

“nplc_tsc”: Implementation of HPF Array Combining Scatter Functions by using Two-Stage Algorithm

Seungjo Bae and Sanjay Ranka

1 Introduction

In High Performance Fortran (HPF) *array combining scatter functions* are generalized array reduction functions. Elements in a subset of a source array are combined and the resulting value is sent to a position in a resulting array. At this time each subset is non-overlapping [3, 4]. Twelve array combining scatter functions are defined in High Performance Fortran (HPF) and have the form `XXX_SCATTER`(ARRAY, BASE, `INDX`₀, . . . , `INDX`_{*r*'-1}, MASK) where XXX is one of the reduction operations defined in HPF, and *r*' is the rank of a base array [3, 4]. Arguments are explained in Table 1. Each array combining scatter function returns a resulting array that is conformable with the base array and has the same type as the source array.

ARRAY	Source array of rank <i>r</i>
MASK	Logical mask array conformable with the source array
BASE	Base array of rank <i>r</i> ', which has the same type as the source array
INDX_{<i>i</i>}	Index array conformable with the source array where $0 \leq i < r'$

Table 1: Description of arguments used in array combining scatter functions

The array combining scatter function can be defined in terms of *random access write* (RAW) [5]. In random access write (RAW) each element *i* ($0 \leq i < N$) may need to write (or send) a datum, *D*(*i*), to another element [5]. At this time *I*(*i*) indicates a pointer such that element *i* needs to write a datum, *D*(*i*), to an element, *I*(*i*). Thus parallel execution of the do-loop,

$$do (i = 0 : n - 1) A(I(i)) = D(i),$$

results in RAW [6].

2 Implementation

The function, “`nplc_tsc`”, has been implemented for HPF array combining scatter functions. In the current implementation, we used the extended two-stage algorithm [1, 2], which was based on Shankar and Ranka’s two-stage algorithm for random access write [6]. For machine-independent implementation, we used only MPI routines, but it can also be optimized on the CM-5.

In the two-stage algorithm, L elements located in each processor are initially partitioned into equal-sized K buckets (bucket size = L'/K) where L' is the size of a local target array and $1 \leq K \leq L$. The basic idea is that, during random access write (RAW), these buckets are dynamically stretched or shrunk to reduce communication overhead. In addition, we implemented *partial local combining scheme* in order to reduce outgoing traffic at source processors. In the partial local combining scheme (PLC) we first obtain information about the existence of hot pointers and the locations of referred hot spots. We then perform local combining for a specific part of the source elements based on such information. For more details, readers are referred to [1, 2].

In function “`nplc_tsc`”, we may either perform partial local combining scheme or non-local combining. Thus, the first part of the function name, “`nplc`”, means non- or partial local combining while “`tsc`” represents two-stage communication-based algorithm.

2.1 Interface

The description of function “`nplc_tsc`” is shown in Figure 1. Each argument is defined as follows:

- `MPI_Comm WORLD_input` : MPI communicator including all source and target processors. Note that we assume the number of source processors is equal to that of target processors.
- `void *source_array` : Pointer to the source array which is the same as input array `ARRAY`.
- `int source_lidx[]` : Local offset vector to `source_array`. Note that the elements of the source array are not always contiguous in memory, so the `source_lidx[]` vector is required. Actually, we assume only source elements with *true* mask values participate in the array combining scatter operation.
- `int NP` : The size of Local offset vector, that is, the total number of local source elements on each source processor. Note that each source processor may have different values of `NP`.
- `int s_N_all_input` : Total number of source elements. If source processor P_i has NP_i source elements, `s_N_all_input` = $\sum_i NP_i$.

```
void nplc_tsc(
    MPI_Comm WORLD_input,
    void *source_array,
    int source_lidx[],
    int NP,
    int s_N_all_input,
    int index_lidx[],
    int index_pnum[],
    void *target_array,
    int MP,
    Data_Types data_type_input,
    Operator_Types operator_type_input
)

    /***** Optional argument *****/
    int K_input,
    int PLC_input,
    double local_dense_factor_input,
    int algorithm
    /***** *****/
```

Figure 1: Description of function “nplc_tsc”

- `int index_lidx[]` : Note that on each source processor, r' local index arrays need to be transformed to two vectors, `index_lidx` and `index_pnum`. That is, for each local source element (say s), r' indices need to be transformed to (1) a target processor number, P_t , where the source element, s , should be sent and (2) a local index on target processor P_t , i , such that source element s needs to contribute its value to the i^{th} local target element on target processor P_t . Thus, all local indices are stored in vector `index_lidx`.
- `int index_pnum[]` : All target processor numbers are stored in vector `index_pnum[]`.
- `void *target_array` : Resulting array which is conformable with base array `BASE` and has the same data type as the `source_array`. We assume `target_array` is initialized with the base array.
- `int MP` : The total number of local target elements on each target processor. We assume `MP` is uniform, that is all target processors have the same numbers of local target elements.
- `Data_Types data_type_input` : Data type of the source and target elements. In the current version, we support the following five data types
 1. `TYPE_int` : Integer numbers
 2. `TYPE_float` : Floating-point real numbers
 3. `TYPE_double` : Double floating-point real numbers
 4. `TYPE_logic` : Logical values (true or false)
 5. `TYPE_complex` : Complex numbers
- `Operator_Types operator_type_input` : Table 2 shows the available operations in each data types.

In addition, we define four optional arguments:

- `K_input` : Number of buckets in each target processor
- `PLC_input` : If *true*, perform partial local combining
- `local_dense_factor_input` : Criteria for deciding locally dense buckets only when partial local combining is performed. If $\lceil \text{MP}/\text{K_input} \rceil \times \text{local_dense_factor_input} >$ number of writes to bucket i , we perform local combining for bucket i in the partial local combining scheme.
- `algorithm` : Selection of schemes for many-to-many personalized communication. Currently, we have two communication schemes: (1) linear permutation (default) and (2) unstructured communication using nonblocking functions.

Operation	Int	Float	Double	Logic	Complex
COPY	Y	Y	Y	Y	Y
ALL	.	.	.	Y	.
ANY	.	.	.	Y	.
PARITY	.	.	.	Y	.
IALL	Y
IANY	Y
IPARITY	Y
MAX	Y	Y	Y	.	.
MIN	Y	Y	Y	.	.
SUM	Y	Y	Y	.	Y
PRODUCT	Y	Y	Y	.	Y
COUNT	.	.	.	Y	.

“Y” Supported, “.”: Not supported

Table 2: Operations supported in function “`nplc_tsc`”

These four arguments are user-provided parameters. In the current version we use default values for these optional arguments. In addition, Figure 2 lists all files required for using function “`nplc_tsc`.” Also, examples are presented in Appendix A.

References

- [1] Seungjo Bae. *Runtime Support for Unstructured Data Accesses on Coarse-Grained Distributed Memory Parallel Machines*. PhD thesis, Syracuse University, May 1997.
- [2] Seungjo Bae, Khaled A. Alsabti, and Sanjay Ranka. Array Combining Scatter Functions on Coarse-Grained, Distributed-Memory Parallel Machines. Will be submitted to IPPS’98, 1997.
- [3] C. H. Koelbel et al. *The High Performance Fortran Handbook*. MIT Press, 1994.
- [4] High Performance Fortran Forum. *High Performance Fortran Language Specification: Version 1.1*, November 1994.
- [5] D. Nassimi and S. Sahni. Data Broadcasting in SIMD Computers. *IEEE Transactions on Computers*, C-30(2):101–107, 1981.

```

raw_global.h      : global variables
raw_type.h       : definition for types and combining operator
nplc-raw.c       : main routine "void nplc_tsc()"
nplc-raw-int.c   : array combining scatter for integer data type
nplc-raw-logic.c : array combining scatter for logic data type
nplc-raw-float.c : array combining scatter for float data type
nplc-raw-double.c : array combining scatter for double data type
nplc-raw-complex.c : array combining scatter for complex data type
comm-raw.c       : communication routines

```

Figure 2: Required files (programs written in C) for function “nplc_tsc”

- [6] Ravi V. Shankar and Sanjay Ranka. Random Data Accesses on a Coarse-Grained Parallel Machine II. One-to-Many and Many-to-one Mappings. To appear in *Journal of Parallel and Distributed Computing*, 1995.

A Example

In this section we present a simple example.

A.1 EXAMPLE: Sample Input data

- Array Rank = 0

- Source and Index Arrays

```

Global Size = 60 (s_N_all_input)
Processors  = 5
Block Size  = 7
Local Size (NP)
  [PN: 0]  14  ---+
  [PN: 1]  14   |
  [PN: 2]  14   |
  [PN: 3]  11   |
  [PN: 4]   7  ---+

```

- Base and Target Arrays

Global Size = 50
Processors = 5
Block Size = 5
Local Size (MP) = 10

- Data_Types data_type_input = INT
Operator_Types operator_type_input = SUM

- Assume that the target array is initialized with the base array or with identity values before we call routine "nplc_tsc()".

That is, for all i, target_array[i] = base_array[i]
or
target_array[i] = identity-values

A.2 EXAMPLE: Sample Global Arrays

***** source_array *****

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

***** source_lidx *****

0	1	2	3	4	5	6
0	1	2	3	4	5	6
0	1	2	3	4	5	6
0	1	2	3	4	5	6
0	1	2	3	4	5	6
7	8	9	10	11	12	13
7	8	9	10	11	12	13
7	8	9	10	11	12	13
7	8	9	10	11	12	13
7	8	9	10	11	12	13

***** index_lidx *****

4	9	1	1	1	5	2
8	1	9	3	7	9	2
2	1	7	7	5	1	0
6	3	3	9	3	5	0
0	3	1	3	1	9	8
3	7	8	4	8	4	4
7	3	2	6	2	4	6
1	7	6	6	6	4	6
5	3	0	8			

***** index_pnum *****

4	4	4	4	4	4	4
4	4	4	4	4	4	4
4	4	4	4	4	4	4
4	4	4	4	4	4	4
4	4	4	4	4	4	4
4	4	4	4	4	4	4
4	4	4	4	4	4	4
4	4	4	4	4	4	4
4	4	4	4	4	4	4
4	4	4	4			

***** TARGET ARRAY Before Combining *****

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0		

***** TARGET ARRAY After Combining *****

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
4	9	5	9	6	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	4	7	6	5	5

A.3 EXAMPLE: Sample Interface

- On processor 0

```
WORLD                = MPI communicator which include all source
                    and target processors.

source_array         = 1 1 1 1 1 1 1 1 1 1 1 1 1 1
source_lidx          = 0 1 2 3 4 5 6 7 8 9 10 11 12 13
NP                   = 14
s_N_all_input        = 60

index_lidx           = 4 9 1 1 1 5 2 3 7 8 4 8 4 4
index_pnum           = 4 4 4 4 4 4 4 4 4 4 4 4 4 4

target_array         = 0 0 0 0 0 0 0 0 0 0 0
MP                   = 10

data_type            = TYPE_int
op_type              = SUM
```