# Structured Parallel Programming with Deterministic Patterns

Michael D. McCool, Intel, michael.mccool@intel.com

*Many-core processors target improved computational performance by making available various forms of architectural parallelism, including but not limited to multiple cores and vector instructions. However, approaches to parallel programming based on targeting these low-level parallel mechanisms directly leads to overly complex, non-portable, and often unscalable and unreliable code.*

*A more structured approach to designing and implementing parallel algorithms is useful to reduce the complexity of developing software for such processors, and is particularly relevant for many-core processors with a large amount of parallelism and multiple parallelism mechanisms. In particular, efficient and reliable parallel programs can be designed around the composition of deterministic algorithmic skeletons, or patterns. While improving the productivity of experts, specific patterns and fused combinations of patterns can also guide relatively inexperienced users to developing efficient algorithm implementations that have good scalability.*

*The approach to parallelism described in this document includes both collective "data-parallel" patterns such as map and reduce as well as structured "task-parallel" patterns such as pipelining and superscalar task graphs. The structured pattern based approach, like data-parallel models, addresses issues of both data access and parallel task distribution in a common framework. Optimization of data access is important for both many-core processors with shared memory systems and accelerators with their own memories not directly attached to the host processor.*

*A catalog of useful structured serial and parallel patterns will be presented. Serial patterns are presented because structured parallel programming can be considered an extension of structured control flow in serial programming. We will emphasize deterministic patterns in order to support the development of systems that automatically avoid unsafe race conditions and deadlock.*

*Keywords*—Deterministic parallel computing, patterns, software engineering, structured programming, many-core computing.

## I. Introduction

Parallel programming is challenging for a number of reasons. In addition to all the challenges of serial computation, parallel programs can also suffer from race conditions and deadlock, and even if correct may be non-deterministic, which complicates testing. Achieving high performance with a parallel program also requires minimization of contention for limited resources such as communication and memory bandwidth. Failure to properly account for these factors can lead to programs which dramatically underperform.

This document discusses and advocates a structured approach to parallel programming. This approach is based on a core set of common and composable patterns of parallel computation and data management with an emphasis on determinism and scalability. By using these patterns and also paying attention to a small number of factors in algorithm design (such as data locality), programs built using this approach have the potential to perform and scale well on a variety of different parallel computer architectures.

The structured approach discussed here has also been called the *algorithmic skeleton* approach. The general idea is that specific combinations of computation and data access recur in many different algorithms. A system that supports the specification and composition of "good" patterns can guide the developer towards the construction of well-structured and reliable, yet high-performance programs. Patterns can be domain-specific, but there are also general-purpose patterns that occur in algorithms from many different domains. A system only has to support a small number of patterns in order to be universal, that is, capable of implementing any algorithm. However, it may be useful for efficiency to support more than the minimal universal subset, since different patterns can also expose different types of data coherency that can be used to optimize performance. It may also be useful to support specific combinations of "fused" patterns.

We will first survey previous work in parallel patterns, algorithmic skeletons, and some systems based on these. The use of patterns in parallel programming bears a strong resemblance to the use of structured control flow in serial programming. Both for reasons of analogy and because serial computation is an important sub-component of parallel computation, some basic patterns for supporting serial computation will be presented and discussed, along with some serial programming models based on universal subsets of these patterns. A useful set of structured and deterministic parallel patterns will then be presented and discussed.

## II. Background

The concept of "algorithmic skeleton" was introduced by Cole [1989,2004] and elaborated by Skillicorn [1998]. It is similar to the modern idea of design pattern [Gamma 1994, Mattson 2004], and so we will use the term "parallel pattern". We will define a *parallel pattern* as specific recurring configuration of computation and data access. In the *View from Berkeley* [Asanovic 2006] some characteristic workloads called *dwarves* or *motifs* are identified. These are workloads that consist primarily of one type of pattern. In most applications, however, a variety of patterns are composed in complicated ways. Programming systems can be based entirely on composition of pattern-oriented operators [Bromling 2002, Tan 2003, Sérot 2002].

In the 1970s, it was noted that *serial* programs could be made easier to understand and reason about if they were built by composing their control flow out of only a small number of specific control flow patterns: sequence, selection, and iteration (and/or recursion). This *structured programming* approach has led to the elimination of goto from most programs, although not without a lot of controversy [Dijkstra1968]. Now, however, the use of structured control flow is so widely accepted that goto is either omitted from or deprecated in most modern programming languages.

In the same way, structured parallel patterns can eliminate the need for explicit threading and synchronization while making programs easier to understand. In particular, one desirable property that *structured* parallel patterns should possess is deterministic semantics that are consistent with a specific serial ordering of the program. In contrast, threads share a strong resemblance to goto in their flexibility but also their lack of structure [Lee 2006]. Like goto, use of threads (at least when combined with global random access to data) can make it difficult to do local reasoning about a program, since it becomes impossible to isolate the effect of one part of a program's code from another.

One alternative is to use functional programming. However, most mainstream programming languages are not functional, and some algorithms, especially graph and matrix problems, are difficult to express efficiently in purely functional languages. However, we can interpret functional programming as one instance of our structured parallel programming, just using a restricted set of patterns.

There is also a large class of collective languages that express parallelism explicitly through operations over entire collections of data. Collective languages can be imperative or functional, and collective operations are often available through libraries or as built-in operations in mainstream languages. NESL is an example of a collective pure functional language [Blelloch 1990, 1993, 1996], while FORTRAN 2003 is an example of an imperative language with built-in collective operations. RapidMind [McCool 2006] and Ct are examples of imperative languages based on collective operations. Deterministic imperative collective languages can be given a consistent sequential interpretation that makes it straightforward to reason about their behavior.

We do not have space to discuss the implementation or specification of patterns in programming models in detail. However, patterns can be implemented or supported in a variety of ways: as conventions; using code generators [Herrington 2003]; supported explicitly in new language designs; or implemented in libraries. The "library" approach can be made as efficient as a compiled language if dynamic code generation is used to support optimization and fusion [McCool 2002, McCool 2006].

In the following will first discuss serial patterns. This is important for three reasons. First, the serial semantics of a programming system needs to mesh well with the parallel semantics. Second, serial computation is still a large part of any program, and often a parallel implementation is derived from a serial implementation by a set of transformations (for example, turning loops into a set of parallel tasks over an index space). Third, studying patterns in a "familiar" space such as serial computation will lay a solid foundation for their extension to parallel patterns.

## III. Serial Control Flow Patterns

Early serial programming languages had a similar structure to the underlying machine language control flow mechanisms, with commands being executed in sequence but with the ability to jump or *goto* a different point in the sequence. It was soon realized, however, that indiscriminate use of a *goto* led to unreadable and unmaintainable programs. Structured programming limited control flow constructs to a small, composable set with desirable properties. Structured control flow constructs also make it possible to assign coordinates and locally reason about a program [Dijkstra1968].

Not all of the serial patterns described in this section are actually needed to allow universal computation. Two common universal subsets of the following patterns lead to the classes of imperative and functional programming languages. In general, either recursion or iteration can be chosen (but both are not required), and likewise for dynamic memory allocation and random write.

### A. Sequence

Two tasks are considered to be in sequence if the execution of the second task may only begin once all operations in the first task have completed, including all memory updates.

### B. Selection

The selection pattern executes one of two tasks based on the result of a Boolean-valued condition expression (whose evaluation constitutes a third task).

### C. Iteration

The iteration pattern repeats a task (its "loop body") until some condition is met. Every invocation of the loop body is finished before the next one is started, as if they were in sequence. Evaluation of the condition is also executed in sequence with the loop body.

Memory locations that are both read and written by a loop body are called loop-carried dependencies. While a loop-carried dependency is often used to compute an index, the loop index can also be considered to be a natural part of the looping pattern itself. Many loops can be parallelized even if they have loop-carried dependencies.

### D. Functions and Recursion

Sequences of operations can be stored in functions which can then be invoked repeatedly. Functions are parameterized by input data that is used for a specific activation of the function, and have their own local state. Functions compute values and return them. A *pure function* cannot modify its inputs or cause other side effects, such as modification of memory.

Functions can have their own local storage. If fresh locations for this local storage are allocated with every activation of the function, then it is possible for a function to be called recursively. Such recursive functions lead to a divide-and-conquer algorithmic pattern.

## IV. Serial Data Management Patterns

The random read and write data access pattern maps directly onto underlying hardware mechanisms. Stack and dynamic memory (heap) allocation are common higher-level patterns.

### A. Random Access Read

Given an index a random access read retrieves the value associated with that index. Arrays are the simplest form of randomly-accessible memory, but all other data structures can be built on top of them: a physical computer's RAM is nothing more than a large 1D array of fixed-length integers.

Indices can be computed and can be stored in other memory elements. The former property allows for the construction of higher-level patterns such as stacks while the ability to store indices in memory allows for complex data structures based on pointers (which are just abstractions for indices).

### B. Stack Allocation

A stack supports a last-in first-out (LIFO) model of memory allocation, which is often useful in nested function calls for supporting allocation of fresh copies of local state. Stack allocation has excellent coherency properties both in space and time.

Stack allocation can take place on serially nested activations of functions for local variables.

### C. Dynamic Memory (Heap) Allocation

When the LIFO model of memory allocation supported by the stack is not suitable, a more general model of memory allocation can be used that allocates a fresh memory location whenever requested.

### D. Collections and Data Abstraction

In addition to simple arrays, collection abstractions can include nested arrays and databases. A nested array can hold a collection of other arrays inside it. The subarrays can be of a fixed size or can vary in size. The former type of nested array we will call a *partitioned* array; the latter we will call a *segmented* array.

One-dimensional segmented arrays and operations on them can be implemented efficiently using auxiliary index and flag arrays [Blelloch 1990]. Likewise, partitioned arrays can be represented and operated on efficiently using a packed representation for the data itself and auxiliary data structures to track the partition boundaries.

A *database* maintains a set of elements, and supports search operations. Specifically, elements can be found by partial matches based on their content.

## V. Serial Programming Models

We can now describe the two most common serial programming models in terms of these patterns.

### A. Imperative Programming

In the imperative model of computation, the programmer directly tells the computer what to do and the order in which to do it. Serial imperative programming models, in order to be universal, need at a minimum to support the sequence, selection, and iteration control-flow patterns and typically the random-read and random-write patterns.

In addition, recursion and functions are usually supported, although they are technically not needed. FORTRAN, in particular, only relatively recently added support for recursion.

Imperative programming is the dominant serial programming model today. Its chief disadvantage from the point of view of parallelization is its over-specification of the ordering of operations. It is difficult to determine automatically, given an imperative program, which ordering constraints are essential for the correct operation of the program and which are an accidental result of the way the programmer expressed the computation. In particular, since pointers can refer anywhere in the global array, the global memory array is a potential source and destination for all operations, making it a universal data dependency.

### B. Functional Programming

The functional model of computation is based on rewriting nested trees or graphs. Pure functional languages typically only support selection and recursion for control flow, and random read for data access. Data structures can be created (using dynamic memory allocation) but not modified. Despite their simplicity, pure functional languages are universal. Some algorithms that depend on incremental modification of data in-place are however difficult to express in purely functional languages.

The chief advantage of functional languages from a parallelization point of view is that only the essential data dependencies are expressed and only these data dependencies constrain the order of operations.

## VI. Parallel Computation Patterns

We will now introduce a collection of parallel computation patterns. We have divided parallel patterns into two categories: computational patterns, which can actually operate on data values, and data access patterns, which cannot. These are often combined, and many of the computational patterns in this section also access and update data in specific (and typically coherent) ways.

### A. Map

The map parallel computation pattern applies a function to every element of a collection (or set of collections with the same shape), and creates a new collection (or set of collections) with the results from the function invocations. The order of execution of the function invocations is not specified, which allows for parallel execution. If the functions are pure functions with no side effects, then the map operation is deterministic while succinctly allowing the specification of a large amount of parallelism. In general, the (pure) functions used by the map can also recursively support other kinds of serial and parallel patterns and data management.

The map operation accesses data for input and output in a way that exposes useful spatial coherence. Many functions are executed at once, and it is known in advance which functions access neighboring values in the input and output collections. This makes it possible to automatically implement a variety of serial, parallel, and memory optimizations in the implementation of the map function, including software pipelining, cache prefetch and eviction, and cache boundary alignment. If the behavior of neighboring elements in a map can be assumed to lead to similar control flow behavior, then some simple approaches to vectorization based on masking can also be effective.

### B. Reduction

A reduction applies a pairwise associative operation to all the elements of a collection, reducing it to a single element.

Sometimes, when writing a function intended to be used in a map operation, it is desired to also compute a reduction at the same time. A good example is an iterative solver. The inner loop of such a solver usually performs both a matrix-

vector operation and a reduction, the latter being used to test convergence. In general, efficient implementations will need to fuse patterns together. There are other examples, such as *pack*, where fusion is even more important for performance.

Some other forms of reduction are sometimes used. These can be seen as fusions of pure reductions with other patterns. *Multidimensional reductions* (for example, reductions of the rows of an array) can be expressed by combining a partitioning pattern with a map and a reduction. In a *category reduction* an operator is applied that labels elements and then a reduction is applied to all elements with the same label. The Google map-reduce programming model is based a single fused map and category reduction operation combined with the serial execution patterns.

### C. Superscalar sequences

Sequence is a fundamental serial pattern. In the sequence pattern, one operation is completely finished before another one is started. However, when the operations are pure functions without side effects, the operations given in a sequence only need to be ordered by their data dependencies, which in the case of pure functions are made explicit.

In general a sequence generates a DAG (task graph) of data dependencies. A simple asynchronous execution rule allows for parallelism while still permitting serial reasoning by the programmer: if a task tries to read data that is not yet ready, it blocks until the input data *is* ready.

Although in this pattern the input code is conceptually serial, the data dependencies in the graph allow independent tasks to execute in parallel.

Under the superscalar model direct communication and synchronization between tasks using message passing is not permitted. In fact, tasks do not need to be simultaneously active, and their execution may in fact be serialized. Instead of unstructured low-level communication, two other structured patterns for sharing and communicating data between simultaneously active tasks can be used: the pipeline pattern and nested parallelism. The pipeline pattern allows for producer-consumer communication, while the nested parallelism pattern allows for child-parent communication.

### D. Pipeline

A pipeline is a set of simultaneously active tasks or "stages" that communicate in a producer-consumer relationship. A pipeline is not expressible as a superscalar task graph, since in a pipeline the data in a stage is persistent and stages are conceptually activated at the same time, unlike the tasks in a superscalar task graph. Pipelines are common in image and signal processing. Their model of local state update is a form of coherence not covered by other patterns. In addition, pipelines can be used to parallelize serially dependent activities ("folds") like compression and decompression.

Pipelines by themselves are not a complete solution to parallelization since pipelines tend to have a fixed number of stages. As such, they do not automatically scale to a large number of cores. However, pipelines can provide a useful multiplier on parallelism in otherwise difficult to parallelize problems.

### E. Nesting

Recursion is another fundamental serial control flow pattern. It is also associated with stack-based data allocation, which has good data coherence properties. When parallel patterns are nested recursively, they can be used to spawn additional parallel tasks. This allows a program to generate an arbitrary amount of nested parallelism. This form of nested parallelism is distinct from the form of nested parallelism that can be derived from segmented collective operations. However, it may be possible in many cases to identify certain patterns of more general recursive nested parallelism and map them into segmented operations for efficiency.

Nested parallelism can be invoked simply by invoking parallel patterns inside other parallel patterns, for example, by using a reduction or a map inside a function used inside another reduction or map. This generates a hierarchical task graph that can be expanded as needed to generate additional parallelism. The nested parallelism can be either task-parallel or data-parallel.

As a practical matter, arbitrary amounts of parallelism may not be useful. One of the advantages of deterministic parallel patterns is that they are all consistent with a specific serial ordering. An implementation needs to target a "grain size" that is most efficient for a given hardware target. Tasks that are too small need to be merged into larger serial tasks, while large serial tasks need to be decomposed into parallel tasks, preferably automatically. Serial consistency allows this to happen automatically without changing the result of the program.

### F. Scans and Recurrences

A recurrence expresses one output from a function in terms of prior outputs. Recurrences often occur in serial code due to the use of loop-carried dependencies, but in certain cases they can be parallelized. One-dimensional recurrences can be parallelized into logarithmic time implementations if the dependency is associative, in which case it is usually called a *scan* [Blelloch 1990]. Multidimensional recurrences with a nesting depth of $n$ can also always be parallelized over $n$-1 dimensions, even if the operator is not associative, using Lamport's hyperplane theorem [Lamport 1974].

A 1D recurrence, even if is not associative, is common and is often known as a *fold*. Folds will typically need to be implemented serially, although sequences of folds inside a map can be transformed into a parallel implementation using pipelines. As with reductions, there is a fundamental problem with identifying associative functions to allow parallelization in scans, as well as the problem of semi-associative operations such as floating point arithmetic.

Examples of recurrences include integration and infinite-impulse response (recursive) filters. Many matrix factorization algorithms, such as Chebyshev factorization, can also often be expressed as recurrences.

Scans (and, in general, recurrences) over segmented and partitioned collections can also be implemented efficiently in a load-balanced form even in the case of segmented arrays where the sub-arrays may not all be the same size. Using such balanced primitive operations, it is possible to implement, for example, a balanced parallel form of recursive and "irregular" algorithms such as quicksort.

## VII. Parallel Data Management Patterns

Data access and management patterns organize access to data but do not operate on the values themselves. Many combinations of specific data-access and computational patterns are common and may be considered patterns in their own right. This is because for efficient implementation it is often imperative for a data-access pattern to be fused with a specific parallel computational pattern.

### A. Gather

Given a collection of indices and an indexable collection, a *gather* generates an output collection by reading from all the locations given by the indices in parallel.

A random read is a serial pattern but when used from within a map it becomes a collective gather. In addition, a gather might be supported by an explicit collective operation.

### B. Search

The *search* pattern is like gather, but retrieves data from a "database" collection based on matching against content. Parallelism is achieved by searching for all keys in parallel, or by searching in different parts of the database in parallel.

### C. Subdivision

In parallel algorithms, we often want to divide the input into a number of pieces and then operate on each piece in parallel.

There are several possible variants of subdivision. The *partition* of a collection divides it into a nested collection of non-overlapping regions of the same size. The *segmentation* of a collection divides it into a segmented collection of non-overlapping regions of possibly different sizes. The *tiling* of a collection creates a collection of possibly overlapping references to regions within the larger collection.

### D. Stencil

A useful extension of map (which can also be seen as a regular variant of tiling followed by map) is the neighborhood stencil. In this pattern, regular spatial neighborhoods in an input array are operated on rather than only single elements. This is known as a finite convolution in signal processing, but this pattern also occurs in many simulation (PDE solvers) and matrix computations.

Some attention has to be paid to neighborhoods that extend past the edge of the array. Such accesses should be transformed (for example, by wrapping or clamping the index) so it maps to a well-defined value.

Implementing this pattern efficiently using low-level operations is surprisingly complicated, which argues for its inclusion as a basic pattern. It is useful to generate separate versions of the task for the interior of the input array and the boundaries. Also, a sliding window over partially overlapping regions of the input array may be useful.

However, for portability these optimizations can and should take place in the language implementation itself, since they vary by hardware target. Also, while induction variable analysis can and should be used to identify the stencil pattern whenever possible, the interface should also allow a straightforward and direct specification of the stencil.

### E. Scatter

Scatter writes data to a random location (given by an integer index) in a collection, overwriting any data that may already be stored at that location. Several varieties of scatter can be identified, depending on how collisions (parallel writes to the same location) are resolved.

A *priority scatter* allows writes into a collection of locations in an existing collection given a collection of data. Collisions (duplicate writes to the same location) are resolved using a deterministic rule.

The priority scatter operation is useful because the serial ordering of loop bodies can be used to generate the disambiguating rule. Loops with scatter operations (as long as they are not also loop-carried dependencies) can then be safely converted into priority scatters.

There are a number of ways to implement priority scatter efficiently. If it can be proven that no collisions are possible with other threads, then it can be implemented using ordinary serial semantics. For example, if the output of a map is a partition, it is only possible for each invocation of a function to scatter into its own output partition. Another case is when output data is allocated dynamically by a thread.

*Atomic scatter* is a non-deterministic pattern (the only one considered in this paper) that only guarantees that one result will survive in an output location when a collision occurs. Implementing atomic scatter is still potentially expensive if locking is necessary to preserve atomic writes.

A *permutation scatter* is a scatter that is only guaranteed to work correctly if there are *no* collisions. It can be typically be implemented efficiently in terms of an unsafe scatter. However, this means that it may produce incorrect results if incorrectly used with a set of write locations that *do* contain duplicate address values, so a debug mode should be provided that checks for such incorrect usage.

A *merge scatter* uses an associative operator to combine elements when a collision occurs. This rule can also be used to combine scattered values with existing data in an array. An example of this is the computation of a histogram.

### F. Pack

The *pack* pattern is used to eliminate wasted space in a sparse collection and to handle variable-rate output from a map. From within map, each function activation is allowed to either keep or discard its outputs. The survivors are then packed together into a single collection. A variant of this is the *expand* pattern that can output zero or more values. A standalone pack collective operation is not as useful as one that can be fused with map, since the latter does not need to allocate memory for data that is to be discarded.

## VIII. Conclusion

Deterministic parallel programs can be built from the bottom up by composing deterministic parallel patterns of computation and data access. However, an implementation of a programming model based on these patterns must not only support a sufficiently wide variety of patterns, it also needs to be able to control the granularity of their execution, expanding and fusing them as needed.

## REFERENCES


**[Aldinucci 2007]** M. Aldinucci and M. Danelutto, *Skeleton-based parallel programming: Functional and parallel semantics in a single shot*, Comput. Lang. Syst. Struct., 33(3-4), 2007, pp. 179-192.

**[Asanovic 2006]** K. Asanovic et al, *The Landscape of Parallel Computing Research: A View from Berkeley*, EECS Department, University of California, Berkeley EECS-2006-183, 2006.

**[Blelloch 1993]** G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein and M. Zagha, *Implementation of a portable nested data-parallel language*, PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, 1993, pp. 102-111

**[Blelloch 1996]** G. E. Blelloch, *Programming parallel algorithms*, Commun. ACM, 39(3), 1996, pp. 85-97.

**[Blelloch 1990]** G. E. Blelloch, *Vector models for data-parallel computing*, MIT Press, 1990.

**[Bosch 1998]** J. Bosch. *Design patterns as language constructs*. Journal of Object-Oriented Programming, 11(2):18-32, 1998.

**[Bromling 2002]** S. Bromling, S. MacDonald, J. Anvik, J. Schaefer, D. Szafron, K. Tan, *Pattern-based parallel programming*, Proceedings of the International Conference on Parallel Programming (ICPP'2002), August 2002, Vancouver Canada, pp. 257-265.

**[Buck 2007]** I. Buck, *GPU Computing: Programming a Massively Parallel Processor*, Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, March 11-14, 2007.

**[Cole 1989]** M. Cole. Algorithmic Skeletons: Structured Management of Parallel Computation, Pitman/MIT Press, 1989.

**[Cole 2004]** M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing, 30(3), pp. 389-406, March 2004

**[Czarnecki 2000]** K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, 2000, ACM Press/Addison-Wesley.

**[Darlington 1995]** J. Darlington, Y. Guo, H. W. To and J. Yang, *Parallel skeletons for structured composition*, SIGPLAN Not., 30(8), 1995, pp.19-28.

**[Dijkstra 1968]** E. Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM, 11 (3), March 1968, 147-148.

**[Dorta 2006]** A. Dorta, P. López and F. de Sande, *Basic skeletons in 11c*, Parallel Computing, 32(7), pp. 491-506, September 2006.

**[Gamma 1994]** E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

**[Gorlatch 1999]** S. Gorlatch, C. Wedler and C. Lengauer, *Optimization Rules for Programming with Collective Operations*, Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, pp. 492-499, April 12-16, 1999.

**[Herrington 2003]** J. Herrington, *Code Generation in Action*, 2003, Manning Publications.

**[Kessler 2004]** C. Kessler, *A practical access to the theory of parallel algorithms*, ACM SIGCSE Bulletin, 36(1), March 2004

**[Klusik 2000]** U. Klusik, R. Loogen, S. Priebe, F. Rubio, *Implementation Skeletons in Eden: Low-Effort Parallel Programming*, Selected Papers from the 12th International Workshop on Implementation of Functional Languages, pp.71-88, September 2000.

**[Lamport 1974]** L. Lamport, *The Parallel Execution of DO Loops*, Communications of the ACM 17(2), February 1974, pp. 83-93.

**[Lee 1996]** P. Lee and M. Leone, *Optimizing ML with run-time code generation*, Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, p.137-148, May 1996.

**[Lee 2006]** E. A. Lee, *The Problem with Threads*, IEEE Computer, 39(5), May 2006, pp. 33-42.

**[MacDonald 2002]** S. MacDonald, J. Anvik, S. Bromling, D. Szafron, J. Schaeffer and K. Tan. *From patterns to frameworks to parallel programs*, Parallel Computing, 28(12);1663-1683, 2002.

**[Massingill 1999]** M. Massingill, T. Mattson, and B. Sanders. *A pattern language for parallel application programs*. Technical Report CISE TR 99-022, University of Florida, 1999.

**[Mattson 2004]** T. Mattson, B. Sanders, B. Massingill, *Patterns for Parallel Programming*, 2004, Pearson Education.

**[McCool 2002]** M. McCool, Z. Qin, and T. Popa, *Shader metaprogramming*, HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2002, pp. 57-68.

**[McCool 2004]** M. McCool, S. Du Toit, T. Popa, B. Chan and K. Moule, *Shader algebra*, ACM Trans. Graph., 23(3), 2004, pp. 787-795.

**[McCool 2004]** M. McCool and S. Du Toit, *Metaprogramming GPUs with Sh*, 2004, AK Peters.

**[McCool 2006]** M. McCool. Data-Parallel Programming on the Cell BE and the GPU Using the RapidMind Development Platform. GSPx Multicore Applications Conference, 9 pages, 2006.

**[Pelagatti 1998]** S. Pelagatti, *Structured development of parallel programs*, Taylor & Francis, Inc., Bristol, PA, 1998.

**[Owens 2005]** J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A. E. Lefohn, and T. J. Purcell, *A survey of general-purpose computation on graphics hardware*, Eurographics 2005, State of Art Report.

**[Schmidt 2000]** D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley & Sons, 2000.

**[Sérot 2002]** J. Sérot, D. Ginhac, *Skeletons for parallel image processing: an overview of the SKIPPER project*, Parallel Computing, 28(12), pp.1685-1708, December 2002.

**[Siu 1996]** S. Siu, M. De Simone, D. Goswami, and A. Singh. *Design patterns for parallel programming*. Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), pp. 230–240, 1996.

**[Skillicorn 1998]** D. B. Skillicorn and D. Talia, *Models and languages for parallel computation*, ACM Comput. Surv., 30(2), 1998, pp.123-169.

**[Tan 2003]** K. Tan, D. Szafron, J. Schaeffer, J. Anvik and S. MacDonald, *Using generative design patterns to generate parallel code for a distributed memory environment*, PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, 2003, pp 203-215.

**[Veldhuizen 1999]** T. L. Veldhuizen, *C++ templates as partial evaluation*, In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 1999.