Parallelisation of equation-based simulation programs on distributed memory systems

Dragan Nikolic Corresp. 1

1 DAE Tools Project

Corresponding Author: Dragan Nikolic
Email address: dnikolic@daetools.com

In this work, a methodology for parallel numerical solution of general systems of non-linear differential and algebraic equations (DAE) on distributed memory systems is proposed and implemented. The methodology consists of the following parts: (1) an algorithm for transformation of model equations into a data structure suitable for parallel evaluation on diverse types of computing devices, (2) data structures for model specification, (3) an algorithm for partitioning of general systems of equations, (4) an algorithm for inter-process data exchange, and (5) simulation software for integration of general DAE systems in time. Model equations are specified in a platform and programming language independent fashion as the postfix notation expression stacks (Compute Stacks) that can represent any type of equations of any size and be evaluated on virtually all computing devices. The model specification contains only the low-level model description with the minimum information required for integration in time: (a) the model structure (properties of model variables), (b) model equations, (c) the sparsity pattern (for evaluation of derivatives), and (d) partition data (for inter-process data exchange). This way, the model specification data structures are used as a simple platform-independent binary interface for model exchange. The partitioning algorithm accepts four different balancing constraints that can be used to precisely balance the computation and memory loads in critical phases of the numerical solution. The central point is the generic simulation software which runs on message passing multiprocessors and integrates arbitrary DAE systems in time. For computationally intensive tasks the software utilises multi-level parallelism techniques such as hybrid MPI/OpenMP and heterogeneous MPI/OpenCL: every processing element integrates a DAE sub-system in time on a full-blown host processor and can optionally perform evaluation of model equations using the OpenMP API on general purpose processors and the OpenCL framework on streaming processors. Simulation inputs are specified in a generic fashion as files in a binary (platform independent) format. The input files are generated by modelling software for each processing element and contain the serialised data structures with the model specification. To illustrate its capabilities and

limitations, the proposed methodology is applied to a medium-scale transient two-dimensional phase separation process. Six different phases of the numerical solution are analysed. Nine different strategies for load balancing are applied. Simulation results, an overall performance and performance of individual phases, an efficiency of the preconditioner, the quality of the load balancing prediction, and overheads due to the load imbalance are discussed in details.

Parallelisation of equation-based simulation programs on distributed memory systems

Dragan D. Nikolić

DAE Tools Project, Belgrade, Serbia, http://www.daetools.com

Corresponding author:
Dragan D. Nikolić

Email address: dnikolic@daetools.com

ABSTRACT

In this work, a methodology for parallel numerical solution of general systems of non-linear differential and algebraic equations (DAE) on distributed memory systems is proposed and implemented. The methodology consists of the following parts: (1) an algorithm for transformation of model equations into a data structure suitable for parallel evaluation on diverse types of computing devices, (2) data structures for model specification, (3) an algorithm for partitioning of general systems of equations, (4) an algorithm for inter-process data exchange, and (5) simulation software for integration of general DAE systems in time. Model equations are specified in a platform and programming language independent fashion as the postfix notation expression stacks (Compute Stacks) that can represent any type of equations of any size and be evaluated on virtually all computing devices. The model specification contains only the low-level model description with the minimum information required for integration in time: (a) the model structure (properties of model variables), (b) model equations, (c) the sparsity pattern (for evaluation of derivatives), and (d) partition data (for inter-process data exchange). This way, the model specification data structures are used as a simple platform-independent binary interface for model exchange. The partitioning algorithm accepts four different balancing constraints that can be used to precisely balance the computation and memory loads in critical phases of the numerical solution. The central point is the generic simulation software which runs on message passing multiprocessors and integrates arbitrary DAE systems in time. For computationally intensive tasks the software utilises multi-level parallelism techniques such as hybrid MPI/OpenMP and heterogeneous MPI/OpenCL: every processing element integrates a DAE sub-system in time on a full-blown host processor and can optionally perform evaluation of model equations using the OpenMP API on general purpose processors and the OpenCL framework on streaming processors. Simulation inputs are specified in a generic fashion as files in a binary (platform independent) format. The input files are generated by modelling software for each processing element and contain the serialised data structures with the model specification. To illustrate its capabilities and limitations, the proposed methodology is applied to a medium-scale transient two-dimensional phase separation process. Six different phases of the numerical solution are analysed. Nine different strategies for load balancing are applied. Simulation results, an overall performance and performance of individual phases, an efficiency of the preconditioner, the quality of the load balancing prediction, and overheads due to the load imbalance are discussed in details.

INTRODUCTION

Large scale systems of non-linear (partial-)differential and algebraic equations (DAE) are found in many engineering problems. Typically, such systems cannot be efficiently solved on shared memory systems due to the high memory and computation requirements and must be solved on distributed memory systems. Clusters of symmetric multiprocessors (SMP) are the most commonly used architecture for large scale simulations and the Message Passing Interface (MPI) is de facto a standard for distributed memory systems. Simulation on distributed memory systems is performed by partitioning the DAE system into a specified number of subsystems. The simulation is then run in parallel on a specified number of processing elements (PE) using the MPI interface, where each PE integrates one DAE subsystem in time and performs an inter-process communication to exchange the data between nodes.

In general, the parallel 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) libraries for Finite Element Analysis (FEA) and Computational Fluid Dynamics (CFD) such as deal.II (Bangerth et al., 2007), libMesh (Kirk et al., 2006), and OpenFOAM (The OpenFOAM Foundation, 2018),
(3) 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).

In most cases, parallel simulations developed in C/C++/FORTRAN using one of the suites for scientific applications are optimised for a particular computing platform and often produce the fastest computation. Model equations are typically specified as user-supplied functions for evaluation of residuals and derivatives. However, the process is error prone, it is difficult to specify expressions for analytical derivatives, manually partition the system and implement an inter-process communication.

FEA/CFD libraries offer an Application Programming Interface (API) to assist in the process of pre-processing, discretisation of PDE, numerical solution and post-processing. However, many low-level tasks still must be done manually. On the other hand, CAE software offer a great degree of generality: all required tasks are performed (mostly) automatically through the graphical user interface. In both approaches, model equations are represented by the data structures resulting from the discretisation process. On unstructured grids, the discretisation is performed using the Finite Element (FE) or Finite Volume (FV) methods and produces the mass and stiffness matrices and load vectors. On structured grids, the discretisation is performed using the Finite Difference (FD) method and yields the stencil data (nodes arrangement and their coefficients). Parallel evaluation of model equations is carried out using matrix-vector/matrix-matrix operations or stencil codes available for all platforms. However, the general non-linear DAE systems cannot be represented in this way.

The focus of this work is a methodology for parallel numerical solution of general systems of non-linear differential and algebraic equations on distributed memory systems. In contrast to problems that can be described by a single system of finite element/finite volume/finite difference equations the focus in this work is on DAE systems that might include mixed/multiple coupled FE/FV/FD equations with additional ordinary differential and algebraic equations. Such mixed

systems of equations are often found in multi-scale models or models of chemical plants and typically cannot be represented using the methods above. For instance, a detailed model of a chemical process plant might include multiple systems with distributed parameters for individual unit operations which, depending on the unit type, utilise different discretisation methods. In addition, auxiliary differential and algebraic equations are required for the connectivity between units and evaluation of the plant performance.

The methodology consists of several parts: (1) an algorithm for transformation of model equations into a data structure suitable for parallel evaluation on different computing platforms (the Compute Stack approach, Nikolić, 2018), (2) data structures for model specification that contain all information required for numerical solution such as: the model structure, the model equations, the sparsity pattern (for evaluation of derivatives), and partition data (for inter-process data exchange), (3) an algorithm for partitioning of general systems of equations, (4) an algorithm for inter-process data exchange, and (5) a simulation software for integration of general DAE systems in time. The idea is to separate (simulator-dependent) generation of a system of equations (i.e. meshing, discretisation and system assembly) and partitioning of the system, typically performed only once, from its parallel (in general, simulator-independent) numerical solution which is computationally the most intensive task. While generation of the system of equations can be performed in different ways depending on the type of the problem and the method applied by a simulator, the numerical solution procedure requires only a low-level model description. For instance, the model description can be built using a modelling language or a CAE software utilising various discretisation methods. However, information required by DAE solvers are essentially identical: the data about the number of variables, their names, types, absolute tolerances and initial conditions, the functions for evaluation of residuals and derivatives, and the function for exchange of adjacent variables between processing elements. Therefore, in this work, the model specification data structures contain only the low-level information directly required by solvers. A generic simulation software has been developed to utilise such a model specification: when the software is run in parallel on message passing multiprocessors, every processing element integrates one part (sub-system) of the overall DAE system in time and performs an inter-process communication to exchange the data between processing elements. Simulation inputs are specified in a generic fashion as files in a (platform independent) binary format. The input files are generated by a modelling software (in this work DAE Tools) and contain the serialised model specification data structures and solver options. In addition, streaming processors/accelerators available on individual processing elements such as General Purpose Graphics Processing Units (GPGPU), Field Programmable Gate Arrays (FPGA) and manycore systems (Intel Xeon Phi) can be utilised for evaluation of model equations (Nikolić, 2018). An overview of the solution procedure is given in Fig. 1. The input data files are generated for every processing element, stored in a local or a Network File System and the parallel simulation started using the MPI interface. Sequential simulations can be performed by generating input files for a single processing element.

The proposed methodology offers the numerous benefits. A single software is used for numerical solution of any system of non-linear differential and algebraic equations. An implementation in standard C99 and C++11 allows compilation for all high-performance computing platforms. Model equations are specified as the postfix notation expression stacks and can be evaluated on virtually all computing devices including heterogeneous systems that contain streaming processors/accelerators (Nikolić, 2018). The partitioning algorithm applies multiple balancing constraints to simultaneously balance the memory and computation loads in the critical

phases of the numerical solution. The format of the inter-process communication data is general enough to allow the data exchange to be performed by any communication interface (not only MPI). Thus, the simulations can be run on various types of distributed systems. Simulation inputs are specified using the data files in a platform-independent binary format. Therefore, the input files are used as a simple binary interface for model exchange and, in general, the required low-level information can be provided by any modelling software. 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 nor a binary interface (C API) implemented in shared libraries as in Simulink (The MathWorks, Inc., 2018) and Functional Mock-up Interface (https://www.fmi-standard.org). For instance, in this approach, the model equations are specified as an array of binary data for direct evaluation by simulators on all platforms/operating systems with no additional processing nor compilation steps.

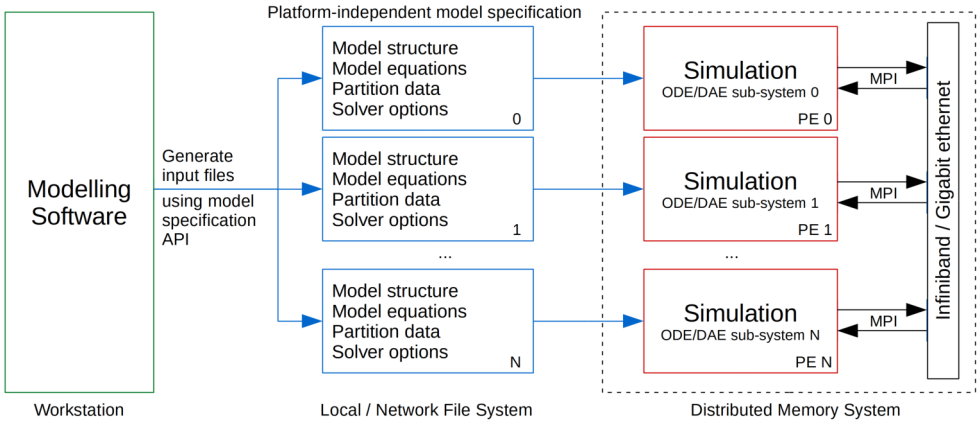

Figure 1. Generic methodology for parallel simulation on distributed memory systems

The limitation of the methodology is that it is difficult to apply to problems that use adaptive grids since the total number of variables/equations change in runtime. In these cases, generation of model equations and partitioning must be carried out after every change in the grid. In addition, the partitioning algorithm treats systems of equations as black-boxes and is therefore unable to exploit their specific structure which might result in inefficient preconditioners.

The article is organised in the following way. First, the required data structures, algorithms and implementations are presented. Then, the proposed methodology is applied to a medium scale phase separation model. The simulation results, an overall performance and performance of individual phases, an efficiency of the preconditioner, the quality of the load balancing prediction, and overheads due to the load imbalance are analysed and discussed. Finally, a summary of the most important capabilities of the methodology and directions for future work are given in the last section.

IMPLEMENTATION

The methodology is implemented in DAE Tools software (Nikolić, 2016) and based on the previously developed methodology for parallel evaluation of general systems of differential and algebraic equations on shared memory systems (Nikolić, 2018). An overview of the solution procedure on shared memory systems is given in Fig. 2. The solution process consists of: (1) numerical integration in time (requires evaluation of equations residuals), (2) linear algebra operations, (3) solution of systems of linear equations (requires evaluation of derivatives and computation of the preconditioner), and (4) (optionally) integration of sensitivity equations (requires evaluation of sensitivity residuals). The parallel solution on distributed memory systems requires the same tasks, but here applied to integration of only one part of the overall system (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. 3. The additional functionality that is required includes: (a) an inter-process communication routine for exchange of adjacent variables (variables that belong to other processing elements), and (b) linear algebra routines for distributed memory systems (already available from the SUNDIALS suite). Both routines are implemented using the MPI C interface. Versions for solution of both ODE and DAE systems are developed. However, the focus in this work is on DAE systems as a more general form of differential equations which are more difficult to solve than ODE systems.

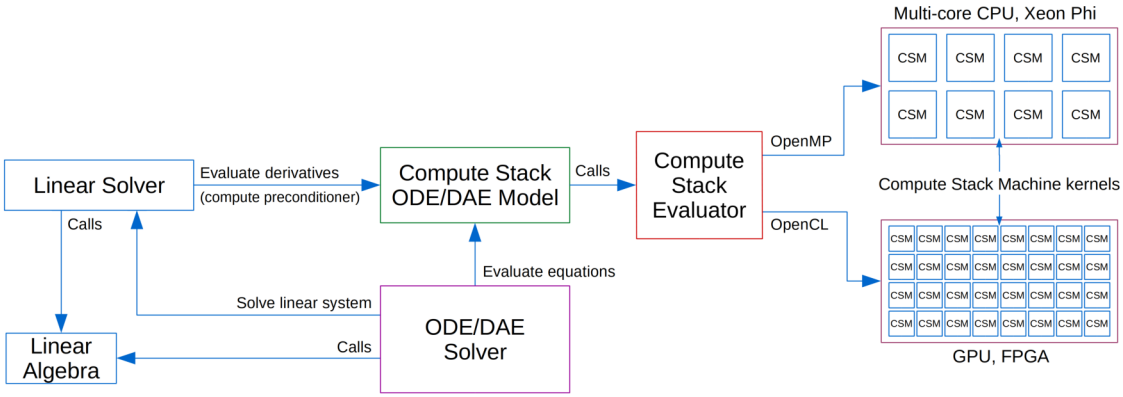

Figure 2. Simulation on shared memory systems (the main building block)

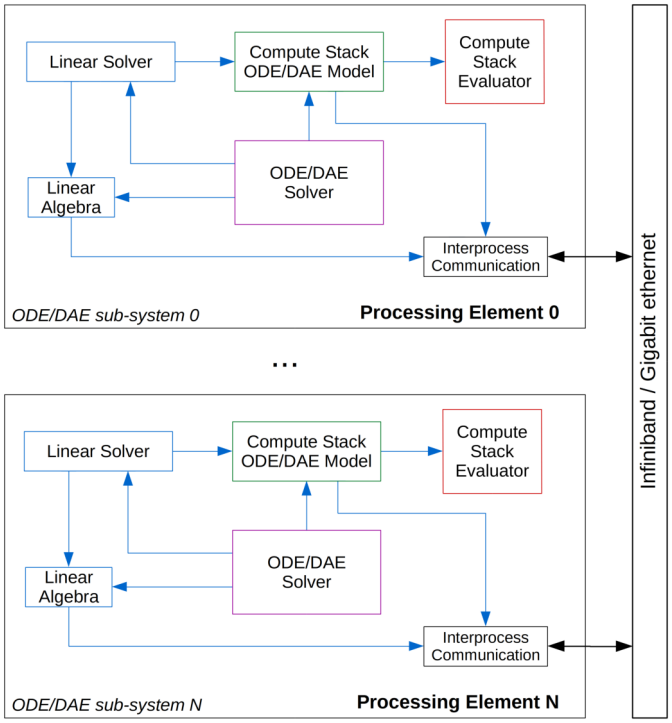

Figure 3. Simulation on distributed memory systems

Key concepts and data structures

The methodology is based on several concepts, each providing a distinct functionality:

Compute Stack The Reverse Polish (postfix) notation expression stack used as a platform and programming language independent method to describe, store in computer memory and evaluate equations of any type and any size (Nikolić, 2018). Equations can be linear or non-linear, algebraic or differential. Each mathematical operation and its operands are described by a specially designed *csComputeStackItem_t* data structure (given in the source code listing 1), and every equation is transformed into an array of these structures (a Compute Stack). Typically, Compute Stacks are automatically generated from simulator-specific data structures. For instance, in DAE Tools equations are transformed into the Evaluation Tree data structure using the operator overloading technique (Nikolić, 2018). The Compute Stacks are generated by traversing the Evaluation Tree nodes.

Compute Stack Machine A stack machine used to evaluate a single equation (that is a single Compute Stack) using Last In First Out (LIFO) queues (function *evaluateComputeStack* in the source code listing 1).

Compute Stack Evaluator An interface for parallel evaluation of systems of equations (*csComputeStackEvaluator_t* class in the source code listing 2). Two implementations are available (Nikolić, 2018): (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.

Compute Stack Model Data structure that holds the model specification - all information required for the numerical solution, either sequentially or in parallel (*csModel_t* data structure

in the source code listing 3). 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 structure of a model with information about the variable names, types, absolute tolerances and initial conditions: *csModelStructure_t* structure, (b) model equations: *csModelEquations_t* structure, (c) the sparsity pattern of the ODE/DAE (sub-) system (required for evaluation of derivatives): *csSparsityPattern_t* structure, (d) partition data (used for inter-process communication): *csPartitionData_t* structure, and (e) the Compute Stack evaluator instance: *csComputeStackEvaluator_t* object.

Compute Stack Differential Equations Model A common interface that provides an API required by ODE/DAE solvers for integration of systems of differential equations in time (*csDifferentialEquationModel_t* class in the source code listing 4). It is derived from *csModel_t* class and provides functions for loading the model from input files, retrieving the sparsity pattern of the ODE/DAE system, setting the variable values/derivatives, exchanging the adjacent variables among the processing elements using the MPI interface, and evaluating equations and derivatives.

Compute Stack Simulator Software for sequential and parallel simulation of general ODE/-DAE systems in time (*csSimulator_ODE* and *csSimulator_DAE*, respectively).

Listing 1. Compute Stack Machine data structures and functions (C99)

```c
/* real_t is defined as a single or a double precision floating point type. */
#define real_t double

/* OP codes */
typedef enum
{
    eOP_Unknown = 0,
    eOP_Constant,
    eOP_Time,
    eOP_InverseTimeStep,
    eOP_Variable,
    eOP_DegreeOfFreedom,
    eOP_TimeDerivative,
    eOP_Unary,
    eOP_Binary
} csOpCode;

/* Auxiliary data structures */
typedef struct adouble_
{
    real_t m_dValue;
    real_t m_dDeriv;
} adouble_t;

typedef struct csEvaluationContext_
{
    real_t   currentTime;
    real_t   inverseTimeStep;
    uint32_t equationEvaluationMode;
    uint32_t sensitivityParameterIndex;
    uint32_t jacobianIndex;
    uint32_t numberOfVariables;
    uint32_t startEquationIndex;
    uint32_t numberOfEquations;
    uint32_t startJacobianIndex;
    uint32_t numberOfIncidenceMatrixItems;
    uint32_t numberOfDOFs;
    uint32_t numberOfComputeStackItems;
    uint32_t valuesStackSize;
    uint32_t lvaluesStackSize;
    uint32_t rvaluesStackSize;
} csEvaluationContext_t;

/* Compute Stack Item data structure. */
typedef struct csComputeStackItem_
{
    uint8_t  opCode;         /* Operation to perform */
    uint8_t  function;       /* Unary or binary function code */
    uint8_t  resultLocation; /* lvalue, value or rvalue LIFO stacks */
    uint32_t size;           /* Size of the compute stack array */
    union data_
    {
        real_t value;        /* For constants */
        struct dof_indexes_  /* For degrees of freedom */
        {
            uint32_t overallIndex;
            uint32_t dofIndex;
        }dof_indexes;
        struct indexes_      /* For variables (algebraic and differential) */
        {
            uint32_t overallIndex;
            uint32_t blockIndex;
        }indexes;
    }data;
} csComputeStackItem_t;

/* Compute Stack Machine function */
adouble_t evaluateComputeStack(CS_KERNEL_FLAG const csComputeStackItem_t*  computeStack,
                               csEvaluationContext_t                       EC,
                               CS_KERNEL_FLAG const real_t*                dofs,
                               CS_KERNEL_FLAG const real_t*                values,
                               CS_KERNEL_FLAG const real_t*                timeDerivatives,
                               CS_KERNEL_FLAG const real_t*                svalues,
                               CS_KERNEL_FLAG const real_t*                sdvalues);
```

Listing 2. Compute Stack Evaluator interface

```
class csComputeStackEvaluator_t
{
public:
    virtual ~csComputeStackEvaluator_t(){}

    virtual void Initialize(bool                     calculateSensitivities,
                            size_t                   numberOfVariables,
                            size_t                   numberOfEquationsToProcess,
                            size_t                   numberOfDOFs,
                            size_t                   numberOfComputeStackItems,
                            size_t                   numberOfIncidenceMatrixItems,
                            size_t                   numberOfIncidenceMatrixItemsToProcess,
                            csComputeStackItem_t*    computeStacks,
                            uint32_t*                activeEquationSetIndexes,
                            csIncidenceMatrixItem_t* incidenceMatrixItems) = 0;

    virtual void FreeResources() = 0;

    virtual void EvaluateEquations(csEvaluationContext_t EC,
                                   real_t*               dofs,
                                   real_t*               values,
                                   real_t*               timeDerivatives,
                                   real_t*               equations) = 0;

    virtual void EvaluateDerivatives(csEvaluationContext_t EC,
                                     real_t*               dofs,
                                     real_t*               values,
                                     real_t*               timeDerivatives,
                                     real_t*               derivatives) = 0;

    virtual void EvaluateSensitivityDerivatives(csEvaluationContext_t EC,
                                                real_t*               dofs,
                                                real_t*               values,
                                                real_t*               timeDerivatives,
                                                real_t*               svalues,
                                                real_t*               sdvalues,
                                                real_t*               sderivatives) = 0;
};
```

Listing 3. Compute Stack Model related data structures

```
/* Model structure. */
struct csModelStructure_t
{
    uint32_t                 Nequations;
    uint32_t                 Nequations_total;
    uint32_t                 Ndofs;
    bool                     isODESystem;
    std::vector<real_t>      dofValues;
    std::vector<real_t>      variableValues;
    std::vector<real_t>      variableDerivatives;
    std::vector<std::string> variableNames;
    std::vector<int32_t>     variableTypes;
    std::vector<real_t>      absoluteTolerances;
};

/* Model equations. */
struct csModelEquations_t
{
    std::vector<csComputeStackItem_t>   computeStacks;
    std::vector<uint32_t>               activeEquationSetIndexes;
};

/* The sparsity pattern of the DAE system (CRS format). */
typedef struct csIncidenceMatrixItem_
{
    uint32_t equationIndex;
    uint32_t overallIndex;
    uint32_t blockIndex;
} csIncidenceMatrixItem_t;

struct csSparsityPattern_t
{
    uint32_t                              Nnz;
    uint32_t                              Nequations;
    std::vector<uint32_t>                 rowIndexes;
    std::vector<csIncidenceMatrixItem_t> incidenceMatrixItems;
};

/* Partition data. */
typedef std::map< int32_t, std::vector<int32_t> > csPartitionIndexMap;
struct csPartitionData_t
{
    std::vector<int32_t>      foreignIndexes;
    std::map<int32_t,int32_t> biToBiLocal;
    csPartitionIndexMap       sendToIndexes;
    csPartitionIndexMap       receiveFromIndexes;
};

/* Compute Stack Model. */
class csModel_t
{
public:
    csModel_t();
    virtual ~csModel_t();

    /* Model I/O functions. */
    void LoadModel(int rank, const std::string& inputDirectory);
    void SaveModel(const std::string& outputDirectory);

    csModelStructure_t         structure;
    csModelEquations_t         equations;
    csSparsityPattern_t        sparsityPattern;
    csPartitionData_t          partitionData;
    csComputeStackEvaluator_t* csEvaluator;

    int pe_rank; /* MPI rank of the processing element. */
};
```

Listing 4. Compute Stack Differential Equations Model interface

```
/* A generic interface to a sparse matrix storage. */
class csMatrixAccess_t
{
public:
    virtual ~csMatrixAccess_t(){}

    virtual void SetItem(size_t row, size_t col, real_t value) = 0;
};

/* Common ODE/DAE Model class interface. */
class csDifferentialEquationModel_t : public csModel_t
{
public:
    virtual ~csDifferentialEquationModel_t(){}

    virtual void Load(int rank,
                      const std::string& inputDirectory,
                      csComputeStackEvaluator_t* csEvaluator) = 0;

    virtual void Free() = 0;

    virtual void GetSparsityPattern(int& N,
                                    int& NNZ,
                                    std::vector<int>& IA,
                                    std::vector<int>& JA) = 0;

    virtual void EvaluateEquations(real_t time,
                                   real_t* equations) = 0;

    virtual void EvaluateJacobian(real_t time,
                                  real_t inverseTimeStep,
                                  csMatrixAccess_t* ma) = 0;

    virtual void SetAndSynchroniseData(real_t  time,
                                       real_t* values,
                                       real_t* time_derivatives) = 0;
};
```

Algorithm for partitioning of general systems of equations

The computationally most intensive phases of the numerical solution are: (1) evaluation of equations residuals, (2) solution of systems of linear equations, and (3) evaluation of derivatives (the Jacobian matrix) for computation of a preconditioner. Combined, they typically amount to more than 95% of the total integration time (Nikolić, 2018). 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 are 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.

The traditional approach to this problem is to partition a DAE system so that the number of equations assigned to each partition is the same, and the number of adjacent variables assigned to different processing elements is minimised. In theory, the first condition balances the computation among the PEs while the second one minimises the volume of inter-process communication data. However, the traditional problem formulation is limited in that it can only balance a single quantity and works well only if the DAE system is composed of the same type of equations or the blocks of equations of the same type. The general DAE systems may include very diverse types of equations that require different number of variables. The traditional approach in this case would produce unbalanced partitions with different workloads. Since it is critical that every processor have an equal amount of work from each phase of the computation, the multiple quantities must be load balanced simultaneously. This is because synchronisation is often performed implicitly or explicitly after every computational phase, and each phase must be individually load balanced.

Graph partitioning is a common way to satisfy the necessary conditions. A graph of the DAE system is constructed by associating a vertex with each equation and adding an edge between two vertices i and j if there is a variable with the index j in the equation i. In this work, partition of an unstructured graph into a user-specified number k of parts is performed using the multilevel k-way partitioning paradigm implemented in METIS (Karypis and Kumar, 1995). The objective of the traditional graph partitioning problem is to compute a k-way partitioning such that the number of edges that straddle different partitions is minimised. This objective is commonly referred to as the edge-cut. In addition, METIS includes partitioning routines that can be used to partition a graph in the presence of multiple balancing constraints. Each vertex is assigned a vector of weights and the objective of the partitioning routines is to minimise the edge-cut subject to the constraints that each one of the weights is equally distributed among the partitions (Karypis and Kumar, 1995). For instance, if the first weight corresponds to the amount of computation and the second weight corresponds to the amount of storage required, then the partitioning algorithm will balance both the computation performed in each partition as well as the amount of memory that it requires.

The partitioning algorithm in this work applies a static load balancing method since the workloads are apriori known. In the Compute Stack approach the workloads can be accurately and precisely estimated by taking into consideration several properties of equations and partitions. The equation properties used in this work are (Table 1): number of Compute Stack items, number of FLOPS required for evaluation, number of non-zero items in one row of the incidence matrix (equal to the number of variables that appear in the equation), and number of FLOPS required for evaluation of a single row of the Jacobian matrix. The partition properties are (Table 2): number of equations, number of adjacent variables, number of items in the Compute Stack array in all equations, number of non-zero items in the partition's incidence matrix, number of FLOPS required for evaluation of equations, and number of FLOPS required for evaluation of derivatives (the Jacobian matrix). The memory and computation workloads in individual phases can be estimated using the partition properties as described in Table 3. For instance, the memory load for evaluation of the Jacobian matrix is proportional to the number of non-zero items in the incidence matrix, while the computation load is proportional to the number of FLOPS required for its evaluation. This way, the memory and computation loads of three critical phases of the numerical solution can be simultaneously balanced.

Table 1. Equation properties related to memory and computation loads

Property	Description
$N_{cs}[i]$	Number of Compute Stack items
$N_{flops}[i]$	Number of FLOPS for evaluation of the equation
$N_{nz}[i]$	Number of non-zero items in the equation
$N_{flops\_j}[i] = N_{nz}[i] \cdot N_{flops}[i]$	Number of FLOPS for evaluation of derivatives

Table 2. Partition properties related to memory and computation loads

Property	Description
N_{eq}	Number of equations/variables
N_{adj}	Number of adjacent variables
$N_{cs} = \sum_{i=0}^{N_{eq}} N_{cs}[i]$	Number of Compute Stack items
$N_{flops} = \sum_{i=0}^{N_{eq}} N_{flops}[i]$	Number of FLOPS for evaluation of equations
$N_{nz} = \sum_{i=0}^{N_{eq}} N_{nz}[i]$	Number of non-zero items in the incidence matrix
$N_{flops\_j} = \sum_{i=0}^{N_{eq}} N_{flops\_j}[i]$	Number of FLOPS for evaluation of derivatives

Table 3. Memory and computation loads in individual phases of the numerical solution

Phase	Memory load	Computation load
1. Evaluation of equations residuals	$\propto N_{cs}$	$\propto N_{flops}$
2. Solution of a linear system	$\propto N_{nz}$	$\propto (N_{eq}, N_{nz})$
3. Evaluation of a Jacobian/preconditioner	$\propto N_{nz}$	$\propto N_{flops\_j}$

The algorithm for partitioning of a general DAE system using the multiple balancing constraints is presented in Algorithm 1. Currently, it is implemented in Python using PyMetis Python wrappers (Kloeckner, 2018) for METIS software. In the future versions it will be re-implemented in C++ for performance reasons. First, properties of all equations are collected and adjacency data and vertex weights created. Then, the partitioning is performed for the specified number of processing elements (Npe) minimising edge-cut and balancing the loads using the vertex weights. Finally, the following data are generated for every partition: (a) a set with all variable indexes (*AllIndexes*), (b) a set with variable indexes owned by this partition (*OwnedIndexes*), (c) a set with indexes owned by other partitions (*AdjacentIndexes*), (d) two dictionaries with indexes for data exchange between partitions (*ReceiveFromIndexes* and *SendToIndexes*), and (e) a dictionary that maps global variable indexes to local partition indexes (*bi_to_bi_local*). These data are used to populate *csPartitionData_t* and *csSparsityPattern_t* data structures.

To illustrate the necessity for additional constraints in partitioning of general systems of equations a small and a very simple system of equations is considered (Eq. 1). The system is split into three partitions with one equation each. The partitioning results are given in Table 4.

$$\begin{aligned} x_1 + x_2 + x_3 &= 0 \\ x_1 + 2 \cdot x_2 &= 0 \\ \tfrac{x_3}{2.0} - 1.0 &= 0 \end{aligned} \tag{1}$$

Table 4. Partitioning results for a simple DAE system in Eq. 1

Partition	N_{eq}	N_{adj}	N_{cs}	FLOPS for residuals	N_{nz}	FLOPS for Jacobian
0	1	2	5	2 additions	3	3 x (2 additions)
1	1	1	5	addition + multiplication	2	2 x (addition + multiplication)
2	1	0	5	division + subtraction	1	1 x (division + subtraction)

The number of equations is uniformly distributed, every equation in the system requires exactly two mathematical operations, and the number of Compute Stack items is identical in all

equations. However, the computation load is not well balanced. Equations include mathematical operations that take different number of FLOPS for evaluation. For example, the multiplication and division operations take more time to finish than addition and subtraction. Hence, the time for evaluation of equations (and consequently the computation loads) are different.

In this work, this issue is resolved using two dictionaries with a number of FLOPS required for unary and binary mathematical operations: *unaryOperationsFlops* and *binaryOperationsFlops*, respectively. Number of FLOPS can be specified for unary (+, -) and binary (+, -, *, / and **) mathematical operators, unary (sqrt, log, log10, exp, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, erf, floor, ceil, and abs) and binary (pow, min, max, atan2) mathematical functions. As a result, the total number of FLOPS can be precisely estimated for every equation. If a mathematical operation is not in the dictionaries, the algorithm assumes that it requires a single FLOP. In addition, this approach allows specification of a separate pair of dictionaries for every computational platform. 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 a DAE system and generation of input data files is performed using the DAE Tools MPI code generator. A typical procedure is presented in the source code listing 5. The MPI code generator produces input files, images with the sparsity pattern, a partition graph, and partitioning statistics in comma separated values and LaTeX file formats.

Listing 5. Code generation procedure (Python language)

```
# Import the MPI code_generator.
from daetools.code_generators.mpi import daeCodeGenerator_MPI

# Instantiate the MPI code generator object.
cg = daeCodeGenerator_MPI()

# Optional: specify the number of FLOPS for mathematical
#           operations that require more than one FLOP.
# All other operations are assumed to require a single FLOP.
unaryFlops  = {eSqrt  : 6, eExp : 9}
binaryFlops = {eMulti : 2, eDivide : 4}

# Generate input files in the specified directory for 10 processing elements.
# The partitioning objective is to minimise edge_cut (the default),
# and balance the FLOPS for evaluation of residuals and the Jacobian.
# The argument "simulation" is an initialised DAE Tools simulation object.
cg.generateSimulation(simulation,
                      directory             = '...',
                      Npe                   = 10,
                      balancingConstraints  = ['Nflops', 'Nflops_j'],
                      unaryOperationsFlops  = unaryFlops,
                      binaryOperationsFlops = binaryFlops)
```

Algorithm 1 Partitioning of a general system of equations using multiple balancing constraints

Inputs: Information about equations in the DAE system.
Outputs: *csPartitionData_t* and *csSparsityPattern_t* data structures.
Step 1. Create the adjacency graph (variable indexes in all equations) and weights
  for $ei := 0$ **to** $N_{equations}$ **do**
    Get $variableIndexes, N_{cs}, N_{flops}, N_{nz}, N_{flops\_j}$ for the current equation
    $EquationsIndexes[ei] := variableIndexes$
    $VertexWeights[ei] := (N_{cs}, N_{flops}, N_{nz}, N_{flops\_j})$
  end for
Step 2. Perform the partitioning (minimise edge-cut and balance the load using the vertex weights)
  $n_{cuts}, partitions := pymetis.part\_graph(Npe, EquationsIndexes, VertexWeights)$
Step 3. For each partition: collect variable indexes owned by this partition (*OwnedIndexes*)
  for $ei := 0$ **to** $N_{equations}$ **do**
    add *ei* index to $OwnedIndexes[partitions[ei]]$ set
  end for
Step 4. For each partition: generate a set with all indexes (*AllIndexes*)
  for $pe := 0$ **to** N_{pe} **do**
    for $ei := 0$ **to** $OwnedIndexes[pe].size$ **do**
      add $EquationsIndexes[pe]$ list to $AllIndexes[ei]$ set
    end for
  end for
Step 5. For each partition: generate a set with indexes owned by other partitions (*AdjacentIndexes*)
  for $pe := 0$ **to** N_{pe} **do**
    $difference := AllIndexes[pe] \setminus OwnedIndexes[pe]$
    add $difference$ subset to $AdjacentIndexes[pe]$ set
  end for
Step 6. For each partition: generate maps with indexes for data exchange between partitions
  for $pe_i := 0$ **to** N_{pe} **do**
    for $pe_j := 0$ **to** N_{pe} **do**
      $intersection := AdjacentIndexes[pe_i] \cap OwnedIndexes[pe_j]$
      if *intersection is not empty* **then**
        insert $\{pe_j, intersection\}$ pair into $ReceiveFromIndexes[pe_i]$ map
        insert $\{pe_i, intersection\}$ pair into $SendToIndexes[pe_j]$ map
      end if
    end for
  end for
Step 7. For each partition: generate a map of global to local partition indexes (*bi_to_bi_local*)
  for $pe := 0$ **to** N_{pe} **do**
    $N_{owned} := OwnedIndexes[pe].size$
    for $i := 0$ **to** $OwnedIndexes[pe].size$ **do**
      $bi := OwnedIndexes[pe][i]$
      insert $\{bi, i\}$ pair into $bi\_to\_bi\_local[pe]$ map
    end for
    for $i := 0$ **to** $AdjacentIndexes[pe].size$ **do**
      $bi := AdjacentIndexes[pe][i]$
      insert $\{bi, N_{owned} + i\}$ pair into $bi\_to\_bi\_local[pe]$ map
    end for
  end for

The generated list of input files for simulations (one set for every processing element) is given in Table 5. *PE* in file names is an integer identifying the processing element equal to the value returned from *MPI_Comm_rank* function. 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, the simulations can be performed for different time horizons and different solver and preconditioner options. 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). The *"Simulation"* section includes the data such as the simulation start and end time, data reporting interval, relative tolerance and the output directory. The *"Model"* section includes the data such as options for evaluation of model equations (the Compute Stack Evaluator). 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 contents of the *simulation_options.json* file are given in the source code listing 6.

Table 5. The description of input data files for distributed 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

Listing 6. Typical simulation input parameters

```
{
    "Simulation": {
        "StartTime":         0.0,
        "TimeHorizon":       500.0,
        "ReportingInterval": 5.0,
        "OutputDirectory":   "results"
    },
    "Model" : {
        "ComputeStackEvaluator" : {
            "Library" : "OpenCL",
            "Name": "Single-device",
            "Parameters": {
                "platformID" :          0,
                "deviceID" :            0,
                "buildProgramOptions" : ""
            }
        }
    },
    "Solver" : {
        "Library" : "Sundials",
        "Name": "IDAS",
        "PrintInfo": false,
        "Parameters": {
            "RelativeTolerance":  0.00001,
            "IntegrationMode":    "Normal",
            "MaxOrd":             5,
            "MaxNumSteps":        500,
            "InitStep":           0.0,
            "MaxStep":            0.0,
            "MaxErrTestFails":    10,
            "MaxNonlinIters":     4,
            "MaxConvFails":       10,
            "NonlinConvCoef":     0.33,
            "SuppressAlg":        false,
            "NoInactiveRootWarn": false,
            "NonlinConvCoefIC":   0.0033,
            "MaxNumStepsIC":      5,
            "MaxNumJacsIC":       4,
            "MaxNumItersIC":      10,
            "LineSearchOffIC":    false
        }
    },
    "LinearSolver" : {
        "Library" : "Sundials",
        "Name": "gmres",
        "PrintInfo": false,
        "Parameters": {
            "kspace":            30,
            "EpsLin":            0.05,
            "JacTimesVecFn":     "DifferenceQuotient",
            "DQIncrementFactor": 1.0,
            "MaxRestarts":       5,
            "GSType":            "MODIFIED_GS"
        },
        "Preconditioner" : {
            "Library" : "Ifpack",
            "Name": "ILU",
            "PrintInfo": false,
            "Parameters": {
                "fact: level-of-fill":      3,
                "fact: relax value":        0.0,
                "fact: absolute threshold": 1e-5,
                "fact: relative threshold": 1.0
            }
        }
    }
}
```

Algorithm for inter-process data exchange

The algorithm for data exchange among processing elements is simple and only the point-to-point communication routines are required. First, the current values and derivatives of state variables are copied from the solver arrays. Second, for each processing element in the map *csPartitionData_t::sendToIndexes* the values and derivatives are asynchronously sent to other processing elements (the resulting *MPI_Request* objects are added to the *requests* array). Next, for each processing element in the map *csPartitionData_t::receiveFromIndexes* the values and derivatives are asynchronously received from other processing elements (the resulting *MPI_Request* objects are added to the *requests* array). Then, the algorithm waits for all MPI send/receive operations to finish. Finally, the received values and derivatives of adjacent variables are copied to the local arrays.

Algorithm 2 Inter-process data exchange (using the MPI C interface)

Inputs: *csPartitionData_t* data structure, variable values and derivatives.
Outputs: Values and derivatives of adjacent variables.
Step 1. Copy the values and derivatives of state variables from the solver to local arrays
Step 2. Asynchronously send values and derivatives to other PE
  for $pe_{send\_to} := 0$ **to** $N_{pe\_send\_to}$ **do**
    $request_1 := MPI\_Isend(values, ..., pe_{send\_to}, ...)$
    $request_2 := MPI\_Isend(derivatives, ..., pe_{send\_to}, ...)$
    Add $request_1$ and $request_2$ to the *requests* array of *MPI_Request* objects
  end for
Step 3. Asynchronously receive the values/derivatives from other PE
  for $pe_{receive\_from} := 0$ **to** $N_{pe\_receive\_from}$ **do**
    $request_1 := MPI\_Irecv(values, ..., pe_{receive\_from}, ...)$
    $request_2 := MPI\_Irecv(derivatives, ..., pe_{receive\_from}, ...)$
    Add $request_1$ and $request_2$ to the *requests* array of *MPI_Request* objects
  end for
Step 4. Wait for all operations to finish
  $MPI\_Waitall(requests)$;
Step 5. Copy the received values and derivatives of adjacent variables to local arrays

Generic simulation software

The central point in the proposed methodology is a generic simulation software. Versions for both ODE and DAE systems are developed: csSimulator_ODE and csSimulator_DAE, respectively. The focus in this work is on DAE systems as a more general form of systems of differential equations. The software can be executed sequentially on a single processor or in parallel on message passing multiprocessors, where every processing element integrates one part (subsystem) of the overall ODE/DAE system in time and performs an inter-process communication to exchange the adjacent variables among the processing elements.

csSimulator_DAE is a part of the Open Compute Stack (OpenCS) framework - an independent component of the DAE Tools equation-based modelling, simulation and optimisation software (Nikolić, 2016). DAE Tools MPI code generator (*daeCodeGenerator_MPI*) is used to generate the input files from DAE Tools simulations for the specified number of processing elements (as given in the source code listing 5). DAE Tools is free software released under the GNU

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

Both versions of the simulator are cross-platform and the simulation inputs are specified in a platform-independent way using input files with the model specification and run-time options, as described in the section *Algorithm for partitioning of general systems of equations*. This way, the same model can be simulated using the same software on all platforms. 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). Systems of linear equations are solved using the Krylov-subspace iterative methods. Currently, two solvers are available: the generalised minimal residual solver from the SUNDIALS suite and the generalised minimal residual solver from the Trilinos AztecOO interface to the Aztec solver library (Heroux et al., 2005). Both solvers utilise preconditioners available from the Trilinos suite: IFPACK, ML and AztecOO built-in preconditioners. For computationally intensive tasks (i.e. evaluation of residuals and derivatives) the software can utilise multi-level parallelism techniques such as hybrid MPI/OpenMP and heterogeneous MPI/OpenCL. The commands for running sequential and parallel simulations on different platforms is presented in the source code listing 7.

Listing 7. Running simulations using csSimulator_DAE

```
# 1. Simulation in GNU/Linux
  # Sequential simulation (for Npe = 1):
  $ csSimulator_DAE "input_files_directory"

  # Parallel simulation (for Npe > 1) using Open MPI:
  $ mpirun -np Npe csSimulator_DAE "input_files_directory"

# 2. Simulation in Windows
  # Sequential simulation (for Npe = 1):
  $ csSimulator_DAE.exe "input_files_directory"

  # Parallel simulation (for Npe > 1) using Microsoft MPI:
  $ mpiexec -n Npe csSimulator_DAE.exe "input_files_directory"
```

CASE STUDY

Transient two-dimensional Cahn-Hilliard equation, unstructured grid

The model describes the process of phase separation, where two components of a binary mixture separate and form domains pure in each component. The problem is carefully selected to illustrate both the capabilities and the current limitations of the proposed methodology. The system is described by the Cahn-Hilliard equations given in eq. 2.

$$\begin{aligned} \frac{dc}{dt} &= D\nabla^2\mu \\ \mu &= c^3 - c - \gamma\nabla^2 c \end{aligned} \tag{2}$$

Here, D is the diffusion coefficient, c is the concentration, μ is the chemical potential and $\sqrt{\gamma}$ defines the length of the transition regions between the domains. The mesh is a simple square (0,100)x(0,100) with 100x100 elements. Input parameters are $D = 1$ and $\gamma = 1$. Boundary conditions for both c an μ are insulated boundary conditions (no flux on boundaries). Initial conditions are set to $c(0) = 0.5 + c_{noise}$ where the noise c_{noise} is specified using the normal distribution with standard deviation of 0.1. The system is integrated for 500 seconds and the outputs are taken every 5 seconds. The source code is given in the Supplemental Listing S1 (case_1.py file) and on DAE Tools website (http://www.daetools.com/docs/tutorials-fe.html#tutorial-dealii-3).

The Cahn-Hilliard equations are spatially discretised using the Finite Elements Method. In DAE Tools, deal.II (http://dealii.org) library is utilised for low-level tasks such as mesh loading, management of finite element spaces, degrees of freedom, assembly of the system stiffness and mass matrices and the system load vector, and setting the boundary conditions. The assembled system matrices are used to generate a set of equations in the following form: $[M]\{\dot{x}\} + [A]\{x\} - \{F\} = 0$, where x and $\dot{x}$ are vectors of state variables and their derivatives, M and A are mass and stiffness matrices and F is the load vector. The generated set of equations (in general case a DAE system) are solved together with the rest of equations in the model.

The model is implemented in Python using the DAE Tools v1.8.1 software (Nikolić, 2016). Systems of linear equations are solved using the SUNDIALS preconditioned generalised minimal residual solver using the IFPACK (Sala and Heroux, 2005) ILU preconditioner from Trilinos suite (Heroux et al., 2005). The total of 11 different runs have been performed (Table 6): (a) sequential run using the DAE Tools software (S-1), (b) sequential run using the csSimulator_DAE simulator (S-2), (c) eight parallel runs using the csSimulator_DAE simulator with different balancing constraints applied to the partitioning algorithm (P-1 to P-8), and (d) one parallel run using the csSimulator_DAE simulator where the system is manually partitioned by dividing the 2D mesh into four quadrants (P-9). 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 S-1 and S-2 (with Npe = 1) and an average for parallel runs P-1 to P-9 (with Npe = 4) are given in Table 7. The input parameters for the IFPACK preconditioner for all runs are given in Table 8 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, 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), and a discrete NVidia GeForce GTX 950M GPU (640 execution units at 914 MHz, 2 GB of RAM).

Table 6. Simulation runs and description of objectives

Run	Balancing constraints	Description
S-1	-	Sequential simulation using DAE Tools
S-2	-	Sequential simulation using csSimulator_DAE
P-1	None	Edge-cut only
P-2	Ncs	Balance the number of ComputeStack items
P-3	Nnz	Balance the number of non-zero items in the incidence matrix
P-4	Nflops	Balance the number of FLOPS for evaluation of residuals
P-5	Nflops_j	Balance the number of FLOPS for evaluation of the Jacobian
P-6	Ncs, Nflops	Balance both the number of ComputeStack items and the number of FLOPS for evaluation of residuals
P-7	Nnz, Nflops_j	Balance both the number of non-zero items in the incidence matrix and the number of FLOPS for evaluation of the Jacobian
P-8	Ncs, Nnz, Nflops, Nflops_j	Balance all (constraints from runs P-6 and P-7 combined)
P-9	-	Manual partition of a 2D mesh into 4 quadrants

Table 7. Workload-related properties for sequential and parallel runs in Case 1

Run	N_{eq}	N_{nz}	N_{cs}	$N_{nz/equation}$	$N_{cs/equation}$	$N_{cs/jacob\_row}$
Sequential	20,000	355,216	15,554,304	17.76	778	13,902
Parallel	∼5,000	∼88,804	∼3,888,576	∼17.76	∼778	∼13,902

Table 8. IFPACK preconditioner parameters for sequential and parallel runs in Case 1

Run	k	ρ	α	ω
S-1	3	1.0	10^{-5}	0.0
S-2	3	1.0	10^{-5}	0.0
P-1	3	2.0	0.1	0.5
P-2	3	2.0	0.1	0.5
P-3	4	2.0	0.1	1.0
P-4	4	2.0	0.1	0.0
P-5	3	2.0	0.1	0.5
P-6	3	2.0	0.1	0.5
P-7	4	2.0	0.1	1.0
P-8	3	2.0	0.1	0.5
P-9	3	2.0	0.1	0.5

RESULTS

The detailed statistical data generated by the partition algorithm such as: the number of equations (N_{eq}), the number of adjacent variables in every partition (N_{adj}) and deviations from the average value (in percents) for the number of equations (N_{eq}^{dev}), the number of adjacent variables (N_{adj}^{dev}) and all available balancing constraints (N_{cs}^{dev}, N_{flops}^{dev}, N_{nz}^{dev} and $N_{flops\_j}^{dev}$) are given in Supplemental Tables S1 to S9. The following six phases of the numerical solution are analysed: (1) evaluation of residuals (*EvaluateResiduals*), (2) evaluation of the Jacobian matrix (*EvaluateJacobian*), (3) computation of the preconditioner, excluding the time for evaluation of the Jacobian (*ComputePreconditioner*), (4) application of the preconditioner to solve the linear system (*ApplyPreconditioner*), (5) Jacobian-vector multiplication, required in every iteration of the linear solver (*JacobianVectorProduct*; in SUNDIALS IDAS the difference quotient approximation is used and requires an additional call to the EvaluateResiduals function), and (6) exchange of adjacent variables between processing elements, required in every call to Evaluate Residuals (*InterProcessDataExchange*). The *EvaluateResiduals* phase includes calls from the DAE solver (once per every DAE step taken) and calls from the linear solver in the Jacobian-vector product function (since the difference quotient approximation is used). The duration of the *JacobianVectorProduct* phase includes the time for evaluation of residuals.

The performance and the simulation results of the sequential runs S-1 and S-2 (csSimulator_DAE versus DAE Tools simulation software) are compared for three different cases where model equations are evaluated: (a) sequentially, (b) using the OpenMP API on a multi-core Intel CPU, and (c) using the OpenCL framework on NVida GPU. The duration of five phases of the numerical solution in runs S-1 and S-2 are presented in Table 9 (the *InterProcessDataExchange* phase is absent in sequential runs). The percentage of the integration time for individual phases in runs S-1 and S-2 where equations are evaluated sequentially is given in Table 10.

The quality of simulation results in all runs are assessed using the normalised global error: $\|E\| = \sqrt{\frac{1}{Neq}\sum\left(x[i] - x_{DAE\_Tools}[i]\right)^2}$, where x and x_{DAE_Tools} are results obtained using the csSimulator_DAE and the DAE Tools software, respectively. For parallel runs, the integration times, the normalised global errors and an estimate for the total overheads due to the load imbalance are given in Table 11. The solver statistics for sequential and parallel runs are given in Table 12. The detailed statistics (the total time, the number of calls, average time per call, total overheads and deviations from the average overhead) for six phases of the sequential and parallel simulations are given in Tables 13 to 18, respectively. Durations of the above six phases are measured for every call throughout the simulation. An overhead for a single call is calculated using: $max(durations) - min(durations)$, where *durations* is an array of Npe items each representing the time required for the phase to complete - one for every processing element. The overhead times for every call during the parallel runs P-1 to P-9 are plotted in Supplemental Fig. S1 to S9. The total overhead due to the load imbalance is calculated by summing up all individual overheads. However, it must be kept in mind that these are only estimates, since there are no explicit synchronisation points after every phase. An average time for each phase is calculated and actual deviations from the average (in percents) obtained for every processing element. This way, the prediction quality of the load balancing algorithm is assessed by comparing the actual (measured) load imbalances in Tables 13 to 18 with the predicted load imbalances given in Supplemental Tables S1 to S9. The comparison between predicted and actual maximum absolute deviations from the average (in percents) are given in Tables 19, 20 and 21.

Table 9. Duration in seconds of individual phases in the sequential runs S-1 and S-2

	DAE Tools (S-1)			csSimulator_DAE (S-2)		
	Sequential	OpenMP	OpenCL	Sequential	OpenMP	OpenCL
Total integration time	181.17	80.32	49.46	180.01	65.57	49.37
EvaluateResiduals	125.80	53.39	36.71	124.75	42.16	36.53
EvaluateJacobian	47.01	15.61	4.14	46.62	13.61	4.09
ComputePreconditioner	3.49	5.26	3.90	3.76	3.87	3.88
ApplyPreconditioner	4.29	5.40	4.17	4.24	5.22	4.24
JacobianVectorProduct	79.78	34.09	23.43	79.65	26.92	23.33

Table 10. Time spent in individual phases of the solution (given as percentage of the total integration time) in runs S-1 and S-2 for the case where equations are evaluated sequentially

	DAE Tools (S-1)	csSimulator_DAE (S-2)
EvaluateResiduals (total)	69.44 %	69.30 %
EvaluateResiduals (DAE solver only)	25.11 %	25.05 %
EvaluateJacobian	25.95 %	25.90 %
ComputePreconditioner	1.92 %	2.09 %
ApplyPreconditioner	2.37 %	2.36 %
JacobianVectorProduct	44.34 %	44.27 %
DAE solver	0.31 %	0.33 %

Table 11. Normalised global errors, integration duration and total overheads in parallel runs

			Overhead	
Run	$\|\|E\|\|$	Integration time, s	Time, s	%
P-1	1.83e-05	136.74	3.17	2.32
P-2	1.55e-05	152.74	3.12	2.04
P-3	3.51e-05	157.93	2.63	1.67
P-4	1.83e-05	156.32	3.19	2.04
P-5	1.51e-05	138.81	2.69	1.94
P-6	1.39e-05	152.17	4.53	2.98
P-7	1.86e-05	151.57	6.96	4.59
P-8	3.07e-05	108.67	12.97	11.94
P-9	2.07e-05	103.14	3.17	3.08

Table 12. Solvers statistics in sequential and parallel runs

Run	DAE solver No. steps	Nonlinear solver No. iterations	Linear solver No. iterations	Linear solver Average no. iterations until convergence
S-1	489	557	987	1.77
S-2	489	555	984	1.77
P-1	738	865	4081	4.72
P-2	723	854	4627	5.42
P-3	756	900	4612	5.12
P-4	744	878	4634	5.28
P-5	726	852	4046	4.75
P-6	747	871	4543	5.22
P-7	738	898	4132	4.60
P-8	781	908	2663	2.93
P-9	721	851	2674	3.14

Table 13. Statistics for the *EvaluateResiduals* phase

Run	Time, s	No. calls	Time/call, ms	Overhead Time, s	Overhead %	Deviation from average, % PE_0	PE_1	PE_2	PE_3
S-1	125.80	1546	81.37	-	-	-	-	-	-
S-2	124.75	1541	80.95	-	-	-	-	-	-
P-1	114.86	4948	23.21	1.36	0.99	0.17	-0.17	-0.39	0.40
P-2	127.52	5483	23.26	1.41	0.93	-0.04	-0.09	0.19	-0.06
P-3	126.97	5514	23.03	1.23	0.78	-0.06	0.03	-0.14	0.16
P-4	127.26	5514	23.08	1.43	0.92	-0.19	-0.06	-0.11	0.36
P-5	115.48	4900	23.57	1.19	0.86	-0.08	0.04	-0.04	0.08
P-6	126.22	5416	23.31	2.07	1.36	0.43	-0.57	-0.09	0.23
P-7	119.77	5032	23.80	2.96	1.95	0.81	-1.02	0.89	-0.67
P-8	85.28	3573	23.87	6.06	5.58	0.25	-2.62	4.09	-1.72
P-9	81.65	3527	23.15	1.43	1.39	0.18	0.23	-0.24	-0.18

Table 14. Statistics for the *EvaluateJacobian* phase

				Overhead		Deviation from average, %			
Run	Time, s	No. calls	Time/call, ms	Time, s	%	PE_0	PE_1	PE_2	PE_3
S-1	47.01	33	1424.55	-	-	-	-	-	-
S-2	46.62	33	1412.72	-	-	-	-	-	-
P-1	12.92	33	391.58	0.18	0.13	0.11	-0.11	-0.57	0.58
P-2	14.30	37	386.52	0.11	0.07	-0.13	-0.12	0.22	0.04
P-3	15.86	41	386.85	0.11	0.07	-0.06	-0.05	0.05	0.06
P-4	13.93	36	386.84	0.12	0.08	-0.03	-0.12	-0.10	0.26
P-5	13.88	35	396.52	0.10	0.08	-0.01	0.25	-0.28	0.03
P-6	15.93	41	388.61	0.23	0.15	0.24	-0.52	-0.19	0.47
P-7	18.56	46	403.38	0.49	0.32	1.36	-0.97	0.45	-0.84
P-8	14.77	37	399.27	1.00	0.92	0.29	-2.65	4.00	-1.65
P-9	14.59	37	394.27	0.23	0.22	0.07	0.04	-0.04	-0.08

Table 15. Statistics for the *ComputePreconditioner* phase (excluding the Jacobian evaluation)

				Overhead		Deviation from average, %			
Run	Time, s	No. calls	Time/call, ms	Time, s	%	PE_0	PE_1	PE_2	PE_3
S-1	3.49	33	105.76	-	-	-	-	-	-
S-2	3.76	33	113.94	-	-	-	-	-	-
P-1	1.09	33	33.03	0.01	0.01	0.18	-0.04	-0.62	0.49
P-2	1.21	37	32.70	0.01	0.01	-0.08	-0.11	0.15	0.04
P-3	2.37	41	57.80	0.01	0.01	0.02	-0.07	-0.00	0.06
P-4	1.93	36	53.61	0.01	0.01	-0.02	-0.12	-0.03	0.16
P-5	1.16	35	33.14	0.03	0.02	-0.05	0.23	-0.37	0.19
P-6	1.28	41	31.21	0.01	0.01	0.27	-0.50	-0.15	0.39
P-7	2.64	46	57.39	0.14	0.09	1.37	-1.07	0.63	-0.94
P-8	1.17	37	31.62	0.07	0.07	0.40	-2.69	3.91	-1.61
P-9	1.18	37	31.89	0.02	0.02	0.05	0.03	-0.03	-0.05

Table 16. Statistics for the *ApplyPreconditioner* phase

				Overhead		Deviation from average, %			
Run	Time, s	No. calls	Time/call, ms	Time, s	%	PE_0	PE_1	PE_2	PE_3
S-1	4.29	1546	2.77	-	-	-	-	-	-
S-2	4.24	1541	2.75	-	-	-	-	-	-
P-1	7.35	4948	1.49	0.48	0.35	-0.14	1.25	1.25	-2.36
P-2	8.47	5483	1.54	0.54	0.36	0.41	1.05	-0.55	-0.90
P-3	11.87	5514	2.15	0.45	0.29	0.17	0.04	0.40	-0.61
P-4	12.03	5514	2.18	0.57	0.36	0.47	0.32	0.34	-1.14
P-5	7.49	4900	1.53	0.37	0.27	0.17	-0.32	0.55	-0.40
P-6	7.96	5416	1.47	0.64	0.42	-1.92	1.93	1.02	-1.03
P-7	9.95	5032	1.98	0.89	0.59	-2.86	2.25	-2.64	3.26
P-8	4.36	3573	1.22	1.25	1.15	0.31	8.27	-16.78	8.21
P-9	5.16	3527	1.46	0.38	0.37	-1.46	0.62	-0.01	0.85

Table 17. Statistics for the *JacobianVectorProduct* phase

				Overhead		Deviation from average, %			
Run	Time, s	No. calls	Time/call, ms	Time, s	%	PE_0	PE_1	PE_2	PE_3
S-1	80.31	987	81.32	-	-	-	-	-	-
S-2	79.65	984	80.84	-	-	-	-	-	-
P-1	93.59	4081	22.93	1.10	0.81	0.18	-0.19	-0.41	0.42
P-2	106.51	4627	23.02	1.02	0.67	-0.00	-0.06	0.08	-0.02
P-3	105.12	4612	22.79	0.83	0.52	-0.02	0.05	-0.11	0.08
P-4	106.08	4634	22.89	1.04	0.67	-0.17	-0.05	-0.06	0.29
P-5	94.41	4046	23.34	0.97	0.70	-0.10	0.04	-0.04	0.11
P-6	104.83	4543	23.08	1.54	1.01	0.48	-0.52	-0.24	0.28
P-7	97.43	4132	23.58	2.46	1.62	0.86	-1.09	0.93	-0.70
P-8	62.36	2663	23.42	4.56	4.20	0.20	-2.80	4.27	-1.67
P-9	61.89	2674	23.14	1.10	1.06	0.21	0.22	-0.25	-0.18

Table 18. Statistics for the *InterProcessDataExchange* phase

				Overhead		Deviation from average, %			
Run	Time, s	No. calls	Time/call, ms	Time, s	%	PE_0	PE_1	PE_2	PE_3
P-1	0.16	4948	0.03	0.04	0.03	9.29	-2.93	-13.07	6.71
P-2	0.17	5483	0.03	0.04	0.03	3.08	-4.66	-11.36	12.94
P-3	0.14	5514	0.03	0.03	0.02	-8.69	4.79	-2.07	5.97
P-4	0.17	5514	0.03	0.03	0.02	3.77	-6.76	-4.83	7.82
P-5	0.14	4900	0.03	0.02	0.01	-1.90	3.22	2.89	-4.21
P-6	0.14	5416	0.03	0.05	0.03	-8.27	1.45	-9.60	16.42
P-7	0.15	5032	0.03	0.03	0.02	5.28	-8.51	-5.99	9.21
P-8	0.09	3573	0.02	0.02	0.02	-11.43	4.63	9.98	-3.18
P-9	0.11	3527	0.03	0.02	0.02	2.23	-3.21	6.32	-5.34

Table 19. Predicted vs. actual load imbalance in the *EvaluateResiduals* phase

	Predicted (%)		Actual (%)
Run	N_{cs}^{dev}	N_{flops}^{dev}	
P-1	0.22	0.22	0.40
P-2	0.01	0.01	0.19
P-3	0.09	0.09	0.16
P-4	0.01	0.01	0.36
P-5	0.06	0.06	0.08
P-6	0.39	0.39	0.57
P-7	0.87	0.87	1.02
P-8	4.06	4.06	4.09
P-9	0.00	0.00	0.24

Table 20. Predicted vs. actual load imbalance in the *EvaluateJacobian* phase

	Predicted (%)		Actual (%)
Run	N_{nz}^{dev}	$N_{flops\_j}^{dev}$	
P-1	0.15	0.30	0.58
P-2	0.10	0.09	0.22
P-3	0.01	0.13	0.06
P-4	0.10	0.09	0.26
P-5	0.10	0.02	0.28
P-6	0.45	0.42	0.52
P-7	0.82	0.89	1.36
P-8	4.00	4.12	4.00
P-9	0.00	0.00	0.08

Table 21. Predicted vs. actual load imbalance in the *InterProcessDataExchange* phase

Run	Predicted (%) N_{adj}^{dev}	Actual (%)
P-1	15.68	13.07
P-2	20.22	12.94
P-3	13.17	8.69
P-4	20.22	7.82
P-5	4.80	4.21
P-6	21.29	16.42
P-7	9.94	9.21
P-8	18.88	11.43
P-9	20.47	6.32

DISCUSSION

The quality of numerical solutions produced by the csSimulator_DAE software is verified by comparison between the sequential runs S-1 and S-2 (Table 9). The simulation results obtained using three different Compute Stack Evaluator implementations (sequential, OpenMP and OpenCL) are compared using the normalised global error ($\|E\|$). Both software use identical solvers and all simulations were performed using identical input parameters. The only difference is that csSimulator_DAE uses the parallel linear algebra routines. As expected, the simulation results are practically identical in all six sequential runs with $\|E\| \approx 0$.

The overall performance and duration of individual phases of the sequential numerical solution in both simulators are approximately the same (Table 9). csSimulator_DAE performs slightly faster due to the overhead of Python functions that the DAE Tools simulator frequently calls. In addition, the solver statistics such as the number of steps taken by the DAE solver, the number of linear and non-linear solver iterations and the number of evaluations of residuals and the Jacobian are approximately the same. From Table 10 it can be seen that computationally most intensive phases are: *EvaluateResiduals*, *EvaluateJacobian* and *JacobianVectorProduct*, where approximately 25%, 26% and 44% of the total integration time is spent, respectively. Since all three phases require evaluation of model equations, the evaluation of model equations, combined from different phases, requires approximately 95% of the total integration time.

The simulation results from parallel runs P-1 to P-9 also agree very well with the results from the sequential run S-1. The normalised global errors (Table 11) are of the order of magnitude 10^{-5} which is in accordance with the absolute and relative tolerances used (10^{-5}).

The observed speed-ups in evaluation of model equations in all parallel runs are as expected: approximately 4 and 3.6 for *EvaluateResiduals* and *EvaluateJacobian* phases, respectively (Tables 13 and 14). Theoretically, since there are no dependency nor data exchange between processing elements, evaluation of equations should scale linearly with the increase in the number of processing elements. In addition, the speed-ups in the *JacobianVectorProduct* phase are approximately 3.5 (Table 17). Thus, the achieved speed-ups correspond well to the maximum theoretical speed-up of four.

Regarding the efficiency of the linear solver, the observed speed-ups in the *ComputePreconditioner* (excluding evaluation of the Jacobian) and the *ApplyPreconditioner* phases are 2.0-3.6 and 1.3-2.3, respectively (Tables 15 and 16). The incomplete LU factorisation and application

of the preconditioner are performed on four times smaller systems of linear equations. Hence, the achieved speed-ups are lower than the maximum speed-up expected in an ideal case. In addition, it can be observed that the number of linear solver iterations until convergence (per non-linear iteration) is significantly higher than in the sequential runs S-1 and S-2. The number of iterations in the linear solver in runs S-1 and S-2 is 1.77, while it is between 2.93 and 5.42 in parallel runs (Table 12). The most probable cause is the structure of partitions, that is the set of adjacent variables produced by the partitioning algorithm. Since the adjacent variables are removed from DAE sub-systems in processing elements, the resulting Jacobian matrices are modified as well. This fact affects the incomplete LU factorisation and, depending on the scale of adjacent variables, can produce less efficient preconditioners. Consequently, the number of iterations to reach the convergence is larger. The lowest number of iterations are recorded in runs P-8 and P-9 (Table 12).

The number of iterations in the linear solver has a large effect on the overall simulation performance. Although all parallel runs perform faster than the sequential runs S-1 and S-2 (Table 11), the overall improvements of only 13 to 43% are far from the theoretical maximum. The main reason is a poor performance of the linear solver. Furthermore, every linear solver iteration requires a call to a costly Jacobian-vector multiply function which in the SUNDIALS implementation involves an additional call to the EvaluateResiduals function and in total takes 60-70% of the integration time (Table 17). The total number of calls to EvaluateResiduals function (from the DAE solver and the Jacobian-vector multiply function combined) are given in Table 13. A large number of linear solver iterations requiring a large number of calls to EvaluateResiduals function is the main cause for the low overall performance, although the performance of other phases (*EvaluateResiduals*, *EvaluateJacobian*, *ComputePrecontitioner* and *ApplyPreconditioner*) increase approximately linearly with the number of processing elements. Therefore, partitioning of a DAE system is an extremely important phase of the parallel numerical solution and, in order to exploit the problem-specific structure of model equations in certain cases, the current implementation of the partitioning algorithm must be modified.

The total overheads from all phases combined are given in Table 11. Again, it must be kept in mind that these are only estimates, since there are no explicit synchronisation points after every phase. Thus, the load imbalance does not have a significant impact on the overall performance except in runs P-7 and P-8 which use multiple balancing constraints with somewhat higher load imbalances. The overheads in individual phases are given in Tables 13 to 18. Significant overheads are recorded only in the *EvaluateResiduals* phase due to a large number of calls. In particular, the largest overhead is recorded in runs P-7 and P-8 where the largest overheads are predicted and expected.

The difference between the predicted and actual load imbalances is quantified by a maximum absolute deviation from the average duration (in percents). The predicted load imbalance in the *EvaluateResiduals* phase is quantified by the maximum deviation from average in N_{cs} and N_{flops} (N_{cs}^{dev} and N_{flops}^{dev}), in the *EvaluateJacobian* phase by deviation in N_{nz} and N_{flops_j} (N_{nz}^{dev} and $N_{flops\_j}^{dev}$) and in the *InterProcessDataExchange* phase by deviation in N_{adj} (N_{adj}^{dev}). The prediction quality of the partitioning algorithm based on 4 available balancing constraints is very accurate. The maximum differences between predicted and actual load imbalance are: (a) for *EvaluateResiduals* phase (Table 19): 0.02 to 0.35% in both N_{cs} and N_{flops}, (b) for *EvaluateJacobian* phase (Table 20): 0.00 to 0.54% in N_{nz} and 0.07 to 0.47% in N_{flops_j}, and (c) for *InterProcessDataExchange* phase (Table 21): 0.59 to 14.15% in N_{adj}. Not only the maximum

deviations from average are well predicted but the deviations in individual PEs also closely follow the predicted values.

The best load balance predictions in individual phases are as expected: (a) for *EvaluateResiduals* phase in run P-2 (N_{cs} is used as a balancing constraint): $N_{cs} = 0.00\%$, $N_{flops} = 0.01\%$, (b) for *EvaluateJacobian* phase in runs P-3 (N_{nz} as a balancing constraint): $N_{nz} = 0.01\%$, $N_{flops\_j} = 0.13\%$, and P-5 (N_{flops_j} as a balancing constraint): $N_{nz} = 0.10\%$, $N_{flops\_j} = 0.02\%$, (c) for *InterProcessDataExchange* phase in run P-5 (N_{flops_j} as a balancing constraint): $N_{adj} = 4.80\%$. Thus, the partitioning algorithm precisely and accurately predicts the workloads in the critical phases of the numerical solution (particularly in phases that involve evaluation of model equations).

CONCLUSIONS

In this work, the methodology for parallel numerical solution of general systems of non-linear differential and algebraic equations on distributed memory systems has been presented. It is based on the previously developed methodology for parallel evaluation of general systems of differential and algebraic equations on shared memory systems (Nikolić, 2018) and consists of the following parts: (1) an algorithm for transformation of model equations into a data structure suitable for parallel evaluation on different computing platforms, (2) data structures for model specification, (3) an algorithm for partitioning of general systems of equations, (4) an algorithm for inter-process data exchange, and (5) simulation software for integration of general DAE systems in time.

Model equations are specified in a platform and programming language independent fashion as the Reverse Polish (postfix) notation expression stacks (Compute Stacks). The Compute Stack can represent any type of equations of any size and be evaluated using a stack machine (Compute Stack Machine) on virtually all computing devices (due to its simplicity). The model specification contains only the low-level model description with the minimum information required for integration in time, stored in C++ data structures. The data structures holding the model specification represent a simple binary interface for model exchange. In contrast to the existing approaches, the model description in this work does not require a human or a machine readable model definition nor shared libraries providing the C API. For instance, in this approach, the model equations are directly evaluated on all platforms/operating systems with no additional processing. The partitioning algorithm accurately balances the computation and memory loads in all important phases of the numerical solution. The simulation software 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 DAE system in time. Simulation inputs are specified in a generic fashion using the data structures with the model specification stored as files in binary format. For computationally intensive tasks the simulation software utilises multi-level parallelism techniques such as hybrid MPI/OpenMP and heterogeneous MPI/OpenCL.

The proposed methodology has been applied to a medium-scale transient phase separation process. Six different phases of the numerical solution (*EvaluateResiduals*, *EvaluateJacobian*, *ComputePreconditioner*, *ApplyPreconditioner*, *JacobianVectorProduct* and *InterProcessDataExchange*) and nine different load balancing strategies have been analysed. Typically, 95% of the total integration time is spent on evaluation of model equations and the Jacobian-vector multiplication in the linear solver. The simulation results have been assessed and verified using the normalised global error. An overall performance and performance in individual phases

have been compared to the sequential simulations. As expected, the performance of phases that include evaluation of model equations scale linearly with the number of processing elements. However, the overall simulation performance in parallel runs is far from the maximum due to the inefficiency of the preconditioner. The reason is that, since the partitioning algorithm treats a DAE system as a black-box, in some cases the generated sets of equations and adjacent variables in partitions does not allow creation of efficient preconditioners. The prediction quality of the static load balancing algorithm and overheads due to the load imbalance have been discussed in details. It has been found that the workloads in the critical phases of the numerical solution (evaluation of equation residuals and derivatives) are very accurately predicted.

The strengths and limitations of the methodology have been discussed. The methodology is difficult to apply to problems that use adaptive grids since the total number of variables/equations change during the simulation. In addition, the partitioning algorithm must be improved to take advantage of the specific structure of model equations in some cases (i.e. the systems produced by discretisation of well known partial-differential equations on uniform grids).

The future work will be focused on unifying the methodologies for parallel solution of general systems of differential and algebraic equations on shared and distributed memory systems into a single framework (Open Compute Stack), improvement of the partitioning algorithm and the load balancing strategies, creation of check points and routines for recovery after errors, wrapping new preconditioner libraries, and new Compute Stack Evaluator implementations for additional types of computing devices (such as Xeon Phi and FPGA).

REFERENCES

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

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.

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

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

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: 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

945 Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):
946 237–254, 2006. https://doi.org/10.1007/s00366-006-0049-3.
947 Andreas Kloeckner, 2018. URL http://mathema.tician.de/software/pymetis.
948 Dragan D. Nikolić. DAE Tools: Equation-based object-oriented modelling, simulation and
949 optimisation software. *PeerJ Computer Science*, 2:e54, April 2016. ISSN 2376-5992. doi:
950 10.7717/peerj-cs.54. URL https://doi.org/10.7717/peerj-cs.54.
951 Dragan D. Nikolić. Parallelisation of equation-based simulation programs on heterogeneous
952 computing systems. *PeerJ Computer Science*, 4:e160, August 2018. ISSN 2376-5992. doi:
953 10.7717/peerj-cs.160. URL http://dx.doi.org/10.7717/peerj-cs.160.
954 M. Sala and M. Heroux. Robust algebraic preconditioners with IFPACK 3.0. Technical Report
955 SAND-0662, Sandia National Laboratories, 2005.
956 Siemens. Multidisciplinary Design Exploration, 2018. URL https://mdx.plm.automation.siemens.
957 com.
958 The MathWorks, Inc. Simulink, 2018. URL https://mathworks.com/products/matlab.
959 The OpenFOAM Foundation. OpenFOAM, 2018. URL http://www.openfoam.org.