Journal of Computer and Communications, 2021, 9, 29-56
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Performance Analysis of Accelerator Architectures and Programming Models for Parareal Algorithm Solutions of Ordinary Differential Equations

Sumathi Lakshmiranganatha, Suresh S. Muknahallipatna*

Department of Electrical and Computer Engineering, University of Wyoming, Laramie, USA
Email: slakshmi@uwyo.edu, *sureshm@uwyo.edu

How to cite this paper: Lakshmiranganatha, S. and Muknahallipatna, S.S. (2021) Performance Analysis of Accelerator Architectures and Programming Models for Parareal Algorithm Solutions of Ordinary Differential Equations. *Journal of Computer and Communications*, **9**, 29-56.
https://doi.org/10.4236/jcc.2021.92003

Received: January 2, 2021
Accepted: February 20, 2021
Published: February 23, 2021

Abstract

Increasing needs for the study of complex dynamical systems require computing solutions of a large number of ordinary and partial differential time-dependent equations in near real-time. Numerical integration algorithms, which are computationally expensive and inherently sequential, are typically used to compute solutions of ordinary and partial differential time- dependent equations. This presents challenges to study complex dynamical systems in near real-time. This paper examines the challenges of computing solutions of ordinary differential time-dependent equations using the Parareal algorithm belonging to the class of parallel-in-time algorithms on various high-performance computing accelerator-based architectures and associated programming models. The paper presents the code refactoring steps and performance analysis of the Parareal algorithm on two accelerator computing architectures: the Intel Xeon Phi CPU and Graphics Processing Unit many-core architectures, and with OpenMP, OpenACC, and CUDA programming models. The speedup and scaling performance analysis are used to demonstrate the suitability of the Parareal to compute the solutions of a single ordinary differential time-dependent equation and a family of interdependent ordinary differential time-dependent. The speedup, weak and strong scaling results demonstrate the suitability of Graphical Processing Units with the CUDA programming model as the most efficient accelerator for computing solutions of ordinary differential time-dependent equations using parallel-in-time algorithms. Considering the time and effort required to refactor the code for execution on the accelerator architectures, the Graphical Processing Units with the OpenACC programming model is the most efficient accelerator for computing solutions of ordinary differential time-dependent equations using parallel-in-time algorithms.

Keywords

Accelerators, Many-Core, Directive-Based, Time-Parallel, Scaling, Speedup

1. Introduction

The study of complex systems to analyze their stability and time evolution in near real-time due to external forces or disturbances is an emerging field of research. A natural or engineered system is defined as a complex system if it exhibits the following characteristics:

- Consists of a large number of interacting subsystems or components or agents.
- Exhibit emergence; that is, a self-organizing collective behavior difficult to determine from the knowledge of components behavior.
- Lack of a central controller controlling the emergent behavior.

Out of the three above characteristics, emergent behavior is considered the most distinguishing feature of complex systems. The nonlinear dynamics of the complex system [1] is the main contributor to the emergent behavior of a complex system, hence the term complex dynamical systems. In the study of emergent behavior, the system's time evolution requires a system model, which is the mathematical representation of the system [2]. If the system dynamics are nonlinear, the mathematical representation involves nonlinear algebraic or nonlinear differential equations or both.

The mathematical models of complex dynamical systems consisting of ordinary differential equations (ODEs), partial differential equations (PDEs), and time independence equations can be found in both natural and engineered systems. Examples of complex natural system models are the protein folding model, weather forecasting models, and crowd simulation models, to mention a few. The models of modern power grid and internet are of two massive, complex engineered systems. These models' common characteristics consist of equations modeling both the complex system's steady-state and dynamic behavior. Depending on the application domain of a complex system, the algebraic or grid equations modeling the steady-state of the complex system are known as the network equations or dynamic-core (dycore), and the ODEs/PDEs modeling the dynamic state of the complex system is known as the system dynamics or physics. For example, the sensor network equations used in weather forecasting constitute the dycore, while the physics consists of the PDEs. In the modern power grid, the steady-state load flow equations constitute the network equations, and the ODEs modeling the grid components are the system dynamic equations.

Therefore, the study of complex dynamical systems to analyze their stability and time evolution involves computing the solutions of a large number of nonlinear algebraic and differential equations, which is computationally expensive.

To date, the research to address the computation burden in the study of com-

plex dynamic systems is focused on using high-performance computation (HPC) techniques with traditional supercomputers to parallelize the computation of network equations or dycore solutions resulting in execution times in terms of days. In the last few years, with the emergence of supercomputers based on many-core architectures due to hardware accelerators like graphical processing units (GPUs) and Intel Xeon Phi, the execution times have been reduced from days to hours or a single day. Recently, the computation time of weather forecasting using the model for prediction across scales (MPAS) was reduced to less than an hour from 24 hours using GPU high-performance computing techniques [3]. The HPC technique used to parallelize the computation of the network equations solutions constitutes distributing the computations across the cores in each socket (CPU) in a supercomputer node. The distribution among the cores is implemented using spatial domain decomposition [4] [5] [6]. However, the underneath system dynamic equations or the physics solutions are computed sequentially using traditional numerical integration techniques resulting in only offline studies of complex systems to evaluate stability and time-evolution.

In recent years, due to the availability of powerful hardware accelerators like GPUs and Xeon Phi, HPC techniques to parallelize numerical integration methods to compute solutions of ODE/PDE using time-domain decomposition [7] [8] [9] approaches in being researched. The time-domain decomposition approaches to parallelize the numerical integration methods are researched to reduce the computational time from hours to a few minutes or seconds, depending on the complexity. This paper investigates the time-domain decomposition approach to compute the solutions of ODEs.

The idea of developing parallel methods for solving the ODEs dates back to the 1960s in [10] [11], where the authors present a parallel numerical integration method to solve the ODEs. In the 1990s, time-parallel multigrid methods were developed and presented in [12] [13] for the Navier-Stokes equation. Other studies were performed on parallel multiple shooting methods [14] and parabolic multigrid-methods [15]. In recent years, several parallel-in-time algorithms are developed namely: The Parareal algorithm (PRA) [16], parallel in time algorithm (PITA) [17], parallel full approximation scheme in space and time (PFASST) [18] [19], revision deferred corrections [20] [21] and space-time multigrid methods [22] [23].

PRA is widely studied and implemented for computing solutions of ODEs and PDEs in several application domains like finance [24], molecular dynamics [8], quantum chemistry [25], non-linear parabolic equations [26], plasma physics [27], and power systems [28] [29] [30]. Several studies are performed to analyze the stability and convergence properties [9] [31] [32] [33] of the algorithm. The scalability of the algorithm is studied in [34] [35] using different computing cores.

The PRA is implemented on heterogeneous and homogeneous computing architectures for several applications. The research in [36] demonstrates the im-

plementation of PRA for unsteady hydrodynamic simulations. The authors focus on analyzing the stability of the PRA for solving the advanced turbulence models to solve evolution fluid problems at high Reynolds number. The instability problem of PRA is addressed by incorporating the windowing technique at a high Reynolds number. The proposed framework is illustrated with a fully turbulent vortex shedding from a cylinder and a flow from the Grand Passage tidal zone in the Bay of Fundy. OpenMP is used to achieve temporal parallelism, and MPI is used to achieve data parallelism introduced by spatial decomposition. The computationally intensive tasks of the application are accelerated using CUDA.

In [37], the authors have implemented PRA to parallelize the time dimension in solving the PDEs modeling neural tissue's electrical activity. The models are mathematically represented as PDEs. Parallelization space in GPU using dynamic grids, *i.e.*, launching multiple streams technique, is used to solve the mono-domain model. CUDA programming model is used to implement on GPUs. The GPU results are compared with the simulations obtained from a multicore processor cluster using the MPI programming model. A speedup of 100 is achieved in computation time between the sequential and parallel execution on the GPU.

The research in [38] demonstrates PRA's implementation, coupled with the Exact Domain Decomposition method (EDD), to solve the Hodgkin-Huxley equation. PRA is implemented to achieve the outer level parallelism, and the EDD algorithm with fine decomposition is used for inner-level parallelism. The method uses dynamic parallelism of CUDA to achieve multi-level parallelism on GPUs. The maximum speedup achieved is 2.5x for the largest matrix size.

A stencil-based implementation of PRA is presented in [39] called STELLA. STELLA provides OpenMP and CUDA backend for shared memory parallelization on CPUs and GPUs for intranode spatial stencils. The node-wise spatial parallelism is combined with PRA, and the MPI programming model is used to parallelize in time across nodes. The performance is analyzed for an advection-diffusion problem with a time-dependent diffusion coefficient. In this framework, the spatial dimension of the PDEs is solved on GPUs for fine-grained parallelism.

In the above literature, the PRA is implemented to solve PDEs, which involves space and time. The majority of the above implementations incorporate both spatial and temporal parallelism. For the heterogeneous computing architectures implementation, only the CUDA programming model is used. In [28], PRA is implemented to solve the power systems dynamics that involve only temporal parallelization. However, the research focuses on the stability and suitability of PRA to solve a system of ODEs. The PRA is implemented on MATLAB, and ODEs are solved sequentially on the CPU. The potential speedups that can be achieved when implemented on multicore processors are presented.

In [40], the use of the time-parallel approach and, in particular, the Parareal algorithm (PRA) implementation on the Graphical Processor Unit (GPU) was

investigated. The Compute Unified Device Architecture (CUDA) programming model was used for solving ODEs representing the electrical components of the power system. The investigation focused on developing a reliable implementation of PRA on heterogeneous architecture to solve ODEs in temporal decomposition to reduce computational time and be applied to achieve real-time or faster than real-time TSA using a large number of GPUs.

In this paper, PRA is implemented using different programming models on homogeneous and heterogeneous computing architectures. OpenMP [41] programming model is used for the PRA implementation for the homogeneous computing architectures. CUDA [42] and OpenACC [43] programming models are used for the PRA implementation on heterogeneous computing architectures. PRA's performance on both computing hardware architectures and three programming models is analyzed by comparing using the speedup and scaling achieved. We study the performance of PRA for the system of interdependent ODEs. First, the PRA is implemented to solve the system of ODEs on two homogeneous computing architectures: Intel Xeon and Intel Xeon Phi processors, and secondly, on the heterogeneous computing architecture: NVidia GPUs. The programming models for the heterogeneous computing architecture are implemented using the OpenACC, a directive-based programming model, and the CUDA programming model. In addition to the PRA implementations' speedup performance analysis, the PRA's scaling is also analyzed. The scaling analysis is performed by increasing the number of fine grids of the PRA, increasing the number of dependent ODEs of a system, and increasing the hardware resources.

This paper is organized as follows: In Section 2, a detailed explanation of the Parareal algorithm is presented. A brief overview of multicore Intel Xeon and Xeon Phi computing hardware architectures are discussed in Section 3. Section 4 discusses implementing the Parareal algorithm on multicore CPUs and NVIDIA GPUs using the three programming models. In Section 5, the performance analysis of PRA implementation is presented. Section 6 presents the conclusion and future work.

2. Parareal Algorithm

The Parareal Algorithm (PRA) is one of the temporal domain decomposition algorithms developed in 2001 [16]. The PRA involves decomposing the entire simulation time into small subintervals and solving each subinterval in parallel with different initial conditions generated by the coarse grid. A computationally inexpensive numerical integrator provides these initial conditions for the intervals with a less accurate solution. The small sub-intervals are solved independently in parallel to obtain a more accurate solution of the differential equation. The system of nonlinear ODEs to be solved is defined as

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

Let the entire simulation time t be decomposed into N sub-intervals as $T_0 < T_1 < \cdots < T_N$ with the step size of $\Delta T = T_n - T_{n-1} \quad \forall 1 \le n < N$ as shown in

Figure 1. The solution of the ODE in Equation (1) is solved using PRA in three major steps with two numerical operators. The two numerical operators defined in PRA are 1) Fine Propagator, which is denoted as $F_{\delta t}$, and 2) Coarse Propagator, which is denoted as $G_{\Delta T}$. The two numerical operators using initial condition $u(T_{n-1}) = U_{n-1}$ compute the approximate solution of Equation (1) at time T_n but with different time steps.

The $F_{\delta t}$ computes the approximate solution of Equation (1) with a small time-step $\delta t \ll \Delta T$ at time T_n, as shown in Figure 1. The solution obtained using $F_{\delta t}$ is computationally expensive but provides a more accurate solution. The solution computed from the fine propagator is denoted as $\widehat{U_n}$. The fine propagator can be mathematically defined as

$$\widehat{U_n} = F_{\delta t}\left(T_{n-1}, \widehat{U_{n-1}}, \delta t\right), \widehat{U_0} = u^0 \tag{2}$$

The $G_{\Delta T}$ computes the approximate solution of Equation (1) at the same time instance T_n but with a time step ΔT. The solution obtained using a coarse propagator is less accurate and is computationally inexpensive. The approximate solution computed by the coarse propagator is denoted as $\widetilde{U_n}$. The coarse propagator is mathematically represented in Equation (3).

$$\widetilde{U_n} = G_{\Delta T}\left(T_{n-1}, \widetilde{U_{n-1}}, \Delta T\right), \widetilde{U_0} = u^0 \tag{3}$$

The two numerical operators defined above are used in three major steps of the PRA algorithm, as shown in Figure 2.

Figure 2 shows three major steps, with steps 2 and 3 that are iterated until convergence. The following steps describe the PRA implementation:

Step 1: Initial coarse propagation

This step is used for the initialization of the algorithm. It generates fast but less accurate initial conditions sequentially using the $G_{\Delta T}$ used as a starting point. The approximate solution obtained at T_n is used as the initial conditions for step 2. The first ***for loop*** in the pseudocode is the initial coarse propagation step. The superscript "0" indicates it is the initial iteration.

Step 2: Fine propagation

In this step, the $F_{\delta t}$, which is computationally expensive, is used to propagate the fine solution in parallel over each sub-interval $t \in [T_{n-1}, T_n]$. This step provides a more accurate solution at time T_n using smaller time step δt. The first inner ***for loop*** in the pseudocode indicates the fine propagation step. The outer ***for loop*** in the pseudocode shows the PRA iterations where k is the iteration number.

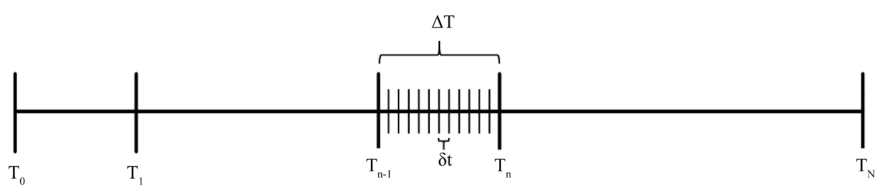

Figure 1. Decomposition of time into smaller sub-intervals.

```
U_0^0 ← Ũ_0^0 ← y_0          //initial condition for iteration 0
for n = 1 to N do
        Ũ_n^0 ← G_ΔT(Ũ_{n-1}^0)
        U_n^0 ← Ũ_n^0
end for                                   Step 1: Initial coarse propagation (sequential)
// PRA iterations
for k=1 to k_max do
        U_0^k ← y_0
        for n=1 to N do
        Û_n^k = F_δt(U_{n-1}^{k-1})
        end for                           Step 2: Fine propagation (parallel)
        for n=1 to N do
        Ũ_n^k ← G_ΔT(U_{n-1}^k)
        U_n^k ← Û_n^k + Ũ_n^k − Ũ_n^{k-1}
        end for                           Step 3: Predictor-Corrector (sequential)
        if |U_n^k − U_n^{k-1}| ∈ ∀ n then
                BREAK // loop is terminated if converged
        end if
end for
```

Figure 2. Pseudocode of PRA algorithm [33].

Step 3: Predictor-Corrector

In this step, the coarse values are corrected using the fine solutions obtained from the previous step. The Predictor-Corrector method is used to correct the solution difference obtained from coarse and fine propagators for the next iteration. The second inner ***for loop*** in the pseudocode is the predictor-corrector method. This step of the algorithm is a sequential process. The coarse values updated using the predictor-corrector method is used as the initial conditions in step 2 for the next iteration.

The notation U in **Figure 2** represents the corrected coarse solution obtained from the predictor-corrector step. The fine propagator corrects the initial conditions which are given by the coarse propagators in every iteration. At the end of the 1st iteration, the coarse value at time T_1 gets corrected to the fine solution. Similarly, at the end of the kth iteration, the coarse value at time T_k gets corrected to its respective fine solution. Steps 2 and 3 of the algorithm are iterated until the difference between the two successive coarse values meets the desired tolerance level. Faster convergence can be obtained by choosing the time step for the coarse propagator properly as it generates the initial conditions for the fine propagators. The coarse solutions are generally less accurate but play an essential role in the convergence of the algorithm [33].

3. Architecture Overview

PRA is implemented on a multicore homogeneous computing architecture and heterogeneous computing architecture. In this section, a brief description of homogeneous computing architectures is provided. The detailed description of heterogeneous computing architecture is provided in [42].

3.1. Homogeneous Computing Architecture

PRA is implemented on the homogeneous computing architecture using the OpenMP programming model. PRA's performance is analyzed on two Intel processors: Intel Xeon processor code-named Haswell and Intel Xeon Phi processor code-named Knights Landing.

3.1.1. Haswell Processor

PRA is implemented on the Haswell (HSW) processor using the OpenMP programming model. The HSW processor is the successor of the Ivy Bridge processor. It is a multicore processor with better vector processing compatibility compared to the Ivy Bridge processor. The hardware block level diagram of the HSW processor is shown in **Figure 3**. HSW is a dual-socket processor with 12 physical cores/socket with 30 MB L3 cache. With hyperthreading enabled, each core supports two logical threads. Therefore, a total of 48 logical threads can be executed in parallel. The multithreaded execution on the hardware enables us to compute 48 fine grids of the PRA in parallel, reducing the computational time.

HSW processor supports advanced vector extension (AVX) 2 instruction set with 256 wide registers, which means it can hold up to 8 single-precision floating-point values or 4 double-precision floating-point values in a register. The AVX instructions allow us to vectorize the application, *i.e.*, perform single instruction multiple data (SIMD) operation and add performance improvement and parallel execution. Data alignment plays a vital role in increasing vectorization opportunities since misaligned data access reduces load and store operations efficiency. **Figure 4** shows the possible ways of data alignment. If the data is aligned, only one clock cycle is needed to perform load/store operations.

In contrast, misaligned data requires multiple CPU clock cycles to perform the same load/store operations. Therefore, the efficient utilization of the vector registers requires data alignment. HSW processor is suitable for vector operations due to 256 bit wide registers and SIMD operations. In PRA, step 2 performs SIMD operation while computing the solution of the fine grids. With vectorization capabilities, additional performance improvement can be achieved for PRA execution using the HSW processor.

Solving ODEs involves multiple addition and multiplication operations to compute the numerical solution using PRA. In PRA, the ODEs are solved twice, once with the coarse propagator and another with the fine propagator. Modern Intel architectures like HSW have a feature that enables performing multiplication and addition in a single clock cycle called fused multiply-add (FMA). The use of FMA also maximizes vectorization capabilities and improves application

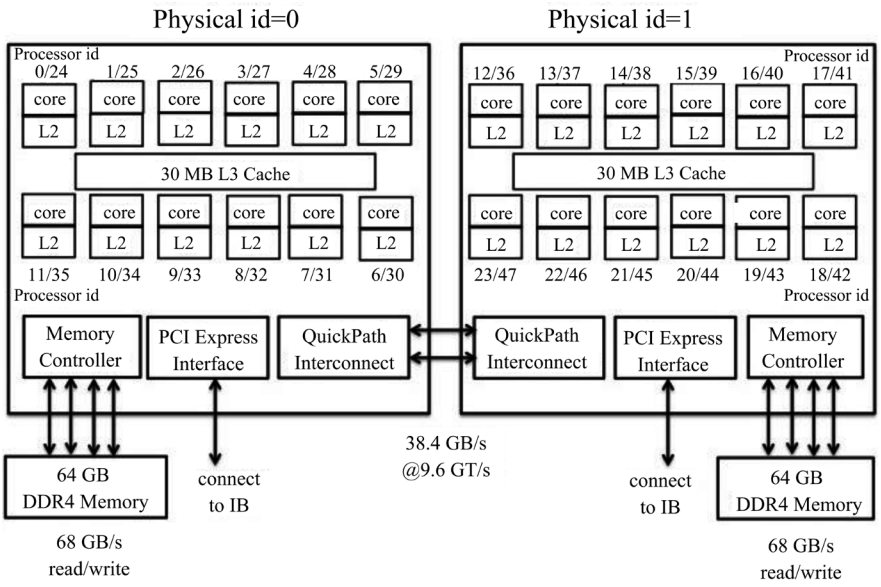

Figure 3. Haswell Architecture block diagram [44].

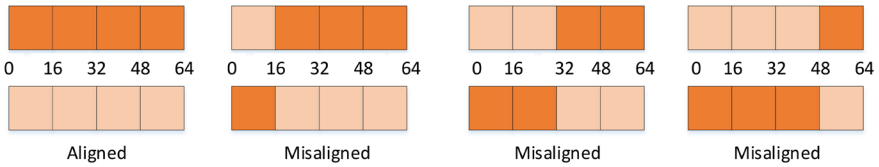

Figure 4. Possible ways of data alignment [45].

performance. The compiler flag **-xCORE-AVX2** is required to utilize FMA with the 256 wide registers present on HSW. This flag enables special instructions like FMA, which maximizes the utilization of the vector registers and AVX version 2 instruction set. Solving the differential equations using PRA involves performing a number of multiplication and addition operations in a single equation in both coarse and fine propagators. Using this flag, the code is optimized by the compiler for the potential FMA operations in both propagators that improves PRA's performance.

3.1.2. Knight's Landing Processor

The PRA is implemented on multicore homogeneous computing architectures called Knights Landing (KNL). KNL is the second generation Intel Xeon Phi architecture and successor to the Intel Xeon Phi coprocessor introduced in 2012. KNL mainly targets high-performance computing by delivering massive thread parallelism, data parallelism, and memory bandwidth in a CPU for high throughputs [46]. KNL is also binary compatible with Intel Xeon processors, *i.e.*, HSW and Broadwell. KNL also has the processor chip capable of supporting the AVX512 instruction set extension, which doubles the vector registers' width for SIMD operations. Unlike the previous generation Xeon Phi coprocessor, KNL is a standalone system with self-boot capability, eliminating the PCIe bottleneck issue due to the data transfer with a host processor. The KNL is highly suitable

for PRA implementation due to many salient hardware features. Two salient features that make KNL more suited for PRA implementation compared to HWS are the 256 simultaneous multithread execution on 64 computing cores on a single socket and the associated thirty-two 512 bit wide registers for SIMD or vector operations.

Figure 5 shows the block diagram of the KNL CPU processor. The architecture description [47] highlights the new features of KNL compared to the first-generation Knights Corner coprocessors. KNL CPU design introduces the concept of tile, the basic computational unit which is replicated. Figure 6 depicts the block diagram of each tile in KNL. Each tile comprises two cores, two vector processing units (VPU) per core, and a 1 megabyte of level-2 (L2) cache and Cache/Homing Agent (CHA) shared between two cores. There are 38 tiles out of which at most 36 tiles are active for computation.

Also, each tile has its own cache 1 MB L2 cache. This feature of KNL is very beneficial, especially while solving the fine propagator of PRA. In the case of PRA implementation, there are ~ten variables that are required in each iteration and are made available using the load/store operations. Each fine propagator is assigned to a thread on the core. Assuming the fine propagator is iterated 100 times to solve the fine propagator value, each thread will require about 4 KB of memory to compute the fine solution *i.e.* 10 floating-point variables iterated for 100 times. Therefore, each core requires ~16 KB of memory since KNL supports 4 threads/core. Hence, the total memory required by the tile is ~32 KB, is much less than 1 MB of L2 cache that is available on the tile. Since the memory requested by each fine propagator is less than the available L2 cache on the tile, the frequently used data can be read/stored on the tile's cache itself for all 8 fine propagators (threads). The lower memory requirement for each thread reduces the cache misses for the fine propagators resulting in PRA's performance improvement.

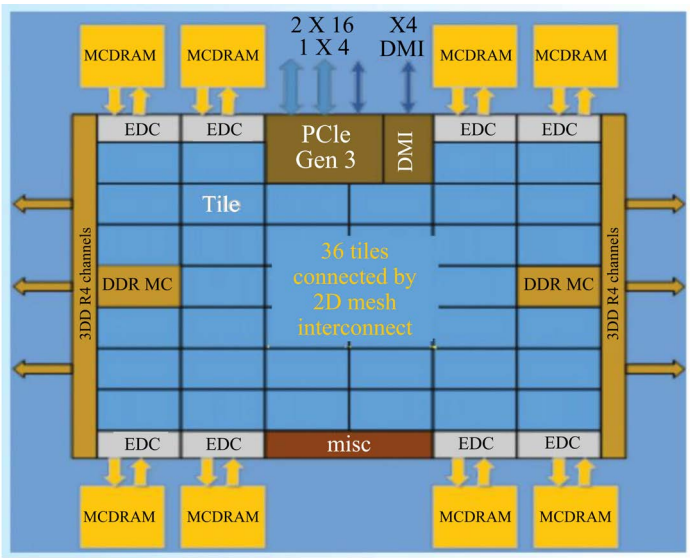

Figure 5. KNL block diagram [47].

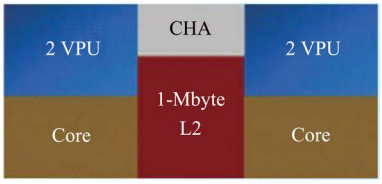

Figure 6. Block diagram of each tile [47].

KNL has two types of memory: MCDRAM and DDR together provide both high bandwidth and large capacity for applications. MCDRAM is in-package high bandwidth memory of size 16 GB to boost the performance. The bandwidth is 450 GB/s and 380 GB/s for stream and read-only bandwidth. The MCDRAM can be configured into several modes at boot time. The first mode is called cache mode, where the MCDRAM is used as a cache for DDR. This mode provides the application benefits of high bandwidth memory cache. In this mode, the 64 B cache lines are direct-mapped cache. The second mode is called flat mode, in which the MCDRAM is treated as a standard memory in the same address space as DDR. Here, the default memory is DDR. The last mode is the hybrid mode, a combination of cache and flat modes. Here, a portion of MCDRAM is cache, and the remaining is flat.

Each core is a two-wide, out-of-order core derived from the Intel Atom processor microarchitecture code-named Silvermont [48]. KNL has significant modifications to the Silvermont microarchitecture to incorporate features to handle high-performance computing workloads. Some of the features include support for four threads per core, more L1 cache, higher L1 and L2 bandwidths, the addition of AVX512 vector instruction set, and many more. KNL has 512-bit wide vector registers, which can hold sixteen single-precision numbers or eight double-precision numbers in each register, making effective use of the AVX512 instruction set.

The compiler flag -xMIC-AVX512 can be used to effectively use the AVX 512 instruction set during compilation on KNL. The flag makes sure that both VPUs are used for computational tasks. Intel AVX512 is a comprehensive instruction set architecture with higher performance than predecessor AVX and AVX2 instruction set architectures. Four functionalities are supported by this compiler flag for KNL architecture [49], and out of which, Intel AVX-512 foundation instructions (AVX-512F) functionality is used for PRA implementation. (AVX-512F) are the base of Intel AVX-512. They include extensions of the Intel AVX and Intel AVX2 family of SIMD instructions. The assembly code generated uses AVX-512 IA.

4. Implementation

This section discusses the PRA implementation details using different programming models. The PRA is implemented to parallelize the numerical integration algorithm for solving a system of ODEs. The system of ODEs is the mathematical model of the synchronous generator used to perform the transient

stability analysis (TSA) of a power grid [40]. The number of ODEs is a function of the generator model order or level used in the TSA. If the TSA focus does not include subtransients, the classical model of a generator consisting of two first-order ODEs is solved at each time step. If the subtransients are taken into account, the fourth-order model of a generator consisting of four first-order ODEs is solved at each time step.

4.1. OpenMP

PRA is implemented using the C++ programming language with OpenMP for multithreaded execution on the Intel processors. The code snippet of PRA implementation using C++ with OpenMP is shown in **Figure 7**. In **Figure 7**, it can be seen that the fine propagators are parallelized by annotating the C++ code with OpenMP directives (code in red color).

The executions on the CPU showed in **Figure 7** uses the Fork-Join approach of OpenMP. The executions are:

- The master thread computes the coarse solutions of the ODE sequentially shown as step 1 in **Figure 7**.
- The master thread initiates the fork by spawning multiple child threads to compute the fine solutions in parallel, shown as step 2 in **Figure 7**. Each child thread is assigned with one fine propagator. Each fine propagator using the initial condition at time $T_{n\text{-}1}$ (coarse solution) computes the fine solution at time T_n.
- Each fine propagator computes the solution at T_n using the time step δt sequentially. If the simulation time is decomposed into N intervals, N fine propagators are executed in parallel.
- After the fine propagators have computed the fine solutions in step 2, the fine propagator threads join the master thread, shown as step 3 in **Figure 7**.
- In step 3, the master thread performs predictor-corrector sequentially to update the solutions.

Figure 7 shows the data are aligned with 64-byte boundary [50] for better vectorization and performance improvement. The code developed for KNL and HSW is identical since the optimization techniques are applicable across all types of Intel processors. Therefore, we have a single optimized PRA codebase working on different Intel processors. The same codebase is compiled using different compiler flags reflecting the different instruction sets of KNL and HSW architectures.

4.2. CUDA

NVIDIA GPUs with CUDA programming model is used for accelerating the parallel step of the PRA algorithm. CUDA provides simple language extensions to programming languages like C, C++, FORTRAN, and Python to expose the fine and coarse grain parallelism. The application needs to be refactored using the CUDA APIs to offload the parallelizable portion of the application on to the GPUs.

```
main()
{
// Allocating memory for data alignment on Intel processors
    float *a = (float*)_mm_malloc((sizeof(float))*(N),64);
    float *b = (float*)_mm_malloc((sizeof(float))*(N),64);

    for(int i=1; i<N; i++)
    {
        compute coarse solution
    }                                                   Step 1

    // PRA iterations
    for(int k=1; k<K_max; k++)
    {
        #pragma omp parallel for private(t_n-1,t_n) shared(var1, var2, var3,…) num_threads(n)
        #pragma vector aligned
        for(int i=1; i<N; i++)
        {
                compute fine solution
        }                                               Step 2

        for(int i=1; i<N; i++)
        {
                Predictor-Corrector
        }                                               Step 3

        Update the coarse values
        Check for the tolerance
    }

    //Freeing the allocated memory
    _mm_free(a);
    _mm_free(b);
}
```

Figure 7. Code snippet for PRA implementation using OpenMP.

The C programming language version of CUDA is used in implementing the PRA for execution on GPUs. From the pseudocode presented in **Figure 2**, the PRA algorithm consists of three major steps, out of which steps 1 and 3 can only be executed sequentially. In contrast, step 2 can be executed in parallel.

- For the GPGPU implementation, the PRA's sequential steps are executed on the host (CPU), and the parallel step of the PRA executed on the device or accelerator (GPU).
- First, the coarse solutions computed on the host are copied from the host-to-device for use by the fine propagators.
- After the fine solutions are computed on the device, the fine solutions are copied back from the device-to-host for the predictor-corrector step.
- The corrected coarse values again copied to the device for the next iteration of the fine propagators.
- Therefore, the memory transfers back and forth between host and device in each iteration increase overall computation time.

In **Figure 8**, the code snippet of PRA implementation using CUDA is shown. The code in black color in **Figure 8** is associated with the host execution. The code in red color in **Figure 8** is associated with the device (GPU) execution. In the research [51], the process of developing optimized CUDA based code is discussed.

4.3. OpenACC

Implementing an algorithm using CUDA for heterogeneous computing involves a significant amount of time and effort to refactor existing sequential code for executing on GPUs. Also, the CUDA programming model requires extensive

memory management, which is a time-consuming process. Therefore, most HPC researchers do not opt for rewriting the existing application code in CUDA for heterogeneous computing, even though the best execution time with high scaling efficiency can be achieved. Also, the CUDA programming model is specific to NVIDIA GPUs. At the same time, supercomputers employ a variety of computing architectures. There is a need for having a single codebase that can be executed on various HPC architectures. In recent years, several programming models like OpenACC, Kokkos [52], oneAPI [53], and many more are developed to achieve performance by porting an existing application code targeting different HPC platforms. The OpenACC programming is a directive-based programming model for porting codes targeted for execution on multiple HPC platforms. Unlike CUDA, OpenACC provides rapid tools to parallelize and port an existing code for execution on GPUs and other architectures with minimal code refactoring.

```
main()
{
    //allocate the memory on the device for the variables
    cudaMalloc((float**)&d_a, N));
    cudaMalloc((float**)&d_b, N));

    for(int i=1; i<N; i++)
    {
        compute coarse solution
    }                                          Step 1

    // PRA iterations
    for(int k=1; k<Kmax; k++)
    {
            // copy the coarse values on the host to the coarse values on the device
            cudaMemcpy(d_a, h_a, N, cudaMemcpyHostToDevice);
            cudaMemcpy(d_b, h_b, N, cudaMemcpyHostToDevice);

            // create grid and block sizes
            dim3 block (x, y, z);
            dim3 grid (x, y, z);

            // Launch the kernel to compute fine solutions
            gpupararеal<<<grid, block>>>(arg1,arg2,arg3,…);   Step 2: Kernel launch on the CPU

            //copy the solution computed by fine propagators from device to the host
            cudaMemcpy(h_a, d_a, N, cudaMemcpyDeviceToHost);
            cudaMemcpy(h_b, d_b, N, cudaMemcpyDeviceToHost);

        for(int i=1; i<N; i++)
        {
                Predictor-Corrector
        }                                      Step 3
        Update the coarse values
        Check for the tolerance
    }
    // Free the allocated memory on the device
    cudaFree(d_a);
    cudaFree(d_a);
}

// GPU kernel function

__global__ void gpupararеal(arg1, arg2, arg3, ...)
{
        const int idx = threadIdx.x + (blockIdx.x*blockDim.x);
        if(idx>n)
        {
                return;
        }
         DifferentialEquations( arg1, arg2, arg3,…);
}                                              Step 2: Execution on the GPU

__device__ void  DifferentialEquations( arg1, arg2, arg3,…)
{
        Compute the solution for differential equations
}
```

Figure 8. Code snippet for PRA implementation using CUDA.

PRA is implemented using the OpenACC programming model to execute on the GPUs. The code is written in the C++ programming language. PGI compiler is used for compiling the code. The code snippet of PRA implementation using OpenACC is shown in **Figure 9**. The code snippet is color-coded to differentiate the host (CPU) and the device (GPU) code. The code in red executes on the device, and the code in black color executes on the host. Steps 1 and 3 are executed on the host while step 2 annotated with OpenACC directives execute in parallel on the device (GPU).

The code snippet in **Figure 9** can be observed to have the code flow similar to the pseudocode shown in **Figure 2**. In OpenACC implementation, the OpenACC directives are annotated to advise the compiler which portion of the code needs to be parallelized. The sequential code can be easily parallelized using OpenACC directives and clauses and achieve performance on multiple HPC architectures. In this paper, we present the OpenACC clauses used for improving the performance of GPU execution. Since the code is implemented using the C++ programming language, the keyword *pragma* is used as the compiler directive to highlight the parallelization code block. The compiler directive is followed by the directive type "*acc*" for OpenACC directives. Followed by the directive type, the compute directive is inserted. There are two types of "*compute*" directives that can be inserted to parallelize the code block. They are kernels and parallel. Kernel directive provides hints to the compiler to look for parallelism in a block of code and parallelize.

In contrast, the parallel directive is an assertion to the compiler to parallelize the code. The parallel directive is used when the developer has prior knowledge about the code suitability for parallelizing. In PRA porting, the parallel construct is used to assert the code block of fine propagators that needs to be parallelized. The syntax *#pragma acc parallel* directive in **Figure 9** identifies the region of the code that needs to be offloaded onto the GPUs.

The OpenACC directive is augmented with different clauses to assist the compiler and achieve better performance. The two main clauses used in step 2 in **Figure 9** are:

- The "*workers*" clause is a clause that distributes the parallelism in three levels to split the work across different hardware units. The three levels are *gang*, *worker,* and *vector*.
  - *Vector* is the finest granularity of a GPU SIMT.
  - *The gang* is the most coarse-grained, which works independently of each other and may not synchronize.
  - *Worker* defines how work is distributed inside a gang.

Figure 10 gives the comparison of the work distribution clauses in OpenACC with CUDA. In **Figure 10**, the vector and worker identify the number of threads along the x and y dimensions of a block, respectively. This hierarchy is similar to CUDA. The number of workers and vector length can be specified using the clauses *num_workers*, *m,* and *vector_length, n*. The compiler automatically generates the number of gangs based on *m*, *n,* and total simulation time for the PRA.

```
// Function executed on GPU
#pragma acc routine
void  DifferentialEquations(arg1, arg2, arg3,…)
{
Compute the solution for differential equations
}

main()
{
    for(int i=1; i<N; i++)
    {
        compute coarse solution
    }

    // PRA iterations
    for(int k=1; k<Kmax; k++)
    {
        #pragma acc parallel num_workers(m) vector_length(n)
        copyin(var1, var2, var3,…) create (var1, var2, var3,…)
        copyout(var1, var2, var3,…)
        {
        #pragma acc loop independent gang
                for(int i=1; i<N; i++)
                {
                        DifferentialEquations( arg1, arg2, arg3,…);
                }
        }
        for(int i=1; i<N; i++)
        {
                Predictor-Corrector
        }
        Update the coarse values
        Check for the tolerance
    }
}
```

Step 2: Solved on the GPU

Step 1

Step 2

Step 3

Figure 9. Code snippet for PRA implementation using OpenACC.

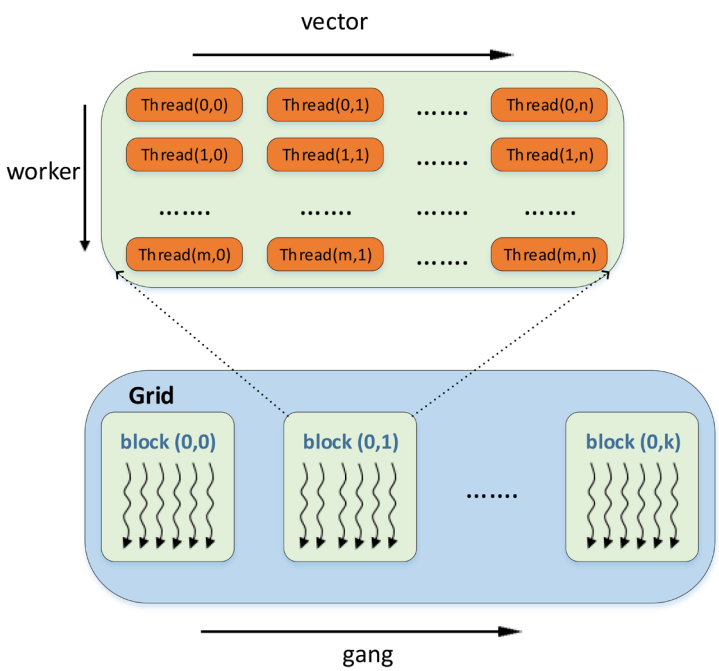

Figure 10. Organization of worker, vector, and the gang in OpenACC in comparison to CUDA thread hierarchy [54].

- The data clause is used to override the default compiler analysis of specified variables movement between the host and the device. The data clause controls how and when to copy the data to and from the device. In **Figure 9**, the three data clauses used for data movement between the host and the device are shown.
  - The first clause is the *copyin* clause that allocates memory on the device for all the variables listed inside the parentheses. This clause initializes the variables by copying data to the device at the beginning of the region. Once the parallel region completes the execution, memory is released on the device.
  - The second data clause used is the *create* clause, which creates all the listed variables on the device. The variables created on the device are local to the device and cannot be copied back to the host. The memory is freed on the device once the execution is complete.
  - The third clause is *copyout,* where all the computed values stored in the variables listed are copied back to the host from the device at the end of the parallel region.

For solving the ODEs, first, the state variables of the differential equations are copied to the device before computation of the fine solution similar to CUDA. The temporary variables used in CUDA implementation to minimize the global memory access are created here using the *create* clause. The variables are created on the device, and the intermediate solutions are computed using these variables.

The *#pragma acc loop* construct is used for defining the work distribution of the for-loop. Augmenting the construct *#pragma acc loop* with additional clauses independent and gang results in the best performance. These two clauses allow the execution of each loop independently and partition the loops across gangs. Similar to the CUDA implementation, a kernel function is launched by the host to execute on the GPU. The function is identified as the kernel function by decorating the function with the *#pragma acc routine*. Similar to CUDA and OpenMP versions, each thread is associated with one fine propagator. The solution is computed for T_n with $T_{n\text{-}1}$ as the initial condition with time step δt.

5. Performance Results

The test system described in [40] is used for the time-domain simulations. The accuracy of the numerical simulations using PRA has been presented in [40]. The focus here is on the computing performance across the different types of accelerators and programming models. The performance is analyzed using three performance metrics: parallel runtime, speedup, and scalability of parallelization referred to as the scaling efficiency. The general definition of parallel runtime or the execution time is the time that elapses from the instance a parallel computation starts to the instance the last processor finishes execution. The parallel runtime metric is dependent on the application and hardware architecture.

For homogeneous computing, PRA is implemented on HSW and KNL pro-

cessors. HSW processor used for the PRA implementation is the Intel Xeon CPU E5-2670 v3 @2.30 - 3.10 GHz with 24 physical cores. With hyperthreading enabled, 48 logical threads are available for computations. KNL [55] processor used for PRA implementation is Intel Xeon Phi CPU 7210 @1.30 - 1.4 GHz with 64 physical cores and 256 logical cores. The total execution time T_{PRA} of the PRA for the OpenMP programming model on multicore core CPU architectures is given in Equation (4).

$$T_{PRA} = t^c + \sum_{i=1}^{N} \left(t^f + t^{pc} \right) \tag{4}$$

where,

t^c is the computation time of the coarse propagator.

t^f is the computation time for the fine propagator.

t^{pc} is the computation time for predictor-corrector.

N is the number of PRA iterations.

For heterogeneous computing, PRA is implemented on a server having an Intel Xeon CPU E5-2670 @2.30 GHz, interfaced through the PCIe bus to the NVIDIA Quadro RTX 6000 GPU hosting 4608 computing cores with 24 GB GPU memory [56]. The parallel runtime is modified to address the GPU architecture. The T_{PRA} is the sum of four components, as shown in Equation (5) [40].

$$T_{PRA} = t_H^c + \sum_{i=1}^{N} \left(t_H^G + t_G^f + t_G^H + t_H^{pc} \right) \tag{5}$$

where,

t_H^c is the computation time of the coarse propagator on the CPU.

t_H^G is the memory transfer latency between the host and the GPU.

t_G^f is the computation time of the fine propagators in parallel on the CPUs.

t_G^H is the memory transfer latency between the GPU and the host.

t_H^{pc} is the computation time of the predictor-corrector on the host.

N is the number of iterations.

The speedup metric is defined as the ratio of the serial or sequential runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem using p processors. The speedup is given by Equation (6) [40].

$$\text{Speedup} = \frac{T_{seq}}{T_{PRA}} \tag{6}$$

where,

T_{seq} is the computation time of the sequential approach.

T_{PRA} is the execution time of the PRA on parallel computing architectures.

The general definition of the scaling efficiency metric is the ratio of speedup to the number of processors. Even though the definition of the scaling efficiency metric involves the number of processors used to parallelize, the number of processors may not be used directly in measuring the scaling efficiency. In the PRA performance analysis, the number of threads running in parallel is used instead of the number of processors. Since the hardware architectures considered

in this research support multiple threads per processor, using the number of processors will result in a higher scaling. Furthermore, the number of threads spawned by the PRA is dependent on the number of fine propagators, which dictates the accuracy of the solution.

The coarse propagator computation time is dependent on the coarse propagator time step t_{step}^{c} and the fixed interval of time T for which the ODEs are solved. For a fixed T, the coarse propagator computational time will increase with smaller t_{step}^{c}. The number of fine propagators N^{f} corresponding to a coarse propagator time step t_{step}^{c} and for a given T is

$$N^{f} = \frac{T}{t_{step}^{c}} \tag{7}$$

The scaling efficiency is classified as strong and weak scaling. The strong scaling indicates that for a fixed problem size, with an increasing number of processors, the scaling efficiency is linear or superlinear. In contrast, weak scaling indicates the speedup is constant or decreasing with an increasing number of processors. Therefore, good performance of PRA is indicated by low execution runtime, large speedup, and at least linear speedup increase (strong scaling) with an increasing number of fine propagators.

By varying N^{f}, the number of threads running in parallel is varied, and varying the fine propagator time step t_{step}^{f} the computation load of each thread is varied. The speedup achieved using the PRA is demonstrated through several simulations with varying t_{step}^{c} or N^{f}, and t_{step}^{f} for the classical and detailed model.

5.1. Classical Model

The performance statistics were collected for the classical model using all four versions of PRA implementation. In Table 1, the execution time for the classical model with varying N^{f} for sequential and all four versions of PRA are provided. In Table 1, it can be observed that the execution time for the PRA implementation is significantly less compared to the traditional sequential approach.

Figure 11 shows the speedup graph for all four versions of the code executed on different computing architectures and programming models with varying N^{f}. For the OpenMP-HSW version, the speedup increases linearly up to 256 fine propagators, showing strong scaling efficiency for the parallel execution. On increasing N^{f} further, the speedup increases nonlinearly. The overhead for spawning the threads and switching stalled threads contributes to the nonlinear scaling. A speedup of 16x is achieved for the PRA on the Haswell architecture with OpenMP. The speedup obtained using OpenMP-KNL is significantly higher for the classical model compared to the HSW processor. The PRA algorithm shows strong scaling with the 36x speedup achieved. The scaling curve is similar to what is observed with HSW execution but has better performance. The number of threads spawned on KNL for solving fine propagators is five times that of

the OpenMP threads on HSW. Therefore, the number of threads stalled for the execution on KNL is less, which improves the performance. Since the PRA consists of a sequential portion, and due to lower processor clock speed, the performance does not scale up linearly with five times the number of processors on KNL. In the case of CUDA-GPU, the speedup increases linearly with an increase in N^f exhibiting a strong scaling efficiency The strong scaling efficiency is due to the t_G^f being significantly large compared to the sum of the other components in Equation (5). The maximum speedup achieved was 25x with the CUDA-GPU version. The CUDA-GPU implementation provides better performance when the fine propagator computation load is large, *i.e.*, smaller t_{step}^f. For OpenACC-GPU, the speedup increases linearly, similar to CUDA, and shows strong scaling efficiency. The maximum speedup achieved using OpenACC is 19x for larger N^f. However, the OpenACC-GPU performance is lower compared to CUDA-GPU. The performance of OpenACC is ~75% of the CUDA performance for the classical model. However, the maximum possible performance is not achieved using GPUs because the number of fine grids running in parallel is significantly less than the number of computing cores available on the GPU, resulting in the inefficient use of the GPU.

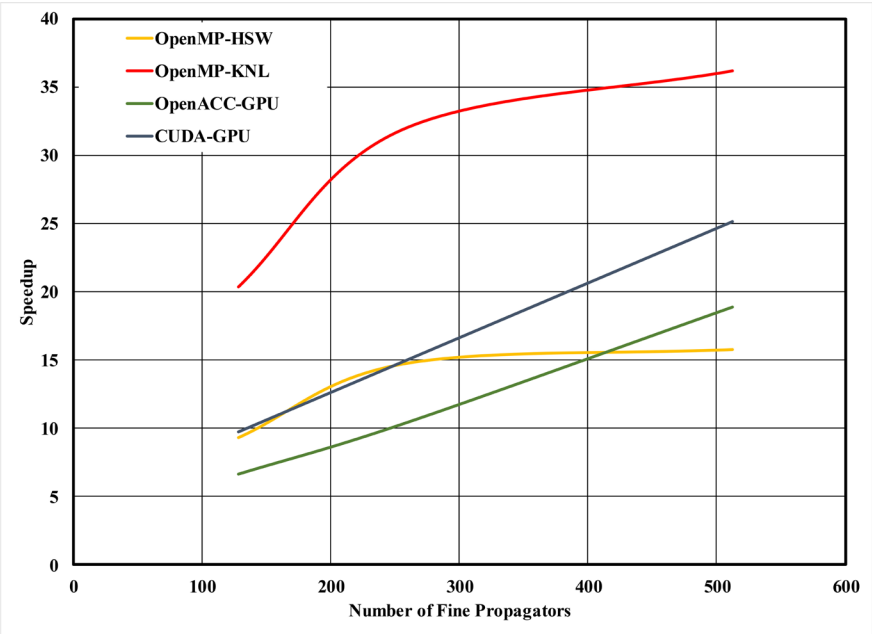

Figure 11. Speedup with varying N^f for the Classical model.

Table 1. Execution time for Classical model with varying N^f.

t_{step}^c (ms)	t_{step}^f (μs)	N^f	Execution Time (ms)				
			Sequential	OpenMP-HSW	OpenMP-KNL	OpenACC-GPU	CUDA-GPU
20	200	128	1.778	0.191	0.087	0.267	0.183
10	100	256	3.594	0.245	0.113	0.347	0.242
5	50	512	7.105	0.452	0.196	0.376	0.283

5.2. Detailed Model

The implementation of the detailed model consists of computing the numerical solutions of four ODEs at each time step. In Table 2, the execution time for the detailed model for sequential and all four versions of PRA are provided. It can be observed in Table 2, the execution time for the PRA implementation, even with computing the solutions of four ODEs at each time step, is significantly less compared to the traditional sequential approach.

Figure 12 shows the comparison of speedup for all the four versions of the code executed on different computing architectures and programming models with varying N^f for the detailed model.

The speedup increases marginally until N^f equals 12,800 and then decreases for OpenMP-HSW. However, the algorithm does not scale linearly with an increase in N^f. The maximum speedup achieved is 18.5x. It is important to note that the number of ODEs solved sequentially in the detailed model is twice the number of ODEs solved in the classical model. This makes the detailed model more compute-intensive compared to the classical model. The total simulation time for the detailed model is significantly more compared to the classical model. From Equation (7), N^f is directly proportional to T, which results in more N^f running in parallel compared to the classical model. Also, there are only 48 threads that are running in parallel due to the number of actual hardware cores but the N^f dictates the use of at least 2560 threads which is the minimum number of fine propagators for the detailed model from the Table 2. All these factors account for the weak scaling performance of the PRA with the increase in N^f. The OpenMP-KNL performs slightly better compared to OpenMP-HSW only for a smaller number of fine propagators. The speedup decreases with an increase in N^f indicating weak scaling with KNL also. As the N^f increases, the performance of OpenMP-KNL reduces in comparison to OpenMP-HSW, and for N^f greater than 5120, the OpenMP-HSW processor performs better. The detailed model having more number of equations solved at each time step sequentially; the lower clock speed of the KNL processor contributes to a larger execution time. For CUDA-GPU also, it can be seen that the speedup does not increase linearly and flattens with increasing N^f indicating a weak scaling efficiency. The weak scaling is due to the sum of the coarse propagator computation time t_H^c and the memory transfer latencies (t_H^G, t_G^H) being larger compared to the fine propagators' computation time t_G^f. The t_H^c is mainly due to four ODEs solved at each time step sequentially and larger memory transfer latencies to transfer the larger coarse propagator solutions from host to GPU and vice versa. The maximum speedup achieved is 31x for CUDA-GPU. The scaling curve of OpenACC-GPU exhibits weak scaling, similar to CUDA PRA. Even though the performance of OpenACC is lower compared to the CUDA implementation, the difference is less when compared to the performance with the classical model. The kernel launch time has a huge overhead in the case of OpenACC. Therefore, when the computations on the device are less due to

Table 2. Execution time for the Detailed model with varying N^f.

t^c_{step} (ms)	t^f_{step} (μs)	N^f	Execution Time (ms)				
			Sequential	OpenMP-HSW	OpenMP-KNL	OpenACC-GPU	CUDA-GPU
10	100	2560	40.786	2.573	2.209	2.233	1.875
5	50	5120	72.57	4.553	4.425	3.04	2.728
2	20	12,800	159.665	8.621	11.052	6.144	5.043
1	10	25,600	289.96	15.972	22.02	13.395	9.714

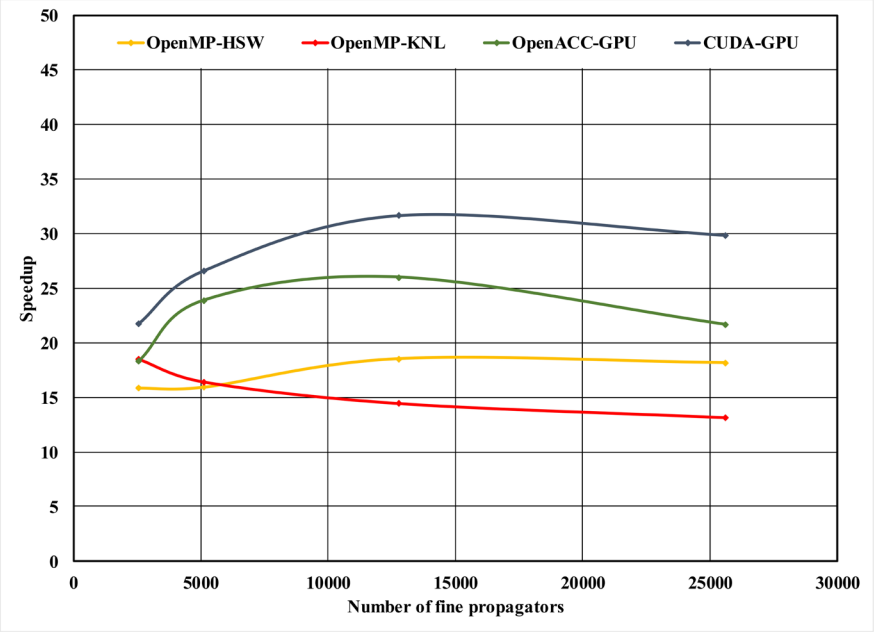

Figure 12. Speedup with varying N^f for the Detailed model.

smaller simulation time, the launch time latency negatively impacts the performance. To hide the kernel launch time latency, solving a larger number of ODEs with longer simulation times is more suitable for OpenACC. The best speedup achieved was 26x. The performance of OpenACC is observed to be approximately 83% of CUDA implementation, with less refactoring of the sequential code for execution on GPUs.

A list of advantages and disadvantages is presented to summarize the findings of PRA implementation on different HPC architectures using different programming models.

- The CUDA-GPU implementation has the highest speedup for a higher number of ODEs solved in each time step.
- OpenACC-GPU implementation has a better performance compared to the OpenMP CPU/KNL implementation for the detailed model. The performance on the GPU is better than the OpenMP CPU/KNL implementations due to a larger number of cores. A large number of cores on the GPU allow a larger number of parallel execution of fine propagators. The OpenACC com-

piler using the directives generates a generic CUDA code that is not optimized for a particular application and leads to a performance lower than the application-specific CUDA implementation.

- The CUDA implementation requires refactoring the code to execute on GPU, and the refactoring requires significant development time and effort. Furthermore, the refactoring of the code has to be performed repeatedly to achieve the best performance on newer architectures of GPUs. The CUDA code portability across different hardware architectures does not exist.
- Both OpenACC and OpenMP codes have the advantage of minimal refactoring with portability across different hardware platforms.
- The scaling of all implementations is dependent on the order of dynamic modeling of the complex system. The number of ODEs solved at each time step and dependency between the ODEs is a function of the system dynamic model order. The increase of sequential execution time results in weaker scaling. For example, the transient stability analysis of the power grid by utilities is typically performed using models consisting of twenty-seven ODES. Solving twenty-seven ODEs in each time step sequentially will weaken the scaling further.

The weak scaling could be improved by using more hardware resources or modify the PRA to reduce the effect of the sequential execution part.

6. Conclusion

In this paper, we investigated the performance of PRA to solve the system of time-dependent ODEs representing using homogeneous and heterogeneous computing architectures. PRA is implemented on the Intel Xeon processor code-named HSW and Xeon Phi processor code-named KNL using the OpenMP programming model. Intel CPU and NVIDIA GPUs are used for heterogeneous computing with CUDA and OpenACC programming models. The data alignment optimization technique is used in the code with readily available optimization flags to improve vectorization on multicore Intel Architectures. For the classical model with two first-order ODEs, the KNL outperformed the GPUs and HSW processors. For the detailed model with more number of equations, the performance of GPUs is significantly better than the KNL and HSW processors. PRA is an iterative algorithm, and a significant amount of time is spent on the sequential steps of the algorithm in the detailed model since four first-order ODEs are solved in each time step. This impacts the overall performance of KNL making GPUs suitable for more compute-intensive problems. In future work, methods will be explored to reduce the computation burden caused by the sequential part of the algorithm to exploit further the data parallelism and improve the overall performance.

Acknowledgements

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

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Alligood, K.T., Sauer, T.D. and Yorke, J.A. (1996) Fractals. In: Alligood, K.T., Sauer, T. and Yorke, J., Eds., *Chaos: An Introduction to Dynamical Systems*, Springer, Berlin, 149-191. https://doi.org/10.1007/978-3-642-59281-2_4

[2] Kantz, H. and Schreiber, T. (2004) Nonlinear Time Series Analysis (Vol. 7). Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511755798

[3] Weather Forecasts. https://markets.businessinsider.com/news/stocks/ibm-makes-higher-quality-weather-forecasts-available-worldwide-1028689623

[4] Koradi, R., Billeter, M. and Güntert, P. (2000) Point-Centered Domain Decomposition for Parallel Molecular Dynamics Simulation. *Computer Physics Communications*, **124**, 139-147. https://doi.org/10.1016/S0010-4655(99)00436-1

[5] Xue, W., Shu, J. and Zheng, W. (2004) Parallel Transient Stability Simulation for the National Power Grid of China. In: *International Symposium on Parallel and Distributed Processing and Applications*, Springer, Berlin, 765-776. https://doi.org/10.1007/978-3-540-30566-8_89

[6] Esmaeili, S. and Kouhsari, S.M. (2007) A Distributed Simulation Based Approach for Detailed and Decentralized Power System Transient Stability Analysis. *Electric Power Systems Research*, **77**, 673-684. https://doi.org/10.1016/j.epsr.2006.06.008

[7] Weiss, R.M. and Shragge, J. (2013) Solving 3D Anisotropic Elastic Wave Equations on Parallel GPU Devices. *Geophysics*, **78**, F7-F15. https://doi.org/10.1190/geo2012-0063.1

[8] Baffico, L., Bernard, S., Maday, Y., Turinici, G. and Zérah, G. (2002) Parallel-in-Time Molecular-Dynamics Simulations. *Physical Review E*, **66**, Article ID: 057701. https://doi.org/10.1103/PhysRevE.66.057701

[9] Staff, G.A. and Rønquist, E.M. (2005) Stability of the Parareal Algorithm. In: *Domain Decomposition Methods in Science and Engineering*, Springer, Berlin, 449-456. https://doi.org/10.1007/3-540-26825-1_46

[10] Newnes. Nievergelt, J. (1964) Parallel Methods for Integrating Ordinary Differential Equations. *Communications of the ACM*, **7**, 731-733. https://doi.org/10.1145/355588.365137

[11] Miranker, W.L. and Liniger, W. (1967) Parallel Methods for the Numerical Integration of Ordinary Differential Equations. *Mathematics of Computation*, **21**, 303-320. https://doi.org/10.1090/S0025-5718-1967-0223106-8

[12] Burmeister, J. and Horton, G. (1991) Time-Parallel Multigrid Solution of the Navier-Stokes Equations. In: *Multigrid Methods III*, Birkhäuser, Basel, 155-166. https://doi.org/10.1007/978-3-0348-5712-3_10

[13] Horton, G. (1992) The Time-Parallel Multigrid Method. *Communications in Applied Numerical Methods*, **8**, 585-595. https://doi.org/10.1002/cnm.1630080906

[14] Kiehl, M. (1994) Parallel Multiple Shooting for the Solution of Initial Value Problems. *Parallel Computing*, **20**, 275-295. https://doi.org/10.1016/S0167-8191(06)80013-X

[15] Hackbusch, W. (1985) Parabolic Multigrid Methods. *Proceedings of the 6th International Symposium on Computing Methods in Applied Sciences and Engineering*,

Vol. 6, 189-197.

[16] Lions, J.L., Maday, Y. and Turinici, G. (2001) Résolution d'EDP par un schéma en temps Ťpararéel ť. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, **332**, 661-668. https://doi.org/10.1016/S0764-4442(00)01793-6

[17] Farhat, C. and Chandesris, M. (2003) Time-Decomposed Parallel Time-Integrators: Theory and Feasibility Studies for Fluid, Structure, and Fluid-Structure Applications. *International Journal for Numerical Methods in Engineering*, **58**, 1397-1434. https://doi.org/10.1002/nme.860

[18] Minion, M. (2011) A Hybrid Parareal Spectral Deferred Corrections Method. *Communications in Applied Mathematics and Computational Science*, **5**, 265-301. https://doi.org/10.2140/camcos.2010.5.265

[19] Emmett, M. and Minion, M. (2012) Toward an Efficient Parallel in Time Method for Partial Differential Equations. *Communications in Applied Mathematics and Computational Science*, **7**, 105-132. https://doi.org/10.2140/camcos.2012.7.105

[20] Christlieb, A.J., Macdonald, C.B. and Ong, B.W. (2010) Parallel High-Order Integrators. *SIAM Journal on Scientific Computing*, **32**, 818-835. https://doi.org/10.1137/09075740X

[21] Christlieb, A.J., Haynes, R.D. and Ong, B.W. (2012) A Parallel Space-Time Algorithm. *SIAM Journal on Scientific Computing*, **34**, C233-C248. https://doi.org/10.1137/110843484

[22] Friedhoff, S., Falgout, R.D., Kolev, T.V., MacLachlan, S. and Schroder, J.B. (2012) A Multigrid-in-Time Algorithm for Solving Evolution Equations in Parallel. No. LLNL-CONF-606952, Lawrence Livermore National Lab. (LLNL), Livermore.

[23] Gander, M.J. and Neumüller, M. (2014) Analysis of a Time Multigrid Algorithm for DG-Discretizations in Time.

[24] Bal, G. and Maday, Y. (2002) A "Parareal" Time Discretization for Nonlinear PDE's with Application to the Pricing of an American Put. In: *Recent Developments in Domain Decomposition Methods*, Springer, Berlin, 189-202. https://doi.org/10.1007/978-3-642-56118-4_12

[25] Maday, Y. and Turinici, G. (2003) Parallel in Time Algorithms for Quantum Control: Parareal Time Discretization Scheme. *International Journal of Quantum Chemistry*, **93**, 223-228. https://doi.org/10.1002/qua.10554

[26] Staff, G. (2003) Convergence and Stability of the Parareal Algorithm: A Numerical and Theoretical Investigation. No. NTNU-N-2003-2, SIS-2003-312.

[27] Samaddar, D., Newman, D.E. and Sánchez, R. (2010) Parallelization in Time of Numerical Simulations of Fully-Developed Plasma Turbulence Using the Parareal Algorithm. *Journal of Computational Physics*, **229**, 6558-6573. https://doi.org/10.1016/j.jcp.2010.05.012

[28] Gurrala, G., Dimitrovski, A., Pannala, S., Simunovic, S. and Starke, M. (2015) Parareal in Time for Fast Power System Dynamic Simulations. *IEEE Transactions on Power Systems*, **31**, 1820-1830. https://doi.org/10.1109/TPWRS.2015.2434833

[29] Duan, N., Dimitrovski, A., Simunovic, S. and Sun, K. (2016) Applying Reduced Generator Models in the Coarse Solver of Parareal in Time Parallel Power System Simulation. 2016 *IEEE PES Innovative Smart Grid Technologies Conference Europe* (*ISGT-Europe*), Ljubljana, 9-12 October 2016, 1-5. https://doi.org/10.1109/ISGTEurope.2016.7856184

[30] Duan, N., Dimitrovski, A., Simunovic, S., Sun, K., Qi, J. and Wang, J. (2018) Embedding Spatial Decomposition in Parareal in Time Power System Simulation. 2018

IEEE Power & Energy Society Innovative Smart Grid Technologies Conference, Washington DC, 19-22 February 2018, 1-6. https://doi.org/10.1109/ISGT.2018.8403389

[31] Bal, G. (2005) On the Convergence and the Stability of the Parareal Algorithm to Solve Partial Differential Equations. In: *Domain Decomposition Methods in Science and Engineering*, Springer, Berlin, 425-432. https://doi.org/10.1007/3-540-26825-1_43

[32] Gander, M.J. and Hairer, E. (2008) Nonlinear Convergence Analysis for the Parareal Algorithm. In: *Domain Decomposition Methods in Science and Engineering XVII*, Springer, Berlin, 45-56. https://doi.org/10.1007/978-3-540-75199-1_4

[33] Nielsen, A.S. (2012) Feasibility Study of the Parareal Algorithm. Doctoral Dissertation, MSc Thesis, Technical University of Denmark, Denmark.

[34] Harden, C.R. (2008) Real Time Computing with the Parareal Algorithm. Doctoral Dissertation, Florida State University, Tallahassee.

[35] Ruprecht, D. and Krause, R. (2012) Explicit Parallel-in-Time Integration of a Linear Acoustic-Advection System. *Computers & Fluids*, **59**, 72-83. https://doi.org/10.1016/j.compfluid.2012.02.015

[36] Eghbal, A., Gerber, A.G. and Aubanel, E. (2017) Acceleration of Unsteady Hydrodynamic Simulations Using the Parareal Algorithm. *Journal of Computational Science*, **19**, 57-76. https://doi.org/10.1016/j.jocs.2016.12.006

[37] Bedez, M., Belhachmi, Z., Haeberlé, O., Greget, R., Moussaoui, S., Bouteiller, J.M. and Bischoff, S. (2016) A Fully Parallel in Time and Space Algorithm for Simulating the Electrical Activity of a Neural Tissue. *Journal of Neuroscience Methods*, **257**, 17-25. https://doi.org/10.1016/j.jneumeth.2015.09.017

[38] Subramaniam, A.S. and Upadrasta, R. (2018) Optimization and Parallelization of Tensor and ODE/PDE Computations on GPU. Doctoral Dissertation, Indian Institute of Technology Hyderabad, Sangareddy District.

[39] Arteaga, A., Ruprecht, D. and Krause, R. (2015) A Stencil-Based Implementation of Parareal in the C++ Domain-Specific Embedded Language STELLA. *Applied Mathematics and Computation*, **267**, 727-741. https://doi.org/10.1016/j.amc.2014.12.055

[40] Lakshmiranganatha, S. and Muknahallipatna, S.S. (2020) Graphical Processing Unit Based Time-Parallel Numerical Method for Ordinary Differential Equations. *Journal of Computer and Communications*, **8**, 39-63. https://doi.org/10.4236/jcc.2020.82004

[41] Chapman, B., Jost, G. and Van Der Pas, R. (2007) Using OpenMP. Portable Shared Memory Parallel Programming.

[42] Cheng, J., Grossman, M. and McKercher, T. (2014) Professional CUDA c Programming. John Wiley & Sons, Hoboken.

[43] Farber, R. (2016) Parallel Programming with OpenACC. Elsevier, Amsterdam. https://doi.org/10.1016/B978-0-12-410397-9.00001-9

[44] Intel Haswell Processor. https://www.nas.nasa.gov/hecc/support/kb/haswell-processors_492.html

[45] Chen, N. and Johnson, R. (2010) Patterns for Cache Optimizations on Multi-Processor Machines. *Proceedings of the* 2010 *Workshop on Parallel Programming Patterns*, March 2010, 1-10. https://doi.org/10.1145/1953611.1953613

[46] Jeffers, J., Reinders, J. and Sodani, A. (2016) Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition. Morgan Kaufmann, Burlington. https://doi.org/10.1016/B978-0-12-809194-4.00002-8

[47] Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R. and Liu, Y.C. (2016) Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, **36**, 34-46. https://doi.org/10.1109/MM.2016.25

[48] Kuttana, B. (2013) Technology Insight: Intel Silvermont.

[49] AVX512 ISA. https://software.intel.com/en-us/articles/compiling-for-the-intel-xeon-phi-processor-and-the-intel-avx-512-isa

[50] Data Alignment to Assist Vectorization. https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

[51] Kumar, R., Muknahallipatna, S. and McInroy, J. (2016) An Approach to Parallelization of SIFT Algorithm on GPUs for Real-Time Applications. *Journal of Computer and Communications*, **4**, 18-50. https://doi.org/10.4236/jcc.2016.417002

[52] Edwards, H.C., Trott, C.R. and Sunderland, D. (2014) Kokkos: Enabling Many-Core Performance Portability through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, **74**, 3202-3216. https://doi.org/10.1016/j.jpdc.2014.07.003

[53] oneAPI. https://www.oneapi.com

[54] Gang, Worker, and Vector with OpenACC. https://www.microway.com/hpc-tech-tips/accelerating-code-with-openacc-and-nvidia-visual-profiler/gang_worker_vector

[55] Intel Xeon Phi. https://ark.intel.com/content/www/us/en/ark/products/94033/intel-xeon-phi-processor-7210-16gb-1-30-ghz-64-core.html

[56] Quadro RTX 6000 GPU. https://www.nvidia.com/en-us/design-visualization/quadro/rtx-6000

List of Abbreviations

AVX	Advanced Vector Extension
CHA	Cache/Homing Agent
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
EDD	Exact Domain Decomposition
FMA	Fused Multiply-Add
GPU	Graphical Processing Unit
HPC	High-Performance Computation
HSW	Haswell
KNL	Knights Landing
MPAS	Model for Prediction Across Scales
ODE	Ordinary Differential equation
PDE	Partial Differential Equations
PFASST	Parallel Full Approximation Scheme in Space and Time
PITA	Parallel In Time Algorithm
PRA	Parareal Algorithm
SIMD	Single Instruction Multiple Data
TSA	Transient Stability Analysis
VPU	Vector Processing Unit