数値計算にハッシュ法を応用する

高田, 健次郎
九州大学理学部物理学科

https://doi.org/10.15017/1470715

出版情報：九州大学大型計算機センター広報. 28 (2), pp.129-135, 1995-06. 九州大学大型計算機センター
バージョン：
権利関係：

数値計算にハッシュ法を応用する

高田 健次郎*

1．はじめに

計算機がずいぶん発達したとはいえ，我々が使えるリソースには限度があり，また用いる言語にも様々な制約があります．例えば九大大型計算機センターにおいてユーザーが利用できる主記憶の最大容量は約 300Mbytes ですし，FORTRAN ではどんなに大きな次元の配列を宣言したくても７次元を越えることは出来ません．しかし，大規模な数値計算をするときに，これらの制限や制約がほんの少し緩和されているならばずいぶん楽になるだろうに，と思うことがしばしばです．皆さんは（主記憶に入りきらないような）巨大な多次元の配列を宣言したくなったことはありませんか．その様な場合，我々は何とかして無駄なメモリーを使わないようあの手この手の工夫をこらして問題を解決します．またどうしても解決できないときは，当面計算の実行をあきらめ，もっと大きな計算機が使えるようになるまで課題を寝かせて（或は自分が寝て）待つより他ありません．

ちょっと待って下さい．本当に解決する良い知恵は無いのでしょうか．私は最近，ランダムな長さの多数の（例えば辞書のような）文字列データを扱う時にしばしば用いられるハッシュ (hashing) 法[1] と呼ばれるよく知られた方法が，場合によっては大規模数値計算にも応用可能であり，上に述べたような我々の悩みを解決してくれる知恵となり得ることを確かめました．勿論これが万能とは言いませんが，同じ悩みをお持ちの方には参考になろうかと思いますので，私の経験を御報告いたします．ハッシュ法については九大大型計算機センター長の有川 節夫 先生と，先生の研究室の大学院生の丸山 修 君に教わったことを白状して，お二人に感謝の意を表します．

2．メモリーの無駄使い

問題点をはっきりさせ，後の記述を分かりやすくするために，簡単な例を挙げることにします．

簡単のため２次元配列 $A(i,j)$ を考えます．$i=1,2,\ldots,30$ および $j=1,2,\ldots,100$ を取るものとすれば，この配列の宣言を `DIMENSION A(30,100)` とすることは言うまでもありません．しかし，場合によっては j の最大値が i の値によって著しく変化するようなことがありま

* 九州大学理学部物理学科

e-mail: kent2scp@mbox.nc.kyushu-u.ac.jp

す．つまり２次元配列 $A(i,j)$ の各行の実際の長さが行によってまちまちである場合です．この場合には，上の配列宣言 DIMENSION A(30,100) で確保したメモリーのうち，かなりの部分が実際には用いられないで無駄になり，“メモリーの無駄使い” がおきます．この２次元配列要素のうち，実際に必要なもののみを縦に１次元的に隙間無く並べて記憶してしまえば，メモリーの無駄は完全に無くなりますが，これだけではある特定の希望する配列要素を呼び出せなくなり，困ったことになります．

たった２次元程度なら，確保するメモリーもそれほど大きくなく，たとえ半分を無駄にしてもたいした問題にはならないでしょうが，６次元や７次元にもなると配列宣言するメモリーが巨大となり，たちまち大問題となります．

また，上の２次元配列 $A(i,j)$ $(i=1,2,\ldots,30,\ j=1,2,\ldots,100)$ が “sparse matrix” （パラパラ行列；0 でない行列要素が “まばら” に分布しているような行列）の場合にもメモリーの無駄使いが起きます．実際に記憶すべきは “まばら” な 0 でない行列要素だけであり，0 の行列要素は記憶する必要が無いにもかかわらず，配列宣言 DIMENSION A(30,100) でメモリーを確保してしまうからです．

別の例を挙げましょう．量子力学において４粒子の角運動量の合成を考えるとき，しばしば現れる 9-j 係数という量があります．これは原子・分子や原子核の物理学で頻繁に現れる量で，

$$\begin{Bmatrix} j_1 & j_2 & J_1 \\ j_3 & j_4 & J_2 \\ L_1 & L_2 & I \end{Bmatrix}$$

と表される９個の引数の関数です．私の専門分野の原子核の研究でしばしば必要となるのは，引数が $j_1,j_2,j_3,j_4 = 1/2,3/2,\ldots,15/2$，$J_1,J_2,L_1,L_2 = 0,1,2,\ldots,15$，$I = 0,1,2,\ldots,10$ といった値をとるものです．ある種の計算においては，この関数が極めて多数回呼び出される必要があります．呼び出される度毎に関数値を計算したのでは cpu 時間が幾らあっても足りません．従って，予めこれら全てを計算して主記憶に貯めておくのが理想的ですが，あまりに大量なので，単純な多次元配列を用意することは不可能です．ところが実はこの 9-j 係数は，その引数が次の６個の条件

$$|j_1-j_2| \le J_1 \le j_1+j_2, \qquad |j_3-j_4| \le J_2 \le j_3+j_4,$$
$$|j_1-j_3| \le L_1 \le j_1+j_3, \qquad |j_2-j_4| \le L_2 \le j_2+j_4,$$
$$|J_1-J_2| \le I \le J_1+J_2, \qquad |L_1-L_2| \le I \le L_1+L_2,$$

を満たすものだけが 0 でない値を持ち，それ以外は全て 0 となるという性質があります．その上，9-j 係数には引数の間に幾つかの対称性があり，その対称性を満たすものは全て等しい値を持つという性質もあります．つまり 9-j 係数を行列と見れば，非常に “パラパラ” な行列です．従って，9-j 係数を記憶するために安易な多次元配列宣言を行うと，大変な “メモリーの無駄使い” をすることになります．

このような“メモリーの無駄使い”を出来るだけ減らしてデータを記録し，かつ，ある特定の希望する配列要素を高速に呼び出す有効な手段を手にすることが出来れば万々歳です．これを実現しようとするのが以下で報告するハッシュ法の応用です．

3．安直派ハッシュ法の応用

この節で述べるハッシュ法の応用は，簡単な場合についてテストした結果十分実用的であることがわかり，かつ安直であるので，私が実際の大規模数値計算で大いに活用している方法です．次節で述べる正統派ハッシュ法と比べてそれ程遜色なく，単純なだけ使い易いと自慢しています．

A は正整数 $i_1, i_2, \ldots, i_n$ の関数（配列）であり，$A(i_1, i_2, \ldots, i_n)$ は引数 $(i_1, i_2, \ldots, i_n)$ に関して幾つかの価の組（点）を除いて 0 であるとします．（0 でない点がパラパラとまばらに分布していても，あるいは団子になっていても構わないが，かなり多くの点で 0 であることを想定します．）

それぞれの i_m の最大数を M_m とします $(m = 1, 2, \ldots, n)$.

この配列を記憶する最も不経済な (最も多数のメモリーを必要とする) 方法は，これを $M_1 \times M_2 \times \cdots \times M_n$ の行列に納めることです．最も経済的なのは，0 でない配列要素のみを 1 次元的に並べて記録することですが，その場合，与えられた $i_1, i_2, \ldots, i_n$ に対する値 $A(i_1, i_2, \ldots, i_n)$ がどのアドレスに存在するかを知る手段が必要です．すなわち，与えられた $i_1, i_2, \ldots, i_n$ の値 $A(i_1, i_2, \ldots, i_n)$ をいかに高速に呼び出すことが出来るかが問題で，その安直な方法を以下で紹介します．

How to memorize

1) 与えられた組 $(i_1, i_2, \ldots, i_n)$ をキーと呼ばれる通し番号 N で表わします．

$$\begin{aligned} N = {} & (i_1 - 1) M_2 M_3 \cdots M_n \\ & + (i_2 - 1) M_3 \cdots M_n \\ & \cdots\cdots\cdots\cdots \\ & + (i_{n-1} - 1) M_n \\ & + i_n \end{aligned} \tag{1}$$

従って，キー N の最大値は $N_{\max} = M_1 M_2 \cdots M_n$ です．$A(i_1, i_2, \ldots, i_n)$ は N の一意関数であるので，以後 $A(N)$ と表すことにします．

2) $N_{\max}$ 個の N を L_{bin} 個づつに等分割します．与えられた N が何番目の分割 k に属するかは

$$k = \left[\frac{N-1}{L_{\text{bin}}} \right] + 1 \tag{2}$$

で与えられます．このようにキー N と分割 k とを一意的に対応させる関数をハッシュ関数と呼びます．

$N_{\max}$ が $L_{\rm bin}$ で割り切れるときは分割の個数は $N_{\rm bin}=N_{\max}/L_{\rm bin}$，$N_{\max}$ が $L_{\rm bin}$ で割り切れないときは分割の個数は $N_{\rm bin}=[N_{\max}/L_{\rm bin}]+1$ で，この時は最後の分割の大きさは $L_{\rm bin}$ より小さいはずです．ここで，$[...]$ は Gauss の記号です．

3) $A(N)$ の 0 でない要素のみを キー N の順番に 1 次元配列 B の中に隙間なく記録します．同時に同じ順番に，その 0 でない要素に対応する N の値を 1 次元配列 N_B に記録します．また，k 番目の N の分割がこれら 1 次元配列の何番目から始まるかその位置 $N_s(k)$ $(k=1,2,\ldots,N_{\rm bin})$，および k 番目の分割の中の 0 でない $A(N)$ の個数 $M_s(k)$ $(k=1,2,\ldots,N_{\rm bin})$ を記録します．

How to read

1) 記録したデータの中から目標とする要素 $A(i_1,i_2,\ldots,i_n)$ を読み出すものとします．引数 $(i_1,i_2,\ldots,i_n)$ から (1) を用いてキー N を計算し，ハッシュ関数 (2) を用いてその N が属する分割 k を計算します．
2) 分割 k に属するデータは 1 次元配列 B と N_B の中の位置 $N_s(k)$ から始まる $M_s(k)$ 個の領域内に記録されているはずですから，この領域内で N_B を検索して N に合致するものがあればその点に対応する B が求める配列要素であり，該当する N が無ければその点に対応する配列要素 $A(i_1,i_2,\ldots,i_n)$ は 0 です．

さて，0 でない $A(N)$ の全個数を M としますと，この方法でデータを記録するために本質的に必要なメモリーは，M 個の B と M 個の N_B です．(それに分割の個数 $N_{\rm bin}$ だけの $N_s(k)$ と $M_s(k)$ が必要ですが，分割の大きさ $L_{\rm bin}$ を数 10 ～ 数 100 と適当な値にとれば，これらはそれほど大きなメモリーを占領しません．言うまでもなく，$N_s(k)$ があれば $M_s(k)$ は必須ではありません．) 従って，この方法は M が $N_{\max}$ に比べて小さければ小さいほど，すなわち 0 の要素が多ければ多いほど経済性が高いことになり，メモリーを節約することができます．

この方法では，先に 9-j 係数のところで述べたような引数の間の何等かの対称性を考慮してメモリーの節約をはかることも簡単です．たとえば，i_1 と i_2 を入れ替えても要素 $A(i_1,i_2,\ldots,i_n)$ の値が等しいようなときには，引数 $(i_1,i_2,\ldots,i_n)$ と $(i_2,i_1,\ldots,i_n)$ に対するキーを２つとも計算して比較し，小さいものに対するデータだけを記録します．読み出すときには常に２つのキーを計算して目的とする要素が２つのうちの大きいものに対応するときは，記録されている小さいものを読み出して代用させればよろしい．

この方法では，ある特定の要素を読みだすとき最大 $L_{\rm bin}$ 個のデータの中から検索しなければならず，運が悪ければ $L_{\rm bin}$ 個 全てを検索しなければならないこともあり，配列宣言をして

直接目的の配列要素にアクセスするのに比べて時間がとられるのが欠点ですが，L_{bin} の値をあまり大きくしないですむ場合は結構実用になります．世の中何でも良いことばかりではない，と悟りを開かねばなりません．

4．正統派ハッシュ法

もともとのデータのうち，0 でない要素が十分均等にパラパラに分布していないで団子状態になっているときには，前節の方法では各分割に記録されるデータの個数 $M_s(k)$ のばらつきが大きくなり，平均的な読み出し速度が若干遅くなる可能性があります．従って，記録するとき様々な分割にランダムに分布させ，$M_s(k)$ の値を k についてなるべく一様にするのが望ましいと思われます．勿論，ランダムに分布させるハッシュ関数は一意的でなければなりません．例えば平方採中法が良いようです．

また前節の方法ではキー N を等分割しましたが，必ずしも等分割が良いとはかぎりません．もっと合理的なハッシュ関数が考えられる場合もあります．

このようなきめ細かな工夫をするのが正統派ハッシュ法です．詳しくは参考文献[1] を参照して下さい．

確かに正統派が理論的には正しいだろうけれども，ランダムに分布させたり，巧妙なハッシュ関数を考案したりしても，そのハッシュ関数を計算するのに手間取って，無視できない余分な時間を食ってしまうのは問題です．また，私の経験によれば，前節の安直派に比べて正統派ではしばしば必要なメモリー容量が増え，それを見積もるのに苦労することがあります．確かにうまくチューニングすれば正統派のほうが平均的には少し速くなるでしょう．

5．簡単なテスト

先に述べた 9-j 係数を FORTRAN で実際に計算して，安直派と正統派のハッシュ法で記録し，それを呼び出すという簡単なテストをやってみました．

計算したのは引数が $j_1, j_2, j_3, j_4 = 1/2, 3/2, \ldots, 15/2$，$J_1, J_2, L_1, L_2 = 0, 1, \ldots, 4$，$I = 0, 1, 2, \ldots, 8$ といった値をとるものです．

これらを何の工夫もせずに，安易に 9 次元配列に記録すべく `real*8` の配列宣言をしたと仮定すると，(FORTRAN では 7 次元が最大ですからこれは出来ない相談ですが，) 約 184 Mbytes のメモリーが必要になります．ところがこの場合 0 でない 9-j 係数の全個数は 590765 個です．これらを安直派ハッシュ法で $L_{\mathrm{bin}} = 300$ の分割に記録すると，分割の個数は 76800，必要なメモリーは約 7.4Mbytes となり，上述の 184 Mbytes の約 1/25 になります．大変なメモリーの節約です．9-j 係数の対称性を考慮すれば，さらに 1/2 ～ 1/4 にすることは容易です．

ついでながら，正統派ハッシュ法で 80000 個の分割に平方採中法を用いてランダムに格納するよう工夫すると，必要なメモリーは約 24Mbytes となって安直派の 7.4Mbytes の約３倍となります.

ちなみに，九大大型計算機センターの UXP を使えば，上の 590765 個の 9-j 係数の計算時間は約 45sec，ハッシュ法で記録した全ての係数を各１回づつ全て読み出すための時間は，安直派と正統派の間に有意の差は無く，約 7sec でした.

計算時間はともかく，必要メモリーを 1/25 ～ 1/100 に節約できるということは，大規模数値計算をする上で大変な福音となるでしょう.

６．おわりに

いろいろと経験を積み重ねた結果，ハッシュ法を大規模数値計算に応用することによって，メモリーを大幅に節約出来る場合があることが分かりました.

なかでも第２節で述べた安直派ハッシュ法がお薦めです．安直派は正統派に比べて面白味が無く，凝り性の人にはうけが悪いでしょうが，何と言っても簡単で必要メモリーが少なく，その見積もりが容易ですので，実用一点張りを良しとする私のような者にはぴったりです.

勿論，メモリーを節約する工夫はもっと他にも考えられるでしょう．たとえば，全ての引数に関して１次元的に記録するのではなく，部分的に１次元化するようなことも考えられるでしょう．私自身もその様な工夫をしたことがありますが，何といってもハッシュ法の良い点はプログラムが単純で，機械的で，余計なあの手この手を考えなくてすむことです.

しかし，ハッシュ法が万能ではありません．最大の欠点はベクトル化にむかないことです．Vector processor を用いた現在のスーパーコンピューターのメリットを享受することができません.（誰か良い知恵を教えて下さいませんか？）逆にハッシュ法は研究室のパソコンやワークステーションにむいています．現に私は自宅で Pentium を搭載したパソコン（速いんだな，これが!!）に 40Mbytes の主記憶を増設し，OS として Windows NT を入れ，１週間も１０日もかかる計算をしました．ハッシュ法様々でした.

先に述べた安直派ハッシュ法はハッシュ関数がとても簡単で安直ですから，このような仕掛け（機能）そのものを埋めこんだ多次元配列専用の新案特許のメモリーチップは考えられませんでしょうか.（愚者のたわごと？）

最初に述べたように，ハッシュ法を私に教えたのは大型計算機センター長の有川 節夫 先生と院生の丸山 修 君でした．それなのに私が，ハッシュ法は大型計算機よりワークステーションやパソコンに適した方法であるなどと言うのは，恩を仇で返す仕業のようで心苦しい極みですが，たとえハッシュ法を使いまくってもなお大型機でなければ出来ない大規模数値計算を行う必要が必ずや生じると思いますので，その時恩に報いることができると思います.

皆様の御意見や御経験をお聞かせ下さい.

参考文献

1) A.V. エイホ, J.E. ホップクロフト, J.D. ウルマン; “データ構造とアルゴリズム” (大野義夫訳, 培風館, 1987 年)