Locality And Its Usage In Parallel Job Runtime Distribution Modeling Using HMM

by Avi Nissimov,
under Supervision of Dror Feitelson

October 22, 2006

Acknowledgments

First of all, my warmest thanks go to my teacher and supervisor Dror Feitelson, for guiding me thoroughly during all this work. Only his guidelines made possible to cover the spectrum of both practical and theoretical targets in this work.

Special thanks go to Prof. Naftali Tishby, for his assistance in selecting a framework for workload characterization, learning and information theory metrics and tools.

Other two people I wish to thank are the friends from Parallel System Laboratory, Dan Tsafrir and David Talby, for sharing their experience in job runtime predictions and scheduling goals and metrics for performance evaluation.

Finally, this work couldn't be made without permanent support of my wife and the rest of my family.

Contents

Chapter 1

Introduction

1.1 Locality

Locality of reference is one of the best known and widely occurring attributes of computer workloads. It means that if a certain memory location is referenced, there is high probability that it (or a nearby location) will be referenced again soon. This is the basis for the success of all caching schemes that maintain recently-referenced data in high-speed memory for possible reuse, including processor caches and paging of virtual memory. The idea has also been used in other types of storage, from caching data blocks in file systems to caching web pages on desktops, proxies, and servers.

Moreover, systems may use their history in other ways as well, in addition to caching. For example, the Network Weather Service collects data about network usage in order to make predictions regarding future resource availability. Adaptive algorithms can be designed that observe the workload and change the system's behavior accordingly. For example, schedulers and load balancers may adapt to their workload and tune their settings so as to provide the best performance.

This work focuses on employing locality for parallel job runtime prediction, and usage of these predictions in scheduling algorithms of supercomputers. Therefore, the basic question is how much locality exists in the workload of a computer system. The assumption of this thesis is that this question mostly refers to how much can one learn on the present and the nearest future from the recent history. The learnability is measured traditionally in terms of Information Gain. This measures how well a given attribute (in the case of job scheduling, the submit time of a job) can classify the target attribute (this job's runtime) [11, Chapter 3]. A well-known problem with this metric is the fact that it operates over discrete space attributes, and in the case of continuous attributes it is very sensitive to the granularity of the discretization. This work presents a little different approach, that provides both the submit time and runtime discretization granularities and the metric value at once.

1.2 Scheduling Algorithms for Parallel Machines

One of the tasks of an operating system, in particular a parallel one, is scheduling its applications. However, there are very large differences between sequential applications and parallel ones. In order to obtain the full benefits of parallel execution of an application, its processes should run together on different CPUs. Therefore the computation on different CPUs should run synchronously — a property named *co-scheduling*. So, many regular tasks for sequential operating system, (like process preemption, or saving the application state to the disk), are not simple at all on parallel machines. For instance, preempting a single process may block a whole application, since this violates co-scheduling. Therefore, scheduling algorithms of these operating systems usually make only two decisions for each job — when the job starts and how long the job may run before it is killed. Once a job starts, it is not preempted till its termination, user cancellation or abortion.

Unlike a sequential job, a parallel one has at least one more parameter — the number of CPUs it requires. A job cannot start before it is assigned this number of CPUs for its ultimate use; no other job takes these CPUs while this job runs. Due to the Ethernet architecture, the topology of the network usually has less importance, so the only question is indeed having the correct number of CPUs, where each two CPUs are equal for scheduling purposes.

The default algorithms used by current job schedulers for parallel supercomputers are all rather simple and similar to each other. In essence, schedulers select jobs for execution in first-come first-served (FCFS) order, and run each job to completion. The problem is that this simplistic approach causes significant fragmentation, as jobs do not pack perfectly and processors are left idle. Most schedulers therefore use *backfilling*: if the next queued job cannot run because sufficient processors are not available, the scheduler nevertheless continues to scan the queue, and selects jobs requiring less CPUs that may utilize the available resources.

A potential problem with this is that the first queued job may be starved as subsequent jobs continually jump over it. The solution is making a *reservation* for this job, and allowing subsequent jobs to run only if they respect it. This basic algorithm is named EASY-backfilling, or EASY for short [9].

Backfilling requires the runtime of jobs to be known: both when computing the reservation (requires knowing when processors of currently running jobs will become available) and when determining if waiting jobs are eligible for backfilling (must terminate before the reservation). Therefore, backfilling schedulers like EASY users to provide a runtime estimate for each submitted job [9], and the practice was adopted by other schedulers. Jobs that exceed their estimates are killed, so as not to violate subsequent commitments. The assumption is that users would be motivated to provide accurate estimates, because (1) jobs would have a better chance to backfill if their estimates are tight, but (2) would be killed if they are too short.

Nevertheless, empirical studies of traces from sites that actually use EASY show that user estimates are generally inaccurate [12]. A possible reason is that

users find the motivation to overestimate — so that jobs will not be killed — much stronger than the motivation to provide accurate estimates and help the scheduler to perform better packing.

1.3 Predictions

A very common problem for OS schedulers, and in particular ones for super-computers, is predicting job runtimes. Some schedulers use these predictions as heuristics for performance policies. For instance, as noted above, EASY-backfilling [9] makes reservations for jobs according to their runtime predictions (made by users as user estimates). The shortest-job-first (SJF) scheduler [8, 4] tries to start the short jobs before the long ones. Other schedulers can reserve some computation time for a user by selling him this computation time; these schedulers are useful for grid computing economic models (for instance, Erne-mann et al. present a framework where a user can set a deadline for his job's start [5]). All these tasks require the system to predict the runtime of the jobs it runs.

Both backfilling-based schedulers and SJF use single-value predictions due to their simplicity. But predicting a job's runtime cannot usually be done de-terministically. A job's runtime depends on many factors, that include not only system internal conditions such as the network load, but also terminations due to errors and user cancellations. The last factors are external from the system, and they greatly complicate runtime prediction. Errors usually show incorrect behavior pretty soon after a job starts, and many faults may be discovered long before a job would terminate (without the error). Users also know this, and they tend to test partial output soundness soon after their jobs start. Therefore, in cases of errors the job is usually terminated or canceled almost immediately. For instance, 2613 out of 5275 (~50%) canceled jobs in the SDSC-SP2 trace whose user estimates were set for at least 200 minutes were canceled within 20 min-utes after their start times (about the input traces, read further in Section 1.5). Modeling these scenarios is impossible with single-value predictions — a single value can give either a mean or a quantile of the job's runtime distribution, but cannot model the whole distribution.

Another problem with single-value predictions is the fact that they should contain all the information upon which scheduling decisions are made. Different deviations from the real runtime cause different and possibly incomparable dam-age. This leads to prediction policies that are scheduler-dependent. An extreme example is backfilling — it kills jobs whose runtimes are longer than user esti-mates. Thus, over-predictions are much less damaging than under-predictions. Therefore the user estimates tend to be biased upwards — the users tend to give high estimates fearing their jobs to be killed.

Tsafrir et al. present an algorithm that predicts runtimes of jobs as the mean of 2 jobs of the same user that have already finished [4]. Gibbons uses a pretty complex Historical Profiler that interpolates data using quadratic regression, but later uses from all this only the 95% quantile as the single-value prediction of the

job runtime [8]. The contribution of this work is that it presents a framework for modeling the distribution of the job runtimes and a scheduling algorithm that uses these complete distribution models.

1.4   Hidden Markov Models (HMM)

A hidden Markov model (HMM) is a statistical model where the system being modeled is assumed to be a Markov process; but unlike a regular Markov model, where the current observed signal depends only on the previous signal (meaning $\Pr(O_t|O_1...O_{t-1}) = \Pr(O_t|O_{t-1})$), in a hidden Markov model the Markov process represents a hidden state sequence $q_1...q_T$, and an observation signal at each time t depends only on the state at time t, meaning $\Pr(O_t|O_1...O_{t-1}, O_{t+1}...O_T, q_1...q_T) = \Pr(O_t|q_t)$ [10]. In fact, an HMM is the simplest Bayesian network model.

HMMs are used in speech recognition, natural language processing, bioinformatics and many other applications. In the computer system community, the main usage of HMMs is modeling user behavior, for instance, locating the most common sequences of commands (for instance, see [7]). This work presents a little different approach — instead of modeling each user's behavior independently, the HMM is used for characterization of the complete system workload.

1.5   This Work

The goals of this work are as follows:

- Defining a Locality metric, such that this metric would measure the learnability / predictability of a job's runtime given its submit time.

- Defining a framework for workload characterization that is employing the locality.

- Defining a framework for modeling jobs' runtime distributions.

- Presenting a usage of these distributions by a scheduler.

This work uses the traces from [6]. The list of used traces includes LANL-CM5, SDSC-Par95/96, CTC-SP2, KTH-SP2, SDSC-SP2 and SDSC-Blue. Only cleaned versions were used. All the jobs whose runtimes or submit times are unknown (set to -1), or that use no CPUs, were removed from the modeling.

The plan of next chapters is as follows: Chapter 2 presents the metric for measuring the locality. Chapter 3 presents the characterization of the workload using HMMs. Chapter 4 presents the framework for runtime distribution modeling based on job submit times. Chapter 5 presents the scheduler that uses these predictions. Finally, Chapter 6 concludes the work and discusses future research directions.

Chapter 2

Locality Metric

Our first goal is to locate a learnability metric given a trace of jobs. For a single job, let S be its submit time, and let R be its runtime. The question is how much information on R one can ever learn from perfect knowledge of S and their inter-connection. This metric is well known as the mutual information, and its formula is $\int \int \Pr(S, R) \log_2 \frac{\Pr(S,R)}{\Pr(S)\Pr(R)} dSdR$, where S and R are submit times and runtimes; and $\Pr(S, R)$ is the probability for a job to have the submit time of S and runtime of R. (see, for instance, [3]). But it appears pretty difficult to estimate.

First of all, it is tempting to calculate the continuous mutual information (by the formula $\int \int \Pr(S, R) \log_2 \frac{\Pr(S,R)}{\Pr(S)\Pr(R)} dSdR$). The problem with this approach is its applicability: there are not enough samples for a continuous density function estimation.

There are several ways for estimation of mutual information, and all of them have the same basic scheme: Map the S and R to some discrete space and calculate the mutual information between the mapped spaces. Meaning, choose two functions $g(S)$ and $h(R)$, and calculate $I(g(S); h(R))$. The following inequality always holds: $I(S; R) \geq I(g(S); h(R))$ [3]. This way we try to determine an "achievable" amount of mutual information — the algorithm maps close values of S and R to same discrete bin, because close values are hardly separable and cannot be effectively used for classification. This method is called the Grenander's Method of Sieves in [13].

In order to use the discrete mutual information it is necessary to use some discretizing. It is a well known fact that as the S and R binning gets finer, the mutual information $I(S; R)$ grows — there is more information on S to use and more information on R to learn. However, in order to use this approach one has to estimate $\Pr(s, r)$, where s and r denote the submit time and runtime bins. The regular way to estimate $\Pr(s, r)$ is to use the fraction of jobs in the bin from all the jobs: $\overline{\Pr(s, r)} = \frac{n_{sr}}{n}$, where n_{sr} and n denote the number of jobs in the bin and the total number respectively. According to [13], this is a Maximum Likelihood Estimator (or MLE for short). The problem with making the bins

8

too small and using this approach for estimating mutual information is the fact that for very fine binning the estimation of mutual information is heavily biased upwards. Indeed, if each submit time and each runtime bin includes at most a single job, the mutual information estimated as described above would give $\log_2 n$ bits, no matter how approximate the values are. Therefore, this estimator is named "naive", since it assumes that the values in bins truly represent the probability.

As a rule of thumb, the best way to improve the probability estimation is enlarging the sample set; however, in our case this is impossible due to the very problem definition — we try to estimate the probability of the job having some runtime and *arriving at time S*. So even if we continue sampling the jobs, they will not arrive at the correct submit time (unless we change the load on the system, but then we learn another system).

At this point we have two problems to solve — discretizing the submit times and runtimes and defining a metric that evaluates the learnability of runtimes from submit times. I will start from solving the second problem, assuming the submit times and runtimes to be discretized to bins, and then return to the problem of discretization.

2.1 The Metric

The problem with the naive calculation of the Information Gain is its bias. There are several techniques to estimate the bias of the naive estimator. For instance, the Miller-Madow bias estimator for entropy is $\frac{\hat{m}-1}{2N\ln 2}$, where $\hat{m}$ is the expected number of bins with non-zero probability and N is the number of samples [13]. It can be used to estimate the bias on $I(S;R) = H(R) - H(R|S)$. There are proofs for the bias estimation convergence in this case when n tends to infinity and discretization remains the same. However, in this work the number of jobs doesn't tend to infinity, and discretization is not given and yet to be determined. Therefore, I used another technique. Instead of estimating the bias I shuffled the samples' runtimes and subtracted the naive information estimation of the shuffled version from the one of the original version. The reason for choosing this metric instead of Information Gain or even Information Ratio is the fact that both these metrics are heavily dependent on the granularity of discretization (This will be discussed further, in Section 2.2).

The idea of the algorithm is as follows:

$$I(S;R) = \sum_{s,r:\Pr(s,r)\neq 0} \Pr(s,r) \log_2 \frac{\Pr(s,r)}{\Pr(s)\Pr(r)}$$

$$R' = Shuffle(R)$$

$$I(S;R') = \sum_{s,r':\Pr(s,r')\neq 0} \Pr(s,r') \log_2 \frac{\Pr(s,r')}{\Pr(s)\Pr(r')}$$

Algorithm 1 Calculation of the metric

```
double Info(int[] S, int[] R, int T, int N){
    double matrix[T][N];
    double sMargin[T];
    double rMargin[N];
    for each s,r do {
        matrix[s][r] = <fraction of jobs s.t.
                        S[job]=s and R[job]=r>;
        sMargin[s] = <fraction of jobs s.t. S[job]=s>;
        rMargin[r] = <fraction of jobs s.t. R[job]=r>;
    }
    return sum over s,r s.t. matrix[s][r]<>0 of
           matrix[s][r]*log2(matrix[s][r]/(sMargin[s]*rMargin[r]));
}
double F(int[] S, int[] R, int T, int N) {
    double info = Info(S,R,T,N);
    for seed = 0..9 do {
        R' = shuffle(R, seed);
        infoShuffled[seed] = Info(S,R',T,N);
    }
    return info - average(infoShuffled);
}
```

$$f = I(S; R) - I(S; R')$$

When the amount of samples tends to infinity, the expected $I(S; R')$ estimation converges to 0 due to the weak law of big numbers (since the fraction of jobs that fall in the submit time bin s and runtime bin r from all the jobs in submit time bin s converges to $\Pr(r)$). The proposed metric actually tries to estimate the information that is not "random". It can be seen as a kind of learnability test — how much information one can learn from the original trace and cannot learn from a random trace with the same marginal distribution of submit times and runtimes. It is similar to another learnability test — the VC-dimension. A concept class is learnable in the formal model iff its VC-dimension is finite. It means, one can learn a concept class if and only if it cannot "explain everything" — there is some sequence that cannot be explained by the concept of the class. The analogy here is: If any random sequence can return this amount of "information", it is simply a bias, and cannot be learned.

Algorithm 1 receives as input the already discretized array of submit times and runtimes of jobs, together with the numbers of submit time (T) and runtime bins (N). It uses 10 seeds (it is more than enough — the coefficient of variation of the metric is always less than 1% in real traces!) and takes an average as an estimate.

Note, however, that negative values for this metric are theoretically possible. For instance, if the jobs are $((s_0, r_0), (s_0, r_1), (s_1, r_0), (s_1, r_1))$, then the mutual

information estimation is 0 bits, and the expected $I(S; R')$ is $1/3$ bit. The latter is calculated in the following manner: Let $R'[0] = r$; the probability that $R'[1] = r$ is exactly $1/3$: it is choosing 1 out of 3 remaining. In this case $I(S; R') = 1$ bit; otherwise, $I(S; R') = 0$ bits; so the expected value is exactly $1/3$ bit. So the expected f is $-1/3$ bit. It means that there is "negative locality". Indeed, if one sees r_0 in some submit time bin, it is not expected to come again. Such "negative bias" is also described in [13]. However, such scenarios in real life traces are highly unexpected, and I have not seen them.

2.2 Discretizing

There are several discretizing algorithms, that have a single parameter that tunes the granularity. For instance the Simple discretizing algorithm maps the values as follows: $f_{t_x}(x) = \left\lfloor \frac{x}{t_x} \right\rfloor$, where x is the value to map and t_x is the parameter that tunes the granularity of x values. Another algorithm that may be used for this approach is K-Means clustering [11, Chapter 6]. This algorithm must receive the number of cluster as a parameter that indirectly sets the granularity (more clusters means the clusters are smaller and the granularity is finer).

In this work, submit times were always binned with the Simple discretizing algorithm. This means that the bins were some constant time long. As for runtimes, they were always binned in Logarithmic space. This way the error metric in runtime prediction was calculated as $d(R_1, R_2) = |\log R_1 - \log R_2| = \left| \log \frac{R_1}{R_2} \right|$. This means, that if R_1 is 10% bigger than R_2, the error metric is the same no matter what the linear distance between the runtimes. For instance, if the prediction is 100 seconds and real runtime is 101 seconds it is less considerable mistake than if the prediction were 10 seconds and the real runtime were 11 seconds; because our mistake is 1% and the counter-example's mistake is 10%. Special treatment was required for the jobs with runtime of 0 seconds (in the used traces, the runtimes are rounded to the closest integer seconds, see [6]; so for jobs whose real runtime was less than 0.5 second, the runtime is set as 0 seconds). The problem with these jobs is that $\log 0$ is undefined; moreover, since all we know about the real runtime is that it is less than 0.5 second, the relational error in this case may be very significant internally; for instance, at least theoretically, the runtimes of 0.25 second and 1 microsecond are represented by the same value of 0. The jobs with runtime less than 0.5 second exist in any trace; but they are less than 0.5% of all the jobs for all the tested traces, and for most of the traces they are less than 0.0003 of all the jobs. In this work, for measuring the distances, runtime of these jobs was set equal to 1 second; thus all the jobs whose runtime is less then 1.5 seconds were binned together. Note, that even when the trace shows 1 second as job's runtime, its real runtime still can vary from 0.5 second to 1.5 second, meaning a factor of 3. Therefore, those jobs were binned all to the bin the smallest number as "very short jobs". Note, that those jobs are only ~2% of all the jobs in the worst case.

Algorithm 2 K-Means algorithm

```
 1 int[] KMeans(double[] input, int k) {
 2    int[] result;
 3    double[] values = create_copy(input);
 4    sort(values);
 5    double[] distances;
 6    distances[i] = d(values[i+1],values[i]);
 7    sortDescending(distances);
 8    double minimalDistance = distances[k-2];
 9    double[] bounds;
10    int currentBin = 0;
11    for each values[i] {
12       if (d(values[i],values[i+1]) >= minimalDistance) {
13            bounds[currentBin] = values[i+1];
14            currentBin++;
15       }
16    }
17    for all i
18       result[i] = bin s.t. bounds[bin-1]<=input[i]<bounds[bin];
19    do {
20        average[bin] = geom_average(input[i], s.t. result[i]=bin);
21        bounds[bin] = geom_average(average[bin],average[bin+1]);
22        for all i
23           result[i] = bin s.t.
24                 bounds[bin-1]<=input[i]<bounds[bin];
25    } while result has changed in the iteration
26    return result;
27 }
```

Runtimes were binned with the Simple and K-Means algorithm. The K-Means algorithm is implemented the following way (see Algorithm 2): As any Expectation-Maximization algorithm, it has two phases: Initialization and Iterations. The Initialization (Lines 3-18) is to locate the maximal empty intervals, and use them to separate the values to bins. For instance, if $k = 2$, and the runtimes are [2,10,100,200] in seconds; or in $\log_{10}$ scale [0.3, 1, 2, 2.3]; distances are [0.7,1,0.3]; they are sorted descending ([1,0.7,0.3]) and the the $k - 2 = 0$, and minimalDistance is set to distance[0]=1. Thus, initial binning is {2,10},{100,200}. Iterations (Lines 19-25) include an Expectation phase — calculating the averages (Line 20); and a Maximization phase — setting the new bounds, such that each runtime is close to its average, meaning the bounds must be just in the middle of the way between the bin averages (Line 21). Note that all the averages are geometric, since the input works over log space; and distance function d() is log-to-log distance. The iterations stop once the result stops changing, meaning it runs till convergence.

The only open question left at this point is the desired granularity. I decided to use the values that will maximize the value of the metric I have chosen; meaning, find $\arg\max_{t_s,t_r} f$, where t_s and t_r are the arguments for the binning algorithms for submit time and runtime, and f is the metric when using that binning.

For instance, suppose there are only 4 jobs, whose submit times are (0s, 1s, 10000s, 10001s) and whose runtimes are (1000s, 1001s, 100s, 101s). If the finest binning is used — each number in an independent bin — then $I(S; R) = \log_2 4 = 2$ bits; the same amount of information one would receive after shuffling the runtimes, and the metric f would be 0. However, the first two jobs and the last two jobs are very similar and very close at submit time. When binning them together, the discrete mutual information formula gives 1 bit; however the $I(S; R')$ is expected to be $1/3$ bits, just as described above. Therefore, if one uses $f = I(S; R) - I(S; R')$ as the metric, he receives better result on the second choice (2/3 bits instead of 0 bits).

This property belongs to the proposed metric alone. For instance, if one tries to maximize the Information Gain, he receives the very large values when the binning is very fine grained (which is mostly due to the bias). Miller-Madow bias correction indeed converges to the real bias, but when the number of samples increases and not when the discretization changes; the same is true for Jackknifed versions of bias corrections (for formulas see [13]). The metric proposed in the previous section converges to 0 both on very coarse and on very fine granularities, and is meaningful only in the informative binnings — when there is enough information to learn effectively, but not too much.

In this work I tried to locate the t_r value as well as t_s value. However, for some purposes one may need only some information on the runtime. In these cases runtime binning is given by the goal of an algorithm. For instance, for some algorithms it is important to learn whether the job will run more that some threshold ρ, like in [14], where all that interests is whether the remaining time is longer or shorter than the context switch duration. In the previous case, there are two bins of runtimes, ones that are longer than ρ and others that are shorter than ρ. In this case, the goal is to locate the best submit time binning (how long the information is still fresh). So the solution changes as needed — the maximum is found over different t_s values, with fixed runtime binning. It is obviously less than $\max_{t_s,t_r} f$ since the maximization is over a single binning of runtime; however, the t_s value may differ from one in the previous setup. For instance, if the algorithm needs only to learn as little information on the runtime as described above, the best t_s is expected to decrease because for this purpose the good adaptation can be done faster — few samples are enough to measure the probability of this single event, but they may be insufficient to model the complete distribution.

The proposed metric is non-linear and not convex (at least, there is no proof for that). Unless the binning is soft (meaning, we bin the values with different probabilities) it is even not continuous, and surely not differentiable — any specific parameter bins job's runtime to a single bin, and changing the binning (even for a single job!) changes the metric for some $\varepsilon > 0$. For instance, let

Trace	t_s	$t_r = N$	$f(s,r)$
SDSC Paragon '95	1.5 h	18	0.73 bits
SDSC Paragon '96	6 h	22	0.78 bits
LANL CM5	12 h	100	0.41 bits
SDSC SP2	2 h	28	0.85 bits
SDSC Blue	1 h	55	0.65 bits
CTC SP2	40 min	32	0.72 bits

Table 2.1: The best values of t_r and t_s for the different traces, when using K-Means binning for runtimes.

Trace	t_s	t_r	N	$f(s,r)$
SDSC Paragon '95	2 h	1.80	23	0.80 bits
SDSC Paragon '96	6 h	1.49	33	0.85 bits
LANL CM5	6 h	1.20	68	0.45 bits
SDSC SP2	3 h	1.25	59	0.94 bits
SDSC Blue	2 h	1.19	77	0.68 bits
CTC SP2	1 h	1.3	43	0.77 bits

Table 2.2: The best values of t_r and t_s for the different traces, when using Simple binning for runtimes.

the jobs be as presented earlier in the section, and suppose the submit time granularity is tuned for 10 seconds, binning submit times of the first two jobs and the last two jobs together; and let $T = \sqrt[10]{101} \approx 1.586$. For any $t_r \in (T; 1.6)$, simple binning bins runtimes of the first two jobs together to bin 9 and last two jobs together to bin 14, and the metric in this case is $2/3$ as shown above. However, if $1.585 < t_r < T$, runtime of 101s is binned to bin 10, and the last two jobs are not binned together. In this case the naive estimator of $H(R)$ gives $-0.5 \log 0.5 - 0.25 \log 0.25 - 0.25 \log 0.25 = 1.5$ bits, and $H(R|S)$ gives $0.5 \cdot 0 + 0.5 \cdot 1 = 0.5$ bits; I(S;R) is still estimated as 1 bit. Shuffled information depends on where the two first runtimes are shuffled. If they fall together on the first or the last two places the $I(S; R') = 1$ bit as in the previous case; and it happens with the probability of 0.5. Otherwise $H(R'|S) = 0.5 \cdot 1 + 0.5 \cdot 1 = 1$ bits, and $I(S; R') = 0.5$ bit. Thus, $E(I(S; R')) = 1 \cdot 0.5 + 0.5 \cdot 0.5 = 0.75$ bits, and $f = 0.25$ bits. It means, that the metric f is not continuous at $t_s = 10s, t_r = T$.

Therefore, the standard methods (Linear Programming, Convex Optimization, Gradients) cannot be applicable here. According to [2, p. 9], in that case the best option is only locating the local maximum. The local maximum was reached by hand.

2.3 Measurement Results

Tables 2.1 and 2.2 present the optimal granularity parameters and metric for different traces; N denotes the number of runtime bins (it is the same as t_r in the case of K-Means, since in this case it is the number of means). The conclusions are that fine submit time binning and coarse runtime binning is usually the most informative. Saying it informally, it is usually better to use fast (≤ 6 h) non-detailed adaptation than detailed adaptation that is based on long runs. The reason maybe the fact that the real-life trace is highly unpredictable and quickly changing, and old info may not be applicable for current learning. Also, simple binning provides $\approx 8.4\%$ more information on average, thus it is probably better to use as discretizing algorithm.

There is a little problem using K-Means as the binning algorithm. K-Means depends on the data, and when using the Grenander's Method of Sieves, the functions should be chosen independent of data [13]. From now on, all the runtimes are always binned with the Simple algorithm over the logarithmic space.

2.4 Source of Locality

A very commonly asked question is what is the source of locality. Indeed, why are the jobs that come close in time similar to one another? Is it due to the system environmental conditions or due to the user sessions? Is it appropriate to study the submit times instead of modeling each user with his regular jobs and their runtimes? Can the information about the submit time of a job add anything if we already know this job's submitting user?

In order to answer this question, I measured the metric proposed earlier both on jobs' attribute "user" and on pair of attributes ("user","submit time"). Users are never binned together — each user is modeled separately. Runtimes (in this part) were binned with the Simple algorithm. Note, that when modeling both users and submit times, the number of jobs submitted by a single user is much lower than the total number of jobs in the system. Therefore, longer learning periods are required (days and weeks instead of minutes and hours). Also, there are less than 1000 users in the traces, this provides enough data per a user to learn runtimes very accurately without fear of bias. This is expressed as a finer granularity: a relatively small t_r and a relatively large number of runtime bins (on average ≈ 4.3 times larger than when using users alone).

The results of these measurements are presented in Table 2.3. They show that, first of all, modeling the user can be more promising than modeling the submit times. However, modeling submit time together with user can give additional information, on average $\approx 32\%$ more than when modeling the user alone. This means, that user sessions play a pretty big role in the locality of sampling, but it is not ultimate. As of practical usage, it means that the recency should take a part when modeling the runtime distribution of a user's jobs, as was recently shown empirically in [4].

Trace	t_r	N	$f(u,r)$	t_s	t_r	N	$f((s,u),r)$	$f(s;r|u)$
SDSC Paragon '95	1.055	247	0.83 bits	3d	1.5	33	1.21 bits	0.38 bits
SDSC Paragon '96	1.04	332	1.03 bits	1w	1.2	72	1.47 bits	0.44 bits
LANL CM5	1.044	288	0.69 bits	1w	1.3	48	0.97 bits	0.28 bits
SDSC SP2	1.057	238	1.27 bits	2w	1.18	80	1.52 bits	0.25 bits
SDSC Blue	1.027	501	1.11 bits	1m	1.08	174	1.45 bits	0.34 bits
CTC SP2	1.06	192	1.22 bits	2w	1.24	52	1.43 bits	0.21 bits

Table 2.3: User vs User & Submit

Chapter 3

Workload Characterization Using an HMM

3.1 Introduction and Model Definition

A Hidden Markov Model (HMM) models an observed sequence of signals $O_1...O_T$ using a sequence of hidden states $q_1...q_T$. Each state q defines the probability for any possible signal O_t to be seen — $\Pr(O_t|q_t)$, and also defines the probability for all of the states to follow it —$\Pr(q_{t+1}|q_t)$. The standard set of operations over an HMM includes calculation of likelihood of a given signal sequence assuming a given HMM, determining the most likely state sequence for a given signal sequence and HMM (solved by the Viterbi algorithm), and finding the optimal parameters for the HMM states to maximize the likelihood of a given sequence of signals (the latter one is solved by the Baum-Welch algorithm, that is a special case of the Expectation-Maximization algorithm) [10].

In order to model the job trace (or anything else) using an HMM first of all one must decide what is the signal. One option is ordering the sequence of jobs by their submit times, and then defining the runtime (or runtime bin) of a single job number t as a signal O_t. This is indeed very simple, however this way we lose the inter-arrival time dependency. Indeed, if there is a long period without submissions of jobs, the jobs still come one after another; so this long inter-arrival time period is not modeled. The dependency between jobs when the inter-arrival time is large is expected to be less than when the inter-arrival time is small. Therefore the signal at a submit time bin t is defined as a list of runtimes of jobs that arrive at this submit time bin.

The other question is how a state defines the probability for a possible signal. There are two options. The first one defines the state as the job runtime distribution, such that each job's runtime is sampled from this distribution independently from all the rest of the jobs. The probability of the given signal is calculated by the following formula: $\Pr(x_{1,t}..x_{N,t}|q_t = i) = \prod_{j=1}^{N} B_{ij}^{x_{j,t}}$, where $x_{j,t}$ is the number of jobs at submit time bin t and runtime bin j, N is the

number of runtime bins, and B_{ij} is the probability of job runtime bin j in state i (this is indeed very similar to the state matrix of [10], with the only difference that we use the same matrix multiple times since there are many jobs to model. Unless specified otherwise, all the variables are from there). The Expectation-Maximization update should maximize over all parameter sets $\hat{\theta}$ the value of

$$\sum_Q \Pr(Q|O,\theta)\ln\Pr(O,Q|\hat{\theta}) =$$

$$= \sum_Q \Pr(Q|O,\theta)\ln[\Pr(O|Q,\hat{\theta})\Pr(Q|\hat{\theta})] =$$

$$= \sum_Q \Pr(Q|O,\theta)\ln\Pr(O|Q,\hat{\theta}) + \sum_Q \Pr(Q|O,\theta)\ln\Pr(Q|\hat{\theta})$$

where O is the observation sequence, Q is the state sequence, and $\theta,\hat{\theta}$ are the parameters of the current and next iterations (see [10, p. 265]). But the right term remains constant when B_{ij} changes (it is subject to state transition and initial state distribution). Therefore, here we only concentrate on the left term.

$$\sum_Q \Pr(Q|O,\theta)\ln\Pr(O|Q,\hat{\theta}) =$$

$$= \sum_{t,i} \gamma_t(i)\ln(\prod_{j=1}^{N} \hat{B_{ij}}^{x_{j,t}}) = \sum_{t,i,j} \gamma_t(i)x_{j,t}\ln\hat{B_{ij}};$$

$$\forall i, \sum_j \hat{B_{ij}} = 1$$

and, after the differentiation $\frac{\partial}{\partial \hat{B_{ij}}}$ and using the Lagrange multiplier technique, we receive

$$\frac{\sum_t \gamma_t(i)x_{j,t}}{\hat{B_{ij}}} = c_i = \sum_{t,j} \gamma_t(i)x_{j,t} \Rightarrow \hat{B_{ij}} = \frac{\sum_t \gamma_t(i)x_{j,t}}{\sum_{t,k} \gamma_t(i)x_{k,t}}$$

where $\gamma_t(i) = \Pr(q_t = i|O,\theta)$.

One problem with the proposed HMM structure is that it cannot generate the workloads. Given a list of submit times of jobs, it can sample the runtimes for the jobs, but it cannot define what the load should be. It can be useful for predicting the currently running and submitted jobs, but it cannot predict the future load. Generativity of the HMMs is also required when comparing two models. Very similar HMM models can be represented by completely different parameters [10, p. 271]. The solution is calculating the expected KL-divergence or JS-divergence, meaning the expected difference of log-likelihood of signal sequences *generated by the HMM* (divided by a length of the generated observation sequence T, when T tends to infinity). Of course, when the HMM cannot

generate load, one solution is giving some load from an outside source, but this inserts outsider influence to the comparison.

An alternative that solves this problem models both runtimes and load — meaning, state defines the runtime distribution and the load distribution. This model can also generate workloads. Then,

$$\Pr(x_{1,t}..x_{N,t}|q_t = i) = \Pr(\sum_j x_{j,t}|q_t = i)\Pr(x_{1,t}..x_{N,t}|\sum_j x_{j,t}; q_t = i) =$$

$$= \Pr(\sum_j x_{j,t}|q_t = i)\prod_{j=1}^{N} B_{ij}^{x_{j,t}}$$

— the probability of the runtime list in state i is the probability of the load in this state to be equal the list length multiplied by the probability of this runtime list at the given load and state (the latter factor is exactly the same as earlier). The only open question is the load distribution. (From here on, load at time t is represented by $l_t = \sum_j x_{j,t}$). Assuming the Poisson distribution for load (meaning $\Pr(l_t|q_t = i) = \frac{e^{-\lambda_i}\lambda_i^{l_t}}{l_t!}$, where λ_i is the expected load at state i), the total distribution is $\frac{e^{-\lambda_i}\lambda_i^{l_t}}{l_t!}\prod_{j=1}^{N} B_{ij}^{x_{j,t}}$. The estimation of this value should maximize

$$\sum \Pr(Q|O,\theta)\ln\Pr(O|Q,\hat{\lambda}_i,\hat{B}_{ij}) = \sum_{t,i}\gamma_t(i)\ln\left[e^{-\hat{\lambda}_i}\frac{\hat{\lambda}_i^{l_t}}{l_t!}\prod_{j=1}^{N}\hat{B}_{ij}^{x_{j,t}}\right] =$$

$$= -\sum_{t,i}\gamma_t(i)\hat{\lambda}_i + \sum_{t,i}\gamma_t(i)l_t\ln\hat{\lambda}_i - \sum_{t,i}\gamma_t(i)\ln(l_t!) + \sum_{t,i,j}\gamma_t(i)x_{j,t}\ln\hat{B}_{ij}$$

Therefore, after differentiation $\frac{\partial}{\partial\hat{\lambda}_i}$ we locate the stationary point:

$$-\sum_t\gamma_t(i) + \frac{\sum_t\gamma_t(i)l_t}{\hat{\lambda}_i} = 0$$

$$\hat{\lambda}_i = \frac{\sum_t\gamma_t(i)l_t}{\sum_t\gamma_t(i)}$$

Matrix B changes by the same formula, with exactly same explanation.

If, on the other hand, the load model is Geometric, then $\Pr(l_t|q_t = i) = p_i(1-p_i)^{l_t}$, where p_i is the termination parameter of the Geometric distribution at state i). The estimation of this value should maximize

$$\sum \Pr(Q|O,\theta)\ln\Pr(O|Q,\hat{\lambda}_i,\hat{B}_{ij}) = \sum_{t,i}\gamma_t(i)\ln\left[\hat{p}_i(1-\hat{p}_i)^{l_t}\prod_{j=1}^{N}\hat{B}_{ij}^{x_{j,t}}\right] =$$

$$= \sum_{t,i}\gamma_t(i)\ln\hat{p}_i + \sum_{t,i}\gamma_t(i)l_t\ln(1-\hat{p}_i) + \sum_{t,i,j}\gamma_t(i)x_{j,t}\ln\hat{B}_{ij}$$

Therefore, after differentiation $\frac{\partial}{\partial \hat{p}_i}$ we locate the stationary point:

$$\frac{\sum_t \gamma_t(i)}{\hat{p}_i} - \frac{\sum_t \gamma_t(i)l_t}{1 - \hat{p}_i} = 0$$

$$\hat{p}_i = \frac{\sum_t \gamma_t(i)}{\sum_t \gamma_t(i)(l_t + 1)}$$

I didn't present proofs that these stationary points are indeed the maxima. However, the proofs for this are very simple. For instance, when $\hat{\lambda}_i \to 0$ or $\hat{\lambda}_i \to \infty$, for any submit time bin t having at least one job, $\Pr(l_t|q_t = i, \hat{\lambda}_i) \to 0$; and therefore the target function tends to $-\infty$. This means that in a compact space of $\hat{\lambda}_i \in [0, \infty]$ the continuous function after the domain compactification receives the global maximum (whose existence is ensured by the topological formulation of the Weierstrass extreme value theorem) in the interior $(0, \infty)$. In this domain, the function is differentiable and therefore the global maximum must appear in a stationary point (due to the Fermat's stationary point theorem). There is only a single stationary point, therefore it is the global maximum. The remainder is the case when $\forall t, l_t = 0 \vee \gamma_t(i) = 0$, since in this case for any submit time t where $\gamma_t(i) \neq 0$ implies $\Pr(l_t|q_t = i, \hat{\lambda}_i \to 0) \to 1$; but in this case $\frac{\sum_t \gamma_t(i)l_t}{\sum_t \gamma_t(i)} = 0$, so it is a part of the common case. Similar proofs work in the cases of $\hat{B}_{ij}$ and $\hat{p}_i$, thus they are omitted.

An assumption that the load is Poisson-distributed may be very questionable. Section 3.5 discusses real load distributions of different states, and they are pretty far from being Poisson. However, as a model, it can work as defined. The parameter estimation subroutine (Baum-Welch) only tries to locate the best Poisson model that can describe the seen observation sequence.

The metric used to evaluate a model is its ability to explain and predict an unseen test. For this the trace is divided in two, and HMM parameter estimation using the Baum-Welch algorithm runs only on the first half of the jobs. The test is the log-likelihood of the second half. In fact this is a well known and widely used training and validation sets approach [11, Chapter 3]. Note that negation of the log-likelihood is a surprise measure — meaning, formally, that amount of information in the message is exactly $-\log \Pr(M)$ [3]. If the HMM models only runtime distributions and contains a single state q, then $\forall t, \gamma_t(q) = 1, B_{qj} = \frac{\sum_t x_{j,t}}{\sum_{t,k} x_{k,t}} = p_1(j)$; and the log-likelihood is exactly $n_v \cdot \sum_j p_v(j) \log p_t(j)$, where n_v is the number of jobs in the validation half of the trace, and $p_t(j), p_v(j)$ are the fractions of all the jobs with runtime bin j in the training and validation halves, respectively. When divided by the number of jobs in the validation set, it is the negation of the entropy of p_v when it is modeled with p_t — it is exactly a negation of $H(p_v) + D_{KL}(p_v||p_t)$. If the HMM contains more than one state, log-likelihood improvement divided by the number of jobs can be thought of as an average information per job obtained by using the HMM as a model on the validation set: $G_{HMM} = \frac{1}{n_v} \log_2 \Pr(R_v) + H(p_v) + D_{KL}(p_v||p_t)$ — the gain of the HMM is defined as the Log-Likelihood of the validation set divided by the

size of the validation set plus the entropy of runtimes in the validation set plus the KL-divergence between the runtime distribution of the validation set (as a real distribution) and of the training set (as a model distribution).

A problem with the given technique is the fact that sometimes the training set (and therefore also the model) includes no job at some runtime bin at all, and the validation set does include such jobs. The meaning is that the surprise measure for these jobs is $-\log 0 = \infty$! This means that the $\log_2 \Pr(R_v) = -\infty$ and $D_{KL}(p_v \| p_t) = \infty$; and the gain is set as undefined. There is a simple way to solve it by adding a safety distribution to the model; meaning, for a safety parameter $\varepsilon > 0$, define the new probability p' as

$$p' = \begin{cases} p(1 - \varepsilon) & : p \neq 0 \\ \frac{\varepsilon}{N_0} & : p = 0 \end{cases}$$

where p is the old probability of a runtime bin, and N_0 is the number of empty runtime bins in the model. Similar safety formulas are possible also when modeling an infinite series of runtime bins; only then the safety should be completed from a geometric distribution. The safety parameter ε should depend on the size of the training set, for instance $\varepsilon = \frac{1}{n_t}$. Note, however, that this safety method doesn't depend on the HMM modeling — it can be applied with any model. The information gain due to the HMM doesn't not change due to this safety method. Indeed, if $p \neq 0$, then $\log p' = \log p + \log(1 - \varepsilon)$, and this last term is independent of the model — it is the same when using an HMM and when not using an HMM; so it is eliminated. Otherwise, if $p = 0$, then $\log p' = \log \varepsilon - \log N_0$, and again this is constant in both models and eliminated. Therefore, in this work the jobs in runtime bins that are not populated in the training set are removed from the validation set.

A similar technique is presented in [11, Chapter 6] as m-estimation of probability. The idea there is to add some m samples to all the bins uniformly, so that the distribution is updated as $p' = \frac{n_t p + m}{n_t + mN}$. I decided to use a little different technique to ensure that the ratio of probabilities of two bins with non-zero probability will remain constant; so it remains constant when removing the jobs that are not modeled.

Another problem arises when we try to compare HMMs with and without load modeling. Log-likelihoods reflect probabilities for different events and cannot be compared directly. The log-likelihood of a trace given an HMM that models both runtime distributions and load is expected to be less since it includes the load modeling and does not accept it as is. In order to compare these options for HMMs, I used the following method: Let R be the runtime sequence of jobs and L to be the Load sequence. When the HMM doesn't model the load, it measures the probability of the runtime distribution given load of the trace — $\Pr(R|L, HMM)$. Therefore, the correct way to measure it with load modeling is $\Pr(R|L, HMM) = \frac{\Pr(R,L|HMM)}{\Pr(L|HMM)}$, where the dividend is the regular likelihood of the HMM that models both runtime distribution and load; and the divisor is modeling the load sequence only, meaning that states model the load only — $\Pr(O_t|q_t = i) = \Pr(l_t|q_t = i)$.

Algorithm 3 Static IV algorithm

```
 1 void initStatic() {
 2    for each state i do {
 3        int startXi = i*M/N;
 4        int endXi = (i+1)*M/N;
 5        for each runtime bin j do {
 6            B[i][j] = 1.0/(endXi-startXi+N);
 7            if (startXi<=j && j<endXi)
 8                B[i][j] *= 2;
 9        }
10    }
11 }
```

$$\log \Pr(R|L, HMM) = \log \Pr(R, L|HMM) - \log \Pr(L|HMM).$$

3.2   Initial Values

Baum-Welch is a special case of the Expectation-Maximization algorithm, and as such defines how the existing parameters from the previous iteration should change; but two issues are left undefined — its initialization (what are the initial parameters) and termination (when the iterations should stop). This section deals with Baum-Welch initialization; the next one discusses different termination criteria.

As written in [10, p. 273], uniform initial values for the state transition matrix and initial state distribution is "adequate for giving the useful reestimates in almost all the cases". The problem is the initial values of the state matrix B. There are two ways that were used in this work to set the state matrix initial values: statically and based on k-means. Algorithm 3 presents the Static initial values (Static-IV) definition. It is independent of data in the trace. The basic idea is as follows: For each state i, it selects a set of runtime bins X_i (lines 3-4; M and N represent the number of states and runtime bins respectively [10, p. 260]), and sets their per-bin probability to be twice higher that the probability of the rest of the runtime bins: $X_i \subset \{1..N\}, \forall x \in X_i, y \notin X_i, B_{ix} = 2B_{iy}$. Since each row in this matrix must represent the distribution of the runtimes bin, it implies $B_{ix} = \frac{2}{|X_i|+N}, B_{iy} = \frac{1}{|X_i|+N}$. The obvious advantage of this static approach is the fact that there is no explicit over-fitting and its applicability to the on-line algorithms.

There are cases when the initial values are very bad; especially when using static initial values. For instance, if there is a submit time bin with two jobs that belong to different runtime bins, and there is no state that can give both bins with good probability. In this case, for each state the probability of being in that state equals to 0. It is a pretty dead-end case. To solve it I added the ability to adjust the state matrix to fit the submit time bin. It is done

as follows: Each state changes to support the submit time bin a little better by setting the probabilities a little closer to this submit time bin. Namely, $\forall i, j; B_{ij} = (1 - \lambda(t))B_{ij} + \lambda(t)p(j|t)$, where $\lambda(t)$ denotes the fraction of jobs in submit time bin t and $p(j|t)$ is the fraction of jobs of runtime bin j among all the jobs of submit time t. The motivation is as follows — let the old state matrix model the rest of the jobs, and adapt it for the part of the jobs that it cannot explain. Although it is not a part of regular HMM implementation, I saw this only on start of initial values. After the initial values are OK, the rest runs very smoothly. It may be seen as the initial adaptation of the initial values to the trace.

An alternative option is to use the k-means algorithm [10, p. 273], see Algorithm 4. The idea of the algorithm is as follows — Cluster the submit time bins of the given trace with the k-means algorithm using Kullback-Leibler divergence as the metric (Lines 20-26), by iteratively dividing each cluster by two and running k-means each time till convergence. The division of the cluster is done as follows (Lines 8-17) — For each state calculate the expectation of runtime bin number (Line 9), and separate the submit time bins such that the submit time bins with the higher averages than the expected in this state are in a separate cluster. This way the submit time bins populated with jobs with high runtimes are separated from submit time bins with low runtimes. It is surely not enough, and in order to ensure the intra-cluster similarity (not only in terms of averaging, but in terms of runtime binning) Lines 20-26 run the regular k-means algorithm till convergence.

The k-means method separates the submit time bins better. It alone cannot be used for the predictions, since the state durations and state transition probabilities are not modeled by clustering unless the HMM is used. However, this clustering does most of the work, and the HMM can later easily learn the state transition matrix pretty fast. Therefore, the Baum-Welch algorithm converges much faster. The drawbacks of this approach is the high over-fitting due to the dependency on data, and inapplicability at on-line predictions, before there are enough samples to learn from.

3.3   Baum-Welch Termination

In this work, two main approaches were taken regarding termination — fixed number of rounds and convergence-based termination criteria. The advantage of running Baum-Welch for a fixed number of rounds is its runtime cost predictability. Its greatest disadvantage is the fact that the results may not reach convergence; therefore it may heavily depend on the initial values and the number of rounds.

As of convergence-based termination, there were two approaches as well. The first one is running Baum-Welch till complete convergence of all the parameters - till the parameters stop changing. Since the parameters are interdependent and their cross dependence is very complex, the EASY way is assuming all the parameters important, and running till convergence in L_∞ metric space —

Algorithm 4 k-means IV algorithm

```
 1 void initKMeans(Trace obs) {
 2    int currentM = 1;
 3    int states[obs.T];
 4    for each submit time bin t do
 5       states[t] = 0;
 6    relax(obs,states,currentM);
 7    while currentM != M do {
 8       for each state i=[0,currentM) do {
 9          double mean = sum over j of B[i][j]*j;
10          for each t s.t. states[t]=i do {
11             double average =
12                (sum j=[0,N) of
13                   obs.matrix[t][j]*j)
14                / obs.load[t];
15             if (average > mean)
16                states[t] += currentM;
17          }
18       currentM *= 2;
19       relax(obs,states,currentM);
20       do {
21          states[t] = argmin i over 0..currentM of
22             sum j=[0,N) of
23                p[t][j]*log2(p[t][j]/B[i][j]),
24                where p[t][j]=obs.matrix[t][j]/obs.load[t];
25          relax(obs,states,currentM);
26       } till convergence of states[t]
27    }
28 }
29
30 void relax(Trace obs, int[] states, int currentM) {
31    for each i=[0,currentM), j=[0,N) {
32       B[i][j] = <fraction of jobs with runtime bin j
33                  from all the jobs in submit time bins t
34                  s.t. states[t]=i>
35    }
36 }
```

---

meaning, the distance between the previous and current parameters is defined as the maximal over all parameters distance between the current step and the next one — $L_\infty(\theta, \hat\theta) = \max_i\{|\theta_i - \hat\theta_i|\}$; where $\theta_i, \hat\theta_i$ are the parameters in current and next steps. This approach gives pretty stable results. However, it requires many iterations, and, more important, it is proven to suffer from over-fitting.

The alternative approach is setting the threshold as a function of the sample set, meaning $\frac{\epsilon}{\sqrt{n_t}}$, where ϵ is a constant and n_t is the number of jobs in the training set. The idea is as follows: Assuming the model is correct, each sample can give some information for the parameter. This amount of information is named the Fisher information. All the samples can give n_t times more information. By Rao-Cramer inequality, the variance of the unbiased estimator (ones that we have are indeed unbiased) has lower bound of reciprocal of this information. The deviation is proportional to the square root of the variance and therefore is inversely proportional to the square root of the number of samples. This approach may prevent the over-fitting.

Also, since there are many parameters to learn, and the number of parameters changes with different binning and number of states, the L_∞ metric is inappropriate. Therefore the metric was changed, and sometimes I used the L_1 metric of the sum of distances — $L_1(\theta, \hat\theta) = \sum_i |\theta_i - \hat\theta_i|$. The reason for this metric is the fact that the parameters the Baum-Welch tries to learn are the distributions, the distance metric between two distributions is Jensen-Shannon divergence and L_1 (or, more exactly, $0.5L_1$) is its good approximation.

3.4 Results

To sum up the previous sections, here are the parameters for the HMM.

- Trace name

- t_s — submit time bin length.

- t_r — runtime bin size (in logarithmic space)

- M — # of states.

- Load Modeling.

- Algorithm for initial values of state matrix

- Baum-Welch termination criterion — one of {Fixed, L_1, L_∞}.

- ϵ — Parameter of termination criterion — number of rounds in case of fixed round number or distance threshold in case of convergence.

3.4.1   Runtime Bins and Number of States

Each state in the HMM represents a separate cluster of submit time bins. The state at a submit time bin includes the complete knowledge of the probabilities of current and next jobs distribution. Therefore it may be seen as a sort of information bottleneck — all the information must pass through the state number. Information we plan to model is defined by runtime bins of jobs. Therefore, a sensible question rises — How does enlarging the state set improve the performance of the HMM as a model of runtime distribution depending on the runtime binning? In other words, if we require more information on runtime, how enlarging the bottleneck helps us?

At this point I want to make clear that the runtime complexity of each iteration of Baum Welch is $O(TNM^2)$. This means that adding states is very expensive in terms on runtime complexity, which is extremely important in cases when the results are required on-line. Furthermore, the total amount of parameters is $M^2 + MN + M = O(M(M+N))$. This poses serious questions on the efficiency of learning due to model complexity grows in quadratic proportion with enlarging the state set.

Figure 3.1 presents the improvement of the HMM performance with more states for different runtime binnings as a function of learning time. Note that the runtime complexity of Baum-Welch increases with the number of states, and the line representing the 32-state HMM shows the real runtime cost in terms of 16-state HMM round for comparison of the real cost of the received information. The results show that while there is in fact not much gain in case of $t_r = 1.8$, there can be more gain when $t_r = 1.4$, meaning the fine grain runtime binning leaves more information, and it can later be learnt by the Baum-Welch algorithm. This is not always true, of course; the necessary condition is enough information to remain in fine grain binning. This question can be answered with our previous tool, the locality metric. For instance, the optimal runtime binning for SDSC Paragon 95 is $t_r = 1.8$, see Table 2.2. Therefore the improvement when the binning is finer, is less attractive. On the other hand, for CTC SP2 optimal $t_r = 1.3$. So, using fine runtime binning in this case can improve the learnability. These are basically bad news — the learnability metric is very dependent on the runtime binning. The optimal runtime binning is not easily discovered on-line. It means that the optimal HMM-based model is not easily defined.

3.4.2   Sensitivity to Submit Time Binning

The results (see Figure 3.2) show that the submit time binning that were found in Chapter 2 are too large. The explanation I find for this is the fact that Locality sees each submit time bin independently from each other; while the HMM builds the model of dependency, and finer submit times give better resolution for modeling the connection between jobs. Finer submit time binning gives better Log-Likelihood, but requires more steps to run. For SDSC Paragon 95, when submit time bins are ~5 minutes long (by the way, it means ~0.5 job per bin on average!) the information received by the HMM grows logarithmically slow
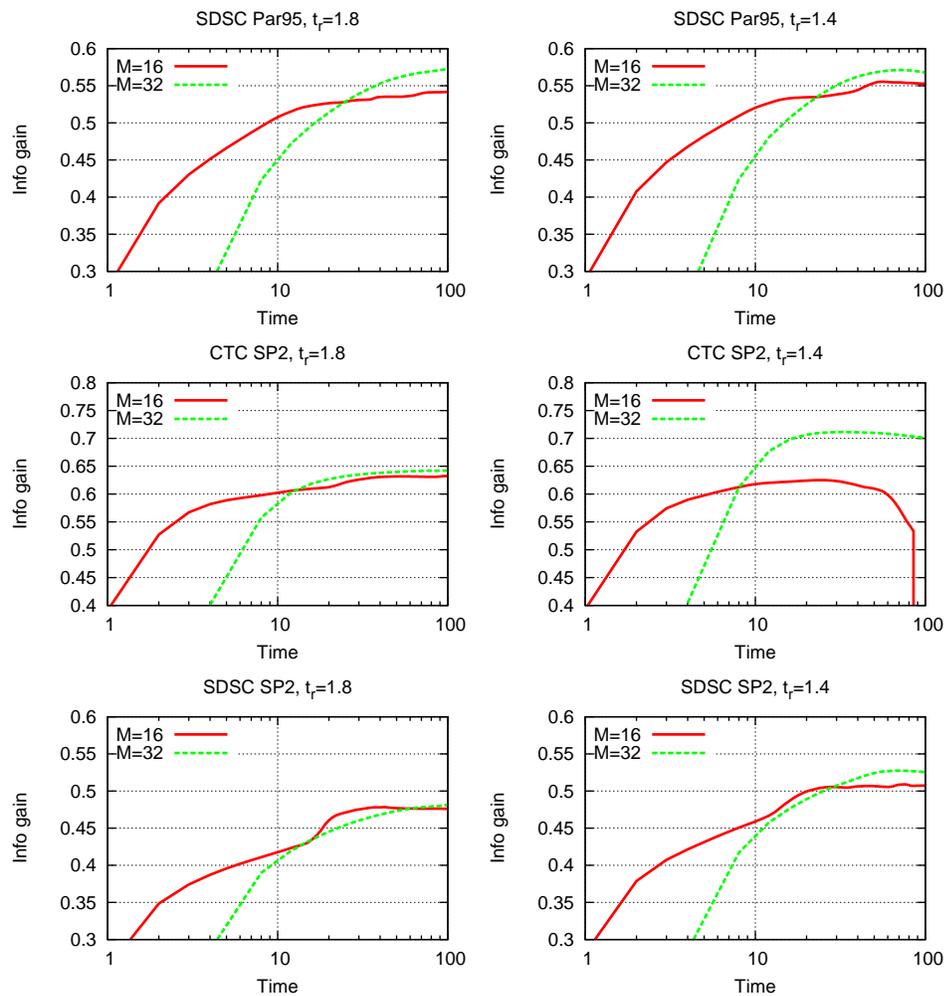
Figure 3.1: State number (M) dependence on runtime binning (t_r). t_s=15 min, no load modeling, Static-IV. Time is in terms of a 16-state HMM round; each round of a 32-state HMM is 4 times larger.
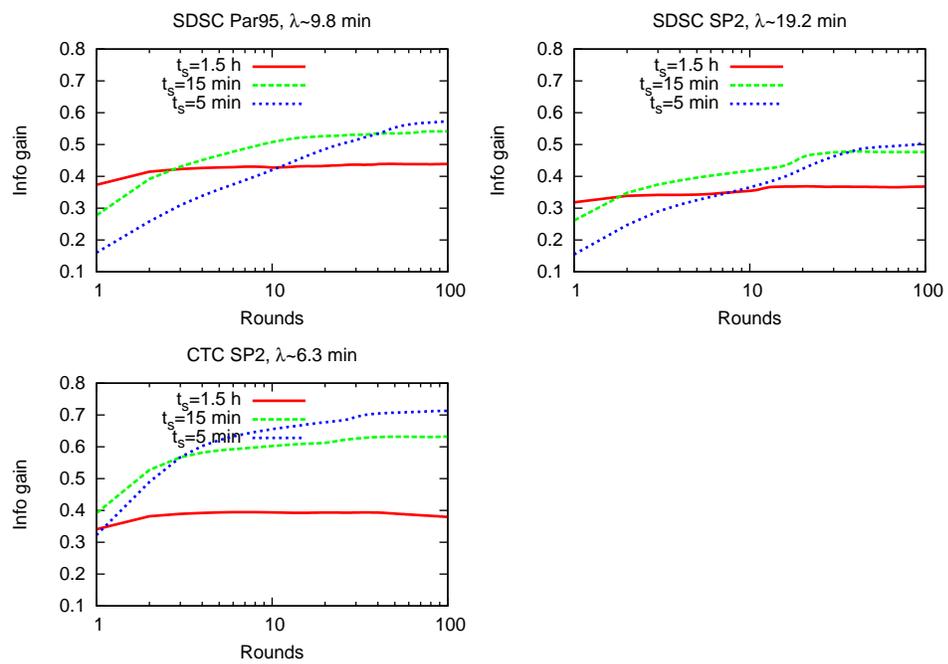
Figure 3.2: Submit time sensitivity. $t_r = 1.8$, $M = 16$, Static-IV. Headers contain the mean inter-arrival time.
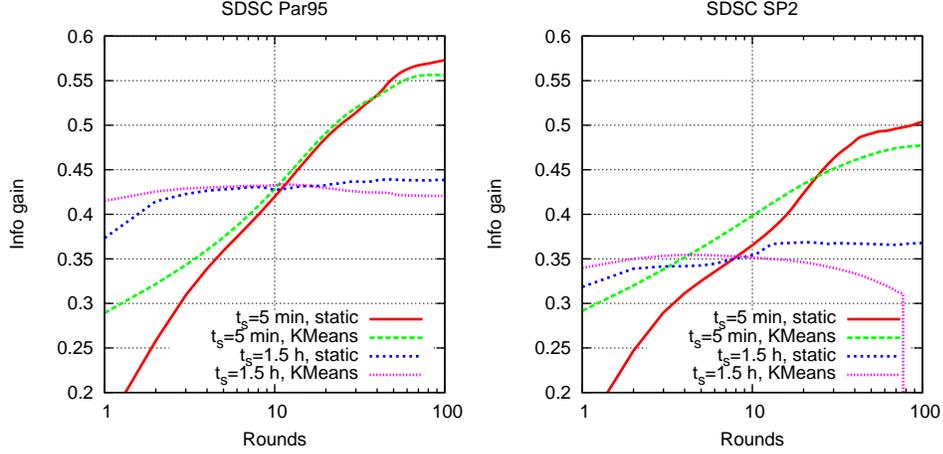
Figure 3.3: Comparison between different initial values. $t_r = 1.8$, $M = 16$

(with base close to 10^5!) and ends up after 100 rounds with 0.57 bits. On the other hand, 15 minute long submit time bin improves much faster but reaches only ~0.54 bit — slightly less than previously. Conclusion — 15 minutes long submit time bins look appropriate both in terms of fast convergence and good performance.

It is possible to make the set the submit time binning first coarse and then finer. This might give the fast convergence initially and good performance at the end. This requires to reset the state transition matrix with change of submit time binning t_s parameter. The proposed way to do so is as follows: Suppose $t'_s = 0.5t_s$, meaning each new submit time bin is twice shorter than the old one, so the old submit time bins are divided by 2. It also means that new state transition probabilities should be set as if states of the new submit time bins equaled ones of the corresponding old submit time bin: $\forall i \neq j, A'_{ij} = 0.5A_{ij}; A'_{ii} = 0.5A_{ii} + 0.5$. This automatically doubles the expected state duration $\overline{d_i}' = \frac{1}{1-A'_{ii}} = \frac{1}{0.5-0.5A_{ii}} = \frac{2}{1-A_{ii}} = 2\overline{d_i}$, and it is correct — twice more new submit time bins are now expected in a single state interval. This idea was not tested in this work; but it may be a promising future work direction.

3.4.3 Dependency on Initial Values

As written above, using k-means for separation of different initial values helps at the beginning — the submit time bins are already separated by k-means, so the HMM has less work to do — just find out the state transition metric (see Figure 3.3). It can help especially when submit time binning is fine. However, eventually the performance metric of both HMM models converges.
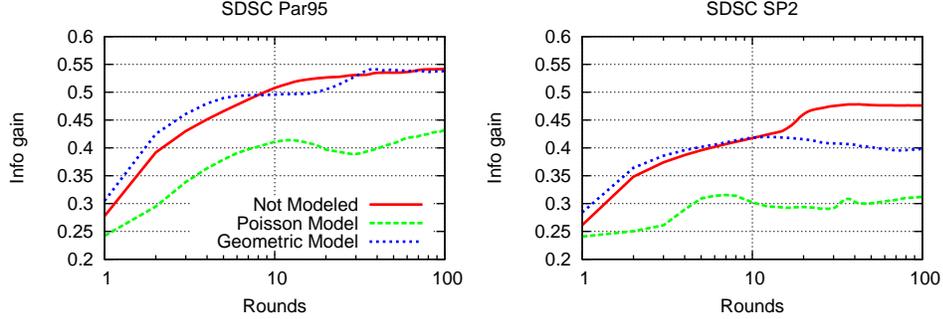
Figure 3.4: Load Modeling Dependency. $t_r = 1.8$, $t_s = 15$ minutes, $M = 16$, Static-IV.

The problem appears when there is a submit time bin in the validation set that has no representation in the k-means-created clusters. Indeed, suppose that there are 2 states, and the complete set of runtime bins in the training set can be separated with two disjoint subsets — R_{t1}, R_{t2}, and at each submit time bin there are jobs in runtime bins of only one of these disjoint subsets, meaning any submit time bin t and any runtime bins $j_1 \in R_{t1}, j_2 \in R_{t2}$ satisfy $x_{tj_1} x_{tj_2} = 0$. k-means clustering may eventually converge such that the submit time bins are clustered as follows: $q_t = q_0 \Leftrightarrow \sum_{j \in R_{t1}} x_{tj} \neq 0$. This clustering satisfies the convergence — KL-divergence between any submit time bin and the other cluster is infinity, so it will never pass to the other cluster. It means that every state i and any runtime bins $j_1 \in R_{t1}, j_2 \in R_{t2}$ satisfy $B_{ij_1} B_{ij_2} = 0$. The HMM parameter estimation procedure Baum Welch doesn't change this forever - if, for any i, j, $B_{ij} = 0$, then for any submit time bin t such that $x_{tj} \neq 0 \Rightarrow \gamma_t(i) = 0$ and therefore this submit time bin doesn't contribute anything in the $\hat{B}_{ij}$. It means that the model forbids submit time bins that include jobs from runtime bins of those disjoint sets. This may lead to very bad over-fitting. Static-IV, at least initially, don't forbid any mixture of runtime bins. Of course, if the complete training set contains no job with some runtime bin, the model will eliminate it from all the states and forbid it; but this problem can be solved with greater ease with the safety techniques described earlier in the chapter that discusses the Model performance metric.

3.4.4   Load Modeling

As written above, the advantage of modeling the load is obtaining a generative model. On the other hand, when modeling load with the runtime distribution, the HMM performance may deteriorate compared to an HMM modeling runtimes alone (see Figure 3.4). Note, that it doesn't mean there is no correspondence between load and runtime. The explanation for this is as follows — suppose at some time slice t, $\Pr(x_{1,t}..x_{N,t} | \sum_j x_{j,t}, q_t = 1) >$

$\Pr(x_{1,t}..x_{N,t} | \sum_j x_{j,t}, q_t = 2)$, but $\Pr(\sum_j x_{j,t} | q_t = 1) \ll \Pr(\sum_j x_{j,t} | q_t = 2)$. The HMM decides what state to choose by multiplication of these two factors — $\Pr(O_t | q_t = 1) < \Pr(O_t | q_t = 2)$ — therefore it is clustered with state 2, although the runtimes are explained better by state 1.

Another disadvantage of modeling load is requirement to choose the correct load distribution family. As appears on Figure 3.4, the Geometric distribution as a model for load works significantly better than the Poisson distribution.

The question of whether the load should be modeled does not have a simple answer. It makes possible designing scheduling algorithms that take into account the future load of the system. However, these algorithms suffer from low utilization due to reservations for unsubmitted jobs. Currently, most scheduling algorithms don't consider the future load in their decisions. Instead, at any time, the algorithm considers only jobs that are present in the system.

3.5 Properties of HMM States

One of the goals of this work was characterizing the workload. It is indeed interesting what do these HMM states represent. When one models any sample set, first of all he must decide on a distribution family. After this decision is made, even arbitrarily, he can calculate the best fitting parameter for the model; this can also be used (and even effectively!) for the real sample modeling. But characterization of the workload may give insights for future improvements.

For instance, it is well known that the state transition matrix approach implies a Geometric state duration distribution, with parameter $p = 1 - A_{ii}$ (in [10, p. 269] it is named "exponential"; but it has a discrete support of natural numbers — submit time bins). Is the real state duration distribution indeed Geometric?

Another question — what is the real load distribution? Earlier, I modeled it with the Poisson distribution. Is it indeed Poisson-distributed? As we earlier saw, a choice of the load model distribution may affect the results when using an HMM that models load.

The approach I took here for comparing the distribution versus the real data was applying the KL-divergence between the "real data distribution" and the maximally likely model distribution that can explain it. For instance, for the Geometric distribution it is when its termination probability parameter p is the reciprocal of the sample average.

There is still a problem to solve: Sometimes the number of samples is extremely low (< 20), so the KL-divergence is pretty big, although the CDF of the distribution looks exactly like the model distribution. The expected cause is the formula of $D_{KL}(p||q) = -\sum p_i \lg q_i - H_{MLE}(p)$. The problem with it is the fact that the Maximal Likelihood Estimator of entropy is always biased downwards, resulting in a large KL-divergence estimation. In order to estimate the bias I used the Miller-Madow bias correction, where the bias is estimated as $\frac{m-1}{2N \ln 2}$ bits, where m is an estimate of the expected number of non-empty bins, and N is the sample size [13, p. 1197]. Two approaches were taken for estimating the

expected number of non-empty bins — one is by simply counting non-empty bins (referred later as Simple bias correction); and the second by calculating the expected number of non-empty bins in the modeled distribution. For the Geometric distribution the latter estimation may be calculated by the formula

$$m = \sum_{n=0}^{\infty} \left(1 - (1 - pq^n)^N \right) = \sum_{n=0}^{\infty} \left(1 - \sum_{k=0}^{N} \left(\begin{array}{c} N \\ k \end{array} \right) (-pq^n)^k \right) =$$

$$= -\sum_{k=1}^{N} \left(\begin{array}{c} N \\ k \end{array} \right) (-p)^k \sum_{n=0}^{\infty} q^{nk} = -\sum_{k=1}^{N} \left(\begin{array}{c} N \\ k \end{array} \right) (-p)^k \frac{1}{1 - q^k}$$

3.5.1 State Duration Distribution

There are 2 possible approaches for determining the "real state duration" — via the HMM-given Viterbi maximal-likely sequence, and by K-Means (without HMM at all). The first approach is not independent of the HMM (it still uses the HMM, so there is a fear of whether our results are not an artifact of the HMM); while the latter one's intervals are usually pretty short, and are probably instable (any change in the small slice causes the interval to be interrupted, while the HMM smooths away these fluctuations). Example: Suppose that some submit time bin is closer to state 1, but both its neighbors are close to state 2 and it itself is pretty close to state 2 (but closer to state 1). In this case, HMM Viterbi that looks for the most likely sequence can cluster the submit time bin with state 2; while K-Means does not learn the interconnections between neighboring submit time bins. Therefore in this work I preferred using HMM Viterbi for state modeling. In fact, Geometric distribution is the maximal entropy distribution over the natural numbers with a given mean; therefore it can be thought of as a default distribution (due to the principle of maximum entropy of Jaynes).

Table 3.1 presents the KL-divergence of the state interval distributions as given by the Viterbi run over the trace per state and its maximally likely Geometric distribution model. The different columns represent the different estimations of the KL-divergence: the Maximal Likelihood Estimation, the estimation with Simple Miller-Madow bias correction and with bias correction that uses the non-empty bin number estimation assuming Geometric distribution. It shows that KL-divergence estimation after bias correction is usually pretty little for most of the states. The exceptions are the states that model the empty time slices (like state 11 and 8). The bias correction is important, as noted earlier; but both estimators for the expected number of appearing values give very close results; therefore from here on only the Simple estimator is used.

The conclusion is that the state durations are indeed usually distributed geometrically. But what does it mean? Suppose I know $q_t = i$ — the state at submit time bin t. What is the probability of being in the same state at submit time bin T, meaning what is the $\Pr(q_t = q_T)$? There are two possibilities — either this is the same state interval, or the state interval is broken, meaning two different intervals. But the dependence between the states in consequent

State	KL MLE	KL corrected, simple	KL corrected, geom.	Notes
0	0.03 bits	0.00 bits	0.01 bits	
1	0.07 bits	0.01 bits	0.01 bits	
2	0.07 bits	0.02 bits	0.03 bits	
3	0.03 bits	0.00 bits	0.01 bits	
4	0.02 bits	0.00 bits	0.00 bits	
5	0.12 bits	-0.08 bits	-0.05 bits	
6	0.09 bits	0.04 bits	0.04 bits	
7	0.09 bits	0.06 bits	0.06 bits	
8	0.56 bits	0.30 bits	0.28 bits	23% empty
9	1.24 bits	0.76 bits	0.77 bits	heavy tailed
10	0.19 bits	0.07 bits	0.08 bits	
11	0.23 bits	0.14 bits	0.12 bits	36% empty
12	0.07 bits	0.03 bits	0.02 bits	
13	0.02 bits	0.02 bits	0.01 bits	
14	0.03 bits	0.01 bits	0.01 bits	
15	0.02 bits	-0.00 bits	-0.00 bits	

Table 3.1: KL-divergence between State interval duration and its maximal likely Geometric model. SDSC Paragon 95, t_s =1.5 hours, $t_r = 1.8$, $M = 16$, Static-IV, L_∞ convergence, $\epsilon = 10^{-5}$.

submit times is as follows $\Pr(q_T|q_t, q_{t+1}...q_{T-1}) = \Pr(q_T|q_{T-1})$. If for any two different states $\Pr(q_T = i|q_{T-1} = j) = \Pr(q_T = i)$, then, if we are interested in the state inter-dependence, we should focus in the fact that we are in the same interval. The probability that we stay in the same interval is proportional to some α^{T-t}. It is exactly the survival function — the probability that the state duration interval is at least $T - t$; and since the distribution is geometric, it is exponentially decreasing in time.

There are many adaptive algorithms that exponentially decrease the significance of the observed signal as the time passes. For instance, the Exponential Moving Average (EMA) is an on-line technique that holds a current average as a state variable, and each time unit inputs the signal and updates the average by the formula $\hat{\mu} = (1 - \alpha)X + \alpha\mu$, where $\mu, \hat{\mu}$ are the average before and after the signal respectively, X is the new input signal and $\alpha \in (0, 1)$ is a parameter. It means, that after the infinite (or very long) signal series of $...X_t....X_1X_0$ the state variable $\mu = (1 - \alpha)\sum_t X_t\alpha^t$; therefore the weighting factor decreases exponentially.

This leads to ideas of what should be done when the number of samples is very small and the HMM ergodic model cannot be learnt effectively due to the very large number of the parameters, like in the case when learning from jobs of a single user. In these cases, the assumption may be that the only interesting value is α — the rate at which the information becomes old. After this, each user's jobs can be modeled with respect to their recency; thus combining user

information with locality as proposed in the end of section 2.4. This part was not done in this work, and it is left for the future.

3.5.1.1 Load Distribution

There were several distributions that were proposed for load models. The basic Poisson appeared pretty weak — it is too concentrated around the average load, with no significant support for tails. There were proposals for heavy-tailed models, that appeared to work pretty well in some cases — Zeta, Log-Normal, Pareto and Shifted Pareto (the latter distribution is not widely known, but it is used in Internet Traffic modeling, see [1]). The CDF of Shifted Pareto is $F(x) = 1 - (1 + \frac{x}{k})^{-a}$ (compare to regular Pareto $F(x) = 1 - (\frac{x}{k})^{-a}$). If both parameters converge to infinity, then for each x the CDF converges to $1 - e^{-cx}$, where $c = \lim \frac{a}{k}$. It means that there are cases when Shifted Pareto converges to an Exponential distribution. The convergence in this case is a point-wise convergence only, since for any finite a, k Shifted Pareto remains heavy tailed.

When the chosen distribution model is continuous, the discretization is used by calculating the probability of the floor function — $p(n) = \Pr(\lfloor x \rfloor = n) = F(n+1) - F(n)$, where $F(x)$ is the CDF of the function. As for the Zeta distribution, a shift of 1 was used since the Zeta distribution does not support 0 (and there are empty submit time bins).

Finding the maximal likelihood for most of the described above distributions is pretty simple. It is well known that the maximal likelihood estimator for Poisson's λ parameter is the average of samples; for Geometric distribution's "success probability" p is the sample average's reciprocal; the log-normal distribution parameter estimators (for continuous case) are also well known. As for the Zeta distribution, its s parameter is estimated as follows: $f(x) = \frac{(x+1)^{-s}}{\zeta(s)}$ for a single observation, log-likelihood is $l(x) = -s \sum \lg(x_i + 1) - n \lg \zeta(s)$; derivative is $\frac{\partial}{\partial s} = -\sum \lg(x_i + 1) - \frac{n\zeta'(s)}{\zeta(s)}$. It means that in stationary point of s, $\frac{\zeta'(\hat{s})}{\zeta(\hat{s})} = -\frac{\sum \lg(x_i+1)}{n}$; and this was solved manually with Matlab. The problem comes with Shifted Pareto. Derivative calculation leads to very complex formula even in continuous case — $f(x) = \frac{a}{k}(1 + \frac{x}{k})^{-a-1}$ for a single observation, meaning the log-likelihood is $l(x) = n \lg a - n \lg k - (a+1) \sum \lg(1 + \frac{x_i}{k})$; the derivatives are $\frac{\partial}{\partial a} = \frac{n}{a} - \sum \lg(1 + \frac{x_i}{k}) \Rightarrow \hat{a} = \frac{n}{\sum \lg(1+\frac{x_i}{k})}$; $\frac{\partial}{\partial k} = -\frac{n}{k} + (a+1) \sum \frac{x_i}{(k+x_i)k}$. Substitution of the formula that was received from a's derivative to the second makes it intractable. Therefore, location of the optimal points was done with use of iterative location of optimal value, with use of Spreadsheet, manually.

Figure 3.5 shows the LLCD's of different states of different traces. The settings are shown in the following table, and they heavily differ from one another. However, as presented down here, both HMMs have states with heavy-tailed load distribution and ones with geometric load distribution.

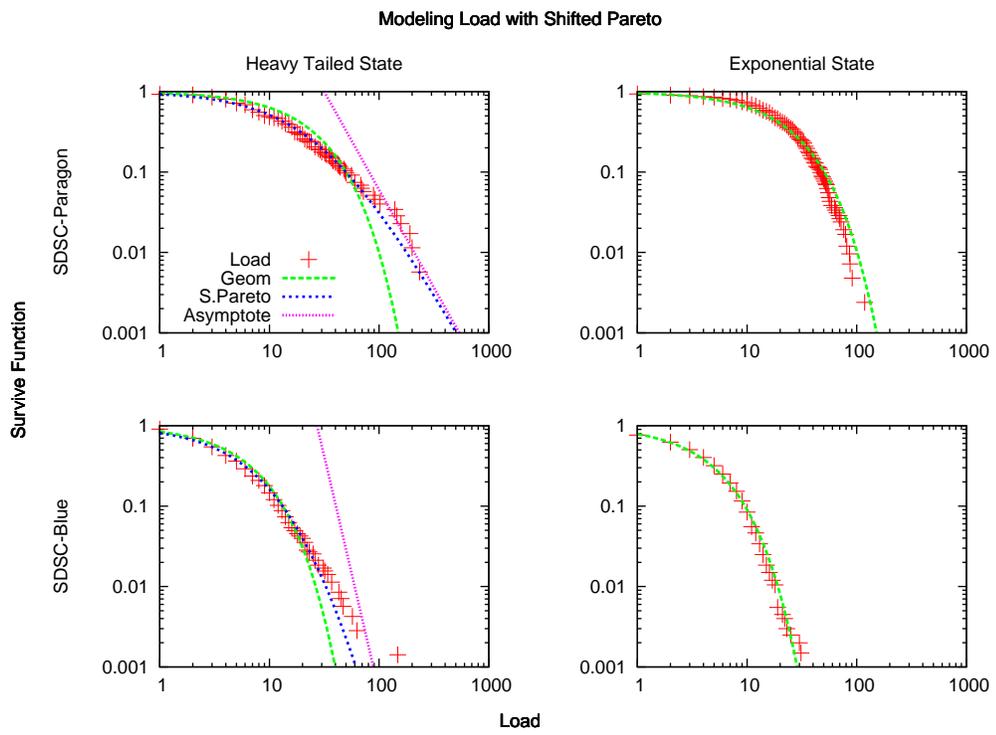| Trace	M	t_r	t_s	Init. values	Termination
SDSC Par95, full	16	1.8	1.5 hours	Static-IV	$L_\infty < 10^{-5}$
SDSC Blue, first half	16	1.19	15 minutes	Static-IV	100 rounds

Figure 3.5: Modeling the load distribution with Shifted Pareto.

Also, the figure presents the best explaining geometric model and optimal Shifted Pareto model (in cases where they differ). The conclusion is that the Shifted Pareto is pretty good model for any case! It covers both geometric and heavy tailed distributions. As noted above, the convergence is point-wise, and is learned with finite sample size; therefore one should be careful with extrapolation of this distribution to the tail; but it is possible to switch the distribution when the optimal parameters of Shifted Pareto tend to infinity.

3.6 Conclusions

It is possible to model the runtime distribution of jobs with an HMM. It learns how long the information is still fresh by determining the state duration probability, it learns the cross-state connections, and is pretty simple to use.

The best practice to run HMM is setting short submit time bins. The correct state duration is learnt by the Baum-Welch itself. There are traces when using many runtime bins and states improves the metrics; but usually 16 states are enough.

Modeling the load is important in cases when we want to predict the future jobs or characterize the load; but when all that we are interested in is runtimes then the load should not be modeled — it complicates the model, and may degrade the results. The model selection is critical.

Static initial values reduce the introduced over-fitting and they usually give (eventually) better performance in terms of the surprise measure and log-likelihood. However, they require more rounds to reach convergence.

However, at this point some things remain open. First of all, HMM models the sequences of the job runtimes. Meaning, the information gain is measured as the increasing of Log-Likelihood of the runtime sequence divided by number of jobs. If the model is good, the sequence becomes more probable. But the number of sequences grows exponentially with number of jobs. Analyzing each such sequence independently is extremely expensive in terms of runtime complexity. The scheduler may compare pairs of jobs, or sort the jobs somehow; but it cannot analyze the probabilities of all the possible sequences.

Another issue is the fact that the HMM's Baum Welch, like any learning algorithm, requires some training set. The schedulers, on the other hand, are usually on-line. At the beginning of a trace the training set is very small. Therefore, the initial learning may suffer from over-fitting.

These two topics are the main issues of the following chapter.

Chapter 4

Single Job Runtime Distribution Modeling

As mentioned at the end of the previous chapter, giving probabilities to observation sequences is not of great interest. Instead, modeling a single job's runtime may be much more useful for the scheduler. There are two main differences between the previous chapter and this one: First, here we try to learn the HMM on-line, having less information and limited time to run the learning process (meaning, limited number of Baum-Welch rounds). Second, we try to assess how much can one learn on a single job instead of the complete sequence.

In order to make the process on-line, one must decide when the information (in our case, runtime) is available and when it is modeled. In this chapter, the trace is used as is; meaning, all we try to do is predict the runtime, but we don't re-schedule the jobs. Therefore, the real job runtime is known when the job is finished. In the next chapter, the scheduler is planned to use the job runtime model to decide when to start it; therefore the runtime distribution is modeled once, at the job's submission, and while the job is in the system its runtime is not re-modeled (with a single exception described later on).

Suppose we predict a distribution $p(r_1)..p(r_N)$ for some job's runtime; then the surprise measure of the job is $-\log_2 p(r_j)$, where r_j is the runtime bin of the job. The average information gain between two distribution models — one is created by the HMM that is run over all the jobs that started before the job's submission, and the other is the distribution of runtimes of all the previous jobs (meaning, all the jobs that have finished till this job submission) — is used as a metric for success of the modeling the distribution of a single job with the HMM.

Another question before one models anything on-line is when to start the learning process and when to stop it. Two options were used. One is running the Baum-Welch some number of rounds each week. Another, more promising, is running Baum-Welch till L_1 metric convergence, meaning till the squared root of the number of terminated runtime jobs multiplied by the L_1 metric is less

than the threshold; and restarting the Baum-Welch once the number of runtime jobs grows bigger and the termination condition for the last L_1 metric value is not satisfied.

Short jobs usually tend to end before the jobs with longer runtime, due to their shorter runtime. The result of this is the fact that the short jobs contribute their data and therefore affect the modeling faster, before the long ones. For example, suppose that two jobs are submitted at submit time bin $t-1$, one long and another one short; and both start immediately. At submit time bin t the short job already finishes and the long one still runs. Adding the short job alone to the information base means inserting the information that the neighboring submit time bin was a submit time bin populated with the short jobs; while ignoring the currently running long job. Thus, currently running jobs must also contribute to the modeling. We know that these jobs run longer than what they have run so far; so this job doesn't belong to some short runtime bins. Its contribution to the probability is the multiplication by $\Pr(r \geq r_{now}|q_t)$ — the probability that job runtime is at least the current runtime. This probability is modeled as the sum of probabilities of the runtime bins where this job can belong. For simplicity, no interpolation was used.

The component that creates the model of a job's runtime is named a Predictor. The essence of the Predictor's interface includes three methods:

- `submitJob()` method indicates a job's submission. It returns the distribution model of the runtime of the job.

- `startJob()` method indicates a job's start. This is required only for supporting the previously described feature of including long jobs in consideration.

- `terminateJob()` method indicates a job's termination. Only at this point the exact information on the job's runtime is available. It is inserted into the HMM's runtime sequence.

Some notes on the Predictor implementation: First of all, Baum-Welch and distribution modeling procedures run on different sets of data. Baum-Welch runs on terminated jobs only. Meaning, the Observation sequence period ends before the first non-terminated submission. This ensures that the learning is done on exact data only. On the other hand, the distribution modeling procedure runs on all the jobs that are already started. There are several reasons for this decision. First is the fact that Baum-Welch tries to locate the optimal model parameters to explain the *complete* Observation sequence and therefore removing non-exact information from one end should not affect it much; while the goal of the distribution modeling is to understand what happens right now, and not some global model.

Another reason why this non-exact information is not used by Baum-Welch is the simplicity — Baum-Welch is a special case of Expectation-Maximization; and it is not clear how to reset the model parameters to maximize the log-likelihood as required. Even if there is only a single job with submit time t_0 and

the runtime that belongs to either of the two last bins, the formula becomes very complex: Log-likelihood is $\sum_{t,i,j} \gamma_t(i) x_{j,t} \ln \hat{B}_{ij} + \sum_i \gamma_{t_0}(i) \ln(\hat{B}_{i,N} + \hat{B}_{i,N-1})$, when the left term is as previously, considering terminated jobs only, and the right one inserts the probability of the running job. This means, for $j = N - 1, N$, meaning for the last two bins, the Lagrange multiplier technique requires $\frac{\sum_t \gamma_t(i) x_{j,t}}{\hat{B}_{ij}} + \frac{\gamma_{t_0}(i)}{\hat{B}_{i,N} + \hat{B}_{i,N-1}} = c_i$ — the divisor of the second term contains multiple model parameters, and this complicates calculations even for this degenerate case. For the general case, this is hardly ever tractable.

In order to support this feature, the Predictor holds at each time an ordered set of the submit times of the jobs that are currently in the system (more correctly, it is a multi-set, since two jobs may arrive at the same time). The Baum-Welch last submit time bin doesn't include any job in the system — meaning, the end point is the minimal submit time bin of a job in the system.

Another issue is the fact that the Predictor may under some circumstances decide not to return the distribution model on a job's submission. This may happen due to two reasons: either there are not enough samples to learn from — the Predictor doesn't start running Baum-Welch before all the jobs from the initial period finish (in this work, this period is one week long), or the current model cannot explain something in the observation, and the model must change. The last reason is since the data is not known from the start, and the initial values may represent the training set pretty well; but they may suffer from over-fitting.

The Predictor works in isolation from all the irrelevant items — all the information it uses is submit, start, and termination times. Jobs' user and user estimates attributes are not used. The Predictor-based scheduler may use the user estimates as described later on, but the Predictor itself doesn't use it.

Consider the following scenario: The system runs for about a year, and some new job arrives. The forward algorithm that is used to calculate the distribution of the runtimes of the jobs in the current submit time bin (see [10]) in a regular way would calculate all the $\alpha_t(i) = \Pr(q_t = i, O_1..O_t | \theta)$. However, unless the model changes, the recalculations for all the old submit time bins give the same values. Therefore, saving the previous calculation results of $\alpha_t(i)$ for all t that don't contain running jobs may reduce the total runtime cost significantly, since they don't change (unless Baum-Welch changes the model parameters, but this happens very rarely. However, it is important to remember to treat this special case correctly).

Figure 4.1 presents the average information gain per week, and the global average gain (as described earlier). The average gain is less than the performance of the HMM when using the sequence gain. There are several reasons for this. One is the fact that the information is not available always, and this explains why the results are not very good for the first weeks; but this is not the only reason for this deterioration in results. The most important factor in my opinion is the fact that when predicting the runtime of the currently submitted job the neighboring jobs (the jobs that were submitted recently before it) are still in the system and their runtime is unknown. The nature of the HMM indicates that
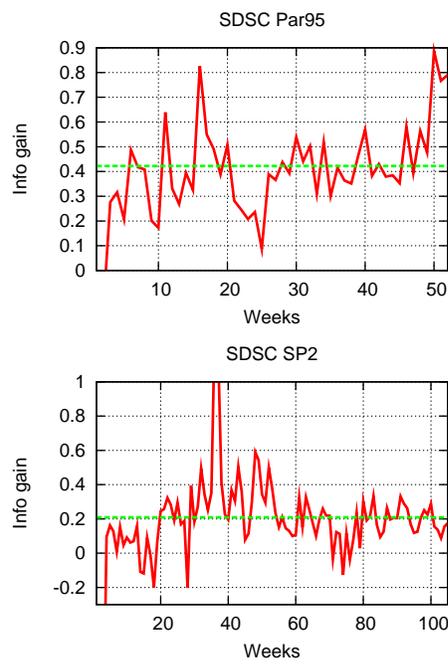
Figure 4.1: Average information gain during each week and globally (dashed line). $M = 16$, $t_r = 1.8$, $t_s = 15min$, $\epsilon = 200$

neighboring jobs are similar in runtime; but it is unclear what are they similar to. Therefore, the runtime of a job is modeled from some partial information on the recent jobs, and better information on the earlier jobs (whose contribution is usually less dominant).

Chapter 5

Distribution Models Usage in Scheduler.

Previous chapters presented the framework for modeling the jobs' runtime distribution. This chapter discusses the ways how this information can be practically used by a scheduler. The scheme of proposed work is as follows

1. Choose an algorithm that knowing (or guessing) the runtime of a job can improve the chosen metric.

2. Convert the algorithm to use the probability distributions instead of single values.

In the traces of [6], for jobs that are canceled before they start, the runtime is 0 seconds and the number of used CPU's is also 0. Those jobs were removed from the model. If a job requires more CPUs than the machine has, the requirement is aligned to the machine size.

In order to avoid the influence of the runtime differences between the traces I used waiting time for the performance metric. The system is a multi-user system, therefore fairness is also an issue. Therefore, the L_1-type metrics that take the average or sum of all the jobs' metric values are not enough — a job that suffers from bad service is not compensated by the fact that in average the jobs wait little in the queue. In order to present the complete picture of what is going on for all the jobs the full CDFs of the waiting times are presented.

5.1 Algorithms that Use Predictions

There are many different algorithms that use predictions or user estimates of job runtimes, including EASY-backfilling and shortest-job-first (SJF). The problem with using runtime predictions other than user estimates with backfilling is the fact that the jobs may be under-predicted. Killing the jobs in this situation

is highly undesirable, since the users have neither tools to avoid it nor indication that this is going to happen. Therefore, the only reasonable way to solve under-prediction is postponing the reservation for the first job in the queue [4]. But there is no promise this postponing will ever stop, unless we forbid future backfilling — the backfilled jobs may in their turn be under-predicted. The same question arises when the predictions are initially set too large, like when using doubling (or tripling, quadrupling and so on) [4]. If the system were a single-user system, then this strategy would probably be good — it pushes forward the jobs with less requirements (on average), so the average waiting time is expected to decrease. However, since we are dealing with multi-user systems, such an approach is insufficient — it may appear extremely unfair.

In this work, however, I used EASY-backfilling as the base algorithm. According to [4], when the predictions are correct, the overall performance of EASY-backfilling usually improves.

5.2 Using Distribution Models in the Scheduling Algorithm

EASY-backfilling holds a queue of waiting jobs (ones that have been submitted but have not yet started) ordered by their submission times. The steps of the EASY-backfilling scheduling procedure, which is executed each time a job arrives or terminates, are as follows:

1. As long as there are enough idle CPUs to start the first job in the wait queue, remove this job from the queue and start it.

2. Given the first job in the queue that cannot start because of insufficient idle CPUs, find when the required number will become free and make the reservation for this job.

3. Continue scanning the queue, and start ("backfill") jobs if they don't violate this reservation.

However, the idea of the algorithm can be expressed more concisely. In fact, Step 2 is for performance improvement only. Steps 2 and 3 can be united as follows: "Continue scanning the queue, and start jobs if this doesn't postpone the start time of the first job in the queue" (assuming runtimes of jobs to equal their user estimates). In EASY-backfilling, job runtime predictions (user estimates) are used to decide whether starting a job will postpone the first queued job. Therefore, the algorithm remains the same with the only change of backfilling condition.

In base EASY-backfilling each job is assigned a single value of its predicted runtime, and this prediction is used as the exact runtime in a very deterministic way. But if we don't have a single-value prediction, but rather a distribution, it is not possible to make such a decision in a deterministic way. Instead, there are

many cases with different probabilities that may contradict each other. Therefore the algorithm receives a single parameter that is the confidence probability τ. The backfilling should happen if and only if the probability that the base algorithm that works with single values would not perform backfilling is less than τ; meaning, the probability that the backfilling postpones the start of the first job in the queue is less than τ.

For simplicity, it is assumed that the job runtimes are independent; meaning, for each two jobs with runtimes R_1, R_2, $\Pr(R_1, R_2) = \Pr(R_1)\Pr(R_2)$ (although the HMM works assuming it is wrong; as usual, here and everywhere, $\Pr(R_1)$ denotes the probability of random variable R_1 to have its value). In particular, this means that the event of the availability of CPUs at different times due to terminations of the currently running jobs and the distribution of the backfilled job's runtime are independent.

Suppose the current time is t_0; the termination time of the backfilled job is t_e (if it is backfilled); $c(t)$ is the number of CPUs that are released by the currently running jobs before and including time t; and c_0 and c are the number of CPUs that must be released to start the first job in the queue and both jobs respectively. The algorithm should backfill iff

$$\Pr(\exists t \in (t_0, t_e) : c_0 \leq c(t) < c) < \tau$$

— the probability that there exists some time t before termination of the backfilled job when the number of released CPU's is enough to start the first job in the queue but not enough to run both jobs, so the backfilling postpones the start of the first job in the queue. Integrating over all the possible termination times of the backfilled job we receive

$$\int \Pr(t_e, \exists t \in (t_0, t_e) : c_0 \leq c(t) < c)dt_e < \tau.$$

Since by assumption of job runtimes independence t_e and $c(t)$ are independent, this probability is

$$\int \Pr(t_e)\Pr(\exists t \in (t_0, t_e) : c_0 \leq c(t) < c)dt_e < \tau.$$

The left factor in the integrand is modeled by the Predictor — it is exactly the current time plus the runtime. The probabilities are modeled discretely, only at the ends of the runtime bins. The right factor is much harder to calculate.

First of all, in order to calculate the right factor, we must calculate the probability $\Pr(c(t) \geq c)$ for any given time t and any given requirement c. The probability of CPU availability given termination probabilities at time t of the currently running jobs is calculated using Dynamic Programming, where the probability that jobs $1..n$ released at least c CPUs (represented as $M_t[n, c]$) equals

$$M_t[n-1, c] + (M_t[n-1, c-c_n] - M_t[n-1, c]) \cdot P_t[n]$$

— the left term denotes the case when the processors are idle without termination of job number n and the right term is the probability that the jobs $1..n-1$

freed at least $c - c_n$ but not c, and the last job terminates (c_n is number of CPU used by job number n, and $P_t[n]$ is the probability that job number n terminates not later than t). The initialization of the Dynamic Programing sets the obvious values: $\forall c \leq 0, M_t[*, c] = 1$ — the number of released CPUs is always a non-negative number; $\forall c > 0, M_t[0, c] = 0$ — zero jobs release zero CPUs, no more. In the algorithm implementation, these values may be calculated on-the-fly; for instance, if $c < c_n$ (the number of required CPUs is smaller than the number of CPUs used by job number n), then $M_t[n - 1, c - c_n]$ doesn't exist in the real matrix, because the index is negative: $c - c_n < 0$; but can easily be substituted by 1, and $M_t[n, c] = M_t[n - 1, c] + (1 - M_t[n - 1, c]) \cdot P_t[n]$. (By the way, this means that $M_t[n, c] = 1 - (1 - M_t[n - 1, c])(1 - P_t[n])$ — this is the probability of disjuction of two independent events: either jobs $1..n - 1$ release c CPUs, or job n terminates. Indeed, in this case the required CPUs cannot be cumulated from different jobs). If n is the number of running jobs, then $\Pr(c(t) \geq c) = M_t[n][c]$.

This requires calculating the probabilities of running job terminations before or at the time t (denoted earlier as $P_t[]$). For this goal a new concept of a *probability event* is introduced, and each such event represents the possible termination of a job. Because our data has been discretized, the job runtime probabilities are estimated only at the ends of the runtime bins. The upper and lower bounds of job runtimes are the user estimate (since the job is killed after it; this is used even before the job starts) and the current runtime of the job (`currentTime-job.startTime`). Log-Uniform intra-bin interpolation is used. Algorithm 5 presents the recalculation procedure for the runtime bin probabilities. **Line 3** receives the job distribution model as proposed by the Predictor (reminder, N is the number of the runtime bins, and $j = 1..N$ is the index of a runtime bin). **Lines 9-12** ensure that the new runtime bin boundaries satisfy the old bin boundaries and global boundaries. If the runtime bin doesn't intersect the global boundaries then `newBinStart==newBinEnd` and therefore `new_p[j]=0`. **Lines 13-15** recalculate the probability measure remainders after Log-uniform interpolation.

After all the probability events are inserted to a list and sorted by the time, one can easily calculate the vector of termination probabilities at time t.

Let $A(t)$ be the event that t is the real start time of Q[0] (the first job in the queue) without backfilling, and the backfilling of the job breaks the reservation. This means that

$$A(t) = (t \in (t_0, t_e)) \wedge (\forall s < t, c(s) < c_0) \wedge (c_0 \leq c(t) < c)$$

— t is before the end time of backfilled job and t is the first time when the first job in the queue can start but only if this job isn't backfilled. Therefore, the backfilling should happen iff

$$\Pr(\exists t \in (t_0, t_e) : A(t)) < \tau \Leftrightarrow \int_{t_0}^{t_e} \Pr(A(t)) dt < \tau$$

— the events are disjoint, therefore the total probability is the integral of probabilities. The problem is to calculate $\Pr(A(t))$.

Algorithm 5 Runtime bin probability recalculation

```
 1 double[] recalculate(Job job)
 2    // old model
 3    double old_p[N] = job.model;
 4    // new distribution model
 5    double new_p[N];
 6    double upperBound = job.userEstimate;
 7    double lowerBound = currentTime-job.startTime;
 8    for each runtime bin j do {
 9       double newBinStart =
10          max{bin[j].start, min{lowerBound, bin[j].end}};
11       double newBinEnd =
12          min{bin[j].end, max{upperBound, bin[j].start}};
13       new_p[j] = old_p[j]*
14             (log(newBinEnd )-log(newBinStart )) /
15             (log(bin[j].end)-log(bin[j].start));
16    }
17    normalize(new_p);
18    return new_p;
19 }
```

Suppose $t \in (t_0, t_e)$. Let us change the definition of t to be discrete time (in any units). Due to the monotonicity of $c(t)$, $\Pr(A(t)) = \Pr(c(t-1) < c_0 \wedge c_0 \leq c(t) < c)$. If $c(t) \geq c$ or $c(t-1) \geq c$, then $c(t) \geq c_0$, since $c > c_0$ and $c(t)$ is monotonous (see Venn diagram in Figure 5.1). Therefore,

$$\Pr(A(t)) = \Pr(c(t) \geq c_0) - \Pr(c(t-1) \geq c_0 \vee c(t) \geq c) =$$

$$= M_t[n][c_0] - \Pr(c(t-1) \geq c_0 \vee c(t) \geq c).$$

But in the right term, both events in the disjunction don't imply each other, so

$$\Pr(c(t-1) \geq c_0 \vee c(t) \geq c) =$$

$$= \Pr(c(t-1) \geq c_0) + \Pr(c(t) \geq c) - \Pr(c(t-1) \geq c_0 \wedge c(t) \geq c) =$$

$$= M_{t-1}[n][c_0] + M_t[n][c] - \Pr(c(t-1) \geq c_0 \wedge c(t) \geq c)$$

The last term is pretty hard to calculate. However, it has a lower bound of $M_{t-1}[n][c]$ — the probability that before the last event there were enough CPUs to run both jobs (which implies $c(t-1) \geq c_0$ and $c(t) \geq c$). Using all the above considerations leads to the bound

$$\Pr(A(t)) \geq (M_t[n][c_0] - M_t[n][c]) - (M_{t-1}[n][c_0] - M_{t-1}[n][c])$$

The sum of these lower bounds is the telescoping series; and since the first item equals 0 (because initially the number of CPUs is less than c_0), the total
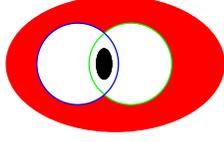
Figure 5.1: Explanation of the $\Pr(A(t))$ formula. The big ellipse represents the event that there are enough CPU to run Q[0] at time t — $c(t) \geq c_0$. The left circle represents the event of enough CPU at time $t - 1$ — $c(t - 1) \geq c_0$. The right circle represents the event of having at time t enough CPU to run both jobs — $c(t) \geq c$. Both last two events imply the first event; but their interdependence is not simple. The goal is to measure the filled area outside the circles. The problematic area is the intersection of the left and right circles. The solution is to calculate the smaller area of $c(t - 1) \geq c$, represented as a black ellipse in the intersection.

sum is $M_{t_e-1}[n][c_0] - M_{t_e-1}[n][c]$. But although each of $M_t[n][c_0], M_t[n][c]$ is monotonically growing as a function of t, their difference is not monotonous; while all $\Pr(A(t)) \geq 0$, and their sum is monotonous. This means, we have a tighter bound of

$$\sum_{t \in (t_0, t_e)} \Pr(A(t)) \geq \max_{t \in (t_0, t_e)} \{M_t[n][c_0] - M_t[n][c]\}$$

To summarize, EASY-backfilling with distribution models works as follows: The scheduler backfills iff $\sum_{t_e} \Pr(t_e) \max_{t \in (t_0, t_e)} \{M_t[n][c_0] - M_t[n][c]\} < \tau$ — the probability that such time exists is less than the threshold.

Algorithm 6 presents the simplified pseudo-code of the EASY+HMM backfilling scheduler. Some notes on implementation. The `result` variable is monotonously growing, so once it is bigger than the threshold the total result is false for sure, so no further calculations are run. The `pMax` variable is also monotonously growing. This means that if the remaining runtime bin probability multiplied with the current `pMax` together with the current `result` are bigger than the threshold, it is also enough to stop calculating and return false. These improvements are very important, since the scheduler runs on-line.

If the Predictor returns no runtime prediction (as you might remember, it is one of the options), then the single probability event is inserted, which is the user estimate with probability of 1. If this is the case for all the running jobs, then the algorithm works exactly like EASY without an HMM: All the Probability Events come from the running jobs' terminations by user estimates, and therefore the algorithm works in a very deterministic way.

5.3 Results

In this work, the threshold used was $\tau = 0.05$. HMM parameters were set as follows: $M = 16$, $t_s = 15min$, $t_r = 1.8$, no load modeling, Static-IV, L_1

Algorithm 6 The test of EASY-backfilling that uses distribution-based predictions.

```
bool shouldBackfill(Job job) {
    List events =
      <the list of probability events,
            sorted by time>
    int n = <# of running jobs>
    double P[n];
    // max(t){M[n][c]-M[n][c0]}
    double pMax = 0;
    double result = 0;
    int c0 = <# of CPUs that must be idle to run Q[0]>
    int c =  <the same, to run both jobs>
    for each j=runtime bin do {
       for each e in events before bin[j].end do {
           P[e.job] += e.probability;
           <calculate M using Dynamic Programming, given P>
           pMax = max{pMax, M[n][c0]-M[n][c]};
       }
       result += job.model[j]*pMax;
    }
    return result < THRESHOLD;
}
```

Trace name	EASY	EASY+HMM	
CTC SP2	21.3 min	18.1 min	-15.2%
SDSC SP2	364 min	373 min	+2.6%
SDSC Blue	131 min	105 min	-19.5%
KTH SP2	114 min	113 min	-0.6%
Total			-8.7%

Table 5.1: Arithmetic mean of waiting times.

Trace name	EASY	EASY+HMM	
CTC SP2	28.2 sec	25.3 sec	-10.1%
SDSC SP2	639 sec	635 sec	-0.7%
SDSC Blue	203 sec	135 sec	-33.6%
KTH SP2	181 sec	147 sec	-18.9%
Total			-16.7%

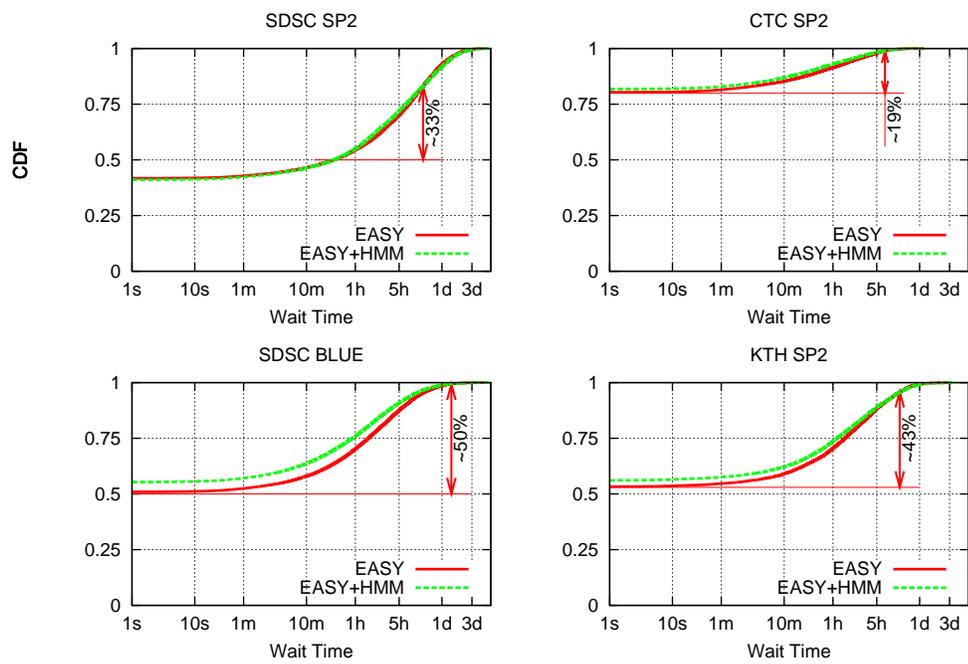Table 5.2: Geometric mean of waiting times.

Figure 5.2: EASY+HMM vs. EASY, CDF of waiting time. The arrows show the jobs that enjoy from addition of the HMM.

convergence criterion, $\epsilon = 200$. Figure 5.2 compares the EASY scheduler with the probability-based scheduler. The X-axis is the waiting times of the jobs in logarithmic scale, and the Y-axis its CDF. The CDF doesn't start from 0, since there are jobs that don't wait in the queue at all. The arrows represent the jobs whose waiting time improved due to the HMM — this is the interval where the dashed line (EASY+HMM) is left of the solid line (EASY).

The conclusions of this chart is that usually most of the jobs are better off using the HMM. Note that the x-axis is logarithmic, with a very small scale — it changes $2.5 \cdot 10^6$ times. Therefore, when the line moves left even for a little, this may represent an improvement factor of 2. Also, it looks that if the job started waiting, it usually waits for at least 10 seconds. Another finding — there is place in the chart where the line is almost straight. This means that the waiting time distribution at some intervals is close to a log-uniform distribution.

Tables 5.1 and 5.2 show the improvements of metrics — arithmetic and geometric means of wait times (note that the calculation formula for the geometric mean is $\exp(\int f(W) \ln \max\{W, W_{\min}\} dW)$, where W is the job's waiting time, $f(W)$ is its PDF and $W_{\min}$ is the commonly used threshold of 10 seconds, see for instance [4, bounded slowdown formula]. Therefore, the improvement in the geometric mean metric value is exactly the area between the lines of the chart that are right to $W = W_{\min}$).

Chapter 6

Conclusions

Locality First of all, neighboring jobs are similar in their runtimes. The main cause for this is indeed the temporal locality of user sessions. But even if one knows the user of the job, knowing its submit time is still important. It seems that at different times the same users submit different jobs to the system.

However, it looks that in order to learn runtime information from the jobs, one should use imprecise but quick adaptation. This adaptation must derive from a small number of jobs, and this restricts the level of detail. Of course, when the required level of detail rises the required learning time should increase as well; but this gives less accurate predictions, since the system and its jobs change very quickly.

When measuring the locality, it is important to leave the data "as is", without clustering with K-Means (or any other learning algorithms). Simple binning is more preferable, both due to simplicity and since it doesn't learn (and therefore doesn't "hide") information.

HMM Hidden Markov models can learn a major part of the information on the jobs' runtimes. A variety of models was suggested and tested. The conclusion is that although there are some common guidelines, every application should use the model that is appropriate to its needs. For instance, if all the application should know is the runtimes of the jobs, then it is more desirable to remove the load from the model.

HMM's Baum-Welch procedure locates the optimal state interval length. Therefore, submit time bins usually should be short for better resolution (usually, not longer than 30 minutes). The runtime bins should be set together with the number of the HMM states. HMM states must be able to tell enough information.

Usage of static initial values include no initial over-fitting. Note, however, that setting some initial values may be very far from the real world; therefore it is usually wise to update them when recognizing that they are very far.

The states of HMMs were thoroughly characterized. Choosing the correct model is a crucial issue. An incorrect load model, for instance, may degrade the

results sharply. This work found that a Geometric distribution of state duration intervals and submit time loads is usually appropriate.

A possible use of the HMM may be creating the workload by pattern reconstruction. This way, given a trace, one may use the HMM to extend it with Pattern Generation techniques. This may be a future research direction of HMM usage.

Another future research direction is different static initial values. For instance, instead of focusing on different runtime bins, they may focus on different means of runtime bin number. In this case, the initial distribution of state i might be the maximal-entropy distribution with a specified mean μ_i. The maximal-entropy distribution for a given mean satisfies $B_{ij} = c_i p_i^j$ for values of p_i, c_i that satisfy $\sum c_i p_i^j = 1$ and $\sum j c_i p_i^j = \mu_i$. This is the "default" distribution for a state with a given mean of runtime bin number, due to the maximal-entropy principal. Thus, the states will expect the jobs with some means, and not concentrate on specific bins.

Scheduler Before using the HMM within the scheduler (or any other on-line algorithm), one must remember that the HMM model cannot apply as is to everything in the same manner. One challenge when using HMMs is the fact that the HMM gives the probabilities for observation sequences (in our case, runtime sequence) and not for a single observation (a single job's runtime). This causes deterioration in results when modeling a single job's runtime. Also, some schedulers should distinguish between the jobs that are usually neighboring in time. For instance, the shortest-job-first algorithm should run the job that is the shortest of the currently waiting ones. But if two jobs are both submitted in the same submit time bin, then their distribution models are very close to one another (in fact, they should equal; the difference is an artifact of the modeling times). Therefore, for some algorithms the HMM model should be used very carefully. This indicates a possible future work direction, where the model includes not only the submit time, but also other job parameters, like the submitting user. Note, however, that there are much fewer jobs per a user, and thus the model must be simpler.

The work demonstrates usage of the HMM with the EASY-Backfilling algorithm. The overall algorithm performance improves almost for all the traces for the majority of jobs. Another future research direction maybe using the HMM with other algorithms. For instance, shortest-job-backfill-first [4] may be converted to use the distribution model. Another important issue is selecting the metric for resource availability time predictability.

Bibliography

[1] URL http://www.samsi.info/marron/Int/Notes03-09-09/ SAMSIstat321Internet03-09-09web.pdf.

[2] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press: Cambridge, UK; New York, 2004.

[3] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley & Sons, New York, 1991.

[4] Yoav Etsion Dan Tsafrir and Dror G. Feitelson. Backfilling using runtime predictions rather than user estimates. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, To Appear.

[5] Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour. Economic scheduling in grid computing. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 128–152. Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[6] Dror G. Feitelson. Parallel workloads archive. URL http://www.cs.huji.ac.il/labs/parallel/wo rkload.

[7] Domenico Ferrari. On the foundation of artificial workload design. In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pages 8–14, Aug 1984.

[8] Richard Gibbons. A historical application profiler for use by parallel schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 58–77. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[9] David Lifka. The ANL/IBM SP scheduling system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[10] Rabiner L.R. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–289, February 1989.

[11] Tom M. Mitchell. *Machine Learning*. McGrow-Hill, 1997.

[12] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, work-loads, and user runtime estimates i n scheduling the IBM SP2 with back-filling. *IEEE Trans. Parallel & Distributed Syst.*, 12(6):529–543, Jun 2001.

[13] L. Paninski. Estimation of entropy and mutual information. *Neural Computation*, 15(6):1191–1253, June 2003.

[14] Dan Tsafrir. Barrier synchronization on a loaded SMP using two-phase waiting algorithms. Master's thesis, Hebrew University, 2001.