Master Thesis

Master's Program in Computer Science

A Backend for the Translation of AIR Intermediate Code Into Intel ArBB for the R Parallelization Platform ALCHEMY

Marc Aurel Kiefer

submitted

May 27, 2013

Supervised by

Dr. Frank Padberg

Prof. Dr. Sebastian Hack

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ______________________ ______________________

(Datum/Date) (Unterschrift/Signature)

Contents

1. Introduction **5**

2. Approach **7**
- 2.1. Input for Our Backend Module 7
- 2.2. Code Transformation 8
- 2.3. ArBB Target Code 8
- 2.4. ArBB Backend / Code Execution 9
- 2.5. Output of Our Backend Module 10

3. Related Work **10**
- 3.1. ALCHEMY 10
- 3.2. Analysis Intermediate Representation (AIR) 11
- 3.3. Parallel Skeletons 11
- 3.4. R Parallel Packages 12

4. Requirements Analysis **13**
- 4.1. Type Inference 13
- 4.2. Code Transformation 16
- 4.3. Data Transfer Between Components 19
- 4.4. Skeletons 19
- 4.5. Intel ArBB Runtime 20
- 4.6. Result Collector 20
- 4.7. Error Handling 21

5. Design and Implementation **21**
- 5.1. System Architecture 21
- 5.2. ArBB Backend 22
- 5.3. Code Translation 24
  - 5.3.1. AIRNodeTypeVisitor 24
  - 5.3.2. AIRNodeArBBVisitor 27
- 5.4. R-to-ArBB-Library 30
- 5.5. Skeleton Generator 37
- 5.6. Extension of the Value-/EnvironmentService 38
- 5.7. Validity of Code Transformation 40

6. Evaluation **44**
6.1. Performance . . . 45
6.1.1. Myfun Example . . . 45
6.1.2. Levenshtein Distance . . . 48
6.1.3. Longest Common Subsequence . . . 50
6.1.4. ZMQ Transfer Rates . . . 51
6.1.5. ArBB Scaling . . . 51
6.1.6. Runtime Composition . . . 52
6.2. Worst Case Speedups . . . 53
6.2.1. MAP Skeleton . . . 53
6.2.2. SCAN Skeleton . . . 54
6.2.3. ZIPW Skeleton . . . 54
6.2.4. DOPAR Skeleton . . . 55

7. Conclusion and Outlook **56**

A. Generated ArBB Code Example **59**

B. Evaluation Code Listings **61**

C. Numerical Evaluation Results **64**

D. Sequential Skeleton Implementations **64**

E. AIR/SEXPR Examples **68**

1. Introduction

The R programming language is being widely used by application programmers in statistics and bioinformatics[MD12, MAL+09] where large data sets are processed. R is an interpreted language that currently does not make use of the multiple processors available in modern computers. Hence, processing large data sets is slow, and applications such as string matching or genom sequence analysis often result in computations taking several hours.

ALCHEMY[Mir11] is an experimentation platform for the automatic parallelization analysis of R programs and the execution of parallelized R programs. The approach ALCHEMY takes to process and execute R programs is shown in Figure 1. The platform and several parallelization analysis modules (PAM) have already been developed; yet, ALCHEMY currently lacks a backend for fast parallel execution.

The goal of this thesis is to provide an ALCHEMY backend for fast parallel execution of parallelized R programs. This requires mapping parallel code structures that result from the parallelization analysis of the PAMs to primitives of the target platform. As the target platform, we use Intels C++ based Array Building Blocks (ArBB) [NSL+11]. ArBB provides means to handle parallelization at a high level of abstraction and transparently supports various multicore processors.

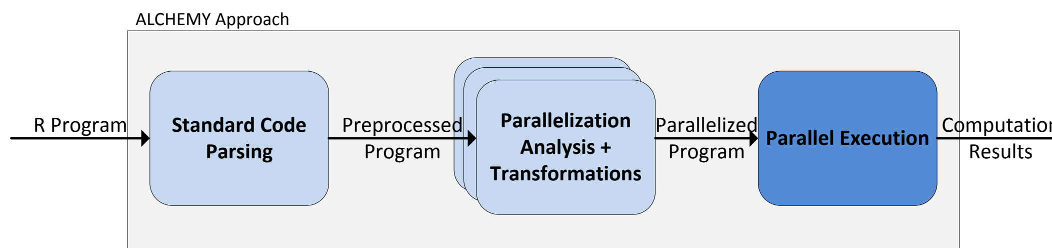

Figure 1: Alchemy approach [Mir11]

Our parallel execution module receives parallelized R programs from the ALCHEMY platform. The input is presented as AIR intermediate code that contains R primitives as well as parallel skeletons (such as MAP or SCAN). The backend module developed in this thesis (see Fig. 2) transforms the AIR input program into ArBB C++ code, and then compiles it to an executable. After completing parallel execution within the ArBB platform, our backend collects the computation results and wraps them into AIR code in order to return them to ALCHEMY.

As part of this thesis, we evaluate the performance of the parallelized R programs.

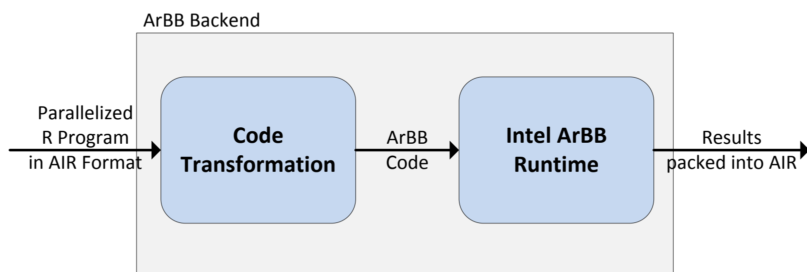

Figure 2: Parallel execution

We assume that the input to our backend is already parallelized. We do not perform parallelization analysis, nor do we aim at improving the parallel structure of the AIR input, as this is the responsibility of the PAMs.

We chose Intel ArBB [NSL+11] as the parallel execution platform as it seems to fit our purpose best. ArBB runs on multicore CPUs and offers native vector data types (similar to R). It already offers some of the parallel skeletons that we need as built-ins. ArBB handles parallel execution automatically for different target hardware, in particular, for a varying number of cores, in its own virtual machine.

We also considered other possible target platforms, such as OpenMP[DM98, ADH+08, KSG09]. OpenMP is widely used, but it does not have vector types and no skeleton support. Some of our parallel patterns could be built manually as library routines (MAP, ZIP), but OpenMP is less suitable for other patterns. F.e., the SCAN skeleton has a specific data-parallel structure, which does not fit well the task parallelism offered by OpenMP. Similarly, OpenCL[Mun11, SGS10] mainly aims at GPU execution, which is not our focus. In addition, OpenCL has a much larger gap to the semantic level of our AIR; it does not offer vector types or skeletons.

In this thesis, we obtained the following results:

- *We built an ALCHEMY backend for the parallel execution of AIR intermediate code on Intel ArBB.* The backend contains modules for type inference, code translation from AIR to C++, and automatic execution including data transfer.

- *We implemented a supporting C++ library for the backend programs.* The library includes ArBB implementations of R primitives, parallel skeleton implementations, and utility classes for communication and the transfer of input and output data.

- *We extended ALCHEMY to accept large amounts of computation results from the*

backend. The transfer of data between the different ALCHEMY components has also been vastly accelerated.

- *We evaluated the performance of the backend and the validity of the code translation.* Our backend achieves large speedups compared to sequential R execution, which is due to a combination of parallel execution and compilation.

Our backend consists of 40 Java classes with ~5.500 lines of code as well as a C++ backend support library and ALCHEMY extension with ~1000 lines of code .

The performance evaluation on an eight-core machine shows speedups of up to a factor of 33 compared to sequential R interpretation for input sizes of 1M elements. This (surprisingly) large speedup is made possible because we do not only execute the R program in parallel but also compile it to C++. With respect to input size, the break-even point for our backend to be faster than the sequential interpretation lies between a few 100K and 3M of input elements in our examples, including some "worst-case" scenarios. Our backend has a reasonable overhead of up to 200 millisec for the code generation (AIR to C++), but requires a significant overhead between 2 and 5 sec for the final compilation on the ArBB platform. Hence, using our backend for small input is not advisable. For very large data sets, our new backend can save substantial amounts of computing time for the end user.

2. Approach

This Chapter presents a high-level overview of our ArBB backend, how the input and output looks like, which large building blocks are involved and how they all work to together to translate AIR code into ArBB code.

2.1. Input for Our Backend Module

Our module receives AIR (analysis intermediate representation) intermediate code as input. It essentially is an abstract syntax tree (AST), which is semantically equivalent to R code, but easier to handle than the R internal representation. Additionally it contains parallel skeleton expressions such as MAP, ZIPW, SCAN and DOPAR. Listing 1 shows the XML representation of a simple AIR program. The program is semantically equivalent to the R statement:

$$sin(c(1,2,3))$$

It applies the *sin* function to a vector with values *1,2,3*. The *sin* function is embedded

in a MAP skeleton to express parallelism. This allows the backend to apply the function to the values of the vector in parallel.

```
1  <AIR>
2    <Program>
3      <SkeletonExpr skeleton="MAP">
4        <func>
5          <BuiltinFunc name="sin"/>
6        </func>
7        <collection>
8          <ConstantExpr>
9            <AIRVector basetype="real">
10             <Data data="1.0,2.0,3.0" length="3"/>
11           </AIRVector>
12         </ConstantExpr>
13       </collection>
14     </SkeletonExpr>
15   </Program>
16 </AIR>
```

Listing 1: MAP Example in AIR XML representation

2.2. Code Transformation

Before the AIR program can be executed, it must be converted into ArBB code for the parallel backend. The code transformation starts with the extraction of type information from the AIR tree, which is the main difficulty of this step. Since R does not have explicit type information in the source code, which is required to create C++ code, this information has to be inferred from the usage of variables and known types of built-in functions. See Chapter 4.2 for more information on how this is done.

After that, R primitives, R built-in base functions, control logic and parallel skeletons (MAP, SCAN, ZIPW, DOPAR) are translated to ArBB source code. Finally, code which handles the transfer of input and output data between ALCHEMY and the ArBB program, is created. To allow this, ALCHEMY offers different services to manipulate R values and environment objects (see Chapter 4.3).

2.3. ArBB Target Code

ArBB code is C++ source code that uses the Intel ArBB library. During code transformation, parallel skeletons which natively exist in ArBB are created from their AIR

counterparts. Some of our skeletons do not exist in ArBB, so they have been implemented in our own library using ArBB primitives. This also applies to R datatypes which are not supported by ArBB.

The ArBB representation of our *sin* example is presented in Listing 2. Since ArBB requires *void* functions for most parallel constructs, we end up with two helper functions which wrap the *sin* function and the call to the MAP skeleton. The *main* function contains the input vector *in* with values *1,2,3* as well as two ArBB vectors *input* and *output*. The input values are introduced to the ArBB virtual machine by *binding* them to a ArBB variable. *arbb::call* starts the virtual machine, which then compiles the function argument using a JIT compiler and executes it. The code for transferring the computation result back to ALCHEMY is omitted. For an example see Chapter 4.3.

```
1  #include <arbb.hpp>
2  void HELPER_sin(arbb::f32 in, arbb::f32& out){
3    out = sin(in);
4  }
5  void CALL_MAP(arbb::dense<arbb::f32> in, arbb::dense<arbb::
     f32>& out){
6    arbb::map(&HELPER_sin)(in, out);
7  }
8  int main(int argc, const char* argv[]){
9    float in[] = {1.0, 2.0, 3.0};
10   arbb::dense<arbb::f32> input;
11   arbb::dense<arbb::f32> output;
12   arbb::bind(input, in, 3);
13   arbb::call(&CALL_MAP)(output, input);
14   //return result to ALCHEMY
15 }
```

Listing 2: MAP example ArBB code

2.4. ArBB Backend / Code Execution

After the AIR program has been transformed into C++ source code, it gets compiled and linked for the ArBB platform. The resulting executable runs on multicore hardware and receives its input data from ALCHEMY. After the computation has finished, it returns its computation results to ALCHEMY. This process is illustrated in Figure 3.

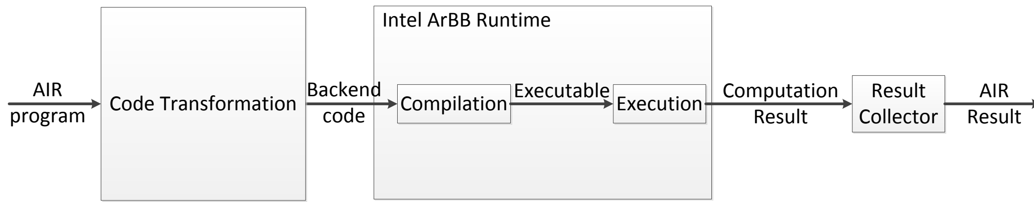

Figure 3: Backend architecture

2.5. Output of Our Backend Module

A result collector receives the computation results after the ArBB program has finished. It creates new AIR code incorporating the results of the computation. The AIR result is handed back to ALCHEMY, where it is presented to the user.

Listing 3 shows the XML representation of an AIR tree containing the result of our *sin* example. The equivalent R code to this AIR tree is

$$c(0.841471, 0.909297, 0.141120)$$

and the user is presented with a vector containing the three computed values.

```
1 <AIR environment-proxy="tcp://127.0.0.1:1985" value-proxy="
     tcp://127.0.0.1:1985" environment-id="28359416">
2   <Program>
3     <ConstantExpr>
4       <AIRVector basetype="real">
5         <Data data="0.841471,0.909297,0.141120" length="3"/>
6       </AIRVector>
7     </ConstantExpr>
8   </Program>
9 </AIR>
```

Listing 3: Example Result in AIR XML Representation

3. Related Work

3.1. ALCHEMY

ALCHEMY [Mir11, PM12] is an experimentation platform for parallelization analysis and parallel execution. It allows to combine different analysis algorithms to get a par-

allel version of an R program. Afterwards, the parallel program can be executed on different parallel backends. The modularity of the system allows to experiment with different configurations of PAMs (e.g., SURE [D+98] for nested loops, or the algorithm of Matszuzaki et al.[KMM+05] for dynamic programming problems) and different parallel backends to find the best performing parallel version of an R program.

End-users might also want to use ALCHEMY in their everyday work with R as a scripting language. Bioinformaticians, for example, working with huge data sets can potentially benefit from the faster execution of R programs with ALCHEMY without changing their R application code.

A typical ALCHEMY scenario looks as follows: A user inputs an R program into the R interpreter. ALCHEMY intercepts the evaluation of the program and translates it into an AIR program. Depending on the configuration, the AIR program is processed by different PAMs and ends up as a parallelized version of the input program. The ArBB backend developed in this thesis now translates it into an ArBB program and runs it on multicore hardware. The computation results are packed into AIR and sent back to the R interpreter. The results are presented to the user as if his R program had been evaluated by R itself.

3.2. Analysis Intermediate Representation (AIR)

The analysis intermediate representation (AIR) is the main construct for communication between modules in ALCHEMY. It is an abstract syntax tree which forms a high-level representation of the program. Although similar to R's internal representation, AIR offers additional skeleton elements to express parallelism and is better suited for processing within ALCHEMY. For a comparison between R's internal SEXPR representation and AIR, see Appendix E.

The input and output of every PAM, as well as the communication language between R and ALCHEMY is AIR. PAMs are also allowed to output multiple AIR programs, i.e. if they can offer different parallelized versions of a program.

Figure 4 again shows our *sin* example in a tree representation of AIR. A MAP skeleton expression applies the *sin* function to a vector with values *1,2,3* in parallel.

3.3. Parallel Skeletons

Parallel programming patterns in AIR are expressed as parallel skeletons. They represent solutions to common parallel problems, where ordering and synchronization details are implicitly defined by the pattern. A comprehensive overview of well-known skeletons can

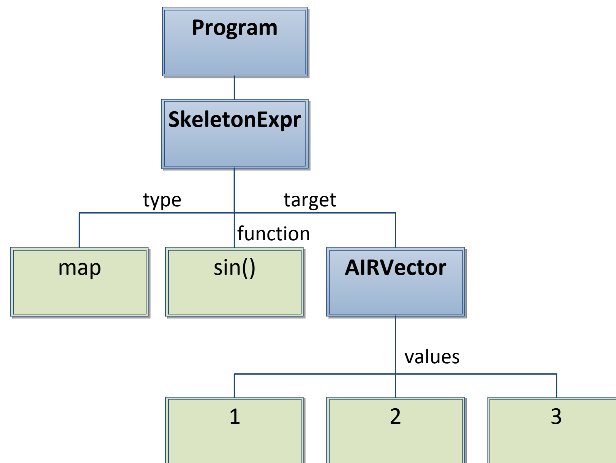

Figure 4: Simple AIR example with parallel skeleton

be found in [Wik11]. Although AIR can easily be extended to support other skeletons if new PAMs require them, these are the ones implemented we currently support:

MAP A function is applied element-wise to a vector in parallel.[Col91]

SCAN Also known as *fold* or *prefix sum*. Computes a vector, where each element is the sum of all preceding elements of the input vector up to the corresponding position. Instead of the + operator, other functions can be used as long as they are associative (required for parallel execution).[McC09, SLO06, Ble90, SHZO07]

ZIPW Similar to MAP, but instead of one input vector ZIPW has two. A function combines an element of each input vector to a single element in the output vector.[KMM+05]

DOPAR Parallel execution of a loop without data dependencies.[D+98]

The above skeletons have been defined in AIR and implemented for the ArBB backend as part of this thesis.

3.4. R Parallel Packages

There already are several solutions which add parallelism to R [SME+09]. Most of them spawn additional R processes to distribute the work on multiple processors.

The *R multicore* package [Urb11] offers a parallel version of the R `lapply` function, which corresponds to our MAP skeleton. It applies a function element-wise to a vector

by splitting up the vector and distributing it to multiple R processes. This approach is also used in an earlier existing R/ALCHEMY backend and will be used for evaluation comparisons.

The *pR* package [MLS07] analyses data dependencies in an R program to distribute independent code parts to different machines in the network via MPI. *pR* is limited to function calls and loops as parallelization targets.

4. Requirements Analysis

This Chapter highlights the architectural relevant pieces in terms of an object-oriented analysis. Figure 5 shows the collaboration between the three main components involved in an ALCHEMY session with our backend. The R Interpreter takes an input program from the user, translates it into AIR and transfers it to ALCHEMY for processing. ALCHEMY runs the AIR through different PAMs until it arrives at our `ArBBBackend`, which translates it into an ArBB program and hands it to the Intel ArBB runtime for execution. The ArBB program fetches input data, does the computation in parallel and transfers the computation results back. Finally the AIR result is sent back to R/ALCHEMY and then presented to the user.

4.1. Type Inference

Before an AIR program can be executed on our parallel backend, it needs to be translated into C++ source code using the ArBB library. The main problem here is the lack of type information coming from the R source code, which is essential for a transformation to C++. At different stages of the ArBB code creation process type information is required, but all of the scenarios are handled by the following four cases:

Variable definition: Whenever a variable is defined, its type is required. In R they mostly come indirectly from variable assignments.

Variable transmission: Fetching data from R or transferring computation results back also requires explicit type information.

Function parameters: User-defined function parameters need to be typed in C++.

Function return types: This also applies to return values.

The main idea of the type inference mechanism is that it all breaks down to R base functions or fixed values. That means the leaves of the AIR tree are either base types

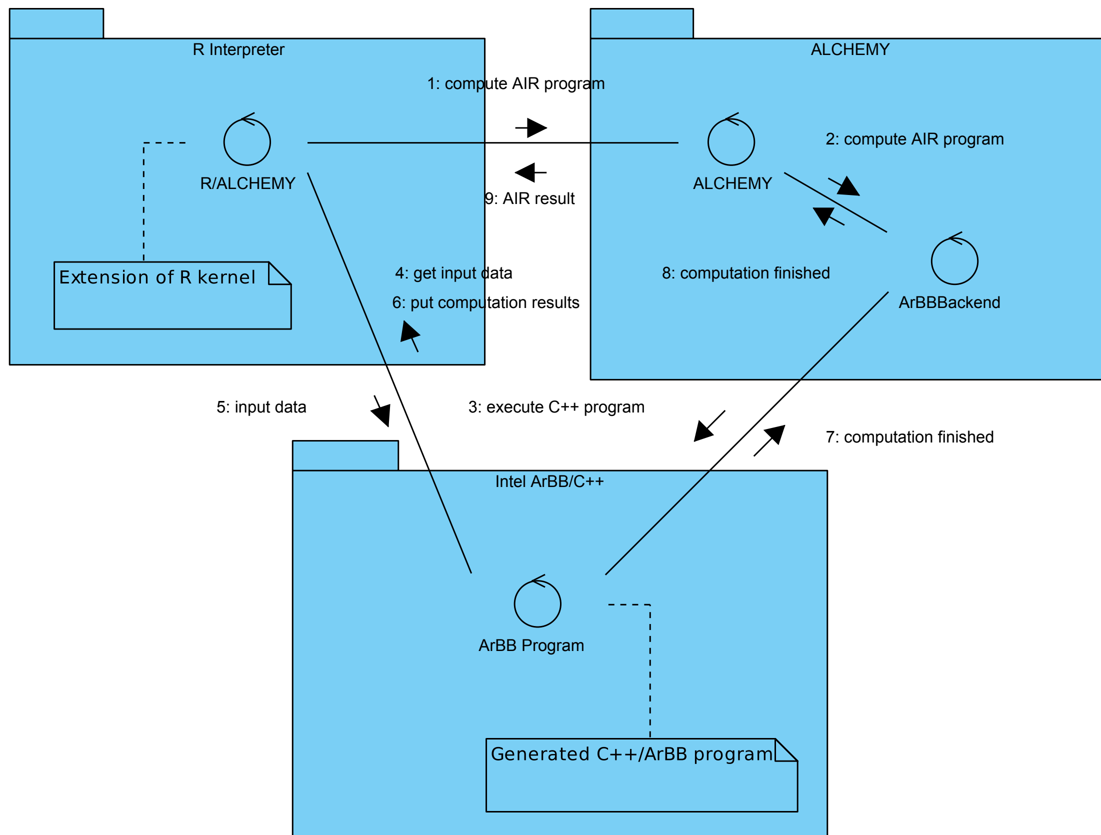

Figure 5: Collaboration between ALCHEMY components

or function calls to known R base functions. A traversal scheme similar to symbolic execution is used to build up a data structure containing all the type information required to do the code transformation. An (easily extensible) database containing all type information of R's built-in functions has been created and it plays an important role in the type inference process. Depending on the situation, different mechanisms are used to determine a type, some of which take into account the context in which the variable is used (x is the variable whose type is determined):

$x \leftarrow 3$ The most simple case, where the right-hand side of the assignment has a fixed type.

$x \leftarrow sin(3)$ Since *sin* is a built-in function, its return type is stored in the database and this determines the type of x.

$sin(x)$ The parameter type of *sin* is also in the database, so again the type of x is set.

The following list describes how each of the AIR primitives are handled during the type inference traversal (see [Mir11] for an overview of AIR primitives):

BinopExpr: Depends on the type of the operator and operands: float*float=float, float*int=float, ... In case of an assignment, the type is *void*

BuiltinFunc: Lookup return type from RBuiltinFunctions, compare and determine type of parameter variables

ClosureExpr: Evaluate closure body, determine return type, compare and determine type of parameter variables

ExprList: Evaluate all expressions, type is type of *return*/last statement

ForStmt: Evaluate body, determine type of collection which is iterated

FuncCall: See ClosureExpr

FuncDef: Similar to FuncCall, it may not be possible to determine the parameter types, but they will be completed as soon as the function is actually used/called

IfExpr: Evaluate if/else branches

ParamExpr: Evaluate parameter value, which also defines the type of the *ParamExpr*

Program: See ExprList

MAPSkeleton: Type is a vector of the return value of the kernel function

SCANSkeleton: Type is the same type as the input vector

ZIPWSkeleton: Type is a vector of the return value of the kernel function

DOPAR: Type is *void*

SymbolExpr: Typically the type of the symbol has already been determined at this point, if not, a lookup is done from the R environment

WhileExpr: Evaluate body, Evaluate condition

After the AIR tree has been traversed, the `TypeInfo` structure is built up and contains all variables with their types, all functions with return type and all parameter types. It also contains the return type of the whole AIR program, which determines the type of the data that is transferred back to R/ALCHEMY and displayed to the user's console after execution. The types that are stored in `TypeInfo` are represented by `ArBBType` shown in Figure 6. They represent the ArBB implementations of AIR data types. Container types are able to nest container types or base types. `ArBBReferenceType` and `ArBBFunctionPointerType` do not map to AIR types, but are used to meet ArBB specific requirements. For more details, see Chapter 5.3.

Unfortunately type inference is not perfect and there are cases where it is not possible to infer a type. If the R user works with unknown (to ALCHEMY) packages or libraries which are not implemented in R, like compiled C libraries, the function signature cannot be inferred in every case. Since the database of built-in function is easily extensible, these function signatures can be added to solve this problem. Another problem arises from the possibility to define functions with different return types based on input. Functions like this

```
1  fishy<-function(x){if(x==1){return(5)}else{return("five")}}
```

are currently not possible with our backend. Possible solutions to this problem are function overloading and union data types, but they are currently not implemented in our backend. When such an error condition happens, the ALCHEMY processing is aborted and the original R program is handed back to the interpreter for sequential interpretation.

4.2. Code Transformation

In the next step, the AIR tree is traversed again and every node is translated into corresponding C++ code using the type information gathered in the previous step.

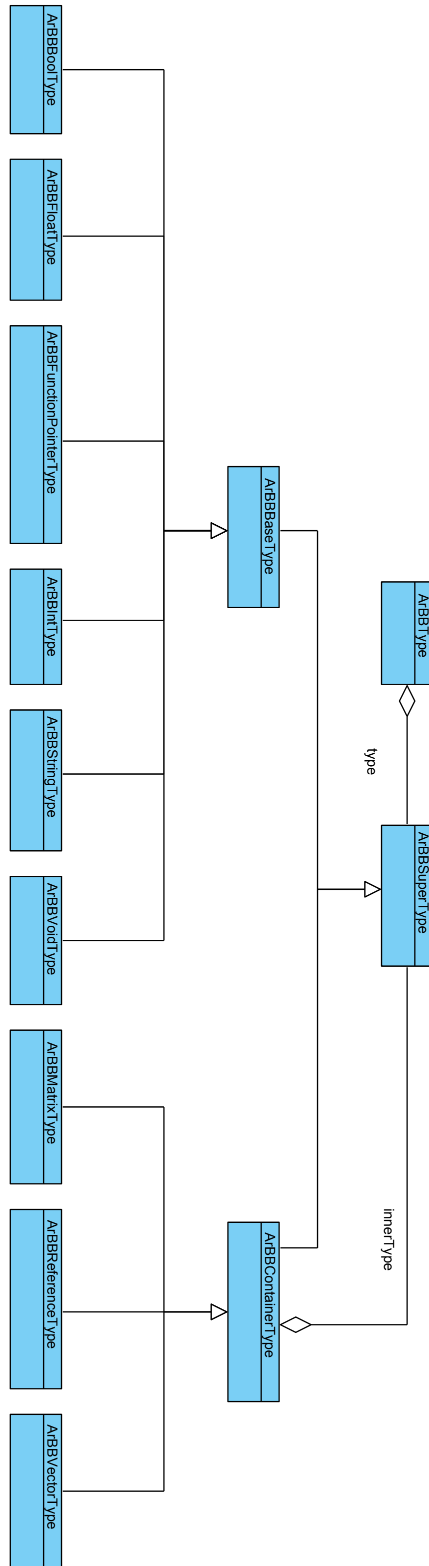

Figure 6: `ArBBType` class diagram

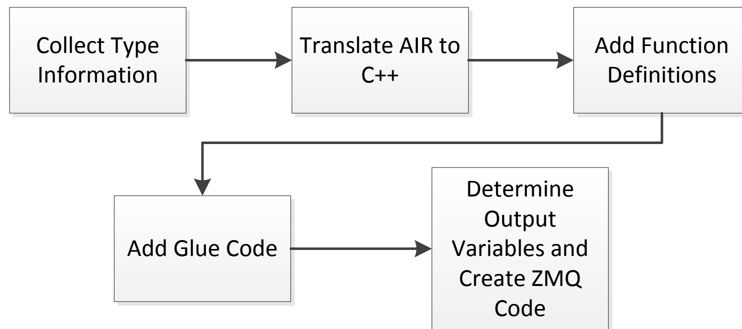

Figure 7: Flowchart code transformation

Figure 7 shows the steps involved in the code transformation. Special attention is needed when encountering function calls. AIR only contains the function name, but not the function body. So we need to retrieve it from the R environment server (see Chapter 4.3), to be able to translate it into backend code. For R closures, we introduce helper functions, because closures are not available in ArBB (ArBB has closures, but they represent something completely different). This only works for user-defined functions, where we can actually fetch a function body. R built-in base functions are translated into calls to our own library which implements them in ArBB.

Another part of this library will be the skeleton implementations [HS86, McC09, LF80, HSO07, Ble90, Gor96] using the parallel primitives available in ArBB. Building this *RtoArBB* library was an integral part of this thesis. For more details, see Chapter 5.4.

The main program will also need some glue code to meet ArBB-specific requirements. Input data must be wrapped into ArBB data types and unwrapped after the computation, before it can be transferred back to ALCHEMY. ArBB functions used in `arbb::call()` and `arbb::map()` must be `void`, so return values must be converted into function parameters. This implies that helper functions are required for even the most simple functions (see Listing 2). The C++ compiler also requires the appropriate `#include` statements, a `main()` method and ArBB initialization code to compile the source code.

After the main program has been translated, the user-defined function definitions are inserted and code to handle input and output of variables is created. During the code translation traversal of the AIR tree a data structure called `AccessInfo` is filled with

Skeleton	Input	Output	Kernel
MAP	vector<T1>	vector<T2>	<T2> map_kernel(<T1>)
SCAN	vector<T>	vector<T>	<T> scan_kernel(<T>,<T>)
ZIPW	vector<T1>, vector<T2>	vector<T3>	<T3> zipw_kernel(<T1>,<T2>)
DOPAR	matrix<T1>, matrix<T1>	matrix<T1>	no “kernel”

Table 1: Skeleton type signatures

information in which order variables are read and written, which finally determines if variables are fetched from and/or transferred back to R/ALCHEMY. The corresponding code for transferring data to and from R/ALCHEMY via ZMQ is also generated by our backend as part of the C++ glue code creation. This communication is described in the next section.

4.3. Data Transfer Between Components

Since R/ALCHEMY and the actual parallel backend execution live in different processes, input and output data has to be transferred between them. ALCHEMY uses ZMQ[Hin11] for data transfer between processes. Since ZMQ also handles network communication, these processes can even reside on different machines.

R/ALCHEMY offers two services via ZMQ which allow manipulation of the R environment and variable values. The `EnvironmentService` offers methods to access the R environment and is used e.g. by ALCHEMY to lookup symbols and function definitions during the code transformation. The `ValueService` is able to load and store values in variables before and after the backend computation. The ZMQ communication channels are visualized in Figure 8.

Listing 4 shows a C++ example of returning a computation result from the backend program to ALCHEMY. First, a ZMQ context and socket is created and the connection to the ALCHEMY listening port is initialized. After creating a message and copying the data into it, the message is sent to ALCHEMY. If the data is large, only a reference will be sent to ALCHEMY, the actual data will directly be set via the `ValueService`. As soon as the message is sent, the connection and context are closed.

4.4. Skeletons

The skeletons implemented in our backend have the following signatures. They have to be followed by PAM developers who want to use our backend. The required types, which are also used by the type inference module, are denoted in a template fashion in Table 1

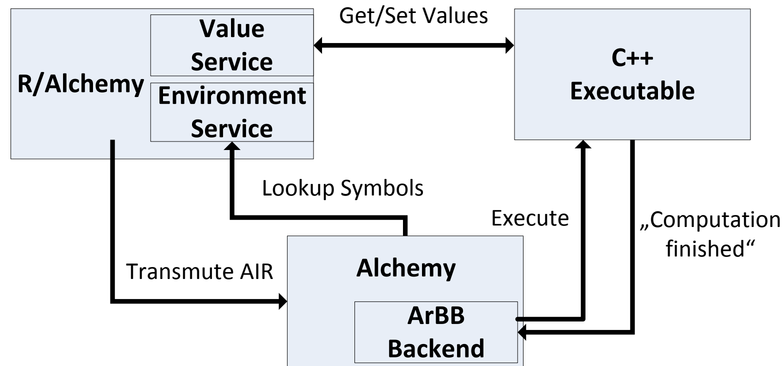

Figure 8: ZMQ communication between ALCHEMY, backend and `Value-/EnvironmentService`

The DOPAR skeleton does not have a kernel function in the typical sense, but there is still a function parameter which should contain the loop body from the original loop construct. How these skeletons can be created from R code is described in Chapter 5.5.

4.5. Intel ArBB Runtime

Before the backend program can be executed, the ArBB source code must be compiled. After linking the executable to the ArBB library and our own `R-to-ArBB-Library` (see Chapter 4.2) the program can be executed in parallel by the ArBB runtime environment. After the computation has finished, the computation results are transferred back from the ArBB program to R/ALCHEMY.

4.6. Result Collector

The computation results must be packed into AIR before they can be returned to R/ALCHEMY. The `ResultCollector` reduces the input AIR tree by replacing the code that has been transformed by its computation results. Modifications to the R environment caused by global variables or side effects of the AIR program are also handled at this stage. The AIR result gets fed back into R/ALCHEMY and is evaluated by the R interpreter. The remaining AIR code typically only displays the results on screen to the R user.

```
char result[] = "computation␣result";
void *context = zmq_init(1);
void *requester = zmq_socket(context, ZMQ_REQ);
int connect = zmq_connect(requester, "tcp://127.0.0.1:1984")
    ;
if (connect == 0) {
  zmq_msg_t message;
  zmq_msg_init_size(&message, strlen(result));
  memcpy(zmq_msg_data(&message), result, strlen(result));
  zmq_send(requester, &message, 0);
  zmq_msg_close(&message);
}
zmq_close(requester);
zmq_term(context);
```

Listing 4: C++ code example: ZMQ result return

4.7. Error Handling

One of the main goals of ALCHEMY is *transparent* automatic parallelization, which should be barely noticeable to the user except for performance. Therefore the error handling in our backend has been carefully planned to not interrupt the workflow of the R user in any case. So if anything goes wrong, a warning appears and the original R program is handed back to the R interpreter for sequential interpretation. The worst case is wasting a few seconds for trying a backend execution which fails.

Errors can happen at different stages of the ALCHEMY processing, the first being in code translation, when type inference fails (see Chapter 4.1). Network problems may cause the ZMQ communication to fail either between R/ALCHEMY and ALCHEMY or ALCHEMY and the ArBB program. The execution of the ArBB program may also fail if the system is out of memory. But in any case, the user gets his computation results, as long as he enters valid R programs.

5. Design and Implementation

This Chapter describes the software design of the ArBB backend, i.e. how the object-oriented analysis from Chapter 4 is realized in software.

5.1. System Architecture

Our ArBB Backend is divided into three main packages (see Figure 9):

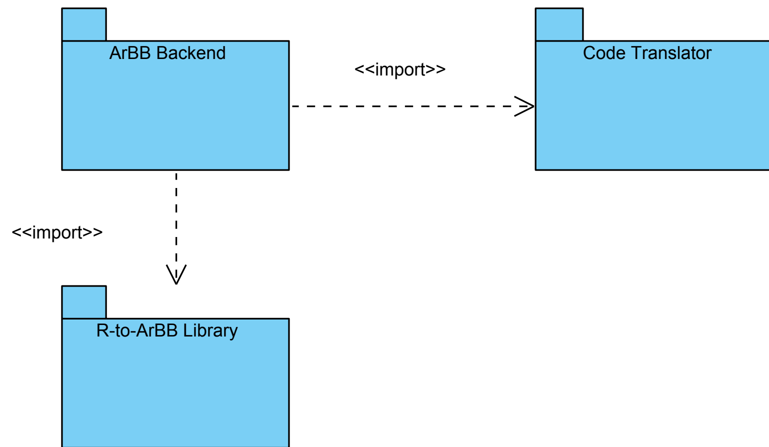

Figure 9: Package Diagram: Overview

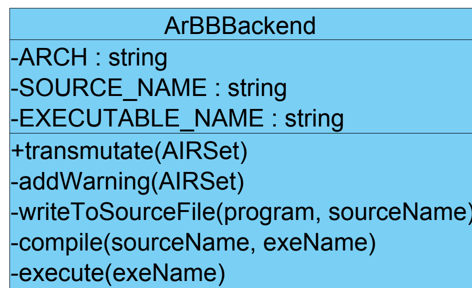

Figure 10: `ArBBBackend` class

The `ArBB Backend` package contains all the classes related to our backend being a PAM for the ALCHEMY platform. PAMs are also known as Transmutators which is a more appropriate description for our backend, as it does computation instead of parallelization analysis.

The translation of AIR code into ArBB code is handled by the `Code Translator` package. All classes involved in type inference and/or code translation are grouped there.

The `R-to-ArBB-Library` contains all the functionality on the C++ side, supporting the translated code for execution with ArBB.

5.2. ArBB Backend

The `ArBB Backend` package mainly contains two classes, the `ArBBBackend` (230 lines of code) and the `ArBBResultCollector` (60 lines of code). The `ArBBBackend` serves as Transmutator for the ALCHEMY platform. It implements the interface provided

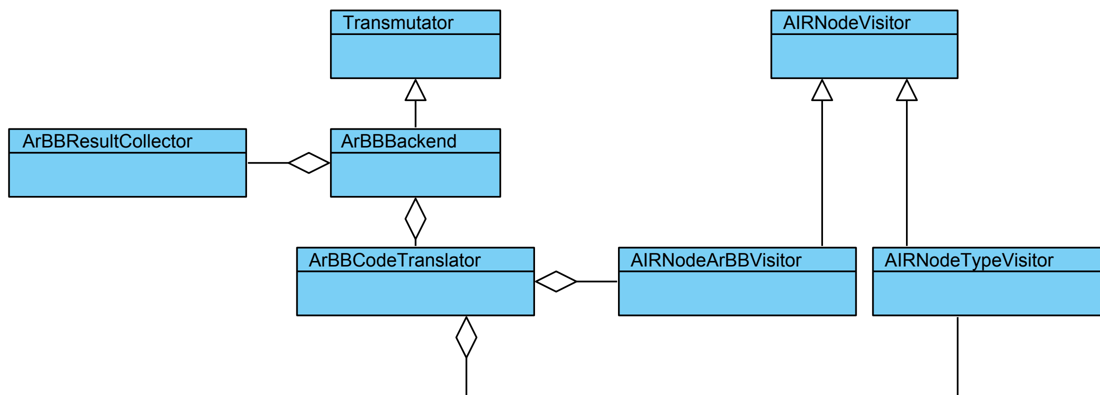

Figure 11: Class diagram: Backend execution

by a Transmutator (see Figure 10) and therefore is able to be part of an ALCHEMY transmutation chain. Since it is not transforming AIR in the sense of parallelization analysis, but does the parallel execution itself, it only makes sense as the "sink" (i.e. last element) of such a transmutation chain. The output AIR will be an AIR program containing the computation results. The `ArBBResultCollector` uses the information from the type inference and code translation to create the resulting AIR which is returned to R after the parallel execution has finished.

Figure 11 shows the top-level classes involved in the backend execution. While the `ArBBCodeTranslator` and the visitor classes belong to the Code Translator package, they are used by the ArBB backend transmutator to do the work.

When ALCHEMY creates an instance of the `ArBBBackend` class, all the required classes for the code translation are also instantiated. This initialization phase can be seen in the sequence diagram in Figure 12. The `ArBBBackend` creates an instance of the `ArBBCodeTranslator`, which in turn creates the `AIRNodeTypeVisitor` and `AIRNode-ArBBVisitor` classes. After that, an `ArBBResultCollector` is created.

As soon as ALCHEMY has transformed the original AIR program provided by the user into a parallel version using one or more PAMs, the parallelized AIR program is handed to our `ArBBBackend` for Transmutation. The backend advises the `ArBBCodeTranslator` to translate the AIR program into ArBB code and writes the source code to a file. It then compiles the source code and links it against the ArBB library as well as our own `R-to-ArBB-Library`. Then it runs the executable within the Intel ArBB run-time and after the computation has finished, collects the computation results from the `ArBBResultCollector`. The resulting AIR is then returned to ALCHEMY which hands it back to the R interpreter. This process can be seen in the sequence diagram in Figure

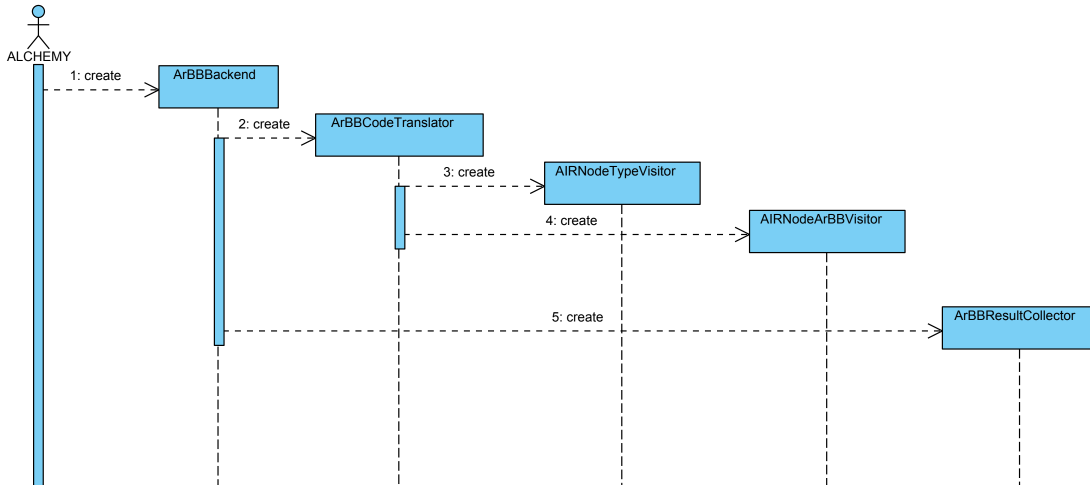

Figure 12: Sequence diagram: `ArBBBackend` initialization phase

13.

5.3. Code Translation

Before the parallel execution can be performed, the AIR program needs to be translated into an ArBB program. This is handled by the `ArBBCodeTranslator` (51 lines of code), which performs the steps outlined in the analysis (shown in Figure 7). The translation is split in two parts: type inference and actual code generation, which are handled by the `AIRNodeTypeVisitor` and `AIRNodeArBBVisitor` respectively. The visitor pattern [GHJV01] is used as a natural way to traverse the AIR tree in both cases. The `AIRNodeVisitor` class provides the interface for the pattern, which is also used to realize various functionalities within ALCHEMY.

The sequence diagram in Figure 14 shows the procedure of code translation: First, the type visitor is called on the AIR tree to collect type information. The ArBB visitor then uses the type information to create a C++/ArBB version of the AIR program and collects access information for each variable to determine if it needs to be fetched from or transferred back to R/ALCHEMY. The translated program is then handed back to the backend for compilation and execution.

5.3.1. AIRNodeTypeVisitor

The first step of the code translation is type inference, which is realized in the `AIRNodeTypeVisitor` class (see Figure 15). Consisting of ~1100 lines of code, it contains

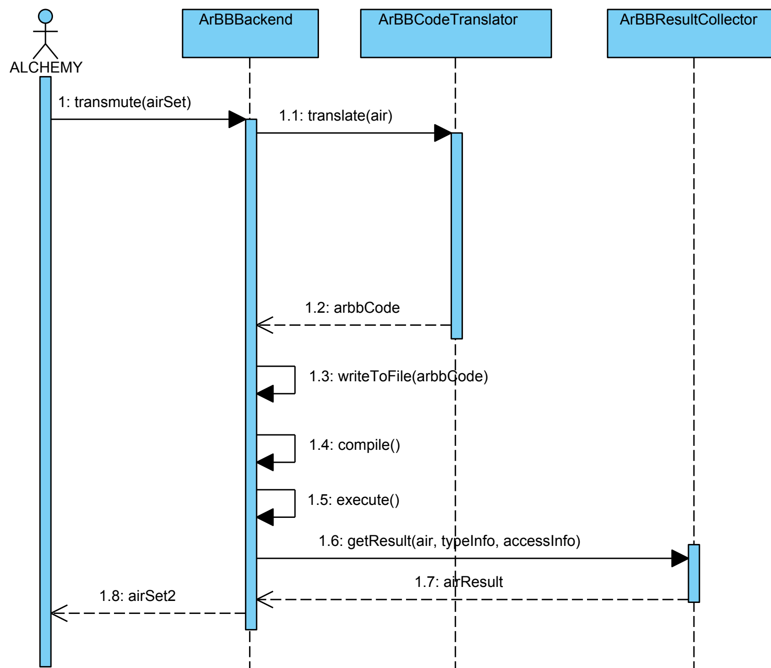

Figure 13: Sequence diagram: `ArBBBackend::transmute()`

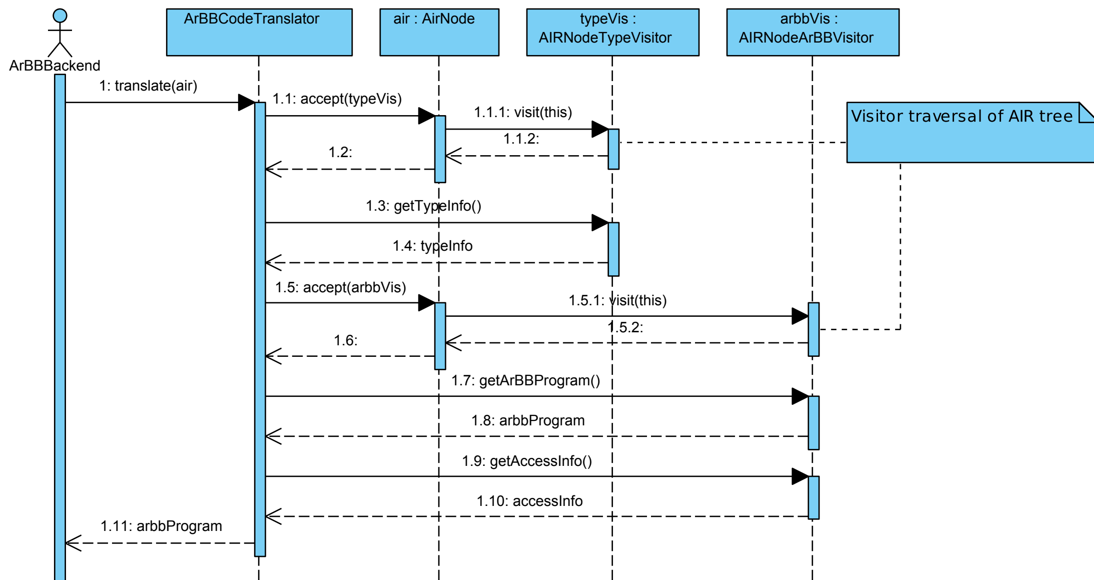

Figure 14: Sequence diagram: `ArBBCodeTranslator` visitor

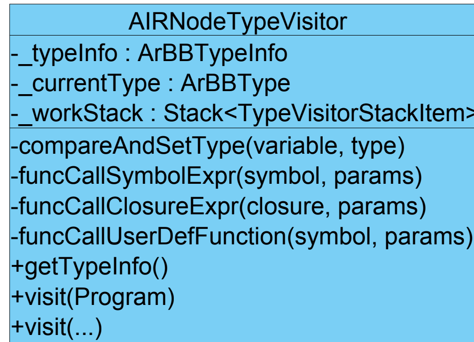

Figure 15: `AIRNodeTypeVisitor` class

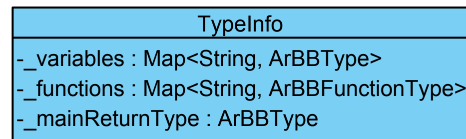

Figure 16: `TypeInfo` data model

the code for all exisiting AIR nodes. The AIR tree is traversed using the visitor pattern while the `TypeVisitorStack` keeps track of the different scopes that occur during code blocks and function calls.

A structure called `TypeInfo` is filled during traversal with vital information for the next step of the actual code generation. It holds a mapping from variable names to types as well as from function names to parameter and return types. It also keeps the return type of the whole AIR program, i.e. the type of the data that is returned to the user. The data model for `TypeInfo` can be seen in Figure 16. Types are represented by the `ArBBType` classes (see Figure 6), which map AIR types to C++/ArBB types.Whenever a type is determined, it gets stored in `TypeInfo`, if the same variable or function already exists there, a mechanism tries to find a common (super)type, in case the types do not match.

Special handling is required for function calls to user-defined functions. Since the AIR code does only contain the function name and its arguments, its signature and function body are unknown. To get this information, the function name is looked up via ZMQ from the `EnvironmentService` inside R/ALCHEMY. The corresponding sequence diagram can be seen in Figure 17. Looking up a function name returns an AIR expression, which contains the function body and parameters. From there, the function signature is created by matching the arguments of the function call to the parameters of the looked

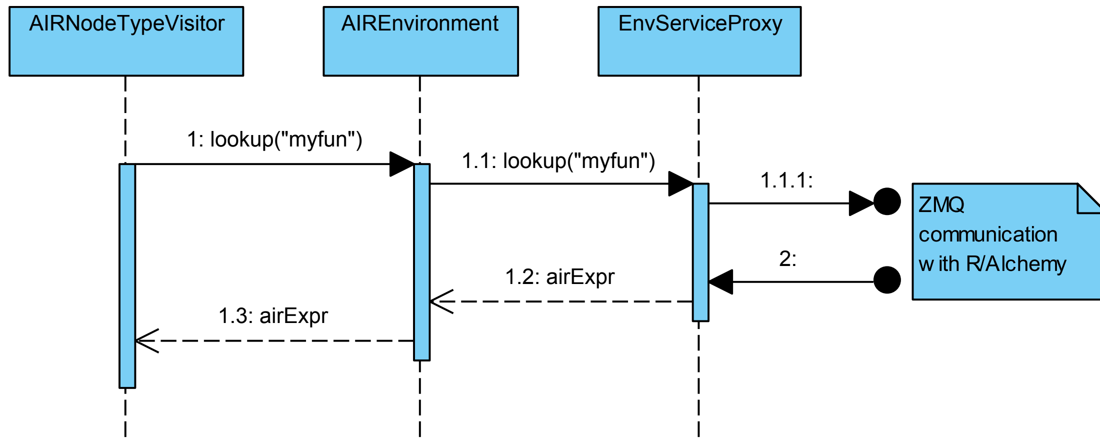

Figure 17: Sequence diagram: `EnvironmentService` lookup

up function and evaluating the function body using the type visitor.

Determining whether a function call goes to a user-defined function or to a R built-in function is not obvious. Looking up a built-in function from the Environment Service would only deliver the fact, that it is a built-in function, but not a function signature, which is required for type inference. That is why we keep a database called `RBuiltinFunctions`, which is queried before performing a lookup at the `EnvironmentService`. The database contains the function signatures of all R base functions with parameter and return type as well as their corresponding function name from our C++ counterpart, the `R-to-ArBB-Library`. If possible, the function signature is kept identical between the R and C++ versions, such that code translation is as easy as possible. The database can be extended if further packages or libraries are used.

5.3.2. AIRNodeArBBVisitor

Next in line is the `AIRNodeArBBVisitor` (~1550 lines of code), which is responsible for the actual code generation. A class diagram can be found in Figure 18. Similar to the type visitor, the AIR tree is traversed by the `AIRNodeArBBVisitor`. In every AIR node visited, the corresponding C++ code is created using the type information gathered in the previous step while keeping track of the read/write accesses to variables in the `AccessInfo` data structure.

`AccessInfo` stores information about variables, whether they have been read, written and if their first access was a write or a read operation. Since the backend executable needs to fetch the variables from the R environment and write them back after the computation, we need to generate code to do this. Deciding which variables to fetch and

AIRNodeArBBVisitor

-_body : StringBuilder
-_includeSet : Set<Include>
-_closureSet : Set<Closure>
-_functionSet : Set<ArBBFuncDef>
-_typeInfo : ArBBTypeInfo
-_accessInfo : ArBBAccessInfo

-createVoidVersion(function)
-handleArBBMain()
-handleMain()
-handleDefineVariables()
-handleTransmitVariables()
-handleFetchVariables()
-handleForwardDeclarations()
-handleFunctions()
-handleIncludes()
+getArBBProgram()
+getAccessInfo()
+visit(Program)
+visit(...)

Figure 18: `AIRNodeArBBVisitor` class

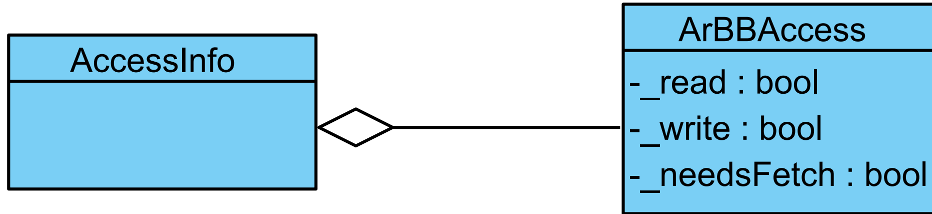

Figure 19: `AccessInfo` data model

which to transfer back is made possible from the information provided by `AccessInfo`. The data model can be found in Figure 19.

Again, special attention is required for functions, especially the ones used as kernel functions for our skeletons. Since the skeletons are implemented in ArBB using `arbb`*::map*, which requires the functions to have a `void` return type, they need to be restructured by our code. Kernel functions like

```
1 fun<-function(x){return(x*2)}
```

end up translated by `createVoidVersion()` to:

```
1 void void_fun(arbb::f32& result, arbb::f32 x){result = x*2}
```

So kernel functions are translated into void functions, where the return value ends up being the first parameter of the function. The surrounding code also needs to adapt to these restrictions. Similar constraints from ArBB lead to the use of many wrapper and helper functions throughout the code.

Once the AIR tree has been visited, the `ArBBCodeTranslator` requests the arbb code, which causes the assembly of a complete source code file. This involves the following steps (see Figure 20): `handleIncludes()` first writes the required #`include` statements to the source code. `handleForwardDeclarations()` then writes the forward declarations of all user-defined functions, their `void` counterparts (if required), as well as the various helper/wrapper functions to the file, such that the order in which `handleFunctions()` writes down all the function definitions does not matter. `handleArBBMain()` creates a function called `arBBMain()` which contains the translated body of the AIR program. The `main()` function required by C++ then contains an `arbb::call()` to the arBBMain function, which hands it over to the ArBB runtime. `handleMain()` also encloses this call with the required glue code, as well as variable definitions and ZMQ code to fetch and transmit variables from/to R/ALCHEMY based on the information from AccessInfo.

An example of a generated ArBB program which applies the sin function with a MAP

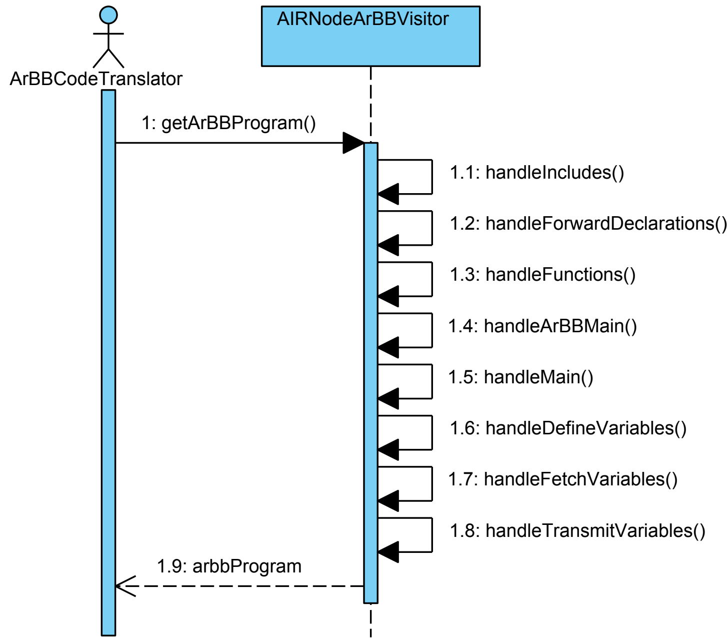

Figure 20: Sequence diagram: ArBB program creation

skeleton in parallel to a vector is presented in Appendix A.

5.4. R-to-ArBB-Library

The `R-to-ArBB-Library` is a C++ library in which we implemented R base functions, parallel skeletons, ZMQ communication, helper and wrapper functions using Intel ArBB. The library, consisting of 530 lines of code, can be found in RtoArBBLib.hpp and gets referenced in every generated ArBB program.

R Base Functions

While R control flow primitives and all AIR primitives (a detailed explanation can be found in [Mir11]) are implemented in the `AIRNodeArBBVisitor` and many instructions like logical operators or simple math functions directly correspond to ArBB counterparts via mapping from `RBuiltinFunctions`, other functions need more attention. Some of them have ArBB counterparts, but differ in parameter order, which lead to some implementation effort. Others do not have an ArBB implementation at all, and therefore required a full implementation. Not all of the R base functions have been implemented as part of this thesis, but will be implemented in an ongoing effort to improve the system. The list of all implemented R primitives and base functions can be found in Table 2.

<table>
<tr><td>if</td><td>return</td><td>while</td><td>for</td></tr>
<tr><td>repeat</td><td>switch</td><td>break</td><td>next</td></tr>
<tr><td><-</td><td>=</td><td>+</td><td>-</td></tr>
<tr><td>*</td><td>/</td><td>^</td><td>==</td></tr>
<tr><td>!=</td><td><=</td><td>>=</td><td><</td></tr>
<tr><td>></td><td>&</td><td>|</td><td>!</td></tr>
<tr><td>&&</td><td>||</td><td>:</td><td>{}</td></tr>
<tr><td>()</td><td>[]</td><td>[[]]</td><td>length</td></tr>
<tr><td>c</td><td>dim</td><td>round</td><td>log</td></tr>
<tr><td>abs</td><td>floor</td><td>ceiling</td><td>sqrt</td></tr>
<tr><td>exp</td><td>cos</td><td>sin</td><td>tan</td></tr>
<tr><td>vector</td><td>matrix</td><td>row</td><td>col</td></tr>
</table>

Table 2: Implemented R primitives and base functions

MAP
SCAN
ZIPW
DOPAR

Table 3: Implemented parallel skeletons

Skeletons

The skeletons implemented in this thesis form the foundation for parallel execution in our backend. They all rely on the Intel ArBB platform and the `arbb::map()` function specifically, which albeit the name, is more flexible than the MAP skeleton. The four implemented skeletons are listen in Table 3. The following passages describe how they are implemented.

MAP The MAP skeleton uses `arbb::map()` directly in its most simple form, but still requires the code transformation to provide a void version of the kernel function.

```
1 template<typename FunctionType>
2 arbb::dense<arbb::f32> MAP_HELPER(FunctionType f, arbb::
      dense<arbb::f32> i1)
3 {
4   arbb::dense<arbb::f32> r;
5   arbb::map(f)(r, i1);
6   return r;
7 }
```

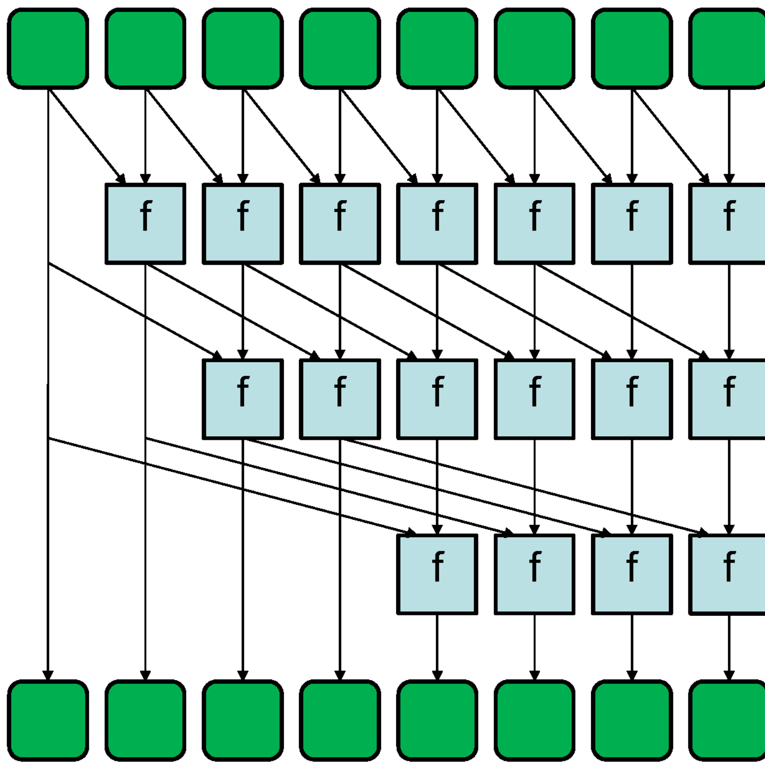

Figure 21: SCAN implementation[McC09]

First, a return vector `r` is created. Then, the kernel function `f` is applied in parallel to every element of the input vector `i1`. The results are stored in `r`.Although it seems wrong to return a local variable, the ArBB memory management allows such constructs.

SCAN The implementation of the SCAN skeleton is based on a step-efficient algorithm from [McC09]. It exploits the (required) associativity of the kernel function to be able to perform parallel computation and combine the results consecutively. The algorithm is illustrated in Figure 21. All the kernel invocations in one horizontal line can be performed in parallel. To finish the computation, $log(N)$ (where N represents the length of the input vector) such parallel steps are required. While the algorithm requires more ($N * lg(N) - \frac{N}{2} - 1$) invocations of the kernel function as the sequential version ($N - 1$), it performs well when executed in parallel (see evaluation in Chapter 6.2.2). The kernel function is also embedded into the `scan_kernel` function, which is generated dynamically during the code generation phase. The `SCAN_HELPER` is fixed and resides in the `R-to-ArBB-Library`.

```
1  template<typename FunctionType>
2  arbb::dense<arbb::f32> SCAN_HELPER(FunctionType scan_kernel,
       arbb::dense<arbb::f32> i1)
3  {
4    arbb::dense<arbb::f32> r = i1;
5    arbb::f32 log = arbb::log(arbb::f32(i1.length())) / arbb::
        log(arbb::f32(2));
6    _for(arbb::f32 i = 0, i < log, i++)
7    {
8      arbb::map(scan_kernel)(r, i);
9    } _end_for;
10   return r;
11 }
12
13 void scan_kernel(arbb::f32& x, arbb::f32 round)
14 {
15   arbb::usize position;
16   arbb::position(position)
17   _if(arbb::f32(position) >= arbb::pow(2,round))
18   {
19     x = kernel(x, arbb::neighbor(x, arbb::f32(position)-arbb
          ::pow(2, round)))
20   }_end_if;
21 }
```

While the `SCAN_HELPER` function handles the $log(N)$ parallel execution rounds, the `scan_kernel` computes the indices for the current round and applies the kernel function to the two proper elements of the vector.

ZIPW The ZIPW skeleton looks and in fact works similar to the MAP implementation, but it requires some additional code in the `zipw_kernel` function which is generated dynamically in the code translation step. This is caused by an ArBB limitation which does not allow function pointers to be passed as arguments to the arbb::map function.

```
1  template<typename FunctionType>
2  arbb::dense<arbb::f32> ZIPW_HELPER(FunctionType zipw_kernel,
       arbb::dense<arbb::f32> i1, arbb::dense<arbb::f32> i2)
```

```
3 {
4   arbb::dense<arbb::f32> r;
5   arbb::map(zipw_kernel)(r, i1, i2);
6   return r;
7 }
8
9 void zipw_kernel(arbb::f32& x, arbb::f32 i1, arbb::f32 i2)
10 {
11   x = kernel(i1, i2);
12 }
```

Instead of taking one input vector with MAP, ZIPW takes two input vectors `i1` and `i2` and applies the `zipw_kernel` function in parallel.

DOPAR Although DOPAR also uses `arbb::map()`, it is not possible to have a predefined implementation in the `R-to-ArBB-Library`, all of the skeleton code is generated dynamically during skeleton handling in the code translation. Since the DOPAR skeleton does not have a kernel function in the sense of the other skeletons, but an assignment of a matrix cell, which might have all kinds of things on the right hand side of the assignment, the kernel function and corresponding `arbb::map()` call need to be generated dynamically. The following Listing contains a sample output:

```
1 template<typename FunctionType>
2 void DOPAR_HELPER(FunctionType dopar_kernel, arbb::dense<
     arbb::f32,2> i1, arbb::dense<arbb::f32,2> i2, arbb::f32
     from, arbb::f32 to, arbb::f32 N)
3 {
4   arbb::dense<arbb::f32> r;
5   _for{arbb::f32 i = 1, i < N, i++)
6   {
7     r = i1.row(i);
8     arbb::map(dopar_kernel)(r, i2, from, to, i);
9     arbb::replace_row(i1,i,r);
10  } _end_for;
11 }
12
13 void dopar_kernel(arbb::f32& x, arbb::dense<arbb::f32,2> y,
```

```
    arbb::f32 from, arbb::f32 to, arbb::f32 i)
14 {
15   arbb::usize j;
16   arbb::position(j);
17   _if (j>=from && j<=to)
18   {
19     x=y(i-1,j);
20   }_end_if;
21 }
```

The DOPAR_HELPER cuts out the rows or columns (depending on how the matrix is iterated in the skeleton) of the matrix which is iterated by the DOPAR skeleton, and applies the dopar_kernel in parallel for all the rows/columns. The kernel function wraps the body part of the skeleton, which contains the actual computation.

ZMQ Communication

The ZMQ library is used to transfer variables between the ArBB backend program and R/ALCHEMY (cf. Figure 8). Since our goal is to keep the generated code fairly human readable and transferring a variable requires many lines of code, we created the ZMQConnection class, which hides all the ZMQ implementation and Environment-/Value-Service protocol details from the generated source code file. The interface provided looks as follows:

```
1 ZMQConnection(const char* svc_name, const char* svc_uri)
2 ~ZMQConnection()
3 //SEND METHODS
4 void send_arbb_value(arbb::f32 val, int env_id,const char
    symbol[])
5 void send_arbb_value(arbb::i32 val, int env_id,const char
    symbol[])
6 void send_arbb_vector(arbb::dense<arbb::f32>& vec, int
    env_id,const char symbol[])
7 void send_arbb_vector(arbb::dense<arbb::i32>& vec, int
    env_id,const char symbol[])
8 void send_arbb_matrix(arbb::dense<arbb::f32, 2> matrix, int
    env_id,const char symbol[])
```

AIR Type	ArBB Type
AIRVector	arbb::dense<T>
AIRMatrix	arbb::dense<T, 2>
AIR integer	arbb::i32
AIR real	arbb::f32
AIR logic	arbb::boolean
AIR string	arbb::dense<arbb::u8>

Table 4: Mapping between AIR and ArBB types

```
9  void send_arbb_matrix(arbb::dense<arbb::i32, 2> matrix, int
      env_id,const char symbol[])
10 //RECEIVE METHODS
11 arbb::f32 receive_arbb_value_float(int env_id, const char
      symbol[])
12 arbb::i32 receive_arbb_value_int(int env_id, const char
      symbol[])
13 arbb::dense<arbb::f32> receive_arbb_vector_float(int env_id,
       const char symbol[])
14 arbb::dense<arbb::i32> receive_arbb_vector_int(int env_id,
      const char symbol[])
15 arbb::dense<arbb::f32, 2> receive_arbb_matrix_float(int
      env_id, const char symbol[])
16 arbb::dense<arbb::i32, 2> receive_arbb_matrix_int(int env_id
      , const char symbol[])
```

The class contains methods for sending and receiving values. The parameters always include the environment id as well as the name of the symbol where the values are stored/retrieved. There are methods for individual values, vectors and matrices by twos, which transfer integers or floats respectively. The implemented communication protocol for the services can be found in Chapter 5.6.

The environment id and service URI, which is just the IP address and port to the `EnvironmentService`, are generated dynamically, since they are specific for each R instance. The data types that are transferred are fixed by the mapping between AIR and ArBB types and can be found in Table 4. . So the set of types is limited by the AIR (R) types, which means we do not have to deal with arbitrary C++ types, we do not even make use of all of the types provided by ArBB. That means R/ALCHEMY and the `R-to-ArBB-Library` know all the types and especially know how to transfer them

and convert them to and from AIR types.

Helper/Wrapper

As already seen in the skeleton section, ArBB requires a lot of helper and wrapper functions to work around the restrictions and limitations. Another group of helper functions and macros simplifies the code translation step by e.g. providing function-like syntax from a macro which wraps a matrix access. Others just wrap existing ArBB counterparts to R functions which have different syntax or parameter order like the ":" operator. Here are some examples:

```
1  #define RETURN_MACRO(x) return x
2  #define RESULT_MACRO(x) result = x
3  #define LENGTH_HELPER_MACRO(x) x.length()
4  #define VECTOR_ACCESS_HELPER_MACRO(x, i) x[i]
5  #define MATRIX_ACCESS_HELPER_MACRO(x, i, j) x(i,j)
6  #define MATRIX_ROW_HELPER_MACRO(x, i) x.row(i)
7  #define MATRIX_COL_HELPER_MACRO(x, j) x.col(j)
8
9  arbb::dense<arbb::f32> SEQUENCE_HELPER(arbb::f32 from, arbb
       ::f32 to)
10 {
11   return arbb::indices(from, to - from + 1, arbb::f32(1.0));
12 }
```

5.5. Skeleton Generator

The R language does not have parallel constructs, like skeletons, which we use to express parallelism in our programs. This is one of the reasons why the AIR intermediate language was created. Skeletons in AIR are typically created inside of PAMs, which analyze the sequential R program, find parallelization opportunities and introduce skeletons into the AIR program to express the parallelism. Since the MATSU and SURE PAMs are not finished yet, we are lacking transmutators which are able to generate skeletons.

Therefore we created the `SkeletonGenerator` transmutator (136 lines of code), which allows us to test our backend by transforming special function calls into skeletons. This allows us to manually insert parallel constructs into the R code. This also simplifies the validation of the code transformation, where me make frequent use of this mechanism (see Chapter 5.7). The `SkeletonGenerator` transforms `FuncCall` nodes with function name "`alchemy.applySkeleton`" into their respective skeleton nodes. E.g. the R code

Figure 22: Data flow diagram: Transmutator Configuration

```
1  alchemy.applySkeleton('MAP', fun, x)
```

creates a `MAPSkeleton` node within the AIR tree, which then is executed in parallel. The `SkeletonGenerator` also dereferences all function calls to user-defined functions, similar to the lookup mechanism during the code translation, to be able to also catch and transform relevant function calls in user-defined functions.

A typical transmutation chain is shown in Figure 22. `FuncDefFilter` is just a pre-filtering PAM which filters out function definitions, since there is nothing to compute and they are stored in the R environment anyway. Then the `SkeletonGenerator` transforms the special function calls into skeletons and the ArBB backend executes the AIR program in parallel. Finally the computation result is returned to R/ALCHEMY.

5.6. Extension of the Value-/EnvironmentService

The `ValueService` and `EnvironmentService` provide interfaces to transfer data between R/ALCHEMY, ALCHEMY and the ArBB backend via ZMQ.

The `EnvironmentService` allows to modify the R environment, lookup symbols (variables, functions) and write computation results or intermediate results in terms of variables back to the R environment. When transferring AIR programs from R/ALCHEMY to Alchemy or computation results back, they might contain large amounts of data. Since the communication is XML encoded, there is a lot of overhead when transferring large vectors/matrices. The `ValuesService` allows to store these values, and putting a proxy instead of the values into the AIR XML code, tremendously reducing the overhead. The interface provided by the `ValueService` allows to transfer plain values instead of the XML encoding.

Transferring the XML encapsulation was not the only overhead, that was slowing down large value transfers. Already the creation of the XML around the values was tremendously expensive. Figure 23 shows the comparison of the XML value transfer

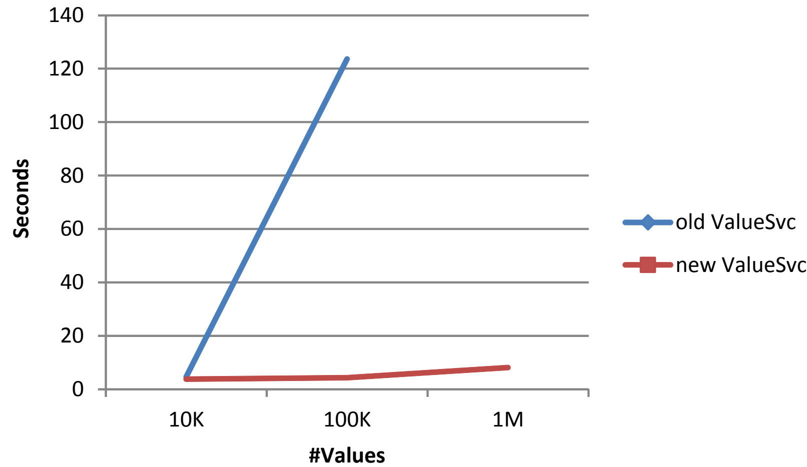

Figure 23: Performance comparison: old/new `ValueService`

(old) with the plain value transfer (new) by means of the myfun example (cf. Chapter 6.1.1) with 10K, 100K and 1M values. With 10K elements, the plain version with 3.77 seconds vs. 4.66 seconds with the XML version is already at an advantage of ~25%. With 100K elements, the XML version already becomes unbearably slow (123.62s vs. 4.25s). The experiment with 1M elements was aborted after 10 minutes, when the XML version still was not finished.

The previous implementation provided by ALCHEMY only allowed to lookup symbols from the EnvironmentService and fetch values from the ValuesService. But the other direction of modifying the R environment and putting values into the `ValueService` was not available. So we extended the services and their interfaces to also allow the ArBB backend to write data back to R/ALCHEMY.

The new `ValueService` endpoint is called "`ValueSvc.setVal`" and conforms to the following protocol:

- Request: [<type>, <length>, <value1>, <value2>, ...]
- Response: ["OK", <value ID>]

The new "`EnvironmentSvc.install`" endpoint interface is:

- Request: [<environment id>, <symbol name>, <type>, <length>, <value1>, <value2>, ...]
- Response: ["OK"]

While there was already a way to get variables from the R environment, namely "`EnvironmentSvc.lookup`", this endpoint responds with XML, which is inappropriate for large data, the new "`EnvironmentSvc.getVal`" returns plain values:

- Request: [<environment id>, <symbol name>]
- Response: ["OK",<length>,<value 1>,<value 2>, ...]

5.7. Validity of Code Transformation

Apart from the performance evaluation in Chapter 6.1, we also wanted to make sure that the main part of our backend, the code translation, as well as the implemented R-to-ArBB-Library works correctly. Therefore we created lots of test cases for all the different aspects of the code translation, as well as the implementations inside our library. The tests are divided into the following categories: AIR nodes, skeletons, R base functions and data input/output. We also have a special category *error handling*, which forces erroneous behavior to see if it is handled gracefully.

AIR Nodes

This set of test cases (~60 tests) focuses on the type inference and code translation of the different AIR nodes which may exist in a given AIR program. This starts from simple type creations

```
1 a<-1
2 b<-1.0
3 c<-1:5
4 d<-"foo"
5 e<-array(1,dim=c(2,2))
```

over checking functionality of the various possibilities of the BinopExpr

```
1 x<-1.0
2 x+2.0
3 x*2.0
4 x/2.0
```

to tests which check the functionality of SubscriptExpr:

```
1 x<-array(1,dim=c(10,10))
2 x[1,1]
```

```
3 x[,1]
4 x[1,]
```

The various control flow nodes are checked

```
1 x<-1
2 for (i in 1:10) x<-x+1
```

as well as more complex cases, like the handling of function pointers in FuncCall nodes:

```
1 fun<-function(x){return(x+5)}
2 fun2<-function(f,x){return(f(x))}
3 fun2(fun,1)
```

Skeletons

Technically, skeletons are also AIR nodes, but since they have far more complex handling in the type inference and code generation steps, they have their own category with ~25 very specific tests. Starting from simple tests, which test the basic functionality

```
1 fun<-function(x,y){return(x+y)}
2 x<-1:10
3 alchemy.applySkeleton("SCAN",fun,x)
```

there are also tests which check the creation of helper functions in case a closure is used as kernel function instead of a function symbol:

```
1 a<-1:10
2 b<-1:10
3 alchemy.applySkeleton("ZIPW",function(x,y){return(x-y)},a,b)
```

We also test the interoperability of multiple skeletons combined:

```
1 a<-1:100
2 map<-function(x){return(x)}
3 scan<-function(x,y){return(x+y)}
4 alchemy.applySkeleton("MAP",map,alchemy.applySkeleton("SCAN"
    ,scan,a))
```

Error scenarios are also checked, if e.g. a kernel function with invalid signature is correctly handled

```
a<-1:10
fun<-function(x,y){return(5)}
alchemy.applySkeleton("MAP", fun, a)
```

or if the skeleton is able to deal with empty vectors:

```
square<-function(x){x*x}
alchemy.applySkeleton("MAP",square,vector("numeric"))
```

Most of these tests apply to all skeletons and exist in every variation, still there are more tests which check specific branches in the type inference or code generation code for a skeleton.

R base functions

For all supported R functions, we have test cases checking basic functionality as well as edge cases of the implementation. This amounts to ~90 tests.

```
x <- c(1,2,3)
```

```
sin(5)
```

```
3:8
```

We also check for edge cases like

```
sqrt(-2)
```

for complex numbers which are currently not supported by AIR. Otherwise the tests in this category are standard tests, making sure the implementations work as expected.

Data input/ouput

Fetching input data and transferring output data within the generated backend code involves modules from R/ALCHEMY, type inference, code generation and the ZMQ class in the R-to-ArBB-Library to work seamlessly together, which requires extensive testing. Some of these tests require storing values in the R environment, before ALCHEMY runs, so they are not generated in the backend, but can be fetched from the environment instead. This can be controlled by calling *alchemy.enable()* or *alchemy.disable()* from the R code. It starts with testing the lookup of function definitions

```
1 fun<-function(x){return(3)}
2 alchemy.enable()
3 fun(1)
```

and goes on with testing the AccessInfo structure which holds information in which order variables have been read/written, which ultimately decides if a variable has to be fetched, transferred or just defined locally. Fetching is tested with

```
1 x<-5
2 alchemy.enable()
3 x
```

transferring with

```
1 x<-5
```

and the following test leads to fetch of x and a local definition and transfer of y:

```
1 x<-5
2 alchemy.enable()
3 y<-x
```

Other tests mainly check the ZMQ transfer implementation of all the different AIR types as well as the type conversion between R types, AIR types and ArBB types in their respective code pieces. The data input/output category contains ~20 tests.

Error Handling

This category holds all the tests (~20) related to error handling, ensuring that they are handled gracefully, i.e. the original R program is handed back to R/ALCHEMY and executed sequentially. Errors range from invalid input, like non-existant functions,

```
1 foobar(x)
```

to programs which can be handled by R, but not by our backend. A variable which takes different types during execution

```
1 x<-5
2 x<-"hello"
```

or system commands that cannot be handled by the backend

```
1 q()
```

are just a few examples.

But this category contains also tests which cannot be simply expressed in R code, like the simulation of network failures or out-of-memory exceptions, when calling:

```
1 x<-array(1,dim=c(2^31,2^31))
```

In any case, we made sure that an error always led to the fallback method of evaluating the input program with R itself.

6. Evaluation

To evaluate the performance of our backend, we performed several measurements. The validity of the code produced in the code transformation step of our backend was also tested. The overhead induced by our code transformation and compilation steps were also measured for the various input programs. The experiments were conducted on a standard PC with a AMD FX-8120 eight-core CPU @ 3.1 GHz. It has 8 GB memory and runs Ubuntu 12.04 (64 bit).

All experiments were executed 10 times to compute the average execution time. The notion of *speedup*, which we use to compare the execution time of parallel programs against their sequential counterparts, is defined as follows:

$$Speedup = \frac{ExecutionTime(P_{sequential})}{ExecutionTime(P_{parallel})}$$

The term *relative speedup* is used when comparing the parallel code run on a single core compared to the same parallel code run on multiple cores.

To understand the results presented in the *Performance* Chapter, it is important to know how the execution time of a backend execution is composed: First the AIR programs needs to be translated into an ArBB program, which then gets compiled. Upon execution of the ArBB program, it fetches the input data from R/ALCHEMY, then the actual computation happens and finally the computation results are transferred back via ZMQ. Because of this, the relative speedup of the parallelized version cannot scale linearly with the number of CPU cores. When measuring the sequential R program without ALCHEMY, nothing of the above happens, R has all the data at hand and just interprets the R code.

Some of the measurements show super linear speedup, which is far above the theoretical limit of eight with eight-core CPUs. This comes from the fact that we do not only execute the code in parallel, but also compile it, which by itself is already much faster than the R interpreter.

6.1. Performance

This Chapter contains the performance evaluation of some example programs, as well as evaluations of the various skeletons, ZMQ and scaling of ArBB programs in general. For our tests we use input sizes of 10K, 100K, 1M and sometimes 10M values, which are realistic numbers for some areas of bioinformatics, e.g. high throughput sequence analysis [MD12, MAL+09].

6.1.1. Myfun Example

The *myfun* example originates from the evaluation of the ALCHEMY framework[Mir11] and serves as a baseline for our measurements. We checked if the results were reproducible and how our approach compares performance-wise. The R source code for the example is shown in Listing 6. The structure of the parallelized code shows no data dependencies, which allows good scaling on a multicore architecture. Also, the relatively high number of operations on every element of the input vector shows the advantages of compilation vs. interpretation.

Backend Comparison

The backend comparison tries to reproduce the results of the evaluation of the myfun example by the ALCHEMY developer and set them into perspective with our own ArBB backend. It compares the sequential R version with a parallel version using the R multicore package on 8 cores and our version with a MAP skeleton, again on 8 cores. The input values are 10K, 100K and 1M elements respectively.

Our measured results from the sequential and R multicore version are similar to the ones found in the evaluation from the ALCHEMY developer, which makes us confident, that ALCHEMY and our measurement environment are set up correctly.

Figure 24 compares the total runtime in seconds of the three tested backends with 10K, 100K am 1M elements. While our backend (3.99s) is merely faster than the sequential version (4.55s) for 10K elements, at 1M elements the sequential version takes almost 6 minutes, our backend finishes in 8.96 seconds. The reason why we do not get a linear scaling here is that we have relatively high fixed costs for setting the backend up which

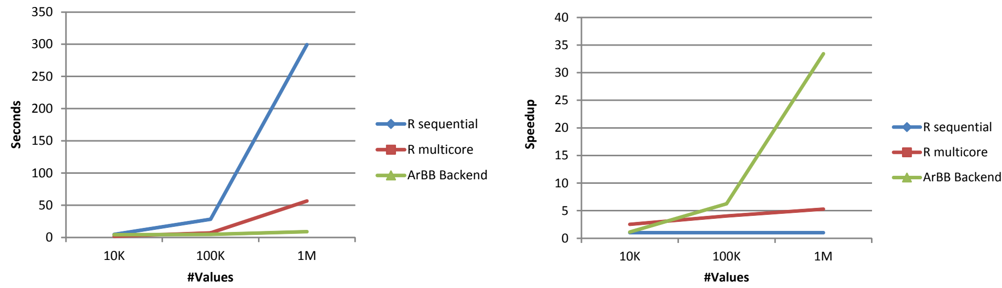

Figure 24: myfun Backend Comparison, total runtime and speedup

then amortizes when computing more elements. More about these costs can be found in Chapter 6.1.6.

The speedup of the two parallel backends with the sequential version as baseline is plotted in Figure 24. With a speedup of 1.14 at 10K elements, 6.24 at 100K and even 33.42 at 1M, we are reaching super linear speedup which comes from the fact that we are compiling the code in addition to running it in parallel.

Runtime Partition

This section sheds light on how the different parts of the backend execution factor into the total runtime of the myfun example. The program was tested again with 10K, 100K and 1M elements. Variation of the amount of used CPU cores from 1-8 shows how the backend is scaling.

Figure 25 contains the plots for 10K, 100K and 1M elements respectively. They are divided into translation, compilation and execution. *Translation* is the time it takes for the ArBB backend to translate the AIR program into ArBB code, it typically takes a few milliseconds (3ms in this case), but depends heavily on how many symbol lookups are made. Since it contributes so few to the total runtime, it is barely noticeable on the plot. *Compilation* is the time it takes the C++ compiler to make an executable out of the generated C++ source code file. With the myfun example, it took 3.04s. Translation and Compilation is pure overhead, and independent of input size or number of cores. While Execution scales with the input size and number of cores, the overhead becomes less and less important with increasing input size. *Execution* ist the time for executing the compiled ArBB program, which includes fetching the input data from the R environment, doing the actual computation and writing the results back to R.

While at 10K and 100K the overhead consumes most of the time, at 1M elements execution takes over. Since execution takes ZMQ data transfer (see Chapter 6.1.4) into account and ArBB itself does not scale perfectly linear (see Chapter 6.1.5), we also

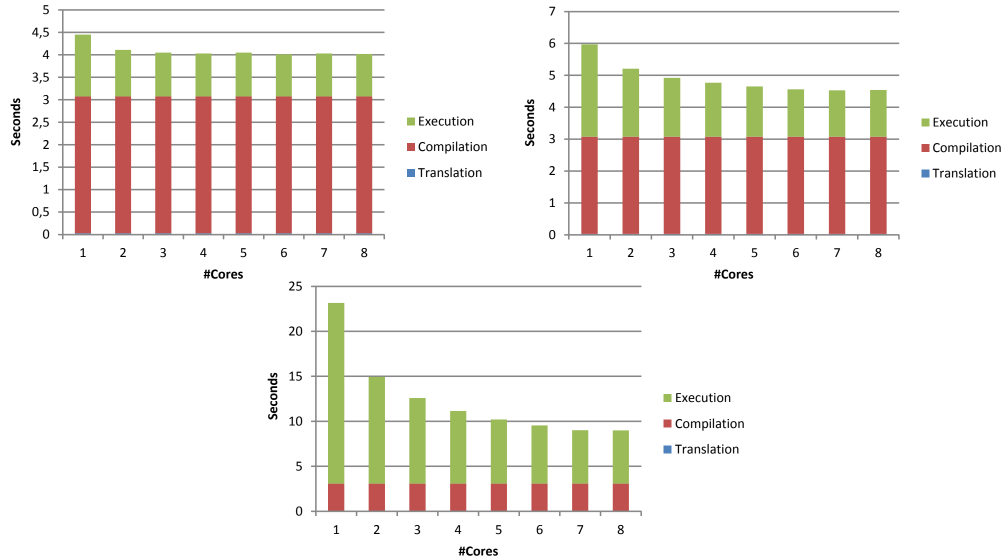

Figure 25: myfun ArBB, 10K, 100K, 1M elements, runtime partition

cannot expect linear scaling for the execution part.

Total Runtime: Relative Speedup

To get an idea how the overall execution time of the backend scales with varying input sizes and number of CPU cores, we measured the relative speedup with these parameters. Figure 26 presents the relative speedup of the total execution time of the backend for the myfun example with 10K, 100K and 1M elements. Again, cores vary from 1-8. On all input sizes, the transformation and compilation steps take a large amount of time, which leads to suboptimal scaling even with 1M elements, where the transfer of results also takes a considerable amount of time. Maximum relative speedups with 8 cores are for 10K: 1.11, 100K: 1.32, 1M: 2.58.

Relative Speedup: Execution

Removing code translation and compilation from the equation, we get the plots which only include execution time of the backend executable. Figure 27 shows the relative speedup of the execution of the backend executable with cores varying from 1-8 for input sizes of 10K, 100K and 1M respectively. Since we removed the fixed overhead, we get slightly better numbers, but they are still far from optimal, which is due to the fact that the backend executable still has to transfer the input and output data between processes and the fact that ArBB itself does not scale linearly. For evaluation without

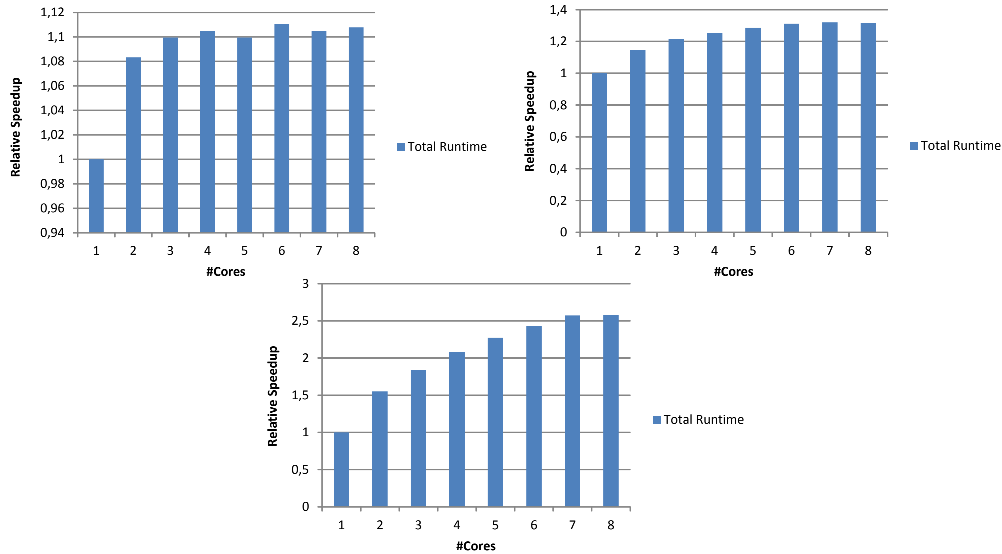

Figure 26: myfun ArBB, 10K, 100K, 1M elements, relative speedup total runtime

the data transfer, see Chapter 6.1.5.

Execution Scaling

Combining the information from above leads to Figure 28, which shows the relative speedup of the backend executable for input sizes 10K, 100K and 1M with 1-8 cores. The main observation here is, that our backend scales better the larger the input size is. Still 1M elements is not a huge input size for typical computations, which are done with R.

6.1.2. Levenshtein Distance

The Levenshtein distance computes a string metric for measuring the difference between two strings. It falls into the category of dynamic programming algorithms. The parallel version is created manually by using the MATSU[KMM+05] algorithm, which is able to parallelize dynamic programming problems. The parallel solution provided by MATSU makes use of the MAP, ZIPW and SCAN skeletons. The sequential R version is presented in Listing 7, the parallelzied version in Listing 8.

While the sequential code only touches every matrix element once, the MATSU parallel version is computationally far more expensive, which is only offset by the use of multiple CPU cores. It (sequentially) iterates over the rows of the input matrix, but applies four skeletons for every column of the matrix, which means iterating (in parallel) four times

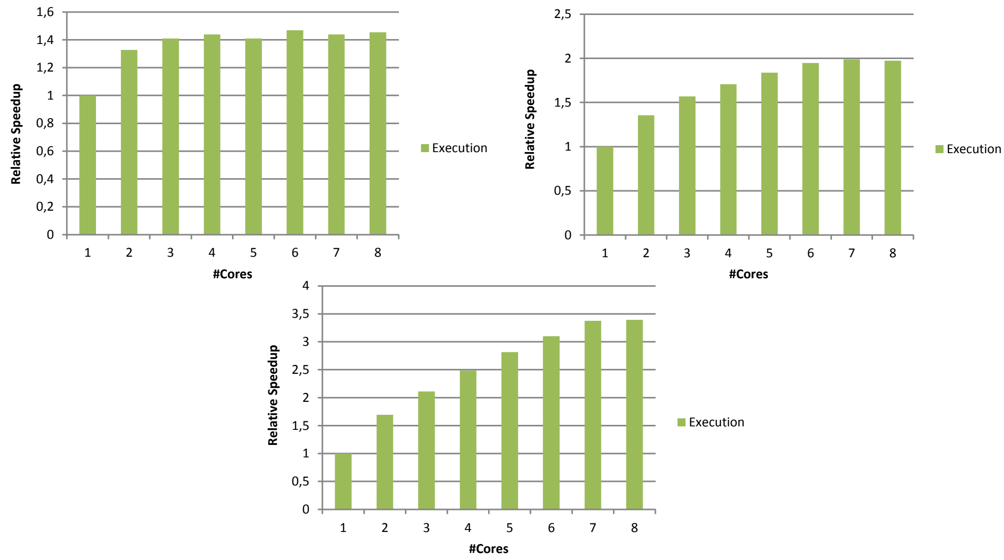

Figure 27: myfun ArBB, 10K, 100K, 1M elements, relative speedup execution only

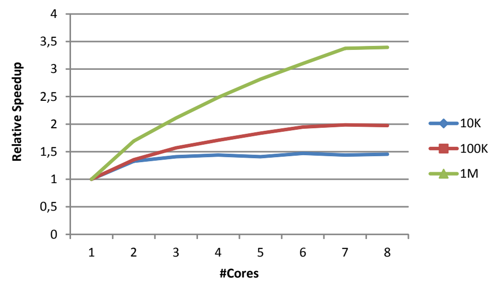

Figure 28: myfun ArBB, scaling execution only

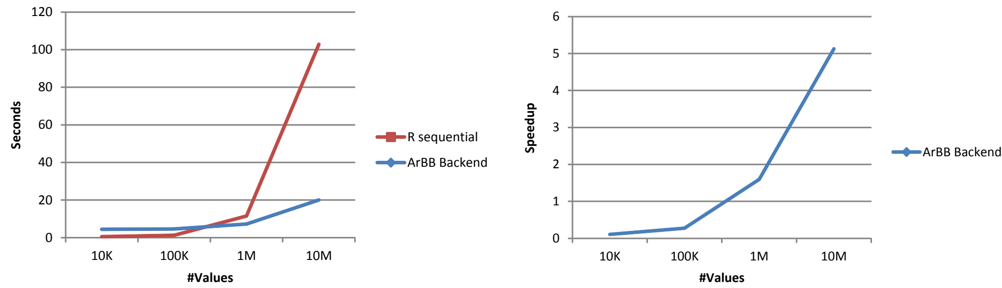

Figure 29: Levenshtein, total runtime and speedup

over each column. The SCAN operation on each column is even more expensive (cf. Chapter 5.4). While the myfun example already shows a very good speedup with 1M elements, R performs much better in terms of total runtime for the Levenshtein example (less work per element of the input vector). This leads to a heavier relative impact of the fixed costs for the parallelized version.

For both versions the execution times have been measured for string lengths of 10K, 100K, 1M and 10M, the results are plotted in Figure 29, the computed speedup to the sequential version can also be found in Figure 29. Using ALCHEMY for the computation only makes sense if the strings are at least a few 100K characters long, otherwise the overhead is too large. From there it gets more and more attractive with increasing string lengths, at 10M characters, we get a speedup of 5.13.

Looking at both versions of the program, the parallelized version is far more expensive in terms of total operations, but since it is parallel we still get a respectable speedup. Parallelizing dynamic programming algorithms is non-trivial and MATSU delivers such solutions. They are only effective when they can benefit from massive parallelism.

6.1.3. Longest Common Subsequence

Another dynamic programming problem is the search for the longest common subsequence (LCS) of two strings. Again we used MATSU to derive a non-trivial parallelization of the LCS algorithm. Listing 9 contains the sequential version, Listing 10 the parallelized version.

The MATSU parallelization of LCS is, similar to Levensthein, ~5-6 times more expensive than the sequential implementation. That means even without considering overhead costs, scaling could not come close to linear with the number of CPU cores. Also the gain from compilation is not as big, because the per-element operations are small.

We measured the execution times for the sequential and parallel versions using se-

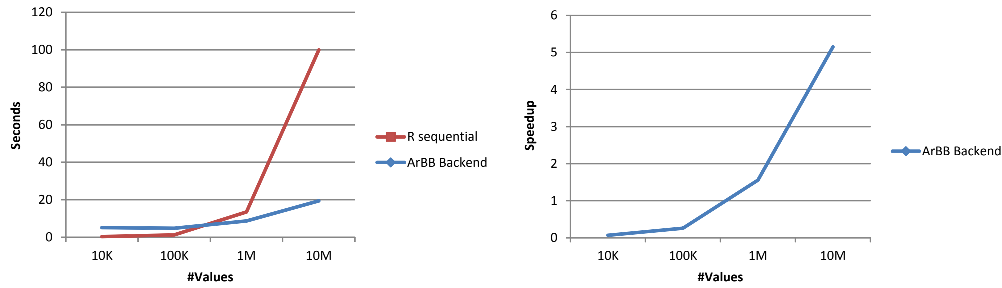

Figure 30: LCS, total runtime and speedup

quence lengths of 10K, 10K, 1M and 10M. Since the computationally intensive part of the MATSU output looks very similar to the one of the Levenshtein distance example, the results are similar, the maximum speedup at 10M elements is 5.15. Figure 30 also shows that choosing the parallelized version starts to get interesting with several 100K elements and becomes increasingly better with more elements.

6.1.4. ZMQ Transfer Rates

ZMQ is used to transfer input data from R/ALCHEMY to the backend executable and write computation results back. Since this means copying data back and forth, it does not come for free. To find out how much of a bottleneck ZMQ can be, we evaluated the transfer rates using the EnvironmentService and an ArBB program. Since the main transfer primitive of ZMQ is a string, all our values are transferred as strings, which means the transfer speed is not only dependent on the number of values but also on the number of digits of the values.

Figure 31 shows the time in seconds it takes to transfer 1M/10M values with 1 to 10 digits via ZMQ. Transferring 1M values with 10 digits takes 1.15s, 10M takes 11.44s. In both cases this is 37% more than transferring 1 digit. Scaling is nearly linear with the number of values and fortunately much better with increasing digits.

While we already get good overall results, ZMQ is still a bottleneck and will be replaced with a shared memory approach in the future, at least for cases when both processes reside on the same machine.

6.1.5. ArBB Scaling

If we could reduce the overhead of our backend to zero, remove it completely, how fast would it be? Then it all comes down to how fast ArBB is. So we took the myfun example and removed the translation, compilation and data transfer steps, which results in an

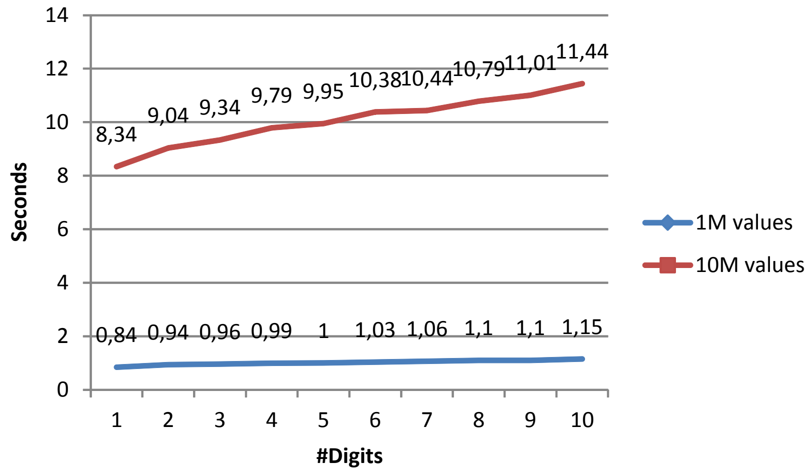

Figure 31: ZMQ transfer speed

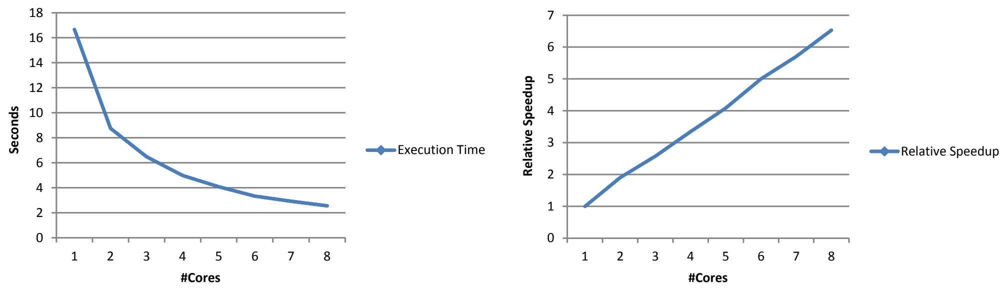

Figure 32: ArBB scaling, 1M elements, execution time and relative speedup

ArBB program which just does the parallel computation on 1M elements. Again we tested it on 1-8 cores.

Execution on a single core takes 16.66s, on 8 cores it takes 2.55s (see Figure 32). The relative speedup goes up a straight line to 6.53 on 8 cores, as can be seen in Figure 32. This marks the hard limit of ArBB on what could be expected in terms of scaling, even if there was no overhead from our backend. This only applies to arbb::map, which is used in the myfun example. But since all our skeletons rely on arbb::map as their parallel primitive, these numbers apply to all of our use cases.

6.1.6. Runtime Composition

This section describes the pieces that contribute to the total runtime of a backend execution. Since our backend contains many stages, they all contribute their part to the total runtime, some of these parts are fixed overhead, others scale with the input size:

Code translation The first step in the backend chain, which consists of the type inference and code translation from AIR to ArBB. Its runtime heavily depends on how much symbol lookups are made to the R environment. Typically it ranges between 1 and 200ms.

Compilation The generated ArBB code gets compiled by a C++ compiler, which typically takes 2-5s.

Fetch input When the ArBB program gets executed, it needs to fetch its input data from R/ALCHEMY via ZMQ, this entirely depends on how much data needs to be fetched.

Computation After the input data is available, the actual computation happens, this also depends on the program and the size of the input data.

Transfer output After the computation, the computation result, as well as all intermediate results are transferred back to R/ALCHEMY. The runtime depends on the size of the output data.

6.2. Worst Case Speedups

Being able to estimate when it starts to make sense to use ALCHEMY for parallelizing R programs is important. Therefore we measured our skeletons with the most simple kernels to get a "worst-case" scenario, where the fixed overhead and ZMQ transfer costs have a very high share of the total runtime. This gives a rough estimate, which input sizes are required for our ArBB backend to be faster then a sequential execution with R.

6.2.1. MAP Skeleton

The sequential version is shown in Listing 13, the parallelized version in Listing 14. Input sizes are 10K, 100K, 1M and 10M elemenents. Figure 33 compares the total runtime in seconds of the sequential against the parallelized version with the given input sizes. The ArBB backend version starts to become faster between 1M and 2M elements and reaches to a speedup of 3.08 at 10M values, as can be seen in Figure 33. So this represents the worst case, where the most simple kernel function forms the closest gap between R sequential execution and our backend. With more complex kernel functions, the benefit of our backend grows rapidly, shifting the break-even point to smaller input sizes, in favor of our backend. Also, the speedup grows faster with increasing input values.

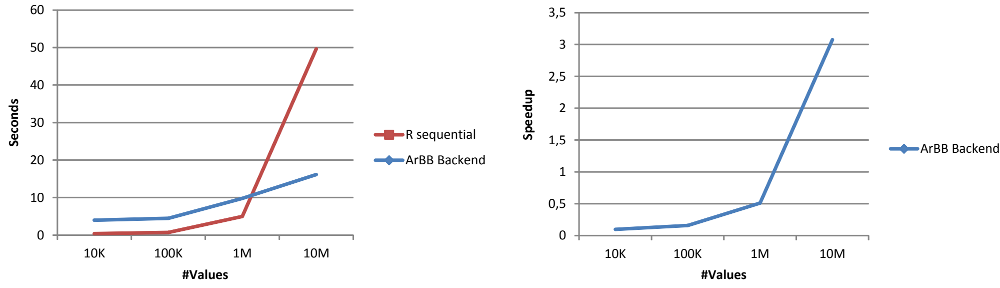

Figure 33: MAP skeleton, total runtime and speedup

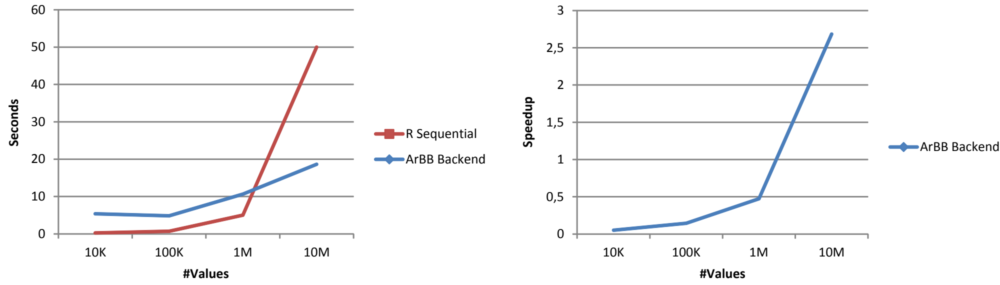

Figure 34: SCAN skeleton, total runtime and speedup

6.2.2. SCAN Skeleton

Analogous to the evaluation of the MAP skeleton, we conducted a worst-case performance analysis for the SCAN skeleton, input sizes were 10K, 100K, 1M and 10M values. Listings 15 and 16 show the sequential and parallelized versions of our SCAN example.

Since the SCAN skeleton is also based on the arbb::map primitive it performs similar, although the added cost of the code performing the SCAN operation, leads to slightly worse results, which is shown in Figure 34. This puts the break-even point around 2-3M elements, which again marks the worst-case scenario for use of the SCAN skeleton. Increasing the complexity of the SCAN kernel or increasing the input size will always favor the use of our backend. Figure 34 shows the speedup of our backend with 10M elements at 2.68 in comparison to the R sequential code as baseline.

6.2.3. ZIPW Skeleton

The ZIPW skeleton is very similar in nature to the MAP skeleton, except that it takes two input vectors and a combining kernel function. The sequential and parallel source programs are shown in Listings 17 and 18. Both programs were evaluated with input sizes of 10K, 100K, 1M and 10M elements for both input vectors.

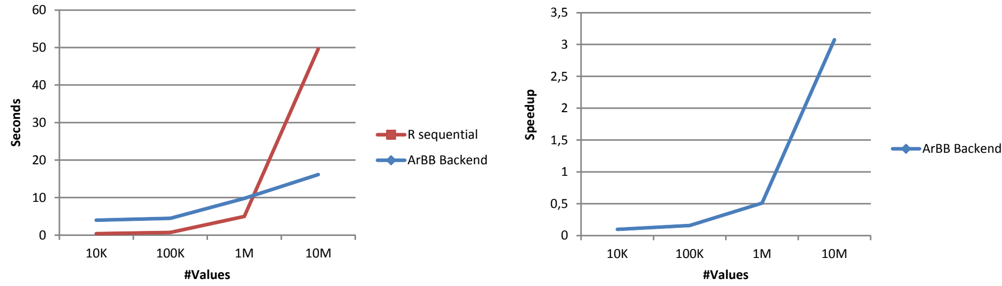

Figure 35: ZIPW skeleton, total runtime and speedup

Unsurprisingly, the ZIPW skeleton performs similar to the MAP skeleton, our backend also becomes viable starting between 1-2M elements for the input vectors. Figure 35 shows a comparison of the total runtime of the backend and R sequential versions. At 10M elements, a speedup of 3.05 can be achieved using 8 CPU cores, which can be seen from Figure 35. Increasingly complex kernel functions become even more profitable with our backend.

6.2.4. DOPAR Skeleton

The DOPAR example comes from the SURE [D+98] approach, which is able to parallelize nested loops where the inner loop operates on matrices. It does so by analyzing dependencies and reordering and defining parallel loops. While a corresponding SURE PAM is in the works, it is not yet ready, so the example is derived by manually applying the SURE algorithm to our sequential R program, which can be found in Listing 11. The parallelized version is presented in Listing 12.

While SURE is not able to parallelize the outer loop of the input program, it rearranges the inner loop to avoid data dependencies, which then allows parallel execution. As with the MATSU examples, the parallelization gains dominate the gains from compilation, because the operations in the inner loop are cheap.

The DOPAR example has been measured in the sequential and parallelized version with square matrices containing 10K, 100K, 1M and 10M elements. Comparing the runtimes of the sequential and parallelized version in Figure 36, shows that the break-even point is around 1M. So anything above a 1000x1000 matrix should benefit from the parallelized version performance-wise. The speedup with 10M elements is 4.17, as shown in the speedup plot in Figure 36.

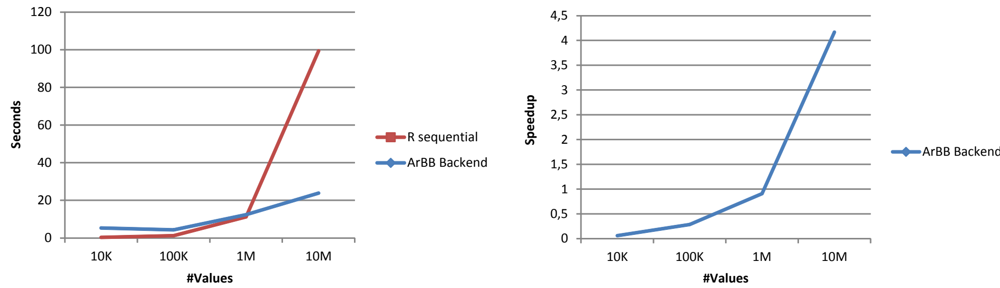

Figure 36: DOPAR example, total runtime and speedup

7. Conclusion and Outlook

This thesis describes the concepts, design, realization and evaluation of a backend for parallel execution of AIR programs for the ALCHEMY platform using Intel ArBB. A transmutator for code translation has been created along with a C++ library that implements R primitives and the parallel skeletons using ArBB code, as well as supporting mechanisms like the ZMQ communication. Also the R/ALCHEMY and ALCHEMY services EnvironmentService and ValueService have been extended to support writing back computation results from the backend executable, which was not provided. Their performance was vastly improved by implementing a plain communication protocol for transferring large data instead of using XML.

While many of the essential R primitives have been implemented, the uncommoner primitives are not yet implemented. The main mathematical functions habe been covered, but some of the statistical functions are missing. At least the R base package is already planned for implementation.

For the code translation to be able to complete the type inference, it requires the function signatures of built-in or library/package functions. Our backend has the RBuiltinFunctions database to hold this information. While it is easily extensible, it currently only holds information about the R base package. It would also be helpful to cover the most common R packages here to support a wider range of R input programs.

Another possible improvement is the reduction of the overhead caused by our backend execution. While we have some fixed overhead coming from the code translation and more important from the compilation of the backend executable, which we cannot remove, the data transfer between the components is a place to start. In most cases, the three involved processes live on the same machine, so there is no need for network communication. A shared memory approach would most certainly bring a huge performance improvement and is planned as future work.

While we currently have support for the MAP, SCAN, ZIPW and DOPAR skeletons, which is all we currently need with the available and currently developed PAMs, at some point there will be other parallelization strategies which require new skeletons. Our backend will be ready for further skeletons and allows to easily implement and incorporate new parallel skeletons.

We already assessed the performance of our backend with a few examples, but the performance in the field yet remains to be seen. When the currently developed MATSU and SURE PAMs are ready for use in a transmutation chain, it will be much easier to expose our backend to new parallelized programs. That way we get a better understanding of its performance in more complex scenarios, in which, as the measurements suggest, it will perform even better.

References

[ADH$^+$08] E. Ayguade, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. An Experimental Evaluation of the New OpenMP Tasking Model. *Languages and Compilers for Parallel Computing*, 1:63–77, 2008.

[Ble90] G. E. Blelloch. Prefix Sums and Their Applications. *Synthesis of Parallel Algorithms*, 1:35–60, 1990.

[Col91] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1991.

[D$^+$98] A. Darte et al. Mathematical Tools for Loop Transformations: From Systems of Uniform Recurrence Equations to the Polytope Model. *Algorithms for Parallel Processing*, 105:147–183, 1998.

[DM98] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE*, 5:46–55, 1998.

[GHJV01] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Abstraction and reuse of object-oriented design.* Springer, 2001.

[Gor96] S. Gorlatch. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In *International Euro-Par Conference on Parallel Processing*, EUROPAR, pages 401–408, 1996.

[Hin11] P. Hintjens. ZeroMQ-The Guide. `zguide.zeromq.org/local--files/main:_start/zguide-c.pdf`, 2011.

[HS86] W. D. Hillis and G. L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29:1170–1183, 1986.

[HSO07] M. Harris, S. Sengupta, and J.D. Owens. Parallel Prefix Sum (Scan) with CUDA. *GPU Gems*, 3(39):851–876, 2007.

[KMM+05] K. Kakehi, K. Matsuzaki, A. Morihata, K. Emoto, and Z. Hu. Parallel Dynamic Programming Using Data-Parallel Skeletons. *Nihon Sofutowea Kagakukai Taikai Koen Ronbunshu*, 22:4B–1, 2005.

[KSG09] Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch. Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-Cores. In *International Euro-Par Conference on Parallel Processing*, EUROPAR, pages 654–665, 2009.

[LF80] R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27:831–838, 1980.

[MAL+09] M. Morgan, S. Anders, M. Lawrence, P. Aboyoun, H. Pagès, and R. Gentleman. Shortread: a bioconductor package for input, quality assessment and exploration of high-throughput sequence data. *Bioinformatics*, 25(19):2607–2608, 2009.

[McC09] M. McCool. Dr. Dobbs Go Parallel - Structured Patterns for Parallel Computation. `http://drdobbs.com/go-parallel/article/showArticle.jhtml?articleID=222000451`, 2009.

[MD12] M. Morgan and N. Delhomme. R/Bioconductor for High-Throughput Sequence Analysis, 2012.

[Mir11] M. Mirold. ALCHEMY - An Experimentation Laboratory for the Automatic Parallelization of Programs Written in the R Language. Master's thesis, Saarland University, 2011.

[MLS07] X. Ma, J. Li, and N.F. Samatova. Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing. In *International Parallel and Distributed Processing Symposium*, IPDPS, pages 1–6, 2007.

[Mun11] Ed. A. Munshi. The OpenCL Specification, Version: 1.1, Document Revision: 44. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, 2011.

[NSL+11] C.J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S.D. Toit, Z.G. Wang, Z.H. Du, Y. Chen, G. Wu, et al. Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language. In *International Symposium on Code Generation and Optimization*, CGO, pages 224–235, 2011.

[PM12] F. Padberg and M. Mirold. An Experimentation Platform for the Automatic Parallelization of R Programs. In *Asia-Pacific Software Engineering Conference*, APSEC, pages 203–212, 2012.

[SGS10] J.E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering*, 12:66–73, 2010.

[SHZO07] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Symposium on Graphics Hardware*, GH, pages 97–106, 2007.

[SLO06] S. Sengupta, A.E. Lefohn, and J.D. Owens. A Work-Efficient Step-Efficient Prefix Sum Algorithm. In *Workshop on Edge Computing Using New Commodity Architectures*, EDGE, pages 26–27, 2006.

[SME+09] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State-of-the-art in Parallel Computing with R. *Journal of Statistical Software*, 47(1), 2009.

[Urb11] S. Urbanek. Multicore: Parallel Processing of R Code on Machines with Multiple Cores or CPUs. `http://www.rforge.net/multicore/index.html`, 2011.

[Wik11] Wikipedia. Algorithmic Skeleton — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/Algorithmic_skeleton`, 2011.

A. Generated ArBB Code Example

This is an example of the code generated by the R input:

```
1  fun<-function(x){return(x*2)}
2  x<-c(1,2,3)
3  x<-alchemy.applySkeleton("MAP",fun,x)
```

```
1  //INCLUDES
2  #include "RtoArBBLib.hpp"
3
4  //FORWARD DECLARATIONS
5  void fun_void(arbb::f32& result, arbb::f32 x);
6  arbb::f32 fun(arbb::f32 x);
7
8  //FUNCTIONS
9  arbb::f32 fun(arbb::f32 x)
10 {
11   RETURN_MACRO(x * 2);
12 }
13
14 void fun_void(arbb::f32& result, arbb::f32 x)
15 {
16   result = fun(x);
17 }
18
19 void arBBMain(arbb::dense< arbb::f32 >& x)
20 {
21   x = MAP_HELPER(fun_void, x);
22 }
23
24 //MAIN
25 int main( int argc, const char* argv[] )
26 {
27   //DEFINE VARIABLES
28   //FETCH VARIABLES
29   ZMQConnection fetch_conn("EnvSvc.getVal", "tcp
        ://127.0.0.1:1985");
```

```
30    arbb::dense< arbb::f32 > x = fetch_conn.
         receive_arbb_vector_float(564764935, "x");
31
32    //CALL PROGRAM
33    arbb::num_threads(8);
34    arbb::call(arBBMain)(x);
35
36    //TRANSMIT RESULTS
37    ZMQConnection trans_conn("EnvSvc.install", "tcp
         ://127.0.0.1:1985");
38    trans_conn.send_arbb_vector(x, 564764935, "x");
39  }
```

Listing 5: Generated ArBB code example

B. Evaluation Code Listings

```
1   myfun <- function (vec) {
2     helper <- function (x) {
3       sum <- 0
4       for (i in 1 :100 ) {
5         sum <- sum + 1/( sin(x) + i*cos(x))^2
6       }
7       return (sum )
8     }
9     alchemy.applySkeleton("MAP", helper, vec )
10  }
11
12  myfun (1:< numelements >)
```

Listing 6: “myfun” example

```
1   levenshtein<-function(x,y)
2   {
3     D<-array(dim=c(length(x),length(y)))
4     for (i in 1:length(x))
5     {
```

```
6      D[i,1]<-i-1
7    }
8    for (j in 1:length(y))
9    {
10     D[1,j]<-j-1
11   }
12   for (k in 2:length(x))
13   {
14     for(l in 2:length(y))
15     {
16       if (x[k] == y[l])
17       {
18         cost<-0
19       }
20       else
21       {
22         cost<-1
23       }
24       D[k,l] = min(D[k-1,l]+1,D[k,l-1]+1,D[k-1,l-1]+cost)
25     }
26   }
27   return(D[length(x),length(y)])
28 }
29 a<-rep("x",<size>)
30 b<-rep("y",<size>)
31 levenshtein(a,b)
```

Listing 7: Levenshtein distance, sequential

```
1 a<-rep("x",<size>)
2 b<-rep("y",<size>)
3 D<-array(0, dim=c(N,N))
4 for (i in 1:N)
5 {
6   D[i,1]<-i-1
7 }
8 for (j in 1:N)
```

```
9  {
10   D[1,j]<-j-1
11 }
12 add<-function(x) {return(x+1)}
13 del<-function(y) {return(y+1)}
14 repp<-function(k,l) {return(min(k+1,l))}
15 minn<-function(m,n){return(min(m,n))}
16 alchemy.loglevel(6)
17 alchemy.enable()
18 for (o in 2:N)
19 {
20   A<-shift(D[,o-1],1 , 0)
21   B<-alchemy.applySkeleton("MAP", del, D[,o-1])
22   C<-alchemy.applySkeleton("MAP", add, A, B)
23   D[,o]<-alchemy.applySkeleton("ZIPW", minn, B, C)
24   D[,o]<-alchemy.applySkeleton("SCAN",repp, D[,o])
25 }
```

Listing 8: Levenshtein distance, parallelized

```
1 N<-<size>
2 M<-<size>
3 D<-1:(N*M)
4 dim(D)<-c(N,M)
5 for(i in 2:N)
6   for(j in 2:M)
7     D[i,j] <- max(max(D[i,j-1]+2, D[i-1,j]+3), D[i-1,j-1]+1)
```

Listing 9: LCS, sequential

```
1 N<-<size>
2 M<-<size>
3 E<-1:(N*M)
4 dim(E)<-c(N,M)
5 maxx<-function(a,b){return(max(a,b))}
6 plus2<-function(c){return(c+2)}
7 plus1<-function(d){return(d+2)}
8 max3<-function(e,f){return(max(e+3,f))}
```

```
9  alchemy.loglevel(6)
10 alchemy.enable()
11 for (j in 2:M)
12 {
13   A<-alchemy.applySkeleton("MAP", plus2, E[,j-1])
14   B<-alchemy.applySkeleton("MAP", plus1, E[,j-1])
15   C<-shift(B,1,0)
16   D<-alchemy.applySkeleton("ZIPW", maxx, A, C)
17   E[,j]<-alchemy.applySkeleton("SCAN", max3, D)
18 }
```

Listing 10: LCS, parallelized

```
1  N<-<size>
2  M<-<size>
3  a<-array(1,dim=c(N,M))
4  b<-array(2,dim=c(N,M))
5  c<-array(3,dim=c(N,M))
6  for (i in 2:N)
7  {
8    for (j in 2:M)
9    {
10     B[i,j]<-A[i-1,j]
11     C[i,j]<-B[i,j-1]
12     A[i,j]<-C[i,j-1]
13   }
14 }
```

Listing 11: DOPAR example, sequential

C. Numerical Evaluation Results

These are the numerical results of the evaluation:

D. Sequential Skeleton Implementations

```
1  map<-function(fun, vec)
2  {
```

Digits	time/s
1	0.84
2	0.94
3	0.96
4	0.99
5	1.0
6	1.03
7	1.06
8	1.1
9	1.1
10	1.15

Table 5: ZMQ transfer rates, 1M values

Digits	time/s
1	8.34
2	9.04
3	9.34
4	9.79
5	9.95
6	10.38
7	10.44
8	10.79
9	11.01
10	11.44

Table 6: ZMQ transfer rates, 10M values

10K	100K	1M
4.55	28.15	299.4

Table 7: Myfun sequential, total runtime in seconds

10K	100K	1M
1.8	6.95	56.65

Table 8: Myfun RMulticore, total runtime in seconds

Cores	10K	100K	1M
1	4.42	59.4	23.12
2	4.08	5.18	14.9
3	4.02	4.89	12.55
4	4.00	4.74	11.12
5	4.02	4.62	10.17
6	3.98	4.53	9.52
7	4.00	4.50	8.99
8	3.99	4.51	8.96

Table 9: Myfun ArBB, total runtime in seconds

	10K	100K	1M	10M
R sequential	0.38	0.71	4.98	49.65
ArBB backend	3.95	4.47	9.79	16.14

Table 10: MAP skeleton, total runtime in seconds

	10K	100K	1M	10M
R sequential	0.27	0.7	5.03	50.01
ArBB backend	5.36	4.82	10.65	18.63

Table 11: SCAN skeleton, total runtime in seconds

	10K	100K	1M	10M
R sequential	0.41	0.74	5.00	49.87
ArBB backend	3.98	4.52	9.94	16.63

Table 12: ZIPW skeleton, total runtime in seconds

	10K	100K	1M	10M
R sequential	0.32	1.21	11.24	99.35
ArBB backend	5.28	4.27	12.39	23.84

Table 13: DOPAR skeleton, total runtime in seconds

	10K	100K	1M	10M
R sequential	0.34	1.21	13.51	99.87
ArBB backend	5.14	4.74	8.68	19.39

Table 14: LCS example, total runtime in seconds

```
1   N<-<size>
2   M<-<size>
3   A<-array(1,dim=c(N,M))
4   B<-array(2,dim=c(N,M))
5   C<-array(3,dim=c(N,M))
6   fun1<-function(){B[i,j]<-A[i-1,j]}
7   fun2<-function(){C[i,j]<-B[i,j-1]}
8   fun3<-function(){A[i,j]<-C[i,j-1]}
9   for (i in 2:N)
10  {
11          alchemy.applySkeleton("DOPAR","j",2,M,i,fun1)
12          alchemy.applySkeleton("DOPAR","j",2,M,i,fun2)
13          alchemy.applySkeleton("DOPAR","j",2,M,i,fun3)
14  }
```

Listing 12: DOPAR example, parallelized

```
1   x<-1:<size>
2   for (i in 1:<size>)
3   {
4     x[i]<-x[i]+1
5   }
```

Listing 13: MAP, sequential

```
3     fun(vec) //implicit vector application by R
4   }
5
6   zipw<-function(fun, v1, v2)
7   {
8     fun(v1,v2) //implicit vector application by R
9   }
10
11  scan<-function(fun, vec)
```

	10K	100K	1M	10M
R sequential	0.46	1.26	11.51	102.76
ArBB backend	4.36	4.62	7.21	20.04

Table 15: Levenshtein example, total runtime in seconds

```
1 x<-1:<size>
2 fun<-function(x){return(x+1)}
3 x<-alchemy.applySkeleton("MAP",fun,x)
```

Listing 14: MAP, parallelized

```
1 x<-1:<size>
2 for (i in 2:<size>)
3 {
4   x[i]<-x[i]-x[i-1]
5 }
```

Listing 15: SCAN, sequential

```
12 {
13   out <- rep(NA, length(vec))
14   out[1] <- vec[1]
15   for(i in 2:length(vec))
16   {
17     out[i] <- fun(out[i-1],vec[i])
18   }
19   return(out)
20 }
```

Listing 19: Sequential skeleton reference implementations

E. AIR/SEXPR Examples

```
1 <AIR>
2   <Program>
3     <FuncCall>
4       <funcexpr>
5         <SymbolExpr name="sin"/>
6       </funcexpr>
7       <params>
8         <ParamExpr>
9           <FuncCall>
10            <funcexpr>
```

```
1  z<-1:<size>
2  kernel<-function(x,y){return(x-y)}
3  alchemy.applySkeleton("SCAN",kernel,z)
```

Listing 16: SCAN, parallelized

```
1  x<-1:<size>
2  y<-1:<size>
3  for (i in 1:<size>)
4  {
5    z[i]<-x[i]+y[i]
6  }
```

Listing 17: ZIPW, sequential

```
11                 <SymbolExpr name="c"/>
12               </funcexpr>
13               <params>
14                 <ParamExpr>
15                   <ConstantExpr type="real">
16                     <RealValue data="1.000000"/>
17                   </ConstantExpr>
18                 </ParamExpr>
19                 <ParamExpr>
20                   <ConstantExpr type="real">
21                     <RealValue data="2.000000"/>
22                   </ConstantExpr>
23                 </ParamExpr>
24                 <ParamExpr>
25                   <ConstantExpr type="real">
26                     <RealValue data="3.000000"/>
```

```
1  x<-1:<size>
2  y<-1:<size>
3  fun<-function(x,y){return(x+y)}
4  z<-alchemy.applySkeleton("ZIPW",fun,x,y)
```

Listing 18: ZIPW, parallelized

```
27                  </ConstantExpr>
28                </ParamExpr>
29              </params>
30            </FuncCall>
31          </ParamExpr>
32        </params>
33      </FuncCall>
34    </Program>
35  </AIR>
```

Listing 20: AIR XML of sin example

```
1  LANGSXP (
2    CAR:
3      SYMSXP (
4        pname:
5          CHARSXP ("{")
6        symvalue:
7          SPECIALSXP (funtabidx: 11)
8        internal:
9          NILXP ()
10     )
11   CDR:
12     LISTSXP (
13       CAR:
14         LANGSXP (
15           CAR:
16             SYMSXP (
17               pname:
18                 CHARSXP ("sin")
19               symvalue:
20                 BUILTINSXP(funtabidx: 156)
21               internal:
22                 NILXP ()
23             )
24           CDR:
25             LISTSXP (
```

```
26              CAR:
27                LANGSXP (
28                  CAR:
29                    SYMSXP (
30                      pname:
31                        CHARSXP ("c")
32                      symvalue:
33                        BUILTINSXP(funtabidx: 87)
34                      internal:
35                        NILXP ()
36                )
37              CDR:
38                LISTSXP (
39                  CAR:
40                    REALSXP (1.000000)
41                  TAG:
42                    NILXP ()
43                  CDR:
44                    LISTSXP (
45                      CAR:
46                        REALSXP (2.000000)
47                      TAG:
48                        NILXP ()
49                      CDR:
50                                    LISTSXP (
51                          CAR:
52                            REALSXP (3.000000)
53                          TAG:
54                            NILXP ()
55                          CDR:
56                            NILXP ()
57                        )
58                    )
59                )
60            )
61          TAG:
```

```
62              NILXP ()
63            CDR:
64              NILXP ()
65          )
66      )
67    TAG:
68      NILXP ()
69    CDR:
70      NILXP ()
71    )
72  )
```

Listing 21: R SEXPR of sin example