eSkel

The **e**dinburgh **Ske**leton **l**ibrary

API reference manual

July 15, 2002

1 Introduction

This document defines the API to the **eSkel** parallel programming library. eSkel provides the C/MPI programmer with a collection of "skeletal" collective operations, each encapsulating the structure of a commonly recurring pattern of computation and communication. The document is not an introduction to eSkel. Prospective users should consult the accompanying tutorial document for a conceptual overview and introduction to the library.

2 Skeletons, Processes and Activities

eSkel library calls operate within the context of a running MPI program which has already created its fixed number of processes. The effect of interaction with MPI-2 dynamic process creation is undefined. Each skeleton is a collective operation to be called by all processes within the group associated with the communicator passed in through the final argument to the call.

During the call, the participating processes will be grouped and regrouped by the implementation, according to the semantics of the skeleton. Each group of processes created in this way constitutes an *activity* (e.g. a stage in a pipeline or a worker in a farm). Each activity is created within the context of a new communicator. At any time, a process in an activity can reference this communicator by calling the library function `mycomm()` and can acquire the corresponding group size and rank within the group with `mycommsize()` and `myrank()` respectively. This gives the programmer a safe communication context within which to express ad-hoc parallelism internal to an activity.

Since skeleton calls can be freely nested a process may be a member of a dynamically varying stack of activities, mirroring the stack of library calls it has made. `mycomm()` and related functions always refer to the activity on the top of this stack. If the stack is empty (for example, immediately after the opening call to `eSkelInit()`) then they refer to `MPI_COMM_WORLD`.

3 eDM: the eSkel Data Model

Data accesses in user code follow conventional C/MPI syntax and semantics within the address space of each participating process. Data oriented interaction with eSkel occurs upon entry to and exit from skeleton function calls, and during calls (whether implicit or explicit) to the interaction functions `Give`, `Take` and `Exchange`. The semantics of these interactions are specified in terms of **eDM**, the **eSkel Data Model**. eDM defines the concepts of an **eDM atom** and an **edM collection**.

3.1 eDM atom

The eDM atom is the unit of transfer between a skeleton function and the component processes of one its activities. It has two forms.

A **local spread** eDM atom is communicated between the skeleton and *exactly one* of its component processes. On the process, it takes the form of a contiguously stored sequence of homogenous data items, and is described by a triple consisting of a pointer to the start of the sequence, the `MPI_Datatype` of the items and an integer indicating the number of items in the sequence.

A **global spread** eDM atom is communicated between the skeleton and *all* component processes of one of its activities. It consists of a set of contiguously stored sequences of homogenous data items, one sequence on each process, with each sequence described by a triple as for a local spread atom. Conceptually, the entire atom is constructed from the concatenation of these sequences (or "portions") in order of process rank within the activity `mycomm()`. This is significant when the atom is communicated (via the skeleton) from one activity to another or between the skeleton and the calling program, allowing both ends of the communication to have a common interpretation of the enclosed data.

3.2 eDM collection

The eDM collection is the unit of data transfer between a calling program and a called skeleton function. It may take one of two forms.

A **local spread** eDM collection is composed of a number of contiguously stored, homogenous sequences of local spread eDM atoms, one sequence per calling process. The sequence on each process is described by a pointer to the start of the sequence, the `MPI_Datatype` and length underlying the constituent atoms and the length of the local sequence of atoms. Note that all constituent atoms on a given process are required to have the same underlying length but that this length may vary from process to process. Conceptually, the entire collection is constructed from the concatenation of these sequences in order of process rank within the activity `mycomm()`.

A **global spread** eDM collection is composed of a number of homogenous global spread eDM atoms. The portions of different atoms held by each calling process are stored contiguously. A global spread eDM collection is identified by, on each process, a pointer to the start of the contiguous area, the `MPI_Datatype` and length underlying the portions of constituent atoms (with the length being common to all portions on a particular process) and the length of the local sequence of portions (i.e. the number of atoms), which must be identical on all processes. Conceptually, the entire collection is constructed from the concatenation of its constituent atoms in the common order stored by each process.

In order to disambiguate the overloaded concept of "length", we will refer to the length of the sequence of data items underlying an eDM atom (or portion thereof) as a *length* and to the length of a sequence of eDM atoms as a *multiplicity*. Thus, for an eDM collection with global spread, the multiplicities provided by each process must be identical but the lengths of the contributed atoms, while uniform on any one process, may vary between processes.

3.3 Balanced distribution of collections

eSkel has no default policy on the load balancing of collections across processes, either for input to or output from a skeleton. Each skeleton is required to specify its policy. For example, we can imagine a one-for-one farm skeleton in which each result is stored by the process which contributed the corresponding task. Processes could contribute tasks with widely varying multiplicity and will receive a correspondingly imbalanced number of outputs. A different farm skeleton could specify implicit balancing of distribution of the output collection, irrespective of the imbalance in distribution of the input collection.

3.4 Tagging data with its spread

In order to indicate the intended interpretation of skeleton input and output collections, eDM provides an enumerated type `spread_t`, defined as

```
typedef enum {SPLOCAL, SPGLOBAL} spread_t;
```

from which values describing the spreads of the input and output collections are passed to the skeleton call. Note that eDM *does not* introduce new data types. The `spread_t` tag merely indicates the way in which the underlying data should be manipulated for the purposes of the call.

4 Utility Functions

4.1 `eSkelInit`

The eSkel initialisation function which must be called once, after `MPI_Init` but before any other eSkel function. The function must be called collectively.

```
void eSkelInit (void)
```

4.2 `mycomm`, `mycommsize` and `myrank`

Look up functions for the communication context of the current eSkel activity.

```
MPI_Comm mycomm (void)
```

returns
a handle for the communicator associated with the current activity of the calling process, or `MPI_COMM_WORLD` if there is no such activity.

```
int mycommsize (void)
```

returns
the size of the communicator associated with the current activity of the calling process, or of `MPI_COMM_WORLD` if there is no such activity.

```
int myrank (void)
```

returns
rank of the calling process within the communicator associated with the current activity, or within `MPI_COMM_WORLD` if there is no such activity.

5 Generic Interactions

Activity code interacts with the system and thereby indirectly with other activities through functions `Give`, `Take` and `Exchange`. Some skeletons make implicit calls to these functions, while others require the activity to make the calls explicitly (as a price for increased flexibility). The syntax of the API is generic. Each skeleton defines its own semantics.

5.1 `Give`

`Give` is called by an activity to pass an eDM atom to the current skeleton, from which the underlying `MPI_Datatype` and spread are inherited. For a global spread atom it is called collectively. For a local spread atom, it must only be called by the process with rank 0 in the activity. It is an error to call `Give` with a NULL pointer as first parameter.

```
void Give (void *out, int len)
```

where

`void *out`
: is a pointer to the local portion of the atom to return. For global spread, it must contain `len` items of the inherited type and spread. For local spread, the contribution of the process with rank 0 in the activity is passed to the skeleton, and must contain `len` items of the inherited type and spread.

`int len`
: is the number of items in the local portion.

5.2 `Take`

`Take` is called by an activity to receive an eDM atom from the current skeleton, from which the underlying `MPI_Datatype` and spread are inherited. For a local spread atom, only the process with rank 0 in the activity actually receives data.

```
void *Take (int *items)
```

returns
: a pointer to a dynamically allocated buffer of length `*items`. For local spread data, the pointer will be `NULL` in all but the process with rank 0 in the activity.

where

`int *items`
: is a pointer to the length (set by the call) of the returned buffer, in units of the `MPI_Datatype` inherited from the current skeleton.

5.3 Exchange

Some skeletons (currently only Butterfly) provide a natural concept of a process's partner. A common algorithmic requirement is for such partners to exchange data. This could be achieved with an appropriate sequence of Gives and Takes, with care needed to avoid deadlock. The Exchange primitive simplifies matters by encapsulating the sequence. For global data, the function is called collectively within an activity. For local spread data, only the process with rank 0 in the current activity should make the call. It is an error to call the function with a NULL pointer as the second parameter.

```
void *Exchange (int *inlen, void *out, int outlen)
```

returns

a pointer to a dynamically allocated buffer of length *inlen.

where

int *inlen
: is a pointer to the length (set by the call) of the returned buffer, in units of the MPI_Datatype inherited from the current skeleton.

void *out
: is a pointer to the local portion of the atom to give to the partner. For global spread, it must contain outlen items of the inherited type and spread. For local spread, the contribution of the process with rank 0 in the activity is passed to the partner, and must contain len items of the inherited type and spread.

int outlen
: is the number of items in the local portion.

5.4 eSkel_atom_t

Some skeletons define activities in which Gives, Takes and Exchanges are implicit. The type eSkel_atom_t provides a generic mechanism through which the results of implicit communications can be interfaced to activity functions, as argument and/or return types. An eSkel_atom_t is a wrapper for an array of data and its length. The information on type and spread which completes the definition of an eDM atom is inherited from the dynamic context.

```
typedef struct {
  void *data;
  int   len;
} eSkel_atom_t;
```

6 Skeleton Functions

Subsequent sections specify the currently available skeletons. Each section includes a conceptual introduction, a prototype, an explanation of the parameters, a definition of the semantics of `Give`, `Take` and `Exchange`, and a definition of the skeleton's policy on balancing of distribution of eDM collections. All skeleton functions must be called collectively.

6.1 Farm

This is a task farming construct with built-in computational load balancing across a number of worker activities. The worker code is invoked once in each worker activity and must make its own calls to `Take` (to request a new task atom) and `Give` (to return a result atom). This allows considerable flexibility. Tasks can be consumed without producing results. Alternatively, a single task may generate many results. The skeleton effects a dynamic, demand driven allocation of tasks to activities (in response to calls to `Take`) and so is most useful when the associated workload is expected to vary widely. There is no explicit "farmer" process - all calling processes are allocated to worker activities.

```
void Farm (int nw, void worker (void), int col,
           void *in,  int inlen, int inmul, spread_t inspr, MPI_Datatype inty,
           void *out, int outlen, int *outmul, spread_t outspr,
           MPI_Datatype outty, int outbuffsz, MPI_Comm comm)
```

where

`int nw`

denotes the number of worker activities to be created

`void worker (void)`

defines the worker activity

`int col`

controls allocation of processes to worker activities. `col` must be an integer in the range `0..nw-1` and indicates the worker activity to which the calling process will be allocated

`void *in`, ..., `MPI_Datatype inty`

describe the eDM collection of input tasks

`void *out`, ..., `int outbuffsz`

describe the eDM collection of output results. The caller must pre-allocate the required buffer space pointed to by `out`, indicating its size in `outbuffsz` (so that subsequent overflow can be detected). This is expressed in units corresponding to the space required to store a single item of type `outty` (NB the unit is *not* the size of a complete (or partial) result atom). Result atoms are ordered by the original ordering on the input collection tasks from which they were generated, and by order of generation between results generated from the same input. The caller defines the (common local) length of each output atom through `outlen`. A run-time error will occur if a returned atom does not conform to the specification[1]. `outmul` is set on return from the call to indicate the actual output multiplicity.

`MPI_Comm comm`
the communicator within which the call is executed as a collective operation

6.1.1 Generic Interactions

`Give` causes the given task atom to be added to the output collection upon termination of the skeleton call. It is non-blocking in the sense that the skeleton is always ready to accept a new output. It must be called collectively within an activity (worker) for global spread data, and by process 0 only of the active communicator for local spread data.

`Take` causes a new task atom from the input collection to be given to the calling activity. It returns NULL if no further tasks are available. It is non-blocking in the sense that the skeleton is always ready to return a new atom, or NULL. It must be called collectively within an activity (worker) for both local and global spread data.

`Exchange` has no natural definition for this skeleton. Its use is not permitted and will result in a run-time error.

6.1.2 Distribution of collections

There are no assumptions or constraints on the distribution of the input collection. The output collection will be balanced across the set of calling processes, with each process receiving an approximately equal number of atoms (or portions thereof in the case of global spread).

6.1.3 Termination

The call completes when all activities (ie. workers) have returned. It is the programmer's responsibilty to ensure that this only happens when no further tasks are available (unless some other, less obvious behaviour is appropriate). This is a natural consequence of the flexible nature of the skeleton's interaction model. The normal structure of the worker function will therefore be a loop, terminating when **Take** returns NULL.

6.1.4 Usage notes

The implementation of `Farm` is threaded and so must only be used with MPI implementations which are fully thread safe, in the sense of `MPI_THREAD_MULTIPLE` (see the `SimpleFarm` skeleton for a non-threaded alternative).

[1]This check is not yet implemented.

6.2 Farm1for1

This is a task farming construct with built-in computational load balancing across a number of worker activities, for use when each input task atom produces exactly one result atom. The computation to be applied to each task is described by the function for the worker activity. The skeleton effects a dynamic, demand driven allocation of tasks to activities and so is most useful when the associated workload is expected to vary widely. There is no explicit "farmer" process - all calling processes are allocated to worker activities.

```
void Farm1for1 (int nw, eSkel_atom_t * worker (eSkel_atom_t *), int col,
                void *in,  int inlen, int inmul, spread_t inspr,
                MPI_Datatype inty, void *out, int outlen, int *outmul,
                spread_t outspr, MPI_Datatype outty, int outbuffsz,
                MPI_Comm comm)
```

where

`int nw`

denotes the number of worker activities to be created

`eSkel_atom_t *worker (eSkel_atom_t *)`
is the worker activity, applied to an individual task (passed as the parameter) to create a returned result

`int col`
controls allocation of processes to worker activities `col` must be an integer in the range `0..nw-1` and indicates the worker activity to which the calling process will be allocated

`void *in`, ..., `MPI_Datatype inty`

describe the input eDM collection

`void *out`, ..., `int outbuffsz`
describe the eDM collection of output results. The caller must pre-allocate the required buffer space pointed to by `out`, indicating its size in `outbuffsz` (so that subsequent overflow can be detected). This is expressed in units corresponding to the space required to store a single item of type `outty` (NB the unit is *not* the size of a complete (or partial) result atom). The value of `outmul` will be set by the system upon return from the call. Result atoms are ordered by the original ordering on the input collection tasks from which they were generated. The caller defines the (common local) length of each output atom through `outlen`. A run-time error will occur if a returned atom does not conform to the specification[2].

`MPI_Comm comm`
the communicator within which the call is executed as a collective operation

6.2.1 Generic Interactions

`Give` should not be called. Its effect is already subsumed by the skeleton.

`Take` should not be called. Its effect is already subsumed by the skeleton.

`Exchange` has no natural definition for this skeleton. Its use is not permitted and will result in a run-time error.

6.2.2 Distribution of collections

There are no assumptions or constraints on the distribution of the input collection. The output collection will be balanced across the set of calling processes, with each process receiving an approximately equal number of atoms (or portions thereof in the case of global spread). This means that, *for any particular process*, the returned output multiplicity may be different from the given input multiplicity, even though the total number of atoms in the input and output must be identical (i.e. “1for1”).

6.2.3 Termination

The call completes when all input tasks have been processed. Since calls to `Give` and `Take` are encapsulated by the skeleton, the only possible problem would arise if a non-terminating `worker` were provided.

6.2.4 Usage notes

The implementation of `Farm1for1` is threaded and so must only be used with MPI implementations which are fully thread safe, in the sense of `MPI_THREAD_MULTIPLE` (see the `SimpleFarm1for1` skeleton for a non-threaded alternative).

[2] This check is not yet implemented.

6.3 SimpleFarm

This is a task farming construct with built-in computational load balancing across a number of worker activities. One process (that with rank 0 in `comm`) is reserved for the role of farmer. Each other process forms a one process `worker` activity, in which the worker code is invoked once and must make its own calls to `Take` (to request a new task atom) and `Give` (to return a result atom). This allows considerable flexibility. Tasks can be consumed without producing results. Alternatively, a single task may generate many results. The skeleton effects a dynamic, demand driven allocation of tasks to activities (in response to calls to `Take`) and so is most useful when the associated workload is expected to vary widely.

```
void SimpleFarm (void worker(void),
                 void *in,  int inlen, int inmul, spread_t inspr,
                 MPI_Datatype inty, void *out, int outlen, int *outmul,
                 spread_t outspr, MPI_Datatype outty, int outbuffsz,
                 MPI_Comm comm)
```

where

`void worker(void)`
defines the worker activity

`void *in`, ..., `MPI_Datatype inty`
describe the eDM collection of input tasks

`void *out`, ..., `int outbuffsz`
describe the eDM collection of output results. The caller must pre-allocate the required buffer space pointed to by `out`, indicating its size in `outbuffsz` (so that subsequent overflow can be detected). This is expressed in units corresponding to the space required to store a single item of type `outty` (NB the unit is *not* the size of a complete (or partial) result atom). Result atoms are ordered by the original ordering on the input collection tasks from which they were generated, and by order of generation between results generated from the same input. The caller defines the (common local) length of each output atom through `outlen`. A run-time error will occur if a returned atom does not conform to the specification[3]. `outmul` is set on return from the call to indicate the actual output multiplicity.

`MPI_Comm comm`
the communicator within which the call is executed as a collective operation

[3]This check is not yet implemented.

6.3.1 Generic Interactions

`Give` causes the given task atom to be added to the output collection upon termination of the skeleton call. It is non-blocking in the sense that the skeleton is always ready to accept a new atom.

`Take` causes a new task atom from the input collection to be given to the calling activity. It returns NULL if no further tasks are available. It is non-blocking in the sense that the skeleton is always ready to return a new atom, or NULL.

`Exchange` has no natural definition for this skeleton. Its use is not permitted and will result in a run-time error.

6.3.2 Distribution of collections

There are no assumptions or constraints on the distribution of the input collection. The output collection will be balanced across the set of calling processes, with each process receiving an approximately equal number of atoms (or portions thereof in the case of global spread).

6.3.3 Termination

The call completes when all activities (ie. workers) have returned. It is the programmer's responsibilty to ensure that this only happens when no further tasks are available (unless some other, less obvious behaviour is appropriate). This is a natural consequence of the flexible nature of the skeleton's interaction model. The normal structure of the worker function will therefore be a loop, terminating when **Take** returns NULL.

6.3.4 Usage notes

The implementation of `SimpleFarm` is not threaded and is therefore available on all systems. Note that the worker is sequential by definition.

6.4 SimpleFarm1for1

This is a task farming construct with built-in computational load balancing across a number of worker activities, for use when each input task atom produces exactly one result atom. One process (that with rank 0 in `comm`) is reserved for the role of farmer. Each other process forms a one process activity. The computation to be applied to each task is described by the code for the worker activity. The skeleton effects a dynamic, demand driven allocation of tasks to activities and so is most useful when the associated workload is expected to vary widely.

```
void SimpleFarm1for1 (eSkel_atom_t * worker (eSkel_atom_t *),
           void *in,  int inlen,  int inmul, spread_t inspr,
           MPI_Datatype inty, void *out, int outlen,  int *outmul,
           spread_t outspr, MPI_Datatype outty, int outbuffsz, MPI_Comm comm)
```

where

`eSkel_atom_t * worker (eSkel_atom_t *)`
is the worker activity, applied to an individual task (passed as the parameter) to create a returned result

`void *in`, ..., `MPI_Datatype inty`
describe the eDM collection of input tasks

`void *out`, ..., `int outbuffsz`
describe the eDM collection of output results. The caller must pre-allocate the required buffer space pointed to by `out`, indicating its size in `outbuffsz` (so that subsequent overflow can be detected). This is expressed in units corresponding to the space required to store a single item of type `outty` (NB the unit is *not* the size of a complete (or partial) result atom). The value of `outmul` will be set by the system upon return from the call. Result atoms are ordered by the original ordering on the input collection tasks from which they were generated. The caller defines the (common local) length of each output atom through `outlen`. A run-time error will occur if a returned atom does not conform to the specification[4].

`MPI_Comm comm`
the communicator within which the call is executed as a collective operation

[4]This check is not yet implemented.

6.4.1 Generic Interactions

`Give` should not be called. Its effect is already subsumed by the skeleton.

`Take` should not be called. Its effect is already subsumed by the skeleton.

`Exchange` has no natural definition for this skeleton. Its use is not permitted and will result in a run-time error.

6.4.2 Distribution of collections

There are no assumptions or constraints on the distribution of the input collection. The output collection will be balanced across the set of calling processes, with each process receiving an approximately equal number of atoms (or portions thereof in the case of global spread). This means that, *for any particular process*, the returned output multiplicity may be different from the given input multiplicity, even though the total number of atoms in the input and output must be identical (i.e. "1for1").

6.4.3 Termination

The call completes when all input tasks have been processed. Since calls to `Give` and `Take` are encapsulated by the skeleton, the only possible problem would arise if a non-terminating `worker` were provided.

6.4.4 Usage notes

The implementation of `SimpleFarm1for1` is not threaded and is therefore available on all systems. Note that the worker is sequential by definition.

6.5 Pipeline

Allows a sequence of activities (called "stages") to be chained together and applied to an input eDM collection, atom by atom, to produce an output eDM collection. The code for each stage is invoked once in the corresponding activity and must make its own calls to `Take` (to request a new atom, indirectly from the preceding stage, or in the case of the first stage, from the input collection) and `Give` (to return a new atom for subsequent consumption by the next stage, or, in the case of the last stage, for addition to the output collection). This allows great flexibility. A stage can consume data without producing results. Alternatively, a stage can produce results without consuming new (or any) data. Either or both input and output collections may be empty, with the pipeline correspondingly creating data dynamically from the participating process state, or similarly storing "results" in that state for subsequent retrieval.

```
void Pipeline (int ns, void (*stages[])(void), int col, spread_t spr[],
               MPI_Datatype ty[], void *in,  int inlen, int inmul,
               void *out, int outlen, int *outmul, int outbuffsz,
               MPI_Comm comm)
```

where

`int ns`
: is the number of stages

`void (*stages[]) (void)`
: are the activities for the stages, in order, from first to last

`int col`
: allows control of allocation of processes to stages. `col` should be an integer in the range `0..ns-1`, indicating the stage to which each process should be allocated.

`spread_t spr[], MPI_Datatype ty[]`
: Type information for the stage interfaces. These arrays must have `ns+1` elements, one for each stage input and one for the final output.

`void *in, int inlen, int inmul,`
: together with `ty[0]` and `spr[0]`, describe the eDM collection of input tasks

`void *out`, `int outlen`, `int *outmul`, `int outbuffsz`,
together with `ty[ns]` and `spr[ns]`, describe the eDM collection of output results. The caller must pre-allocate the required buffer space pointed to by `out`, indicating its size in `outbuffsz` (so that subsequent overflow can be detected). This is expressed in units corresponding to the space required to store a single item of type `outty` (NB the unit is *not* the size of a complete (or partial) result atom). Output atoms are ordered by their emergence from the final stage. The caller defines the (common local) length of each output atom through `outlen`. A run-time error will occur if a returned atom does not conform to the specification[5]. `outmul` is set on return from the call to indicate the actual output multiplicity.

`MPI_Comm comm`
the communicator within which the call is executed as a collective operation

6.5.1 Generic Interactions

`Give` returns a processed atom for subsequent handling by the next stage, or, if called in the final stage, for addition to the output collection upon termination of the skeleton call. `Give` called from any stage except the last is blocking, in the sense that the call will only complete when a corresponding `Take` has also completed. `Give` from the final stage is non-blocking, in the sense that the skeleton is always ready to accept the processed atom. It must be called collectively within an activity (stage) for global spread data, and by process 0 only of the active communicator for local spread data.

`Take` acquires the next atom, previously processed by the preceding stage, or if called from the first stage, from the input collection. It returns NULL if the sequence of atoms is finished. `Take` called from any stage except the first is blocking, in the sense that the call will only complete when a corresponding `Give` has also completed. `Take` from the first stage is non-blocking, in the sense that the skeleton is always ready to produce a new atom or NULL. It must be called collectively within an activity (stage) for both local and global spread data.

`Exchange` has no natural definition for this skeleton. Its use is not permitted and will result in a run-time error.

6.5.2 Distribution of collections

There are no assumptions or constraints on the distribution of the input collection. The output collection will be balanced across the set of calling processes, with each process

[5]This check is not yet implemented.

receiving an approximately equal number of atoms (or portions thereof in the case of global spread).

6.5.3 Termination

The call completes when all activities (ie. stages) have returned.

6.5.4 Usage notes

Given the semantics of termination and of `Give` and `Take`, the natural form for a stage is a loop, terminating on return of a NULL pointer from `Take`. Note that there is no need for a stage to explicity signal termination, other than by returning from its call. Indeed, it would be an error to explicitly try to `Give` and NULL pointer (see the generic semantics of `Give` earlier).

`Pipeline` has both threaded and non-threaded implementations, currently selectable by a minor tweak to the library `include` file. The threaded implementation must only be used with MPI implementations which are fully thread safe, in the sense of `MPI_THREAD_MULTIPLE`.

6.6 Pipeline1for1

Allows a sequence of activities (called "stages") to be chained together and applied to an input eDM collection, atom by atom, to produce an output eDM collection, for use when each stage produces exactly one result for each input. The code for each stage is therefore invoked once for each atom.

```
void Pipeline1for1 (int ns, eSkel_atom_t *(*stages[])(eSkel_atom_t *),
                    int col, spread_t spr[], MPI_Datatype ty[],
                    void *in,  int inlen, int inmul, void *out, int outlen,
                    int *outmul, int outbuffsz, MPI_Comm comm)
```

where

`int ns`

is the number of stages

`eSkel_atom_t *(*stages[])(eSkel_atom_t *)`

are the activities for the stages, in order, from first to last

`int col`

allows control of allocation of processes to stages. `col` should be an integer in the range `0..ns-1`, indicating the stage to which each process should be allocated.

`spread_t spr[], MPI_Datatype ty[]`

Type information for the stage interfaces. These arrays must have `ns+1` elements, one for each stage input and one for the final output.

`void *in`, `int inlen`, `int inmul`

together with `ty[0]` and `spr[0]`, describe the input eDM collection

`void *out`, ... `MPI_Datatype outty`

together with `ty[ns]` and `spr[ns]`, describe the eDM collection of output results. The caller must pre-allocate the required buffer space pointed to by `out`, indicating its size in `outbuffsz` (so that subsequent overflow can be detected). This is expressed in units corresponding to the space required to store a single item of type `outty` (NB the unit is *not* the size of a complete (or partial) result atom). The value of `outmul` will be set by the system upon return from the call. Result atoms are ordered by the original ordering on the input collection tasks from which they were generated. The caller defines the (common local) length of each output atom through `outlen`. A run-time error will occur if a returned atom does not conform to the specification[6].

`MPI_Comm comm`
the communicator within which the call is executed as a collective operation

6.6.1 Generic Interactions

`Give` should not be called. Its effect is already subsumed by the skeleton.

`Take` should not be called. Its effect is already subsumed by the skeleton.

`Exchange` has no natural definition for this skeleton. Its use is not permitted and will result in a run-time error.

6.6.2 Distribution of collections

There are no assumptions or constraints on the distribution of the input collection. The output collection will be balanced across the set of calling processes, with each process receiving an approximately equal number of atoms (or portions thereof in the case of global spread). This means that, *for any particular process*, the returned output multiplicity may be different from the given input multiplicity, even though the total number of atoms in the input and output must be identical (i.e. "1for1").

6.6.3 Termination

The call completes when all activities (ie. stages) have returned. Since calls to `Give` and `Take` are encapsulated by the skeleton, the only possible problem would arise if a non-terminating `stage` were provided.

6.6.4 Usage notes

`Pipeline1for1` has both threaded and non-threaded implementations, currently selectable by a minor tweak to the library `include` file. The threaded implementation must only be used with MPI implementations which are fully thread safe, in the sense of `MPI_THREAD_MULTIPLE`.

[6] This check is not yet implemented.

6.7 Butterfly

The first in a family of skeletons which support divide & conquer algorithms. `Butterfly` specifies a temporal sequence of sets of activities corresponding to the levels of a divide tree. At the first level, a single activity involves all calling processes. Subsequent steps partition each group from the preceding step set into two equally sized sub groups at the next level, terminating after completion of a step with group size of two (NB not one). The initial group size must be a power of 2 (in fact, 2^{nd}). `level()` is called once within each activity. Its return signifies the end of participation at the current step. Within `level()`, calls to `Give` and `Take` interact with the butterfly partner at that level (i.e. with process `(myid()+mycommsize()/2)%mycommsize()`), as does `Exchange` which combines the effect a quartet of matching `Give`s and `Take`s. Note that **Butterfly** has no explicit data parameters, and must inherit and pass on state from and to the calling program.

```
void Butterfly (int nd, void level(void), MPI_Datatype ty, MPI_Comm comm)
```

where

`int nd`
is the number of levels. This must be the base 2 logarithm of the number of participating processes.)

`void level (void)`
is run once within each activity

`MPI_Datatype ty`
Type for transfers assumed by `Give`, `Take` and `Exchange`.

`MPI_Comm comm`
the communicator within which the call is executed.

6.7.1 Generic Interactions

`Give` passes an atom to the skeleton for subsequent collection (with `Take`) by the butterfly partner at the current level (i.e. with process `(myid()+mycommsize()/2)%mycommsize()`). Note that no spread is specified since partners are by definition sequential (and so global and local spreads are indistinguishable). `Give` is blocking in the sense that it will only complete when a corresponding `Take` also completes. Note that `Give` should not be called simultaneously across all activities at a given level, since this will result in deadlock (waiting for matching `Take`s).

`Take` acquires an atom from the skeleton, as previously generated with `Give` by the butterfly partner at the current level (i.e. process `(myid()+mycommsize()/2)%mycommsize()`). Note that no spread is specified since partners are by definition sequential (and so global and local spreads are indistinguishable). `Take` is blocking in the sense that it will only complete when a corresponding `Give` also completes. Note that `Take` should not be called simultaneously across all activities at a given level, since this will result in deadlock (waiting for matching `Give`s).

`Exchange` encapsulates the set of four `Give`s and `Take`s between partners which would otherwise be necessary to enact an exchange of atoms, one in each direction. Note that no spread is specified since partners are by definition sequential (and so global and local spreads are indistinguishable). `Exchange` is most naturally used collectively across all activities at the current level, but the specification does not prohibit calls by isolated pairs of partner activities. In such circumstances it is the programmer's responsibility to ensure that both partners make the call.

6.7.2 Distribution of collections

This skeleton has neither input nor output collections.

6.7.3 Termination

The call terminates when all calls of `level` within activities of size 2 have terminated.

6.7.4 Usage notes

Note that the function is only defined for calls occuring within a communicator cointaining a "power of two" number of processes.