HPC Based Algorithmic Species Extraction Tool for Automatic Parallelization of Program Code

Mustafa Basthikodi, Ahmed Rimaz Faizabadi, Waseem Ahmed

ABSTRACT--- Without a doubt, multiple core processors have become primary stream in parallel computing. Therefore, future generations of applications pivotal role will be played by parallelism. It must be noted that, the compilers and programmers could immensely benefit from a program source code classified in a structured manner. Such a classification surely helps programmers to identify parallelization scopes or reasoning about the program code, and associate with other programmers. To address the challenge of parallel programming, we worked on source-to-source compiler Bones and developed species extraction tool extended A-Darwin to ease parallel programming. In the work done, we present 'Algorithmic Species', a new algorithm classification, that encapsulates required information for parallelization in classes, and embeds memory transfer requirements for optimization of communication on heterogeneous platforms. The evaluation of algorithmic species and the validation of extended A-Darwin are done by testing the tool against the benchmark suit HPCC. The unique approach is developed to generate code automatically for parallel target machines.

I. INTRODUCTION

For the most part, a significant number of the computer software are produced for serial computation. Be that as it may, with the development of multi-core processors, parallel design is promptly accessible on practically every PC and the product should exploit the benefits of parallel computing. There has been a gigantic progress in chip innovation. The clock rate of the chip has expanded from 40MHz to 2.5GHz, in the meantime processors are fit for executing various instructions in a similar cycle. The normal number of CPIs (Cycles per direction) has progressed. So this parallel computing has made an enormous effect on a variety of areas going from computational simulations for engineering and scientific applications to business applications in information mining and transaction processing.

The hypothesis supporting algorithmic species is liable to polyhedral model, expecting the source code to be described as an arrangement of static relative nested loops. The characterizations of array references are acquainted with deference with nested loops. Changes are stated to consolidate characterization referring to a similar array and then to make an interpretation of those into algorithmic species, permitting classifying of non-static relative loop nests. Classification is liable to more point by point deliberations that hold extra execution important data and that consider the structure of loop nest. A tool is altered in light of the exhibited hypotheses to consequently classify the program code.

Currently an array of areas are investigated which include frameworks for new programming, language extensions, auto tuning and optimizations of compilers and auto parallelizing the source to source compilers. A program code that is represented in a structured manner can be of great benefit for compilers and programmers targeting parallel and heterogeneous systems, which is a set of rules instrumental in dictating the class of code based upon a set of properties. The examples for the existing classifications of algorithm are given as Berkeley dwarfs [1], Galois classification system [2] and the algorithmic skeletons [3]. We propose to present a algorithmic species, a novice classification of algorithm on the basis of polyhedral model [4] that realizes the following goals:

- Programmers focusing on parallel processors will have the capacity to reason about their program code by methods for algorithm classes
- Specialists and compiler architects will have the capacity to plan their compilers in view of this classification

As it were, species of algorithm can be viewed as changing polyhedral data into a classification of algorithm. Algorithmic species is an algorithmic classification that satisfies the afore said objectives. Every individual classes are formally stated, straightforward, pertinent to any loop nest that is affine, and depict program source code in detail, catching structure of parallelism, open doors for information reuse, region data, and information sizes. Moreover, the utilization of algorithmic species in various tools and models is recognized:

- Removal of species naturally from C source code making use of a tool
- A compiler that is source-to-source and skeleton-based
- An expectation model for species based execution

The first inadequacy for the most part influences application developers who are new to parallel models and simultaneous programming, while the second deficiency generally influences smart software engineers who are utilizing compilers to play out the underlying parallelization and will additionally enhance the subsequent code. The third

Revised Manuscript Received on July 10, 2019.

Mustafa Basthikodi, Dept.of CSE, Bearys institute of Technology, Mangalore, Karnataka India. (mustafa@bitmangalore.edu.in)

Ahmed Rimaz Faizabadi, Dept.of CSE, Bearys institute of Technology, Mangalore, Karnataka India. (ahmedrimaz@bitmangalore.edu.in)

Waseem Ahmed, Faculty of Computing and IT, King AbdulAziz University, Jeddah, Saudi Arabia. (waseem.pace@gmail.com)

deficiency influences a wide range of clients. A classification of algorithm is utilized to manage a compiler in light of algorithmic skeletons. A method is displayed to consequently produce proficient and intelligible parallel code belong to the parallel designs (with an emphasis to GPUs). We construct this strategy with respect to ’algorithmic species’, a classification of algorithm of program code in view of the polyhedral model. Algorithmic species typify data, for example, memory access patterns and information re-utilize. Algorithmic species frame the foundation of our methodology, which incorporates a device to naturally extricate species from affine static loops(ASET) and then source-to-source compiler in view of skeletons (Bones).

- A distinctive integration of a compiler which is based of bones compiler with an algorithm classification also referred as algorithmic species. This skeleton based compiler could be used in flows of compilation that are fully automatic as manually identification of skeletons is no longer required.
- Host-accelerator exchanges (CPU/GPU) in a ASET and expanded bones are optimized and introduced with new optimizations, skeletons and targets that include caching of registers coarsening of threads and transfers that are of zero copy.
- We debate and illustrate the advantages of our distinctive approach by creating OpenMP source code for the benchmark suit HPCC.

The primary objective of this work is to build an automatic species extraction tool which works for both array references and pointer references. The tool is the extended version of A-Darwin which extracts the algorithmic species to automatically classify the program code. The tool also covers the following additional functionalities:

- Classification of program code in light of pointer references
- Classification of Conditional expressions
- Classification of Incremental statements
- Classification of Mathematical functions
- Classification of User defined functions
- Classification of Variants and Constants

The tool generates the code in more readable form, allowing users to further optimize the algorithms. The work proposes a fully automatic compiler and does not require any code restructuring. The tool also to be optimized for efficient storage of data in memory and reduction of execution time.

II. EASE OF USE

Identified as working on design technologies for parallelism [5][6][7]. The pattern languages are proposed to direct software engineers by giving descriptions of much of the time happening issues. They commonly provide patterns at various levels, yet regularly begin at a high level of abstraction. An OPL is used as illustration for the pattern language [5][7], which utilize motifs (termed in OPL as computational patterns), that is a first classification step. A moment step includes auxiliary patterns, which depict the association of patterns related to computations. In the event that we accept the yield matrix S belong to stencil calculation (listing 2.2) to be utilized as contribution to the grid and vector multiplication (listing 2.1), we may classify the arrangement of illustrations as "pipe-and-channel" structural example. A pattern language besides gives examples to parallel programming technologies. For the illustrations we can choose the "information parallelism" algorithmic procedure, a "loop parallelism" usage methodology, and the parallel execution design such as SIMD. Albeit most descriptive than motifs, an OPL is as yet planned for the purpose of manual classification, making it unsatisfactory to meet both objectives. Work on algorithmic skeletons [3] have prompted a substantial count of algorithm groupings. A summary of traditional skeletons is identified in a skeleton review [8], this overview of regular algorithmic

Skeletons finish up with a general grouping, catching numerous skeletons belong to works done already. We utilize this classification to assess the cases. In the grid and vector multiplication (listing 2.1), every calculation S[m][n] * v[n] brings about a halfway result of a individual component of vector r, which needs recombination. This fits well the “divide-and - conquer” or "recursively partitioned" skeleton. The stencil calculation given in listings registers an outcome specifically, preparing it to fit the "queue of task" or "homestead" skeletons. Such established skeletons are extremely natural, however give no mechanization, incompleteness ensures, no definition in formal way, and these are excessively coarse-grained, making it impossible to reach our objectives. Later contemporary skeleton work [9][10][11][12] utilizes lower level abstraction characterizations.

The skeletons which are used as example are map-array,map-overlap,map-reduce, map and reduce[11] or pixel-to-global, neighborhood-to-pixel, pixel-to-pixel, and bucket handling [8]. Identified with the current skeleton work are idioms [13], a classification framework characterizing 6 classes: stencil, scatter, stream, gather, transpose and reduction. While classifying examples utilizing contemporary idioms and skeletons, we locate the accompanying outcomes. The 2D Jacobi stencil calculation of posting 2.2 groups as "map-overlap" [11],"neighborhood-to-pixel" [11], or as the comparable "stencil" [13]. In any case, these classification methods can’t arrange the full grid-vector multiplication illustration, in spite of the fact that the example can even now be classified somewhat: the calculation in the inward loop j can be delegated "reduce"[11], "scalar diminishment" [10] or "reduction" [13]. Contrasted with established skeletons, contemporary skeletons and idioms are as of now a superior fit for our objectives: formally they are characterized in few cases[10], and every so often give tools to automation[14]. All things considered, we can’t recognize a solitary skeleton grouping which satisfies all necessities, lacking angles, for example, fulfillment and granularity for instance. The numerical portrayals of code, for example, Æcute [15], the polyhedral model [16], and the SUIF loop change detailing [17] are dissected keeping in mind the end goal to get appropriate portrayals. The compiler directives, for example,OpenACC and OpenHMPP are firmly coupled to program code. In spite of the fact that directives which are not entirely considered algorithm characterizations, they have the likelihood to catch data on code sections. OpenACC for instance issued by different compilers to indicate locales of code which are to be offloaded to the accelerators [18]. It is utilized by for instance HMPP Workbench [19] and PGI Accelerator [20]. As more of OpenACC mandates are updated to the program code, an expanding measure of data will wind up noticeably accessible to the compiler.

The algorithm classification is intended for software engineers and tools to catch and reason about parallel algorithms. As indicated by [11], the classification is at first proposed to be utilized to address the test of parallel programming, and expectation of performance for heterogeneous and parallel frameworks. With a specific end goal to address these two difficulties, algorithm classification is described in the work done. A classification utilizes a constrained vocabulary and a very much defined syntax, making a modular classification. Furthermore, the classification is termed as parameterisable. Both the parameterisability and modularity of the classified algorithm make it conceivable to empower an extremely fine-grained and the classification which is generally applicable.

The "Algorithmic Species" is presented [13], which epitomizes pertinent data for parallelization in classes, and inserts memory exchange prerequisites to streamline communication on heterogeneous stages. Work is assessed by physically characterizing the species of algorithms in two genuine applications and benchmark sets. For identification of algorithmic species in source code, the ASET is planned. This algorithmic species is strong base for present and upcoming work based on parallel technologies, fit for tackling numerous issues identified with parallel computing.

The updated hypothesis of algorithmic species is exhibited in [21]. The hypothesis comprises of a five-tuple portrayal of every single array references and respective joining operations. Second, an augmentation of this hypothesis termed SPECIES+ is introduced, giving more itemized six-tuple portrayal. With that, it is conceivable to hold important access patterns data not caught by first species of algorithms, for example, row-major versus column-major grid accesses. Both the new speculations are actualized as a tool, empowering the program code classification.

A model [22] is introduced to anticipate the execution of a stated application on processor having many/multi cores. Considering the complexities involved in programming these processors, this model does not need program code to be accessible for the objective processor. This is as opposed to available execution forecast methods, for example, scientific models and test systems, which expect code to be accessible and enhanced for the architecture targeted. To empower execution expectation before algorithm usage, algorithms are characterized utilizing a current classification of algorithm. For every class, a particular occurrence of roofline demonstrate is made, bringing about another class related show. This model, termed as the boat hull model, empowers execution expectation and choice of processor before the improvement of specific code related to design. The boat hull structure is exhibited utilizing GPUs and CPUs as target designs. This demonstrates execution is precisely anticipated for a case genuine application. Maintaining the Integrity of the Specifications

The template is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin in this template measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings, and not as an independent document. Please do not revise any of the current designations.

III. ALGORITHMIC SPECIES

The discussions done in previous section show that, our requirements are not fulfilled by existing algorithm classifications. Therefore, algorithmic species, an extended classification introduced in this work. The classification characterizes species at a lower deliberation level, by classifying nested or individual relative loops, such as loops with relative array accesses and relative static loop control. The species are inspired upon the classification of skeletons done before. Here we discuss the construction of algorithmic species based on the array access patterns by giving examples.

```
for(m = 0;m < 128;m++) {
for(n = 0;n < 256;n++){
r[m][n] = 3 * s[m][n];
}
}
```

Listing 3.1 : Embarrassingly parallel algorithm

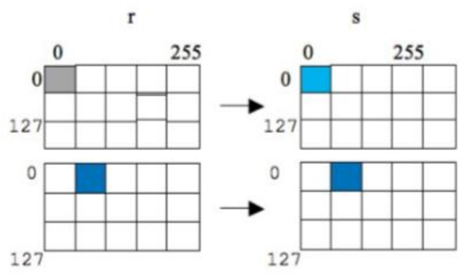

Figure 3.1: Demonstration of initial two iterations

```
for(m = 0;m < 50;m++){
  res[m] = 0;
  for(n = 0;n < 100;n++){
    res[m] += ip[m][n] * v[m];
  }
}
```

Listing 3.2 : Matrix-vector multiplication

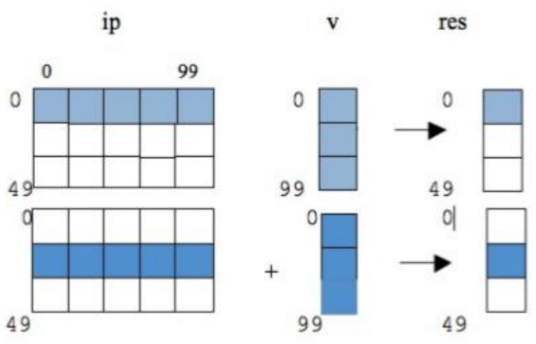

Figure 3.2: Demonstration of initial two iterations for Listing 3.2

In listing 3.1, given a case of a loop nest where iterations can be executed freely. In each iteration of this illustration a component of array S[][] is perused and increased by 3, to deliver a subsequent component of an array R[][].The arrays are accessed from lists 0 to 127 and from 0 to 255 in the first and second dimensions separately. At the point when the names of the arrays are consolidated alongside data, the outcome is acquired as appeared in the primary column of the Table 3.1. The outcome is translated as: on each iteration of the dimensions 0 to 127 and 0 to 255, single element is required belong to input array S[][] to create single element belong to yield R[][]. The listing 3.2 spreads the grid-vector multiplication example code. Here, yield of a solitary element of res[] requires a whole row belong to array ip[][] and the entire array v[]. Those accesses are recognized as: chunk for row access of ip[][] and full for entire access of v[]. A subsequent algorithmic species is appeared in second row of Table 3.1, which is translated as: to yield a solitary element out of the aggregate 50 elements in res [], the whole array v[] of size 100 and a lump of data in next dimension of ip[][] are required. Presently, consider the Jacobi stencil calculation illustration given in listing 3.3. To get a solitary element of array d [], a neighborhood of 3 components from s [] is required. A neighborhood access and a chunk access contrast from every other in a way that the last suggests cover between consequent repetitions, similar to the reality in the case of stencil operations. The total classification is found in the third row of the Table 3.1, where the measure of the neighborhood is given extending from −1 to +1. In every one of the classifications of the Table 3.1, the measure of parallelism is demonstrated, for example, PARALLEL (128,256), which is equivalent to the measure of loop cycles

```
for(m = 1;m < 128 − 1;m++){
    d[m] = 0.78 ∗ (s[m − 1] + s[m] + s[m + 1]) ;
}
```

Listing 3.3 : 2D Jacobi stencil (1D version)

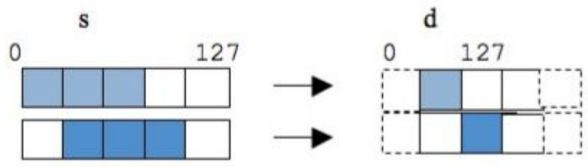

Figure 3.3: Demonstration of initial two iterations for Listing 3.3

```
for(m = 0;m < 8;m++){
    res[0]+ = x[m] + y[m + 2];
}
```

Listing 3.4 Reduction to a scalar example

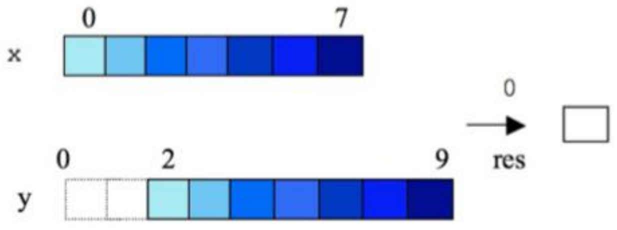

Figure 3.4: Demonstration of initial two iterations for Listing 3.4

```
for(m = 0;m < 2;m++){
    for(n = 0;n < 2;n++) {
A[m][n] =
B[2∗m][2∗n] + B[2∗n + 1][2∗n] + B[2∗m][2∗n + 1] + B[2∗m + 1][2∗n + 1];
    }
}
```

Listing 3.5: 2D chunk access example

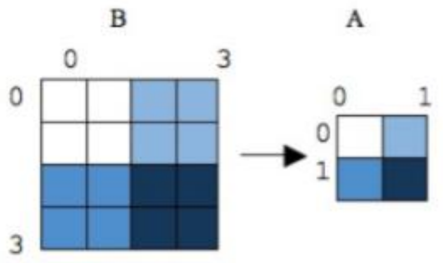

Figure 3.5: Demonstration of 4 iterations for Listing 3.5

In the decrease to scalar case of listing 3.4, we can find that, the outcome res[0] is produced using the commitment of each input element of arrays x[] and y[]. The outcome is considered as shared in view of the fractional contribution. This conduct is caught in the classification given as algorithm species in fourth row of Table 3.1. The offset access to array y[] is caught by indicating the reaches from 2 till 9.The classification of example in listing 3.6 requires a 2x2 tile from B[][] to acquire a solitary yield in A[][]. The tile is classified as 2-dimensional chunk access, creating classification as given in the fifth line of Table 3.1.

Example	Algorithmic species
Listing 3.1	PARALLEL (128,256) s[0:127, 0:255] | element → r[0:127,0:255]| element
Listing 3.2	PARALLEL (50,100) ip[0:49,0:99]|chunk(-,0:99) ∧v[0:99]|full → res[0:49]| element
Listing 3.3	PARALLEL (127) s[1:126]|neighbourhood(-1:1) → d[1:126]| element
Listing 3.4	PARALLEL (8) x[0:7]| element ∧ y[2:9]| element → res[0:0]| shared
Listing 3.5	PARALLEL (2,2) B[0:3,0:3]|chunk(0:1,0:1) → A[0:1,0:1]| element

Table 3.1: Classification of examples as algorithm species

Algorithmic species is a classification that catches lower level calculation descriptions belong to nested loops or singular loops and statements in loop's bodies. The primary key to the algorithmic species methodology is that each array, referenced in the nested loop which is classified, is appointed as one of access patterns. A group of access patterns, belong to input and yield arrays from the nested loop, and after that structures the species.

The algorithmic species extraction tool extended A-Darwin is developed that takes a sequential C code as input and automatically generates the species-annotated C code. 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.

As the Algorithm 1 illustrates, while both matrices X and Y are zero, the access pattern is classified as update. When matrix X is non zero and Y is zero, the access pattern is classified as element. When X is zero and Y is non zero, the access pattern is classified as full. When X and Y both are non-zero, the access pattern is classified either as chunk or as neighborhood in light of regardless of whether there exists a re-use between the diverse accesses to array. And when there is a constant, the access pattern is classified as constant. The algorithm takes as input the access descriptions for the entire array and gives the access patterns Pt for these inputs. The variable Sp will contain the respective species for the array patterns.

Algorithm 1 Deriving the access patterns for arrays with constant initialization

```
Input: Descriptions of array access
Output: The access patterns for the inputs
Pt = ø
repeat
Sp = ø
$(\mathbf{X}_p;\mathbf{Y}_p;\vec{C_p})$ ← Access description of array p
switch $(\mathbf{X_p};\mathbf{Y_p})$ do
    case$X_p$ = 0 &$Y_p$ = 0
        $S_p$ ←"update"
    case$X_p$ ≠ 0 &$Y_p$ = 0
        $S_p$← "element"
    case$X_p$ = 0 &$Y_p$ ≠ 0
        $S_p$← "full"
    case$X_p$ ≠ 0 &$Y_p$≠ 0
        if (equation 2 holds good for array p) then
        Sp ← "neighbourhood"
        else
        Sp ←"chunk"
        end
        if (equation 1 has constant ~c) then
            Sp←"constant"
        end
    end
endsw
Pt ←PtUSp
until all patterns are derived;
Result: Pt
```

The Algorithm 2 gives the idea how the patterns for the conditional statements are retrieved from source code. The algorithm takes the input as array access descriptions and produces as output the parallel patterns for the conditional statements. The function get_if in the algorithm scans the source code for any occurrences of conditional statements by using pattern matching step. If the comparison operation is found, then the species compare is added to the pattern to be returned from the function.

I	II	III	IV	V	VI	VII
Kernels	Species	Classification	Functions	Conditional Statements	Pointers	Constants
donecpu.c	4	4/4 (100%)	✓	✓	✓	✓
dtstdgemm.c	6	6/6 (100%)	✓	✓		✓
sonecpu.c	4	4/4 (100%)	✓	✓	✓	✓
dstream.c	13	11/13 (85%)	✓	✓	✓	✓
fbcrand.c	8	7/8 (88%)	✓			✓
ftstfft.c	6	5/6 (83%)	✓		✓	✓
fwrapfftw.c	4	4/4 (100%)	✓	✓	✓	✓
pmem.c	7	7/7 (100%)	✓	✓	✓	✓
psclapack.c	7	6/7 (86%)	✓		✓	✓
rbuckets.c	5	3/5 (60%)	✓			✓
rutility.c	7	6/7 (86%)	✓			✓
hmatgen.c	4	4/4 (100%)	✓			✓
hpdmatgen.c	4	4/4 (100%)	✓			✓

Table 4.1: Number of kernels in each algorithmic class of HPCC and its state of execution

Algorithm 2 Deriving the patterns for array access with conditional statements

```
Input: Descriptions of array access
Output: The conditional statements access patterns
Function get_if
scope_code=get input from source code
X ← scope_code.scan(/\(?\[?\w+\]?\]/)
flag ← false
for each |a| in X do
    if a contains '('
        if previous element of a== "if"
            a+=[start]
            flag ← true
        end if
    end if
    if flag == true and a == ')'
        replace ')' with "[end]"
    end if
    if a==comparison operator
        a ← '='
    end if
end for
Sp ← generate species code from pattern species
Y ← get start and end intervals from Sp
Pt ← pattern from S
for each |a| in Y do
    if a== "end:end"
        Pt+=} | compare
    end if
end for
replace pattern part in species to Pt
remove intervals "start:start" and "end:end"
end function
```

IV. EXPERIMENTATION AND RESULTS

To validate the work done against the standard benchmarks, the required running environment is setup by making ready hardware and installing the required software. The Bones, a parallel compiler, extended A-Darwin and the required gems are configured and installed in the framework containing quad core systems for experimentation and analysis.

In order to analyze and evaluate the usage of hypothesis belong to algorithmic species and their extractions automatically, the validation of extended A-Darwin is done by testing the code against the four benchmark suits such as HPCC . The unique approach is developed to generate code automatically for parallel target machines.

The 13 modules are taken from the 7 kernels of HPC Challenge benchmarks, such as HPL, STREAM, Random-access, PTRANS, FFT, DGEMM and b_eff. All the selected modules contain nested loops and tested with extended A-Darwin for the classification. The results after the classification tabulated in Table 5.3. Many of the modules are classified with 100% hit ratio. The least hit ratio we achieved is 60% for rbuckets.c module, which had data dependencies carried from the previous loops. The 71 species are classified successfully out of 79, achieving 90%

success ratio overall. The columns IV through VII shows the recognition of programming constructs such as functions, conditional statements, pointers, constants which are in the body of the loops. The executed kernels produce different access patterns according to the variables, function call, built-in function and the calculation used in a particular kernel. The results analyzed in Table 4.1 is represented graphically in Fig. 4.1 for more understanding about success rate of the benchmark execution.

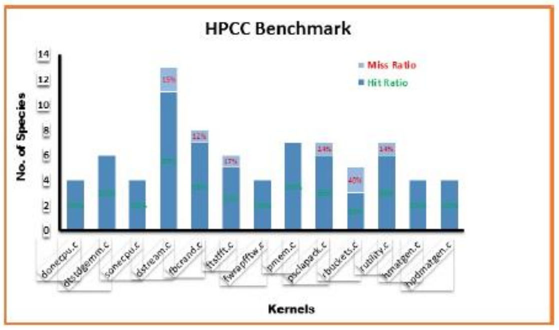

Figure 4.1: Graphical analysis of execution of HPCC Kernels

V. CONCLUSIONS

The parallel computing has played a vital role in improving the performance of applications. In order to make manual programmer right programs that are free from errors and comparatively save time, automatic parallelization is needed. In this work, ’algorithmic species’ is presented, which is an algorithmic classification that captures algorithmic details of low level and presents them using few easy to understand access patterns. This algorithmic classification is designed to capture and reason about parallel algorithms for programmers.

The future work is identified in the direction of irregular algorithms, i.e. algorithms that are composed of data structures such as trees, graphs, matrices that sparse. By classifying such irregular algorithms, the insights in to structures of data locality and parallelism could help in producing efficient code for programmers and compilers. Moreover, the extension of the algorithmic species with additional information and information about inter-species could help programmers and compilers in fusing the multiple species.

REFERENCES

1. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek et al., “A view of the parallel computinglandscape,” Communications of the ACM, 2009.
2. K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H.Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo et al., “The tao of parallelism inalgorithms,” in ACM Sigplan Notices, vol. 46, no. 6. ACM, 2011, pp. 12–25.
3. M. I. Cole, Algorithmic skeletons: structured management of parallel computation. Pitman London, 1989.
4. L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache, “Iterative optimization in the polyhedral model: Part i, one-dimensional time,” in Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society, 2007,pp. 144–156.
5. K. Keutzer and T. Mattson, “A design pattern language for engineering (parallel) software,” Intel Technology Journal, vol. 13, no. 4, 2010.
6. B. L. Massingill, T. G. Mattson, and B. A. Sanders, “A pattern language for parallel application programs,” in European Conference on Parallel Processing. Springer, 2000, pp. 678–681.
7. T. Mattson, B. Sanders, and B. Massingill, “Patterns for parallel programming. The software patterns series; ed. by john vlissides,” 2004.
8. D. K. Campbell, “Towards the classification of algorithmic skeletons,” REPORTUNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS, 1996.
9. W. Caarls, P. P. Jonker, and H. Corporaal, “Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications,” in Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. IEEE, 2006, pp. 9–pp.
10. C. Nugteren and H. Corporaal, “A modular and parameterisable classification of algorithms,” Eindhoven University of Technology, Tech. Rep. ESR-2011-02, 2011.
11. R. M. Stallman, “Gnu compiler collection internals,” Free Software Foundation, 2002.
12. C. Nugteren and H. Corporaal, “Introducing bones: a parallelizing source-to-source compiler based on algorithmic skeletons,” in Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units. ACM, 2012, pp. 1–10.
13. P. Custers, “Algorithmic species: Classifying program code for parallel computing,” Master’s thesis, Eindhoven University of Technology, 2012.
14. L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole, “An idiom-finding tool for increasing productivity of accelerators,” in Proceedings of the international conference on Supercomputing. ACM, 2011, pp. 202–212.
15. L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. Kelly, “Deriving efficient data movement from decoupled access/execute specifications.” HiPEAC, vol. 9, pp. 168–182, 2009.
16. P. Feautrier, “Dataflow analysis of array and scalar references,” International Journal of Parallel Programming, vol. 20, no. 1, pp. 23–53, 1991.
17. M. E. Wolf and M. S. Lam, “A data locality optimizing algorithm,” in ACM Sigplan Notices, vol. 26, no. 6. ACM, 1991, pp. 30–44.
18. A. Cohen, A. F. Donaldson, M. Huisman, and J.-P. Katoen, “Correct and efficient accelerator programming (dagstuhl seminar 13142),” in Dagstuhl Reports, vol. 3, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.
19. R. Dolbeau, S. Bihan, and F. Bodin, “Hmpp: A hybrid multi-core parallel programming environment,” in Workshop on general purpose processing on graphics processing units (GPGPU 2007), vol. 28, 2007.
20. M.Wolfe, “Implementing the pgi accelerator model,” in Proceedings of the 3rd Workshop
21. on General-Purpose Computation on Graphics Processing Units. ACM, 2010, pp. 43–50.
22. C. Nugteren, R. Corvino, and H. Corporaal, “Algorithmic species revisited: A program code classification based on array references,” in Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on. IEEE, 2013, pp. 1–8.
23. C. Nugteren and H. Corporaal, “The boat hull model: enabling performance prediction for parallel computing prior to code development,” in Proceedings of the 9th conference on Computing Frontiers. ACM, 2012, pp. 203–212.