48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition
4 - 7 January 2010, Orlando, Florida
AIAA 2010-181

High Performance Computing Implementation on a Risk Assessment Problem

Juan D. Ocampo[1] and Carlos A. Acosta[2]
University of Texas at San Antonio, San Antonio, TX, 78249

Harry Millwater[3]
University of Texas at San Antonio, San Antonio, TX, 78249

Risk assessments of engineering structures are notoriously time consuming. Due to the aging of current structures, risk evaluations are needed more often, and in many cases decision makers need the results in almost real time. This work aims to evaluate the impact of using compiler optimizations, Message Passing Interface (MPI), and OpenMP directives for a Monte Carlo sampling fatigue code used in risk assessment of General Aviation. Compiler optimization permits high performance speed with accurate solutions without additional coding effort. OpenMP and MPI provide directives and functions to parallelize programs but demand more coding work and sometimes do not give large speed up results. These two methods can meet the growing need for real time risk assessments.

I. Introduction

Monte Carlo simulations are used in engineering to evaluate uncertainties caused by random variables. In many cases, this process is lengthy and new tools are needed to improve computation time. To improve computational time, it is possible to use compiler optimization options (single processor optimization) that do not require additional coding time or to implement MPI or OpenMP directives, which can be tedious but effective for speed up (Pacheco, 1997). The focus of this work was to improve computation time using compiler optimization options, MPI, and OpenMP directives. To test the implementation, a code that performs risk assessment of the General Aviation fleet was used to examine the advantages, disadvantages, and speed up calculations of each method.

General Aviation risk assessment encompasses the required elements necessary to conduct a structural integrity evaluation and moreover considers real-world airplane-to-airplane and flight-to-flight variations such that a realistic risk assessment can be made of an aircraft structural detail. A schematic overview of the process is shown in Figure 1.
The probabilistic methodology is explained step by step as follows:

- Values for variables such as the maneuver and gust load limit factors, one g stress, and ground stress are loaded.
- According to the airplane usage, the respective data (exceedance curves, sink rate data, etc.) are loaded.

[1] Master's Student, Mechanical Engineering Department, EB 1.04.06, AIAA Student Member
[2] Bachelor's Student, Mechanical Engineering Department, EAFIT University, AIAA Student Member.
[3] Associate Professor, Mechanical Engineering Department, EB 3.04.50, AIAA Member

- Realizations of the random data as sink rate velocity, airplane velocity, flight duration, take off weight, etc. needed for Monte Carlo sampling are generated.
- Inside each Monte Carlo run, the code generates a characteristic stress spectrum.
- Damage is accumulated for each Monte Carlo run until Miner's critical value is reached and flights-to-failure is recorded.
- When the Monte Carlo sampling is finished, the random variables and flights-to-failure are stored for risk assessment.

General Aviation risk assessment uses Monte Carlo simulations to evaluate uncertainties through random variables and provide important insight to the criticality and severity of a potentially serious structural issue. Monte Carlo simulations are time consuming; for this specific problem, each Monte Carlo run takes around 2.3 seconds on average without any optimization. If a user needs to run 10,000 samples, the code will take around six and a half hours to compute the results. Considering that computers have more powerful processors and now often run on multiple cores, there is a possibility of running optimized codes with parallel options with less computing cost. Compiler optimizations, OpenMP, and MPI directives offer tools that can speed up computational problems and give accurate solutions.

II. Methodology

To evaluate compiler optimization, OpenMP, and MPI; a risk assessment code was used. The code computes Flights/Hours to Failure (time to crack initiation) and the airplane damage at any flight hours given a load history, structural details, and material stress-life (S-N curves) using Miner's damage rule. This algorithm was created following the guidelines used for safe life evaluation in FAA reports AFS-120-73 and AC-23-13A. The code was implemented through a Monte Carlo sampling algorithm.

The Monte Carlo simulation randomly selects values from the distributions of each variable listed in Table 1. With this information, the code generates a representative stress spectrum according to the airplane usage. Using the material stress-life information and invoking Miner's damage accumulation rule, the damage per flight is calculated. This process repeats multiple times to develop a statistical representation of the Flights/Hours to Failure and other important variables.

A first approximation to speed up the code was made applying single processor optimization using two different compilers (Intel and PGI) with four compiler optimization levels. The degree of optimization can be set up using -O1, -O2, -O3 categories in order to achieve different levels of performance, such as architecture optimization.

In many cases, compiler optimization levels can achieve better speed. An overview of the different optimization levels is explained as follows: –O1 optimizes for speed, updates changes in variables to avoid data dependences, but normally increases the size of the code. –O2 optimizes loops inside the code by reallocating vector positions in order to access the information faster and minimize the code size. –O3 specifies more aggressive optimization. This option rearranges the code in order to transform loops by doing loop unrolling and code replication to eliminate branches. This optimization can get better time performances but at the cost of code size. Option –O0 specifies fast compilation and disables optimization.

After doing the evaluation with different compilers and optimization levels High Performance Computing (HPC) was implemented. HPC is a branch of computer science that deals with technologies and techniques that let multiple computer systems and applications work together to solve common problems with efficient performance and improved processing speed (Chandra et

al., 2001). Those techniques include OpenMP and MPI. Since each Monte Carlo run is independent, a second approach using OpenMP and MPI multi-threading compiler directives can be used to execute the code in parallel threads and achieve better performance. A thread refers to a runtime entity that is able to independently execute a stream of instructions. A team of threads is created to execute the code in a parallel region of an OpenMP/MPI program. Figure 2 illustrates a team of threads within a parallel region.

The implementation takes place by calling headers and runtime functions that enable the compiler to create sequences of instructions to execute portions of work concurrently. Each sequence has a data address space that contains the variables specified in the program with dynamic allocation in the stack memory. This includes global variables or variables within a subroutine.

OpenMP and MPI are application programs for a set of library routines, compiler directives, environment variables, and functions that can perform shared memory parallelism in Fortran and C/C++ programs. OpenMP specification provides a model for parallel programming that is portable across shared memory architectures (Chapman et al., 2008) and MPI provides an independent memory protocol that provides essential virtual topology, synchronization and communication between a set of processes. OpenMP and MPI implementation can achieve maximum speed up by taking advantage of supercomputers and clusters since they operate with the highest level of performance. The environment variables prior the execution of the program let multiple threads process symmetric streams of information to control threaded parallelism.

Computer codes for engineering problems often use loops with many of iterations and large data sets. In these problems it is important to perform the correct synchronization and the appropriate implementation distributions among different processes by setting up barriers and communications among processors. The method of sending and receiving messages (MPI) with a portion of information to the correct processor is a crucial point in order to design a good parallel program.

The implementation of threaded parallelism using OpenMP and MPI uses environment directives to support data parallelism between processors. This allows parallel regions to achieve high performance by accessing the correct address space where the variables are allocated. Due to the shared memory architecture presented in OpenMP no more than eight processors can be used during the execution, that is the maximum number of processor per node or per machine. MPI can archive bigger number of processor compared with OpenMP thanks to its communication capability between nodes.

III. Results

Compiler optimization evaluation for two different compilers (Intel and PGI) is shown in Table 2. The results show the time in seconds that the code used to calculate the first Monte Carlo run when the Flights-to-Failure is equal to 44,000 flights.

Parallel implementation results using OpenMP directives with different numbers of threads and Intel compiler optimization -O2 were compared with the results obtained running the serial code. The Flight-to-Failure results were exactly the same showing that there is no variable corruption during the parallel execution. Computation time decreased as expected running the parallel version of the code on eight processors exploring two different environments: a Unix machine that contains 2 Quad-Core 2.8 GHz Intel Xeon —See Table 3, and a Linux Cluster that contains 2 Quad-Core 2.3 GHz Intel Xeon processors per node — see Table 4.

Figure 3 shows the comparison between the ideal speed up of a parallel application versus the

real speed up of the program with two, four, and eight threads for both computing environments using OpenMP.

Table 5 and Figure 4 shows the comparison between the ideal speed up of a parallel application versus the real speed up of the program with two, four, eight, sixteen, thirty-two, fifty and one hundred threads using MPI.

Using 8 processors a speed up of 6.91 was archived applying OpenMP; it provides the results in 8 minutes instead of one hour. Using MPI with 100 processors, the results can be obtained in 49 seconds achieving a speed up of 78.54.

The speed up shown in Figure 3 and Figure 4 is not completely linear because the application of OpenMP and MPI adds overhead to the parallel solutions due to the communications and synchronization between threads.

IV. Conclusions

Without any optimization, the PGI compiler is faster than the Intel compiler, when optimization is applied the Intel compiler shows better time performance, and as expected optimization -O3 showed the best time execution for a single processor optimization.

One difficulty executing the OpenMP and MPI implementation is that parallel random number generators like Leap Frog or SPRNG have not been implemented. For that reason, all the numbers were generated outside of the Monte Carlo loop to assure the correct execution of the code.

The memory required for multiple copies of all sample data is large; to avoid memory issues, OpenMP parallel regions were implemented in order to set a work-sharing construct within the proper distribution of information between threads. MPI implementation does not have any memory problem issue because the code has only one copy of memory and the messaging between the processors pass the required memory for the correct execution.

Parallel computing is a tool that permits calculations of expensive computational operations at a lower cost, however, to implement high performance computing in a code it is necessary to be aware of the possibility of memory issues and inappropriate thread communications.

The benefits from the correct implementation of OpenMP and MPI directives imply that multiple processors are available to generate sequences of executable information concurrently. The right load of information among threads is necessary to achieve acceptable efficiency of the parallel solution.

Acknowledgments

The authors are grateful to the Federal Aviation Administration (FAA) for grant # 26-2205-05, which supports this research. In addition, the authors would like to thanks Dr. Michael Shiao, Dr. Felix Abali, Dr. Herb Smith, Eric Meyer, Marvin Nuss, and Michael Reyer for the invaluable collaboration in this project.

References

PACHECO P, Parallel Programming With MPI, Morgan Kaufmann Publishers, University of San Francisco, USA, 1997.

Chandra R, Leonardo D, Kohr D, Maydan D, Mcdonald J, Menon R, Parallel Programming in OpenMP, Morgan Kaufmann Publishers, ISBN 1-55860-671-8, 2001.

Chapman B, Jost G, Van Der Pas R, Using OpenMP, Massachusetts Institute of Technology, ISBN-13: 978-0-262-53302-7, 2008.

FAA Report AFS120-73-2, “Fatigue Evaluation of Wing and Associated Structure on small Airplanes.” 1973.

FAA Report AC23-13A, “Fatigue Fail-Safe, and Damage Tolerance Evaluation of Metallic Structure for Normal, Utility, Acrobatic, and Commuter Category Airplanes.” 2005.

Tables

Random variables
Gust/Maneuver Load Exceedances
Flight Velocity
Flight Distance
Sink Rate Velocity
Miner's Damage Coefficient
Probabilistic Stress Life Curves (S-N)

Table 1 Risk assessment Random Variables

Compiler/option	-O0(s)	-O1(s)	-O2(s)	-O3(s)
Intel	2.62	1.11	1.01	1.00
PGI	1.92	1.71	1.38	1.39

Table 2 Compiler Optimization Time Evaluation For 44000 Flights

Number of processors	Execution time (Seconds)	Speed Up
1	3503	1.00
2	1862	1.88
4	951	3.68
8	506	6.91

Table 3 OpenMP Speed Up Evaluation For 10000 Iterations Unix Machine 2 Quad-Core 2.8 GHz Intel Xenon

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 4 OpenMP Speed Up Evaluation For 10000 Iterations Linux Cluster 2 Quad-Core 2.3 GHz Intel Xenon Processors per Node

Number of processors	Execution time (Seconds)	Speed Up
1	3920	1.00
4	1005	3.91
8	522	7.48
16	274	14.30
32	137	21.61
50	90	43.52
100	49	78.54

Table 5 MPI Speed Up Evaluation For 10000 Iterations Linux Cluster 2 Quad-Core 2.3 GHz Intel Xenon Processors per Node

Figures

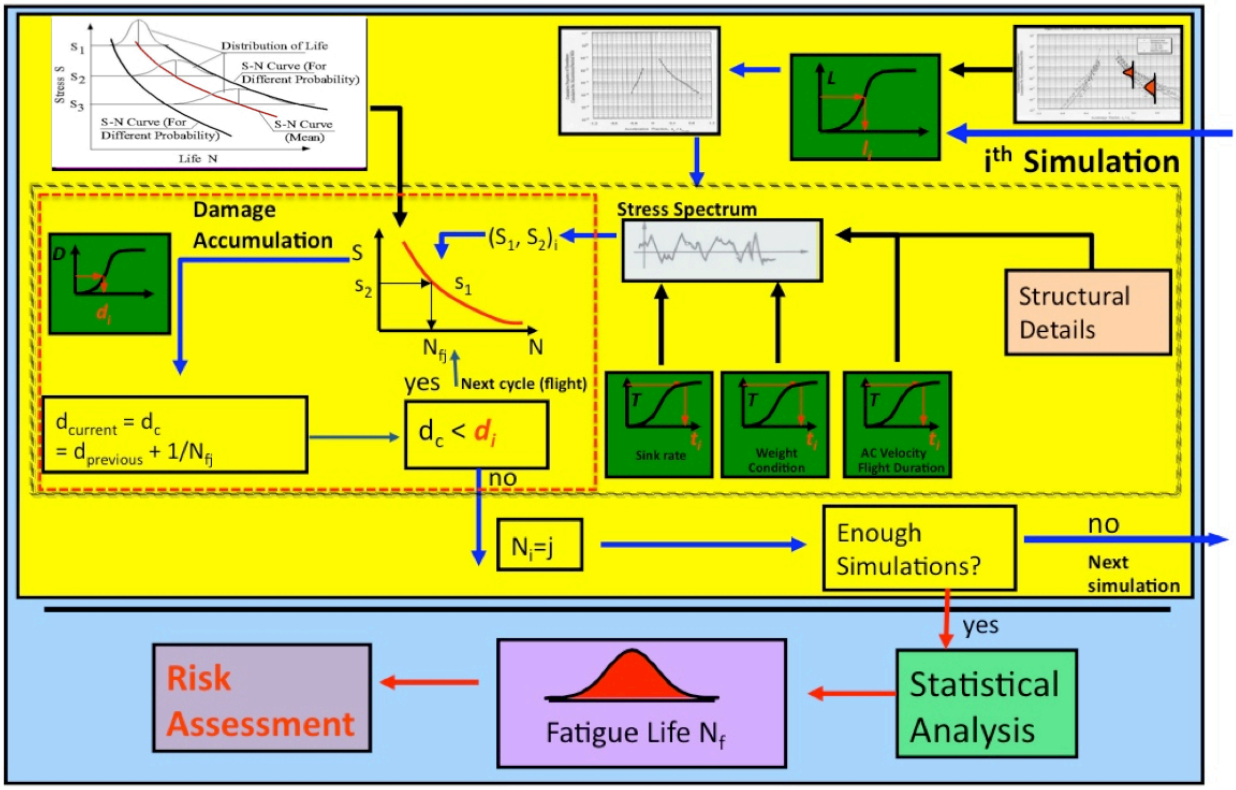

Figure 1 Schematic Risk assessment Methodology

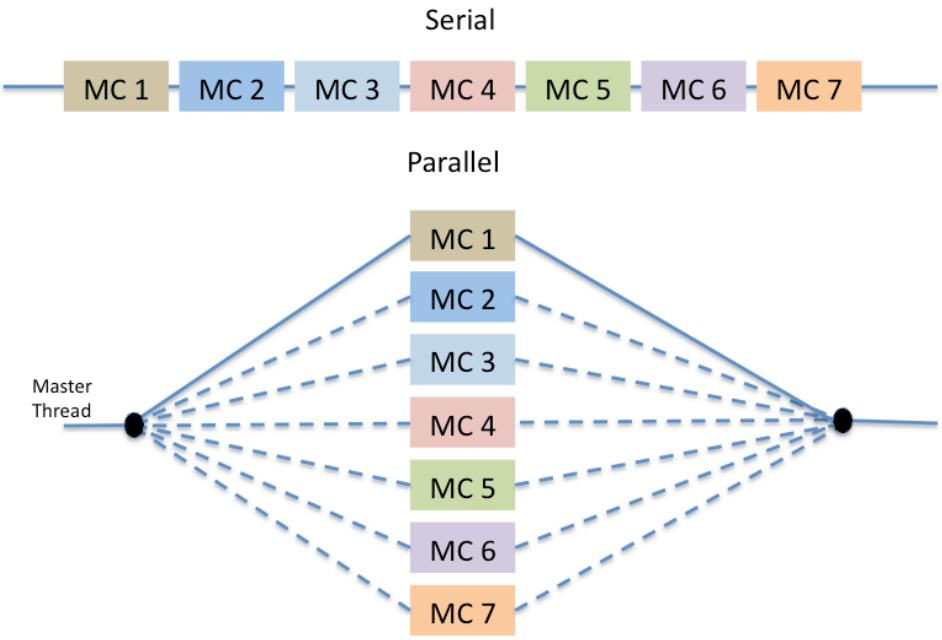

Figure 2 Serial Execution and Parallel Model

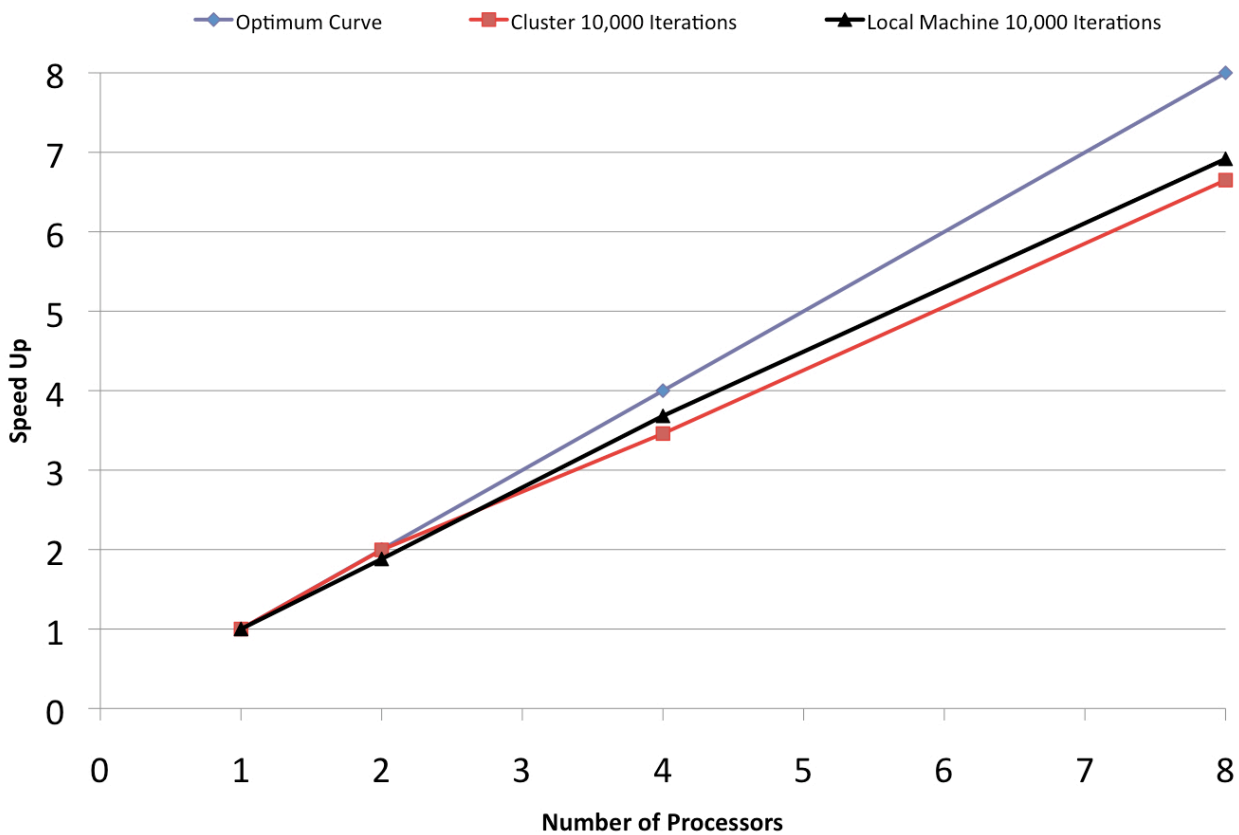

Figure 3 Number of processors vs Speed Up Using OpenMP

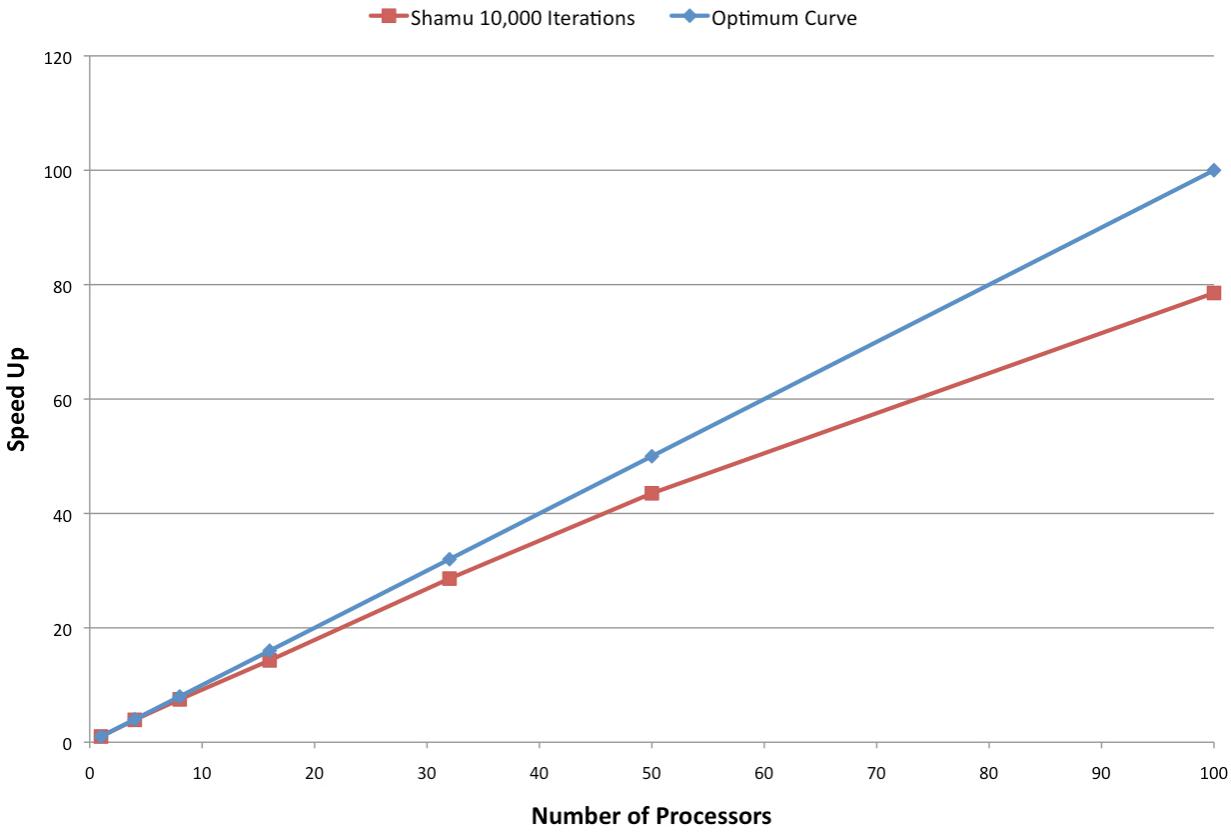

Figure 4 Number of processors vs Speed Up Using MPI