Flexible Skeletal Programming with eSkel

Anne Benoit, Murray Cole, Stephen Gilmore and Jane Hillston

School of Informatics, The University of Edinburgh, James Clerk Maxwell Building,
The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
enhancers@inf.ed.ac.uk
http://homepages.inf.ed.ac.uk/mic/eSkel

Abstract. We present an overview of *eSkel*, a library for skeletal parallel programming. *eSkel* aims to maximise the conceptual flexibility afforded by its component skeletons and to facilitate dynamic selection of skeleton compositions. We present simple examples which illustrate these properties, and discuss the implementation challenges which the model poses.

1 Introduction

The skeletal approach to parallel programming is well documented in the research literature (see [1–4] for surveys). It observes that many parallel algorithms can be characterised and classified by their adherence to one or more of a number of generic patterns of computation and interaction. For instance, a variety of applications in image and signal processing are naturally expressed as process pipelines, with parallelism both between pipeline stages, and within each stage by replication and/or more traditional data parallelism [5].

Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations but implementations which recognise these to enhance performance. This level of abstraction makes it easier for the disciplined programmer to experiment with a variety of parallel structurings for a given application, enabling a clean separation between structural aspects and the application specific details. Meanwhile, the explicit structural information provided by skeletons enables static and dynamic optimisations of the underlying implementation.

In [2] we presented a number of observations and proposals designed to facilitate the popularisation of the skeletal approach, and outlined the first prototype of *eSkel*, a library of skeleton operations for C and MPI. In the light of subsequent experiences we isolated and described two key concepts, *nesting mode* and *interaction mode*, which capture important choices in the design of any skeletal programming framework [6]. This caused us to substantially refine and partly redesign *eSkel*. While these and other concepts embedded in *eSkel* may be found in various combinations in previous skeletal systems, we believe that this paper discusses the first attempt to enumerate and compose them systematically and explicitly within a single framework.

This paper describes current work on the latest instantiation of *eSkel*. We describe the ways in which key concepts have been incorporated into the programming model and API in order to maximise the flexibility with which the

programmer can shape and reshape skeletal programs, and on the substantial implementation challenges which this flexibility creates.

2 Dynamic Selection of Skeletons and Modes

eSkel is designed to provide the skeletal parallel programmer with flexibility in a number of ways. As in the initial prototype [2], a skeleton abstracts the pattern of interactions between a number of component *activities*, each of which may be internally parallel, and which may choose to invoke further skeletons. The skeletons provided each encompass the ability to specify the use of either localised or distributed data (the data *spread*), and the choice of either explicit or implicit *interaction mode* [6] to trigger interactions between the various activities. The aspects of the interface which capture these choices have been rationalised. Guided by the concept of *nesting mode* [6], the programmer can choose to incorporate both *transient* and *persistent* skeleton calls within the same skeleton nest. Crucially, all of these decisions, as well as the choice of overall skeleton structure for each phase of a program, are made dynamically, and can therefore be based, where appropriate, on information gathered during execution (for example, data values or sizes). In summary the significant improvements over *eSkel*'s first version [2] are as follows.

- The definition of 'families' of related skeletons, capturing distinctions in interaction style, have been replaced by the explicit selection of interaction mode. For example, there is now only one pipeline skeleton, which can be parameterized to behave as either of its predecessors.
- All skeletons in the original *eSkel* were (implicitly) in transient mode. There were no persistent nestings. Both transient and persistent modes are now available (by setting the corresponding parameter).
- The data model has been revised and extended to incorporate the concept of *molecules*, enabling the expression of skeletons (such as `Haloswap`) in which several distinct pieces of data are exchanged at each interaction.

2.1 *eSkel* interface

The current specification of *eSkel* defines five skeletons, each with flexibility in the dimensions discussed above. We will focus on the two of these, `Pipeline` and `Deal`, which have so far been implemented in the current prototype, since in combination these can illustrate all the points we wish to make. The reader is referred to the *eSkel* homepage [7] for discussion of the `Farm`, `HaloSwap` and `Butterfly` skeletons.

The `Pipeline` skeleton abstracts classical pipeline parallelism. It allows an indexed set of activities (called "stages") to be chained together and applied to a sequence of inputs. Data is passed from a stage to its successor following the natural ordering on stage indices. The user can choose between an implicit pipeline, in which the transfer of the data is done automatically, constraining

the stages to produce one result for each input, or an explicit mode, in which a stage can produce arbitrarily many results without necessarily consuming data (for example, a generator or filter stage). In the latter case, the data transfer is controlled by the programmer, by calls to the generic interaction functions `Give` and `Take`.

The `Deal` skeleton is similar to a traditional farm, but with tasks distributed strictly cyclically to workers. Thus it is most appropriately used when the workload associated with individual tasks is expected to be homogeneous. This skeleton is useful nested in a pipeline, in order to internally replicate a stage. `Deal` semantics require the ordering of outputs from the skeleton to match that of the corresponding inputs, irrespective of the internal speed of the workers.

The price for this flexibility is paid in the complexity of the API, where each skeleton call must specify its choices. In addition to the issues already discussed, the pragmatic decision to borrow and build upon MPI's data model with its already substantial parameter set, leads to rather long parameter lists. However, when understood as a set of groups, each addressing orthogonal issues, the parameters become more conceptually manageable. We now present these groupings informally. Their formal use will be illustrated in Section 2.2.

One set of parameters deals with the called skeleton's *data buffers*. As with any MPI collective operation, there are distributed input and output buffers, each requiring the standard MPI pointer, type, length definition. These, together with the enclosing MPI communicator, capture the call's interface to the rest of the program. A second set of parameters deals with the call's *internal structure and interfaces*. For example, in a pipeline, we must describe the number of stages and the types and data modes on their interfaces, and the allocation of processes from the calling group to the distinct stages (captured by borrowing MPI's *colouring* parameter mechanism to describe the required sub-groups). Finally, a group of parameters describe the details of the skeleton's various *activities* (stages in a `Pipeline`, workers in a `Deal`) with a choice of interaction mode and a pointer to a function for each.

In the interests of regularity, we have chosen a single generic interface for all functions which contain code for skeleton activities. The essence of activities is that they *interact* in predefined patterns, via the skeleton infrastructure, with each other and with skeleton calling programs. Each such interaction may involve different numbers of data atoms, according to the semantics of the skeleton. For example, "getting the next task" in a `Deal` skeleton involves a single atom, whereas "updating the halo" in a `HaloSwap` skeleton will involve two atoms for each neighbour (one arriving, one leaving). The `eSkel_molecule_t` type collects atoms involved in an interaction into a single sequence. Activity functions both accept and return a single item, of type `eSkel_molecule_t *`. The specification of each skeleton defines its detailed molecule usage.

2.2 Examples

We present two variants of a toy program in order to concisely illustrate the ease with which the *eSkel* programmer can describe and revise the parallel structure

of an application. The first version illustrates the use of persistent nesting and implicit interactions. The second uses transient nesting and explicit interactions.

Example 1: persistent nesting and implicit interactions

This program calls a `Pipeline` skeleton, in which the second stage contains a persistently nested `Deal`. Some of the activities (pipeline stages and deal workers) are assigned several processes, and process data with "global spread" (i.e. the data is distributed across such processes and accessed in SPMD style).

The constant declarations define the number of pipeline stages (`STAGES`), the number of workers (`DEALWORKERS`) in the deal, and the number and size of input items (`INPUTNB`, `INPUTSZ`). When run with 8 processes, the input to this pipeline consists of a total of (8*4*5) integers, which are treated as 4 (INPUTNB) inputs to the pipeline, each of which is constructed from a 5 (INPUTSZ) integer contribution per process. The first stage in the pipeline has been assigned two processes. Thus, since we are in "global spread" mode, each pipeline input will be fed to the two first stage processes as a block of (8*5)/2 integers each.

The `main` function (lines 31-57) makes the pipeline call. It defines all the pipeline parameters and assigns processes to the stages. Lines 34-39 describe the structure of the pipeline: all inter-stage data transfers use global spread (`SPGLOBAL`) and involve sequence of integers. All activities except the second stage use "implicit" mode (`IMPL`) transfers (in other words, handled implicitly by the skeleton). The second stage has "devolved" mode (`DEV`), indicating that this will be inherited from a persistently nested skeleton (the deal in this case). All stages will execute the `SimpleStage` activity, except the second, which executes `DealStage`. Lines 42-43 create some (distributed) input data. Lines 45-48 compute an assignment of processes to stages. Line 50 creates the (distributed) output buffer. The pipeline call itself is on lines 52-54, and is passed the various parameters as created, together with details of the input and output buffers in the style of a conventional MPI collective operation.

Function `SimpleStage`, on lines 6-12, describes the actions of the first, third and fourth stages. Since implicit interaction mode is used, the function takes an *eSkel* molecule as a parameter and returns a molecule. Here we simply add 1000 to each integer in the block and return the modified molecule. In contrast, the second stage is described by `DealStage`. It computes an assignment of processes (from the stage) to workers in the internal deal, then calls the `Deal` skeleton on lines 26-28. The data mode is stream (`STRM`) indicating that the data are streamed into the deal from the *parent* skeleton. The input and output buffer related parameters are therefore redundant (all the `NULL`s and `0`s). Deal workers run the function `DealWorker`. In our simple example this is identical to `SimpleStage`. We leave the duplication to emphasise that these functions can be programmed independently, if appropriate.

Figure 2 illustrates the requested assignment of processes to stages and workers, and the different communicators involved. This will help to clarify the discussion of *eSkel* implementation in Section 3. We have a 4-stage pipeline where the second stage is a deal with two workers, of which one has two processes. **Px**

```
1  #define STAGES  4       // Number of pipeline stages
2  #define DEALWORKERS 2   // Number of workers in the deal
3  #define INPUTNB 4       // Input multiplicity
4  #define INPUTSZ 5       // Size of each input datum, per process
5
6  eSkel_molecule_t * SimpleStage (eSkel_molecule_t *thingy) {
7    int i;
8    for (i=0;i<thingy->len[0];i++) {
9        ((int *) thingy->data[0])[i] += 1000;
10   }
11   return thingy;
12 }
13
14 eSkel_molecule_t * DealWorker (eSkel_molecule_t *thingy) {
15   int i;
16   for (i=0;i<thingy->len[0];i++) {
17       ((int *) thingy->data[0])[i] += 1000;
18   }
19   return thingy;
20 }
21
22 void DealStage (void) {
23   int workercol, outmul;
24   if ((myrank() < 1))  workercol = 0; else workercol = 1;
25
26   Deal (DEALWORKERS, IMPL, DealWorker, workercol, STRM,
27         NULL, 0, 0, SPGLOBAL, MPI_INT,
28         NULL, 0, &outmul, SPGLOBAL, MPI_INT, 0, mycomm());
29 }
30
31 int main (int argc, char *argv[])
32 {
33   int i, j, p, next, outmul, mystagenum, mymult, *inputs, *results;
34   spread_t spreads[STAGES+1]   = {SPGLOBAL, SPGLOBAL, SPGLOBAL, SPGLOBAL, SPGLOBAL};
35   MPI_Datatype types[STAGES+1] = {MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT};
36   Imode_t imodes[STAGES]       = {IMPL, DEV, IMPL, IMPL};
37
38  eSkel_molecule_t *(*stages[STAGES])(eSkel_molecule_t *) =
39    {SimpleStage, (eSkel_molecule_t * (*)(eSkel_molecule_t *))DealStage, SimpleStage, SimpleStage};
40
41   MPI_Init(&argc, &argv); SkelLibInit();
42   inputs = (int *) malloc (sizeof(int)*INPUTNB*INPUTSZ);
43   for (i=0; i<INPUTNB*INPUTSZ; i++) inputs[i] = myrank()*100 + i + 1;
44
45   if      (myrank()<2)  mystagenum = 0;
46   else if (myrank()<5)  mystagenum = 1;
47   else if (myrank()==6) mystagenum = 2;
48   else                  mystagenum = 3;
49
50   results = (int *) malloc (sizeof(int)*INPUTNB*INPUTSZ);  // Create output buffer
51
52   Pipeline (STAGES,  imodes, stages, mystagenum, BUF, spreads, types,
53             (void *) inputs, INPUTSZ, INPUTNB, (void *) results, INPUTSZ, &outmul,
54             INPUTNB*INPUTSZ, mycomm());
55
56   MPI_Finalize();  return 0;
57 }
```

Fig. 1. Example 1

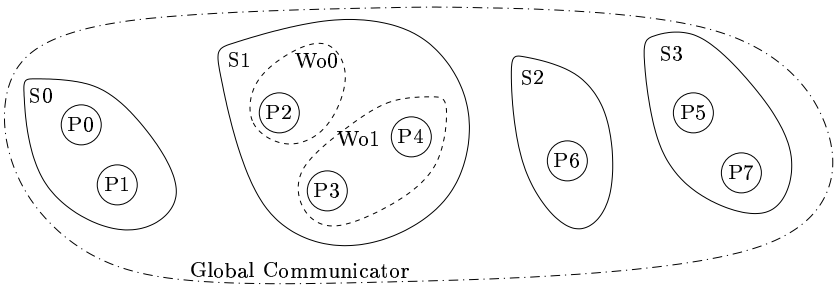

Fig. 2. Example 1 - Mapping of the processes

represents process x ($0 \leq x \leq 7$), as indexed in the original `MPI_COMM_WORLD`. **Si** represents stage i ($0 \leq i \leq 3$), and **Woj** represents worker j ($0 \leq j \leq 1$).

Example 2: transient nesting and explicit interactions
Our second example demonstrates the use transient nesting, explicit interactions and local data spread, and also illustrates the ease with which the programmer can experiment with the high level structure of an application. Working from the program of example 1, the programmer decides to

- turn the second stage (formerly the `Deal`) into a simple single process stage;
- reallocate the two spare processes from the second stage to the first stage;
- adapt the first stage (which now has four processes) so that every second input is processed by a dynamically created four stage pipeline, with one process per stage. These are transiently nested pipelines, in contrast to the persistently nested `Deal` of example 1. Other inputs are treated as in the original simple stages.

The structural changes at the outer level are captured in lines of 3, 61, 63, 64, 70 and 71 of Figure 3. Line numbers in parentheses are those of the corresponding lines in Figure 2. The rest of `main` is unchanged, and is omitted here. The behaviour of the new first stage, and its sub-stages are described by the new functions `StrangeStage` and `SubStage`. `StrangeStage` creates and calls the nested pipeline for every second input (controlled by line 44). The nested sub-stages, called with explicit (`EXPL`) interaction mode (line 35), use *eSkel* functions `Give` and `Take` to control interactions with neighbouring stages (lines 21 and 26).

3 Implementation challenges

We now consider the challenges raised by *eSkel*'s implementation, focusing particularly on the call tree, which is built dynamically, and allows processes to find communication partners. We also discuss the management of the input and output buffers.

```
 3  #define SUBSTAGES 4     // Number of stages in the nested pipeline


15  void SubStage (void) {
16    int i, **len;
17    eSkel_molecule_t *tempitem;
18
19    len = (int **) malloc (sizeof (int *));
20    tempitem = (eSkel_molecule_t *) malloc (sizeof (eSkel_molecule_t));
21    while (tempitem->data = Take(len)) {
22      tempitem->len = *len;
23      for (i=0;i<tempitem->len[0];i++) {
24        ((int *) tempitem->data[0])[i] += 250;
25      }
26      Give(tempitem->data, tempitem->len);
27    }
28  }
29
30  eSkel_molecule_t * StrangeStage (eSkel_molecule_t *thingy) {
31    static int count=0;
32    int i, outmul;
33    spread_t spreads[SUBSTAGES+1]   = {SPLOCAL, SPLOCAL, SPLOCAL, SPLOCAL, SPLOCAL};
34    MPI_Datatype types[SUBSTAGES+1] = {MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT};
35    Imode_t imodes[SUBSTAGES]       = {EXPL, EXPL, EXPL, EXPL};
36
37    eSkel_molecule_t *(*stages[SUBSTAGES])(eSkel_molecule_t *) =
38       {(eSkel_molecule_t * (*)(eSkel_molecule_t *))SubStage,
39        (eSkel_molecule_t * (*)(eSkel_molecule_t *))SubStage,
40        (eSkel_molecule_t * (*)(eSkel_molecule_t *))SubStage,
41        (eSkel_molecule_t * (*)(eSkel_molecule_t *))SubStage};
42
43
44    if (count++ % 2) {  // handle it directly
45       for (i=0;i<thingy->len[0];i++) {
46           ((int *) thingy->data[0])[i] += 1000;
47       }
48    } else {            // handle with a transient pipeline!
49       Pipeline (SUBSTAGES,  imodes, stages, myrank(), BUF, spreads, types,
50                 thingy->data[0], 1, thingy->len[0],  thingy->data[0], 1 , &outmul,
51                 thingy->len[0], mycomm());
52    }
53    return thingy;
54  }

(36)61    Imode_t imodes[STAGES]       = {IMPL, IMPL, IMPL, IMPL};

(38)63    eSkel_molecule_t *(*stages[STAGES])(eSkel_molecule_t *)
(39)64      = {StrangeStage, SimpleStage, SimpleStage, SimpleStage};

(45)70    if (myrank()<4)       mystagenum = 0;
(46)71    else if (myrank()<5)  mystagenum = 1;
```

Fig. 3. Example 2 - changes from example 1

3.1 Building the call tree dynamically

In order to correctly implement the communications which have been abstracted by the skeleton calls, it is necessary for the processes assigned to the various activities to understand their position within the overall skeleton nest. Since the structure of the nest emerges dynamically, and independently within distinct branches, this investigation can only be performed dynamically, at the point at which no further changes to the structure are possible. In *eSkel* this occurs when each activity calls its first explicit (or implicit) interaction. Building the call tree requires collaborative communications between all processes in the nest.

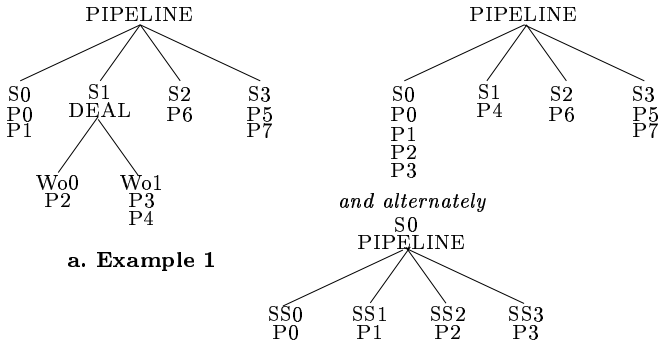

Fig. 4. Call trees for the two examples

The root of the tree corresponds to the main skeleton call. The children describe the different activities, and if there are some persistently nested skeletons, they appear in the tree. The transiently nested structure is not built in the main tree. The sub-tree will be built dynamically when the transient skeleton call is performed. Fig. 4 represents the call tree for the examples introduced in the previous section. The tree of the nested pipeline of Fig. 4b is created at each call of this pipeline.

3.2 Finding communication partners

Communication partners are found by traversing the call tree. For example, in Fig. 4a, we discover that Wo1 takes data from S0 and gives data to S2. Interactions with a neighbouring deal require different partners at each interaction (because of the cyclic nature of interactions in that skeleton). In our example, when S0 performs a `Give`, it needs to give data alternately to Wo0 and Wo1. To ensure this cyclic distribution we keep track of the number of `Give` and `Take` calls performed. This allows us to find the appropriate partner.

Particular attention must be paid to boundary cases. When S0 performs a `Take`, it cannot find a communication partner in the tree. It needs instead to take the data from the input buffer to the pipeline call. Similarly, S3's `Give` must store data in the output buffer. We detail the management of these buffers below.

3.3 Managing the input and output buffers

The *input buffer* must be created collaboratively by all the processes, since each of them potentially has some of the data at the point of the skeleton call. It is created just after the call tree has been built, during the first interaction of each activity. At that time, the processes which have to take data from the input buffer (processes of S0 in our pipeline examples) collect all the data through group communication, and build the input buffer.

The *output buffer* is created by the processes which write results into the buffer (processes of S3 in the first example). After completion of the skeleton nest, the results are distributed among all the processes.

More care must be taken when the group of processes dealing with the input or output buffer are part of a deal. For example, if the nested deal was in S0 instead of S1, both Wo0 and Wo1 need to take data from the input buffer while maintaining the cyclic order. To address this issue, we dynamically create a thread which distributes the data to the workers. A symmetrical approach is taken for the output buffer.

3.4 Context of use

The *eSkel* library operates within the context of a running MPI program which has already created its fixed number of processes. Moreover, a fully thread safe version of MPI (in the sense of `MPI_THREAD_MULTIPLE`) must be used. We are developing *eSkel* using the Los Alamos Message Passing Interface LA-MPI [8] and the Sun HPC Cluster Tools [9].

4 Future Work

A full implementation of *eSkel* is in progress, with the current status reported on the project web pages [7]. After completion of the initial prototype, work will focus on development of a set of demonstrator applications, internal optimisations and expansion of the skeleton set. We will also consider ways in which the API can be simplified, for example by moving to a more powerfully descriptive host language such as Java, as and when the Java-MPI combination gains more widespread practical acceptance. Current work on *eSkel* is conducted under the umbrella of the *Enhance* project [10], which exploits the modelling power of the Performance Evaluation Process Algebra (PEPA) [11] to predict the performance of skeletally structured Grid applications, with a view to assisting in their scheduling and re-scheduling in the presence of the dynamically heterogeneous performance and availability characteristics of Grid computing platforms.

Acknowledgements

The authors are supported by the Enhance project ("Enhancing the Performance Predictability of Grid Applications with Patterns and Process Algebras") funded by the Engineering and Physical Sciences Research Council under grant number GR/S21717/01.

References

1. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press & Pitman, ISBN 0-262-53086-4 (1989)
2. Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing **30** (2004) 389–406
3. Pelagatti, S.: Structured Development of Parallel Programs. Taylor & Francis, London (1998)
4. Rabhi, F., Gorlatch, S., eds.: Patterns and Skeletons for Parallel and Distributed Computing. Springer Verlag, ISBN 1-85233-506-8 (2003)
5. Subhlok, J., O'Hallaron, D., Gross, T., Dinda, P., Webb, J.: Communication and memory requirements as the basis for mapping task and data parallel programs. In: Proceedings of Supercomputing '94, Washington, DC (1994) 330–339
6. Benoit, A., Cole, M.: Two Fundamental Concepts in Skeletal Parallel Programming. In: Computational Science - ICCS 2005. Number 3515 in LNCS, Atlanta, USA, Springer (2005) 764–771
7. Benoit, A., Cole, M.: `http://homepages.inf.ed.ac.uk/mic/eSkel` (2005)
8. Aulwes, R.T., Daniel, D.J., Desai, N.N., Graham, R.L., Taylor, L.D.R.M.A., Woodall, T.S., Sukalski, M.W.: Architecture of LA-MPI, A Network-Fault-Tolerant MPI. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe, New Mexico, IEEE Computer Society (2004)
9. Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303-4900, USA: Sun HPC ClusterTools 3.1 User's Guide. (2000)
10. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: `http://groups.inf.ed.ac.uk/enhance` (2004)
11. Gilmore, S., Hillston, J.: The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In: Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation. Number 794 in LNCS, Vienna, Springer-Verlag (1994) 353–368