Event-based parareal: A data-flow based implementation of parareal


Oak Ridge National Laboratory, P.O. Box 2008 MS 6169, Oak Ridge, TN 37831, USA
University of Alaska, Fairbanks, AK, USA
University Carlos III de Madrid, Madrid, Spain
ITER Organization, Saint Paul Lez Durance, France

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. Very 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. As a result, 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 \( \frac{1}{K^2} \), 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. In addition, the framework used for this implementation executes a task when the data dependencies are satisfied and computational resources are available. This leads to improved computational efficiency over previous approaches that pipeline or schedule groups of tasks to a particular processor or group of processors.

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 presently limited to a very small fraction of that time for realistic physics. Advances in algorithms and use of ever-larger high
Parareal is an algorithm for effectively utilizing high performance parallel computers to, in effect, decompose the time domain to obtain the numerical solution for 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 domain 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 \( \lambda_{i-1} \) to time \( t = t_i \) and state \( \lambda_i \). It is described by notation \( \lambda_i \equiv F_G(\lambda_{i-1}) \) 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 \( \lambda_N = F_{FN}(\lambda_0) \) where the initial conditions are given by \( \lambda_0 \).

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 Ref. [5], the effectiveness of \( G \) is, in practice, determined by testing. As with \( F \), the notation \( \lambda_i \equiv G_M(\lambda_{i-1}) \) is used to describe \( G \). The wall–clock times for executing a step of \( \Delta t \) for the coarse and fine solvers are given by \( T_C \) and \( T_F \), respectively, with the ratio denoted by \( \beta = T_F/T_C \). The notation \( \lambda_{k,i}^{GF} \) will be employed to distinguish between states for the coarse/fine solvers (\( G/F \)), for iteration \( k \) 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 \( \lambda_{0,i}^G \) and \( \lambda_{0,i}^F \); 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 \( \lambda_{k,i+1}^F = F_M(\lambda_{k,i}^G) \), 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 \( (k) \), present time slice \( (i) \) is \( \lambda_{k,i-1} = \lambda_{k,i-1}^G + \lambda_{k,i-1}^F \). The notation \( \lambda \) is used to distinguish between a state that is the result of a propagator, \( G(\lambda) \) or \( F(\lambda) \), and the state \( \lambda \) that is the result of the linear combination of states that constitutes the parareal iterate. If needed, operators to transform states for one solver space to the other are implied. This update only depends on fine results from the previous iteration, thus allowing all of 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 a given state update does not require that all of the coarse or fine steps for that iteration be completed. Thus, a given fine step \( \lambda \) can be started as soon as the \( (i-1) \) parareal update (using results from the \( (i-1) \) coarse step) is completed. 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. This opportunity for improved performance was recognized in previous work where tasks were pipelined [9], or scheduled [10], to take advantage of this overlap. The present work extends these ideas by implementing parareal with a distributed, data-driven, event-based framework that decouples any a priori assignment of tasks for a given time-slice to a processor or group of processors or nodes (referred to processors for conciseness). This allows minimizing idle computational resources. For example, it is not necessary to request the same number of processors as time slices. The approach we employ file-based data communications between components. For some applications, this could impose a significant performance penalty. However, for a large class of applications on high performance computing systems, this overhead does not result in a significant performance penalty.

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 Ref. [8]. A summary is presented in Section 5. Both the sequential and event-based parareal algorithms were implemented using a lightweight Python framework, the Integrated Plasma Simulator (IPS) that was developed for multiphysics simulations of magnetically confined fusion plasmas [11–13].

2. Sequential parareal

The work flow for sequential parareal implementation follows from the algorithm description presented in the previous section. However, the details are slightly different for the first, second, and subsequent iterations. These details are described in the pseudo code of Fig. 1 in order to provide background for the event-based work flow. For the first iteration, lines 4–7, the coarse propagator is applied sequentially to the initial state \( \lambda_0^G \). The total execution time for this step is given by \( N \Delta t_C \). The states \( \lambda_0^F \) are then, in parallel, used as initial conditions in lines 9–15 to compute a corresponding set of fine states. If needed, the use of a transformation to elevate the coarse states to the fine description is implied. The wall–clock time is given by \( T_F \) giving a total execution time for the first iteration of \( N \Delta t_C + T_F \). When parareal is implemented in a single executable, a specific time slice is typically assigned to a designated processor.
The basic execution pattern is maintained for subsequent iterations with two main changes. We determine the first time slice to be processed based on convergence tests. For some implementations, convergence tests are only used to determine the completion of the algorithm, and the first time slice is simply incremented by one for each subsequent iteration. The second change involves the use of the parareal update as input to the coarse and fine propagators for the second time slice of each iteration. The total time for the second iteration is again \( \frac{\text{NTG}}{C} + \text{TF} \). For \( k < N \), the total time is again \( \frac{\text{NTG}}{C} + \text{TF} \). Even when the condition \( k < N \) is not met, the use of the parareal algorithm was successfully applied to a model plasma turbulence using a single executable with internal management of the MPI calls. For the present effort, we used the fine, coarse, converge, coarse-to-fine, fine-to-coarse, and parareal-advance methods (six independent executables) from that effort to recreate the algorithm within the IPS simulation framework. This led to exactly the same computational operations, but within a framework that could facilitate applications to new problems.

1 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 only be applied 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.

---

**Fig. 1.** Pseudo code for sequential parareal.

```c
1 // First iteration
2 // Sequential propagation of coarse solution through all slices
3 \( \lambda^C_{\Delta} = G_m(\lambda^C_{\Delta-1}) \)
4 for \( i = 1, \ldots, N-1 \) do
5 \( \lambda^C_{i-1} = G_m(\lambda^C_i) \)
6 end
7 // Fine solves executed in parallel
8 for \( i \in \{0, N-1\} \) do in parallel
9 if \( i == 0 \) then
10 \( \lambda^F_{0} = F_m(\lambda^F_{0}) \)
11 else
12 \( \lambda^F_{i+1} = F_m(\lambda^F_{i}) \)
13 end
14 end
15 // Subsequent iterations
16 for \( k = 2, \ldots, N \) do
17 // Find first unconverted slice
18 \( n = \max_{\{k,N\}} | \text{converge}(j) = \text{True} \ \forall j < m \)
19 if \( n == N \) then
20 break
21 end
22 // Sequential propagation of coarse solution through all slices
23 \( \lambda^C_{\Delta-1} = G_m(\lambda^C_{\Delta-1}) \)
24 for \( i = n+1, \ldots, N-1 \) do
25 \( \lambda^C_{i-1} = \lambda^C_{i-1} - \lambda^C_{i-1} + \lambda^F_{i-1} \)
26 \( \lambda^C_{i-1} = G_m(\lambda^C_{i-1}) \)
27 end
28 // Fine solves executed in parallel
29 for \( i \in \{n,N-1\} \) do in parallel
30 if \( i == n \) then
31 \( \lambda^F_{n} = F_m(\lambda^F_{n-1}) \)
32 else
33 \( \lambda^F_{i} = F_m(\lambda^F_{i-1}) \)
34 end
35 end
36 end
```
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 constructed coarse, fine, and converge components. In addition, a driver component was used to control the sequence of component operations, calling the coarse, fine, and converge components for each iteration in an outer ‘iteration’ loop. These components are, at the simplest level, Python wrappers for the executables that carry out the operations described in the previous section. The use of independent executables leads to an overhead for file-based intra-component communications and an operating system overhead for starting and shutting down multiprocessor jobs.

The Python components developed for this application can serve as templates for new applications with only relatively small changes to the scripts. As a result, the additional programming for each new application at the parallelization level, e.g., MPI communicator management, is required to integrate the coarse, fine, and converge tasks within one executable is not necessary. For a given application, the IPS implementation also facilitates the development of an acceptable coarse solver.

The convergence history for one run with this new implantation is shown in Fig. 2. The total time for the simulation was $N \Delta T = 12800 \tau_{di}$ where $\tau_{di}$ 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 1024 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 resource management capability of the IPS allowed this run to be completed with fewer nodes than time slices. While increasing the wall clock time for early iterations where the number of active slices was greater than 128, computational efficiency was improved for later iterations because of the reduction in idle nodes. The overall effect on wall clock time was minimized because, for this highly non-linear problem, the execution time varied by as much as a factor of two for different time slices, and waiting fine tasks could begin executing while the slower fine tasks were still running. The VODPK [14] adaptive integrator was used in the fine solver while 4th order Runge-Kutta was used in the coarse solver. Convergence was attained in 14 iterations using a tolerance of $1.5e-6 (\Delta E/E)$ on the time-integrated energy for each slice. Additional detail can be found in Ref. [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 $T_C < T_F$, the gain $H$ is given by the ratio of the time for a full solution with the fine solver applied sequentially, $N T_F$, to the parareal wall-clock time $K T_F$ or $H = N/K$. Finite $T_C$ will modify the time per iteration from $T_F$ to $T_F + N T_G$ and decrease the gain to

$$H = \frac{N}{K(1 + \frac{N}{\beta})},$$

(1)

where, again, $\beta = T_F/T_C$.

Fig. 2. Convergence history for a 160-slice BETA simulation using parareal. No values are shown for the first iteration as there is no convergence data from the first iteration.
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 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 $N/\beta$ 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 Fig. 3. The processor utilization during the ~14000 second sequential run of Fig. 3 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, in large part, the condition $N/\beta \ll 1$ was not met. Meeting this condition and also obtaining convergence in a small number of steps is the central issue for the successful 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 logical in a sequential programming sense, 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. This potential for improving the parareal algorithm was recognized and implemented in Refs. [9,10]. The advantages of using the IPS to implement the sequential IPS also apply to this improved algorithm. In addition, the event handling and resource management capability of the IPS also provide the possibility of additional improvements. In particular, they allow idle resources to execute a task for which required data are available independent of iteration or time slice. This is not possible for implementations that are deterministic in the sense that coarse and fine tasks for a given time slice are a priori assigned to a particular processor. Such idle resources result, as examples, from some fine tasks finishing in a much shorter time than others or time slices that have converged, thus freeing the assigned processor. In addition, the IPS resource management capability allows fewer processors to be requested for a job than time slices. For the case when $NT_C \geq T_f$, data dependences reduce the number of fine tasks that can execute in parallel, and as a result there may be impact on wall-clock time. The optimum resource level depends on $NT_C$ compared to $T_f$ and on the variability of $T_f$. As with the sequential IPS implementation of parareal, the algorithm and the data employed are exactly the same, but task execution is started just as soon as the needed data are available.
The first element to restructuring the parareal algorithm is the realization that each fine and coarse solver step can be thought of as an independent task that can proceed as soon as the needed data are available. This allows overlapping coarse and fine tasks of a given iteration as well as coarse and/or fine tasks for different iterations. The only requirement is that the data dependencies are satisfied and that resources are available. Scheduling, Ref. [10], and pipelining Ref. [9], implementations of parareal have taken advantage of this restructuring within the tasks for a given iteration and were able to realize most of the potential gains in wall-clock reductions. However, because tasks for a time slice were assigned to a specific processor, they did not exploit the potential for task scheduling across iterations and did not realize the potential for improvements in computational efficiency that could be realized. The second element to the restructuring is the elimination of an independent control element—the driver or main routine. Instead, a distributed control model is used where the logic is split across coarse, fine, and converge components. Each component maintains a collection of tasks that can be executed once their external and internal dependencies are satisfied. The asynchronous event services of the IPS are used to signal the availability and path for the data. Because of this asynchronous, decentralized control, it is, in general, not possible to predict when or on what resource a task will be executed.

Fig. 4 indicates the four phases of the dependency driven parareal for the coarse component. While there are differences in detail, the structure for the other two components is similar. In the initialization phase (lines 3–8), the component subscribes to event topics where the other two components publish their updates. In addition, the first coarse task is initialized with its only external dependency, the initial coarse state. The task is then placed in a FIFO queue for subsequent execution. In the task dispatch phase (lines 12–18), queued tasks that are ready to run are submitted for execution by the IPS framework. These tasks are standalone MPI tasks that are run using the platform's native execution command, e.g., mpiexec. Tasks are submitted until the component runs out of ready tasks or until tasks fail due to the unavailability of resources. In the latter case, the task is re-queued for future execution.

```plaintext
// Phase 1: Initialization
subscribe(FINE EVENTS)
subscribe(CONVERGE EVENTS)
taskQueue = Queue()
task1 = newTask()
satisfyDependency(task1, A)
addTask(taskQueue, task1)

while not done do
  // Phase 2: Task dispatch
  for task ∈ taskQueue do
    if launchTask(task) == success then
      continue
    else
      break
  end

  // Phase 3: Event Processing
  events = pollTopics()
  for e ∈ events do
    if e == ALL CONVERGE then
      done = TRUE
      break
    else if e == TASK FINISHED then
      data = e(task output)
      for task ∈ dependent tasks do
        satisfyDependency(task, data)
        if readyToRun(task) then
          addTask(taskQueue, task)
      end
    end

  // Phase 4: Completed task processing
  finishedTasks = checkTasks(runningTasks)
  for task ∈ finishedTasks do
    publish(COARSE EVENTS, task)
    for deptask ∈ dependent tasks do
      satisfyDependency(deptask, data)
      if readyToRun(deptask) then
        addTask(taskQueue, deptask)
    end
  end
end
```

Fig. 4. Coarse component pseudo code for the data-driven parareal.
The component then proceeds to the event processing phase of the algorithm (lines 21–33) where events (if any) are processed. When all dependencies are satisfied, as reported by ReadyToRun(), the task is placed in the ready queue. The final phase (lines 36–46) involves the processing of tasks that have been submitted during the task dispatch phase and that have finished execution. The output from such tasks is used to update the dependencies of other local tasks. Events from these tasks are published for other components that signal task completion. These events contain a 'payload' of information that is needed for the execution of dependent tasks.

The performance of the data-driven parareal can be estimated using the same approach as was used for the sequential approach. This estimate assumes that system and communication overheads are negligible and is sufficiently accurate to understand the scaling characteristics of the implementation. We assume that a simulation length of \( N \Delta T \) converges in \( K \) iterations (with \( K \) always \( \leq N \)) and a convergence history in which there are no converged slices until the last iteration. For this case, the data dependencies lead to the task sequence \( F_{1,1} \Rightarrow F_{2,2} \Rightarrow F_{3,3} \Rightarrow \ldots F_{K-1,K-1} \) followed by \( G_{k,k} \Rightarrow G_{k,k+1} \Rightarrow \ldots \Rightarrow G_{N-1,N-1} \Rightarrow G_{K,N} \) for a total time for the \( K \) iterations of \( K \Delta T + (N-K)T_C \). Graphically, the data dependencies allow tasks to be executed along the diagonal time-slice index = iteration index line until convergence and then across the time-slice dimension. The resulting gain is

\[
H = \frac{N}{K + \frac{N-K}{\Delta T}}.
\]  

This estimate differs from that presented in Ref. [9], \((N-K)\) compared to \((N-K+1)\), because we used a slightly different algorithm for convergence estimation.

The event-based execution flow was implemented by using IPS event-serves capability, and applied to the same case as described in Section II. The convergence history for the new implementation was exactly the same as shown in Fig. 2 for the original parareal work flow, as were the detailed numerical results. However, as presented in Fig. 5, the total wall clock simulation time decreased by over 50% and the processor utilization increased by over 50%. The initial ramp to near 100% utilization was set by the time required for all fine tasks to start, \( NT_C \), while the fall off was caused by lack of fine tasks to execute in parallel as the simulation converged.

**4. Analysis of data-driven parareal**

In order to understand the performance characteristics of data-driven parareal with variations in the coarse solver, two studies were conducted. The first used a test simulation case for BETA using the 4th order Runge-Kutta coarse solver. The second used the empirical convergence model that was developed in Ref. [8]. The test case, with \( N = 32 \) and \( \Delta T = 80 \tau_{gs} \), converged in seven iterations. For the first study, gain properties were varied by artificially increasing the coarse solver time, \( T_C \).
by a “sleep” command. These results are summarized in Fig. 6 as a function of \( \beta = T_f/T_c \) with \( T_f \) and \( T_c \) designating the average of fine and coarse tasks, respectively, in the simulation. Beta varied from less than five \( (\beta << N) \) to almost a hundred \( (\beta > N) \), encompassing the range of interest for the gain models described in Section 2. The measured simulation speed up times were based on the ratio of \( N T_f \) to the simulation wall clock time. The prediction of both the event-based (Eq. (2), \( H = N(K + (N-K)/\beta) \)) and sequential (Eq. (1), \( H = N/K (1 + N/\beta) \)) models are also shown in Fig. 6. The measured simulation times for the event-based implementation show gains of over two even when the sequential model values were less than one. For large \( \beta \), as suggested by the modeling, the event-based and sequential models are converging to the same gain, \( \sim 4.6 \sim N/K = 4.6 \), and are within ~30% of the measured values. The \( \beta \) 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 \( \beta \)s is likely due to two factors. First, peak values of \( T_f \) for each iteration are typically a factor of two or more larger than the average because the VODPK integrator adapts time steps to achieve a given precision. For both sequential and event-based implementations, peak values of \( T_f \) have a larger impact on the gain than 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 that particular iteration and slice is a critical dependency. The drops in processor utilization during the first two thirds of the simulation in Fig. 5 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 two, 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 Ref. [8] for the convergence characteristics of coarse solvers based on the VODPK and 2nd and 4th 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 to a time \( t_1 \) followed by a faster rate of convergence, \( b \) slices per iterations, at time \( t_2 \) with a linear ramp for times between \( t_1 \) and \( t_2 \). Estimates for these 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 Ref. [8].

The model values for \( K(N, \Delta T) \) for the three coarse solvers were used in the approximate gain expressions that were presented in Eqs. (1) and (2). Two types of scaling for a particular coarse solver were analyzed: weak scaling where \( \Delta T \) 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 \( N \).

Fig. 7 presents the results of the strong scaling model for both the sequential and event-based implementations. Fig. 8 presents 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 (4th order RK) to seven (VODPK) times better. More importantly, the improvement in performance is greatest for the poorest performing solver for the sequential implementation. Thus the coarse solver that would previously have been judged as not useful (VODPK), now results in gains that are greater than were found with the best coarse sequential coarse solver (4th order RK). The same behaviors are observed for

![Fig. 6. Comparison of measured speed ups 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 \( (N/K = 32/?) \) for this test problem for the BETA model using the RK4 coarse solver. The simulations had \( N = 32 \) and \( \Delta T = 80t_a \) and converged in seven iterations.](image-url)
the weak scaling analysis. In addition, the optimum in gain previously observed when the number of slices is \( \sim \beta \) is no longer observed.

The above discussion has focused on estimating wall-clock improvements. Computational efficiency (fraction of busy resources) improvements also result when the execution of tasks for a time slice is not locked to a resource. When, for example, the work (one coarse task and one fine task) for the first time slice is completed, this resource is no longer available for use if locked to first-slice tasks. For the case when \( NTG > TF \) and the first iteration, tasks for slices with \( i = TF/TG \) do not start until after work for \( i = 1 \) has completed. This same potential exists for later iterations and increases as slices converge. We have not developed quantitative expression for the improvements in efficiency, but have observed that, depending on the case, requesting fewer resources (nodes/processors) than would be required by the one time slice per resource prescription does not impact overall execution time. While not included in the speed up analysis, this capability also allows tasks for a slice-iteration pair to run in parallel. The improvements in total execution time for this capability is relatively small, \( = KTG \).

5. Summary

Sequential and event-based parareal algorithms have been implemented using the IPS framework. The even-based implementation builds on the concept of task overlap that was developed in previous efforts, but provides additional performance improvements in reduced wall clock time and in increased computational efficiency by extending the overlap possibilities across both time and iteration tasks. The use of independent executables and file-based data does result in overheads.
However, for many applications, especially those that target large high performance computing systems, these overheads do not significantly impact overall performance.

For this project, the target application was BETA, a model plasma turbulence application. However, the Python-based IPS provides a general parareal framework that allows applying parareal to new applications with a minimum of application code modifications. Similarly, testing new coarse solvers or using different convergence evaluation techniques requires less code modification. 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 these improvements is recognizing the data dependencies of parareal and beginning work on any task for which the data and resources are available. As a result, requirements on the coarse solver are reduced from $\beta \geq N$ to $\beta \geq N/K$.

While not examined in the present effort, the IPS event driven implementation of parareal could likely be modified to accommodate any of the hybrid parareal algorithms that have been developed including, for example the spectral deferred corrections method of Ref. [9].

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.

References