Coven Module Writer's Guide

Parallel Architecture Research Laboratory
version 1.1

Nathan DeBardeleben

February 13, 2003

Contents

1 Introduction

This document is intended to serve as a manual on how to write modules for Coven. Coven is a product of the Parallel Architecture Research Laboratory at Clemson University.

2 Module Concepts

A module in Coven is a single function with inputs and outputs and is contained within a single C file. The file may have multiple C functions that are called from the Coven module function but only one module may reside inside of each file.

Arguments are passed into and out of functions through specific constructs which are defined in this document. As such, the normal prototype of a function:

```
int function(int arg1, char **arg2, float arg3, double *arg4)
```

will appear very different in Coven.

Modules are self contained. It is important when you write modules to not make any assumptions about what other modules this one will operate with. This is because those modules may be replaced by other modules and as long as the interface between them (inputs and outputs) remain constant the program writer will expect the behavior to remain the same. The module writer should focus on making the module perform a specific, detailed task.

3 Coven Data Types

The following basic data types are defined in Coven. You can make scalar or buffers of these types:

- char
- unsigned char
- short
- unsigned short
- int
- unsigned int
- float
- double
- string

When creating data of these types they should be written exactly as above. For instance, `uchar` or `char **` are invalid types in Coven.

4 Coven Type Modifiers

All types must be modified in Coven. The following are possible Coven modifiers:

- const
- input

- output
- inout

The modifiers have slightly different meanings depending on whether they are modifying scalar values or buffers.

4.1 Scalar Modifiers

4.1.1 const Modifier

The `const` modifier is only for input scalar values. A special portion of the Coven program (see Coven Program Writer's Guide) is used to define constants and their values before program execution.

While Coven does nothing directly to keep a programmer from modifying a constant value, the action of modifying any constant is undefined. In general it can be assumed that any change to a constant will not be seen outside of that module, even during multiple executions of the same module.

Examples of `const` types are things like:

- the number of grid points in a simulation
- the maximum temperature
- the address of a machine to visualize the data on

In C, a Coven const might look something like:

```
const float max_temperature;
```

4.1.2 input Modifier

The `input` modifier is used to get a copy of a value that can not be modified. This is similar to a `const` but differs in that this value was not known at the start of a program. Therefore, to use the `input` modifier some module before hand must have created the required datum.

Examples of `input` types are things like:

- the current iteration
- the number of grid points processed so far
- the next portion of a buffer to search

In C, a Coven input might look something like:

```
int cur_iteration;
```

4.1.3 output Modifier

The `output` modifier is used create a value that will be set at some point in the module. This value can be used by later modules by reading it with an `input` modifier.

When the programmer asks for an `output` type they will be handed a pointer to the variable after it has been created. Therefore, to set the value the programmer must access it as they would any C pointer. It is helpful to think of the `output` type as having been passed by reference.

Examples of `output` types are things like:

- the current iteration
- the number of grid points processed so far
- the next portion of a buffer to search

Notice that these are the same examples as from `input` because they must first be output before they can be read as input.

In C, a Coven output might look something like:

```
int * cur_iteration;
. . .
*cur_iteration = 0; /* set it to zero */
```

4.1.4 inout Modifier

The `inout` modifier is used to access a value that *has previously been created* with an `output` call. This differs from an `input` type in that the programmer is returned the variable as a pointer. With this pointer, the programmer can read the value by dereferencing it, or set it by also dereferencing it.

Examples of `inout` types are things like:

- the current iteration
- the number of grid points processed so far
- the next portion of a buffer to search

In C, a Coven output might look something like:

```
int * cur_iteration;
. . .
*cur_iteration = *cur_iteration + 1; /* increment by one */
```

4.2 Buffer Modifiers

4.2.1 const Modifier

Constant buffers do not make any sense in Coven so you cannot use this modifier.

4.2.2 input and inout Modifiers

The `input` and `inout` modifiers both mean the same thing in Coven. Coven does not maintain any consistency to prevent module programmers from modifying `input` typed data. The rule of thumb that must be followed by module programmers, however, is that any buffer data that is modified within the module should be declared as `inout`.

The distinction is particularly important to Coven because `inout` data imposes a data dependency on the dataflow graph used in the graphical interface. The reason for this is that data that is declared as `input` only Coven reserves the right to perform optimizations on the data, such as replicating the buffer to separate modules. If the modules were to modify the data and expect the results to propagate further then undefined behavior may result.

4.2.3 output Modifiers

The `output` modifier instructs Coven to create a buffer of the specified type and size. Size here is defined as the *number of elements*, not the number of bytes. A buffer of the requested size and type will be created and returned so that it can be accessed and used throughout the module.

4.3 static Modifier

In addition to the above modifiers a type may be modified further with the use of the `static` modifier. It is important to note that `static` must be used in conjunction with the other modifiers input, output, and inout. It does *not* make sense to modify `const` with `static`.

The `static` modifier allows data to be created or access *that will only be seen by this module and instances of this module running on the same process.* This is a very important distinction. Static variables will *not* be propagated to other modules. Therefore, it is likely that the only acceptable modifier to `static` is the `inout` type modifier.

An example of the use of a `static` modifier is when performing socket operations through a visualization. For example, new sockets are assigned a file descriptor `int` value. That value is used for all socket transactions until the socket is closed. If we were to write a module to send visualization data across a socket it would be to our advantage to leave the socket open for as long as possible. Additionally, assume that we were sending visualization data every iteration, perhaps over thousands of iterations. In Coven, the best way to handle this would be through a `static inout` variable. In this way, the first time the module was called we could notice the file descriptor was -1 (unopen), open the connection, and assign the socket ID to the `static inout` variable. Then, when this module was called on another iteration the socket ID could be read from this `static` variable.

In the above example it would additionally be possible to create the value through a basic `inout` modified variable but would unduly bother other modules by exposing this variable to them when it was only needed by this module.

5 Variable Names / Local Bindings

Each Coven variable obviously requires a variable name. This will only be the variable name of the data within the module. Other modules may refer to the same variable by a different name.

For example, we have a module, `ModuleA`, which declares an output scalar value and sets it to zero with the following code:

```
output int * running_sum;
. . .
*running_sum = 0;
```

We can have another module, `ModuleB`, which increments the value. However, we are in no way constrained in `ModuleB` to reference the variable as `running_sum`. For example, `ModuleB` could do:

```
inout int * sum;
. . .
*sum = *sum + 1;
```

In Coven we refer to the symbols `running_sum` and `sum` as *local bindings.* These are the local (to the module) binding of the variable names. This will be important to understand when you place modules together to form Coven programs.

NOTE: No variable you create should ever be prepended with _ or __. These are reserved for Coven special variables and the variable symbols may be placed in locations you are unaware and could result in *extremely* difficult to debug behavior.

6 Buffers and Buffer Sizes

In Coven, when you create a buffer you must specify the size (in elements) of the buffer. This size may be any valid C expression that resolves to an `int`. Therefore, it is possible to do something like:

```
input int number_of_gridpoints,
output buffer float gridpoints[number_of_gridpoints],
output buffer float example[number_of_gridpoints/2],
```

In the above example two buffers would be created. If `number_of_gridpoints` were set to `100`, the buffer `gridpoints` would have size `100` and the buffer `example` would have size `50`.

The above example was only for `output` buffers. For `input` and `inout` buffers the buffer has already been created so setting a size does not make sense. However, for these types the programmer must specify a variable name to store the size (number of elements) of the buffer. This allows the programmer to very easily not only get a reference to the buffer but also get the size of the buffer, as in this example:

```
input buffer float gridpoints[number_of_gridpoints],
inout buffer float example[number_of_example],
```

In the above example the buffers `gridpoints` and `example` would not be created, they would be accessed. Additionally, two integer variables will be created automatically, `number_of_gridpoints` and `number_of_example`, which would be set to `100` and `50` respectively (assuming the first `output` buffer example were as stated).

7 Putting it all Together

The module header and entire module must be of the form:

```
COVEN_Module module_name
  (
   argument1,
   argument2,
   . . .
   argumentN
  )
{
  . . .
  . . . /* module code */
  . . .
}
```

You must assign each module a module name and replace the above `module_name` with that name. The `argument1-N` in the above example must conform to the constraints outlined above. It is here that the scalars and buffers are accessed and created.

Scalar arguments are of the form:

```
[static] <const/input/output/inout> <type> <name>
```

The `static` modifier is optional, as described above. One additional modifier must be present (`const`, `input`, `output`, or `inout`). The type and name must conform to the style described earlier. An example of scalar arguments are:

```
static inout int * visualization_socket,
const float timestep,
output double * temperature
```

Buffer arguments are of the form:

```
<input/output/inout> buffer <type> <name>[<size_name>]
```

Notice the required keyword `buffer`. Examples of buffer arguments are:

```
output buffer float gridpoints[number_of_gridpoints],
input buffer int temperature[size_of_temp_matrix]
```

See the appendix for a good example of a simple module that uses most of the concepts discussed in this document.

8 User Defined Data Types

In Section 4 we declared the primitive types. Coven allows any string to go through the module parser because it assumes that the type will be correctly defined and interpreted on the back-end side.

The first step in setting up a new type is to go to the directory `CovenDriver/src/PSE` directory. Define your type in a new file in the `include` directory. Here is an example from the `medea_structs.h` file:

```
/* a generic vector struct */
typedef struct Medea_Vector {
        double x, y, z;
} Medea_Vector;

/* a simple struct for what makes up a body */
typedef struct Medea_Body {
        int id;
        double mass;
        Medea_Vector position;
        Medea_Vector velocity;
        Medea_Vector force;
} Medea_Body;
```

Notice that `Medea_Body` is the real type we are defining and the one that describes a body in our system. This is the one that we want to expose to Coven so that we can create buffers and scalars of this type.

We expose this to Coven as a PSE (problem solving environment) type. Take a look at `medea.h`:

```
#include "medea_structs.h"
typedef Medea_Body * MEDEA_BODY_TYPE;
#define MEDEA_BODY 0 + PSE_TYPE_OFFSET
```

Here we define a new type `MEDEA_BODY`. Additionally, we must give a unique integer value to each of our types that we will use. This value must start from `PSE_TYPE_OFFSET`.

Next, to tell Coven *which* set of types (and therefore PSE) we are interested in using at this time we change the file `PSE_types.h` to include the chosen header, such as:

```
#include "medea.h"
```

Finally, we go to the `CovenDriver/src/PSE/src` directory and modify `PSE_types.c` so that it recognizes the size (in bytes) of the new type we defined. Here is the content of the file for this example:

```
#include "coven.h"
#include "PSE_types.h"

int PSE_TypeSizeArray[] = {
        sizeof(Medea_Body),
};

int PSE_get_type_size(int which)
{
        return PSE_TypeSizeArray[which-PSE_TYPE_OFFSET];
}
```

The only change we made was to include the `sizeof` operation for the only type that we defined.

One very important note. Coven requires that all types defined such that the typedef of the type itself is of the form `TYPE_NAME_TYPE` and that in modules you reference the type as `TYPE_NAME`. This would be like defining a float type as `float_TYPE` but when we use it calling it like:

```
float temperature
```

An example of the body type appears in the appendix and the above PSE changes all are present in the PSE subdirectory of the source tree.

9 Advanced Concepts

There are many advanced concepts which are available to Coven module writers. Coven's philosophy is to keep the advanced concepts from being used by average module writers. Scientific users should never need the following knowledge. These ideas should be left only to the advanced system module programmer. If there seems to be a need for these concepts in other modules please bring this to the attention of the team so that these issues can be addressed and resolved.

9.1 Scatter and Gather

Coven has scatter and gather routines built in that are callable from modules. These were modeled after MPI routines and MPI datatypes. An understanding of MPI datatypes is required to use Coven scatter and gather methods.

These methods can best be described by example.

9.1.1 Scatter Example

The following module is an example of a scatter module. The basic idea is that a buffer of bodies (`my_bodies`) of size `total_num_bodies` is passed into this module and this buffer will be scattered into `num_slave_procs` smaller buffers. The module uses a Coven internal call, `COVEN_get_num_slave_processor` to get the number of slave processors that the job is currently using.

We use MPI-style datatype construction calls to set up a `body_type` type which will hold the elements of our buffer when transmitted by Coven (and MPI at a lower level). We chose here to create an indexed type which for each element of the indexed array specifies how many bodies are in the segment and what the offset from the first body will be. Coven implements *many* (read: nearly all to more than) MPI typing routines. This allows for extraordinarily complex data decomposition.

For each buffer we want to scatter we must create a `COVEN_Collective_Action`. We specify here to create a scatter action and give it the previously created scatter type and the buffer name to scatter. This name is the local variable name (local binding) of the buffer. Finally, we execute the scatter by telling Coven the number of tasks to scatter to (this is the number of TPHs that are created) and the collection actions to perform on them (as well as the number of collective operations, `1` in the following example).

The scatter example follows.

```
COVEN_MODULE nbody_scatter
  (
   inout buffer MEDEA_BODY my_bodies[total_num_bodies]
  )
{
        int num_slave_procs = COVEN_get_num_slave_processors();
        int target_num_bodies = (int)ceil((double)total_num_bodies /
          (double)num_slave_procs);

        int i, total_assigned_bodies;
        COVEN_size *array_of_blocks, *array_of_displacements;
        COVEN_Type body_type, scatter_type;

        COVEN_Type_contiguous(sizeof(struct Medea_Body), COVEN_BYTE,
          &body_type);

        total_assigned_bodies = 0;
        i = 0;

        array_of_blocks = (COVEN_size*)malloc(num_slave_procs*
          sizeof(COVEN_size));
        array_of_displacements = (COVEN_size*)malloc(num_slave_procs*
          sizeof(COVEN_size));

        while(total_assigned_bodies < total_num_bodies) {
            /* try and fit this many bodies */
```

```
            int num_bodies = target_num_bodies;
            /* starting at how many we've assigned so far */
            int start_body = total_assigned_bodies;

            total_assigned_bodies += num_bodies;
            /* if we've assigned more than we have, then back off by that
             * amount */
            if(total_assigned_bodies > total_num_bodies)
                num_bodies -= (total_assigned_bodies - total_num_bodies);

            /* now num_bodies is the number of bodies for this TPH to take */
            array_of_blocks[i] = num_bodies;
            array_of_displacements[i] = start_body;
            i++;
        }

        COVEN_Type_indexed(num_slave_procs, array_of_blocks,
          array_of_displacements, body_type, &scatter_type);
        {
            struct COVEN_Collective_Action *col_action;

            col_action = (struct COVEN_Collective_Action*)malloc(1*
              sizeof(struct COVEN_Collective_Action));
            COVEN_Create_Scatter_Action(&(col_action[0]), scatter_type,
              "my_bodies");
            COVEN_Scatter(num_slave_procs, col_action, 1);
            free(col_action);
        }

        free(array_of_blocks);
        free(array_of_displacements);
}
```

9.1.2 Gather Example

The following module is an example of a gather module. The basic idea is that many previously scattered buffers of bodies (`to_gather_bodies`) of size `num_to_gather` are being gathered. It is important to understand that this module is executed one time for each buffer that comes in. Therefore, if we scattered the original buffer into `32` segments this module will be executed `32` times, each time gathering more of the total result together into a single large buffer.

Again, we use MPI-style datatypes and datatype calls. In the example we set up the indexed type exactly the same way and go straight to the collective action. Here we choose a gather action and simple tell Coven to gather the buffer `to_gather_bodies` together into a large buffer. The buffers are gathered into a large buffer which resides in the static TPH. It is in this way that the buffer fills in with each gather call until it finally is complete. The use of the static TPH is completely transparent here.

The gather example follows.

```
/* we're going to use the static TPH to gather into behind the scenes */

/* the buffer here is the buffer to gather from, but is also the buffer that
 * all the gathered results will be placed in.  it's important to understand
```

```
 * here that the buffer is of size N for the buffer to BE gathered and of
 * size N * P for the buffer to gather INTO.  this example assumes that
 * there are P pieces (processes perhaps) of equal size, N.  it can easily
 * be extended to the more complex distributes that are possible. */

COVEN_Module nbody_gather
  (
   const int total_num_bodies,
   inout buffer MEDEA_BODY to_gather_bodies[num_to_gather]
  )
{
        int total_assigned_bodies = 0;
int num_slave_procs = COVEN_get_num_slave_processors();
        int target_num_bodies = (int)ceil((double)total_num_bodies /
          (double)num_slave_procs);
        int i = 0;
        COVEN_size *array_of_blocks, *array_of_displacements;
        COVEN_Type body_type, scatter_type;

        COVEN_Type_contiguous(sizeof(struct Medea_Body), COVEN_BYTE,
          &body_type);

        array_of_blocks = (COVEN_size*)malloc(num_slave_procs*
          sizeof(COVEN_size));
        array_of_displacements = (COVEN_size*)malloc(num_slave_procs*
          sizeof(COVEN_size));

        while(total_assigned_bodies < total_num_bodies) {
            /* try and fit this many bodies */
            int num_bodies = target_num_bodies;
            /* startingaat how many we've assigned so far */
            int start_body = total_assigned_bodies;

            total_assigned_bodies += num_bodies;
            /* if we've assigned more than we have, then back off by that
             * amount */
            if(total_assigned_bodies > total_num_bodies)
                num_bodies -= (total_assigned_bodies - total_num_bodies);

            /* now num_bodies is the number of bodies from this TPH */
            array_of_blocks[i] = num_bodies;
            array_of_displacements[i] = start_body;
            i++;
        }

        /* now that we have our displacements and blocks setup we can make
         * the COVEN data type */
        COVEN_Type_indexed(num_slave_procs, array_of_blocks,
          array_of_displacements, body_type, &scatter_type);

        {
            struct COVEN_Collective_Action *col_action;

            col_action = (struct COVEN_Collective_Action*)malloc(1*
```

```
            sizeof(struct COVEN_Collective_Action));
          /* this creates a collective action where we describe a region
           * in the STATIC TPH to gather the results into.  This will
           * gather buffers of the global binding "to_gather_bodies"
           * into a larger static TPH buffer of the same global binding.
           * Once they're all gathered, we will copy this buffer into the
           * output buffer before sending the TPH onward */
          COVEN_Create_Gather_Action(&(col_action[0]), scatter_type,
            "to_gather_bodies");
          /* make an action which says to gather into the static TPH
           * and we want that buffer to be of size 'total_num_bodies' */
          COVEN_Gather(total_num_bodies, col_action, 1);
        }
        free(array_of_blocks);
        free(array_of_displacements);
}
```

9.2 Direct TPH Access

It is the intent of Coven that module programmers should never directly access TPHs themselves. There should always be helper functions provided to make access to TPHs transparent. However, module programmers have complete access to any call in the Coven library (see the source tree). There are three special TPH variables passed to every module: `_default_tph`, `_const_tph`, and `_static_tph`. Direct TPH access will require knowing which TPH to pull what data from.

This is extremely discouraged.

9.3 Interesting extern Variables Available

The following variables are available to any module by simply placing the directive `extern` before the type and variable name.

9.3.1 int myProcessorNumber

`myProcessorNumber` is the processor number under which this module is currently running. If this module is executing on the *Master* task then this will be `0`. Otherwise, it will be a value between `1` and the number of slave tasks.

9.3.2 int num_Slave_Processors

`num_Slave_Processors` is the number of slave processors under the current configuration. Sometimes this is available through another means and at some point we may get rid of this variable all together.

9.3.3 MPI_Comm MPI_COMM_THIS_PARALLEL_TASK

`MPI_Comm MPI_COMM_THIS_PARALLEL_TASK` is a an MPI Communicator whose members are ONLY the instances of this particular module running on each parallel process. Therefore, if a module wanted to send a message to a copy of itself running on another processor it would use this Communicator Group. This does away with the module having to derive the MPI task IDs of those parallel instances of itself.

A Example Module

The following is an example module which shows many of the concepts outlined in this document. It is part of the N-Body suite of Coven modules. The module gathers the body data onto a single parallel task and then sends it through a TCP socket to a visualizer on another host and port.

Notice the use of a user defined data type `MEDEA_BODY`, a `static` modifier, some `extern` variables, and MPI communication to other instances of itself running on other processors. Additionally, note that there are calls to a separate library, libsockio, which can be found in the Coven source tree. Finally, notice the use of a buffer and the size of the buffer being assigned a variable named `my_num_bodies`.

```
#include "coven.h"
#include "medea.h"
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>

extern int myProcessorNumber;
extern MPI_Comm MPI_COMM_THIS_PARALLEL_TASK;

COVEN_Module nbody_visualize
  (
   const float timestep,
   const string vis_hostname,
   const int vis_portnumber,
   const int total_num_bodies,
   const int num_slave_procs,
   input int current_iteration,
   input buffer MEDEA_BODY my_bodies[my_num_bodies],
   static inout int * visualization_socket
  )
{
        int i, j, recv_target, *recv_counts, *displacements;
        MPI_Status status;
        MPI_Datatype body_type;
        MEDEA_BODY_TYPE total_bodies; /* it's typedefed as a pointer */

        MPI_Type_contiguous(sizeof(struct Medea_Body), MPI_BYTE, &body_type);
        MPI_Type_commit(&body_type);

        /* only the head node needs to malloc space for it */
        if(myProcessorNumber == 1) {
            int offset;

            total_bodies = (struct Medea_Body*)malloc(total_num_bodies*
              sizeof(struct Medea_Body));
            recv_counts = (int*)malloc(num_slave_procs*sizeof(int));
            displacements = (int*)malloc(num_slave_procs*sizeof(int));
            recv_counts[0] = my_num_bodies;
            displacements[0] = 0;
            offset = recv_counts[0];
            for(recv_target=2; recv_target<=num_slave_procs; recv_target++)
            {
                MPI_Recv(&(recv_counts[recv_target-1]), 1, MPI_INT,
                  recv_target-1, 0, MPI_COMM_THIS_PARALLEL_TASK, &status);
                /* we displace based on the PREVIOUS guy */
                displacements[recv_target-1] = offset;
                offset += recv_counts[recv_target-1];
            }
        }
        else {
            MPI_Send(&my_num_bodies, 1, MPI_INT, 0, 0,
```

```
            MPI_COMM_THIS_PARALLEL_TASK);
        }

        /* we gather at 0 b/c proc 1 is the 0th node of SLAVE_COMM, strange,
         * I know */
        MPI_Gatherv(my_bodies, my_num_bodies, body_type, total_bodies,
          recv_counts, displacements, body_type, 0,
          MPI_COMM_THIS_PARALLEL_TASK);
        if(myProcessorNumber == 1) {
            Medea_Header info;
            int sock;
            int outfd;

            info.numbodies = total_num_bodies;
            info.cycle = current_iteration;
            info.timestep = timestep;
            if(*visualization_socket == -1) { /* we havn't initted yet */
                int *iptr;
                int ret;

                sock = new_sock();
                ret = connect_sock(sock, vis_hostname, vis_portnumber);
                if(ret < 0) COVEN_Error("connect_sock");

                /* write to the attribute the socket value */
                *visualization_socket = sock;
            }
            else sock = *visualization_socket;

            COVEN_write(sock, &info, sizeof(Medea_Header));
            COVEN_write(sock, total_bodies, sizeof(struct Medea_Body)*
              total_num_bodies);

            free(total_bodies);
            free(recv_counts);
            free(displacements);
        }
}
```