Available online at www.sciencedirect.com
SCIENCE DIRECT®
Parallel Computing 30 (2004) 389–406
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming

Murray Cole

Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh, King's Buildings, Edinburgh, EH9 3JZ Scotland, UK

Received 15 September 2002; received in revised form 15 August 2003; accepted 20 December 2003

Abstract

Skeleton and pattern based parallel programming promise significant benefits but remain absent from mainstream practice. We consider why this situation has arisen and propose a number of design principles which may help to redress it. We sketch the *eSkel* library, which represents a concrete attempt to apply these principles. *eSkel* is based on C and MPI, thereby embedding its skeletons in a conceptually familiar framework. We present an application of *eSkel* and analyse it as a response to our manifesto.

Keywords: Parallel programming; Libraries; Skeletons; Patterns; MPI

1. Introduction

The skeletal approach to parallel programming is well documented in the research literature (see [3,31] for surveys and Sections 2 and 6 for a discussion of many related projects). 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 example, many diverse applications share the underlying control and data flow of the pipeline paradigm [2].

Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations

E-mail address: mic@inf.ed.ac.uk (M. Cole).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2003.12.002

but implementations which recognise these to enhance performance. In this way, it promises to address many of the traditional issues within the parallel software engineering process:

- it will *simplify* programming by raising the level of abstraction;
- it will enhance *portability* and *re-use* by absolving the programmer of responsibility for detailed realisation of the underlying patterns;
- it will improve *performance* by providing access to carefully optimised, architecture specific implementations of the patterns;
- it will offer scope for static and dynamic *optimisation*, by explicitly documenting information on algorithmic structure (e.g. sharing and dependencies) which would often be impossible to extract from equivalent unstructured programs.

These aspirations are common to a number of models which have proved very successful within the wider world of software engineering, most notably *structured programming*, *object-oriented programming* and *design patterns*. Yet skeletal programming has still to make a substantial impact on mainstream practice in parallel applications programming. In contrast, MPI was designed to address similar issues (to varying degrees) and has proved very popular. It is instructive to consider why this is the case. We believe that two key factors are the need to make new concepts accessible to those comfortable with existing practice and the need to show a quick pay-back for the effort involved in embracing them.

2. Background

As an application of the principles of data and functional abstraction, skeletal parallel programming has its roots firmly in the Computer Science tradition, and has existed in name for around 15 years. In this time, a number of projects have built real systems. Although technologically impressive, none of these have achieved significant popularity in the wider parallel programming community. While attempting neither an analysis of the complexities of popular tastes, nor a lengthy, comprehensive survey it is instructive to consider the features which have characterised these systems:

- Many have chosen to embed the skeletal concept entirely within a functional programming language (e.g. [9,16–18,28,30,33]). This is entirely natural given the conceptual connection between skeletons and higher-order functions. However, the typical user is challenged with a massive conceptual shift, and a sense of dislocation (however justified) from control of performance;
- others have integrated imperative code within a skeletal framework expressed either in a functional language [11,12,32] as above, or in some new language [29] or library [1]. These have uniformly required the imperative code fragments to be sequential, thereby making skeletons the only means of introducing parallelism.

Meanwhile, research on patterns has begun to consider facets of concurrency and parallelism [7,24,25]. While initially targeting issues of synchronisation and non-determinism more relevant to distributed computing, recent work has moved closer to the concerns of High Performance Computing (HPC) and the connection to skeletons has become increasingly apparent. In particular, systems such as CO_2P_3S [23] and *PASM* [15] have used class hierarchy and inheritance to capture skeletons in object-oriented languages, and open, layered implementations which allow customisation of parallelism by the knowledgeable user.

Noting the increasing stability and portability of direct parallel programming frameworks (and in particular MPI) we believe that the time is now ripe to harness the convergent experiences of the skeletal and pattern-based approaches. Thus, Section 3 is a manifesto for a research programme which aims to take skeletal programming into the parallel mainstream. We argue for a more pragmatic approach than has previously been adopted in order to enhance accessibility. Section 4 describes experiences with *eSkel*, a system which begins to address the issues raised. Section 5 considers *eSkel* in the light of our manifesto while Section 6 discusses related work. This paper is not in itself an introduction to skeletal programming or MPI, with which the reader is assumed to be familiar. The list of references provides many starting points for reading on the former, while the latter can be studied in most undergraduate textbooks on parallel programming and through many easily accessible online tutorials.

3. A pragmatic manifesto

We present four principles which we believe should guide the future design and development of skeletal programming systems. Various previous systems have addressed the principles to different degrees in different combinations. In order to keep our presentation concise we do not elaborate on these and their relationships here, but provide selected references in Section 6.

3.1. Propagate the concept with minimal conceptual disruption

The core principle of skeletal programming is conceptually straightforward. Its simplicity should be a strength. In order to convey this to practitioners we must be careful not to bundle it with other conceptual baggage, no matter how natural this may seem from the perspective of the researcher. Skeletal programming is not functional programming, even though it may be concisely explained and expressed as such. Nor is it necessarily object-oriented programming, although the increasing interest in such technologies for HPC will make such an attractive embedding viable soon. Instead, we should build bridges to the de facto standards of the day, refining or constraining only where strictly necessary. We should respect the conceptual models of these standards, offering skeletons as enhancements rather than as competition. This need not be too difficult. For example, it is arguable that MPI already

embodies simple skeletons in its collective operations. We will exploit this link in our own work.

3.2. Integrate ad-hoc parallelism

Many parallel applications are not obviously expressible as instances of skeletons. Some have phases which require the use of less structured interaction primitives. For example, Cannon's well-known matrix multiplication algorithm [22] invokes an initial step in which matrices are skewed across processes in a manner which is not efficiently expressible in many skeletal systems. Other applications have conceptually layered parallelism, in which skeletal behaviour at one layer controls the invocation of operations involving ad-hoc parallelism within. It is unrealistic to assume that skeletons can provide all the parallelism we need. We must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well-defined way.

3.3. Accommodate diversity

Previous research has seen the emergence of a common core of simple skeletons and a variety of more exotic forms. When described informally, the core operations are straightforward. Precise specification reveals variations in semantics which reflect the ways skeletons are applied in real algorithms. The result is that some algorithms, which intuitively seem to represent an instance of a skeleton, cannot be expressed in certain systems because of constraints imposed by the specification. For example, an algorithm which seems naturally pipelined may have a stage in which several outputs are generated for each input. Another may have stages which generate no output for certain inputs. A pipeline specification which requires each stage to produce one output for each input excludes such algorithms. Similarly, one can imagine applications of task farming in which some tasks are filtered out without producing results. A farm specification which requires each task to produce one result is an unnatural framework for such situations. We must be careful to draw a balance between our desire for abstract simplicity and the pragmatic need for flexibility. This is not a quantifiable trade-off.

3.4. Show the pay-back

A new technology will only gain acceptance if it can be demonstrated that adoption offers some improvement over the status quo. The principles above can be summarised as an attempt to minimise the disruption experienced in a move to skeletal parallelism. We must also be able to show that there are benefits which outweigh the initial overheads and that it is possible to experience these early on the learning curve. Ultimately these must result from direct experience of high quality implementations, but in the first instance we must build a convincing catalogue of case studies. In doing so, we should be careful to understand in advance what it is we aim to demonstrate. For example, at the very least on individual applications, we must be able

to outperform the conventional implementation constructed with "equivalent" programming effort. Perhaps more easily, but equally impressively, we should be able to show that skeletal programs can be ported to new architectures, with little or no amendment to the source, and with *sustained performance*. This can be contrasted with the performance pitfalls inherent in transferring semantically portable but performance vulnerable ad-hoc programs, and echoes familiar arguments in favour of the use of collective communication operations over hand coded equivalents [14]. More ambitiously, we may be able to show that the structural knowledge embedded in skeletons allows optimisation within and across uses which would not be realistically achievable by hand.

Motivated by this manifesto, we have recently begun development of *eSkel* (*e*dinburgh *Ske*leton *l*ibrary). The library and its documentation can be downloaded from the *eSkel* home page [4]. Section 4 presents an overview of the most recently released version, with the help of a simple example program. The interested reader should consult the on-line reference documents for a full specification and further examples. Section 5 discusses *eSkel* in the context of our manifesto.

4. An overview of *eSkel*

4.1. Collective calls, skeletons, processes and activities

eSkel is a library of C function and type definitions which extend the standard C binding to MPI with skeletal operations. Its underlying conceptual model is that of SPMD distributed memory parallelism, inherited from MPI, and its operations must be invoked from within a program which has already initialised an MPI environment.

MPI programmers are 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 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 interaction" 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.

In *eSkel* each skeleton is a collective operation, called by all processes within the group associated with the communicator which is passed as an argument to the call. During the call, the participating processes are grouped and regrouped by the implementation, according to the semantics of the skeleton. Each group of processes created in this way constitutes an *activity* (such as a stage in a pipeline or a worker in a farm). Interactions between activities are implemented implicitly by the skeleton, according to its semantics. For example, in the `Farmlforl` skeleton, processes are allocated to a programmer specified number of *worker* activities. If several processes are allocated to a worker, then it can exploit internal parallelism, using either another skeleton or direct calls to MPI. The skeleton itself handles all aspects of task distribution, including collating the initial set of tasks, distributing these dynamically to workers in order to achieve a good load balance, and collating and storing the results returned. In the current implementation this is achieved by multi-threading one worker process to also act as a traditional *farmer*, but more sophisticated distributed schemes are possible without changing the API or its semantics. The application programmer must simply specify the collection of input tasks and the operations performed by a worker activity to process a task.

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()`. 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.

In its current prototype, *eSkel* supports skeletons for pipelining, task farming and butterfly style divide-and-conquer.

4.2. Handling data

Explicit manipulation of data within activities uses standard C (and MPI for ad-hoc communication). This raises the issue of how our skeletons interface to the C/MPI data model. In fact, there are two related questions:

- how does the code for an activity interface to the skeleton in order to receive and return individual data items?
- how does the programmer specify the collection of data upon which a skeleton is to operate?

These questions are answered by the *eSkel Data Model* (eDM), which defines the concepts of the *eDM atom* and the *eDM collection*.

4.2.1. The eDM atom

MPI requires the programmer to specify communicated data (and receiving buffers) as (pointer, length, type) triples on participating processes, as appropriate to each communication operation. This localised concrete view of data is often in conflict with the programmer's conceptual view of data as consisting of logically coherent items which happen to be physically distributed across the address spaces of the processes. For example, in an image processing application, MPI provides no way of indicating that the segments of an image stored by various processes are part of a logical whole. It is the programmer's responsibility to ensure that the local segments are treated accordingly.

This situation creates a problem for a skeletal system: if the programmer identifies data to be processed in the usual MPI way, then how is the system to know whether the contributions from individual processes should be treated as self-contained items, or as as pieces of a ''distributed-shared'' whole? In an ad-hoc MPI program, the programmer would keep this knowledge implicit and would use MPI calls directly to achieve the required effect. In a skeletal system (which will handle the data interactions itself) the programmer must state the intended interpretation. In *eSkel* we call this property the *spread* of a data item and distinguish between *local* (processwise contributions are self-contained) and *global* (processwise contributions are parts of a distributed whole) cases. The programmer is required to explicitly choose between these interpretations when specifying data. Fig. 1 illustrates the concept. Each process provides a block of data. These can be interpreted as comprising three data items (local spread) or one distributed item (global spread).

Thus, an eDM atom is an MPI triple, augmented with a tag indicating spread. Activities interact with the skeleton, and indirectly each other, in terms of eDM atoms. It is crucial to stress that this is not a new abstract data type. The tags have

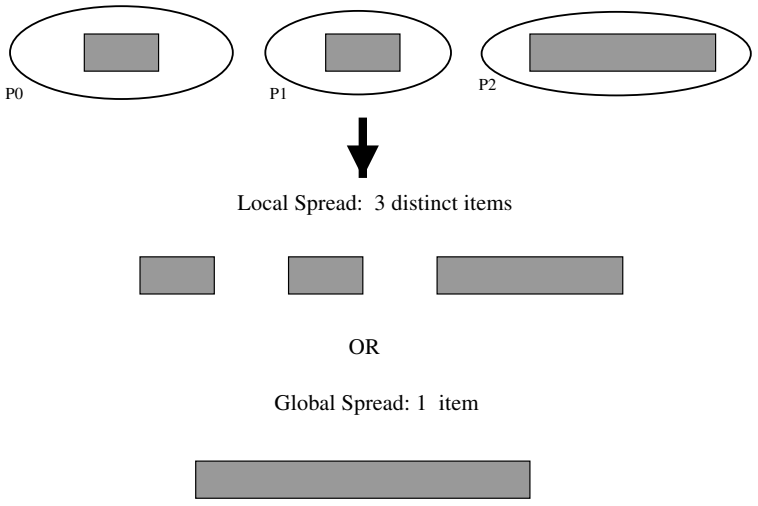

Fig. 1. Local and global spread.

implications for the way data will be passed around a skeleton, but not for its manipulation within activity code. Just as there is no requirement in MPI for an item received as a triple to be manipulated in any particular way, so there are no requirements on how an activity manipulates the atoms it receives (though it is of course aware, through the tag, of the intended interpretation).

4.2.2. The eDM collection

Input data for a typical skeleton call will consist of a number of atoms (for example, the sequence of items to be passed through a pipeline). In *eSkel* this is known as an eDM *collection* and is simply a sequence of eDM atoms, all having the same spread. To distinguish the number of items in a collection from the number of items in a single atom (i.e. the "length" part of the triple) we refer to this as the collection's *multiplicity*. Skeleton calls have arguments detailing their input and output collections. The output collection is a specification of some pre-allocated space into which the results will be stored, following the MPI convention of requiring programmer-allocated buffers for the receipt of communications. Fig. 2 illustrates the concept. Each of the three processes provides (multiplicity) four chunks of data. These can be interpreted as comprising 12 separate data items (local spread) or three distributed items (global spread).

4.3. Implementation

The first prototype of *eSkel* is implemented in C and MPI. The focal run-time data structure is a stack (replicated on each process) of structures capturing information about the nest of active skeleton calls. Each structure stores information on the called skeleton, its actual parameters and the communication context. These are required to support the various structuring and communication activities abstracted by the skeleton. The bulk of the code itself is concerned with skeleton specific data mar-

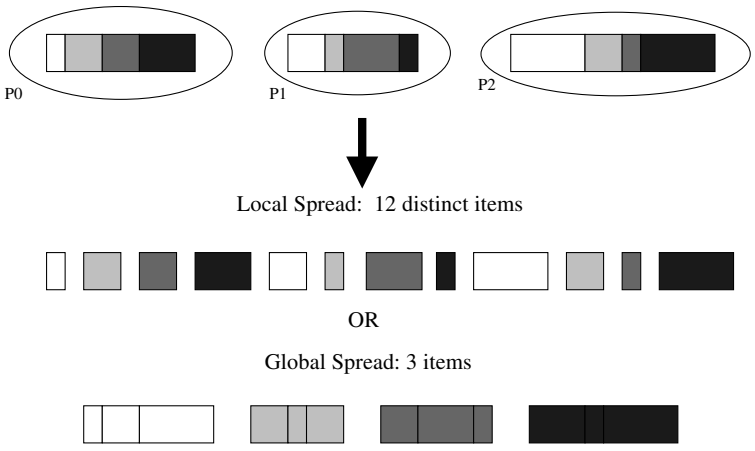

Fig. 2. Multiplicity and spread.

shalling. Most of this involves gathering and scattering the buffer contributions of each participating process upon entry to and exit from the skeleton call and similarly arranging internal communications. All of this must respect the spreads and multiplicities as specified in the skeleton call (and stored in the stack). Extending the repertoire of skeletons requires a thorough understanding of these internal data structures and operations.

4.4. Using eSkel

As an early test of the usability of the library, we took an open source sequential program for drawing the Mandelbrot set from the web [34] and adapted it to run as an *eSkel* `SimpleFarmlforl`. This variant of `Farmlforl` assigns one process to act as solely as farmer, for machines in which thread safe MPI is not available, and constrains each worker to be a single process. This was straightforward, with no amendments required to the core of the code.

The prototype for `SimpleFarmlforl` is as follows

```
void SimpleFarmlforl (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 the first parameter selects the worker function. The next five specify the input data (pointer, length of task, number of tasks, spread of tasks, underlying task element type). The following five parameters similarly specify the output buffer, with the output multiplicity parameter `outmul` passed by reference so that it can be set during the call. The second last parameter gives the length of the output buffer, so that overflow can be detected, and the last parameter provides the communicator (and implicitly, the process group) within which the farm should be constructed.

Fig. 3 presents an extract of the main program. The first item of note occurs on lines 27–28, where MPI types for points and pixels are constructed and registered. This is standard MPI. Lines 30–40 create the array representing the domain for our image. We use a one dimensional array with index arithmetic because *eSkel*, like MPI, currently has no explicit concept of arrays of higher dimension. The whole array is created by process 0, but it would have been equally possible for different processes to create distinct sub-arrays. The assignments to `inmul` on lines 37 and 39 declare the input multiplicity of each process (i.e. 0 for all processes except P_0, which creates a number dependent upon the chosen granularity `CHUNK`). Lines 42–44 call the skeleton. Lines 47–51 use conventional C/MPI to gather the results to process 0 for output.

Fig. 4 presents the code for the task function `mandelcheck`. This perform the Mandelbrot calculation for each of the `CHUNK` points in a task, returning a corresponding set of pixels. The task is presented to the function as a instance of the library defined type `eSkel_atom_t`, a structure containing a `(void *)` pointer

```
#define hxres   256       // horizonal resolution
#define hyres   256       // vertical resolution
#define itermax 100000    // bound on iterations for convergence
#define CHUNK   1         // number of pixels in a task

typedef struct {
  int r;  int g;  int b;
} pixel_t;

typedef struct {
  int x;  int y;
} point_t;

point_t point[hxres*hyres];
pixel_t pixel[hxres*hyres];
pixel_t *results=NULL;

void main (int argc, char *argv[])
{
  int i, j;
  int inmul, outmul;
  MPI_Datatype MPI_POINT, MPI_PIXEL;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  SkelLibInit();
  MPI_Type_contiguous (2, MPI_INT, &MPI_POINT);  MPI_Type_commit (&MPI_POINT);
  MPI_Type_contiguous (3, MPI_INT, &MPI_PIXEL);  MPI_Type_commit (&MPI_PIXEL);
  // set up the points
  if (myrank()==0) {
     for (j=0; j<hyres; j++)  {
       for (i=0; i<hxres; i++)  {
         point[j*hxres+i].x = i+1;
         point[j*hxres+i].y = j+1;
       }
    }
    inmul = hxres*hyres/CHUNK;
  } else {
    inmul = 0;
  }

  SimpleFarm1for1 (mandelcheck, (void *) point, CHUNK, inmul, SPLOCAL, MPI_POINT,
                   (void *) pixel, CHUNK, &outmul, SPLOCAL, MPI_PIXEL,
                   hxres*hyres, mycomm());

  // gather results to p0 for output
  if (myrank() == 0) {
    results = (pixel_t *) malloc (sizeof(pixel_t)*hxres*hyres);
  }
  MPI_Gather (pixel, hxres*hyres/P, MPI_PIXEL, results, hxres*hyres/P,
              MPI_PIXEL, 0, MPI_COMM_WORLD);
  if (myrank() == 0) { // output pixels to a file }
  MPI_Finalize();
}
```

Fig. 3. Main program for the Mandelbrot example.

and an `int`, place-holders for the data pointer and length which are the only properties of an eDM atom left unconstrained by the enclosing skeleton call. We must

```
 1 eSkel_atom_t *mandelcheck (eSkel_atom_t *pointatom)
 2 {
 3   point_t *points;
 4   pixel_t *pixels;
 5   eSkel_atom_t *result;
 6   int i;
 7
 8   if (pointatom) {
 9     points = (point_t *)(pointatom->data);
10     pixels = (pixel_t *) malloc (sizeof (pixel_t)*CHUNK);
11     for (i=0; i<CHUNK; i++) {
12       // around 20 lines of the original code
13       // computing pixels[i] from points[i]
14     }
15     result = (eSkel_atom_t *) malloc (sizeof(eSkel_atom_t));
16     result->data = (void *) pixels;
17     result->len = CHUNK;
18     return result;
19   } else {
20     return NULL;
21   }
22 }
```

Fig. 4. Code for the Mandelbrot task function.

stress that `eSkel_atom_t` is not an abstract data type: its definition is available to the programmer and its contents are manipulated directly in user code. The real work is performed by dummy lines 12–13, with around 20 lines of code (omitted from the figure) taken verbatim from the original. Lines 15–17 construct the atom to be returned.

Other skeletons in the preliminary implementation have similar prototypes and usage. For example, the `Pipeline` skeleton has prototype shown below, in which the first parameter indicates the number of stages, the second is the array of stage functions and the third indicates allocation of participating processes to stages (using integer "colours" in the style of MPI's communicator splitting operation). The remaining parameters are similar to `SimpleFarmlforl`.

```
void Pipeline (int ns, void (*stages[])(),
  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);
```

Giving the programmer the freedom to allocate processes to stages is a useful tool. For example, the textbook description [22] of the well-known Gaussian Elimination algorithm has two phases, both pipelined, but with opposite directions of data flow. Additionally, the processes in the pipelines are required to maintain some internal state between the two phases. This is easy to express in eSkel. Fig. 5 presents an extract of the code. Notice that the third (colour) parameter is used to reverse the order of pipeline flow. `myrank()` is the process' rank within the `P` processes active overall.

```
 1 int main (int argc, char *argv[])
 2 {
 3   // Initialisation code ...
 4
 5   // Scatter data with ordinary MPI ...
 6
 7   for (i=0; i<STAGES; i++) {
 8     stages[i] = (void (*) (void)) reduce;
 9   }
10
11   Pipeline (STAGES, stages, myrank(), spreads, types,
12             NULL, 0, 0, NULL, 0, &outmul, 0, mycomm());
13
14   for (i=0; i<STAGES; i++) {
15     stages[i] = (void (*) (void)) backsub;
16   }
17
18   Pipeline (STAGES,  stages, P-myrank()-1, spreads, types,
19             NULL, 0, 0, NULL, 0, &outmul, 0, mycomm());
20
21   // Gather and display the data with ad-hoc MPI
22 }
```

Fig. 5. Code for the Mandelbrot task function.

`reduce` and `backsub` define the activity required of each stage in the corresponding phases. It is also interesting to note that the both input and output buffers to the whole call are empty! This is a natural reflection of the textbook algorithm, in which data and results are stored in distributed fashion across the participating processes. A more constrained skeleton, requiring conventional input and output buffers or streams, would only complicate or even prohibit expression of the algorithm.

5. *eSkel* in context

eSkel was designed to address the issues raised by our skeletal programming manifesto. We now discuss these point by point.

5.1. Propagate the concept with minimal conceptual disruption

Our first decision was to present the skeletons in the form of a library, in order to avoid the introduction of any new syntax. In tandem with our decision to base our ad-hoc parallelism on MPI, we chose to provide a C binding initially. Other bindings, following those available for MPI, are equally possible. Furthermore, we decided not to introduce any new abstract types for distributed-shared data. Our skeletons add the facility to move data between activities, following the skeleton specification, without the need to explicitly invoke MPI communications. As demonstrated, these decisions make it simple to re-use existing components.

5.2. Integrate ad-hoc parallelism

We have chosen to build our library around MPI because this is the most widely portable of popular contemporary frameworks. Since any program which uses *eSkel* must already be running MPI, we trivially satisfy the requirement for ad-hoc parallel phases independent of skeleton calls. In order to facilitate ad-hoc parallelism within the activities controlled by skeletons, the library semantics define a dynamically evolving hierarchy of MPI communicators within which processes may communicate. The top of the communicator stack is available via a call to the library function `mycomm()`. For example, if the current skeleton is a pipeline with several processes allocated to each stage, then the processes in some particular stage can interact using direct calls to MPI within `mycomm()`, in the secure knowledge that these will be isolated from communications elsewhere in the pipeline, including those generated by the implementation to handle interactions with the preceding and succeeding stages.

5.3. Accommodate diversity

As we have noted, informal agreement over skeleton semantics can mask a range of variations. Should a pipeline stage return exactly one output for each input? Should a farm worker behave similarly? Should a pipeline stage be allowed to generate outputs without any input at all? Should a stage act as a function, instantiated afresh for each input, or as a process instantiated once and maintaining state from item to item?

It would be possible to address such diversity by providing "lowest common denominator" skeletons in each case, perhaps with a number of parameters selecting specific behaviour. Instead, we have chosen to define *families* of skeletons, adhering to a common naming discipline, parameter structure and semantics. For example, in the pipeline family we currently have `Pipeline` and `Pipelinelforl`. In the former, a stage is a process, active for the duration of the skeleton call, and free to produce new outputs entirely arbitrarily. In the latter, a stage is a function, instantiated for each input and required to produce exactly one output at each instantiation. Analogously, in `Farm` the worker is a process which can return results arbitrarily, whereas in `Farmlforl` it is a function, returning one result for each task. This structure also allows families to be expanded without disruption to existing member functions. This would not be possible with a single skeleton, catch-all approach. The conceptual distinction between plain and "1for1" forms necessitates a corresponding variation in the interface between skeleton and activity. In the former case, an activity must explicitly indicate to the skeleton its readiness to receive or produce a new data item. Certain parameters of the transfer are determined by the actual parameters to the skeleton. For example, in a `Farm`, the underlying type, length and spread of the tasks and results are fixed at this point. The pointer to each data object to be returned is provided dynamically by the activity. This exchange is handled by the *eSkel* function `Give`, called by the activity and parameterised by the pointer and the length. The analogous work required to acquire a new item from the skeleton is handled by the function `Take`. Prototypes for `Give` and `Take` are as follows

```
void Give (void *out, int length);
void *Take (int *length);
```

where in each case the pointer and length capture the only aspects of the transferred data which are not already constrained by the enclosing skeleton. `Give` and `Take` are generic functions which may be called with a common interface from many skeleton functions. Their semantics are determined by the currently active skeleton (get an item from the previous stage, return a value to the farmer, and so on). In contrast, "1for1" skeletons constrain activities to produce one output for each input. The definition of an activity can therefore be encapsulated as a function in which explicit calls to `Give` and `Take` are forbidden.

As an example, in the Gaussian elimination program sketched earlier, the activity of a stage during the reduction phase has a complex behaviour which cannot be expressed as a simple `lforl` pipeline. *eSkel* allows this behaviour to be expressed neatly with `Give` and `Take`, as outlined in Fig. 6 (from which we omit the computational detail to emphasise the structure).

5.4. Show the pay-back

The development of *eSkel* is at a preliminary stage. The early results are promising, but it is clear that much remains to be done to build the convincing suite of performance portable applications demanded by this point of the manifesto. We now report briefly on the results of running the Mandelbrot program on a 24-processor Sunfire 6800 UltraSPARC-III based SMP, hosted by the Edinburgh Parallel Computing Centre. For convenience, an image size and convergence criterion which produced a sequential run time of a few tens of seconds were found by trial and error. We then experimented by varying the number of processors and the number of pixel computations allocated to a single farmed task. These involved changing only a single constant in each case. Timings shown are averaged over a small number of runs: only very small variation in timings between identically parameterised runs was observed (around 1% of run time), including all I/O activity. Fig. 7 shows the effect of varying the number of processors P for a fixed image size of 256×256 pixels and a

```
 1 void reduce (void)
 2 {
 3   float *rowin;
 4
 5   while (rowin = Take(&ncols)) {
 6     // ....
 7     if (myrealrank<N-1) Give(rowin, ncols);
 8     // ....
 9   }
10   // ....
11   if (myrealrank<N-1) Give(mya, N+1);
12 }
```

Fig. 6. Outline of the Gaussian reduction stage.

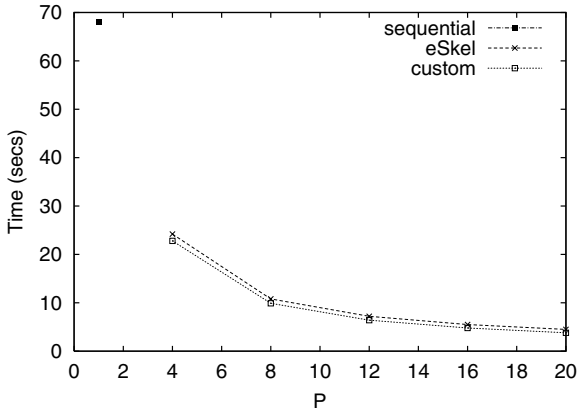

Fig. 7. Effect of varying P.

maximum iteration count (in the standard Mandelbrot convergence check) of 100,000. Each task involved performing the Mandelbrot calculation for 32 pixels. The time for one processor is for the original sequential C program with no MPI or *eSkel* setup overheads. Since in a `SimpleFarm`, one process(or) performs the role of farmer we should not expect more than $P-1$ fold speed-up for P processors. Fig. 7 demonstrates good speed-up, almost matching that of a custom coded farm which incorporated some small problem specific optimisations (essentially because the *eSkel* version has to create, populate and manage a data structure for the tasks, which the custom version can generate directly and dynamically). Our initial farm skeleton implementation is quite simple and offers considerable scope for generic optimisation and we regard these initial results as promising. Fig. 8 shows the effect of varying the number of pixels assigned as a single task (i.e. adjusting the granularity at which

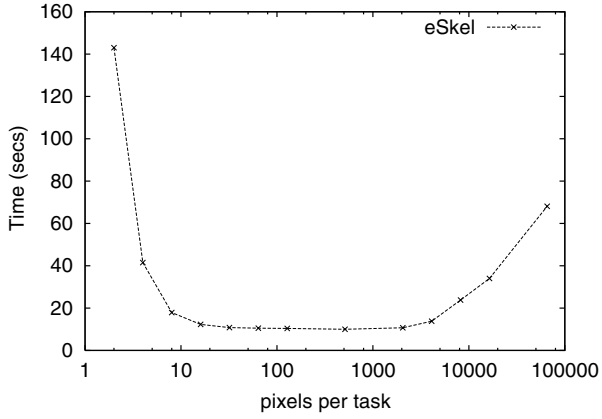

Fig. 8. Effect of varying the granularity.

farming is implemented, just as a programmer would do in the activity code) for the same image size and iteration count and with P fixed at 8. The expected pattern of behaviour is observed, with significant improvements as the granularity is increased (in this case to around 32 pixels per task), with this performance maintained until the granularity becomes so high that there is a shortage of tasks. The final point on the graph corresponds to a granularity of 65,536, in other words only one task, and a run time very close to that of the sequential program.

6. Acknowledgements and related work

The conceptual theme of this paper has emerged over a number of years and has been influenced by many colleagues. Most recently I must thank Herbert Kuchen whose concurrent work on his own library [19,20] and collaboration [21] have been invaluable. A lively debate on standardisation of skeletons emerging from the workshop on High Level Parallel Programming held in Orleans in 2001 was also very helpful, particularly the contributions of Gaetan Hains, Sergei Gorlatch, Chris Lengauer, Frederic Loulergue, Susanna Pelagatti,and Herbert Kuchen again. I similarly thank Kevin Hammond, Hans-Wolfgang Loidl, Greg Michaelson and Phil Trinder for their contributions to the debate which followed HLPP'01. Discussions during an EC funded visit to Edinburgh by Andrea Zavanella [5] were also beneficial.

A number of other works are due particular attention. Concern for accessibility has been evident in the library systems of Danelutto [6] and Danelutto and Stigliani [8] with their echoes of the pragmatism of the earlier P3L project [29]. Similarly, the work of Darlington's group on the SCL structured co-ordination language [10–12] sought to integrate conventional imperative code fragments (in Fortran) within a structured parallel environment. Kuchen and Botorog's earlier work on Skil [1] was similarly motivated, while the OTOSP model [13] uses OMP style skeletal *pragmas* to expand an underlying ad-hoc threaded model.

Other systems have sought to exploit the conceptual link between skeletons and design patterns [31] to provide pattern based parallel code generators. The CO_2P_3S system [23] generates multi-threaded Java, and encourages the programmer to specialise the resulting program to improve performance. This endeavour is supported by the provision of a three layer view of an application, with successive layers opening up more internal structure for refinement. In a similar vein, the *PASM* system [15] take an open approach, building a library (underpinned by MPI) which the programmer is explicitly encouraged to amend and extend. Papers on both CO_2P_3S and *PASM* argue that extensibility, sensitivity to the diverse requirements of superficially similar patterns and the ability to make application specific performance enhancements are key attributes in the quest for mainstream acceptability. The systems exploit object-oriented technology to capture the refinement relationships between patterns in general and local variations of specific patterns. The link to patterns is also apparent in [24] which focuses on the design process. The admission of ad-hoc parallelism to a closed skeletal framework was considered in [26,27]. Finally,

Gorlatch [14] makes a strong pragmatic case for the use of structured, collective operations.

7. Future directions

We intend to build upon the current *eSkel* prototype in many ways. At the conceptual level we will extend the set of available skeletons to encompass the full range of structures addressed by previous systems, within one coherent framework. At the language level, we will consider the software engineering benefits which might emerge from an object-oriented casting of the library, following the lead of [15,20,23]. Similarly, it is interesting to consider how our principles (but not this specific realisation) might apply to the design of libraries based around threaded parallelism in the ad-hoc layer (following [31]), or more exotically, to the extremes of Grid or System-on-Chip parallelism.

References

[1] G. Botorog, H. Kuchen, Efficient high-level parallel programming, Theoretical Computer Science 196 (1998) 71–107.
[2] P. Brinch-Hansen, Studies in Computational Science: Parallel Programming Paradigms, Prentice-Hall, 1995.
[3] M. Cole, Algorithmic skeletons, in: K. Hammond, G. Michaelson (Eds.), Research Directions in Parallel Functional Programming, Springer, 1999, pp. 289–303.
[4] M. Cole, eSkel library home page. Available from <http://www.dcs.ed.ac.uk/home/mic/eSkel>.
[5] M. Cole, A. Zavanella, Coordinating heterogeneous parallel systems with skeletons and activity graphs, Journal of Systems Integration 10 (2) (2001) 127–143.
[6] M. Danelutto, Efficient support for skeletons on workstation clusters, Parallel Processing Letters 11 (1) (2001) 41–56.
[7] M. Danelutto, On skeletons and design patterns, in: Parallel Computing, Fundamentals and Applications, Proceedings of the International Conference ParCo99, Imperial College Press, 2000, pp. 460–467.
[8] M. Danelutto, M. Stigliani, SKElib: parallel programming with skeletons in C, in: Proceedings of Euro-Par 2000, LNCS 1900, Springer, 2000, pp. 1175–1184.
[9] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, R. While, Parallel programming using skeleton functions, in: Proceedings of PARLE'93, LNCS 694, Springer, 1993, pp. 146–160.
[10] J. Darlington, Y. Guo, H.W. To, Structured parallel programming: theory meets practice, in: R. Milner, I. Wand (Eds.), Research Directions in Computer Science, Cambridge University Press, 1996.
[11] J. Darlington, Y. Guo, H.W. To, J. Yang, Parallel skeletons for structured composition, in: Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press, 1995, pp. 19–28.
[12] J. Darlington, Y. Guo, H.W. To, J. Yang, Functional skeletons for parallel coordination, in: Proceedings of Euro-Par'95, LNCS 966, Springer, 1995, pp. 55–69.
[13] A. Dorta, J. Gonzalez, C. Rodriguez, F. Sande, Structured parallel programming, in: Proceedings of the European Workshop in OpenMP 2002, EWOMP 02, Roma, Italy. Available from <http://www.caspur.it/ewomp02/prog.html>.
[14] S. Gorlatch, Send-Recv considered harmful? myths and truths about parallel programming, in: Proceedings of PaCT2001, LNCS 2127, Springer, 2001, pp. 243–257.

[15] D. Goswami, A. Singh, B. Preiss, From design patterns to parallel architecture skeletons, Journal of Parallel and Distributed Computing 62 (4) (2002) 669–695.
[16] M. Hamdan, A combinational framework for parallel programming using algorithmic skeletons, Ph.D. Thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 2000.
[17] C. Herrmann, C. Lengauer, HDC: a higher-order language for divide-and-conquer, Parallel Processing Letters 10 (2–3) (2000) 239–250.
[18] U. Klusik, R. Loogen, S. Priebe, F. Rubio, Implementation skeletons in Eden: low-effort parallel programming, in: Proceedings of IFL'00, LNCS 2011, Springer, 2001, pp. 71–88.
[19] H. Kuchen, The skeleton library web pages. Available from <http://danae.uni-muenster.de/lehre/kuchen/Skeletons>.
[20] H. Kuchen, A skeleton library, Report 6/02-I, Angewandte mathematik und informatik, University of Münster, 2002.
[21] H. Kuchen, M. Cole, The integration of task and data parallel skeletons, in: Proceedings of Constructive Methods for Parallel Programming'02, Dagstuhl, 2002.
[22] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin Cummings, 1994.
[23] S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, Generating parallel program frameworks from parallel design patterns, in: Proceedings of Euro-Par 2000, LNCS 1900, Springer, 2000, pp. 95–104.
[24] B. Massingill, T. Mattson, B. Sanders, A pattern language for parallel application programs, in: Proceedings of Euro-Par 2000, LNCS 1900, Springer, 2000, pp. 678–681.
[25] B. Massingill, Experiments with program parallelization using archetypes and stepwise refinement, Parallel Processing Letters 9 (4) (1999) 487–498.
[26] M. Marr, M. Cole, Hierarchical skeletons and ad-hoc parallelism, Advances in Parallel Computing 11 (1996) 673–676.
[27] M. Marr, Descriptive simplicity in parallel computing, Ph.D. Thesis, University of Edinburgh, 1998.
[28] G. Michaelson, N. Scaife, P. Bristow, P. King, Nested algorithmic skeletons from higher order functions, Parallel Algorithms and Applications 16 (2001) 181–206.
[29] S. Pelagatti, Structured Development of Parallel Programs, Taylor & Francis, 1997.
[30] J. Schwarz, F. Rabhi, A skeleton-based implementation of iterative transformation algorithms using functional languages, in: M. Kara et al. (Eds.), Abstract Machine Models for Parallel and Distributed Computing, IOS, 1996, pp. 119–133.
[31] F. Rabhi, S. Gorlatch (Eds.), Patterns and Skeletons for Parallel and Distributed Computing, Springer, 2002.
[32] J. Serot, D. Ginhac, J.P. Derutin, SKiPPER: a skeleton-based parallel programming environment for real-time image processing applications, in: Proceedings of PaCT-99, LNCS 1662, Springer, 1999, pp. 296–305.
[33] P. Trinder, K. Hammond, H.-W. Loidl, S. PeytonJones, Algorithm + strategy = parallelism, Journal of Functional Programming 8 (1) (1998) 23–60.
[34] E. Weeks, Mandel.c source code for the simplest C & BASIC programs to do the Mandelbrot set. Available from <http://glinda.lrsm.upenn.edu/~weeks/software/mand.html>.