

**Partial Overhaul and Initial Parallel Optimization of
KINETICS, a Coupled Dynamics and Chemistry
Atmosphere Model**

Howard Nguyen
NASA Jet Propulsion Laboratory
Major: Astrophysics
USRP Fall 2012
21 December 2012

Partial Overhaul and Initial Parallel Optimization of KINETICS, a Coupled Dynamics and Chemistry Atmosphere Model

Howard Nguyen¹

San Francisco State University, San Francisco, CA 94132

Karen Willacy² and Mark Allen³

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

KINETICS is a coupled dynamics and chemistry atmosphere model that is data intensive and computationally demanding. The potential performance gain from using a supercomputer motivates the adaptation from a serial version to a parallelized one. Although the initial parallelization had been done, bottlenecks caused by an abundance of communication calls between processors led to an unfavorable drop in performance. Before starting on the parallel optimization process, a partial overhaul was required because a large emphasis was placed on streamlining the code for user convenience and revising the program to accommodate the new supercomputers at Caltech and JPL. After the first round of optimizations, the partial runtime was reduced by a factor of 23; however, performance gains are dependent on the size of the data, the number of processors requested, and the computer used.

I. Introduction

The main concerns for any numerically laborious program are stability, accuracy, speed, and minimal errors. KINETICS, a FORTRAN program developed at Caltech and JPL, simulates various atmospheric models and computes its chemical composition. Like many other computational fluid dynamics codes, it models the fluid by using grids, is computationally burdensome, and is data heavy. The main numerical method used in KINETICS is based on Prather's advection scheme². Prather's algorithm that conserves the second-order moments of the tracer distribution during advection¹ was expanded to include higher-order moments². It turns out that conserving second-order moments is more than adequate accuracy-wise for small grid cell sizing, and higher-order moments are only necessary for larger grid cell lengths². Other improvements or additions in KINETICS are eddy diffusion, chemistry, and accounting for the sphericity of the atmosphere². A major reason KINETICS is using Prather's scheme over other advection algorithms is the use of integration in the calculations rather than the finite-difference procedure². The integration method allows the program to store additional information about the tracer distribution at each time step². Prather pointed out that some of the other advection schemes might outperform his method in one or more of the following criteria: stability, accuracy, speed, and errors; but his method is better balanced¹. To verify the accuracy and stability of the model with the additional features, the sequential version of KINETICS has been tested with six benchmark cases where the analytical solutions are known². The other significant criteria to improvement is reducing the total runtime. It is apparent that scaling KINETICS up to a parallelized version will offer the greatest results in terms of speed.

The use of a supercomputer could greatly reduce the total runtime of a program, but it requires some foresight and understanding of the networks to utilize it efficiently. In the mid-2000s, there was a dramatic change in the CPU architecture; chip manufactures started putting multiple cores on a single die rather than increasing the single processor's clock speed. Afterwards, there seemed to be a clock speed plateau at around 3.4 ghz. The main reasons for the shift from more powerful single cores to less powerful multi-cores are power consumption and heat. The multi-core architecture would offer a better performance-per-watt. It is also worth mentioning that the total

¹ USRP Intern, Space and Astrophysical Plasmas, NASA JPL, San Francisco State University.

² Research Scientist and Mentor, Space and Astrophysical Plasmas, M/S 169-506, NASA JPL

³ Principal Research Scientist, Planetary Science, M/S 183-801, NASA JPL.

clock speed for multiple cores do not add linearly; there is overhead required to manage the multiple cores. It looks as if the future of computation will rely on parallel computing. Many groups have taken notice leading to a slew of parallel APIs like MPI, OPENMP, OPENCL, Pthreads, CUDA, STREAM, C++ AMP and more, created to assist the programmer in their parallel executions. While there are many APIs for parallel computing, many of them target different aspects of parallel programming. Some are designed for CPUs while others are designed for GPUs. Some are used to manage work in a single node and others are used for multiple nodes. There are many APIs that are also computer language exclusive. The one used in this project is MPI, which is the standard compiler wrapper for node-to-node communication. Even though the performance of CPUs has been increasing each year, the rate of advancement for memory and networking hardware are substantially lower; thus, the performance gap between processor and communication technologies also increases each year. This means that the main bottlenecks for parallel computation might not necessarily come from the amount of calculations, but rather from the transferring and storing of data. Parallelization will be necessary in the future, and the understanding of architecture and networks are important to hone the program.

The main objective of my internship was to speed up KINETICS by optimizing the parallelized sections. There was also an added priority for user convenience, which includes aspects such as readability, ease of understanding, and ease of use. The two main researching groups that will use this program are at JPL and Caltech. KINETICS needs to be altered to accommodate for each supercomputer’s environment and interoperability between the following compilers: PGI, GNU, Absoft, and Intel. Thus, a second objective of partially overhauling KINETICS was established. Section 2 will cover in detail the incentive for overhauling and section 3 will go over the initial round of optimizations.

II. Overhaul

Originally, KINETICS was written for VAX/VMS machines, which in the past had low memory when compared to modern standards. To accommodate for the lack of memory, each subroutine had to be separated into individual files and then overlaid. Overlaying is a method where the individual parts are brought into memory one at a time. Overlaying also allows the programmer to customize the code for specific conditions or options by overwriting the preferred subroutine over other subroutines of the same name. Tasks such as splitting, linking, copying, moving, making directories, removing, and archiving were orchestrated in a seemingly convoluted dance among several scripts and makefiles in order to overlay.

Since most computers that are used for scientific research today are using UNIX rather than VMS, the need to streamline the code for user convenience was a high priority. The major goal was to purge the program of the unnecessary segments that involves splitting and overlaying. Same-named subroutines had to be condensed to a single subroutine or removed. The essential pieces of code from the complicated scripts and makefiles had to be identified, then stripped and incorporated into a new command script and a new Makefile. The command script was a project-requested file that acts like a control center where the user can choose different simulation options and basically run any combination of available modes. Figure 1 is a snippet of command script and it lists most of the available options a user can choose. The other options were not shown because they are in another area the command script such as choosing the number of processors to run. The idea was that the command script would be the only file that the user needs to edit. It would create additional scripts or files the computer environment requires it, and the chosen conditions and variables would be piped to the Makefile and other scripts. As for customizing the instructions for particular conditions, C preprocessor commands, also known as pragmas, are a standard feature on most, if not all, modern FORTRAN compilers. These directives start with a hashtag like #ifdef and #end. The compiler would comb through the code before compilation and omit the pragmas and the lines between them if the specified condition was not satisfied.

```
#####
LOCATION=CALTECH           to
#LOCATION=JPL
#LOCATION=NEITHER

#BUILD_TYPE=absoft
#BUILD_TYPE=intel
BUILD_TYPE=gnu
#BUILD_TYPE=pgi

#MODEL_TYPE=__ISM
#MODEL_TYPE=__MARS       in
MODEL_TYPE=__TITAN
#MODEL_TYPE=__EARTH

MPI=YES                   if
#MPI=NO

NETCDF=YES
#NETCDF=NO
#####
```

Fig. 1 Simulation options.

There were many challenging facets of this project. While some problems were anticipated, others were deceptively troublesome. The first two major hurdles were getting acquainted with the code and purging the VMS-

specific parts. It was expected that the first few passes through the code for familiarity would be rough, but the elaborate web of interactions between multiple scripts and makefiles added a whole other level of complication. At one point, the files just had to be printed out and examined line by line. After identifying the essential parts, it was decided that it would be more favorable to start the scripts from scratch rather than mending the previous files. In this case, the script would start with a clean state and every line appended to it would be known and understood.

Table 1. Environmental Differences Between the Computers at JPL and Caltech.

	JPL	Caltech
Dynamical Environments	modules	paths
Fortran Compilers	PGI Intel GNU Absoft	PGI Intel GNU
Compilers linked to MPI	PGI Intel GNU	Intel
MPI API	MPICH2	SGI or MVAPICH
Job Submission	PBS	none

Just as in the first two obstacles, the problems with familiarity and complications are now applied to computer environments and scripts. In terms of hardware, the code will be used on ZODIAC at JPL and GANESHA/FUSI at Caltech. The difficulties lie in the user-convenience purpose of the command script. The capability to have all commands in one script proved to be more difficult than anticipated. Another issue is that the environment setting for the two supercomputers are vastly different. Table 1 highlights a few differences between the environments of each machine. Usually, the environment would be set in the login/start-up files such as .bashrc, .profile, .chsrc, and .bash_profile. The program’s makefiles and scripts would be executed in the primary shell. With a command script, all executions are done in a subshell. The complication comes from the passing of environment setting and variables, similar to the issue with scope and functions. Not all environment settings and variables are automatically passed when a parent process (primary shell) forks a child process (subshell). There are commands to explicitly pass variables; however, the passing is in one direction because any changes made to the environment in the child process cannot be passed back to the parent. In the case of having the command file, the environment has to be set in the subshell’s scope and passed

to any other script like a Makefile or PBS script. The simulation particular variables had to be defined in the command file, then piped to the Makefile for specific-compiler flags, preprocessor defined variables, and libraries. Using dynamical environment software like a module is an example of an environment setting that is not passed from parent to child. The module path had to be sourced in the PBS script if modules are used within that process. GANESHA/FUSI does not use PBS or modules, so there is minimal confusion with subshells. Instead, paths to specific compiler configurations have to be built prior to linking. For ZODIAC, a PBS script is required so the entire environment has to be created/alterd in the command script and later passed along. If the JPL location was chosen, the command script will create a PBS script and an environment file from which both will source. The command script needs to be able to execute any combination of the options shown in Fig. 1 and an alternative course of action has to take place if the combination is invalid like choosing a compiler that is not present on the computer.

III. Optimization

While there are many fronts in optimization, the two most common targets are the numerical calculations and the communication. Since the algorithm has already been chosen, this section will primarily focus on the communication with MPI and FORTRAN. Ideally, a fully parallelized program would rarely have its processor communicate with one another. An ideal parallel process is pictorial shown in Fig. 2a). In our case, a less-than-ideal semi-parallel execution, Fig. 2b), was used for ease of understanding. The program would be running sequentially and when it encounters a section with heavy calculations, the root processor will split up the work and send it to the other processors. After each processor completes its part of the calculations, the root processor would then collect the necessary data. As we can see, the scattering and gathering of data along with having idle processors is not the most efficient use of resources; however, it is easier for people to follow and troubleshoot.

The timings for communication calls can be generalized into two groups: latency and bandwidth. Network latency is the timing cost of each message and is data-size independent. It is analogous to the postage stamp cost of sending a letter. Bandwidth is the speed at which data can be transferred, somewhat like the price per pound to a parcel. Typical units of measurements for latency are nanoseconds or microseconds and MB/s for bandwidth. sources that affect these timings are mostly hardware dependent; therefore, little can be done in a code to reduce these fundamental values. The area that programmers can control is the total runtime, which is a direct consequence of the communication calls. Normally, the cost from latency is much greater than the from bandwidth, but there is a tipping-point where size the message will define which is more influential. For small messages, latency dominates the total time it takes send the message and for large messages, bandwidth has greater effect. It is unfavorable to send many small messages. Grouping the smaller messages together and sending the package will save more time. Large messages can be sent individually because the latency be insignificant and it might take more time to copy the to a buffer.

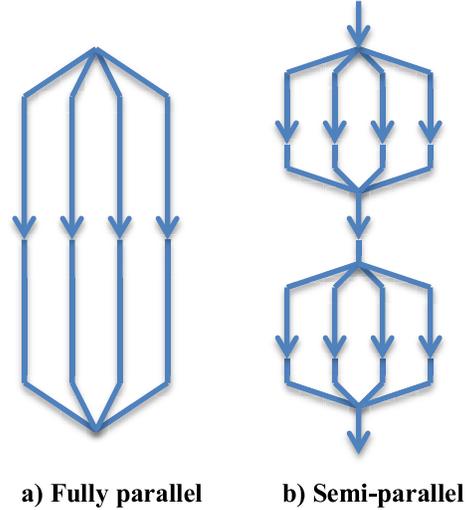
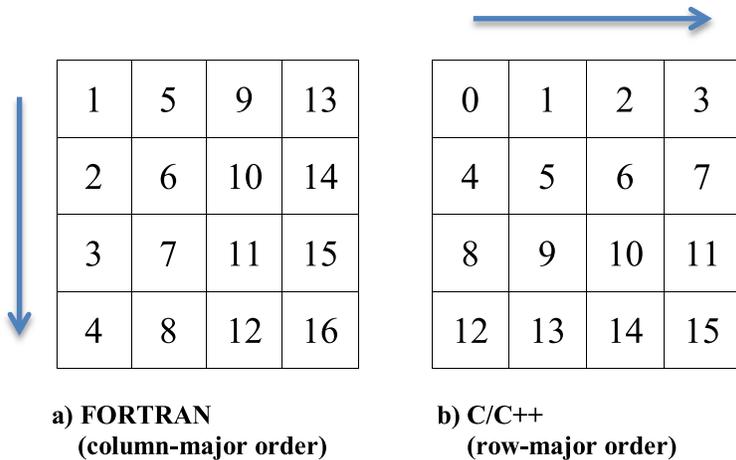


Fig. 2 Parallel executions

I believe one of the most important factors in minimizing communication overhead is taking into account how the computer language stores data into memory. The reason is the nature of how MPI sends data. For example, consider the following MPI_SEND commands for FORTRAN and C respectively:

```
MPI_SEND (buffer, count, datatype, dest, tag, comm, ierr)
MPI_Send (&buffer, count, datatype, dest, tag, comm)
```

It is explicitly shown in the C version, with the “&”, that the first argument is the address of the message being sent. The same case is true for the FORTRAN version. The second and third argument tells MPI the number and datatype of the message. These arguments are for MPI to know where to look in the memory and how far down the strip it should go for the complete message. This shows that MPI could only send contiguous blocks of memory. If we want



to send the minimal number of messages with the least overhead, then arranging the calculations and data around contiguous sets of data is the best direction. Since most sets of data will be in the form of an array, figures 3a and 3b show how FORTRAN and C store a two-dimensional array of data into contiguous strip of memory. The difference may seem trivial at first, but it is very crucial because it dictates the best way to work with data. This allocation method still holds true for multidimensional arrays that are greater than two. For FORTRAN, the best method to split up the data into blocks of contiguous memory is to prioritize dividing up the rightmost index and continue leftward if necessary. For

Fig. 3 Memory allocation sequence

example, with an array of shape (3,6,4,8), we would want to split up the farthest right index, which has the value 8. When calling 4 processors, the data should be split up into four arrays with the shape (3,6,4,2). If the number of processors requested is greater than 8, then we will split up the rightmost index and the index to the left of it. For shape (3,6,4,8) and calling 16 processors, we would want the data grouped as (3,6,2,1) to keep the data contiguous. The idea is never divide the leftmost index when programming in FORTRAN. These criteria spur other practices

like keeping the largest, divisible variable in the rightmost index and using the leftmost index for data that should not be split.

There are situations where we are forced to work with noncontiguous data. Two likely scenarios are when the data in the array are not independent of each other such as a FFT (fast Fourier transform) or when parallelization was not considered when assigning variables to specific array indices. The latter is our case because KINETICS was developed before parallel computing became mainstream. The initial round of parallelization that was done on KINETICS was sending a group of arrays to each processor before the calculations and afterwards the root processor would retrieve the essential data one element at a time. The method was a rough, but exceedingly safe procedure. Every processor had the data it needs for the calculations and the root processor fetches the correct amount of data. The major problem is the overabundance of communication calls that create very large bottlenecks, which drastically slows down the speed. The solution is rather straightforward; just send larger chunks of data. Implementing the solution, however, requires a bit more work. Since MPI can only send chunks of contiguous data, we are forced to create some type of mechanism that can store the noncontiguous data as contiguous. The puzzle is getting the root processor to also know where to store each piece of data after it receives the package. Fortunately, the MPI developers provided a set of tools to accomplish this. The core of the procedure involves the four following commands:

```
MPI_TYPE_CREATE_SUBARRAY(dim, arr_sizes, arr_subsizes, arr_starts, order, datatype, oldtype, ierr)
MPI_TYPE_GET_EXTENT(datatype, lb, extent, ierr)
MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype, ierr)
MPI_TYPE_COMMIT(newtype, ierr).
```

With the above commands, a new type of datatype would be created. `MPI_TYPE_CREATE_SUBARRAY` will create a datatype that is a subarray of the full array. This is the command that creates a contiguous platform. `MPI_TYPE_GET_EXTENT` will return the lower bound and extent or stride of the datatype. Because different datatypes will occupy different amounts of space in memory, the extent is required to know the spacing for each element. `MPI_TYPE_CREATE_RESIZED` is the actual command that reshapes the previously created datatype with the lower bound and stride provided by the first two commands. The last command, `MPI_TYPE_COMMIT`, allows the newly created datatype to be used in communication calls. With these new datatypes, noncontiguous sets of data from the full array can be stored, then sent and retrieved. This overhead would only need to be paid once.

A sample FORTRAN code that contains these commands is provided in the Appendix under Noncontiguous Data. This is a toy code, where hard numbers replace all of the general functions and allocatable arrays. The intent was not to make it easy to read and identify the MPI commands. The actual implementation is much more involved. Arrays such as displacement and shape would need to be calculated. Another point that should be noted is that MPI has a C bias. In Fig 3a) and Fig. 3b) the number ordering is different. FORTRAN's intrinsic number ordering starts at 1 and C's ordering starts at 0. The start and displace array in the sample code is for MPI commands and thus require the 0 start. One has to keep track of whether the part of the code requires MPI/C indices or FORTRAN indices. Also included in the Appendix is the contiguous sample of the same data. The `gatherv` and `scatterv` calls were used in the contiguous version only for consistency when compared to its noncontiguous counterpart. The basic MPI calls such as `send` and `recv` can be used with working with contiguous data. With both versions, it is easier to see the overhead. On the right side of each code is an output.

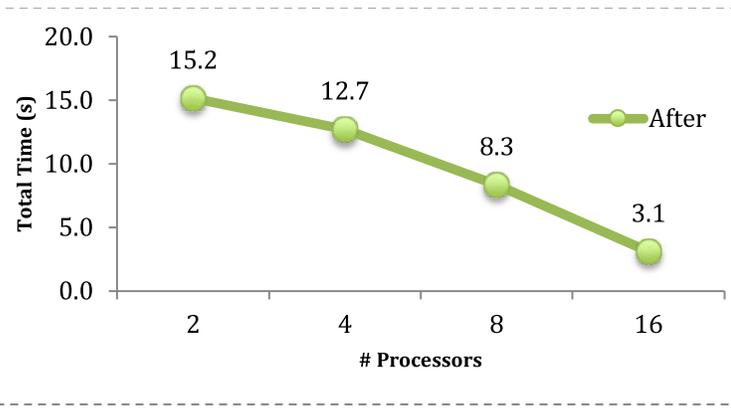


Fig. 4 Partial Runtime

With the new datatype and a few other changes, there was great improvement in speed. When comparing the before and after partial runtimes, there were almost no difference in time with two processors. The partial runtime when using four processors showed a significant reduction, from 302.1 seconds to 12.7 seconds, by a factor of 23.7. These numbers reflect the average time, among the four processors, that was spent on executing the first major

bottleneck. The timing function used for these results was `MPI_WTIME()`. The reason for a partial runtime instead of a total runtime was because some of the other bottlenecks in the code have not been resolved. There are many other factors that needed to be sorted out before attempting to alter those sections. The time comparison between the before and after for number of processors greater than four is not necessary because the before run had an upward sloping trend while the after, shown in Fig. 4, has a decreasing slope.

IV. Future Work

The solution presented in the previous section is merely a hot-fix to address the current largest bottleneck and a rework of the parallelization might be required to achieve the best results. Many of the problems were caused by the method of splitting the data and work in a noncontiguous fashion. The formulas that are doing the division of work needs to be revised for contiguous data and to account for scenarios where the number of processors does not divide evenly into the amount to data. Another procedure to reduce the amount of communication is by scattering only the necessary data needed for the calculations. Work has been started to identify these arrays, but it has not been completed. When most of the apparent problem areas have been dealt with, running the program alongside a profiler will help identify other high traffic sections. Running a latency and bandwidth benchmark to pinpoint the critical size for each supercomputer will help with decisions on whether to group the data or to just send the data. It might be a lot of work, but it might be worthwhile to rearranging the order of indices on the essential arrays for ease of division and collection of contiguous data.

V. Conclusion

A lot of progress was made in the last four months. The overlaying and file splitting was removed from KINETICS. The several old scripts were replaced by a relatively easy-to-read command script and makefile. The user would not have to access any file other than the command file to change a simulation option. As for the optimization, the main cause for the upward sloping direction as the number of processors increase has been determined. A temporary solution was implemented and it has shown a speed increase of at least a factor 23 for the duration of the first bottleneck. More importantly, the temporary solution shows a downward sloping trend as the number of processors increase. Many other sources of possible stagnation have been identified and a few pathways have been established for the different scenarios.

Appendix

Noncontiguous Data

```

1   PROGRAM noncontig_2d                               % mpi90 noncontig_2d.f
2   IMPLICIT NONE                                     % mpirun -n 4 ./a.out
3
4   INCLUDE 'mpif.h'                                  My rank is 2
5                                                    25 31 37 43
6   INTEGER :: ierr, nproc, rank                      26 32 38 44
7   INTEGER :: m, n, i, dimen, tot                    27 33 39 45
8   INTEGER :: shape_big(2), shape_sub(2), start(2)
9   INTEGER :: big(6,8), sub(3,4), twin(6,8)         My rank is 3
10  INTEGER :: displ(4), counts(4)                   28 34 40 46
11  INTEGER :: temp_type, newtype                     29 35 41 47
12  INTEGER(KIND=MPI_ADDRESS_KIND) :: lb, extent     30 36 42 48
13
14  CALL MPI_INIT(ierr)                               My rank is 1
15  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)  4 10 16 22
16  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)   5 11 17 23
17                                                    6 12 18 24
18  shape_big = (/6,8/)                               ! shape of main array
19  shape_sub = (/3,4/)                               ! shape of subarray
20  start = (/0,0/)                                   ! start pos of subarray in C
21  dimen = 2                                         ! dimension of array
22                                                    3 9 15 21 27 33 39 45
23  IF (rank.EQ.0) THEN                               4 10 16 22 28 34 40 46
24    big = RESHAPE( (/i,i=1,48)/, (/6,8/) )         ! create matrix 1:tot
25                                                    5 11 17 23 29 35 41 47
26    WRITE(*, '(/A)') 'The big array:'             6 12 18 24 30 36 42 48
27    WRITE(*, '(8i4)') ((big(n,m), m=1,8), n=1,6)
28  END IF                                           My rank is 0
                                                    1 7 13 19

```

NASA USRP – Internship Final Report

```

29                                     2  8 14 20
30 CALL MPI_TYPE_CREATE_SUBARRAY(dimen, shape_big, shape_sub,          3  9 15 21
31 > start, MPI_ORDER_FORTRAN, MPI_INTEGER, temp_type, ierr)
32 CALL MPI_TYPE_GET_EXTENT(MPI_INTEGER, lb, extent, ierr)
33 CALL MPI_TYPE_CREATE_RESIZED(temp_type, lb, extent, newtype, ierr)
34 CALL MPI_TYPE_COMMIT(newtype, ierr)
35
36 counts = ((1, i=1,nproc)/)          ! number of 'items' being sent
37 displ = (/0,3,24,27/)              ! displace array for pos in C
38
39 CALL MPI_SCATTERV(big, counts, displ, newtype, sub, SIZE(sub),
40 > MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
41
42 WRITE(*, '(/A,I3)') 'My rank is ',rank
43 WRITE(*, '(4I4)') ( (sub(n,m), m=1,4), n=1,3)
44
45 sub = sub - 1                      ! alter data value for fun
46
47 CALL MPI_GATHERV(sub, SIZE(sub), MPI_INTEGER, twin, counts, displ,
48 > newtype, 0, MPI_COMM_WORLD, ierr)
49
50 IF (rank.EQ.0) THEN
51   WRITE(*, '(/A)') 'The twin array after -1:'
52   WRITE(*, '(8i4)') ( (twin(n,m), m=1,8), n=1,6)
53 END IF
54
55 CALL MPI_FINALIZE(ierr)
56 END PROGRAM

```

Contiguous Data

```

1 PROGRAM contig_2d
2 IMPLICIT NONE
3
4 INCLUDE 'mpif.h'
5
6 INTEGER :: ierr, nproc, rank
7 INTEGER :: m, n, i, tot
8 INTEGER :: big(6,8), sub(6,2), twin(6,8)
9 INTEGER :: displ(4), counts(4)
10
11 CALL MPI_INIT(ierr)
12 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
13 CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
14
15 IF (rank.EQ.0) THEN
16   big = RESHAPE( ((i,i=1,48)), (/6,8/) ) ! create matrix value 1:48
17
18   WRITE(*, '(/A)') 'The big array:'
19   WRITE(*, '(8i4)') ( (big(n,m), m=1,8), n=1,6 )
20 END IF
21
22 counts = ((SIZE(sub), i=1,nproc)/)
23 displ = (/0,12,24,36/)
24
25 CALL MPI_SCATTERV(big, counts, displ, MPI_INTEGER, sub,
26 > SIZE(sub), MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
27
28 WRITE(*, '(/A,I3)') 'My rank is ',rank
29 WRITE(*, '(2I4)') ( (sub(n,m), m=1,2), n=1,8 )
30
31 sub = sub - 1                      ! alter data value for fun
32
33 CALL MPI_GATHERV(sub, SIZE(sub), MPI_INTEGER, twin, counts,
34 > displ, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
35
36 IF (rank.EQ.0) THEN
37   WRITE(*, '(/A)') 'The twin array after -1:'
38   WRITE(*, '(8i4)') ( (twin(n,m), m=1,8), n=1,6 )

```

NASA USRP – Internship Final Report

```
39     END IF                               3  9
40                                           4 10
41     CALL MPI_FINALIZE(ierr)             5 11
42     END PROGRAM                           6 12
```

```
The twin array after -1:
0  6 12 18 24 30 36 42
1  7 13 19 25 31 37 43
2  8 14 20 26 32 38 44
3  9 15 21 27 33 39 45
4 10 16 22 28 34 40 46
5 11 17 23 29 35 41 47
```

Acknowledgments

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by NASA Undergraduate Student Research Program (USRP) and the National Aeronautics and Space Agency. Howard Nguyen would like to thank his mentors Karen Willacy and Mark Allen for the opportunity to work on this project and for creating a very welcoming and productive atmosphere. He would also like to thank Michael Black, the systems administrator at Caltech, for answering his questions and setting up compiler paths on the supercomputer at Caltech.

References

- ¹ Prather, J. M., “Numerical Advection by Conservation of Second-Order Moments,” *Journal of Geophysical Research*, Vol. 91, No. D6, May 1986, pp. 6671-6681.
- ² Shia, L. R., Ha, L. Y., Wen S. J. and Yung, L. Y., “Two-Dimensional Atmospheric Transport and Chemistry Model: Numerical Experiments with a New Advection Algorithm,” *Journal of Geophysical Research*, Vol. 95, No. D6, May 1990, pp. 7467-7483.