Automatic Tuning of Bag-of-Tasks Applications

Majed Sahli [*1], Essam Mansour [†2], Tariq Alturkestani [*3], Panos Kalnis [*4]

[*] *King Abdullah University of Science & Technology, Thuwal, Saudi Arabia*
[1] majed.sahli@kaust.edu.sa
[3] tariq.alturkestani@kaust.edu.sa
[4] panos.kalnis@kaust.edu.sa

[†] *Qatar Computing Research Institute, Doha, Qatar*
[2] emansour@qf.org.qa

***Abstract*—This paper presents APlug, a framework for automatic tuning of large scale applications of many independent tasks. APlug suggests the best decomposition of the original computation into smaller tasks and the best number of CPUs to use, in order to meet user-specific constraints. We show that the problem is not trivial because there is large variability in the execution time of tasks, and it is possible for a task to occupy a CPU by performing useless computations. APlug collects a sample of task execution times and builds a model, which is then used by a discrete event simulator to calculate the optimal parameters. We provide a C++ API and a stand-alone implementation of APlug, and we integrate it with three typical applications from computational chemistry, bioinformatics, and data mining. A scenario for optimizing resources utilization is used to demonstrate our framework. We run experiments on 16,384 CPUs on a supercomputer, 480 cores on a Linux cluster and 80 cores on Amazon EC2, and show that APlug is very accurate with minimal overhead.**

I. Introduction

Many scientific and commercial applications, such as chemoinformatics [1], bioinformatics [2], or data analytics [3], are designed to leverage large scale parallel computing infrastructures. In Bag-of-tasks Applications [4], a large computational problem is decomposed into many (i.e., thousands or millions) loosely coupled tasks, which are executed on many CPUs on a private cluster, a supercomputer, or a commercial cloud. Typically, users request computational resources according to their budget. In a research environment, budget is an awarded amount of core-hours, whereas budget represents actual financial cost on a commercial cloud. In both cases, there is an incentive to efficiently utilize the computational resources.

Users can optimize various quantities; for example, minimizing the total execution time given a financial cost constraint. In this paper, we use our efficient and accurate estimation of serial and parallel execution times to optimize resource utilization. Resource utilization is quantified by *speedup efficiency* (SE). Given the time of serial execution $\mathcal{T}_1$ and the time of parallel execution of the same problem on C cores $\mathcal{T}_C$, SE is calculated as follows:

$$SE = \frac{\mathcal{T}_1}{\mathcal{C} \times \mathcal{T}_\mathcal{C}} \quad (1)$$

$SE = 1$ indicates perfect parallelism and, consequently, optimal resource utilization. In practice, resource utilization is considered good enough if $SE \geq SE_{min}$, where SE_{min} is a cutoff threshold. In this paper we will assume $SE_{min} = 0.8$.

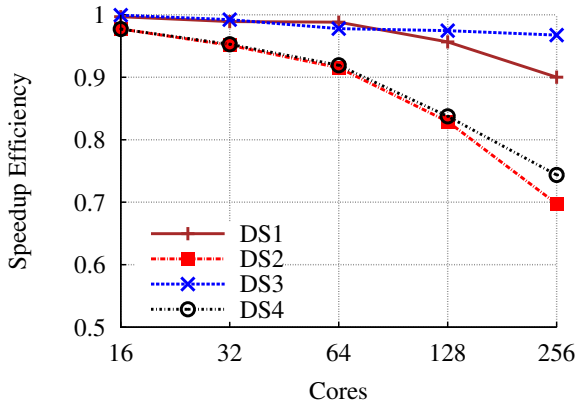


Fig. 1. Speedup efficiency of VinaLC using four subsets, DS1 to DS4, of 2,000 lead-like compounds each (i.e., same dataset and same size). DS3 scales very well to 256 cores and beyond. On the other hand, assuming SE_{min}=0.8, DS2 does not scale efficiently to more than 128 cores.

Users predict the expected scalability of a program by consulting studies on typical workloads. Consider the following example from computational chemistry: VinaLC [1] computes dockings between molecules. The developers state that VinaLC "scales up to more than 15K CPUs" [1], which is true for their dataset and computing infrastructure. We extracted four subsets, DS1 to DS4, each containing 2,000 lead-like compounds[1] from the ZINC dataset. Each compound was docked against the Thermus thermophilus gyrase B complex. We executed VinaLC on a local Linux cluster by varying the number of cores; the resulting speedup efficiencies are shown in Figure 1.

Interestingly, although all subsets come from the same dataset and contain the same number of compounds, the execution times of their tasks vary significantly. On the one hand, DS3 achieves excellent speedup efficiency for 256 cores and beyond. Therefore, it is advisable to employ more cores in order to finish execution faster. On the other hand, the speedup efficiency of DS2 drops below 0.8 (i.e., our cutoff point) after 128 cores; using more cores would waste resources. This experiment demonstrates that the scalability study in the VinaLC paper is too generic for accurate predictions in practical usage scenarios.

Two points must be noted: (*i*) We are interested in the efficient CPU utilization; *not* minimization of runtime. In the

[1] See Section VI-A for details about the experimental settings and datasets.

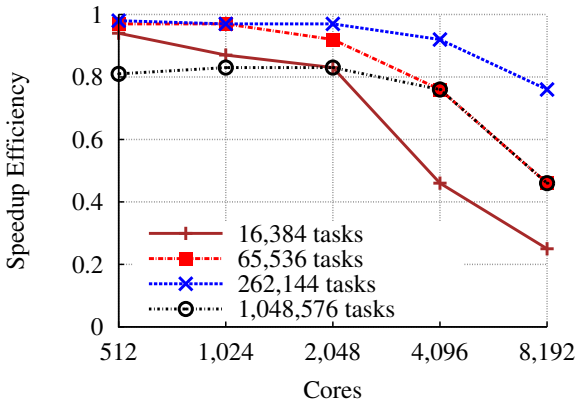


Fig. 2. Speedup efficiency of ACME for a query on the human DNA. Each line corresponds to a different decomposition of the same query. Too few tasks (i.e., 16,384) cannot achieve load balance. Too many tasks (i.e., 1,048,576) result in useless computation. For this particular combination of dataset and query parameters, the optimal decomposition generates 262,144 tasks.

previous example, DS2 would finish faster using 256 cores, although part of the user's core-hour allocation would be wasted. This fact is better appreciated if financial cost is involved. For example, in a commercial cloud a user would pay twice as much for only a moderate decrease in execution time. (*ii*) Wasting CPU resources does *not* mean the CPU is idle. We will see next that a CPU can be fully occupied by useless calculations. To the operating system, such a CPU looks busy and cannot be assigned to a different job, so there is no easy way to achieve efficient utilization. Observe that the speedup efficiency metric captures this issue.

The previous example corresponds to the easiest case with only one degree of freedom; the number of tasks is predetermined by the input, so the user can only control the number of cores. There exist applications with an additional degree of freedom; the ability to vary the granularity of the decomposition of the initial problem into tasks. ACME [3] is a data mining application representative of this category. Its input is a long sequence (e.g., DNA, text, or web log) and the output is a set of frequent patterns. In a nutshell, ACME generates a combinatorial search space represented by a tree, and executes a branch-and-bound algorithm. A decomposition of the original problem into smaller tasks corresponds to a horizontal cut of the search tree at a specific level. For instance, if the input is a DNA sequence (i.e., alphabet of 4 symbols), a cut at level 7 generates $4^7 = 16{,}384$ tasks.

We ran ACME on an IBM BlueGene/P supercomputer for different decompositions, varying the number of cores; the results are shown in Figure 2. When there are too few tasks, the granularity is not fine enough to balance the workload. Hence, the program does not scale to more than 2,048 cores. Surprisingly, if the problem is decomposed into too many tasks, scalability again suffers. This happens because many CPUs performing useless computation by processing false positive branches of the tree. We will further investigate this issue in Section IV. The best speedup efficiency up to 8,192 cores is achieved for a moderate number of 262,144 tasks. Note that the optimal decomposition cannot be determined in advance because it depends on the input sequence and the query parameters (e.g., minimum frequent pattern support, or maximum errors allowed) [5].

This paper presents APlug, an *automatic tuning* framework for large-scale bag-of-tasks applications that enables users to efficiently achieve high utilization of CPU resources. Our framework considers both degrees of freedom, namely, the degree of parallelism and the problem decomposition. APlug suggests the best combination of number of CPUs and tasks to achieve the highest speedup efficiency for a specific combination of application, dataset, query parameters, and computing infrastructure.

APlug initiates the automatic tuning process by executing a sample of tasks to generate a coarse-grained histogram of expected runtimes, which is approximated by a positively skewed gamma distribution. Then, it models the parallel execution of the entire application by a single-queue multiple-server model that approximates a typical dynamic scheduler with work stealing. APlug runs on the model a discrete event simulator that draws tasks from the gamma distribution. The output of the simulator is an estimation of speedup efficiency. The simulator is run for different decompositions and varying number of targeted CPUs in order to predict the best combination. Our experiments show that the overhead of this process is small while accuracy is very high.

We provide a C++ API that allows APlug to be easily integrated with any parallel application. We also implement APlug as a standalone utility that can be used without modifying the application's code. APlug can be used in many creative ways. For example, instead of extracting a separate sample, the application can start running on an arbitrary number of CPUs. APlug will gather statistics and then suggest to scale in or out *elastically* during the actual execution. Such online adaptivity can compensate for the expected performance variability on public clouds and shared infrastructures [6], such as the case for applications that run inside virtual machines on OpenStack. We also present in the experimental section a case study of deploying APlug on Amazon EC2. The user specifies two constraints: maximum execution time and maximum financial cost. APlug takes into account the coarse-grained hourly pricing model of Amazon EC2 and suggests the optimal number of instances to rent.

In summary, our contributions are:

- We propose APlug, an automatic tuning framework for large-scale applications with many independent tasks. APlug suggests the best task decomposition to support massive parallelism while keeping speedup efficiency within an acceptable range.
- We provide a C++ API to integrate APlug within applications, as well as a standalone implementation that does not require modification of the application code.
- We present three case studies: VinaLC [1] for computational chemistry, SSW [7] alignment tool for bioinformatics, and ACME [3] for pattern mining.
- We evaluate APlug using 16,384 CPUs on a Blue Gene/P supercomputer, 480 cores on a Linux cluster, and 80 cores on Amazon EC2. Results confirm the accuracy of APlug with minimal overhead.

The rest of this paper is organized as follows. Section II introduces the three case studies. We discuss the foundations of our model in Section III. Section IV details the APlug framework, whereas Section V describes the API. Section VI presents the experimental evaluation. Section VII presents the related work and Section VIII concludes the paper.

II. Scientific Applications

We integrated APlug into three representative scientific applications for molecular docking, sequence alignment, and pattern mining. The chosen parallel systems are *VinaLC* [1], *P-SSW* (an MPI version of SSW [7]), and *ACME* [3], respectively. VinaLC and P-SSW handle static task decomposition problems, while ACME uses dynamic task decomposition.

A. Molecular Docking: VinaLC

Molecular docking predicts the structural orientation in which two chemical molecules bind to make a stable complex [8]. It is used to discover new drugs and assist in drugs repositioning [9]. There are about 30,000 genes in the human genome that can bind to known drug molecules [10]. It is estimated that 10^{60} drug-like molecules exist [11]. To dock m molecules against n protein structures, $m \times n$ docking operations are required.

We integrated our framework with VinaLC [1]. VinaLC is the MPI implementation of Vina [12], a popular serial molecular docking system. Each docking operation is treated as a task that gets scheduled independently. However, docking operations vary in execution times, as shown in Figure 1. Users do not know the workload of their tasks in advance. Hence, it is challenging to choose the ideal degree of parallelism for high resource utilization and better response time.

B. Sequence Alignment: P-SSW

The problem of aligning sequences involves arranging sequences to identify regions of similarity. It is the first step in studying functional, structural, or evolutionary relationships between DNA, RNA, or protein sequences [13], [14]. Multiple sequence alignment is an NP-hard problem [15]. A preprocessing step for multiple sequence alignment is to find pairwise alignments between all input sequences. Given a dataset of n sequences, the number of pairwise alignments is $\binom{n}{2} = \frac{n^2-n}{2}$. So aligning 1,000 sequences requires 499,500 independent pairwise alignments.

We integrated APlug in P-SSW, our parallel implementation of the SSW [7] alignment tool. SSW is a recent extension of Farrar's implementation [16] of the optimal pairwise alignment algorithm, the Smith-Waterman dynamic programming algorithm [17]. P-SSW adopts a dynamic scheduling policy to assign tasks to workers, where each pairwise alignment is a task. This policy guarantees better load balancing among cores when processing pairwise alignments of arbitrary workloads.

C. Pattern Mining: ACME

Pattern mining finds sequential patterns that appear frequently in a very long sequence. It is an important process in applications such as genome analysis in bioinformatics [18], predicting stocks using time series [19], and running analytics on web logs [20]. The search space is expanded to cover all combinations from the alphabet of the underlying sequence.

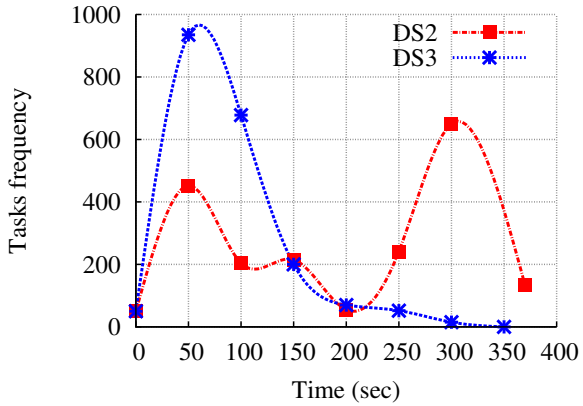


Fig. 3. Workload frequency distributions for the two subsets, DS2 and DS3, whose speedup efficiencies are shown in Figure 1. While the two subsets are of the same size (# tasks) and features (both of lead-like molecules), they have different workload distributions. In Figure 1, the speedup efficiency of DS3 is significantly higher than DS2's.

We used ACME [3], a recent combinatorial parallel solution for pattern mining, to validate our framework. ACME represents the search space of pattern mining as a combinatorial tree over the input alphabet. Tasks are created by partitioning the search space tree using prefixes. The number of tasks is exponential to the prefix length. A short prefix limits parallelism by creating a few tasks while a long prefix creates redundant work and useless tasks. It is challenging to find the optimal decomposition especially that tasks workloads are variable and unknown before execution, as shown in Figure 2.

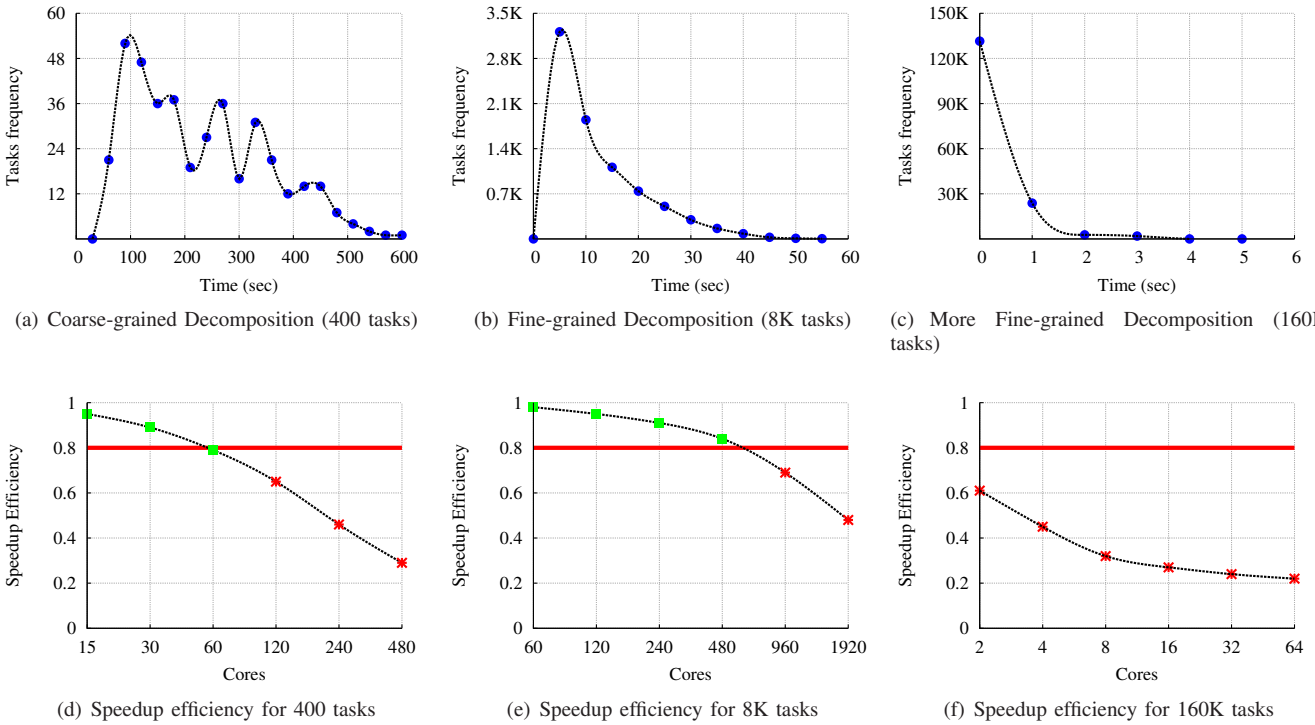
III. APlug Modeling Foundation

A framework for automatic tuning needs to estimate, with high precision, the expected parallel execution time and speedup efficiency using a certain number of cores from a specific system. Typically, regression analysis is done for each individual parallel system to model the relationships among query workload, number of nodes, and speedup efficiency. This makes the integration difficult and slow to accomplish. This section highlights our investigation to develop a generic model that can be applied to a wide range of parallel applications.

A. Modeling Using Workload Frequency Distributions

A workload frequency distribution depicts the workload density of different sets of tasks and the degree of skewness among them. Both aspects affect the parallel workload balance. While workload frequency distributions are discrete, we represent them with continuous curves for visual convenience. We analyzed the workload frequency distributions of several workloads from static and dynamic task decompositions. Due to the limited space, we present our results from molecular docking for static decompositions and pattern mining for dynamic decompositions.

Static process decomposition: Following our example from Section I, docking four subsets of lead-like molecules against Thermus thermophilus gyrase B resulted in different speedup



(a) Coarse-grained Decomposition (400 tasks) (b) Fine-grained Decomposition (8K tasks) (c) More Fine-grained Decomposition (160K tasks)

(d) Speedup efficiency for 400 tasks (e) Speedup efficiency for 8K tasks (f) Speedup efficiency for 160K tasks

Fig. 4. Decomposing our example process to 400 large tasks leads to significant load imbalance. Consequently, the speedup efficiency drops below 0.8 when using more than 60 cores. Decomposing our example process to 8,000 small tasks leads to a nearly optimal load balance. High speedup efficiencies are maintained up to 500 cores. Decomposing our example process to 160,000 tiny tasks may lead to better load balance. Nevertheless, speedup efficiency is poor even at 2 cores because most of the 160,000 tasks are useless and would have been pruned given a coarser decomposition.

efficiencies. Figure 1 shows the speedup efficiencies, where DS3 is best and DS2 is worst. Figure 3 illustrates the workload frequency distributions of DS2 and DS3.

The tasks from DS2 have an irregular workload frequency distribution with many "heavy" tasks. Over half of the 2,000 tasks run in more than 200 seconds. It is hard to balance the workload of DS2 tasks since the probability of a large task being executed last is very high, rendering most cores idle towards the end. In contrast, only 67 DS3 tasks run in more than 200 seconds with most tasks finishing in less than a minute. Hence, most of the cores will be busy processing a large number of small tasks, i.e., workload is balanced.

Dynamic process decomposition: We use ACME [3] on a Blue Gene/P supercomputer to mine a protein sequence for patterns. Prefixes of different lengths (2 to 4) are used to decompose the combinatorial search space tree to 400, 8,000, and 160,000 tasks. Next, we study the workload frequency distribution for each decomposition, and the relationship between the number of cores and speedup efficiency.

The workload frequency distribution when the query is decomposed to 400 tasks is irregular with many "heavy" tasks, as shown in Figure 4(a). For instance, there are about 70 tasks that run in less than 100 seconds, but there are also around 130 tasks that need more than 300 seconds; some extreme cases need more than 500 seconds. Even with dynamic scheduling, balancing such workload on a parallel system is challenging. The speedup efficiency using this decomposition is poor, as shown in Figure 4(d).

When the same pattern mining query was decomposed to 8,000 tasks, the workload frequency distribution changed dramatically as shown in Figure 4(b). While we do not know the processing time of tasks beforehand, we expect their execution times to decrease monotonically as they are further decomposed. Indeed, the figure shows that the majority of tasks run in around 5 seconds, whereas very few need more than 40 seconds. Consequently, there are enough small tasks to keep all cores busy while the few larger ones are executed. Since the probability of a large task executing last is low, we expect good load balance. Figure 4(e) shows the speedup efficiency using this decomposition. Observe that ACME scales well up to about 500 cores using this decomposition, almost an order of magnitude more compared to Figure 4(d).

It is tempting to generate a finer decomposition in order to scale to more cores. Figure 4(c) shows the workload frequency distribution of our pattern mining query decomposed to 160,000 tasks. The graph resembles a power-law distribution. Out of the 160,000 generated tasks, very few take 3 to 5 seconds, whereas the vast majority (i.e., around 130,000 tasks) execute in time close to zero. Unfortunately, most of these tasks are of useless work that accumulates as overhead. They represent areas in the search space that would have been pruned if a coarser decomposition was used. Figure 4(f) shows the speedup efficiency in this case. ACME cannot scale efficiently using this decomposition, not even to 2 cores.

B. Degree of Parallelism and Decomposition

There is a strong correlation between the speedup efficiency of a process and its workload frequency distribution. Our results for pattern mining (dynamic decomposition) are consistent with our molecular docking results (static decomposition). They follow the same trend, where similar task workload frequency distributions lead to similar speedup performance.

Degree of parallelism: APlug estimates the number of cores: (*i*) that satisfy user constraints (time or budget) and (*ii*) at which a parallel system achieves a minimum threshold of speedup efficiency. APlug simulates parallel execution to predict the parallel time when using a specific number of cores. Our models reflect the scheduling mechanism adopted by a parallel system. For example, a single-queue multiple-server model is used to simulate centralized dynamic scheduling. Using our predicted serial and parallel times, we predict speedup efficiency using Equation 1. Our models capture workload skewness and achieve high accuracy by drawing a random sample of the workloads and approximating the actual workload distribution.

Decomposition: APlug aims at automatically specifying the near-optimal parameter value for dynamic task decomposition. Therefore, we analyzed the main factors that determine the quality of a decomposition leading to the efficient utilization of resources. Our findings show that a parallel system can efficiently utilize as many cores as possible for a set of tasks, if there are: (*i*) enough tasks per core to achieve high utilization; (*ii*) few or no useless tasks to avoid overhead; and (*iii*) few or no heavy tasks to avoid workload imbalance. If a heavy task is scheduled last, performance suffers. However, APlug will choose a decompositions with many small tasks, where even the few bulky tasks are not heavy. In Figure 4(b), the heavy tasks are an order of magnitude lighter than the tasks in Figure 4(a). This minimizes the negative effects of scheduling a relatively heavy task last and reduces the probability that this would happen.

We found that a leptokurtic and positively skewed non-symmetric distribution reflects the three aforementioned properties. APlug framework provides an automatic tuning model that guarantees a minimum threshold for speedup efficiency. Assuming the given threshold is 0.8, Figure 4(d) shows that this particular decomposition does not allow this process to scale efficiently to more than 60 cores. If more cores are used the total execution time will decrease, but due to load imbalance many cores will be underutilized. In our example, if instead of 60 cores we use 480 cores (i.e., 8x increase), the total execution time drops from 30 minutes to 10 minutes (i.e., only 3x improvement).

IV. The APlug Framework

APlug is a practical and easy to plug framework due to our novel idea of automatic tuning based on modeling workload frequency distributions. The performance of parallel systems is significantly affected by the workload density and skew in a set of tasks, as discussed in Section III. Our novel idea facilitates the shift toward truly integrated estimation models, where regression analysis per parallel system is avoided. APlug infers the workload frequency distribution to: (*i*) adapt the parallelism of the scientific applications that vary dynamically,

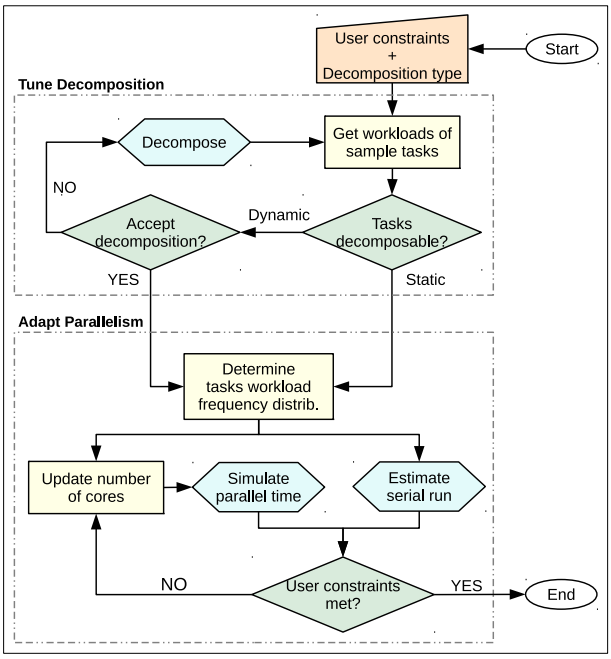


Fig. 5. A simple flowchart of the APlug framework.

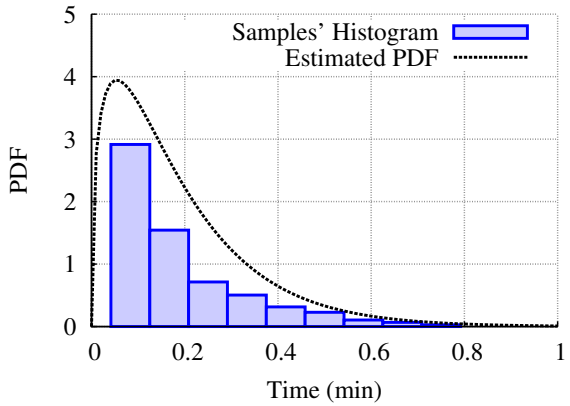


Fig. 6. Estimated probability density function (PDF) for tasks workload frequency and the histogram from the actual execution times of 160 sample tasks.

and (*ii*) tune dynamic process decompositions automatically. This section presents the workflow and algorithms of APlug.

A. The Framework Overview

An abstract workflow diagram of APlug is shown in Figure 5. APlug's inputs are user constraints and task decomposition type. A random sample of tasks from the entire dataset is executed to find the sample workload (i.e., runtimes). The decomposition type is either static or dynamic. For a static decomposition, APlug directly adapts the parallelism. For a dynamic decomposition, APlug automatically tunes the decomposition by finding the decomposition parameter leading to the the highest scalability in terms of number of cores while the utilization threshold is satisfied.

B. Parameterisation of workload frequency distributions

The curve of the workload frequency distribution takes different shapes for different sets of tasks. The gamma distribution is well known by its shape flexibility and the ability to give good approximations of workload frequency distributions [21]. Moreover, our analysis in Section III shows that the workload frequency distribution of an optimal decomposition follows a positive gamma distribution. Hence, APlug adopts the gamma probability density function (PDF) to predict the workload (runtime) per task. In Section VI, we show that APlug, based on the gamma distribution, achieves accurate estimations (with average error below 10%) with distributions of different shapes.

A gamma distribution Γ is characterized by a shape parameter α and a scale parameter β. We use the sample to calculate approximations for the mean μ_Γ and standard deviation σ_Γ of Γ. Then, we calculate α and β as follows [22]:

$$\alpha = \frac{\mu_\Gamma^2}{\sigma_\Gamma^2}, \qquad \beta = \frac{\mu_\Gamma}{\alpha} \tag{2}$$

The probability density function (PDF) of Γ is defined as:

$$\Gamma(x;\alpha,\beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{(\alpha-1)!} \tag{3}$$

As an example, consider the same settings as in Figure 4(b). We draw a random sample of 160 tasks from the 8,000 tasks. Figure 6 depicts the PDF we arrive to using the samples histogram. Observe that the PDF of Γ resembles closely the actual workload frequency distribution in Figure 4(b). Let $\Lambda(t_i, t_j)$ be the expected number of tasks (in the entire space for a given $\mathcal{D}$) with runtime between t_i and t_j. Let $|\mathcal{D}|$ be the number of tasks generated by decomposition parameter $\mathcal{D}$. Given Γ, Λ is calculated as follows:

$$\Lambda(t_i, t_j) = |\mathcal{D}| \int_{t_i}^{t_j} \Gamma(x;\alpha,\beta)\, dx \tag{4}$$

C. Degree of Parallelism Estimation

APlug furnishes users with an accurate estimation of the minimum amount of resources required to process a query within specific constraints. User constraints may involve the maximum allowed execution time; maximum amount of core hours, if the system is deployed in a typical shared research computing infrastructure; or a limit on the financial cost, if a commercial cloud computing provider is used. Moreover, a threshold for minimum speedup efficiency is to be given.

Algorithm 1 describes the degree of parallelism model. It takes the execution time of each of the tasks in the random sample and the user constraints as input, and outputs the number of cores to use, together with the estimated time and speedup efficiency. APlug estimates serial and parallel execution to calculate the expected speedup efficiency at a certain amount of cores, as defined in Equation 1. The serial time $\mathcal{T}_1$ is the summation of the the execution times of all tasks. The lower bound of runtime for a task is zero, but the upper bound is unknown. Let x be an integer time unit. Then $\mathcal{T}_1$ is defined as:

Input: Sample times $sample\_t$, $user\_inpt$
Output: Suggested no. of cores $\mathcal{C}_p$, estimated parallel time $\mathcal{T}_\mathcal{C}$

1 *// estimate PDF from sample execution times*
2 $\alpha \leftarrow (\text{MEAN}(sample\_t)/\text{STDEV}(sample\_t))^2$
3 $\beta \leftarrow \text{MEAN}(sample\_t)/\alpha$
4 *// predict remaining workload*
5 **if** $user\_inpt.\mathcal{T}_{elapsed} > 0$ **then**
6 $\quad$ SETUPQUEUE($user\_inpt.\mathcal{T}_{elapsed}, user\_inpt.C_{curr}$)
7 $\quad$ $user\_inpt.Tasks_{total} \leftarrow$ SIMULATEQUEUE($sample\_t$)
8 *// predict serial time left*
9 $\mathcal{T}_1 \leftarrow \sum_{t=0}^{\infty} (\frac{2t+1}{2}\Lambda(t, t+1))$
10 *// predict parallel time and utilization*
11 $\mathcal{C}_p \leftarrow user\_inpt.C_{max}$
12 **while** $user\_inpt \neq$ TRUE **do**
13 $\quad$ DECREMENT($\mathcal{C}_p$)
14 $\quad$ SETUPQUEUE($user\_inpt, \mathcal{C}_p$)
15 $\quad$ $(\mathcal{T}_\mathcal{C}, \mathcal{C}_p) \leftarrow$ SIMULATEQUEUE($sample\_t$)

Algorithm 1: DEGREE OF PARALLELISM ALGORITHM

$$\mathcal{T}_1 = \sum_{x=0}^{\infty} \frac{2x+1}{2}\Lambda(x, x+1) \tag{5}$$

We employ the queuing theory [23] to estimate the parallel execution time $\mathcal{T}_\mathcal{C}$. We model the parallel process as a finite-source queue of n tasks served by $\mathcal{C}$ servers (i.e., cores). Without loss of generality, we assume homogeneous servers. Since our population is finite, numerically simulating the queue provides an accurate representation of the real system [24]. APlug implements a discrete event simulator. We start with all tasks in the queue. The workloads of the tasks follow the workload frequency distribution of our sample tasks. Equation 4 is used to create bins of tasks. The servers randomly consume tasks from different workload bins until all bins are empty. The output of the simulator is our estimation for the parallel execution time $\mathcal{T}_\mathcal{C}$.

Given the expected performance variability on public clouds [6], users should be able to reevaluate the situation online and adapt accordingly. Lines 4 to 7 allow our model to consider elapsed time to account for the expected remaining workload only. APlug utilizes the workload frequency distribution to: (*i*) estimate the remaining tasks, and (*ii*) suggest the amount of cores to add or remove online. The output of our model can be used in many ways, such as predicting accurately the expected financial cost. We present such a case study in Section VI.

D. Automatic Decomposition

APlug's automatic decomposition feature is optional. Its goal is to solve the following optimization problem: Find the

Input: Query inputs $\mathcal{I}$ and ; threshold SE_{min}
Output: optimal decomposition param $\mathcal{D}$; no. of cores $\mathcal{C}_{max}$

1 $\mathcal{D} \leftarrow \mathcal{I}.\mathcal{D}_{min}$
2 $\mathcal{C}_{max} \leftarrow 1$
3 **while** *Decomposable* **do**
4 *// randomly draw x tasts of param $\mathcal{D}$*
5 $sample \leftarrow$ RANDOMTASTS$(x, \mathcal{D})$
6 $sample\_times \leftarrow$ GETWORKLOADS$(sample)$
7 $t\mathcal{C} \leftarrow$ ESTSPDUPEFF$(sample_times, SE_{min})$
8 **if** $t\mathcal{C} < \mathcal{C}_{max}$ **then**
9 **break**
10 **else**
11 INCREMENT$(\mathcal{D})$
12 $\mathcal{C}_{max} \leftarrow t\mathcal{C}$

Algorithm 2: AUTOMATIC DECOMPOSITION ALGORITHM

process decomposition that maximizes scalability (i.e., number of cores) under the constraint that speedup efficiency SE is over a user specified threshold SE_{min}. We solve this problem as follows: (*i*) Partition the process at a specific parameter value and draw a random sample of tasks to run. (*ii*) Estimate the expected speedup efficiency SE. (*iii*) Repeat these steps until we find the decomposition parameter that allows us to scale to the largest number of cores with $SE \geq SE_{min}$. Our workload analysis indicates that the best decomposition should not just provide enough tasks to use more cores but should also consider the overheads of parallelism to keep efficiency high. Hence, APlug will indicate to the users the maximum number of cores they can use for a certain workload ensuring high utilization of resources.

Algorithm 2 describes our automatic tuning method. For simplicity, we assume the decomposition parameter is atomic. In line 1, $\mathcal{D}$ is initialized to the minimum decomposition parameter value $\mathcal{I}.\mathcal{D}_{min}$. To reduce the overhead of the automatic tuning method, sample tasks can be generated and evaluated in parallel (i.e., lines 5 and 6). In practice, the main loop of the algorithm is executed only a few times before finding a near-optimal decomposition. If the process is not decomposable using $\mathcal{D}$ the algorithm returns the last working decomposition parameter. Function ESTSPDUPEFF in line 7 is the heart of the algorithm. Given a decomposition, for a specific number $\mathcal{C}$ of cores, it estimates the expected speedup efficiency. The function iterates over a range of values for $\mathcal{C}$ and returns the one that achieves the maximum SE for the given decomposition. Algorithm 2 finds the best decomposition with a certain margin of error. The sample size is chosen to bound this error using the equation $n >= ((z/e)^2)/4$, where z is the z-score for the confidence interval and e is the desired margin of error [25].

V. APLUG API AND INTEGRATION

The complete C++ source code of APlug, including its standalone utility program, is implemented in less than 800 lines of code. The framework is available for download[2] and as a public Amazon Machine Image[3]. This section highlights the most important aspects of our implementation. We present the main C++ API and discuss the integration of APlug in scientific applications. Most APlug methods shown in Figure 7 are implemented and ready to use with any application.

```
class APlug {
 public:
   void setNumOfTasks(int numOfTasks);
   void setSampleSize(int sampleSize);
   void loadSamples(double[] sampleTimes);
   int changeDecomposition(int last);
   int getRecCores(double tLimit, int cLimit);
   double getSerialTime(void);
   double getParallelTime(int cores);
   double getSpeedupEff(int cores);
};
```

Fig. 7. The APlug C++ API foundation.

A typical scenario for integrating APlug with an application starts by including the APlug class and overriding `changeDecomposition()`, if needed. Users with an application of a fixed number of tasks do not need to implement or use this method. If the process can be decomposed dynamically, APlug needs to know how different decompositions are achieved. For example, decomposing an application with a tree-based search space involves using prefixes to partition the search space into sub-trees.

APlug uses sample tasks to predict workloads and adapt parallelism accordingly. The user needs to execute a random sample of tasks and pass their workloads to the framework using `loadSamples()`. The sampled tasks can (and should) be part of the final results. Calling the method `getRecCores()` provides the number of cores needed to meet user constraints. The rest of the framework methods are provided for users to use them as needed and to customize their experience.

We integrated APlug with the three scientific applications discussed in Section II. We did not reuse the samples in order to minimize code modification. Note that the overhead of running the sample is minimal compared to the gain from tuning. In the case of integrating APlug with VinaLC and P-SSW, only 20 lines of code were added or changed in their source codes. The method `changeDecomposition()` was not used because the number of tasks is fixed. In the case of ACME, the process decomposition is not fixed. We integrated APlug with ACME in less than 50 lines of code including overriding the method `changeDecomposition()`.

The APlug framework comes with a standalone utility program. This is useful in cases where the application source code is not available or if code integration is not preferred. The APlug utility accepts a file of sample tasks workloads and takes user constraints as input. The utility will output useful statistics including expected execution times, speedup efficiencies, and recommended number of cores.

[2]http://cloud.kaust.edu.sa/pages/aplug.aspx
[3]Amazon Machine Image ID: ami-df556bb6

VI. Evaluation

Our APlug framework and the datasets detailed below are available for download online[2] and on Amazon EC2[3]. We evaluate APlug in the three different scientific applications discussed in Section II. The accuracy of APlug's predictions and APlug's minimal overhead are shown. We conduct sensitivity analysis experiments to study the effects of the sample size on APlug. The effective scalability of the applications after integrating APlug with them is then demonstrated.

A. Experimental Setting

1) Datasets: We used real datasets to test APlug. (*i*) For molecular docking, similar to the authors of VinaLC, we dock lead-like compounds from the ZINC database [26] against Thermus thermophilus gyrase B[4]. (*ii*) For sequence alignment, 71,501 human protein sequences[5] and 65,685 random shotgun sequences[6] of Shewanella oneidensis bacteria [27]. (*iii*) For pattern mining, 2.6GB DNA[7] for the entire human genome.

2) Systems: We used various systems with different architectures. Namely, we used a supercomputer, a Linux cluster, and a public cloud cluster. The supercomputer is an IBM Blue Gene/P with 16,384 quad-core PowerPC processors @850MHz with a total memory of 64TB. The Linux cluster is an HP system of 480 cores @2.1GHz with each 24 cores sharing 148GB of RAM. The public cloud cluster was rented from Amazon EC2 and consisted of 40 on-demand M3 large instances[8]. Each instance had 2 cores and 7.5GB RAM.

3) Queries: For the molecular docking experiments, we randomly chose 10,000 lead-like compounds from the ZINC dataset to dock against Thermus thermophilus gyrase B. For sequence alignments, we created over 1 million tasks (1,000,405 pairwise alignments) using 1,415 random Human protein sequences. All the experiments with VinaLC and P-SSW were run on the Linux cluster. VinaLC could not run on the supercomputer because of library incompatibilities and P-SSW uses architecture specific SIMD operations. In the case of pattern mining, the number of tasks is not fixed and we test our decomposition tuning. On EC2, we chose a light pattern mining query that ends in reasonable time. APlug partitioned the search space tree of this query to 4,096 tasks (using prefixes of length 6). For the supercomputer, a pattern mining query that has enough workload to scale to thousands of cores was used. This query was decomposed automatically by APlug to 262,144 tasks (using prefixes of length 9).

4) Baseline: The baseline implementation that we compare to is the naïve solution to the degree of parallelism problem. Given the runtimes of a sample of the tasks, the serial time is estimated by multiplying the average sample runtime by the total number of tasks. The parallel runtime is then calculated by dividing the estimated serial time by the number of cores. The samples used are the ones used with APlug and in the case of decomposition tuning we use APlug's decomposition.

TABLE I. The baseline method provides fairly accurate parallel time estimations when tasks workloads are uniform. In this experiment, P-SSW is used to alig 1,415 random DNA shotgun sequences from the Shewanella oneidensis bacteria. Most shotgun sequences have similar lengths and tasks workloads are close to uniform.

Cores	Actual time (hours)	APlug error (%)	Baseline error (%)
16	45	0.8	0.9
32	23	0.8	0.8
64	12	0.8	0.9
128	6	0.9	1.0
256	3	0.9	1.0

TABLE II. Accuracy of APlug and the baseline compared to actual parallel times of P-SSW on the Linux cluster.

Cores	Actual time (hours)	APlug error (%)	Baseline error (%)
16	145	0.1	7.4
32	73	0.1	7.5
64	36	0.2	7.5
128	18	0.2	7.5
256	9	0.3	7.5

B. Accuracy of APlug

It is not difficult to predict runtimes of highly scalable applications when tasks have similar workloads. Indeed, Table I shows that the baseline method achieves fair accuracy in predicting the runtimes of P-SSW up to 256 cores. The query was to align 1,415 shotgun sequences from the Shewanella oneidensis bacteria dataset. Over a million tasks are run but most of the alignments have similar workloads because the sequences are of similar lengths. While we do not know this fact in advance, it is risky to use the baseline method. Generally, it is not common to have queries with uniform workload frequency distributions in practice. Next, we show queries for sequence alignment, molecular docking, and pattern mining where tasks workloads are skewed, which is the common case.

We verify the accuracy of APlug and compare it to the baseline method on different architectures. For each application, we run the same query using different numbers of cores and compare the actual runtimes with the estimations from APlug and the baseline method. The query for Table II has the same number of tasks as in Table I only this time we align sequences from the human protein dataset, where sequences are of variable lengths leading to skewed task workloads. APlug captures workload skewness to better estimate runtimes and accurately predict speedup efficiencies. Similarly, Tables III and IV show that APlug significantly outperforms the baseline method.

Figures 8, 9, and 10 show the actual workload frequencies and the PDFs APlug used to predict them. Since we care about the total runtime, the shapes of the actual workload frequencies and the PDFs need not be similar. The total runtime is a function of the area under the curves. The values of the corresponding integrals are similar, leading to low estimation errors. APlug achieves this by implicitly considering the infrastructure performance properties using the runtimes of the sample tasks.

[4] http://www.rcsb.org/pdb/explore.do?structureId=1KIJ
[5] ftp://ftp.ncbi.nih.gov/refseq/H_sapiens/mRNA_Prot/human.protein.faa.gz
[6] ftp://ftp.cbcb.umd.edu/pub/data/asmg_benchmark/
[7] http://webhome.cs.uvic.ca/ thomo/HG18.fasta.tar.gz
[8] http://aws.amazon.com/ec2/instance-types/

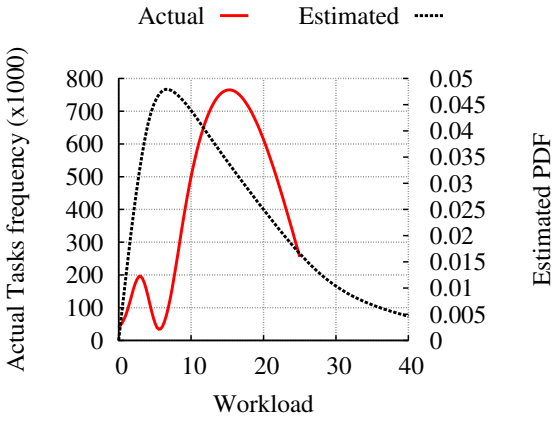


Fig. 8. P-SSW query from Table II comparing actual workload frequency distribution and PDF used by APlug.

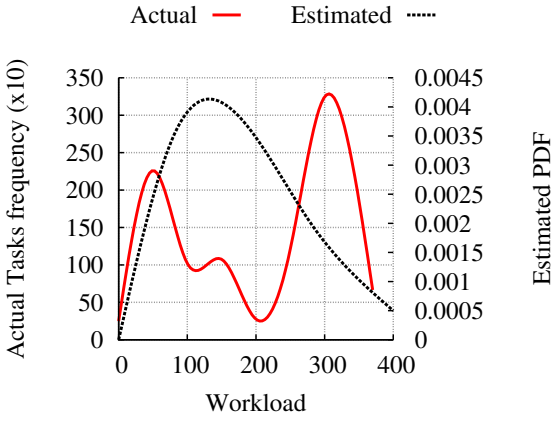


Fig. 9. VinaLC query from Table III comparing actual workload frequency distribution and PDF used by APlug. While a single Gamma distribution can not have the shape of the actual workload frequency distribution, it captures workload skewness and approximates the total runtimes accurately.

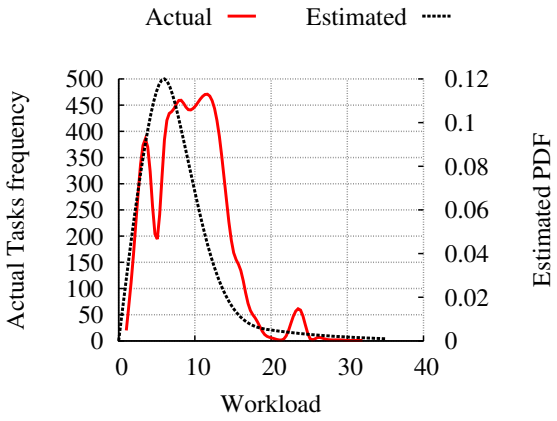


Fig. 10. ACME query from Table IV comparing actual workload frequency distribution and PDF used by APlug.

TABLE III. ACCURACY OF APLUG AND THE BASELINE COMPARED TO ACTUAL PARALLEL TIMES OF VINALC ON THE LINUX CLUSTER.

Cores	Actual time (hours)	APlug error (%)	Baseline error (%)
16	31	0.4	8.2
32	16	0.4	8.1
64	8	0.4	7.8
128	4	0.4	7.0
256	2	0.3	5.3

TABLE IV. ACCURACY OF APLUG AND THE BASELINE METHOD COMPARED TO ACTUAL PARALLEL TIMES OF ACME ON AMAZON EC2.

Cores	Actual time (minutes)	APlug error (%)	Baseline error (%)
4	130	0.9	17.8
8	66	0.9	17.9
16	33	0.8	18.1
32	17	1.2	18.8
64	9	1.8	20.3

The next experiment shows the accuracy of APlug decomposition tuning. Table V shows the speedup efficiencies when different decompositions are used for our query. APlug chooses the decomposition that results in the best speedup efficiency using different numbers of cores. However, the default decomposition method of ACME fails in most cases because it considers number of cores to decompose the problem.

Accurate decomposition and runtime prediction saves money for users of supercomputing centers and clouds. Our next experiment shows how APlug guides user decisions to stick to their budget. We decompose and adapt the parallelism of a pattern mining query given a pricing scheme. The user needs to mine human DNA for patterns in less than 3 hours and without spending more than $20. Table VI shows that renting 10 instances meets the budget but not the time constraint. Similarly, renting 30 instances meets the time constraint but not the budget. It is interesting that renting more instances, in this case 40, meets both constraints. It is not straightforward to arrive at this conclusion without utilizing APlug capabilities.

During execution, APlug guides users if scaling out or in is desired. Table VII shows that APlug provides good expected runtimes and speedup efficiencies online. The error in APlug's online predictions is slightly higher than its initial run because

TABLE V. THE DIFFERENT SPEEDUP EFFICIENCIES OF ACME ON THE SUPERCOMPUTER USING DIFFERENT DECOMPOSITIONS. APLUG CONSISTENTLY CHOOSES THE BEST DECOMPOSITION (256K TASKS).

Cores	Speedup Efficiency wrt #Tasks			
	16K	64K	256K	1024K
512	0.94	0.97	0.98	0.81
1,024	0.87	0.97	0.97	0.83
2,048	0.83	0.92	0.97	0.83
4,096	0.46	0.76	0.92	0.76
8,192	0.25	0.46	0.76	0.46

TABLE VI. ACCURACY OF THE ESTIMATED PARALLEL TIME (AND COST) OF ACME ON AMAZON EC2 CLOUD. APLUG IS ABLE TO PREDICT TIME SO USERS CAN MEET BUDGET AND TIME CONSTRAINTS.

	Number of Amazon EC2 Instances				
	1	10	20	30	40
Cores	2	20	40	60	80
Cost/Hour	$0.24	$2.40	$4.80	$7.20	$9.60
Est. Time	2.5 Days	5.9 Hr	3.1 Hr	2.1 Hr	1.5 Hr
Act. Time	2.9 Days	5.1 Hr	4.3 Hr	2.3 Hr	1.6 Hr
Est. Cost	$14.40	$14.40	$19.20	$21.60	$19.20
Act. Cost	$16.80	$14.40	$24.00	$21.60	$19.20

TABLE VII. APLUG PROVIDES USERS WITH ACCURATE PREDICTIONS FOR ADDING (OR REMOVING) RESOURCES ONLINE. IN THIS EXPERIMENT 2,000 VINALC TASKS ARE STARTED WITH 130 CORES THEN AFTER HALF THE TASKS WERE DONE, APLUG WAS CONSULTED TO ADD MORE CORES.

Additional Cores	Total Time (min)		Speedup Efficiency	
	Actual	Predicted	Actual	Predicted
0	33.23	34.43	0.95	0.92
100	28.85	27.78	0.62	0.64
200	28.17	25.13	0.44	0.49
300	26.80	24.58	0.35	0.38

TABLE VIII. SENSITIVITY ANALYSIS OF THE SAMPLE SIZE USED IN APLUG TO ESTIMATE THE EXECUTION TIME OF VINALC USING 256 CORES ON THE LINUX CLUSTER. THE ACTUAL TIME WAS OVER 2 HOURS. THE SAMPLE MARGIN OF ERROR IS BETWEEN 9% AND 3%.

Sample size (%)	Sample time (minutes)	APlug error (%)	Baseline error (%)
1	2	8.0	13.9
2	3	4.7	10.9
4	5	0.4	5.3
8	10	0.3	4.2

it is difficult to capture the execution of the tasks that are running during online prediction. Initially, 130 cores were used. Additional cores were added after half the tasks were done. Counterintuitively, adding cores at this stage negatively affects speedup efficiency since the remaining tasks do not have enough workload to fully utilize a larger number of cores. In this experiment, adding 300 cores improves the time by less than 7 minutes at the cost of a drop in efficiency from 0.95 to 0.35. APlug accurately provides online facts for users to make informed decisions.

C. Sensitivity Analysis for APlug

APlug is expected to be sensitive to the sample size because it uses sample tasks to build its workload model. We use random sampling, the simplest form of probability sampling, where all tasks have the same probability of being in the sample. Random sampling is preferred in cases where the information we know about the total population (i.e., tasks) is little. Statistically, the precision of the estimator given a large number of total tasks depends on sample size, and not sample percentage of the total tasks [28]. For example, the precision of a random sample of 100 tasks from 100,000 tasks and from a million tasks is the same. For each experiment in this section, we note the sample margin of error range for a 95% confidence interval.

Tables VIII, IX, and X show that practically acceptable accuracy is achieved with small sample sizes. The overhead of running samples increases with sample size but is not steep. The overhead is basically the runtimes of the sample tasks, which is dependent on sample workload not only sample size. The time it takes to run the sample tasks is orders of magnitude lower than the total runtime. The accuracy of APlug is significantly better than the baseline even for small samples because APlug captures the workload skewness and irregularity of tasks workloads.

TABLE IX. SENSITIVITY ANALYSIS OF THE SAMPLE SIZE USED IN APLUG TO ESTIMATE THE EXECUTION TIME OF P-SSW USING 256 CORES OF THE LINUX CLUSTER. THE ACTUAL TIME WAS OVER 9 HOURS. THE SAMPLE MARGIN OF ERROR IS BETWEEN 1% AND 0.3%.

Sample size (%)	Sample time (minutes)	APlug error (%)	Baseline error (%)
1	5	0.5	7.2
2	11	0.6	7.1
4	22	0.2	7.5
8	44	0.1	7.7

TABLE X. SENSITIVITY ANALYSIS OF THE SAMPLE SIZE USED IN APLUG TO ESTIMATE THE SERIAL EXECUTION TIME OF ACME ON AMAZON EC2 CLOUD. THE ACTUAL TIME WAS OVER 8.5 HOURS. THE SAMPLE MARGIN OF ERROR IS BETWEEN 15% AND 5%.

Sample size (%)	Sample time (minutes)	APlug error (%)	Baseline error (%)
1	6	13.5	42.0
2	11	11.5	32.4
4	21	1.2	10.8
8	41	1.4	10.2

TABLE XI. SENSITIVITY ANALYSIS OF SAMPLE SIZE IN ACME'S DECOMPOSITION TUNING. BEST DECOMPOSITION EMPIRICALLY FOUND TO BE 262,144 TASKS. THE SAMPLE MARGIN OF ERROR IS BETWEEN 21% AND 7%.

Sample Size (# Tasks)	Tuning cost (seconds)	Suggested decomposition (# Tasks)
10	9	16,384
20	11	16,384
40	7	65,536
80	4	262,144
160	6	262,144
320	12	262,144

The next experiment studies APlug's decomposition tuning with respect to different sample sizes. We empirically find the best decomposition by exhaustively using different prefix lengths to partition the search space tree of a pattern mining query. Decomposing the example process to 262,144 tasks creates the most fine-grained tasks with minimal useless work. Table XI shows that APlug finds the optimal decomposition with small sample sizes. The tuning time is minimal, especially when compared to the full query execution time.

D. Effective Scalability using APlug

Guided by APlug; P-SSW, VinaLC, and ACME adapt their parallelism to achieve high speedup efficiencies with minimal overhead. Tables XII, XIII, and XIV show that APlug is able to accurately adapt the parallelism of different applications on different architectures. Due to time constraints, speedup efficiencies are calculated according to a 15-core system for VinaLC and P-SSW and to a 256-core system for ACME. No manual tuning for ACME on the supercomputer was needed as its decomposition is automatically tuned by APlug. VinaLC was not run on the supercomputer because of library incompatibilities. P-SSW uses architecture specific SIMD operations that prevents it from running on the supercomputer.

TABLE XII. APLUG ACCURATE PREDICTION FOR VINALC ON THE LINUX CLUSTER.

Cores	Predicted Speedup Efficiency	Actual Speedup Efficiency
15	1.00	1.00
30	0.99	0.98
60	0.99	0.97
120	0.99	0.95
240	0.98	0.94
480	0.96	0.93

TABLE XIII. APLUG ACCURATE PREDICTION FOR P-SSW ON THE LINUX CLUSTER.

Cores	Predicted Speedup Efficiency	Actual Speedup Efficiency
15	1.00	1.00
30	0.99	0.98
60	0.99	0.98
120	0.99	0.96
240	0.98	0.95
480	0.97	0.93

TABLE XIV. ACME IS ABLE TO SCALE TO 16,384 CORES WITH HIGH SPEEDUP EFFICIENCY ON THE SUPERCOMPUTER USING APLUG'S AUTOMATIC DECOMPOSITION TUNING. APLUG ACCURATELY PREDICTS SPEEDUP EFFICIENCY OF ACME USING THOUSANDS OF CORES.

Cores	Predicted Speedup Efficiency	Actual Speedup Efficiency
256	1.0	1.00
1,024	0.99	0.99
2,048	0.99	0.98
4,096	0.98	0.96
8,192	0.98	0.91
16,384	0.97	0.98

VII. RELATED WORK

Automatic tuning is needed for applications that handle big data and scale out on large infrastructures. The de facto application is data analytics using MapReduce or MPI-based systems. In MapReduce frameworks, data analytics is accomplished using multiple iterations. PREDIcT [29] introduced an experimental methodology for estimating the number of iterations and the time of each iteration in iterative analytics. PREDIcT does not estimate the number of machines required to meet user-specific constraints or suggest the best decomposition into a certain number of iterations. Both problems are challenging due to variance in iteration times. We address these problems for bag-of-tasks MPI-based applications, where the variance among tasks is even higher. Starfish [30] automatically tunes a Hadoop cluster to enhance its performance for data analytics. It uses dynamic instrumentation to profile jobs and a what-if engine to predict performance. Cumulon [31] is aimed specifically at statistical analysis on public clouds. Our work targets different large-scale infrastructures, such as supercomputers, and suggests resources based on workload.

Resource allocation and scheduling are to provision resources and balance the load on them. Middleware solutions, such as Falkon [32], were used to manage resources for applications with many-tasks. Recent work provided middleware solutions with cost-efficient algorithms for tasks assignment across multiple clouds [33]. Middleware solutions are infrastructure and provider specific. Applications require code modification to utilize such services. To deal with trailing tasks, the number of workers can be shrunk at some point to keep utilization high [34]. BaTS [35] replicates trailing tasks on idle cores to increase the chances of completing them faster. Resource allocation is done by service providers and scheduling is done either by applications or middleware. In this paper, we consider these problems from a user point of view. Our work is orthogonal to resource allocation and scheduling as we decide the appropriate number of cores to use for an instance of a parallel application.

Runtime and scalability estimation are used to assist the decision making processes of schedulers and resource management modules [36]. One way of estimating task runtime is to analyze and profile code [37]. Compiler knowledge can be used to extract performance metrics [38]. Full code needs to be analyzed in order to create and validate scalability predictive models. The inputs for such models are low level system metrics, such as communication latencies, data type sizes, and memory contention. A more practical approach is to use statistical modeling to estimate runtime and scalability [39]. Bayesian and neural networks were used in grids to learn the execution times of running tasks [40]. Nevertheless, existing execution time estimators for large-scale applications were shown to be ineffective in clouds [41]. We model the workload distribution of tasks using a sample run. Our model does not require code profiling and is not infrastructure specific. APlug is less aggressive and requires no or minimal coding.

Performance modeling is a useful tool to optimize existing systems and design future ones [42]. Works on performance prediction are mostly specific to certain applications and architectures requiring code inspection and instrumentation [43]. Regression techniques were used to estimate the performance of scientific workflows in heterogeneous environments [44]. Recently, a framework for predicting application performance on systems with hardware accelerators was introduced [45]. Our goal is to meet user constraints for a specific run of an bag-of-tasks application. We implicitly capture hardware performance in order to automatically tune the degree of parallelism and problem decomposition.

In this paper, we are not concerned with allocation and scheduling strategies at the infrastructure level. We do not measure nor optimize performance of applications or computing infrastructures. We propose a novel and generic framework to predict the appropriate degree of parallelism for a particular process based on user constraints with minimal overhead.

VIII. CONCLUSION

This paper presented APlug, an automatic tuning framework for large-scale parallel applications with many independent tasks. APlug adapts the degree of parallelism and automatically decomposes the parallel process to enable users to achieve efficient utilization of CPU resources. We studied the correlation between the workload frequency distributions of a set of tasks and the resource utilization. Our models are based on this correlation. Therefore, APlug facilitates the shift towards truly integrated estimation models.

Our experiments show the viability of our framework for molecular docking, sequence alignment, and pattern mining using 16,384 cores in a supercomputer, 480 cores in a Linux cluster, and 80 cores from a public cloud. APlug estimates the serial and parallel times to suggest the best degree of parallelism in very short time with less than 10% error. Users can use APlug to optimize various quantities; such as, execution time and financial cost.

While many commercial and scientific applications have independent tasks, our future work will target tasks with inter-dependencies. It is challenging to predict dependencies because they differ according to data size, decomposition, and execution environment. We attempt to address tasks dependencies by two ways: profiling the code, and collecting data during runtime. Machine learning techniques can then be used to construct models for typical application behaviors.

ACKNOWLEDGMENT

For computer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia.

REFERENCES

[1] X. Zhang, S. E. Wong, and F. C. Lightstone, "Message passing interface and multithreading hybrid for parallel molecular docking of large databases on petascale high performance computing machines," *Journal of Computational Chemistry*, vol. 34, no. 11, 2013.

[2] C. Wu, A. Kalyanaraman, and W. Cannon, "pGraph: Efficient parallel construction of large-scale protein sequence homology graphs," *Parallel and Distributed Systems, IEEE Trans. on*, vol. 23, no. 10, 2012.

[3] M. Sahli, E. Mansour, and P. Kalnis, "Parallel motif extraction from very long sequences," in *Proc. of the ACM Intl. Conf. on Information & Knowledge Management*, 2013.

[4] G. R. Andrews, "Paradigms for process interaction in distributed programs," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, 1991.

[5] S. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. Collazo-Mojica, "A modeling approach for estimating execution time of long-running scientific applications," in *Parallel and Distributed Processing. IEEE Intl. Symposium on*, 2008.

[6] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proc. of the VLDB Endowment*, vol. 3, no. 1-2, 2010.

[7] M. Zhao, W.-P. Lee, E. P. Garrison, and G. T. Marth, "SSW library: An simd smith-waterman c/c++ library for use in genomic applications," *PLoS ONE*, vol. 8, no. 12, 2013.

[8] T. Lengauer and M. Rarey, "Computational methods for biomolecular docking," *Current Opinion in Structural Biology*, vol. 6, no. 3, 1996.

[9] H. Li, A. Liu, Z. Zhao, Y. Xu, J. Lin, D. Jou, and C. Li, "Fragment-based drug design and drug repositioning using multiple ligand simultaneous docking (MLSD): identifying celecoxib and template compounds as novel inhibitors of signal transducer and activator of transcription 3 (stat3)," *Journal of medicinal chemistry*, vol. 54, no. 15, 2011.

[10] A. L. Hopkins and C. R. Groom, "The druggable genome," *Nature reviews Drug discovery*, vol. 1, no. 9, 2002.

[11] P. Kirkpatrick and C. Ellis, "Chemical space," *Nature*, vol. 432, no. 7019, 2004.

[12] O. Trott and A. J. Olson, "AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading," *Journal of comp. chemistry*, vol. 31, no. 2, 2010.

[13] C. Sander and R. Schneider, "Database of homology-derived protein structures and the structural meaning of sequence alignment," *Proteins: Structure, Function, and Bioinformatics*, vol. 9, no. 1, 1991.

[14] D. W. Mount, *Sequence and genome analysis*, 2004.

[15] I. Elias, "Settling the intractability of multiple alignment," *Journal of Computational Biology*, vol. 13, no. 7, 2006.

[16] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, 2007.

[17] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, 1981.

[18] X. Xie, T. S. Mikkelsen, A. Gnirke, K. Lindblad-Toh, M. Kellis, and E. S. Lander, "Systematic discovery of regulatory motifs in conserved regions of the human genome, including thousands of ctcf insulator sites," *Proc. of National Academy of Sciences*, vol. 104, no. 17, 2007.

[19] A. Mueen and E. Keogh, "Online discovery and maintenance of time series motifs," in *Proc. of the ACM Intl. Conf. on Knowledge Discovery and Data Mining*, 2010.

[20] K. Saxena and R. Shukla, "Significant Interval and Frequent Pattern Discovery in Web Log Data," *Intl. Journal of Computer Science Issues*, vol. 7, no. 1(3), 2010.

[21] A. O'Connor, *Probability distributions used in reliability engineering*, 2011.

[22] A. Papoulis and S. U. Pillai, *Probability, random variables, and stochastic processes*, 2002.

[23] L. Kleinrock, *Queueing Systems*, 1975, vol. I: Theory.

[24] D. Meisner and T. F. Wenisch, "Stochastic queuing simulation for data center workloads," in *Exascale Evaluation and Research Techniques Workshop*, 2010.

[25] J. L. Fleiss, B. Levin, and M. C. Paik, *Statistical methods for rates and proportions*. John Wiley & Sons, 2013.

[26] J. J. Irwin and B. K. Shoichet, "Zinc-a free database of commercially available compounds for virtual screening," *Journal of chemical information and modeling*, vol. 45, no. 1, 2005.

[27] J. F. Heidelberg, I. T. Paulsen, K. E. Nelson, E. J. Gaidos, W. C. Nelson, T. D. Read, J. A. Eisen, R. Seshadri, N. Ward, B. Methe *et al.*, "Genome sequence of the dissimilatory metal ion–reducing bacterium shewanella oneidensis," *Nature biotechnology*, vol. 20, no. 11, 2002.

[28] S. Lohr, *Sampling: design and analysis*, 2009.

[29] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki, "PREDIcT: Towards predicting the runtime of large scale iterative analytics," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1678–1689, Sep. 2013.

[30] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics." in *CIDR*, vol. 11, 2011, pp. 261–272.

[31] B. Huang, S. Babu, and J. Yang, "Cumulon: Optimizing statistical data analysis in the cloud," in *Proc. of the 2013 ACM SIGMOD Intl. Conf. on Management of Data*, 2013, pp. 1–12.

[32] I. Raicu, I. Foster, M. Wilde, Z. Zhang, K. Iskra, P. Beckman, Y. Zhao, A. Szalay, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, and D. Thain, "Middleware support for many-task computing," *Cluster Computing*, vol. 13, no. 3, 2010.

[33] M. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya, "Randomized approximation scheme for resource allocation in hybrid-cloud environment," *The Journal of Supercomputing*, 2014.

[34] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster, "Scheduling many-task workloads on supercomputers: Dealing with trailing tasks," in *Many-Task Computing on Grids and Supercomputers, Workshop on*, 2010.

[35] A.-M. Oprescu, T. Kielmann, and H. Leahu, "Stochastic tail-phase optimization for bag-of-tasks execution in clouds," in *Utility and Cloud Computing, Intl. Conf. on*, 2012.

[36] M. Tao, S. Dong, and L. Zhang, "A multi-strategy collaborative prediction model for the runtime of online tasks in computing cluster/grid," *Cluster Computing*, vol. 14, no. 2, 2011.

[37] M. Kiran, A.-H. A. Hashim, L. M. Kuan, and Y. Y. Jiun, "Execution time prediction of imperative paradigm tasks for grid scheduling optimization," *Int J Comput Sci Netw Secur*, vol. 9, no. 2, 2009.

[38] C. Mendes and D. Reed, "Integrated compilation and scalability analysis for parallel systems," in *Parallel Architectures and Compilation Techniques, 1998. Proc.. 1998 Intl. Conf. on*, 1998.

[39] T. Miu and P. Missier, "Predicting the execution time of workflow activities based on their input features," in *High Performance Computing, Networking, Storage and Analysis (SCC), SC Companion*, 2012.

[40] R. Duan, F. Nadeem, J. Wang, Y. Zhang, R. Prodan, and T. Fahringer, "A hybrid intelligent method for performance modeling and prediction of workflow activities in grids," in *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM Intl. Symposium on*, 2009.

[41] J. Delgado, A. Eddin, M. Adjouadi, and S. Sadjadi, "Paravirtualization for scientific computing: Performance analysis and prediction," in *High Performance Computing and Communications, Intl. Conf. on*, 2011.

[42] L. Carrington, M. Laurenzano, and A. Tiwari, "Characterizing large-scale hpc applications through trace extrapolation," *Parallel Processing Letters*, vol. 23, no. 04, 2013.

[43] D. J. Kerbyson, K. J. Barker, D. S. Gallo, D. Chen, J. R. Brunheroto, K. D. Ryu, G. L.-T. Chiu, and A. Hoisie, "Tracking the performance evolution of blue gene systems," in *Proc. of the 28th Intl. Supercomputing Conf.*, 2013.

[44] Q. Wu and V. Datla, "On performance modeling and prediction in support of scientific workflow optimization," in *Services, IEEE World Congress on*, 2011.

[45] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole, "Modeling and predicting performance of high performance computing applications on hardware accelerators," *Intl. Journal of High Performance Computing Applications*, vol. 27, no. 2, 2013.