CS229: Project Report

Machine-Assisted Performance Optimization of Legion Applications

Sean Treichler
December 13, 2013

1 Background

In our research group, we're working on Legion, a new parallel programming model that targets large, distributed memory systems (e.g. large HPC clusters). It is not uncommon for such a machine to have thousands of nodes, multiple thousands of memories, and not just support, but require for performance, the use of millions of threads of execution. The many challenges that exist when programming for these systems can be lumped into three general categories:

1. writing a functionally correct program
2. achieving good performance, ideally on a variety of different machines
3. writing the program in a reasonable amount of time

In most parallel programming models, the program text has to simultaneously express both what computation is being performed and how that computation should be distributed around the machine. This entangles the first two goals, often to the detriment of the third. The Legion separates these concerns by providing a separate "mapping" step that determines how the logical tasks and data structures should be placed in the system. This provides a guarantee that changes made to the mapping in an effort to improve performance cannot introduce new functional bugs. Legion's operational semantics are also designed such that this invariant functional behavior happens to be equivalent to sequential execution of the program's tasks, further simplifying the functional debugging effort.

Unfortunately, debugging of performance issues remains a hard problem. The profiling of a sequential execution can help identify some particularly gross performance problems, but many of the performance bottlenecks in parallel systems only show up when multiple threads, cores, or nodes are contending for the same resources. It is straight-forward to gather profiling data from a large run, but it is nearly impossible for a human to look over the reams of raw data to first determine where in time and space the performance bottlenecks are, and then determine the likely cause of the bottleneck. However, both of these tasks seem well suited to assistance from machine learning algorithms.

In particular, this project attempts take raw profiling data from one or more runs of an application and *categorize* individual instances of tasks into groups with similar execution environments and runtime behavior and then *characterize* the performance of the instances in each group with respect to properties of the execution environment. Although the categorization can help identify representative instances for debugging (e.g. the "medians" of each group) and instances that don't fit any of the groups (i.e. outliers), the primary output of the algorithm should be human-readable descriptions of the groups and their performance characteristics that help the programmer understand the different operating modes for a given task and what factors have the strongest influence on those modes' performance.

The description of the algorithm below consists of the following parts:

- Section 2 gives a brief description of the raw data that is generated by the Legion runtime's profiling code.
- Section 3 list the features that are extracted from this raw data.
- The choice of model and rationale behind the choice are presented in Section 4.
- The algorithm for optimization of the model is covered in Section 5.
- Finally, the conversion of the model's parameters to human-readable output is described in Section 6.

- An evaluation of the algorithm's performance on both a synthetic example and a real-world example is provided in Section 7.
- Section 8 concludes with a brief discussion of future work.

2 Raw Data

The Legion runtime has existing code to generate a log of the tasks that are run as part of an application. An entry in the log is generated for each instance of the task that is run, and contains the following information:

inst_id	-	a unique identifier for this task instance
task_id	-	a numeric identifier ($1..T$) corresponding to which task this was an instance of
proc_id	-	the identifier ($1..P$) of the processor on which this task was run
start	-	the time at which this task started executing
finish	-	the time at which this task finished executing

In addition to the execution log, the Legion runtime provides information about the topology of the machine. For each processor, the information includes the type of processor (e.g. CPU or GPU) and which node and memory domain the processor resides in. This information will be used to estimate which other task instances may have contended for resources with a given instance.

3 Feature Definition

The profiling data is not useful in its raw form. However, many potentially useful features can be extracted from the data. (The future work discussion in Section 8 will list additional features that are desirable, but require additional profiling output.) The first value extracted for each task is the runtime:

$$y^{(i)} = finish_i - start_i$$

The remaining features are grouped into two vectors: $x_d^{(i)}$ captures the discrete (i.e. qualitative) features while $x_c^{(i)}$ captures the continuous (i.e. quantitative ones). Discrete features are extracted for each processor, memory domain, node, and processor type. In each case, a feature is 1 for a task instance if that instance ran on the particular processor/domain/node/type and 0 otherwise. Continuous features are used to extract the runtime overlap between an instance and other tasks. For example, the average number of task instances of a given type that are contending for execution resources with a given task can be calculated as:

$$proc_overlap_j^{(i)} = \frac{1}{y^{(i)}} \sum_k \mathbf{1}\{task_id^{(k)} = j \wedge proc_id^{(k)} = proc_id^{(i)}\} overlap(i,k)$$

where $overlap(i,k)$ gives the absolute amount of time that task instances i and k were both running. A corresponding $domain_overlap_j^{(i)}$ estimates the amount of contention for memory system resources by tasks running in the same memory domain. Finally, $node_overlap_j^{(i)}$ tries to estimate the contention for the network resources by instances running on the same node.

4 Model Selection

The first major decision in the model selection process was that the performance of each type of task would be modeled separately. Legion tasks come in all shapes and sizes, and without any common metrics to compare them (see Section 8), there is no reason to believe that a model for one type of task would have any predictive power for another. However, even after splitting instances by their task type (this is why *task_id* isn't listed as a feature above), observed data (see Figure 1) indicated that a simple per-task model would not suffice either.

The next most obvious choice, a more complicated model that tries to characterize all the instances of a given task type, was rejected for two reasons. First, a more complicated model introduces the risk of

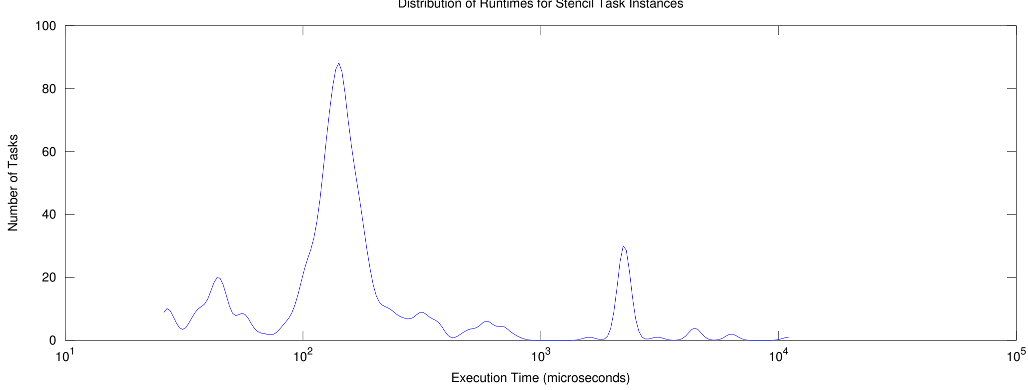

Figure 1: Distribution of Runtime for Stencil Task Instances

overfitting, especially when the number of samples has been reduced to those of a single task type. Second, and more importantly, the final goal of this algorithm is a model that can be succinctly described to the programmer, and a model in a very high dimensional space is unlikely to satisfy that requirement, even if it fits the data well.

Instead of trying to fit the data with one complicated model, we hypothesized that the data could be fit by a relatively small number of simple models. Conceptually this amounts to a "piecewise-linear" approximation of a curve rather than the use of higher-order polynomials.

Our model therefore consists of a variable number of submodels. Each submodel has a Gaussian distribution in the feature space and will use a simple linear estimator for the runtime of instances as a function of the continuous features. We introduce a latent feature $z^{(i)}$ and then describe each submodel as:

$$\begin{aligned} x_c^{(i)}, x_d^{(i)} \mid z^{(i)} = j &\sim \mathcal{N}(M_j, \Sigma_j) \\ y^{(i)} \mid x_c^{(i)}, z^{(i)} = j &\sim \mu_j + \Theta_j^T x_c^{(i)} + \mathcal{N}(0, \sigma_j) \end{aligned}$$

We further chose to constraint the variance in the feature space Σ_j to be a diagonal matrix. This was again driven by the desire to avoid overfitting and to yield models that can be easily described to the programmer.

5 Optimization

For a chosen number of submodels, the algorithm initializes the parameters of each submodel to exactly fit a randomly chosen sample. (When the desired number of clusters is not known, the algorithm uses the common approach of trying increasing numbers of clusters until a "knee" in the curve is found.) The optimization of the model uses a minor variation of the EM-algorithm to maximize the log-likelihood of the training data. We start with an equation for the log-likelihood that incorporates the latent feature $z^{(i)}$ and then select a probability distribution $Q_i(z^{(i)})$ for each sample, allowing us to write:

$$\begin{aligned} l(\mu, \Theta, \sigma, M, \Sigma) &= \log \prod_i p(x_c^{(i)}, x_d^{(i)}, y^{(i)} | \mu, \Theta, \sigma, M, \Sigma) \\ &= \sum_i \log \sum_{z^{(i)}} p(z^{(i)})\, p_c(x_c^{(i)}, x_d^{(i)}; M_{z^{(i)}}, \Sigma_{z^{(i)}})\, p_l(y^{(i)} | x_c^{(i)}; \mu_{z^{(i)}}, \Theta_{z^{(i)}}, \sigma_{z^{(i)}}) \\ &\geq \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p_c(x_c^{(i)}, x_d^{(i)}; M_{z^{(i)}}, \Sigma_{z^{(i)}})\, p_l(y^{(i)} | x_c^{(i)}; \mu_{z^{(i)}}, \Theta_{z^{(i)}}, \sigma_{z^{(i)}})}{Q_i(z^{(i)})} \\ &= \sum_{z^{(i)}} \underbrace{\left[\sum_i Q_i(z^{(i)}) \log p_c(x_c^{(i)}, x_d^{(i)}) \right]}_{\text{mixture of Gaussians}} + \sum_{z^{(i)}} \underbrace{\left[\sum_i Q_i(z^{(i)}) \log p_l(y^{(i)} | x_c^{(i)}) \right]}_{\text{weighted least-squares}} + C \end{aligned}$$

The E-step uses the normal posterior distribution of the latent feature with respect to other features and model parameters. For the M-step, the optimization of the submodel's clustering parameters uses the standard mixture of Gaussians approach (ignoring the off-diagonal terms of each Σ_j). Similarly, the optimization of the linear performance estimate can be done with a standard weighted least-squares algorithm.

In a minor deviation from the stock EM-algorithm, our implementation does not optimize both of these terms at the same time. Instead, an additional E-step is done after optimizing the linear fit and before the optimization of the clustering parameters. This "asynchronous" update seemed to result in more stable behavior of the optimization process.

6 Model Description

The optimization process is run many times with random starting points, and the best result is chosen. The algorithm then attempts to describe each of the submodels. In addition to simple statistics such as the percentage of samples covered and the mean performance of the samples in the cluster, an attempt is made to describe both the clustering and performance estimations. For clustering, a listing of the mean and variance for every single feature is not likely to be helpful. Instead, an analysis is done to see which of the features is the most *discriminative*, i.e. is the best at excluding the samples that are not in the cluster. The top 3 features are listed, along with the percentages of samples they exclude, both individually and cumulatively. Similarly, a full dump of Θ_j for each model is eschewed in favor of a list of the 3 features with the strongest correlation coefficients (i.e. r). A sample output can be seen in Listing 1.

Listing 1: Model Description for Synthetic Data

```
Model # 1:      44 samples ( 29.3%) - (u = 1031.498156, sigma_u = 36.714952)
  Clustering:
   83.3%  83.3%  continuous variable 1 == 7.974265 (var = 0.398880)
   57.4%  98.2%  discrete variable==A == 0.186448 (var = 0.151685)

  Performance Factors:
   0.994  continuous variable 2 (d/dx = 99.992169)

Model # 2:      50 samples ( 33.3%) - (u = 751.169350, sigma_u = 53.834588)
  Clustering:
   99.4%  99.4%  discrete variable==A == 0.835549 (var = 0.137407)

  Performance Factors:
   0.179  continuous variable 1 (d/dx = 9.500091)

Model # 3:      56 samples ( 37.3%) - (u = 194.074130, sigma_u = 146.683049)
  Clustering:
   84.5%  84.5%  continuous variable 1 == 4.873936 (var = 2.727712)
   53.8%  98.6%  discrete variable==A == 0.013897 (var = 0.013704)

  Performance Factors:
   0.494  continuous variable 1 (d/dx = 59.152974)
```

7 Evaluation

To evaluate the effectiveness of this algorithm, we first analyzed its behavior on a synthetic data set. By choosing a low-dimensional feature space and having known clusters is it much easier to visualize both the intended and actual behavior of the algorithm. Our synthetic data used two continuous features (plotted along the two spatial dimensions) and a single discrete feature (shown as X's vs. O's in the scatter plot). The X's form a single cluster, while the O's form two distinct clusters, each with different performance behavior. The top-left pane of Figure 2 shows the distribution of runtimes in the synthetic data while the bottom-left pane shows the intended cluster labels. The corresponding figures on the right-hand side of Figure 2 show the output of the learning algorithm. We see that it reliably learns the importance of separating O's from X's, but it is not able to do a perfect job of separating the two clusters of O's. This is because the final clustering decision is based only on the clustering parameters, and the mixture of Gaussians model cannot describe the actual boundary between the two clusters. (It's likely that if the final clustering choice were allowed to consider the goodness of the linear fit, it would get the right labels, but we chose not to allow clustering based on the $y^{(i)}$'s as a performance model that gets a good fit by simply excluding the points that don't fit isn't useful.)

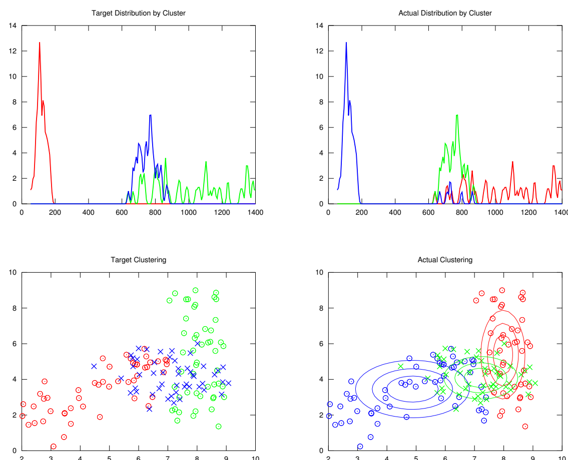

Figure 2: Results: Synthetic Data

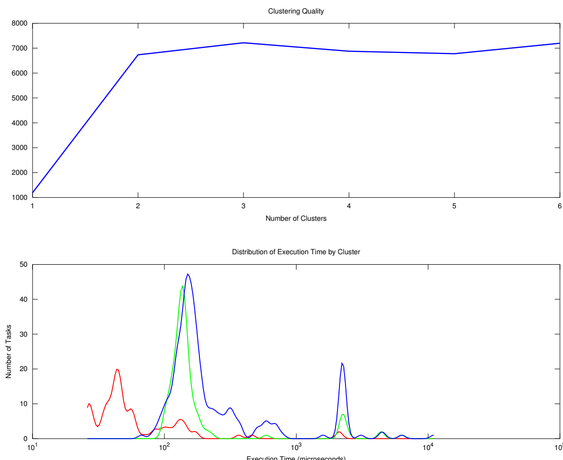

Figure 3: Results: Real Data

Figure 3 shows the algorithm's results on a real Legion application example (the same one that was shown in Figure 1). In this case, we don't have the correct labels for each individual instance, nor do we even know the right number of clusters. The top half of Figure 3 shows the best fit found for various cluster sizes, with the knee in the curve being at 2 or 3. The bottom half shows the separation in the performance histogram for the 3-cluster case. In this case, the algorithm has managed to separate the tasks that were run on GPUs (in red) from those run on CPUs but also managed to highlight two slightly different operating modes for the tasks on the CPUs, possibly related to NUMA effects in the memory system. However, the algorithm was unable to find a way to separate the spike of tasks with runtimes around 30 milliseconds - more analysis is needed of the actual application to determine what features (if any) could be used to distinguish those cases.

8 Future Work

We are optimistic that this algorithm will assist us in our performance optimization of other Legion applications, but there are a number of additional improvements that are likely needed before we get there. First, there are a number of additional features we'd like to extract from the execution of an application. The most obvious is counts of the number of operations performed by each task (e.g. instructions, memory references, cache misses). These will help in two ways. First, they may allow the creation of models that cover more than one kind of task at a time. And second, it's likely that the actual runtime of a task is correlated with some of these features (although possibly a different set for different operating modes).

Second, we want to explore whether it's better to estimate runtime in linear space or logarithmic space. A logarithmic space is probably better able to cover the wide range of runtimes we observe, and many interactions are likely to have multiplicative effects on the runtime, which are easier to capture in a logarithmic space.

Third, the model descriptions for our real data sets aren't as useful as we'd like. These data sets have many more features than the synthetic model shown above, and it appears that many of these features are strongly correlated with each other. As a result, attempts to select the most discriminative or correlated features often choose one of these correlated "dependent" features rather than the "real" feature. As a simple improvement, we'd like to weight the features by "intuitiveness" so that we favor descriptions based on those over descriptions that use less comprehensible features.

Finally, as with any clustering algorithm that uses a small number of clusters, the results of the optimization process are highly dependent on the initial seeds. It's worth spending some time to see if an agglomerative clustering algorithm is better able to fit submodels to smaller clusters (e.g. the 30ms spike in Figure 3) without resulting in so many clusters that the programmer doesn't have time to look at them all.