修士論文

Performance Modeling of Task Parallel Programs
(タスク並列プログラムのパーフォマンスモデリング)

平成29年2月3日提出

指導教員　　田浦 健次朗 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-156445 ナムスライジャブ　ビャンバジャブ

Abstract

Task parallel programming makes it easy for programmers to write parallel applications by removing the burden of dealing with low-level details of thread management, task scheduling, and load balancing. Since task parallel run-time systems employ dynamic work-stealing scheduler for running an application on multiple threads, the performance modeling of a task parallel program, i.e. how the program performs as the number of cores increases or how long it executes on a different input, is hard to predict with conventional analytical models.

Regression can be used to build a performance model when formulating reliable analytical model is unfeasible. First, we run the target application multiple times to learn the execution time for differing input parameters and worker counts. Then a regression model is built using that training data.

However, past such models, mainly developed for well load-balanced applications, perform poorly when applied to task parallel applications where there is much uncertainty. Also, the accuracy of those models significantly decreases when the number of workers used in the prediction target execution becomes bigger than the number of maximum workers used during the training.

We build performance models with much better generalization than current methods, exploiting not only the execution time but also the additional information gathered during the training profilings.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

From 1985 to 2005 the performance of CPUs increased dramatically, on average 50% per year [17]. This fast growth meant that users and programmers could often simply wait for the next generation of processors to obtain increased performance from an application program. However, this growth has flattened since 2005 due to physical difficulties such as power consumption and heat dissipation. Since then, development of mainstream computer hardware has shifted from increasing the clock speed of a single-core CPU to increasing the number of cores integrated into a multi-core CPU.

Due to this trend in computer hardware, parallel programming is becoming more and more ubiquitous. However, explicitly specifying all the details of a parallel application is a complicated and tedious work for programmers. Conventional parallel programming methods, such as POSIX Threads (pthreads) [23] and the message passing interface standard (MPI) [26], require developers to deal with low-level details of thread management, load balancing, and task scheduling. This makes writing parallel programs very difficult as both programs and the hardware get more sophisticated.

Task parallel programming models are recently gaining interest to remove this burden from programmers and make writing parallel applications easier. In task parallel programming, programmers use *tasks* to express computations and their logical order. Then task scheduler is responsible for running these tasks on multiple processors dynamically.

1.2 Motivation

The dynamic nature of task parallel run-time systems makes it difficult to predict scalability behavior of task parallel applications. Therefore, performance modeling tools for task parallel applications are in high demand.

There are many uses for performance models. One use case for a performance model is that it enables to gauge the execution time of parallel applications on large many-core systems. On big systems where full hardware availability is scarce, it is important to know

execution time without running to make hardware reservations appropriately or to notice performance bottlenecks of the application before conducting an expensive execution.

Another example usage of performance models, especially for task parallel applications, is that they can be used choose optimal *configuration* parameters which will minimize the execution time of the application. Many task parallel applications have *configuration* parameters which do not affect the program's output value and correctness but significantly affects execution time of the application. Task parallelism is well-suited for divide conquer applications; task *cutoff* threshold determines when to create a new separate task or to just process the divided part inside the current task. Choosing correct cutoff values is important because if we create too many tasks, the program will suffer from excess task creation overhead. Conversely, if we do not create enough tasks, it cannot exploit enough parallelism available in the system. The number of cores to run an application is also a configuration parameter because in some cases using only 80 cores on a 100 core system may result in the shortest execution time. There are also application-specific configuration parameters. For example, the `sort` application included in Barcelona OpenMP tasks suite (BOTS) [18] takes three configuration parameters, which each define when to do following algorithm transitions: parallel mergesort $\rightarrow$ serial mergesort $\rightarrow$ serial quicksort $\rightarrow$ serial insertion sort.

Existing modeling techniques can be divided into two main types: *profiling based performance models* specifically designed for certain programming models and general-purpose *regression based performance models*. Profiling based performance models use the information gained during its profiling phase with analytical models or simulators to predict the execution time for the target number of workers. However, the prediction is only limited for inputs seen in the profiling phase. General purpose regression-based performance models, on the other hand, execute many training runs to train a regression model. These models can estimate the execution time for unseen inputs, but the estimation by currently existing models does not extrapolate well if the input size or the number of workers is beyond the training range.

Our goal is to build a regression based performance model which generalizes accurately on unseen inputs and extrapolates reasonably.

1.3 Organization of The Thesis

Chapter 2 introduces the basics of task parallel programming with its directed acyclic graph (DAG) model, a widely used analytical modeling method of task parallel programs. This chapter also presents several representative task parallel programming implementations with examples. In Chapter 3, we introduce several theoretical formulations in which the performance modeling problem can be seen as. Chapter 4 introduces the major performance modeling methods. The past works are divided into profiling based techniques (Section 4.1) and regression based techniques (Section 4.2). several representative works about performance modeling of task parallel programs as well as other parallel programming models in general.

We propose our direct regression based performance model in Chapter 5. In Chapter 6,

we present two different performance models based on profiling assisted regression. They employ two-step linear regression and multi-task neural networks regression respectively. These two profiling assisted regression methods exploit some additional information which we can gather during the training, such as the number of tasks created, *work*, and the task-stealing overhead, to improve generalization.

Chapter 7 discusses the results of our performance modeling techniques and compare the results with some past works. The paper finishes with the summary and introduction of future works in Chapter 8.

Chapter 2

Task Parallel Programming Models

Conventional parallel programming languages and frameworks such as the message passing interface standard (MPI) [26] and Pthreads [43] provide programming models based on the *direct* abstraction of hardware. In these programming models, programmers need to explicitly specify when and where each computation should be executed and how the computations communicate, load balance, and synchronize with each other. This puts high burden on programmers and decreases the productivity heavily. The burden is becoming heavier as machines become more hierarchical and more sophisticated.

Task parallel programming model is recently proposed to address these issues. In task parallel programming, programmers ubiquitously use *tasks* to express logical tasks and their order. Tasks can be nested and created at arbitrary points of execution. Then it should be left to the runtime environment, or the scheduler, to decide how to actually divide the tasks between underlying hardware processors.

2.1 Distinguishing Characteristics

Less Burden on Programmers

Since programmers ubiquitously use tasks to express computation without any hardware specific synchronization and hardware allocation, it is much easier for programmers compared to the conventional programming models.

Better Performance

Some conventional programming models use operating system (OS) threads. Due to their relatively expensive context switching and synchronization mechanisms, their overhead is very large and efficiently leveraging a massive degree of parallelism with these solutions may be difficult. However, task parallel programming implementations use dynamic scheduling and lightweight user-level threads, offering efficient context switching and synchronization operations. The overhead of user-level threads is much smaller than the OS-level threads because the thread management does not require system calls.

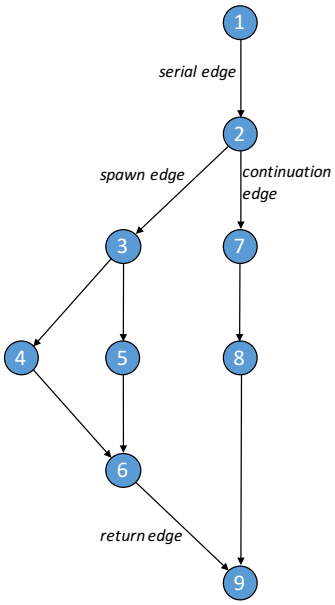


Figure 2.1: A DAG Representation of Multithreaded Application

2.2 The Directed Acyclic Graph (DAG) Model

The directed acyclic graph (DAG) model [10] for multithreading provides a general and precise quantification of parallelism. The DAG model views the execution of a multithreaded program as a graph of vertices called *strands*, sequences of serially executed instructions containing no parallel control, with graph edges indicating ordering dependencies between strands. An example of DAG is shown in Figure 2.1. This DAG contains 2 spawns and 9 strands in total. The dependencies between strands are represented by graph edges. If a strand x must complete before a strand y can begin, we say that x *precedes* y and write $x \prec y$. If neither $x \prec y$ nor $y \prec x$, we say that x and y are *parallel* and write $x \parallel y$. For example, $2 \prec 3,\ 2 \prec 7 \prec 8,\ 3 \parallel 7$, and $3 \parallel 8$ in Figure 2.1.

The DAG model provides two measures for describing program's parallelism quantitatively. These two measures are explained in the following sections.

2.2.1 The Work Law

The first measure for describing parallelism is *work*, which is the total time spent in all the strands. The execution time of the program on P processors is usually denoted as T_p. The work is equal to the execution time on one processor, thus, we denote it as T_1. The work for the example DAG in Figure 2.1 is 9 if we assume that all strands can be executed in unit time.

In a simple theoretical model of *Work Law*, P processors can execute at most P

instructions at a time. Thus, the following inequality holds for T_p.

$$T_p \geq T_1/P \tag{2.1}$$

The above inequality provides a lower bound for the parallel execution time on P processors T_P and is called *work law*.

Generally, the ratio T_1/T_p is called the *speedup* of a program. Work law implies that the speedup never exceeds P. When $T_1/T_p = P$, it is called *linear speedup*. Although it is rare, if linear speedup is achieved the program is *perfectly scalable*. Sometimes the speedup becomes bigger than P due to some practical factors, such as caching, which is not accounted in the work law. It is called *super-linear speedup*.

2.2.2 The Span Law

The other measure *span* is the maximum time to execute along any path in the DAG. With the simplified assumption that it takes exactly unit time to execute a strand, the span of the DAG in Figure 2.1 is 6. It corresponds to the paths $1 \prec 2 \prec 3 \prec 5 \prec 6 \prec 9$ or $1 \prec 2 \prec 3 \prec 4 \prec 6 \prec 9$. This path is also called *critical path* of the DAG.

Span is usually written as T_∞ because it is the fastest possible time the DAG could be executed on a machine with an infinite number of processors. Obviously, a finite number of processors can not perform better than an infinite number of processors. Thus, span provides the following lower bound for the P-processor execution time.

$$T_p \geq T_\infty \tag{2.2}$$

2.2.3 Parallelism

The ratio of work to span, T_1/T_∞, is called *parallelism*. For example, the parallelism of the DAG in Figure 2.1 is $9/6 = 1.5$. Using the work and laws we can conclude that perfect linear speedup cannot be achieved if the number of processors P is greater than the parallelism T_1/T_∞. Intuitively it can be explained that if the parallelism is less than P, there won't be enough work to keep the P processors busy all the time.

2.2.4 DAG Recorder

DAG Recorder is a profiler included in MassiveThreads [41] which records the directed acyclic graph of a program execution with timestamps and worker *ids* at each computation node (strand). In order to capture the DAG structure, DAG Recorder instruments measurement code at start and end of a task. Then it builds the directed acyclic graph of the entire computation with timestamps and worker ids at each computation node (strand). The graph recorded by DAG Recorder can be visualized with DAGViz tool [28].

Along with the DAG file, DAG Recorder also generates a stats text file that summarizes various pieces of aggregate information from the execution. The following list explains some of the aggregate metrics.

- `create_task`: The number of times tasks are created, not including the main task.
- `wait_tasks`: The number of times wait operations are issued. Each wait may wait for multiple tasks, so this number may not match `create_task`.
- `work (T1)`: The cumulative time (clock cycles) spent in executing the application code. Total across all cores. This does not include time spent in the runtime system (e.g., task creation overhead). If the application perfectly scales, this number should be constant no matter how many cores you used for execution. It is same as the *work* introduced in Section 2.2.1.
- `delay`: The cumulative time available tasks are not executed despite there are "spare" cores not executing any task. This value would be zero under a hypothetical greedy scheduler, a scheduler which immediately dispatches any available task to if any available core, without any delay.
- `no_work`: The cumulative time cores spent without available tasks.
- `critical_path (T_inf)`: Critical path of the DAG, i.e, the longest path in the DAG.

2.3 Implementations

There are many languages, framework, and libraries which supports lightweight threads task parallelism, such as Cilk Plus [1], Intel threading building blocks (TBB) [46], Java fork/join framework [35], Nanos++ [2], Qthreads [55], and Argobots [48].

In this section, OpenMP and Intel Cilk Plus, two of the most famous task parallel programming models, are introduced. Each programming model is introduced with an example implementation of quicksort in that model; thus it may be useful to look at the corresponding implementations to see differences between the programming models.

2.3.1 OpenMP Tasks

OpenMP (Open Multi-Processing) is an application programming interface (API) for shared memory multithreaded programming in C, C++, and Fortran languages. It supports most major platforms, operating systems, and processor architectures. OpenMP consists of three main components: a set of compiler directives for specifying parallel behavior in an application's code, run-time library routines, and environment variables for altering the execution features of an application. It is standardized by a joint committee of most major computer hardware and software vendors, such as Intel, ARM, AMD, and IBM. Therefore, most C/C++ and Fortran compilers support OpenMP preprocessing directives, including GNU C compiler (GCC) and Intel C++ and Fortran compiler (icc and ifort).

An OpenMP implementation example of quicksort algorithm is shown in List 2.1.

OpenMP specific directives starts with keyword `#pragma`. The `#pragma omp parallel for` directive on line 23 spawns a group of threads equal to the number of cores in the underlying system, or a specified value if the `OMP_NUM_THREADS` environment variables is set, then divides the following loop iterations between the spawned threads. There

are two main types for how loop iterations are divided into threads: static and dynamic scheduling. The static scheduling divides the loop into equal-sized groups and each thread executes a single group. The dynamic scheduling uses an internal work queue to give loop iterations to threads dynamically [3]. Here, other details of the scheduling policies are left out for simplicity. Again `#pragma omp parallel` directive creates a group of threads equal to the number of cores in the underlying system. Following `#pragma omp single` directive suppresses these threads so that only one thread executes the following statement. The suppressed threads will be used when a task is spawned from the currently active thread.

The `#pragma omp task` directive on line 13 spawns its next statement, i.e. line 14, as a separate task, so that it may be executed by a different thread. On the other hand, the task on line 15 is executed on the current thread. Then the `#pragma omp taskwait` directive on line 16 synchronizes these two tasks. These two new task parallel directives, `#pragma omp task` for task creation and `#pragma omp taskwait` for synchronization of created child tasks, are supported since OpenMP version 3.0 [5].

List 2.1: Parallel Quicksort Implementation in OpenMP [5]

```
1  #include <algorithm>
2  #include <iterator>
3  #include <functional>
4  #include <math.h>
5  #include <omp.h>
6
7  using namespace std;
8
9  template <typename T>
10 void quicksort(T begin, T end) {
11   if (begin != end) {
12     T middle = partition(begin, end, bind2nd(less<typename iterator_traits<T>::
           value_type>(), *begin));
13 #pragma omp task
14     { quicksort(begin, middle); }
15     quicksort(max(begin + 1, middle), end);
16 #pragma omp taskwait
17   }
18 }
19
20 int main() {
21   int n = 100;
22   double a[n];
23 #pragma omp parallel for
24   for (int i = 0; i < n; i++) {
25     a[i] = cos((double) i);
26   }
27
28 #pragma omp parallel
29 #pragma omp single
30   quicksort(a, a + n);
31 }
```

2.3.2 Cilk Plus

Intel Cilk Plus [1] is an extension to the C and C++ languages which supports the task parallel programming model. The Cilk Plus extension is supported through Intel C++ compiler and GCC with Cilk Plus extensions.

A Cilk Plus implementation example of quicksort algorithm is shown in List 2.2. Cilk Plus extension keywords are `cilk_for`, `cilk_spawn`, and `cilk_sync`. Unlike OpenMP, programmers do not need to declare a parallel region creation at the beginning of their Cilk Plus application. The special clauses `cilk_for`, `cilk_spawn`, and `cilk_sync` are similar to OpenMP's `#pragma omp parallel for`, `#pragma omp task`, and `#pragma omp taskwait` respectively. The `cilk_for` is implemented by calling `cilk_spawn` in divide-and-conquer pattern.

List 2.2: Parallel Quicksort Implementation in Cilk Plus [1]

```
1  #include <algorithm>
2  #include <iterator>
3  #include <functional>
4
5  #include <math.h>
6
7  using namespace std;
8
9  template <typename T>
10 void quicksort(T begin, T end) {
11   if (begin != end) {
12     T middle = partition(begin, end, bind2nd(less<typename iterator_traits<T>::
           value_type>(), *begin));
13     cilk_spawn quicksort(begin, middle);
14     quicksort(max(begin +1, middle), end);
15     cilk_sync;
16   }
17 }
18
19 int cilk_main() {
20   int n = 100;
21   double a[n];
22   cilk_for (int i = 0; i < n; i++) {
23     a[i] = cos((double) i);
24   }
25
26   quicksort(a, a + n);
27 }
```

Chapter 3

Problem Formulation

The ultimate goal of performance modeling is to be able to estimate the required execution time for the application for any given load (input parameters) and hardware (number of cores) combination. We can fulfill this goal using many We can interpret this performance model in many different ways. In this section, we introduce several such interpretations. It should be noted that the problems formulated are *not mutually exclusive*.

3.1 Regression

We first measure several different execution times of the target program on varying number of processors and varying inputs. Using these measurement results as a training data, we build a *regression model* which can estimate the execution time for the application for any unseen input parameters and number of processors combination. This enables us to interpret performance modeling as a **regression** problem.

Predicting real-valued attributes is called *regression* in the statistical literature, and it is researched widely in both machine learning and statistics. Other alternative names used for regression are functional prediction, real value prediction, continuous class learning, and function approximation.

3.1.1 Linear Regression

One of the most widely used models for regression is *linear regression*. This assumes that the response variable is a linear function of the inputs as in the following equation.

$$y(X) = \sum_{j=0}^{D} w_j x_j + \epsilon \tag{3.1}$$

where X is the input features vector, $w = (w_0, \ldots, w_p)$ is the *weight vector*, and ϵ is the *residual error* between linear predictions and the true response. We usually set $x_0 = 1$ to

create a constant intercept term. Then the goal of linear regression is to find the weights that best *fits* the given training data.

The simplest linear regression technique (ordinary least squares) finds weights that minimize the mean squared error (MSE) between the predicted and actual values:

$$\min_w \|Xw - y\|_2^2 \tag{3.2}$$

Whereas, more advanced techniques like Ridge (L2-regularized) regression and Lasso (L1-regularized) regression minimizes following values respectively.

$$\min_w \|Xw - y\|_2^2 + \alpha\|w\|_2^2 \tag{3.3}$$

$$\min_w \frac{1}{2n_{samples}}\|Xw - y\|_2^2 + \alpha\|w\|_1 \tag{3.4}$$

These techniques impose a penalty on the size weights to prevent from *overfitting*. Here, $\alpha \geq 0$ is a parameter that controls the amount of regularization.

Linear Regression With Nonlinear Terms

With linear regression, we can only model a linear relationship between the input features X and the response y. The simplest way to create a nonlinear relationship between the them is to create extra predictor variables that are transforms of some x. For example, if we add a predictor that is the square of x_1 we get the following model.

$$y = w_0 + w_1 x_1 + w_2 {x_1}^2 \tag{3.5}$$

Now, if just substitute $x_2 = {x_1}^2$ we get following *usual* linear model.

$$y = w_0 + w_1 x_1 + w_2 {x_1}^2 = w_0 + w_1 x_1 + w_2 x_2 \tag{3.6}$$

Therefore, if know the relationship formula between the input features X and the response y, we can fit the model using linear regression.

3.1.2 Nonparametric Regression

Linear regression is simple to fit; its results are easy to understand, methods for inference are well established, However, if there is an unknown nonlinear relationship between the response and one or more input features, linear regression fails to account for this important feature of the data. In such cases, *nonparametric regression* methods may be more appropriate. Unlike to parametric regression techniques like linear regression, nonparametric regression do not impose any strong parametric assumptions on the model, hence the name. Instead nonparametric methods "let the data speak for itself". However, the definitions of *parametric* and *nonparametric* are a bit ambiguous and there is no

precise and universally acceptable definition of the distinction between two terms [52]. It is common to classify some of the techniques introduced here as parametric model.

Nonparametric regression techniques are theoretically more complex than the usual linear and nonlinear parametric modeling methods. Also, they are computationally more rigorous and require a large amount of data sets. Thus, nonparametric regression methods are not used widely in practice.

There are many nonparametric regression techniques in literature, such as, kernel smoothing, Gaussian process regression, and neural network regression.

In this section

k-Nearest-Neighbors Regression

k-nearest-neighbors regression uses an average of the k nearest neighbors (k is an fixed integer), sometimes weighted by the inverse of their distance. If we have training data set of pairs (X_i, y_i), the prediction can be written as follows:

$$\hat{y}(X) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(X)} y_i, \tag{3.7}$$

where, $\mathcal{N}_k(X)$ is the set of indices of $X_1, \ldots, X_n$ which are closest to X.

While this is not a bad estimator and widely used in practice because of its simplicity, it has a few limitations. First, the regression output always has sudden steps because of the discontinuity. Also, the definition of k-nearest-neighbors is vague when the input is multi-dimensional and its sensitive to input normalization.

Kernel Smoothing Regression

Kernel smoothing regression generalizes k-nearest-neighbors using a *kernel covariance* function. Kernel covariance function is a function satisfying following conditions.

$$\int K(x)dx = 1, \int xK(x)dx = 0, 0 < \int x^2 K(x)dx < \infty \tag{3.8}$$

Then given a bandwidth $h > 0$, the response is estimated as follows using Nadaraya-Watson kernel regression [40], [54].

$$y(x) = \frac{\sum_{i=1}^{n} K\left(\frac{x-x'}{h}\right) y_i}{\sum_{i=1}^{n} K\left(\frac{x-x'}{h}\right)} \tag{3.9}$$

The kernel regression usually suffers from poor bias at the boundaries of the domain of the inputs $x_1, \ldots, x_n$.

Some popular kernel functions are as follows. The de-facto radial basis function (RBF) kernel:

$$K_{RBF}(x, x') = \sigma^2 \exp(-\frac{(x - x')^2}{2l^2}), \tag{3.10}$$

where σ^2 is the output variance and l is called *lengthscale*.

d-degree polynomial kernel,

$$K_{poly}(x, x') = (x^T x' + c)^d \tag{3.11}$$

where $c \geq 0$ is a parameter specifying the ratio of influence of higher-order versus lower-order terms. This kernel is called *linear kernel* when $d = 1$.

Rational quadratic (RQ) kernel which is equivalent to adding together many RBF kernels with different lengthscales.

$$K_{RQ}(x, x') = \sigma^2(1 + \frac{(x - x')^2}{2\alpha l^2})^{-\alpha}, \tag{3.12}$$

where α is the determines the relative weighting parameter.

Periodic kernels shown below allows to models periodic functions.

$$K_{per}(x, x') = \sigma^2 \exp(-\frac{2\sin^2(\pi|x - x'|/p)}{l^2}), \tag{3.13}$$

where p is the parameter specifying the distance between repetitions and l is the lengthscale.

Epanechnikov kernel [20]:

$$K_{Epanechnikov}(x, x') = \begin{cases} 3/4(1 - (x - x')^2) & \text{if} |x| \leq 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.14}$$

Sigmoid kernel:

$$K_{sig}(x, x') = \tanh(x^T x' + c) \tag{3.15}$$

It is common to combine these kernels to get a kernel more suitable for the problem at hand.

Support Vector Machine Regression

Support vector machines (SVM) are a set of famous supervised learning methods widely used for classification and regression. Support vector machines result from minimizing the hinge loss $(1 - y_i w \cdot x_i)_+$ with ridge regularization.

$$\min_w \sum_i (1 - y_i w \cdot x_i)_+ + \lambda ||w||^2. \tag{3.16}$$

The intuition behind this is to find a function that has at most ϵ (predefined constant) deviation from the actually obtained targets y_i for all the training data, at the same time using the simplest form of function as possible. There have been many methods for finding the optimal functions for SVM efficiently.

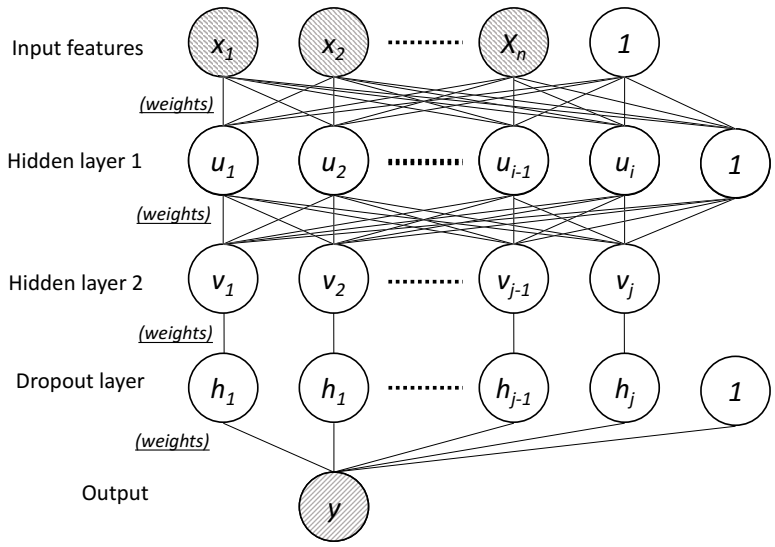


Figure 3.1: Neural Network Regressor With Two Hidden Layers

Gaussian Process Regression

The Gaussian process is also a famous nonparametric kernel based learning method, which can be used in regression [57, 58, 56, 47].

Neural Networks Regression

Neural network regression is technically not a nonparametric method since the number of parameters in a neural net does not change with the number of sample data. However, due to the fact that neural networks with nonlinear activation functions are universal approximators [16, 51], it is common to label them as nonparametric. Although neural nets are mostly used in classification problems, they can be used in regression too (Figure 3.1).

Recently, it became preferred to use neural networks with more than one hidden layers. For example, Delalleau and Bengio [7] showed that a shallow network requires exponentially many more sum-product hidden units than a deep sum-product network in order to compute certain families of polynomials.

3.2 Domain Adaptation

In most cases, we do not want to conduct multiple training executions on p cores to predict the execution time on just same p cores. Alternatively, we also do not want to measure execution times with input loads as large as that of the target prediction. Because that will take longer than just running the target execution.

Therefore, the ideal prediction model must be capable of *extrapolating* outside its training range. In most cases, machine learning regression models assume independent

and identically distributed (i.i.d.) which in our case means that the training and test input distribution should be same. This causes the regression models to be poor at extrapolating.

3.3 Multi-task Learning

Multi-task learning [12] is a branch of transfer learning [44] whose main goal is to improve generalization performance by leveraging the domain-specific information contained in the training signals of related tasks. It does this by training tasks in parallel with a shared representation. The training signals for the extra tasks serve as a *regularization*. The effectiveness of multi-task learning is proved in many literatures [12].

In our regression approach, during the training executions, we can record metrics other than plain execution time using profiling tools. Then by utilizing these other metrics as extra tasks, we formulate our performance prediction as a multi-task learning problem.

Multi-task learning approaches assume that individual models for related tasks should share some parameters or prior distributions of hyperparameters. Since multi-task learning tries to learn both the main and extra tasks simultaneously, weights of the loss functions for the source and target data are usually the same. In some instances, we can generalize it to give more weight to the loss on the main task.

Multi-task learning approaches are applied in many regression frameworks as listed below.

Multi-task Learning in Gaussian Processes

Bonilla et al. [11] investigated multi-task learning in the context of Gaussian Processes (GP). It uses a covariance matrix over tasks to model inter-task dependencies, where a GP prior is used to induce correlations between tasks. Lawrence et al. [34] also proposed an algorithm known as multi-task informative vector machine (MT-IVM) based on GP, to handle the multi-task learning. MT-IVM tries to learn parameters of a Gaussian Process over multiple tasks by sharing the same GP prior. Schwaighofer et al. [47] proposed a model which learns GP kernels for multi-task learning using the hierarchical Bayesian framework (HB).

Multi-task Learning in Support Vector Machines

Evgeniou et al. [21] proposed a multi-task learning method for SVMs which assumed that each task could be separated into two terms, one is a common term for all tasks and the other is a task-specific term.

Multi-task Learning in Neural Networks

Multi-task learning is used in neural networks as a way of improving generalization by pooling the examples arising out of multiple tasks. Caruana et al. [12] showed that

applying multi-task learning method in real world regression problems increases the generalization ability. Collobert et al. [14] proposed a multi-task convolutional neural network architecture that, given a sentence, outputs multiple language processing related estimates, such as part-of-speech tags, chunks, named entity tags, semantic roles, and semantically similar words. Li et al. [37] proposed a technique for estimating human pose from an image by with deep convolutional neural network with multi-task outputs. Zhang et al. [61] used a similar method to estimate head pose and facial attributes.

Chapter 4

Related Works

4.1 Profiling Based Performance Modeling

4.1.1 Profiling Tools

Profiling based performance analysis tools utilize runtime information which is collected during the execution. In this subsection, we introduce several profiling tools which are used by most of the performance analysis tools. Profiling methods can be divided into two groups based on whether they are using *sampling* or *instrumentation*.

The sampling profiling methods collect statistical data during a profiling execution of the application. Sampling has little overhead and usually does not require modifying the source code. The instrumentation profiling methods collect detailed timing for the function calls in the target application by injecting code into the binary file that captures timing information for each function call.

Gprof [25] is a popular open-source performance analysis tool, which uses a hybrid of instrumentation and sampling. It helps programmers investigate which parts of the program is the bottleneck and needs optimization. The common usage of gprof is as follows.

1. Compile and link the target program with profiling enabled.
2. Execute the program to generate a profile data log.
3. Run gprof the analyze the profile data.

There are also many other performance analysis tools which use gprof as their backend. Kremlin [24] is a tool to identify program regions where parallelization is necessary, in the same way gprof identifies regions requiring optimization in a serial program.

Pin [38] is a proprietary dynamic binary instrumentation tool created by Intel. It can be used by application developers to insert instrumentation into any function call in their application. Valgrind [42] is also a popular dynamic binary instrumentation tool which can be extended build performance analysis tools.

The Performance API (*PAPI*) is a standard application programming interface (API) for accessing hardware performance counters on various hardware with a common interface. It provides easy to use abstraction from underlying particular hardware events.

4.1.2 Cilkview

Cilkview [27] is a scalability analyzer tool for multithreaded applications. Specifically, the input to Cilkview is restricted to parallel programs written in Cilk Plus. It then provides a lower and upper bound estimation of how the program ' s performance will change as the number of cores increase. The upper bound estimation of scalability is calculated using the DAG model described in Section 2.2. On the other hand, the lower bound estimation of scalability is calculated using their proposed *burdened DAG model*.

A Cilk Plus implementation of quicksort algorithm with Cilkview annotations is shown in List 4.1. Cilkview specific annotations are highlighted in red. The target region for analysis can be specified using `cilkview::start()` and `cilkview::stop()` functions as in the example code. If the region is not specified, the whole program is analyzed.

List 4.1: Parallel Quicksort Implementation in Cilk Plus with Cilkview annotations

```
1  #include <algorithm>
2  #include <iterator>
3  #include <functional>
4  #include <math.h>
5  #include <cilkview>
6
7  using namespace std;
8
9  template <typename T>
10 void quicksort(T begin, T end) {
11   if (begin != end) {
12     T middle = partition(begin, end, bind2nd(less<typename iterator_traits<T>::
           value_type>(), *begin));
13     cilk_spawn quicksort(begin, middle);
14     quicksort(max(begin +1, middle), end);
15     cilk_sync;
16   }
17 }
18
19 int cilk_main() {
20   int n = 100;
21   double a[n];
22   cilk::cilkview cv;
23   cilk_for (int i = 0; i < n; i++) {
24     a[i] = cos((double) i);
25   }
26
27   cv.start();
28   quicksort(a, a + n);
29   cv.stop();
30   cv.dump("quicksort");
31 }
```

Burdened DAG Model

The DAG model introduced in Section 2.2 does not consider practical factors such as the performance of the scheduling algorithms and the overhead of migrating tasks between threads. Cilkview tries to account these factors by introducing a new model called *burdened DAGs*, which incorporates the migration overheads.

The Cilk Plus randomized work-stealing scheduler can execute a program with T_1 work and T_∞ span on P processors in following expected time:

$$T_p \leq T_1/P + \delta T_\infty \ , \tag{4.1}$$

where δ is a constant called *span coefficient* [22]. Intuitively inequality Equation (6.2) means that, if the parallelism T_1/T_∞ exceeds the number of processors P sufficiently, the bound warrants near-perfect linear speedup. It comes from the fact that if $T_1/T_\infty \gg P$ we have $T_\infty \ll T_1/P$. Thus, from the inequality Equation (6.2), $T_1/T_p \approx P$ is derived.

The burdened DAG model calls the overhead of migration, such as the cost of setting up the context to run the migrated task and the implicit costs of cache misses due to the migration, *burden*. Then Cilkview assumes that this burden has a fixed value of around 15,000 instructions. The burdened DAG model then incorporates the burden of each continuation and return edge of the DAG into the normal DAG model. Also, the *burdened span* is defined as the longest path in the burdened DAG. Then, a work-stealing scheduler running on P processors can execute the program in expected time

$$T_p \leq T_1/P + 2\delta\hat{T}_\infty, \tag{4.2}$$

where $\hat{T}_\infty$ is the burdened span. The proof of this equation can be found in the original paper [27]. This can be further transformed to the following equation to give a lower bound on the speedup.

$$\frac{T_1}{T_p} \geq \frac{T_1}{T_1/P + 2\delta\hat{T}_\infty} \tag{4.3}$$

Cilkview employs this equation to compute an estimated lower bound on speedup. It uses $\delta = 0.85$ as the span coefficient. Thus, the final equation is $T_p \leq T_1/P + 1.7\hat{T}_\infty$.

Implementation

Cilkview collects an application's parallelism information, i.e. DAG and length of the each strand's length, during a serial execution of the application under the PIN [38] dynamic binary instrumentation framework. The length of the each strand is measured in instruction count instead of time. Since PIN is binary instrumentation tool, meaning that it operates directly on the executable binary of the application as opposed to the source code, recompilation of the application is not required.

The Cilk Plus compiler embeds metadata relevant to multithreaded execution to into the executable binary in multithreaded executable format (MEF). Cilkview uses

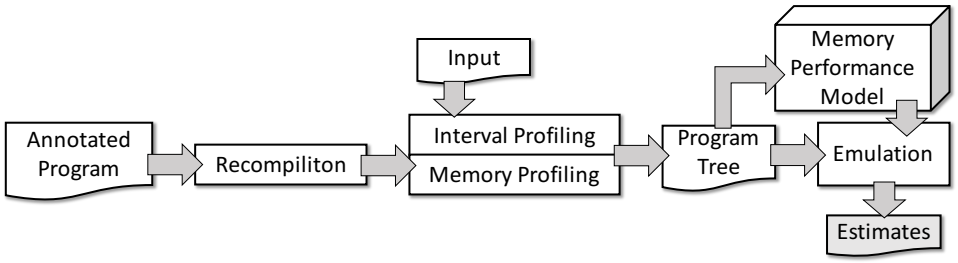


Figure 4.1: Workflow of Parallel Prophet

these metadata to know where Cilk Plus's parallel control constructs `cilk_spawn` and `cilk_sync` appear in the executable. When an application is run under Cilkview, it reads the metadata and instruments the corresponding addresses in the binary to collect measurements. Running an application under Cilkview is usually 2-10 times slower than the normal run.

Limitations

Cilkview does not consider the limitations of memory bandwidth on its calculation of program's scalability. Therefore, it shows substantial inaccuracy in predicting memory intensive applications' scalability.

4.1.3 Parallel Prophet

Parallel Prophet [31] predicts potential speedup of a serial application from information gathered from profiling and emulations. It answers the question of how much speedup could be gained if the application is parallelized. Unlike Cilkview, the input to Parallel Prophet is a serial application without any parallelization. Instead, programmers are required to insert specific annotations in Table 4.1 into the serial application to describe the application's parallelism. Another main difference with Cilkview is that it tries to account memory limitations by introducing a memory performance model.

The workflow of Parallel Prophet is illustrated in Figure 4.1. First, the annotated program is recompiled; then *interval profiling* and *memory profiling* is performed using the recompiled executable. It produces a *program tree* which contains all the necessary information for running the memory performance model and the emulators. The emulators calculate final estimates for the program's parallel speedup.

Memory Performance Model

Parallel Prophet introduces *burden factors* to model the parallel speedup slowdown due to increased memory traffic. The burden factor is calculated for each top-level parallel section of an application. Then, when estimating the application's execution time, the burden factor is multiplied to each corresponding section.

Table 4.1: Annotations in Parallel Prophet

Interface	Description
`PAR_TASK_BEGIN()`	This task may run in parallel.
`PAR_TASK_END()`	The end of the task.
`PAR_SEC_BEGIN()`	Parallel section begins.
`PAR_SEC_END()`	The end of the current section.
`LOCK_BEGIN()`	Acquire a lock.
`LOCK_END()`	Release the lock.

Parallel Prophet makes following assumptions in their memory performance model.

- Execution time of a program can be separated into two disjoint parts: computation cost and memory cost
- Work is equally divided among all threads.
- Memory system has following properties: only last-level cache (LLC) is present, the latencies of memory read and write are the same, hardware multithreading is not present, and hardware prefetchers are disabled.
- The value of LLC misses per instruction does not vary significantly between serial and parallel execution.
- Super-linear speedup is not considered.

Using these assumptions, the execution time (in cycles) of an application can be written as follows:

$$T = CPI \cdot N = CPI_{\$} \cdot N + \omega \cdot D, \tag{4.4}$$

where N is the number of all instructions, $CPI_{\$}$ is the average cycles per instruction if there are no DRAM access, D is the of DRAM accesses, and ω is the average CPU stall cycles for one DRAM access.

The burden factor β_t for a thread number t represents the performance degradation only caused by the memory performance. Thus, β_t can be expressed as the following equation:

$$\beta_t = \frac{T^t}{T_i^t} = \frac{CPI^t \cdot N^t}{CPI_i^t \cdot N_i^t} = \frac{CPI_{\$}^t \cdot N^t + W^t \cdot D^t}{CPI_{\$,i}^t \cdot N_i^t + \omega_i^t \cdot D_i^t} \tag{4.5}$$

where, the super-script ^{t} means the value is for t threaded execution and the sub-script $_i$ means the value is for the execution on *ideal* machine with infinitely large memory bandwidth.

Using the assumptions, the above equation can be further simplified to the following.

$$\beta_t = \frac{CPI_{\$} + MPI \cdot \omega^t}{CPI_{\$} + MPI \cdot \omega}. \tag{4.6}$$

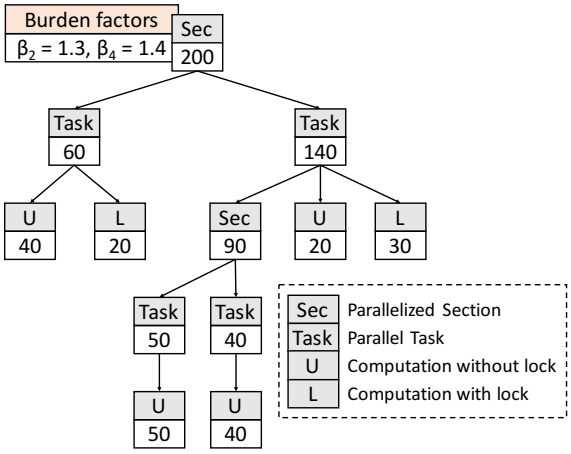


Figure 4.2: An Example of Program Tree in Parallel Prophet

Also, Parallel Prophet assumes following relationship:

$$\delta^t = \Psi(\delta),\ \omega^t = \Phi(\delta^t)\ , \tag{4.7}$$

then finds the empirical functions Ψ and Φ via a specific microbenchmark. A detailed explanation of the rationale behind this is in the original paper.

In summary, Parallel Prophet obtains N, T, D, MPI and δ values from the profiling of a serial program; estimates ω^t from equation Equation (4.7); then finally calculates the burden factor β_t from equation Equation (4.6). Here, the memory profiling uses the information obtained from low overhead hardware performance counters by PAPI [39].

Building a Program Tree

The profiling in Parallel Prophet is performed by running the application under PIN [38], same as Cilkview. In the recompilation phase, the annotations – C/C++ macros – insert trigger functions for the PIN. Then it collects the lengths of all annotation pairs in the interval profiling phase. The length of the each annotation pair is measured in instruction count same as the Cilkview analyzer.

Parallel Prophet uses this information to build a *program tree* like that is shown in Figure 4.2. Each node in the tree contains information about its length and its node type (section, task, computation without a lock, and computation with lock). Also, each node for a top-level section (the tree root) has a value set for its burden factor β.

Emulation

Parallel prophet performs *fast forwarding emulation* of a parallel execution by traversing the program tree. To emulate the execution of a parallel application accurately, it needs to consider the scheduling policies of the real parallel run-time model. Also, it means

that each run-time model needs to be separately implemented in Parallel Prophet. The paper only implements an emulator for OpenMP.

Parallel prophet also has a program synthesis based emulator. However, it is left out in this paper, because the merit of the program synthesis based emulator is not clear.

Limitations

The fast forwarding emulator produces significant errors in some applications. The primary reason for such cases is the frequent lock contention or frequent fork/join (spawn/wait) in the application. In such cases, the parallel overhead is too high, making the program tree inaccurate.

4.1.4 DraMon

DraMon [53] is high accuracy model for predicting memory bandwidth usage of multithreaded applications. Cilkview [27] and Parallel Prophet [31] predicts the speedup of a whole application or a certain annotated region in the application; on the other hand, DraMon only focuses on predicting the memory bandwidth usage. Parallel Prophet tries to build a memory performance model for multithreaded applications; however, it makes several aggressive assumptions about the model, which causes it to become inaccurate in some cases. On the other hand, DraMon builds more detailed performance model by accounting a wide range of hardware and software factors which is ignored in Parallel Prophet. They propose two versions of DraMon: *DraMon-T* which uses a memory-trace, and *DraMon-R* which utilizes run-time hardware performance counters.

DraMon only predicts the memory bandwidth usage of an application. However, since its memory performance model outperforms the previous works, it can be used to improve the speedup prediction performance of the previous works such as Parallel Prophet.

DRAM

The memory controller (MC) in a CPU is connected to several *channel*s. A channel consists of several *rank*s, i.e. a memory module. Each rank has several memory *chip*s, each consisting of several *bank*s. Each bank is a cell array where a cell stores a bit.

The operations of a DRAM read cycle is shown in Figure 4.3. When data is accessed, the row containing this data is read into the row buffer. First, the connections between the row buffer and the bank are precharged in tRP time. Then, the memory controller reads the row into the row buffer in $tRCD$ time. When the row becomes ready, MC sends the column address and locates the data in $tCAS$ time. Lastly, the data is sent to MC using $tBurst$ time. A DRAM request categorized into following types, based on the status of the target bank of the request:

- *Hit*: the row buffer has the requested row; the latency is $tCAS + tBurst$.
- *Miss*: the banks is precharged, but the requested row is not in the row buffer; the latency is $tRCD + tCAS + tBurst$.

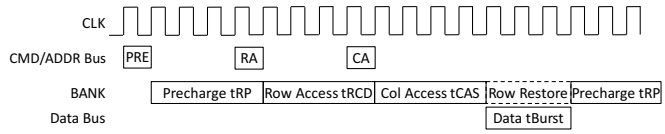


Figure 4.3: A Read Cycle of DRAM [29]

- *Conflict*: the bank is not precharged for the request: the latency is $tRP + tRCD + tCAS + tBurst$.

The Bandwidth Model

Hardware and software parameters used in the bandwidth model is listed in Table 4.2 and Table 4.3 respectively. In the DraMon paper, hardware parameters are garnered from the data sheets and PCI configuration registers of the used hardware [9]. The software parameters are measured using a memory trace in DraMon-T or performance monitoring units (PMU) in DraMon-R. Similar to the Cilkview and Parallel Prophet, it the memory traces are collected by running the application under PIN instrumentation framework.

Memory bandwidth usage can be expressed in the following self-explanatory equation:

$$BW = chnl_{cnt} \times Rate_{mem} \times Size_{mem}, \tag{4.8}$$

where $Size_{mem}$ is the size of each request and memory request rate $Rate_{mem}$ is the minimum of memory issue rate of determined by the program and the DRAM bound limited by the hardware, i.e.:

$$Rate_{mem} = min(Rate_{issue}, Rate_{dram}) \tag{4.9}$$

Also, following equations hold:

$$\begin{aligned} Rate_{dram} &= \frac{1}{Lat_{dram}}, \\ Lat_{dram} &= Ratio_r \times Lat_r + Ratio_w \times Lat_w + O_{wtr} + O_{rtr}. \end{aligned} \tag{4.10}$$

Furthermore, the average read and write latency can be calculated from following equations:

$$\begin{aligned} Lat_r = Ratio_{hit} \times Lat_{r,hit} + Ratio_{miss} \times Lat_{r,miss} + \\ Ratio_{conf} \times Lat_{r,conf} \end{aligned} \tag{4.11}$$

$$\begin{aligned} Lat_w = Ratio_{hit} \times Lat_{w,hit} + Ratio_{miss} \times Lat_{w,miss} + \\ Ratio_{conf} \times Lat_{w,conf} \end{aligned} \tag{4.12}$$

Table 4.2: Hardware Parameters in DraMon

Parameter	Description
$Size_{mem}$	size of each DRAM request
tRCD	row activation time
tCAS	column access time
tRP	precharge time
tBurst	data transfer time
tWR	write recovery time
tRTRS	rank switching time
tWTR	write-to-read switching time
chnl_cnt	number of channels
bk_cnt	number of banks per rank
D_{ac}	row buffer auto-close distance

Finally, by combining the above equations Equation (4.15), Equation (4.16), Equation (4.10), Equation (4.11), and Equation (4.12), the following equations for bandwidth usage is derived.

$$BW = chnl\_cnt \times Size_{mem} \times min(Rate_{issue}, \frac{1}{Ratio_r \times Lat_r + Ratio_w \times Lat_w + O_{wtr} + O_{rtr}}) \tag{4.13}$$

$$Lat_x = \sum_{type} Ratio_{type} \times Lat_{x,type}, \quad \text{where } x \in \{r, w\}, type \in \{hit, miss, conf\} \tag{4.14}$$

Therefore, the bandwidth usage of multithreaded execution can be calculated by substituting the values listed in Table 4.2 and Table 4.3 into the equations Equation (6.5) and Equation (6.6). The values not directly retrievable from Table 4.2 and Table 4.3, such as $Rate_{issue}, O_{wtr}, O_{rtr}, Ratio_{type}$, and $Lat_{x,type}$, can also be estimated as explained in the paper. The detailed argumentation of how these values are estimated is left out here.

4.1.5 Performance Modeling Using Replay Techniques

There are several works, which use *deterministic replay* technique to predict application performance on many cores. They run an application with target workload sequentially to collect trace. Afterwards the parallel execution is simulated by replaying the trace. These approaches take very long time because each process must be run sequentially.

Table 4.3: Software Parameters in DraMon

Parameter	Description
$Rate_{issue,single}$	single thread issue rate
Δ and P_{Δ}	bank reuse distances and their probabilities
$Ratio_{hit,single}$	single thread hit ratio
$Ratio_{miss,single}$	single thread miss ratio
$Ratio_{conf,single}$	single thread conflict ratio
P_{SmRw}	same-row accessing probability
P_{SmBk}	same-bank-diff-row accessing probability
P_{SmCh}	diff-bank-same-channel accessing probability
P_{DfCh}	different channel accessing probability
$Ratio_{wr}$	write request ratio
$Ratio_{wtr}$	write-to-read switching request ratio
$Ratio_{rtr}$	rank switching request ratio
rk_cnt	number of ranks accessed

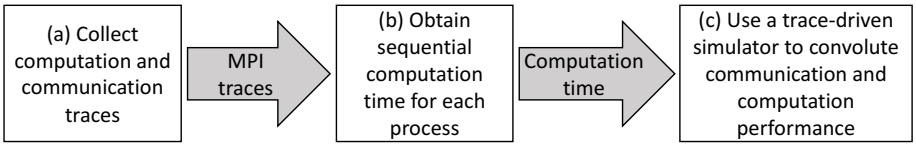


Figure 4.4: Workflow of Phantom

Phantom

Phantom [60] predicts performance on multiple cores using only one of the nodes. First, it collects computation and communication traces by running each of the processes one by one. Then, a log database is built using the recorded computation and communication traces. Lastly, a trace-driven simulator convolutes communication and computation performance on any given number of cores (Figure 4.4).

Phantom only works on MPI based applications, since the each process should be clearly separated. It uses PMPI profiling interface [26] for recording the traces.

ScalaExtrap

ScalaExtrap [59] is a method to automatically generate the application trace for large numbers of nodes by extrapolation from a set of smaller traces. Then the extrapolated trace can be replayed to assess communication performance for a larger number of cores.

4.1.6 Others

ESTIMA [13] is performance model for extrapolating scalability of in-memory applications. They use *stalled cycles* as an intermediate feature. The number of stalled cycles is modeled using a predefined function set where the function argument is the number of cores. Then builds an execution time prediction model with an assumption that the number of stalled cycles per core have a high correlation with execution time. Stalled cycles are divided into two types hardware and software stalled cycles. Examples of hardware stalled cycles include frontend (instruction cache misses) and backend (resource or data cache miss). Software stalled cycles are caused by spinning lock and software transactions

Kismet [30] is a tool to detect parallelism automatically in serial applications. It uses hierarchical critical path analysis (HCPA) technique analyze the application regionally, then detects parallelizable regions.

4.2 Regression Based Performance Modeling

Regression-based performance modeling techniques use results of many training runs of the program with statistical models to create performance models which can predict execution times for new inputs. Although, they can predict the performance for new inputs, in most cases the prediction parameter domain range is limited to that of the training phase. Therefore, to predict the performance of an application for big inputs on a large system, training measurements have to be as resource intensive as prediction target execution.

As described in detail below, Barnes et al. [6] try to use fewer cores in the training phase than the prediction target cores by modeling computation and communication separately. However, their technique is specific to the evaluated MPI applications only and not effective for task parallel applications.

4.2.1 Methods of inference and learning for performance modeling of parallel applications

[36] introduces performance modeling of parallel applications using two different techniques: piecewise polynomial regression and artificial neural networks. Their performance modeling techniques are very general and not limited to a specific programming paradigm. The applications used as evaluation in the paper are all implemented using MPI.

Lee et al. employ hierarchical clustering, association analysis, and correlation analysis to assist their piecewise polynomial regression model. Hierarchical clustering is used to find similarity between predictors which in turn used to ensure redundant predictors are not included in the model. Pruning the number of predictors also helps in controlling the number of potential interactions between predictors. Association analysis examines each predictor's association with the response. Correlation analysis quantifies the association relationship results. It helps to find predictors with higher rankings, which may require non-linear transformations. Arguing that polynomials have undesirable peaks and valleys,

their paper divides the predictor domain into *knots* with different polynomial fits. Since the piecewise polynomial regression model only models the parameter domain range of training data, it cannot predict performance for inputs and number of cores outside its training range.

Their other model, artificial neural networks, is more automatic and does not require statistical analysis and application specific configuration which were necessary for the linear model. Median error rates range from 2.2 to 9.4 percent in the linear regression model and 3.6 to 10.5 percent in the ANN model.

4.2.2 A regression-based approach to scalability prediction

Barnes et al. [6] introduced regression based technique [6] to predict the scaling behavior of parallel programs written in MPI. They model the execution time of a parallel application as

$$\begin{aligned} log_2(T) = \beta_1 log_2(x_1) + \beta_2 log_2(x_2) + \ldots \\ + \beta_n log_2(x_n) + g(q) + error \end{aligned} \tag{4.15}$$

where,

$$g(q) = \gamma_0 + \gamma_1 log_2(q) \texttt{or} g(q) = \gamma_0 + \gamma_1 log_2(q) + \gamma_2 (log_2(q))^2 \tag{4.16}$$

They employ three different techniques. The most straightforward technique uses the total execution time for T in equation Equation (4.15). The second approach uses the maximum computation time across all processors and the communication time from that same processor. The last technique uses the parallel execution's *critical path*. It helps avoiding blocking time since any communication on the critical path is pure communication. The last two techniques model computation and communication separately, then combine the modeled computation and communication time to determine the final execution time.

Their goal is similar to ours in a sense that the target number of cores for prediction is bigger than the number of cores used training. However, the application parameter domain is same in the training experiments and prediction (strong scaling), which makes it unable to predict performance for inputs outside the training application parameter domain. They also assume that the computational load is well balanced, which is true in some MPI applications they evaluated, but rarely holds for task parallel applications. Also unlike MPI applications, task parallel applications have to consider many more factors besides just computation and communication, such as task migration overhead and availability of parallelism.

Chapter 5

Performance Modeling With Direct Regression

Network Structure

We use a neural network regressor with two hidden layers and one dropout layer as illustrated in Figure 5.1. Each hidden layer has 64 nodes with ReLU activation function. We call this model *Direct NN* to differentiate it from the multi-task neural network introduced in Section 6.2. Also we denote the neural network regression model by Lee et al. [36] by *LeeNN*.

Rectified Linear Unit (ReLU)

We use the rectified linear units (ReLU) as activation function as opposed to the sigmoid function used in *LeeNN*. We preferred ReLU for following reasons.

- Since the output is not squashed into a limited range like sigmoid; it is more suitable for extrapolation.
- Computation is faster.
- It converges much faster in stochastic gradient descent optimization [33]. Since our network is not large, this point is not as important, but still, it favors ReLU.

Dropout

Building a model that will perform well not just on the training data, but also on new unseen inputs, is a central problem in machine learning. There are many *regularization* methods which are explicitly designed to reduce the test error at the expense of increased training error.

Recently proposed by Srivastava et al. [49], *dropout* is a surprisingly effective regularization method for neural networks. Our network has a dropout layer between the second hidden layer and the readout layer, as shown in Figure 5.1. Nodes in the dropout layer *turns off* with a predefined probability *keep_prob* during training. This can be thought

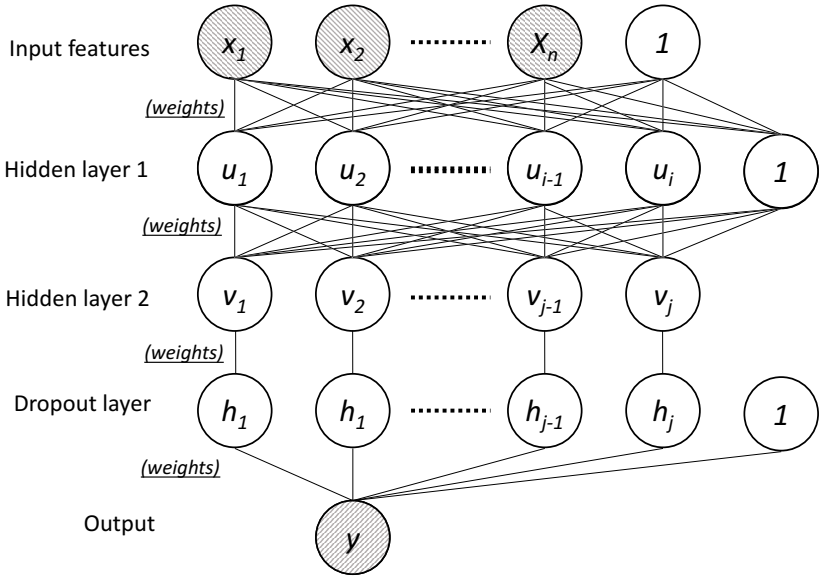


Figure 5.1: Neural Network With Two Hidden Layers And One Dropout Layer (*Direct NN*)

of as a method of building a computationally cheap ensemble of many smaller neural networks which are subsets of the original network without dropout. Therefore, since ensemble methods improve generalization, incorporating dropout layer improved our model performance.

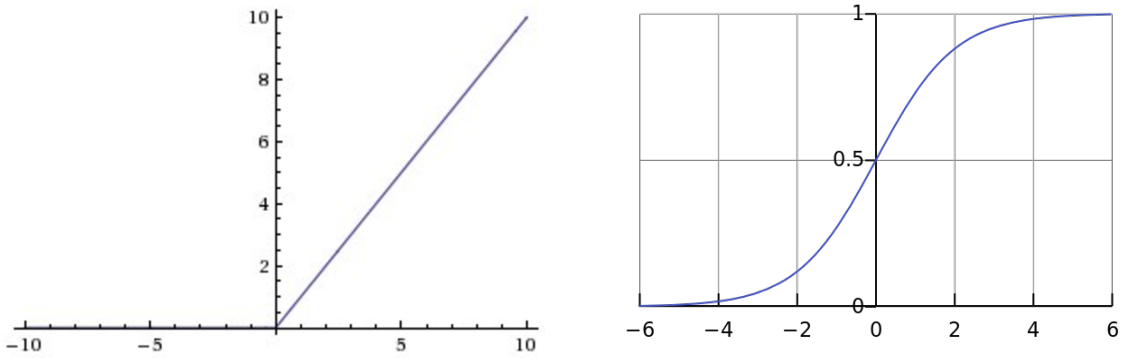
Weighted Loss Function

One of the commonly used techniques in domain adaptation (Section 3.2) is use of *weighted loss function*. The intuition behind this is that if the input probability distribution of the train and test set is different, we should give more importance to the train samples which are closer to the test distribution.

Since in our case, we the number of workers is larger in the prediction target than training, we need to give more importance the samples with a larger number of workers. That is, we define the loss function as follows.

$$loss = \sum_{i=1}^{n} (y_i - y(x_i))^2 workers_i, \tag{5.1}$$

where $workers_i$ is the number of workers used in sample i, y_i is the multiplication of actual execution time and workers (we denote $time\_workers$ from now), x_i is the input features vector ($[workload\_size_i, workers_i, config\_param_i, 1]$), and $y(x_i)$ is the predicted $time\_workers$.



(a) ReLU: $\max(0, x)$ (b) Sigmoid: $(1 + e^{-x})^{-1}$

Figure 5.2: Activation functions

Stochastic Gradient Descent Optimization

We optimize the weights in the network with backpropagation using *AdamOptimizer* (adaptive moment estimation optimizer) [32]. AdamOptimizer is a method for stochastic gradient descent optimization with individual adaptive learning rates for each parameter.

Hyperparameter Optimization

We have to choose two *hyperparameters* in our neural network regressor: dropout *keep probability* and initial *learning rate* for AdamOptimizer. During the training, we take a small part of the training data to make a validation dataset. We randomly sample the parameters and choose the values with smallest validation loss [8]. The dropout keep probability is chosen uniformly random from $(0, 0.9)$ and the learning rate is chosen uniformly random in *log* scale from $[10^{-6}, 1]$.

We summarize the main differences between our model (*Direct NN*) and *LeeNN* in Table 5.1.

Table 5.1: Main differences between *LeeNN* and *Direct NN*

	LeeNN	*Direct NN*
number of hidden layers	1	2
number of nodes in each hidden layer	16	64
hidden layer activation function	$Sigmoid(x) = (1 + e^{-x})^{-1}$	$ReLU(x) = \max(0, x)$
cost function	mean squared error	weighted mean squared error
additional layer	*none*	dropout
hyperparameter optimization	*none*	random sampled search
early stopping	yes	yes
weight optimization method	full gradient descent with *RPROP*	stochastic gradient descent with *AdamOptimizer*

Chapter 6

Profiling Assisted Performance Modeling

During the training executions, we can record not only the execution time, but also other metrics such as `work`, `delay`, `create_task`, and `critical_path`. Then we assist the regression models using this extra information gained with profiling.

In this chapter, we introduce two performance modeling techniques, which exploits this additional information. The first is a two-step linear regression model. The second model is a multi-task regression neural network.

6.1 Two-step Linear Regression

6.1.1 Model Overview

The overall prediction process of our two-step linear regression performance model is shown in Figure 6.1. It involves three main steps:

1. Execute the target application on *measurement configurations* with DAG Recorder. It records five properties from the recorded DAG: `T1` (work), `delay`, `create_task`, `wait_tasks`, and `no_work`; and the `T1'` (serial execution work) of each execution.
2. Next, using the measured data, it trains the six intermediate models: `T1` (work), `delay`, `create_task`, `wait_tasks`, `no_work`, and `T1'` (Figure 6.3).
3. Finally, by feeding the prediction target values (*problem size*: n, *no. of workers*: p) to the intermediate prediction models, it will predict the target execution time. Specifically, the execution time is determined from `T1` (work), `delay`, `no_work`, and Equation (2.1).

6.1.2 Model Details

We formulate each edge in Figure 6.1 using a linear equation. We find the coefficients of each linear model using $l1$-regularized linear regression (Lasso) [50] The most important

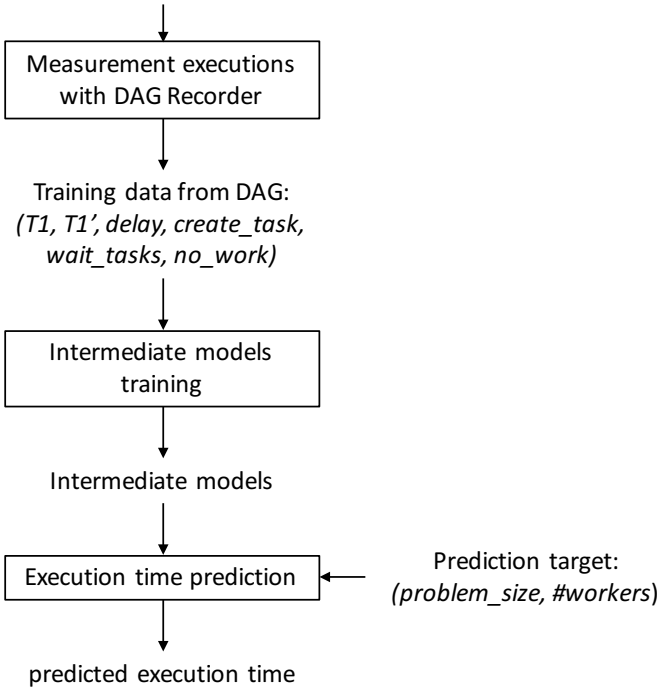


Figure 6.1: Two-step LassoLars Prediction Process Overview

property of Lasso is that it tends to produce some coefficients that are exactly zero, making the models much more interpretable and strong for extrapolation. We find the $l1$-regularization penalty coefficient using cross-validation.

We can further write Equation (2.1) as following

$$time(n,p) = \frac{1}{p}(T_1(n,p) + delay(n,p) + no\_work((n,p))) \tag{6.1}$$

where we denote the input problem size by n.

Our model predicts $T_1(n,p)$, $delay(n,p)$, and $no\_work((n,p))$ separately, then applies Equation (6.1) to determine the final execution time.

`Work (T1)`

We model `work (T1)` using following equation

$$\begin{aligned} T_1(n,p) &= T_1(T_1'(n),p) \\ &= T_1'(n) + a_1 \cdot T_1'(n) \cdot \frac{p-1}{p} + a_2 \cdot T_1'(n) \cdot (p-1) \end{aligned} \tag{6.2}$$

where a_i are some non-negative constant values.

The reasoning behind this model is that, when p workers participate in the execution, some part of the serial execution work proportional to $p-1$ or $\frac{p-1}{p}$ will be moved around different cores. Therefore, the work time inflation will be proportional to $p-1$ and $(p-1)/p$.

On the other hand, the `work` of the serial execution is modeled as follows,

$$T_1'(n) = \sum b_i n^j log^k(n) \tag{6.3}$$

where b_i are non-negative constants and $0 \leq j \leq 3, 0 \leq k \leq 2$. This representation is, of course, not exhaustive, but is sufficient in most practical applications since it is a consequence of how most computer algorithms are designed.

`Delay`

In order to model `delay`, we utilize another two: DAG properties `create_task` and `wait_tasks`. Because `delay` is the overhead caused by task creation and synchronization. Also, we imagine that inter-core task movement is proportional to some combination of $p-1$ and $(p-1)/p$. Consequently, we model `delay` as follows, where c_i are application & platform specific constants.

$$\begin{aligned} delay(n,p) &= delay(create\_task, wait\_tasks, p) \\ &= create\_task(n) \cdot (c_1 + c_2(p-1) + c_3\frac{p-1}{p}) \\ &+ wait\_tasks(n) \cdot (c_4 + c_5(p-1) + c_6\frac{p-1}{p}) \end{aligned} \tag{6.4}$$

On the other hand, `create_task` and `wait_tasks` are modeled using following general equations, where $0 \leq j \leq 3, 0 \leq k \leq 2$ (same as $T_1'(n)$).

$$create\_task(n) = \sum d_i n^j log^k(n) \tag{6.5}$$

$$wait\_tasks(n) = \sum e_i n^j log^k(n) \tag{6.6}$$

`No_work`

Lets assume that the gray area in Figure 6.2 shows the available parallelism of an application with `time` as x-axis. In this case, `no_work` is the sum of the area of the green rectangles when *no. of workers* is p_1, that of red rectangles when *no. of workers* is p_2. Thus, we can see that `no_work` is somewhat proportional to $(p-1)^2$. Also, if the application has inherently serial parts, the `no_work` caused by these parts will be proportional to $(p-1)$.

Therefore, *no_work* is modeled as follows, where

$$no\_work(n,p) = \sum f_i (p-1)^j n^k log^l(n) \tag{6.7}$$

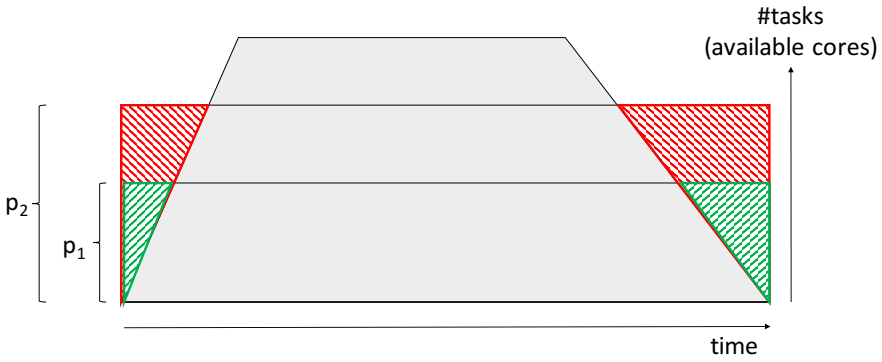


Figure 6.2: `no_work` Modeling Example Illustration

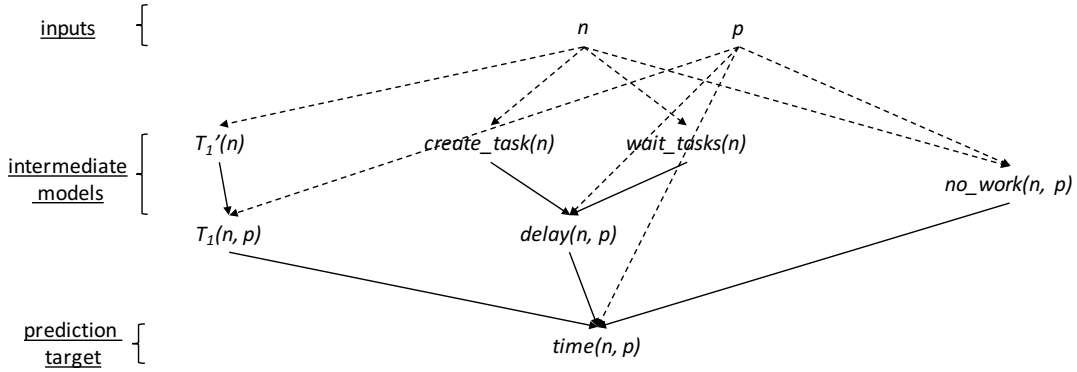


Figure 6.3: Two-step LassoLars *time* Prediction Flow

where f_i are non-negative constants and $1 \leq j \leq 2, 0 \leq k \leq 2, 0 \leq l \leq 1$.

By combining above models, when the target n and p is given, we can predict $time(n, p)$ as shown in the prediction flow Figure 6.3.

6.2 Multi-task Neural Network Regression

If we have multiple (supposedly related) labels for training, we can utilize the relation by training them with shared layers. For example, if we have 1000 samples and 10 labels for each datum, the shared layer will be trained with 10000 different samples. This increases the model's generalization ability. In our model we use `T1` (work), `delay`, `create_task`, `wait_tasks`, and `no_work` as additional tasks. We also tried using hardware performance values (such as L1, L2 cache miss count) as additional tasks; however, it was not effective.

We use a neural network regressor with two shared hidden layers, one dropout layer, and a separate output layer for each task as illustrated in Figure 6.5. We denote this model *Multi-task NN*.

The workflow of performance modeling with *Multi-task NN* is shown in Figure 6.4.

In the correlation analysis step, we choose the extra tasks from `work`, `delay`, `no_work`, `create_task`, `wait_tasks`, and `critical_path`. Since, $timeworkers = work + delay + no\_work$, we choose the two with highest Pearson correlation coefficient with `time` from `work`, `delay`, and `no_work`. `create_task`, `wait_tasks`, and `critical_path` are included automatically as extra task.

During the training, we update the weights in the network by alternatively choosing the task randomly from extra tasks + `time`. Then running a backpropagation on that task's loss.

Other details of network are same as that of `Direct NN` model introduced in Chapter 5.

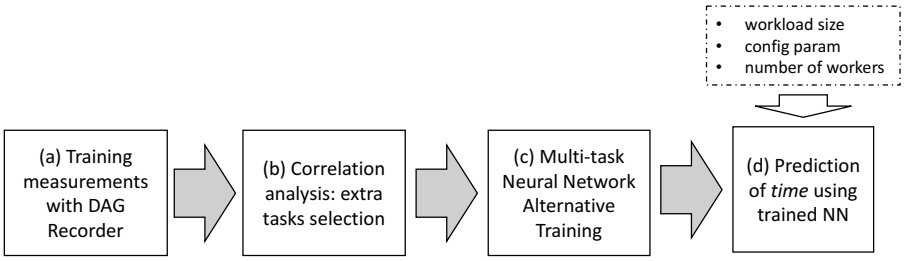


Figure 6.4: Workflow of *Multi-task NN* performance model

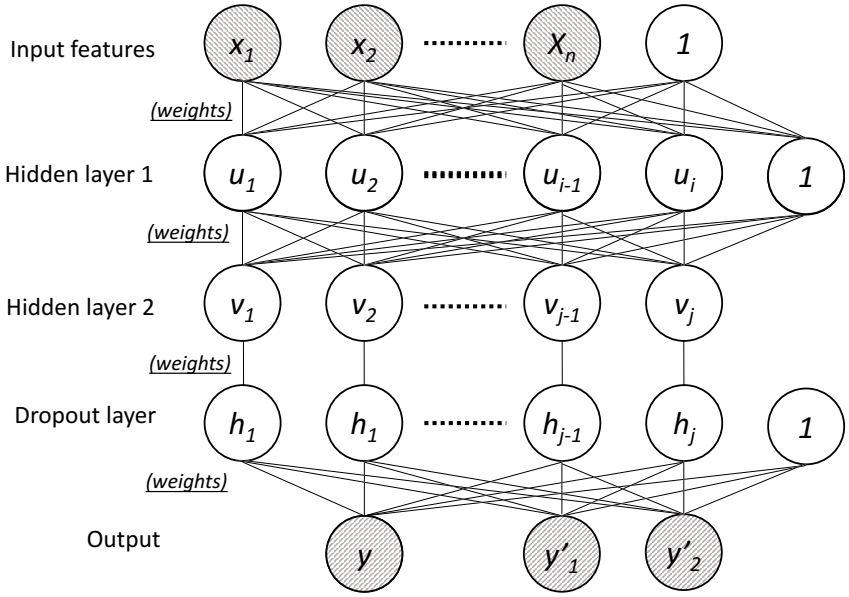


Figure 6.5: Multi-task neural network with two shared hidden layers, one dropout layer, and one separate output layer for each task

Chapter 7

Evaluation

7.1 Model Implementation

7.1.1 Linear Regression

The Lasso regression model in Section 6.1 is implemented using Python's `scikit-learn` [45] machine learning library. It provides a efficient least angle regression (LARS) implementation of Lasso [19] as `sklearn.linear_model.LassoLarsCV` module.

7.1.2 Neural Network Regression Models

We implement the neural network model described in Chapter 5 and Section 6.2 using TensorFlow, open source machine learning library [4]. It allows deploying models on both CPUs and GPUs. Since our model is not that large, we only used CPU version. TensorFlow also provides a built-in AdamOptimizer implementation.

7.2 Experiment Overview

7.2.1 Benchmark Applications

We run experiments on four applications included in Barcelona OpenMP tasks suite (BOTS) [18]: fft,sort, sparseLU, and, strassen. We modified the test suite such that it can use any task parallel runtime/library instead of OpenMP. In our evaluation, we use the MassiveThreads library [41]. MassiveThreads employs *work-first* scheduling strategy, which is the most widely used strategy for task parallel runtime/system/libraries.

FFT computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey [15] algorithm. This is a divide-and-conquer algorithm that recursively breaks down a Discrete Fourier Transform (DFT) into many smaller DFTs. 0In each of the divisions, multiple tasks are generated. We only used twos powers as n, since the algorithm is optimized for such values.

Sort sorts a random permutation of n 32-bit numbers with a fast parallel sorting variation of the ordinary mergesort. As the divided array partition becomes smaller than certain thresholds (configuration parameters), the sorting algorithm is changed in following way: parallel mergesort → serial mergesort → serial quicksort → serial insertion sort.

SparseLU computes an LU matrix factorization over sparse matrices of size n. A first level matrix is composed by pointers to small submatrices whose size is also a configuration parameter. In each of the sparseLU phases, a task is created for each block of the matrix that is not empty.

Strassen algorithm uses the hierarchical decomposition of a matrix for multiplication of large dense matrices with a size of n. A task is created for each decomposition. Creation of too many small tasks is avoided by using a depth based cutoff value (also a configuration parameter). The BOTS suite also includes other six applications: alignment, fib, floorplan, health, nqueens, and uts. Those are not used in our evaluation since there is no easy way to change the input problem size of the application (alignment, floorplan, health, and uts) or the application has exponential complexity and running them in large scale is not practical (fib and nqueens).

7.2.2 Datasets

We prepared two different datasets for evaluation. *Dataset A* has variable workload size and number of workers. It consists of a *train* and a *test* portions. On the other hand, *Dataset B* variable configuration parameter value in addition to variable workload size and number of workers. It has a *train* part and three *test* parts. We use *Dataset A* to evaluate the two-step linear regression model (Section 6.1). *Dataset B* is used for evaluating the neural regression based models: *LeeNN* [36], *Direct NN* (Chapter 5), and *Multi-task NN* (Section 6.2).

Dataset A

The training & test measurements description for each application in *Dataset A* is shown in Table 7.1. The input parameters are chosen uniformly at *log* scale for fft, sort, and strassen; at linear scale for sparseLU.

The configuration parameters used for both training and test executions are listed in Table 7.2.

Dataset B

The training & test measurements description for each application in *Dataset B* is shown in Table 7.3. The *train* part uses smaller workload sizes and fewer number of workers (up to 16). On the other hand, *test1* has larger workload sizes, *test2* has a larger number of worker, and *test3* has both larger workload sizes and a number of workers.

Table 7.1: Training & Test Measurements Description (*Dataset A*)

(a) Training (1 to 8 *workers*)

	problem size range	no. of data points per worker choice
FFT	$2^{10} \leq n \leq 2^{27}$	18
Sort	$2^{13} \leq n < 2^{27}$	34
SparseLU	$60 \leq n \leq 190$	10
Strassen	$2^{10} \leq n \leq 2^{12}$	3

(b) Test (28, 32, 36 *workers*)

	problem size range	no. of data points per worker choice
FFT	$2^{28} \leq n \leq 2^{30}$	3
Sort	$2^{27} \leq n \leq 2^{30}$	5
SparseLU	$200 \leq n \leq 500$	7
Strassen	$2^{13} \leq n \leq 2^{14}$	2

Table 7.2: Configuration Parameter Values (*Dataset A*)

	config params	explanation
FFT	None	
Sort	`-a 512 -y 512 -b 20`	algorithm change thresholds
SparseLU	`-m 30`	submatrix size
Strassen	`-x 7 -y 32`	runtime & app task cut-off values

7.2.3 Environment

The experiments run on a machine with 36 physical cores (two sockets, Xeon E5-2699 v3 Haswell) and 768GB memory (PC4-17000) running Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-45-generic x86_64). Programs were compiled with GCC 5.4.0 and MassiveThreads version 0.97.

7.3 Results

7.3.1 Evaluation With Variable Workload Size and Workers

We train the two-step linear regression model (Section 6.1) with train part of *Dataset A*. The model's prediction error on the test part of *Dataset A* is shown in Figure 7.1. It is a scatter plot overlayed with box-and-whisker plots showing the error percentage of execution *time* prediction. The error percentage is defined as $|actual - predicted|/actual$. Red line the graph represents the median (also written in gray above) error, while the bottom and top of the blue box are the first and third quartiles. The whiskers represent

Table 7.3: Training & Test Measurements Description (*Dataset B*)

(a) Training (1 to 16 *workers*)

	problem size range	config param range	no. of data points per worker choice
Sort	$2^{13} \leq n < 2^{27}$	$512 \leq y \leq 2048$	100
SparseLU	$60 \leq n \leq 195$	$20 \leq m \leq 40$	100

(b) Test1 (1 to 16 *workers*)

	problem size range	config param range	no. of data points per worker choice
Sort	$2^{27} \leq n < 2^{30}$	$512 \leq y \leq 2048$	6
SparseLU	$200 \leq n \leq 515$	$20 \leq m \leq 40$	6

(c) Test2 (32 to 36 *workers*)

	problem size range	config param range	no. of data points per worker choice
Sort	$2^{13} \leq n < 2^{27}$	$512 \leq y \leq 2048$	6
SparseLU	$60 \leq n \leq 195$	$20 \leq m \leq 40$	6

(d) Test3 (32 to 36 *workers*)

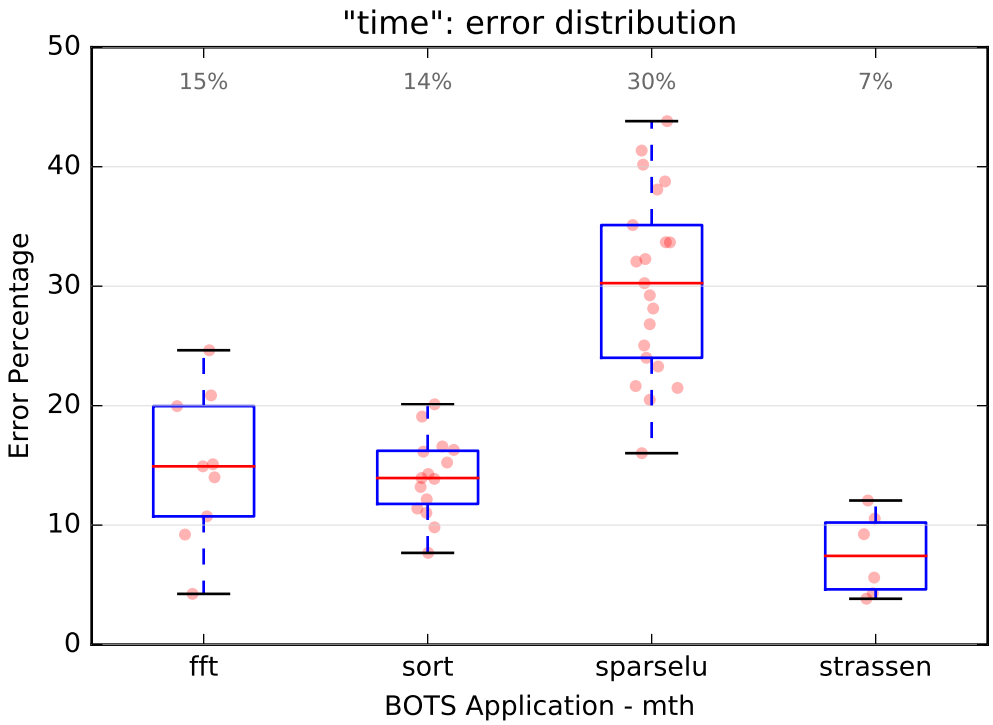
	problem size range	config param range	no. of data points per worker choice
Sort	$2^{27} \leq n < 2^{30}$	$512 \leq y \leq 2048$	6
SparseLU	$200 \leq n \leq 515$	$20 \leq m \leq 40$	6

the lowest datum still within 1.5 IQR (1st quartile subtracted from the 3rd quartile) of the lower quartile, and the highest datum still within 1.5 IQR of the upper quartile. Points outside the whiskers are considered statistical outliers.

We can see that prediction error is low, except on *sparselu*. Figure 7.2 shows the detailed plot comparing actual execution time versus predicted execution times. On these *actual vs. prediction* plots, its x-axis represents the actual measured value and y-axis represents the predicted value for each data point. The red line is the ideal prediction line, meaning that points close to the line represent high-accuracy predictions. Since $time = (work + delay + no\_work)$, from the figure, it is observed that the `time`'s prediction error is wholly caused by the error in predicting`no_work`.

7.3.2 Evaluation With Variable Workload Size, Configuration Parameter, and Workers

We train *LeeNN*, *Direct NN*, and *Multi-task NN* models with the train part of *Dataset B*. Then the trained models are evaluated on each three test sets. The model's prediction error on the test part of *Dataset A* is shown in Figure 7.1.



(a) *mth*, training up to 8 cores

Figure 7.1: Scatter box plot showing the error percentage of *time* prediction made by two-step linear model on *Dataset A – test*

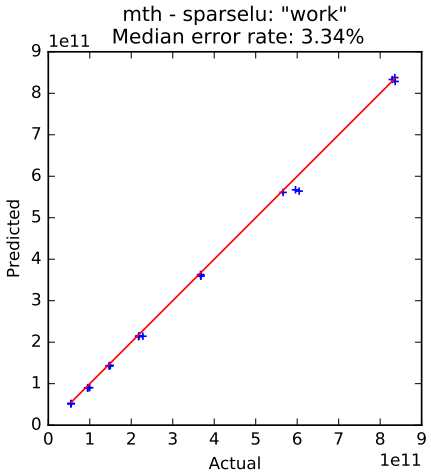
Actual vs prediction plot by *LeeNN*, *Direct NN*, and *Multi-task NN* are shown Figure 7.3, Figure 7.4, and Figure 7.5 respectively. As seen from the figures, prediction by *LeeNN* cannot extrapolate outside its training range. It is especially easy to see from Test1 results.

On the other hand, *Direct NN* and *Multi-task NN* models are extrapolating well outside their training range. The exception is *Test3 – sparsely*, where both model's median error rate is more than 50%. It is also worth noting that the *Multi-task NN* is performing slightly better across most cases.

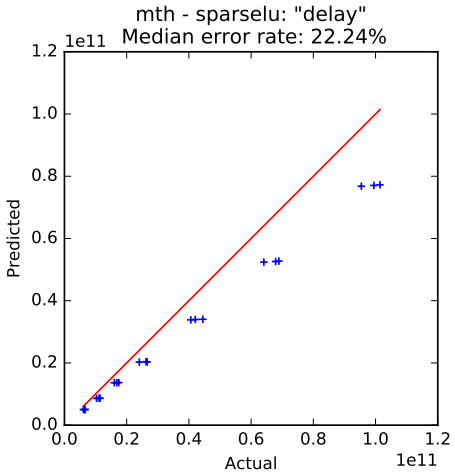
We summarize the results in Table 7.4.

Table 7.4: *Dataset B*: Prediction Error Summary

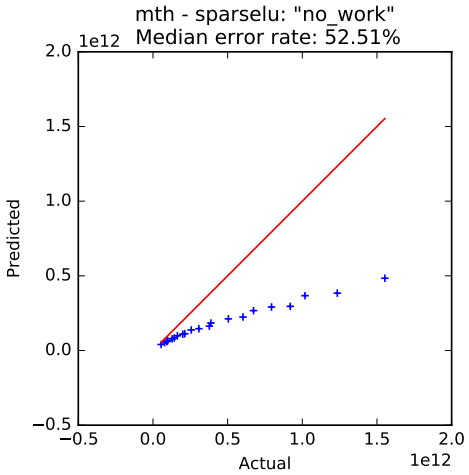
		LeeNN	*Direct NN*	*Multi-task NN*
Sort	Test1	26.56%	**9.04%**	10.29%
	Test2	85.57%	17.47%	**8.03%**
	Test3	17.56%	**3.15%**	5.52%
SparseLU	Test1	46.97%	38.82%	**24.33%**
	Test2	**18.11%**	38.38%	30.19%
	Test3	67.73%	70.56%	**57.29%**



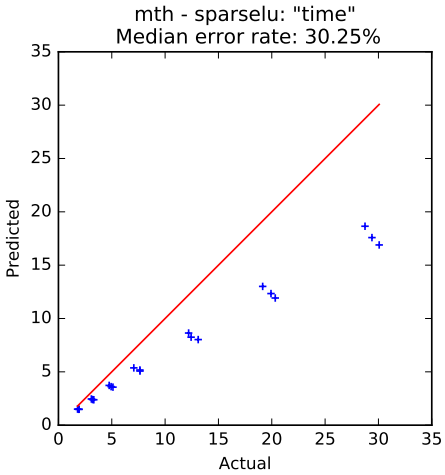
(a) `work`



(b) `delay`

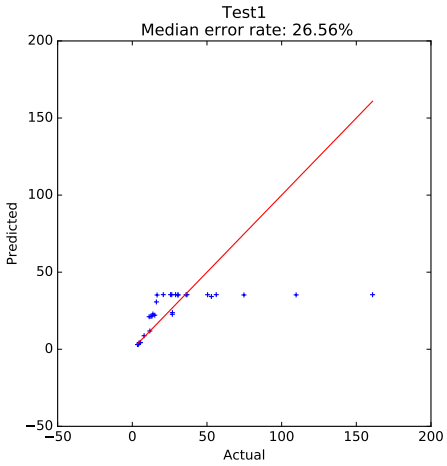


(c) `no_work`

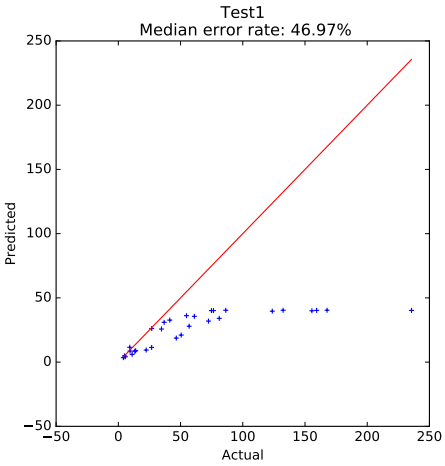


(d) `time`

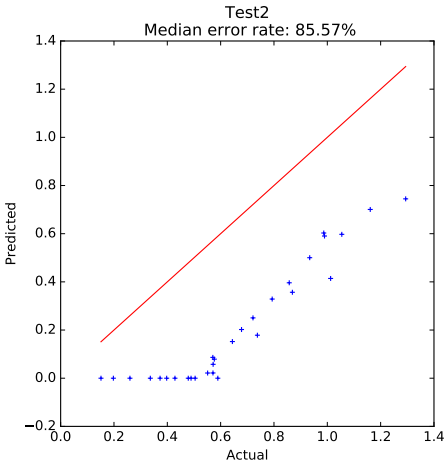
Figure 7.2: Actual vs prediction plot for sparseLU/BOTS application on *Dataset A - test*.



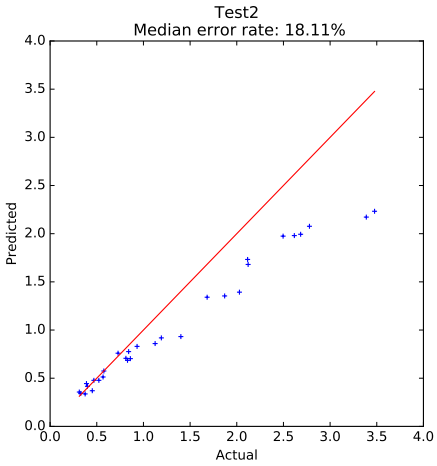
(a) Test1 – sort



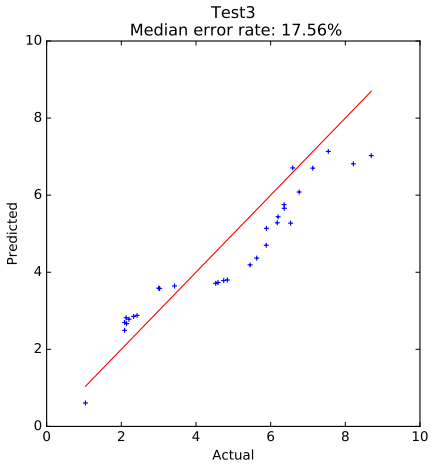
(b) Test1 – sparselu



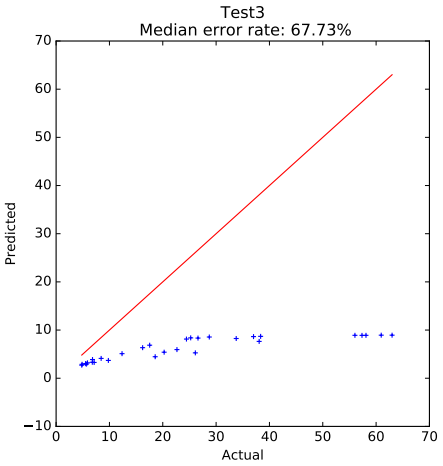
(c) Test2 – sort



(d) Test2 – sparselu

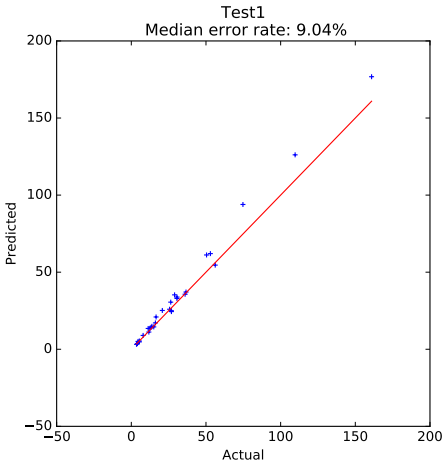


(e) Test3 – sort

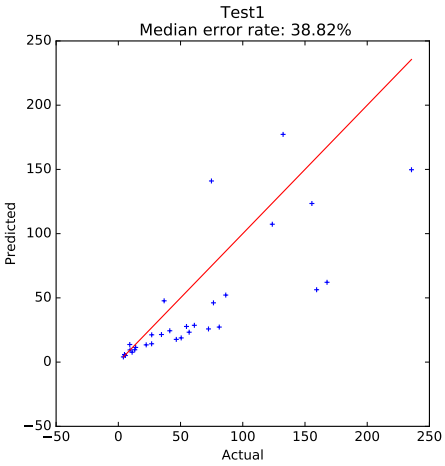


(f) Test3 – sparselu

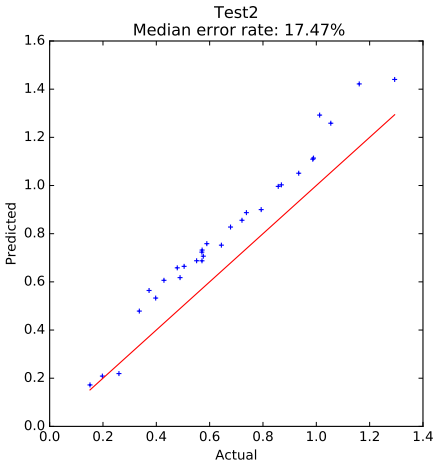
Figure 7.3: Actual vs prediction by *LeeNN* plot on *Dataset B*.



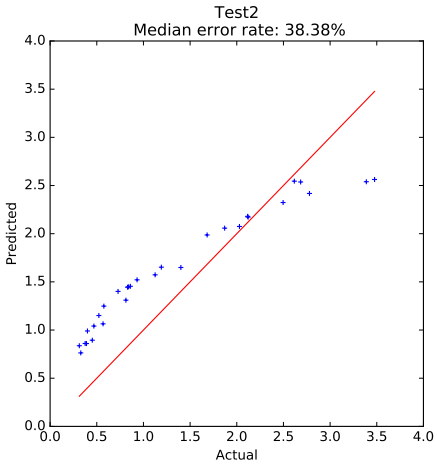
(a) Test1 – sort



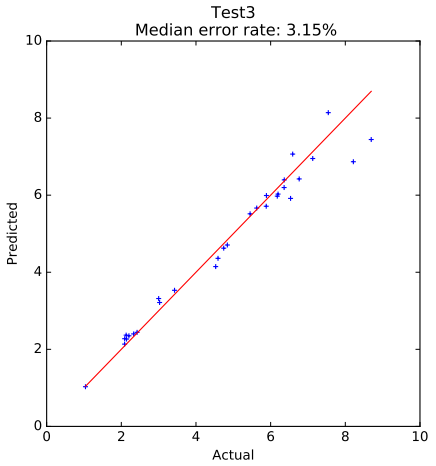
(b) Test1 – sparselu



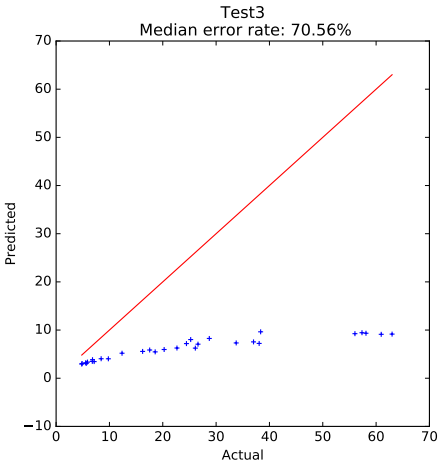
(c) Test2 – sort



(d) Test2 – sparselu

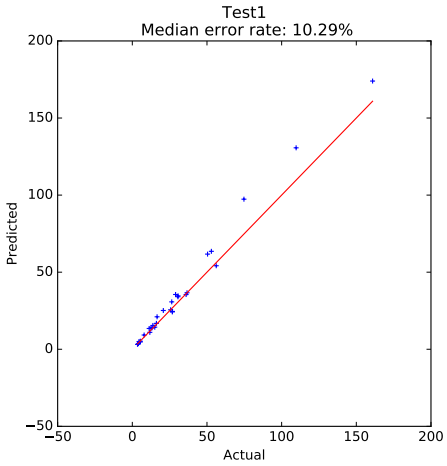


(e) Test3 – sort

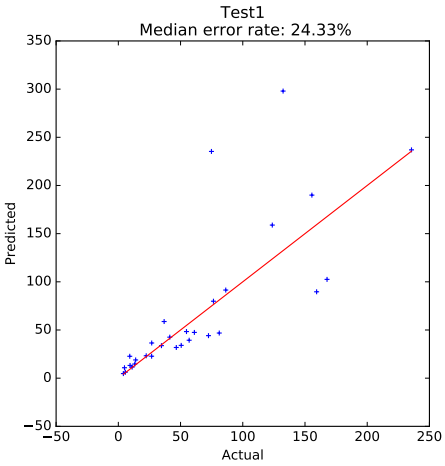


(f) Test3 – sparselu

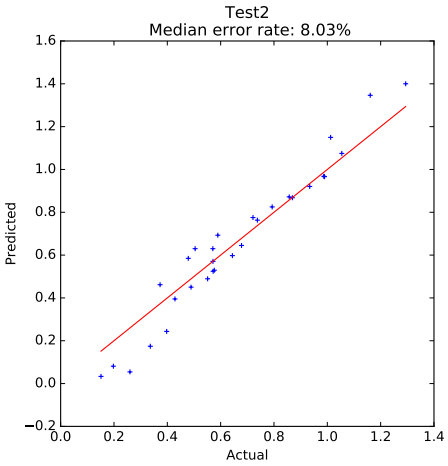
Figure 7.4: Actual vs prediction by *Direct NN* plot on the *Dataset B*.



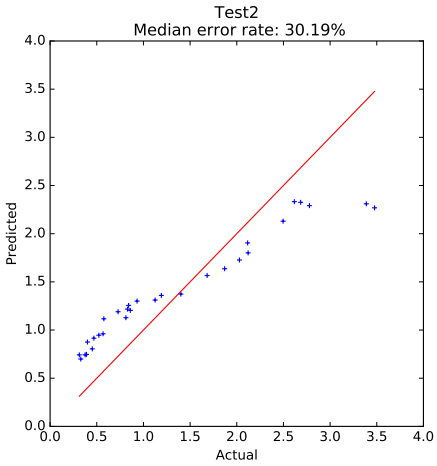
(a) Test1 – sort



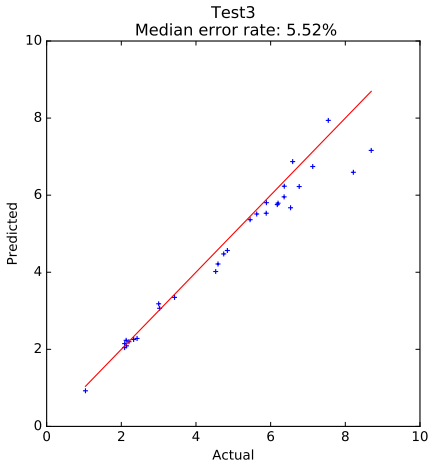
(b) Test1 – sparselu



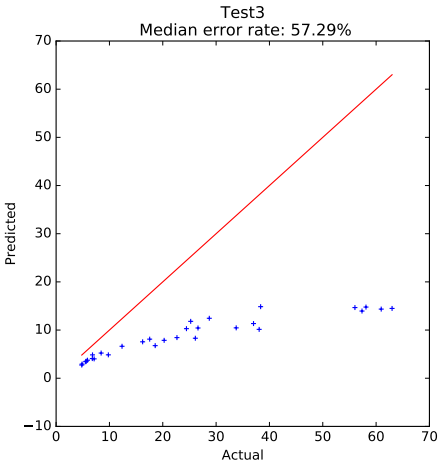
(c) Test2 – sort



(d) Test2 – sparselu



(e) Test3 – sort



(f) Test3 – sparselu

Figure 7.5: Actual vs prediction by *Multi-task NN* plot on *Dataset B*.

Chapter 8

Conclusion

8.1 Summary

We presented a novel DAG-based performance model for predicting the execution time of task parallel applications. We presented three different models for performance modeling. The two-step linear regression model builds intermediate models on various directed acyclic graph (DAG) properties: *work*, *no. tasks created*, *no. of task waits*, *delay*, and *no work*. Those intermediate models are combined to determine the final execution time. *Direct NN* model predicts performance without requiring any additional information other than the execution time of training measurements. *Multi-task NN* model improves the direct neural network regression by using DAG properties as extra tasks, leading to better generalization.

Our performance models produce accurate predictions, as conveyed by our evaluations. Despite the prediction target being much larger than the training runs regarding both no. of workers and input problem size, the prediction error rate is small enough that the model is useful in practice.

8.2 Future Works

There are a few promising nonparametric regression/extrapolation techniques outside neural networks. Especially, we want to try the Gaussian process extrapolation techniques proposed by Andrew et al. [57, 56, 58].

Acknowledgments

First and foremost I offer my sincerest gratitude to my advisor, Prof. Kenjiro Taura, who has supported me throughout my thesis with his patience, knowledge, and close guidance.

I would also like to thank all fellow lab mates of the Taura Laboratory for their kindest support, and for all the fun we have had in the last two years.

I am also enormously grateful for the Japanese Government Scholarship Program for making my dream of studying in Japan reality with their generous financial support.

Last but not the least, I would like to thank my family and friends for supporting me spiritually throughout my life.

Publications

Domestic Conferences

[1] Byambajav Namsraijav, Kenjiro Taura. Performance Modeling of Task Parallel Programs. *SWoPP2016*, Nagano, 2016/8.

[2] Byambajav Namsraijav, Kenjiro Taura. Performance Modeling of Task Parallel Programs Using Machine Learning. *PRO2016-4*, Okinawa, 2017/1.

[3] S. Iwasaki, A. Huynh, C. Helm, B. Namsraijav, W. Endo, and K. Taura. TP-PARSEC: a task-parallel PARSEC benchmark suite. *RECS2016*, Tokyo, 2016/12.

Bibliography

[1] Intel cilk plus. `https://www.cilkplus.org/`.

[2] Nanos++. `https://https://pm.bsc.es/nanox/`.

[3] Openmp loop scheduling. `https://software.intel.com/en-us/articles/openmp-loop-scheduling`. , Accessed: 2015.12.01.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[5] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.

[6] Bradley J Barnes, Barry Rountree, David K Lowenthal, Jaxk Reeves, Bronis De Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 368–377. ACM, 2008.

[7] Yoshua Bengio and Olivier Delalleau. On the expressive power of deep architectures. In *International Conference on Algorithmic Learning Theory*, pages 18–36. Springer, 2011.

[8] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[9] AMD BIOS. kernel developers guide (bkdg) for amd family 10h processors, 2013.

[10] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[11] Edwin V Bonilla, Kian Ming Adam Chai, and Christopher KI Williams. Multi-task gaussian process prediction. In *NIPs*, volume 20, pages 153–160, 2007.

[12] Rich Caruana. Multitask learning. In *Learning to learn*, pages 95–133. Springer, 1998.

[13] Georgios Chatzopoulos, Aleksandar Dragojević, and Rachid Guerraoui. Estima: Extrapolating scalability of in-memory applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 27:1–27:11, New York, NY, USA, 2016. ACM.

[14] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[15] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[16] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[17] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. Cpu db: recording microprocessor history. *Communications of the ACM*, 55(4):55–63, 2012.

[18] Alejandro Duran González, Xavier Teruel, Roger Ferrer, Xavier Martorell Bofill, and Eduard Ayguadé Parra. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *38th International Conference on Parallel Processing*, pages 124–131, 2009.

[19] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[20] Vassiliy A Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability & Its Applications*, 14(1):153–158, 1969.

[21] Theodoros Evgeniou and Massimiliano Pontil. Regularized multi–task learning. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 109–117. ACM, 2004.

[22] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[23] Felix Garcia and Javier Fernandez. Posix thread libraries. *Linux Journal*, 2000(70es):36, 2000.

[24] Saturnino Garcia, Donghwan Jeon, Christopher M Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI*. ACM, 2011.

[25] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[26] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[27] Yuxiong He, Charles E Leiserson, and William M Leiserson. The cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 145–156. ACM, 2010.

[28] An Huynh, Douglas Thain, Miquel Pericàs, and Kenjiro Taura. Dagviz: A dag visualization tool for analyzing task-parallel program traces. In *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, VPA '15, pages 3:1–3:8, New York, NY, USA, 2015. ACM.

[29] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[30] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: parallel speedup estimates for serial programs. In *OOPSLA*. ACM, 2011.

[31] Minjang Kim, Pranith Kumar, Hyesoon Kim, and Bevin Brett. Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1318–1329. IEEE, 2012.

[32] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[34] Neil D Lawrence and John C Platt. Learning to learn with the informative vector machine. In *Proceedings of the twenty-first international conference on Machine learning*, page 65. ACM, 2004.

[35] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000.

[36] Benjamin C Lee, David M Brooks, Bronis R de Supinski, Martin Schulz, Karan Singh, and Sally A McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258. ACM, 2007.

[37] Sijin Li, Zhi-Qiang Liu, and Antoni B Chan. Heterogeneous multi-task learning for human pose estimation with deep convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 482–489, 2014.

[38] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[39] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[40] Elizbar A Nadaraya. On estimating regression. *Theory of Probability & Its Applications*, 9(1):141–142, 1964.

[41] Jun Nakashima and Kenjiro Taura. Massivethreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, pages 222–238. Springer, 2014.

[42] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[43] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[44] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[45] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[46] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[47] Anton Schwaighofer, Volker Tresp, and Kai Yu. Learning gaussian process kernels via hierarchical bayes. In *NIPS*, pages 1209–1216, 2004.

[48] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, P Beckman, C Bordage, G Bosilca, A Brooks, A CastellAs, D Genet, T Herault, et al. Argobots: A lightweight low-level threading/tasking framework, 2015.

[49] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[50] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[51] Gregory Valiant. Learning polynomials with neural networks. 2014.

[52] John E Walsh. *Handbook of nonparametric statistics*, volume 1. Van Nostrand Princeton, NJ, 1962.

[53] Wei Wang, Tamal Dey, Jack W Davidson, and Mary Lou Soffa. Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 380–391. IEEE, 2014.

[54] Geoffrey S Watson. Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 359–372, 1964.

[55] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[56] Andrew Wilson, Elad Gilboa, John P Cunningham, and Arye Nehorai. Fast kernel learning for multidimensional pattern extrapolation. In *Advances in Neural Information Processing Systems*, pages 3626–3634, 2014.

[57] Andrew Gordon Wilson. Covariance kernels for fast automatic pattern discovery and extrapolation with gaussian processes. *University of Cambridge*, 2014.

[58] Andrew Gordon Wilson and Ryan Prescott Adams. Gaussian process kernels for pattern discovery and extrapolation. In *ICML (3)*, pages 1067–1075, 2013.

[59] Xing Wu and Frank Mueller. Scalaextrap: Trace-based communication extrapolation for spmd programs. In *ACM SIGPLAN Notices*, volume 46, pages 113–122. ACM, 2011.

[60] Jidong Zhai, Wenguang Chen, and Weimin Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM Sigplan Notices*, volume 45, pages 305–314. ACM, 2010.

[61] Zhanpeng Zhang, Ping Luo, Chen Change Loy, and Xiaoou Tang. Facial landmark detection by deep multi-task learning. In *European Conference on Computer Vision*, pages 94–108. Springer, 2014.