OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

ORNL/TM-2010/176

# An Incremental Parallelization Approach Applied to the ORNL/NRC FAVOR Code

**August 2010**

**Prepared by**
**Srikanth B. Yoginath**
**Kalyan S. Perumalla**

UT-BATTELLE
ORNL-27 (4-00)

ORNL/TM-2010/176

Computational Sciences and Engineering Division

# AN INCREMENTAL PARALLELIZATION APPROACH APPLIED TO THE ORNL/NRC FAVOR CODE

Srikanth B. Yoginath
Kalyan S. Perumalla
Paul T. Williams
Richard B. Bass

Date Published: August 2010

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
Managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

## CONTENTS

## LIST OF FIGURES

**Figure** **Page**

## ABSTRACT

Parallelizing a domain-specific production code with thousands of lines that is developed over several years is a daunting task. Also, throwing away the existing serial code completely to design a parallel algorithm from scratch is not always a viable solution. Ideally, one wishes to morph the serial code to make it compute in parallel, so that much of knowledge built over years in the form of serial code is retained and the performance gain due to parallel computing is also achieved. Hence, the parallelization task of the production code must be very conservatively approached.

With such a guiding principle in parallelizing a complex, production-version of a serial code, we start with the functional serial execution and attempt a series of parallelization steps designed to uncover the issues and problems that arise when the serial code is incrementally transformed into a parallel code. Addressing each of the arising issues individually, we incrementally transform the serial code into a functional parallel code that retains the correctness of the serial code yet executes in parallel, ultimately delivering significant reduction in run time. Since the issues are incrementally addressed, we refer to this strategy as an Incremental Parallelization Approach (IPA). We demonstrate the applicability of IPA, by parallelizing over 25,000 lines of Probabilistic Fracture Mechanics (PFM) module code of Fracture Analysis of Vessels Oak Ridge (FAVOR) code that was developed for the Nuclear Regulatory Commission (NRC) by the HSST program at Oak Ridge. As a result of applying the IPA methodology, we ultimately reduced the run time of FAVOR PFM module from several hours to only a few minutes, without any loss of accuracy in the computed result. In this article, we discuss our experience gained in the effort in parallelizing the FAVOR's PFM module using our IPA methodology.

# 1. INTRODUCTION

In general, a team of experts develops domain-specific (serial) computer code over years, often incurring enormous investments of time and effort to bring it to the production level. When such a production serial code is stressed, either to overcome limitations in handling large input scenarios or improve accuracy, often, parallel execution is the only effective solution. However, parallel programming, with its own set of challenges and idiosyncrasies, makes the task of transforming thousands of lines of production code into an efficient parallel code hard and challenging.

Fracture Analysis of Vessels – Oak Ridge (FAVOR), is one such serial code that was developed for the Nuclear Regulatory Commission (NRC) by the HSST program at Oak Ridge.[1,2] It has three modules, (a) Deterministic Load Generator module (FAVLOAD), (b) Monte-Carlo PFM module (FAVPFM) (c) Post Processor module (FAVPOST). Of the three modules, the FAVPFM module requires a very long execution time (usually days) in order to execute the number of scenarios to effectively cover the parameter domain of interest. Parallel execution of the FAVPFM can help dramatically reduce the run time of the FAVOR application. However, a major challenge in transforming the serial FAVOR code into a parallel FAVOR system is posed by the large size of the code that is over 25,000 lines long, making it practically and prohibitively expensive. Hence, an approach is needed to be able to parallelize the serial code, and performing such a conversion in a time-efficient manner is of significant importance.

As an alternative to the cost of understanding the serial code and re-implementing it, an alternative approach is explored here for the parallelization that achieves the goal and also retains the same confidence levels in the correctness of the output results as those from its serial counter part. In this report, we propose the Incremental Parallelization Approach (IPA) to achieve this task in several of iterative steps. We document our experience in applying the IPA for parallelizing the serial PFM module of FAVOR and the final performance improvement we obtained.

The Monte-Carlo simulation algorithm that underlies the FAVOR system is widely known to be an embarrassingly parallel problem[3,4,5] and thus easily parallelizable if one started development from scratch. The work reported here explores the challenges involved when a Monte Carlo code such as the FAVOR system is to be transformed into the parallel execution while the system is handled only as a black box.

In the following sub-sections, we introduce the IPA methodology and the FAVOR system. The application of IPA for the parallelization of FAVOR (specifically, the serial FAVPFM module of FAVOR) is discussed in Sect. 2–4. This is followed by the performance evaluation study of the parallel FAVOR (FAVPFM) in Sect. 5. The report is summarized and concluded in Sect. 6.

## 1.1 INCREMENTAL PARALLELIZATION APPROACH

In the IPA approach, we first execute multiple instances of the same unchanged serial code in parallel to begin to uncover the interdependencies of the parallel computing processes. The interdependencies among the processes executing in parallel

(a) can silently disappear resulting in the erroneous computation, or
(b) can blatantly fail resulting in the abrupt termination of the execution process, or
(c) can freeze up the execution process due to a deadlock condition.

These are the only three ways by which the interdependencies non-existent in the serial code can surface when the same unchanged serial code is evaluated in parallel. If each of these failures in this failure space were to be resolved incrementally, then the resulting algorithm that the code manifests must be the required parallel algorithm. This algorithm will not only compute in parallel but will also replicate the results of the serial code exactly to the machine precision.

In general, both types of failures (b) and (c) are more apparent, easy to detect and hence could be relatively easy to resolve after detection. On the other hand the failure (a) is subtler and could be hard to detect and overcome. While the detection of errors will be evident when the results from the serial and parallel runs are compared, the actual point of error generation is hard to track down. In our IPA procedure we aim to eliminate the easily detectable and observable failures early on and address the subtler failures later, i.e. after we resolve the apparent ones. Figure 1 gives the flow chart of the IPA process.

The knowledge on the characteristic of parallelization of the resulting algorithm obtained at the end of from the IPA procedure is essential to resolving the failures and errors. For example, in the case of FAVOR, we are aware that the final parallel algorithm may be viewed as embarrassingly parallel and hence the processors will be computing in complete independence of one another. Also this knowledge helps us to overcome the incorrect execution caused by dependencies on initialization by duplicating the initialization routine execution for every processor. This knowledge-based resolution of failures is extremely important because it ensures that the changes introduced into the parallel algorithm to overcome the failures wouldn't perpetuate additional failures.

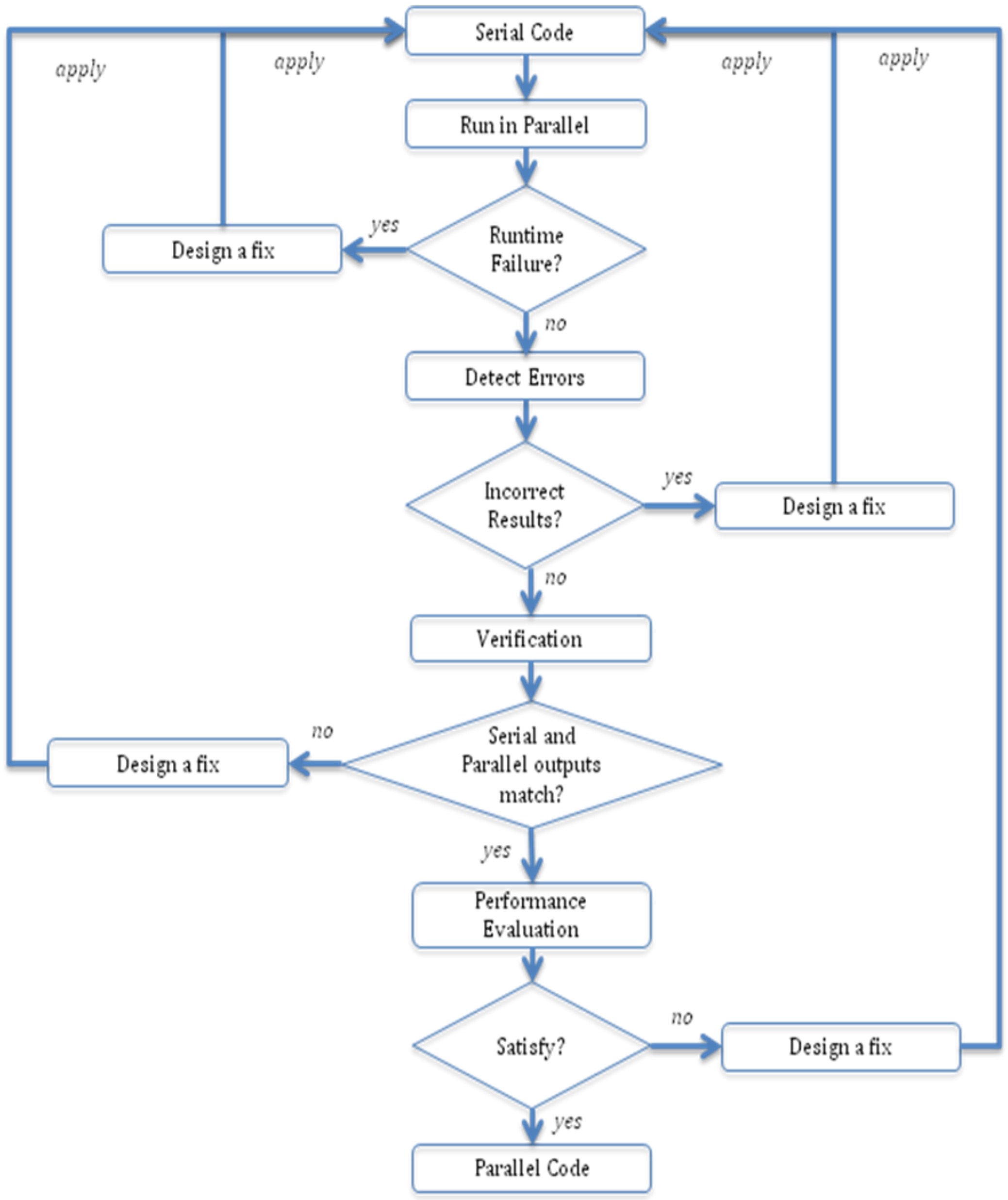

**Fig. 1. Incremental Parallelization Approach (IPA) flow chart.**

## 1.2 FRACTURE ANALYSIS OF VESSELS – OAK RIDGE (FAVOR) CODE

FAVLOAD module accepts input data containing multiple thermal-hydraulic transients, and, for each transient, it performs deterministic calculations to produce a load-definition input file for FAVPFM.

The PFM module in FAVOR is based on the application of Monte-Carlo techniques in which the deterministic fracture analyses are performed on a large number of stochastically generated RPV (Reactor Pressure Vessel) trials and realizations.

Each vessel realization containing a specified number of flaws is analyzed to determine the conditional probability of initiation (CPI) and the conditional probability for failure (CPF) for an RPV challenged by thermal hydraulic transient at selected time in vessel's operating history. The Monte-Carlo method involves sampling of appropriate probability distributions to simulate many possible combinations of flaw geometry and RPV (Reactor Pressure Vessel) material embrittlement, all exposed to same transient loading conditions.

The Post Processor module combines three primary results to generate discrete distributions of the frequency of vessel initiation and frequency of vessel failure. The results that are combined are the distribution of the transient initiating frequencies obtained from probabilistic risk assessment studies, the values of conditional probability of fracture (contained in the FAVPFM-generated matrix *PFMI*), and the values of the conditional probability of vessel failure (contained in the FAVPFM-generated matrix *PFMF*).

The PFM module is the part that needs to be parallelized, since it is the most time-consuming part of the FAVOR code. The PFM module is implemented in FORTRAN-90 that results in an executable after compilation. The entire module is implemented in 25,880 lines of code and it uses the Monte-Carlo approach. The PFM executable interactively takes input files from the command prompt *during* execution. The following 5 files are taken as input by the FAVPFM executable:

(a) Output of FAVLoad
(b) FAVPFM input file
(c) Flaw characterization file for surface-breaking flaws applicable to weld and plate regions (default=S.DAT)
(d) Flaw characterization file for embedded flaws in weld region (default=W.DAT)
(e) Flaw characterization file for embedded flaws in plate region (default=P.DAT).

## 1.3 PARALLEL FAVPFM REALIZATION USING IPA

Parallelization of FAVOR code is a challenging task due to the complexity of the algorithm and the instantiation characteristics of the algorithm in the code. One of the simplest parallelization approaches can be applied to exploit the Monte Carlo structure of execution that is inherent in FAVOR. However, the software structure is not readily amenable to parallel execution, making it necessary to incorporate modifications to the code. Since the application is complex, its original modifications must be made extremely carefully, such that its verified and validated status must be retained to the extent possible. Refactoring approximately twenty six thousand lines of serial FAVOR code in its entirety to design an alternative parallel algorithm seems impractical. Hence, the IPA is used.

The parallelization was carried out in four steps listed below.

- Step 1: Pseudo-parallel run of serial code - To be able to run the serial code in parallel using MPI, without partitioning the models to processors, such that each of the parallel processes duplicates all work, but compute the exact same serial result at every processor.
- Step 2: Task partitioning in pseudo-parallel code - To be able to partition the tasks across the parallel application that is now enabled to start running in parallel as a result of Step 1
- Step 3: Verification of parallel code – Verify that the parallel results exactly match the corresponding serial results.
- Step 4: Performance evaluation – to empirically determine the speed gain obtained from the whole parallelization exercise.

Each of these steps is discussed separately in detail in Sect. 2–5, respectively.

## 2. PSEUDO-PARALLEL RUN OF SERIAL CODE

If we are able to launch the serial program as a parallel program (sans communication dependencies, for a first cut), that would constitute a first, necessary step towards parallelization. The necessary conditions are satisfied if such a first-cut parallel execution completes without runtime errors and, if each of the process ranks were to produce exactly same results as the serial run. This offers the confidence of having eliminated global dependencies and other necessary conditions (e.g., file name overlaps, and read-write conflicts). This task expects that the serial procedure can be partitioned into multiple completely independent tasks, which when executed to yield the same result as the serial program.

This step is completely based on the assumption that there exists a parallel algorithm that can be applied for its parallelization and algorithm in this case is task-parallelism. Since, we are aware that the PFM code is Monte-Carlo based and also know that task-parallel algorithm for parallelization can be applied to Monte-Carlo based applications, we carry out this step

### 2.1 REALIZATION

As mentioned in Sect. 1, the PFM executable takes the input filenames interactively. The first modification that we performed to the serial code was the removal of the interactive input capability. For this purpose we hard coded the input filenames in the code. Also, the serial code was MPI-enabled and was compiled using Open-MPI with its wrapper for FORTRAN compiler. Here MPI-enabled means that MPI statements were added to the serial code enable the process to run as an MPI process.

```
      :
forrtl: No such file or directory
forrtl: severe (28): CLOSE error, unit 16, file "Unknown"
Image              PC                Routine            Line        Source
mfavpfm            00000000005565F6  Unknown               Unknown  Unknown
:
libc.so.6          0000002A9699C40B  Unknown               Unknown  Unknown
mfavpfm            000000000040F8EA  Unknown               Unknown  Unknown
--------------------------------------------------------------------------
mpiexec has exited due to process rank 1 with PID 11432 on
node b07n013.oic.ornl.gov exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpiexec (as reported here).
--------------------------------------------------------------------------
```

**Fig. 2. Excerpt of error message from failed pseudo-parallel run.**

The idea here was to run the serial code in parallel as it were, without any modification and see where it fails; once we know the failure point, we trace back the reason for failure and fix it or come up with a strategy that would efficiently circumvent the problem at the source of the runtime error. Throughout the process of parallelization, except during the performance studies, we have used two MPI processes for parallel runs for simplicity and ease in debugging.

All most all the errors that we encountered, when we ran the MPI enabled serial code in parallel were related to the handling of file operations by the parallel program. An excerpt of the error is shown in Fig. 2.

```
                    ***********************************************
                    * Results for running averages of cpi and cpf *
                    *   See cpi_history.out and cpf_history.out   *
                    *      for the same data in a text file.      *
                    ***********************************************
         |------------------------------||--------------------------------|
         |   running average of cpi     ||    running average of cpf      |
   ntrial|----------------------------- ||--------------------------------|
         |     1         2         3    ||     1         2         3      |
         |------------------------------||--------------------------------|
      1  | 0.000E+00 7.486E-06          || 0.000E+00 0.000E+00            |
      2  | 0.000E+00 3.743E-06          || 0.000E+00 3.000E+00            |
      3  | 0.000E+00 2.495E-06          || 0.000E+00 0.000E+00            |
      :        :        :                      :          :
     44  | 4.003E-04 4.077E-05          || 0.000E+00 0.000E+00            |
     45  | 3.914E-04 3.987E-05          || 0.000E+00 0.000E+00            |
     46  | 3.829E-04 3.900E-05          || 0.000E+00 0.000E+00            |

     97  | 1.821E-04 1.924E-05          || 0.000E+00 0.000E+00            |
     98  | 1.803E-04 1.905E-05          || 0.000E+00 0.000E+00            |
     99  | 1.784E-04 1.902E-05          || 0.000E+00 0.000E+00            |
    100  | 1.766E-04 1.883E-05          || 0.000E+00 0.000E+00            |
      :        :        :                      :          :
                              COMPLETING PFM ANALYSIS

               Creating a FAVPFM binary restart file: restart.bin
               Time Stamp -- DATE: 14-May-2010  TIME: 14:08:05
               RANDOM NUMBER GENERATOR SEEDS:      1270544027     2101961695

                             GENERATING OUTPUT REPORTS
** Normal Termination **
```

**Fig. 3.  Excerpt of serial run output of the FAVPFM module.**

Since, many processes can read a single file at the same time with out any problem and our parallel experimentation platform's file system was based on Networked File System (NFS), the concurrent reading of data by all processors from input files completed without runtime errors. However, the files, which were opened in write mode, like the output files, error files, restart files (used for check-pointing) were the points of failure. These file errors were overcome by making each of the process ranks write into their own files. This involved identification of the contentious files and addition of very few lines of code to enable the each of the processes running in parallel to create/write/delete their own file.

After the fixes we were able to run the serial code completely in parallel. Figures 3–4 show the excerpts of output from a serial code and the parallel code respectively. Comparing the results, it is observed that both serial and parallel runs print exactly the same results. Further, from Fig. 4 we also see that both the process ranks involved in the parallel run print exactly the same result.

In this step, we converted the serial code into independently running parallel code, where in each of the parallel processes print out the exact same results. Hence, by doing this we thus can be sure of having eliminated any global dependencies in the parallel code.

```
                    ***********************************************
                    * Results for running averages of cpi and cpf *
                    *   See cpi_history.out and cpf_history.out   *
                    *      for the same data in a text file.      *
                    ***********************************************
          |------------------------------||--------------------------------|
          |   running average of cpi     ||    running average of cpf      |
    ntrial|----------------------------- ||--------------------------------|
          |     1         2         3    ||     1         2         3      |
          |------------------------------||--------------------------------|
                CREATING PROBABILITY DISTRIBUTIONS FOR FLAWS

                         **************************
                         * BEGINNING PFM ANALYSIS *
                         **************************


                    ***********************************************
                    * Results for running averages of cpi and cpf *
                    *   See cpi_history.out and cpf_history.out   *
                    *      for the same data in a text file.      *
                    ***********************************************
          |------------------------------||--------------------------------|
          |   running average of cpi     ||    running average of cpf      |
    ntrial|----------------------------- ||--------------------------------|
          |     1         2         3    ||     1         2         3      |
          |------------------------------||--------------------------------|
       1  | 0.000E+00 7.486E-06          || 0.000E+00 0.000E+00            |
       1  | 0.000E+00 7.486E-06          || 0.000E+00 0.000E+00            |
       2  | 0.000E+00 3.743E-06          || 0.000E+00 0.000E+00            |
       2  | 0.000E+00 3.743E-06          || 0.000E+00 0.000E+00            |
       :        :        :                     :          :                |
      45  | 3.914E-04 3.987E-05          || 0.000E+00 0.000E+00            |
      45  | 3.914E-04 3.987E-05          || 0.000E+00 0.000E+00            |
      46  | 3.829E-04 3.900E-05          || 0.000E+00 0.000E+00            |
      46  | 3.829E-04 3.900E-05          || 0.000E+00 0.000E+00            |
       :        :        :                     :          :                |
      99  | 1.784E-04 1.902E-05          || 0.000E+00 0.000E+00            |
      99  | 1.784E-04 1.902E-05          || 0.000E+00 0.000E+00            |
     100  | 1.766E-04 1.883E-05          || 0.000E+00 0.000E+00            |

                              COMPLETING PFM ANALYSIS

               Creating a FAVPFM binary restart file: restart.bin
               Time Stamp -- DATE: 13-May-2010  TIME: 12:21:14
               RANDOM NUMBER GENERATOR SEEDS:      1270544027     2101961695

                             GENERATING OUTPUT REPORTS
     100  | 1.766E-04 1.883E-05          || 0.000E+00 0.000E+00            |

                              COMPLETING PFM ANALYSIS

               Creating a FAVPFM binary restart file: restart.bin
               Time Stamp -- DATE: 13-May-2010  TIME: 12:21:15
               RANDOM NUMBER GENERATOR SEEDS:      1270544027     2101961695

                             GENERATING OUTPUT REPORTS
 Ending run at: Thu May 13 12:21:15 EDT 2010
Epilogue Initiated
Removing /scratch/90940.b15l01.oic.ornl.gov on node(s): b08n048
Floaters flushed on node(s):
Epilogue Complete
```

**Fig. 4. Excerpt of Pseudo-parallel run output of FAVPFM.**

## 3. TASK PARTITIONING IN PSEUDO-PARALLEL CODE

In step 1, we were able to run the serial code in parallel, completely independent of one another and were also able to get each of the MPI processes to compute the exact same results as that of the serial run. In this step, we identify the independent tasks within PFM and perform *task parallel* execution. To do this we need to look into the structure of the code and see where task parallelism can be applied. The most time-consuming part of the PFM subroutine in FAVPFM follows the algorithm in Fig. 5.

```
VESSELS (1:NTRIAL)
      {read in FLAWGROUP files unique for the TRIAL or VESSEL ID}
      FLAWS (1:NUMFLW)
              TRANSIENTS (1:MTRAN)
                      TIME STEPS (1:NTIMES)
                              :
                              :
                      EXHAUST TIME STEPS
              EXHAUST TRANSIENTS
      EXHAUST FLAWS
EXHAUST VESSELS

Ref: FAVPFM.for code – comments
```

**Fig. 5. The core algorithm structure of FAVPFM module.**

The above algorithm documented in the FAVPFM code (and learnt from subsequent discussions with the original developers of the system), revealed that the runs across the trials (the outer loop) were independent of one another. This suggests that we could run each trial as an independent task.

However the random number seed used in trials in parallel runs would differ from those in serial runs. This is because we partition the tasks across trials, which results in differing outputs of serial and parallel runs. This does compromise the correctness of the computed results and hence, we will not be able to verify the correctness of the parallel computed results with the results from the serial run, in this step.

### 3.1 REALIZATION

We use the same example scenario as in step 1 and the code is modified such that the trials are equally partitioned across multiple processes. This was achieved by adding few variables and few lines of code at the start and the end of the NTRIAL loop of the algorithm shown in Fig. 5 above.

```
10    NTRIAL = NTRIAL + 1
          :
          :
          IF (NTRIAL.GE.NSIM_TEST) THEN
          GOTO 9999
      ELSE
          GOTO 10
      ENDIF
```

**Fig. 6. Serial code fragment.**

```
      NTRIAL = START_TRIAL
10    NTRIAL = NTRIAL + 1
          :
          :
 IF ((NTRIAL.GT.END_TRIAL).OR.(NTRIAL.GE.NSIM_TEST)) THEN
          GOTO 9999
      ELSE
          GOTO 10
      ENDIF
```

**Fig. 7. Parallel code fragment.**

```
         |------------------------------||--------------------------------|
         |   running average of cpi     ||    running average of cpf      |
   ntrial|----------------------------- ||--------------------------------|
         |     1         2         3    ||     1         2         3      |
         |------------------------------||--------------------------------|
 ***** NTRIAL :           51
                CREATING PROBABILITY DISTRIBUTIONS FOR FLAWS
 NSUB_TRIALS:           50
 N_TRIALS:            0
 ISIZE:            2
 NSUB_TRIALS:           50
 START_TRIAL:            0
 RANK:           0
 NSUB_TRIALS:           50
 END_TRIAL:           49
 RANK:           0
 ******** START_TRIAL :            0
 ******** END_TRIAL :           49

                         **************************
                         * BEGINNING PFM ANALYSIS *
                         **************************


                    ***********************************************
                    * Results for running averages of cpi and cpf *
                    *   See cpi_history.out and cpf_history.out   *
                    *      for the same data in a text file.      *
                    ***********************************************
         |------------------------------||--------------------------------|
         |   running average of cpi     ||    running average of cpf      |
   ntrial|----------------------------- ||--------------------------------|
         |     1         2         3    ||     1         2         3      |
         |------------------------------||--------------------------------|
 ***** NTRIAL :            1
     51  | 3.416E-06 0.000E+00          || 0.000E+00 0.000E+00            |
 ***** NTRIAL :           52
      1  | 0.000E+00 7.486E-06          || 0.000E+00 0.000E+00            |
 ***** NTRIAL :            2
     52  | 3.350E-06 0.000E+00          || 0.000E+00 0.000E+00            |
 ***** NTRIAL :           53
      2  | 0.000E+00 3.743E-06          || 0.000E+00 0.000E+00            |
      :        :        :                      :         :
     99  | 1.169E-05 1.576E-08          || 0.000E+00 0.000E+00            |
 ***** NTRIAL :          100
     49  | 3.596E-04 3.768E-05          || 0.000E+00 0.000E+00            |
 ***** NTRIAL :           50
    100  | 1.157E-05 1.560E-08          || 0.000E+00 0.000E+00            |

                              COMPLETING PFM ANALYSIS

               Creating a FAVPFM binary restart file: restart.bin
               Time Stamp -- DATE: 14-May-2010  TIME: 14:00:40
               RANDOM NUMBER GENERATOR SEEDS:      2128235567      491493355

     50  | 3.524E-04 3.693E-05          || 0.000E+00 0.000E+00            |

                              COMPLETING PFM ANALYSIS

               Creating a FAVPFM binary restart file: restart.bin
               Time Stamp -- DATE: 14-May-2010  TIME: 14:00:40
               RANDOM NUMBER GENERATOR SEEDS:      1503797025     1823000556

                             GENERATING OUTPUT REPORTS
                             GENERATING OUTPUT REPORTS
 Ending run at: Fri May 14 14:00:40 EDT 2010
Epilogue Initiated
Removing /scratch/91240.b15l01.oic.ornl.gov on node(s): b08n031
Floaters flushed on node(s):
Epilogue Complete
```

**Fig. 8. Output excerpt from the parallel run after task partitioning.**

START_TRIAL and END_TRIAL variables were used for partitioning the trials among parallel processes and their values were calculated based on the process rank of the corresponding MPI process. Also, minor additional changes within the NTRIAL loop were done to accommodate this change.

Figure 8 shows the excerpt of the result of the parallel run. In this particular example, of the 100 trials that FAVPFM runs, 50 trials were run by MPI process with rank 0 and remaining 50 were run by MPI process with rank 1. As can be verified from Fig. 3 and Fig. 8, the first 50 trials of the parallel run, which startup with same random-seed as the serial run, produce exact same result as that of first 50 trials of the serial run. However, the last 50 trials do not match because the initial random seed with which the trial 51 starts during the parallel run is different from that of its serial counter part.

In this step we converted the parallel code, where in the parallel processes that were duplicating each other's work into a parallel program that share the work load. The sharing of workload was achieved by partitioning the trials equally among the parallel processes, after having learnt that the computations across trials are independent of one another.

## 4. VERIFICATION OF PARALLEL CODE

In Sect. 3, we noted that that the first 50 trial outputs by the process of rank 0, were exactly same as the serial run output, while the trials from 51 onto 100 (run on MPI process of rank 1) were not consistent with the output from the serial run. The variation of the seed values in the random number generator used in FAVPFM was reasoned out to be responsible for the results obtained by parallel run.

We know that any two sequence of random numbers generated are exactly same, if their corresponding initial seeds are same. If for each trial in the FAVPFM code, if we were to know the initial random seed used during the serial execution, and if we use the same random seed in a task partitioned parallel execution, then we can ensure the exact correspondence of the result from the serial and parallel runs. Hence, the knowledge of initial random seed is a must for achieving verifiably correct result during the parallel computation of the FAVPFM

### 4.1 REALIZATION

The very first task we carried out in this step is to check, how the random number generation and its subsequent usage varies across trials. If the random numbers were to be generated and used with in the conditional statements in the code, then this leads to varying number of random numbers across trials or else the number of random numbers generated will remain constant across trials. To count the number of random numbers generated in each trial, a counter was placed in the uniform random number generator function (`ranf2`). After every trial, the random number counter value along with the trial number that starts next was printed, before the counter is reset.

```
[y54@b06l01 siam]$ cat count_seq.out | grep rnd_count
      :
 rnd_count NTRIAL     33616827           2
 rnd_count NTRIAL     32624853           3
 rnd_count NTRIAL     33015309           4
      :
 rnd_count NTRIAL     33934609          48
 rnd_count NTRIAL     33828715          49
 rnd_count NTRIAL     33253161          50
 rnd_count NTRIAL     32882785          51
 rnd_count NTRIAL     33544459          52
      :
 rnd_count NTRIAL     33788905          96
 rnd_count NTRIAL     33200427          97
 rnd_count NTRIAL     32889553          98
 rnd_count NTRIAL     32671177          99
 rnd_count NTRIAL     32856421         100

 Max (rnd_count) = 35257681
```

**Fig. 9. Total number of random numbers generated in each trial.**

Figure 9 shows the excerpt of the output that counts the random numbers in each trial. The varying counter values in each trial as shown in the Fig. 9 confirms that the number of random numbers generated in each trial varies, which suggest that the random numbers were generated and used with in the conditional statements as well.

#### 4.1.1 Resolving Random Number Initialization Issue

The initial random seeds can be known before hand only if, the number of random numbers generated across each of the trials is deterministic. Hence, in the task parallel algorithm for the serial FAVPFM code obtained from step 2, we fix the number of random numbers that can be generated in each trial to a constant number, i.e., we set an upper limit to the number of random numbers that a trial can generate. This value was found empirically by counting the random numbers generated in each trial and we found it to be always less than 36 million, as observed in Fig. 9. Hence, we fixed the maximum number of random numbers that any trial could generate to 36 million and also added a condition that if any trial were to exceed this upper-bound value than the parallel execution would exit prematurely. In our implementation the number of random numbers generated per trial to (initial_seed + rank * 36,000,000), by doing this we ensured that a fixed set of random numbers are generated in each trial.

For the purpose of verification, we altered the initially used serial FAVPFM code and the task-parallel FAVPFM code obtained from step 2, so that the results could be compared. In the serial code, as we were fixing the constant number of random number generated per trial to a number is greater than the maximum of the number of random numbers generated across any trial, we end up throwing out extra random numbers at the end of each trial

In the parallel code, in addition to the throwing away of the generated random numbers at the end (as in serial code), each MPI process had to exhaust generating a known set of number of random numbers to obtain its initial random seed.

### 4.1.2 Verification Setup

To demonstrate the correctness in the execution of the parallel FAVPFM, we compare the random numbers generated at each trial during serial run with that of parallel run. This we claim as right measure because the number of random numbers generated or used, in a particular trial depends on the initial random seed and with the computations with in the each trial being same in both serial and parallel codes, the number of random numbers generated must be same. Hence, a success in verification process is claimed, if the number of random numbers used in serial and parallel runs were observed to be exactly same in each and every trial.

We adopt this measure because the general output of FAVPFM is the running average of CPF and CPI values, which print incorrect results, unless the results computed across the trials, which are distributed across the parallel processes, are used. We abstain from modifying the serial code further, since we can prove the correctness in the computation of parallel processes using an alternative means of comparing random numbers generated in each trials in serial and parallel runs, as discussed before.

### 4.1.3 Verification Result

```
      :
rnd_count NTRIAL     33292957          8
rnd_count NTRIAL     32915719          9
rnd_count NTRIAL     32902650         10
      :
rnd_count NTRIAL     33087948         25
rnd_count NTRIAL     32552058         26
      :
rnd_count NTRIAL     32704406         55
rnd_count NTRIAL     32367023         56
      :
rnd_count NTRIAL     34483647         79
rnd_count NTRIAL     33511330         80
      :
rnd_count NTRIAL     32889778         98
rnd_count NTRIAL     32671260         99
rnd_count NTRIAL     32856346        100
```

**Fig. 10. Total number of random numbers generated in each trial of the serial run.**

Figures 10–11 show the excerpts of the output from serial and parallel runs. They show that the number of random numbers generated in each trial of both serial and parallel computations exactly match each other. However, the outputs corresponding to running average of CPI and CPF, for the

MPI process rank not equal to 0, do not match (as expected). This is because the results of the trials performed on ranks 1 to M, do not take previous CPIs and CPFs into consideration while computing the running average.

```
      :
rnd_count NTRIAL     33292957          8
rnd_count NTRIAL     32915719          9
rnd_count NTRIAL     32902650         10
      :
rnd_count NTRIAL     32704406         55
rnd_count NTRIAL     32367023         56
rnd_count NTRIAL     34735077         16
rnd_count NTRIAL     33670039         17
      :
rnd_count NTRIAL     35257457         64
rnd_count NTRIAL     33087948         25
rnd_count NTRIAL     33511128         65
rnd_count NTRIAL     32552058         26
      :
rnd_count NTRIAL     34483647         79
rnd_count NTRIAL     33511330         80
rnd_count NTRIAL     32823220         89
`     :
rnd_count NTRIAL     33828984         49
rnd_count NTRIAL     33213537         90
rnd_count NTRIAL     33253350         50
      :
rnd_count NTRIAL     32889778         98
rnd_count NTRIAL     32671260         99
rnd_count NTRIAL     32856346        100
```

**Fig. 11. Total number of random numbers generated in each trial of the parallel run. The first 50 trials were run on MPI process rank-0 and remaining 50 trials were run on rank-1 process. We find that for every trial, the random number count here are exactly same as that in Fig. 10. Hence, we conclude that they are computing exactly same result.**

In this step, errors observed in the output of the task parallel system of step2 were addressed. The errors were reasoned to be originating from the disruption in the random number stream that surfaced as an artifact of task partitioning across the trials. This issue was addressed by setting an upper limit for the random numbers generated in each trial, which was found out empirically. The results from the serial runs and parallel runs were compared and were verified to be exactly same based on the random number counts.

Note that the solution of fixed number of random number generations, is to verify the correctness of the parallel algorithm, by comparing it with its serial counter part. If there is no need to maintain the same stream of random numbers across the trials, or if we could initialize each of the trials with different RNG streams, then we already have reached completion of parallelization task at this point.

## 5. PERFORMANCE EVALUATION

As we know that parallel computing code is embarrassingly parallel in nature, we expect to achieve a linear speed up.

### 5.1 HARDWARE

All the runs were carried out on Oak Ridge Institutional Clusters (OIC). They consist of a bladed architecture from Ciara Technologies (http://www.ciara-tech.com) called VXRACK. Each VXRACK contains two login nodes, three storage nodes and 80 compute nodes. Each compute node has Dual Intel 3.4GHz Xeon EM64T processors, 4GB of memory and dual Gigabit Ethernet Interconnects. All nodes run Red Hat Linux Enterprise WSv4 OS.

### 5.2 SOFTWARE

Only FAVPFM.for source file from the FAVOR9.1 source code distribution was used. The Intel® FORTRAN90 compiler was used to compile the serial code and Intel® FORTRAN90 compiler based OpenMPI v1.3.2, was used to compile the parallel code.

The FAVPFM_MPI executable named *mfavpfm* takes all the input file names as command line arguments; in addition to these inputs, it also takes the random seed file name and the maximum number of random numbers generated per trial.

```
[y54@b06l02 mpifavor]$ ./mfavpfm
USAGE: ./mfavpfm [FAVPFM.in] [FAVLOAD.out] [S.DAT] [W.DAT] [P.DAT]
[SEEDS.TXT] [MAX_RN_PER_TRIAL]
```

**Fig. 12. The parallel code usage specifics.**

### 5.3 PERFORMANCE RUNS

Figure 13 shows the runtime performance of the serial runs, the solid blue curve gives the runtime performance of serial FAVPFM for varying number of trials. The dashed red curve is the modified code that implements the throwaway of the random numbers at the end of each trial and is used for verification purpose. As is expected and can be seen from Fig. 13, their performance results closely correspond to each other.

During parallel runs, as mentioned to achieve the correctness we generate and throwaway many generated random numbers, during the startup of the parallel MPI process and during the end of each trial. This needless computation does impact performance significantly.

Figure 14 shows the runtime when 100 trials were evaluated using 100 MPI processes in parallel. In this scenario, each MPI process handles the execution of exactly one trial. As seen, the minimum time taken in the trial execution by a MPI process is little over 2 s (2.36 s) while the maximum time is close to 50 s (49.86 s). This clearly is an artifact of the generation and subsequent throw away of random numbers. This can be verified by the random number generation performance curve in Fig. 15, where we see that 48 s is used for RNG generation for 100 trials and in the throw away scheme, the 100th trial generates and throws away random numbers pertaining to the previous 99 trials, before executing the 100th trial and this explains the overhead seen in the Fig. 14.

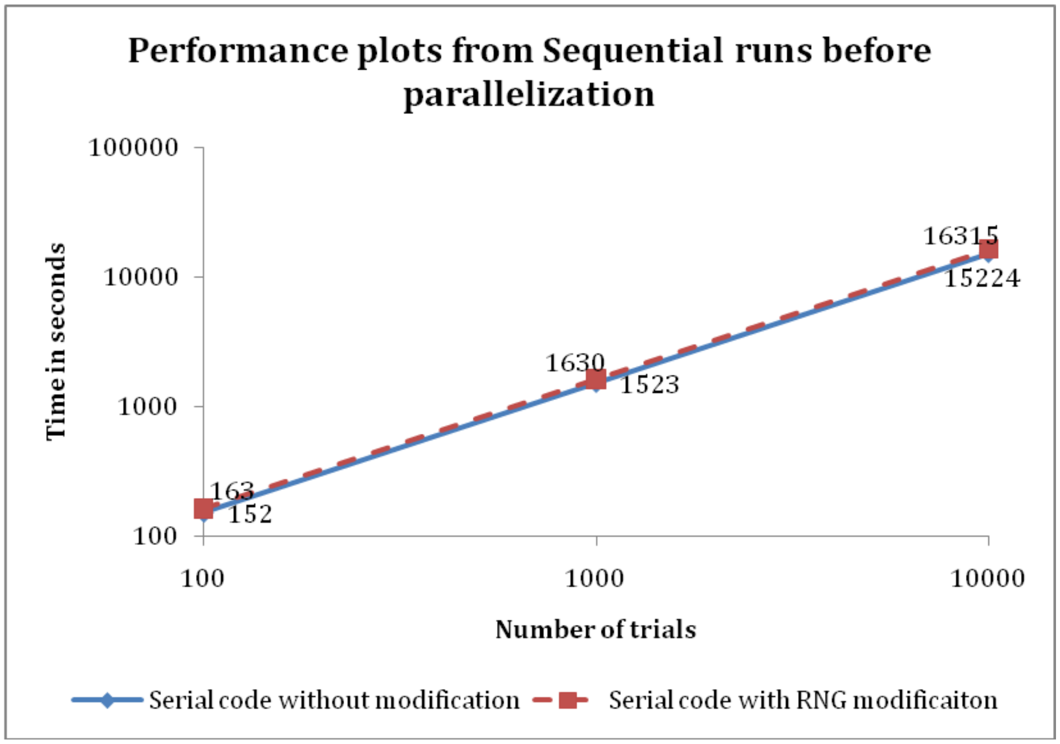

**Fig. 13. Serial run with and without random number generator modifications.**

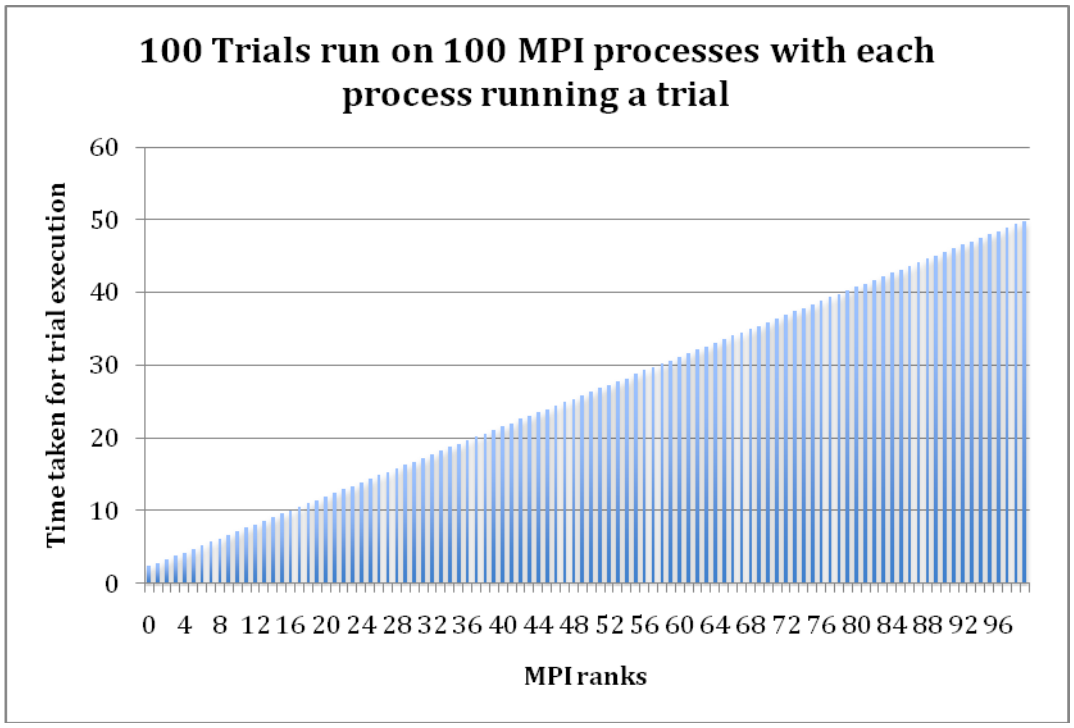

**Fig. 14. Runtime at every MPI process rank (generate and throw-away strategy).** Task partitioning is done such that the lower rank MPI process gets the lower trial number and higher rank MPI process gets higher trial number. Hence we see MPI process handling higher trial number needs more runtime, since most of its time is used in generating and throwing away the unwanted random numbers.

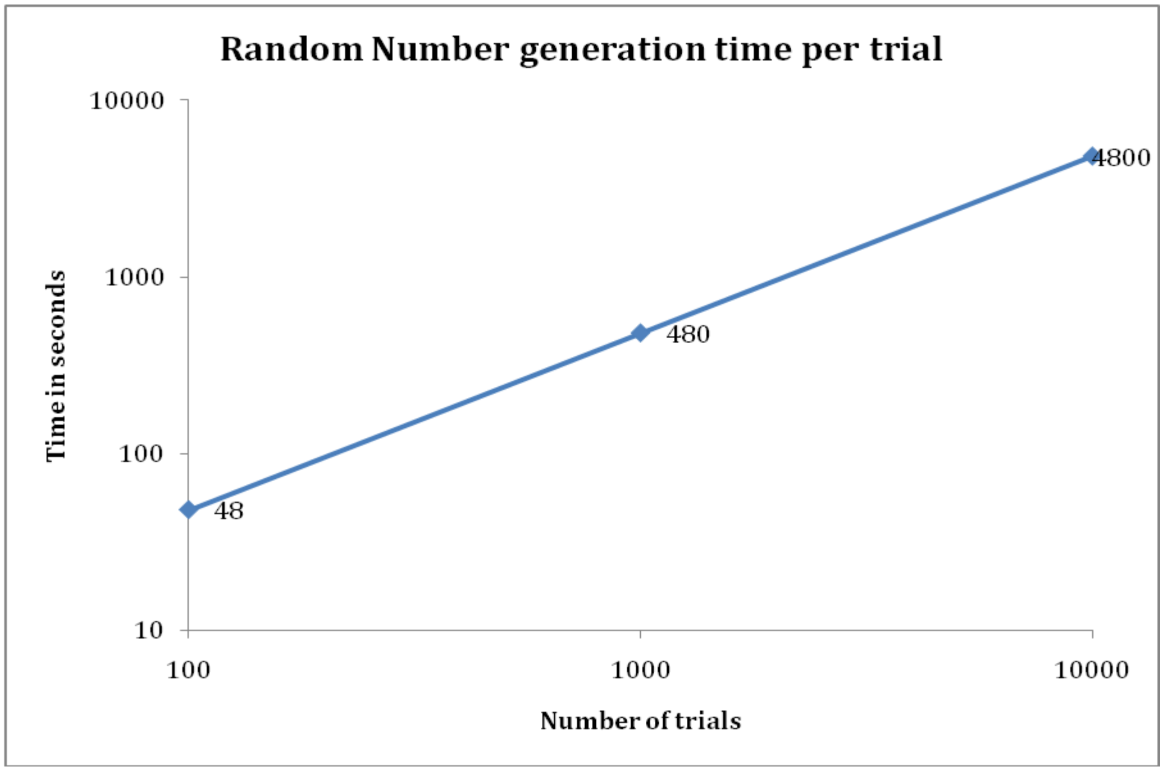

**Fig. 15. Random number generation time across trials.** In Fig. 14, the MPI process of rank 0 used just over 2 s to compute the result, while the process with rank 99 used around 50 s. We notice from this figure that almost 48 s rank 99's time is consumed in random number generation.

As seen in Fig. 15, if the same strategy for parallelization of FAVPFM is used the overhead increases with the increase in the number of trials and hence for 10000 trials the overhead is around 4800 s or 1.34 h, that is 1/3 of the evaluation time (16314 s or 4.5 h) is spent in the generation and throw away process.

Alternatively, if the serial RNG process is carried out before parallel execution, we can reduce several hours of parallel execution time to minutes. In the 10000-trial scenario the parallel computation time would be around 234 s, i.e. less than 4 minutes by eliminating the unwanted (generate and throw-away) computations performed by the MPI processes. By doing this we achieve linear speed up.

Hence, we generate the initial random number seeds before hand and read them from a file during the execution. Figure 14 shows the time consumed by each MPI process during the execution of a 100 trial scenario in parallel, using 100 processes. Average time taken by each MPI process is 2.34 s and the maximum time taken is 2.42 s. Hence, 100 trials are evaluated in less than 2.5 s. Note that the needless computation that made the maximum computation time for 100 trials around 50 s (shown in Fig. 6) is eliminated here.

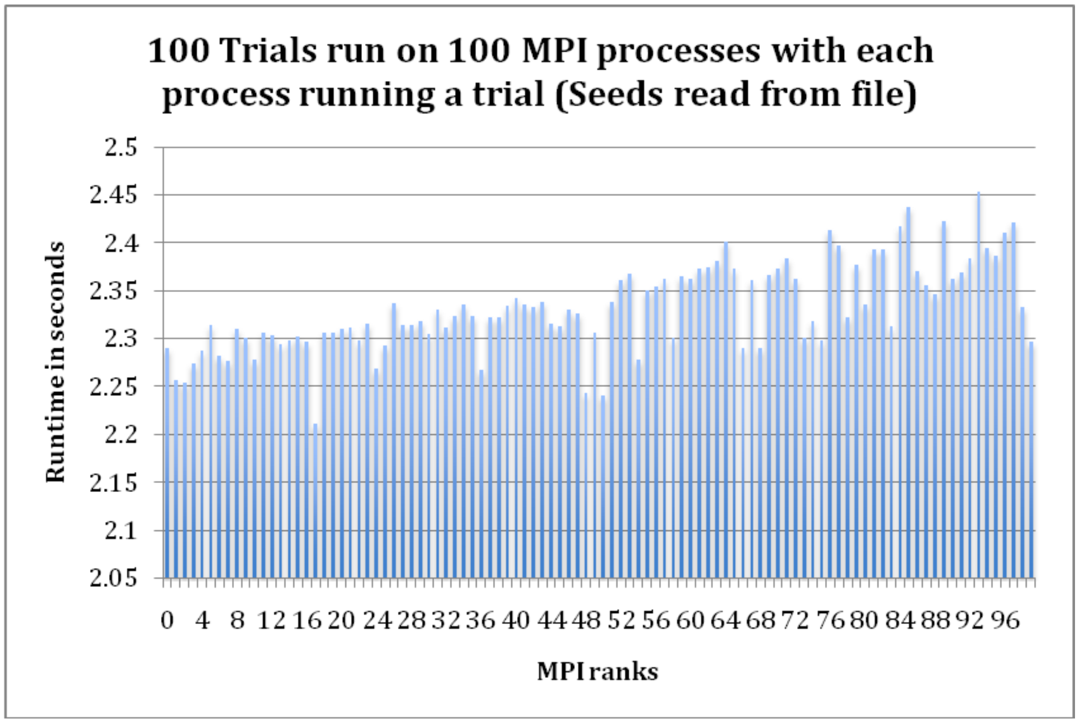

**Fig. 16. Runtime of every MPI process rank (Initial random-seed read from a file strategy).** Note here the runtime of all the MPI processes fall in between 2.2 s and 2.5 s. Comparing this with Fig. 14 (where the runtime was dependent on the trial number the MPI process was running), we ensure that the overhead due to the unnecessary generation of random numbers is eliminated.

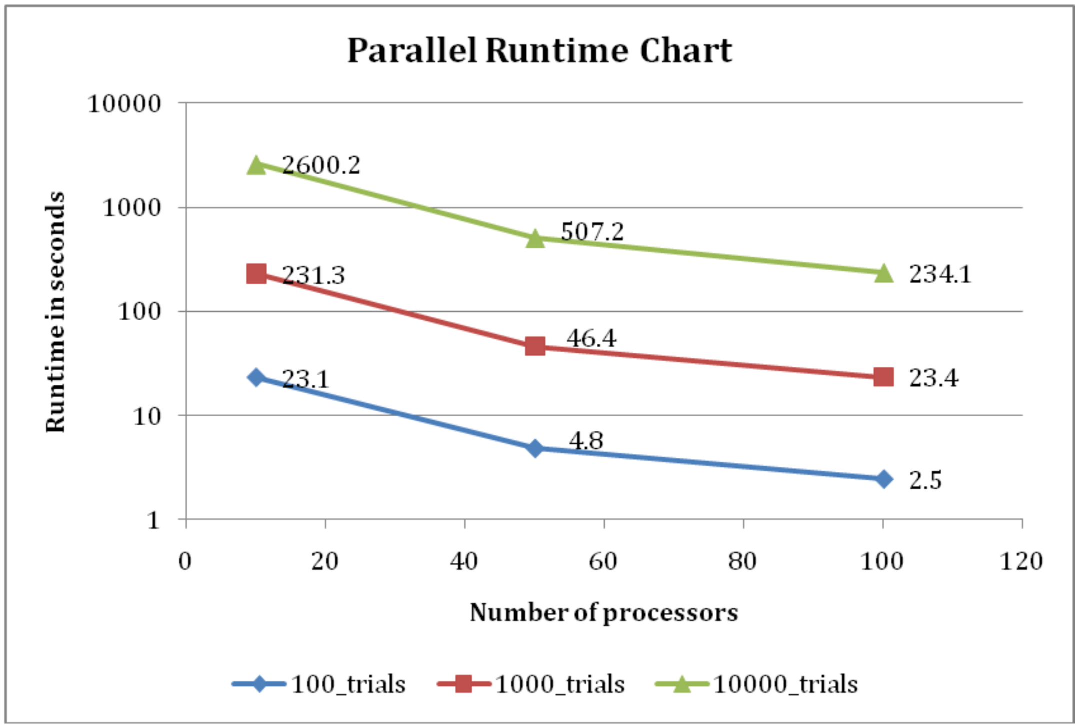

**Fig. 17. Parallel runtime across varying number of trials of FAVPFM.** Comparing this result with Fig. 13, we show that the runtime in parallel run with 10000 trials is reduced from 4.5 h to less than 4 min with 100 processor cores.

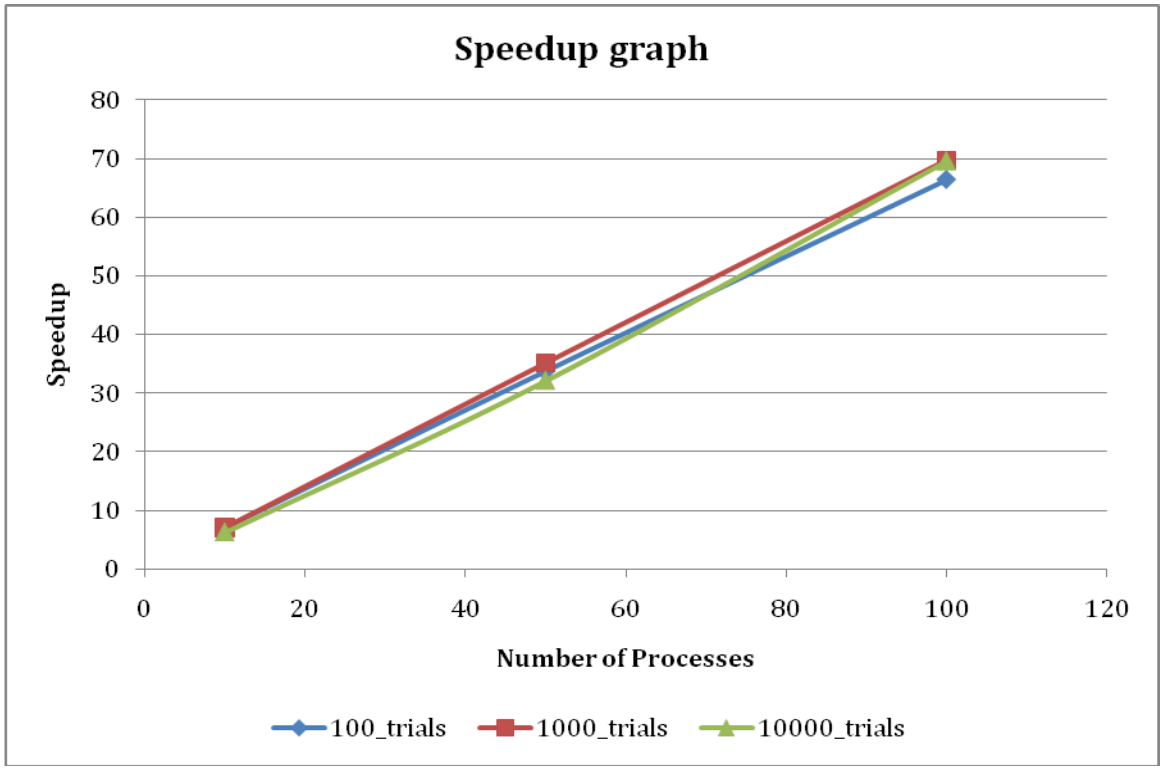

**Fig. 18. Speed up curves for 10, 50, and 100 processor-cores show that the runtime speed in the FAVPFM computation increases by 7, 35, and 70, respectively suggesting 70% efficiency.** Further, the curves suggest this efficiency remains consistent regardless of the change in the number of trials in the simulation.

Figure 17 shows the time taken by parallel processes (10, 50, and 100) to execute (100, 1000, and 10000) FAVPFM trials. The maximum-time taken by a parallel process in the parallel execution is used to plot the speed-up graph shown in Fig. 18. The speedup graph suggests a linear speedup with a consistent 70% efficiency across varying number of trials.

## 6. SUMMARY

As opposed to a standard way of designing parallel algorithms for a particular problem, this report discusses a methodology to infuse the parallel computing capability into a production code. The report started with a brief introduction to the Incremental Parallelization Approach (IPA) and Fracture Analysis of Vessels – Oak Ridge (FAVOR). Using IPA, we incrementally overcame the dependencies of the parallel processes using various strategies and finally verified the correctness of the result by comparing it with the result from the serial code. We evaluated the performance of the parallel runs for scenarios with varying number of trials using 10, 50, and 100 processor-cores and achieved linear speedup with 70% efficiency. In the largest scenario involving 10000 trials that we ran, we were able to reduce the runtime of the FAVPFM module from 4.5 h to less than 4 min using 100 processor cores and with zero loss in accuracy.

.

## REFERENCES

1. P. T. Williams, T. L. Dickson, and S. Yin, *Fracture Analysis of Vessels – Oak Ridge: FAVOR, v0.61, Computer Code: Theory and Implementation of Algorithms, Methods and Correlations*, ORNL/NRC/LTR=05/18, Oak Ridge National Laboratory, Oak Ridge, TN.
2. T. L. Dickson, P. T. Williams, and S. Yin, *Fracture Analysis of Vessels – Oak Ridge FAVOR,v05.1, Computer Code: User's Guide*, ORNL/NRC/LTR=05/17, Oak Ridge National Laboratory, Oak Ridge, TN.
3. G. M. Fox, G. Lyzenga Johnson, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, Vol.1, Prentice Hall, Eaglewood Cliffs, New Jersey, 1988.
4. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI Portable Parallel Programming with the Message-Passing Interface*, MIT press, Cambridge, Massachusetts, 1994.
5. B. Wilkinson and M. Allen, *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, New Jersey, 1999.