Gather-scatter library in Nek5000

Documentation of the gs library developed by James Lottes

Nicolas Offermans

November 21, 2017

1 Introduction

The gather-scatter operation within Nek5000, using global communication between processes, has been implemented in a convenient library by James Lottes (some of the issues regarding the implementation are discussed in [2]). In this document, an overview of the capacities of this tool and a description on how to use it are presented. The explanations that follow come mostly from the comments in the gs code.

1.1 Basics of gather-scatter

When Nek5000 solves its equations on a global domain Ω made of E elements, denoted Ω^e $(e = 1, ..., E)$, nodes may be numbered globally on Ω or locally on Ω^e. Following the notation and an example from [1] (see section 4.5.1), if polynomial order N is considered, let $\bar{\mathcal{N}} = (N+1)^d E$ denote the number of distinct nodes in Ω and $\underline{u} \in \mathbb{R}^{\bar{\mathcal{N}}}$ denote the associated vector of nodal values. Then, let $\underline{u}^e \in \mathbb{R}^{(N+1)^d}$ denote the vector of local basis coefficient associated with Ω^e. The collection of all the vectors $\underline{u}^e$ is denoted $\underline{u}_L$. For example, if we refer to figure (1), then the vectors $\underline{u}$ and $\underline{u}_L$ are given by:

$$\begin{aligned} \underline{u} &= (u_1, u_2, u_3, ..., u_{15})^T, \\ \underline{u}_L &= (u^1_{1,1}, u^1_{1,2}, ..., u^1_{3,3}, u^2_{1,1}, u^2_{1,2}, ..., u^2_{3,3})^T. \end{aligned}$$

We denote Q the Boolean connectivity matrix that maps $\underline{u}$ to $\underline{u}_L$. The operation

$$\underline{u}_L = Q\underline{u} \tag{1}$$

is called a scatter from the global to the local vector. The action of Q is to copy the global values into the local vector. On the other hand, the operation

$$\underline{v} = Q^T \underline{u_L} \tag{2}$$

is called a gather. The action of Q^T is to sum entries from corresponding nodes.

Typically, a gather is usually very often followed directly by a scatter. The combined gather-scatter operation is the result of the action of QQ^T.

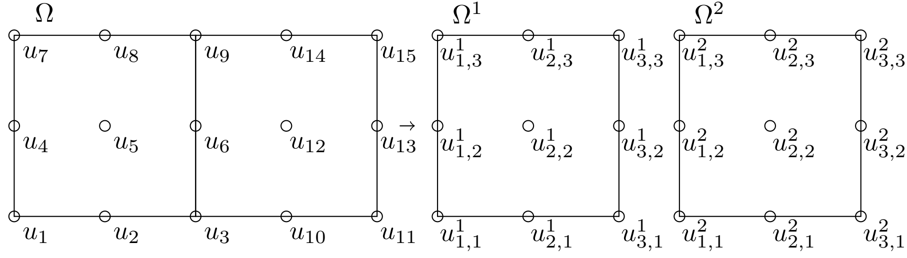

Figure 1: Example of a mapping between local and global numberings for a domain made of two spectral elements.

2 Practical use of the library

In order to use the gather-scatter library, it is first necessary to initialize and setup a handle based on a local-to-global mapping. Then, this handle can be used for any gather-scatter operation on any corresponding data. In fact, the way the library is built allows the user to perform more than a simple gather-scatter operation. Depending on how the mapping is built, is is possible to perform either a gather or a scatter operation only. It is also possible to compute the minium or maximum value of a node. Furthermore, the flexibility of the library allows the user to perform any kind of specific communication between nodes with a proper mapping. First, we define some concepts and terms that are necessary for the good comprehension of the library. Afterward, we explain more in details the prototypes for the setup and the operation functions.

2.1 Concepts

2.1.1 Local-to-global mapping

For each element, a local-to-global mapping is a correspondence between the local and the global numbering of the nodes. Consequently, for each group of n_{el} elements (each one being made of N^d nodes) owned by process p, the mapping, denoted $id^{(p)}$, is simply an array of size $n_{el} \times N^d$, such that if i is the local number of a node on the process, then its global number is given by $id^{(p)}[i]$. For example, if we consider the example in figure 1 and we assume that nodes are owned by different processes (Ω^1 on process 0 and Ω^2 on process 1), the local-to-global mapping arrays, are given by

$$\begin{aligned} id^{(0)} =& (1, 2, 3, 4, 5, 6, 7, 8, 9), \\ id^{(1)} =& (3, 10, 11, 6, 12, 13, 9, 14, 15). \end{aligned}$$

If both elements are on the same process, the resulting mapping array is simply the junction of $id^{(0)}$ and $id^{(1)}$. The elements of the mapping might positive or negative, depending upon the desired behavior. If the element is negative, then it is said to be “flagged”. This mapping allows the gs library to create a topology for sending data between the processes and apply the gather-scatter operation.

2.1.2 Symmetric behavior

If all the elements of the mapping are positve, then the behavior of the gather-scatter operation is said to be symmetric. In that case, a gs operation performs a "real" gather-scatter. If we assume that the vector $\underline{u}_L$ is partioned among several processes, each process p having a piece of the whole vector denoted by $\underline{u}_L^{(p)}$, then the resulting global vector $\underline{v}$, which is partioned as well according to $\underline{v}^{(p)}$, is given after the gather-scatter operation by:

$$\underline{v}^{(p)}[i] = \sum_{(p,j)\in S_{id^{(p)}[i]}} \underline{u}_L^{(p)}[j], \tag{3}$$

where S_i is the group of all local nodes having global index i. This means that element i of $\underline{v}^{(p)}$ is the sum of all elements having global index $id^{(p)}[i]$ (whatever process they're on).

2.1.3 Asymmetric behavior

If some elements of the mapping are negative, then the behavior is said to be asymmetric. If only one pair (p, i) is unflagged (i.e. only one $id^{(p)}[i]$ is positive on all p), the topology is said to be *unique*. If the topology is unique, only interface elements are flagged negatively. Two subcases of the asymmetric behavior exist: the *transposed* asymmetric and the *non-transposed* asymmetric behaviors.

In the non-transpose case, a local node having a negative global index (i.e. that is flagged) **does not participate** in the sum from equation (3) but **does receive** the result. In the specific case when the topology is unique (i.e. only one $id^{(p)}[i]$ is positive on all p), this corresponds to applying the Q operator from equation (2) or scatter operation.

In the transpose case, a local node having a negative global index (i.e. that is flagged) **does participate** in the sum from equation (3) but **does not receive** the result. In the specific case when the topology is unique (i.e. only one $id^{(p)}[i]$ is positive on all p), this corresponds to applying the Q^T operator from equation (1) or gather operation.

2.2 Setup

Before performing a gather-scatter operation, a handle needs to be created. This handle is a structure that contains the information about to how do the communication.

2.2.1 In C

In C, the prototype to call the setup is the following:

```
struct gs_data *gs_setup(const slong *id, uint n, const struct comm
    *comm, int unique, gs_method method, int verbose);
```

Let us detail the inputs and output for that function.
Output:

- ***struct gs_data****: the output is a pointer of type ***struct gs_data***, which is the structure built for gs operations. This handle is the one that is used for any gs operation using the topology built by *gs_setup*.

Inputs:

- ***const slong *id***: constant pointer to a vector containing the local to global mapping. *slong* denotes signed long integer and is a data type defined in *types.h*.
- ***uint n***: number of elements in the mapping. In the case of parallel computation, n is the number of local nodes possessed by a process (i.e. this is the size of vector id).
- ***const struct comm *comm***: constant pointer to a structure called *comm*. This is a structure that is specific to Nek and this is not a simple MPI communicator. Basic usage of the communicator structure can be found in *comm.h*.
- ***int unique***: if $unique = 0$, topology is based explicitly on the mapping that is given to the function. If $unique \neq 0$ (typically $unique = 1$), a unique topology is built. This means that, no matter the signs of the mapping elements, the topology is such that only one pair (p, i) is unflagged (i.e. only one $id^{(p)}[i]$ is positive on all p). Let us not that in any case, the vector id is left unchanged.
- ***gs_method method***: method for the communication between processes. The different options are:
  - *gs_pairwise*: ?
  - *gs_crystal_router*: ?
  - *gs_all_reduce*: ?
  - *gs_auto*: tries approximately 10 runs of each of the previous methods and then chooses the fastest one.
- ***int verbose***: statistics for the gather-scatter operation are displayed in command line if $verbose \neq 0$ and are not otherwise.

Once the handle is not needed any more, it can be freed with *gs_free(struct gs_data *gsh)*.

2.2.2 In Fortran

The gather-scatter library can also be called from Fortran in a very similar, but more limited, way as it is in C. The setup is done with

```
subroutine gs_setup(gs_handle, id, n, comm, mp),
```

where

- ***gs_handle*** is the Fortran version of the gs handle, which is being set up,
- ***id*** is an integer array of the global mapping,
- ***n*** is the number of elements in the array,
- ***comm*** is an MPI communicator and not a specific structure like in the C version,

- ***mp*** is the number of parallel processes.

Let us note that it is not possible to create a unique topology or to chose the method for parallel communication. To free the handle, use *gs_free(gs_handle)*.

2.3 Gather-scatter operation

2.3.1 In C

Once the handle has been set up, gather-scatter operation can be performed. In C, the prototype of the corresponding function is given by:

```
void gs(void *u, gs_dom dom, gs_op op, unsigned transpose, struct
    gs_data *gsh, buffer *buf);
```

The function has not output so let us analyze the inputs and how u is affected by the operation.

Inputs:

- ***void *u***: pointer to the local array of data.
- ***gs_dom dom***: type of the data stored in u. The different types are:
  - *gs_int*,
  - *gs_double*,
  - *gs_float*,
  - *gs_long*: long integer,
  - *gs_sint*: *sint* is defined in *types.h*,
  - *gs_slong*: *slong* is defined in *types.h*.
- ***gs_op op***: operation performed by the function. Not only the simple gather-scatter operation (sum) is available. It is also possible to get the minimum of maximum value of each local element or to multiply the elements. Same logic as for the sum applies concerning the symmetric/asymmetric behaviors, unique topology, etc. Operations available are:
  - *gs_add*: "basic" gather-scatter,
  - *gs_mul*: the sum from equation 3 is replaced by a multiplication,
  - *gs_max*: the sum from equation 3 is replaced by the max operator,
  - *gs_min*: the sum from equation 3 is replaced by the min operator,
  - *gs_bpr*: unknown.
- ***unsigned transpose***: if $transpose = 0$, the non-transpose asymmetric behavior is used. This means that flagged interface elements do not participate to the operation but still receive the result (scatter). Inversely, if $transpose \neq 0$ (typically $= 1$), the transpose asymmetric behavior is used. This means that flagged interface elements do participate to the operation but do not receive the result (gather).
- ***struct gs_data *gsh***: pointer to the handle that has been created by *gs_setup*.
- ***buffer *buf***: ??. This buffer can be the *NULL* pointer, in which case, a static buffer is used, shared across all gs handles. Buffer structure is defined in *mem.h*.

2.3.2 In Fortran

A gs operation is called with

```
subroutine gs_op(gs_handle, u, dom, op, transpose),
```

where

- ***gs_handle*** is the gs handle,
- ***u*** is the local array containing the data,
- ***dom*** defines the data type:
  - *gs_double* if $dom = 1$,
  - *gs_sint* if $dom = 2$,
  - *gs_slong* if $dom = 3$.
- ***op*** defines the operation to be performed:
  - *gs_sum* if $op = 1$,
  - *gs_mult* if $op = 2$,
  - *gs_min* if $op = 3$,
  - *gs_max* if $op = 4$.
- ***transpose***: just like in C, if $transpose = 0$, the non-transpose asymmetric behavior is used. Inversely, if $transpose \neq 0$ (typically $= 1$), the transpose asymmetric behavior is used.

2.4 Other functions

Some additonnal functions that make the use of the library easier are available.

2.4.1 gs_unique

This function is available in C only and the prototype is:

```
void gs_unique(slong *id, uint n, const struct comm *comm);
```

This call modifies *id*, "flagging" (by negating $id[i]$) all (p, i) pairs in each group except one. The sole "unflagged" member of the group is chosen in an arbitrary but consistent way. If the "unique" flag is set when calling *gs_setup*, the behavior is equivalent to first calling *gs_unique*, except that the id array is left unmodified.

2.4.2 gs_many

This function combines the communication for gs operations on multiple arrays. Its prototype is:

```
void gs_many(void *const *u, unsigned k, gs_dom dom, gs_op op,
    unsigned transpose, struct gs_data *gsh, buffer *buf);
```

If u is an array of k arrays (each of length n as specified in *gs_setup*), built like

```
double v1[n], v2[n], ..., vk[n];
double (*u)[k] = {v1, v2, ..., vk};
```

then the function

```
gs_many(u,k, gs_double,op, t, g,&buf);
```

is equivalent to

```
gs(v1, gs_double,op, t, g, &buf);
gs(v2, gs_double,op, t, g, &buf);
...
gs(vk, gs_double,op, t, g, &buf);
```

A call in Fortran is equivalent but the different arrays need to be given separately (up to 6):

```
subroutine gs_many(gs_handle, u1, u2, u3, u4, u5, u6, k, dom, op,
    transpose).
```

2.4.3 gs_vec

In C:

```
void gs_vec(void *u, unsigned k, gs_dom dom, gs_op op, unsigned
    transpose, struct gs_data *gsh, buffer *buf)
```

This operation is like "gs" operating on the data double u[n][k], with summation here being vector summation. Number of messages sent is independent of k.

In Fortran:

```
subroutine gs_vec(gs_handle, u, k, dom, op, transpose)
```

3 Examples

We now implement the example from figure 1 and analyze the results for different configurations. We assume that Ω^1 and Ω^2 are on two different processes. In table 1, we remind the local-to-global mapping $id()$ and we present its asymmetric version $id_asym()$, obtained by applying function *gs_unique()* to $id()$, and a vector of arbitrary values u, on which the gs operations will be applied. Columns corresponding to interface elements are colored in red, blue and green for nodes 3, 6 and 9 respectively because they are the nodes of interest.

3.1 Symmetric behavior

We test the four different operations *gs_sum*, *gs_mul*, *gs_min* and *gs_max* in the case of symmetric behavior, when $id()$ is used. Results are presented in table 2.

We see that each interface element participates to the operation and receives the result.

Ω^1	$id()$	1	2	3	4	5	6	7	8	9
	$id_asym()$	1	2	-3	4	5	6	7	8	-9
	u	1.0	1.5	2.0	2.0	0.8	0.4	0.5	0.1	2.5
Ω^2	$id()$	3	10	11	6	12	13	9	14	15
	$id_asym()$	3	10	11	-6	12	13	9	14	15
	u	1.0	0.3	0.9	1.2	1.2	2.1	0.8	0.3	0.7

Table 1: Mappings and initial array u. Columns corresponding to interface elements are colored in red, blue and green for nodes 3, 6 and 9 respectively.

gs_add	Ω^1	1.0	1.5	3.0	2.0	0.8	1.6	0.5	0.1	3.3
	Ω^2	3.0	0.3	0.9	1.6	1.2	2.1	3.3	0.3	0.7
gs_mul	Ω^1	1.0	1.5	2.0	2.0	0.8	0.48	0.5	0.1	2.0
	Ω^2	2.0	0.3	0.9	0.48	1.2	2.1	2.0	0.3	0.7
gs_min	Ω^1	1.0	1.5	1.0	2.0	0.8	0.4	0.5	0.1	0.8
	Ω^2	1.0	0.3	0.9	0.4	1.2	2.1	0.8	0.3	0.7
gs_max	Ω^1	1.0	1.5	2.0	2.0	0.8	1.2	0.5	0.1	2.5
	Ω^2	2.0	0.3	0.9	1.2	1.2	2.1	2.5	0.3	0.7

Table 2: Results of the different gs operations when applied on array u in the symmetric case.

3.2 Asymmetric behavior, transpose = 0

We test the four different operations *gs_sum*, *gs_mul*, *gs_min* and *gs_max* for the asymmetric behavior, when *id_asym*() is used, and when the *transpose* parameter is equal to 0. Results are presented in table 3.

As we can see for that very simple case, all the operations produce the same result. The important thing to notice here is that nodes that have a negative id do not participate to the operation but still receive the result (scatter operation).

gs_add	Ω^1	1.0	1.5	1.0	2.0	0.8	0.4	0.5	0.1	0.8
	Ω^2	1.0	0.3	0.9	0.4	1.2	2.1	0.8	0.3	0.7
gs_mul	Ω^1	1.0	1.5	1.0	2.0	0.8	0.4	0.5	0.1	0.8
	Ω^2	1.0	0.3	0.9	0.4	1.2	2.1	0.8	0.3	0.7
gs_min	Ω^1	1.0	1.5	1.0	2.0	0.8	0.4	0.5	0.1	0.8
	Ω^2	1.0	0.3	0.9	0.4	1.2	2.1	0.8	0.3	0.7
gs_max	Ω^1	1.0	1.5	1.0	2.0	0.8	0.4	0.5	0.1	0.8
	Ω^2	1.0	0.3	0.9	0.4	1.2	2.1	0.8	0.3	0.7

Table 3: Results of the different gs operations when applied on array u in the symmetric case.

gs_add	Ω^1	1.0	1.5	2.0	2.0	0.8	1.6	0.5	0.1	2.5
	Ω^2	3.0	0.3	0.9	1.2	1.2	2.1	3.3	0.3	0.7
gs_mul	Ω^1	1.0	1.5	2.0	2.0	0.8	0.48	0.5	0.1	2.5
	Ω^2	2.0	0.3	0.9	1.2	1.2	2.1	2.0	0.3	0.7
gs_min	Ω^1	1.0	1.5	2.0	2.0	0.8	0.4	0.5	0.1	2.5
	Ω^2	1.0	0.3	0.9	1.2	1.2	2.1	0.8	0.3	0.7
gs_max	Ω^1	1.0	1.5	2.0	2.0	0.8	1.2	0.5	0.1	2.5
	Ω^2	2.0	0.3	0.9	1.2	1.2	2.1	2.5	0.3	0.7

Table 4: Results of the different gs operations when applied on array u in the symmetric case.

3.3 Asymmetric behavior, transpose = 1

We test the four different operations *gs_sum*, *gs_mul*, *gs_min* and *gs_max* for the asymmetric behavior, when *id_asym*() is used, and when the *transpose* parameter is equal to 1. Results are presented in table 4.

Once again, results are pretty straightforward because the case is very simple. The nodes which have a negative id do participate to the operation but do not receive the result (gather operation).

References

[1] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow.* Cambridge University Press, 2002. Cambridge Books Online.

[2] P. F. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale algorithms for reactor hydrodynamics. *Journal of Physics: Conference Series*, 125(1):012076, 2008.