CGPACK manual

Anton Shterenlikht

Mech Eng Dept, The University of Bristol, UK, mexas@bris.ac.uk

ABSTRACT

CGPACK is a cellular automata library for microstructure evolution, deformation and fracture. The library is designed for use on HPC systems. It is implemented in Fortran 2015 with extensive use of coarrays for parallelisation.

1 August 2017

Table of Contents

References

Vichniac, 1984.
G. Y. Vichniac, “Simulating physics with cellular automata,” *Physica D* **10**, pp. 96-116 (1984).

Chopard, 1998.
B. Chopard and M. Droz, *Cellular Automata Modeling of Physical Systems*, Cambridge (1998).

Phillips, 2001.
R. B. Phillips, *Crystals, Defects and Microstructures: modeling across scales*, Cambridge (2001).

LeSar, 2013.
R. LeSar, *Computational Materials Science*, Cambridge (2013).

Shterenlikht, 2015.
A. Shterenlikht and L. Margetts, “Three-dimensional cellular automata modelling of cleavage propagation across crystal boundaries in polycrystalline microstructures,” *Proc. Roy. Soc. A* **471**, p. 20150039 (2015). DOI: 10.1098/rspa.2015.0039.

Shterenlikht, 2015.
A. Shterenlikht, L. Margetts, L. Cebamanos, and D. Henty, “Fortran 2008 coarrays,” *ACM SIGPLAN Fortran Forum* **34**, pp. 10-30 (2015). DOI: 10.1145/2754942.2754944.

Smith, 2014.
I. M. Smith, D. V. Griffiths, and L. Margetts, *Programming the finite element method*, Wiley, 5ed (2014).

Kouznetsova, 2001.
V. Kouznetsova, W. A. M. Brekelmans, and F. P. T. Baaijens, “An approach to micro-macro modeling of heterogeneous materials,” *Comp. Mech.* **27**, pp. 37-48 (2001).

1. CGPACK library layout

The library is made up of a set of modules. All modules are named following this scheme: `cgca_m`*L*<*module name*>.f90 or `cgca_m`*L*<*module name*>.F90, where file extension `f90` means that the file does not need pre-processing, and file extension `F90` means the file requires pre-processing. *L* is an integer number, giving the *level* of each module. <*module name*> is a string of characters `[a-z]`.

The lowest level is 1. Modules of level *L+1* depend only on modules of level *L* and lower. So module of level 3 depends only on modules of levels 2 and 1. Modules of level 1 do not depend on any other modules. The only exception is the top level module that is named `cgca.F90`. At the time of writing these are the CGPACK modules:

```
cgca.F90
cgca_m1co.f90
cgca_m2alloc.f90
cgca_m2gb.f90
cgca_m2geom.f90
cgca_m2glm.f90
cgca_m2hx.f90
cgca_m2lnklst.f90
cgca_m2mpiio.f90
cgca_m2out.F90
cgca_m2pck.F90
cgca_m2phys.f90
cgca_m2red.f90
cgca_m2rnd.f90
cgca_m2rot.f90
cgca_m2stat.f90
cgca_m3clvg.F90
cgca_m3gbf.f90
cgca_m3nucl.f90
cgca_m3pfem.f90
cgca_m3sld.F90
cgca_m4fr.f90
```

2. Code availability

The code is available from

```
http://cgpack.sourceforge.net
```

Full sources are also available as ROBOdoc auto-generated documents:

```
html: http://cgpack.sourceforge.net/robodoc/toc_index.html
 pdf: http://cgpack.sourceforge.net/robodoc/cgpack.pdf
text: http://cgpack.sourceforge.net/robodoc/cgpack.txt
```

Most CGPACK code is distributed under BSD licence. Selected parts of CGPACK are distributed under the Apache license.

3. Building CGPACK

To build CGPACK do (adjust for your environment):

```
mkdir $HOME/cgpack
mkdir $HOME/lib
```

On some platforms, e.g. when using the Intel Fortran compiler, separate module files, extension `.mod`, will be created. On such platforms add a directory to store the module files, e.g.:

```
mkdir $HOME/mod
```

Then put the source code somewhere, e.g.:

```
cd $HOME/cgpack
svn co https://svn.code.sf.net/p/cgpack/code/ .
```

And build using one of the provided `Makefile`s, adjusting for your environment, e.g.:

```
cd head
make -f Makefile-mpiifort
make -f Makefile-mpiifort install
```

This will build the library under `$HOME/cgpack` and install it under `$HOME/lib/libcgpack.a`. If separate module files were built too, these will be installed under `$HOME/mod`.

4. Cellular automata (CA)

There is extensive background literature on CA (Vichniac, 1984; Chopard, 1998; Phillips, 2001; LeSar, 2013). The CGPACK library is designed for 3D analysis. The 26 cell neighbourhood (Moore's neighbourhood) is assumed (Shterenlikht, 2015).

The library provides parallelisation via coarrays, which are a new language element introduced in Fortran 2008 standard (Shterenlikht, 2015). The central feature of the library is the *space* coarray. This is an allocatable integer array coarray, defined as follows.

```
integer( kind=iarr ), allocatable, intent(inout) :: coarray(:,:,:,:)[:,:,:]
```

where `iarr` is an integer kind used for the space coarray. All kinds and other parameters are defined in module `cgca_m1co.f90`. Space coarray is allocated by routine `cgca_as` from module `cgca_m2alloc.f90`. Routine `cgca_as` takes many input parameters:

```
subroutine cgca_as( l1, u1, l2, u2, l3, u3, col1, cou1, col2, cou2, &
                    col3, props, coarray )
```

and allocates the space coarray as follows.

```
allocate( coarray(                                                  &
          l1-halo:u1+halo, l2-halo:u2+halo, l3-halo:u3+halo, props) &
          [col1:cou1, col2:cou2, col3:*] )
```

5. Halo exchange

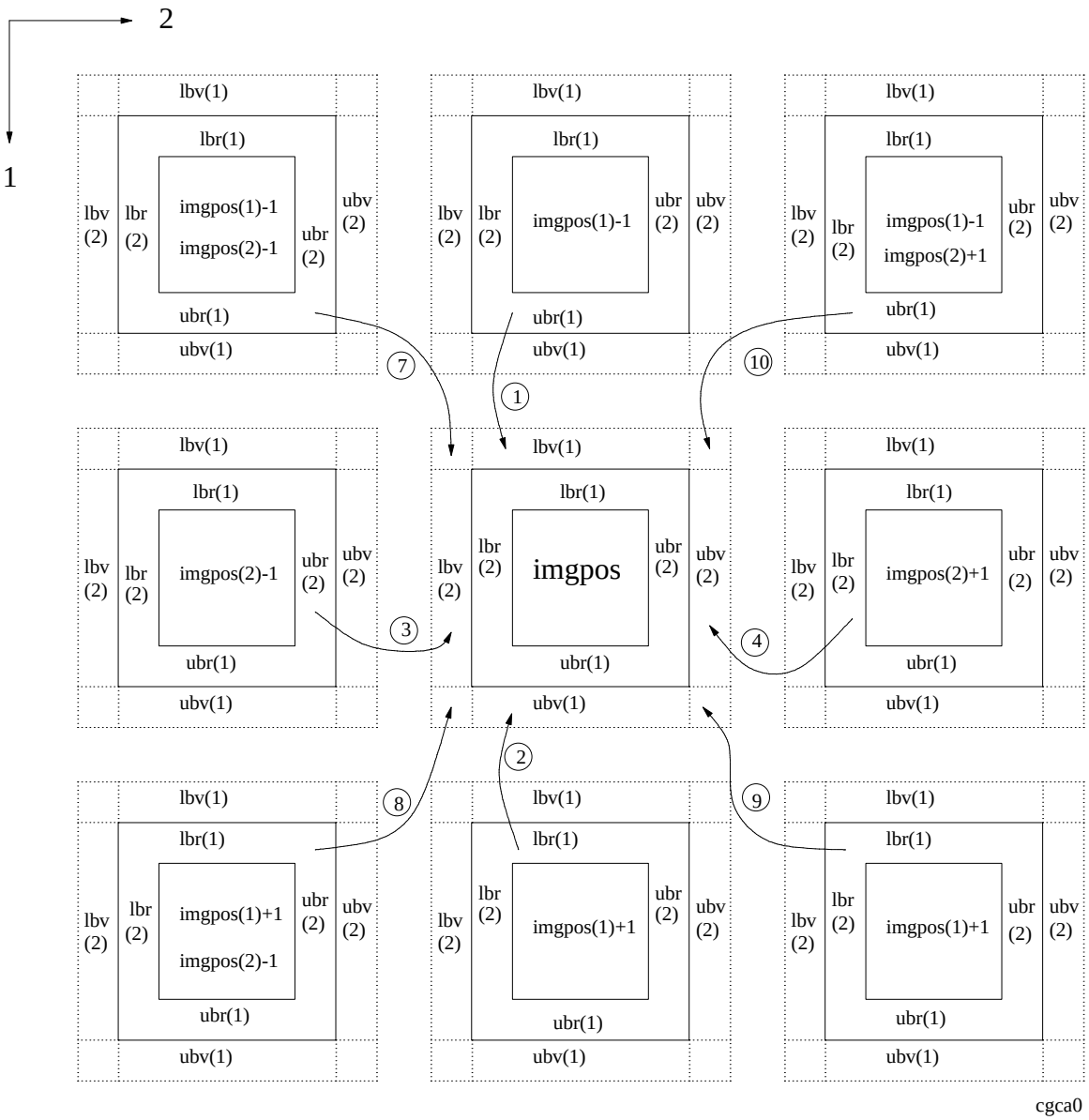
5.1. Global halo exchange

This is required if self-similar (or periodic) boundary conditions are preferred. Self-similar BC are probably more useful than fixed BC when grain statistics are important. For example, if fixed BC are used grains touching the boundary are not growing and evolving in the same way as grains which border only other grains. If sufficiently many grains are simulated, then self-similar BC improve grain size statistics. Otherwise the grains touching the model boundary must be excluded from the analysis.

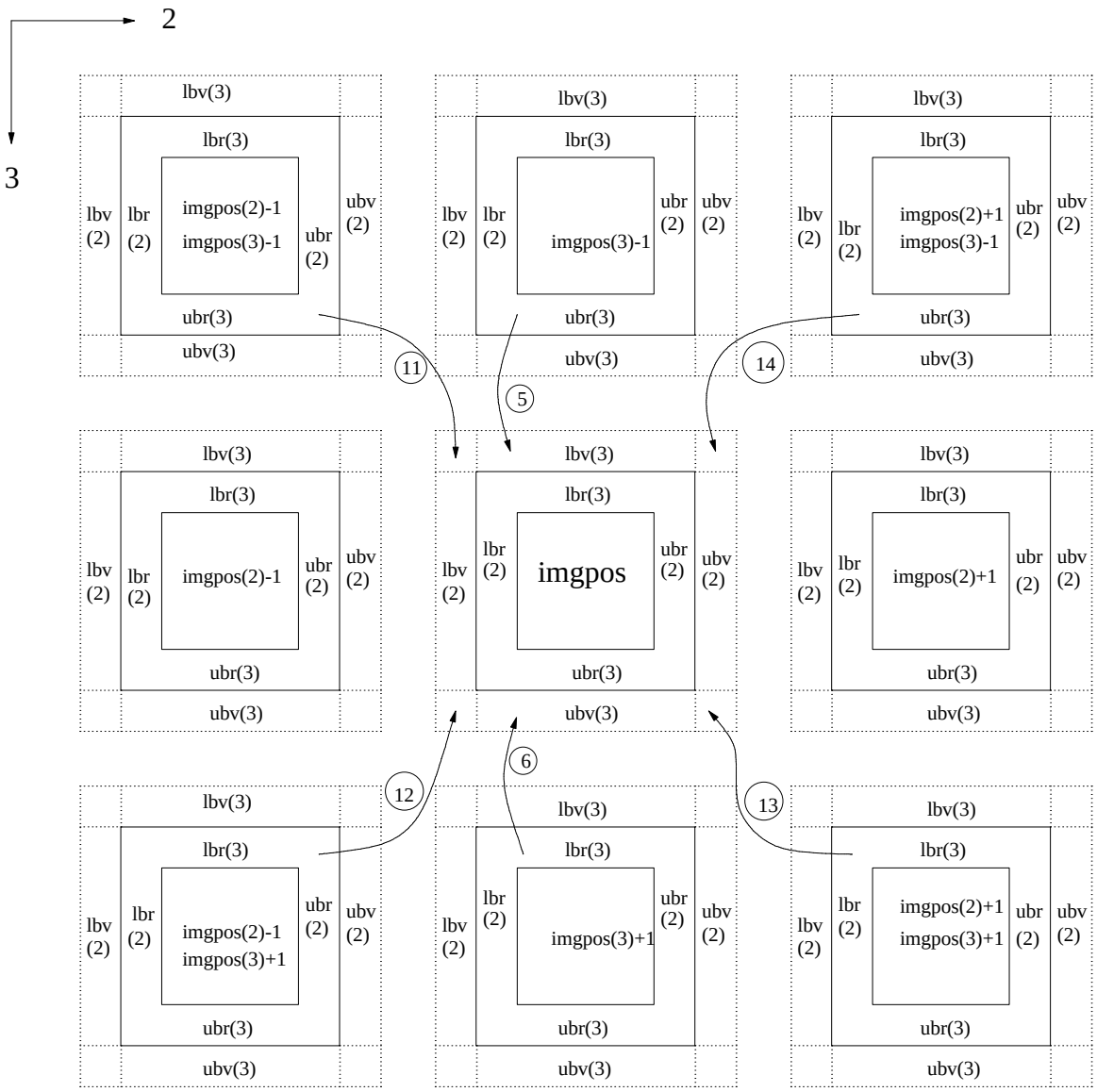
Global halo exchange is more complex than local.

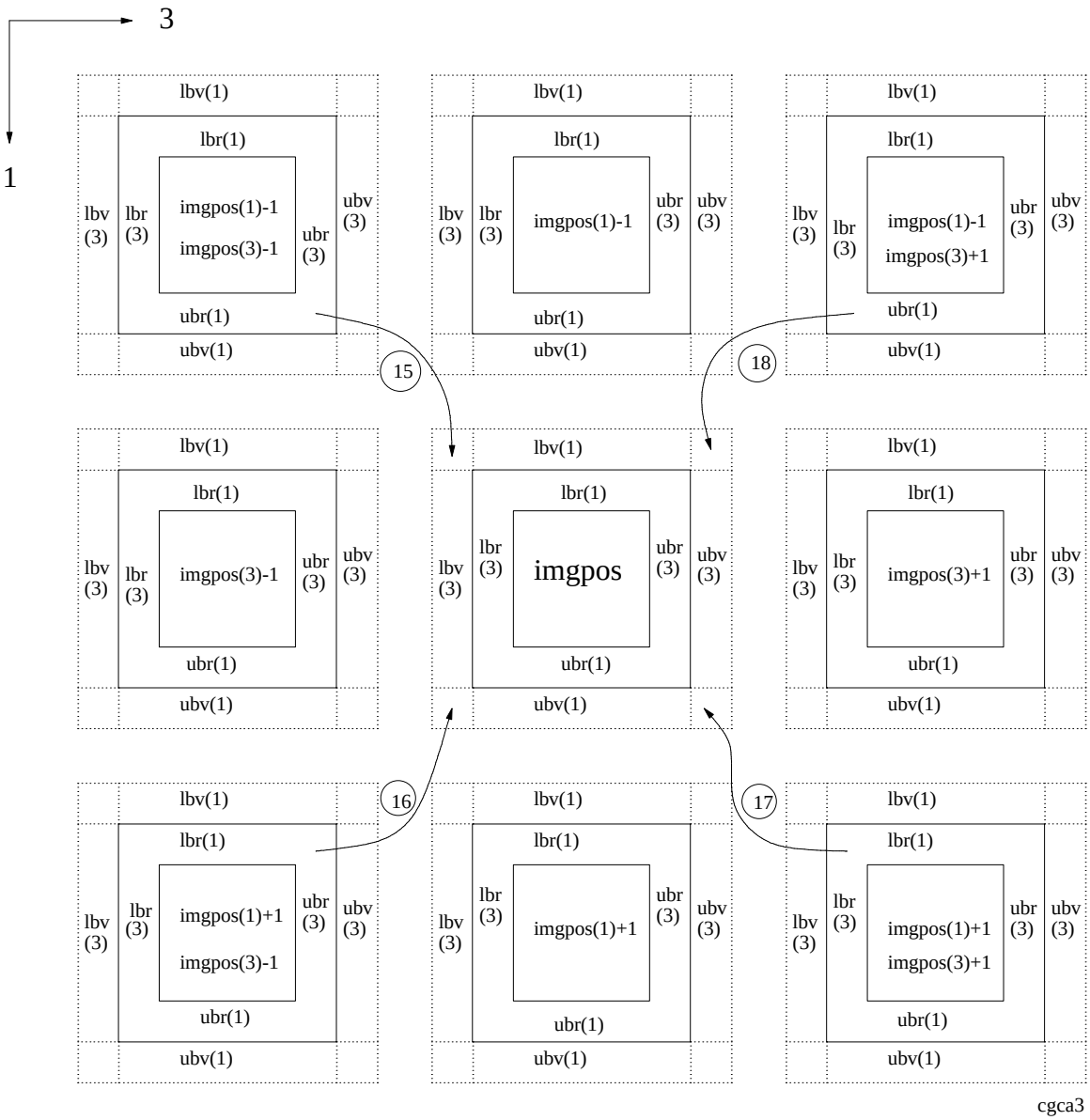
5.1.1. 2D planes

5.1.2. 1D edges

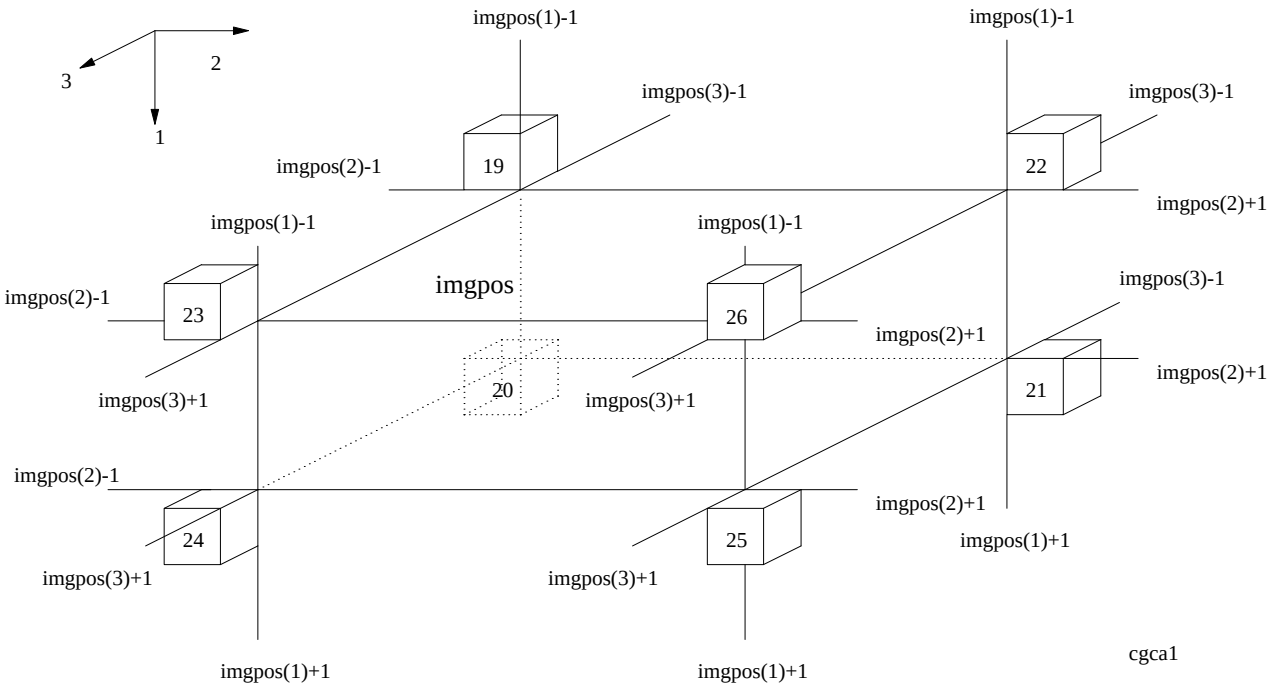


The same diagram is used for exchanging edges along all three axes. The coord. systems in the middle are to help understand the array assignments.


2
3
lbv(3)
lbr(3)
imgpos(2)-1
imgpos(3)-1
lbv (2)
lbr (2)
ubr (2)
ubv (2)
ubr(3)
ubv(3)
imgpos(3)-1
imgpos(2)+1
imgpos(3)-1
imgpos(2)-1
imgpos
imgpos(2)+1
imgpos(2)-1
imgpos(3)+1
imgpos(3)+1
imgpos(2)+1
imgpos(3)+1
11
5
14
12
6
13
cgca2


3
1
lbv(1)
lbr(1)
lbv (3)
lbr (3)
imgpos(1)-1
imgpos(3)-1
ubr (3)
ubv (3)
ubr(1)
ubv(1)
15
imgpos(1)-1
imgpos(3)+1
18
imgpos(3)-1
imgpos
imgpos(3)+1
16
imgpos(1)+1
imgpos(3)-1
imgpos(1)+1
17
imgpos(1)+1
imgpos(3)+1
cgca3

5.1.3. Corners



along all three axes. The coord. systems in the middle are to help understand the array assignments.}

6. Messages

CGPACK design assumes that routines might need to pass messages to the user. There are four levels of user messages:

1. information
2. warning
3. error
4. debugging

Each message is written to stdout as a single line of text. Some messages can be long, so they might wrap when viewed in some editors. The decision not to break a message into multiple one-line chunks was taken to simplify searching for messages, e.g. with `grep`. Having a message written as a single line makes searching for the complete message easier.

The first field in all messages is a label showing the level of the message, followed by a colon, `:`, exactly one of: `INFO:`, `WARN:`, `ERROR:` or `DEBUG:`. The standard labels are used to search for a particular type of messages. For example to check if there are any warnings issued by the program the user can use this command:

`grep WARN` *<outfile>*.

or

`grep "^WARN:"` *<outfile>*

or to see all debugging output, the user can issue this command:

`grep DEBUG:` *<outfile>*

or

`grep "^DEBUG"` *<outfile>*

The second field in all messages is the name of the routine that issued that message, followed by a colon, `:`, e.g. `cgca_clvgsd:`, `cgca_nr:`, etc.

Colons are used to help separate the messages into fields, with tools such as `awk`.

The third field is the content of the message. This field is not standardised and will differ from one routine to another, hopefully giving the user some helpful information.

Some messages will contain the number of the image that issued the message.

6.1. Information messages

Information messages start with `INFO:`. These messages are intended to give the user progress information, where it is expected that routines might take a while to complete. Examples:

```
INFO: cgca_sld: iterations completed: 140
INFO: cgca_sld: iterations completed: 150
INFO: cgca_clvgp: iterations completed: 10
INFO: cgca_clvgp: iterations completed: 20
```

Accordingly the routines which provide the information messages allow the user to specify the frequency of messaging. For example input argument `heartbeat` in routine

```
subroutine cgca_clvgp( coarray, rt, t, scrit, sub, periodicbc, iter,  &
                       heartbeat, debug )
```

gives the frequency of the information messages, in this case given in the number of CA iterations.

6.2. Warning messages

Warning messages start with `WARN:`. These messages are intended to warn the user of unexpected input, undesirable settings, unexpected, but not fatal, runtime conditions, etc. Example:

```
WARN: cgca_gcr: image 1144: No match found for given pair: 3904 2126.
WARN: cgca_nr: too many nuclei - no physical sense! nuclei/model size:  0.683E+06
```

The user generally does not have the control over warning messages. These are issued based on the logic of the CGPACK library. It is up to the user what action to take on encountering one of the warning messages. Some warnings might safely be ignored. Other warnings, such as the second warning above, might indicate that the results have no physical sense. The user can then decide to terminate the analysis early, change the input values and rerun. Yet other warnings might indicate unexpected conditions demanding further investigation, e.g. the first warning above.

6.3. Error messages

Error messages start with `ERROR:`.

CGPACK was designed to treat all errors as fatal. In fact all error conditions are implemented with `ERROR STOP` Fortran 2008 intrinsic which initiates error termination. The program is expected to exit as soon as possible. Therefore it is expected that there will not be more than a single error message issued by any CGPACK program. This means that searching for error messages might not be needed.

Typical conditions resulting in the issuing of error messages, e.g. in error termination, are insufficient memory when allocating arrays or conditions which violate the model logic, e.g. passing non-existent grain numbers or image numbers. Examples:

```
ERROR: cgca_dv: cannot deallocate coarray
ERROR: cgca_gc: coarray not allocated, img: 17
```

6.4. Debugging messages

Debugging messages start with `DEBUG:`.

Some routines provide a capability to print debugging output on request. For example, input argument `debug` in routine

```
subroutine cgca_clvgp( coarray, rt, t, scrit, sub, periodicbc, iter,  &
                       heartbeat, debug )
```

is a logical variable. If `debug = .true.` then some debugging information will be dumpted to stdout by that routine, and by all routines invoked by this routine, if any of them provide debugging capability. For example, routine `cgca_clvgp` above calls routine `cgca_clvgsd` which prints debug ouput when `debug = .true.` is passed to cgca_clvgp:

```
DEBUG: cgca_clvgsd: img: 6: newstate=-3, calling cgca_gcf, gcupd=(34 9 1),
marr=(9 9 27 9 27 27 9 27 27 9 9 27 9 34 34 34 34 34 9 34 34 34 34 34 34
 34 34 ).
DEBUG: cgca_clvgsd: img: 2: newstate=-4, calling cgca_gcf, gcupd=(27 34 1),
marr=(27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 34 27 27 9 9 9 34 34 34
 34 34 34 ).
```

Note that the long lines of debugging output have been wrapped here.

The volume of debugging output can be very large. The user is advised to use caution when asking for debugging output, particularly on large core counts. In some cased debugging output can exceed several GB.

7. Interfacing with ParaFEM

ParaFEM is a parallel scalable finite element library written in Fortran 90/95/2003 with MPI library for communications. The main web site is:

`http://parafem.org.uk`.

The code is available via svn from:

`http://sourceforge.net/projects/parafem`

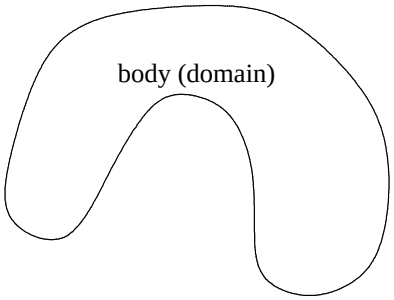
The best documentation for the code is Dr Margetts' book (Smith, 2014), specifically chapter 12. ParaFEM is distributed under BSD licence.

Intefrace of CGPACK with ParaFEM is implemented in module `cgca_m3pfem.f90`. The module includes the following routines:

```
cgca_pfem_boxin
cgca_pfem_cellin
cgca_pfem_cenc
cgca_pfem_cendmp
cgca_pfem_centroid_tmp
cgca_pfem_ctalloc
cgca_pfem_ctdalloc
cgca_pfem_ealloc
cgca_pfem_edalloc
cgca_pfem_enew
cgca_pfem_intcalc1
cgca_pfem_integalloc
cgca_pfem_integdalloc
cgca_pfem_integrity
cgca_pfem_partin
cgca_pfem_salloc
cgca_pfem_sdalloc
cgca_pfem_sdmp
cgca_pfem_simg
cgca_pfem_stress
cgca_pfem_uym
cgca_pfem_wholein
```

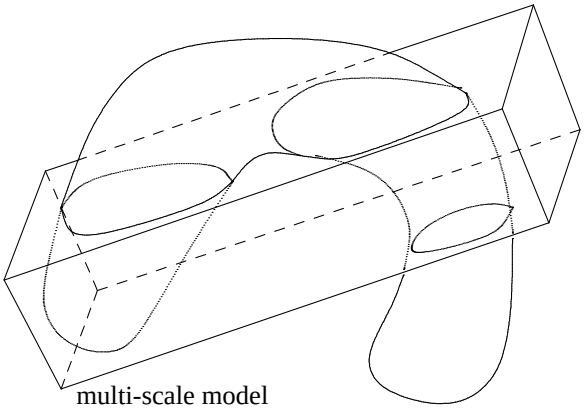
7.1. Mapping

Consider a body of arbitrary shape and size:



Consider a "box" of cells, i.e. a rectilinear array of arbitrary size

that occupies some or all of the space occupied by the body:

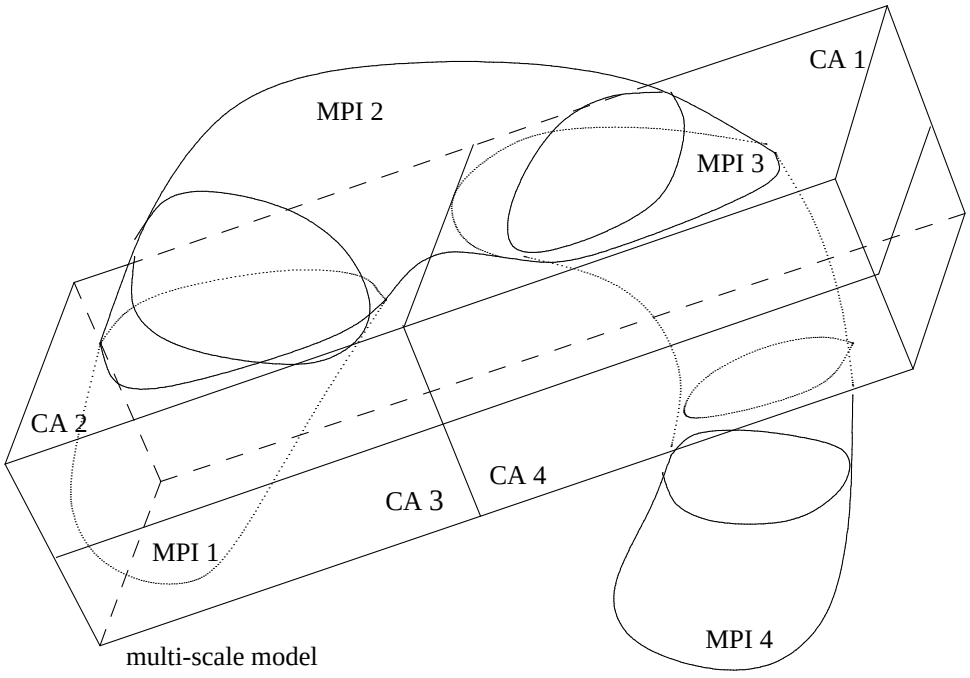


Here the dashed lines are the box edges obscured by the body, and thin dotted lines are the outlines of the body obscured by the box.

So there are bits of the material box which are outside of the body, and there are bits of the body which are outside of the material box. These regions are of no interest to us. However, it will be important to identify such regions. The regions of space which are occupied by the body and by the material box are subject to multi-scale analysis.

We assume that the number of coarray images and MPI processes are always identical. This is currently the case on Cray and Intel systems.

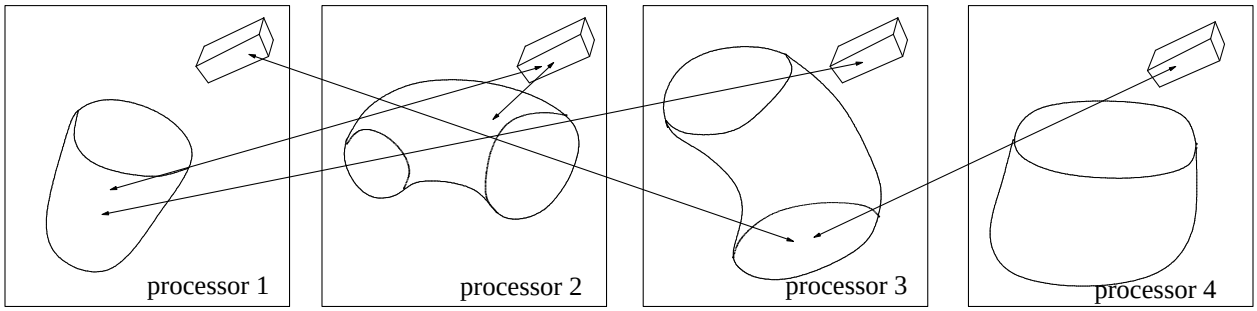
The following illustration of the MPI/coarray multi-scale model is drawn for 4 MPI processes and 4 coarray images. ParaFEM is written in fortran. Hence MPI ranks start from 1.



There are 4 sub-domains, processed by 4 MPI processes with ranks 1 to 4. These are denoted by thin solid lines drawn on top of the body. There are 4 images with the coarrays of cellular microstructure. These are also denoted by thin solid lines drawn on top of the material box.

Each process has its chunk of finite elements and its microstructure coarray. There are two problems.

1. Some FE have no corresponding cells and some cells have no corresponding FEs. Corresponding here means occupying the same space.
2. FEs and cells that do correspond to each other are not always allocated to the same processor.



The boxes represent the processors, or in Cray terminology, processing elements (PE). Each PE is given an MPI rank and a coarray image number. These are *not* guaranteed to be identical. On Cray systems on any PE MPI rank is identical to the image number. At present the interface relies on the fact that MPI rank is identical to the coarray image number. If this is not the same of any system, then the interface will not work.

The arrows in this diagram indicate data transfer between CA and FE. Coarray on PE 1 will have to communicate with FE on PE 3. FE on PE 1 will have to communicate with coarrays on PEs 2 and 3. And so on. Note that FE on PE 4 do not communicate with CA at all, because none of these FE occupy the same physical space as CA. Likewise, it is possible, although not shown in this diagram, that coarrays on some image will not need to communicate with FE at all. This will happen if none of these cells will share space with FE.

It is a major assumption of the framework that the mapping between the FE and CA is not affected by the deformation. For small deformation problems this assumption is natural. For large deformation problems this assumption means that the CA array is *deforming* together with FE. This assumption means that the mapping between CA and FE is established based on the initial, undeformed, geometry, and is not changed throughout the analysis.

The benefits of this approach are that the CA halo exchange algorithm is unaffected by linking CA to FE. Indeed, as far as CA is concerned, not much is changing at all.

One drawback of this simple mapping scheme is that there might be lots of communications across the node boundary. These correspond to the arrows crossing the box boundaries in the diagram above. In the example of the illustration there are four inter-processor CA $\leftrightarrow$ FE arrows, and only a single CA $\leftrightarrow$ FE arrow on the same processor, PE 2. However, this is a performance problem, which will have to be addressed later.

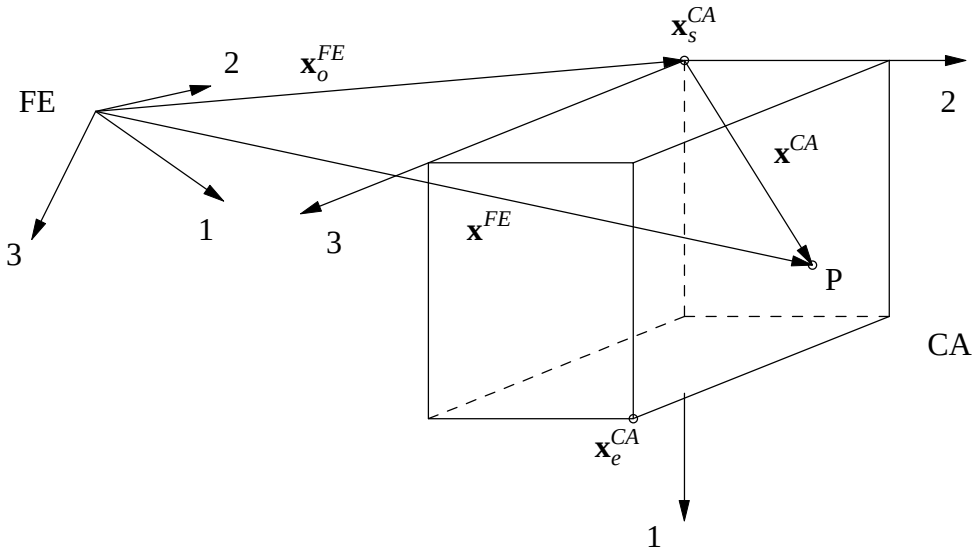
The first problem in constructing the interface following this general scheme is that each image must know which MPI processes to communicate with to obtain the FE data.

Let's denote the FE coordinate system (CS) by $\mathbf{x}^{FE}$, and the CA CS by $\mathbf{x}^{CA}$. The CA model "box" is always aligned with the CA CS. Let's denote the origin of the CA CS in the FE CS system by $\mathbf{x}_o^{FE}$. Let $\mathbf{R}$ be the rotation tensor *from* the FE CS *to* the CA CS. Then coordinates of some point P in the FE and the CA CS are related as follows:

$$\mathbf{x}^{CA} = \mathbf{R} \cdot (\mathbf{x}^{FE} - \mathbf{x}_o^{FE})$$

$$\mathbf{x}^{FE} = \mathbf{R}^T \cdot \mathbf{x}^{CA} + \mathbf{x}_o^{FE}$$

These relationships are illustrated below for a coarray on a single image.



Here $\mathbf{x}_s^{CA}$ and $\mathbf{x}_e^{CA}$ denote the extents of the coarray model space on each image.

Each image knows its own $\mathbf{x}_s^{CA}$ and $\mathbf{x}_e^{CA}$. The algorithm for constructing the mapping includes the following steps:

1. Find out what MPI processes contain FEs with *intial* centroid coordinates inside the coarray "box" on each image.
2. Find all element numbers from each MPI process that have to communicate with the coarray "box" on each image.

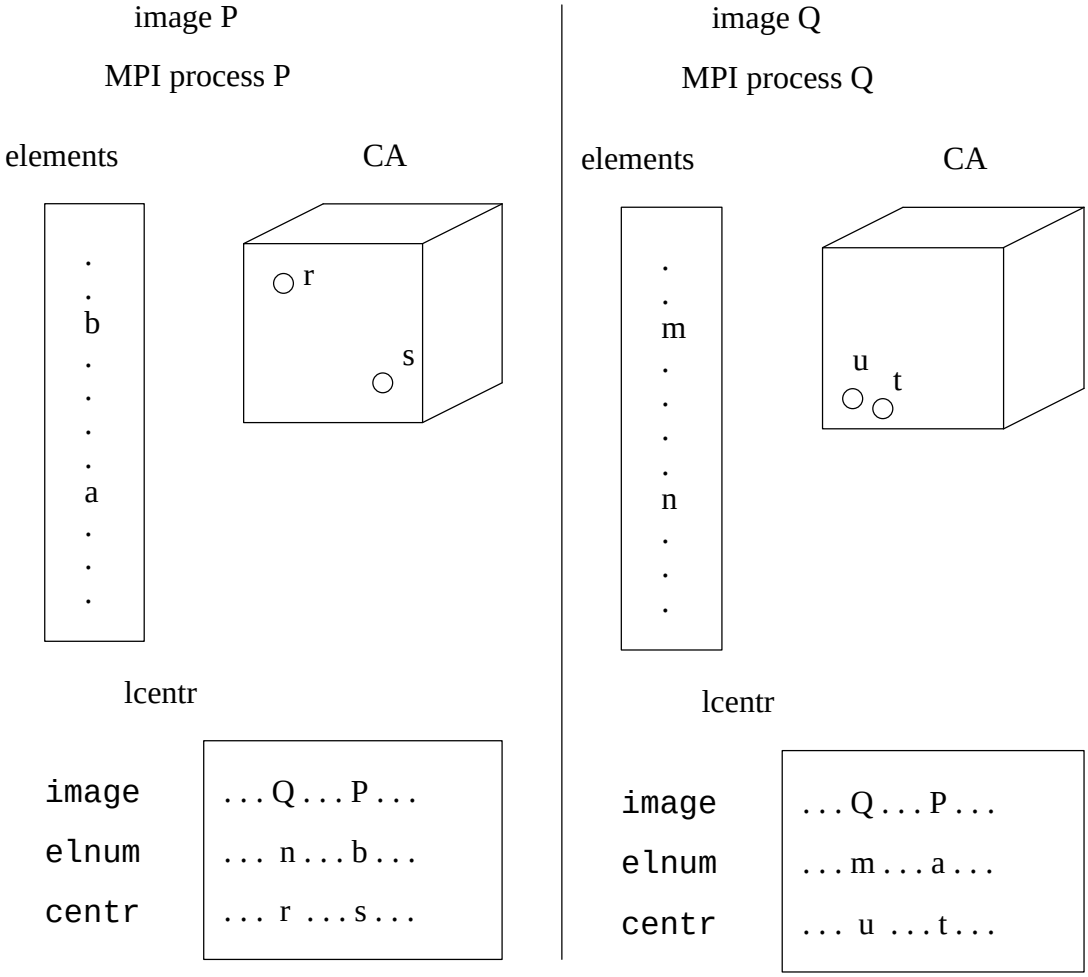
The key ParaFEM/CGPACK interface data structure is the allocatable private local, i.e. non-coarray, array of derived type `lcentr` established on each image. This array contains centroids of all finite elements linked to this image, including the element numbers and the images where these elements are stored. This array is defined as:

```
type mcen
  integer( kind=idef ) :: image
  integer( kind=idef ) :: elnum
  real( kind=cgca_pfem_iwp ) :: centr(3)
end type mcen
type( mcen ), allocatable :: lcentr(:)
```

Below is the illustration of the use of `lcentr` array. Consider 2 images, P and Q. The CA array on image P might have to communicate with FE which are stored on the same image, and also with those FE which are stored on other images. For example FE with centroid coordinates **r** is FE number n, that is stored on image Q. Getting and sending data to this FE will involve inter-image communication. FE with centroid coordinates **s** is FE number b, which is stored on the same image.

Similarly on some other image Q, a FE with centroid coordinates **u** is FE number m, that is stored locally on the same image. However, FE with centroid coordinates **t** is FE number a, that is stored on image P, so remote put/get operations will be required to communicate with that FE.

Note the key role played by `lcentr` array in linking FE with CA.



Using `lcentr` array it is possible for any image to know what finite elements it needs to exchange the data with, and to what images these belong:

```
CA on img 91 <-> FE 1266 on img 163 centr. in CA cs 2.55,  8.45,  4.95
CA on img 57 <-> FE 3104 on img 34 centr. in CA cs 5.00E-2,  1.75,  4.95
CA on img 36 <-> FE 435 on img 131 centr. in CA cs 3.85,  6.75,  2.45
CA on img 1 <-> FE 2292 on img 2 centr. in CA cs 2*5.00E-2,  2.45
CA on img 141 <-> FE 4000 on img 162 centr. in CA cs 5.15,  8.45,  7.45
CA on img 1 <-> FE 2293 on img 2 centr. in CA cs 0.15,  5.00E-2,  2.45
CA on img 1 <-> FE 2294 on img 2 centr. in CA cs 0.25,  5.00E-2,  2.45
CA on img 1 <-> FE 2295 on img 2 centr. in CA cs 0.35,  5.00E-2,  2.45
```

This data is dumped by routine `cgca_pfem_cendmp` of module `cgca_m3pfem`. The finite element numbers are local to their images. It is possible to calculate the global FE number, but there is no need. A combination of the image number and the finite element number identify the FE uniquely. For example, the first line in the above output says that finite element 1266 stored on image 163 has centroid coordinates $(2.55, 8.45, 4.95)$ in the CA coord. system. Material with these coordinates is processed with CA on image 91.

After `lcentr` has been established, the second important mapping issue can be resolved. Cells which are outside of the FE model must not be processed. This means no fracture propagation can occur in such cells. However, since there is finite resolution in FE model and in CA, this problem cannot be posed precisely. Depending on the FE size and the CA cell size, a cell can be deemed to lie inside the FE model or out. The algorithm implemented in the library uses some characteristic distance measure, L_c. The

criterion is this - if the distance between a cell and the centroid of *any* FE in `lcentr` is less than L_c, then this cell is considered to lie inside the FE model, otherwise it is considered to lie outside of the FE model. All cells in the fracture layer, which lie outside of the FE model, are given state `cgca_state_null`, defined in module `cgca_m1co`. These cells are not processed at all in any of the fracture routines. Although these cells represent material in the material layer, this material is simply ignored in all fracture calculations.

The algorithm is very simple. Just loop through all cells on an image. Routine `cgca_pfem_cellin` checks whether a cell is in or out.

However, some optimisations are possible. For example, if some assumptions on smoothness of FE model boundary are made, then it can be assumed that if all 8 corner cells of some sub-image CA box are either in or out, then other cells inside the box don't have to be checked. These can be assumed to have the same state as the 8 corner cells. Routine `cgca_pfem_boxin` does this check.

How to choose a sub-image box? A simple algorithm is to start from the whole box and do binary division along the longest dimension of the box. If a sub-box is neither in nor out, i.e. some of the 8 corner cells are in and some are out, then this sub-box is divided into 2 smaller boxes, and so on until the smallest box gets to be a single cell. Routine `cgca_pfem_partin` implements this algorithm.

There also an even simple routine. If the whole of the CA array on an image is out, i.e. if `lcentr` is empty on an image, then all cells are marked as out. Routine `cgca_pfem_wholein` implements this algorithm.

The sub-box algorithm `cgca_pfem_partin` needs to maintain a list of boxes to check. Module `cgca_m2lnklst` has several routines to establish, maintain and delete a linked list. When a sub-box is partly in, and is split into two smaller boxes, the sub-box is removed from the list and the new smaller boxes are added to it.

7.2.

Current work on the interface is carried under `xx14` ParaFEM developer programs:

`http://sourceforge.net/p/parafem/code/HEAD/tree/trunk/parafem/src/programs/dev/xx14/`

There are 3 programs, `xx14.f90`, `xx14noio.f90`and `xx14std.f90`. Program `xx14.f90` and `xx14noio.f90` include Cray extensions to the f2008 standard, namely `CO_SUM`. `xx14noio.f90` has no CA model IO. This program should be used for timing the optimising CGPACK without being biased by IO timing issues. `xx14std.f90` conform to f2008 strictly. Specifically it has no collectives, and is thus suitable for use with Intel compilers, which do not yet support collectives.

The programs simulate deformation in a 3D body. A part of this body is mapped to a microstructure array, established and processed via CGPACK. Currently `xx14` all programs build and run on Archer, and `xx14std` builds and runs on Intel systems.

7.3.

The stress coarray of derived type with allocatable array component is established on every image:

```
type type_stress
  real( kind=cgca_pfem_iwp ), allocatable :: stress(:,:,:)
end type type_stress
type( type_stress ) :: cgca_pfem_stress[*]
```

Its allocatable array component is allocated by routine `cgca_pfem_salloc`:

```
subroutine cgca_pfem_salloc( nels_pp, intp, comp )

!  INPUTS
!    nels_pp - number of elements on this image
!    intp - number of integration points per element
!    comp - number of stress tensor components

integer, intent( in ) :: nels_pp, intp, comp
```

as

```
call cgca_pfem_salloc( nels_pp, nip, nst )
```

where `nels_pp` is the number of elements per MPI process, `nip` is the number of integration points per element and `nst` is the number of stress tensor components, 6 in 3D case. So any image can access stress tensors on any other image, e.g.:

```
 img 96 FE 1 int. p. 8 stress  -3.12E-01 -3.31E-01 -1.40E-01 -5.59E-03
   1.38E-02  4.41E-02
```

This is the stress tensor for integration point 8 of element 1 on image 96. In this example elements have 8 integration points.

3. Each load/time iteration FE passes to the CA level the new stress tensor, $\boldsymbol{\sigma}$, There will be a provision for passing other variables to CA in future.
4. Do a certain number of iterations of crack propagation algorithm. The number of iteration is set with the characteristic length scale. (At present the characterstic length scale is taken equal to the longitudinal wave speed, more precisely the length travelled by the longitudinal wave per unit of time.)
5. The damage variable, D, is calculated for each CA "box". The Young's modulus of all integration points within the box is scaled as

$$E^{new} = E^{original} \times D$$

Originally $D = 1$. When $D = 0$ material has no remaining load bearing capacity. D is the ratio of the number of the fractured cells to the cracteristic area.

8.

8.1.

8.2.

8.2.1.

Module with routines for creating and updating of a linked list. The module contains two derived types:

and the following routines:

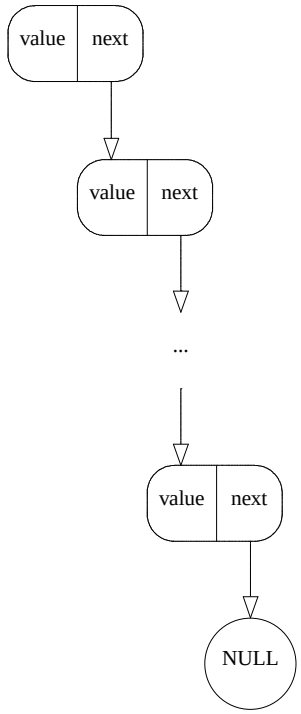
The list has to be established first with

8.2.1.1.

This is a derived type with a pointer component of the same type to maintain a link to other variables of the same type. The type is:

```
type cgca_lnklst_node
  type( cgca_lnklst_tpayld ) :: value
  type( cgca_lnklst_node ), pointer :: next
end type cgca_lnklst_node
```

This type allows for a standard one-direction linked list, which can be schematically illustrated as:



lnklst1.gv

where `NULL` means the pointer is not associated.

Access to the list is possible only from the head (top) node.

8.2.1.2.

This is the type for payload, i.e. the data contained in each node of the list.

```
type cgca_lnklst_tpayld
  integer :: lwr(3), upr(3)
end type cgca_lnklst_tpayld
```

The intention is that the data consists of the coordinates of the two corners of a CA box, the lower and the upper. The lower corner is that which has the lowest coordinates along all 3 directions. The upper corner is that which has the upper coordinates along all 3 directions. For example

```
lwr = (/  1 , 200 ,    3 /)
upr = (/ 10 , 202 , 3000 /)
```

would denote a box extending from 1 to 10 along direction 1, from 200 to 202 along direction 2, and from 3 to 3000 along direction 3.

8.2.1.3.

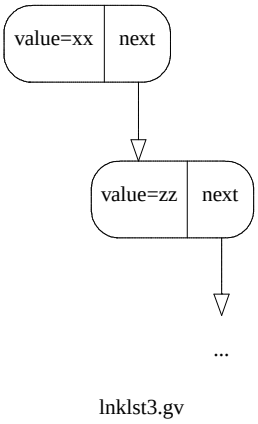
This routine adds a new node *on top* of the current head. The list becomes longer by one node.

```
subroutine cgca_addhead( head, payload )
type( cgca_lnklst_node ), pointer, intent( inout ) :: head
type( cgca_lnklst_tpayld ), intent( in ) :: payload
```

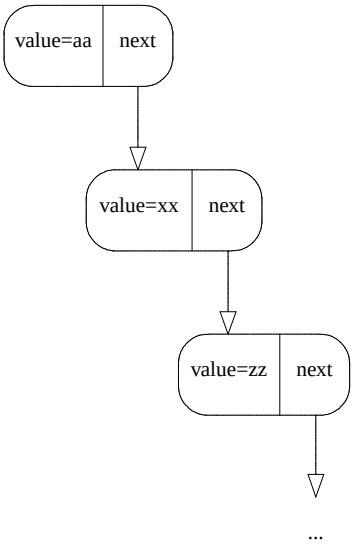
Pointer to the head node is given as input. Memory is allocated for one node. It's `value` is set to `payload`. It's `next` is pointed to the previous `head`, i.e. to `head%next`. The pointer to the new `head` is returned.

```
type( cgca_lnklst_node ), pointer :: tmp
allocate( tmp )
tmp%value = head%value
tmp%next => head%next
allocate( head )
head%value = payload
head%next => tmp
```

The action of this routine is illustrated schematically below. Imagine the `value` of the `head` node was `xx` before calling this routine. The list prior to calling this routine is shown on the left. The list after this routine was called with `value=aa` is shown on the right.



lnklst3.gv



lnklst4.gv

8.2.1.4.

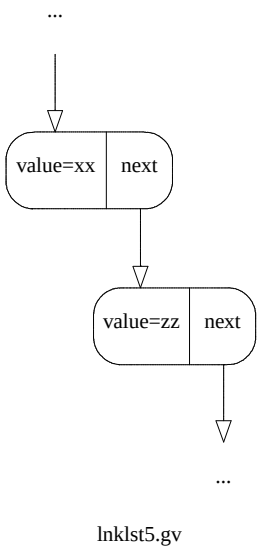
This routine adds a new node *below* the given node. The list becomes longer by one node.

```
subroutine cgca_addmiddle( node, payload )
type( cgca_lnklst_node ), pointer, intent( in ) :: node
type( cgca_lnklst_tpayld ), intent( in ) :: payload
```

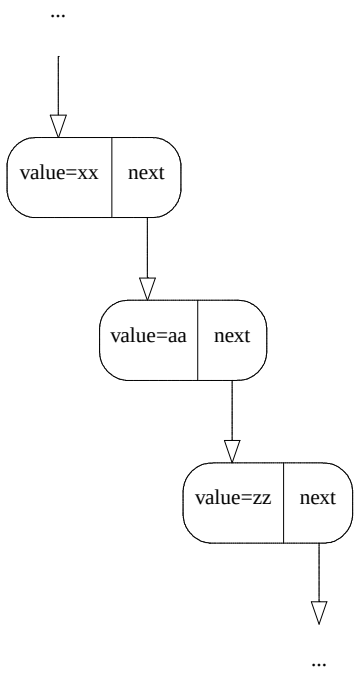
The pointer to an arbitrary `node` is given as input. Note that this does not need to be the head node. Memory is allocated for one node and its `value` set to `payload`. `node%next` points to the new node. The new node's `%next` points to where the `node%next` was pointing before calling this routine.

```
type( cgca_lnklst_node ), pointer :: tmp
allocate( tmp )
tmp%value = payload
 tmp%next => node%next
node%next => tmp
```

If the routine was called with `node` pointer set to pointing on the node with `value=xx`, and with `payload=aa`, then the diagrams below schematically illustrate the lists before calling this routine (left) and after (right).



lnklst5.gv



lnklst6.gv

8.2.1.5.

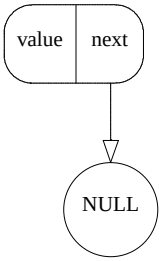
This routine is used to initialise the list, i.e. create the first node.

```
subroutine cgca_inithead( head, payload )
type( cgca_lnklst_node ), pointer, intent( out ) :: head
type( cgca_lnklst_tpayld ), intent( in ) :: payload
```

The routine allocates memory for one node, sets its `value` to `payload` and returns the pointer to this node in `head`. The node's `next` is set to `NULL`, i.e. the pointer is not associated:

```
allocate( head )
head%value = payload
head%next => null()
```

After calling this routine the linked list will look like this:



lnklst2.gv

8.2.1.6.

This routine dumps the `value` fields of all nodes on the list to stdout.

```
subroutine cgca_lstdmp( head )
type( cgca_lnklst_node ), pointer, intent( in ) :: head
```

The algorithm is to start with the head node and move downwards one node at a time until a node's `%next` is not associated.

```
  type( cgca_lnklst_node ), pointer :: tmp
  if ( .not. associated( head ) ) return
  tmp => head
  do
    write (*,*) tmp%value
    tmp => tmp%next
    if ( .not. associated( tmp ) ) exit
  end do
```

A typical output might look like this.

```
      40          31          59          49          39          77
      40          21          59          49          30          77
      40          21          40          58          39          58
      40           1          40          77          20          77
      40           1           1          77          39          39
       1           1           1          39          39          77
```

8.2.1.7.

This routine removes the head node. The node that was below the head becomes the head node. The list becomes shorter by one node.

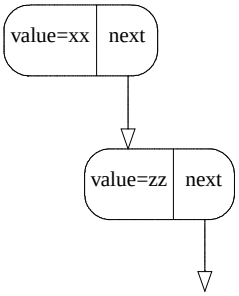
```
subroutine cgca_rmhead( head, stat )
type( cgca_lnklst_node ), pointer, intent( inout ) :: head
integer( kind=idef ), intent( out ) :: stat
```

Pointer to the `head` node is given as input. Pointer to `head%next` is returned as new `head`. Memory for the old `head` node is deallocated. On output `stat` is set to `1` if `head` is `NULL`, i.e. if `head` is not associated. Otherwise `stat` is set to `0`. If `head` is not associated on entry, then the routine just sets `stat` to `1` and exits. If the list consists of only a single node, then on exit `head` will be not associated. So `stat` will be set to `1`.

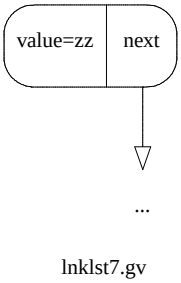
```
type( cgca_lnklst_node ), pointer :: tmp
stat = 0
if ( associated( head ) ) then
  tmp => head
  head => head%next
  deallocate( tmp )
end if
if ( .not. associated( head ) ) stat = 1
```

The action of this routine is illustrated schematically below.

First consider the case when the list is longer than a single node. Before calling this routine the `value` of the `head` node is `xx` (left diagram). After calling this routine the head node becomes the one with the `value` of `zz`. Since `head%next` is associated, `stat` is set to `0`.

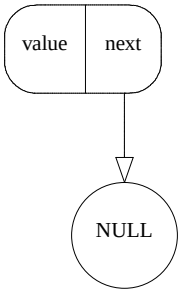


lnklst3.gv



lnklst7.gv

Now consider the case when the list consists of a single node. On input the list can be shown schematically as follows.



lnklst2.gv

On output, the `head%next` will be not associated, so `stat` will be set to `1`. The linked list is no more. If the list is required again, it has to be re-initialised with `cgca_inithead`.

8.2.1.8.

This routine removes a node *below* the given node.

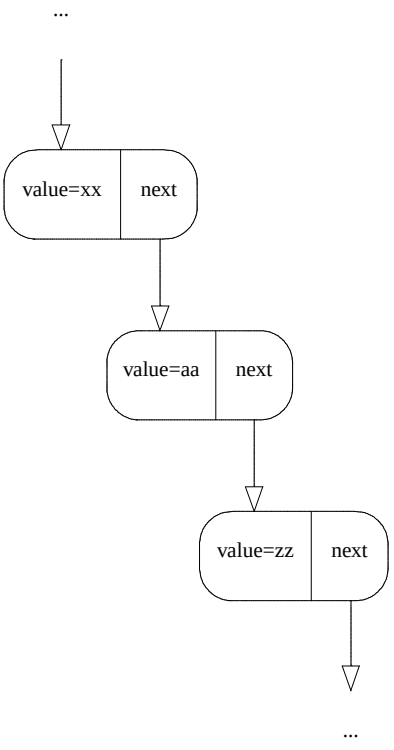
```
subroutine cgca_rmmiddle( node, stat )
type( cgca_lnklst_node ), pointer, intent( in ) :: node
integer( kind=idef), intent( out ) :: stat
```

The memory occupied by the removed node is freed. The `node%next` now points to where the node below `node%next` was pointing. If the node below the given `node` was `NULL`, i.e. if `node%next` was not associated, then `stat` is set to `1`. Otherwise `stat` is set to `0`.

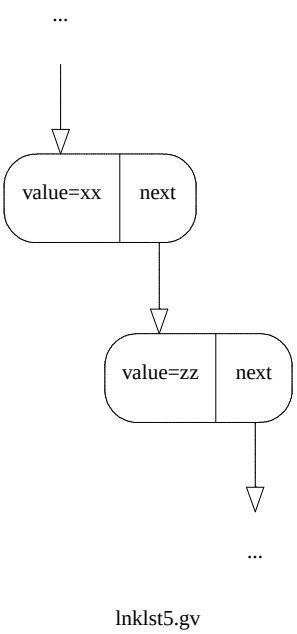
```
type( cgca_lnklst_node ), pointer :: tmp
stat = 0
tmp => node%next
if ( associated( tmp ) ) then
  node%next => tmp%next
  deallocate( tmp )
else
  stat = 1
end if
```

The action of this routine can be schematically illustrated below.

First consider the case when `node%next` is associated. If on input `node%value` is `xx`, then the list before calling this routine is shown on the left. The right diagram shows the list after a call to this routine. `stat` is set to `0`.

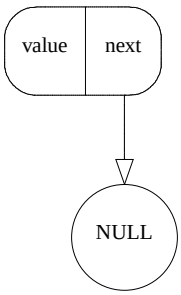


lnklst6.gv



lnklst5.gv

Now consider the case when the list consists of a single node, i.e. if the `node` is the head. On input the list can be shown schematically as follows.



lnklst2.gv

On output, the `head%next` will be not associated, so `stat` will be set to `1`. The linked list is no more. If the list is required again, it has to be re-initialised with `cgca_inithead`.

8.3.

8.3.1.

Module with routines for simulating cleavage propagation. The module contains a number of routines:

8.3.1.1.

This routine updates the grain boundary connectivity array `gc`. This is a private array in module `cgca_m2gb`. That module contains routines for analysing grain boundaries. Access to `gc` is only via routines of module `cgca_m2gb.`

Routine `cgca_gcupd` calls `cgca_gcf` from module `cgca_m2gb` to update `gc`.

The point of the grain connectivity array `gc` is to record the state of grain boundaries between grains. The boundary can be intact or fractured. The idea of the grain boundary array is a workaround the lack of accurate stress/strain redistribution across the microstructure. If an accurate physically sound algorithm for stress/strain redistribution has been implemented, then the grain connectivity array or routine `cgca_gcupd` would not be needed. Redistribution of stress/strain or other FE fields over CA is usually called *localisation*, a process where the coarse scale field is redistributed (localised) over finer scale (Kouznetsova, 2001).

However, at present, no good physically sound algorithm has been implemented in CGPACK. This means that local CA stresses are not updated quickly enough with each crack propagation step. This leads to situations where a grain boundary fracture on one image is not resulting in reduction/elevation of stress in neighbouring images. This can lead to situations where grain boundary between two grains is fractured multiple times on multiple images containing this grain boundary. This result is not physical.

The grain connectivity array, `gc`, is created to compensate for this lack of physics. At present `gc` is updated after each CA iteration by calling `cgca_gcupd`.

It is possible that grains are very large compared to coarrays on each image, e.g. much larger than the coarray size on each image. This would mean that a grain can span many images. If two adjacent grains are large, then the grain boundary between these grains can be present on many images. In this case the fact that the specific grain boundary has been fractured on some image might have to be propagated to many images *beyond* the nearest neighbours. The only way to be sure that the information reaches all relevant images is to communicate it to all images. Hence a very undesirable all-to-all communication pattern emerges.

Grain boundary failures are recorded by each image in its coarray array `gcupd`, which is defined in module `cgca_m3clvg`. At the end of every CA iteration, each image adds to its `gcupd` information on failed grain bondaries collected from other images.

The routine loops over all images, starting at a randomly chosen image. It reads `cgupd` array from each image into a local copy `gcupd_local:`

```
gcupd_local( : , : ) = gcupd( : , : ) [img_curr]
```

`gcupd` is reset to `cgca_gb_state_intact` at the start of each CA iteration. So when `gcupd` is analysed at the end of the CA iteration, only entries with `gcupd( : , 3 ) .ne. cgca_gb_state_intact` trigger a call of `cgca_gcf` to update `gc` on that image.

All-to-all does not scale. For that reason a simplified version of this routine is implemented in `cgca_gcupdn`, where the information is collected only from the nearest neighbouring images.