Machine Learning Based Online Performance Prediction for Runtime Parallelization and Task Scheduling

Jiangtian Li*, Xiaosong Ma*†, Karan Singh‡, Martin Schulz§, Bronis R. de Supinski§ and Sally A. McKee¶
*Dept. of Computer Science, North Carolina State University, Raleigh, NC 27606
Email: jli3@ncsu.edu, ma@csc.ncsu.edu
†Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831
‡Computer Systems Laboratory, Cornell University, Ithaca, NY 14850
Email: karan@csl.cornell.edu
§Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550
Email: {schulzm, bronis}@llnl.gov
¶Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden
Email: mckee@chalmers.se

***Abstract*—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.**

In this paper, we approach this fundamental problem using machine learning techniques first to generate performance models for all tasks and then applying those models 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 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.

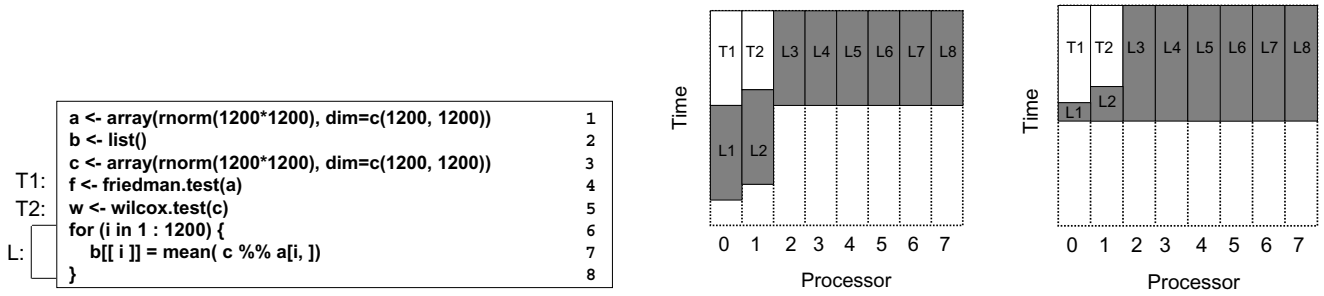
***Index Terms*—Performance Prediction, Artificial Neural Networks, Automatic Task Scheduling, Scripting Languages**

I. Introduction

Parallelization of all application types is critical with the trend towards an exponentially increasing number of cores per chip. Although active research on new programming models [1], [2] may simplify explicit parallelization of many applications, transparent or semi-transparent parallelization assisted by hardware, compilers, or runtime systems would significantly improve the applicability of many-core systems to existing sequential programs and better enable parallel commercial applications [3], [4]. However, automatic parallelization remains an elusive goal, despite much work to parallelize general-purpose programming languages transparently [4], [5], [6]. Alternatively, automatic parallelization of scripting languages [7], [8] would alleviate the application developers' parallel programming burden while avoiding many of the difficulties of parallelizing compilers. This approach, which covers not only scientific applications but also those traditionally run on commercial servers, personal computers or even cell phones, must transparently exploit both *task parallelism* and *data parallelism*.

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. Figure 1(a) shows a target scenario, a data processing script written in R [9], 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 multiplications and calculates mean values. Since L, T1, and T2 are mutually independent, they can 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 1(b) 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 1(c) shows a much more desirable schedule that optimizes the loop decomposition in light of the tasks to provide perfect load balance, but requires

978-1-4244-4184-6/09/$25.00 ©2009 IEEE



(a) A sample R script (b) Imbalanced schedule (c) Balanced schedule

Figure 1. A motivating example with sample schedules

an accurate prediction of all tasks' execution times before scheduling L.

We could apply many existing static task scheduling techniques [10] 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 use transparent performance models to predict those costs. Our models use performance observations to predict later executions, which particularly suits programs written in scripting languages. Although such languages provide a large number of functions and utilities, *on a given system* programs typically utilize a small set of functions (often with relatively stable parameter ranges). This limited set of functions are used repeatedly both within a single job and across jobs, typically by one or several users. Thus, by enabling "personal" or "per-system" performance data collection and modeling, we can archive enough samples for accurate predictions given fairly predictable user behaviors.

This paper presents 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 [8], 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 – pronounced "aspire").

For the example in Figure 1(a), ASpR 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 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 the Modified Critical Path (MCP) scheduling algorithm [11], to generate an informed loop partitioning and scheduling plan that is close to the one shown in Figure 1(c). The major contributions of this paper are:

- Lightweight online profiling and performance prediction techniques for a widely used scripting language;
- A fully transparent self-adaptive, platform- and application-independent parallelization mechanism for R;
- An extension of the MCP algorithm [11] to use cost estimates from our models for loop partitioning and task scheduling, and runtime heuristics for more effective processor allocation and scheduling;
- The first study (to the best of our knowledge) to apply machine learning to *online task partitioning and scheduling*, especially with *a priori* unknown parameter spaces;
- A detailed evaluation of our implicit parallelization approach, in terms of the overall performance of automatically parallelized scripts, using both a real-world R application and synthetic benchmarks composed of popular functions sampled from the R internal library and the BioConductor project [12].

Our results using up to 32 processors demonstrate that machine learning based performance prediction can produce schedules that run up to 40.3% faster than the baseline schedule. Accumulating more performance observations improves prediction accuracy and, thus, the resulting schedules. Further, our results demonstrate that the overall overhead of the profiling and prediction is small.

The paper is organized as follows. Section II presents the ASpR architecture. Sections III-V present our proposed approaches for ASpR: online function performance modeling and prediction, loop cost prediction test driving, and online task decomposition/scheduling using the predictions. Section VI discusses our experimental results.

II. System Architecture Overview

Although we propose a general purpose approach, we present our new Adaptively Scheduled parallel R (ASpR) framework within the context of our test platform, the pR framework [8] for transparent R script execution [9]. 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

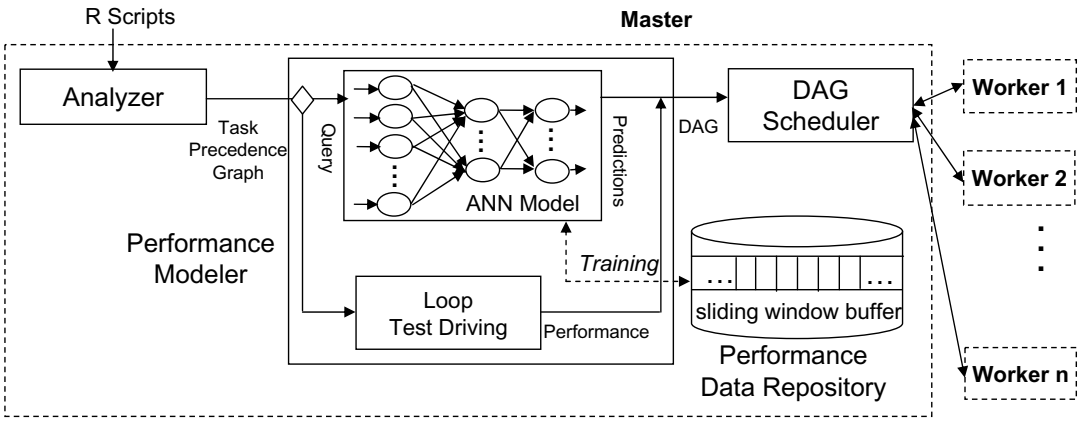


Figure 2. Adaptively Scheduled parallel R (ASpR) architecture

in this work do not require them and can be applied to systems with hierarchical parallelization without major modifications.

ASpR extends the pR master to include three new modules, as Figure 2 illustrates: (1) the *analyzer*, which performs dependence analysis, parses the R scripts and generates a Task Precedence Graph (TPG); (2) the *performance modeler*, which predicts task computation and data communication costs for the TPG; and (3) the *DAG scheduler*, a static algorithm that uses the cost predictions to determine an appropriate schedule.

ASpR predicts the performance of the two task types, function calls and blocks of independent loop iterations, differently. Generally, each user repeatedly invokes a limited set of functions, standard or user-defined. We can identify the functions by their names, although the calls to the same function may use different input parameters. Loops, on the other hand, are harder to identify, but often have stable per-iteration cost in data processing scripts. Thus, the performance modeler uses a lightweight performance repository and adopts machine-learning methods to model and to predict function costs. As we detail in Section IV, ASpR "test-drives" the first iteration of a loop to give the per-iteration loop execution cost based on actual measurement.

We collect the local script execution history into a file-based repository for function cost prediction. We retrain the performance model as we obtain new data. To reduce modeling overhead, we allocate a sliding window buffer to only store the most recent data. When applied to the master-worker paradigm, such as pR, the system overlaps retraining with script execution on the workers resulting in low overhead.

Many machine learning techniques can be used for performance prediction in a framework like ASpR. In our work we chose Artificial Neural Networks (ANNs) [13] 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 [14], to construct neural networks and perform training and prediction.

Typical ANNs have a connected, feed-forward network architecture with an *input layer*, one or more *hidden layers*, and an *output layer*. 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. During training, 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.

III. Function Performance Profiling and Model Construction

The ASpR system continuously collects cost information from function executions at runtime. Each function execution generates a training data point, which includes the function name, input parameter sizes, measured execution time, and output parameter sizes. We then use these data points to train separate ANNs for each R function. The ANN input is the data sizes of function input parameters and its outputs are the predicted function execution time and the predicted output data sizes. We include the latter since the output data size determines the data communication cost as well as the input data size for subsequent tasks depending on the predicted task.

Our implementation uses 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 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.

A self-learning runtime parallelization system transparent to script users should not require explicit training prior to use or between runs. We achieve this goal 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 must 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 experimental results (see Section VI-D). Initially, with no training data available, the window grows up to the maximum sliding window size and training is performed incrementally on all data points.

We employ a simple linear model [15] 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.

SNNS requires normalized training data to achieve good accuracy, which creates a challenge since we do not know the range of all parameters in advance. Using too wide of a range hurts prediction accuracy but it can also drop dramatically for data outside the range. 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. Thus, 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.

IV. 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 that 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 tasks, we cannot execute the initial iteration until the entire loop can be dispatched. Thus, the loop test drive approach may create a loss of concurrency that introduces a bubble into the script's 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. As we will show in Section VI, 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 VI-C).

V. Task Partitioning and Scheduling

We extend the existing Directed Acyclic Graph (DAG) scheduling algorithm MCP (Modified Critical Path) [11] to include task and loop iteration cost predictions. Given a set of homogeneous processors and a a weighted 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 (ALAP) start time 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 find holes in the schedule and to fill them with tasks. The MCP has complexity $O(v^2 \log v)$, 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 an effective and efficient task graph scheduling algorithm [16].

MCP only accommodates non-malleable tasks, i.e., tasks that are not divisible and must be assigned as one unit. Thus, it does not handle task partitioning as part of the scheduling. In our target problem, however, loops with no inter-iteration dependence must be transparently decomposed and scheduled. Further, the decision of how to decompose a loop interacts with the DAG scheduling algorithm. To map our problem to MCP's standard input, one intuitive approach would break up the loop so that each iteration is a task, run MCP to decide

Algorithm 1 Extended MCP

Input: DAG of tasks (G)
Output: Schedule of tasks to p processors

```
/* Phase 1: Determine granularity and update DAG*/
avg_func_time ← average time of functions in G.
for each node v in G do
  if v is a loop with no dependence then
    num_iters ← number of iterations of the loop.
    iter_time ← per-iteration time from test drive.
    loop_time ← iter_time × num_iters.
    if (loop_time > avg_func_time × (p × M)) then
      /* if G has no functions or the loop dominates */
      partition v into p nodes in G and update edges.
    else if avg_func_time < loop_time × M then
      /* if 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.
    else
      /* function dominates, do not partition */
    end if
  end if
end for
/* Phase 2: Schedule */
calculate MCP on G.
/* Phase 3: Merge fine grain loop blocks */
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
```

the task mapping to processors, and then merge the loop tasks back to contiguous blocks of loop iterations in the actual task scheduling. However, this approach could lead to very large task graphs and significantly increase scheduling overhead. Including many iterations in each loop task, on the other hand, risks producing a few large tasks that cannot effectively fill available schedule holes.

To address this problem, we designed 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 dynamically determines the desired level of granularity in loop tasks fed into MCP, according to the predicted costs of non-loop tasks, loop iteration test drive results, and the number of iterations.

Algorithm 1 describes our extended MCP algorithm, which consists of three phases. 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. Recall that, as described in Section IV, 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 a loop test drive. In this case, we then update the input task DAG accordingly.

We perform test drives for independent loops and the loop decompositions depend 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, we re-decompose the loop into contiguous chunks consistent with the iteration distribution determined by the MCP algorithm.

ASpR's M parameter also bounds the number of tasks into which we will decompose the original DAG. 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 1 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.

VI. Experimental Results

We evaluate the ASpR's prediction-assisted task partitioning and scheduling performance through experiments on the opt^{64} cluster located at NCSU, which has 16 2-way SMP nodes

connected using Gigabit Ethernet. Each node has two dual-core AMD Opteron 265 processors, 2GB memory and runs Red Hat's Fedora Core 5. A single NFS server manages 750GB of shared RAID storage.

A. Sample Results of Prediction-Assisted Task Partitioning and Scheduling

We illustrate the benefits of our proposed task parallelization assisted with online performance prediction through results for the sample R code given in Section I. Figure 3 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 for loop partitioning and task scheduling, based on our online cost predictions; 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. Loop blocks (shaded boxes) are marked with a number in parentheses showing the number of iterations. 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%.

B. Comparison with Dynamic Scheduling

Although dynamic scheduling works well on independent tasks, such as a do-all loop, static scheduling uses prior knowledge to consider task dependences, data locality and global load information [17]. Thus, it better suits transparently parallelized tasks and loops, especially with a mixture of loops and function calls and possible dependences.

However, with independent tasks, one may suspect that dynamic scheduling approaches may work more efficiently. Our study indicates challenges in fine-tuning dynamic scheduling parameters to balance between the tradeoff of load balance and communication overhead for parallel scripts. Thus, static scheduling remains a sound choice even for independent tasks.

TABLE I
MCP OVERHEAD

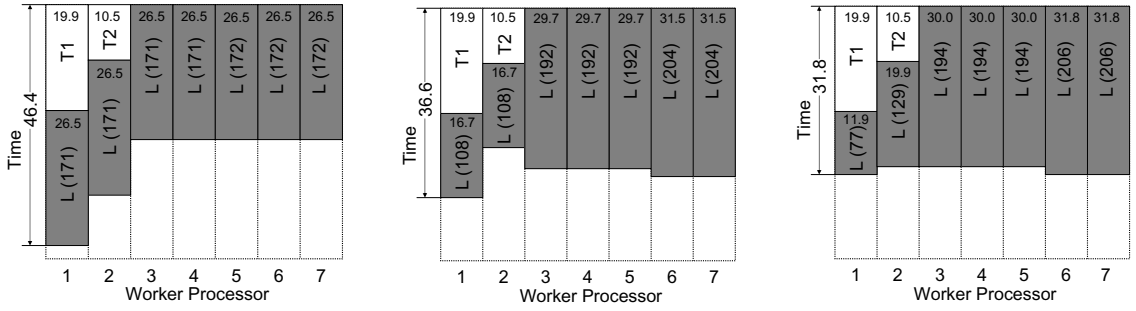
Graph size	Running time
41	0.001243
81	0.003279
161	0.004021
321	0.009392
640	0.024071
1279	0.073890
2553	0.213416
5104	0.751423
10206	3.701525
20410	14.927635

To illustrate the advantage of performance prediction based scheduling, we continue to use the sample code from Figure 1(a) to compare ASpR with a simple dynamic scheduling scheme. This approach partitions loops into smaller chunks and schedules them as independent tasks, with the master assigning, tasks to idle workers from a ready queue on request in order to fill the holes created by the function calls.

Figure 4 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 interprocess 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 5. The processing of this script is primarily a computation-intensive loop that has a 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 mentioned in Section V, the complexity of the MCP algorithm is $O(v^2 \log v)$, where v is the number of nodes in the task DAG. Table I 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 runtime 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 adjusting the parameter M to increase the minimum iteration count.



(a) Baseline (b) ASpR (ANN+MCP) (c) MCP-ideal

Figure 3. Schedules for the sample code shown in Figure 1

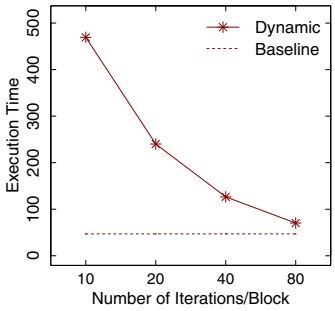


Figure 4. Data intensive script

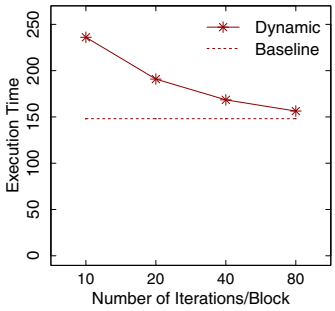


Figure 5. Computation intensive script

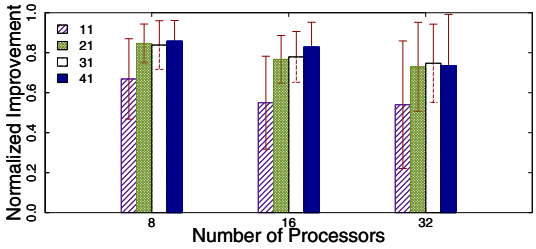


Figure 6. "Top 10" independent microbenchmarks

C. 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 micro-benchmarks with task and parameter ranges selected randomly, and discuss their performance in Section VI-D and Section VI-E.

We composed synthetic microbenchmarks from function calls and loops. In particular, 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 [12]. 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 II. 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 [18], [19], [20]. Table II 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

TABLE II
OVERVIEW OF EXPERIMENTAL SELECTED R FUNCTIONS AND PARAMETER SPACES

Functions	Description	Arguments	Parameters	Range	Running time
eigen	computes eigenvalues and eigenvectors	A	N_A	$12^2 - 2400^2$	$< 1 - 342$
prcomp	performs a principal components analysis	A	N_A	$12^2 - 2400^2$	$< 1 - 305$
qr	computes the QR decomposition	A	N_A	$25^2 - 4000^2$	$< 1 - 191$
svd	computes the singular-value decomposition	A	N_A	$12^2 - 2400^2$	$< 1 - 243$
hclust	hierarchical cluster analysis	A	N_A	$8^2 - 1600^2$	$< 1 - 180$
kmeans	performs k-means clustering	A	N_A	$20^2 - 3200^2$	$< 1 - 223$
friedman.test	performs a Friedman rank sum test	A	N_A	$16^2 - 2000^2$	$< 1 - 74$
ks.test	performs one or two sample Kolmogorov-Smirnov tests	A, B	N_A, N_B	$16^2 - 3200^2$	$< 1 - 93$
mood.test	performs Mood's two-sample test	A, B	N_A, N_B	$20^2 - 4000^2$	$< 1 - 144$
wilcox.test	performs one and two sample Wilcoxon tests	A	N_A	$16^2 - 3200^2$	$< 1 - 112$

TABLE III
OVERVIEW OF EXPERIMENTAL SELECTED R LOOPS AND NUMBER OF ITERATIONS

Loops	Description	Number of iterations	Running time
loop 1	implements boosting algorithm	640	340
loop 2	performs matrix-vector multiplication	$950 - 1150$	$95 - 160$

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.

D. 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 II. 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.

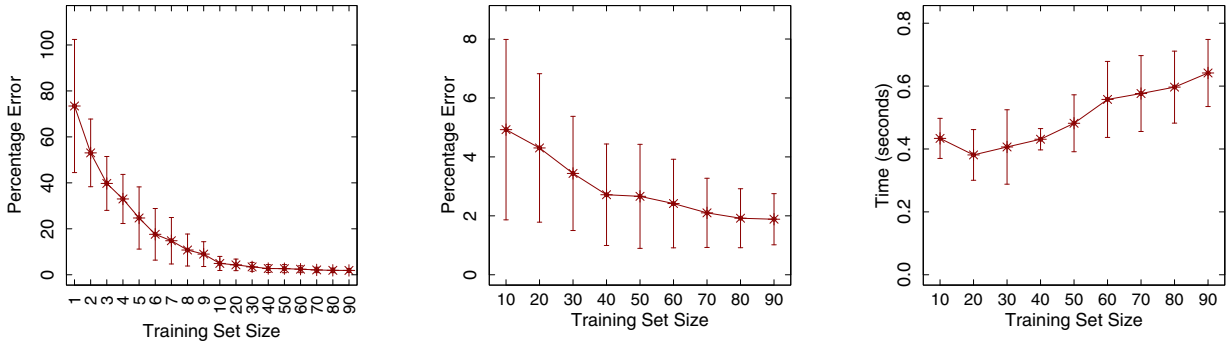
We perform incremental online training by inserting training points one at a time for each function in our experiments. We perform five queries at different intervals during the training: after inserting one new data point into the model; and after after the insertion of every 10 data points. We report 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 7(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 7(b) provides focused details of the error rate for queries executed after 10 or more training points. Using more 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 7(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 7.

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.

E. Impact on Transparently Parallelized R Code Execution

We now assess the benefit of ASpR in task decomposition and scheduling on our microbenchmarks. First we evaluate ASpR using the class of scripts with independent tasks. As in Section VI-A, 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 IV summarizes the improvement in overall execution time on 8, 16 and 32 processors for the 100 R microbenchmarks with independent tasks, where we measure ASpR's performance ("ANN+MCP") by running the



(a) Accuracy across training set size (b) Focus on larger training set sizes (c) On-line training overhead

Figure 7. Online training accuracy and overhead for various training data sizes

TABLE IV
ASPR AND INDEPENDENT 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

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 6 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 we calculate normalized improvement as:

$$(t_{baseline} - t_{ANN+MCP})/(t_{baseline} - t_{MCP-ideal})$$

This quantity is the fraction of the maximum possible 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.

Figure 6 shows that ASpR's performance generally improves 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 8 shows the parallel execution time of these top ten microbenchmarks. Due to space constraints, we only show the performance with 8 processors; we observed similar results with 16 and 32 processors. 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.

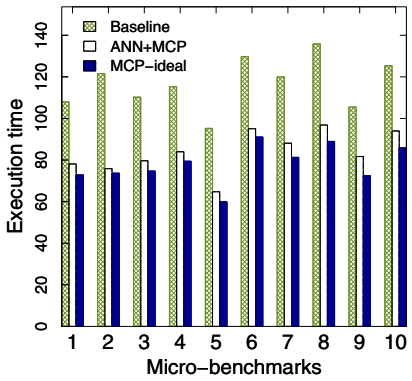


Figure 8. "Top 10" microbenchmarks (8 procs)

TABLE V
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

We evaluate the effectiveness of ASpR for inter-dependent tasks through our second class of 200 test scripts described in Section VI-C, which we categorize by the expected number of

TABLE VI
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

edges and whether they include one of our loops in Table V. 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 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 V 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 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.

F. Real-world Application Performance Results

We also assessed the ASpR's effectiveness 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 a unified framework and are widely used in modeling data with complex variance structures. Our test LMM code 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 VI, 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.

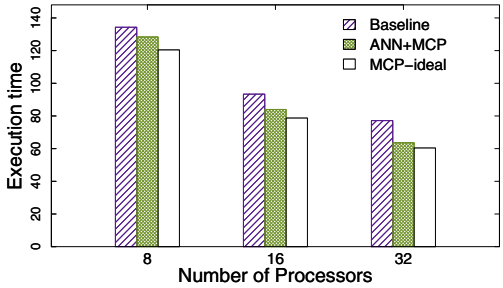


Figure 9. Real R application

Figure 9 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 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.

VII. RELATED WORK

We have covered some of the most closely related work, pR and ANNS, in Section II. In this section, we cover other related work, specifically in performance profiling and prediction, parallel job scheduling as well as self-learning and self-configuring systems. Several tools trace or analyze application performance, including Open|SpeedShop [21], TAU/ParaProf [22], Paraver [23], svPablo [24] and VampirTrace [25]. 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 [26], PerfTrack [27] and Prophesy [28], 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 lightweight 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 in 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 [29], parallel applications [30], [31] and architectural design space exploration [32]. Lee et al. [33] 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 [34]. Other work estimated applications' execution times based on parametric code profiling and analytical benchmarking techniques [35]. Perhaps most similar to our work, previous work used ANNs to guide concurrency throttling at runtime [36]. In contrast, our system adapts the ANNs, as well as using their results, at runtime and we target a self-learning system for unbounded parameter spaces.

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 [10]. DAG scheduling is quite generic and applies to many systems and applications. Since optimal DAG scheduling is an NP-complete problem [37], most research focuses on finding good heuristic solutions. Our scheduling scheme is most related to scheduling algorithms such as DLS [38] and MCP [11], 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. In this paper, we extended MCP to evaluate our ANN-based online performance prediction, and conduct optimized loop partitioning.

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 [39], using convex programming to decide how many processors to allocate to a task. Bansal et al. proposed a two-step Modified Critical Path and Area-based (MCPA) scheduling heuristic to balance processor allocation and task assignment [40]. Another study employed performance modeling in workflow scheduling on Grid resources [41]. While we share goals with these projects, we do not requires in-advance application profiling or hardware configurations.

Other research has investigated scheduling with mixed task and data parallelism, presenting approximation algorithms, both online [42] and offline [43]. 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 [44]. Currently, we perform online prediction-assisted scheduling based only on cost estimations. However, we could generalize our approach to consider efficiency in decomposing tasks.

Self-configuring, self-managing and self-learning systems have received increasing interest. Wildstrom et al. proposed a self-configuring system that adapts to the current workload in distributed computer systems [45]. They focus on varying CPU and memory resources and do not require any instrumentation of the middleware or operating system. Neural-network-based self learning was applied in dynamic resource allocation on a chip multiprocessor [46]. 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.

VIII. Conclusions and Future Work

In this paper, we proposed, designed and evaluated a novel online task decomposition and scheduling approach for transparent parallelization. Our approach collects runtime task costs transparently and performs online static scheduling, using cost estimates generated by ANN-based performance models and loop iteration test runs. Our self-learning system, ASpR, conducts end-to-end transparent parallelization and prediction-assisted task decomposition/scheduling of the popular R language. Through our study using both a real-world application and automatically generated micro-benchmarks, we verified that our approach achieves high prediction accuracy with few training data points and low runtime overhead, and that ASpR uses these prediction results to improve performance significantly for transparently parallelized R scripts. Overall, the combination of online performance prediction and runtime parallelization/scheduling is promising for developing transparent parallel computing systems for emerging many-core processors.

We plan to extend this work. For example, we will use feedback to examine runtime prediction errors and to adjust window sizes. We will also investigate using multiple ANN models for disjoint parameter spaces. Further, we will extend our approach to functions that exhibit data dependent runtimes by categorizing data across sessions.

Acknowledgments

We greatly appreciate the anonymous reviewers for their valuable comments and feedback. We are thankful to Hao (Helen) Zhang and Mihye Ahn from Department of Statistics at NCSU for providing the LMM application for our evaluation. We appreciate discussions with colleagues Nagiza Samatova, Frank Mueller, and Xiaohui Gu. We also thank Tyler Bletsch for his help and technical support with the opt^{64} cluster setup.

The research at NCSU was sponsored in part by a DOE ECPI Award (DE-FG02-05ER25685), an NSF CAREER Award (CNS-0546301), and Xiaosong Ma's joint appointment between NCSU and ORNL. Also this work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-407723) and under NSF CPA award E70-8321.

References

[1] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum, "Streamware: Programming General-purpose Multicore Processors Using Streams," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, 2008.

[2] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A Programming Model for Heterogeneous Multi-core Systems," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, 2008.

[3] D. August, "Automatic Parallelization is Key for Manycore Success," in *Manycore Computing Workshop*, 2007.

[4] W. Hwu, S. Ryoo, S. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Implicitly Parallel Programming Models for Thousand-core Microprocessors," in *Design and Automation Conference (DAC '07)*, 2007.

[5] F. Putze, P. Sanders, and J. Singler, "MCSTL: The Multi-core Standard Template Library," in *Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[6] S. Rus, M. Pennings, and L. Rauchwerger, "Sensitivity Analysis for Automatic Parallelization on Multi-cores," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, 2007.

[7] R. Choy and A. Edelman, "Parallel MATLAB: Doing It Right," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[8] 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.

[9] 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. [Online]. Available: http://www.R-project.org

[10] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Comput. Surv.*, 1999.

[11] M. Y. Wu and D. D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 3, pp. 330–343, 1990.

[12] Bioconductor Core, *An Overview of Projects in Computing for Genomic Analysis*, 2002. [Online]. Available: http://www.bioconductor.org/

[13] T. Mitchell, *Machine Learning*. Boston, MA: WCB/McGraw Hill, 1997.

[14] A. Zell and et. al., *SNNS: Stuttgart Neural Network Simulator.*, University of Stuttgart, User Manual, Version 4.2.

[15] 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.

[16] Y. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *J. Parallel Distrib. Comput.*, 1999.

[17] M. Y. Wu, W. Shu, and Y. Chen, "Runtime Parallel Incremental Scheduling of DAGs," in *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, 2000.

[18] D. R. Barr and T. Davidson, "A Kolmogorov-Smirnov Test for Censored Samples," *Technometrics*, 1973.

[19] W. J. Conover and R. L. Iman, "Rank Transformations as a Bridge Between Parametric and Nonparametric Statistics," *The American Statistician*, 1981.

[20] K. J. Levy, "Pairwise Comparisons Associated with the K Independent Sample Median Test," *The American Statistician*, 1979.

[21] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Monotya, and S. Cranford, "Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis," *Scientific Programming, Special Issue on Large-Scale Programming Tools and Environments*, vol. 16, no. 2,3, pp. 105–121, 2008.

[22] 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)*, Aug. 2003, pp. 17–26.

[23] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A Tool to Visualise and Analyze Parallel Code," in *Proceedings of WoTUG-18: Transputer and Occam Developments*, ser. Transputer and Occam Engineering, vol. 44, Apr. 1995, pp. 17–31.

[24] L. DeRose and D. A. Reed, "SvPablo: A Multi-Language Architecture-Independent Performance Analysis System," in *Proceedings of the International Conference on Parallel Processing (ICPP'99)*, Sep. 1999.

[25] 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*, Sep. 2007.

[26] 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.

[27] 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.

[28] X. Wu, V. E. Taylor, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld, and I. R. Judson, "Design and Development of Prophesy Performance Database for Distributed Scientific Applications," in *Proc. the 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[29] 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*, 2002.

[30] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An Approach to Performance Prediction for Parallel Applications," in *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2005)*, Aug 2005, pp. 196–205.

[31] 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.

[32] 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.

[33] B. Lee and J. M. Schopf, "Run-Time Prediction of Parallel Applications on Shared Environments," in *CLUSTER*, 2003.

[34] 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.

[35] 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.

[36] 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*, Sep. 2007.

[37] J. K. Lenstra and A. H. G. R. Kan, "Complexity of Scheduling under Precedence Constraints," *Operation Research*, 1978.

[38] G. C. Sih and E. A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 175–187, 1993.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] S. Chakrabarti, J. Demmel, and K. A. Yelick, "Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs," *J. Parallel Distrib. Comput.*, 1997.

[45] 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.

[46] 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.