Proceedings of the 7th World Congress on Electrical Engineering and Computer Systems and Sciences (EECSS'21)
~~Prague, Czech Republic~~ Virtual Conference – July, 2021
Paper No. EEE 110
DOI: 10.11159/eee21.110

Modified Parareal Algorithm for Solving Time-Dependent Differential Equations

Sumathi Lakshmiranganatha[1], Suresh S. Muknahallipatna[1]
[1]University of Wyoming
1000 E. University Ave., Laramie, USA
slakshmi@uwyo.edu; sureshm@uwyo.edu

Abstract – Parallel algorithms are implemented to compute the solutions of partial differential equations and ordinary differential equations of complex dynamical systems to achieve near real-time solutions. One of the parallel algorithms widely implemented is the Parareal algorithm to solve time-dependent differential equations for various scientific applications. Parareal algorithm has shown promising speedups in achieving near real-time solutions using accelerators. However, it has been observed that the sequential predictor-corrector step of the Parareal algorithm impacts the computational performance. This paper analyses the Parareal algorithm and proposes modification to the predictor-corrector step of the Parareal algorithm to exploit data parallelism more and reduce the computation time. The modified algorithm is implemented to solve two systems of interdependent ODEs. The numerical accuracy and performance analysis of the modified algorithm is shown to be same as the original Parareal. The performance analysis of the modified algorithm on two accelerator computing architectures: Intel Xeon Phi CPU and Graphical processing units with OpenMP, OpenACC, and CUDA programming models are presented. The modified algorithm demonstrates performance improvement ranging from 1.2x-2x with respect to the original Parareal algorithm.

***Keywords*:** Parareal, Parallel-in-Time, Accelerators, Speedup, CUDA, OpenACC

1. Introduction

Time-domain simulations are performed to analyse the stability and response of complex dynamical systems to external disturbances that are mathematically represented by time-dependent partial differential equations (PDEs) or ordinary differential equations (ODEs). A system of time-dependent PDEs or ODEs is typically solved to obtain an approximate solution using a suitable numerical integration method that is an inherently sequential process. Hence, solving a large system of ODEs or PDEs is a computationally intensive problem and is often performed offline. With recent advancements in high-performance computing (HPC) resources and parallel algorithms based on domain decomposition techniques, it is possible to solve the compute-intensive problem much faster than the traditional methods with a near real-time solution.

One of the domain decomposition techniques widely used for solving time-dependent ODEs in parallel is temporal domain decomposition (TDD). TDD, also known as the parallel-in-time technique achieves parallelization across the time domain. This method solves a single time-dependent PDE or ODE by decomposing the entire simulation time into small intervals to solve in parallel using computing cores. In recent years, several parallel-in-time algorithms are developed that are suitable for modern-day HPC platforms namely: Parareal algorithm (PRA) [1], parallel in time algorithm (PITA) [2], parallel full approximation scheme in space and time (PFASST) [3], revision deferred corrections [4] and space-time multigrid methods [5].

This paper focuses on PRA. PRA is used for solving PDEs/ODEs of various applications in the scientific community like quantum chemistry [6], finance [7], neuroscience [8], hydrodynamic simulations [9], and many more. In the research work [10] [11], we have demonstrated the PRA's performance on two different computing architectures, the Intel Xeon Phi and GPU accelerators. The results demonstrated that a significant amount of time is spent in the sequential steps of the algorithm impacting the overall performance for compute-intensive problems.

To improve the computational efficiency by reducing the computational time of the PRA, multiple approaches were investigated. The computational performance of the PRA is dependent on how fast the coarse propagator can compute the solution sequentially and how fast the fine propagator computes the solution in parallel. In [12], the authors propose the use of the reduced basis method to provide a computationally inexpensive coarse propagator. In [13], a macro-micro PRA is proposed. A high dimension microscopic model is solved using a fine propagator whereas the coarse propagator is an integrator of a low dimension approximate macroscopic model. The authors in [14] demonstrate the use of different numerical integrator methods for coarse and fine propagators that potentially improve the computational performance. A simplified generator model was used for the coarse propagator in addition to different numerical integrator methods in [15] to reduce the sequential computational time. The authors in [16] demonstrated a learned coarse propagator of the PRA for a predictive model in a robot manipulation task involving multiple objects to reduce the computational time.

The methods explored and investigated to improve computational speedup of PRA are by using simpler physics models at the coarse propagator stage or learned coarse propagator using the deep learning approach. However, the predictor-corrector step of the PRA is still a sequential step. In this paper, we propose the modified PRA that combines the sequential predictor-corrector step of the PRA to reduce the computational burden caused by the sequential steps. With the proposed modification to the PRA, we can exploit the data parallelism and achieve better computational performance with the same numerical accuracy as the original PRA implementation. We demonstrate the modified PRA's performance on Intel Xeon Phi and GPU accelerators along with a multi-core Intel Xeon processor using OpenMP, OpenACC, and CUDA programming models.

The paper is organized as follows: Section 2 discusses the modification of the PRA for better performance improvement and data parallelism. In section 3, the implementation details, numerical and performance results are presented. Section 4 presents the discussion and conclusion.

2. Modified Parareal Algorithm

In PRA, the entire simulation time is decomposed into small subintervals with two time steps ΔT and δt as shown in figure 1. Each subinterval is then solved in parallel with some initial condition. A computationally inexpensive numerical integrator provides these initial conditions for the intervals with a less accurate solution with time step ΔT. The small sub-intervals are solved independently in parallel to obtain a more accurate solution of the differential equation with time step δt << ΔT. The algorithm has three major steps with two numerical operators namely coarse (GΔT) and fine (Fδt) propagators shown in equations 2 and 3, respectively, operated on the same initial conditions u^0 to compute the solution of a system time-dependent ODEs presented in equation 1.

$$\dot{u} = f(u,t), t \in [0,T] \tag{1}$$

$$\widetilde{U_n} = G_{\Delta T}(T_{n-1}, \widetilde{U_{n-1}}, \Delta T), \widetilde{U_0} = u^0 \tag{2}$$

$$\widehat{U_n} = F_{\delta t}(T_{n-1}, \widehat{U_{n-1}}, \delta t), \widehat{U_0} = u^0 \tag{3}$$

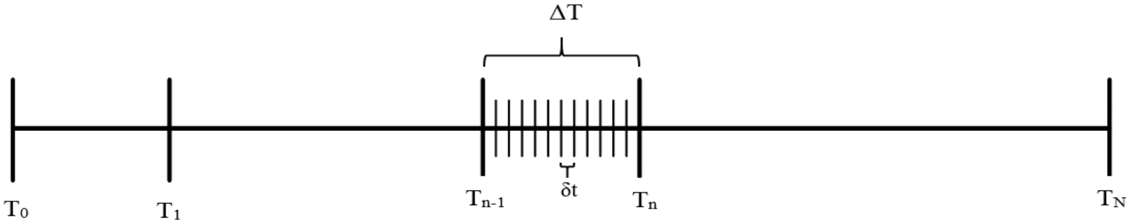


Fig 1. Time domain decomposition

In figure 2, the pseudocode for the original PRA is shown. The first and third steps are executed sequentially, while only the second step is executed in parallel. The first step is executed only once to get the initial coarse values. Steps two and three are iterated continuously to obtain the solution matching the sequential solution within the desired tolerance. At the end of the first iteration, the solution at T1 gets corrected to the fine solution. Similarly, at the kth iteration, the solution at Tk gets corrected to its respective fine solution. The algorithm is analysed to exploit additional parallelism since only step 2 is parallelizable. On further analysis in the predictor-correct step, the difference $\widehat{U_n^k} - \widetilde{U_n^{k-1}}$ can be parallelized since we already have the values of $\widetilde{U_n^{k-1}}$ from the previous iteration, but $\widetilde{U_n^k}$ needs to be calculated, which is a sequential process.

The results in [10] [11] showed that a significant amount of time is spent on the predictor step computing $\widetilde{U_n^k}$ values. This step impacted the overall performance of the original PRA and showed poor scaling when implemented on many-core and multi-core architectures for a system of inter-dependent ODEs. Therefore, we propose to modify the predictor-corrector step of the PRA algorithm to reduce the sequential computational overhead.

```
$U_0^0 \leftarrow \widetilde{U_0^0} \leftarrow y_0$ //initial condition for iteration 0
for n = 1 to N do                                                   Step 1: Coarse
    $\widetilde{U_n^0} \leftarrow G_{\Delta T}(\widetilde{U_{n-1}^0})$     propagation (Sequential)
    $U_n^0 \leftarrow \widetilde{U_n^0}$
end for
// PRA iterations
for k=1 to k_max do
    $U_0^k \leftarrow y_0$
    for n=1 to N do                                                 Step 2: Fine
        $\widehat{U_n^k} = F_{\delta t}(U_{n-1}^{k-1})$              propagation (parallel)
    end for

    for n=1 to N do                                                 Step 3:
        $\widetilde{U_n^k} \leftarrow G_{\Delta T}(U_{n-1}^k)$       Predictor-
        $U_n^k \leftarrow \widehat{U_n^k} + \widetilde{U_n^k} - \widetilde{U_n^{k-1}}$    Corrector
    end for                                                         (sequential)
    if $|U_n^k - U_n^{k-1}|$ ∈ ∀ n then
        BREAK // loop is terminated if converged
    end if
end for
```

Fig 2. Pseudocode of the original PRA [17]

```
$U_0^0 \leftarrow \widetilde{U_0^0} \leftarrow y_0$     //initial condition

for n = 1 to N do                                                   Step 1: Coarse
    $\widetilde{U_n} \leftarrow G_{\Delta T}(\widetilde{U_{n-1}^0})$       propagation (Sequential)
    $U_n^0 \leftarrow \widetilde{U_n^0}$
end for

// PRA iterations
for k=1 to k_max do
    for n=k to N do                                                 Step 2: Fine
        $\widehat{U_n^k} = F_{\delta t}(U_{n-1}^{k-1})$              propagation
        $U_n^k \leftarrow \widetilde{U_n^0} + \widehat{U_n^k} - \widetilde{U_n^0}$   with corrector
    end for                                                         (parallel)

    if $|U_n^k - U_n^{k-1}|$ ∈ ∀ n then
        BREAK // loop is terminated if converged
    end if
end for
```

Fig 3. Pseudocode of the modified PRA

In figure 2, for each iteration, U_0^k is initialized to the initial condition y_0. In the predictor step, the same initial condition is used to generate the coarse values $\widetilde{U_n^k}$ sequentially for $1 \leq n < N$. Since the initial condition is the same, the predictor value $\widetilde{U_n^k}$ in each iteration remains the same. The predictor value is the same as the initial coarse propagator solution obtained in step 1. Therefore, we can eliminate this step in the predictor-corrector method and combine the second for loop in the PRA iterations with the first for loop, as shown in figure 3. From the pseudocode in figure 3, it is now possible to parallelize the PRA iterations since there is no data dependency. Also, the original PRA algorithm is now reduced to only two steps.

- Step 1 is the initial coarse propagator solution, which computes all the U_n^k coarse values for $1 \leq n < N$ sequentially.
- Step 2 consists of the fine propagators and the corrector, which can be performed in parallel. In the predictor-corrector step, the predictor is eliminated, and only the corrector is retained. The fine values are corrected at the end of each iteration.

The numerical accuracy is the same as the original PRA algorithm. At the end of the first iteration, the solution at T1 gets updated to the fine solution, at the end of the kth iteration, the coarse value at Tk gets updated to the fine solution. Another important modification done is the number of iterations performed in the inner for loop of the original PRA iterations. In pseudocode 1, the inner for loops in the original PRA iterations performs computation even for already computed values. In the modified PRA pseudocode, computation is done only for the values for which the fine solution needs to be done. At the end of the first iteration, the solution at T1 gets corrected to the fine solution. In the next iteration, i.e., the second iteration, the computation begins from T1 as the initial condition instead of T0 since we have already obtained the corrected solution for time T1. Similarly, for iteration 3, the computation starts with the initial condition at T2 instead of T0. This way, we are computing the solutions required for the correction and do not re-compute the corrected solutions.

3. Implementation and Results

The modified PRA is implemented for two systems of ODEs mathematically modeling the dynamic behaviour of a synchronous generator of a power system.

- **Case 1**: In this case, we consider the classical model of a synchronous generator that is mathematically modeled in equations 4 and 5. The two time-dependent ODEs represent the rotor angle δ and angular velocity ω respectively.

$$\frac{d\delta}{dt} = \omega - \omega_s = \Delta\omega \tag{4}$$

$$\frac{d\Delta\omega}{dt} = \frac{\pi f_0 P_a}{H} \tag{5}$$

Where, H is the inertia constant (MJ/MVA), Pa is the accelerating power, f_o is the nominal frequency, $\omega_s = 2\pi f_o$ is the rated angular speed, δ is the rotor angle. $\omega = \frac{d\delta}{dt}$ is the relative speed or angular velocity with respect to the synchronously revolving magnetic field (reference frame)

- **Case 2:** In this case, we consider the fourth-order mathematical model of a synchronous generator shown in equations 6 to 9. This model consists of four time-dependent ODEs addressing the direct and quadrature axis parameters of the synchronous generator.

$$\frac{d\delta}{dt} = \omega - \omega_s = \Delta\omega \tag{6}$$

$$T'_{d0}\frac{dE'_q}{dt} = -E'_q - (X_d - X'_d)i_d + E_{fd} \tag{7}$$

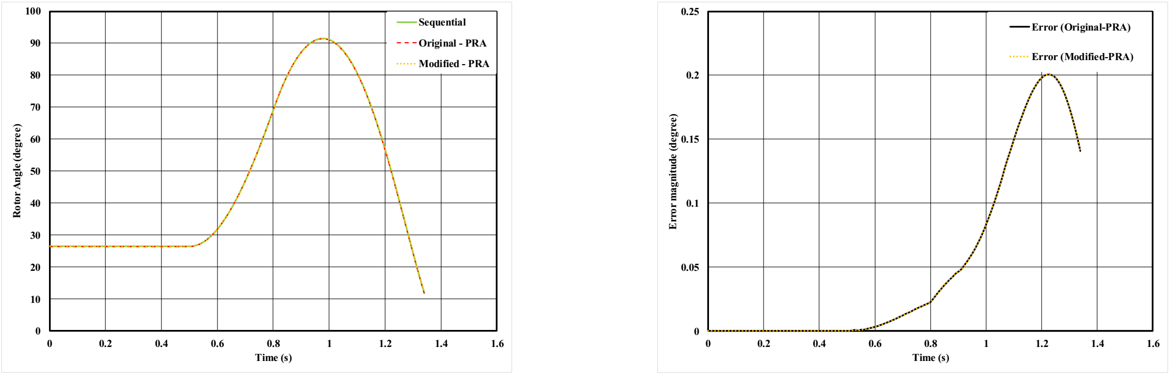
$$T'_{q0}\frac{dE'_d}{dt} = -E'_d + \left(X_q - X'_q\right)i_q \tag{8}$$

$$\frac{H}{\pi f_o}\frac{d\omega}{dt} = T_m - T_e - D(\omega - \omega_s) \tag{9}$$

Where, $E'_d$and E'_q are the transient voltages along direct (d) and quadrature (q) axis respectively of the generator, $i_d$and $i_q$are the stator currents of the d and q axis respectively, D is the damping constant, $X_d$and $X_q$are the d and q axis synchronous reactances respectively, $X'_d$and $X'_q$are the d and q transient reactances respectively, $T'_{d0}$and $T'_{q0}$are the open-circuit transient time constants for d and q axes, $T_m$and $T_e$are the mechanical and electrical torques, respectively.

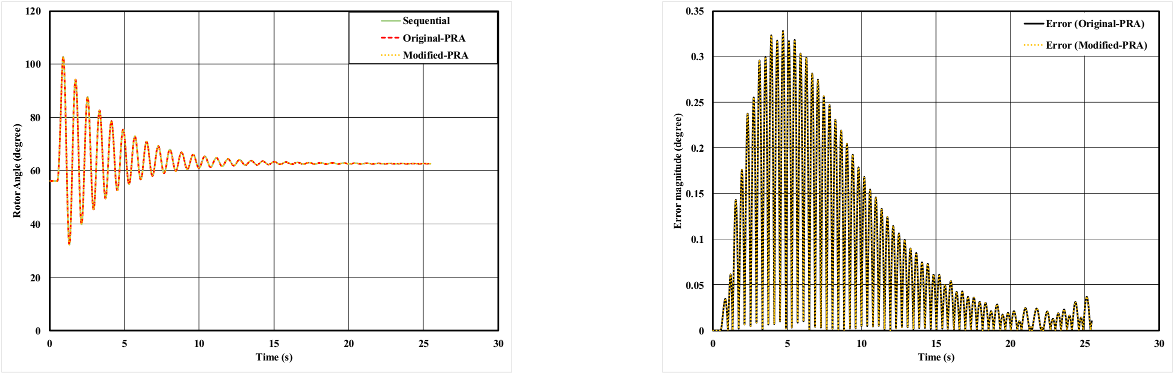
3.1 Numerical Results

The time domain simulations of the test systems discussed in [10] is performed using the modified PRA implementation. In figures 4a and 5a, the variation of the rotor angle for cases 1 and 2 are presented. The numerical solution of the rotor angle obtained using the modified PRA is compared with the numerical solution obtained using the traditional sequential and the original PRA methods in figures 4a and 5a. It can be observed in figures 4a and 5a, the rotor angle variation of the modified PRA is very close to the Original PRA and sequential methods. In figures 4b and 5b, the difference in the rotor angle computed using the modified and the original PRA algorithms in reference to the sequential computation are shown. It can be seen that the rotor angle error of the modified and the original PRA is same demonstrating the numerical accuracy of the modified PRA.



a) Rotor Angle variation comparison b) Rotor Angle variation error

Fig 4. Time domain Simulations for Case 1



a) Rotor Angle variation comparison b) Rotor Angle variation error

Fig 5. Time domain Simulations for Case 2

3.2 Performance Results

The performance of the modified PRA is analyzed by collecting the parallel runtime to compute the numerical solutions for cases 1 and 2. The modified PRA is implemented using OpenMP, OpenACC, and CUDA programming models across homogeneous and heterogeneous computing platforms. The homogeneous computing architecture comprises of Intel Xeon processor codenamed Haswell (HSW) and Xeon phi accelerator Knights Landing (KNL) while the heterogeneous computing architecture consists of Nvidia GPUs [11]. . The optimization techniques presented in [11] are used in the implementation of the modified PRA. The homogenous architecture parallel runtime of the modified PRA is given in equation 10 addressing the modifications to the parallel runtime of the original PRA [11].

$$T_{PRAm}^{CPU} = t^{c} + \sum_{i=1}^{N} (t^{fcr}) \tag{10}$$

In equation 10, t^{c} is the computation time of the coarse propagator, t^{fcr} is the computation time for the fine propagator with the corrector, N is the number of PRA iterations.

The parallel runtime for the modified PRA implementation using GPUs is given in equation 11.

$$T_{PRAm}^{GPU} = t_{H}^{c} + \sum_{i=1}^{N} \left(t_{H}^{G} + t_{G}^{fcr} + t_{G}^{H}\right) \tag{11}$$

Where, t_H^c is the computation time of the coarse propagator on the host, t_H^G is the memory transfer latency between the host and the GPU, t_G^{fcr} is the computation time of the fine propagators with corrector on the GPU, t_G^H is the memory transfer latency between the GPU and the host, N is the number of iterations. The performance of the modified-PRA is compared with the original PRA using the achieved speedup with varying number of fine propagators.

In table 1, the sequential and parallel runtime of sequential, original and modified PRA for case 1 with varying N^f on different computing architectures and programming models are presented. The smallest execution time is achieved for OpenMP implementation of modified PRA on KNL. For all hardware architectures, and programming models, the parallel runtime of the modified PRA is reduced in comparison to the original PRA. The performance of the modified PRA on the GPU with CUDA and OpenACC is lower in comparison to KNL due to lower occupancy (computations performed by each thread is small) of the GPU.

Table 1: Sequential and parallel runtime for Case 1.

N^f	Sequential (ms)	Original PRA (ms)				Modified PRA (ms)			
		OpenMP-HSW	OpenMP-KNL	OpenACC-GPU	CUDA-GPU	OpenMP-HSW	OpenMP-KNL	OpenACC-GPU	CUDA-GPU
128	1.78	0.19	0.09	0.27	0.18	0.122	0.059	0.22	0.177
256	3.59	0.25	0.11	0.358	0.24	0.186	0.073	0.23	0.2
512	7.1	0.45	0.196	0.38	0.28	0.346	0.123	0.256	0.224

In figure 6, the speedup comparison for modified PRA and original PRA implementations are presented. The modified-PRA implemented on KNL using OpenMP achieves the best performance with a speedup of 57x and 31x on the GPU using CUDA. The lower speedup with GPU is mainly due to the underutilization of the GPU resources due to a fewer number of equations of case-1 being solved. The modified and original PRA both executing on the GPU exhibit a strong scaling efficiency. In general, the modified PRA speedup improvement with respect to the original PRA ranges from 1.2x to 1.7x.

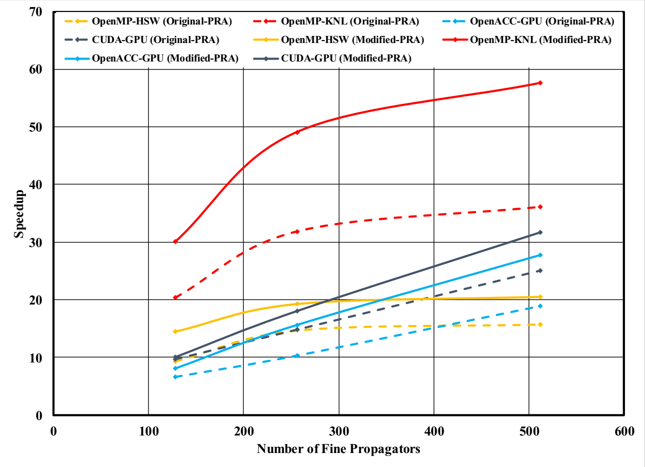


Fig 6. Speedup w.r.t sequential runtime for Case 1

In table 2, the sequential and parallel runtime for original and modified PRA for case-2 with varying N^f on different computing architectures and programming models are presented. The execution time of the modified PRA on the GPU with CUDA and OpenACC is significantly less compared to that on HSW/KNL with OpenMP for larger values of N^f increases, the execution time difference between the CPU and GPU implementations increases significantly. This is due to the massive number of cores available on GPUs in comparison to a CPU and KNL accelerator. The other significant difference that can be observed is the total execution time for KNL. For the original PRA, the execution time for KNL was more than HSW for higher N^f due to the sequential predictor-corrector step. Since the KNL processor clock speed is lower compared to the HSW, the time taken to compute the coarse values sequentially was significantly high compared to HSW. This impacted the overall execution time, and KNL performed poorly. However, in the modified PRA, with the elimination of the sequential predictor-corrector method, the performance of KNL has improved significantly, and the execution time is lower than the HSW. Also, the performance of OpenACC and CUDA has improved significantly compared to original PRA implementations.

Table 2: Sequential and parallel runtime for Case 2.

N^f	Sequential (ms)	Original PRA (ms)				Modified PRA (ms)			
		OpenMP-HSW	OpenMP-KNL	OpenACC-GPU	CUDA-GPU	OpenMP-HSW	OpenMP-KNL	OpenACC-GPU	CUDA-GPU
2560	40.786	2.57	2.21	2.23	1.88	2.12	1.15	1.41	1.23
5120	72.57	4.55	4.425	3.04	2.73	3.72	2.192	2.04	1.80
12800	159.67	8.62	11.05	6.14	5.043	6.88	5.36	4.13	3.87
25600	289.96	15.97	22.02	13.39	9.714	12.24	10.67	7.29	6.80

In figure 7, the speedup achieved with the modified and the original PRA is shown. It can be seen there is an improvement in the speedup across the different platforms. The range of improvement is 1.3x to 2x. KNL and OpenACC implementation have maximum improvement with the CUDA implementation exhibiting overall best performance. This shows that the GPUs perform better when the problem is compute-intensive and has more work for each thread. Also, the performance of OpenACC is very close to CUDA. However, the scaling efficiency of the modified PRA is still weak, and similar to the original PRA.

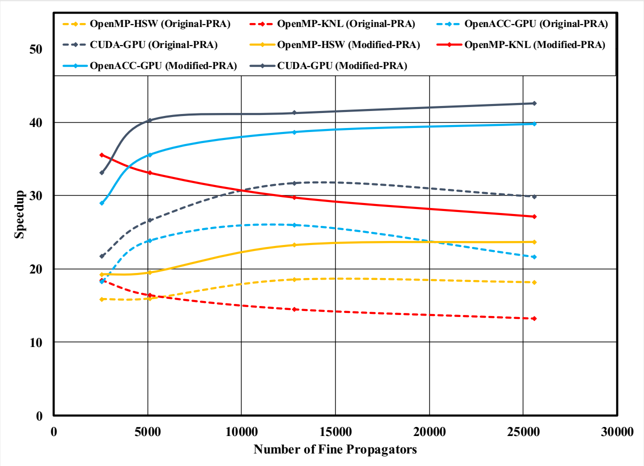


Fig 7. Speedup w.r.t sequential runtime for Case 2

4 Discussion and Conclusion

In this paper, we have proposed the modified PRA to reduce the computation burden caused by the sequential steps of the PRA. In the proposed algorithm, we eliminate the sequential predictor-corrector step of the PRA by combining the corrector with the fine propagator step. The modified PRA showed an improvement between 1.2x -2x for the system of ODEs demonstrated in this paper. However, not much improvement in the scaling efficiency was observed and a weak scaling efficiency is noticed for the four time-dependent ODEs. Traditionally, the approach to improve weak scaling after algorithm optimization is to use additional hardware and software resources. The additional software resource can be incorporated using task parallelism. Task parallelism involves parallelizing the computations of each system of ODEs by distributing the computations spatially. In the case of CPUs, the task parallelism would involve distributing the PRA computations across multiple nodes using the message passing interface (MPI) programming model. In the case of GPUs, the task parallelism can be achieved by implementing multiple streams on a single GPU initially, and later distributing the computations across multiple GPUs and nodes using the MPI programming model. If the ODEs are not interdependent, i.e., do not need an exchange of information during the time evolution, task parallelism will improve the scaling. The ODEs of the complex systems usually being interdependent will require an exchange of information that will introduce significant communication latencies between GPUs in a node and between nodes, further weakening the scaling. Therefore, the maximum possible reduction in running time while computing the solutions of interdependent ODEs using PRA on current HPC hardware and programming models has been achieved.

Acknowledgements

This work was supported in part by the National Science Foundation under the award ECCS-1828066.

References

[1] J. L. Lions, Y. Maday, and G. Turinici, “Résolution d’EDP par un schéma en temps Ŧpararéel ŧ,” *Comptes Rendus de l’Académie des Sciences-Series I-Mathematics*, vol. 332, no. 7, pp. 661–668, 2001.

[2] C. Farhat and M. Chandesris, “Time-decomposed parallel time-integrators: theory and feasibility studies for fluid, structure, and fluid-structure applications,” *Int. J. Numer. Methods Eng.*, vol. 58, no. 9, pp. 1397–1434, 2003.

[3] M. Minion, “A hybrid parareal spectral deferred corrections method,” *Comm. App. Math. Comp. Sci.*, vol. 5, no. 2, pp. 265–301, 2010.

[4] A. J. Christlieb, C. B. Macdonald, and B. W. Ong, “Parallel High-Order Integrators,” *SIAM J. Sci. Comput.*, vol. 32, no. 2, pp. 818–835, 2010.

[5] S. Friedhoff, R. D. Falgout, T. V. Kolev, S. MacLachlan, and J. B. Schroder, “A multigrid-in-time algorithm for solving evolution equations in parallel,” no. LLNL-CONF-606952). Lawrence Livermore, 2012.

[6] Y. Maday and G. Turinici, “Parallel in time algorithms for quantum control: Parareal time discretization scheme,” *Int. J. Quantum Chem.*, vol. 93, no. 3, pp. 223–228, 2003.

[7] G. Bal and Y. Maday, “A ‘parareal’ time discretization for nonlinear PDE’s with application to the pricing of an American put,” Berlin, Heidelberg: Springer, 2002, pp. 189–202.

[8] M. Bedez et al., “A fully parallel in time and space algorithm for simulating the electrical activity of a neural tissue,” *J. Neurosci. Methods*, vol. 257, pp. 17–25, 2016.

[9] A. Eghbal, A. G. Gerber, and E. Aubanel, “Acceleration of unsteady hydrodynamic simulations using the parareal algorithm,” *J. Comput. Sci.*, vol. 19, pp. 57–76, 2017.

[10] S. Lakshmiranganatha and S. S. Muknahallipatna, “Graphical processing unit based time-parallel numerical method for ordinary differential equations,” *J. Comput. Commun*., vol. 08, no. 02, pp. 39–63, 2020.

[11] S. Lakshmiranganatha and S. S. Muknahallipatna, “Performance analysis of accelerator architectures and programming models for parareal algorithm solutions of ordinary differential equations,” *J. Comput. Commun.,* vol. 09, no. 02, pp. 29–56, 2021.

[12] L. He, “The reduced basis technique as a coarse solver for parareal in time simulations,” *J. Comput. Math*., vol. 28, no. 5, pp. 676–692, 2010.

[13] F. Legoll, T. Lelièvre, and G. Samaey, “A micro-macro parareal algorithm: Application to singularly perturbed ordinary differential equations,” *SIAM J. Sci. Comput.*, vol. 35, no. 4, pp. A1951–A1986, 2013.

[14] G. Gurrala, A. D. Dimitrovski, S. Pannala, S. Simunovic, and M. R. Starke, “Parareal in Time for Dynamic Simulations of Power Systems,” *Oak Ridge National Lab*, 2015.

[15] N. Duan, A. Dimitrovski, S. Simunovic, and K. Sun, “Applying reduced generator models in the coarse solver of parareal in time parallel power system simulation,” in *2016 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, 2016.

[16] W. Agboh, O. Grainger, D. Ruprecht, and M. Dogar, “Parareal with a learned coarse model for robotic manipulation,” *Comput. Vis. Sci.*, vol. 23, no. 1–4, 2020.

[17] A. S. Nielsen, “Feasibility study of the parareal algorithm,” Technical University of Denmark, 2012.