

Parallelization in Time: Applications to Plasma Turbulence

L. A. Berry¹, W. Elwasif¹, J. Reynolds-Barredo^{2,3}, D. Samaddar⁴, R. Sanchez³ and D. E. Newman²

¹Oak Ridge National Laboratory, P. O. Box 2008 MS 6169, Oak Ridge TN 37831, USA

²University of Alaska, Fairbanks AK, USA

³University Carlos III de Madrid, Madrid, Spain

⁴ITER Organization, Saint Paul Lez Durance, France

Email: berryla@ornl.gov

Abstract. Parareal is an iterative algorithm that, in effect, achieves temporal decomposition for a time-dependent system of differential or partial differential equations. A solution is obtained in a shorter wall-clock time, but at the expense of increased compute cycles. The algorithm combines a fine solver that solves the system to acceptable accuracy with an approximate coarse solver. The critical task for the successful implementation of parareal on any system is the development of a coarse solver that leads to convergence in a small number of iterations compared to the number of time slices in the full time interval, and is, at the same time, much faster than the fine solver. Fast coarse solvers may not lead to sufficiently rapid convergence, and slow coarse solvers may not lead to significant gains even if the number of iterations to convergence is satisfactory. We find that the difficulty of meeting these conflicting demands can be substantially eased by using a data-driven, event-based implementation of parareal. For this implementation, for example, tasks for one iteration do not wait for the previous iteration to complete, but are started when the needed data are available. For given convergence properties, the event-based approach relaxes the speed requirements on the coarse solver by a factor of $\sim K$, where K is the number of iterations required for a converged solution. This may, for many problems, lead to an efficient parareal implementation that would otherwise not be possible or would require substantial coarse solver development.

1. Introduction

Modern high-performance computers utilize thousands to hundreds of thousands of processors in parallel to support simulations with increasingly detailed descriptions of the underlying science. However, the highly nonlinear nature of the time evolution of these systems often leads to long (sometimes impossibly long) run times for simulations of interest. For magnetic fusion, extended MHD [1,2] and short-wavelength plasma turbulence [3] are two examples of this type of problem. For example, an ITER [4] discharge may last a thousand seconds, while simulations of extended MHD and plasma turbulence are limited to a very small fraction of that time for realistic physics. Advances

in algorithms and use of ever-larger high-performance computing contribute to narrowing this gap but, in many cases, may not be sufficient, and a range of algorithms will have to be developed and/or implemented in the future to address this general issue. In this paper we discuss one such approach, parareal.

Parareal is an algorithm for effectively utilizing high-performance parallel computers to, in effect, decompose the time domain to obtain the numerical solution to a time dependent system of differential or partial differential equations in a shorter wall-clock time at the expense of increased compute cycles [5]. Successful applications include molecular dynamics [6], fluid dynamics [7], and plasma turbulence [8]. Parareal utilizes a fine solver, F , that, over time domains of interest, advances the target system with acceptable accuracy. Functionally, F is a propagator that advances the system state, for example, from time $t = t_{i-1}$ and state λ_{i-1} to time $t = t_i$ and state λ_i . It is described by notation $\lambda_i = F_{\Delta T}(\lambda_{i-1})$ with $\Delta T = t_i - t_{i-1}$. The desired solution over the interval $[t_0, t_N] = N\Delta T$ is then given by $\lambda_N = F_{N\Delta T}(\lambda_0)$, where the initial conditions are given by λ_0 .

The second element of parareal is a coarse solver, G . The coarse solver must be much faster than the fine solver but will be less accurate. However, it must be sufficiently accurate to enable rapid convergence. Techniques for developing a coarse solver include reduced spatial resolution, reduced time resolution, different basis functions, or even a simplified system of equations. While specific mathematical requirements (apart from speed) are given in [5], the effectiveness of G is, in practice, determined by testing. As with F , the notation $\lambda_i = G_{\Delta T}(\lambda_{i-1})$ is also used to describe G . The wall-clock times for executing a step of ΔT for the coarse and fine solvers are given by T_G and T_F , respectively, with the ratio denoted by $\beta = T_F / T_G$. The notation $\lambda_{k,i}^{G/F}$ will be employed to distinguish between states for the coarse/fine solvers (G/F), for iteration index k at the end of slice i . These states are explicitly the result of applying propagators to input states. In addition to G , a method for evaluating convergence; initial states $\lambda_{1,0}^G$ and $\lambda_{1,0}^F$; and, if the states are not compatible, operators for transforming states between the coarse and fine solvers are needed. Depending on the coarse solver, the incompatibility could arise because of grid dimensions, basis sets, or even independent variable choice. Whenever states are used as arguments or in an operator statement, use of the appropriate transformation, for example, $\lambda_{k,i}^G \Rightarrow \lambda_{k,i}^F$ within $\lambda_{k,i+1}^F = F_{\Delta T}(\lambda_{k,i}^G)$, is implied and must be carried out before the propagator is applied.

The iterative state update is the defining element of parareal. The update for the input state for the present iteration, present time slice is $\Lambda_{k,i-1} = \lambda_{k,i-2}^G - \lambda_{k-1,i-2}^G + \lambda_{k-1,i-2}^F$. The notation λ is used to distinguish between a state that is the result of a propagator, $G(\lambda)$ or $F(\lambda)$, and the state Λ that is the result of the linear combination of states defined in the previous sentence that constitutes the parareal iterate. If needed, operators to transform states for one solver space to the other are implied. This update depends only

on fine results from the previous iteration, thus allowing all the fine tasks for a given iteration to proceed in parallel after the sequential coarse steps are completed for that iteration.

The statement of the parareal algorithm in the previous paragraphs naturally leads to a sequential implementation of parareal that alternates sequential execution of coarse tasks and parallel execution of fine tasks. However, we note that the state update does not require that all of the coarse steps for a given iteration be completed before starting the fine step that depends on that update. A given fine step i can be started as soon as the $(i-1)$ parareal update (using results from the $(i-1)$ coarse step) is completed. This suggests that a data-driven, event-based implementation might be possible. Since some coarse tasks can be done in parallel with fine tasks, there is the expectation that the need for a very fast coarse solver might be at least partially reduced.

Details of the classical, sequential parareal implementation are presented in Section 2. The steps are discussed in detail in order to provide background for describing the improved implementation that is described in Section 3. The performance of the two implementations is analyzed in Section 4 for the plasma turbulence application presented in [8]. Section 5 shows how computational work may be reduced from the standard parareal algorithm by extending the total simulation time as slices converge. This technique may be effective when the total simulation time exceeds capability of the coarse e.g., when the convergence error is or order unity. A summary and future research are presented in Section 6. Both the sequential and event-based parareal algorithms were implemented by using a lightweight Python framework, the Integrated Plasma Simulator (IPS) that was developed for multiphysics simulations of magnetically confined fusion plasmas [9, 10, 11] and is described in the Appendix.

2. Sequential Parareal

The workflow for sequential parareal implementation follows from the algorithm description present in the previous section. However, the details are slightly different for the first, second, and subsequent iterations. These details are described below in order to provide background for the event-based workflow.

First iteration:

1. Compute (sequentially) $\lambda_{1,i+1}^G = G_{\Delta t}(\lambda_{1,i}^G)$ for $i = 0, 1, \dots, N-1$. The total execution time for this step is given by NT_G .
2. Calculate $\lambda_{1,1}^F$ by applying the fine propagator to $\lambda_{1,0}^F$. For $i = 2, 4, \dots, N-1$, the $\lambda_{1,i}^F$ are obtained from $F_{\Delta t}(\lambda_{1,i-1}^G)$ using the coarse- to fine-grain transformation if needed. All the operations in this step can be done in parallel. The wall-clock time is given by T_F , giving a total execution time for the first iteration of $NT_G + T_F$.

3. Since there are no prior iterations, no convergence tests are possible.*

Second iteration

1. For the first coarse step, compute $\lambda_{2,2}^G = G_{\Delta t}(\lambda_{1,1}^F)$. The slice, with state $\lambda_{1,1}^F$, is converged since it is given by $\lambda_{1,1}^F = F_{\Delta t}(\lambda_{0,1}^F)$, the “exact” solution. For slices $i = 3, 4, \dots, N-1$, (sequentially) compute $\lambda_{2,i}^G = G_{\Delta t}(\Delta_{2,i-1}^G)$ using $\Delta_{2,i-1}^G = \lambda_{2,i-1}^G - \lambda_{1,i-1}^G + \lambda_{1,i-1}^F$.
2. Apply the fine propagator to $\lambda_{1,1}^F$ for $i = 2$ and to $\Lambda_{2,i-1}^G = \lambda_{2,i-1}^G - \lambda_{1,i-1}^G + \lambda_{1,i-1}^F$ for $i = 3, 4, \dots, N-1$, all in parallel.
3. Test for convergence for $i = 3, 4, \dots, N-1$. A convergence test for $i = 2$ is not required because on the third iteration slice $i = 2$ will be converged since $\lambda_{2,2}^F = F_{\Delta t}(\lambda_{1,1}^F)$, the “exact” solution. Designate the first nonconverged slice by $i_{f\text{irst}}$. The total time for the second iteration is again $\approx NT_G + T_F$. For $k \ll N$, the total time is again $\approx NT_G + T_F$.

Subsequent (k th) iterations

1. As in the second iteration, for $i_{f\text{irst}}$ compute $\lambda_{k,i_{f\text{irst}}}^G = G_{\Delta t}(\lambda_{k-1,i_{f\text{irst}}}^F)$ and then (sequentially) $\lambda_{k,i}^G = G_{\Delta t}(\Delta_{k,i-1}^G)$ using $\Delta_{k,i-1}^G = \lambda_{k,i-1}^G - \lambda_{k-1,i-1}^G + \lambda_{k-1,i-1}^F$ for $i = i_{f\text{irst}}+1, i_{f\text{irst}}+2, \dots, N-1$. We are (again) using the fine output from the last converged slice on the previous iteration as the input to the first nonconverged slice during the next iteration.
2. Following “2” above, apply the fine propagator to $\lambda_{k-1,i_{f\text{irst}}}^F$ for $i = i_{f\text{irst}}$ and, in parallel, to $\lambda_{k,i-1}^F = G_{\Delta t}(\lambda_{k,i-1}^G) - G_{\Delta t}(\lambda_{k-1,i-1}^G) + F_{\Delta t}(\lambda_{k-1,i-1}^F)$ for $i = i_{f\text{irst}}+1, i_{f\text{irst}}+2, \dots, N-1$.
3. Test for convergence for $i = i_{f\text{irst}}+1, \dots, N-1$. A convergence test for $i = i_{f\text{irst}}$ is not required because on the third iteration slice $i = i_{f\text{irst}}$ will be converged since $\lambda_{k,i_{f\text{irst}}}^F = F_{\Delta t}(\lambda_{k-1,i_{f\text{irst}}}^F)$, the “exact” solution. The total time is again $\approx NT_G + T_F$. This estimate is high because the actual number of coarse tasks is smaller than N but is sufficiently accurate to provide an understanding of trends as is done in the next section.

In Reference [8], the parareal algorithm was successfully applied to a model plasma turbulence using a single-executable with complex internal management of the MPI calls.

* Convergence tests can be implemented in a number of ways. For the BETA application, the normalized difference of a time-averaged energy measure of the fine solver was used. Thus convergence tests could be applied only on the second iteration. If the comparison were made between a measure of convergence that could be applied between fine and coarse states, then convergence tests could have been applied on the first iteration.

For the present effort, we used the fine, coarse, converge, coarse-to-fine, fine-to-coarse, and parareal-advance methods (five independent executables) from that effort to recreate the algorithm within the IPS simulation framework.

An IPS simulation is formulated in terms of components. While many definitions are possible for a component, for our purposes a component implements a particular functionality with defined data interfaces to other components. Thus, to implement parareal, we implemented coarse, fine, and converge components. In addition, a driver component controlled the sequence of component operations, calling the coarse, fine, and converge components sequentially for each iteration. These components are, at the simplest level, Python wrappers for the executables that carry out the operations described in the previous section.

The Python components developed for this application can serve as templates for new applications with only relatively small changes to the scripts. As result, the additional programming for each new application at the parallelization level, for example, MPI communicator management, required to integrate the coarse, fine, and converge tasks within one executable is not necessary.

The convergence history for one run with this new implantation is shown in Figure 1.

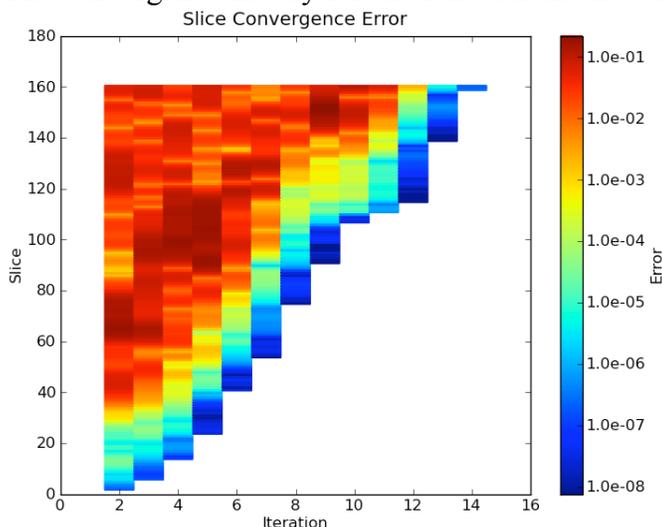


Figure 1. Convergence history for a 160-slice beta simulation using parareal. No values are shown for the first iteration as there is no data from a “zeroth” iteration.

was attained in 14 iterations using a tolerance of $1.5e-6$ on the time-integrated energy for each slice. Additional detail can be found in [8]. There were no differences between the present results and those previously obtained by executing the component elements of parareal within one executable: the numerical algorithms were the same; thus the results are the same.

Iterations to convergence is one measure of performance for parareal. A better measure is the wall-clock gain. At the simplest level, with $T_G \ll T_F$, the gain H is given by the ratio of the time for a full solution with the fine solver applied sequentially, NT_f ,

The total time for the simulation was $N\Delta T = 12800\tau_{di}$ where τ_{di} is a characteristic ion diamagnetic drift time. This interval was divided into $N = 160$ slices of length $\Delta T = 80\tau_{di}$ each. A total of 1,024 processors (128 nodes with eight processors each) were used in the run. The fast Fourier transforms in the fine and coarse solvers were parallelized using eight and four processors each, respectively. The VODPK [12] adaptive integrator was used in the fine solver while fourth-order Runge-Kutta was used in the coarse solver. Convergence

to the parareal wall-clock time KT_f or $H = N/K$. Finite T_G will modify the time per iteration from T_f to $T_f + NT_G$ and decrease the gain to

$$H = \frac{N}{K \left(1 + \frac{N}{\beta}\right)} \quad (0.1)$$

where, again, $\beta = T_f/T_G$.

As discussed previously, this expression slightly overestimates the impact of T_G because only on the first iteration will N sequential coarse solves be required, but it is sufficiently precise to understand the trends. The gain expression by itself is not particularly useful because K is a function of N and the “goodness” of the coarse solver. It does make clear, however, that there is a tradeoff between T_G and K in optimizing H . If we make the reasonable assumption that there is a trend for better coarse solvers to result in smaller K but have larger T_G , the net gain will be reduced when $N/\beta \geq 1$. For this case, even if the system converges more rapidly, the extra sequential time taken by the coarse solver could well reduce the net gain. For example if K were reduced by a factor of two at the expense of N/β going from one to four, the net gain would be reduced by 20%.

The impact of a too-slow coarse solver can be seen in Figure 2. The processor

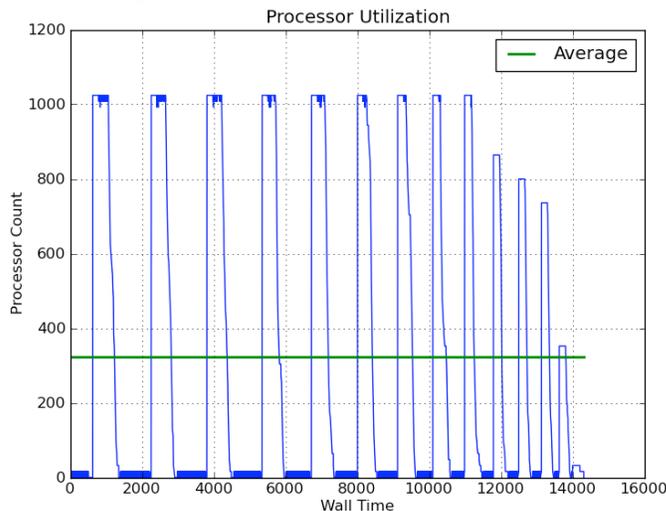


Figure 2, Processor utilization for the simulation displayed in Figure 1. The peak utilization falls off for wall times over 11,000 seconds because the number of fine updates is no longer large enough to occupy the number of available processors.

utilization during the ~14000 second sequential run of Figure 1 is shown, and the periods for coarse tasks (with very low utilization) and fine tasks (near 100% utilization) can be clearly distinguished. Well over half of the time was spent in the sequential coarse computations. The average processor utilization was limited to ~30% because the condition $N/\beta \ll 1$ was not met. Meeting this condition and also obtaining convergence in a small number of steps are the central issue for the application of parareal to a particular application.

3. Event -Based Parareal

During development of the IPS implementation of parareal, it became clear that while an implementation based on following the steps outlined in the previous section was, in a sequential programming sense, logical, it was also unduly restrictive and resulted in time

inefficiency. For example, in the first iteration, the first coarse task and first fine task could be carried out in parallel. Similarly, subsequent fine tasks could be performed as soon as the preceding coarse task was complete. For the second iteration the first fine task ($k = 2, i = 2$) could be carried out as soon as the fine task from the proceeding iteration and preceding time slice was complete. This realization led to the concept of a data-driven, event-based implementation of parareal. The algorithm and the data employed are the same, but task execution is started just as soon as the required data are available.

Component dependencies—coarse, fine, and converge, enclosed in the three groups of rows enclosed by dark borders and labeled by the bold task name in the first column—are summarized in Table 1 for all time slices except the first in a given iteration. Rows under the bold label designate the source of the dependency. The second through fourth columns refer to each of the three principal types of iterations described in the previous section. Dependencies are indicated by a darkened box in the table. These dependencies were extracted from the detailed descriptions in the previous section.

Table 1. Component dependencies

Dependencies for (k, i) th bold task	$k = 1$ and $i > 1$	$k = 2$ and $i > 2$				$k > 2$ and $i > i_{first}$		
Dependency	$i-1$	$1, i-1$	$1, i$	$2, i-1$	$2, i$	$k-1, i$	$k, i-1$	k, i
coarse								
coarse								
converge								
fine								
coarse								
converge								
converge								
fine								

As a simple example, consider the coarse task for the third slice, $i = 3$ of the second iteration, $k = 2$. The column labeled $k = 2$ and $i > 2$ in Table 1 shows the coarse dependences of slice $i = 2$, iterations $k = 1$ and $k = 2$ and the fine dependence of slice $i = 2$, iteration $k = 1$.

If we assume that a simulation length $N\Delta T$ converges in K iterations, an approximate gain can be estimated that is sufficiently accurate to understand the scaling characteristics of the implementation. To derive this estimate, we assume a convergence history where there are no converged slices until the last iteration. For this case, the data dependencies lead to the sequence of tasks $F_{1,1} \Rightarrow F_{2,2} \Rightarrow F_{3,3} \Rightarrow \dots \Rightarrow F_{K-1,K-1} \Rightarrow G_{K,K} \Rightarrow G_{K,K+1} \Rightarrow \dots \Rightarrow G_{K,N-1} \Rightarrow F_{K,N}$. This leads to a total time for the K iterations of $KT_F + (N - K)T_G$. The resulting gain is

$$H = \frac{N}{K + \frac{N - K}{\beta}}. \quad (0.2)$$

The event-based execution flow was implemented by using IPS event services and dependencies equivalent to those in Table 1. This implantation was applied to the same case as described in Section 2. The convergence history for the new implementation was the same as shown in Figure 1 for the original parareal work flow, as were the detailed numerical results. However, the total wall-clock simulation time decreased by

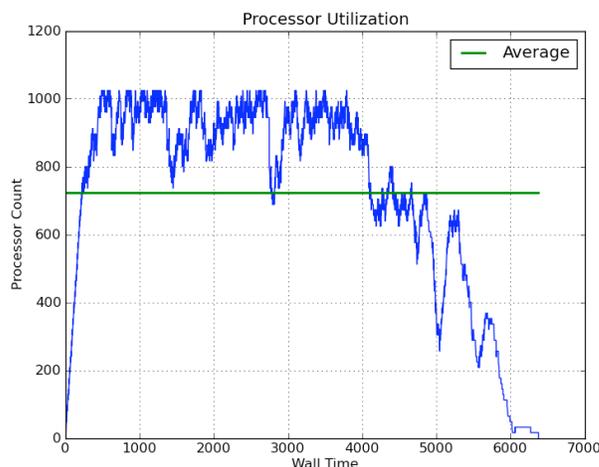


Figure 3. Processor utilization for the event-based parareal implementation. The processor utilization is now over 70%.

over 50% and the processor utilization increased by over 50% as presented in Figure 3. The initial ramp to near 100% utilization is a result of the time required for all fine tasks to start, NT_G , while the falloff was caused by the lack of fine tasks to execute in parallel as the simulation converged. This demonstration validated the premise that an event-based parareal could have better performance.

4. Analysis of Event-Based Parareal

In order to understand the performance characteristics of the event-based parareal implementation, two studies were conducted. The first used a test simulation case for BETA using the fourth-order Runge-Kutta coarse solver. The second used the empirical convergence model that was developed in [8].

The test case, with $N = 32$ and $\Delta T = 80\tau_{di}$, converged in seven iterations. Gain properties were varied by artificially increasing the coarse solver time, T_G by a “sleep” command in the coarse solver. These results are summarized in Figure 4 as a function of $\beta = \bar{T}_F / \bar{T}_G$ with \bar{T}_F and \bar{T}_G designating the average of, respectively, fine and coarse tasks in the simulation. Beta varied from less than five ($\beta \ll N$) to almost a hundred ($\beta > N$), encompassing the range of interest for the gain models described in Section 2. The measured simulation speedup times were based on the ratio of $N\bar{T}_f$ to the simulation wall-clock time. The predictions of both the event-based (Eq. (0.2), $H = N / (K + (N - K) / \beta)$), and sequential (Eq. (0.1) $H = N / K (1 + N / \beta)$) models are

also shown in Figure 4. The measured simulation times for the event-based implementation show gains of over two even when the sequential model values were less

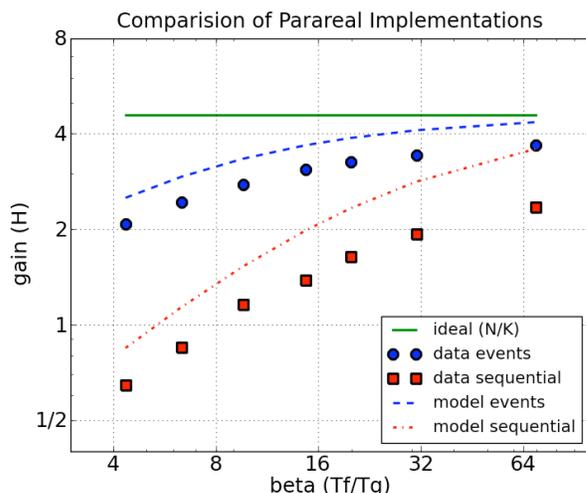


Figure 4. Comparison of speedups for the sequential (squares) to the event-based (circles) implementations. Corresponding model calculations are shown in the dashed and dash-dot lines. The solid line indicates the ideal gain ($N/K = 32/7$. for this test problem for the BETA mode lusing the RK4 coarse solver. The simulations had $N = 32$ and $\Delta T = 80\tau_{di}$ and converged in seven iterations.

both sequential and event-based implementations, peak values of T_F have a larger impact on the gain than does the average. This can most readily be seen for the sequential version where the time to complete fine steps for a given iteration is determined by the longest T_F . The effect of peak T_F on the event-based algorithm is not as obvious but can clearly slow progress when it is that particular iteration and the slice is a critical dependency. The drops in processor utilization during the first two-thirds of the simulation in Figure 3 are almost certainly due to this phenomenon. Second, with respect to the sequential implementation, the number of coarse tasks is overestimated by a factor of about 2, because the simple gain model assumes that N coarse tasks are needed for each iteration, while the actual number depends on how far convergence has progressed.

To extend our understanding of the performance potential for event-based parareal from one particular run to a broad range of conditions for BETA, we used the empirical model developed in [8] for the convergence characteristics coarse solvers based on the VODPK and second- and fourth-order Runge-Kutta time-integrators. The result of this analysis was an expression for the number of iterations required for convergence, $K(N, \Delta T)$, for each of the three integrators. This empirical model was based on the observation that the qualitative behavior of a given coarse solver could be characterized by a series of slow convergence iterations (\sim a few slices per iteration) up until a time t_1 followed by a faster rate of convergence, b slices per iterations, at time t_2 with a linear

than one. For large β , as suggested by the modeling, the event-based and sequential models are converging to the same gain, $\sim 4 \sim N/K = 4.6$, and are within $\sim 30\%$ of the measured values. The β s at which a 50% reduction in gain is observed is consistent with $\beta = N/K = 4.7$ for the event-based implementation and $N/\beta = 1$ for the sequential version.

The difference between measured and modeled gains for all β s is likely due to two factors. First, peak values of T_F for each iteration are typically a factor of 2 or more larger than the average because the VODPK integrator adapts time steps to achieve a given precision. For

ramp for times between t_1 and t_2 . Estimates for these constant coefficients were found from the properties of large number of runs for each of the coarse integrators. Despite the large uncertainties and variability in these parameters, they could still be used as a semi-quantitative model that was sufficient to analyze convergence trends. The analysis in this paper uses the parameters obtained in [8].

The model values for $K(N, \Delta T)$ for the three coarse solvers were used in the approximate gain expressions presented in Sections 2, Eq. (0.1) and 3, Eq. (0.2). Two types of scaling for a particular coarse solver were analyzed: weak scaling, where ΔT is held fixed, and strong scaling, where the total time, $N\Delta T$, was held fixed. In both cases N , the number of time steps, was varied. The number of processors (or groups of processors if the problem utilizes spatial domain decomposition) is also equal to N .

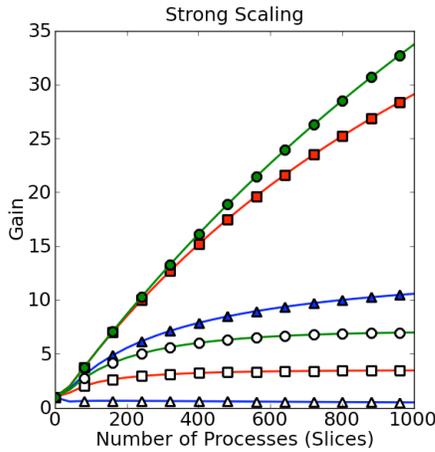


Figure 5. Strong-scaling (increasing slices, fixed simulation time $T = 25600\tau_{di}$) model results are presented for three coarse solvers: diamonds= \Rightarrow VODPK integrator; squares= \Rightarrow second-order Runge-Kutta; and circles= \Rightarrow fourth-order Runge-Kutta. For each solver, two estimates of wall-clock gain or speed up are shown—the lower curve for the standard implementation of parareal and the upper for the event-based algorithm.

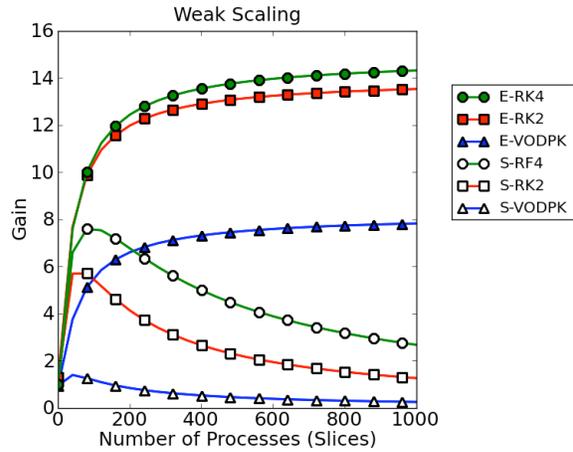


Figure 6. Weak-scaling (fixed $\Delta T = 80\tau_{di}$ per processor) model results are presented for three coarse solvers: diamonds= \Rightarrow VODPK integrator; squares= \Rightarrow second-order Runge-Kutta; and circles= \Rightarrow fourth-order Runge-Kutta.. For each solver, two estimates of wall-clock gain or speed up are shown—the lower curves (open symbols) for the standard implementation of parareal and the upper curves (filled symbols) for the event-based algorithm.

Figure 5 presents the results of the strong-scaling model for both the sequential and event-based implementations. Figure 6 present the results for weak-scaling analysis. For all cases, the event-based implementation has substantially improved performance. For a 600-slice simulation, strong scaling performance is three (fourth-order RK) to seven (VODPK) times better. More important, the improvement in performance is greatest for the poorest-performing solver for the sequential implementation. Thus the coarse solver

that would be judged as not useful (VODPK) now results in gains that are greater than were found with the best coarse sequential coarse solver (fourth-order RK). The same behaviors are observed for the weak-scaling analysis. In addition, the optimum in gain previously observed when the number of slices is $\sim \beta$ is no longer observed.

5. Dynamic Slice Addition

The previous sections have focused on reducing the wall-clock time completing a simulation, albeit at the expense of compute cycles. In many cases, improvements in compute efficiency over the traditional parareal algorithm can be obtained. This possibility can be seen in Figure 1 where, in this 160-slice simulation, we see that after about 30 slices the convergence error changes from the 10^{-8} to 10^{-3} range to over 10^{-1} . This suggests that the coarse solver is, in some sense, not effective past the $30\Delta T$ range and that calculations with the coarse solver and related fine solve runs are “wasted.” In order to test this observation, the IPS components were modified to add additional slices as early

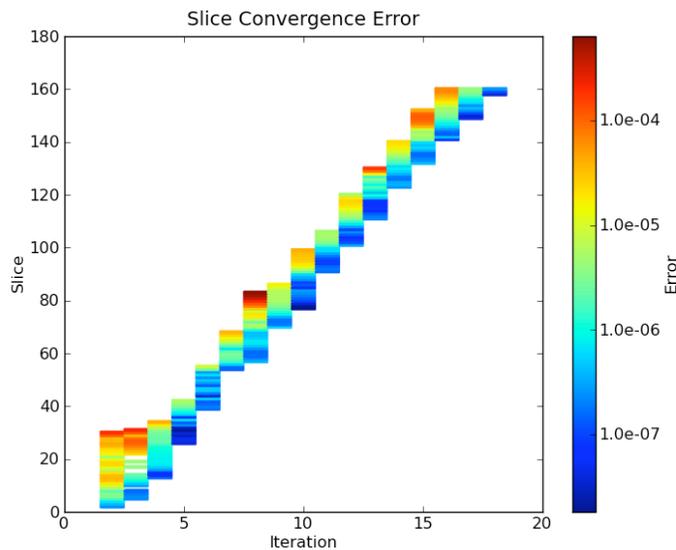


Figure 7. Convergence history for a 160-slice beta simulation using parareal with dynamic slice addition.

factor of ~ 20 improvement in processor utilization.

time slices converge. These slices were started using the same initialization as employed to start the simulation as described in Section 2. The results of this modification are shown in Figure 7. While the simulation took four additional iterations to converge (eighteen versus fourteen), the computational work has been reduced by a factor of ~ 4 . In addition, only 30 (or a few more, depending on details) processor groups are required for minimum wall-clock time, as opposed to the 160 required for the result in Figure 1, thus realizing a

6. Summary and Future Research

Sequential and event-based parareal algorithms have been implemented using the IPS framework. For this project, the target application was BETA, a model plasma turbulence application. However, the Python-based IPS allows development of parareal for new applications with a minimum of code modifications. Target applications include fusion transport simulations, gyrokinetic turbulence simulations, and extended MHD. Implementation of the event-based algorithm required only superficial modification of the application code used in the sequential implementation. Event-based parareal provides the potential for successful development of parareal to applications for which

previous efforts have not been successful, as well as improved performance for applications with successful parareal solvers. The key to this improvement is recognizing the data dependencies of parareal and beginning work on any task for which the data are available. As a result, requirements on the coarse solver are reduced from $\beta \geq N$ to $\beta \geq N/K$.

Acknowledgements

Research funded in part by Spanish National Project No. ENE2009-12213-C03-03. Part of the research was carried out at the University of Alaska Fairbanks, funded by the DOE Office of Science Grant No. DE-FG02-04ER54741. Work supported in part by the U.S. DOE under Contract DE-AC05-00OR22725 with UT-Battelle, LLC. The authors are grateful for grants of supercomputing resources at the University of Alaska’s Arctic Region Supercomputing Center (ARSC) in Fairbanks.

Appendix Implementation of Parareal Using the IPS

Implementation of parareal as described in Section 3 can be a daunting task. It, for example, requires a good understanding of how to manage multiple MPI communicators in order to control and launch the multiple executables for the fine and coarse solvers on different groups of processors and/or nodes. In order to reduce the programming burden for new parareal applications and to reduce the time for developing a successful parareal application, parareal was implemented using the Integrated Plasma Simulator (IPS).

The IPS is a lightweight Python framework for large parallel computers that provides services for multiphysics (or, more generally, multicomponent of any simulations for domain) simulations with file-based communication between components [9, 10, 11]. The IPS manages computing resources, launches tasks, moves input and output files, locally archives data, and provides event services using a publish and subscribe model. These functions are implemented in a set of managers as displayed in Figure 8. The IPS is shown running on a head node as is the case for Compute-Node Linux high-performance computers such as the Cray XT-5. (We note that the IPS, which has run on a wide range of systems including Cray XT-5s and “standard” Linux clusters, does not work on systems like the Blue Gene because of limitations on how multiple, independent, executables can managed.) Component executables are launched from the IPS as standard `mpirun` or `aprun` processes with a return to the IPS when finished.

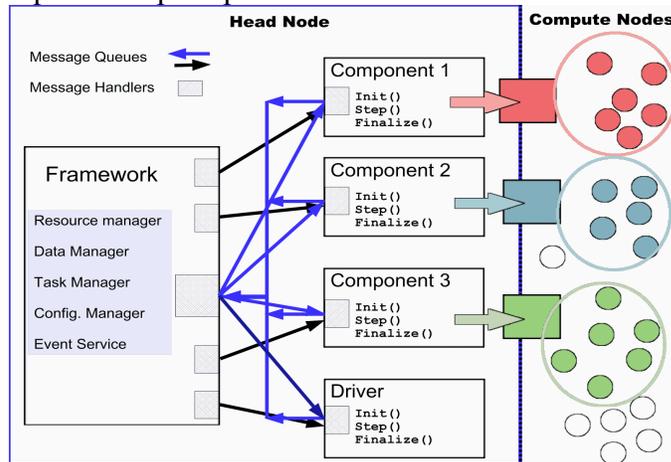


Figure 8. Block diagram of the IPS framework.

Simulations are composed of a set of components, each of which implements a needed functionality. The needed functionality for parareal is provided by coarse, fine, and converge components. These components are built from Python wrappers to (typically) parallel MPI application codes. An additional component, the driver, is a required component that defines and controls the execution flow of a simulation or an algorithm. Each component is required to implement Init, Step, and Finalize procedures, which can be empty. Checkpoint and restart methods are also required if that functionality is desired. A configuration file is used to specify component scripts, initial data files, input and output files, and global and component-specific configuration parameters. Global parameters (that can also be set and read by any component during a run by any component) include the number of time slices, maximum iteration count, and convergence status. Local configuration parameters include processor count for MPI, paths to binaries and to static input files (e.g., fortran namelists), and runtime code parameters.

For the conventional, sequential, parareal implementation, the driver provides an iteration loop in which the coarse, fine, and converge components are sequentially called. Within each component, loops over time slices used to execute the coarse and fine propagators and to test for convergence. The Resource Manager implements a task pool “queue” for managing compute resources in the event that processor groups are not currently available. Use of the task pool reduces the wall-clock gain from parareal but increases overall processor utilization and allows longer simulations within constrained resources. For the sequential parareal, first the coarse, then the fine, and finally the converge calculations for a given iteration were completed before proceeding to the next iteration.

For the data-driven, event-based implementation, tasks (coarse, fine, or converge) were launched as the needed data was produced. Neither iteration nor time loops were used. At completion, a given task published an event whose body contains data that, for example, included iteration and slice indices, absolute paths to input files and, for the converge component, the results of the convergence test. Components subscribe to events on which they depend. When the first dependency is satisfied, a task is created, then, when all dependences are satisfied, the task is launched. In order to begin the simulation, required dependencies were satisfied manually as initial conditions.

In moving from the sequential to event-based implementations, none of the underlying executables had to be changed. The same was true for the implementing the dynamic slice addition capability.

References

- [1] C.R. Sovinec, A.H. Glasser et al., *J. Comput. Phys.* 195 (2004) 355.
- [2] L. Chacon, *Comp. Phys. Comm.* 163 (2004) 143.
- [3] F. Jenko, W. Dorland, M. Kotschenreuther, and B.N. Rogers, *Phys. Plasmas* 7 (2000) 1904.
- [4] M. Shimada et al., *Nuclear Fusion* 47 (2007) S1.

- [5] J. Lions, Y. Maday, G. Turinici, “A parareal in time discretization of pde’s,” *CR Acad. Sci. I – Math.* 332 (7) (2001) 661–668.
- [6] L. Baffico, S. Bernard, Y. Maday, G. Turinici, G. Zérah, “Parallel in time molecular dynamics simulations,” *Phys. Rev. E* 66 (5) (2002) 057706.
- [7] P.F. Fischer, F. Hecht, and Y. Maday, in “Lecture Notes in Computational Science and Engineering,” 2005, p. 2017.
- [8] D. Samaddar, D.E. Newman, R. Sánchez, “Parallelization in time of numerical simulations of fully-developed plasma turbulence using the parareal algorithm,” *J. Comput. Phys.* 229 (18) (2010) 6558.
- [9] W.R. Elwasif, D.E. Bernholdt, L.A. Berry, and D.B. Batchelor, “Components framework for coupled integrated fusion plasma simulation,” in *HPC-GECO/CompFrame—Joint Workshop on HPC Grid Programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing*, Montreal, Canada, October 2007.
- [10] S.S. Foley, W.R. Elwasif, A.G. Shet, D.E. Bernholdt, and R. Bramley, “Incorporating concurrent component execution in loosely coupled integrated fusion simulations,” *Component-Based High- Performance Computing (CBHPC)*, Karlsruhe, Germany, 16-17 October 2008.
- [11] W.R. Elwasif, D.E. Bernholdt, A. Shet, S. Foley, R. Bramley, D.B. Batchelor, and L. Berry, “The design and implementation of the SWIM integrated plasma simulator,” in *18th Euromicro Int’l. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, Pisa, Italy, 17-19 February 2010.
- [12] S.D. Cohen and A.C. Hindmarsh. “CVODE, a stiff/nonstiff ODE solver,” *Computers in Physics* 10 (2) (1996) 138-143.