Parallel Random Access Machines (PRAM)

►PRAM (Parallel Random Access Machines)

Parallel Skeletons

Sergei Gorlatch[1], Murray Cole[2]
[1]Westfälische Wilhelms-Universität Münster, Münster, Germany
[2]University of Edinburgh, Edinburgh, UK

Synonyms

Algorithmic skeletons

Definition

A parallel skeleton is a programming construct (or a function in a library), which abstracts a pattern of parallel computation and interaction. To use a skeleton, the programmer must provide the code and type definitions for various application-specific operations, usually expressed sequentially. The skeleton implementation takes responsibility for composing these operations with control and interaction code in order to effect the specified computation, in parallel, as efficiently as possible. Abstracting from parallelism in this way can greatly simplify and systematize the development of parallel programs and assist in their cost-modeling, transformation, and optimization. Because of their high level of abstraction, skeletons have a natural affinity with the concepts of higher-order function from functional programming and of templates and generics from object-oriented programming, and many concrete skeleton systems exploit these mechanisms.

Discussion

Traditional tools for parallel programming have adopted an approach in which the programmer is required to *micromanage* the interactions between concurrent activities, using mechanisms appropriate to the underlying model. For example, the core of MPI is based around point-to-point exchange of data between processes, with a variety of synchronization modes. Similarly, threading libraries are based around primitives for locking, condition synchronization, atomicity, and so on. This approach is highly flexible, allowing the most intricate of interactions to be expressed. However, in reality many parallel algorithms and applications follow well-understood patterns, either monolithically or in composition. For example, image processing algorithms can often be described as *pipelines*. Similarly, *parameter sweep* applications present a parallel scheduling challenge which is orthogonal to the detail of the application and parameter space.

Programming with Skeletons

The term *algorithmic skeleton* stems from the observation that many parallel applications share common internal interaction patterns. The parallel skeleton approach proposes that such patterns be abstracted as programming language constructs or library operations, in which the implementation is responsible for implicitly providing the control "skeleton," leaving the programmer to describe the application-specific operations that specialize its behavior to solve a particular problem. For example, a pipeline skeleton would require the programmer to describe the computation performed by each stage, while the skeleton would be responsible for scheduling stages to processors, communication, stage replication, and so on. Similarly, in a parameter sweep skeleton, the programmer would be required to provide code for the individual experiment and the required parameter range. The skeleton would decide upon (and perhaps dynamically adjust) the number of workers to use, the granularity of distributed work packages, communication mechanisms, and fault-tolerance. At a more abstract level, a divide-and-conquer skeleton would require the programmer to specify the operations used to divide a problem, to combine subsolutions, to decide whether a problem is (appropriately) divisible, and to solve indivisible problems directly. The skeleton would take on all other responsibilities, from whether to use parallelism at all, to details of dynamic scheduling, granularity, and interaction.

Thus, in contrast to the micromanagement of traditional approaches, skeletons offer the possibility of *macromanagement* – by selection of skeletons, the programmer conveys macro-properties of the intended computation. This is clearly attractive, provided that the skeletons offered are sufficiently comprehensive collectively, while being sufficiently optimizable individually and in composition. Sometimes a skeleton may have several implementations, each geared to a particular parallel architecture, for example, distributed- or shared-memory, multithreaded, etc. Such customization has the potential for achieving high performance, portable across various target machines.

Application programmers gain from abstraction, which hides much of the complexity of managing massive parallelism. They are provided with a set of basic abstract skeletons, whose parallel implementations have a well-understood behavior and predictable efficiency. To express an application in terms of skeletons is usually simpler than developing a low-level parallel program for it.

This high-level approach changes the program design process in several ways. First, it liberates the user from the practically unmanageable task of making the right design decisions based on numerous, mutually influencing low-level details of a particular application and a particular machine. Second, by providing standard implementations, it increases confidence in the correctness of the target programs, for which traditional debugging is too hard to be practical on massively parallel machines. Third, it offers predictability instead of an *a posteriori* approach to performance evaluation, in which a laboriously developed parallel program may have to be abandoned because of inadequate efficiency. Fourth, it provides semantically sound methods for program composition and refinement, which open up new perspectives in software engineering (in particular, for reusability). And last but not least, abstraction, that is, going from the specific to the general, gives new insights into the basic principles of parallel programming.

An important feature of the skeleton-based methodology is that the underlying formal framework remains largely invisible to application programmers. The programmers are given a set of methods for instantiating, composing, and implementing diverse skeletons, but the development of these methods is delegated to the community of implementers.

In order to understand the spectrum of skeleton programming research, it is helpful to distinguish between skeletons that are predominantly data-parallel in nature (with an emphasis on transformational approaches), skeletons that are predominantly task-parallel or related to algorithmic classes, and the concrete skeleton-based systems that have embedded these concepts in real frameworks.

Data-Parallel Skeletons and Transformational Programming

The formal background for data-parallel skeletons can be built in the functional setting of the Bird–Meertens formalism (BMF), in which skeletons are viewed as higher-order functions (functionals) on regular bulk data structures such as lists, arrays, and trees.

The simplest – and at the same time the "most parallel" – functional is *map*, which applies a unary function f to each element of a list, that is,

$$map\, f\, [x_1, x_2, \ldots, x_n] \;=\; [f\, x_1, f\, x_2, \ldots, f\, x_n] \tag{1}$$

Map has the following natural data-parallel interpretation: each processor of a parallel machine computes function f on the piece of data residing in that processor, in parallel with the computations performed in all other processors.

There are also the functionals *red* (reduction) and *scan* (parallel prefix), each with an associative operator $\oplus$ as parameter:

$$red\,(\oplus)\,[x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \ldots \oplus x_n \tag{2}$$

$$scan\,(\oplus)\,[x_1, x_2, \ldots, x_n] = [\,x_1,\, x_1 \oplus x_2,\, \ldots,\, x_1 \oplus \ldots \oplus x_n\,] \tag{3}$$

Reduction can be computed in parallel in a tree-like manner with logarithmic time complexity, owing to the associativity of the base operation. There are also parallel algorithms for computing the scan functional with logarithmic time complexity, despite an apparently sequential data dependence between the elements of the resulting list.

Individual functions are composed in BMF by means of backward functional composition $\circ$, such that $(f \circ g)\, x \;=\; f\,(g\, x)$, which represents the sequential execution order on (parallel) stages.

Special functions, called *homomorphisms*, possess the common property of being well-parallelizable in a data-parallel manner.

Definition 1 (**List Homomorphism**) *A function h on lists is called a* homomorphism *with combine operation* $\circledast$*, iff for arbitrary lists* x, y*:*

$$h\,(x +\!\!+\, y) = (h\, x) \circledast (h\, y) \tag{4}$$

Definition 1 describes a class of functions, operation $\circledast$ being a parameter, which is why it can be viewed as defining a skeleton. Both map and reduction can obviously be obtained by an appropriate instantiation of this skeleton.

The key property of homomorphisms is given by the following theorem:

Theorem 1 (**Factorization**) *A function h on lists is a homomorphism with combine operation ⊛, iff it can be factorised as follows:*

$$h = red(\circledast) \circ map\ \phi \tag{5}$$

where $\phi\, a \; = \; h\,[a]$.

In this theorem, homomorphism has one more parameter beside ⊛, namely function ϕ. The practical importance of the theorem lies in the fact that the right-hand side of the equation (5) is a good candidate for parallel implementation. This term consists of two stages. In the first stage, function ϕ is applied in parallel on each processor (*map* functional). The second stage constructs the end result from the partial results in the processors by applying the *red* functional. Therefore, if a given problem can be expressed as a homomorphism instance then this problem can be solved in a standard manner as two consecutive parallel stages – map and reduction.

The standard two-stage implementation (5) may be time-optimal, but only under an assumption that makes it impractical: the required number of processors must grow linearly with the size of the data. A more practical approach is to consider a bounded number p of processors, with a data block assigned to each of them. Let $[\alpha]_p$ denote the type of lists of length p, and subscript functions defined on such lists with p, for example, map_p. The partitioning of an arbitrary list into p sublists, called *blocks*, is done by the *distribution function*, $dist\,(p) \; : \; [\alpha] \; \rightarrow \; [\,[\alpha]\,]_p$. The following obvious equality relates distribution to its inverse, flattening: $red\,(+\!\!+) \circ dist\,(p) \; = \; id$.

Theorem 2 (**Promotion**) *If h is a homomorphism w.r.t. ⊛, then*

$$h \circledast red\,(+\!\!+) \; = \; red\,(\circledast) \circ map\, h \tag{6}$$

This general result about homomorphisms is useful for parallelisation via data partitioning: from (6), a standard distributed implementation of a homomorphism h on p processors follows:

$$h \; = \; red\,(\circledast) \circ map_p\, h \circ dist\,(p) \tag{7}$$

Sometimes, it can be assumed that data is distributed in advance: either the distribution is taken care of by the operating system, or the distributed data are produced and consumed by other stages of a larger application.

The development of programs using skeletons differs fundamentally from the traditional process of parallel programming. Skeletons are amenable to formal transformation, that is, the rewriting of programs in the course of development, while ensuring preservation of the program's semantics. The transformational design process starts by formulating an initial version of the program in terms of the available set of skeletons. This initial version is usually relatively simple and its correctness is obvious, but its performance may be far from optimal. Program transformation rules are then applied to improve performance or other desirable properties of the program. The rules applied are semantics-preserving, guaranteeing the correctness of the improved program with respect to the initial version. Once rules for skeletons have been established (and proved by, say, induction), they can be used in different contexts of the skeletons' use without having to be reproved.

For example, in the SAT programming methodology [10] based on skeletons and collective operations of MPI, a program is a sequence of stages that are either a computation or a collective communication. The developer can estimate the impact of every single transformation on the target program's performance. The approach is based on reasoning about how individual stages can be composed into a complete program, with the ultimate goal of systematically finding the best composition. The following example of a transformation rule from [8] is expressed in a simplified C+MPI notation. It states that, if binary operators ⊗ and ⊕ are associative and ⊗ distributes over ⊕, then the following transformation of a composition of the collective operations scan and reduction is applicable:

$$\left[\begin{array}{l}\texttt{MPI\_Scan (}\otimes\texttt{);}\\[4pt] \texttt{MPI\_Reduce (}\oplus\texttt{);}\end{array}\right. \Longrightarrow$$

```
Make_pair;
MPI_Reduce (f(⊗,⊕));
if my_pid==ROOT then Take_first;
```
(8)

Here, the functions `Make_pair` and `Take_first` implement simple data arrangements that are executed locally in the processes, that is, without interprocess communication. The binary operator $f(\otimes,\oplus)$ on the right-hand side is built using the operators from the left-hand side of the transformation.

Rule (8) and other, similar transformation rules have the following important properties: (1) they are formulated and proved formally as mathematical theorems; (2) they are parameterized in one or more operators, for example, $\oplus$ and $\otimes$, and are therefore usable for a wide variety of applications; (3) they are valid for all possible implementations of the collective operations involved; (4) they can be applied independently of the parallel target architecture.

Task- and Algorithm-Oriented Skeletons

In contrast to the data-parallel style discussed in the preceding section, there are many instances of skeletons in which the abstraction is best understood by reference to an encapsulated parallel control structure and/or algorithmic paradigm. These can be examined along a number of dimensions, including the linguistic framework within which the skeletons are embedded, the degree of flexibility and control provided through the API, the complexity of the underlying implementation framework and the range of intended target architectures.

While *map* is the simplest data-parallel skeleton, *farm* can be viewed as the simplest task-parallel skeleton. Indeed, in its most straightforward form, *farm* is effectively equivalent to *map*, calling for some operation to be applied independently to each component of a bulk data structure. More subtly, the use of *farm* often carries the implication that the execution cost of these applications is unpredictable and variable, and therefore that some form of dynamic scheduling will be appropriate – the programmer is providing a high-level, application-specific hint to assist the implementation. Typical *farm* implementations will employ centralized or decentralized master–worker approaches, with internal optimizations which try to find an appropriate number of workers and an appropriate granularity of task distribution (trading interaction overhead against load balance). From the programmer's perspective, all that must be provided is code for the operation to be applied to each task, and a source of tasks, which could be a data structure such as an array or a stream emerging from some other part of the program. The *bag-of-tasks* skeleton extends the simple *farm* with a facility for generating new tasks dynamically.

Pipeline skeletons capture the pattern in which a stream of data is processed by a sequence of "stages," with performance derived from parallelism both between stages, and where applicable, within stages. Even such a simple structure allows considerable flexibility in both API design and internal optimization. For example, the simplest pipeline specification might dictate that each item in the stream is processed by each stage, that each such operation produces exactly one result, and that all stages are stateless. For such a pipeline, the programmer is required to provide a sequential function corresponding to each stage. More flexible APIs may admit the possibility of stateful stages, of stages in which the relationship between inputs and outputs is no longer one-for-one (e.g., filter stages, source, and sink stages), of bypassing stages under certain circumstances, and even of controlled sharing of state between stages. From the implementation perspective, internal decisions include the selection of the number of implementing processes or threads, allocation of operations to processes or threads, whether statically or dynamically, and correct choice of synchronization mechanism.

The *divide-and-conquer* paradigm underpins many algorithms: the initial problem is divided into a number of sub-instances of the same problem to be solved recursively, with subsolutions finally combined to "conquer" the original problem. For the situation in which the sub-instances may be solved independently the opportunities for parallelism are clear. Within this context, there is considerable scope to explore a skeleton design space in which constraints in the API are traded against performance optimizations within the implementation. A very generic API would simply require the programmer to provide operations for "divide" and "conquer," together with a test determining whether an instance should be solved recursively, and a direct solution method for those instances failing this test. Less-flexible APIs, could, for example, require the degree of division to be fixed (every call of divide returns the same number of sub problems), the depth of recursion to be uniform across all branches of divide tree, or even for some aspects of the "divide" or "conquer" operations to be structurally constrained. Each such constraint provides useful information, which can be exploited within the implementation.

Wavefront computations are to be found at the core of a diverse set of applications, ranging from numerical solvers in particle physics to dynamic programming approaches in bioinformatics. From the application perspective, a multidimensional table is computed from initial boundary information and a simple stencil that defines each entry as a function of neighboring entries. Constraints on the form of the stencil allow the table to be generated with a "wavefront" of computations, flowing from the known entries to the unknown, with consequent scope for parallelism. A *wavefront* skeleton may require the programmer to describe the stencil and the operation performed within it, leaving the implementation to determine and schedule an appropriate level of parallelism. As with *divide-and-conquer*, specific *wavefront* skeleton APIs may impose further constraints on the form of these components.

Branch-and-bound is an algorithmic technique for searching optimization spaces, with built-in pruning heuristics, and scope for parallelization of the search. Efficient parallelization is difficult, since although results are ultimately deterministic, the success of pruning is highly sensitive to the order in which points in the space are examined. This makes it both promising and challenging to the skeleton designer. With respect to the API, a *branch-and-bound* skeleton can be characterized by small number of parameters, capturing operations for generating new points in the search space, bounding the value of such a point, comparing bounds, and determining whether a point corresponds to a solution.

Skeleton-Based Systems

Past and ongoing research projects have embedded selections of the above skeletons into a range of programming frameworks, targeting a range of platforms. The P3L language [15] was a notable early example in which skeletons became first-class language constructs, together with sequential "skeletons" for iteration and composition. Subsequent projects within the Pisa group have taken similar skeletons into object-oriented contexts, with a focus on distributed and Grid implementations. In contrast, Muesli [3] allows data-parallel skeletons to be composed and called from within a layer which itself composes task-parallel skeletons, all within a C++ template-based library. Several systems have taken a functional approach. Hermann's HDC [12] focuses exclusively on a collection of divide-and-conquer skeletons within Haskell, and implemented on top of MPI, while the Eden skeleton library [14] and related work [11] have implemented skeletons on top of both implicitly and explicitly parallel functional languages. The COPS project [2] presents a layered interface in which expert programmers can also have access to the implementation of the provided "templates," all embedded in Java with both threaded and distributed RMI-based implementations. The SkeTo project offers a BMF-based collection of data-parallel skeletons for lists, matrices, and trees, implemented in C with MPI. Domain-specific skeletons are represented within the Mallba project [1] (focusing on combinatorial search) and Skipper and QUAFF projects (image processing). Higher-Order Components (HOCs) [5], Enhance [19], and Aspara [6] extend the idea of skeletons toward the area of distributed computing and Grids. HOCs implement generic parallel and distributed processing patterns, together with the required middleware support and are offered to the user via a high-level service interface. Users only have to provide the application-specific pieces of their programs as parameters, while low-level implementation details such as transfer of data across the grid are handled by the HOC implementations. HOCs have become an optional extension of the popular Globus middleware for Grids.

Skeleton principles are also evident in a number of other emerging parallel programming tools. Most notably, the MapReduce paradigm (related to, but distinct from the similarly named BMF skeletons) represents a pattern common to many applications in Google's infrastructure. Emphasis in the original implementation was placed on load balancing and fault-tolerance within an unreliable massively parallel computational resource, a task strongly facilitated by the structurally constraining nature of the skeleton. Thain's Cloud Computing Abstractions [18] implement a range of distributed patterns with applications in Biometrics and Genomics. As with MapReduce, these are trivial to implement sequentially, and relatively straightforward on a reliable, homogeneous parallel architecture. The strength of the approach becomes apparent when ported to less predictable (or reliable) targets, with no additional effort on the part of the programmer. Finally, skeletal approaches can also be discerned in MPI's collective operations [9], where `MPI_Reduce` and `MPI_Scan` are parameterized by operations as well as data, and Intel's Threading Building Blocks [17], which in particular includes a *pipeline* skeleton.

Related Entries

►Collective Communication
►Eden
►Glasgow Parallel Haskell (GpH)
►NESL
►Reduce and Scan
►Scan for Distributed Memory, Message-Passing Systems

Bibliographic Notes and Further Reading

The term "algorithmic skeleton" was originally introduced by Cole [4]. A considerable body of work now exists. Helpful snapshots can be obtained by consulting the 2003 book edited by Rabhi and Gorlatch [16], the September 2006 special edition of the journal Parallel Computing [13], and the recent survey by Gonzalez-Velez and Leyton [7].

Bibliography

1. Alba E, Almeida F, Blesa MJ, Cabeza J, Cotta C, Díaz M, Dorta I, Gabarró J, León C, Luna J, Moreno LM, Pablos C, Petit J, Rojas A, Xhafa F (2002) MALLBA: a library of skeletons for combinatorial optimisation. Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Springer, London, pp 927–932
2. Anvik J, Schaeffer J, Szafron D, Tan K (2003) Why not use a pattern-based parallel programming system? Euro-Par, Klagenfurt Austria, pp 81–86
3. Ciechanowicz P, Poldner M, Kuchen H (2009) The Münster skeleton library muesli – a comprehensive overview. Technical report, University of Münster
4. Cole M (1989) Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge
5. Dünnweber J, Gorlatch S (2009) Higher-order components for grid programming: making grids more usable. Springer, New York
6. Gonzalez-Velez H, Cole M (2010) Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms. Concurrency Comput Pract Exp 22(15):2073–2094
7. Gonzalez-Velez H, Leyton M (2010) A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software Pract Exp 40:1135–1160
8. Gorlatch S (2000) Towards formally-based design of message passing programs. IEEE Trans Softw Eng 26(3):276–288
9. Gorlatch S (2004) Send-receive considered harmful: Myths and realities of message passing. ACM TOPLAS 26(1):47–56
10. Gorlatch S, Lengauer C (2000) Abstraction and performance in the design of parallel programs: an overview of the SAT approach. Acta Inf 36(9/10):761–803
11. Hammond K, Berthold J, Loogen R (2003) Automatic skeletons in template Haskell. Parallel Process Lett 13(3):413–424
12. Herrmann CA, Lengauer C (2000) HDC: a higher-order language for divide-and-conquer. Parallel Process Lett 10(2/3):239–250
13. Kuchen H, Cole M (eds) (2006) Algorithmic skeletons. Parallel Comput 32(7/8):604–615
14. Loogen R, Ortega Y, Pena R, Priebe S, Rubio F (2003) Parallelism abstractions in Eden. In: Rabhi F, Gorlatch S (eds) Patterns and skeletons for parallel and distributed computing. Springer, London, pp 95–128
15. Pelagatti S (1997) Structured development of parallel programs. Taylor & Francis, London
16. Rabhi FA, Gorlatch S (eds) Patterns and skeletons for parallel and distributed computing. Springer, London, ISBN 1-85233-506-8
17. Reinders J (2007) Intel threading building blocks. O'Reilly, Sebastopol CA
18. Thain D, Moretti C (2009) Abstractions for cloud computing with condor. In: Ahson S, Ilyas M (eds) Cloud computing and software services. CRC Press, Boca Raton
19. Yaikhom G, Cole M, Gilmore S, Hillston J (2007) A structural approach for modelling performance of systems using skeletons. Electr Notes Theor Comput Sci 190(3):167–183

Parallel Tools Platform

Gregory R. Watson
IBM, Yorktown Heights, NY, USA

Synonyms

PTP

Definition

The Parallel Tools Platform (PTP) is an integrated development environment (IDE) for developing parallel programs using the C, C++, Fortran, and Unified Parallel C (UPC) languages. PTP builds on the Eclipse platform