An Experimentation Platform for the Automatic Parallelization of R Programs

Frank Padberg
Faculty of Informatics
Karlsruhe Institute of Technology KIT
Karlsruhe, Germany
frank.padberg@kit.edu

Michael Mirold
Mirold Softwareentwicklung
Fürth, Germany
mail@michael-mirold.de

***Abstract*— We present our ALCHEMY platform that supports the automatic parallelization of R programs during execution. Parallelization occurs fully transparent to the user. Different parallelization techniques can be implemented as modules, linked into the platform, and combined with each other. The parallelization analysis modules and code transformation modules use a new intermediate representation for sequential and parallelized R code. Successfully parallelized parts of the R program are executed on a multicore processor; the results and the remaining sequential parts are fed back into the standard R interpreter and evaluated to completion. This way, an R user can benefit from multiprocessor performance without writing a single line of parallel code. At this stage of the research project, the main goal is to enable ample experimentation with different approaches to the automatic parallelization of scripting languages such as R.**

Automatic parallelization; R language; scripting languages; data-parallelism; parallel intermediate languages

I. INTRODUCTION

Scripting languages such as R [2][3] and Matlab are frequently used in science and engineering practice. They are mostly used by end users who are experts in their application domain, such as bioinformatics or electrical engineering, but who are no experts in programming. The application problems often require repeated computations with large amounts of data, resulting in long response times. For example, bioinformatics applies standard string matching algorithms to find common segments in genome sequences that consist of millions of characters. R has become a preferred programming tool in bioinformatics; see the recent efforts of the Bioconductor project [1].

R is an *interpreted* language that was not built with such tough processing requirements in mind. Yet, its interactive nature, its graphics capabilities, and the many special-purpose libraries that are available make it an attractive programming tool and environment for practitioners. As a solution to their growing performance requirements, it is natural that R practitioners want to take advantage of the computing power of modern multicore processors. This is not easy, though, because it requires parallel programming skills.

Writing parallel programs is known to be difficult and error-prone, and it requires not only specific knowledge of parallel algorithms, parallel data-structures, and when to best use them, but also a deep understanding of synchronization mechanisms, concurrency defects, hardware characteristics, cache effects, etc. In addition, support for parallel programming in R currently is limited. In essence, one must use an R frontend to one of the well-known, low-level libraries for parallel programming, such as MPI or PVM; see section II. This seems a mismatch with the high-level programming features offered by R.

The situation calls for the *automatic parallelization* of sequential R programs, as an alternative solution. The burden of hand-writing parallel code should best be avoided by an R application programmer. This view is underlined by the fact that R practitioners typically do not want to spend much time on development, but want to get their computation results quickly; this desire actually is a major driver for using a scripting language in the first place. In addition, a perfectly optimal usage of the parallel hardware typically is not required; a good performance and reasonable speedup on a standard multicore personal computer often will help in practice.

In this paper, we focus on the automatic parallelization of R programs, expecting that our findings will be useful for other scripting languages as well, including Matlab. We consider R to be very suitable for this kind of research: The R language offers many functions that already operate on vector or matrix data. These functions are naturally used by programmers and offer immediate potential for parallelization. Also, R does not provide pointers, which makes static code analysis much easier. R programs tend to be short and highly modular. Finally, the complete R environment and interpreter are available as source code.

More specifically, the goal of our research project is to build a *platform* – ALCHEMY – that allows us to *experiment* with the automatic parallelization of R programs. We want to try out different approaches and techniques for detecting and exploiting potential parallelism in a piece of R code. The best parallelization depends on the specific program and input data, of course; hence, a single parallelization technique will not do for all programs. We also want to *combine* different approaches and study experimentally if and how they might fit together.

For example, it might make sense to first transform the key algorithm of an R program (say, a string matching algorithm in bioinformatics) into a parallel version using parallel design patterns, then add an analysis that looks for remaining data-parallel operations that can be efficiently transformed into parallel code. Additional code transformations usually follow that compile the parallel design patterns into code that can be immediately executed on the target multicore processor, using a library such as Intel's TBB or OpenMP. Fig. 1 shows a simplified data-flow diagram of the workflow in our platform, starting from the R program entered by the user and ending with its execution on a parallel backend.

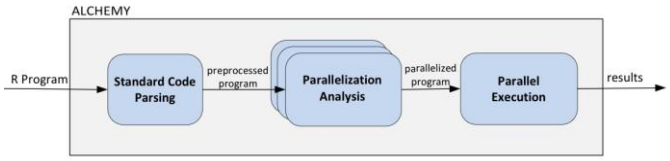

Figure 1. Overview of the ALCHEMY workflow

The key concept in ALCHEMY that allows us to combine several parallelization analyses and code transformations in a pipeline fashion is a new, common intermediate representation for (sequential and parallelized) R programs, the AIR ("analysis intermediate representation"). AIR offers parallel skeletons [13] as primitives, the basic R operations, and vectors of arbitrary length similar to the VCODE intermediate language [16], see section III.B. The original R program is automatically translated into AIR code, then analyzed and transformed by the different parallelization modules. Every parallelization module takes AIR as input and produces AIR as output. Clearly, not all analysis modules will use all features available in AIR; it depends on the specific analysis which features are present in its output. Basically, all analysis modules can be connected to each other. When starting ALCHEMY, a particular configuration is put into place using a configuration file.

A particular difficulty for the automatic parallelization of R stems from the fact that R is an interpreter, not a compiler, and is used interactively during program development. Hence, we must analyze, transform, and execute the code that was typed in by the user at the R console (or read in from a file) "on the fly," intercepting the normal flow of evaluation inside the R kernel. We solve this problem by re-programming the inner R interpreter loop in *one* location. We redirect control to our platform and resume normal interpreter operation after the automatically parallelized parts of the R program have been executed on the parallel backend; see section III.C for more details.

In the paper, we present the architecture and key design elements of the ALCHEMY platform, including the new AIR intermediate language and the interface between ALCHEMY and the R interpreter. We illustrate the internal workflow of ALCHEMY using a simple program example. We describe how control flows from the R interpreter to ALCHEMY and back. Finally, we present some early measurements that show the speedup gained by a simple analysis module that automatically detects data-parallel R operations and executes them in parallel on a multicore processor. We would like to point out that these examples and measurements serve illustrative purposes. In particular, they show that the core of the ALCHEMY platform is fully functional and can be used transparently by the end user.

II. Related Work

Currently, R users who want to take advantage of parallel hardware must *explicitly* write parallel code using one of the available parallel libraries for R, such as the well-known R implementations of MPI ("Rmpi" [5]) and PVM ("rpvm" [6]). Other useful parallel programming packages for R include "snow" [7] and "R/parallel" [8]. "snow" offers uniform programming access to Rmpi, rpvm, NetWorkSpaces, and raw sockets. "R/parallel" offers a special "runParallel" command for loops that contain no data dependencies; the implementation is based on threads. Another popular library is the "R multicore package" [9] that offers a "pvec" command for data-parallel operations; the implementation is based on the "fork()" system call. For a good overview of parallel programming for the R language see [4].

There is a substantial body of knowledge on automatic parallelization, although mostly in the context of high performance computing for numerical or graphics applications; see f.e. [11] and [22]. In particular, there are powerful algorithms for parallelizing nested loops over matrices. In addition, there are tools that provide data dependence analysis and pointer analysis for languages such as C or Java, f.e., as part of the LLVM compiler framework [12]. Part of our motivation for building the ALCHEMY platform is to make such knowledge available for the parallelization of general R programs.

Automatic parallelization of arbitrary R programs is not available at this time. We are aware of only one approach ("pR" [10]) that automatically detects and refactors loops in R programs, but this applies only to loops that contain no data dependencies at all. Such loops are split automatically and the work pieces are distributed among worker R processes.

III. The ALCHEMY Platform

For the sake of brevity, we introduce the term "transmutator" to denote both parallelization analysis modules and code transformation modules. By definition, a transmutator takes AIR code as input, analyzes and/or transforms it somehow, and outputs AIR code. According to this definition, backend components that execute the final parallelized code on some parallel hardware are transmutators as well, since any backend must return the computation results in AIR format, even if it were just one number. Other than the input and output format, the ALCHEMY platform itself does not impose restrictions to what a transmutator can do internally. Clearly, to provide useful service a transmutator must preserve the semantics of its input program; but this requirement applies to any automated parallelization technique, or compiler.

A. *High-Level Design*

The ALCHEMY platform consists of several (potentially distributed) software components, see the UML component diagram in Fig. 2. The two main components are `RAlchemy` and `AlchemyCore`, see below. In addition, there are a number of separate transmutator components, including backends.

The various components can be deployed on different nodes in a network and communicate through a lightweight network protocol, ZeroMQ [18]. The idea was that we wanted to be able to use remote parallelization analyses, and remote backends. In addition, we wanted to copy application data only when necessary; hence, the data initially remains stored in the R process and gets delivered to the backend (or some intermediate analysis module) only upon request via the `ValueSvc` in the `RAlchemy` component. The same applies to things stored in the R environment, such as function definitions, which are fetched via the `EnvironmentSvc` from the `RAlchemy` component. The transmutators are implemented as services, too, following a simple `TransmutationSvc` API.

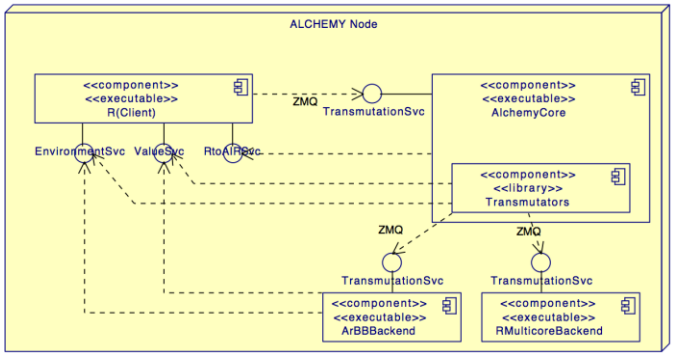

Figure 2. ALCHEMY components

The `RAlchemy` component provides an extension to the standard R kernel. In particular, it contains the interface `(AlchemyAdapter)` between ALCHEMY and the R inner interpreter loop (see subsection III.C below). In addition, it contains the value and environment services necessary for the communication with the transmutators, plus two converter services `(RtoAIRConverter, AIRtoRConverter)` that convert between the SEXPRs produced by the R parser and AIR code.

Being an extension to R, `RAlchemy` is written in C language. It consists of 15 modules, totaling to about 5 KLOC of code.

The `AlchemyCore` component implements the logic that controls the flow of AIR code between the transmutators (`TransmutationController`). The controller reads in a configuration file that describes which transmutators should be active under which conditions; a typical condition is that some particular other transmutator must have run to completion before. After receiving a transmutation request and corresponding AIR input from the `RAlchemy` component, the controller puts the transmutators into a queue and starts the first transmutator. Afterwards, it checks the conditions, possibly rearranges the queue, and starts the next transmutator in the queue. This occurs until all transmutators have finished. The AIR output of the last one is returned to the `RAlchemy` component.

`AlchemyCore` is written in Java. It consists of about 130 classes, totaling to about 10 KLOC of code, including the simple EMBA and RMulticoreBackend transmutators (see section IV), a small ZeroMQ server, and the classes that implement the AIR language.

B. *Intermediate Language*

We have developed a special intermediate language AIR ("analysis intermediate representation") for the ALCHEMY platform. The purpose is to provide a common code representation for all the transmutators in the platform, including the parallelization analysis modules, the code transformation modules, and the backend modules that wrap the parallel execution units. R's native SEXPR language [23] seemed too low-level and too remote from parallel features for this purpose. Similarly, concurrency support in current intermediate languages is limited, see [17].

Every transmutator takes AIR as input and produces AIR as output. Hence, in principle the transmutators can all be linked together to form a processing pipeline; of course, only certain combinations of parallelization analyses and code transformations make sense. But for an experimentation platform, it is important that it is easy to combine different parallelization approaches so that we can try out how well they complement each other. Forming processing pipelines also makes it easier to design stages within a parallelization analysis.

To be suitable as a common intermediate language for parallelization experiments in R, the AIR supports three distinct aspects of functionality:

- AIR supports the most relevant language features of R.
- AIR supports several data types of arbitrary length.
- AIR includes a number of parallel skeletons to express parallelism efficiently.

AIR supports most of the R language features, including control structures, user-defined functions, recursion, and the basic built-in functions. The UML class diagram in Fig. 3 (on the next page) shows the expression types that are currently supported by AIR.

One language feature of R that we deliberately do not support is the "eval"-statement, because this introduces some kind of self-modification capability into an R program and makes parallelization analysis difficult.

AIR supports the basic R data types for numbers and strings, including vectors and matrices. The UML class diagram in Fig. 4 (on the next page) shows the types that are currently supported by AIR.

AIR vectors and matrices are special: they can be of arbitrary size. This feature resembles the parallel intermediate language VCODE [16]. VCODE was designed to express data-parallel problems efficiently in environments that offer fast, native vector operations, such as a Cray. VCODE served as an intermediate language for NESL [19] which has influenced Intel's Ct technology [21]. Arbitrary length types proved useful in VCODE, and they align well with the vector and matrix data types in a scripting language such as R.

AIR also provides the special types "AIRVectorStorage" and "AIRMatrixStorage". They are *proxies* for the actual data. To avoid copying large chunks of application data back and forth between different transmutators, the actual data remains stored in the R environment and gets fetched only upon request via the `ValueSvc`. Any transmutator can fetch the data at any time. The "storage"-type values serve as placeholders in the meantime.

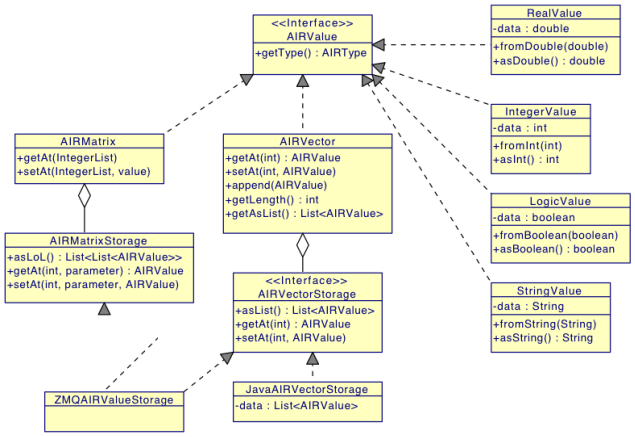

Figure 4. AIR value types

Building upon the basic types, we are currently extending the set of types supported by AIR to include the more sophisticated R data types, notably inhomogeneous lists and data frames; see [3] for more details on these R features.

To address parallelism in a program, AIR currently includes the most common parallel skeletons (type "SkeletonExpr" in Fig. 3), such as MAP, ZIP, ZIPW, SCAN, SHIFT, and DOPAR. For example, MAP applies a given function to a data vector element-wise; ZIPW merges two data vectors while applying a given function element-wise; DOPAR is a parallel loop; see [14] and [15] for more details. Using these skeletons, parallel solutions can often be expressed in a natural and compact form [13]. For example, two-dimensional parallel dynamic programming can be expressed efficiently without loops using list skeletons such as ZIPW and SCAN [20]. We add skeletons to the AIR whenever necessary to support a new parallelization technique. This currently is the most dynamic part of the AIR.

There are known efficient parallel implementations for each skeleton. An ALCHEMY configuration typically includes such implementations in the backend module for execution.

In addition, AIR comes with capabilities that allow writers of analysis or transformation modules to query and modify a piece of AIR code; more precisely, the corresponding abstract syntax tree. The queries are specified using the XPATH language based on a canonical representation of an AIR syntax tree as XML code (see also section IV for examples of this representation).

For example, the XPATH query expression

```
/Program/WhileStmt/body/*/
ForStmt[./ForCondition/@itervar="i"]
```

would specify a subtree of AIR code that contains a for-loop nested inside a top-level while-loop. Such specifications can be used both for searching and for modifying an AIR syntax tree.

C. *Interface to the R Interpreter*

R is an interpreted language. After startup, R enters its interpreter loop, called the REPL – read, evaluate, print, loop. When the user enters code at the console, the input first gets

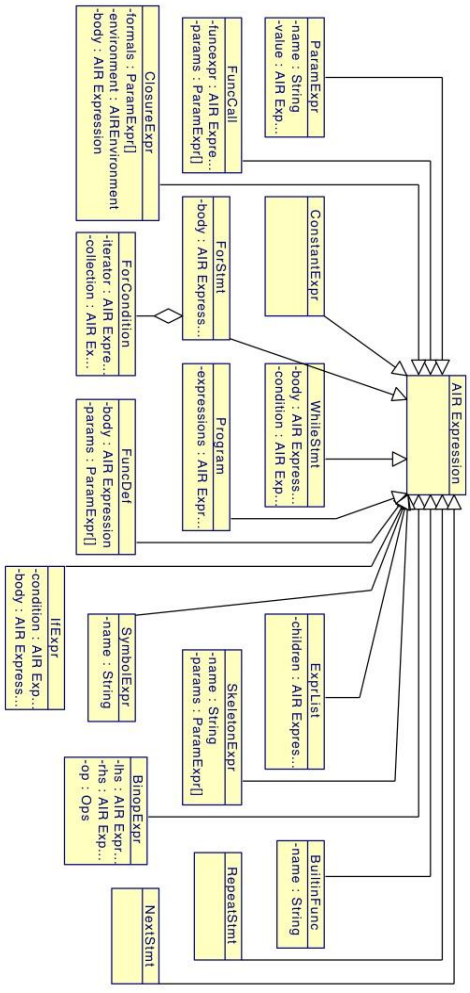

Figure 3. AIR expression types

parsed and translated into R's SEXPR intermediate format; then it is evaluated stepwise and the results are displayed.

Hence, any parallelization of R code must occur "on the fly" during interpretation. To this end, ALCHEMY intercepts the normal sequence of operations in the innermost R interpreter loop ("R_ReplIteration"). ALCHEMY breaks up the REPL at the point where the user input has been parsed into SEXPR code, and inserts its own code ("AlchemyProxy…") into the REPL, see step 4 in the simplified call graph in Fig. 5.

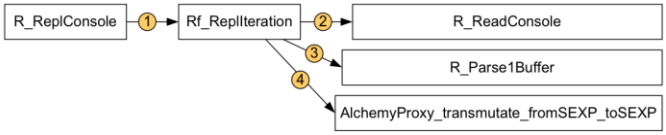

Figure 5. Intercepted R interpreter loop

The proxy code hands over the SEXPRs to the `AlchemyAdapter`, which converts them into AIR by calling the `RtoAIRConverter` service. Then, the AIR code is sent to the `AlchemyCore` component for parallelization and (possibly partial) parallel execution.

When the AIR code has been parallelized and executed, the computation results are wrapped into AIR code again by the last transmutator in the current ALCHEMY configuration. The results (including any code fragments that ALCHEMY was unable to parallelize) are returned to the `AlchemyAdapter`, which calls the `AIRtoRConverter` service to translate the AIR results back into SEXPRs. Finally, control is returned to the R interpreter loop, which resembles normal evaluation on the R code (SEXPRs) returned from ALCHEMY.

In our design, we have kept the changes to the R code base minimal. In particular, we extended the code of the interpreter loop in one location only. This minimal set of changes makes it as easy as possible to keep in sync with new versions of R in the future.

IV. A Sample Transformation Chain

To illustrate the inner workings of the ALCHEMY platform, we show how the simple, single-line R input at the bottom of the listing in Fig. 6 (a call to the sine function with a list of three numbers as operands) gets processed by the platform.

For this example, we assume that the "EMBA" analysis module and the "RMulticoreBackend" module have been configured into ALCHEMY. We'll describe these modules in more detail in the course of this section. For the moment being, it suffices to know that EMBA can detect data-parallel operations in R code, such as a math function being applied to a vector of data; and the RMulticoreBackend module is a simple backend that distributes work chunks among the cores of a multicore processor.

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> alchemy.loglevel(3)
setting loglevel to 3
> alchemy.enable()
ALC: Trying to initialize Alchemy session with server tcp://localhost:1984
ALC: Initialization succeeded (session ID = 1227083285)
ALC: Registering service 'ValSvc.getVal'
ALC: Registering service 'EnvSvc.lookup'
ALC: Registering service 'AIRtoR.convert'
> sin(c(1,2,3))
```

Figure 6. Start of ALCHEMY, and simple data-parallel R command

After having started the R environment, and prior to entering the sine statement, the user activates ALCHEMY ("alchemy.enable" in the screenshot of Fig. 6). This has to be done only once for each interactive session. ALCHEMY connects to its internal services, reads its configuration file, pushes the corresponding transmutators (here, EMBA and RMulticoreBackend) onto its internal processing queue, and returns control to the command window.

```
1  <AIR environment-proxy="tcp://127.0.0.1:1985"
2           value-proxy="tcp://127.0.0.1:1985" environment-id="
              169965928">
3      <Program>
4          <FuncCall>
5              <funcexpr>
6                  <SymbolExpr name="sin"/>
7              </funcexpr>
8              <params>
9                  <ParamExpr>
10                     <FuncCall>
11                         <funcexpr>
12                             <SymbolExpr name="c"/>
13                         </funcexpr>
14                         <params>
15                             <ParamExpr>
16                                 <ConstantExpr type="real">
17                                     <RealValue data="1.0"/>
18                                 </ConstantExpr>
19                             </ParamExpr>
20                             <ParamExpr>
21                                 <ConstantExpr type="real">
22                                     <RealValue data="2.0"/>
23                                 </ConstantExpr>
24                             </ParamExpr>
25                             <ParamExpr>
26                                 <ConstantExpr type="real">
27                                     <RealValue data="3.0"/>
28                                 </ConstantExpr>
29                             </ParamExpr>
30                         </params>
31                     </FuncCall>
32                 </ParamExpr>
33             </params>
34         </FuncCall>
35     </Program>
36 </AIR>
```

Figure 7. AIR code corresponding to the sine command

The user then enters the sine command. The input line gets parsed by the R interpreter and reaches the inner interpreter loop. At this point, ALCHEMY intercepts the R code (more precisely, the corresponding SEXPR code produced by the R parser, see section III.C) and transforms it into AIR code. This AIR code is listed in Fig. 7, using XML notation for convenience. The sine function appears in lines 5 to 7, the c-operator of R (which combines its arguments into a list) appears in lines 11 to 13, and the three operand numbers appear

in lines 15 to 29. All this happens in the `RAlchemy` component of the system.

The AIR code then gets transferred to the `AlchemyCore` component, where the transmutation controller schedules the EMBA analysis module. EMBA detects the data-parallel sine operation and transforms it into a MAP skeleton. The resulting AIR code is listed in Fig. 8, again using XML notation. The MAP appears in line 4, its vector argument in lines 6 to 10, and the function to be applied to the vector elements (the sine function) in lines 11 to 13.

```
1  <AIR environment-proxy="tcp://127.0.0.1:1985"
2           value-proxy="tcp://127.0.0.1:1985" environment-id="
              169965928">
3      <Program>
4          <SkeletonExpr name="MAP">
5              <params>
6                  <param name="collection">
7                      <AIRVector basetype="real">
8                          <Data data="1.0,2.0,3.0" length="3" />
9                      </AIRVector>
10                 </param>
11                 <param name="kernel">
12                     <SymbolExpr name="sin"/>
13                 </param>
14             </params>
15         </SkeletonExpr>
16     </Program>
17 </AIR>
```

Figure 8. AIR code after its transformation to MAP skeleton code by the EMBA analysis module

At this point, the code has been completely analyzed and basically is ready for execution on a multicore target. All that is needed is some more code generation for, say, Intel's TBB, or some other common multicore library.

In this example, we instead use our special backend module "RMulticoreBackend" that transforms MAP skeletons back into *R code* based on the "R multicore library" package [9]. We found this special backend module very useful for testing the operation of our platform quickly and with low overhead. The "R multicore library" implements a simple fork-join-algorithm to distribute R code and data among processes on a multicore machine under Linux.

Fig. 9 lists the AIR code (XML notation) after the RMulticoreBackend has transformed the MAP skeleton. The output now includes a call to the R multicore library-function "pvec" in lines 15 to 17. By its definition, "pvec" is semantically equivalent to a MAP. In addition, the backend module automatically inserts a call that loads the R multicore library, see lines 4 to 13. Other than that, our backend module relies on the semantic correctness of the "R multicore" package.

```
2  <AIR>
3      <Program>
4          <FuncCall>
5              <funcexpr>
6                  <SymbolExpr name="library"/>
7              </funcexpr>
8              <params>
9                  <ParamExpr>
10                     <SymbolExpr name="multicore"/>
11                 </ParamExpr>
12             </params>
13         </FuncCall>
14         <FuncCall>
15             <funcexpr>
16                 <SymbolExpr name="pvec"/>
17             </funcexpr>
18             <params>
19                 <ParamExpr>
20                     <FuncCall>
21                         <funcexpr>
22                             <SymbolExpr name="c"/>
23                         </funcexpr>
24                         <params>
25                             <ParamExpr>
26                                 <ConstantExpr type="real">
27                                     <RealValue data="1.0"/>
28                                 </ConstantExpr>
29                             </ParamExpr>
30                             <ParamExpr>
31                                 <ConstantExpr type="real">
32                                     <RealValue data="2.0"/>
33                                 </ConstantExpr>
34                             </ParamExpr>
35                             <ParamExpr>
36                                 <ConstantExpr type="real">
37                                     <RealValue data="3.0"/>
38                                 </ConstantExpr>
39                             </ParamExpr>
40                         </params>
41                     </FuncCall>
42                 </ParamExpr>
43                 <ParamExpr name="FUN">
44                     <SymbolExpr name="sin"/>
45                 </ParamExpr>
46             </params>
47         </FuncCall>
48     </Program>
49 </AIR>
```

Figure 9. AIR code after its transformation to R multicore library-calls

The backend module is the last module in the queue for this sample configuration of ALCHEMY. Hence, the AIR code produced by the last module is transferred back to the `RAlchemy` component, where it is converted to R code (more precisely, to SEXPR code) and fed back into the inner R interpreter. The interpreter resumes work and evaluates the R code. This leads to the execution of the "pvec"-call in the R multicore library. In effect, the whole procedure triggers the same evaluation inside the R interpreter and produces the same result as if the user would have entered the commands listed in Fig. 10 at the R console prompt manually.

```
1 {
2     library(multicore)
3     pvec(c(1, 2, 3), FUN = sin)
4 }
```

Figure 10. R code returned to the inner interpreter loop after the parallelization analysis

The screenshot in Fig. 11 shows the final output at the console. One can see that the whole analysis, transformation, and execution process has occurred *fully transparent* to the user. All the user gets to see is the result of the computation triggered by his/her sine command. The only noticeable difference is that for large computations the prompt returns more quickly due to the higher execution speed.

```
> sin(c(1,2,3))
[1] 0.8414710 0.9092974 0.1411200
>
```

Figure11. Final output at the R console window

The UML collaboration diagram in Fig. 12 shows the transmutators that have been configured into ALCHEMY for this example, the workflow between the modules, and the different versions of AIR code that occur in the workflow, as described above. The transmutation controller in `AlchemyCore` pushes all three transmutator modules onto its processing queue at startup, beginning with the "FuncDefFilter" module.

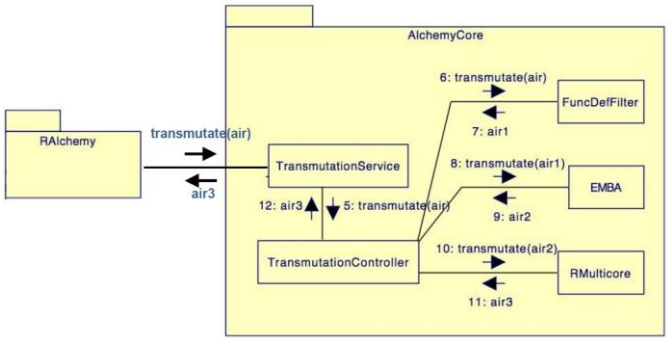

Figure 12. Transmutation workflow for the example

The FuncDefFilter is a special transmutator that checks whether the piece of code under analysis consists of *only* a function definition. In such a case, the AIR code is left unchanged, parallelization analysis is aborted, and the AIR code is returned to the `RAlchemy` component, where it is converted back to SEXPRs and returned to the R interpreter for further evaluation. The reason is that function definitions are analyzed "just in time" when the function is actually *called* in the code. At that point in time the function definition is fetched from the R environment (where such definitions are stored) using the `EnvironmentSvc` and analyzed.

Such a "just in time" lookup occurs in the example given in the following section V, where the EMBA module first fetches the definition of the function "myfun," and later the definition of the "helper" function from the `EnvironmentSvc` for further analysis.

The EMBA parallelization analysis module detects "embarrassingly parallel" operations in R code. As we have seen in the example, this includes a number of built-in R functions operating on vector data, such as the sine function.

In addition, EMBA detects calls to the special R function "lapply" which applies a given function to each element of a vector. The "lapply"-function is quite common in R programs because it can be used as a substitute for a loop, making the program run faster. Loops typically are slow when using an interpreter, but "lapply" has a fairly fast implementation in the R base library.

EMBA transforms any such data-parallel operation into a MAP skeleton. MAP is a simple skeleton that applies a given function to a vector of data element-wise; hence, this program transformation obviously preserves the semantics of the input program. EMBA does not replace subtrees in AIR code that have already been processed by EMBA to avoid inefficient processing of nested parallelism. The flow chart in Fig. 13 shows the steps taken by the current version of EMBA.

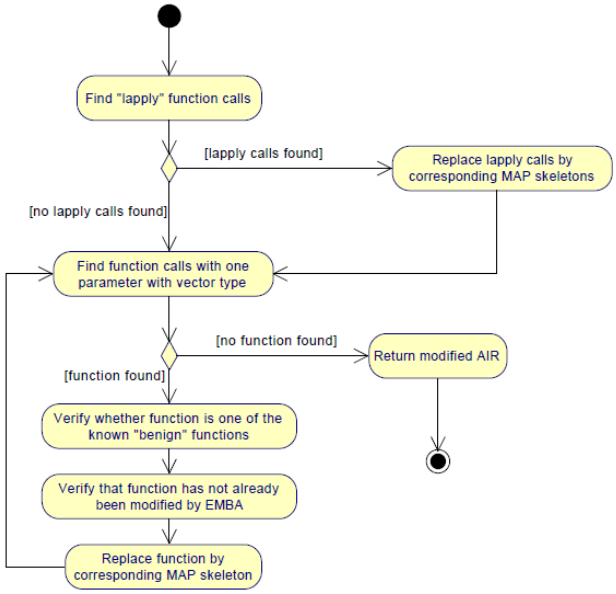

Figure 13. Workflow of the EMBA analysis module

V. Some Early Measurements

As discussed in the previous section, our EMBA analysis module automatically detects data-parallel operations in an R program, such as a sine function that is applied to a vector of data. In R, many functions operate on vectors or matrices of data; hence, there naturally is some potential for automatic parallelization in R code at a fine-grained level.

To illustrate how easily this potential might be exploited using an automatic parallelization approach such as our ALCHEMY platform, Fig. 14 shows a listing of some R code that applies a summation loop involving sine and cosine computations (function "helper") to a data vector of configurable size ("numelements"). The "lapply" statement is standard R and applies its second argument (a function) to each element of its first argument (the data).

```
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
10     lapply (vec, helper)
11 }
12
13 alchemy.loglevel(6)
14 alchemy.enable()
15
16 system.time(myfun (1:<numelements>))
```

Figure 14. Sample data-parallel program

As a sequential program executed on a single core, this computation can require up to 5 minutes of execution time for a data vector containing one million elements. Our EMBA module automatically detects the lapply-operation and transforms the R code into a parallel AIR version using the MAP skeleton. The AIR code then is fed into the special RMulticoreBackend, which transforms the MAP into a pvec-operation of the R multicore library; see section IV for details.

Fig. 15 illustrates the speedup that can be gained with EMBA for various sizes of the data vector. On a PC with 8 cores, the max speedup is about a factor of 5 as compared to the sequential program. The measurements were carried out under Ubuntu on an AMD FX-8120 with 8 cores running at 3.1 GHz, with 8 GB of main memory.

This speedup is not spectacular, but one should bear in mind that the backend used in this example is fairly simple. We expect to see better results with more advanced multicore libraries; a backend module for Intel's ArBB library is already under development.

For simply-structured massively data-parallel programs such as in our example, one can expect the speedup to scale fairly well with the number of processing units. For application programs that contain significant data-dependencies, the speedup typically is limited, though. In particular, the max speedup might well be reached with the six or eight cores currently available in off-the-shelf multicore PCs. In general, the speedup is hard to estimate and depends on the application, the input data size, and the particular parallelization technique applied.

With the ALCHEMY platform, any speedup comes *completely for free* for the R user – he doesn't have to write a single line of parallel code to benefit from the multicore performance.

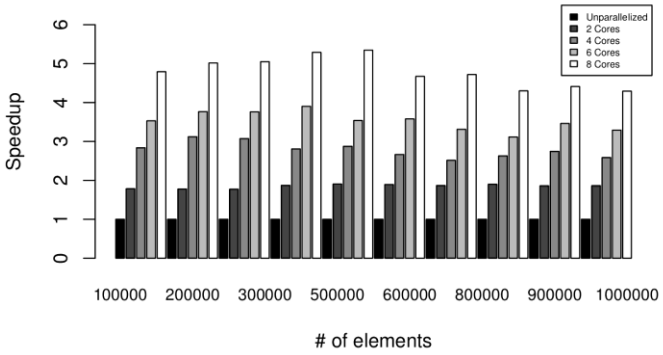

Figure 15. Speedup for the parallelized version of the sample program with varying input and number of cores

VI. ONGOING WORK

We are currently implementing several more sophisticated analysis and transformation modules (transmutators) in our experimentation platform. As the platform currently is still at an early stage, we discuss this ongoing work in more detail here to indicate in which direction our research is headed.

A few years ago, a group of researchers from the University of Tokyo have developed an interesting approach for parallelizing dynamic programming structures [20]. Their approach is based on list skeletons, such as ZIPW, SCAN, SHIFT, and MAP. They perform a dependency analysis of the statements in the nested loops of the dynamic program, then duplicate the underlying data matrix and handle each copy separately by transforming the loops and the statements in the loop body into suitable list skeletons. Fig. 16 shows a sample transformation, using pseudo-code notation. Since dynamic programming techniques occur quite frequently in applications, in particular in string matching for bioinformatics, we are currently implementing this approach. The detection part of our analysis module MATSU employs tree grammars, the code transformations are implemented at the syntax tree level. We expect first experimental results to be available soon.

```
for i in 2:n
for j in 2:m
  D[i][j] <- min(D[i][j-2]+1, D[i][j-1]+2, D[i-1][j]+1)

maps to (pseudo-code notation)

define op(x,y) :=  min(x+1,y)
E[_,0] = D[_,0]

for j in 1:m-1
  F[_,j] = zipw(min,
                map(f(x)=x+2, shift(1, E[_,j-1],
                map(f(x)=x+1, shift(0, E[_,j-1])
  E[_,j] = scan(op, F[_,j])
```

Figure 16. Sample transformation of dynamic programming into list skeletons

At the same time, we are developing an analysis module that implements automated dependency analyses in nested loops over matrices, based on the well-known SURE technique ("systems of uniform recurrence equations") [22]. Basically, SURE computes a compact representation of the data dependencies from the index accesses that occur in the loop body, then detects cyclic dependencies using linear programming, and finally computes a multi-dimensional schedule for the statements in the loop body according to the cycles found in the previous step. From this schedule, a parallel version of the nested loops can be computed that is naturally expressed using DOPAR skeletons. Fig. 17 shows a sample transformation, using pseudo-code notation. We expect the SURE analysis module to be useful for a range of R application programs.

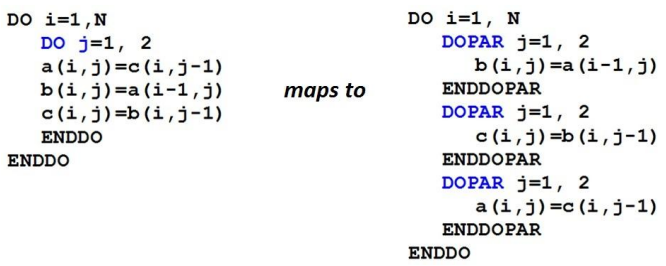

Figure 17. Sample transformation of nested loops into parallel loops

To provide state-of-the-art performance at the end of the parallelization pipeline, we are currently writing a backend that transforms AIR code into C++ code for Intel's ArBB platform ("array building blocks") [24]. Our backend will support parallel skeletons such as MAP, SCAN, and ZIPW for general function arguments, and the basic mathematical functions of R. Some of these skeletons are already supported by ArBB, but only for specific function arguments such as addition and basic operand types such as integers. We are implementing more general versions, based on known techniques for a parallel execution of the skeletons.

Recall that any AIR code that is not processed completely by a backend gets automatically fed back into the R interpreter for final evaluation. Hence, no code gets lost even if the program contains some statements that are not yet supported by the backend. For future versions of our ArBB backend, we also plan to add logic that *dynamically* finds an optimal distribution of the code and data code among the cores, resp., computing units. To this end, we shall apply the auto-tuning techniques that have previously been developed in our group [25].

The MATSU and SURE analysis modules are based on static code analysis. We are also working on dynamic analysis techniques that perform runtime measurements in the code to detect promising locations for parallelization. For example, we are experimenting with a prototype tool that identifies groups of functions that seem to behave according to a master-worker design pattern. The analysis tool measures and compares the call frequencies and computing times of the function calls. At a more local code level, we experiment with identifying program statements that can be computed asynchronously to surrounding statements, such as function calls whose results are not used immediately, or function calls that implement expensive computations in different branches of switch blocks. Such calls can be automatically transformed into "futures" [26] and distributed among different threads. All this is ongoing work.

VII. Conclusions

We have presented a platform for the automatic parallelization of programs written in the R language. R is an interpreted interactive language; hence, parallelization must take place "on the fly" during evaluation of the code in the interpreter. Our platform intercepts the R code after it has been parsed by the inner R interpreter loop, runs a (multi-stage) parallelization analysis and corresponding code transformation, executes the parallelized parts of the program on a multicore processor, and then feeds back the (partial) computation results with the remaining R program into the interpreter loop. This way, parallelization is fully transparent to the user.

At this early stage, our platform aims at encouraging researchers to experiment with different approaches and techniques for the automatic parallelization of R. Any parallelization technique can be implemented as a separate analysis module and linked into the platform. The analysis and code transformation modules can be freely linked together in the platform using a configuration file, allowing for automatic parallelization in stages. In particular, our platform makes it easy to experiment with combinations of different parallelization techniques. This flexibility is made possible by our design decision that all modules in the platform use a common intermediate code representation as both input and output. We described this parallel intermediate language in detail.

We described the internal workflow of our platform using an example, showing all intermediate code versions that occur during the analysis. Parallelization is performed by a simple module that detects "embarrassingly parallel" operations in R code. This includes many built-in math functions operating on vector data, but also R's special "lapply"-statement that applies an arbitrary function argument to a vector of data. Such data-parallel operations occur frequently in real R programs; hence, being able to parallelize such code automatically is highly relevant in practice. The parallelized R code was executed using a special, standard R library on a multicore processor. We presented some early measurements that indicate the speedup possible with this simple parallelization module. Clearly, more experiments and measurements need to be performed in the future to evaluate the ALCHEMY approach and the performance of various parallelization techniques.

We also sketched several more sophisticated parallelization modules that we are currently implementing, based on known parallelization techniques. This includes a module that parallelizes dynamic programming code using data dependency analysis and list skeletons; and a module that parallelizes nested loops over matrices using mathematical recurrence equations and scheduling techniques. In addition, we are currently implementing a more powerful backend module for multicore processors using Intel's ArBB library. We expect to present first experimental results soon.

The goal remains the same for all ALCHEMY modules that are under development: R users shall benefit from multicore performance without having to write a single line of parallel code.

ACKNOWLEDGMENT

We would like to thank Prof. Sebastian Hack from the computer science department at Saarland University, Germany, for fruitful discussions about the subject and valuable hints to the literature.

REFERENCES

[1] R. C. Gentleman, V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Maechler, A. J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. Yang, J. Zhang, "Bioconductor: Open software development for computational biology and bioinformatics," J. Genome Biology, vol. 5, issue 10, 2004

[2] R. Ihaka, R. Gentleman, "R: A language for data analysis and graphics," J. of Computational and Graphical Statistics, vol. 5, 1996, pp. 299-314

[3] W. N. Venables, D. M. Smith: *An introduction to R*. Network Theory Ltd., 2002

[4] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, U. Mansmann, "State of the Art in Parallel Programming with R," J. of Statistical Software, vol. 31, issue 1, August 2009, pp. 1-27

[5] H. Yu, "Rmpi: Parallel Statistical Computing in R," R News, vol. 2, issue 2, 2002, pp. 10-14, online at http://CRAN.R-project.org/doc/Rnews/

[6] N. M. Li, A. J. Rossini, "rpvm: Cluster Statistical Computing in R," R News, vol. 1, issue 3, 2001, pp. 4-7, online at http://CRAN.R-project.org/doc/Rnews/

[7] A. J. Rossini, L. Tierney, N. M. Li, "Simple Parallel Statistical Computing in R," J. of Computational and Graphical Statistics, vol. 16, issue 2, 2007, pp. 399-420

[8] G. Vera, R. C. Jansen, R. L. Suppi, "R/parallel. Speeding up Bioinformatics Analysis with R," BMC Bioinformatics, vol. 9, 2008, pp. 390-396

[9] S. Urbanek, "Package multicore," online at http://cran.r-project.org/web/packages/multicore/multicore.pdf

[10] X. Ma, J. Li, N. F. Samatova, "Automatic Parallelization of Scripting Languages," 21st Int. Parallel and Distributed Processing Symposium IPDPS, 2007, pp. 1-6

[11] U. Banerjee: *Loop Parallelization. Loop Transformations for Restructuring Compilers*. Kluwer, 1994

[12] C. A. Lattner: LLVM. *An Infrastructure for Multi-Stage Optimization*. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002

[13] M. Cole, "Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming," J. Parallel Computing, vol. 30, issue 3, 2004, pp. 389-406

[14] T. G. Mattson, B. A. Sanders, B. L. Massingill: *Patterns for Parallel Programing*. Addison-Wesley, 2005

[15] M. McCool, J. Reinders, A. Robison: *Structured Parallel Programming*. Morgan Kauffman, 2012

[16] G. E. Blelloch, S. Chatterjee, "VCODE: A Data-Parallel Intermediate Language," 3rd Symposium on the Frontiers of Massively Parallel Computation, 1990, pp. 471-480

[17] S. Marr, M. Haupt, T. D'Hondt, "Intermediate language design of high-level language virtual machines: Towards comprehensive concurrency support," 3rd Workshop on Virtual Machines and Intermediate Languages VMIL October 2009, extended abstract

[18] ZeroMQ Project, online at http://www.zeromq.org

[19] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, M. Zagha, "Implementation of a Portable Nested Data-Parallel Language," J. of Parallel and Distributed Computing JPDC, vol. 21, issue 1, April 1994, pp. 4-14

[20] K. Kakehi, K. Matsuzaki, A. Morihata, K. Emoto, Z. Hu, "Parallel Dynamic Programming using Data-Parallel Skeletons," 22nd Conference of the Japan Society for Software Science and Technology, 2005

[21] E. Anwar Ghuloum, J. Sprangle, G. Fang, X. Z. Wu, "Ct: A Flexible Parallel Programming Model for Tera-scale Architectures," Intel White Paper, October 2007, online at http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf

[22] A. Darte, Y. Robert, F. Vivien: *Scheduling and Automatic Parallelization*. Birkhäuser Verlag, 2000, chapter 4.2

[23] R Development Core Team: *R Internals*. online at http://cran.**r**-project.org/doc/manuals/R-ints.pdf

[24] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Du Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, D. Zhang, "Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language," 9th Int. Symposium on Code Generation and Optimization CGO, 2011, pp. 224-235

[25] C. A. Schaefer, V. Pankratius, W. F. Tichy, "Engineering parallel applications with tunable architectures," 32nd Int. Conference on Software Engineering ICSE, 2010 pp. 405-414

[26] K. Molitorisz, J. Schimmel, F. Otto, "Automatic Parallelization using AutoFutures," Int. Conference on Multicore Software Engineering MSEPT, May 2012