pR: Automatic, Transparent Runtime Parallelization of the R Scripting Language

Jiangtian Li *† Xiaosong Ma *† Srikanth Yoginath† Guruprasad Kora†
Nagiza F. Samatova†

Abstract

Scripting languages such as R and Matlab are widely used by scientists for data processing. As the amount of data and the complexity of analysis tasks both grow, sequential data processing using these tools often becomes the bottleneck in scientific workflows. We describe pR, a runtime framework for automatic and transparent parallelization of the popular R language used in statistical computing.

Recognizing R's interpreted nature and computation-intensive R codes' use pattern, pR adopts several novel techniques: (1) runtime whole-program dependence analysis and code transformation assisted with evaluation results, (2) a selective parallelizing scheme that only parallelizes the expensive parts of the program, namely loops and function calls, and (3) a master-worker scheduling and execution engine that only "outsources" expensive tasks to the workers. Our framework uses MPI for inter-processor communication and does not require any modification to either the source code or the underlying R implementation. Experimental results demonstrate that pR can exploit both task and data parallelism in a totally transparent manner and overall has better performance as well as scalability compared to an existing parallel R package that requires code modification.

1 Introduction

Ultra-scale simulations and high-throughput experiments have become increasingly data-intensive, routinely producing terabytes or even petabytes of data. Hidden in this "hay of stack" is "a needle" – a scientific discovery. Finding the needle is a tedious and time-consuming task that requires advanced high-performance data analytics.

Scientific data analysis is a complex but inherently parallel process. It is often highly repetitive: performing the same set of statistical functions iteratively over many data objects, which are generated from different timesteps, data partitions, etc. In most cases, both *task parallelism* and *data parallelism* are present. For example, computing the eigenvalue decomposition and generating a histogram of a matrix, two common R tasks, can be perfectly overlapped with each other and executed in a task parallel fashion. Likewise, a loop processing the elements of a large array may be broken into blocks and executed in a data parallel manner.

While inherently parallel, scientific data analytics lacks the proper statistical tools and libraries that could efficiently exploit this parallelism. Many popular tools such as R [21], S-plus [1], and IDL [2] have not been fully parallelized. As a result, the data analysis capacity has increasingly become a rate limiting factor on the path to scientific discovery. To overcome this bottleneck, scientists are spending a lot of their time on writing customized parallel data analysis utilities rather than focusing on their science. In this paper, we address this problem with an automatic and transparent runtime parallelization framework for R, a popular scripting language for statistical computing (more background information about R will be given in Section 2.1).

Although exploiting task and data parallelism in the data analysis scenarios does not seem difficult, the key challenge in doing so lies in the fact that such parallelization should be done in a transparent manner as much as possible. Ideally, the domain scientists should be able to re-use in a parallel environment their existing sequential data analysis codes with little or no changes. Throughout their

*Department of Computer Science, North Carolina State University `xma@ncsu.edu`

†Computer Science and Mathematics Division, Oak Ridge National Laboratory, `samatovan@ornl.gov`

scientific career, many scientists have accumulated hundreds or even thousands of lines of their favorite codes. Re-writing or making changes to such collectively lengthy codes is an error-prone, tedious, and psychologically painful exercise that scientists will often resist to. Also, domain scientists, especially experimentalists, often lack the knowledge and experience of parallel computing. Hence, an approach that requires these scientists' explicit parallel programming will likely fail.

The requirement of transparency greatly complicates the parallelizing task. In a user-driven explicit parallelization set-up, it is the user's job to directly specify what sessions to run in parallel, how to distribute the data across processors, what interprocess communication scheme to use, when to synchronize the processors, and which parts of the code to run in task- or data-independent fashion. Performing all these tasks automatically, without direct user supervision, is difficult. Doing it efficiently is even more challenging. The nature of the underlying programming language brings another dimension to this problem.

The fact that R is a scripting language is both a curse and a blessing. On one hand, its scripting feature facilitates *runtime parallelization*, as a lot of information not known *a priori* becomes available at execution time while an R script is being interpreted. On the other hand, since the dependence analysis and scheduling of such tasks are done online, guaranteeing overall performance efficiency gets complicated.

In this paper, we present our work towards supporting efficient, yet transparent R parallelization. We have designed and implemented pR, a modified R environment that executes R scripts in parallel. We consider the major contributions, as well as key novelties of this paper to be as follows:

- We developed several automatic runtime techniques to parallelize R, which do not require any source code modification and hide all the parallelization details from users. To our best knowledge, this is the first transparent parallelizing tool for a high-level language without using special hardware support.
- We applied parallelizing compiler's technology to parallelizing an interpreted language and coupled it with incremental runtime analysis.
- We evaluated the pR prototype with both real-world data analysis applications and synthetic codes. Our experiments demonstrate that a significant speedup can be achieved and the parallelizing overhead is relatively small.

The rest of the paper is organized as follows. Section 2 gives background information on R and discusses related work. Section 3 presents an overview of pR's design rationale and system architecture, while Section 4 discusses design and implementation details. Section 5 evaluates both the user interface and the performance. Section 6 talks about the system limitations and future work, and Section 7 concludes the paper.

2 Background

2.1 About R

R [21] is an open-source software and language for statistical computing and graphics, which is widely used by the statistics, bioinformatics, engineering, and finance communities. It has a center part that was developed by its core development team and provides add-on hooks for external developers to write and add extension packages. The R source codes were written mainly in C. R can be used on various platforms such as Linux, Macintosh and Windows and can be downloaded from the CRAN (Comprehensive R Archive Network) site at http://www.r-project.org/.

As a data processing tool, R can perform diverse statistical analysis tasks such as linear regression, classical statistical tests, time-series analysis, and clustering. It also provides a variety of graphical functions such as histograms, pie charts and 3D surface plots. Built upon R, many other data processing software tools are developed, such as BioConductor, an open-source and open-development software package for the analysis of biological data [6].

R is an interpreted language, whose basic data structure and entity is an *object*. Internally an R object is implemented as a C struct `SEXPREC`, whose naming roughly corresponds to a Lisp "S-expression." For example, an object may be a vector of numeric values or a vector of character strings. R also provides a *list*, an object consisting of a collection of objects.

R can be used in both interactive and batch modes. In the interactive mode, R issues a prompt to expect input commands. Upon receiving one correct command, the R environment executes the command. The user retrieves the results from the standard output by typing variable names. In the batch mode, an R script is supplied as a file and executed from the R prompt. Results can be written into output files or retrieved from the standard output as in the interactive mode. This paper focuses on the parallelization of the batch-mode execution.

```
a <- 1                                                    1
b <- 2                                                    2
c <- rnorm(9)                                             3
d <- array(0:0, dim=c(9,9))                               4
for (i in b:length(c))                                    5
{                                                         6
   c[i] <- c[i-1] + a                                     7
}                                                         8
for (i in 1:length(c))                                    9
{                                                        10
   d[i,] <- matrix(scan(paste(“test.data”, i, sep=“”)))  11
}                                                        12
if (c[length(c)] > 10)                                   13
{                                                        14
   e <- eigen(d)                                         15
}                                                        16
else                                                     17
{                                                        18
   e <- sum(c)                                           19
}                                                        20
```

Figure 1. A sample R script.

In Figure 1, we present a running example of an R script that will be used to illustrate how pR works. It is designed more to cover important pR issues than to perform a meaningful job. Line 1 and line 2 are simple constants assignment. Line 3 generates a vector of nine normally distributed real numbers. Line 4 initializes a 2-D array. Lines 5-8 are a loop performing a simple arithmetic operation. Lines 9-12 read the matrix data from the file. Lines 13-20 are a conditional branch. The first branch computes eigenvalues and eigenvectors of the rectangular matrix and the second branch finds the summation of all the vector elements.

2.2 Related Work

Several projects aiming at parallelizing programs in widely used interpreted languages have recently emerged in response to an increasing demand for their parallel processing capabilities. Choy and Edelman [10] have surveyed 27 parallel Matlab systems and categorized them as: (1) embarrassingly parallel, (2) message passing, (3) back-end support, (4) Matlab compilers, and (5) shared memory. Among them, the embarrassingly parallel, message passing and shared memory approaches require explicit parallel programming. Programmers either need to mark embarrassingly parallel task regions or to orchestrate communication as well as synchronization. Back-end support has been a popular approach utilizing high-performance numerical libraries such as ScaLAPACK [9] to carry out Matlab routines in parallel on multiple servers. Star-P [11], a typical example from this category, provides transparent parallelization through function overloading and supports user-controlled row-wise, column-wise or block-wise distribution of data matrices. Matlab compiler-based approaches require special compiler support to translate Matlab programs into codes in a compiled language or directly into parallel executables. While efficient, these approaches are less portable and fail to utilize any runtime information.

Similar categorization applies to existing work on other scripting languages. For example, pyMPI [20] provides MPI interfaces for parallel programming in Python and belongs to message passing category. For R, the snow package [22] exploits embarrassingly parallel tasks and belongs to category 1, while our RScaLAPACK [24, 29] uses the ScaLAPACK library along with the dynamic process management for parallel computation and could be broadly considered to fall under category 3.

In contrast, our proposed framework is highly complementary to all the aforementioned categories. It parallelizes sequential R code without requiring any source code modification or specialized compilers. Similar to our preliminary task-pR package [27], it performs runtime dependence analysis and scheduling of task parallel jobs within the user-specified blocks of code. However, it brings task-pR's capabilities to a completely new level: (a) it automatically detects data parallel blocks inside of *for* loops, unrolls those loops and runs them concurrently, (b) it performs dependence analysis on the entire program thus eliminating the need for code changes previously required for marking the blocks of interest, and (c) it increases the efficiency through selectively "outsourcing" only expensive tasks to worker processors, while letting the master execute light-weight tasks. Our current framework assumes that individual functions are sequential codes and it does not explicitly parallelize them. However, our approach can be combined with back-end support methods through adjusting its scheduling strategy (this work is in-progress and not reported here).

The snow package [22] mentioned above is probably the closest related project to our framework, in the sense that both tools allow users to parallelize independent operations. snow works with interactive execution while pR currently only supports batch-mode runs. However, pR's parallelization is much more general (for example, it can parallelize two heterogeneous function calls), and unlike snow, pR does not require any modification to the sequential R source code. Performance wise, our experiments indicate that pR is able to achieve similar speedup with snow on an embarrassingly parallel code, and an order-of-magnitude better speedup on a more communication-intensive code. Section 5 give more related details.

Regarding transparent or semi-transparent parallelization, there have also been a few related projects. A simple yet powerful interface called MapReduce was designed for processing and generating large data sets in Google [13]. With MapReduce, the run-time system handles data partitioning and task scheduling automatically. However, it is targeted toward a certain type of data parallelism (where the same operation is performed over a large set of objects, with a reduction operation at the end to merge the results). Users still need to explicitly specify the Map and Reduce operations. pR, again, exploits both data parallelism and task parallelism, with no extra effort requested from R programmers. To a certain extent, pR resembles OpenMP [3], a set of compiler directives and callable runtime library routines that enables transparent shared-memory parallelism. However, with OpenMP, programmers are responsible for checking dependencies, deadlocks, race conditions, etc., while pR handles parallelization transparently. In addition, Seinstra et al. designed a software architecture for parallel image processing [26]. Users are supplied with a parallel library with interfaces identical to those from a sequential image library. In this case, although the parallelization is hidden from the user, different library calls cannot be parallelized even when they are independent from each other.

In a nutshell, our work performs what parallelizing compilers do to compiled languages on an interpreted language. Two representative parallelizing compilers, SUIF [17] and Polaris [7], transform sequential C or Fortran codes into parallel codes (e.g., using OpenMP). The design of pR has borrowed mature techniques from these projects, such as loop dependence analysis. However, dealing with online parallelization of an interpreted language, we have to address new issues such as runtime task scheduling. In addition, the scope of parallelization of the aforementioned compilers may be significantly limited by the lack of information at compile time, while pR is assisted with runtime dependence analysis.

Several runtime parallelization techniques have been proposed. Salz et al. presented run-time methods to automatically parallelize and schedule iterations of a *do* loop in a situation where compile-time information is inadequate [23]. Gupta et al. presented a set of new run-time tests for speculative parallelization of loops [16]. In contrast, our work moves the static analysis performed by compilers to run time and combines such analysis with run-time parallelization. Chen et al. proposed the Java runtime parallelizing machine (Jrpm), a complete system for parallelizing loops in sequential Java programs automatically [8]. Like other systems using speculative multi-threading (e.g., [14]), Jrpm relies on hardware support to ensure correctness and requires additional profiling. Our framework performs a similar task in the R context but works at the user level. Both systems allow the sequential code to be parallelized without modifications to the original sequential codes, but Jrpm deals with *while* loops more easily, while pR supports task parallelism beyond loops.

In addition, the dynamic, incremental dependence analysis technique presented in this paper is related to but quite different from the existing dynamic compilation technology. Mechanisms such as just-in-time compilation [12] and incremental compilation [25] are designed to reduce the overhead of runtime bytecode interpretation or interactive compilation. Our work is closer to dynamic compiling systems such as DyC [15]. The difference is that these systems perform runtime optimization while our framework takes advantage of the interpreted nature of the R language to perform runtime parallelization.

3 pR Overview

In this section we describe the a high-level design of the pR framework, with more details given in Section 4.

3.1 Design Rationale

The approach we adopt in building pR is based on several motivating observations regarding computation-intensive and/or data-intensive R codes:

- As a high-level scripting language, the use pattern of R is significantly different from those of general-purpose compiled languages such as C/C++ or Fortran. Most R codes are composed of high-level pre-built functions [10] typically written in a compiled language but made available to R environment through dynamically loadable libraries. For example, the R function `svd()` utilizes LAPACK's Fortran `dgesdd()` code underneath rather than providing an explicit low-level matrix decomposition in a scripting language.

- While users would not frequently write their own nested loops to implement tasks such as matrix operations (as many such operations are already provided by R), loops are widely used to carry out similar tasks repeatedly, such as Markov Chain Monte Carlo, bootstrap sampling or likelihood maximization or going through a collection of data files. These "coarse-grained" operations, compared with "fine-grained" loops used in numerical functions, typically have less inter-iteration dependency and higher per-iteration execution cost, making them ideal candidates for data-level parallelization.

- Several characteristics of the R language simplify dependency analysis. First, there are no data pointers. The input parameters to the functions are read-only, while the modified or newly-created variables by the function are returned explicitly. Returning several such variables is accomplished through the formation of a list object, which is a collection of other objects. Second, R only uses pass-by-value. While this may cause un-necessary memory-copies (especially for read-only input parameters to a function), this removes the aliasing problem, a major limiting factor in general-purpose compilers. Still, users may call external functions that are written in other languages such as C, which may contain the use of pointers.

As a result, in designing this proof-of-concept parallel R framework, we focus on parallelizing two types of operations: function calls and loops. In a typical computation-intensive R program (and programs in other languages as well), these two form the bulk of the execution time.

Here we highlight the two major innovations in the pR design. The first one is *runtime analysis/parallelization*. We perform dynamic dependency analysis before interpreting R statements and identify tasks and loops that can be parallelized. This allows us to go beyond loop parallelization, which has been the primary focus of parallelizing compilers, to also exploit task parallelism between any two statements. In addition, we perform incremental analysis that delays the processing of conditional branches and dynamic loop bounds until the related variables are evaluated.

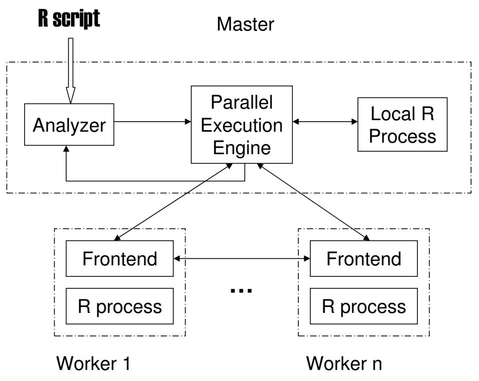

Figure 2. pR Architecture.

To parallelize an entire program at the granularity of individual statements, however, may generate too much scheduling and data communication overhead and hurt the overall performance. We address this with our second innovation - a *selective and asymmetric parallelization model*. Instead of generating a symmetric Single Process Multiple Data (SPMD) type of parallel code using one or more "fork-join" sessions, we adopt a master-worker paradigm that only "outsources" the expensive jobs (i.e., function calls and loops) to the workers. All the light-weight operations, such as simple statements and conditional statements that do not contain any loops or function calls, are executed locally by the master. This selective and asymmetric parallelization approach reduces the parallel execution overhead as well as the communication cost.

Details of our design will be given in Section 4.

3.2 Framework Architecture

The key feature of pR is that it dynamically and transparently analyzes a sequential R source script and accordingly parallelizes its execution. The results of partial execution are collected to perform further analysis at run time. The framework is built on top of and does not require any modifications to the native R environment. Internally, the Message Passing Interface (MPI) [19] library is used for inter-node communication.

When users run their R scripts in parallel using pR, one of the processors, the one with the MPI rank 0, is assigned as the *master node*, while the others become *worker nodes*. As shown in Figure 2, there is an R process running on the master and each of the worker nodes. This process executes individual R tasks: functions and parallelized loops on the workers and all the other tasks on the master.

The basic execution unit in pR is an *R task* (or *task* for brevity), which is the finest unit for scheduling. A task is essentially one or multiple R statements grouped together as a result of parsing, dependence analysis and loop transformation. A task can be a part of a parallelized loop, a standard function call, or a block of other statements between these two types of objects. As shown in Figure 2, there is an R process running on the master and each of the worker nodes. This process executes R tasks: functions and parallelized loops on the workers and all the other tasks on the master.

The major complexity of pR resides at the master side, which performs dynamic code analysis, on-the-fly parallelization, task scheduling, and worker coordination. These are carried out by two components: an *analyzer* and a *parallel execution engine.*

The analyzer forms the front-end of our pR system. Its primary functionality is to perform syntactic and semantic analysis of R scripts. Such analysis helps pR identify execution units and their precedence relationship to exploit task and data parallelism.

The parallel execution engine works as the back-end of pR and takes input from the analyzer. It is responsible for dispatching tasks, coordinating the communication among the workers, supervising the local R processing, and collecting results.

The analyzer pauses where static analysis is not sufficient to perform parallelization, such as conditional branches and loops with dynamic bounds. The analyzer resumes its analysis after the parallel execution engine provides appropriate runtime evaluation results. In this case, these results are fed-back to the analyzer, as shown in Figure 2.

Each of the worker nodes also has a front-end process, which interacts with the master and other worker nodes. This way, data communication can be performed without interrupting the R task execution carried out by the R process. The front-end process manages the data, tasks, and messages for the worker. It supplies the R process with task scripts and input data, and collects the output data from the latter.

The inter-node communication in pR is performed via MPI, while the inter-process communication on each node is performed via the UNIX domain sockets.

4 pR System Design and Implementation

In this section, we give the design and implementation details of the pR components.

4.1 The Analyzer

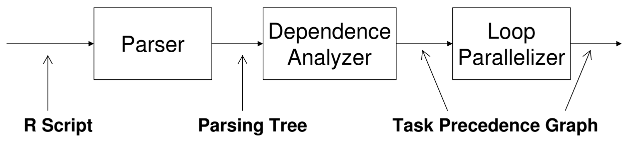

Figure 3. The pR analyzer's internal structure.

As illustrated in Figure 3, there are three basic modules inside the analyzer: a *parser*, a *dependence analyzer*, and a *loop parallelizer.* Below we discuss the task performed by each of these modules.

Parsing The pR parser focuses on identifying R tasks for subsequent execution. Given an input R script, the parser carries out a pre-processing pass of the code using R's internal lexical functions to identify tokens and statements. The parser then breaks down the script into a hierarchical structure and outputs a *parse tree.*

Although the parsing task sounds very similar to the one performed by compilers, the parse tree generated by the pR parser is a simplified version. Here the sole purpose of parsing is to perform dependence analysis and automatic parallelization, while the actual interpretation and evaluation of R statements are carried out by the native R environment. Because the basic unit of pR's task scheduling is at least one R statement, its parse tree stops at this granularity, with each leaf node representing one individual R statement. For all the leaf nodes, the input and output variable names as well as array subscripts, if any, are extracted and stored.

Each internal node of the parse tree represents a region of statements in the script that share the same entry or exit point. A region is a loop (including nested loops), or a conditional branch, or one or more consecutive other statements. For example, a region that goes from the very beginning of a script till the beginning of the first loop or branch within the same scope forms an internal node in the parse tree. A loop makes up another internal node and inside the loop scope, the same procedure of constructing internal nodes recursively applies. Similarly, a conditional branch node includes the *if* clause and the *else* clause (if there is one) and inside the branch scope for which the parsing sub-tree is recursively built in the same way.

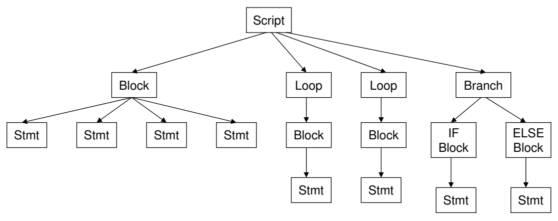

Figure 4. A sample parse tree.

Figure 4 shows the generated parse tree for the sample code given in Figure 1. The leaf nodes are all statements corresponding to lines 1-4, 7, 11, 15, and 19, respectively. Each internal node covers a loop, a branch, or another region in the code, where a "block" stands for a code block that does not contain loops or branches.

Dependence Analysis The pR dependence analyzer takes the parse tree generated by the parser and performs both statement dependence analysis and loop dependence analysis. Because the basic scheduling unit is a task, the dependence analyzer will first group consecutive simple statements (those not containing any function calls) inside non-loop code blocks into tasks. For example, the first two statements in our sample code will become one schedulable R task, which will be executed locally on the master.

In statement dependence analysis, we compare the input and output variables between the statements to identify all three dependency types: true dependence (write-read), anti-dependence (read-write), and output dependence (write-write). Statements dependence analysis is applied across the tasks. If a dependence is discovered between two tasks, the dependence type and the dependent variables are recorded and to be used by the scheduler subsequently.

In loop dependence analysis, we perform the same task as in parallelizing compilers to explore data parallelism inside the loop task. While statement dependence analysis is relatively simple and straightforward to implement, loop dependence analysis is much more challenging and has been studied with many years of research efforts in the compiler community. Exact loop data dependence analysis has been shown to be NP-complete [18]. Our dependence analyzer employs the *gcd test* [5], a method of data-dependence test generally used in automatic program parallelization. The test compares subscripts of two array variables and is linear (affine) in terms of the loop index variables. Basically, it builds a linear Diophantine equation and calculates the greatest common divisor to see whether a solution to that equation exists. The test is effective for simple array subscripts but gets inefficient for complex array subscripts, in which case the compiler may fail to detect the parallelism between loop iterations, and hence, will execute the loop sequentially. Our experience shows that the gcd test suffices for pR's purposes, since typical R scripts used in scientific data analysis are not expected to contain elaborate user-defined loops with a complex array index structure.

pR takes advantage of the fact that the analysis tool can closely interact with the runtime R interpretation environment to perform *incremental analysis*. This allows the analyzer to temporarily pause at points where it requires runtime information to continue with the code analysis and parallelization. In this initial implementation, we define an *pause point* in the dependence analysis as a task that is either a loop with an unknown outer-most loop bound or a conditional branch that contains function calls or loops. These tasks are considered worth parallelization and remote execution, but cannot be parallelized before runtime. When the key evaluation results are returned from the execution engine, the analysis and parallelization resumes and advances to the next pause point or the end of the script.

Finally, when it comes to file I/O operations, pR takes a slightly more aggressive approach than compilers do. Traditionally, compilers do not attempt to parallelize operations involving system calls. In the R context, however, users would greatly benefit from parallelizing the analysis of different files (such as a batch of time-series simulation results). This type of processing is very common and the operations across different files are typically independent. Therefore, we perform a simple file name check when determining whether there exists dependence between tasks containing file I/O operations.[1] Any two operations working on the same files are considered dependent. Further, all calls of user-defined functions, each considered as one single task, are considered dependent with any task that performs I/O.

The precedence relationship among the tasks is stored in a *task precedence graph (TPG)*, as the output of the dependence analyzer. A TPG is essen-

[1]This solution assumes that there are no file aliasing problems, for example, that created by symbolic links.

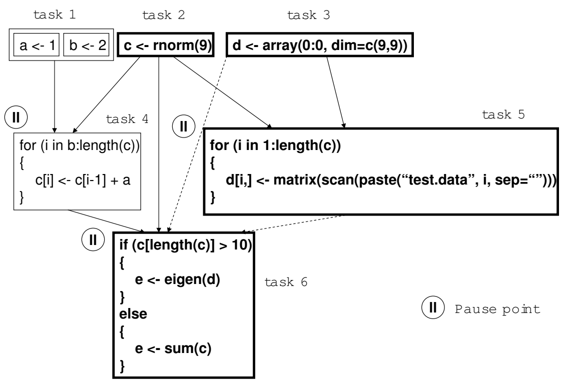

Figure 5. A sample task precedence graph.

tially a directed acyclic graph, where each vertex represents one task and each directed edge represents the dependence between two tasks. In addition, if the task group consists of a loop, the dependence distance (defined as the difference between dependent iteration numbers) for each loop index will also be recorded. This information is not used in the current implementation, but will be useful in future extensions performing loop transformation to parallelize loops where limited inter-iteration dependence exists.

Figure 5 shows the task precedence graph generated from the parse tree in Figure 4. Each box stands for one task and each edge stands for a known dependence. There are three pause points, as marked in this TPG. A dashed edge indicates that the dependence relation might hold, depending on the result of branch condition evaluation. In this figure, the loop task in a bold box denotes a parallelizable loop, while a simple task in a bold box denotes an expensive operation (a function call) that should be outsourced to a worker. Here the loop in task 4 cannot be parallelized because it possesses inter-iteration data dependence.

Loop Parallelization Loops are parallelized automatically by our system if no loop dependency is identified. Similar to parallelizing compilers, we focused on the outer-most loop for nested loops. If necessary, mature techniques such as loop interchange [4] can be applied to better exploit data parallelism, though our current implementation does not support this feature. Currently, pR does not further parallelize the content of the loop (even if there are parallelizable operations such as function calls). Basically, the loops are executed in an embarrassingly parallel fashion with all the statements inside the loop executed by a single processor on processor-specific portion of the data.

For all the parallelizable loop tasks, the iterations of the outer-most loop are split into disjoint blocks and executed in parallel. In this initial prototype, we simply set the number of blocks as the number of workers. Consequently, the original loop task is divided into multiple tasks, each with a similar-sized block of iterations from the original loop (some of the tasks will have a few more iterations than the others). The corresponding TPG vertex is split into multiple vertices, with the dependence edges replicated for each of them. Also, the start and end indices of its sub-loop are stored in each of the newly created TPG vertices.

4.2 The Parallel Execution Engine

Our parallel execution engine is responsible for executing the tasks. As mentioned earlier, it adopts an asymmetric, master-worker model. The workers are responsible for executing tasks that our system considers heavy-weight, namely parallelized loops and function calls within our current implementation. The master, on the other hand, plays two roles. Besides performing all the analysis, scheduling, and worker coordination, it also possesses a local R process that executes light-weight tasks. This simplifies scheduling as well as reduces communication overhead.

The parallel execution engine takes the task precedence graph from the dependence analyzer and makes a scheduling table for dispatching tasks, which essentially holds job information packet entries. Each job information packet consists of the R statements in the task, a map of variables that this task depends on along with the tasks where these variables are last modified and a list of output variables. The job information packet is distributed either to workers if the task in question is considered heavy-weight or to the master, otherwise. Therefore, the execution engine maintains two separate ready queues, for the workers and for the master, respectively. It also keeps a list of all the free workers and the status of the master, i.e., whether the master is executing an R task or is idle. Whenever a worker or the master is available, the first task in the corresponding ready queue is dispatched by sending the job information packet. This process continues until the execution of the entire script is complete, upon which the master will instruct all the workers to quit.

The key challenge in coordinating the parallel execution of R tasks is to overlap the transfer of data generated from previously completed tasks with the computation of the currently active tasks. For example, suppose worker w_1 executes task 2 ("`c <- rnorm(9)`") in Figure 5. The master can direct w_1 to send array `c` to itself because it knows that task 4 (the loop that cannot be parallelized) will be executed on the master locally. However, upon the completion of task 2 it may not be known where tasks 5 and 6 will be assigned, since these tasks depend on another task (task 3), which may be waiting for an idle node or currently under processing.

One way to handle this problem is to let each worker send back its output data to the master since it does not know which worker will later need the data. The master then sends such data to the appropriate worker when the corresponding task is scheduled. This solution may significantly increase the communication traffic and can potentially make the master node a bottleneck. Therefore, we make the workers responsible for the management and peer-to-peer communication of the task input/output data, which are accomplished by the worker front-end process. This process does not execute any R task and is alert for incoming messages both from the master and from the other workers. In the above example, the R process on w_1 will hand over array `c` to the front-end process on the same node, and w_1 will report itself ready for the next task. When the master schedules task 5 to w_2, it tells w_2, as a part of the job information packet, that array `t` is to be retrieved from w_1. w_2 will then contact w_1 to ask for the array, without interrupting the R task processing on w_1.

The parallel execution engine is also responsible for sending runtime information back to the analyzer and triggering the analysis to resume when key variables at a pause point have been evaluated.

5 Experimental Results

In this section, we demonstrate the ease of use and efficiency of the pR framework. We show that pR can be easily used without modifying the sequential R script. We also illustrate pR's performance using several R codes. Note the emphasis of this paper is not on achieving the highest possible speedup and our current implementation is an early proof-of-concept prototype that does not contain sufficient optimization or refinement. However, we show that pR is able to achieve reasonable speedup on a real-world computation-intensive R application, while matching or exceeding the performance of an existing parallel R package, which does require code modification.

5.1 Experiment Settings

Our experiments were performed on the *opt*^{64} cluster located at NCSU, which has 16 2-way SMP nodes, each with two dual-core AMD Opteron 265 processors. The nodes have 2GB memory each and are connected using Gigabit Ethernet and run Fedora Core 5. A single NFS server manages 750GB of shared RAID storage.

We performed each test multiple times and observed that the performance variance was very small (less than 5%), so error bars were omitted.

5.2 Ease of Use Demonstration

a <- matrix(1:1000, 100, 10)	*library(Rmpi)*
b <- list()	*library(snow)*
c <- mean(a)	
d <- sum(a)	*cl <- makeCluster(2, type = "MPI")*
for (i in 1:dim(a)[1])	a <- matrix(1:1000, 100, 10)
{	b <-list()
b[i] <- sum(a[i,])	c <- mean(a)
}	d <- sum(a)
	b <- parApply(cl, a, 1, sum)
	stopCluster(cl)

Figure 6. Comparison with the snow package interface.

To illustrate the advantage of pR's interface, we compare it with the snow (Simple Network Of Workstations) parallel R package [22], which allows users to parallelize embarrassingly parallel operations. Figure 6 lists, side by side, the sequential version and the snow version of a small piece of sample R code. These two codes perform the same R operations and generate the same results.

Statements that call snow APIs are printed in italic. With snow, users must first include both the RMPI and the snow libraries, then indicate that a cluster consists of two processors using `makeCluster`. The user then executes the function `sum` on the target matrix along the 1st dimension in parallel on this cluster using `parApply`. In this case the results will be stored as the 1000 elements of the list `b`. Finally the user has to remove the cluster by calling `stopCluster`.

The sequential version on the left side, however, carries out the `sum` operations in a loop, as typically will be done to perform such a task. With pR, this sequential version can be automatically parallelized without any modification as an ordinary MPI job. Suppose the script is stored in file `sum.R`, the regular command to execute it in batch mode with R is

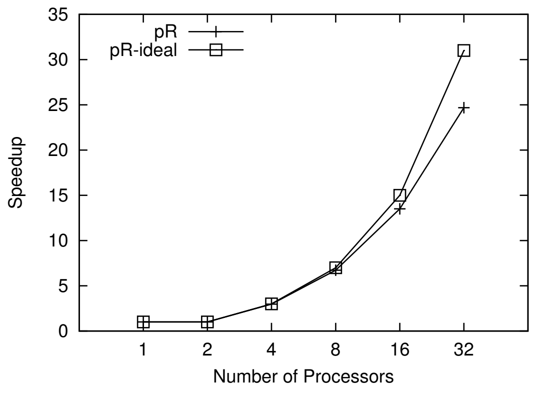

Figure 7. Performance of pR with the Boost application.

```
R CMD BATCH sum.R,
```

then the command to execute it in parallel with pR is simply

```
mpirun -np <num_procs> pR sum.R
```

This allows the sequential code to run unmodified, which has not been enabled by any existing parallel scripting language environments.

The only assumption pR makes for the parallel execution is that all the files used in the sequential script must be stored in the shared file system and the appropriate paths are provided in the code.

5.3 Parallel Execution Performance

First, we evaluate pR using Boost, a real-world application that we acquired from the Statistics Department at NCSU. This code is a simulation study evaluating an in-house boosting algorithm for the nonlinear transformation model with censored survival data. The nonlinear transformation model is complex, and the boosting algorithm is computationally intensive. Moreover, the simulation study often requires a large number of repeated data generation and model fitting, and the total computational time can be forbidding.

The bulk of computation in Boost is spent on a loop, which contains other loops. The only modification we made to Boost before running it in pR is to change the number of iterations in one inner loop (which is not parallelized) to reduce the execution time, as the original code runs for dozens of hours.

Figure 7 shows the speedup of running Boost with pR, on 2 to 32 processors, with the "1 processor" data point marking the sequential running time of Boost in the native R environment. We also plot the ideal speedup for pR, which grows linearly with the number of workers (note that the master does not carry out any heavy-weight computation). For example, with 8 processors the ideal speedup is 7. The results indicate that the actual pR performance, including all the preprocessing, analysis, and scheduling overhead, follows the ideal speedup pretty well, until when there are 15 workers. Up to this point, the R task computation time still decreases linearly, but the pR initialization and data communication overhead becomes more significant (Table 1 will give more details). The overall speedup with 15 workers is 13.5. When the number of processors is creased to 32, the gap between the ideal speedup and the pR actual performance widens: the actual speedup is 24.7 rather than the ideal speedup of 31. This is mainly due to the fact that the contention between the two processors on each SMP node, as the computation speedup (the speedup in executing Boost's main loop) drops to around 1.5 from 16 to 32 processors. Meanwhile, the pR overhead also increases when both processors on a node are used.

Next, we compare pR's performance with that of the snow package. We select two representative synthetic test cases here.

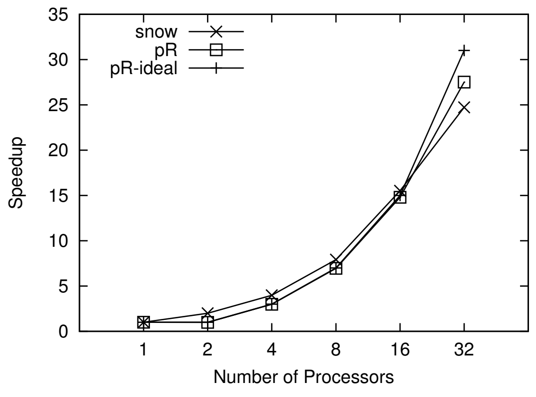

Figure 8. Performance with the bootstrap code.

The first test case was the bootstrap example taken from an online snow tutorial [28]. It performs bootstrap in a loop using the R boot function and the nuclear data provided in R. This forms an ideal case for parallelization, as it is computation-intensive but not data-intensive. We created the corresponding sequential code using a for loop. Figure 8 portraits the performance of snow and pR.

We can see from the figure that both snow and pR perform well with the bootstrap code. The pR curve closely follows the ideal speedup line until the 32-processor point, where the hardware contention becomes heavier. Initially, snow outperforms pR because snow uses all of the processors in running the parallelized operations. With 32 processors, however, pR slightly beats snow.

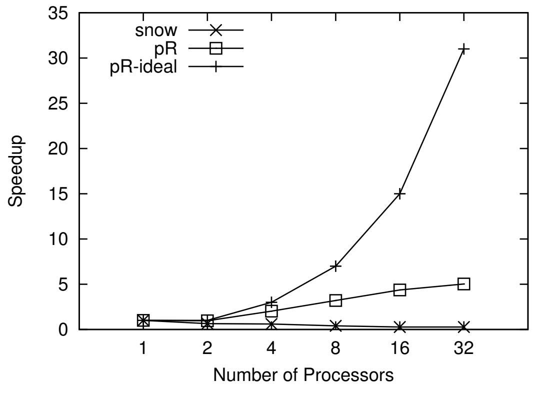

Figure 9. Performance with the SVD code.

The second synthetic test case resembles the example we gave in Figure 6, but at a much larger scale. It performs SVD on each 2-D slide in a large 3-D array. In this case, the initialized array must be partitioned and distributed to all the workers, while the results must be gathered. This code is both computation- and data-intensive.

Figure 9 shows that the speedup achieved by pR is significantly worse than the ideal value. The parallel performance saturates beyond 16 processors and peaks around 4.2. Still, this performance is over an order of magnitude better than snow's, which never produces any speedup starting from 2 processors and actually slows down the application by over 4 times with 16 and 32. This behavior is consistent with what the snow authors reported with communication-intensive codes [22].

For data to be communicated between processes and interpreted correctly as R objects, both snow and pR uses the serialization function provided by R. This helps to keep the parallelization package high-level and easy to work with R updates. However, we have found through our measurement that the R serialization can be more costly than the inter-processor communication. To verify this, we benchmarked the point-to-point MPI communication time and the R serialization time of an 8MB array. On our test cluster, we measured the MPI bandwidth to be 72.5MB/s, while the R serialization bandwidth is only 1.9MB/s. In pR, since the array initialization function call is treated as one task, one worker performs this initialization, serializes partitions of the array, and sends these partitions to the appropriate workers. Therefore the array initialization time remains constant and the communication time increases as the number of workers grows. Such overhead becomes more dominating as more workers are used and the parallel R task execution time shrinks.

The reason that pR's performance is much better than snow, we suspect, is due to the fact that pR is implemented in C and directly issues MPI calls. In contrast, snow is implemented in R itself and calls R's high-level functions for message passing, which may result in worse communication performance.

	2	4	8	16	32
Initialization	0.05	0.13	0.31	0.65	1.28
Analysis	0.00	0.00	0.00	0.01	0.04
Master MPI	0.00	0.00	0.00	0.00	0.01
Max wkr. serial.	0.42	0.69	1.15	2.05	3.19
Max wkr MPI	0.00	0.03	0.07	0.15	0.26
Max wkr socket	0.01	0.01	0.02	0.04	0.05

Table 1. Itemized overhead with the Boost code, in percentage of the total execution time. The sequential execution time of Boost is 2070.7 seconds.

	2	4	8	16	32
Initialization	0.02	0.09	0.17	0.39	0.77
Analysis	0.00	0.00	0.00	0.00	0.01
Master MPI	0.00	0.02	0.00	0.00	0.00
Max wkr serial.	0.00	0.00	0.00	0.00	0.01
Max wkr MPI	0.00	0.00	0.01	0.01	0.00
Max wkr socket	0.00	0.00	0.00	0.00	0.00

Table 2. Itemized overhead with the bootstrap code, in percentage of the total execution time. The sequential execution time of bootstrap is 2918.2 seconds.

	2	4	8	16	32
Initialization	0.23	0.49	0.78	1.12	1.27
Analysis	0.00	0.00	0.00	0.01	0.02
Master MPI	0.00	0.00	0.00	0.00	0.01
Max wkr serial.	11.70	26.46	41.71	52.98	57.98
Max wkr MPI	0.00	2.10	4.32	6.44	7.83
Max wkr socket	1.45	1.56	1.99	2.40	2.51

Table 3. Itemized overhead with the SVD code, in percentage of the total execution time. The sequential execution time of SVD is 227.1 seconds.

Tables 1-3 list the itemized overhead measured from pR tests, in the percentage of the total execution time. E.g., "0.05" in a cell means 0.05% of the total execution time is spent on this particular category of overhead.

We measure six types of pR overhead. "Initialization" includes the cost of initializing the master and the worker processes, performing the initial communication, and loading necessary libraries. "Analysis" includes the total dependence analysis time. "Master MPI" is the sum of time spent on message passing after the initialization phase on the master node. The next three categories stand for the data serialization, inter-node communication (MPI), and intra-node communication (socket), respectively. For each type of operation, we sum up the total overhead spent on such operations on each worker, and then report the maximum value across all the workers.

The first observation we can draw from these tables is that analysis overhead is very insignificant, counting for less than 0.005% in most cases. The slight increase in the relative cost of analysis when there are more workers is more due to the decrease of the overall execution time.

Initialization, on the other hand, steadily increases with the number of workers, because this process involves loading libraries at the workers. This overhead grows as the I/O contention increases, especially with the NFS server equipped at our test cluster. The initialization cost also varies from application to application. Note that the SVD code has a very small initialization cost since it does not load extra libraries (which is not reflected directly in the tables as SVD's execution time is significantly shorter than the other two test cases).

After the initialization phase, the master has little MPI communication overhead, since most of the inter-processor data communication happens between the workers.

The worker-side overhead heavily relies on how data-intensive an application is. For bootstrap, there is almost no data communication between workers, and we measured minimal worker communication overheads. In contrast, with SVD such overheads may dominate the total execution time. With 32 processors, the SVD code spends 58% of the total execution time on data serialization, and a total of around 10% on data communication. This explains the small speedup we observed in Figure 9.

Overall, it appears that the analysis and scheduling protocol of pR is quite efficient, while the data serialization procedure provided by R requires a lot of improvement.

```
a <- array(rnorm(1000000), dim=c(1000,1000))
b <- matrix(scan("test.data"), 1000, 1000)
c <- rnorm (1000)
s <- prcomp(b)
sd <- svd(a)
l <- lm.fit(b,c)
st <- sort(a)
f <- fft(b)
sv <- solve (a,c)
sp <- cor(b, method = "spearman")
q <- qr(a)
```

Figure 10. The task parallelism test code.

Finally, we use another synthetic test case to test pR's capability of parallelizing non-loop tasks. Figure 10 lists the source code. The first three statements create a matrix and a vector with normal distribution, and read a 2-D matrix from an input file. Following those are 8 R function calls that perform a variety of tasks on one or more of these data objects. These tasks include principal components analysis (`prcomp`), SVD (`svd`), linear model fitting (`lm.fit`), variance computation (`cor`), sorting (`sort`), FFT (`fft`), equation solving (`solve`), and QR decomposition (`qr`). The sequential running times of these tasks range from less than 3 seconds for the majority of them, to 19 seconds for SVD and 27 seconds for the linear model fitting. The total sequential time spent on the three data object initialization statements is around 5 seconds.

	1	2	4	8
Exec time	87.69	114.86	45.8	37.9
Speedup	1	0.76	1.91	2.31

Table 4. The parallel execution time and speedup of the task parallelism test script.

When pR is used to run this test case, the three data initialization tasks can be parallelized, and after these data objects are ready, all the 8 computation tasks are independent of each other and can be fully parallelized. This type of parallelization cannot be performed by snow or tools using the back-end support approach.

Table 4 shows the results. Due to the small number of tasks, we stopped at 8 processors. With 2 processors, the single worker is carrying out all the work, while the serialization involved in communicating the data back and forth between the R process and the worker front-end process adds significant overhead. This causes the overall execution time to grow by 31%. With more processors, the parallel execution performance picks up and pR achieves a speedup of 2.3 with 8 processors. Considering that the longest execution path (including the initialization of `b` and `c`, and the `lm.fit` call) costs 30.1 seconds, the total execution time with 8 processors at 37.9 seconds is reasonable given the known high expense of the R serialization.

6 Discussions

In this section, we describe several limitations of the pR design, and discuss related plans for future work.

First, like most parallelizing compilers, our loop parallelization only deals with *for* loops and only partitions the outer-most loop at this point. This may heavily restrict the parallelism available for pR to exploit. One future direction we are considering is to apply existing compiler techniques such as loop interchange and loop flattening [4] to explore inner-loop parallelism.

Second, our current design does not perform load balancing. With a total of p processors, each parallelizable loop is cut into $p-1$ similar-sized partitions, one for each worker. However, some workers may be assigned additional heavy-weight tasks such as function calls. This may cause load imbalance and calls for finer-granule loop partitioning, or non-uniform partitioning plus more intelligent scheduling algorithms. Similarly, in the current design we do not perform data locality-aware scheduling optimizations, which are among our future work items. Further, load balance can be improved if the master is more involved in executing R tasks. It appears from our experiments that the master has a very light workload, especially with a small number of workers. Although it is not worthwhile to distribute light-weight tasks remotely, it helps to let the master share heavy-weight tasks.

Also, the current framework executes each function call as one unit. This limits the parallel execution performance if a script contains a small number of very expensive function calls. Having more workers will not be able to help in this case. We plan to combine pR with existing parallelized back-end engines such as RScaLAPACK [29]. This will allow each heavy-weight standard function call to be executed in parallel, in addition to its task and loop parallelization, while keeping both types of parallelization transparent to users. This, however, generalizes the load balancing problem discussed above and increases the scheduling challenge. When there are multiple independent tasks and each one can be parallelized itself, pR must decide how many processors to allocate to each task, in order to shorten the overall execution time. Existing scheduling algorithms for this type of optimization often require the cost of each task to be known in advance. We believe an on-the-fly performance monitoring module will provide dynamic cost observations and will work well with an interpretation environment.

Finally, as mentioned earlier, the data serialization procedure needs to be optimized to generate better performance for data-intensive applications. Extending pR to work in the interactive mode is another important and challenging task. We also plan to obtain more real-world applications to evaluate pR, especially those with a mixture of task parallelism and loop parallelism, which has not been sufficiently evaluated in this paper.

7 Conclusions

In this paper, we presented pR, a framework that automatically and transparently parallelizes the R language for high-performance statistical computing. We illustrated that scripting languages like R possess unique characteristics and use patterns that facilitate automatic parallelization. Performance results with both real-world and synthetic R codes show that pR achieves good speedup and reasonable scalability in most cases, without any modifications to the sequential script. Environments like this can improve scientists' data processing productivity and boost the currently handled size of the problems to a more realistic scale without imposing requirements for explicit parallel programming.

References

[1] http://www.insightful.com/.

[2] http://www.ittvis.com/.

[3] http://www.openmp.org/drupal/.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[5] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[6] Bioconductor Core. *An Overview of Projects in Computing for Genomic Analysis.*, 2002.

[7] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, number 1375, pages 10.1 – 10.18, 1994.

[8] Michael K. Chen and Kunle Olukotun. The jrpm system for dynamically parallelizing java programs. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 434–446, New York, NY, USA, 2003. ACM Press.

[9] J. Choi, J. Dongarra, R. Pozo, and D. Walker. Scalapack: a scalable linear algebra library for distributed memoryconcurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[10] R. Choy and A. Edelman. Parallel matlab: doing it right. *Proceedings of the IEEE*, 93(2), 2005.

[11] Ron Choy, Alan Edelman, John R. Gilbert, Viral Shah, and David Cheng. Star-P: High productivity parallel computing. In *Proceedings of the Eighth Annual Workshop on High Performance Embedded Computing (HPEC 2004)*, 2004.

[12] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling java just in time. *IEEE Micro*, 17(3), 1997.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, pages 137–150, 2004.

[14] Z. Du, C. Lim, X. Li, C. Yang, Q. Zhao, and T. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, 2004.

[15] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in dyc. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, 1999.

[16] Manish Gupta and Rahul Nim. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–12, Washington, DC, USA, 1998. IEEE Computer Society.

[17] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.

[18] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 1–14, New York, NY, USA, 1991. ACM Press.

[19] Message Passing Interface Forum. *MPI: Message-Passing Interface Standard*, June 1995.

[20] Patrick Miller. Parallel, distributed scripting with Python. In *The Third LCI International Conference on Linux Clusters*, 2002.

[21] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2005. ISBN 3-900051-07-0.

[22] A. Rossini, L. Tierney, and N. Li. Simple parallel statistical computing. *UW Biostatistics working paper series*, 2003.

[23] Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40(5):603–612, 1991.

[24] N. Samatova et al. High performance statistical computing with parallel R: applications to biology and climate modelling. *Journal of Physics: Conference Series*, (46), 2006.

[25] M. Schwartz, N. Delisle, and Vimal S. Begwani. Incremental compilation in magpie. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, 1984.

[26] Frank J. Seinstra, Dennis Koelma, and Jan-Mark Geusebroek. A software architecture for user transparent parallel image processing on mimd computers. In *Euro-Par*, pages 653–662, 2001.

[27] taskPR: Task-parallel r package. http://cran.r-project.org/src/contrib/Descriptions/taskPR.html.

[28] L. Tierney. Simple network of workstations for r. http://www.stat.uiowa.edu/̃luke/R/cluster/cluster.html.

[29] S. Yoginath, N. Samatova, D. Bauer, G. Kora, G. Fann, and A. Geist. RScaLAPACK: High performance parallel statistical computing with R and ScaLAPACK. In *Proceedings of the 18th International Conference on Parallel and Distributed Computing Systems*, 2005.