Algorithmic Species: Classifying Program Code for Parallel Computing

P.J.J.M. Custers,
Electronic Systems group, Eindhoven University of Technology
E-mail: p.j.j.m.custers@student.tue.nl

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. Emerging processor architectures such as multi-cores and many-cores exploit an increasing amount of parallelism, requiring programmers and compilers to deal with issues such as threading, concurrency and synchronization. We advocate that programmers and compilers can greatly benefit from a structured classification of program code. Such a classification can help programmers to find opportunities for parallelization or can serve as front-end to performance prediction models or parallelizing-compilers for example.

In this paper, we introduce a new algorithm classification, ‘Algorithmic Species’, which encapsulates relevant information for parallelization in classes, and embeds memory transfer requirements to optimize communication on heterogeneous platforms. We evaluate our work by manually classifying 134 algorithmic species in a benchmark set and two real-life applications. For the automatic identification of algorithmic species in program code we design ASET, which is able to automatically identify 99% of the algorithmic species and to automatically extract memory transfer requirements. We conclude that algorithmic species are a solid base for current and future work on parallel programming, capable of solving many problems related to parallel computing.

Index Terms—Parallel Programming, Algorithm Classification, Polyhedral Model

I. INTRODUCTION

For the past decades single-processor performance has increased exponentially. Following Moore’s law, smaller feature sizes resulted in processor chips with more and faster transistors. Although Moore’s law is still valid today, the growth in single-processor performance came to a halt around 2004, mainly due to power dissipation [1].

Processor performance growth was re-enabled by exploiting parallelism in programs by multi- and many-core architectures. With the introduction of many-core processors, used as accelerators (e.g. a GPU), the computing platform became heterogeneous. Both the parallelism and the heterogeneity introduce challenges in programming for these parallel architectures. The available parallelism must explicitly be programmed as where a heterogeneous platform requires memory communication and synchronized execution between host and accelerator. The challenges for programmers lie not only in the concepts of parallelism but in the implementation as well. Although there is a significant amount of work done on compilers, auto-parallelization and auto-tuners, many programmers still use low-level programming models and programming languages like CUDA, OpenCL and OpenMP. Furthermore, despite the fact that parallel acceleration is promising in most cases, performance gains are, especially in heterogeneous architectures, not guaranteed. Prediction of the performance of parallel program code is therefore necessary to decide whether the parallelization will be beneficial and which part of the (heterogeneous) platform should execute the parallel program.

Many of the existing auto-parallelization techniques are not completely automatic. Tools often need directives (e.g. hiCUDA [2], PGI Accelerator [3]) or implement algorithmic skeletons for which the programmer must provide annotations for the skeleton selection (e.g. SkelPU [4], Bones [5]). As a result, these tools place a heavy burden on programmers to produce such annotations. The work of programmers would greatly be alleviated by a method that can produce these annotations automatically. Moreover, it would be beneficial to produce a general form of annotations, only related to the program code in the form of an algorithmic classification. In case automatic classification is theoretically impossible, an easy to understand theory behind these classifications could enable programmers to manually classify program code.

The focus of this work is a new algorithm classification, ‘Algorithmic Species’ which encapsulates relevant information for parallelization (e.g. available parallelism, data re-use, memory footprint) of the algorithm in classes. 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 or compile to a parallel program. An example of a classified algorithm is given in Listing 1, the species of the algorithm is shown in lines 1 and 2.

The contributions of this work are as follows:

• We introduce a new algorithm classification, ‘Algorithmic Species’, that encapsulates relevant information about an algorithm, which in turn can be used by e.g. auto-parallelization tools.
• We introduce a new method to annotate memory transfer requirements. These requirements can be used to optimize communication between two processors for example.
• A new tool, ASET, is designed and presented that automatically extracts both the algorithmic species as well as the memory transfer requirements.
• We evaluate the applicability of the algorithmic species by manual- and automatic classification of a benchmark set.

The remainder of this paper is organized as follows. Section II motivates the need of a new algorithm classification and the requirements such a classification must fulfill. In Section III a background to related theory and earlier classifications is given. Sections IV and V present the definition of algorithmic species and how the species are automatically extracted from source code. Then, in Section VI, memory transfer requirements are added to the species to enable optimized communication. Section VII details the limitations to our approach and the related work on algorithm classifications is discussed in Section VIII. Section IX lists the results and evaluates the algorithmic species in terms of the set goals and requirements. The last section concludes this paper and sums up future work.

1The classification is named ‘Algorithmic Species’, we refer to the specific class of an algorithm as the (algorithmic) species of the algorithm.
II. MOTIVATION

The shift towards parallel computing introduced challenges in both efficient programming and compilation: managing multi-threading and efficiently using a processor’s memory hierarchy are two examples faced by programmers (looking for a manual solution) and compilers (looking for an automatic solution). The rapid evolution of, for example, vector extensions (e.g. SSE, AVX) and parallel hardware (e.g. GPUs, APUs) introduces new platforms on a regular basis, which makes programming them even more challenging as programs need to be re-optimized for every new architecture. In [6] the extent of the “Ninja gap” is quantified: the performance gap between naively written program code that is parallelism unaware (often serial) and best-optimized code on modern multi- or many-core processors. An average Ninja gap of 24x (up to 53x) is found, which increases with every new processor architecture. An algorithm classification, describing the algorithm characteristics in a target platform independent way abstracts away from these problems. The classification does not change over time 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 has similar thoughts as the production layer and efficiency layer as an existing classification [7]. In the production 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 with e.g. a skeleton implementation or programming framework. With our algorithmic species we introduce these so-called design patterns for the production layer.

We introduce three goals for an algorithm classification which we believe to be vital to aid 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 with a target architecture in mind or to fine-tune a classified algorithm. This also enables them to discuss their algorithms at an abstract level, making it easier to reason about their computational problems.

2) Performance prediction is a key factor in a development process, during design space exploration different (parallel) architectures or accelerator configurations are explored to determine the optimal configuration. Instead of creating parallel programs for all architectures, performance is predicted based on the classified algorithm and an appropriate performance model (e.g. the Boat Hull Model [8]).

3) We believe the algorithm classification facilitates the design and optimization of parallelizing-compilers, source-to-source compilers and auto-tuners. These tools are able to base their decisions on the information embedded in the algorithm classification which greatly lifts the burden of designing such tools as they do not have 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.

Fig. 1 shows how the algorithm classification serves as front-end for the three goals introduced above. The figure shows how the classified program code can automatically be parallelized to parallel program code or how it can be compiled to a parallel program, how performance prediction over a design space can select the optimal target architecture and that the classification can be used for reasoning about the algorithm. The focus of this paper is on the ‘blue box’ in Fig. 1.

In order to achieve the goals we defined for a new algorithm classification, we set six requirements that need to be met:  

1) To lift the burden for programmers to manually classify their algorithms and to achieve a fully automatic process of program-
derive the domain description and access function as used in the Polyhedral Model. For statement $S$ we describe the domain as $D_S = \{(i,j)|0 \leq i \leq 63 \land 0 \leq j \leq 127\}$ which we can write in so-called homogeneous coordinates as can be seen in (1).

The array access to array $M$ in this statement is represented by $M(\vec{f}(\vec{t}))$ for which $\vec{t}$ is the iteration vector containing the statement-enclosing loop variables, global variables (none in this example) and the constant $1$. The access function $f(\vec{t})$ for this array access is given in (2). The vector represents the array access and consists of one row per dimension of the array. Note that the access function for array $M$ as given in (2).

$$D_S : D_S \cdot \vec{t}_S = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 63 \\ 0 & 1 & 0 \\ 0 & -1 & 127 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \geq 0 \tag{1}$$

$$M : f(\vec{t}_S) = F_M \cdot \vec{t}_S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \tag{2}$$

### B. Earlier classifications

The algorithmic species are inspired by the algorithm classification as presented in [12]. In this work, a classification is presented to implement performance prediction and source-to-source compilation with the use of algorithmic skeletons as introduced in [5], [13]. The classification as presented in [12] is used as starting point in this work. Some of its shortcomings are, 1) ambiguity during classification (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 to automatically extract the classes from program code. Its vocabulary, parameters and data size ranges are adjusted in this work to accommodate the fine-grained level of detail required and the classification is formally defined to meet all the requirements as proposed in the motivation.

### IV. Algorithmic Species

In previous sections we have introduced the challenges in parallel computing and motivated the need for a new algorithm classification. In this section we present our approach ‘Algorithmic Species’. Algorithmic species is a classification which captures low-level algorithm details from individual loops or loop nests and their bodies. To get acquainted with the algorithmic species we will give some examples next in which we will introduce five access patterns that combine to an algorithmic species. 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, then form the species. This modular approach enables us to form an unlimited amount of species with the use of only five access patterns.

#### A. Example species

Before introducing the theory behind algorithmic species we will give a flavor of how an algorithmic species is constructed based on the access patterns of the arrays. We will classify the examples in listings 3 to 6 and give a short explanation of each. An overview of classified algorithms is given in Table I and illustrated through

<table>
<thead>
<tr>
<th>Example</th>
<th>Algorithmic Species</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 3</td>
<td>par(64,128) M[0:63,0:127][element] \rightarrow R[0:63,0:127][element] \land v[0:127][full] \rightarrow r[0:63][element]</td>
</tr>
<tr>
<td>Listing 4</td>
<td>par(64) M[0:63,0:127][chunk(0,0,0,127)]</td>
</tr>
<tr>
<td>Listing 5</td>
<td>par(126) a[0:127][neighborhood=1][element] \rightarrow m[1:126][element]</td>
</tr>
<tr>
<td>Listing 6</td>
<td>par(8) a[0:7][element] \land b[2:9][element] \rightarrow r[0:16][shared]</td>
</tr>
</tbody>
</table>

Listing 2. An example of an embarrassingly parallel algorithm.

Listing 3. An example of an embarrassingly parallel algorithm.

Listing 4. An example of a matrix-vector multiplication algorithm.

Listing 5. An example of a matrix-vector multiplication algorithm.

Listing 6. An example of a matrix-vector multiplication algorithm.
needs the element at location \( i \) and the two surrounding elements. The three elements in the so-called stencil operation are classified as the pattern neighborhood. Inherently to the pattern neighborhood is the re-use of the input elements as illustrated in Fig. 4. The figure shows that two elements read in the first iteration (element 1 and 2) are re-used in the second iteration. The neighborhood range spans from \(-1\) to \(+1\) with respect to the loop iteration variable \( i \). The input array \( a \) is accessed from 0 to 127 and 126 output elements can be produced in \( n \) (from 1 to 126) in parallel. With the available parallelism and the array access ranges the algorithmic species is constructed as shown in Table I.

The last access pattern, \( shared \), is introduced in the example in Listing 6. In the \( i \)-loop we see that the arrays \( a \) and \( b \) are read element-wise and that in all iterations there is a write at output-location \( r[0] \). The output-location is \( shared \) among all iterations of the \( i \)-loop. The input consists of elements 0 to 7 for array \( a \) and elements 2 to 9 for array \( b \); the output is the single element array \( r[0] \). The patterns and ranges combine to the algorithmic species of this example as shown at line 4 of Table I.

The five array access patterns, as introduced in this section are; \textbf{element}, \textbf{chunk}, \textbf{neighborhood}, \textbf{full} and \textbf{shared}. The input arrays (i.e. arrays that are read) are assigned either one of the patterns whereas the output arrays (i.e. arrays that are written) are assigned either one of the patterns, excluding \textbf{neighborhood}. We make a remark about the fact that the access ranges, as introduced in the \textbf{neighborhood} pattern, are considered to be an upper bound. If an array is accessed from element 0 to 63 it might be that some elements in this range never get accessed at all. The same holds for \textbf{neighborhoods} and \textbf{chunks}, \textbf{partial neighborhoods} or \textbf{partial chunks} are still classified as \textbf{neighborhood} or \textbf{chunk}.

Finally, we note that the \textbf{full} pattern can be seen as a special case of the \textbf{chunk} pattern. When the size of a chunk access is equal to the array access range, the complete array is accessed and classified as \textbf{full}. The inclusion of the full in the algorithmic species is therefore not required for completeness, but it does improve the compactness and readability of the class names.

Having introduced the access patterns, we will show in the next sections how to derive these patterns from the access functions.

### B. Base loops and structure loops

The access patterns are derived by analyzing the array access functions. As we will show next, the access function as used in the Polyhedral Model is not suited for our purpose and we will use a modified access function to derive the access patterns. We will illustrate this with the access functions of the array accesses to array \( M \) in the embarrassingly parallel algorithm (Listing 2) and the matrix-vector multiplication example (Listing 4).

For the statement on line 4 in the matrix-vector multiplication we give the domain description in (3) according to the Polyhedral Model. We use the vector \( \vec{i}_{S,1} \) to define the access function to array \( b \) in (4). The domain description and access function of the embarrassingly parallel algorithm where already derived in 1 and 2, respectively. Now note that array \( b \) has the same domain description and access function in both examples, but shows different access properties and is thus classified with the access patterns \textit{element} and \textit{chunk}, respectively. As the above shows, the access pattern cannot be determined by only analyzing the access function, but additional information is needed on the other array accesses in the same loop nest. With a change in the access function we can capture this extra information, which enables us to derive the access pattern.

![Fig. 4. Illustration of the first two iterations of Listings 5.](image)

**Fig. 4. Illustration of the first two iterations of Listings 5.**

**Fig. 5. Illustration of the first two iterations of Listing 6.**

The last access pattern, \( shared \), is introduced in the example in Listing 6. In the \( i \)-loop we see that the arrays \( a \) and \( b \) are read element-wise and that in all iterations there is a write at output-location \( r[0] \). The output-location is \( shared \) among all iterations of the \( i \)-loop. The input consists of elements 0 to 7 for array \( a \) and elements 2 to 9 for array \( b \); the output is the single element array \( r[0] \). The patterns and ranges combine to the algorithmic species of this example as shown at line 4 of Table I.

The five array access patterns, as introduced in this section are; \textbf{element}, \textbf{chunk}, \textbf{neighborhood}, \textbf{full} and \textbf{shared}. The input arrays (i.e. arrays that are read) are assigned either one of the patterns whereas the output arrays (i.e. arrays that are written) are assigned either one of the patterns, excluding \textbf{neighborhood}. We make a remark about the fact that the access ranges, as introduced in the \textbf{neighborhood} pattern, are considered to be an upper bound. If an array is accessed from element 0 to 63 it might be that some elements in this range never get accessed at all. The same holds for \textbf{neighborhoods} and \textbf{chunks}, \textbf{partial neighborhoods} or \textbf{partial chunks} are still classified as \textbf{neighborhood} or \textbf{chunk}.

Finally, we note that the \textbf{full} pattern can be seen as a special case of the \textbf{chunk} pattern. When the size of a chunk access is equal to the array access range, the complete array is accessed and classified as \textbf{full}. The inclusion of the full in the algorithmic species is therefore not required for completeness, but it does improve the compactness and readability of the class names.

Having introduced the access patterns, we will show in the next sections how to derive these patterns from the access functions.

**Listing 5.** An example of a neighborhood stencil operation.

<table>
<thead>
<tr>
<th>for (i=0; i&lt;8; i++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i] += a[i-1] + a[i+2];</td>
</tr>
</tbody>
</table>

**Listing 6.** An example of a reduction.

<table>
<thead>
<tr>
<th>for (i=1; i&lt;127; i++){</th>
</tr>
</thead>
<tbody>
<tr>
<td>m[i] = 0.33 +</td>
</tr>
<tr>
<td>(a[i-1]+a[i]+a[i+1]);</td>
</tr>
</tbody>
</table>

For the statement on line 4 in the matrix-vector multiplication we give the domain description in (3) according to the Polyhedral Model. We use the vector \( \vec{i}_{S,1} \) to define the access function to array \( b \) in (4). The domain description and access function of the embarrassingly parallel algorithm where already derived in 1 and 2, respectively. Now note that array \( b \) has the same domain description and access function in both examples, but shows different access properties and is thus classified with the access patterns \textit{element} and \textit{chunk}, respectively. As the above shows, the access pattern cannot be determined by only analyzing the access function, but additional information is needed on the other array accesses in the same loop nest. With a change in the access function we can capture this extra information, which enables us to derive the access pattern.

\[
D_{S,1} = \{(i,j) | 0 \leq i \leq 63 \land 0 \leq j \leq 127\} \\
D_{S,1} : D_{S,1} \cdot \vec{i}_{S,1} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 63 \\ 0 & 1 & 0 \\ 0 & -1 & 127 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \end{bmatrix} \geq \vec{0} \\
f(\vec{i}_{S,1}) = \vec{F}_M \cdot \vec{i}_{S,1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} (4)
\]

To be able to capture the structure of parallelism and data re-use into algorithmic species, we modify the representation as used by the Polyhedral Model by splitting the loop iterator vector (\( \vec{t} \)) and the iteration domain (\( D_{S} \)) in base loops (with domain \( D_{J} \)) and structure loops (\( \vec{y} \) (with domain \( D_{y} \)). Instead of analyzing the access function of a single statement, the base- and structure loops are derived for the complete loop nest. Structure loops are inner loops that access more than one element in an array, creating accesses of for example rows, columns, chunks and neighborhoods. Formally, we define a loop as a structure loop if it body holds a statement with either 1) at least one read access which is dependent on the loop iterator and all write accesses in the loop nest are not, or 2), at least one write access which is dependent on the loop iterator and all read accesses in the loop nest are not. All other loops which contain at least one statement with an array access are considered base loops. Deriving the loops is done according to Algorithm 1.

For the embarrassingly parallel example (Listing 2) we identify both loops as base loops because both the read and write depend on both the \( i \)-loop and \( j \)-loop. In the matrix-vector multiplication example (Listing 4) we identify the \( i \)-loop as base loop and the \( j \)-loop as structure loop because the read access to array \( M \) (and to \( v \)) is dependent on \( j \) whereas the write access to array \( v \) is not. Note that, in contrary to the Polyhedral Model, the vectors \( \vec{t} \) and \( \vec{y} \) are derived for the loop nest only once and are thus the same for every statement whereas in the Polyhedral Model the vector \( \vec{i}_{S} \) is derived per statement.

With the base loop iterators (\( \vec{t} \)) and structure loop iterators (\( \vec{y} \)) introduced, we introduce the matrices \( A \) and \( B \) to express the relation between the array access and the loop iterators. Furthermore we introduce the vector \( \vec{e} \) to represent constant offsets in the array access. The access for a given array \( p \) can now be described as:

\[
\vec{f}_{p} = A_p \cdot \vec{t} + B_p \cdot \vec{y} + \vec{e}_p (5)
\]

As an example we derive the base loops and structure loops and the matrices \( A \) and \( B \) for the array access to array \( M \) in the embarrassingly parallel example and the matrix-vector multiplication example. To construct the array access for array \( M \) in the embarrassingly parallel example, following the definition as in (5), we define the base loops (\( \vec{e}_1 \)) and structure loops (\( \vec{e}_1 \)) as in (6), the matrices as in (7) and find the array access as in (8).
Algorithm 1: Deriving base- and structure loops

Input: Polyhedral description of all statements \( S \) in the loop nest
Output: Base loop iteration vector \( \bar{x} \) and structure loop iteration vector \( \bar{y} \)

Update all statements in \( S \): strip iteration vectors \( \bar{t} \) to loop iteration variables only and update all matrices \( F \), accordingly.

Gather writes and reads: \( \bar{W} = \bar{\bar{g}} \) and \( \bar{R} = \bar{\bar{g}} \)

\[ \bar{x} = \emptyset, \bar{y} = \emptyset \]

repeat

- \( \bar{t} \leftarrow \) modified iteration vector per statement
- \( F \leftarrow \) sum columns of \( F_w \), describing a write access, to row vector \( \bar{W} \)
- \( R \leftarrow \) zeros(dim(\( F_t \)))

for all read accesses in current statement do

- \( F \leftarrow \) sum columns of \( F_r \), describing a read access, to row vector \( \bar{R} \)

end

until all statements are processed;

if \( (\bar{W} \neq \emptyset \) and \( \bar{R} \neq \emptyset \) then

for \( i = \text{length}(\bar{t}) - 1 \) to \( 2 \) do

- if \( (\bar{R} \neq \emptyset \) and \( \bar{W} = 0 \) ) or \( (\bar{R} = 0 \) and \( \bar{W} \neq 0 \) ) then

  \( \hat{y} \leftarrow \bar{y} \cup \bar{t}_i \)

else

  break

end

end

\[ \bar{x} \leftarrow \bar{x} \cap \bar{y} \]

Result: \( \bar{x}, \bar{y} \)

\[ \bar{x}_1 = \begin{pmatrix} i \\ j \end{pmatrix}, \quad \bar{y}_1 = [0] \quad (6) \]

\[ A_{M_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B_{M_1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \bar{c}_{M_1} = [0] \quad (7) \]

\[ \bar{x}_2 = \begin{pmatrix} i \\ j \end{pmatrix}, \quad \bar{y}_2 = (j) \quad (9) \]

\[ A_{M_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad B_{M_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \bar{c}_{M_2} = [0] \quad (10) \]

\[ \bar{x}_2 = \begin{pmatrix} i \\ j \end{pmatrix}, \quad \bar{y}_2 = (j) \quad (11) \]

C. Deriving array access patterns

As is shown in the previous section, the access function of the Polyhedral Model cannot uniquely distinguish the access patterns as introduced in the first section of this chapter. Therefore, we introduced a new access description in (5) with the matrices \( A, B \) and the vector \( \bar{c} \).

The matrices \( A \) and \( B \) relate the array access to the loop iterators of the loop nest and can now be used to derive the access pattern as Algorithm 2 shows.

In case both \( A \) and \( B \) are zero, the access is independent of all the loops meaning an access to a constant address that is \textit{shared} among all iterations (see the two-vector reduction example in Listing 6).

With \( A \) non-zero and \( B \) zero, every iteration reads or writes \textit{element-wise}. With \( A \) zero and \( B \) non-zero, the \textit{full} array is accessed every time the structure loop is completely executed. Finally, when both \( A \) and \( B \) are non-zero the access is classified as \textit{neighborhood} if (12) holds, otherwise as \textit{chunk}.

Algorithm 2: Deriving the array access patterns

Input: Access descriptions of all arrays
Output: The access patterns of all arrays \( P = \emptyset \)

repeat

- \( S_P = \emptyset \)
- \( (A_p, B_p, \bar{c}_p) \leftarrow \) access description of array \( p \)
- switch \( (A_p, B_p) \) do

  - case \( A_p = 0 \) \& \( B_p = 0 \)
    - \( S_p \leftarrow \text{"shared"} \)
  - case \( A_p \neq 0 \) \& \( B_p = 0 \)
    - \( S_p \leftarrow \text{"element"} \)
  - case \( A_p = 0 \) \& \( B_p \neq 0 \)
    - \( S_p \leftarrow \text{"full"} \)
  - case \( A_p \neq 0 \) \& \( B_p \neq 0 \)
    - if (equation 12 holds for array \( p \)) then
      - \( S_p \leftarrow \text{"neighborhood"} \)
    - else
      - \( S_p \leftarrow \text{"chunk"} \)

endsw

endsw

\( P \leftarrow P \cup S_p \)

until all patterns are derived;

Result: \( P \)

As Algorithm 2 shows, when both matrices \( A \) and \( B \) are not zero we classify the access pattern either as \textit{neighborhood} or as \textit{chunk} depending on whether or not there is re-use between the different accesses to the array. Re-use occurs when two different iterations access the same location in an array. The condition for re-use is given in (12).

\[ \exists (\bar{x}_1, \bar{y}_1), (\bar{x}_2, \bar{y}_2) \text{ such that} \]

\[ A_p \cdot \bar{x}_1 + B_p \cdot \bar{y}_1 = A_p \cdot \bar{x}_2 + B_p \cdot \bar{y}_2, \]

with \( (\bar{x}_1 \neq \bar{x}_2 \lor \bar{y}_1 \neq \bar{y}_2) \) and \( (\bar{x}_1, \bar{x}_2) \in \mathcal{D}_x \) and \( (\bar{y}_1, \bar{y}_2) \in \mathcal{D}_y \)

D. Array access ranges

In our motivation we stated three goals for our classification and six requirements to fulfill. One of the requirements of the classification is that it must be fine-grained enough to capture essential information about the algorithm in terms of 1) available parallelism, 2) data reuse inside the algorithm (captured in the array access patterns) and 3) data sizes of input and output. These data sizes as well as the sizes of the neighborhood and chunk-access are derived next to complete the requirement.

As guidance to explain the theory behind the ranges of the array access, the matrix-vector multiplication example in Listing 4 is used. For the loop nest we already derived the base loop \( (\bar{x} = (i)) \) and the structure loop \( (\bar{y} = (j)) \). The domain of these loops is represented

\[ \bar{x}_1 = \begin{pmatrix} i \\ j \end{pmatrix}, \quad \bar{y}_1 = [0] \]

\[ A_{M_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B_{M_1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \bar{c}_{M_1} = [0] \]

\[ \bar{x}_2 = \begin{pmatrix} i \\ j \end{pmatrix}, \quad \bar{y}_2 = (j) \]

\[ A_{M_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad B_{M_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \bar{c}_{M_2} = [0] \]

\[ \bar{x}_2 = \begin{pmatrix} i \\ j \end{pmatrix}, \quad \bar{y}_2 = (j) \]

\[ \mathcal{D}_x = \{ (i) \} \]

\[ \mathcal{D}_y = \{ (j) \} \]
as follows:
\[
D_x = \{i | 0 \leq i \leq 63 \}
\]
\[
D_y = \{j | 0 \leq j \leq 127 \}
\]
\[
\bar{h} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad \bar{H} = \begin{bmatrix} -1 & 0 \end{bmatrix}
\]

The actual loop bounds are stored in the last column of the matrices \(D_x\) and \(D_y\). Note that, due to the construction of the domain and representation in homogeneous coordinates, the lower bound is stored negated. To represent the actual lower- and upper bound values as row vector we introduce a helper vector to select the appropriate row (\(\bar{h}\)) and a helper matrix to negate the lower bound (\(\bar{H}\)). They are set as:
\[
\bar{h} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad \bar{H} = \begin{bmatrix} -1 & 0 \end{bmatrix}
\]

A row vector, containing the actual lower- and upper bound values, can then be calculated as: \(\bar{h} \cdot D_x^y \cdot \bar{H}\).

To calculate the range of the chunk access to array \(A\) we need the stride of the array access with respect to the structure loop \((B_{Mx}\) in (10)), the constant offset of the access \((\vec{c}_{Mx}\) in (10)) and the information stored in \(D_y\). The range of the chunk access to array \(H\) is then calculated as:
\[
B_{Mx} \cdot \bar{h} \cdot D_y \cdot \bar{H} + (\vec{c}_{Mx} \cdot \vec{e}_{Mx}) = \begin{bmatrix} 0 & 0 \\ 0 & 127 \end{bmatrix}
\]

This must be interpreted as: no chunk access in the first dimension of the array (row 1) and a chunk access from 0 to 127 in the second dimension (row 2).

To complete the algorithmic species the access range of the complete array is calculated. The range of the chunk access (accessed in the structure loop) and the range accessed in the base loop together form the range of the complete array access. The range accessed in the base loop is added to the chunk access range and the complete access range is computed as follows:
\[
\begin{bmatrix} 0 & 0 \\ 0 & 127 \end{bmatrix} + B_{Mx} \cdot \bar{h} \cdot D_y \cdot \bar{H} = \begin{bmatrix} 0 & 63 \\ 0 & 127 \end{bmatrix}
\]

In words, the result is the following: the array is accessed from 0 to 63 in the first dimension (row 1) and from 0 to 127 in the second dimension (row 2).

With the ranges of the array and chunk access found, all information on the array access pattern is derived and the species can be composed. As stated before, an algorithmic species is combined from the access patterns corresponding to arrays accessed inside the algorithm’s loop nest. For array \(A\) in the matrix-vector multiplication example we construct the complete array access pattern using its name: \(‘M’\), its array range: \([0;63,0;127]\), its access pattern: \(‘chunk’\) and the chunk range: \(0;0;127\) as: \([M;0;63,0;127]|chunk[0;0,0;127]\). For the example we find a second input array, \(v\) and the output array \(x\).

We concatenate multiple arrays at either the input side or output side with a hat \(\hat{\cdot}\) and relate the input to the output using an arrow \(\rightarrow\). The resulting species is shown in Table I.

### E. Available parallelism

To meet the requirement that the algorithmic species must capture the available parallelism, we explicitly add the available parallelism because it is not always uniquely coupled with the accessed data ranges and access patterns. A key feature of the algorithmic species is the amount of available parallelism in the classified algorithm. This parallelism is often implicitly encapsulated in the algorithmic species but it is not always uniquely determinable by inspecting the access ranges and access patterns. To illustrate that the amount of parallelism and the algorithmic species can mismatch we use the two neighborhood examples in Listing 7 (lines 1-4 and 6-12). We see that array \(\tilde{B}\) is accessed from 0 to 63 (64 elements) in the first two loops, which matches with the amount of parallelism in the first loop being 62, but matches the amount of parallelism in the second loop (64). These neighborhood access patterns show that the accessed range is not directly related to the amount of parallelism. The access to array \(\tilde{A}\) does match with the amount of parallelism in both loops as every loop iteration writes an element to \(\tilde{A}\). To clearly separate the parallelism from the accessed data ranges, the parallelism is annotated explicitly as can be seen in the example.

The third loop (lines 14-17) shows a ‘neighborhood to shared’ example. This example shows that neither the input, nor the output data range can be related to the amount of parallelism and it is therefore explicitly annotated. The available parallelism is denoted in terms of the base loop iterations because those loops are allowed to be executed independently.

The available parallelism equals the amount of base loop iterations because when all iterations of a loop are independent, the loop is classified as base loop. With dependent iterations, a loop is classified as structure loop. The parallelism is annotated in the algorithmic species with the keyword \(\text{par}\) and the amount of base loop iterations separated by commas between round brackets.

### F. Dependencies

The algorithmic species theory is designed to classify program code for parallel computing. An essential element in deriving these algorithmic species is determining if the algorithm exhibits loops that can in fact be executed in parallel. For a loop to be executed in parallel we require that no array depends on itself. An array can depend on itself if all arrays hold that \(A[i+1] = A[i]\) or more statements in the same loop or in multiple loops (layered dependency). As is illustrated in [14], when no array depends on itself, the loop nest can be parallelized or vectorized. The theory in [14] can be applied to the algorithmic species in terms of the matrices \(A\) and \(B\) and the vector \(c\).

As a result, there are no dependencies when for all arrays holds that there is no single read (\(\vec{I}_{p,w}\)) and write (\(\vec{I}_{p,w}\)) to the same location in different iterations and an array is not written multiple times at the same location. The loop iterations for which the dependencies are checked must be inside the bounds of the base loops \((D_x)\) and structure loops \((D_y)\). There is an absence of dependencies when the following holds:
\[
\vec{I}_{p,w}(\vec{x}_1, \vec{y}_1) = \vec{I}_{p,w}(\vec{x}_2, \vec{y}_2) \quad \text{and} \quad \vec{I}_{p,w}(\vec{x}_1, \vec{y}_1) = \vec{I}_{p,w}(\vec{x}_2, \vec{y}_2),
\]

where \(\vec{x}_1 \neq \vec{x}_2 \lor \vec{y}_1 \neq \vec{y}_2\) and \((\vec{x}_1, \vec{x}_2) \in D_x\) and \((\vec{y}_1, \vec{y}_2) \in D_y\).
V. AUTOMATIC EXTRACTION OF ALGORITHMIC SPECIES

In order to automatically classify algorithms in program code we developed ASET, an algorithmic species extraction tool based on the algorithmic species theory. Because the theory behind algorithmic species is built upon the Polyhedral Model, we use a polyhedral representation of the program code as input to ASET. For this we use PET (Polyhedral Extraction Tool) [11], chosen for its support of a large amount of C99 language constructs and its preservation of source code line numbering, crucial for code annotation.

A short overview of the main functionality of ASET is shown in Fig. 6. Before we elaborate more on ASET, we will first give a brief overview of the steps (a through f) as identified in the figure:
(a) Invoke PET to extract a per-statement polyhedral representation of the program code. The representation specifies the statement domain, array access functions, statement scheduling information and the statement line number.
(b) With the output of PET an AST of statements is built and artificial loops are added where necessary.
(c) Identify the per loop nest base loops (⃗x) and structure loops (⃗y) and their respective domains D_⃗x and D_⃗y using Algorithm 1. With this information, the per statement, per array matrices A, B and ⃗c are derived.
(d) The dependency check of (18) is applied to check if the iterations of the loop nest can be executed independently. When dependencies are found, ASET stops and continues with the next loop.
(e) Based on the information obtained in step (c), the array access patterns are classified using Algorithm 2 and the ranges are calculated.
(f) When 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.

A SET

with species

PET

statement tree

apply
dependency

check

extract

access patterns

combine

to species

program code

Fig. 6. An overview of the functions performed by ASET during the extraction of algorithmic species.

A. Statement tree and transformations

The scheduling information from PET is used to build an abstract syntax tree (AST) of loops and the statements inside the loop bodies as is illustrated by the AST in Fig. 7, representing the loop nest of Listing 8. ASET classifies based on this tree and starts with the outer loop of the loop nest as root node (node i). When the dependency check reports dependencies for this loop or when ASET is configured to classify all loops, the loops inside the root node are classified recursively.

In order to apply algorithms 1 and 2 we require code to be written in a pre-defined manner. After constructing the tree of loops and statements, code transformations are applied to meet this requirement. We take the neighborhood operation in Listing 5 as example. In this example, the neighborhood access to a is explicit rather than expressed by a loop. As a result, Algorithm 1 will yield incorrect results as only the i-loop is classified as base loop and no structure loop is detected which in turn yields incorrect results in classifying access patterns in Algorithm 2.

In order to classify code structured as in the example, ASET identifies statements with multiple reads and derives address offsets with respect to the loop variables (in the example the offsets are -1, 0 and +1). When offsets are spanning a range of two or more, a temporary loop is introduced. For example, the neighborhood example of Listing 5 is represented internally in ASET as shown in Listing 8.

B. Dependency check

Accesses to the same array inside a loop nest need to be checked for dependencies according to equation 18 (step d). Because an exhaustive search is not scalable, an initial dependency check, GCD-test [15] and the Banerjee-test [15] are applied to compare addresses of reads and writes. The initial check proves an absence of dependencies when the matrices A and B and the vector ⃗c, of the two array accesses that are compared, are equal. The GCD-test searches for integer solutions, but does not consider loop bounds. As a result, if dependencies exist outside the loop bounds, the GCD-test might yield a false positive. On the contrary, the Banerjee-test does consider loops bounds (if they can be statically determined), but searches for non-integer solutions as well. Again, this might result in false positives. Within ASET, both tests are combined: the GCD-test searches for integer solutions and the Banerjee-test searches for solutions within the loop bounds. This combination is sufficient for single-dimensional, affine access expressions: an integer solution outside the loop bounds and a non-integer solution within the loop bounds can only co-exist if multiple solutions exist, which in turn cannot occur for affine address expressions. For example, the statement $A[2+i+i+1+2] = A[5+1]$ yields an equation with two solutions: $2 \cdot i^2 + 2 = 5 \cdot i$, however, the statement on the left hand side is non-affine. For multi-dimensional and affine access expressions both tests are used to check for dependencies subscript-by-subscript. This approach however, is conservative and might result in false positives [16]. Exact dependency tests, which propagate dependencies from one dimension to the next, are found in e.g. the Delta-test [16] or Omega-test [17] but are currently not implemented in ASET. For our purposes, a combination of the GCD-test and the Banerjee-test will find all real dependencies although it might find a few false positives as well.

The extraction tool ASET classifies all loops in program code for which the dependency check holds. Because ASET iterates through all loops in a loop nest, nested classifications can occur. To limit the number of classifications found, nested classifications can be prevented by configuring ASET to not further consider already classified code. To further limit the number of classifications, the polyhedral extraction tool PET allows the user to identify source code targeted at parallelization, thus omitting boundary or debug code such as the initialization of variables or the printing of results.

C. Annotating species

With all possible loops classified, the species are annotated in the source file. In extracting the polyhedral information from the program code (step a), PET delivers the program code line number

```c
1 for (i=1; i<127; i++){
2    temp = 0;
3 for (j=-1; j<1; j++){
4    temp += a[i+j];
5 }
6 m[i] = 0.33*temp;
7 }

Listing 8. The internal representation of the explicit neighborhood from Listing 5 to enable correct classification of loops and access patterns.

Fig. 7. AST representation as used in ASET for Listing 8.
```
per statement. These line numbers are used to find the boundaries of the enclosing loops which are annotated with pragmas.

An example of annotated program code is given in Listing 9. Due to a dependency in the statement at line 6 the i-loop cannot be executed in parallel and is therefore not classified. Both the j-loop and k-loop are dependency free and are classified as is illustrated in the listing. The pragma containing the algorithmic species is placed right above the loop and consists of:

- the header: #pragma species kernel,
- a name for the algorithmic species (optional),
- the available parallelism: par(...),
- the array access patterns and ranges.

The annotation #pragma species endkernel is placed directly after the loop denoting the end of the algorithmic species.

```c
for (i=1; i<64; i++) {
  #species kernel par(128) A[i−1:i−1,0:127]|element ∧
  C[0:127,0:255]|chunk(0:0,0:255) →
  B[i,0:127]|element ∧ A[i,0:127]|element
}
```

Listing 9. An example to show a dependency in the i-loop and nested species in the j-loop and k-loop.

In this section we presented our tool ASET which automatically derives the algorithmic species, classifying an algorithm in program code. When an algorithm consists of several parts that are individually classified as an algorithmic species, communication between those species might influence certain properties (e.g. performance). The next section gives an example of how this communication can be optimized by linking algorithmic species together.

VI. INTER-SPECIES MEMORY REQUIREMENTS

Next to the algorithm performance, inter-processor communication can play a major role in overall performance. For example, in an heterogeneous architecture where algorithmic kernels are offloaded to accelerators, data must be copied back and forth between the host and the accelerator. When implemented in a naive way (upload all the input, download all the output) there might be a lot of redundant data copies. This 'communication overhead' is subject to optimization.

The memory requirements of an algorithmic kernel are already explicitly annotated in the corresponding species by the input- and output arrays and their sizes. These sizes reflect only the memory requirements per kernel and do not provide information on whether or not the data is used (before or after) by other kernels. The problem to solve is that kernels themselves are unaware of the memory requirements of other kernels. In this section we introduce the addition of memory transfer requirements to the species to indicate at what instance their inputs need to be uploaded and at what instance their outputs need to be downloaded. This enables us to 'link' species together through their memory requirements and to optimize memory communication as will be shown in this section.

A. Naive memory transfers

Algorithmic species are used to classify program code for parallel computing, which can be performed on many different platforms.

We use a heterogeneous computing platform with a host (e.g. CPU) and an accelerator (e.g. GPU) in this section to illustrate how communication can be optimized.

As an example we use an algorithm which performs three matrix multiplications in a row ($A \cdot B = E$, $C \cdot D = F$ and $E \cdot F = G$). When implemented on the example platform, the host offloads the computations to the accelerator. With unoptimized, naive data transfers all input data is copied to the accelerator just before the execution starts and copied back to the host immediately after the execution has finished. When the data transfers are implemented in a naive way, redundant copies exist as is shown in Fig. 8a. The data of matrices E and F is uploaded just before executing $E \cdot F = G$ but was already available on the accelerator as result of the first and second computation.

Fig. 8. Computation (C) and memory transfers (T) involved in 3 consecutive matrix multiplications executed on an accelerator. Naive communication is shown in (a), optimized communication is shown in (b).

In [18], an optimization strategy is proposed as to launch transfers from host to accelerator as early as possible and launch those from accelerator back to host as late as possible. Additionally, launching transfers inside loops must be avoided wherever possible. Following this strategy, optimizing the memory transfers and executions of Fig. 8a is done by two means. First, only the input data is copied to the accelerator (A, B, C and D) but not the intermediate results, and second, the data transfers can be overlapped with the computational kernels (provided that the platform supports this).

The scheduling of the memory transfers and computation is not part of this work, Fig. 8b merely shows a possible result. Algorithmic species are platform independent and we do therefore not further discuss scheduling. What we do provide, coupled with the algorithmic species, is information on the upload and download requirements in so-called memory regions. These requirements provide all the information needed to efficiently implement the memory transfers, as seen in Fig. 8b for example.

B. Memory regions

To link species together through memory requirements, we give instances (points in execution) names that pair-wise form memory regions. The memory regions are numbered and for region ‘N’ the start is ‘eN’ and the end is ‘iN’, the complete region from start to end is referred to as eN:iN. With every instance having its own name, regions can be formed in which data transfers must be completed. Memory regions can be explicit (annotated in the program code) or implicit (species automatically create instances at which their execution starts and ends).

Upload regions are formed by an instance from which the upload is allowed to start and an instance before which the upload must be finished. The same holds for the downloads, the region denotes the instance from which the download is allowed to start and the instance before which the download must be finished.

We will further elaborate on memory regions by discussing the example in Listing 10, which is structured as the three matrix multiplications example as used in the previous section. In this...
example, we show one explicit memory region (region 0) which marks the bounds of the complete program code. The contents inside this memory region are executed starting at line 1 and ending at line 16.

Next to the explicit memory region in the example, there are three implicit memory regions introduced by the kernels k1, k2 and k3, all nested inside region 0. Kernel k1 is executed at instance e1 and finishes at instance f1. The upload requirement of the arrays A and B is therefore e0|e1 for both arrays indicating that the upload can already start at e0 and must be finished at e1. The result of kernel k1 is not modified anymore after instance f1 and can be downloaded in the memory region f1:f0. The memory transfer requirements of the three kernels are illustrated with instances and ranges in Fig. 9.

The annotations used to provide these requirements are shown in line 4 and are formatted as: <u|d>-<mem-region>-<arrays> where u and d denote upload and download respectively, mem-region denotes the range in which the transfer needs to be completed, and arrays denotes a comma-separated list containing the arrays and their ranges to transfer. For kernel k2 we can derive similar requirements and find for the upload requirements u-e0|e1(C,D) and for the download requirements d-f2:f0(F). For the last kernel k3, we do not need any input as the results of the previous kernels still reside in the accelerator memory. The download must be completed in the final region: d-f3:f0(G).

A second example is given in Listing 11 in which a stencil operation (kernel k1) is applied 8 times on array A. As is shown in lines 6 and 7, the data transfers are moved outside the loop and consequently, the upload of arrays A and B must take place at instance e0 and the result is downloaded at instance f0. If the data transfers in this example would be implemented in a naive way, it would result in 16 uploads (8x A for k1, 8x B for k2) and 16 downloads (8x B for k1, 8x A for k2). In this example the data transfers can be reduced by 10x using information provided by the memory region annotations, a factor dependent on the number of repetitions of the stencil operation.

C. Automatic extraction of memory regions

The memory regions are extracted in a post-processing stage of ASET. When all species are extracted by ASET (stage (f) of Fig. 6) the memory regions are created. The AST of loops and statements, as produced in stage (b), defines the order of execution of the species and enables to relate the memory transfers of species to each other.

To determine the upload- and download regions, two algorithms are used to find the largest region in which the transfers must be completed. The algorithms determine these regions based on the program structure. For example, for programs where species are nested inside loops, the memory transfers are moved outside the loops. Currently, two program structures are supported by ASET:

1) Programs for which all loop nests are classified as an algorithmic species. No algorithmic species is nested inside an unclassified loop and no statements are executed in between two species.

The structure of the example in Listing 10 is of type 1.

2) Programs for which all loop nests are classified as an algorithmic species. At least one algorithmic species is nested inside an unclassified loop and no statements are executed in between two species. The structure of the example in Listing 11 is of type 2.

In case the program structure is either of type 1 or of type 2, the upload ranges of arrays are determined as is shown in Algorithm 3. The default upload instance is e0, which is the start of the classified program code. All uploads can start at e0 because no memory is accessed on the host in region e0:f0. The instance at which the upload must be finished is either the species execution instance (when the species is not nested in a loop) or the end of the previous species which is not nested inside a loop. To determine the download range, Algorithm 4 is used. The default download instance is f0, the end of the classified program code. All downloads must end at f0 because, again, no memory is accessed on the host in region e0:f0. The downloads can start at an instance that is either the end of execution of the species (when the species is not nested in a loop) or the start of the next species which is not nested inside a loop.

In this section we have introduced memory regions to facilitate the efficient scheduling of data transfers in relation to algorithmic species. For two program structures we have shown policies for determining the upload and download ranges. We believe that the concept of memory regions is easy to understand and enables programmers to manually add memory regions and memory transfer ranges when a program structure is encountered that is not yet supported by ASET.
VIII. Related Work

We identify two types of related work. 1) algorithmic classifications, and, 2) automatic extraction tools. Algorithm classifications have been designed in the past for many different purposes, resulting in a large body of work with many different properties. We group existing work, based on their abstraction level, in four categories. These categories range from high abstraction level classifications to low-level classifications that are close to the program code. In each category we identify the most prominent work and briefly discuss main differences in comparison to algorithmic species.

1) At the highest level of abstraction we find the 13 dwarfs from Berkeley [7]. These dwarfs are introduced as a method to capture patterns of computation and communication, and group algorithms with the same properties (e.g. linear algebra operation) in a dwarf. Related to dwarfs is the work on pattern languages for parallelism [19]–[21] which are intended to guide programmers by providing descriptions of frequently occurring problems. Typically, patterns at multiple levels of abstraction are provided, but often start at a high abstraction level. The Galois system [22] provides another classification at a high level of abstraction, entirely focused on the classification of irregular algorithms. Classifications in this group have a high level of abstraction, making the level of detail much coarser-grained than that of algorithmic species. Furthermore, they lack the ability to classify program code automatically.

2) The work on algorithmic skeletons [23] has lead to a large number of algorithm classifications. In [24] a survey of common skeletons is presented, listing skeletons such as farm, pipe, fork, join, divide and conquer, client-server and zip. More recent work on skeletons [4], [5], [12], [13] introduces lower-level classifications such as pixel-to-pixel, neighborhood-to-pixel, pixel-to-global and bucket processing [13]. Although the level of detail is finer-grained than with the dwarfs and pattern languages, it is still not as detailed as algorithmic species. Next to that, the classification tied to skeletons is not complete as for new algorithms, new skeletons (and thus classes) need to be created.

3) Closer to program code is the work introducing ‘Æcute’, a decoupled access/execute specification [25]. Æcute represents program code in classes using a mathematical formulation, as a result it is formally defined, complete, and fine grained but lacks, intuitive understanding and automatic extraction.

4) The last group of classifications do not use algorithm classes but rather give a mathematical representation of the program code. Such classifications typically work on loop nests and represent aspects such as iteration spaces, loop dependencies and data locality and are often used for loop optimizations such as tiling, skewing or fusion. Examples of such classifications are the Polyhedral Model [9], [26] and the SUIF loop transformation formulation [27]. The classifications in this group share most properties with algorithmic species (complete, formally defined, fine-grained, automatic extraction, and language- and platform independence) but lack the intuitive understanding which is mainly caused by fact that no algorithm classes are used in these ‘classifications’.

As for automatic extraction tools, we can find no direct related work. ASET is unique in the fact that it automatically classifies program code that can be used for many purposes. The most related work on automatic extraction is found in PIR [28]. PIR is an idiom (access pattern) recognizer which will find promising loops for optimization in a huge search space, effectively reducing this space. The idiom must be manually provided in compiler intermediate representation and the optimizations are left for the programmer, resulting in manual effort, although less that searching through the complete search space. Furthermore, somehow related is the work
on Par4All [18] and Pluto [29]. Where such tools are designed for a single purpose (e.g. auto-parallelization), ASET’s uniqueness lies in the fact that the automatically classified program code can be used for many purposes, including auto-parallelization. The work on Par4All also provides memory optimizations with a similar scheduling to what we presented in Section VI. Par4All provides a library to optimize communication during run-time of a program. Compared to Par4All, ASET’s strength is the ability to extract the memory transfer requirements automatically from program code, enabling programmers or tools to implement their own optimizations for example. The power to both classify the program code and to extract the memory transfer requirements are unique to ASET, making it a well-suited tool to serve as front-end to several uses related to parallel programming.

IX. RESULTS AND EVALUATION

To evaluate the applicability of the algorithmic species theory and the automatic extraction of algorithmic species and memory transfer requirements we use the PolyBench/C benchmark set and two real-life applications. The PolyBench set consists of 30 algorithms in six domains of scientific computing. In the benchmark set we find a total of 115 species of which 55 species are nested inside other species, and in the real-life applications we find 19 species of which 7 are nested inside other species. The classification details of the complete PolyBench set and applications are presented in Table II. In total 78% of the lines of code are captured in algorithmic species. The lines of code that are not classified are part of loops that have dependencies, or cannot be classified because they are not part of any loop. When we use ASET we are able to automatically identify 133 species, 1 species is not found due to a false positive dependency in the dependency check.

In the PolyBench set we find 17 algorithms that have a program structure supported by ASET to derive the memory upload- and download ranges. Of these 17 algorithms, 14 are completely classified and 3 have species nested inside a loop. The remaining PolyBench algorithms and the two real-life applications have program structures that require new algorithms to determine the upload- and download ranges.

As stated in the motivation, we see a new algorithm classification as vital aid in aspects related to parallel programming. Such a classification can help programmers to discuss their computational problems, enable performance prediction, and function as general front-end to e.g. auto-tuners or parallelizing compilers. In Section II we presented six requirements to meet the goals for the new classification. The algorithmic species classification is evaluated in terms of these requirements:

1) With ASET we showed that it is possible to extract algorithmic classes automatically from program code. This greatly lifts the burden for programmers as there is no manual effort required.
2) An algorithmic species is composed of five intuitive and easy to understand access patterns. These easy to understand access patterns allow programmers to reason about their algorithms or to fine-tune extracted classes.
3) The species theory is formally defined with the Polyhedral Model as underlying basis. The classification of loops and access patterns is well-defined which leads to an unambiguous classification of program code in species.
4) The classification is complete within certain constraints. Any loop nest with static control, affine array accesses and free of inter-loop dependencies can be classified under the algorithmic species theory. The use of access patterns to build the algorithmic communication during run-time of a program, this in contrast to pattern languages or algorithmic skeletons which require new classes as new types of algorithms are encountered.

3Available at: http://www.cse.ohio-state.edu/~pouchet/software/polybench/

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Species</th>
<th>SLoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear algebra kernels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2mm</td>
<td>2 (6)</td>
<td>16/16 (100%)</td>
</tr>
<tr>
<td>3mm</td>
<td>3 (9)</td>
<td>24/24 (100%)</td>
</tr>
<tr>
<td>atax</td>
<td>2 (6)</td>
<td>12/12 (100%)</td>
</tr>
<tr>
<td>bicg</td>
<td>2 (6)</td>
<td>12/12 (100%)</td>
</tr>
<tr>
<td>cholesky</td>
<td>2 (6)</td>
<td>10/14 (71%)</td>
</tr>
<tr>
<td>doigen</td>
<td>2 (6)</td>
<td>17/17 (100%)</td>
</tr>
<tr>
<td>gemm</td>
<td>1 (3)</td>
<td>8/8 (100%)</td>
</tr>
<tr>
<td>gemver</td>
<td>4 (7)</td>
<td>18/18 (100%)</td>
</tr>
<tr>
<td>gesummv</td>
<td>1 (3)</td>
<td>9/9 (100%)</td>
</tr>
<tr>
<td>mvt</td>
<td>2 (4)</td>
<td>10/10 (100%)</td>
</tr>
<tr>
<td>symm</td>
<td>1 (3)</td>
<td>11/13 (85%)</td>
</tr>
<tr>
<td>syrk</td>
<td>1 (3)</td>
<td>9/9 (100%)</td>
</tr>
<tr>
<td>syrzk</td>
<td>1 (3)</td>
<td>8/8 (100%)</td>
</tr>
<tr>
<td>trisolv</td>
<td>1 (3)</td>
<td>4/7 (57%)</td>
</tr>
<tr>
<td>trmm</td>
<td>1 (3)</td>
<td>3/7 (43%)</td>
</tr>
<tr>
<td>Linear algebra solvers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grasmchmidt</td>
<td>3 (5)</td>
<td>16/21 (76%)</td>
</tr>
<tr>
<td>durbin</td>
<td>2 (3)</td>
<td>6/18 (33%)</td>
</tr>
<tr>
<td>dynpro</td>
<td>1 (2)</td>
<td>5/17 (29%)</td>
</tr>
<tr>
<td>lu</td>
<td>2 (3)</td>
<td>8/10 (80%)</td>
</tr>
<tr>
<td>ladmmp</td>
<td>4 (7)</td>
<td>12/33 (36%)</td>
</tr>
<tr>
<td>Data-mining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>correlation</td>
<td>4 (8)</td>
<td>30/34 (88%)</td>
</tr>
<tr>
<td>covariance</td>
<td>3 (5)</td>
<td>15/21 (71%)</td>
</tr>
<tr>
<td>Graph algorithms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>floyd-warshall</td>
<td>0</td>
<td>0/0 (0%)</td>
</tr>
<tr>
<td>Image processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg_detect</td>
<td>3 (5)</td>
<td>14/27 (52%)</td>
</tr>
<tr>
<td>Stencil operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>adi</td>
<td>3 (5)</td>
<td>9/30 (30%)</td>
</tr>
<tr>
<td>ftdl-2d</td>
<td>4 (7)</td>
<td>20/20 (100%)</td>
</tr>
<tr>
<td>ftdl-2d-apml</td>
<td>1 (4)</td>
<td>28/28 (100%)</td>
</tr>
<tr>
<td>jacobi-1d-imper</td>
<td>2</td>
<td>6/8 (75%)</td>
</tr>
<tr>
<td>jacobi-2d-imper</td>
<td>2 (4)</td>
<td>12/12 (100%)</td>
</tr>
<tr>
<td>seidel-2d</td>
<td>1 (3)</td>
<td>0/7 (0%)</td>
</tr>
<tr>
<td>Applications</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast Focus on Structures</td>
<td>8 (10)</td>
<td>70/70 (100%)</td>
</tr>
<tr>
<td>K-Means Clustering</td>
<td>4 (9)</td>
<td>70/80 (88%)</td>
</tr>
</tbody>
</table>

5) Species are fine-grained in the sense that they capture information on the amount and structure of parallelism, communication requirements, atomicity, data-reuse, data-locality and data-sizes.
6) The algorithmic species theory is able to capture all program code independent of language or target platform with the presented limitations in mind. Although in all examples C-code is shown, the theory is equally applicable to other languages, as long as the program code can be represented in the Polyhedral Model.

Since the classification meets all of our requirements, we are convinced that algorithmic species are suitable to meet our goals. We have shown the applicability of algorithmic species by identifying a total of 134 species in a benchmark set and two real-life applications. Algorithmic species can therefore serve as a base for current and future work related to parallel programming as we illustrate in Fig. 1.
We briefly reflect on the goals:

1) A manual use of the classification can be found for programmers performing various tasks when developing parallel programs, e.g. designing, optimizing, profiling or debugging program code. For example, programmers can implement known solutions in terms of algorithmic species for their computational problems or converse about their implementations with other programmers.

2) For automated uses that benefit from the work on algorithmic species we identify the ‘Boat Hull Model’ [8] for performance prediction.

3) A second automated use is found in ‘Bones’ [5], a parallelizing compiler. Still, we believe more tools can build upon algorithmic species, making it a common front-end to capture essential information of the program code in the process of creating parallel code.

X. CONCLUSION AND FUTURE WORK

The recent shift towards heterogenous- and parallel computing platforms introduced challenges for parallel programmers and compiler designers, e.g. efficiently exploiting memory hierarchy and dealing with concurrancy or synchronization. In this work we presented ‘Algorithmic Species’, an algorithm classification which captures low-level algorithmic details and represents them with the use of five easy to understand array access patterns. Algorithmic species can be used by programmers to converse on algorithms or serve as a front-end to performance prediction models or parallelizing compilers for example. Furthermore, we introduced memory regions, a method to link species together through memory transfer requirements, in order to optimize communication in heterogeneous computing platforms.

To conclude, we have contributed with 1) an algorithm classification, 2) a method to optimize communication, 3) a tool which automatically classifies the algorithms and extracts the memory transfer requirements, and, 4) an evaluation of the theory and the tool on a benchmark set and real-life applications. We see the algorithmic species and memory transfer requirements as a solid base for current and future work related to parallel programming.

Future work on algorithmic species can expand the classification to more types of algorithms 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 [30]) can be incorporated before classifying programs in order to extract even more parallelism. As future possible use, next to the performance-centric uses we presented so far, we foresee algorithmic 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


