VOL. 12, NO. 8, APRIL 2017 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences


www.arpnjournals.com

PARALLELIZATION OF THE MOVING PARTICLE PRESSURE MESH (MPPM) FLUID FLOW SOLVER BY USING OPENMP

Abdalla M. E. Ibrahim, K. C. Ng and M. Z. Yusoff
Centre of Fluid Dynamics, Universiti Tenaga Nasional, Kajang, Selangor, Malaysia
E-Mail: elfatih@uniten.edu.my

ABSTRACT

The MPPM numerical method parallelization by utilizing the OpenMP environment is reported in this paper. The objective of this work is to reduce the MPPM method computation time by exploiting the otherwise idle processors and overall improve the efficiency of the method. The code solving the Rayleigh-Taylor instability model was originally written using Fortran77 and is considerably slow and was rewritten in C language then parallelized using OpenMP directives. The highest obtained speedup value resulting from OpenMP parallelization is found to be 2.3 which is achieved through utilizing 6 threads out of the available 12 threads. The work presents the possible performance improvement and the importance of addressing the sequential nature of the code in order to further enhance its efficiency.

Keywords: computational fluid dynamics, MPPM, OpenMP, parallel computing.

INTRODUCTION

Eulerian schemes are commonly adopted in fluid flow computation. The conventional Eulerian approach such as the Finite Volume (FV) method has gained significant popularity amongst the CFD practitioners. Owing to the existence of convective terms in the momentum equation, the Eulerian scheme is subjected to convective instability and wiggling solutions can be expected if the grid is not sufficiently refined in the case of high Reynolds number (if 2^{nd}-order central differencing scheme is used). Furthermore, Eulerian schemes are based on mesh which is a rigid factor [1]. Therefore, it is of crucial importance to have a numerical scheme that can negate the convective discretization.

Lagrangian schemes are an alternative way to solve the fluid flow governing equations, whereby the convective acceleration can be combined with the local acceleration term in the momentum equation (hence the total acceleration). Therefore, the fluid particles can travel based on the total acceleration in the Lagrangian manner. This technique has been proven to be effective in resolving interfacial flow whereby large deformation of fluid interface can be expected. Some examples of Lagrangian schemes are Smoothed Particle Hydrodynamics (SPH) introduced in [2], whereby the method was originally created to simulate astrophysical problemsas reported in [3]. However, a lot of numerical tunings are required in order to attain numerical stability.

Recently, a new Lagrangian method have been developed called the Moving Particle Pressure Mesh (MPPM) method reported in [4], which is free from any tuning parameters. In this method, the pressure is considered as an Eulerian field variable. MPPM has been proven to be effective in simulating single- and multi-fluid flow as reported in [5]. However, its application in large-scale problem is prohibited by the serial nature of the existing MPPM code.

The objective of this work is to enhance the running performance of the existing MPPM method as much as possible through the incorporation of OpenMP directives. Too, to study the effects of the variation of the number active threads and mesh density on speedup specifically

OPENMP

By default, all codes are treated as serial or 'sequential' codes by processors regardless of the amount of CPU cores existing. Unless these codes or programs are designed to purposely utilize existing cores. One method to almost effortlessly utilize additional cores and introduce speedup is to incorporate the OpenMP environment (openmp.org). OpenMP, through its directives, simply enables a task to be executed in a theoretically simultaneous manner by the team of threads it creates. A common use of OpenMP is to distribute loop iterations between threads which is what is mostly what has been applied in this work.

The sequential nature of N-body methods in general and MPPM especially makes it difficult to consider distributing subroutines amongst threads, seeing as how every step or subroutine depends on the outcome of the previous subroutine which will be shown further on in this paper.

Parallel computing has become the basis of high-end applications development. Most of these applications are designed based on MPI (Message Passing Interface) according to Amritkar *et al.* [6]. Even though MPI is of high maintenance and cost, it is still the standard option in the architecture design of SMP (Shared-memory Multi Processor). A more recent substitute to MPI is OpenMP. OpenMP is considerably flexible; based on the fact that it can utilize, much like MPI, Single Processor Multi Data (SPMD) as well as both functional parallelism and task parallelism all in the same program which was investigated by Huang *et al.* [7].

In previous studies, Amritkar *et al.* [6,8] have proven OpenMP to present further improvement if compared to MPI; the reason being that OpenMP threads, if tuned properly, are not bound to a distinct mode. Furthermore, OpenMP was arguably the preferred method

in designing N-body based solutions where the focus is on the distinct particle numbers [8].

Computational details MPPM

Widely known particle methods such as SPH, MPS and finite-volume particle method (FVPM) were reviewed in [4] and were all found to have shortcomings when it comes to simulating incompressible flow. This issue originated from the mishandling of pressure gradient calculation where the incompressibility constraint is concerned. To cater for this type of issue, and to avoid further computational trouble, Hwang [4] suggested the treatment of pressure as a field variable seeing how it is not a property of fluid particles.

The process flow of the parallelized MPPM can be seen in Figure-1. The steps of the MPPM calculation can be summarized in the following:

1) Calculation of intermediate particle velocity $\vec{u}_p^*$ and its location $\vec{r}_p^*$ using:

$$\overrightarrow{u_p^*} = \overrightarrow{u_p^n} + \Delta t v \nabla^2 \overrightarrow{u_p^n} \quad (1)$$

$$\overrightarrow{r_p^*} = \overrightarrow{r_p^n} + \Delta t \overrightarrow{u_p^*} \quad (2)$$

2) Interpolation of Intermediate cell face velocity $u_e^* and v_n^*$ based on the intermediate velocity found from the first step.
3) Establishing the resulting pressure field by solving the following pressure equation:

$$2\left(\frac{\Delta x}{\Delta y}+\frac{\Delta y}{\Delta x}\right)P_P^{n+1} = \frac{\Delta y}{\Delta x}P_E^{n+1} + \frac{\Delta y}{\Delta x}P_N^{n+1} + \frac{\Delta x}{\Delta y}P_W^{n+1} + \frac{\Delta x}{\Delta y}P_S^{n+1} - \frac{\rho}{\Delta t}[(u_e^* - u_w^*)\Delta y + (v_n^* - v_s^*)\Delta x] \quad (3)$$

4) Lastly, updatingthe final particle velocity and particle location using:

$$\overrightarrow{u_p^{n+1}} = \overrightarrow{u_p^*} - \Delta t \frac{\nabla P_P^{n+1}}{\rho} \quad (4)$$

$$\overrightarrow{r_p^{n+1}} = \overrightarrow{r_p^*} - \frac{\Delta t^2}{\rho}\nabla P_P^{n+1} \quad (5)$$

Rayleigh-taylor instability model

The model simulated in this work is the Rayleigh-Taylor Instability (RTI) Multiphase model where a light fluid displacement occurs due to the heavy fluid introduced atop it. The fluids have a density of 500 kg/m^3 and 1000 kg/m^3 respectively. The initial testing included a grid of size 0.5x4 (WxH). The number of cells was varied between 6x48 and 10x80. The physical time period over which the simulation was conducted was set to 60 seconds.

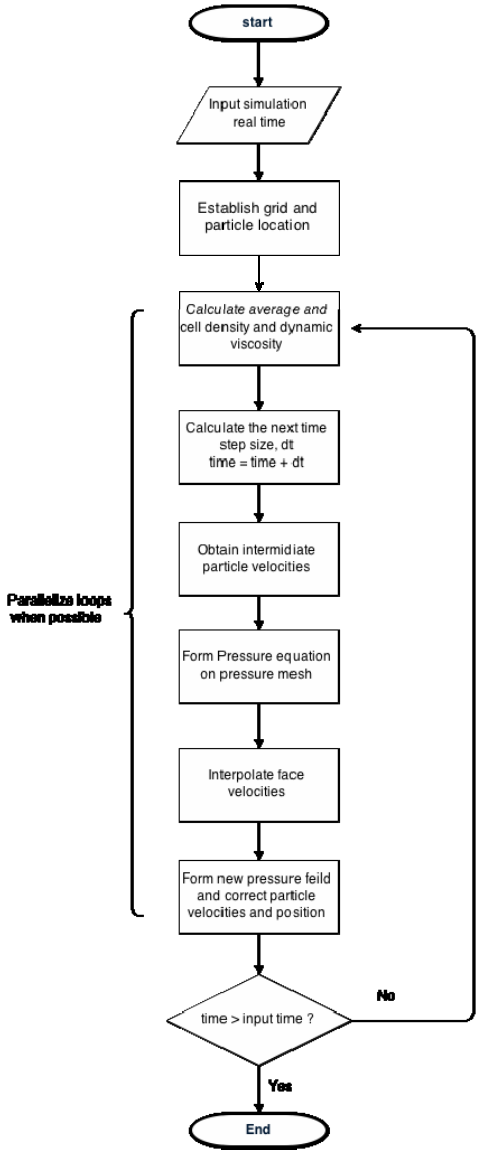

Figure-1. Process flow of parallelized MPPM method.

Serial code writing

Originally, the serial code was written using Fortran77 language. This language is considered to be quite slow on its own if compared to C language. This can be attributed to the sheer amount of statement labels involved in every loop. Also, the difficulty of detecting errors increases due to the fact that Fortran77 declares variables as they are introduced which means that any spelling mistakes will not be treated as an error by the compiler; making it all the more difficult to debug the code. The code was then re-written using C language with as many and as necessary simplifications possible. One advantage of Cover Fortran77 is dynamic array allocation,

this feature alone has improved the quality and speed of the code greatly. Quality improvement was in terms of easier and less costly computation requirements. The speedup results of the serial C code can be seen in the results section.

Technical details

In this work, the code was be developed and tested on a machine running on Intel® Xeon® Processor E5-2650 v2. 2.6 GHz. This processor consists of only 6 physical cores and 12 threads in total. Amritkar *et al.* [6] conducted the study using 256 cores; and John *et al.* [9] utilized up to 16348 cores to simulate numerous problems. However, it was found that if the number of threads handling a process was higher than 16, the speed of the process would decrease. This is due to the cache miss ratio being high in the case of 16 or less cores running in parallel and would have to wait longer to retrieve data once the number of cores is greater as concluded in [6].

Moreover, in order to obtain the actual speedup for any parallelization process, it is imperative that both codes are executed on the same machine. To satisfy this condition, both the Fortran77 code and the C code were run on the same machine and using the same compiler, Microsoft Visual Studio 2012®.

RESULTS

Serial speedup

The code for simulating the RTI model was first re-written in C language and then was parallelized using OpenMP directives. These are some of the directives and functions used and their brief descriptions:

- #pragma omp parallel: For parallel sections of the code, a thread team needs to be created. This directive is responsible for thread creation.
- omp_set_num_threads: This function sets the number of threads utilized by the subsequent parallel sections.
- #pragma omp section: Creates sections to be distributed amongst threads.
- #pragma omp parallel for: Distributes 'for' loop iterations between the team of threads. However, part of the parallelization process in OpenMP is to distribute tasks to threads upon request. This means that almost at all times the process is not sequential in nature which could be problematic if not treated properly. One method to accommodate sequential areas that exist within parallel loops is the 'critical' directive.
- #pragma omp critical: Used to synchronize and force sequence.
- Private: Used to localize variables for each thread when needed. Particle identity, *np*, is an example of a private variable for each thread where with each iteration each thread handles a different cell and searches for particles within it. Each thread then proceeds to have its own value for *np* which would not be possible if not for the 'Private' feature.

Results show great similarity as seen in Figure-2 which displays an enlarged representation of particle locations after simulation of 30 seconds (physical time). However, despite being visually identical, there exists a very minor difference in values. This is due to the way that the compiler handles different languages. For instance, when writing the C code, most variables were declared as double which has 18 decimals on average whereas Fortran77 only has 15. Eventually, the propagation of this variance causes said difference in some properties such as particle location.

Furthermore, the C code was focused on eliminating most of, if not all, the goto statements which are the main disadvantage of Fortran77 as well as utilize dynamic array allocation to provide rapid handling of array creation and deletion. The result of the serial C speed up can be seen in the next section along with the parallel speedup.

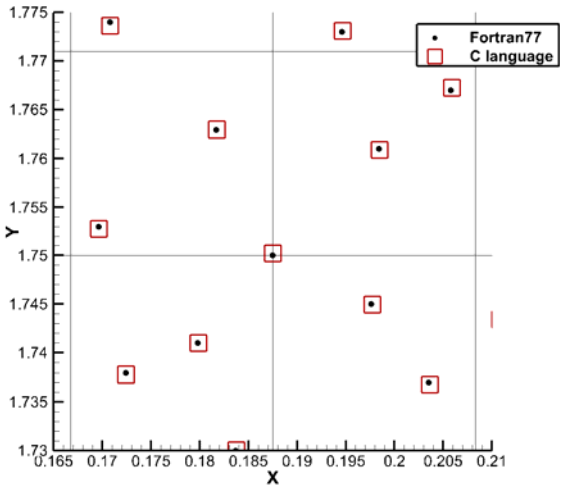

Figure-2. Comparison of particle locations produced by Fortran 77 and C.

Parallelization

OpenMP directives were introduced into the code where applicable and with emphasis on loop iteration distribution. The more common use of OpenMP is to divide whole subroutines to be executed by separate thread teams. However, the sequential nature of MPPM makes that all the more difficult. It can be seen Figure-1 that almost all calculation steps depend on the outcome of their respective preceding step or sub routine. Thus, the focus was on parallelizing loops. Even so, some difficulties arise due to the shared variables between threads. Nonetheless, Private and Critical directives were utilized to resolve such issues.

The speedup result of parallelization can be seen in **Error! Reference source not found.** as the bar chart represents the total process time consumed by each method in order to simulate 60 seconds of physical time for the RTI problem.

Speedup of N threads, $S(n)$ can be found using the following equation:

$$S(n) = \frac{Executiontimeusingoneprocessor\ (T_s)}{ExecutiontimeusingNprocessors\ (T_n)} \quad (6)$$

Where T_s is the execution time for one thread and T_n is the execution time after parallelization [10]. However, it must be mentioned that the speedup of any process depends greatly on how much of said process is sequential. From there, and by using equation (7) named Amdahl's law which governs speedup calculation:

$$S(n)_{max} = \frac{1}{F + (1-F)/N} \quad (7)$$

Where *F* is the percentage of the process or code that is sequential. What Amdahl's law indicates is that regardless of the number of processors used, as long as the process itself is mostly sequential, the anticipated speedup will be low. Also, in general, as the number of processors *N* increases the lower the price/performance ratio will fall.

The grid size over which the simulation took place was varied between 6x48 and 10x80 cells and after simulating 60 seconds it was found that indeed the parallelization of the code offered notable speed if compared against both serial alternatives. However the when comparing the speedup values seen in Table-1 where both Serial C and Parallel C speedup is put up against Fortran77, it can be seen that despite the high number of threads involved (N = 12) the speedup for the parallel code for the finer grid is 2.61. This means that the parallel C code is 2.61 times faster than the Fortran77 code and double of that for a smaller grid size.

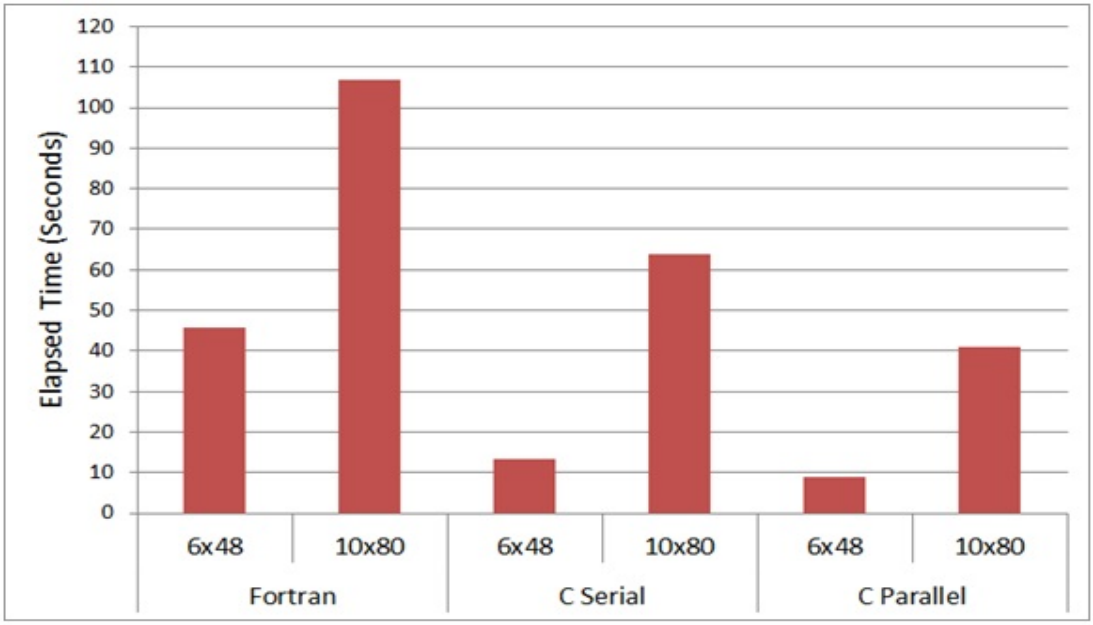

Figure-3. Comparison of time spent simulating the RTI model for 60 seconds of physical time using Fortran77, serial C and parallel C (12 threads).

Table-1. Speedup calculation.

Physical time	C Parallel Vs Fortran77		C Serial Vs Fortran77		C Parallel Vs C Serial	
	6x48	10x80	6x48	10x80	6x48	10x80
60 seconds	5.17	2.61	3.40	1.67	1.52	1.56

However, from **Error! Reference source not found.** and from Amdahl's law we are able to deduce that after parallelizing the loops where applicable, roughly 60% of the code is considered sequential. Nonetheless, the current speedup is noteworthy if compared against the original code. Further improvement to the performance is anticipated with additional tuning for the parallelization process.

In order to obtain a clear view of the scalability of the system, the effect of the number of threads processing on the speedup was investigated. The number of threads in the thread team was varied from 1 (serial) to the maximum number available which is 12 threads. This is performed at the beginning of the code using omp_set_num_threads**. Error! Reference source not found.**demonstrates the speedup obtained by comparing computation time for the parallelized against the serial code for each respective grid size.

The graph exhibits an expected behavior where the speedup increases with number of threads involved up to the point where it reaches the highest possible value of

2.3 for the 6x48 grid and 1.64 for the 10x80 grid. The highest speedup value was found to typically occur when only 6 threads are utilized. As mentioned previously, the machine on which the simulation was run operates using 6 physical cores which 'spawn' 12 logical threads. We believe that while considering the current parallelization structure, the time consumed by the threads to send and retrieve data to and from the shared memory affects the overall process time. This is apparent from the drop of speedup value as the number of threads included increases. The speedup drop was nonetheless anticipated based on Amdahl's Law.

CONCLUSIONS

This work aimed to improve and investigate the performance of the MPPM method when parallelized using Open MP. Re-writing the code in C language proved to have reasonably enhanced the running performance of the code prior to the code parallelization. The parallelized C code achieved a maximum speedup value of 2.3 for the 6x48 case. Generally, the highest values of speedup were accomplished when utilizing the exact number of physical cores existing in the processor so far. However, we believe that fine-tuning the parallelization structure will further improve the results.

Future works will focus on refining the parallelization structure with emphasis on MPPM segmentation possibilities to enable more efficient parallel sectioning. Additional numerical problems will be solved using parallel MPPM and will have their performance investigated. Moreover, other numerical methods based on MPPM will be parallelized and investigated.

ACKNOWLEDGEMENTS

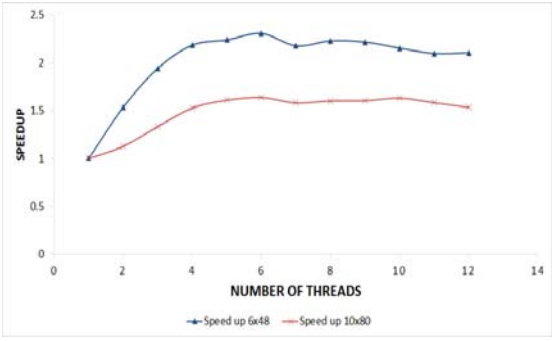

Figure- 4. Speedup results of parallel C against serial C.

The financial supports provided by the ministry of education Malaysia (Project no: FRGS/2/2013/TK01/UNITEN/02/1) and the Ministry of Science, Technology and Innovation (MOSTI) Malaysia (Project no: 06-02-03-SF0258) are greatly acknowledged and appreciated.

REFERENCES

[1] Chieh-Sen, H., Arbogast, T. and Qiu, J. 2012. An Eulerian–Lagrangian WENO finite volume scheme for advection problems. Journal of Computational Physics. 231(11): 4028–4052.

[2] Gingold, R. A. and Monaghan, J. J. 1977. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. Monthly Notices of the Royal Astronomical Society.181: 375-389.

[3] Lucy, L. B. 1977. A numerical approach to the testing of the fission hypothesis. The astronomical journal.82:1013-1024.

[4] Hwang, Y.H. 2011. A Moving Particle Method with Embedded Pressure Mesh (MPPM) for Incompressible Flow Calculations. Numerical Heat Transfer, Part B: Fundamentals. 60(5): 370-398.

[5] Ng, K.C., Yusoff, S. M., Hwang, Y.-H., Sheu, T. W.-H., and Yusoff, M.Z. 2014. Simulation of Unsteady Incompressible Flow using a New Particle Method. International Meeting on Advances in Thermofluids (IMAT), Swiss-Garden Hotel, Kuala Lampur. Paper 10145.

[6] Amritkar, A., Tafti, D., B, R. L., Kufrin, R. and Chapman, B. 2012. OpenMP parallelism for fluid and fluid-particulate systems. Parallel Computing. 38: 501-517.

[7] Huang, W. and Tafti, D. 2004. A parallel adaptive mesh refinement algorithm for solving nonlinear dynamical systems. International Journal of High Performance computing Applications.18(2): 171-181.

[8] Amritkar A., Deb, S. and Tafti, D. 2014. Efficient parallel CFD-DEM simulations using OpenMP. Journal of Computational Physics. 256: 501-519.

[9] John, B., Emerson, D. R., and Gu, X.-J. 2015. Parallel Navier–Stokes simulations for high speed compressible flow past arbitrary geometries using FLASH. Computers & Fluids. 110: 27-35.

[10] Amdahl, G. M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. Spring Joint Computer Conference. Sunnyvale, California. Retrieved: http://dl.acm.org/citation.cfm?doid=1465482.1465560