pR: AUTOMATIC PARALLELIZATION OF DATA-PARALLEL STATISTICAL COMPUTING CODES FOR *R* IN HYBRID MULTI-NODE AND MULTI-CORE ENVIRONMENTS

Paul Breimyer[1,2] Guruprasad Kora[2] William Hendrix[1,2] Neil Shah[1,2] Nagiza F. Samatova[1,2,*]

[1]North Carolina State University
890 Oval Drive
Raleigh, NC 27695-8206

[2]Oak Ridge National Laboratory
1 Bethel Valley Rd
Oak Ridge, TN 37830-8050

* Corresponding author: samatovan@ornl.gov

ABSTRACT

The increasing size and complexity of modern scientific data sets challenge the capabilities of traditional statistical computing. High-Performance Statistical Parallel Computing is a promising strategy to address these challenges, especially as multi-core parallel computing architectures become increasingly prevalent. However, parallel statistical computing introduces implementation complexities and, therefore, an automatic parallelization approach would be ideal. Data-parallel statistical computations that aim to evaluate the same function on different subsets of data represent natural candidates for automatic parallelization due to their inherent inter-process independence.

In this paper, we extend the *pR* middleware for the *R* open-source statistical environment to support automatic parallelization of data-parallel tasks in multi-node, multi-core, and hybrid environments. *pR* requires few or no changes to existing serial codes and yielded over 50% end-to-end execution time improvements in our tests, compared to the commonly used *snow R* package.

KEYWORDS

Statistical Computing, Automatic Parallelization, Data-Parallel

1. INTRODUCTION

The growing demand for analyzing data of increasing size and complexity has led to the emergence of the High-Performance Statistical Parallel Computing (HPSPC) field, resulting in the development of several open source and commercial software tools and libraries, such as those that build upon parallel mathematical libraries like *ScaLAPACK* (Choi et al. 1992). However, parallel statistical computing is not a trivial task. An ideal HPSPC system would automatically execute scientists' serial data analysis codes in parallel. However, this holy grail of parallel statistical computing remains elusive. A simpler, yet powerful, approach involves automatic execution of the same task on different subsets of the data in parallel, or *data-parallel statistical computing*, thereby avoiding issues related to inter-process dependencies.

R, a powerful and widely-used open-source statistical computing engine, is a natural target for automatic parallelization of data-parallel statistical computing methods. *R* supports a commonly used family of *apply* methods that independently and serially execute a specified function against each variable in a list. *R* has limited native support for parallel statistical computing. Instead, previous research has produced add-on packages that support high-level parallel implementations of *apply*-like functions that shield users from the low-level details of explicit parallelization. For example, a parallel *lapply* method that accepts a list and a function definition as its arguments can pass the first variable in the list with the specified function to one process, the second element and the same function to the next process, and so on, and gather the derived results. Current projects that implement the parallel *lapply* function include *snow* (L. Tierney et al. 2004) and *multicore* (Urbanek 2009) for multi-node and multi-core environments, respectively. However, to the best of our knowledge, neither of these is designed for hybrid multi-node and multi-core environments.

To address the challenges of parallel statistical computing (sharing compute spaces, protecting users from underlying complexities, etc.), we extend the ***pR*** middleware library to support automatic parallelization of data-parallel statistical computations in *R* using a hybrid approach for both multi-node and multi-core environments. For example, if a cluster has 1024 nodes, and each node has 8 cores, then with few or no changes to the serial *R* code, *pR* will distribute the data equally among the 1024 nodes and implicitly migrate the *R* environment. The work per node is further divided among the 8 cores and each process uses its own copy of the environment, thereby striving for a 'best of both worlds' approach. Using *R*'s *lapply* method, we demonstrate *pR*'s benefits particularly in improved overall performance and transparent parallelization. *pR* is available upon request.

2. BACKGROUND

R (Ihaka & Gentleman 1996) is a command-line driven open-source implementation of the *S* (Becker et al. 1988) statistical environment. *R* is capable of performing diverse statistical tasks including linear regression, classical statistical tests, clustering, and a variety of publication-quality graphical representations. It is widely used in academia and is becoming more common in industry (Schwartz 2004) (Vance 2009). *R* provides extensibility by allowing users to load external codes called *packages*. Among them, *snow*, *multicore*, and *pR* provide parallel implementations of *R* methods from the *apply* family.

Specifically, *snow* (Simple Network of Workstations) (L. Tierney et al. 2004) supports basic parallel computing in *R* and manages communication between compute nodes using several protocols, including *Rmpi* (Hao Yu 2009), an *R* wrapper around a C++ implementation of the MPI (Snir 1995) specification. End-users can apply functions on clusters of compute nodes and retrieve results. Likewise, *multicore* (Urbanek 2009) supports running a parallel *lapply* method in *R* on Unix machines with multiple cores only. The *multicore* package is intended to improve performance on multi-core machines and is not designed for multi-node environments.

pR is an umbrella framework that contains several parallel computing components for *R*, including a compiler parallelization approach to automatically parallelize *R* scripting codes (J. Li et al. 2007) (Ma et al. 2007), and a component that creates a bridge between existing compiled parallel codes and *R* (Breimyer et al. 2009) (Samatova, Breimyer et al. 2008). The work presented in this paper augments these approaches by introducing automatic parallelization of data-parallel statistical codes for *R* in hybrid multi-node and multi-core environments with specific demonstration of its performance using *R*'s *lapply* method. Throughout the rest of this paper, when we refer to *pR*, we are referring to this component of the *pR* framework.

3. APPROACH

3.1 Software Architecture

There are three layers in the *pR* software architecture stack, illustrated in Figure 1, and each layer exists on both the *R* end-user node and compute nodes: the *R* interface, the Environment/Data/Instruction (EDI) layer, and the MPI communication layer. On a compute node, the *R* interface layer executes the user specified instructions on the node-specific data in the transmitted and loaded *R* environment. The EDI layer is responsible for the exchange of the *R* environment, data, and instructions between the *R* end-user session and the compute nodes via the MPI communication layer. Specifically, on the end-user side, it broadcasts the *R* environment and the code instructions from the end-user session to the compute nodes, properly distributes the data by leveraging the *multicore* package, gathers results from the compute nodes, and performs an in-memory transfer of the results to the end-user *R* interface. On the compute node side, the EDI layer launches a compute engine, receives the environment and loads it into the native *R* engine, receives and passes the corresponding data and instructions, and sends the results back to the EDI layer on the end-user node. *pR* manages the cluster and interacts with the communication layer using MPI; it does not require the *R* end-user to explicitly deal with these issues.

It is critical to note that *snow* does not copy the *R* entire environment, meaning that users must explicitly send any dependent data to compute nodes, which can decrease the amount of consumed memory at the cost of user interaction. Both the *multicore* and *pR* packages copy the entire environment; however, *multicore* is limited to a single system, while the *pR* package can distribute the work over many multi-core machines, thereby mitigating single-system resource restraints.

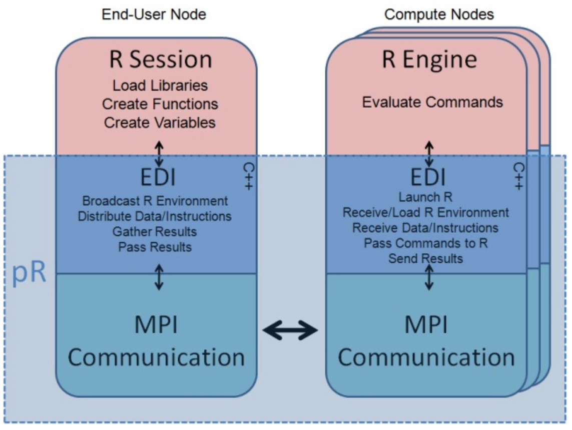

Figure 1 *pR* software stack for automatic data-parallel statistical computation.

The *R* interfaces for the *snow* and *multicore* packages are fairly similar to *pR*'s, except for the distinctions discussed in Section 3.3. The *snow* package launches standard *R* sessions on compute nodes, while *multicore* forks the parent *R* session. While the *snow* middleware layer is written entirely in *R*, the *multicore* package calls *C* code from its custom *R* scripts to fork compute processes and manage the cluster.

3.2 Communication Layer

There are different communication approaches that packages can implement, including sockets, MPI, or system-level pipes (if the processes coexist on the same machine). *pR* interfaces with MPI in compiled C++ due to MPI's widespread availability and design for massively parallel applications. Furthermore, socket communication restrictions are prevalent in secure High-Performance Computing (HPC) environments, and *pR* is designed for multi-node, as well as multi-core, environments, which prevents the use of system level pipes for inter-node communication.

The *snow* package supports MPI communications by leveraging the *Rmpi R* package that provides a wrapper for MPI communications. While *Rmpi* makes MPI calls in compiled C, *Rmpi* users (e.g.: *snow*) interact with the package in the interpreted *R* environment, which introduces some overhead. Therefore, execution will likely take longer than calling the MPI functions directly within a compiled language like *C* or *Fortran*, particularly, if large amounts of data need to be distributed. The *snow* package provides other communication mechanisms, including PVM, sockets, and NetworkSpaces (nws) (Computing 2008), which face similar concerns because they are also managed through the interpreted *R* environment.

The *multicore* project, which is intended for use on single-system multi-core machines, uses direct process-to-process communication through *C* level *read()* and *write()* pipe communications between processes. This approach provides good performance, but cannot support multi-node environments.

3.3 Ease-of-Use

Ease-of-use is a paramount concern within parallel statistical computing environments, especially for systems aiming to protect users from the complexities of parallel computing, such as *pR*, which strives to minimize or avoid parallel computing code. *pR* mimics the *R* function library by overloading *R*'s existing *lapply* function with additional environment-specific parameters. This interface was chosen to give *pR* the look-and-feel of native *R* functions and to smooth the transition for *R* users to parallel *lapply* functions. The *pR* approach strives to improve ease-of-use by not requiring users to explicitly manage compute node environments.

In comparison, *snow* users are responsible for creating and destroying clusters of compute nodes (*makeCluster* and *stopCluster*, respectively), sending self-contained functions or explicitly broadcasting dependent libraries, functions and variables to compute nodes (*clusterEvalQ* and *clusterExport*, respectively), and calling the desired function to be executed.

The *multicore* package exposes a custom *mclapply* method signature, requiring users to change their code to call this method. By default, *multicore* uses the maximum number of cores on the machine. However, there is no standard way to determine this value and *multicore* uses eight cores if it cannot assess a machine's configuration. Users can explicitly specify a different number of cores by setting the *mc.cores* parameter.

3.4 Execution Flow

pR implements a two-tiered execution strategy to take advantage of both multi-node and multi-core environments. In the first tier, *pR* broadcasts the *R* environment, the function to evaluate, sends a subset of the data to each compute node using MPI, and then loads the end-user's *R* environment on each compute node. In the second tier, *pR* leverages the *multicore* package to use Unix's *fork* command to create local processes, and each process opens system-level pipes to read data from the compute node, and write results. Once each compute node receives its results, *pR* uses MPI to send the results back to the end-user node, which assembles the results and returns them to the *R* session.

4. RESULTS

To evaluate the performance of *pR*, we conducted experiments to compare against *R*, and the *multicore* and *snow* parallel computing packages because they are the most active and robust packages that provide parallel *lapply* capabilities in multi-core and multi-node environments, respectively (Schmidberger et al. 2009). The *snow* package is more established and frequently used than *multicore,* and it can leverage the *Rmpi* package to manage communication between compute nodes.

Two test environments were used: E_1 is a dedicated 64 node Intel-based Infiniband cluster and each node has two 3.4 GHz Pentium IV processors and 6GB of memory. The E_2 system is a single node with 4 dual-core AMD processors and 64GB of memory. Both environments use Linux kernel 2.6.18. The E_2 system was used for the multi-core experiments described below. We use the term *processes* in the results below to connote the processes instantiated in either the hybrid E_1 or the multi-core E_2 environment.

4.1 Speedup

We calculate the speedup of each of the *R* packages and compare them to the ideal. To normalize the comparison, we measure speedup as the relative performance between single-processor *R* and the corresponding multi-processor system (i.e., *pR*, *snow*, and *multicore*), defined as: $S(p) = \frac{T_{serial}}{T_{parallel}(p)}$, where T_{serial} is the *R* execution time on one processor and $T_{parallel}(p)$ is the execution time of *pR*, *snow*, or *multicore* using *p* processors. The ideal speedup was determined by dividing the execution time using serial *R* on one processor by the number of processors.

Figure 2 depicts the speedup using a test case that sums 4096 vectors, each of size 2^{24}. Both the *multicore* and *pR* packages perform close to the ideal for 2, 4, and 8 cores, and *pR* closely reflects *multicore*'s performance, implying that *pR* introduces low overhead (data is not shown). The *snow* package yields consistent speedup for 2 and 4 processes, and then speedup decreases. A performance drop-off at 8 cores is observed for all three packages, which may be due to the increased memory latency experienced as the

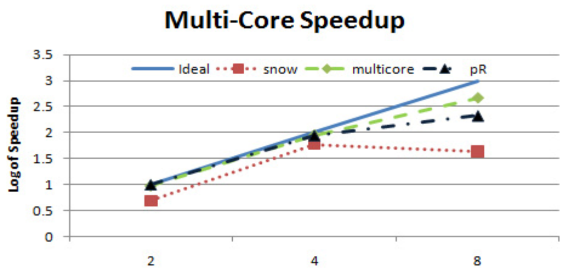

Figure 2 Multi-core speedup for *pR*, *snow*, and *multicore* compared to a serial *R* implementation. The results were gathered in the E2 environment with 8 cores.

number of cores increases; this latency can consume 30% or more of processing time (Murphy 2009) and further investigation is necessary to evaluate this possibility.

We also evaluated the speedup in the hybrid multi-node and multi-core E_1 environment to compare *pR* to *snow*, and the results are shown in Figure 3. *pR* closely mirrors the ideal speedup for all processor values and performs better than *snow*, which appears to level-off at 32 processors. It would be interesting to scale-up the experiment using larger numbers of processors, however, we currently do not have access to such a system. The average performance improvement for *pR* compared to *snow* on this test case was 57%.

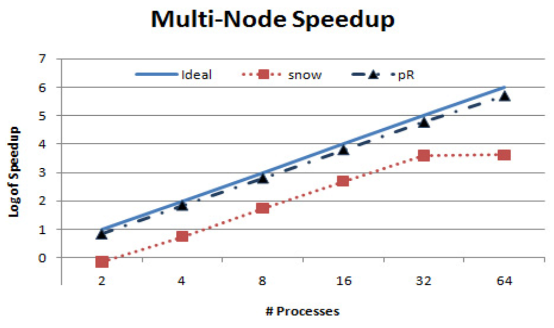

Figure 3 Multi-node speedup for *pR* and *snow* compared to the serial *R* integration implementation.

4.2 Communication Overhead

We expect *pR* to yield reduced communication time compared to *snow* because *pR* calls MPI functions directly in compiled C++, rather than using *Rmpi* through interpreted *R*. The improvement should be more pronounced in communication intensive scenarios. Furthermore, when multiple cores are available per system, we expect *pR*'s communication time to approximately equal *snow*'s communication divided by the number of cores because *pR* will create smaller clusters than *snow*, and then distribute the computations among the cores on each compute node. For example, if a test case needs 1024 processes and each compute node has 256 cores, such as the picoChip PC200 (picoArray 2009), then *snow* will create a cluster of size 1024, whereas *pR* will create a 4 node cluster, and then distribute work among the cores. While the distribution among cores will consume some time, we expect the distribution to be noticeably faster than initializing a larger cluster. This cluster size reduction should impact the time for cluster initialization and teardown, data broadcasts, and all data distribution because the end-user node is communicating with fewer nodes. As expected, using the hybrid E_1 environment in which each compute node has 2 cores, *pR*'s MPI communication overhead averaged 54% less than *snow*'s across powers of 2 processors from 2 to 64.

5. RELATED WORK

There are two general categories of Parallel Paradigms: *implicit* and *explicit* parallelism. In the latter, end-users are responsible for explicitly managing all details of the parallel computation, including designing the parallel algorithm, data distribution, and inter-node communication (ex: MPI). This category grants the end-user the most power and control, at the expense of performance and ease-of-use. Some examples of *explicit* parallelism within *R* include *Rmpi* (Hao Yu 2009) and *rpvm* (M. N. Li & Rossini 2001).

In contrast, *implicit* parallelism places the responsibility of managing the parallel statistical computing details on efficient, compiled codes, tools and libraries, rather than on the end-users. Implicit parallelism can be further divided into *task*-parallelism and *data*-parallelism. *Task*-parallelism involves a heterogeneous set of tasks that can be performed on the same, or different, datasets in parallel. *Task*-parallelism can suffer from under-utilized processors due to the common scenario, where tasks wait for dependencies to be resolved. *taskPR* (Samatova, Bauer et al. 2008) is one example of a task-parallel *R* package; it provides a simple way to detect out-of-order independent execution of *R* tasks and intelligently delegates each task to remote compute node processes to execute them concurrently.

In contrast to *task*-parallelism, *data*-parallelism distributes different subsets of the data across the compute nodes, and performs the same task on each subset. *data*-parallelism can be further divided by the amount of inter-process communication: either intense or light-weight. Approaches that involve intense communication include the *RScaLAPACK* package (Yoginath et al, 2005), a specialized parallel *R* project that exposes *ScaLAPACK* routines within *R*. While *RScaLAPACK* is specific to *ScaLAPACK*, the *pR* component presented in (Breimyer et al. 2009) generalized the approach.

6. CONCLUSION

In this paper we presented an extension to the *pR* package that introduces automatic parallelization of *R*'s *lapply* method for data-parallel statistical methods in hybrid multi-core and multi-node environments. Related packages include the *multicore* package for multi-core environments, and the *snow* package designed for multi-node environments. As demonstrated using *pR*'s implementation of *R*'s *lapply* method, *pR* achieves improved overall end-to-end performance by utilizing a two-tier execution strategy. The first tier allocates work to nodes in a multi-node infrastructure, and the second partitions the work for local execution on individual cores in multi-core environments. The performance results in Section 4 demonstrate that *pR* can outperform *snow* in hybrid multi-core and multi-node environments. Specifically, *pR* achieved an average performance improvement of over 50% compared to *snow* in our tests.

REFERENCES

Becker, R. A., Chambers, J. M., et al., 1988. The New S Language: A Programming Environment for Data Analysis and Graphics. Wadsworth & Brooks, Pacific Grove, CA.

Breimyer, P. et al., 2009. pR: Lightweight, Easy-to-Use Middleware to Plugin Parallel Analytical Computing with R. In *The International Conference on Information and Knowledge Engineering (IKE)*.

Choi, J. et al., 1992. ScaLAPACK: A scalable linear algebra library for distributed memoryconcurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*. pp. 120-127.

Computing, R., 2008. *nws: R functions for NetWorkSpaces and Sleigh*, Available at: http://cran.r-project.org/web/packages/nws/index.html.

Ihaka, R. & Gentleman, R., 1996. R: A Language For Data Analysis And Graphics. *Journal of Computational and Graphical Statistics*, 5, 299-314.

Li, J. et al., 2007. *Automatic, Transparent Runtime Parallelization of the R Scripting Language*, Technical Report TR-2007-3, North Carolina State University.

Li, M.N. & Rossini, A.J., 2001. RPVM: Cluster Statistical Computing in R. *R News*.

Ma, X., Li, J. & Samatova, N.F., 2007. Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. pp. 1-6.

Murphy, R., 2009. DOE's Institute for Advanced Architecture and Algorithms: an Application-Driven Approach. *SciDAC*. Available at: http://www.ebi.ac.uk/sbo/.

picoArray, 2009. *picoArray PC202*, Available at: http://www.picochip.com/products_and_technology /PC202_integrated_communications_and_signal_processor

Samatova, N.F., Bauer, D. & Yoginath, S., 2008. *taskPR*, http://cran.r-project.org/web/packages /taskPR/index.html.

Samatova, N.F., Breimyer, P. et al., 2008. *Parallel R for High Performance Analytics: Applications to Biology*, CRC Press/Taylor and Francis.

Schmidberger, M. et al., 2009. State-of-the-art in Parallel Computing with R.

Schwartz, M., 2004. The Decision to Use R: A Consulting Business Perspective. *R News*, 4, 2-5.

Snir, M., Otto, S.W., et al., 1995. MPI: The complete reference. MIT Press Cambridge, MA, USA.

Tierney, L. et al., 2004. *The snow Package*, Available at http://cran.r-project.org/web/packages/snow/index.html.

Urbanek, S., 2009. *The multicore Package*, Available at: http://cran.r-project.org/web/packages/multicore/index.html.

Vance, A., 2009. Data Analysts Captivated by R's Power. *The New York Times*. Available at: http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?_r=3&ref=business.

S. Yoginath, N. F. Samatova, D. Bauer, G. Kora, G. Fann, and A. Geist, "RScaLAPACK: High Performance Parallel Statistical Computing with R and ScaLAPACK," in the *18th International Conference on Parallel and Distributed Computing Systems*. ISCA, 2005, pp. 61–67.

Yu, H., 2009. *The Rmpi Package*, Available at: http://cran.r-project.org/web/packages/Rmpi/index.html.