Using SUNDIALS on GPU-based HPC platforms

Center for Advanced Computation

September 25, 2024

Daniel R. Reynolds (SMU), David J. Gardner (LLNL)

U.S. DEPARTMENT OF ENERGY | Office of Science

SciDAC
Scientific Discovery through Advanced Computing

Lawrence Livermore National Laboratory

SMU Mathematics

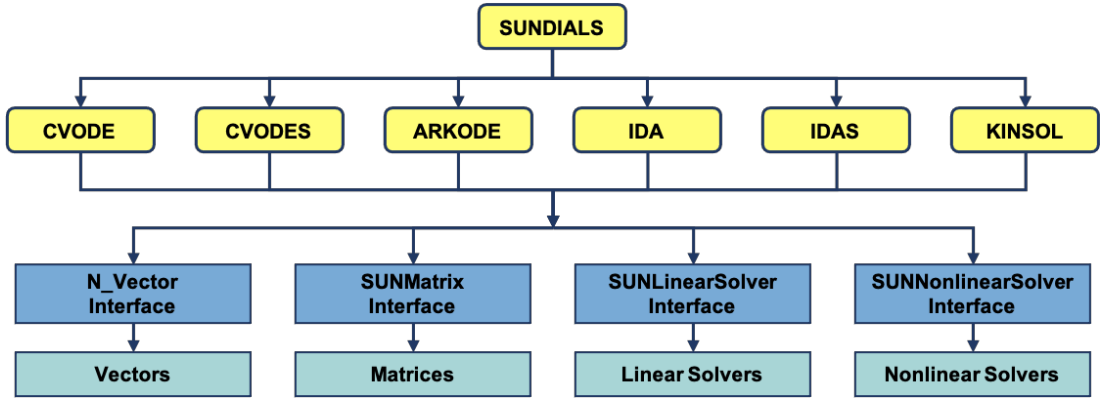
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. LLNL-PRES-870155

Tutorial Outline

- **Overview of SUNDIALS (Dan)**
- How to use the time integrators (Dan)
- Using SUNDIALS on GPU-based HPC platforms (David)
- Brief: How to download and install SUNDIALS (David)
- Closing Remarks (Dan)

SUite of Nonlinear and DIfferential-ALgebraic Solvers

- SUNDIALS is a software library consisting of ODE and DAE integrators and nonlinear solvers
- Written in C with interfaces to modern Fortran
- New interfaces in Matlab starting with release R2024a
- Designed to be incorporated into existing codes
- Through the DOE Exascale Computing Project, developed a rich infrastructure of support on exascale systems and applications
- Freely available; released under the BSD 3-Clause license (>100,000 clones/downloads per year)
- Active user community supported by sundials-users email list
- Detailed user manuals included with each package and online at https://sundials.readthedocs.io



- Nonlinear and linear solvers and all data use is fully encapsulated from the integrators and can be user-supplied
- All parallelism is encapsulated in vector and solver modules and user-supplied functions

https://computing.llnl.gov/casc/sundials

SUNDIALS offers packages with linear multistep and multistage time integration methods

- CVODE, IDA, and sensitivity analysis variants (forward and adjoint), CVODES and IDAS, use linear multistep methods
  - CVODE solves ODEs, $\dot{y} = f(t, y)$
  - IDA solves DAEs, $F(t, y, \dot{y}) = 0$
  - Adaptive in both order and step sizes
  - Both packages include stiff BDF methods
  - CVODE includes nonstiff Adams-Moulton methods

- ARKODE provides adaptive one-step, multistage time integration methods
  - Provides support for users with changing problem structures (e.g., adaptive mesh algorithms)

- All 5 packages include event detection
  - After each time step check whether user-provided functions change sign and stop integration if so

ARKODE, an adaptive Runge-Kutta package, was released in SUNDIALS in 2015

- Provides an infrastructure for rapid development of general, adaptive, one-step methods:
  - ARKODE provides the outer time integration loop
  - Time-stepping modules handle problem-specific components: problem definition, algorithm for a single step
  - Modules leverage a shared infrastructure
    - Different methods for controlling step size adaptivity
    - Other features such as rootfinding, constraint handling, etc.
  - Supports parallel-in-time capabilities via a native interface to the XBraid library
- There are currently four time-stepping modules available:

ARKStep: ERK, DIRK, and ImEx-ARK methods

$$M(t)\,y' = f^E(t,y) + f^I(t,y), \qquad y(t_0) = y_0$$

ERKStep: streamlined module for ERK methods

$$y' = f(t,y), \qquad y(t_0) = y_0$$

MRIStep: Multirate infinitesimal methods

$$y' = f^F(t,y) + f^S(t,y), \qquad y(t_0) = y_0$$

SPRKStep: Symplectic partitioned RK methods

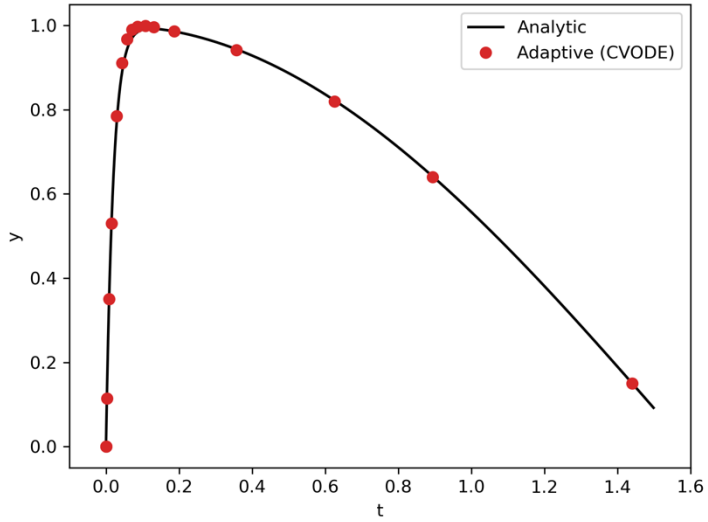
$$p' = f(t,q), \qquad p(t_0) = p_0$$
$$q' = f(t,p), \qquad q(t_0) = q_0$$

SUNDIALS provides adaptive time stepping choosing time steps to minimize local error and maximize efficiency

- Time step selection
  - Based on the method, estimate the time step error, E(Δt), using an embedded method of one lower order (RK) or direct error estimate (LMM)
  - Accept step if $||E(\Delta t)||_{WRMS} < 1$; Reject it otherwise

$$\|y\|_{\text{wrms}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(w_i\ y_i)^2} \qquad w_i = \frac{1}{RTOL|y_i| + ATOL_i}$$

  - Choose next step, $\Delta t'$, so that $||E(\Delta t')||_{WRMS}$ is expected to be small
- Some algorithms also allow order adaption: choose the order that gives the largest step expected to meet the error condition
- ARKODE supplies advanced "error controllers" which can adapt these step sizes to meet other objectives:
  - minimize failed steps
  - maximize step sizes
  - maintain smooth transitions in the step sizes as integration proceeds



Adaptivity can give much more efficient (and accurate) results

KINSOL solves systems of nonlinear algebraic equations, F(u) = 0

- Newton Solvers: update iterate via $u^{k+1} = u^k + s^k, k = 0, ..., 1$
  - Get update by solving: $J(u^k)s^k = -F(u^k) \qquad J(u) = \frac{\partial F(u)}{\partial u}$
  - Inexact method approximately solves this equation
- Dynamic linear tolerance selection for use with iterative linear solvers

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

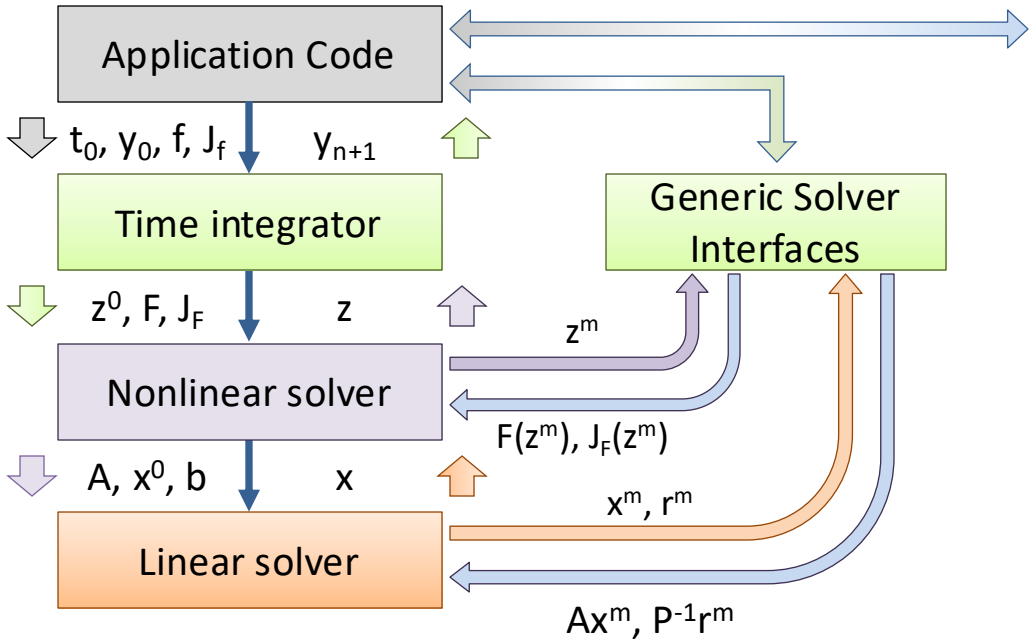
- Can separately scale equations and unknowns
- Backtracking and line search options for robustness
- KINSOL also solves fixed point and Picard iterations with optional Anderson acceleration

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

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

SUNDIALS uses modular design and control inversion to interface with application codes, external solvers, and encapsulate parallelism

- Control passes between the integrator, solvers, and application code as the integration progresses



Application / discretization framework:
RHS Function, f
RHS Jacobian, J_f
Preconditioner, P

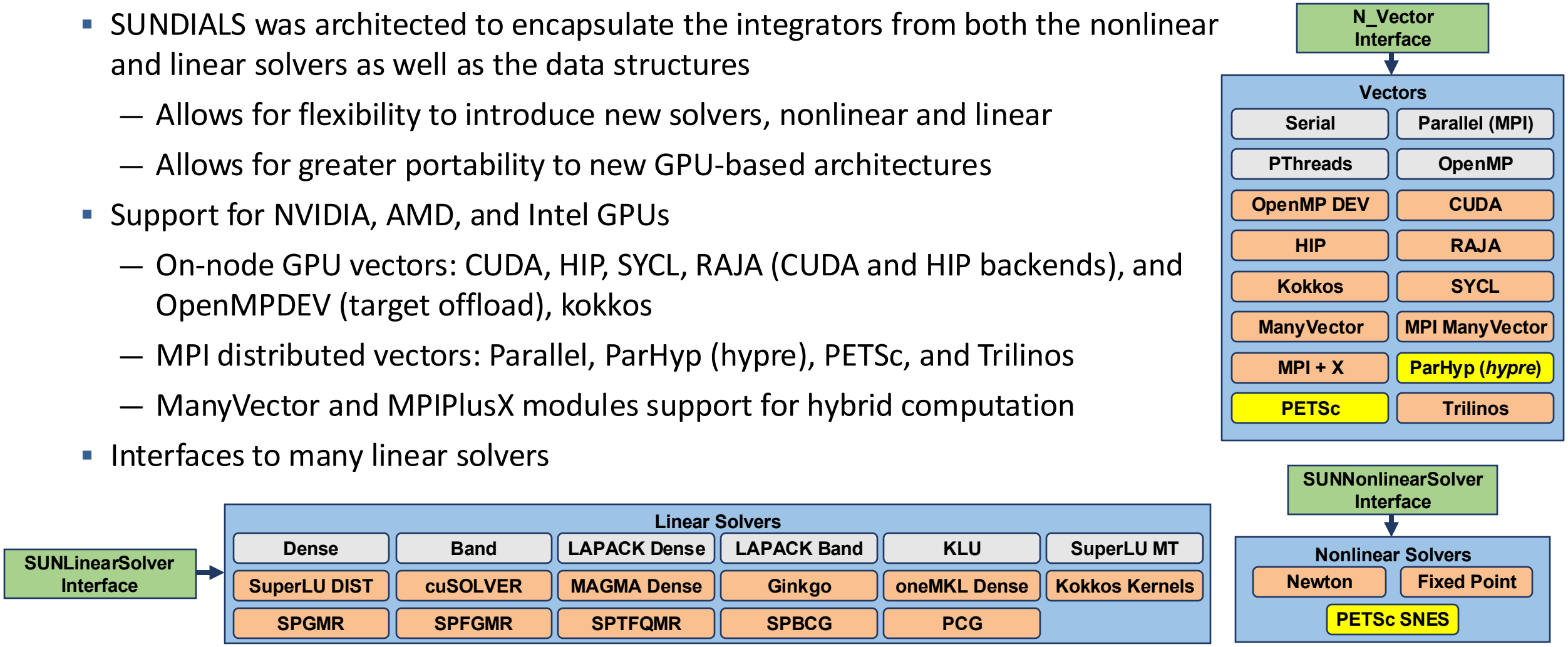
Time integrator and nonlinear solver are agnostic of vector data layout and specific solvers used

- Nonlinear and linear solver modules are designed for generic systems

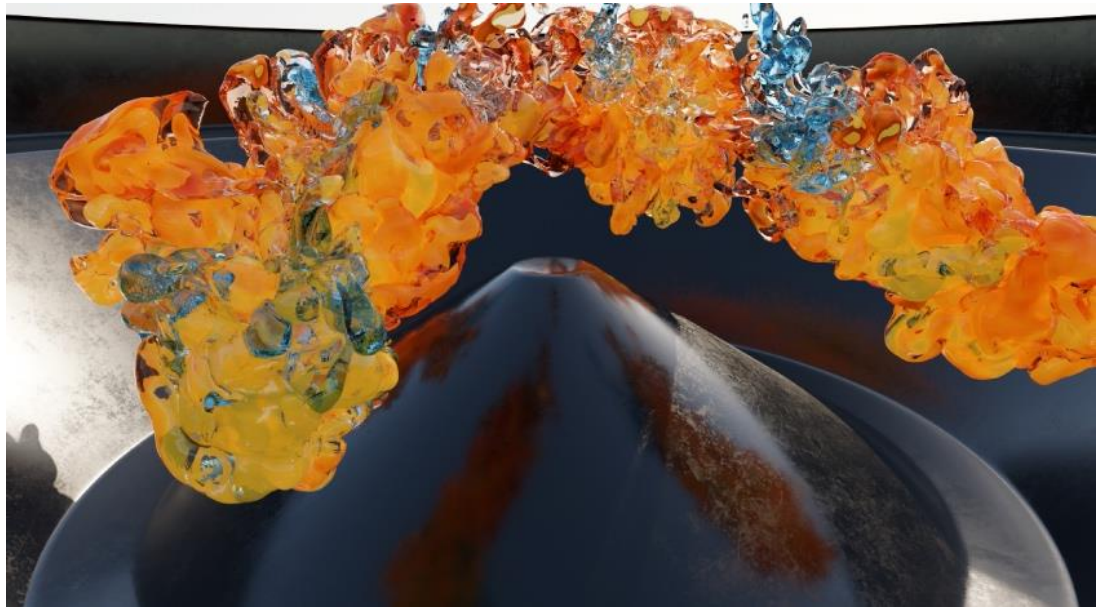
$$F(y) = 0 \qquad G(y) = y \qquad Ax = b$$

SUNDIALS integrators and nonlinear solvers are written on top of a set of common vector and solver APIs

- SUNDIALS was architected to encapsulate the integrators from both the nonlinear and linear solvers as well as the data structures
  - Allows for flexibility to introduce new solvers, nonlinear and linear
  - Allows for greater portability to new GPU-based architectures
- Support for NVIDIA, AMD, and Intel GPUs
  - On-node GPU vectors: CUDA, HIP, SYCL, RAJA (CUDA and HIP backends), and OpenMPDEV (target offload), kokkos
  - MPI distributed vectors: Parallel, ParHyp (hypre), PETSc, and Trilinos
  - ManyVector and MPIPlusX modules support for hybrid computation
- Interfaces to many linear solvers



With efficient solvers and algorithms, Pele codes were some of the first to run on the Frontier exascale system



Isosurfaces of diesel fuel entering a turbulent methane-air premixture at 60 atm. High temperature pockets (seen in red and yellow) form when local kernels of diesel fuel ignite. The simulation size increases over time, growing up to 1B degrees of freedom, as fine-grid resolution tracks the evolving turbulent jets.

Courtesy of Marc Day and Jon Rood (NREL). Animation created by Nicholas Brunhart-Lupo (NREL) based on a simulation performed on ORNL's Crusher machine.

This problem was run on 7,000 nodes of the ORNL Frontier Exascale machine where SUNDIALS used GPUs to solve chemistry systems on every grid cell in a 7-layer adaptive mesh hierarchy for a problem with 60B grid cells and approximately 2.4T degrees of freedom

Several SUNDIALS team members were included in the Pele Suite Developers who were 2024 R&D100 Finalists.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

SUNDIALS Team

Current Team:

Cody Balos

David Gardner

Alan Hindmarsh

Dan Reynolds

Steven Roberts

Carol Woodward

Postdocs:

Mustafa Aggul

Sylvia Amihere

Alumni:

Radu Serban

Scott D. Cohen, Peter N. Brown, George Byrne, Allan G. Taylor, Steven L. Lee, Keith E. Grant, Aaron Collier, Lawrence E. Banks, Steve G. Smith, Cosmin Petra, Homer Walker, Slaven Peles, John Loffeld, Dan Shumaker, Ulrike M. Yang, James Almgren-Bell, Shelby L. Lockhart, Rujeko Chinomona, Daniel McGreer, Hunter Schwartz, Hilari C. Tiedeman, Ting Yan, Jean M. Sexton, and Chris White

Tutorial Outline

- Overview of SUNDIALS (Dan)
- **How to use the time integrators (Dan)**
- Using SUNDIALS on GPU-based HPC platforms (David)
- Brief: How to download and install SUNDIALS (David)
- Closing Remarks (Dan)

The “Skeleton” for Using SUNDIALS Integrators

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

C/C++ vs Fortran

While written in C, SUNDIALS fully supports applications written in C++ or modern Fortran:

- Leverage the `iso_c_binding` module and the `bind(C)` attribute from the F2003 standard.
- SUNDIALS' F2003 interfaces closely follow the C/C++ API.
- Generic SUNDIALS structures, e.g. , `N_Vector`, are interfaced as Fortran derived types, and function signatures are matched but with an `F` prepending the name, e.g. `FN_VDotProd` instead of `N_VDotProd`.
- Constants are named exactly as they are in the C/C++ API.
- Accordingly, using SUNDIALS via the Fortran 2003 interfaces looks just like using it in C/C++.

The remainder of this tutorial will therefore focus on C/C++; please reserve questions regarding the Fortran interface for one-on-one discussions.

Supplying the Initial Condition Vector(s)

- As discussed earlier, all SUNDIALS integrators operate on data through the N_Vector API.
- Each provided vector module has a unique set of "constructors", e.g.

```
N_Vector N_VMake_Hip(sunindextype length, sunrealtype* h_data, sunrealtype* d_data,
                     SUNContext sunctx);

N_Vector N_VMake_Sycl(sunindextype length, sunrealtype* h_data, sunrealtype* d_data,
                      ::sycl::queue* Q, SUNContext sunctx);

N_Vector N_VMake_MPIPlusX(MPI_Comm comm, N_Vector x, SUNContext sunctx);
```

- The CUDA, RAJA and OpenMPDEV vector modules have similar constructors to HIP, above.
- Existing application codes may provide a thin N_Vector wrapper to "teach" SUNDIALS how to operate directly on their data structures.
- Once an application creates a vector for their data, they fill it with the initial conditions for the problem and supply it to the integrator, who "clones" it to create its workspace.
- For Kokkos, PETSc, *hypre,* and Trilinos, SUNDIALS NVector wrappers require only the native vector structure and the SUNDIALS context.

Supplying the IVP to the Integrator – RHS/Residual Functions

Once the problem data is encapsulated in a vector, all that remains for basic SUNDIALS usage is specification of the IVP itself:

- CVODE and ARKODE specify the IVP through right-hand side function(s):

```
int (*RhsFn)(sunrealtype t, N_Vector y, N_Vector ydot, void *user_data)
```

- IDA specifies the IVP through a residual function:

```
int (*ResFn)(sunrealtype t, N_Vector y, N_Vector ydot, N_Vector r,
  void *user_data)
```

- The *user_data pointer enables problem-specific data to be passed through the SUNDIALS integrator and back to the RHS/residual routine (i.e., no global memory).

CVODE/ARKODE RHS Functions

```c
/*
 * RHS function
 * The form of the RHS function is controlled by the flag passed as f_data:
 *   flag = RHS1 -> y' = -y
 *   flag = RHS2 -> y' = -5*y
 */

static int f(sunrealtype t, N_Vector y, N_Vector ydot, void* f_data)
{
  int* flag;

  flag = (int*)f_data;

  switch (*flag)
  {
  case RHS1: NV_Ith_S(ydot, 0) = -NV_Ith_S(y, 0); break;
  case RHS2: NV_Ith_S(ydot, 0) = -5.0 * NV_Ith_S(y, 0); break;
  }

  return (0);
}
```

Example:
cvDisc_dns.c

Initializing the Integrators – CVODE and IDA

The IVP inputs are supplied when constructing the integrator.

```c
/* Create solver memory structure, and specify use of BDF method */
void* cvode_mem = CVodeCreate(CV_BDF, sunctx);
if (cvode_mem == NULL) { MPI_Abort(comm, 1); }

/* Initialize integrator memory with problem specification */
int retval = CVodeInit(cvode_mem, f, T0, y);
if (retval != CV_SUCCESS) { MPI_Abort(comm, 1); }
```

CVODE

```c
/* Create solver memory structure */
void* ida_mem = IDACreate(sunctx);
if (ida_mem == NULL) { MPI_Abort(comm, 1); }

/* Initialize integrator memory with problem specification */
int retval = IDAInit(ida_mem, res, T0, y, yp);
if (retval != IDA_SUCCESS) { MPI_Abort(comm, 1); }
```

IDA

Initializing the Integrators – ARKODE

```c
/* Create solver memory structure, and specify ImEx problem */
void* arkode_mem = ARKStepCreate(fe, fi, T0, y, sunctx);
if (arkode_mem == NULL) { MPI_Abort(comm, 1); }
```

```c
/* Create solver memory structure, and specify implicit problem */
void* arkode_mem = ARKStepCreate(NULL, f, T0, y, sunctx);
if (arkode_mem == NULL) { MPI_Abort(comm, 1); }
```

```c
/* Create solver memory structure, and specify explicit problem */
void* arkode_mem = ARKStepCreate(f, NULL, T0, y, sunctx);
if (arkode_mem == NULL) { MPI_Abort(comm, 1); }
```

ImEx (top), implicit (middle), explicit (bottom)

```c
/* Create fast solver memory structure, and specify ImEx problem */
void* inner_mem = ARKStepCreate(ffe, ffi, T0, y, sunctx);
if (inner_mem == NULL) { MPI_Abort(comm, 1); }

/* Set up fast integrator as normal */
int retval = ARKodeSet...(inner_mem, ...);
if (retval != ARK_SUCCESS) { MPI_Abort(comm, 1); }

/* Create inner stepper */
MRIStepInnerStepper inner_stepper = NULL;
retval = ARKStepCreateMRIStepInnerStepper(inner_mem, &inner_stepper);
if (retval != ARK_SUCCESS) { MPI_Abort(comm, 1); }

/* Create slow solver memory structure, and specify multirate problem */
void* arkode_mem = MRIStepCreate(fse, fsi, T0, y, inner_stepper, sunctx);
if (arkode_mem == NULL) { MPI_Abort(comm, 1); }
```

Multirate with ImEx at both time scales

Optional Inputs (all Integrators)

A variety of optional "Set" routines allow enhanced control over the integration process. Here we discuss the most often-utilized options (see documentation for the full set).

- Tolerance specification – rtol with scalar or vector-valued atol, or user-specified routine to compute the error weight vector

$$w_k \approx \frac{1}{\mathrm{rtol}\,|y_k| + \mathrm{atol}_k} > 0, \quad k = 1, \ldots, N$$

- SetNonlinearSolver, SetLinearSolver – attaches desired nonlinear solver, linear solver and (optionally) matrix modules to the integrator.
- SetUserData – specifies the (void *user_data) pointer that is supplied to user routines.
- SetMaxNumSteps, SetMaxStep, SetMinStep, SetInitStep – provides guidance to time step adaptivity algorithms.
- SetStopTime – specifies the value of t_{stop} to use when advancing solution (this is retained until this stop time is reached or modified through a subsequent call).

Supplying Options to the Integrators

After constructing the integrator, additional options may be supplied through various “Set” routines (example from ark_heat1D_adapt.c):

```
/* Set routines */
int retval;
retval = ARKodeSetUserData(arkode_mem, (void*) udata); /* Pass udata to user functions */
if (retval != ARK_SUCCESS)  { return 1; }

retval = ARKodeSetMaxNumSteps(arkode_mem, 10000);      /* Increase max num steps  */
if (retval != ARK_SUCCESS)  { return 1; }

retval = ARKodeSStolerances(arkode_mem, rtol, atol);   /* Specify tolerances */
if (retval != ARK_SUCCESS)  { return 1; }

retval = ARKodeSetPredictorMethod(arkode_mem, 0);      /* Set implicit predictor method */
if (retval != ARK_SUCCESS)  { return 1; }
```

Usage Modes for SUNDIALS Integrators

While t_0 is supplied at initialization, the *direction* of integration is specified on the first call to advance the solution toward the output time t_{out}. This may occur in one of four "usage modes":

- **Normal** – take internal steps until t_{out} is overtaken in the direction of integration, e.g. for forward integration $t_{n-1} < t_{\text{out}} \leq t_n$; the solution $y(t_{\text{out}})$ is then computed by interpolation.
- **One-step** – take a single internal step $y_{n-1} \to y_n$ and then return control back to the calling program. Interpolation in not preformed $y(t_{\text{out}})$ is returned.
- **Normal + TStop** – take internal steps until the next step will overtake t_{stop} ; limit the next internal step so that $t_n = t_{\text{stop}}$. No interpolation is performed.
- **One-step + TStop** – performs a combination of both "One-step" and "TStop" modes above.

Advancing the Solution

Once all options have been set, the integrator is called to advance the solution toward t_{out}.

```
retval = CVode(cvode_mem, tout, y, &tret, CV_NORMAL);
if (retval != CV_SUCCESS) { MPI_Abort(comm, 1); }
```

CVODE

```
retval = IDASolve(ida_mem, tout, &tret, y, yp, IDA_ONE_STEP);
if (retval != IDA_SUCCESS) { MPI_Abort(comm, 1); }
```

IDA

```
retval = ARKodeEvolve(arkode_mem, tout, y, &tret, ARK_NORMAL);
if (retval != ARK_SUCCESS) { MPI_Abort(comm, 1); }
```

ARKODE

Optional Outputs – General Time Integration

Either between calls to advance the solution, or at the end of a simulation, users may retrieve a variety of optional outputs from SUNDIALS integrators via “Get” routines.

- GetDky (Dense solution output) – using the same infrastructure that performs interpolation in “normal” use mode, users may request values $\frac{d^k}{dt^k} y(t)$ for $t_{n-1} \leq t \leq t_n$, where $0 \leq k \leq k_{\max}$.
- Time integration statistics:
  - GetNumSteps – the total number of internal time steps since initialization
  - GetCurrentStep – the current internal time step size
  - GetCurrentTime – the current internal time (since this may have passed t_{out})
  - GetCurrentOrder (IDA/CVODE) – the current method order of accuracy
  - GetEstLocalErrors – returns the current temporal error vector, $\text{error} \in \mathbb{R}^N$

Optional Outputs – Algebraic Solver Statistics

- GetNumNonlinSolvIters – number of nonlinear solver iterations since initialization.
- GetNumNonlinSolvConvFails – number of nonlinear solver convergence failures.

- GetNumLinSolvSetups – number of calls to setup the linear solver or preconditioner.
- GetNumLinIters – number of linear solver iterations since initialization.
- GetNumLinConvFails – number of linear solver convergence failures.
- GetNumJacEvals, GetNumJtimesEvals, GetNumPrecEvals, GetNumPrecSolves – the number of calls to user-supplied Jacobian/preconditioner routines.

Retrieving Output from the Integrators

```
long int nst, nfe, nsetups, netf, nni;
int retval;

retval = CVodeGetNumSteps(cvode_mem, &nst);
if (retval != CV_SUCCESS) { return 1; }

retval = CVodeGetNumRhsEvals(cvode_mem, &nfe);
if (retval != CV_SUCCESS) { return 1; }

retval = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
if (retval != CV_SUCCESS) { return 1; }

retval = CVodeGetNumErrTestFails(cvode_mem, &netf);
if (retval != CV_SUCCESS) { return 1; }

retval = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
if (retval != CV_SUCCESS) { return 1; }
```

```
/* If TSTOP was not set, we'd need to find y(t1) */
retval = CVodeGetDky(cvode_mem, t1, 0, y);
if (retval != CV_SUCCESS) { return 1; }
```

Above: dense solution output from cvDisc_dns.c

Left: scalar-valued solver statistics from cvAdvDiff_kry_cuda.cu

Advanced Features

This tutorial is only the beginning; SUNDIALS also supports a number of 'advanced' features to examine auxiliary conditions, change the IVP, and improve solver efficiency.

- *Root-finding* – while integrating the IVP, SUNDIALS integrators can find roots of a set of auxiliary user-defined functions $g_i(t, y(t)),\ i = 1, \ldots, N_r$; sign changes are monitored between time steps, and a modified secant iteration (along with `GetDky`) zeros in on the roots.

- *Reinitialization* – allows reuse of existing integrator memory for a "new" problem (e.g., when integrating across a discontinuity, or integrating many independent problems of the same size). All solution history and solver statistics are erased, but no memory is (de)allocated.

- *Constraint-handling* – positivity / negativity / non-positivity / non-negativity constraints may be set on individual solution components (handled through time step size adjustments).

Advanced Features – Continued

- *Resizing* (ARKODE) – allows resizing the problem and all internal vector memory, without destruction of temporal adaptivity heuristic information or solver statistics. This is primarily useful when integrating problems with spatial adaptivity.

- *Relaxation* (ARKODE) – adjusts the step size to help preserve a scalar-valued quantity of interest (e.g., total energy), without requiring use of special integrators or recomputing time steps.

- *Sensitivity Analysis* (CVODES / IDAS) – allows computation of forward and adjoint solution sensitivities with respect to problem parameters.

- *Problem-specific algebraic solvers* – users are encouraged to supply custom nonlinear solvers, linear solvers, or preconditioners that leverage problem structure and domain-specific knowledge.

Tutorial Outline

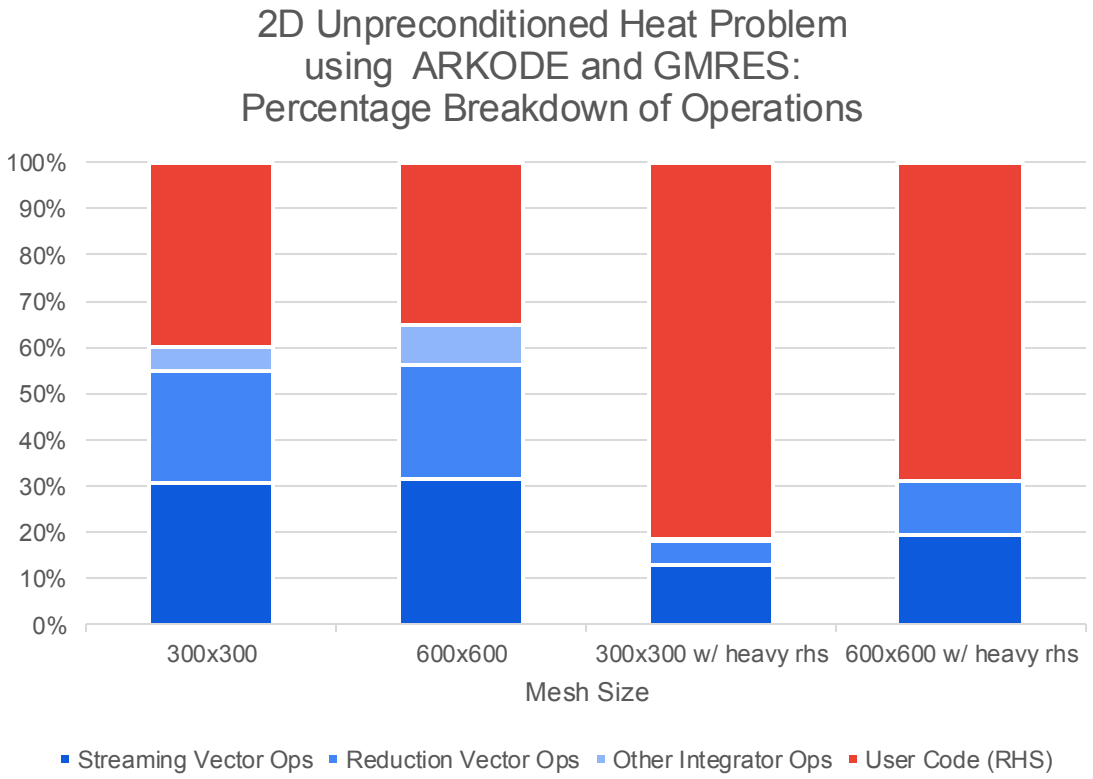
- Overview of SUNDIALS (Dan)
- How to use the time integrators (Dan)
- **Using SUNDIALS on GPU-based HPC platforms (David)**
- Brief: How to download and install SUNDIALS (David)
- Closing Remarks (Dan)

SUNDIALS Supports AMD, Intel, and NVIDIA GPUs

- SUNDIALS' object-oriented design enables supporting various GPUs with class implementations targeting different programming models e.g., CUDA, HIP, SYCL, Kokkos, RAJA, etc.
- To leverage GPU acceleration:
  - Compile SUNDIALS with GPU features enabled e.g., ENABLE_CUDA=ON, ENABLE_HIP=ON, etc.
  - Utilize ***GPU-enabled class implementations*** i.e., vectors, matrices, and algebraic solvers
  - Supply ***callback functions*** that leverage GPU acceleration e.g., ODE right-hand side functions
- Primary uses cases:
  1. SUNDIALS controls the ***main time-integration*** *loop*, and evolves a large ODE system in a distributed manner (MPI+X) e.g., FEM, FD, or FV applications
  2. SUNDIALS is used as a ***local integrator*** for many independent subsystems within a larger problem e.g., combustion applications evolving local reaction systems in each grid cell within the spatial mesh

Key Considerations When Using SUNDIALS With GPUs

- The SUNDIALS integrator and solver logic is on the host and launches kernels to perform operations on data in device memory
- In case 1, speedup is easier to obtain
  - Long vectors and “heavy” right-hand side functions can overcome overhead
- In case 2, need to group independent subsystems into a larger system
  - Can introduce other problems, like heterogeneity in subsystem stiffness
- The key factor is to ensure the **GPUs have sufficient work** to hide overheads from kernel launch latency



The “heavy” RHS includes a 2x cost sleep function in the RHS evaluation to mimic applications with more work in the RHS function

Key Considerations When Using SUNDIALS With GPUs

- The user must **ensure data coherency** between the CPU host and GPU-device
  - SUNDIALS integrators *do not internally migrate data* from one memory space to another
  - The location of the data depends entirely on the object implementations utilized
- For optimal performance it is critical to **minimize data movement** between the host and device
  - It is recommended to only access data in the device memory space as much as possible
  - Ideally, data would reside in device memory for the entire duration of the simulation
- SUNDIALS-provided GPU-enabled objects **keep data resident in the GPU-device memory**
  - When control passes ***from the user to SUNDIALS***, simulation data must be up-to-date in the device memory space (unless using UVM)
  - Similarly, when control ***returned from SUNDIALS to the user***, it should be assumed that any simulation data is only up-to-date in the device memory space

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Walkthrough a GPU-enabled SUNDIALS example

- Consider a case with many independent ODEs combined into a larger group and evolved together
- In this example, we evolve a stiff, three species **Brusselator reaction problem**

$$\frac{du}{dt} = a - (w+1)u + vu^2 \qquad \frac{dv}{dt} = wu - vu^2 \qquad \frac{dw}{dt} = (b-w)/\epsilon - wu$$

- The problem is replicated ngroups times giving a total problem size of 3*ngroups to evolve
- Advance the system in time with CVODE adaptive order and step BDF methods with a modified Newton iteration and the MAGMA batched direct linear solver
- **CUDA / HIP (MAGMA):** examples/cvode/magma/cv_bruss_batched_magma.cpp
- **Kokkos (Kokkos Kernels):** examples/cvode/kokkos/cv_bruss_batched_magma.cpp
- **SYCL (oneMKL):** examples/cvode/CXX_onemkl/cvRoberts_blockdiag_onemkl.cpp (similar)
- **CUDA (cuSPARSE):** examples/cvode/cuda/cvRoberts_block_cusolversp_batchqr.cu (similar)

Brusselator Example: SUNDIALS Headers

```
#include <sundials/sundials_core.hpp>       /* C++ Convenience classes   */
#include <sunmemory/sunmemory_cuda.h>       /* CUDA Memory Helper        */
#include <nvector/nvector_cuda.h>           /* CUDA Vector               */
#include <sunmatrix/sunmatrix_magmadense.h> /* MAGMA Dense Matrix        */
#include <sunlinsol/sunlinsol_magmadense.h> /* MAGMA Dense Linear Solver */
#include <sunlinsol/sunlinsol_spgmr.h>      /* GMRES Linear Solver       */
#include <cvode/cvode.h>                    /* CVODE Integrator          */
```

- SUNDIALS is comprised of a collection of independent and interchangeable modules each with their own header file and library
- Typical includes: vector, matrix (if using a matrix-based solver), linear solver, and integrator
- `sundials_core.hpp` provides access to C++ RAII safe classes that wrap some SUNDIALS objects e.g., the SUNContext (more on this next)

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Brusselator Example: SUNContext Object

```
int main(int argc, char* argv[])
{
  ...

  /* Create the SUNDIALS context */
  sundials::Context sunctx;
```

- Users ***must*** create a SUNContext object prior to any other calls to SUNDIALS library functions
- All SUNDIALS objects hold a reference to a common SUNContext object (in this case wrapped in the sundials::Context convenience class for C++ codes)
- For MPI problems, the constructor takes an MPI communicator i.e., `sunctx(global_comm)`
- This class supports common capabilities cross SUNDIALS:
  - Error handling
  - Performance profiling (with optional support for Caliper, software.llnl.gov/Caliper)
  - Status logging

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Brusselator Example: SUNMemoryHelper Object *(optional)*

```
/* Create a CUDA SUNMemoryHelper */
SUNMemoryHelper memhelper = SUNMemoryHelper_Cuda(sunctx);
```

- The SUNMemoryHelper provides an interface that allows applications to utilized their own memory pools or memory abstraction layer under SUNDIALS GPU enabled objects.

Alloc	Creates SUNMemory object and allocates memory of a given type and size, *required*
Dealloc	Frees memory own by a SUNMemory object and destroys the object, *required*
Copy	Synchronously copies data between SUNMemory objects, *required*
CopyAsync	Asynchronously copies data between SUNMemory objects, *optional*
Clone	Creates a clone of a SUNMemoryHelper, *optional*
Destroy	Destroys a SUNMemoryHelper, *optional*

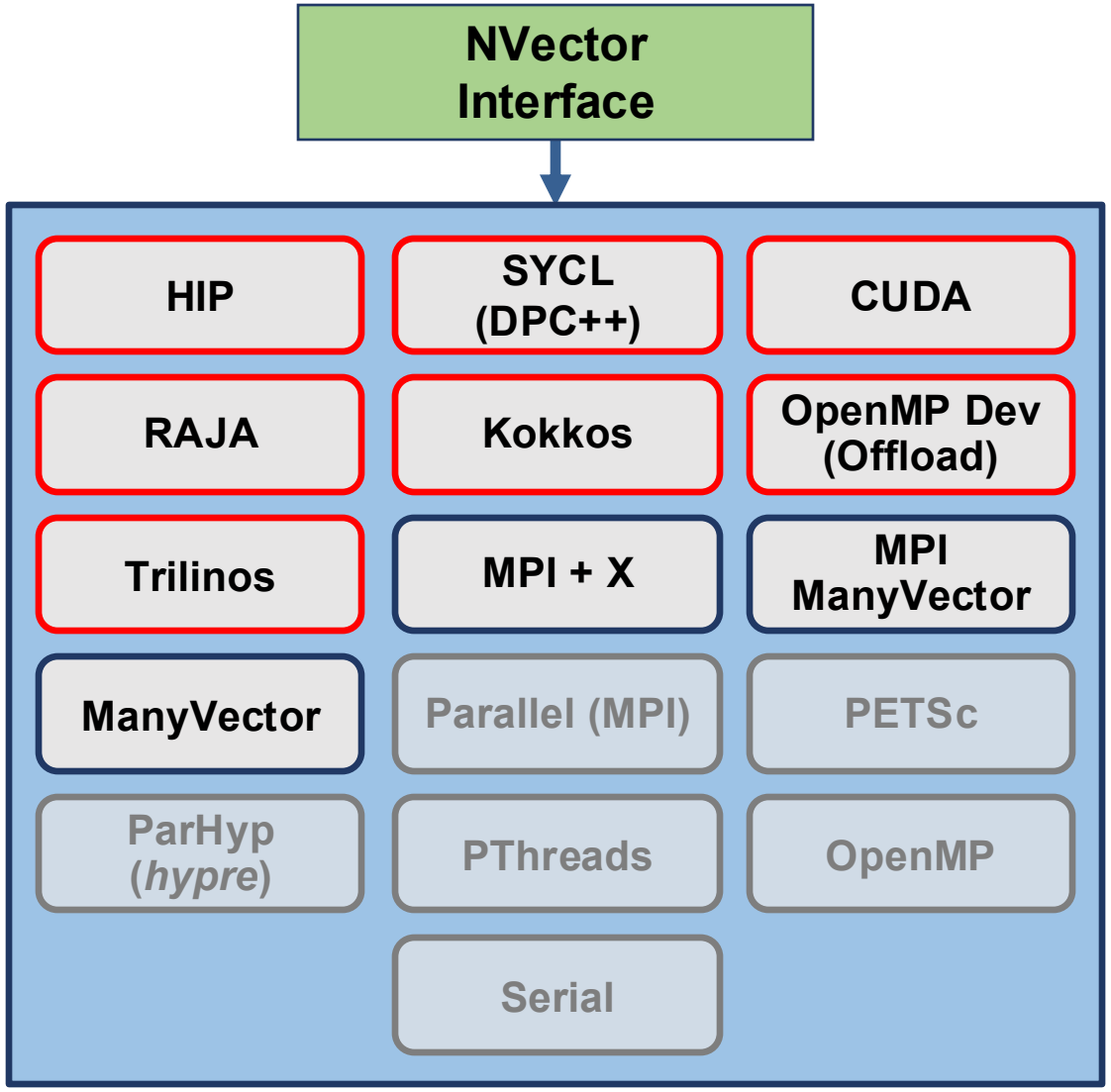
- Native SUNMemoryHelper implementations are provided for **Cuda**, **Hip**, and **Sycl**
- AMReX and MFEM provide SUNMemoryHelpers wrapping their own memory pool / allocators

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

SUNDIALS GPU Enabled Vectors

- SUNDIALS modifies data through vector operations defined by the *NVector interface* (sum, norms, etc.)
- GPU implementations are provided with SUNDIALS:
  - **HIP**, **SYCL**, **CUDA**, **RAJA** *with CUDA, HIP, or SYCL backends*, and **OpenMP DEV** (target offloading)
  - **MPI ManyVector** and **MPI+X** modules enable data partitioning and support for hybrid MPI+X computation
- Many of the native GPU vectors support:
  - Separate host and device or managed (UVM) memory
  - User-defined memory pools (SUNMemoryHelper)
  - User-defined execution policies (ExecPolicy)
- Straightforward to create a vector e.g., AMReX and SAMRAI provide their own NVector implementations



SUNDIALS GPU Enabled Vectors

- All GPU-enabled vectors support separate host and device memory
- Nearly all support managed memory (UVM), the only exception is the OpenMP Dev vector
- Using separate host and device memory offers significantly better performance than UVM and is **recommended for production usage**
- When using separate host and device memory, ***users must manage data coherency!***
  - *SUNDIALS only operates on the device data and* ***never moves it***
  - After receiving control from SUNDIALS, you must call `N_VCopyFromDevice_Cuda` if you want to access the data on the host
  - If you modify data on the host, you must copy it to the device with before passing control back to SUNDIALS using `N_VCopyToDevice_Cuda`

Creating SUNDIALS GPU Vectors

- For the **Cuda**, **Hip**, **Sycl**, **Raja**, or **OpenMPDEV** vectors

```
// Create vector with separate host and device data arrays
N_Vector N_VNew_**(sunindextype length, SUNContext ctx)

// Create vector with a “unified memory” data array
N_Vector N_VNewManaged_**(sunindextype length, SUNContext ctx)
```

- For the **Sycl** vector, the constructor also requires a queue e.g.,

```
N_Vector N_VNew_**(sunindextype length, sycl::queue queue, SUNContext ctx)
```

- For the **Kokkos**, we use a C++ class that is convertible to an NVector and wraps Kokkos views

```
// Create a Kokkos vector using the CUDA execution space
sundials::kokkos::Vector<Kokkos::Cuda> y{length, ctx}
```

Creating SUNDIALS GPU Vectors

- **Cuda**, **Hip**, **Sycl**, **Raja**, or **OpenMPDEV** vectors can also be created from existing data

```
// Create vector from existing separate host and device data arrays
N_Vector N_VMake_**(sunindextype length, sunrealtype* h_data,
                    sunrealtype* d_data, SUNContext ctx)

// Create vector from an existing "unified memory" data array
N_Vector N_VMakeManaged_**(sunindextype length, sunrealtype* uvm_data,
                           SUNContext ctx)
```

- Like before, the Sycl vector also takes the queue as an input

- Kokkos vectors can also be created from an existing view

```
// Create a Kokkos vector using an existing view
sundials::kokkos::Vector<Kokkos::Cuda> y{view, ctx}
```

Accessing and Copying Data with SUNDIALS GPU Vectors

- For the **Cuda**, **Hip**, **Sycl**, **Raja**, or **OpenMPDEV** vectors

```
// Access the host or device data pointer
sunrealtype* h_data = N_VGetArrayPointer(y)
sunrealtype* d_data = N_VGetDeviceArrayPointer(y)

// Copy data To or From the device
N_VCopyToDevice_Cuda(y)
N_VCopyFromDevice_Cuda(y)
```

- For the **Kokkos** vector

```
// Access the host or device view
auto h_view = y.HostView()
auto d_view = y.View()

// Copy data To or From the device
sundials::kokkos::CopyToDevice(y)
sundials::kokkos::CopyFromDevice(y)
```

Brusselator Example: Creating the Initial Condition Vector

```
/* Create CUDA vector for the initial condition */
N_Vector y = N_VNew_Cuda(batchSize * nbatches, sunctx);

/* Initialize y on the host (alternatively could initialize on the device) */
double* ydata = N_VGetArrayPointer(y);

for (int batchj = 0; batchj < udata.nbatches; ++batchj)
{
  ydata[batchj * udata.batchSize]     = udata.u0[batchj];
  ydata[batchj * udata.batchSize + 1] = udata.v0[batchj];
  ydata[batchj * udata.batchSize + 2] = udata.w0[batchj];
}

/* Copy data to the device */
N_VCopyToDevice_Cuda(y);
```

- Alternatively, the vector data can be set by accessing the device pointer and launching a kernel

The SUNDIALS MPI+X Vector

- For multi-node usage, the MPI+X vector adds MPI capabilities to any local (single-node) vector
  - See examples/ida/cuda/idaHeat2D_kry_cuda.cu for an MPI+CUDA example
- Include the MPI+X header i.e., `#include <nvector/nvector_mpiplusx.h>` and construct the MPI+X vector from the local vector

```
/* Create the local and global vectors */
N_Vector y_local  = N_VNew_Cuda(local_length, sunctx);
N_Vector y_global = N_VMake_MPIPlusX(mpi_comm, y_local, sunctx);
```

- Accessing the local vector

```
N_Vector y_local = N_VGetLocalVector_MPIPlusX(y_global);
double* N_VGetDeviceArrayPointer(y_local)
N_VCopyToDevice_Cuda(x)
```

- Remember to destroy both the local and global vector with `N_VDestroy`

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Brusselator Example: Create and Initialize CVODE

```
/* Create CVODE and specify using Backward Differentiation Formula methods */
void* cvode_mem = CVodeCreate(CV_BDF, sunctx);

/* Initialize the integrator */
retval = CVodeInit(cvode_mem, f, T0, y);
```

- With CVODE the initialization function sets
  - The user-defined function for evaluating the ODE right-hand side, $f(t, y)$
  - The initial condition time, t_0
  - The initial condition state, y
  - CVODE makes a copy of y so this vector may be reused later for holding the output state

Brusselator Example: ODE Right-hand Side Function

```
/* Access the data from the input vectors and launches a CUDA kernel to do the
   actual computation */
int f(double t, N_Vector y, N_Vector ydot, void* user_data)
{
  UserData* udata   = (UserData*)user_data;
  double*   ydata   = N_VGetDeviceArrayPointer(y);
  double*   ydotdata = N_VGetDeviceArrayPointer(ydot);

  unsigned block_size = 256;
  unsigned grid_size  = (udata->nbatches + block_size - 1) / block_size;

  f_kernel<<<grid_size, block_size>>>(t, ydata, ydotdata, udata->a.get(),
                                      udata->b.get(), udata->ep.get(),
                                      udata->nbatches, udata->batchSize);
  return 0;
}
```

- In MPI problems, communication (e.g., filling ghost cells) takes place inside the RHS function

Brusselator Example: ODE Right-hand Side Kernel

```
__global__ void f_kernel(double t, double* ydata, double* ydotdata,
                         double* A, double* B, double* Ep, int nbatches,
                         int batchSize)
{
 for (int batchj = blockIdx.x * blockDim.x + threadIdx.x; batchj < nbatches;
     batchj += blockDim.x * gridDim.x)
 {
  double a = A[batchj]; double b = B[batchj]; double ep = Ep[batchj];
  double u = ydata[batchj * batchSize];
  double v = ydata[batchj * batchSize + 1];
  double w = ydata[batchj * batchSize + 2];

  ydotdata[batchj * batchSize] = a - (w + 1.0) * u + v * u * u;
  ydotdata[batchj * batchSize + 1] = w * u - v * u * u;
  ydotdata[batchj * batchSize + 2] = (b - w) / ep - w * u;
 }
}
```

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Brusselator Example: Setting Integrator Tolerances

```
/* Set the vector absolute tolerance */
double* abstol_data = N_VGetArrayPointer(abstol);

for (int batchj = 0; batchj < udata.nbatches; ++batchj)
{
  abstol_data[batchj * udata.batchSize] = 1.0e-10;
  abstol_data[batchj * udata.batchSize + 1] = 1.0e-10;
  abstol_data[batchj * udata.batchSize + 2] = 1.0e-10;
}

N_VCopyToDevice_Cuda(abstol);

/* Specify the scalar relative tolerance and vector absolute tolerances */
retval = CVodeSVtolerances(cvode_mem, reltol, abstol);
```

- Alternatively, the vector data can be set by accessing the device pointer and launching a kernel
- Or we could use a scalar absolute tolerance with `CVodeSStolerances`

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Solving Nonlinear Systems in SUNDIALS Time Integrators

- SUNDIALS implicit time integrators require solving one or more nonlinear systems of the form $F(y) = 0$ or $\mathrm{G}(y) = y$ in each time step
- SUNDIALS provides several nonlinear solver implementations

- The Newton and Fixed-Point solvers inherit their GPU capability from the underlying objects (vectors, matrices, and linear solvers) and user-supplied callback functions e.g., the ODE RHS
- User-defined or problem-specific nonlinear solver modules can be supplied by wrapping the solver as a **SUNNonlinearSolver** implementation
  - See examples/arkode/CXX_parallel/ark_brusselator1D_task_local_nls.cpp for an example utilizing a problem-specific task-local nonlinear solver on GPUs

Solving Linear Systems in SUNDIALS

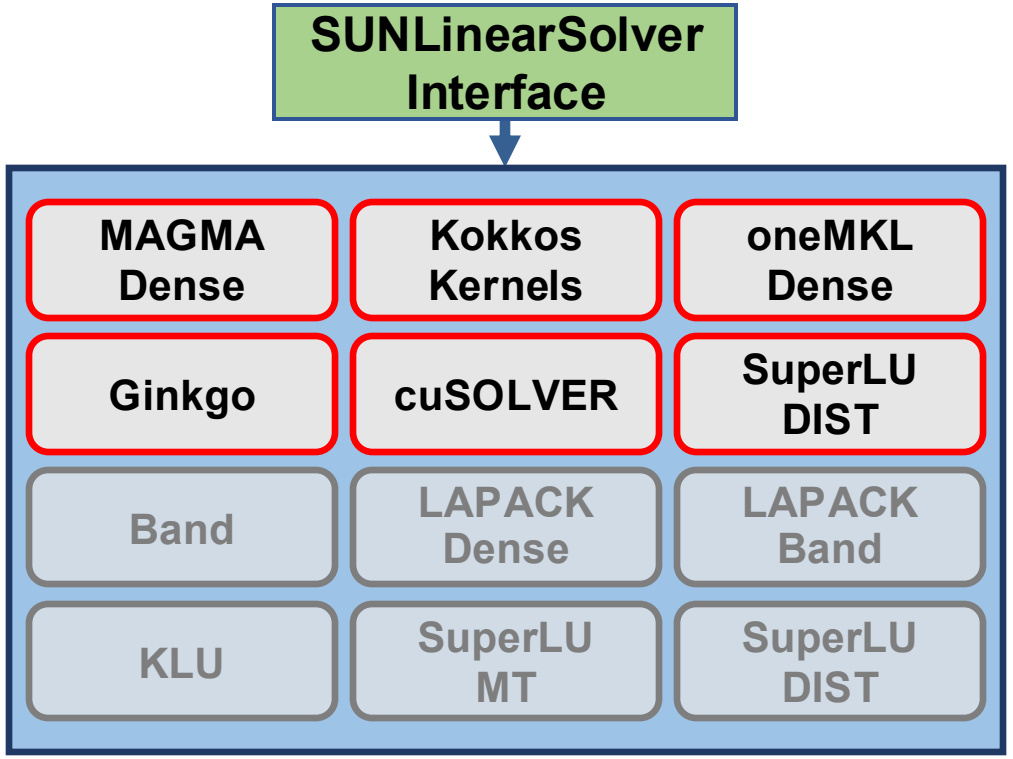
- By default, SUNDIALS integrators use a Newton method which requires a linear solver
- Users need to create and attach a linear solver object and, if necessary, a matrix object
- SUNDIALS provides several GPU-ready linear solver implementations/interfaces
  - **Iterative:** SUNDIALS' matrix-free iterative (Krylov) linear solvers inherit their GPU capability from the vector utilized and user-supplied functions e.g., the ODE RHS, preconditioner, etc.
  - **Direct:** SUNDIALS provides interfaces to linear solver libraries with support for AMD, Intel, and NVIDIA GPUs
- User-defined or problem-specific linear solver modules can be supplied by wrapping the solver as a **SUNLinearSolver** implementation
  - See examples/cvode/CXX_parhyp/cv_heat2D_hypre_ls.cpp for an example wrapping a linear solver from the *hypre* library

Interfaces to GPU Enabled External Linear Solvers

- Several of the external libraries provide solvers for batched systems e.g., the block diagonal systems that arise when solving independent systems together

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_1} & 0 & \cdots & 0 \\ 0 & \mathbf{A_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_n} \end{bmatrix}$$

- All blocks A_j must be the same size (and have the same sparsity pattern with sparse solvers)
- Dense blocks A_j
  - **MAGMA** interface supports HIP and CUDA
  - **Kokkos Kernels** HIP / CUDA / SYCL
  - **oneMKL** interface supports SYCL
- Sparse blocks A_j, **cuSPARSE** interface (CUDA)
- **Ginkgo** for sparse or dense solvers – HIP / CUDA / SYCL



Each of the above linear solvers also has a compatible SUNMatrix object e.g., dense, MAGMA dense, sparse, Ginkgo, etc.

Brusselator Example: Creating and Attaching a Linear Solver Object

```
/* Create a matrix and linear solver */
SUNMatrix A = SUNMatrix_MagmaDenseBlock(udata.nbatches, udata.batchSize,
                                        udata.batchSize, SUNMEMTYPE_DEVICE,
                                        memhelper, NULL, sunctx);

SUNLinearSolver LS = SUNLinSol_MagmaDense(y, A, sunctx);

/* Attach the matrix and linear solver to CVODE */
retval = CVodeSetLinearSolver(cvode_mem, LS, A);

/* Set the user-supplied Jacobian routine Jac */
retval = CVodeSetJacFn(cvode_mem, Jac);
```

- Alternatively, this example can also use GMRES to solve the linear systems

```
/* Create and attach GMRES solver (matrix-free) */
SUNLinearSolver LS = SUNLinSol_SPGMR(y, SUN_PREC_NONE, 0, sunctx);

retval = CVodeSetLinearSolver(cvode_mem, LS, NULL);
```

Brusselator Example: ODE Jacobian Function

```
/* Access the data from the input vectors and launch a CUDA kernel to do the
   actual computation */
int Jac(double t, N_Vector y, N_Vector fy, SUNMatrix J, void* user_data,
        N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{
  UserData* udata = (UserData*)user_data;
  double*   Jdata = SUNMatrix_MagmaDense_Data(J);
  double*   ydata = N_VGetDeviceArrayPointer(y);

  unsigned block_size = 32;
  unsigned grid_size  = (udata->nbatches + block_size - 1) / block_size;

  j_kernel<<<grid_size, block_size>>>(ydata, Jdata, udata->a.get(),
                                      udata->b.get(), udata->ep.get(),
                                      udata->nbatches, udata->batchSize);

  return 0;
}
```

Brusselator Example: ODE Jacobian Kernel

```
__global__ void j_kernel(double* ydata, double* Jdata, double* A, double* B,
                         double* Ep, int nbatches, int batchSize)
{
 int N = batchSize; int NN = N * N;

 for (int batchj = blockIdx.x * blockDim.x + threadIdx.x; batchj < nbatches;
     batchj += blockDim.x * gridDim.x)
 {
  double ep = Ep[batchj];          double u = ydata[N * batchj];
  double v = ydata[N * batchj + 1]; double w = ydata[N * batchj + 2];

  /* first col of block */
  Jdata[NN * batchj] = -(w + 1.0) + 2.0 * u * v;
  Jdata[NN * batchj + 1] = u * u;
  Jdata[NN * batchj + 2] = -u;
  ...
 }
}
```

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Brusselator Example: Optional “Set” Functions

- In this example, we attach a “user data” pointer to access problem specific data inside the callback functions for computing the right-hand side and Jacobian

```
/* Attach a pointer to the user-defined data structure */
retval = CVodeSetUserData(cvode_mem, &udata);
```

- Various other “Set” functions could be called here to adjust the behavior of the integrator or enable advanced options
  - Modify method or adaptivity options
  - Adjust algebraic solver settings
  - Enable rootfinding
  - Enable constraint handling
  - etc.

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Brusselator Example: Advancing the System in Time

```
double tout = T0 + dTout;

for (int iout = 0; iout < Nt; iout++)
{
  /* Advance in time and return the solution at tout */
  retval = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
  if (retval < 0) break;

  /* Output some solution components */
  N_VCopyFromDevice_Cuda(y)
  for (int batchj = 0; batchj < udata.nbatches; batchj += 10)
  {
    ...
  }

  tout += dTout;
  tout = (tout > Tf) ? Tf : tout;
}
```

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get optional integrator/solver statistics
10. Free integrator and objects

Brusselator Example: Getting Integrator Statistics

```
/* Get and print some final statistics */
long int nst, netf, nfe, nsetups, nje, nni, ncfn;


retval = CVodeGetNumSteps(cvode_mem, &nst);

retval = CVodeGetNumErrTestFails(cvode_mem, &netf);

retval = CVodeGetNumRhsEvals(cvode_mem, &nfe);

retval = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);

retval = CVodeGetNumJacEvals(cvode_mem, &nje);

retval = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);

retval = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
```

- Alternatively, `CVodePrintAllStats` can be used to output all of the integrator stats as well as some derived metrics e.g., nonlinear iterations per step, linear iterations per nonlinear, etc.

SUNDIALS Time Integrator Program Skeleton

1. Create a SUNDIALS Context object
2. Create a SUNMemoryHelper *(optional)*
3. Create an NVector and fill it with the initial condition
4. Create and initialize the time integrator
5. Set the integrator tolerances
6. Create and attach algebraic solver objects *(if necessary)*
7. Set optional integrator/solver inputs
8. Advance the solution in time
9. Get integrator/solver statistics
10. Free integrator and objects

Free Integrator and Other Objects

```
/* Free vectors */
N_VDestroy(y);
N_VDestroy(abstol);

/* Free the matrix */
SUNMatDestroy(A);

/* Free the linear solver */
SUNLinSolFree(LS);

/* Free the integrator */
CVodeFree(&cvode_mem);
```

- The order in which the objects are freed is not important
- Because we used the C++ context class, the object is destroyed when it goes out of scope

Tutorial Outline

- Overview of SUNDIALS (Dan)
- How to use the time integrators (Dan)
- Using SUNDIALS on GPU-based HPC platforms (David)
- **Brief: How to download and install SUNDIALS (David)**
- Closing Remarks (Dan)

Acquiring SUNDIALS

- Download the tarball from the SUNDIALS website
  - computing.llnl.gov/projects/sundials/sundials-software
  - Latest (v7.1.1), archived versions, individual packages (e.g., CVODE) available
  - Most configurable
- Clone from the SUNDIALS GitHub repo
  - github.com/LLNL/sundials
  - Also has tarballs of the latest and archived versions available
  - Most configurable
- Install with Spack
  - `spack install sundials`
  - Latest and recent versions available
  - Highly configurable via spack variants e.g., `spack install sundials +cuda`

Installing SUNDIALS with CMake

- When installing from source, build SUNDIALS with the standard CMake procedure

```
# clone the SUNDIALS repo
> git clone git@github.com:LLNL/sundials.git
> cd sundials

# create a build directory
> mkdir build
> cd build

# configure e.g., with MPI and CUDA enabled
> cmake ../. \
  –D CMAKE_INSTALL_PREFIX=<prefix> \
  –D ENABLE_MPI=ON -D ENABLE_CUDA=ON \
  –D CMAKE_CUDA_ARCHITECTURES=80

# build and install
> make –j 8
> make install
```

- See the online documentation for a complete list of options
- Some common options:
  - `CMAKE_<lang>_COMPILER` the compiler to use (default to automatically determine from the user environment)
  - `SUNDIALS_PRECISION` the floating-point precision to use (default is double)
  - `SUNDIALS_INDEX_SIZE` use 32 or 64-bit indexing, must match external solver libraries (default is 64-bit)
  - `BUILD_FORTRAN_MODULE_INTERFACE` enable the F2003 module interfaces (default is OFF)

Installing SUNDIALS with Spack

Spack

- Spack (see https://spack.io/) is an easy way to install SUNDIALS
- The SUNDIALS team maintains a Spack package that allows installing with one command

```
spack install sundials
```

- To see the configuration and dependencies Spack will install use the command

```
spack spec sundials
```

- To see the available Spack "variants" for configuring the install use the command

```
spack info sundials
```

- SUNDIALS with MPI and CUDA or HIP enabled can be installed with the command:

```
spack install sundials +mpi +cuda cuda_arch=80
```
(NVIDIA A100)

```
spack install sundials +mpi +rocm amdgpu_target=gfx90a
```
(AMD MI250X)

Linking to SUNDIALS in an Application Code

- For CMake projects, SUNDIALS provides a `SUNDIALSConfig.cmake` configuration file with imported targets for each module

```
# When configuring, set the variable SUNDIALS_DIR to the install location i.e.,
#   cmake -D SUNDIALS_DIR=/path/to/sundials/installation
find_package(SUNDIALS REQUIRED)
add_executable(myexec main.cpp)

# Link to SUNDIALS libraries through the appropriate exported targets
target_link_libraries(myexec PUBLIC SUNDIALS::cvode SUNDIALS::nveccuda)
```

- If using Makefiles files
  - Headers are installed in subdirectories under `<prefix>/include`
  - Libraries are installed in `<prefix>/lib` or `<prefix>/lib64`
  - A list all the installed files is available in the installation guide
- With Spack, load SUNDIALS with `spack load` or get the path with `spack location –i`

Getting Help with Building and Using SUNDIALS

- The online documentation includes an in-depth installation guide:
  - See the “Getting started” section: sundials.readthedocs.io/en/latest/sundials
  - The tarballs also have this information in `INSTALL_GUIDE.pdf`
- These include details on configuring with CMake and every possible SUNDIALS CMake option
- Users can post queries to GitHub, the sundials-users email list, and check the list archive or FAQ:
  - GitHub issues: github.com/LLNL/sundials/issues
  - Mailing list: computing.llnl.gov/projects/sundials/mailing-list
  - List archive: groups.google.com/g/sundials-users
  - FAQ: computing.llnl.gov/projects/sundials/faq

Tutorial Outline

- Overview of SUNDIALS (Dan)
- How to use the time integrators (Dan)
- Using SUNDIALS on GPU-based HPC platforms (David)
- Brief: How to download and install SUNDIALS (David)
- **Closing Remarks (Dan)**

Where to learn more and get the software

- Visit the SUNDIALS website
  https://computing.llnl.gov/projects/sundials
- SUNDIALS tutorials: At top of presentations list at the SUNDIALS publications page: https://computing.llnl.gov/projects/sundials/publications
- Download from the SUNDIALS website:
  https://computing.llnl.gov/projects/sundials/sundials-software
- Download the tarball from the SUNDIALS GitHub page:
  https://github.com/LLNL/sundials/releases
- Install SUNDIALS using Spack "spack install sundials"
- View online documentation on all SUNDIALS packages at readthedocs.org:
  https://sundials.readthedocs.io

Where to learn more

- Visit the SUNDIALS Documentation: https://sundials.readthedocs.io
- Visit the SUNDIALS GitHub: https://github.com/LLNL/sundials
- Visit the SUNDIALS website at LLNL: https://computing.llnl.gov/projects/sundials
- Where to get this tutorial: SUNDIALS Publications page (bottom), https://computing.llnl.gov/projects/sundials/publications
  - This page also includes prior tutorials on the basic uses of SUNDIALS

sundials

computing.llnl.gov/sundials

github.com/LLNL/sundials

sundials.readthedocs.io

Lawrence Livermore National Laboratory

SMU