International Journal of Grid and Distributed Computing
Vol. 9, No. 11 (2016), pp.143-156
http://dx.doi.org/10.14257/ijgdc.2016.9.11.12

A Survey of Loop Parallelization: Models, Approaches, and Recent Developments

Hong Yao[1,*], Huifang Deng[2] and Caifeng Zou[3]

[1,2,3]*School of Computer Science and Engineering, South China University of Technology, Guangzhou, 510006, China*
[1,*]*hong_yao@126.com* , [2]*2465372809@qq.com* , [3]*caifengzou@gmail.com*

Abstract

In cloud computing era, automatic parallelization is still significant for virtualization platform. However, after several decades of development, the overall effect is still to be improved. Summary of the mainstream technology developments will be beneficial to reveal the future direction and trend. This paper reviews the technology of loop parallelization, which is the key issue in automatic parallelization. After introducing the basic models and approaches, we focus on the recent developments, on which we obtain the trend of this field and the conclusions about future.

Keywords*: Automatic parallelization, Loop parallelization, Loop transformation, Data dependence, Polyhedral model, Program dependence graph.*

1. Introduction

Despite decades of development, the automatic parallelization remains to be an extremely challenging field. Meanwhile, with the development of cloud computing, automatic parallelization is especially suitable for being taken as an underlying technique to maintain transparent and efficient computing. Whereas, the key work of automatic parallelization is loop parallelization, which can enable sequential loop program to be executed in parallel. The main process include data dependence analysis, loop transformation and code generation or dynamic scheduling [1, 2].

Early literature mainly focuses on the loop transformation itself, and builds the data dependence relations and boundary constraints as Polyhedral model or PDG (Program dependence graph). Early surveys are mainly about the collection and classification of related approaches. There are several surveys about early developments [1-4], which demonstrate the basic framework of loop parallelization and introduce the main models and approaches. Banerjee *et al*. [1] described a basic framework, which includes data and control dependence analysis, loop transformation, dynamic scheduling, dependence breaking, locality optimization, overhead reduction, and effectiveness (verification). These issues are still been concerned widely today. Boulet *et al*. [2] summarized the main static parallelization techniques, which include dependence analysis and loop transformation. The static analysis includes dependence analysis and loop transformation, which are used in conjunction with code generation. Except for these, perfect and unperfect loop are distinguished by in [2]. However, static analysis can not obtain all of dependence due to some information is unknown in the compile time. For the deficiency of static analysis, dynamic analysis can be a good supplement, which is discussed more by Gupta *et al*. [3]. Dynamic analysis must relate to detailed running models on some architecture and involve the parallelism of instruction level, shared memory, and distributed memory. Bacon *et al*. [4] described a comprehensive survey, which pays special attention to the compilation and evaluation in addition to the previous issues.

Recent literature considers and integrates more techniques such as dynamic schedule, intelligent analysis, and the collaborative work of three aspects: computing, storage, and

ISSN: 2005-4262 IJGDC

communication. On the other hand, the integration with a variety of architectures is the ongoing work. Recent surveys can pay more attention to the abstract patterns and characteristics of approaches, which can support a higher level of classification [5].

Based on reviewing the mainstream models and related approaches, this paper introduces the developments of this field. The following sections are arranged as below. Section 2 demonstrates polyhedral model and loop transformation; section 3 is about graph theory based models. Section 4 focuses on the models based on semantics directives, or stencils, or some other high level abstracts; Section 5 discusses the dynamic approaches of run time. Section 6 introduces the application of intelligent algorithms for this area. The discussion of future trend and conclusions are included in section 8.

2. Polyhedral Model and Affine Transformations

The polyhedral model originates from the constraints of the nested loop bounds or from its generators (vertices and rays) [44]. Let the lower and upper bounds of a nested loop are $L=(L_1,\cdots,L_n)$, and $U=(U_1,\cdots,U_n)$ respectively, then for $i=1,\cdots,n$, any iteration vector $I=(I_1,\cdots,I_n)$ need to satisfy $L_i \le I_i \le U_i$.

An inner loop nest bound can be a function of the outer nest bounds. For example, we can let $L_i = f_i(L_1,\cdots,L_{i-1}), U_i = g_i(U_1,\cdots,U_{i-1})$, for $i=1,\cdots,n$. In addition to bound constraints, some other inequalities may be induced by data dependences, such as a read at $J=(J_1,\cdots,J_n)$ after a write at $I=(I_1,\cdots,I_n)$ and both of them access a same variable, then $I<J$ is a constraint with respect to the sequential execution order when a loop transformation is performed. In most cases, these functions and expressions within the constraints are affine. These affine inequalities form the polyhedral model.

2.1 Data Dependence Analysis

A data dependence is induced by two adjacent accesses to a same memory variable, and at least one of them is a write [4, 6]. For convenience, we call an execution of a statement as a statement instance. In a nested loop, it is equivalent to an iteration of a statement. The basic issue of loop parallelization is restructuring the statement instances and allocating them into some execution units.

However, different accesses to a same memory variable requires that the statement instances should be allocated to a same processor, or a synchronization is needed between different processors. So the data dependence analysis is the foundation of the whole parallelization process. However, it is not an easy problem to determine whether any two statement instances are dependent because it involves to the problem of integer program, so the mainstream approaches usually obtain the approximation solutions by testing [7-18].

The most classical methods are GCD test and Banerjee test [7] which are based on two basic theories respectively: number theory and inequality system of constraints. λ-test [8] extends Banerjee test for coupled dimensions. I-test [9] converts the constraints into equation system of integer adjacent intervals and each time solves one of interval equation.

However, the original I-test can only obtain the existence of dependence, whereas DVI test [10] adds direction vector constraints to the interval equations based on the I test, which results in more exact solutions. GDVI test [11] extends DVI test by adding variable bound constraints. PVI test [12] focuses on the polynomial variable interval, which makes great progress from GCD and Banerjee test. Power test [13]

integrates extended GCD algorithm and Fourier-Motzkin algorithm, which can eliminate the variables within a loop bound constraint system. It takes into account all of constraints, direction vectors, and some other unknown variables. Omega test [14] is based on the remainder algorithm and FMVE (Fourier-Motzkin variables elimination). Although the worst time complexity is exponential degree, it has polynomial time complexity in most of real time. Omega test can obtain the exact dependence direction and distance vectors. IR test [15] employs repeated projections to reduce the interval to find the solution or till the interval is reduced to an empty set. It is suitable for any direction vector and triangle boundary constraints.

With the development of the runtime scheduling approaches, dynamic data dependence test are also emerging, such as LRPD test [16], which employs speculative approach to execute threads in test mode, and then obtains statistics at run time, which are used to modify the old parallelism.

In recent years, some tests for nonlinear data dependence have been proposed, which greatly expands the ability of data dependence analysis. Range test [17] is implemented in a library named PLATO, which has been ported into a compiler named Polaris. It uses bound functions to define interval for each constraint and employs variable interval theory to solve dependence problem. NLVI test [18] is another nonlinear variable interval test, which can effectively employ direction vector information, bound functions, and monotonicity.

2.3. Loop Transformation

The loop transformation means changing the nested loop structure or form but keeping the original semantics to decouple some data dependences for parallel execution. Bacon *et al*. [4] summarized a lot of early transformations, such as loop un-switching, loop reordering, loop interchange, loop skewing, loop reversal, loop tiling, loop distribution (splitting), loop fusion, loop restructuring, loop unrolling etc, and also discussed many issues about early parallelization in software and hardware.

Loop fusion is an important transformation, which involves the merger of two or more loops. This transformation is not only help for parallelization, but also contributes benefits to other aspects, such as reducing loop bound testing, promoting data locality etc [19]. Manjikian *et al*. [20] presented a typical approach, which integrates both parallelism and memory locality optimization. Darte [19] discussed the complexity of loop transformation, which is help for knowing the ultimate potentialities of this method.

The transformation of a single loop is usually modeled as a transformation matrix acting on the iteration vector. This transformation is classified as unimodular or non-unimodular according to the matrix types. The unimodular transformation is the basic transformation because the transformation matrix determinant can only be +1,0, and -1, while the regular unimodular matrix has only determinant value of +1 or -1. Furthermore, reducing a matrix to a triangular form need to use unimodular matrices [21], so the unimodular matrix is very important in matrix transformation.

For a nested loop, if each nested statement is located in the innermost loop nest, it is perfect loop. Xue [22] proposed a framework to convert some imperfect loops to perfect loops by unimodular transformations. The basic idea is to move the statements of outer loop nest to the inner nest and adds related loop index constraints to the statements. The transformed iteration space is still a union of convex polyhedra, which is the base of the proposed code generation method.

If the transformation matrix determinant >1, the transformation is non-unimodular. This transformation can produce some “holes” in the transformed iteration space. The iterations of statements at the holes need not to be executed. Ramanujam [23] focused the problem and gave a method to skip the holes by

changing the step size, which is determined by the transformation matrix, specifically, by the diagonal elements of the Hermite normal form of the matrix.

Lim *et al*. [24] proposed the concepts of parallelism degree and synchronization degree, which are similar to the complexity of algorithm, such as $O(n^k)$ parallel executed operations represent k degree of parallelism. The concept has great significance in both theory and practice. In theory, parallelism degree and synchronization degree can give a classification for algorithms, which results in more optimization of sub problems. In practice, the two type degrees can afford the mechanism to make full use of the potential of platforms. In [25], Lim el al. continue formulated the problem: for a given communication or synchronization degree constraints, how much parallelism can be achieved? Especially, a schema of synchronization free parallelism was developed and an algorithm which can combine many transformations was proposed.

The Λ-transformation [26] is for the singular loop, whose transformation matrix is singular. This framework can unite loop transformations of permutation, reversal, skewing and scaling. At last it generates code based on lattice theory.

Pouchet *et al* [27] focused on feedback-directed and iterative optimization of loop transformations, and built an optimization space of loop transformations which were equivalent to one-dimensional affine schedules, then discussed the reduction of the search space. The reduced space is small and amenable to fast-converging. This method is compatible with some other feedback-directed methods, such as machine-learning techniques.

2.4. Code Generation

Tilling can be taken as both an important transformation and a code generation technique, which is concerned by a large amount of literature [28, 30]. The classical definition of the "tile" should include three points [28]: 1) it is constrained by bounds; 2) it is identical by translation; and 3) it is an atomic unit. The bound constraints are obvious. The identity means it can be the image of a translation of any other tile, so, in a sense, tiling is also a code generation method because it can reproduce a basic pattern and have definite bound constraints. The atomic means that a tile is an independent scheduling unit and there are no synchronization points within a tile. Another restrict is a tile is for perfect loop and the dependence must be uniform. Since this definition is more general and abstract, it is generally able to represent most of the code generation methods in a polyhedral model.

Tiling can not only improve the data locality, but also reduce the communication, and even can achieve an overall optimization. Xue [29] proposed an communication optimization approach, which is based on arithmetic inequality and geometric means, i.e. the communication volume of each tile face, as basic variables in an inequality, must be equal. Griebl *et al* [30] summarized four categories of tiling approach: the first category is to minimize idle time but ignore communication cost; the second is only to minimize the volume of communication. The third is to reduce the synchronization; and the forth is to minimize execution time. Griebl *et al*. [30] also proposed a space-time mapping approach before tiling. The approach focuses on two basic functions in the parallelization: allocation and scheduling. In essence, the allocation is to map each statement instance to a spatial location (an identified processor or core), while the scheduling is to to map each of them to a time location (time step). This is space-time mapping. In the proposed approach, tiling space dimensions is realized by aggregating virtual processors to a physical processor, and tiling time dimensions is realized by message vectorization.

Tiling can be performed in different level. Multilevel means a tile can be divided into smaller tiles and each division is a level. This mechanism can adapt to different

shapes of polytope (closed convex polyhedra). Hartono *et al*. [31] proposed a more better method than old ones. The proposed method can process imperfect loops and even use the feedback information at run time to tune related parameters. Fine grained tiling can obtain more parallelism, such as Jimenez *et al*. [32] proposed a multidimensional tiling approach, which includes four phases: 1) iteration space tiling, 2) index set splitting, 3) unrolling phase, 4) scalar replacement. Among them, scalar replacement is to remove unnecessary load and store operations, which realizes the optimization of registers.

Tile size selection (TSS) is an important issue in tiling process, which can be set by hand-crafting or be automatically determined by the system. Yuki *et al*. [33] proposed an automatic setting model, which is based on program features, synthetic kernels, and machine learning techniques. The model can also dynamically be tuned to adapt to the up-to-date situation.

Previous discussion is about the definition, classifications, functions, objects and the process methods of tiling. However, data dependence system is the key issue to determine a tile or tiles. Therefore, another considerable optimization method is eliminating false dependences, which are usually induced by reusing some temporary variables with short live ranges [34]. The conventional way to eliminate the false dependences is variable privatization and expansion, which will lead to excessive memory consumption, because both privatization and expansion are reproducing variables, the distinguish is that privatization is for each thread and expansion is for each statement instance. Recently, Baghdadi *et al*. [35] proposed an approach to ignoring the false dependence without variable privatization and expansion. The main idea is to find the live range of temporary variables, and use the live range noninterference to relax the criterion for tiling.

Ancourt *et al*. [36] considered basic issues in code generation: to scan tiles of polyhedra, to scan iterations within a tile, and to scan array elements within a tile. Another is to determine the loop bounds for each dimension. Three algorithms were presented to obtain the bounds, which are general and can encompass most transformations on the nests of loops. In the framework, hyperplane partition is used to determine tiles, the techniques of projection to lower dimension, integer pairwise elimination and integer division are used to obtain bounds. Boulet *et al*. [37] proposed another scanning method, which can generate low level code, so it dose not need to use do-loops. This method can scan the image of transformed polyhedron, which may be a single polyhedron or unions of polyhedra. On the other hand, it can process multiple parameters which provides users with greater flexibility.

Clauss *et al*. [38] proposed a parametric polyhedral model, which is based on convex polytope, i.e. bounded convex polyhedron. The details is to compute the parametric vertices, the domains of a parametric polytope and the number of integer points within the domain. These parameters can be used for the estimate of execution cost, memory optimization, load balance and parallelization etc.

From the viewpoint of unified time and space, the scheduling is regarded as the mapping of time dimension, and the distribution is regarded as the mapping of spatial dimension. It takes the nested loop as index set [39], which is split into different parts such that each part has the regularity of dependences. Then, other scheduling algorithms can be applied for each parts.

Control flow may destroy the structure of original polyhedron in static analysis. Benabderrahmane *et al*. [40] proposed an approach to removing the control flow effect by employing prediction to replace the condition expression.

Code representation model has a great impact on the code generation. Polyhedral model can usually be used for both parallelization and memory locality optimization. The work of Bondhugula *et al*. [41] is a source to source transformation framework,

which can optimize parallelism and locality simultaneously. Many mainstream techniques are integrated into the framework including integer linear optimization, affine transformations, and tiling. The framework is implemented in a tool and can generate OpenMP parallel code. Grosser *et al*. [42] employed abstract syntax tree (AST) as representation of generation code or schedule. This AST has multiple advantages: it can support Presburger relations, piecewise schedules etc. Much information is reflected in the location and context of AST, which simplify the representation and the process. Furthermore, the related information is not only used for constructing loop bounds, but also for the user control. Some atomic options can be used to achieve more fine-grained control, such as the control of code size, the application of user-directives. Upadrasta *et al*. [43] proposed a sub-polyhedral scheduling algorithm, which is based on TVPI (Two Variables Per Inequality) polyhedra with form of constraints: $ax_i + bx_j \le c$, $a,b,c \in Q$. One of advantage of the polyhedra is they are closed under projection, which is suitable for some algorithms to generate them.

Code generation need to scan the polyhedron or polyhedra. Quillere *et al*. [44] focused on the problem of scanning multiple polyhedra, which induced by different statements in imperfect loop. In geometric viewpoint, the constraints can be viewed as hyperplane of the polyhedron. Unlike the Fourier-Motzkin method, this method only considers adjacent constraints, which eliminate most of redundant bounds. Bastoul [45] proposed an improved algorithm based on the state-of-the-art code generation algorithm, which can overcome the drawbacks of existing optimization algorithms, such as occupying more resources, and high costs of program control.

3. Graph Theory Based Models and Partitions

Ferrante *et al*. [46] introduced Program Dependence Graph (PDG), which includes data dependence subgraph and control dependence subgraph. In the PDG, statements, prediction expressions, operations and operands are taken as nodes, and the dependences are taken as edges. The PDG is used for detecting parallelism, code motion, loop fusion, and slicing etc.

Graph based models can be used for different levels of a program, whereas the basic mechanism is consistent. System dependence graph (SDG) is used for interprocedural analysis and it is defined as a supergraph of PDG by Horwitz *et al*. [47]. The SDG contains main and auxiliary procedures and adds some edges: 1) call edge, 2) parameter-in edge, and 3) parameter-out edge. Based on the SDG, a slicing algorithm is developed by Horwitz *et al*. [47], while Livadas *et al*. [48] formally represented it and gave an optimization algorithm to construct SDG.

Multi-dimensional data-flow graph (MDFG) was introduced by Passos *et al*. [49]. It is an extended PDG, which can be represented by a tuple of four elements, (V,E,d,t). V is the set of computation nodes and E is the set of edges. The set d is defined as a mapping $d: E \to Z^n$, which maps each edge to the multidimensional time delay between two nodes. The set t is defined as $t: V \to Z^+$, which maps each node to a computing time for the node. This model is general and has more flexible in real scheduling process. Based on the model, Lee *et al*. [50] developed a method named two-level scheduling method (TSM) to achieve more parallelism. There are two approaches to realize the method, one of them employs unimodular transformations for parallelization of inner loop, and the other uses the tiling technique.

Brandner *et al*. [51] introduced a data dependence graph for register analysis. This graph is used for the issue that some variables may be spilled to memory because registers are not enough for variables and the additional store and load

operations may decrease the performance. In the graph, register variables are included in the data dependence graph. Based on the data dependence graph, the algorithm for eliminating copies is implemented by code motion and parallel copy, which copy variables from some registers to the other registers. In fact, this graph is still an extended PDG because it takes an instruction as a vertex, and a dependence as an edge, but for each edge, an label is added to identify the type of register dependence.

Whether high-level task graph, or the low-level instruction dependence graph, in essence, the edges are usually used to represent dependences. However, the nodes can be statements/instruments, or variable, or a whole procedure.

Good graph structure is more helpful for the realization of the related algorithm. Johnson *et al*. [52] proposed that some edges can be ignored in computing the strong connected component of DAG, which results in the computations is more effective.

The model of Cosnard [53] is based on a task graph, which is still a DAG and has the form (V,E,T,C), where V, E, and T are nodes, edges, and computation time of nodes respectively. Different from other DAG is the element C represents the set of communication volumes. In the approach, the task graph can be generated by parameters at run time, so it is also named parameterized task graph (PTG). Based on the PTG and parameters, a generic scheduling algorithm is developed, which only requires portion of the PTG in memory.

Demand-based dependence graph (DDG) is a general graph, whose advantage is its explicit expression of data and state, and the implicit expression of control flow. Regionalized Value State Dependence Graph (RVSDG) is a directed acyclic graph derived from DDG [54], which is very suitable for loop transformation. RVSDG uses two kinds of nodes: simple and complex nodes. The simple nodes are used to represent the primary operations and the complex nodes are divided into two categories: γ-nodes and θ-nodes. A γ-node contains two or more subgraphs of RVSDGs, which represent the different program branches. A θ-nodes contains a loop body. This graph has more expressive for loop transformation and keep all of the semantics of CFG. Bahmann *et al*. [54] developed algorithms to realize the mutual transformation between CFG and DDG and proved the correctness, so a valuable suggestion is presented that CFG can be lift up to RVSDG.

In general, computing the components of graph is more convenient and intuitive, which is very suitable for synchronization-free parallelization. Beletska *et al*. [55] presented the theory of Iteration Space Slicing Framework (ISSF) which is based on transitive closure graph, and compared it with the Affine Transformation Framework (ATF). The results demonstrate the ISSF has more advantages than ATF.

In the model of Liu *et al*. [56], the graph is (V,E,w), where the node set V represents statements, the edge set E represents dependences. The set w is a mapping $w: E \to Z$, which represents the dependence distance for each edge. The model can extract the maximal number of iterations from a nested loop.

4. Semantics, Stencil and High Level Abstract

The basic application of semantics is directives, which are inserted into program to denote some operations. In parallelization, they contribute to transformation, synchronization, code generation, and verification, etc. Szafaryn *et al*.[57] introduced a prototype programming framework named Trellis, which is to support that a single codebase can be executed on both CPU and GPU. This object is achieved with the help of some high-level directives which are derived from some existing parallel APIs: OpenMP and OpenACC. Furthermore, an addition directive

of thread synchronization and more transformations can obtain more optimization of parallelization.

The application of implicit semantics do not need directives. Some significant data access patterns can be extracted from the function semantics of related library, and the collaborative operations of related statement set. The approach of Liao *et al*. [58] considers the high-level abstracts of special functions or data structures, which are usually derived from the well-defined semantics, such as the function list::size () of STL in C++, is read-only. The semantics of abstractions can be used for automatic parallelization. This mechanism can also encourage the classes or functions in system libraries are equipped with well-defined semantics for user exploitation. Binkley *et al*. [59] gave an empirical study of concepts expression related to data dependence. The conclusion is that the domain-level concepts in source code are cohesion in some extents, so it is suitable for partitioning a program with respect to the concepts.

The general feature of algorithms can be seen as abstract semantics, which can be used to classify algorithms. Then for different algorithm classes, different processes will be used. Nugteren *et al*. [60] introduced algorithmic species, which is a classification for loop algorithms with array operations and can be used by both compilers and programmers. The species are defined based on the array access patterns: element, chunk, etc. In the literature, the species and other classifications are compared based on the basic rules: 1) automatically extracted, 2) intuitive, 3) formally defined, 4) each algorithm must be in one species, 5) fine grained control. These basic rules make the classifications are inherent rigorous, effective, and suitable for both compilers and programmers. Therefore, a class itself may imply rich high-level semantics, which can be used for more effective parallelization.

Skeleton and stencil based approaches are mainly based on the high-level framework, which can be taken as a high-level semantic essentially. Aguston *et al*. [61] proposed a skeleton approach to converting pointer-based data structures to arrays. Formally, skeleton code and source code are not semantically equivalent, but in a sense, they may be equivalent in high-level abstract. For a simple instance, a linked list can be replaced by a one-dimensional array. The skeleton code can be a hint of possible parallelization for the programmers. Stencil computations usually have regular computational structures, which are suitable for a series algorithms, so the parallelization of stencil code can be more general, Krishnamoorthy *et al*. [62] proposed two approaches: overlapped tiling and split tiling, which can remove inter-tile dependences.

A higher level of abstraction is based on algebraic representation. These approaches are effective against irregular programs. Pingali *et al*. [63] argued that the traditional abstraction of dependence graph is unsuitable for some algorithms in advanced areas such as machine learning etc, where the irregular data structures are widely used. Then, Pingali *et al*. [63] proposed a new model, named operator formulation, in which an algorithm is regarded as some operations on some data structures. Obviously, each operator can be implemented in an independent thread, so these operators are executed essentially in parallel.

5. Dynamic Analysis and Scheduling

5.1. The Use of Compile Time Information

Static scheduling only uses compile time information, whereas modern compilers need to synergistically optimize multiple aspects, such as parallelization, load balance, and scalable, but it is hard to do everything well. The approach of Baskaran *et al*. [64] integrates these aspects of optimization. The compile time technique is used for extracting

inter-tile dependences, and the dynamic scheduling makes the system is scalable. Ottoni *et al*. [65] proposed a compiler framework named COCO, which can optimize inter-thread dependences. The main techniques include data-flow analyses, min-cut algorithm of a graph, and the insertion of synchronization and communication instructions.

5.2. The Dynamic Scheduling

Dynamic scheduling can further utilize the relevant information at run time. Kong *et al*. [66] presented a dynamic graph based approach, which compiles loop nests to a task graph, and then used different parallelization approaches for different types of dependences and communication patterns. The partitioning algorithm only uses the tile-to-tile dependences, so it can reduce many runtime work. Ketterlin *et al*. [67] introduced a tool named Parwiz, which synthetically use static and dynamic techniques for multiple types of parallelism. The main functions include using one or more trail executions to detect parallelism, performing loop transformation, and obtaining the runtime behavior of the loop. At last, the tool can give suggestions for different parallelization actions.

5.3. Pipeline Parallelization

Pipeline vectorization, proposed by Weinhardt *et al*. [68], is a typical dynamic method for loop parallelization at run time. It exploits the potential of pipelined hardware and combines the results of loop parallelization. It is different from software vectorization that all loop instructions are chained and inputted to the pipeline according to the data flow graph, without other vectorization instructions generated. However, some operations, named nonsynthesizable operations, can not be pipelined in the system, such as recursive function calls, library calls etc. Therefore, the system affords two work modes: hardware mode is used to process synthesizable operations and codesign mode is used to filter out the nonsynthesiable operations.

Decoupled software pipelining (DSWP), can decouple loop-carried dependences at runtime. However, its scalability is limited. The approach proposed by Raman *et al*. [69] uses non-speculative parallelization pipeline, named parallel-stage decoupled software pipelining (PS-DSWP), which integrates the ability of DSWP and DOALL parallelization. By isolating some recurrences in the same stage, the approach can achieve more parallelism.

Tournavitis *et al*. [70] proposed a profiling approach to extract more parallelism in pipeline stage, which is claimed to be orthogonal to other parallelization, i.e. it does not exclude other available parallelism. It performs based on intermediate representation profiling, which bridges the gap between low-level execution and the information within the compiler. The profiling for data and control dependences can be verified only by user's approval, no need for manual code annotations.

5.4. Speculative Parallelization

The general process of speculative parallelization is relative simple. That is running threads in parallel at first, and then detecting the threads violating dependence constraints and at last withdrawing and re-executing them. However, the detailed implements must involve the detailed architectures. At present, there is a large amount of literature related to speculative parallelism, we only review some characteristic contributions of recent years to know the general trend.

For the speculative parallelism, the chunking size is a key parameter affecting the performance. The approach of Estebanez *et al*. [71] employs historical information to estimate the parameter for the next step, so it dose not need prior knowledge and previous running.

Dynamic testing is a fundamental feature of speculative parallelism, and the test program is usually implanted into the original program. Jimborean *et al*. [72] presented an analyzer, which can dynamic test dependences by trial executing samples. The obtained information is used to determine the speculative optimization and parallelization. In the test, the dependence distance vectors and dependence patterns, and some scalar values can be computed. Huang *et al*. [73] proposed a parallelization technique named DOMORE, which includes a compiler and a runtime library. The compiler inserts a test engine into programs at compile time. The engine will be triggered to check the dependences at runtime and decide whether perform synchronization as needed. Herzeel *et al*. [74] focused on the dynamic parallelization of recursive procedure. The main idea is to introduce a data structure named continuator, which can trace the control dependences at runtime. The continuators are build as a tree and each continuator is a node. The main function of the continuators is to accumulate or reduce the computation results. Essentially, the tree is the model of the execution path of the recursive procedure. Each node can trace and manage the speculative branch.

Loop parallelization and non loop parallelization can be integrated together essentially. Liu *et al*. [75] proposed a method to integrating loop and non-loop partitions for speculative execution. Both sequential code and loop iterations will be partitioned, even a large loop body within a inner loop. Different from others, the method evaluates speculative sections by quantitative values instead of qualitative value. Jimborean *et al*. [76] proposed a framework based on algorithm skeletons for speculative parallelism. The skeleton is build at compiler time and is patched at runtime. By some profiling information produced by short instrumented version, the transformation is performed to support speculative parallelization.

6. Intelligent Algorithms

Recent years, some intelligent algorithms have been applied for prediction and verification in loop parallelization. These algorithms involve neural network, machine learning, and evolution algorithms.

Palkowski *et al*. [77] employed a feed-forward neural network to predict the loop transformation time, which has been integrated into a source to source compiler named TRACO. Usually, it is hard to estimate the execution time of loop transformation algorithm, especially for the transitive closure calculation. This approach can effectively predict it to support transformation decision.

The machine learning techniques are mainly applied in the prediction and valuation of parallelization. The key techniques includes support vector machine and decision tree [78]. The supervised methods usually distinguish and label the code regions which are suitable for parallelization or not, and then use the information to train the model [78]. While unsupervised learning methods are mainly based on clustering techniques and employ the historical data or profile data [79] to find program fragments with similar patterns [80].

Based on evolution computing framework, Parsa *et al*. [81] developed an genetic algorithm for loop tiling. The main work is to obtain the optimal tile size and shape in multidimensional space. GAPS [82] is another framework based on GA algorithm for restructuring loop over all of iterations of statements. UTLEA [83] aims at uniformization of non-uniform relations in perfect nested loops, and employs evolution methods to minimize both the dependence cone size and the number of vectors.

7. Conclusion and Future Trend

This paper gives a survey of the developments of loop parallelization technology in multiple aspects, which involve data dependence analysis, loop transformation, code

generation, polyhedral model, dynamic parallelization, the parallelization directed by semantic or framework, and intelligent algorithms. Models and the derived approaches are always the focus of attention. Either polyhedral model or graph based model has profound algebra background. In the future, the more general abstract models are to be expected, which may unify, or integrate, or merge some of existing models. Semantic equivalence is the theoretical basis of parallelization, and with the models becoming more abstract, parallelization can be performed based on a higher level of semantic equivalence. On the other hand, due to the wide application of dynamic parallelism, intelligent algorithms also have great potential in this field. For the implementation, the parallelization techniques will be integrated into the all kinds of architectures to achieve both the optimization of underlying running and the simplification of the user interface.

Acknowledgments

We are grateful and thankful to anonymous reviewers for the helpful comments.

References

[1] U. Banerjee, R. Eigenmann, A. Nicolau and D. A. Padua, "Automatic program parallelization", Proceedings of the IEEE., vol. 81, no. 2 **(1993)**, pp. 211-243.
[2] P. Boulet, A. Darte, G. A. Silber and F. Vivien, "Loop parallelization algorithms: From parallelism extraction to code", Parallel Computing, vol. 24, no. 3-4 **(1998)**, pp. 421-444.
[3] R. Gupta, S. Pande, K. Psarris and V. Sarkar, "Compilation techniques for parallel systems", Parallel Computing, vol. 25, no. 13-14 (1999), pp. 1741-1783.
[4] D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler transformations for high-performance computing", ACM Comput. Surv., vol. 26, no. 4 **(1994)**, pp. 345-420.
[5] C. Nugteren, P. Custers and H. Corporaal, "Algorithmic species: A classification of affine loop nests for parallel programming", Acm Transactions on Architecture and Code Optimization, vol. 9, no. 4 **(2013)**.
[6] M. Gokhale and W. Carlson, "An introduction to compilation issues for parallel machines", Journal of Supercomputing, vol. 6, no.3-4 **(1992)**, pp. 283-314.
[7] K. Psarris, "The banerjee-wolfe and gcd tests on exact data dependence information", Journal of Parallel and Distributed Computing, vol. 32, no. 2 **(1996)**, pp. 119-138.
[8] Z. Li, P. C. Yew and C. Q. Zhu, "An efficient data dependence analysis for parallelizing compilers", IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1 **(1990)**, pp. 26-34.
[9] X. Kong, D. Klappholz and K. Psarris, "The i test: An improved dependence test for automatic parallelization and vectorization", IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 3 **(1991)**, pp. 342-349.
[10] K. Psarris, X. Kong and D. Klappholz, "The direction vector i test", IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 11 **(1993)**, pp. 1280-1290.
[11] W. L. Chang and C. P. Chu, "The generalized direction vector i test", Parallel Computing, vol. 27, no. 8 **(2001)**, pp. 1117-1144.
[12] J. Zhou and G. Zeng, "A general data dependence analysis for parallelizing compilers", Journal of Supercomputing, vol. 45, no. 2 **(2008)**, pp. 236-252.
[13] M. Wolfe and C. W. Tseng, "The power test for data dependence", IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 5 **(1992)**, pp. 591-601.
[14] W. Pugh, "The omega test: A fast and practical integer programming algorithm for dependence analysis", in Proceedings of the 1991 ACM/IEEE conference on Supercomputing, ACM, Albuquerque, New Mexico, USA **(1991)**, pp. 4-13.
[15] T. C. Huang and C. M. Yang, "Data dependence analysis for array references", Journal of Systems and Software, vol. 52, no. 1 **(2000)**, pp. 55-65.
[16] L. Rauchwerger and D. A. Padua, "The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization", IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 2 **(1999)**, pp. 160-180.
[17] W. Blume and R. Eigenmann, "Nonlinear and symbolic data dependence testing", IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 12 **(1998)**, pp. 1180-1194.
[18] K. Kyriakopoulos and K. Psarris, "Nonlinear symbolic analysis for advanced program parallelization", Ieee Transactions on Parallel and Distributed Systems, vol. 20, no. 5 **(2009)**, pp. 623-640.
[19] A. Darte, "On the complexity of loop fusion", Parallel Computing, vol. 26, no. 9 **(2000)**, pp. 1175-1193.
[20] N. Manjikian and T. S. Abdelrahman, "Fusion of loops for parallelism and locality", Ieee Transactions on Parallel and Distributed Systems, vol. 8, no. 2 **(1997)**, pp. 193-209.

[21] E. H. D'Hollander, "Partitioning and labeling of loops by unimodular transformations", Parallel and Distributed Systems, IEEE Transactions on, vol. 3, no. 4 **(1992)**, pp. 465-476.
[22] J. L. Xue, "Unimodular transformations of non-perfectly nested loops", Parallel Computing, vol. 22, no. 12 **(1997)**, pp. 1621-1645.
[23] J. Ramanujam, "Beyond unimodular transformations", Journal of Supercomputing, vol. 9, no. 4 **(1995)**, pp. 365-389.
[24] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine partitions", Parallel Computing, vol. 24, no. 3-4 **(1998)**, pp. 445-475.
[25] A. W. Lim and M. S. Lam, "Communication-free parallelization via affine transformations", Languages and Compilers for Parallel Computing. 7th International Workshop Proceedings **(1995)**, pp. 92-106.
[26] W. Li and K. Pingali, "A singular loop transformation framework based on non-singular matrices", Int J Parallel Prog, vol. 22, no. 2 **(1994)**, pp. 183-205.
[27] L. N. Pouchet, C. Bastoul, A. Cohen and N. Vasilache, "Iterative optimization in the polyhedral model: Part i, one-dimensional time", in Cgo 2007: International Symposium on Code Generation and Optimization **(2007)**, pp. 144-156.
[28] P. Y. Calland, J. Dongarra and Y. Robert, "Tiling with limited resources", Proceedings. IEEE International Conference on Applications-Specific Systems, Architectures and Processors **(1997)**, pp. 229-238.
[29] J. L. Xue, "Communication-minimal tiling of uniform dependence loops", Journal of Parallel and Distributed Computing, vol. 42, no. 1 **(1997)**, pp. 42-59.
[30] M. Griebl, P. Faber and C. Lengauer, "Space-time mapping and tiling: A helpful combination", Concurrency and Computation-Practice & Experience, vol. 16, no. 2-3 **(2004)**, pp. 221-246.
[31] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy and B. Norris, "Parametric multi-level tiling of imperfectly nested loops", in Ics'09: Proceedings of the 2009 Acm Sigarch International Conference on Supercomputing **(2009)**, pp. 147-157.
[32] M. Jimenez, J. M. Llaberia and A. Fernandez, "Register tiling in nonrectangular iteration spaces", Acm Transactions on Programming Languages and Systems, vol. 24, no. 4 **(2002)**, pp. 409-453.
[33] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, K. O'Brien and Acm, "Automatic creation of tile size selection models", in Cgo 2010: The Eighth International Symposium on Code Generation and Optimization, Proceedings **(2010)**, pp. 190-199.
[34] S. Hack, D. Grund and G. Goos, "Register allocation for programs in ssa-form", in Lect notes comput sc, eds. A. Mycroft and A. Zeller **(2006)**, pp. 247-262.
[35] R. Baghdadi, A. Cohen, S. Verdoolaege and K. Trifunovic, "Improved loop tiling based on the removal of spurious false dependences", Acm Transactions on Architecture and Code Optimization, vol. 9, no. 4 **(2013)**.
[36] C. Ancourt and F. Irigoin, "Scanning polyhedra with do loops", SIGPLAN Notices, vol. 26, no. 7 **(1991)**, pp. 39-50.
[37] P. Boulet and P. Feautrier, "Scanning polyhedra without do-loops", Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques **(1998)**, pp. 4-11.
[38] P. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces", Journal of Vlsi Signal Processing Systems for Signal Image and Video Technology, vol. 19, no. 2 **(1998)**, pp. 179-194.
[39] M. Griebl, P. Feautrier and C. Lengauer, "Index set splitting", Int J Parallel Prog, vol. 28, no. 6 **(2000)**, pp. 607-631.
[40] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen and C. Bastoul, "The polyhedral model is more widely applicable than you think", in Lect notes comput sc, ed. R. Gupta **(2010)**, pp. 283-303.
[41] U. Bondhugula, A. Hartono, J. Ramanujam and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer", Acm Sigplan Notices, vol. 43, no. 6 **(2008)**, pp. 101-113.
[42] T. Grosser, S. Verdoolaege and A. Cohen, "Polyhedral ast generation is more than scanning polyhedra", Acm Transactions on Programming Languages and Systems, vol. 37, no. 4 **(2015)**.
[43] R. Upadrasta and A. Cohen, "Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra", Acm Sigplan Notices, vol. 48, no. 1 **(2013)**, pp. 483-495.
[44] F. Quillere, S. Rajopadhye and D. Wilde, "Generation of efficient nested loops from polyhedra", Int J Parallel Prog, vol. 28, no. 5 **(2000)**, pp. 469-498.
[45] C. Bastoul and s. ieee computer, "Code generation in the polyhedral model is easier than you think", in 13th International Conference on Parallel Architecture and Compilation Techniques, Proceedings **(2014)**, pp. 7-16.
[46] J. Ferrante, K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization", ACM Trans. Program. Lang. Syst., vol. 9, no. 3 **(1987)**, pp. 319-349.
[47] S. Horwitz, T. Reps and D. Binkley, "Interprocedural slicing using dependence graphs", Acm Sigplan Notices, vol. 39, no. 4 **(2004)**, pp. 232-243.
[48] P. E. Livadas and T. Johnson, "An optimal algorithm for the construction of the system dependence graph", Information Sciences, vol. 125, no. 1-4 **(2000)**, pp. 99-131.
[49] N. L. Passos and E. H. M. Sha, "Scheduling of uniform multidimensional systems under resource constraints", Ieee Transactions on Very Large Scale Integration (Vlsi) Systems, vol. 6, no. 4 **(1998),** pp. 719-730.

[50] Y. H. Lee and C. Chen, "A two-level scheduling method: An effective parallelizing technique for uniform nested loops on a dsp multiprocessor", Journal of Systems and Software, vol. 75, no. 1-2 **(2005)**, pp. 155-170.
[51] F. Brandner and Q. Colombet, "Elimination of parallel copies using code motion on data dependence graphs", Computer Languages Systems & Structures, vol. 39, no. 1 **(2013)**, pp. 25-47.
[52] N. P. Johnson, T. Oh, A. Zaks and D. I. August, "Fast condensation of the program dependence graph", Acm Sigplan Notices, vol. 48, no. 6 **(2013)**, pp. 39-49.
[53] M. Cosnard and E. Jeannot, "Compact dag representation and its dynamic scheduling", Journal of Parallel and Distributed Computing, vol. 58, no. 3 **(1999)**, pp. 487-514.
[54] H. Bahmann, N. Reissmann, M. Jahre and J. C. Meyer, "Perfect reconstructability of control flow from demand dependence graphs", ACM Trans. Archit. Code Optim., vol. 11, no. 4 **(2015)**, pp. 1-25.
[55] A. Beletska, W. Bielecki, A. Cohen, M. Palkowski and K. Siedlecki, "Coarse-grained loop parallelization: Iteration space slicing vs affine transformations", Parallel Computing, vol. 37, no. 8 **(2011)**, pp. 479-497.
[56] D. Liu, Y. Wang, Z. L. Shao, M. Y. Guo and J. L. Xue, "Optimally maximizing iteration-level loop parallelism", Ieee Transactions on Parallel and Distributed Systems, vol. 23, no. 3 **(2012),** pp. 564-572.
[57] L. G. Szafaryn, T. Gamblin, B. R. de Supinski and K. Skadron, "Trellis: Portability across architectures with a high-level framework", Journal of Parallel and Distributed Computing, vol. 73, no. 10 **(2013)**, pp. 1400-1413.
[58] C. H. Liao, D. J. Quinlan, J. J. Willcock and T. Panas, "Semantic-aware automatic parallelization of modern applications using high-level abstractions", Int J Parallel Prog, vol. 38, no. 5-6 **(2010)**, pp. 361-378.
[59] D. Binkley, N. Gold, M. Harman, Z. Li and K. Mahdavi, "An empirical study of the relationship between the concepts expressed in source code and dependence", Journal of Systems and Software, vol. 81, no. 12 **(2008)**, pp. 2287-2298.
[60] C. Nugteren, P. Custers and H. Corporaal, "Algorithmic species: A classification of affine loop nests for parallel programming", Acm Transactions on Architecture and Code Optimization, vol. 9, no. 4 **(2013)**.
[61] C. Aguston, Y. Ben Asher and G. Haber, "Parallelization hints via code skeletonization", Ieee Transactions on Parallel and Distributed Systems, vol. 26, no. 11 **(2015)**, pp. 3099-3107.
[62] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev and P. Sadayappan, "Effective automatic parallelization of stencil computations", Acm Sigplan Notices, vol. 42, no. 6 **(2007)**, pp. 235-244.
[63] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos and X. Sui, "The tao of parallelism in algorithms", Acm Sigplan Notices, vol. 46, no. 6 **(2011)**, pp. 12-25.
[64] M. M. Baskaran, N. Vydyanathan, U. K. Bondhugula, J. Ramanujam, A. Rountev and P. Sadayappan, "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors", Acm Sigplan Notices, vol. 44, no. 4 **(2009)**, pp. 219-228.
[65] G. Ottoni and D. I. August, "Communication optimizations for global multi-threaded instruction scheduling", Acm Sigplan Notices, vol. 43, no. 3 **(2008)**, pp. 222-232.
[66] M. Kong, A. Pop, L. N. Pouchet, R. Govindarajan, A. Cohen and P. Sadayappan, "Compiler/runtime framework for dynamic dataflow parallelization of tiled programs", Acm Transactions on Architecture and Code Optimization, vol. 11, no. 4 **(2014)**.
[67] A. Ketterlin, P. Clauss and I. C. Society, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization", in 2012 ieee/acm 45th international symposium on microarchitecture **(2012)**, pp. 437-448.
[68] M. Weinhardt and W. Luk, "Pipeline vectorization", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 20, no. 2 **(2001)**, pp. 234-248.
[69] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, D. I. August and Acm, "Parallel-stage decoupled software pipelining", in Cgo 2008: Sixth International Symposium on Code Generation and Optimization, Proceedings **(2008)**, pp. 114-123.
[70] G. Tournavitis, B. Franke and Acm, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information", in Pact 2010: Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques **(2010)**, pp. 377-388.
[71] A. Estebanez, D. R. Llanos, D. Orden and B. Palop, "Moody scheduling for speculative parallelization", in Euro-par 2015: Parallel processing, eds. J. L. Traff, S. Hunold and F. Versaci **(2015)**, pp. 135-146.
[72] A. Jimborean, P. Clauss, J. M. Martinez and A. Sukumaran-Rajam, "Online dynamic dependence analysis for speculative polyhedral parallelization", in Euro-par 2013 parallel processing, eds. F. Wolf, B. Mohr and D. A. Mey **(2013)**, pp. 191-202.
[73] J. L. Huang, T. B. Jablin, S. R. Beard, N. P. Johnson, D. I. August and Ieee, "Automatically exploiting cross-invocation parallelism using runtime information", in Proceedings of the 2013 ieee/acm international symposium on code generation and optimization **(2013)**, pp. 246-256.
[74] C. Herzeel and P. Costanza, "Dynamic parallelization of recursive code part i: Managing control flow interactions with the continuator", Acm Sigplan Notices, vol. 45, no. 10 **(2010)**, pp. 377-396.

[75] B. Liu, Y. L. Zhao, Y. X. Li, Y. J. Sun and B. Q. Feng, "A thread partitioning approach for speculative multithreading", Journal of Supercomputing, vol. 67, no. 3 **(2014)**, pp. 778-805.
[76] A. Jimborean, P. Clauss, J. F. Dollinger, V. Loechner and J. M. M. Caamano, "Dynamic and speculative polyhedral parallelization using compiler-generated skeletons", Int J Parallel Prog, vol. 42, no. 4 **(2014)**, pp. 529-545.
[77] M. Palkowski and W. Bielecki, "Using an artificial neural network to predict loop transformation time", in Artificial intelligence and soft computing, pt i, eds. L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh and J. M. Zurada **(2015)**, pp. 102-111.
[78] D. Fried, Z. Li, A. Jannesari and F. Wolf, "Predicting parallelization of sequential programs using supervised learning", 2013 12th International Conference on Machine Learning and Applications **(2013)**, Vol 2, pp. 72-77.
[79] Z. Wang, G. Tournavitis, B. Franke and M. F. P. O'Boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping", Acm Transactions on Architecture and Code Optimization, vol. 11, no. 1 **(2014)**.
[80] J. Demme and S. Sethumadhavan, "Approximate graph clustering for program characterization", Acm Transactions on Architecture and Code Optimization, vol. 8, no. 4 **(2012)**.
[81] S. Parsa and S. Lotfi, "A new genetic algorithm for loop tiling", Journal of Supercomputing, vol. 37, no. 3 **(2006)**, pp. 249-269.
[82] A. Nisbet, "Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation", in High-performance computing and networking, eds. P. Sloot, M. Bubak and B. Hertzberger **(1998)**, pp. 987-989.
[83] S. Mahjoub and S. Lotfi, "The utlea: Uniformization of non-uniform iteration spaces in three-level perfect nested loops using an evolutionary algorithm", in Software engineering and computer systems, pt 2, eds. J. M. Zain, W. M. B. Mohd and E. ElQawasmeh **(2011)**, pp. 605-617.

Authors

Hong Yao, is currently a Ph.D. student of computer science and engineering, South China University of Technology. His main research interests include parallel and distributed computing, cloud computing and theory of computation.

Huifang Deng, is currently a professor and Ph.D. supervisor of computer science and engineering, South China University of Technology. Before 2005, he had been engaged in teaching and scientific research in UK for 16 years. He received his Ph.D. in University of London. His main interests include high performance computing, big data processing.

Caifeng Zou, is currently a PH.D. student of computer science and engineering, South China University of Technology. Her main research interests include big data processing and cloud computing.

