Parapint

Michael Bynum Carl Laird Bethany Nicholson Denis Ridzal

Mar 18, 2022

CONTENTS:

Parapint is a Python Package for parallel solution of dynamic optimization problems.

CHAPTER

ONE

OVERVIEW

Parapint is a package for parallel solution of dynamic optimization problems. Parapint currently includes a Schur-Complement decomposition algorithm based on [Word2014]. Parapint utilizes Pynumero *BlockVector* and *BlockMatrix* classes (which in turn utilize Numpy arrays and Scipy sparse matrices) for efficient block-based linear algebra operations such as block-matrix, block-vector dot products. These classes enable convenient construction of block-structured KKT systems. Parapint also utilizes Pynumero interfaces to efficient numerical routines in C, C++, and Fortran, including the AMPL Solver Library (ASL), MUMPS, and the MA27 routines from the Harwell Subroutine Library (HSL).

Parapint is designed with three primary modules:

- The algorithms. The algorithms drive the solution process and perform high level operations such as the fraction-to-the boundary rule or inertia correction for the interior point algorithm. The interior point algorithm is designed to work with any `BaseInteriorPointInterface` and any `LinearSolverInterface` as long as the interface and the linear solver are compatible.
- The interfaces. All interfaces should inherit from `BaseInteriorPointInterface` and implement all abstract methods. These are the methods required by the interior point algorithm. The interfaces are designed to work with a subset of linear solvers. The table below outlines which interfaces work with which linear solvers.
- The linear solvers. All linear solvers should inherit from `LinearSolverInterface` and implement all abstract methods. These are the methods required by the interior point algorithm. The linear solvers are designed to work with certain interface classes. The table below outlines which linear solvers work with which interfaces.

Table 1: Compatible linear solvers and interfaces

Linear Solver	Compatible Interface Class
`InteriorPointMA27Interface`	`InteriorPointInterface`
`MumpsInterface`	`InteriorPointInterface`
`ScipyInterface`	`InteriorPointInterface`
`SchurComplementLinearSolver`	`DynamicSchurComplementInteriorPointInterface`
`MPISchurComplementLinearSolver`	`MPIDynamicSchurComplementInteriorPointInterface`

CHAPTER

TWO

INSTALLATION

Parapint can be installed by cloning the parapint repository from https://github.com/parapint/parapint

```
git clone https://github.com/parapint/parapint.git
cd parapint/
python setup.py install
```

2.1 Requirements

Parapint requires Python (at least version 3.7) and the following packages:

- Numpy (version 1.13.0 or greater)
- Scipy
- Pyomo (Parapint currently only works with the master branch of Pyomo)

Pyomo should be installed from source and used to build Pynumero extensions:

```
pip install numpy
pip install scipy
git clone https://github.com/pyomo/pyomo.git
cd pyomo/
python setup.py install
cd pyomo/contrib/pynumero/
python build.py -DBUILD_ASL=ON -DBUILD_MA27=ON -DIPOPT_DIR=<path/to/ipopt/build/>
```

Pymumps also needs to be installed in order to use MUMPS:

```
conda install pymumps
```

CHAPTER

THREE

SOLVING DYNAMIC OPTIMIZATION PROBLEMS WITH SCHUR-COMPLEMENT DECOMPOSITION

In order to solve a dynamic optimization problem with schur-complement decomposition, you must create a class which inherits from *MPIDynamicSchurComplementInteriorPointInterface*. This class must implement the method *build_model_for_time_block()*:

```python
import parapint

class Problem(parapint.interfaces.MPIDynamicSchurComplementInteriorPointInterface):
    def __init__(self, your_arguments):
        # do anything you need to here
        super(Problem, self).__init__(start_t, end_t, num_time_blocks, mpi_comm)

    def build_model_for_time_block(self, ndx, start_t, end_t, add_init_conditions):
        # build the dynamic optimization problem with Pyomo over the time horizon
        # [start_t, end_t] and return the model along with two lists. The first
        # list should be a list of pyomo variables corresponding to the states at
        # start_t. The second list should be a list of pyomo variables
        # corresponding to the states at end_t. Continuity will be enforced
        # between the states at end_t for one time block
        # and the states at start_t for the next time block. It is very important for
        # the ordering of the state variables to be the same for every time block.

        return model, start_states, end_states

problem = Problem(some_arguments)
```

The *build_model_for_time_block()* method will be called once for every time block. It will be called within the call to `__init__()` of the super class (*MPIDynamicSchurComplementInteriorPointInterface*). Therefore, if you override the *__init__* method, it is very important to still call the *__init__* method of the base class as shown above. There is an example class in schur_complement.py in the examples directory within Parapint.

In addition to the implementation of the class described above, you must create an instance of *MPISchurComplementLinearSolver*. This linear solver requires a sub-solver for every time block:

```python
cntl_options = {1: 1e-6}  # the pivot tolerance
sub_solvers = {ndx: parapint.linalg.InteriorPointMA27Interface(cntl_options=cntl_
↪options) for ndx in range(num_time_blocks)}
schur_complement_solver = parapint.linalg.InteriorPointMA27Interface(cntl_options=cntl_
↪options)
linear_solver = parapint.linalg.MPISchurComplementLinearSolver(subproblem_solvers=sub_
↪solvers,
```

(continues on next page)

(continued from previous page)

```
                                                                    schur_complement_
↪solver=schur_complement_solver)
```

The linear solver and interface instances can then be used with the interior point algorithm:

```
options = parapint.algorithms.IPOptions()
options.linalg.solver = linear_solver
status = parapint.algorithms.ip_solve(interface, options)
assert status == parapint.interior_point.InteriorPointStatus.optimal
problem.load_primals_into_pyomo_model()
for ndx in problem.local_block_indices:
    model = problem.pyomo_model(ndx)
    model.pprint()
```

CHAPTER

FOUR

API DOCUMENTATION

4.1 parapint.linalg

4.1.1 Base Linear Solver Class

`class LinearSolverInterface`

Bases: `abc.ABC`

This is the base class for linear solvers that work with the interior point algorithm. Derived classes must implement the following abstract methods:

- do_symbolic_factorization
- do_numeric_factorization
- do_back_solve
- get_inertia

`abstract do_symbolic_factorization`(*matrix*, *raise_on_error=True*, *timer=None*)

Perform symbolic factorization with the nonzero structure of the matrix.

`abstract do_numeric_factorization`(*matrix*, *raise_on_error=True*, *timer=None*)

Factorize the matrix. Can only be called after do_symbolic_factorization.

`abstract do_back_solve`(*rhs*)

Solve the linear system matrix * x = rhs for x. Can only be called after do_numeric_factorization.

`abstract get_inertia`()

Get the inertia of the factorized matrix. Can only be called after do_numeric_factorization.

4.1.2 MA27 Interface

`class InteriorPointMA27Interface`(*cntl_options=None*, *icntl_options=None*, *iw_factor=1.2*, *a_factor=2*)

Bases: `parapint.linalg.base_linear_solver_interface.LinearSolverInterface`

An interface to HSL's MA27 routines for use with Parapint's interior point algorithm. See http://www.hsl.rl.ac.uk/archive/specs/ma27.pdf for details on the use of MA27.

Note: The pivot tolerance, cntl(1), should be selected carefully. Larger values result in better precision but smaller values result in better performance.

Parameters

cntl_options: dict See http://www.hsl.rl.ac.uk/archive/specs/ma27.pdf

icntl_options: dict See http://www.hsl.rl.ac.uk/archive/specs/ma27.pdf

iw_factor: float The factor for memory allocation of the integer working arrays used by MA27. This value is increased by the increase_memory_allocation method.

a_factor: float The factor for memory allocation of the A array used by MA28. This value is increased by the increase_memory_allocation_method.

`do_symbolic_factorization`(*matrix*, *raise_on_error=True*, *timer=None*)
Perform symbolic factorization. This calls the MA27A/AD routines.

Parameters

matrix: scipy.sparse.spmatrix or pyomo.contrib.pynumero.sparse.block_matrix.BlockMatrix The matrix to factorize

raise_on_error: bool If False, an error will not be raised if an error occurs during symbolic factorization. Instead the status attribute of the results object will indicate an error ocurred.

timer: HierarchicalTimer

Returns

res: LinearSolverResults A LinearSolverResults object with a status attribute for the LinearSolverStatus

`do_numeric_factorization`(*matrix*, *raise_on_error=True*, *timer=None*)
Perform numeric factorization. This calls the MA27B/BD routines.

Parameters

matrix: scipy.sparse.spmatrix or pyomo.contrib.pynumero.sparse.block_matrix.BlockMatrix The matrix to factorize

raise_on_error: bool If False, an error will not be raised if an error occurs during numeric factorization. Instead the status attribute of the results object will indicate an error ocurred.

timer: HierarchicalTimer

Returns

res: LinearSolverResults A LinearSolverResults object with a status attribute for the LinearSolverStatus

`increase_memory_allocation`(*factor*)
Increas the memory allocation for factorization. This method should only be called if the results status from do_symbolic_factorization or do_numeric_factorization is LinearSolverStatus.not_enough_memory.

Parameters

factor: float The factor by which to increase memory allocation. Should be greater than 1.

`do_back_solve`(*rhs*)
Performs a back solve with the factorized matrix. Should only be called after do_numeric_factorization.

Parameters

rhs: numpy.ndarray or BlockVector

Returns

result: numpy.ndarray or BlockVector

`get_inertia()`

Get the inertia. Should only be called after do_numeric_factorization.

Returns

num_pos: int The number of positive eigenvalues of A

num_neg: int The number of negative eigenvalues of A

num_zero: int The number of zero eigenvalues of A

`set_icntl`(*key*, *value*)

Set the value for an icntl option.

Parameters

key: int

value: int

`set_cntl`(*key*, *value*)

Set the value for a cntl option.

Parameters

key: int

value: float

`get_icntl`(*key*)

Get the value for an icntl option.

Parameters

key: int

Returns

val: int

`get_cntl`(*key*)

Get the value for a cntl option.

Parameters

key: int

Returns

val: float

4.1.3 MumpsInterface

`class MumpsInterface`(*par=1*, *comm=None*, *cntl_options=None*, *icntl_options=None*)

Bases: `parapint.linalg.base_linear_solver_interface.LinearSolverInterface`

`do_symbolic_factorization`(*matrix*, *raise_on_error=True*, *timer=None*)

Perform symbolic factorization with the nonzero structure of the matrix.

`do_numeric_factorization`(*matrix*, *raise_on_error=True*, *timer=None*)

Factorize the matrix. Can only be called after do_symbolic_factorization.

`do_back_solve`(*rhs*)

Solve the linear system matrix * x = rhs for x. Can only be called after do_numeric_factorization.

`get_inertia()`
Get the inertia of the factorized matrix. Can only be called after do_numeric_factorization.

4.1.4 ScipyInterface

class `ScipyInterface`(*compute_inertia=False*)
Bases: `parapint.linalg.base_linear_solver_interface.LinearSolverInterface`

`do_symbolic_factorization`(*matrix*, *raise_on_error=True*, *timer=None*)
Perform symbolic factorization with the nonzero structure of the matrix.

`do_numeric_factorization`(*matrix*, *raise_on_error=True*, *timer=None*)
Factorize the matrix. Can only be called after do_symbolic_factorization.

`do_back_solve`(*rhs*)
Solve the linear system matrix * x = rhs for x. Can only be called after do_numeric_factorization.

`get_inertia()`
Get the inertia of the factorized matrix. Can only be called after do_numeric_factorization.

4.1.5 Parallel Schur-Complement Linear Solver

class `MPISchurComplementLinearSolver`(*subproblem_solvers: Dict[int, parapint.linalg.base_linear_solver_interface.LinearSolverInterface]*, *schur_complement_solver: parapint.linalg.base_linear_solver_interface.LinearSolverInterface*)
Bases: `parapint.linalg.base_linear_solver_interface.LinearSolverInterface`

Solve the system Ax = b.

A must be block-bordered-diagonal and symmetric:

```
K1          transpose(A1)
    K2      transpose(A2)
        K3  transpose(A3)
A1  A2  A3  Q
```

Only the lower diagonal needs supplied.

Some assumptions are made on the block matrices provided to do_symbolic_factorization and do_numeric_factorization:

- Q must be owned by all processes
- K $_i$ and A $_i$ must be owned by the same process

Parameters

subproblem_solvers: dict Dictionary mapping block index to linear solver

schur_complement_solver: LinearSolverInterface Linear solver to use for factorizing the schur complement `do_symbolic_factorization`(*matrix: pyomo.contrib.pynumero.sparse.mpi_block_matrix.MPIBlockMatrix*, *raise_on_error: bool = True*, *timer: Optional[pyomo.common.timing.HierarchicalTimer] = None*) → parapint.linalg.results.LinearSolverResults

Perform symbolic factorization. This performs symbolic factorization for each diagonal block and collects some information on the structure of the schur complement for sparse communication in the numeric factorization phase.

Parameters

matrix: MPIBlockMatrix A Pynumero MPIBlockMatrix. This is the A matrix in Ax=b

raise_on_error: bool If False, an error will not be raised if an error occurs during symbolic factorization. Instead the status attribute of the results object will indicate an error ocurred.

timer: HierarchicalTimer A timer for profiling.

Returns

res: LinearSolverResults The results object

`do_numeric_factorization`(*matrix: pyomo.contrib.pynumero.sparse.mpi_block_matrix.MPIBlockMatrix*, *raise_on_error: bool = True*, *timer: Optional[pyomo.common.timing.HierarchicalTimer] = None*) → parapint.linalg.results.LinearSolverResults

Perform numeric factorization:

- perform numeric factorization on each diagonal block
- form and communicate the Schur-Complement
- factorize the schur-complement

This method should only be called after do_symbolic_factorization.

Parameters

matrix: MPIBlockMatrix A Pynumero MPIBlockMatrix. This is the A matrix in Ax=b

raise_on_error: bool If False, an error will not be raised if an error occurs during symbolic factorization. Instead the status attribute of the results object will indicate an error ocurred.

timer: HierarchicalTimer A timer for profiling.

Returns

res: LinearSolverResults The results object

`do_back_solve`(*rhs*, *timer=None*)

Performs a back solve with the factorized matrix. Should only be called after do_numeric_factorixation.

Parameters

rhs: MPIBlockVector

timer: HierarchicalTimer

Returns

result: MPIBlockVector

`get_inertia`()

Get the inertia. Should only be called after do_numeric_factorization.

Returns

num_pos: int The number of positive eigenvalues of A

num_neg: int The number of negative eigenvalues of A

num_zero: int The number of zero eigenvalues of A

`increase_memory_allocation`(*factor*)

Increases the memory allocation of each sub-solver. This method should only be called if the results status from do_symbolic_factorization or do_numeric_factorization is LinearSolverStatus.not_enough_memory.

Parameters

factor: float The factor by which to increase memory allocation. Should be greater than 1.

4.2 parapint.algorithms

4.2.1 InteriorPoint

4.3 parapint.interfaces

4.3.1 Base IP Interface

class `BaseInteriorPointInterface`

Bases: `abc.ABC`

A base class for interfacing with Parapint's interior point algorithm. This class is responsible for function evaluations and for building the KKT system (matrix and rhs).

4.3.2 IP Interface

class `InteriorPointInterface`(*pyomo_model*)

Bases: *`parapint.interfaces.interface.BaseInteriorPointInterface`*

4.3.3 Dynamic SC IP Interface

class `DynamicSchurComplementInteriorPointInterface`(*start_t: float*, *end_t: float*, *num_time_blocks: int*)

Bases: *`parapint.interfaces.interface.BaseInteriorPointInterface`*

A class for interfacing with Parapint's interior point algorithm for the serial solution of dynamic optimization problems. This class is primarily for testing purposes. Users should favor the MPIDynamicSchurComplementInteriorPointInterface class because it supports parallel solution. To utilize this class, create a class which inherits from this class and implement the build_model_for_time_block method. If you override the __init__ method make sure to call the super class' __init__ method at the end of the derived class' __init__ method. See ex1.py in the examples directory for an example.

Parameters

start_t: float The starting time for the dynamic optimization problem

end_t: float The final time for the dynamic optimization problem

num_time_blocks: int The number of time blocks to split the time horizon into for parallel solution. This is typically equal to the number of processes available (i.e., comm.Get_size()).

`abstract build_model_for_time_block`(*ndx: int*, *start_t: float*, *end_t: float*, *add_init_conditions: bool*) → Tuple[pyomo.core.base.block._BlockData, Sequence[pyomo.core.base.var._GeneralVarData], Sequence[pyomo.core.base.var._GeneralVarData]]

This method should be implemented by derived classes. This method should build (and return) the model for the time interval [start_t, end_t] and return a list of states at start_t and a list of states at end_t (in the same order). This method will be called once for each time block. The start_states and end_states returned by this method must be in the same order for every time block.

Parameters

ndx: int The time block index

start_t: float

end_t: float

add_init_conditions: bool This will only be True for time block 0.

Returns

pyomo_model: pyomo.core.base.block.Block The model for the time interval [start_t, end_t].

start_states: Sequence of _GeneralVarData a list of the states at start_t; the order of this list should be the same for every time block

end_states: Sequence of _GeneralVarData a list of the states at end_t; the order of this list should be the same for every time block

`n_primals()` → int

Returns

n_primals: int The number of primal variables

`primals_lb()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

primals_lb: BlockVector The lower bounds for each primal variable. This BlockVector has one block for every time block and one block for the coupling variables.

`primals_ub()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

primals_ub: BlockVector The upper bounds for each primal variable. This BlockVector has one block for every time block and one block for the coupling variables.

`init_primals()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

init_primals: BlockVector The initial values for each primal variable. This BlockVector has one block for every time block and one block for the coupling variables.

`set_primals`(*primals: pyomo.contrib.pynumero.sparse.block_vector.BlockVector*)

Set the values of the primal variables for evaluation (i.e., the evaluate_* methods).

Parameters

primals: BlockVector The values for each primal variable. This BlockVector should have one block for every time block and one block for the coupling variables.

`get_primals()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

primals: BlockVector The values for each primal variable. This BlockVector has one block for every time block and one block for the coupling variables.

`evaluate_objective()` → float

Returns

objective_val: float The value of the objective

`evaluate_grad_objective()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

grad_obj: BlockVector The gradient of the objective. This BlockVector has one block for every time block and one block for the coupling variables.

`n_eq_constraints()` → int

Returns

n_eq_constraints: int The number of equality constraints, including the coupling constraints

`n_ineq_constraints()` → int

Returns

n_ineq_constraints: int The number of inequality constraints

`ineq_lb()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

ineq_lb: BlockVector The lower bounds for each inequality constraint. This BlockVector has one block for every time block.

`ineq_ub()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

ineq_lb: BlockVector The lower bounds for each inequality constraint. This BlockVector has one block for every time block.

`init_duals_eq()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

init_duals_eq: BlockVector The initial values for the duals of the equality constraints, including the coupling constraints. This BlockVector has one block for every time block. Each block is itself a BlockVector with 3 blocks. The first block contains the duals of the equality constraints in the corresponding time block. The second block has the duals for the coupling constraints linking the states at the beginning of the time block to the coupling variables between the time block and the previous time block. The third block has the duals for the coupling constraints linking the states at the end of the time block to the coupling variables between the time block and the next time block.

`init_duals_ineq()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

init_duals_ineq: BlockVector The initial values for the duals of the inequality constraints. This BlockVector has one block for every time block.

`set_duals_eq(`*duals_eq: pyomo.contrib.pynumero.sparse.block_vector.BlockVector*`)`

Parameters

duals_eq: BlockVector The values for the duals of the equality constraints, including the coupling constraints. This BlockVector has one block for every time block. Each block is itself a BlockVector with 3 blocks. The first block contains the duals of the equality constraints in the corresponding time block. The second block has the duals for the coupling constraints linking the states at the beginning of the time block to the coupling variables between the time block and the previous time block. The third block has the duals for the coupling constraints linking the states at the end of the time block to the coupling variables between the time block and the next time block.

`set_duals_ineq(`*duals_ineq: pyomo.contrib.pynumero.sparse.block_vector.BlockVector*`)`

Parameters

duals_ineq: BlockVector The values for the duals of the inequality constraints. This Block-Vector has one block for every time block.

`get_duals_eq()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

duals_eq: BlockVector The values for the duals of the equality constraints, including the coupling constraints. This BlockVector has one block for every time block. Each block is itself a BlockVector with 3 blocks. The first block contains the duals of the equality constraints in the corresponding time block. The second block has the duals for the coupling constraints linking the states at the beginning of the time block to the coupling variables between the time block and the previous time block. The third block has the duals for the coupling constraints linking the states at the end of the time block to the coupling variables between the time block and the next time block.

`get_duals_ineq()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

duals_ineq: BlockVector The values for the duals of the inequality constraints. This Block-Vector has one block for every time block.

`evaluate_eq_constraints()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

eq_resid: BlockVector The residuals of the equality constraints, including the coupling constraints. This BlockVector has one block for every time block. Each block is itself a BlockVector with 3 blocks. The first block contains the residuals of the equality constraints in the corresponding time block. The second block has the residuals for the coupling constraints linking the states at the beginning of the time block to the coupling variables between the time block and the previous time block. The third block has the residuals for the coupling constraints linking the states at the end of the time block to the coupling variables between the time block and the next time block.

`evaluate_ineq_constraints()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

ineq_resid: BlockVector The residuals of the inequality constraints. This BlockVector has one block for every time block.

`evaluate_jacobian_eq()` → pyomo.contrib.pynumero.sparse.block_matrix.BlockMatrix

Returns

jac_eq: BlockMatrix The jacobian of the equality constraints. The rows have the same structure as the BlockVector returned from evaluate_eq_constraints. The columns have the same structure as the BlockVector returned from get_primals.

`evaluate_jacobian_ineq()` → pyomo.contrib.pynumero.sparse.block_matrix.BlockMatrix

Returns

jac_ineq: BlockMatrix The jacobian of the inequality constraints. The rows have the same structure as the BlockVector returned from evaluate_ineq_constraints. The columns have the same structure as the BlockVector returned from get_primals.

`load_primals_into_pyomo_model()`
This method takes the current values for the primal variables (those you would get from the get_primals() method), and loads them into the corresponding Pyomo variables.

`pyomo_model`(*ndx: int*) → pyomo.core.base.block._BlockData

Parameters

ndx: int The index of the time block for which the pyomo model should be returned.

Returns

m: _BlockData The pyomo model for the time block corresponding to ndx.

`get_pyomo_variables`(*ndx: int*) → Sequence[pyomo.core.base.var._GeneralVarData]

Parameters

ndx: int The index of the time block for which pyomo variables should be returned

Returns

pyomo_vars: list of _GeneralVarData The pyomo variables in the model for the time block corresponding to ndx

`get_pyomo_constraints`(*ndx: int*) → Sequence[pyomo.core.base.constraint._GeneralConstraintData]

Parameters

ndx: int The index of the time block for which pyomo constraints should be returned

Returns

pyomo_cons: list of _GeneralConstraintData The pyomo constraints in the model for the time block corresponding to ndx

`get_primal_indices`(*ndx: int*, *pyomo_variables: Sequence[pyomo.core.base.var._GeneralVarData]*) → Sequence[int]

Parameters

ndx: int The index of the time block

pyomo_variables: Sequence of _GeneralVarData The pyomo variables for which the indices should be returned

Returns

var_indices: Sequence of int The indices of the corresponding pyomo variables. Note that these indices correspond to the specified time block, not the overall indices. In other words, the indices that are returned are the indices into the block within get_primals corresponding to ndx.

`get_constraint_indices`(*ndx*, *pyomo_constraints*) → Sequence[int]

Parameters

ndx: int The index of the time block

pyomo_constraints: Sequence of _GeneralConstraintData The pyomo constraints for which the indices should be returned

Returns

con_indices: Sequence of int The indices of the corresponding pyomo constraints. Note that these indices correspond to the specified time block, not the overall indices.

4.3.4 MPI Dynamic SC IP Interface

class `MPIDynamicSchurComplementInteriorPointInterface`(*start_t: float*, *end_t: float*, *num_time_blocks: int*, *comm: mpi4py.MPI.Comm*)

Bases: *parapint.interfaces.schur_complement.sc_ip_interface.DynamicSchurComplementInteriorPointInterface*

A class for interfacing with Parapint's interior point algorithm for the parallel solution of dynamic optimization problems using Schur-Complement decomposition. Users should inherit from this class and, at a minimum, implement the *build_model_for_time_block* method (see DynamicSchurComplementInteriorPointInterface.build_model_for_time_block for details).

Parameters

start_t: float The starting time for the dynamic optimization problem

end_t: float The final time for the dynamic optimization problem

num_time_blocks: int The number of time blocks to split the time horizon into for parallel solution. This is typically equal to the number of processes available (i.e., comm.Get_size()).

comm: MPI.Comm The MPI communicator to use. Typically, this is mpi4py.MPI.COMM_WORLD.

`n_primals()` → int

Returns

n_primals: int The number of primal variables

`evaluate_objective()` → float

Returns

objective_val: float The value of the objective

`n_eq_constraints()` → int

Returns

n_eq_constraints: int The number of equality constraints, including the coupling constraints

`n_ineq_constraints()` → int

Returns

n_ineq_constraints: int The number of inequality constraints

`property ownership_map: Dict[int, int]`

Returns

ownership_map: dict This is a map from the time block index to the rank that owns that time block.

`property local_block_indices: Sequence[int]`

Returns

local_block_indices: list The indices of the time blocks owned by the current process.

`abstract build_model_for_time_block`(*ndx: int*, *start_t: float*, *end_t: float*, *add_init_conditions: bool*) → Tuple[pyomo.core.base.block._BlockData, Sequence[pyomo.core.base.var._GeneralVarData], Sequence[pyomo.core.base.var._GeneralVarData]]

This method should be implemented by derived classes. This method should build (and return) the model for the time interval [start_t, end_t] and return a list of states at start_t and a list of states at end_t (in the same order). This method will be called once for each time block. The start_states and end_states returned by this method must be in the same order for every time block.

Parameters

ndx: int The time block index

start_t: float

end_t: float

add_init_conditions: bool This will only be True for time block 0.

Returns

pyomo_model: pyomo.core.base.block.Block The model for the time interval [start_t, end_t].

start_states: Sequence of _GeneralVarData a list of the states at start_t; the order of this list should be the same for every time block

end_states: Sequence of _GeneralVarData a list of the states at end_t; the order of this list should be the same for every time block

`evaluate_eq_constraints()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

eq_resid: BlockVector The residuals of the equality constraints, including the coupling constraints. This BlockVector has one block for every time block. Each block is itself a BlockVector with 3 blocks. The first block contains the residuals of the equality constraints in the corresponding time block. The second block has the residuals for the coupling constraints linking the states at the beginning of the time block to the coupling variables between the time block and the previous time block. The third block has the residuals for the coupling constraints linking the states at the end of the time block to the coupling variables between the time block and the next time block.

`evaluate_grad_objective()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

grad_obj: BlockVector The gradient of the objective. This BlockVector has one block for every time block and one block for the coupling variables.

`evaluate_ineq_constraints()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

ineq_resid: BlockVector The residuals of the inequality constraints. This BlockVector has one block for every time block.

`evaluate_jacobian_eq()` → pyomo.contrib.pynumero.sparse.block_matrix.BlockMatrix

Returns

jac_eq: BlockMatrix The jacobian of the equality constraints. The rows have the same structure as the BlockVector returned from evaluate_eq_constraints. The columns have the same structure as the BlockVector returned from get_primals.

`evaluate_jacobian_ineq()` → pyomo.contrib.pynumero.sparse.block_matrix.BlockMatrix

Returns

jac_ineq: BlockMatrix The jacobian of the inequality constraints. The rows have the same structure as the BlockVector returned from evaluate_ineq_constraints. The columns have the same structure as the BlockVector returned from get_primals.

`get_constraint_indices(`*ndx*, *pyomo_constraints*`)` → Sequence[int]

Parameters

ndx: int The index of the time block

pyomo_constraints: Sequence of _GeneralConstraintData The pyomo constraints for which the indices should be returned

Returns

con_indices: Sequence of int The indices of the corresponding pyomo constraints. Note that these indices correspond to the specified time block, not the overall indices.

`get_duals_eq()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

duals_eq: BlockVector The values for the duals of the equality constraints, including the coupling constraints. This BlockVector has one block for every time block. Each block is itself a BlockVector with 3 blocks. The first block contains the duals of the equality constraints in the corresponding time block. The second block has the duals for the coupling constraints linking the states at the beginning of the time block to the coupling variables between the time block and the previous time block. The third block has the duals for the coupling constraints linking the states at the end of the time block to the coupling variables between the time block and the next time block.

`get_duals_ineq()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

duals_ineq: BlockVector The values for the duals of the inequality constraints. This BlockVector has one block for every time block.

`get_primal_indices`(*ndx: int*, *pyomo_variables: Sequence[pyomo.core.base.var._GeneralVarData]*) → Sequence[int]

Parameters

ndx: int The index of the time block

pyomo_variables: Sequence of _GeneralVarData The pyomo variables for which the indices should be returned

Returns

var_indices: Sequence of int The indices of the corresponding pyomo variables. Note that these indices correspond to the specified time block, not the overall indices. In other words, the indices that are returned are the indices into the block within get_primals corresponding to ndx.

`get_primals()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

primals: BlockVector The values for each primal variable. This BlockVector has one block for every time block and one block for the coupling variables.

`get_pyomo_constraints`(*ndx: int*) → Sequence[pyomo.core.base.constraint._GeneralConstraintData]

Parameters

ndx: int The index of the time block for which pyomo constraints should be returned

Returns

pyomo_cons: list of _GeneralConstraintData The pyomo constraints in the model for the time block corresponding to ndx

`get_pyomo_variables`(*ndx: int*) → Sequence[pyomo.core.base.var._GeneralVarData]

Parameters

ndx: int The index of the time block for which pyomo variables should be returned

Returns

pyomo_vars: list of _GeneralVarData The pyomo variables in the model for the time block corresponding to ndx

`ineq_lb`() → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

ineq_lb: BlockVector The lower bounds for each inequality constraint. This BlockVector has one block for every time block.

`ineq_ub`() → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

ineq_lb: BlockVector The lower bounds for each inequality constraint. This BlockVector has one block for every time block.

`init_duals_eq`() → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

init_duals_eq: BlockVector The initial values for the duals of the equality constraints, including the coupling constraints. This BlockVector has one block for every time block. Each block is itself a BlockVector with 3 blocks. The first block contains the duals of the equality constraints in the corresponding time block. The second block has the duals for the coupling constraints linking the states at the beginning of the time block to the coupling variables between the time block and the previous time block. The third block has the duals for the coupling constraints linking the states at the end of the time block to the coupling variables between the time block and the next time block.

`init_duals_ineq`() → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

init_duals_ineq: BlockVector The initial values for the duals of the inequality constraints. This BlockVector has one block for every time block.

`init_primals`() → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

init_primals: BlockVector The initial values for each primal variable. This BlockVector has one block for every time block and one block for the coupling variables.

`load_primals_into_pyomo_model`()

This method takes the current values for the primal variables (those you would get from the get_primals() method), and loads them into the corresponding Pyomo variables.

`primals_lb()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

primals_lb: BlockVector The lower bounds for each primal variable. This BlockVector has one block for every time block and one block for the coupling variables.

`primals_ub()` → pyomo.contrib.pynumero.sparse.block_vector.BlockVector

Returns

primals_ub: BlockVector The upper bounds for each primal variable. This BlockVector has one block for every time block and one block for the coupling variables.

`pyomo_model(`*ndx: int*`)` → pyomo.core.base.block._BlockData

Parameters

ndx: int The index of the time block for which the pyomo model should be returned.

Returns

m: _BlockData The pyomo model for the time block corresponding to ndx.

`set_duals_eq(`*duals_eq: pyomo.contrib.pynumero.sparse.block_vector.BlockVector*`)`

Parameters

duals_eq: BlockVector The values for the duals of the equality constraints, including the coupling constraints. This BlockVector has one block for every time block. Each block is itself a BlockVector with 3 blocks. The first block contains the duals of the equality constraints in the corresponding time block. The second block has the duals for the coupling constraints linking the states at the beginning of the time block to the coupling variables between the time block and the previous time block. The third block has the duals for the coupling constraints linking the states at the end of the time block to the coupling variables between the time block and the next time block.

`set_duals_ineq(`*duals_ineq: pyomo.contrib.pynumero.sparse.block_vector.BlockVector*`)`

Parameters

duals_ineq: BlockVector The values for the duals of the inequality constraints. This BlockVector has one block for every time block.

`set_primals(`*primals: pyomo.contrib.pynumero.sparse.block_vector.BlockVector*`)`

Set the values of the primal variables for evaluation (i.e., the evaluate_* methods).

Parameters

primals: BlockVector The values for each primal variable. This BlockVector should have one block for every time block and one block for the coupling variables.

CHAPTER FIVE

SANDIA FUNDING STATEMENT

This work was supported by Sandia National Laboratories' Laboratory Directed Research and Development program. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Word2014] Word, D. P., Kang, J., Akesson, J., & Laird, C. D. (2014). Efficient parallel solution of large-scale nonlinear dynamic optimization problems. Computational Optimization and Applications, 59(3), 667-688.

PYTHON MODULE INDEX

INDEX