実行時自動チューニングのための逐次実験計画の一手法

○須田礼仁 [1]

[1] 東京大学情報理工学系研究科コンピュータ科学専攻

(e-mail) [1] reiji@is.s.u-tokyo.ac.jp

1 はじめに

数値計算に限ることではないが，同一の計算内容であっても，プログラムの書き方によって計算機による実行時間が異なることはよく知られている．このような現象は，深い CPU パイプライン，多段キャッシュなどのメモリ階層，SIMD 風演算器・マルチコア・SMP・クラスタ・グリッドなどの並列化階層といった計算機アーキテクチャの複雑化（ある意味での進歩）により，より強く現れる傾向にある．そのため，計算を行うマシンの特性にあったプログラムを書けば非常に高い性能が得られるが，そうでないと極めて低い性能に留まってしまうという現象が，今後はさらに広い分野でさらに顕著な形で現れてくるものと思われる．現在でも特段の考慮をしていないソフトウェアの性能はほとんどの場合ピークの 1/10 未満と思われるが，これがさらに 1/100 （すなわち，書き方次第で 2 桁速さが違う）となれば，さすがに多くのユーザが改善を望むであろう．

しかし，日進月歩の計算機アーキテクチャにあわせてプログラムを書き改めてゆくことは，とりわけ大規模なプログラムでは，非現実的である．しかも，特定のアーキテクチャで高い性能を出すプログラムほど，他のアーキテクチャに適応するためには大掛かりな修正が必要となる．また，改変の際にバグが混入する危険性もある．そこで，単一（できるだけ少数）のプログラムで多くのアーキテクチャに適応できるような仕組みが望まれる．

そのような可能性のひとつが**自動チューニング** [1] である．片桐による ABCLibScript では，指示行を用いて複数の実装をひとつのプログラムに記述しておくと，実際のマシンで試行をして最適な実装を自動的に選択してくれる．必ずしもぎりぎりにチューニングされたプログラムが常に生成されるわけではないが，コンパイラの最適化と相補的に働くことにより，一定のレベルの性能を常に得られるようなソフトウェアが，比較的容易に構築できるものと期待される．

1.1 自動チューニングのための実験計画

コンパイラの最適化においては，一般にコンパイラ内部に構築されたマシンモデルに基づいてコードの最適化がなされるのに対し，自動チューニングでは実機で**試行**することにより最適な実装を選択する．試行のタイミングは，インストール時・実行直前・実行中などいろいろあるが，いずれにせよ試行にはコストがかかる．従って，できるだけ少ないコストの試行により，できるだけよい実装を選択することが望まれる．これは明らかに**実験計画**の問題である．

少ないコストでよい実装を選択するといっても，試行のタイミングにより問題設定が異なってくる．本稿で考察するのは，実行時自動チューニングである．想定しているのは次のような問題である．

プログラムがある一定の計算（たとえば LU 分解のような）を繰り返し行うとする．そして，その計算を実現する実装が複数あるとする．実装のバリエーションは離散的な場合も，連続値パラメタで制御されている場合もありうるが，今回は最も簡単な場合として，2 種類の実装から選択する場合について考える．また，目的関数は**所要時間**とする．

事前にはいずれの実装が速いかは未知とする．それでも，プログラムが反復して計算を実行するうちの何回かを試行に用いることにより，どちらの実装が速いか調べることができる（**実行時自動チューニング**）．もし実行時間にぶれがなければ，最初の 2 回の呼び出しで 1 回ずつ試行し，速かった方を残りすべての反復で使用すればよい．しかし，実行時間にぶれがある場合，1 回ずつの試行だけでは判断を誤る可能性がある．判断を誤る確率を低くするためには，遅い実装も繰り返し呼び出す必要があるが，遅い実装を多く呼ぶと実行時間が長くなってしまう．そこで，判断の誤りと実行時間の増大という 2 つのファクターを最適にバランスする実験回数は何回なのか，というのが今回の問題である．

2　既存の知見

本節では既存知見について述べる．まず問題設定からすぐに分かることを確認しておく．

最適な実験回数は実行時間の差や「ぶれ」の大きさに依存する．もし所要時間のぶれに対して所要時間の差が十分に大きければ，やはり 1 回ずつの試行で十分である．しかし，ぶれに対して差が小さい場合には，相当な回数試行しなければ優劣を判定できない．従来の実験計画では，ぶれと差をある程度推定しておいて必要な反復回数を見積もっておくとしている．しかし我々の問題設定では，実験結果からぶれと差を評価して，それによって反復回数を決めることができる．但し，ぶれと差の評価値にも誤差が含まれていることを考慮しなければならない．また，目的は優劣をつけることではなく，所要時間が最小に近ければよいということにも注意が必要である．

チューニングの対象となっている計算が何回反復されるか，というのが非常に重要である．もし無限に繰り返されると仮定すれば，十分なコストをかけて真に最適な実装を明確にしてしまえばよい．逆に反復回数が非常に少ない場合はかけられる試行のコストは限られる．極端な場合，反復回数が 3 回であれば，最初の 2 回で両方の実装を 1 度ずつ試して，速かったほうを 3 回目で使用するのが自明に最適である．反復回数が 4 回の場合も，3 回目までは同じで，4 回目は平均実行時間が短い方を選択するしかない（一般に，最終反復ではそれまでの平均実行時間で速かった方を選択するのが最適である）．しかし反復回数が 5 回の場合は，4 回目の実行として，遅い方を選択して情報を集めるという選択肢と，速い方を選択して所要時間の短縮を図るという 2 つの選択肢がある．

2.1　Bandit problem

古典的な実験計画では，実験計画・試行・分析の 3 つのフェーズが分離している．しかし上述の実行時自動チューニングでは，実験計画・試行・分析という作業が毎回繰り返されることになる．このような統計分野を sequential analysis と言うようである．

今回我々が扱う問題は **bandit problem**[2] と呼ばれている．きちんと定義すると次のようになる．本稿の問題設定では実装は 2 種類としているが，以下では一般に k 種類とする．

k 個の確率過程 X_{im} $(1 \leq i \leq k, m = 1, 2, \cdots)$ があるとする．第 m 回目の試行では，このうちひとつを選んで観測する．すなわち τ_m 番目の確率過程を選んだとすると，m 番目の試行は $Z_m = X_{\tau_m m}$ となる．確率過程の選択 τ_m は **strategy** と呼ばれ，それ以前の観測値に依存して $\tau_m(z_1, z_2, \cdots, z_{m-1})$ のように決められる（小文字は確率過程の実現値を表す）．

Discount sequence $A = (a_m)_{m=1}^{\infty}$ が与えられており，$\sum_{m=1}^{\infty} a_m < \infty$ を満たすものとす

る．目的関数は **payoff** $\sum a_m Z_m$ の最大化である．反復回数が n 回であれば，

$$a_m = \begin{cases} 1 & m \leq n \\ 0 & m > n \end{cases}$$

のように定義すればよい．このように，ある n があって $m > n$ に対して $a_m = 0$ となるとき，**finite horizon** という．

2.2 ベイズ統計による bandit problem の解法

ベイズ統計を用いれば，この最適化問題の解は次のような動的計画法により与えられる．

事前情報が G, discount sequence が A のときの最適戦略による payoff の期待値を **value** と呼び $V(G, A)$ で表す．すると，finite horizon であれば

$$V(G, A) = \max_{1 \leq i \leq k} E\left(a_1 X_{i1} + V((X_{i1})_i G, A^{(1)}) \mid G\right)$$

となる．ここで $(x)_i G$ は第 i 選択肢で x を観測したときの事後分布，$A^{(1)} = (a_2, a_3, \cdots)$ である．すなわち，上記の右辺の最大値を達成する i を最初の選択肢とするのが最適な戦略である．

事後分布の計算は一般に積分となるが，上記の積分は n 回の観測ごとに k 個の選択肢があるため非常に高次元となり，そのまま計算するのは非現実的である．とりわけ実行時間というものには原理的に上限がないので，無限積分となりなおさら大変である．

これに対し，各確率過程 X_i が独立なベルヌイ試行の場合には，最適戦略を求める効率的なアルゴリズムが知られている．このため，bandit problem という用語をベルヌイ試行に限っている論文も多い．しかし実行時間がベルヌイ試行のようになるというのは期待しにくく，この成果はそのままは使えそうにない．

2.3 ベイズ統計によらない bandit problem の解法

統計パラメタが θ のときに strategy τ による payoff の期待値を **worth** と呼び $W(\theta, A, \tau)$ で表す．このとき常に最適な選択をした場合に対するロス

$$R(\theta, A, \tau) = \sum_m a_m \max_i \{E(X_i(\theta_i))\} - W(\theta, A, \tau)$$

を **regret** と呼ぶ．**Minimax strategy** は，統計パラメタが範囲 Θ に含まれるときの最悪の regret, すなわち $\sup_{\theta \in \Theta} R(\theta, A, \tau)$ を最小にする strategy である．

興味深い問題設定であるが，あいにく非常に簡単な場合にしか最適解が知られていない．また，統計パラメタによって regret が小さくできる場合も大きくならざるを得ない場合もあるが，それらを同じスケールで評価して最大値を取ってよいのかどうか，著者には俄かには判断できない．

3 提案手法

このように，既存の定式化は我々の問題に適用するのがなかなか難しい．そこで本稿では異なる定式化によりこの問題に取り組むことを試みる．あらかじめ述べておくと，以下の定式化は，少なくとも現状のままでは，統計学的には適切さを欠く部分があることを認めざるを得ない．しかし，2 種類の実装の選択では比較的うまく動作したように思われる．提案手法のよさと統計学的な適切さをうまく折り合いをつけられるよう，今後努力をしてゆきたい．

3.1 問題設定

まず問題設定を確認する．最初に述べたように，実装は 2 種類あるものとする．また反復回数は既知であり n 回とする．所要時間はとりあえず正規分布，また簡単のため分散が同じであるとしておく．すると，シフトとスケーリングにより所要時間の分布は $N(0,1)$ および $N(\delta,1)$ と仮定することができる．統計パラメタは δ ひとつだけであるが，その値が取りうる範囲は $(-\infty,\infty)$ すなわち実数全体とする．

3.2 パラメタ固定での最適試行回数

2 つの実装を m 回ずつ実行し，所要時間の平均値が短かった方を残りの $n-2m$ 回実行するという方法を仮定する．すると，統計パラメタ δ と試行回数 m を与えると全体の所要時間の分布が決まる．そこで，全体所要時間の期待値を最小にする m を数値的に計算する．

図 1: 左 : 反復回数 n と最適試行回数 m との関係，右 : 性能差 δ と最適試行回数 m との関係

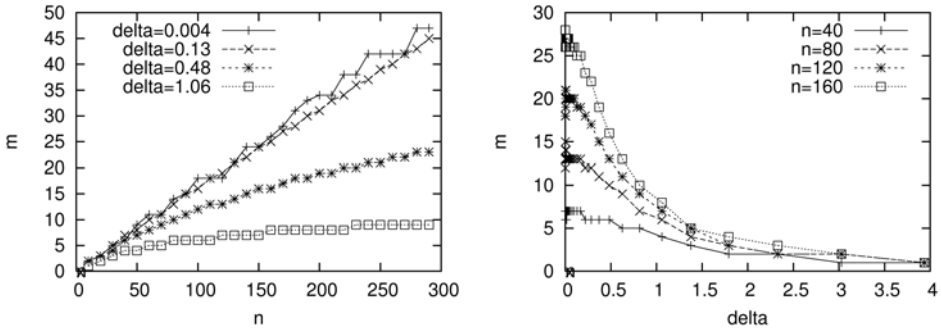


図 1 は得られた最適試行回数 m を，反復回数 n や性能差 δ に対してプロットしたものである．反復回数 n に対するプロットでは，性能差 δ が小さいところでは n に比例するある回数に収束しているように見える．性能差が大きくなると，n と m との関係は非線形になっている．性能差 δ に対するプロットでは，$\delta\to 0$ での挙動が確認できるほか，$\delta\to\infty$ で $m\to 1$ となることがよく見える．

以下では最適試行回数 m を $m(n,\delta)$ と表す．

3.3 パラメタ推定による実験計画

実際には統計パラメタは未知であるが，平均や分散はよく知られた方法で推定することができる．そこで次のような 2 種類の実験計画を考えた．

手法 1　2 つの実装を同じ回数ずつ試行し，性能差 δ を推定して，試行回数が $m(n,\delta)$ に達したら試行フェーズを終了する．具体的には以下の通り．

1. まず 2 つの手法を 2 度ずつ試行し，$m=2$ とする．
2. これまでの試行の結果から平均 μ_1,μ_2 と分散 σ_1^2,σ_2^2 の推定値を出す．
3. $\sigma=(\sigma_1+\sigma_2)/2$ とし，性能差を $\delta=|\mu_1-\mu_2|/\sigma$ で推定する．
4. 推定した δ と与えられた n から，$m(n,\delta)$ を表引きと補間で求める．$m<m(n,\delta)$ であれば，2 つの実装をもう 1 度ずつ試行し，m を 1 増やしてステップ 2 に戻る．

5. そうでなければ試行フェーズを終了し，残りの $n-2m$ 回は，平均所要時間が短かったほうで実行する．

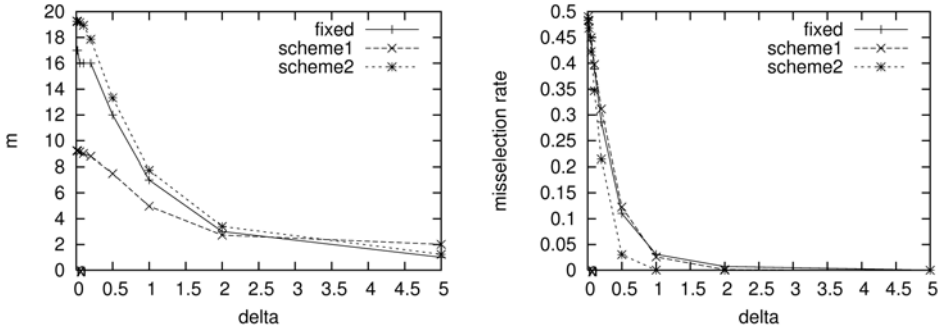
手法 2　性能差 δ を推定し続け，平均所要時間が長い実装の試行回数が $m(n,\delta)$ 未満であれば追加試行を行う．具体的には以下の通り．

1. まず 2 つの手法を 1 度ずつ試行し，速かったほうをもう 1 度試行する．
2. これまでの試行の結果から平均 μ_1, μ_2 と分散 σ_1^2, σ_2^2 の推定値を出す．
3. 実装 1 の試行回数 m_1 と実装 2 の試行回数 m_2 とする．$m_1+m_2=n$ なら終了する．
4. そうでなければ，
$$\sigma = \frac{(m_1-1)\sigma_1 + (m_2-1)\sigma_2}{m_1+m_2-2}$$
とし，性能差を $\delta = |\mu_1-\mu_2|/\sigma$ で推定する．
5. 推定した δ と与えられた n から，$m(n,\delta)$ を表引きと補間で求める．$\mu_1 < \mu_2$ の場合，$m_1 > m_2$ かつ $m_2 < m(n,\delta)$ であれば，実装 2 をもう 1 度，それ以外は実装 1 をもう 1 度実行する．$\mu_1 \geq \mu_2$ の場合も同様．ステップ 2 に戻る．

両者の大きな違いは 2 点ある．まず，手法 1 では遅い実装も最低 2 回試行されるが，手法 2 では 1 回ということもありうる．また，手法 1 では試行回数が十分と一度判定されれば判定は覆らないが，手法 2 では一度出した判断を取り消すこともありうる．

図 2 の左は，全体反復回数 n が 100 の場合について，真の性能差 δ と試行回数 m との関係を示している．3.2 節で論じたパラメタ固定の場合の $m(n,\delta)$ が `fixed` として示してある．また，手法 1, 手法 2 はそれぞれ `scheme1`, `scheme2` として示してある．なお，手法 1, 手法 2 での値を求めるにはモンテカルロ法を用い，後述の相対リグレットの推定精度が 2% になるまで反復している．

図 2: 左：性能差 δ と試行回数 m との関係 ($n=100$)，右：性能差 δ と選択誤り率との関係 ($n=100$)



手法 1 では試行回数が目標値 $m(n,\delta)$ よりも少ないのに対して，手法 2 では試行回数が目標値 $m(n,\delta)$ よりも多い．実験から推定する δ は真の値の前後にばらつくが，手法 1 では試行が足りたと一旦判断すれば終了してしまうため少なめになり，手法 2 では試行が足りていても δ のばらつきで不足と判断されれば追加試行がなされるため多めになったものと思われる．

図 2 の右は，同じ実験における選択誤り率，すなわち平均所要時間の真値が大きいほうが「速い」と判断された（手法 2 ではより多く選択された）割合を示している．パラメタ固定の場合と手法 1 とはほとんど変わりがないが，手法 2 は明らかに誤り率が低い．

すなわち，手法 2 は手法 1 に比べて，試行のコストがより多くかかっているが，速い実装をより多く選択している．両方のコストのバランスはどうなるだろうか．図 3 はリグレットを $n\delta/2$ で割ったもの（相対リグレットと呼ぶことにする）を示している．

図 3: 左：性能差 δ と相対リグレットとの関係 ($n = 100$)，右：同 ($n = 5$)

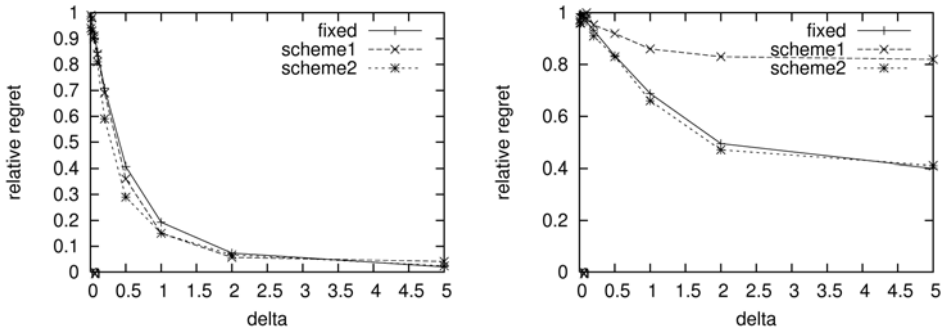


図 3 左は全体反復回数 $n = 100$ の場合の相対リグレットである．手法 1 はパラメタ固定の場合よりもよくなっているが，試行回数が少なくなった割に誤り率があまり高くないためであろう．手法 2 はさらによくなっているが，これは選択誤り率が低くなったためだと思われる．図 3 右は全体反復回数 $n = 5$ という最も少ない場合であるが，手法 1 では試行回数が少なくとも 2 回必要であるためリグレットが大きいが，手法 2 では試行回数を 1 回にできるのでパラメタ固定の場合と同じようなリグレットとなっている．

4 おわりに

本稿では実行時自動チューニングのための実験計画について考察した．従来手法そのままの定式化では解きにくい問題に対し，2 つの手法を提案した．実験的評価を行った結果，手法 2 は手法 1 よりもリグレットが小さかったが，これは試行フェーズと実行フェーズを厳密に分けず，常に実行データから判断の誤りを検出して修正できるためだと思われる．

今回はパラメタ固定の場合の最適試行回数 $m(n, \delta)$ を目標値にしたが，このような目標値の選択は統計学的に正当化できない．今後さらに検討を加え，統計学的に正当化できる手法を導きたい．もちろん，2 種類以上の選択肢がある場合や，連続パラメタの最適値の推定にも取り組まなければならない．

謝辞 竹村彰通，山本有作，直野健，今村俊幸，片桐孝洋の各氏からいただきました貴重なコメント・情報に感謝いたします．

本研究の一部は文部科学省の科学研究費「情報爆発時代のロバストな自動チューニングソフトウェアに向けた数理的基盤技術の研究」(特定領域研究 18049014) により行われています．

参考文献

[1] 片桐孝洋：ソフトウェア自動チューニング，慧文社 (2004).

[2] Berry, D. A., and Fristedt, B.: Bandit Problem, Chapman and Hall (1985).