JAERI-M
93-230

PARALLELIZATION OF MCNP4 CODE BY USING SIMPLE FORTRAN ALGORITHMS

December 1993

Putranto Ilham YAZID*, Makoto TAKANO
Fumihiro MASUKAWA and Yoshitaka NAITO

日本原子力研究所
Japan Atomic Energy Research Institute

JAERI-Mレポートは，日本原子力研究所が不定期に公刊している研究報告書です。
入手の問合わせは，日本原子力研究所技術情報部情報資料課（〒319－11茨城県那珂郡東海村）あて，お申しこしください。なお，このほかに財団法人原子力弘済会資料センター（〒319-11茨城県那珂郡東海村日本原子力研究所内）で複写による実費頒布をおこなっております。

JAERI-M reports are issued irregularly.
Inquiries about availability of the reports should be addressed to Information Division, Department of Technical Information, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibaraki-ken 319-11, Japan.

編集兼発行　日本原子力研究所
印　　刷　日立高速印刷株式会社

JAERI-M 93-230

Parallelization of MCNP4 Code by Using Simple FORTRAN Algorithms

Putranto Ilham YAZID*, Makoto TAKANO
Fumihiro MASUKAWA and Yoshitaka NAITO

Department of Fuel Cycle Safety Research
Tokai Research Establishment
Japan Atomic Energy Research Institute
Tokai-mura, Naka-gun, Ibaraki-ken

(Received November 5, 1993)

Simple FORTRAN algorithms, that rely only on open, close, read and write statements, together with disk files and some UNIX commands have been applied to parallelization of MCNP4. The code, named MCNPNFS, maintains almost all capabilities of MCNP4 in solving shielding problems. It is able to perform parallel computing on a set of any UNIX workstations connected by a network, regardless of the heterogeneity in hardware system, provided that all processors produce a binary file in the same format. Further, it is confirmed that MCNPNFS can be executed also on Monte-4 vector-parallel computer.

MCNPNFS has been tested intensively by executing 5 photon-neutron benchmark problems, a spent fuel cask problem and 17 sample problems included in the original code package of MCNP4. Three different workstations, connected by a network, have been used to execute MCNPNFS in parallel. By measuring CPU time, the parallel efficiency is determined to be 58% to 99% and 86% in average. On Monte-4, MCNPNFS has been executed using 4 processors concurrently and has achieved the parallel efficiency of 79% in average.

---

* Research Center for Nuclear Techniques, National Atomic Energy Agency, INDONESIA.

Keywords : MCNP4, MCNPNFS, Parallel Processing, Message Passing, Locking, Semaphore, Parallelization, Hybrid Model, Speedup, Parallel Efficiency.

JAERI-M 93-230

単純なFORTRANアルゴリズムによるMCNP4コードの並列化

日本原子力研究所東海研究所燃料サイクル安全工学部
Putranto Ilham YAZID*・高野　誠・増川　史洋・内藤　俶孝

（1993年11月5日受理）

数種のUNIXコマンドとディスクファイルに，OPEN,CLOSE,READ,WRITE文のみを利用した単純な並列処理用FORTRANアルゴリズムによりMCNP4の並列化を行った。MCNPNFSと名付けたこのコードは，MCNP4の遮蔽解析における殆ど全ての機能を引き継いでいる。本コードは，ネットワーク上の複数のUNIXワークステーションで並列処理を行うことができる。ここでワークステーションは異機種でも構わないが，全てのワークステーションは同一形式のバイナリファイルを作成できる必要がある。さらにMCNPNFSはベクトル並列計算機Monte4においても正常に動作した。MCNP4コードパッケージに含まれるサンプルインプット17題，光子／中性子ベンチマーク5題，および使用済燃料キャスク問題を用いてMCNPNFSの性能評価を行った。ネットワーク上の3台の異なるワークステーションを使用してMCNPNFSの並列実行を行った。CPU時間の測定から得た並列化効率は，58～99％であり，平均は86％であった。Monte4においては，4プロセッサで並列に実行された場合，並列化効率は平均79％であった。

（本研究はSTA科学者交換プログラムにより，1992年10月～1993年9月に行われたものである）

東海研究所：〒319－11 茨城県那珂郡東海村白方字白根2－4

＊　インドネシア原子力庁

Contents

目　　次

I. Introduction.

Parallel processing, in which multiple processors are used concurrently to solve a single computing problem, promises a fastest way to performing large scale computations. It enables the scientists to compute and simulate[1]: more realistic global climate modeling, large N-body computation (up to million bodies) in astrophysics research, huge amount of complex linear equations in electromagnetic problems, highly resolved flow over 3D aircraft configuration, etc., in a matter of minutes or hours of CPU time. In the past time, all of those computations needed thousands of hours of CPU time or were even impossible to realize.

For those purposes, a large variety of parallel computer architectures and systems have been developed. They are supported with sophisticated automatic parallelizing compiler, parallel programming languages, parallel libraries and operating systems.

On the other hand, cluster of workstations linked across a high-speed network are becoming more and more interesting to performing parallel processing, owing to their high speed performances and more user friendly environments. For this, parallel programming softwares, such as: Express, Linda, P4, PVM, etc., are available for parallel program developments.

The common thing that one will face in performing parallel processing on the above systems is that one has to learn many new things, such as operating systems, programming languages or environments, in spite of new programming styles. Even though it is not quite a matter, however, this always leads to an opinion that the parallel processing is much more difficult than the sequential one, but also interesting and challenging.

This report describes some simple algorithms, written in FORTRAN, for performing a parallel processing in distributed memory systems, such as workstation networks and multiple instruction, multiple data (MIMD) parallel computers. The algorithms rely entirely on read/write access to disk files in realizing the interprocessor communication/message passing. Whenever all processors executing parallel program are able to access the files, the interprocessor communication can then

be established with the help of only 4 well-known (FORTRAN) statements, i.e., ***open***, ***read***, ***write*** and ***close*** (!).

We also used the FORTRAN runtime library function ***system***, which is always available in every FORTRAN compiler running on UNIX operating system. By using it, automatic loading of parallel program on any processor in a network can be established, simply, by exploiting the UNIX command ***rsh*** (remote shell). The use of other functions and UNIX commands in performing parallel processing is described in Section II of the report.

At a glance, the above approach seems to be awkward, but the recent development of high speed network and hard disk technology has enabled to minimize the access time of disk files to a reasonable rate, which is ideal enough for interprocessor communication.

The main advantage is that the (FORTRAN) parallel program developed by using those algorithms are highly portable. The program logic is simple so that one can easily apply or modify the algorithms to other problems. High speedup can be gained easily as long as the data that are transferred during the interprocessor communications are still in a reasonable amount, which are also likely to occur in the parallel processing using the "conventional" methods. Also, the algorithms make the parallel programming easier and more understandable. Nothing is hidden from or transparent to the programmers, since most of them can be written in a style that conforms to the FORTRAN 77 standard.

In order to prove the ability of such simple algorithms in performing parallel processing, they have been applied to the MCNP version 4[2-4], a general purpose continuous energy Monte Carlo neutron and photon transport code, which is developed by Los Alamos National Laboratory.

There are several reasons in choosing the MCNP4 as our working code for parallelization:

1) Recently, this code is widely used all over the world, especially in the reactor engineering field, for shielding and criticality calculations, material activations, health physics problems, etc. High utilization of the code makes this work highly contribute.

2) The code is written in a well-structured manner and well documented. This enables us easily to understand the program flow from the very first time and in turn to decide where modifications should or could be performed in parallelizing the code.
3) During the execution of the program, MCNP4 dynamically updates some parameters that are needed in the random walk calculations, i.e., the Russian roulette criteria. Also, some arrays needed for tallying purposes will be automatically updated. Depending on the problem input, some output will be periodically printed. These mean that in parallelized MCNP all processors, in the moment, should be synchronized and temporary calculation results should be transferred from one processor to another. It is a challenging case for our simple algorithms in handling a large amount of data transfer, which occur very frequently, especially in the initial times of program execution.
4) MCNP code has been parallelized by various ways[5-9]. It is interesting to see "how bad" or "how good" our method is to be compared with the others in parallelizing the MCNP code.

During the parallelization of the MCNP4, we adopted a philosophy that the parallelized version should maintain, as far as possible, all capabilities of the original one. This means:

- All modifications should not alter the algorithm of the sequential version.
- Therefore, the calculating results of the modified version should be exactly the same as the original version.
- Alterations, if any, should be given as a choice, but not as an obligation.

However, our parallelized version of MCNP4 does have some capabilities of the sequential MCNP4 that are not supported any more, as follows:

1) KCODE problems, i.e., the criticality calculation problems are not accepted any longer. This is due to that the criticality calculations have radically different program flow than the shielding calculations. In the first cycle, M source particles, which are easy to be

distributed among the processors in the network, are generated and their histories are tracked. Each of the particles will create none, one or more particles that will be tracked in the next cycles. These newborn particles will be stored in the file SRCTP. In the parallelized MCNP, due to inappropriate data format of SRCTP file, it is impossible to combine all the files, generated by each processor, into one file that exactly has the same sequence as one resulted by the sequential code. Furthermore, the reading of SRCTP file, which should be performed in the subsequent cycles, is impossible or, at least, can only be performed in an inefficient manner. These all will make the parallel processing inefficient, due to the large interprocessor communication time during the data transfer.

2) TTY interrupts as well as plotting capabilities are deactivated. It is understandable, because using our simple method it is impossible to perform a safe and convenient I/O process through the terminal. However, this limitation is also commonly faced in performing parallel processing using other methods.
3) Time interrupt (as in automatic termination of program execution owing to the very long history tracking) and any input cards that have "time-relation" values (such as CTME card and negative number of particles in NPS card) become unavailable. These limitations are also understandable, since every processor will have their own (CPU) time reference, which are impossible to be synchronized.
4) The SSR problems, i.e., the problems that need a surface-source file, can not be properly performed. Up to the time this report is written, we are not able yet to handle the reading of surface-source file RSSA in a proper and efficient manner.
5) Event log, lost of particles diagnostic printing, debug print and some warning messages become lost. This is because of that they are performed during the time, where all processors will print out their output to their own (temporary) files. These files will automatically be deleted whenever the MCNP terminates its execution. However, this is not

really a big problem, because those capabilities are rarely used or even never be used in all practical cases.

6) No extra dump will be produced (in the case of SSW problem), when the execution terminates due to "bad trouble." This is because of that the data are partly incomplete (i.e., which are produced by the processor that detected "bad trouble"), so that it is useless to dump.

In spite of that incapability, the parallelized MCNP4 (named MCNPNFS) has two new execution-line-message options:

* **S**, which denotes the sequential mode. Executing the parallelized MCNP with this option, such as in: MCNP S, will give the users a chance to executing the MCNP sequentially, i.e., the original MCNP.
* With the new option **H**, users can enjoy the capability of MCNPNFS running on a heterogeneous system. In this report, heterogeneous system is defined as a collection of several distributed memory computer systems, such as workstation networks or multitasking parallel computers (such as Monte4), which have different hardware architecture, but running under the same operating system, UNIX. The system is also characterized by which the parallelized program should be compiled individually, using different compiler, on each processor or on some of them.

Another new feature that is offered by MCNPNFS is that the users can alter the default values of the first and periodical updates of the Russian roulette criteria. This feature can be extremely useful for getting high parallel efficiency (means also high speedup) since the frequency of "unnecessary" synchronization of the processors will be drastically reduced. However, it will cause the output of MCNPNFS differs from the original code. It will happen whenever the code is executed for solving the problems involving point detectors and DXTRAN spheres.

In spite of those new features, all capabilities of MCNPNFS remain the same as original one, except those previously mentioned. This means that users can order the code to periodically print out the output to OUTP and MCTAL files, dump the results on RUNTPE file, create the surface-source file WSSA, execute the MCNPNFS in continue run, etc. All binary files (RUNTPE and WSSA) generated by MCNPNFS have exactly the same data format as the original ones, so that they can be used by the original MCNP4, if needed.

In parallelizing the MCNP4 we have chosen a *hybrid* model parallel program, i.e., a combination of *host-node* and *hostless* parallel program. By adopting this model we can exploit all processors as efficient as possible. Also, we can program the MCNPNFS with less effort. Section III will describe in detail our parallelization strategies.

Some requirements should be fulfilled before executing the MCNPNFS. First, all processors should run under UNIX operating system. Second, all processors should have read/write access permission to the files that are used as interprocessor communication tools and input/output files. It means that the files should be the shared files. Under UNIX operating system, shared files, i.e., the files that are able to be accessed by all processors in the network, are commonly established by using the NFS (Network File System). In executing MCNPNFS on a heterogeneous system, users should also be sure that all processors will read/write all binary files in exactly the same way. This means that the processors should have identical bit pattern in their external data representation, but not necessarily in their internal data representation. Appendix 3 will describe in detail how to handle this problem.

The MCNPNFS has been intensively tested by executing 17 out of the 25 sample problems supplied in the original code package, four neutron- and photon benchmark problems[10,11] and a spent fuel cask problem. From the executions we found that MCNPNFS will always produce exactly the same calculating results as the original one, except those deviations mentioned previously. Three SUN workstations of various models linked by Ethernet have been used to execute MCNPNFS concurrently. Based on the CPU time of the fastest workstation in our

network, the speedup is found to vary from 1.22 to 2.13 and average value of 1.83. It is corresponding to the parallel efficiency of 57.6% to 98.6% and average value of 85.7%.

We also executed the MCNPNFS on JAERI Monte4 multitasking parallel computer. By using all 4 processors available in this computer concurrently, the MCNPNFS showed a remarkably high speedup that ranges from 1.83 to 3.62 and average value of 3.17, meaning, the parallel efficiency of 45.7% to 90.5% and average value of 79.2%. Section IV of this report will describe in detail the performance measurements and their results.

II. Simple FORTRAN Algorithms for Parallel Processing on Distributed Memory Systems :

This section does not intend to give detail explanations of the theory of parallel processing, since it is not our expertise. However, here will be described somewhat in detail about the very simple way to perform parallel processing in a distributed memory system by exploiting disk files as the tools for message passing. To simplify the discussions and to avoid mistakenly citations, we do not try to make any comparison with other methods on what or how they do parallel processing. Readers can consult the Refs. (12) - (15) for further topics on parallel processing.

II.1 Message Passing :

In a distributed memory system, all processors process their own data at their own pace independently of each other. The key of successful parallel processing in that system is how to manage all processors in a proper manner, so that they can cooperate with each other in solving a single computational problem. In other words, we have to find out a way how the processors can communicate with each other, so that they can process different data in a given time, but when needed, process the same data in the subsequent time.

During the interprocessor communication, data are transferred from one to another. The process of sending data from one processor to another is called *message passing*. This can be done by copying one processor's memory, then, send them through communication network and copy them onto other processor's memory. For that, special library routines are needed.

Another approach is by using the disk files as media for sending and receiving messages. One processor sends its message by simply writing it to a given file. The other processors that want to receive the message can then read the file. In FORTRAN, reading from and writing to a file can readily be performed by using ***read*** and ***write*** statements. The file, called *message file*, may have arbitrary file type and data format. However, the sender and receiver should agree to each other, meaning that they have to use exactly the same file type and data format.

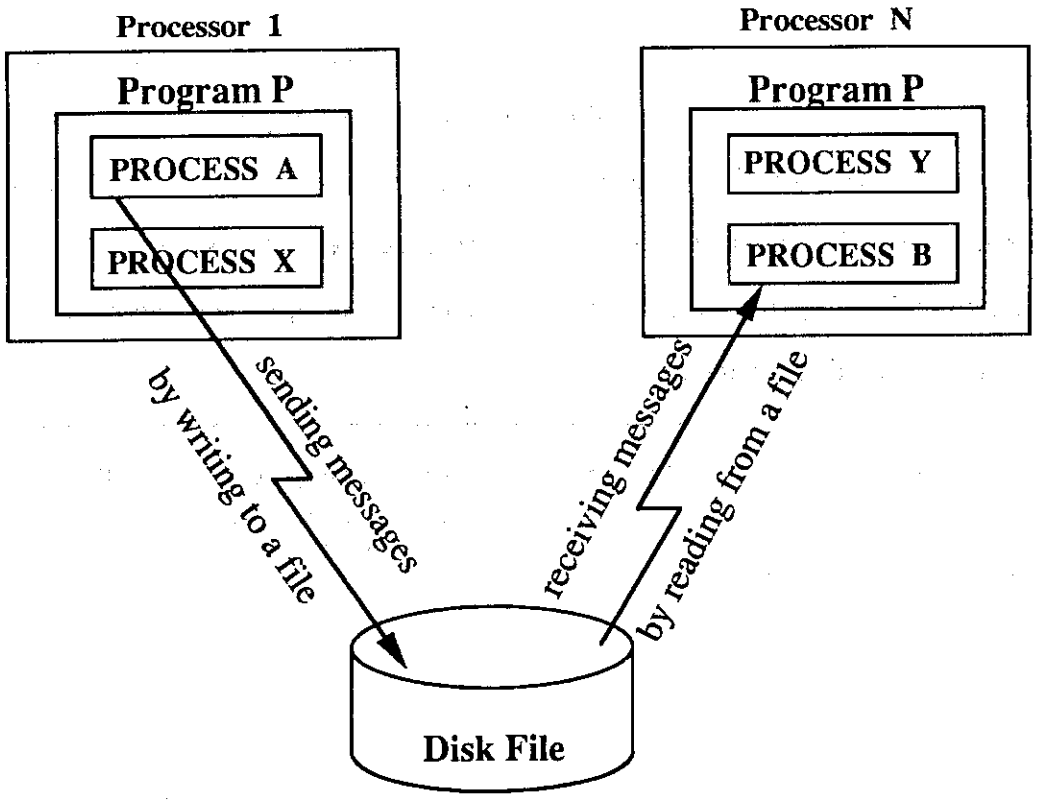

Figure 2.1 : Message passing between two processors during the execution of parallel program P. The disk file must be accessible to both processors.

Under UNIX operating system, multiple processors may share their file system with other processors in the network. It is commonly done by using Network File System (NFS). One processor that serves as a file server may export its file system to the other processors, that have to mount it on their own file system. Then, any files that reside on the file system can be used as message files. One should be sure that all processors must be able to access the files. This means that the file server must also grant read/write permission on the file system to all the processors.

II.2 Flag files:

Due to their independence and the nature of computational problem, any processor may send or receive their messages any time during the execution of parallel program. It may happen that the receiver, i.e., the processor that want to receive a message from the other, has to wait for the arrival of the message, because the sender has not sent it yet.

Again, disk files can be exploited for this purpose. The trick is that the receiving processor should, at first, check the existence of a given file, before it tries to read the contents of a message file. Upon a successful detection of the file, which is called *flag file*, the processor may then continue to read the message file. Otherwise, it has to suspend the reading of the message file, namely by retrying the checking action as long as the flag file does not exist yet.

This can be realized simply by exploiting the capability of ***open*** statement. The following FORTRAN statement will show the trick.

```
10      open(99,file='flag_file',status='old',err=10)
```

The specifier *status* above requires that the file flag_file must exist before the ***open*** statement takes into action, otherwise, an error will rise. Whenever error does occur, the specifier *err* will avoid immediate termination of the program and will redirect the program flow to the statement number 10, instead. Clearly, the statement above represents a loop as long as the file flag_file does exist.

Similarly, the nonexistence of the flag_file can also be used for checking purpose, before any message reading process is performed.

```
10      open(99,file='flag_file',status='new',err=10)
```

One should be aware that the above statement will automatically create the file flag_file, when it does not exit before. This will cause a problem when more than one processors want to read the same file. Those processors will be suspended, unless the flag_file is deleted by the first processor. Later, it will be shown that this feature plays a key role in realizing a dynamically parallel processing.

II.3 Application of the Algorithms to a Simple Parallel Program :

The algorithms explained so far are sufficient for us to write a complete, however, simple parallel program, as shown follows.

Suppose, processor A sends a message to processor B. Upon receiving the message, processor B will print out the message on terminal and send back another message to processor A, which in turn will also display it on the terminal. Notice that the program must be executed concurrently on two processors only.

```
      program para00                                                    000001
        character*80  message                                           000002
        open(99,file='flag_file',status='new',err=30)                   000003
c---this part will only be executed by processor A---                   000004
        close(99)                                                       000005
        open(99,file='message_A')                                       000006
        write(99,'(a)') 'This message is sent by' //                    000007
     1                  'processor A'
        close(99)                                                       000008
10      open(99,file='message_B',status='old',err=10)                   000009
        read(99,'(a)',err=20) message                                   000010
        close(99,status='delete')                                       000011
        print *, message                                                000012
        stop                                                            000013
20      close(99)                                                       000014
        goto 10                                                         000015
c--this part is executed on processor B---                              000016
30      open(99,file='flag_file')                                       000017
        close(99,status='delete')                                       000018
        open(99,file='message_A')                                       000019
        read(99,'(a)') message                                          000020
        close(99,status='delete')                                       000021
        print *, message                                                000022
        open(99,file='message_B')                                       000023
        write(99,'(a)') 'This message is received ' //                  000024
     1                  'from processor B'
        close(99)                                                       000025
        stop                                                            000026
      end                                                               000027
```

Although, the parallelism is hardly to see in the above program, it shows many tricks how to perform message passing between two processors. Later we can see that the program is indeed parallel, in which one processor is waiting for the message, while the other is sending it.

The statement line 3 functions not only to check the existence of the flag file, but it also decides the role of the executing processors. The first processor will be assigned as processor A, because the file flag_file does not exist yet. The second processor, that executed later on will have the role of processor B, because it will discover the file flag_file, which is created automatically by processor A. This kind of programming style is called *hostless model*, where any processor will decide its own role by itself. In this model, there is only one parallel program executed concurrently on multiple processor. The statement line 9 is used to check the existence of the flag file that is also a message file (i.e., file message_B). This algorithm is worthily, by which users can avoid creating too many files. However, the algorithm shown in the statement line 10 should be applied to reading such a file. Users should realize that at the time the processor B executes the statement line 23, the file message_B is immediately created, but its content remains nil, unless the statement line 25 is executed (!). The same algorithm may also be applied to the statement line 19, however, it is unlikely needed in the above simple program.

The program para00 also shows us how to manage the files that are not any longer needed. They are deleted simply by using the ***close*** statements together with the specifier *status='delete'*. This algorithm will help programmers avoiding an excessive use of unit numbers. The immediate closing of a unit number and immediate using of the same unit number for connecting to other file are the key of successful parallel programming using those simple algorithms. This guarantees that the data will be flushed into the physical disk files (!).

After compiling the program, it can be executed as follows. Assuming that the user is using a workstation under a Window system, such as: XWindow, XView, SunView, OpenWindows, etc. It is also assumed that the workstation is sharing its file system with other workstations in the network by using NFS. The user must develop as well as execute the program on the shared file system. The user should then create 2 command windows, and login on the other workstation from one of the windows. Then, make sure that the two windows are on the same working directory, namely the directory where the program para00 resides on. Execute the program from

one of the windows (by typing: para00) and then change to the other one to execute the same program. The output "This message is sent by processor A" and "This message is received from processor B" will show up on the separate windows. Notice that nothing is displayed on the windows, before the second invocation of para00 (from the second window) is performed.

II.4 Automatic Loading of Parallel Program :

It seems so far that the execution of a parallel program is inconvenient and impractical. The users have to prepare the command windows, as many as the number of processors that they want to use in the parallel processing (!). This problem arises mainly due to the lack of automatic loading of parallel program, so that one has to invoke it from different processors manually. Unfortunately, the FORTRAN 77 standard does not specify any subroutines or functions used for parallel processing. However, almost all recent FORTRAN compilers running under UNIX operating system (hereafter, they will be called simply as UNIX FORTRAN compilers) such as: Sun FORTRAN[17], Convex FORTRAN[18], FORTRAN 77/SX[19], etc., provide the users with a powerful function ***system***. By using it, one can invoke (almost) any UNIX commands from within a FORTRAN program. For instance, the statements:

```
j  =  system('hostname')
j  =  system('echo  $HOME')
```

will display the hostname of the processor and its home directory, respectively.

The following program will demonstrate how to load a parallel program on any processor available in the network, automatically. It exploits the UNIX command ***rsh*** together with the function ***system***. Some other functions and subroutine (***iargc***, ***hostnm*** and ***getarg***), which are also provided by UNIX FORTRAN compilers, are used to distinguish the executing processors.

```
      program para01                                          000001
        character*16  cpu_name                                000002
        integer  hostnm                                       000003
```

```
          i = iargc()                                                    000004
          if (i .ne. 0) then                                             000005
              do 10 j = 1, i                                             000006
                call  getarg(j,cpu_name)                                 000007
                k = system('rsh ' // cpu_name //                         000008
     1                     ' /pathname/para01 &')
10              continue                                                 000009
              else                                                       000010
                i = hostnm(cpu_name)                                     000011
                print *,  'I am processor ', cpu_name                    000012
              endif                                                      000013
              stop                                                       000014
            end                                                          000015
```

The executable program para01 can be invoked, for example, as follows:

para01 cpu_1 cpu_2 cpu_3 cpu_N

Where : cpu_1, cpu_2, cpu_3 and cpu_N are arguments that denote the hostnames of remote processors. The program para01 will then be executed concurrently on 4 processors and the output will be displayed on the same window (!). The program may also be invoked without any argument, such as: para01. However, it will be executed only by the *host* processor, the one is currently being used.

In the program, number of arguments given at the invocation of the program para01 is determined by the statement line 4. This number is then used to decide the role of the respective processors. The automatic loading of parallel program para01 is performed by the statement line 8, which orders the *remote* processor "cpu_name" by using the UNIX command ***rsh*** to execute the program para01. Notice that the host processor loads itself on the remote processors (!). The statement line 8 also guarantees that the remote processors will always detect zero argument, so that it is only the host processors that can load the program para01 on the remote processors. The ampersand (&) is very important, because it will signal the operating system that the ***rsh*** command should be executed in background. Without it, the program flow will continue to the next statements only if the program para01 has been completely executed by the remote processor, which means that the parallelism can not be achieved (!!). In the statement line 8, the program

para01 should be invoked by specifying its full pathname (denoted by "/pathname"), because ***rsh*** command will always put the user on the home directory of the remote processor, which normally differs from the user's current working directory.

II.5 Locking Algorithms for Dynamic Load Balance Parallel Processing:

In the Section II.3, it is shown that during the execution of parallel program, the processors communicate with each other through different disk files (i.e., message_A and message_B). However, it is also possible that the executing processors use a single disk file as a medium for sending and receiving their messages. It is not only practical to use such a single shared file for interprocessor communication, but it is also a natural way, whenever the processors receive and send messages of the same type (but do not mean the same "value") from and to other processors. Suppose a processor receives a message by reading a given file that contains only one integer data. Upon receiving the data, it increases the data by one and updates the content of the message file by writing the new data on the file. This processor will then continue its own task, processing the data that it has gotten. In the subsequent time, another processor will read the message file and, again, update its content in the same way as the first processor did. Then, this processor will also process the data. The above process sequences are repeated until the content of the message file reaches a certain limit value.

One thing is hidden in the above process, namely how we deal with the situation in which two or more processors try to read or write the message file simultaneously. In a multitasking system, such as UNIX, two or more processes executed by one or more processors may access the same file at the same time. Simultaneous read access to a file during the execution of parallel program will cause the processors to get exactly the same data. Of course, it will lead to an erroneous final calculating results, if the parallel program is intended to use multiple processor for processing different data. The situation is even worse when the processors try to write their messages to the file simultaneously. The final status, i.e., the final content of the file is unpredictable because all data will be written one over another.

The algorithms that we may use to carry out the above problems are very similar to the ones already described in Section II.2, namely, by exploiting the flag file. Prior to reading a message file, any processor has to check the existence of a given flag file. As the file does not exist, the processor will create it immediately and it can then read the message, update the received data and store it again to the message file. At last, this processor should never forget to delete the file, otherwise, the other processors that are suspending their actions will be blocked forever (!).

The above algorithm is called *locking*, whose use guarantees that only one processor can perform some actions or use some resources at a time, while the others will be suspended to carry out the same things.

The following parallel program simulates a Monte Carlo code solving the random walk calculations. During the execution of the program, all processors will use only one message file as a tool in the interprocessor communication. This file, namely batch_file, contains one integer data that represents number of particles and an integer data that symbolizes a seed used for generating the random numbers. Once a processor succeeds to read the batch_file, it increases the number of particles by one and stores it again together with a new seed. The process will be repeated by different processors, until the number of particles reaches a certain value. Through this way, any executing processor will always process different data.

```
      program para02                                                    000001
        character*80  path_name,batch_file,flag_file,                   000002
     1                node_str*16,cpu_name(4)*16
        data  cpu_name  / 'cpu_1','cpu_2','cpu_3','cpu_4' /             000003
        path_name = '/pathname/'                                        000004
        i_p = lnblnk(path_name)                                         000005
        batch_file = path_name(:i_p) // 'batch_file'                    000006
        i = iargc()                                                     000007
        if (i .lt. 2) then                                              000008
          call  getarg(1,node_str)                                      000009
          read(node_str,'(i10)',err=999) nodes                          000010
c---initialize the batch file---                                        000011
          open(98,file = batch_file,form='unformatted')                 000012
          write(98)  0, 1234567                                         000013
          close(98)                                                     000014
```

```
            if (nodes .gt. 4) nodes = 4                                  000015
c---load the parallel program to all processors---                       000016
            do 10 i = 1, nodes                                           000017
              write(node_str,'(i4)')  i                                  000018
              j = system('rsh ' // cpu_name(i) //                        000019
     1                   path_name(:i_p) // ' para02 '
     2                   // 'dummy_arg ' // node_str // ' &')
10          continue                                                     000020
            stop                                                         000021
          endif                                                          000022
          call  getarg(2,node_str)                                       000023
          read(node_str,'(i4)')  node                                    000024
          flag_file = path_name(:i_p) // 'flag_file'                      000025
c---check the existence of the flag file---                              000026
20        open(99, file = flag_file,status='new',err=20)                 000027
            open(98,file=batch_file,form='unformatted')                  000028
            read(98)  n,nseed0                                           000029
            n = n + 1                                                    000030
            x = rand(nseed0)                                             000031
            nseed = 1.e7 * rand(0)                                       000032
            rewind(98)                                                   000033
            write(98)  n, nseed                                          000034
            close(98)                                                    000035
            close(99,status = 'delete')                                  000036
            if (n .gt. 100) goto 30                                      000037
            call Random_Walk(nseed0, i)                                  000038
            print *, cpu_name(node), n, i                                000039
            call flush(6)                                                000040
          goto 20                                                        000041
30        stop                                                           000042
999       print *, 'Invalid number of processors. Execution ',           000043
     1             'canceled !!!.'
          stop                                                           000044
        end                                                              000045


        subroutine Random_Walk(nseed,iter)
          double precision  d1,d2,d3,d4,d5,d6
          x = rand(nseed)
          iter = 100000 * rand(0)
          d1 = 0.5d0
          d2 = 0.5d0
          d3 = 0.499975d0
          d4 = 2.0d0
          d5 = 0.75d0
          d6 = 0.50025d0
          do 10  i = 1, iter
            d1 = d3 * datan(d4 * dsin(d1) * dcos(d1) /
     1           (dcos(d1+d2) + dcos(d1-d2) - 1.0d0))
            d2 = d3 * datan(d4 * dsin(d2) * dcos(d2) /
     1           (dcos(d1+d2) +dcos(d1-d2) - 1.0d0))
            d5 = dsqrt(dexp(dlog(d5) / d6))
10        continue
```

```
      return
    end
```

The program para02 introduces two new functions that are also provided by UNIX FORTRAN compilers, namely, ***lnblnk*** for getting the length of the content of a string variable and ***rand*** for generating random numbers. The subroutine ***flush*** in the statement line 40 will make sure that the output will be immediately displayed on the terminal, so that users can see directly how the parallel processing is performed. Without it, the output will be stored in the processor's buffer and will be print out on terminal only when the buffer is already full or at the end of program execution. The subroutine Random_Walk is included here for completeness. It serves mainly to reduce the frequency of the moments, where two or more processors access the message file simultaneously, by providing enough calculation time. Also, it simulates the random walk calculations in a Monte Carlo code.

The statement line 19 show how the host processor passes two arguments ("dummy_arg" and "node_str") to the remote processors, while it loads the program para02 (i.e., itself) on them. The second argument ("node_str") will be used by the remote processors to assign their *node number* (node) with the help of statement line 23 and 24. However, the first argument ("dummy_arg") is merely a dummy argument. By using this technique, the remote processors will always detect 2 arguments, meaning that they will certainly skip the statements line 9 to 21.

The locking algorithm is applied to the statement line 27. Any processor that detects the existence of the flag_file will be suspended by performing a loop on the same statement line. Once a processor success to open the flag_file, it will read the batch_file to get the number of particles (n) as well as the seed (nseed), update them and store them again in the file batch_file. Before the processor invokes the subroutine Random_Walk, it deletes the flag_file by executing the statement line 36, so that the other processors will have a chance to read the batch_file.

During the message passing above, an unformatted file type has been exploited. This file type enables us to open a file for both read and write actions at once. It offers also the most efficient use of disk space and great flexibility & fast read/write access.

Due to the randomness in the execution time of the subroutine Random_Walk, the total number of processes that are performed by each processor is different from one to another. This technique of distributing the tasks to the executing processors is called *dynamic load balance*. Every processor is kept busy, by taking a new task as soon as its previous task is already completed, without having to wait for the other processors completing their tasks.

To execute the program para02, user has only to give the number of processor as an argument while invoking the program, such as: para02 3. It will then order the host processor to load the program para02 on three remote processors, namely cpu_1, cpu_2 and cpu_3. The program will also work well whenever one of the remote processor is accidentally also the host processor. Invoking para02 without any argument will simply cause the program terminates, without loading the program para02 on any remote processor.

UNIX FORTRAN compilers also provide us with another runtime library function that can be exploited to realizing the locking algorithm. The integer function ***rename*** that serves to change the name of a file will help the users to develop parallel program in a more efficient and elegant way. From the experience we found that the locking algorithm using the ***rename*** function works superior to the one that uses only the ***open*** and ***close*** statements explained above.

With the ***rename*** function, message passing can be performed without having to create a flag file. Figure 2.2 shows schematically the locking process during the message passing between two processors. Both processors use a disk file A as the place to receive and send the message from one and to another. At a certain moment, processor 1 finds that the file A does exist, so that the rename process will be performed successfully. Now, the file A is renamed to file A.001. Then, the processor 1 begins to read the data contained in the file and sends back the other data simply by writing them to the file A.001. Meanwhile, processor N is also trying to access the file A,

however, the renaming process fails because the file A has already vanished. It will keep on trying as long as the processor 1 has not yet completed its work. The processor 1 will change again the name of file from A.001 to A, when it has completed the receiving and sending of data. Through this way, the processor N may begin to access the file.

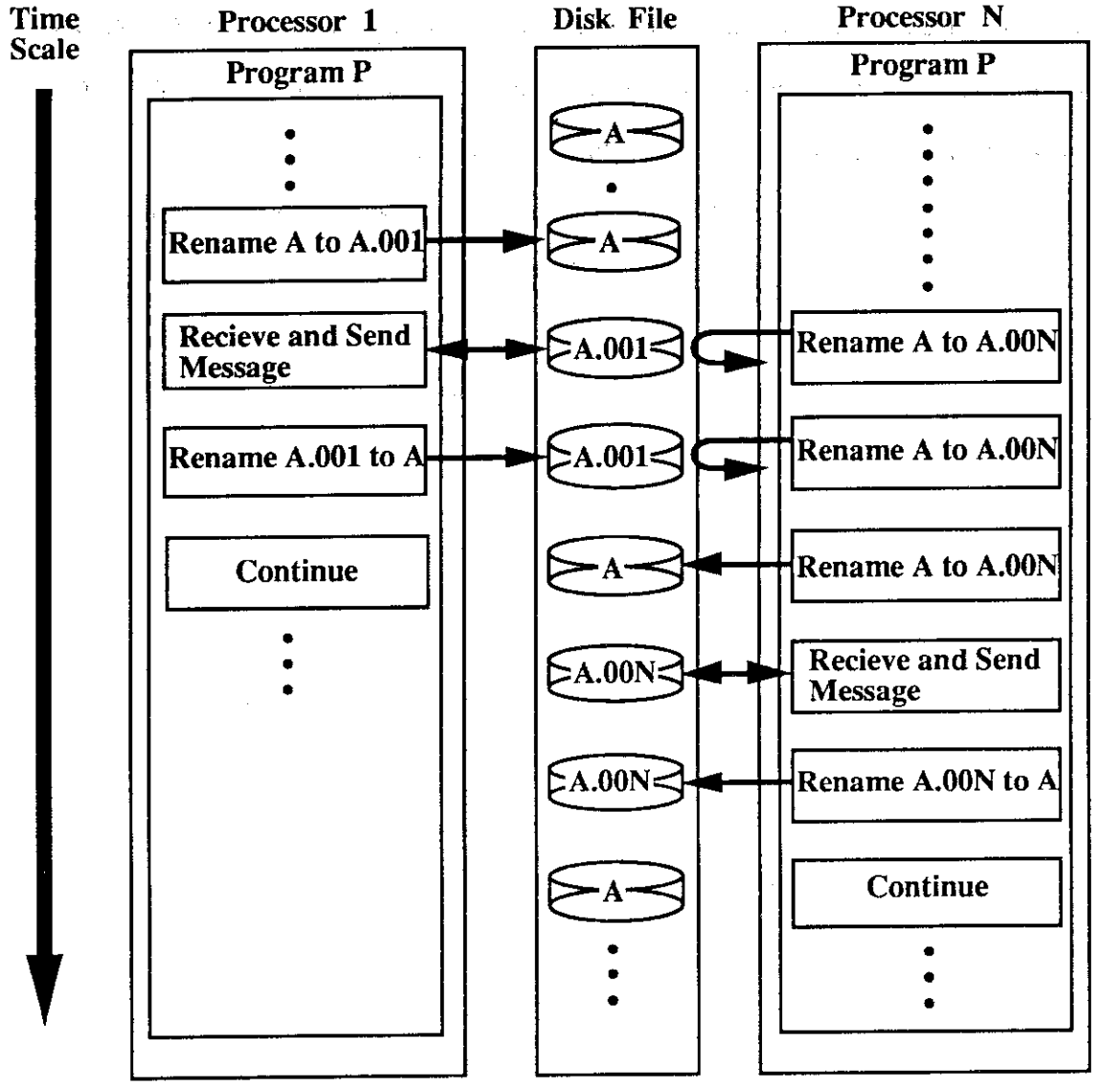

Figure 2.2 : Locking algorithm using the rename function during the message passing between two processors.

The modified version of program para02, below, will show how to realize the locking algorithm by using a ***rename*** function. Further, it demonstrates the algorithms for merging and sorting the contents of output files, which are produced by the executing processors, into one final output file.

```
      program para03                                                    000001
        integer rename                                                  000002
        character*80  path_name,batch_file,dummy,out_file,              000003
     1                node_str*16,cpu_name(4)*16
        data  cpu_name  / 'cpu_1','cpu_2','cpu_3','cpu_4' /             000004
        path_name = '/pathname/'                                        000005
        i_p = lnblnk(path_name)                                         000006
        batch_file = path_name(:i_p) // 'batch_file'                     000007
        i = iargc()                                                     000008
        if (i .lt. 2) then                                              000009
          call  getarg(1,dummy)                                         000010
          read(dummy,'(i10)',err=999) nodes                             000011
c---initialize the batch file---                                        000012
          open(98,file = batch_file,form='unformatted')                 000013
          write(98)  0, 1234567                                         000014
          close(98)                                                     000015
          if (nodes .gt. 4) nodes = 4                                   000016
          write(dummy,'(i6)') nodes                                     000017
c---load the parallel program to all processors---                      000018
          do 10 i = 1, nodes                                            000019
            write(node_str,'(i4)')  i                                   000020
            j=system('rsh '// cpu_name(i) // path_name(:i_p)            000021
     1           // ' para03 '// dummy // node_str // ' &')
10        continue                                                      000022
          stop                                                          000023
        endif                                                           000024
        call  getarg(1,dummy)                                           000025
        read(dummy,'(i4)')  nodes                                       000026
        call  getarg(2,dummy)                                           000027
        read(dummy,'(i4)')  node                                        000028
        write(out_file,'(a,i2.2)')  path_name(:i_p) // '                000029
     1                              out_', node
        open(99,file=out_file)                                          000030
        write(dummy,'(a,i2.2)')  path_name(:i_p) // 'dummy_',           000031
     1                           node
c---rename the batch file to dummy file---                              000032
20      if (rename(batch_file,dummy) .ne. 0)  goto 20                   000033
          open(98,file=dummy,form='unformatted')                        000034
          read(98)  n,nseed0                                            000035
          n = n + 1                                                     000036
          x = rand(nseed0)                                              000037
          nseed = 1.e7 * rand(0)                                        000038
```

```
            rewind(98)                                                    000039
            write(98)  n, nseed                                           000040
            close(98)                                                     000041
            i = rename(dummy,batch_file)                                  000042
            if (n .gt. 100) goto 30                                       000043
            call Random_Walk(nseed0, i)                                   000044
            print *, n, i, cpu_name(node)                                 000045
            call flush(6)                                                 000046
            write(99,*)  n, i , cpu_name(node)                            000047
          goto 20                                                         000048
30        close(99)                                                       000049
          open(99,file=path_name(:i_p) // 'STOP',status='new',             000050
     .....1       err=40)
          call  Merge(nodes,node,path_name,i_p)                           000051
          stop                                                            000052
40        write(dummy,'(a,i2.2)') path_name(:i_p) // 'finish_',           000053
     1         node
          i = rename(out_file,dummy)                                      000054
          stop                                                            000055
999       print *, 'Invalid number of processors. Execution ',            000056
     1           'canceled !!!.'
          stop                                                            000057
        end                                                               000058



        subroutine Merge(nodes,node,path_name,i_p)                        000059
          integer rename                                                  000060
          character*(*)  path_name,out_file*80,finish*80                  000061
          logical skip(4), repeat                                         000062
          do 10  i = 1, nodes                                             000063
10        skip(i) = .false.                                               000064
          skip(node) = .true.                                             000065
          write(out_file,'(a,i2.2)')  path_name(:ip) // 'out_',           000066
     1         node
          repeat = .false.                                                000067
20        do 40  i = 1, nodes                                             000068
            if ( skip(i) )  goto 40                                       000069
              write(finish,'(a,i2.2)')  path_name(:i_p) //                000070
     1                                  'finish_' , i
              open(99,file=finish,status='old',err=30)                    000071
                close(99)                                                 000072
                skip(i) = .true.                                          000073
                j = system('cat  ' // finish // '>>' //                   000074
     1              out_file)
                open(99,file=finish)                                      000075
                close(99,status='delete')                                 000076
                goto 40                                                   000077
30            repeat = .true.                                             000078
40        continue                                                        000079
          if (repeat )  goto 20                                           000080
          write(finish,'(a,i2.2)')  path_name(:i_p) //                    000081
     1                              'batch_file'
```

```
      open(99,file=finish)                                          000082
      close(99,status='delete')                                     000083
      open(99,file=path_name(:i_p) // 'STOP')                       000084
      close(99,status='delete')                                     000085
      j=system('sort -n -o ' // out_file // out_file)               000086
      j=rename(out_file,path_name(:i_p) // 'OUTPUT'                 000087
      return                                                        000088
    end                                                             000089
```

The statement line 2, that defines the ***rename*** function as an integer function, is very important. Ignorance of such a statement will cause an improperly execution of the program para03 (!). The statement line 33 serves to rename the message file batch_file to "dummy_xx". It also uses the return value of the ***rename*** function, in a direct manner, to check whether the rename process is successful or not. If necessary, i.e., whenever the ***rename*** function returns a non zero value, it will loop on the same line.

The program also demonstrates how to manage the temporary calculating results produced by any of the executing processors. They save the results in their own files. The node number (node) will be used by the statement line 29 to determine the filename of respective processor. The algorithm adopted in the statement line 50 is used to determine which processor will merge all the output files. Only one processor will perform this task, namely, the one that takes the least time to complete its last calculations. This processor will then combine all the output file, through the statement line 74, with the help of UNIX command ***cat***. The other UNIX command, ***sort***, is also exploited to sorting the contents of the merging file.

II.6 Semaphore in Host-node Model Parallel Program :

So far, the algorithm that we have been discussed previously deal with only one parallel programming style, namely, the hostless model. The advantage of this model is that the users have only to develop one parallel program, so that it is more compact and easier to maintain. However, the other computing problems will lead naturally to another style of parallel programming, the *host-node model*, in which two different programs are needed to solving a

single computing problem. The first program, called *host* or *master program*, will be executed on the host processor and the second, called *node* or *slave program*, will be loaded and executed on the remote processors. The host program mainly has the function to perform the non-time-critical tasks and "once-only" aspects of the calculation, such as initialization and automatic loading of the node program on the remote processors. It may also act as a coordinator, which supervises and distributes the tasks to the node processors, combines the results and performs the final calculations. On the other hand, the node program will act as a "work horse," which performs the main computing actions.

As a good example, one may take the MCNP4 code. At the initialization phase, a huge file (RUNTPE) will be generated to store cross section data, problem input, geometry data, etc. It will consume an excessive amount of disk space, if all node processors try to generate their own files. Also, a large file will be generated by the code, as the output file (OUTP) during and at the end of an execution. Clearly, the host model is a better choice in parallelizing such a code.

In the following, we will reconstruct the program para03 into two seperate programs, which represent a host-node model parallel program. The host program is extracted from the first part of program para03 and the subroutine Merge, while the node program is taken from the last part, including the subroutines Random_Walk.

During the execution of the program, the interprocessor communications are established only between the host and node processors. No data will be passed from one node processor to another. Each node processor uses its own message file for receiving and sending data from and to the host processor, which is completely separated from other's files.

```
      program host                                                      000001
        logical notyet(4), repeat                                       000002
        character*80  path_name,batch_file(4),dummy,out_file,           000003
     1                node_str*16,cpu_name(4)*16,log_nm(4)*16
        data  cpu_name  / 'cpu_1','cpu_2','cpu_3','cpu_4' /,            000004
     1        log_nm    / 'log_1','log_2','log_3','log_4' /
        path_name = '/pathname/'                                        000005
        i_p = lnblnk(path_name)                                         000006
```

```
      i = iargc()                                                       000007
      if (i .eq. 0) goto 777                                            000008
      call getarg(1,dummy)                                              000009
      read(dummy,'(i10)',err=888) nodes                                 000010
      if (nodes.gt.4 .or. nodes.le.0) nodes = 4                         000011
      call  getarg(2,dummy)                                             000012
      read(dummy,'(i10)',err=10) nps                                    000013
      if (nps .le. 0) nps = nodes                                       000014
      goto 20                                                           000015
10    nps = nodes                                                       000016
c---initialize the batch files and---                                   000017
c---load the parallel program to all processors---                      000018
20    do 30 i = 1, nodes                                                000019
        write(node_str,'(i2.2)')  i                                     000020
        batch_file(i)=path_name(:i_p)//'batch_'// node_str              000021
        open(98,file = batch_file(i),form='unformatted')                000022
        write(98) 0, -12345, 0                                          000023
        close(98)                                                       000024
        j = system('chmod 666 ' // batch_file)                          000025
        j=system('rsh '// cpu_name(i) // '-l ' // log_nm(i)             000026
     1           // path_name(:i_p) // 'node '//
     1           batch_file(i) // ' &')
                                                                        000027
30    continue                                                          000028
      write(out_file,'(a)') path_name(:i_p) // 'OUTPUT'                 000029
      open(99,file=out_file)                                            000030
      nseed0 = 1234567                                                  000031
      k = 1                                                             000032
      do 60 i = 1, nps                                                  000033
40      do 50 j = k, nodes                                              000034
          open(98,file=batch_file(j),form='unformatted')                000035
          read(98) m, n, nn                                             000036
          if (m .eq. 0) then                                            000037
            if (n .gt. 0) then                                          000038
              write(99,*) n, nn, cpu_name(j)                            000039
              print *, n, nn, cpu_name(j)                               000040
            endif                                                       000041
            rewind(98)                                                  000042
            write(98) i, need0, nn                                      000043
            close(98)                                                   000044
            x = rand(nseed0)                                            000045
            nseed0 = 1.e7 * rand(0)                                     000046
            if (j .eq. nodes)  then                                     000047
                k = 1                                                   000048
              else                                                      000049
                k = j + 1                                               000050
            endif                                                       000051
            goto 60                                                     000052
          endif                                                         000053
          close(98)                                                     000054
50      continue                                                        000055
        k = 1                                                           000056
        goto 40                                                         000057
60    continue
```

```
      do 70 i = 1, nodes                                                000058
70    notyet(i) = .true.                                                000059
80    repeat = .false.                                                  000060
      do 90 i = 1, nodes                                                000061
        if ( notyet(i) ) then                                           000062
          open(98,file=batch_file(i),form='unformatted')                000063
          read(98) m, n, nn                                             000064
          if (m .eq. 0) then                                            000065
            write(99,*) n,nn,cpu_name(i)                                000066
            print *, n,nn,cpu_name(i)                                   000067
            rewind(98)                                                  000068
            write(98) -1000, -1000                                      000069
            close(98)                                                   000070
            notyet(i) = .false.                                         000071
            goto 90                                                     000072
          endif                                                         000073
          close(98)                                                     000074
          repeat = .true.                                               000075
        endif                                                           000076
90    continue                                                          000077
      if ( repeat ) goto 80                                             000078
      close(99)                                                         000079
      j=system('sort -n -o ' // out_file // out_file)                   000080
      stop                                                              000081
777   print *, 'Please, give the number of processors and '             000082
     1         'number of particles as arguments !'
      stop                                                              000083
888   print *, 'Invalid number of processors. Execution ',             000084
     1         'canceled !!!.'
      stop                                                              000085
      end                                                               000086



      program node                                                      000001
      double precision  d1,d2,d3,d4,d5,d6                               000002
      character*80  batch_file                                          000003
      call getarg(1,batch_file)                                         000004
10    open(99,file=batch_file,form='unformatted')                       000005
      read(99) n, nseed                                                 000006
      if (n .ne. 0) then                                                000007
          if (n .eq. -1000) then                                        000008
            close(99,status='delete')                                   000009
            stop                                                        000010
          endif                                                         000011
          x = rand(nseed)                                               000012
          iter = 100000 * rand(0)                                       000013
          rewind(99)                                                    000014
          write(99) 0, n, iter                                          000015
          close(99)                                                     000016
        else                                                            000017
          close(99)                                                     000018
          goto 10                                                       000019
```

```
      endif                                                          000020
      d1 = 0.5d0                                                     000021
      d2 = 0.5d0                                                     000022
      d3 = 0.499975d0                                                000023
      d4 = 2.0d0                                                     000024
      d5 = 0.75d0                                                    000025
      d6 = 0.50025d0                                                 000026
      do 20  i = 1, iter                                             000027
        d1 = d3 * datan(d4 * dsin(d1) * dcos(d1) /                   000028
     1          (dcos(d1+d2) + dcos(d1-d2) - 1.0d0))
        d2 = d3 * datan(d4 * dsin(d2) * dcos(d2) /                   000029
     1          (dcos(d1+d2) +dcos(d1-d2) - 1.0d0))
        d5 = dsqrt(dexp(dlog(d5) / d6))                              000030
20    continue                                                       000031
      goto 10                                                        000032
      end                                                            000033
```

The statement line 25 of the program host uses the UNIX command ***chmod*** to grant access permission to the file batch_file. The option *666* that is included there guarantees that the owner, group and other user may freely perform read and write access to the file. This is extremely important whenever the executing processors belong to different *local area network*. In such environment, the same user may have a given *GPID* in one local area network, but he may belong to a different GPID on the other local area network. This also may happen even in the same local area network, where users are grouped to different GPIDs.

Another problem that one will face in using multiple processors that reside on different local area networks is that one must deal with the login names. The same user may login on one machine by using a given username, but he has to use a different username (or even borrow one from other user) to login on different machine. To tackle the above problem, the statement 26 uses an additional option, namely, "-l " together with "log_nm", the username which has a privilege to access the remote machine. Notice that the host processor, now, loads the program node, not itself (!). The technique described above are implemented in the program code of MCNPNFS for automatically loading of the code on all processor in a homogeneous system.

The program host uses the loop number 60 to farm out the task to all the node processors. Before assigning a task to a node processor, the host processor will check the first integer data on

the file batch_file. A zero value of the data signals the host processor that the node processor is ready to execute a new task. It means also that node processor has completed its previous task. On the other hand, the node processor will interpret the same data differently, i.e., that the processor host processor is still busy with the other works. This integer data, which determines both host and node processors whether they are able to perform further actions is called *semaphore*. In our programs above, this semaphore will be used by host and node processor to check whether it can use and update the contents of the batch_file.

Notice that no flag file is used during the message passing, because the interprocessor communication is entirely established by checking the status of the semaphore.

Both the programs also show how to use the message as efficient as possible. The host program uses it for sending the new seed to and for getting the "computing results" from the node processor. The node processor, in the opposite, will receive the new seed and send back the results. This apparently can reduce excessive use of files. Of course, this method of data transfer may only be applied when the data is still in a reasonable amount. Otherwise, separate files should be used for interprocessor communication and storing the temporary computing results. The statement line 6 in the program node shows how to read the message file efficiently, namely, by reading the necessary data. Again, it is to mentioned here that the ***close*** statement in line 16 of program node is very important. Without it, the content of the file batch_file remains in unchanged, even though the ***write*** statement in line 15 has already been executed. By this way, the host processor can distribute as soon as possible another new task to the node processor. The similar reasoning should also be applied to the statement line 53 of program host and statement line 74 of program node. However, the programs will (normaly) abort immediately, because they try to use the same unit number that is still connected to other file.

III. Parallelization of MCNP4 Code :

III.1 Program Flow of Sequential MCNP4 :

This section will describe briefly the program flow of sequential MCNP4 (hereafter, it will be called simply as MCNP4), so that it is easier for us to enlighten its differences with the MCNPNFS and to clarify the parallelization strategies. It is assumed that the code is of the version used under the UNIX operating system and supports the plotting capabilities.

MCNP, a general purpose, continuous-energy, three-dimensional Monte Carlo computer code for neutron-photon-electron transport was first released in June 1977[1] by the Los Alamos National Laboratory. Since then, it has been revised many times. Up to the time this report is written, the code has been named as MCNP4A.

The MCNP4 main program has 5 major subroutines, namely:

1) IMCN : This subroutine serves as problem setup, such as: initialize the addresses of dynamically allocated storage, reading the surface-source file, initialize the tally arrays, set up the geometry, etc. In a continue run, the set up is mainly be performed by reading the RUNTPE file.
2) XACT : Process all necessary calculations for preparing the cross-section table. Depending on the problem input, this subroutine frequently generates a huge file RUNTPE, because all cross-section data will be dumped there with other fixed data.
3) MCRUN : Supervise the whole transport calculations and generation of output file OUTP.
4) PLOT : Enable users to plot two-dimensional slices of a problem geometry given in the input file INP.
5) MCPLOT : Plot the tally results produced at the end of MCNP executions.

The two subroutines mentioned later are not implemented anymore in the MCNPNFS, because of their interactive plotting capabilities that are impossible to handle in a safe way by using our simple methods of parallelization.

The above subroutines are basically independent of each other, as can be seen in Figure 3.1 that visualizes the program flow of the main program MCNP. The only exception is that subroutine IMCN will always be called prior to the calling of subroutine MCRUN or XACT.

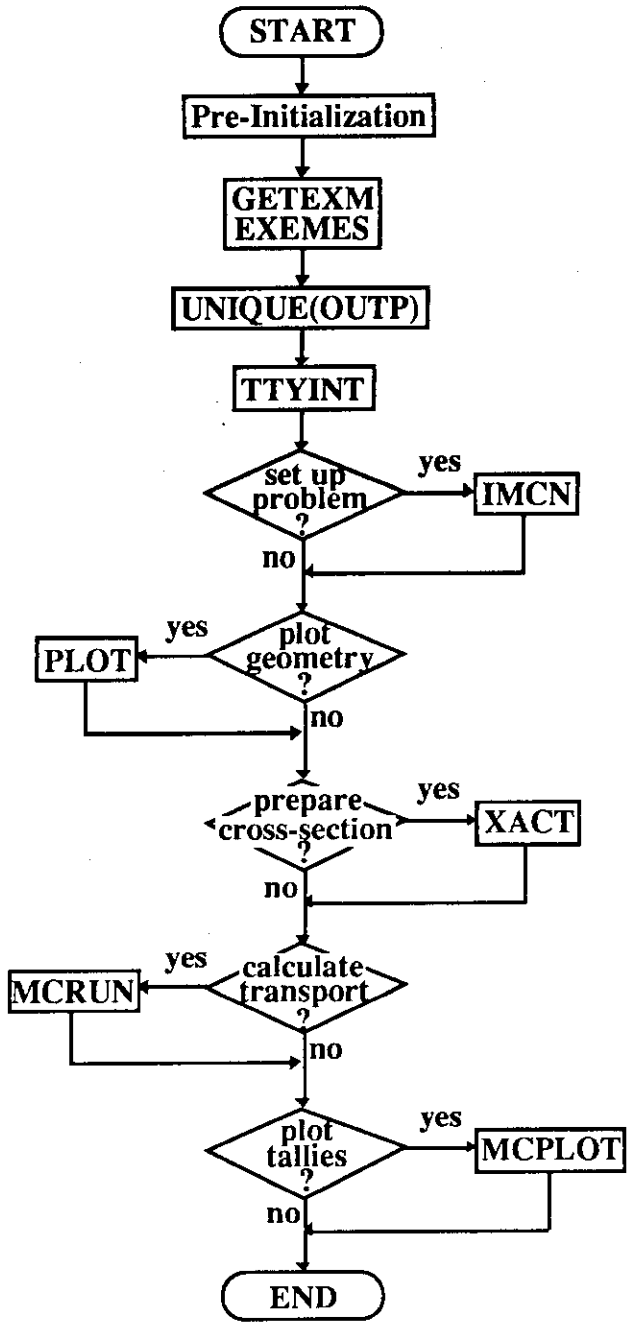

Figure 3.1 : Flow chart of main program MCNP4.

MCNP begins the execution with pre-initializing some global data. Most of them will be set to zero, while the rests are initialized to one. Then, subroutine GETEXM will get the execution line message (i.e., the arguments given at the invocation of MCNP4) and it will be interpreted by subroutine EXEMES. The latter will also read the input file INP, however, it is only the input cards in the message block that will be processed. Next, subroutine UNIQUE will search

a unique file name for OUTP file and create it as well. Further, the subroutine TTYINT is executed to enable a TTY interrupt, so that MCNP4 can respond, interactively, to the input given by users through the terminal. This capability enables users to plot the problem geometry and tally results during the history calculations. Any errors found during those stages will cause the program to terminate.

Depending on the problem input, the execution will continue to calling, in sequence, the subroutines: IMCN, PLOT, XACT, MCRUN and MCPLOT. Meanwhile, the OUTP file is generated through out the whole program stages.

As already mentioned, the problem setup is performed by subroutine IMCN. In an initiate-run, the subroutine will call many other subroutines, as shown in Figure 3.2. Any subroutines depicted in dashed boxes are called only when certain conditions are satisfied, as given in the input file INP. In the mean time, IMCN will also directly initialize some other data needed for the future calculations. However, in a continue-run, IMCN will setup the problem in a more straight forward fashion, mainly by invoking the subroutine TPEFIL, that reads the dump file RUNTPE. Upon finishing the reading of RUNTPE, some other data are also initialized and other subroutines are called, if required.

During the execution of MCNP4, most of time is spent in the subroutine MCRUN. Its program flow is very simple, because it invokes only 2 subroutines, namely, TRNSPT and OUTPUT (see Figure 3.3). TRNSPT calls subroutine UTASK, by which temporary arrays are zeroed and offsets of dynamically array storage are set. An initial random number is generated by subroutine ADVIJK and automatically transferred to subroutine HSTORY, where the actual history trackings are performed. The ADVIJK and HSTORY will be repeatedly called by TRNSPT until certain conditions are fulfilled, such as: number of particle histories is multiple of the preset value of periodical print out, Russian roulette should be updated, temporary results have to be dumped, etc. Also, the loop will terminate immediately if a fatal error occurs during the random walk calculations. Next, subroutine TRNSPT will call VTASK to sum up some

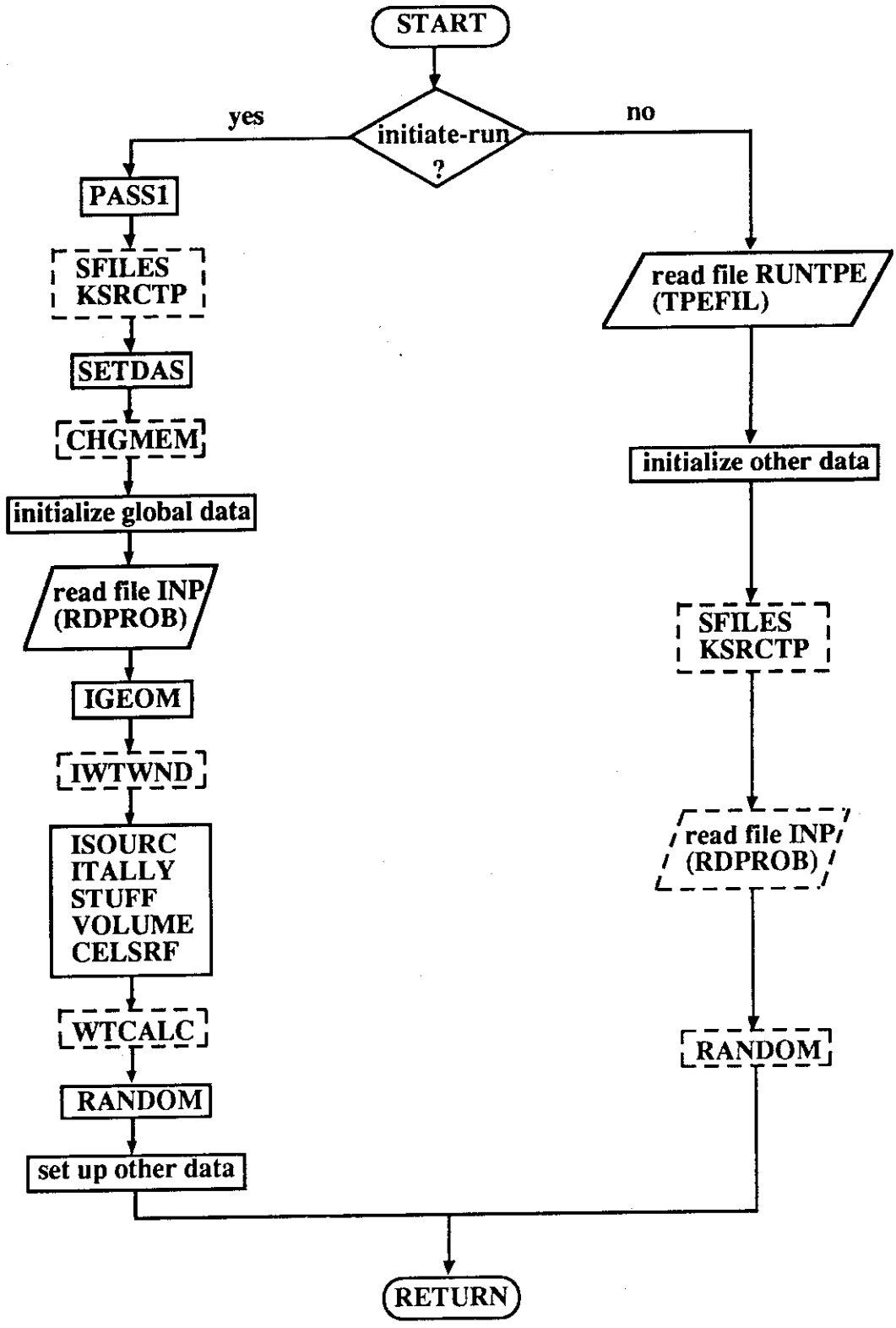

Figure 3.2 : Flow chart of subroutine IMCN. Dashed boxes indicate that the subroutines will be called only when necessary.

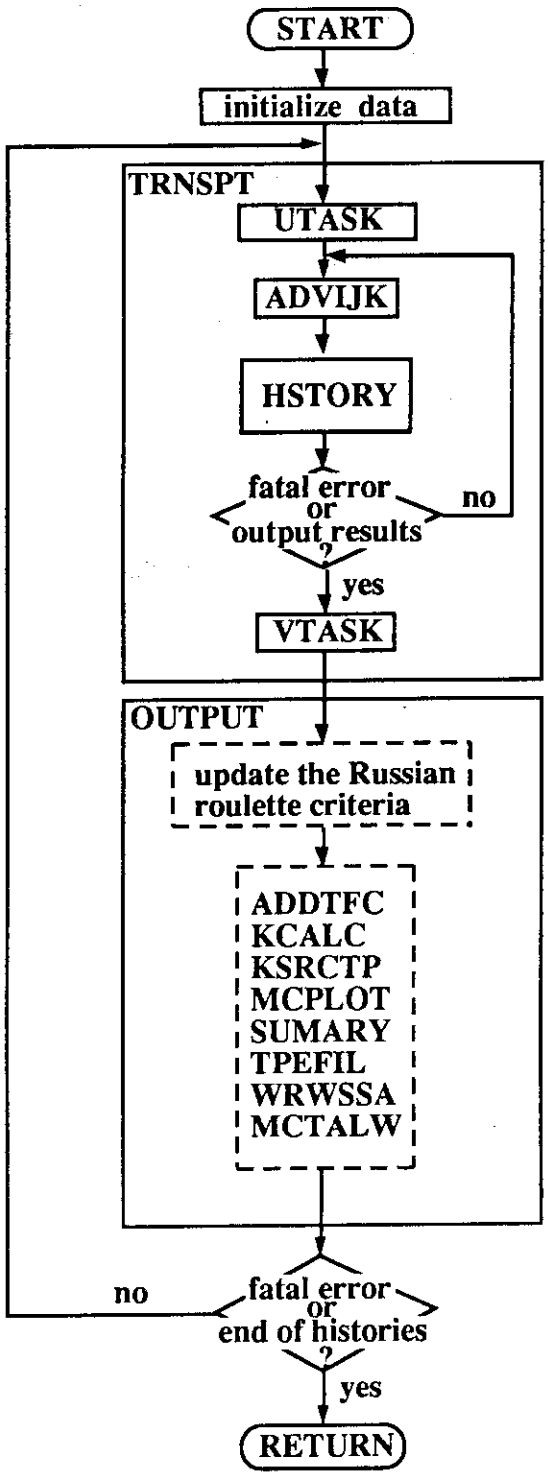

Figure 3.3 : Program flow of subroutine MCRUN. Dashed boxes indicate that the subroutines are invoked only when necessary.

temporary arrays to the permanent storage. Then, the program continues to calling subroutine OUTP, that updates the Russian roulette criteria when the number of particles reaches 200, 1000, 2000 and so forth. At that interval of 1000 particles, subroutine ADDTFC will be called

to add or to update the tally fluctuation charts. The interval will automatically double when number of particles reaches 20000, 40000 and so on. In a criticality calculation, OUTPUT will call subroutine KCALC to calculate the eigenvalue and fission source file be written and updated periodically by subroutine KSRCTP. Depending on the problem input, subroutine MCPLOT will be called to plot the tally results periodically or at the end of the problem. The output of MCNP is always written to the file OUTP. This is done periodically and at the end of the execution by calling the subroutine SUMARY. Similarly, MCTALW will save, however, only the tally results in the file MCTAL. The subroutine will always call RUNTPE to dump all the calculation results periodically and at the end the problem. Whenever a SSW card is found in the input file, subroutine WRWSSA is called to produce surface source file WSSA at the end of last history calculations. Finally, the program flow will loop back to the subroutine TRNSPT, whenever not all particle histories have been tracked yet. Otherwise, MCRUN will directly return to the main program. Also, fatal error will make MCRUN to terminate immediately.

III.2 Parallelization Strategies :

III.2.1. Hybrid Model of Parallelized MCNP:

Regardless of the programming model, all executing processors must perform the pre-initialization and problem setup. Also, the cross-section table should be prepared. Clearly, the first three major subroutines, i.e., IMCN, PLOT and XACT, should be performed sequentially, because:

- Those subroutines should be executed one after another and the results of previous subroutines will be needed by the next ones.
- All data that are to be processed and all operations performed in the process are exactly the same.
- Hence, the results are identical.

Theoretically, it is still possible to parallelize the MCNP at those stages, however, it is hardly to perform and it seems that it will lead to a poor parallel efficiency. On the other hand,

the subroutine MCRUN can be performed totally in parallel. This is due to the nature of Monte Carlo simulation, in which any particle history is independent of the previous histories. Again, the parallelism is disappeared in the execution of MCTAL, because it processes merely the tally results produced by MCRUN by displaying it on the terminal. On the basis of the above discussions, we adopted a hybrid model parallel program in parallelizing the MCNP4. The model is a combination of hostless and host-node model.

At the first stage, MCNPNFS (i.e., the host program) behaves as a host-node model. It will do whatever needed at the initialization stage of MCNP4 by exploiting subroutines IMCN and XACT. Then, all output files, especially OUTP and RUNTPE, are closed and the node program of MCNPNFS will be loaded on all processors specified in a configuration file that is read before the loading process begins. Upon finishing the automatic loading of the node program, the host program will immediately terminate. All those above processes are performed sequentially by the host processor.

At the next stage, MCNPNFS (i.e., the node program) will run concurrently on the remote processors and, if specified in the configuration file, also on the host program. This time the MCNPNFS adopts the hostless model, in which all executing processors communicate with each other, mainly, through one message file.

The adoption of hybrid model in parallelizing MCNP4 has the following advantages:

* It is easy to program. By separating the original program into two parts, we can concentrate in one part solely on how to distribute the tasks to all executing processors, while keeping the host program the same as the original code, as close as possible.
* The latter enables us to combine the parallelized and sequential version as a unity, so that MCNPNFS can offer users a new execution line option S, the sequential mode.
* The host program will reduce the burden of the file server, because it is the only one that accesses the cross-section file and creates the RUNTPE as well as OUTP files, during the initialization stage. Also, excessive creation of files can be avoided.
* The fact that the host program terminates immediately after the node program has been loaded is worthily, because the host processor will later be able to execute merely the

node program, when it is loaded on itself. This means that we can get the maximum performance of any executing processors in performing the random walk calculations.

III.2.2. Host Program of MCNPNFS:

As can be seen in Figure 3.4, the host program of MCNPNFS virtually does not differ from the original version of MCNP4. Indeed, it is only the ability of subroutine EXEMES in accepting two new options (S = sequential and H = heterogeneous mode) and the inclusion of new subroutine load_node that make the difference. Whenever MCNPNFS is executed in parallel mode, the host program will call subroutine load_node to load the node program of MCNPNFS on all executing processors. Then, the host program will immediately terminate. Notice that by the time node program is loaded, all the results of problem setup done by IMCN and cross section table prepared by XACT have been stored entirely in RUNTPE file. This file then serves as a message file, by which the host processor sends the data to the node processor. On the other hand, the MCNPNFS will behave exactly as the original MCNP4 when it is executed in sequential mode. Of course, only the host processor will be used in this case.

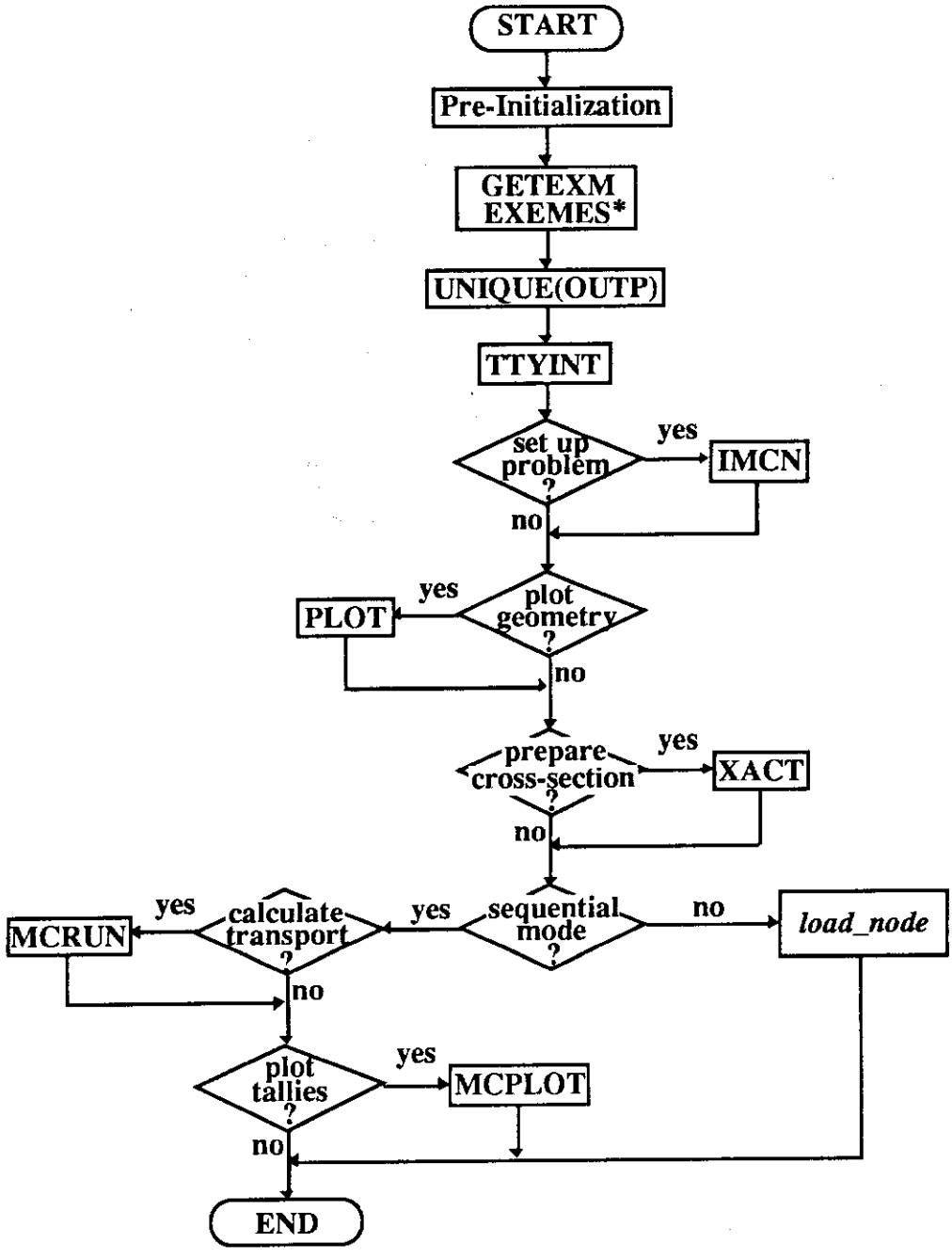

Figure 3.4 : Flow chart of host program of MCNPNFS. The asterisk denotes that the subroutine is modified.

III.2.3. Automatic Loading of Node Program :

Before loading the node program, the subroutine load_node will performs some preliminary steps as follows:

- Close all output files produced in the initialization stage. This will guarantees that the node program can later access them properly. Also, unnecessary files are deleted.
- Read a configuration file. The file contains: number of nodes (logical processors), number of (physical) processors, hostnames, full pathname of the directory, etc. Appendix A.3 will explain the detail of this configuration file.
- Deal with the changing of default filenames. As already known, MCNP4 always searches a unique filename before any output file is created. MCNPNFS keeps maintaining this capability.
- When necessary, MCNPNFS will move the input and output file to the directory specified in the configuration file.
- Check the validity of INP file, existence of RUNTPE file, etc., so that the node program won't terminate prematurely due to such fatal errors. The existence of KCODE card in INP file will also be detected here and when it is found, MCNPNFS will terminate immediately.
- Check whether the remote processors are in order or not. This done by a very simple way and it may not work well.

When everything is in order, the subroutine load_node will load the node program on all remote processors by using similar algorithm explained in Section II.6.

In a homogeneous computer system, the automatic loading is performed by the following FORTRAN statement:

```
      j = system('rsh ' // cpu_name // '-l ' // login // program //
     1           node //nodes // speed // speed_tot // path //
     2           b_fac // nps200 // npd1000 // exemes // ' &')
```

All arguments given in the above statement are having the same type, namely, character (string) variables. Their meaning and function are as follows:

- cpu_name : hostname of the respective processors.
- login : user login name.
-program : name of the node program of MCNPNFS.
- node : node number of the respective processor. This node number is unique and used by each processor to identify themselves.
- nodes : total number of nodes (logical processors). Due to the multitasking capability of UNIX operating system, user may load the node program on a given processor more than once. However, user can not gain any advantage and the performance of that processor will decline, otherwise.
- speed : relative speed of respective processor.
- speed_tot : sum of relative speed of all processors.
- path : full pathname of the directory where the shared file system resides on. The pathname may be different from one processor to another. However, it should refer to the same physical directory system.
- b_fac : batch factor. Together with speed and speed_tot, b_fac will determine how the task is distributed to the executing processors. Section III.2.4.D will explain it in detail.
- nps200 and npd1000 : preset-value for first and periodical update of Russian roulette criteria, respectively. By using it, user can change the default values (200 and 1000, respectively) to gain higher parallel efficiency.
- exemes : execution line message. This is nothing else than the (slightly modified) execution line given by user at the invocation of MCNPNFS.

MCNPNFS can also be well executed in a heterogeneous computer system, such as on a network that consists of workstations of different architectures and multitasking parallel computer (such as MONTE4).

From the point of view of parallel programming by using our method, the main problem that one must deal with is that every computer system may have its own internal and external data format, that differs from one to another. MCNPNFS won't have any problem with

internal data format, because all processors rely entirely on disk files during the message passing. However, those files must be interpreted exactly the same way by all processor, which means that different external data format may cause much problems. It is especially true when one uses binary (unformatted) data format for message files.

Fortunately, many systems provide users with FORTRAN compiler that can automatically convert (inappropriate) internal data format to a more standard external data format (such as IEEE format). Some compilers can (such as FORTRAN 77/SX used in MONTE4) even detect the *environment variables* and use them later for automatic data conversion. To exploit such capability, MCNPNFS uses a different algorithm in loading the node program on the remote processors in a heterogeneous system as follows:

```
      j = system('rsh ' // cpu_name // '-l ' // login // source //
     1           // sh_file)
```

By using the above FORTRAN statement, the host program simply orders the remote processor to execute the UNIX command ***source*** and passes the argument "sh_file" to it. The remote processor will then interpret the argument as a UNIX *batch file*, which will be read during the execution of ***source*** command. All data lines in the batch file will then be interpreted by the remote processor as UNIX command lines, so that the node program of MCNPNFS and other commands will automatically be executed, if they are specified in the batch file.

To pass the arguments to the remote processors, a given file called *argument file* is created and all arguments are stored in it. Later, the file will be read by all the executing processors. This is done just to simplify the user of MCNPNFS, as they do not need to prepare files other than a configuration file, that has a very similar format as the one use in running MCNPNFS in homogeneous system, and the batch files that have very simple format. Appendix A.3 will describe more detail about the contents of the batch file.

III.2.4. Node Program of MCNPNFS:

A. Main Program :

The node program begins with pre initialization of some global data, that is performed exactly the same as in MCNP4. Then, subroutine GETEXM will interpret the 9 arguments sent by the host processor or read the argument file, that also created by the host processor (in heterogeneous mode). From now on, all processors can identify themselves by using their own node number. Next, subroutine assign_file is called to assign all file names that will be used by the processor itself and by other processor. By this way, all processors will be introduced to each other. Further, subroutine EXEMES will interpret the execution line message that already prepared earlier by GETEXM.

Instead of searching a unique name for OUTP file, MCNPNFS will then simply open the OUTP file that is created by the host program and locates the file pointer at the end of file, so that the new entries can be appended to the old ones. However, this is only done by the (node) processor number 1. The other processors will create their own new (empty) files (i.e., para_OUTPxxxx, where xxxx denotes the node number of respective processor), instead. By using this trick, the OUTP file will always be written properly.

The subroutine TTYINT is deactivated, so that users can not perform a TTY interrupt during the execution of MCNPNFS. This is important because the loading of MCNPNFS is done by using UNIX command ***rsh***, that does not allow an invocation of any interactive command. This is also the reason MCNPNFS does not support plotting subroutines (PLOT and MCPLOT) any longer.

MCNPNFS will always call subroutine IMCN to setup the problem and random walk calculations are performed under the supervision of subroutine MCRUN. Figure 3.5 shows the flow of the main program of the node program. The subroutines with asterisk mean that they are already modified.

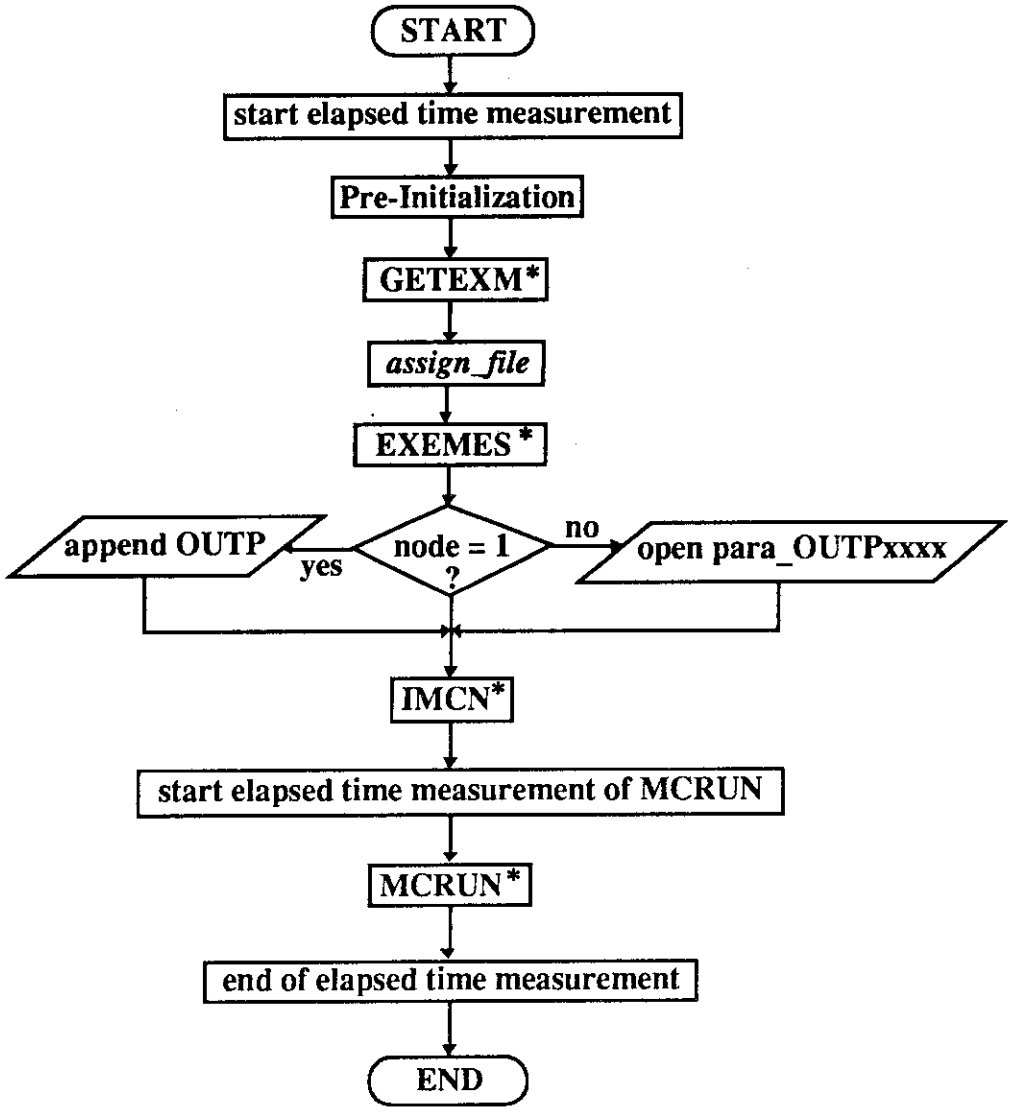

Figure 3.5 Flow chart of main program of MCNPNFS. Asterisks denote that the subroutines are modified.

B. Subroutine IMCN of MCNPNFS :

Because the host processor has already performed the problem setup and prepared the cross section table, IMCN will always begin with reading of RUNTPE file. In the code, it can be realized simply by forcing the program to flow directly to the statement number 440. (However, we have commented out all the statements above the statement number 440, so that the code size will be reduced, because many unnecessary subroutines are excluded.) This means that the IMCN will act likely as it is in a continue-run case. Comparing to the Figure 3.2, the flow chart of IMCN of MCNPNFS, as shown in Figure 3.6, is nothing else than the

right-hand side of the original IMCN. The differences are that the existence of KCODE card is checked and the program will directly terminate when it is found, also subroutine KSRCTP is not supported any longer. This redundant checking is performed because the criticality calculation problem (in continue-run) can only be detected after RUNTPE file is already read.

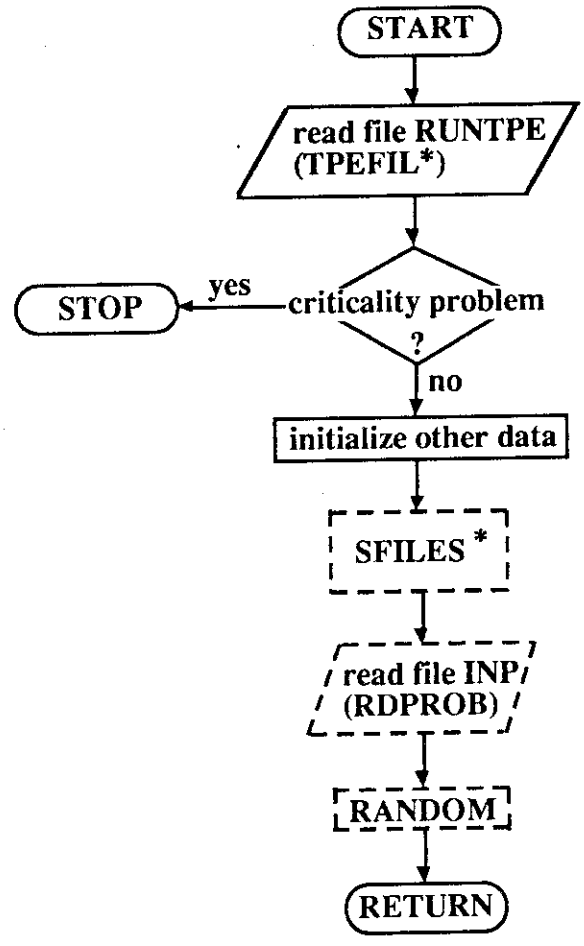

Figure 3.6 Flow chart of subroutine IMCN of MCNPNFS. Asterisks denote that the subroutines are modified.

C. Subroutine MCRUN of MCNPNFS :

There is no apparent modification that can be found in the subroutine MCRUN of MCNPNFS except the inclusion of one new subroutine initialize. The subroutine performs some initialization needed in the next calculation steps. Here, the role of each processor differs from the others. The processor number 1 will close the OUTP file and open a new file para_OUTP0001, that is used for its temporary output file. It also prepares the initial random

number, that is stored in the file para_RANDOM together with other data. The file will serve later as a batch file. On the other hand, the other processors will continue with other initialization process, that will also be performed by processor number 1.

Figure 3.7 shows only the main flow of subroutine IMCN, because subroutines TRNSPT and OUTPUT are completely different and will be explained separately for clarity.

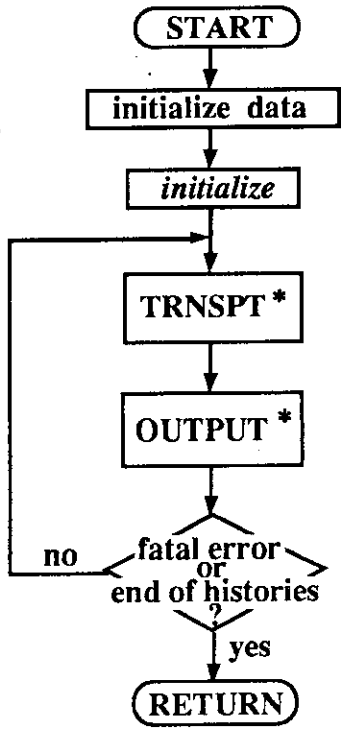

Figure 3.7 Flow chart of subroutine MCRUN of MCNPNFS. Asterisks denote that the subroutines are modified.

D. Subroutine TRNSPT, get_seed and OUTPUT of MCNPNFS:

In MCNP4, the whole random walk calculations are directly controlled by subroutine TRNSPT. Intensive checking is performed before and after one history tracking is carried out.

In MCNPNFS, most actions are transferred to a new subroutine, get_seed, although, the final checking is still carried out by TRNSPT. By using subroutine get_seed, each processor will determine what particle numbers should be tracked by it. This is realized by subdividing the total number of particles NPP into several *batches*. Every processor will then perform the random walk calculations as many times as number of particles in a batch, before they take

another new batch. In other words, the whole task will be divided into smaller sub tasks that are distributed to all executing processors.

Subroutine get_seed starts with checking the existence of the batch file para_RANDOM in a very similar way discussed in Section II.5, namely, by using ***rename*** function in realizing the locking algorithm. Every processor will try to rename the file para_RANDOM to para_OLDxxxx. They will keep trying to rename the file as long as they fail to do it. Upon a successful rename process, the processor will read 8 data stored in the file. The first three data of type real (i.e., RANI, RANJ and RANIJK) are needed for generating the initial random number. The next integer datum (INIF) is used to determine whether the random number should be advance by one, before the next random numbers are generated. The fifth datum (NPS, also integer) is the last particle number whose history is already tracked by other processor or even by itself. The next integer datum is a semaphore called now_output, that is used to determine whether the processor may continue to take another batch of particles or it should directly return to subroutine TRNSPT. The seventh datum is i_am_master, i.e., an integer semaphore that will later be used to determine the role of any processor when it executes the subroutine OUTPUT. The last datum is NST, another semaphore for determining whether fatal error has occurred or end of problem has been reached. After detecting a non-zero value of now_output, any processor will realize that the next batch should be (temporary) suspended. Before returning to TRNSPT, it will check the value of i_am_master. When its value is equal to 1, then, the contents of file para_OLDxxxx will be updated, so that the next processor will find out that i_am_master is zero. Next, the file will be renamed again to para_RANDOM, so that the other processors will be able to access it. Further, some parameters needed for output purposes will be updated, when necessary.

As soon as the processor finds out that now_output is zero, it will immediately determine how many particles exist in the next batch. It is calculated as follows. Suppose MCNPNFS is executed to solving the shielding problem by simulating NPP particles. Then, number of particles per batch, n_batch, is:

n_batch = nps200 * speed / (speed_tot * b_fac) + 1 for nps < nps200 or

n_batch = npd1000 * speed / (speed_tot * b_fac) + 1 for else.

We have already used the same notation (and meaning) as in Section III.2.3, of course, they are now treated as numerical values. The addition of one guarantees that n_batch will never be zero, so that infinite loop can be avoided. Parameters nps200 and npd1000, instead of NPP, are taken as basic values because they are used by MCNPNFS (also, by MCNP4) for updating the Russian roulette criteria as well as tally fluctuation charts.

By using those relations, one may expect that every processor will spent the same time interval in tracking the all the particles in a batch. This means the whole task will be distributed to all executing processors in a balance manner. This is true because we have include the speed ratio in the above relations. However, due to the nature of Monte Carlo simulation and instabilities of the system performances, the time spent for one history tracking will differ from the others. This means one processor will need longer time than the other. To compensate such "imperfection" one may use the batch factor b_fac. By giving an appropriate value (greater than or equal to 1) to b_fac, one may subdivide the total number of particles to a larger number of batches. This will give a chance to all processors to help each other, namely, by taking the batches that are supposedly taken by other "still-busy" processors.

After n_batch is determined, subroutine get_seed will set a new variable called nps_batch. This variable (i.e., nps_batch = nps + n_batch) will later be used for stopping criteria in the random walk calculation loop. Next, nps_batch will be check whether its value exceeds the preset values. When it is the case, nps_batch will be reset to the minimum preset value. By using this algorithm, MCNPNFS can periodically update the Russian roulette criteria and tally fluctuation result, print out all temporary result to OUTP file, etc. Further, subroutine get_seed will advance the random number as many times as n_batch and then update the contents of file para_OLDxxxx with the new values of nps_batch and of semaphore i_am_master. The semaphore is written to para_OLDxxxx as zero if nps_batch is not equal to any preset value or to 1, otherwise. The INIT value is always rewritten to the file as zero. After renaming again the para_OLDxxxx to para_RANDOM, get_seed will return to TRNSPT.

Subroutine TRNSPT will continue with checking the value of now_output. A zero value indicates that TRNSPT should call subroutine HSTORY to begin with the random walk calculations. Now, subroutines HSTORY and ADVIJK will be repeatedly called as many times as n_batch. Then, the program loops back to get a new batch. The only checking that is performed by TRNSPT is whether fatal error has occurred during the calculations. Upon detecting a fatal error (i.e., NST is not equal to zero), TRNSPT will call subroutine node_quit and the program will flow directly to call OUTPUT. Subroutine node_quit broadcasts a fatal error message to all executing processors by creating an *error file* (i.e., para_FATAL_ERROR). It also updates the content of para_RANDOM file by writing a non zero value of NST to the file.

Subroutine OUTPUT is called, normally, when Russian roulette criteria and tally fluctuation charts should be updated, periodically prints out the results to OUTP file, dumps all temporary result in RUNTPE file, as well as at the of the problem. In MCNPNFS, any fatal error will also lead to calling the OUTPUT.

In contrast to the execution of TRNSPT, where all processors perform their batch calculations simultaneously and (relatively) independent of each other, subroutine OUTPUT will force all executing processors to wait for others to complete their last task. It is the critical moment of parallel processing, where all processors should be synchronized and may or may not do the same actions. Because of this synchronization, the parallelism is disappeared, meaning that the parallel efficiency will be lower.

Subroutine OUTPUT begins with calling the subroutine v_task, which is similar to, but not the same as VTASK. The subroutine mainly serves to sum up all temporary arrays to other temporary arrays, so that unnecessary data, such as accounting data, do not have to be transferred to other processor, when not needed. This subroutine is extremely important as it can reduce the communication time during the synchronization, significantly.

By using parameter i_am_master, subroutine OUTPUT will decide the role of each executing processor. The processor that has now_output of value one will act as *master* processor. Otherwise, it will be a *slave* processor.

The master, meaning that it is the fastest processor that completed its last batch calculations, will call subroutine receive_slave, by which it receives necessary data sent by all slave processors. It begins with calling the VTASK and then reads the message file para_NEWxxxx. Once the file is detected, meaning that the slave processor already sent the data, the master processor will read its contents and sum them up to its own data. Then, the file will be deleted. The master processor will repeat the above reading process as long as not all processors have sent their data yet.

On the other hand, the slave processors will send their data by calling subroutine send_master. By using this subroutine, every slave processor will write its data to its own file para_OLDxxxx. After completing the writing process, it renames the file to file para_NEWxxxx, so that the master processor can receive the data. By using such algorithm, the master processor will always can read file para_NEWxxxx properly, as soon as it detects it. Then, the slave processor will call subroutine receive_master, by which it waits for the master processor preparing the file para_DATA_POOL. This performed by reading a semaphore contained in the file para_SLAVExxxx. After detecting a non-zero value of semaphore, it will continue to read the file para_DATA_POOL, otherwise, it has to try the reading again until the master is ready.

After the master processor receives the data from all slave processors, the summation results will be written to file para_DATA_POOL. Then, it signals the slaves that the file has completed, i.e., by writing a non-zero value to files para_SLAVExxxx. They are performed in the subroutine send_slave.

Further, both master and slave processors will successively update the Russian roulette criteria and tally fluctuation charts, when the number of particles meets the preset values. However, it is only the master processor that will call subroutines SUMARY, TPEFIL, WRWSSA and MCTALW. Again, this algorithm guarantees that all files involved in these stages can be accessed properly.

Before returning to subroutine MCRUN, both master and slave processors will call subroutine reset_value, in which some temporary accounting arrays will be assign to the newest values. Also, master processor will again redirect the OUTP file to para_OUTPxxxx.

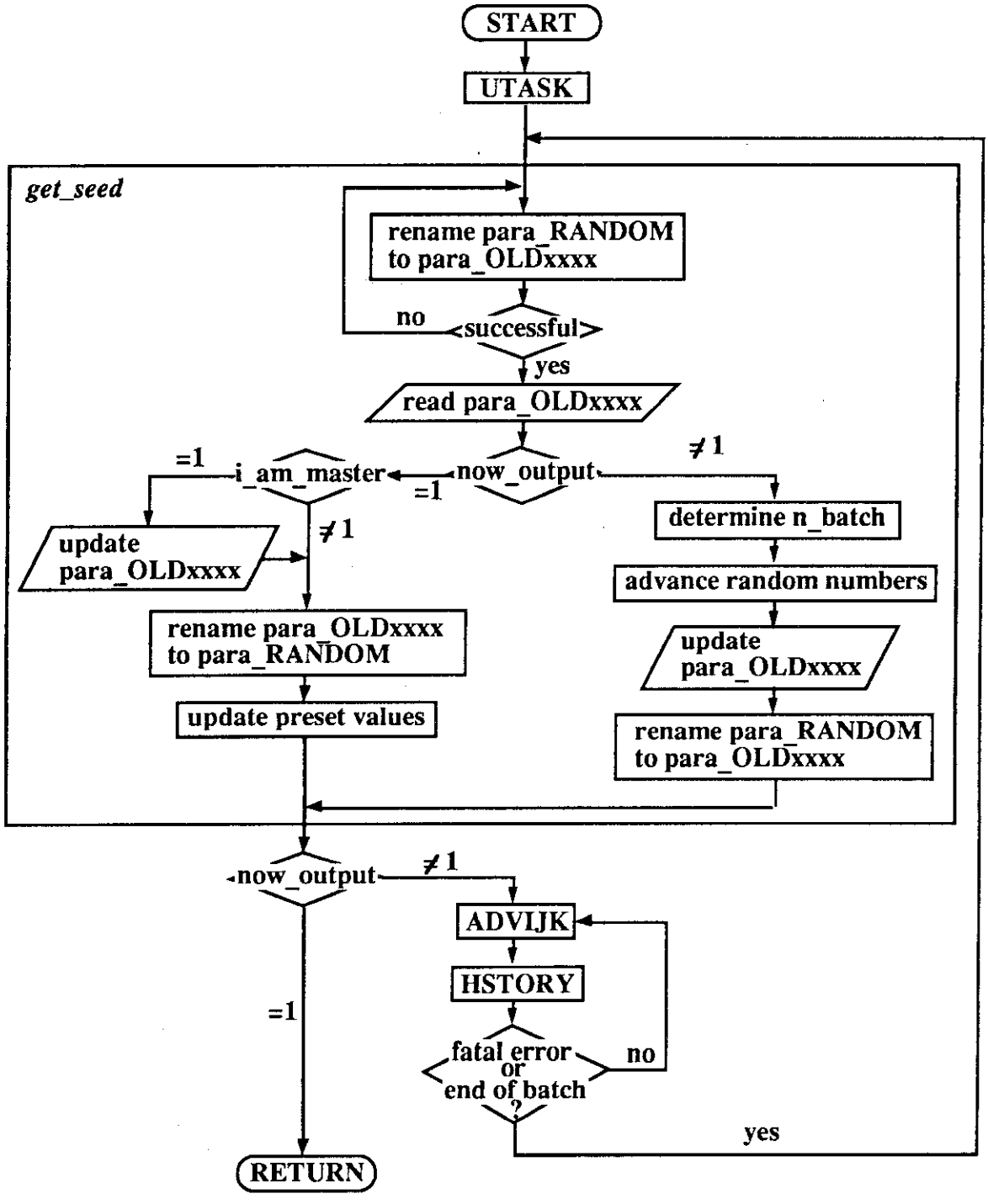

Figure 3.8 : Program flow of subroutine TRNSPT of MCNPNFS. Dashed boxes indicate that the subroutines will be called only when necessary.

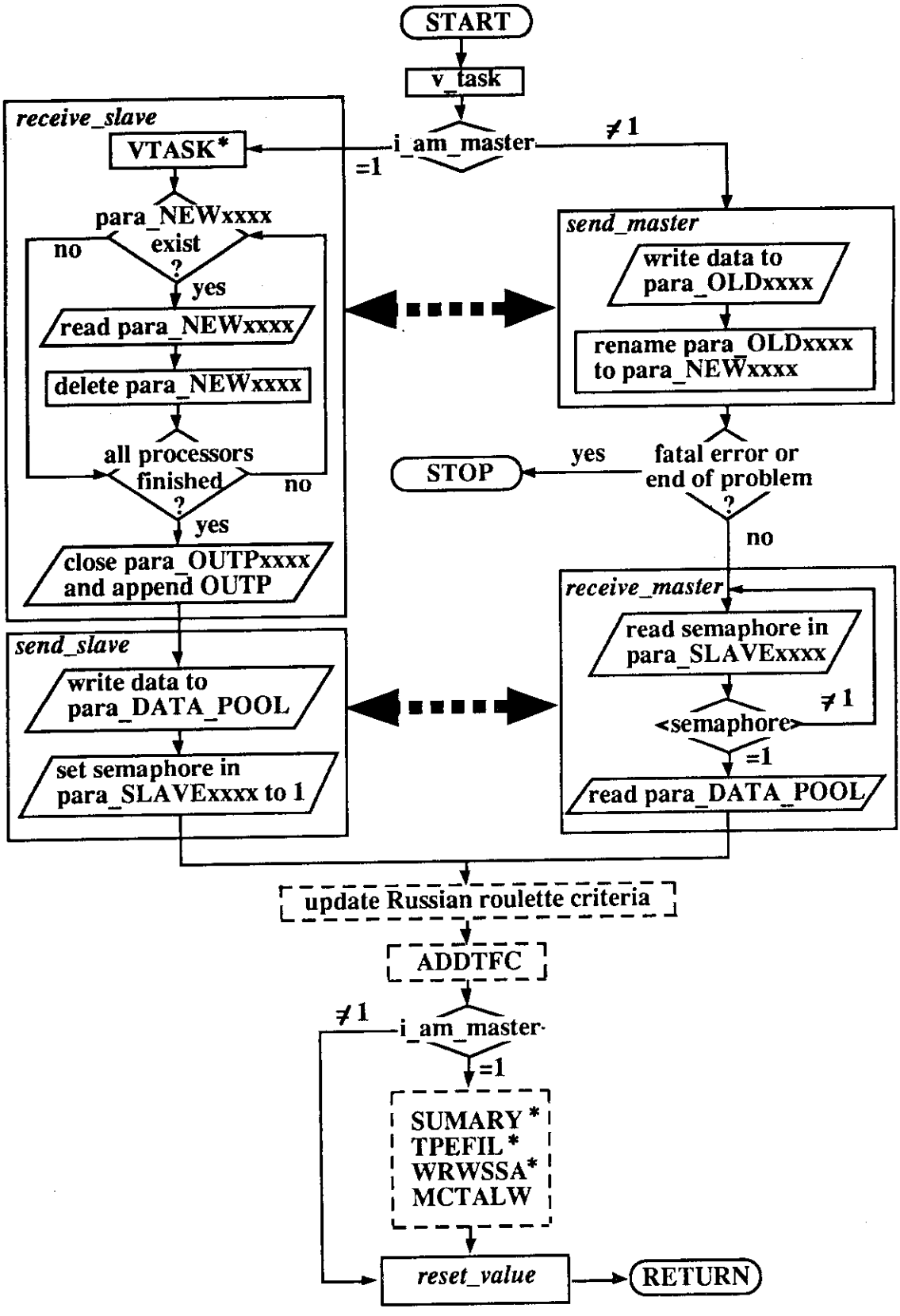

Figure 3.9 : Program flow of subroutine OUTPUT of MCNPNFS. Dashed arrowed-lines indicate that master processor communicates with slave processor.

IV. Performance Measurements and Analyses of the Results.

IV.1. Working Environment :

A. Workstation Network :

Three workstations of different model have been used to execute MCNPNFS concurrently. Two of them, Sun SPARC 10 model 40 (S10/40) and Sun SPARC 2 (S2), are available in our laboratory. They are running under the SUN-OS 4.1.1 Rev. B. The other one is a Fujitsu S-4/10 model 30 (S10/30) that is located in the other laboratory and is running under the SUN-OS 4.1.3. They are all linked through Ethernet communication network. One of the workstation (S2) exports its file system to the others by using Network File System (NFS) service. All files needed in the interprocessor communication during the execution of MCNPNFS reside on the file system.

During the executions, all workstations were dedicated only for this purpose, meaning that no other users were using them nor other applications were executed, except the OpenWindows, under which we always worked. These guarantee that all workstations will give their best performance, so that the true speedup of the workstation network can be achieved.

B. Monte4 :

Monte4 is a parallel computer with a capability for vector processing and special pipelines called Monte Carlo pipelines which accelerate performances in processing of Monte Carlo Codes for particle transport problem. We have never used these capabilities in this work since we focused on the investigation of the applicability MCNPNFS to parallel computers. So the execution time on Monte4 in this report will be reduced by coding suitable to the hardware features of Monte4.

No special preparation is needed to execute MCNPNFS in this system. The execution is performed in the interactive (normal user) mode. No attempt has been made to execute MCNPNFS in batch mode.

In this system, we also executed the MCNPNFS whenever all the 4 processors were idle, i.e., no other tasks were being executed, except the system's tasks, nor other users were logging in. Again, these enable us to reach the maximum speedup of Monte4 in executing the MCNPNFS.

MCNPNFS can be well executed on Monte4 because the computer also runs under the UNIX operating system, i.e., Monte-UX Release 3.1 Version 1, so that multitasking is supported.

IV.2. Problems to solve :

A wide variety of problems have been executed by MCNPNFS. The executions serve mainly to get the performance of MCNPNFS by comparing with MCNP4 executing the same problems. The calculating results of both codes were also compared to make sure that there is no discrepancy.

The 25 problem inputs that are available in the original code package served as the main problem inputs. From those, 17 problem inputs can be executed properly by MCNPNFS. Three photon benchmark problems and two neutron benchmark problems were also executed. Likewise, one spent fuel cask problem was tested. Table IV.1 shows the problem input file names, type of problems and origin of the files. Time measurement data of the input filenames marked with asterisk are not available. They are included here just to clarify that MCNPNFS is also able to execute those files properly.

Table IV.1. : Filenames, type of problem and origin of the files used to test MCNPNFS.

INPUT FILENAME	Problem to solve	Origin
INP01	Neutron source in a graphite ball.	Original MCNP code package.
INP02	Neutron volume source in a Boron ball.	"
INP03	General neutron source.	"
INP04	Photon problem in Uranium hydride ball.	"
INP05	Neutron problem in a toroidal TOKAMAK.	"
INP06	Variance reduction, exponential transform.	"
INP07*	Generation surface source file.	"
INP10	Neutron-photon problem in complicated geometry.	"

INP11	Neutron problem in complicated geometry.	
INP12	Well logging problem.	"
INP13	Neutron problem in a rotational symmetry geometry.	
INP14	General sources in repeated structures.	"
INP15	Filled lattice and skewed lattice geometry.	"
INP16	General source in lattice.	"
INP20	Continuous energy electron problem.	"
INP21*	Surface source file in photon-electron problem.	"
INP23	Forward 80-group electron-photon detector chip problem.	"
PA5	Skyshine radiation from Co-60 gamma source.	Photon benchmark problem.
PA6	Co-60 air-over-ground problem.	"
PA9	Thermo Luminescent Dosimeter radiation experiment.	"
NA2	Neutron energy spectra from a pulsed sphere experiment.	Neutron benchmark problem.
NA4	Neutron energy spectra from fusion shielding experiment.	"
CASKNG	Cf source in a fuel spent cask.	Reference (6)

IV.3. Performance Measurements :

The performance of MCNPNFS is obtained by measuring both CPU and elapsed (wall-clock) time needed to solve any of the problems above. The results are then compared with ones of MCNP4. However, it is only the MCRUN-part, not the total execution time that is taken into account, because it is only this part that is parallelized in MCNPNFS.

The CPU time measurements are already available at the end of OUTP file listing. However, no measurement of elapsed time is available in MCNP4. For that reason, we have implemented the function ***itime*** to measure the elapsed time, somewhere in the code, as can be seen in Figure 3.5.

It is important to measure the elapsed time as the CPU time does not include the "sleep" time, such as the time spent in the network communication. The determination of *speedup* that is based only on the CPU time will always lead to a too optimistic (i.e., ideal) value. Contrarily, the measurement of speedup based on the elapsed time will give a too pessimistic value, because it includes all the time spent by other processes, which do not have any relationship with the parallel processing itself (such as other processes owned by other applications that are running simultaneously in a multitasking system, etc.). By determining the speedup based on both time measurements we can, at least, predict the speedup in a more reasonable way, says, by taking the average from those values.

IV.4. Measurement Results :

A. CPU- and Elapsed Time :

All measurement results are tabulated on Table IV.2, IV.3 and IV.4. The measurements are all given in seconds. The CPU time as well as elapsed time required by single processor on the workstation network to execute MCNP4 are measured individually for determining the parallel efficiency of MCNPNFS on the network. However, on Monte4, any problem input is only executed (sequentially) once, because all the four processors are identical.

B. Speedup and Parallel Efficiency of MCNPNFS :

To determine the speedup and parallel efficiency of a parallel processing on a computer system with uniform speed, such as Monte4, is very simple. By defining T_s as the time spent in solving a computing problem using a sequential code on a single processor and T_p as the time required to solve the same problem but using a parallelized code on N processors concurrently, speedup and parallel efficiency can be calculated as follows:

$$S = \text{speedup} \equiv \frac{T_s}{T_p} \qquad (4.1)$$

$$E = \text{parallel efficiency} \equiv \frac{S}{N} \qquad (4.2)$$

However, in a network that comprises workstations of different model and speed, the speedup and parallel efficiency must be determined in rather indirect manner as follows. We define T_{is} as the time needed by a sequential code to solve a certain computing problem on a single processor number i, T_p as the time taken by a parallelized code to solve the same problem concurrently on N processors and T_{js} (i.e., reference time) is chosen arbitrarily from one of the N T_{is}'. Then, we can determine the speedup similar to Eq.(4.1), namely:

$$S_j = \text{speedup} \equiv \frac{T_{js}}{T_p} \qquad (4.1.a)$$

To obtain a similar expression for parallel efficiency, we have to define a new parameter, called *equivalent number of processor*, as follows:

$$N_j = \text{equivalent number of processors} \equiv T_{js} * \sum_{i=1}^{N} \frac{1}{T_{is}} \qquad (4.3)$$

Notice that the Eqs.(4.1.a) and (4.3) will always give different values of speedup as well as equivalent number of processor for different values of T_{js}. One commonly takes the time required by the fastest or the slowest processor as T_{js}.

However, as can be seen from Eq.(4.2.a) below, the parallel efficiency of a network having non-uniform speed of workstations will have a value, which is independent of T_{js}, as follows:

$$E = \text{parallel efficiency} \equiv \frac{S_j}{N_j} \qquad (4.2.a)$$

Table IV.5 and IV.6 show the speedup as well as parallel efficiency of MCNPNFS, which are based on the CPU time and elapse time measurements, for workstation network and Monte4, respectively. In case of workstation network, T_{is} of the fastest processor (i.e., S10/40) is taken as the reference time T_{js}.

Table IV.2. : CPU time of MCNP4 executed on single processor in workstation network and Monte4.

INPUT FILENAME	Number of Particles	CPU time (seconds)			
		S2	S10/30	S10/40	Monte4
INP01	800,000	4368	2496	1615	1125
INP02	200,000	3444	1884	1233	841
INP03	200,000	2676	1440	1187	836
INP04	80,000	3384	1800	1348	847
INP05	20,000	3996	1908	1380	727
INP06	160,000	1042	629	469	-
INP10	50,000	5550	2775	2141	1153
INP11	80,000	3168	1608	1195	711
INP12	40,000	4116	2148	1411	926
INP13	400,000	4286	2160	1694	1078
INP14	320,000	2624	1952	1259	915
INP15	200,000	2532	1428	1160	745
INP16	100,000	2140	1250	946	618
INP20	200,000	3324	1476	1271	811
INP23	50,000	3110	1390	1213	835
PA5	38,000	4720	2337	1783	1120
PA6	200,000	-	-	-	905
PA9	5,000	3348	1590	985	766
NA2	100,000	2700	1500	1097	739
NA4	200,000	3276	1812	1306	923
CASKNG	10,000	4185	2130	1654	1166

Table IV.3. : Elapsed time of MCNP4 executed on single processor in workstation network and Monte4.

INPUT FILENAME	Number of Particles	Elapsed time (seconds)			
		S2	S10/30	S10/40	Monte4
INP01	800,000	4480	2520	1622	1125
INP02	200,000	3500	1920	1240	841
INP03	200,000	2720	1460	1201	837
INP04	80,000	3440	1820	1372	847
INP05	20,000	4060	1940	1388	727
INP06	160,000	1056	632	477	-
INP10	50,000	5625	2800	2157	1153
INP11	80,000	3220	1620	1220	712
INP12	40,000	4180	2220	1456	928
INP13	400,000	4343	2200	1722	1081
INP14	320,000	2645	1579	1304	917
INP15	200,000	2580	1460	1188	746
INP16	100,000	2167	1267	968	619
INP20	200,000	3360	1480	1291	819
INP23	50,000	3167	1417	1245	836
PA5	38,000	4769	2375	1790	1120
PA6	200,000	-	-	-	907
PA9	5,000	3400	1610	989	766
NA2	100,000	2740	1520	1103	740
NA4	200,000	3340	1860	1327	924
CASKNG	10,000	4250	2175	1664	1166

Table IV.4. : CPU- and elapsed time of MCNPNFS executed concurrently on 3 processors in workstation network and 4 processors in Monte4.

INPUT FILENAME	Number of Particles	CPU time (seconds)		Elapsed time (seconds)	
		Workstation Network	Monte4	Workstation Network	Monte4
INP01	800,000	894	319	1046	351
INP02	200,000	728	283	845	317
INP03	200,000	585	262	725	295
INP04	80,000	673	254	780	272
INP05	20,000	785	210	934	221
INP06	160,000	241	-	365	-
INP10	50,000	1007	328	1101	343
INP11	80,000	592	203	766	220
INP12	40,000	1152	455	1612	618
INP13	400,000	950	352	1204	397
INP14	320,000	681	318	854	382
INP15	200,000	628	241	788	273
INP16	100,000	488	190	643	211
INP20	200,000	692	260	843	293
INP23	50,000	696	252	792	278
PA5	38,000	947	322	1125	340
PA6	200,000	-	265	-	320
PA9	5,000	528	215	592	233
NA2	100,000	555	233	721	258
NA4	200,000	1069	505	1403	716
CASKNG	10,000	804	322	911	346

Table IV.5. : Speedup and parallel efficiency of MCNPNFS executed concurrently on 3 processors in workstation network. Figures in parentheses indicate the relative errors (in per cent).

INPUT FILENAME	Number of Particles	Based on CPU time		Based on Elapsed time	
		Speedup	Parallel Efficiency(%)	Speedup	Parallel Efficiency(%)
INP01	800,000	1.81	89.6	1.55	77.3
INP02	200,000	1.70	84.2	1.47	73.4
INP03	200,000	2.03	89.5	1.66	73.2
INP04	80,000	2.00	93.3	1.76	81.7
INP05	20,000	1.76	85.0	1.49	72.2
INP06	160,000	1.95	88.6	1.31	59.2
INP10	50,000	2.13	98.6	1.96	91.0
INP11	80,000	2.02	95.2	1.59	74.7
INP12	40,000	1.23	61.3	.90	45.1
INP13	400,000	1.78	81.8	1.43	65.6
INP14	320,000	1.85	87.0	1.53	65.9
INP15	200,000	1.85	81.4	1.51	66.3
INP16	100,000	1.94	88.2	1.51	68.1
INP20	200,000	1.84	81.9	1.53	67.9
INP23	50,000	1.74	77.0	1.57	69.2
PA5	38,000	1.88	88.0	1.59	74.7
PA6	200,000	-	-	-	-
PA9	5,000	1.87	97.5	1.67	87.7
NA2	100,000	1.98	92.5	1.53	71.9
NA4	200,000	1.22	57.6	0.95	44.8
CASKNG	10,000	2.06	94.7	1.83	84.7
AVERAGE values		***1.83 (13)***	***85.7 (12)***	***1.52 (16)***	***70.7 (16)***

Table IV.6. : Speedup and parallel efficiency of MCNPNFS executed concurrently on 4 processors in Monte4. Figures in parentheses indicate the relative errors (in per cent).

INPUT FILENAME	Number of Particles	Based on CPU time		Based on Elapsed time	
		Speedup	Parallel Efficiency(%)	Speedup	Parallel Efficiency(%)
INP01	800,000	3.53	88.2	3.21	80.1
INP02	200,000	2.97	74.3	2.65	66.3
INP03	200,000	3.19	79.8	2.84	70.9
INP04	80,000	3.34	83.4	3.12	77.9
INP05	20,000	3.46	86.6	3.29	82.3
INP06	160,000	-	-	-	-
INP10	50,000	3.52	87.9	3.36	84.1
INP11	80,000	3.50	87.6	3.24	80.9
INP12	40,000	2.04	50.9	1.50	37.6
INP13	400,000	3.06	76.6	2.72	68.1
INP14	320,000	2.88	71.9	2.40	60.0
INP15	200,000	3.09	77.3	2.73	68.3
INP16	100,000	3.25	81.3	2.93	73.4
INP20	200,000	3.12	78.0	2.80	69.9
INP23	50,000	3.31	82.9	3.01	75.2
PA5	38,000	3.48	87.0	3.29	82.4
PA6	200,000	3.42	85.4	2.83	70.9
PA9	5,000	3.56	89.1	3.29	82.2
NA2	100,000	3.17	79.3	2.87	71.7
NA4	200,000	1.83	45.7	1.29	32.3
CASKNG	10,000	3.62	90.5	3.37	84.3
AVERAGE values		***3.17 (14)***	***79.2 (14)***	***2.84 (19)***	***70.9 (19)***

IV.5 Analyses of the Results :

By observing Table IV.5 and 6, it is clear now that speedup and parallel efficiency of MCNPNFS depend, more or less, on the problems that are to be solved. It is quite normal, because some problems are computing-intensive, while the others need more interprocessor communications, in which data transfer occurs frequently.

Problem INP06 is a good example, where the history trackings are performed very fast, due to variance reduction methods that are applied to the problem. The time needed to track all particles in a batch are very short (see Table IV.2 and 3), so that simultaneous accesses to the message files occur frequently, meaning that the executing processor waste too much time waiting for the message file to be "free."

Different thing happened in the case problem INP12. From our observations, it seems that it is an ill-problem, in which the random sequences exceed the number of strides many times. When it happened, the remaining processors, which have already done their tasks and are ready to update the Russian roulette criteria, must wait for the processors that performed the "long" history tracking. Consequently, too much time is spent in the synchronization stage.

In the case of problem NA4, the parallel efficiency and speedup were low due to a huge amount of data that should be transferred during the synchronization. The MCNP4 adopt an algorithm in which all data needed in updating the tally fluctuation charts must be transferred together with other data during the update of Russian roulette criteria. It occurs so frequently, especially in the initial phase, so that it causes too many disk accesses, which will lower the parallel efficiency.

Even though MCNPNFS has already introduced some modifications in avoiding to transfer unnecessary data during the synchronization (i.e., in the subroutines v_task and VTASK), however, MCNPNFS keeps maintaining the algorithm of MCNP4 in updating the tally fluctuation charts. By doing it, we can still be able to get exactly the same tally results as of MCNP4. In the future, this algorithm should be changed, as far as possible, in order to get a higher parallel efficiency of MCNPNFS.

From Table IV.5 and 6 we can also see how the speedup and parallel efficiency that are determined from CPU time differ greatly from the ones that are calculated by using elapsed time. Table IV.7 show the percentage difference between the parallel efficiency that is calculated by using CPU time and the one that is based on elapsed time, in the workstation network and Monte4, respectively.

An attempt to execute MCNPNFS simultaneously on workstation network and Monte4 has also been conducted. Unfortunately, due to the shortage of time, we can not have any measurement data. However, from the preliminary executions we have confirmed that the computing results agreed very well with of the MCNP4, with some minor differences. Those differences came entirely from the difference of computing precision between Monte4 and workstations. This discrepancies can even be observed when one compares the computing results of MCNP4 on Monte4 with ones that produced on workstation. In the future, it may be fruitful to continue this attempt more seriously.

Table IV.7. : Percentage difference between parallel efficiency that is based on elapsed time and that is based on CPU time. Figures in parentheses indicate relative errors (in per cent).

INPUT FILENAME	Workstation Network	Monte4
INP01	12.3	8.1
INP02	10.8	8.0
INP03	16.3	8.9
INP04	11.6	5.5
INP05	12.8	4.3
INP06	29.4	-
INP10	7.6	3.8
INP11	20.5	6.7
INP12	16.2	13.3
INP13	16.2	8.5
INP14	21.1	11.9
INP15	15.1	9.0
INP16	20.1	7.9
INP20	14.0	8.1
INP23	7.8	7.7
PA5	13.3	4.6
PA6	-	14.5
PA9	9.8	6.9
NA2	20.6	7.6
NA4	12.8	13.4
CASKNG	10.0	6.2
AVERAGE values	***14.92 (35)***	***8.25 (35)***

V. Conclusions :

Some conclusions may be drawn here:

1) It is proved that such our simple FORTRAN algorithm, together with shares files and UNIX FORTRAN compilers are highly capable in realizing parallel processing in a distributed memory system.
2) The algorithms are simple, easy to understand and completely conform to FORTRAN 77 standard. Also, no parallel programming environment are required, so that no need to learn new things, except some basic knowledge in UNIX command.
3) Therefore, the parallelized code developed by using our method are highly portable. It is worthwhile to mention here that MCNPNFS (if one may want to) can be transferred from workstation to Monte4 with only two statements that should be modified.
4) MCNP4 has been successfully parallelized for workstation network and Monte4.
5) High speedup and parallel efficiency are achieved by MCNPNFS in executing various shielding problems. Based on the CPU time measurements, the speedup of MCNPNFS on our workstation network is found to vary from 1.22 to 2.13 times the MCNP4 executions on the fastest workstation, while the average speedup is 1.83. Those values correspond to the parallel efficiency of 57.6% to 98.6% and average value of 85.7%. On Monte4, MCNPNFS gained speedup from as low as 1.83 to as high as 3.62 and average value of 3.17. It is proportional to parallel efficiency of 45.7% to 90.5% and 79.2% in average.

   With such high parallel efficiencies, MCNPNFS has already surpassed, or at least equal to, the other parallelized versions of MCNP4, which are developed by other more sophisticated methods (see Refs. 5 - 9), in executing the same or equivalent problems.
6) We also found that lower speedup as well as parallel efficiency must be faced, when we determined those values by using elapsed time measurement data. In this case, the execution of MCNPNFS on the workstation network achieved speedup of 0.9 to 1.96 and 1.52 in average. The corresponding parallel efficiencies were 45.1% to 91.0% and 70.7% in average.

On the other hand, the execution of MCNPNFS on Monte4 showed the speedup ranging from 1.29 to 3.37 and average value of 2.84. Likewise, the parallel efficiency lowered from 32.3% to 84.3% and average value of 70.9%.

In average, discrepancies as large as 14.9% (in workstation network) and 8.3% (in Monte4) of parallel efficiency were found, when we compared the parallel efficiencies obtained by using CPU time with ones expressed in elapsed time.

7) The previous facts indicate that one may not rely solely on CPU time measurement in determining the speedup and parallel efficiency of parallelized codes. The results based on the elapsed time measurements should, at least, be considered to avoid mistakenly interpretations.
8) Our strategies in parallelizing the MCNP4 have proved to be correct, in which we can still maintain almost all important capabilities of MCNP4. The choice of hybrid model enables us to combine the parallelized and sequential MCNP4 as a unity, so that users do not need to maintain two separate codes.

Acknowledgments :

The first author would like to express his gratefulness to the Department of Fuel Cycle Safety Research, Tokai Research Establishment, Japan Atomic Energy Research Institute for all research facilities during one-year stay in Japan. He also acknowledges the Science and Technology Agency of Japan for the financial supports during his work under the Scientist Exchange Program. His sincere thanks are expressed to all of his colleagues in the Fuel Cycle Safety Evaluation Laboratory for their continuous supports, cooperation, discussions and their friendship. Without them, he could never enjoy his daily life in Japan. All authors would also pay special thanks to Dr. S. Kambayashi and Dr. M. Tomiyama for providing us with their computing facilities and for the fruitful discussions.

On the other hand, the execution of MCNPNFS on Monte4 showed the speedup ranging from 1.29 to 3.37 and average value of 2.84. Likewise, the parallel efficiency lowered from 32.3% to 84.3% and average value of 70.9%.

In average, discrepancies as large as 14.9% (in workstation network) and 8.3% (in Monte4) of parallel efficiency were found, when we compared the parallel efficiencies obtained by using CPU time with ones expressed in elapsed time.

7) The previous facts indicate that one may not rely solely on CPU time measurement in determining the speedup and parallel efficiency of parallelized codes. The results based on the elapsed time measurements should, at least, be considered to avoid mistakenly interpretations.
8) Our strategies in parallelizing the MCNP4 have proved to be correct, in which we can still maintain almost all important capabilities of MCNP4. The choice of hybrid model enables us to combine the parallelized and sequential MCNP4 as a unity, so that users do not need to maintain two separate codes.

Acknowledgments :

The first author would like to express his gratefulness to the Department of Fuel Cycle Safety Research, Tokai Research Establishment, Japan Atomic Energy Research Institute for all research facilities during one-year stay in Japan. He also acknowledges the Science and Technology Agency of Japan for the financial supports during his work under the Scientist Exchange Program. His sincere thanks are expressed to all of his colleagues in the Fuel Cycle Safety Evaluation Laboratory for their continuous supports, cooperation, discussions and their friendship. Without them, he could never enjoy his daily life in Japan. All authors would also pay special thanks to Dr. S. Kambayashi and Dr. M. Tomiyama for providing us with their computing facilities and for the fruitful discussions.

References :

1) Messina, P. : "The Concurrent Supercomputing Consortium : Year 1," IEEE Parallel & Distributed Technology, Vol. 1, No. 1 (February 1993).

2) Briesmeister, J. F., ed. : "MCNP - A General Monte Carlo Code for Neutron and Photon Transport, Version 3A," LA-7396-M, Rev. 2 (September 1986).

3) Briesmeister, J. F. : "MCNP3B Newsletter," Los Alamos National Laboratory (July 18, 1988)

4) Briesmeister, J. F. : "MCNP4 Newsletter," Los Alamos National Laboratory (April 4, 1991)

5) Higuchi, K., Asai, K., Akimoto, M., Shingu, S., Watari, N. and Sasaki, M. : "Effective Performance of JAERI Monte Carlo Machine," Proceeding of the Joint International Conference on Mathematical Methods and Supercomputing in Nuclear Applications, Karlsruhe, Germany (April 19-23, 1993), Vol.2, pp. 356-365.

6) Yamazaki, T., Fujisaki, M., Okuda, M., Takano, M., Masukawa. F. and Naito, Y.: "A Parallelization Study of the General Purpose Monte Carlo Code MCNP4 on a Distributed Memory Highly Parallel Computer," ibid., pp. 374-383.

7) Schmitz, F., Fischer, U. : "MCNP4, a Parallel Monte Carlo Implementation on a Workstation Network," ibid., pp. 384-394.

8) Yong, M., Chiaramonte, J. : "Parallel Monte Carlo on a Workstation Network," Transactions of the American Nuclear Society, Vol. 66, pp. 280-281 (1992)

9) McKinney, G.W., West, J.T., Whittaker, C.C. : "Multiprocessing MCNP on an IBM RS/6000 Cluster," Sixth Annual Inel Computing Symposium, Idaho Falls, Idaho, USA (September 21-24, 1992).

10) Whalen, D.J., Hollowell, D.E., Hendricks, J.S. : "MCNP : Photon Benchmark Problems," LA-12196 (September 1991).

11) Whalen, D.J., Cardon, D.A., Uhle, J.L., Hendricks, J.S. : "MCNP : Neutron Benchmark Problems," LA-12212 (November 1991).

12) Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., Sunderam, V. : "PVM 3.0 User's Guide and Reference Manual," ORNL/TM-12187 (February 1993).

13) "Express Fortran, User's Guide, Version 3.0," Para Soft Corporation, Pasadena, USA (1990).

14) Kuck, D.J. : "Parallel Processing of Ordinary Programs," Advance in Computers, Vol. 15, pp. 119-179, Academic Press, New York (1976).

15) Fleckenstein, C.J., Gill, D.H., Hemmendinger, D., McCreary, C.L., McGregor, J.D., Pargas, R.P., Riehl, A.M. and Wallentine, V. : "Multiprocessing," Advances in Computers, Vol. 35, pp. 255-324, Academic Press (1992).

16) "SunOS Reference Manual, Vol. I," Fujitsu Company (1990).

17) "Sun FORTRAN Reference Guide," Sun Microsystems, Inc. (1991).

18) "CONVEX FORTRAN Guide," Convex Computer Corporation (1991).

19) "FORTRAN 77/m4, Programming Guides," NEC (1993).

Appendices :

A.1 List of Modified and New Subroutines :

In the following, it will be listed all subroutines that are modified as well as new subroutines that are used in MCNPNFS. The host program and node program will be listed separately. The same subroutine names that are listed in both programs **do not** mean that they are identical. The only exceptions are subroutine fast_real.f, delete_file.f, RUNTPR.f and GETLDD.f, which are exactly the same in both programs.

1) Host Program of MCNPNFS :

a) Modified Subroutines : (8 in total)

EXEMES.f	GETLDD.f	MCNP.f	MCRUN.f
OUTPUT.f	RUNTPR.f	SFILES.f	TPEFIL.f

b) New Subroutines : (3 in total)

delete_file.f	fast_real.f	load_node.f

2) Node Program of MCNPNFS :

a) Modified Subroutines : (21 in total)

DXTRAN.f	EXEMES.f	EXPIRE.f	GETEXM.f
GETLDD.f	IMCN.f	MCNP.f	MCRUN.f
MCTALW.f	OUTPUT.f	RUNTPR.f	SFILES.f
SUFWRT.f	SUMARY.f	TALLYD.f	TPEFIL.f
TRNSPT.f	UFILES.f	UNIQUE.f	VTASK.f
WRWSSA.f			

b) New Subroutines : (12 in total)

assign_file.f	delete_file.f	fast_real.f	get_seed.f
initialize.f	nodes_quit.f	receive_master.f	receive_slave.f
reset_value.f	send_master.f	send_slave.f	v_task.f

Some differences exist in the MCNPNFS code used in workstations and Monte4 as follows:

1) One statement line in RUNTPE .f.

2) Some major modifications in load_node.f. It is so modified to meet with the facts that Monte4 has already multiple processors, so that it does not need to specify the different hostname for each processor. These modifications will ease the user to executing MCNPNFS on Monte4.

3) Subroutine MCTAL and PLOT may not be invoked in Monte4.

A.2. Programming Environment :

1) Workstation Network :

During the program development of MCNPNFS, we have used SUN Workstations (i.e., SUNSparc 10 or SUNSparc 2) that run under SUN-OS version 4.1.1 Rev. B. The compiler is SUN FORTRAN 1.4, that fully supports the FORTRAN 77 standard and provides with many extensions. The following additional functions (which are provided as run time library routines in *libF77.a*) are used in MCNPNFS: ***iargc***, ***rename***, ***lnblnk***, ***system*** and ***itime***.

Also, two extensions in SUN FORTRAN have been used to enable to open a sequential file and immediately put the file pointer to the end of file, so that the new entries can be appended to old file's contents. They are as follows: *access* = *'append'* and *fileopt* = *'eof'*. By using them, we can respectively manage the file OUTP and RUNTPE in a proper way.

We have performed the compiling and linking processes separately by using the following options and batch files.

- Contents of the batch file for compiling MCNPNFS source files:

```
for i
do
   f77   -Bstatic -O3 -misalign -dalign -cg89 -Nn6000 -Nq6000 -Ns6000 -Nx6000 \
         $FORTHOME/SC1.0/misalign.il $FORTHOME/SC1.)/cg89/libm.il \
         -I$GKSDIR/../../include -I$OPENWINHOME/include \
         -c $i
done
```

- Contents of the batch file used to link MCNPNFS object files:

```
f77  -Bstatic -O3 -misalign -dalign -cg89 -Nn6000 -Nq6000 -Ns6000 -Nx6000 \
     *.o  -o MCNPNFS
     -I$OPENWINHOME/lib  -I$GKSDIR/.. \
     -lgks77 -lgks -lxview -lolgx -lX11 -lX11 -lmle -lm -libmil
```

2) Monte4 :

The source files used in workstation are directly transferred to Monte4. Here, FORTRAN 77/SX has been used to compile the MCNPNFS. However, we also have to replace the extension *fileopt = 'eof'* simply with *access = 'append'*, because FORTRAN 77/SX compiler does not support the former extension. Other modifications were also made to ease user in preparing the configuration file.

The following batch files are used to compile and link the source file and object files respectively.

- Contents of the batch file for compile the source files in Monte4:

```
for i
do
f77m4 -Nv -L f=fort.comp summary b -float1 -NO -NW -c $i
done
```

- Contents of the batch file used to link all object files in Monte4:

```
f77m4 -Nv -L f=fort.comp summary b -float1 -NO -NW -o MCNPNFS *.o
```

A.3. User's Manual :

This appendix will describe some guidelines to executing MCNPNFS. Even though it is still far from completion, the authors believe that this section can provide users with enough information on how to execute MCNPNFS properly.

MCNPNFS maintain all capabilities of MCNP4 on receiving execution line message. Whatever needed by MCNP4 are also required by MCNPNFS. For example, input files INP,

RUNTPE and cross-section file must reside on the current directory, physically or logically. However, the output file OUTP and RUNTPE will automatically be written on the shared file system. Later we will explain more about it.

Preliminary Steps :

Make sure that the following guidelines are already confirmed and performed properly. Ignoring one of them may lead MCNPNFS to disaster.

1) Confirm that shared file system does exist and is accessible for read/write action to all the executing processors.
2) All remote processors must be accessible to users, meaning that they must have access permission to them. The UNIX command ***rsh*** may be used for this purpose.
3) Be sure that in the directory where the shared files reside on there are no files named para_RANDOM, para_FATAL_ERROR, para_DATA_POOL, para_OUTPxxxx (xxxx denotes, e.g., 0001, 0002, etc.), para_SLAVExxxx, para_NEWxxxx and para_OLDxxxx. Those files will automatically be deleted, prior to and after the execution of MCNPNFS. In normal termination of any MCNPNFS executions, they are transparent to users.
4) Make sure that no other users are using MCNPNFS on the same directory you are using. MCNPNFS will always produce the same filenames mentioned above, so that it is impossible to execute it from multiple users using identical directory. It also means that multiple calling of MCNPNFS on the same directory are not allowed.

Invocation of MCNPNFS :

MCNPNFS can be called in a similar way as MCNP4, as follows:

MCNPNFS INP=xxxx RUNTPE=yyyy

By default, it will then execute the MCNPNFS in parallel mode (homogeneous mode). However, if one executes the code as follows:

MCNPNFS INP=xxxx RUNTPE=yyyy S

Then, the sequential mode is activated, i.e., one is running the MCNPNFS exactly as MCNP4. By specifying option H, instead of S, one can enjoy the capability of MCNPNFS running concurrently on multiple processors in a heterogeneous system, such as using workstation network and Monte4 simultaneously.

Other execution messages are treated exactly the same way as in MCNP4.

Configuration File :

Except in the sequential mode, MCNPNFS will always need a special file, called *configuration file*, in which all informations about the executing processors are specified.

The file has, more or less, properties similar to UNIX batch file. Any lines that have the character # as their first character (not necessarily at the first column of the lines) will be ignored. They can then be used as comment lines. Users may insert as many comment lines as they like, anywhere, as long as they are written with # as the first character. The comments may also be appended, not embedded, after the last data item in a line.

Every data line contains many data items and must be written orderly, which strongly depends on the computer system that is being used. Any data item may be started anywhere in the line and separated from other data items with minimum one blank character. MCNPNFS will check the validity of any data specified in the file. Some error will be treated as fatal error, in which MCNPNFS will terminate immediately without loading the node program. Other error may still be tolerated, and default value is used, instead.

A) Homogeneous system :

A computer network comprising workstations of different model, but all of them have exactly the same external data format, are treated as homogeneous system. In this system, the configuration file named **mcnpHOMOGEN.cfg** has the following structure of data lines.

First data line: nodes ncpu, where nodes denotes the number of nodes (logical processors) that also means how many node programs (in total) must be loaded on the remote processors, while ncpu represents the number of physical processor that are available in the network (nodes and ncpu are of type integer). User may specify nodes equal to or greater than or less than ncpu. Whenever nodes less than ncpu, the remaining processors will be idle, i.e., not be used. However, specifying nodes greater than ncpu will cause some or all processors will execute more than one node programs. It is still allowed to do that. However, it will gain no advantage and even the performance of MCNPNFS may decrease. Normally, one specifies nodes equal to or less than ncpu.

The next data lines are exactly ncpu data lines that contain the following data items.

cpu_name login speed code path.

cpu_name is the hostname of respective executing processor. Every processor will always have unique hostname. Specifying the same cpu_name in two or more data lines is also allowed, however, only one processor (i.e., the one that is specified) will be assigned. login specifies the username who has access permission to the remote processor cpu_name. Normally, it is one's owned username that will be used. However, one may borrow other's username, when he/she does not have any access permission to the remote processor. The next data item is speed, i.e., the relative speed of respective processor. If no data is available, then one may specify the same value for all processors. speed may be specified as real or integer number. code is a string that specifies the code name, i.e., the node program. It must be given with its full pathname or its relative (to home directory) pathname. The last data item in the data line is path, a string variable that denotes the name of the shared directory. Any processor may have its own directory name, which differs from the others, but it must be point to an identical directory.

The next data line has only one data item of type real or integer, called b_fac. As already explained in Section III.2.3, this data may be used to divide whole NPS-particle histories into many batches. The bigger the value of b_fac, the larger the number of batches will be. Its default

value is one. Specifying b_fac larger than one will reduce the number of particles per batch. It is useful to reduce the time spent in one batch calculation, so that any processors may "help" any other processors which still busy in performing the long history tracking. However, it is always accompanied by other bad side effect, namely, the frequency of interprocessor communication will be higher, so that simultaneous access to the batch file by multiple processors will also occur more frequently. The latter will also lead a lower parallel efficiency. Giving a value of 2.0 to 3.0 for b_fac in a network consisting of 3 workstations may seem to be reasonable. However, one should always try first to assign b_fac as 1.0, and then modify it if necessary. By assigning value 1.0 to b_fac means that MCNPNFS will ideally balance the load.

The last data line consists of two integer data, namely, nps200 and npd1000. Their default values are 200 and 1000 respectively. Assigning higher values than the default values to them will always lead to a higher parallel efficiency. The accompanied bad side effects are:

- The computing results may differ from the results produced by sequential MCNP. It occurs, especially, when one deals with point detectors and DXTRAN spheres.
- The sequence of tally fluctuation charts may differ form the original MCNP4, when one changes the default value of npd1000. Setting npd1000 to NPS or greater will cause the tally fluctuation chart to show only one line data, i.e., only the tally data at the end of problem (NPS).

The example below will show how the file mcnpHOMOGEN.cfg looks like.

```
#  nodes      ncpu
     2          4
#  cpu_name          speed
          #  username          code                          path
    s10          xxxx     3.1   /home1/xxxx/mcnp/node     /home1/xxxx/mcnp
    sipx         xxxx     0.9   /home1/xxxx/mcnp/node     /home1/xxxx/mcnp
    s4a          xxxx     2.2   /home1/xxxx/mcnp/node     /home1/xxxx/mcnp
#  it belongs to other lab.
    c3005        zzzz     1.1   /home2/zzzz/MCNP/other    /home2/zzzz/mcnp
#   b_fac
      1.0
#    nps200    and    npd1000
      200                2000
```

The file begins with comment lines. It is a good habit to give such comment, so that user may not make any error when he/she wants to modify the file in the later time. The second line is the first data line that indicates that user wants to run MCNPNFS on two processors. However, there are 4 processors available in the network. In this case, MCNPNFS will be executed on processor s10 and sipx, respectively. The other two processors will simply be ignored. The line number 3 and 4 show another style how to insert the comment lines. The next three lines specify any information on the respective processors. Notice that the relative speed does not to be normalized. Here, the node program is called node, which reside on the directory /home1/xxxx/mcnp. This directory will also be used as the shared directory by MCNPNFS, because path is so assigned. It is also allowed to insert the comment line as shown on the eighth line of that example file. The next data line shows that the processor c3005 can only be access by zzzz, from whom user xxxx borrows his username. There, the node program is called other that seems to be separately compiled, because c3005 has its own compiler that suits to this computer only. The program other resides on directory /home2/zzzz/MCNP that is not known by other processor nor can be accessed by them. The one thing that must be met in this case is that both program node and other must be from the same source code. However any modifications are also allowed. The other thing is that both codes must produce exactly the same binary file, even though they compiled with different compilers. Notice that processor c3005 assigns the shared directory as /home2/zzzz/mcnp. This can be accepted as long as both /home2/zzzz/mcnp and /home1/xxxx/mcnp represent the same logical file directory. It may happen, for example, when s4a exports it file system, namely, /home1 to all other processors. However, the processor c3005 seems to mount only a part of the file system (i.e., /home1/xxxx/mcnp) on its own file system named /home2/zzzz/mcnp. The rest data lines contained in the file are already self explained.

B) Heterogeneous system:

Executing MCNPNFS on a heterogeneous computer system may only be performed if the system has capability to convert their internal data format to a standard external data format (such as IEEE format), automatically. If the automatic data conversion can be performed simply by setting certain environment variables, then MCNPNFS could be executed properly. Monte4 has such capability. By setting the environment variable F_RECLUNIT to BYTE and specifying F_UFMTIEEE to any I/O unit numbers, Monte4 will read as well as write any files in a way conforms to IEEE standard format.

In this system, MCNPNFS will read a configuration file named **mcnpHETERO.cfg**. The properties and structure of this file are very similar to mcnpHOMOGEN.cfg, except, on has to specify code as the name of a batch file, instead of the name of the node program. The example below will show more clearly how to prepare it.

```
# nodes      ncpu
    2          4
# cpu_name          speed
        # username        batch file                    path
  s10          xxxx  3.1  /home1/xxxx/mcnp/ws.sh    /home1/xxxx/mcnp
  sipx         xxxx  0.9  /home1/xxxx/mcnp/ws.sh    /home1/xxxx/mcnp
  s4a          xxxx  2.2  /home1/xxxx/mcnp/ws.sh    /home1/xxxx/mcnp
# it belongs to other lab.
  monte4       zzzz  4.1  /home2/zzzz/mcnp/m4.sh    /home2/zzzz/mcnp
#  b_fac
    1.0
#   nps200   and   npd1000
    200              2000
```

In the file above, it is assumed that 3 processors, i.e., s10, sipx and s4a are having the same binary format and may access the shared directory /home1/xxxx/mcnp. Therefore, one may use the same batch file ws.sh for all the 3 processors. However, the processor monte4 has different internal binary data format, so it will read the batch file m4.sh. This file resides on the directory /home2/zzzz/mcnp that is accidentally also the shared directory used by all processors, i.e., it is identical to directory /home1/xxxx/mcnp.

It is understandable that one has also to prepare the batch file needed by the node program. This file is really a UNIX batch file, so all UNIX conventions are valid. All environment variable setup must be written first. The last line must be a command line that calls the node program. The general form of the batch file can be depicted as follows.

```
<set up all necessary environment variables >
code    path
```

code and path have the same interpretation as in the case of homogeneous system.

By using the case shown in the last example above, one may create the following batch files. For the first three processors (i.e., s10, s4a and sipx) user needs only to create one batch file, named ws.sh. It is assumed that their data format does not need to be converted to other format, so one may simply write the batch file ws.sh as:

```
/home1/xxxx/mcnp/node   /home1/xxxx/mcnp
```

Which means that the node program will be executed by the remote processor and the argument /home1/xxxx/mcnp (i.e., the shared directory) will be passed to the node program.

In the case of processor monte4, one may write the following batch file m4.sh.

```
setenv F_RECLUNIT BYTE
setenv  F_UFMTIEEE 31,32,33,34,35,38,41,42,44,45,46,60,99
/home2/zzzz/XXXX/other   /home2/zzzz/mcnp
```

The first two command lines are used to set the environment variables needed in the automatic data format conversions. The numbers are the I/O units used in program MCNPNFS. The third command line simply orders the processor monte4 to execute the node program, named other, and passes the argument /home2/xxxx/mcnp to it.

C) Monte4 :

To execute MCNPNFS on Monte4, one may also choose the heterogeneous mode, i.e., by invoking the MCNPNFS as: MCNPNFS INP=xxxx RUNTPE=yyyy H. The configuration file

mcnpHETERO.cfg must be prepared and its structure is exactly the same as the one used in workstation.

However, MCNPNFS used in Monte4 will require a different configuration file, when one executes it in homogeneous mode (default in MCNPNFS), namely, **mcnpMONTE4.cfg**. The structure is much simpler than the file mcnpHOMOGEN.cfg, because one does not have to specify many parameters. The file mcnpMONTE4.cfg has the following form:

```
nodes
code
b_fac
nps200    npd1000
```

Killing the node program of MCNPNFS :

MCNPNFS, in the parallel mode, does not support TTY interrupt anymore. Also, they are actually running in background. Those make it impossible to stop the processes by normal action, when some trouble has occurred. When necessary, one may kill the MCNPNFS by using UNIX command ***kill***. For that, one have to know first the process number of MCNPNFS. It may be done by using UNIX command ***ps -aux***. Find out what processes number it has and then kill it by ***kill -9 ProcNum***, where ProcNum denotes the process number.