Turkish Journal of Electrical Engineering and Computer Sciences

Volume 24 | Number 5 Article 90

1-1-2016

A Monte Carlo-based Poisson's equation solver parallelized with Coarray Fortran

NEVSAN ŞENGİL

ÖZGÜR TÜMÜKLÜ

MEHMET CEVDET ÇELENLİGİL

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2016) 24: 4545 – 4553

doi:10.3906/elk-1402-60

A Monte Carlo-based Poisson's equation solver parallelized with Coarray Fortran

Nevsan ŞENGİL[1,*], Özgür TÜMÜKLÜ[2], Mehmet Cevdet ÇELENLİGİL[2]
[1]Department of Astronautical Engineering, Faculty of Aeronautical and Astronautical Sciences, University of Turkish Aeronautical Association, Ankara, Turkey
[2]Department of Aeronautical Engineering, Faculty of Engineering, Middle East Technical University, Ankara, Turkey

Received: 08.02.2014 • **Accepted/Published Online:** 18.08.2015 • **Final Version:** 20.06.2016

Abstract: Poisson's equation is found in many scientific problems, such as heat transfer and electric field calculations. Although many different techniques are involved in solving Poisson's equation, we focused on the Monte Carlo method (MCM). We preferred the MCM not only because of its simple algorithm but also for its excellent parallel efficiency. Parallelization is one of the most effective techniques for reducing computation time. Among many parallelization paradigms, such as OpenMP (open multiprocessing), MPI (message passing interface), and PGAS (partitioned global address space), we adopted the PGAS-based Coarray Fortran (CAF). In this paper, we demonstrated that parallelization of Poisson's equation solver with CAF was quite painless. After parallelization, we solved Poisson's equation for a nonrectangular domain. We started with a workstation that consisted of 8 cores and we continued with a Cray supercomputer of up to 512 cores. The results of the parallel solvers were validated using exact solutions. We demonstrated that the error was less than 1.6%. Additionally, solution times and speedups of the CAF-based solver were compared with a solver that utilized MPI or OpenMP. OpenMP was not able to compete against CAF and MPI because of the "while" loop restriction. The CAF-based solver performed slightly better (7.5%) than the MPI provided that core numbers were between 2 and 32. However, CAF and MPI performed similarly for higher numbers of cores.

Key words: Poisson's equation, Monte Carlo methods, Coarray Fortran, message passing interface, OpenMP

1. Introduction

Computer simulations have become the third scientific method, after experimental and theoretical methods. Occasionally computer calculations are time-consuming. In order to perform computer calculations within a reasonable time frame parallel programming is used. However, parallel programming requires sophisticated multiprocessor platforms. In the early days of computing, parallel working platforms were expensive and their numbers were limited. With the advent of the electronic age, the numbers and computing capabilities of parallel platforms have considerably increased. Currently supercomputers consist of millions of cores [1]. Furthermore, dramatic improvements in the processing capabilities of PCs (personal computers) have made it possible to run parallel programs in economical PC environments, such as clusters. To meet the demand for economical parallel computing, microprocessor developers have started to create PCs with multiprocessors, that is, processors with multicores and multiunit graphic processors. Therefore, parallel programs on PCs have begun to be developed. However, developing highly efficient and bug-free parallel programs is complicated. Parallel programs should

*Correspondence: nsengil@thk.edu.tr

have 3 distinct features [2]. First, they should be easy to write. Second, their programming tools should be easy to understand for program developers. Finally, they should be executable on different parallel architectures.

The first parallel programming attempts resulted in the development of directive-based techniques, such as OpenMP (open multiprocessing). OpenMP is a parallelization tool based primarily on loop-level directives [3]. In this technique, programs are developed sequentially. However, the codes between OpenMP directives run concurrently. In OpenMP, all processors can reach and manipulate data using global memory. As a result, communication time is not wasted between the processors. Implementation of OpenMP is relatively easy, but it also has some significant shortcomings. First, partition of the global memory among the multiprocessors causes racing conditions and performance reduction. Second, this technique has little control over data layout. As a result, this parallelization tool is inadequate.

Subsequently the message-passing paradigm was developed. Each processor has its own memory unit and communication among the processors involves messages using high-speed data channels. The MPI (message passing interface) is the leading model of this paradigm. A 2-sided communication technique is implemented in the MPI. The MPI technique enables programmers to write portable, scalable, and high-performance parallel programs. Although there is no need to share global memory, in some cases communication time among the processors may increase the computation time considerably. Synchronization of computing and communication is the responsibility of the programmer. Additionally, MPI program debugging is complex. Consequently, programmers require compiler-based parallel programming tools instead of library-based ones.

Additionally, parallelism can be added into programming languages, such as CAF (Coarray Fortran) and UPC (Unified Parallel C). In these languages, a PGAS (partitioned global address space) model is employed. The advantage of this model is that each processor has its own memory unit [4]. At the same time, the processors can access each other's memory units when necessary.

In this study, we investigated the advantages of CAF as a parallelization tool in order to parallelize our sequential Poisson's equation solver. This Monte Carlo method (MCM)-based solver was developed by Sengil et al. [5] to be used in a particle-in-cell (PIC) solver. The Poisson equation should be solved as fast as possible in order to increase the efficiency of the PIC method. Parallelization is a leading method to speed up the calculations. Among many available parallelization techniques, we decided to implement CAF in order to minimize development and debugging time. At the same time, however, we wanted to evaluate the parallel performance of CAF in order to compare it with OpenMP and MPI. To the best of our knowledge, this study is the first attempt to implement CAF in a MCM-based Poisson's equation solver. In Section 2, we briefly define Poisson's equation and explain how the MCM was implemented. Furthermore, we modified our sequential solver to incorporate parallelism with CAF. Additionally, we modified the same solver using MPI and OpenMP for comparison purposes. In Section 3, we validate the results of the parallel Poisson's solvers with an analytical solution. Next we compare CAF, MPI, and OpenMP results in terms of solution times and speedups. Finally, we summarize our findings and analyze the advantages of CAF as a parallelization tool.

2. Implementation of CAF

2.1. MCM-based Poisson's equation solver

We first developed the MCM-based Poisson's equation solver as an integral part of our PIC simulation software. This software was developed to analyze plasma flows under the influence of electric fields. Later, we extended its use to heat transfer problems. Although various techniques are available to solve Poisson's equation [6–10], we focused on the MCM. The computation time of the MCM is generally longer than that of other methods

[11]. Nevertheless, this method is still used for computations in many fields of science as it has a number of distinct advantages. High-dimensional problems, complex boundary conditions, and complicated geometries can be effectively dealt with by the MCM, as shown by Shentu et al. [12]. Moreover, the efficiency of the MCM can be increased considerably when calculations are limited to a few important points instead of solving the whole computational domain [13]. Furthermore, it has been demonstrated that the MCM is superior to other methods, such as finite difference and finite element methods, if severe gradients exist near boundaries [14]. Finally, the MCM is so suitable to parallel computing that it is claimed as 'embarrassingly parallelizable' [15].

A 2-dimensional (2D) Poisson equation is given by:

$$\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = g(x, y). \tag{1}$$

In this study we solved Poisson's equation on a 2D-nonrectangular geometry, as illustrated in Figure 1. Here the physical domain's left, right, and bottom edges were parallel to the Cartesian axes, while the upper edge was a quadratic polynomial given in Eq. (2):

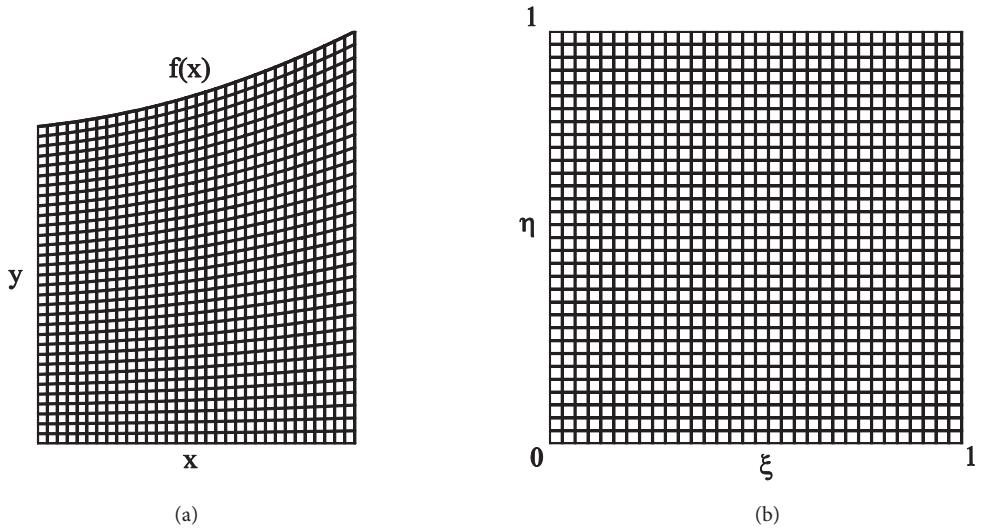


Figure 1. (a) 2D-nonrectangular physical domain. (b) Square computational domain.

$$f(x) = 0.2x^2 + 0.1x + 1. \tag{2}$$

In order to use uniform meshes the physical domain was transformed to a rectangular computational domain. Consequently, Poisson's equation was also transformed to the computational domain as follows:

$$\alpha \varphi_{\xi\xi} - \beta \varphi_{\xi\eta} - \chi \varphi_{\eta} + \delta \varphi_{\eta\eta} = h\left(\xi, \eta\right). \tag{3}$$

Coefficients $(\alpha, \beta, \chi, \delta)$ in front of the derivatives are given in [5].

2.2. Parallelization of Poisson's equation solver with CAF

CAF emerged as an important programming model for the development of scientific applications that were designed to be executed on high-performance parallel-processing platforms. CAF is an enhanced version of

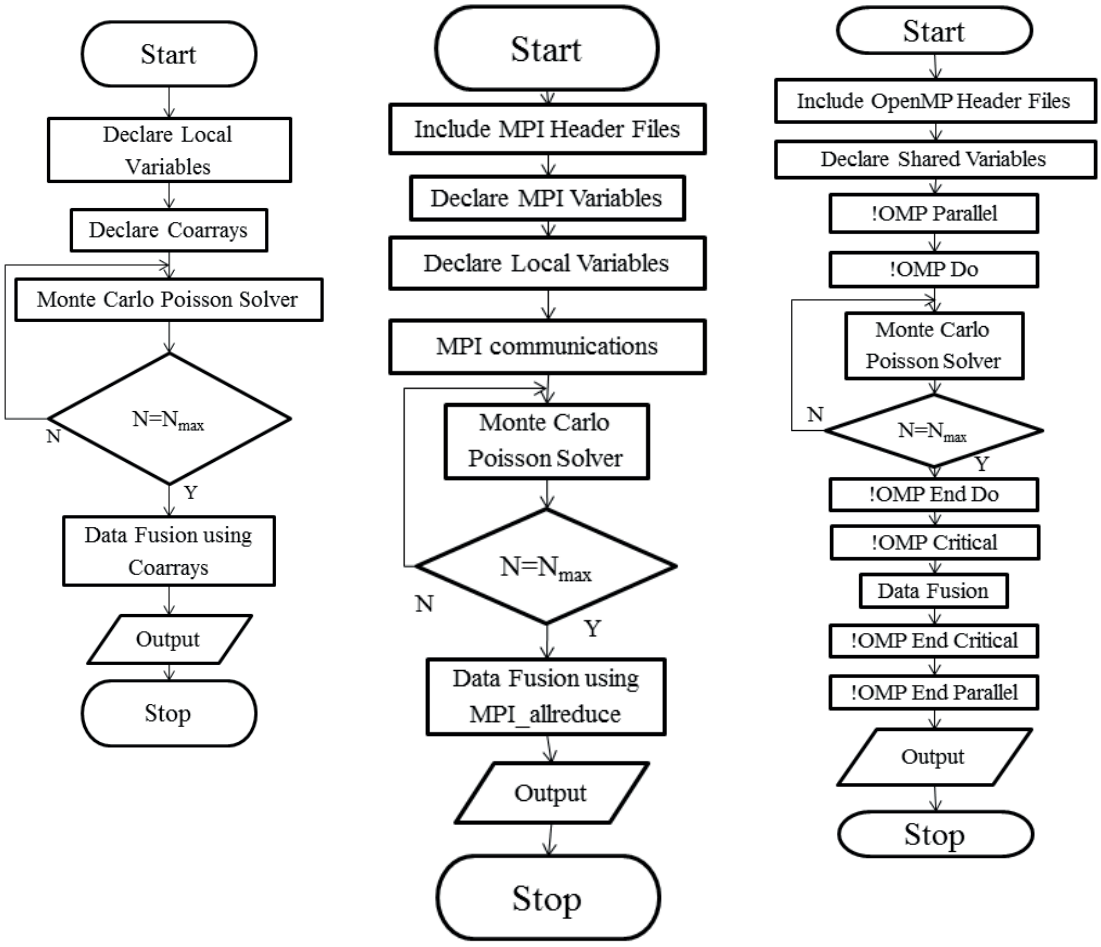
the Fortran programming language. Parallelization of Fortran was accomplished with a limited number of new rules and syntactic extensions [16]. CAF is designed to be compatible with both the distributed and shared memory architectures. To increase the communication efficiency among the processors, a single-program multiple-data (SPMD) model with a 1-sided communication technique is used [17]. In this model, both data and work are distributed on the parallel computational environment. In the CAF program, replications are called "images". The number of "images" can be different from the number of processors or cores. Each "image" works asynchronously while the compiler enforces optimization independently. The programmer is responsible for providing data transfer synchronization between the memory "images". In CAF, index numbers between square brackets point to another "image". These "images" contain their own private set of data objects. Developing parallel programs with CAF is simpler compared to MPI, with fewer lines of code without needing to sacrifice any performance. Numrich [18] revealed that the communication time for CAF was less than that of MPI. Coarfa et al. [2] discovered the performance of CAF was comparable to MPI on cluster-based architectures. Wallcraft observed that the application area of CAF was much wider than that of OpenMP [3]. However, these studies were problem-specific. More extensive studies are required to determine the exact parallel performance of CAF. In this study we intend to extend these studies further into the MCM-based Poisson's equation solver.

Our sequential MCM-based Poisson's equation solver was parallelized with CAF, MPI, and OpenMP respectively. Using the parallel performance advantage of the MCM, we intended to neutralize its low computational efficiency. Only a small number of changes were introduced in the original sequential solver. This solver was named "PMCS2D".

In the case of CAF, we started with the declaration of local variables. Next, global data structures were declared as coarrays, shown in Figure 2a. Then a predetermined number of samplings were realized for each grid point. This number was taken as 80,000 and 2,560,000 for a workstation and a CRAY supercomputer, respectively, in all cases. Following this, different random number generator sequences were initiated using "seed=(/this_ image()+a/)" and Fortran's intrinsic "call random_ seed(put=seed)" subroutine for each "image". Here "this-image()" and "a" represent processor number and an arbitrary number, respectively. Random numbers were generated using the intrinsic Fortran function "call random_ number(rann)". In the beginning and end of the calculation loop, "sync all" was used to synchronize the "images". The average of the gathered results was calculated in "image" number 1. To measure the solution time, we used Fortran's intrinsic "call cpu_ time(time)" subroutine.

In the case of MPI, we included MPI header files, MPI libraries, and MPI variables in the solver. After declaring the local variables, MPI communication was started. Different random number generator sequences were initiated using "seed=(/rank+a/)" and Fortran's intrinsic "call random_ seed(put=seed)" subroutine for each "rank". Here "rank" and "a" represent processor number and an arbitrary number, respectively. Random numbers were generated using the intrinsic Fortran function "call random_ number(rann)". Poisson's equation calculations were summed and averaged in rank number 1. To measure the solution time, we used Fortran's intrinsic "call cpu_ time(time)" subroutine. This process was very similar to the CAF structure and is given in Figure 2b.

In the case of OpenMP, we tested both "do" and "sections" constructs for work-sharing. We found that the "section" construct had lower efficiency figures than the "do" construct so we eliminated the "sections" construct. In the "do" construct option, parallelization was realized through the loop level. After interface declaration and shared variables, we determined the number of threads through the environment routine "omp_ set_ num_ threads". The OpenMP shared the work among the threads. We equalized the number of threads with



(a) (b) (c)

Figure 2. Monte Carlo method based Poisson's equation parallel solver flowcharts: (a) CAF, (b) MPI, (c) OpenMP.

the number of processors. When the iterations were completed, local data were added together and averaged in the "critical" construct. Finally, the "omp_ get_ wtime" routine was used to measure the solution time. The OpenMP version of the parallel Poisson's equation solver flowchart is given in Figure 2c.

3. Results

3.1. Validation

In order to validate PMCS2D, results were compared to the exact solution of Poisson's equation. In the present study, the right hand side (RHS) of Poisson's equation is given in Eq. (4):

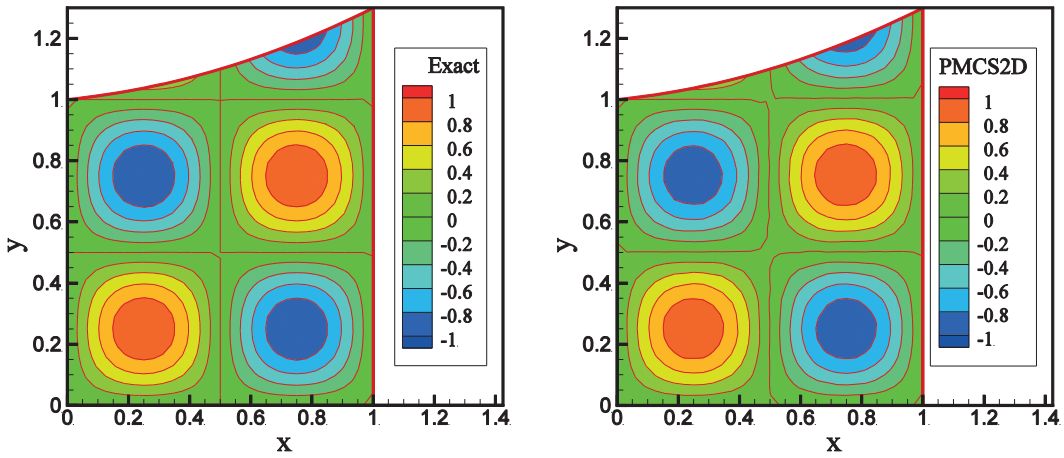
$$g\left(x,y\right) = \text{-}8\ \pi^2 \sin\left(2\pi \text{x}\right)\ \sin\left(2\pi \text{y}\right). \tag{4}$$

This equation was a benchmark function for measuring the efficiency of the Poisson's equation solvers. This function was also used in the Intel Fortran compiler as an example test case. In our PIC simulations, RHS values were discrete values computed through the calculation of charge densities on grid points [19]. PMCS2D

was validated with the results acquired from the exact solution given in Eq. (5):

$$\varphi = \sin(2\pi x)\sin(2\pi y)\,. \tag{5}$$

First, we used a workstation that had 2 Intel Xeon E5506 2.13 GHz processors with a quadcore. The total memory was 4 GB RAM. The Intel Visual Fortran Composer XE 2013 was used in these simulations. The PMCS2D produced very close results to the exact solutions as shown in Figures 3a and 3b.



(a) (b)

Figure 3. Comparison of the results: (a) exact solution, (b) solution of the PMCS2D.

Additionally, 5 different grid points in the computational domain were chosen for statistical purposes. Results taken from the PMCS2D calculations and exact solutions on these grid points were demonstrated and compared in Table 1. The solution of PMCS2D revealed that the results were similar to the exact solution. We observed 1.6% error at most.

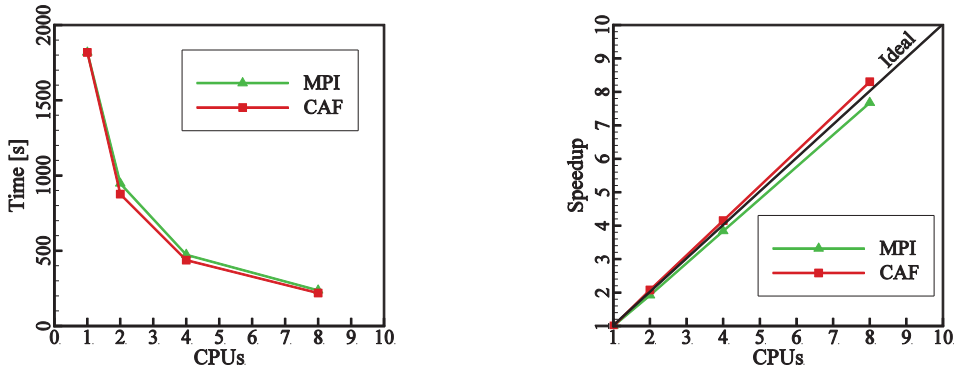
Table 1. Exact and PMCS2D results in the computational domain.

ξ	η	**Exact**	**PMCS2D**	**Error**
0.25	0.25	0.998266	0.981847	0.016
0.25	0.75	–0.95694	–0.950862	0.006
0.5	0.5	–1.94011E-007	0.00751333	0.008
0.75	0.25	–0.984427	–0.970429	0.014
0.75	0.75	0.634394	0.639635	0.005

3.2. Parallel solution times and speedups

Parallel performance of the CAF, MPI, and OpenMP techniques were compared in terms of the solution times and speedups. The same Poisson's equation solver was used for a fair comparison. No optimization option was chosen during program compilations. We realized our tests in 2 different computational environments.

In the first part, we compiled and ran PMCS2D on the same workstation environment used for validation. We used Intel's Fortran compiler and the MPICH2 parallel library. In the first run, the test problem was solved with a sequential solver and the solution time was recorded. The total number of samplings was 80,000 for each grid point. Next, the same solver was parallelized with CAF, MPI, and OpenMP. Subsequently, we ran our parallel solvers with 2, 4, and 8 cores working in parallel. Solution times for all the runs were recorded as shown in Figure 4a.



(a) (b)

Figure 4. Parallel performance figures of the CAF and MPI on a workstation: (a) solution times, (b) speedup values.

Although the parallel solution times of the CAF and MPI techniques were very similar, we observed that CAF performed slightly better (7.5%). Speedup values were very close to ideal values for both techniques as shown in Figure 4b. This was expected, because the MCM is known as being very suitable for parallelization. CAF's speedup performance was found to be 3.6% higher on average than the ideal value. This superlinear speedup is caused by cache effects [20]. More data can be stored in caches as the number of processors increases. As a result, memory access time reduces.

In the case of OpenMP, however, the results were very discouraging. Even if we increased thread numbers, solution times increased, as shown in Table 2. We determined that the cause of this inefficiency was the "while" loop used in the MCM-based solver. Chandra did not recommend the use of "while" or "do-while" loops in OpenMP [21].

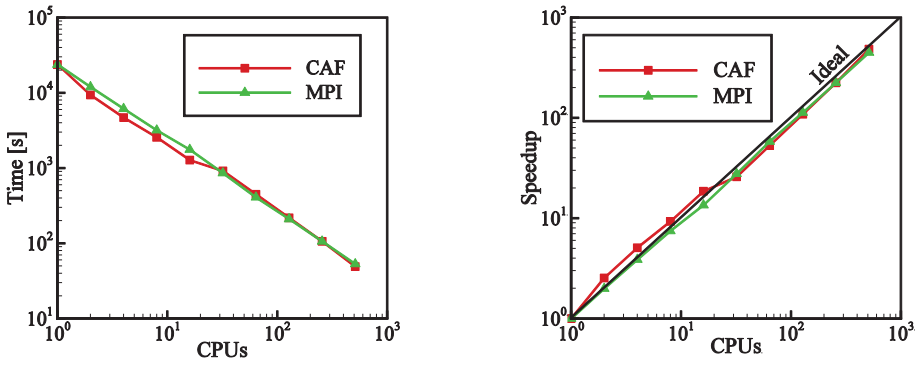
Table 2. Solution times for OpenMP.

Number of threads	2	4	8
Solution time (s)	2300	2894	5425

Subsequently we ran PMCS2D on a Cray supercomputer composed of AMD Opteron 6276 processors in order to extend our studies up to 512 CPUs. We increased the total sampling number to 2,560,000. Here we employed a Cray Fortran compiler with MPICH2 libraries. CAF language constructs are recognized by default with the Cray Fortran compiler. The solution times of CAF were slightly better than those of MPI if the number of CPUs was less than 32. MPI caught up with the CAF performance as the number of CPUs increased, as in Figure 5a. Speedup values were also very close to ideal values for both techniques, as shown in Figure 5b. This was expected because PMCS2D is a MCM-based solver.

4. Conclusions

Parallelization is an important tool to increase the efficiency of numerical solvers. Among many alternatives, we preferred the recently developed CAF to parallelize our PMCS2D solver. We demonstrated that parallelization using CAF was relatively straightforward. Moreover, parallel solvers developed with CAF are easy to read, understand, and maintain. After implementing CAF, we successfully validated the PMCS2D with the exact solution. The solution times of PMCS2D showed that parallel performance of CAF was slightly better than MPI (7.5%) and ideal (3.6%) if the number of CPUs was limited. However, the parallel performances of both techniques were found to be identical and very close to ideal when the number of CPUs increased drastically. We



(a) (b)

Figure 5. Parallel performance figures of the CAF and MPI on a Cray supercomputer: (a) solution times, (b) speedup values.

additionally concluded that OpenMP was not a suitable tool to parallelize PMCS2D because of the "while" loop restrictions. Within the scope of this work, we demonstrated that CAF was an alternative parallelization tool for program developers who are willing to devote less development time without sacrificing parallel performance.

References

[1] Budanur S, Mueller F, Gamblin T. Memory trace compression and replay for SPMD systems using extended PRSDs. Comput J 2012; 55: 206-217.

[2] Coarfa C, Dotsenko Y, Mellor-Crummey J. Experiences with Sweep3D implementations in Co-array Fortran. J Supercomput 2006; 36: 101-121.

[3] Wallcraft AJ. A comparision of Co-Array Fortran and OpenMP Fortran for SPMD programming. J Supercomput 2002; 22: 231-250.

[4] Barrett RF. Co-Array Fortran experiences with finite differencing methods. In: The 48th Cray User Group Meeting; 10 May 2006; Lugano, Switzerland.

[5] Sengil N, Tumuklu O, Celenligil MC. Implementation of a Monte Carlo method to a two-dimensional particle-in-cell solver using algebraic meshes. Nukleonika 2012; 57: 313-316.

[6] Wang T, Gu Y. Superconvergent biquadratic finite volume element method for two-dimensional Poisson's equations. J Comput Appl Math 2010; 234: 447-460.

[7] Kim YP, Kweon JR. The Fourier-finite element method for the Poisson problem on a non-convex polyhedral cylinder. J Comput Appl Math 2009; 233: 951-968.

[8] Swarztrauber PN, Sweet RA. Vector and parallel methods for the direct solution of Poisson's equation. J Comput Appl Math 1989; 27: 241-263.

[9] Hendrickx J, Barel MV, Kronecker A. Product variant of the FACR method for solving the generalized Poisson equation. J Comput Appl Math 2002; 140: 369-380.

[10] Liniger W, Odeh F, Hara VA. Second-order sparse factorization method for Poisson's equation with mixed boundary conditions. J Comput Appl Math 1992; 44: 201-218.

[11] Hoshino S, Ichida K. Solution of partial differential equations by a modified random walk. Numer Math 1971; 8: 61-72.

[12] Shentu J, Yun S, Cho NA. Monte Carlo method for solving heat conduction problems with complicated geometry, Nucl Eng Technol 2007; 39: 207-214.

[13] Thompson J, Chen P. Heat conduction with internal sources by modified Monte Carlo methods. Nucl Eng Des 1970; 12: 207-214.

[14] DeLaurentis JM, Romero LA. A Monte Carlo method for Poisson's equation. J Comput Phys 1990; 90: 123-140.

[15] Rosenthal JS. Parallel computing and Monte Carlo algorithms. Far East J Theor Stat 2000; 4: 207-236.

[16] Reid JK. Coarrays in the next Fortran Standard, Tech. Rep.: ISO/IEC JTC1/SC22/WG5 N1824. Geneva, Switzerland: ISO, 2010.

[17] Friedley A, Lumsdain A. Communication optimization beyond MPI. In: IEEE 2011 International Symposium on Parallel and Distributed Processing Workshops and PhD Forum; 16–20 May 2011; Shanghai, China. pp. 2018-2021.

[18] Numrich RW. Parallel numerical algorithms based on tensor notation and Coarray Fortran syntax. Parallel Comput 2005; 31: 588-607.

[19] Birdsall CK, Langdon AB. Plasma Physics Via Computer Simulation. New York, NY, USA: Taylor & Francis, 2005.

[20] Haveraaen M, Morris K, Rouson D, Radhakrishnan H, Carson C. High-performance design patterns for modern Fortran. Sci Prog 2015; 942059: 1-14.

[21] Chandra R. Parallel Programming in OpenMP. San Diego, CA, USA: Academic Press, 2001.