An Investigation into Parareal

Abhijit Chowdhary

New York University

May 2019

Problem Statement

We would like to numerically solve the ordinary differential equation:

$$\begin{cases} u'(t) = f(t, u), \quad t \in [t_0, t_f] \\ u(t_0) = u_0 \end{cases}$$

where $f : \mathbb{R}^d \to \mathbb{R}^d$ and $u : \mathbb{R} \to \mathbb{R}^d$.
Framing this in a HPC context, how can we solve the above while taking advantage of massively parallel hardware?

Serial Methods

- Forward and Backward Euler:

$$u_{n+1} = u_n + hf(t, u_n), u_n + hf(t, u_{n+1})$$

- Linear Multistep Methods

$$\sum_k \alpha_k y_{n+k} = h \sum_k \beta_k f(t_{n+k}, y_{n+k})$$

- Runge Kutta Methods
- Etc.

Parallel Techniques

Problem!

Most of the previous methods are iterative, with dependency on the previously computed values.

Not embarrassingly parallel... We have to get creative.

Parallel-In-Time

One technique is to parallelize the problem across time, i.e. split $[t_0, t_f]$ into slices $[t_n, t_{n+1}]$ and in parallel solve each slice.

Iteration Dependence

But since each slice depends on the previous, how can we chain together these individual solutions to each slice?

Parareal

Predictor-Corrector

What if we were to predict the portions that $[t_n, t_{n+1}]$ are dependent on with a cheap course operator, and then correct with a fine operator!

Yes!

Parareal

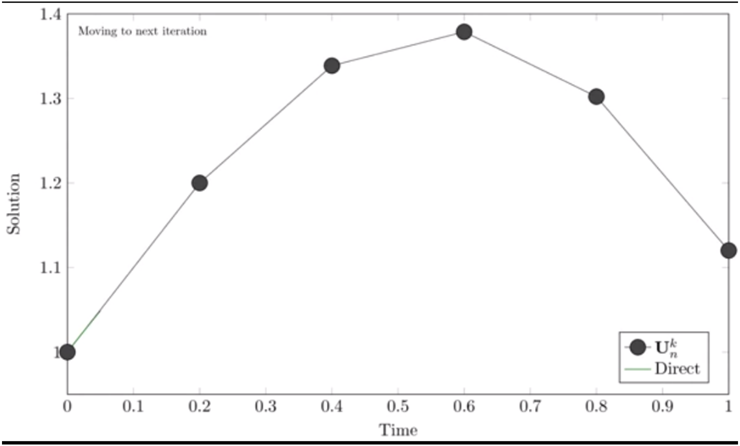
Let $\mathcal{G}$ be a course and inexpensive operator, and let $\mathcal{F}$ be a fine operator of high order. Then Parareal iteration is:

$$\lambda_{n+1}^{k+1} = \mathcal{G}(t_{n+1}, t_n, \lambda_n^{k+1}) + \left[\mathcal{F}(t_{n+1}, t_n, \lambda_n^k) - \mathcal{G}(t_{n+1}, t_n, \lambda_n^k)\right]$$

Parallel Potential

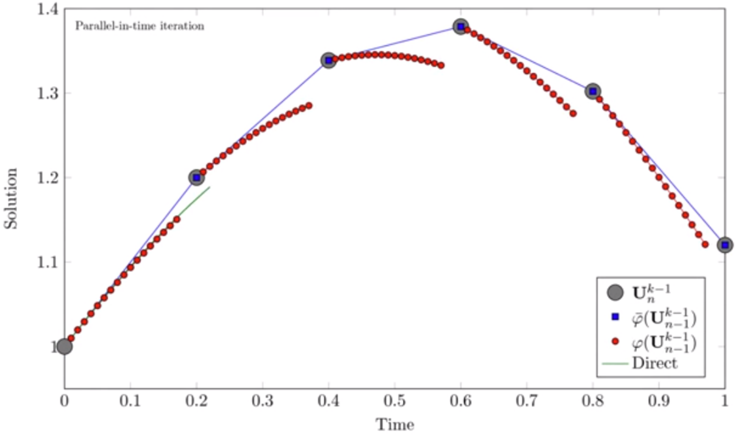
Notice! The $\mathcal{F}$ term depends only on the previous solutions, which means it can be computed in parallel for each k step. Furthermore, $\mathcal{G}(\lambda_n^k)$ satisfies the FSAL property!

Parareal: Visual Example



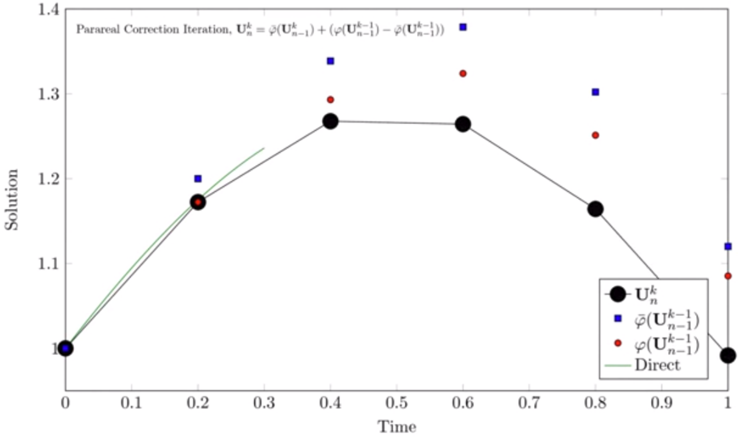
Full beautiful animation found on wikipedia's Parareal page.

Parareal: Visual Example



Full beautiful animation found on wikipedia's Parareal page.

Parareal: Visual Example



Full beautiful animation found on wikipedia's Parareal page.

Pseudocode

```
$\lambda_{n+1}^{k+1} = \mathcal{G}(t_{n+1}, t_n, \lambda_n^{k+1}) + \left[\mathcal{F}(t_{n+1}, t_n, \lambda_n^k) - \mathcal{G}(t_{n+1}, t_n, \lambda_n^k)\right]$
Require: $y_0$ and course and fine solvers $\mathcal{G}$, $\mathcal{F}$.
  $y_c \leftarrow \mathcal{G}(t_f, t_0, y_0)$.                ▷ Coarsely approximate solution
  $y \leftarrow y_c$.
  while $\mathrm{iter} < \mathrm{max\_iter}$ && not converged do
      for $n = 0 \rightarrow P$ do                                    ▷ Parallel capable
          $y_f(n) = \mathcal{F}(t_{n+1}, t_n, y(n))$.
          $\delta y(n) = y_f(n) - y_c(n)$.              ▷ corrector term. FSAL
      end for
      for $n = 0 \rightarrow P$ do
          $y_c(n) = \mathcal{G}(t_{n+1}, t_n, y(n))$.                         ▷ Predict.
          $y(n) = y_c(n) + \delta y(n)$.                                    ▷ Correct.
      end for
  end while
```

Parallel Pseudocode

```
Require: $y_0$ and course and fine solvers $\mathcal{G}$, $\mathcal{F}$.
  $y_c \leftarrow \mathcal{G}(t_f, t_0, y_0)$.                ▷ Coarsely approximate solution
  $y \leftarrow y_c$.
  while $\mathrm{iter} < \mathrm{max\_iter}$ && not converged do
      #pragma omp parallel for
      for $n = 0 \rightarrow P$ do
          $y_f(n) = \mathcal{F}(t_{n+1}, t_n, y(n))$.
          $\delta y(n) = y_f(n) - y_c(n)$.          ▷ corrector term. FSAL
      end for
      for $n = 0 \rightarrow P$ do
          $y_c(n) = \mathcal{G}(t_{n+1}, t_n, y(n))$.                  ▷ Predict.
          $y(n) = y_c(n) + \delta y(n)$.                                ▷ Correct.
      end for
  end while
```

Speedup Analysis

Suppose we have P processors, and suppose our fine method takes T_f time, T_g for the course method. Furthermore, assume that our Parareal iteration needs k steps. Then the speedup it provides is:

$$S = \frac{PT_f}{PT_g + k(PT_g + T_f)} = \frac{1}{\frac{T_g}{T_f} + k(\frac{T_g}{T_f} + \frac{1}{P})} = \frac{1}{\frac{T_g}{T_f}(1+k) + \frac{k}{P}}$$

Note:

$$S_\infty = \lim_{P \to \infty} \frac{1}{\frac{T_g}{T_f}(1+k) + \frac{k}{P}} = \frac{T_f}{T_g(1+k)}$$

No Free Lunch

$$S_\infty = \frac{T_f}{T_g(1+k)}$$

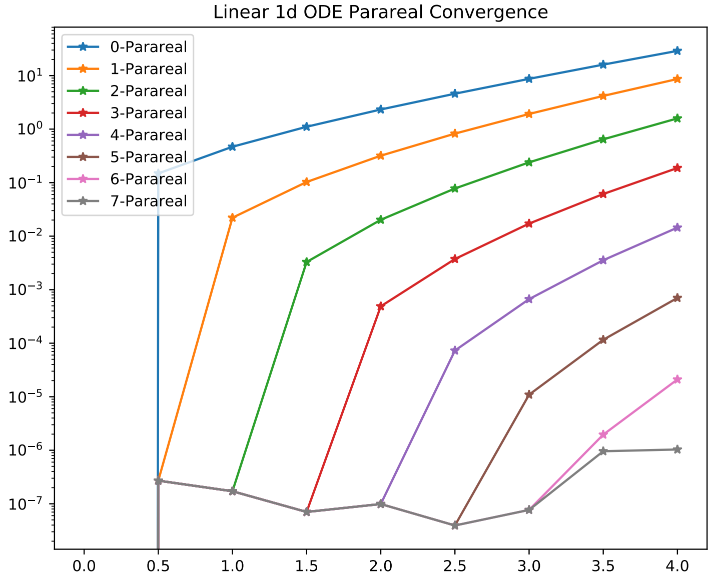
This tells us something very important, we want the ratio T_f/T_g to be as large as possible, and we want k to be as small as possible.

Theorem

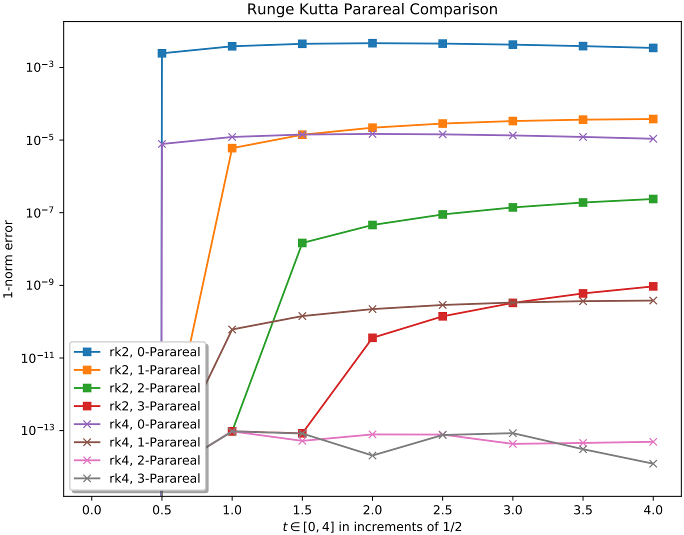
The parareal method has order of accuracy mk, where $k-1$ is the number of parareal iterations made and m is the order of $\mathcal{G}$, assuming $\mathcal{F}$ is close to truth. (Bal)

Therefore, we have this uncomfortable optimization problem. We want k small for speedup, but large for order. If we make m large instead, then the ratio T_f/T_g becomes smaller...

Forward Euler Convergence



Runge-Kutta Convergence



Improvements | Pipelined Parareal

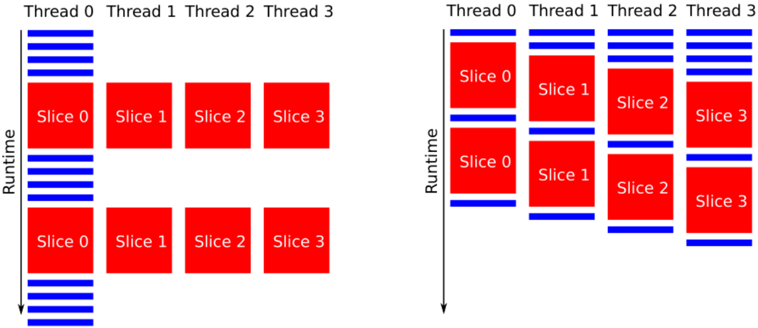


Figure borrowed from Ruprecht's *Implementing Parareal*.

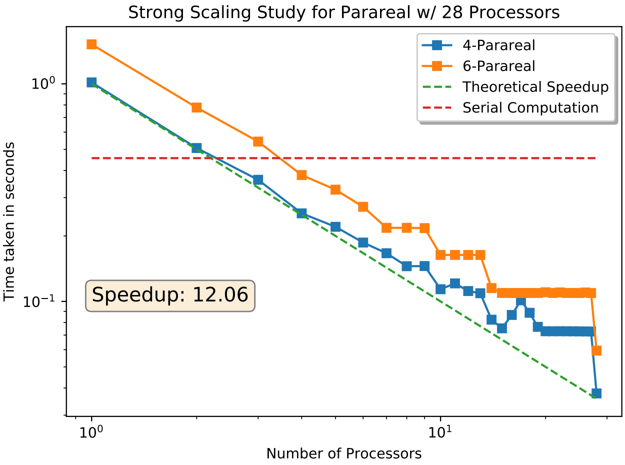
Pipelined Parareal Pseudocode

It's a bit too complicated to present in a slide, but the general idea is to begin the omp parallel statement at the beginning and then pretend like were in a MPI enviornment.

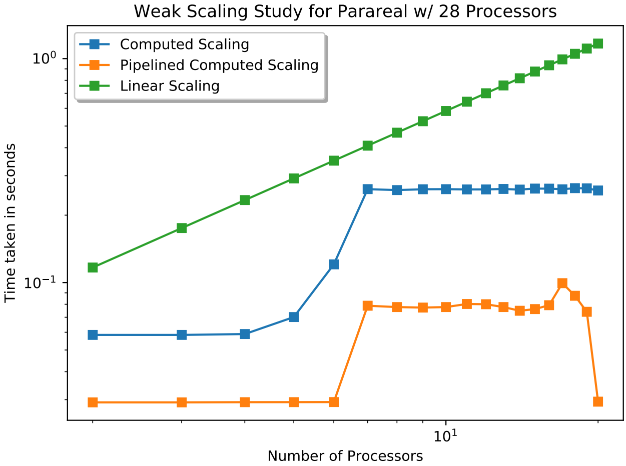
General Idea

Thread p processes the new iteration at $p + 1$. MPI communication is mimicked with locks, and done in place.

Strong Scaling



Weak Scaling



Conclusions

- We see significant speedup as well as proper weak and strong scaling of parareal, supposing ideal choices of $\mathcal{F}, \mathcal{G}$.

Conclusions

- We see significant speedup as well as proper weak and strong scaling of parareal, supposing ideal choices of $\mathcal{F}, \mathcal{G}$.
- The algorithm is very picky with choices of $\mathcal{G}$ and $\mathcal{F}$, and choices of Δt and δt vary wildly from problem to problem.

Conclusions

- We see significant speedup as well as proper weak and strong scaling of parareal, supposing ideal choices of $\mathcal{F}, \mathcal{G}$.
- The algorithm is very picky with choices of $\mathcal{G}$ and $\mathcal{F}$, and choices of Δt and δt vary wildly from problem to problem.
- To generalize to a larger scale, the code should be formulated first with MPI to distribute work across nodes, and then with OpenMP to parallelize locally.