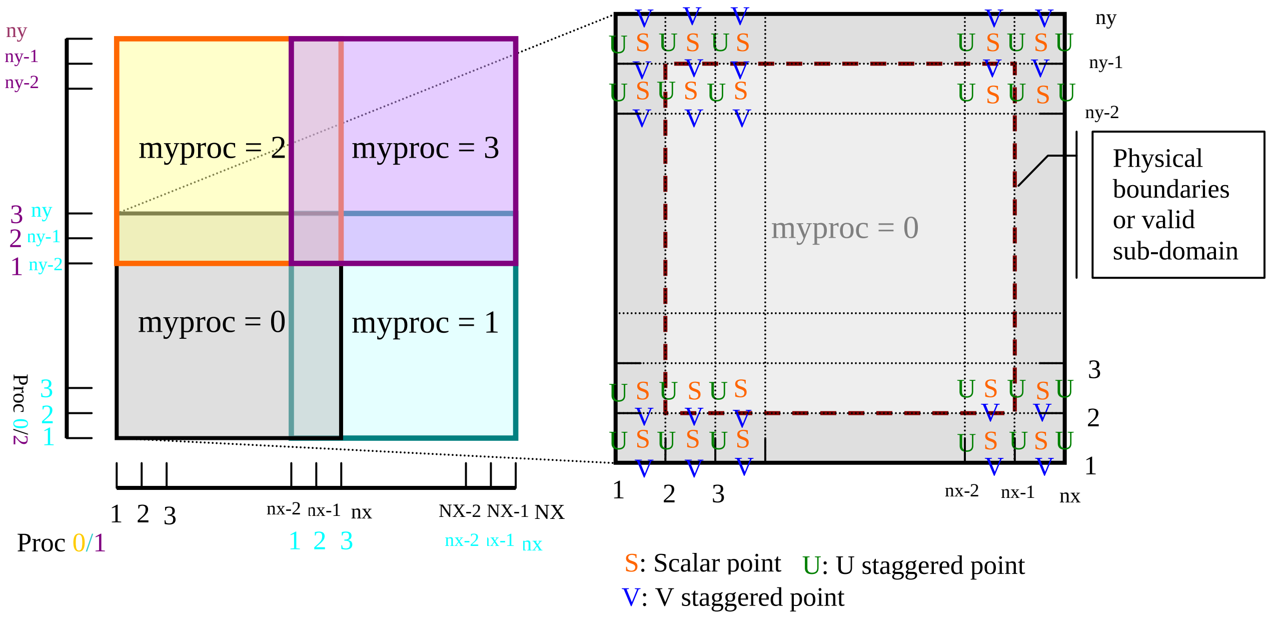


Figure 1a. Illustration of ARPS domain decomposition with a 2x2 processor configuration

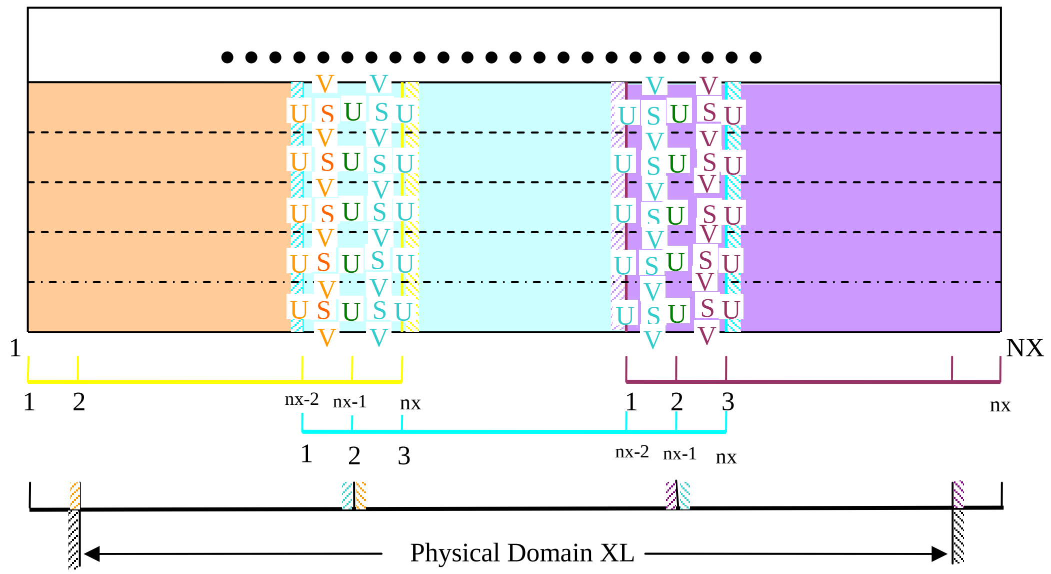


Figure 1b. Illustration of inner processor relationship in X direction

The ARPS horizontal grid is defined over a staggered Arakawa C grid with one more fake point ("halo" or "fake" zone) for the scalars and velocity components outside the physical boundaries (2 – *nx*-1 in the *x*-direction, 2 – *ny*-1 in the *y*-direction) to facilitate the implementation of boundary conditions and the message passing between neighboring processes. The length of the model physical dimension within each processor or during a single processor application is *XL*=(*nx*-3)d*x* (*YL*=(*ny*-3)d*y*) in the *x* and *y* directions, respectively.

4.2 NAMELIST Parameter Configurations

The most significant parameters for the parallel ARPS model are the global domain grid size *nx*, *ny* and the values of *nproc_x* & *nproc_y* which represent the number of processors in the X and Y direction, respectively. The total number of processors used in a parallel run is *nproc_x***nproc_y*. The values of *nx* and *ny* represent the ARPS global grid size for both the parallel and sequential runs. However, the choice of the global number of grid points is not arbitrary for parallel runs. For load balance purposes, the size of the global number of grid points in the *x* and *y* directions must be divisible by *nproc_x* and *nproc_y*, respectively. For the case in which the global number of grid points is not divisible by *nproc_x* or *nproc_y*, the size of the global domain will be adjusted to become divisible by *nproc_x* (*nproc_y*) in the *x*-direction (*y*-direction) automatically. The grid size per processor, as shown in Figure 1 and 2, has the following relationship with the global grid dimensions:

$$global\_x\_size = (nx-3)*nproc\_x + 3$$
$$global\_y\_size = (ny-3)*nproc\_y + 3$$

Note that all sources inside the ARPS model use variables *nx* and *ny* for representing the local grid size. The only exception is in the NAMELIST input file *arps.input* where the values of *nx* and *ny* represent global dimension sizes. The model handles the conversion from global grid sizes to local grid sizes automatically for the users in subroutine *initpara* (see file *src/arps/initpara3d.f90*). The user sets the values of *nproc_x* and *nproc_y* depending on the size of your model domain and the available resources on the computing platform. As a rule of thumb, the total number of processors should be a power of 2 but is not required. The total number of CPUs declared in NQS scripts or the command-line parameter to "*-np*" of **mpirun** (or "*-procs*" of **poe**) must be the same number as *nproc_x***nproc_y*. Table 3 lists all the parameters that must be set before running the parallel version of ARPS.

Table 3. MPI-related ARPS input file variables.

Variable	Meaning	Defaults
nproc_x	Number of processors in the X direction	1
nproc_y	Number of processors in the Y direction	1
max_fopen	Maximum number of files allowed open	8
inisplited	Flag indicate whether input file are split or not (see below)	0 – joined input data 1 – split input data
dmp_out_joined	Flag for history file dumping	0 – dump split files 1 – join and dump

The maximum number of open files for reading or writing must be limited to avoid saturating the network and disk sub-systems, especially on large parallel computing platforms. Users can specify the number of open files with parameter *max_fopen*. Small cluster will perform best with a *max_fopen* = 2 to 4 and large parallel computing centers have the capacity to allow up to 20-30 simultaneous writes/reads. The ARPS model, however, will set *max_fopen* = *nproc_x***nproc_y* automatically when it is demanded to read in a sounding file for *initopt* = 1, no matter what value is set for *max_fopen* in file *arps.input*.

After ARPS version 5.1 and later, a new feature was implemented in the ARPS model to split the input data and join the output data on-the-fly. With this new feature, it is not necessary to split the data files before starting the parallel model, or to join the data files after the model simulation (assumes that this is a model run and not ext2arps etc) is completed. If variable *inisplited* = 1 is defined within the ARPS input file, the input data files, including files specified by parameters: *inifile*, *inigbf*, *exbcname*, *terndta*, *sfcdtfl*, *soilinfl*, *rstinf* etc. (see chapter 4 of the ARPS User's Guide), were split using the utility **splitfile** (see below). It is the original behavior of the parallel ARPS model before version 5.1.0. However, if *inisplited* = 0, the auto-split feature newly implemented in the ARPS model will split the input data. With this new feature, the root processor (processor with rank 0) will read the input data files that contain data covering the global domain and then split the read-in data into smaller patches and distribute those patches to their corresponding processors.

Similarly, the flag *dmp_out_joined* controls the joining of data for writing. If *dmp_out_joined* = 1, data patches from all processors are collected by the root processor for joining and writing to a single file. The joined data file contains data from the global domain. If *dmp_out_joined* = 0, the ARPS data files will be written out in split file form (one file per processor) and can be joined at a later time using **joinfiles** (see below).

5 Running ARPS on Parallel Machines

Depending on the runtime parameters specified in the input file, running a parallel simulation of the ARPS model may involve three steps, *i.e.* splitting the data files, running the model and joining the data files. On some platforms, the parallel ARPS can be run through an MPI startup script, such as **mpirun**, **mpiexec**, or **poe** etc. On supercomputers or clusters with job scheduling capabilities, it is required to submit your MPI application to a Network Queuing System (NQS) or scheduler, such as the Portable Batch System (PBS) scheduler, Load Sharing Facility (LSF) or the Loadleveler job management system. This section also provides several script samples for job submission on those scheduler systems.

5.1 Splitting the data files

If *inisplited* = 1 in the ARPS input file, the external data files must be split prior to the execution of the parallel ARPS. The ARPS package provides a utility **splitfiles** to split ARPS data files. **Splitfiles** will read the NAMELIST input file for filename information and split the files one by one. The command is:

$> bin/**splitfiles** < *arps.input*

Note that **splitfiles** only supports unformatted binary files and HDF 4 files at present. The split files have the same filename as the input files except that their names are appended with a 4 digit number denoting its location in the global processor grid. For example, with the configuration in Figure 1a, we will get four ARPS initial files which correspond to the 4 processors used, *arps_test.bin000000_0101*, *arps_test.bin000000_0102*, *arps_test.bin000000_0201* and *arps_test.bin000000_0202*.

To split HDF-4 files manually without providing the NAMELIST input, you can also run command

```
$> bin/splithdf
```

and the program will interactively ask you for the HDF 4 filename to be split.

5.2 Running the model

To execute the parallel version of the ARPS model, the user can, if the system allows, submit the parallel ARPS executable, *arps_mpi* through a MPI submission script, such as **mpirun**, **mpiexec**, and **poe** on IBM/AIX systems or **prun** for the HP Alpha servers on PSC. For example, to perform a 16 processor ARPS simulation on a SGI 2000 or a PC running Linux system:

```
$> mpirun –np 16 bin/arps_mpi < arps.input > arps.output
```

As we have mentioned before, it is required that only one processor (the root processor with rank 0) reads the standard input channel for configuration parameters. The option "-stdinmode 0", available in the IBM Parallel Operating Environment, instructs POE to allow only the processor with rank 0 read the standard input. The recommended command for **poe** is:

```
$> poe bin/arps_mpi -procs 16 -eager_limit 64k -stdinmode 0 -stdoutmode
unordered -wait_mode poll -buffer_mem 256M < arps.input > arps.output
```

For instructions about those options, please refers to "POE: Parallel [Operating] Environment Documentation" from IBM.

5.3 Joining the data files

The ARPS tool **joinfiles** should be used to obtain a set of combined history data files if *dmp_out_joined* = 0 is set while running the model. The tool will read the NAMELIST input file for the *runname* information and the frequency of history writes and combine all the history files over all processor patches into a single file covering the global domain. **Joinfiles** supports unformatted binary files and HDF 4 formats.

```
$> bin/joinfiles < arps.input
```

To join one set of files at a time, you can use ARPS tools **joinfile** or **joinhdf**. The ARPS package also provides **joinbin2hdf**, which combines a complete set of binary split files into a single HDF 4 file.

```
$> bin/joinbin2hdf
```

The user will be prompted to provide the base filename.

5.4 Job Queue Scripts

Most supercomputers have an interactive job limit and queuing systems allow you to run larger and longer jobs in a non-interactive manner. On some computer systems, all applications/jobs must be submitted to the batch system/queue.

5.4.1 Network Queuing System (NQS) Script

Cray computer systems use the Network Queuing System (NQS) to schedule jobs. The job resource requirements and execution environment are defined in a shell-like script with additional keywords. An example NQS script for the obsolete Cray T3D is:

```
#QSUB -lM 7Mw -lT 15000
#QSUB -q mpp
#QSUB -l mpp p=16        # sets the number of processors
#QSUB -l mpp t=9000      # sets the computation time limit for each
                         # individual processor
#QSUB -s /usr/local/bin/tcsh
#QSUB -eo
#QSUB -o $AFS/arps.log

cd $workdir
bin/arps_mpi < arps.input >! arps.output
```

An example NQS script for the Cray T3E is:

```
#QSUB -lT 15000         # Mandatory qsub options specifying command
#QSUB -l mpp_p=2        # PE time limit and number of application PEs.
#QSUB -l mpp_t=9000     # Strongly recommended option specifying
                        # application PE time limit.
#QSUB -o t3e.output     # Optional qsub option specifying the output
                        # file name.
#QSUB -eo
```

To submit your NQS job, type:

$> **qsub** *NQS_script_file*

To check your job status, use command

$> **qstat** -a

5.4.2 Portable Batch System (PBS) Queue Script

The Portable Batch System (PBS) is a POSIX-compliant suite of commands intended to manage jobs running on multiple computer servers. Many well-known supercomputer systems are managed by the PBS scheduler, such as the Pentium4 Xeon Linux Cluster (boomer.oscer.ou.edu) at OU and the HP Alpha Cluster (lemieux.psc.edu) at PSC. A typical script is as follows:

```
#!/bin/csh
#
#PBS -l walltime=04:00:00  # Wall clock time requests
#PBS -l nodes=2:ppn=2      # Requests 2 nodes with 2 processors each
                           # works on boomer, a Linux Xeon cluster
##PBS –l rmsnodes=1:4      # On Lemieux, requests 4 processors on 1 node

#PBS -q queue_name         # Queue name, it is defautl_q on boomer,
                           # can be batch or debug on Lemieux
##PBS -A account_string    # Note that "##PBS" will be ignored by PBS
#PBS -o stdout_file_name
#PBS –e stderr_file_name
#PBS -N your_job_name

set dir_name = /home/xxxx/arps5.2
set execode  = /home/xxxx/arpss5.2/bin/arps_mpi
set inputf   = arps.input
set outputf  = arps.output

set echo

cd $dir_name

#
#  Run the model on boomer.
#
#mpirun $execode < $inputf >& $outputf

#
#  Run the model on Lemieux
#
#prun –N ${RMS_NODES} –n ${RMS_PROCS} $execode < $inputf >& $outputf
```

To submit the job,

$> **qsub** *pbs.script*

To check the job status,

$> **qstat**

To kill the job,

$> **qdel** <*jobid*>

5.4.3 IBM LoadLeveler Script

LoadLeveler is a network job management and scheduling system developed by IBM. Although the *LoadLeveler clusters* environment can include heterogeneous clusters, it is mainly used on IBM RS/6000 SP systems, such as the IBM Regatta p690 system (sooner.oscer.ou.edu) with Power4 symmetric multiprocessors (SMP) at OU. A typical NQS LoadLeveler script is:

```
#!/bin/sh
#
# Batch Job Specifications
#
```

```
# How much computing resources are needed (maximum)
# ConsumableCpus denotes number of cpu required per process
#   @ resources  = ConsumableCpus(1) ConsumableMemory(100mb)
# How many seconds of wall clock time are needed (maximum)
#   @ wall_clock_limit = 7200
# The filename (including full path) of the executable.
#   @ executable = /usr/bin/poe
#
# Command line arguments
#
#   !!! The following arguments must be in one line
#   @ arguments  = /home/xxxx/arps5.2/bin/arps_mpi -procs 4
      -eager_limit 64k -stdinmode 0 -wait_mode poll -buffer_mem 256M
#
# Files to redirect standard input, standard output and standard error
#   @ input      = /home/xxxx/arps5.2/arps.input
#   @ output     = /home/xxxx/arps5.2/arps.output.$(Cluster)
#   @ error      = /home/xxxx/arps5.2/arps.err.$(Cluster)
# Poe-related information for LoadLeveler
#   @ class      = parallel
#   @ job_type   = parallel
#   @ job_name   = your_job_name
#   @ tasks_per_node = 4
# Environment variables to pass to poe
#   @ environment = COPY_ALL; MP_HOLD_STDIN=yes; MP_SHARED_MEMORY = yes
#
# This command tells LoadLeveler to execute the command described above.
#   @ queue
```

To submit the job script to LoadLeveler,

$> **llsubmit** *loadleveler.script*

To check your job status,

$> **llq** –s <*jobid*>

To cancel a job,

$> **llcancel** <*jobid*>

5.4.4 Load Sharing Facility (LSF) Script

LSF is a general purpose distributed queuing system that can be used to unite a cluster of computers, even in a heterogeneous environment, into a single virtual system to obtain a more flexible computing resource on the network. The newly implemented Itanium2 Linux cluster (schooner.oscer.ou.edu) at OU was instrumented with the LSF queuing system. A simple LSF script looks like the following:

```
#!/bin/csh
#BSUB -q normal          # Queue name
#BSUB -a mvapich
#BSUB –x                 # Request exclusive access
#BSUB -n 4               # use 2 nodes, 2 CPU per node
#BSUB -R "span[ptile=2]" # states 2 processors per node are desired
#BSUB -o /home/xxxx/arp5.2/out.%J
#BSUB -e /home/xxxx/arp5.2/err.%J
#BSUB -u xxxx@ou.edu
```

```
#BSUB -W 01:00
#BSUB -J job_name        # Declare a job name

set wrkdir     = /home/xxxx/arps5.2
set executable = /home/xxxx/arps5.2/bin/arps_mpi
set input      = arps.input
set output     = arps.output

cd $wrkdir
mpirun.lsf $executable < $input >! $output
```

To submit the job script to LSF queue:

$> **bsub** < *lsf.script*

Remember that the script file (*lsf.script*) must be executable (**chmod** +x *lsf.script*) and the redirection (**bsub** < *lsf.script*) operator in the submittal command is required.

To display information about pending, running and suspended jobs, execute:

$> **bjobs**

To remove a job from the LFS system, execute:

$> **bkill** <*jobid*>

6 Other ARPS Programs with DMP Capability

Besides the ARPS weather simulation model, many ARPS pre-processing and post-processing programs can be run on distributed-memory platforms using the same message passing interface developed in the ARPS package. These parallel programs are ADAS, EXT2ARPS, ARPSPLT, ARPS2WRF, WRF2ARPS, ARPSEXTSND and ARPSVERIF. Because of the difference in capability and complexity of each program, each one has its own special requirements in program settings and job submission. This section describes the specific features in the each parallel version of these programs. For a general description about each program, please refer to their respective user's guides. This section also outlines the procedures of code compilation and job submission for each parallel version of the programs.

6.1 DMP version of ADAS

To be added by Kevin Thomas

6.2 DMP version of EXT2ARPS

To be added by Kevin Thomas

6.3 DMP version of ARPSPLT

ARPSPLT is a vector graphics plotting program provided within the ARPS package. ARPSPLT reads in the ARPS history format, performs various analyses and generates graphic output as 2-D fields and 1-D profiles. The program is described in Chapter 10 of the ARPS User's Guide. Since the ARPS version 5.1, the graphic package ZXPLOT on which ARPSPLT is based, is distributed within the ARPS package. The DMP capability of ARPSPLT was added since ARPS version 5.0 (IHOP_4) and later. This parallel plotting program has been fully tested during the CAPS-PSC spring forecast programs in 2005 and significant enhancements were added since then (ARPS version 5.2 and later).

6.3.1 Program Settings

The parallel version of plotting program reads the same NAMELIST input file as sequential program mentioned in the ARPS User's Guide. The MPI-specific parameters are listed in the following table.

Table 4. MPI-related ARPSPLT input file variables.

Variable	Meaning	Defaults
nproc_x	Number of processors in the X direction	1
nproc_y	Number of processors in the Y direction	1
max_fopen	Maximum number of files allowed to be opened	8
nproc_node	Number of processes allocated on one node	0
readsplit	Flag indicate whether input file are split or not (see below)	1 – joined input data 0 – split input data
nprocx_in	Number of processor for data in X direction	1
nprocy_in	Number of processor for data in Y direction	1

The parameters *nproc_x*, *nproc_y* and *max_fopen* have the same meanings as those defined in section 4.2.

There are two scenarios to run the parallel plotting program. In the first scenario, the ARPS history file(s) to be plotted is a big file that contains data covering the global domain. Then the parameter *readsplit* should be 1. It instructs the root process (with rank 0) to read the data and then split the read-in data into smaller patches and distribute those patches to their corresponding processors. With this scenario, parameters *nprocx_in*, *nprocy_in*, *max_fopen* and *nproc_node* are ignored by the program. The program will set *max_fopen* = *nproc_x*nproc_y* and *nproc_node* = 1 automatically.

In the second scenario, the program reads ARPS history data in split file form, i.e. *readsplit*=0. The number of split data files can be the same as the number of processors to be run, such that

```
readsplit = 0,
nprocx_in = nproc_x,
nprocy_in = nproc_y,
```

It means that each processor reads its own data file, no split is needed.

The program also supports a case when the ARPS history data files are generated using more processors than the number of processors for ARPSPLT run. For example, the user runs the ARPS simulation model using more processors because the simulation model is a computation-demand program. The plotting job, however, is usually much easier to be scheduled when less processor is claimed from the system. The parameter *nprocx_in* and *nprocy_in*, however, cannot be arbitrary and they must be multiples of *nproc_x* and *nproc_y*, respectively,

```
readsplit = 0,
nprocx_in = n*nproc_x,
nprocy_in = m*nproc_y,
```

where *n* and *m* are any natural numbers.

When *readsplit* = 0, the program uses either *nproc_node* or *max_fopen* to control the number of simultaneous file opened. The first case is *nproc_node* = 0 or 1, the parameter *max_fopen* is used just as it has been used in section 4 of this document. The second case is *nproc_node* > 1, the parameter *max_fopen* is ignored by the program. The parameter *nproc_node* is used to specify the number of processes allocated on each computing node. This parameter was introduced because ARPSPLT is an I/O intensive program. The large number of opened files on one node may saturate the network and the disk sub-systems, for example the nodes on the HP Alpha servers at PSC, if more than one processes are allocated on the node, especially when *nprocx_in* (*nprocy_in*) is much larger than *nproc_x* (*nproc_y*). The parameter *nproc_node*, however, is used to order ARPSPLT I/O operations only. It has no way to control the allocation of parallel processes on computing nodes. It is the PBS option "*-rmsnodes*" that controls the actual allocation of processes on nodes. So it is required that the PBS parameter ${*RMS_PROCS*} should be consistent with the parameter *nproc_node* specified in the input file.

NOTE: The parameter *nproc_node* was introduced for advanced users only. If you are not quite sure how it works, it had better be 0 always.

6.3.2 Compiling and Running the Parallel Program

Similar to the ARPS main program, the compilation and linking of parallel version of ARPSPLT is handled by the UNIX script ***makearps.*** For ARPSPLT to produce CGM metafile output (NCAR Graphics required), the command is:

$> **makearps** [options] *arpspltncar_mpi*

and to produce PostScript graphic output, the command is:

$> **makearps** [options] *arpspltpost_mpi*

The generated executable programs are *arpspltncar_mpi* and *arpspltpost_mpi* and they are put in subdirectory *bin/* within the ARPS root directory.

Similar to the steps mentioned in Section 5, the parallel plotting program can be run on front end or be submitted to a job queue. It is recommended always submitting parallel jobs to a job queue, especially when the job needs many computer resources. To submit the plotting job to a system queue, a platform specific job script is needed. Basing on the samples mentioned in Section 5, the executable should be replaced with either **arpspltpost_mpi** or **arpspltncar_mpi** and the standard input file should be *arpsplt.input.*

To submit an interactive job, for example, on a PC running Linux system with MPICH implementation,

$> **mpirun –**np *proc_num* bin/***arpspltpost_mpi*** **<** *arpsplt.input* > *arpsplt.output*

or

$> **mpirun –**np *proc_num* bin/***arpspltncar_mpi*** **<** *arpsplt.input* > *arpsplt.output*

where option "–np *proc_num*" specifies the number of processors to be used.

6.4 DMP version of ARPS2WRF

ARPS2WRF was provided in the ARPS package since version 5.0.0IHOP_6 and it performs the same functions as program **real.exe** in the WRF package. Working together with programs **ext2arps** and **wrfstatic**, the ARPS processing (**wrfstatic**, **ext2arps**, and **arps2wrf**) is used to replace both the WRFSI and **real.exe** steps when processing the ARPS model data for WRF simulation. The DMP version of ARPS2WRF is possible because both WRF horizontal grid and ARPS horizontal grid are defined over Arakawa-C grid. This common feature ensures minimum message passing between each processor. The sequential program **arps2wrf** also supports horizontal interpolations if the WRF grid is not defined over the same physical domain as the ARPS grid. However, DMP version of **arps2wrf** is only supported when both WRF grid and ARPS grid are the same, *i.e.* parameter *use_arps_grid* = 1 in the ARPS2WRF NAMELIST input file (the ARPS2WRF User's Guide for details).

Since WRF data does not contain fake zones in the I/O process, the size of WRF staggered grid (*nx_wrf*/*ny_wrf*) is two points less than the size of ARPS grid (*nx*/*ny*) in each horizontal direction (*x*-direction and *y*-direction, see section 4 for description about the ARPS fake zones). The size of WRF scalar grid is three points less than the size of ARPS grid (*nx*/*ny*). In order to conform to the ARPS programming standards, all WRF arrays in the program source are declared with dimension (*nx_wrf*, *ny_wrf*, *nz_wrf*). This means that ARPS2WRF has introduced a one point fake zone to the WRF mass or scalar arrays. To facilitate the message passing for these one-fake-zone arrays, a set of MPI wrappers are written specifically for doing message passing with WRF arrays and they are located in the source file *src/arps2wrf/wrf_mpsubs.f90*. The UNIX script ***makearps*** handles the compilation and linking of the file or its no-mpi counterpart (*src/arps2wrf/wrf_nompsubs.f90*) automatically. Note that all subroutines defined in both source files are prefixed with string "*wrf_*" to distinguish them from the general ARPS message passing wrappers with 3 point fake zone.

To compile the DMP version of ARPS2WRF,

$> **makearps [**options**]** *arps2wrf_mpi*

where *options* can be any **makearps** options mentioned in the ARPS User's Guide. Please note that ARPS2WRF cannot read HDF4 formatted ARPS history files because of a conflict between netCDF 3.0 library and HDF4 library. See ARPS2WRF User's Guide for details.

The parallel program **arps2wrf_mpi** can be run just as all other ARPS MPI programs. If it is submitted to a job queues, the executable should be specified to be **arps2wrf_mpi** and the standard input file should be *arps2wrf.input*. If it is an interactive job, execute the command:

$> **mpirun** –np *number_of_processor* **arps2wrf_mp**i < *arps2wrf.input*

Table 5. MPI-related ARPS2WRF input file variables.

Variable	Meaning	Defaults
nproc_x	Number of processors in the X direction	1
nproc_y	Number of processors in the Y direction	1
readsplit	Flag indicate whether input file are split or not	1 – joined input data 0 – split input data
nprocx_in	Number of processor for data in X direction	1
nprocy_in	Number of processor for data in Y direction	1

The MPI parameters to be set in the NAMELIST input file are listed in the table 5. Parameters *nproc_x* and *nproc_y* have the some meanings as they are described in section 4. Parameter *readsplit* indicates whether the ARPS history files provided to ARPS2WRF are in split form or in joined form. If *readsplit* = 1, only the root processor of the program does file-reading and the data is split on-the-fly. If *readsplit* = 0, however, the program **arps2wrf_mpi** can read ARPS history data in split file form. The number of data patches can be the same as the number of processors to be run, such that

```
readsplit = 0,
nprocx_in = 1,
nprocy_in = 1,
```

Otherwise, the number of ARPS data patches can be larger than the number of processors to be run. Parameter *nprocx_in* (*nprocy_in*) specifies the number of data patches in *x*-direction (*y*-direction). It is required that *nprocx_in* (*nprocy_in*) must be divisible by *nproc_x* (*nproc_y*). *i.e.*

```
readsplit = 0,
nprocx_in = n*nproc_x,
        nprocy_in = m*nproc_y,
```

where *n* and *m* are any natural numbers.

NOTE: The generated WRF files, *wrfinput_d01* and *wrfbdy_d01* are always in joined format.

6.5 DMP version of WRF2ARPS

WRF2ARPS is a similar program as EXT2ARPS, except that WRF2ARPS reads WRF history files in either joined format or split format. The support WRF2ARPS history file formats are netCDF format, WRF internal binary format and PHDF5 format. As in ARPS2WRF, the DMP version of WRF2ARPS is realized based on the same similarity between WRF horizontal grid and ARPS horizontal grid. So it is required that NAMELIST parameter *use_wrf_grid* = 1 should be specified in order to run DMP version of WRF2ARPS.

Program **wrf2arps** and **wrf2arps_mpi** reads the same runtime configuration file *input/wrf2arps.input*. All the parameters in file *input/wrf2arps.input* contain the same variables and have the same meanings as those provided in file *input/arps.input* and described in chapter 4 of the ARPS User's Guide, except for the NAMELIST block *&wrfdfile*. This subsection first defines the variables in NAMELIST block *&wrfdfile* and then provides some hints to set up MPI parameters for DMP version of WRF2ARPS.

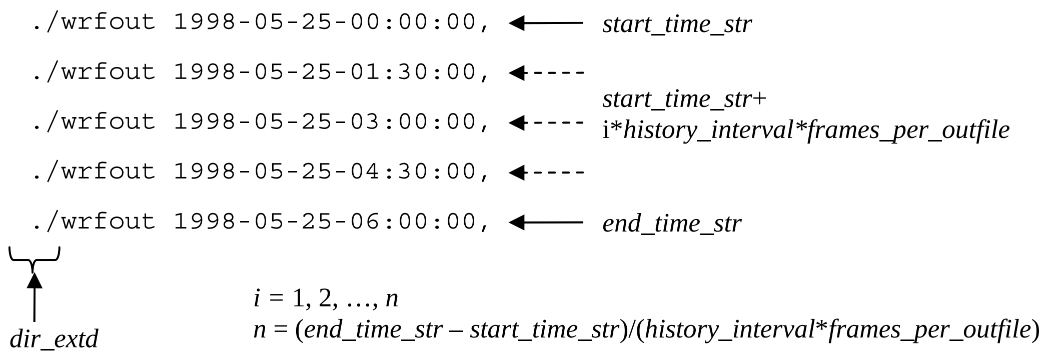
Table 6. WRF file specification for WRF2ARPS2WRF

Variable	Meaning	Defaults
dir_extd	Directory of WRF data files	./
init_time_str	Initial time string of WRF model simulation	'YYYY-MM-DD_hh:mm:ss'
io_form	WRF data file format	= 7 (default) 1 – WRF internal binary format 5 – PHDF 5 format 7 – netCDF format 101 – Binary file in split format 107 – netCDF file in split format
start_time_str	Time string to indicate the first WRF data file	'YYYY-MM-DD_hh:mm:ss'
history_interval	WRF history file output time interval	'DD_hh:mm:ss'
end_time_str	Time string to indicate the last WRF data file	'YYYY-MM-DD_hh:mm:ss'
frames_per_outfile	Output time per WRF data file	1

Since WRF history file names are based on the model simulation time, WRF2ARPS uses parameters *start_time_str*, *history_interval*, *end_time_str* and *frames_per_outfile* to determine the WRF file names to be read. For example,

```
        dir_extd = './',
        start_time_str   = '1998-05-25_00:00:00',
        history_interval =         '00_01:30:00',
        end_time_str     = '1998-05-25_06:00:00',
        frames_per_outfile = 1,
```

WRF2ARPS reads the following files:



```
./wrfout 1998-05-25-00:00:00,
./wrfout 1998-05-25-01:30:00,
./wrfout 1998-05-25-03:00:00,
./wrfout 1998-05-25-04:30:00,
./wrfout 1998-05-25-06:00:00,
```

$i = 1, 2, \ldots, n$
$n = (end\_time\_str - start\_time\_str)/(history\_interval * frames\_per\_outfile)$

Parameter *io_form* specifies the WRF history file format and the current supported formats in WRF2ARPS are WRF internal binary format, netCDF format, and PHDF5 format. Please note that PHDF5 format is only support when the program is in MPI mode, just as it has been supported in the WRF system. When *io_form* > 100, WRF files are in split form with a four-digit processor rank appended to each WRF file name. For WRF version earlier than V2.1.0, the user should plan in advance if split WRF files are to be read because of the difference in domain decomposition schemes used in the WRF system and the ARPS system. The required modifications within the WRF model source is described in file *input/wrf2arps.input*. It is not a problem any more since WRFV2.1.1 and later, thanks to the improvement in WRF domain decomposition.

Parameter *init_time_str* specifies an initial time string that denotes the initial time of the WRF model simulation. It is used only for constructing the ARPS file names and has nothing to do with the WRF data files. It was introduced to handle the cases when the first WRF data file (specified by parameter *start_time_str*) is not at the initial simulation time of the WRF model.

Table 7 lists all the variables that should be set or checked before running DMP version of WRF2ARPS. Many parameters are the same as in all other ARPS MPI programs, such as *nproc_x*, *nproc_y*, *max_fopen*, *nprocx_in*, *nprocy_in*, *dmp_out_joined*. The only new parameter is *io_form* that is used to replace parameter *readsplit* in other MPI programs. When *io_form* > 100, all WRF history files are in split form and specifications of parameter *nprocx_in* and *nprocy_in* are required. Otherwise, WRF data files are in joined form and the read-in data should be split on-the-fly.

Table 7. MPI-related WRF2ARPS input file variables.

Variable	Meaning	Defaults
nproc_x	Number of processors in the X direction	1
nproc_y	Number of processors in the Y direction	1
max_fopen	Maximum number of files allowed to be opened	8
io_form	WRF data file format	= 7 (default)
nprocx_in	Number of processor for data in X direction	1
nprocy_in	Number of processor for data in Y direction	1
dmp_out_joined	Flag for ARPS outputs	1 – joined ARPS file 0 – split ARPS files

NOTE:

- In MPI mode, parameter *frames_per_outfile* must be 1.
- WRF PHDF5 files are always in one joined form and they are supported in MPI mode only.

DMP version of WRF2ARPS can be compiled as all other programs by executing command:

$> **makearps** [options] *wrf2arps_mpi*

The generated program **wrf2arps_mpi** can be executed interactively,

$> **mpirun** –np *number_of_processor* **wrf2arps_mpi**

or be submitted through a job script as has mentioned in section 5.

6.6 DMP version of ARPSEXTSND

To be added by Kevin Thomas

6.7 DMP version of ARPSVERIF

To be added by Kevin Thomas