Distributed Computing for Probabilistic Structural Integrity Analysis of Aircraft Structures

Gulshan Singh*, Juan Ocampo†, Carlos A. Acosta‡ and Harry Millwater §

University of Texas at San Antonio, San Antonio, TX, 78249

Probabilistic risk assessment of aircraft structures is inherently time consuming as multiple probabilistic structural integrity analyses are required to account for the airplane-to-airplane and flight-to-flight variations and each probabilistic analysis requires a sufficient amount of Monte Carlo samples (MCS) to insure convergence. In this paper, high performance computing tools, OpenMP and Message Passing Interface (MPI), are investigated with an aim to significantly reduce the computational time required for risk assessment. Although the inherit independence of MCS is favorable, efficient distributed computing demands additional coding work and tuning. OpenMP and MPI directives are implemented to parallelize the risk assessment analysis and tested with a different number of processors. The results show a 6.64 **times** *speedup* **for OpenMP implementation on** 8 **processors and** 287.93 **times** *speedup* **for MPI implementation on** 512 **processors.**

I. Introduction

A risk assessment of the General Aviation fleet can provide important insight to the critically and severity of a potentially serious structural issue. A comprehensive probabilistic methodology was developed that can assist Federal Aviation Administration (FAA) engineers in conducting a risk assessment of a General Aviation structural issue in support of policy decisions. A schematic of the strategy is shown in Figure 1. For realistic results, probabilistic distributions are developed for the relevant parameters such as gust and maneuver load exceedances, flight or aircraft (A/C) velocity, flight distance, sink rate velocity, Miner's damage coefficient, and probabilistic stress life curves (S-N).[1,2] Using a probabilistic code written in Fortran and Monte Carlo sampling, a statistical representation of the flights-to-failure was developed. These statistics were then post-processed to determine cumulative distribution function of the probability of failure.

Many times the probabilistic risk assessment results are required in almost real-time

*Post-doctoral researcher and AIAA Member.
†Graduate research assistant and AIAA Member.
‡Research assistant and AIAA Member.
§Associate Professor and AIAA Member.

to make decisions. This requirement forces one to consider high performance computing techniques because computation time for a Monte Carlo analyses can be long and multiple analyses may be needed. The distributive computing techniques[3,4] such as OpenMP and MPI[5,6] make it possible to achieve a certain level of *speedup* but the implementation requires additional coding. The *speedup* is defined, as shown in equation 1, by the ratio by which the wall time to solution can be improved compared to using only a single processor. The *speedup* shows how much faster the problem can be solved when using multiple processors. Any algorithm requires close examination to avoid data- or memory-bound serial bottle-necks that must be avoided to achieve efficient parallelization.

$$Speedup = \frac{(\text{Executation time})_{parallel}}{(\text{Executation time})_{serial}} \tag{1}$$

The advantage of the MCS-based probabilistic calculation is that the Monte Carlo samples are independent of each other and different processors can be used to perform a portion of samples without depending on the results from the other processors. A typical serial and distributed strategy for MCS is shown in Figure 2. As shown in the figure, a serial strategy calculates results for one Monte Carlo sample at a time. A distributed strategy calculates results for more than one (the number depends upon the number of processors) Monte Carlo sample.

Even though distributed computing for Monte Carlo seems to be a straightforward exercise, a significant amount of additional coding and careful communication of the data among processors is required. The SMall Airplane Risk Technology (SMART) probabilistic code was modified to implement OpenMP and MPI. An MPI implementation required more modification to the risk assessment algorithm as compared to the OpenMP implementation. The performance of the modified codes is compared with the performance of a serial code.

II. Probabilistic Risk Assessment

Deterministic risk assessment is not an adequate method to evaluate the continued operational safety of the general aviation fleet because of the uncertainties involved in the analysis. A deterministic approach does not consider variations in structure and load parameters whereas a probabilistic risk assessment does. In a deterministic approach, the damage accumulated in a flight is assumed to be repeated for all the flights; however, there are always variations between flights due to different load parameters and operating conditions. To account for these variations a probabilistic risk assessment is essential.

In the probabilistic risk assessment gust and maneuver load exceedances, flight velocity, flight distance, sink rate velocity, Miner's damage coefficient, and stress life curves are modeled using probabilistic distributions. The list of the probabilistic variables is shown in Table 1. The probabilistic distributions are constructed from the information available in the literature.[1,2,7] This approach provides a different damage from each flight and the damages are accumulated until Miner's critical value is reached.

The probabilistic methodology begins with reading the user specified data for gust and maneuver load limit factors, one g and ground stress, airplane usage, exceedance curves, and sink rate. A schematic of the methodology is shown in Figure 1. The specified data provide the distribution of each parameter. This user input data is employed to develop mission profile and stress spectrum for each Monte Carlo simulation. Probabilistic stress life (S-N)

curve information is taken from the literature. This randomness includes flight-to-flight and airplane-to-airplane variations. The provided data is used to generate the parameter values for all the Monte Carlo simulations. A set of these parameters represent one flight and is used to calculate damage from the flight. The damage is accumulated to a maximum value for each Monte Carlo sample. When all the Monte Carlo estimations are finished the results are post-processed to calculate mean, standard deviation, and 95% confidence bound on mean and standard deviation of flights-to-failure. The same calculations are also performed for hours-to-failure.

The calculations performed for each Monte Carlo sample to calculate flights-to-failure take 0.45 second after compiler optimization. $10,000$ to $50,000$ MC samples are necessary for reasonable risk assessment. The total time required is $4,531$ *second* (≈ 1.25 *hours*) and $22,655$ *second* (≈ 6.3 *hours*) for $10,000$ and $50,000$ samples, respectively. Therefore, it is essential to investigate distributed computing options that can reduce the time required to obtain the results.

III. Distributed Computing

With the increased speed of processors, multi-processor desktop machines, and the availability of supercomputing clusters it is possible to perform the risk assessment at a faster speed by using distributed computing. Distributed computing requires additional design and programming complexity to solve the problem but as a result provides efficient computation. The two options, OpenMP and MPI, are investigated in an attempt to achieve the research goal.

OpenMP and MPI are programming tools including library routines, complier directives, environment variables, and functions that can perform distributed computing in Fortran and C/C++ programs.[8] The two key steps to distributed computing are to divide a computation into smaller computations and assign them to different processors for parallel execution. The size of the smaller computations or tasks, to be distributed to processors, can be statically or dynamically defined. Tasks are programmer-defined units of computation into which the main computation is subdivided by means of decomposition. These tasks are defined and distributed based on dependences. The tasks can be easily defined and distributed in risk assessment using Monte Carlo sampling. In a cluster, if speed and availability of all processors are the same then an equal number of samples can be distributed among all the processors. Figure 2 illustrates the equal distribution of 6 Monte Carlo samples among 6 processors.

Distributed computing may achieve desired *speedup* for many problems especially for uncertainty quantification (risk assessment) is using Monte Carlo sampling. Before distribution of the tasks it is necessary to ensure that the results (flights-to-failure and probability of failure) do not depend upon the number of processors (serial or distributed). Generating the random numbers (different realization) before distributing the tasks can ensure that results independent of computing type. The random numbers are distributed to processors accordingly to the tasks assigned to the processors.

A. OpenMP

OpenMP[9] is comprised of a set of compiler directives and a supporting library of subroutines that works in conjunction with either Fortran or C/C++. OpenMP[8–10] is primarily designed

for shared memory multiprocessors. Typical shared and distributed (message passing) architectures are shown in Figure 3 where $P\#$ indicates the processors number and $M\#$ indicates the respective memory. In a shared memory architecture all of the processors have access to all the memory of the machine.

An advantage of OpenMP is that the same code can be used on single- and multi-processor platforms. This allows one to implement OpenMP on a part of the program and increase its usage incrementally. Another advantage is that the tasks are defined and distributed by directives without specific coding from the programmer. OpenMP runs efficiently on shared-memory multiprocessor platforms. OpenMP implementation does not require additional library installation. There are several disadvantages to OpenMP such as scalability limited to shared memory architectures, only efficient in shared-memory architecture, can be difficult to debug, and lack of thread control.

Monte Carlo sampling-based probabilistic risk assessment is an ideal problem for OpenMP because of its advantage when applied to loop-level (Monte Carlo sampling loop) distributed computing. OpenMP directives are embedded within the risk assessment Fortran program. The master processor creates a thread for each processor with a total number of threads equal to total number of processors. Some input and result variables are declared as private to add synchronization in computations. Once the required number of Monte Carlo samples are performed, the master processor post-processes the results.

B. Message Passing Interface (MPI)

As shown in Figure 3, a set of processors have access to only local memory but are able to communicate with processors by sending and receiving messages. This memory architecture, called message passing or distributed memory architecture, is commonly used in high performance computing[11,12] machines. Data transfer from the local memory of one process to the local memory of another process is performed by message passing. MPI[8,10,13] is a library and specification consists of subroutine that allows processors to communicate with one another.

The advantages of MPI are universality, ease of debugging, and resulting performance. The universality means that it can be implemented on a parallel supercomputer, a work station network, or a personal computer cluster connected through a fast or slow network. MPI can also be applied to wider range of problems because it exploits both task and data parallelism. The task distribution can be adaptive to manage imbalance in processor speed. One disadvantage of MPI is that its implementation requires more additional programming changes (from serial to distributed) as compared to OpenMP.

A flow chart of the MPI implementation is shown in Figure 4. In the implementation the realizations (random numbers) are generated by a master processor. The realization generation in not distributed because the cost of this task is not significant as compare to risk assessment calculations. The master processor equally distributes the tasks among all the processors. The remainder tasks, if any, are distributed one task at time to the processors until all tasks are distributed. The process is such that each processor gets access to its portion of data to copy it into local variables. Dynamic allocation is used to allocation memory for the local variables. *MPI_Scatter*, *MPI_Gatherv*, and *MPI_BARRIER* are used to distribute the data, gather the results, and synchronize the processors, respectively.

Once a processor has finished its part of the Monte Carlo simulation, it sends back the

results to the master processor. The master collects all the results and performs the required post-processing. The post processing includes calculating probabilities and writing the results. The post-processing is also not distributed because its computational time is not significant as compared to the risk assessment.

IV. Results and Discussion

The timing results presented in this section are wall time. The CPU time was not used because it does not include the communication time between processors. The communication time is part of the computation and should be considered in the total time.The time reported here do not include wait time. The local cluster and personal computer used in this research does not have any wait time. However distributed computing is mostly performed on clusters that are shared by many researchers. All the jobs are first submitted in a queue and the job is started based on its priorities. The time difference between the submission and beginning of the job is called wait time. Typically, the wait time is longer for a job requests more processors as compare to a job that request fewer. The wait time is not included in the results.

The OpenMP study was performed on a unix machine, 2 quad-core 2.8 GHz Intel Xenon. Table 2 shows the results from the OpenMP implementation for $10,000$ samples. Figure 5 shows a comparison between the *speedup* from an ideal case and from an OpenMP parallel implementation. The maximum number processors investigated for the OpenMP implementation was 8 and the corresponding *speedup* achieved was 6.64. The *speedup* for 2 and 4 processors was 1.96 and 3.49. Only 8 processors were used because the shared memory architecture required for OpenMP was not available for higher number of processors.

Two clusters were used for MPI investigations: a local cluster titled *Shamu* and a cluster at Texas Advanced Computing Center (TACC) called *Ranger*.[11] The Shamu system is comprised of 24 nodes, each node contains a 2-Quad-Core 2.28 GHz Intel Xenon (192 total processors) running a Linux operating system. The Ranger system is comprised of $3,930$ 16-way SMP compute nodes (62,976 total processors). The Ranger nodes used in the MPI implementation contain 2 Quad-Core 2.3 GHz Intel Xenon processors per node running a Linux operating system.

Table 3 shows the results from the MPI implementation. Figure 6 shows a comparison between the *speedup* from an ideal case, the *speedup* from MPI implementation on Shamu, and the *speedup* from the MPI implementation on Ranger (TACC). A maximum 150 Shamu processors were used providing the *speedup* of 117.06. A maximum of 512 Ranger processors were used providing a *speedup* of 287.93. Multiple runs for the same number of processors show a small variations in the *speedup*. This is due to the difference in communication time for various runs.

The *speedup* for 8 and 16 processors on the local cluster was 8.09 and 16.70, indicating super-linear speedup. A reason for super-linear *speedup* is that computation time is not the only bottle-neck for the serial case. The overall speed of a computation is determined not just by the speed of processor, but also by the ability of the memory system to feed data to it. The total time depends upon the slower of the two task: feeding rate and computation. Depending upon the computer there are multiple levels[6] of memories. Without going into the details of memory hierarchy, it is faster to access data from caches as compare to disk

memory. Among caches, the data feeding rate of a cache decreases with increase in its size. If all the data can fit in caches than the feed rate is higher as compared to accessing the data from disk. The super-linear *speedup* is obtained because 8 and 16 processors are able fit more or even all data into their caches.

The results (Table 3) show that the *speedup* saturates on both Shamu and Ranger. The *speedup* reaches saturation when the wall time is not reducing by increasing the number of processors. One reason behind saturation is that the communication time between processors becomes the larger part of total time as compared to the computation part. It can be inferred from the results that if the number of samples is larger than $50,000$, the saturation may occur for a larger number of processors. The saturation will happen for a higher number of processors because it will take higher number of processors for communication time to be the larger part of the total time. The saturation for Ranger occurs at a larger number of processors then Shamu because the interconnection architecture are different.

MPI results depend upon many factors such as the algorithm, implementation, and cluster architecture. A MPI implementation also dependents upon number of inputs, number of outputs, and the intermediate interaction steps. The results show that MPI was implemented efficiently. The cluster architecture is an important factor in distributed computing. A linear array interconnection[6] between cluster nodes is beneficial when a lower number of nodes are used in computation. For example the the super-linear *speedup* from local cluster indicates direct connection nodes in use. The 2 & 3 dimension meshing interconnections (with or without wrap)[6] can be beneficial if a higher number of processors are needed. At the same time, a 2 or 3 mesh interconnection is expensive when compared to a linear array interconnection.

Although it is possible to control the processor allocation through MPI but it is an internal process in most applications. This allocation also affects the *speedup* results. If two nodes are requested, MPI allocated nodes could have a direct connection between them or have a connection that includes multiple nodes. The computation time will increase in case of a longer path to send and receive messages between clusters. A use of the cluster nodes by other users can create congestion between nodes and can affect the results. All the factors combined together can significantly affect the performance of the distributed computing.

V. Summary Remarks and Conclusions

A probabilistic risk assessment tool was developed that can take 1.25 *hours* for $10,000$ and 6.3 *hours* for $50,000$ Monte Carlo samples in serial on a personal computer. Distributed computing options, OpenMP and MPI, were employed to obtain the same results in a faster time. The probabilistic risk assessment code was modified to implement these options.

The results indicate that both approaches provide significant *speedup*. The OpenMP and MPI implementations with 8 processors provide a *speedup* of 6.64 and 8.0; MPI *speedup* was 20% higher than OpenMP. The MPI implementation for $50,000$ samples provided a maximum *speedup* of 117.06 for 150 processors on Shamu and a maximum *speedup* of 287.93 for 512 processors on Ranger. The trends in Figure 6 show that higher *speedup* than 287.93 can be achieved at Ranger. The ratio of the *speedup* and number of processors reduces as number of processors are increased more than 32. The *speedup* saturations for Shamu and Ranger are at different number of processors and both saturation points can increased if

more than $50,000$ samples are needed.

The *speedup* numbers show significant run-to-run variability when more than 100 processors are used because of the communication time and interconnections uncertainties. This happens when the total computation time is less than 5 minutes but not so much for a longer simulation time. The random number generation and post processing are performed in serial in all the cases, the computation time can be further reduced by parallelization these tasks.

Acknowledgments

This research is funded by the Federal Aviation Administration (FAA) contract number 07-G-011. The authors would like to thank Marvin Nuss, Michael Reyer, Dr. Felix Abali, and Dr. Michael Shiao of the FAA and Dr. Herb Smith and Eric Meyer of The Boeing Company for their technical input and feed-back in this project. The authors would also like to thank John Grafton of UTSA for his help in implementing MPI on Shamu.

References

[1] "Fatigue Evaluation of Wing and Associated Structure on small Airplanes," FAA Report AFS120-73-2, Federal Aviation Administration, 1973.

[2] "Fatigue Fail-Safe, and Damage Tolerance Evaluation of Metallic Structure for Normal, Utility, Acrobatic, and Commuter Category Airplanes," FAA Report AC23-13A, Federal Aviation Administration, 2005.

[3] Zhang, W., Accorsi, M., and Leonard, J., "Parallel Implementation of Structural Dynamic Analysis for Parachute Simulation," *AIAA Journal*, Vol. 44(7), 2006, pp. 1419–1427.

[4] Ghidella, J., Wakefield, A., Silvina, G.-F., Friedman, J., and Cherian, V., "Use of Computing Clusters and Automatic Code Generation to Speed Up Simulation Tasks," *AIAA Modeling and Simulation Technologies Conference and Exhibit, Hilton Head, South Carolina, USA, Aug. 20-23*, 2007, pp. AIAA 2007–6880.

[5] Bhardwaj, M., Reese, G., Drissen, B., Alvin, K., and Day, D., "Salinas - An implicit Finite Element Structural Dynamic Code Developed for Massively Parallel Plateforms," 41^{st} *AIAA Structures, Structural Dynamics, and Materials Conference Atlanta, GA, USA*, 2000, pp. AIAA 2000–1651.

[6] Grama, A., Gupta, A., Karypis, G., and Kumar, V., *Introduction to Parallel Computing*, Pearson Education Limited, 2^{nd} ed., 2003.

[7] Keshavanarayana, S., Smith, B. L., Gomez, C., Caido, F., Shiao, M., and Nuss, M., "Fatigue-based severity factors for shear-loaded faster joints," *Journal of Aircraft*, Vol. 47(1), 2010, pp. 181–191.

[8] Chapman, B., Jost, G., and Pas, R. V. D., *Using OpenMP: Portable Shared Memory Parallel Programming*, Massachusetts Institute of Technology, 2008.

[9] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R., *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers, 2001.

[10] Pacheco, P., *Parallel Programming with MPI*, Morgan Kaufmann Publishers, 1997.

[11] TACC, "Texas Advanced Computing Center : HPC, The University of Texas at Austin," 2008 (accessed March 8, 2010), Website: http://www.tacc.utexas.edu/.

[12] OSC, "Ohio Supercomputer Center," 2009 (accessed March 8, 2010), Website: http://www.osc.edu/.

[13] Gropp, W., Lusk, E., and Skjellum, A., *Using MPI*, The MPI Press, 1999.

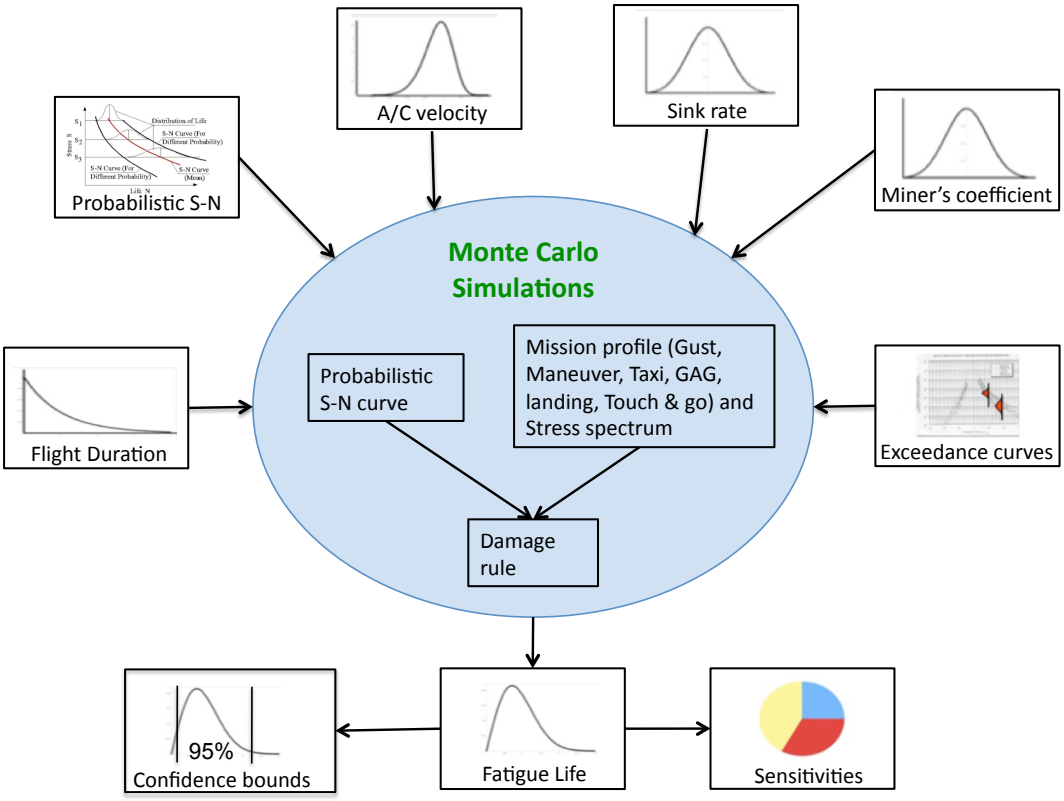

Figure 1. MCS-based probabilistic risk assessment flow-chart

Table 1. List of random variable in the structural integrity analysis

	Random variables
1	Gust/Maneuver Load Exceedances
2	Flight or Aircraft (A/C) velocity
3	Flight distance
4	Sink rate velocity
5	Miner's damage coefficient
6	Probabilistic stress life curves (S-N curves)

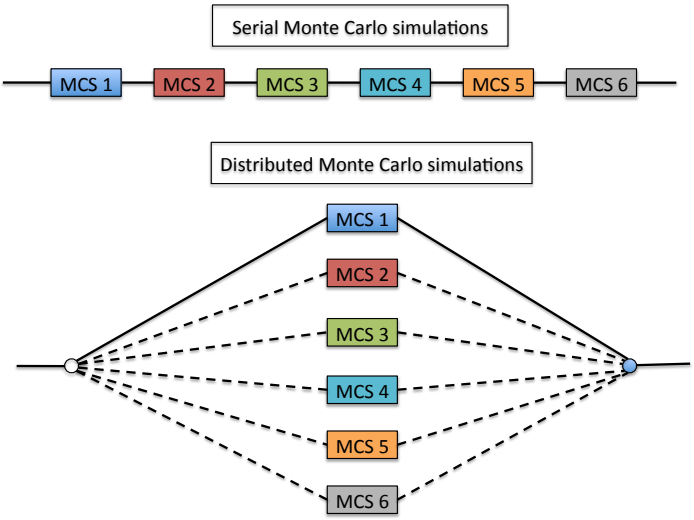

Figure 2. Serial and parallel execution of MCS

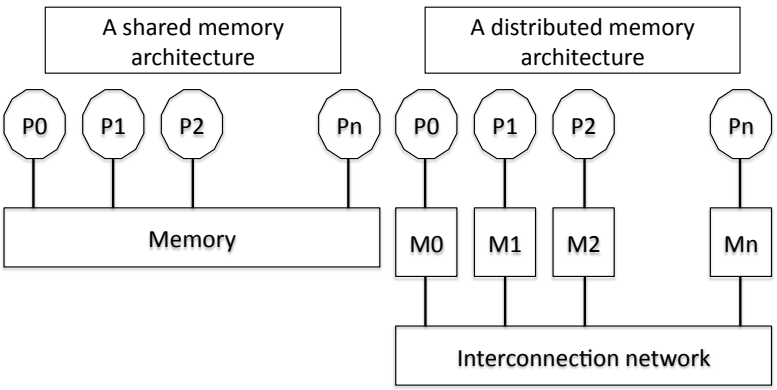

Figure 3. Typical shared and message passing memory architectures

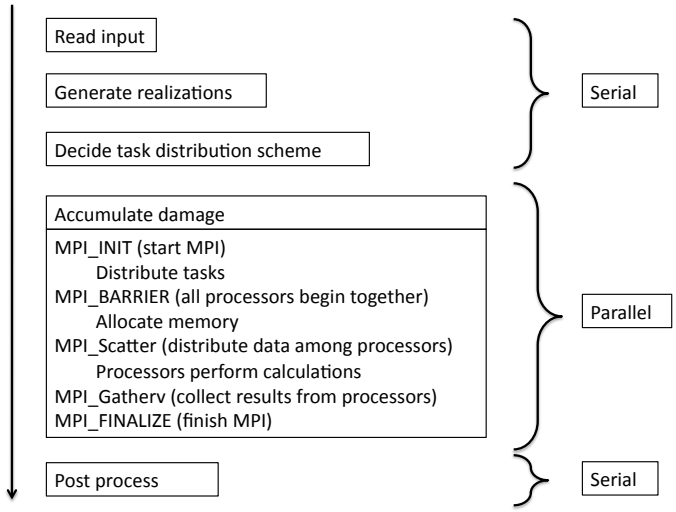

Figure 4. Probabilistic risk assessment flowchart with MPI

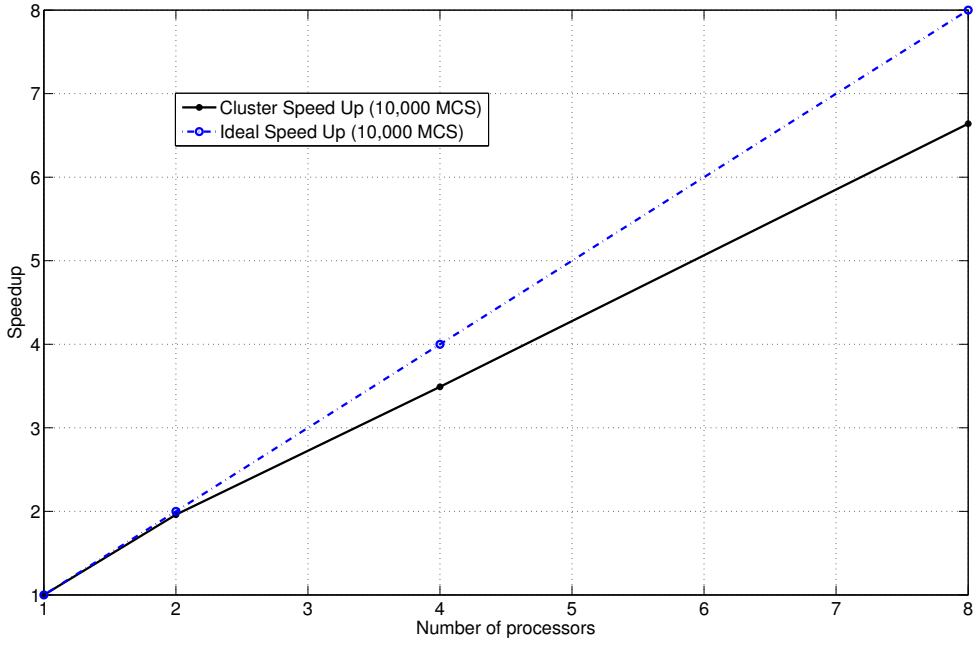

Figure 5. OpenMP *speedup* with respect to number of processors

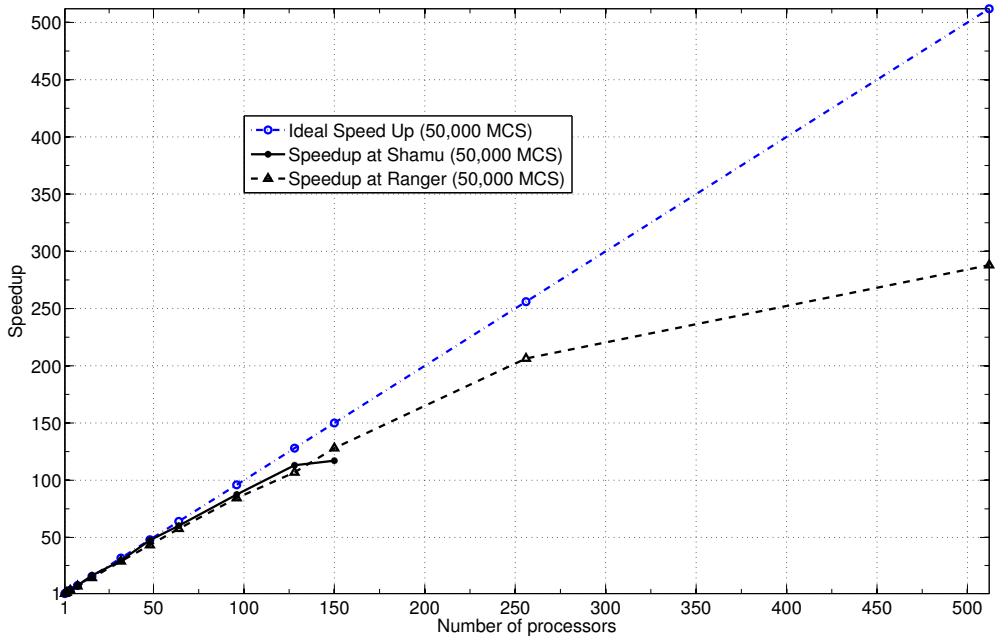

Figure 6. MPI *speedup* on Shamu and Ranger with respect to number of processors

Table 2. OpenMP results for 10,000 samples

Number of processors	Execution time (Seconds)	Speed Up
1	4531	1.00
2	2272	1.96
4	1301	3.49
8	680	6.64

Table 3. MPI results for 50,000 samples

	Shamu		Ranger	
Number of processors	Execution time in seconds (h:min)	Speedup	Execution time in seconds (h:min)	Speedup
1	13892 (3:52)	1.00	14045 (3:54)	1.00
2	7164 (1:59)	1.94	7505 (2:05)	1.87
4	3749 (1:02)	3.70	3775 (1:03)	3.72
8	1715 (0:28)	8.09	1899 (0:32)	7.40
16	831 (0:14)	16.70	953 (0:16)	14.74
32	467 (0:08)	29.74	482 (0:08)	29.16
48	292 (0:05)	47.64	324 (0.05)	43.35
64	231 (0:04)	60.20	244 (0:04)	57.66
96	159 (0.03)	87.54	166 (0:03)	84.45
128	123 (0:02)	113.11	131 (0:02)	106.87
150	119 (0:02)	117.06	109 (0:02)	128.00
256	-	-	68 (0:01)	206.33
512	-	-	49 (0:01)	287.93