Automatic Parallelization Tool: Classification of Program Code for Parallel Computing

Mustafa Basthikodi, Research Scholar, Dept. Of CSE, BIT, Mangalore, India, mbasthik@gmail.com

Dr. Waseem Ahmed, Dept. Of CSE, HKBKCE, Bangalore, India, waseem.pace@gmail.com

ABSTRACT

Performance growth of single-core processors has come to a halt in the past decade, but was re-enabled by the introduction of parallelism in processors. Multicore frameworks along with Graphical Processing Units empowered to enhance parallelism broadly. Couples of compilers are updated to developing challenges forsynchronization and threading issues. Appropriate program and algorithm classifications will have advantage to a great extent to the group of software engineers to get opportunities for effective parallelization. In present work we investigated current species for classification of algorithms, in that related work on classification is discussed along with the comparison of issues that challenges the classification. The set of algorithms are chosen which matches the structure with different issues and perform given task. We have tested these algorithms utilizing existing automatic species extraction toolsalong with Bones compiler. We have added functionalities to existing tool, providing a more detailed characterization. The contributions of our work include support for pointer arithmetic, conditional and incremental statements, user defined types, constants and mathematical functions. With this, we can retain significant data which is not captured by original speciesof algorithms. We executed new theories into the device, empowering automatic characterization of program code.

Index Terms—**Access Patterns, Bones, Parallel Programming, Algorithm Classification**

1. INTRODUCTION

With the influx of many cores every processor has parallel computational force in built which is completely used when code in execution written consequently. Existing auto-parallelization systems are not completely automatic. The focus of this work is a new algorithm classification, 'Algorithmic Species' which typifies pertinent data for parallelization of the algorithm in classes[1]-[5]. Algorithmic species is built using access patterns which are in arrays of loop nest. The classification intends to satisfy below objectives: i) the program code can be reasoned by the developers using algorithm classes, ii) class data is utilized by execution models to anticipate execution, and iii) the classification may be used to plan the compilers. For achieving these necessities classes set asextractedautomatically, instinctive, fine-grained, characterized formallyand completed. Species are used for classification of parallel code other than means to separate parallelism. The classification can in turn be used for various purposes, e.g. to predict performance on a given parallel architecture, reason about the algorithm; generate parallel program code orcompile to a parallel program. An example of a classified algorithm is given in section 3. Thiswork is unitof current research of constructing novel programming model for parallelization in multi-core architectures.

Additionally the parallelization tool, the par4 tool is introduced, which is a fully automatic parallelization tool. It automaticallygenerates the species for the given sequential C source code without any manual intervention.

2. MOTIVATION

Parallelism is a significant feature to consider while making new applications since it promises execution gains utilizing multicores. For exist legacy applications there is a need to rewrite or change them to parallel utilizing some devices [6]. The shift towards parallel computing introduced challenges in both efficient programming and compilation: managing multi-threading and effectively utilizing the memory of processor are the cases which programmers and compilers face. Algorithm classification, describing the algorithm characteristics in a target platform independent way abstract away from these problems [7]-[10]. The classification does not change overtime and new parallel code can be created when the tools are accommodated to the changes in the parallel hardware. As a result, we foresee an algorithm classification as a suitable instrument to face the current and future challenges in parallel computing. In this light, our approach hassimilar thoughts as the production layer and efficiency layer as an existing classification. In theproduction layer, a programmer writes his program code with an associated “design pattern”, an expert parallel programmer implements an highly efficient solution in the efficiency layer withe.g. a skeleton implementation or programming framework [11]-[14]. With our algorithmic species weintroduce these so-called design patterns for the production layer.We introduce three goals for an algorithm classification which we believe to be vital toaid in the process of creating parallel programs:

1) With a descriptive classification, programmers can discuss algorithms in natural language enabling them to select known optimizations withtargetarchitecture in mind or to finetune a classified algorithm. This also enables them to discuss their algorithms at anabstract level, making it easier to reason about their computational problems.

2) Performance prediction is a key factor in a development process, during design spaceexploration different (parallel) architectures or accelerator configurations are explored todetermine the optimal configuration. Instead of creating parallel programs for all architectures, performance is predicted based on the classified algorithm and anappropriate performance model.

3) We believe the algorithm classification facilitates the design and optimization ofparallelizing-compilers, source-to-source compilers and auto-tuners.

To endeavor parallelism new devices, structures, dialects or augmentations to current dialects are required. Today a plenty of parallelization devices, systems, dialects are accessible in the business sector. Everyone has its significance and may be suitable for parallelizing various types of applications. These devices can create their decisions on data inserted in the algorithm classification which significantly lifts the burden of designing such devices as they don't need to extract this information themselves. The classification acts as an enabler to these tools, providing a common front-end which extracts information and represents it in a formally defined manner.

3. RELATED WORK

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently. There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. As power consumption (and consequently heat generation) by computers has become a concern in recent years, computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.

Recent advances in multi-core and many-core processors require programmers to exploit an increasing amount of parallelism from their applications. Data parallel languages such as CUDA and OpenCL make it possible to take advantage of such processors, but still require a large amount of effort from programmers. To address the challenge of parallel programming, we introduce Bones .

Bones is a source-to-source compiler based on algorithmic skeletons and a new algorithm classification. The compiler takes C-code annotated with class information as input and generates parallelized target code. Targets include NVIDIA GPUs (through CUDA), AMD GPUs (through OpenCL) and x86 CPUs (through OpenCL and OpenMP). Bones is open source, written in the Ruby programming language. The compiler is based on the C-parser CAST, which is used to parse the input code into an abstract syntax tree (AST) and to generate the target code from a transformed AST. The original algorithmic species theory included ASET, a polyhedral based algorithmic species extraction tool. Along with a new non-polyhedral theory, a new automatic extraction tool named A-Darwin (short for `automatic Darwin') was introduced .

A-Darwin is a tool to automatically extract algorithmic species. The new tool is largely equal to ASET in terms of functionality, but is different internally. The tool is based on CAST, a C99 parser which allows analysis on an abstract syntax tree (AST). From the AST, the tool extracts the array references and constructs a 5 or 6-tuple for each loop nest. Following, merging is applied and the species are extracted. Finally, the species are inserted as pragmas in the original source code. To perform the dependence tests in A-Darwin, a combination of the GCD and Banerjee tests was made. Together, these tests are conservative, so it might not find all species.

The automatic use of algorithmic species is presented using BONES source-to-source compiler. The compiler is based on algorithmic skeletons, a technique using parameterized program code(skeletons) to generate high performance code. A single skeleton can be seen as template code for a specific class of computations on a specific target processor. Skeletons can be added as new classes are identified, creating a flexible compiler. Typically, users of skeleton-based compilers are required to manually select a suitable skeleton for their algorithm. However, in the case of BONES, algorithmic species information is used to automatically select a skeleton for a given algorithm. This makes BONES, combined with an automatic species extraction tool such as ASET, a fully automatic source-to source compiler.

Substantial measure of classification of algorithms has been studied before planning novel species. Few algorithms are been analyzed by numerous individuals, for example the approach exhibited by Allen, Kennedy et.al. The scientific notations and toolsused by these algorithms are different. Moreover, these don't rely on representation of information dependence. The procedure for transformation of similar loops where reliance vector expresses priority requirements on emphases loops [14].

The species of algorithms are utilized in [5] to attain portability over diverse architectures. The skeletons may be visualized as parameterizedsample code for particular class of computations on particular destination processor. The instantiation of skeleton and the creation of proficient destination code are done by the compiler. Here, the skeletonsrefers to species of algorithms such as picking a skeleton may be a subject of code classification.

The classifications related comparisons including array regions proposed by B. Creusillet and F.Irigoi in their work compilers and languages used in parallel computing and Ecute by High Performance Embedded Architectures and Compilers by L. Howes, A Lokhmotov, species of algorithms givesproject code abstraction i.e. the data lost in interpretation from program to the species.

Thealgorithmic species inspired by the algorithm classification is presented in [13]. In [2],the theory is according to polyhedral model, insisting code to bedenoted as set of loop nests which are static and affine. The use of the Polyhedral Model, imposes some fundamental restrictions to the program code classified such as 1) ambiguity duringclassification (an algorithm can be classified in multiple ways), 2) classes as upper bounds, 3)lack of validation for completeness and applicability in real-life, and 4) the inability toautomatically extract the classes from program code.

The original algorithmic species hypothesis comprised ASET, a polyhedral based extraction tool for algorithmic species. With new non-polyhedral hypothesis there is a novel extraction tool which is automatic called ADarwin. The device is similar to ASET for its functions but internally it is different. The device is according to CAST, a parser which permits analysis on AST. The device obtains from abstract syntax tree array references which build 5 to 6-tuple for every loop which is used to extract species after merging operation. At last, theinsertion of species happens as pragma's in original source code. To perform reliance tests in ADarwin, we use GCD and Banerjee tests combination. Combining these tests we get moderate results, that is, all species may not be discoveredthe code segments which are not recognized in existing works are identified and accordingly the tool is modified in our work.

4. IMPLEMENTATION

Bones and A-Darwin along with required gems are installed in quad core system for experimentation the species is extracted. The species are inserted as pragmas in the original program code. The code segments of various algorithm classes are executed to analyze the output of A-Darwin. We have executed and analyzed classes using Bones Compiler and found that there are few kernels of which A-Darwin is not considering all possibilities of code.

Skeleton-based compilation has several benefits. Firstly, compilation requires only basic transformations that can be performed at abstract syntax tree level, omitting the need for intermediate representations which often lose code structure and variable naming. This allows the compiler to generate readable code, enabling opportunities for further fine-tuning and manual optimization. Because of the integration of algorithmic species, Bones is the first skeleton-based compiler that can be used in a fully-automatic tool-flow. This removes the requirements of existing skeleton-based approaches to manually identify a skeleton and modify the code such that the skeleton can be used. Furthermore, algorithmic species provides a clear, structured, and formally defined way of using skeletons, which can be beneficial in cases where manual classification is unavoidable.

Algorithmic species is a classification which captures low-level algorithm details from individual loops or loop nests and their bodies. Key to the algorithmic species approach is that every array, accessed in the classified loop nest, is assigned with one of the five access patterns. The combination of access patterns, of the input and output arrays of the loop nest, and then form the species. This modular approach enables us to form an unlimited amount of species with the use of only five access patterns. Because the theory behind algorithmic species is built upon the Polyhedral Model, we use a polyhedral representation of the program code as input to A-Darwin. Algorithmic species can therefore serve as a base for current and future work related to parallel programming. A-DARWIN also includes basic dependence analysis, to be usable as a stand-alone tool. All arrays in all loop nests are classified, their ranges derived and the

available parallelism extracted, they are combined into a species and annotated in the source code.

The major contributions of our work include:

- Support for pointer arithmetic.
- Support for Conditional and Incremental Statements.
- Support for Mathematical functions
- Support for User defined Types and Constants.
- Design of a GUI that make easy for programmers to use the tool.

The algorithmic species extraction tool,takes a sequential C code as input andautomatically generates the species-annotated C code. This generated species-annotated C codeis then given as an input to the 'Bones' skeleton based compiler which gives the efficient parallelcode as output as depicted in Fig. 1.

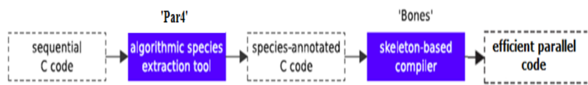

Fig1: Overview of our auto-parallelization approach.

Algorithmic species is a classification which captures low-level algorithm details fromindividual loops or loop nests and their bodies. Key to the algorithmic species approach is thateveryarray retrieved in classified loop nest, is assigned with one of access patterns. Access patterns, of input and the output arrays of loop nest combination formsthe species [1].

One of the objectives behind the work on algorithmic species is to have the capacity to develop the pertinence and expressiveness of species to support the utilization of pointers. Implanting extra data in the array reference characterizations can enhance the present utilizations of the algorithmic species thus empowering new users.

```
1    for (i = 0; i<= 256; i++) {
2    *ptr = val;
3    val = &a[i];
4    }
```

Listing 1: Example of one dimensional pointer reference.

In Listing 1,The reference would be characterized as (*ptr) with respect to the loop nest. Thisalgorithm takes the pointer statement in loop body as the input and outputs species annotations.Initially, the variable name, initial value and its ranges are extracted theand stored in arrays. Itsindex specifies the order of the loop and this allows the extraction of the above values of aparticular “for” loop.The output pattern for this algorithm is as follows:

a[0:256]|element →ptr|pointer

The listing 2 creates a more detailed characterization,enablingus to differentiate the mathematical functions among the other accesses. 5-tuple array reference characterization has to be modified toobtain the prefix of Math to thefunction.

```
1    for (m = 0; m<50; m++){
2    for (n = 0; n<50; n++){
3    c[m][m+1] = sin(b[m][n]) * cos(a[m][n]);
4    }
5    }
```

Listing2: Example of two dimensional loop nest with mathematical functions.

The lines 1-5 shows the use of the mathematical functions sine andcosine.The output pattern for this algorithm is as follows:

b[0:49,0:49]||chunk(0:0,0:49),Math.sin^a[0:49,0:49]|| chunk(0:0,0:49), Math.cos→c[0:49,1:50]||element

The applicability of the static analysisis limited.Loop nests such cannot alwaysbe fully analyzed. Therefore, species in overapproximations is a type of some cases in user defined classification. The over approximations is tightened using manual approach dynamic approach.

```
1    for (m =0; m<50; m++){
2    for (n = 0; n<50; n++){
3    c[m][n] = sum(a[m][n]);
4    }
5     }
```

Listing3:Example of two dimensional loop nest with user defined type.

In listing 3, the lines 1-5 shows the use of the user defined functionsum.The output pattern for this algorithm is as follows:

a[0:49,0:49]|element, usertype.sum→c[0:49,0:49]|element

The large number of such code segments are created and given as input to the tool to get the design patterns consisting of algorithmic species. There are set of bench programs used to test the tool for working. The modified tool works comparatively well for all the set of code segments. The algorithms given below summarizes the actual working of the tool.

***Input**: Access descriptions of all arrays*
***Output**: The access patterns of all arrays*
P = ø
repeat
Sp= ø
(Ap;Bp;~cp) ←access description of array p
switch**(Ap;Bp) **do
***case**Ap= 0 &**B**p= 0*
Sp← “increment”
***case**Ap≠ 0 &**B**p= 0*
Sp← “element”
***case**Ap= 0 &**B**p≠ 0*
Sp← "full”
***case**Ap≠ 0 &**B**p≠ 0*
if**(equation 2 holds for array p) **then
Sp← “neighborhood”
else
Sp← “chunk”
end
if** (equation 1 has constant ~c) **then
Sp← “const”
end
endsw
endsw
P←PUSp
***until**all patterns are derived;*

Result: P

Algorithm1 : Deriving the array access patterns

```
Input: Access descriptions of all arrays
Output: The access patterns for pointer species
functionget_pointer
scope_code=get input from source code
X←scope_code.scan(/\*?\w+)
for each |a| in X do
ifa contains '*'
remove * from element a
a+=[pointer]
end if
end for
S←generate species code from pattern species
Y ← get adarwin interval from S
P←get pattern part from S
for each |a| in Y do
ifa == "pointer:pointer"
pattern ← "|pointer"
end if
end for
replace pattern part in species to P
removeadarwin interval "pointer:pointer" from species code
end function
```

Algorithm 2: Deriving the Pointer Species

```
Input: Access descriptions of all arrays
Output: The access patterns for conditional statements
functionget_if
scope_code=get input from source code
X←scopecode.scan(/\(?\[?\w+\]?\]/)
flag← false
for each |a| in X do
ifa contains '('
ifprevious element of a== "if"
a+=[start]
flag←true
end if
end if
if flag == true and a == ')'
replace ')' with "[end]"
end if
ifa==comparing operator
a←'='
end if
end for
S←generate species code from pattern species
Y ← get adarwin interval from S
P ← pattern from S
for each |a| in Y do
ifa== "end:end"
P+=}|compare
end if
end for
replace pattern part in species to P
removeadarwin interval "start:start" and "end:end"
end function
```

Algorithm 3: Deriving the array access patterns with conditional statements

```
Input: Access descriptions of all arrays
Output: The access patterns for mathematical and user-defined types
functionget_function
scope_code=get input from source code
X←scope_code.scan(/\(?\[?\w+\)?\]?)
flag← false
for each |a| in X do
ifa contains '('
ifprevious element of a does not include for
function_name←previous element of a
previous element of a ← " "
flag←true
end if
end if
end for
Math ←array containing list of all mathematical function
S←generated species code from pattern species
Y ← get adarwin interval from S
P ← pattern part from S
for each |a| in Y do
ifa == function_name:function_name
iffunction_name element of math
P+="}|Math."+function_name
else
P+="}|UserType."+function_name
end if
end if
end for
replace the pattern part in species to P
removeadarwin interval function_name:function from species code
end function
```

Algorithm 4: Deriving the Mathematical and User-defined Species

5. EXPERIMENTAL RESULTS AND ANALYSIS

The parallel compiler Bones and altered ADarwin along with the gems are introduced in the quad core framework for analysis and experimentation.

The Bones is composed by taking according to skeletons and species of algorithms. The compiler takes the input C source code and creates an output as parallel code. Destination processors incorporate CUDA based NVIDIA GPUs,OpenCLbased AMD GPUs, OpenMPandOpenCLbasedCPUs. The Bones depends on CAST C program code parser, which can be utilized to parse data source code into the AST and then produce wanted source code from converted AST.

Automatic Darwin, is an automatic extraction device which depends on CAST, which is C99 parser that permits investigation on AST. After the AST is obtained, the apparatus separates array references which develop a 5 to 6-tuple for every loop nest. Then consolidation is applied to extract the species. The species which are embedded as pragmas for original source code at the end.

The code sections of different algorithm classes were executed to analyze the output obtained from the tool. To approve the utilization of algorithmic species, we classify many algorithms.Hence we have taken the benmark suite as PolyBench/C3.2, which comprises upto 35 algorithms chosen from 7 domains for scientific processing, guaranteeing wide range of algorithms.

An outline for algorithms inbenmarkPolyBench gathered by various domains are tabulated in Table 1.

Algorithmic Class	Number of Kernels	Number of KernelsExecuted	Hit Ratio(%)
2mm.c	2	2	100
3mm.c	3	3	100
adi.c	5	4	80
atax.c	2	2	100
bicg.c	2	2	100
cholesky.c	4	2	50
correlation.c	5	3	60
covariance.c	3	2	66
doitgen.c	2	2	100
durbin.c	3	2	66
dynprog.c	2	2	100
fdtd-2d.c	2	4	100
floyd-warshall.c	1	1	100
gemm.c	1	1	100
gemver.c	4	4	100
gesummv.c	1	1	100
jacobi-1d-imper.c	2	2	100
Jacobi-2d-imper.c	2	2	100
ludcmp.c	7	4	57
mvt.c	2	2	100
reg_detect.c	4	2	50
syr2k.c	1	1	100
syrk.c	1	1	100
trisolv.c	1	1	100
trmm.c	1	1	100

Table1.PolyBench benchmark Suit classification results

6.CONCLUSION AND FUTURE WORK

The recent advancement inheterogeneous and parallel computing platforms introducedchallenges for parallel programmers and compiler designers. This papersaid about Algorithmic Species which is an classifier of algorithm whichcaptures low-level algorithmic details and represents them with the use of additional access patterns that takes pointers, mathematical functions, user defined types, incrementalvariables, if-conditions and constants into consideration. Algorithmic species can be usedby programmers to converse on algorithms or serve as a front end to performance prediction nmodels or parallelizing compilers. Furthermore, we introduced a tool which automaticallyclassifies the algorithms and generates the species. This also automates the completeparallelization process.

Future work on algorithmic species can expand the classification to more typesofalgorithms with a particular focus on irregular algorithms and remove limitations to the theory (e.g. using explicit multidimensional array accesses). Next to only classifying program code, common dependency resolving transformations (e.g. loop peeling) can beincorporated before classifying programs in order to extract even more parallelism. Asfuture possible use, next to the performance-centric uses we presented so far, we foreseealgorithmic species as input to a model to estimate the energy consumption of an algorithm as energy is becoming an increasingly important topic in parallel computing.

REFERENCES

[1] Mustafa B., Waseem Ahmed, Extended species for code parallelization through algorithmic classification, IEEE IACC 2015.

[2] P. Custers. Algorithmic Species: Classifying Program Code for Parallel Computing. Master's thesis, Eindhoven University of Technology, 2012

[3] C. Nugteren, R. Corvino, and H. Corporaal. Algorithmic Species Revisited: A program Code Classification Based on Array References, Eindhoven University of Technology, 2014.

[4] C. Nugteren, P. Custers, and H. Corporaal. Algorithmic Species: An Algorithm Classification of Affine Loop Nests for Parallel Programming. ACM TACO: Transactions on Architecture and Code Optimisations, 9(4):Article 40, 2013.

[5] C. Nugteren, P. Custers, and H. Corporaal. Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification. In APPT '13: Advanced Parallel Processing Technology. Springer, 2013.

[6] S. Guelton, M. Amini, and B. Creusillet. Beyond Do Loops: Data Transfer Generation with Convex Array Regions. In LCPC '12: Languages and Compilers for Parallel Computing. Springer, 2012.

[7] C. Nugteren and H. Corporaal, "Introducing 'Bones': A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons," in GPGPU-5: 5th Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2012.

[8] W. Caarls, P. Jonker, and H. Corporaal, "Algorithmic Skeletons for Stream Programming in Embedded Heterogeneous Parallel Image Processing Applications," in IPDPS '06: 20th International Parallel and Distributed Processing Symposium. IEEE,2006.

[9] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," ACM Trans. Program. Lang. Syst., vol. 9, no. 4, pp. 491–542, Oct. 1987.

[10] X. Kong, D. Klappholz, and K. Psarris, "The I test: an improved dependence test for automatic parallelization and vectorization," Parallel and Distributed Systems, IEEE Transactions on, vol. 2, no. 3, pp. 342 –349, jul 1991. [16] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," SIGPLAN Not., vol. 26, no. 6, pp. 15–29, May 1991.

[11] 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, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 4–13.

[12] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static compilation analysis for hostaccelerator communication optimization," in 24th Int. Workshop on Languages.

[13] C. Nugteren and H. Corporaal, A Modular and Parameterisable Classification of Algorithms," ES Reports. ISSN 1574-9517. ESR-2011-02.

[14] Wolf, M.E., Computer System Lab, Stanford University.,CA, USA Lam, M.S. "A looptransformation theory and an algorithm to maximize parallelism".