Learning the optimal degree of concurrency for performance-critical Python processes

Nicholas Shieh
ns665@cornell.edu

Peter Li
pl488@cornell.edu

Abstract

Python threading is (unfortunately) an illusion. At runtime, a Global Interpreter Lock restricts execution to one native thread per process at any given time. Thus, the illusion of multithreading is created by a lot of fast context switching which directly results in trade-offs between degree of concurrency and runtime performance. Too little concurrency doesnt make full utilization of computational resources while too much concurrency introduces memory bottlenecks that make context switching not worth the expenditure on handling overhead.

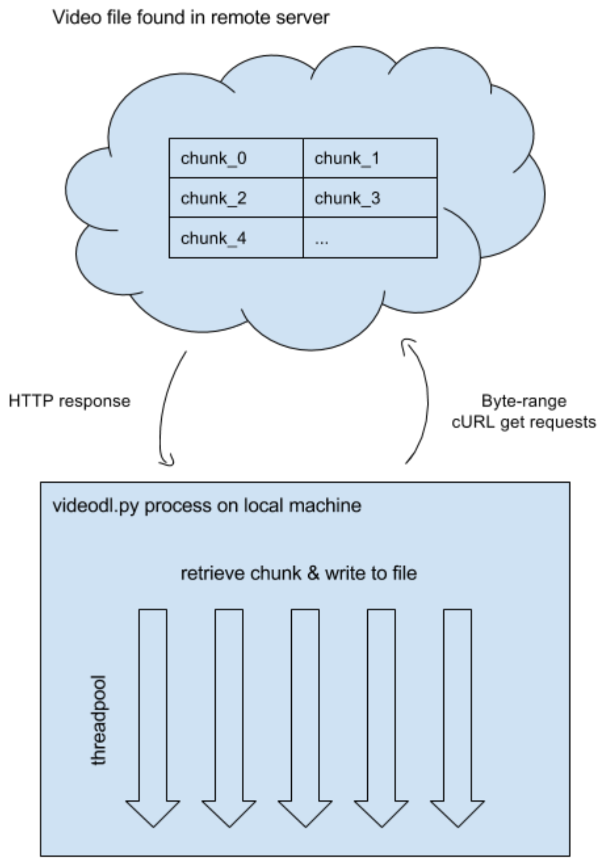
We aim to figure out where the sweet spot of concurrency is. To tackle this problem, which is applicable to a myriad of Python programs, we will set up an experiment that uses machine learning techniques to seek out a degree of concurrency which minimizes runtime and generalizes well.

Our ultimate goal is to have a tangible conclusion or prototype that such a learning tool can be implemented efficiently and that its prediction remains consistent to a wide variety of scenarios that users encounter in everyday computing. The code for our project can be found here https://github.com/peteli3/ai-prac-project

1 Introduction

1.1 Baseline

In order to quantify the relationship between degree of concurrency and performance, we will select a "parallelizable" mostly cpu-bound process that has a notable serial runtime to benchmark from: a performance-focused Python procedure that uses cURL requests to download a video file from a remote URL and run checksum comparisons to verify integrity of delivery. We will refer to this procedure as `videodl.py`. For clarity, we will provide an illustration of the networking component:


Video file found in remote server
chunk_0
chunk_1
chunk_2
chunk_3
chunk_4
...
HTTP response
Byte-range cURL get requests
videodl.py process on local machine
retrieve chunk & write to file
threadpool

The base functionality of `videodl.py` is broken down into 3 parts:

1) Make a cURL request to obtain the HTTP header and relevant meta data; needed to decide if the target server supports byte-range requests.

2) If byte-range requests supported, spawn a thread pool of predetermined size to make all sequential byte-range cURL requests and write their content body into temp files.

3) Consolidate all chunks by writing them sequentially into a final result MP4 video file and then compute its MD5 and CRC-32C hashes with those obtained in step 1.

1.2 Motivation

From a profiling standpoint of the procedure, it is clear that the performance bottle-necking section is the sequential cURL requests in section 2. Since we have fine-grain control over performance here by controlling the degrees of concurrency used, we will create an experiment on how to select the degrees such that runtime is minimized for the general Python procedure execution.

"Parallelism" can be underused or overused. Under using concurrency makes poor use of available thread resources since not enough are allocated to handle tasks given. On the other hand, over using creates a scaled amount of memory overhead due to context switching, enough so that the benefits are outweighed. Both weaknesses become more pronounced with respect to workload. Thus, for many performance-critical purposes, it may be necessary to figure out the optimal degree of concurrency to use.

1.3 Approach

In this experiment, we will perform our selected task on a remote URL that contains a 1GB video and time its runs over multiple different degrees of concurrency to get an idea of the relationship between runtime and concurrency. We have selected a 1GB video since it is a very practical situation and its execution is long enough such that we have enough timing information to be comfortable with isolating the changes in runtime (dependent variable) with respect to concurrency (independent variable).

We will then approximate the relationship by collecting timing data; running `videodl.py` on various degrees of concurrency and fitting a polynomial curve to the results. We originally attempted to fit using a penalized spline regression, mimicking polynomial degree with spline degree, but found that the amount of extra computation needed for the coefficients was not enough to justify the difference in judgment over a poly-fit curve. This small time difference can potentially be overlooked in our experiment but not in future generalizations of this work (to be discussed in Desiderata). As such, we decided to continue the experiment using a poly-fitting as our main approach, while investigating other methods such as linear regression and spline interpolation.

A negative side of poly-fitting is the introduction of enough uncertainty to invoke observations of bias vs. variance. For example, we expect low degree polynomials to underfit on the data and have low predictive power (high bias), while high degree polynomials suffer from overfitting and too much specificity (high variance). Thus a new problem arises: how to select a degree of poly-fitting such that we get the most generalized model possible. This will be accomplished by running a short validation phase where the poly-fit of various degrees are computed on the training set it is fit to and then compared amongst new timing data via the residual sum of squares.

With a good enough approximation, we can then apply various optimization algorithms to find the degree of concurrency (the independent variable) which minimizes runtime (a loss function, and in this case also the dependent variable) on our fitted curve.

It is noteworthy to mention that our approach excludes supervised learning algorithms and instead focuses on optimization techniques due the nature of the task and information available. It is not reasonable to assume that there will be enough useful labelled data to extrapolate from for performance benchmarking since many external variables play a role in observed runtimes: involvement of networking and all associated variables like network statistics, number of flops per execution, number of cpu cores available in system, serial tuning of the code, and so on.

2 Experiment

2.1 Logistics & Thoughts

Collection of the data was done on an Early 2014 Macbook Air running MacOS High Sierra with 1.4 GHz Intel Core i5 CPU and 8 GB 1600 MHz DDR3 RAM. The network statistics are 20 ping with 60/60 Mbps download/upload speed. We run `videodl.py` on a 1 GB video in 5 MB chunks.

We expect to see a sharp dropoff in runtime going from a single thread to multiple threads. Past a certain amount of threads x, the time it takes to context switch between the threads and for threads to yield may outweigh the benefit of concurrency. After x, we will expect to see an upward trend in runtime.

2.2 Data Collection

To get a representative dataset, we change our parameter θ (num threads) to values between $[1,100]$, and time how long it takes to download the video with θ. We have 5 runs of data collection that we average over. In each run, we obtain 32 data points where $\theta \in \{1,2,10,12,14,16,18,20,22,24,26,28, 30,33,36,39,42,45,48,51,54,57,60,64,68,72,76, 80,85,90,95,100\}$. All data points are averaged over 5 runs to rule out some external variances. For $\theta < 10$, we decided to only sample 1 and 2 in the interest of time, and we expected those two points to capture the initial sharp decline in runtime. The plotted data:

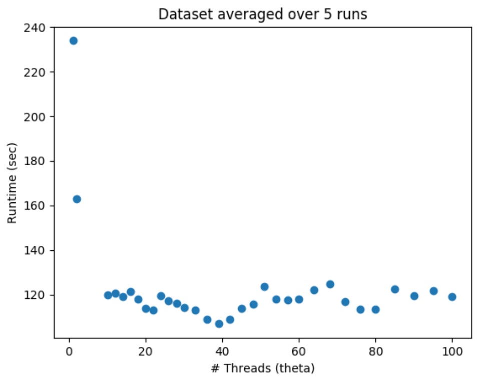


Figure 2.2 - Initial timing data averaged over 5 runs

In order to maintain as much constant as possible during data collection, we also need to ensure that between `videodl.py` runs, no performance boosts are observed on a round $i+1$ that result from changes made to computing environment by a previous round i. This means that between runs, we will need to ensure that nothing cached can be used to result in a speedup for a subsequent round. We are able to accomplish this by spawning new processes to handle data collection rounds, such that no two will share any memory space.

2.3 Analysis of Dataset

As expected, there is initially a sharp dropoff between $\theta = 1$ and $\theta = 10$. The pattern is similar to an exponential decay, where past a certain θ, we experience diminishing returns in runtime. It is interesting to note that we observe several 'kinks' in the dataset, around the 40 and 80 mark.

For this dataset, we try to approximately fit a polynomial curve with d degrees by minimizing least squared error. Since a polynomial of degree $d+1$ fits at least as well as that of degree d, fitting a polynomial of degree d as d grows larger will eventually result in a curve with zero error. Here, choosing d leads to a bias-variance tradeoff. Choosing a $d = len(train)$ is not generalizable as a large d overfits on the training dataset, essentially connecting the dots. Though overfitting lowers the training error, we risk capturing the training data's noise or its specific relationship, and thus fail to adequately capture the underlying structure of the data's distribution. Likewise, choosing a small d may underfit on our data, failing to capture the underlying relationship and lead to wider fit curves with lower gradient magnitudes, to which a first-order optimization method might easily bias towards a substantially higher or substantially lower prediction value than expected.

Below is an example of a polynomial fit to the data obtained during our unsupervised training process. The equation of the curve is given by $f(x) = 1.885 \times 10^{-11}x^8 - 8.332 \times 10^{-9}x^7 + 1.522 \times 10^{-6}x^6 - 0.0001484x^5 + 0.008319x^4 - 0.2689x^3 + 4.779x^2 - 41.79x + 253.1$. Provided is an overlay of the best fit curve onto the data:

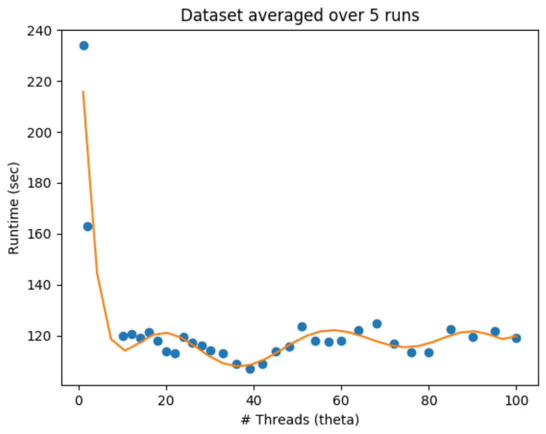


Figure 2.3a - SSE = 1250.50 with d = 8

We wanted to capture the general trend of the data, which includes the sharp decrease in runtime between 1 thread and 10 threads, and also any local extrema such as $\theta = 40$ and 80. Due to the proof-of-concept nature of our project, we do not have any labelled data to validate our model on, so our best guess for choosing d is from manual examination of the plots. What we can do instead is take new runs of the data, and validate our original curve to see how well it generalizes, and then tune d accordingly. The subsequent plots in figures 2.3 b-f below show the original poly-fitted curve of $d = 8$ overlaid on 5 new runs of datasets with their sum of squared errors.

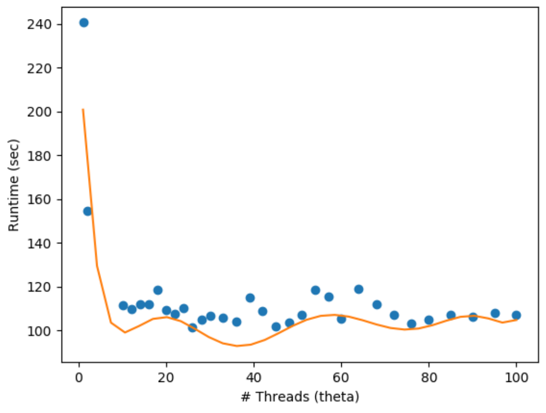


Figure 2.3b - SSE = 4184.33

Note: across multiple runs of timing we have observed visual and consistent periodic fluctuations in runtime past $\theta = 20$, indicating that some underlying factor is affecting performance that we've been unable to isolate and identify.

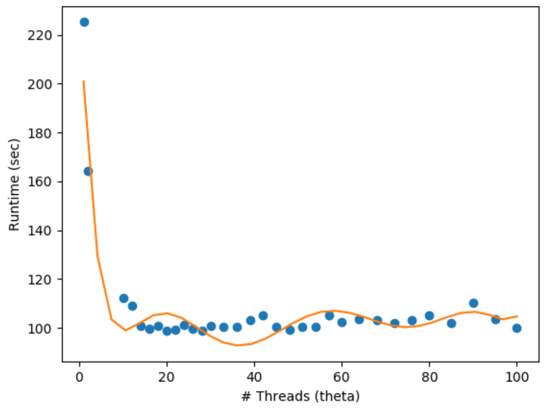


Figure 2.3c - SSE = 1517.17

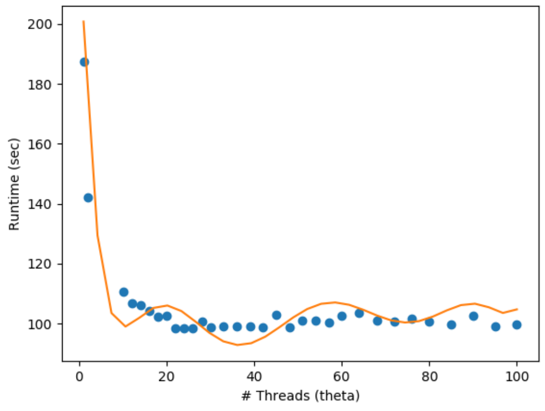


Figure 2.3d - SSE = 1690.15

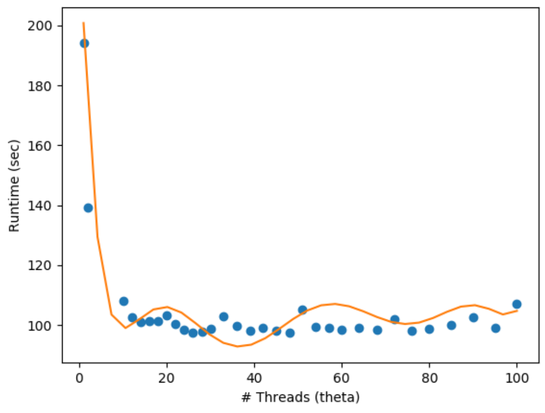


Figure 2.3e - SSE = 1788.51

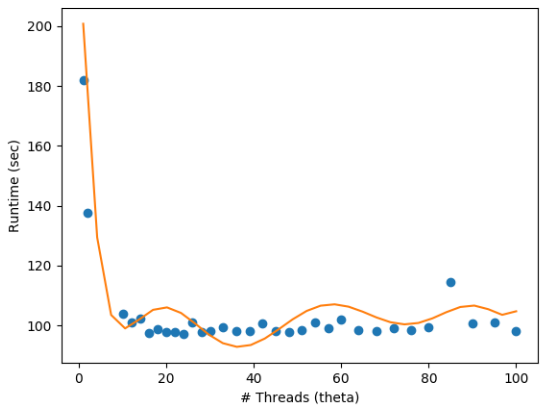


Figure 2.3f - SSE = 2322.53

This procedure is essentially an adapted version of "validation" that aims to show us how well our fit generalizes to unseen trials of data and outlines what the testing error of our various poly-fit will look like.

We observe that the curve is slightly offset for these 5 new runs, as seen by their sum of squared errors. Each graph yields large sum of squared errors since lots of data points are used and any amount of residual is punished harshly by taking the square. There are also very clear parts of the fit that do not generalize well. For example, we notice that among all 5 runs (and even considering prior timing data collected), the true data shows upward bumps where runtime suddenly becomes almost 1.5 to 2 times higher and then comes back down. The positions of these may be found at different x-values and can be explained by fluctuating network slowness: for example, two packets leaving a source can differ in travel time to reach the destination due to possible faults in the path along the way, creating a need for dynamic re-routing and therefore a possibly longer path for one packet.

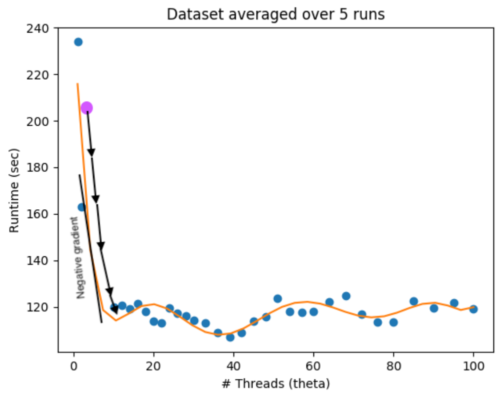
Networking is amongst many variables that can explain the fluctuation in data we collected. In the real world and in unsupervised ML, these factors arise often and obtaining good quality datasets is a separate problem on its own. Some identifiable external factors that may affect the quality of our data in an un-quantifiable way include network reliability, server-side performance, fluctuations in broadband speed, and CPU load usage from running other programs at the same time. Under ideal conditions, we would have a constant, reliable broadband connection and a machine dedicated to collecting data over thousands of runs. Furthermore, the video was downloaded over Wi-Fi, which has a whole host of performance variability based on physical environment and configuration. Therefore, we can only reasonably deduce conclusions via approximations of true behavior.

2.4 Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. Given a convex, continuous, and differentiable loss function $\ell(w)$, we find its minimum through the following algorithm:

```
Initialize $\vec{w}_0$
Repeat until convergence:
$\vec{w}^{t+1} = \vec{w}^t + \vec{s}$
If $|\vec{w}^{t+1} - \vec{w}^t\|_2 < \varepsilon$, converged
```

Here, $\vec{s} = -\alpha g(\vec{w})$ for some small learning rate $\alpha > 0$ and $g(\vec{w})$ is the gradient (first order). In the context of our experiment, we obtain a polynomial curve that serves as our loss function, where the x-axis represents the number of threads and the y-axis represents the time it takes to download the 1 GB video in seconds. Running gradient descent converges to a local minimum, giving us the approximate optimal parameter (num threads) for this specific task. Specifically, we have a step size of $\alpha = 0.01$ and $\varepsilon = 0.00001$. The following plot demonstrates gradient descent, taking steps until converging at a local minimum of $\theta = 10$.

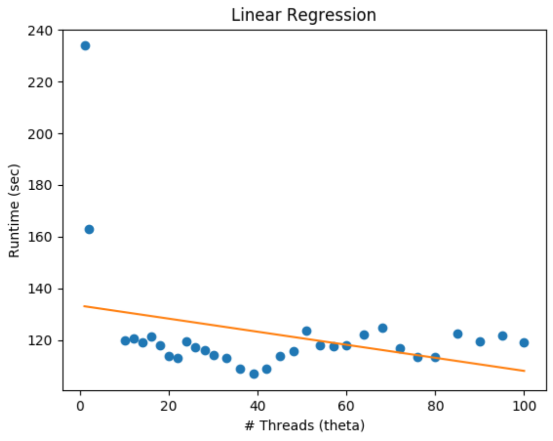


2.4.1 Linear Regression

To experiment with gradient descent and its uses in AI/ML, we wanted to try out linear regression on our dataset. The linear regression problem is defined as such: given a set of data points, find the 'best' line of fit. The model is defined by $y = \beta_0 + \beta_1 x$. We define 'best' by the model that minimizes the error or loss function: $\ell(\beta) = \sum(y - \hat{y})^2$, where y is the prediction from the model with parameters β and $\hat{y}$ is the actual value.

One can also directly solve the linear regression problem by computing the matrix $\beta = (X'X)^{-1}X'Y$. However, this is often computationally slow as we need to compute the inverse of the matrix $X'X$; gradient descent instead can find the optimal parameters efficiently, especially in higher dimensional data. We now have our loss function that is differentiable, so we can run gradient descent to find the optimal β. We randomly initialize the weights and find that gradient search converges on $\beta = [133.3, -0.2528]$.

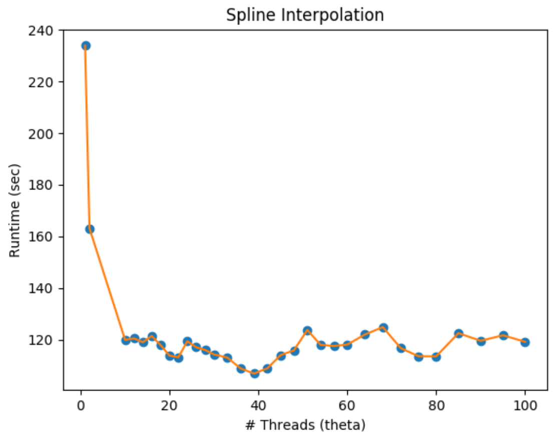
The following plot shows the best fit line for linear regression. Clearly with a SSE of 13691.37, the dataset is not linear in nature. Linear regression is not ideal for this problem, which is what led us to experiment with polynomial fitting.



2.5 Spline Interpolation

Interpolation also works well to approximate a function given a set of points. Specifically, we test out cubic spline interpolation to see how well the curve compares to polynomial fitting. In spline interpolation, the interpolant is a piecewise polynomial (spline) that passes through each interval of the dataset. Since we have $n = 32$ points, we have $n - 1 = 31$ intervals between them, in which there is a separate cubic polynomial for each interval, each with its own coefficients:
$S(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$
for $x \in [x_i, x_{i+1}]$.

This method results in a SSE of 3.84e-27. We see that the curve follows the points closely in every interval, basically a perfect fitted polynomial between consecutive points. The results of interpolation seem promising when comparing the error, but as mentioned before, having a close-fit curve can lead to overfitting, which is why we still chose the polynomial fit over spline.



2.6 Newton's Method

Another minimization technique we experimented with is Newton's method. The prerequisite is that the loss function $\ell(w)$ is twice-differentiable. Essentially, each iteration we approximate $\ell(w)$ by a quadratic function/parabola around x_n, and then take a step towards the minimum of the approximated quadratic function. The Hessian matrix is a square matrix of second-order partial derivatives of a function given by $\mathbf{H}_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$.
This method often converges to a local optimum in several iterations, significantly faster than regular gradient descent, but at the cost of computing the Hessian matrix. If the step size is too great, the algorithm may diverge. This can be counter-

$$H(\mathbf{w}) = \begin{pmatrix} \frac{\partial^2 \ell}{\partial w_1^2} & \frac{\partial^2 \ell}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 \ell}{\partial w_1 \partial w_n} \\ \vdots & \cdots & \cdots & \vdots \\ \frac{\partial^2 \ell}{\partial w_n \partial w_1} & \cdots & \cdots & \frac{\partial^2 \ell}{\partial w_n^2} \end{pmatrix}$$

acted by first running several iterations of gradient descent, and then finishing with the Hessian calculation. Running Newton's method on our loss function also results in a local minimum at $\theta = 10$.

2.7 Analysis of Results

The global minimum is actually at $\theta = 39$ with a runtime of 106.79 seconds. Gradient descent and Newton's method starting at 1 only converge to local minima, which is $\theta = 10$ in our case with a runtime of 119.86 seconds. To counteract this, we can randomly initialize our starting points for gradient search, and then take the θ that gives the minimum runtime. Thus, random initialization of weights combined with gradient descent gives us a result of $\theta = 39$, the optimal degree of concurrency for `videodl.py`.

Other processes may have different optimal degrees of concurrency. As such, not much extrapolation can be done with these exact results due to the difficult nature of obtaining unlabelled datasets, let alone labelled data; however, our proof-of-concept details out the initial framework and core concepts for a generalized model. Our methodology can also easily be replicated for numerous types of Python processes. We discuss this further in the Desiderata section.

3 Conclusions

Our findings all conclude that there is a clear relationship between runtime and degree of concurrency. We can leverage this relationship to find optimal parameters that make the best use of computing resources.

As of right now, any meaningful procedure that can help us accomplish this goal is very significant itself in terms of runtime and may be subject to evaluation of worth in a case-by-case basis. For example, it may not be worthwhile in a situation where a 1GB recreational video file takes 10 seconds longer to download on a sub-optimal number of threads. But on a numerical code or scientific simulation that normally takes 5 days to compute, a 5% performance bonus saved by better use of caching or context switching created by a learning procedure that takes 10 minutes to compute may result in saving 6 hours worth of computing time, which is substantial.

We will discuss in the following section what possible extensions of our work might look like. Since we now know that it might not always be worth it to take the time to figure out where the concurrency sweet spot is, extensions can be built that focus on one of two possible remedies: scaling back the detail of the study or on preprocessing data; both have similar risk of jeopardizing generalization, but may not yield differences strong enough such that the results will be wildly different. Thus, we conclude that any practical application of our work can at best provide an initial framework or benchmark for future projects.

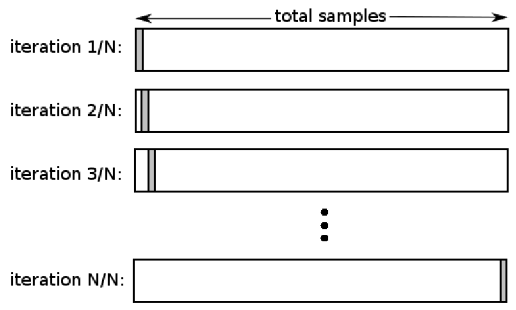
It was interesting to finally have the opportunity perform a study like this one where we attempt to analyze the relationship between runtime and degrees of concurrency. Many current methods that people employ to determine the answer to this problem center around the assumption that it cannot possibly make too big of a difference: just pick some number in a reasonable ballpark and call it a day. Despite the small differences in runtime, we can see now, after hundreds of polynomial fits applied, that there definitely is a complex underlying relationship who's magnitude of extrema scale with the size of task.

3.1 Further Generalization

The work done in this project was an experiment hinting towards a software development process that can tune parallelism to a general case. We hope that our work can provide helpful insights for others working on performance-intensive tasks, since the performance benefits of finding optimal parallelism scale up with size of workload.

In practical settings, the application of our work is determined by whether or not it is worth taking the time to dynamically compute the optimal degree of parallelism. For large workload settings where performance may be affected by a number of variables, performance is everything and any amount of scalable tuning can go a long way in leading to lower runtimes, so the fixed amount of time needed to determine optimal concurrency is worthwhile considering. Examples include large matrix computations, complex information network analysis, or even training convolutional neural networks.

A completed learning algorithm for the concurrency problem would break up workload into three sections: training, validation, and testing. Given the experimental nature of the problem, there is no labelled datasets for one to train and test on. Instead, collection of data akin to our methodology is needed. Given the large computational effort and time expenditure to collect that dataset, evaluation should be done using "leave one out cross validation". This is shown in the following diagram:



Essentially, in a generalized setting, one would obtain n datasets from n similar processes or programs that are parallelizable. One program is left out, and a predictor is trained on the remaining $n-1$ programs, and predict the n^{th} program with the trained model. This is repeated n times for each program and in turn, we always evaluate the model on an unseen program. This method of evaluation is useful for small datasets and thus should be the preferred method of evaluation for our generalized concurrency problem.

The purpose of training is to gather data to formulate a relationship between runtime and degree of concurrency. Like it does in our experiment, various degree polynomials will be fit to the data and evaluated in the subsequent section for generalization error. Among the results, the fit computed with lowest residual sum of squares is selected as the loss function of performance plotted against degrees of concurrency.

We then use a derivative approximation method to get as close to the absolute minimum as possible: using either random-seeded first order approximation with adaptive step size or using second order approximation supposing we have been enough historical information to be able to compute a Hessian matrix during preprocessing. The independent variable that minimizes runtime is then selected out and applied to the remainder of the workload: the testing set.

At this point, we've given full consideration to the information available and have devised a procedure that makes an educated guess at what the optimal degree of concurrency to use is.

3.2 Desiderata

3.2.1 Built-in Defaulting

A possible implementation of a learning program that finds optimal concurrency parameters can be built in to be shipped with a completed compiler (or interpreter, in the case that we are considering Python processes).

In which case, a similar experiment can be done upon preparing the compiler or interpreter for deployment. Using the method highlighted in this report, the work needed for computing optimal concurrency will be done only once but be detailed and optimized for compiler/interpreter specific execution. Unfortunately, it will be influenced by external factors like network or various user configurations that are out of control of the manufacturers.

However since the optimization made during deployment is compiler specific, current methodology will likely find an optimal parameter that differs with low error from the true optimal of networking tasks and negligible error from the true optimal of localized tasks. Thus for a pre-deployment defaulting approach, it is further reasonable to collect training, validation, and testing datasets across various hardware configurations such that more data is available for future consideration. With more historical data, dynamically computed optimal concurrency will also be subject to more device-specific details, and result in selecting polynomial fitting curves which fit better to routine performance needs.

3.2.2 Data Collection

Obtaining data for our proof-of-concept was difficult since there wasn't a library of available, ready-to-download hosted MP4 videos of multiple standard sizes. It would be interesting to see how well our polynomial curve generalizes to datasets produced by larger or smaller video sizes. Furthermore, there are no pre-existing labelled datasets for us to use supervised ML algorithms on, due to the nature of the project. Translating code into generalizable feature vectors to be used for predicting processes' runtimes is an extremely hard problem on its own. The labelled runtimes would vary over the user's system and network conditions; handcrafting the dataset is nearly impossible.

3.2.3 Algorithms

A further extension could involve using Adagrad to speed up the gradient descent by “learning” the best gradient descent step size to use. If the feature vector for the concurrency problem is large and sparse, then Adagrad becomes suitable as it learns the step size for each feature adaptively. It is mainly an attempt to tackle efficiency during the step of evaluating the selected fit curve to find the optimal parameters. But since this procedure is only run once after best generalizing poly-fit is chosen and we only have one feature, we are not likely to see a huge performance change and have therefore decided to leave this as a possible future extension.

In more complex problems or if our feature vector was expanded to model program features (static, load/store, branch instructions) and we had a large dataset, there may be many local minima that exist on the error manifold. Regular gradient descent may converge quickly to a bad local minimum, which is where Stochastic Gradient Descent would be applicable. Instead of updating the weights over the entire training dataset, we do it for a mini-batch of points. This decreases the chances of the gradient search being stuck in bad local minimum, as the noisier gradient computed from the mini-batch tends to pull the model out 'bad' local regions and into potentially better regions with better optima. Furthermore, SGD is computationally faster than regular gradient descent as we no longer pass over the entire training set each iteration, allowing for faster convergence of our parameters.

4 Technical Background

In this section, we will discuss the intuition that went behind taking our described approach and elaborate on which areas of our curriculum we drew ideas from.

4.1 Systems and Concurrency

Throughout this text, we carefully select our usage of the words parallelism and concurrency, since they are not the same, and we did not intend for any likening of the two terms. The difference between the two is a crux point of this experiment and the main reason why we have results to report at all.

True parallelism refers to a paradigm in which separate **processes** are being run concurrently. Each process has its own memory space (without sharing between, unless specified ahead of time, in which case synchronization primitives are necessary), and lots of overhead is needed to achieve this in exchange for a linear scaling of speedup with regard to degree of parallelism used. For example, if an embarassingly parallel task is run in serial, it might take 100s, while we might see 50s for 2 threads, 33s for 3 threads, and so on. Timing results may observe insignificant deviations in runtime expectation due to amount of fixed overhead necessary to set up separate processes.

Given the interpreted nature of Python (non-Cython code specifically), true parallelism can never be achieved since a synchronization primitive known as the Global Interpreter Lock is put in place to serialize CPU access to CPython (most commonly used implementation of Python) bytecode, due to non thread-safe memory management in the implementation of CPython. Subsequent implementations of Python suffer from legacy code and integration that is built upon the assumption of the presence of a GIL. Therefore, this issue with finding optimal concurrency degree will exist for as long as the GIL does, prompting possible future work that generalizes this experiment.

4.2 Machine Learning

As mentioned above from related works, we isolated the problem down into the core issue: given what we can collect using a Python interpreter, how do we predict the degree of concurrency that will minimize runtime?

We then considered appropriate tools (learned from AI vector courses such as CS 4700 and 4780) and ended up taking a learning and optimization approach where we estimate the relationship and then find parameters that minimize loss (or in this case, runtime). The simplest optimization tactic was to use first and second order approximation to find local minima from a starting point. In order to do this, we needed a continuous relationship that is differentiable everywhere, thus creating the need to select a polynomial that fits the training data.

From poly-fitting, we also introduced a new controllable variable: the degree of polynomial used in fitting. No labelled data is given so the only way of evaluating the fitted polynomial is through an approximation on its generalization error. We thus decided to use a Sum of Squared (SSE) error analysis to quantify generalization error. We then performed a simplified alternate version of validation that mimics unseen datasets and evaluates the fit of our prospective polynomial curve with its degree as a parameter. We then run the gradient descent to obtain the final estimation of optimal concurrency degree.

This project has been a meaningful compilation of topics covered in both our CS-heavy systems courses and theory/math-heavy machine learning courses, and we look forward to working on similar topics in the future.

5 Related Work

This project was partially inspired by the desire to tune concurrent programs to the finest-granularity possible. One such source of inspiration was found in a study done at the University of Edinburgh [2], where graduate students created a predictor that selects thread scheduling policies that offer optimal scalability of resource allocation by machine. Their analysis dives deep into the effects of assembly-level and memory-level behavior, such as branch missing and cache missing, on overall performance of programs.

While determining the relationship between speedup and degree of concurrency, a predictive analysis was used to guess at the relationship between concurrency and speedup. Our project is more or less an extension on this implication since our primary goal is to determine if degree of concurrency optimization can be done in a reasonable amount of time given any amount of computing resources available.

When considered at a high level, this problem can be likened to other automated read/write tasks such as data transfers. Another major source of inspiration for our experiment was by a study published in the IEEE journal for Parallel and Distributed Systems [1]. Yildrim et. al. hypothesized that there was a tradeoff between parallelism used in wide area data transfers and performance improvements. Similar to how we hypothesize that high concurrency creates a lot of unnecessary work for the interpreter to do due to context switching, Yildrim et. al. hypothesized that for wide area data transfers, large degrees of concurrency lead to congestion problems along the network that eventually serve to hinder speedup.

References

[1] Esma Yildrim, Dengpan Yin, T. K. Prediction of optimal parallelism level in wide area data transfers. *IEEE Transactions on Parallel and Distributed Systems 22* (2011).

[2] Zheng Wang, M. F. O. Mapping parallelism to multi-cores: A machine learning based approach. *ACM PPoPP* (2009).