Research on AGAPIA language, compiler and applications

by

Ciprian I. Paduraru

A dissertation
submitted in partial fulfilment of the
requirements for the degree of

Doctor of Philosophy in Computer Science

Faculty of Mathematics and Computer Science
University of Bucharest

2015
Bucharest, Romania

Doctoral committee:

Prof. Dr. Gheorghe Ştefănescu, *Supervisor, University of Bucharest*

Prof. Dr. Emil Slusanschi, *Reviewer, Politehnica University of Bucharest*

Assoc. Prof. Dr. Florin Crăciun, *Reviewer, Babes-Bolyai University Cluj-Napoca*

Assoc. Prof. Dr. Traian Şerbanuta, *Reviewer, University of Bucharest*

Contents

Chapter 1

Introduction

As distributed applications became more commonplace and more sophisticated, new programming languages and models for distributed programming were created. The scope of these languages was to increase expressiveness and productivity. However, there are aspects still unresolved which causes more efforts put from programmers to create distributed applications. In the process of writing programs for parallel systems with distributed memory, using a common solution, such as MPI, users are concentrating on a set of sequential steps, create multiple tasks that can run concurrently and then handle their communications and synchronization explicitly. Before doing the implementation in a programming language, users are thinking on the architecture of the program as something more appropriate to a data flow diagram [80] where different entities are computing and exchanging data. Because of the sequential style to write a program, it is often hard to understand exactly what the interactions between the entities are. This could have some negative consequences to the programs' implementations. In the first place there is a lack of modularity which makes code harder to follow, specially to understand the underlying communication and synchronization. Another drawback is that programs can become more error prone and harder to maintain.

AGAPIA language was originally introduced in [28] (version 0.1) and improved to allow recursion in [75] (version 0.2). My main contributions were:

1. To build a toolchain for programs compilation and execution

2. Analyse categories of applications that are suitable for AGAPIA

3. Propose an extension of the language by adding patterns

4. Temporal pointers and analyse some schedulers which can be used for modules' execution

The goal of AGAPIA is to express maximum parallelism on both multi-core and many-core platforms while providing an intuitive programming model with modular, reusable code and transparent communication. The main goal of this thesis is to analyse the potential usage of AGAPIA, its backend for compilation and execution, and make a plan for future development. By taking advantage of the transparent communication model and high level statements intended to simplify the development process, the implementation of distributed programs become modular, easier to write and closer to the original solution formulation. Because the AGAPIA code is composed mostly from C/C++ language code plus a few specific language constructs and specifications it is expected that users can easily understand this new language. The potential of AGAPIA is proven through the implementation of some of the well-known patterns and applications in distributed computing. By "parallel implementation" we understand both parallel implementations with shared memory and distributed computing. Most of the programs written in AGAPIA have the same source code for both shared and distributed memory models - the exceptions are when users want to take advantage of the shared memory and use their own data structures instead of AGAPIA's "buffer" data type. All the advantages of using a high-level programming language usually comes with a cost - a performance penalty in either performance, memory or both. As this thesis will prove these are minimal when comparing AGAPIA implementation for several types of problems with their counterpart in MPI and C++.

The presentation is structured in two parts: the first part (Chapter 2,3 and 4) discusses about AGAPIA language, programming basis and its backend architecture and implementation, while the second one analyses categories of applications where users can take advantage by using AGAPIA.

Chapter 2 presents basics about AGAPIA language. It starts by explaining how to get started with coding simple programs, the syntax of the language and basics about how the communication works. In the continu-

ation, high level statements and patterns are presented together with their semantics, potential usage and their backend implementation.

The tool chain of AGAPIA compilation and programs execution is presented in Chapter 3. It starts with a global view over how modules get executed. Then it describes how the interpreter and code analyser tools work together with the current possible deployment types and macros. Chapter 4 contains a discussion on a few types of schedulers used for modules execution.

Chapter 5 has four sections. The first two presents two types of applications that are suitable to implement in AGAPIA: dataflow and wavefront applications. Section 1, contains a motivation why AGAPIA is suitable for dataflow programming, along with the implementation of one distributed signal processing application. The implementations will demonstrate that by using this language programmers can be more productive with minimal performance losses. The second section presents how to benefit from using AGAPIA in parallelizing the wavefront pattern. Programmers can solve these types of problems for parallel computation almost with the same effort as their corresponding serial implementation. Again, the performance/memory overheads are minimal.
The last two sections propose an extension for the current version of AGAPIA and a comparison to another new programming language for parallel computing. Thirth section provides another perspective for using the language: spatio-temporal data streaming applications. It is a conceptual discussion of future implementation, not included in the current compiler version. Can be used for building applications that run on moving objects such as airplanes, cars, and so on. With very high-level specifications, users could easily build applications that takes spatio-temporal data streams as an input and produces streams as outputs for use by other applications such as actuator controls. By providing ready to use modules for data manipulation AGAPIA enables a separation of concerns: application programmer can focus on their application model rather than low-level details about data manipulation and communication. Finally, the last section contains a comparison between a new programming language for parallel computing, X10, and AGAPIA, including an analysis for using X10 in its runtime.

Finally, Chapter 6 presents other ideas for future development and a conclusion.

Most of the results presented in this thesis can be found in these publications:

1. C.I. Paduraru, *Dataflow programming using AGAPIA*, Proceedings ISPDC 2014, IEEE, CPS pp. 327-334 (conference, rank C)

2. I.T. Banu-Demergian, C.I. Paduraru, and Gh. Stefanescu, *A new representation of two-dimensional patterns and applications to interactive programming*, Proceedings FSEN 2013, LNCS 8161, pp. 183-198. Springer, 2013 (conference, LNCS).

3. C.I. Paduraru, *A new online load balancing algorithm in distributed systems*, Proceedings SYNASC 2012, IEEE, CPS, pp. 327-334 (conference, rank C).

4. C.I. Paduraru, *An online load balancing algorithm for a hierarchical ring topology*, Proceedings ICCCC 2014, confirmed for publication International Journal on Computers, Communications & Control, No 6, 2014 (journal, rank C).

5. C.I.Paduraru, *Distributed programming using AGAPIA*, International Journal of Advanced Computer Science and Applications, Vol 5, No 3, 2014 (journal, rank D).

6. C.I. Paduraru, *A Greedy Algorithm for Load Balancing Jobs with Deadlines in a Distributed Network*, International Journal of Advanced Computer Science and Applications, Vol 5, No 2, 2014 (journal, rank D).

Note: For the evaluation process, I choose the current rules, applied after 01.10.2013.

Chapter 2

AGAPIA language

AGAPIA language was originally introduced in [28] (version 0.1) and improved to allow recursion in [75] (version 0.2). The goal of AGAPIA is to express maximum parallelism on both multi-core and many-core platforms while providing an intuitive programming model with modular, reusable code and transparent communication. This chapter begins with a presentation about AGAPIA's basics: how modules are defined and connected. Then it continues with some high level statements, programming patterns and their semantics.
In AGAPIA, a module (or program) is the basic programming block. By using composition operator between modules users can create applications with complex data flow. A program can be viewed as a dataflow graph where nodes are modules and links define the data flow between nodes. Recursion inside nodes is possible and a node can become this way a sub-graph. An important feature of AGAPIA model is that the dataflow graph is dynamic. New node instances, links and data flow paths can be created at runtime. This can improve the potential parallelism by dynamic task creation at runtime, which is usefully for applications such as radio cognitive or other applications that generate tasks depending on the intermediary computed result. Another important feature is that users do not have to define explicitly the links between nodes. Each module has input and output interfaces and the composition of modules will try to match the interfaces and automatically create links between inputs and outputs if the interfaces match according to the composition operator. An AGAPIA program code is mostly composed of C language code and a few high level statements, composition operators and interfaces descriptions. This decision was made to support legacy code

and to be more appropriate to well-known programming languages. Users can also include external libraries or code to facilitate the development of complex programs in AGAPIA.

2.1 Modules, basic communication and syntax

The basic block in AGAPIA programming is the `module`. A module has four input/output interfaces. The input can be received in north and west sides while output could go to east and south sides. Each interface could contain zero, one or more variables. A module's interface could be represented as a tuple of interfaces: `(west | north | east | south)`. The interface of the module in Fig. 2.1 is `( int | string ; float | nil | int,int ; int )`. There is a significant difference between symbol "`;`" and "`,`" when used in the modules' interfaces. This will be discussed in details in the next sections but for now, the first one separates data for different processes - which are similar to `struct` symbols in C/C++ programming language - while the second one separates different input items inside data for a single process. In Fig. 2.1 data for different processes are separated by rectangles.

```
module main{listen a:int}{read s:string ; b:float}
{
   // ..source code for program..
}{speak nil}{write c:int, d:int ; e : int}
```

The basic operation for obtaining higher-level programs is the composition. In the pictures below all the three composition operators that can be defined over two programs `A` and `B` are shown, along with the necessary restrictions and resulted interfaces.

- Horizontal (Spatial) composition: A#B. Resulted program interface is: (west(A) | north(A) ; north(B) | east(B) | south(A) ; south(B)).
- Vertical (Temporal) composition: A % B. Resulted program interface is: (west(A) ; west(B) | north(A) | east(A) ; east(B) | south(B)).
- Diagonal composition (Figure 2.4): A $ B. Resulted program interface is: (west(A); north(A); east(B), south(B)).

Two types of dependencies can be defined between modules:

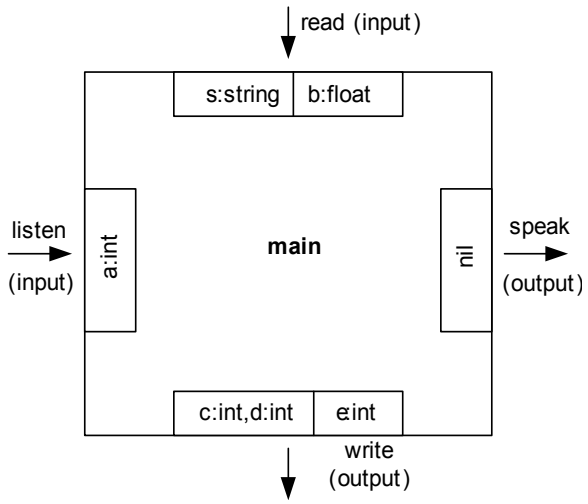

Figure 2.1: Simple program in AGAPIA

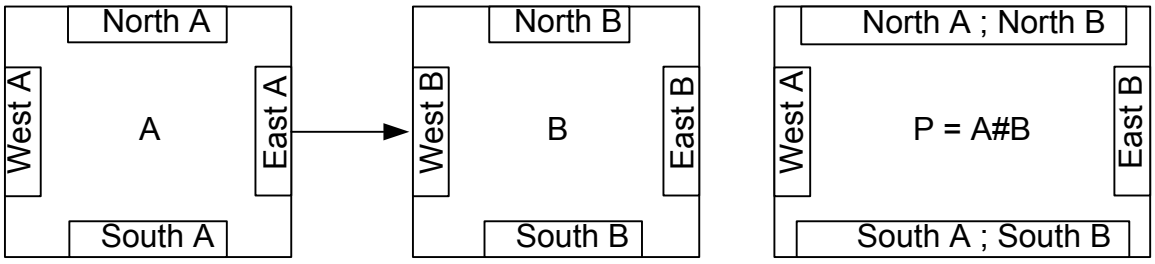

Figure 2.2: Horizontal composition. East(A) should match west(B)

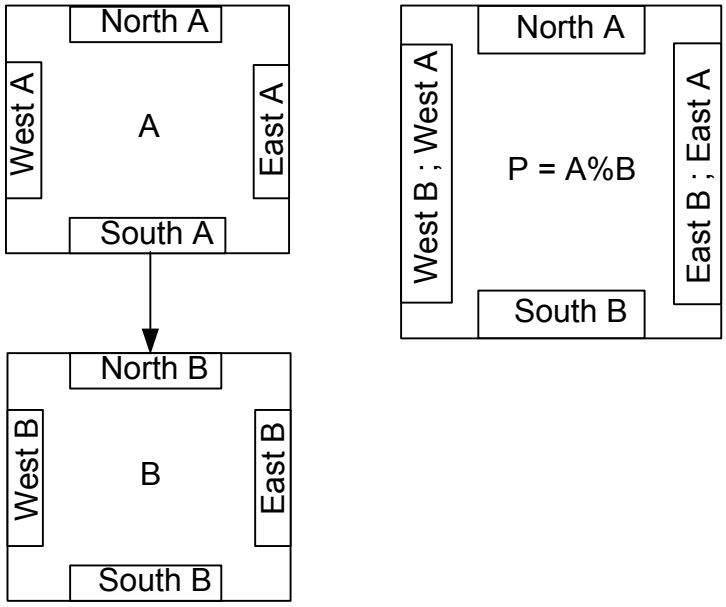

Figure 2.3: Vertical composition. South(A) should match North(B)

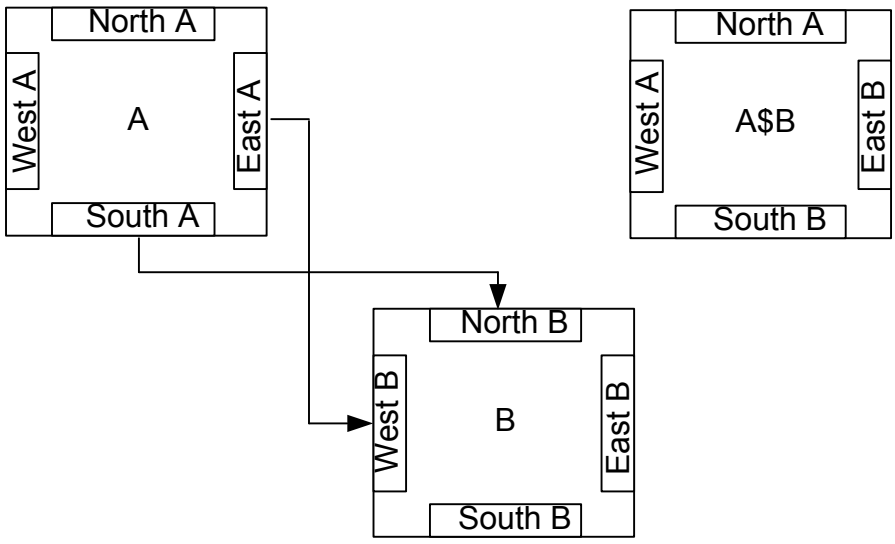

Figure 2.4: Diagonal composition. Both output interfaces of A should match the input interfaces of B

- north-south (or read-write) dependency: can occur in the vertical or diagonal composition.
- east-west (listen-speak) dependency: can occur in the horizontal or diagonal compositions.

A dependency exists if the interface on the corresponding side is not nil. Dependencies are usefully when coordinating the execution and preventing a program being executed before another one. For example, the diagonal composition could have both types of dependencies and it can be usefully when implementing barriers. These dependencies, which are automatically created when composition operators are used, represents the links of the AGAPIA program's data flow graph.

The original syntax of AGAPIA v0.2 [75] language is presented below:

```
Interfaces
SST ::= nil|sn|sb|(SST U SST) | (SST,SST) | (SST)*
ST  ::= (SST) | (ST U ST) | (ST; ST) | (ST; )*
STT ::= nil | tn | tb | (STT U STT) | (STT,STT) | (STT)*
TT  ::= (STT) | (TT U TT) | (TT; TT) | (TT; )*


Expressions
V ::= x : ST | x : TT | V (k) | V.k | V.[k] | V@k | V@[k]
E ::= n | V | E + E | E * E | E - E | E/E
B ::= b | V | B && B | B||B | !B | E < E

Programs
W ::= null | new x : SST | new x : STT | x := E |
      if (B){ W }else{ W }| W;W |while (B){ W }

M::= module module_name{listen x:STT}{read x:SST}
    { W }
    {speak x : STT}{write x : SST}

P::= nil | M | if (B){ P } else{ P } | P % P | P # P | P $ P|
    while_t(B){P}|while_s(B){P}|while_st(B){P}|
```

The modified version used in this thesis and in the current version of the compilation tools, called `AGAPIA v0.2+`, is based on AGAPIA v0.2 language and makes a few modifications in order to be more friendlier for users. Its syntax is presented below.

```
Interfaces
SI::= nil|int|bool|float|string|buffer| (SI,SI) | (SI [])
MI::= (SI) | (SI;SI) | (SI;)*

Expressions
V ::= x : MI | V(k) | V.k | V.[k] | V@k | V@[k]
E ::= n | V | E + E | E * E | E - E | E/E
B ::= b | V | B && B | B || B  | !B | E < E

Programs
W ::= null | new x : SI | x := E |
      if (B){ W }else{ W }| W;W |while (B){ W }
     ..and all other C\C++ language constructs

M::=module module_name{listen x:MI}{read x:MI}
    { W }
    {speak x : MI}{write x : MI}

P::=nil | M | if (B){ P } else{ P } | P % P | P # P | P $ P|
    while_t(B){P}|while_s(B){P}|while_st(B){P}|
    foreach_t(B){ P } | foreach_s (B){ P}|foreach_st(B){P}|
    for_t(E; B; E)|for_s(E;B;E)|for_st(E;B;E) |
    gather_s|gather_t|scatter_s|scatter_t|map_t|map_s
    reduce_s|reduce_t| scan_s | scan_t
```

To be more appropriate to common programming languages some changes were made to the syntax of AGAPIA v0.2. `SI` represents a simple interface declaration. Structures can be obtained by adding toghether simple data types: `(SI, SI)`. `( SI[] )` represents an array of simple data types. Instead of using `sn/tn` or `sb/tb` the decision was to merge them and use just `int` and `bool` for both temporal and spatial interfaces, but without losing the information from which category they are. Two new basic data types, `string` and `buffer` types were added for storing strings and sending buffers between distributed programs in an easy way. Another imporntant feature

was added to make AGAPIA code closer to common programming languages: users are now allowed to use the entire C/C++ language constructs and existing code, libraries inside atomic modules (the concept of "atomic" module will be introduced in the next section).

`MI` is used for defining a program interface and it basically uses `SI` for this. From (or in) a module interface, the output (or input) can flow to one or more other modules. `(SI;SI)` represents two different processes while `(SI;)*` is an array of processes. For example, if we are vertically composing a module `M` with a `foreach_s` statement, then the south output interface of `M` should be something of type `(SI;)*`. In AGAPIA programs users can use all type of C/C++ language constructs. At the high-level programs section, the language offers simple and high level composition and flow branching statements. The `foreach` and patterns `gather`, `scatter`, `map`, `reduce`, `scan` were added in order to increase the productivity. These patterns will be explained in details later in this chapter.

Because array of processes are something AGAPIA specific, some more details must be given. A simple array of structures (named `A`) of a pair containing an `int` and a `bool` is defined as `A:(int, bool)[]`, while `A[i]` is used to access an index from this array. An array of processes (named `V`) with each process containing the same pair is defined as `V:(int, bool;)*`, while `V@[i]` is used to access an index. The main difference is that elements from a simple array can't be split to different AGAPIA programs just by composition, while the array of processes can. If there is a program which has as spatial input an array of processes and inside this program there is a composition like `M # N # Z`, each one accepting a simple pair of `int` and `bool` as spatial input, then the first three indices from the array will go in the right order to `M`, `N` and `Z`. It is best to use array of processes when dealing with AGAPIA's high level iterative statements (`for`/`each`/`while` or patterns). In the case of the above example with `M # N # Z` then it suffices, and it is even clearer, to have a spatial input like `( (bool,int) ; (bool, int) ; (bool, int))` - data for three different processes, one for each program.

2.2 High level statements and their semantics

To change the input/output flow at runtime by conditional branching, we can use the `if` program. It has the following syntax: `if (condition) P_IF else P_ELSE` , where `P_IF` and `P_ELSE` are also two programs. There are two restrictions regarding these two programs:

- `P_IF` and `P_ELSE` programs should have the same interfaces (and the current version of compiler needs even input interfaces with the same variable names) to make the input/output matching correctly.
- `condition` can only contain variables defined in the input interfaces of `P_IF` and `P_ELSE`.

Fig. 2.5 shows how an `if` program looks like inside. An `if` program becomes a node in the dataflow graph, where the `P_IF` and `P_ELSE` are child nodes. When inputs are received they are buffered in a data structure inside this node until all the variables referenced in condition can be evaluated. After the condition is evaluated the AGAPIA's runtime automatically connects the inputs from west and north interfaces to the chosen child node, and outputs from the child node to south and east output interfaces. This is an important feature that AGAPIA provides: dynamic links creation depending on a condition which evaluates at runtime.

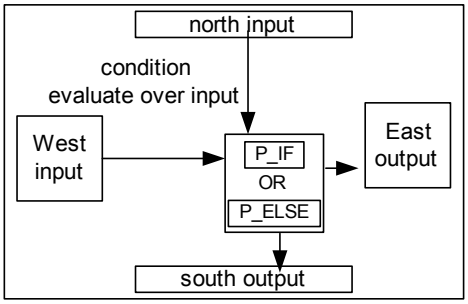

Figure 2.5: Inside an "if" program in AGAPIA

To create iterations of nodes and links at runtime, AGAPIA v0.2+ provides the following statements: `for_s, for_t, for_st, foreach_s, foreach_t, foreach_st, while_s, while_t, while_st`. These are doing the same

things as the `for` and `while` statements in the common programming languages, with one exception: for each iteration an AGAPIA program is instantiated and linked with other programs. The type of the composition between the programs instantiated in the iteration is indicated by the letters that comes after underscore in the statement's name: "s" means spatial (#), "t" temporal (%), and "st" diagonal ($). This rule is valid for all types of "`for/each/_`" and "`while_`" statements. As we can see from the syntax, these statements become AGAPIA programs too. Fig. 2.6, 2.7, 2.8 shows how the `for`, `foreach` and `while` programs look internally for each iteration type. The figures are conclusive about how the input/output flows inside.

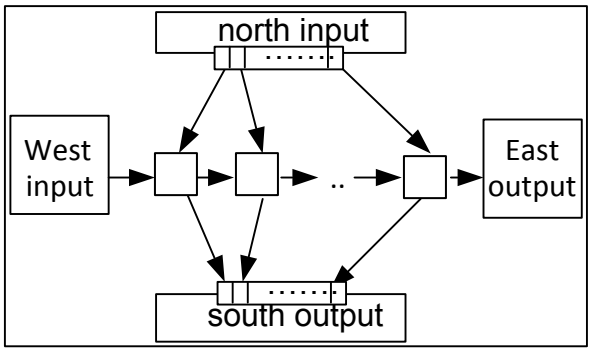

Figure 2.6: Inside an `for_s/foreach_s/while_s` program. There could be a spatial composition between consecutive iterating programs.

Similar to `if` program restrictions, the condition of a `while_` program refers to variables defined in the input interface of the program it is iterating on. So if we have a program like `while_t(condition) P`, then condition can refer to the north and west input interfaces of `P`. If user knows the number of iterations then it is better in terms of performance to use the `for/foreach` statements, because in background all internal instances could be created directly allowing the maximization of parallelism. There is one big difference between `for_` and `foreach_`. `for_` should be used when we want to impose a certain order on how the internal programs are instantiated. On the other hand, if we have a program like `foreach_s(n) P`, and the `n` value does not depend on the input/output of `P`, and `P` does not have any listen-speak dependency, then those `n` instances of type P could be created and executed in parallel in any order. If we use `for_s` instead, then the internal instances will be instantiated in the order of the iteration (although they

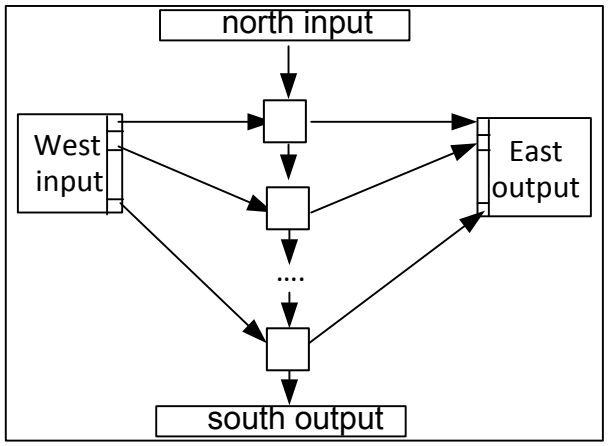

Figure 2.7: Inside an `for/each_t/while_t program`. There could be a temporal composition between consecutive iterating programs.

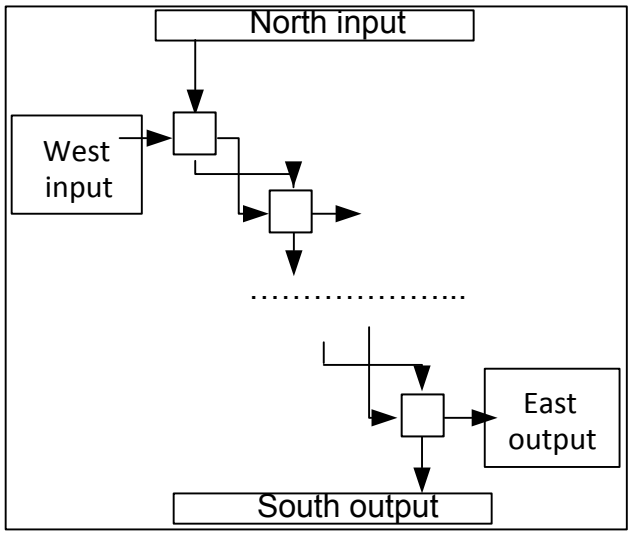

Figure 2.8: Inside an `for_st/foreach_st/while_st program`. There could be a diagonal composition between consecutive iterating programs.

could also run in parallel, if there is no listen-speak dependency between them). Speaking in terms of interfaces, the iterating side of these programs has as interface an array of processes. If the interface of P is `(west | north | east | south)`, then the `for_s / foreach_s/while_s` interface is `(west | (north;)* | east | (south;)*)`.

Some clarifications must be made about how the parameters containing arrays of processes are sent between programs. If the program which sends input for an array of processes is an atomic program, then all data is sent in a chunk. The same thing happens if the receiver program is an atomic program (we achieve this by buffering the inputs and detecting when all input expected arrived). If none of the programs are atomic, and both programs are connecting an `(int;)*` to an `(int;)*` then the mapping is made on the same indices, one to one. If not, and for example the array of processes has type `(int;)*`, then the input for this array of processes can contain in its specification just a single array of processes and this should be the last element. A correct example is connecting `(int;)*` to `(int; int;...;(int;)*`. Connecting `(int; (int;)*; int; ...)` to `(int;)*` is not allowed, because there is no mechanism to know how much the second item in the specification will expand. Considering these recursively, the compiler knows exactly the order of how elements comes in the array. Also, for optimization purposes, if communicating programs are not atomic then we send array indices individually. Imagine a program which does some parallel computations and set individual items in an array of processes in the south interface. If this program is vertically composed with a `foreach_s` statement then sending array indices individually is a performance advantage. Each time an item is sent to the `foreach_s` program a new instance inside of it can start, maximizing this way the potential parallelism.

The other high level statements of AGAPIA v0.2+, which represents some ready to use common patterns, were created in order to improve the productivity in building complex applications. `Scatter` is used to transform from a simple array to an array of processes while `Gather` transforms an array of processes to a simple array. Common usage examples includes creating an array of tasks then splitting each item to a different program instance or receiving results from different programs in a simple array. The `Map` pattern can be used to apply an operation (represented by a given module) over a set of items and produce another set of items. Examples of usage include

image processing, ray tracing or Monte Carlo sampling. AGAPIA v0.2+ also provides `Scan` and `Reduce` primitives which does the typical operations in logarithmic time over a set if inputs coming from different programs. Other kind of patterns specific to dataflow computing, such as `pipeline` or `wavefront` can be easily expressed with composition.

2.3 "Buffer" data type

As mentioned in the first section of this chapter, one of the new data type added to the syntax of AGAPIA v0.2+ is the `buffer` type. Its main role is to help programmers optimize the communication times for their applications in a transparent way and to make their life easier when modules need to exchange chunks of data.

A `buffer` is a data type that allow users to allocate, read and write memory. It has three members inside: the address of the memory where this buffer holds data `(void * m_pData)`, the size of the data `(m_iBufferSize)` and the maximum size of the data `(m_iMaxSize)`. To initialize a buffer defined as `buff : buffer` the following code can use: `buff = < pDataBegin, iDataSize>`, where `pDataBegin` is an address of memory (in the virtual memory address space of the current process) and `iDataSize` is the size of data. To access data from it one can call `buff->m_pData` and `buff->m_iBufferSize`. User can use this data type to serialize/deserialize data or to send already serialized data between modules in an easy way.

The most interesting part of this data type is its behaviour when being transmitted between modules through input/output interfaces. The objective is to avoid unnecessary copies of data. If the target platform for executing the program is a parallel one with shared memory then the only copy needed is the address of memory and the size of data, which means two integers (shallow copy). In a distributed platform we have two cases. If the modules which are transmitting between them a buffer type are executed by the same machine then there is no need to copy the entire data in the buffer so again it will copy just two integers. However, if modules are executed on different machines then a deep copy of data is needed.

2.4 Parallel programming patters in AGAPIA

Identifying themes and idioms that can be codified and reused to solve specific problems in parallel and distributed computing is an important topic in computer science. The semantics of each pattern is the same for every programming language, but the way to implement it differs between programming languages. When dealing with parallel and distributed programming the user has to take an important decision when choosing the programming language because each one has its own advantages and disadvantages.

This section presents how some of the most popular patterns in parallel and distributed computing can be implemented by using AGAPIA language. By taking advantage of its semantics and high level statements, the patterns implementations becomes easier to write, more modular and less error prone. Understanding patterns and their implementation in AGAPIA helps understanding what problem categories can benefit by using this language. The patterns presented in this section are: `Fork-Join`, `Map`, `Reduction`, `Scan` and `Stencil`, `Pipeline` and `Filters`, `Gather`, `Scatter` and `Geometric Decomposition`.

2.4.1 Fork-Join

The Fork-Join pattern lets control flow fork into multiple parallel flows that rejoin later [59]. It is the base of many patterns and its main usage is to split a process (parent) into two or more parts that could be computed in parallel. Below is an example of a simple implementation of this pattern in AGAPIA.

By simple composition of programs we can create a Fork-Join pattern in AGAPIA. Because of the read-write dependency, the program `Join`, described in Fig. 2.9, knows that it needs to get input from both programs `A` and `B` to continue execution. Both programs can be executed in parallel and having the listen-speak dependency between `A` and `B` guarantees that `A` start before `B`. The easiness of the implementation comes from the fact that the user just needs to write the correct interfaces for programs and use the composition operators.

```
module ForkExample {listen nil}{read nil}
{
    A#B
```

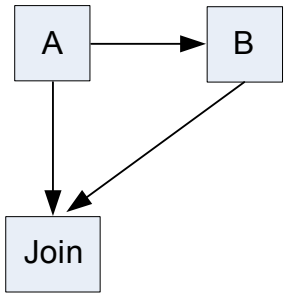

Figure 2.9: Example of a Fork-Join. The process that execute program A spawns a new process that execute program B, continues execution in parallel, and after some time they join.

```
    %
    Join
}{speak nil}{write nil}

module A{listen nil}{read nil}
{ .. code ..}
{speakta:int}{write sa:int}

module B{listen ta:int}{read nil}
{ .. code ..}
{speak nil}{write ba:int}
module Join{listen nil}{read sa:int ; sc:int}
{ .. code ..}
{speak nil}{write nil}
```

Creating a fork-join in MPI is possible by using the `MPI_Comm_Spawn` function. But there are some disadvantages over the AGAPIA solution. First thing is that user has to write different code/executable for the parent and child process. Then, communication between spawned child and joining is more complicated than in AGAPIA - user have to be carefully about calling `MPI_Wait` and `MPI_Finalize` in the right places and use the correct communication channel and process id.

2.4.2 Map

The map pattern replicates a function over every element of an index set. The set can be abstract or associated with the elements of a collection [59]. Usually, it produces a new set of values, like in Fig. 2.10. This pattern simplifies the process of creating programs for solving problems such as image processing, Monte Carlo sampling or ray tracing, in a parallel environment.

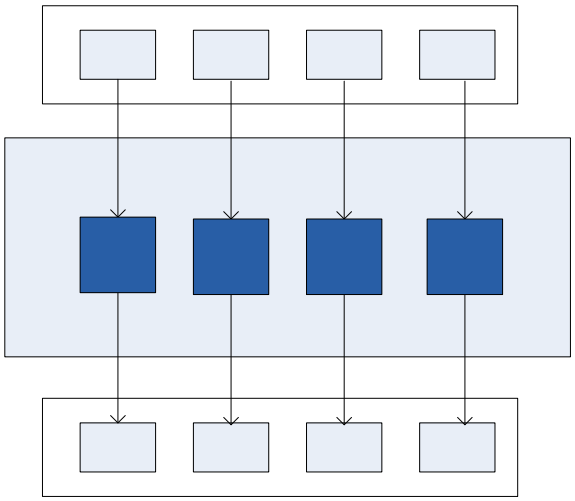

Figure 2.10: Map pattern example. The input is a set of values, it applies the same function over all items in the set and usually obtain another set of values.

A Map can be defined by hand if complex situation needs. To exemplify this and show the background implementation of a Map, an implementation in AGAPIA of this pattern is presented below. In this case the set is an array of processes of numeric values. A simple function which multiplies a number by 2 is used (the function being replicated is called elemental function).

```
module MapExample{listen n : int }{read inputs:(int;)*}
{
   foreach_s(n)
   {
      ElementalFunc
   }
}{speak nil}{write outputs:(int;)*}
```

```
module ElementalFunc{listen nil}{read in:int}
{
    out = in*2;
}{speak nil}{write out:int}
```

The first `n` elements from `inputs` will get through the `ElementalFunc`, get multiplied by 2, and then goes to the correct index in the "outputs" array. Because there is no listen-speak dependency, all `ElementalFunc` tasks can be executed in parallel. Compiler knows how to send the correct inputs from array to each `ElementalFunc` because the elements in the `inputs` array will be available in the order they came in. Then, each input received will be sent to the correct iteration of the foreach loop. If a needed input is not available yet in the array, the corresponding `ElementalFunc` instance will wait until it becomes available.
AGAPIA v0.2+ provides an existing implementation of this pattern that users can use to simplify a program implementation. Users have to define what the map operations does on the input element with the correct input and output types in the interface and to give as parameter the number of items the map should apply to. Map automatically adjusts depending on the type of composition and data types. An example of usage where the map is applied over the output of `n` modules of type `A`, then results are used as input for `n` modules of type `B` is given below:

```
foreach_s(n){A}
%
Map(ElementalFunc,n)
%
foreach_s(n){B}.
```

To implement this in MPI we first need to send the input to different processes (either calling a `Scatter` operation, or using parallel I/O where processes read data on their own). Then, these processes compute the desired operation - the elemental function - and finally, a gather operation will be used to copy the results back to a root process. As `Scatter` and `Gather` operations are implemented, we need to create another communication channel to contain just the processes that needs to run the elemental function. Also, for the two operations to complete, the root and workers should call them in the correct order. These disadvantages make this pattern implementation in MPI a slightly more error prone and harder to understand that it

is by using AGAPIA which provides a clearer picture for users.

2.4.3 Gather and Scatter

The Gather pattern reads values from a set of processes and stores them in a collection. The Scatter pattern is the inverse of the Gather pattern - the values from a collection are distributed to multiple processes. These are base operations for parallel programming with distributed memory and are also implemented in MPI by `MPI_Gather` and `MPI_Scatter` routines. Below are both operations implemented in AGAPIA.

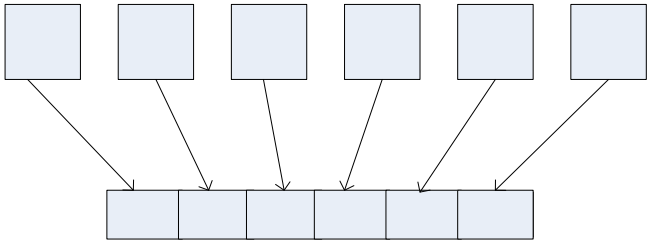

Figure 2.11: Gather example.

```
module Gather{listen n : int}{read v:(int;)*}
{
    for (int i = 0; i < n; i++)
        out[i] = v@[i];
} {speak n : int}{write out : int[]}
```

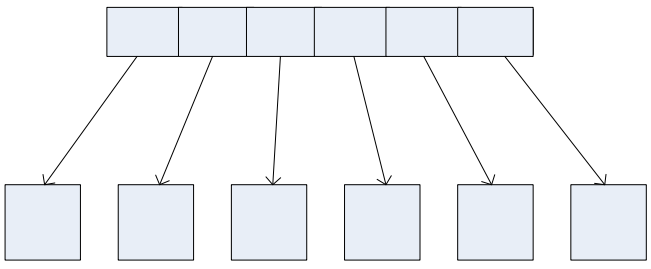

Figure 2.12: Scatter example.

```
module Scatter{listen n : int}{read int: int[]}
{
    for (int i = 0; i < n; i++)
        v@[i] = in[i];
}{speak n : int}{write v:(int;)*}
```

Both patterns implementations are using the temporal interface for transmitting the number of items in the arrays. User could also choose to transmit the number of items through the spatial interface, but then he needs some identity operators to match correctly the interfaces. Examples can be found in [75]. AGAPIA v2.0+ already provides implementations for `Gather` and `Scatter`. A parameter representing how many items should be gathered/scattered must be given. An example to gather the results from a `foreach_s` statement in an array is: `foreach_s(n) A % gather(n)`. This will gather the outputs from the south interface of all `n` modules of type `A` in a simple array. The gather/scatter statements automatically define and checks the input/output interfaces depending on the source/destination of data.

2.4.4 Pipeline

The pipeline pattern is usefull when the computation involves performing a calculation on many sets of data and this calculation can be viewed in terms of data flowing through a sequence of stages. It is common in the implementation of real time applications, signal processing, online applications, compilers or systolic algorithms. There are two types of pipelines: linear pipelines where all stages that are applied over an input are executed serially and non-linear pipelines which can contain stages that could execute in parallel (See also Fig. 2.13 and Fig. 2.14). Both types can be easily implemented in AGAPIA. Below is presented an implementation of a linear pipeline.

```
module Pipeline {listen InImagesArray :(image;)*}{read nil}
{
    for_t (int i = 0; i < NrImages; i++)
    {
        S1 # S2 # S3 # S4
    }
}{speak nil}{write OutImagesArray:(image;)*}
```

We can make sure that a certain stage program can't be executed in parallel on different levels by creating a write-read dependency. An example of this kind of behavior can be obtained for program S1 like this:

```
module S1 {listen img: image}{read check:int}
{
.. code to compute the imgout..
}{speak imgout:image}{write check:int}
```

We can even play with groups of dependencies between stages on the same level. This is all about how the user put dependencies between programs. Non-linear pipelines can be easily obtained too (See Fig. 2.14).

A program with a non-linear pipeline like in Fig.2.14 can be implemented by changing the source code inside the `for_s` statement from the previous pipeline implementation with: S1 # (S2 % S3) #S4. Note that modules `S2`

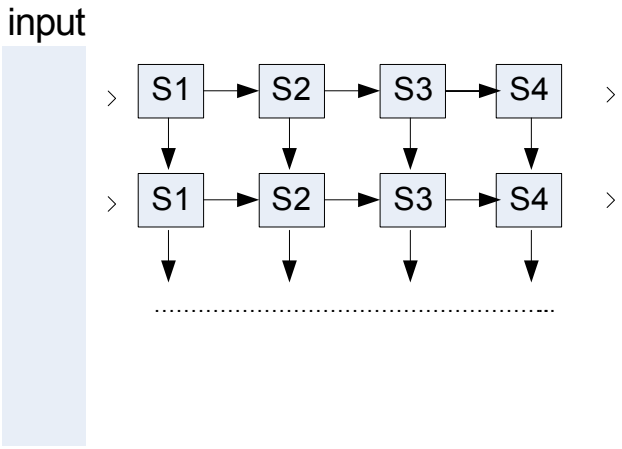

Figure 2.13: Linear pipeline. There are read-write dependencies between levels, and listen-speak dependencies for consecutive programs of a level.

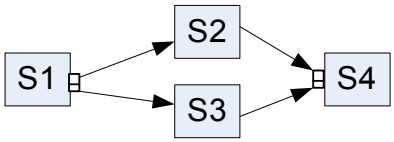

Figure 2.14: Non-linear pipeline. S2 and S3 can be executed in parallel.

and `S3` can run in parallel only if their connection on the spatial side is nil. By combining the pipelines with `if` statements, we can easily create some other kind of patterns like filters.

If we consider that a distributed system could run multiple non-linear pipelines in parallel then an implementation in MPI needs to use the dynamic process spawning or a custom scheduler created by user. The simplest way to do it, using dynamic process spawning, has some disadvantages. First, we need separate code files/executable for each component or group of components from pipeline that needs to be executed on different processes. This makes the code hard to follow, in contrast to AGAPIA where we have the entire code in one file, together with the entire pipeline flow. A second issue that appears often in pipeline applications is the diversity of parameters and data sent between components of the pipeline. In MPI we need several calls to `MPI_Send` and `MPI_Recv` functions. In AGAPIA the parameters are serialized and sent automatically according to programs interfaces.

2.4.5 Geometric Decomposition

The Geometric Decomposition pattern breaks data into a set of sub collections. The purpose is to give this data to different processes for parallel execution. Sometimes, it is not necessary to transfer the data, like in the case of programming for a shared memory model. Stencil operations, which are used in image processing and simulations, are good examples of usage for this pattern.
Below we present an example of an image filter skeleton implementation in AGAPIA which uses a shared memory model. The `Decomposition` program is responsible for breaking data: in our example it gives to each process an equal number of consecutive lines from the input image. The number of tasks in which we want to break the computation of filter over the image is decided in this program by a call to an external function defined by user and transmitted through the temporal interface further. The `Task` program is the one responsible for executing the given part of the image.

```
module Filter{listen nil}{read w:int, h:int, pixels:int[]}
{
     Decomposition
     $
```

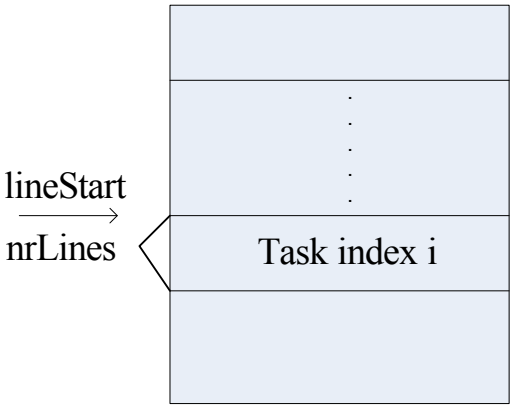

Figure 2.15: Image decomposition.

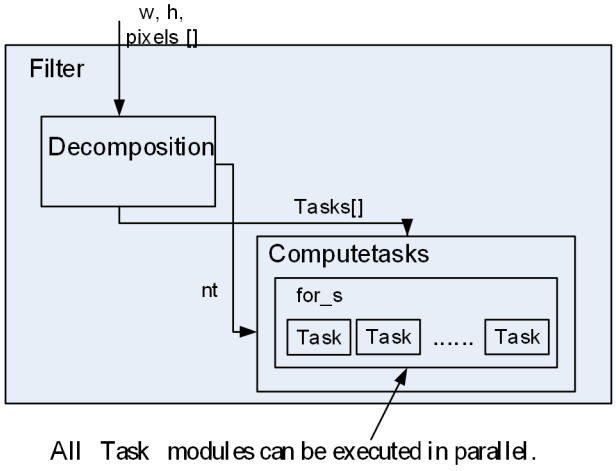

Figure 2.16: The flow of input and execution of Filter program in AGAPIA. The involved programs are represented by rectangles and their name is in the top-left corner.

```
     ComputeTasks
}{speak nil}{write nil}

module Decomposition{listen nil}
{read w:int,h:int,pixels:int[]}
{
     nt = Utils::GetNbOfTasks(w,h);
     for (int i = 0; i < nt; i++)
     {
          tasks@[i].lineStart= (h/nt) * i;
          tasks@[i].nrLines= h/nt;
      }
}{speak nt:int}
{write tasks:((lineStart:int, nrLines:int);)* }

module ComputeTasks{listen nt:int}
{read tasks:((lineStart:int, nrLines;int);)* }
{
      for_s (int  i = 0; i <  nt; i++)
      {
          Task
      }
}{speak nil}{write nil}
```

If the user wants to solve this problem in the distributed case, then the only necessary change to the source code is to distribute the pixels data instead of line start and number of lines.

An implementation in MPI will make the code more complicated because we need to serialize the parameter and image buffer and then scatter data from master to workers. Also, user has to:

1. Split code in two flows for master and workers
2. Be carefully with indices
3. Call the `MPI_Scatter` function on all processes that are doing tasks.

2.4.6 Reduction and Scan

A reduction combines every element in a collection into a single element using an associative combiner function [59]. Scan pattern computes all partial reductions in a collection. These two patterns could be used for a broad category of applications, including numerical analysis (dot products and row-column products in matrix multiplication, convergence testing for linear equations, etc.) or image analysis. Because scan does not differ too much than reduction from the AGAPIA implementation point of view, only the reduction operation is presented here. Below we present a reduction operation with an associative combiner function, implemented in AGAPIA. The tree has a span of length $\log_2 N$ where `N` is the number of nodes in the tree.

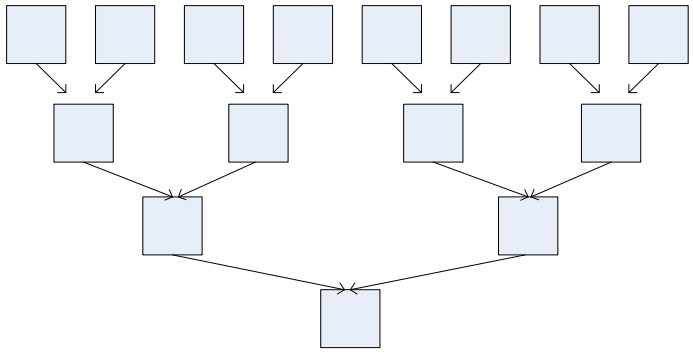

Figure 2.17: Tree reduction pattern for an associative combiner function.

The program Reduce receives as input an array of processes each one having an integer value. Inside, it uses a program `CombineFunc` which receives as input two integers values and produces as output a single value - the value resulted by combining the inputs. A simple example of `CombineFunc` could be the addition of numbers.

```
module Reduce{listen nil}{read v:(int;)*) }
{
  for_t (int i = 1; i <= log2(N); i++)
  {
     for_s(j = 1; j <= 2^i; j++)
     {
           CombineFunc
```

```
      }
    }
    ... use the result here ...
}{speak nil}{write nil}

module CombineFunc{ listen nil } { read a : int, b : int }
{
     c = a + b;
}{speak nil} { write c : int}
```

In this case, the `for_t` will spawn levels one by one, while the `for_s`, will spawn all tasks needed for that level. All tasks on a level can be computed in parallel because there is no listen-speak dependency between the `CombineFunc` program instancies (i.e., between tasks created at each level). On the other side, because of the read-write dependency, the computation respects the expected flow: the levels are guaranteed to be executed in the correct order.

Reduce and Scan patterns are already implemented in AGAPIA v0.2+ and can be re-utilized by users in order to improve the development process and to make the code clearer. They receive three parameters: the number of elements to reduce/scan, a module defining the function to combine the elements and a module defining the neutral element of the combination (needed when the number of elements is not a power of 2 ; Indeed, in addition to the implementation shown above, if the number of elements is not a power of 2 then we use this neutral element to add fictive elements until we get a power of 2). For example the sum reduce presented above where the `Source` produces `n` items and `Neutral` produces a neutral element as output without receiving any input, can be defined as: `Source % Reduce(n, ComputeFunc, Neutral)`. Reduce will automatically adapt to the type of composition used and performs type checking for the input values, Neutral and ComputeFunc.

MPI has two functions that implement these two patterns: `MPI_Scan` and `MPI_Reduce`. However it has some slightly disadvantages compared to AGAPIA. First, we need to create a separate communication channel for all processes implied in the process of scan/reduce, then, these functions acts like a barrier and needs to be called in the right order on all those processes. These things can make the code difficult to understand in comparison with

AGAPIA code, where the pattern help the user to keep a code closer to the natural way of the solution formulation.

2.4.7 Wavefront pattern

The Wavefront pattern appears in programs with data elements laid out as multidimensional grids and which have data dependencies between elements that resemble a diagonal sweep. This is very common for dynamic programming problems or systolic algorithms. The temporal interface in AGAPIA makes the implementation of this pattern to be easy and clear.

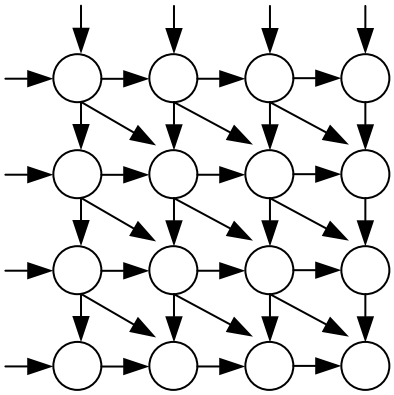

Figure 2.18: Data dependencies for the longest-common-subsequence problem.

To implement this pattern in AGAPIA we need first to analyze the data dependencies and make sure that we can send all data needed by a program through its north and west interfaces. For example, the longest-common-subsequence problem has a diagonal dependency. If we consider that each cell (or group of cells) is a program instance responsible for computing the formula, then we need to transfer somehow that diagonal element from `F(i-1,j-1)` to `F(i,j)`. We can do this by sending first the item from `F(i-1,j-1)` to `F(i-1,j)` and then both data values from `F(i-1,j)` to `F(i,j)`. In the distributed memory model we also need to distribute the characters of the two arrays `A` and `B`. Below we present the main source code body (without initializations or data splitting). The `chA` and `chB` denotes the characters that each cell should compare when computing the value.

```
for_t(int i = 0; i < n; i++)
```

```
     for_s(int j = 0; j < m; j++)
     {
          ComputeCellValue
     }

module ComputeCellValue{listen left:int, chA:int}
{read up:int, diag:int, chB:int}
{
    // F[i][j] = max(F[i-1][j-1]+1*(A[i] == B[j]),
    //              F[i][j-1], F[i-1,j]);
    result = max( diag+1*(chA==chB), left, up);
}{speak result:int,chA:int }
 {write result:int,result:int, chB:int}
```

To implement this in MPI we have to create a scheduler by hand to compute the cells that are ready for execution - which takes some important time to code. Alternatively, one can use dynamic process spawn in MPI - but this has the same disadvantages as the previous examples. Splitting the code/executable for different cells computation and communication issues will transform an implementation in MPI into something much different that the simple sequential implementation and how to user thinks about the solution generally. In AGAPIA, after a cell is computed it sends the output further making other programs ready for execution. These will be automatically scheduled in the backend and the user concerns are just to use the correct recursion and initialization as in a sequential program.

2.5 Conclusion

The first part of this chapter made an introduction in AGAPIA language starting from its basis: syntax, modules and composition. Then it continued with low-level explanations about branching and iteration of programs at runtime and the features of the modified version used in this paper: AGAPIA v0.2+. This new version is supposed to make AGAPIA language closer to common programming languages by modifying the syntax for data types definitions, allowing the entire set of C/C++ code inside atomic modules, and by adding high-level statements such as `buffer` or some well-known programming patterns for distributed programming. The `buffer` is an important feature because it optimizes performance without any effort from user: deep

copy is performed only when the data inside buffer is copied between different machines and the entire functionality is transparent for users.
A collection of well-known programming patterns was also presented with their source code and low-level details about programs involved and connections. This set is composed of: `Fork-Join`, `Map`, `Reduction`, `Scan` and `Stencil`, `Pipeline` and `Filters`, `Gather`, `Scatter` and `Geometric Decomposition`. A few of them are already implemented in the compiler distribution. The exceptions are the `Pipeline` and `Geometric Decomposition` which, as demonstrated in this chapter, are very easy to implement by using programs' composition. The explanation of each pattern is accompanied by a comparison to its corresponding implementation in MPI language. Excepting Reduction, Scan, Gather and Scatter where MPI also provides buit-it routines, the AGAPIA implementations needs less effort to create, has a clearer source code and keeps a better high-level view over the program which means that it's less error-prone and easier to mentain.

Chapter 3

Description and implementation of AGAPIA's tool-chain

The purpose of this chapter is to present the process of writing and building an AGAPIA program, the tools used in this process and what is happening behind the scenes. The presentation is a top-down style starting with a practical presentation on how to write a simple program, add user defined source files or libraries and get an executable. This example will be used in the rest of the chapter to explain how the backend of AGAPIA works. The tools behind compiling AGAPIA programs are presented in deep together with the execution model, input/output flow, deployment macros and tips to optimize performance by minimizing the communication times. Concrete examples are used in order to get a better understanding over its backend and to provide a continuation point for future AGAPIA development.

3.1 Practical introduction on how to write and execute an AGAPIA program

A toy example is used to exemplify how to think, write and execute an AGAPIA program for solving the problem of Edge Detection in a given input image. The implementation uses the `Sobel` filter method [76] which we are supposing that is already implemented as a procedure (for serial execution) in a library written in C/C++ and called `SobelIterative`.

A typical input for this problem is the name of the input image file and the number of hardware processes which are available for executing tasks. The

first step to implement the program in AGAPIA is to design the solution from a high-level point of view, establish the modules, the links between them and their input/output interfaces.

The program should do three main things: read the input image, split the computation in different tasks, execute these tasks in parallel and save the output file. All these main operations will become modules: `READ`, `DISPATCHJOBS`, and `COMPUTEALLJOBS`. Module `MAIN`, which is the entry point of an AGAPIA program, will receive the user input and connect these three modules such that it can solve the problem. Modules `READ` and `DISPATCHJOBS` can run directly on the master machine (the place where the input image is located). We can specify this by adding `@MASTER` near the input interface declaration. Figure 3.1 shows a high-level view of modules and input/output flow between them.

The code for this program will be written in a file created by user (named "`agapia.txt`" in this example). One possible implementation for module `MAIN`, where the `READ` and `DISPATCHJOBS` module are composed horizontally and the result is composed vertically with `COMPUTEALLJOBS` is given below:

```
module MAIN{listen nil}{read filenamein:string;nrofprocs:int}
{
    READ # DISPATCHJOBS
    %
    COMPUTEALLJOBS
}{speak nil}{write nil}
```

The `READ` module gets the input from the `MAIN` module then use functions (which are supposed to be given by user in a separate `.h/.cpp` file for reading TGA files format) to read the input image and send it further to `DISPATCHJOBS`.

```
module READ{listen nil}{read filenamein:string}@MASTER
{
    TGAReader::STGA* tgaFile = new TGAReader::STGA();
    TGAReader::loadTGA(*filenamein, *tgaFile);
    printf("File received to read from:%s\n",*filenamein);
    imageaddress = (int) tgaFile;
}{speak imageaddress:int}{write nil}
```

Module `DISPATCHJOBS` will read the number of processes (`nrofprocs`) from the `MAIN` module and the TGA image address from `READ`. Its main purpose

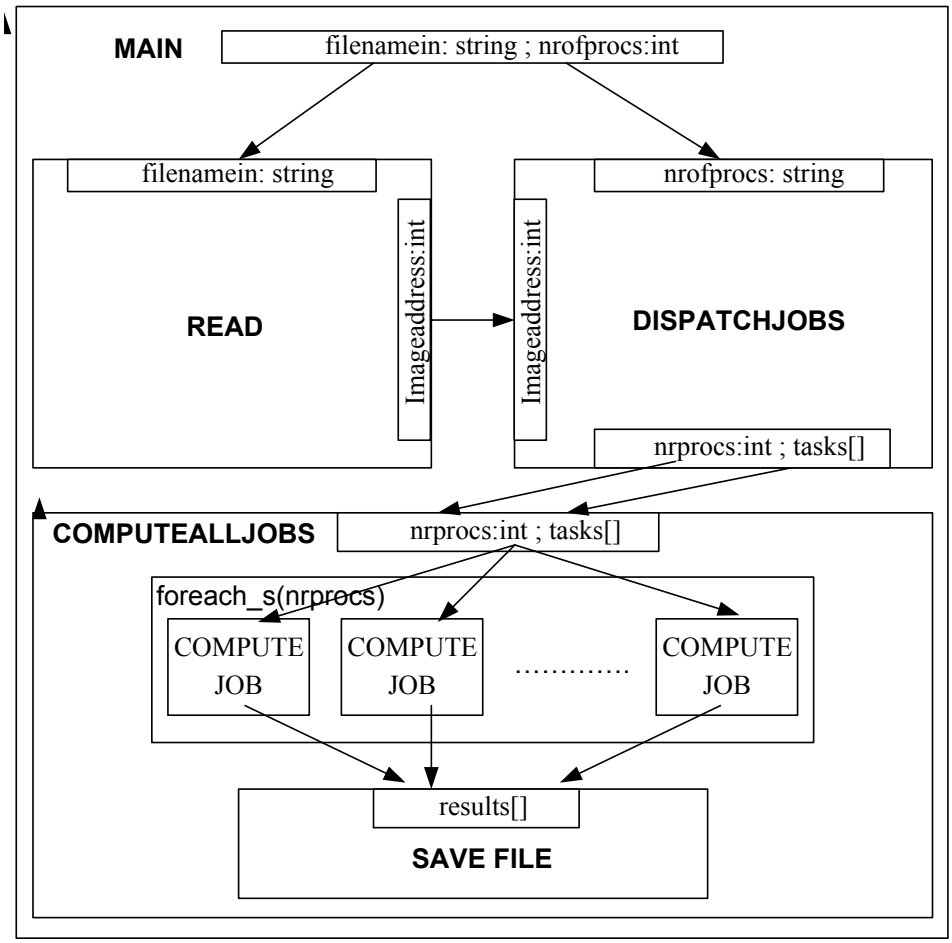

Figure 3.1: High level view of modules and input/output in the edge detection application.

is to split the computation of the input image using line blocks and fill an array of inputs for different processes with tasks informations - each element of the array corresponds to a line block.

```
module DISPATCHJOBS{listen imageaddress:int}
{read nrofprocs:int}@MASTER
{
  nrprocs = nrofprocs;

  // Take and recover the header
  TGAReader::STGA* pTGAFile=(TGAReader::STGA*)imageaddress;
  const int pixelBytes = pTGAFile->byteCount;
  const int imageLineBytes = pixelBytes * pTGAFile->width;

  // Set up the tasks
  int iLinesPerProc = pTGAFile->height / nrofprocs;
  int iLinesToAddOne = pTGAFile->height % nrofprocs;
  int iLineIndex = 0;
  for (int i = 0; i < nrofprocs; i++)
  {
    int linesToThisProc=iLinesPerProc+(iLinesToAddOne>i?1:0);

    // Every index with the exception of the first one, and last one
    const int nBorders = 2-1*(i ==0||i ==(nrofprocs-1));
    unsigned int iDataSize=imageLineBytes*(linesToThisProc+nBorders));
    // Every index with the exception of first, gets a previous line
    const int off = imageLineBytes*(iLineIndex-1*(i>0));
    unsigned char* pDataBegin = (unsigned char*) &pTGAFile->data[off];

     // For the first row we would like to not modify the
     // first row of pixels
     int lineStart = iLineIndex + 1*(i == 0);
     int numRows=linesToThisProc-1*(i==0||i==(nrofprocs-1))

     tasks@[i].objects = <pDataBegin, iDataSize>;
     tasks@[i].rowstart = lineStart;
     tasks@[i].numrows = numRows;
     tasks@[i].colstart = 0;
     tasks@[i].width = pTGAFile->width;
```

```
      iLineIndex += linesToThisProc;
    }
  }{speak nil}{write nrprocs:int;((objects:buffer,rowstart:int,
                numrows:int,colstart:int,width:int )tasks[];)}
```

The purpose of COMPUTEALLJOBS is to read the array with tasks, then instantiate a number of task modules (COMPUTEJOB) equal to the number of available hardware processes given as input. Finally, it uses a module (SAVEFILE) to gather the partial results and save the final output.

```
module COMPUTEALLJOBS{listen nil}
{read nrprocs:int;
 ((objects:buffer,r:int,height:int,c:int,width:int)tasks[];)}
{
  // I (identity module was used here to consume the nrprocs
  // input from north interface
  I # foreach_s (nrprocs)
  {
    COMPUTEJOB
  }
  %
  SAVEFILE
}{ speak nil } { write nil }
```

The COMPUTEJOB module will receive a task, call the library function to compute the gradient in its own part of the image, then sends the output further. As the interfaces show, there is no dependency between horizontal composition of COMPUTEJOB modules (both listen and speak interfaces are nil). This means that all the instances created by COMPUTEALLJOBS using foreach_s(nrprocs){ COMPUTEJOB } can run in parallel.

```
module COMPUTEJOB{listen nil}{read imagebuffer:buffer,
         linestart:int,height:int,colstart:int,width:int}
{
   SobelLib::ComputeSobel(imagebuffer,linestart,height,
                          colstart,width,imageoutbuffer);
   outwidth  = width;
   outheight = height;
   outbpp    = 3;  // hardcoded
}{speak nil}{write imageoutbuffer:buffer,outwidth:int,
   outheight:int,outbpp:int}
```

The SAVEFILE module automatically waits for all elements of the array to be filled - acts like a gather because it is an atomic module and, as stated in Chapter 2, needs all input available before being ready for execution. The next step is to reconstruct the output image using the results and save it on disk.

```
module SAVEFILE { listen nil }
{read((imageoutbuffer:buffer,width:int,height:int,
 bytesperpixel:int)results[];)}
{
 int nrprocs = GetNumItemsInVectorProcessItem(results);

 // Test the results validity and get the total width,
 // height and bytes perpixel of the header
 int totalHeight = 0;
 int width = 0;
 int bpp = 0;
 for (int i = 0; i < nrprocs; i++)
 {
   SimpleProcessItem* pSBPP = results@[i];
   bpp = ((IntDataItem*)pSBPP->GetItem(3))->GetValue();
   width = ((IntDataItem*)pSBPP->GetItem(1))->GetValue();
   totalHeight += ((IntDataItem*)pSBPP->GetItem(2))->GetValue();
 }

 // Set up and save the header
 TGAReader::STGA outHeader;
 outHeader.width = width;
 outHeader.height = totalHeight;
 outHeader.byteCount = bpp;

 FILE *file = fopen("output.tga", "wb");
 TGAReader::saveTGAHeader(file, outHeader);

 // Save file parts
 int iBytesPerLine = outHeader.byteCount * outHeader.width;
 for (int i = 0; i < nrprocs; i++)
 {
   SimpleProcessItem* pResult = results@[i];
   BufferDataItem* pDataItem = (BufferDataItem*) pResult->GetItem(0);
```

```
   int iBeginOffset=iBytesPerLine*(i>0);
   int iReceivedLines=pDataItem->m_iBufferSize/iBytesPerLine;
   const int linesReceived = (iReceivedLines-1-1*(i>0&&i<nrprocs-1));
   unsigned int iSize=iBytesPerLine*linesReceived;
   fwrite(pDataItem->m_pData+iBeginOffset,
          sizeof(pDataItem->m_pData[0]), iSize,file);
 }
 fclose(file);
}{ speak nil} { write nil}
```

Modules `SAVEFILE`, `COMPUTEJOB` and `READ` are atomic because they do not contain any AGAPIA statement or composition operator (even if it contains operator "`@`" for accessing arrays of processes, it still remains atomic because "`@`" and some other operators can be translated in C/C++ code). As we will see in the sequel, their entire code is built and executed as C/C++ code.

After writing the code, users have to write a definition file containing include files, libraries, additional code files, libs/dll for linker and additional include/library directories. One exemple of definition file for our example is:

```
% // Include files
#include <stdio.h>
#include <math.h>
#include "TGAReader.h"
% // Additional include directories
% // Additional library directories
% // Additional linker libs/dll etc
Sobel.lib
% // Additional source files
TGAReader.cpp
```

Finally, users need to copy and modify an "`execution.bat`" file from distribution's example to set and set exactly in this order the following things: the type of execution (serial or distributed), definition file, code file, input for main module and all other files used by the programs which are needed for proper execution. Below is the content of this file for our example. We need to include the `TGAReader` source and header and the input image, "`images.tga`".

```
set GENPATH=%AGAPIAPATH%\GenerateApp\Release\GenerateApp.exe
%GENPATH% exectype=distributed Def.txt agapia.txt MainInput.txt
TGAReader.h TGAReader.cpp images.tga
```

The `MainInput.txt` file contains the input for the `MAIN` module and must contain all the variables in its interface. In our example it can be something like this:

```
filenamein images.tga
nrofprocs 3
```

After executing the `.bat` file an executable of the program that can run independently of the installed package and configuration will be generated in the local folder.

3.2 Compilation process

3.2.1 Global view

The current version of AGAPIA package can be found at
`https://code.google.com/p/agapia-programming-language/` [101]. Informations about how to install and a collection of toy examples can be found there. Figure 3.2 shows the process of building an AGAPIA program from the user's source code and the intermediary tools used in this process. The actions are marked with rectangles, while input/output for actions are represented as ellipses.

There are four main tools in the distribution:

- `Compiler` - This contains C/C++ template projects files for different execution types and source files for parsing and runtime.
- `ModulesAnalyzer` - It is used to separate atomic modules from the non-atomic ones and to translate as much as possible from AGAPIA code to C/C++.
- `SolutionGeneration` - It is used to modify a template project file for Compiler and to include user's input parameters like additional files, directories, etc.
- `Generate` - It contains the main entry for compiling a program, coordinates `Compiler` and `ModulesAnalyzer` tools for obtaining an exectutable.

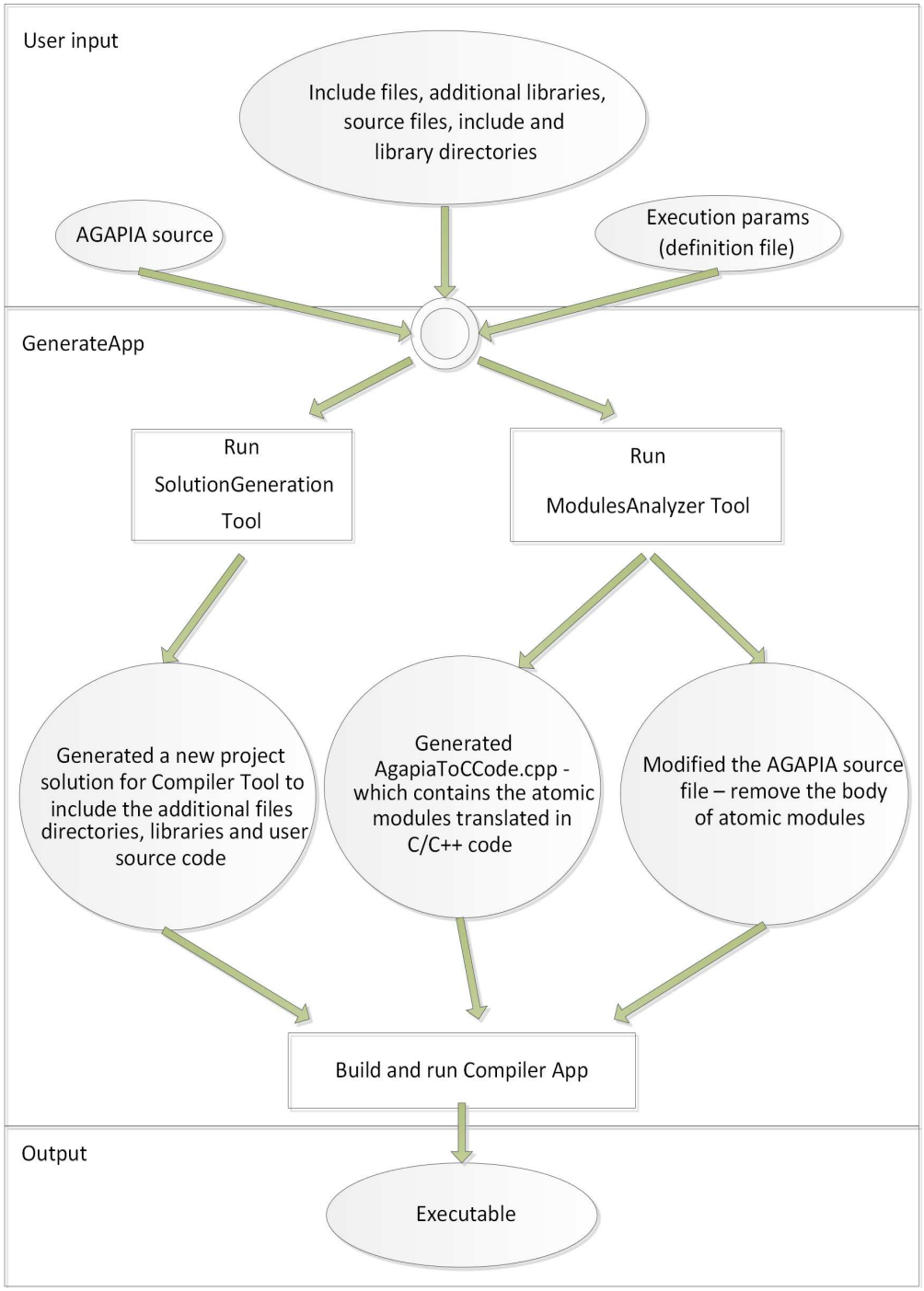

Figure 3.2: High level steps of the compilation process in AGAPIA

The `Generate` tool is the main entry of the compilation process and it is responsible for coordinating the other tools to achieve an executable from the user's source code and execution parameters. There are three actions performed by this tool:

1. The first one is done inside `SolutionGeneration` tool which considers as input the execution parameters, all additional source files, libraries and include/library directories specified by the user. The `Compiler` tool contains several templates of project configurations depending on the execution type: distributed, iterative, or shared memory. After reading the execution parameters one of these templates is chosen and modified to add all other user's requests. This way, the `Compiler` tool can be built by including user source files and libraries.

2. The second action - which can be done in parallel with the first one - is to run the `ModulesAnalyzer` to separate the code for atomic modules and include them in a CPP source file.

3. The last action - which needs the outputs from the previous two - is to build the `Compiler` using the new generated project configuration and the source file which includes the code of atomic modules. The output of this action is an executable which can run independently of all these tools.

The rest of this section presents these tools and their implementation in low-level details.

3.2.2 ModulesAnalyzer

This tool takes as input the AGAPIA source file given by the user and has two main steps:

- Parse and convert as much as possible from AGAPIA code to C/C++, identify which modules are atomic, and modify the original source file.

- Generate `AgapiaToCCode.cpp` file which contains the C/C++ code of all atomic modules.

To improve the performance it is mandatory that as much as possible from the original AGAPIA code to be converted and built as C/C++ code.

- To accomplish this, the first step of the tool is to analyse the code line by line and convert everything that is possible from AGAPIA to C/C++ code. Currently this is applied to vector of processes and data buffer types. Expressions such as `array@[i]` are translated in a C/C++ function call that implements this operation. Same strategy is used for accessing data buffers types.

- The next step is to check all modules using a finite state machine and create a new translated source file. If a module is atomic - its code doesn't contain any AGAPIA specific statement or composition operator that can't be translated in C/C++ code - then its entire source code is added in a file named `AgapiaToCCode.cpp` as a function (the next section will explain this process in more details) while in the new translated source code file the body of the atomic modules will be represented as a "`@`". An example is module `SAVEFILE` from the previous example which is atomic. Instead of the original module source code as defined in "`agapia.txt`" file, the new translated source code file will contain just:

```
module SAVEFILE { listen nil }
{read((imageoutbuffer:buffer,width:int,
height:int,bytesperpixel:int)results[];)}
{
@
}{ speak nil} { write nil}
```

- This strategy is applied to all other atomic modules: COMPUTEJOB, DISPATCHJOBS, READ. If a module is not atomic, then it is copied in the new translated source code as it was written in the original code.

At the end of `ModulesAnalyser` execution, we have a file containing all the atomic modules converted as C/C++ functions and a new AGAPIA source code file which substituted with a "`@`" all the bodies of the atomic modules.

3.2.3 Compiler

This subsection explains how the `Compiler` tool works. Figure 3.3 shows a high level overview of what is happening inside this tool. In the first layer there is a component to parse the code. When a grammar rule is recognized, a corresponding entity is instantiated using an internal API and then added

to the abstract syntax tree (AST). The construction of the AST is finalized after the process of parsing the AGAPIA source file is finished. In this tree, as the next sub-sections will present, there are two types of nodes: nodes that represents modules and nodes that represents input/output processes. The links between these nodes represents the flow of the input/output in the program.

An important thing to understand is that the AST is not transformed in binary code. As the next section shows in more details, the AST is used to do the input/output flow and detect which modules are ready for execution. The current version of AGAPIA compiler saves this tree in a binary file at the compilation process ("`agapia.dat`" file). Before each execution of the program this tree is loaded in memory. The `AgapiaToCCode.cpp` file containing the code for atomic modules is included as binary code in the executable along with the **AGAPIA runtime** component (Figures 3.3, 3.5) which is capable of doing the input/output flow and execution of the modules. The final executable is built using the previously generated project configuration.

Lexical analyser

Flex [54] is used to build a lexical analyser which will be used later in the parsing process. It receives as input a file with all the tokens that should be recognized from the source code. The specification supports most of the C/C++ language constructs plus typical AGAPIA statements.

Parser generator and creation of abstract syntax tree

The parsing process is done using Bison [54]. It receives as input a specification of a context-free language for AGAPIA and generates a parser in C++ which reads sequences of tokens (generated by Flex) from the input file and decides whether the sequence conforms to the syntax specified by the grammar.

Each time when a grammar rule is matched the code uses an abstract factory object [29] (ABSTFactory) to add/modify nodes in the syntax tree. The following code snippets presents how some of the rules are modifying the syntax tree.

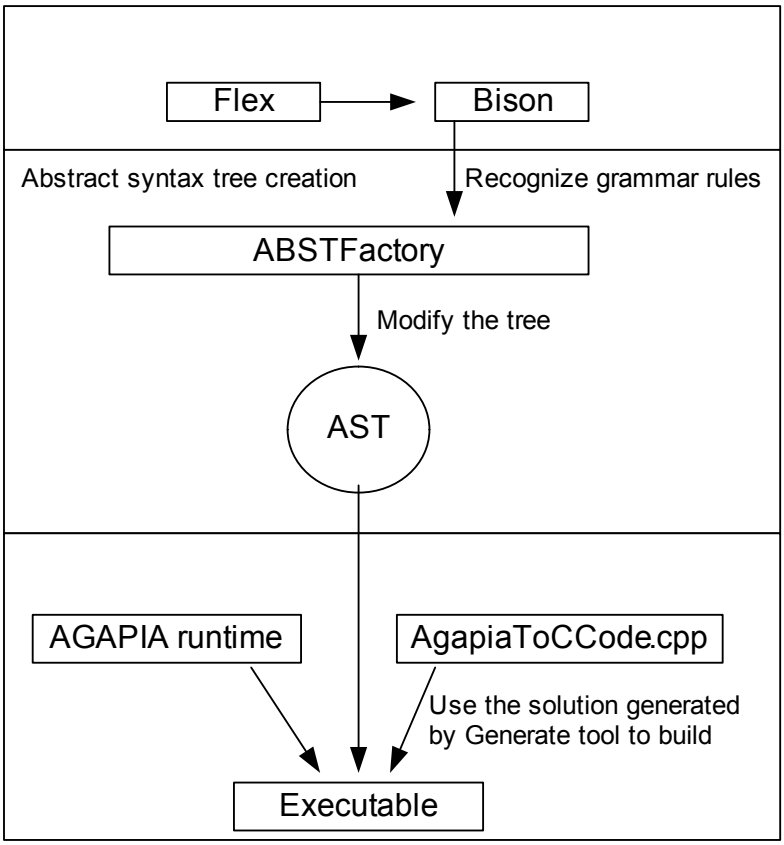

Figure 3.3: Inside the `Compiler` tool

Data types and module's interfaces

Each input/output process represents a node in the abstract syntax tree. Any of the four input/output sides can contain zero, one or multiple such nodes (organized internally in a linked list). The four sides are instance of "`InputBlock`" and recognized by "`processinputlist`" rule. The input processes (recognized by rule "`processinputlist`") can contain a list of "`simplevariablelist`" (eg. "`a:int, b:char, c:buffer`") - instances of `SimpleProcessItem` - or a vector of processes ("`simplevariablelist[]`") - instance of `VectorProcessItem`. A module input is recognized by "`module_input`", while the output by "`module_output`". A module is created when the "`module`" rule is recognized. Using the ABSTFactory component, compiler creates various types of modules and data types.

```
variable:
   IDENTIFIER_LOW ':' datatype
     { $$ = ItemTypeFactory::CreateInputItem($3, $1); }
 | IDENTIFIER_LOW ':' datatype '[' ']'
     { $$ = ItemTypeFactory::CreateInputItem($3, $1); }
;

simplevariablelist:
   simplevariablelist ',' variable
   { SimpleProcessItem* pSPI = (SimpleProcessItem*) $1;
     pSPI->AddItem((IDataTypeItem*)$3); $$ = pSPI;
   }
 | variable
   { SimpleProcessItem* pSPI = new SimpleProcessItem();
     pSPI->AddItem((IDataTypeItem*)$1); $$ = pSPI;
   }
   ;

singleproceslist:
    simplevariablelist { $$ = $1; }
   |'(''(''processinputlist ')'IDENTIFIER_LOW '[']'';'')'
    { VectorProcessItem* pVPI=new VectorProcessItem();
      pVPI->SetItemType(((InputBlock*)$3)->m_InputsInBlock, $5);
      $$ = pVPI;
    }
    ;
```

```
processinputlist:
processinputlist ';' singleproceslist
    {  InputBlock* pIB = (InputBlock*) $1;
       pIB->AddInput((BaseProcessInput*)$3);
               $$ = pIB;
    }
| singleproceslist
    { InputBlock* pIB = new InputBlock();
      pIB->AddInput((BaseProcessInput*)$1);
      $$ = pIB;
           }
|  NIL
            { InputBlock* pIB = new InputBlock();
             $$ = pIB;
            }
;

module_input:
   '{' LISTEN processinputlist '}' '{' READ processinputlist '}'
    {
   $$ = ABSTFactory::CreateInputBlocks($3, $7);
   InputBlock* pIB = (InputBlock*) $3;
   pIB = (InputBlock*) $7;
    }
    ;

module_output:
   '{' SPEAK processinputlist '}' '{' WRITE processinputlist '}'
    {
$$ = ABSTFactory::CreateOutputBlocks($3, $7);
InputBlock* pIB = (InputBlock*) $3;
pIB = (InputBlock*) $7;
    }
    ;

module:
    MODULE IDENTIFIER_BIG module_input'{ module_body'}'module_output
    { ABSTFactory::CreateAgapiaModule($2, $3, $5, $7); }
    ;
```

Programs, compositions and high-level statements

The module body can be either an atomic program - created with the following function `ABSTFactory::CreateCCodeProgramType` - or a non-atomic one - recognized by "`program`" rule and created depending on its type with `ABSTFactory` functions. The following code snippets show examples of `FOREACH` and `IF` statements which are created using `ABSTFactory::CreateFOREACHProgram` and `ABSTFactory::CreateIFProgram`. A program resulted by composition of two programs is created using `ABSTFactory::CreateIntermediateProgram`.

```
module_body:
program { $$ = $1; }
| '@' EXEC_TARGET_MASTER
   { $$ = ABSTFactory::CreateCCodeProgramType(E_CCODE_FORMASTER); }
| '@'
   { $$ = ABSTFactory::CreateCCodeProgramType(E_CCODE_FORALL); }
|  NIL
   { $$ = ABSTFactory::CreateIdentityProgram(); }
;

program:
   IDENTIFIER_BIG
   { $$ = ABSTFactory::CreateModuleRef($1); }
  | asignment
   { $$ = $1; }
  | variable
   { $$ = ABSTFactory::CreateDeclarationProgram($1); }
  | program COMPOSITION_HORIZONTAL program
   { $$ = ABSTFactory::CreateIntermediateProgram($1,$3,E_COMP_HORIZONTAL); }
  | program COMPOSITION_VERTICAL program
   { $$ = ABSTFactory::CreateIntermediateProgram($1,$3,E_COMP_VERTICAL); }
  | program COMPOSITION_DIAGONAL program
   { $$ = ABSTFactory::CreateIntermediateProgram($1,$3,E_COMP_DIAGONAL); }
  | '(' program ')'
   { $$ = $2; }
  | IF '(' boolean_expr ')' '{' program '}' %prec IFX {}
   { $$ = ABSTFactory::CreateIFProgram($3, $6, NULL); }
  | IF '(' boolean_expr ')' '{' program '}' ELSE '{' program '}' { }
   { $$ = ABSTFactory::CreateIFProgram($3, $6, $10); }
  | WHILE_S '(' boolean_expr ')' '{' program '}'
```

```
   { $$ = ABSTFactory::CreateWHILEProgram(DTYPE_WHILE_S,$3,$6);}
  | WHILE_ST '(' boolean_expr ')' '{' program '}'
   { $$ = ABSTFactory::CreateWHILEProgram(DTYPE_WHILE_ST,$3,$6);}
  | WHILE_T '(' boolean_expr ')' '{' program '}'
   { $$ = ABSTFactory::CreateWHILEProgram(DTYPE_WHILE_T,$3,$6);}
  | FOREACH_S '(' expr ')' '{' program '}'
   { $$ = ABSTFactory::CreateFOREACHProgram(DTYPE_FOREACH_S,$3,$6); }
  | FOREACH_T '(' expr ')' '{' program '}'
   { $$ = ABSTFactory::CreateFOREACHProgram(DTYPE_FOREACH_T,$3,$6); }
  | FOREACH_ST '(' expr ')' '{' program '}'
   { $$ = ABSTFactory::CreateFOREACHProgram(DTYPE_FOREACH_ST,$3,$6); }
  | FORNORMAL_ST '(' asignment ';' expr ';' expr ')' '{' program '}'
   { $$ = ABSTFactory::CreateFORNormalProgram(DTYPE_FORNORMAL_ST,$3,$5,$7,$10);}
  | FORNORMAL_S '(' asignment ';' expr ';' expr ')' '{' program '}'
   { $$ = ABSTFactory::CreateFORNormalProgram(DTYPE_FORNORMAL_S,$3,$5,$7,$10);}
  | FORNORMAL_T '(' asignment ';' expr ';' expr ')' '{' program '}'
   { $$ = ABSTFactory::CreateFORNormalProgram(DTYPE_FORNORMAL_T,$3,$5,$7,$10);}
  |  { }
;
```

Linking nodes through composition and aggregation

Figure 3.4 contains a low-level representation of nodes and links in the running problem (discussed in the first section of this chapter). The programs with dotted lines represents the programs created by composition, while the other ones are defined by user. The input/output nodes are represented by rectangles which are present on the four cardinal sides.

A composition of programs results in another program with unified input/output nodes, in the order of composition - see `READ # DISPATCHJOBS` and `(READ# DISPATCHJOBS) % SAVEFILE` modules. AGAPIA's backend will automatically try to match the input/output nodes at composition (according to the rules presented in the Chapter 2) and aggregation. For example, the input from the north side of the `MAIN` module goes to `(READ # DISPATCHJOBS) % SAVEFILE` module, then to `(READ # DISPATCHJOBS)` and finally to `READ` and `DISPATCHJOBS`, which are atomic modules. If the parenthesis don't exist, they will be automatically added in the background for the operands of any composition defined by a composition operator. The target is to make all compositions defined by one of the three operators, a binary one. This way the compiler can easily try linking the input/output nodes. If

the composition can not be done because the interfaces needed to be linked do not match, then a compilation error occurs.

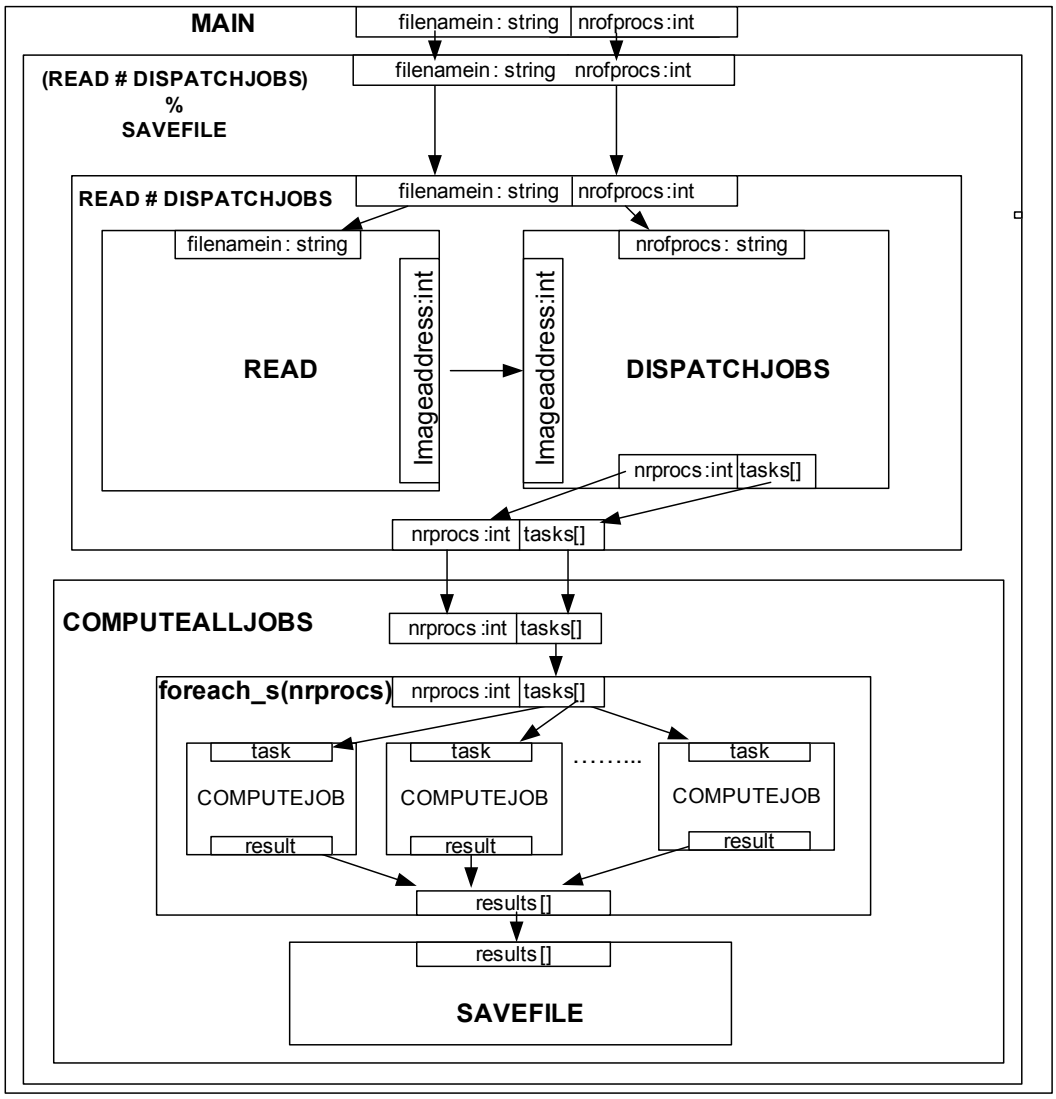

Figure 3.4: Internal representation of an image processing problem in AGAPIA

The `if`, `foreach`, `for`, and `while` statements

The previous chapter presented the semantics of all these nodes. As explained earlier, the input/output nodes of a `foreach` program are defined by the type of composition inside program (`foreach_s`, `foreach_t`, or `foreach_st`) and by the type of the internal program. In a `foreach` node, the internal programs are instantiated after the expression given as parameter can be evaluated. At that point the links between the input/output nodes of the `foreach` node and each internal program instance are created. In the running example, when the `nrprocs` variable will be received by the `COMPUTEALLJOBS`, the `foreach_s` node can instantiate all the internal programs and link its input/output nodes with the input/output nodes of the internal programs. Same strategy is applied inside `while` and `for` nodes with the exception that for these two, usually the internal programs are created one by one after checking the condition.

In the case of an `if` node, the input/output nodes are the same as the internal programs (the programs which defines the two branches must also coincide in terms of interfaces). When the condition can be evaluated, the correct branch program will be instantiated and have its input/output nodes connected with the parent `if` input/output nodes.

How is the code of atomic modules included as binary code in the executable

As the previous sections stated, the entire code of an atomic module will be, at the end of the compilation process, compiled and linked as C/C++ code. All the source code of atomic modules will be separated in a file named `AgapiaToCCode.cpp`, while the original input file containing AGAPIA code will contain inside the atomic module's body a "`@`". An example is the `READ` module. AGAPIA code file will contain:

```
module READ{listen nil}{read filenamein:string}
{
@MASTER
}{speak imageaddress:int}{write nil}
```

The body code of the `READ` module will be converted to a C/C++ function which receives as parameters pointers to the instances of the four input/output sides. The trick is how can compiler use the same variables (names)

defined inside the module's interfaces, in the C/C++ code. The solution for this is to automatically generate code that declares a local variable for each of the variable defined in a module's interfaces. These variables will be actually references to the correct data in the input/output sides. Below it is presented the entire code for the `READ` module inside `AgapiaToCCode.cpp` file after adding the auto-generated code:

```
void READ(InputBlock* pNorth,InputBlock* pWest,
  InputBlock* pSouth, InputBlock* pEast)
{
   // Local variables declaration:
   char** filenamein = ((StringDataItem*)((SimpleProcessItem*)
pNorth->m_InputsInBlock[0])->m_InputItems[0])->GetValueRef();
   int& imageaddress = ((IntDataItem*)((SimpleProcessItem*)
pEast->m_InputsInBlock[0])->m_InputItems[0])->GetValueRef();

   // User code:
   TGAReader::STGA* tgaFile = new TGAReader::STGA();
   TGAReader::loadTGA(*filenamein, *tgaFile);
   printf("File received to read from: %s\n", *filenamein);
   imageaddress = (int) tgaFile;
}
```

The `N`-th input process in the interface defined by `InputBlock` is accessible by using `InputBlock::m_InputsInBlock[N]`. (1) If this input process is not an array of processes then to access the `M`-th variable inside the `N`-th input process the code generates `((SimpleProcessItem*)InputBlock::m_InputsInBlock[N])->m_InputItems[M]`. (2) If the process is a vector of processes then to accesses the `M`-th index the code generates `(VectorProcessItem*)InputBlock::m_InputsInBlock[M]`.

`AgapiaToCCode` will be compiled and linked when building the custom compiler project solution to generate the final executable of the AGAPIA program. This is how the atomic modules code will get inside executable.

3.3 Execution model

This section discusses how the modules are executed and deployed, the input/output flow process, and some examples and optimizations tips to obtain

a peek performance. Also, some basic informations about scheduling are presented.

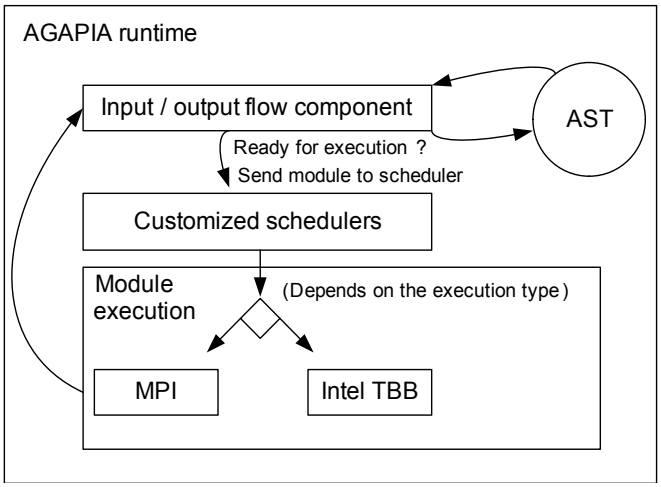

Figure 3.5: Inside AGAPIA's runtime

Figure 3.5 shows how the AGAPIA's runtime process looks and what high-level operations are done inside. As mentioned in the previous section, the AST needs to be loaded in memory each time a program is executed. The AST will communicate with the input/output flow component each time there is input or output available that can be moved through nodes and links. For performance reasons, it is very important that input/output flow component to run on a high-priority dedicated thread. Having the input/output flow process running frequently will maximize the potential parallelism. When an atomic module is ready for execution it sends its input parameters to the active scheduler. Currently, there is a single scheduler available, but users can write their own schedulers (by adding classes in the `Compiler` tool solution). These schedulers can have access to MPI and Intel TBB for distributed communication and shared memory parallelism, respectively. After a module execution is finished it sends back the output to the input/output flow component. In the distributed execution type the default platform where we execute AGAPIA programs is a centralized topology with a master which is supposed to be able to communicate directly with a given number of workers. There are other types of topologies that are discussed in the next chapter as future work for the next versions of AGAPIA compiler.

3.3.1 Input/output flow and examples

The input/output flow process usually starts by feeding the input nodes of the MAIN module with the input given by the user. If MAIN module doesn't need any input, then, as the next subsection will explain, the process starts from the output nodes of the modules that initially don't have any input dependencies and are ready for execution from the start of the program. In the current version, the coordination of input/output flow is done entirely by the master while the atomic modules can be executed either by master or workers.

When an input node is being feed the information flows through the input/output links until the end of the chain.

An example is how the input received by the `MAIN` module of the application in Figure 3.4 flows to the atomic modules. First, `filenamein` and `nrofprocs` are received by the MAIN module, then they go the input nodes in the internal instantiated program `(READ # DISPATCHJOBS)` and from here they follow the input links to the atomic modules: **filenamein** goes to input node of `READ` while `nrofprocs` to `DISPATCHJOBS`.

Another example including output nodes can be considered after execution of `DISPATCHJOBS` module. The output nodes `nrprocs` and `tasks` flow the information to the output nodes of `(READ # DISPATCHJOBS)` and from here to the input nodes of `COMPUTEALLJOBS`. From here, the input flows to the input node of `foreach_s` program. This is a the end of chain for `nrprocs` which is used just as a variable inside the parent program. But `tasks`, which is an array of processes, is automatically split into individual components and sent to the `COMPUTEJOB` instances one by one. This is actually an operation known as Scatter. The example also has a Gather operation when the results are automatically collected from the output of `COMPUTEJOB` instances to the input node of `foreach_s` program, which is as expected an array of processes.

To obtain a peek performance we must avoid unnecessary copies of data. When the input/output data is flown through the nodes it doesn't do any unnecessary copies. In the default model, because the master machine coordinates the input/output flow, the only moment when copies are needed is when an atomic module needs to be solved on another machine.

3.3.2 Modules execution and deployment

An atomic module is ready for execution when all its input nodes had received their necessary data. At this point the internal AGAPIA scheduler will place the new module name and a data buffer containing all its inputs on a queue. It is important to understand how to optimize the performance using deployment macros. There is one type of macro for deployment in the current AGAPIA version and other three are considering (work in progress). There are also plans to add deployment macros such that a program, atomic module or not, can stick its execution and coordination to a given machine.

- `@MASTER` - used to execute an atomic module on the master process.
  An effective example is in the running application where `READ` and `DISPATCHJOBS` modules had this specifier such that both were executed on the `MASTER` process. The reason is that the input image is on the master machine and executing these two modules on another machine would make unnecessary copies of the original image when there is no parallelism to achieve from this.

- `@SAME` (work in progress) - used to execute a group of atomic modules on the same machine for the same reason: to reduce the number of data copies.
  Imagine that we have three modules composed which are sending big chunks of data between them. If we execute these modules on different machines, then the communication time to send data between modules might be too high. For this reason we can force these three modules to execute on the same machine using this macro.

- `@DIFF` (work in progress) - this can be applied to non-atomic modules (programs) and the purpose is to move the responsibility of the coordination inside a module on a different machine other than master.
  Using this we can make sure that the group of modules (programs) using this are running on different machines. It is usefully to implement web-services or pipeline applications where the execution of a component must stick to a specified machines.

- `@GPU` (work in progress) - this can be used to execute a module on a GPU, if there exists one in the system.

3.4 Conclusion

The first part of the chapter contained a practical introduction on how to implement a program in AGAPIA: edge detection in a given image using a **Sobel** filter. The implementation had at its core a geometric decomposition pattern presented in the previous chapter. Explanations on how to design, write code for an AGAPIA program and configure build were given. The high-level design and general code writing demonstrates that an implementation in AGAPIA can be more natural than an implementation in MPI for example because it hides the low-level communication part and keeps the code structured closer to the high-level solution.

The compilation process behind AGAPIA was presented as having four tools: `Compiler`, `ModulesAnalyzer`, `SolutionGeneration` and `Generate`. These are used to take the user's input files and produce an executable which can run with the given specifications. User defined modules where classified in two classes: atomic modules (can be translated in C/C++ code) and non-atomic modules (contains at least one AGAPIA statement which cannot be translated in C/C++ code).
An important aspect that minimizes the performance loss is that the code of atomic modules is included as binary code in the final executable which means that calling an atomic module will have the same overhead as a function call in C/C++. Another important optimization is the avoidance of useless data copies: data is copied through references if the modules are executed on the same machine. Also, in the default model the data flow inside non-atomic modules is done entirely on master which minimizes data copies. The last part of the chapter is dedicated to show a few deployment macros. The role of these is to allow user to tune more the performance or consider dedicated machines for handling some defined modules.
There are other things that can be improved in the future. One of them is to generate code that builds AST and include it as binary code in the executable instead of loading the binary file and create the tree in memory before each execution. Another important aspect is pattern observation and developing deployment macros to increase performance and usability.

Chapter 4

Schedulers

This chapter contains a discussion on how the atomic modules are executed, the default scheduler and topology available in the curent version of AGAPIA and some ideas to bring fairness, deadlines, and other topologies such as rings. These extensions can be easily integrated since the source code for AGAPIA compiler and runtime are open source and use a `Strategy pattern` ([29]). An idea for future versions of AGAPIA would be to create a set of these schedulers and allow users to select between them using the configuration file. Having a default scheduler and some others is another step forward for providing a complete solution for parallel programming and together with the transparent communication model and modularity, AGAPIA can make the programmers' job easier.

As stated in the previous chapter, when an atomic module is ready for execution (there is no input dependency remained) it is added to a queue of tasks. An entry in this queue contains the name of the module to be executed and a data buffer containing the serialization of the module's parameters. The default scheduler is a simple one created just to present applications in AGAPIA and to benchmark its overhead. However, the code is open source and users can modify the Compiler tool solution to add more schedulers and make them active. There are also plans to use the X10 ([21]) instead of handling schedulers manually. This way AGAPIA can have a smart work-stealing scheduler for both distributed and shared memory model.

4.1 The default scheduler

In the distributed execution type modules are sent from master to be executed on workers. The topology used for the distributed case is a centralized model where master can communicate with all workers. When an worker is idle it takes one task from the queue and executes. The execution consists on deserializing the parameters of the module and then call the C/C++ function generated in `AgapiaToCCode.cpp` for that module. Communications between master and workers is realized using MPI which is the standard in the industry for message passing systems [35], [36]. After an atomic module is executed its output is serialized and sent back to the master where the coordination of the output data starts: the data is sent through nodes and links until the end of the chain (Figure 4.1).

In the shared memory execution type the same principles applies except that there is no need to serialize the parameters and do multiple copies of input/output. The scheduler for shared memory model is using in background the Intel TBB solution [59].

4.2 Balancing tasks in a distributed network considering average response time and fairness

This subsection is proposing an algorithm [68], [72] that can work for AGAPIA schedulers to execute atomic modules on the default platform where atomic modules can be consider as tasks that need to be solved for different clients. These clients are supposed to have a license type: the average time of response per task and the number of tasks per unit of time. If there isn't enough computing power available then we must ensure fairness between clients.

4.2.1 Assumptions

The proposed online load balancing algorithm functions in a distributed system where incoming requests are all handled by a single component (a workstation, a process, or a thread). The same holds for outgoing results. Still, the nature of this distributed architecture, i.e. whether it is a cluster, a grid,

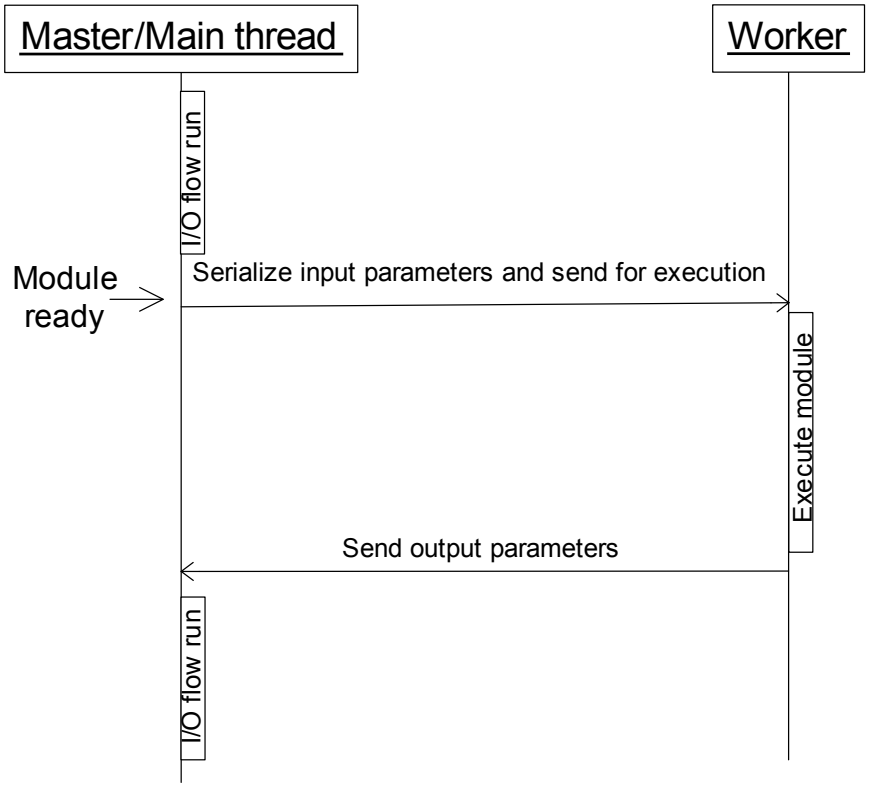

Figure 4.1: Communication between master and workers in the default scheduler.

or any of the previous camouflaged by a cloud, has no bearing on the proper functioning of the algorithm. Also, the algorithm allows for the system to be heterogeneous, workstations may differ in processing capacity (CPU, GPU) and RAM memory.

The processing time of tasks is expressed as the "task's length"[1] and can be predetermined. The processing of tasks is deterministic, or at least an upper bound for the incurred processing time can be precomputed. This is a key restriction of the proposed algorithm, as it allows the master to make accurate estimations about the workload[2] of the workers.

We consider task processing to be workstation independent (all types of tasks can be processed by any of the normally functioning workstations). Tasks are independent (the order in which requests are processed does not affect the correctness of the result). Tasks are indivisible (can only be handled by a single worker at one moment). Also, a worker processes a single task at a certain moment in time (this does not exclude parallel processing of the task). In extreme cases, the algorithm may reject tasks.

The system should allow for a centralized load balancing algorithm to be implemented, i.e. all workers are reachable from the master component and global information can be collected. Even more, the algorithm assumes that direct communication between the master and each worker is allowed; in other words, communication between the master and a worker does not involve intermediates. Assuming hardware or software failures may take place, the algorithm will adapt to a fluctuating number of workers. No other coordination and distribution entity or component is used in corroboration with the load balancing algorithm.

No redistributions of tasks are made as long as the worker is functioning in a normal state. Task redistributions only take place in two situations: 1) the worker where the task is being processed has suffered a failure, and 2) the task has timed out.

For the sake of simplicity in our implementation, we add some assumptions (which may be removed):

- All handled tasks can be classified by length in distinct classes (length intervals), e.g. small, average, big, and very big tasks.

[1] In [98] a load balancing policy is proposed, where the ith workstation handles tasks of length falling in the ith interval.

[2] The workload of a processor is defined as the amount of processing time needed to process all tasks attributed to it [79].

- The communication times between the master (server) and each of the workers in the network differ only by a negligible amount, so we consider them equal.

The worker component should be able to provide the following information: the processing time incurred by a task, structures of the form ⟨length of the task, average processing time⟩[3], available CPU/GPU time, and available RAM memory. As a worker is able to receive only a limited number of tasks at some point in time, a queue-structure holding tasks waiting to be processed should be associated to each worker.

The algorithm is highly dependent on good time synchronization between the master and all the workers. We suggest obtaining this at a software level by periodical synchronization of all workers with the master.

4.2.2 Goals

The following goals are given in the order they are considered by the algorithm.

Goals regarding clients' satisfaction

1. Satisfy the number of requests processed per time interval, according to the user license.

2. Satisfy the average response time for the request, according to the user license.

Goals regarding system's availability, reliability and performance

3. Balance the workload between the processing components.

4. Insure high-availability and reliability of the system by handling in a negligible time occuring resource failures.

5. Maximize the throughput of each processing component.

Goals regarding the algorithm's goodness

6. Insure fairness (with respect to processing time) between requests of the same license type.

[3]We assume that these statistics are not drastically changing in real time.

7. Insure proportionality (with respect to processing time) between requests of different license types.

Goals regarding the optimality of the algorithm

8. Minimize the additional time incurred by the algorithm for data transfer.

9. Minimize the algorithm's processing time.

4.2.3 Master view

As briefly stated before, the algorithm is comprised of two sub modules: a master module and a worker module. The master module (depicted in Fig. 4.2) handles requests' arrival, task distribution among workers, workers' performance monitoring, results collection, and response delivery.

User requests are received (1) by the master through a `Task Receiver` (A), which inserts them (2) in a shared data structure. This shared data structure, called `Local Data`, can be accessed by all components in the master.

While no network overload is detected and `Local Data` contains unprocessed tasks, the `Task Distributor` (B) will keep taking tasks (3), choose the worker for the respective task according to a selection rule (4-5) and send the task to be processed (7). A `Results Receiver` (C) waits for results from the workers (8) and sends them back to the user (9), also announcing other components in the master that the task finished (10-12) by means of shared memory.

`Local Data` should contain the following data:

(a) A list of all undistributed tasks. A task may be inserted in this list either by the `Task Receiver` (when it first arrives in the system) or by the `Priority Monitor` (E) (if it needs to be redistributed, following a timeout or a worker's failure). The exact time when the task has entered the system and the user who issued the request should be stored for every task.

(b) A list of all tasks currently being processed. For each task, the list indicates the user associated to it, the worker where it was sent, and the exact time when it was sent to the worker. A task can be deleted by the `Results Receiver`, when the successful termination of processing

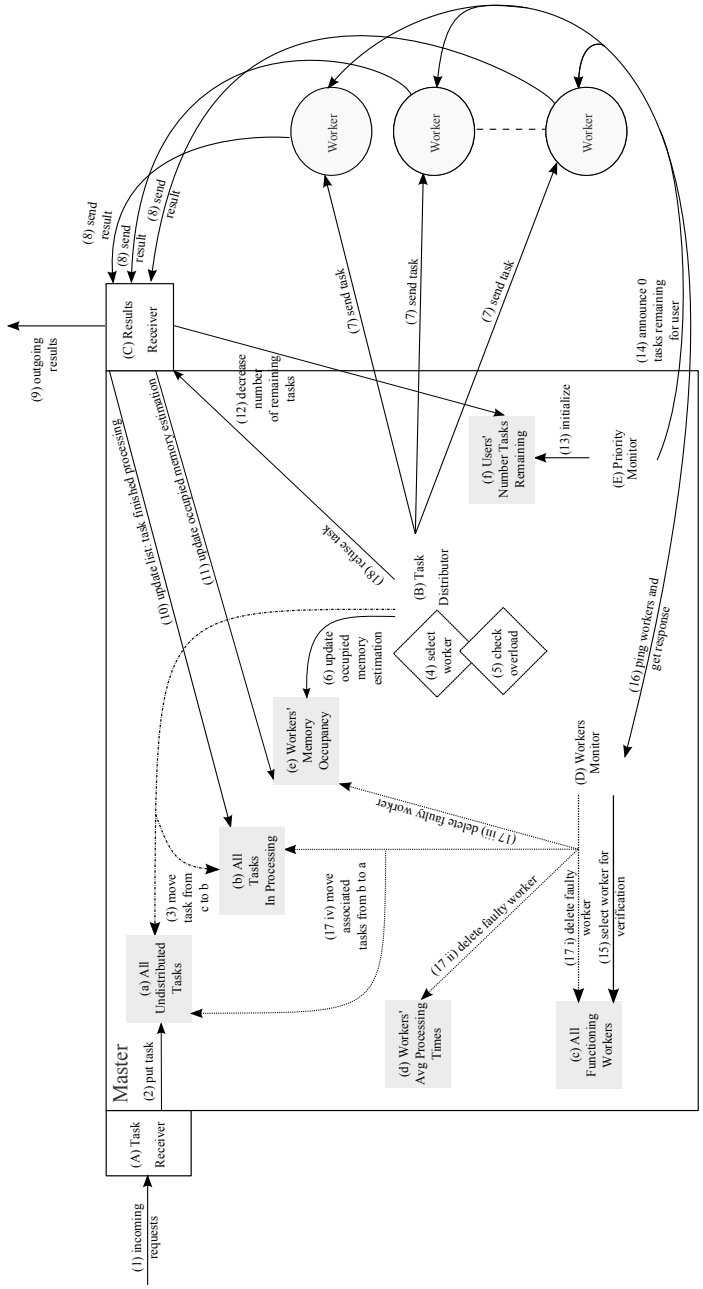

Figure 4.2: Components, data structures and interactions in the master module

is announced. It can also be moved in the list of all undistributed tasks by the `Priority Monitor` when detecting a failure of the worker to whom it was attributed. Statistical data regarding the number of tasks on a worker (for each length class of tasks) should be stored separately in order to increase efficiency. This information will be used to estimate a worker's workload.

(c) A list of all functioning workers, which will be updated if a worker becomes unresponsive.

(d) , Workers' average processing times for each length class of tasks. Remember, a length class of tasks refers to all tasks with length in a certain interval. The algorithm does not depend on real time information about workers' performance, instead it is based on estimations. Therefore, the overhead of collecting status data - the most important problem in the centralized model - should be minimum. As we assume the processing/memory capacity of workers does not change in real time, the algorithm's accuracy is not affected.

(e) Workers' memory occupancy contains values for each worker indicating the current estimated memory occupancy and the safe memory, i.e. the maximum memory of the worker that can be occupied in order to avoid overloading the worker.

(f) A list of the number of tasks remaining for each user in the current time interval of length T. Every T units of time, this list is initialized with the parameter "number of tasks per time interval" specified in the user license. After processing a task for a certain user the value will be decremented by one, reflecting the difference between the number of tasks guaranteed to be processed and the number of tasks already processed.

In the enumeration above, (d) and (e) are seen as input for the algorithm.

The `Task Receiver` component acts as an entry point for all incoming requests, which are placed at arrival in the `All Undistributed Tasks List` (a). The `Task Distributor` represents the master's main component, being responsible for balancing the load between the workers, while the actual task scheduling takes place on each worker.

The `Task Distributor` takes the current task[4] and sends it to the worker that can process it the fastest, i.e. the one that is estimated to finish its current workload first.

The Worker Selection Rule (4) fits the Least Loaded class of task placement policies, as the worker which finishes its load first is selected to process the task [20], [99].

The time when each worker will have processed its current workload is estimated by the product of the number of attributed tasks and the average processing time for each task. Remember that this estimation can be improved by dividing the tasks into more refined classes by their length. The worker for which the above product is minimized will be selected by the algorithm.

If there are n workers in the system, then the algorithm will select worker $1 \leq i \leq n$, given by the following equation.

$$\sum_{p=1}^{m} N_p[i] \cdot A_p[i] = \min_{1 \leq j \leq n} \left(\sum_{p=1}^{m} N_p[j] \cdot A_p[j] \right), \tag{4.1}$$

where:

- $N_p[k]$ is the number of tasks in length class p attributed to worker k, $1 \leq k \leq n$, and
- $A_p[k]$ gives the average processing time of tasks in length class p on worker k, $1 \leq k \leq n$.

This selection rule ensures that the workload will be balanced among the workers, avoiding the exploitation of the worker with the best performance.

The load balancer's reliability is augmented by a preventive measure meant to protect the system from overload situations (5). To this goal, the `Task Distributor` may refuse tasks (18) and the `Results Receiver` will inform the user accordingly. The `Task Distributor` refuses tasks if:

- worker i, chosen by the above selection rule, has a finish time that is greater than T_{max}, where T_{max} is a previously fixed maximum waiting time of tasks.

[4]Tasks are ordered by their entry time in the system.

- the sum between the estimated currently occupied memory of worker i, chosen by the above selection rule, and the memory estimated to be occupied by the new task[5] is greater than the maximum allowed memory of that worker (a value that is established a priori and stored in `Workers' Memory Occupancy List` (e)).

The `Results Receiver` gets the results from the Workers and communicates them to the users. It is also responsible with updating `Local Data`:

- it removes the finished task (10) from the `All Tasks in Processing List` (b),
- it updates the occupied memory estimation of the worker that processed the finished task by subtracting the length of the task (11), and
- it decrements the number of remaining tasks for the appropriate user by one (12).

The `Priority Monitor` focuses on goal 1 (see 4.2.2). Each T units of time, it will initialize (13) the values in `Users' Number of Tasks Remaining List` (f) with the "number of requests per time interval" (of length T) parameter specified in the user license. When the `Priority Monitor` detects that a counter in the list has reached 0, it informs the workers (14), that the respective user has finished his number of tasks in that time interval, thus decreasing the user's priority[6].

The `Workers' Monitor` (D) is responsible for maintaining an accurate overview of the functioning workers. At each time interval of length T_1 (chosen a priori), this component selects each worker for verification (15) and

[5]The memory occupancy of a task is estimated by the task's length, i.e. when a task is sent to a worker, the memory occupancy of that worker is incremented with the task's length, and vice versa when a worker finishes processing a task, that task's length is subtracted from the estimated value. The `Task Distributor` will update the memory occupancy of a worker (6), each time the latter receives a task.

[6]It is possible that precisely when step 14 takes place some workers process requests belonging to that exact user. Once a task processing starts, the task is processed in its entirety, therefore the user may benefit from more handled requests in that time interval than promised. Still, this is a negligible risk, as, in practice, the number of workers is much smaller than the "number of requests per time interval" parameter. A viable solution to this problem involves determining the extra-number of requests handled for the user. This can be done by storing the number of results per time interval for each user and comparing this list of values to the `Users' Number of Tasks Remaining List`. The user may be penalized by allowing him a smaller number of requests the next time interval.

pings it (16). If the response time of the worker is greater than a certain fixed delay time T_{exp}, the `Workers' Monitor` assumes the worker is faulty and deletes it from `Local Data` (17 i-iv):

- It deletes the faulty worker from the following: the `All Functioning Workers List` (c), the `Workers' Average Processing Times List` (d) and the `Workers' Memory Occupancy List` (e).
- It also removes all the tasks that were sent for processing to the faulty worker in the `All Tasks in Processing List`, and inserts them in the `All Undistributed Tasks List`.

4.2.4 Worker view

The worker module (Fig. 4.3) represents the algorithm for each worker component, its main part being local task scheduling. When the master sends a task to a worker (1), the `Task Receiver` (A) places it in a shared data structure (`Local Data`). A `Task Scheduler` (B) establishes the next task (3) to be processed by the `Task Processor` (C). Upon completion of the task processing, the latter puts the result in `Local Data`. A `Results Sender` (D) takes the results and communicates them back to the master.

`Local Data` should contain the following data:

(a) A list of all active tasks. A task is active if it belongs to an user who still has remaining requests in the current time interval. In other words, the user has had less requests served in the current time interval than the "number of requests per time interval" parameter from his user license.

(b) A list of all inactive tasks. A task is inactive if it belongs to an user who has used up all the requests guaranteed to him in the current time interval. Active tasks are considered by the algorithm to have a higher priority than inactive ones.

(c) A list of the number of tasks processed for each user in the current time interval.

(d) The results of the tasks, which have not yet been sent to the master.

When a task sent by the master enters the context of the worker, the `Task Receiver` verifies if it is active or inactive and places it in the appropriate

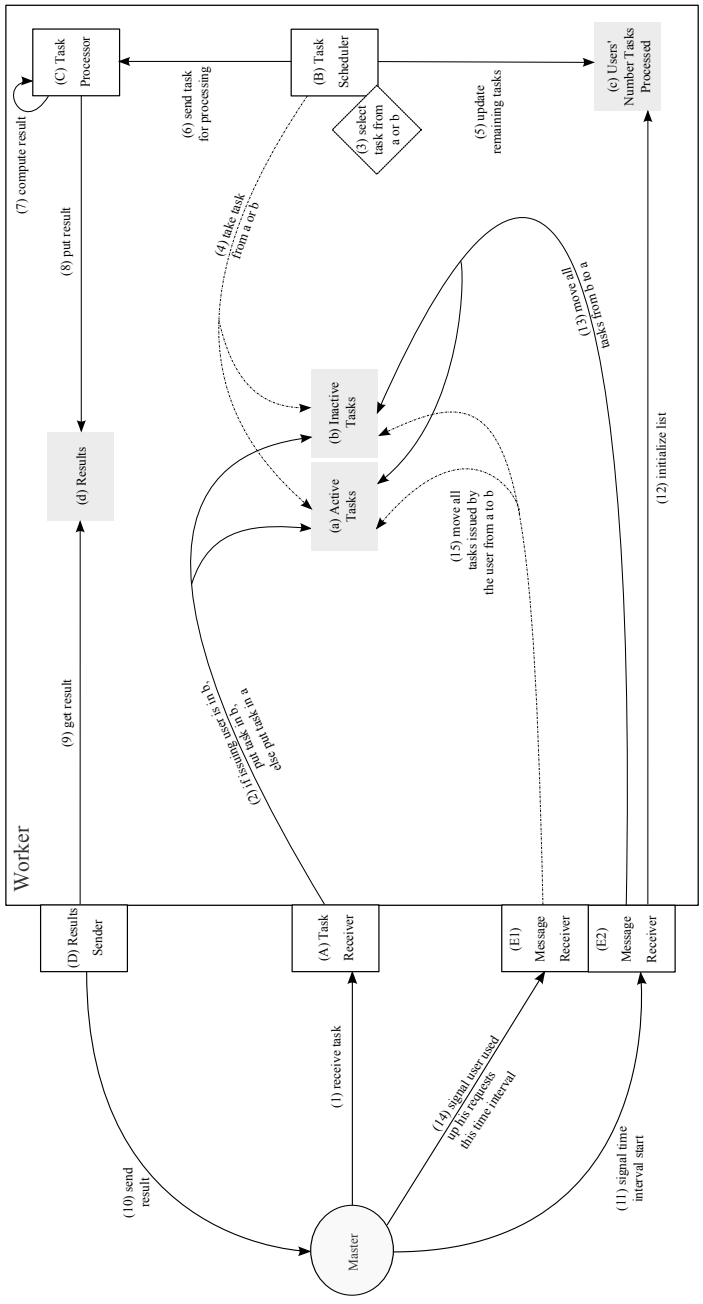

Figure 4.3: Components, data structures and interactions in the worker module

list (2). Specifically, if the corresponding user can be found in the `Inactive Tasks List` (b), then the task is inserted in this list, otherwise it will be inserted in the `Active Tasks List` (a). The `Task Scheduler` uses the algorithm given below to choose the next task for processing either from the `Active Tasks List` or the `Inactive Tasks List`. The `Task Scheduler` removes the selected task from its location (4) and sends it (6) to be processed. It also makes the appropriate update (5) to the `Users' Number of Tasks Processed` (c) incrementing the issuing user's counter by one.

Scheduling algorithm on worker:
$A :=$ `All Active Tasks List`
$B :=$ `All Inactive Tasks List`
$T :=$ length of a time interval (in time units)
$I :=$ current time interval
$t :=$ number of time units passed in I
$\hat{\alpha} :=$ estimated time for processing all tasks $\in A$
$\beta := T - t$ remaining time units in I
if $\hat{\alpha} > \beta$ **then**
  for $\forall$ client $c \in A$ **do**
    $np_c :=$ number of requests issued by client c processed in I
    $\overline{a_c} :=$ normalized value of the "number of requests per time interval" parameter from client c's user license
  end for
  Find $c_{min} \in A$ such that $\frac{np_{c_{min}}}{\overline{a_{c_{min}}}} = \min_{c \in A} \frac{np_c}{\overline{a_c}}$
  Send the oldest task of client c_{min} to be processed.
else
  if $A \neq \emptyset$ **then**
    for $\forall$ client $c \in A$ **do**
      $tp_c :=$ waiting time of client c's tasks
      $\overline{b_c} :=$ normalized value of the "average response time" parameter from client c's user license
    end for
    Find $c_{max} \in A$ such that $\frac{tp_{c_{max}}}{\overline{b_{c_{max}}}} = \max_{c \in A} \frac{tp_c}{\overline{b_c}}$
    Send the oldest task of client c_{max} to be processed.
  else
    if $B \neq \emptyset$ **then**
      for $\forall$ client $c \in B$ **do**

tp_c := waiting time of client c's tasks
$\overline{b_c}$:= normalized value of the "average response time" parameter from client c's user license
end for
Find $c_{max} \in B$ such that $\frac{tp_{c_{max}}}{\overline{b_{c_{max}}}} = \max_{c \in B} \frac{tp_c}{\overline{b_c}}$
Send the oldest task of client c_{max} to be processed.
end if
end if
end if[7].

Let n_1 indicate the number of license types with distinct "number of requests per time interval" parameters, and let $a_1, a_2, \ldots, a_{n_1}$ be the respective values. The normalized value of $a_k,\ 0 \leq k \leq n_1$ is:

$$\overline{a_k} = \frac{a_k}{a_i},\ where\ 0 \leq i \leq n_1,\ a_i = \max_{0 \leq k \leq n_1} a_k \tag{4.2}$$

Similarly, let n_2 indicate the number of license types with distinct "average response time" parameters, and let $b_1, b_2, \ldots, b_{n_1}$ be the respective values. The normalized value of $b_k,\ 0 \leq k \leq n_2$ is:

$$\overline{b_k} = \frac{b_k}{b_j},\ where\ 0 \leq j \leq n_2,\ b_j = \max_{0 \leq k \leq b_1} b_k \tag{4.3}$$

To save time, the normalized values may be computed once, and updated only if the license types change.

The time needed to process all active tasks ($\hat{\alpha}$) is estimated by using counters (for each length class) indicating the number of tasks (with that length class) contained in a list. When multiplying each counter with the average processing time of a task (with the same length class) of the worker in question, the desired estimation is obtained. The "number of requests per time interval" parameter is considered first by the algorithm, as it tries to satisfy goal 1 (see 4.2.2).

After the task has finished processing (7), the `Task Processor` puts the result (8) in the `Results List` (d), from where it is taken (9) by the `Results Sender` and communicated to the master (10).

The `Message Receiver` (E) has two functions. It is informed when a new time interval begins (11), and it is signaled by the master when a user has

[7]Whenever a condition is satisfied by more than one user, the algorithm picks the user randomly from them

exhausted the number of requests attributed to him in that time interval[8] (14). In the first case, the `Message Receiver` (E2) initializes the counters in `Users' Number of Tasks Processed List` according to the values specified in the user licenses (12) and moves all the inactive tasks to the `Active Tasks List` (13). By contrast, in the second case, the `Message Receiver` (E1) will make all the tasks associated to the respective user inactive (15).

4.2.5 A discussion on the algorithm's correctness

The worker selection policy used by the master component is based on a typical workload estimation and does not need further discussion. In this section we will interpret the behavior of the task scheduling algorithm illustrated by a few examples.

The first formula in the algorithm uses the "number of requests per time interval" parameter, being also called the "number of requests per time interval" method. Consider two clients A and B with $a_A = 200$ and $a_B = 400$. The normalized values ($\overline{a_c}$) will be $\overline{a_A} = 0.5$ and $\overline{a_B} = 1$. When all of the users have the same number of tasks processed in the current time interval ($np_A = np_B > 0$)[9], the algorithm will choose to process the oldest task belonging to user B. User B is associated with the higher normalized value of parameter a_c, thus his corresponding fraction $\frac{np_c}{\overline{a_c}}$ will always be less than A's fraction if the given condition stands. In this situation, the algorithm favors the users with greater values of the "number of requests per time interval" parameter, as it should. By contrast, if both users have 50% of their tasks processed ($np_A = 100$ and $np_B = 200$), their fraction becomes equal and the `Task Scheduler` chooses randomly among them. This goes to show that users will be treated fairly and in a balanced way.

Let us assume that the worker we focus on is in an overloaded situation, in the sense that it cannot process all active tasks in the remaining time of the current interval. In this case, the algorithm addresses the "average response time" parameter. Let clients A and B have the associated parameter values $b_A = 2$ and $b_B = 10$, leading to the normalized values ($\overline{b_c}$): $\overline{b_A} = 0.2$ and $\overline{b_B} = 1$. Again, if their waiting time (i.e. the time passed since a task belonging to the user has been selected for processing) is equal, tasks issued

[8] At the beginning of each time interval of length T, the workers must initialize the `Users' Number of Tasks Processed List`. To ensure that all of them are acting in a synchronized manner, a message is passed to them by the master.

[9] If $np_A = np_B = 0$, then the fraction $\frac{np_c}{\overline{a_c}}$ is minimized by any of the users.

by user A are prioritized. This is desired in the stated conditions, since client A possesses a user license with a smaller value of the "average response time" parameter, which leads to a smaller normalized value of it and a higher fraction $\frac{tp_c}{b_c}$.

Table 4.1: Example for the "average response time" method

Step	0	1	2	3	4	5	6	7	8	9	10	11	12
tp_A	0.01	2.01	2	2	2	2	2	4	2	2	2	2	2
tp_B	0.1	2	4	6	8	10	12	2	4	6	8	10	12
$\frac{tp_A}{b_A}$	0.05	**10.05**	**10**	**10**	**10**	**10**	10	**20**	**10**	**10**	**10**	**10**	10
$\frac{tp_B}{b_B}$	**0.1**	2	4	6	8	10	**12**	2	4	6	8	10	**12**

The following example, portrayed in Table 4.1[10], shows how the algorithm tries to satisfy the average response time of users and maintain proportionality between their user licenses. Suppose a worker receives requests from clients A and B with the same specifications as above, the first task arriving from user B, closely followed by one from A. We begin observing the behavior of the `Task Scheduler` from this initial phase, where user A and B have a waiting time of 0.01 seconds and 0.1 seconds, respectively. For simplicity we assume that processing any task takes 2 seconds and no overhead in communication between the worker's components occurs. As the algorithm chooses the user with the maximum value of $\frac{tp_c}{b_c}$, it will send the oldest task of client B for processing, although his expected average time of response is much higher than A's. But during the course of the time interval, the time gained by user B will be averaged towards the value specified in the user license or corresponding to his associated priority. The proportion between $b_A = 2$ and $b_B = 10$ intuitively indicates that for every four or five tasks processed for user A, one should be processed for user B. And this is exactly how the algorithm treats the two users. At steps 5 and 11, where the fractions are equal, and the algorithm selects a user randomly, we have assumed user A has been selected. This being said, after every five requests handled for user A, a request for user B will be served, reaching the desired proportion.

The second part of the algorithm (the "average response time" method) clearly focuses on goals 2 and 6. Because no obvious relationship between the "number of requests per time interval" and the "average response time"

[10]Boxed numbers indicate the maximum values of the fractions. The oldest task of the user associated with these values will be processed next. At each step - represented as a column - a task selection and a task processing take place.

parameters exists, this method overlooks goal 1. Therefore, as a protection measure, a global control strategy is applied through the master's informative announcement to the workers whenever an user has exhausted his requests in a time interval (step 14 in Fig. 4.2). As a consequence of this, the workers will make all tasks belonging to the respective user inactive (step 15 in Fig. 4.3). Inactive tasks will be processed only if all active tasks have been handled.

In this way, when a worker is not overloaded (i.e. it can satisfy goals 1 and 2), the selection of tasks, by both the "average response time" method and the "number of requests per time interval" method, leads to better than expected values for the two parameters. This behaviour can be restricted by putting the worker in a sleep mode until the next time interval, in order to not surpass the guaranteed parameters.

It can easily be seen that between the foremost considered goals (1 and 2) there exist some linear dependencies:

- If goal 1 is not satisfied, then goal 2 cannot be satisfied.
- If goal 2 is satisfied, then goal 1 is also satisfied.

The algorithm will try to satisfy goals 1, 2, 6 and 7 according to their priority. Given that the first two goals depend on the availability and performance of hardware resources, the proposed load balancer cannot guarantee them. Still, goals 6 and 7, regarding fairness and proportionality, can be guaranteed. From this point of view, the load balancer's algorithm has the following logical interpretation:

if goal 1 can be satisfied **then**
  if goal 2 can be satisfied **then**
    Goals 1, 2, 7 (implied by 1 and 2) and 6 are satisfied.
  else
    Goals 1, 7 (implied by 1) and 6 are satisfied.
  end if
else
  Goals 7 and 6 are satisfied.
end if

Simulation results

Two tests are relevant to our algorithm:

1. In the first test the workers cannot satisfy the two parameters, "number of requests per time interval" and "average time of response", thus goals 6 and 7 should be fulfilled.

2. In the second situation the workers capacities are good enough to meet the parameters in the license types, and goals 1 and 2 should be satisfied.

Table 4.2: License types used during the simulation

License type	Nb. tasks per time interval	Avg. time of response
1	800	200
2	400	100
3	100	300

Table 4.3: Average processing times of tasks on the workers

Worker	1	2	3	4	5	6	7	8
Test 1	400	700	100	400	1000	4000	90	2000
Test 2	4	7	1	4	10	40	9	20

Tables 4.2 and 4.3 give the general setting of these tests. All time values in this section are given in milliseconds, unless otherwise specified. Requests are issued by three clients/users, each one uniquely associated with one of the license types. Generated tasks were uniformly distributed among the three users, so that they all contribute with an almost equal number of requests. The implemented simulation starts with replicating the arrival of a high number of tasks in a short interval (a spike in tasks arrival), then incoming tasks are received at a normal pace (in the sense that their rate becomes stable). During the simulation only tasks of length in the same interval (similar length) have been considered. The environment consists of eight workers and is assumed heterogeneous from a performance point of view, as it can be seen in Table 4.3.

A more intuitive adapted best-first method has been tested in parallel in order to compare results. In this second approach tasks are sent for processing to the worker estimated to finish the fastest[11]. Also, each worker is attributed

[11]The estimation is made using the known average processing times of tasks for each worker and refers to the current workload of the worker plus the task in question.

a priority queue of tasks, where tasks are sorted by the intuitive formula in equation 4.4, where the two operands on the right side correspond to the two parameters in the license type.

$$\text{TaskPriority} = \frac{\text{NumberOfTasksPerTimeInterval}}{\text{AverageTimeOfResponse}} \tag{4.4}$$

Table 4.4: Results in test 1

	The algorithm		Adapted best-first	
User	Nb. tasks per time interval	Avg. time of response	Nb. tasks per time interval	Avg. time of response
1	152	4572	74	4247
2	74	4717	74	5025
3	20	4259	98	5069

Table 4.5: Results in test 2

	The algorithm		Adapted best-first	
User	Nb. tasks per time interval	Avg. time of response	Nb. tasks per time interval	Avg. time of response
1	71	34	73	32
2	70	18	70	32
3	69	45	67	32

Tables 4.4 and 4.5 contain the results of the two tests, which show that in practice the algorithm acts according to its desired behaviour. The average among three time intervals[12] of the number of tasks processed and the average response time has been computed for each client and each of the two methods.

The results obtained in 4.4 show that the algorithm maintains the proportionality between tasks issued by users with distinct license types, whereas the other method does not. Even the average response time is overall better, mostly because of the additional control strategy (see step 14 in Fig. 4.2).

[12]In the simulation a time interval was 10 seconds long.

In the second case, the average response times specified in the user licenses are met and fairness between them is also satisfied.

Both result sets suggest that the best-first method does not achieve the desired results. This happens because it does not select tasks according to the needed "number of tasks per time interval" parameter versus the potential execution capacity.

A conclusive argument against the best-first method stands out in the following situation. Consider two users A and B whose user licenses' parameters resolve to the same priority value using formula 4.4. If A sends n tasks at time t and B sends m tasks at time $t+1$, then, assuming not all n tasks can be processed in $t+1-t$ units of time, all n tasks sent by A will be processed before any of B's tasks. Given that their priority is the same, the task execution should alternate between the two users. Our algorithm solves this problem, and commutes between tasks of different users, by using distinct queues for each user instead of a single task queue.

4.3 Load balancing when the topology is a ring

This subsection presents an algorithm, called "Ring distributed solver" ([69]), that can work as a scheduler for executing atomic modules where the topology of the distributed network is a hierarchical ring. The master will be the leader of the hierarchical ring network, while the workers are all other nodes. Similar to the default platform, master is handling the coordination of input/output, while workers execute the atomic modules. The proposed algorithm is considering that the atomic modules are created for certain clients. Each client has a license type which specifies the average response time to execute a request. If the average response time can't be satisfied because the distributed network does not have enough computing power, then the algorithm must ensure fairness between clients.

4.3.1 Design of the algorithm

The proposed online load balancing algorithm functions in a hierarchical ring network model where incoming requests are all handled by a single component - the leader node of the hierarchical ring network. The results obtained after solving a request on a node are sent back to owner of the request, directly from the same node. The algorithm allows for the system to be

heterogeneous, for instance workstations may differ in processing capacity. The processing time of a requests is expressed as the "requests' length" and can be predetermined. We assume that `GetEstTimeToCompute(request)` returns the estimated processing time of a request on any node and its time complexity is constant. One simple way to do this is to benchmark how fast each node can execute each type of request length, then group and store these results in a data structure on the node. It is considered that request processing is workstation independent (all types of requests can be processed by any node). Requests are independent (the order in which requests are processed does not affect the correctness of the result) and indivisible (any request can only be handled by a single worker at one moment). The communication time is not generally important for the algorithm. The reason is that while a request spends time moving through the network the priority of the request increases.

High level implementation and the communication protocol

To send data between nodes, two functions are used: `Send`(data) - used to send data to the next node on the same sub-network and `SendToSubNetwork(data)` - which sends a message from a leader to its coordinated sub-network. (This helps moving a message from a higher level network to a sub-network). Another important function is `IsLeaderNode` which has two prototypes. The first one does not have any parameter - testing if the node is the leader of a sub-ring - and the second one, has a parameter representing a message which test if the current node is a leader and it is the one who created/added that message in its sub-network. There are two types of messages used in the communication protocol: `Gather` and `Request`. A Request message is used for sending requests between nodes and contains the following: data context for request execution, the average response time specified in the owner's license, timestamp when created, the time when should ideally finish, and the current bonus time. The code below shows the high level implementation of the decision making when a Request message is received by a node. A node that is not the leader at the level where a request message is sent can either store the request for later execution or send it further in the same network level. Additionally, a leader node has the option to send the request down in its sub-network. Users might also want to relax the conditions and not decrease the bandwidth performance with requests that are travelling the ring many times in order to find a node that accepts them. To make this possible, when

a leader receives back a request that it previously sent to its sub-network, the bonus time variable on the request will be increased. An interesting property of this communication protocol is that if a request travels again back to the leader node of a sub-network because of the high workload, it can eventually get to another sub-network, if the leader evaluates that it is better to do so.

```
OnRequestReceived(request)
   if (CanExecuteRequest(request))
      AddRequest(request);
   else if (IsLeaderNode(request) AND
CanExecuteOnMySubNetwork(request))
   {
      if (already received this request)
         request.bonusTime += BONUS_STEP
      SendToSubNetwork(request)
   }
   else
      Send(request);
```

A `Gather` message is initiated by leaders of each sub-network at fixed time periods and sent only to the nodes on the same level. The role of this message is to have a snapshot of the node's state such that it can later take decisions. This information will be used to take a decision if a leader node should accept a request or not to be executed in its sub-network. The gather messages are asynchronous between different sub-networks. When a node receives a gather message, it will add its local state information (like how much load is in there) to the message and sends it further. When the message is received by the leader it will update its state information table. The code below presents the action code of every node and the handler function for receiving `Gather` messages. `lastTimeGatherSent` is a variable where we store the timestamp of the last `Gather` message sending occurred. `T` is threshold value set by the user, depending on how often he wants to send the Gather message. `Solve` function is supposed to run the effective job on the request. Function `ExtractNextRequest` is selecting the task with the highest priority from the local list of tasks.

```
OnUpdate()
   if (IsLeaderNode()AND(GetCurrentTime()-lastTimeGatherSent) >T)
   {
```

```
      lastTimeGatherSent = GetCurrentTime()
      Gather msg
      SendToSubNetwork(msg)
   }
   request = ExtractNextRequest()
   if (request != null)
      Solve(request)

OnGatherReceived(msg)
   if (IsLeaderNode(msg))
      UpdateLocalState(msg)
   else
   {
      AddLocalState(msg)
      Send(msg)
   }
```

4.3.2 Decision making to execute a request locally on a node (leaf or leader)

To make computations easier, the average response times specified in the clients' licenses are normalized. If $t_1, t_2, t_3, ..., t_n$ are the average response times and $t_{max} = \max t_i$, then $\overline{t_k} = \frac{t_k}{t_{max}}$. Requests are stored and evaluated based on their priorities. The priority of a request is defined as the waiting time for the request to be solved divided by the inverse of the normalized average response time specified in the owner's license. If we denote with`WaitingTime(request)` the waiting time of the request to be solved, and `Owner(request)` the index of the owner license then the priority computation can be written as `Priority(request) = WaitingTime(request)*Owner(request)`, where `WaitingTime(request)=CurrentTime()-request.createTime`. Priorities of requests waiting on a node are dynamic and could modify in time. This is a key point of the algorithm which gives the fairness between the clients with respect to their average response times. The formula used also helps in the case of requests that travel in the network for a long time. The priority of a request increases with its waiting time regardless of the license specifications. If a requests travel a long time then it has a bigger priority and its chances to be added in a node are increased.

Requests are stored in a node using a linked list. At each query of func-

tion `CanExecuteRequest(request)` the algorithm iterates over the existing items and sum up the time needed to compute all requests that have a priority higher than the new request. Using the bonus modifier and comparing the result with the average response time of the request's owner we can find out if the request can be executed on that node or not. Also, in the case of requests that are close to ideal execution deadline or should have been executed until now, we check if their priorities are higher than all other priorities waiting in the node. If this happens, the node is a good fit for the request because it will be the first request selected for execution, thus minimizing the delays in response time. A pseudocode for this is given below. Variable `requestsList` is storing requests of a node; `request.bonus` represents the bonus time given by the leader, and `request.idealTimeOfFinish` is the precomputed time when the request should be solved in order to satisfy the owner's license specification. Variable `T` is a threshold value defined by user. It could be either the average time for moving data between consecutive nodes, the average time needed for processing it, or a heuristics combining these.

```
CanExecuteRequest(request)
  totalEstTime = 0;
  newEstTime = GetEstTimeToCompute(request) / request.bonus
  bestPriority = null;
  foreach req in the requestsList
  {
     if (Priority(req) > Priority(request))
     totalEstTime = totalEstTime + GetEstTimeToCompute(req);
     if (Priority(req) > bestPriority)
     bestPriority = Priority(req)
  }
  remainingTime=request.idealTimeOfFinish-(GetCurrentTime()+totalEstTime)
  isCloseToDeadline = (request.idealTimeOfFinish - GetCurrentTime()) <= T
  return (remainingTime >= 0 OR
  (isCloseToDeadline AND bestPriority < Priority(request)*request.bonus)
```

4.3.3 Decision making to execute a request on a sub-network

This type of decision can be taken by a leader node. In order to make this possible the gather messages are sent in the sub-network in order to collect

workload information. In an ideal case, a leader would obtain information about all requests from its sub-network nodes and run the same CanExecuteRequest function. But such a message would be too large creating a bandwidth and processing time overhead. A tradeoff solution between performance and decision result is to gather statistics on how much time the current requests would take to execute on different intervals of priorities. If P_{high} and P_{low} are estimated bounds of the priorities, then splitting on N equal intervals would result in $TimeSum_i$ storing the sum of times to solve the requests with priorities in interval $P_{low} + \frac{P_{high}-P_{low}}{N} * i, P_{low} + \frac{P_{high}-P_{low}}{N} * (i+1) - 1$. The tradeoff can be adjusted using variable N.

The gather message will contain the `TimeSum` array. When adding the local state to a gather message, the nodes' main responsibility is to iterate through all its waiting requests and for each one to add in the corresponding array index (the correct priority interval) the time needed by the node to solve it. The leader will keep the final `TimeSum` array and use it for decision making. To find out if a request can be executed by the leader's sub-network we need to sum up the values of all intervals of greater priority than the considered request. Then, we divide this sum to the number of nodes in the sub-network to find out the average time needed to finish all higher priority requests. The close to ideal deadline test is used here too, but this time we check if there are any values bigger than zero on intervals with greater priority than the considered request. Below is presented the pseudocode for adding a local state to the gather message and the decision making of a leader if it should accept or not a request in its sub-network. `GetPriorityInterval` does simple mathematical calculations to get the interval index from the priority of a request.

```
AddLocalState(message)
  foreach req in the requestsList
   message.TimeSum[GetPriorityInterval(req)]+=GetEstTimeToCompute(req)

CanExecuteOnMySubNetwork(request)
   P = Priority(request)
   totalTime = 0;
   for i = P +1 to N
      totalTime += TimeSum[i]
   averageTotalTime = totalTime / NumNodesInSubNetwork
   remainingTime=(request.idealTimeOfFinish -
(GetCurrentTime() + averageTotalTime))
```

```
isCloseToDeadline=(request.idealTimeOfFinish-GetCurrentTime())<=T
return remainingTime>= 0 OR
(isCloseToDeadline AND there is no TimeSum[k]>0 with k from P+1 to N)
```

4.3.4 Simulation results and conclusion

To demonstrate that the algorithm satisfies the proposed goals, a simulator in MPI has been created. The nodes are processed on different machines connected in a network. The test implies a total of 64 processes over 8 machines. Random requests where continuously generated with a normal distribution in length classes. The estimated times to compute the requests were between $[10, 500]$ milliseconds, depending on the computing power of the node. The same interval was used for the average response times in the client's licenses. Two relevant tests are used to show how the load balancer works. The results are compared to results of the algorithm in [2].

Test 1: Check the response times with different workloads. Table 4.6 shows a comparison between both algorithms in terms of response time delays, given as a percentage value from the value promised in the owner's license. In the proposed algorithm the average response time goal is satisfied, with important delays appearing just when the workload was too high (A workload of 200% means that in the analyzed time interval, the system had twice the amout of work that it could solve for the given time interval).

System workload	Average delays in response time - Ring distributed solver	Average delays in response time - algorithm in [2]
20 %	1.23 %	15 %
50 %	1.45 %	26.4 %
100 %	1.72 %	54.8 %
200 %	103.14 %	104.2 %

Table 4.6: Shows average response times with different system workloads.

Test 2: Check the fairness between requests with respect to the average response time when the available hardware resources are not enough for satisfying the requests. The simulator created random requests to simulate a high workload.

The results tables above demonstrate that the two goals are satisfied by using the new algorithm: satisfy the average response time if the computing

Intervals of average response times	Average number of requests solved in proposed algorithm	Average number of requests solved in [2]
10-100	4233	1651
101-200	2397	1675
201-300	1683	1649
301-400	779	1693
401-500	541	1680

Table 4.7: Number of requests solved for different average response time intervals.

power allows this and keep the fairness between clients with respect to the response times specified in their license. We can observe the fairness by checking the ratio between the number of request solved and the average time guaranteed. Ideally, the ratios should be equal between all intervals.

4.4 Balancing tasks considering deadline time and fairness

The algorithm presented in [70] and called "Greedy-deadline balancing", can be used to schedule the atomic modules in the case that they have a deadline that need to be satisfied. As with the previous algorithms, we must ensure fairness as a secondary objective if the first one is not technically possible.

4.4.1 Overview

In this section we propose an algorithm for load balancing of jobs in a distributed network assuming central coordination. Jobs are non-preemptive, received by a single machine in the distributed network (master) and sent to workers. Each job has a given deadline which is assigned by the owner of the request. The master must decide if the job can be executed by one of the workers considering its deadline and an error window, and if it does, then who is the best worker to execute it. Once received by a worker, it must decide where the new job should be added on its own waiting list of jobs. The algorithm can work both for homogeneous and heterogeneous workers. It all depends on the ability of a worker to determine the execution time of

a job. If workers can estimate how much time it will take to execute a given job within the considered error window then we can have heterogeneous machines in the distributed system. Various methods for doing such estimation are presented in [1]. One method is to assign to each job a length-class. and test each worker in the system how much time it will take to execute each kind of length-class. There is a linear search complexity to determine the best fit for a new job, but this is needed to make the load balancing better. Also, we assume that:

- there is a communication link between the master machine and each worker
- we can estimate the average communication time for each job (for simplicity, this communication time is included in the execution time of a job).

4.4.2 Algorithm design and implementation

Jobs are received and sent further by the master machine. The algorithm does not move any job from a worker to another because each time when we give a job to a worker, we know that it can satisfy its deadline constraint. Also, jobs are non-preemptive. The algorithm is separated into two views:

- **Master view**: responsible for assigning a new job to a worker if there is one available to execute this job satisfying its deadline constraint.
- **Worker view**: responsible for managing a data structure that holds jobs and extracting / executing these jobs.

A data type that defines a job can be defined as: `JobType= { timeToExecute, deadline, timeToStart, timeToEnd, dataContext }`. Variable `timeToExecute` is the time needed by an worker to execute the job, while `dataContext` is the data associated with the job execution; `timeToStart` represents the time when a job can start on a worker; finally, `timeToEnd` is the computed value of `timeToStart + timeToExecute`.

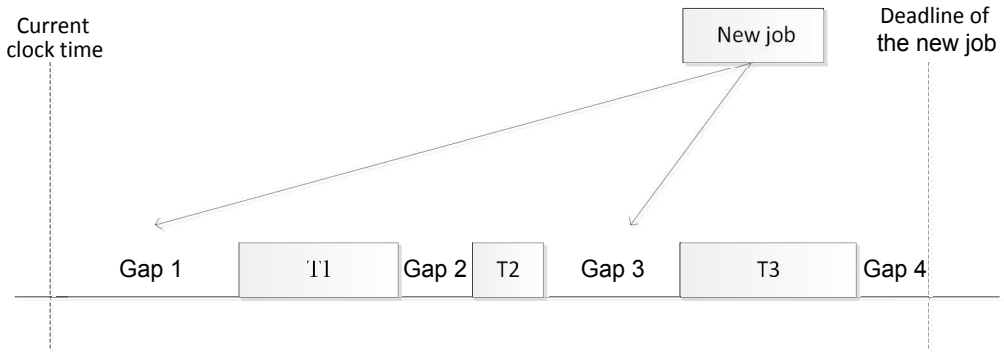

Figure 4.4: Adding a new task to an existing list of jobs (T1, T2 and T3). Considering that the width of the rectangle represents the execution time of a job, then gaps with index 1 and 3 can fit the new job. The end of gap 3 will be preferred for the new job to schedule it as late as possible.

Master view

Master's role is to send a new job to the worker which can start it as late as possible, but still satisfying the deadline constraint. It is a greedy solution which can keep workers available for earlier deadlines. The function `GetTimeToStart` returns the time when a worker can start a job given as parameter, and -1 if it is not able to execute it and satisfy its deadline goal. The pseudocode that master runs when a new job is received is presented below.

```
OnNewTaskArrived( JobType task)
   bestWorkerId = -1
   bestWorkerTime = 0
   foreach worker W do
      Wtime = Controller[W]->GetTimeToStart()
      if  Wtime != -1 AND  bestWorkerTime < WTime
         bestWorkerTime = Wtime
         bestWorkerId = W

   if bestWorkerId != -1
      SendTask(job, bestWorkerId)
   else
      // Code for refused job
```

The `Controller` array is stored on master and represents the state of each worker - the data structure which stores informations about the currently assigned jobs for each client, excluding the `dataContext` field which is only needed by workers.

4.4.3 Worker view

There are two issues at worker's view:

- the `GetTimeToStart` function implementation (called by the master and having its context data stored on the master in the `Controller` array)
- how a worker manages its internal data structure to execute jobs.

The same "greedy idea" as in previous subsection is used here: schedule a new job as late as possible. Workers are using a linked list to store the assigned jobs. In this linked list, jobs are sorted in ascending order by the value of the field `timeToStart`. The value of this field for a new job should ideally be: `deadline - timeToExecute`, because the main objective is to promote free spaces for new jobs that have earlier deadlines. But if this is not possible due to existing jobs, then it should be set to the last gap found between assigned jobs that allows us to execute the new job and satisfy its deadline.

Pseudo-code for this operation is presented below. The `mList` variable represents the linked list where the jobs are stored. `mList.end/mList.start` represents the last/first element in the list. If there is no other assigned job in the list or we can schedule the new job after the last one, then the ideal value for `timeToStart` will be deadline - timeToExecute. Otherwise, the algorithm tries to fill the first gap found starting from the end of the list and going to its beginning. Each time we compare two consecutive elements in the list (`T` and `prevT`) and check if the new job can be added between the `timeToEnd` of `prevT` and the minimum between its deadline and the `timeToStart` of `T`. If we find such a position then we set the `timeToStart` as late as possible in this gap. Finally, if no gap was found yet, we try to add it in the gap starting from current clock time to the beginning of the first job in the list. The implementation of `GetTimeToStart` will also cache the `timeToStart`, `timeToEnd` and the position in the linked list where it should be added (`mLastCachedPos`). This information will be sent together with the

job in the SendTask function to avoid doing the linear search twice.

```
GetTimeToStart(newJob)
  // Check the back of the list first
  If (mList.isEmpty()  OR
     mList.last().timeToEnd<=(newJob.deadline-newJob.timeToExecute))
     newJob.timeToStart = newJob.deadline - newJob.timeToExecute
     mLastCachedPos = mList.end
     return newJob.timeToStart

  // Check for gaps between tasks, starting from last to first
  foreach job T in mList (reverse order)
    revT = T->prev
    if  prevT == NULL continue
    deadlineImposed = min(newJob.deadline, T.timeToStart)
    if prevT.timeToEnd+ newJob.timeToExecute<=deadlineImposed
       newJob.timeToStart=deadlineImposed-newJob.timeToExecute
       mLastCachedPos = position of T in mList
       return newJob.timeToStart

   //Check for a gap between current clock time and first job begin
   Tfirst = mList.first()
   deadlineImposed = min(newJob.deadline, Tfirst.timeToStart)
   if (clock() <= (deadlineImposed - newJob.timeToExecute))
     mLastCachedPos = mList.first()
     newJob.timeToStart = Tfirst.timeToStart - newJob.timeToExecute
     return  newJob.timeToStart
```

On each worker there is a function which continuously polls for jobs in the list and grabs them for execution. The trick is to allow grabbing and execution of the first job from the list even if the current clock time is less than its timeToStart field. Doing this will keep the machines busy. It is possible that some of the jobs with a deadline close to current clock time will be refused, but it has the same probability (assuming a normal distribution of jobs in time) that other new jobs will benefit from this. The complexity of searching for the best place to add a new job inside a worker is linear in the number of existing jobs on that worker. The worst case happens when there are many jobs received in a short time interval and the jobs execution time is higher than the arrival time rate of new jobs.

4.4.4 Simulation and results

To test the performance of the "Greedy-deadline balancing" algorithm, a simulation was made in order to see its behaviour in comparison with another two load balancing algorithms(Round Robin and Greedy worker selection). The first one sends the jobs in a round robin policy while the second one to the worker that can execute the job as late as possible. Both have just a simple policy at the worker's view: add the new job to the end of the queue if it can be executed before its deadline expires. Ideally, the load balancing should use the resources correctly by keeping the hardware busy most of the time and minimizing the number of refused jobs.
The results in Fig. 4.5 are obtained using test samples of 1000 jobs with a normal distribution of execution times between 10 and 200 milliseconds. The arrival time rate of new jobs was between 1 and 10 milliseconds. Deadline time extension of each job (time since a new job was received to when it should finish) was also chosen by a normal distribution in interval [10, 2000] milliseconds (considering the job execution time too). The tests where run on 24, 16, 12, 8 and 4 machines in a local network (workers) each having a single hardware process dedicated for our job execution. The process of receiving and assigning a new job to a worker was done by a separate machine, called "master".

Figure 4.5 shows how many jobs where refused depending on the number of machines and the algorithm used. The results graph shows that the "Greedy-deadline scheduling" algorithm is better than "Greedy worker selection" and "Round robin" selection, despite its overhead. The difference between it and the other two algorithms increases with the number of machines used. When using 16 machines, the number of jobs refused by the other two algorithms is with 49% higher than the proposed algorithm. With 24 machines, the "Greedy-deadline scheduling" algorithm succeeded to obtain 0 refused jobs while the other two solutions had 18 respectively 21 refused jobs. The samples used create jobs in a short time interval (defined at the beginning of this section) to represent a worst case scenario for the proposed algorithm. When jobs have longer execution time and the arrival time has a different time distribution, the proposed algorithm can perform even better than this because there is less overhead spent on decision making.

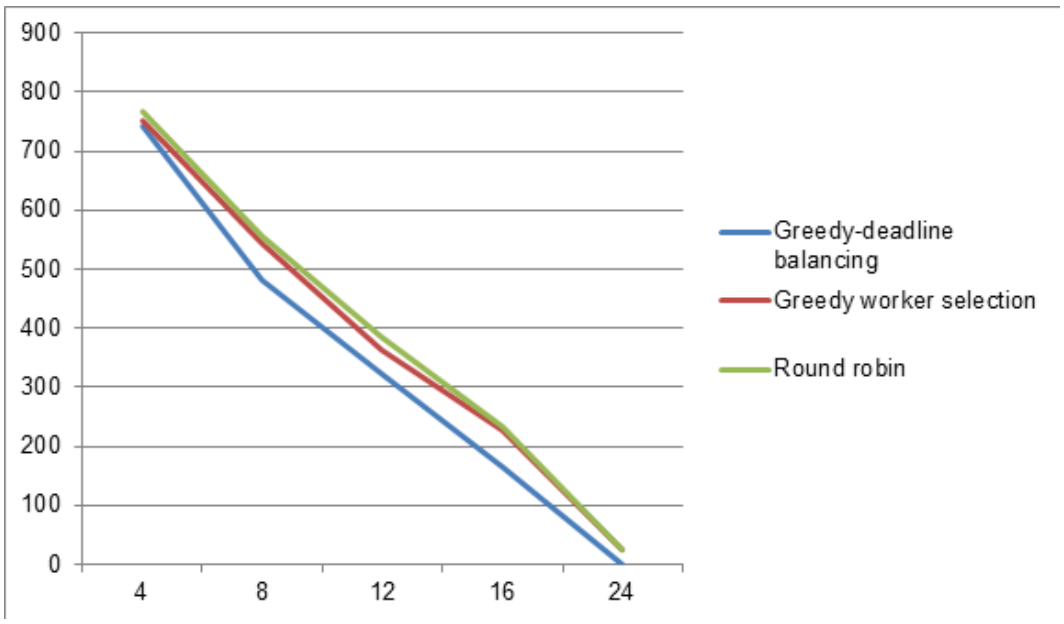

Figure 4.5: The average number of refused jobs (1000 was the total number of jobs) in test samples by each algorithm.

4.5 Conclusion

This chapter started by presenting what is happening behind scenes when atomic modules are ready for execution, how they are executed, and the communication involved between the master process and workers. The default scheduler and topology implemented in the current version of AGAPIA are also presented.

The second part of the chapter presented some algorithms for adding a diversity in the schedulers and topologies used. There are three algorithms presented and each one can be considered as and improvement over the existing ones in the field. The first one is for balancing the tasks in a centralized distributed network where master can comunicate directly with all workers, similar to the default topology used by AGAPIA. Tasks are supposed to be atomic modules owned by clients (the same analogy between tasks and atomic modules is used in all three algorithms). Each client has a licence type which specifies the guaranteed average number of atomic modules per a time unit. The algorithm is trying to schedule tasks such that to achieve fairness between clients with respect to their licence types.

Another algorithm, that has the same requirements but uses a hierarchical ring topology is presented in the continuation. Finally, the last algorithm is for balancing tasks (using the default topology), where each one is considered to have a deadline. The main objective of this algorithm is to schedule as many tasks as possible.

Chapter 5

Applications and extensions

This chapter is composed of four sections. The first two sections presents two types of applications that are suitable to implement in AGAPIA: dataflow and wavefront applications. Section 5.1 contains a brief presentation of dataflow programming, how AGAPIA can be used for this paradigm and the implementation of one distributed signal processing application. The implementation demonstrates that by using this language programmers can be more productive with minimal performance losses. Section 5.2 presents how to benefit from using AGAPIA in parallelizing the wavefront pattern. Programmers can solve these types of problems in AGAPIA, for parallel computation, with an effort similar to their corresponding serial implementation. Again, the performance/memory overheads are minimal.

The last two sections propose an extension for the current version of AGAPIA and a comparison to another new programming language for parallel computing, X10. Section 5.3 provides another perspective for using the language: spatio-temporal data streaming applications. It is a conceptual discussion of future implementation, not included in the current compiler version. It could be used for building applications that run on moving objects such as airplanes, cars, and so on. With very high-level specifications, users could easily build applications that takes spatio-temporal data streams as an input and produces streams as outputs for use by other applications such as actuator controls. By providing ready to use modules for data manipulation AGAPIA enables a separation of concerns: application programmer can focus on their application model rather than low-level details about data manipulation and communication. Finally, section 5.4 contains a comparison between a new programming language for parallel computing, X10, and

AGAPIA, including an analysis for using X10 in its runtime.

5.1 Dataflow programming using AGAPIA

Dataflow Programming is a programming paradigm whose execution model can be represented by a directed graph, representing the flow of data between nodes, similarly to a data flow diagram. Considering this comparison, each node is an executable block that read data inputs, performs transformations over them and then forwards the results to the next block. A data flow application is then a composition of processing blocks, with one or more initial source blocks and one or more ending blocks, linked by directed edges. The dataflow model provides pipeline parallelism by its very nature. As an operator consumes input it pushes its output downstream. Operators normally do not have to see all of their input before producing output. As each operator works, a pipeline of data makes its way through the graph. This makes it suitable for many-core parallelism because it hides communication details while keeping a potential for massive parallelism. Dataflow programming is applicable in both many-core and multi-core platforms. In classical imperative programming, a program is basically a set of operations working with mutable state, which can actually hide data paths. Building a parallel program in a traditional programming require the usage of callbacks and synchronization objects, such as locks, to coordinate tasks and access to shared data. This can lead to deadlocks, priority inversions, hard to understand and maintain programs with possible limited parallelism.

The main goal of this section is to analyse the possibility of using AGAPIA language in dataflow computing. To this end, we start from low-level concepts then we present a concrete application development.

5.1.1 Related work in Dataflow programming languages and a comparison to AGAPIA

Switf/T is a recent programming language presented in [89]. The idea of Swift/T's backend is similar to AGAPIA. The "`Turbine execution model`" handles data dependencies between tasks, flows the data inside the execution graph, and outputs the tasks - user defined functions - that can be executed. These tasks are then distributed, similar to AGAPIA's backend, using a load balancer over MPI (Asynchronous Dynamic Load Balancer). There are two

main differences between AGAPIA and Swift/T. First, the code in Swift/T is similar to an imperative programming language using specific dependencies or constructs that Turbine understands, while the code in AGAPIA is structured like the data flow graph of a program in a modular way with defined input and output interfaces for nodes. Also, AGAPIA automatically understands which code can be compiled directly with C/C++, while Switft/T needs to define "leaf functions" separately. SigmaC [8] is another new language for dataflow computation targeted for execution on low-power consumption Kalray MPPA processors. To create an application in SigmaC users have to define the execution graph with nodes represented by agents and links between them. In comparison with SigmaC which has a static structure in terms of nodes and size of data transferred between nodes, AGAPIA supports a dynamic execution with the possibility to add nodes and links at runtime. Also, AGAPIA supports data flow branching.

Yet another dataflow programming solution is Task Parallel Library - Dataflow in Microsoft .Net Framework 4.5 which contains helper blocks to create data flow actor-based programming only for multi-core architectures. [80] is a survey which describes how visual programming languages build on top of dataflow programming can be used for end-user programming and how easy it is to achieve concurrency by applying the paradigm, without any development overhead.

5.1.2 Why is AGAPIA suitable for Dataflow programming

AGAPIA is capable of expressing massive parallelism in a manageable way for programmers, allowing to build dynamic nodes and links in the data flow graph at runtime. The nodes of the dataflow graph, also called programs, are modular and reusable. The communication is transparent for users allowing them to concentrate on the high level flow and algorithm.

There are a few distinguished features of AGAPIA, compared with other dataflow languages, which will be presented in the continuation of this section. A program can be viewed as a dataflow graph where nodes are modules and links define the data flow between nodes. Recursion inside nodes is possible and a node can become this way a sub-graph. An important feature of AGAPIA model is that the dataflow graph is dynamic. New nodes instances,

links and data flow paths can be created at runtime. This can improve the potential parallelism by dynamic task creation at runtime, which is usefully for applications such as radio cognitive or other applications that generate tasks depending on the intermediary computed result. Another important feature is that users do not need to define explicitly the links between nodes. Each module has input and output interfaces and the composition of modules will try to match the interfaces and automatically create links between inputs and outputs if the interfaces match according to the composition operator. Having two input/output interfaces, as next subsection presents, adds more expressivity, allowing for both horizontal and vertical programs linking. An AGAPIA program's code is mostly composed of C/C++ language code and a few high level statements, composition operators and interfaces descriptions. This decision was made to support legacy code and to be more appropriate to well-known programming languages. Users can also include external libraries or code to facilitate the development of complex programs in AGAPIA.

5.1.3 A dataflow application implemented in AGAPIA

This subsection is dedicated to the presentation of a more complex example by using a combination of high-level statements and to the evaluation of the performance, time of development and expressivity of the language. A short comparison with MPI in terms of performance is shown in order to evaluate the memory and speed overheads caused by using AGAPIA. The accent is put on the architecture of the application and how the language hides communication details and keeps a program modular and closer to how users think a solution.

The problem discussed here is to build a complete application where a user reads images from a camera, applies a preprocessing (blur) over the images then broadcast it to several clients, and also save it on local storage drive for later reuse. The application uses OpenCV and Socket API to demonstrate the ability of AGAPIA to reuse exiting code and libraries.

First, we define an image information structure which contains pixels data and dimensions: `struct ImgInfo {data:buffer, width:int, height:int }`.

Fig. 5.1 shows the high-level picture and code of the application proposed. It is composed of 5 high-level modules:

- SCREENSHOT: an atomic program with OpenCV code which captures a screenshot from a webcam device. This image will be sent to the POSTPROCESSING module.
- POSTPROCESSING: blurs the input image in parallel using multiple tasks assigned to machines in a local area network. The output image is then sent to BROADCAST and SAVETOHDD modules.
- BROADCAST: broadcast the received image to a list of IP addresses. It uses a common socket API for the low level implementation.
- SAVETOHDD: saves the received image to local storage for later reuse.
- MAIN: It is the start module and coordinates the high-level flow of the application.

Between POSTPROCESSING programs we use a vertical dependency because it is better to use all available hardware resources in order to complete the post processing of a current frame than to buffer many frames. The entire application is composed by pipelines where a SCREENSHOT program from a new frame can be executed in parallel with a POSTPROCESSING program from the previous frame. This kind of dependency is easily achieved in AGAPIA just by defining at least one variable in the input/output interfaces and using composition operators.

```
module MAIN {listen nil}{read nil}
{
  while_t(true)
  {
    SCREENSHOT # POSTPROCESSING # (BROADCAST
                                % SAVETOHDD)
  }
}{speak nil}{write nil}


module SCREENSHOT{listen nil}{read nil}@MASTER
{
 CvCapture* capture=cvCreateFileCapture( "C:\\test.avi" ) ;
 IplImage* image = cvQueryFrame( capture );
 img.Copy(image->imageData,image->imageSize);
 img.width = image->width;
```

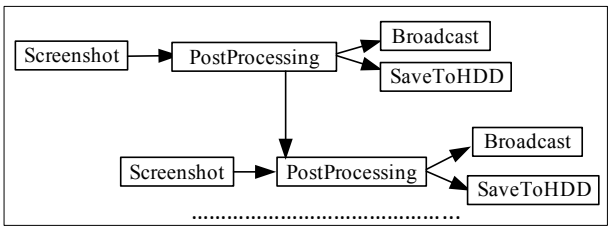

Figure 5.1: High level code and input flow between main modules.

```
  img.height = image->height;
 }{speak img:ImgInfo}{write nil}
```

The `POSTPROCESSING` program is generating a number of tasks from the input image to be computed in a local area network. Fig. 5.2 shows how this program looks like and its code. First, this program flows its input to a child: `DISTRIBUTETASKS`. This has the role to split the image received into multiple tasks with (almost) equal amount of work. Each task is also an `ImgInfo` structure but containing just a linear block of lines from the original image. A program of type `COMPUTETASK` has the role to get an image part, apply a simple blur algorithm and send it further. All the results are then gathered into `GATHERANDSEND` program which will send the final output of the `POSTPROCESSING` program.

Notice that programs `DISTRIBUTETASKS` and `GATHERANDSEND` have `@MASTER` specifier. This is used to reduce the communication times. When `buffer` data type is sent between programs on the same machine it will use the address of the data instead of making a copy of it. Since the master is producing the actual data (in `SCREENSHOT`) then it makes no sense to make another copy of the image just for setting some values. The same feature is used for `GATHERANDSEND`: since the partial results are received by master machine, it does not make sense to assemble them in a worker machine. This is an important topic that users must understand in order to optimize the communication time.

Another important thing to notice is how the `GATHERANDSEND` program has an implicit gather. As stated in Chapter 2, if a program is atomic and its input interface contains an array of processes, then it will wait for all indices to be set before executing the program. The mapping in this case is one to one between array indices and tasks. Each task will execute and set its own index in the process array. This way, the image parts will be ordered correctly

when GATHERANDSEND starts execution. The COMPUTETASK instances do not have any dependencies between them in the vertical composition (since the north and the south interfaces are nil) so they can be executed in parallel, in any order on any of the machines.

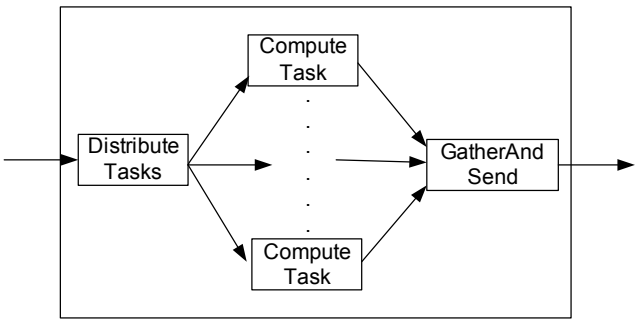

Figure 5.2: Tasks computation input flow.

```
module POSTPROCESSING{listen img:ImgInfo}
{read:nil}
{
  // N - number of worker tasks. In the
  // simulation we used the number of machines
  DistributeTasks # foreach_t(N){Compute Task}
  #GatherAndSend
}{speak img1:ImgInfo,img2:ImgInfo }{write nil}

module DISTRIBUTETASKS{listen img:ImgInfo}
{read nil}@MASTER
{
 for (int i = 0; i < N; i++)
 {
  // do simple math to compute the dataBegin-
  // address in the image data for this task,
  // height, width of this task and the size
  // of the data (also send the border lines)
  ......
  // set task info
  imgArray@[i].data.SetData(dataBegin, size);
  imgArray@[i].height = height;
```

```
  imgArray@[i].width = width;
 }
}{speak imgArray : (ImgInfo;)* }{write nil}

module COMPUTETASK{listen imgIn:ImgInfo}{read nil}
{
 // C language code to load an image with
 // OpenCV using imgIn.data.GetData() and
 // call opencvblur to copy this in imgOut
}{speak imgOut : ImgInfo }{write nil}

module GATHERANDSEND{listen imgArray:(ImgInfo;)* }
{read nil}@MASTER
{
 // assemble the image from the individual
 // results (they are in the right order in
 // the array) then fill in the same image
 // into imgOut1 and imgOut2  (again, C
 // language code)
}{speak imgOut1:ImgInfo,imgOut2:ImgInfo}{write nil}
```

`BROADCAST` and `SAVETOHDD` programs are atomic ones. The former receives an image and sends it to a predefined list of IPs (`BROADCAST`) while the latter saves it for later reuse (`SAVETOHDD`). Both programs do not have dependencies or `@MASTER` specifier. This means that they will be executed in parallel on any of the workers.
In terms of development productivity and code size the AGAPIA implementation performs better than a message passing programming solution such as MPI when it comes to applications with complex data flow because:

- The communication is transparent, the user does not need to write specific primitives for sending/ receiving data
- By having the scheduler already implemented in the background there is no need to create a complicated scheduler by hand as the MPI implementation needs in order to compute the similar jobs.
- Faster development time because a user can write the code closer to how he thinks the solution.

- Better modularity and less error prone than MPI. User can change at any time a module implementation just by keeping the correct interfaces. It is less error-prone because of the transparent communication model and modularity.

These should be targets for many dataflow programming languages but what makes AGAPIA a good solution for dataflow programming is:

- Small memory/speed overhead when comparing with an MPI/C++ core implementation of the same problem.

  Both MPI/C++ and AGAPIA implementations succeeded to obtain 30 fps from a 1920x1080 camera on a local area network with 16 hardware processes. We analysed the average times to compute a single frame and memory overhead to run an application instance. Table 5.1 show that the results are very close.

Table 5.1: Comparative results of one frame cycle speed and memory overhead

Solution	Average time to finish all tasks for a frame (milliseconds)	Memory footprint (excluding the memory used for storing image data) - in bytes
MPI	31.7	1756
AGAPIA	32.3	1911

- AGAPIA can add nodes (programs) and links to the dataflow graph at runtime: the code above shows this when it creates and links at runtime `N` instances of `COMPUTETASK` programs. Instead of `N` we could have complex functions for computing this value.

- Transparent communication and high-level statements and patters which increase productivity: As the example shows users do not need to define explicitly any links, the communication being implicit.

- The ability to coordinate the execution with deployment specifiers.

- An application can be expressed in the same file for better visibility not within different files.

5.2 Distributed Wavefront pattern applications using AGAPIA

5.2.1 Introduction to Wavefront pattern

Wavefront is a programming pattern which appears, for instance, in dynamic programming or in sequence alignment applications [58]. In this processing pattern, data elements are organized in a multidimensional grid. The dependecies between elements are given by the formula used to compute each cell (known as dynamic programming). The computation usually starts from a corner of the grid and propagates its effect diagonally. This type of computation is also known as wavefront. The problem of work distribution between parallel processing units creates a nontrivial problem [5]. Parallelization of this pattern is important because in many cases the input dimensions of these applications are large enough to create performance problems. As the next section will show, there are many papers discussing about how to parallelize this pattern using task based frameworks, but still there is no easy way for programmers to use it on distributed architecture without understanding low level details. The aim of this Section is to present another method - and language - for writing wavefront applications for both distributed and shared memory architecture such that user does not have to deal with low level details of the underlying architecture or framework. This section performs an analysis on how the wavefront pattern can be implemented in AGAPIA, its advantages and drawbacks. To demonstrate the productivity and to evaluate the performance, a known dynamic programming problem was chosen for implementation - *the longest common sequence of two given strings.*

5.2.2 Related work and a comparison to AGAPIA

The field of parallelizing the wavefront pattern contains several interesting results. We mention only two contribution in the shared memory category, one of the newest and most important papers is [25], where authors are using a task based model for computing the tasks. Cells become tasks and once

their dependencies are solved they are added in an internal queue to be executed by harware threads. It is shown than the Intel TBB implementation outperforms the OpenMP one. On the other hand, [5] is considering parallelizing dynamic programming problems which can be solved with topdown methods using lock-free hash tables. It includes a distributed solution for wavefront pattern. It uses a modified version of Bulk Synchronous Parallel Model - BSP which assume that the execution runs in steps with barriers for synchronizations.

In comparison with these solutions, AGAPIA implementation of wavefront pattern can run in usual computer machines in both shared and distributed memory models, using the same source code. Another important aspect when considering this language instead of other frameworks is that code is easier to write and user has less things to learn and to care about, as this thesis shows. This makes AGAPIA a more productive language when deadling with wavefront problems, with minimal performance loses. Its drawback, as generally with high level programming languages, is less user control over what is happening in the background.

5.2.3 Implementation of a solution for Longest common subsequence problem in AGAPIA

Longest common subsequence description

Given a sequence $A = A_1A_2...A_N$, a subsequence of A can be formed by deleting some entries from A. For example, (a, d, z) is a subsequence of (a, c, d, b, r, z). The longest common subsequence (LCS) problem accepts as input two sequences - $A = A_1A_2...A_N$ and $B = B_1B_2...B_N$ - and finds the longest sequence that is a subsequence of both A and B. For example, if $A = (c, a, d, g, r, z)$ and B = (a, s, g, z), the longest common subsequence of A and B is (a, g, z). If $LCS[i, j]$ denotes the length of the longest common subsequence of the first i elements of A and the first j elements of B. The objective of the longest common subsequence problem is to determine $LCS[n, m]$, where n and m are length of sequences A and B, respectively. The formula below computes the two dimensional table LCS:

$$LCS[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ LCS[i-1, j-1] + 1, & \text{if } i, j > 0 \text{ and } A_i = B_j \\ max(LCS[i, j-1], LCS[i-1, j]), & \text{if } i, j > 0 \text{ and } A_i \neq B_j \end{cases}$$

A possible solution for coding the LCS problem serially is presented below. The first part is initializing the array while the second one is computing the rest of the two dimensional table.

```
for (i = 0; i <= n; i++)  { LCS[i][0] = LCS[0][i] = 0; }
for (i = 1; i <= n; i++)
     for (i = 1; i <= m; i++)
     {
           LCS[i][j] = max { LCS[i-1][j], LCS[i][j-1] };
           if (A[i] == B[j])
           {
              LCS[i][j]=max{ LCS[i][j], F[i-1][j-1] + 1 };
           }
     }
```

Strategies for parallelization

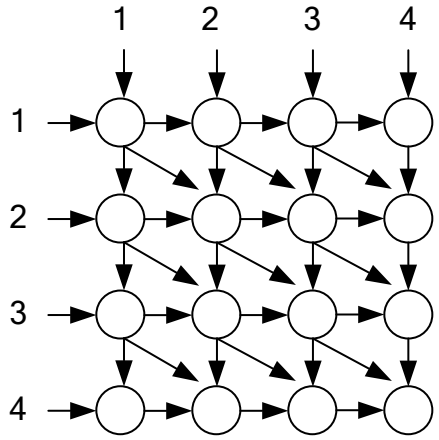

Figure 5.3: Tasks computation input flow.

Assuming that each cell takes one time unit to compute, then the numbers in Fig. 5.3 shows the time unit when each of the input cell is computed.

The work distribution is the most important factor for parallel formulation of a solution for the wavefront pattern. The main objective of work distribution is to minimize the idle time and ensure equal amount of work between processing units. Since the data layout takes the form of a multi-dimensional grid and computation direction is diagonal, computational load at given instance of time differs throughout the sweeping process. Simple partitions across rows or columns are not efficient because of the increased idle time. For example, if simple rows partition is used, then the processing units at the end of the computation must stand idle until the diagonal wavefront reaches the end row. One technique widely used for data distribution in this problem is a cyclic block distribution of data. In this scheme, the grid computation is partitioned into equal sized 2D blocks, and distributed cyclically to all the processing units. Because of cyclic distribution one processing unit that was assigned less for some partition is compensated by the next partition in the cyclic blocks. For sufficiently small sized blocks, this will create relatively even workload for the processing units, but this isn't the case with large inputs.

As with the other papers and solutions presented in the previous subsection, the implementation in AGAPIA considers as tasks, the blocks of cells which can be executed by processing units. When a task is finished it informs its neighbors that the dependency is satisfied and when all dependencies are satisfied, that task is ready for execution. Looking at Fig. 5.3, after cell (1,1) is computed, cells (1,2), (2,1) and (2,2) are informed by this event. Executing and tracking these dependencies is not easy. First, the platform must have a task scheduler in background (such as Intel TBB has) and methods to keep track of dependencies. The newest version of MPI supports dynamic process spawning but it has some disadvantages. First, we need to separate the source code files/executables for main program and the tasks we need to spawn dynamically (which in our case is the one which computes the cells). Then, we need to send and receive messages between the tasks processes and the master process in order to do the dependencies bookkeeping. These two issues makes the code harder to understand, to develop, and to modify later. As the next sub-section will show, AGAPIA has a clear implementation with code in a single file, hiding all the communications and dependencies bookkeeping aspects.

5.2.4 AGAPIA implementation and performance

The first implementation assumes that each cell is computed by a different task. Later on, we extend it to have a task computing a block of cells.

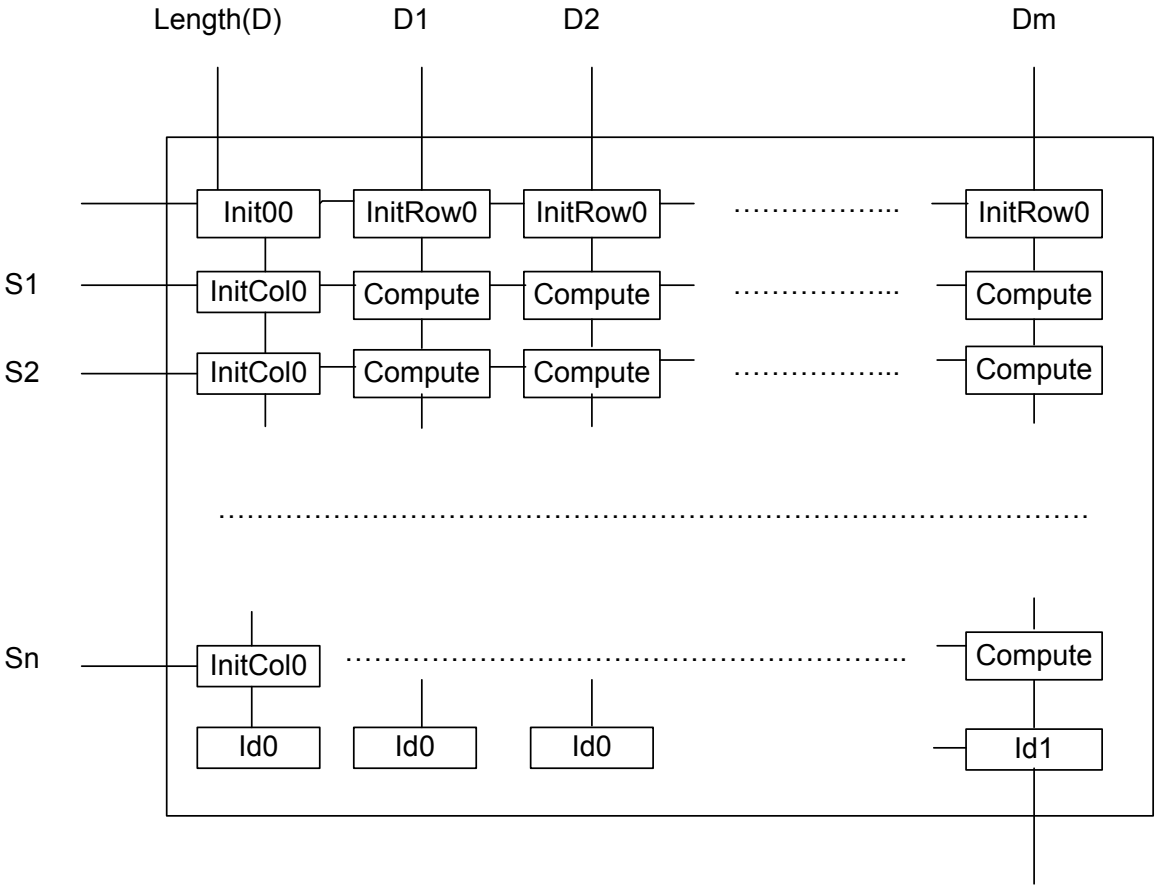

Figure 5.4: Tasks computation input flow.

```
MAIN{listen lengthA:int,A:(char;)[]}{read lenghtB:int,B:(char;)[]}
{
   Init00 # for_s (i=1; i <= lengthB; i++) { InitRow0 }
   %
   for_t (j =1; j <= lengthA; j++)
   {
      InitCol0 # for_s (j =1; j <= lengthB; j++)
```

```
                {
                  Compute
                }
   }
   %
   for_s (j = 0; j < lengthB; j++) { Id0 }  # Id1
} {speak nil } {write Result : int}

Compute {listen DRes: int, LRes : int, chA : char}
{read URes, chB : char}
{
   res = max(LRes, URes);
   if (chA == chB)
   {
       res = max(res, DRes + 1);
   }
}{speak URes : int, res : int , chA : char}
{ write res : int, chB : char }

Init00  {listen nil} {read nil} {res = 0;}{speak res : int}
{write res : int}

InitRow0{listen res : int }{read chA : char} { res = 0; }
{speak res : int } {write res:int chB:char }

InitCol0{listen chA : char} {read URes : int}{ res = 0}
{speak URes, res, chA : char } {write res : int}

Id0 { listen nil }{ read URes:int}{}{speak nil}{write nil}
Id1 { listen nil }{ read URes:int}{}{speak nil}{write URes:int}
```

It is straightforward to extend the code to compute a block of cells in each task. Instead of receiving and sending just one result and one character, we send an array of items. Below is the code for `COMPUTE` module modified (considering a square block size partition).

The connections between modules are presented in Fig. 5.4, while the entire AGAPIA code for LCS problem is presented below. String `A` is received by the main module in the temporal interface while string `B` in the spatial interface. Their characters will flow through the wavefront like in a systolic system such that any cell located at (i, j) will receive chars A_i and B_j. The

dependencies and communication are expressed directly through modules interfaces and compositions. One thing to note is that the diagonal value will be received from the left cell, because AGAPIA has just two input/output interfaces. However, this does not affect the performance since the cell already has a dependency on the left cell so it cannot continue without it. In the `COMPUTE` module, variable `DRes` represents the diagonal result value, `LRes` the result coming from the left cell, and `URes` from the upper side. When writing the output for the temporal side of `COMPUTE` module, `DRes` is replaced by `URes` because this value is the diagonal result that should be considered for the cell in the right side. The other modules are used for keeping the interfaces correctly connected (`Id0`), to do the initialization part (`Init00`, `InitRow0`, `InitCol0`) or to output the final result of the main module (`Id1`).

```
Compute {listen DRes: int[], LRes : int[], chA : char[]}
{read URes[],  chB : char[]}
{
   const int blockSize = LRes.size();
   for (int i = 0; i < blockSize; i++)
      for (int j = 0; j < blockSize; j++)
      {
           // Compute serially and fill results in the output
           // arrays
      }
}{speak URes : int[], res : int[] , chA : char[]}
{ write res : int[], chB : char[] }
```

As the code proves, the implementation of the wavefront pattern in AGAPIA is easier than other solutions, faster to develop, and requires less effort. The only concern of a programmer using it is to imagine the modules, their interfaces, and connections between them. The background scheduler will run the modules as soon as their dependencies are satisfied. The dependencies are specified by modules compositions which should be easier than writing explicitly code as MPI or Intel TBB requires. Also, there is no low level communication involved in the implementation and the same code will work on both shared and distributed memory model. The variables or arrays are stored either in shared memory or sent between machines, depending on the memory model. As often happens, the mentioned advantages come together with a few disadvantages, as it generally happens to high level programming languages. First, because many things happens in the background users

might feel that they do not have full control over the execution. Second, there is a small performance overhead by using AGAPIA. Table 5.2 shows a comparison with MPI for executing an LCS problem with size of $32^4 * 32^4$ on a distributed memory architecture having 16 hardware threads and choosing the block size for tasks that given the best results (in our case 4096). As expected, the performance lose was minimal because the module `COMPUTE`, which is the mostly expensive one, is atomic - does not contain any AGAPIA composition or high level operator - and its code is translated directly into C/C++ code.

Table 5.2: Comparative times to run the LCS problem in AGAPIA and MPI.

MPI time	AGAPIA time
9.4 seconds	9.51 seconds

5.3 Spatio-temporal data streaming

This section presents a possible extension to the current version of AGAPIA language to support temporal pointers. There is an important range of applications that could benefit by using temporal pointers: applications that needs to finish certain tasks in a limited amount of time (real time processing) or those that run on moving objects and communicate between their sub-systems through data streams.

Two case studies will show through code examples how temporal pointers and data streams can simplifying the process of program development in AGAPIA. Creating a global memory (data streams) accessible by modules creates an advantage to some sort of applications.

The first two subsections explain what the temporal pointers are, and how to declare and use them. Next, there are two case studies: one with a DSP application which needs synchronization on the temporal data stream and another one for flight planning which uses multiple independent data streams. Finally, the last subsection is a technical discussion on how the temporal pointers can be implemented in the current version of AGAPIA compiler.

5.3.1 Definition of temporal pointer

A temporal pointer represents a physical address on the interaction stream (a physical time) and is declared by `^X`. To access the data value on the interaction stream at the time address specified by the temporal pointer `X` we use `&X`. The interaction stream looks like a global memory zone, storing data for read/write by modules. When a temporal pointer is declared on a stream, its physical address value is the same as the current virtual time of that stream. By using pointers arithmetic we can refer to certain physical times on the stream. For example, `(t + N)` refers to the physical time specified by `t` plus `N` time units. Internally, a data stream can be represented by a deque. An AGAPIA program can have multiple interaction streams (as the second case study shows). Each interaction stream has a specification given by the user: name, time unit (how long is a tick of the virtual clock inside the stream, in milliseconds), unit size - how much memory does each cell of the stream takes, capacity - number of maximum cells to be stored in the stream.

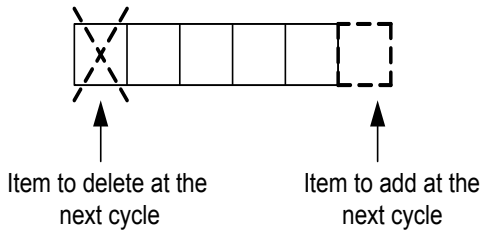

Figure 5.5: An interaction stream with 5 components and operations over the data structure at each new tick.

A possible way to declare an interaction stream is to use the `DECLARE_STREAM` at the top of the AGAPIA code file. It could have the following prototype:

```
DECLARE_STREAM(Name:char*,time unit in milliseconds:int,
cell size:int,capacity:int);
```

E.g, below we show how to declare a stream with a capacity of 100 cells each one having 4 bytes and with a time unit of 1 second, then take a pointer at the current time of the stream and do a few operations:

```
DECLARE_STREAM("Temperature", 1000, 4, 100);
int ^p in "Temperature";
&(p + 1) = 29;
prevTemp = &(p-1);
```

There are three types of outcomes when accessing data in an interaction stream at a time specified by a physical address `t`, depending if the operation requested is a read or a write. If it is a write then:

- `t = current time`, OK.
- `t` $<$ `current time` - invalid access and an exception occurs, as we can't modify data in the past.
- `t` $>$ `current time` - valid access, but the module calling this will be blocked until current time has value `t`.

If it is a read then:

- `t = current time`, OK.
- `t` $<$ `current time` - valid access if the accessed time is still in the deque (`current time - t` $<$ `Capacity of the stream`)
- `t` $>$ `current time` - valid access, but the module calling this can be blocked until current time has value `t`. An exception could be the boolean values where execution can split in two separate branches (using true and false values) and run in parallel until the value is written in the data stream. At that time, the incorrect branch will be cut.

A few rules must be added to support the temporal pointers in a simple way:

- Temporal pointers can be declared only inside the code of a module and not in their interface as in the case of other data types.
- Once declared in a module, it is visible to the modules that are spatial linked to that module. Two modules are called to be spatial linked if there is a chain of spatial compositions between those modules.
- The synchronization between modules containing temporal pointers is made by extending the functionality of keywords `write` and `listen`, to work inside modules. If `tx` is a temporal pointer declared or visible in a module, then a module's body code can use : `write tx` and `listen tx`.

An example is in the Fig. 5.6 which present a program `P = A # B`. In the program `P` there is a module `A` that allocates the temporal pointer `tx` and at each time unit is writing an integer value on the temporal stream. Being declared in module `A` and because there is spatial composition between `A` and `B`, the temporal pointer tx becomes visible in `B`. In applications that need synchronization on the temporal data stream, we can use `listen` and `write` keywords to coordinate this access. In this example, the module `B` can read data from the temporal stream only after `A` writes in. If `B` does not need that synchronization we would omit the `listen tx` instruction and each usage would read whatever data was on the temporal stream.

```
module A{ listen ... } { read ...}
{
  int^ tx in "Name";

  int i = 0;
  while(true)
  {
    &(tx + i) = i++;
    speak tx;
  }
}{ speak ... } { write ...}
```

```
module B{ listen ... } { read ...}
{
  int i = 0;
  while(true)
  {
    listen tx;
    print(&(tx + i));
  }
}{ speak ... } { write ...}
```

Figure 5.6: Two modules which synchronize through temporal pointers.

5.3.2 Case study: Multiple streams in a flight planning application

This case study presents an application for a plane that needs to consider GPS position, weather data forecast from external services, and wind angle/speed from local sensors in order to compute the angle needed to correctly fly towards its destination (wind affects the plane movement). There is a mathematical model described in [42], [47], [43], [44] for finding this. First, the data streams must be defined:

```
DECLARE_STREAM("GPS", 5000,  12, 100);
DECLARE_STREAM("AirSpeed", 1000, 24, 100);
```

```
DECLARE_STREAM("Weather", 2000, 128, 100);
DECLARE_STREAM("CrabSpeed", 1500, 12, 100);
```

The first three streams are input ones, while the last one is output. The main role of the application is to read data from the input streams and fill the output stream with data. This stream will be possibly used by other systems for identifying the correct angle and speed that the plane need to achieve its destination.

All of the streams have different time units and since there are no dependencies between them on read/write operations we must establish a rule for knowing if a cell of a stream was successfully written for the current moment of time or not. Having the default value of each cell initialized in the constructor of the stream cell's data type will solve this problem: each cell has a default value for cells that were never filled. Typically, similar applications would interpolate or consider eliminating noise data so a large number of cells inside streams would be useful.

```
module MAIN { listen nil}{read nil}
{
   RunInputStreams
   %
   RunCrabStream
}{speak nil}{write nil}

module RunInputStreams { listen nil}{read nil}
{
   RunGPS # RunAirSpeed # RunWeatherForecast
}{speak nil}{write nil}

module UpdateCrabSpeed{listen nil}{read nil}
{
   CrabSpeed ^tx in "CrabSpeed";
   time = 0;
   while (true)
   {
      &(tx+time) = GetNewCrabSpeed();
      time++;
   }
}{speak nil}{write nil}
```

Despite of the compositions, all the modules run independently since they do not have any types in their interfaces to create a dependency. Modules `RunGPS`, `RunAirSpeed` and `RunWeatherForecast` are similar to `UpdateCrabSpeed`: they take a pointer on the stream and pass through each time cycle completing the values. Execution of these modules will mostly sleep because they access future time units. The role of the `GetNewCrabSpeed` module is to read data from the three input streams and produce the correct crab speed and angle. The usage of streams and temporal pointers presents a huge simplification in the application's implementation: less variables in interfaces and less complicated compositions.

5.3.3 Case study: A DSP application for image processing

The purpose of this case study is to show the usefulness of temporal pointers in an application which needs to capture data images from a web cam at 30fps, apply a noise filter, then an edge detection and finally save the result on disk for later reconstruction. The image is supposed to have dimensions $W * H$ and BPP bytes per pixels.

The implementation of this program and a figure containing its architecture can be viewed in Figure 5.7. Usually, applications like this use a double buffer technique: one for processing and one for reading a new image, since these operations can run in parallel (`ReadFrame` module can run in parallel with the other modules). The same technique is applied to this implementation, so the data stream for `ImageBuffer` needs to have two cells, each one having dimensions $W * H * BPP$ bytes.

The first benefit of having temporal pointers is that we can impose a minimal speed over operations which is usually needed in real time processing applications or software that runs on moving objects. An exception should raise if program is accessing an invalid time (too old) through pointers. Programmers can have automatic control and easier adaption mechanisms (by optimizing their applications or set a different minimal time on the speed of operations until no exception occurs) in these types of application. In the source code of this case study the time unit of the `ImageBuffer` data stream decides the time needed to complete one frame processing. This was set to 33 milliseconds in order to provide a 30fps processing. If the processing is too slow and considering that the virtual time of the data stream will move

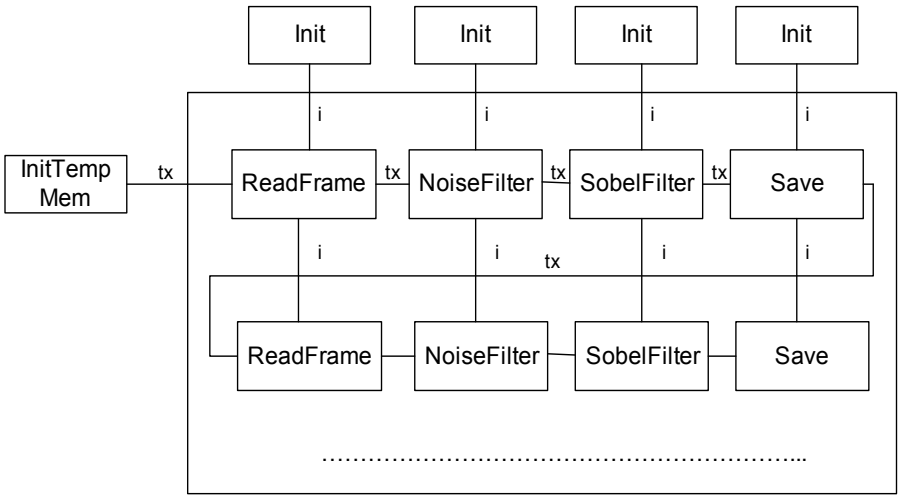

Figure 5.7: Modules and links in a Sobel filter implementation using temporal pointers

on, then at some point the application will try to access data that is too old and an exception occurs.

Another benefit of using temporal pointers is that we can achieve easier communication and even better performance. Having a distributed shared memory and modules running at fixed points we can avoid costly data copies. In an AGAPIA implementation of this application without using temporal pointers the modules will need to exchange the image data if they are executed on different nodes, while the usage of distributed shared memory could be faster.

Another possible optimization available with temporal pointers only is to write on disk large chunks of data at once, not just one frame. An AGAPIA program without temporal pointers should use the `Save` module at each frame to write the image on disk. By using streams and temporal pointers, we can define a new data stream with high capacity that stores multiple images and modify the `Save` module to buffer multiple images and write on disk only when its capacity is full. This can improve the speed of the application.

```
DECLARE_STREAM("ImageBuffer", 33, W*H*BPP, 2);

module main{listen nil}{read nil}
```

```
{
      (Init      #    Init     #    Init      # Init)
      %
      InitTemMem # while_st(true)
                   {
                      ReadFrame # NoiseFilter # SobelFilter # Save
                   }
}{speak nil}{write nil}

module InitTempMem{listen nil}{read nil}
{
ImgInfo ^tx in "ImageBuffer";
}{speak nil}{write nil}

module Init{listen nil}{read nil}
{ i = 0;
}{speak nil}{write i}

module ReadFrame{listen nil}{read i}
{
  copy(&(tx+i), GetFramePixels(), W*H*BPP);
  speak tx;
  i = i+1;
}{listen nil}{write i}

module NoiseFilter{listen nil}{read i}
{
  listen tx;
  for (int i = 0; i < H-1; i++)
     for(int j = 0; j < W-1; j++)
     {
       // Compute the average of the four neighbour pixels
       const ImgInfo&  img =  &(tx+i);
       int val = img.pixel(i,j) + img.pixels(i,j+1) +
                 img.pixels(i+1,j) + img.pixels(i+1,j+1)
       img.setpixel(i,j) = val*0.25f;
     }
  speak tx;
  i = i+1;
}{speak nil}{write i}
```

```
module Sobel{listen nil}{read i}
{
 listen tx;
 // Compute sobel filter here..
 speak tx;
 i = i+1;
}{speak nil}{write i}

module Save{listen nil}{read i}
{
  listen tx;
  WriteToDisk(&(tx+i), W*H*BPP);
  i = i+1;
}{speak nil}{write i}
```

5.3.4 Technical comments for an implementation

A complex issue in implementing the temporal pointers and interaction streams is how to store the global data representing the streams. The data structure need to be distributed through several machines. Some solutions such as HADOOP ([38]) or X10 ([21]) could be used to speedup the implementation of this by using their shared distributed memory data structures. The code for temporal pointers usage and data stream declarations could be expressed directly in C/C++ code, so for maintaining a peek performance the `AgapiaToCPP` API and `ModulesAnalyzer` need to be modified to express all the internal `listen`, `speak`, declarations and usages of temporal pointers as C/C++ code.

Another thing to consider is something similar to branch prediction for modern processors. When reading types from the data stream that represents future we have two options: either block the execution and sleep until that moment of time becomes current or guess / try combinations of values and move the execution forward with separate branches in parallel. When that moment of time becomes current remove the unnecessary branches. This prediction is possible for boolean types and needs to be analyzed for other data types too.

5.4 A comparison between X10 and AGAPIA

X10 is a new programming language designed specifically for parallel computing using the partitioned global address space mode ([21], [23], [85], [41], [77], [61], [96], [66], [34], [86], [60]). A computation can be divided among a set of places each one holding data and activities that operates on those data. Other important features includes structured/unstructured parallelism and globally distributed arrays.

5.4.1 A short introduction to X10 programming language

Async

The fundamental concurrency construct in X10 is `async`. The execution of `async S` is actually a creation of an activity - a very light-weight thread of execution that runs in parallel with the current activity and has access to the same heap of objects as the current activity. An example which multiplies each element in an array with 2 is given below:

```
// Initialize the i'th element to i
val a:Rail[Long]=new Rail[Long](N,(i:Long)=>i);
// Double every element of the array in Asynchronously
for (i in 0..(a.size-1)) async
     (a(i))*=2;
```

Finish

An activity can spawn further activities which is a common practice in recursive programs. X10 distinguishes between local and global termination. Activity `S` has locally terminated if `S` has terminated. Activity `S` has globally terminated if it has locally terminated and any activities spawned during its execution have themselves (recursively) globally terminated. `finish` is a construct that converts global termination to local termination. In the code snippet below `Fib` is a class which contains a function `Fib::fib` for computing the `Fibonacci` of a number recursively. One of the two branches is computed asynchronously with the other one (note the `async` keyword). To compute the final results correctly it must wait for the asynchronous operation to complete so a `final` construct needs to be added.

```
class Fib
{
 var n:Int = 0;
 def this(n:Int} { this.n = n;}
 def fib()
  {
   If (n <= 2)
   {
      n = 1;
      return;
  }
  val f1 = new Fib(n-1);
  val f2 = new Fib(n-2);
  finish{
    async f1.fib();
    f2.fib();
   }
  n = f1.n + f2.n;
};
```

Atomic

X10 supports conditional atomic statements: `while(condition) atomic S`. This statement executes atomically `S` when the condition (also called the guard of `S`) is valid. If the condition is not valid then it blocks until it becomes valid. Some restrictions are imposed over the condition's semantic: should not spawn any activities, its evaluation should be local without side effects. For simplicity, if the condition is always valid - `while(true) atomic S` - then it can be simple written as: `atomic S`. Below is an example which creates a histogram (array `b` from the values in array `a`):

```
def hist(a:Rail[Int], b:Rail[Int])
{
   finish for (i in 0..(a.size-1)) async
   {
      val bin = a(i) % b.size;
      atomic b(bin)++;
   }
}
```

Places

A central innovation of X10 is the notion of `places`. They allows programmers to explicitly deals with notions of locality. Locality issues arise in three primary ways:

- If we write a program to deal with enormous amounts of data we may not have enough main memory on a single node to store all this data. The idea behind is to use `distributed shared memory` (`DSM`), but by doing this the programmer will lose the ability to estimate the latency of memory accesses.

- Heterogeneity of the nodes is another concern. In a cluster there could be specialized nodes very good at particular kinds of computations such as Cell Broadband Engine or GPGPU.

- Multiple cores may share important resources such as L1 and L2 cache. Running instructions in parallel on these cores can pollute the cache lines of the other. In some cases is it ideal to bind activities to particular cores which do not shares cache memory.

A `place` in X10 is a collection of data and activities that operate on the data. A program written in X10 runs on a fixed number of places and the binding of places to hardware resources (e.g., nodes in a cluster) is provided externally by a configuration file.

The primary construct for exposing places to the programmer is : `at (p) S`, where `S` is a statement. An activity executing this expression suspends execution in the current place. The object graph `G` containing all the variables `V` used in `S` is serialized and transmitted to place `p`, deserialized (creating `G'` isomorphic to `G`), an environment is created with variables `V`, and finally `S` is executed at `p` in this environment. After `S` terminates locally, computation resumes after `at (p) S` in the original location. The object graph is not automatically transferred back to the originating place which means that any updates made to objects copied by an `at` will not be reflected in the original object graph. The `at` operation is also called a place-shifting operation because communication may potentially happen across the network. Using the `GlobalRef` construct programmers can use global partitioned objects.

More details, examples and how X10 programs compiles and executes can be found at their website - http://x10-lang.org/.

5.4.2 AGAPIA versus X10

The most important difference between AGAPIA and X10 is the programming style used. X10 is an imperative programming language with constructs for structured and unstructured parallelism while AGAPIA is a combination of imperative programming (specially the code inside atomic modules) with data flow programming (the code connecting modules). The main advantage of X10 over AGAPIA is that it is more complete when it comes to scheduling and configuration of execution. At this level, AGAPIA's compiler is still in its early phases. However, there are a few distinguished advantages that AGAPIA has over X10 from a conceptual level:

- In X10 programmers specify concurrent work using two statements: `async` and `finish`. The runtime is not responsible for finding more concurrent work that was originally specified. In AGAPIA, things are different since users specify just the modules and their interaction. The concurrent tasks are represented by the atomic modules and runtime is trying to maximize the potential parallelism automatically by identifying which modules are ready for execution and making them ready as soon as possible. This is an important advantage since users have less responsibilities for maximizing parallelism.

- In X10 the object graph is serialized even when it is not needed for example when running an `at` in the same place. This is generally bad if the object graph defines large chunks of data because the programmers can't control its useless copy. AGAPIA has the `buffer` data type which can greatly improve performance in this case by copying the data inside only when modules are executed on different nodes.

- When using a place shifting operation in X10 (via `at` construction) it copies an entire graph of properties (and an entire class if called from a function inside of a class). This can imply a bigger overhead in analyzing the code and data transfer, while AGAPIA model is clearer: transfer just the input parameters defined by the user. It is the programmer's responsibility to control the type and data size of parameters.

5.4.3 Using X10 as a backend for AGAPIA

It is very difficult, if not impossible to translate an AGAPIA program to an equivalent X10 program with the same semantics. The main difficulty is

supporting recursion and input/output flow. However, X10 can be used in the AGAPIA runtime instead of using low level libraries for communication and parallelism (could replace the MPI and Intel TBB). We still need the `AST` loaded when executing a program plus the Input/Output flow component inside AGAPIA runtime (Figure 3.5). The main benefit of doing this is that AGAPIA can use the tools, schedulers, distributed shared memory, and the notion of `places` from X10.

The temporal pointers can be successfully implemented by using the distributed shared memory concept that X10 provides. The tools for execution configuration can also be used to make AGAPIA more user-friendly. Using its task-stealing schedulers could improve the performance and avoid memory problems. The current scheduler of AGAPIA holds in the master node all tasks and sends them further when an worker becomes idle. This is not very efficient because if the tasks are light then it will spent too much time on handshaking and if the tasks are huge in terms of memory consumption then master node can't hold all the tasks in memory. The `async` construct can be used as a base for executing atomic modules in parallel. The deployment macro for fixing a module execution and input/output flow coordination to a specified node can be easily solved by using the notion of `places`. When a module's code should be executed on a specified node, the backend could use: `at (node) { module code }`.

5.5 Conclusion

The first part of this chapter presented two of the applications types that are suitable for AGAPIA: dataflow programming and wavefront pattern. For both categories of applications a specific problem was chosen and implemented. The implementations shows that AGAPIA can provide easier and less development time, modularity and some other advantages, with minimal performance lose over.

The second part of the chapter presented technical descriptions for extending AGAPIA's capabilities. Spatio-temporal data streaming and temporal pointers can provide a huge simplication for some categories of applications due to the usage of global data memory: less variables in the interfaces, simpler composition and dataflow graph, and automatic control for applications that need to have strict timing for some of their operations. The other

possible extension presented is the usage of X10. The most interesting things to integrate from this language are the schedulers behind and the configuration mechanism that allows locality. However, when deciding to use X10 for AGAPIA runtime an important point is to analyse correctly the border between what should be used from X10 and what should be kept from the current code. Using too much from it can cause lower control over what is happening behind scenes and this would probably hurt the performance.

Chapter 6

Conclusions and Future work

This thesis presented how to write and execute programs in AGAPIA, its current version of compilation and runtime, some examples of patterns and real problems implemented to analyze its features, and possible ideas for extending the tools for increasing the productivity.

AGAPIA could be an interesting choice when deciding to implement an application for parallel computing. Its main features like transparent communication model, modularity, same source code that can be used for both shared and distributed model makes applications faster to develop, easier to understand and maintain existing code. And all these features comes with a very low performance cost, as the examples presented in Chapter 5 show. The most interesting parallel model where to apply AGAPIA is the distributed memory one where the current solutions available, such as MPI, are less efficient than AGAPIA when it comes to development time, understanding, and code maintenance. X10 has some common feature with AGAPIA, like the transparent communication model. However, as Chapter 6 presented, there are still a few core advantages of using AGAPIA over X10.

There are many things that can be improved in the current version of the AGAPIA compiler to make it more powerful. In the tools category, an important technical improvement would be to create a visual tool for designing modules, interfaces, and links between them. This kind of tool for AGAPIA has the potential to improve the development time especially for applications with a complex data flow, because users could automatically check if a program is well defined or not and simulate visually the input/output flow.

The runtime of AGAPIA can also be improved. First, a task-stealing scheduler and one that is capable of running atomic modules on the GPU

(using macro @GPU) could be valuable add-ons. Another possible optimization to improve the runtime execution of AGAPIA programs is to do some preprocessing inside the AST before starting the execution. For example, in the currrent version, each output data moves through a chain until it gets to an end (usually an atomic module). We can cut this time lost on moving data through links by connecting an output node directly to the final input node.

More macros for deployment and an implementation of the temporal pointers could enlarge the categories of applications where AGAPIA can be used. A possible way to attract more users to use this language is to allow them to call AGAPIA modules directly from C/C++ code or other common programming languages. To do this, we need to build the AGAPIA compiler as a dynamic link library.

From the parsing and error reporting process point of view, there could be done some optimizations to be more user-friendly (more details on some restrictions and small but important things to do are on the website [101]). Finally, we must study other types of applications that can benefit by using AGAPIA language and develop rich sets of ready to use modules for a larger range of applications.

Bibliography

[1] Ahmad I. and Kwok Y.-K., On exploiting task duplication in parallel program scheduling, IEEE Trans, Parallel Distributed Systems, 9(9):872-892, 1998.

[2] Akay Oguz, Erciyes Kayhan, A Dynamic Load Balancing model for a distributed system, Mathematical & Computational Applications, Vol.8, No. 3, pp 353-360, 2003.

[3] Alsabti K., Ranka S., and Singh V., Clouds: A decision tree classier for large datasets, Technical report, University of Florida, 1998.

[4] Aho Alfred V., Sethi Ravi, Ullman Jeffrey D., Compilers, Principles, Techniques, and Tools, Columbia University, January 1, 1986.

[5] Alves C. E. R., Caceres E. N., Dehne F., and Song S. W., A parallel wavefront algorithm for efficient biological sequence comparison, Proceeding ICCSA'03 Proceedings of the 2003 international conference on Computational science and its applications.

[6] Amestoy P. R., Guermouche A., L'Excellent J.-Y., and Pralet S., Hybrid scheduling for the parallel solution of linear systems, Parallel Computing, 32(2):136-156, 2006.

[7] Armstrong Timothy G., Wozniak Justin M., Wilde Michael, Foster Ian T., Compiler techniques for massively scalable implicit task parallelism, Proceedings SC 2014.

[8] Aubry Pascal, Beaucamps Pierre-Edouard, Blanc Frederic, Bodin Bruno, Carpov Sergiu, Cudennec Loic, David Vincent, Dore Philippe, Dubrulle Paul, Dupont de Dinechin Benoit, Galea Francois, Goubier Thierry, Harrand Michel, Jones Samuel, Lesage Jean-Denis, Louise Stephane,

Morey Chaisemartin Nicolas, Nguyen Thanh Hai, Raynaud Xavier, Sirdey Renaud, Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor, Proceedings of ICCS 2013: 1624-1633.

[9] Augonnet C., Thibault S., Namyst R., and Wacrenier P.-A., Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience, 23(2):187-198, 2011.

[10] Banu-Demergian Iulia Teodora, Paduraru Ciprian, Stefanescu Gheorghe, A New Representation of Two-Dimensional Patterns and Applications to Interactive Programming, FSEN 2013, Fundamentals of Software Engineering, LNCS 2013 pp 183-198.

[11] James Reinders, Intel Threading Building Blocks, Outfitting C++ for Multi-core Processor parallelism, O'Reilly, 2007.

[12] Balaji, P., Buntinas, D., Goodell, D., Gropp, W.D., Thakur, R., Toward Efficient Support for Multithreaded MPI Communication, EuroPVM/MPI 2008, LNCS, Springer vol. 5205, pp. 120-129.

[13] Ben-Ari Mordechai (Moti), Principles of Concurrent and Distributed Programming, second edition, Addison-Wesley, 2006.

[14] Bender A. M., and Rabin M. O., Online scheduling of parallel programs on heterogeneous systems with applications to Cilk, Theory of Computing Systems, Special Issue on SPAA 2000, volume 35, pages 289-304, 2002.

[15] Bender A. M. and Rabin M. O., Scheduling cilk multithreaded parallel programs on processors of different speeds, 12th annual ACM Symposium on Parallel Algorithms and Architectures - SPAA, pages 13-21, Bar Harbor, Maine, USA, 2000.

[16] Bergman D., Cire A. A., Sabharwal A., Samulowitz H., Saraswat V., and Van Hoeve W.-J.. Parallel Combinatorial Optimization with Decision Diagrams, Proceedings of CPAIOR, Springer, 2014.

[17] Blumofe R. D., Joerg C. F., Kuszmaul B. C., Leiserson C. E., Randall K.H., and Zhou Y., Cilk: An Efficient Multithreaded Runtime System,

Proceedings of 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95, pages 207-216, Santa Barbara, California, July 1995, MIT.

[18] Bosilca G., Bouteiller A., Danalis A., Herault T., Lemarinier P., and Dongarra J.. DAGuE: A generic distributed dag engine for high performance computing. Parallel Computing, 38(1):37-51, 2012.

[19] A. Bouch, A. Kuchinsky and N. Bhatti, Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service, Technical Report, Internet Systems and Applications Laboratory, HP Laboratories Palo Alto, 2000.

[20] Cardellini V., Casalicchio E., Colajanni M. and Yu P.S., The State of the Art in Locally Distributed Web-Server Systems, ACM Computing Surveys, 34(2):263–311, 2002.

[21] Charles Philippe, Grothoff Christian,Saraswat Vijay, Donawa Christopher, Kielstra Allan, Ebcioglu Kemal, Von Praun Christoph, Sarkar Vivek, X10: an object-oriented approach to non-uniform cluster computing, Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications, 2005.

[22] Cheng Long, Kotoulas Spyros,Ward Tomas E., and Theodoropoulos Georgios, A two-tier index architecture for fast processing large RDF data over distributed memory, In Proceedings of the 25th ACM conference on Hypertext and social media (HT '14). September, 2014.

[23] Cunningham David,Grove David,Herta Benjamin,Iyengar Arun, Kawachiya Kiyokuni ,Murata Hiroki, Saraswat Vijay, Takeuchi Mikio, Tardieu Olivier, Resilient X10: Efficient failure-aware programming, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14), Feb 2014.

[24] Cho Kwang Soo, Joo Un Gi,Sub Lee Heyung, Tae Kim Bong, and Don Lee Won, Efficient Load Balancing Algorithms for a Resilient Packet Ring Using Artificial Bee Colony, Applications of Evolutionary Computation, Lecture Notes in Computer Science, Volume 6025, 2010, pp 61-70.

[25] Dios Antonio J., Asenjo Rafael, Navarro Angeles, Corbera Francisco and Zapata Emilio L., Evaluation of the Task Programming Model in the

Parallelization of Wavefront Problems, High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on

[26] Duran Alejandro, Corbalan Julita, and Ayguade Eduard. Evaluation of OpenMP Task Scheduling Strategies, Lecture Notes in Computer Science: Proceedings of the 4th International Workshop on OpenMP, volume 5004, pages 100-110. Springer, Springer, May 2008.

[27] Donald, S. and Le Vie, Jr. (2000), Understanding Data Flow Diagram, Proceedings of the 47th annual conference on Society for Technical Communication, Texas: Integrated Concepts, Inc.

[28] Dragoi, C., Stefanescu, G.: AGAPIA v0.1: A Programming Language for Interactive Systems and its Typing System, FInCo, Electr. Notes Theor. Comput. Sci. 203(3): 69-94 (2008).

[29] Erich Gamma, Helm Richard, Johnson Ralph, Vlissides John M., Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley, 2000.

[30] Fizzano Perry and Stein Clifford, Scheduling on a Ring with Unit Capacity Links, Technical Report, Dartmouth College Hanover, 1994.

[31] Foster Ian. T., Fidler M., Rpy A., Sander V., and Winkler L., End-to-end quality of service for high-end applications, Computer Communications, 27(14):1375-1388, 2004.

[32] Gao G. R., Sterling T. L., Stevens R., Hereld M., and Zhu W., Parallex: A study of a new parallel computation model, Proceedings of IPDPS, pages 1-6, 2007.

[33] Gehrke Johannes E. , Greg Plaxton C. and Rajaraman Rajmohan, Rapid Convergence of a Local Load Balancing Algorithm for Asyncronous Rings, Distributed Algorithms, Lecture Notes in Computer Science, Volume 1320, 1997, pp 81-95.

[34] Gligoric Milos, Mehlitz Peter C., and Marinov Darko, X10X: Model Checking a New Programming Language with an "Old" Model Checker, ICST 2012 - 5th International Conference on Software Testing, Verification, and Validation, Montreal, Canada, April 2012.

[35] Gropp William, Huss-Lederman Steven, Lumsdaine Andrew, Lusk Ewing, Nitzberg Bill, Saphir William, and Snir Marc, MPI-The Complete Reference: Volume 2, The MPI-2 Extensions, MIT Press, Cambridge, MA, 1998.

[36] Gropp William, Lusk Ewing, Skjellum Anthony, Using MPI, 2nd Edition: Portable Parallel Programming with the Message Passing Interface, Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series, 1999.

[37] Gropp W., Lusk E., and Thakur R., Using MPI-2 Advanced Features of the Message-Passing Interface, The MIT Press, Cambridge, Massachusetts, USA, 1999.

[38] White, T Hadoop: The Definitive Guide second edition, Oxford: O'Reilly Media (2010).

[39] Hall L. A.l, Schulz A. S., Shmoys D. B., and Wein J., Scheduling to minimize average completion time: Off-line and on-line approximation algorithms, Mathematics of Operations Research, 22:513-544, August 1997.

[40] Hermann J., Marchal L., and Robert Y., Model and complexity results for tree traversals on hybrid platforms, Proceedings of Euro-Par 2013 - Parallel Processing, LNCS Springer, 2013.

[41] Houngkaew Charuwat and Suzumura Toyotaro, X10-Based Distributed and Parallel Betweenness Centrality and Its Application to Social Analytics, HiPC 2013 (IEEE International Conference on High Performance Computing), India, December, 2013.

[42] Imai Shigeru, Klockowski Richard, and Varela Carlos A. Self-Healing Spatio-Temporal Data Streams Using Error Signatures, In 2nd International Conference on Big Data Science and Engineering (BDSE 2013), Sydney, Australia, December 2013.

[43] Imai Shigeru and Varela Carlos A., A Programming Model for Spatio-Temporal Data Streaming Applications, In Dynamic Data-Driven Application Systems (DDDAS 2012), Omaha, Nebraska, pages 1139-1148, June 2012.

[44] Imai Shigeru and Varela Carlos A., Programming Spatio-Temporal Data Streaming Applications with High-Level Specifications, In 3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012, Redondo Beach, California, USA, November 2012.

[45] Intel Corporation, Intel Parallel Advisor, http://software.intel.com/en-us/articles/intel-parallel-advisor/

[46] Kahn Gilles, The Semantics of Simple Language for Parallel Programming. IFIP Congress 1974: 471-475

[47] Klockowski Richard S., Imai Shigeru, Rice Colin, and Varela Carlos A. Autonomous Data Error Detection and Recovery in Streaming Applications, In Proceedings of the International Conference on Computational Science (ICCS 2013). Dynamic Data-Driven Application Systems (DDDAS 2013) Workshop, pages 2036-2045, May 2013.

[48] Krieder Scott J., Wozniak Justin M., Armstrong Timothy, Wilde Michael, Katz Daniel S., Grimmer Benjamin, Foster Ian T., Raicu Ioan, Design and evaluation of the GeMTC framework for GPU-enabled many-task computing, Proceedings of HPDC 2014.

[49] Kamal, H., Mirtaheri, S.M., Wagner, A, Scalability of communicators and groups in MPI, Proceedings of the 19th ACM Intl. Symposium on High Performance Distributed Computing, HPDC 2010, pp. 264-275. ACM, New York (2010).

[50] Kamal, H., Wagner, A, FG-MPI: Fine-Grain MPI for multicore and clusters, in 11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) held in conjunction with IPDPS-24, pp. 1-8 (April 2010).

[51] Kumar, Blackburn, Grove, Friendly Barriers: Efficient Work-Stealing With Return Barriers, Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2014). Feb, 2014.

[52] Kumar Vivek,Frampton Daniel, Blackburn Stephen M., Grove David, Tardieu Olivier, Work-Stealing Without The Baggage, Proceedings of the

25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications (OOPSLA'12). October 2012.

[53] Larus J. R., Loop-level parallelism in numeric and symbolic programs, IEEE Trans, Parallel Distributed Systems, 4(7), 1993.

[54] Levine John, Flex & Bison, O'Reilly Media, 2009, p. 9.

[55] Lin Hong, Kemp Jeremy, Gilbert Padraic, Computing Gamma Calculus on Computer Cluster, International Journal of Techonology Diffusion, Volume 1, Issue 4, 2010.

[56] Ltaief H., Luszczek P., and Tomov S., Numerical linear algebra on emerging architectures: The plasma and magma projects, Journal of Physics: Conference Series, 180(1):012037, 2009.

[57] Lee Ben, Hurson Ali R., Issues in Dataflow Computing, Advances in Computers 01/1993; 37:285-333

[58] Lewis E Christopher and Snyder Lawrence. Pipelining wavefront computations: Experiences and performance, Fifth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS,1999).

[59] McCool Michael, Reinders James, Robison Arch, Structured Parallel programming: Patterns for Efficient Computation, Morgan Kaufmann July 9, 1012.

[60] Milthorpe Josh, Ganesh V., Rendell Alistair P. and Grove David, X10 as a Parallel Language for Scientific Computation: Practice and Experience, Proceedings of IEEE International Parallel and Distributed Processing Symposium, May 2011.

[61] Milthorpe Josh, Huber Thomas, and Rendell Alistair, PGAS-FMM: Implementing a distributed fast multipole method using the X10 programming language, Concurrency and Computation: Practice and Experience, 26(3), pp 712-727, 2014.

[62] Moerkotte Guido , Neumann Thomas, Dynamic programming strikes back, SIGMOD '08 Proceedings of the 2008 ACM SIGMOD international conference on Management of data Pages 539-552.

[63] Meyer, L., Scheftner, D., Voeckler, J., Mattoso, M., Wilde, M. and Foster, I., An Opportunistic Algorithm for Scheduling Workflows on Grids, Proceedings of VECPAR'06, Rio De Janiero, 2006.

[64] Mattson Timothy G., Sanders Beverly A., Massingill Berna L., Patterns for Parallel Programming, Addison-Wesley, September 25, 2004.

[65] Muller Stefan and Chong Stephen, Towards a Practical Secure Concurrent Language, Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications (OOPSLA'12). October 2012.

[66] Ogata Hidefumi, Dayarathna Miyuru, and Suzumura Toyotaro, Towards highly scalable X10 based spectral clustering, Proceedings of the 19th International Conference on High Performance Computing (HiPC), 2012.

[67] Paduraru Ciprian I., Dataflow Programming using AGAPIA, Proceedings of Parallel and Distributed Computing (ISPDC), 13th International Symposium, IEEE, 2014.

[68] Paduraru Ciprian I., A New Online Load Balancing Algorithm in Distributed Systems, Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, IEEE, Timisoara, Romania, 2012.

[69] Paduraru Ciprian I., An online load balancing algorithm for a hierarchical ring topology, International Journal on Computers, Communications & Control, vol. 9, no. 6, 2014.

[70] Paduraru Ciprian I., A Greedy Algorithm for Load Balancing Jobs with Deadlines in a Distributed Network, International Journal of Advanced Computer Science and Applications, vol. 5, no. 2, 2014.

[71] Paduraru Ciprian I., Distributed programming using AGAPIA, International Journal of Advanced Computer Science and Applications, vol. 5, no. 2, 2014.

[72] Paduraru Ciprian I., Ciucu Ana-Maria, Stefanescu Gheorghe, Internal technical report, Softwin, 2012.

[73] Palis M. A., Liou J.-C., and Wei D. S. L., Task clustering and scheduling for distributed memory parallel architectures, IEEE Trans. Parallel Distributed Systems, 7(1):46-55, 1996.

[74] Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana, IL. The Charm++ Programming Language Manual.

[75] Popa Alexandru, Sofronia Alexandru, Stefanescu Gheorghe, High-level Structured Interactive Programs with Registers and Voices, Journal of Universal Computer Science, vol. 13, no 11 (2007) 1722-1754.

[76] Peters A K (2006), Markus Hadwiger, Joe Kniss, Christof Rezk-Salama, Daniel Weiskopf, Real-time volume graphics, pp. 112-114, A K Peters, 2006.

[77] Paudel Jeeva, Tardieu Olivier and Amaral Jose Nelson, Hybrid Parallel Task Placement in X10, ACM SIGPLAN 2013 X10 Workshop, June 2013.

[78] Ravindran G. and Stumm M., Hierarchical Ring Topologies and the Effect of their Bisection Bandwidth Constraintsm, Proceedings Intl. Conf.Parallel Processing, pp.I/51-55, 1995.

[79] Sharma S., Singh S. and Sharma M., Performance Analysis of Load Balance Algorithms, World Academy of Science, Engineering and Technology, 28:269–272, 2008.

[80] Sousa Tiago Boldt Dataflow Programming Concept, Languages and Applications , Proceedings of Doctoral Symposium on Informatics Engineering, 2012.

[81] Suzumura Toyotaro and Kanezashi Hiroki, Highly Scalable X10-based Agent Simulation Platform and its Application to Large-scale Traffic Simulation, 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications, Dublin, Ireland, 2012/10.

[82] Suzumura Toyotaro, Takeuchi Mikio, Kato Sei, Imamichi Takashi, Kanezashi Hiroki, Ide Tsuyoshi, and Onodera Tamiya, X10-based Massive Parallel Large-Scale Traffic Flow Simulation, ACM SIGPLAN 2012 X10 Workshop, June 2012.

[83] Stivala Alex, Stuckey Peter J., Garcia de la Banda Maria, Hermenegildo Manuel, Wirtha Anthony, Lock-free Parallel Dynamic Programming, Journal of Parallel and Distributed Computing, Volume 70, Issue 8, August 2010, Pages 839-848.

[84] Tanenbaum Andrew S., Modern Operating Systems (3rd Edition), Prentice Hall, December 2007.

[85] Tardieu Olivier, Herta Benjamin, Cunningham David,Grove David, Kambadur Prabhanjan, Saraswat Vijay, Shinnar Avraham, Takeuchi Mikio, Vaziri Mandana, X10 and APGAS at Petascale, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14), Feb 2014.

[86] Tardieu Olivier, Bo Lin Hai, and Wang Haichuan, A work-stealing scheduler for X10's task parallelism with suspension, PPoPP'12 - 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Feb 2012.

[87] Wadge W. and Ashcroft E.A., Lucid, the data flow programming language, Academic Press, 1985.

[88] Wilde Michael, Hategan Mihael, Wozniak Justin M., Clifford Ben, Katz Daniel S., Foster Ian, Swift: A language for distributed parallel scripting, Parallel Computing 2011, 37(9) 633-652, 2013.

[89] Wozniak Justin M., Armstrong Timothy G., Wilde Michael, Katz Daniel S., Lusk Ewing, Foster Ian T., Swift/T: scalable data flow programming for many-task applications, PPOPP, page 309-310. ACM, (2013).

[90] Wozniak Justin M., Wilde Michael, Foster Ian T., Language features for scalable distributed-memory dataflow computing, Proceedings Data-flow Execution Models for Extreme-scale Computing at PACT 2014.

[91] Wozniak Justin M., Armstrog Timothy G., Maheshwari Ketan, Lusk Ewing L., Katz Daniel S., Wilde Michael, Foster Ian T., Turbine: A distributed-memory dataflow engine for high performance many-task applications, Fundamenta Informaticae 128(3) 2013.

[92] Wozniak Justin M., Peterka Tom , Armstrong Timothy G., Dinan James, Lusk Ewing, Wilde Michael, Foster Ian T., Dataflow coordination of data-parallel tasks via MPI 3.0, Proceedings EuroMPI 2013.

[93] Wozniak Justin M., Armstrong Timothy G., Wilde Michael, Katz Daniel, Lusk Ewing, Foster Ian T., Swift/T: Large-scale application composition via distributed-memory data flow processing, Proceedings CCGrid 2013.

[94] Xu C., Chen X., Dick R., and Mao Z. Cache contention and application performance prediction for multi-core systems, Proceedings of ISPASS 2010.

[95] Xu D., Wu C., and Yew P.-C., On mitigating memory bandwidth contention through bandwidth-aware scheduling, Proceedings of PACT '10, Sep 2010.

[96] Xie Chenning, Hao Zhijun, Chen Haibo. PMAM 2013, X10-FT: Transparent Fault Tolerance for APGAS Language and Runtime, Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycore.

[97] Yan Yonghong, Zhao Jisheng, Guo Yi, Sarkar Vivek, Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement, 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC), October 2009.

[98] Zhang Q., Riska A., Sun W., Smirni E. and Ciardo G., Workload-Aware Load Balancing for Clustered Web Servers, IEEE Transactions on Parallel and Distributed Systems, 16(3):1-15, 2005.

[99] Zhou S. and Ferrari D., An Experimental Study of Load Balancing Performance, Technical Report, Computer Science Division (EECS), University of California, 1987.

[100] Zhuravlev S., Blagodurov S., and Fedorova A., Addressing shared resource contention in multicore processors via scheduling, ASPLOS '10, Mar 2010.

[101] AGAPIA website, https://code.google.com/p/agapia-programming-language/