A Dependency-Driven Formulation of Parareal: Parallel-in-Time Solution of PDEs as a Many-Task Application

Wael R. Elwasif, Samantha S. Foley, David E. Bernholdt, Lee A. Berry
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831 USA
{elwasifwr, foleyss, bernholdtde, berryla}@ornl.gov

Debasmita Samaddar
ITER Organization
13115, St. Paul-lez-Durance
France
debasmita.samaddar@iter.org

David E. Newman
University of Alaska
P.O. Box 755920
Fairbanks, AK 99775 USA
denewman@alaska.edu

Raul Sanchez
Universidad Carlos III de Madrid
28911, Madrid, Spain
rsanchez@fis.uc3m.es

ABSTRACT

Parareal is a novel algorithm that allows the solution of time-dependent systems of differential or partial differential equations (PDE) to be parallelized in the temporal domain. Parareal-based implementations of PDE problems can take advantage of this parallelism to significantly reduce the time to solution for a simulation (though at an increased total cost) while making effective use of the much larger processor counts available on current high-end systems. In this paper, we present a dynamic, dependency-driven version of the parareal algorithm which breaks the final sequential bottleneck remaining in the original formulation, making it amenable to a "many-task" treatment. We further improve the cost and execution time of the algorithm by introducing a moving window for time slices, which avoids the execution of tasks which contribute little to the final global solution. We describe how this approach has been realized in the Integrated Plasma Simulator (IPS), a framework for coupled multiphysics simulations, and examine the tradeoffs among time-to-solution, total cost, and resource utilization efficiency as a function of the compute resources applied to the problem.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.11 [**Software Engineering**]: Software Architectures; I.6.3 [**Simulation and Modelling**]: Applications; J.2 [**Computer Applications**]: Physical Sciences And Engineering—*Physics*

General Terms

Algorithms, Design, Performance

Keywords

Coupled Simulation, Framework, Algorithm, Concurrency

MTAGS2011 '2011 Seattle, Washington, USA

1. INTRODUCTION

In partial differential equation (PDE) based problems, which include a wide range of computational science problems of current interest, time is one aspect of the problem that has long been considered to be strictly sequential by most practitioners. Although parallelization in time has been a niche research topic in mathematics since the earliest days of computing, it was not until the 2001 publication of the *parareal* algorithm by Lions, Maday and Turinici [3, 14] that parallel-in-time approaches reached broader awareness. Today, as the means of scaling computational capability has shifted from increasing processor clock speeds to increasing processor (core) counts, the prospect of exposing an additional dimension of parallelism in PDE problems is attracting growing attention from computational scientists in a variety of disciplines (for example, [3, 9, 13, 22]).

Briefly, the "classic" parareal algorithm (explained in greater detail below) utilizes a "coarse solver" to quickly step through time to compute relatively cheap approximate solutions for all time slices of interest, and then refines all of these approximate solutions simultaneously using an accurate "fine solver". The application of the fine solver to each time slice is independent, and thus parallelizable. The refined solutions are then fed back through the coarse solver, and the iterative cycle continues until all time slices are converged. However, the unavoidable dependencies between time slices are carried through the coarse solve, which, in

the classic parareal formulation, still involves stepping sequentially through time. This represents a sequential bottleneck, albeit a much smaller one, assuming the coarse solver is much faster than the fine solver. At the heart of a successful parareal treatment of a given problem, then, is the ability to find a coarse solver which provides sufficient accuracy to rapidly converge the time slices, while remaining fast enough to minimize the time spent in this sequential operation.

However, by looking at the parareal algorithm from a different perspective, it is possible to reformulate the problem in a form, which, for practical purposes, breaks the sequential bottleneck and allows much more effective parallelization of the algorithm. The insight that gave rise to this innovation, some ten years after the parareal algorithm was first introduced, comes from considering parareal through the lens of many-task computing. The term "many-task computing" (MTC), first coined in 2008 [21], refers to the middle ground between traditional high-performance computing, in which major computational resources are brought to bear on a single problem, and high-throughput computing, which focuses on using large-scale computing to solve many computational tasks in a short time period.

More specifically, this innovation was inspired by the capabilities of the Integrated Plasma Simulator (IPS) [7, 11], a framework developed since 2005 for loosely coupled integrated simulation of fusion plasmas. The IPS was designed as an extremely flexible environment in order to support simulations with a wide variety of control structures, and provides, among other features, multiple levels of concurrency [8], and dynamic task and resource management, which have been used to allow the framework to execute multiple independent simulations concurrently [10] in a many-task context. Add to this the framework's event service [23], which allows components to communicate asynchronously, and it becomes possible for the IPS to easily represent dependencies among components in a non-procedural form, which was the key insight into a dependency-driven formulation of the parareal algorithm. Thus, the parareal algorithm maps neatly onto the one of the motivating MTC examples of a loosely-coupled application involving many tasks with limited dependencies among them which must be respected in their execution.

This paper makes three primary contributions, which are noted here in the context of the paper's organization. After discussing related work (Sec. 2), briefly describing the IPS (Sec. 3), and introducing the classic formulation of the parareal (Sec. 4),

- We present the dependency-driven parareal formulation, and explain its implementation in the IPS (Sec. 5).
- We offer a "moving window" variant of the parareal algorithm, which reduces that total cost of a parareal-based simulation by avoiding largely unproductive computation on time slices that are far from convergence (Sec. 6).
- Using a plasma turbulence problem, we illustrate the performance and trade-offs involved in the parareal approach, comparing the classic, dependency-driven, and windowed versions (Sec. 7).

We close with a summary of the key points of the paper (Sec. 8).

2. RELATED WORK

The idea of parallelizing the time domain has been explored since early in the computer age [12], of which parareal [14] is one of the most recent and at present the most widely studied.

Research related to parareal has primarily focused on the mathematical properties as it has been applied to problems in a variety of different domains, including molecular dynamics [3], fluid dynamics, [9], plasmas [22], reservoir simulation [13]. Notably, the idea of combining parareal's temporal domain decomposition with the spatial domain decomposition approach commonly used to parallelize PDE problems, and a central feature of our implementation, was first proposed by Maday and Turinici in 2005 [16].

However, very little work has been done from the algorithmic perspective. A recent paper by Aubanel [2] proposes several new ways of scheduling tasks with the goal of improving speedup. Compared to the classic parareal implementation, Aubanel's "master-worker" algorithm increases the overlap between coarse solves (on the master process) and fine by launching fine solves on worker processes as soon as their individual coarse dependencies are satisfied, rather than waiting until the entire coarse solve completes. His "distributed" algorithm assigns each slice (both the coarse and fine solves) to a different parallel process, sequencing them via the MPI send/receive operations that transmit results between slices. Minion uses the term "pipelined" to describe this same algorithm [18].

Our work, carried out contemporaneously with Aubanel's and Minion's, has a similar goal, but takes a very different approach. Whereas their algorithms remain very much loop driven, we have used a completely dependency-driven approach, without any loop structure whatsoever. The result is similar to the distributed (or pipelined) algorithm, but with even greater flexibility in terms of what tasks can run at any given moment, and thus the potential for even better resource utilization and speedup.

Additionally, to the best of our knowledge, ours is the first implementation of parareal which eschews two-sided message passing (MPI) as the basis for the high-level orchestration of tasks. While MPI is widely used in high-performance computing, it suffers two drawbacks in a context like this. First, unless they have been designed for it, it can be hard to integrate MPI-based parallel components into a framework which uses MPI to provide a second level of parallelism. Second, two-sided messaging is not the most natural way to express the parareal algorithm, typically leading to implementations that impose more synchronization than is necessary. Approaching the problem from a many-task perspective has helped us avoid these issues.

The concept of many-task computing spans a broad range of target problems and computational capabilities necessary to support them. While our work has been carried out using the IPS framework's many-task computing environment, the same approach could, at least in principle, be implemented in any many-task environment with the capability to express dependencies among tasks. Relying on literature reports, it appears that many of the frameworks previously discussed in a many-task context could express a dependency-driven formulation of parareal, including Uintah [17], Kepler [15], Taverna [20], Triana [4], GXP Make [25], Swift [27], Pegasus [6], Nephele [26], and PLASMA [24] however such publications rarely provide sufficient detail to determine whether,

and how effectively, the system can handle the dynamic nature of the dependency-driven parareal, including convergence rates that can vary significantly by slice, or the more complex "windowed" variant of the algorithm described in Sec. 6. Additionally, it must be noted that these frameworks provide a wide range of execution environments. Depending on the framework, tasks may be batch queue submissions or grid jobs, local, or even tightly integrated into a single executable. In the IPS, tasks essentially amount to the launch of distinct executables, and the entire parareal simulation takes place within a single batch submission on a single system.

3. INTEGRATED PLASMA SIMULATOR

The Integrated Plasma Simulator (IPS) has been developed by the Center for Simulation of RF Wave Interactions with Magnetohydrodynamics (SWIM) [1], an effort devoted to improving the understanding of the interactions of radio frequency (RF) waves and the extended magnetohydrodynamic (MHD) phenomena in fusion plasmas. However the IPS itself has been designed to support a broad range of integrated modeling of fusion plasmas as well as domains outside of plasma physics with similar characteristics.

The IPS, which has been described in greater detail elsewhere [7, 11], was designed for time-stepped simulation with relatively loose explicit coupling among the components. Components in the IPS are generally wrappers around standalone executables for physics codes which adapt their native data and control formats to those used by the IPS, allowing the executable (task) to remain unchanged yet interact with the framework and other components. This has allowed the SWIM project to take advantage of the many decades of effort the community has put into the development, verification, and validation of modeling tools for the individual physics and focus on the new issues associated with coupling them. In general, the data exchanged by components is modest, and is easily managed through files. The IPS is designed primarily for use in a batch processing environment, with a batch job typically comprising a single invocation of the framework, calling the individual physics codes underlying the components many times as the simulation progresses. IPS simulations are typically orchestrated by a "driver" component, though simulation-controlling logic can also be built into individual components, as in the dependency-driven parareal implementation described here. The IPS also provides both checkpoint/restart coordination, and task re-execution capabilities for fault tolerance. The task re-execution capability can be controlled on a per-component basis, with decisions based on the nature of the component, as well as the fault experienced [23].

The capabilities of the IPS that play a key role in supporting the novel parareal formulation presented here were all originally put into the IPS for other reasons. This work merely presents a novel way to use them.

Existing modeling codes in the plasma physics community span a wide range of parallelism. Some codes remain strictly sequential, while others are quite scalable. Many are in between. In order to allow better overall utilization of resources on a parallel computer, the IPS supports multiple levels of concurrency, providing the flexibility for IPS users to maximize the number of compute nodes in use at any point in the simulation [8,10]. Individual computational tasks (the physics executables underlying IPS components) can be parallel, components can launch multiple computational tasks simultaneously, multiple tasks can be executed concurrently in a simulation, and multiple simulations can also be run simultaneously. The dependency-driven parareal utilizes all but the last form of parallelism.

In order to support multiple levels of parallelism, the IPS provides flexible resource and task management. When the IPS is started, it detects the computational resources which have been allocated to it, and instantiates an internal resource manager. This resource manager tracks their use throughout the run, responding to allocation requests from the task manager and releasing allocations when tasks complete, using a simple first-come first-served algorithm with first-fit backfill. The IPS task manager provides non-blocking task launch capabilities which allow the concurrency described. The framework uses the appropriate mechanism for the host system, such as `mpiexec` or `aprun` (Cray) to launch the individual computational tasks within the resources allocated to the job. IPS tasks generally exchange data via files, as there is no MPI connectivity between distinct tasks.

Finally, the IPS event service provides a simple publish–subscribe mechanism, allowing components to asynchronously post and receive user-defined messages. The event service was originally introduced in order disseminate fault-related information within the framework and components in order to allow for more informed responses to system faults [23]. It was also used to make information about the progress of the simulation available through an external web-based portal. The first use of the event service in *controlling* simulations has been in SWIM's work on the use of radio-frequency (RF) waves to control plasma stability, in which a continuously running component uses the event service to signal another to update a key simulation quantity, and then incorporates the new data on the next iteration after it receives a signal that the update is complete [11]. While the dependencies in the parareal algorithm are significantly more complex, the concept is fundamentally the same.

4. PARAREAL ALGORITHM

Consider a time-dependent system with initial state λ_0 at time t_0. Parareal utilizes a fine solver, F, that, over time domains of interest (slices), 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^F_{i-1} to time $t = t_i$ and state λ^F_i. It is described by notation $\lambda^F_i = F_{\Delta t}(\lambda^F_{i-1})$ 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 $\lambda^F_N = F_{N\Delta t}(\lambda^F_0)$ where the initial conditions are given by $\lambda^F_0 = \lambda_0$.

The second element of parareal is a coarse solver, G. Since it is the coarse solver in parareal that embodies the remaining time dependence between the different time slices, it must balance speed and accuracy. It must be sufficiently accurate to ensure rapid convergence of the fine solves, while being fast enough to minimize the time spent in the sequential time propagation. 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 [14], the effectiveness of G is, in practice, determined by testing. As with F, the

coarse solver implements a state advance operator $\lambda_i^G = G_{\Delta t}(\lambda_{i-1}^G)$.

The notation $\lambda_{k,i}^{G/F}$ will be employed to distinguish between states for the coarse/fine solvers G/F, for iteration k at the end of time slice i. Whenever states are used as arguments or in an operator statement, use of the appropriate transformation, for example $\lambda_{k,i}^G \Longrightarrow \lambda_{k,i}^F$ 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. Additionally, the parareal algorithm requires a method for evaluating convergence and initial states $\lambda_{1,0}^G$ and $\lambda_{1,0}^F$.

The iterative state update (sometimes also called state correction) is the defining element of parareal. The update for the state for the present iteration k, present time slice i is $\Lambda_{k,i} = \lambda_{k,i}^G - \lambda_{k-1,i}^G + \lambda_{k-1,i}^F$. The notation Λ is used to distinguish between a state that is the result of a propagator, G or F, and the state Λ that is the result of the linear combination of states that constitutes the parareal iterate. It should be noted that $\Lambda_{k,i}$ can be considered as a correction applied to the state $\lambda_{k,i}^G$ computed by the coarse propagator G (for iteration $k > 1$) before being used by both the coarse and fine propagator in time slice $i + 1$.

The basic outline for the parareal algorithm is shown in Fig. 1. In the first iteration of the algorithm, the coarse propagator G is applied sequentially to the initial state λ_0 to compute system state $\lambda_i^G \forall i \in [1, N]$ (lines 4–7). This set of states is then used to compute (in parallel) a corresponding set of fine states $\lambda_i^F \forall i \in [1, N]$ (lines 9–15). Note the special treatment of the first time slice where the given initial state λ_0 is used by both the fine and coarse propagators.

This basic execution pattern is maintained in subsequent iterations, with two main changes. The first time slice to be processed is determined based on the results of convergence testing of all slices. The first slice to be processed in iteration $k > 1$, slice n, is the *first* slice that fails the convergence test (line 21). The convergence test can involve comparing the state computed by the fine propagator in two consecutive iterations ($\lambda_{k-1,i}^F$ and $\lambda_{k-2,i}^F$), which means that testing for convergence can only start in the third iteration. Other forms of convergence testing can also be used (e.g. comparing the results computed by the coarse and fine solvers for the same time slice). The other main change in the algorithm involves the use of updated (or corrected) coarse state (line 27 in Fig. 1) as input to subsequent coarse and fine propagators.

It should be noted that mathematically, *at least* one slice will converge per iteration in the parareal algorithm. This stems from the fact that the first slice is processed by the "accurate" fine solver, using either the overall initial state λ_0, or the converged accurate solution computed in the previous iteration. As such, the algorithm is guaranteed to converge in *at most* $K = N$ iterations, though in practice it is possible to achieve much faster conversion with judicious choice of the coarse propagator.

Direct parallel implementation of the parareal algorithm results in repeated cycles of low resource utilization (when the coarse propagator is sequentially applied), followed by high utilization where the fine propagator is applied in parallel to all time slices (e.g., the "classic" (red) curve in Fig. 4(b)). This pattern necessitates the choice of coarse propagators whose execution time T_G is significantly less than that of the fine propagator T_F. Based on the particular problem at hand, ratio T_F/T_G of 100 or more is needed for the algorithm

```
 1 // First iteration
 2
 3 // Sequential propagation of coarse solution
       through all slices
 4 λ^G_{1,1} = G_Δt(λ_0)
 5 for i = 1, .., N − 1 do
 6     λ^G_{1,i+1} = G_Δt(λ^G_{1,i})
 7 end
 8 // Fine solves executed in parallel
 9 for i ∈ [0, N − 1] do in parallel
10     if i == 0 then
11         λ^F_{1,i+1} = F_Δt(λ_0)
12     else
13         λ^F_{1,i+1} = F_Δt(λ^G_{1,i})
14     end
15 end
16
17 // Subsequent iterations
18
19 for k = 2, .., N do
20     // Find first unconverged slice
21     n = max_{m∈[k,N]} | converge(j) = True ∀j < m
22     if n == N then
23         break
24     // Sequential propagation of coarse solution
           through all slices
25     λ^G_{k,n} = G_Δt(λ^F_{k−1,n−1})
26     for i = n + 1, .., N − 1 do
27         Λ^G_{k,i−1} = λ^G_{k,i−1} − λ^G_{k−1,i−1} + λ^F_{k−1,i−1}
28         λ^G_{k,i} = G_Δt(Λ^G_{k,i−1})
29     end
30     // Fine solves executed in parallel
31     for i ∈ [n, N − 1] do in parallel
32         if i == n then
33             λ^F_{k,i} = F_Δt(λ^F_{k−1,i−1})
34         else
35             λ^F_{k,i} = F_Δt(Λ^G_{k,i−1})
36         end
37     end
38 end
```

Figure 1: The parareal algorithm.

to achieve adequate wall clock speedup. The need to find coarse solvers that are both fast enough to avoid excessive waste of resources during the sequential portions of the algorithm, while having enough accuracy to enable convergence in $K << N$ iterations, where K is the number of iterations it takes to converge the N slices, has been the main hurdle towards more widespread use of the parareal algorithm.

In the following sections, we present two modifications to the parareal algorithm that recast the computation as a many-task problem, allowing for increased efficiency in resource utilization and reduced wall-clock execution time, opening the door towards exploring the use of slower (and usually more accurate) coarse solvers.

5. DEPENDENCY-DRIVEN PAREAREAL

The classic parareal algorithm presented in Sec. 4 achieves faster wall clock solution time (relative to the standard solution using successive application of the "accurate" fine solver),

provided that a coarse solver that is both fast and sufficiently accurate can be found for the problem at hand. Historically, parareal has been implemented using an MPI-based multiple-program multiple-data (MPMD) approach that links all elements of the algorithm (coarse solver, fine solver, convergence testing, and control flow) into a single executable. This implementation methodology makes it difficult to experiment with different coarse solvers. Furthermore, direct implementation of the algorithm makes poor use of available computational resources, as a significant fraction of the time is spent in the sequential coarse propagation phase of the algorithm, during which most parallel resources go idle.

The key to restructuring the parareal algorithm is the realization that each application of the coarse and fine solvers to any given time slice can be thought of as an *independent task*, that can proceed as soon as the inputs on which the task depends are available. For example, in the first iteration, the two operations $\lambda^G_{1,1} = G_{\Delta T}(\lambda_0)$ and $\lambda^F_{1,1} = F_{\Delta T}(\lambda_0)$ can proceed concurrently, since both depend *only* on initial condition λ_0. Furthermore, using the same arguments, it is possible to overlap the execution of fine and coarse solver tasks from different iterations, provided their input dependencies are satisfied.

Re-casting the parareal algorithm as a dependency-driven problem, where tasks are ready to execute as soon as their input dependencies are satisfied, leads to the second element of the re-structured algorithm, namely the elimination of centralized control entity and the use of a distributed control model. Under this model, the logic is split across the three primary components: the coarse solver, the fine solver, and a convergence testing component. Each component maintains a collection of tasks that can execute once their (internal and external) dependencies are satisfied. In the IPS implementation, data are stored in files by the generating tasks, and the asynchronous event service of the IPS is used to signal the availability of the data that the components depend on.

Fig. 2 shows the dependencies between the different tasks in dependency-driven parareal. We distinguish between several kinds of dependencies used in the algorithm. *Data flow* dependencies and *state update* dependencies constitute the core of the parareal algorithm. Note that we choose to use the fine output from two successive iterations to determine the convergence of any given time slice. In addition, we choose to express the state update dependency as a precondition for executing a coarse solve task, even though the data is only needed *after* the coarse task has finished. This choice avoids the need to perform the state update twice for both the dependent fine and coarse tasks, and adds no additional delays to the algorithm. *Convergence status* dependencies communicate the decisions of the convergence component. Note that the coarse and fine tasks for time slice i, iteration k depend on convergence of both time slice i and $i-1$ in iteration $k-1$. This added dependency is used to determine if i is the first slice to run in iteration k, which then will use the output from the fine solver in slice $i-1$, iteration $k-1$ ($\lambda^F_{i-1,k-1}$) instead of the output from the (not executed) coarse task for time slice $i-1$, iteration k ($\lambda^G_{i-1,k}$). This added dependency also has no impact on overall timing, since the convergence status for slice i can only be made after the convergence status for all slices $j < i$ has been determined in any given iteration.

While some particular details vary among the three components, they share a general overall structure which is presented in Fig. 3 for the coarse component. As can be seen in Fig. 3, the algorithm for a constituent component in the task-based parareal implementation is composed of four phases. In the initialization phase (lines 3–8), the component subscribes to event topics where the other two components publish their status updates. In addition, the first task to compute $\lambda^G_{1,1}$ is initialized with its only external dependency λ_0. This task is then placed into a FIFO queue maintained by the component for later execution.

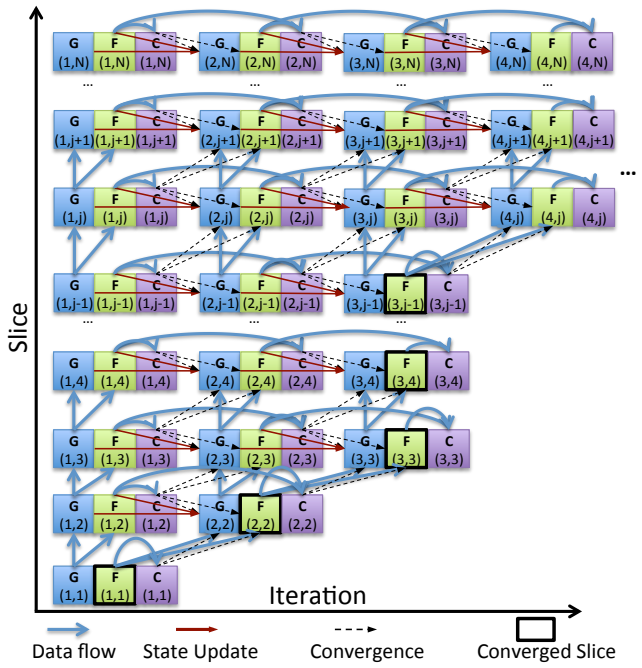


Figure 2: Dependencies in the parareal algorithm. G,F,C represent coarse, fine, and convergence testing tasks, respectively. Indices are (iteration, slice).

In the task dispatch phase of the algorithm (lines 12–18), queued tasks that are ready to run are submitted for execution via the IPS framework. Tasks in the IPS are standalone MPI programs that are executed, within the IPS's batch resource allocation, using the host platform's native MPI execution command (e.g., `mpiexec`). Using the framework's non-blocking task launch capability, tasks are submitted until the component runs out of ready tasks, or until task submission fails due to unavailability of computational resources. In the latter case, the failed task is re-queued for future execution.

The component then proceeds to the event processing phase of the algorithm (lines 21–33) where the event service of the IPS is polled for events published to the subscribed topics, and where such events (if any) are processed. A polling frequency of $O(0.1 - 1.0\text{s})$ is perfectly adequate to avoid delays, as tasks typically run for seconds to hours, while minimizing the load due to polling. Between polls, the component process itself sleeps, waiting for computational tasks it has launched to complete. The `ALL CONVERGE` event published by the convergence component signals the termination of the algorithm. Other events signal the termination of tasks executed by the other components (fine tasks or con-

```
1 // Phase 1 :  Initialization
2
3 subscribe(FINE EVENTS)
4 subscribe(CONVERGE EVENTS)
5 taskQueue = Queue()
6 task_{1,1} = newTask()
7 satisfyDependency(task_{1,1}, λ_0)
8 addTask(taskQueue, task_{1,1})
9
10 while not done do
11    // Phase 2:  Task dispatch
12    for task ∈ taskQueue do
13       if launchTask(task) == success then
14          continue
15       else
16          break
17       end
18    end
19
20    // Phase 3:  Event Processing
21    events = pollTopics()
22    for e ∈ events do
23       if e == ALL CONVERGE then
24          done = TRUE
25          break
26       else if e ==TASK FINISHED then
27          data = e[task_output]
28          for task ∈ dependent tasks do
29             satisfyDependency(task, data)
30             if readyToRun(task) then
31                addTask(taskQueue, task)
32          end
33    end
34
35    // Phase 4:  Completed tasks processing
36    finishedTasks = checkTasks(runningTasks)
37    for task ∈ finishedTasks do
38       publish(COARSE EVENTS, task)
39       data = task[output]
40       for deptask ∈ dependent tasks do
41          satisfyDependency(deptask, data)
42          if readyToRun(deptask) then
43             addTask(taskQueue, deptask)
44       end
45    end
46 end
```

Figure 3: The dependency-driven parareal algorithm (coarse solver).

vergence check tasks). In addition to a distinguishing type, these events carry a payload which describes the context of the event. In the case of convergence events, that payload will contain the time slice and iteration indices of the task, along with the result of the convergence test. In the case of fine task completion events, the payload will contain the indices of the task, along with file system path to any data generated by the task that is used in the parareal algorithm.

Data from external events are used to update the dependencies of the component's own local tasks. A task is ready to run, and is placed in the ready queue, once all its dependencies are satisfied (as reported by the `readyToRun()` method and depicted in Fig. 2). Note that for the convergence component, tasks may not be computationally intensive, hence can be directly executed by the component with no need for scheduling and launching using the framework. The `readyToRun()` method also verifies whether the task actually needs to be run. Time slices beyond the simulation's configuration, and those which have converged before all dependencies have been satisfied do not need to be executed.

The final phase in the algorithm (lines 36–46) involve the processing of tasks that had been submitted during the task dispatch phase, and which have finished execution. The output from such tasks is used to update the input dependencies of other local tasks. Furthermore, events are published to the appropriate topic signaling the termination of the task, along with a payload that contains the task indices and task output that is needed by the other components in the simulation.

While the classic formulation of parareal serializes the coarse solve, the dependency-driven approach allows complete flexibility to execute any task for which the previous dependencies have been satisfied, on any available resources. This effectively eliminates the sequential bottleneck of the coarse propagation by allowing tasks from multiple time slices and multiple iterations to execute concurrently. As a result, resource utilization can be significantly improved, and the wall-clock time to solution significantly reduced (e.g., the "dependency-driven" (blue) curve in Fig. 4(b)).

6. MOVING WINDOW PARAREAL

In both the classic parareal algorithm, and the dependency-driven formulation, system state is calculated for all non-converged time slices in any given iteration up to and including the last time slice, N. In the early iterations ($k << K$), this aspect of the algorithm results in the fairly inaccurate coarse solver being used to compute a relatively large number of provisional system states whose "quality" rapidly deteriorates the further it is away from the first non-converged time slice in any given iteration (i.e. from the last known accurate state). This means that work performed in the upper left triangle in Fig. 2 tends to contribute little to the overall solution, which manifests as large error values (Fig. 4(a)). These calculations contribute to the resources consumed by the algorithm, while contributing little to the rate of convergence.

To address this problem, we introduce a moving window variation of the parareal algorithm that aims to reduce the overall resource utilization of the algorithm by not executing tasks that contribute little or nothing to the speedup. In this formulation, a window size of N_0 is chosen, such that $N_0 \leq N$, and the first N_0 non-converged slices are processed during each iteration. (Thus, when $N_0 = N$ it is the same as the original dependency-driven algorithm.) As slices converge, new slices are added, maintaining the window size, until the edge of the window reaches slice N. The difference may be understood visually by comparing Fig. 4(a) to Fig. 5(a). The diagonal band in Fig. 5(a) represents the window, which advances as slices converge. The white area in the upper left are slices not evaluated in the windowed variant, or those that do not have enough fine solver results to compare in the current iteration.

In the dependency-driven formulation, this variant amounts to an extension of the logic of testing for tasks beyond the

system size to allow for a different set of active slices in each iteration. The other noteworthy change involves extending the special processing for coarse and fine tasks in the first iteration to those tasks that correspond to time slices freshly added during any iteration $k > 1$.

It should be noted that choosing the right value for the window size N_0 is crucial to achieving the right balance between efficient resource utilization and speedy convergence. A too large a value for N_0 increases the likelihood that tasks will be executed with no meaningful contribution to the speedup. Choosing a too small a value for N_0 reduces the "effective depth" of the algorithm where successive parareal state updates for the same time slice improves the accuracy of the state produced by the coarse solver to speed convergence. In the limit, with $N_0 = 1$, the algorithm reduces to sequential application of the fine solver, with no speedup.

7. RESULTS AND ANALYSIS

In order to illustrate the performance and cost characteristics of the different parareal approaches described above, we have applied them to a fusion plasma turbulence problem using the BETA code [19]. This same code had previously been used in a monolithic MPI-based implementation of the classic parareal algorithm by Samaddar, Newman, and Sanchez [22], which facilitated verification of the IPS implementation.

Both the coarse and fine solvers utilized the fast Fourier transform-based BETA code. However the fine solver was configured to use the VODPK [5] adaptive integrator, while the coarse solver used 4th-order Runge-Kutta integration, and also used a reduced number of harmonics. The simulation has been divided into 160 time slices.

The first set of results, comparing the classical and dependency-driven variants were run on a Cray XT5 at the University of Alaska consisting of 432 8-core nodes ("Pingo"). Both the coarse and fine solvers were run with 16 cores per task. Mean execution times were 3.45 s for the coarse task and 223.15 s for the fine. Because of job size limitations on this system, these runs were carried out on 128 nodes (1024 cores), limiting the concurrency to a maximum of 64 tasks compared to the theoretical maximum of 160 (the number of slices).

Fig. 4(a) depicts the convergence of the parareal algorithm in 14 iterations (the same for both formulations). However Fig. 4(b) illustrates the difference in resource utilization and time to completion for the two variants. The processor utilization of the classic version shows a cyclical pattern as the algorithm alternates between the sequential coarse solve (low utilization) and the parallel fine solves (high utilization), converging all 160 slices in 14330.81 s, with a total cost of 4076.32 CPU-hours and an average resource utilization of 31.6%. The dependency-driven version, on the other hand, shows generally high utilization as the coarse and fine solves overlap, completing in 6379.66 s, with a total cost of 1814.66 CPU-hours and an average resource utilization of 70.56%.

Both the classic and dependency-driven versions of the algorithm perform the same amount of work (the same numbers of coarse and fine tasks), but the dependency-driven variant performs it more efficiently because it exposes more parallelism. The windowed variant, on the other hand, reduces the total amount of work by limiting the number of slices evaluated in each iteration. Fig. 5(a) illustrates the convergence heat map for a window size of 40 slices.

Table 1: Summary of Moving Window Parareal Runs and Simulations

Cores	Window Size	Iterations	Observed Time (s)	Simulated Time (s)
160	20	22	5293.08	5433.8
160	30	18	5127.39	4939.2
240	30	18	4653.46	4742.4
160	40	16	5301.94	5205.2
256	160	12	7696.08	7000.8

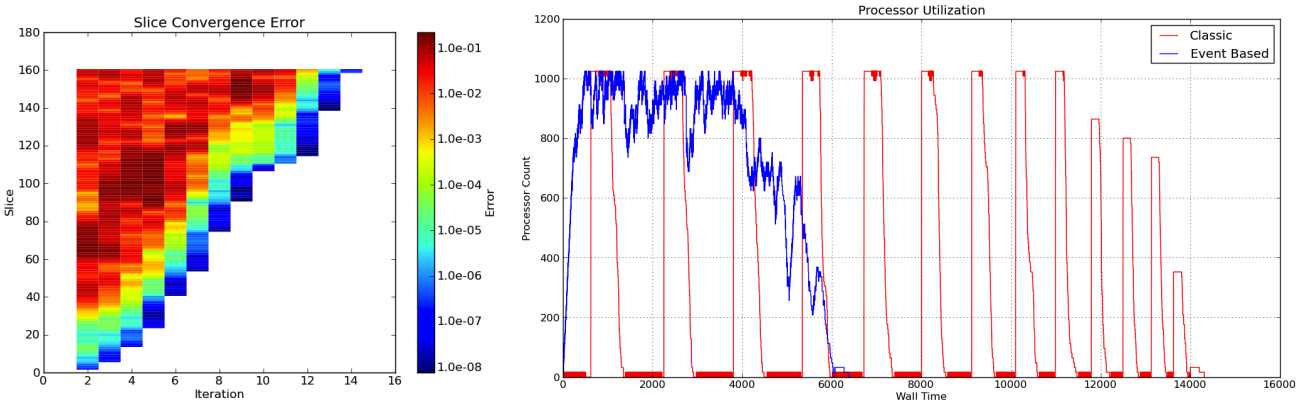
Fig. 5(b) depicts the resource utilization of the moving window dependency-driven parareal implementation for a 40-slice window. Because the window size limits the amount of parallelism available, this job was run on 160 cores, allowing 20 concurrent tasks. We must note that the timings in Fig. 5(b) (and Table 1) are not directly comparable to those in Fig. 4(b). During this work, the Pingo system was repurposed for classified use, and the windowing runs were carried out on a 150-node Linux cluster at the University of Alaska ("Pacman") with a main compute partition made up of 12- and 16-core nodes (1936 cores). Additionally, BETA was run on 8 cores rather than 16 on this system.

Table 1 illustrates the trade-off between window size and the number of iterations required to converge,[1] including the 160-slice window, which, as discussed in Sec. 6, is merely another way of denoting the dependency-driven algorithm *without* the limiting window. As expected, the number of iterations required for convergence increases as the window narrows. This is because the tasks outside the window (which are not run in the windowed version) do make *some* contribution to convergence. For the same resource allocation, the run times vary little for the different window sizes, but all are significantly faster than the full (160-slice) algorithm, even though they were run on fewer nodes.

To get a better understanding of the performance and cost trade-offs of the windowed parareal, we employed the IPS Resource Usage Simulator (RUS) [10], extended to model the various parareal formulations presented here. Task execution times for the simulations were sampled from probability distributions that best matched the observed timings on Pacman. We used the actual observed convergence pattern for the simulations because although an empirical convergence model has been developed for the full problem [22], that model has yet to be modified to account for the use of a moving window of time slices. The final column of Table 1 shows the results of RUS simulations of the actual Pacman runs.

We then used RUS to model a larger range of resource allocations ranging from 8 to 1024 processors, considering both the solution time and the total cost (CPU-hours). Fig. 6(a) shows the timings as a function of the number of nodes allocated. The serial solution (160 successive invocations of

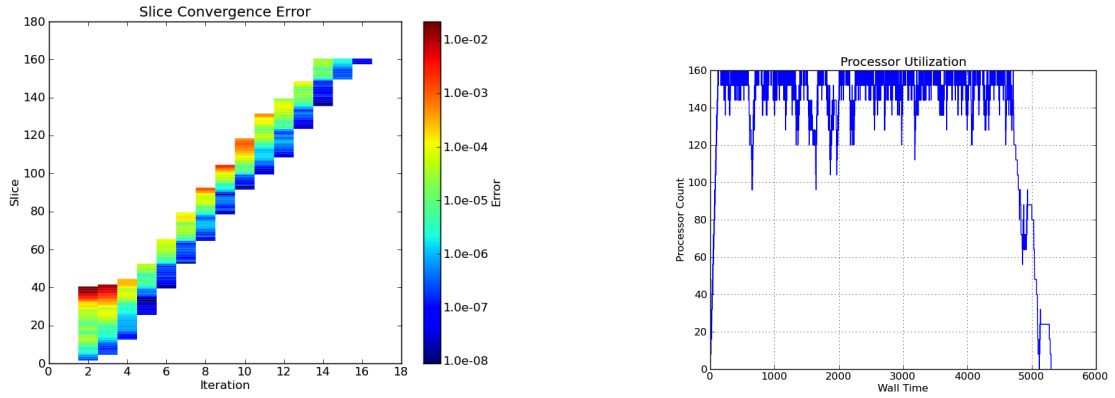
[1]Due to the nonlinear and chaotic nature of the plasma turbulence problem modeled using the BETA code, the number of iterations to convergence may vary, as seen in the results from Pingo and Pacman for the 160-slice dependency-driven algorithm. The results presented here have been verified by the authors as sound.



(a) Convergence Heat Map

(b) Resource Utilization Classic v. Dependency-based

Figure 4: (a) Heat map representation of the convergence error for a parareal formulation of a plasma turbulence problem. The vertical axis represents the 160 time slices into which the time domain has been decomposed. The horizontal axis represents successive iterations of the parareal algorithm. Colors indicate distance from convergence, with red representing the highest errors, and blue representing converged results. (b) Resource utilization graph of classic parareal (red) vs. event-based (blue) parareal of the same 160-slice problem on 1024 cores of Pingo.



(a) Convergence Error Heat Map, 40-Slice Window

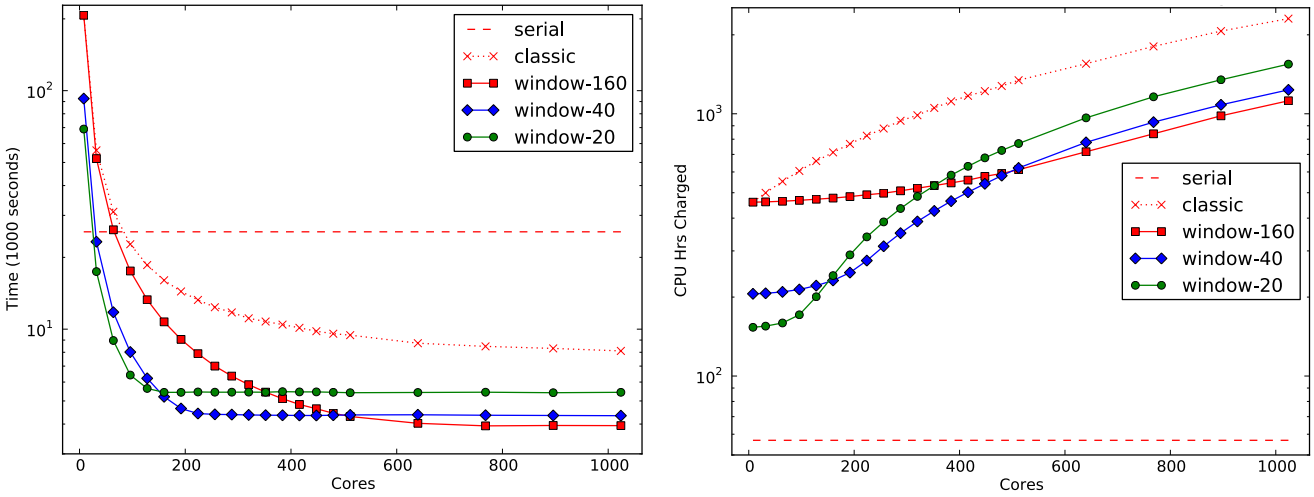
(b) Resource Utilization, 40-Slice Window, 160 cores

Figure 5: (a) Heat map representation of the convergence error for a 40-slice window run. (b) Resource utilization graph of a 40-slice window run on 160 cores of Pacman.

the fine solver) and the classic parareal implementation are shown for reference, along with the dependency-driven versions with 20-, 40-, and 160-slice windows. The 20-slice and 40-slice windowed algorithms are significantly faster than the full (160-slice) algorithm initially, due to the reduced work. However they soon exhaust the parallelism available due to the window's limitations, and flatten out near 160 and 320 cores, respectively. The full 160-slice algorithm provides more parallelism, and so eventually becomes faster than the smaller windows, but for modest resource allocations, the reduced work of the windowed approach provides significant benefits in the run time. In this region, the particular choice of window size is less significant. It is also worth noting that all versions of parareal quickly become faster than the serial solution, starting at 48–96 cores.

Fig. 6(b) shows the cost of the approaches. First, we must note that all of the parareal variants are significantly more expensive than the serial version—the whole point of parareal is to trade resources (total cost) for solution time. Next, we observe that the classic parareal is significantly more expensive than any of the dependency-driven versions because the sequential treatment of the coarse solve makes it less efficient. Finally, we see that the smaller windows are much less expensive than the full 160-slice algorithm because of the reduced work. However, as observed in the timings, when the windowed algorithm exhausts the available parallelism, its costs start to rise rapidly.

Taken together, these two plots showcase the efficiency afforded by using the dependency-driven moving window approach to parareal, when executed in a flexible environment such as the IPS. The algorithm provides "knobs" to allow the user to select the set of parameters that satisfies both the specific run time and resource utilization constraints that govern the modern high-end scientific computing community.



(a) Simulated Time of Parareal Implementations

(b) Simulated Cost of Parareal Implementations

Figure 6: Log scale plots of time (a) and cost (b) versus allocation size for just running the fine solver (red dashed line), classic parareal (red Xs and dotted line), event-based implementation (red squares and solid line), and windowed implementations for 20 slices (green circles) and 40 slices (blue diamonds).

8. SUMMARY AND FUTURE WORK

In this paper, we have presented a novel dependency-driven formulation of the parareal algorithm for parallelization of time-dependent PDE problems in the temporal domain. This approach to the algorithm was inspired by many-task computing concepts and implemented in the Integrated Plasma Simulator, which supports both many-task style computations and more traditional HPC simulations. Dependency-driven parareal exposes more parallelism than the "classic" formulation, thus allowing better utilization of resources allocated to the simulation, and faster completion. For a plasma turbulence simulation, we demonstrated improvements of more than a factor of two in both average resource utilization and time to completion.

We also described a variant of parareal which uses a moving window to limit the number of time slices evaluated in each iteration of the algorithm. This approach avoids treating slices that are far from convergence and therefore contribute little to the overall solution. Experiments show that the windowed dependency-driven parareal easily offers a 30% reduction in time to solution and even larger reductions in total cost over the full (non-windowed) version. Using a simulator to explore a broader range of resource allocations, we see that windowed parareal can be very effective compared to the full algorithm for modest resource allocations. Once the allocation is large enough to exhaust the available parallelism, which is limited by the window size, the benefits plateau and eventually, the full algorithm will beat the windowed, both in terms of time to solution and total cost.

These formulations of parareal offer significant benefits to researchers seeking to expose more parallelism in their simulations in order to take advantage of the rapidly increasing core and processor counts on today's massively parallel systems. Because we have eliminated the sequential bottleneck of the classic parareal, the success of the method is much less dependent on the ratio of execution times T_F/T_G being large. This opens the door for the exploration of coarse solvers which are much more accurate than those typically used today, which have the potential to significantly accelerate convergence. We expect this to be the most significant area of future work following this paper.

9. ACKNOWLEDGMENTS

This work has been supported by the U. S. Department of Energy, Offices of Fusion Energy Sciences and Advanced Scientific Computing Research, and by the ORNL Postmasters Research Participation Program which is sponsored by ORNL and administered jointly by ORNL and by the Oak Ridge Institute for Science and Education (ORISE). ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725. ORISE is managed by Oak Ridge Associated Universities for the U. S. Department of Energy under Contract No. DE-AC05-00OR22750. The authors are grateful for grants of supercomputing resources at the University of Alaska, Arctic Region Supercomputing Center (ARSC) in Fairbanks.

10. REFERENCES

[1] Center for Simulation of RF Wave Interactions with Magnetohydrodynamics. `http://cswim.org`.

[2] E. Aubanel. Scheduling of tasks in the parareal algorithm. *Parallel Computing*, 37(3):172–182, Mar. 2011.

[3] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zérah. Parallel-in-time molecular-dynamics simulations. *Phys. Rev. E*, 66(5):057701, Nov 2002.

[4] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[5] S. D. Cohen and A. C. Hindmarsh. CVODE, a stiff/nonstiff ODE solver in C. *Comput. Phys.*, 10:138–143, March 1996.

[6] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13:219–237, July 2005.

[7] W. Elwasif, D. Bernholdt, A. Shet, S. Foley, R. Bramley, D. 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.

[8] W. R. Elwasif, D. E. Bernholdt, S. S. Foley, A. G. Shet, and R. Bramley. Multi-level concurrency in framework for integrated loosely coupled plasma simulations. In *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2011, Sharm El-Sheikh, Egypt*, December 2011. to appear.

[9] P. Fischer, F. Hecht, and Y. Maday. A parareal in time semi-implicit approximation of the Navier-Stokes equations. In T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick, R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Widlund, and J. Xu, editors, *Domain Decomposition Methods in Science and Engineering*, volume 40 of *Lecture Notes in Computational Science and Engineering*, pages 433–440. Springer Berlin Heidelberg, 2005.

[10] S. Foley, W. Elwasif, D. Bernholdt, A. Shet, and R. Bramley. Many-task applications in the Integrated Plasma Simulator. In *3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS 2010*, Nov 2010.

[11] S. S. Foley, W. R. Elwasif, and D. E. Bernholdt. The Integrated Plasma Simulator: A flexible Python framework for coupled multiphysics simulation. In *PyHPC 2011: Python for High Performance and Scientific Computing*, November 2011. accepted.

[12] M. J. Gander and S. Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM J. Sci. Comput.*, 29(2):556–578, 2007.

[13] I. Garrido, M. Espedal, and G. Fladmark. A convergent algorithm for time parallelization applied to reservoir simulation. In *Domain Decomposition Methods in Science and Engineering*, volume 40 of *Lecture Notes in Computational Science and Engineering*, pages 469–476. Springer Berlin Heidelberg, 2005.

[14] J.-L. Lions, Y. Maday, and G. Turinici. A"parareal" in time discretization of PDE's. In *Mathématique*, volume 332 of *1*, pages 661–668, Paris, 2001. Comptes rendus de l'Académie des sciences.

[15] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[16] Y. Maday and G. Turinici. The parareal in time iterative solver: a further direction to parallel implementation. In T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick, R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Widlund, and J. Xu, editors, *Domain Decomposition Methods in Science and Engineering*, volume 40 of *Lecture Notes in Computational Science and Engineering*, pages 441–448. Springer Berlin Heidelberg, 2005.

[17] Q. Meng, J. Luitjens, and M. Berzins. Dynamic task scheduling for the Uintah framework. In *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, pages 1–10, nov. 2010.

[18] M. Minion. A hybrid parareal spectral deferred corrections method. *Communications in Applied Mathematics and Computational Science*, 5:265–301, Dec. 2010.

[19] D. Newman, P. Terry, and P. Diamond. A two-nonlinearity model of dissipative drift wave turbulence. *Physics of Fluids*, 4(3):599–610, 1992.

[20] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[21] I. Raicu, I. Foster, and Y. Zhao. Many-Task Computing for Grids and Supercomputers. In *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)*. co-located with IEEE/ACM Supercomputing 2008, Nov 2008.

[22] D. Samaddar, D. E. Newman, and R. Sánchez. Parallelization in time of numerical simulations of fully-developed plasma turbulence using the parareal algorithm. *J. Comput. Phys.*, 229(18):6558–6573, September 2010.

[23] A. G. Shet, W. R. Elwasif, S. S. Foley, B. H. Park, D. E. Bernholdt, and R. Bramley. Strategies for Fault Tolerance in Multicomponent Applications. In *Proceedings of the International Conference on Computational Science, ICCS 2011*, volume 4, pages 2287–2296, 2011.

[24] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11, New York, NY, USA, 2009. ACM.

[25] K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C. S. Jun, and J. Tsujii. Design and implementation of GXP Make – a workflow system based on Make. In *e-Science (e-Science), 2010 IEEE Sixth International Conference on*, pages 214–221, Dec. 2010.

[26] D. Warneke and O. Kao. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.

[27] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206, July 2007.