

# Towards the Classification of Algorithmic Skeletons

Duncan K. G. Campbell  
Department of Computer Science,  
University of York

December 3, 1996

## Abstract

Algorithmic skeletons are seen as being high-level, parallel programming language constructs encapsulating the expression of parallelism, communication, synchronisation, embedding, and costing. This report examines the classification of algorithmic skeletons, proposing one classification, and examining others which have been devised. Various algorithmic skeletons are examined, and these are categorised to form a core of algorithmic skeletons suitable for a general classification which is based on practical experience in the use of such skeletons. This categorisation is compared with others which have been proposed. Similarly, other skeleton-like approaches are briefly examined.

## 1 Introduction

### 1.1 Algorithmic Skeletons

Algorithmic skeletons are envisaged as high-level, parallel programming language constructs encapsulating the expression of parallelism, communication, synchronisation and embedding, and having an associated cost complexity. Skeletons are to parallel threads as sequential looping constructs are to `goto` and `label` [BdRP93] providing structured expression of certain algorithmic forms. So algorithmic skeletons can help reduce parallel programming errors as part of a “concurrency toolbox” with which programmers can construct the abstraction required to solve their problems.

Indeed, the requirement for algorithmic skeletons has been long known. In 1978 Backus [Bac78] called for programming languages to be designed from a fixed set of high-level constructions capturing common computation patterns.

These constructs are necessary because parallel programming is more complex than serial programming due to the greater degrees of freedom involved. Serial programming involves only a single thread of computation at any one time, while parallel programming involves multiple threads of computation which also need to communicate and synchronise with each other. This complexity is increased with massively parallel processing, which increases the number of threads executing concurrently and needing to communicate and synchronise with each other.

Furthermore, algorithmic skeletons correspond to the *programming* level in terms of McColl’s classification of models of parallel computation [McC93], as they provide a notation for the precise, high-level description of correct and efficient methods for the solutions of computational problems.

### 1.2 Overview

A survey of several example skeletons is presented in Section 2. This is then summarised in Section 3, where the basis for a general classification of algorithmic skeletons is presented

from the conclusions of the survey. Other candidate classifications are presented in Section 4, along with an overview of other skeleton-like approaches in Section 5. The various classifications described are compared with the author’s classification in Section 6. Finally, a summary is presented in Section 7.

## 2 Sample Skeletons

Various authors have presented lists of algorithmic skeletons, and several are examined by the author in [Cam94], summaries of which are presented here. These are from Cole, Darlington *et al.*, Nelson and Snyder, Gehringer *et al.*, Quinn, Goodeve, Rabhi, the *tropes* scheme, and for the  $P^3L$  language.

The reader should note that the skeleton names are written as the original authors have written them, rather than referring to them by a standard notation. For example, Darlington *et al.* [DFH<sup>+</sup>93] refer to their pipeline skeleton as “PIPE”, so it is written as “PIPE” here.

### 2.1 Cole’s Skeletons

Cole [Col89] first coined the term “Algorithmic Skeletons”, and gives the following example skeletons:

**Fixed Degree Divide and Conquer** This is a restriction of the ubiquitous *Divide and Conquer* method, requiring that the degree of all non-leaf nodes in the process tree is a constant known before execution. (See Figure 1 for an illustration.) This restriction provides a greater control of distribution at the expense of a degree of flexibility at the programmer’s level. Suitable problems include discrete Fourier transforms, approximate integration, matrix multiplication, etc.

**Iterative Combination** A problem to be solved by *Iterative Combination* is described by a set of homogeneous objects (with details of any relevant internal structure) and of any relationship between them. Given a rule for combining two objects, and a measure of the value of the combination, the skeleton iterates through a loop in which each object is combined (possibly in parallel) with the most suitable remaining other object, if such exists, until either all objects have been combined into one, or no further acceptable combinations exist. This method is used when it is appropriate to solve problems by progressively imposing structure onto an initially uncoordinated collection of objects. Suitable problems include minimum spanning tree and connected components.

**Cluster** This skeleton is designed from the perspective of implementation on rectangular grids of processors. Suitable problems have data-sets of instances which can be described as a collection of homogeneous objects whose individual descriptions may include information which relates them to each other. These problems are solved by recursively clustering (possibly in parallel) the objects into independent sub-clusters corresponding to every maximal sub-set of objects which are connected directly or transitively by a specific notion of “closeness”. This continues as often as possible (or suitable), with members of clusters being considered together with all other members of their parent cluster and operated upon in some way with respect to each of these. The clustering process imposes a hierarchy of clusters onto the set of objects, with the original complete set at the root, and the measure of “closeness” can be parameterised by the level in the hierarchy. When recombining clusters, all pairs of objects will similarly be considered and manipulated appropriately.

**Task Queue** This skeleton is a generalisation of the *Farm* skeleton illustrated in Figure 2. Problems have both instance and solutions represented in terms of a large data structure, and proceed by repeated concurrent execution of many instances of a task ma-

nipulating some part of the data structure with respect to certain others. Some task instances may generate details of further task instances which are added back to the task queue. Suitable applications are one-to-all shortest paths and LU matrix decomposition

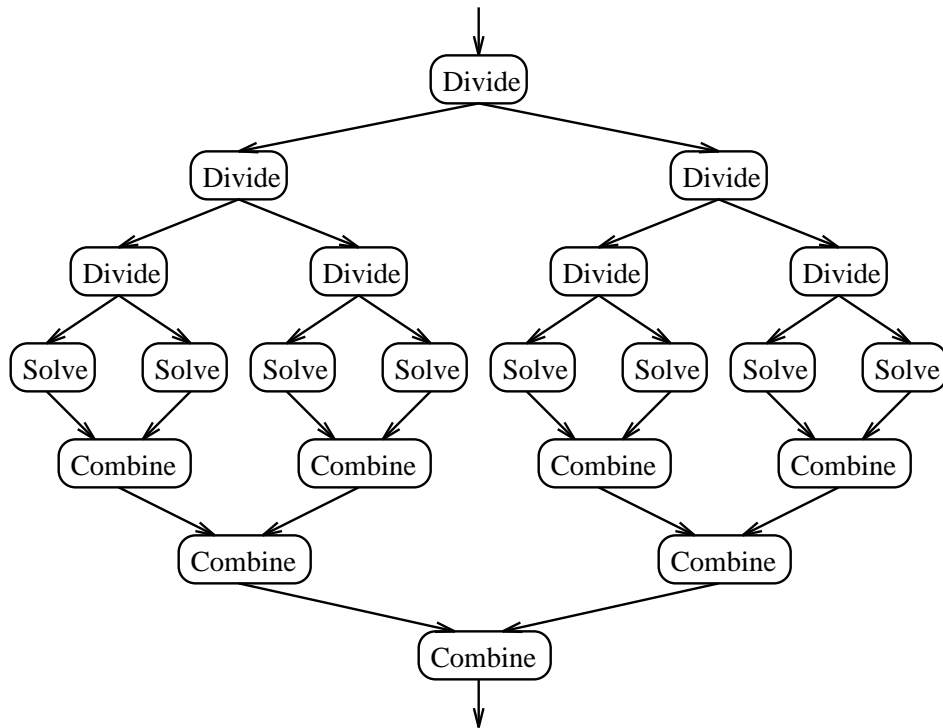


Figure 1: 3-level Binary Divide and Conquer Skeleton

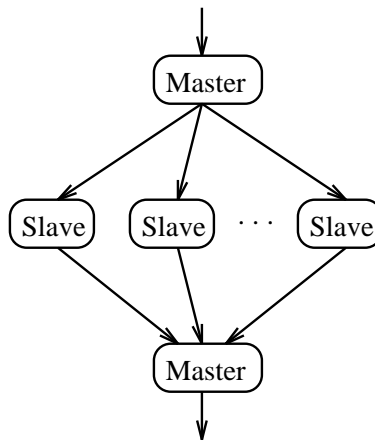


Figure 2: Farm Skeleton

## 2.2 Darlington *et al.*'s Skeletons

Darlington *et al.* [DFH<sup>+</sup>93] have put forward the following list of skeletons:

**PIPE** This captures simple linear process-parallelism. (See Figure 3 for an illustration of a *Pipeline* of  $n$  processes.) It models simple pipelining of processes, where a list of (possibly different) functions are composed together so that elements can be streamed through them.

**FARM** This captures the simplest form of data-parallelism, where a function is applied to each of a list of independent jobs (potentially in parallel) and the results are then combined by the controlling process (see for example Figure 2).

**DC** This models the ubiquitous *Divide and Conquer* method as described above (see Figure 1).

**RaMP** This is the *Reduce and Map over Pairs* skeleton, where each object in the system can potentially interact with any other object. Each individual interaction is calculated and the results are combined to produce a result for each object. This skeleton is typically used for initial specification and implementation by transformation to an alternative form such as by farming out the calculation for each object.

**DMPA** In the *Dynamic Message Passing Architecture* skeleton any process can interact directly with any other process via message-passing, the actual connections being determined at run-time. Each process has an initial state which records local values; messages from other processes may modify the process's state and generate new messages to other processes. This skeleton describes most dynamic algorithms where the interactions between processes are determined using run-time data.

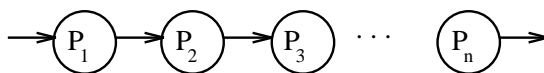


Figure 3: Pipeline Skeleton

## 2.3 $P^3L$

Danelutto *et al.* [DMO<sup>+</sup>91] use the following list of skeletons in the  $P^3L$  language:

**pipe** This is the *Pipeline* method as described above (see Figure 3).

**loop** This models iterative or recursive computations about a sequential or parallel process.

**farm** Various forms of process *Farms* are modelled here. These include cases where identical slave functions process the input data, or where all the input data are processed by each of the (different) slave functions, also where a portion of the input data are processed by the slave functions conditioned by some guards. (See for example Figure 2.)

**geometric** Vector processing and data-parallel computations are modelled here, with the process topology being specified along with the particular mapping of the data partitions and the function which executes on its partition of the data. (See for example Figure 4.)

**tree** This structure can be either a combining or a distributing *tree*, with the programmer specifying the leaf and non-leaf processes, and the number of levels being data-dependent. (See for example Figure 5.)

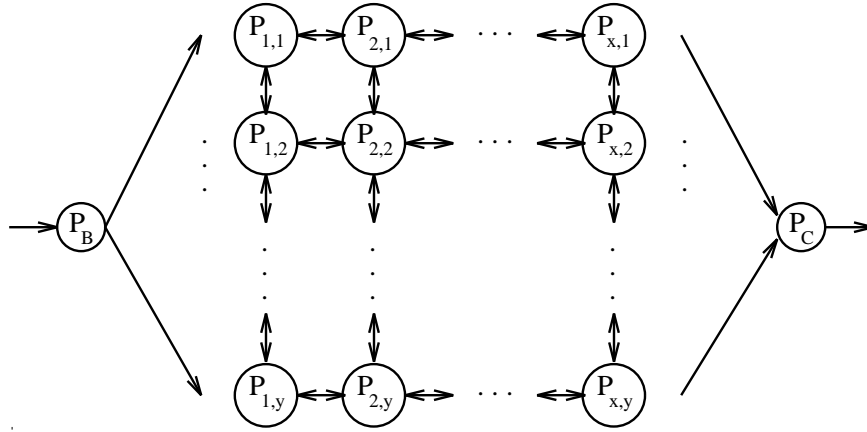


Figure 4: Example Geometric Skeleton

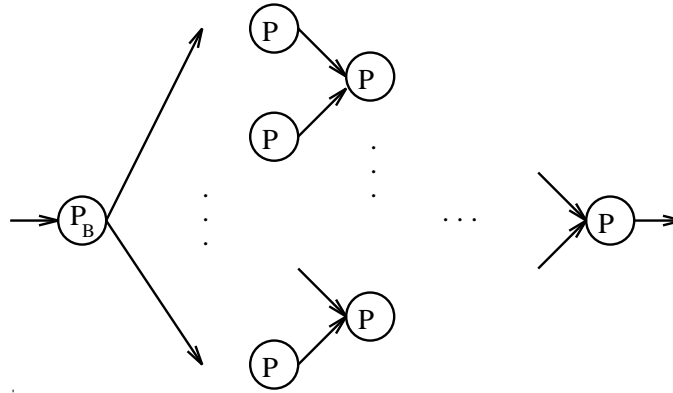


Figure 5: Example Combining Tree Skeleton

## 2.4 Nelson and Snyder's Skeletons

Nelson and Snyder [NS88] examined the following three skeletons:

**Compute-Aggregate-Broadcast** Appropriate algorithms are composed of three basic phases: a *Compute* phase performing some basic computation, an *Aggregate* phase (usually a tree-based computation) combining local data into one, or a few, global values, and a *Broadcast* phase returning global information, or a directive based upon it, back to each process. This skeleton may iterate on the three phases, not necessarily starting with the *Compute* phase, but the phases are generally in the same order. A suitable problem for solution with this skeleton is the parallel implementation of the Jacobi iterative method for solving Laplace's equation.

**Pipeline and Systolic** Suitable algorithms have sub-computations that are assigned to dedicated processes with the data "flowing" through the processes, and are expected to have locality of communication, a regular communication structure, and only a few different types of simple processes. Problems suitable for solution using this skeleton include band matrix multiplication, dynamic programming and Hough transform. (See Figure 3 for an example *Pipeline* form, and Figure 6 for an example *Systolic* form.)

**Divide and Conquer** This is as described above (see Figure 1).

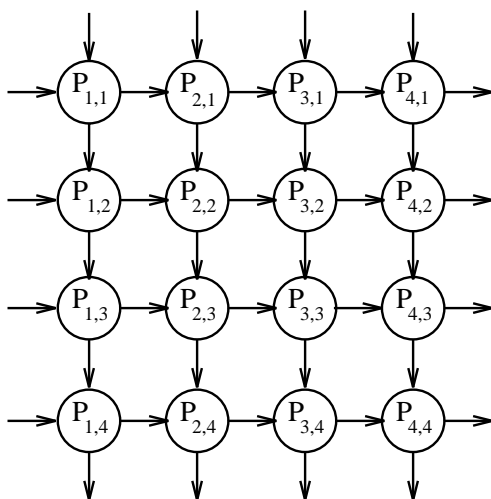


Figure 6: Example Systolic Skeleton

## 2.5 Gehringer *et al.*'s Skeletons

Gehringer *et al.* [GSS87] provide the following list of skeletons described as task graphs:

**Asynchronous** Processes work independently with no precedence constraints between each others' operation, communicating via shared data or message-passing, provided this induces no explicit dependencies between the processes. Suitable problems include matrix multiplication and solving partial differential equations.

**Synchronous** Processes execute in parallel and are explicitly synchronised, executing in a lock-step fashion, with communication taking place at the synchronisation points. This skeleton underlies other skeletons, being a special case of the *Multiphase* skeleton with an empty serial phase, also a linear synchronous *Pipeline* with data progressing between processes in lock-step under a global clock. Suitable problems for the *Synchronous* skeleton include the synchronous Jacobi algorithm for solving partial differential equations. (See for example Figure 7.)

**Multiphase** This is composed of a serial phase (with a single active process) and a parallel phase (with several concurrently active processes) alternating during the computation. The serial phase (master) often controls the actions of processes in the parallel phase (slaves), such as allocating work to the parallel processes according to some criteria. Suitable problems include power-systems simulations and travelling-salesman. (See for example Figure 8.)

**Partitioning** Processes divide the work amongst themselves during a divide phase, compute on their partition during a work phase, and combine the results together during the merge phase. This skeleton tends to be characteristic of sorting and searching algorithms. It may be used to implement *Divide and Conquer* algorithms (as described above), such as Quicksort, also for decomposing a computation for parallel execution. (See for example Figure 1.)

**Pipeline** This models synchronous and asynchronous *Pipelining*, as described above. A suitable problem is the systolic Metropolis algorithm. (See for example Figure 6.)

**Transaction-Processing** This covers the algorithms which are not described by the rest of this list. Requests arrive at the task force from an external source and are fielded by one of the parallel processes. It is usually important to maintain the consistency of

global data, and how synchronisation of access can be performed efficiently. A suitable problem is railway simulation. The variety of problems is too broad to construct a graph for this skeleton, hence it should not really be described as a skeleton.

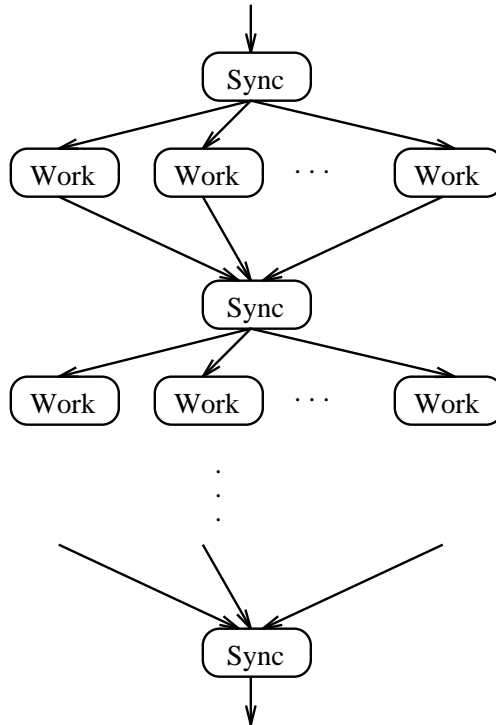


Figure 7: Example Synchronous Skeleton

## 2.6 Quinn's Skeletons

Quinn [Qui87] provides the following classes:

**SIMD** This is for data-parallel algorithms, where processes work in lock-step with zero synchronisation costs.

**MIMD Pipelined** A *Pipelined* algorithm is as described above (see Figure 3). A *systolic* algorithm is a generalised kind of *Pipelined* algorithm with more than one direction of data flow. (See for example Figure 6.)

**MIMD Partitioning** The problem is divided into sub-problems that are solved by individual processors, then the solutions are combined to form the problem solution. The implicit synchronisation among the processors means such *Partitioning* algorithms are sometimes called Synchronised algorithms. There are two categories of *Partitioning* algorithms: pre-scheduled algorithms, where a process is allocated its share of the computation at compile-time, and self-scheduling, where processes are allocated work from a global list of work at run-time.

**MIMD Relaxation** All processes work asynchronously towards the same goal (as in *Partitioning*), or there may be some specialisation of purpose (as in *Pipelined*), but essentially no processor has to wait for data from another processor, working with the most recently available data. Such algorithms are sometimes called *Asynchronous* algorithms.

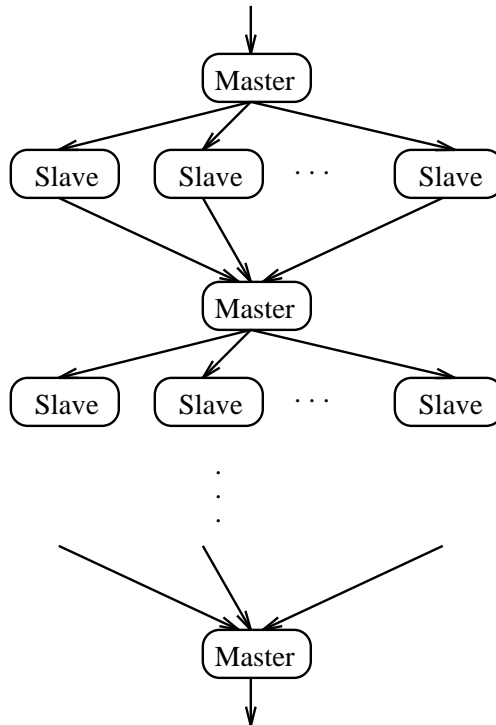


Figure 8: Example Multiphase Skeleton

## 2.7 Goodeve's Skeletons

Goodeve [Goo94] lists the following skeletons:

**Data-Parallel** This involves a single thread of control, executing in parallel over many computing elements. Example problems include vector and matrix operations and scan operations.

**Function Evaluation** Function notations tend to expose data dependencies, hence potential concurrency. Independent sub-trees formed from expression evaluation by the application of recursive relations can then be executed in parallel.

**Divide-and-Conquer** This is as described above (see Figure 1).

**Client-Server** The algorithm consists of a collection of entities that provide services. A service is requested by a client from an entity providing that service (server) by passing a request, with the result being returned to the client. This is usually referred to as an aspect of Object-Oriented programming.

**Farming** This is as described above (see Figure 2).

**Pipelining** This is as described above (see Figure 3). It is appropriate if the *Pipeline* latency can be traded for the frequency at which results are produced. Problems suitable to this skeleton include those associated with signal processing.

## 2.8 Gabber's Skeletons

Gabber [Gab90] provides the following classification:

**Tree Computations** Problems are solved by being broken into several simpler sub-problems which are solved recursively. The solution process resembles a tree of computations in

which the data flow from the root into the leaves, and solutions flow back up towards the root (see Figure 1). Examples include *Divide and Conquer* (as described above), game tree search and combinatorial search.

**Crowd Computations** Algorithms are defined in terms of a set of co-operating processes, each an independent process communicating and synchronising with its peers, mainly by message-passing. The communication structure of *Crowd Computations* normally follows some regular graph, such as a ring, mesh, torus, hypercube or tree, which may change during the computation. Also, processes may use shared-memory to pass information asynchronously.

## 2.9 Rabhi's Skeletons

Rabhi [Rab93] provides the following categorisation, referring to skeletons as paradigms:

**Recursively Partitioned** A problem is solved dynamically by recursively dividing it into sub-problems which are solved independently, in parallel, with the final result obtained by recursively combining the solutions of the sub-problems. This is more general than *Divide and Conquer* (as described above), which exploits conservative parallelism because all sub-problems need to be solved to compute the solution. Speculative parallelism is also exploited by this skeleton, by building partial solutions and testing in parallel all possible improvements, discarding a partial solution when no further improvements can be made, so solving sub-problems without knowing the usefulness of their results. Suitable problems include sorting, computing convex hulls, N-queens and combinatorial search (see for example Figure 1).

**Process Networks** The computation is divided into stages, with the data flowing through the stages, which can operate concurrently. *Process Networks* may be uniform (if the same operation is repeated at every stage), multi-function (if stages implement different functions), static or dynamic, and may have any communication pattern. A *Pipeline* is an instance of this skeleton, illustrated in Figure 3, as is the *Systolic* skeleton, illustrated in Figure 6.

**Distributed Independent** This is the *Farm* skeleton, as described above (see Figure 2). A suitable problem is random number generation.

**Iterative Transformation** This operates on a set of objects, each with local data and possibly sharing global values, where the objects are transformed through several iteration steps. During an iteration step, one or several of the following operations can be performed: local (each object computes using local data, or data in other objects, including global data), combine (combining groups of objects to form another set of objects), and global (computing global data, possibly using local data). The *Iterative Transformation* is static if there is no combine operation (parallelism arising from the concurrent application of local operations to each object; this has also been implemented [SR96]), and dynamic if the number of objects changes at run-time (additional parallelism arising when a group of objects can be combined independently from other groups). Special cases of *Iterative Transformation* are Cole's *Iterative Combination* (see above) and Nelson and Snyder's *Compute-Aggregate-Broadcast* (see above). Suitable problems include numerical analysis, image processing, parallel branch-and-bound and graph problems.

## 2.10 Tropes

An approach to the utilisation of algorithmic skeletons with generative communication was undertaken in the production of *tropes* [HMS93]. Tropes are described as primitive program

schemes taking the form of parameterised conditional rewrite rules, and these are applied to the GAMMA language.

GAMMA (General Abstract Model for Multi-set manipulation) [BCL88] is a minimal language based upon a single data structure, the multi-set, and the corresponding control structure metaphor, the chemical reaction. The state of the system is a multi-set of molecules (chemical solution), and molecules interact according to reaction rules (chemical reactions). Programs consist of sets of reaction pairs (multi-set rewritings) whose components are a condition (predicate characterising the molecules to transform) and an action (function yielding new molecules). Execution of a GAMMA program is a succession of reactions consuming molecules according to a specific reaction’s condition and producing new ones. Execution terminates once a stable state is reached, i.e., no molecules satisfy the reaction rules.

Tropes are a series of five rewrite rules skeletons for GAMMA:

**Transmuter** applies the same transformation operations to all elements of the multi-set until none satisfy the conditions, keeping the size of the multi-set constant. This is equivalent to the *Task Queue* skeleton, repeatedly applying a function to available jobs (see Figure 2).

**Reducer** reduces the size of the multi-set by applying a function to pairs of elements satisfying the given condition. This is equivalent to the combining *tree* skeleton (see Figure 5).

**Optimiser** optimises the multi-set according to a particular criterion while preserving the structure of the multi-set.

**Expander** decomposes the elements of a multi-set into a collection of basic values. This is equivalent to the distributing version of the *tree* skeleton.

**Selector** acts as a filter, removing elements from the multi-set satisfying a certain condition. From BMF (c.f. Section 4.1), this is a combination of *map* and *reduce* operations, hence a combination of the *transmuter* and *reducer* tropes.

As can be seen, the list of tropes is similar to other skeletons proposed elsewhere.

### 3 Towards a General Classification

It can be seen readily that there are several commonalities to the various lists.

Most lists of algorithmic skeletons surveyed in the previous section involve a *Divide and Conquer*-like skeleton: *DC* [DFH<sup>+</sup>93], *Fixed Degree Divide and Conquer* [Col89], *Divide and Conquer* [NS88], *Divide-and-Conquer* [Goo94], *tree* [DMO<sup>+</sup>91], *Partitioning* [GSS87], *Tree Computation* [Gab90], *Reducer*, *Expander* [HMS93] and *Recursively Partitioned* [Rab93]. Not all of these skeletons contain the same level of abstraction or functionality, but they are all of a *Divide and Conquer*-like form. The relationships between these skeletons are illustrated in Figure 9, with the most general skeletons being higher up the hierarchy, and those at the same level in the hierarchy being in the same box.

Similarly, most of the surveyed lists involve a *Pipeline* and/or *Systolic*-like skeleton: *PIPE* [DFH<sup>+</sup>93], *pipe* [DMO<sup>+</sup>91], *Pipelining* [Goo94], *Pipeline and Systolic* [NS88], *Pipeline* [GSS87], *MIMD Pipelined* [Qui87] and *Process Networks* [Rab93]. Again, the skeletons offer different levels of abstraction or functionality, but are all of the same general *Pipeline* and/or *Systolic* form. The relations between these skeletons are illustrated in Figure 10, with the more general skeletons being higher up the hierarchy, and those at the same level in the hierarchy being in the same box.

Also, many of the lists involve a *Farm*-like skeleton: *FARM* [DFH<sup>+</sup>93], *farm* [DMO<sup>+</sup>91], *Task Queue* [Col89], *Farming* [Goo94], *Transmuter* [HMS93] and *Distributed Independent*

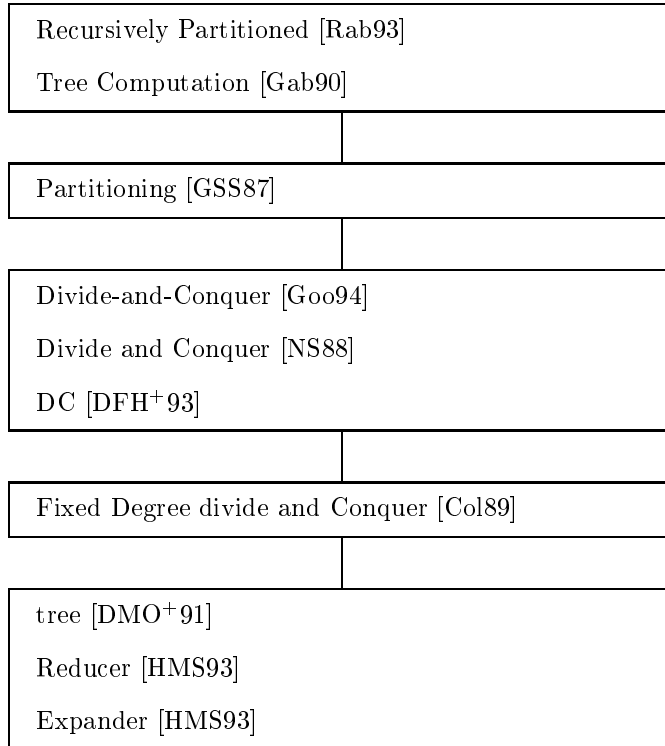


Figure 9: Hierarchy of Divide and Conquer-like Skeletons

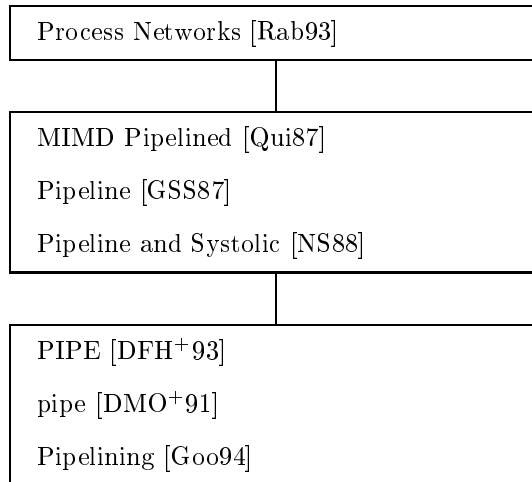


Figure 10: Hierarchy of Pipeline and/or Systolic-like Skeletons

[Rab93]. These skeletons are generally on the same level of a hierarchy of functionality, except for the *Task Queue*, which is more general because the slave processes may produce new work to be performed by other slaves.

A potential classification involves an *Iterative Transformation*-like skeleton: *Iterative Combination* [Col89], *Computer-Aggregate-Broadcast* [NS88] and *Iterative Transformation*

[Rab93].

Another potential classification involves skeletons for particular process graphs: *Crowd Computation* [Gab90]; this includes *Data Parallel*-like skeletons as restrictive cases requiring homogeneous processes: *geometric* [DMO<sup>+</sup>91], *Data-Parallel* [Goo94] and *SIMD* [Qui87].

This still leaves the following seemingly disparate skeletons to be allocated to form a general classification: *RaMP* [DFH<sup>+</sup>93], *DMPA* [DFH<sup>+</sup>93], *loop* [DMO<sup>+</sup>91], *Cluster* [Col89], *Function Evaluation* [Goo94], *Client-Server* [Goo94], *Asynchronous* [GSS87], *Synchronous* [GSS87], *Multiphase* [GSS87], *Partitioned* [Qui87], *Relaxed* [Qui87] and *OPTimiser* [HMS93].

The beginnings of a general classification of algorithmic skeletons are seen by the author as involving various of the above-cited collections of skeletons. A tentative proposal for the basis of such a classification is presented in [Cam94], and is as follows:

**Recursively Partitioned:** this is taken from Rabhi’s classification, covering the *Divide and Conquer*-like skeletons identified above. It consists of a generalised, tree-like structured skeleton exploiting both conservative and speculative parallelism. Suitable problems are solved through dynamic, recursive division of the problem into sub-problems which are solved independently in parallel, then recursively combining the sub-solutions. Such problems include Divide and Conquer, game trees, quicksort, combinatorial search, approximate integration, matrix multiplication, connected ones, etc.

**Task Queue:** this is Cole’s generalisation of the *Farm* skeleton, allowing slave processes to produce new work to be performed by other slaves. It covers the *Farm*-like skeletons identified above. This includes such problems amenable to solution by the broadcasting of jobs to be solved independently. Example problems include one-to-all shortest paths, random number generation, etc.

**Systolic:** such a skeleton covers the *Pipeline* and/or *Systolic*-like skeletons identified above. It consists of stages which have data flowing between them and may operate concurrently. Example problems include signal processing applications, the systolic Metropolis algorithm, band matrix multiplication, dynamic programming, Hough transform, etc.

**Crowd:** this is from Gabber’s classification covering problems represented by graphs including data-parallel problems. It is distinct from the *Systolic* skeleton in that there is no flow of data between the concurrently operating stages. Examples include the “combine” skeleton [FJL<sup>+</sup>88] with its butterfly graph structure, which is also used in problems such as Batcher’s Bitonic Merge Sort [Bat68], FFT, etc.

These categories all appear to be sufficiently distinct to merit separate classification, and sufficiently general to encapsulate many minor algorithmic variations. Both *Recursively Partitioned* and *Task Queue* capture problems suitable for solution by being distributed, solved independently then their solutions combined. On the other hand, both *Systolic* and *Crowd* capture problems solved by application to distributed graphs of processes. *Recursively Partitioned* differs from *Task Queue* in that it recursively divides the problem into indivisible sub-problems for solution rather than broadcasting ready sub-problems. *Systolic* differs from *Crowd* in that it has the notion of flow between stages of processes operating concurrently, with the data being supplied as a (continuous) stream rather than a single collection of data being applied to the process graph together.

## 4 Other Skeleton Classification Schemes

### 4.1 BMF

The Bird-Meertens Formalism (BMF) [Bir89] was originally conceived as a derivational style for functional programs [Ski92, Ski90]. BMF consists of a set of theories built on

a base algebra with unary and binary functions, each theory captures the behaviour of a particular class of data structures. A BMF theory begins with base types and extends them to new types using type functors, adding to the base algebra a set of second-order functions and laws which relate them to one another. The laws provide a set of meaning-preserving transformations which can be applied for optimisation, or regarded as rewrite rules, while the completeness result is intended to guarantee that the language is sufficiently expressive. So BMF extends the concept of Abstract Data Types (ADTs) to that of categorical data types<sup>1</sup> [Ski91]. Unlike ADTs, categorical data types have operations, equations relating them, and a guarantee that all of the required operations and equations are present. A theory is built from the constructors of a type using a categorical construction.

However, it was discovered that BMF<sup>2</sup> also has an application as the basis for a locality-based, data-parallel, functional programming language. A BMF program consists of a composition of functions on a particular data type [CS92, CS91]. Communication in the model is restricted to a set of second-order functions, each of which encapsulates a particular communication pattern requiring only a constant size of neighbourhood locality [Ski90]. BMF does not directly express parallelism, it is the compiler's task to implement the operations in a parallel manner. Both parallelism and communication are thus hidden from the direct concern of the programmer.

Considering the theory of join or concatenation lists in BMF, any homomorphism on lists can be expressed as the composition of a *map* followed by a *reduce*. So, *map* and *reduce* are the basic algorithmic skeletons for operations on lists. Their shared topology (the standard topology for the theory of lists) is one including a logarithmic depth spanning-tree and a Hamiltonian cycle.

Similarly, for other type theories (matrices, graphs, sets, etc.) there are constructors such that homomorphisms on those types can be expressed as the composition of those constructors (skeletons).

The major drawbacks of BMF are that it is solely data-parallel and the functions cannot be nested. Though Skillicorn does now appear to acknowledge the possibility of the nesting of functions in BMF by having the compiler flatten the nested functions into correct BMF [Ski94]. However, despite it having been recognised that data-parallelism is more applicable than otherwise previously thought [HS86], it is still lacking in expressibility, unless it is loosened to become multi-threaded with predominantly breadth-first evaluation, as is the case with F-code [MSS93].

F-code expresses data-parallelism through operations on non-scalar objects. It supports the whole range of parallel operations by providing a rank coercion mechanism, which permits the application of parallel operators to any combination of objects, irrespective of whether their ranks match or not. F-code also exploits functional parallelism by representing the concurrency of evaluating the operands of an operation.

## 4.2 BACS

The Basel Algorithm Classification Scheme (BACS) [BKG<sup>+</sup>93, BG95] is driven by the goals of reusability and portability. It is more of a scheme for describing algorithms, rather than placing algorithms into a set of categories.

According to BACS, algorithms can be described by the following six-tuple:

**process structure** either static or dynamic;

**process topology** either worker, pipeline-like, mesh-like, tree-like or hypercube;

**process macro execution structure** the series of *events* (see below), used to distinguish between similar execution structures;

---

<sup>1</sup>Categorical data types come equipped with a set of second-order operations such as *map* and *reduce* that can be used as the primitives of a data-parallel computation model. The form of such operations guarantees the existence of much parallelism.

<sup>2</sup>References to BMF in this section are generally specific to the particular BMF theory of lists.

**interaction** either global-coupled (barrier, broadcast, line broadcast, fan-in, pipe-step, rotate, red-black-shuffle), global-decoupled (local, semaphore, mailbox, monitor, LINDA-like) or direct (signal-wait, send-receive, exchange);

**data placement** what data is placed;

**data distribution** either global, static or dynamic.

*Events* are of three types: **C**alculation step, **I**nteraction step, or **D**aemon call, and are either total or partial. Total events have all processes participate, and are indicated by a superscripted *t*. Partial events have a non-empty subset of processes participating, and are indicated by a superscripted *p* which may also include a parenthesised description of the set of participating processes.

The different types of event may be even further specified. Calculation steps may be subscripted to indicate different sequences of statements. Interaction steps may be subscripted to refer to the type of interaction. Daemon calls may be subscripted to distinguish between creation and deletion calls, where creation calls are further parameterised by the number and kind of created processes.

Furthermore, macroscopic execution structures can indicate the type of loops, distinguishing between loops with **FIX**ed numbers of repetitions, and those with **COND**itional numbers of repetitions. Each is subscripted by the topology and contains the sequence of events in parentheses.

The BACS classification is principally based on process structure and data distribution, resulting in a categorisation of the form:

$$\{ \text{Static or Dynamic} \} \text{Process}, \{ \text{Static and/or Global and/or Dynamic} \} \\ \text{Data}$$

Pure classes have all data distributed in the same way, viz: **SPSD**, **SPGD**, **SPDD**, **DPSD**, **DPGD**, **DPDD**.

Simple mixed classes have data distributed in two ways, viz: **SPSGD**, **SPGDD**, **DPSGD**, **DPGDD**.

Total mixed classes have data distributed in all ways, viz: **SPSGDD**, **DPSGDD**.

A further time-related classification can be used based on the macroscopic execution structure which contains information about the lapse of time, so dividing algorithms into further sub-classes.

Additionally, topology can be used as a further means of sub-classification. The combination of topology and data distribution leads to a huge set of possible classes. So, one can restrict to a fixed hardware topology to reduce the possible combination of process topologies and data distributions.

The BACS classification scheme forms the basis for the ALWAN language, which is used for writing algorithmic skeletons, offering reuse-in-the-small constructs [BFHO94, BFHO94]

### 4.3 Geerling's Classification

Geerling [Gee95] presents a classification of algorithmic skeletons according to broad classes of style, rather than general functionality. There are two classifications, the first is:

**control-oriented** a skeleton is described by means of a control structure. In this style, various skeletons operate on a common data structure;

**data-oriented** a skeleton is described by valid operations on a data structure, e.g., a processor array modelled by a list, and communication is abstracted into skeletons permuting it. In this style, the control structure is used to arrange the operations specified by the skeleton.

This distinction is related to control- and data-parallelism, but is not a completely comparable relation.

The second classification captures a sense of skeletons being found at different levels of abstraction:

**architectural** a skeleton for a particular architecture, having a straightforward interpretation as a typical elementary computation on that architecture;

**algorithmic** a function expressible in terms of the available architectural skeleton.

Geerling views the process of using skeletons by composing individual skeletons for forming programs, and mapping algorithmic skeletons onto architectural skeletons. This is similar to the view proposed by the author in [Cam94], where skeletons are seen as composable structures, and an algorithmic skeleton is mapped to the architectural model in the CLUMPS model of computation, then mapped to the target architecture.

## 5 Other Skeleton-like Approaches

There exist several approaches to parallel computation which, though not describing themselves as using algorithmic skeletons, are definitely skeleton-like. A brief overview of these is presented here.

Shared Abstract Data Types (SADTs) [DDGN96] are designed to be an alternative to message passing and shared memory to support high performance sharing in applications. They are intended to be an effective abstraction mechanism which does not compromise performance, but insulates the programmer from details of data management. Essentially, they represent high-level abstract data types (ADTs) shared between parallel processes, and implemented in parallel, hiding the implementation from the user. SADTs offer guarantees of behaviour based on linearizability conditions, and high scalable performance on scalable architectures. The implementation strategy is for SADTs to be distributed (so activity is not focussed at any one point), and free from locks and critical-sections (to provide maximal concurrency). Example SADTs include FIFO queues and the *accumulator*, which captures a more general form of several existing combining-style operations. They are similar to algorithmic skeletons in terms of motivation, encoding and their particular mode of computation.

Another skeleton-like abstract data type is the “M-tree” (mesh-tree) Abstract Data Type [WFK96], which is a high-level ADT for programming parallel applications involving adaptive computation, capturing the data and computational structures in adaptive problems. It is equipped with a rich set of access functions, including higher-order operators abstracting commonly used computational patterns in parallel adaptive computation. The M-tree is a generalisation of the quadtree, where the degree of a non-leaf node in the  $x$  dimension is  $R_x$ , and the degree in the  $y$ -dimension is  $R_y$  (for a quadtree,  $R_x = R_y = 2$ ). Each node in an M-tree represents a region of the domain, and its subtrees are sub-regions overlaying the node region. This provides an abstraction of the hierarchical partition in some arbitrary order. First order operations perform basic query and update operations on each node, whilst higher order operations abstract commonly used patterns of parallel adaptive computation.

BMF++ [Cro96] is a paradigm which uses BMF constructs to add parallel operations to the traditional sequential programming language C++. The parallelism is provided by the *map*, *reduce* and *zip* operations from the BMF theory of lists.

Stages and Transformations (SAT) [Gor96] is designed to support the derivation of parallel distributed memory programs, based on the transformation rules for BMF and higher order functions over lists. A program is viewed as a single thread of stages, with parallelism occurring within stages. A *target view* is considered to be the perception when developing a program, whose aim is the parallel composition of sequential processes. A *design view* is considered to be orthogonal to the target view, regarding a parallel program as a sequence

of stages, each encapsulating parallelism of possibly different kinds. The design view in SAT consists of two layers: at the inter-stage layer, stages are sequentially composed and invoked, and; at the intra-stage layer, each stage is a parallel program invoking mixtures of computations and communications. The SAT approach consists of specifying an algorithm in the design view in an obvious way, then transforming it into a provably correct target view. The BMF functionals are the source of parallelism, and are used to represent the stages of the program under development.

Communication Skeletons [Ski96] are intended to extend and unify existing models, such as systolic arrays, fixed-connection network algorithms, reconfigurable network algorithms, bus-based networks, and homomorphic skeletons. They incorporate aspects of fixed-connection-topology approaches and BSP [Val90], where fixed-connection skeletons are based on algorithms that assume a particular interconnection structure with unit-time, nearest-neighbour communication, and BSP computations are in the form of supersteps which involve local computation and global communication which is not guaranteed to be complete until the barrier synchronisation at the end of each superstep. A Communication Skeleton is an interleaving of a set of uniform communication steps with computation steps, encapsulating useful operations with internal parallelism, and allowing the nearest-neighbour limitation to be weakened so that in a unit time step, a message can be received by all processors along a path. Communication Skeletons do not abstract from topology, instead an assumption is made about the underlying topology (design topology), which is then mapped to the target topology. The best choices for a design topology (in terms of simulation capabilities) are meshes and complete binary trees.

Design Patterns [BFVY96] are described as an object-oriented algorithmic templating scheme intended to capture the solutions to building recurring design problems in systematic and general ways. Besides problem and solution descriptions, Design Patterns include explanations of the applicability, trade-offs and consequences of the solution. The Design Pattern also illustrates how to implement the solution in an object-oriented programming language. Design Patterns are not code, so must be implemented each time they are applied. Designers supply application-specific names of the key classes and object in the pattern, then implement class declarations and definitions as the pattern prescribes.

## 6 Comparison

The author's classification is primarily empirical, being justified experientially by its basis in fellow researchers' actual experience with algorithmic skeletons. The classification represents a separation of the proposed skeletons into distinctly identifiable functionality categories to form the classification proposed here.

BMF is a mathematically formal, but computationally restrictive scheme, limited to data parallel skeletons. However, its *map* and *reduce* operations do correspond to the *Task Queue* and *Recursively Partitioned* skeletons, respectively. In order to represent the *Crowd* skeleton, BMF's theory of lists needs to be extended to include Compound List Operators (CLOs) [KS93] in order to be able to represent the necessary communication patterns. Therefore, the BMF theory of lists is not as general a classification of skeletons as that proposed by the author here. Also, the formal approach of BMF led to a similar (though limited) range of skeletons as the empirical approach detailed here.

The BACS scheme appears to be more of a means for surveying algorithms than classifying algorithmic skeletons. The classification is particularly broad, incorporating a large range of parameters to be accounted for. So, it is not as abstract as the algorithmic skeleton classification proposed by the author here. However, this has formed the basis for the ALWAN language, specifying algorithms in a similar style to the BACS characteristics.

Geerling has divided algorithmic skeletons into two groups, for the software side, and the hardware side. The skeletons on the software side correspond more closely to algorithmic skeletons than the hardware side skeletons do, at least as far as their general aim is

concerned. Also, the software skeletons are based on broad classes of style, rather than on functionality. So abstracting too far from the skeletal structures which algorithmic skeletons are intended to capture.

SADTs are a promising scheme, but the examples referred to are of a different level to traditional algorithmic skeletons, considering more specialised operational structures.

The M-tree is a particularly limited kind of abstract data type, equivalent to a single skeleton, like the *Recursively Partitioned* skeleton.

Due to BMF++ being C++ extended with BMF constructs, it suffers from the same draw-backs as BMF does (as above).

Stages and Transformations are also limited in their source of parallelism being the BMF operations for the theory of lists, so suffer from similar expressibility problems as BMF does.

Communication Skeletons, in borrowing from BMF, make assumptions about the target topology, rather than providing a truly architecture-independent mapping, as other skeletons do. However, their application to the BSP model of parallel computation is particularly interesting, given that BSP is a particularly promising model.

Design Patterns are object-oriented in nature, and explicitly incorporate additional documentation of each pattern. Despite their popularity within the object-oriented community, they appear to be limited to the object-oriented domain and provide a seemingly *ad hoc* mechanism for code reuse.

## 7 Summary

Algorithmic skeletons are seen as being high-level parallel programming constructs necessary in parallel programming for the programmer to be able to program parallel programs easily. There are currently many algorithmic skeletons proposed, of which several have been re-examined here.

The profusion of algorithmic skeletons necessitates some form of a classification of them. One such classification was proposed based upon the survey of the skeletons re-examined. Other skeleton classifications were also examined. Each classification scheme has its own perspective. BMF is from the set of operations provided by category theory for data types. BACS is a general categorisation of algorithms. Geerling classifies skeletons according to broad classes of style. The author's classification attempts to take existing skeletons which have been proven useful by experience, and groups them together under general but distinct classifications.

Further work is underway in the examination of LINDA[Gel85, CG89] for Case-Based Reasoning (CBR), where part of this work involves the identification of macro operations (algorithmic skeletons) for performing CBR in LINDA. Out of this work has resulted the proposal of two such operations (skeletons) for the intersection and comparison of tuple spaces. The intersection operation has itself been generalised to a skeleton for performing operations on each tuple in a tuple space. This work clearly demonstrates the practical development of algorithmic skeletons in a particular application (CBR) using a particular programming style (LINDA-based).

## Acknowledgements

Thanks are offered to Murray Cole of Edinburgh University for his various comments on this work, and to Andrew Vickers at York University for his suggestions.

## References

- [Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
- [Bat68] K. E. Batcher. Sorting networks and the application. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.
- [BCL88] J.-P. Banâtre, A. Coutant, and D. LeMétayer. A parallel machine for multi-set transformations and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.
- [BdRP93] N. Berrington, D. de Roure, and J. Padget. Guaranteeing unpredictability. *The Computer Journal*, 36(8):723–733, 1993.
- [BFHO94] H. Burkhart, R. Frank, G. Hächler, and P. Ohnacker. Structured parallel programming: How informatics can help overcome the software dilemma. In *Proceedings of the Priority Programme Informatics Research Information Conference*, 1994.
- [BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.
- [BG95] H. Burkhart and S. Gutzwiller. Software reuse and portability of parallel programs. In E. El-Rewini and D. Shriver, editors, *Proceedings of the twenty-eighth Annual Hawaii International Conference on System Sciences*, volume 2, pages 289–298. IEEE Computer Society Press, 1995.
- [Bir89] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computer Science*, pages 151–216. NATO ASI Series F55, Springer-Verlag, 1989.
- [BKG+93] H. Burkhart, C. F. Korn, S. Gutzwiller, P. Ohnacker, and S. Waser. Bacs: Basel algorithm classification scheme. Technical Report 93-3, University of Basel, 1993.
- [Cam94] D. K. G. Campbell. *CLUMPS: a candidate model of efficient, general purpose parallel computation*. PhD thesis, Department of Computer Science, University of Exeter, October 1994.
- [CG89] N. Carrierio and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, 1989.
- [Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.
- [Cro96] D. C. Crooke. Robustness and performance in structured parallelism. In M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 135–145. IOS Press, 1996.
- [CS91] W. Cai and D. B. Skillicorn. Evaluating a set of message-passing routines on Transputer networks (extended abstract). Department of Computing and Information Science, Queen’s University, Kingston, October 1991.
- [CS92] W. Cai and D. B. Skillicorn. Evaluation of a set of message-passing routines on Transputer networks. In A. R. Allen, editor, *Transputer systems - Ongoing Research*. IOS Press, 1992.

- [DDGN96] J. Davy, P. Dew, D. Goodeve, and J. Nash. Concurrent sharing through abstract data-types: A case study. In M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 91–103. IOS Press, 1996.
- [DFH<sup>+</sup>93] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In *Parallel Architecture and Languages Europe (PARLE '93)*, 1993.
- [DMO<sup>+</sup>91] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. Technical Report TR-25/91, Dipartimento di Informatica, Università di Pisa, December 1991.
- [FJL<sup>+</sup>88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall International, 1988.
- [Gab90] E. Gabber. VMMP: A practical tool for the development of portable and efficient programs for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):304–317, 1990.
- [Gee95] M. Geerling. *Transformational Development of Data-Parallel Algorithms*. PhD thesis, University of Nijmegen, 1995.
- [Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–121, 1985.
- [Goo94] D. M. Goodeve. *Performance of Multiprocessor Communications Networks*. PhD thesis, Department of Electronics, University of York, 1994.
- [Gor96] S. Gorlatch. Stages and transformations in parallel programming. In M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 147–161. IOS Press, 1996.
- [GSS87] E. F. Gehringer, D. P. Siewiorek, and Z. Segall. *Parallel Processing: The Cm\* Experience*. Digital Press, 1987.
- [HMS93] C. Hankin, D. Le Métayer, and D. Sands. A parallel programming style and its algebra of programs. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93: Parallel Architectures and Languages Europe*, pages 367–378. Springer-Verlag, 1993. LNCS 694.
- [HS86] W. D. Hillis and G. L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(2):1170–1183, 1986.
- [KS93] K. G. Kumar and D. B. Skillicorn. Data parallel geometric operations on lists. Department of Computing and Information Science, Queen's University, Kingston, Canada, January 1993.
- [McC93] W. F. McColl. An architecture independent programming model for scalable parallel computing. In *General Purpose Parallel Computing*, pages 1–17. British Computer Society Parallel Processing Specialist Group, December 1993. University of Westminster.
- [MSS93] V. B. Muchnick, A. V. Shafarenko, and C. D. Sutton. F-code and its implementation: a portable software platform for data-parallelism. *The Computer Journal*, 36(8):712–722, 1993.

- [NS88] P. A. Nelson and L. Snyder. Programming paradigms for nonshared memory parallel computers. In L.H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, chapter 1. MIT Press, 1988.
- [Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987.
- [Rab93] F. A. Rabhi. Exploiting parallelism in functional languages: A “paradigm-oriented” approach. In P. Dew and T. Lake, editors, *Abstract Machine Models for Highly Parallel Computers*. Oxford University Press, 1993.
- [Ski90] D. B. Skillicorn. Architecture-independent parallel computation. Technical Report ISSN-0836-0227-90-268, Department of Computing and Information Science, Queen’s University, Kingston, Ontario, March 1990.
- [Ski91] D. B. Skillicorn. Practical parallel computation. Technical Report ISSN-0836-0227-91-312, Department of Computing and Information Science, Queen’s University, Kingston, Ontario, August 1991.
- [Ski92] D. B. Skillicorn. Parallelism and the Bird-Meertens Formalism. Department of Computing and Information Science, Queen’s University, Kingston, April 1992.
- [Ski94] D. B. Skillicorn. Department of Computing and Information Science, Queen’s University, Kingston, 1994. (Personal communication).
- [Ski96] D. B. Skillicorn. Communication skeletons. In M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 163–178. IOS Press, 1996.
- [SR96] J. Schwarz and F. A. Rabhi. A skeleton-based implementation of iterative transformation algorithms using functional languages. In M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 119–133. IOS Press, 1996.
- [Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
- [WFK96] Q. Wu, A. J. Field, and P. H. J. Kelly. Data abstraction for parallel adaptive computation. In M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 105–118. IOS Press, 1996.