ISSN: 0976-3104
SPECIAL ISSUE: ASPM
Basthikodi and Ahmed

THE IIOAB JOURNAL

ARTICLE OPEN ACCESS

SPECIES GENERATION FOR PARALLEL CODE BY CLASSIFYING THE ALGORITHMS

Mustafa Basthikodi[1] and Waseem Ahmed[2]
[1]Research Scholar, Dept. Of CSE, BIT, Mangalore, INDIA
[2]Dept. Of CSE, HKBKCE, Bangalore, INDIA

ABSTRACT

Many-core and multi-core systems are expected to be major trends for the future decades. In this way of parallel computing, it may become great difficult to choose on which target architecture to execute a certain algorithm or application. Many core machines along with GPUs increased the extensive amount of parallelism. Some compilers are updated to emerging issues with respect to the threading and synchronization. Proper classification of algorithms and programs will benefit largely to the community of programmers to get chances for efficient parallelization. In this work we have analyzed the existing species for algorithm classification, where we discus s the classification of related work and compare the amount of problems which are difficult for classification. We have selected set of algorithms which resemble in structure for various problems but perform given specific tasks. These algorithms are tested using existing tools such as Bones compiler and A-Darwin, an automatic species extraction tool. The access patterns are produced for various algorithmic kernels by running against A-Darwin and analysis is done for various code segments. We have identified that all the algorithms cannot be classified using only existing patterns and created new set of access patterns.

Received on: 24th-Dec-2015
Revised on: 27th-Jan-2016
Accepted on: 04th -Feb-2016
Published on: 17th -Apr -2016

KEY WORDS

Index Terms— Algorithm Classification, Access Patterns, Bones, A-Darwin, Parallel Programming.

***Corresponding author:** Email: mbasthik@gmail.com **Tel:** +919844535720 **Fax:** +91-8242235775

INTRODUCTION

In the past few decades, we have seen a tremendous growth of single-core processor performance. This growth has enabled technology to exist everywhere in the society. To overcome the limitations in the performance of the single-core processor parallelism is exploited. Enabled by the Moore's law, large number of processors per chip (i.e. multi-core) is already a major trend and expected to continue for the next decade [1].

When multi-core is expected to grow 100-core processors by 2020 [1], other trend already enables larger than 2000 cores. This trend (many-core) uses much smaller processing cores, making high throughput parallel processors. An example for such a many-core processor is GPU. Even though many-core processors are suitable for a certain types of applications, other applications may prefer multi-core processors. This creates a heterogeneous environment, with dual types of processors in single system or a single chip.

The multi core architectures enabled performance growth of processors by introducing the parallelism in programs. Because of the heterogeneous computing environment and parallelism, the parallel architectures face challenges in code development and in predicting the performance of the processor. Even though there is amount of research work carried on compilers, parallelization and automatic parallelization, programmers use programming languages such as OpenMP, OpenCL and MPI. Determination of the parallel program performance is needed to check whether the parallelization is required and which part of the program to be parallelized. All the available auto parallelization solutions are not fully automatic. Because of the revolutions in the hardware technologies, such diversified platforms and the need of programming the hardware efficiently has increased the programming models accordingly. The programming models such as Intel TBB[2], OpenACC [3], OpenMP [4], OpenCL [5], Nvidia CUDA [6] and OpenHMPP [7] are some of the models presently available for parallel programmers for writing the applications.

Present trends in the parallel architectures are largely towards the bigger number of processing devices on the chip. This led to the parallel architectures growing as mainstream, with the growth of many specialized multi-core architectures and accelerators. The problem of programming with these architectures effectively using several processing elements [8] is a biggest challenge. There are many approaches [9] to address this issue. The one that

ISSN: 0976-3104

is very challenging is the automatic parallelization. Many difficult applications often spend most of their run time in the nested loops. This is most common in many of the engineering and scientific applications.

The idea of auto parallelization is to free programmers from the error-prone and the time consuming manual parallelization process. Even though the automatic parallelization process has improved its quality in the last few decades, completely auto parallelization of the serial programs by translators remains a biggest challenge because of its need for the complicated program analysis and unknown facts during compilation process. The focus of the automatic parallelization is on programming control structures such as loops, because, maximum of the run times of a program is taken inside some of the loops. For example, an algorithm for the parallel loop identification may be integrated in the parallelization platform as shown in the diagram below which converts automatically serial code in to parallel code.

The front end view of the classification of the algorithm is shown in **Figure– 1**. The figure indicates how classified source code can be parallelized automatically to parallel source code. The figure also shows how the serial code can be compiled in to a parallel code, how to predict the performance. Our focus is on Algorithm Classification part in **Figure– 1**. The overall system will be given with serial code as a input and produces output as parallel program.

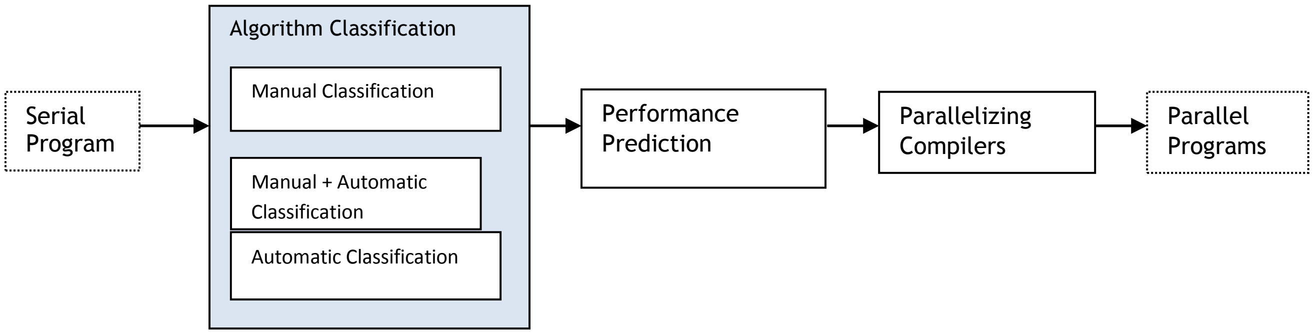


Fig: 1. Front end view of the Algorithm Classification.

In this work, the existing classification of algorithms [10], 'Algorithmic species' that summarizes significant data for parallelizing the algorithm based on classes is studied in depth by compiling various code segments in BONES[11] compiler. The access patterns are generated using ADARWIN and analyzed for different algorithmic kernels.
We retake 'algorithmic species' a access pattern based classification of algorithms [12], and created a increased set of patterns, that summarizes significant data for parallelizing the algorithm based on classes.

The theory behind algorithmic species is subject to the polyhedral model, requiring code to be represented as a set of static affine loop nests. Characterization of array references is introduced with respect to loop nests. Transformations are defined to merge characterizations referring to the same array and to translate them into algorithmic species, allowing classification of non static affine loop nests. The classification is subject to the more detailed abstractions that retain additional performance-relevant information and that take the loop nest structure into account. A tool is modified based on the presented theories to automatically classify program code.

MOTIVATION

The change towards diverse and parallel computing conditions has made programming the challenging tasks. Making complete use of multi-threading and using efficiently memory hierarchy of a processor are the best examples of issues faced by programmers and compilers where programmers looking for a manual solution and compilers looking for an automated solution. We recommend that the classification of algorithms can largely reduce those tasks for compilers and programmers.

We encourage programmers to use an classification of algorithms as a tool to make parallel programming easier. This can help the programmers in identification of problems similar to same class of algorithms and identify available parallelism by applying known parallel patterns.

THE IIOAB JOURNAL
ISSN: 0976-3104

We visualize classification of algorithms as way that facilitates communication between programmers to describe computational issues. This also facilitates design and improves compiler's quality, for example, automatic parallel compilers, source code to- source code compilers, and automatically tuning compilers.

The classification of algorithms that describes the characteristics of algorithm in destination framework shortens many of the architecture related issues. The decisions taken by the underlying compiler for parallelizing the code will be on the data attached in algorithm classes. These will largely solve the problem of compiler designs.
The categorization does not change around time and new feature code may be created when the tools lodge to the changes in the feature hardware. We shun an algorithm categorization as a suitable tool to meet face to clash the futuristic and infinity challenges in parallel computing.

More ever, the algorithm classes may contain in a superior way the algorithm information than is presently available to a compiler that may cause to increase the quality of code. We devise the classification of algorithm to adopt a mean compiler study technique, by bringing the pedigree of algorithm plan into a common place.
Below given are the requirements that to be considered for the goals to be achieved.

(i) Algorithm classes must be automatically extracted from the source code. If not, the program code may have to be extracted manually by the compilers. Ultimately, this is not capable of, considering the concern of classes that may be fault prone and may place a big burden on programmer's head.
(ii) A categorization intend be innate and ethereal to understand. Although we demand classes to be automatically extracted, we contemplate algorithm to be manually used, as a appliance for programmers.
(iii) The algorithm classes must be defined formally. This will guarantee the compilers correctness, and also allows programmers to understand completely the class properties and facilitates automatically extract the classes from source code.
(iv) There should be set boundaries for defining the completeness of algorithm classification, i.e. a single algorithm must belong to one of the predefined classes.

The classes must involve in it, amount of parallelism, the structure, the data reuse information to produce the code efficiently. Also provide circumstances to travel through caching and locality.
Keeping in mind the goals and algorithm classification requirements, we glance at the existing classification of algorithms and check whether our requirements are met.

RELATED WORK

Many algorithmic classifications have been referred before designing the new species. Few of them, with the similar work are highlighted here. Several algorithms have been discussed by many people, such as the ones presented by Allen, Kennedy, Darte and Vivien, Wolf, Lam, and Feautrier [13][14][15]. These algorithms uses of various mathematical notations and tools. Additionally, these do not depend on common representation of the data dependences. The schemes of transformations for common loops in which dependence vectors express precedence constraints on the iterations of loops are presented [16][17]. As per the Wolf and Lam presentation, the dependences extracted from the loop nests must be positive lexicographically. This gives us a simple test for the legality of compound transformations such as, It is legal to have any code transformation that ignores the dependences which are lexicographically positive. The theory of loop transformation is applied to the issue of increasing the degree of coarse-grain or fine-grain parallelism in the loop nests.

Pierre Boulet and Alain Darte surveyed many algorithms on loop parallelization[18], analyzed their dependence representations, generated loop transformations, the required code generation methods, and capability to incorporate different optimization techniques such as the maximal detection of parallelism, the permutable loop detection, the minimization of synchronizations, the code generation easiness, and so on[19][20]. They ended the study by presenting results which are related to the code generation and loop fusion for the specific class of the shifted linear schedules.

Most of the existing automatic parallelization techniques are not fully automatic. The parallel computing introduces challenges in programming and compilation both. The work in [21] introduced the challenges in parallel computing and algorithm classification. It explains the algorithmic species that captures algorithm details from single loop or nested loops and loop bodies. Five access patterns that combine to an algorithmic species are illustrated along with the examples. One of the five access patterns is assigned to each array, accessed in the loop nest. The access patterns combination, of input and output data of the nested loop, forms the species. This

THE IIOAB JOURNAL ISSN: 0976-3104

approach enables to form an unlimited number of species with the use of only five access patterns. The limitations of the approach are highlighted below.

First, the paper uses the polyhedral model, which imposes limits to the program code, which are classified. According to the paper, for loops to be classified must have static and affine loop bounds and the loop bounds must remain constant throughout the scope of the loop. Second, the array accesses are affine and explicit, the pointer arithmetic is not supported in the approach. This also requires that the multidimensional arrays are explicitly addressed per dimension instead of a flat array with an index calculation. Third, the extraction tool ASET used by the author works only for C code, although the species is program language independent. Fourth, The tool ASET considers fully filled iteration space polyhedra in classifying loops and no gaps in the iteration space are supported, which means loops must have stride 1. Fifth, all the program structures (e.g. multiple initializations in loop nest, conditional statements within loops, programs with statements in between algorithmic species, etc) are not supported in ASET which limits the number of statements of various types to be addressed.

PARALLEL ACCESS PATTERNS

The array access patterns element, chunk, neighborhood, full and shared, where, the input arrays are assigned either one of the five patterns whereas the output arrays are assigned either one of the patterns, excluding neighborhood. If an array is accessed from element 0 to 127 it might be that some of the elements in this range are not accessed at all. The same holds for neighborhoods and chunks, partial neighborhoods or partial chunks are still classified as neighborhood or chunk. In addition to these five patterns, We introduce species variant, constant, compare and update to deal with the programming statements such as variable and constant initialization, comparison statements, increment and decrement statements.

```
for(i=0;i<=256;i++){
    for(j=0;j<=256;j++){
      p[i][j]=999;
     }
   }
```

Listing 1: An example for constant initialization in doubly nested loop

We show an example algorithm, in Listing 1, where all loop iterations are independent and these may be executed parallelly. The amount of parallelism is equal to the number of loop iterations and is denoted as parallel(256,256). The loop body consists of initialization statement which assigns constant to array variable p[i][j]. The constant number is input and the two dimensional array p is output in this algorithm. The array is accessed at index 0 to 256 in both the dimensions. When we combine the input and output, their access range and their array access pattern, we find the algorithmic species as:

parallel(256,256)   **constant** --> p[0:256,0:256] | **element**

Here, output array is accessed element-wise and that the combination of array names, ranges and access type defines the algorithmic species of this algorithm.

```
for(k=0;k<=128;k++){
    for(i=0;i<=128;i++){
      for(j=0;j<=128;j++){
      p[i][j]=max(p[i][j],
p[i][k]&&p[k][j]);
      }
    }
  }
```

Listing 2: An example for comparison in triple nested loop

The Listing 2 shows an example of a triple nested loop with comparison and assignment operations. The matrix p elements are compared and the maximum of compared elements is assigned to p[i][j]. The algorithm produces each element of result p by comparing with two elements of same matrix every time. Finding the maximum of two

ISSN: 0976-3104

elements of the matrix is classified as compare and the result produced as element. The resulting species is given below:
parallel**(128,128,128) (p[0:128,0:128], p[0:128,0:128]) | compare →p[0:128,0:128]|**element

The species can be understood as: in order to create an element of p in the range from 0 to 128 we need to compare two elements of matrix p with row and column length 128 each. This pattern-combination of 'compare and producing element' can be performed in parallel a total of 128 times each. Consider the following example algorithm for illustrating the access pattern variant.

```
for(i=0;i<=64;i++){
      for(j=0;j<=64;j++){
         if((i==0)||(j==0))
         V[i][j]=0;
         else V[i][j]=-1 ;
          }
        }
```

Listing 3: An example for comparison and initialization in doubly nested loop

In order to initialize the matrix element V[i][j], the variables i and j are compared with zero . The variable initialization, comparison and constant initializations are classified as the patterns variant, compare and constant. The variables i and j are accessed from 0 to 64 and accordingly the output elements can be produced in parallel. The algorithmic species is constructed as shown below:
parallel(65,65) variant[0:64],constant | compare → V[0:64,0:64] | constant

Here, to produce every element of V, which is constant, is compared with the constant value. Where, V is variable with index ranging from 0 to 65 and is classified as variant.

```
for(i=0;i<8;i++){
    res+=A[i]+B[i+2];
     }
```

Listing 4: An example for reduction to scalar element

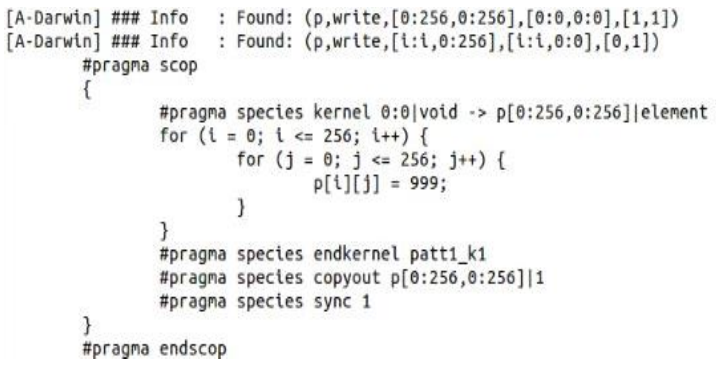
In listing 4, illustrated the example of reducing sum of two vectors in to a scalar quantity. Here, the index of vector A ranges from 0 to 7 and B ranges from 2 to 9. In every iteration, the element of A and B are added and the output variable res is updated. This operation of incrementing the value for every iteration of the loop is classified as the pattern update. The algorithmic species for the listing 4 is constructed as below:
parallel(8) A[0:7] | element ^ B[2:9] | element → res| update

IMPLEMENTATION AND RESULTS

This Bones and A-Darwin along with required gems are installed in quad core system for experimentation. Bones is the source-to-source compiler that is designed based on algorithmic skeletons and the algorithmic species. This compiler takes as input C-code and generates parallel code as output. Target processors include NVIDIA GPUs for CUDA, AMD GPUs for OpenCL and CPUs for OpenCL and OpenMP [22][23]. The Bones compiler is based on CAST C-parser , which is used to parse input source code into the AST (Abstract Syntax Tree) and to generate desired code from the transformed abstract syntax tree.

A-Darwin (short for `automatic Darwin') is automatic extraction tool, which is based on CAST, a C99 parser which allows analysis on 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 program code.

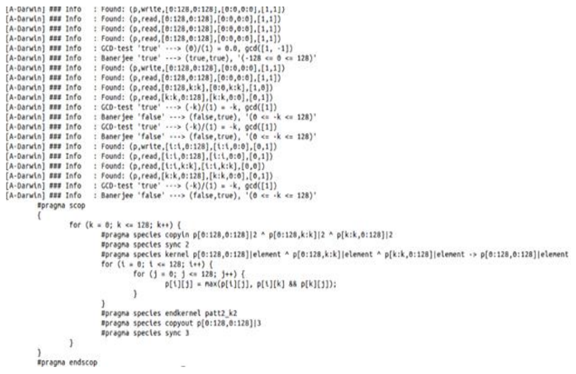
The code segments of various algorithm classes are executed to analyse the output of A-Darwin.



```
[A-Darwin] ### Info   : Found: (p,write,[0:256,0:256],[0:0,0:0],[1,1])
[A-Darwin] ### Info   : Found: (p,write,[i:i,0:256],[i:i,0:0],[0,1])
        #pragma scop
        {
                #pragma species kernel 0:0|void -> p[0:256,0:256]|element
                for (i = 0; i <= 256; i++) {
                        for (j = 0; j <= 256; j++) {
                                p[i][j] = 999;
                        }
                }
                #pragma species endkernel patt1_k1
                #pragma species copyout p[0:256,0:256]|1
                #pragma species sync 1
        }
        #pragma endscop
```

Fig: 2. Output of A-Darwin for listing 1

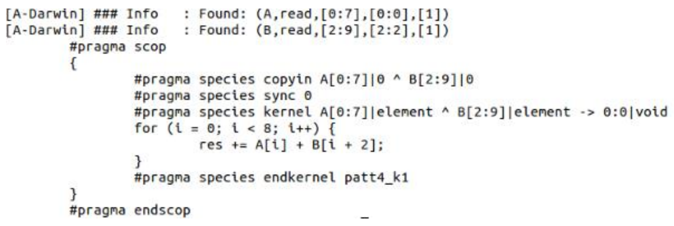
The output for listing1 in **Figure–2** shows that the constant initialization is not taken care. The pattern 0:0 | void -> p[0:256,0:256] | element could be written according to new proposal as: constant -> p[0:256,0:256] | element



```
[A-Darwin] ### Info   : Found: (p,write,[0:128,0:128],[0:0,0:0],[1,1])
[A-Darwin] ### Info   : Found: (p,read,[0:128,0:128],[0:0,0:0],[1,1])
[A-Darwin] ### Info   : Found: (p,read,[0:128,0:128],[0:0,0:0],[1,1])
[A-Darwin] ### Info   : Found: (p,read,[0:128,0:128],[0:0,0:0],[1,1])
[A-Darwin] ### Info   : GCD-test 'true' ---> (0)/(1) = 0.0, gcd([1, -1])
[A-Darwin] ### Info   : Banerjee 'true' ---> (true,true), '(-128 <= 0 <= 128)'
[A-Darwin] ### Info   : Found: (p,write,[0:128,0:128],[0:0,0:0],[1,1])
[A-Darwin] ### Info   : Found: (p,read,[0:128,0:128],[0:0,0:0],[1,1])
[A-Darwin] ### Info   : Found: (p,read,[0:128,k:k],[0:0,k:k],[1,0])
[A-Darwin] ### Info   : Found: (p,read,[k:k,0:128],[k:k,0:0],[0,1])
[A-Darwin] ### Info   : GCD-test 'true' ---> (-k)/(1) = -k, gcd([1])
[A-Darwin] ### Info   : Banerjee 'false' ---> (false,true), '(0 <= -k <= 128)'
[A-Darwin] ### Info   : GCD-test 'true' ---> (-k)/(1) = -k, gcd([1])
[A-Darwin] ### Info   : Banerjee 'false' ---> (false,true), '(0 <= -k <= 128)'
[A-Darwin] ### Info   : Found: (p,write,[i:i,0:128],[i:i,0:0],[0,1])
[A-Darwin] ### Info   : Found: (p,read,[i:i,0:128],[i:i,0:0],[0,1])
[A-Darwin] ### Info   : Found: (p,read,[i:i,k:k],[i:i,k:k],[0,0])
[A-Darwin] ### Info   : Found: (p,read,[k:k,0:128],[k:k,0:0],[0,1])
[A-Darwin] ### Info   : GCD-test 'true' ---> (-k)/(1) = -k, gcd([1])
[A-Darwin] ### Info   : Banerjee 'false' ---> (false,true), '(0 <= -k <= 128)'
        #pragma scop
        {
                for (k = 0; k <= 128; k++) {
                        #pragma species copyin p[0:128,0:128]|2 ^ p[0:128,k:k]|2 ^ p[k:k,0:128]|2
                        #pragma species sync 2
                        #pragma species kernel p[0:128,0:128]|element ^ p[0:128,k:k]|element ^ p[k:k,0:128]|element -> p[0:128,0:128]|element
                        for (i = 0; i <= 128; i++) {
                                for (j = 0; j <= 128; j++) {
                                        p[i][j] = max(p[i][j], p[i][k] && p[k][j]);
                                }
                        }
                        #pragma species endkernel patt2_k2
                        #pragma species copyout p[0:128,0:128]|3
                        #pragma species sync 3
                }
        }
        #pragma endscop
```

Fig: 3. Output of A-Darwin for listing 2

The output for listing2 in **Figure–3** shows that the comparison operation is not concentrated during pattern generation. The pattern p[0:128,0:128] | element ^ p[0:128,k:k]|element ^ p[k:k,o:128]|element -> p[0:128,0:128] | element could be written according to new proposal as: *(p[0:128,0:128], p[0:128,0:128]) | compare -> p[0:128,0:128]*|element



```
[A-Darwin] ### Info   : Found: (A,read,[0:7],[0:0],[1])
[A-Darwin] ### Info   : Found: (B,read,[2:9],[2:2],[1])
        #pragma scop
        {
                #pragma species copyin A[0:7]|0 ^ B[2:9]|0
                #pragma species sync 0
                #pragma species kernel A[0:7]|element ^ B[2:9]|element -> 0:0|void
                for (i = 0; i < 8; i++) {
                        res += A[i] + B[i + 2];
                }
                #pragma species endkernel patt4_k1
        }
        #pragma endscop
```

Fig: 4. Output   of A-Darwin for listing 4

The output for listing4 in **Figure-4** shows that the updation of variables such as increment is not addressed properly. The pattern A[0:7] | element ^ B[2:9]|element -> 0:0 | void could be written according to new proposal as: A[0:7] | element ^ B[2:9]| element -> res| update

THE IIOAB JOURNAL
ISSN: 0976-3104

We have executed and analyzed 50 kernels of 15 algorithmic classes using Bones Compiler and A-Darwin tool. We have found that there are few kernels of various algorithmic classes for which A-Darwin is not considering all the possibilities of code.

Table: 1. Hit Ratio for Algorithm Classed considering multiple kernels of each class

Algorithmic Class	Kernels	Hit Ratio
Arithmetic and Logic	4	75%
1 D Convolution	2	100%
Computer vision	4	75%
MinMax Computation	3	67%
Floyds	2	50%
Warshals	2	50%
Horspool	2	50%
Kruskals	3	67%
Bankers	4	75%
Knapsack	3	67%
K-Means	4	75%
Jacobi	4	100%
2mm	3	100%
3mm	3	100%
Sorting	7	72%

The evaluation of the algorithms listed in **Table– 1** is done by running the specified number of kernels in each algorithmic class mentioned. The third column Hit Ratio is the amount of kernels executed successfully, i.e. the percentage of kernels for which patterns are generated accurately. We have executed 4 kernels of the Arithmetic and Logic algorithmic class, out of which 3 kernels successfully executed resulting 75% of the hit ratio. Similarly two kernels each from 1 D convolution, Floyd algorithm, Warshals algorithm, Horspool algorithm are considered for execuion, resulting the hit ratio of 100%, 50%, 50%, 50% respectively. The three kernels each from MinMax Computation,Kruskals algorithm, knapsack algorithm, 2mm and 3mm algorithmic classes are executed resulting the hit ratio of 67%,67%,67%, 100% and 100% respectively. The four kernels each from computer vision ,Bankers , K-means and Jacobi algorithmic classes are executed resulting the hit ratio of 75%, 75%, 75% and 100% respectively. The seven kernels from sorting algorithmic classes are executed to achieve the 72% of hit ratio.

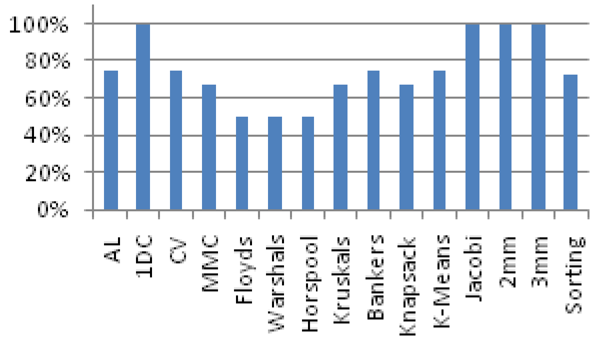


Fig: 5. Graph illustrating the Hit Ratio of kernels for algorithmic classes

The chart in **Figure–5** illustrates the Hit Ratio of the kernels for 15 different algorithmic classes. There are few kernels having statement like comparisons, conditional statements, mathematical functions comes under miss ratio.

CONCLUSION AND FUTURE WORK

The Automatic parallelization is needed to make manual programmer to write the programs without errors and save the time comparatively. In this work, we have analyzed the existing access patterns thoroughly and listed out many of the kernels those cannot be classified based on these patterns. The new access patterns are designed in order to improve the classification ratio.

The A-Darwin source code is modified to incorporate the changes to tool by adding the new functionalities. Currently the tool is working for the additional access patterns. The work is carried to include more number of mathematical functions which maximize the hit ratio for algorithmic classification. The present work is also based on the array references. So, the final work will be to classify the algorithms based on pointer references.

CONFLICT OF INTEREST

Authors declare that there is no conflict of interest.

ACKNOWLEDGEMENT

None.

FINANCIAL DISCLOSURE

No financial assistance received during my research work.

REFERENCES

[1] B Catanzaro, A Fox, K Keutzer, et al. [2010], Ubiquitous Parallel Computing, *IEEE Micro*, 30:41–55.

[2] C Pheatt. [2008] Intel threading building blocks, J. Comput. Sci. Colleges, 23(4):298–298.

[3] S Wienke, P Springer, C Terboven, D an Mey.[2010] OpenACC:First experiences with real-world applications, in Proc. 18th Int. Conf. Parallel Process., 859–870.

[4] L Dagum and R Menon. [1998] OpenMP: An industry standard API for shared-memory programming, *IEEE Comput. Sci. Eng.*, 5(1): 46–55.

[5] JE Stone, D Gohara, and G Shi. [2010] OpenCL: A parallel programming standard for heterogeneous computing systems, *Comput.Sci. Eng.*,12(3): 66.

[6] J Nickolls, I Buck, M Garland, and K Skadron.[2008] Scalable parallel programming with CUDA, ACM Queue, 6(2): 40–53.

[7] R Dolbeau, S Bihan, and F Bodin.[2007] HMPP: A hybrid multicore parallel programming environment, in Proc. Workshop General Purpose Process. Graph. Process. Units.

[8] Wolf ME.[1998] A loop transformation theory and an algorithm to maximize parallelism, Computer System Lab, Stanford University., CA, USA Lam, M.S.

[9] Pierre Boulet, Alain Darte. [1997] The parallelization Algorithms for loops: parallelism extraction to code generation. *Technical Report* 97–17, LIP,ENS-Lyon, France.

[10] PJJM Custers. [2013] Algorithmic Species: Classifying Program Code for Parallel Computing, Electronic Systems group, Eindhoven University of Technology.

[11] H Corporaal, C Nugteren. [2012] Introducing 'Bones': A Paralleliz-ing Source-to-Source Compiler Based on Algorithmic Skeletons, GPGPU-5: The Workshop on General Purpose Processing on Graphics Processing Units. ACM.

[12] Kathryn S McKinley, Ken Kennedy.[1994] Improving data locality and maximizing loop parallelism via loop fusion and distribution" ,Springer.

[13] M Wolfe. [2010] Implementing the PGI Accelerator Model, in GPGPU-3: Workshop on the General Purpose Processing on Graphics Processing Units. ACM.

[14] Paul Feautrier.[1992] Some of the efficient solutions to the affine scheduling problem, *International Journal on Parallel Programming* :313–348.

[15] Paul F.[1992] Some of efficient solutions to the affine scheduling problem, Part II: multi dimensional time, *International Journal on Parallel Programming*:389–420.

[16] Frederic Viven, Alain Darte. [1996] A Optional line and medium grain parallelism detection for polyhedral reduced dependence graphs, PACT'96, Boston, *IEEE*.

[17] Johnson R, Beck K. [1994] Patterns generating architectures, The European Conference on Object- Oriented Programming: 821. Springer, 139–149.

[18] K Kennedy, R. Allen. [2002] Optimization of Compilers for Modern Architectures, Morgan- Kaufman.

[19] Thomas Holl, Dirk Heuzerot, Gustav Hogstrom. [2003] Automatic design pattern detection: IWPC-03:IEEE , International Workshop, page 94,Washington, DC, USA.

[20] Monica S Lam, Micheal E Wolf. [1991] Loop transformation theory and an algorithm to maximize parallelism. *IEEE*, Parallel Distributed Systems :452–471.

[21] CW Kessler, J Enmyren. [2010] SkePU: A Multi-backend Skeleton Programming Library for Multi-GPU Systems,in HLPP '10: 4th International Workshop on the High-level Parallel Programming and Applications. ACM.

[22] Mustafa B, Waseem Ahmed. [2014] Parallelization Approaches using OpenMP for Strassens Matrix Multiplication and Canny Edge Detector, International Journal of Information Processing(IJIP),8(4):89–97,ISSN: 0973–8215.

[23] Mustafa B, Waseem Ahmed. [2015] Parallel Algorithm Performance Analysis using OpenMP for Multicore Machines, International *Journal of Advanced Computer Technology(IJACT)*, 4 (5):ISSN: 2319–7900.

THE IIOAB JOURNAL ISSN: 0976-3104

ABOUT AUTHORS

Mustafa Basthikodi *was born in Mangalore, India, in 1979. He received the B.E. degree in Computer Science and Engineering from the Mysore University, Mysore, India, in 2001, and the M.E. degree in Computer Science and Engineering from the Bangalore University, Bangalore in 2008. Currently Pursuing PhD in High Performance Computing and Embedded Systems from Visvesvaraya Technological University (VTU), Belgaum. In 2001, he joined the Department of Computer Science & Engineering, PACE, Mangalore, as a Lecturer, and worked till 2006. From 2006 to 2008, He worked as Senior Lecturer in Department of Computer Science & Engineering in SJBIT, Bangalore. In 2008, He joined IBM as Senior Software Engineer and worked till 2010. Since 2010, He is working as Associate Professor and Head, Department of Computer Science & Engineering, in BIT, Mangalore. He has published in various National and International Conferences. He has received few best technical paper awards and also Best performer award in industry. He has also worked as Technical Programme committee member for the various conferences. He is a Life Member and resource person for Computer Society of India. His subjects of Interest include High Performance Computing & Embedded Systems, Green & Cloud Computing, Compiler construction tools & technologies.*

Dr.Waseem Ahmed *is currently a Professor in the Department of CSE at HKBK College of Engineering, Bangalore. Prior to this he has been served at different capacities in academic/work environments in the USA, UAE, Malaysia, Australia and India. He obtained his BE from RVCE, Bangalore, MS from the University of Houston, USA and PhD from the Curtin University of Technology, Perth, Western Australia. He has published extensively in various reputed International Journals and Conferences. He is a reviewer for various IEEE/ACM Transactions and magazines. His current research interests include heterogeneous computing in HPC and embedded Systems. He is a member of the IEEE.*