

ABSTRACT

LI, JIANGTIAN. Towards Transparent Parallel Processing on Multi-core Computers.
(Under the direction of Professor Xiaosong Ma).

Parallelization of all application types is critical with the trend towards an exponentially increasing number of cores per chip, which needs to be done at multiple levels to address the unique challenges and exploit the new opportunities brought by new architecture advances. In this dissertation, we focus on enhancing the utilization of future-generation, many-core personal computers for high performance and energy effective computing.

On one hand, computation- and/or data-intensive tasks such as scientific data processing and visualization, which are typically performed sequentially on personal workstations, need to be parallelized to take advantage of the increasing hardware parallelism. Explicit parallel programming, however, is labor-intensive and requires sophisticated performance tuning for individual platforms and operating systems. In this PhD study, we made a first step toward transparent parallelization for data processing codes, by developing automatic parallelization tools for scripting languages. More specifically, we present pR, a framework that transparently parallelizes the R language for high-performance statistical computing. We apply parallelizing compiler technology to runtime, whole-program dependence analysis and employ incremental code analysis assisted with evaluation results. Experimental results demonstrate that pR can exploit both task and data parallelism transparently and overall achieve good performance as well as scalability. Further, we attack the performance tuning problem for transparent parallel execution, by proposing and designing a novel online task decomposition and scheduling approach for transparent parallel computing. This approach collects runtime task cost information transparently and performs online static scheduling, utilizing cost estimates generated by ANN-based runtime performance prediction, as well as by loop iteration test runs. We implement the above techniques in the pR framework and our proposed approach is demonstrated to significantly improve task partitioning and scheduling over a variety of benchmarks.

On the other hand, multi-core personal computers will inevitably be under-utilized when their owners perform light-weight tasks such as editing and web browsing, making volunteer computing more appealing than ever. In this study, we made a first step towards a novel computation model, energy-aware volunteer computing on multi-core processors, by

evaluating the potential energy/performance trade-off of a more aggressive execution model that selects active nodes over idle nodes for scheduling foreign application tasks, in order to better utilize idle cores and achieve energy savings. Our experiment results suggest that aggressive volunteer computing can bring significant energy saving compared to common existing execution modes and provides an attractive computation model.

Towards Transparent Parallel Processing on Multi-core Computers

by
Jiangtian Li

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Frank Mueller

Dr. Nagiza Samatova

Dr. Xiaosong Ma
Chair of Advisory Committee

Dr. Xiaohui Gu

DEDICATION

To my parents *Guangyi Li, Jianfen Huang*
and
my wife *Liang Tang*

BIOGRAPHY

Jiangtian Li was born in Huaian, a small town in east China in 1978. He started his undergraduate study in the Special Class for the Gifted Young at the University of Science and Technology of China, where he received his B.S. degree in Chemical Physics in 1998. He later obtained his M.S. degree in Industrial Engineering from Purdue University, majoring in Operation Research in 2003. In spring 2005, he started his Ph.D. study in Computer Science at North Carolina State University. His research interests are in parallel systems and algorithms. He will join Microsoft upon graduation.

ACKNOWLEDGMENTS

This dissertation would not have been accomplished without the support and inspiration of many people.

First of all, I would like to express my deep gratitude to my advisor, Dr. Xiaosong Ma, for her expert guidance, consistent support, and patience to shepherd me in my Ph.D. pursuit. I feel very fortunate to have such a great mentor leading me through my research with insightful thoughts and sound advice. I am also much indebted to Dr. Nagiza Samatova for mentoring me and initializing the pR project, particularly during my internship and visits at Oak Ridge National Laboratory. Her knowledge and suggestions have been of great value to me. Also I would like to convey my appreciation to Dr. Xiaohui Gu and Dr. Frank Mueller for serving on my advisory committee and offering me valuable suggestions on my dissertation.

I am grateful to the ORNL team in the development of the pR prototype: Srikanth Yoginath and Guruprasad Kora for important contributions in code, thoughts, and discussions. Besides, they provided great convenience during my stay at Oak Ridge. Also I am grateful to our collaborators at LLNL and Cornell University: Karan Singh, Dr. Martin Schulz, Dr. Bronis de Supinski, and Dr. Sally McKee for valuable suggestions, software support and assistance in paper writing.

I want to thank all the members in PALM group for their friendship and the fun time spent together in the lab. Thanks to Heshan Lin for his generous help since I came to NCSU. I benefit a lot from brainstorming research ideas and discussing technical problems with him. Thanks to Zhe Zhang, Tao Yang, and Feng Ji for their unselfish help in research, courses, and in life.

My sincere thanks also go to Mihye Ahn, Tyler Bletsch, Kristy Boyer, Jaydeep Marathe, Ripal Nathuji, Christopher Symons, Vivek Thakkar, Chao Wang for their warm-hearted help during my doctoral research.

I gratefully acknowledge the funding sources of this dissertation work: DOE ECPI Award (DE-FG02-05ER25685), NSF CAREER Award (CNS-0546301), the DOE SciDAC SDM center, and the joint appointment between NCSU and ORNL for Xiaosong Ma. I also sincerely thank the following for providing computing facilities: ORNL, High Performance Center and System Research Lab at NCSU.

Finally, I would like to thank my parents, my wife and my brother for their unwavering support, encouragement, and love.

TABLE OF CONTENTS

LIST OF TABLES.....	viii
LIST OF FIGURES	ix
1 Introduction.....	1
1.1 Problem Statement and Motivation	1
1.2 Contributions	3
1.3 Dissertation Outline	5
2 Transparent Parallelization of the R Scripting Language	6
2.1 Motivation for Transparent Parallel Data Processing	6
2.2 Background information of R	8
2.3 pR Architecture	9
2.3.1 Design Rationale	9
2.3.2 Framework Architecture	10
2.4 pR Design and Implementation	12
2.4.1 The Analyzer	12
2.4.2 The Parallel Execution Engine	17
2.5 Ease of Use Demonstration	18
2.6 Performance Evaluation	20
2.6.1 Synthetic Loop	20
2.6.2 Boost	21
2.6.3 Sensor Network Code	22
2.6.4 Bootstrap	23
2.6.5 SVD	24
2.6.6 Overhead Analysis	24
2.6.7 Task Parallelism Test	27
2.7 Summary	28
3 Autonomic Task Scheduling.....	30
3.1 Motivation	30
3.2 Background on Artificial Neural Networks	33
3.3 ASpR System Architecture	34
3.4 Function Performance Profiling and Model Construction	36
3.5 Loop Cost Prediction through Test Driving	37
3.6 Task Partitioning and Scheduling	38
3.7 Experimental Results	42
3.7.1 Sample Schedule	42
3.7.2 Comparison with Dynamic Scheduling	43

3.7.3	Benchmark Generation	45
3.7.4	Accuracy and Overhead of Online Prediction	47
3.7.5	Impact on Transparently Parallelized R Code Execution	49
3.7.6	Real-world Application Performance Results	53
3.8	Summary	54
4	Energy and Performance Impact Analysis	56
4.1	Motivation	57
4.2	Workloads	58
4.2.1	Native Workloads	58
4.2.2	Foreign Workloads	61
4.3	Methodology	62
4.3.1	Objectives and Metrics	62
4.3.2	Experimental Platform	63
4.3.3	Measurement Scheme	64
4.4	Results	66
4.4.1	Impact of Concurrency in the Cluster Mode	66
4.4.2	Consolidated Execution in Native OS	67
4.4.3	Consolidated Execution with Virtual Machine	71
4.4.4	Throttling with Vulnerable Workload	72
4.5	Summary	73
5	Related Work	80
5.1	Transparent Parallelization	80
5.2	Performance Prediction and Self-learning Systems	83
5.3	Resource Allocation and Task Scheduling	84
5.4	Energy-aware Computing	85
5.5	Co-scheduling and Performance Impact	87
6	Conclusions and Future Work	88
	Bibliography	91

LIST OF TABLES

Table 2.1 Itemized overhead with the Boost code, in percentage of the total execution time. The sequential execution time of Boost is 2070.7 seconds.	25
Table 2.2 Itemized overhead with the sensor network code, in percentage of the total execution time. The sequential execution time is 825.2 seconds.	25
Table 2.3 Itemized overhead with the bootstrap code, in percentage of the total execution time. The sequential execution time of bootstrap is 2918.2 seconds.	25
Table 2.4 Itemized overhead with the SVD code, in percentage of the total execution time. The sequential execution time of SVD is 227.1 seconds.....	26
Table 2.5 The parallel execution time and speedup of the task parallelism test script ..	28
Table 3.1 MCP overhead in seconds measured in our testbed.....	44
Table 3.2 Description of selected R functions and their arguments in the experiments. A and B denote $n \times n$ numeric matrices.....	45
Table 3.3 Overview of parameter spaces of selected R functions used in the experiments. N_A and N_B denote their sizes in terms of the number of double-precision numbers.	46
Table 3.4 Performance summary of ASpR on the 100 automatically generated microbenchmarks.....	49
Table 3.5 ASpR and dependent microbenchmarks (8 procs)	51
Table 3.6 Overview of tasks in the real R application.....	53
Table 4.1 Benchmarks used as native workloads.....	59
Table 4.2 Benchmarks used as foreign workloads	59
Table 4.3 L2 cache miss ratios of native workloads	70
Table 4.4 Average CPU utilization and power consumption of native workloads, running alone or with foreign workloads	70
Table 4.5 Performance degradation of foreign workloads running within VMware	71

LIST OF FIGURES

Figure 2.1 A sample R script	9
Figure 2.2 pR Architecture	11
Figure 2.3 The pR analyzer's internal structure	12
Figure 2.4 A sample parse tree and 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.	13
Figure 2.5 Comparison with the snow package interface	19
Figure 2.6 Performance of pR with various benchmarks. Error bars are omitted since variances are small. The average 95% confidence interval (CI) of pR is 0.98% and maximum 95% CI is 2.86%. (a) Performance with the synthetic loop code. (b) Performance with the Boost application.	20
Figure 2.7 Performance of pR with various benchmarks. Error bars are omitted since variances are small. The average 95% confidence interval (CI) of pR is 0.98% and maximum 95% CI is 2.86%. The average 95% CI of snow is 1.44% and maximum 95% CI is 5.63%. (a) Performance with the sensor network code. (b) Performance with the bootstrap code. (c) Performance with the SVD code.	22
Figure 2.8 The task parallelism test code	27
Figure 3.1 A sample R script	31
Figure 3.2 A motivating example with sample schedules that show task lengths proportional to their execution time on an eight core system	31
Figure 3.3 A sample feed-forward neural network	33
Figure 3.4 A sample hidden layer unit with a sigmoid activation function (borrowed from [81])	34
Figure 3.5 Adaptively Scheduled parallel R (ASpR) architecture overview	35
Figure 3.6 Schedules for the sample code shown in Figure 3.2	42

Figure 3.7 Comparison with dynamic scheduling	43
Figure 3.8 Online training accuracy and overhead for various training data sizes	48
Figure 3.9 Average normalized improvement for “top 10” microbenchmarks	50
Figure 3.10 Comparison of scheduling approaches on “top 10” microbenchmarks	52
Figure 3.11 Comparison of scheduling approaches on the real R application	54
Figure 4.1 Power states of Thinkpad t61	58
Figure 4.2 Power Experimental Setup	63
Figure 4.3 Measurement scheme overview	65
Figure 4.4 Performance/energy efficiency of foreign workloads on 8-core server with varied level of concurrency	67
Figure 4.5 Energy saving of consolidated execution in the native OS. The pair of numbers above each group of bars show ESR^C/ESR^V for that workload combination.	75
Figure 4.6 Performance impact when workloads consolidated in the native OS	76
Figure 4.7 Energy saving of consolidated execution in VM. The pair of numbers above each group of bars show ESR^C/ESR^V for that workload combination.	77
Figure 4.8 Performance impact with foreign workloads running in VM	78
Figure 4.9 Effect of foreign process throttling with selected workload combinations ...	79

Chapter 1

Introduction

1.1 Problem Statement and Motivation

General-purpose computing is taking an irreversible step toward parallel architectures, as stated in a recent report on the landscape of parallel computing research [4] by researchers from Berkeley. The number of cores on a chip is expected to double with each semiconductor process generation. As we enter this multi-core era, we are facing unprecedented challenges and opportunities for scientific computing and data analytics.

On one hand, multi-core computers will greatly benefit scientific data analytics in particular, as it is traditionally performed in personal computing settings. The increasing number of cores provides more computing power and degree of parallelism at the thread level. This brings good news to scientific data analytics, which 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 time-steps, data partitions, etc.

Moreover, the single-computer parallel data processing can naturally be extended to use multi-computer environments, through mechanisms such as volunteer computing, as a result of relatively loosely coupled relationship between work units. In this regard, projects such as BOINC [19] and CONDOR [73] have been deployed for parallel execution of data processing tasks [31, 37, 63, 89, 115]. The high-impact BOINC project consists of independent work units and is naturally parallel.

On the other hand, scientific data analytics currently lacks the proper statistical

tools or libraries that could efficiently exploit this parallelism. Unlike simulations that have utilized supercomputers for decades, 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. Further, parallel computing techniques used predominantly on clusters often are not applicable or usable for parallel data analysis.

In particular, several concerns arise with respect to the ease-of-use and energy-efficiency of parallel data analytics, which potentially hurdle the effective utilization of the underlying architecture parallelism.

First, the complexity of parallel programming is well known and has been documented in the literature [78, 91]. One widely adopted parallel programming model is message passing [49]. However, this model requires that programmers have parallel programming skills to specify ways of data partitioning, task scheduling and inter-process synchronization, which are challenging, time-consuming, and platform-dependent tasks. Most data processing tool developers are domain scientists, who prefer spending time studying their data than developing parallel codes and dealing with the porting/execution overheads associated with explicit parallel programming. One alternative to explicitly parallel programming is to let the compiler perform parallelization work and produce binaries that can be executed in parallel [1, 5, 7, 18, 54]. The research has been conducted for programming languages such as C and Fortran. However, to the best of our knowledge, little work has been done in automatic parallelization for interpreted languages, which are used in popular data processing tools such as Matlab [77] and R [96].

Even with compiler technology transplanted to the scripting language problem, transparent parallel data processing tools must be able to exploit both *task parallelism* and *data parallelism* (or *loop parallelism*), which was not an emphasis of parallelizing compilers. To give a concrete example, interactive scientific data navigation and analysis is frequently performed sequentially on desktop computers despite possessing significant task and data parallelism. Common task parallelization opportunities include the computation of the eigenvalues of a matrix while also generating a histogram of its values. Data parallelization opportunities for scripting languages, such as a loop that processes the output files from a series of time-steps or the elements of a large array, are even more common. To achieve flexible and scalable parallelization, the data processing software need to automatically

identify both kinds of parallelism and coordinate resources wisely to achieve a better overall performance.

Finally, while performance may remain the dominating goal for scientific computing and data analytics, the energy cost has become an increasing concern. High-end server systems and clusters have traditionally been used to deliver higher performance in scientific computing. However it comes at the energy cost of operating and cooling of server systems. With more powerful multi-core personal computers, offloading scientific workloads to personal computers and utilizing their idle cores provides an alternative that can potentially save energy. On the other hand, though the multi-core architecture naturally provides more room for the concurrent execution of multiple workloads, contention for resources, especially non-CPU ones, can still lead to performance degradation visible to resource owners. How to achieve energy-performance balance for volunteer scientific computing emerges as an urgent question to be answered.

1.2 Contributions

In this dissertation study, we have taken several initial steps towards transparent parallel processing on multi-core computers. Specifically, we addressed the problems outlined above by examining the ease-of-use and energy-effective side of parallel data processing on multi-core computers.

Regarding user-friendly parallel scientific data processing, *we explored transparent parallelization and task scheduling for scripting languages*. Major contributions along this direction include:

- We designed and implemented a transparent parallelization framework, pR, for the R scripting language. We apply parallelizing compiler technology to runtime, whole-program dependence analysis and propose incremental code analysis. 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 evaluate 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.

- We propose a fully transparent self-adaptive, platform- and application-independent loop and task scheduling mechanism, in an effort to optimize pR. We build lightweight online profiling and performance prediction models based on machine learning. Moreover, we extend the current static scheduling algorithm to use cost estimates from our models for loop partitioning and task scheduling, and runtime heuristics for more effective processor allocation and scheduling. Our results using both real-world applications and synthetic benchmarks show that our proposed approach significantly improves task partitioning and scheduling, with a maximum improvements of 40.3% and average improvement of 12.3%.
- To our best knowledge, pR is the first transparent parallelizing tool for a scripting language without using special hardware or virtual machine support. Also, we are the first to apply machine learning technology such as Artificial Neural Networks (ANN) to parallel task scheduling. Meanwhile, though our work used R for a proof-of-concept investigation and case study, the techniques developed in this study can be applied in more general settings, such as the transparently parallel execution of other scripting languages, and performance profiling based self-learning systems.

With tools like pR emerging for the next-generation multicore computers, it is more appealing than ever to leverage mechanisms such as volunteer computing, to exploit under-utilized personal computers for high-throughput, computation- and/or data-intensive data processing. To this end, we examined a new computing model, *aggressive volunteer computing*, which distributes resource-intensive background workloads to active PCs rather than waiting for them to quiet down as done in traditionally volunteer computing.

More specifically, in this work we estimate the efficacy of aggressive volunteer computing, by evaluating the energy saving and performance impact of co-executing resource-intensive foreign workloads with native personal computing tasks. Our experiments examine the pairwise combination of 5 foreign and 6 native workload benchmarks. Our results suggest that aggressive volunteer computing can achieve an average energy saving of around 50% compared to running the foreign workloads on high-end cluster nodes, and around 30% compared to using the traditional, more conservative volunteer computing model. We have also observed highly varied performance interference behavior between the workloads, and evaluated the effectiveness of foreign workload intensity throttling.

1.3 Dissertation Outline

The remaining chapters are organized as follows: In an effort to address the issues in transparent parallel processing on multi-core computers, Chapter 2 introduces pR, a transparent runtime parallelization framework for the R scripting language. The background knowledge of R is described at the beginning of the Chapter. Chapter 3 describes several optimization techniques on top of pR to improve task scheduling. Chapter 4 provides evaluation of energy efficiency and performance impact in aggressive volunteer computing. Chapter 5 gives a comprehensive review on the related work. Chapter 6 summarizes our research and discusses future work.

Chapter 2

Transparent Parallelization of the R Scripting Language

In this chapter, we present our first step toward transparent parallel processing on multi-core computers. Given the pressing need for transparent parallelization in scientific data processing nowadays, as we will discuss in details in section 2.1, we tackle the problem by experimenting on the R scripting language, a widely used data processing tool in statistical computing. Section 2.3 and 2.4 present our transparent parallelization framework for R and design and implementation issues, followed by ease-of-use demonstration in section 2.5 and performance evaluation in section 2.6.

The pR framework builds the groundwork of our endeavor in transparent parallelization for scientific data analytics. It includes fundamental functionalities to handle dependence analysis, task and data distribution, synchronization, and orchestration. Based on this framework, we will further explore possible optimizations in the next Chapter.

2.1 Motivation for Transparent Parallel Data Processing

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

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 [77] and R [96] (Section 5.1 gives the detailed description of related work). 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/libraries. First, there lacks support for transparent parallelization that would enable domain scientists to reuse 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 painful exercise that scientists will often resist to. Hence, for domain scientists, 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 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 an automatic and transparent runtime parallelization framework for scripting languages. We argue that data dependence analysis techniques developed for compiled languages are more than adequate in parallelizing loops and function calls in data processing scripts. Meanwhile, the interpreted nature of scripting languages provides a very favorable environment for runtime, 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 back-

ground information about R is given in the next section.

2.2 Background information of R

R [96] 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. The R source codes were written mainly in C. R can be used on various platforms such as Linux, Macintosh and Windows and can be downloaded from the CRAN (Comprehensive R Archive Network) site at <http://www.r-project.org/>.

As a data processing tool, R can perform diverse statistical analysis tasks such as linear regression, classical statistical tests, time-series analysis, and clustering. It also provides a variety of graphical functions such as histograms, pie charts and 3D surface plots. Built upon R, many other data processing software tools are developed, such as BioConductor [14], an open-source, open-development software package for the analysis of biological data.

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

R can be used in both interactive and batch modes. In the interactive mode, R works via a command-line interface. The user retrieves the results from the standard output by typing variable names. In the batch mode, an R script is supplied as a file and is executed from the R prompt. Results can be written into output files or retrieved from the standard output as in the interactive mode. In our research we focus on parallelization of the batch-mode execution.

In Figure 2.1, we present a running example of an R script that will be used to illustrate how pR works. It is designed more to cover important pR issues than to perform a meaningful job. Line 1 and line 2 are simple constants assignment. Line 3 generates a vector of nine normally distributed real numbers. Line 4 initializes a 2-D array. Line 5 is a loop performing a simple arithmetic operation. Lines 6-8 read the matrix data from files.

```

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

```

Figure 2.1: A sample R script

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.

2.3 pR Architecture

2.3.1 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 [29] 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. There are no explicit data pointers.¹ The input parameters to the functions are read-only, while the modified or newly-created variables by the function are returned explicitly. Returning several such variables is accomplished through the formation of a list object, which is a collection of other objects. 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 run time 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.

2.3.2 Framework Architecture

The key feature of pR is that it dynamically and transparently analyzes a sequential R source script and accordingly parallelizes its execution. The results of partial execution are collected to perform further analysis at run time. The pR 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 [49], 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

¹R does provide several types of reference objects, whose handling is described in Section 2.4.1.

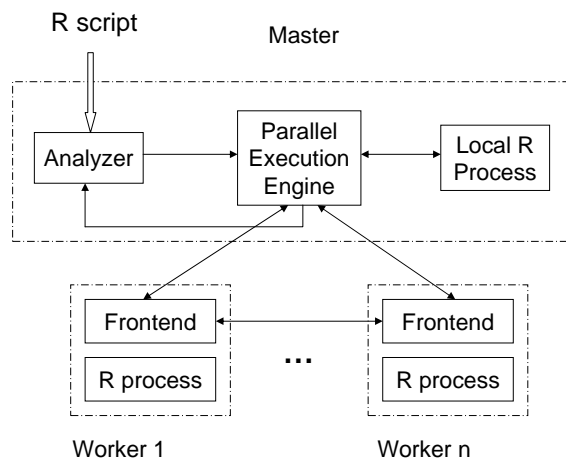


Figure 2.2: pR Architecture

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 Figure 3.5, 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 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. 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 3.5.

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.

2.4 pR Design and Implementation

2.4.1 The Analyzer

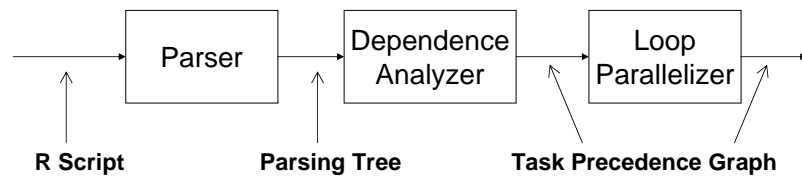


Figure 2.3: The pR analyzer’s internal structure

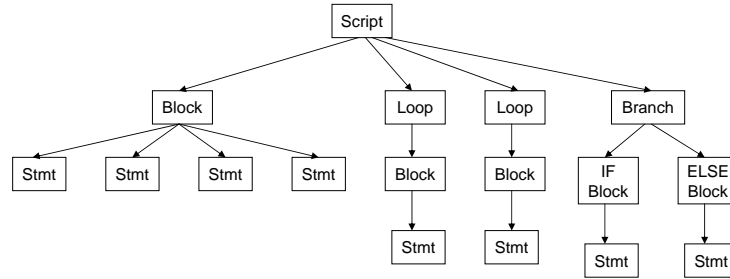
As illustrated in Figure 2.3, there are three basic modules inside the analyzer: a *parser*, a *dependence analyzer*, and a *loop parallelizer*. Below we discuss the task performed by each of these modules.

Parsing 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*.

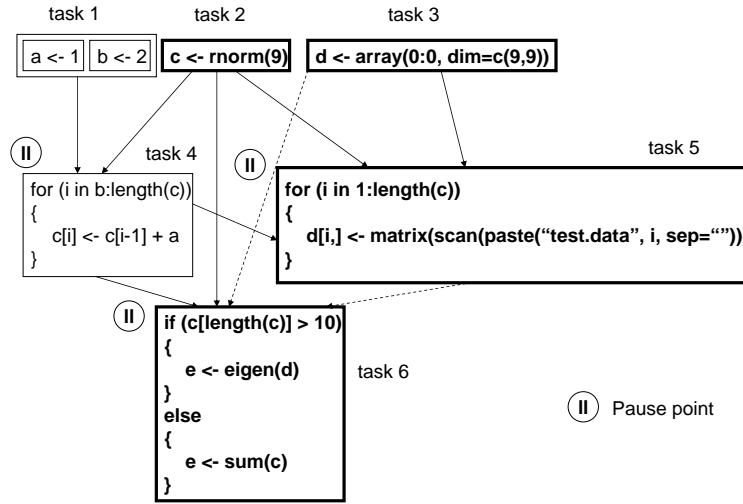
The parse tree generated by the pR parser is a simplified version compared to those by compilers. Here the sole purpose of parsing is to perform dependence analysis and automatic parallelization, while the actual interpretation and evaluation of R statements are carried out by the native R environment. 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.



(a) A sample parse tree.



(b) A sample task precedence graph.

Figure 2.4: A sample parse tree and 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.

Figure 2.4(a) shows the generated parse tree for the sample code given in Figure

2.1. The leaf nodes are all statements corresponding to lines 1-4, 7, 11, 15, and 19, respectively. Each internal node covers a loop, a branch, or another region in the code, where a “block” stands for a code block that does not contain loops or branches.

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 re-

quire 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 [79], many advanced dependence modeling and loop transformation techniques have been proposed and used in compilers [1, 6, 18, 54, 94, 100, 117].

As mentioned in Section 2.3, 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* [8], 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 a *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* (TPG), as the output of the dependence analyzer. A TPG is essentially a directed acyclic graph, where each vertex represents one task and each directed edge represents the dependence between two tasks. In addition, if the task consists of a loop, the dependence distance (defined as the difference between dependent iteration numbers) for each loop index will also be recorded. This information is not used in the current implementation, but will be useful in future extensions performing loop transformation to parallelize loops where limited inter-iteration dependence exists.

Figure 2.4(b) shows the task precedence graph generated from the parse tree in

Figure 2.4(a). 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.

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 [5] 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.

2.4.2 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. Further work on improving scheduling is presented in Chapter 3.

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 Figure 2.4(b). 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 .

2.5 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 [101], which allows users to parallelize embarrassingly parallel operations. Figure 2.5 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`.

<pre> ... u<-TRUE la<-list() for (i in 1:length(l)) la[[i]]<-attributes(l[[i]]) o<-la[[1]] if (o\$type=="factor") { o<-o[names(o)!="levels"] } o<-o[names(o)!="type"] o<-o[names(o)!="dimnames"] ... </pre>	<pre> <i>library(Rmpi)</i> <i>library(snow)</i> ... <i>cl <- makeCluster(4, type = "MPI")</i> u<-TRUE <i>la<-clusterApply(cl,l,attributes)</i> o<-la[[1]] if (o\$type=="factor") { o<-o[names(o)!="levels"] } o<-o[names(o)!="type"] o<-o[names(o)!="dimnames"] <i>stopCluster(cl)</i> ... </pre>
--	--

Figure 2.5: Comparison with the snow package interface

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 `makeCluster`. The user then executes the function `attributes` on the elements of the target list `l` in parallel on this cluster using `clusterApply`. In this case the results will be stored as the elements of the list `la`. Finally the user has to remove the cluster by calling `stopCluster`. In addition, snow has inconvenient restrictions with parallel interfaces like `clusterApply`. For example, here the list `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 `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.

2.6 Performance Evaluation

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

We performed each test multiple times and observed that the performance variance was very small (with standard deviations less than 6%), so error bars were omitted.

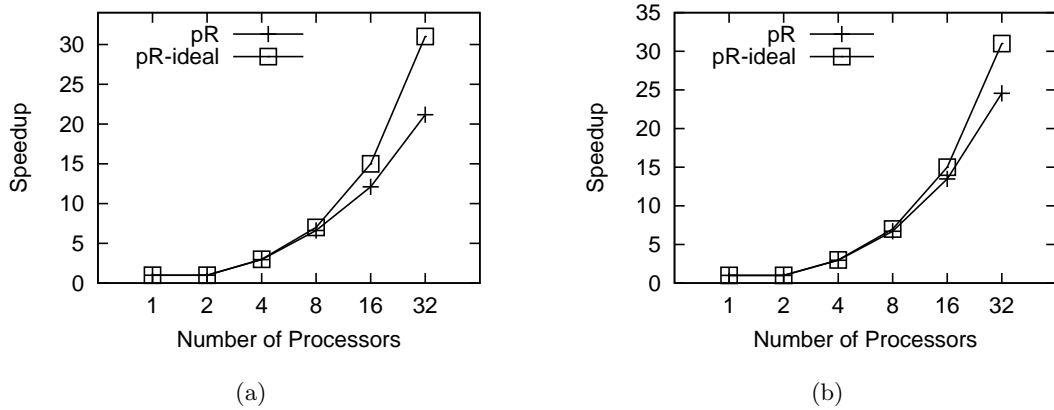


Figure 2.6: Performance of pR with various benchmarks. Error bars are omitted since variances are small. The average 95% confidence interval (CI) of pR is 0.98% and maximum 95% CI is 2.86%. (a) Performance with the synthetic loop code. (b) Performance with the Boost application.

2.6.1 Synthetic Loop

To further demonstrate the ease-to-use feature of pR, we made 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.

Figure 2.6(a) 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.

2.6.2 Boost

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. 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 2.6(b) 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 2.1 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.

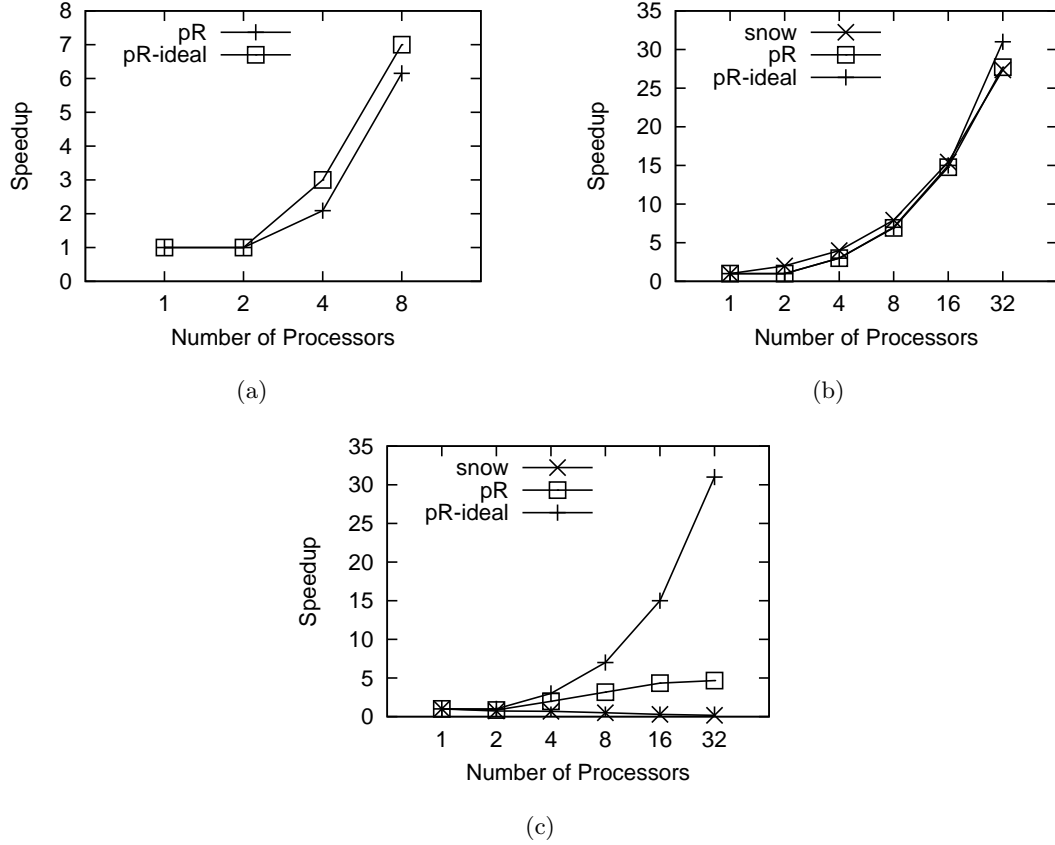


Figure 2.7: Performance of pR with various benchmarks. Error bars are omitted since variances are small. The average 95% confidence interval (CI) of pR is 0.98% and maximum 95% CI is 2.86%. The average 95% CI of snow is 1.44% and maximum 95% CI is 5.63%. (a) Performance with the sensor network code. (b) Performance with the bootstrap code. (c) Performance with the SVD code.

2.6.3 Sensor Network Code

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 (when a reading is not sent on purpose because it hasn't changed). Normally, it is impossible to distinguish a failure from suppression. Low-cost redundancy is added to transmitted messages to help identify when failures and suppressions have occurred, and that information is used to constrain Bayesian inference of the missing values [109]. 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 iteration performs statistical analysis on different input data and calls several computation-intensive functions. The loop is embarrassingly parallel in nature and therefore ideal for parallelization.

Figure 2.7(a) 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 two factors. First, the independent iterations are not able to be evenly distributed to the workers. Second, the execution time of each iteration varies from 32 to 138 seconds depending on input data.

2.6.4 Bootstrap

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 [116]. It performs bootstrap in a loop using the R boot function and the nuclear data provided in R. This forms an ideal case for parallelization, as it is computation-intensive but not data-intensive. We created the corresponding sequential code using a for loop. Figure 2.7(b) 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.

2.6.5 SVD

The second synthetic test case 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.

Figure 2.7(c) 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 behavior is consistent with what the snow authors reported with communication-intensive codes [101].

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.

2.6.6 Overhead Analysis

Tables 2.1-2.4 list the itemized overhead measured from pR tests, in the percentage of the total execution time. E.g., “0.05” in a cell means 0.05% of the total execution time

Table 2.1: Itemized overhead with the Boost code, in percentage of the total execution time. The sequential execution time of Boost is 2070.7 seconds.

	2	4	8	16	32
Initialization	0.05	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 2.2: Itemized overhead with the sensor network code, in percentage of the total execution time. The sequential execution time is 825.2 seconds.

	2	4	8
Initialization	0.09	0.23	0.52
Analysis	0.00	0.01	0.01
Master MPI	0.00	0.00	0.00
Max wkr. serial.	0.00	0.00	0.01
Max wkr MPI	0.00	0.00	0.00
Max wkr socket	0.00	0.00	0.00

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

Table 2.3: Itemized overhead with the bootstrap code, in percentage of the total execution time. The sequential execution time of bootstrap is 2918.2 seconds.

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

Table 2.4: Itemized overhead with the SVD code, in percentage of the total execution time. The sequential execution time of SVD is 227.1 seconds.

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

(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 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 Figure 2.7(c).

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.

2.6.7 Task Parallelism Test

```
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")
```

Figure 2.8: The task parallelism test code

Finally, we use another synthetic test case to test pR's capability of parallelizing non-loop tasks. Figure 2.8 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.

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 2.5 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

Table 2.5: The parallel execution time and speedup of the task parallelism test script

	1	2	4	8
Exec time	90.37	122.26	49.37	35.82
Speedup	1	0.74	1.83	2.52

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.

2.7 Summary

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 propose 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.

Chapter 3

Autonomic Task Scheduling

In Chapter 2 we have presented and built a transparent framework for parallel data processing. However, one key issue we have not sufficiently addressed in this transparent framework is task scheduling. There have been numerous studies of task scheduling in parallel processing, applying to various scenarios and constraints. Data processing tools written in scripting languages such as R often exhibit multi-level of parallelism, which usually remain unexploited. This brings new opportunity for task scheduling.

In this chapter, we discuss optimization techniques with respect to task partitioning and scheduling. We first illustrate the problem and the opportunity using an example R script. In the rest of this chapter, we present several techniques developed in this thesis study, including using Artificial Neural Networks for runtime task cost prediction and extending an existing static scheduling algorithm. We then discuss our evaluation methods and present experiment results.

3.1 Motivation

In this chapter we specifically explore techniques that can exploit both types of parallelism simultaneously. Figure 3.1 shows a target scenario, a data processing script written in R [96], a popular statistical computing package. Lines 1-3 initialize a list and two matrices. The tasks, T1 and T2, on lines 4 and 5 perform a Friedman rank sum and a sample Wilcoxon test. The loop L in lines 6-8 performs vector-matrix multiplication and calculates mean values. Since L, T1, and T2 are mutually independent, they can

```

a <- array(rnorm(1200*1200), dim=c(1200, 1200))      1
b <- list()                                           2
c <- array(rnorm(1200*1200), dim=c(1200, 1200))      3
T1: f <- friedman.test(a)                             4
T2: w <- wilcox.test(c)                               5
L: for (i in 1 : 1200) {                             6
    b[[ i ]] = mean( c %% a[i, ] )                   7
  }                                                  8

```

Figure 3.1: A sample R script

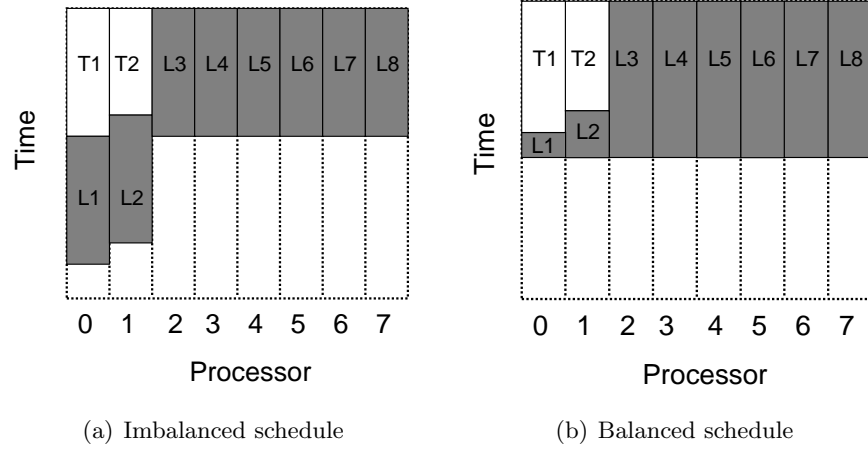


Figure 3.2: A motivating example with sample schedules that show task lengths proportional to their execution time on an eight core system

be executed in parallel. Further, there are no dependences between the iterations of L so they can also be parallelized. However, a straightforward loop decomposition and task distribution will not produce an efficient parallel execution schedule, as demonstrated by the example schedules that use lengths proportional to the execution times for L , $T1$ and $T2$ on an eight core system. Figure 3.2(a) shows a naive schedule that divides L iterations evenly by the number of processors. Since the functions and the loop blocks execute whenever a processor is available, two loop blocks use the same processors as $T1$ and $T2$. Thus, this schedule leaves six processors idle while those loop blocks complete. In contrast, Figure 3.2(b) shows a much more desirable schedule that optimizes the loop decomposition in light of the tasks to provide perfect load balance, but requires an accurate prediction of all tasks' execution times before scheduling L .

In fact, we could apply many existing static task scheduling techniques [67] to our target scenario – if we knew the cost of the tasks and loop iterations in advance. Since

our goal is an implicit parallelization mechanism, we must use transparent performance models that predict those costs. We achieve this by observing the performance of application components and then using the collected information to predict later executions of those components. This approach particularly suits scripting languages since they typically have a small set of functions (often with relatively stable parameter ranges) that are used repeatedly both within a single job and across jobs, whether belonging to the same user or not. Thus, we can gather enough samples, potentially from multiple jobs or even from multiple users in a shared performance data repository, for accurate predictions.

We present our novel runtime parallelization tool that performs intelligent loop decomposition and task scheduling, based on performance models derived from past performance data collected on the local platform. We use artificial neural networks (ANNs), which are application-independent and self-learning black box models, as required for implicit parallelization. We integrate these techniques into the existing pR framework [74], which performs runtime, automatic parallelization of R scripts. The result is an adaptive scheduling framework for the parallel execution of R, which we call ASpR (Adaptively Scheduled parallel R).

For the example in Figure 3.1, ASpR now estimates the costs of T1 and T2 based on past calls to those functions. Further, it can infer the cost of the individual iterations of L based on the early feedback collected by “test driving” the same loop. We then use these cost estimates as inputs for our scheduling algorithm, which is an extension of MCP scheduling algorithm [121] to generate an informed loop partitioning and scheduling plan that is close to the one shown in Figure 3.2(b).

ASpR employs Artificial Neural Networks (ANNs) for task performance prediction. These networks have been previously utilized in performance modeling of parallel applications [69]. Compared to statistical techniques such as clustering, correlation analysis, and piecewise polynomial regression, ANNs are fully automated and require no domain-specific knowledge. Therefore they are well suited to be incorporated into our pR framework and automatically assist its task scheduling. In the next section, we will give a brief background discussion on ANNs.

3.2 Background on Artificial Neural Networks

Artificial Neural Networks (ANNs) are machine learning models based on biological neural networks. ANNs have been designed as a powerful and automatic method for solving a variety of problems including pattern recognition, clustering, prediction and forecasting [60].

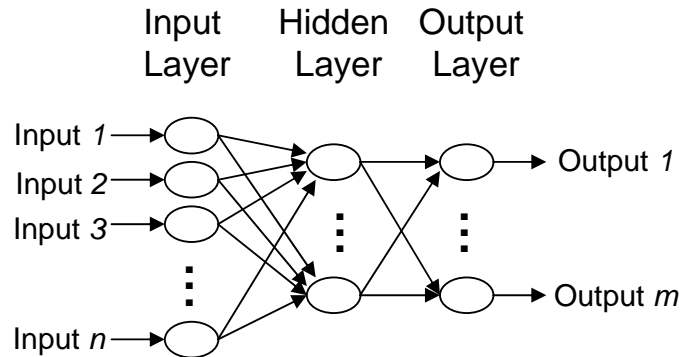


Figure 3.3: A sample feed-forward neural network

Typical ANNs have a connected, feed-forward network architecture with an *input layer*, one or more *hidden layers*, and an *output layer*, as is shown in Figure 3.3. Each layer consists of a set of neurons that are each connected to all neurons in the previous layer. Input values are fed into the input layer, with computation passing through the hidden layer(s) and finally predictions are extracted from the output layer. Each neuron computes its output by applying an activation function to a weighted sum of its inputs. Different types of activation functions involve a threshold function, a piecewise linear function, a sigmoid function, and a Gaussian function. Figure 3.4 shows an example of neuron in the hidden layer with a sigmoid activation function.

Training process in ANNs is achieved by applying learning rules to example data. Typical learning rules include error-correction, Boltzmann, Hebbian, and competitive learning. For example, an error-correction learning rule updates the weights based on the comparison between calculated, i.e., predicted, and observed values for all input samples, gradually minimizing the error between training examples and predictions.

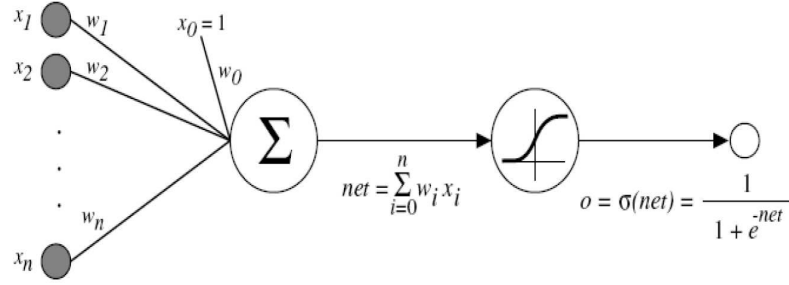


Figure 3.4: A sample hidden layer unit with a sigmoid activation function (borrowed from [81])

3.3 ASpR System Architecture

Although we propose a general purpose approach, we present our new Adaptively Scheduled parallel R (ASpR – pronounced “aspire”) framework within the context of our test platform, the pR framework [74] for transparent R script execution [96]. pR takes a sequential R script as input and transparently executes it in parallel using a master-worker model. The master parses the input script, conducts dependence analysis, and schedules non-trivial *tasks* (function calls and loop blocks) to the workers, where they are computed in parallel. The workers are assigned these tasks and carry out communication to exchange task information with the master, as well as data communication among themselves. Currently pR does not further parallelize the content of function calls or loops, and with nested loops only the outermost loop is parallelized. Only the underlying framework imposes these restrictions. Our concepts introduced in this work do not require them and can be applied to systems with hierarchical parallelization without major modifications.

In our proposed approach, ASpR, we extended the intelligence within the pR master to include three new modules, as illustrated in Figure 3.5: (1) the *analyzer*, which performs dependence analysis as well as parsing the R scripts and generates a Task Precedence Graph (TPG); (2) the *performance modeler*, which predicts task computation costs and data communication costs for the TPG; and (3) the *DAG scheduler*, which determines an appropriate schedule through a static algorithm that uses the cost predictions from the performance modeler as input and dispatches the tasks to workers accordingly.

ASpR treats the two types of tasks, function calls and blocks of independent iterations of a loop, differently in carrying out online performance prediction. Our observation

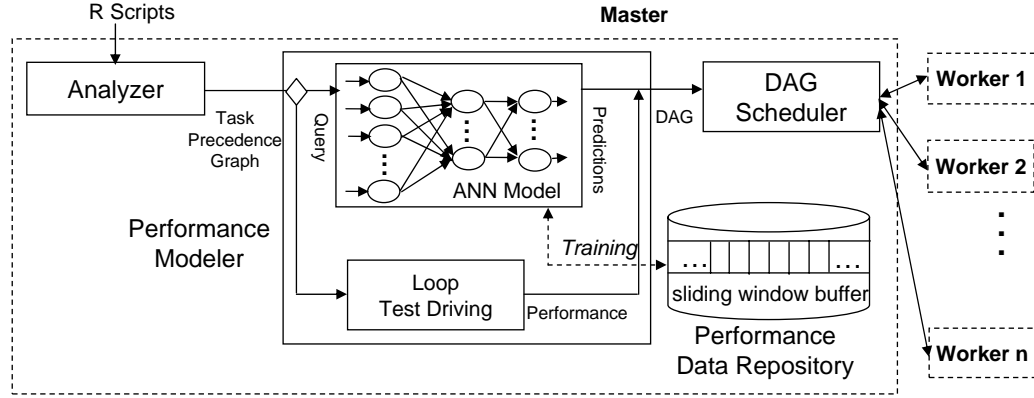


Figure 3.5: Adaptively Scheduled parallel R (ASpR) architecture overview

is that each individual user often uses a limited set of functions, standard or user-defined, which tend to be called repeatedly in this user’s scripts. The functions can be easily identified by their function names, though the calls to the same function may likely use different input parameters. Loops, on the other hand, are harder to identify but often have stable per-iteration cost in data processing scripts. Therefore, the performance modeler employs a light-weight performance data repository and adopts machine-learning methods to model and predict function task costs. For loops, it takes a more straight-forward approach by “test-driving” the initial iteration to give the per-iteration loop execution cost based on actual measurement.

A file-based performance repository collects the local script execution history for function cost prediction. We retrain the performance model as new performance data become available. To reduce the performance modeling overhead, a sliding window buffer is allocated to store the most recent data points. When applied to systems adopting the master-worker paradigm, such as pR, the system overlaps retraining with script execution on the workers resulting in low training overhead.

Many machine learning techniques can be used for performance prediction in a framework like ASpR. In our work, we chose to use Artificial Neural Networks (ANNs) [81] since they do not require language- or application-specific knowledge and they can learn automatically from periodically updated examples. Further, they provide reliable predictions across a wide variety of problems including pattern recognition, nonlinear system modeling, forecasting and prediction, and automated control. More precisely, we use SNNS,

the Stuttgart Neural Network Simulator [127], to construct neural networks and perform training and prediction.

3.4 Function Performance Profiling and Model Construction

To achieve effective task partitioning and scheduling, the ASpR system continuously collects cost information from function executions at run time to make cost predictions for future calls. Each function execution generates a training data point, which includes the function name, input parameter sizes, measured execution time, and the output parameter sizes. We train an ANN for each R function, using data points stored in the sliding window buffer. The ANN input is the data sizes of function input parameters and its outputs are the predicted function execution time and the predicted data sizes of function output variables. We predict the output data size since it corresponds to the data communication cost and the input data size for subsequent function tasks with a true dependence on the predicted task.

In our implementation, we employ a three-layer ANN with two hidden neurons, using the sigmoid function as the activation function. The edge weights are updated via back-propagation with a momentum term to improve the stability in gradient descent. Based on our previous experience with neural networks, we chose a learning rate of 0.1, a momentum of 0.9 and initial weights uniformly distributed between -0.01 and 0.01.

As a self-learning system transparent to script users, it is desirable that such a runtime parallelization tool does not require explicit training prior to use or between runs. In our design, this is achieved by reusing the query data points (the function calls whose costs are to be predicted) as training points after the corresponding tasks complete. When such new training points become available, the relevant ANNs need to be re-trained. Since the training overhead corresponds to the volume of training data, we maintain a desired size of training data by adopting a reasonable sliding window size, which is chosen according to our experiment results. Initially, with no training data available, the window grows and training is performed incrementally on all data points.

We employ a simple linear model [34] to compute the communication cost between tasks. With l as the latency and b as the size-dependent cost, the communication cost is calculated as $l + b \times data_size$. ASpR uses MPI for inter-processor communication and we

verify the above model by measuring the point-to-point MPI communication costs using a pingpong test in our testbed. Our calibration experiments show that the communication cost fits the model well and it yields an estimate $l = 0.69$ ms and $b = 2.6^{-6}$ ms/byte with less than 1% standard error.

One challenge with online learning is that the range of the parameter space is not known in advance. SNNS ANN requires normalized training data in order to achieve good accuracy. Prediction accuracy can drop dramatically for data outside the range. Alternatively, using too wide of a range also hurts prediction accuracy. Therefore, we employ a dynamic normalization scheme. When we receive data points outside of the current range, we save the new performance data and re-normalize the values in the sliding window so that the next online training uses the wider range. This way, we adapt to the performance data stream, to provide accurate predictions for a wider range and lower the exposure to outliers.

However, a problem arises if the outlier data is spatially too far away in the parameter space since re-normalization can make the range too large. Fortunately, data tend to aggregate spatially in the parameter space and our sliding-window-based online training gradually shifts parameter ranges by tracking minimum and maximum parameter values and periodically examining the sliding window to ensure that those values reflect the current range of interest. Through our experience on scientific data processing, we observed that users tend to focus on a certain range of problem sizes during a certain time period, for example, when analyzing output data generated by a group of simulation runs. The range moves to another area as users move to new simulation codes, input problems, or simulation approaches. The sliding window allows ASpR to keep recent training data in the model, which are likely to reflect the current parameter range of interest.

3.5 Loop Cost Prediction through Test Driving

The costs of loop iterations are less well defined than function tasks and are not easily identified across different executions and different scripts. Therefore, we use a novel loop “test drive” approach to predict loop execution costs. In this measurement-based approach, the master executes the initial iteration and extrapolates the performance observation to the rest of the loop. This approach works well in practice since most loops we

have seen in real-world data processing scripts have relatively stable per-iteration costs.

Unlike ANN-based prediction, the loop test drive may not be easily overlapped with concurrent task processing due to data dependences (without them, our master-worker framework performs the test drive concurrently with other task processing). With data dependences on previous task, the initial iteration cannot be executed until the entire loop may be ready to be dispatched. Thus, the loop test drive approach may create a loss of concurrency that introduces a bubble into the script processing schedule.

Alternatively, we could engage the workers in a parallel test drive that computes additional initial results. However, fine grain loop distribution and data communication (for subsequent tasks working on the output data) can be costly, especially with environments such as R. The high overhead of fine grain loop distribution makes dynamic scheduling an expensive option, even with a set of independent tasks. Therefore, a parallel test drive is unlikely to outperform our sequential test drive due to the extra cost of distributing the input and collecting the output from these small tasks.

Overall, our results demonstrate that our sequential test drive approach works well for loops with large iteration counts. However, we are unlikely to compensate for the overhead of the test drive and the possible loss of concurrency with a small number of iterations. We therefore disable the test drive in such cases. Specifically, we simply decompose the loop evenly if the number of iterations is smaller than the number of processors p times M . M is a tunable parameter that is set to 8 in our tests. Once their dependent tasks have completed, we schedule these loop tasks as soon as a processor is available (the baseline scheduling described in Section 3.7.1).

3.6 Task Partitioning and Scheduling

Once we are able to predict both task and loop iteration costs we need to include this information in our online scheduling strategy. For this we extend the existing DAG scheduling algorithm MCP (Modified Critical Path) [121] as described below. Given a set of homogeneous processors, as well as a weighted Directed Acyclic Graph (DAG) representing a group of tasks (where each node denotes one task and each edge denotes the dependence between a pair of tasks), the MCP algorithm schedules the tasks to the processors by as late as possible start time (ALAP) of a node to minimize the overall makespan. A list of

nodes is constructed according to this scheduling priority and each node is scheduled to a processor that allows the earliest start time. The intuition is to look for holes in time slots to fill in tasks. The complexity of MCP is $O(v^2 \log v)$, where v is the number of nodes in the task DAG. We choose this algorithm because it matches our problem well, where arbitrary computation and communication costs may be specified. Also, MCP is considered effective and efficient according to a benchmarking study that compared task graph scheduling algorithms [66].

One of the limitations of MCP is that it only accommodates non-malleable tasks, i.e., tasks that are not divisible and must be assigned as one unit. Therefore, it does not handle task partitioning as part of the scheduling. In our target problem, however, loops with no inter-iteration dependence need to be transparently decomposed and scheduled. Further, the decision of how to decompose the loop depends on the decision to be made by the DAG scheduling algorithm. To map our problem to MCP’s standard input, one intuitive approach is to break up the loop in a way that each iteration becomes one individual task, run MCP to decide the task mapping to processors, then merge the loop tasks back to contiguous blocks of loop iterations in the actual task scheduling. This could, however, lead to very large task graphs and significantly increase scheduling overhead. Including many iterations in each loop task, on the other hand, risks producing too large and too few tasks to effectively “fill the holes” left by other tasks.

To address this problem, we design a new algorithm that extends the MCP approach by employing a cut-and-merge scheme and list scheduling. The main heuristic adopted in the algorithm is to dynamically determine the desired level of granularity in loop tasks fed into MCP, according to the predicted costs of non-loop tasks, loop iteration test drive result, and the number of iterations.

Algorithm 3.1 describes our extended MCP algorithm, which consists of three phrases. In the first phase, the estimated execution times of divisible tasks and those of indivisible tasks are compared to determine the granularity of loop partitioning. In ASpR, the divisible tasks are the loops that can be partitioned, while the indivisible ones are function calls. When the number of iterations in a loop is considered small (compared to the number of processors p times a tunable factor M), we evenly partition it into p loop tasks without performing loop test drive. The input task DAG is then updated accordingly.

We perform test drives for independent loops and the loop decompositions depend

Algorithm 3.1 Extended MCP

Require: DAG of tasks (G)

Ensure: Schedule of tasks to p processors

/* cut phase */

 $avg_func_time \leftarrow$ average computation time of functions in G .

for each node v in G **do**

 if v is a loop with no dependence **then**

 $num_iters \leftarrow$ number of iterations of the loop.

 $iter_time \leftarrow$ per-iteration computation time from loop test driving.

 $loop_time \leftarrow iter_time \times num_iters$.

 if ($loop_time > avg_func_time \times (p \times M)$) **then**

 /* if there is no functions in G or loop dominates */

 partition v into n nodes in G and update edges.

 else if $avg_func_time > loop_time \times M$ **then**

/* if function dominates, don't partition */

else

/* function and loop costs are comparable */

 $block_size = ceiling((avg_func_time/M)/iter_time)$

 partition v into $ceiling(num_iters/block_size)$ nodes in G and update edges.

 end if

 end if
end for

/* scheduling phase */

 calculate MCP on G .

/* merge phase */

for each assignment at processors **do**

 if nodes belong to the same loop **then**

merge those nodes into one node;

repartition the loop iterations for the loop tasks;

end if
end for

on their results. If we expect a loop to dominate the total computation time (i.e., we estimate the loop computation time as at least $p \times M$ times of the average function computation time, with p the number of worker processors and M the tunable parameter from above), we distribute the loop evenly. If functions dominate the execution, then we treat the loop as a single task. Otherwise, we partition the loop into grains that will possibly fit into holes in the schedule, with the granularity again controlled by the M parameter. Thus, the algorithm selects a loop task granularity that corresponds to the function lengths. The average function execution time may not reflect the size of holes in a schedule, as large functions may still be arranged into well-aligned execution timelines. However, the potential performance benefit of using loop tasks to fill the holes would be small in this case.

The second phase applies the original MCP algorithm to the newly generated DAG, which outputs the task-to-processor assignment and execution schedule. The third phase then merges the fine-grained loop blocks from the MCP output into a condensed assignment and schedule. Basically, the loop is re-decomposed into contiguous chunks, according to the distribution of iteration counts as dictated by the MCP algorithm.

ASpR’s M parameter also bounds the number of tasks into which we will decompose the original. In the worst case, a loop node can be replaced with pM^2 nodes. By adjusting M , we can control the overhead of running MCP if necessary. As the code region to run MCP scheduling is often limited by the granularity of automatic parallelization, in environments like ASpR we do not expect large task DAGs. In our experiments, we find that MCP scheduling, even on the modified DAGs with decomposed loops, incurs very little overhead compared to other costs such as online training.

We use the sample code in Figure 3.2 to illustrate the algorithm. We perform a test drive since the number of loop iterations is 1200, which is greater than $pM = 56$ with $p = 7$ worker processors. From the test drive, we predict that the loop will take about 185 seconds, while functions `friedman.test` and `wilcox.test` will take 14.4 and 13.3 seconds respectively and 13.9 seconds on average. Since $13.9 < 185 < 13.9 \times 56$, neither the loop nor the tasks dominate. Therefore, we cut the loop into tasks with a granularity of $\lceil 13.9 / (8 \times (185/1200)) \rceil = 12$. MCP generates a schedule for the expanded graph with 9, 9, 16, 16, 16, 17, and 17 loop tasks (each of 12 iterations) on the processors. Thus, processor 1, for example, is assigned a contiguous chunk of 108 iterations.

3.7 Experimental Results

To evaluate the actual prediction-assisted task partitioning and scheduling performance, we run ASpR experiments on the *opt*⁶⁴ cluster located at NCSU, which has 16 2-way SMP nodes, each with two dual-core AMD Opteron 265 processors running Fedora Core 5. The nodes have 2GB memory each and are connected using Gigabit Ethernet. A single NFS server manages 750GB of shared RAID storage.

3.7.1 Sample Schedule

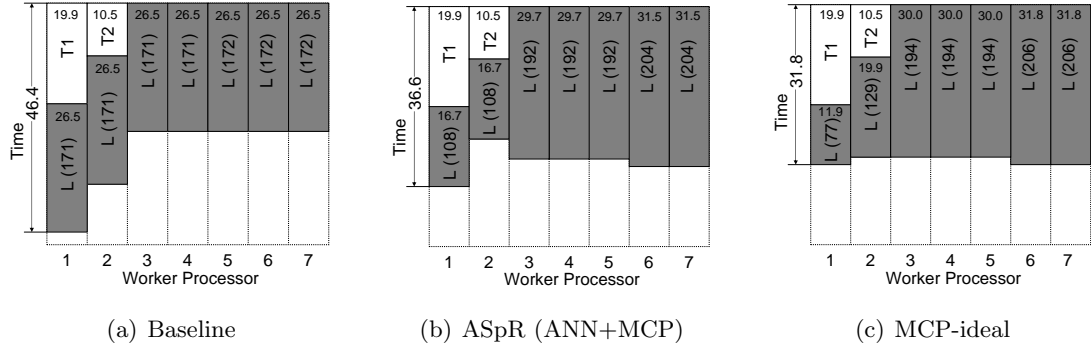


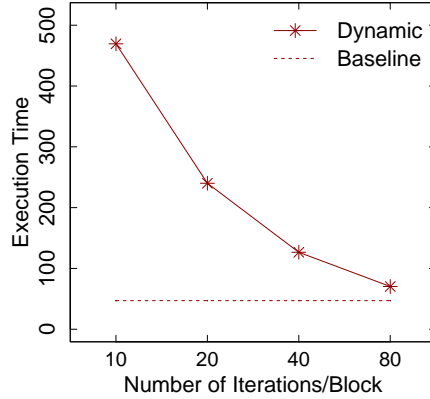
Figure 3.6: Schedules for the sample code shown in Figure 3.2

We illustrate the benefits of our proposed task parallelization assisted with on-line performance prediction through results for the sample R code given in Section 3.1. Figure 3.6 portrays the schedules generated for eight processors by three approaches: (a) *baseline*, the original pR approach, which partitions loops across processors and assigns functions in their original order to available worker processors; (b) *ASpR (ANN+MCP)*, our proposed approach, which uses the extended MCP algorithm to loop partitioning and task scheduling, based on online cost predictions given by the ANN model; and (c) *MCP-ideal*, which supplies the extended MCP algorithm with accurate task costs measured offline. The boxes again illustrate function calls and loop partitions, with their vertical lengths corresponding to measured execution times. Each loop block (shaded boxes) is marked with a number in parentheses showing the number of iterations it includes. The number at the top of each box shows the corresponding task's execution time in seconds. Since pR employs a master-worker paradigm and exports loops and function calls to workers, the schedules

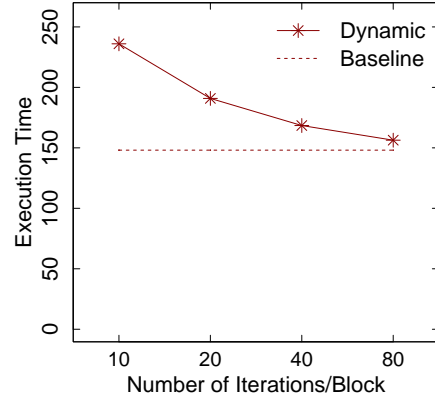
show 7 processors rather than 8.

The ASpR schedule, although not matching the schedule generated with perfect performance predictions, generates a much more balanced execution plan compared to the baseline schedule. Based on the online prediction results with 20 data points trained, the extended MCP algorithm decomposes the loop unevenly, smoothing out the workload to all the processors. It assigns the same number of iterations to worker processors 1 and 2 because the predicted execution time of `friedman.test` and `wilcox.test` is almost the same (14.4 seconds and 13.3 seconds respectively). Processors 3-5 and processors 6-7, on the other hand, receive different iteration counts due to the scheduling granularity of 12 iterations. Overall, ASpR improves the total execution time over the baseline case by 21.1% (from 46.4 seconds to 36.6 seconds). Compared to MCP-ideal, though, the scheduling granularity introduces a performance gap of 15.1%.

3.7.2 Comparison with Dynamic Scheduling



(a) R script in Figure 3.1, data intensive application



(b) Computation intensive application

Figure 3.7: Comparison with dynamic scheduling

With independent tasks, one may suspect that dynamic scheduling approaches may suit our master-worker architecture better. To illustrate the advantage of performance prediction based scheduling, we continue to use the sample code from Figure 3.1 to compare

Table 3.1: MCP overhead in seconds measured in our testbed

Graph size	Running time
41	0.001243
81	0.003279
161	0.004021
321	0.009392
640	0.024071
1279	0.073890
2553	0.213416
5104	0.751423
10206	3.701525
20410	14.927635

ASpR with a simple dynamic scheduling scheme. This approach partitions loops into smaller chunks and schedules them as independent tasks, with the master assigning, on request, tasks to idle workers from a ready queue in order to fill the holes created by the function calls.

Figure 3.7(a) portrays the results with different loop partitioning granularities, again collected from 8 processors. Although dynamic scheduling balances the load, it dramatically increases communication overhead since it uses many more loop tasks than pR or ASpR. This overhead largely arises from *serialization*, in which the system packs and unpacks R objects for inter-process communication. Although the iterations are independent, their parallel execution requires distribution of the corresponding data with the loop tasks. Thus, the overall execution time is still longer than the baseline schedule even with a relatively large partitioning granularity (80 iterations).

Since the serialization cost is script-dependent, we repeated the experiment with a different workload, as shown in Figure 3.7(b). The processing of this script is primarily a computation-intensive loop that has very limited communication footprint. Thus, this test limits the impact of serialization from the increase in messages required for dynamic scheduling. Still, dynamic scheduling performs considerably worse than the baseline scheduling, although the gap is much smaller for small iteration counts.

Meanwhile, using a static scheduling algorithm like MCP to make more globally optimized scheduling decisions comes with the cost of running the scheduling algorithm. As

Table 3.2: Description of selected R functions and their arguments in the experiments. A and B denote $n \times n$ numeric matrices.

Functions	Description	Arguments
eigen	computes eigenvalues and eigenvectors	A
prcomp	performs a principal components analysis	A
qr	computes the QR decomposition	A
svd	computes the singular-value decomposition	A
hclust	hierarchical cluster analysis	A
kmeans	performs k-means clustering	A
friedman.test	performs a Friedman rank sum test	A
ks.test	performs one or two sample Kolmogorov-Smirnov tests	A, B
mood.test	performs Mood’s two-sample test	A, B
wilcox.test	performs one and two sample Wilcoxon tests	A

mentioned in Section 3.6, the complexity of the MCP algorithm is $O(v^2 \log v)$, where v is the number of nodes in the task DAG. Table 3.1 gives the MCP execution time measured in ASpR, for different DAG sizes. With small and moderately sized DAGs (<2000 nodes), the MCP scheduling overhead is negligible. Even for a DAG size of 5000, the overhead of less than one second is small compared to the run time of typical sets of tasks in data processing codes that we target. The scheduling overhead does grow fast due to MCP’s super-quadratic complexity. However, we can limit the size of the DAG by increasing the minimum iteration count by adjusting the parameter M .

3.7.3 Benchmark Generation

To evaluate the effectiveness of online task cost prediction using neural networks, and to examine the impact of such prediction on parallel task and loop scheduling further, we need a reasonable number of R scripts with various sizes of input data as test cases. Meanwhile, to study the effect of adjusting prediction parameters, to observe the self-learning capability of our online prediction method, and to evaluate the scheduling performance with different system sizes, and finally, to assess the potential benefit of our prediction-assisted scheduling fairly, our test codes must have diverse task compositions. Therefore, we automatically generated synthetic microbenchmarks with task and parameter ranges selected randomly, and discuss their performance in Section 3.7.4 and Section 3.7.5.

We composed synthetic microbenchmarks from function calls and loops. In par-

Table 3.3: Overview of parameter spaces of selected R functions used in the experiments. N_A and N_B denote their sizes in terms of the number of double-precision numbers.

Functions	Parameters	Range	Running time
eigen	N_A	$12^2 - 2400^2$	$< 1 - 342$
prcomp	N_A	$12^2 - 2400^2$	$< 1 - 305$
qr	N_A	$25^2 - 4000^2$	$< 1 - 191$
svd	N_A	$12^2 - 2400^2$	$< 1 - 243$
hclust	N_A	$8^2 - 1600^2$	$< 1 - 180$
kmeans	N_A	$20^2 - 3200^2$	$< 1 - 223$
friedman.test	N_A	$16^2 - 2000^2$	$< 1 - 74$
ks.test	N_A, N_B	$16^2 - 3200^2$	$< 1 - 93$
mood.test	N_A, N_B	$20^2 - 4000^2$	$< 1 - 144$
wilcox.test	N_A	$16^2 - 3200^2$	$< 1 - 112$

ticular, we used function calls with non-trivial costs. To select functions from the large pool of R standard and extended functions, we performed an exhaustive study on function call frequencies in the R internal library and the well-known BioConductor project [14]. Among all 3412 R functions, the function call frequency ranges from 1 to 5943, with a median of 2. We then selected the 7 most frequently used functions for statistical tests and matrix computation, as listed in Table 3.2-3.3. The call frequency of these functions ranges from 3 to 63 in the code base we examined. In addition, we included functions `friedman.test`, `ks.test`, and `mood.test`, which are not called in the R internal library or BioConductor, but are commonly used by statisticians [11, 32, 71]. Table 3.2-3.3 lists these 10 functions, each with a short description. For each function, we also list its arguments, which in these cases include one or two $n \times n$ matrices (A and B), with their sizes in double-precision numbers denoted as N_A and N_B . The corresponding parameters used in the ANN for performance modeling are the sizes of these matrices, whose ranges are given in the table, as well as the ranges of the function execution time, shown in the last column.

To synthesize loops, we choose one expensive loop from *Boost*, a real-world application written in R from the Statistics Department at NCSU. This loop evaluates an in-house boosting algorithm for the nonlinear transformation model with censored survival data. Due to resource and time constraints, we reduced the number of iterations to 640 so that it runs for about 6 minutes sequentially. In addition, we create a synthetic loop test case, which computes the standard matrix-vector multiplication $C = A \times B[i]$ in each

iteration i , where A , B , and C are $n \times n$ matrices and i ranges from 1 to n . With this loop, the number of iterations is n .

We then generated two classes of test scripts: with independent and inter-dependent tasks. The first class contains 100 random microbenchmarks generated from our pool of functions and loops, with no inter-task dependency. In creating these microbenchmarks, we first randomly select one to ten functions and one loop. We then generate a benchmark script composed of these tasks. As these tasks are mutually independent, their relative ordering is not important. The script begins by creating random data to populate input matrices following the normal distribution. We randomly select the matrix sizes from the range of 950 to 1150.

The second class of 200 microbenchmarks are created using a similar script composition method, but with data dependency. Each benchmark includes 20 function tasks randomly sampled from a task pool, whose inter-dependency is determined by a randomly generated DAG with a given expected number of edges (5 for half of the benchmarks and 20 for the rest). To ensure one function task's input type and size match those of another task's output, while the two task have different execution costs, the function task pool used here contains ten functions that perform various iterations of matrix inversion. In half of the microbenchmarks, we also include one of the loops described above, selected at random, with no dependences on the functions.

3.7.4 Accuracy and Overhead of Online Prediction

We evaluate the effectiveness and efficiency of ASpR's online prediction using our independent microbenchmarks in this section. Given a microbenchmark, we collected training data and queries for the 10 selected functions. For training, we uniformly select 200 data points in the pre-defined function parameter range shown in Table 3.3. As we observed significant deviations in predictions but little impact on the overall schedule for small matrices, we focus instead on the queries with larger matrices. Thus, we sampled 100 random query data points within the range beyond the bottom 5% for the query data set. The execution time of these sample tasks ranges from less than 0.1 seconds to 342 seconds. The training and query sample pools do not overlap.

We perform incremental online training by inserting training points one at a time for each function in our experiments. We perform five queries at different intervals during

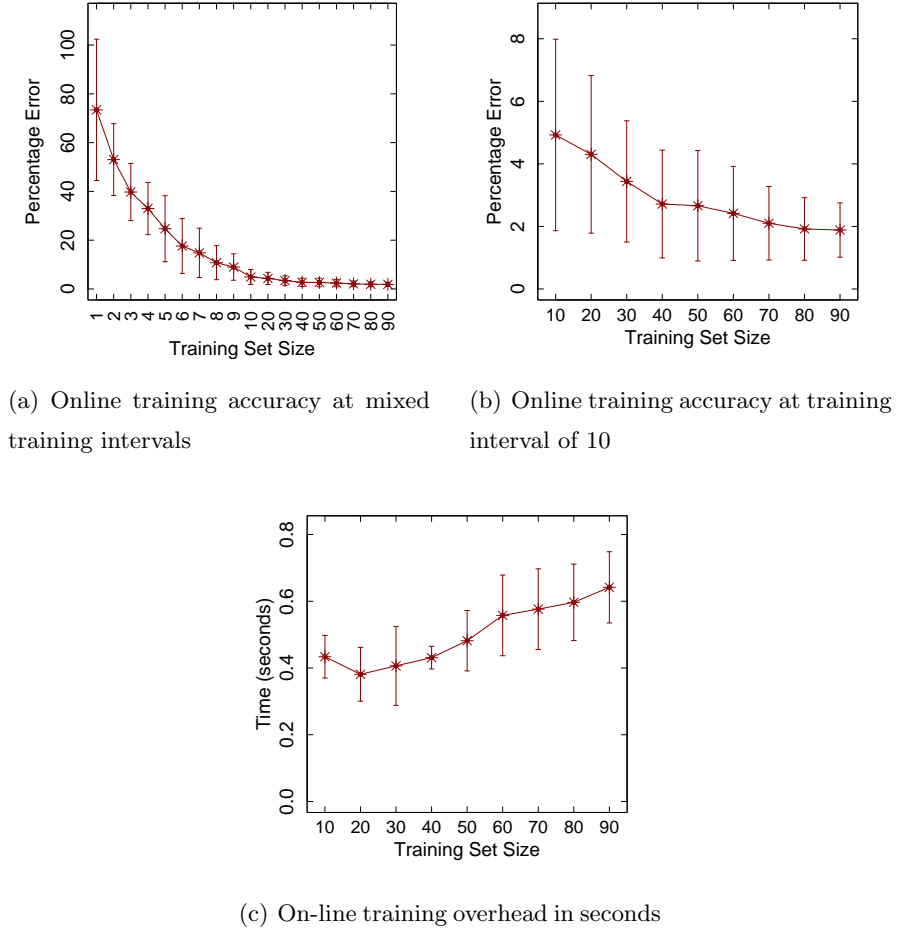


Figure 3.8: Online training accuracy and overhead for various training data sizes

the training: after inserting one new data point into the model; and after after the insertion of every 10 data points. We report the average results, as well as the 95% confidence level over the average results (from 5 queries) for the 10 functions.

Our experiments demonstrate that ASpR’s ANN-based performance prediction is highly accurate even with a relatively small training set (10-20 data points). Figure 3.8(a) demonstrates the increasing prediction accuracy as more training data are accumulated. The prediction error starts from around 80%, but quickly declines as more training points are included: with just 10 data points, the error is 4.9%. Figure 3.8(b) provides focused details of the error rate for queries executed after 10 or more training points. Using more

Table 3.4: Performance summary of ASpR on the 100 automatically generated microbenchmarks

Number of processors	8	16	32
Best improvement	40.3%	39.8%	29.4%
Worst improvement	-9.9%	-1.5%	-12.3%
Average improvement	16.7%	21.2%	12.9%
No. of enhanced cases	96	99	99

training points steadily decreases the average error rate down to 1.9% at 90 points. Though this improvement in accuracy appears small, our experiments found that it did significantly improve scheduling performance. Meanwhile, online re-training overhead grows as we increase the training data size, as shown in Figure 3.8(c): the overhead increases 48% from 10 data points to 90 (0.43 seconds to 0.64). Based on these results, we currently choose a sliding window size of 40 for ASpR, which balances the accuracies and overheads seen in Figure 3.8.

The ANN query time is independent of the training data size and fairly constant, measured as 0.07 seconds on average in our tests. Thus, the overall overhead of our online prediction approach is small, as the training and query overheads are trivial compared to task execution time in computation- or data-intensive scripts. The worst case for online prediction would be a long-running script with a long series of short function calls. However, scripts that use many short function calls normally embed them in loops and, thus, ASpR will parallelize the loops instead of the function tasks. Further, individual users tend to use a limited pool of functions so we can quickly populate the performance repository with training points. We can easily reduce ASpR online training frequency in that case. For example, ASpR could retrain the model for a function only if feedback from the workers indicates that the predictions deviate significantly from the actual runtimes.

3.7.5 Impact on Transparently Parallelized R Code Execution

Now we assess the benefit of ASpR in task decomposition and scheduling on the automatically generated microbenchmarks. Like in Section 3.7.1, we compare the baseline scheduling used in pR with the prediction-assisted scheduling in ASpR, and in some cases, with the “ideal” performance of MCP using perfect predictions.

Table 3.4 summarizes the improvement in overall execution time on 8, 16 and

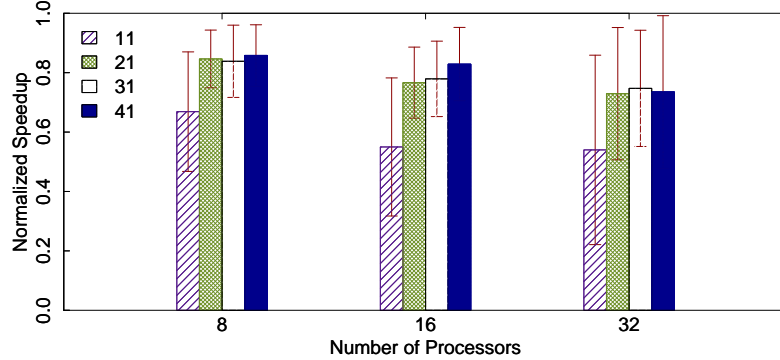


Figure 3.9: Average normalized improvement for “top 10” microbenchmarks

32 processors for the 100 R microbenchmarks with independent tasks, where we measure ASpR’s performance (“ANN+MCP”) by running the microbenchmarks after training the system with 40 data points (as allowed by the sliding window size). This performance includes the online training, query, and scheduling overhead of ASpR, which we choose not to hide with ASpR’s master-worker scheme to assess the cost of our approach without making software architecture assumptions. Again the function parameters used in the microbenchmarks do not overlap with the samples in the training data. Performance improves for 96 of the 100 microbenchmarks with 8 processors, with an average improvement of 16.7%, and a maximum improvement of 40.3%. The worst case performs about 10% worse than the baseline approach. With 16 and 32 processors, only one case out of the 100 shows a degradation. The average improvement over all cases is 21.2% and 12.9% for 16 and 32 processors and the maximum improvement is 39.8% and 29.4% respectively.

To reduce the test space for the rest of our experiments, we select the ten microbenchmarks (of the 100 total) where ASpR achieves the most significant improvement over the baseline approach with 8 processors. Figure 3.9 illustrates the self-learning property of ASpR, by repeating the microbenchmark runs after 10-data-point training intervals. We use the “normalized improvement” to examine ASpR’s performance relative to both the baseline and the MCP-ideal performance. If the execution time of a microbenchmark is $t_{baseline}$, $t_{ANN+MCP}$, and $t_{MCP-ideal}$, for the baseline, ANN+MCP, and MCP-ideal scheduling schemes, respectively. Then the normalized improvement is calculated as $(t_{baseline} - t_{ANN+MCP}) / (t_{baseline} - t_{MCP-ideal})$, which is the fraction the maximum possible

Table 3.5: ASpR and dependent microbenchmarks (8 procs)

Expected number of edges	5		20	
Containing loop	No	Yes	No	Yes
Best improvement	15.5%	19.6%	15.3%	22.1%
Worst improvement	-8.3%	-7.6%	-7.6%	-5.5%
Average improvement	1.9%	5.7%	2.5%	6.8%
No. of enhanced cases out of 50	38	46	44	49

improvement with MCP as captured by $t_{MCP-ideal}$. We also repeat these experiments on 16 and 32 processors. The y error bar shows the 95% confidence intervals.

As can be seen from Figure 3.9, in general ASpR’s performance grows as we collect more training points but the gain is marginal after 20 training points. Even with just 10 training points, ASpR realizes about 50% of the ideal MCP improvement. After 20 data points, ASpR obtains 70-80% of that ideal gain. Again, this performance is impressive considering that ASpR’s execution time includes the online training and prediction overhead that does not exist in the ideal MCP case. To illustrate the relative performance of the three approaches further, Figure 3.10(a)-3.10(c) shows the parallel execution time of these top ten microbenchmarks. Our microbenchmarks, for which the baseline approach generates imbalanced task schedules, clearly demonstrate the effectiveness of ASpR’s prediction-assisted scheduling, which nearly achieves the ideal MCP improvement. As a side note, the absolute speedup depends on the task composition in the individual benchmark scripts. In particular, data-intensive scripts suffer from high data serialization cost in R – although the tasks are mutually independent, the input matrices need to be transported to the workers. However, the effectiveness of ASpR’s prediction-assisted scheduling is evident.

We evaluate the effectiveness of ASpR for inter-dependent tasks through our second class of 200 test scripts described in Section 3.7.3, which we categorize by the expected number of edges and whether they include one of our loops in Table 3.5. With these tests, the possible performance gain depends on the specific task graph. The baseline pR tends to assign dependent tasks to the same processor when it finishes a prerequisite task, hence naturally exploiting data locality. Thus, for the ten test cases ASpR has the worst improvement (where the baseline approach generates efficient schedules), MCP-ideal only improves the performance by 1.1% on average. Nonetheless, MCP has two advantages

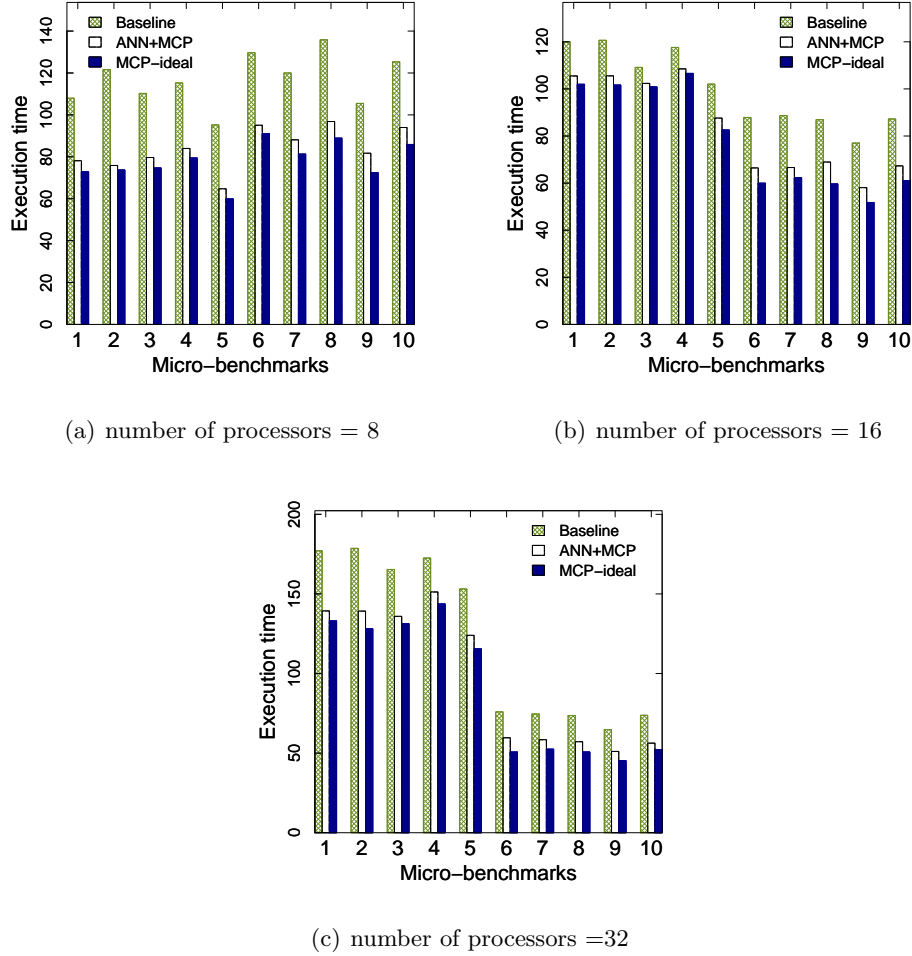


Figure 3.10: Comparison of scheduling approaches on “top 10” microbenchmarks

over the baseline: 1) it identifies the critical path and schedules those tasks early; and 2) it partitions loops unevenly, as for independent task scripts. Our results in Table 3.5 confirm that ASpR achieves more significant improvement (up to around 20%) when a loop is included in the script, and when there are more inter-task dependencies (20 edges vs. 5), which results in more paths including a longer critical path. Compared with our independent task experiments, a relative performance decrease is more likely with ASpR due to training overhead when the baseline approach happens to work well for the task graph. However, this effect will be less significant with longer runs (we used short experiments to

Table 3.6: Overview of tasks in the real R application

Tasks	Description
loop	performs model simulation
matrix	creates a matrix
rnorm	generates the normal distribution
lm	fits linear models
diag	extracts the diagonal of a matrix
crossprod	computes matrix cross-product

accommodate a large number of test cases). Finally, although space precludes more details, ASpR’s performance lags behind MCP-ideal by an amount that closely reflects the training overhead.

3.7.6 Real-world Application Performance Results

We also assessed the effectiveness of ASpR on a real-world R application obtained from the NCSU Statistics Department. This code implements a moment-based method for automatically selecting the random effects in linear mixed-effects models (LMMs). LMMs include a mixture of fixed and random factors in one unified framework and are widely used in modeling data with complex variance structures. The particular LMM code that we test enhances the model interpretation and improves the outcome prediction in the long run. It first performs multiple model simulation runs in a loop with no dependences across iterations. The remainder of the code consists of 7 function calls for adaptive Lasso, a popular technique for simultaneous estimation and variable selection. As listed in Table 3.6 the functions take vectors and matrices as input and dependences exist between the function calls. However, the functions do not depend on the loop and therefore we can execute them concurrently to the loop. We reduced the input size to reduce the total execution time (the normal input requires hours to process) but made no other changes to the application. The loop consists of 1500 iterations with a total runtime of 528 seconds, while the runtimes of the functions range from 0.2 seconds to 9.7 seconds with our input.

Figure 3.11 gives the result of running LMMs on 8, 16 and 32 processors. For ANN+MCP, we collect 30 runs of online training data in advance. With more processors, the baseline schedule becomes more imbalanced since each partition of the loop is smaller

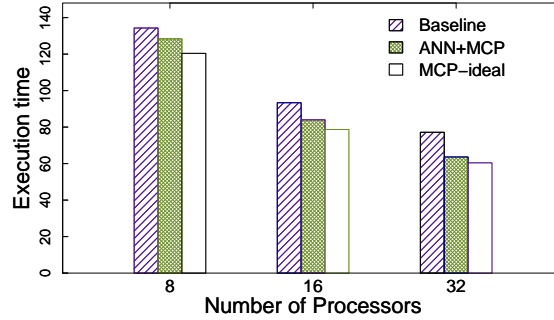


Figure 3.11: Comparison of scheduling approaches on the real R application

and the cost of the function calls becomes a greater percentage of their execution time. Thus, the difference between the ideal MCP approach and the baseline performance increases from 10.4% to 21.8%. Further, the performance of ASpR, using ANN-based prediction, more closely approaches the ideal result when we use more processors, resulting in a 17.5% improvement compared to the baseline at a scale of 32 processors. We conject that the improvement results from decreased overall prediction error when each processor receives fewer tasks.

3.8 Summary

With the emerging many-core paradigm, parallel programming must extend beyond its traditional realm of scientific applications. Converting existing sequential applications as well as developing next-generation software requires assistance from hardware, compilers, and runtime systems to exploit parallelism transparently within applications. These systems must decompose applications into tasks that can be executed in parallel and then schedule those tasks to minimize load imbalance. However, many systems lack a priori knowledge about the execution time of all tasks to perform effective load balancing with low scheduling overhead.

We approach this fundamental problem using machine learning techniques to first generate performance models for all tasks and then applying them to perform automatic performance prediction across program executions. We also extend an existing scheduling algorithm to use generated task cost estimates for online task partitioning and scheduling.

We implement the above techniques in the pR framework, which transparently parallelizes scripts in the popular R language, and evaluate their performance and overhead with both a real-world application and a large number of synthetic representative test scripts. Our experimental results show that our proposed approach significantly improves task partitioning and scheduling, with a maximum improvements of 21.8%, 40.3% and 22.1% and average improvements of 15.9%, 16.9% and 4.2% for LMM (a real R application) and synthetic test cases with independent and dependent tasks, respectively.

Chapter 4

Energy and Performance Impact Analysis

In previous chapters we demonstrate that transparent parallel processing tools such as pR and ASpR can deliver good performance as well as self-tuning capability with almost no extra development effort requested from users beyond sequential programming. Harnessing such tools for computation- and/or data-intensive scientific data analytics on multi-core personal computers brings new appeal to volunteer computing. The traditional data analytics now extends its boundary from single personal computer to multiple distributed computers in volunteer computing. On the PC owners' side, the availability of massive compute power under-utilized by personal computing tasks is a blessing to volunteer computing customers. Meanwhile the reduced performance impact of running a foreign workload, thanks to the increased hardware parallelism, makes volunteering resources more acceptable to the PC owners.

In this chapter, we set out to evaluate the potential benefit and cost of a more aggressive mode of volunteer computing, where foreign workloads are assigned to share resources on PCs with *active* native activities. We discuss the current opportunities for aggressive volunteer computing with the popularity of multi-core computers in section 4.1. Next we introduce our measurement infrastructure and workloads employed in our experiments.

In section 4.4 we estimate the efficacy of the aggressive volunteer computing model by evaluating the energy saving and performance impact of co-executing resource-

intensive foreign workloads with native personal computing tasks. Our results from executing 30 native-foreign workload combinations suggest that aggressive volunteer computing can achieve an average energy saving of around 50% compared to running the foreign workloads on high-end cluster nodes, and around 30% compared to using the traditional, more conservative volunteer computing model. We have also observed highly varied performance interference behavior between the workloads, and evaluated the effectiveness of foreign workload intensity throttling.

4.1 Motivation

Volunteer computing has been utilized for high-throughput computing for more than two decades. Tools such as Condor [73] and Boinc [19], as well as projects such as SETI@home [107], allow people in a community to pool together under-utilized resources to solve large problems or provide powerful services. The emergence and growth of multi-core personal computers currently provide additional two-fold incentives for volunteer computing. First, as the amount of compute power increases with the number of cores (which is expected to double every two years [124]), it is likely that there are more “residue resources” left idle on average PCs as light-weight, labor-intensive (and therefore highly sequential) tasks such as email processing, web browsing, and spread sheet editing are expected to stay as major components of personal computing. However, traditional volunteer computing software has been conservative in resource borrowing: existing tools typically wait until the interactive workload on a PC has been quiet for a certain amount of time (e.g., when the screen saver is on) to schedule volunteer computing tasks.

We investigate a more aggressive mode of volunteer computing, where foreign workloads are assigned to share resources on PCs with *active* native activities. The aggressive volunteer computing is motivated by two observations.

First, the *incremental* energy cost incurred by running an additional workload on an active PC is smaller than the cost of running this additional workload on a separate machine by itself. Figure 4.1 illustrates the power levels measured on a dual-core Thinkpad laptop for a variety of states, which shows that the idle state runs at about half of the peak power level, which means that energy wise, awakening an idle machine from one of the low-power states (suspended, hibernating, or shut down) is at least as costly as

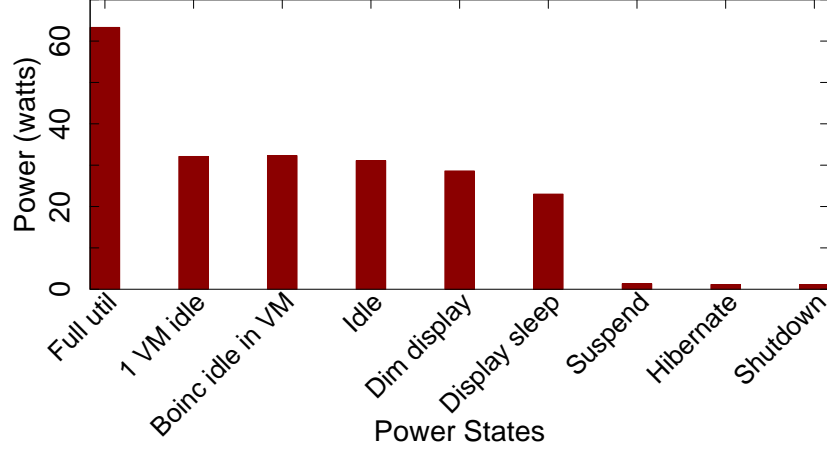


Figure 4.1: Power states of Thinkpad t61

the additional energy consumed by packing the same workload on an active machine. In addition, Figure 4.1 shows that running a virtual machine, and a volunteer computing tool (Boinc Client in this case) brings neglectable power increases on top of the idle state.

Second, with the increasing hardware parallelism in multi-core computers, aggressive volunteer computing is more feasible, as performance impact perceived by the resource owners is expected to become lower.

4.2 Workloads

Our research evaluates the energy and performance impact of running volunteer computing workloads on active personal computers carrying out the resource owners’ native tasks. In our experiments we examined the separate and concurrent execution of 30 pairwise application combinations, generated by 6 native and 5 volunteer computing (“foreign”) workloads. In the rest of this section, we give brief descriptions and summarize the resource and energy usage characteristics of these test workloads.

4.2.1 Native Workloads

The native workloads simulate the routine personal computing tasks, which includes text editing, file downloading and browsing, keyword searching, software installa-

Table 4.1: Benchmarks used as native workloads

Name	Average Run time	CPU util	RSS (peak/avg)	Total Disk I/O	L2 cache miss ratio	Average Power
OpenOffice	484.3 s	3.6%	7.5/4.4 MB	125 MB	3.8%	31.6 W
Composite	143.8 s	37.7%	14.2/8.8 MB	1186 MB	4.9%	38.6 W
Compiling	80.2 s	36.8%	12.8/8.8 MB	263 MB	6.6%	38.7 W
Viewperf	185.8 s	50.0%	595.7/233.1 MB	123 MB	2.1%	53.8 W
MESA	101.0 s	50.0%	25.7/24.9 MB	17 MB	12.2%	41.5 W
SWIM	138.2 s	50.8%	211.0/204.6 MB	15 MB	7.5%	46.2 W

Table 4.2: Benchmarks used as foreign workloads

Name	Average Run time	CPU util	RSS (peak/avg)	Total Disk I/O	L2 cache miss ratio	Average Power
BT	109.7 s	50.5%	90.3/90.1 MB	1.1 MB	0.97%	43.6 W
EP	28.3 s	51.1%	3.49/3.49 MB	2.8 MB	0.01%	39.5 W
MCF	75.5 s	50.7%	94.6/51.1 MB	1.7 MB	9.7%	42.5 W
SWIM	138.1 s	50.8%	211.0/204.6 MB	13 MB	7.5%	46.2 W
BLAST	42.1 s	47.9%	1217/638.8 MB	1688 MB	0.55%	41.0 W

tion, graphics processing, and scientific computing. They form a wide range of workload intensity levels and we selected six sample application benchmarks. Table 4.1 summarizes workload statistics collected when running each benchmark individually on a test laptop. CPU utilization and disk I/O activities are obtained from the `/proc` file system. For memory usage, the RSS (Resident Set Size) is aggregated from workload process and all its children processes obtained from `/proc/pid/status`. The L2 cache miss ratio is collected from hardware performance counters using `Perfmon2` [90] and calculated as in a previous study [9]. The average power level measures the whole-system power and is calculated from readings from a power monitor, with a sampling frequency of one per second. More details about our test platform will be given in Section 4.3.2.

OpenOffice is obtained from the Linux Battery Life Toolkit (BLTK) [17]. It is developed to measure laptop battery life and consists of six example workloads. We use its Office workload and modified its functionality just to replay Office-like editing workload without measuring battery life. The benchmark automatically launches OpenOffice Writer, Calc, and Draw applications (which mirror Word, Excel, and Visio in Microsoft Office,

respectively) and uses stored keystroke traces to replay the document editing, spreadsheet calculation and picture manipulation. The delay (think time) between a certain pair of keystrokes is stored in a file. E.g., the delay between two ASCII text keys is 150 *ms* and after the *return* key, 3 seconds. The total execution time and the total response time (the former excluding the total think time) are reported. As expected, this is a very light-weight workload.

Composite is a synthetic benchmark that simulates a series of desktop file-processing routines. It consists of the following operation sequence: (1) downloading files of a total size of 520MB from a remote machine across the network on campus via *scp*, (2) unzipping the copied file into a local directory, (3) searching for files and keywords in the directory using *find* and *grep*, (4) Compressing the file data data with *bzip*, and (5) removing all the files. Between these operations, a random time interval lasting 1-3 seconds is inserted. This workload exercises the CPU, disk and NIC, and has light memory use.

Compiling unzips and extracts the source tarball of GNU binutils version 2.18, builds binutils, and finally deleting the source directory. The workload creates multiple processes (using two cores on the test laptop simultaneously) and involves relatively heavy disk I/O.

Viewperf is a portable OpenGL performance benchmark from SPEC [110] and we use the latest version SPECviewperf 10. Viewperf renders a given data set for a pre-specified amount of time or number of frames with animation between frames and reports performance in frames per second. In our experiments we selected one SPEC dataset, CATIA (car engine and submarine models), which runs for around 3 minutes on our test laptop. This workload is both CPU- and memory-intensive.

MESA and *SWIM* are two CPU-intensive floating point benchmarks from SPEC CPU2000 [110]. *MESA* is a 3D graphics, OpenGL work-alike library. Unlike *Viewperf*, it outputs to an image file rather than the display. *SWIM* is used to test the case where a volunteered computer is used by its owner to perform scientific computing tasks. It performs shallow water modeling using a finite-difference method. Similar to *Viewperf*, it is both CPU- and memory-intensive.

Note that *viewperf*, *MESA*, and *SWIM* are CPU saturating benchmarks. However, they are only able to use one out of the two cores available, therefore showing a CPU utilization of around 50%.

4.2.2 Foreign Workloads

The foreign workloads, summarized in Table 4.2, represent the typically more computation- and/or data-intensive tasks likely to exploit volunteer computing via tools such as Condor [73] and Boinc [19]. It is most desirable to test with existing volunteer computing workloads and one of our test workloads, BLAST [3], has been offered in a volunteer computing setting [88]. However, it is not easy to obtain stand-alone, repeatable work units from existing tools. For example, projects such as SETI@home [107] usually assigns unique work units to volunteer machines, which cannot be saved or re-executed. Therefore, for the rest of our experiments we use traditional HPC and CPU-intensive benchmarks, with a goal of evaluating different application kernels.

The SPEC floating point benchmark *SWIM*, which was described earlier, doubles as a foreign workload.

Two applications are selected from the NAS Parallel Benchmark Suite, version 3.3 [84]. *BT* is a fluid dynamics simulation that solves multiple independent systems of non-diagonally dominant, block tridiagonal equations. *EP* is a numerical computation kernel that accumulates 2-D statistics from a large number of Gaussian pseudo-random numbers. It represents a CPU-intensive, memory-light, and embarrassingly parallel behavior that is particularly attractive for utilizing volunteer computing.

MCF is an integer benchmark from SPEC CPU2000 that solves minimum cost network flow. *BLAST* [3, 16], which also performs integer computation, is a widely used alignment tool to identify similarities between a query sequence and known sequences in a database. By partitioning and distributing databases into volunteer machines, it aggregates storage capacity and I/O bandwidth beyond CPU cycles. BLAST has already been deployed in volunteer computing [88].

From Table 4.2, we see that all five foreign workloads are CPU-intensive. Among them, *SWIM* and *BLAST* are much more memory-intensive than the others. Together, these benchmarks cover 5 out of the 13 application kernel types (“dwarfs”) considered significant for high-performance computing by the well-known Berkeley report on the landscape of parallel computing research [4], published in late 2006: Dense Linear Algebra (*BT*), Monte Carlo (*EP*), Backtrack and Branch-and-Bound (*MCF*), Structured Grids (*SWIM*), and Dynamic Programming (*BLAST*).

For NAS benchmarks, we use the class A problem sizes, to limit the memory usage of foreign applications. For BLAST we use version 2.2.18 and *env-nr* database, while the query sequences are randomly sampled from *nr* database.

4.3 Methodology

4.3.1 Objectives and Metrics

A key metric used in our study is the Energy Saving Ratio (ESR). Our goal is to assess how much energy can be saved by “piggybacking” a foreign workload on the personal computer where a native workload is running. Our calculation of “saving” with this execution model, *consolidated mode*, has two versions, based on comparison with two common alternative execution scenarios. The first one, *cluster mode*, represents the usual execution setting for the foreign workloads, which tend to be computation- or data-intensive, by running them on dedicated cluster nodes. The second one, *VC mode*, represents the current practice in volunteer computing, by running the foreign workloads on *idle* personal computers.

We now define how the Energy Saving Ratio for the above two base scenarios, ESR^C and ESR^V , are calculated for a given pair of native and foreign workloads. The total energy for consolidated execution, E_C , measures the energy consumed by both the native workload and the foreign workload running concurrently on a PC, with a complete or close-to-complete overlap. E_N denotes the energy consumed by the native workload running on a personal computer. Therefore $\Delta E = E_C - E_N$ is the *incremental energy* consumed by running the foreign workload in addition to the native one. We further measure E_F^C and E_F^V , the energy consumed by the foreign workload running by itself on a cluster node and a PC respectively. The energy saving ratios against the two alternative execution modes, ESR^C and ESR^V respectively, can then be calculated: $ESR^C = (E_N + E_F^C - E_C) / (E_N + E_F^C)$ and $ESR^V = (E_N + E_F^V - E_C) / (E_N + E_F^V)$. We will use these energy saving ratios to evaluate each pair of workloads.

Note that the above calculation of ESR^V assumes that when a volunteer computing tool assigns a piece of foreign work to an active PC rather than an idle one, the latter can go asleep or be shut down when running a VC daemon. This is possible with today’s volunteer computing tools running on PCs with aggressive energy-saving policies

and we have verified this with Boinc on our test platform. Although as a result, favoring consolidated execution may reduce the total number of responding machines in a VC pool, active volunteers will remain a steady work force, while a VC tool can control the size of active workers by selectively scheduling to idle nodes.

Besides energy savings, we also observe the performance impact of running the native and foreign workload concurrently on a multi-core personal computer, expressed with the percentage performance loss due to consolidation. For a given native or foreign workload, let T denote its execution time when run alone, and T' its execution time in the consolidated mode with another workload, the *slowdown* is calculated as $SD = (T' - T)/T$. For native workloads, it is desirable to keep the slowdown small so that the resource owners do not perceive obvious performance loss due to volunteer computing. For the foreign workloads, slowdown is less of a concern as volunteer computing customers often target high-throughput rather than high-performance computing [73]. However, a severe performance loss for the foreign workload when executed in the consolidated mode will prolong the total execution time and likely reduce the relative energy benefit compared to running in the cluster or traditional VC modes.

4.3.2 Experimental Platform

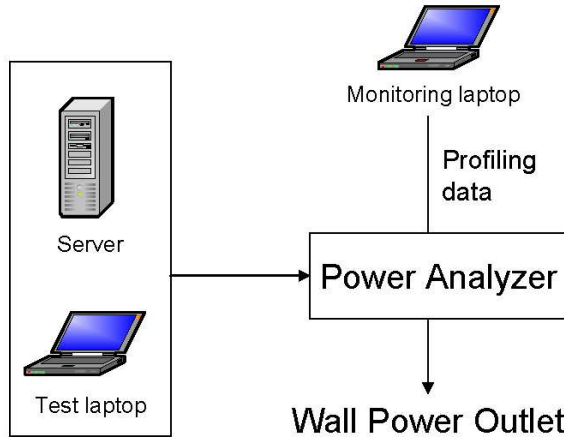


Figure 4.2: Power Experimental Setup

Figure 4.2 illustrates our testbed setup. An 8-core Poweredge server is used for experiments in the cluster mode, representing higher-end configurations typically found

in clusters. A dual-core Thinkpad T61 laptop (test laptop) is used for testing both the consolidated and the (traditional) VC modes. The server is equipped with 2 quad-core 2.33GHz Xeon CPU, 128KB L1 8-way associative cache, 12MB L2 cache, 16GB memory, and gigabit network cards. It runs CentOS 5 with kernel 2.6.18. The laptop has 2.4GHz Core 2 Duo CPUs, 64KB 8-way associative L1 cache, 3072KB 8-way associative L2 cache, 2GB memory, and gigabit Ethernet connection. It runs Fedora Core 8 with kernel 2.6.25. Our experiments performed with the native power management turned on. On the test laptop, the `ondemand cpufreq` governor dynamically adjusts cpu frequency according to the current workload, from 800MHz to 2.4GHz in steps of 400MHz allowed by this Core 2 Duo processor. The `ondemand` governor on the server works on two available frequency steps of 2.0GHz and 2.33GHz.

For power measurement, we connect the systems under test to a *Watts up? Pro* power analyzer, which measures the power from wall outlet for the entire system. It samples the power level periodically and we calculate energy consumption by integrating wattages over time. The power profile is output via a USB interface to a separate monitoring laptop, so that power profiling does not interfere with the native or foreign workloads tested.

Virtualization has become common in desktop and server environment. It provides better isolation and reliability for consolidated workloads to share resources while maintain the same level of performance. In addition, it provides added reliability, security, and portability that improves the experience for both volunteer computing customers and resource owners. For testing consolidated execution with the foreign workloads running on a virtual machine, we have VMware workstation 6.5 run on top of the native OS on the test laptop. It is configured with 1 processor, 1GB memory, and 11.5 GB pre-allocated disk. Fedora Core 8 with kernel 2.6.23 is installed as host OS. When the system allocates memory for virtual machine, we choose the default option, which allows virtual machine memory to be swapped. In this way, the foreign workload does not have higher priority over native workload. Memory trimming and page sharing are disabled in order to enhance I/O performance.

4.3.3 Measurement Scheme

Now we describe in detail our measurement scheme that allows us to obtain the metrics discussed in Section 4.3.1. For comparing energy consumption, we need to keep track

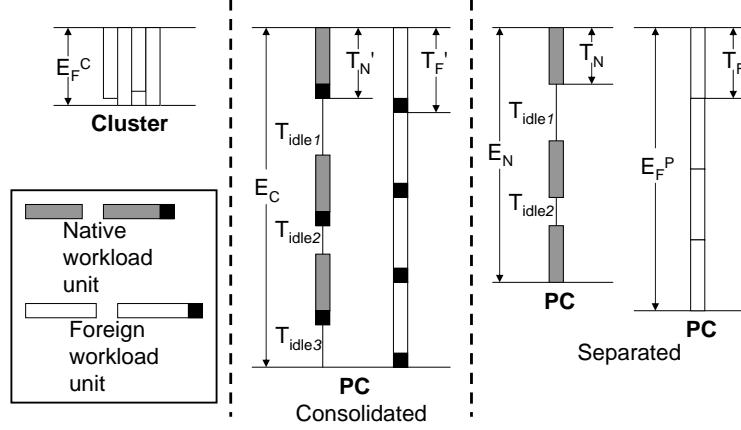


Figure 4.3: Measurement scheme overview

of the total amount of work performed in both the native and foreign workloads. Figure 4.3 illustrates the running and overlapping of workloads in different execution modes. The gray blocks indicate native work units and white ones represent foreign work units. The black segments illustrate the slowdown due to concurrent execution. The lines between blocks refer to idle intervals.

First, we decide the amount of work for each foreign workload, so that we have enough work units to 1) achieve a balanced execution schedule on the 8-core server for a fair comparison of energy consumption against the cluster mode, and 2) overlap properly with the native workloads selected. For workloads other than BLAST, there are no straightforward way to select a variety of the inputs, therefore we replicate the benchmark work units. However, we believe that replicating the work units with identical input problems does not affect their execution, and have verified that repeating the same work units sequentially does not have an impact on each unit's performance or energy consumption. For BLAST, each work unit is a unique sequence query to the same sequence database.

After selecting the amount of work, each foreign workload is executed on the 8-core server to measure E_F^C , its energy consumption in the cluster mode. To make a conservative comparison, we search for the most energy-efficient execution plan, by trying out different levels of concurrency. Our results are presented in Section 4.4.1. Based on these results, we choose to measure E_F^C with 4 concurrent instances for MCF and 8 instances/threads for the rest of foreign workloads.

Next, we calculate the number of native work units to maximize the overlap with the foreign workloads, by calculating the maximum number of units whose sequential run time will not exceed the concurrent, sequential execution of all the foreign work units on the test laptop. To simulate realistic personal computing patterns, idle periods are inserted between native work units, creating a roughly even-paced work schedule. The idle intervals account for less than 11% on average of the schedule across all workload combinations.

The native and foreign workloads are then run together in the consolidated mode. We measure the total consolidated energy consumption E^C for the entire schedule, the first native work unit execution time T'_N , and the total foreign work load execution time T'_F , as shown in Figure 4.3.

Finally, we re-execute the native and foreign work units separately, on the same test laptop, to assess the energy consumption and performance when consolidated execution is not used. Figure 4.3 shows the intuitive method we used to measure T_N , T_F , E_N , and E_F . Note that in measuring E_N , we replicate the idle intervals inserted between the native workloads in the consolidated executions.

To avoid the cache effect, a 2GB file is read into memory to flush the cached blocks before the beginning of each execution session, whose time and energy costs are not included in our measurement.

4.4 Results

4.4.1 Impact of Concurrency in the Cluster Mode

First, as mentioned in Section 4.3.3, to fairly calculate ESR^C , we determine the most energy-efficient configuration to run foreign workloads on the 8-core server.

For benchmarks that can run with multiple threads (BT, EP, and BLAST), we vary the number of concurrent threads in processing the fixed set of work units. Otherwise, we vary the number of concurrent instances (processes). Figure 4.4 portrays three representative behaviors with varied level of concurrency, showing the total execution time, total energy consumption, and average CPU utilization as well as power levels, normalized against the values obtained with 1 thread/process running sequentially. The results of EP and BLAST are removed as they are very similar to that of BT.

All workloads show speedup with increasing concurrency, up to a certain point,

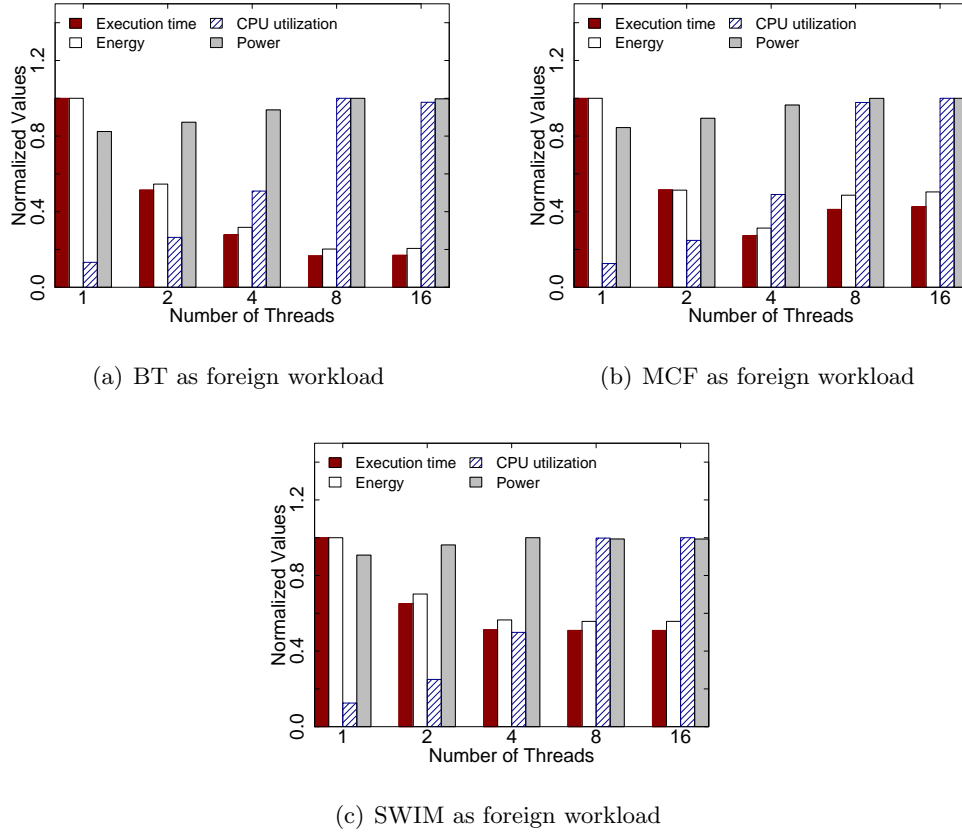


Figure 4.4: Performance/energy efficiency of foreign workloads on 8-core server with varied level of concurrency

which not surprisingly also appears as the most energy-efficient setting. Figure 4.4 shows that the difference in average power between different concurrency levels is rather small, and therefore the energy consumption is roughly proportional to the total execution time. Compared to other workloads, SWIM's power saturates faster and its speedup is smaller, due to its heavy memory usage and contention. Based on these results, we measure E_F^C using 8 concurrent threads/instances for all foreign workloads except MCF, which consumes the least energy when 4 instances run concurrently.

4.4.2 Consolidated Execution in Native OS

Figure 4.5 shows our major results for all 30 workload combinations: the energy saving of using the consolidated execution mode, as opposed to the cluster and the tradi-

tional VC mode. For each combination, the group of three bars represent the total energy consumption running in the cluster, traditional VC, and consolidated mode respectively. With the first two modes, the foreign and native workload are executed separately, therefore the total consumption is calculated as the sum of their individual consumption (stacked bars). As the native workload is run only on the laptop, E_N does not change between the cluster and the traditional VC mode. The consolidated mode is represented with a single bar plotting E_C . Note that the figures have different y axis scales.

In general, these results demonstrate the significant energy saving we can obtain by aggressively overlapping volunteer computing workloads with native activities. Compared to the cluster mode, doing so saves energy by 52.1% on average, with a maximum saving of 78%. Note that for the cluster mode, the energy consumption we measured only considers the system’s operation, and does not include the cost of machine-room cooling, which is typically significant due to the power-density of clusters [82]. Compared to the traditional VC mode, the saving is smaller but still significant: average of 29.7% and up to 42%.

The difference between ESR^C and ESR^V reveals the energy-performance trade-off of using high-end cluster nodes for intensive workloads. Although multi-core server nodes offer higher processing throughput and for the unit amount of compute power, a more compact form factor, they consume considerably more energy to complete the same amount of work (again even without considering the cooling expense). As current multi-core systems are not able to shut down idle cores, inefficiency in parallel/concurrent execution will translate into wasted energy. This is demonstrated by the different foreign workload behaviors: with EP, which runs more efficiently in the cluster mode, the energy savings are smaller, so is the difference between ESR^C and ESR^V . In contrast, SWIM and BLAST benefit more from avoiding the cluster mode, where the memory and I/O contention produces low energy efficiency.

The major cost of such energy saving, of course, is the potential slowdown of the workloads when executed in the consolidated mode. In particular, a non-trivial native workload slowdown may discourage resource owners from participating in volunteer computing. Figure 4.6 depicts the SD_N and SD_F , the relative slowdown of the native and foreign workloads respectively, from separated to consolidated execution on the test laptop.

With multi-core PCs, the impact on a sequential native workload of running a foreign workload concurrently is expected to be modest. Many workload combinations (12

out of 30) meet this expectation, with a native workload slowdown of 10% or less. These do not include OpenOffice, whose results are calculated in a pessimistic manner: we report the slowdown based on the measured *response time*, which counts for less than 3% of the total benchmark execution time (> 480 seconds). We further found that most of the delay comes from the program startup and shutdown stages. This means that for the majority of time a user is performing the editing work, the performance impact would be hardly visible considering that it is quite normal that one spends more than 8 minutes in an editing session of Office files. On the other hand, slower application startup time is often annoying to PC users.

It appears from Figure 4.6 that the most vulnerable native workload is Compiling, which is less CPU/memory intensive than most other workloads. We found that this is due to two factors: (1) This workload contains a large number of inexpensive commands, with more than 1000 processes created within 80 seconds. As mentioned above, the impact of running foreign workload on program startup seems to be especially heavy. (2) This benchmark is able to use both CPU cores with concurrent processes and is therefore hit harder by an additional workload.

In addition, our results show that with multi-core PCs, memory contention becomes the major factor contributing to performance slowdown. Overall, the more memory-intensive benchmarks (MCF and SWIM) cause much more native slowdown than the CPU intensive ones (BT and EP). BLAST affects the I/O intensive workloads (OpenOffice, Composite, and Compiling) significantly, while introducing little impact on the others. Also, the foreign workloads, being more CPU-intensive, appear more resilient than the native ones. Notable exceptions are the slowdown causing by memory-intensive SWIM and Viewperf on similar foreign benchmarks (SWIM and MCF).

The relationship between memory contention and performance slowdown is illustrated by Table 4.3, which lists the L2 cache miss ratio for each native workload, measured with its solo execution (“original”), and with consolidated runs pairing with different foreign workloads. The results for the Compiling benchmark’s consolidated runs are not reported, as we found that the execution of the native work units is heavily prolonged when Perfmon2 is used to for hardware counter profiling, resulting in altered overlap behavior. This is due to the dramatically increased profiling overhead brought by the large number of small processes created in this benchmark. Overall, we see a strong correlation between the increase

Table 4.3: L2 cache miss ratios of native workloads

	Original	BT	EP	MCF	SWIM	BLAST
OpenOffice	3.8%	7.9%	5.5%	11.0%	12.0%	5.9%
Composite	4.9%	10.5%	8.1%	24.1%	26.4%	7.5%
Compiling	6.6%	NA	NA	NA	NA	NA
Viewperf	2.1%	3.1%	2.3%	5.7%	14.5%	2.7%
MESA	12.2%	16.3%	13.1%	27.6%	33.2%	13.7%
SWIM	7.5%	9.0%	7.7%	10.7%	20.0%	8.8%

in L2 cache miss ratio and the native workload slowdown shown in Figure 4.6. A notable exception is the MESA benchmark, which sees a significant rise in L2 miss ratio when running with MCF or SWIM, but appears to be less sensitive in its overall performance due to its less memory-intensive nature.

Table 4.4: Average CPU utilization and power consumption of native workloads, running alone or with foreign workloads

	Individual	BT	EP
OpenOffice	3.6%/31.5	52.7%/43.7	52.5%/39.5
Composite	37.7%/38.6	82.8%/47.4	81.8%/43.9
Compiling	36.8%/38.7	70.6%/45.8	69.2%/41.8
Viewperf	50.0%/53.8	95.8%/60.4	96.0%/58.3
MESA	50.0%/41.5	96.2%/49.1	94.3%/45.0
SWIM	50.8%/46.2	94.5%/51.7	95.5%/50.1
	MCF	SWIM	BLAST
OpenOffice	53.4%/42.7	53.3%/46.3	48.6%/40.8
Composite	87.4%/46.7	89.7%/49.3	78.1%/43.1
Compiling	71.4%/44.9	74.6%/48.2	70.0%/43.5
Viewperf	92.4%/56.4	91.5%/56.8	94.8%/58.6
MESA	96.2%/48.1	96.1%/50.9	96.6%/47.0
SWIM	91.9%/48.1	96.0%/49.7	94.2%/51.1

Finally, we check the relationship between the energy impact and the performance impact incurred by the foreign workloads. Table 4.4 summarizes the average CPU utilization and the average power level for each native workload, when running alone or paired with a foreign workload. While we did not find obvious correlation between the power level increase and the native/foreign workload performance slowdown shown in Figure 4.6, Table 4.4

Table 4.5: Performance degradation of foreign workloads running within VMware

	BT	EP	MCF	SWIM	BLAST
In native OS	109.7	28.3	75.5	138.1	184
In VM	123.9	31.6	91.1	155.3	225
slowdown	12.9%	11.7%	20.7%	12.5%	22.3%

confirms the speculation that the lower the CPU utilization of the native workload is, the higher the power level increase. It also shows that CPU utilization does not dictate the power level. For example, applications like Viewperf produces high power level with the heavy use of additional hardware (such as GPU).

More interestingly, results shown in Table 4.4 and Figure 4.5 reveal that the increase in power level does not directly translate into incremental energy cost. For example, SWIM sees a small 3.5-Watts power level increase when running against another instance of the same application as the foreign workload, which is lower than other combinations, such as SWIM-BT. However, Figure 4.5 indicates that SWIM-SWIM bring a relatively high incremental energy cost ($E_C - E_N$), which is almost equivalent to E_F^V . This is because that the energy consumption involves both the power and the time factor. Although the incremental power cost is low due to the heavy memory contention, the incremental energy cost is high due to the prolonged execution.

4.4.3 Consolidated Execution with Virtual Machine

Next, we inspect the effect of using a virtual machine environment, by repeating the consolidated runs and the traditional VC mode runs within VMware Workstation 6, which runs on top of the native operating system. Figure 4.7 and 4.8 report the energy saving and performance impact results, respectively.

The overall results are quite similar to those obtained without using a VM, suggesting that aggressive volunteer computing using the consolidated execution mode retains its energy benefit and performance trade-off when the foreign workload is contained inside a virtual machine.

Energy wise, the major differences are that the traditional VC mode becomes slightly more expensive compared to the cluster mode, due to the extra energy cost of running the virtual machine. Overall, the energy saving numbers are quite similar to the

native OS results, with an average of 48.2% and maximum of 75.6% for ESR^C , and an average of 30.9% and maximum of 42.1% for ESR^V .

Performance wise, the good news is that VMware does provide a certain level of performance isolation, so that the performance of Compiling, our most vulnerable native workload, is better protected from the foreign workloads. Also, the slowdown on the foreign workload caused by the native activity looks similar to that measured without VM, with slight increases in several cases. The bad news, however, is that the foreign workloads take longer to complete within a VM. Table 4.5 shows the execution time of the foreign workloads with and without VMware, with percentage slowdown ranges from 11.7% to 22.3%. Fortunately, the performance impact on foreign workloads for volunteer computing is less important. But we need to point out that the BLAST work units in these tests used a smaller, 600MB database, as the original database is larger than the VM memory limit and caused heavy swapping.

4.4.4 Throttling with Vulnerable Workload

Finally, we investigate the potential impact of making aggressive volunteer computing more “polite” to resource owners, by throttling the intensiveness of the foreign workloads. This technique has been examined for volunteer computing previously [113]. Our goal here is to observe the performance and energy consequences together.

Due to experiment time and space concern, we focused on four pairs of native-foreign workload combinations where the severe native workload slowdown has been observed and the workloads show different resource usage characteristics: Composite-SWIM, Composite-BLAST, Compiling-MCF, and Compiling-SWIM. In our experiments, we assigned lower scheduling priority to the foreign process, by setting its nice number to 10 and 15 respectively (as opposed to the default value of 0), through the Unix `nice` command. Figure 4.9 shows the comparison between different priority levels, in the native and foreign workload slowdown, as well as the energy saving ratio ESR^V .

For the majority of cases, throttling down the foreign workload generates the expected effect: the native workload slowdown is reduced while the foreign workload suffers more performance loss. The amount of relative changes in performance is also mirrored between the native and the foreign sides. An exception in the effectiveness appears to be the Composite-SWIM combination, where the throttling does not seem to generate

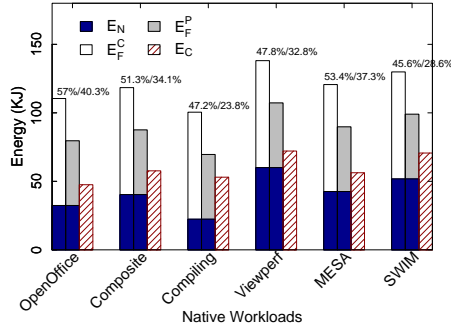
much of an impact. The reason is likely that, Composite, being a more I/O-intensive workload, does not fully benefit from the CPU scheduling advantage it receives. In contrast, when it is paired with BLAST, another I/O-intensive workload, the performance gain is obvious though further reducing the BLAST priority failed to generate additional relief for Composite. This indicates that mechanisms beyond scheduling priority throttling may be necessary and it is important to dynamically adjust the throttling strategy according to the native workload’s behavior.

When it comes to the impact on energy saving, Figure 4.9(c) shows that interestingly, ESR^V slightly increases when we slow down the pace of the foreign workload. The intuition here is that by being more polite to the resource owner, a larger portion of the resources are consumed by the native workload, decreasing the difference between E_N and E_C . One may argue that the amount of native work units will not increase just because the foreign workload runs slower. However, we perceive an aggressive VC framework that can migrate foreign workloads out from PCs that has become idle, to another volunteer node with active native activities.

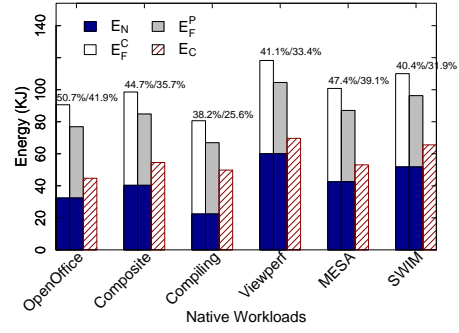
4.5 Summary

In this work, we assess the efficacy of aggressive volunteer computing, by evaluate the energy saving and performance impact of co-executing resource-intensive foreign workloads with native personal computing tasks. Our experiments examined the pairwise combination of 5 foreign and 6 native workload benchmarks, and investigated the effect of running the foreign workloads within a virtual machine, as well as throttling their CPU scheduling priority. The results confirm that aggressive volunteer computing provides an attractive computation model, offering an average energy saving of 52.1% compared with running the foreign workloads on multi-core high-end cluster nodes, and 29.7% compared with running them on idle PCs using the traditional volunteer computing model. With the foreign workloads running inside a virtual machine, the average savings are 48.2% and 30.9% respectively. While many combinations display small (less than 10%) slowdown of the native workload, the performance of certain native benchmarks appear to be highly vulnerable to resource sharing, highlighting the necessity of intelligent, dynamic techniques to contain the intensity of foreign workloads. Our results also suggest that slowing down

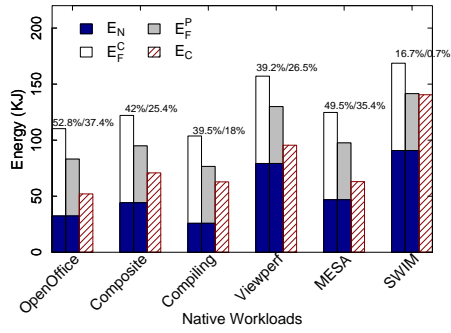
the foreign workloads does not significantly shrink the energy saving.



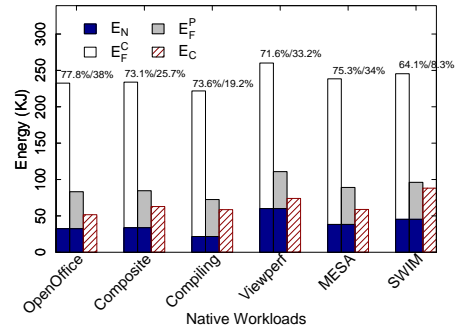
(a) BT as foreign workload



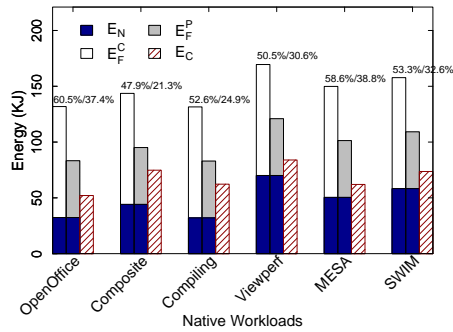
(b) EP as foreign workload



(c) MCF as foreign workload

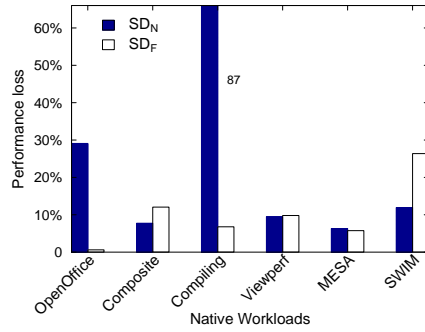


(d) SWIM as foreign workload

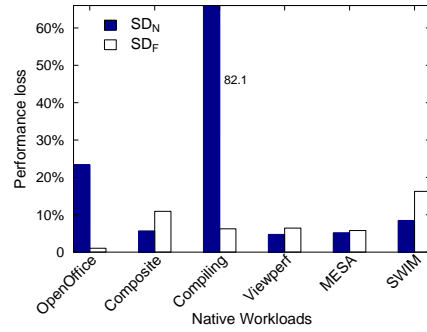


(e) BLAST as foreign workload

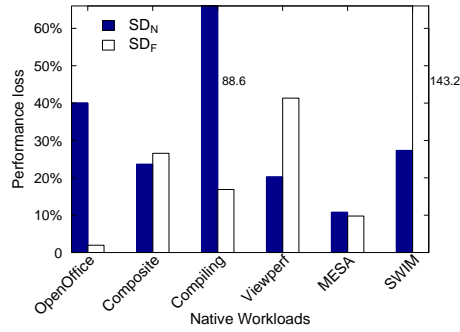
Figure 4.5: Energy saving of consolidated execution in the native OS. The pair of numbers above each group of bars show ESR^C/ESR^V for that workload combination.



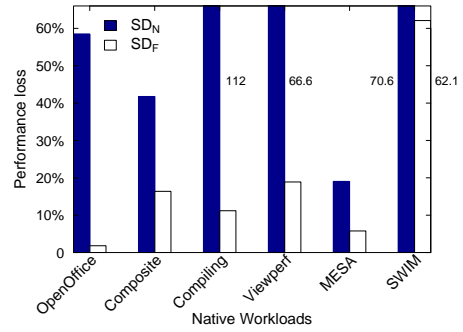
(a) BT as foreign workload



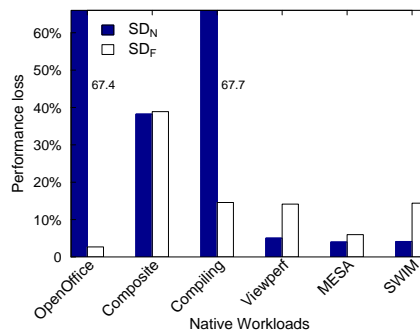
(b) EP as foreign workload



(c) MCF as foreign workload

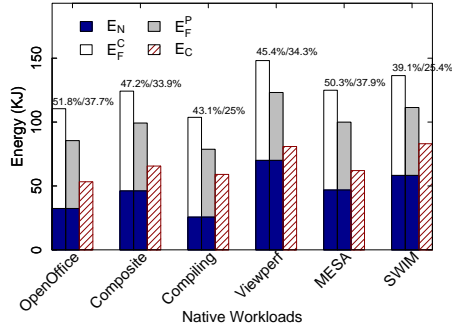


(d) SWIM as foreign workload

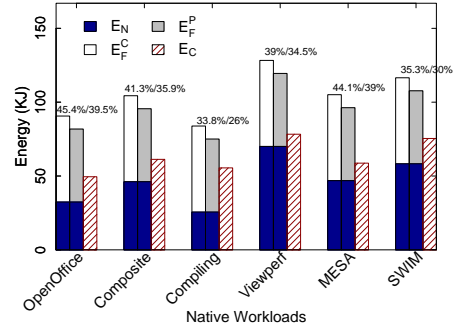


(e) BLAST as foreign workload

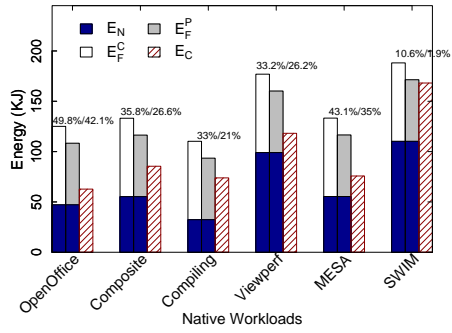
Figure 4.6: Performance impact when workloads consolidated in the native OS



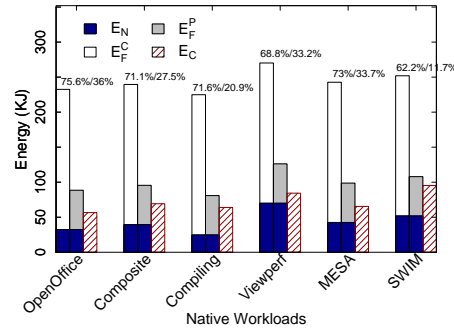
(a) BT as foreign workload in VM



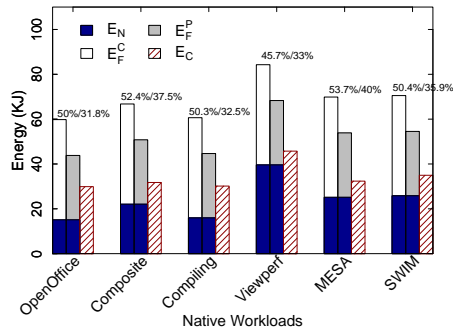
(b) EP as foreign workload in VM



(c) MCF as foreign workload in VM

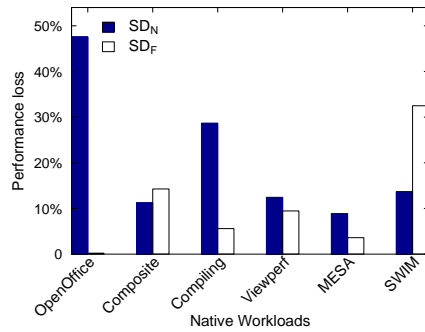


(d) SWIM as foreign workload in VM

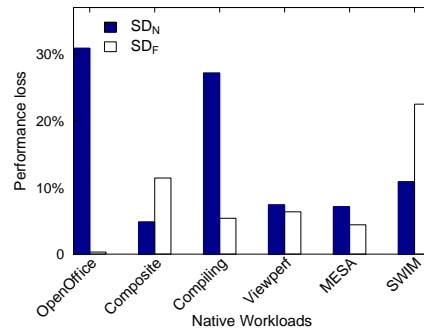


(e) BLAST as foreign workload in VM

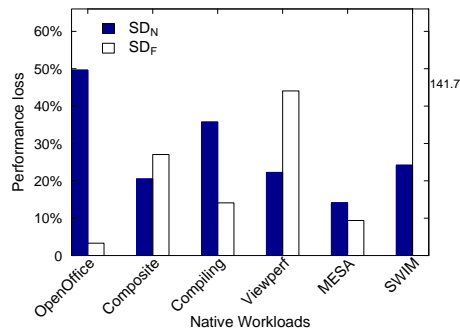
Figure 4.7: Energy saving of consolidated execution in VM. The pair of numbers above each group of bars show ESR^C/ESR^V for that workload combination.



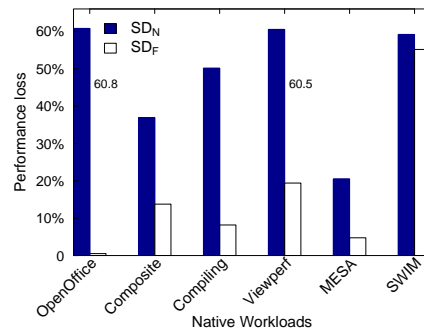
(a) BT as foreign workload in VM



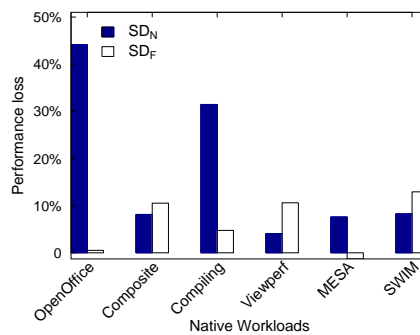
(b) EP as foreign workload in VM



(c) MCF as foreign workload in VM



(d) SWIM as foreign workload in VM



(e) BLAST as foreign workload in VM

Figure 4.8: Performance impact with foreign workloads running in VM

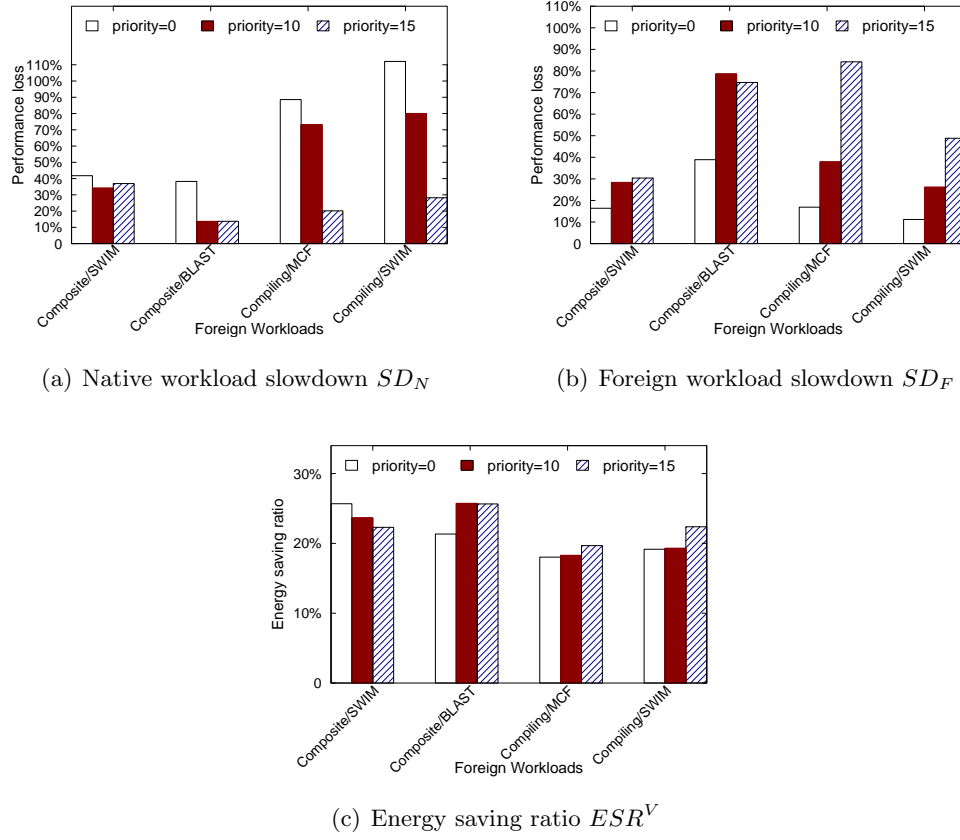


Figure 4.9: Effect of foreign process throttling with selected workload combinations

Chapter 5

Related Work

With the emergence of multi-core architectures, transparent parallel processing has re-gained tremendous research interests. In this chapter we categorize our survey of related work in the following five areas. In section 5.1 we describe the current state-of-the-art in transparent parallelization with a focus on scripting languages. Techniques in performance prediction and task scheduling that will assist transparent parallel processing are reviewed in section 5.2 and 5.3. Finally we examine the power-performance issues and relevant projects in section 5.4 and 5.5.

5.1 Transparent Parallelization

Automatic parallelization has been extensively studied in the context of parallelizing compilers. Representative work on parallelizing compilers, such as SUIF [54], Polaris [18], and Paradigm [7] transform sequential C or Fortran code into parallel code (e.g., using OpenMP). Our transparent parallelization framework has borrowed mature techniques from these projects, such as loop dependence analysis and loop transformation [1, 6, 18, 54, 94, 100, 117]. 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 our framework utilizes 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 [99]. Chen et al. proposed an algorithm for run-time parallelization by inspecting memory access patterns first [23]. 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 [102]. Gupta et al. presented a set of run-time tests for speculative parallelization of loops [52]. Most of these approaches focus on exploiting parallelism from loops where there exist certain forms of inter-iteration dependence. In contrast, our approach 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 [25]. Like other systems using speculative multi-threading (e.g., [41]), 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 our framework supports task parallelism beyond loops.

In addition, our dynamic, incremental dependence analysis technique is related to but quite different from the existing dynamic compilation technology. Mechanisms such as just-in-time compilation [33] and incremental compilation [105] are designed to reduce the overhead of runtime bytecode interpretation or interactive compilation. Our technique is closer to dynamic compiling systems such as DyC [48]. 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.

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 [29] 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 orches-

trate communication as well as synchronization. Back-end support has been a popular approach utilizing high-performance numerical libraries such as ScaLAPACK [27] to carry out Matlab routines in parallel on multiple servers. Star-P [30], 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 [53] is a runtime system that can compile and automatically execute Matlab script in parallel. This work is perhaps the closest to our framework in performing runtime automatic parallelization for scripting languages. MaJIC [2] provides a just-in-time and transparent framework for compiling Matlab while maintaining its interactive nature. Otter [95] 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. Besides, those tools typically do not parallelize I/O operations. For example, with Otter [95], file operations are handled by one single processor. Our framework 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.

Similar categorization applies to existing work on other scripting languages. For example, pyMPI [80] provides MPI interfaces for parallel programming in Python and belongs to message passing category. For R, the snow package [101] exploits embarrassingly parallel tasks and belongs to category 1, while RScalAPACK [103, 125] 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 [46].

In contrast, our framework 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 [114], 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, our runtime parallelization can be incorporated with back-end support to speed up expensive standard function calls.

The snow package [101] 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 our approach currently only supports batch-mode runs. However, our proposed parallelization scheme is more general (for example, it can parallelize two heterogeneous function calls), and unlike snow, it does not require any modification to the sequential R source code.

Regarding transparent or semi-transparent parallelization, there have also been a few related projects. A simple yet powerful interface called MapReduce was designed for processing and generating large data sets in Google [36]. With MapReduce, the run-time system handles data partitioning and task scheduling automatically. However, it is targeted toward a certain type of data parallelism (where the same operation is performed over a large set of objects, with a reduction operation at the end to merge the results). Users still need to explicitly specify the Map and Reduce operations. To a certain extent, our transparent parallelization framework resembles OpenMP [86], 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.. In contrast, our framework handles parallelization transparently and exploits both data parallelism and task parallelism, with no extra effort requested from programmers. In addition, Seinstra et al. designed a software architecture for parallel image processing [106]. 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.

5.2 Performance Prediction and Self-learning Systems

Several tools trace or analyze application performance, including Paraver [92], Open|SpeedShop [104], TAU/ParaProf [13], svPablo [38], and VampirTrace [83], just to name a few. These tools provide developers of compiled language applications with data that guides optimization. In contrast, we profile a scripting language to guide automated

parallelization and scheduling decisions.

Many projects, such as PerfDMF [57], PerfTrack [62], and Prophesy [122], combine the use of a repository with performance profiling and modeling to facilitate performance data storage and management. However, these existing tools use profile data in an offline manner for performance modeling and debugging. We integrate a light-weight database in the form of a file-based data repository for online performance prediction, which supports runtime parallelization and scheduling even in the same run as which the data is gathered.

Past research has adopted machine learning methods in performance modeling and prediction, including automatic performance modeling for parallel I/O systems [126], parallel applications [59, 69], and architectural design space exploration [58]. Lee et al [68] used regression methods and filter techniques to predict application execution times. The MetaSim automated prediction framework convolves application signatures and machine profiles to form application predictions [20]. Other work estimated applications' execution times on various machines based on parametric code profiling and analytical benchmarking techniques [123]. Perhaps most similarly to our performance prediction approach, previous work used ANNs to guide concurrency throttling at runtime [35]. In contrast, our approach adapts the ANNs, as well as using their results, at runtime and we target a self-learning system for unbounded parameter spaces.

Self-configuring, self-managing and self-learning systems have received increasing interest. Wildstrom et al. [120] proposed a self-configuring system that adapts to the current workload in distributed computer systems. Their work focuses on varying CPU and memory resources and does not require any instrumentation of the middleware and operating system. Neural-network-based self learning was applied in dynamic resource allocation on a chip multiprocessor [47]. The learning, which targets the intra-chip level, serves to assign cache banks to processing cores. We take this approach one step further by adaptively parallelizing the applications according to runtime performance observations, without user-supplied *a priori* knowledge on tasks or parameters.

5.3 Resource Allocation and Task Scheduling

Parallel job or task scheduling is another mature research area; here we briefly summarize the most related topic, scheduling a parallel program represented by a directed

acyclic graph (DAG) on multiprocessors [67]. DAG scheduling is quite generic and applies to many systems and applications. Since optimal DAG scheduling is an NP-complete problem [70], most research focuses on finding good heuristic solutions. Our scheduling scheme is most related to scheduling algorithms such as DLS [108] and MCP [121], which deal with arbitrary computation and communication costs, on a limited number of fully connected processors. However, these algorithms do not readily handle loops that are partitioned at runtime as a part of the scheduling task.

Resource allocation and scheduling have been jointly investigated, often with the assistance of performance profiling. The Paradigm compiler proposed a two step allocation and scheduling approach [98], using convex programming to decide the number of processors allocated to a task. Bansal et al. [10] proposed a two-step Modified Critical Path and Area-based (MCPA) scheduling heuristic to balance processor allocation and task assignment in scheduling. In addition, a prior study employed performance modeling in workflow scheduling on Grid resources [76]. While sharing the same goals with these existing projects, our scheduling strategy does not require in-advance profiling of application behaviors, nor the knowledge on hardware resource configuration.

Other research has investigated scheduling with mixed task and data parallelism, presenting approximation algorithms, both online [12] and offline [118]. Target problems of these algorithms make efficiency assumptions on task speedup functions. Chakrabarti et al. conducted an empirical study on the performance of mixed task and data parallelism utilizing an efficiency profile [21]. Currently, we perform online prediction-assisted scheduling based only on cost estimations. However, our approach can easily be generalized to consider efficiency in decomposing tasks by extending our online profiling and predictions to efficiency data.

5.4 Energy-aware Computing

Numerous efforts have been made to address conserving energy in desktop and mobile systems. Different frequency-voltage settings in many processor architectures provides opportunity to reduce CPU power consumption via DVFS [44, 50, 75]. In the context of multi-core processor, Bircher et al. [15] study core power adaptation and subsequent performance impact on workloads. Other work attempts to save disk energy since disks consume

a large percentage of energy on some architectures [40, 56]. The basic idea is to put disks to a low power mode when there are no disk requests. Also there have been studies on reducing energy consumption by main memory and network cards where multiple energy states of these devices can be exploited [39, 65, 64, 72, 87]. Our energy measurement is conducted on systems with energy optimization turned on and future advances in this area will make the consolidated execution mode examined in our research more efficient.

The power management in server systems such as web clusters and data centers has attracted extensive studies recently due to the high energy cost of operating such high-end, large-scale systems. Pinheiro et al. [93] proposed a power management policy that dynamically turns on/off nodes in clusters. Elnozahy et al. [43] combined DVFS and node on/off to reduce energy consumption in a server cluster. Chase et al. [22] used an economic approach to allocate server resources where resources cost such as energy is balanced against utility values. Chen et al. [26] further formalized the problem by considering DVFS, service-level agreement (SLA), and reboot cost. Heath et al. [55] studied energy and throughput optimization in a heterogeneous web server cluster. Raghavendra et al. [97] addressed the integrated, multi-level power management problem for data centers. Chen et al. [24] examined dynamic provisioning and load dispatching in connection servers. Our findings and analysis suggests that offloading center workload to *active* personal computers may become an attractive alternative solution/optimization, with the predicted continuous growth of PC processing power through increased hardware parallelism.

In addition, recent studies have exploited workload aggregation techniques on server systems that “pack” concurrent processes/threads into fewer nodes, so that underutilized nodes can be shut down to save power [28, 45, 85]. The aggressive volunteer computing mode that we are interested in can be viewed as one version of such workload aggregation, but extended across the cluster/server border to personal computing environments.

There has also been a number of studies on power management in virtual machine environments. Stoess et al. [111] presented dynamic mapping and migration of virtual machines in server systems to consolidate workloads and save energy. Verma et al. [119] studied power models for HPC applications and their dynamic allocation in a virtualized environment. Nathuji et al. [85] proposed soft power states in virtual machines and examined power policy coordination among them. Energy management of both host-level and guest-level for VMs has also been studied [112]. In our measurement, we report the

additional energy and performance impact of aggressive volunteer computing on top of a popular virtual machine product. Again, techniques and advances in this area will benefit future development of energy-aware volunteer computing tools.

5.5 Co-scheduling and Performance Impact

Finally, we briefly review co-scheduling and performance impact on multi-core computers. Regarding performance interaction of volunteer computing, Gupta et al. studied the resource owner perceived performance impact of resource borrowing [51]. Strickland et al. developed Governor[113], a volunteer computing tool that dynamically throttles the foreign workload according to the observed native resource usage patterns. Both projects were conducted before multi-core PCs became widely available and our measurement differs in that we focus on multi-core computers.

In addition, several studies [9, 42, 61] investigated the interaction and performance degradation when running workloads concurrently on chip multiprocessors under different schedules, which can be leveraged for aggressive volunteer computing. Again, our analysis complements existing studies by investigating the cross-workload energy and performance interaction side by side.

Chapter 6

Conclusions and Future Work

In this dissertation, we have presented three projects in an effort to benefit transparent parallel processing on multi-core computers. In these projects, we prototype and implement a transparent runtime parallelization framework for R, which has been used by the statistics department at NCSU. Further we optimize the loop partitioning and task scheduling and evaluate extensively with micro-benchmarks as well as real applications. Besides, we conduct extensive measurement to investigate energy-performance trade-off.

In the first project, we present pR, a framework that transparently parallelizes the R language for high-performance statistical computing. We illustrate that scripting languages like R possesses 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.

In the second project, we propose, design, and evaluate a novel online task decomposition and scheduling approach for transparent parallel computing. This approach collects runtime task cost information transparently and performs online static scheduling, utilizing cost estimates generated by ANN-based runtime performance prediction, as well as by loop iteration test runs. Our implementation produces a self-learning system, ASpR, which conducts end-to-end transparent parallelization and prediction-assisted task decomposition/scheduling of the popular R language. Our study experiments on automatically

generated micro-benchmark as well as a real-world application. We verify that (1) this approach achieves high prediction accuracy with a fairly small number of training data points and low runtime overhead, and (2) the prediction results subsequently bring about significant performance benefit to the transparently parallelized R script executions.

In the third project, we evaluate the potential energy/performance trade-off of a more aggressive execution model of volunteer computing, which selects more active nodes over idle nodes for scheduling foreign application tasks to achieve higher energy savings. Below we summarize our major findings and conclusions: (1) Aggressive volunteer computing brings significant energy savings compared to the current practice of running resource-intensive workloads using clusters or idle volunteer nodes; (2) Running the foreign workload within a virtual machine does provide additional (though not ideal) performance isolation and performance protection for the native workload, with little impact on the amount of energy savings; (3) Performance wise, though many personal computing tasks are quite resilient to concurrent foreign workloads, applications involving the creation of a large number of processes, as well as memory-intensive applications, appear to be quite vulnerable; (4) By throttling the intensity level of foreign workloads, the impact on native workloads' performance without sacrificing the energy gain in most cases. However, for certain native workloads the effectiveness of such techniques requires more intelligent throttling mechanism and runtime monitoring/adaption.

The research in this dissertation suggests many potential research topics for our future work.

Our current framework for transparent parallel processing works at the granularity of task level. This limits the parallel execution performance if only a small number of very expensive function calls are present. We plan to parallelize beyond the task level by decomposing tasks into finer granularity. Interprocedural analysis is required to analyze the dependence among tasks. It is also of further interest to combine our framework with existing parallelized back-end engines such as RScalLAPACK [125]. 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.

More optimization approaches can be employed to improve parallel processing performance. For example, to deal with large data set and intensive communication patterns, we will use asynchronous communication to overlap communication cost with computation.

With large and disjoint parameter space of R functions, we intend to build multiple ANN models and use feedback to adjust window sizes.

In aggressive volunteer computing, we would like to work on autonomic tools that can identify the characteristics of native workloads and decide the foreign workload scheduling (both across machines and within a node) dynamically. Another interesting direction is to investigate the ideal data and task partitioning strategies for foreign applications to achieve energy-efficient volunteer computing.

Bibliography

- [1] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High performance Fortran compilation techniques for parallelizing scientific codes. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, 1998.
- [2] G. Almasi and D. Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.
- [3] S. Altschula, W. Gisha, W. Millerb, E. Meyersc, and D. Lipmana. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3), 1990.
- [4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.
- [5] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 1994.
- [6] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 41–53, New York, NY, USA, 1989. ACM Press.
- [7] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 1995.

- [8] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 1993.
- [9] M. Banikazemi, D. Poff, and B. Abali. Pam: A novel performance/power aware meta-scheduler for multi-core systems. In *Supercomputing '08: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, 2008.
- [10] S. Bansal, P. Kumar, and K. Singh. An Improved Two-step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines. *Parallel Computing*, 2006.
- [11] D. R. Barr and T. Davidson. A Kolmogorov-Smirnov Test for Censored Samples. *Technometrics*, 1973.
- [12] K. P. Belkhale and P. Banerjee. An Approximate Algorithm for the Partitionable Independent Task Scheduling Problem. In *International Conference on Parallel Processing(ICPP)*, Vol. 1, 1990.
- [13] R. Bell, A. Malony, and S. Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 17–26, August 2003.
- [14] Bioconductor Core. *An Overview of Projects in Computing for Genomic Analysis*, 2002.
- [15] W. Lloyd Bircher and Lizy K. John. Analysis of dynamic power management on multi-core processors. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 327–338, New York, NY, USA, 2008. ACM.
- [16] BLAST:Basic Local Alignment Search Tool. <http://blast.ncbi.nlm.nih.gov/Blast.cgi>.
- [17] Linux Battery Life Toolkit. <http://www.lesswatts.org/projects/bltk/>.
- [18] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, 1994.

- [19] Berkeley Open Infrastructure for Network Computing. <http://boinc.berkeley.edu/>.
- [20] L. Carrington, N. Wolter, A. Snavely, and C. B. Lee. Applying an Automated Framework to Produce Accurate Blind Performance Predictions of Full-scale HPC Applications. In *Proceedings of the 2004 Department of Defense Users Group Conference*, 2004.
- [21] S. Chakrabarti, J. Demmel, and K. A. Yelick. Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. *J. Parallel Distrib. Comput.*, 1997.
- [22] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 103–116, New York, NY, USA, 2001. ACM.
- [23] D. Chen, J. Torrellas, and P. Yew. An efficient algorithm for the run-time parallelization of DOACROSS loops. In *Supercomputing*, 1994.
- [24] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 337–350, Berkeley, CA, USA, 2008. USENIX Association.
- [25] M. K. Chen and K. Olukotun. The jrpm system for dynamically parallelizing java programs. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, 2003.
- [26] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 303–314, New York, NY, USA, 2005. ACM.
- [27] J. Choi, J. Dongarra, R. Pozo, and D. Walker. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, 1992.

- [28] Jeonghwan Choi, Sriram Govindan, Bhuvan Urgaonkar, and Anand Sivasubramanian. Profiling, prediction, and capping of power consumption in consolidated environments. In *Proceedings of the Sixteenth Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, Sep 2008.
- [29] R. Choy and A. Edelman. Parallel MATLAB: Doing It Right. *Proceedings of the IEEE*, 93(2), 2005.
- [30] R. Choy, A. Edelman, J. R. Gilbert, V. Shah, and D. Cheng. Star-P: High productivity parallel computing. In *Proceedings of the Eighth Annual Workshop on High Performance Embedded Computing (HPEC 2004)*, 2004.
- [31] Carl Christensen, Tolu Aina, and David Stainforth. The challenge of volunteer computing with lengthy climate model simulations. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 8–15, Washington, DC, USA, 2005. IEEE Computer Society.
- [32] W. J. Conover and R. L. Iman. Rank Transformations as a Bridge Between Parametric and Nonparametric Statistics. *The American Statistician*, 1981.
- [33] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3), 1997.
- [34] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *SIGPLAN Not.*, 1993.
- [35] M. Curtis-Maury, K. Singh, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Identifying Energy-Efficient Concurrency Levels Using Machine Learning. In *Proc. of the International Workshop on Green Computing*, September 2007.
- [36] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, 2004.

- [37] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.
- [38] Luiz DeRose and Daniel A. Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing (ICPP'99)*, September 1999.
- [39] Bruno Diniz, Dorgival Guedes, Jr. Wagner Meira, and Ricardo Bianchini. Limiting the power consumption of main memory. *SIGARCH Comput. Archit. News*, 35(2):290–301, 2007.
- [40] Fred Douglass, P. Krishnan, and Brian N. Bershad. Adaptive disk spin-down policies for mobile computers. In *MLICS '95: Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, Berkeley, CA, USA, 1995. USENIX Association.
- [41] Z. Du, C. Lim, X. Li, C. Yang, Q. Zhao, and T. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, 2004.
- [42] Ali El-Moursy, R. Garg, David H. Albonesi, and Sandhya Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *IPDPS*, 2006.
- [43] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *In Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, pages 179–196, 2002.
- [44] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. *Wirel. Netw.*, 8(5):507–520, 2002.
- [45] Vincent W. Freeh, Tyler K. Bletsch, and Freeman L. Rawson III. Scaling and packing on a chip multiprocessor. In *IPDPS*, pages 1–8, 2007.

- [46] J. Garvin. Rcc: A compiler for the R language for statistical computing. Master's thesis, Rice University, Houston, TX, 2004.
- [47] F. Gomez, D. Burger, and R. Miikkulainen. A Neuroevolution Method for Dynamic Resource Allocation on a Chip Multiprocessor. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-01)*, 2001.
- [48] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, 1999.
- [49] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.
- [50] Flavius Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 46–51, New York, NY, USA, 2001. ACM.
- [51] Ashish Gupta, Bin Lin, and Peter A. Dinda. Measuring and understanding user comfort with resource borrowing. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 214–224, Washington, DC, USA, 2004. IEEE Computer Society.
- [52] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, 1998.
- [53] M. Haldar, A. Nayak, A. Kanhere, P. G. Joisha, N. Shenoy, A. N. Choudhary, and P. Banerjee. Match virtual machine: An adaptive runtime system to execute MATLAB in parallel. In *ICPP*, 2000.
- [54] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S. Liao, and M. S. Lam. Interprocedural parallelization analysis in SUIF. *ACM Trans. Program. Lang. Syst.*, 2005.
- [55] Taliver Heath, Bruno Diniz, Enrique V. Carrera, Wagner Meira Jr., and Ricardo Bianchini. Energy conservation in heterogeneous server clusters. In *PPoPP '05:*

- Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 186–195, New York, NY, USA, 2005. ACM.
- [56] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 130–142, 1996.
 - [57] K. A. Huck, A. D. Malony, and A. Morris. Design and Implementation of a Parallel Performance Data Management Framework. In *International Conference on Parallel Processing (ICPP '05)*, 2005.
 - [58] E. Ipek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.
 - [59] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An Approach to Performance Prediction for Parallel Applications. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2005)*, pages 196–205, Aug 2005.
 - [60] A. K. Jain, J. Mao, and K. M. Mohiuddin. Artificial neural networks: A tutorial. *IEEE Computer*, 1996.
 - [61] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.
 - [62] K. L. Karavanic, J. May, K. Mohror, B. Miller, K. A. Huck, R. Knapp, and B. Pugh. Integrating Database Technology with Comparison-based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool. In *Supercomputing '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (CDROM)*, 2005.
 - [63] Michael Karo and Christopher Dwan. Applying grid technologies to bioinformatics. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High*

- Performance Distributed Computing*, page 441, Washington, DC, USA, 2001. IEEE Computer Society.
- [64] Ronny Krashinsky and Hari Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 119–130, New York, NY, USA, 2002. ACM.
 - [65] Robin Kravets, Karsten Schwan, and Ken Calvert. Power-aware communication for mobile computers. In *In Proc. 6th International Workshop on Mobile Multimedia Communications*, 1999.
 - [66] Y. Kwok and I. Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *J. Parallel Distrib. Comput.*, 1999.
 - [67] Y. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 1999.
 - [68] B. Lee and J. M. Schopf. Run-Time Prediction of Parallel Applications on Shared Environments. In *CLUSTER*, 2003.
 - [69] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of Inference and Learning for Performance Modeling of Parallel Applications. In *Principles and Practices of Parallel Programming (PPOPP)*, 2007.
 - [70] J. K. Lenstra and A. H. G. R. Kan. Complexity of Scheduling under Precedence Constraints. *Operation Research*, 1978.
 - [71] K. J. Levy. Pairwise Comparisons Associated with the K Independent Sample Median Test. *The American Statistician*, 1979.
 - [72] Xiaodong Li, Zhenmin Li, Francis David, Pin Zhou, Yuanyuan Zhou, Sarita Adve, and Sanjeev Kumar. Performance directed energy management for main memory and disks. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 271–283, New York, NY, USA, 2004. ACM.

- [73] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. In *Proceedings of the 8th Intl. Conf. of Distributed Computing Systems*, pages 104–111, 1988.
- [74] X. Ma, J. Li, and N. F. Samatova. Automatic Parallelization of Scripting Languages: Toward Transparent Desktop Parallel Computing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.
- [75] Arindam Mallik, Jack Cosgrove, Robert P. Dick, Gokhan Memik, and Peter Dinda. Pictel: measuring user-perceived performance to control dynamic frequency scaling. *SIGARCH Comput. Archit. News*, 36(1):70–79, 2008.
- [76] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, and L. Johnsson. Scheduling Strategies for Mapping Application Workflows onto the Grid. In *IEEE Symposium on High Performance Distributed Computing (HPDC 2005)*, 2005.
- [77] MATLAB. <http://www.mathworks.com/>.
- [78] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns of Parallel Programming*. Addison-Wesley Professional, 2004.
- [79] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991.
- [80] P. Miller. Parallel, distributed scripting with Python. In *The Third LCI International Conference on Linux Clusters*, 2002.
- [81] T. Mitchell. *Machine Learning*. WCB/McGraw Hill, Boston, MA, 1997.
- [82] Justin Moore, Jeff Chase, Parthasarathy Ranganathan, and Ratnesh Sharma. Making scheduling ”cool”: temperature-aware workload placement in data centers. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.
- [83] M. Müller, H. Brunst, M. Jurenz, A. Knüpfer, M. Lieber, H. Mix, and W. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In

Proceedings of the Minisymposium on Scalability and Usability of HPC Programming Tools at PARCO 2007, September 2007.

- [84] NAS Parallel Benchmarks. <http://www.nas.nasa.gov/Resources/Software/npb.html>.
- [85] Ripal Nathuji and Karsten Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 265–278, New York, NY, USA, 2007. ACM.
- [86] OpenMP. <http://www.openmp.org/drupal/>.
- [87] Soyeon Park, Weihang Jiang, Yuanyuan Zhou, and Sarita Adve. Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures. *SIGMETRICS Perform. Eval. Rev.*, 35(1):169–180, 2007.
- [88] S. Pellicer, G. Chen, K.C.C. Chan, and Y. Pan. Distributed sequence alignment applications for the public computing architecture. *IEEE Transactions on NanoBioscience*, 2008.
- [89] Stephen Pellicer. Gene sequence alignment on a public computing platform. In *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 95–102, Washington, DC, USA, 2005. IEEE Computer Society.
- [90] Perfmon2 the hardware-based performance monitoring interface for Linux. <http://perfmon2.sourceforge.net/>.
- [91] G. Pfister. *In Search of Clusters*. Prentice Hall, 1997.
- [92] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVÉR: A Tool to Visualise and Analyze Parallel Code. In *Proceedings of WoTUG-18: Transputer and Occam Developments*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, April 1995.
- [93] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on Compilers and Operating Systems for Low Power*, 2001.

- [94] W. Pugh. Uniform techniques for loop optimization. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, 1991.
- [95] M. J. Quinn, A. G. Malishevsky, N. Seelam, and Y. Zhao. Preliminary results from a parallel MATLAB compiler. In *IPPS/SPDP*, 1998.
- [96] 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.
- [97] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: coordinated multi-level power management for the data center. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 48–59, New York, NY, USA, 2008. ACM.
- [98] S. Ramaswamy, S. S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. *IEEE Trans. Parallel Distrib. Syst.*, 1997.
- [99] L. Rauchwerger. Run-time parallelization: Its time has come. *Parallel Computing*, 1998.
- [100] A. Rogers and K. Pingali. Compiling for distributed memory architectures. *IEEE Trans. Parallel Distrib. Syst.*, 5(3):281–298, 1994.
- [101] A. Rossini, L. Tierney, and N. Li. Simple parallel statistical computing in R. *UW Biostatistics working paper series*, 2003.
- [102] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 1991.
- [103] N. Samatova et al. High performance statistical computing with parallel R: applications to biology and climate modelling. *Journal of Physics: Conference Series*, 2006.
- [104] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Monotya, and S. Cranford. Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis.

to appear in Special Issue of Scientific Programming on Large-Scale Programming Tools and Environments, 2008.

- [105] M. Schwartz, N. Delisle, and Vimal S. Begwani. Incremental compilation in magpie. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, 1984.
- [106] F. J. Seinstra, D. Koelma, and J. M. Geusebroek. A software architecture for user transparent parallel image processing on mimd computers. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2001)*, 2001.
- [107] SETI@home. <http://setiathome.berkeley.edu/>.
- [108] G. C. Sih and E. A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, 1993.
- [109] Adam Silberstein, Alan Gelfand, Kamesh Munagala, Gavino Puggioni, and Jun Yang. Making sense of suppressions and failures in sensor data: A bayesian approach. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.
- [110] Standard Performance Evaluation Corporation. <http://www.spec.org/>.
- [111] Jan Stoess, Christoph Klee, Stefan Domthera, and Frank Bellosa. Transparent, power-aware migration in virtualized systems. In *GI/ITG Fachgruppentreffen Betriebssysteme*, number 2007-23, pages 3–8, Karlsruhe, Germany, October 2007. Fakultät für Informatik, Universität Karlsruhe (TH). Interner Bericht.
- [112] Jan Stoess, Christian Lang, and Frank Bellosa. Energy management for hypervisor-based virtual machines. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.
- [113] Jonathan W. Strickland, Vincent W. Freeh, and Sudharshan S. Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 64–75, Washington, DC, USA, 2005. IEEE Computer Society.

- [114] taskPR: Task-Parallel R package. <http://cran.r-project.org/src/contrib/Descriptions/taskPR.html>.
- [115] M. Taufer, D. Anderson, P. Cicotti, and C. L. Brooks III. Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 1*, page 119.1, Washington, DC, USA, 2005. IEEE Computer Society.
- [116] L. Tierney. Simple network of workstations for R. <http://www.stat.uiowa.edu/~luke/R/cluster/cluster.html>.
- [117] R. Triolet, P. Feautrier, and F. Irigoin. Automatic parallelization of fortran programs in the presence of procedure calls. In *ESOP '86: Proceedings of the European Symposium on Programming*, 1986.
- [118] J. Turek, J. L. Wolf, and P. S. Yu. Approximate Algorithms Scheduling Parallelizable Tasks. In *SPAA '92: Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.
- [119] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of hpc applications. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 175–184, New York, NY, USA, 2008. ACM.
- [120] J. Wildstrom, P. Stone, E. Witchel, R. J. Mooney, and M. Dahlin. Towards Self-Configuring Hardware for Distributed Computer Systems. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, 2005.
- [121] M. Y. Wu and D. D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):330–343, 1990.
- [122] X. Wu, V. E. Taylor, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld, and I. R. Judson. Design and Development of Prophecy Performance Database for Distributed Scientific Applications. In *Proc. the 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

- [123] J. Yang, A. Khokhar, S. Sheikh, and A. Ghafoor. Estimating Execution Time for Parallel Tasks in Heterogeneous Processing (HP) Environment. In *Proceedings of the Heterogeneous Computing Workshop*, 1994.
- [124] Kathy Yelick. Multicore: Fallout of a hardware revolution. <http://newscenter.lbl.gov/wp-content/uploads/2008/07/yelick-berkeleyview-july081.pdf>.
- [125] 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.
- [126] S. Yu, M. Winslett, J. Lee, and X. Ma. Automatic and Portable Performance Modeling for Parallel I/O: A Machine-Learning Approach. *ACM SIGMETRICS Performance Evaluation Review*, 30(3):3–5, December 2002.
- [127] A. Zell and et. al. SNNS: Stuttgart neural network simulator. *User Manual, Version 4.2*, University of Stuttgart.