Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing

Xiaosong Ma[1,2], Jiangtian Li[1,2], and Nagiza F. Samatova[2]

[1]North Carolina State University
Department of Computer Engineering
Raleigh, NC 27695-8206 USA
ma@csc.ncsu.edu

[2]Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37831-6367 USA
samatovan@ornl.gov

Abstract

Desktop computing remains indispensable in scientific exploration, largely because it provides people with devices for human interaction and environments for interactive job execution. However, with today's rapidly growing data volume and task complexity, it is increasingly hard for individual workstations to meet the demands of interactive scientific data processing. The increasing cost of such interactive processing is hindering the productivity of end-to-end scientific computing workflows. While existing distributed computing systems allow people to aggregate desktop workstation resources for parallel computing, the burden of explicit parallel programming and parallel job execution often prohibits scientists to take advantage of such platforms. In this paper, we discuss the need for transparent desktop parallel computing in scientific data processing. As an initial step toward this goal, we present our on-going work on the automatic parallelization of the scripting language R, a popular tool for statistical computing. Our preliminary results suggest that a reasonable speedup can be achieved on real-world sequential R programs without requiring any code modification.

[1]

[1]This research was funded by an NSF CAREER Award, CNS-0546301 and Xiaosong Ma's joint appointment with Oak Ridge National Laboratory (ORNL). It was also partly supported by the DOE Scientific Data Management Center (http://sdmcenter.lbl.gov). ORNL is managed by the University of Tennessee-Battelle, L.L.C. for the Department of Energy under contract DOE-AC05-00OR22725.

1 Introduction

Powerful parallel computers are increasingly relied upon in scientific exploration, but these high-end facilities often lack means of direct interaction with scientists (such as through a display device). Further, these systems are shared resources normally located off-site and accessed through batch schedulers. Therefore, desktop computing will continue to be an indispensable part of high-performance computing workflows. By allowing direct interaction between end-users and their programs/data, personal computers form a window by which people interface daily with high-end computing/storage facilities. Even the heaviest supercomputer users perform part of their tasks, such as data visualization, on personal computers. However, with the growth speed of high-end supercomputers and instruments (*e.g.*, the aggregate FLOPS supplied by the world's No.1 supercomputer has increased by 800 times in the past 10 years), and consequently the rapid growth of data volumes generated by these systems [5, 8, 7], it is difficult for individual PCs to catch up even with the highest configuration. As a result, limited throughput in interactive (as opposed to batch-mode) desktop processing often compromises people's productivity in using powerful remote systems and slows down scientific exploration cycles.

Meanwhile, PCs have bursty workload since they are also used for daily tasks such as email and text processing, and their owners do not use them constantly. In fact, large portions of desktop computing/storage resources sit idle [1, 10]. Such idle resources, once aggregated, will significantly boost the performance and capacity of scientists' desktop processing. Most importantly, we are facing a historical opportunity for developing the next-generation desktop resource aggregation infrastructure: as the challenge increases for sustaining the speed growth predicted by Moore's Law due to power limitations, leading chip manu-

facturers such as Intel and AMD have directed their development efforts to multi-core processors. These processors bring unprecedented hardware parallelism to ordinary desktop machines. As exploiting parallelism in traditional personal applications may often be limited by the inherently sequential processing manner of the human brain, it is natural to explore aggregating idle hardware for running demanding scientific computing jobs as a secondary workload.

Harnessing idle resources on networked personal workstations for running parallel/distributed jobs is not a new idea. However, this computation mode has not become a main-stream option in scientific data processing. One of the key problems with existing systems is that *much effort has been focused on resource aggregation and management, with little emphasis on enabling easy parallel application development*. Scientists are often expected to explicitly parallelize their existing sequential applications, which is a very demanding and time-consuming task, usually requiring both careful programming and intensive debugging.

Even when scientists are willing to spend the time and effort parallelizing their codes, it is hard for the parallelized code to run in the networked desktop workstation context. Traditional parallel programming interfaces (such as MPI, PVM, and OpenMP) expect reliable and homogeneous nodes. In particular, as the de facto standard for parallel programming, MPI requires programmers to define explicit data distribution and inter-processor communication schemes, even when supported on heterogeneous, unreliable systems [6]. This requirement works well for simulations running on dedicated nodes in batch mode, but does not fit the dynamic and opportunistic desktop environments where scientists run their data analysis or visualization applications. While there exist flexible interfaces for master-worker style parallel computing through programming APIs or tools like DAGMan [4], they still require users explicitly compose the master-worker framework.

To provide desktop parallel computing as a general service to assist scientists in their daily data analysis, we need to find out how they can easily parallelize their crucial and time-consuming applications. Or better, we want to automatically parallelize such codes in a transparent and flexible way that is suitable for opportunistic parallel computing on unreliable idle nodes.

In this paper, we present our on-going work towards automatically parallelizing one type of common desktop scientific computing tasks: data processing using scripting languages such as Matlab and Python. More specifically, we are going to focus on R [11], a popular language and environment for statistical computing. Techniques developed in our work, however, may apply to other high-level scripting languages.

The rest of the paper describes pR, our automatic parallelization and execution framework for R.[2] pR borrows the parallelizing compiler technology to perform whole-program dependence analysis and couples it with runtime analysis as well as dynamic task scheduling. Our preliminary results show that pR produces good speedup without any modification to the sequential script. Although our currently prototype still runs in batch mode on a computer cluster, the pR design allows it to be extended to the desktop environment and interactive execution in a fairly straightforward way. Given the pervasive use of scripting language tools and the increasing amount of data to be processed, such a framework will be able to increase scientists' data analysis productivity without bringing them the hassle of traditional parallel computing.

2 The pR framework overview

2.1 R background

R [11] 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/.

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. Our current pR prototype targets batch mode execution, while our next step is to extend it to support interactive runs. 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.

2.2 Parallel R tools

A detailed discussion of related work in parallelizing high-level scripting languages can be found in our technical report [9]. Here we briefly describe several existing parallel R tools or environments.

[2] An expanded version discussing pR in more details is currently under submission for publication [9].

Libraries such as Rmpi [16] and rpvm [13] provide wrappers to popular parallel programming packages like MPI and PVM. Users of these libraries need to explicitly orchestrate the message passing in the parallel execution of their scripts. Another category of tools, including our own RScaLAPACK [14, 15], support transparent execution of standard R functions by using external numerical computation packages such as the ScaLAPACK library [2]. However, this type of tools cannot exploit task parallelism and are only suitable for closely coupled, homogeneous environments.

The snow package [12] 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.

2.3 pR design issues

Our key observation is that the use pattern of high-level scripting languages is significantly different from those of general-purpose compiled languages such as C/C++ or Fortran. Most R codes are composed of high-level prebuilt functions [3] typically written in a compiled language but made available to R environment through dynamically loadable libraries. On the other hand, 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.

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. To our best knowledge, pR is the only system that performs whole-program automatic parallelization of a scripting language (the state of the art in parallel scripting language was nicely summarized for Matlab [3] and the discussion also applies to R).

Here we highlight the two major innovations in the most recent 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.

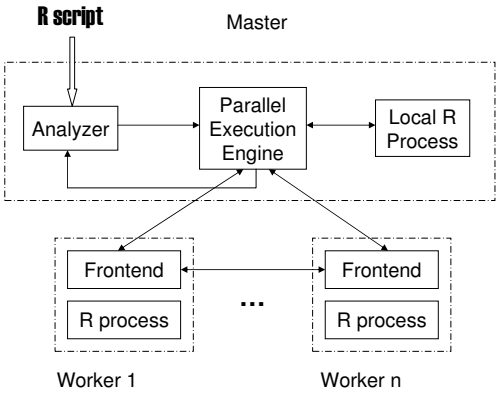

Figure 1. pR Architecture.

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.

2.4 pR 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 MPI 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 1, 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 1, 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 1.

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.

3 Preliminary results

In this section, we give part of our experimental results in evaluating pR. 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.

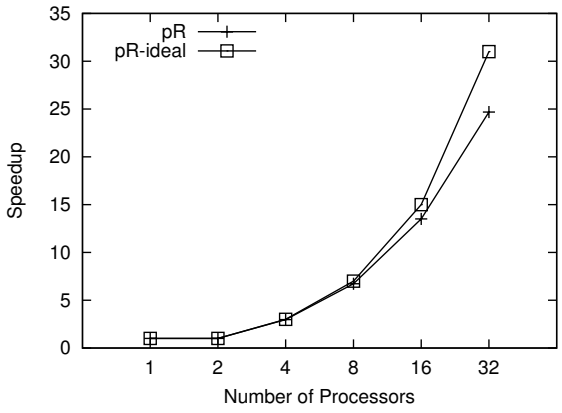

Figure 2. Performance of pR with the Boost application.

Considering the space limit, here we only present results from 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 2 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 increased 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.

	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.

Tables 1 lists the itemized overhead measured from the Boost 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. The data serialization stands for the process where the underlying R environment packs and unpacks R data objects into buffers.

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.

From Table 1, we see that the analysis and the master node scheduling overhead (from the MPI communication time at the master) are both quite small, even with 31 workers. 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 worker-side overhead heavily relies on how data-intensive an application is. With Boost, the total amount of data distributed across the workers is around 10MB, and the data serialization and message passing overhead may take as much as 3.5% of the total execution time. In particular, we have found that the R serialization procedure is significantly slower than the inter-processor communication.

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.

In our other experiments [9], we have compared the performance of pR with that of the snow package [12], which exploits embarrassingly parallel tasks and does require code modification. We used two synthetic test cases, one computation-intensive and the other both computation- and data-intensive. We have found that pR matches snow's performance (achieving a nearly linear speedup) with the first code, and significantly outperforms snow with the second one.

4 Discussion

With pR, we made the first step toward transparent desktop parallel computing. Based on our experience of implementing and evaluating pR, we are convinced that scripting languages are (1) suitable for parallelization due to the abundance of both task and data parallelism in user applications, and (2) ready candidates for runtime, automatic parallelization due to their limited syntax and the interpreted nature of their execution.

Although it is much more challenging to perform automatic parallelization on desktop data processing applications in general, we believe that this first step attacks a wide range of tools that are used extensively in many fields of science, especially for data analysis. With frameworks like pR, users can simply run their sequential code in a parallel environment, without seeing any details regarding task/data decomposition, inter-process communication and synchronization, and task scheduling.

However, as mentioned earlier, our current system targets batch-mode execution on a traditional multi-processor platform, while the ultimate goal of our project is to enable transparent, interactive parallel computing on distributed desktop machines. In the next phase of the project, we plan to approach this goal in two directions.

First, we plan to enable interactive execution with pR. Since pR can already perform incremental dependence analysis, conducting online parallelization while a user types in commands is not difficult. However, we must deal with the data placement/communication problem: the data access pattern is not known in advance. While transferring all intermediate results to the client machine may significantly increase data traffic, it can reduce the response time if the user directly requests the data. One possible strategy is to intelligently replicate data objects, so that the client can retrieve the intermediate or final output quickly, while the data locality of subsequent computation at the worker nodes can still be exploited.

The second step is more demanding: we will improve pR

to work on heterogeneous and unreliable desktop machines. This requires us to have more flexible parallelization and scheduling schemes, as well as appropriate fault-tolerance measures. For one, the current proof-of-concept implementation of pR has not taken into account issues such as load balancing or locality-aware scheduling. These issues will become very important when we move to the desktop environment, with large differences between individual machines' computation capacity, as well as much lower internode communication performance than on clusters. Finally, a large-capacity shared file system is often not available in a desktop environment and we have to deal with distributed I/O too.

5 Conclusion

In this paper, we gave the motivation for interactive, transparent parallel computing in an opportunistic desktop environment. We presented pR, an automatic runtime parallelization framework for the popular R scripting language, as our initial effort in bringing transparent parallel computing to assist scientists' in their increasingly demanding desktop data processing tasks.

With our preliminary results obtained through pR, we show that a reasonable speedup can be achieved on a real-world statistics application without any modification to the sequential source code. With extensions to handle interactive workloads as well as heterogeneous and unreliable machines, we expect to enable scientists to make use of idle workstation resources around them to run many data processing scripts without changing either the execution interface or the scripts themselves.

References

[1] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[2] 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.

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

[4] Directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman/.

[5] DOE Office of Science. *The Office of Science Data Management Challenge Report*, 2004.

[6] J. Dongarra, G. Fagg, A. Geist, J. Kohl, P. Papadopoulos, S. Scott, V. Sunderam, and M. Magliardi. HARNESS: Heterogeneous Adaptable Reconfigurable NEtworked SystemS. In *Proceedings of the The 7th IEEE International Symposium on High Performance Distributed Computing*, 1998.

[7] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, G. Heber, and D. DeWitt. Scientific data management in the coming decade. Technical Report MSR-TR-2005-10, Microsoft, 2005.

[8] J. Gray and A. Szalay. Scientific data federation. In I. Foster and C. Kesselman, editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, 2003.

[9] J. Li, X. Ma, S. Yoginath, G. Kora, and N. Samatova. Automatic, transparent runtime parallelization of the R scripting language. Technical Report TR-2007-3, North Carolina State University, 2007.

[10] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[11] 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.

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

[13] rpvm: R interface to PVM (Parallel Virtual Machine). http://cran.r-project.org/src/contrib/Descriptions/rpvm.html.

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

[15] 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.

[16] H. Yu. Rmpi package for R. http://www.stats.uwo.ca/faculty/yu/Rmpi/.