OpenCS: a framework for parallelisation of equation-based simulation programs

Dragan D. Nikolić

DAE Tools Project, Belgrade, Serbia, http://daetools.sourceforge.io

Corresponding author:
Dragan D. Nikolić

Email address: dnikolic@daetools.com

ABSTRACT

In this work, the main ideas, the key concepts and the implementation details of the Open Compute Stack (OpenCS) framework are presented. The OpenCS framework is a common platform for modelling of problems described by large-scale systems of differential and algebraic equations, parallel evaluation of model equations on diverse types of computing devices (including heterogeneous setups), parallel simulation on shared and distributed memory systems, and model exchange.
The main components and the methodology of OpenCS are described: (1) model specification data structures for a description of general systems of differential and algebraic equations, (2) a method to describe, store in computer memory and evaluate model equations on general purpose and streaming processors, (3) algorithms for partitioning of general systems of equations in the presence of multiple load balancing constraints and for inter-process data exchange, (4) an Application Programming Interface (API), and (5) a cross-platform generic simulation software. The benefits provided by the framework are discussed in detail. The model specification data structures provide a simple platform-independent binary interface for model exchange and allow the same model representation to be used on different high-performance computing systems and architectures. Model equations are stored as an array of binary data (bytecode instructions) which can be directly evaluated on virtually all computing devices with no additional processing.
The capabilities of the framework are illustrated using two large-scale problems. Simulations are performed sequentially on a single processor and in parallel using the MPI interface. Multi-core CPU, discrete GPU and heterogeneous CPU/GPU setups are used for the evaluation of model equations utilising the OpenMP API and the OpenCL framework for parallelism. The overall performance and performance of four main and four sub-phases of the numerical solution are analysed and compared to the maximum theoretical.

INTRODUCTION

Equation-based mathematical modelling is one of efficient methods for simulation of engineering problems described by a system of ordinary differential (ODE) or differential-algebraic equations (DAE). On shared memory systems, the procedure for numerical solution of equation-based models includes the following computationally intensive tasks: (1) numerical integration of the overall ODE/DAE system in time by a suitable solver (requires evaluation of model equations), (2) linear algebra operations (mostly BLAS L1 vector operations and some of BLAS L2 matrix-vector operations), (3) solution of systems of linear equations (requires evaluation of derivatives), and (4) if requested, integration of sensitivity equations (requires evaluation of sensitivity residuals). On distributed memory systems, every processing element (PE) performs the same tasks on one part of the overall system (ODE/DAE sub-sybstem) and an inter-process data exchange required by the linear algebra and equation evaluation functions. In general, simulation programs for this class of problems are developed using:

1. General-purpose programming languages such as C/C++ or Fortran and one of available suites for scientific applications such as SUNDIALS (Hindmarsh et al., 2005), Trilinos (Heroux et al., 2005) and PETSc (Balay et al., 2015)
2. Modelling languages such as Ascend (Piela et al., 1991), gPROMS (Barton and Pantelides, 1994), APMonitor (Hedengren et al., 2014) and Modelica (Fritzson and Engelson, 1998)
3. Multi-paradigm numerical languages such as Matlab (The MathWorks, Inc., 2018a), Mathematica (Wolfram Research, Inc., 2015) and Maple (Waterloo Maple, Inc., 2015)

4. Higher-level fourth-generation languages (i.e. Python) and modelling software such as DAE Tools (Nikolić, 2016) and Assimulo (Andersson et al., 2015)
5. Libraries for finite element (FE) analysis and computational fluid dynamics (CFD) such as deal.II (Bangerth et al., 2007), libMesh (Kirk et al., 2006), and OpenFOAM (The OpenFOAM Foundation, 2018)
6. Computer Aided Engineering (CAE) software for finite element analysis and computational fluid dynamics such as HyperWorks (Altair, 2018), STAR-CCM+ and STAR-CD (Siemens, 2018), COMSOL Multiphysics (COMSOL, Inc., 2018), ANSYS Fluent/CFX (Ansys, Inc., 2018) and Abaqus (Dassault Systemes, 2018)

A detailed discussion of capabilities and limitations of the available approaches for model specification and development of large-scale simulation programs are given in Nikolić (2016, 2018, 2023). In all approaches, an interface to a particular ODE/DAE solver must be implemented to provide the information required for numerical integration in time (Fig. 1). The solver interface is directly implemented in general-purpose programming languages (i.e. as user-supplied functions). In other approaches, the solver interface is built around the internal simulator-specific data structures representing the model.

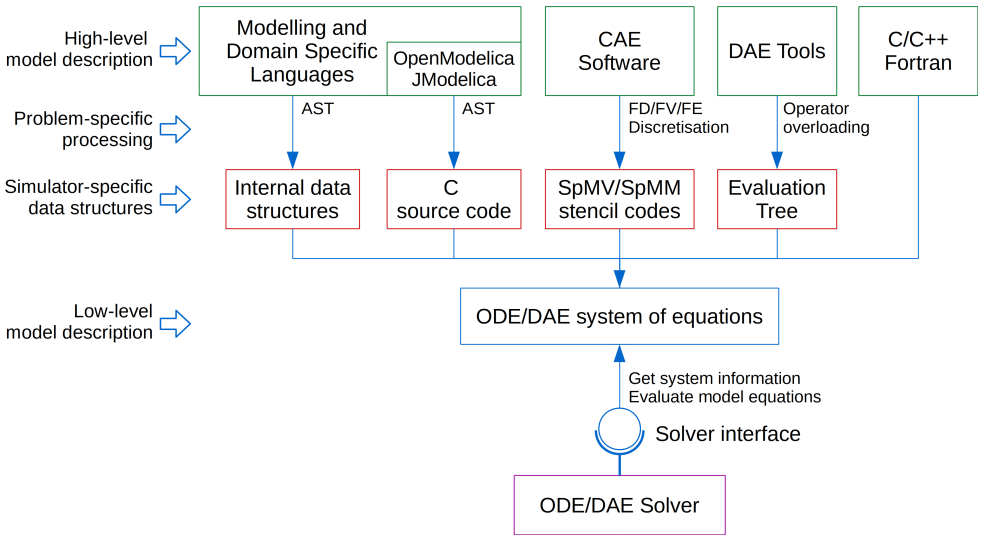

Figure 1. An overview of available modelling approaches. A high-level model description is created using the modelling or general purpose programming languages, FEA/CFD libraries and CAE software. A low-level model description is generated in a problem and simulator specific way and used to implement an interface to ODE/DAE solvers. The solver interface utilises the simulator-specific data structures to provide the information required for integration of the ODE/DAE system in time.

The idea in this work is to separate a high-level (simulator-dependent) model specification procedure, typically performed only once, from its parallel (in general, simulator-independent) numerical solution. While description of a model and generation of a system of equations can be performed in many different ways depending on the type of the problem and the method applied by a simulator, the numerical solution procedure always requires the same (low-level) information. For instance, a high-level model specification of the problems governed by partial differential equations can be created using a modelling language or a CAE software. The low-level model description is internally generated by simulators utilising various discretisation methods and results in a system of equations (ODE or DAE). However, the information required for numerical solution in both cases are essentially identical: the data about the variables (such as their number, names, types, absolute tolerances and initial conditions), and the functions for evaluation of equations and derivatives. Therefore, the low-level model description coupled with a method for parallel evaluation of model equations on different computing devices can be a basis for a universal software for parallel simulation of general systems of differential equations on all important platforms. In general, such a model description, due to its simplicity, can be generated and utilised by any existing simulator. This way, simulations can be performed on platforms not supported by that particular simulator or the simulation performance on the supported platforms can be improved by evaluating model equations in parallel on devices that are not currently utilised. In addition, the same platform-independent model description can be used for model exchange and benchmarks between different simulators, solvers, individual computing devices and high performance computing platforms (i.e. between heterogeneous systems, where evaluation of model equations is currently not available for

different architectures). An efficient evaluation of model equations is of utmost importance. For instance, very often more than 85% of the total integration time is spent on evaluation of equations and derivatives (Nikolić, 2018). Since most of the modern computers and many specially designed clusters are equipped with additional stream processors/accelerators such as Graphics Processing Units (GPU) and Field Programmable Gate Arrays (FPGA), the simulation software must be specially designed to effectively take advantage of multiple architectures. While parallel evaluation of model equations on general purpose processors is fairly straightforward and different techniques are applied by different simulators, evaluation on streaming processors is rather difficult. Stream computing differs from traditional computing in that the system processes a sequential stream of elements: a kernel is executed on each element of the input stream and the result stored in an output stream. Thus, the data structures representing the model equations must be designed to support evaluation on both systems (often simultaneously in heterogeneous computing setups).

To this end, the OpenCS framework is developed to provide:

1. Model specification data structures for a platform-independent description of general ODE/DAE systems of equations
2. A method to describe, store in computer memory and evaluate general systems of equations of any size on diverse types of computing devices
3. Algorithms for partitioning of general systems of equations in the presence of multiple load balancing constraints and for inter-process data exchange (for simulations on message passing multiprocessors)
4. An Application Programming Interface (API) for model specification, parallel evaluation of model equations, model exchange and a generic interface to ODE/DAE solvers
5. A cross-platform simulation software for parallel numerical solution of general ODE/DAE systems of equations on shared and distributed memory systems

This way, the OpenCS framework offers a common platform for specification of equation-based models, parallel evaluation of equations on diverse types of computing devices, model exchange and parallel simulation on shared and distributed memory systems (including heterogeneous systems).

OpenCS is free software released under the GNU Lesser General Public Licence. The installation packages, compilation instructions and more information about the OpenCS software can be found on the DAE Tools website (https://daetools.sourceforge.io/opencs.html). The source code is available from the SourceForge subversion repository: https://sourceforge.net/p/daetools/code and located in the *trunk/OpenCS* directory.

The framework is based on the methodology for parallelisation of equation-based simulation programs on heterogeneous and distributed memory systems presented in Nikolić (2018, 2023). In the OpenCS approach, the model specification contains only the low-level information directly required by solvers and, in general, it can be generated from any modelling software. The OpenCS model specification, represented by a Compute Stack Model, provides a common interface to ODE/DAE solvers and can be generated using the OpenCS API in two ways (Fig. 2): (a) direct implementation in C++ or Python application programs, and (b) export of existing models from third-party simulators. The model specification data structures are stored as files in a binary format and used as inputs for parallel simulations on all platforms. This way, they provide a simple binary interface for model exchange. This approach differs from the typical model-exchange/co-simulation interfaces in that it does not require a human or a machine readable model definition as in modelling and model-exchange languages such as Modelica and CellML (https://www.cellml.org) nor a binary interface with the C API implemented in shared libraries as in Simulink (The MathWorks, Inc., 2018b) and Functional Mock-up Interface (https://www.fmi-standard.org). However, it must be kept in mind that the primary goal in the OpenCS framework is exchange of individual large-scale models for simulation on different high-performance computing platforms and whose equations can be evaluated on a single or simultaneously on multiple computing devices. Although technically possible, use of OpenCS models as building blocks in other simulators is not the major objective of OpenCS.

Model equations, specified in a symbolic form in infix notation, are transformed using the operator overloading technique into the bytecode and stored as an array of binary data (a Compute Stack) for direct evaluation on all platforms with no additional processing nor compilation steps. A limited set of bytecode instructions is utilised (only memory access to supplied data arrays and unary and binary mathematical operations). Individual equations (Compute Stacks) are evaluated by a stack machine (Compute Stack Machine) using the Last In First Out (LIFO) queues. Systems of equations are evaluated in parallel using a Compute Stack Evaluator interface which manages the Compute Stack Machine kernels. Two APIs/frameworks are used for parallelism: (a) the Open Multi-Processing (OpenMP) API for parallelisation on general purpose processors (multi-core CPUs), and (b) the Open Computing Language (OpenCL) framework for parallelisation on streaming processors (GPU, FPGA) and heterogeneous

systems (CPU+GPU, CPU+FPGA). Switching to a different computing device for evaluation of model equations is straightforward and controlled by an input parameter.

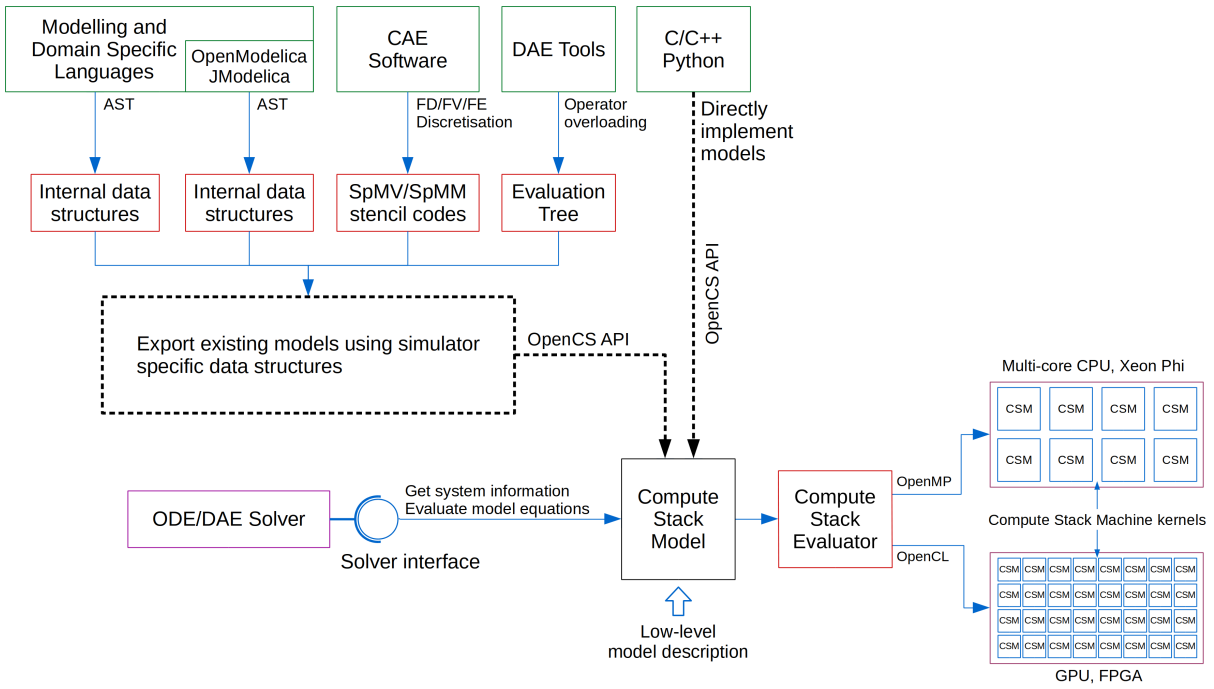

Figure 2. The OpenCS modelling approach. The (low-level) model specification is created using the OpenCS API and stored in a Compute Stack Model data structure which provides a generic interface to ODE/DAE solvers. Model equations, transformed into the postfix notation and stored as an array of binary data, are evaluated using the Compute Stack Machine kernels managed by a Compute Stack Evaluator.

A generic simulation software is provided by the framework to utilise the low-level information stored in Compute Stack Models (Nikolić, 2023). Simulations can be executed sequentially on a single processor or in parallel on message passing multiprocessors, where every processing element integrates one part (sub-system) of the overall ODE/DAE system in time and performs an inter-process communication between the processing elements (Fig. 3). Simulation inputs are specified in a generic fashion as files in a (platform independent) binary format. The input files are generated using the OpenCS API (one set per processing element) and contain the serialised model specification data structures and solver options. Simulation results are available in HDF5 (https://www.hdfgroup.org) or Comma Separated Value (.csv) formats.

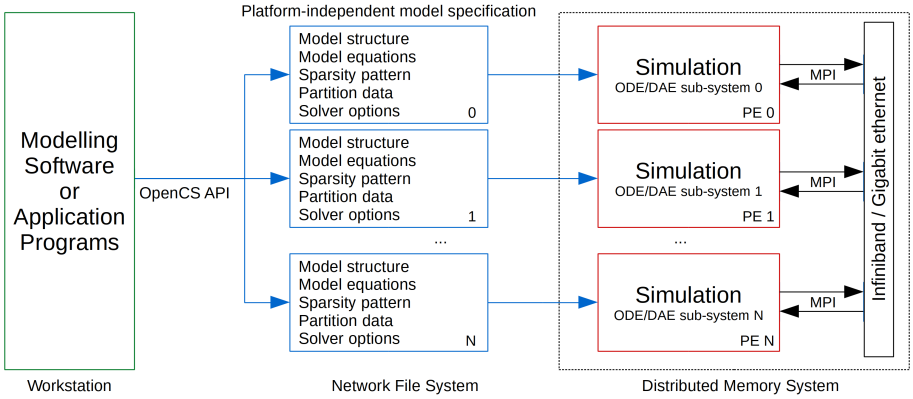

Figure 3. Parallel simulation on distributed memory systems using the OpenCS framework

The typical applications of the OpenCS framework include:

1. Development of custom large-scale models in C++ and Python.
2. Parallel evaluation of model equations (i.e. in simulators with no support for parallel evaluation or using the computing devices which are currently not utilised).
3. Parallel simulation on shared and distributed memory systems.
4. Model-exchange.
5. Use as a simulation engine for Modelling or Domain Specific Languages.
6. Benchmarks between different simulators, ODE/DAE solvers, computing devices and high performance computing systems (since a common model-specification is used on all platforms, OpenCS models can be used to benchmark memory and computation performance of individual computing devices or high performance computing systems; for example, benchmarks between heterogeneous CPU/GPU and CPU/FPGA systems could be performed without re-implementation of the model for a completely different architecture).

The article is organised in the following way. First, the methodology, the key concepts, data structures, and the implementation details are presented. Next, an API for typical applications accompanied with the sample ODE/DAE problems are analysed. Finally, a summary of the most important capabilities of the OpenCS framework and directions for future work are given in the last section.

METHODOLOGY

The framework is based on the methodology for parallel numerical solution of general systems of differential and algebraic equations on heterogeneous and distributed memory systems presented in Nikolić (2018, 2023). The methodology consists of the following parts:

1. A method for transformation of model equations into a data structure suitable for parallel evaluation on different computing platforms (the Compute Stack approach).
2. Data structures for model specification.
3. An algorithm for partitioning of general systems of equations.
4. An algorithm for inter-process data exchange.
5. Simulation software for integration of general ODE/DAE systems in time.

The key concepts and data structures

The OpenCS methodology is based on several concepts, each providing a distinct functionality (Nikolić, 2023): Compute Stack, Compute Stack Machine, Compute Stack Evaluator, Compute Stack Model, Compute Stack Differential Equations Model, Compute Stack Simulator, Compute Stack Model Builder, Compute Stack Number.

Model equations

In the Compute Stack approach, model equations are specified in a symbolic form in infix notation and internally transformed into the Reverse Polish (postfix) notation (Nikolić, 2018). Each mathematical operation and its operands are described by a specially designed bytecode data structure (*csComputeStackItem_t*) and every equation is transformed into an array of these structures (a Compute Stack). In general, any type of expressions involving standard mathematical operators and functions, numerical constants, variables and their derivatives are supported (linear or non-linear, algebraic or differential equations). Most of the functions from C numerics library (the <math.h> header) are available such as: unary (+, -) and binary operators (+, -, * and /), and unary (sqrt, log, log10, exp, floor, ceil, fabs, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh and erf) and binary functions (min, max, pow, atan2).

Individual equations are evaluated by a stack machine (Compute Stack Machine) using a LIFO queue. Due to its simplicity, the Compute Stack Machine can be implemented on virtually all computing devices (Nikolić, 2018). Since the model equations are stored as an array of binary data they can be directly evaluated on all platforms with no additional processing nor compilation steps.

Systems of equations are stored in memory as a single one-dimensional array of *csComputeStackItem_t* objects populated with Compute Stacks from all equations. Parallel evaluation of systems of equations is performed through a common interface called a Compute Stack Evaluator. Two implementations are available: (a) the OpenMP API is used for parallelisation on general purpose processors, and (b) the OpenCL framework is used for parallelisation on streaming processors and heterogeneous systems. In the OpenMP implementation, every thread evaluates a chunk of the total number of equations, one at the time. In the OpenCL implementation, every work-item evaluates only a single equation. Each thread/work-item executes a for loop where mathematical operations are performed in a single IF block controlled by the type of mathematical operation.

Model specification data structures

In the OpenCS approach, the model specification contains only the low-level information directly required by ODE/DAE solvers, stored in the *csModel_t* data structure (Nikolić, 2023). For sequential simulations, the system is described by a single *csModel_t* object. For parallel simulations, the system is described by an array of *csModel_t* objects each holding information about one ODE/DAE sub-system. Every model contains the following data: (a) the model structure with the information about the variable names, types, absolute tolerances and initial conditions: *csModelStructure_t* structure, (b) the model equations represented by Compute Stack arrays: *csModelEquations_t* structure, (c) the sparsity pattern of the ODE/DAE (sub-) system (required for evaluation of derivatives): *csSparsity-Pattern_t* structure, (d) the data for exchange of adjacent unknowns between processing elements using the MPI interface: *csPartitionData_t* structure, and (e) the Compute Stack evaluator instance: *csComputeStackEvaluator_t* object. Models are created using the Compute Stack Model Builder that provides an API for model specification and hides the implementation details of the OpenCS framework.

Model exchange capabilities and an interface to ODE/DAE solvers are provided by the *csDifferentialEquation-Model_t* class. It contains an instance of the *csModel_t* class and functions for loading of models from input files, retrieving the information and the sparsity pattern of the ODE/DAE system, setting the variable values/derivatives, exchanging the adjacent unknowns among the processing elements, and evaluating equations and derivatives.

Partitioning of general systems of equations

Large-scale numerical simulations on parallel computers require the distribution of equations among the processing elements so that the duration of each phase of the numerical solution is approximately the same. Therefore, the workload (storage and computation) in each phase and the inter-process communication volume must be well balanced among the processing elements for maximum performance. Computationally the most intensive phases of the numerical solution are: (1) evaluation of equations, (2) solution of a system of linear equations, and (3) evaluation of derivatives. Combined, they often amount to more than 95% of the total integration time (Nikolić, 2018). Since it is critical that every processor has an equal amount of work from each phase of the computation, the multiple quantities must be load balanced simultaneously.

The algorithm for partition of general systems of equations is described in Nikolić (2023). In the OpenCS framework, this algorithm is improved and re-implemented in C++ (for performance reasons). In the original algorithm, a graph of the ODE/DAE system is constructed and always partitioned using the METIS library (Karypis and Kumar, 1995). In this work, the graph partitioning is performed by the Compute Stack Graph Partitioner interface (*csGraphPartitioner_t* class) and separated from the main algorithm. This way, the algorithm can support the user-defined graph partitioners to exploit a problem-specific structure of model equations. At the moment, the following graph partitioner implementations are available: (1) Simple graph partitioner (*csGraphPartitioner_Simple*) - splits a graph into the specified number of partitions with no load balancing analysis (typically used for generation of Compute Stack models for sequential simulations), and (2) Metis graph partitioner (*csGraphPartitioner_Metis*) - partitions the graphs into a user-specified number k of parts using either the *Multilevel k-way partitioning paradigm* or the *Multilevel recursive bisectioning paradigm* implemented in METIS. (3) 2D-Npde graph partitioner (*csGraph-Partitioner_2D_Npde*) - partitions the specified number of partial differential equations (N_{pde}) distributed on a uniform two-dimensional grid by dividing the grid into the requested number of regions and with no load balancing analysis. The partitioning algorithm applies a static load balancing method. The workloads can be accurately and precisely estimated by taking into consideration several properties of equations and partitions. The partition properties used by the algorithm are: number of equations (N_{eq}), number of adjacent unknowns (N_{adj}), number of items in the Compute Stack array (N_{cs}), number of non-zero items in the partition's incidence matrix (N_{nz}), number of floating point operations (FLOPs) required for evaluation of equations (N_{flops}), and number of FLOPs required for evaluation of derivatives (N_{flops_j}). The memory and computation workloads in individual phases can be estimated using the partition properties as discussed in (Nikolić, 2023). N_{cs}, N_{nz}, N_{flops} and N_{flops_j} can be specified as additional balancing constraints for the graph partitioning. In particular, the number of FLOPs required for evaluation of equations and derivatives, that is the computation load, can be very accurately estimated by analysing the Compute Stack arrays. Moreover, the partitioning algorithm accepts a pair of dictionaries specifying the number of FLOPs for individual unary and binary mathematical operations (Nikolić, 2023). For instance, evaluation time of trigonometric functions on a traditional CPU is different from the evaluation time on a GPU. Thus, the algorithm can produce the load balanced partitions for diverse types of computing devices.

Partitioning of systems of equations in many cases is problem-specific and the generic graph partitioners often produce partitions with the excellent balance of workloads but poor overall simulation performance. The reason for this is the structure of partitions resulting in inefficient preconditioners and a high number of iterations to

reach convergence in the linear solver, as discussed in Nikolić (2023). Therefore, custom user-defined partitioners are required to take advantage of a problem-specific structure of model equations (i.e. the systems produced by discretisation of well known partial-differential equations on uniform grids and the systems which require the coupled treatment of all differential equations to ensure conservation such as the compressible Euler and incompressible Navier-Stokes equations).

Inter-process data exchange

Numerical solution on distributed memory systems requires an inter-process communication routine for exchange of adjacent unknowns (unknowns that belong to other processing elements). The algorithm for data exchange among processing elements is simple and only the point-to-point communication routines are required (Nikolić, 2023). It is fully generic and utilises the data resulting from the partitioning algorithm stored in the *csPartitionData_t* data structure. The algorithm always produces fully symmetrical point-to-point send/receive requests, and in addition, the send/receive data can be tested before the start of the simulation to prevent dead-locks or live-locks. The data exchange is performed asynchronously and the simulation resumes once all MPI requests are completed.

Generic simulation software

The OpenCS framework provides a simulator (*csSimulator*) for integration of general ODE and DAE systems in time (Nikolić, 2023). The simulator is cross-platform and can be executed sequentially on a single processor or in parallel on message passing multiprocessors. Simulation inputs are specified in a platform-independent binary format using input files with the model specification and run-time options. This way, the same model can be simulated using a single software on all platforms.

An overview of the solution procedure on shared memory systems is given in Fig. 4. The solution process consists of: (1) numerical integration in time, (2) linear algebra operations, (3) solution of systems of linear equations (in general, iterative methods are used for large scale systems), (4) (optionally) integration of sensitivity equations. The Compute Stack Evaluator is utilised by the Compute Stack Model for parallel evaluation of equations residuals (DAE systems) or the right hand side (ODE systems), and for evaluation of derivatives required for computation of the preconditioner and integration of sensitivity equations. Depending on the simulation options, the Compute Stack Evaluator can utilise a single or multiple computing devices.

The parallel solution on distributed memory systems requires the same tasks, but applied to integration of only one part of the overall system (ODE/DAE sub-system). Therefore, the software for numerical solution on shared memory systems is used as the main building block for distributed memory systems as depicted in Fig. 5. The additional functionality that is required includes: (a) an inter-process communication routine for exchange of adjacent unknowns, and (b) linear algebra routines for distributed memory systems (already available from the SUNDIALS suite). Both routines are implemented using the MPI C interface.

For integration of DAE systems in time the software uses the variable-step variable-order backward differentiation formula available in SUNDIALS IDAS solver (Hindmarsh et al., 2005). For integration of ODE systems in time the software uses the variable-step variable-order Adams-Moulton and backward differentiation formulas available in SUNDIALS CVodes solver (Serban and Hindmarsh, 2005). Systems of linear equations are solved using the sparse direct or Krylov-subspace iterative solvers. At the moment, the generalised minimal residual solver (GMRES) from the SUNDIALS suite and AztecOO and Amesos solvers from the Trilinos suite (Heroux et al., 2005) are available. All iterative solvers utilise IFPACK, ML and AztecOO built-in preconditioners available in the Trilinos suite.

Simulation inputs are specified using the data files with the serialised model specification data structures. The list of input files (one set for every processing element) is given in Table 1. *PE* in file names is an integer identifying the processing element equal to the value returned from *MPI_Comm_rank* function. For instance, *model_equations-00000.csdata* file is used by the PE with the rank 0. For sequential simulations a single set of input files is required. Each file contains a serialised data structure member of the *csModel_t* class: *csModelStructure_t*, *csModelEquations_t*, *csSparsityPattern_t* and *csPartitionData_t*. While the model specification remains unaltered, simulations can be performed for different time horizons, different solver and preconditioner options and using different computing devices for evaluation of model equations. Thus, the simulation options are specified in a human readable JSON format and contain four sections: *"Simulation"* (run-time data), *"Model"* (ODE/DAE model options), *"Solver"* (options for the ODE/DAE solver) and *"LinearSolver"* (the linear solver and the preconditioner options). Names of the solver/preconditioner parameters are identical to the original names used by the corresponding libraries or to the names of *Set_* functions (i.e. the *MaxOrd* parameter specified using the *IDASetMaxOrd* function in the SUNDIALS suite). The typical content of the simulation_option.json file for ODE and DAE problems are given in the supplemental source code listings S8 and S9, respectively.

Simulation results are saved in HDF5 (https://www.hdfgroup.org) or Comma Separated Value (.csv) formats into the output directory specified by the *Simulation.OutputDirectory* option. In addition, the detailed solvers statistics is generated for every processing element and saved in JSON format into the output directory.

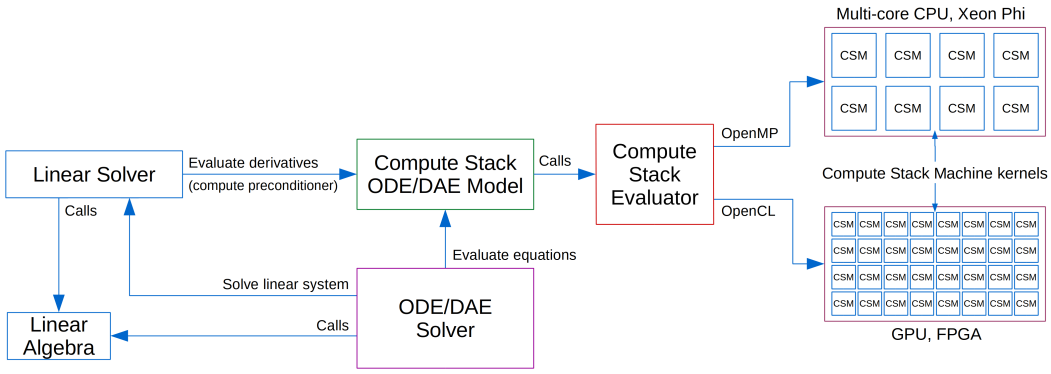

Figure 4. OpenCS simulation on shared memory systems

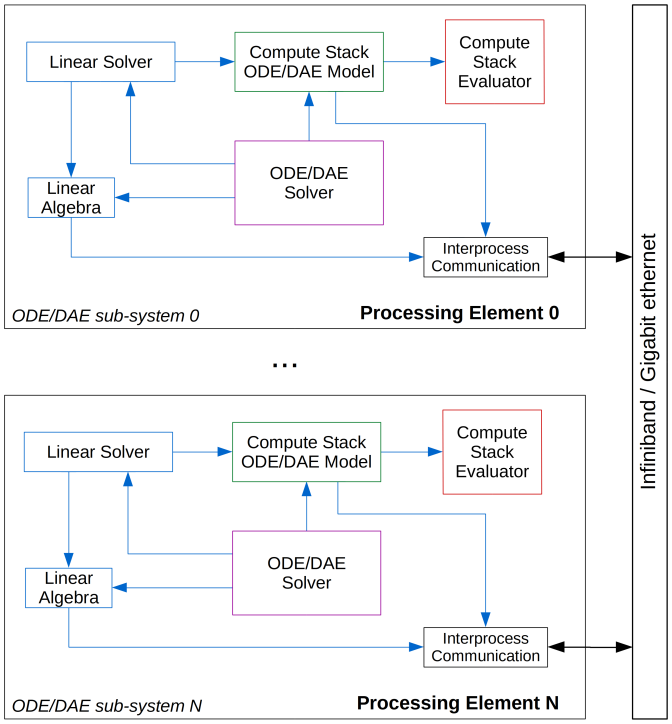

Figure 5. OpenCS simulation on distributed memory systems

Table 1. Input data files for OpenCS simulations.

Input file	Contents
model_structure-[PE].csdata	Serialised *csModelStructure_t* data structure
model_equations-[PE].csdata	Serialised *csModelEquations_t* data structure
sparsity_pattern-[PE].csdata	Serialised *csSparsityPattern_t* data structure
partition_data-[PE].csdata	Serialised *csPartitionData_t* data structure
simulation_options.json	Simulation, DAE and linear solver parameters

APPLICATION PROGRAMMING INTERFACE

The OpenCS framework provides an Application Programming Interface for model specification and typical applications such as: parallel evaluation of model equations, simulation on shared memory systems, simulation on distributed memory systems and model exchange. The key concepts of the OpenCS framework and the corresponding API are implemented in the following libraries: (1) cs_machine.h (header-only Compute Stack Machine implementation in C99), (2) libOpenCS_Evaluators (sequential, OpenMP and OpenCL Compute Stack Evaluator implementations), (3) libOpenCS_Models (Compute Stack Model, Compute Stack Differential Equations Model and Compute Stack Model Builder implementations), (4) libOpenCS_Simulators (Compute Stack ODE and DAE Simulator implementations) and a standalone simulator *csSimulator* (for both ODE and DAE problems). The framework internally utilises computing devices for evaluation of model equations and performs file system I/O operations and inter-process communication using the MPI interface (in parallel simulations). The structure and the main components of the framework are illustrated in Fig. 6.

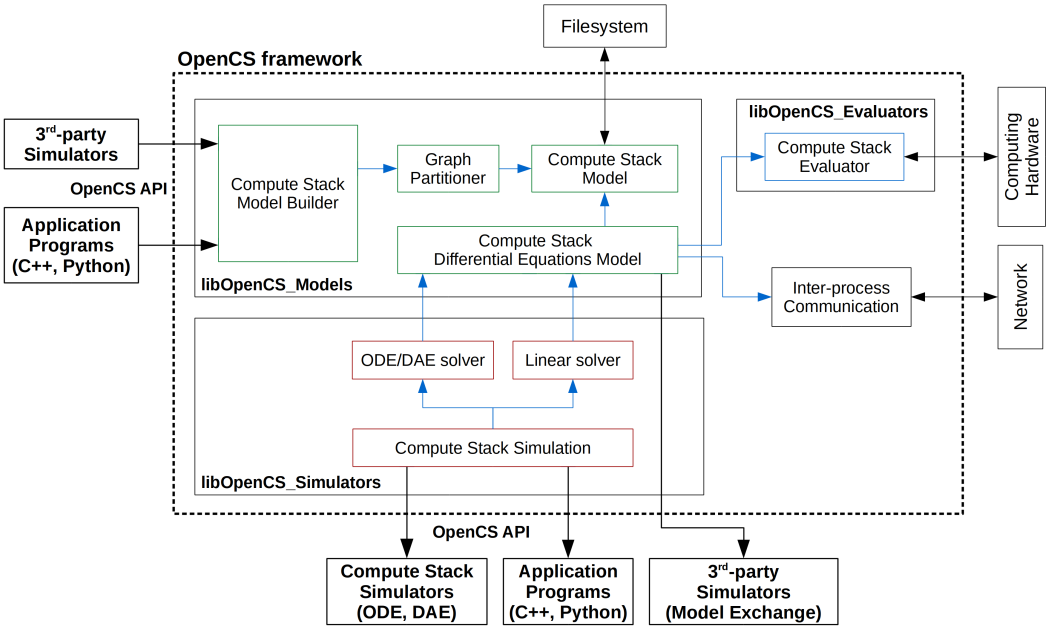

Figure 6. The structure and the main components of the OpenCS framework

Model specification

The OpenCS models are specified using the Model Builder interface (*csModelBuilder_t* class) in two ways: (1) direct implementation in C++/Python application programs, or (2) by exporting existing models in third party simulators. The procedure is practically identical in both cases. For simulations on shared memory systems it includes the following steps (source code listing 1):

Step 1. Initialise the Model Builder with the number of variables and the number of degrees of freedom. Other options such the default variable value, absolute tolerance, and variable name can be optionally set.

Step 2. Specify model equations using the provided *csNumber_t* objects representing variables, their time derivatives and degrees of freedom. *csNumber_t* is a user-defined Real number class providing all standard mathematical operators and functions as in the C numerics library (<math.h> header). This way, the equations expressions in OpenCS are identical to those implemented in C/C++.

Step 3. Set the initial values and names of variables and degrees of freedom, absolute tolerances etc. For DAE problems, a consistent set of initial conditions will be calculated before simulation.

Step 4. Generate Compute Stack models by partitioning the system of equations. For simulations on shared memory systems the whole system is used as a single partition and no graph partitioner is required.

Step 5. Use the generated model(s) directly or export them into a specified directory.

For parallel simulations on distributed memory systems the procedure is identical. The only difference is in the Step 4. where the system is partitioned into a specified number of partitions (one per processing element). The partitioning procedure for parallel simulations includes the following steps (source code listing 2):

Step 4.2 Instantiate one of the available graph partitioners (at the moment Simple, METIS and 2D_Npde) or provide a user-defined one. The METIS graph partitioner supports two algorithms: (1) *PartGraphKway* (Multilevel k-way partitioning algorithm), and (2) *PartGraphRecursive* (Multilevel recursive bisectioning algorithm). Optionally, change the default options of the partitioning algorithm.

Step 4.2.1 Specify the load balancing constraints. Four additional balancing constraints are available: N_{cs}, N_{nz}, N_{flops} and N_{flops_j}. For example, N_{cs} and N_{flops} can be used to balance the memory load (proportional to the number of Compute Stack items, N_{cs}) and the computation load for evaluation of model equations (proportional to the number of FLOPs for evaluation of equations, N_{flops}).

Step 4.2.2 Set the graph partitioner options (METIS specific).

Step 4.2.3 By default, the partitioning algorithm assumes that all mathematical operations require a single FLOP. This behaviour can be changed by specifying a pair of dictionaries with a number of FLOPs for individual mathematical operations: (1) *unaryOperationsFlops* for unary operators (+, -) and functions (sqrt, log, log10, exp, sin, cos, tan, ...), and (2) *binaryOperationsFlops* for binary operators (+, -, *, /) and functions (pow, min, max, atan2). If a mathematical operation is not in the dictionary, it is assumed that it requires a single FLOP. This way, the total number of FLOPs can be accurately estimated for every computing device.

Step 4.3 Partition the system into the specified number of processing elements (Npe).

Listing 1. Model specification procedure for simulation on shared memory systems

```
/* 1. Initialise the model builder with the number of variables
 *    and the number of degrees of freedom in the DAE system. */
csModelBuilder_t mb;
uint32_t Nvariables = ...;                          /* MODEL SPECIFIC PART */
uint32_t Ndofs      = ...;                          /* MODEL SPECIFIC PART */
mb.Initialize_DAE_System(Nvariables, Ndofs);

/* 2. Create and set model equations using the provided objects. */
const csNumber_t&              time  = mb.GetTime();
const std::vector<csNumber_t>& x     = mb.GetVariables();
const std::vector<csNumber_t>& dx_dt = mb.GetTimeDerivatives();
const std::vector<csNumber_t>& y     = mb.GetDegreesOfFreedom();

std::vector<csNumber_t> equations(Nvariables);
for(uint32_t i = 0; i < Nvariables; i++)
    equations[i] = F(x, dx_dt, y, time);            /* MODEL SPECIFIC PART */
mb.SetModelEquations(equations);

/* 3. Set the initial conditions, variable names, absolute tolerances, etc. */
std::vector<real_t>      x0            = {...};  /* MODEL SPECIFIC PART */
std::vector<std::string> varNames      = {...};  /* MODEL SPECIFIC PART */
std::vector<real_t>      absTolerances = {...};  /* MODEL SPECIFIC PART */
mb.SetVariableValues(x0);
mb.SetVariableNames(varNames);
mb.SetAbsoluteTolerances(absTolerances);

/* 4. Generate Compute Stack models by partitioning the DAE system. */
/* 4.1 Specify the output directory and simulation options.
 *     TimeHorizon and ReportingInterval are required options. */
std::string inputFilesDirectory = "...";
csSimulationOptionsPtr options = mb.GetSimulationOptions();
options->SetDouble("Simulation.TimeHorizon",       100.0);  /* MODEL SPECIFIC PART */
options->SetDouble("Simulation.ReportingInterval",   1.0);  /* MODEL SPECIFIC PART */
std::string simulationOptions = options->ToString();

/* 4.2 Partition the DAE system to generate a single Compute Stack model
 *     (graph partitioner is not required for sequential simulations). */
std::vector<csModelPtr> cs_models = mb.PartitionSystem(1, nullptr);

/* 5. Export the model(s) into a specified directory (or use them directly). */
mb.ExportModels(cs_models, inputFilesDirectory, simulationOptions);
```

Listing 2. Graph partitioning for simulation on distributed memory systems

```
/* 4.2 Instantiate METIS graph partitioner. */
csGraphPartitioner_Metis partitioner(PartGraphRecursive);

/* Change the input arguments of the partitioning algorithm. */
/* 4.2.1 Specify the load balancing constraints (optional). */
std::vector<std::string> balancingConstraints = {"Ncs", "Nflops"};

/* 4.2.2 Set the METIS partitioner options (optional). */
std::vector<int32_t> options = partitioner.GetOptions(); /* default values */
options[METIS_OPTION_NITER]   = 10;
options[METIS_OPTION_UFACTOR] = 30;
partitioner.SetOptions(options);

/* 4.2.3 Specify the number of FLOPs for mathematical operations (optional). */
std::map<csUnaryFunctions,uint32_t>  unaryOperationsFlops;
std::map<csBinaryFunctions,uint32_t> binaryOperationsFlops;
unaryOperationsFlops[eSqrt]    = 12; /* i.e. the sqrt function requires 12 FLOPs */
binaryOperationsFlops[eDivide] =  6; /* i.e. the operator / requires 6 FLOPs */

/* 4.3 Partition the system to generate Npe models (one per processing element). */
std::vector<csModelPtr> cs_models = mb.PartitionSystem(Npe, &partitioner,
                                                       balancingConstraints,
                                                       true,
                                                       unaryOperationsFlops,
                                                       binaryOperationsFlops);
```

Model exchange and parallel evaluation of model equations

The main goal of the OpenCS framework is specification of large scale equation-based models for simulation on shared and distributed memory systems. In addition, the developed models can also be used for model-exchange and for parallel evaluation of model equations (to improve the simulation performance in existing simulators). The Compute Stack Differential Equations Model is used for loading a model into a host simulator and as a common interface to the data required for integration in time by ODE/DAE solvers (i.e. evaluation of equations and derivatives). The procedure is identical in both cases and includes the following steps (source code listing 3):

Step 1. Initialise MPI.

Step 2. Instantiate the *csDifferentialEquationModel* object (a reference implementation of the *csDifferentialEquationModel_t* interface).

Step 3. Load the model from the specified directory with input files (or load the existing Compute Stack Model objects directly).

Step 4. Instantiate and set the Compute Stack Evaluator. In this example the OpenMP Compute Stack Evaluator is used. It accepts the number of threads as an argument in its constructor. If zero is specified, the default number of threads will be used (typically equal to the number of cores).

Step 5. Obtain the necessary information from the model such as the number of variables, variable names, types, absolute tolerances, initial conditions and the sparsity pattern in the Compressed Row Storage (CRS) format.

Step 6. Evaluate model equations and derivatives (typically in a loop).

- **Step 6.1** Set the current values of state variables and derivatives using the *SetAndSynchroniseData* function. At this point, for simulations on message passing multiprocessors the MPI interface will be used to exchange the adjacent unknowns between processing elements.
- **Step 6.2** Evaluate equations residuals (for DAE problems) or a right hand side (for ODE problems) using the *EvaluateEquations* function.
- **Step 6.3** Evaluate derivatives (the Jacobian matrix) using the *EvaluateJacobian* function. Here, *csMatrixAccess_t* is used as a generic interface to the sparse matrix storage in linear solvers. *inverseTimeStep* is an inverse of the current step taken by the solver. *SetAndSynchroniseData* should be called only before a call to the *EvaluateEquations* function. It is assumed that *SetAndSynchroniseData* has already been called and the current values set and exchanged between processing elements. This is a typical procedure in ODE/DAE solvers where the model equations are always evaluated first and then, if required, the derivatives re-evaluated and a preconditioner re-computed (in iterative methods) or the Jacobian matrix re-factored (in direct methods).

Step 7. Free the resources allocated in the model and the evaluator.

Step 8. Finalise MPI.

Listing 3. Procedure for model exchange

```
/* 1. Initialise MPI. */
int rank;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* 2. Instantiate the Compute Stack model. */
csDifferentialEquationModel model;

/* 3. Load the model from the specified directory with input files. */
model.Load(rank, inputFilesDirectory);

/* 4. Instantiate and set the Compute Stack Evaluator. */
csComputeStackEvaluator_OpenMP evaluator(0);
model.SetComputeStackEvaluator(&evaluator);

/* 5. Get the information about the model structure, */
/*    i.e. the number of equations and variable names */
/*    and the sparsity pattern (in CRS format): */
csModelStructure_t& ms = model.GetModel()->structure;
uint32_t                  Neqns    = ms.Nequations;
std::vector<std::string>& varNames = ms.variableNames;
int N, Nnz;
std::vector<int> IA, JA;
model.GetSparsityPattern(N, Nnz, IA, JA);

/* 6. Evaluate model equations and derivatives (typically in a loop). */
/*    6.1 Set the values/derivatives of state variables. */
model.SetAndSynchroniseData(time, x, dx_dt);

/*    6.2 Evaluate residuals/Right Hand Side. */
model.EvaluateEquations(time, residuals);

/*    6.2 Evaluate derivatives (the Jacobian matrix). */
model.EvaluateJacobian(time, inverseTimeStep, ma);

/* 7. Free the resources allocated in the model and the evaluator. */
model.Free();

/* 8. Finalise MPI. */
MPI_Finalize();
```

Simulation on shared memory systems

Simulation on shared memory systems is performed by embedding a simulation into a host simulator or using a standalone OpenCS simulator (*csSimulator*). Simulations using the standalone *csSimulator* are performed by executing the simulator with a single argument specifying the directory with input files. Simulations are started using the OpenCS *cs::Simulate* function (the source code listing 4) or, if a user-defined schedule is required, using the OpenCS simulation API (the source code listing 5). In the latter case, the procedure includes the following steps:

Step 1. Initialise MPI.
Step 2. Load the simulation_options.json and get run-time options.
Step 3. Instantiate model, simulation and ODE/DAE solver objects.
Step 4. Load the model from the input directory.
Step 5. Create and set the Compute Stack Evaluator. The application-specific evaluator can be instantiated or the information about the type of evaluator and its parameters can be obtained from the "Model.ComputeStackEvaluator" section.
Step 6. Initialise the simulation.
Step 7. Calculate corrected initial conditions at time = 0 (for DAE systems only).
Step 8. Run the simulation using the default Run function or implement a custom schedule using the functions provided by the *daeSimulation_t* class.

Step 9. Print the solver stats and finalise the simulation.
Step 10. Free the resources allocated in the model and the evaluator.
Step 11. Finalise MPI.

Listing 4. Simulation on shared memory systems using the *Simulate* function

```
/* 1. Initialise MPI. */
MPI_Init(&argc, &argv);

/* 2a. Run simulation using the input files from the specified directory: */
cs::Simulate(inputFilesDirectory);

/* 2b. Run simulation using an existing model: */
csModelPtr  model;                    // model is generated by the Model Builder
std::string simulationOptions = ...; // options in JSON format
std::string outputDirectory   = ...; // a directory for simulation outputs
cs::Simulate(model, simulationOptions, outputDirectory);

/* 3. Finalise MPI. */
MPI_Finalize();
```

Listing 5. Simulation on shared memory systems using the simulation API

```
/* 1. Initialise MPI. */
MPI_Init(&argc, &argv);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

std::string inputFilesDirectory = "...";

/* 2. Load the simulation_options.json and get run-time options. */
std::string simulationOptionsFile = inputFilesDirectory + "/simulation_options.json";
daeSimulationOptions& cfg = daeSimulationOptions::GetConfig();
cfg.Load(simulationOptionsFile);

std::string outputDirectory = cfg.GetString("Simulation.OutputDirectory");

/* 3. Instantiate model, simulation and ODE/DAE solver objects. */
daeModel_t      model;
daeSolver_t     daesolver;
daeSimulation_t simulation;

/* 4. Load the model from the input directory. */
model.Load(rank, inputFilesDirectory);

/* 5. Create and set the Compute Stack Evaluator
 *    (in general, using the data from the "Model.ComputeStackEvaluator" section). */
csComputeStackEvaluator_Sequential evaluator;
model.SetComputeStackEvaluator(&evaluator);

/* 6. Initialise the simulation. */
simulation.Initialize(&model, &daesolver,
                      startTime, timeHorizon, reportingInterval, outputDirectory);

/* 7. Calculate corrected initial conditions at time = 0 (DAE systems only). */
simulation.SolveInitial();

/* 8. Run the simulation using the default Run function
 *    (or implement a custom schedule). */
simulation.Run();

/* 9. Print the solver stats and finalise the simulation. */
simulation.PrintStats();
simulation.Finalize();

/* 10. Free the resources allocated in the model and the evaluator. */
```

```
model.Free();

/* 11. Finalise MPI. */
MPI_Finalize();
```

Simulation on distributed memory systems

Simulation on distributed memory systems is typically performed using the standalone *csSimulator* and the parallel jobs are started using the commands specific to a particular implementation of the MPI standard. Examples for three different operating systems (GNU/Linux, Windows and macOS) and MPI implementations are given in the source code listing 6. Details on the available options for starting the parallel jobs can be found in the documentation of a particular implementation.

Listing 6. Simulation on distributed memory systems using the standalone OpenCS simulator

```
# 1. GNU/Linux (i.e. using OpenMPI)
# On a local machine:
$ mpirun -np <Npe> csSimulator "inputFilesDirectory"
# On multiple nodes:
$ mpirun --hostfile <hostfilename> -np <Npe> csSimulator "inputFilesDirectory"

# 2. Windows (i.e. using MS-MPI):
# On a local machine:
$ mpiexec /np <Npe> csSimulator "inputFilesDirectory"
# On multiple nodes:
$ mpiexec /gmachinefile <hostfilename> /np <Npe> csSimulator "inputFilesDirectory"

# 3. macOS (i.e. using MPICH):
# On a local machine:
$ mpiexec -n <Npe> csSimulator "inputFilesDirectory"
# On multiple nodes:
$ mpiexec -f <hostfilename> -n <Npe> csSimulator "inputFilesDirectory"
```

APPLICATIONS

Case 1: transient two-dimensional diffusion-reaction equations, uniform grid

The model describes the process of auto-catalytic chemical reaction with oscillations known as the Brusselator PDE. The net reaction is $A+B \rightarrow D+E$ with transient appearance of intermediates X and Y, where A and B are reactants and D and E are products (Strogatz, 1994). The model is originally implemented using SUNDIALS IDAS suite (Serban and Hindmarsh, 2016). Under conditions where components A and B are in vast excess during the chemical reaction the system dynamics is described by the following equations (DAE system):

$$\begin{aligned} \frac{du}{dt} &= k_1 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) + R_u(u,v,t) \\ \frac{dv}{dt} &= k_2 \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) + R_v(u,v,t) \end{aligned} \tag{1}$$

where the reaction rates R_u and R_v are defined as:

$$\begin{aligned} R_u(u,v,t) &= u^2 v - (B+1)u + A \\ R_v(u,v,t) &= -u^2 v + Bu \end{aligned} \tag{2}$$

Here, k_1 and k_2 are diffusion constants, A and B are the concentrations of components A and B, and u and v are concentration of intermediaries X and Y. The equations are distributed on a square domain $x \in [0,10]$ and $y \in [0,10]$ and discretised by central differencing on a uniform 1000x1000 spatial mesh resulting in 2,000,000 unknowns. The boundary conditions are homogeneous Neumann (no normal flux at boundaries). The initial conditions are given by: $u(x,y,t_0) = 1.0 - 0.5\cos(\pi y)$ and $v(x,y,t_0) = 3.5 - 2.5\cos(\pi x)$. The concentrations of components A and B and the diffusion constants are held constant ($A = 1$, $B = 3.4$, $k_1 = k_2 = 0.002$). The relative and absolute tolerances for all unknowns are set to 10^{-5}. The system is simulated for 10 seconds and the outputs are taken every 0.1 second. The C++ source code and the compilation and usage instructions are given in the Supplemental Listing S10. The

precompiled C++ binary is available in the OpenCS distribution. In addition, the C++ and Python source code can be found on the OpenCS website (dae_example_3, https://daetools.sourceforge.io/opencs-examples.html).

Five different runs have been performed (Table 2). Simulations (that is integration in time) in the first four runs are performed on a single processor. Model equations are evaluated sequentially in run C1-SEQ, using the OpenMP API in C1-OMP (8 core CPU), and using the OpenCL framework in C1-CL (single device setup, all equations evaluated on a GPU) and C1-HET (heterogeneous CPU/GPU setup where 70% of equations are evaluated on a GPU and 30% on a CPU). In C1-MPI run the system is partitioned by dividing the 2D mesh into eight quadrants, the simulation carried out on 8 processors using the MPI interface while the model equations are evaluated sequentially in every processing element. The DAE system is integrated in time using the variable-step variable-order backward differentiation formula from SUNDIALS IDAS solver (Hindmarsh et al., 2005). Systems of linear equations are solved using the SUNDIALS GMRES solver and the IFPACK ILU (Sala and Heroux, 2005) preconditioner from Trilinos suite (in the original IDAS model the band-block-diagonal preconditioner has been applied). The input parameters for the IFPACK preconditioner are given in Table 3 where k is the fill-in factor, α is the absolute threshold, ρ is the relative threshold and ω is the relax value. The simulations are carried out in 64-bit Debian Stretch GNU/Linux and compiled using the gcc 6.3 compiler, OpenCS 1.1.0, MPI-3.1 from the Open MPI v2.0.2 package, OpenMP 4.5 from the GOMP library, and OpenCL 1.2 from NVidia CUDA 9.0 with v384.90 display driver. The hardware configuration consists of Intel i7-6700HQ CPU (4 cores/8 threads at 2.6 GHz, 8 GB of RAM, 34.32 GFLOPs peak double precision) and a discrete NVidia GeForce GTX 950M GPU (640 execution units at 914 MHz, 2 GB of RAM, 36.56 GFLOPs peak double precision).

Table 2. Case 1: Simulation runs

Run	Simulation	Evaluation of model equations
C1-SEQ	1 CPU	Sequential
C1-OMP	1 CPU	OpenMP (8 threads)
C1-CL	1 CPU	Single device OpenCL (100% on GPU)
C1-HET	1 CPU	Heterogeneous OpenCL (70% on GPU, 30% on CPU)
C1-MPI	8 CPUs	Sequential on every PE

Table 3. Case 1: IFPACK ILU preconditioner parameters

Run	k	ρ	α	ω
C1-SEQ	3	1.0	0.1	0.5
C1-OMP	3	1.0	0.1	0.5
C1-CL	3	1.0	0.1	0.5
C1-HET	3	1.0	0.1	0.5
C1-MPI	1	1.0	0.1	0.0

Case 2: transient two-dimensional convection-diffusion-reaction equations, uniform grid

The model describes the Chapman mechanism for ozone kinetics arising in athmospheric simulations (Schiesser and Lapidus, 1976). The reaction involves three components: ozone singlet (O), ozone (O_3) and oxygen (O_2), where the first two reactions are photo-chemical and contain diurnal rate coefficients. The model is originally presented in Wittman (1996) and implemented using SUNDIALS CVodes suite (Serban and Hindmarsh, 2015). The system dynamics is described by the following equations (ODE system):

$$\frac{dc_i}{dt} = K_h \frac{\partial^2 c_i}{\partial x^2} + \frac{\partial}{\partial y}\left(K_v(y)\frac{\partial c_i}{\partial y}\right) + V\frac{\partial c_i}{\partial x} + R_i(c_1, c_2, t),\ i = 1,2 \tag{3}$$

where the reaction rates R_1 and R_2 are given by:

$$\begin{aligned} R_1(c_1, c_2, t) &= -q_1 c_1 c_3 - q_2 c_1 c_2 + 2q_3(t)c_3 + q_4(t)c_2 \\ R_2(c_1, c_2, t) &= q_1 c_1 c_3 - q_2 c_1 c_2 - q_4(t)c_2 \end{aligned} \tag{4}$$

Here, c_1, c_2 and c_3 are concentrations of O, O_3 and O_2, respectively, q_1, q_2, q_3 and q_4 are reaction rate coefficients, V is velocity, and K_h and K_v are diffusion coefficients. The numerical values of the input parameters are: $V = 10^{-3}$,

$K_h = 4 \cdot 10^{-6}$ and $K_v(y) = 10^{-8}\exp(0.2y)$. q_1, q_2 and c_3 are constant ($q_1 = 1.63 \cdot 10^{-16}$, $q_2 = 4.66 \cdot 10^{-16}$, $c_3 = 3.7 \cdot 10^{16}$) while q_3 and q_4 vary diurnally:

$$q_3(t) = \begin{cases} \exp\left(-A_3/\sin(\omega t)\right), & \text{if } \sin(\omega t) > 0 \\ 0, & \text{otherwise} \end{cases}$$
$$q_4(t) = \begin{cases} \exp\left(-A_4/\sin(\omega t)\right), & \text{if } \sin(\omega t) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

where $\omega = \pi/43200$ and A_3 and A_4 are coefficients ($A_3 = 22.62$, $A_4 = 7.601$). The equations are distributed on a square domain: $x \in [0,20]$ km and $y \in [30,50]$ km and discretised by central differencing on a uniform 1000x500 spatial mesh resulting in 1,000,000 unknowns. In the original CVodes model the equations were also discretised using the central differences, except for the advection term where a biased 3-point difference formula was used. The boundary conditions are homogeneous Neumann (no normal flux at boundaries). The initial conditions are given by:

$$\begin{aligned} c_1(x,y,t_0) &= 10^6 \alpha(x)\beta(y) \\ c_2(x,y,t_0) &= 10^{12} \alpha(x)\beta(y) \\ \alpha(x) &= 1 - (0.1(x - x_{mid}))^2 + 0.5(0.1(x - x_{mid}))^4 \\ \beta(y) &= 1 - (0.1(y - y_{mid}))^2 + 0.5(0.1(y - y_{mid}))^4 \end{aligned} \tag{6}$$

where $x_{mid} = (0+20)/2 = 10$ and $y_{mid} = (30+50)/2 = 40$ are mid points of the x, y domains. The relative and absolute tolerances for all unknowns are set to 10^{-5}. The system is integrated for 86,400 seconds (1 day) and the outputs are taken every 100 seconds. The C++ source code and the compilation and usage instructions are given in the Supplemental Listing S11. The precompiled C++ binary is available in the OpenCS distribution. In addition, the C++ and Python source code can be found on the OpenCS website (ode_example_3, https://daetools.sourceforge.io/opencs-examples.html).

Five different runs have been performed (Table 4) identical to those in Case 1. The ODE system is integrated in time using the variable-step variable-order backward differentiation formula available in SUNDIALS CVodes solver (Serban and Hindmarsh, 2005). Systems of linear equations are solved using the SUNDIALS generalised minimal residual solver and the IFPACK ILU (Sala and Heroux, 2005) preconditioner from Trilinos suite (in the original CVodes model the 2x2 block-diagonal preconditioner has been applied). The input parameters for the IFPACK preconditioner for all runs are given in Table 5. The simulations are carried out using the same software and hardware as in Case 1.

Table 4. Case 2: Simulation runs

Run	Simulation	Evaluation of model equations
C2-SEQ	1 CPU	Sequential
C2-OMP	1 CPU	OpenMP (8 threads)
C2-CL	1 CPU	Single device OpenCL (100% on GPU)
C2-HET	1 CPU	Heterogeneous OpenCL (70% on GPU, 30% on CPU)
C2-MPI	8 CPUs	Sequential on every PE

Table 5. Case 2: IFPACK ILU preconditioner parameters

Run	k	ρ	α	ω
C2-SEQ	1	1.0	10^{-5}	0.0
C2-OMP	1	1.0	10^{-5}	0.0
C2-CL	1	1.0	10^{-5}	0.0
C2-HET	1	1.0	10^{-5}	0.0
C2-MPI	1	1.0	0.1	0.0

RESULTS

Four main and four sub-phases of the numerical solution are analysed:

1. EvaluateEquations – evaluation of model equations (residuals or right-hand side).
2. LinearSystemSetup – setup of the linear equations solver, with two sub-phases:
   2.1. EvaluateJacobian – evaluation of a Jacobian matrix.
   2.2. ComputePreconditioner – computation of a preconditioner using the Jacobian data.
3. LinearSystemSolve – solution of a linear system of equations, with two sub-phases:
   3.1. ApplyPreconditioner – application of the preconditioner to solve the linear system.
   3.2. JacobianVectorProduct – Jacobian-vector multiplication, required in every iteration of the linear solver (in SUNDIALS the difference quotient approximation is used and requires an additional call to the EvaluateEquations function).
4. InterProcessDataExchange – exchange of adjacent unknowns between processing elements, required before every call to EvaluateEquations.

The number of equations (N_{eq}), the number of non-zero items in the Jacobian matrix (the total number $N_{nz} = \sum_{i=1}^{N_{eq}} N_{nz}[i]$ and the average number per equation $N_{nz/equation}$), the number of Compute Stack items (the total number $N_{cs} = \sum_{i=1}^{N_{eq}} N_{cs}[i]$ and the average number per equation $N_{cs/equation}$) and the average number of Compute Stack items for evaluation of a single row of the Jacobian matrix ($N_{cs/jacob\_row} = \frac{1}{N_{eq}} \sum_{i=1}^{N_{eq}} N_{nz}[i] N_{cs}[i]$) for the sequential runs in both cases are given in Table 6.

The numerical results are compared to the original SUNDIALS IDAS (Serban and Hindmarsh, 2016) and CVodes (Serban and Hindmarsh, 2015) models. Comparison of the concentration u at the bottom-left point (x=0, y=0) between the OpenCS and the original IDAS model for two different operating regimes are presented in Fig. 7 (stable regime for $B = 1.7$) and Fig. 8 (unstable regime for $B = 3.4$). Comparison of the concentration c_1 at the bottom-left point (x=0, y=30) between the OpenCS and the original CVodes model is presented in Fig. 9.

The total integration time, the duration of individual phases of the numerical solution and the percentage of the total integration time in individual phases are presented in Table 7 for Case 1 and Table 8 for Case 2.

The speed-ups of individual phases of the numerical solution, the maximum theoretical overall speed-ups and the achieved overall simulation speed-ups are given in Table 9 for Case 1 and Table 10 for Case 2. The maximum theoretical speed-ups for evaluation of model equations can be determined using the maximum peak performance for individual platforms. For instance, for C1-CL and C2-CL runs the theoretical speed-up is 36.56 GFLOPs/(34.32 GFLOPs/8 cores) = 8.52, for C1-OMP and C2-OMP runs it is 8.00 (the number of cores), while for C1-HET and C2-HET runs it is 8.52 (GPU) + 8.00 (CPU) = 16.52. The maximum theoretical overall simulation speed-ups can be calculated from the Amdahl's law using the data from Tables 7 and 8 and the maximum peak performance for individual platforms: $1/\left(1-p+p/s\right)$, where p is the portion of the solution that can be parallelised and s is the maximum theoretical speed-up. For runs that utilise OpenMP and OpenCL (only the model equations and derivatives are evaluated in parallel) they are: (a) 2.02 for C1-OMP and 3.78 for C2-OMP, (b) 2.04 for C1-CL and 3.87 for C2-CL, and (c) 2.19 for C1-HET and 4.75 for C2-HET. The maximum theoretical overall speed-up for the MPI runs in an ideal case is 8.00 (the number of processing elements). However, not all parts of the simulation can be parallelised and the performance in some phases does not scale linearly with the number processing elements. In addition, there is an overhead due to the load imbalance and time required for inter-process communication. In this work, as a rough estimate, it is assumed that evaluating model equations and solving linear systems of equations can be parallelised. Combined, they amount to 89.5% (Case 1) and 95.7% (Case 2) of the total integration time in sequential runs. Thus, the maximum theoretical overall speed-up for MPI runs is 4.61 for Case 1 and 6.14 for Case 2.

Table 6. Workload-related properties for sequential runs

	N_{eq}	N_{nz}	N_{cs}	$N_{nz/equation}$	$N_{cs/equation}$	$N_{cs/jacob\_row}$
Case 1	2,000,000	11,976,024	67,832,168	5.98	33.92	203.09
Case 2	1,000,000	5,988,000	69,928,000	5.99	69.93	418.73

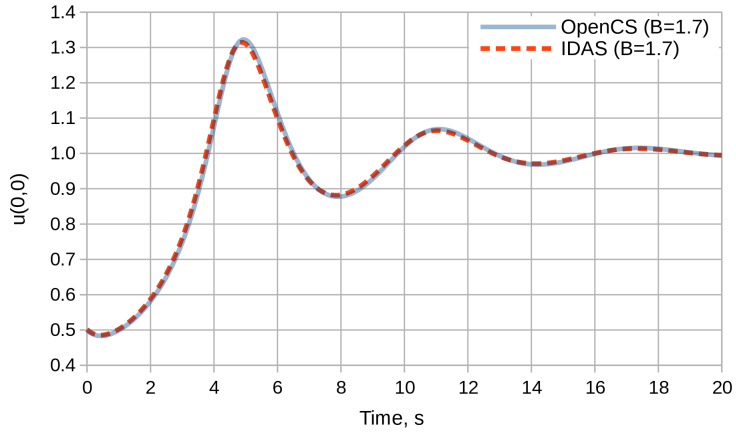

Figure 7. Case 1: Plot of the concentration u at the bottom-left point (x=0, y=0) - stable regime ($B = 1.7$)

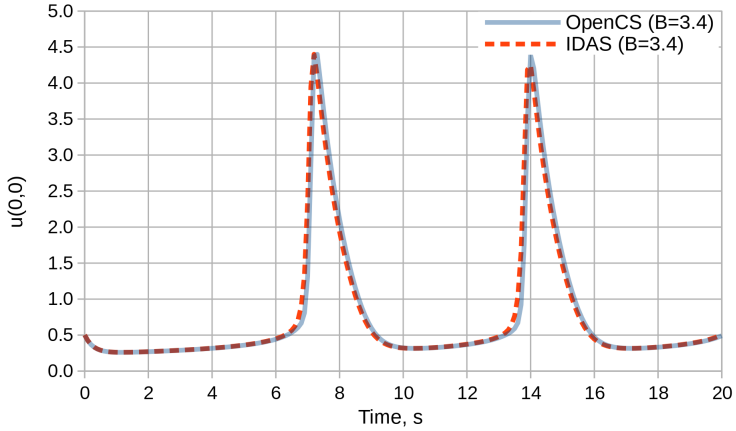

Figure 8. Case 1: Plot of the concentration u at the bottom-left point (x=0, y=0) - unstable regime ($B = 3.4$)

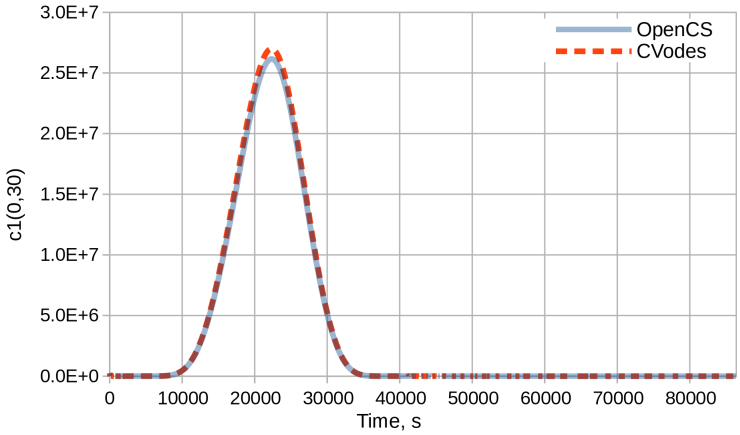

Figure 9. Case 2: Plot of the concentration c_1 at the bottom-left point (x=0, y=30)

Table 7. Case 1: Execution times for individual phases of the numerical solution

	C1-SEQ		C1-OMP		C1-CL		C1-HET		C1-MPI	
Phase	Time, s	%	Time, s	%	Time, s	%	Time, s	%	Time, s	%
EvaluateEquations	211.96	20.53	60.22	11.43	69.56	13.26	54.37	11.27	61.64	15.06
LinearSystemSetup (total)	65.14	6.31	39.06	7.41	33.31	6.35	33.83	7.02	9.71	2.37
EvaluateJacobian	29.85	2.89	9.88	1.87	6.45	1.23	7.16	1.49	6.47	1.58
ComputePreconditioner	35.29	3.42	29.18	5.54	26.86	5.12	26.67	5.53	3.24	0.79
LinearSystemSolve (total)	646.71	62.63	340.19	64.54	339.32	64.69	311.72	64.64	248.43	60.69
ApplyPreconditioner	192.46	18.63	158.78	30.12	146.63	27.96	145.70	30.22	56.88	13.89
JacobianVectorProduct	355.12	34.39	100.91	19.07	116.54	22.21	91.09	19.07	108.38	26.48
InterProcessDataExchange	-	-	-	-	-	-	-	-	15.54	3.81
Integration (total)	1023.87		522.06		520.05		477.76		407.44	

Table 8. Case 2: Execution times for individual phases of the numerical solution

	C2-SEQ		C2-OMP		C2-CL		C2-HET		C2-MPI	
Phase	Time, s	%	Time, s	%	Time, s	%	Time, s	%	Time, s	%
EvaluateEquations	570.70	29.57	132.84	20.39	132.58	21.73	95.58	17.58	85.39	17.51
LinearSystemSetup (total)	319.24	16.54	114.81	17.62	76.43	12.53	96.98	17.84	66.47	13.63
EvaluateJacobian	270.21	14.00	72.31	11.10	33.69	5.52	52.52	9.66	52.61	10.79
ComputePreconditioner	49.03	2.54	42.50	6.52	42.74	7.01	44.46	8.18	13.86	2.84
LinearSystemSolve (total)	956.58	49.56	328.86	50.48	325.11	53.29	275.08	50.59	287.82	59.03
ApplyPreconditioner	83.15	4.31	73.67	11.30	70.30	11.52	72.66	13.37	40.41	8.29
JacobianVectorProduct	781.10	40.47	181.82	27.91	181.78	29.79	131.08	24.11	173.82	35.66
InterProcessDataExchange	-	-	-	-	-	-	-	-	11.70	2.39
Integration (total)	1930.08		651.47		610.11		543.66		487.54	

Table 9. Case 1: Speed-ups for individual phases of the numerical solution, maximum theoretical speed-ups for each phase (in brackets) and the overall speed-ups

Phase	C1-OMP (Max.Th.)	C1-CL (Max.Th.)	C1-HET (Max.Th.)	C1-MPI (Max.Th.)
EvaluateEquations	3.52 (8.00)	3.05 (8.52)	3.90 (16.52)	3.36 (8.00)
EvaluateJacobian	3.02 (8.00)	4.63 (8.52)	4.17 (16.52)	4.36 (8.00)
ComputePreconditioner	-	-	-	10.29
ApplyPreconditioner	-	-	-	3.40
JacobianVectorProduct	3.52 (8.00)	3.05 (8.00)	3.90 (8.00)	3.36 (8.00)
Max. theor. overall	2.02	2.04	2.19	4.61
Overall	1.96 (97%)	1.97 (96%)	2.17 (98%)	2.51 (55%)

Table 10. Case 2: Speed-ups for individual phases of the numerical solution, maximum theoretical speed-ups for each phase (in brackets) and the overall speed-ups

Phase	C2-OMP (Max.Th.)	C2-CL (Max.Th.)	C2-HET (Max.Th.)	C2-MPI (Max.Th.)
EvaluateEquations	4.30 (8.00)	4.30 (8.00)	5.97 (16.52)	4.18 (8.00)
EvaluateJacobian	3.74 (8.00)	8.02 (8.00)	5.14 (16.52)	3.72 (8.00)
ComputePreconditioner	-	-	-	2.56
ApplyPreconditioner	-	-	-	1.66
JacobianVectorProduct	4.30 (8.00)	4.30 (8.00)	5.96 (8.00)	4.18 (8.00)
Max. theor. overall	3.78	3.87	4.75	6.14
Overall speed-up	2.96 (78%)	3.16 (82%)	3.55 (75%)	3.96 (64%)

DISCUSSION

Comparison of the numerical results between the OpenCS model and the original SUNDIALS IDAS (for Case 1, Fig. 7 and 8) and CVodes (for Case 2, Fig. 9) models show a good agreement. The observed small variations can be attributed to the internal implementation details: SUNDIALS models use different (and less efficient) preconditioners and, in addition, a different discretisation method has been applied to the advection term in Case 2.

The overall performance is in the following order for both cases: sequential < OpenMP < OpenCL (GPU) < OpenCL (CPU+GPU) < MPI runs (Tables 9 and 10) and agree well with the theoretical limits. The achieved overall simulation speed-ups in Case 1 are 1.96, 1.97, 2.17 and 2.51 for C1-OMP, C1-CL, C1-HET and C1-MPI runs, respectively (97, 96, 98 and 55% of the maximum theoretical overall speed-up, respectively). In Case 2, the achieved overall simulation speed-ups are 2.96, 3.16, 3.55 and 3.96 for C2-OMP, C2-CL, C2-HET and C2-MPI runs, respectively (78, 82, 75 and 64% of the maximum theoretical overall speed-up, respectively).

In the OpenMP and the OpenCL runs, the simulation is carried out on a single processor and only evaluation of model equations is parallelised. Here, the heterogeneous CPU+GPU configurations perform faster due to the highest maximum peak performance (70.88 GFLOPs versus 36.56 GFLOPs in NVidia GPU and 34.32 GFLOPs in the Intel CPU). The reason for somewhat lower overall speed-ups in single processor simulations (especially in Case 1) is that both cases are dominated by the time for solution of linear systems (62.63% of the total integration time in C1-SEQ and 49.56% in C2-SEQ run, Tables 7 and 8). The main contributor in this phase is a costly Jacobian-vector multiplication sub-phase required in every iteration of the linear solver: the SUNDIALS GMRES solver uses a difference quotient approximation of the Jacobian and requires 34.39% of the total integration time in C1-SEQ and 40.47% in C2-SEQ runs.

As expected, the best performance is achieved in C1-MPI and C2-MPI runs where the whole system is partitioned into eight sub-systems and simulated in parallel. Again, the overall simulation speed-ups are lower than the maximum theoretical since not all phases of the numerical solution can be parallelised, the performance of the solution of linear systems does not scale linearly with the size of the problem and there is an additional cost due to the load imbalance and the inter-process data exchange during the linear algebra operations and before every evaluation of model equations. While the data-exchange costs in the linear algebra operations are not measured, for the evaluation of model equations they amount to 3.81% of the total integration time in Case 1 and 2.39% in Case 2 (Tables 7 and 8). In addition, it has been found that partitioning of the overall ODE/DAE system into a number of sub-systems does not produce a significant effect on the performance of the solution of linear systems phase (although the linear systems are eight times smaller in every PE the performance does not scale linearly with the size of the problem). In total, the time for solution of linear systems is 63.06% of the total integration time in C1-MPI and 72.66% in C2-MPI run (Tables 7 and 8). Therefore, the process of selection of the most suitable preconditioner, tuning of its parameters and the Jacobian-vector multiplication sub-phase (as the most important factor) need to be further improved in the future work.

The speed-ups in the EvaluateEquations phase are 3.52, 3.05, 3.90 and 3.36 for C1-OMP, C1-CL, C1-HET and C1-MPI runs, respectively (Table 9) and 4.30, 4.30, 5.97 and 4.18 for C2-OMP, C2-CL, C2-HET and C2-MPI runs, respectively (Table 10). The speed-ups in the EvaluateJacobian phase are 3.02, 4.63, 4.17 and 4.36 for C1-OMP, C1-CL, C1-HET and C1-MPI runs, respectively (Table 9) and 3.74, 8.02, 5.14 and 3.72 for C2-OMP, C2-CL, C2-HET and C2-MPI runs, respectively (Table 10). The maximum theoretical speed-up is 8.00 for OpenMP and MPI runs, 8.52 for OpenCL runs and 16.52 for heterogeneous OpenCL. The computation load in a Compute Stack Machine kernel for evaluation of equations is directly proportional to the average number of Compute Stack items per equation ($N_{cs/equation}$) while the computation load for evaluation of the Jacobian is proportional to the average number of Compute Stack items for evaluation of a single row of the Jacobian matrix ($N_{cs/jacob\_row}$). It can be observed that higher speed-ups are achieved in Case 2 for both evaluation of equations and the Jacobian. The reason is that the model equations are more complex in Case 2, as it can be seen from Table 6: $N_{cs/equation}$ and $N_{cs/jacob\_row}$ are more than two times higher in Case 2. Thus, a much larger amount of computation per single call is required and the hardware is better utilised. Furthermore, the speed-ups in the Jacobian evaluation are always higher than speed-ups in the evaluation of equations (in both cases) since a larger number of evaluations are required and the hardware is again better utilised. Similar trends are also found in the benchmarks in Nikolić (2018, 2023). Consequently, a general rule is that a larger amount of computation per single kernel call always leads to a better performance (higher speed-up). This is in particular evident for evaluations on streaming processors (such as GPU). On the other hand, the performance of heterogeneous configurations are far from the maximum theoretical. Again, similar results have been obtained in Nikolić (2018) for different models. Although the additional time is required for memory transfers to/from the devices and for the management of OpenCL kernels, the most possible reason is that the current implementation of the multi-device OpenCL Compute Stack Evaluator is not optimised for best

performance and must be improved in the future work.

CONCLUSIONS

The main ideas, the key concepts, the components, the algorithms and the API of the OpenCS framework are presented in this work. OpenCS provides a universal platform for modelling of problems described by systems of differential and algebraic equations, parallel evaluation of model equations on diverse types of computing devices (including heterogeneous setups), parallel simulation on shared and distributed memory systems and model exchange.

The framework offers the numerous benefits. A single simulation software is used for numerical solution of systems of differential and algebraic equations on all platforms. Model equations, internally transformed into the postfix notation expression stacks and stored as an array of binary data, can be evaluated on virtually all computing devices with no additional processing and switching to a different computing device is controlled by an input parameter. The low-level model specification data structures, stored as files in binary format, are used as an input for parallel simulations on all platforms and provide a simple platform-independent binary interface for model exchange. The partitioning algorithm can accurately balance the computation and memory loads in all important phases of the numerical solution. Since a common model-specification is utilised on all platforms, OpenCS models can be used for benchmarks between different simulators, solvers, individual computing devices and high performance computing systems. For example, benchmarks between heterogeneous CPU/GPU and CPU/FPGA systems could be performed without re-implementation of the model for a completely different architecture.

The capabilities of the framework are illustrated using two large scale problems. The overall performance and the performance of four main and four sub-phases of the numerical solution have been analysed and compared to the maximum theoretical. For simulations carried out on a single processor the OpenMP API and the OpenCL framework have been utilised for parallelisation of model equations. The MPI interface has been used for simulation on message-passing multiprocessors. It has been observed that the overall simulation performance is in the following order: sequential < OpenMP < OpenCL (single device) < OpenCL (heterogeneous CPU+GPU) < MPI implementations. As it has been expected, the MPI simulations offer the best performance since the original ODE/DAE system is partitioned into the smaller ODE/DAE sub-systems and simulated in parallel. However, in order to approach the expected maximum theoretical speed-ups the Compute Stack Machine and Evaluator implementations must be further optimised.

The future work will focus on applications of the framework to large multi-scale and multi-physics problems, further improvement of the performance and reduction of the memory requirements, implementation of problem-specific graph partitioners, and Compute Stack Evaluator implementations for additional types of computing devices (such as FPGA).

REFERENCES

Altair. HyperWorks, 2018. URL https://altairhyperworks.com.

Christian Andersson, Claus Fuhrer, and Johan Akesson. Assimulo: A unified framework for ODE solvers. *Math. Comput. Simulat.*, 116(0):26 – 43, 2015. ISSN 0378-4754. doi: 10.1016/j.matcom.2015.04.007. URL http://dx.doi.org/10.1016/j.matcom.2015.04.007.

Ansys, Inc. ANSYS Fluent, 2018. URL http://www.ansys.com.

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015. URL http://www.mcs.anl.gov/petsc.

W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

Paul I. Barton and Constantinos C. Pantelides. Modeling of combined discrete/continuous processes. *AIChE J.*, 40(6): 966–979, 1994. ISSN 1547-5905. doi: 10.1002/aic.690400608. URL http://dx.doi.org/10.1002/aic.690400608.

COMSOL, Inc. COMSOL Multiphysics, 2018. URL http://www.comsol.com.

Dassault Systemes. Abaqus, 2018. URL http://www.simulia.com.

Peter Fritzson and Vadim Engelson. Modelica — a unified object-oriented language for system modeling and simulation. In Eric Jul, editor, *ECOOP'98 — Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer Berlin, Heidelberg, Germany, 1998. ISBN 978-3-540-64737-9. doi: 10.1007/BFb0054087. URL http://dx.doi.org/10.1007/BFb0054087.

John D. Hedengren, Reza Asgharzadeh Shishavan, Kody M. Powell, and Thomas F. Edgar. Nonlinear modeling, estimation and predictive control in APMonitor. *Comput. Chem. Eng.*, 70:133 – 148, 2014. ISSN 0098-1354. doi: 10.1016/j.compchemeng.2014.04.013. URL http://www.sciencedirect.com/science/article/pii/S0098135414001306. Manfred Morari Special Issue.

Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005. ISSN 0098-3500. doi: 10.1145/1089014.1089021. URL http://doi.acm.org/10.1145/1089014.1089021.

Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, September 2005. ISSN 0098-3500. doi: 10.1145/1089014.1089020. URL http://doi.acm.org/10.1145/1089014.1089020.

George Karypis and Vipin Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, 1995.

B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. libMesh: A c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. https://doi.org/10.1007/s00366-006-0049-3.

Dragan D. Nikolić. DAE Tools: Equation-based object-oriented modelling, simulation and optimisation software. *PeerJ Computer Science*, 2:e54, April 2016. ISSN 2376-5992. doi: 10.7717/peerj-cs.54. URL https://doi.org/10.7717/peerj-cs.54.

Dragan D. Nikolić. Parallelisation of equation-based simulation programs on heterogeneous computing systems. *PeerJ Computer Science*, 4:e160, August 2018. ISSN 2376-5992. doi: 10.7717/peerj-cs.160. URL https://doi.org/10.7717/peerj-cs.160.

Dragan D. Nikolić. Parallelisation of equation-based simulation programs on distributed memory systems. *SIAM J. Sci. Comput.*, February 2023. doi: submitted-for-review. URL https://daetools.sourceforge.io/docs/parallelisation-dsm-preprint.pdf.

P. C. Piela, T. G. Epperly, K. M. Westerberg, and A. W. Westerberg. ASCEND: an object-oriented computer environment for modeling and analysis: The modeling language. *Comput. Chem. Eng.*, 15(1):53–72, 1991. ISSN 0098-1354. doi: 10.1016/0098-1354(91)87006-U. URL http://www.sciencedirect.com/science/article/pii/009813549187006U.

M. Sala and M. Heroux. Robust algebraic preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories, 2005.

W. E. Schiesser and L. Lapidus. Academic Press, Inc, New York, United States, 1976. ISBN 0-12-4366406.

Radu Serban and Alan C. Hindmarsh. CVODES, the sensitivity-enabled ODE solver in SUNDIALS. Proceedings of the 5th International Conference on Multibody Systems, Nonlinear Dynamics and Control, Long Beach, CA, USA, 2005.

Radu Serban and Alan C. Hindmarsh. Example programs for CVODES v2.8.2. Technical Report UCRL-SM-208115, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2015. URL https://computation.llnl.gov/sites/default/files/public/cvs_examples.pdf.

Radu Serban and Alan C. Hindmarsh. Example programs for IDAS v1.3.0. Technical Report LLNL-TR-437091, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2016. URL https://computation.llnl.gov/sites/default/files/public/idas_examples.pdf.

Siemens. Multidisciplinary Design Exploration, 2018. URL https://mdx.plm.automation.siemens.com.

Steven H. Strogatz. CRC Press LLC, Boca Raton, Florida, USA, 1994. ISBN 0-73-8204536.

The MathWorks, Inc. MATLAB, 2018a. URL https://mathworks.com/products/matlab.

The MathWorks, Inc. Simulink, 2018b. URL https://mathworks.com/products/matlab.

The OpenFOAM Foundation. OpenFOAM, 2018. URL http://www.openfoam.org.

Waterloo Maple, Inc. Maple, 2015. URL http://www.maplesoft.com/products/maple/.

Michael R. Wittman. Testing of PVODE, a parallel ode solver. Technical Report UCRL-ID-125562, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 1996.

Wolfram Research, Inc. Mathematica, 2015. URL https://www.wolfram.com/mathematica.