TACC Technical Report IMP-16

Processor-local code in the Integrative Model for Parallelism

Victor Eijkhout*

October 31, 2016

* eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin

Abstract

We discuss aspects of local operations in the Integrative Model for Parallelism.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a 'communication avoiding' compiler in the Integrative Model (under construction)
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
IMP-19 Report on NSF EAGER 1451204.
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design, with John McCalpin (under construction)
IMP-25 Dense linear algebra in IMP (under construction)

1 Introduction

The abstract description of the IMP model is based on data parallel application of functions

Datatype Func: functions with a single output

$$\text{Func} \equiv \text{Real}^k \to \text{Real}$$

Applying this to the computation of an array gives rise to the definition of a signature function: Let $x, y \in$ Array and let the function f compute each element of y from certain elements of x. We use a function σ (or σ_f to indicate its provenance explicitly) called the 'signature function' to determine the mapping from output indices to input sets of indices:

Datatype Signature: Signature of data parallel functions

$$\text{Signature} \equiv N \to \text{Ind}$$

Example 1 *For instance, in our motivational example of three-point averaging (*IMP-11*, section 3), y_i was computed from x_{i-1}, x_i, x_{i+1}, making $\sigma(i) = \{i-1, i, i+1\}$.*

Now, letting γ be the output distribution, the local function application on processor p becomes

$$\forall_{i \in \gamma(p)} : y_i = f\left(x_{\sigma(\gamma(p))}\right)$$

Here we see the first issue in generating the local code: the output index i may not correspond to local array index i.

- In shared memory, a process would actually write the result y_i to array location `y[i]`.
- In distributed memory, where each process has a local array, y_i would correspond to `y[i-my_first]`, where `my_first` describes the embedding of the local array in the global one.
- For multi-dimensional arrays this becomes more complicated, and the global-to-local mapping from the previous item becomes a more complicated function.

Mapping the input indices to array locations is even more tricky; see the example below. In summary we have:

The mapping problem: given the data parallel application of a function between distributed input and output vectors, how do we transform the input and output indices to address only the processor-local part of the arrays.

After this mapping problem, we then have the traversal problem: above we only specified the execution with a 'for-all' enumeration. The exact traversal of the output index set can have serious

performance ramifications. Matters such as keeping intermediate results in register, and optimal use of cachelines become an issue here.

2 Uniform indexing

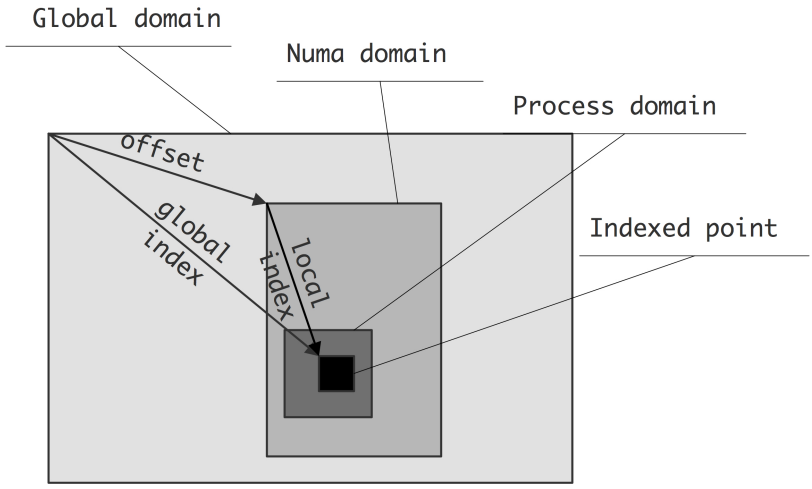

Figure 1: Illustration of the various indexing concepts

We want to do all indexing in global numbering. In the sequential case that makes sense, and probably even in the shared memory case. However, as soon as we introduce distributed memory it becomes harder. We then have to compute a point (i, j, k) in memory relative to (figure 1):

- The global domain: this is the domain of definition of the operation, typically an interval $0 \ldots N-1$ in each direction.
- Numa domain: this is the set of all indices that a process has access to.
- Process domain: the set of all indices that a process owns and where it computes the output.

Two limiting cases are immediately clear:

- In MPI distributed memory, there is only one process running on each address space, so the numa domain and process domain coincide; figure 2.
- In OpenMP shared memory the global domain and numa domain coincide because there is only one address space; figure 3.

Thus, the we use mode-independent macros for translating a global index to a local one in the numa space:

```
%% imp_functions.h
#define INDEX1D( i,offsets,nsize ) \
  i-offsets[0]
#define INDEX2D( i,j,offsets,nsize ) \
```

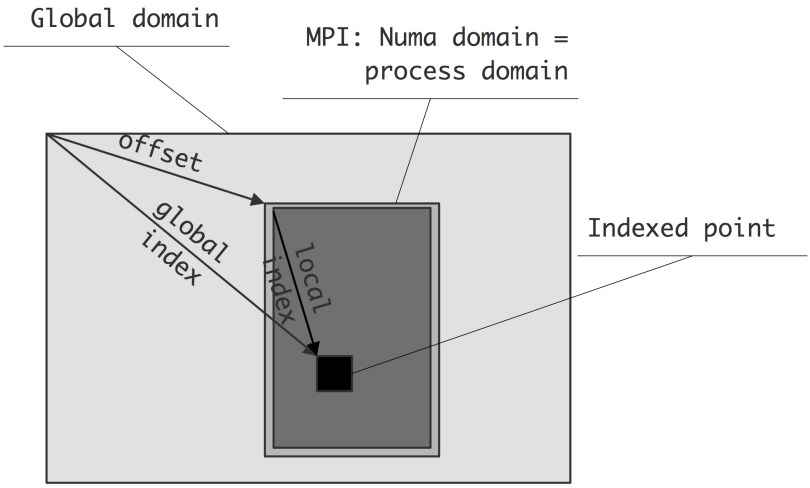

Figure 2: Illustration of the various indexing concepts in MPI distributed memory

```
  (i-offsets[0])*nsize[1]+j-offsets[1]
#define INDEX3D( i,j,k,offsets,nsize ) \
  ( (i-offsets[0])*nsize[1]+j-offsets[1] )*nsize[2] + k-offsets[2]
#define COORD1D( i,gsize ) \
  ( i )
#define COORD2D( i,j,gsize ) \
  ( (i)*gsize[1] + j )
#define COORD3D( i,j,k,gsize ) \
  ( ( (i)*gsize[1] + j )*gsize[2] + k )
```

These macros can be optimized away by any competent compiler.

The above leads to the following definitions of memory offsets for an operation with an `invector` and an `outvector`:

```
%% imp_functions.cxx
multi_indexstruct
  *in_nstruct = invector->get_numa_structure(),
  *out_nstruct = outvector->get_numa_structure(),
  *in_gstruct = invector->get_global_structure(),
  *out_gstruct = outvector->get_global_structure();
domain_coordinate
  in_nsize = in_nstruct->local_size_r(), out_nsize = out_nstruct->local_size_r(),
  in_offsets = invector->offset_vector(),
  out_offsets = outvector->offset_vector();
```

See for instance a copy loop in two dimensions:

```
%% imp_functions.cxx
for (index_int i=pfirst[0]; i<=plast[0]; i++) {
```

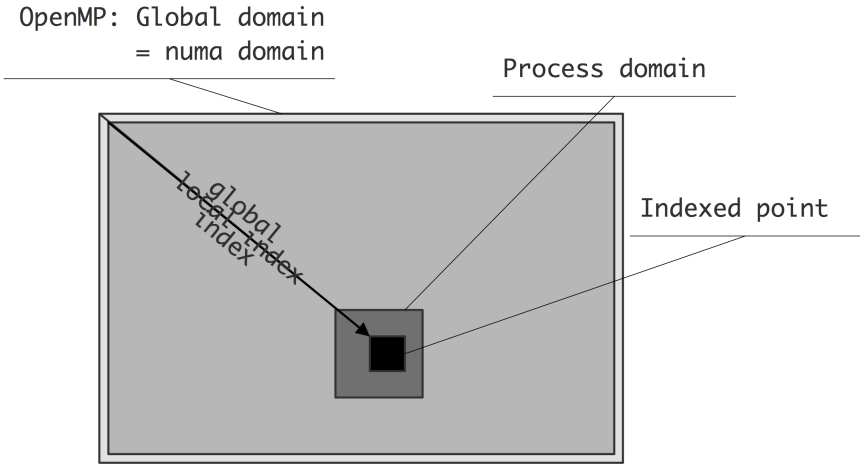

Figure 3: Illustration of the various indexing concepts in OpenMP shared memory

```
for (index_int j=pfirst[1]; j<=plast[1]; j++) {
  index_int IJ = INDEX2D(i,j,out_offsets,out_nsize);
  if (!done) {
    fmt::print("{} copy: {}\n",p->as_string(),indata[IJ]); done = 1; }
  outdata[IJ] = indata[IJ];
}
}
```

The distinction between the parallelism modes is achieved by an overloaded function. In MPI it is the processor structure of 'this' process:

```
%% mpi_base.cxx
compute_numa_structure = [this] (distribution *d) -> void {
  processor_coordinate *pcoord = d->coordinate_from_linear(procid());
  multi_indexstruct *numa_struct = d->get_processor_structure(pcoord)->make_clone();
  d->set_numa_structure( numa_struct );
};
```

For OpenMP it is the enclosing structure:

```
%% omp_base.cxx
compute_numa_structure = [this] (distribution *d) -> void {
  multi_indexstruct *numa_struct = d->get_enclosing_structure();
  d->set_numa_structure( numa_struct,numa_struct->first_index(),numa_struct->volume() );
};
```

3 Message structures

Find the parts of an α-distribution that make up a given `beta_block`:

```
%% imp_base.cxx
std::vector<message*> *distribution::messages_for_segment
    ( processor_coordinate *mycoord,self_treatment doself,
      multi_indexstruct *beta_block,multi_indexstruct *halo_struct ) {
  int mytid = mycoord->coord(0); int dim = get_dimensionality();
  int P = this->get_global_ndomains();
  processor_coordinate *farcorner = get_farcorner();
  std::vector<message*> *messages = new std::vector<message*>;
  messages->reserve(P);
  multi_indexstruct *buildup = new multi_indexstruct(dim);
  int pstart = mytid; if (has_random_sourcing()) pstart += rand()%P;
  for (int ip=0; ip<P; ip++) {
    int p = (pstart+ip)%P; // start with self first, or random
    if (!lives_on(p)) continue; // deal with masks
    processor_coordinate *pcoord = coordinate_from_linear(p);
    if (doself==self_treatment::ONLY && !mycoord->equals(pcoord))
      continue;
    multi_indexstruct *pstruct = get_processor_structure(pcoord);
    multi_indexstruct *mintersect = beta_block->intersect(pstruct);
    if (!mintersect->is_empty() && !(doself==self_treatment::EXCLUDE && p==mytid)
        && !buildup->contains(mintersect) ) {
      multi_indexstruct *simstruct = mintersect->force_simplify();
      message *m = new_message(pcoord,mycoord,simstruct);
      if (beta_has_local_addres_space)
m->relativize(halo_struct);
      messages->push_back( m );
      buildup = buildup->struct_union(mintersect);
      if (buildup->contains(beta_block)) goto covered;
    }
  }
```

where the beta block comes from a β-distribution:

```
%% imp_base.cxx
multi_indexstruct
  *beta_block = beta_dist->get_processor_structure(pid),
  *halo_block = beta_dist->get_numa_structure(); // to relativize against
try {
  self_treatment doself;
  if ( invector->has_collective_strategy(collective_strategy::MPI)
   && d->get_is_collective() )
    doself = self_treatment::ONLY;
  else doself = self_treatment::INCLUDE;
  auto msgs = alpha_distro->messages_for_segment( pid,doself,beta_block,halo_block );
```

For MPI, the message is relativized again the processor structure of the β-distribution:

```
%% mpi_base.cxx
get_visibility = [this] (processor_coordinate *p) -> multi_indexstruct* {
```

```
  return get_processor_structure(p); };
```

For OpenMP it's the whole structure:

```
%% omp_base.cxx
get_visibility = [this] (processor_coordinate *p) -> multi_indexstruct* {
  return this->get_enclosing_structure(); };
```

The relativize methods are separately implemented for each type of `indexstruct`.

```
%% imp_base.cxx
void message::relativize(multi_indexstruct *container) {
  local_struct = global_struct->relativize(container);
};
```

Thus the `globalstruct` of a message is in global indexing, while the `localstruct` can be used for indexing in the actual array.

If the beta sticks out, we use an 'embed message'

```
%% imp_base.cxx
processor_coordinate *pleft = mycoord->left_face_proc(id,farcorner);
intersect = beta_block->intersect
  (get_processor_structure(pleft)->operate(new shift_operator(id,g)));
if ( !intersect->is_empty()
     && !(doself==self_treatment::EXCLUDE && pleft->equals(mycoord)) ) {
  buildup = buildup->struct_union(intersect);
  message *m = new_embed_message
    (pleft,mycoord,intersect,intersect->operate(new shift_operator(id,-g)));
  if (beta_has_local_addres_space) {
    multi_indexstruct *tmphalo = halo_struct->operate(new shift_operator(id,-g));
    m->relativize(tmphalo);
  }
  messages->push_back(m);
}
```

3.1 MPI

3.1.1 Send message

We send elements from an α-structure.

Remark 1 *Note: the α can be embedded in a beta, but retain its own numbering in 1D. In higher dimensions that is wrong.*

For MPI, the message uses the MPI 'subarray' type, which handles any dimension elegantly:

```
%% mpi_base.cxx
MPI_Isend( data,1,mmsg->embed_type,receiver_no,msg->get_tag_value(),comm,&req);
```

```
%% mpi_base.cxx
void mpi_message::compute_src_index() {
  message::compute_src_index();
  int
    ortho = get_in_object()->get_orthogonal_dimension(),
    dim = get_in_object()->get_dimensionality();
  int err;
  err = MPI_Type_create_subarray(dim+1,numa_sizes,struct_sizes,struct_starts,MPI_ORDER_C,MPI_DOUBLE,&er
  if (err!=0) throw(fmt::format("Type create subarray failed for src: err={}",err));
  MPI_Type_commit(&embed_type);
  if (dim==1) {
    lb = struct_starts[0]; extent = struct_sizes[0];
  } else if (dim==2) {
    lb = struct_starts[0]*numa_sizes[1]+struct_starts[1];
    extent = (struct_starts[0]+struct_sizes[0]-1)*numa_sizes[1] + struct_starts[1]+struct_sizes[1] - lb
  } else
    throw(fmt::format("Can not compute lb/extent in dim>2"));
  if (extent<=0)
    throw(fmt::format("Zero extent in message <<{}>>",as_string()));
};
```

based on

```
%% imp_base.cxx
void message::compute_src_index() {
  if (src_index!=-1)
    throw(fmt::format("Can not recompute message src index in <<{}>>",get_name()));
  try {
    multi_indexstruct
      *send_struct = get_global_struct(), // local ???
      *proc_struct = get_in_object()->get_processor_structure(get_sender());
    src_index = send_struct->linear_location_in(proc_struct);
  } catch (std::string c) { throw(fmt::format("Error <<{}>> setting src_index",c)); }

  {
    multi_indexstruct
      *outer = get_in_object()->get_numa_structure(),
      *inner = get_global_struct();
    int
      ortho = get_in_object()->get_orthogonal_dimension(),
      dim = outer->get_dimensionality();
    try {
      compute_subarray(outer,inner,ortho);
    } catch (std::string c) { fmt::print("Error <<{}>> computing subarray\n",c);
      throw(fmt::format("Could not index <<{}>> in <<{}>>",
inner->as_string(),outer->as_string())); }
  }
};
```

and

```
%% imp_base.cxx
void message::compute_subarray(multi_indexstruct *outer,multi_indexstruct *inner,int ortho) {
  int dim = outer->get_same_dimensionality(inner->get_dimensionality());
  numa_sizes = new int[dim+1]; struct_sizes = new int[dim+1]; struct_starts = new int[dim+1];
  for (int id=0; id<dim; id++) {
    numa_sizes[id] = outer->local_size()->at(id);
    struct_sizes[id] = inner->local_size()->at(id);
    annotation.write(" {}:{}@{}in{}",id,struct_sizes[id],struct_starts[id],numa_sizes[id]);
  }
  numa_sizes[dim] = ortho; struct_sizes[dim] = ortho; struct_starts[dim] = 0;
};
```

3.1.2 Receive message

The receive message writes into a β structure.

```
%% mpi_base.cxx
MPI_Irecv( data,1,mmsg->embed_type,sender_no,msg->get_tag_value(),comm,&req);
```

using

```
%% mpi_base.cxx
void mpi_message::compute_tar_index() {
  message::compute_tar_index();
  int
    ortho = get_out_object()->get_orthogonal_dimension(),
    dim = get_out_object()->get_dimensionality();
  int err;
  err = MPI_Type_create_subarray(dim+1,numa_sizes,struct_sizes,struct_starts,MPI_ORDER_C,MPI_DOUBLE,&er
  if (err!=0) throw(fmt::format("Type create subarray failed for src: err={}",err));
  MPI_Type_commit(&embed_type);
  lb = struct_starts[dim]; // compute analytical lower bound and extent
  extent = struct_sizes[dim];
  for (int id=dim-1; id>=0; id--) {
    lb += struct_starts[id]*numa_sizes[id+1];
    extent += (struct_sizes[id]-1)*numa_sizes[id+1];
  }
  {
    MPI_Aint true_lb,true_extent;
    MPI_Type_get_true_extent(embed_type,&true_lb,&true_extent);
    if (true_lb!=lb*sizeof(double))
      throw(fmt::format("Computed lb {} mismatch true lb {}",lb,true_lb));
    if (true_extent!=extent*sizeof(double))
      throw(fmt::format("Computed extent {} mismatch true extent {}",extent,true_extent));
  }
  if (extent<=0)
    throw(fmt::format("Zero extent in message <<{}>>",as_string()));
```

```
};
```

(Note the `embed_structure`, which is equal to the local structure, except when the halo sticks out.)

Again, we have a basic routine

```
%% imp_base.cxx
void message::compute_tar_index() {
  if (tar_index!=-1)
    throw(fmt::format("Can not recompute message tar index in <<{}>>",get_name()));
  tar_index = get_local_struct()->linear_location_in
    ( get_out_object()->get_processor_structure(get_receiver()) );

  multi_indexstruct
    *outer = get_out_object()->get_processor_structure(get_receiver()),
    *inner = get_embed_struct();
  int
    ortho = get_out_object()->get_orthogonal_dimension(),
    dim = outer->get_dimensionality();
  compute_subarray(outer,inner,ortho);
};
```

3.2 OpenMP

In OpenMP, we 'post a receive' by adding a request object to a task.

```
%% omp_base.cxx
void omp_task::acceptReadyToSend
    ( std::vector<message*> *msgs,request_vector *requests ) {
  if (msgs->size()==0) return;
  if (find_other_task_by_coordinates==nullptr)
    throw(fmt::format("{}: Need a task finding function",get_name()));
  for ( auto m : *msgs ) {
    auto sender = m->get_sender(), receiver = m->get_receiver();
    object *halo = m->get_out_object();
    halo->set_data_is_filled(receiver);
    halo->compute_enclosing_structure();
    int instep = m->get_in_object()->get_object_number();
    requests->add_request
      ( new omp_request( find_other_task_by_coordinates(instep,sender),m,halo ) );
  }
};
```

The actual work is then done in the 'wait' call:

```
%% omp_base.cxx
void omp_request_vector::wait() {
  int outstanding = size();
  for ( ; outstanding>0 ; ) { // loop until all requests fullfilled
```

```
    for ( auto r : requests ) {
      omp_request *req = dynamic_cast<omp_request*>(r);
      if (req==nullptr) throw(std::string("could not upcast to omp request"));
      if (!req->closed) {
        task *t = req->tsk; object *task_object;
        if (!t->get_has_been_executed())
          try {
            t->execute();
          } catch (const char *c) { fmt::print("Error <<{}>>\n",c);
            throw(fmt::format("Could not execute {} as dependency",t->get_name()));
  }
omp_message *recv_msg = dynamic_cast<omp_message*>(req->msg);
if (recv_msg==nullptr)
  throw(std::string("Could not upcast to omp_message"));
message *send_msg = recv_msg->send_msg;
if (send_msg==nullptr)
  throw(fmt::format("Can not find send message for <<{}>>",req->msg->as_string()));
        try {
          task_object = t->get_out_object();
        } catch (const char *c) { printf("Error <<%s>>\n",c);
          throw( fmt::format("Could not get out data from dependent task {}",t->get_name())); }
        req->obj->copy_data_from( task_object,send_msg,recv_msg );
        req->closed = 1;
        outstanding--;
      }
    };
  }
};
```

using an elaborate copy routine

```
%% omp_base.cxx
void omp_object::copy_data_from( object *in,message *smsg,message *rmsg ) {
  if (has_data_status_unallocated() || in->has_data_status_unallocated())
    throw(std::string("Objects should be allocated by now"));
  object *out = this;

  auto p = rmsg->get_receiver(), q = rmsg->get_sender();
  int dim = get_same_dimensionality( in->get_dimensionality() );
  double
    *src_data = in->get_data(q), *tar_data = out->get_data(p);
  multi_indexstruct
    *src_struct = smsg->get_local_struct(), *tar_struct = rmsg->get_local_struct(),
    *src_gstruct = smsg->get_global_struct(), *tar_gstruct = rmsg->get_global_struct();
  domain_coordinate
    *struct_size = tar_struct->local_size();

  int k = in->get_orthogonal_dimension();
```

```
  if (k>1)
    throw(std::string("copy data too hard with k>1"));

  domain_coordinate
    pfirst = tar_gstruct->first_index_r(), plast = tar_gstruct->last_index_r(),
    qfirst = src_gstruct->first_index_r(), qlast = src_gstruct->last_index_r();

  multi_indexstruct
    *in_nstruct = in->get_numa_structure(),
    *out_nstruct = out->get_numa_structure(),
    *in_gstruct = in->get_global_structure(),
    *out_gstruct = out->get_global_structure();
  domain_coordinate
    in_nsize = in_nstruct->local_size_r(), out_nsize = out_nstruct->local_size_r(),
    in_offsets = in_nstruct->first_index_r() - in_gstruct->first_index_r(),
    out_offsets = out_nstruct->first_index_r() - out_gstruct->first_index_r();

  if (dim==0) {
  } else if (dim==2) {
    if (src_struct->is_contiguous() && tar_struct->is_contiguous()) {
      int done = 0;
      for (index_int isrc=qfirst[0],itar=pfirst[0]; itar<=plast[0]; isrc++,itar++) {
for (index_int jsrc=qfirst[1],jtar=pfirst[1]; jtar<=plast[1]; jsrc++,jtar++) {
  index_int
    Iout = INDEX2D(itar,jtar,out_offsets,out_nsize),
    Iin = INDEX2D(isrc,jsrc,in_offsets,in_nsize);
  if (!done) {
    fmt::print("{}->{} copy data {} between {}->{}\n",
       q->as_string(),p->as_string(),src_data[ Iin ],Iin,Iout);
    done = 1; }
  tar_data[ Iout ] = src_data[ Iin ];
}
      }

    } else {
      throw(std::string("omp 2d copy requires cont-cont"));
    }
  } else if (dim==1) {
    multi_indexstruct *local = rmsg->get_local_struct(), *global = rmsg->get_global_struct();
    if (global->is_contiguous() && local->is_contiguous()) {

      for (index_int i=pfirst[0]; i<=plast[0]; i++) {
tar_data[ INDEX1D(i,out_offsets,out_nsize) ] =
  src_data[ INDEX1D(i,in_offsets,in_nsize) ];
      }
```

```
    } else {
      index_int localsize = local->volume();
      index_int len = localsize*k;

      auto
src_struct = global->get_component(0), tar_struct = local->get_component(0);
      index_int
        src0 = in->get_enclosing_structure()->linear_location_of(local),
        tar0 = this->get_enclosing_structure()->linear_location_of(global);
      if (src_struct->is_contiguous() && tar_struct->is_contiguous()) {
        throw(std::string("This should have bee done above"));
      } else if (tar_struct->is_contiguous()) {
        index_int itar=tar0;
        for ( auto isrc : *src_struct )
          tar_data[itar++] = src_data[isrc];
      } else {
        for (index_int i=0; i<len; i++)
          tar_data[ tar_struct->get_ith_element(i) ] = src_data[ src_struct->get_ith_element(i) ];
      }
    }
  } else
    throw(std::string("Can not omp copy in other than 1-d or 2-d"));
};
```

4 More examples

```
%% unittest_functions.cxx
if (tar0+len-1>gsize-1) // omit elements that don't exist
  len--;
for (index_int i=0; i<len; i++) {
    outdata[tar0+i] = indata[src0+i+1];
}
```