FASTMATH

SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers

Carol S. Woodward

Lawrence Livermore National Laboratory
P. O. Box 808
Livermore, CA 94551

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC
LLNL-PRES-641695

Argonne NATIONAL LABORATORY · BERKELEY LAB · Sandia National Laboratories · Berkeley · Rensselaer · SMU · UBC THE UNIVERSITY OF BRITISH COLUMBIA

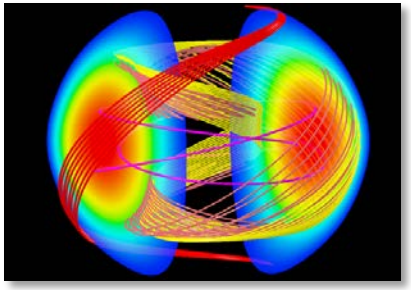
SUite of Nonlinear and DIfferential-ALgebraic Solvers

- Suite of time integrators and nonlinear solvers
  - ODE and DAE time integrators with forward & adjoint sensitivity integration
    - *Adaptive in time step and (for the multistep codes) order*
  - Newton and fixed point nonlinear solvers
  - Written in ANSI C with Fortran interfaces
  - Designed to be easily incorporated into existing codes
  - Modular implementation with swappable components
    - Linear solvers – direct dense/band/sparse, iterative
    - Vector structures (core data structure for all packages) – supplied with serial, threaded, and MPI parallel
- Freely available, released under BSD license

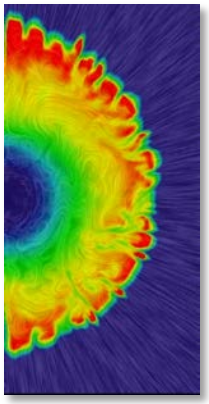
https://computation.llnl.gov/casc/sundials/main.html

SUNDIALS has been used worldwide in applications from research and industry

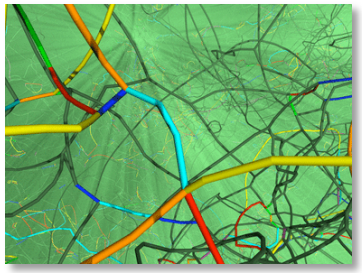
- Power grid modeling (RTE France, LLNL, ISU)
- Simulation of clutches and power train parts (LuK GmbH & Co.)
- Magnetism at the nanoscale (Magpar, Nmag)
- 3D parallel fusion (SMU, U. York, LLNL)
- Spacecraft trajectory simulations (NASA)
- Dislocation dynamics (LLNL)
- Combustion and reacting flows (Cantera)
- Large-scale subsurface flows (CO Mines, LLNL)
- 3D battery simulation (ORNL - AMPERE)
- Computational modeling of neurons (NEURON)
- Micromagnetic simulations (U. Southampton)
- Released in third party packages:
  - Red Hat Extra Packages for Enterprise Linux (EPEL) [Old versions in Debian and Ubuntu]
  - SciPy – python wrap of SUNDIALS
  - Cray Third Party Software Library (TPSL)



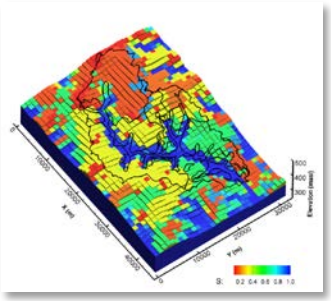
Magnetic reconnection



Core collapse supernova



Dislocation dynamics



Subsurface flow

Over 4,500 downloads / year

CVODE solves $\dot{y} = f(t, y)$

- Variable order and variable step size Linear Multistep Methods

$$\sum_{j=0}^{K_1} \alpha_{n,j} y_{n-j} + \Delta t_n \sum_{j=0}^{K_2} \beta_{n,j} \dot{y}_{n-j} = 0$$

- Nonstiff: Adams-Moulton; $K_1 = 1$, $K_2 = k$, $k = 1,\ldots,12$
- Stiff: Backward Differentiation Formulas [BDF]; $K_1 = k$, $K_2 = 0$, $k = 1,\ldots,5$
- Optional stability limit detection based on linear analysis
- The stiff solvers execute a predictor-corrector scheme:

Explicit predictor to give $y_{n(0)}$

$$y_{n(0)} = \sum_{j=1}^{q} \alpha_j^p y_{n-j} + \Delta t \beta_1^p \dot{y}_{n-1}$$

Implicit corrector with $y_{n(0)}$ as initial iterate

$$y_n = \sum_{j=1}^{q} \alpha_j y_{n-j} + \Delta t \beta_0 f_n(y_n)$$

Convergence and errors are measured against user-specified tolerances

- User-defined tolerances:
  - Absolute tolerance on each solution component, $ATOL^i$
  - Relative tolerance for all solution components, RTOL
- Norm calculations are weighted by:

$$ewt^i = \frac{1}{RTOL|y^i| + ATOL^i}$$

- Errors are measured with a weighted root-mean-square norm:

$$\|y\|_{WRMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(ewt^i \cdot y^i)^2}$$

- Choose time steps to bound an estimate of the local truncation error

Time steps are chosen to minimize local truncation error and maximize efficiency

- Time step selection criteria:
  - Estimate the error: $E(\Delta t) = C(y_n - y_{n(0)})$
    - Accept step if $||E(\Delta t)||_{WRMS} < 1$
    - Reject step otherwise
  - Estimate error at the next step, $\Delta t'$, as (q is current method order)

$$E(\Delta t') \approx (\Delta t'/\Delta t)^{q+1} E(\Delta t)$$

  - Choose next step so that $||E(\Delta t')||_{WRMS} < 1$
- Method order selection criteria:
  - Estimate error and prospective steps for orders {q, q-1, q+1}
  - Choose order resulting in largest time step meeting error condition

ARKode solves IVPs $M\dot{y} = f_E(t,y) + f_I(t,y)$

- Split system into stiff, f_I, and nonstiff, f_E, components
- M may be the identity or any nonsingular mass matrix (e.g. FEM)
- Variable step size additive Runge-Kutta Methods – combine explicit (ERK) and diagonally implicit (DIRK) RK methods to enable ImEx solver (disable either for pure explicit/implicit). Let $t_{n,j} = t_{n-1} + c_j \Delta t_n$:

$$Mz_i = My_{n-1} + h_n \sum_{j=0}^{i-1} A^E_{i,j} f_E(t_{n-1} + c_j h_n, z_j) + h_n \sum_{j=0}^{i} A^I_{i,j} f_I(t_{n-1} + c_j h_n, z_j),$$

$$My_n = My_{n-1} + h_n \sum_{i=0}^{s} b_i \left(f_E(t_{n-1} + c_i h_n, z_i) + f_I(t_{n-1} + c_i h_n, z_i) \right),$$

$$M\tilde{y}_n = My_{n-1} + h_n \sum_{i=0}^{s} \tilde{b}_i \left(f_E(t_{n-1} + c_i h_n, z_i) + f_I(t_{n-1} + c_i h_n, z_i) \right).$$

- Solve for stage solutions, $z_i = 1, \ldots, s$, sequentially (via Newton, fixed-point, linear solve, or just vector updates)

ARKode is the newest package in SUNDIALS

- Time-evolved solution, y_n embedded solution, $\tilde{y}_n$
- Error estimate: $E(\Delta t_n) = \|y_n - \tilde{y}_n\|_{WRMS}$
- Fixed order within each solve:
  - ARK : $O(\Delta t^3) \rightarrow O(\Delta t^5)$
  - DIRK: $O(\Delta t^2) \rightarrow O(\Delta t^5)$
  - ERK: $O(\Delta t^2) \rightarrow O(\Delta t^6)$
  - user-supplied
- Multistage embedded methods (as opposed to multistep):
  - High order without solution history (enables spatial adaptivity)
  - Sharp estimates of solution error even for stiff problems
  - But, DIRK/ARK require multiple implicit solves per step
- User interface modeled on CVODE -> simple transition between packages

Initial value problems (IVPs) come in the form of ODEs and DAEs

- The general form of an IVP is given by

$$F(t, y, \dot{y}) = 0$$
$$y(t_0) = y_0$$

- If $\partial F / \partial \dot{y}$ is invertible, we solve for $\dot{y}$ to obtain an ordinary differential equation (ODE), but this is not always the best approach
- Else, the IVP is a differential algebraic equation (DAE)

- A DAE has *differentiation index i* if *i* is the minimal number of analytical differentiations needed to extract an explicit ODE

IDA solves $F(t, y, \dot{y}) = 0$

- Variable order and step size BDF (no Adams-Moulton)
  - Generally assume DAEs are more stiff than ODEs
- Originally, C rewrite of DASPK [Brown, Hindmarsh, Petzold]
- Targets: implicit ODEs, index-1 DAEs, and Hessenberg index-2 DAEs
- Optional routine solves for consistent values of y_0 and $\dot{y}_0$
  - Semi-explicit index-1 DAEs
  - differential components known, algebraic unknown OR
  - all of $\dot{y}_0$ specified, y_0 unknown
- Nonlinear systems solved by Newton-Krylov method
- Optional constraints: $y^i > 0$, $y^i < 0$, $y^i \geq 0$, $y^i \leq 0$

Implicit solutions result in nonlinear systems at each time step

- Use predicted value as the initial iterate for the nonlinear solver
- Nonstiff systems: Functional iteration

$$y_{n(m+1)} = \beta_0 \Delta t_n f(y_{n(m)}) + \sum_{i=1}^{q} \alpha_{n,i} y_{n-i}$$

- Stiff systems: Newton iteration

$$M\left(y_{n(m+1)} - y_{n(m)}\right) = -G\left(y_{n(m)}\right)$$

ODE $\dot{y} = f(y)$

$M \approx I - \gamma \partial f / \partial y \qquad \gamma = \beta_0 \Delta t_n$

$$G(y_n) \equiv y_n - \beta_0 \Delta t_n f(t, y_n) - \sum_{i=1}^{k} \alpha_{n,i} y_{n-i} = 0$$

DAE $F(\dot{y}, y) = 0$

$M \approx \partial F / \partial y + \gamma \partial F / \partial \dot{y} \qquad \gamma = 1/(\beta_0 \Delta t_n)$

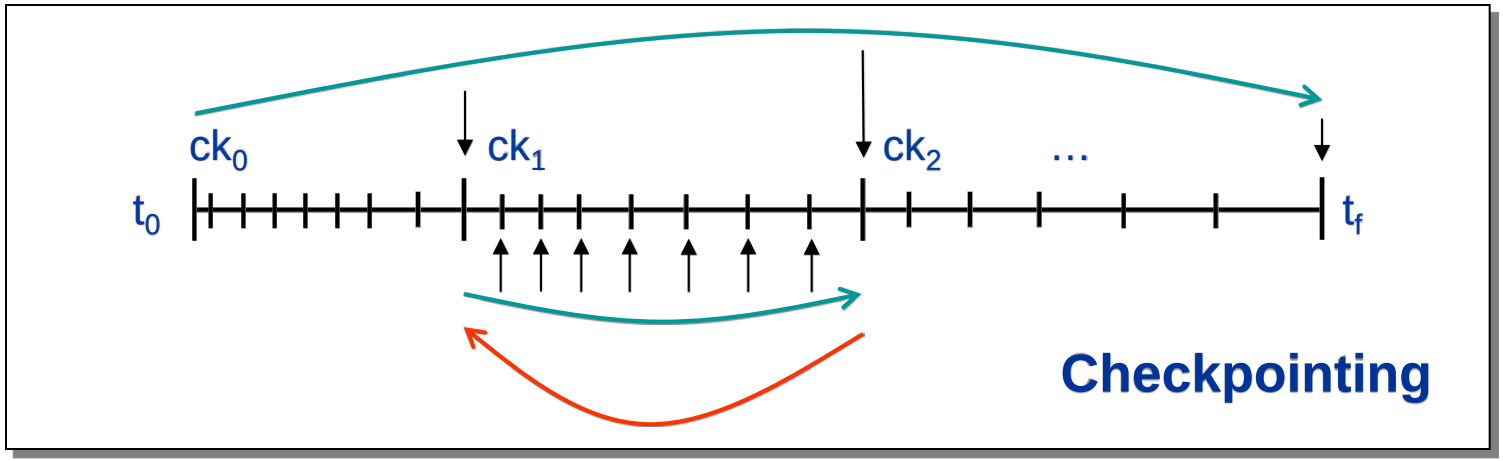
$$G(y_n) \equiv F\left(t, (\beta_0 \Delta t_n)^{-1} \sum_{i=1}^{k} \alpha_{n,i} y_{n-i}, y_n\right) = 0$$

Sensitivity Analysis: CVODES and IDAS

- Sensitivity Analysis (SA) is the study of how the variation in the output of a model (numerical or otherwise) can be apportioned, qualitatively or quantitatively, to different sources of variation in inputs.
- Applications:
  - Model evaluation (most and/or least influential parameters)
  - Model reduction
  - Data assimilation
  - Uncertainty quantification
  - Optimization (parameter estimation, design optimization, optimal control, …)
- Approaches:
  - Forward SA– augment state system with sensitivity equations
  - Adjoint SA– solve a backward in time adjoint problem (user supplies the adjoint problem)

Adjoint Sensitivity Analysis Implementation

- Solution of the forward problem is required for the adjoint problem → need predictable and compact storage of solution values for the solution of the adjoint system



- Simulations are reproducible from each checkpoint
- Cubic Hermite or variable-degree polynomial interpolation
- Store solution and first derivative at each checkpoint
- Force Jacobian evaluation at checkpoints to avoid storing it
- Computational cost: 2 forward and 1 backward integrations

KINSOL solves: F(u) = 0

- Originally, C rewrite of Fortran NKSOL (Brown and Saad)
- Newton Solvers: update iterate via $u^{k+1} = u^k + s^k, k = 0, ..., 1$
  - Inexact: approx. solves $J(u^k)s^k = -F(u^k)$
  - Modified: directly solves $J(u^{k-l})s^k = -F(u^k)$

$$J(u) = \frac{\partial F(u)}{\partial u}$$

- Optional constraints: $u_i > 0$, $u_i < 0$, $u_i \geq 0$ or $u_i \leq 0$
- Can separately scale equations and/or unknowns
- Backtracking and line search options for robustness
- Dynamic linear tolerance selection for use with iterative linear solvers

$$\|F(x^k) + J(x^k)s^{k+1}\| \leq \eta^k \|F(x^k)\|$$

KINSOL also includes fixed point and Picard iteration

Fixed point iterations use recursion to solve the fixed-point problem,

$$u = G(u) \qquad u^{k+1} = G(u^k), k = 0, 1, ...$$

- Picard iteration is a fixed point method for a rootfinding problem
  - Splits *F* into linear and nonlinear parts, $F(u) \equiv Lu - N(u)$
  - Defines a fixed point iteration based on the splitting

$$G(u) \equiv L^{-1}N(u) = u - L^{-1}F(u) \Rightarrow u^{k+1} = u^k - L^{-1}F(u^k)$$

  - Like Newton but with $L \approx J(u^k)$
- Fixed point iteration has a global but linear convergence theory
- Requires G to be a contraction $\|G(x) - G(y)\| \leq \gamma \|x - y\|, \quad \gamma < 1$

KINSOL includes both Picard and fixed point iterations *with acceleration*; ARKode includes accelerated fixed point

SUNDIALS provides many options for linear solvers

- Iterative Krylov linear solvers (all allow scaling and preconditioning)
  - All packages have GMRES, BiCGStab, & TFQMR; KINSOL also has FGMRES; ARKode also has PCG and FGMRES
  - Only require matrix-vector products, *Jv* may be user-supplied, or estimated via $Jv \approx \frac{1}{\epsilon}[F(u+\epsilon v) - F(u)]$
  - Require preconditioning for scalability
- Dense direct (dense or banded)
  - Require serial/threaded vector environments; banded requires some pre-defined structure to the data
  - J can be user-supplied or estimated with finite differences
- Sparse direct interfaces to external libraries: KLU, SuperLU_MT (threaded)
  - Currently requires serial or threaded vector environments
  - J must be supplied in compressed sparse column format (CSR soon)

Preconditioning is essential for large problems as Krylov methods can stagnate

- Competing preconditioner goals:
  - P should approximate the Jacobian matrix
  - P should be efficient to construct and solve
- Typical P (for time-dep. ODE problem) is $I - \gamma\tilde{J}, \quad \tilde{J} \approx J$
- The user must supply two routines for treatment of P:
  - Setup: evaluate and preprocess P (infrequently)
  - Solve: solve systems Px=b (frequently)
- The user can save and reuse P as directed by the solver
- Band and block-banded preconditioners are supplied for use with the included serial vector structures
- SUNDIALS offers hooks for user-supplied preconditioning
  - Can use *hypre* or PETSc or SuperLU_DIST, or …

CVODE , IDA, and ARKode are equipped with a rootfinding capability

- Finds roots of user-defined functions, $g_i(t, y) = 0$ or $g_i(t, y, \dot{y}) = 0$
- Important in applications where problem definition may change based on a function of the solution
- Roots are found by looking at sign changes, so only roots of odd multiplicity are found
- Checks each time interval for sign change
- When sign changes are found, apply a modified secant method with a tight tolerance to identify root

- Rootfinding is a critical feature for applications like power grid where solution-dependent system adaptations are common, e.g. voltage limit on a generator

SUNDIALS provides Fortran interfaces

- CVODE, ARKode, IDA, and KINSOL (not CVODES or IDAS)
- Cross-language calls go in both directions:
  - Fortran program calls solver creation, setup, solve, and output interface routines
  - Solver routines call users' problem-defining function/residual, matrix-vector product, and preconditioning routines
- For portability, all user routines have fixed names
- Examples are provided for each solver

SUNDIALS encapsulates all operations on data

- Vector structures can be user-supplied for problem-specific needs
- Essentially follows an object-oriented base/derived class approach (but in C), with most solvers defined on the base class
- The generic NVECTOR module defines:
  - A `content` structure (void *)
  - An `ops` structure, containing function pointers to actual vector operations supplied by a vector definition
- Each implementation of NVECTOR defines:
  - Content structure specifying the actual vector data and any information needed to make new vectors (problem or grid data)
  - Implemented vector operations
  - Routines to clone vectors

Interfacing SUNDIALS with other software is done in three areas

Vector interface

- Specifies:
  - 3 constructors/destructors
  - 3 utility functions
  - 9 streaming operators
  - 10 reduction operators
- SUNDIALS does not introduce parallelism outside vector ops
- Entire interaction w/ app. data is through these 19 ops
- All are level-1 BLAS ops
- Individual modules require only a subset

Application interface

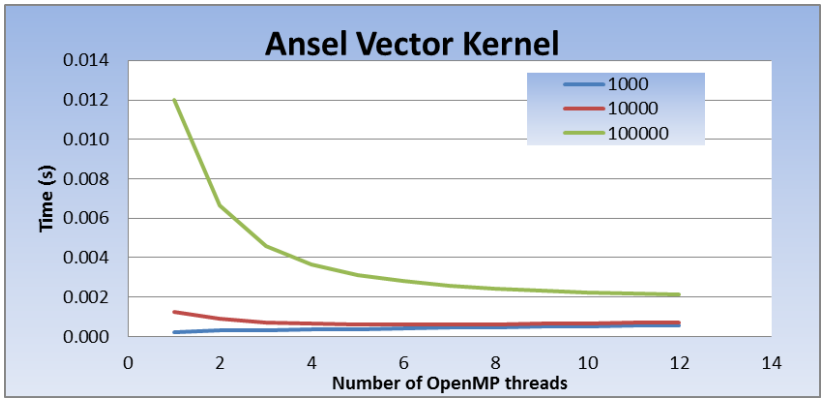
- Problem-defining function
- Jacobian evaluation (or Jv eval)
- Tolerances (vector or scalars)

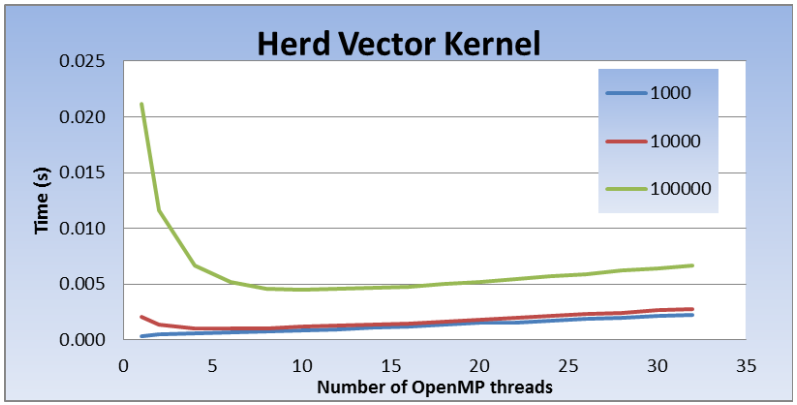
Linear solver interface

- Specifies the following five functions: init, setup, solve, perf, and free
- Optional: Preconditioner setup and solve
- SUNDIALS is independent of solve strategy

SUNDIALS provides serial and parallel NVECTOR implementations

- *Use is optional*
- Vectors are laid out as an array of doubles (or floats)
- Appropriate lengths (local, global) are specified
- Operations are fast since stride is always 1
- All operations provided for each implementation: Serial, Threaded (OpenMP), Threaded (pThreads), and Distributed memory (MPI)
- Can serve as templates for creating a user-supplied vector
- Threaded kernels require long vectors (>10K) for appreciable benefit:



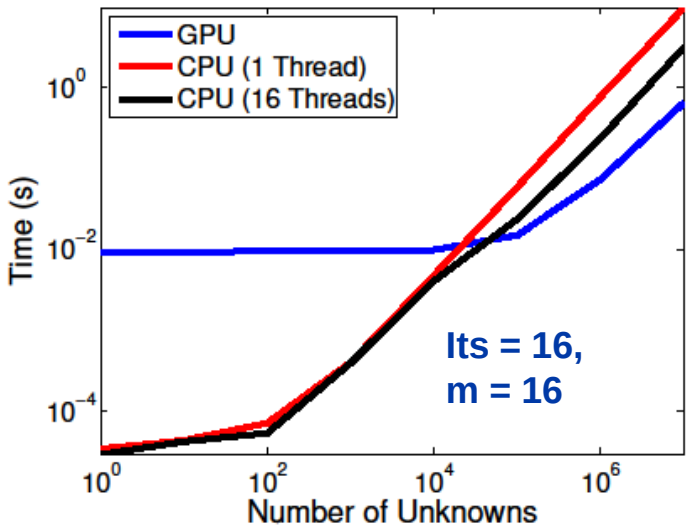


We have begun evaluating user model for GPUs

- SUNDIALS integrators operate almost solely on vectors
- Experimented with a first GPU implementation with accelerated fixed point (FP)
  - Main loop of FP runs on CPU
  - All vectors are created on and remain on GPU
  - Vector operations run on GPU using CuBLAS from Nvidia
  - Data resides in GPU RAM
- Expect the solver to be memory bound
- Peak bandwidth of GPU on LLNL Surface machine is ~5 x greater than CPU bandwidth
- Compared with a CPU-only implementation using standard BLAS and keeping data in CPU RAM

See definite benefit from use of GPU

Run times for CPU and GPU (fcn cost not timed)

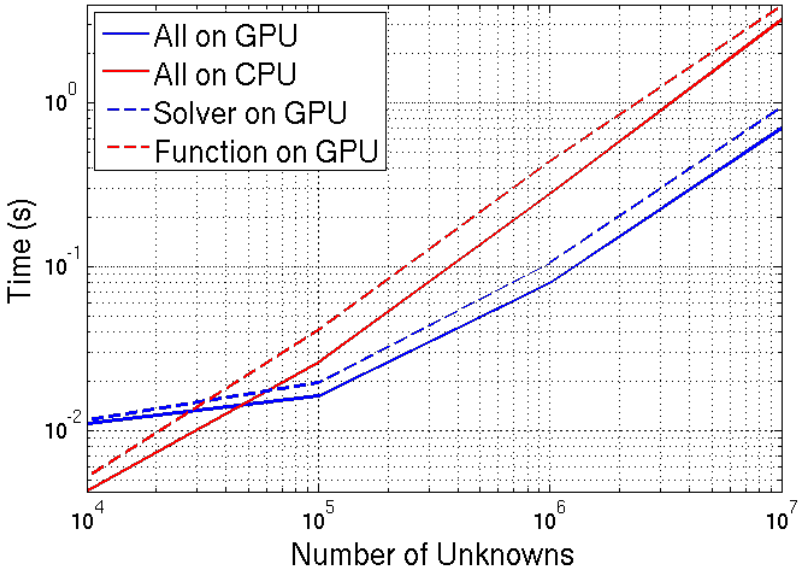


- For vectors less than 10,000, CPU versions take less time than GPU version
- CPU version costs remain ~constant until vector lengths reach 100
- GPU version cost is constant until vector is 10,000 – length at which the work per vector dominates overhead per vector op
- Times approach linear with vector length
- When both CPU and GPU versions are in linear regime, we expect ratio between timings to be ~ratio of bandwidth
- Threading reduces runtime on CPU
- GPU gives more benefit on large problem

We experimented with the user model for GPU with simple function eval

- Applied 16 iterations with simple function, f(x) = x; touches 2 vectors
- CPU runs used all 16 cores and 16 threads
- Timed 4 combinations:
  - Both function eval (FE) and vector ops on same side of bus
  - And on opposite side: CPU (GPU) = vectors on CPU, FE on GPU

- Fastest times occur when vectors are on GPU
- FE on opposite side of bus causes data transfer every iteration
- For this “light weight” function, not worth computing it on GPU if vector operations are not also on GPU (in fact, this gives worst performance)



SUNDIALS code usage is similar across the suite

Core components to any user program:

1. `#include` header files for integrator/solver(s) and vector implementation
2. Create data structure for solution vector
3. Set up integrator
    1. Create integrator object (allocates solver-specific memory structure)
    2. Initialize integrator (sets solution vector, problem-defining function pointers, default solver parameters)
    3. Call “Set” routines to customize integrator behavior/parameters
4. Set up linear solver (if needed for Newton/Picard)
    1. Create linear solver object
    2. Call “Set” routines to customize behavior/parameters
5. Call integrator (once or repeatedly)
6. Destroy integrator and vectors to free memory

SUNDIALS code usage is similar across the suite

For CVODE with parallel vector implementation and GMRES solver:

```
#include “cvode.h”
#include “cvode_spgmr.h”
#include “nvector_parallel.h”

y = N_VNew_Parallel(comm, local_n, NEQ);
cvmem = CVodeCreate(CV_BDF,CV_NEWTON);
flag = CVodeSet*(…);
flag = CVodeInit(cvmem,rhs,t0,y,…);
flag = CVSpgmr(cvmem,…);
flag = CVSpilsSet*(cvmem, …);
for(tout = …) {
   flag = CVode(cvmem, …,y,…);  }

NV_Destroy(y);
CVodeFree(&cvmem);
```

Availability

Open source BSD license

`https://computation.llnl.gov/casc/sundials`

Publications

`https://computation.llnl.gov/casc/sundials/documentation/documentation.html`

Web site:

- Individual codes or full suite download
- User manuals
- User group email list (~1,500 subscribers)
- SUNDIALS Uses

The SUNDIALS Team:

Carol S. Woodward, Daniel R. Reynolds, Alan C. Hindmarsh, Slaven Peles, David J. Gardner, and Lawrence E. Banks

We acknowledge significant past contributions of Radu Serban