# For developers

## A. I/O channels in SELFE

You need to exercise caution when dealing with parallel I/O especially for writing. For writing outputs, you'd generally let only 1 process do the job, e.g.

if(myrank==0) write(10,*)…..

If you do need to have all processes write e.g. debug messages, you'd then use channel 12 (see below).

Here are all I/O channel numbers used in different sub-models of SELFE (red numbers mean they have been used by other sub-models and you'd avoid using them):

1. Hydro/

   Channels between 8 and 200 are used by various codes for I/O. In particular:

   a) **101 to 100+noutputs** (inclusive of both): reserved for global outputs (including from tracers from sediment, EcoSim, ICM, as well as WWM);
   b) **201-250: non-standard outputs (e.g. at sidecenters, prism centers);**
   c) **251 to 259**: reserved for station outputs
   d) 10, 31, 32: used for one-off I/O – can be used by other sub-models as long as you close them immediately after use
   e) 12: this channel is initialized by different processes to point to files *outputs/nonfatal_xxxx*, where "xxxx" are the process IDs. Therefore it's very useful for debugging purpose; you can use it anywhere in your part of the code to dump messages to these files
   f) 16: this channel points to mirror.out (on rank 0), the main message output for info about the run. You should use it with if(myrank==0)
2. WWM
   a) **1100 to 1200**: handles for inputs/outputs etc

3. EcoSim
    a) **600**: outputting some messages
4. ICM
    g) **301 to 323**: reading channels for non-point source inputs for ICM
5. Sediment
    **a) 26, 2626**

**B. Rules for contributing your code to be included in the SELFE package**

Here are some rules for preparing your own code:

1) No spaces between "#" (pre-processor) and if/else/end;
2) Don't use "!" inside write/print or any other non-comment portion;

## Notes on SELFE Code

**A. Domain partitioning**

The domain is first portioned into non-overlapping sub-domains (in element sense; see Fig. 1). Then each sub-domain is augmented with 1 layer of ghost elements. This is accomplished by the call partition_hgrid() early in the main program. After calling aquire_hgrid(.true.) immediately after, the elements, nodes, sides in each augmented and non-augmented (i.e., without ghosts) domains are shown in Fig. 1. Intuition is typically followed although there are exceptions as described below.

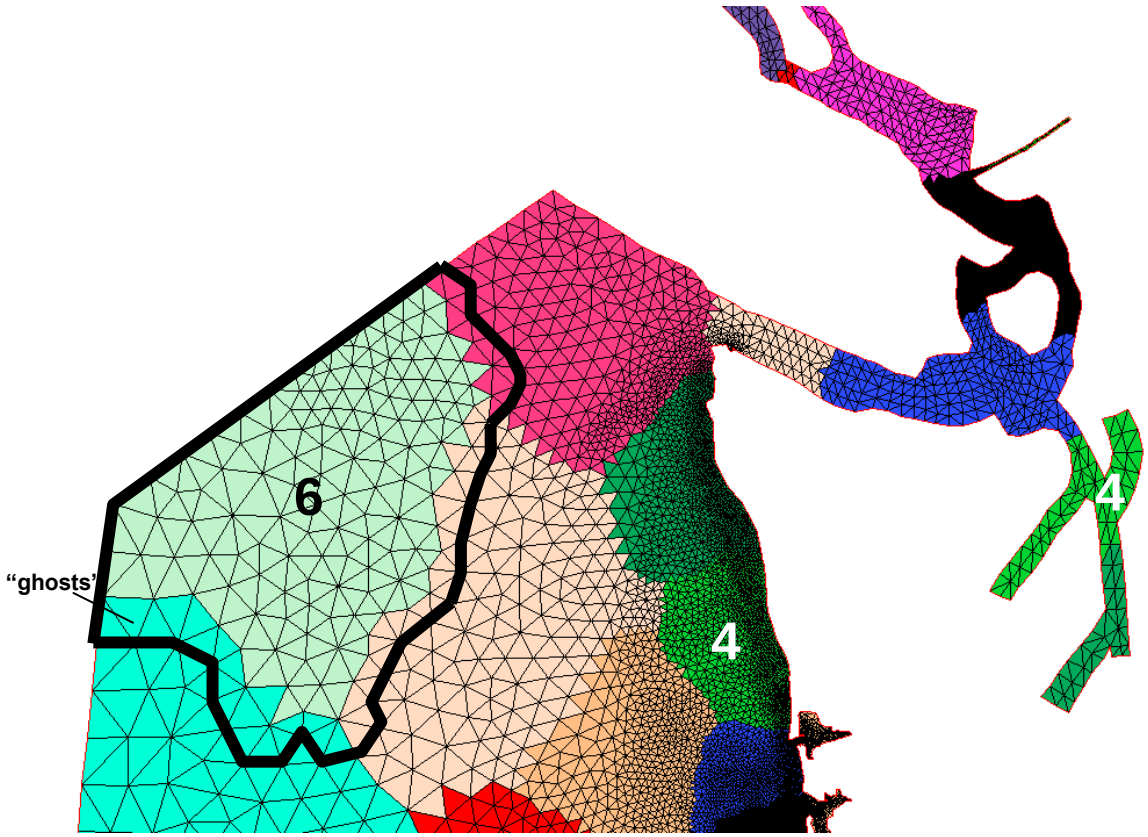


Fig. 1 Domain decomposition. Numbers indicate process (CPU rank) numbers, and each color represents a sub-domain. Inside each sub-domain, the nodes/sides/elements are called "residents". Note that the sub-domain of each process may not be contiguous (e.g., "4"). The thick black line indicates the boundary of the augmented domain of rank 6. Those nodes/sides/elements inside the augmented domain but outside the resident domain are called "ghosts". Note that each element is owned by 1 (and 1 only) CPU, but each side or node can be owned by many CPUs (when they are on the border of adjacent sub-domains).

**B. Arrays and constants**

| | Global | Local non-augmented | Ghost | Augmented |
|---|---|---|---|---|
| Elements | ne_global | ne | neg | nea=ne+neg |
| Nodes | np_global | np | npg | npa=np+npg |
| Sides | ns_global | ns | nsg | Nsa=ns+nsg |

1. llsit_type :: iegl(iegb)

   iegb is a global element #. If the element is resident (not ghost), iegl(iegb)%rank=myrank, and iegl(iegb)%id = local element index, and iegl%next=null. If iegb is a ghost, then iegl list has two entries: myrank and the rank where iegb is resident. All processors have this info, but the lists are different (1st entry is usually myrank etc).
2. llsit_type :: ipgl(ipgb)

   ipgb is a global node #. Use this list only when the node is resident (not ghost); it's confusing when ipgb is ghost. If ipgb is resident, ipgl(ipgb)%rank=myrank, and ipgl(ipgb)%id = local node index. ipgl%next%next%next…. is the linked list, with ranks in ascending order. Unless ipgb is an interface node (i.e., resident in more than 1 process), the list has only 1 entry. All processors have this info, but the lists are different (1st entry is usually myrank etc).
3. llsit_type :: isgl(isgb)

   isgb is a global side #. Similar to ipgl, if the side is resident (not ghost), isgl(isgb)%rank=myrank, and isgl(isgb)%id = local side index. isgl%next%next… is the list, with ranks in ascending order. All processors have this info, but the lists are different (1st entry is usually myrank etc).
4. int :: ielg(ie), iplg(ip), islg(isd)

   The global element index of local element ie in the aug. domain. Similar for the other two (nodes/sides).
5. int :: iegrpv(iegb)

   The rank # for global element iegb (before augmenting the domain). Used mainly in partitioning the grid.
6. Arrays that have similar meaning between global and local *aug*. domains, i.e., they do not have problem of getting outside the aug. domain: i34, elnode (old name: nm), elside (js), *ssign*, snx, sny …
7. Arrays that need special attention in the aug. domain:

   int :: ic3(1:3, ie) – positive if the neighbor element is inside the aug. domain as well (and in this case it is the local index of the neighbor element); 0 if (global)

boundary; negative if the neighbor element is outside the aug. domain and in this case, the absolute value is the *global* element index.

int :: nne(ip) – total # of neighbor elements around local node ip, including those outside the aug. domain (i.e., same as nnegb()).
int :: indel(1: nne(ip),ip) – surrounding element indices. If inside aug. domain, this is the *local* element index; if outside, this is the negative of the *global* element index.
int :: iself(1: nne(ip),ip) – same as global iselfgb, i.e., the elemental local index for node ip in neighbor element indel() (even if it is outside).

int :: nnp(ip) – total # of surrounding nodes for node ip, excluding all nodes outside the aug. domain. For SELFE, include all nodes outside.
int :: indnd(1: nnp(ip),ip) – list of surrounding nodes, excluding all nodes outside the aug. domain. For SELFE, all nodes outside will have negative global index returned.

int :: isdel(1:2,isd) & isidenode(1:2,isd) – order of the two adjacent elements follows global indices, and so the vector from node 1 to 2 in isidenode(1:2,isd) forms local y-axis while the vector from element 1 to 2 in isdel(1:2,isd) forms local x-axis. Therefore either of isdel(1:2,isd) can be negative. The element index is local (positive) if it is inside the aug. domain; otherwise the minus of global element index is returned. The local side isd is on the boundary if and only if isdel(2,isd)=0, and in this case, isdel(1,isd) must be inside the aug. domain (i.e., positive) (if isd is resident). If isd is resident and not ghost, isdel() has the same meaning as serial code, i.e., is(1,isd)>0 (inside the aug. domain), and isdel(2,isd)>=0, and isd is on the boundary if and only if isdel(2,isd)=0.

double :: delj(isd) – meaningful only if isd is resident.

8. Boundary arrays:

nope_global – total # of open bnd segements in the global domain.
nope – total # of open bnd segements in the aug. domain.
iopelg(1:nope) – returns global open bnd segment # for a local open bnd segment.
iopegl(0,k) - # of local fragmentations of global open bnd segment k.
iopegl(j,k) - local segment # of jth (1<=j<=iopegl(0,k)) fragmentation of global open bnd segment k.
nond(1:nope) – total # of open bnd nodes on each segment. The corresponding global array is nond_global().
iond(nope,1: nond(1:nope)) – list of local node indices on each open bnd segment. The corresponding global array is iond_global().
nland - total # of land bnd segements in the aug. domain. nland_global is global.
nlnd(1:nland) - total # of land bnd nodes on each segment.
ilnd(nland,1: nlnd(1:nland)) – list of local node indices on each land bnd segment.
isbnode(ip,1:2) (ELCIRC) – 0: node ip not on open bnd; positive (up to 2 indices to account for overlapping of open bnd segments) if on the *local* open bnd.
nosd(nope) (ELCIRC) - # of open bnd sides on each segment.
iosd(nope, 1: nosd(nope)) (ELCIRC) – list of open bnd side on each segment.
noe(nope) (ELCIRC) - # of open bnd elements on each segment.
ioe(nope, 1: noe(nope)) (ELCIRC) – list of open bnd elements on each segment.
isbe(2,1:nea) (ELCIRC) – if the element is on the *local* open bnd, this returns the local segment # and element #. 0 otherwise.
isbs() (ELCIRC) – similar to isbe.

Notes for SELFE: most arrays point to global bnd segment #. Most b.c. arrays are global as well.
isbnd(-2:2,ip) (SELFE) - If ip is on 1 open bnd only, isbnd(1,ip) points to the *global* segment # of that open bnd and isbnd(2,ip)=0; if ip is on 2 open bnds, isbnd(1:2,ip) point to the *global* segment #s for the 2 open bnds. If ip is on land bnd only (i.e., not on open bnd), isbnd(1,ip)=–1 and isbnd(2,ip)=0. If ip is an internal node, isbnd(1:2,ip)=0. **Therefore, ip is on open bnd if isbnd(1,ip)>0 (and in this case isbnd(2,ip) may also be positive, even though isbnd(2,ip)**

**may be outside the aug. domain), on land bnd (not on any open bnd) if isbnd(1,ip)=−1, and an internal node if isbnd(1,ip)=0. If on open bnd, isbnd(-2:-1,ip) are global index (i.e., isbnd(-1,ip)*th* node on the isbnd(1,ip)*th* open bnd);**

isbs(nsa) (SELFE) - positive if a local side is on the global open bnd (in this case, isbs() is the *global* segment #); −1 if it is on land bnd; 0 if internal side.

iettype, ifltype, itetype, and isatype all take global bnd segment as argument; other b.c. arrays (eth etc) are also global.

uth(nvrt,nsa),vth(nvrt,nsa) – local b.c. for ifltype/=0 for a local side.

uthnd(nvrt,mnond_global, nope_global) vthnd() – global arrays.

elbc(ip) – ip is a local node.

9. Arrays defined in elfe_msgp.F90

   nnbr - # of neighbor processors (excluding myrank).

   nbrrank(nnbr) – rank of each neighbor processor.

   int :: ranknbr(0:nproc-1) – neighbor # for each processor (0 if not neighbor).

   nerecv(nnbr) - # of elements to be received from each neighbor.

   ierecv(1: nerecv(nnbr),nnbr) – list of element indices (ghost in myrank) to be received from each neighbor (where the elements are resident and not ghost).

   nesend(nnbr) - # of elements to be sent to each neighbor.

   iesend(1: nesend(nnbr),nnbr) – list of element indices (local resident in myrank) to be sent to each neighbor (where the elements are ghost).

   Similar for nodes/side (nprecv, iprecv etc).

   A note on ghost exchange: since the message exchanges between processors have to wait for each other in order to communicate collectively, it's not necessary to synchronize the processes.

10. ParMetis routine:

    call ParMETIS_V3_PartGeomKway(vtxdist,xadj,adjncy,vwgt,adjwgt,wgtflag, &
                numflag,ndims,xyz,ncon,nproc,tpwgts,ubvec,options, &
                edgecut,part,comm)

    My notes from manual:

    (p : # of processors;

n : total # of vertices (local) in graph sense;
m : total # of neighboring vertices ("edges"); double counted between neighboring vertice u and v.
ncon : # of weights for each vertex.)

int(in) vtxdist(p+1) : Processor j stores vertices vtxdist(j):vtxdist(j+1)-1
int (in) xadj(n+1), adjncy(m) : locally, vertex j's neighboring vertices are adjncy(xadj(j):xadj(j+1)-1). adjncy points to global index;
int(in) vwgt(ncon*n), adjwgt(m) : weights at vertices and "edges". Format of adjwgt follows adjncy;
int(in) wgtflag : 0: none (vwgt and adjwgt are NULL); 1: edges (vwgt is NULL); 2: vertices (adjwgt is NULL); 3: both vertices & edges;
int(in) numflag : 0: C-style numbering from 0; 1: FORTRAN style from 1;
int(in) ndims: 2 or 3 (D);
float(in) xyz(ndims*n) : coordinate for vertex j is xyz(j*ndims:(j+1)*ndims-1) (C style; ndims*(j-1)+1: ndims*j for FORTRAN);
int(in) nparts: # of desired sub-domains (usually nproc);
float(in) tpwgts(ncon*nparts) : =1/nparts if sub-domains are to be of same size for each vertex weight;
float(in) ubvec(ncon) : imbalance tolerance for each weight;
int(in) options : additonal parameters for the routine (see above);
int(out) edgecut : # of edges that are cut by the partitioning;
int(out) part() : array size = # of local vertices. It stores indices of local vertices.

**Appendix**: Important MPI calls (C code; FORTRAN code just adds a last argument as status)

- ```
  int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int
  source, int tag, MPI_Comm comm, MPI_Status *status )
  Inputs:
    count - maximum number of elements in receive buffer (integer);
    datatype - datatype of each receive buffer element (handle);
  ```
  ```

source - rank of source (integer);
tag - message tag (integer);
comm - communicator (handle).
Outputs:
buf - initial address of receive buffer (choice);
status - status object (Status).

- int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request ) – nonblock receive.
  Inputs:
  buf - initial address of receive buffer (choice);
  count - number of elements in receive buffer (integer);
  datatype - datatype of each receive buffer element (handle);
  source - rank of source (integer);
  Tag - message tag (integer);
  comm - communicator (handle).
  Output:
  request - communication request (handle)
- int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )
  Inputs:
  buf - initial address of send buffer (choice);
  count - number of elements in send buffer (nonnegative integer);
  datatype - datatype of each send buffer element (handle);
  dest - rank of destination (integer);
  tag - message tag (integer);
  comm - communicator (handle).
- int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request ) – non-block send
  Inputs:
  buf - initial address of send buffer (choice);
  count - number of elements in send buffer (integer);
  datatype - datatype of each send buffer element (handle);
  dest - rank of destination (integer);
  tag - message tag (integer);
  comm - communicator (handle).
  Output:
  request - communication request (handle).
- int MPI_Allreduce ( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm ) – Combines values from all processes and distribute the result back to all processes
  Inputs:
  sendbuf - starting address of send buffer (choice);
  count - number of elements in send buffer (integer). Also the size of the output (i.e., *i*th elements from each processor are summed up and returned as *i*th element of output);
  datatype - data type of elements of send buffer (handle);
  op - operation (handle) (e.g., MPI_SUM, MPI_LOR);
  comm - communicator (handle).
  Output:
  recvbuf - starting address of receive buffer (choice).

- int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,

MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) – only difference from MPI_Allreduce is that the result is sent to rank "root".

- int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm ) - Gathers together values from a group of processes.

  Inputs:
  sendbuf - starting address of send buffer (choice)
  sendcount - number of elements in send buffer (integer)
  sendtype - data type of send buffer elements (handle)
  recvcount - number of elements for any single receive (integer, significant only at root)
  recvtype - data type of recv buffer elements (significant only at root) (handle)
  root - rank of receiving process (integer)
  comm - communicator (handle)

  Output:

  Recvbuf - address of receive buffer (choice, significant only at root). The received values are stacked according to the rank number (i.e., first recvcount are from rank 0 etc).

- int MPI_Allgatherv ( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm ) - Gathers data from all tasks and deliver it to all.

  Inputs:
  **sendbuf -** starting address of send buffer (choice);
  **sendcount -** number of elements in send buffer (integer)
  **sendtype -** data type of send buffer elements (handle);
  **recvcounts -** integer array (of length group size) containing the number of elements that are received from each process;
  **displs -** integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i;
  **recvtype -** data type of receive buffer elements (handle);
  **comm. -** communicator (handle).

  Output:
  **recvbuf -** address of receive buffer (choice).

- int MPI_Type_indexed(
  int count,
  int blocklens[],
  int indices[],
  MPI_Datatype old_type,
  MPI_Datatype *newtype ) - Creates an indexed datatype; the corresponding routine in MPI2 is mpi_type_create_indexed_block().

  Inputs:

**count -** number of blocks -- also number of entries in indices and blocklens;
**blocklens -** number of elements in each block (array of nonnegative integers);
**indices -** displacement of each block in multiples of old_type (array of integers);
**old_type -** old datatype (handle).

Output:
**newtype -** new datatype (handle)

Notes: the new MPI type treats multi-dimensional arrays in FORTRAN as 1D array, expanding with 1st index varying before 2nd etc. So this routine can be used to grab discontiguous data blocks from multi- dimensional arrays.

So if a 2D array is a(nvrt,nea) the corresponding 1D array is illustrated below:

$$\overbrace{\begin{matrix} (1,1) & (2,1) & \cdots & (nvrt,1) \\ (1,2) & (2,2) & \cdots & (nvt,2) \\ \vdots & \vdots & \vdots & \vdots \\ (1,nea) & (2,nea) & \cdots & (nvrt,nea) \end{matrix}}^{nvrt} \quad nea$$

Now suppose we need to grab all ghost elements iesend(1:nesend), these will correspond to rows iesend(i) of the table. In this case the # of blocks is nesend, block length is nvrt, and displacement of ith block is (iesend(i)-1)*nvrt.

- int MPI_Barrier (MPI_Comm comm ) - Blocks until all process have reached this routine.
  Input: comm. - communicator (handle)
- int MPI_Type_struct(
  int count,
  int blocklens[],
  MPI_Aint indices[],
  MPI_Datatype old_types[],
  MPI_Datatype *newtype ) - Creates a struct datatype.

  Inputs:
  count - number of blocks (integer) -- also number of entries in arrays array_of_types , array_of_displacements and array_of_blocklengths;
  **blocklens -** number of elements in each block (array);
  **indices –** byte displacement of each block relative to the start of the type (array);
  **old_types -** type of elements in each block (array of handles to datatype objects).

  Output:
  **newtype** - new datatype (handle)
- int MPI_Alltoall( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm ) - Sends data from all to all processes.

Inputs:
**sendbuf -** starting address of send buffer (choice);
**sendcount -** number of elements to send to each process (integer);
**sendtype -** data type of send buffer elements (handle);
**recvcount-** number of elements received from any process (integer);
**recvtype -** data type of receive buffer elements (handle);
**comm. -** communicator (handle).

Outputs
**recvbuf** - address of receive buffer (choice)