Using the SCOTCH Graph Partitioning Library

Aleksandar Donev

July 2001

1 Module SCOTCH_Interface

This contains a somewhat unfinished interface to the SCOTCH library of Pellegrini. It follows the same basic design as the interface to CHACO and TAUCS.

This documentation is not finished.

"WEAVE.f90" 1.0.1 =

MODULE SCOTCH_Constants
  USE Precision
  IMPLICIT NONE
  PUBLIC

  INTEGER, PARAMETER :: scotch = iword, SCOTCH_MAX_LEN = 200
  // The scotch integer data-type and maximal length of string needed

  INTEGER, PARAMETER :: SCOTCH_ARCHDIM = 7, SCOTCH_GEOMDIM = 15,
  SCOTCH_GRAPHDIM = 14, SCOTCH_MAPDIM = 32, SCOTCH_ORDERDIM = 9,
  SCOTCH_STRATDIM = 1
  // Sizes of arrays for opaque SCOTCH types

END MODULE SCOTCH_Constants

MODULE SCOTCH_Interface
  USE Precision
  USE SCOTCH_Constants
  USE Error_Handling
  USE System_Monitors
  USE Graph_Algorithms
  USE Network_Data_Types
  IMPLICIT NONE
  PRIVATE

  PUBLIC :: SCOTCH_InitializeMappingOrdering, SCOTCH_ComputeMappingOrdering,
            SCOTCH_DestroyMappingOrdering

1
ML_EXTERNAL :: SCOTCH_graphInit, SCOTCH_graphExit, SCOTCH_graphWrite,
           SCOTCH_graphBuild, SCOTCH_graphMap, SCOTCH_graphOrder, SCOTCH_stratInit,
           SCOTCH_stratExit, SCOTCH_stratMap, SCOTCH_stratOrder, SCOTCH_archInit,
           SCOTCH_archExit, SCOTCH_archAlgorithmic
INTEGER :: SCOTCH_graphInit, SCOTCH_graphBuild, SCOTCH_graphMap, SCOTCH_graphOrder,
           SCOTCH_stratInit, SCOTCH_stratMap, SCOTCH_stratOrder, SCOTCH_archInit,
           SCOTCH_archExit, SCOTCH_archAlgorithmic
PUBLIC :: SCOTCH_Graph_, SCOTCH_Mapping_, SCOTCH_Ordering_, SCOTCH_Mapping_Ordering
 // Public types
TYPE SCOTCH_Graph_
PRIVATE
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_GRAPHDIM) :: graph
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_GRAPHDIM) :: n_vertices, n_edges = 0
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_GRAPHDIM) :: incitab // Tricky in F95!
ENDTYPE
TYPE SCOTCH_Mapping_
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_MAPDIM) :: mapping
 // No need to use it yet
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_StrategyDIM) :: strategy
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_ARCHDIM) :: architecture
 CHARACTER (LEN = SCOTCH_MAX_LEN) :: architecture_string = "" // The name for the algorithmic architecture
 CHARACTER (LEN = SCOTCH_MAX_LEN) :: mapping_string = "" // Strategy string
ENDTYPE
TYPE SCOTCH_Ordering_
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_ORDERDIM) :: ordering
 // Not used at present
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_StrategyDIM) :: strategy
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_ORDERDIM) :: n_blocks = 0 // Number of blocks
 INTEGER (KIND = i_scotch), DIMENSION (SCOTCH_ORDERDIM) :: block_pointers
 // The returned blocking range pointer
 CHARACTER (LEN = SCOTCH_MAX_LEN) :: ordering_string = "" // Ordering strategy string
ENDTYPE
TYPE SCOTCH_Mapping_Ordering
 LOGICAL :: map_graph = T, order_graph = F
 _TYPE (Directed_Graph), POINTER :: graph
 _TYPE (Support_Tree_Mapping), POINTER :: ST_mapping
 _TYPE (Fill_Reducing_Ordering), POINTER :: graph_ordering
 _TYPE (SCOTCH_Graph_) :: scotch_graph
 _TYPE (SCOTCH_Mapping_) :: scotch_mapping
 _TYPE (SCOTCH_Ordering_) :: scotch_ordering
 LOGICAL :: allocated_mapping = F, allocated_ordering = F
ENDTYPE
CONTAINS
 (SCOTCH_InitializeMappingOrdering 1.2.1)
 (SCOTCH_ComputeMappingOrdering 1.3.1)
 (SCOTCH_DestroyMappingOrdering 1.4.1)
END_MODULE SCOTCH_Interface
1.1 Auxillary macros

"WEAVE.f90" 1.1.1 ≡

@m _mapping mapping_ordering
@m _map_graph _mapping % map_graph
@m _order_graph _mapping % order_graph
@m _graph _mapping % graph
@m _graph_ordering _mapping % graph_ordering
@m _scotch_graph _mapping % scotch_graph
@m _scotch_mapping _mapping % scotch_mapping
@m _scotch_ordering _mapping % scotch_ordering
@m _ST_mapping _mapping % ST_mapping
@m _arcs_weights _ST_mapping % arcs_weights
@m _nodes_mapping _ST_mapping % nodes_mapping
@m _n_blocks scotch_ordering % n_blocks
@m _block_pointers scotch_ordering % block_pointers
@m _nodes_renumbering _graph_ordering % nodes_renumbering
@m _nodes_reordering _graph_ordering % nodes_reordering
@m _allocated_mapping _mapping % allocated_mapping
@m _allocated_ordering _mapping % allocated_ordering
@m _architecture_string scotch_mapping % architecture_string
@m _mapping_string scotch_mapping % mapping_string
@m _ordering_string scotch_ordering % ordering_string
@m _n_edges scotch_graph % n_edges
@m _n_vertices scotch_graph % n_vertices
@m _velotab scotch_graph % velotab
@m _edgetab scotch_graph % edgetab
@m _vertab scotch_graph % vertab
@m _edlotab scotch_graph % edlotab
@m _incitab scotch_graph % incitab

1.2 Initialization/Format conversion

This routine will convert an SSCNO graph into a SCOTCH graph, and can be used to reorder or partition the graph. The routine needs to be called only once per graph (even if numerical values change):

<SCOTCH.InitializeMappingOrdering 1.2.1> ≡

SUBROUTINE SCOTCH.InitializeMappingOrdering(mapping_ordering)
  IMPLICIT NONE
  TYPE (SCOTCH_Mapping_Ordering), INTENT (INOUT), TARGET :: mapping_ordering
_TYPE (Directed_Graph),  POINTER :: graph
INTEGER :: alloc_status, result
INTEGER (KIND = i_scotch) :: n_vertices, n_edges
INTEGER (KIND = i_wp) :: n_special_nodes, n_nodes, n_special_arcs, n_arcs
LOGICAL :: valid_graph

graph ⇒ _graph
valid_graph = FALSE
IF (~ASSOCIATED(graph)) THEN
  valid_graph = FALSE
ELSE
  IF (~ASSOCIATED(_heads_tails)) valid_graph = FALSE
END IF
IF (~valid_graph) THEN
  CALL CriticalError(message = "Incomplete or no graph passed to SCOTCH library", caller = "SCOTCH_InitializeMappingOrdering")
  RETURN
END IF

n_special_nodes = n_special_nodes
n_nodes = n_nodes
n_special_arcs = n_special_arcs
n_arcs = n_arcs

n_vertices = n_nodes + n_special_nodes + 1
n_edges = 2 * (n_arcs + n_special_arcs + 1)  // May not be true of there are self-loops!

result = SCOTCH_graphInit(_scotch_graph % graph)
result = SCOTCH_archInit(_scotch_mapping % architecture)
IF (~map_graph) result = SCOTCH_stratInit(_scotch_mapping % strategy)
IF (~order_graph) result = SCOTCH_stratInit(_scotch_ordering % strategy)

_AllocateArray(_vertab, 0, n_vertices + 1, _i_scotch, NON_NULL, "SCOTCH_InitializeMappingOrdering")
_AllocateArray(_edgetab, 0, n_edges, _i_scotch, NON_NULL, "SCOTCH_InitializeMappingOrdering")
_AllocateArray(_incitab, 0, n_edges, _i_scotch, NON_NULL, "SCOTCH_InitializeMappingOrdering")
  // The major step is converting my data-structures into one suitable for SCOTCH:
  CALL CreateAdjacencyArrays (node_offset = n_special_nodes, arc_offset = n_special_arcs,
     _heads_tails = _heads_tails, _neighbours = _edgetab, _my_neighbours = _vertab,
     _incident_arcs = _incitab, _n_edges = _n_edges, _include_loops = FALSE)

n_vertices = n_vertices  // Save this number
n_edges = n_edges  // Save this number

IF (~map_graph) THEN
  IF (~ASSOCIATED(_ST_mapping)) THEN
    ALLOCATE (_ST_mapping, STAT = alloc_status)
    _allocated_mapping = FALSE
  END IF
  _AllocateArray(_adjlotab, 0, n_edges, _i_scotch, NON_NULL, "SCOTCH_InitializeMappingOrdering")
  _AllocateArcArray(_arcs_weights, 1.0_wp, NON_ASSOCIATED, "SCOTCH_InitializeMappingOrdering")
  _AllocateNodalArray(_nodes_mapping, _i_wp, ASSOCIATED, "SCOTCH_InitializeMappingOrdering")
END IF

IF (~order_graph) THEN  // A node fill-reducing ordering will be computed

4
IF (~ASSOCIATED(graph_ordering)) THEN
   ALLOCATE (graph_ordering, STAT = alloc_status)
   allocated_ordering = T
END IF

AllocateArray(_block_pointers, 0, n_vertices + 1, l_i_scatch, ~NON_NULL,
   "SCOTCH.InitializeMappingOrdering")
AllocateNodeArray(_nodes_reordering, l_i_wp, ~NON_NULL, "SCOTCH.InitializeMappingOrdering")
AllocateNodeArray(_nodes_renumbering, l_i_wp, ~NON_NULL, "SCOTCH.InitializeMappingOrdering")
END IF

END SUBROUTINE SCOTCH.InitializeMappingOrdering

This code is used in section 1.0.1.

1.3 Graph partitioning and fill-reducing reordering

This routine will call SCOTCH and perform either a mapping (partitioning) of the graph or compute a fill-reducing reordering:

⟨SCOTCH_ComputeMappingOrdering 1.3.1⟩ ≡

SUBROUTINE SCOTCH_ComputeMappingOrdering (mapping_ordering , mapping_offset)
IMPLICIT NONE
_TYPE(SCOTCH_Mapping_Ordering), INTENT (INOUT), TARGET :: mapping_ordering
INTEGER (KIND = i_wp), INTENT (IN), OPTIONAL :: mapping_offset
   // Shift in numbering for mapping
_TYPE(Directed_Graph), POINTER :: graph
INTEGER (KIND = i_wp) :: arc
INTEGER (KIND = i_scatch) :: edge, max_scatch_weight
REAL (KIND = r_wp) :: max_weight
INTEGER :: alloc_status, result

graph ⇒ _graph

MapGraph: IF (_map_graph) THEN
   result = SCOTCH_archAlgorithmic(_scotch_mapping % architecture, _architecture_string || CHAR(0))
   result = SCOTCH_stratMap(_scotch_mapping % strategy, _mapping_string || CHAR(0))
   max_weight = MAXVAL(_arc_weights)  // This needs to be improved—maybe logarithmic scaling!
   max_scatch_weight = HUGE(l_i_scatch) / 10/ (n_vertices + n_edges)
   DO edge = 1, n_edges  // Normalization of edge weights
      arc = jncitab_edge
      edotab_edge = INT(REAL(max_scatch_weight) * _arc_weights arc / max_weight, i_scatch) + l_i_scatch
   END DO

   * This is tricky in F95, and I am working on a proposal to change this in F2K. Namely, we need to pass a reference to the _tab arrays here that denotes their memory location. I do that here by passing the first address of the array as the first element. This may not be portable. */
result = SCOTCH_graphBuild(scotch_graph % graph, _VAL(int(ν vertices, i_scotch)), _verttab1,
    _VAL(0), _VAL(0), _VAL(int(ν edges, i_scotch)), _edgetab1, _eddotab1, _VAL(1, i_scotch), 1, i_scotch)
call SCOTCH_graphWrite(scotch_graph % graph, "graph.out" || CHAR(0))
result = SCOTCH_graphMap(scotch_graph % graph, _scotch_mapping % strategy,
    _scotch_mapping % architecture, _nodes_mapping % _n_special_nodes)
if (present(mapping_offset)) then
    _nodes_mapping = _nodes_mapping + mapping_offset
end if
end if MapGraph

OrderGraph: if (_order_graph) then
    result = SCOTCH_stratOrder(scotch_ordering % strategy, ordering_string || CHAR(0))
result = SCOTCH_graphOrder(scotch_graph % graph, _scotch_ordering % strategy, _n_blocks,
    _block_pointer1, _nodes_renumbering % _n_special_nodes, _nodes_reordering % _n_special_nodes)
    _nodes_reordering = _nodes_reordering - _n_special_nodes - 1  // Correct the index base
    _nodes_renumbering = _nodes_renumbering - _n_special_nodes - 1  // Renumber
end if OrderGraph
end subroutine SCOTCH_ComputeMappingOrdering

This code is used in section 1.0.1.

1.4 Destruction/release of data-structures

This routine should be called at the end to release all the allocated data by the other routines in this interface to SCOTCH.

〈SCOTCH_DestroyMappingOrdering 1.4.1〉 ≡

subroutine SCOTCH_DestroyMappingOrdering(mapping_ordering)
    implicit none
    _type (SCOTCH_Mapping_Ordering), intent (inout), target :: mapping_ordering
    integer :: alloc_status, result
    call SCOTCH_graphExit(scotch_graph % graph)
call SCOTCH_archExit(scotch_mapping % architecture)
if (_map_graph) call SCOTCH_stratExit(scotch_mapping % strategy)
if (_order_graph) call SCOTCH_stratExit(scotch_ordering % strategy)
if (_allocated_mapping) then
    DeallocateArray(_arcs_weights, 1.0_wp, associated)
    DeallocateArray(_nodes_mapping, 1.0_wp, associated)
deallocate (_ST_mapping, stat = alloc_status)
end if
if (_allocated_ordering) then
_DealocateArray(nodes_reordering, iWp, ASSOCIATED)_
_DeallocatesArray(nodes_renumbering, iWp, ASSOCIATED)_
DEALLOCATE (_graph_ordering, stat = alloc_status)_
END IF

_DeallocatesArray(_verttab, iWscotch, NULL)_
_DeallocatesArray(_edgetab, iWscotch, NULL)_
_DeallocatesArray(_incitab, iWscotch, NULL)_
_DeallocatesArray(_edlotab, iWscotch, NULL)_
_DeallocatesArray(_block_pointers, iWscotch, NULL)_

END SUBROUTINE _SCOTCH_DestroyMappingOrdering_

This code is used in section 1.0.1.
@m CASE_TYPE TYPE
@m _TYPE TYPE
@m NULL > NULL()
@m PRIVATE PRIVATE
@m _SIZE(array, _kind,...)
  $ELSE (#O, 0, INT(SIZE(array), KIND=_kind), INT(SIZE(array,#), KIND=_kind))
@m _MAXLOC(array, _kind,...)
  $ELSE (#O, 0, INT(MAXLOC(array), KIND=_kind), INT(MAXLOC(array,#), KIND=_kind))
@m _MINLOC(array, _kind,...)
  $ELSE (#O, 0, INT(MINLOC(array), KIND=_kind), INT(MINLOC(array,#), KIND=_kind))
@m _BOUND(array, _kind,...) $ELSE (#O, 0, INT(_BOUND(array, DIM=1), KIND=_kind),
  INT(_BOUND(array, #), KIND=_kind))
@m _UBOUND(array, _kind,...) $ELSE (#O, 0, INT(_BOUND(array, DIM=1), KIND=_kind),
  INT(_BOUND(array, #), KIND=_kind))
@m _GenericInterface(generic_name,...)
  INTERFACE generic_name MODULE PROCEDURE #.
  END INTERFACE generic_name
@m _Declare_i_word...
  INTEGER :: #.
@m _Declare_i_wp...
  INTEGER (KIND = i_wp) :: #.
@m _Declare_r_wp...
  REAL (KIND = r_wp) :: #.
@m _Declare_r_sp...
  REAL (KIND = r_sp) :: #.
@m _Declare_r_dp...
  REAL (KIND = r_dp) :: #.
@m _FullExtent(_rank) :$DO (DIM, 2, _rank) { : : }
@m _VarSequence(_variable, _start, _end)
  _variable##_start$DO (DIM, $SEVAL (_start + 1), _end) { , _variable@&DIM }
@m _NestedLoopStart(_variable, array, _rank, _kind)
  $DO (_DIM, _rank, 1, -1) { DO _variable@&DIM = _BOUND(_array, _kind, DIM = _DIM),
    _BOUND(_array, _kind, DIM = _DIM) }
@m _NestedLoopEnd(_rank) $DO (_DIM, 1, _rank) { END DO }
@m _Dummy(...)
@m _DisplayArray(message, array)
  IF (SIZE(array) <= 20) THEN
    WRITE (message_PRINT_UNIT, "(A)" ) message
    WRITE (message_PRINT_UNIT, "((2005,2))" ) array
  END IF