Abstract

Scripting languages such as R and Matlab are widely used in scientific data processing. As the data volume 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 scripting languages’ interpreted nature and data analysis codes’ use pattern, we propose several novel techniques: (1) applying parallelizing compiler technology to runtime, whole-program dependence analysis of scripting languages, (2) incremental code analysis assisted with evaluation results, and (3) runtime parallelization of file accesses. Our framework 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 transparently and overall has better performance as well as scalability compared to an existing parallel R package that requires code modification.

Keywords: Runtime parallelization, incremental analysis, scripting languages.

1. Introduction

Ultra-scale simulations and high-throughput experiments have become increasingly data-intensive, routinely producing terabytes or even petabytes of data. While most of the research efforts in scientific computing have focused on the data producer side, the growing needs for high-performance data analytics have not been addressed sufficiently.

The work was done while Jiangtian Li was a graduate student at North Carolina State University.
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. However, scientific data analytics currently lacks the proper statistical tools or libraries that could efficiently exploit this parallelism. Unlike simulations, which commonly use general-purpose, compiled languages (such as C, C++, and Fortran), as well as well-adopted parallel interfaces (such as MPI and OpenMP), data processing programs are often written in scripting languages and are currently much less likely parallelized.

In fact, people have recognized the pressing need for parallel data analytics and there have been a number of projects working on parallel tools/libraries for popular scripting languages used by scientists, such as Matlab [1] and R [2] (Section 2.2 gives detailed description of related existing approaches). However, the majority of long-running data processing scripts today remain unparallelized.

Through our interaction with domain scientists, we identified two major factors contributing to the scientists' reluctance in using existing parallel scripting language tools or libraries. First, there lacks support for transparent parallelization that would enable domain scientists to reuse, in a commodity parallel computing environment, their existing sequential data analysis codes with little or no changes. Throughout their scientific careers, 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 does not require explicit parallel programming is highly desirable.

Second, there is not sufficient support to exploit parallelism at multiple levels by an individual tool/library. In most data processing scripts, both task parallelism and data parallelism are present. For example, computing the eigenvalue decomposition and generating a histogram of a matrix, two common tasks, can be perfectly overlapped with each other and executed in a task parallel fashion. Likewise, a loop processing simulation output files generated in a series of timesteps or a loop processing the elements of a large array may be broken into blocks and executed in a data parallel manner. Currently, existing tools/libraries each only focuses on exploiting one type of parallelism.

In an attempt to approach the above problems, we describe in this paper an automatic and transparent runtime parallelization framework for scripting languages. We argue that existing data dependence analysis techniques developed for compiled languages are more than adequate in parallelizing loops and function calls used in data processing programs written in a scripting language. Meanwhile, the interpreted nature of scripting languages provides a very favorable environment for runtime, more aggressive dependence analysis that often cannot be afforded by compilers. By using runtime, full-program dependence analysis, our proposed framework does not require any source code modification, and exploits both task parallelism and data parallelism.

Although the framework and the techniques we propose are general-purpose
for scripting languages, our discussion and implementation target R, a popular scripting language for statistical computing used in many computational science domains (more background information about R will be given in Section 2.1). We have designed and implemented pR, a modified R environment that executes R scripts in parallel. This paper discusses many unique issues that we have encountered through the development of this prototype, on transparently parallelizing a scripting language.

We consider the major contributions, as well as key novelties of this paper to be as follows:

- To our best knowledge, this is the first work that applies parallelizing compiler technology to parallelizing an interpreted language and couples it with incremental runtime analysis. Meanwhile, the runtime analysis used in pR surpasses parallelizing compilers by exploiting both task and data parallelism and even parallelizing I/O operations, rather than focusing on loops only.

- We implemented and integrated our proposed techniques, in the pR prototype, and designed for pR an master-worker scheme for efficient parallel task and data communication. To our best knowledge, pR is the first transparent parallelizing tool for a scripting language without using special hardware or virtual machine support.

- We evaluated the pR prototype with six benchmarks, including two real-world data analysis applications and four synthetic codes. Our experiments demonstrate that a significant speedup over the sequential execution can be achieved and the parallelizing overhead is reasonably 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 [2] 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 developed by its core development team and provides add-on hooks for external developers to add extension packages. 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/.

R can be used in both interactive and batch modes. In the interactive mode, R works via a command-line interface. The user retrieves the results from the
Fig. 1: A sample R script.

```r
1 a <- 1
2 b <- 2
3 c <- rnorm(9)
4 d <- array(0:0, dim=c(9,9))
5 for (i in 1:length(c)) { c[i] <- c[i-1] + a }
6 for (i in 1:length(c)) {
7   d[i,j] <- matrix(scan(paste("test.data", i, sep="")))
8 }
9 if (c[length(c)] > 10) { e <- eigen(d)
10 } else { e <- sum(c) }
```

standard output by typing variable names. In the batch mode, an R script is supplied as a file and is 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 parallelization of the batch-mode execution.

In Fig. 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 as- signment. Line 3 generates a vector of nine normally distributed real numbers. Line 4 initializes a 2-D array. Line 5 is a loop performing a simple arithmetic operation. Lines 6-8 read the matrix data from files. Lines 9-10 are a conditional branch. The first branch computes eigenvalues and eigenvectors of the rectangular matrix and the second one 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. In 2005, Choy and Edelman [3] 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 [4] to carry out Matlab routines in parallel on multiple servers. Star-P [5], 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 enhance performance by transforming Matlab scripts into intermediate code or executable code to avoid the overhead of interpreting. Among them, the Match Virtual Machine [6] is a runtime system that can compile and automatically execute Matlab script in parallel. This work is perhaps the closest to ours in performing
runtime automatic parallelization for scripting languages. MaJIC [7] provides a just-in-time and transparent framework for compiling Matlab while maintaining its interactive nature. Otter [8] is a compiler that translates Matlab scripts into parallel C programs using the MPI library. However, these approaches require special compiler or virtual machine support to translate Matlab programs into codes in a compiled language or directly into parallel executables. While efficient, these approaches are less portable and require complex type analysis.

Similar categorization applies to existing work on other scripting languages. For example, pyMPI [9] provides MPI interfaces for parallel programming in Python and belongs to message passing category. For R, the snow package [10] exploits embarrassingly parallel tasks and belongs to category 1, while RScALAPACK [11, 12] uses the ScaLAPACK library along with the dynamic process management for parallel computation and could be broadly considered to fall under category 3. There also exists an R compiler that translates function definition and control flow in R to C [13].

In contrast, our proposed framework, as first proposed in [14, 15], is complementary to all the aforementioned categories. It parallelizes sequential R code without requiring any source code modification or using compilers. Similar to the task-pR package [16], 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 detects and exploits loop-level parallelism, (b) it performs whole-program, incremental dependence analysis, and (c) it parallelizes file operations. In addition, pR’s runtime parallelization can be incorporated with back-end support to speed up expensive standard function calls.

The snow package [10] 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 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.

To a certain extent, pR resembles OpenMP [17], 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 [18]. 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. Representative work on parallelizing compilers, such as SUIF [19] and Polaris [20] transform sequential C or Fortran code into parallel code (e.g., using OpenMP). The design of pR has borrowed mature techniques from these projects, such as loop dependence analysis and
loop transformation [19, 20, 21, 22, 23, 24, 25]. 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 limited by the lack of information at compile time, while pR is assisted with runtime dependence analysis. While parallelizing compilers deal with system calls such as I/O operations in a conservative way by executing them sequentially, we exploit parallelism in I/O operations by identifying file access and parallelizing I/O operations accessing different physical files.

Many runtime parallelization techniques have been proposed, as summarized by Rauchwerger [26]. Chen et al. proposed an algorithm for run-time parallelization by inspecting memory access patterns first [27]. Saltz 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 [28]. Gupta et al. presented a set of run-time tests for speculative parallelization of loops [29]. Most of these approaches focus on exploiting parallelism from loops where there exist certain forms of inter-iteration dependence. In contrast, our work moves the static analysis performed by compilers to run time, with the assistance of runtime information, and will not parallelize a loop if its iterations are not totally independent.

Chen et al. proposed the Java runtime parallelizing machine (Jrpm), a complete system for parallelizing loops in sequential Java programs automatically [30]. Like other systems using speculative multi-threading (e.g., [31]), 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 [32] and incremental compilation [33] are designed to reduce the overhead of runtime bytecode interpretation or interactive compilation. Our work is closer to dynamic compiling systems such as DyC [34]. 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.

Finally, existing automatic parallelization tools typically do not parallelize I/O operations. For example, with Otter [8], file operations are handled by one single processor. pR takes advantage of runtime information to parallelize I/O, which not only utilizes underlying parallel I/O bandwidth, but also prevents I/O operations from becoming serialization points.

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 [3] 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 certain tasks repeatedly, often processing a collection of data files. These “coarse-grained” operations, compared with “fine-grained” loops used in numerical functions, typically have less inter-iteration dependence and higher per-iteration execution cost, making them ideal candidates for data-level parallelization.

- Several characteristics of the R language simplify dependence analysis. First, there are no explicit data pointers. The input parameters to the functions are read-only, while the modified or newly-created variables by the function are returned explicitly. 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.

The key innovation in the pR design is coupling existing compiler technology with runtime analysis for scripting languages. We perform dynamic dependence 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. Particularly, with runtime file information check we can parallelize I/O operations, at least with a file-level granularity. In addition, more aggressive and complete parallelization is achieved through incremental analysis that delays the processing of conditional branches and dynamic loop bounds until the related variables are evaluated.

---

1R does provide several types of reference objects, whose handling is described in Section 4.1.
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 framework is built on top of and does not require any modifications to the native R environment. The only external package needed by pR is the standard MPI library [35], for inter-processor 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. 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 function call, or a block of other statements between these two types of objects. As shown in Fig. 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 to help pR identify execution units and their precedence relationship. 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.
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 Fig. 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-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. These components bear many similarities with those in parallelizing compilers. Since our work does not aim at advancing compiler technology, we try to give a complete system overview yet focusing on unique issues related to runtime parallelization of a scripting language.

4.1. The Analyzer

As illustrated in Fig. 3, there are three basic modules inside the analyzer: a parser, a dependence analyzer, and a loop parallelizer. The final output is a task precedence graph (TPG). TPG is essentially a directed acyclic graph, where each vertex represents one task and each directed edge represents the dependence between two tasks. Below we discuss the task performed by each of these modules.

4.1.1. Parsing R code

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.

The parse tree generated by the pR parser is a simplified version compared to those by compilers. 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.

Fig. 4: A sample parse tree.

Fig. 5: A sample task precedence graph. In the task precedence graph, the dependence analysis first pauses at task 4 since the loop bound is unknown at that point. The analysis resumes when task 2 completes and the loop bound is evaluated. A similar process repeats for all the other pause points.
Fig. 4 shows the generated parse tree for the sample code given in Fig. 1. The leaf nodes are single statement 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.

4.1.2. Runtime incremental 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 dependence types: true dependence (write-read), anti-dependence (read-write), and output dependence (write-write). Statements dependence analysis is applied across the tasks.

One unique contribution of pR is file operation parallelization, where we take a 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 runtime file check when determining whether there exists dependence between tasks containing file I/O operations. To handle the aliasing problem caused by file links, we recursively follow symbolic links using the `readlink` system call until a regular file or hard link is reached. The inodes of such files are checked to determine the dependence between I/O operations. In the current prototype the file dependence check is coarse-grain and conservative. Any two operations working on the same physical file 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.

In addition, there are several tricky situations in statement dependence analysis. One is caused by R’s “superassignment” operator, which enables global variables to be read or written inside a user-defined function. Unlike compiled languages like C, R does not require variable declaration. Meanwhile, the usage of “superassignment” does not necessarily indicate an access to global variables. Therefore a global variable can not be identified in a single step. Fortunately, R uses lexical scoping and global variables can be identified with static whole-program analysis. This allows pR to fix the problem by parsing user-defined function bodies although the current pR prototype does not parallelize these functions. The pR analyzer traverses each parse tree to identify global variables. If a function accesses global variables, we treat them also as input/output parameters of that function call.

Another issue arises when reference objects such as `environment` are used, since they work as implicit pointers. As a solution, we treat any statement
involving reference objects as a barrier to ensure correctness.

Finally, we identified several types of pR operations that require special handling by the analyzer. These operations are “location-sensitive” in the sense that distributing tasks to different nodes, even when they appear to be independent or when they are dependent but executed in the correct order, will cause a problem. For example, two function calls both using random number generation (such as `rnorm` shown in our sample code) may yield undesirable results if parallelized. Similarly, file accesses may be a problem. Our coarse-granularity file operation dependence analysis considers two subsequent operations dependent if they access the same file. However, these operations, may yield incorrect results if executed distributedly (albeit sequentially) on two workers, depending on the file system implementation. Accessing the reference objects mentioned above is another example. Since pR strives to provide transparent parallelization, our current prototype takes the cautious approach by “binding” such location-sensitive operations to one processor. This is done by the analyzer marking certain edges or adding marked edges if necessary in the TPG to instruct the pR scheduler to ensure each group of such dependent tasks be executed on the same processor.

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. Although exact loop data dependence analysis has been shown to be NP-complete [36], many advanced dependence modeling and loop transformation techniques have been proposed and used in compilers [19, 20, 21, 22, 23, 24, 25].

As mentioned in Section 3.1, typical R scripts used in scientific data analysis are not expected to contain elaborate user-defined loops with a complex array index structure. Therefore many mature loop dependence analysis algorithms suffice for pR’s purposes. In our current prototype, we adopted the gcd test [37], a method of data-dependence test generally used in automatic program parallelization. The test compares subscripts of two array variables which are 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. Our experience up to now with a limited set of time-consuming R applications show that gcd test is powerful enough for user-defined loops in R. Plus, pR adopts widely-used compiler data structures and it is easy to plug in different analysis algorithms, in case future pR users encounter applications that call for more sophisticated analysis.

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 outermost 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 (or parallelized aggressively enough) 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.

The precedence relationship among the tasks is stored in a task precedence graph, as the output of the dependence analyzer. Fig. 5 shows the TPG generated from the parse tree in Fig. 4. Each box stands for one task and each edge stands for a known dependence. There are three pause points, one each for task 4, 5, and 6. 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.

4.1.3. Loop parallelization

Loops are parallelized automatically by our system if no loop dependence is identified. Similar to parallelizing compilers, we focused on the outermost loop for nested loops. If necessary, mature techniques such as loop interchange [38] 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). Each loop iteration is executed by one processor. For all the parallelizable loop tasks, the iterations of the outermost loop are split into 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 to be scheduled independently.

4.2. The Parallel Execution Engine

Our parallel execution engine is responsible for executing the tasks. It is also responsible for sending runtime information back to the analyzer and triggering the incremental analysis to resume when key variables at a pause point have been evaluated. As mentioned earlier, we adopt an asymmetric, master-worker model. The workers are responsible for executing heavy-weight tasks, i.e., loops and function calls. 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 schedules the tasks accordingly, dispatching a ready task whenever there is an idle node. A task is distributed either to a worker if it 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. In many cases, this rather brute-force scheduling approach
may not yield the optimal schedule. Related in-progress and future scheduling studies are discussed in Section 6.

The key implementation 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 Fig. 5. This task produces array c, which is needed as input by three tasks, namely task 4, 5, and 6. 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 $w_1$ may not be known where tasks 5 and 6 are assigned: these two tasks may have not yet been scheduled as they depend on another waiting or running task (task 3).

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 tasks 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 using inter-process communication, and report itself ready for the next task. When the master schedules task 5 to $w_2$, it tells $w_2$ that array c is to be retrieved from $w_1$. $w_2$ will then contact the front-end process on $w_1$ to ask for the array, without interrupting the new R task being processed by the R process on $w_1$.

5. Experimental Results

In this section, we demonstrate the ease of use and efficiency of pR. We show that pR can be easily used without modifying sequential R scripts, and then discuss pR’s performance and scalability. We have evaluated pR with six benchmarks, including two real-world applications we obtained from scientists and four synthetic programs. The six test cases presented here carry different application characteristics: data parallelism with or without heavy data communication, task parallelism, and file operations.

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 real-world computation-intensive R applications, 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. Error bars are omitted in the figures as the performance variance observed was very small. For pR, the average and maximum 95\% confidence interval is 0.98\% and 2.86\% respectively (1.44\% and 5.63\% for snow).

5.2. Ease of Use Demonstration

To illustrate the advantage of pR’s interface, we compare it with the snow (Simple Network Of Workstations) parallel R package \[10\], which allows users to parallelize embarrassingly parallel operations. Fig. 6 lists, side by side, the sequential version and the snow version of a small piece of R code taken from a standard R package, adehabitat. 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 explicitly create a “cluster” consisting of four processors using \texttt{makeCluster}. The user then executes the function \texttt{attributes} on the elements of the target list \texttt{l} in parallel on this cluster using \texttt{clusterApply}. In this case the results will be stored as the elements of the list \texttt{la}. Finally the user has to remove the cluster by calling \texttt{stopCluster}. In addition, snow has inconvenient restrictions with parallel interfaces like \texttt{clusterApply}. For example, here the list \texttt{l} is allowed to have at most as many elements as the number of nodes in the cluster.

The sequential version on the left side, however, carries out the \texttt{attributes} 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 an script is stored in file `example.R`, the regular command to execute it in batch mode with R is

```
R CMD BATCH example.R
```

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

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

This allows a sequential code to run unmodified, a feature not yet provided 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 appropriate paths are provided in the code.

5.3. Parallel Execution Speedup

5.3.1. Evaluation with Synthetic Code

To further demonstrate the ease-to-use feature of pR, we used another small synthetic R code. The code invokes more than one statistical functions in a loop. There is data dependence within each iteration, but not cross iterations. This is indeed a common situation with many programs in either scripting or compiled languages. The independent iterations allow this code to be parallelized in a data-parallel fashion with pR or snow. With pR, no modifications are needed. In contrast, it is not straight-forward to apply the snow interface here. In order to use snow, all the functions inside the iteration have to be wrapped up into one function and passed as a parameter to the snow interface. pR avoids this additional step and provides totally transparent parallelization.

Fig. 7 illustrates the performance of pR with the above small synthetic R code on 2 to 32 processors, with the “1 processor” data point marking the sequential running time 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 speedup of pR follows closely the ideal speedup up to 8 processors. When we use 16 processors or more, the hardware contention on the SMP nodes and the unevenly distributed number of iterations are the major reasons for the gap between the pR and the ideal speedup.

Fig. 7: Parallel execution speedup of pR with one synthetic code, compared with the ideal pR speedup.
5.3.2. Evaluation with Real Applications

Now we evaluate pR using two real applications. The first application is Boost, a real-world application 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. 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.

The second real-world application that we used to evaluate pR is obtained from the Computer Science Department at Duke University. This R code supports a missing data problem in sensor networks, where reading reports are either received, or not received due to failure or suppression [39]. The main body of this sensor network R code consists of a loop of 8 iterations, which makes up the bulk of its computation. Each of these iterations performs statistical analysis on different input data and calls several computation-intensive functions. The loop is embarrassingly parallel in nature and therefore ideal for automatic parallelization.

Fig. 8(a) shows the speedup of running Boost with pR on up to 32 processors. 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 increased to 32, the gap between the ideal speedup and the pR actual performance widens: the actual speedup is 24.6 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.

Fig. 8(b) shows the speedup of running the sensor network code with pR. Since the loop has only 8 iterations, we use up to 8 processors. The speedup achieved using pR is decent, reaching 6.15 with 8 processors total. The gap between the pR curve and the ideal speedup curve is due to load imbalance. The independent iterations are not able to be evenly distributed to the workers. Also, there is a large difference between the iteration run time. In our experiment the execution time of each individual iteration varies from 32 to 138 seconds, depending on the input data.

5.3.3. Comparison with snow Package

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

The first test case was the bootstrap example taken from an online snow tutorial [40]. 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. Fig. 9(a) portrays 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.

The second synthetic test case resembles the example we gave in Fig. 6, but at a much larger scale. It performs SVD on each 2-D slice 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.

Fig. 9(b) shows that the speedup achieved by pR is significantly worse than
the ideal value. The parallel performance saturates beyond 16 processors and
peaks around 4.7. 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 be-
havior is consistent with what the snow authors reported with communication-
-intensive codes [10].

For data to be communicated between processes and interpreted correctly
as R objects, both snow and pR uses the serialization function provided by
R which can write an R object to a scalar string. 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.

5.4. pR Overhead Analysis

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

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.05</td>
<td>0.13</td>
<td>0.31</td>
<td>0.65</td>
<td>1.28</td>
</tr>
<tr>
<td>Analysis</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.04</td>
</tr>
<tr>
<td>Master MPI</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Max wkr. serial.</td>
<td>0.42</td>
<td>0.69</td>
<td>1.15</td>
<td>2.05</td>
<td>3.19</td>
</tr>
<tr>
<td>Max wkr MPI</td>
<td>0.00</td>
<td>0.03</td>
<td>0.07</td>
<td>0.15</td>
<td>0.26</td>
</tr>
<tr>
<td>Max wkr socket</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.04</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Tables 1-4 list the itemized overhead measured from pR tests, in the per-
centage 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 com-
munication, and loading necessary libraries. “Analysis” includes the total de-
pendence analysis time. “Master MPI” is the sum of time spent on message
Table 2: Itemized overhead with the sensor network code, in percentage of the total execution time. The sequential execution time is 825.2 seconds.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.09</td>
<td>0.23</td>
<td>0.52</td>
</tr>
<tr>
<td>Analysis</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Master MPI</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Max wkr. serial.</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Max wkr MPI</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Max wkr socket</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 3: Itemized overhead with the bootstrap code, in percentage of the total execution time. The sequential execution time is 2918.2 seconds.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.02</td>
<td>0.09</td>
<td>0.17</td>
<td>0.39</td>
<td>0.77</td>
</tr>
<tr>
<td>Analysis</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Master MPI</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Max wkr serial.</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Max wkr MPI</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>Max wkr socket</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

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 category, we sum up the total overhead on each worker, and then report the maximum value across all the workers.

The first observation we can draw here 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 as more workers are used 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

Table 4: Itemized overhead with the SVD code, in percentage of the total execution time. The sequential execution time is 227.1 seconds.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.23</td>
<td>0.49</td>
<td>0.78</td>
<td>1.12</td>
<td>1.27</td>
</tr>
<tr>
<td>Analysis</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>Master MPI</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Max wkr serial.</td>
<td>11.70</td>
<td>26.46</td>
<td>41.71</td>
<td>52.98</td>
<td>57.98</td>
</tr>
<tr>
<td>Max wkr MPI</td>
<td>0.00</td>
<td>2.10</td>
<td>4.32</td>
<td>6.44</td>
<td>7.83</td>
</tr>
<tr>
<td>Max wkr socket</td>
<td>1.45</td>
<td>1.56</td>
<td>1.99</td>
<td>2.40</td>
<td>2.51</td>
</tr>
</tbody>
</table>
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. This is not reflected directly in the tables as SVD’s execution time is much 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 Fig. 9(b).

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.

### 5.5. Task Parallelization Performance

```r
a <- matrix(scan("testa.data"), 1000, 1000)
b <- matrix(scan("testb.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)
save(sp, file="result1.data")
save(q, file="result2.data")
```

Fig. 10: The task parallelism test code.

Finally, we use another synthetic test case to test pR’s capability of parallelizing non-loop tasks. Fig. 10 lists the source code. The first three statements read two 2-D matrices from two separate input files, and create a vector with normal distribution. 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`). Finally, a subset of results are written back to two output files. The sequential running times of these tasks range from less than 3 seconds for most 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 3 seconds.
Table 5: The parallel execution time and speedup of the task parallelism test script

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exec time</td>
<td>90.37</td>
<td>122.26</td>
<td>49.37</td>
<td>35.82</td>
</tr>
<tr>
<td>Speedup</td>
<td>1</td>
<td>0.74</td>
<td>1.83</td>
<td>2.52</td>
</tr>
</tbody>
</table>

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. The two output operations can also be overlapped with each other or other tasks that they do not depend on. This type of parallelization cannot be performed by snow or tools using the back-end support approach.

Table 5 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 35%. With more processors, the parallel execution performance picks up and pR achieves a speedup of 2.5 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 35.82 seconds is reasonable given the known high expense of the R serialization.

This sample test also demonstrates file dependence analysis and parallel file operations. With the runtime file dependence check turned on and pR used system calls to examine inodes, the total analysis cost was indeed considerably increased, by 81%, from 0.0036 to 0.0066 second. Still, the absolute cost of making the four file operation checks are quite small. The two input operations using scan are scheduled at almost the same time when 4 or more processors are used, and in this case the total execution time of these two tasks is reduced from 2.4 to around 2 seconds. Consider our test cluster uses a single NFS server, we expect to see larger benefit of parallelizing I/O operations on platforms with parallel file system support.

6. Discussions

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

First, like most parallelizing compilers, our loop parallelization only deals with for loops and only partitions the outermost 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 [38] to explore inner-loop parallelism.

Second, our current design does not perform load balancing or cost-aware scheduling. 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, where we can leverage existing approaches [41, 42]. Similarly, we need to look into adding data locality-aware scheduling optimizations [43] for smart task placement. Effective function cost modeling is desirable to perform global scheduling optimizations, and again runtime parallelization provides unique opportunities in this regard. For example, the scheduler may be able to “learn” about the complexity of functions used in the program currently running. We have addressed those issues in more recent followup work [44].

Also, pR 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 [12]. 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, e.g., use binary serialization, to generate better performance for data-intensive applications. In the future work, a more granular I/O access dependence checking needs to be employed to explore more opportunity for parallelism. 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.
Acknowledgments

We greatly appreciate the anonymous reviewers for their valuable comments and suggestions. The research at NCSU was sponsored in part by a DOE ECPI Award (DE-FG02-05ER25685), an NSF CAREER Award (CNS-0546301), an NSF award (CNS-0915861), and Xiaosong Ma’s joint appointment between NCSU and ORNL. The work of Nagiza F. Samatova, Guruprasad Kora and Srikanth Yoganath was funded by the Scientific Data Management Center under the Department of Energy’s Scientific Discovery through Advanced Computing program. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under contract no. DEAC05-00OR22725.

References


