Geochemical Journal, Vol. 44, pp. 000 to 000, 2010

*NOTE*

# A general parallelization approach to improve computation efficiency in a global chemical transport model (GEOS-Chem)

ROKJIN J. PARK,[1] DAEOK YOUN,[1] SUNG HOON YOO,[2] DONG SEOK HAN,[2] BOO JOONG KANG[2] and EUL GYU IM[2]*

[1]School of Earth and Environmental Sciences, Seoul National University, Seoul 151-742, Korea
[2]Division of Computer Science and Engineering, Hanyang University, Seoul 133-791, Korea

(*Received July 22, 2009; Accepted December 10, 2009*)

Despite increasing processing power of systems, sequential programs have limitations to achieve corresponding performance gains. Parallelization of sequential programs can exploit the computation potential to reduce overall execution expenditures. However, parallelization of a very large legacy program arises great challenges, because 1) former design of the program did not consider parallelization, 2) the large-scale program has been developed for long time even decades, and 3) some developers of the program are no longer available. In the atmospheric science field, there are many such legacy programs and GEOS-Chem model developed in Harvard University is one of examples. In this paper, we suggest simple procedures of the MPI parallelization which can be easily applied to a large legacy program. We applied this method to the GEOS-Chem model which is already parallelized with the OpenMP method alone. Our test results showed that the good performance improvement can be obtained by parallelizing a small portion of the program.

Keywords: MPI, OpenMP, parallelization, GEOS-Chem, modeling performance

## INTRODUCTION

Despite increasing processing power of systems, sequential programs have limitations to achieve corresponding performance gains. Therefore, newly developed programs adopt various parallelization techniques and tools (Jarvis *et al*., 2006; Xu *et al*., 2007). However, there are many large-scale legacy programs which are not optimally suited to take advantage of the current advanced computation technology, and they are still widely used in scientific communities nowadays. Those legacy programs were developed without any considerations of parallelization techniques at first, and parallelization of those programs is quite challenging because 1) designers of the program did not consider parallelization at that time, 2) the program has been developed by many people for long time, and 3) some developers of the program are no longer available for parallelization.

In this paper, we propose a simple and efficient procedure to parallelize a large legacy program. Our method is a bit common and straightforward, and does not require in-depth background knowledge of the program itself. We introduce a general framework for parallelization of a large-scale legacy program. We mainly focused on finding parallelizable parts with large performance overheads and parallelized the selected parts. Basic steps of our approach are as follows: 1) get profiling data for the model to analyze performance overheads of each function, 2) based on that profiling data, select candidate functions for parallelization, 3) parallelize the selected functions, if possible, and 4) verify results of the parallelized program.

We applied our method to a state-of-the-art global 3-D chemical transport model (GEOS-Chem) to test the effectiveness of our procedure on performance improvement. GEOS-Chem is used to simulate and evaluate changes in atmospheric composition and pollutants with huge range of application to environmental issu (Harvard Atmospheric Chemistry Modeling Group, year). This model comprises many complex modules to account for various processes in the atmosphere and is now widely used in the atmospheric chemistry community. As we will show, our method is simple but effective to reduce computational time, and it is easily applicable to other large-scale legacy programs.

## PARALLELIZATION AND PROFILING

In this section, we will discuss some basic concepts of parallelization techniques and profiling methods.

*Corresponding author (e-mail: imeg@hanyang.ac.kr)

Parallelization refers to converting sequentially executed programs to parallel executable programs. There are two most widely used methods: Message Passing Interface (MPI), and OpenMP (Open Multi-Processing). The former is a set of programming interfaces used to exchange data among individual computing nodes in parallel computing, and the latter provides programming language directives for parallelization using multi-processors in a single machine. A profiling method is used to analyze overheads of functions in the program and provides a standard basis to select target functions which can be parallelized most efficiently.

*Parallelization*

Traditional programs have been written in sequential computing environment. The performance of traditional programs depends on the computing power of a single processor in a machine. Recently, the growth of a single processor's computation power has been limited, and this restricts the performance of the programs running that on a single processor machine. On the other hand, parallel computing uses multiple machines or multiple processors simultaneously to process a program. For parallel processing, data and/or program codes are divided and distributed to several independent machines, called "nodes" in cluster computing, so that each machine can process a part of the whole data or codes simultaneously with the other machines.

The potential speedup gained from parallelized program codes can be estimated by the Amdahl's law, originally formulated by G. Amdahl (Amdahl, 1967). The Amdahl's law is often used in parallel computing to predict the theoretical maximum speed-up using multiple processors. According to the Amdahl's law, maximum speedup follows the following equation (1).

$$S = \frac{1}{1 - P} \tag{1}$$

where $S$ is the speedup of the program and $P$ is the fraction of execution time for parallelizable part.

In clustering environments, Amdahl's Law cannot be directly applied because there are overheads of data exchanges when an application is executed in parallel. There are tradeoffs between advantages of parallel computing and increased data exchange overheads.

*Message Passing Interface (MPI):* In parallel computing, multiple computers cooperate to generate final results. Each node participating in parallel computing is in charge of its own computing partial data and transferring that data to other nodes. MPI is one of the "Message passing" model standards, and it is a language-independent communication protocol. Data exchange among computing nodes is an important step of parallelization using multiple nodes. Both unicast (point-to-point) and broadcast (one-to-many) data transfer mechanisms are supported in MPI. MPI has been standardized in MPI Forum, and currently MPI version 1 and 2 are widely used.

There are many available MPI library implementations in FORTRAN as well as in C, and we used MPICH2 library, which is one of most widely used MPI libraries. The original implementation of MPICH, called MPICH1, implemented the MPI-1.1 standard. As of 2008, the latest implementation is MPICH2, the adopted MPI-2.1 standard, but does not yet support data translation between different hardware architectures.

*Open Multi-Processing (OpenMP):* OpenMP is an application programming interface (API). With OpenMP, we can make a program run in a single multi-processor machine with shared memory. The major difference between OpenMP and MPI is whether operations are performed on shared memory in a single node or distributed memory in multiple nodes. As for MPI, programmers must design and implement applications including communication modules and computation modules using MPI libraries, while, as for OpenMP, programmers use directives provided by OpenMP and codes are automatically generated. Since execution codes are automatically generated using OpenMP directives, the generated codes may not be optimal codes in a performance aspect.

Source code implemented with OpenMP may have much higher execution overheads when parallelized with MPI, because communication overheads using shared memory in a single machine are usually quite different from those using distributed memory in multiple nodes in a clustering environment. Increases of data exchange time may give huge obstacles to make source codes generated with OpenMP to run in a clustering environment.

*Profiling*

In software engineering, performance analysis, or *profiling*, is the investigation of a program's behavior using information gathered while the program is executed. A profiling method can show execution flows of a program and can estimate the execution time and the number of executions of a certain function. In this paper, we used two kinds of profiling methods: function profiling and loop profiling.

*Function profiling:* Function profiling provides function-level performance analysis results, such as the number of calls of a function and overall execution time of a function. Most function-level profiler creates profiling results based on a raw debugging data file that is generated upon execution of program compiled with a special option, such as the "-pg" option of the gcc compiler.

Even though function profiling provides performance analysis information of each function, one of disadvantages of a function-level profiler is that it cannot esti-

mate overheads of fractions (e.g., loops) inside a function since the analysis granularity is a function.

*Loop profiling:* Except for the operations such as file input/output or disk input/output operations, overheads of functions are usually caused by iteration operations, i.e., loops, especially in large-scale modeling programs. If a function has several loops, a loop profiling method can find out performance overheads of each loop.

A loop profiling method (Moseley *et al*., 2007) is usually carried out for functions that need additional performance analysis. If a function has more than one loop, a loop profiling method is essential to find out estimated overheads of each loop.

## Parallelization of the GEOS-Chem Model Program: A Case Study

GEOS-Chem is a global 3-D chemical transport model driven by assimilated meteorological observations from the Goddard Earth Observing System (GEOS) of the NASA Global Modeling Assimilation Office (GMAO). It has been developed by many researchers at Harvard University since 1988 and used by more than 40 research groups worldwide as a research tool to investigate a wide range of atmospheric chemistry and climate issues. Because of the long history of development GEOS-Chem is a large legacy program, with the size of more than 150,000 lines of codes mostly in FORTRAN. We used GEOS-Chem version 7-04-09 with GEOS-4 assimilated meteorology, which is already parallelized with OpenMP, but not with MPI. In a single machine, the GEOS-Chem can utilize maximum processors using OpenMP. But, the performance improvement by the MPI method is highly required to utilize more resources on a computer cluster environment.

This paper explains processes and experiences with parallelization of GEOS-Chem model program, and the parallelization procedures in this paper can be easily applied to other legacy programs. We parallelize a part of the GEOS-Chem model program and show results in Subsection "Experimental results".

Because parallelization requires lots of human resources as well as system resources and takes long time to make an entire legacy program run in parallel, we need to select some of parallelizable parts of the program, i.e., some modules or functions that cause large execution overheads. By parallelizing best candidate modules for the MPI method, we can achieve overall performance improvement with less effort. Therefore, how to select target modules or functions is quite important. Since GEOS-Chem model program is widely used in the atmospheric research community, the impact of parallelization of the program is expected to be quite huge.

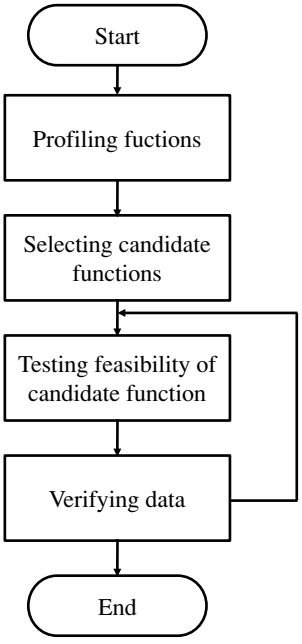


*Fig. 1. Overview of our parallelization steps.*

*Proposed procedures*

We used the following steps to make GEOS-Chem model program run in parallel environments. Figure 1 shows a brief overview of these steps. These steps can be easily applied to parallelization of other legacy programs.

*(1) Profiling of functions* There are many available profiling techniques and tools (Moseley *et al*., 2007), including Digital Continuous Profiling Infrastructure (Anderson *et al*., 1997), Oprofile, and gprof (Graham *et al*., 1982). Among these tools, gprof is most widely used to analyze programs. Tools such as gprof can estimate the execution time spent in each function as well as the execution time spent in its all children functions, and these tools can generate a call graph that shows each function's callers and callees, and the number of times each function is called. As a result of profiling, we can estimate the average execution time spent in each call of a function.

A function may have several loops, but gprof does not provide the execution time for each loop inside a function. So, more specific profiling method is needed. Loop profiling estimates elapsed time of a certain loop using timestamps at both the starting point and the ending point of the loop. In sequential execution testing, time-related functions of FORTRAN are used, and in parallel execution testing, the MPI_WTIME function provided by MPI is used to estimate execution time.

*(2) Selecting functions based on profiling results* Results of function profiling can be used to determine parallelization candidate functions. To select functions for

parallelization, we need to focus on specific fields of gprof results, such as "self seconds," which represents total spent time in a function, and "calls," which represents the number of invocations of a function. If we divide the value of the "self seconds" field by that of the "calls" field, we will get the average spent time of a specific function. If profiling results show that a function consumes long execution time in average, the function is a good candidate for parallelization.

Starting from the longest execution time in average, additional analysis, such as loop profiling, is carried out. Several functions of GEOS-Chem model program have triple or quadruple nested loops. If parallelization of these functions is possible, we can get significant performance enhancement.

*(3) Investigating feasibility of parallelization for selected functions* For the selected candidate functions based on results of function profiling and loop profiling, feasibility tests will be performed. To execute a function in multiple computing nodes, data must be decomposed. In this step, we look into data processed in a function, and examine possible execution time improvement with decomposed data.

The main target of parallelization is usually loops. In a loop, original data are loaded and used to calculate new results, and results are stored. If results from other nodes are used again in the same loop, this loop is hard to be parallelized because fetching results from different nodes in the same cycle becomes very expensive.

If a loop has double or triple nested loops, outer loops are easier to be parallelized. We also need to consider how data are stored or loaded into main memory. For analysis of feasibility, the followings should be considered:

- Data size: Size of data used in the loop.
- Data dependency: Range of affected data after execution of each loop.
- Data access pattern: Relationship between loop indexes and affected data's ranges.

Most data structures used in GEOS-Chem model program are three-dimensional or four-dimensional arrays, so a loop usually has double or triple inner loops. Triple nested loops perform operations on *X*-axis, *Y*-axis and *Z*-axis, and quadruple nested loops include more operations on the tracer-axis. We need to determine which axis of the data is to be decomposed. Before decomposing an array, we need to check dependency of data stored in an array. In GEOS-Chem model program, data on the *Z*-axis usually have less dependency than those on the *X*-axis or the *Y*-axis.

Next, we need to find out the affected scope of the decomposed data calculation. If the decomposed data calculation affects a wide range of data, communication overheads increase dramatically. So, loops with these kinds of data are difficult to be parallelized.

*(4) Parallelizing selected functions* While OpenMP supports directives that can be used for parallelization in a single machine, MPI does not support directives, and developers must change source codes manually using MPI libraries.

We wrote a function (called divide_range_block()) that determines divided ranges from whole data domain for each node, and each node performs data calculation for a part of data specified by this function. After each phase of data calculation, result data are exchanged using MPI libraries. We used MPICH2 for our implementation, and MPI_BCAST and MPI_ALLGATHERV functions are used for data exchange.

Data operations are decomposed and assigned to individual nodes, and MPI library functions are responsible for exchanging data to synchronize result data in all nodes. Programming code 1 shows an example of MPI-applied source codes. The original loop iterates 54 times in GEOS-Chem model program to process the whole data, but the modified loop processes only a small portion of data assigned by the divide_range_block() function.

In addition, when making GEOS-Chem model program run in parallel, we also need to reduce communication complexity to reduce communication overheads caused by MPI library functions.

*(5) Data verification* Data verification is the most important task, since results should remain same regardless of parallel execution or sequential execution. The most frequent error in parallel execution occurs due to incorrect calculation of data offsets. If a node calculates an incorrect set of data, and final results become incorrect.

For data verification, we used 24-hour simulation data of GEOS-Chem model program, and compared results of sequential computation with those of parallel computation.

## Experiments and Results

*Experimental environments*

We constructed 8-node Linux cluster for our experiments. Each machine has two Pentium D 3.2 GHz processors, 1 GB RAM, and connected through 1 Gbps Ethernet network. Intel FORTRAN compiler, ifort (version 10.1 with OpenMP version 2.5), was installed on the machine, and MPICH2 library (version 1.0.6p1) is also installed. The GEOS-Chem driven by GEOS-4 meteorological fields is installed on one machine, and the other machines shared original copy through NFS (Network File System). The program runs in parallel on shared NFS directories of each machine.

We implemented two experimental codes to compare the MPI_BCAST function and the MPI_ALLGATHERV function. Both functions are used to exchange data among

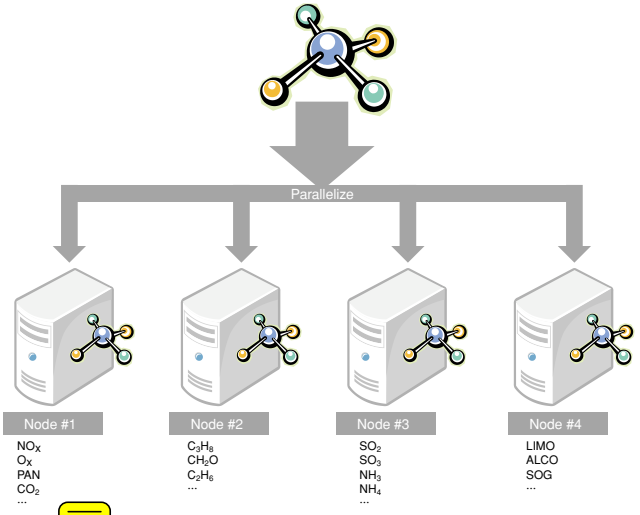


*Fig. 2. Parallelization concept of TPCORE_FVDAS subroutine.*

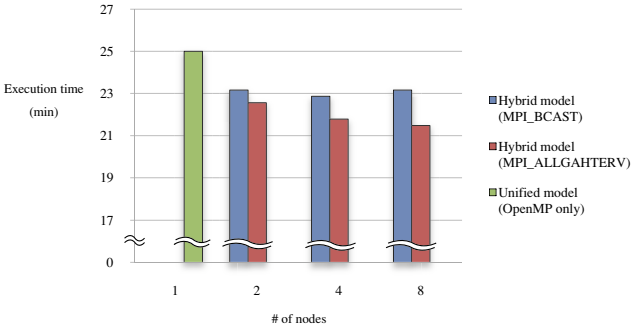


*Fig. 3. Experimental results of the GEOS-Chem model program (24-hr simulation time).*

nodes. The MPI_BCAST function supposes to broadcast a message (i.e., data) from one node to all other nodes in the cluster. However, actual implementation of the MPI_BCAST function does not use real broadcasting transmission. Instead, the function establishes unicast connection to each destination node. So, it is slower than the real broadcasting method. MPI_ALLGATHERV function gathers data from all nodes, and the function delivers data to all nodes again. Unlike the MPI_BCAST function, the MPI_ALLGATHERV function is called once for whole data exchange in a phase.

We have parallelized a part of GEOS-Chem model program. We modified two subroutines, TPCORE_FVDAS and QMAP. The first consumes about 0.6% of whole simulation time, and the latter consumes about 6.3%. Our selected subroutines consume about 7% of total execution time of GEOS-Chem model program. TPCORE_FVDAS subroutine and its QMAP subroutine are selected because they have high average computation time and their data usage patterns are suitable for parallelization.

Figure 2 shows a parallelization concept of the TPCORE_FVDAS subroutine. We analyzed four indexes of the data which the TPCORE_FVDAS subroutine handles with. The outer index of quadruple arrays in GEOS-Chem model represents a kind of chemical substances. In the TPCORE_FVDAS subroutine, there is no interference among computations of these chemical substances, so we divided these computations into several nodes by allocating a specific range of outer index to each node.

*Experimental results*

The original GEOS-Chem model is a unified model with the OpenMP method, and in our experiments, maximum resource is up to two processors in a single machine. Our MPI-applied hybrid model can utilize all 8 nodes.

Figure 3 shows the results of 24-hour simulation data. The original OpenMP-unified model takes about 25 minutes. As expected, MPI_ALLGATHERV function is more efficient than MPI_BCAST function, because data exchanging time for MPI_ALLGATHERV function increases more slowly than that of the MPI_BCAST, as the number of nodes increases. So, the most efficient execution (Hybrid model using the MPI_ALLGATHERV function on 8-node cluster) takes about 21 minutes and 29 seconds with about 3-minute improvement compared with the original execution time. This means that we achieved about 10% performance improvement for the GEOS-Chem model program.

## Limitations and Future Directions

Only small portion of functions in GEOS-Chem model program are possible for parallelization because of complex data dependencies. Recently there are some activities to make data parallel-friendly, such as Earth System Modeling Framework (ESMF) (Yantosca, 2009).

In addition, MPI is different from OpenMP, as mentioned earlier. If each node has multiple processors, MPI-only approaches cannot fully take advantages of overall computing resources. So, we need to make a balanced environment with both OpenMP and MPI methods.

## Related Work

Our efforts of parallelization of GEOS-Chem model program are not the first one. NASA Jet Propulsion Laboratory (JPL) has studied GEOS-Chem model to improve its performance with distributed computing. Kevin Bowman at JPL had experiments about distributed computing of GEOS-Chem model in clustering environments (Bowman, 2005). His report addressed that about 60% of

GEOS-Chem model program are chemistry-related modules and he applied MPI techniques to those modules in master-slave architecture. A master node distributes data to all slave nodes and retrieves processed data from the slave nodes. Bowman estimated overall computing costs, but his estimation did not consider communication and contention costs. Since communication overheads caused between a master and multiple slaves occur in every phase, overall communication overheads are much larger than performance improvement gained from distributed computing. However, in our approach, we applied peer-to-peer communications instead of mater-slave communications; therefore, we could get overall performance improvement even though we parallelized only two functions.

## Conclusions

GEOS-Chem model is a large-scale legacy program developed for a couple of decades. Since the OpenMP method is used in GEOS-Chem, recent top-cluster gives almost no performance improvement compared to a single machine. In this paper, we proposed a general parallelization procedure that can be used to parallelize a large-scale legacy program by application of MPI method. Our test results of MPI-applied GEOS-Chem model program show that we can get good performance improvement by parallelizing a small portion of the program. More specifically, we parallelized only two functions and got about 10% performance improvement with 24-hour data simulation.

Performance improvement using OpenMP-only approach can have bottleneck problems because a certain system resource is shared by processes in multiple cores. One of advantages using our hybrid approach relative to OpenMP-only approach is to have higher scalability as resources increase, depending on the ratio of communication to computation time. However, new technologies such as fiber optics, storage area network (SAN), infinite band, etc., are fast developing to minimize the communication overhead. We argue that our proposed parallelization procedure and the hybrid GEOS-Chem model program can contribute to atmospheric research communities.

**Acknowledgments**—We thank researchers and developers who have contributed to GEOS-Chem model developments. This work was supported by the IT R&D program of MKE/IITA (2008-F-034-02, Development of Security-Quality Gurantee Technology in Resilient Networks).

## References

Amdahl, G. (1967) The validity of the single processor approach to achieving large scale computing capabilities. *1967 AFIPS Spring Joint Computer Conference (AFIPS '67)*, Atlantic City, New Jersey, 483–485.

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S. T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A. and Weihl, W. E. (1997) Continuous profiling: where have all the cycles gone? *ACM Transaction on Computer Systems J*. **15**, 357–390.

Bowman, K. (2005) Parallelization of GEOS-Chem on a 1024-node Linux cluster based on MPI, *2nd GEOS-Chem Users' Meeting*, Cambridge, Massachusetts.

Graham, S. L., Kessler, P. B. and McKusick, M. K. (1982) Gprof: a call graph execution profiler. *SIGPLAN Notices* **39**, 49–57.

Harvard Atmospheric Chemistry Modeling Group (year) The GEOS-Chem model, available at http://acmg.seas.harvard.edu/geos/index.html

Jarvis, S. A., Spooner, D. P., Lim, H. N., Keung, C., Cao, J., Saini, S. and Nudd, G. R. (2006) Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems J*. **22**, 745–754.

Moseley, T., Connors, D. A., Grunwald, D. and Peri, R. (2007) Identifying potential parallelism via loop-centric profiling. *4th International Conference on Computing Frontiers*, Ischia, Italy, 143–152.

MPICH2 (year) available at http://www.mcs.anl.gov/research/projects/mpich2/

OpenMP (year) The OpenMP API specification for parallel programming, available at http://www.openmp.org/

Xu, G., Lu, F., Yu, H. and Xu, Z. (2007) A distributed parallel computing environment for bioinformatics problems. *6th International Conference on Grid and Cooperative Computing (GCC 2007)*, Urumchi, Xinjiang, China, 593–599.

Yantosca, B. (2009) GEOS-Chem code: new developments, future directions, *4th GEOS-Chem Scientific and Users' Meeting*, Cambridge, Massachusetts.

## Appendix

*Programming code*

*Data exchanging code using MPI_ALLGATHERV*

```
call comm_tstart
    agv_displs(1) = 0
    do i=0,comm_size-1
        call comm_qmap_block(ixj1, ixj2, nq, i)
        if (rank == i) then
            q2(:,:,:,1:(ixj2-ixj1+1)) = q(:,:,:,ixj1:ixj2)
        endif
        mpi_tsize = im*jlast*km*(ixj2-ixj1+1)
        agv_rcount(i+1) = mpi_tsize
        if (i+1 <= comm_size) then
            agv_displs(i+1) = im*jlast*km*(ixj1-1)
        endif
    enddo
    call mpi_allgatherv(q2,agv_rcount(rank+1),MPI_REAL8,&
                        q, agv_rcount, agv_displs, MPI_REAL8, &
                                        MPI_COMM_WORLD, ierr)
call comm_tend
```

*Profiling Results of GEOS-Chem model program*

| % time | cumulative seconds | self seconds | calls | name |
|---|---|---|---|---|
| 6.82 | 228.09 | 228.09 | 12348 | fvdas_convect_mod_mp_convtran_ |
| 6.67 | 451.28 | 223.19 | 119232 | tpcore_fvdas_mod_mp_map1_ppm_ |
| 6.43 | 666.29 | 215.01 | 555699 | subfun_ |
| 6.28 | 876.25 | 209.96 | 48 | tpcore_fvdas_mod_mp_qmap_ |
| 6.23 | 1084.69 | 208.44 | 197521 | pderiv_ |
| 5.37 | 1264.27 | 179.58 | 197521 | decomp_ |
| 5.27 | 1440.51 | 176.24 | 119232 | tpcore_fvdas_mod_mp_ppm2m_ |
| 4.86 | 1603 | 162.49 | 3372840 | tpcore_fvdas_mod_mp_xmist_ |
| 4.64 | 1758.14 | 155.14 | 43872 | smvgear_ |
| 4.63 | 1912.91 | 154.77 | 117547140 | gen_ |
| 4.05 | 2048.49 | 135.58 | 2954880 | tpcore_fvdas_mod_mp_fxppm_ |
| 3.89 | 2178.75 | 130.26 | 315168 | blkslv_ |
| 3.44 | 2293.92 | 115.17 | 511827 | backsub_ |
| 3.24 | 2402.45 | 108.53 | 315168 | opmie_ |
| 2.44 | 2483.98 | 81.53 | 13248 | fvdas_convect_mod_mp_hack_conv_ |
| 1.7 | 2540.93 | 56.95 | 96 | wetscav_mod_mp_wetdep_ |
| 1.64 | 2595.92 | 54.99 | 119232 | tpcore_fvdas_mod_mp_steepz_ |
| 1.62 | 2650.22 | 54.3 | 48 | transport_mod_mp_do_global_transport_ |
| 1.49 | 2700.06 | 49.84 | 117547140 | matin4_ |
| 1.44 | 2748.07 | 48.01 | 6969600 | tpcore_fvdas_mod_mp_xtp_ |
| 1.17 | 2787.1 | 39.03 | 77760 | tpcore_fvdas_mod_mp_fyppm_ |
| 1.16 | 2826.02 | 38.92 | 77760 | tpcore_fvdas_mod_mp_ymist_ |
| 0.89 | 2855.72 | 29.7 | 48 | diag1_ |
| 0.84 | 2883.96 | 28.24 | | log.J |
| 0.78 | 2909.99 | 26.03 | 144 | dao_mod_mp_convert_units_ |
| 0.77 | 2935.66 | 25.66 | 79200 | tpcore_fvdas_mod_mp_tp2d_ |
| 0.66 | 2957.86 | 22.2 | 48 | tpcore_fvdas_mod_mp_tpcore_fvdas_ |

*Loop profiling results of TPCORE_FVDAS subroutine*

| Loop | Clocks | Comments |
|---|---|---|
| Loop 1 | 1 | |
| Loop 2 | 40 | |
| Call AIR_MASS_FLUX | 543 | |
| Loop 3 | 55148 | Candidate for parallelization |
| Call QMAP | 60813 | Candidate for parallelization |
| Loop 4 and 5 | — | Not used |

*Execution Time of TPCORE_FVDAS and QMAP subroutines (unit: seconds)*

| Experiments | Subroutine execution time | | Data exchange time | Total execution time |
|---|---|---|---|---|
| | TPCORE_FVDAS | QMAP | | |
| Original OpenMP unified | 1.5434 | 4.1948 | — | 5.7392 |
| BR 2 Hybrid | 0.7728 | 2.0976 | 0.8194 | 3.6898 |
| AG 2 Hybrid | 0.7651 | 2.0937 | 0.7229 | 3.5817 |
| BR 4 Hybrid | 0.4024 | 1.0861 | 1.8877 | 3.3762 |
| AG 4 Hybrid | 0.3987 | 1.0829 | 0.9972 | 2.4788 |
| BR 8 Hybrid | 0.2024 | 0.5437 | 2.8665 | 3.6126 |
| AG 8 Hybrid | 0.2256 | 0.5376 | 1.1441 | 1.9073 |

*Notations: BR = MPI_BCAST, AG = MPI_ALLGATHERV, # = the number of nodes.*