A Parallel Regression Approach to ODE Integration: for Application to Relativistic Stellar Oscillations and Beyond

Martin Veselský[1,*]

[1]*Institute of Experimental and Applied Physics, Czech Technical University in Prague, Husova 240/5, 11000 Prague, Czech Republic*

(Dated: October 13, 2025)

A non-sequential, massively parallel approach to the numerical integration of ordinary differential equations (ODEs) is proposed and demonstrated. The method constructs a global regression representation of the dependent variable $F(y,t)$ in terms of orthogonal basis functions over the domain $(y_0-q, y_0+q)\times(t_i,t_f)$. Coefficients of this expansion are determined by minimizing the residual between the analytical time derivative $\partial F/\partial t$ and the known right-hand side $f(y,t)$. This transforms the ODE integration problem into a global optimization solvable on GPUs with large-scale parallelism. A toy model $dy/dt = -y^3 + \sin(t)$ solved using this method reproduces a reference RK4 solution with relative error of 3×10^{-4} while avoiding sequential time stepping. The method offers a new path toward fully parallel ODE solvers, potentially accelerating computation in various domains such as chemical kinetics, control, climate modeling, plasma physics, modern machine learning and nuclear physics and astrophysics, in particular concerning perturbation and oscillation calculations in relativistic stellar models.

I. INTRODUCTION

Integration of ordinary differential equations (ODEs) is a fundamental problem in computational physics. Standard solvers such as Runge–Kutta (RK) or implicit methods [1, 2]are inherently sequential, limiting the exploitation of modern massively parallel hardware such as GPUs. Parallel-in-time algorithms such as Parareal attempt partial parallelization [3], but only an approach that removes time-marching altogether enables maximal concurrency on GPUs. This limitation resulting from sequntial time-marching is particularly severe in problems where many ODE systems must be solved repeatedly, in typical domains such as chemical kinetics, control, climate modeling, plasma physics, and modern machine learning (ML) featuring Neural ODEs. In nuclear physics and astrophysics such methods are used among other in the calculation of non-radial oscillation spectra of relativistic stars [5, 6], which is a topic of interest at present, following the observation of gravitational waves [7, 8] and construction of next generation of gravitational wave observatories such as Einstein telescope [9] or LISA [10]. In the present work, we propose and test a novel approach where the ODE integration problem is reformulated as a global regression task. The resulting "parallel regression integrator" evaluates the full time-evolution map without sequential stepping and is thus naturally suited to GPU execution.

II. METHOD

For a general ODE

$$\frac{dy}{dt} = f(y,t), \qquad y(t_0) = y_0, \tag{1}$$

usually solved by stepwise integration, we assume a representation

$$F(y,t) = \sum_{i,j} a_{ij}\,\phi_i(y)\,\psi_j(t), \tag{2}$$

where $\{\phi_i\}$ and $\{\psi_j\}$ form orthogonal polynomial bases in y and t respectively. The coefficients a_{ij} are determined from the condition that

$$\frac{\partial F(y,t)}{\partial t} \approx f(y,t) \tag{3}$$

over a large set of randomly sampled points (y_k,t_k). A least-squares minimization of

$$\mathcal{L} = \sum_k \left| \frac{\partial F(y_k,t_k)}{\partial t} - f(y_k,t_k) \right|^2 \tag{4}$$

* Martin.Veselsky@cvut.cz

yields a linear system for a_{ij}, which can be solved iteratively using GPU-accelerated matrix operations. The initial condition is imposed by enforcing $F(y_0, t_0) = y_0$. Once $F(y_0, t_0)$ is obtained, the solution at any time t follows from the implicit condition $F(y, t) = y$, which is solved independently for each t.

III. RESULTS

The method was implemented in `PyTorch` framework [11] with double precision arithmetic and executed on an NVIDIA GPU. We tested the prototype on the equation $dy/dt = -y^3 + \sin(t)$, $y(0) = 0.5$, over $t \in [0, 0.1]$. A monomial $M \times N$ basis was used. In particular, five configurations were subsequently run:

1. 5×5 basis, $K = 3000$ random samples,
2. 10×10 basis, $K = 3000$ random samples,
3. 5×5 basis, $K = 50000$ random samples,
4. 5×5 basis, $K = 200000$ random samples,
5. 10×10 basis, $K = 200000$ random samples.

The regression coefficients converged within 15 Newton iterations. Figure 1 shows the RK4 reference and the three regression solutions approaching RK4 solution. Increasing basis size and sample count consistently improves agreement with RK4. This reflects the crucial point of this method which is precision of the parallel regression.

Table I compares the fixed-point regression results with a reference RK4 solver.

t	y_{RK4}	F_{reg}
0.000	0.5000	0.5000
0.010	0.4988	0.4989
0.050	0.4951	0.4955
0.090	0.4931	0.4936

TABLE I. Comparison of regression-based solver with 10×10 basis and 200000 random samples with RK4 reference for $dy/dt = -y^3 + \sin(t)$. Relative L_2 error: 3×10^{-4}.

The runtime for the regression method on GPU was 0.04 s, significantly shorter than any sequential RK integration over comparable sampling points. The approach scales trivially with batch size, enabling simultaneous solution of thousands of IVPs.

TABLE II. Convergence diagnostics for the ParODE fixed-point solver (toy ODE $\dot{y} = -y^3 + \sin t$, $y(0) = 0.5$). Shown are basis sizes $M \times N$ (number of coefficients $N_b = M \cdot N$), number of collocation samples K, initial condition enforcement weight, Newton iterations (or max iters), wall-clock time for the Newton loop, relative L_2 error vs RK4, finite-difference residual $\|dF/dt - f\|$ and the enforced IC residual.

$M \times N$	N_b	K	IC wgt	Newton iters	Time (s)	Rel. L_2 error	Residual $\|dF/dt - f\|$	IC residual
5×5	25	3×10^3	10^3	200 (max)	0.122	4.28×10^{-1}	4.55×10^{-1}	~ 0
5×5	25	5×10^4	10^3	200 (max)	0.124	1.168×10^{-1}	2.9074	~ 0
10×10	100	2×10^5	10^3	15 (conv.)	0.0396	3.28×10^{-4}	5.167×10^{-2}	2.22×10^{-16}
10×10	100	8×10^5	10^3	200 (max)	0.154	3.235×10^{-2}	3.6785	1.11×10^{-16}
15×15	225	8×10^5	10^3	15 (conv.)	0.0380	3.02×10^{-5}	3.9813×10^{-2}	1.53×10^{-13}

Table II shows convergence behavior. For this purpose the number of collocation samples was increased to 800000 for 10x10 basis and subsequently convergence for this number of samples was again achieved using 15x15 basis. For each base there appears to exist optimum set of collocation samples, where the method converges fast, while further increase of the number leads to problem with convergence, possibly due to imbalance with the number of available coefficients a_{ij}. As the Table II demonstrates, increasing the basis dimension (from 5×5 to 15×15) while finding optimum sample set dramatically improves accuracy: with 15×15 and K=8×10^5 the solver converges in 15 Newton iterations and achieves relative L_2 error 3×10^{-5}. Slow or oscillatory Newton behaviour (200 iterations, i.e. the iteration cap) indicates an under-expressive basis for the sampled region; increasing basis size or improving sampling brings fast convergence and small residuals.

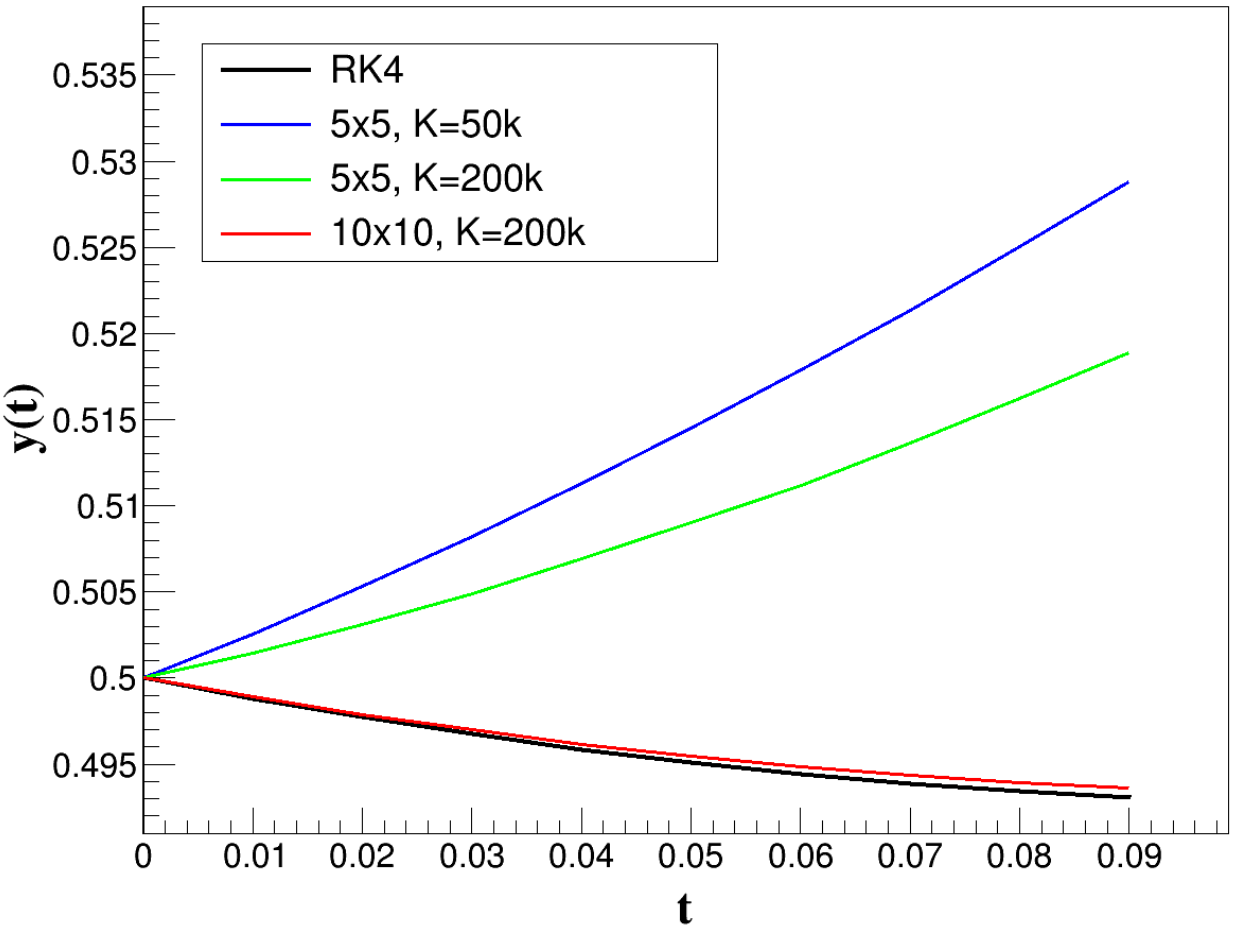

FIG. 1. Convergence: RK4 reference (black) and three regression-based solutions (colored). Larger basis and larger sample budgets progressively reduce discrepancy.

IV. APPLICATIONS

This method is particularly attractive for problems involving large ensembles of ODEs, occuring in many scientific fields as listed above. In nuclear physics and astrophysics can be mentioned :

- relativistic stellar perturbations and oscillation spectra (f-, p-, g-modes),
- slow-rotation corrections in the Hartle–Thorne framework,
- tidal deformability calculations in binary neutron stars.

By removing the sequential dependence in time, the entire system can be mapped onto GPU-accelerated tensor operations. With increasing performance of parallel hardware this method has potential to outperform and replace existing methods. Performance of the method can be further improved by using Chebyshev basis.

V. CONCLUSIONS

We presented a proof-of-concept for a global regression formulation of ODE integration, avoiding time marching and thus allowing massively parallel evaluation of dynamical systems. The method reproduces RK4 results for a simple test equation with high accuracy while leveraging full GPU parallelism. Extension to vector ODEs governing relativistic stellar perturbations

is underway. Applications in other fields such as chemical kinetics, control, climate modeling, plasma physics, modern ML are foreseen.

ACKNOWLEDGMENTS

The author used OpenAI's ChatGPT (GPT-5, 2025) to assist with language refinement, code translation, and discussion of numerical methods. All conceptual and scientific content is the author's own. Computations reported here were performed using an NVIDIA GeForce RTX 4080 SUPER.

[1] J. C. Butcher, Numerical Methods for Ordinary Differential Equations (Wiley, 2008).
[2] E. Hairer, S. P. Nørsett, and G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems (Springer, 1993).
[3] J.-L. Lions, Y. Maday, and G. Turinici, A 'parareal' in time discretization of pdes, Comptes Rendus de l'Académie des Sciences (2001).
[4] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Menon, Neural ordinary differential equations, in NeurIPS (2018).
[5] S. Detweiler and L. Lindblom, Astrophys. J. **292**, 12 (1985).
[6] C. Krüger, Ph.D. thesis, University of Southampton (2015).
[7] LIGO Scientific, Virgo Collaboration, B. P. Abbott et al., "Observation of Gravitational Waves from a Binary Black Hole Merger," Phys. Rev. Lett. 116 no. 6, (2016) 061102, arXiv:1602.03837 [gr-qc].
[8] B. P. Abbott et al., "GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral," Phys. Rev. Lett. 119 no. 16, (2017) 161101, arXiv:1710.05832 [gr-qc].
[9] M. Punturo et al., "The Einstein Telescope: A third-generation gravitational wave observatory," Class. Quant. Grav. 27 (2010) 194002.
[10] Monica Colpi et al., "LISA Definition Study Report", arXiv:2402.07571 [astro-ph.CO].
[11] Paszke, A. et al., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024–8035.

Appendix A: Appendix: Python implementation (monomial basis, M=N=10, K=200000)

The script below is the version used to produce the final (10×10, 200k) run. It uses monomial basis expansions, Monte-Carlo sampling in the domain $y \in [y_0 - \delta, y_0 + \delta]$, and parallel Newton solves for all time nodes.

```
# ParODE.py
# Parallel fixed-point solver (toy ODE) with enforced initial condition.
# dy/dt = -y^3 + sin(t), y(0)=0.5
# Fit separable polynomial basis f_{m,n}(y,t)=y^m t^n by collocation,
# add weighted constraint F(y0,t0)=y0, then solve F(y,t)-y=0 in parallel (Newton).

import torch, time, math

# ---------- config (adjustable) ----------
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
dtype = torch.float64

t0, tf = 0.0, 2.0
ntime = 201
M = 10                 # monomial degree+1 in y
N = 10                 # monomial degree+1 in t
Nb = int(M) * int(N)
K = 200000             # number of collocation samples (increase/decrease to taste)

lambda_reg = 1e-8     # Tikhonov
ic_weight = 1e3       # weight for initial-condition constraint (large -> enforces)

max_newton_iters = 200
newton_tol = 1e-10
damping = 0.8

# ---------- helpers ----------
def f_rhs(y, t):
    return -y**3 + torch.sin(t)

def rk4_integrate(f, y0, t_grid):
    y = y0.clone()
    ys = [y.clone()]
    for i in range(len(t_grid)-1):
        t = t_grid[i]
        dt = t_grid[i+1] - t
        k1 = f(y, t)
```

```
        k2 = f(y + 0.5*dt*k1, t + 0.5*dt)
        k3 = f(y + 0.5*dt*k2, t + 0.5*dt)
        k4 = f(y + dt*k3, t + dt)
        y = y + dt*(k1 + 2*k2 + 2*k3 + k4)/6.0
        ys.append(y.clone())
    return torch.stack(ys, dim=0)

# ---------- prepare ----------
t_grid = torch.linspace(t0, tf, ntime, device=device, dtype=dtype)
y0 = torch.tensor(0.5, device=device, dtype=dtype)

M_int = int(M)
N_int = int(N)
Nb = M_int * N_int

print(f"Device: {device}, dtype: {dtype}")
print(f"Basis: {M_int} x {N_int} (Nb={Nb}), samples K={K}, IC weight={ic_weight}")

# ---------- sample collocation points ----------
torch.manual_seed(0)
delta_y = 0.8
ys_samples = (y0.item() - delta_y) + 2*delta_y * torch.rand(K, device=device, dtype=dtype)
ts_samples = t0 + (tf - t0) * torch.rand(K, device=device, dtype=dtype)
s_k = f_rhs(ys_samples, ts_samples)  # (K,)

# ---------- build basis powers and derivatives ----------
Phi = torch.stack([ys_samples**m for m in range(M_int)], dim=1)   # (K,M)
Tpow = torch.stack([ts_samples**n for n in range(N_int)], dim=1)   # (K,N)

dPsi = torch.zeros((K, N_int), device=device, dtype=dtype)
if N_int > 1:
    for n in range(1, N_int):
        dPsi[:, n] = (n) * ts_samples**(n-1)

dPhi = torch.zeros((K, M_int), device=device, dtype=dtype)
if M_int > 1:
    for m in range(1, M_int):
        dPhi[:, m] = (m) * ys_samples**(m-1)

# ---------- assemble collocation matrix B (K, Nb) ----------
B = torch.zeros((K, Nb), device=device, dtype=dtype)
col = 0
for m in range(M_int):
    pm = Phi[:, m]
    dpm = dPhi[:, m]
    for n in range(N_int):
        B[:, col] = pm * dPsi[:, n] + dpm * Tpow[:, n] * s_k
        col += 1

# compute normal matrices
BTB = B.t() @ B        # (Nb, Nb)
rhs = B.t() @ s_k      # (Nb,)

# ---------- add initial-condition equation weighted ----------
# row for IC: phi(y0,t0) dot a = ???  where phi element is d/dt of basis at (y0,t0) + dphi/dy * t^n * s_k?
# Simpler: we enforce F(y0,t0) = y0 directly by adding equation sum_{m,n} a_{m,n} y0^m t0^n = y0
# Build phi_ic vector and add to normal equations with weight ic_weight.
y0_scalar = float(y0.item())
t0_scalar = float(t0)
phi_ic = torch.zeros((Nb,), device=device, dtype=dtype)
col = 0
for m in range(M_int):
    ypow = (y0_scalar**m)
    for n in range(N_int):
        tpow = (t0_scalar**n)
        phi_ic[col] = ypow * tpow
        col += 1

# Add weighted normal equations: (BTB + w phi phi^T) a = rhs + w phi * y0
BTB = BTB + ic_weight * (phi_ic.unsqueeze(1) @ phi_ic.unsqueeze(0))
rhs = rhs + ic_weight * phi_ic * y0_scalar

# regularize and solve
I = torch.eye(Nb, device=device, dtype=dtype)
a = torch.linalg.solve(BTB + lambda_reg * I, rhs)  # (Nb,)
A = a.view(M_int, N_int)

# ---------- diagnostics for IC enforcement ----------
F_y0_t0 = (phi_ic @ a).item()
print("Enforced F(y0,t0) (should equal y0):", F_y0_t0, " target y0 =", y0_scalar)
ic_residual = abs(F_y0_t0 - y0_scalar)
print("IC residual:", ic_residual)
```

```
# ---------- vectorized eval of F and dFdy ----------
def F_and_dFdy_vec(y_vec, t_vec, A):
    L = y_vec.shape[0]
    y_pows = torch.stack([y_vec**m for m in range(M_int)], dim=1)  # (L,M)
    t_pows = torch.stack([t_vec**n for n in range(N_int)], dim=1)  # (L,N)
    YT = y_pows.unsqueeze(2) * t_pows.unsqueeze(1)                 # (L,M,N)
    F = (YT * A.unsqueeze(0)).sum(dim=(1,2))                        # (L,)
    if M_int > 1:
        y_pows_m1 = torch.stack([y_vec**(m-1) if m>=1 else torch.zeros_like(y_vec)
                                 for m in range(M_int)], dim=1)
        m_idx = torch.arange(M_int, device=A.device, dtype=A.dtype).unsqueeze(1)
        mA = (m_idx * A)
        dF = (y_pows_m1.unsqueeze(2) * t_pows.unsqueeze(1) * mA.unsqueeze(0)).sum(dim=(1,2))
    else:
        dF = torch.zeros_like(y_vec)
    return F, dF

# ---------- fully parallel Newton ----------
L = ntime
t_vec = t_grid
y_batch = torch.full((L,), y0.item(), device=device, dtype=dtype)

start = time.time()
for it in range(max_newton_iters):
    Fv, dFdy = F_and_dFdy_vec(y_batch, t_vec, A)
    G = Fv - y_batch
    dG = dFdy - 1.0
    small_mask = dG.abs() < 1e-12
    denom = dG.clone()
    denom[small_mask] = torch.sign(denom[small_mask]) * 1e-12 + 1e-12
    delta = G / denom
    y_new = y_batch - damping * delta
    max_change = torch.max(torch.abs(y_new - y_batch)).item()
    rms_resid = torch.sqrt(torch.mean(G**2)).item()
    y_batch = y_new
    if (it % 10) == 0 or it == 0:
        print(f"Iter {it:3d}: max_change={max_change:.3e}, rms_resid={rms_resid:.3e}")
    if max_change < newton_tol:
        print(f"Converged at iter {it}, max_change={max_change:.3e}, rms_resid={rms_resid:.3e}")
        break
end = time.time()
print("Newton loop time:", end - start)

# ---------- compare with RK4 ----------
y_ref = rk4_integrate(lambda y,t: f_rhs(y,t), y0.to(device=device), t_grid)

rel_err = torch.norm(y_ref - y_batch) / torch.norm(y_ref)
dt = t_grid[1] - t_grid[0]
dydt_fd = torch.empty_like(y_batch)
dydt_fd[:-1] = (y_batch[1:] - y_batch[:-1]) / dt
dydt_fd[-1] = dydt_fd[-2]
residual = torch.norm(dydt_fd - f_rhs(y_batch, t_grid))

print("\nDiagnostics after IC enforcement:")
print("Relative L2 error vs RK4:", rel_err.item())
print("Residual (FD) norm ||d/dt F - f||:", residual.item())
print("IC residual after solve (F(y0,t0)-y0):", ( (phi_ic @ a) - y0_scalar ).abs().item())

# print a few values
print("\n t    RK4_ref       F_fixedpt")
for i in range(0, min(10, L)):
    print(f"{t_grid[i].item():6.3f} {y_ref[i].item():12.6e} {y_batch[i].item():12.6e}")
```