Performance Modeling of Task Parallel Programs

Byambajav Namsraijav[†1,a) Taura Kenjiro[†1,b)

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. This paper proposes a method which combines profiling based analytical performance modeling techniques with regression-based models. Our evaluation shows that predicting the execution time of task parallel applications using the proposed two-step method is significantly more accurate than predicting the execution time directly with regression models.

Keywords: Task-parallel programming, performance modeling, directed acyclic graph model

1. Introduction

From 1985 to 2005 the performance of CPUs increased dramatically, on average 50% per year [1]. 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, the development of computer hardware has shifted from increasing the clock speed of a single-core CPU to increasing the number of cores integrated into a multicore CPU.

Due to this recent trend in hardware, parallel programming is becoming more and more ubiquitous. However, explicitly specifying all the details of a parallel application is complicated and tedious work for a programmer. Conventional native threading libraries, such as POSIX Threads (pthreads) [2], require developers to deal with low-level details of thread management, load balancing, and task scheduling. Task parallel programming models are recently gaining interest to remove this burden from programmers and make writing parallel applications easier.

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 demand. One use case for such performance model is that it enables to gauge the execution time of parallel applications on large many-core systems. Because 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.

Existing modeling techniques can be divided into two main types: regression based models and analytical models. Analytical performance models use the information gained during its profiling phase with analytical models to predict the execution time for the target no. of workers. However, the prediction is only limited for inputs seen in the profiling phase. Regression-based performance models execute many training runs to train a regression model. These models can estimate the execution time for unseen inputs, but the estimation does not extrapolate well if the input size or no. of workers is beyond the training range.

We present a novel task-parallel program performance model, which can predict the execution times of task-parallel applications, even when the target problem size and no. of workers are bigger than that used in the training. We build several intermediate models using regression, but at the same time, we also make use of the logical relations between them.

1.1 Organization of the Paper

Section 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. Section 3 introduces several representative works about performance modeling of parallel programs. Section 4 presents the proposed DAG-based task-parallel programming performance model which combines regression models with analytical properties of the DAG. Section 5 discusses the results of our performance modeling techniques. The paper finishes with the summary and conclusion in Section 6.

2. Task Parallel Programming

2.1 Overview

The central principle behind the design of task parallel programming models is that the application developer should be responsible for recognizing elements that can safely be executed synchronously and expressing the parallelism. Then it should be left to the run-time environment, or the scheduler, to decide how

[†1] Presently with The University of Tokyo
[a)] byambajav@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

© 2016 Information Processing Society of Japan

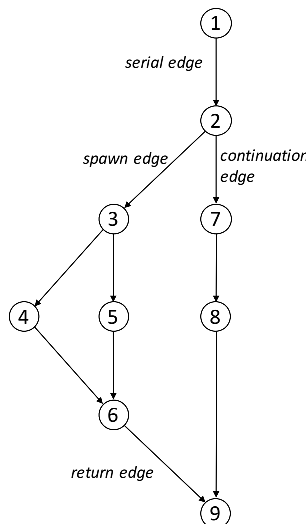


Fig. 1: A DAG Representation of Multithreaded Application

actually to divide the tasks between underlying processors. In task parallel programming, programmers ubiquitously use *tasks* to express logical tasks and their order.

There are many languages, frameworks, and libraries that support task parallelism, such as OpenMP tasks [3], MassiveThreads library [4], Intel Cilk Plus [5], Java fork/join framework [6], Intel® threading building blocks [7], and Qthreads [8].

2.2 The Directed Acyclic Graph (DAG) Model

The directed acyclic graph (DAG) model [9] for multithreading provides a general and precise quantification of parallelism. The DAG model is the base of most analytical modeling formulas of task parallel programs. 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 **Fig. 1**. This DAG contains two spawns and nine strands in total. Graph edges represent the dependencies between strands. 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 Fig. 1.

The DAG model provides two measures for describing program's parallelism quantitatively. These two measures are introduced 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 workers is usually denoted as T_p. The work is equal to the execution time of one worker; thus, we denote it as T_1. The work for the example DAG in Fig. 1 is nine if we assume that all strands can be executed in unit time.

In a simple theoretical model of *Work Law*, P workers can execute at most P instructions at a time. Thus, the following inequality holds for T_p.

$$T_p \geq T_1/P \quad (1)$$

The above inequality provides a lower bound for the parallel execution time on P workers 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 Fig. 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. Apparently, 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-worker execution time.

$$T_p \geq T_\infty \quad (2)$$

3. Related Works

We divide the related works into two parts: profiling based analytical performance analysis tools specifically designed for task parallel applications and general-purpose regression based performance modeling techniques.

3.1 Profiling Based Performance Analysis Tools for Task Parallel Applications

In this section, profiling based performance analysis tools are introduced. These type of performance analysis tools predicts the performance of an application on many cores using two steps. In the first step *profiling*, the application is run on single core, during which the necessary profiling information is collected. Then in the *prediction* step, the obtained profiling information is used by analytical models or emulation to predict performance on the target number of cores.

Cilkview [10] presents a model to predict speedup of task parallel programs written in Cilk Plus. Parallel Prophet [11] augments Cilkview by introducing a memory performance model. These two are described in detail below. Other related works of literature include DraMon [12], which builds high accuracy memory bandwidth usage model by taking account of nuts and bolts of memory hardware. Although DraMon cannot be directly used to predict speedup, it can be combined to the with the former two or other scalability prediction methods. Kismet [13] combined with Kremlin [14] provides parallel speedup estimates for serial programs. They employ a technique called Hierarchical Critical Path Analysis (HCPA) to compute dependency chains in the execution of a program. Then that information is used by a parallelization

© 2016 Information Processing Society of Japan

planner to provide speedup estimates. All of these profiling based tools are naturally affected by the same limitations that apply to other profiling tools that make use of dynamic information that is input-dependent and does not predict the application's execution with other inputs.

3.1.1 The Cilkview Scalability Analyzer

Cilkview [10] is a scalability analyzer tool for multithreaded applications. Specifically, the input to Cilkview is restricted to parallel programs written in Cilk Plus. It provides a lower and upper bound estimation of how the program's performance will change as the number of cores increase. The upper bound estimate of scalability is calculated using the DAG model described in Section 2.2. On the other hand, the lower bound estimate of scalability is calculated using their proposed *burdened DAG model*.

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 workers in following expected time:

$$T_p \leq T_1/P + \delta T_\infty \, , \tag{3}$$

where δ is a constant called *span coefficient* [15]. Intuitively inequality Eq. (3) means that, if the parallelism T_1/T_∞ exceeds the number of workers 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 Eq. (3), $T_1/T_p \approx P$ is derived.

The burdened DAG model calls the overhead of migration, such as a 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 standard DAG model. Also the *burdened span* is defined as the longest path in the burdened DAG. Then, a work-stealing scheduler running on P workers can execute the program in expected time

$$T_p \leq T_1/P + 2\delta \hat{T}_\infty, \tag{4}$$

where $\hat{T}_\infty$ is the burdened span. The proof of this equation can be found in the original paper [10]. This can be further transformed to the following equation to give a lower bound on the speedup.

$$\frac{T_1}{T_p} \geq \frac{T_1}{T_1/P + 2\delta \hat{T}_\infty} \tag{5}$$

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 $T_p \leq T_1/P + 1.7\hat{T}_\infty$.

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.

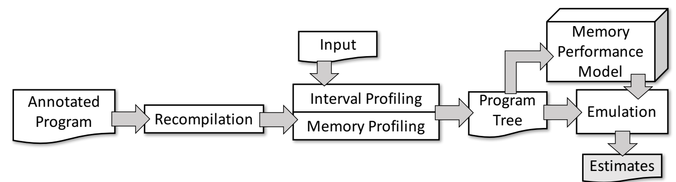


Fig. 2: Workflow of Parallel Prophet [11]

3.1.2 Parallel Prophet

Parallel Prophet [11] 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 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 **Fig. 2**. 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.

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.

Parallel Prophet calculates the burden factor β_t using these assumptions. The memory profiling uses the information obtained from low overhead hardware performance counters by PAPI [16].

Building a Program Tree

The interval profiling in Parallel Prophet is performed by running the application under PIN [17], 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

© 2016 Information Processing Society of Japan

Cilkview analyzer.

Parallel Prophet uses this information to build a *program tree*. 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.

3.2 Regression-based Performance Modeling for Parallel Applications

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, these can predict the performance for new inputs, 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. [18] tries 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 MPI applications only and not directly effective for task parallel applications.

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

[19] 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.

3.2.2 A regression-based approach to scalability prediction

Barnes et al. introduced regression based technique [18] to predict the scaling behavior of parallel programs written in MPI. They model the execution time of a parallel application as

$$log_2(T) = \beta_1 log_2(x_1) + \beta_2 log_2(x_2) + \dots + \beta_n log_2(x_n) + g(q) + error \quad (6)$$

where,

$$g(q) = \gamma_0 + \gamma_1 log_2(q) \texttt{or} g(q) = \gamma_0 + \gamma_1 log_2(q) + \gamma_2 (log_2(q))^2 \quad (7)$$

They employ three different techniques. The most straightforward technique uses the total execution time for T in equation Eq. (6). The second approach uses the maximum computation time across all workers and the communication time from that same worker. 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 scheduler behavior, task migration overhead, and effects of non-uniform memory access (NUMA).

4. DAG-based Performance Model

We propose a DAG-based performance model, which can predict the execution times of task-parallel applications, even when the target problem size and no. of workers are bigger than that used in the training.

4.1 DAG Recorder

DAG Recorder is a tracing tool to analyze the execution of task parallel programs. It is included as a part of MassiveThreads library [4]. It records all relevant events in an execution of a task-parallel program, such as task start, task creation, and task synchronization and stores them in a manner that can reconstruct the computational DAG of the execution. The generated DAG file can be viewed using DAGViz visualization tool [20]. Besides

© 2016 Information Processing Society of Japan

MassiveThreads applications, DAG Recorder can be used to analyze applications using other task-parallel runtimes such as TBB, OpenMP, CilkPlus, and QThreads.

Along with the DAG file, DAG Recorder also generates a stats text file that summarizes various pieces of information of the execution. The following list explains some of the properties that are relevant to our performance model.

- `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.

At any moment, a core either execute meaningful work, do overhead, or waits without any work to do. Therefore, by definition,

$$\texttt{T1} + \texttt{delay} + \texttt{no_work} = p \cdot elapsedtime, \qquad (8)$$

where p is the number of workers. Perfectly scalable executions have `T1` approximately the same as that of serial execution and have both `delay` and `no_work` nearly zero. On the other hand, if the application does not have enough intrinsic parallelism compared to the number of available cores, `no_work` will be large. Applications that have enough parallelism that cannot be utilized by the runtime system will show a large `delay` value and those that have their work time increased (presumably due to cache misses due to inter-core communication, false sharing, or capacity overflows on shared caches) will show a `T1` value significantly larger than that of serial execution: `T1'`.

4.2 Model Overview

The overall prediction process of our DAG-based performance model is shown in **Fig. 3**. 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'` (**Fig. 5**).

(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 Eq. (8).

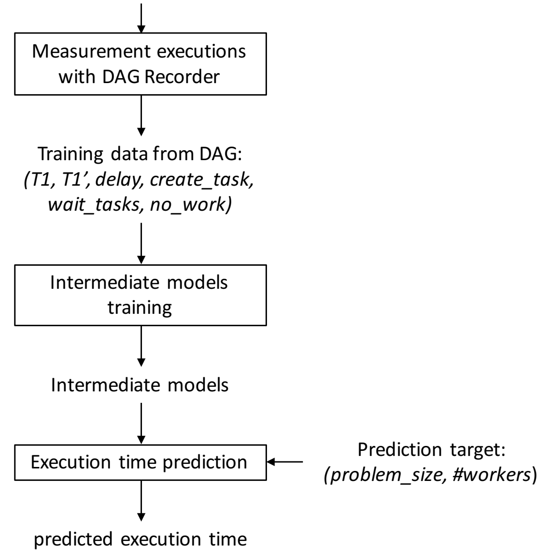


Fig. 3: Prediction process overview

4.3 Model Details

We can further write Eq. (8) as following

$$time(n, p) = \frac{1}{p}(T_1(n, p) + delay(n, p) + no_work((n, p))) \quad (9)$$

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 Eq. (9) to determine the final execution time.

`Work (T1)`

We model `work (T1)` using following equation

$$T_1(n, p) = T_1(T'_1(n), p) \qquad (10)$$
$$= T'_1(n) + a_1 \cdot T'_1(n) \cdot \frac{p-1}{p} + a_2 \cdot T'_1(n) \cdot (p-1)$$

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,

$$T'_1(n) = b_1 + b_2 n + b_3 n log(n) + b_4 n^2 + b_5 n^2 log(n) \qquad (11)$$
$$+ b_6 n^3 + b_7 n log(log(n))$$

where b_i are non-negative constants. 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.

© 2016 Information Processing Society of Japan

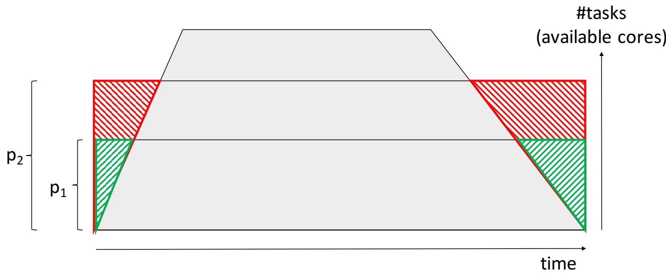


Fig. 4: no_work example illustration

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.

$$\begin{aligned} delay(n,p) &= delay(create_task, wait_tasks, p) \quad (12)\\ &= create_task(n)\cdot(c_1 + c_2(p-1) + c_3\frac{p-1}{p})\\ &+ wait_tasks(n)\cdot(c_4 + c_5(p-1) + c_6\frac{p-1}{p}) \end{aligned}$$

On the other hand, create_task and wait_tasks are modeled using following general equations.

$$\begin{aligned} create_task(n) &= d_1 n + d_2 n\cdot log(n) + d_3 n^2 \quad (13)\\ &+ d_4 n^3 + d_5 n\cdot log(log(n)) \end{aligned}$$

$$\begin{aligned} wait_tasks(n) &= e_1 n + e_2 n\cdot log(n) + e_3 n^2 \quad (14)\\ &+ e_4 n^3 + e_5 n\cdot log(log(n)) \end{aligned}$$

No_work

Lets assume that the gray area in **Fig. 4** 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 $(p-1)^2$.

Therefore, *no_work* is modeled as follows.

$$no_work(n,p) = (p-1)^2(f_1 + f_2 n + f_3 n\cdot log(n) + f_4 n^2) \quad (15)$$

By combining above models, when the target n and p is given, we can predict $time(n,p)$ as shown in the prediction flow Fig. 5.

4.4 Implementation

The models are implemented using Python's scikit-learn machine learning library. Specifically, sklearn.linear_model.LassoLarsCV model is used to find coefficients of the above models. It is a cross-validated version of an L1-regularized linear model using LARS algorithm [21]. L1-regularized models are more robust than the common

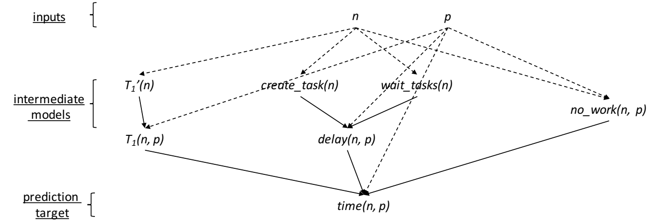


Fig. 5: Time Prediction Flow

least-squares linear model if only some of the coefficients should be non-zero. This is useful in our case, since, for example, we do not expect all seven b_i coefficients to be non-zero in the $T_1'(n)$ model.

5. Evaluation

5.1 Overview

We run experiments on six applications included in Barcelona OpenMP tasks suite (BOTS) [22]: fft, fib, nqueens, sort, sparseLU, and, strassen. These applications are modified to use MassiveThreads [4] as a task parallel library instead of OpenMP. *FFT* computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey [23] algorithm. This is a divide-and-conquer algorithm that recursively breaks down a Discrete Fourier Transform (DFT) into many smaller DFTs. In each of the divisions, multiple tasks are generated. We only used twos powers as n, since the algorithm is optimized for such values. *Fib* computes the n-th Fibonacci number using a recursive parallelization. It is not a good example of an efficient Fibonacci computation but is still considered meaningful because it is a simple test case of a deep tree composed of very fine-grained tasks. It employs a depth-based cut-off (i.e., after a certain level in the task tree it will not generate more tasks) to avoid the creation of too fine-grained tasks. *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. *NQueens* computes all solutions of the nqueens problem. Its objective is to find a placement for n queens on an $n \times n$ chessboard such that none of the queens attack any other. The algorithm uses a backtracking search with pruning. A task is created for each step of the solution. *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 four applications: alignment, floorplan, health, and uts. Those are not used in our evaluation since there is no easy way to change the input problem size of these applications.

© 2016 Information Processing Society of Japan

Table 1: Training & Test Measurements Description

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

	problem size range	no. of data points per worker choice
FFT	$2^{10} \le n \le 2^{27}$	18
Fib	$22 \le n \le 45$	24
NQueens	$8 \le n \le 14$	7
Sort	$2^{13} \le n \le 2^{27}$	63
SparseLU	$60 \le n \le 200$	29
Strassen	$2^{10} \le n \le 2^{13}$	4

(b) Test (30 to 36 *workers*)

	problem size range	no. of data points per worker choice
FFT	$2^{27} \le n \le 2^{30}$	4
Fib	$45 \le n \le 47$	3
NQueens	$14 \le n \le 17$	4
Sort	$2^{27} \le n \le 2^{31}$	63
SparseLU	$200 \le n \le 600$	17
Strassen	$2^{13} \le n \le 2^{15}$	3

The training & test measurements description for each application is shown in **Table 1**. The input parameters are chosen uniformly at *log* scale for fft, sort, and strassen, in linear scale for fib, nqueens, and sparseLU. Our models are not expecting an exponential work compared to the input variable n. Therefore, when applying our prediction model for fib and nqueens, we execute $n \rightarrow 2^n$ transformation.

The configuration parameters used for both training and test executions are listed in **Table 2**. The experiments run on a machine with 36 physical cores (two sockets, Xeon E5-2699 v3) and 768GB memory (PC4-17000). In future, we plan to experiment on a bigger machine with multiple nodes.

Table 2: Configuration Parameters

	config params	explanation
FFT	None	
Fib	`-x 19`	runtime task cut-off value
NQueens	`-x 7`	runtime task cut-off value
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

5.2 Results

We train the models using the training measurements data. Then the model is evaluated on the test measurements. **Fig. 6** shows the error percentage distribution of the model on the test measurements data. The error is defined as $|actual - predicted|/actual$. As seen from the graph, prediction error for fft, nqueens, and sparselu are very small. However, the prediction error is significant for fib and strassen.

Actual vs prediction plots of intermediate models and the final execution time prediction for sparselu is shown in **Fig. 7**. **Figures 7a to 7c and 7e** shows that the intermediate models almost perfectly predicts `create_task`, `wait_tasks`, `delay`, and `work`. However, **Fig. 7f** shows that there is a tendency to underestimate execution time as the actual time increases. As seen from **Fig. 7d**, the prediction error is wholly caused by the `no_work` intermediate model. The main prediction error cause in the other evaluated applications was the `no_work` intermediate model too.

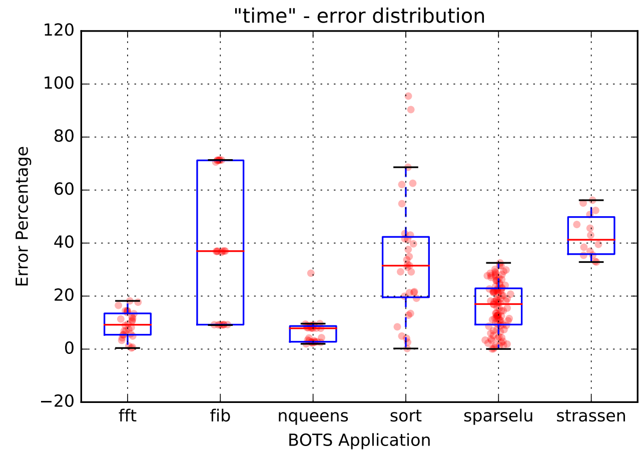


Fig. 6: Scatter plots overlayed with box-and whisker plots showing the error percentage of execution *time* prediction. The red line represents the mean, while the bottom and top of the blue box are the first and third quartiles. The whiskers represent the lowest datum still within 1.5 IQR (1st quartile 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 stastical outliers.

6. Conclusion

We presented a novel DAG-based performance model for predicting the execution time of task-parallel applications. To achieve that, our model uses intermediate models on various 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. Our model is general enough that it can be applied to any task-parallel application.

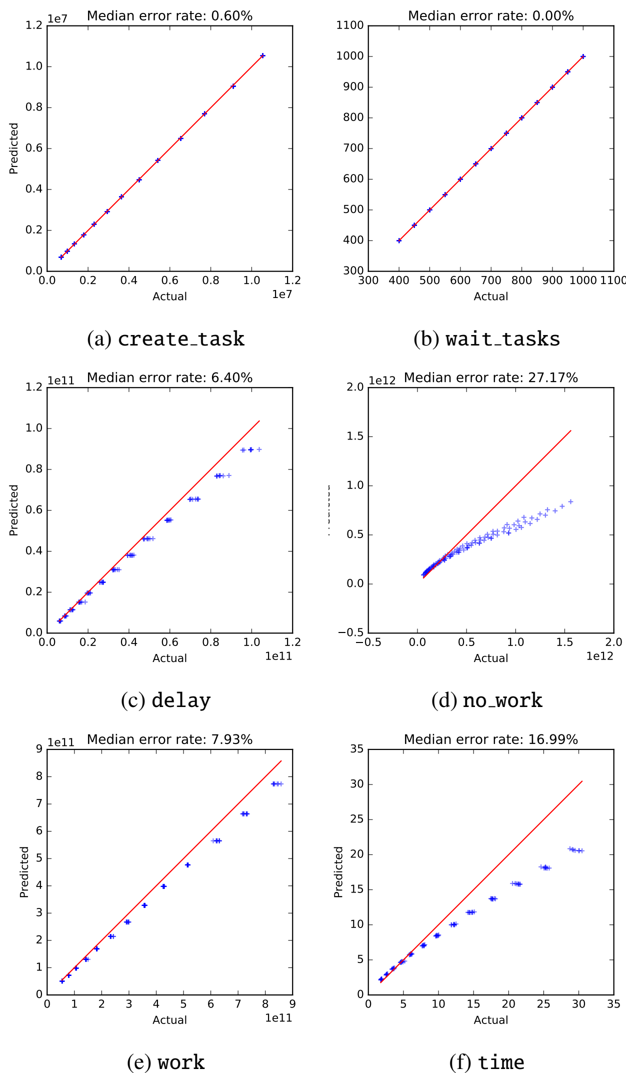
Our performance model produces accurate predictions, as conveyed by our evaluation. Despite the prediction target being much larger than the training runs regarding both no. of workers and input problem size, the median errors were lower than 10% for two out of six applications and were less than 45% for all the applications.

That being said, there is still much improvement room for our performance model. Currently, our model does not account for any limitations which may be caused by memory bottlenecks on large problem sizes not seen during the training measurements. In most applications, the accuracy of the intermediate model for `no_work` was bad. Our current DAG-based performance model does not make use of some important DAG properties such as *critical path* which is explained in Section 2.2.2. Future works may include incorporating such DAG properties to build better intermediate models, especially for `no_work`.

Acknowledgements

This work was in part supported by Grant-in-Aid for Scientific Research (A) 16H01715.

© 2016 Information Processing Society of Japan



(a) `create_task` (b) `wait_tasks`

(c) `delay` (d) `no_work`

(e) `work` (f) `time`

Fig. 7: Actual vs prediction plot for sparseLU/BOTS application on the test dataset. For each data point, its x-axis value represents the actual measured value, whereas y-axis is the predicted value. The red line is the ideal prediction line, meaning that points close to the line represent high-accuracy predictions.

References

[1] Danowitz, A., Kelley, K., Mao, J., Stevenson, J. P. and Horowitz, M.: CPU DB: recording microprocessor history, *Communications of the ACM*, Vol. 55, No. 4, pp. 55–63 (2012).

[2] Garcia, F. and Fernandez, J.: POSIX thread libraries, *Linux Journal*, Vol. 2000, No. 70es, p. 36 (2000).

[3] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P. and Zhang, G.: The design of OpenMP tasks, *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 20, No. 3, pp. 404–418 (2009).

[4] Nakashima, J. and Taura, K.: MassiveThreads: A thread library for high productivity languages, *Concurrent Objects and Beyond*, Springer, pp. 222–238 (2014).

[5] : Intel Cilk Plus, `https://www.cilkplus.org/`.

[6] Lea, D.: A Java fork/join framework, *Proceedings of the ACM 2000 conference on Java Grande*, ACM, pp. 36–43 (2000).

[7] Pheatt, C.: Intel® threading building blocks, *Journal of Computing Sciences in Colleges*, Vol. 23, No. 4, pp. 298–298 (2008).

[8] Wheeler, K. B., Murphy, R. C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, IEEE, pp. 1–8 (2008).

[9] Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, *Journal of the ACM (JACM)*, Vol. 46, No. 5, pp. 720–748 (1999).

[10] He, Y., Leiserson, C. E. and Leiserson, W. M.: The Cilkview scalability analyzer, *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, ACM, pp. 145–156 (2010).

[11] Kim, M., Kumar, P., Kim, H. and Brett, B.: Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model, *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, IEEE, pp. 1318–1329 (2012).

[12] Wang, W., Dey, T., Davidson, J. W. and Soffa, M. L.: DraMon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead, *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, IEEE, pp. 380–391 (2014).

[13] Jeon, D., Garcia, S., Louie, C. and Taylor, M. B.: Kismet: parallel speedup estimates for serial programs, *OOPSLA*, ACM (2011).

[14] Garcia, S., Jeon, D., Louie, C. M. and Taylor, M. B.: Kremlin: rethinking and rebooting gprof for the multicore age, *PLDI*, ACM (2011).

[15] Frigo, M., Leiserson, C. E. and Randall, K. H.: The implementation of the Cilk-5 multithreaded language, *ACM Sigplan Notices*, Vol. 33, No. 5, ACM, pp. 212–223 (1998).

[16] Mucci, P. J., Browne, S., Deane, C. and Ho, G.: PAPI: A portable interface to hardware performance counters, *Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10 (1999).

[17] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. and Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation, *ACM Sigplan Notices*, Vol. 40, No. 6, ACM, pp. 190–200 (2005).

[18] Barnes, B. J., Rountree, B., Lowenthal, D. K., Reeves, J., De Supinski, B. and Schulz, M.: A regression-based approach to scalability prediction, *Proceedings of the 22nd annual international conference on Supercomputing*, ACM, pp. 368–377 (2008).

[19] Lee, B. C., Brooks, D. M., de Supinski, B. R., Schulz, M., Singh, K. and McKee, S. A.: Methods of inference and learning for performance modeling of parallel applications, *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, pp. 249–258 (2007).

[20] Huynh, A., Thain, D., Pericàs, M. and Taura, K.: DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces, *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, VPA '15, New York, NY, USA, ACM, pp. 3:1–3:8 (online), DOI: 10.1145/2835238.2835241 (2015).

[21] Efron, B., Hastie, T., Johnstone, I., Tibshirani, R. et al.: Least angle regression, *The Annals of statistics*, Vol. 32, No. 2, pp. 407–499 (2004).

[22] Duran González, A., Teruel, X., Ferrer, R., Martorell Bofill, X. and Ayguadé Parra, E.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp, *38th International Conference on Parallel Processing*, pp. 124–131 (2009).

[23] Cooley, J. W. and Tukey, J. W.: An algorithm for the machine calculation of complex Fourier series, *Mathematics of computation*, Vol. 19, No. 90, pp. 297–301 (1965).

© 2016 Information Processing Society of Japan