Bringing Skeletons out of the Closet

Murray Cole

Institute for Computing Systems Architecture

Division of Informatics

University of Edinburgh.

Abstract

We note the absence of skeletal programming from mainstream parallel computing practice and suggest a number of design principles for skeletal systems which may help to redress the situation. We sketch the **eSkel** library which represents a preliminary attempt to address the issues raised.

1 Introduction

The skeletal approach to parallel programming is well documented in the research literature [RaGo02]. 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. Skeletal programming proposes that these patterns be abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations 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 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 parallel applications programming practice. 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. We will expand on what these may mean in more detail in the next section.

The next section 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. We believe that it is appropriate to set aside some of the more technologically advanced aspects of earlier work where this enhances accessibility, hoping that some of these may be subsequently re-introduced. The paper is completed by a discussion of a system which begins to address the issues raised.

2 A Pragmatic Manifesto

We now introduce four principles which we believe should guide the 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 references and a brief analysis in the final section.

1. Propogate the concept with minimal conceptual disruption

The core principle of skeletal programming is neither particularly deep nor particularly complex. Its simplicity should be a strength. In order to convey this simplicity 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. Skeletal programming is not object oriented programming, even though this may be a similarly attractive vehicle[1]. Instead, we should build bridges to the 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.

2. Integrate ad-hoc parallelism

Many parallel applications are not obviously expressible as instances of skeletons, whether existing or imagined. Some have phases which require the use of less structured interaction primitives. Some have conceptually layered parallelism, in which skeletal behaviour at one layer controls the invocation of operations involving such *ad-hoc* parallelism within. It is clearly 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.

[1] Perhaps the increasing interest in OO technologies for HPC will make such an embedding viable soon.

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 some 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. We must be careful to draw a balance between our desire for abstract simplicity and the pragmatic need for flexibility. This is not a well defined trade-off.

4. Show the pay-back

A new technology can only gain acceptance if it can be demonstrated that adoption offers some improvement to the status quo. The principles above can be summarised as an attempt to minimise the disruption experienced in a switch 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 quite 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. This echoes familiar arguments in favour of the use of collective communication operations over hand coded equivalents [Gor01] 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.

In the remainder of the paper we sketch the first prototype of **eSkel**, a library which adds skeletal programming features to the C/MPI parallel programming framework. We explain how eSkel addresses our first three principles. The final challenge will be the subject of considerable future work, undertaken concurrently with the task of expanding and refining the library itself.

3 An Overview of the eSkel Library

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 [Cole02]. We now present an overview of the current version, focusing on its responses to our design principles. The next section will expand on this with the help of a simple example program. The interested reader should consult the reference documents for a full specification and further examples.

eSkel is a library of C functions 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. It is most readily understood as an extension to MPI's set of **collective** operations: both MPI's `MPI_Reduce` and eSkel's `Farm` are collective functions which co-ordinate the execution of instances of a given operation (which we call an *activity*) and associated communications, within the context of a given MPI communicator and process group.

1. Propogate 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, in other words, data objects which exist conceptually as integrated items, but semantically as isolated chunks on distinct processes (think of "the image" in a parallel image processing operation). Concrete access to data is entirely conventional and already familiar to the C/MPI programmer. Our skeletons add the facility to move such data around the active processes, following the skeleton specification, without the need to explicitly invoke MPI communications. 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 within a skeleton (e.g. a farm worker, a pipeline stage) 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). Building upon MPI's approach of specifying communicated data objects as (pointer, length, type) triples, this defines the concepts of the *eDM atom* and the *eDM collection*. An eDM atom is effectively an MPI triple, augmented with a tag indicating whether the data contributed by each participating process is an object in its own right, or a component of an object formed by the concatenation of the data provided by all processes in the group. This allows the programmer to distinguish between "distributed-shared" data, as described above, and collections of self-contained items. We call this property the *spread* of an atom and distinguish between *local* and *global* (to the group) cases. 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 implications for the way data will be passed around a skeleton, but not for its manipulation within user 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.

The second question is addressed by the eDM collection. This 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.

2. Integrate ad-hoc parallelism

We have chosen to build our library around MPI because this is the most widely portable and best defined 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. A process which is active in an n level call tree of skeletons will be associated with a corresponding stack of $n+1$ communicators (one for each call and one for the original communicator, `MPI_COMM_WORLD`). For any process, the top of the communicator stack is always 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. The semantics of each skeleton explain how new communicators are constructed and destroyed during a call.

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 have `Pipeline` and `Pipeline1for1`. 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 `Farm1for1` 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" skeleton 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 specific 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 (which must match that specified in the corresponding skeleton call). The analogous work required to acquire a new item from the skeleton is handled by the function `Take`. `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). A further generic function, `Exchange`, is defined for skeletons in which there is a natural concept of a partner process.

In contrast, "1for1" skeletons constrain activities to produce one output for each input. The definition of an activity is thus encapsulated as a function in which explicit calls to `Give` and `Take` are forbidden. This raises the issue of how to type the skeleton parameter corresponding to the activity. This is conceptually a function between arbitrary eDM atom types. To handle this genericity in C, we have introduced an explicitly named type `eSkel_atom_t`, a structure containing a `(void *)` pointer 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 stress that `eSkel_atom_t` is not an abstract data type: its definition is available to the programmer and its contents must be manipulated directly in user code.

4. Show the pay-back

The development of eSkel is at a very preliminary stage and the task of building a convincing set of examples on top of a robust implementation is the subject of ongoing work. For now, we report briefly on the results of a simple experiment conducted during testing of the current prototype.

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 [Weeks] and adapted it to run as an eSkel `SimpleFarm1for1` (this variant of farm assigns one process to act as solely as farmer, for machines in which thread safe MPI is not available, and further constrains each worker to be a single process). This was pleasingly straightforward, with no amendments required to the core of the code. The resulting program was run 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

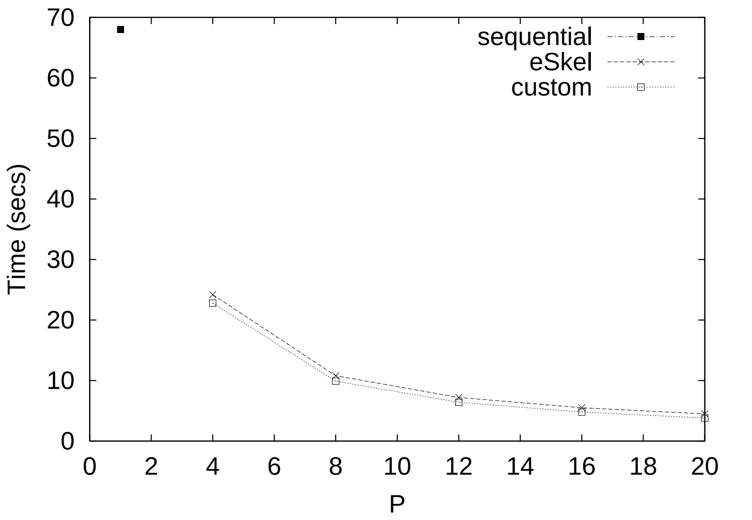

Figure 1: Effect of varying P

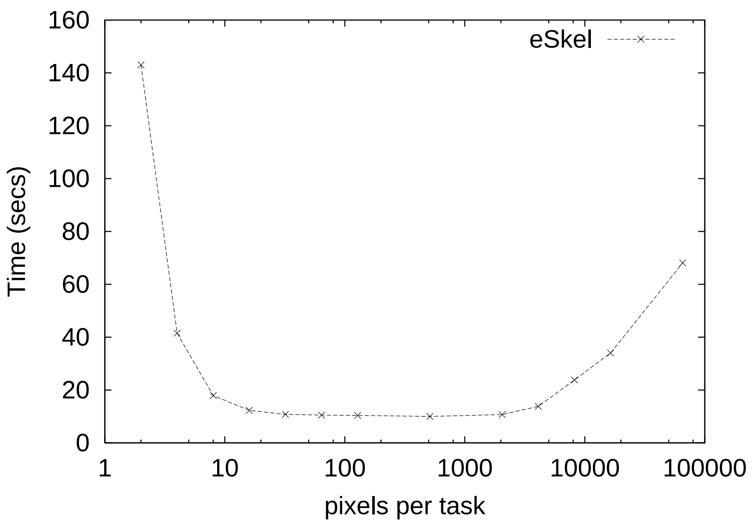

Figure 2: Effect of varying the granularity

was observed (around 1% of run time), including all I/O activity. Figure 1 shows the effect of varying the number of processors P for a fixed image size of 256×256 pixels and a maximum iteration count (in the standard Mandelbrot convergence check) of 100000. 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. Since in a `Simple Farm`, one process(or) performs the role of farmer we should not expect more than $P - 1$ fold speed-up for P processors. We achieve good speed-up, almost matching that of a custom coded farm which incorporated some problem specific optimisation. Our initial farm skeleton implementation is quite simple and offers considerable scope for generic optimisation and we regard these initial results as promising. Figure 2 shows the effect of varying the number of pixels assigned as a single task (i.e. adjusting the granularity at which farming is implemented) 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), and 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 65536, in other words only one task, and a run time pleasingly similar to that of the sequential program.

4 An Example using eSkel

In order to give a flavour of the eSkel library, we now outline the Mandelbrot farm example. Figure 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 entirely conventional 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, has no 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 number of tasks being created by each process (i.e. none for all processes except P_0, which creates a number dependent upon the chosen granularity `CHUNK`). Lines 42-44 call the skeleton. 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 returned from 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. Lines 47-51 use conventional C/MPI to gather the results to process 0 for output. Figure 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 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.

```
#define P       8
#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);
  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();
}
```

Figure 3: Main program for the Mandelbrot example

```
eSkel_atom_t *mandelcheck (eSkel_atom_t *pointatom)
{
  point_t *points;
  pixel_t *pixles;
  eSkel_atom_t *result;
  int i;

  if (pointatom) {
    points = (point_t *)(pointatom->data);
    pixels = (pixel_t *) malloc (sizeof (pixel_t)*CHUNK);
    for (i=0; i<CHUNK; i++) {
      // around 20 lines of the original code
      // computing pixels[i] from points[i]
    }
    result = (eSkel_atom_t *) malloc (sizeof(eSkel_atom_t));
    result->data = (void *) pixels;
    result->len = CHUNK;
    return result;
  } else {
    return NULL;
  }
}

```

Figure 4: Code for the Mandelbrot task function

5 Acknowledgements and Related Work

The conceptual theme of this paper has emerged over a number of years and has been influenced by the work of, and discussions with, many colleagues. Most recently I must thank Herbert Kuchen whose concurrent work on his own library [Ku02a, Ku02b] and collaboration [KuCo02] 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 and hope that my colleagues from the functional branch of skeletal programming will not be too horrified by the content! Discussions during am EC funded visit to Edinburgh by Andrea Zavanella [CoZa01] were also beneficial.

A number of other works are due particular attention. Concern for accessibility has been evident in the library systems of Danelutto [Da01, DaSt00] with their echoes of the pragmatism of the earlier P3L project [Pe97]. Similarly, the work of Darlington's group on the SCL structured co-ordination language [DaGu95a, DaGu95b, DaGu96] sought to integrate conventional imperative code fragments (in Fortran) within a structured parallel environment. Kuchen and Botorog's earlier work on Skil [BoKu98] was similarly motivated.

Other systems have sought to exploit the conceptual link between skeletons and design patterns [RaGo02] to provide pattern based parallel code generators. The CO_2P_3S system [MacSz00] 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 [GoSi02] 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 strongly 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 themselves make strong use of an object oriented methodology to capture the refinement relationships between patterns in general and local variations of specific patterns. The link to patterns is also strong in [MaMa00] which focuses on the design process. The admission of ad-hoc parallelism to a closed skeletal framework was considered in [MaCo96, Marr98]. Finally, Gorlatch [Gor01] makes a strong pragmatic case for the use of structured, collective operations.

References

[BoKu98] G.Botorog, H. Kuchen: Efficient High-Level Parallel Programming, Theoretical Computer Science 196, pages 71-107, 1998.

[Cole02] M. Cole, eSkel library home page, http://www.dcs.ed.ac.uk/home/mic/eSkel.

[CoZa01] M. Cole, A. Zavanella: Coordinating Heterogeneous Parallel Systems with Skeletons and Activity Graphs, Journal of Systems Integration 10(2), pages 127-143, 2001.

[Da01] M. Danelutto: Efficient support for skeletons on workstation clusters, Parallel Processing Letters 11(1), pages 41-56, 2001.

[DaSt00] M. Danelutto, M. Stigliani: SKElib: parallel programming with skeletons in C, in Proceedings of Euro-Par 2000, LNCS 1900, pages 1175-1184, 2000.

[DaGu96] J. Darlington, Y. Guo, H. W. To: Structured Parallel Programming: Theory meets Practice, in Robin Milner and Ian Wand, editors, Research Directions in Computer Science, Cambridge University Press, 1996.

[DaGu95a] J. Darlington, Y. Guo, H. W. To, and J. Yang: Parallel skeletons for structured composition, Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 19-28, ACM Press, 1995.

[DaGu95b] J. Darlington, Y. Guo, H. W. To, and J. Yang: Functional skeletons for parallel coordination, in Proceedings of Euro-Par '95, LNCS 966, pages 55-69, 1995.

[Gor01] S. Gorlatch: Send-Recv Considered Harmful? Myths and Truths about Parallel Programming in Proceedings of PaCT 2001, LNCS 2127, pages 243-257, 2001.

[GoSi02] D. Goswami, A. Singh, and B. Preiss: From Design Patterns to Parallel Architecture Skeletons, Journal of Parallel and Distributed Computing, 62(4), pages 669-695, 2002.

[Ku02a] H. Kuchen: The Skeleton Library Web Pages,
http://danae.uni-muenster.de/lehre/kuchen/Skeletons/

[Ku02b] H. Kuchen: A Skeleton Library, Report 6/02-I, Angewandte Mathematik und Informatik, University of Münster.

[KuCo02] H.Kuchen, M.Cole: The Integration of Task and Data Parallel Skeletons, to appear in the Proceedings of CMPP '02, Dagstuhl, 2002.

[MacSz00] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling: Generating Parallel Program Frameworks from Parallel Design Patterns", in Proceedings of Euro-Par 2000, LNCS 1900, pages 95-104, 2000.

[MaMa00] B. Massingill, T. Mattson, and B. Sanders: A Pattern Language for Parallel Application Programs, in Proceedings of Euro-Par 2000, LNCS 1900, pages 678-681, 2000.

[Marr96] B. Massingill: Experiments with Program Parallelization Using Archetypes and Stepwise Refinement, Parallel Processing Letters, 9(4), pages 487-498, 1999.

[MaCo96] M. Marr and M. Cole: Hierarchical Skeletons and ad-hoc Parallelism, Advances in Parallel Computing 11, pages 673-676, 1996.

[Marr98] M. Marr: Descriptive Simplicity in Parallel Computing, PhD thesis, University of Edinburgh, 1998.

[Pe97] S.Pelagatti: Structured Development of Parallel Programs, Taylor & Francis, 1997.

[RaGo02] F.A. Rabhi and S. Gorlatch (eds), Patterns and Skeletons for Parallel and Distributed Computing Springer Verlag, 2002.

[Weeks] E. Weeks: Mandel.c Source code for the simplest C & BASIC programs to do the Mandelbrot set, http://glinda.lrsm.upenn.edu/~weeks/software/mand.html