Data Reordering based on Space-Filling Curves

Aleksandar Donev

February 2000

Contents

1 Module Space_Filling_Curves  
   1.1 SFC Data Reordering  

2 Formatting rules for HPF/F90 files
1 Module Space_Filling_Curves

This module provides an interface to some of the routines in the data and computation reordering library to be found at the ASCI software page:

http:\\www.cs.rice.edu\~system\asci\software

These routines can compute the index of a given point along a space filling curve and they use some efficient bit-twisting operations to do this. At present the Hilber-Peano and Morton SFCs are supported. I modified the source code of these routines slightly to provide a uniform Fortran-like interface. The interfaces for these two new routines, fhilbert_c2i and fmorton_c2i (c2i stands for coordinate-to-index) are given below.

"WEAVE.f90" 1.0.1 ≡
@m INTERFACE_c2i(function_name)
   FUNCTION function_name(n_dim, n_bits, coords) RESULT(SFC_index)
   USE Precision
   INTEGER, INTENT (IN) :: n_dim, n_bits       // The point number
   INTEGER, DIMENSION (n_dim), INTENT (IN) :: coords
   INTEGER (KIND = i_dp) :: SFC_index       // Double precision bitmask
END FUNCTION function_name

"WEAVE.f90" 1.0.2 ≡

MODULE Space_Filling_Curves
   USE Precision
   USE Error_Handling
   USE System_Monitors
   USE Sorting_Ranking
   PUBLIC :: SFCOrder
PRIVATE
   INTERFACE
      INTERFACE_c2i(fhilbert_c2i)
      INTERFACE_c2i(fmorton_c2i)
   END INTERFACE
   ⟨GenericInterfaces 1.1.2⟩
CONTAINS
   ⟨SFCOrder 1.1.1⟩
END MODULE Space_Filling_Curves
1.1 SFC Data Reordering

The routine \texttt{SFCOrder} is the main routine in this module. It calculates the ordering of a collection of points along a space filling curve. This provides an ordering for the points, in our case the nodes of the network, which tends to maximize data spatial locality and thus dramatically improve cache performance of the code. This routine does not do the actual reordering but rather returns the permutation \texttt{SFC\_order} that reorders the nodal arrays. One thing to notice is that the routine can accept coordinates of different types and kinds, overloaded with a generic interface. The user can provide lower and upper bounds for these coordinates in \texttt{coords\_min} and \texttt{coords\_max}, or the routine will calculate these.

One would also like to perform computation reordering to increase temporal data locality in the cache. There is no truly great way to do this in general, but for lattice-like networks simply sorting the arc arrays according to the heads and tails arrays (in lexicographical order) is a good choice. This transformation is left to the user, but an example of usage is in the module \texttt{Network\_Geometry}.

```
"WEAVE.f90" 1.1 ≡
@m _SFCOrder(kind, kind)
SUBROUTINE SFCOrder(kind, points\_coords, SFC\_order, SFC, coords\_min, coords\_max)
IMPLICIT NONE
    \_\_kind(kind \_= kind), DIMENSION (\_, \_), INTENT (IN) :: points\_coords
    // Coordinates of the points
    INTEGER, DIMENSION (\_, \_), INTENT (OUT) :: SFC\_order    // SFC ordering of the points
    CHARACTER (LEN = *), INTENT (IN), OPTIONAL :: SFC
    // Either Hilbert or Morton
    \_\_kind(kind \_= kind), DIMENSION (\_, \_), INTENT (IN), OPTIONAL :: coords\_min, coords\_max
    // Optional bounds for the coordinates
    INTEGER (kind = i64), DIMENSION (;), ALLOCATABLE :: SFC\_keys
    // The indices along the SFC curve
    INTEGER (kind = i64) :: max\_key    // Maximal expected key along the SFC curve
    INTEGER :: n\_points, n\_dim, n\_bits    // Counters
    INTEGER :: point, dim, alloc\_status    // Temporaries
    INTEGER (kind = i64) :: max\_int\_coord    // Maximum allowed integer coordinate
    INTEGER, DIMENSION (SIZE(points\_coords, DIM = 1)) :: i\_point\_coords    // Integer coordinates
    \_\_kind(kind \_= kind), DIMENSION (SIZE(points\_coords, DIM = 1)) :: coords\_lb, coords\_ub,
    coords\_range    // Coordinate bounds and ranges
    REAL :: scale\_factor    // Scale for the coordinates to integer numbers
    LOGICAL :: Hilbert\_SFC    // An indicator
n\_dim = SIZE(points\_coords, DIM = 1)    // Number of spatial dimensions
n\_points = SIZE(points\_coords, DIM = 2)    // Number of points
Hilbert\_SFC = \_\_    // Use Hilbert curve by default
IF (PRESENT(SFC)) THEN
    IF ((SFC, \_1 \_= 'M')) | (SFC, \_1 \_= 'm')) Hilbert\_SFC = \_\_    // Use Morton
END IF
IF (PRESENT(coords\_min)) THEN    // Calculate coordinate lower bound
    coords\_lb = coords\_min
ELSE
    coords\_lb = MINVAL(points\_coords, DIM = 2)
END IF
IF (PRESENT(coords\_max)) THEN    // Upper bounds
    coords\_ub = coords\_max
```

else
    coords_ab = MAXVAL(points.coords, dim = 2)
end if

coords_range = coords_ab - coords_lb    // Range of coordinates

allocate (SFC_keys(n_points), stat = alloc_status)    // Allocate the key array
call RecordAllocation(n_elements = n_points, mold = 1_i_dp, caller = "SFCOrder",
                       alloc_status = alloc_status)

n_bits = MIN(INT(BIT_SIZE(1_i_dp) - 1) / n_dim, 31)
    // The key has a total of 63 available bits (excluding the sign bit) which are divided among
    // n_dim coordinates with at most 31 available bits
max_int_coord = IBITS(HUGE(1_i_dp), 0, n_bits)    // All bits from 0 to n_bits are 1
scale_factor = REAL(max_int_coord) / REAL(MAXVAL(coords_range))
    // The scale factor—all axes must be scaled the same

do point = 1, n_points    // Now calculate the point indices along the SFC curve
    i_point_coords = INT(scale_factor * REAL(points.coords(i_dim, point) - coords_lb))
    // Normalize the coordinates to large integers for best results
    if (Hilbert_SFC) then    // Use Hilbert SFC
        SFC_keys_point = fhilbert_c2i(n_dim, n_bits, i_point_coords)
    else    // Use Morton
        SFC_keys_point = fmorton_c2i(n_dim, n_bits, i_point_coords)
    end if
end do

max_key = 2_i_dp * INT(n_bits * n_dim - 1)
call QuickRank(array = SFC_keys, permutation = SFC_order, pivot_selection = 'U',
               mean_value = max_key, standard_deviation = INT(max_key / SQRT(3.0), KIND = i_dp))
    // Rank by key value—presumably uniformly distributed
call RecordAllocation(n_elements = SIZE(SFC_keys), mold = 1_i_dp)
deallocate (SFC_keys)    // Deallocate the key array
end subroutine    // SFCOrder

Here are specific instances of the routine SFCOrder for single and double precision integer and real coordinates:

\[ \text{SFCOrder} \]
\begin{align*}
\text{SFCOrder}(\text{INTEGER, } i_sp) \\
\text{SFCOrder}(\text{INTEGER, } i_dp) \\
\text{SFCOrder}(\text{REAL, } r_sp) \\
\text{SFCOrder}(\text{REAL, } r_dp)
\end{align*}

This code is used in section 1.0.2.
And the generic interface overloading:

\[
\begin{align*}
\langle \text{GenericInterfaces 1.1.2} \rangle & \equiv \\
\text{INTERFACE } & \text{SFCOrder} \\
\text{MODULE PROCEDURE } & \text{SFCOrder}_i^{sp} \\
\text{MODULE PROCEDURE } & \text{SFCOrder}_i^{dp} \\
\text{MODULE PROCEDURE } & \text{SFCOrder}_j^{sp} \\
\text{MODULE PROCEDURE } & \text{SFCOrder}_j^{dp} \\
\text{END INTERFACE}
\end{align*}
\]

This code is used in section 1.0.2.
2 Formatting rules for HPF/F90 files

These are just same auxiliary formatting rules and useful macros I use from time to time.

\@m \_GenericInterface(generic_name,...)
  \_Interface generic_name
  \_Module Procedure #.
  \_End Interface generic_name
\@m \_Declare\_Word(...)
  \_Integer :: #.
\@m \_Declare\_Lwp(...)
  \_Integer (kind = i\_wp) :: #.
\@m \_Declare\_Rwp(...)
  \_Real (kind = r\_wp) :: #.
\@m \_Declare\_Sp(...)
  \_Real (kind = r\_sp) :: #.
\@m \_Declare\_Dp(...)
  \_Real (kind = r\_dp) :: #.
\@m \_FullExtent(rank) ::do (dim, 2, rank) { , }
\@m \_VarSequence(variable, start, end)
  variable##start##do (dim, \$$eval(start + 1), end) { , variable@&dim }
\@m \_NestedLoopStart(variable, array, rank)
  do (dim, rank, 1, -1) { do variable@&dim = lbound(array, dim), ubound(array, dim) }
\@m \_NestedLoopEnd(rank) $do (dim, 1, rank) { end do }
\@m \_Dummy(...)
\@m \_DisplayArray(message, array)
  if (size(array) \leq 20) then
    write(message\_print\_unit, "(a)"") message
    write(message\_print\_unit, "(2055,2)") array
  end if