Parallel CFD Computing Using Shared Memory OpenMP

Hong Hu and Edward L. Turner

Department of Mathematics
Hampton University, Hampton, VA 23668, USA
hong.hu@hamptonu.edu

Abstract. The eXtended Full-Potential (FPX) helicopter rotor Computational Fluid Dynamics (CFD) code in its reduced two-dimensional version is successfully converted into a parallel version. The FPX code solves the full potential equation using an approximately factored finite-difference scheme. The parallel version of the code uses Open Multi-Processing (OpenMP) directives as parallel programming tool. Open MP based parallel code is portable and can be compiled with Fortran compiler that supports the OpenMP Fortran standard. OpenMP based parallel code can also be compiled using non-parallel Fortran compiler into a serial executable. The performance study of the parallel code is made. The results show that OpenMP is easy to use and a very efficient parallel programming tool for the present problem. Keywords: Parallel Computing, Computational Fluid Dynamics

1 Introduction

Computational Fluid Dynamics is the one of the areas that needs super-fast computation power. The numerous calculations that are needed to execute CFD codes may require hours and even days of Central Processing Unit (CPU) time. Parallel computation using more than one CPU is highly considered in the field of Computational Fluid Dynamics. Parallel computation allows CFD codes to run fast, since the computational workload is distributed among computer processors.

There are two major approaches in multiprocessing parallel computational architectures: distributed memory where each CPU has a private memory, and shared memory where all CPUs access common memory. Different parallel processing architectures give the different parallel performance characteristics, and different applications perform differently on different architectures. Today's new multiprocessing parallel computers are a combination of the best parts of shared- and distributed- memory architectures, such as distributed shared- memory system of Silicon Graphics (SGI) Origin 2000. Parallel program can be developed on the SGI Origin 2000 using either a shared- memory or distributed- memory architecture.

Open MP shared-memory parallel processing is employed in the present work to develop a parallel version of a helicopter rotor FPX CFD code in a reduced two-dimensional form. This paper presents the work on the parallel code development along with the performance analysis of the resulting parallel code.

2 Methodology of the FPX CFD Code

While in the fixed-wing aerodynamic computational community increasingly expensive and complex Euler and Navier-Stokes methods have been used recently, potential methods are still major analysis tools in rotary-wing aerodynamics computational community. The FPX [1] rotor code is an efficient and accurate potential method in this field. The code represents an industry standard for rotary-wing computations. The FPX code is a modified and enhanced version of Full-Potential Rotor (FPR) code [2]. The code (either FPX or FPR) solves three-dimensional unsteady full-potential equation. The code has been used in various helicopter hover and forward flight cases. The application of the code produces excellent results.

The unsteady, three-dimensional full-potential equation in strong conservative form in blade-fixed body-conforming coordinates (ξ, η, ζ, τ) is written as

$$\frac{\partial}{\partial \tau}\left(\frac{\rho}{J}\right)+\frac{\partial}{\partial \xi}\left(\frac{\rho U}{J}\right)+\frac{\partial}{\partial \eta}\left(\frac{\rho V}{J}\right)+\frac{\partial}{\partial \zeta}\left(\frac{\rho W}{J}\right)=0 \tag{1}$$

with

$$\rho=\{1+\tfrac{\lambda-1}{2}[-2\Phi_\tau-(U+\xi_t)\Phi_\xi-(V+\eta_t)\Phi_\eta-(W+\zeta_t)\Phi_\zeta]\}^{\frac{1}{\gamma-1}} \tag{2}$$

where Φ is the velocity potential, U, V and W are contravariant velocity components, ρ is the density, and J is the grid Jacobian.

The FPX/FPR codes solve Eq. (1) using an implicit finite-difference scheme, where the time-derivative is replaced by a first-order backward differencing and the spatial-derivatives are replaced by second-order central differencing. The resulting difference equation is approximately factored into three operators L_ξ, L_η and L_ζ in ξ, η and ζ directions, respectively,

$$L_\xi L_\eta L_\zeta(\Phi^{n+1}-\Phi^n)=RHS \tag{3}$$

The detail of the scheme is presented in [1,2].

The FPX is the substantially modified version of the FPR code. Both entropy and viscosity corrections are included in the FPX code. The entropy correction potential formulation accounts for shock produced entropy to enhance physical modeling capabilities for strong shock cases. Either a two-dimensional or three-dimensional boundary layer model is coupled with the FPX code to account for viscosity effects.

In addition, an axial flow capability is added into the FPX code to treat tilt-rotors in forward flight.

3 Parallel Implementation

3.1 Background

SGI Origin 2000 High Performance Computer (HPC) is chosen as the platform to use for developing the parallel code. The SGI Origin 2000 is a distributed shared-memory system, with hardware designed like distributed-memory architecture. However, the system keeps track of which memory space holds which variable, therefore parallel programs can be developed using either a shared or distributed-memory model on the SGI Origin 2000. The programmer can use either Message Passing Interfaces (MPIs) for distributed memory programming, or OpenMP for shared-memory programming, or some combinations of both to best suit the application [3].

MPI has become accepted as a portable style of distributed-memory parallel programming, but has several significant weaknesses that limit its effectiveness and scalability [4]. Message passing in general is difficult to program and doesn't support incremental parallelization of an existing sequential program. The MPI is therefore not chosen for this work.

Shared-memory parallel programming directives have not been standardized in the industry before the introduction of OpenMP. An earlier standardization effort was never formally adopted. Thus, vendors have each provided a different set of directives, very similar in syntax and semantics, and each used a unique comment or programming notation for "portability". OpenMP consolidates these directive sets into a single syntax and semantics, and finally delivers the long-awaited promise of single source portability for shared-memory parallelism [5]. OpenMP is a specification for a set of compiler directives, library routines, and environment variables for specifying shared memory parallelism. The OpenMP is available for both Fortran and C/C++ languages. The FPX rotary code was written in Fortran, therefore OpenMP for Fortran is used as the parallel-programming tool.

OpenMP directives are portable and can be compiled using non-MIPSpro Fortran Compiler that supports the OpenMP Fortran standard. The parallel code developed on SGI Origin 2000 can be executed on the Sun Supercomputer and IBM Power 3 computer, for examples.

Fig. 1 gives an example of using OpenMP directive, where "C$OMP PARALLEL DO" directive instructs the parallel Fortran compiler to compile the loop into parallel executable. It should be mentioned that every OpenMP directive starts with the word "C$OMP".

```
C$OMP PARALLEL DO DEFAULT(SHARED),PRIVATE(J)
      DO 100 J = 1, JMAX
      A(J) = B(J)*C(J)
100   CONTINUE
C$OMP END PARALLEL DO
```

Fig. 1. Example of an OpenMP implementation

It is seen that the OpenMP directives are essentially command line options specified within the source code. Parallel version of the code can be compiled using non-parallel Fortran compiler. In non-parallel Fortran compiler, these OpenMP directives are treated as comment lines. Thus the code is portable between parallel and non-parallel compilers.

3.2 OpenMP Implementation on the FPX Code

The FPX rotor CFD code is converted to a parallel version by using OpenMP parallel directives. The code is about 13,000 lines in length, which is a reduced version of the current FPX release version (which is about 30,000 lines in length). The most of the developmental work is performed on NASA-LaRC's Origin 2000 that has a total of 6 processors, while the performance study is made on the U.S. Army's Origin 2000 that has a total of 112 processors.

Parallelization is the process of analyzing sequential codes for parallelism and restructuring them to run efficiently on multiprocessor computers by distributing the computational workload among the processors. Parallelization can be done automatically or manually. Before manual parallelization, Auto-Parallelizing Option (APO) is used to parallelize the code to determine if APO works for the FPX code. APO is a compiler extension that invokes the MIPSpro auto-parallelizing compilers, and automatically generates code that distributes the computational workload among processors. It is found that APO works fairly well on the FPX code when no more than 16 processors are used. The APO automatically parallelizes about 83% of the computational workload. However, when more than 16 processors are used, the APO produces a totally wrong solution. It is also noticed that Auto-Parallelizing Option fails to produce a parallel source code. Therefore, it is impossible to debug the code generated by APO and further hand-code the program for more efficient parallelization, since the source code in parallel version cannot be generated using APO.

As a consequence, manual parallelization through hand-coding become necessary. Among a total of 42 subroutines in the FPX code, the parallelization is done on those subroutines that carry non-negligible amount of computation workload, that is, on the subroutines that carry over 1% of the total CPU time. Manual parallelizing the code using OpenMP is an easy task, sometimes, by simply inserting the parallel directives. Fig. 2 is an example of how a DO-loop can be parallelized using PARALLEL DO

OpenMP Directive, where parallel directive instructs the compiler to parallelize the loop allowing the computational workload to be distributed among multiple CPUs.

```
C$OMP PARALLEL DO DEFAULT(SHARED),PRIVATE(J,K,HU)
      DO 200 J = 1, 1000
      DO 100 K = 1, 250
      HU = A(J) + B(K)
      HT(J,K) = 0.5 * (X(J+1,K,1) - X(J-1,K,1))+HU
100   CONTINUE
200   CONTINUE
C$OMP END PARALLEL DO
```

Fig. 2. Example of how a Do-loop is parallelized

For multiprocessing to work properly, however, the iterations or order of the execution within the loop must not depend on each other. The variable in the loop must standalone and produce the same answer regardless of the order of execution. Loops that dependents on the order cannot be parallelized. If a loop cannot be parallelized in its original form it may be rewritten to run wholly or partially in parallel.

In a Fortran program, memory locations are represented by variable names [5]. To determine whether a particular loop can be parallelized, studying the way variables are used is made throughout many loops in the FPX code. The essential approach to parallelize a loop correctly is to make sure that each iteration of the loop is independent of all other iterations. If a loop meets this condition, then the order in which the iterations are executed in the loop is not important. The iterations can be executed backward or even at the same time, and the answer will still be the same. This property is captured by the notion of data independence [6]. Based on these principles, some parts of the code are rewritten so that the parallelization is done either wholly or partially.

In addition to the PARALLEL DO directive, there are many other OpenMP parallel directives that can be used to parallelize a code. PARALLEL, SECTIONS, and DO directives are also used in this parallel version of FPX code, for example.

4 Parallel Performance Analysis

After successfully converting the FPX code into a parallel version using the OpenMP parallel directives, a series of runs of the code in both serial and parallel versions is made to study the performance of the parallel computation. For parallel version, a different number of CPUs is used for executing the FPX code. The performance analysis is made on varying computational workload by varying the computational mesh size. The mesh sizes of 80x25, 160x49, and 320x97 grid points are used. These cases take from 42 seconds to about 1 hour CPU time in both serial and parallel versions of the code with a single CPU. Both serial and parallel versions of the code produce the same solutions. The parallel version of the code with one CPU runs as fast as the serial version of the code.

Table 1. Computational performance of the parallel code

No. of CPUs (n)	Problem Size in terms of Number of Grid Points					
	80x25		160x49		320x97	
	CPU Time in Seconds	SpeedUp	CPU Time in Seconds	SpeedUp	CPU Time in Seconds	SpeedUp
1	42	1.0	448	1.0	3,517	1.0
2	25	1.7	247	1.8	1,866	1.9
4	17	2.5	147	3.0	1,178	3.0
8	13	3.2	108	4.1	691	5.1
16	14	3.0	93	4.8	519	6.8
32	14	3.0	93	4.8	496	7.1

Table 1 details the performance results. Up to 32 processors are used for parallel computations. In addition to CPU time, $SpeedUp$ is also given for each run in the table. $SpeedUp(n)$ is defined as the ratio of CPU time with 1 processor to that of n processors, or, $SpeedUp(n) = CPUTime(1)/CPUTime(n)$. If all 100% computational workload were parallelized and there were no communication overhead among processors, $SpeedUp(2) = 2$, theoretically. However, there is always some part of the code's computational workload (such as I/O statements) that has to be carried out serially by a single processor. This sets the lower limit on code CPU run time. The fraction of the computational workload that is parallelized can never be 100%. Moreover, there is less and less benefit from each added CPU after a certain point due to hardware constraints.

The data from Table 1 are presented in Figs. 3-6. Fig. 3 gives CPU time and $SpeedUp$ for the problem with $80x25$ grid points. This case takes 42 CPU seconds with a single CPU. It is seen that up to 8 CPUs can be used efficiently, and after this point adding more CPUs has no benefit at all. Maximum $SpeedUp$ is 3.2 when 8 CPUs are used.

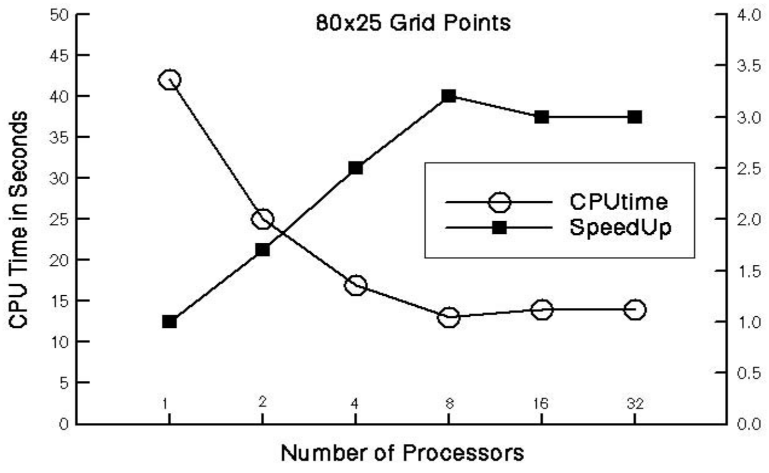

Fig. 3. CPU time and SpeedUp for the problem with 80x25 grid points

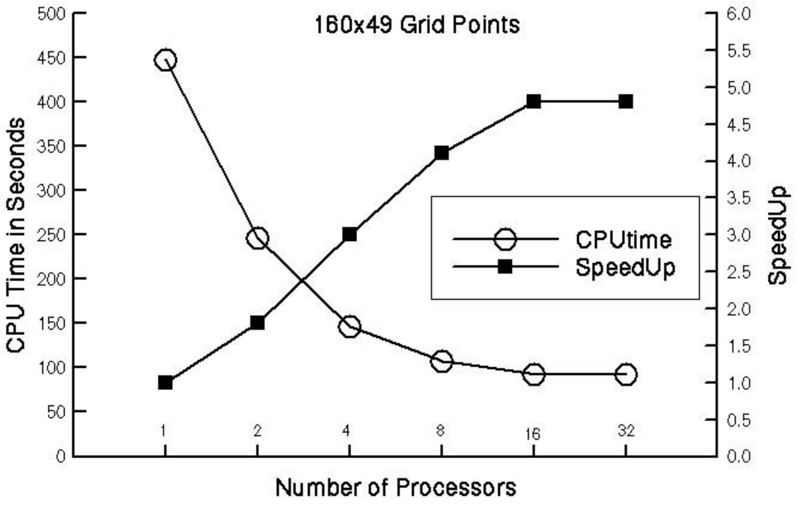

Fig. 4. CPU time and SpeedUp for the problem with 160x49 grid points

Fig. 4 gives CPU time and *SpeedUp* for the problem with 160x49 grid points. This case takes 448 second to execute on one CPU. It is seen that up to 16 CPUs can be efficiently used to achieve a maximum *SpeedUp* of 4.8 due to larger computational workload than the previous case with 80x25 grid points. Fig. 5 gives CPU time and $SpeedUp$ for the problem with 320x97 grid points. With this mesh size, the code takes about1 hour CPU time to execute on single CPU. It is seen once again from this figure that when the computational workload increases, increasingly more CPUs can be efficiently used. It is seen that all 32 CPUs can be efficiently used for parallel computation to achieve a maximum $SpeedUp$ of 7.1. Using the value of $SpeedUp(2)$ for this case, the fraction of the computational load that is parallelized is calculated to be 95%, which is considered to be substantial.

Finally, Fig. 6 gives a comparison of $SpeedUp$ with varying problem sizes and number of CPUs. The results are self-explanatory. For small computational problem (for example, with 80x25 grid points) less number of CPUs can be efficiently used; with the increase of the computational workload, number of CPUs that can be efficiently used increases also.

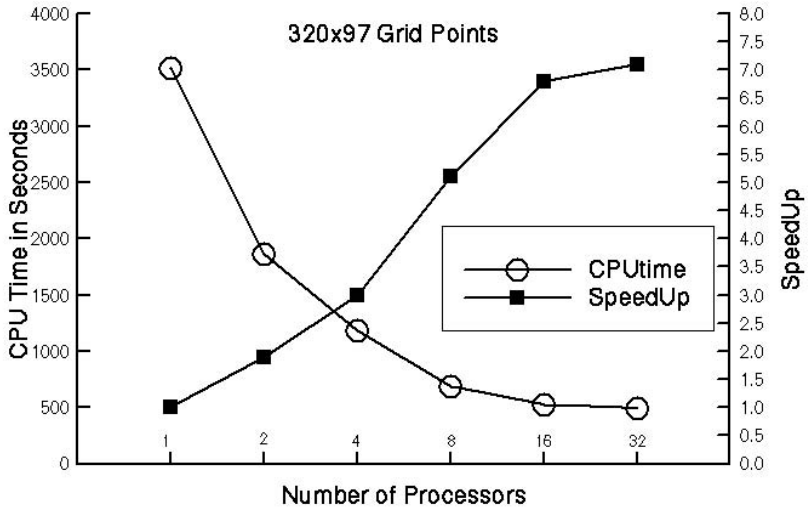

Fig. 5. CPU time and SpeedUp for the problem with 320x97 grid points

5 Conclusion

The eXtended Full-Potential (FPX) helicopter rotor Computational Fluid Dynamics code in its reduced two-dimensional version is successfully converted into a parallel version. The parallel version of the code uses OpenMP directives as parallel programming tool. OpenMP based parallel code is portable and can be compiled with Fortran compiler that supports the OpenMp Fortran standard. OpenMp based parallel code can also be compiled using non-parallel Fortran compiler. As a consequence, no separate parallel and serial versions of the code are needed, the maintenance cost of the code is thus reduced and the portability of the code between parallel and non-parallel computers increases.

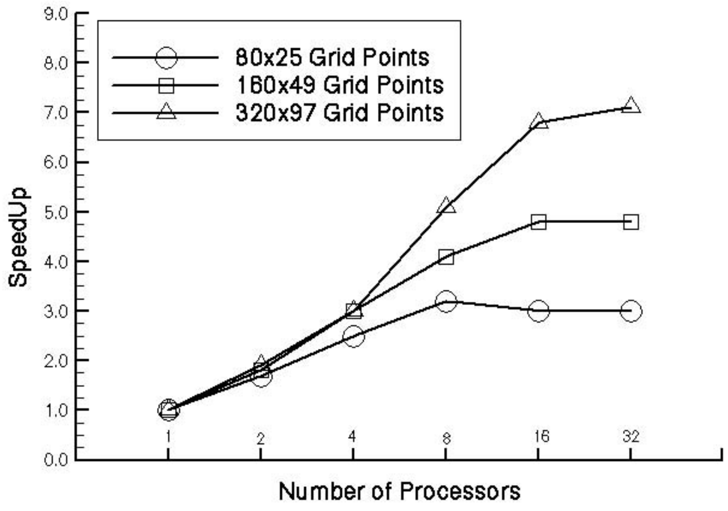

Fig. 6. Comparison of SpeedUp for the problems of all sizes

A performance study of the parallel code is made. From the research presented here, it is concluded that:

(1) Based on the *SpeedUp* results presented here, it is believed that no less than 95% of the computational workload is parallelized; unparallelized part of computational workload may mainly due to I/O statements of the code and the internal grid generator.

(2) For the smallest computational problem tested here, the one with 80x25 grid points, no more than 8 CPUs can be used for efficient parallel computation.

(3) When the computational workload increases, the number of CPUs that can be used efficiently for parallel processing increases also. For example in the case of 320x97 grid points, which is typical for CFD computations, all 32 CPUs can be used efficiently for parallel computation to achieve a maximum *SpeedUp* of 7.1. Scalability increases with the increase of the computational workload.

(4) OpenMP is easy to use and a very efficient parallel programming tool for the present problem. The method is recommended for use in the future work on developing a parallel version of the full three-dimensional FPX code.

Acknowledgement

This work is supported by the NASA Grant NAG-2-1331 from Ames Research Center under FAR program with Dr. Roger Strawn as the Technical Officer and with Dr. Henry Jones of Langley Research Center as a local Point Of Contact. Their support and advice made this work challenging and fulfilling. The computational resources are provided by a grant of HPC time from the DoD HPC Center, ERDC Major Shared Resource Center, and by NASA Langley Research Center.

References

1. Bridgeman, J. O., Prichard, D., Caradonna, F. X.: The Development of A CFD Potential Method for the Analysis of Tilt-Rotors. Presented at the AHS Technical Specialists Meeting on Rotorcraft Acoustics and Fluid Dynamics, Philadelphia, PA, (1996).

2. Strawn, R. C., Caradonna, F. X..: Conservative Full Potential Model for Unsteady Transonic Rotor Flows. AIAA Journal, Vol. 25, No. 2, (1987) 193-198.

3. Breshears, C. P.: Four Different Parallel Architectures- Which One Is Best? The Resource, U.S. Army Engineer Research and Development Center Newsletter, Spring (2000).

4. OpenMP - Frequently Asked Questions. http://www.openmp.org.

5. MIPSpro Fortran 77 Programmer's Guide - OpenMP Multiprocessing Directives. http://techpubs.sgi.com/library.

6. Fortran 77 Programmer's Guide - Fortran Enhancements for Multiprocessors. http:// techpubs.sgi.com/library.