eSkel

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

Tutorial introduction

July 15, 2002

1 Introduction

This document provides a tutorial introduction to the **eSkel** parallel programming library. It assumes a reasonable level of familiarity with MPI and its C binding. Full details of the eSkel API can be found in the accompanying API reference manual.

2 Collective operations and skeletons

MPI programmers are already familiar with "collective operations" and the benefits they bring. Consider the `MPI_Broadcast` function. This provides a simple interface to a conceptually useful operation which occurs frequently in a range of applications. Without it, the programmer would have to choose and code an implementation of broadcast with simple sends and receives. As well as being tiresome and error prone, this would have the effect of embedding the choice of broadcast algorithm in the code, irrespective of its suitability (in terms of performance) for porting to other architectures. With it, the programming task is reduced to making a single function call with a handful of parameters, benefiting (in principle at least) from carefully tuned implementations on each architecture to which the code is ported.

While `MPI_Broadcast` abstracts only a pattern of communication, `MPI_Reduce` encapsulates a more complex operation also involving computation. The API makes no statement about the algorithm. The requirements placed on the reduction operator (associativity and commutativity) hint that one possibility will involve internal parallelism. As with broadcast, the detailed exploitation is left to the MPI implementation.

`MPI_Broadcast`, `MPI_Reduce` and MPI's other collective operations are useful tools. However, the experienced parallel programmer is aware that there are other "patterns of computation and communication" which occur in a range of applications but which are not catered for directly. For example, pipelines and task farms are well established concepts, helpful during program design, but must be implemented directly in terms of MPI's simpler operations. The goal of the eSkel library is to add such higher level collective operations to the MPI programmer's toolbox.

3 A first example

The concept of pipeline parallelism is long standing and of well proven usefulness in both hardware and software. In its simplest form (although we shall soon see that there are many variants), a sequence of *stages* process a sequence of *inputs* to produce a sequence of *outputs*. Each input passes through each stage in the same order. In the spirit of "Hello World!" introductions we now present a small example which achieves nothing useful, but serves to illustrate the mechanism. Figure 1 illustrates a pipeline (flowing from right to

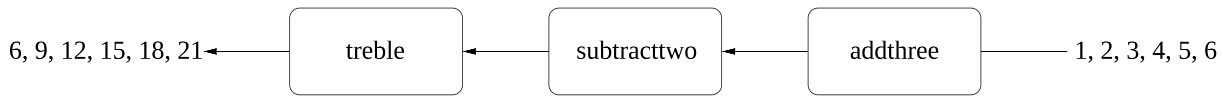

Figure 1: A simple pipeline

left) of three stages which respectively add three to, subtract two from and multiply by three the integers they receive.

The code for a complete program implementing the pipeline is presented in figure 2. The bad news is that this may look a little complex for an apparently simple application. The good news is that things don't get any harder for much more complex situations. The long parameter list of the pipelining library function allows substantial flexibility, which will bring benefits in subsequent examples. An easier API could satisfy this example, but would be too simplistic in many realistic situations. Look first at lines 57-58. These are the call to the eSkel function `Pipeline1for1`.

```
57    Pipeline1for1 (P, stages, myrank(), spreads, types, (void *) inputs, 1, 2,
58                   (void *) results, 1, &outmul, 2, mycomm());
```

The "`1for1`" indicates a pipeline in which every stage produces exactly one output for each input (as we will see, not all pipelines are like this). The first two parameters specify the number of stages `P` (previously defined as 3) and the stages themselves. These are described by the functions defined between lines 5 and 31 and collected together as an array `stages` by line 50.

```
50    stages[0] = addthree; stages[1] = subtracttwo; stages[2] = treble;
```

A key feature of eSkel is that it allows nested parallelism. For example, in a pipeline, we may wish the internal processing of one or more stages to be parallel itself. eSkel uses the term *activity* to refer to the workings of a skeleton defined group of processes such as a pipeline stage or a farm worker. For maximum flexibility, we should be able to control the allocation of participating processes to activities. This is the role of the third parameter. In this simple example we want only one process per stage and so it suffices to call the eSkel function `myrank`, which returns the rank of the calling process within the current skeleton communicator. The effect is to assign each process to the stage corresponding to its rank.

Our next task is describe the type of information which will flow between stages. Since we have three stages there will be four such interfaces, including the input to and output from the whole pipeline:

- the array `spreads` uses the eSkel value `SPLOCAL` to indicate that all interfaces pass items which are stored in their entirety by single processes (in contrast to `SPGLOBAL` items, which may be distributed across many processes);
- the array `types` indicates the types of these items, being `MPI_INT` in all interfaces in this example;

The contents of `spreads` and `types` were set up prior to the call in lines 52-55.

```
52    for (i=0; i<P+1; i++) {
53      spreads[i] = SPLOCAL;
54      types[i]   = MPI_INT;
55    }
```

Here they are homogenous, but this is not a restriction imposed by the library. Each interface can have its own type and spread.

Having described the required pipeline structure we must now specify the data which it is to process. Conceptually, both input to and output from a pipeline consist of homogenous sequences of data items. In the *eSkel data model*, both the input sequence and the storage space for the output sequence are assumed to be distributed across the processes making the skeleton call. Each process describes the contribution it is making, and the model specifies the interpretation of the contributions as a single sequence. The input contributions from each participating process are described by the sixth, seventh and eighth parameters. The eighth parameter (here “2”) indicates the number of items to which a process will contribute. Since there are three processes in the example, each indicating its intention to provide two items, and since these items are of local spread (meaning that each item is held in its entirety by a single process), there will be six inputs in all to the pipeline. We call this parameter the input *multiplicity*. The eSkel data model specifies that the six inputs will be presented to the first stage of the pipeline in an order governed by the rank of the originating process and by storage order within that process (in other words all inputs from process 0, followed by all inputs from process 1 and so on).

The sixth and seventh parameters provide the input data itself. For the purposes of this toy example, each process has previously created and filled (lines 44-47) a local array of integers called `inputs`.

```
44    inputs   = (int *) malloc (sizeof(int)*2);
45    for (i=0; i<2; i++) {
46      inputs[i]   = myrank()*2+i+1;
47    }
```

These are passed as the seventh parameter, with the “1” for the eighth parameter indicating that an individual input is an array of length 1 (and so, had we instead wanted our basic atom of data to be an array of two integers, we would have passed "2" here and correspondingly either created twice as much data or halved the specified input multiplicity).

The specification of output data is similar. Following the MPI model for receiving communications, eSkel requires the programmer to pre-allocate storage for the results of skeleton calls. This takes place on line 48. The buffer `results` is passed to the pipeline skeleton as the ninth parameter, and its length as the twelfth parameter. The tenth parameter indicates the length (as a multiple of the output type) of a result item. The eleventh parameter is a pointer to an integer through which the system will return the number of output items allocated to a process (i.e. the output multiplicity). Note that in true SPMD style each process is providing a buffer. The output results will be left scattered across this distributed buffer.

The final parameter indicates the communicator (and by implication the groups of processes) within which the collective call is executed, in the usual way for MPI collective calls.

The stages themselves share a common prototype required of any function used as a `Pipeline1for1` stage. For example, the stage which returns the treble of each integer it sees has the header

```
23  eSkel_atom_t *treble(eSkel_atom_t *datum) {
```

where an `eSkel_atom_t` is a structure containing an array of items and an integer denoting its length.

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

Thus the integers which `treble` actually receives are wrapped up as structures. The data pointer and length can be extracted by inspecting the relevant fields.

```
25    int *data = (int *)(datum->data);
26
27    printf("Final stage will return %d\n", data[0]*3);
28    data[0] *= 3;
29    return datum;
```

The `printf` statement on line 27 shows the integers passing out of the final stage and into the output collection. We could equally well have coded print statements after the call to `Pipeline1for1` to illustrate the distributed nature of the output as finally stored (each process will have two of the results in its `results` buffer).

```c
#include "../eSkel.c"

#define P 3

eSkel_atom_t *addthree (eSkel_atom_t *datum) {

  int *data = (int *)(datum->data);

  data[0] += 3;
  return datum;

}

eSkel_atom_t *subtracttwo (eSkel_atom_t *datum) {

  int *data = (int *)(datum->data);

  data[0] -= 2;
  return datum;

}

eSkel_atom_t *treble(eSkel_atom_t *datum) {

  int *data = (int *)(datum->data);

  printf("Final stage will return %d\n", data[0]*3);
  data[0] *= 3;
  return datum;

}

int main (int argc, char *argv[]) {
  int *inputs;
  int *results;
  int i, outmul;
  spread_t spreads[P+1];
  MPI_Datatype types[P+1];
  eSkel_atom_t *(*stages[P])(eSkel_atom_t *);

  MPI_Init(&argc, &argv);
  SkelLibInit();

  inputs  = (int *) malloc (sizeof(int)*2);
  for (i=0; i<2; i++) {
    inputs[i]  = myrank()*2+i+1;
  }
  results = (int *) malloc (sizeof(int)*2);

  stages[0] = addthree; stages[1] = subtracttwo; stages[2] = treble;

  for (i=0; i<P+1; i++) {
    spreads[i] = SPLOCAL;
    types[i]   = MPI_INT;
  }

  Pipeline1for1 (P, stages, myrank(), spreads, types, (void *) inputs, 1, 2,
                 (void *) results, 1, &outmul, 2, mycomm());

  MPI_Finalize();
  return 0;
}
```

Figure 2: Code for the simple pipeline

4 Global spread data and *ad-hoc* parallelism

Our second example introduces the use of data items which are distributed across a number of processes (with *global spread*) and also demonstrates how the use of eSkel can be interwoven with direct calls to MPI, both at a program's outermost layer and within the activities controlled by skeletons. We call this *ad-hoc* parallelism. Figure 3 illustrates a

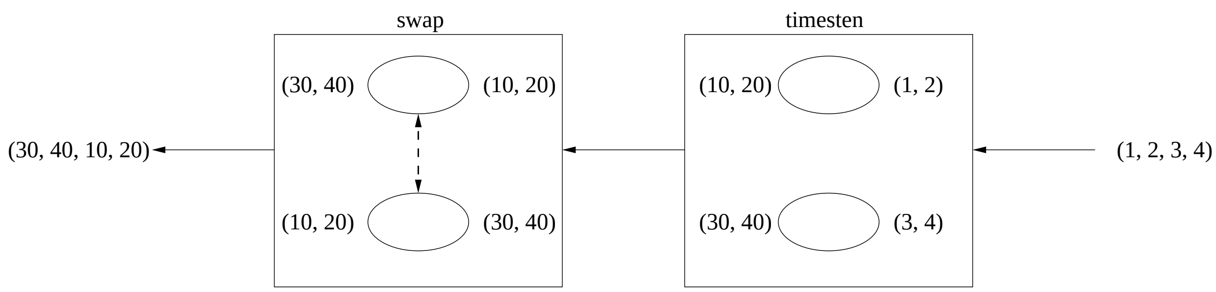

Figure 3: A global spread pipeline

two stage pipeline in which each stage involves two processes. The code for a complete program implementing the pipeline is presented in figure 4. The data items passed down the pipeline have `SPGLOBLAL` spread, indicating that each item is distributed across all the processes involved in an activity. In this example each item consists of a sequence of four integers. When passed to a stage involving two processes, the eSkel data model dictates that these will be distributed two to each process, blockwise.

The first stage, `timesten`, simply multiplies each integer in its portion of the data by 10.

```
eSkel_atom_t *timesten (eSkel_atom_t *datum) {
  int i;
  int *data = (int *)(datum->data);

  for (i=0; i<datum->len; i++) {
    data[i] *= 10;
  }
  return datum;
}
```

The second stage, `swap`, illustrates the use of ad-hoc parallelism within a stage. The two processes call `MPI_Sendrecv_replace` to exchange the pairs of integers they hold (so that the distributed item `A B C D` becomes `C D A B`). The "other" process in this two process stage is identified by simple arithmetic on `myrank()`, which, together with the exchange itself, takes place within the context of `mycomm()` the communicator encompassing the stage.

```
eSkel_atom_t *swap (eSkel_atom_t *datum) {
  MPI_Status status;

  int *data = (int *)(datum->data);
```

```
    MPI_Sendrecv_replace (data, datum->len, MPI_INT, (myrank()+1)%2, 0,
                          (myrank()+1)%2, 0, mycomm(), &status);
    return datum;
}
```

The data to be processed consists of a sequence of three items (and so the input multiplicity is three). Each item consists of four integers, with global spread. This means that each of the four processes making the skeleton call must provide a portion of each of the three items. However, these portions need not be of identical size. To illustrate the point, we make the process with rank 0 provide all the data.

```
    inmul = 3;
    if (myrank() == 0) { // P0 creates all the data
      inputs = (int *) malloc (sizeof(int) * 12);
      for (i=0; i<12; i++) {
        inputs[i] = i+1;
      }
      inlen = 4;
    } else {
      inputs = NULL; inlen = 0;
    }
```

Notice that the data itself is simply an array of 12 integers. The multiplicity and length parameters to the call are responsible for determining that this should be treated as three chunks of four. Note also that the existence of only one "length" parameter per process dictates that the lengths of segments provided by any process must be uniform. Thus, we can have different processes providing different lengths (as here), but it would inexpressible, for example, to have process 0 provide a final portion of length 3, with the remaining integer coming from some other process.

In contrast, distribution of the pipeline outputs is under the control of the system and, following the eSkel data model, will result in each output item being distributed blockwise, as regularly as possible, across the calling processes. The processes must allocate enough space to accommodate their shares of each item in a contiguous block of memory (with interpretation, as for inputs, determined by the specified spread and length). In the example, we know that the output will have the same overall size as the input (since the arrays are manipulated but not resized by stage activities) and so each process must reserve space for three integers (or to be precise, three portions, each of one integer, each contributing to a four integer distributed output item).

```
    results = (int *) malloc (sizeof(int)*3); outlen = 1;
```

In order to simplify printing, we finish by having the distributed outputs gathered into a single array on process 0. This is achieved by a series of calls to `MPI_Gather`, one for each distributed item.

```
    if (myrank() == 0) {
      gathereddata = (int *) malloc (sizeof(int)*12);
    }
    for (i=0; i<outmul; i++) {
      MPI_Gather (&results[i], 1, MPI_INT, &gathereddata[i*4], 1, MPI_INT, 0, MPI_COMM_WORLD);
    }
```

The program prints the following output

```
Inputs on P0: 1 2 3 4 5 6 7 8 9 10 11 12

Gathered results on P0: 30 40 10 20 70 80 50 60 110 120 90 100
```

illustrating the expected effect on each of the items (i.e. each value multiplied by 10, and the first two and second two components of each item exchanged).

```c
#include "../eSkel.c"
#define P 4   // number of processes
#define S 2   // number of stages

eSkel_atom_t *timesten (eSkel_atom_t *datum) {
  int i;
  int *data = (int *)(datum->data);

  for (i=0; i<datum->len; i++) {
    data[i] *= 10;
  }
  return datum;
}

eSkel_atom_t *swap (eSkel_atom_t *datum) {
  MPI_Status status;

  int *data = (int *)(datum->data);
  MPI_Sendrecv_replace (data, datum->len, MPI_INT, (myrank()+1)%2, 0,
                        (myrank()+1)%2, 0, mycomm(), &status);
  return datum;
}

int main (int argc, char *argv[]) {
  int *inputs, *results, *gathereddata = NULL;
  int i, inmul, outmul, inlen, outlen;
  spread_t spreads[S+1]; MPI_Datatype types[S+1];
  eSkel_atom_t *(*stages[S])(eSkel_atom_t *);

  MPI_Init(&argc, &argv);
  SkelLibInit();
  inmul = 3;
  if (myrank() == 0) { //P0 creates all the data
    inputs = (int *) malloc (sizeof(int) * 12);
    for (i=0; i<12; i++) {
      inputs[i] = i+1;
    }
    inlen = 4;
  } else {
    inputs = NULL; inlen = 0;
  }
  results = (int *) malloc (sizeof(int)*3); outlen = 1;
  stages[0] = timesten; stages[1] = swap;
  for (i=0; i<S+1; i++) {
    spreads[i] = SPGLOBAL;
    types[i]   = MPI_INT;
  }
  if (myrank() == 0) {
    printf("Inputs on P0: ");
    for (i=0; i<12; i++) {
      printf ("%d ", inputs[i]);
    }
     printf("\n");
  }
  Pipeline1for1 (S, stages, myrank()/S, spreads, types, (void *) inputs, inlen, inmul,
                 (void *) results, outlen, &outmul, 3, mycomm());
  if (myrank() == 0) {
    gathereddata = (int *) malloc (sizeof(int)*12);
  }
  for (i=0; i<outmul; i++) {
    MPI_Gather (&results[i], 1, MPI_INT, &gathereddata[i*4], 1, MPI_INT, 0, MPI_COMM_WORLD);
  }
  if (myrank() == 0) {
    printf("\nGathered results on P0: ");
    for (i=0; i<12; i++) {
      printf ("%d ", gathereddata[i]);
    }
    printf("\n");
  }
  MPI_Finalize();
  return 0;
}
```

Figure 4: Code for the global spread pipeline

5 Explicit interaction with skeletons: `Give` and `Take`

As a result of their "1for1" nature, the examples so far have used a simple interface between a skeleton and the code for the activities it coordinates: an activity is a function which is called by the system with its data item as parameter, and which must return its result in a similar format. Many applications are less regular in their interactions. For example, it is common for algorithms which are recognisably "pipelined" to have stages which produce more (or less) than one result for each item received. Similarly, some pipelines have no recognisable input and/or output, in the sense we have seen so far. Instead, the effects of the pipeline are seen in the state of the participating processes. eSkel has pipeline variants which allow such structures to be expressed. These requires the introduction of a new feature, the so-called `Give` and `Take` functions. The issue here is that the activity code must now have a means of signalling to the skeleton that it is ready to receive (or return) a new data item. Conceptually, an activity is now more like a process (although syntactically it is still just a C function), active for the duration of the skeleton call. For pipelines, these ideas are embodied by the skeleton function "`Pipeline`".

As an example, we implement the Sieve of Eratosthenes prime number generator. In the form presented here and illustrated in Figure 5 we have `P` processes generating the

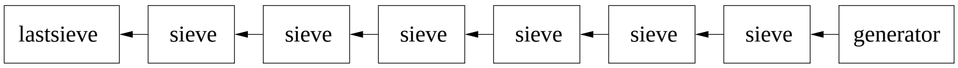

Figure 5: Sieve of Eratosthenes as a pipeline

first `P-1` primes. Process 0 is a generator, pumping out a stream of integers from 2 up to a "big enough" preset limit (we side-step the prime number theorem for simplicity of presentation). Other stages accept the first number received as "their" prime, then remove multiples of this from the subsequent stream. After the pipeline call, each process displays the prime it found, giving the output

```
Process 0 was the generator
Process 1 found 2
Process 2 found 3
Process 3 found 5
Process 4 found 7
Process 5 found 11
Process 6 found 13
Process 7 found 17
```

The full code for the example is presented in figure 6. It has a number of interesting features. Firstly, in the skeleton call itself,

```
Pipeline (P, stages, myrank(), spreads, types, NULL, 0, 0, NULL, 0, &outmul, 0, mycomm());
```

all the `NULL`s and `0`s indicate that there are neither input nor output data streams, since all the data is generated and stored within the pipeline. Now consider the stage functions. The first stage is "`generator`" a process which produces a stream of integers from `2` to predefined constant `N`.

```
void generator (void)
{
  int i, next = 2;

  while (next < N) {
    Give(&next, 1);
    next++;
  }
}
```

The function has the rather uninformative prototype required of all `Pipeline` stages (but note the difference from the standard prototype required of `Pipeline1for1` stages). The function calls `Give` to indicate generation of an item for the stage's output stream. Remember that the spread and type of such items have already been defined in the skeleton call (as `SPLOCAL` and `MPI_INT` respectively). It remains for the caller of `Give` to specify the location and length of each item as parameters.

The sieve processes (except for the final one) are described by the function `sieve`.

```
void sieve (void)
{
  int *cand, count = 1;

  myprime = Take (&count);
  while (cand = Take (&count)) {
    if ((*cand)% (*myprime)) Give (cand, 1);
  }
}
```

Notice that `myprime` is a global variable, whose value will persist beyond the pipeline call. As well as using `Give` to pass the sieved stream to the succeeding stage, `sieve` calls `Take` to receive new inputs. As above, the spread and type are already fixed. The call specifies the expected input length and returns a pointer to a dynamically allocated (within the call) buffer storing the data, or `NULL` if no further inputs are (or ever will be) available. The final stage is similar, but tidies up by consuming, but not forwarding, the remaining input stream.

```
void lastsieve (void)
{
  int *cand, count=1;

  myprime = Take (&count);
  while (cand = Take (&count)) {
    ;
  }
}
```

5.1 Other uses of `Give` and `Take`

In the preceding section we introduced `Give` and `Take` in the context of pipelines, for the sake of concreteness. However, it is important to realise that these functions are also active in the specification of other skeletons. In general, the programmer should think of `Give` and `Take` as the means by which details of interactions left more or less unspecified by a particular skeleton can be controlled dynamically. In `Pipeline1for1` there were no such gaps and so `Give` and `Take` are not required (and indeed must not be used). In the more flexible `Pipeline`, they control the flow of data down the stages, within the constrained connectivity and spread/type context already laid down by the skeleton. In our family of task farming skeletons we will see that they are used to guide aspects of what it means to be a "worker". In summary, while the API for `Give` and `Take` is fixed, their semantics are determined by the currently active skeleton.

```c
#include "../eSkel.c"
#define P 8
#define N 100

int *myprime;

void generator (void)
{
  int i, next = 2;

  while (next < N) {
    Give(&next, 1);
    next++;
  }
}

void sieve (void)
{
  int *cand, count = 1;

  myprime = Take (&count);
  while (cand = Take (&count)) {
    if ((*cand)% (*myprime)) Give (cand, 1);
  }
}

void lastsieve (void)
{
  int *cand, count=1;

  myprime = Take (&count);
  while (cand = Take (&count)) {
    ;
  }
}

int main (int argc, char *argv[])
{
  spread_t spreads[P+1];
  MPI_Datatype types[P+1];
  int outmul;
  void (*stages[P])(void);
  int i;

  MPI_Init(&argc, &argv);
  SkelLibInit();
  stages[0] = (void (*) (void))    generator;
  for (i=1; i<P-1; i++) {
    stages[i] = (void (*) (void))  sieve;
  }
  stages[P-1] = (void (*) (void))  lastsieve;

  for (i=0; i<P+1; i++) {
    spreads[i] = SPLOCAL; types[i]    = MPI_INT;
  }
  outmul = 0;
  Pipeline (P, stages, myrank(), spreads, types, NULL, 0, 0, NULL, 0, &outmul, 0, mycomm());
  for (i=0; i<P; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (myrank() == i) {
      if (myrank() > 0)
        printf("Process %d found %d\n", myrank(), *myprime);
      else printf("Process 0 was the generator\n");
    }
  }
  MPI_Finalize();
  return 0;
}
```

Figure 6: Code for the primes pipeline

6 Farm skeletons

The term *Task Farming* is widely used to describe parallel applications which have the following characteristics. An ordered, or otherwise structured collection of data items (the *tasks*) are each to be processed by the same operation. Processing of distinct items is independent (in other words no state is carried by the operation from one instance to another) and so may be undertaken in parallel. Furthermore, it is implicitly understood that the expected processing time of distinct items will vary significantly and unpredictably, making a static scheduling of tasks to a smaller number of processes potentially poorly load balanced, though semantically undetectable. A task farmed solution implements a dynamically scheduled allocation with a view to ensuring better balance and faster overall completion. In many descriptions a special process, the *farmer* is explicitly set aside as scheduler, with the remaining *worker* processes undertaking tasks at the farmer's behest. A typical example might be a "parameter sweep " simulation, in which a system is simulated for a range of starting configurations. Each simulation is a task, running the same program with different data. Convergence properties of the system being simulated typically result in widely varying task execution times.

eSkel has a family of task farming skeletons. For illustration we use a toy application, investigating Ulam's function (aka the $3n+1$ problem). Our program computes and prints the number of iterations required to terminate when seeded by each integer between 1 and N. Along the way, we also note the number of tasks computed by each process and the overall execution time. Since each task returns exactly one result (the iteration count), we can use a "1for1" farm. The worker function is

```
eSkel_atom_t *work (eSkel_atom_t *task)
{
  int n, iters;

  n = ((int *) (task->data))[0];
  iters = 0;
  while (n>1) {
    if (n%2) n=3*n+1;
    else n=n/2;
    iters++;
  }
  ((int *) (task->data))[0] = iters;
  tasksdone++;
  return task;
}
```

Notice that for convenience we reuse the input task structure for the result. The inputs and result buffers are created in a distributed fashion, but as we have seen, could equally well be created by a single process. There are no implications either way for the eventual dynamic distribution.

```
  tasks   = (int *) malloc (sizeof(int)*MULT);
  ulam    = (int *) malloc (sizeof(int)*MULT);
  for (i=0; i<MULT; i++) {
    tasks[i] = myrank()*MULT + i + 1;
  }
```

The call to `Farm1for1` is very similar in structure to that for `Pipeline1for1`.

```
Farm1for1 (P, work, myrank(), (void *) tasks, 1, MULT, SPLOCAL, MPI_INT,
           (void *) ulam, 1, &outmul, SPLOCAL, MPI_INT, MULT, mycomm());
```

eSkel's concept of "farm" is relatively abstract in that no process is explicitly set aside to act as farmer. It is the responsibility of the implementation to arrange for dynamic scheduling of tasks to the full set of processes (for example, by multi-threading one process as farmer, or indeed multi-threading all processes as a distributed farmer). The first three parameters specify the number of workers (which may be internally parallel), the worker function and the allocation of processes to workers (just as processes were allocated to stages in pipelines). The complete program is presented in figure 7. One particular run produced the following output:

```
Ulam values: 0 1 7 2 5 8 16 3 19 6 14 9 9 17 17 4 12 20 20 7
P0 ran 6 task(s) in 0.00 seconds
P1 ran 6 task(s) in 0.00 seconds
P2 ran 4 task(s) in 0.00 seconds
P3 ran 4 task(s) in 0.00 seconds
```

where the execution time was too small to register. For a larger example, with `N` being 20000) and listing of iteration counts omitted, the output was

```
P0 ran 5331 task(s) in 35.67 seconds
P1 ran 4912 task(s) in 35.67 seconds
P2 ran 4862 task(s) in 35.67 seconds
P3 ran 4895 task(s) in 35.67 seconds
```

where the overall execution times are identical because of synchronisation at the end of farm skeleton call.

6.1 More farms

By analogy with its pipeline skeletons, eSkel has a more flexible `Farm` skeleton, in which the worker function becomes a process responsible for its own calls to `Give` and `Take`. Further extensions of the farming skeleton set are provided by the pair `SimpleFarm` and `SimpleFarm1for1`. Simple farms provide a more traditional form of farm in which

- one process (that with rank 0) is set aside as the farmer;
- all workers are internally sequential.

This allows the API to be simplified and, more pragmatically, allows implementation on all MPIs, irrespective of whether threading is fully supported. In contrast, the current implementations of the `Farm` skeleton pair require a fully thread-safe MPI. The amendments required to express our Ulam's function example as a simple farm are very easy. The worker code is unchanged. Only the call itself requires amendment (in fact simplification, since we remove the parameters dealing with the number of processes and their allocation to workers). The call becomes

```
SimpleFarm1for1 (work, (void *) tasks, 1, MULT, SPLOCAL, MPI_INT,
                 (void *) ulam, 1, &outmul, SPLOCAL, MPI_INT, MULT, mycomm());
```

and a sample output (with N=20000 again) is

```
P0 ran 0 task(s) in 50.87 seconds
P1 ran 6822 task(s) in 50.87 seconds
P2 ran 6706 task(s) in 50.87 seconds
P3 ran 6472 task(s) in 50.87 seconds
```

highlighting the fact that P0 was a dedicated farmer. No comparison should be made to the previous execution time, since the example was run on a timeshared parallel server.

```c
#include "../eSkel.c"
#define P 4
#define MULT 5
#define N (MULT*P)
int tasksdone=0;

int *tasks;
int *ulam;
int *allulam;

double mpi_start, mpi_end;

eSkel_atom_t *work (eSkel_atom_t *task)
{
  int n, iters;

  n = ((int *) (task->data))[0];
  iters = 0;
  while (n>1) {
    if (n%2) n=3*n+1;
    else n=n/2;
    iters++;
  }
  ((int *) (task->data))[0] = iters;
  tasksdone++;
  return task;
}

int main (int argc, char *argv[])
{
  int i;
  int outmul;

  MPI_Init(&argc, &argv);
  SkelLibInit();

  tasks  = (int *) malloc (sizeof(int)*MULT);
  ulam   = (int *) malloc (sizeof(int)*MULT);
  for (i=0; i<MULT; i++) {
    tasks[i] = myrank()*MULT + i + 1;
  }

  mpi_start = MPI_Wtime();
  Farm1for1 (P, work, myrank(), (void *) tasks, 1, MULT, SPLOCAL, MPI_INT,
             (void *) ulam, 1, &outmul, SPLOCAL, MPI_INT, MULT, mycomm());
  mpi_end = MPI_Wtime();

  if (myrank() == 0) {
    allulam = (int *) malloc (sizeof(int)*N);
  }

  MPI_Gather (ulam, MULT, MPI_INT, allulam, MULT, MPI_INT, 0, MPI_COMM_WORLD);

  if (myrank() == 0) {
    printf("\nUlam values: ");
    for (i=0; i<N; i++) {
      printf ("%d ", allulam[i]);
    }
    printf("\n");
  }

  printf("P%d ran %d task(s) in %.2f seconds\n", myrank(), tasksdone, mpi_end-mpi_start);
  MPI_Finalize();
  return 0;
}
```

Figure 7: Code for the Ulam farm

7 Supporting divide & conquer

Divide & Conquer is probably the best known example of a generic problem solving technique. Simply put, it applies to problems in which the overall solution can be described as a combination of solutions to a number of sub-problems, each of the same type as, and derived from the original. To bound the inherent recursion, a direct approach must also be derived for problem instances which are in some sense "simple". A divide & conquer algorithm for a specific problem is then produced by describing appropriate operations for dividing problems, combining solutions and identifying and solving simple instances. There are numerous well known examples, with quicksort and mergesort often cited in introductory discussions. With respect to parallelism, the approach itself has inherent potential in that sub-problems may be solved concurrently. Specific instances may also have scope for parallelism within the various customising operations.

Divide & conquer is a powerful generic technique, but this makes it difficult to devise an implementation which is efficient in all possible circumstances. Many examples of divide & conquer algorithms are naturally more constrained. For example, quicksort has trivial combine operation, mergesort has a trivial and data independent divide operation. eSkel recognises this by providing a range of skeletons, each supporting a particular form of divide & conquer.

7.1 The butterfly skeleton

The eSkel `Butterfly` skeleton supports divide & conquer algorithms which have the following properties

- all activity occurs in the divide phase;
- division is process, rather than data oriented, with the number of processes in each branch being halved at each level (and therefore starting from a power of 2);
- interactions within a level occur between pairs of processes which will have the same rank (in distinct groups) at the next level down.

This structure is common in many algorithms originally derived with implementation on a binary hypercube network in mind. Indeed, it could equally well be called "DownCube" (suggesting working down the dimensions of a hypercube).

As a concrete example, consider the well known "hyperquicksort" algorithm. Initially, each process stores an unordered portion of the data to be sorted. The goal is to produce a distributed array, in which the same data is sorted within each process and by process rank between processes. The core of the algorithm is a divide step in which each active group of processes (initially, a single group containing all processes),

1. determine a pivot;
2. partition their local data against that pivot;

3. exchange “smaller” and “larger” items with the process which will have the same rank in what will become the “other” group at the next level;

The net effect is to rearrange the data so that all “smaller” items are in one sub group at the next level, and all “larger” items in the other group. The sub groups can then sort independently (and concurrently) at the next level. The sorting is completed by a final step in which each process sorts its resulting data internally (some variants make this a first step, to assist partitioning).

eSkel `Butterfly` supports such algorithms in two ways. Firstly, given an initial group of processes (of size some power of 2) it arranges for a given work function to be executed by the processes of each group at each level of a sequence of groupings, starting with the original group, dividing each group in half (blockwise by rank) at each level, and finishing with groups of size 2 (NB not 1, since running a process in a group of size 1 is MPI’s inherent SPMD mode, and in practice many algorithms do something different at this level anyway). Secondly, it provides a new explicit interaction function `Exchange` which enacts an exchange of data between the pairs of *partner* processes in a group. Partners are processes within a group which will share the same rank in the two groups formed from their current group at the next level (in hypercube terms, they occupy corresponding positions in the sub-cubes at the next level down).

The following simple example illustrates the skeleton in action. Each of eight processes creates some data by initialising an array to contain three copies of its rank. The `level` function calls `Exchange` to swap arrays between partners.

```
void level (void)
{
  int i, length = THINGS;

  b = Exchange(&length, a, THINGS);
  a = b;
}
```

Since this happens at each level, the net effect of the `Butterfly` is to swap data between process i and process $7 - i$ as can be seen by the printed results:

```
P0 starts with 0 0 0
P1 starts with 1 1 1
P3 starts with 3 3 3
P5 starts with 5 5 5
P6 starts with 6 6 6
P7 starts with 7 7 7
P2 starts with 2 2 2
P4 starts with 4 4 4
P0 finishes with 7 7 7
P1 finishes with 6 6 6
```

```
P2 finishes with 5 5 5
P3 finishes with 4 4 4
P5 finishes with 2 2 2
P4 finishes with 3 3 3
P7 finishes with 0 0 0
P6 finishes with 1 1 1
```

A number of points should be noted about the call to `Butterfly` itself.

```
Butterfly (L, level, MPI_INT, mycomm());
```

Most notably, and in contrast to previously introduced skeletons, there are no data or result parameters. The `level` function works with the data within its process' state. The type parameter (`MPI_INT` here) concerns the data handled by calls to `Exchange`, and also to `Give` and `Take` which are defined as interacting with the current partner process. The first parameter, `L`, is the number of levels, and must be the base 2 logarithm of the size of the communicator passed as the final parameter.

```c
#include "../eSkel.c"

#define P 8
#define L 3        // number of levels, ie log2 8
#define THINGS 3

int *a;
int *b;

void level (void)
{
  int i, length = THINGS;

  b = Exchange(&length, a, THINGS);
  a = b;
}

int main (int argc, char *argv[])
{
  int i, p;

  MPI_Init(&argc, &argv);
  SkelLibInit();

  a = (int *) malloc (THINGS * sizeof(int));
  for (i=0; i<THINGS; i++) a[i] = myrank();
  for (p=0; p<P; p++) {
    if (myrank() == p) {
      printf("P%d starts with ", myrank());
      for (i=0; i<THINGS; i++) {
          printf("%d ", a[i]);
      }
      printf("\n");
    }
  }

  Butterfly (L, level, MPI_INT, mycomm());

  for (p=0; p<P; p++) {
    if (myrank() == p) {
      printf("P%d finishes with ", myrank());
      for (i=0; i<THINGS; i++) {
          printf("%d ", a[i]);
      }
      printf("\n");
    }
  }
  MPI_Finalize();
  return 0;
}
```

Figure 8: Code for the butterfly example