1. Introduction

1.1. PURPOSE OF THIS DOCUMENT

This document is intended to provide a detailed description of the structure, function, and rationale behind the distributed-memory (DM) parallel MM5 code for maintainers and developers. It may also be of some interest to MM5 users, though the focus is primarily how the code is implemented and how it works on distributed memory parallel computers.

1.2. OTHER SOURCES

2. Design of the DM-parallel MM5

The DM-parallel option to MM5 is a *same-source* implementation. This means that in addition to providing a single-source code for both DM-parallel and non-DM users, the code also preserves the original look and operation of the non-DM code. Most of the complexity of the parallel implementation is invisible in the code, residing instead in the build mechanism associated with the DM-parallel option and in files located in a single, optional, directory tree that is distributed as a separate tar file. The design, first implemented with MM5 Version 2.8 in March of 1998, has worked well in that it has proven to be as efficient on distributed memory parallel computers as earlier hand-coded separate versions of the model. And it has allowed the DM-parallel option to be maintained as part of the official model through more than seven subsequent code releases and one new version (the code is at 3.3 at this writing). (See <citation>).

The same-source parallel implementation has five components:

1. RSL, a high-level library that provides interprocessor communication for stencil-exchange and inter-domain exchange of forcing and feedback data, support for data domain decomposition and control of iteration over decomposed dimensions within local processor subdomains, and distributed Fortran input and ouput;
2. FLIC, a Fortran source translator that recognizes and converts loops over decomposed dimensions to use RSL-provided loop ranges for iteration within local subdomains and converts between global and local uses of indices where needed;
3. DM-specific code fragments that are included at a number of locations (28) in the MM5 source code;
4. DM-specific subroutines (16) that are included in the program at linkage; and
5. A set of DM-related makefiles, tables, and other miscellaneous files that install, preprocess, compile, and link the code in DM-parallel mode.

and these are used to address the 5 functional aspects of MM5 in which parallelism must be addressed:

1. Two-dimensional (horizontal) decomposition of two-and three-dimension arrays containing model state data;
2. Interprocessor communication to address horizontal data dependencies and reduction operations in model initialization;
3. Interprocessor communication and/or replicated computation to address horizontal data dependencies in the main and time-split solvers that advance the model solution forward one time step;
4. I/O:
    a. Input of configuration data from the namelist file,
    b. Input of initial conditions from files in MM5 data format,
    c. Input of lateral boundary conditions from files in MM5 data format,

d. Input of gridded data sets for FDDA analysis nudging,

e. Input of point-observation data sets for FDDA observational nudging,

f. Output of model history files in MM5 data format, and

g. Output of model restart data sets; and

5. Interprocessor communication associated with nest forcing and feedback.

A minimal parallelization of MM5 must address items 1-3, 4 a-c, and 4 f.

Section 3 describes the parallel implementations in terms of the steps one would take to parallelize the code anew, beginning with a serial version of MM5, modifying so that it first can be built as a serial code but with the parallel build mechanism, then as a pre-parallel code capable of initializing itself and running on one processor, and then finally with communication added to enable execution in parallel. Section 4 details the aspects of the parallel implementation in module-by-module fashion. Section 6 discusses performance measurement and optimization strategies. Section 7 provides information on debugging the parallel code.

3. How to parallelize MM5

Although the officially available version of MM5 is already parallel, the discussing the process by which the model was parallelized is instructive for basic understanding and also for users faced with retrofitting parallelism within older versions of this model or in models other than MM5. The basic flow is implement the new build-infrastructure and get the model to compile and run on a single processor then add parallelism, beginning with I/O and initialization and finishing with automatic loop restructuring and interprocessor communication in the solver. The remainder of this section provides a is a top-level task list for parallelizing MM5, starting with an existing, non-parallel version of the model.

3.1. SET UP BUILD INFRASTRUCTURE

The same source approach to parallelizing MM5 dictates that as little of the model itself by modified as possible. What this means, however, is that a large amount of the complexity of the parallel MM5 -- the invocation of the FLIC preprocessor on model subroutines, the linkage to the RSL library, the modification of model data structures for local processor memory, etc. -- is actually encapsulated within the build mechanism. Therefore, the first step in parallelizing MM5 is to start with a working serial version and then adapt it to the mechanism that is used to build the parallel model. The result is a serial version of the code that is built the same way as the parallel code.

3.1.1. The MPP directory

This discussion assumes the existence and availability of the officially available parallel version of MM5 and in particular the MPP directory containing parallel code an build-infrastructure. Download and unpack ftp://ftp.ucar.edu/mesouser/MM5V3/MPP.TAR.gz so that the MPP directory resides at the top-level of the model source tree. Think of this directory as a "kit" for parallelizing the code; not everything will be ready-to-go, but all the components are here in at least example form so that parallelization is accomplished by modifying components of the kit and of the model to be parallelized. You should also download and install the model itself from ftp://ftp.ucar.edu/mesouser/MM5V3/MM5.TAR.gz to have a working example to refer to.

MPP directory

All of the files for the DM-parallel option are maintained within a single directory tree that is separate from the rest of the code, in keeping with the same-source design. The DM-parallel source tree is rooted in the directory MPP, that appears among the top level directories in the MM5 source distribution if the user has downloaded the file MPP.TAR.gz. The model can be built and used on non-DM computer platforms without the MPP directory.

The MPP directory contains files and directories applying to the DM-option in general and also a subdirectory, RSL, that contains files and directories that are specific to the RSL implementation, including the RSL library itself.

This structure allows other parallel libraries to be utilized instead of RSL. If a different package is used, a new directory (e.g. NNTSMS) should be created under the MPP directory and all NNTSMS specific files would appear there. The MPP_LAYER variable in section 7 of the configure.user file is set to the name of the library to be used.

The MPP directory also contains a directory named FLIC. This contains the FLIC source translator, used to pre-compile the MM5 source code for use with RSL or other message passing libraries. The FLIC tool generates library-independent M4 macros within the code that are then expended prior to compilation using library-specific macro definitions that are contained in the package-specific directory. For example, the RSL-specific macro definitions are in the file RSL/LMexp.m4 (except for the FujitsuVPP, which uses RSL/LMvpp.exp).

At the top level of the MPP directory are files that apply to the parallel MM5 regardless of what library is used. These include:

- **mpp_objects_all**. A list of all the .o files that make up the DM version of the model. This file contains CPP #ifdefs that key on the contents of the file include/defines.incl, which is generated by the parseconfig program using settings in the configure.user file when either version of MM5 (non-DM or DM) is built. This file is used to generate.
- **FLICFILE**. Small file of options to FLIC to direct the translation of the MM5 prior to compilation.
- **namelist.data**. File containing a table of the namelist parameters used by MM5; this table is used to generate the code in param.F that broadcasts namelist information to the other distributed-memory processes after it has been read by process zero. The script **namedata.awk** processes the table.
- **col_cutter.c**. A C program used during precompilation to ensure that statements in the modified code do not go past column 72.
- **mpaspect.F**. A subroutine that computes the dimensions of a two-dimensional processor mesh, given the available number of processors and lower bounds for the number of north/south rows and east/west columns in the mesh. MM5 calls this routine to get default processor mesh dimensions (or failure if a mesh cannot be computed) but allows the default dimensions to be overridden via the namelist (NPROC_LT and NPROC_LN).
- **Makelinks**. A csh shell script that creates the MPP/build directory (described below) in which the DM-parallel version of the code is compiled.
- **mpp_install**. Existence of this file indicates that MPP has installed itself.

MPP/build directory

The DM-parallel version of the code is built in the MPP/build directory. This directory is created the first time the code is compiled with ‘make mpp’. The MPP/build directory is created by the script MPP/Makelinks. It is removed using ‘make uninstall’. This directory contains symbolic links to every .F and .c file in the rest of the MM5 directory tree. This may be a benefit if you prefer working in a flat source directory structure but remember these are links, not copies: changes made to .F files that appear in the MPP/build directory affect the original file. The MPP/build also contains a link to the library specific makefile from the directory MPP/$(MPP_LAYER)/Makefile.$(MPP)_LAYER) – for example, MPP/RSL/Makefile.RSL.

When the DM-parallel code is built using ‘make mpp’, all of the intermediate files reside in MPP/build, including pre-processed .f files that are input to the compiler and the resulting .o object files. The resulting executable file, mm5.mpp, is copied automatically into the Run directory at the end of a successful link. Building the DM-parallel code in its own directory and having a separately named executable allows both the DM and non-DM codes to exist simultaneously in the same directory tree. This is useful for testing the DM-parallel code with respect to the non-DM code.

MPP/FLIC directory

The MPP/FLIC directory contains the source code for the FLIC precompiler. FLIC and a small csh driver script, MPP/FLIC/FLIC/flic, are built the first time that the code is compiled with ‘make mpp’. The directory is automatically cleaned with ‘make uninstall’. Absolute pathnames are used in the driver script; therefore, whenever the MM5 code is moved to a different directory, it should be uninstalled and remade using ‘make uninstall’ and then ‘make mpp’ in the new location.

FLIC is a lex/yacc based translator. However, the lex and yacc generated C files and not the lex and yacc sources themselves are distributed. This is a mixed blessing: it ensures that FLIC will compile even on systems where lex and yacc are not available; however, there have been occasional incompatibilities when porting to new versions of Unix.

MPP/RSL

The MPP/$(MPP_TARGET) directory contains every file that is specific to a particular library, such as RSL. The parallel implementation of MM5 is intended to be package independent and so, allows for other libraries by adding a directory beneath MPP for that target library. Currently only RSL is targeted. For simplicity, the remainder of this discussion will simply refer to RSL.

Library specific include files: MPP/RSL/mpp_*.incl

In addition to the directory containing the RSL library itself (MPP/RSL/RSL), the MPP/RSL directory contains a large collection of “include” files that are incorporated into MM5 source routines as a precompilation step using CPP #include directives. The includes are triggered by definition of the CPP macro MPP1, which is defined as part of CPPFLAGS in MPP/RSL/Makefile.RSL. These files have names mpp_*subroutine_label*.incl, where *subroutine* refers to the name of the including MM5 subroutine and *label* is an integer identifier for the modification. This technique allows the inclusion of library-specific and parallel-specific modifications with only minimal impact on readability or essentially same-source look and feel of the original source code. The mpp_*.incl files include files may include declarations, executable lines of code, or directives. This approach minimizes the number of lines of code changed for parallelism, it eliminates references to a particular parallel library in the code itself, and it further facilitates a more library independent approach to the parallelization. Here is an example:

```
#ifdef MPP1
#  include "mpp_solve_10.incl"
#endif
```

The MPP/RSL directory also contains the file rslcom.inc, which contains declarations for constants, data structures and subroutines that are used in MM5 subroutines in distributed memory parallel mode (or, more properly, that are used in the mpp_*.include files that are included in MM5 when the CPP macro MPP1 is defined). Among the important data structures defined in rslcom.inc is DOMAINS(RSL_MAXDOMAINS), an array storing the integer RSL domain descriptor for each active domain. Many RSL routines, including stencil exchange and I/O routines, need this descriptor to know which domain the operation pertains to. DOMAINS(1) always contains an active descriptor, the mother domain. The index into DOMAINS corresponds to the MM5 variable INEST in most model routines.

Although this file is included by almost all MM5 subroutines, the #include directives for rslcom.inc do not appear in the source code itself. The includes are inserted automatically by FLIC using the the -CPP='include<rslcom.inc>' option in the definition of FLIC_FLAGS in MPP/RSL/Makefile.RSL.

Library specific subroutines: MPP/RSL/parallel_src

Subroutines specifically for parallelization with the RSL library appear in the directory MPP/RSL/parallel_src. These are:

- define_comms.F. This subroutine defines communication constructs (stencil exchanges) that are used at various places in the calculation of a model time step, primarily in the SOLVE and SOUND routines but also initialization, nesting, and elsewhere.
- fkill_model.F and kill_model.c. These routines provide graceful program termination through the MPI_ABORT routine in the case of errors. The FLIC program automatically coverts all STOP statements in MM5 to call FKILL_MODEL using the -STOP=FKILL_MODEL option in the definition of FLIC_FLAGS in MPP/RSL/Makefile.RSL.
- bcast_size.F and merge_size.F. These routines calculate the buffer sizes used in communication for nest forcing and feedback.
- mp_equate.F. Contains the parallel definitions of the EQUATE and EQUATEO routines used in the MM5 input (rdinit.F) and output (outtap.F) routines and elsewhere.
- mp_initdomain.F. Used to initialize runtime data structures in rslcom.inc and in MPP/RSL/LMexp.m4 for computation on a particular domain.
- lb_alg.c. Contains the algorithms used to decompose MM5. The routines in this file take into account the static memory size of model arrays, possible differences in computational speed of processors in the run-time partition (IBM only), and static imbalances associated with domain boundaries. The routines here override the simpler decomposition algorithms built into the RSL library.
- mp_blw.F. Used in parallel implementation of FDDA analysis nudging.
- mp_feedbk.F, mp_stotndt.F, and vpp_stotndt.F. Related to nest forcing and feedback. The vpp_stotdnt.F file is used instead of mp_stotndt.F on the Fujitsu VPP.
- Error_dupt3d.c. Cray T3D and T3E specific routines for redirecting standard output and standard error to files from each processor. Other machines use the RSL library routine RSL_ERROR_DUP.
- upshot_mm5.c. Routines related to upshot profiling. Upshot profiling is enabled when the model is compiled with –DUPSHOT in defined for CPPFLAGS and CFLAGS in the configure.user file.

Library specific macros: MPP/RSL/LMexp.m4

This file contains a number of expansions for macros that the FLIC translator inserts as part of precompiling the source code. Having FLIC generate library-independent macros instead of the RSL-specific code means it is only necessary to redefine the macros in this file to have FLIC target a different library package. The expansions that must be defined in this file are:

- FLIC_RUN_DECL: expands to the data structures that contain information for controlling loops over decomposed dimensions in the model and converting between global and local indices. The macro is inserted automatically by FLIC into each subroutine via the -H='FLIC_RUN_DECL' option, specified in FLICFLAGS in MPP/RSL/Makefile.RSL.
- FLIC_INIT_RUNVARS: expands to calls to the RSL library to initialize the data structures included by the expansion of FLIC_RUN_DECL. This macro is part of MPP/RSL/parallel_src/mp_initdomain.F.
- FLIC_DO_N, FLIC_DO_M, FLIC_ENDDO: macros that FLIC inserts at the beginning and end of loops over decomposed dimensions (M corresponds to I in MM5; N corresponds to J). These expand to use loop control data structures in the expansion of the macro FLIC_RUN_DECL.

Include directories

Non-DM builds of MM5 include header files from the include directory. DM-parallel builds include header files from the pick directory and from the directory MPP/RSL/RSL. The header files for DM-parallel builds are placed in the pick directory by the include/Makefile and MPP/RSL/Makefile.RSL. Three header files, read_config1.h, read_config2.h, and read_config3.h, appear only in DM-parallel builds. These are included when

domain/initial/param.F is compiled and contain declarations and code pack and unpack configuration data in buffers for broadcast to all processors.

3.1.2. Edit the top-level Makefile and configure.user

Modify the top-level makefile to add targets mpp, MPP/mpp_install, mpclean, and uninstall, using Makefile in the official MM5 as a guide. Add the following definitions to your configure.user file, assuming you have one in your version of the model (very old versions of MM5 do not). If there is no configure.user file included by your Makefile, then add the definitions to the Makefile directly or make other provisions for these being defined.

- RUNTIME_SYSTEM = *system*
- MPP_TARGET = RSL
- MPP_LAYER = RSL
- PROCMIN_NS = 1
- PROCMIN_EW = 1
- MFC = *fortran compiler*
- MCC = *C compiler*
- MAKE = make –i –r
- CPP = cpp –C –P
- AWK, SED, M4, CUT, and EXPAND set to local command names
- IWORDSIZE, RWORDSIZE, and LWORDSIZE: size in bytes of integer, real, and logical

The value of *system* for RUNTIME_SYSTEM should be set to a string identifier (no quotes) for your system corresponding to those supported by RSL. Type 'make' in the MPP/RSL/RSL directory for a list.

3.1.3. Edit list of object files in MPP/mpp_objects_all

The file MPP/mpp_objects_all contains a list of all the .o files that make up a compilation. The file defines the make macro OBJS that is used to construct MPP/build/tmpobjs, which is included by MPP/RSL/Makefile.RSL. Object files that are linked unconditionally should be part of BASE_OBJ. The rest of the file may contain definitions for other object files that are only linked in when certain objects are specified in the configure.user file[1]. This conditional inclusion is controlled by CPP macros set in the file include/defines.incl.

3.1.4. Modify the file MPP/RSL/Makefile.RSL

Makefile.RSL is the primary makefile for the DM-parallel code. It contains the command to link the executable mm5.mpp from the constituent object files and all rules for creating those object files. The generic make rule for compiling .F files and .c files .o is near the beginning of the file. This is followed by certain special rules that cover:

- Defining the preamble to the parame.incl file,
- Compiling the param.F with special mechanisms for broadcasting namelist data,
- Compiling define_comms.F without FLIC and with special M4 macros local to that routine,
- Routines that are not compiled with FLIC because they contain FLIC macros already hand-inserted,
- Modules that do not need FLIC because they are column callable by design,
- Modules that do not use FLIC because they are related to nesting,
- Functions and block-data routines that do not need to be compiled with FLIC,

---

[1] This conditional inclusion is controlled by CPP macros defined in the file include/defines.incl. This file is created automatically in both the DM and non-DM versions when the program parseconfig (Util/parseconfig.c) is invoked from the top-level Makefile.

- Packages needing special handling because of miter loops (Blackadar and Gayno-Seaman PBLs), and
- a few miscellaneous other special cases.

At this stage, none of the actual parallelization work has begun we are only implementing the build mechanism for the DM code. Therefore, one may wish to disable actual transformation of the code by commenting out the definitions of FLICFLAGS and CPP_FLAGS[2]. This will cause FLIC to simply pass the code through unmodified. This will allow you to develop and test the DM-build mechanism with the non-DM parallel code. It may also be necessary to remove certain DM-specific object files from MPP/mpp_objects_all and set configure.user for as simple a configuration as possible (e.g., MAXNES=1, for no nesting). It may also be necessary to add objects from the non-DM code that are not part of the DM-parallel build.

The result of this exercise should be a version of the code that is compiled using the DM-parallel build mechanism but that runs as a non-DM parallel executable and produces the same results.

Defining the preamble to parame.incl: array size parameters for distributed memory

Two- and three-dimensional arrays in MM5 are dimensioned using MIX, MJX, and MKX integer parameters for north-south, east-west, and vertical, respectively. These are defined in the configure.user file. This file, information in include/parame, and mechanism in include/Makefile are used to create include/parame.incl, the file that is actually included by MM5 subroutines when the non-DM version is built. In DM-parallel builds, configure.user, include/parame, and MPP/RSL/Makefile.RSL are used to construct pick/parame.incl.

MPP/RSL/Makefile.RSL contains a set of rules that add a preamble to the file in include/parame to create the pick/parame.incl, included by nearly all the subroutines in the DM-parallel build. The mechanism is analogous to that in include/Makefile for the non-DM code In the case of the DM-parallel code, the horizontal array dimensions MIX and MJX are modified on the DM-parallel code to include extra memory for halo data around the local processor subdomain and also to reduce the overall amount of memory required on each processor. MIX and MJX in the configure.user file are renamed to MIX_G and MJX_G (for "global") in the pick/parame.incl file, and the local MIX and MJX are redefined as:

```
PARAMETER( MIX=MIX_G/$(PROCMIN_NS)+2*$$RDP+2 )
PARAMETER( MJX=MJX_G/$(PROCMIN_EW)+2*$$RDP+2 )
```

where PROCMIN_NS is the minimum number of processors allowed in the north-south dimension and PROCMIN_EW is the minimum in the east-west dimension. These are also defined in configure.user. RDP is the amount of pad area needed by RSL for stencil exchanges (3, as defined in MPP/RSL/RSL/rsl.h) and the extra 2 is to allow an additional element on each end of the dimension for spurious boundary calculations[3]. The values of MIX and MJX are used in subsequent parameter statements in parame.incl to define a number of related constants associated with particular optional packages in the model – for example, MIXIC and MJXIC are related to ice-physics, MIXR and MJXR to radiation, and so on. All of these become local memory dimensions in the DM-parallel version.

On a related note, MIX and MJX in their various forms are listed in MPP/FLICFILE using the mdim and ndim directives. This provides FLIC with the information it needs to recognize loops over decomposed dimensions when it precompiles the model.

---

[2] In the case of certain architectures, it may be necessary to preserve certain architecture specific definitions such as –DDEC_ALPHA on the Compaq Alpha systems. Rule of thumb would be if the CPP definition is also needed for the non-DM parallel build, it should be retained here as well.

[3] This occurs when the code on a processor computes a value for a boundary that isn't stored locally. One could avoid this by inserting conditionals; however, we simply allow the processor to compute the value anyway, but in a safe extra zone of memory. This is why all memory is initialized to a non-zero value in MPP/RSL/mpp_mm5_05.incl, to prevent spurious floating point exceptions in these calculations.

3.2. TOP-LEVEL PARALLEL INITIALIZATION

The main routine of the model is adapted to initialize the underlying parallel system (RSL and MPI).The main routine for MM5, both in the DM-parallel and non-DM versions of MM5, is contained in the file Run/mm5.F. This routine contains calls to read in namelist configuration information, initialize the model from initial or restart data, other initializations, and the main time loop of the model (the 10 loop) which includes call to the nesting, the solver, and model output routines. In addition, the DM-parallel version of mm5.F includes:

- MPP/RSL/mpp_mm5_00.incl:
  1. CALL RSL_INITIALIZE: Initialize RSL.
  2. CALL ERROR_DUP: Redirect standard output and standard error on each processor
  3. CALL SET_DEF_DECOMP_FCN(MAPPING): Establish MAPPING (defined in MPP/RSL/parallel_src/lb_alg.c) as the decomposition function (overriding the default algorithm in RSL)
- MPP/RSL/mpp_mm5_05.incl: Initial value (1.0) to all state data[4].
- MPP/RSL/mpp_mm5_10.incl:
  1. CALL RSL_OUTPUT_BUFFER_YES, RSL_IO_NODE_YES (or NO): enable (or disable) use of processor zero as a dedicated I/O server for the other nodes.
  2. CALL STATMEM_STAT(MIX,MJX): provide the static array sizes in the minor I-dimension (north-south) and the major J-dimension (east-west) to the MAPPING function that was passed to RSL in MPP/RSL/mpp_mm5_00.incl. The MAPPING function needs this information to avoid decompositions that would require larger memory on each processor than statically defined.
  3. CALL RSL_MESH(NPROC_LT, NPROC_LN): Specify processor mesh as the number of processors in minor (north-south) and major (east-west) grid dimensions. The mesh is computed by the routine MPASPECT, defined in MPP/mpaspect.F (it is not library-specific), called within MPP/RSL/mpp_param_30.incl (which, as the name indicates, is included by domain/initial/param.F).
  4. CALL INEST_STAT(1,1): informs decomposition function MAPPING that this is the mother domain (not a nest).
  5. CALL RSL_MOTHER_DOMAIN(DOMAINS(1), RSL_168PT, IL, JL, MLOC, NLOC): defines the mother domain and stores the RSL descriptor in the first element of the DOMAINS array (defined in rslcom.inc). The second argument, RSL_168PT, tells RSL the largest stencil that will be used on the domain for halo exchanges. The third and fourth argument give the logical (global)domain dimensions) and the last two arguments are returned by the routine. These contain the minimum array sizes in the north-south and east-west dimensions, respectively, for the local processor subdomain RSL has computed. Following the call to RSL_MOTHER_DOMAIN, these are tested to make sure that the static dimensions of the MM5 arrays are large enough and, if not, an error is generated.
  6. CALL SHOW_DOMAIN_DECOMP(DOMAINS(1)): Output a file, show_domain_0000, containing decomposition information.
  7. CALL DEFINE_COMMS(1): Define stencil communications for this domain.
  8. CALL MP_INITDOMAINS(1): Enable computations on this domain by initializing the data structures that are inserted into each subroutine by FLIC via the FLIC_RUN_DECL macro.
- MPP/RSL/mpp_mm5_20.incl: Call RSL_SHUTDOWN on normal model completion.

---

[4] State data is initialized to 1, not zero, to avoid floating point exceptions in certain places where spurious calculations are allowed to occur outside the local partition: computing results in halo regions or dealing with boundary conditions on processors that don't actually own the boundary.

The call to RSL_INITIALIZE is accomplished early so that RSL may be used broadcast configuration information from processor zero, which reads the input namelist, to the other processors. By the time the code in mpp_mm5_10.incl is executed, all namelist configuration information is available on all processors. Thus, processor mesh specifications in the namelist can override the default if, for example, the user wishes to run with a 2x8 mesh instead of a 4x4 mesh on 16 processors.

The interface to RSL for controlling iteration over decomposed dimensions and for switching between model domains requires that source files include the library Fortran header file MPP/RSL/RSL/rsl.inc and the MM5 specific RSL data structures in MPP/RSL/rslcom.inc. These need not be inserted by hand; FLIC can do this automatically. Renable the FLICFLAG options in MPP/RSL/Makefile.RSL so that it reads:

```
FLICFLAGS = -F=$(MPPTOP)/FLICFILE -CPP='include<rsl.inc>' \
            -CPP='include<rslcom.inc>' -H='FLIC_RUN_DECL' \
            -STOP=FKILL_MODEL
```

The –F option directs FLIC to look for code translation directives in the file MPP/FLICFILE. The two –CPP options instruct FLIC to insert CPP-style #include directives at the beginning of each subroutine it translates. The –H option causes FLIC to include the macro FLIC_RUN_DECL in the declarations section of each model subroutine. The expansion for this macro is contained in MPP/RSL/LMexp.m4. The last option instructs FLIC to replace STOP statements in the model with calls to the routine FKILL_MODEL (defined in MPP/RSL/parallel_src/fkill_model.F). Also renable the CPP_FLAGS macro in MPP/RSL/Makefile.RSL.

The main MM5 routine (Run/mm5.F) should contain a call to MP_INITDOMAIN (defined in MPP/RSL/parallel_src/mp_initdomain.F) early in the routine before any loops over decomposed dimensions have occurred. In the officially supported MM5 this is done in the included file MPP/RSL/mpp_mm5_10.incl.

At this point, all subroutines have access to the run-time data structures that provide the interface to RSL for controlling loops over decomposed dimensions and distributed I/O.

3.3. I/O

The next step is to implement input and output so that the model can input initial data sets on one and then multiple processsors. Output is also implemented to allow a check on input, with the exception of restart data sets. MM5 I/O differs markedly between Version 2 and Version 3 and so does the DM-parallel implementation of I/O in those versions. However, in both versions, the DM-parallel option reads and writes the unformatted Fortran files same files as the non-DM version.

A useful strategy for implementing I/O is to pattern the modifications after what's been done already in the official versions (2 and 3) of MM5. Reference is made to these in the description which follows in this section. Chose which according to the vintage of the version you are parallelizing and according to the data format you wish to support. Other things being equal, Version 3 is preferred since that format is self-describing, but one may have legacy reasons for chosing the older Version 2 format and mechanisms.

During this phase of implementation, one may wish to include a temporary call to model output imediately after model input and before any time-stepping has occurred to verify that input is working properly on single and multiple processors. Specifically, place a call to OUTPUT and then RSL_SHUTDOWN and STOP immediately after the call to INIT in the main routine in Run/mm5.F.

3.3.1. Model input

Model input consists of input of namelist data, initial data from MMINPUT_DOMAINx files, where 'x' is the domain number, periodic input of lateral boundary conditions from the file BDYOUT_DOMAIN1 (only the coarse domain), and – in the case of FDDA – other periodic input from MMINPUT files. The initial data is read by the subroutine RDINIT (domain/io/rdinit.F); the boundary data by BDYIN (domain/boundary/bdyin.F). The discussion here will focus on initial and lateral-boundary data, and will concentrate on MM5 Version 3. Version 2 input is discussed in the next subsection.

Namelist data is read by node zero and broadcast to the other nodes in code added to the PARAM routine (domain/initial/param.F). This is accomplished in the code included with MPP/RSL/mpp_param_30.incl :

```
      IF(RSL_IAMMONITOR())THEN
C       CODE ON MONITOR ONLY
        READ (ILIF10,OPARAM)
        READ (ILIF10,LPARAM)
        READ (ILIF10,NPARAM)
        READ (ILIF10,PPARAM)
        READ (ILIF10,FPARAM)
C       FILE CONTAINING AUTOMATICALLY GENERATED PACK STATEMENTS
#       include "read_config2.h"
      ENDIF
      CALL RSL_MON_BCAST(IBUF,IBUFLEN*IWORDSIZE)
      CALL RSL_MON_BCAST(LBUF,LBUFLEN*LWORDSIZE)
      CALL RSL_MON_BCAST(RBUF,RBUFLEN*RWORDSIZE)
C
      call dm_bcast_string   ( cdatest ,  19)
C
      IF(.NOT.RSL_IAMMONITOR())THEN
C       CODE ON OTHERS ONLY
C       FILE CONTAINING AUTOMATICALLY GENERATED UNPACK STATEMENTS
#       include "read_config3.h"
      ENDIF
```

The reads of the namelist file are conditional on RSL_IAMMONITOR() so that they are only performed on node zero. The data from the namelist variables are packed into one of three buffers -- IBUF, LBUF, or RBUF, for integer, logical, and real items, respectively – which are then broadcast in the three calls to RSL_MON_BCAST. The buffers are defined in the automatically generated include file pick/read_config1.h, included near the top of the PARAM routine. The actual packing code is in the include file pick/read_config2.h, executed on node zero. The code to unpack the buffers on the other nodes after the broadcast is in pick/read_config3.h.

The three read_config include files are automatically generated when param.F is compiled in MPP/RSL/Makefile.RSL. The UNIX awk command is called using the script in MPP/namedata.awk to process the table in MPP/namelist.data. The MPP/namelist.data table lists all the namelist variables, their type, and dimensionality. Thus, to parallelizing a non-parallel version of MM5, it is necessary to go through this table and ensure the accuracy and completeness of the entries for the model version in question. When one adds or removes a namelist variable from the model, it is necessary to reflect that change in MPP/namelist.data. The read_config files themselves should not be edited, since they are generated automatically and changes would be lost after the next compile.

The call to DM_BCAST_STRING (domain/io/dm_io.F) in the code fragment above is used to broadcast the date string to other processors in MM5 Version 3. In Version 2, the date is stored as an integer and broadcast as part of the header information when the initial input data file is first read. This was changed in Version 3 for Y2K compliance.

More needed on Version 3 I/O.

3.3.2. Model input (V2)

Namelist data is handled in the same manner as Version 3.

Initial data is input from fort.11; boundary data from fort.9. The DM version of rdinit is named MPRDINIT (MPP/RSL/parallel_src/mprdinit.F). The boundary data read routine is MPBDYIN (also in the parallel_src directory).

The MPRDINIT routine is called from the same location in the INIT routine (domain/initial/init.F) as the non-DM input routine, RDINIT. MPRDINIT routine follows the control flow of the non-DM RDINIT routine but with reads of 2- and 3-dimensional arrays performed by calls to the RSL_READ routine. MPRDINIT takes one additional argument, INEST, the index of the domain being initialized. This is needed by the DM version in order to index the RSL domain descriptor from the array DOMAINS (defined in the included file MPP/RSL/rslcom.inc), for use in the call to RSL_READ.

Within RSL_READ, node zero performs an unformatted Fortran read and then distributes the data to the other processors. It is therefore important to remember that RSL_READ must be called on all nodes and that only node zero actually manipulates the file. That is why one finds, partway through the calls to RSL_READ in MPRDINIT, two conditional calls to Fortran read statements to skip two unwanted fields in the model input:

```
IF (RSL_IAMMONITOR()) READ(IUNIT)
IF (RSL_IAMMONITOR()) READ(IUNIT)
```

Other non-distributed actions on the fort.11 file – conditional rewinding of the file, reading of the large header record that precedes each frame (time period) of data in the file, the close of the input file at the end of the routine -- are also conditional on RSL_IAMMONITOR(), a logical function in the RSL library that returns true on node zero and false on all other nodes. This function is defined in the rsl.inc file, a #include directive for which is inserted automatically into the routine when it is preprocessed by FLIC. Certain data read in from the header record is broadcast to the other nodes using RSL_MON_BCAST (note that this routine is outside the node-zero only code; all nodes must call this routine).

The routine begins with a call to MP_INITDOMAIN(INEST) to set up the GLEN, LLEN, and DECOMP arrays used by RSL_READ and also to set up iteration over decomposed I and J dimensions midway through the routine where the non-hydrostatic base state is calculated. The routine ends with a call to RSL_EXCH_STENCIL to update the halo regions on the just-read-in fields. The stencil STEN_INIT, like all stencils in the DM-parallel version of MM5, is defined in MPP/RSL/parallel_src/define_comms.F.

In MM5 Version 2, there is also a read of the first record in the initial data file fort.11 from the PARAM routine (domain/initial/param.F). With the DM option, this read is conditional on RSL_IAMMONITOR() and the four header arrays (two on the T3E) are broadcast to the other processors using RSL_MON_BCAST (see MPP/RSL/mpp_param_10.incl). This read of fort.11 in PARAM is the reason a REWIND is needed in RDINIT and MPRDINIT.

The MPBDYIN code is used to read data into slab-boundary arrays that apply boundary forcing to the solution through the NUDGE routine and elsewhere in SOLVE3. The slab-boundary arrays are dimensioned (MIX,MKX,5) for the western and eastern boundaries and (MJX,MKX,5) for the southern and northern boundaries. Thus, west-east slab arrays are distributed in the I-dimension and south-north slab arrays are distributed in the J-dimension[5]. This atypical data structure is handled using a special MM5-V2 specific routine in RSL: RSL_MM_BDY_IN. This routine reads and distributes all four slab-boundary arrays simultaneously.

RSL routines are called directly from these routines rather than hidden in #included files, as is done in the rest of the code. This is because these are already separate RSL-specific routines that appear under the MPP/RSL directory. The need for separate DM-parallel routines for model input is a Version 2 artifact. This has been corrected in MM5 Version 3.

3.3.3. Model output

MM5 output is primarily "history"; that is, periodic output of model state and diagnostic variables. Optionally, MM5 also outputs restart data sets. This discussion concentrates on history output.

More needed here for V3.

---

[5] These may also be distributed in the narrow dimension if the decomposition is fine enough.

3.3.4. Model output (V2)

MM5 V2 history is written by the OUTTAP routine (domain/io/outtap.F) and there is no special version for DM-parallel. However, most of the actual distributed output mechanism is hidden within the #included file MPP/RSL/mpp_outtap_20.incl. As with the distributed input mechanism, the output of distributed 2- and 3-dimensional arrays is handled with a call to an RSL routine, in this case RSL_WRITE. Distributed data is collected from the other processors and assembled on node0, where it is written to file using a Fortran unformatted write statement. As with model input, all non-distributed access to the output file is performed in RSL_IAMMONITOR() conditional code.

The header record for each new frame (time period) in the history file is written by node zero from the JUNK arrays that are set in OUTTAP prior to the include of mpp_outtap_20.incl. Note dependence on the fact that node zero is the only node that has a complete copy of the header information, because of the way the input header is read in MPRDINIT.

The MPEQUATE routine (MPP/RSL/paralle_src/mpequate.F) is a special purpose array-copying routine analogous to EQUATE (domain/util/equate.F) in the non-DM code. It is used only to zero out the extra memory in the array around the actual subdomain to prevent garbage data from showing up in the last row and column of cross-point arrays in the output records.

The other peculiar aspect of the DM-parallel implementation of OUTTAP is the use of MP_DOMAINSTAT and MP_INITDOMAIN at the beginning of OUTTAP and then again at the end of mpp_outtap_20.incl. This mechanism provides a way of setting the RSL I/O and loop-control data-structures for a particular domain within the routine and then restoring them to the domain of the caller on return. The reason for this is that at model output time, OUTTAP is called from within OUTPUT (domain/io/output.F) successively for each active domain and the RSL-set run-time data-structures must reflect the correct domain at all times. The call to MP_DOMAINSTAT saves the currently active domain identifier in the integer variable ISAVEDOMSTAT (declared at the top of OUTTAP) so that this can be restored at the end of the mpp_outtap_20.incl file.

3.4. ITERATION STRUCTURE

Once I/O has been implemented and tested for DM, the next step is to implement the mechanism that will control loops over decomposed dimensions in the parallel code. These are the north-south loops, generally over the I-dimension, and the east-west loops generally over J. To a large extent, this is handled automatically by FLIC and the macro expansions in MPP/RSL/LMexp.m4, which modify east-west and north-south loops to use RSL-set data structures that specify the local start and end for the loop on each processor. What the programmer needs to do is ensure that the data structures are set properly and to ensure that FLIC hasn't overlooked certain special cases.

The RSL data structures that control loop iteration are set by calling MP_INITDOMAIN(INEST), where INEST is the index of the domain to be interated over (e.g., 1, for the mother domain). For the mother domain, this is called before the call to SOLVE3 for INEST=1 in Run/mm5.F. MP_INITDOMAIN is also called at certain other points as needed for switching between domains in a nested scenario, which we will not address at this time, since the first thing to do in a new parallelization effort is to get a single-domain running.

The special cases for FLIC are 1) instances where a subroutine is called from within a J-loop, 2) cases where the call to the subroutine is the *only* statement in the J-loop, and 3) miscellaneous instances where there are conflicting uses of a loop variable as local and global indices.

When a subroutine is called within a J-loop, say from within the large 640 loop in the SOLVE3 routine of Version 2, the J-index is passed in through the argument list. Since FLIC does not do interprocedural analysis, it has no way to know that J is a loop variable inside the routine (because the loop resides externally, in the caller). FLIC will erroneously assume that array references using J as index are loop-invariant and attempt to convert J from a global to a local index (when it is, in fact, already a local index, because FLIC converted the J-loop in the calling routine). The solution is simple: tell FLIC that the argument is J-loop variable. This is done in the MPP/FLICFILE, using the n=*subroutine*:*index* directive. The 'n' specifies that the directive refers to the 'n' dimension (east-west in MM5), *subroutine* is the name of the subroutine and *index* is the name of the loop index argument as it is known within the subroutine.

The second special case occurs when a subroutine is called within a J-loop and it is the only statement in the J-loop. This causes a problem because FLIC needs at least one array reference in the loop body to deduce whether the loop variable is indexing a decomposed dimension in order to translate the loop statement. The solution used in MM5 is to place a reference to a dummy array anywhere in the loop body:

```
DO J = 1, JL
   X = DUMMY(J)      ! INSERTED TO HELP FLIC
   CALL SUB( J, … )
ENDDO
```

This occurs only in Version 3, where the J-loop loops have been broken up into many smaller loops throughout the solver. J-loops in the Version 2 solver, in contrast, are few and cover many lines of code.

The third sort of special case occurs when a loop variable is used within the body of the loop both as a local index and a global index. An example of this occurs in the subroutine NUDGE (domain/boundary/nudge.F):

```
C
C-----INTERIOR J SLICES:
C
#ifndef MPP1
      DO 10 I=2,IP
#else
      DO 10 I2=2,IP
        I=NOFLIC(I2)
#endif
        FCX=FCOEF*XFUN(I)
        GCX=GCOEF*XFUN(I)
        DO 10 K=1,KD
C
C.....SOUTH BOUNDARY:
C
          FLS0=(FSB(J,K,I)+DTB*FSBT(J,K,I))-FB(I,J,K)
          FLS1=(FSB(J-1,K,I)+DTB*FSBT(J-1,K,I))-FB(I,J-1,K)
          FLS2=(FSB(J+1,K,I)+DTB*FSBT(J+1,K,I))-FB(I,J+1,K)
          FLS3=(FSB(J,K,I-1)+DTB*FSBT(J,K,I-1))-FB(I-1,J,K)
          FLS4=(FSB(J,K,I+1)+DTB*FSBT(J,K,I+1))-FB(I+1,J,K)
          FTEN(I,K)=FTEN(I,K)+FCX*FLS0-GCX*C203*
     +             (FLS1+FLS2+FLS3+FLS4-4.* FLS0)
```

In this case, the original code used the loop variable I both as a local index into decomposed I-dimension of array FB (dimensioned MIX,MJX,MKX) and array FTEN (dimensioned MIX,MKX) and as a global index over the narrow dimension of the two boundary-slab arrays, FSB and FSBT (dimensioned MJX, MKX,5). Furthermore, the index I is being used globally to compute coefficients relating to distance from a boundary (FCX and GCX). The solution is to recast the loop variable I as I2 and then assign I to the value of NOFLIC(I2). The NOFLIC directive prevents FLIC from tracing back through the assignment statement to determine that I is an expression of the loop variable I2 and thus prevents FLIC from incorrectly converting the DO 10 loop. As a result, I is a global index (running from 1 to 2) which is correct for indexing the undecomposed dimensions in the references to XFUN, FSB, and FSBT. The other references to FB and FTEN, in which I does index decomposed dimensions, are also handled correctly because FLIC considers I invariant with respect to any loop over a decomposed dimension (we have forced FLIC to ignore the DO 10 loop). So it handles the references to FB and FBTEN as it would any loop-invariant index of a decomposed dimension: it converts the index from global to local.

3.5. INTERPROCESSOR COMMUNICATION

At this point in the implementation of the parallel model we have

- installed the MPP directory and set up the model code to use the DM-parallel build mechanism,
- modified the main routine Run/mm5.F to initialize parallelism,
- modified the mechanism that constructs the parame.incl file to decompose memory arrays,
- set up model input and output, and
- implemented the iteration structure

so that the model should run on one processor in DM mode and produce reasonable (if not bit-for-bit) history output when compared with the same code compiled in non-DM mode. The remaining work involves implementing interprocessor communication so that it will run on multiple processors.

This section first describes how data dependencies are uncovered and how RSL interprocessor communicatoin constructs, called "stencils, are implemented.

3.5.1. Data dependency analysis

Data dependencies arise in MM5 as a result of, horizontal advection horizontal diffusion or horizontal interpolation between staggered grids (dot-cross) or for smoothing and nest forcing. A data dependency is a non-local use (appearance on a right hand side of an assignment statement) of a decomposed array whose index is some offset from an arbitrary point IJ in the domain; for example, I+1,J or I,J-1. Decomposed arrays may be considered to be in one of two states at any point in the code: valid or invalid for non-local use. Arrays are always valid for local use. The relevant operations on an array are:

- set – by assignment, input, etc. Invalidates an array for non-local use.
- non-local use – requires valid data on a stencil, a pattern of points around the local points
- stencil-exchange – interprocessor communication to update the stencil.
- runpad computation – computation onto the halo to update the stencil.

When an array is invalidated by a set, it cannot be used non-locally again until it is updated – that is, until the halo regions of the array are made to contain the updated values stored locally on other processors. This is done either through communication with the other processors, or by duplicating the operations that set the non-local values of the array by computing out onto the halo. Both strategies are employed in MM5. Through the course of an MM5 time step, arrays are set from non-local references to other arrays and are then themselves used nonlocally. The relationships between variables can be described through set-use chains (sometimes called def-use chains).

Because the cost for initiating a message between processors (latency) typically dominates in the overall cost of communication, there is considerable performance advantage to aggregating messages into few stencil exchange operations that communicate many fields at once. The RSL library is designed to facilitate message aggregation. However, effort is required to analyze the set-use chains in a program and exploit the opportunities for aggragation.

For a given section of code, there will be a set of arrays that will be subject to non-local use at some later point in the code. Of this set, a subset of these arrays will have been invalidated by a set at some earlier point in the code. It is this subset of arrays that must be updated and to minimize latency, we wish to do it with as few exchanges as possible. This subset will have a last-set point in the code and first non-local used point. For a subset to be aggregated in a single exchange, the last-set point must occur before the first non-local use, and the stencil exchange must occur at some point between those two points. If the last-set point occurs after the first-used point, the subset will need to be divided into smaller subsets and each of these subsets will require its own exchange.

3.5.2. Setting up RSL stencil communication

The principal interface file to the RSL communication library is MPP/RSL/parallel_src/define_comms.F. This file contains the calls to RSL that register MM5 data structures, binding them to stencil-descriptors that may then be used within the code to update the halos of the arrays on each local processor subdomain.

Note for AER presentation: Go through the RSL Relaxation example at this point (PowerPoint slides)

The define_comms.F file defines the following stencils:

Version 2				
Stencil name	**Shape**	**Purpose**	**Affected arrays**	**Used**

`sten_init`	RSL_24PT	Initialize pad areas for 2- and 3-d arrays, including constant arrays	Very many	`mprdinit.F`
				`mp_feedbk.F`
				`mpp_bdyval_10.incl`
				`mpp_init_10.incl`
				`mpp_initnest_30/33/37/40/50.incl`
				`mpp_initsav_10.incl`

`sten_a`	RSL_12PT	update pad areas at the start of a new time step	PS, U, V, T, W, PP, Q, (A&B time-levels)	`mpp_solve3_10.incl`
`sten_b`	RSL_12PT	Update after these are modified in DO 90 loop	UA, VA, THA, PR1, RHO1, TBP	`mpp_solve3_20.incl`
`sten_c`	RSL_8PT	Update velocity tendencies after influence of PBL	UCD, VCD	`mpp_solve3_40.incl`
`sten_d`	RSL_8PT	Update qdot for computation of dot-point averaged divergence (DO 961)	QDOT	`mpp_solve3_50.incl`
`sten_sa`	RSL_8PT	Update fields going into sound	U3D, V3D, T3D, PP3D, and T3D Slow tendencies for U,V,PP,W, B-time level values of T, QV, and PP	`mpp_sound_20.incl`
`sten_sb`	RSL_8PT	Update fields within minor time loop of sound	U3D, V3D, PP3D, TB, QVB	`mpp_sound_30.incl`
`sten_e`	RSL_12PT	Update UA and VA after influence of sound	UA, VA	`mpp_solve3_60.incl`

4. Module specific descriptions

4.1. SOLVER

4.1.1. Communication points

4.1.2. Loop control

4.1.3. Performance measurement (upshot and milliclock)

4.2. PHYSICS

4.2.1. General

4.2.2. PBL schemes with miter loops

4.2.3. Implementation

Implications for bit-for-bit agreement

4.2.4. Radiation schemes with CCM-data structures and loop constructs

4.2.5. Routines that are called within a J-loop

Notation in FLIC file

May require a dummy assignment to a J-array

4.2.6. Other physics

4.3. INPUT AND OUTPUT

4.3.1. Rdinit.F

4.3.2. Rdter.F

4.3.3. Bdyin.F

4.3.4. In4dgd.F

4.3.5. Outtap

4.3.6. Restarts

1. Make uninstall, make mpp, and reproduce the error
2. Use interactive debugger to determine where fault is occurring; then make mpclean and recompile with –g debugging and fine the line number
3. Deterimine if the fault is in a model routine, the RSL library, or some routine you've never seen before (a thread or operating system routine). Note that errors in malloc and free may be caused by memory overwrites or simply running out of memory (stack, heap, or thread-stack) on a processor
4. Determine if the fault still occurs without the most recent change to the model code
5. Determine if the error occurs in the same place every time or if it is non-deterministic
6. Determine which processors incur the fault
7. Determine if the non-DM code works correctly on the problem (if possible)
8. Determine if the problem occurs on other data sets/problem sizes

7.2. NaN's and floating point exceptions

7.3. Errors in model results

Conditions under which the DM-parallel version should be expected to give bit-for-bit identical results with the non-DM version: same processor hardware, same compiler, all modules compiled without optimization, no fast libraries, and in the case of routines with mitre loops, the CPP constant BIT_FOR_BIT_KLUDGE #defined at the top of the routine (HIRPBL and G-S PBL. Warning, never run operationally with this defined).

1. Determine if the DM-parallel version of the model gives different results on different numbers of processors. Visualize output fields and look for tell-tale "windowpane" patterns in the data itself or in difference plots with output from runs on diffferent numbers of processsors or from the non-DM model. This may indicate a missed stencil-exchange on an array. Hint: many such errors eventually trace back to stencil definition code in MPP/RSL/parallel_src/define_comms.F.
2. Use difference plots to determine if results are correct on processor zero but incorrect on other processors. This might indicate failure to broadcast a constant, some part of the namelist data, or some part of the input data file header. Note that MM5 does not broadcast the character-string information from the MM5 input file header to other processors, only the REAL and INTEGER header fields[6]. Therefore, MM5 model subroutines should not evaluate conditionals using this character information because it will not be available on all processors. If such a test must be performed, evaluate the conditional on processor zero and then broadcast the logical result of the test to the other processors.
3. Use difference plots and select a point in the domain known to diverge deterministically, identify it's i, j, and k indices and then systematically begin inserting print statements for that point, tracing backwards though the code to the source of divergence. This is a powerful technique but it may consume a good deal of computer time and human effort. See the next section for additional information on this technique.
4. Keep in mind that the error might be *anywhere*, including in the tools that you are using to post-process the data or in the model output routine of the model. This can show up as errors in output that seem too extreme for the model to be able to run, when in fact that data inside the model is fine.

7.4. Inserting print statements to compare data in DM and non-DM runs

When all else fails, it may become necessary to begin looking at and comparing the values of arrays between a reference version (usually the non-DM version or the DM-parallel version on a different number of processors) and the suspect code. Here are some things to keep in mind when debugging in this manner:

1. Remember that what the compiler sees is the .f files in the MPP/build directory but what you edit are the .F files.
2. Array indices between the DM-parallel and non-DM parallel versions will not correspond (but there are easy ways around this)
3. If you print the value of a single I,J point, remember that the print will only produce valid output from one processor, not all of them (because the points are distributed).
4. Write your debugging output to standard error rather than standard output, because on most machines, standard error is not buffered. In other words, use WRITE(0,*) rather than PRINT*.
5. Loop variables used in conditionals will be converted from local to global, as long as they are not hidden from FLIC by an intrinsic (MAX or MIN) or by the NOFLIC macro.
6. Loop variables used as array indices are always local indices.

---

[6] The reason character information is not broadcast through RSL is because strings are represented differently on different vencors machines and compilers. The Cray T3E representation is problematic in this regard.

7. Loop invariant variables used as array indices are converted by FLIC are considered global indices and will be automatically converted to local indices.
8. Output mechanism may be the problem.
9. Don't blindly assume the reference code is the correct code. It has happened, albeit infrequently, that a discrepancy between the two has turned out to be a problem with the non-DM version.

The remainder of this section will describe the author's favored technique for debugging the code using print statements. Some of this is a matter of personal preference but it will help to illustrate an effective technique identifying pitfalls and showing works around.

The first thing one does is to find the I,J,K coordinates of a point that is discrepant compared with the reference version. Since the discrepancy was probably first observed in model output, the obvious place to begin looking is in a difference plot. Unlike data visualization for scientific analysis or forecasting, a good plotter for debugging should *not* smooth or contour the output and it should provide an easy way for identifying the indices of individual points within the data. HMV is a freeware tool favored by the author, please refer to the Appendix. Remember that I increases from south to north, J-dimension increases from west to east, and K increases from top to bottom. Assume for the sake of example that the point we wish to instrument is I=12, J=7, K=23.

Identify the routine that you wish to begin your search. This is usually SOLVE (SOLVE3 in MM5v2) or SOUND. Find the place in the routine where the output variable you're interested in is set and begin inserting print statements here. Many of the prognostic variables (UA, VA, TA, WA, PPA) in MM5 are set in SOUND; the moisture variables (QVA, QCA, QRA, …) are set in SOLVE, prior to the call to SOUND. In all cases a tendency array for the variable is a major term in the new value for a prognostic variable – for example, U3DTEN for UA. The tendency arrays are what are actually accumulated over the course of a time step so debugging will usually start with these. The author's favored place to start with debugging statements is immediately before the call to the SOUND routine, since all tendencies are available at that point. Also, the SOUND routine in MM5 is quite complicated to debug. If one can make the initial determination that a problem occurs before the call to SOUND, one may be able to avoid plunging into this particularly difficult routine.

Begin by placing three CPP preprocessor definitions at the beginning of the solve.F file, above the SUBROUTINE SOLVE declaration:

```
#define IDEBUG 12
#define JDEBUG 7
#define KDEBUG 23
      SUBROUTINE SOLVE(. . .)
      . . .
      WRITE(0,*)'U3DTEN   ',U3DTEN(IDEBUG,JDEBUG,KDEBUG)
      WRITE(0,*)'V3DTEN   ',V3DTEN(IDEBUG,JDEBUG,KDEBUG)
      . . .
      CALL SOUND(IYY,JXX,UB,VB,. . .
```

To write out the values of the tendency arrays before the SOUND routine, insert write statements as shown in the example. This makes it easy to change the IJK point without modifying the write statements. Once debugging statements are in place in the code, make the DM-parallel and non-DM versions by typing 'make mpp' and 'make' and then run the resulting mm5.mpp and mm5.exe programs. The standard error output from the DM version is automatically captured in the file rsl.error.xxxx (where xxxx is the number of the processor containing the point IJK). Capture standard error from the non-DM version into a file "errlog" using the sh (Bourne shell) command and then compare the rsl.error.xxxx and errlog files using the UNIX diff utility .

```
sh –c "mm5.exe 1> runlog 2> errlog"
```

The write statements in the first example are outside any decomposed loop. The next example shows how to print the value of point within a loop. The IF statement is used to test when to execute the debugging write statements.

Even though the loops and arrays or over local memory indices on each processor, the writes will occur for the correct IJK – that is, point corresponding to the same IJK in the non-DM code, because FLIC automatically converts the uses of I, J, and K in the conditional statement to global coordinates before the comparison with IDEBUG, JDEBUG, and KDEBUG occurs. (To see how the conversion occurs, examine the corresponding section of code after FLIC has translated the statements in the MPP/build/solve.f file.) As with the earlier example, the write statements will only execute on the processor that contains the point IJK in local memory. Keep this in mind when changing the IJK coordinates of the point you are inspecting since you may end up moving the output to a different processor, and thus to a different rsl.error.xxxx file. Refer to the show_domain_0000 file to see how the domain is decomposed over processors in your run.

```
      DO J=JBNES
        DO K=1,KL
          DO I=IBNES,IENES
       IF (I.EQ.IDEBUG.AND.J.EQ.JDEBUG.AND.K.EQ.KDEBUG) THEN
         WRITE(0,*)'DEBUG UB ',UB(I,J,K)
         WRITE(0,*)'DEBUG VB ',VB(I,J,K)
       ENDIF
            <statements in loop>
          ENDDO
        ENDDO
      ENDDO
```

If, based on the first set of prints before SOUND, one is able to determine that the problem is in a tendency array, then an alternate debugging technique may be employed to more quickly narrow down the source of the problem. One may begin selectively removing (commenting out) calls to physics, advection, or diffusion routines that modify tendency arrays earlier in SOLVE. When the problem disappears in the difference plot, its possible that the bug is in, or at least traveling through, the subroutine that was removed from the computation. When you comment out a routine, put a WRITE(0,*) statement in its place warning that the routine has been removed.

Dumping whole fields.

8. Appendices

8.1. LIST OF MPP RELATED #IFDEFS AND #INCLUDES IN CODE

8.2. USEFUL TOOLS FOR LOOKING AT DATA FOR DEBUGGING PURPOSES

8.2.1. Diffv3

8.2.2. HMV

http://rotang.com/HMV

8.3. USEFUL TOOLS FOR PERFORMANCE MEASUREMENT AND BENCHMARKING

8.3.1. Mm5etime

8.3.2. Stats

8.3.3. Gprof

8.3.4. Jumpshot