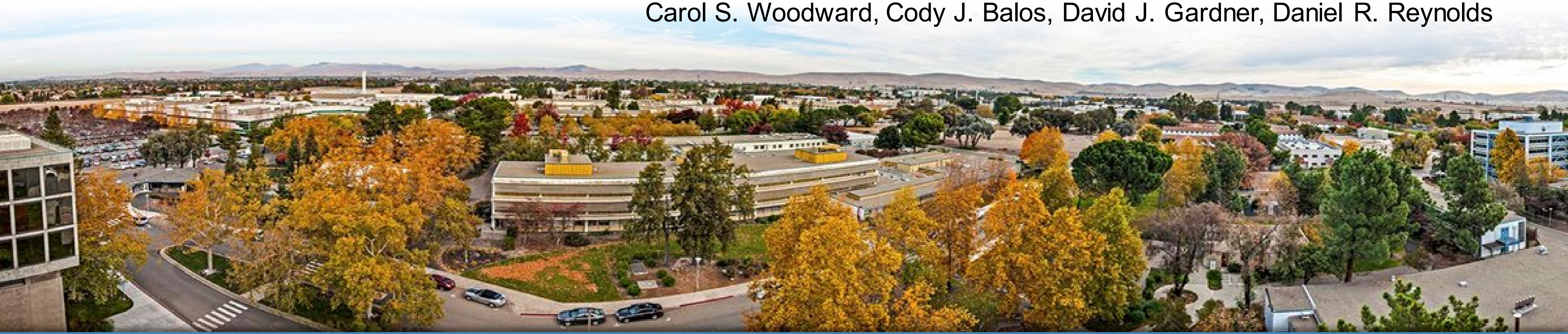
Overview and Use of New Features in the SUNDIALS Suite of Nonlinear and Differential/Algebraic Equation Solvers

ECP Annual Meeting

May 2, 2022

Carol S. Woodward, Cody J. Balos, David J. Gardner, Daniel R. Reynolds



LLNL-PRES-834716
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

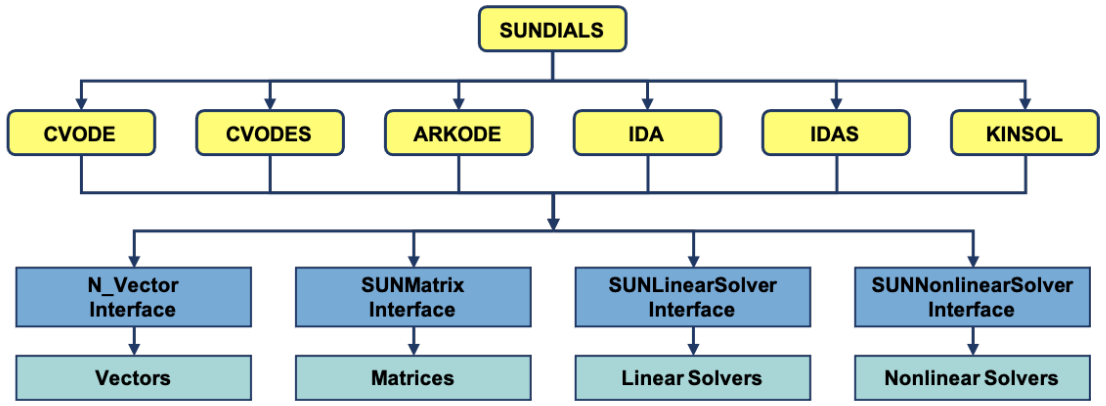
Tutorial Outline

- **Introduction (Carol Woodward)**
- Multirate time integrators (Daniel Reynolds)
- Enhanced GPU support (David Gardner)
- Performance profiling, analysis, and logging (Cody Balos)
- Scalable demonstration code (Daniel Reynolds)
- Closing Remarks (Carol)

- Where to get this tutorial: SUNDIALS/hypre ECP Project Confluence (Under Software Technologies/2.3.3.12) Tutorials page: https://confluence.exascaleproject.org/display/STLM12/Tutorials

SUite of Nonlinear and DIfferential-ALgebraic Solvers

- SUNDIALS is a software library consisting of ODE and DAE integrators and nonlinear solvers
- Packages: CVODE(S), IDA(S), ARKODE, KINSOL
- Written in C with interfaces to Fortran (Python coming soon)
- Designed to be incorporated into existing codes
- Through the ECP, developing a rich infrastructure of support on exascale systems and applications
- Freely available; released under the BSD 3-Clause license (>100,000 downloads in 2021)
- Active user community supported by sundials-users email list
- Detailed user manuals included with each package (and at https://sundials.readthedocs.io)

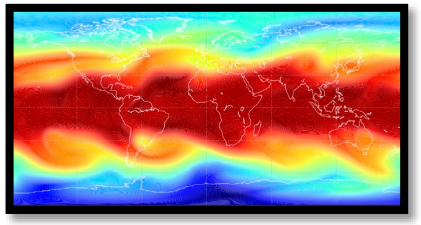


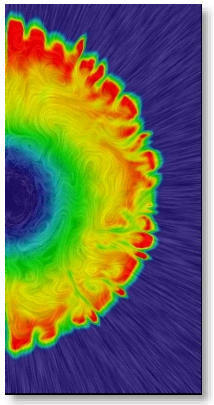
- Nonlinear and linear solvers and all data use is fully encapsulated from the integrators and can be user-supplied
- All parallelism is encapsulated in vector & solver modules and user-supplied functions

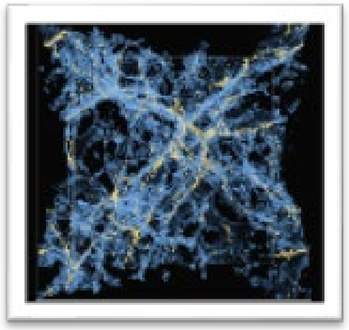
https://computing.llnl.gov/casc/sundials

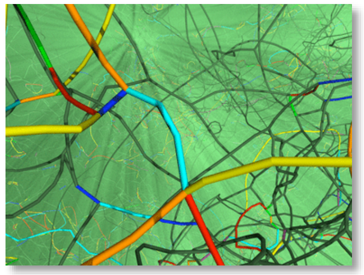
SUNDIALS: Used Worldwide in Applications from Research & Industry

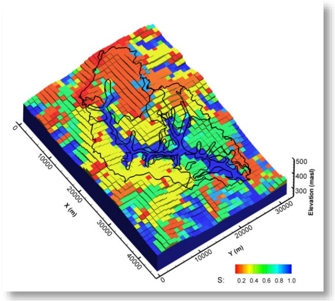
- Computational Cosmology (Nyx)
- Combustion (PELE)
- Atmospheric dynamics (DOE E3SM)
- Fluid Dynamics (NEK5000) (ANL)
- Dislocation dynamics (LLNL)
- 3D parallel fusion (SMU, U. York, LLNL)
- Power grid modeling (RTE France, ISU, LLNL)
- Sensitivity analysis of chemically reacting flows (Sandia)
- Large-scale subsurface flows (CO Mines, LLNL)
- Micromagnetic simulations (U. Southampton)
- Chemical kinetics (Cantera)
- Released in third party packages:
  - Red Hat Extra Packages for Enterprise Linux (EPEL)
  - SciPy – python wrap of SUNDIALS
  - Cray Third Party Software Library (TPSL)


Atmospheric Dynamics


Core collapse supernova


Cosmology


Dislocation dynamics


Subsurface flow

SUNDIALS offers packages with linear multistep and multistage methods

- CVODE, IDA, and sensitivity analysis variants (forward and adjoint), CVODES and IDAS, are based on 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 is designed to work as an infrastructure for developing adaptive one-step, multistage time integration methods
  - Originally designed to solve $M\dot{y} = f_I(t,y) + f_E(t,y), \quad y(t_0) = y_0$

    $M(t)$ may be the identity or any nonsingular (and optionally time-dependent) mass matrix (e.g., FEM)
  - Multistage embedded methods give rise to adaptive time steps
  - Three steppers: ARKStep (explicit, implicit, and additive ImEx Runge-Kutta methods), ERKStep (streamlined explicit RK implementation), and MRIStep (multirate infinitesimal step methods)
  - Paralell-in-Time support: Xbraid wrappers for SUNDIALS vectors and explicit, implicit, and IMEX methods in ARKStep

- KINSOL solves nonlinear algebraic systems with Newton or accelerated fixed point methods

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

- Time step selection
  - Based on the method, estimate the time step error
  - Accept step if $||E(\Delta t)||_{WRMS} < 1$; Reject it otherwise

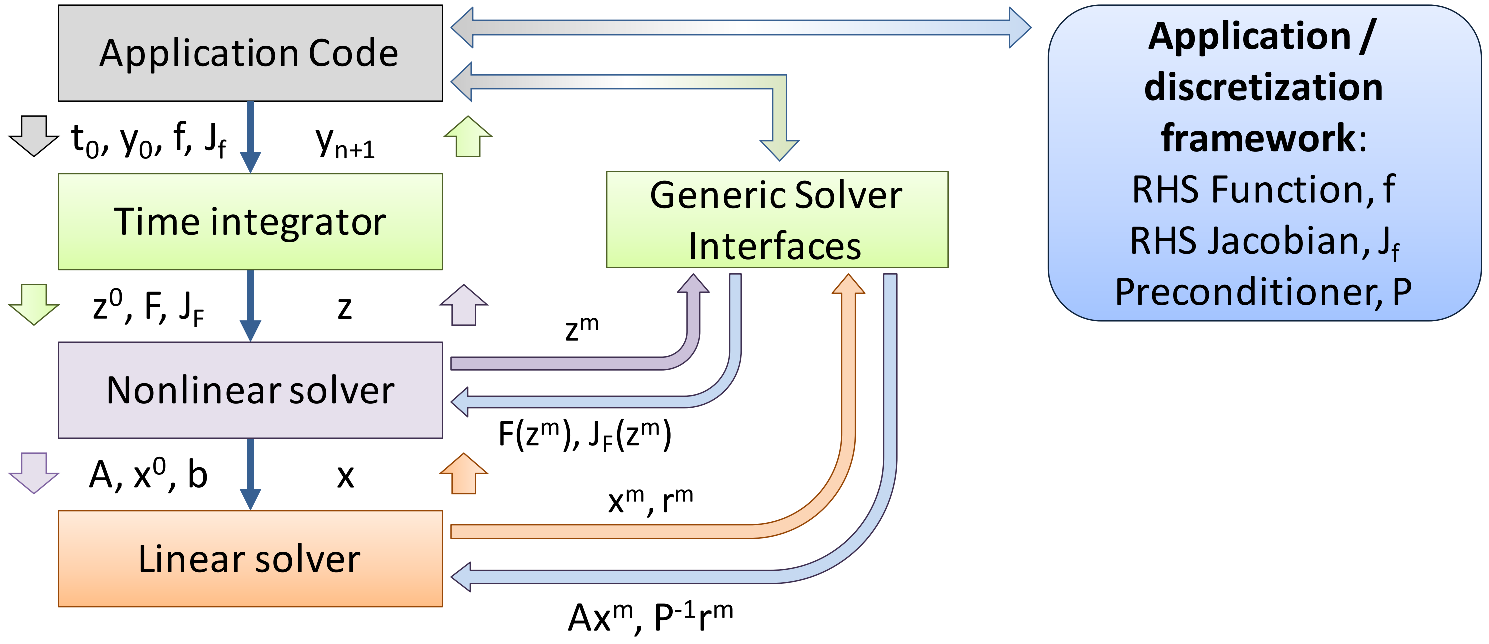
$$\|y\|_{\text{wrms}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(w_i\ y_i)^2} \qquad w_i = \frac{1}{RTOL|y_i| + ATOL_i}$$

  - Choose next step, $\Delta t'$, so that $||E(\Delta t')||_{WRMS} < 1$
- CVODE and IDA also adapt order
  - Choose next order resulting in largest step meeting error condition

- Relative tolerance (RTOL) controls local error relative to the size of the solution
  - RTOL = 10^{-4} means that errors are controlled to 0.01%
- Absolute tolerances (ATOL) control error when a solution component may be small
  - Ex: solution starting at a nonzero value but decaying to noise level, ATOL should be set to noise level

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



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

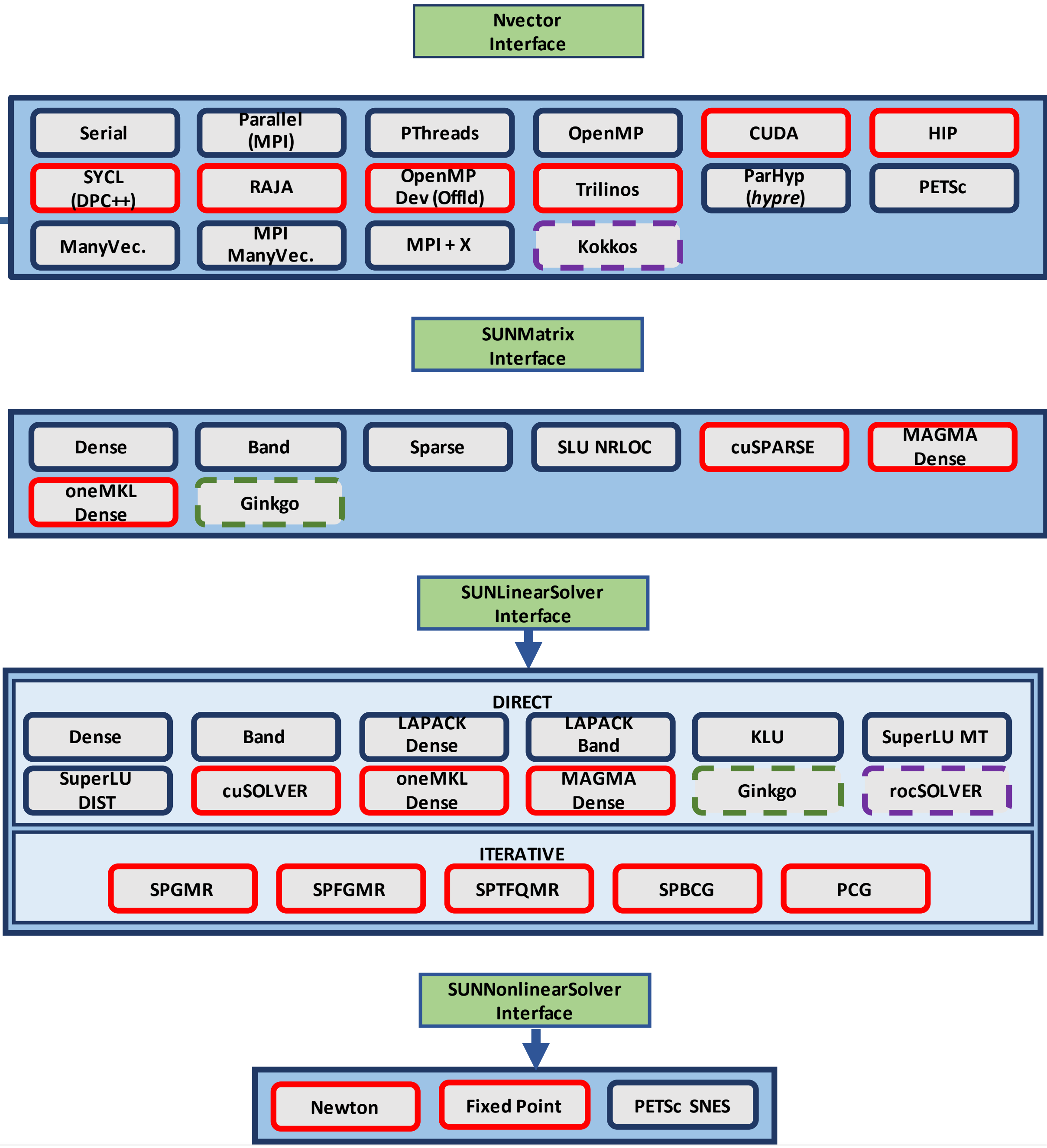
- Nonlinear and linear solver modules are designed for generic systems

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

Status in pre-exascale environments

- SUNDIALS supports AMD, Intel, and NVIDIA GPUs
- Vector implementations using CUDA, HIP, SYCL, OpenMP offload, and RAJA (with CUDA, HIP, or SYCL backends)
- Iterative nonlinear and matrix-free linear (Krylov) solvers inherit GPU support from vectors and user-defined functions
- Interfaces to MAGMA (CUDA and HIP) and oneMKL (DPC++) for dense batched LU linear solvers
- Interface to cuSOLVER for batched sparse QR linear solver
- SUNMemoryHelper class enables application supplied allocators under SUNDIALS objects
- Performance profiling and instrumentation layer
- Benchmark problems utilizing CUDA, HIP, and RAJA incorporated into LLNL GitLab CI for automated performance testing
- Installation via Spack with smoke tests for CUDA, HIP, and SYCL
- OLCF now has install of SUNDIALS on Spock with HIP enabled

Blue indicates new in the last year



What’s new in SUNDIALS?

- High-order multirate methods that can integrate different portions of the problem with different time step sizes
  - Implicit and IMEX at the slow scale
  - Custom integrators for the fast scale
- New vector and solver support for SYCL-based applications
  - Direct SYCL and RAJA with SYCL backend vectors
  - OneMKL dense solve support
- Support for logging more run diagnostic information (extremely helping in debugging and better understanding performance)
- Performance profiling layer with optional use of Caliper
- Added the ability for CVODES to project the solution onto an invariant manifold as the solution is evolved
- New online documentation: https://sundials.readthedocs.io
- Moved development repo fully to GitHub: https://github.com/LLNL/sundials

What are we working on now?

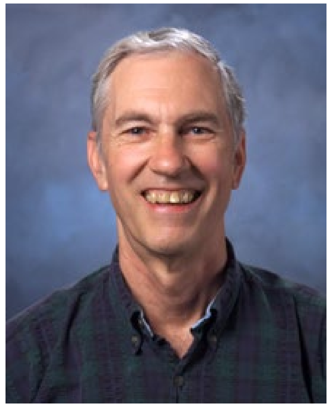
- Greater support on AMD and Intel GPUs
  - Optimizations for HIP and SYCL vectors
  - Interfaces to more batched solvers – Gingko, ROCm, MKL batched solvers
- Python interfaces for CVODE, ARKODE, IDA, and KINSOL
- More multirate methods and options
- Greater interoperability to discretization packages
  - AMReX – multifab-based vector for SUNDIALS
  - Chombo – Chombo vector for SUNDIALS
  - MFEM – integrators already available from MFEM, new GPU-based examples
  - PETSc – updating interfaces to SUNDIALS integrators from PETSc

SUNDIALS Team

Current Team:

Cody Balos

David Gardner


Alan Hindmarsh

Dan Reynolds

Steven Roberts

Carol Woodward

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, 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

Folks with red outlines are part of the ECP time integration effort

Tutorial Outline

- Introduction (Carol Woodward)
- **Multirate time integrators (Daniel Reynolds)**
- Enhanced GPU support (David Gardner)
- Performance profiling, analysis, and logging (Cody Balos)
- Scalable demonstration code (Daniel Reynolds)
- Closing Remarks (Carol)

Multirate Time Integration

- Multirate methods consider a general initial-value problem of the form:

$$\dot{y}(t) = f^S(t,y) + f^F(t,y), \quad t \in (t_0, t_f], \quad y(t_0) = y_0$$

  – $f^S(t,y)$ contains the "slow" dynamics, evolved with a time step H.
  – $f^F(t,y)$ contains the "fast" dynamics, evolved with smaller time steps $h \ll H$.

- Historically, such problems have been treated using low-order operator splitting methods:
  – Lie—Trotter computes $y_n \to y_{n+1}$ via

$$\begin{aligned} \dot{y}^{\{1\}}(t) &= f^{\{1\}}\left(t, y^{\{1\}}\right), \quad t \in (t_n, t_{n+1}], \quad y^{\{1\}}(t_n) = y_n, \\ \dot{y}^{\{2\}}(t) &= f^{\{2\}}\left(t, y^{\{2\}}\right), \quad t \in (t_n, t_{n+1}], \quad y^{\{2\}}(t_n) = y^{\{1\}}(t_{n+1}), \\ y_{n+1} &= y^{\{2\}}(t_{n+1}). \end{aligned}$$

  – Strang—Marchuk symmetrizes this loose "initial-condition" coupling to achieve 2nd order.

Higher-Order "Infinitesimal" Multirate Methods (MIS/MRI)
[Schlegel et al. 2009; Sandu 2019; Chinomona & R. 2021]

- Multirate infinitesimal step (MIS or MRI) methods arose in the numerical weather prediction community, but have seen dramatic advances in recent years.

- Fast time scale is again evolved using any desired solver (of sufficient accuracy).

- Slow time scale is advanced through solving a sequence of modified "fast" initial-value problems.

- These achieve higher order (3rd or even 4th) through:
  - initial condition coupling (as with Lie—Trotter and Strang—Marchuk), *and*
  - temporal interpolation of slow information ($f^S(t,y)$) onto the fast time scale, through the modifications to each fast IVP.

- Extremely efficient – higher order is attainable with *only a single traversal of* $(t_n, t_{n+1}]$, unlike extrapolation or deferred correction approaches that bootstrap Lie—Trotter or Strang—Marchuk to higher order at significantly higher cost.

MRI Method Skeleton

A single step $y_n \to y_{n+1}$ of size $H = t_{n+1} - t_n$ proceeds as:

1. Let: $z_1 = y_n$.
2. For each slow stage $z_i,\ i = 2, \ldots, s$:
   a) Define: $r_i(\tau) = \sum_{j=1}^{i} \gamma_{i,j}\left(\frac{\tau}{(c_i - c_{i-1})H}\right) f^S(t_n + c_j H, z_j)$.
   b) Evolve: $\dot{v}(\tau) = f^F(t_n + \tau, v) + r_i(\tau),\ \text{for } \tau \in (c_{i-1}H, c_i H],\ v(c_{i-1}H) = z_{i-1}$.
   c) Let: $z_i = v(c_i H)$.
3. Let: $y_{n+1} = z_s$.

- $\gamma_{i,j}(\theta)$ is a polynomial in θ, defined by coefficients that satisfy underlying order conditions.
- When $c_i = c_{i-1}$ step 2b reduces to a standard ERK/DIRK Runge—Kutta stage update.
- Implicitness at the slow time scale depends on the “diagonal” $\gamma_{i,i}(\theta)$, typically only used when $c_i = c_{i-1}$.

MRI Methods in SUNDIALS

- ARKODE's MRIStep module additionally supports ImEx treatment of the slow time scale:

$$\dot{y}(t) = f^I(t,y) + f^E(t,y) + f^F(t,y), \quad t \in (t_0, t_f], \quad y(t_0) = y_0.$$

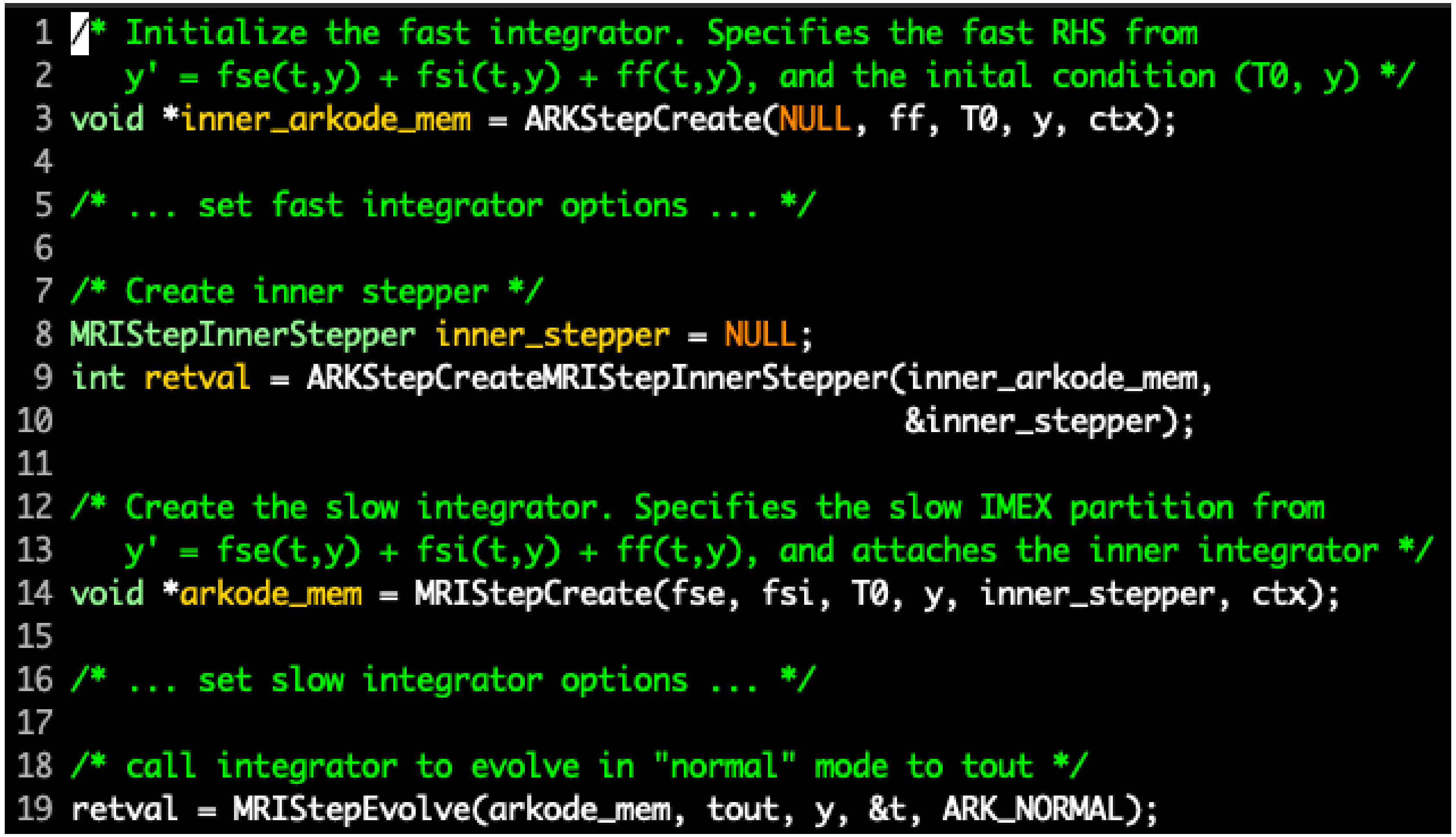
  where both $f^I(t,y)$ & $f^E(t,y)$ are evolved with the large step size H.

- The slow time scale may be handled using explicit, implicit, or ImEx MRI-GARK methods, with orders of accuracy from 2nd through 4th. Additionally supports user-provided MRI-GARK or IMEX-MRI-GARK tables $\{\Gamma^{\{k\}}, \Omega^{\{k\}}\}$.

- Slow time scale requires a user-defined H that can be varied between steps. The fast time scale can be evolved using ARKStep or any viable user-supplied IVP solver.

- Robust multirate adaptivity (H and h) is under development [Fish & R., arXiv:2202.10484, 2022].

MRI Code Example (from ark_brusselator1D_imexmri.c)



```c
/* Initialize the fast integrator. Specifies the fast RHS from
   y' = fse(t,y) + fsi(t,y) + ff(t,y), and the inital condition (T0, y) */
void *inner_arkode_mem = ARKStepCreate(NULL, ff, T0, y, ctx);

/* ... set fast integrator options ... */

/* Create inner stepper */
MRIStepInnerStepper inner_stepper = NULL;
int retval = ARKStepCreateMRIStepInnerStepper(inner_arkode_mem,
                                              &inner_stepper);

/* Create the slow integrator. Specifies the slow IMEX partition from
   y' = fse(t,y) + fsi(t,y) + ff(t,y), and attaches the inner integrator */
void *arkode_mem = MRIStepCreate(fse, fsi, T0, y, inner_stepper, ctx);

/* ... set slow integrator options ... */

/* call integrator to evolve in "normal" mode to tout */
retval = MRIStepEvolve(arkode_mem, tout, y, &t, ARK_NORMAL);
```

- We request a DIRK method from ARKStep for the fast [reaction] time scale (NULL explicit RHS, ff implicit RHS).
- This utility routine wraps the ARKStep integrator as an “inner” stepper for MRIStep.
- We request an IMEX-MRI-GARK method at the slow scale [advection + diffusion].
- We “evolve” the IVP as normal for SUNDIALS integrators.

Additional MRI Comments

- Custom “inner” integrators have a simple API:
  - Required: a routine to *evolve* the fast IVP system over an interval (t_0,t_f) with a given initial condition, $v(t_0)$.
  - Required: a routine to *evaluate* the fast RHS function $f^F(t,v)$ [for MRIStep dense output].
  - Optional: a routine to *reset* the inner integrator’s internal data to a given state, $(t_R,v(t_R))$ [called before the *evolve* routine to set the initial condition].
- The example program `examples/arkode/CXX_parallel/ark_diffusion_reaction_p.cpp` even wraps CVODE as a custom inner integrator for MRIStep.
- **Note**: I will also discuss another multirate example at the end of the tutorial, when discussing our scalable demonstration code.

Tutorial Outline

- Introduction (Carol Woodward)
- Multirate time integrators (Daniel Reynolds)
- **Enhanced GPU support (David Gardner)**
- Performance profiling, analysis, and logging (Cody Balos)
- Scalable demonstration code (Daniel Reynolds)
- Closing Remarks (Carol)

SUNDIALS Supports AMD, Intel, and NVIDIA GPUs

- SUNDIALS' object-oriented design enables supporting various GPUs with class implementations targeting different programming models e.g., HIP, SYCL, CUDA, RAJA, etc.

- To leverage GPU acceleration:
  – Compile SUNDIALS with GPU features enabled e.g., ENABLE_HIP=ON
  – 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

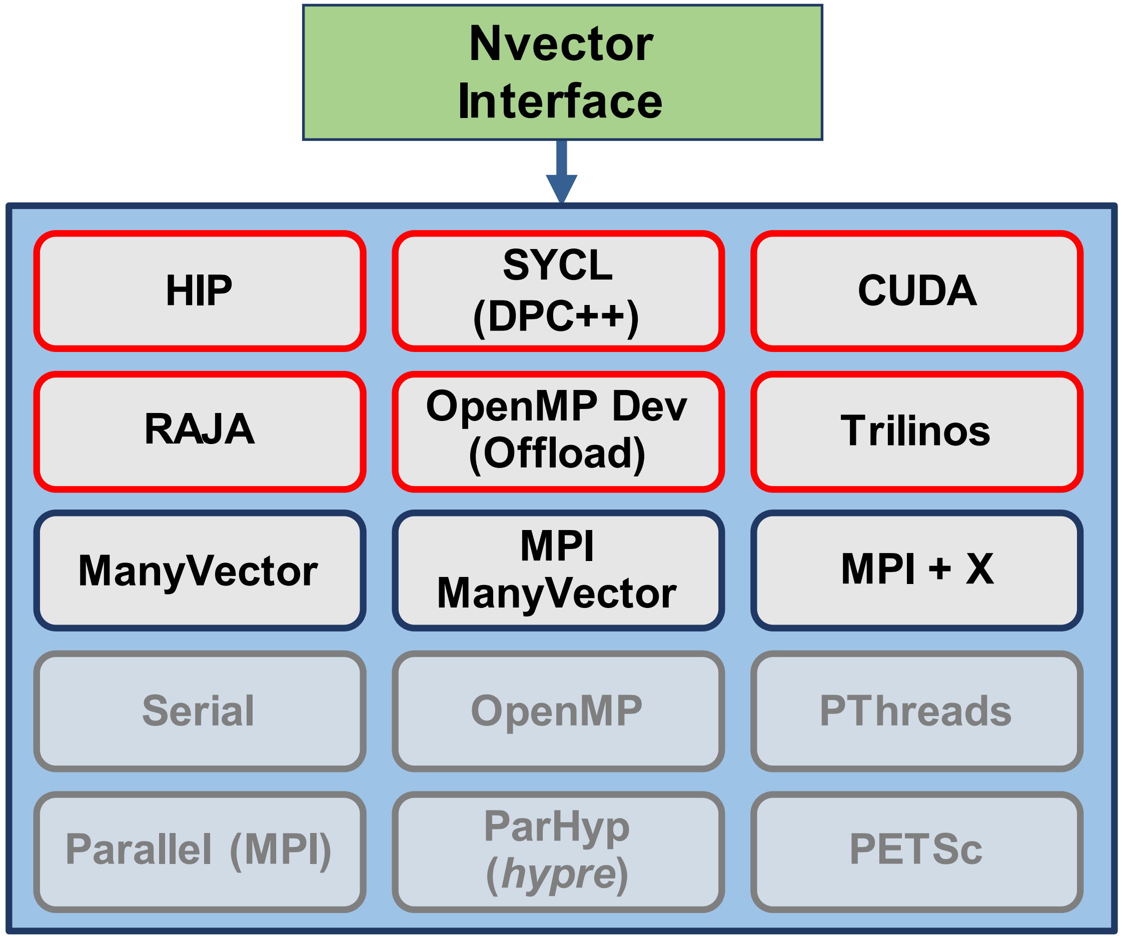
- Primary uses cases:
  – SUNDIALS controls the ***main time-integration*** *loop*, and evolves a large ODE system in a distributed manner (MPI+X) e.g., FEM, FD, or FV applications
  – SUNDIALS is used as a ***local integrator*** for numerous independent subsystems within a larger problem e.g., local reactions in each grid cell within an adaptive mesh refinement application

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 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)
  - **ManyVector** and **MPIPlusX** 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 allocators (SUNMemory API)
  - User-defined execution policies (ExecPolicy)
- Straightforward to create a vector e.g., AMReX and SAMRAI provide their own NVector implementations



Creating GPU Vectors

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

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

// Create vector with a UVM data array
N_Vector N_VNewManaged_**(sunindextype length, SUNContext ctx);

// Create vector from an existing UVM data array
N_Vector N_VMakeManaged_**(sunindextype length, sunrealtype* umv_data,
                   SUNContext ctx);
```

- Here ** is the vector implementation name i.e., **Hip**, **Sycl**, **Cuda**, **Raja**, or **OpenMPDEV**
- Note: SYCL functions include an addition SYCL queue input and the OpenMPDEV vector currently does not support UVM i.e., separate host and device memory must be used

Creating Vectors with a User-defined Allocator and the SUNMemory API

```
// Create vector with a user-supplied memory allocator
N_Vector N_VNewWithMemHelp_**(sunindextype length, sunbooleantype managed,
                              SUNMemoryHelper helper, SUNContext ctx);
```

- A **SUNMemory** object contains a void* data pointer, memory type, and ownership flag
- The **SUNMemoryHelper** base class provides the following operations

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 **Hip**, **Sycl**, and **Cuda** (** above)
- AMReX and MFEM use the SUNMemory API to leverage their own memory tools under SUNDIALS

Creating and Attaching GPU Execution Policies to Vectors

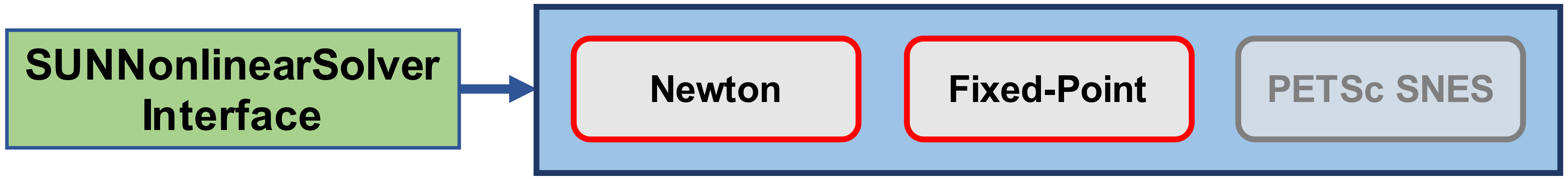
- The HIP, SYCL, and CUDA vectors support attaching **ExecPolicy** objects for determining kernel launch parameters, setting GPU streams, and selecting reduction algorithms (HIP and CUDA only)
- Setting a GPU stream enables concurrent kernel execution (beneficial when running **multiple integrator instances**) and the reduction algorithm is critical **depending on hardware capabilities**
- SUNDIALS provided **hip**, **sycl**, and **cuda** (** below) class implementations

ThreadDirectExecPolicy(blockDim, stream)	One thread per work unit
GridStrideExecPolicy(blockDim, gridDim, stream)	Fixed grid and block size
BlockReduceAtomicExecPolicy(blockDim, gridDim, stream)	Block reduce with atomics
BlockReduceExecPolicy(blockDim, gridDim, stream)	Block reduce with shared memory

```
// Set the execution policies for steaming and reduction operations
int N_VSetKernelExecPolicy_**(N_Vector v, sundials::**::ExecPolicy stream_exec,
                    sundials::**::ExecPolicy reduce_exec);
```

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 $G(y) = y$ 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

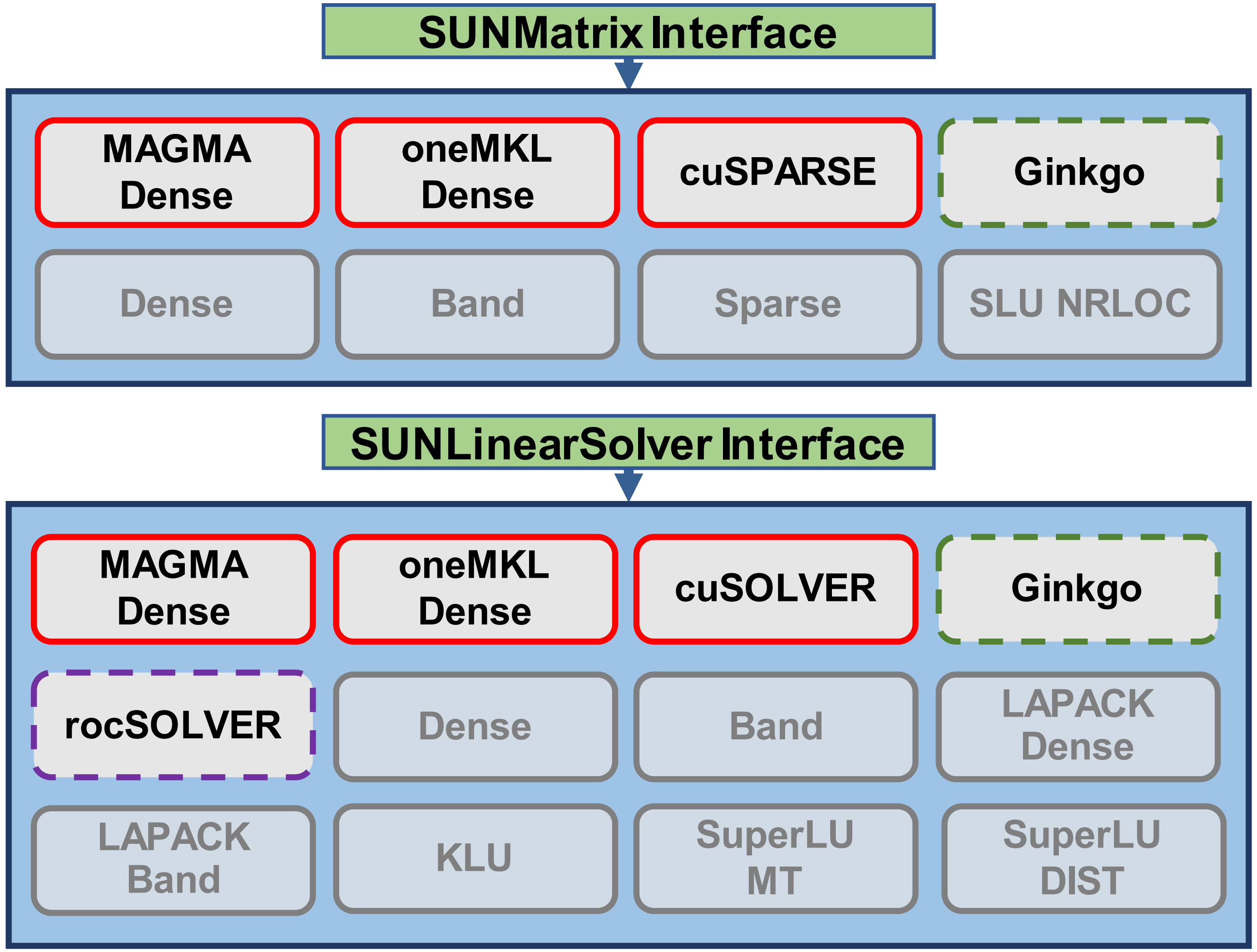
- By default, SUNDIALS integrators use a Newton method which requires linear solve each iteration
- In this case, users need to attach a linear solver object and, if necessary, a matrix object
- SUNDIALS provides several GPU-ready linear solver implementations/interfaces
  - **Iterative:** SUNDIALS' 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 batched direct linear solvers 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

SUNDIALS GPU Enabled Batched Direct Linear Solvers

- Interfaces to external linear solver libraries provide access to batched direct solvers for block diagonal systems that arise when solving independent systems together

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

- Dense blocks A_j,
  - **MAGMA** interface supports HIP and CUDA
  - **oneMKL** interface supports DPC++
- Sparse blocks A_j,
  - **cuSPRASE** interface supports CUDA
- **Ginkgo** will support sparse and dense batched iterative solvers - HIP/CUDA/SYCL



Creating GPU Enabled Dense Batched Matrices and Linear Solvers

```
// Create a block diagonal matrix of Nb blocks of size M x N
SUNMatrix SUNMatrix_MagmaDenseBlock(sunindextype Nb, sunindextype M,
                        sunindextype N, SUNMemoryType memtype,
                        SUNMemoryHelper helper, void* queue,
                        SUNContext sunctx);

// Create a MAGMA batched dense linear solver
SUNLinearSolver SUNLinSol_MagmaDense(N_Vector y, SUNMatrix A, SUNContext sunctx);
```

- All blocks A_j in the block-diagonal system must be the same size
- For sparse blocks (cuSPARSE, not shown), all blocks A_j must share the same sparsity pattern
- The user must provide a callback function for filling the Jacobian matrix, ideally this should launch a GPU kernel to compute and set the matrix entries
- The interface to the oneMLK is nearly identical, replace “Magma” with “OneMkl” and “void* queue” with “sycl::queue”

A High-Level Look at a GPU-enabled SUNDIALS example

- Consider the case where independent ODEs are combined into a larger group that is evolved together as a single system

- In this example, we use the Robertson example for a stiff autocatalytic reaction

$$\frac{dy_1}{dt} = -0.04\, y_1 + 10^4\, y_2\, y_3 \qquad \frac{dy_2}{dt} = 0.04\, y_1 - 10^4\, y_2\, y_3 - 3 \times 10^7\, y_2^2 \qquad \frac{dy_2}{dt} = 3 \times 10^7\, y_2^2$$

- 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

- **MAGMA HIP/CUDA** – see examples/cvode/magma/cvRoberts_blockdiag_magma.cpp

- **oneMKL DPC++** – see examples/cvode/CXX_onemkl/cvRoberts_blockdiag_onemkl.cpp

- **cuSPARSE CUDA** – see examples/cvode/cuda/cvRoberts_block_cusolversp_batchqr.cu

User-Supplied Functions: ODE RHS Evaluation

```
// ODE RHS function y' = f(t,y) launches a GPU kernel to do the computation
static int f(sunrealtype t, N_Vector y, N_Vector ydot, void* user_data)
{
  UserData*    udata      = (UserData*) user_data;
  sunrealtype* ydata      = N_VGetDeviceArrayPointer(y);
  sunrealtype* ydotdata   = N_VGetDeviceArrayPointer(ydot);
  unsigned     block_size = gpuBlockSize;
  unsigned     grid_size  = (udata->ngroups + block_size - 1) / block_size;

  f_kernel<<<grid_size, block_size>>>(t, ydata, ydotdata, udata->ngroups);

  return 0;
}

// Right hand side function evaluation kernel
__global__ void f_kernel(sunrealtype t, sunrealtype* ydata, sunrealtype* ydotdata, int ngroups)
{
  for (int j = blockIdx.x * blockDim.x + threadIdx.x; j < ngroups; j += blockDim.x * gridDim.x)
  {
    ydotdata[j]   = -0.04 * ydata[j] + 1.0e4 * ydata[j+1] * ydata[j+2];
    ydotdata[j+1] =  0.04 * ydata[j] - 1.0e4 * ydata[j+1] * ydata[j+2] - 3.0e7 * ydata[j+1] * ydata[j+1];
    ydotdata[j+2] = 3.0e7 * ydata[j+1] * ydata[j+1];
  }
}
```

User-Supplied Functions: ODE Jacobian Evaluation

```
// ODE Jacobian function J = df/dy(t,y) launches a GPU kernel to do the computation
static int J(sunrealtype 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;
  sunrealtype* ydata      = N_VGetDeviceArrayPointer(y);
  sunrealtype* Jdata      = SUNMatrix_MagmaDense_Data(J);
  unsigned     block_size = gpuBlockSize;
  unsigned     grid_size  = (udata->ngroups + block_size - 1) / block_size;

  j_kernel<<<grid_size, block_size>>>(ydata, Jdata, udata->ngroups);

  return 0;
}

// Jacobian function evaluation kernel
__global__ void j_kernel(sunrealtype* ydata, sunrealtype* Jdata, int ngroups)
{
  for (int j = blockIdx.x * blockDim.x + threadIdx.x; j < ngroups; j += blockDim.x * gridDim.x)
  {
    Jdata[GROUPSIZE * GROUPSIZE * j]     = -0.04;
    Jdata[GROUPSIZE * GROUPSIZE * j + 1] =  0.04;
    Jdata[GROUPSIZE * GROUPSIZE * j + 2] =  0.0;
    // Fill other matrix entries column-wise...
  }
}
```

Creating SUNDIALS Vector, Matrix, and Solver Objects

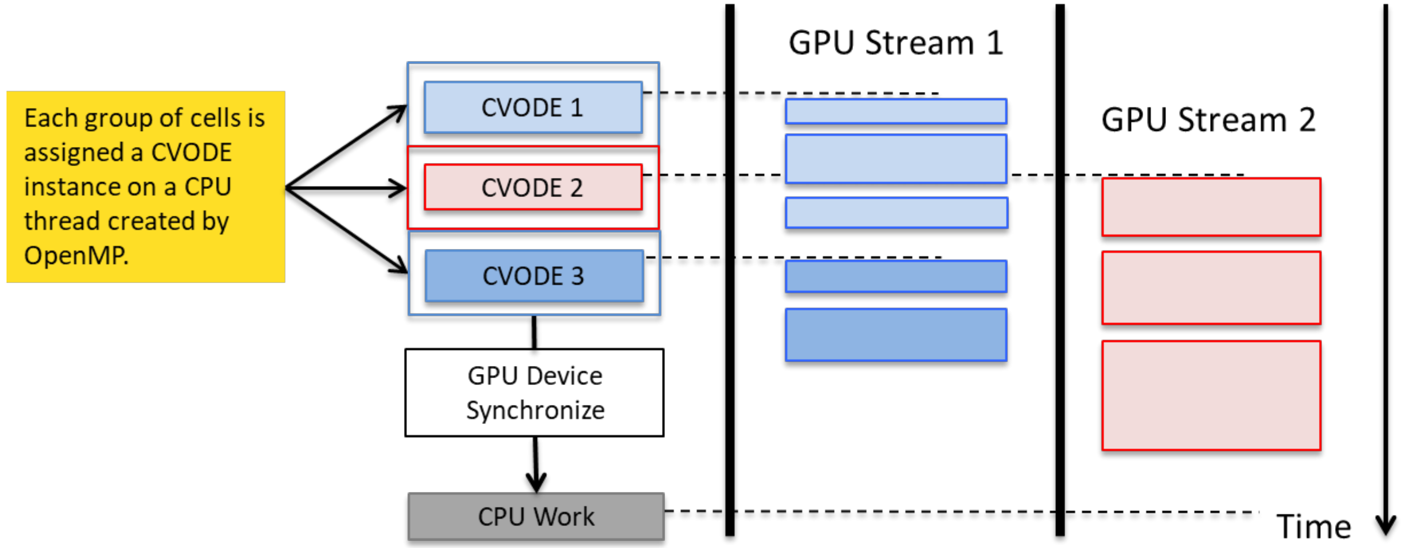
```
int main(int argc, char* argv[])
{
  sundials::Context sunctx;                                   // Create the SUNDIALS context

  // Read input parameters...

  sunindextype neq = GROUPSIZE * ngroups;                     // Number of ODE equations

  SUNMemoryHelper helper = SUNMemoryHelper_Hip(sunctx);       // SUNDIALS HIP Memory Allocator

  N_Vector y = N_Vnew_Hip(neq, sunctx);                       // Create the initial condition vector

  sunrealtype* ydata = N_VGetArrayPointer(y);                 // Fill host data and copy to device
  for (int j = 0; j < neq; j += GROUPSIZE)
  {
    ydata[j] = Y1; ydata[j+1] = Y2; ydata[j+2] = Y3;
  }
  N_VCopyToDevice_Hip(y);

  SUNMatrix A =                                               // Create MAGMA block dense SUNMatrix
    SUNMatrix_MagmaDenseBlock(ngroups, GROUPSIZE, GROUPSIZE,
                              SUNMEMTYPE_DEVICE, helper, NULL, sunctx);

  SUNLinearSolver LS = SUNLinSol_MagmaDense(y, A, sunctx);    // Create MAGMA SUNLinearSolver object

  // Setup CVODE...
```

Create, Initialize, and Configure CVODE then Evolve in Time

```
void* cvode_mem = CVodeCreate(CV_BDF, sunctx);        // Create and initialize CVODE, attaches the ODE RHS
retval = CVodeInit(cvode_mem, f, t0, y);             // function and sets the initial condition

UserData udata = {ngroups};                         // Create and attach the user data structure
retval = CVodeSetUserData(cvode_mem, &udata);

// Create and fill absolute tolerance vector...

retval = CVodeSVtolerances(cvode_mem, 1.0e-4, abstol);  // Specify the integration tolerances

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

retval = CVodeSetJacFn(cvode_mem, Jac);               // Set the Jacobian function

for (int iout = 0; iout < NOUT; iout++)
{
  retval = CVode(cvode_mem, tout, y, &tret, CV_NORMAL); // Evolve to output time

  N_VCopyFromDevice_Hip(y);                          // Copy solution to host for output

  // Output solution and update output time...
}

retval = CVodePrintAllStats(cvode_mem, stdout, SUN_OUTPUTFORMAT_TABLE);  // Print final statistics

// Destroy object, free memory, and return...
```

Using Multiple CVODE Instances with OpenMP and GPU Streams

- Consider same Robertson example where the larger group of independent systems is divided across multiple CVODE instances each associated with an **OpenMP thread and GPU stream**


Each group of cells is assigned a CVODE instance on a CPU thread created by OpenMP.
CVODE 1
CVODE 2
CVODE 3
GPU Device Synchronize
CPU Work
GPU Stream 1
GPU Stream 2
Time

- The use of OpenMP threads and GPU streams **enables concurrent kernel execution** which is beneficial when different groupings of systems require differing amounts of work

- We now **need to create arrays of objects** and potentially adjust the kernel launch parameters otherwise, the steps are largely the same as in the non-OpenMP case.

Creating SUNDIALS Vector, Matrix, and Solver Objects

```
int main(int argc, char* argv[])
{
 // Read input parameters and determined the problem size per thread...

 SUNContext sunctx[num_threads];
 // Arrays of other SUNDIALS objects...

 for (int i = 0; i < num_threads; i++)
 {
  hipStreamCreate(&stream[i]);                          // Create GPU streams
  retval    = SUNContext_Create(NULL, &sunctx[i]);              // Create the SUNDIALS contexts
  helper[i] = SUNMemoryHelper_Hip(sunctx[i]);                   // SUNDIALS HIP Memory Allocator
  y[i]      = N_Vnew_Hip(neq_per_thread, sunctx[i]);            // Create the vector and exec policy

  SUNHipExecPolicy* stream_exec = new SUNHipGridStrideExecPolicy(threads_per_block, blocks_per_grid,
                                  stream[i]);
  SUNHipExecPolicy* reduce_exec = new SUNHipBlockReduceExecPolicy(threads_per_block, blocks_per_grid,
                                  stream[i]);
  retval = N_VSetKernelExecPolicy_Hip(y, stream_exec, reduce_exec);
  delete stream_exec; delete reduce_exec;

  A[i] = SUNMatrix_MagmaDenseBlock(ngroups_per_thread, GROUPSIZE, GROUPSIZE, // Create MAGMA SUNMatrix
                 SUNMEMTYPE_DEVICE, helper[i], stream[i], sunctx[i]);

  LS[i] = SUNLinSol_MagmaDense(y[i], A[i], sunctx[i]);            // Create MAGMA SUNLinearSolver object
 }
```

Create, Initialize, and Configure CVODE then Evolve in Time

```
#pragma omp parallel for
 for (int i = 0; i < total_num_groups; i++)
 {
  int tid = omp_get_thread_num();                    // Get the thread ID

  retval = FillInitialCondition(y[tid]);             // Set the initial condition

  if (!cvode_initialized[tid])                       // Initialize and configure CVODE if not done yet
  {
   retval = CVodeInit(cvode_mem[tid], f, t0, y[tid]);
   cvode_initialized[tid] = 1;
   // Configure CVODE...
  }
  else
  {
   retval = CVodeReInit(cvode_mem[tid], t0, y[tid]);   // Reinitialize CVODE to evolve a new group
  }

  for (int iout = 0; iout < NOUT; iout++)
  {
   retval = CVode(cvode_mem[tid], tout, y[tid], &tret, CV_NORMAL);

   // Output solution and update output time...
  }
  // Output integrator statistics...
 }
```

Tutorial Outline

- Introduction (Carol Woodward)
- Multirate time integrators (Daniel Reynolds)
- Enhanced GPU support (David Gardner)
- **Performance profiling, analysis, and logging (Cody Balos)**
- Scalable demonstration code (Daniel Reynolds)
- Closing Remarks (Carol)

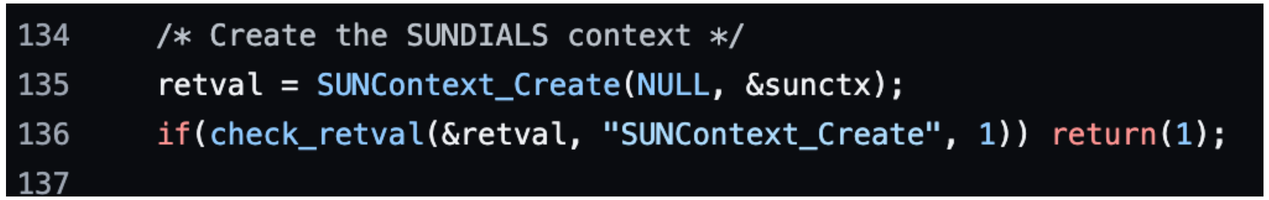
Built-In Profiling & Logging Makes Identifying Bottlenecks Easier

- SUNDIALS v6.0.0+ has a built-in, MPI-aware, performance profiler, `SUNProfiler`
  - Low-overhead when enabled and no overhead when disabled (choose at build-time)
  - Key regions within the time-integration loop are profiled out-of-the-box
  - Environment variable and run-time API
  - Can optionally use Caliper[1] for more advanced profiling without any additional code
- SUNDIALS v6.2.0+ adds new functions for printing stats and a logging capability, `SUNLogger`
  - `PrintAllStats` functions allow you to choose between human- and machine-readable formats
  - Choose max logging level at build-time to minimize overhead
  - Separate channels for errors, warnings, informational output, and debugging output
  - Lots of new informational output has been added
    - Internal integrator decisions and state etc.
  - Environment variable and run-time API
- Together, these make measuring and analyzing SUNDIALS performance easier than ever

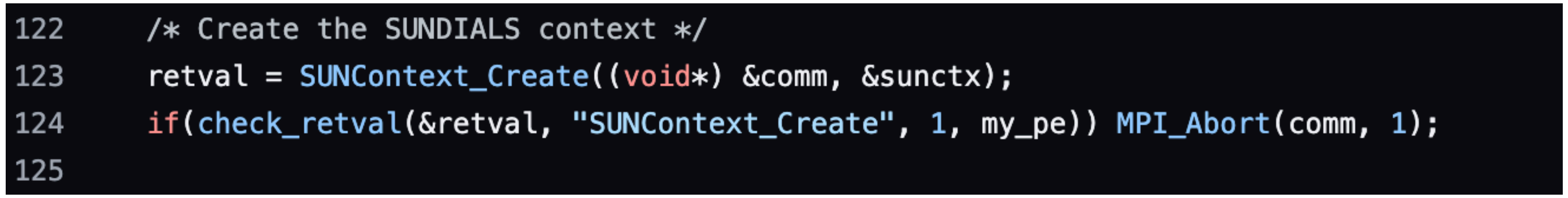
[1]http://software.llnl.gov/Caliper/

SUNContext

- To facilitate profiling, logging and error handling, v6.0.0 introduced the SUNContext object
- All the SUNDIALS objects (vectors, linear and nonlinear solvers, matrices, etc.) that collectively form a SUNDIALS simulation, hold a reference to a common simulation context object defined by the SUNContext class
- The SUNContext should be created before all other calls to the SUNDIALS library



```c
/* Create the SUNDIALS context */
retval = SUNContext_Create(NULL, &sunctx);
if(check_retval(&retval, "SUNContext_Create", 1)) return(1);

```



```c
/* Create the SUNDIALS context */
retval = SUNContext_Create((void*) &comm, &sunctx);
if(check_retval(&retval, "SUNContext_Create", 1, my_pe)) MPI_Abort(comm, 1);

```

Creating a SUNContext is simple. For serial programs (top), the first argument is NULL and the second is a pointer that will be the new context on output. For MPI programs (bottom) the first argument is a pointer to the communicator.

- See https://sundials.readthedocs.io/en/latest/sundials/SUNContext_link.html for more

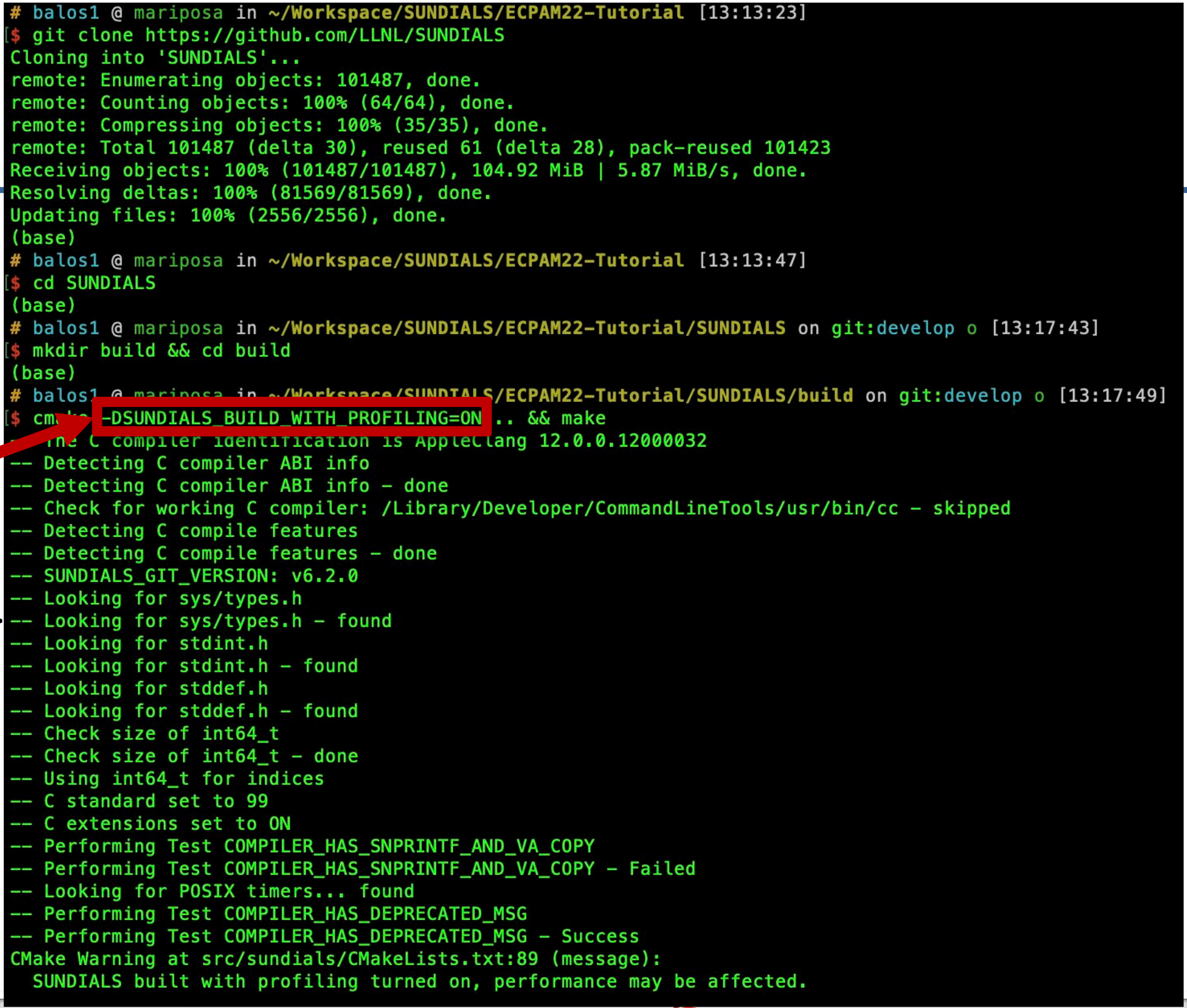
Profiling Demo

1. Clone SUNDIALS

2. Configure CMake with profiling ON

3. Set the environment variable SUNPROFILER_PRINT=<0|1|filename>

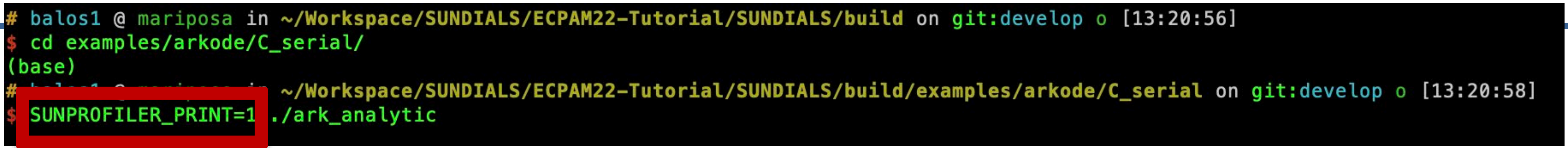
4. Run



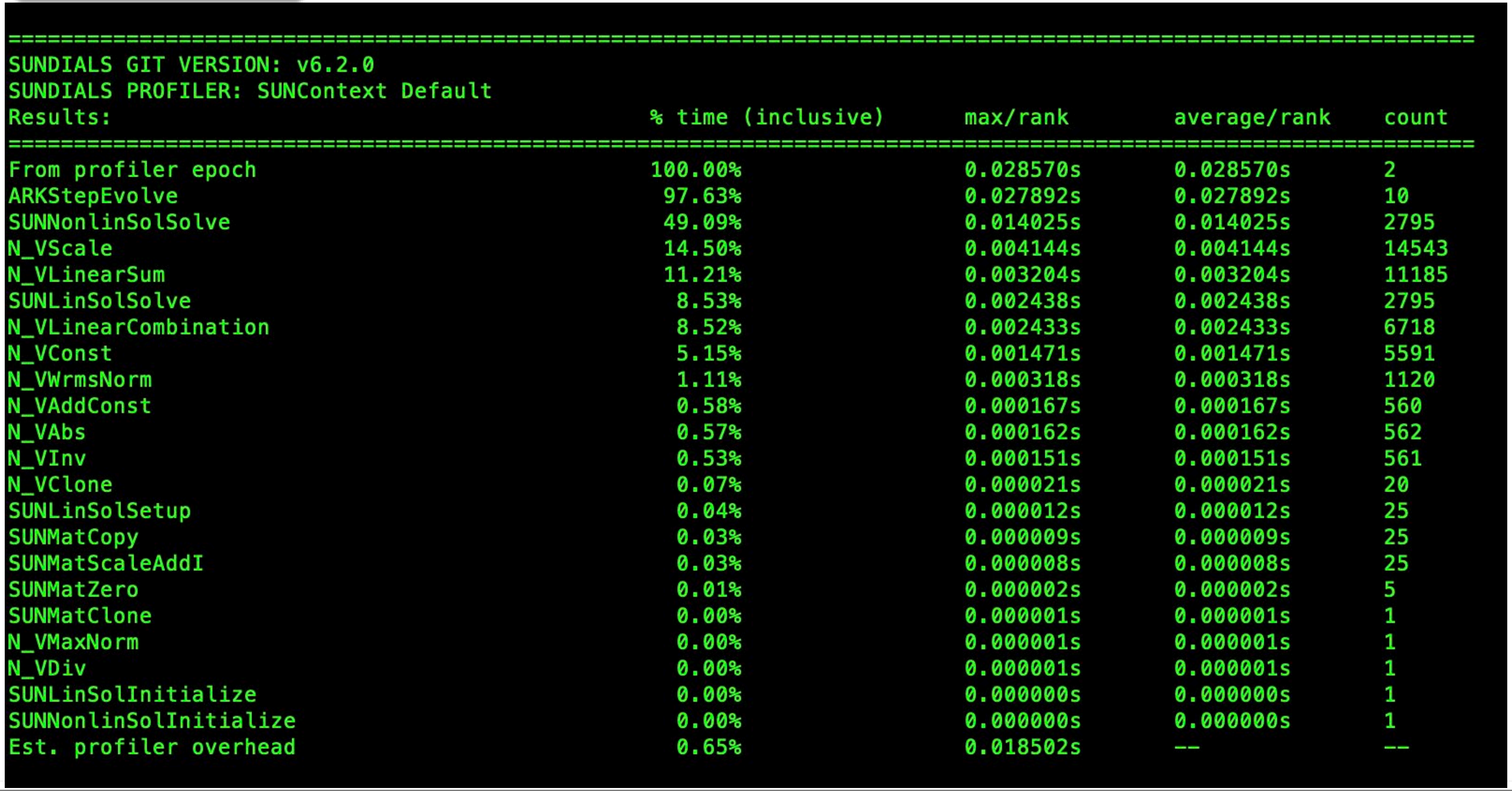
```
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial [13:13:23]
[$ git clone https://github.com/LLNL/SUNDIALS
Cloning into 'SUNDIALS'...
remote: Enumerating objects: 101487, done.
remote: Counting objects: 100% (64/64), done.
remote: Compressing objects: 100% (35/35), done.
remote: Total 101487 (delta 30), reused 61 (delta 28), pack-reused 101423
Receiving objects: 100% (101487/101487), 104.92 MiB | 5.87 MiB/s, done.
Resolving deltas: 100% (81569/81569), done.
Updating files: 100% (2556/2556), done.
(base)
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial [13:13:47]
[$ cd SUNDIALS
(base)
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS on git:develop o [13:17:43]
[$ mkdir build && cd build
(base)
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build on git:develop o [13:17:49]
[$ cmake -DSUNDIALS_BUILD_WITH_PROFILING=ON .. && make
-- The C compiler identification is AppleClang 12.0.0.12000032
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- SUNDIALS_GIT_VERSION: v6.2.0
-- Looking for sys/types.h
-- Looking for sys/types.h - found
-- Looking for stdint.h
-- Looking for stdint.h - found
-- Looking for stddef.h
-- Looking for stddef.h - found
-- Check size of int64_t
-- Check size of int64_t - done
-- Using int64_t for indices
-- C standard set to 99
-- C extensions set to ON
-- Performing Test COMPILER_HAS_SNPRINTF_AND_VA_COPY
-- Performing Test COMPILER_HAS_SNPRINTF_AND_VA_COPY - Failed
-- Looking for POSIX timers... found
-- Performing Test COMPILER_HAS_DEPRECATED_MSG
-- Performing Test COMPILER_HAS_DEPRECATED_MSG - Success
CMake Warning at src/sundials/CMakeLists.txt:89 (message):
  SUNDIALS built with profiling turned on, performance may be affected.
```

https://sundials.readthedocs.io/en/latest/sundials/Profiling_link.html

Profiling Demo



```
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build on git:develop o [13:20:56]
$ cd examples/arkode/C_serial/
(base)
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build/examples/arkode/C_serial on git:develop o [13:20:58]
$ SUNPROFILER_PRINT=1 ./ark_analytic
```



```
================================================================================================================
SUNDIALS GIT VERSION: v6.2.0
SUNDIALS PROFILER: SUNContext Default
Results:                                            % time (inclusive)      max/rank        average/rank    count
================================================================================================================
From profiler epoch                                  100.00%                0.028570s       0.028570s       2
ARKStepEvolve                                         97.63%                0.027892s       0.027892s       10
SUNNonlinSolSolve                                     49.09%                0.014025s       0.014025s       2795
N_VScale                                              14.50%                0.004144s       0.004144s       14543
N_VLinearSum                                          11.21%                0.003204s       0.003204s       11185
SUNLinSolSolve                                         8.53%                0.002438s       0.002438s       2795
N_VLinearCombination                                   8.52%                0.002433s       0.002433s       6718
N_VConst                                               5.15%                0.001471s       0.001471s       5591
N_VWrmsNorm                                            1.11%                0.000318s       0.000318s       1120
N_VAddConst                                            0.58%                0.000167s       0.000167s       560
N_VAbs                                                 0.57%                0.000162s       0.000162s       562
N_VInv                                                 0.53%                0.000151s       0.000151s       561
N_VClone                                               0.07%                0.000021s       0.000021s       20
SUNLinSolSetup                                         0.04%                0.000012s       0.000012s       25
SUNMatCopy                                             0.03%                0.000009s       0.000009s       25
SUNMatScaleAddI                                        0.03%                0.000008s       0.000008s       25
SUNMatZero                                             0.01%                0.000002s       0.000002s       5
SUNMatClone                                            0.00%                0.000001s       0.000001s       1
N_VMaxNorm                                             0.00%                0.000001s       0.000001s       1
N_VDiv                                                 0.00%                0.000001s       0.000001s       1
SUNLinSolInitialize                                    0.00%                0.000000s       0.000000s       1
SUNNonlinSolInitialize                                 0.00%                0.000000s       0.000000s       1
Est. profiler overhead                                 0.65%                0.018502s       --              --
```

Not setting SUNPROFILER_PRINT or setting it to 0 disables profiling output but not the profiling itself.

SUNPROFILER_PRINT can alternatively be set to a filename.

https://github.com/LLNL/sundials/blob/v6.2.0/examples/cvode/serial/cvAdvDiff_bnd.c

Profiling Runtime API

```
flag = SUNContext_GetProfiler(ctx, &prof);
if (check_flag(&flag, "SUNContext_GetProfiler", 1)) return 1;
```

```
UserData    udata(prof);
```

SUNProfiler runtime API allows users to a) configure profiling b) add profile regions to user-code.

1. Get the default SUNProfiler object from the SUNContext
2. Store it in user data
3. Access it in the RHS function
4. Mark RHS function for profiling

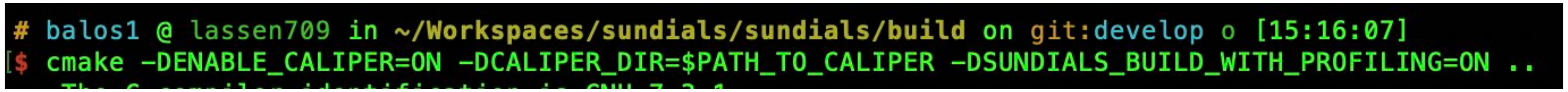
```
int diffusion(realtype t, N_Vector u, N_Vector f, void *user_data)
{
#ifdef SUNDIALS_BUILD_WITH_PROFILING
  // Access problem data
  UserData *udata = (UserData *) user_data;
#endif

  SUNDIALS_CXX_MARK_FUNCTION(udata->prof);

  // Compute the Laplacian
  int flag = laplacian(t, u, f, user_data);
  if (check_flag(&flag, "laplacian", 1))
    return -1;


  return 0;
}
```

https://github.com/LLNL/sundials/tree/develop/benchmarks/diffusion_2D

Profiling with Caliper

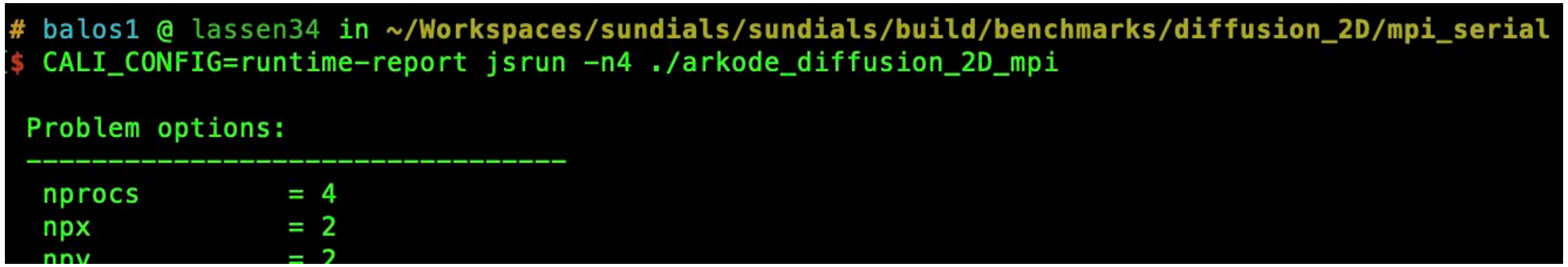
- Caliper is a program instrumentation and performance measurement framework.

- To use Caliper instead of the SUNDIALS native profiler:
  1. Install Caliper
  2. When building SUNDIALS provide CMake with:
     - ENABLE_CALIPER=ON
     - CALIPER_DIR=path/to/caliper
     - SUNDIALS_BUILD_WITH_PROFILING=ON



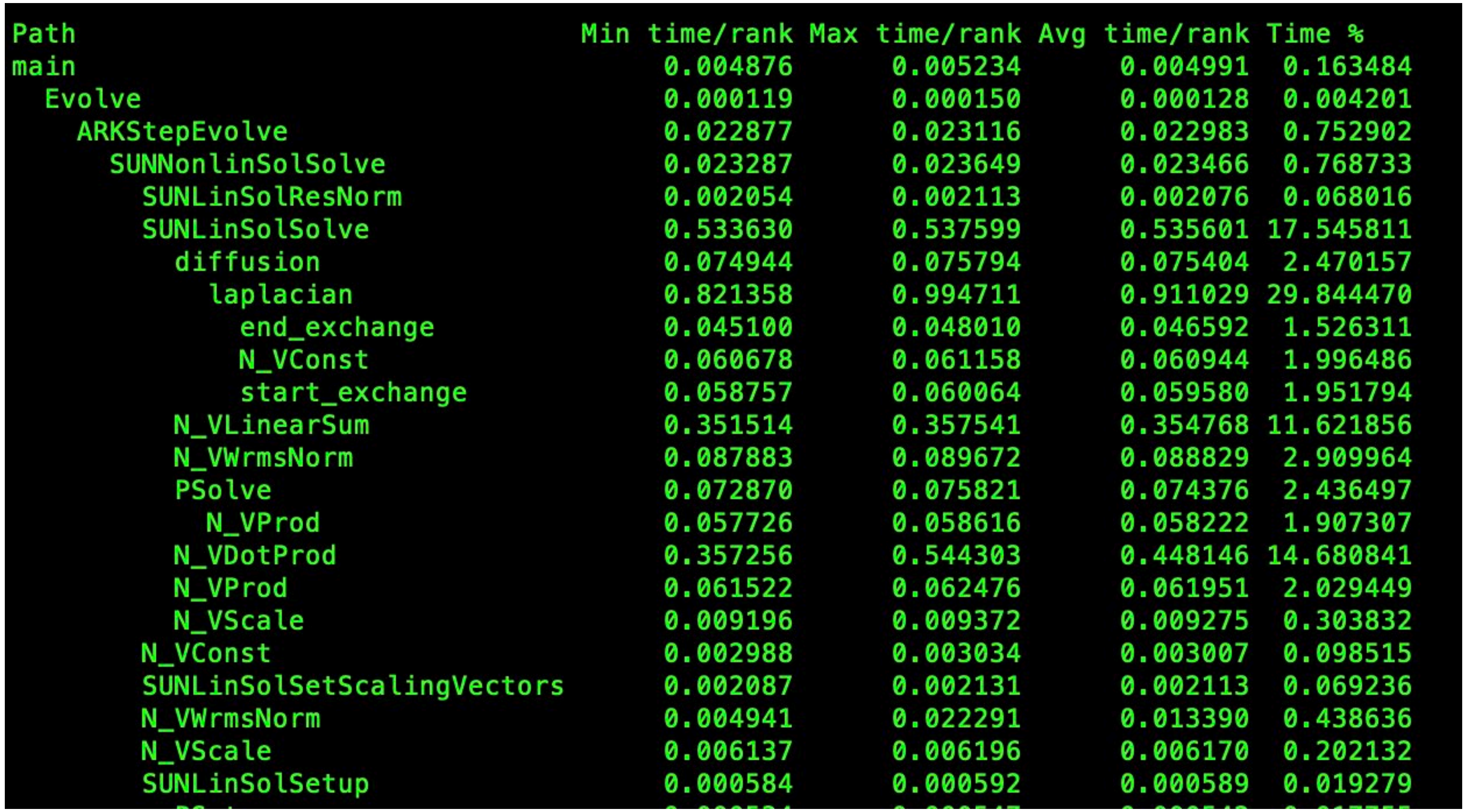
```
# balos1 @ lassen709 in ~/Workspaces/sundials/sundials/build on git:develop o [15:16:07]
[$ cmake -DENABLE_CALIPER=ON -DCALIPER_DIR=$PATH_TO_CALIPER -DSUNDIALS_BUILD_WITH_PROFILING=ON ..
```

  3. Use Caliper environment variables to configure it
  4. Run



```
# balos1 @ lassen34 in ~/Workspaces/sundials/sundials/build/benchmarks/diffusion_2D/mpi_serial
$ CALI_CONFIG=runtime-report jsrun -n4 ./arkode_diffusion_2D_mpi

 Problem options:
 --------------------------------
  nprocs          = 4
  npx             = 2
  npy             = 2
```

Profiling with Caliper Demo

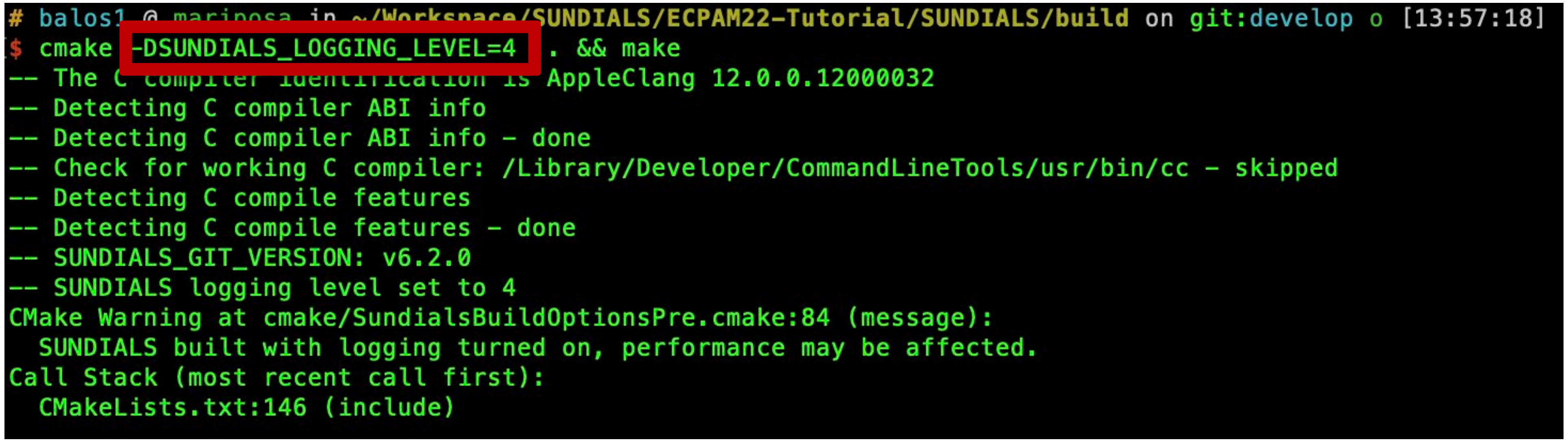


```
Path                               Min time/rank Max time/rank Avg time/rank Time %
main                                    0.004876      0.005234      0.004991  0.163484
  Evolve                                0.000119      0.000150      0.000128  0.004201
    ARKStepEvolve                       0.022877      0.023116      0.022983  0.752902
      SUNNonlinSolSolve                 0.023287      0.023649      0.023466  0.768733
        SUNLinSolResNorm                0.002054      0.002113      0.002076  0.068016
        SUNLinSolSolve                  0.533630      0.537599      0.535601 17.545811
          diffusion                     0.074944      0.075794      0.075404  2.470157
            laplacian                   0.821358      0.994711      0.911029 29.844470
              end_exchange              0.045100      0.048010      0.046592  1.526311
              N_VConst                  0.060678      0.061158      0.060944  1.996486
              start_exchange            0.058757      0.060064      0.059580  1.951794
          N_VLinearSum                  0.351514      0.357541      0.354768 11.621856
          N_VWrmsNorm                   0.087883      0.089672      0.088829  2.909964
          PSolve                        0.072870      0.075821      0.074376  2.436497
            N_VProd                     0.057726      0.058616      0.058222  1.907307
          N_VDotProd                    0.357256      0.544303      0.448146 14.680841
          N_VProd                       0.061522      0.062476      0.061951  2.029449
          N_VScale                      0.009196      0.009372      0.009275  0.303832
        N_VConst                        0.002988      0.003034      0.003007  0.098515
        SUNLinSolSetScalingVectors      0.002087      0.002131      0.002113  0.069236
        N_VWrmsNorm                     0.004941      0.022291      0.013390  0.438636
        N_VScale                        0.006137      0.006196      0.006170  0.202132
        SUNLinSolSetup                  0.000584      0.000592      0.000589  0.019279
```

Sample output from running the SUNDIALS 2D diffusion benchmark problem with Caliper profiling enabled (left).

https://github.com/LLNL/sundials/tree/develop/benchmarks/diffusion_2D

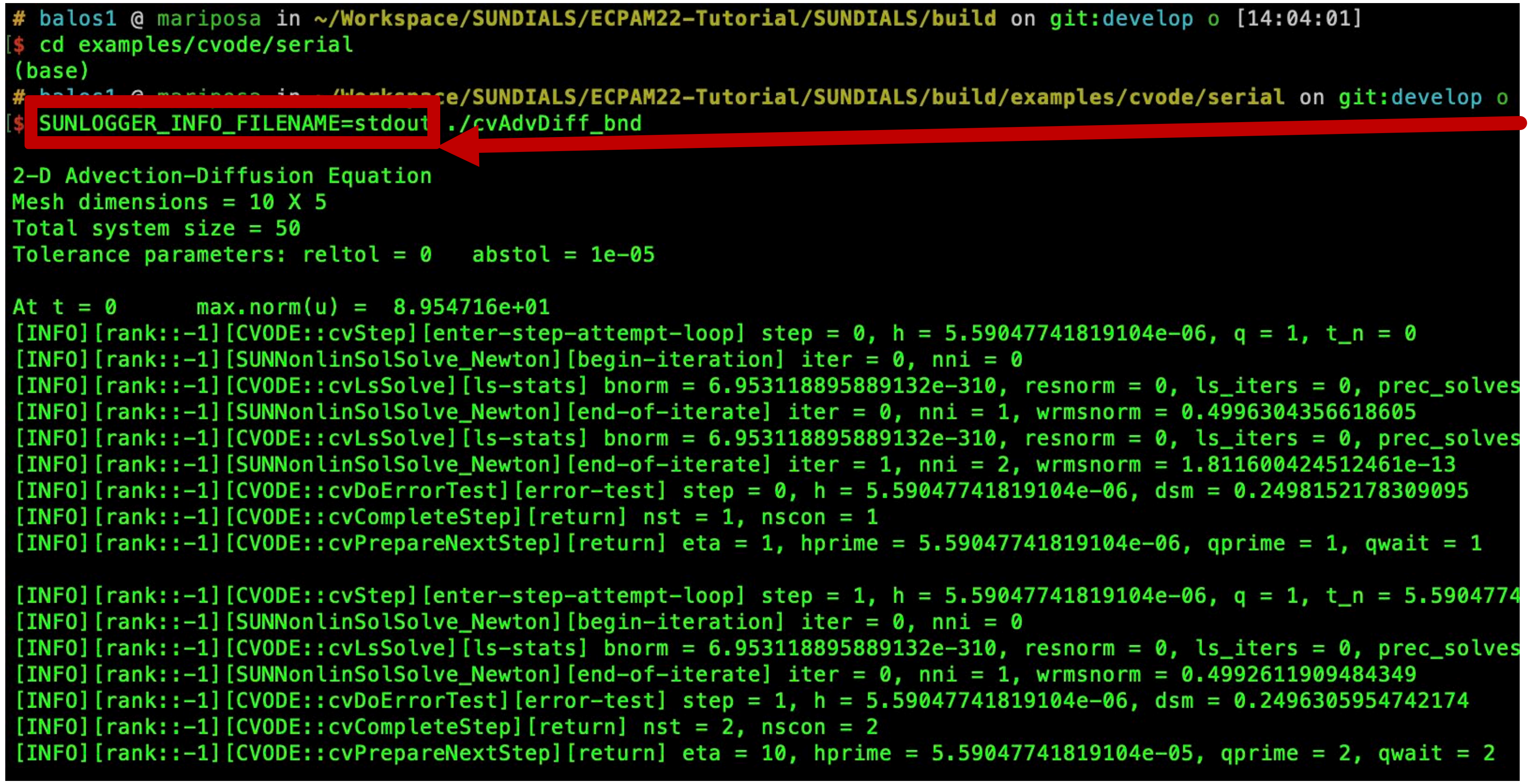
Logging Demo



```
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build on git:develop o [13:57:18]
$ cmake -DSUNDIALS_LOGGING_LEVEL=4 . && make
-- The C compiler identification is AppleClang 12.0.0.12000032
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- SUNDIALS_GIT_VERSION: v6.2.0
-- SUNDIALS logging level set to 4
CMake Warning at cmake/SundialsBuildOptionsPre.cmake:84 (message):
  SUNDIALS built with logging turned on, performance may be affected.
Call Stack (most recent call first):
  CMakeLists.txt:146 (include)
```

1. Configure CMake with SUNDIALS_LOGGING_LEVEL set to
   - 1 – errors only
   - 2 – errors + warnings
   - 3 – errors + warnings + info
   - 4 – errors + warnings + info + debugging
2. Set output location for levels through environment variables
   - SUNLOGGER_<ERROR|WARNING|INFO|DEBUG>_FILENAME
3. Run any example and see the output

Logging Demo



```
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build on git:develop o [14:04:01]
[$ cd examples/cvode/serial
(base)
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build/examples/cvode/serial on git:develop o
[$ SUNLOGGER_INFO_FILENAME=stdout ./cvAdvDiff_bnd

2-D Advection-Diffusion Equation
Mesh dimensions = 10 X 5
Total system size = 50
Tolerance parameters: reltol = 0   abstol = 1e-05

At t = 0      max.norm(u) =  8.954716e+01
[INFO][rank::-1][CVODE::cvStep][enter-step-attempt-loop] step = 0, h = 5.59047741819104e-06, q = 1, t_n = 0
[INFO][rank::-1][SUNNonlinSolSolve_Newton][begin-iteration] iter = 0, nni = 0
[INFO][rank::-1][CVODE::cvLsSolve][ls-stats] bnorm = 6.953118895889132e-310, resnorm = 0, ls_iters = 0, prec_solves
[INFO][rank::-1][SUNNonlinSolSolve_Newton][end-of-iterate] iter = 0, nni = 1, wrmsnorm = 0.4996304356618605
[INFO][rank::-1][CVODE::cvLsSolve][ls-stats] bnorm = 6.953118895889132e-310, resnorm = 0, ls_iters = 0, prec_solves
[INFO][rank::-1][SUNNonlinSolSolve_Newton][end-of-iterate] iter = 1, nni = 2, wrmsnorm = 1.811600424512461e-13
[INFO][rank::-1][CVODE::cvDoErrorTest][error-test] step = 0, h = 5.59047741819104e-06, dsm = 0.2498152178309095
[INFO][rank::-1][CVODE::cvCompleteStep][return] nst = 1, nscon = 1
[INFO][rank::-1][CVODE::cvPrepareNextStep][return] eta = 1, hprime = 5.59047741819104e-06, qprime = 1, qwait = 1

[INFO][rank::-1][CVODE::cvStep][enter-step-attempt-loop] step = 1, h = 5.59047741819104e-06, q = 1, t_n = 5.5904774
[INFO][rank::-1][SUNNonlinSolSolve_Newton][begin-iteration] iter = 0, nni = 0
[INFO][rank::-1][CVODE::cvLsSolve][ls-stats] bnorm = 6.953118895889132e-310, resnorm = 0, ls_iters = 0, prec_solves
[INFO][rank::-1][SUNNonlinSolSolve_Newton][end-of-iterate] iter = 0, nni = 1, wrmsnorm = 0.4992611909484349
[INFO][rank::-1][CVODE::cvDoErrorTest][error-test] step = 1, h = 5.59047741819104e-06, dsm = 0.2496305954742174
[INFO][rank::-1][CVODE::cvCompleteStep][return] nst = 2, nscon = 2
[INFO][rank::-1][CVODE::cvPrepareNextStep][return] eta = 10, hprime = 5.59047741819104e-05, qprime = 2, qwait = 2
```

Enable info-level output with the SUNLOGGER_INFO_FILELNAME environment variables

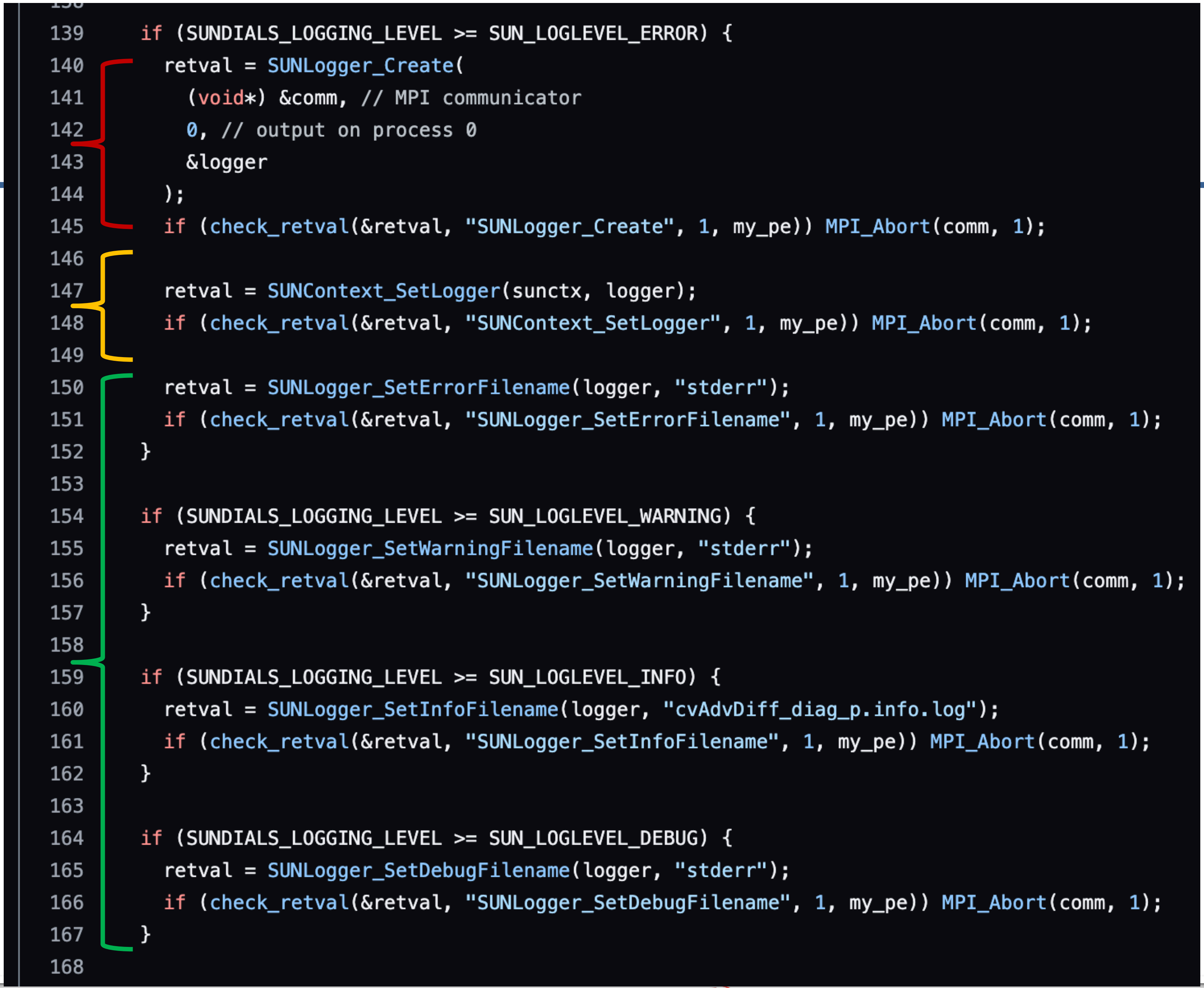
In this case we send the informational output to stdout

Output is structured to be machine-readable and easily filterable: [LEVEL][MPI_RANK][SCOPE][LABEL]

https://github.com/LLNL/sundials/blob/v6.2.0/examples/cvode/serial/cvAdvDiff_bnd.c

Logger Runtime API

1. Create SUNLogger object
2. Attach logger to simulation SUNContext
3. Set filenames for level output

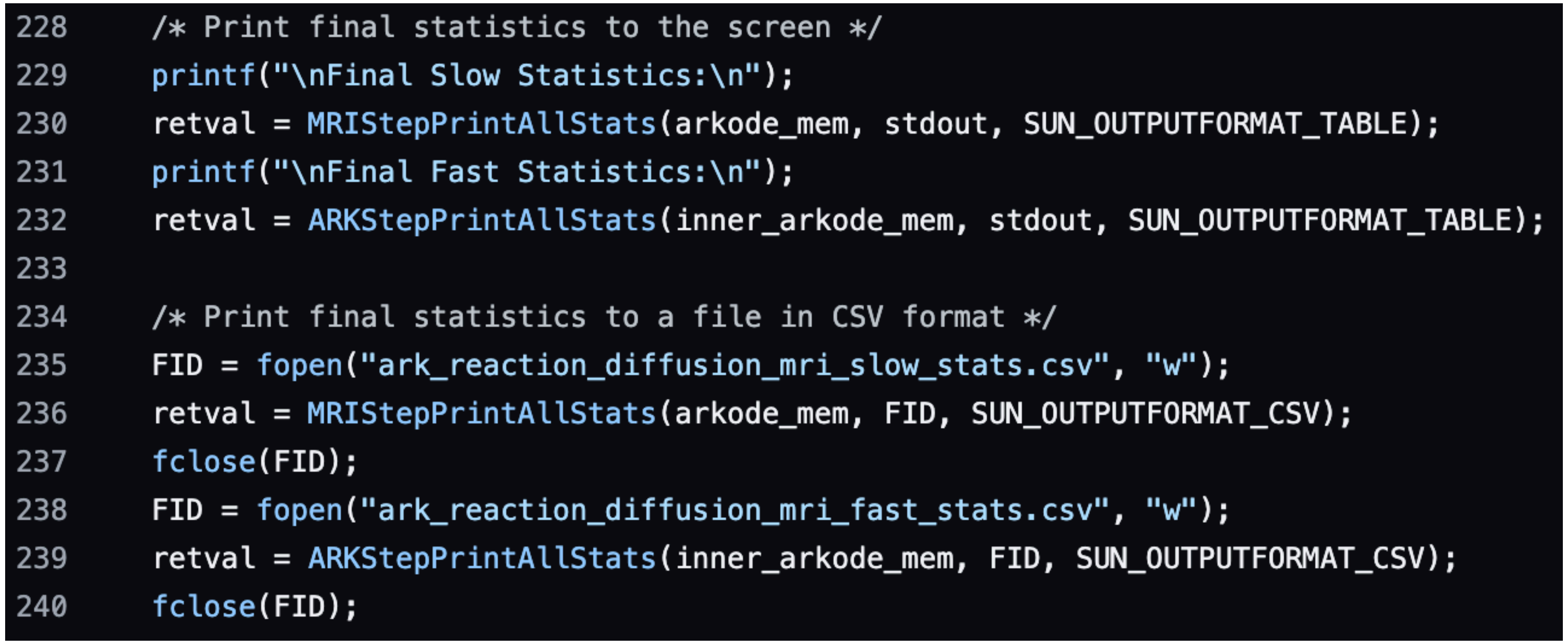


```
  if (SUNDIALS_LOGGING_LEVEL >= SUN_LOGLEVEL_ERROR) {
    retval = SUNLogger_Create(
      (void*) &comm, // MPI communicator
      0, // output on process 0
      &logger
    );
    if (check_retval(&retval, "SUNLogger_Create", 1, my_pe)) MPI_Abort(comm, 1);

    retval = SUNContext_SetLogger(sunctx, logger);
    if (check_retval(&retval, "SUNContext_SetLogger", 1, my_pe)) MPI_Abort(comm, 1);

    retval = SUNLogger_SetErrorFilename(logger, "stderr");
    if (check_retval(&retval, "SUNLogger_SetErrorFilename", 1, my_pe)) MPI_Abort(comm, 1);
  }

  if (SUNDIALS_LOGGING_LEVEL >= SUN_LOGLEVEL_WARNING) {
    retval = SUNLogger_SetWarningFilename(logger, "stderr");
    if (check_retval(&retval, "SUNLogger_SetWarningFilename", 1, my_pe)) MPI_Abort(comm, 1);
  }

  if (SUNDIALS_LOGGING_LEVEL >= SUN_LOGLEVEL_INFO) {
    retval = SUNLogger_SetInfoFilename(logger, "cvAdvDiff_diag_p.info.log");
    if (check_retval(&retval, "SUNLogger_SetInfoFilename", 1, my_pe)) MPI_Abort(comm, 1);
  }

  if (SUNDIALS_LOGGING_LEVEL >= SUN_LOGLEVEL_DEBUG) {
    retval = SUNLogger_SetDebugFilename(logger, "stderr");
    if (check_retval(&retval, "SUNLogger_SetDebugFilename", 1, my_pe)) MPI_Abort(comm, 1);
  }
```

https://github.com/LLNL/sundials/blob/develop/examples/cvode/parallel/cvAdvDiff_diag_p.c

*PrintAllStats functions print integrator and solver statistics in a human-readable format or in a machine-readable CSV format

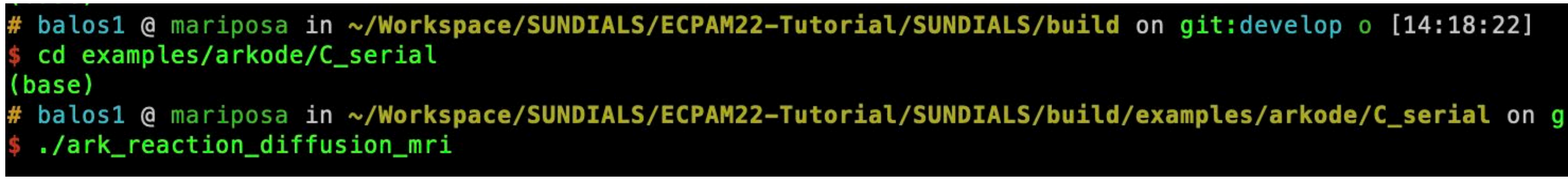
CVodePrintAllStats, ARKStepPrintAllStats, ERKStepPrintAllStats, MRIStepPrintAllStats, IDAPrintAllStats, KINPrintAllStats



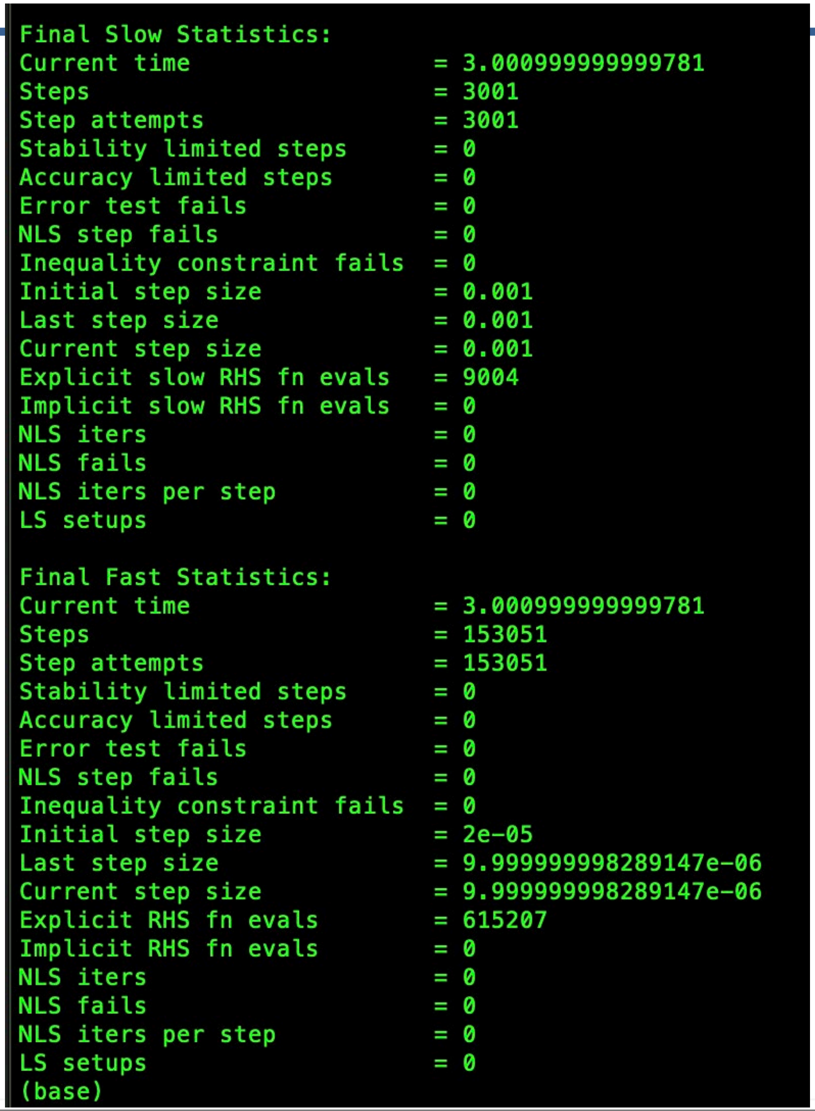
```
/* Print final statistics to the screen */
printf("\nFinal Slow Statistics:\n");
retval = MRIStepPrintAllStats(arkode_mem, stdout, SUN_OUTPUTFORMAT_TABLE);
printf("\nFinal Fast Statistics:\n");
retval = ARKStepPrintAllStats(inner_arkode_mem, stdout, SUN_OUTPUTFORMAT_TABLE);

/* Print final statistics to a file in CSV format */
FID = fopen("ark_reaction_diffusion_mri_slow_stats.csv", "w");
retval = MRIStepPrintAllStats(arkode_mem, FID, SUN_OUTPUTFORMAT_CSV);
fclose(FID);
FID = fopen("ark_reaction_diffusion_mri_fast_stats.csv", "w");
retval = ARKStepPrintAllStats(inner_arkode_mem, FID, SUN_OUTPUTFORMAT_CSV);
fclose(FID);
```

https://github.com/LLNL/sundials/blob/v6.2.0/examples/arkode/C_serial/ark_reaction_diffusion_mri.c

PrintAllStats Demo

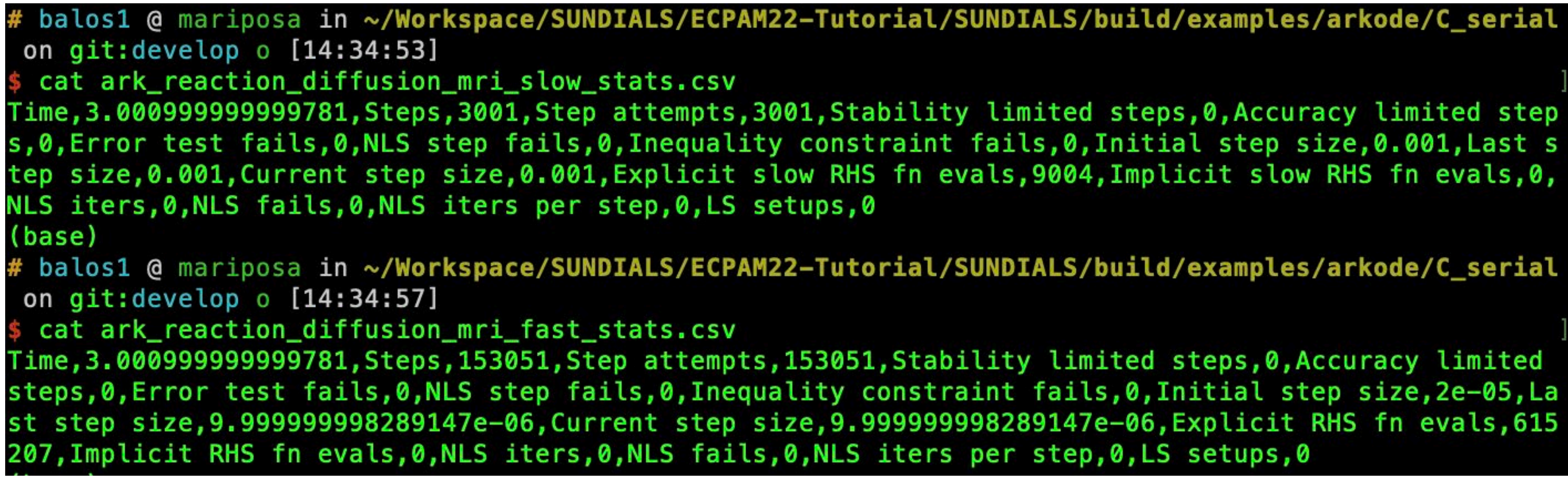


```
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build on git:develop o [14:18:22]
$ cd examples/arkode/C_serial
(base)
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build/examples/arkode/C_serial on g
$ ./ark_reaction_diffusion_mri
```



```
Final Slow Statistics:
Current time                = 3.000999999999781
Steps                       = 3001
Step attempts               = 3001
Stability limited steps     = 0
Accuracy limited steps      = 0
Error test fails            = 0
NLS step fails              = 0
Inequality constraint fails = 0
Initial step size           = 0.001
Last step size              = 0.001
Current step size           = 0.001
Explicit slow RHS fn evals  = 9004
Implicit slow RHS fn evals  = 0
NLS iters                   = 0
NLS fails                   = 0
NLS iters per step          = 0
LS setups                   = 0

Final Fast Statistics:
Current time                = 3.000999999999781
Steps                       = 153051
Step attempts               = 153051
Stability limited steps     = 0
Accuracy limited steps      = 0
Error test fails            = 0
NLS step fails              = 0
Inequality constraint fails = 0
Initial step size           = 2e-05
Last step size              = 9.999999998289147e-06
Current step size           = 9.999999998289147e-06
Explicit RHS fn evals       = 615207
Implicit RHS fn evals       = 0
NLS iters                   = 0
NLS fails                   = 0
NLS iters per step          = 0
LS setups                   = 0
(base)
```

Running the ark_reaction_diffusion_mri.c example (top) produces both human-readable output (left) and machine-readable CSV format (bottom) with PrintAllStats.



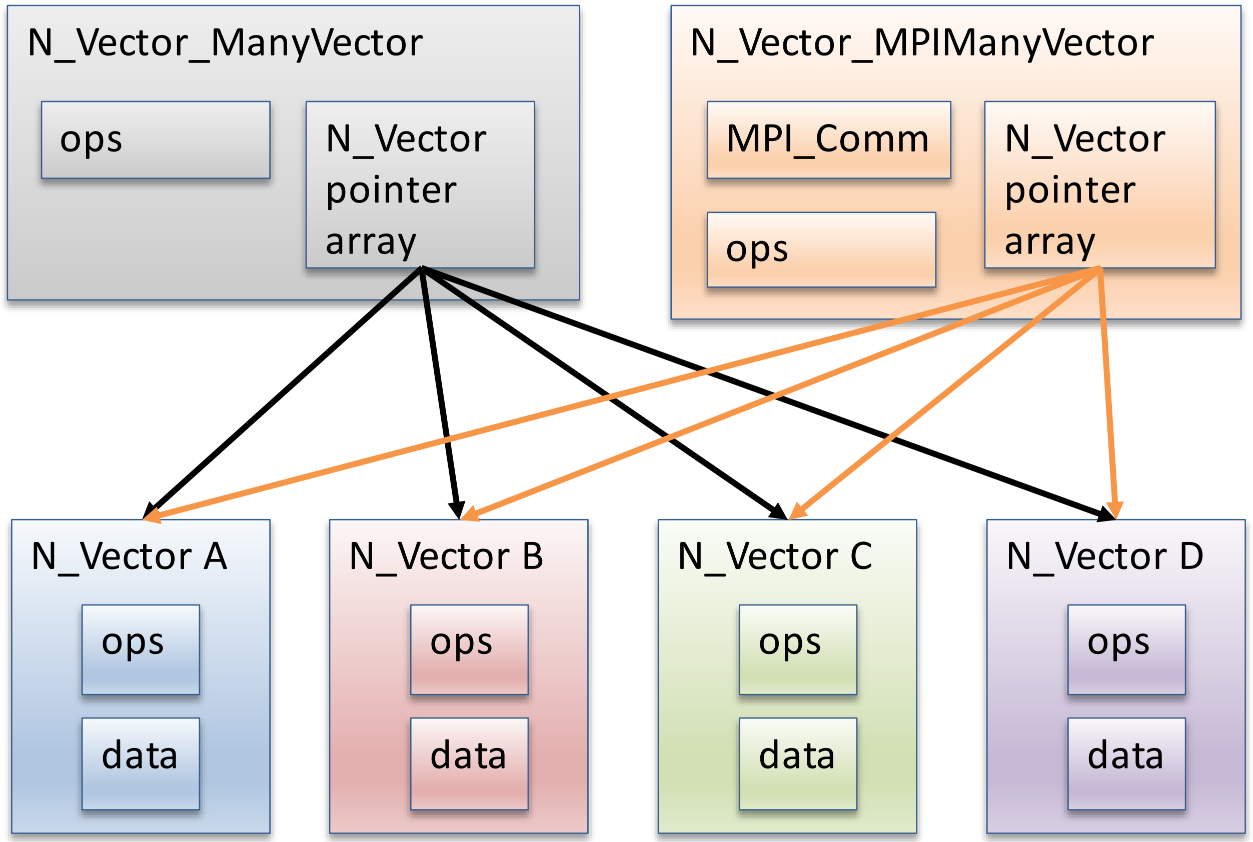
```
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build/examples/arkode/C_serial
 on git:develop o [14:34:53]
$ cat ark_reaction_diffusion_mri_slow_stats.csv
Time,3.000999999999781,Steps,3001,Step attempts,3001,Stability limited steps,0,Accuracy limited step
s,0,Error test fails,0,NLS step fails,0,Inequality constraint fails,0,Initial step size,0.001,Last s
tep size,0.001,Current step size,0.001,Explicit slow RHS fn evals,9004,Implicit slow RHS fn evals,0,
NLS iters,0,NLS fails,0,NLS iters per step,0,LS setups,0
(base)
# balos1 @ mariposa in ~/Workspace/SUNDIALS/ECPAM22-Tutorial/SUNDIALS/build/examples/arkode/C_serial
 on git:develop o [14:34:57]
$ cat ark_reaction_diffusion_mri_fast_stats.csv
Time,3.000999999999781,Steps,153051,Step attempts,153051,Stability limited steps,0,Accuracy limited
steps,0,Error test fails,0,NLS step fails,0,Inequality constraint fails,0,Initial step size,2e-05,La
st step size,9.999999998289147e-06,Current step size,9.999999998289147e-06,Explicit RHS fn evals,615
207,Implicit RHS fn evals,0,NLS iters,0,NLS fails,0,NLS iters per step,0,LS setups,0
```

Tutorial Outline

- Introduction (Carol Woodward)
- Multirate time integrators (Daniel Reynolds)
- Enhanced GPU support (David Gardner)
- Performance profiling, analysis, and logging (Cody Balos)
- **Scalable demonstration code (Daniel Reynolds)**
- Closing Remarks (Carol)

ManyVector – a Conceptual Interface for Data Flexibility

- SUNDIALS' ManyVector and MPIManyVector objects are thin software layers that treat a collection of vector objects as a single cohesive vector.
- Do not touch any data directly; their ops coordinate an operation by calling subvector ops.
- Each subvector may stage data as it wishes (e.g., CPU or GPU).
- Collective operations (norms, dot-products) utilize MPI at the higher MPIManyVector level, to minimize overhead.



SUNDIALS' Scalable Demonstration Code – Reacting Flow

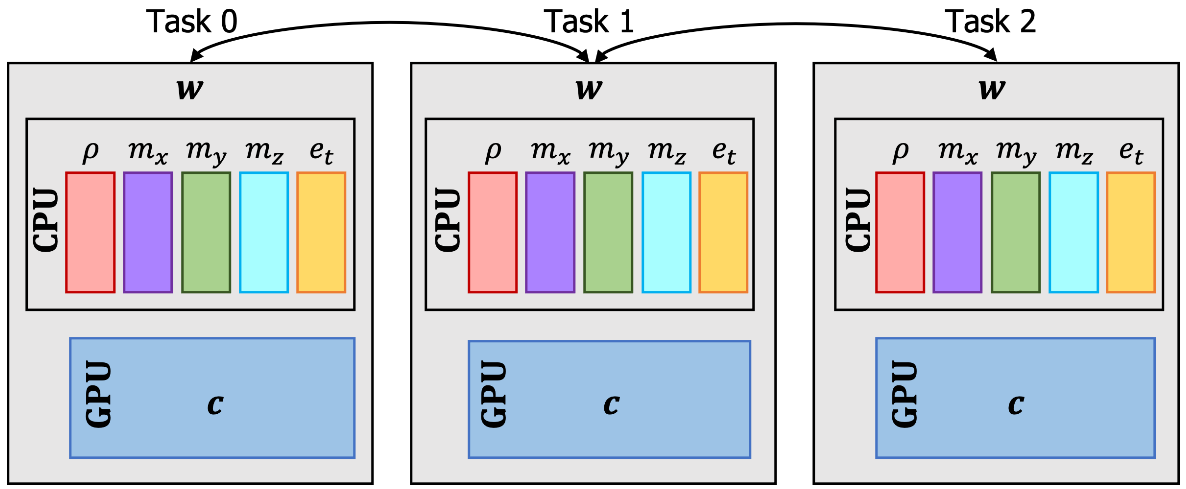
3D nonlinear compressible Euler equations combined with stiff chemical reactions for a low-density primordial gas (molecular & ionization states of H and He, free electrons, and internal gas energy), present in models of the early universe.

$$\partial_t \mathbf{w} = -\nabla \cdot \mathbf{F}(\mathbf{w}) + \mathbf{R}(\mathbf{w}), \quad t \in (t_0, t_f], \quad \mathbf{w}(t_0) = \mathbf{w}_0,$$

- $\mathbf{w}$: density, momenta, total energy, and chemical species (10)
- $\mathbf{F}$: advective fluxes (nonstiff/slow); and $\mathbf{R}$: reaction network (stiff/fast)

$\mathbf{w}$ is stored as an MPIManyVector:

- Fluid species (density, momenta, total energy) each stored in main memory
- Chemical densities stored in GPU memory, using NVECTOR_RAJA interface.
- ManyVector handles MPI collectives; manual point-to-point communication for fluxes.



Reacting Flow Solver Strategy

- Method of lines: $(X,t) \in \Omega \times (t_0, t_f]$, with $\Omega = [x_l, x_r] \times [y_l, y_r] \times [z_l, z_r]$.
- Regular n_x x n_y x n_z grid for Ω, parallelized using standard 3D MPI domain decomposition.
- $\mathcal{O}(\Delta x^5)$ FD-WENO flux reconstruction for $\mathbf{F}(\mathbf{w})$ [Shu, 2003].
- Resulting IVP system: $\dot{\mathbf{w}}(t) = f_1(\mathbf{w}) + f_2(\mathbf{w}), \;\; \mathbf{w}(t_0) = \mathbf{w}_0$, where $f_1(\mathbf{w})$ contains $-\nabla \cdot \mathbf{F}(\mathbf{w})$ and is evaluated on the CPU, while $f_2(\mathbf{w})$ contains spatially-local reaction network $\mathbf{R}(\mathbf{w})$ and is evaluated on the GPU.
- We compare two forms of temporal evolution:

  a) Temporally-adaptive, 3rd order ARK-ImEx method from ARKStep: f_1 explicit and f_2 implicit.

  b) Fixed-step, 3rd order explicit MRI-GARK method from MRIStep (temporally adaptive fast step h): f_1 slow/explicit and f_2 fast/DIRK.

IMEX Approach

- At each stage z_i within the ARK-ImEx method, we must solve a nonlinearly implicit system

$$\underbrace{z_i - hA^I_{i,i} f_2(z_i)}_{\text{implicit}} \underbrace{- y_n - h\sum_{j=1}^{i-1}\left(A^E_{i,j} f_1(z_j) + A^I_{i,j} f_2(z_j)\right)}_{\text{explicit}} = 0,$$

- Since f_2 contains only spatially-local reaction terms, Newton's method applied to this results in block-diagonal linear systems

$$J = \begin{bmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_{n_p} \end{bmatrix}, \qquad J_p = \begin{bmatrix} J_{p,1,1,1} & & & \\ & J_{p,2,1,1} & & \\ & & \ddots & \\ & & & J_{p,n_{xloc},n_{yloc},n_{zloc}} \end{bmatrix}, \qquad J_{p,i,j,k} \in \mathbf{R}^{10\times 10}$$

- We construct a custom SUNLinearSolver that solves each $J_p x_p = b_p$ using SUNDIALS' new GPU-enabled SUNLinSol_MagmaDense batched solver interface. The only communication required is a single MPI_Allreduce to gauge success/failure of the overall linear solve with J, along with norms associated with Newton's method.

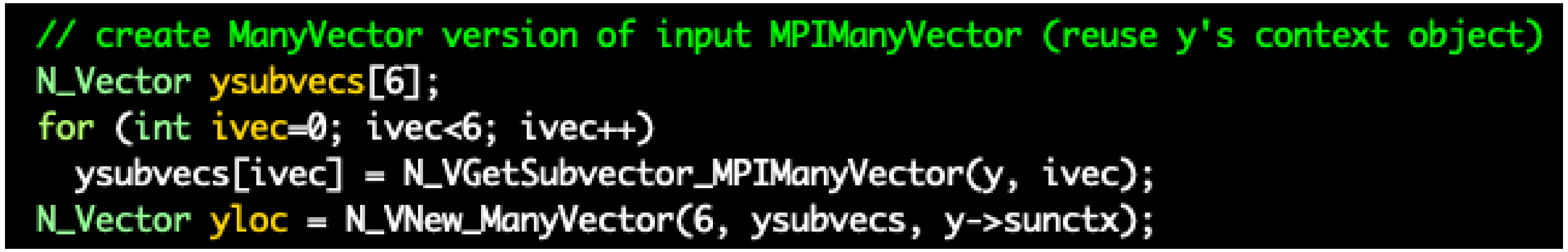
Multirate Approach

- The 3rd order explicit MRI method evaluates f_1 three times *per slow step*, and requires three modified fast IVPs:

$$v'(\tau) = f_2(v) + r_i(\tau), \quad \tau \in (c_{i-1}H, c_iH], \quad v(c_{i-1}H) = z_i$$

corresponding with a system of $n_x n_y n_z$ *decoupled* 15-variable IVPs.

- We construct a custom MRIStepInnerStepper that evolves these separately on each MPI rank.
  - The MRIStep-provided z_i and $r_i(\tau)$ use MPIManyVectors
  - Custom stepper repackages as rank-local ManyVectors, calling ARKStep to evolve each

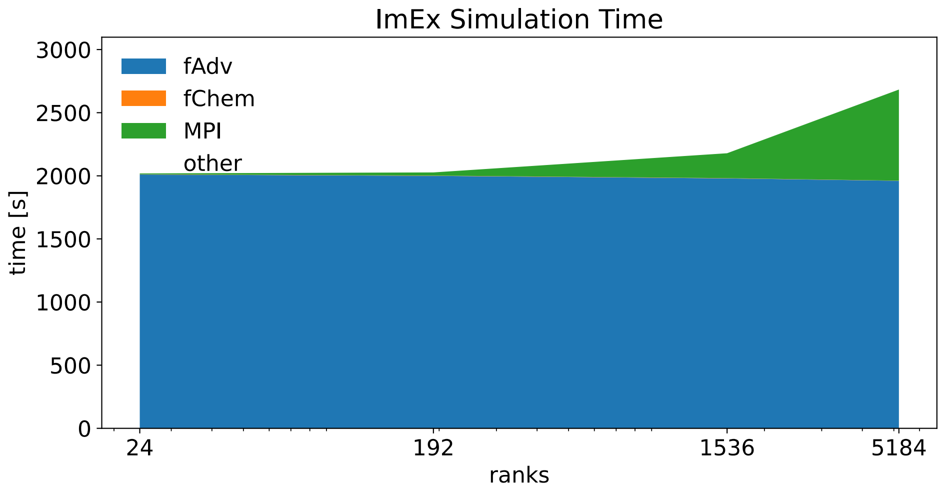


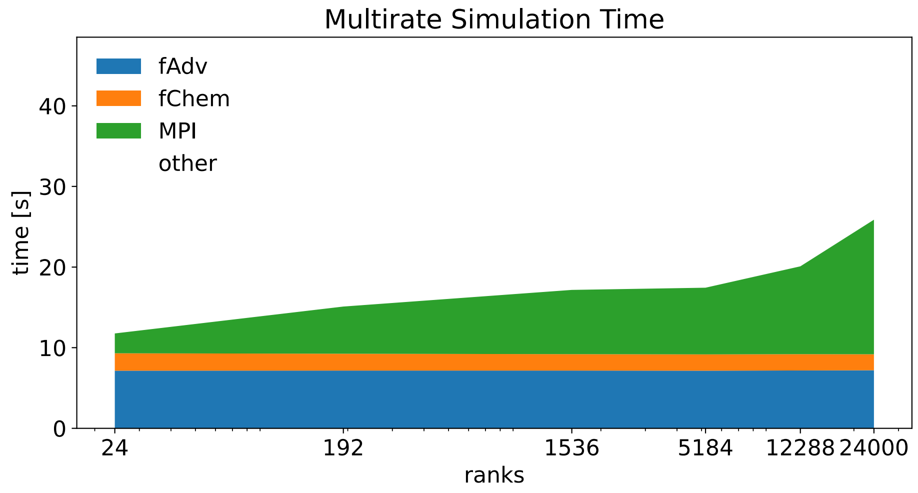
```
// create ManyVector version of input MPIManyVector (reuse y's context object)
N_Vector ysubvecs[6];
for (int ivec=0; ivec<6; ivec++)
  ysubvecs[ivec] = N_VGetSubvector_MPIManyVector(y, ivec);
N_Vector yloc = N_VNew_ManyVector(6, ysubvecs, y->sunctx);
```

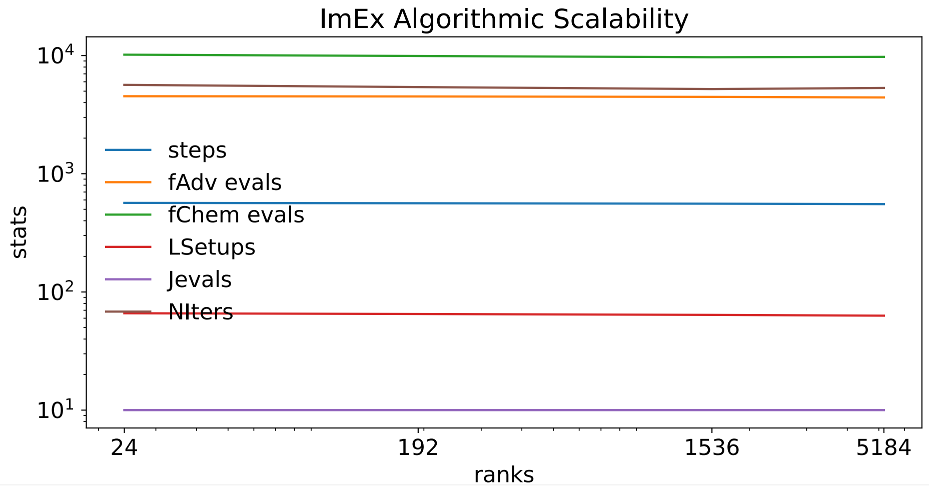
  - Implicit solves at the fast time scale involve rank-local Newton solvers, with nearly identical GPU-enabled SUNLinSol_MagmaDense batched solver interface.
  - MPI_Allreduce call to gauge success/failure of fast IVP solves [at slow time scale].

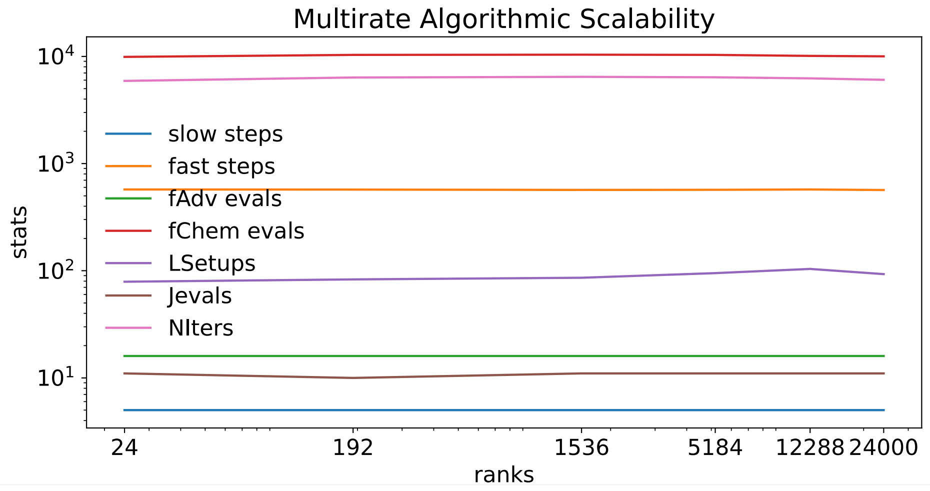
Weak Scaling Results (Summit)

- Weak scaling runs with 1 MPI rank per GPU.
- Multirate H chosen proportional to CFL condition on f_1.
- Both approaches show excellent alg. scalability.
- Huge reduction in f_1 evals allows MR / IMEX speedup of ~70x.
- GPU synchronization more severely hinders runtime scalability of IMEX than MR, due to increased frequency (fast vs slow stages).









Tutorial Outline

- Introduction (Carol Woodward)
- Multirate time integrators (Daniel Reynolds)
- Enhanced GPU support (David Gardner)
- Performance profiling, analysis, and logging (Cody Balos)
- Scalable demonstration code (Daniel Reynolds)
- **Closing Remarks (Carol)**

Where to learn more

- Visit the SUNDIALS website (Google LLNL SUNDIALS) https://computing.llnl.gov/projects/sundials
- Visit the SUNDIALS GitHub page: https://github.com/LLNL/sundials
- Where to get this tutorial:
  - SUNDIALS/hypre ECP Project Confluence Tutorials page: https://confluence.exascaleproject.org/display/STLM12/Tutorials
  - SUNDIALS Publications page (bottom): https://computing.llnl.gov/projects/sundials/publications
    - This page also includes prior tutorials on the basic uses of SUNDIALS
- **Come to our poster – Thur. 4:00-6:00 (EDT)**
- **Come to our breakout session and learn about user experiences with SUNDIALS. Wed. 10:00-11:00 (EDT)**
- Send any of us an email. We frequently do WebEx discussions with ECP users to go through interfaces and discuss use cases

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientic Discovery through Advanced Computing (SciDAC) Program through the FASTMath Institute

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.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

computing.llnl.gov/sundials

CASC

Center for Applied Scientific Computing

SMU®

Lawrence Livermore National Laboratory