Explorations to Optimize the Parareal Algorithm to Solve ODEs in Given Applications

Wesley Chen, Brandon Sim, Andy Shi
Harvard University, Applied Math 205 Final Project

December 10, 2014

Abstract

We look to achieve strong scaling in parallelizing numerical ODE methods. We test both parallelism by result space as well as parallelism by time, with the Parareal algorithm, to solve various ODEs. We measure performance and compare in terms of accuracy, speedup and efficiency. We provide some theoretical analysis for the Parareal algorithm's runtime, speedup, error convergence, and stability. For the Parareal algorithm, we observe preliminary promising but not convincing results in our small scale tests up to 32 processors on Harvard's Odyssey computing cluster. We analyze possible reasons as to why our implementation may not be ideal and some tradeoffs that are taken in the Parareal implementation. Further optimizations are proposed and rationalized as next steps including ports of to C++ using BLAS. Our code is written in Python using mpi4py. We compare tests from various data sources from basic exponential ODEs to larger grids of ODEs to solve for in applications like sound wave propagation.

1 Background

1.1 Numerical Methods and Parallelization

Solving systems of differential equations can be a computationally expensive task. The error of most algorithms scales on the step size of the discretization of time. However, step size in time is also proportional to the computation required. It would be nice to allow for strong scaling where a problem can be solved in a reasonable time for very small time steps.

Another obstacle to trying to parallelize numerical methods is that the methods are inherently serial in time—in that evaluation of the next time point $n+1$ depends on the previous values, say those at time $n-1$ and n. This setup does not allow for parallelism. The only way to incorporate parallelism is to create schemes that have parallelizable components—such as an update or a refinement built on top of a more basic method first computed in serial. As will be discussed and analyzed in the methodology section, one such scheme would be the Parareal algorithm. As an overview, the Parareal algorithm first does a coarse (low-order and fast) approximation, then seeks to iteratively correct using smaller, more refined numerical methods done in parallel from approximated starting points approximated by the coarse solution.

An alternative paradigm to applying parallelism to numerical methods is the parallelism by space paradigm, where the result vector space is divided up to different regions and sent to different responsible processors. In this way, each processor sees a similar but smaller-scale problem—weak scaling. This method is expected to outperform the parallel in time paradigm simply because the task is embarrassingly parallel, being able to be naturally divided into smaller but identical in method problems. There is no serial component in this algorithm and should be exhibit both strong and weak scaling.

In general, the division in space paradigm is more intuitive to parallelize and will often lead to greater speedups because there is no serial component. The main tradeoff however, is that knowing how to divide the resultant system requires knowledge of problem-specific details. Thus, this approach lacks the general stability and theoretical proofs (of error and convergence) that schemes which parallelize in time can offer.

1.2 Measuring Parallel Performance and Amdahl's Law

Performance of parallel algorithms are usually measured in terms of 3 different metrics: speedup, strong scaling efficiency and weak scaling efficiency. Strong and weak scaling are common measures for the different ways a program can benefit from increased parallelism and more processors as well as the limitations that dictate the speedups [1]. In each of the below subsections, $T(n)$ represents the computation of using n processors and the same

notation for speedup.

1.2.1 Speedup

Speedup, S is defined simply as the ratio of the serial implementation to the parallelized version with a given number of processors. These numbers are usually reported per number of processors that are tested:

$$S(n) = \frac{T_{\text{serial}}}{T_{\text{parallel in n cores}}} \tag{1}$$

1.2.2 Strong Scaling Efficiency

Strong scaling is a way to measure performance of a given application depending on if it is CPU-bound. This means that given a fixed problem size, with more processors, the computation time will decrease for each. Programs that are difficult to parallelize will have efficiency expressions like $\frac{1}{\ln p}$ which approaches zero as the number of processors increases by a large amount meaning that adding additional processors does not give much more speedup. We can compute the strong scaling efficiency with the follow expression:

$$\text{Eff}_{\text{strong}} = \frac{T(1)}{n \times T(n)} \times 100\% \tag{2}$$

1.2.3 Weak Scaling Efficiency

Weak scaling measures the memory-bound nature of the parallelized application. A weak scaling parallel implementation is one that allows a larger problem proportionally increased for each processor to be solved in the same time as the original. This is usually the more intuitive usage and measure to quantify parallel application performance. Below is the computation for weak scaling efficiency:

$$\text{Eff}_{\text{weak}} = \frac{T(1)}{T(n)} \times 100\% \tag{3}$$

1.3 Amdahl's Law

The final principle when discussing parallel performance is the potential existence of an Amdahl speedup limit. Amdahl's Law seeks to describe the maximum parallel speedup in an overall piece of code when the problem is not embarrassingly parallel—not all of the system can be divided up in parallel. The parallel in time methods must always have some serial component due to the time-dependent nature of the ODE's which will fall under the limitations stated by Amdahl's Law. His law expresses both best time given n processors as well as the speedup for n processors as a function of B, the fractional amount of the algorithm that cannot be parallelized—due to strict serial constraints:

$$T(n) = T(1)\left(B + \frac{1}{n}(1-B)\right) \tag{4}$$

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)} \tag{5}$$

2 Methodology

The Parareal algorithm, developed by Lions, Maday and Turinici in 2001 [2], is a generalized algorithm which allows for parallelization in time. It does so by using a cheaper ($g_{\Delta t}$)—a lower order or lesser resolution approximation first, which it then corrects by evaluating segments with a finer method g_{fine}. The correction step from the approximation can be done in parallel. Multiple iterations can be performed to achieve an error provably asymptotically equal to a full serial computation of g_{fine}. The exact number of iterations required to achieve certain levels of convergence depends on the problem and must be tuned top optimize for speedup for a given system. In summary, the algorithm is k repetitions of a correction process which requires running updated by a finer method run in parallel for subsections.

The beauty of the Parareal method is that it can be adapted into many different versions with different choices of the $g_{\Delta t}$ and g_{fine} solution operators. The g_{fine} solution is a more computationally expensive method but assumed to have lesser error. Most of the time this will be a higher order method, but can also be a greater sampling of the same order method used in the coarse version. The exact optimum pairing is problem-specific and the code is written so that a different "n-order method step" function could be written and then replaced into the code. We experiment with testing different pairs of $g_{\Delta t}$ and g_{fine} to see how influential the choice of solution scheme is.

2.1 Parareal Algorithm and Visualization

Algorithm 1: Parareal

Input: Temporal discretization $t_n = t_0 + n\Delta t$, $n = 1, 2, \ldots, N$
Input: Coarse scheme $g_{\Delta t}$
Input: Finer scheme g_{fine}
1 Compute $u^1_{n+1} = g_{\Delta t}(t_n, u^1_n)$;
2 Compute the corrections $\delta g(u^1_n) = g_{\text{fine}}(t_n, u^1_n) - g_{\Delta t}(t_n, u^1_n)$ in parallel;
3 Add the prediction and correction terms as $u^2_{n+1} = g_{\Delta t}(t_n, u^2_n) + \delta g(u^1_n)$;
4 Repeat steps 2 and 3, incrementing the iteration label and using $u^{k+1}_0 = u^1_0$ as the initial condition;

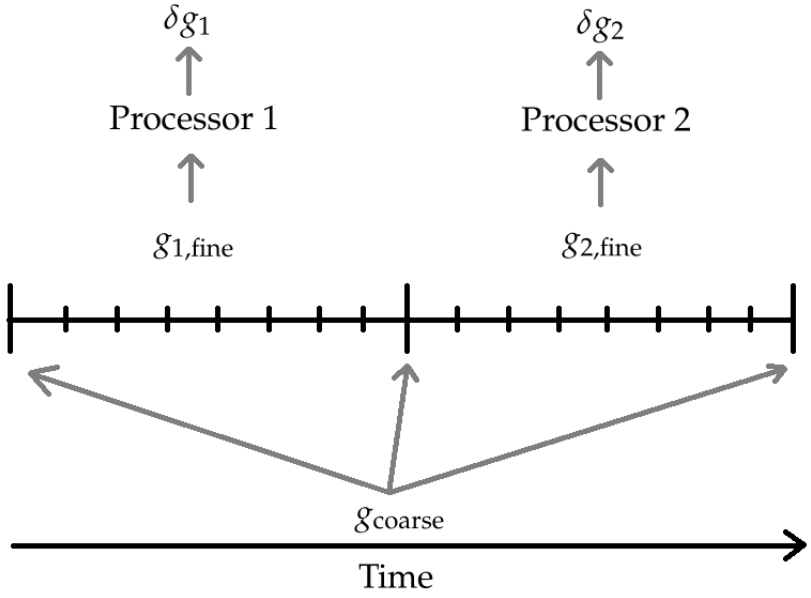

Figure 1: Graphical representation of the Parareal scheme.

2.2 Convergence of the Parareal

Assume the coarse operator $g_{\Delta t}$ is Lipschitz and order m: for some constant L,

$$|g_{\Delta t}(t_n, u) - g_{\Delta t}(t_n, v)| \leq (1 + L\Delta t)|u - v| \; \forall t \in (0, t_n),$$

$$|u(t_n) - u^1_N| \leq C(\Delta t)^m |u_0|.$$

Additionally, assume the function u is bounded on $(0, t_n)$. Furthermore, assume the fine solution operator g_{fine} is a sufficiently accurate approximation to the analytic operator so we may replace $g_{\text{fine}} \to g$.

Theorem: The order of accuracy of the Parareal method with coarse solution operator $g_{\Delta t}$ and fine operator g is mk. We can prove this using induction [3].

Proof: By induction.

Case $k = 1$: This is just the coarse operator, which is order m.

Assume that this is true for k, that $|u(t_N) - u_N^k| \leq C(\Delta t)^{mk}|u_0|$.

For the case $k+1$, we have

$$\begin{aligned}
|u(t_N) - u_N^{k+1}| &= |g(u(t_{N-1})) - g_{\Delta t}(u_{N-1}^{k+1}) - \delta g(u_{N-1}^k)| \\
&= |g_{\Delta t}(u(t_{N-1})) - g_{\Delta t}(u_{N-1}^{k+1}) - \delta g(u_{N-1}^k) + \delta g(u(t_{N-1}))| \\
&\leq |g_{\Delta t}(u(t_{N-1})) - g_{\Delta t}(u_{N-1}^{k+1})| - |\delta g(u_{N-1}^k) + \delta g(u(t_{N-1}))| \\
&\leq (1 + C\Delta t)|u(t_{N-1} - u_{N-1}^{k+1}| + C(\Delta t)^{m+1}|u_{N-1}^k - u(t_{N-1})| \\
&\leq (1 + C\Delta t)|u(t_{N-1}) - u_{N-1}^{k+1}| + C(\Delta t)^{m(k+1)+1}|u_0|
\end{aligned}$$

We can continue to expand $|u(t_{N-1} - u_{N-1}^{k+1}|$ and get time indices of $N-1, N-2, \ldots, 2, 1$. This implies that

$$|u(t_N) - u_N^{k+1}| \leq C(\Delta t)^{m(k+1)}|u_0|,$$

as desired.

2.3 Error of the Parareal

With the assumptions of the previous section, this Parareal method will approach, with large enough k (correction iterations) to approach the error of the fine method. However, there is a time vs. accuracy tradeoff. Let Q be the *Quality Factor* for $g_{\Delta t}$ and g_{fine}. Suppose that, for constant number of processors and Δt, $g_{\Delta t}$ runs in time T. Then, g_{fine} runs in time QT. With too large of a k and a lower Q, the time for the Parareal could potentially take longer than the direct serial computation. We note that there could be other relationships between the time of the coarse and fine schemes other than multiplication by a constant term.

Disregarding the time relationship between $g_{\Delta t}$ and g_{fine}, we can look at what happens to the error as $k \to N$ (recall that N is the total number of time points in the time discretization).

For $k = 1, 2, \ldots$ we have

$$u_{n+1}^{k+1} = g_{\Delta t}(t_n, u_n^{k+1}) + \left(g_{\text{fine}}(t_n, u_n^k) - g_{\Delta t}(t_n, u_n^k)\right).$$

As $k \to N$, the Parareal algorithm gives $u_n^{k+1} = u_n^k$, so the order of accuracy approaches that of g_{fine} [4].

2.4 Stability of the Parareal

With the Parareal method, it is possible to combine ODE solvers. The stability region depends on both $g_{\Delta t}$ and g_{fine}, and the equation being solved.

Let us consider the ODE $du/dt = \lambda u$. Let $g_{\text{fine}}(t_n, u_n) = \bar{g_{\text{fine}}} u_n$ and $g_{\Delta t}(t_n, u_n) = \bar{g_{\Delta t}} u_n$. As shown by Staff et al. [5], the Parareal method becomes

$$u_n^k = \left(\sum_{j=0}^{k} \binom{n}{j} (\bar{g_{\text{fine}}} - g_{\Delta t})^j \bar{g_{\Delta t}}^{n-j} \right) u_0 = H(\bar{g_{\Delta t}}, \bar{g_{\text{fine}}}, n, k, \lambda) u_0$$

This is stable if $\max_{n,k} |H| \leq 1$.

When $\lambda \in \mathbb{R}$ and $\lambda \leq 0$, Staff et. al also show that

$$\begin{aligned} |H| &\leq \sum_{j=0}^{n} \binom{n}{j} |\bar{g_{\text{fine}}} - g_{\Delta t}|^j |\bar{g_{\Delta t}}|^{n-j} \\ &= (|\bar{g_{\text{fine}}} - g_{\Delta t}| + |\bar{g_{\Delta t}}|)^n \leq 1 \end{aligned}$$

To satisfy these conditions, we have that

1. $|\bar{g_{\text{fine}}}| \leq 1$: This is the usual stability requirement.
2. $|\bar{g_{\text{fine}}} - 2\bar{g_{\Delta t}}| \leq 1$.

2.5 Parallel Tradeoff Analysis

2.5.1 Parareal

We look to compute a theoretical maximum speedup enabled by the nature of the Parareal algorithm. As this is a tradeoff between error (with parameter k being the number of iterations of corrections, each which takes another cycle of $g_{\Delta t}$, we seek to express the tradeoff as a function of the quality factor, Q, the number of iterations, k and the number of processors n. Our quality factor is defined as the multiplicative factor of additional time necessary to compute the solution using the g_{fine} scheme instead of the $g_{\Delta t}$ method.

Then, the runtime of Parareal is, assuming negligible overhead and ignoring communication bandwidth and letting t be the time in seconds to compute the coarse $g_{\Delta t}$:

$$t + k\left(t + \frac{Qt}{N}\right). \tag{6}$$

The first t seconds comes from the first coarse approximation, without which the Parareal algorithm degenerates to $g_{\Delta t}$. Then for each of the k correction iterations, the t term comes from n, element by element evaluations of $g_{\Delta t}$ followed by a parallel $\frac{1}{N}$ evaluation of g_{fine}. In order for there to be a speedup relative to the fine operator, we can rearrange to get:

$$\begin{aligned} t + k\left(t + \frac{Qt}{N}\right) &< Qt \\ k &< \frac{Q-1}{1+Q/N} \\ N &> \frac{Qk}{Q-1-k} \end{aligned}$$

This can be tuned with by either adjusting the the quality factor or reducing the number of iterations, but the number of iterations, k, is important for error convergence to that of g_{fine}. However, finding the optimal k is a problem-specific parameter and traded off for the desired accepted error.

2.5.2 Parallel By Space

For the parallelism by space, we are dividing our grid on which we which to solve our wave diffusion equation. This division is inherently parallelizable and allows us to follow a very similar approach in computing this as in the serial version - just divided in space, where the stencil approach is well-defined. A fully parallelizable approach allows for n times speedup and no computational tradeoffs if we ignore communication. As we are dividing our processors into a processor grid (dividing a 2D space into a 2D arrangement of processors) we are only limited by when the processor grid size approaches the problem size—upon which further processors would not receive any tasks since that would require a division by less than a unit of computation.

2.6 Incorporating Communication

To delve one step deeper in our analysis, parallel communications are usually expressed by architecture parameters—α, β and γ explained in the below table. The expressions measure the communication times spent by each processor (on average as not all processors

Variable	Description
α	Latency of communication. (sec)
β	Inverse communication bandwidth. (sec/byte)
γ	Inverse computational performance. (sec/flop)
n	Number of processors
P	Abstracted problem size

Table 1: Table of Communication Architecture Parameters

may communicate equally depending on the size of the problem). This helps model the communication overhead.

2.6.1 Parareal

For the parareal method, the communications are solely group communications, a broadcast and gather for each corrective iteration, so 2k α. The β term is from the amount being sent - which is one vector for the results of the $g_{\Delta t}$. Finally the cost of the computation we seek to multiply γ by an abstract function $\xi(P_n)$ which returns the cost in flops of computing on the input block, P_n.

$$C_{\text{parareal}} = \alpha * 2k + \beta \cdot kt + \gamma \cdot \xi(kQt/n)$$

2.6.2 Parallel By Space

The main communication of the parallel in space method is the sendrecv (a round robin style send and receive for the ghost points - the corners have 2 pairs of sendrecvs) which contribute to 4 α plus the initial scatter and final gather each with an α term. The amount communicated is the answer as well as the original broadcasted space which includes boundary points. Then, as before, there will be the gamma term multiplied into the computation of the component—given again by an abstract function $\xi(P_n)$ which returns the cost in flops of computing on the input block, P_n.

$$C_{\text{paraspace}} \leq \alpha \cdot 6 + \beta \cdot P/n + \gamma \cdot \xi(P/n)$$

2.7 Amdahl's Law Analysis

One of the main drawbacks of using Amdahl's Law as benchmarks in analysis is in the generally abstract way the paralleizable fraction is determined, the B term. The value of this is estimated based on an understanding of the algorithm from a high level and not derived from any underlying mathematical expressions.

2.7.1 Parareal

We first try to express B as function of the quality factor Q and the number of correction steps of the Parareal k.

From our analysis in equation 6, when we set N to 1 for the generic setup, we see that our total computation is $t + k(t + Qt)$. Of this, the only parallelizable part is the kQt term. Thus to find the expression of the non-parallelizable portion, we simplify:

$$B = \frac{t + kt}{t + kt + kQt} = \frac{1 + k}{kQ + k + 1}$$

Now to replace this into Amdahl's Law, we have:

$$\begin{aligned} S(n) = \frac{T(1)}{T(n)} &= \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1 - B)\right)} = \frac{1}{\left(B + \frac{1}{n}(1 - B)\right)} \\ &= \frac{T(1)}{T(1)\left(\frac{1+k}{kQ+k+1} + \frac{1}{n}(1 - \frac{1+k}{kQ+k+1})\right)} \\ &= \frac{n(kQ + k + 1)}{n + nk + kQ} \end{aligned}$$

As written below, we have used $Q = 100$ with various k.

When $k = 2$ and $Q = 100$, for maximal speedup (fewest iterations) of:

$$\begin{aligned} S(n) &= \frac{203n}{3n + 200} \\ \lim_{n\to\infty} S(n) &\approx 67 \end{aligned}$$

When $k = 10$ and $Q = 100$, for more iterations and less error, we'd see an Amdahl bounded max speedup of

$$\begin{aligned} S(n) &= \frac{1011n}{11n + 1000} \\ \lim_{n\to\infty} S(n) &\approx 91 \end{aligned}$$

It is important to note that this speedup is measured with respect to the Parareal algorithm done in parallel. This does not compare the Parareal to doing g_{fine} but rather shows how given infinite processors, our speedup will approach certain asymptotic bounds. We do see however large constants in the denominator which means that we will get better speedup as we increase n past a certain size.

2.7.2 Parallel By Space

Because this task is embarrassingly parallel and nearly 100% of the computation code can be parallelized (disregarding any uneven divisions of the space by the number of processors), we achieve the Amdahl's Law optimal speedup bound of (with B of 0):

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)} = \frac{T(1)}{T(1)\left(0 + \frac{1}{n}(1-0)\right)} = \frac{T(1)}{T(1) * \left(\frac{1}{n}\right)} = n$$

With embarrassingly parallel problem setups, we can optimally achieve up to n times speedup with n processors though this number if quite ideal and will be very difficult to approach (large scaling will have different memory management and greater communication overheads and costs).

2.8 Planned Tests to Run

We wished to explore certain tests that would demonstrate the effects of changing various parameters be it from the solution schemes in the Parareal to the exact system we were trying to solve.

Due to the sheer number of things to explore, we usually try to compare each run to a benchmark we decided was most useful for comparison's sake and the list of parameters we wished to test are:

- General Scaling with Increased N
- Adjusting K in the Parareal
- Using Different Integration Schemes in Parareal for g_{fine}
- Run the Solver on Various Complexities of ODEs

3 Experimental Results and Discussion

3.1 Expectations

We hoped to show that under the properly tuned parameters (namely the k number of iterations), we would get error for the Parareal method that would approach that of

running g_{fine}. There would be some convergence above which increasing k would not help convergence by as much. The expectation was that we would only being to see the best speedups for the Parareal algorithm with large n due to the strong scaling. With limited access to processors, we expected to see speedup comparing the Parareal algorithm to itself but not necessarily to the evaluation of g_{fine}. Finally, we fully expect the parallel in space algorithm to demonstrate good weak scaling, as well as for it to be faster overall since it is a more embarrassingly parallel design. As disclaimed before, the main tradeoff for the parallel in space method is the lack of extensibility across different ODEs applications.

3.2 General Remarks about Results

In general, we were unable to see the promised theoretical speedups and error convergence for the Parareal algorithm. We believe this could be because of several factors. First, we may have some subtle bugs in our implementation of the Parareal algorithms. Additionally, we only tested our implementation on relatively simple problems. The problem with this testing approach is that the fine serial computation of the solution g_{fine} is very cheap. We did not see the running time of the Parareal beat that for the serial implementation of g_{fine}. If we tested our implementation for more expensive g_{fine} and more complicated problems, we might be able to see to see more speedup and better error convergence.

3.3 Comparison to Serial with g_{fine} as a Forward Euler with Smaller Δt

We implemented the Parareal algorithm in Python, using mpi4py to parallelize it. Our coarse operator was forward Euler step, with 100 steps, while the fine operator was forward Euler, with $Q \cdot 100$ steps, where Q is the quality factor. We tested the Parareal algorithm on two sets of differential equations:

Example 1:

$$y'(t) = f(t, y) = \lambda y,\ y(0) = 1$$

Example 2:

$$y''(t) + 2y'(t) + 5y(t) = 0,\ y(0) = 1,\ y'(0) = 0.$$

We ran the algorithm on different numbers of processors and varied k and Q. We define the quality factor Q in this case for how many times smaller the Euler method used for g_{fine} is compared to the Euler method used for $g_{\Delta t}$.

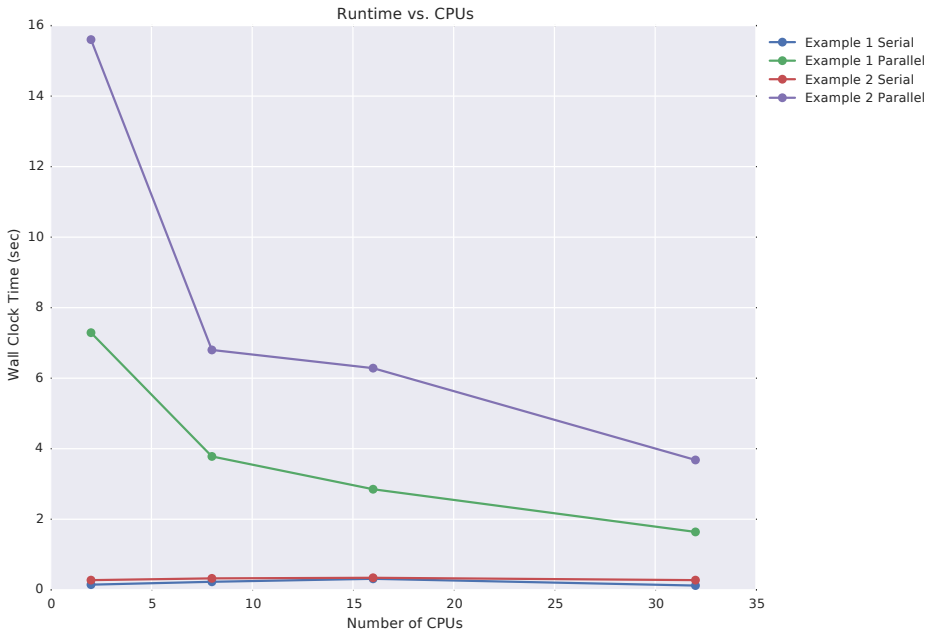

Figure 2: Running time vs. number of CPUs.

Figure 2 shows the running time (wall clock) for the Parareal algorithm compared to different number of CPUs used. The serial version is the Euler method with Q times as many steps. As we can see, the running time decreases as more processors are added. However, there is a base time associated with setting up the parallel infrastructure, so asymptotically the running time is nonzero.

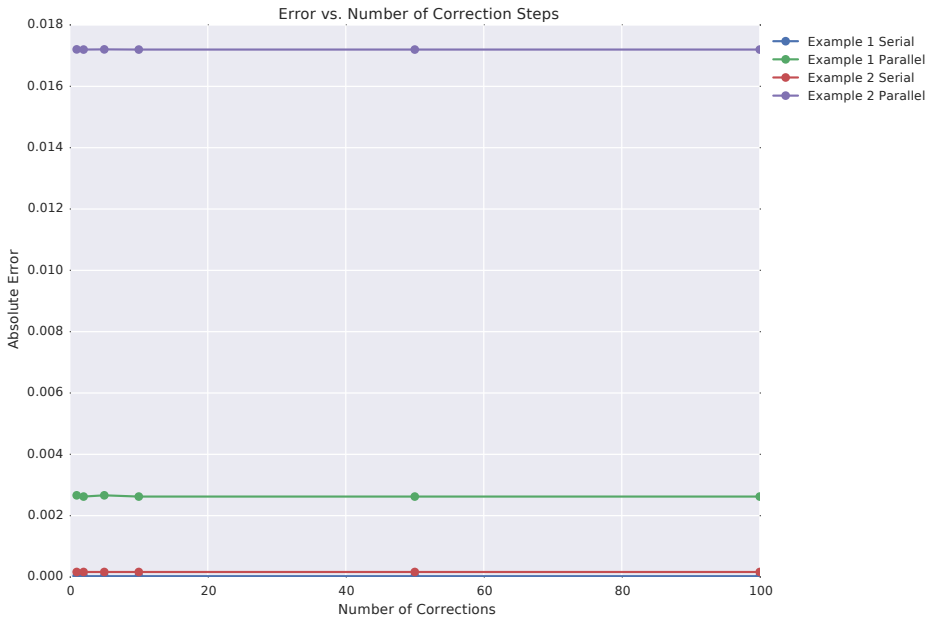

Figure 3: Error vs. number of correction steps.

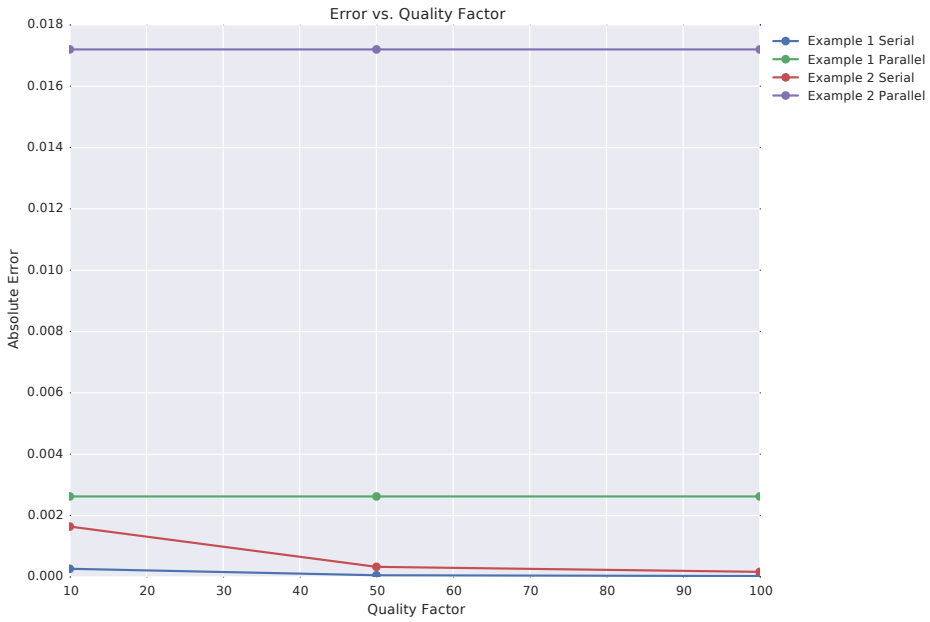

Figure 4: Error vs. quality factor of the fine operator

As seen in Figures 3 and 4, the error remains constant for the parallel versions of the algorithm even as we increase the number of correction steps and the quality factor. This

is not exactly what we expect. We predicted that the error would decrease as we did more correction steps, or if we increased the quality of the g_{fine} operator.

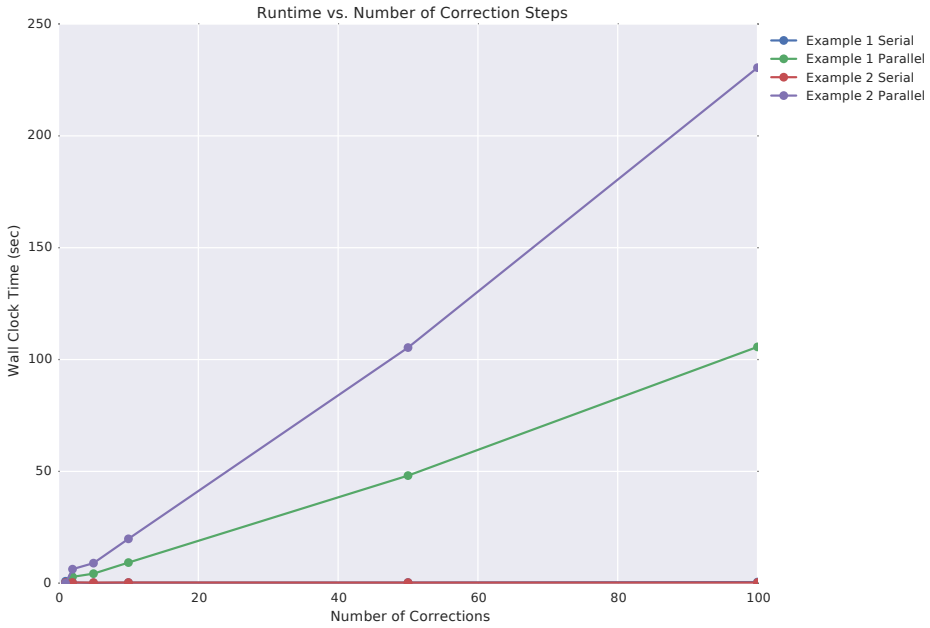

Figure 5: Running time as a function of the number of correction steps.

Figure 5 shows the running time (wall clock) as a function of the number of correction steps. For the parallel implementations, we see an increase in running time as we increase the number of correction steps. This is to be expected, since increasing the number of correction steps increases the number of iterations of our algorithm. Therefore, even though we proved before that as the number of iterations $k \to N$, where N is the number of time points, the order of convergence of the algorithm converges to the order of accuracy of g_{fine}, this comes at the expense of increasing computational cost.

3.4 Comparison to Serial with g_{fine} as Higher Order Methods

We considered using a higher order g_{fine} in our implementation, but we were unable to show higher order convergence of the error when we tested our method serially.

3.5 Parallelism by Space Paradigm

To demonstrate parallelism by space, we solve the two-dimensional wave equation by partitioning a two-dimensional space domain amongst multiple processors. Specifically, we look to numerically solve

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

on the two-dimensional domain $(x, y) \in [0, 1] \times [0, 1]$ for evolution in time. We partition the domain by space using a second order central difference approximation, assigning a rectangular portion of the two-dimensional domain to each of a grid of (P_x, P_y) processors. This requires the passing of information along the edges of the processors to each other, as shown graphically in the figure below:

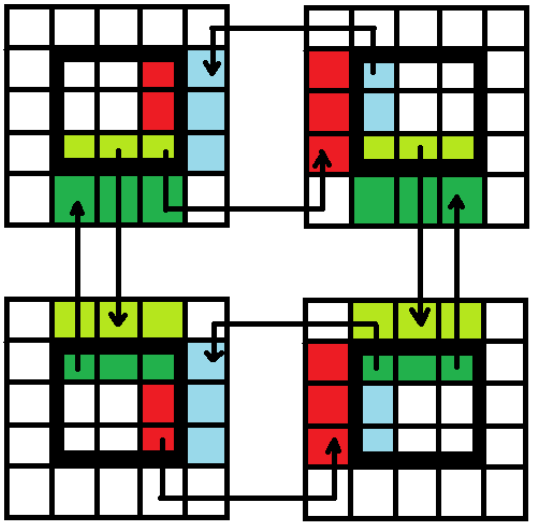

Figure 6: Parallelization by space schematic.

Note that each processor receives information from only those adjacent to it; that is, the top left processor in the graphic does not communicate directly with the bottom right processor in the graphic. Rather, information is passed only along shared boundary points to propagate the wave equation. We run this code for two different grid sizes: 256×256 and 512×512, in both serial and parallel.

In serial, we have:

(N_x, N_y)	Time (seconds)
(256, 256)	7.889
(512, 512)	63.913

In parallel, we have:

(N_x, N_y)	P_x	P_y	Time (sec)	Speedup
(256, 256)	2	2	2.57	3.07
(256, 256)	4	2	11.32	0.70
(256, 256)	4	4	(still queued on Odyssey)	
(256, 256)	8	8	(still queued on Odyssey)	
(512, 512)	2	2	39.65	1.61
(512, 512)	4	2	14.46	4.42
(512, 512)	4	4	8.02	7.97
(512, 512)	8	8	(still queued on Odyssey)	

Note that as the workload assigned to each processing element stays constant and more processors are used to solve a larger problem, near-linear scaling is achieved. However, scaling is **not** achieved for the $256\times$ 256 grid when we simply add more processing elements to try to solve the same-sized problem, as there is greater constant startup costs which do not necessarily improve the speed (or perhaps even decrease speed). We do see scaling with approximately 50% efficiency for the 512×512 grid as we add more processors. From the results, as we go from the 256×256 grid to the 512×512 grid, we quadruple the size of the problem; quadrupling the processors from 2×2 to 4×4 yields a somewhat linear relationship. Hence, this is an example of weak scaling. An advantage to this paradigm is as follows: because each processor only communicates with its nearest neighbors, communication overhead is relatively constant regardless of how large problem size / number of processors used becomes. Therefore, such a paradigm should scale well to larger problem sizes or more fine grid spacings.

3.6 Possible Optimizations and Future Work

We do not see the theoretical speedups and error convergence in our experimental tests of the Parareal algorithm. In the future, we would like to debug our implementation of the algorithm and apply it to more complicated systems of ODEs and more expensive and higher fine operators g_{fine}. We also would like to test different combinations of k and Q and see how the error converges after varying step size.

More further work could be in optimization. We are trying to measure performance in Python, which is not the ideal benchmarking language for efficiency and speedups. A port over to C++ using the MPI libraries in C++ rather than the mpi4py libraries in python would be ideal. Other optimizations mentioned above could also be implemented. To further explore the Parareal algorithm would involve many test cases and to see how the Parareal algorithm compares to other approaches—since the actual time and iterations necessary depend on the system of ODEs to solve.

Some drawbacks also include the use of the `mpi4py` library instead of the MPI libraries in C++. Because `mpi4py` is a set of Python bindings for MPI, based on the standard MPI-2 C++ bindings, there is additional overhead involved with mapping `mpi4py` commands

to the standard C++ library. For example, since Python is dynamically typed, additional information about size may first need to be sent as a part of the Python bindings, creating greater overhead as compared to the C++ libraries. In addition, C++ compilers may be more optimizable according to specific use case, as specific `-o` flags or other compiler shortcuts may be specified directly, rather than through a standard wrapper class as provided by `mpi4py`.

On the other hand, parallelism by space already requires a more problem-specific design. Parallelism by space is intuitive, and conceptually, is embarrassingly parallel and easier to create. However, the limitations are in the requirement for the code to be designed for the problem—how to divide by space so as to allow for the maximum number of even divisions. The lack of generality is not as beautiful as the Parareal algorithm but may lend itself to faster speedups.

4 Conclusion

All in all, we have implemented and begun to explore some techniques of looking for strong and weak scaling efficiencies for solving ODEs. The Parareal algorithm is beautiful in its theoretical advantages—in terms of stability, error and efficiency. However, the method, as a very generalized method, requires deeper analysis for the specific problem. The main variable will be in the difference in the coarse and fine methods (be it lower and higher order or a higher and lower resolution for the same method). The tradeoffs that must be computed and optimized for would be problem specific.

References

[1] Measuring parallel scaling performance. `https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance`, June 2013.

[2] J Lions, Yvon Maday, and Gabriel Turinici. A "parareal" in time discretization of pde's. *Comptes Rendus de l'Academie des Sciences Series I Mathematics*, 332(7):661–668, 2001.

[3] Guillaume Bal. On the convergence and the stability of the parareal algorithm to solve partial differential equations. In *Domain decomposition methods in science and engineering*, pages 425–432. Springer, 2005.

[4] Scott Field. Parareal methods. `http://www.cfm.brown.edu/people/jansh/page5/page10/page40/assets/Field_Talk.pdf`, December 2009.

[5] Gunnar Andreas Staff and Einar M Rønquist. Stability of the parareal algorithm. In *Domain decomposition methods in science and engineering*, pages 449–456. Springer, 2005.