Tailoring exponential integrators for computational efficiency

John Loffeld, Mayya Tokman

University of California at Merced

May 14, 2012

Implementation has a significant impact on CPU efficiency of an integrator.

Basic idea

- Choose coefficients not just for order, but also implementation

Outline

- Description and implementation of EPIRK methods
- Krylov adaptivity friendly methods
- Brief description of work so far on parallel implementation

General form of EPIRK methods

For an s-stage method:

$$Y_i = y_0 + a_{i1}\psi_{i1}(g_{i1}hJ)hf(y_0) + \sum_{j=2}^{i-1} a_{ij}\psi_{ij}(g_{ij}hJ)h\Delta^{j-1}r(y_0)$$

$$i = 1, ..., (s-1)$$

$$y(x_0 + h) = y_0 + b_1\psi_{s1}(g_{s1}hJ)hf(y_0) + \sum_{j=2}^{s} b_j\psi_{sj}(g_{sj}hJ)h\Delta^{j-1}r(y_0),$$

General form of EPIRK methods

For an s-stage method:

$$Y_i = y_0 + a_{i1}\psi_{i1}(g_{i1}hJ)hf(y_0) + \sum_{j=2}^{i-1} a_{ij}\psi_{ij}(g_{ij}hJ)h\Delta^{j-1}r(y_0)$$

$$i = 1, ..., (s-1)$$

$$y(x_0 + h) = y_0 + b_1\psi_{s1}(g_{s1}hJ)hf(y_0) + \sum_{j=2}^{s} b_j\psi_{sj}(g_{sj}hJ)h\Delta^{j-1}r(y_0),$$

$$\psi_{ij}(z) = \sum_{k=1}^{s} p_{ijk}\varphi_k(z), \quad \varphi_0(z) = e^z, \quad \varphi_k(z) = z\varphi_{k+1}(z) + \frac{1}{k!}$$

General form of EPIRK methods

For an s-stage method:

$$Y_i = y_0 + a_{i1}\psi_{i1}(g_{i1}hJ)hf(y_0) + \sum_{j=2}^{i-1} a_{ij}\psi_{ij}(g_{ij}hJ)h\Delta^{j-1}r(y_0)$$

$$i = 1, ..., (s-1)$$

$$y(x_0 + h) = y_0 + b_1\psi_{s1}(g_{s1}hJ)hf(y_0) + \sum_{j=2}^{s} b_j\psi_{sj}(g_{sj}hJ)h\Delta^{j-1}r(y_0),$$

$\psi_{ij}(z) = \sum_{k=1}^{s} p_{ijk}\varphi_k(z), \quad \varphi_0(z) = e^z, \quad \varphi_k(z) = z\varphi_{k+1}(z) + \frac{1}{k!}$

$r(y) = f(y) - f(y_0) - f'(y_0)(y - y_0),$

General form of EPIRK methods

For an s-stage method:

$$Y_i = y_0 + a_{i1}\psi_{i1}(g_{i1}hJ)hf(y_0) + \sum_{j=2}^{i-1} a_{ij}\psi_{ij}(g_{ij}hJ)h\Delta^{j-1}r(y_0)$$

$$i = 1, ..., (s-1)$$

$$y(x_0 + h) = y_0 + b_1\psi_{s1}(g_{s1}hJ)hf(y_0) + \sum_{j=2}^{s} b_j\psi_{sj}(g_{sj}hJ)h\Delta^{j-1}r(y_0),$$

$\psi_{ij}(z) = \sum_{k=1}^{s} p_{ijk}\varphi_k(z)$, $\varphi_0(z) = e^z$, $\varphi_k(z) = z\varphi_{k+1}(z) + \frac{1}{k!}$

$r(y) = f(y) - f(y_0) - f'(y_0)(y - y_0)$,

- *The $\psi(ghJ)v$ terms are estimated using Krylov subspace projection.*
  - g coefficients are useful for controlling Krylov cost

General form of EPIRK methods

For an s-stage method:

$$Y_i = y_0 + a_{i1}\psi_{i1}(g_{i1}hJ)hf(y_0) + \sum_{j=2}^{i-1} a_{ij}\psi_{ij}(g_{ij}hJ)h\Delta^{j-1}r(y_0)$$

$$i = 1, ..., (s-1)$$

$$y(x_0 + h) = y_0 + b_1\psi_{s1}(g_{s1}hJ)hf(y_0) + \sum_{j=2}^{s} b_j\psi_{sj}(g_{sj}hJ)h\Delta^{j-1}r(y_0),$$

$\psi_{ij}(z) = \sum_{k=1}^{s} p_{ijk}\varphi_k(z)$, $\varphi_0(z) = e^z$, $\varphi_k(z) = z\varphi_{k+1}(z) + \frac{1}{k!}$

$r(y) = f(y) - f(y_0) - f'(y_0)(y - y_0)$,

- *The $\psi(ghJ)v$ terms are estimated using Krylov subspace projection.*
  - g coefficients are useful for controlling Krylov cost
- Derive order conditions using B-series
  - Mathematica routines based on Butcher trees

Brief overview of Krylov approximation of $f(A)v$

A is $N \times N$. Project $f(A)v$ onto the Krylov subspace

$$K_m = span\{v, Av, A^2v, ..., A^{m-1}v\}$$

Use the Arnoldi iteration to generate

- orthonormalized Krylov basis V_m
- $H_m = V_m^T A V_m$

$f(A)v$ is approximated as

$$\begin{aligned} f(A)v &\approx VV^T f(A)v \\ &= VV^T f(A) VV^T v \\ &\approx Vf(H)V^T v \\ &= ||v||_2 Vf(H)e_1 \end{aligned}$$

Reducing CPU cost

Krylov projections are the greatest CPU cost.

Therefore, to reduce cost

- Minimize the # of Krylov projections per step
- Minimize the # of Krylov vectors per projection

Minimizing # of Krylov projections

The H matrix is scale-invariant, i.e.

$$\lambda H = V^T \lambda A V$$

Therefore,

$$f(\lambda A)v \approx ||v||_2 V f(\lambda H) e_1$$

EPIRK4:

$$Y_1 = y_0 + a_{11} h\varphi_1(\frac{1}{3}hJ)f(y_0)$$

$$Y_2 = y_0 + a_{21} h\varphi_1(\frac{2}{3}hJ)f(y_0) + a_{22} h\varphi_2(\frac{2}{3}hJ)r(Y_1)$$

$$y_1 = y_0 + h\varphi_1(hJ)f(y_0) + b_1 h\varphi_2(hJ)r(Y_1) + b_2 h[6\varphi_3(hJ) - \varphi_2(hJ)](-2r(Y_1) + r(Y_2))$$

- *Three Krylov projections instead of six*
- Basis sizes generally get smaller as move from blue to red

Minimizing # of Krylov vectors

- Scaling the Jacobian affects convergence
- Choice of Ψ-function affects convergence

2D Advection-Diffusion-Reaction problem

$$u_t = \frac{1}{100}(u_{xx} + u_{yy}) + 10(u_x + u_y) + 100u(u - \frac{1}{2})(1 - u)$$

with homogeneous Neumann boundary conditions and initial conditions

$$u_0 = 256(xy(1-x)(1-y))^2 + 0.3$$

h	$\varphi_1(hJ)f(y_0)$	$\varphi_2(hJ)f(y_0)$	$\varphi_3(hJ)f(y_0)$
0.01	49	46	42
0.005	30	27	24
0.0025	19	17	15

Example method

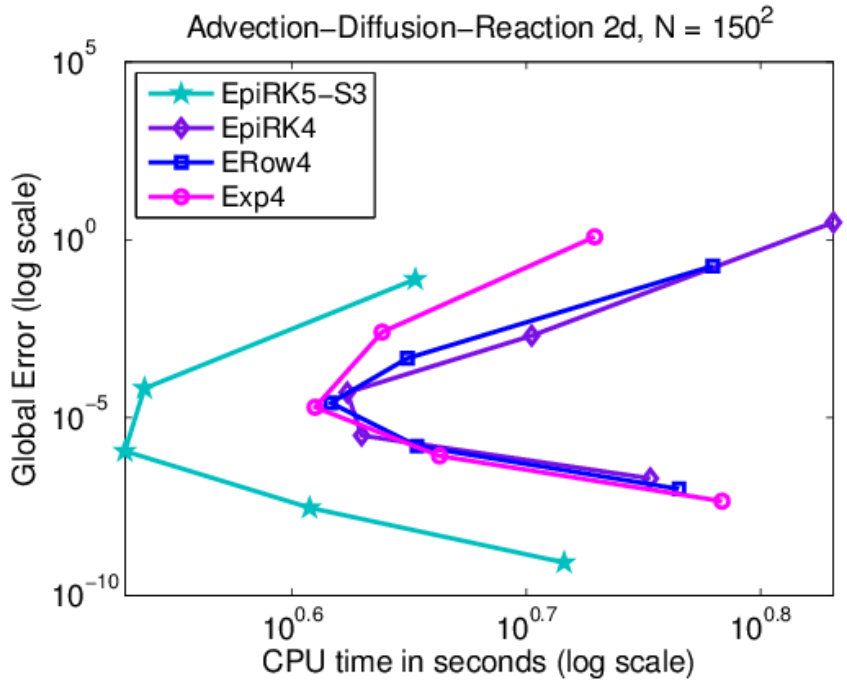
- Seek high order for low # of projections
- Minimize magnitude of g coefficients
  - *Specifically, the maximum g coefficient of each projection*

Example - EpiRK5S3

$$\begin{bmatrix} g_{11} & & \\ g_{21} & g_{22} & \\ g_{31} & g_{32} & g_{33} \end{bmatrix} = \begin{bmatrix} 0.42 & & \\ 0.86 & 0.5 & \\ 1.0 & 0.73 & 0.33 \end{bmatrix}$$

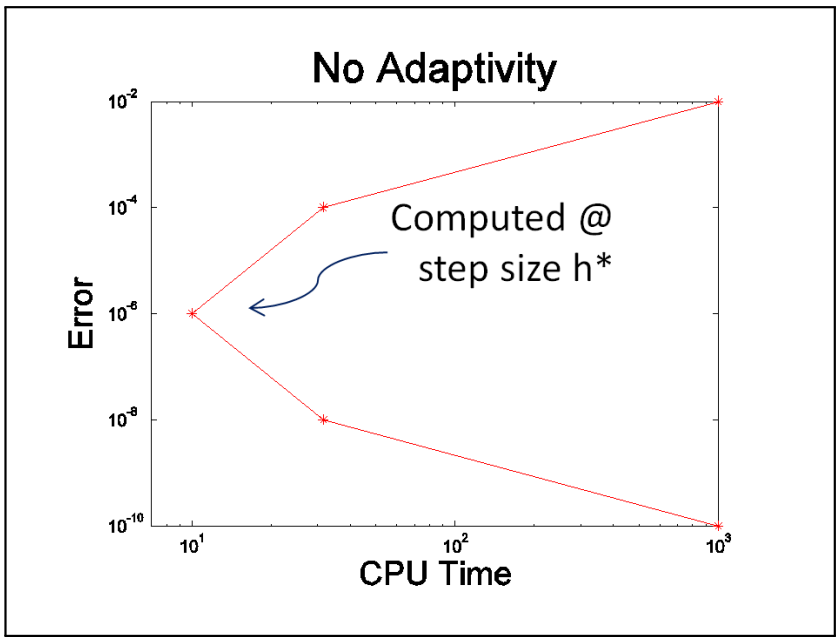
- *Classical* order five

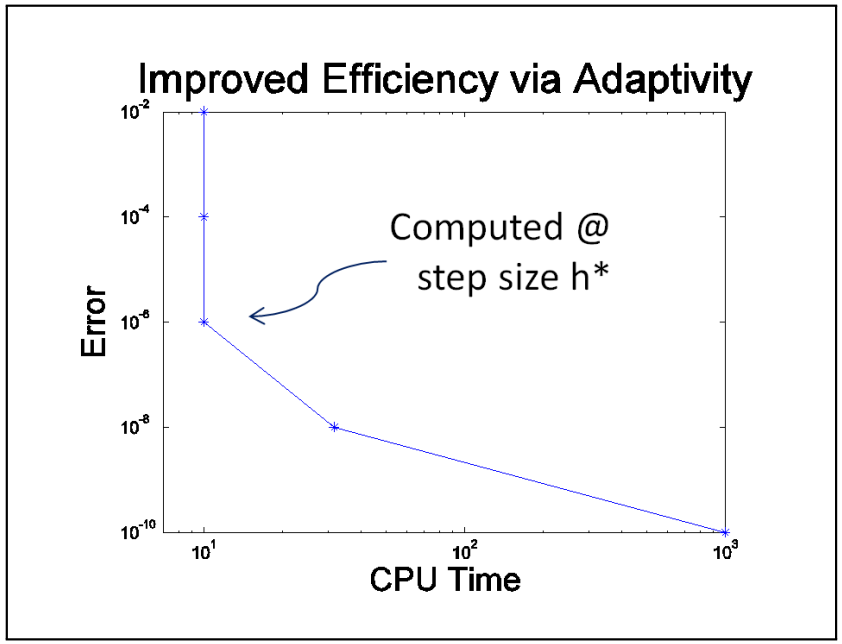
Numerical comparison



	Proj. 1	Proj. 2	Proj. 3	time	% diff
h = 0.02:					
EPIRK5S3	50	36	22	3.8	
EPIRK4	51	45	45	5.5	43%
h = 0.005:					
EPIRK5S3	30	22	14	3.6	
EPIRK4	30	26	25	4.5	23%
h = 0.00125:					
EPIRK5S3	13	9	7	5.6	
EPIRK4	13	10	10	5.9	6%

The "C"





- At step size $h*$, CPU cost is minimum

Two approaches to dealing with the "C"

- Keep step size below $h*$
  - More RHS and Jacobian calculations
  - Can't scale cost specifically to each $\varphi(hJ)v$ term

- Subdivide the projected term

$$\varphi_0(A)v = e^A v = \underbrace{e^B e^B \ldots e^B v}_{\text{n times}}, \quad B = \frac{1}{n}A$$

  - n chosen to balance # projections vs. basis size
  - Fewer RHS and Jacobian calculations

Extending to φ_k

$u(t) = \varphi_0(tA)v_0 + t\varphi_1(tA)v_1 + ... + t^p\varphi_p(tA)v_p$ can be iteratively computed

$$u(t_{k+1}) = \tau_k^p \varphi_p(\tau_k A)w_p + \sum_{j=0}^{p-1} \frac{\tau_k^j}{j!} w_j, \quad t_{k+1} = t_k + \tau_k,$$

where

$$w_j = A^j u(t_k) + \sum_{i=1}^{j} A^{j-i} \sum_{l=0}^{j-i} \frac{t_k^l}{l!} v_{i+l}, \quad j = 0, 1, ..., p$$

At $t = 1$

$$u(1) = \varphi_0(A)v_0 + \varphi_1(A)v_1 + ... + \varphi_p(A)v_p$$

- τ_k analogous to $1/n$
- Sofroniou and Spaletta 06, Niesen and Wright 11

"Horizontal" vs. "vertical" evaluation

EPIRK4:

$$Y_1 = y_0 + a_{11} h\varphi_1(\frac{1}{3}hJ)f(y_0)$$

$$Y_2 = y_0 + a_{21} h\varphi_1(\frac{2}{3}hJ)f(y_0) + a_{22} h\varphi_2(\frac{2}{3}hJ)r(Y_1)$$

$$y_1 = y_0 + h\varphi_1(hJ)f(y_0) + b_1 h\varphi_2(hJ)r(Y_1) + b_2 h[6\varphi_3(hJ) - \varphi_2(hJ)](-2r(Y_1) + r(Y_2))$$

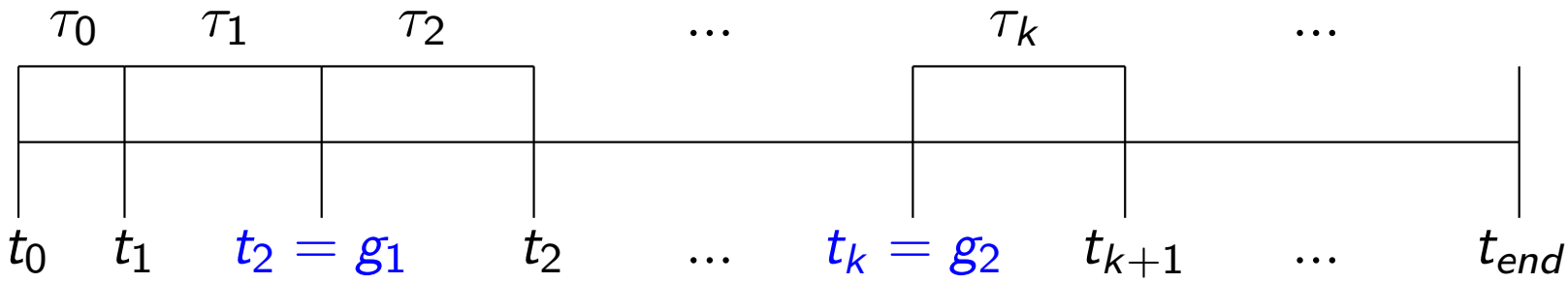
For "horizontal" (stage-wise)

- All g coefficients must be the same in a stage
- Good if early stages have small g coefficients
- More polynomial terms in the iteration

For "vertical" (per-vector)

- Must compute at interim $u(t_k = g_{ij})$
- Exposes falloff in basis size as go from blue to red

The "vertical" approach



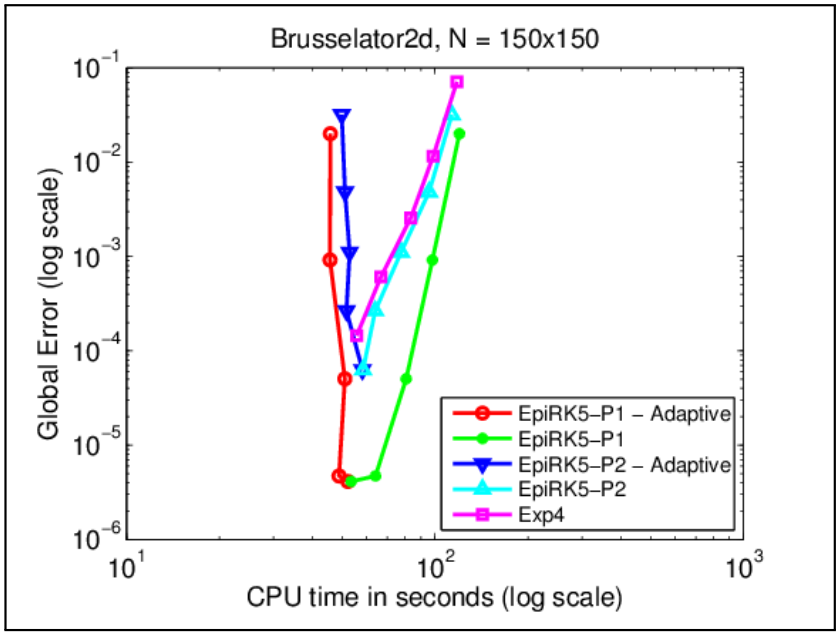
Since $u(t) = \varphi_0(tA)v_0 + t\varphi_1(tA)v_1 + ... + t^p\varphi_p(tA)v_p$

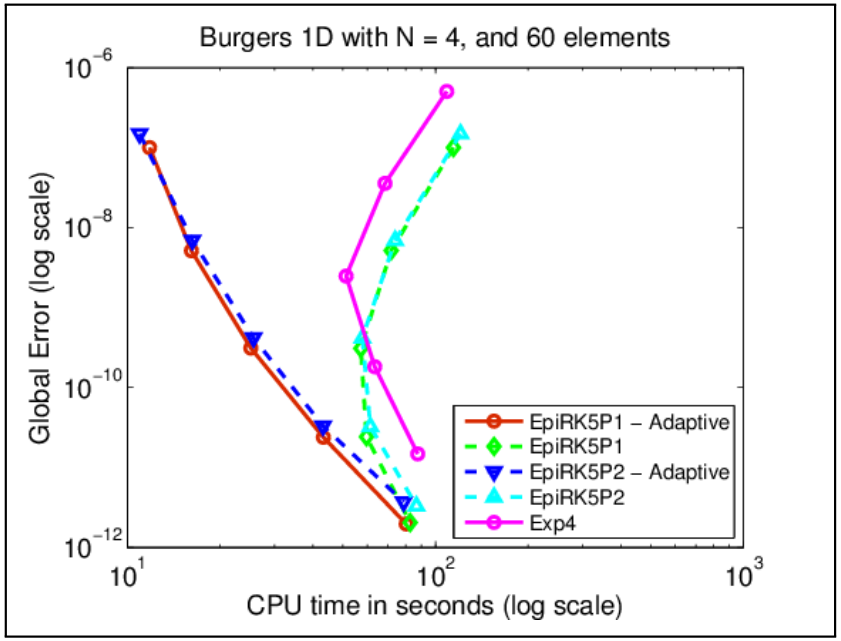
$$u(t_k = g_{ij}) = g_{ij}^k \varphi_k(g_{ij}A)v_k$$
$$\varphi_k(g_{ij}A)v_k = u(g_{ij})/g_{ij}^k \qquad \text{Ok!}$$

$$u(t_k = g_{ij}) = g_{ij}^k \varphi_k(g_{ij}A)v_k + g_{ij}^l \varphi_l(g_{ij}A)v_l$$
$$\varphi_k(g_{ij}A)v_k + \varphi_l(g_{ij}A)v_l = \; ???$$

Two example methods

Two fifth-order adaptive methods were built with the "vertical" approach - "EPIRK5P1" and "EPIRK5P2"





- Adaptivity provides significant savings

Adaptivity - future work

Current choice of τ is suboptimal

- Current error estimator derived from Taylor polynomials
  - Though a free parameter is fit during iteration according to actual error
- More accurate error estimator would improve efficiency
- Use polynomial approximation instead of Krylov?
  - Cost scales linearly with iteration instead of quadratically

Parallelization

- Building SUNDIALS-compatible serial and parallel C++ implementation of EPIRK methods
  - Problems written for SUNDIALS can be run on our integrator
- Compatibility with SUNDIALS forces vector-based parallelization
  - Good scalability
  - Poor arithmetic density
- Three basic operations
  - Linear sum - no communication!
  - $f(y)$ and Jv functions - problem author's concern
  - Dot products - implemented using mpi_allreduce() - high communication!

Dot products

Dot products in the Arnoldi iteration are the main communication cost.

- Modified Gram-Schmidt is more stable but requires k reductions for the $k+1$-th Arnoldi iteration
- Classical Gram-Schmidt is less stable but can be done with one reduction-tree per Arnoldi iteration
  - *Big scalability improvement*
  - Deal with impending loss of orthogonality by restarting
    - On exponential integrators, this has low penalty
- For difficult problems, resort to Householder orthogonalization
  - Very numerically robust
  - No dot products, scales well
  - High flop count, 2-3x slower than Gram-Schmidt

Thank you!