

仮想記憶機構とプログラミング

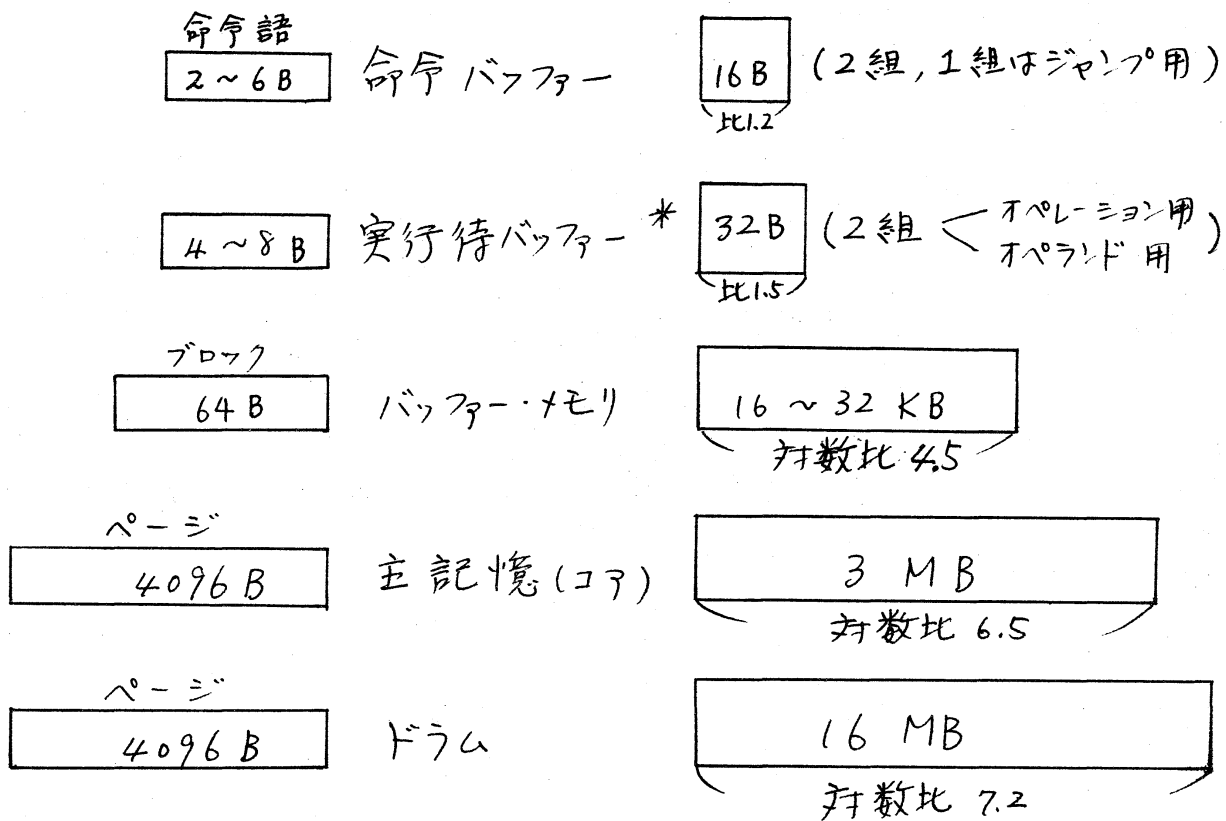
東京大学大型計算機センター

石田 晴久

1. はじめに

最近いろいろな計算機 (HITAC 8800/8700, IBM 370 シリーズ, FACOM 230 の 8 シリーズなど) に採用されるようになってきた仮想記憶は広義にとれば, バッファ・メモリ, 主記憶, ドラム (またはディスク) などからなるメモリのハイラーク構造もユーザから隠し, あたかも非常に大きなメモリがあるかのごとくみせかける技術である。東大の HITAC 8800/8700 システムの例をとれば, このメモリ・ハイラークは図 1 のようになっている。

こうした現実のメモリー・ハイラークは仮想記憶では (プログラマの意識する論理アドレスと計算機が使う実アドレスとが完全に分離されているため) プログラマは意識しなくてもよいことになっているが, 実際にはやはりそれを意識してプログラムを書く方が処理効率が高くなることが多い。これ



(a) 単位

(b) 記憶媒体

(c) 記憶容量 (Bはバイト)

図1. メモリ・ハierarchy例 (HITAC 8800/8700)

[*他に汎用レジスタ16個(各4B), 浮動小数点数レジスタ4組(各8B)]

はとくに大きなプログラムや大きなデータを扱うときにそうであって、現在の技術では大きなメモリを使うにはそれなりの工夫がいることを意味している。

ところで仮想メモリ系でメモリ・ハイラークシーを意識するとはいっても、単にプログラムを書く立場からいえば、仮想記憶の制御機構をそう細かに知る必要はない。最も知る必要のあるのは、プログラムが主記憶内のある与えられた領域(こゝを主記憶枠と呼ぶ)に入りきらないとき、残りの部分はドラムに置かれ、その後必要に応じて主記憶とドラムの間で情報の転送が行われるが、そのときどんなアルゴリズムでこの転送が行われるかということである。さらにもう一段下のレベルではバッファ・メモリのサイズを意識するとよいであろう。

本稿では HITAC 8800/8700 システムを例にとって、仮想記憶を中心にメモリ・ハイラークシーの制御機構と簡単に紹介し、それとプログラミングとの関係と考察する。

2. 仮想記憶とスワッピング・アルゴリズム

HITAC 8800/8700 では IBM 370 と同様にメモリは 4 キロバイト (KB) = 1 キロ語 (KW) のページという単位に分れて

いる。主記憶（コア）とドラムとの間で情報が転送されるとき、転送の単位となるのがこのページである。次にこの転送の行われ方も HITAC 8800/8700 のオペレーティング・システム（OS7）におけるメモリ制御にもとづいて説明する。

図2に示すように、OS7では実行可能な形になったオブジェクト・プログラムはいったんディスクに用意され、その後OSにより主記憶およびドラム内にメモリ領域の割当てを受けて実行を始めてゆくことになる。この際、主記憶は多重プログラミングで同時に走る他のプログラム（ジョブあるいはタスク）との共同利用になるから、ひとつのプログラムでその全体が使えるわけではなく、OSによって割当てられたある領域に限られる。この領域が前述の主記憶枠である。

もっと具体的にいえば、東大センターのシステムでは、ユーザの指定できるジョブ・クラス毎に、主記憶枠の標準値も次のように決めている。

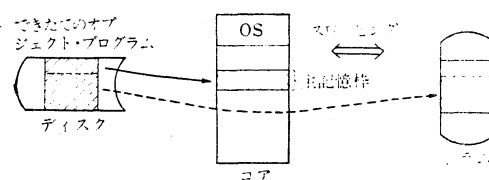


図2 プログラムのローディング

ジョブクラス	CPU時間 ^α 上限	プログラムの大きさ ^α 上限	主記憶枠
A	10秒	60 KW	34 KW
B	1分	320 KW	60 KW
C	10分	320 KW	80 KW
D	60分	1500 KW	256 KW (最大600 KW)

そこで、たとえばジョブ・クラスCで320 KWのメモリを使うプログラムがあったとすると、それが実行される際、図1でその4分の1の80 KW^{*}だけが主記憶にはいり、残りの240 KWはドラムに置かれることになる。

こうした主記憶とドラムとの間のメモリの割当て制御は、主記憶とドラムとの間の転送(スワッピング)も含めて、OSでは次のように行なわれる。

- (1) プログラムが割当てられた主記憶枠より小さいときは、ここはスワップはいりきるので、その中でそのまま実行

* ただしそのとき主記憶がたまたまあいていれば、もっと大きな主記憶枠がとられることもある。またプログラムが主記憶枠上限より小さければプログラムの大き^α分の主記憶枠がとられる。OSではこの意味で主記憶枠はプログラム実行中にダイナミックに変化する。

される。スワッピングはもうろん起らない。

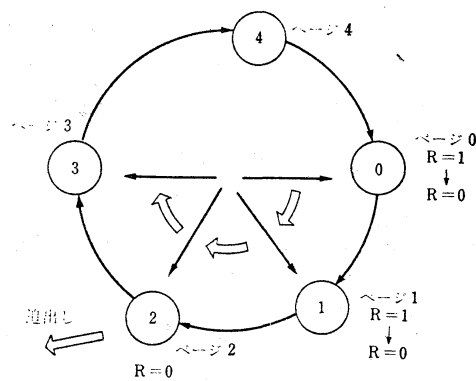
- (2) プログラムが主記憶域より大きいときは、プログラムの最初の部分から主記憶域へ入れ、はみ出した部分はドラムへ入れる。このときのドラムへの転送をスワップアウトという。
- (3) プログラムの実行が始まって、命令やデータの取出しのために、アドレスの参照が起るたびに、システム内で論理アドレスから実アドレスへの変換が行なわれる（実はこの変換をどのようにして行なうかが、仮想メモリの中心的課題であるが、ここではふれない）。この実アドレスが決められる過程で、そのアドレスの含む情報が主記憶にはないことが判明すると割込みが起る。
- (4) この割込みが起ると、ドラムからいま必要な情報を転送してやることになる。このドラムから主記憶への転送をスワップインとよぶ。OSではこの転送の単位は 1 ページ (1KB = 1024B) である。そこにおめる情報のはいっているページとドラムからちぎってくることはなすが、それには主記憶域 1 行かのどれか 1 ページを追い出さなければならぬ。しかしこの追い出しの際、そのページにまったく変更（書込み）がなく、しかもそのコピーがドラムに残っていれば、そのページは改めてドラムに

転送する必要はない。このため主記憶枠内1各ページごとにそのページに対して変更が行なわれたかどうかを表わす1ビットの目印が主記憶内[※]に必要となる。これをCビットという。このCビットは初め0でそのページへ書き込みが起るとハードウェアでC=1にするわけである。さてページを追いつけに当たって重要なのは、どのページを追いつけるかである。この決め方をスワッピング・アルゴリズムあるいは置換アルゴリズムという。OSで使われているものはLRU (Least Recently Used) アルゴリズムと単純化してFIFO (First In Not Used First Out) アルゴリズムである。この面白い名のアルゴリズムは、要するに“初めにはいつまで使われていないページからまず追いつけ”という意味である。そこでこれを実現させるために、各ページごとにそのページが参照されたかどうかを表わす1ビットの目印を主記憶内[※]に設けておく。これをRビットとよび、R=0は最近参照されていない状態を表わし、そのページ内の少なくとも1語が参照されると、ハードウェアでR=1にするものとしおく。ここでいう参照とは、読み出し・書き込み・実行のいずれかである。

※ 正確に言えば主記憶内のストレージキーはRビット、Cビットおよび3ビットのキーよりなる。

さて各主記憶枠内で常駐ページ（制御スワップ領域など）を除いて追いつき可能な各ページは図3のように連結リスト構造になっている。このページ・リストにはポインタがひとつあり、それが常にどれかのページをさしている。追いつきページを決めてスワップを行うためのFINLIFO アルゴリズムによる手順は次のとおりである。

(a) 再びポインタの示しているページの R ビットを調べる。^{*}
 $R=0$ なら最近参照されていない証拠だから、そのページを
 追出す。 $R=1$ ならこれを $R=0$ にかえて、ポインタを次のペー
 ジを示す位置に進めて、同じことを繰り返す。



Ⅳ) 3 FINLIFO アルゴリズム

* = 此には TRSK (Test and Reset Storage Key) 命令が使われる。

(b) 追い出すべきページが決まったら、そのページのビットを調べ^{*}。C=0なら変更がなかったしただから(しかもドラムにはそのページのコピーがある通常の場合には)、そのページのドラムへの転送はしないが、C=1ならこのページをドラムへ転送(スワップアウト)する。

(c) 次にドラムから新たに必要とするページをいまあったページへ転送(スワップイン)し、C=R=0にする^{**}。またポインタは次のページへ進めておく。

このようにすれば、プログラム全体に比べて小さな主記憶碑を使って、適当にスワッピングを起こせながらプログラムを実行させることができるわけである。なお実際には多重の多重プログラミングのもとでは、このユーザ空間あるいはシステム空間からページを追い出すかの選択がある。これはOSでは、システム空間および各ユーザごとに単位時間当りのスワップ回数を常時測定しておき次のようにしている。

(A) システム空間の追い出し可能ページ数が大きく、スワップ回数が少ない場合には、システム空間から追い出す。

* これは ISK (Insert Storage Key) 命令が使われる

** これは SSK (Set Storage Key) 命令が使われる

(B) n 個のページ空間のいずれかから追いつきときは、主記憶枠以上にメモリを使っている割にスワップ回数が少ないページの空間から追いつき出す。

この場合実行するジョブの多重度が高すぎると、スワップ回数ばかり増えて有効な仕事ができなくなる。これをスラッシング (thrashing, 原義はバタバタ打つこと) とよび、OSではこの状態が検出されると、自動的に多重度を減らすような制御が行われている。

3. 仮想記憶におけるプログラミング上の注意

前述のスワッピング・アルゴリズムから明らかのように、仮想記憶上を走るプログラムを書く際には、プログラム実行中の各時点で必要とされるメモリ量 (これを working set と呼ぶ) を小さくするように心がけるとよい。ワーキング・セットの小さいプログラムはまた局所性 (locality) が高いともいう。一方バッファ・メモリおよび仮想記憶両方について、データの読出しに比べて書込みの方はずっと遅いので、書込み (代入) もなるべくしない方がよい。これらを具体的にコーディング上の注意としてまとめてみると次のようになる。

(1) プログラムは小さなループをまわりながら少しずつ先へ

進むように構成する。

(2) 中間結果の書込み(代入)も避ける。あるいは中間変数の個数もへらす。

(3) 配列はメモリ内での格納順にアクセスする。また同じ配列へのアクセスを繰返すときは往復する形でアクセスする。

とくに FORTRAN で DIMENSION A(1024, 100) のとき

```
D0 100 J=1, 100
```

```
D0 100 I=1, 1024
```

```
100 A(I, J) = 0.0
```

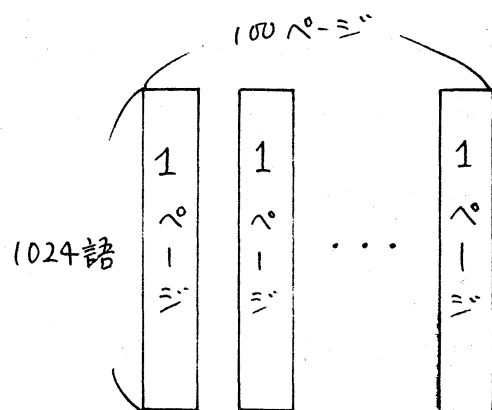
とするのはよい(図4(b))が、(主記憶枠が 100 kW 以下で最適化の指定をしないとき)逆に

```
D0 100 I=1, 1024
```

```
D0 100 J=1, 100
```

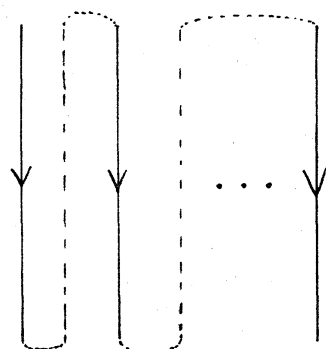
```
100 A(I, J) = 0.0
```

としたら大変である。これでは図4(c)のように格納順序とは逆の方向にアクセスすることになり、仮に主記憶枠内でこの配列の入る部分が 50 kW しかないとするとき、図4(d)のように当の主記憶枠の内容はめずか 50 語アクセスする毎にすっかり入れ替ってしまう。つまり 1 ページスワップしては 1 語アクセスするだけとなり、これでは計算機は全くのドラム・コンピュータになってしまうわけである。



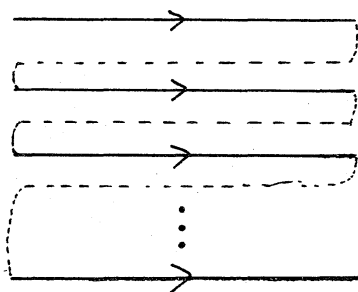
(a) 配列の格納

77 方向に
順に入る



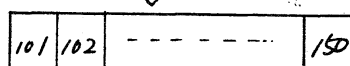
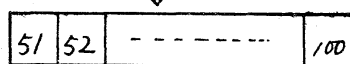
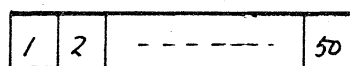
(b) 順方向アクセス

1 ページ内の
1024 語を順
にアクセスして
次のページへ



(c) 逆方向アクセス

1 ページ内で
1 語だけアク
セスして次のペ
ージへ



⋮

(D) 逆方向アクセスときの
主記憶の内容の変化

各4角は1ページ
(1kw)と表わす。
番号はJに対応

図 4 配列のアクセス法

(4) ひとつのループ内で使う変数・配列・副プログラムなどは連続したメモリ領域にまとめて格納されるようにする。この意味で DIMENSION 文に書く変数や配列の順序には注意した方がよい。プログラム単位の順番もそうである。

(5) 大きな配列は小さな部分に分割して一部分ずつアクセスする。

4. バッファ－・メモリとプログラミング

仮想記憶向きのプログラムを作った場合、あるいはプログラムやデータが比較的小さい場合、次に問題になるのはバッファ－・メモリの効果である。

図5にバッファ－・メモリのサイズ (HITAC 8800 で 8kW, HITAC 8700 で 4kW) を意識した計算法の1例を示す。このやり方で一度にアクセスする配列要素の数を l とすると、各段階でのデータ領域の広さは次のようになる。

	データ数	$N=100, l=20$	$N=200, l=20$	$N=300, l=20$
第1段階	$2l+1$	41	41	41
第2段階	$N(l+1)+l$	2,120	4,220	6,320
第3段階	$N(N+2l)$	14,100	48,000	92,000
従来の方法	$3N \times N$	30,000	120,000	270,000

$$\begin{matrix} A & & B & & C \\ \left(\begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \end{array} \right) & \times & \left(\begin{array}{c} | \\ | \\ | \end{array} \right) & = & \left(\begin{array}{c} \cdot \\ \cdot \\ \cdot \end{array} \right) \end{matrix}$$

(a) 第1段階 $A_{11} \sim A_{1i}$ と $B_{11} \sim B_{i1}$ を加えて C_{11} の部分積を出す (実線部)

$$\left(\begin{array}{c} \equiv \\ \equiv \\ \equiv \\ | \\ \text{---} \end{array} \right) \times \left(\begin{array}{c} | \\ | \\ | \\ | \\ | \end{array} \right) = \left(\begin{array}{c} | \\ | \\ | \\ | \\ | \end{array} \right)$$

(b) 第2段階 A の列を 1 から N まで変えて C_{j1} の部分積を出す

$$\left(\begin{array}{c} \equiv \\ \equiv \\ \equiv \\ | \\ \text{---} \end{array} \right) \times \left(\begin{array}{c} ||| \text{---} | \\ ||| \text{---} | \\ ||| \text{---} | \\ ||| \text{---} | \\ ||| \text{---} | \end{array} \right) = \left(\begin{array}{c} ||| \text{---} | \\ ||| \text{---} | \\ ||| \text{---} | \\ ||| \text{---} | \\ ||| \text{---} | \end{array} \right)$$

(c) 第3段階 B の行を 1 から N まで変えて C 全体の部分積を出す

(d) 以下 A の次の部分と B の次の部分 (実線部) について同様 $T_F = 1$ とを繰返す

(e) 残りの部分についても同様繰返す。

図5 バッファ・メモリ向きの計算法

図6はこの部分積による方法と従来の方法とのCPU時間比較を行うプログラムと測定された時間の例である。ここではメモリは計131kW, 主記憶棒60kWであるが, 主記憶がすいていて主記憶棒が溢れられたらしく, スワッピングは計47回しか起こらなかった。スワッピングがもっと起ってれば時間の差はもう少し開いたものと思われる。^{*}

次にこのような行列をひとつのループ内で数多く使うときには, バッファ・メモリがセクタ (HITAC 8800 で $2kB = 512w$, HITAC 8700 で $1kB = 256w$) という単位に分れていて, しかもそうしたセクタが16個しかないという事実が利いてくることがある。こうしたセクタは通常プログラム(純手続を)部分, 定数・変数部分, アドレス定数部分, 作業エリアに各1個ずつ使われることが多いので, 行列を入れる分は12セクタ程度となる。そこで各行列の大きさが1セクタ分以上あるいは各行列の格納場所が1セクタ以上離れている場合, 各行列は別々のセクタに入ることになるから, ひとつのループ内に13個以上の行列が出てくると, 全部はバッファ・メモリに入りきれなくなる。図7のようなプログラムを試してみると, この効果は図8のようにはっきり現われる。

^{*} ジョブ・クラス区を使い緊急タスクとして HITAC 8800 のみで処理させた。

```

      INTEGER A(200,200),B(200,200),C(200,200)
      L=20
      DO 500 N=L,200,L
      DO 100 J=1,N
      DO 100 I=1,N
      A(I,J)=I/10
      B(I,J)=J/10
      C(I,J)=0
1     CONTINUE
      CALL CLOCK(IC1,3)
      DO 300 K=1,N
      DO 300 I=1,N
      DO 300 J=1,N
30    C(I,K)=A(I,J)*B(J,K)+C(I,K)
      CALL CLOCK(IC2,3)
      C1=(IC2-IC1)/10.0
      DO 400 J=1,N
      DO 400 I=1,N
40    C(I,J)=0
      CALL CLOCK(IC1,3)
      DO 200 M=1,N,L
      MM=M+L-1
      DO 200 K=1,N
      DO 200 I=1,N
      DO 200 J=M,MM
200  C(I,K)=A(I,J)*B(J,K)+C(I,K)
      CALL CLOCK(IC2,3)
      C2=(IC2-IC1)/10.0
      WRITE(6,601)C1,C2,N
601  FORMAT(1H,2F15.3,I10)
500  CONTINUE
      STOP
      END

```

普通の
求積法

部分積
による
方法

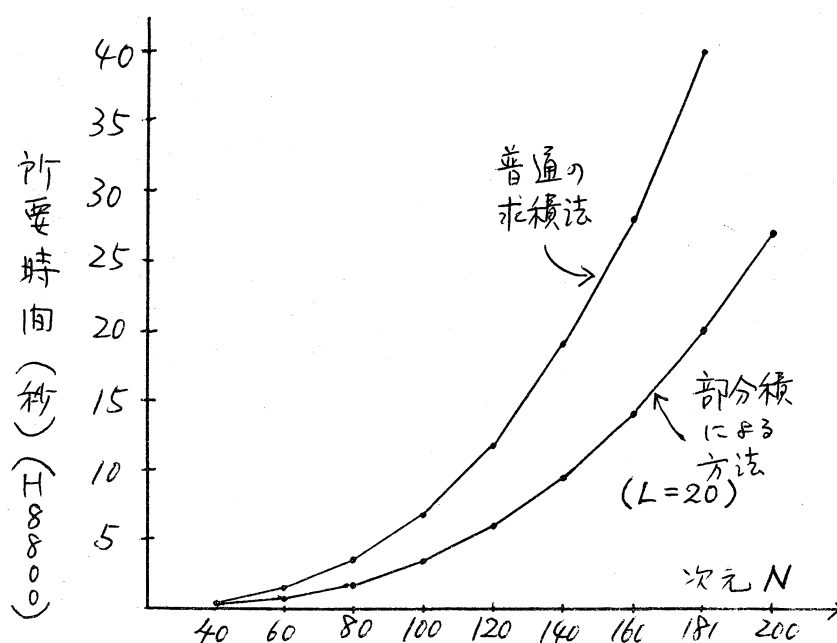


図6 行列の積を求めるプログラム

<pre> DIMENSION DUMMY(903) DIMENSION A1(1024),A2(1024),A3(1024),A4(1024),A5(1024) DIMENSION A6(1024),A7(1024),A8(1024),A9(1024),A10(1024) DIMENSION A11(1024),A12(1024),A13(1024),A14(1024),A15(1024) DIMENSION A16(1024),A17(1024),A18(1024),A19(1024),A20(1024) DIMENSION A21(1024),A22(1024),A23(1024),A24(1024),A25(1024) DIMENSION A26(1024) DO 300 L=1,13 CALL CLOCK(IC,3) DO 100 J=1,10 DO 100 I=1,1024 A1(I)=A2(I) IF(L.LE.1) GO TO 200 A3(I)=A4(I) IF(L.LE.2) GO TO 200 A5(I)=A6(I) IF(L.LE.3) GO TO 200 A7(I)=A8(I) IF(L.LE.4) GO TO 200 A9(I)=A10(I) IF(L.LE.5) GO TO 200 A11(I)=A12(I) IF(L.LE.6) GO TO 200 A13(I)=A14(I) IF(L.LE.7) GO TO 200 A15(I)=A16(I) IF(L.LE.8) GO TO 200 A17(I)=A18(I) IF(L.LE.9) GO TO 200 A19(I)=A20(I) IF(L.LE.10) GO TO 200 A21(I)=A22(I) IF(L.LE.11) GO TO 200 A23(I)=A24(I) IF(L.LE.12) GO TO 200 A25(I)=A26(I) 200 CONTINUE 100 CONTINUE CALL CLOCK(IA,3) TIME=(IA-IC)/10.0 WRITE(6,600) L,TIME 600 FORMAT(1H ,15,5X,F10.1) 300 CONTINUE STOP END </pre>		
	SYMBOL	TYPE LOCATION (16進)
	A1	AR 4 00001000
	A6	AR 4 00006000
	A11	AR 4 0000B000
	A16	AR 4 00010000
	A21	AR 4 00015000
	A26	AR 4 0001A000
	DUMMYを入れたので各 配列は16進で1000バイト、 すなわち10進で4KB(ページ) の区切りから始まっている。	
	(答)	(msec)
	1	105.9
	2	129.8
	3	153.9
	4	177.4
	5	202.0
	6	226.6
	7	253.8
	8	285.4
	9	325.2
	10	370.6
	11	420.2
	12	473.5
	13	534.3

図7 配列の代入プログラム

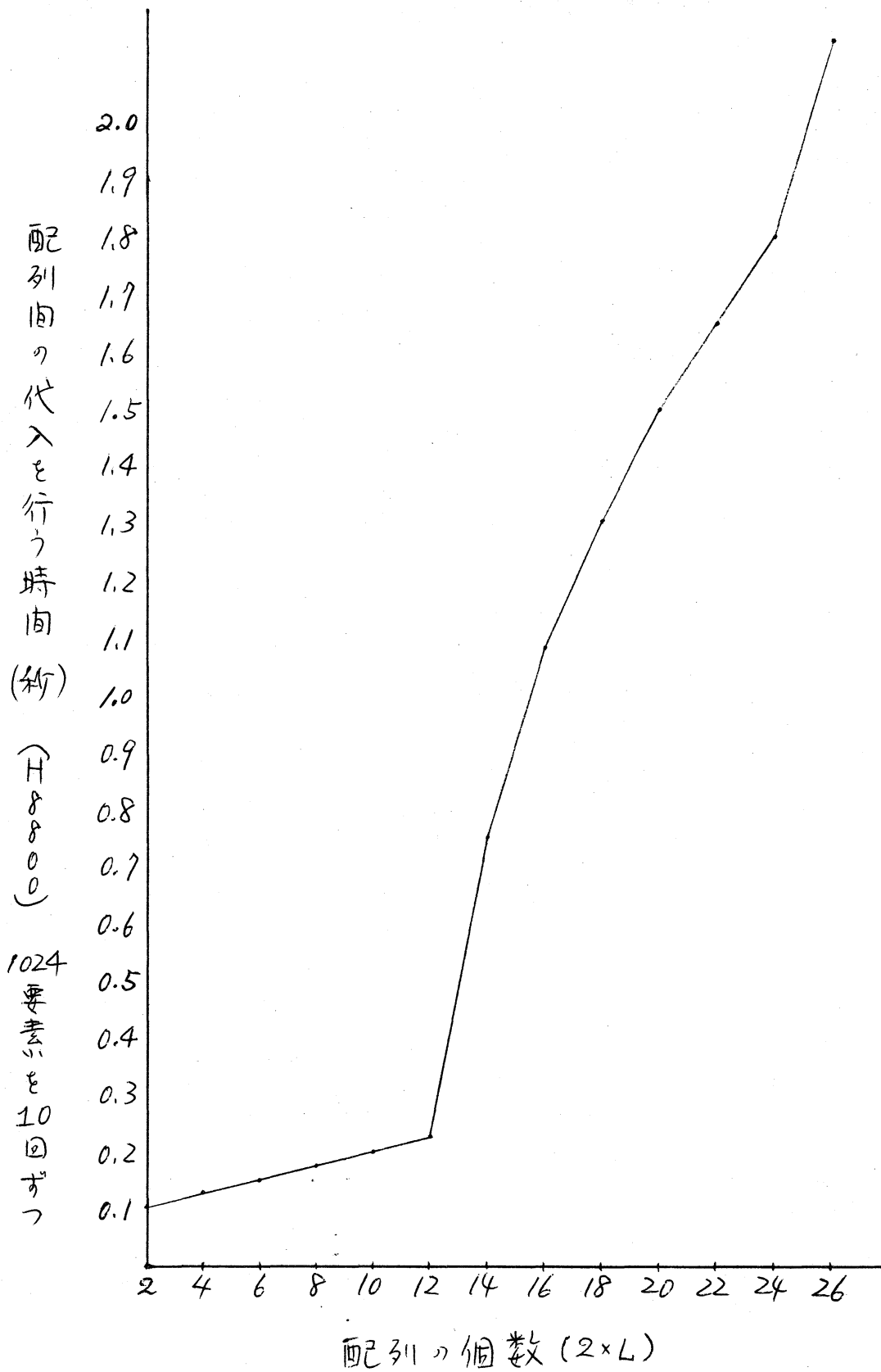


図8 バッファ-メモリの効果

5. パイプライン制御とプログラミング

仮想記憶とは直接関係ないが HITAC 8800 ではパイプライン制御 (高度先行制御) 方式がとられ, 図1にも示したように16 B の命令バッファが使われている。このため

命令バッファ内に入っている

→ LOOP: LD 0,0 (11,8) ④ / ^{バイト}B(L,J) を FLO へ
 AR 8,4 ② / L+8 → L (配列 B 用)
 MD 0,0 (12,7) ④ / FLO 上で A(I,L) をかける
 AR 7,5 ② / L+N*8 → L (配列 A 用)
 ADR 2,0 ② / FL2 に FLO をたしこむ
 BCTR 9,10 ② / 終りでなければジャンプ
 のようにループは 15 サイクル (約 800 nsec) の高速で実行される。

```

C PRIME NUMBERS
  DIMENSION M(600000)
  M(1)=1
  K=600000
  N=0
  DO 1 I=1,K
    IF(M(I).NE.1) GO TO 1
    DO 2 J=1,K,I
      M(J)=I
    2 CONTINUE
    N=N+1
    M(N)=I
  1 CONTINUE
  WRITE(6,100) N
100 FORMAT(6H SOSUU,I5)
  L=N-800
  WRITE(6,101)(M(I),I=L,N)
101 FORMAT(15I7)
  STOP
  END

```

図9 素数探しプログラム ----- 仮想記憶向きでない例
 (原プログラムは東大計数工学科の筧氏による)

6. おわりに

図1に示したようなメモリー・ハイラーキーをもつシステムでは、本稿でのべたように処理速度が非常に context sensitive であり、プログラムの性質（ふるまい）に大幅に左右される。したがって従来使われていたプログラムの中には、図9に示したエラトステネスのふるいによる素数探しプログラムのように、大きな実メモリには向いているが、仮想記憶には向かないものもある。実際このプログラムでは、主記憶枠を最大の600KWにとり、DIMENSION M(600000)として600000以下のすべての素数がわずかに数秒で求められたのに対し、DIMENSION (700000)としたら15分待つことも答の出なかった経験がある。

このことから想像されるように、従来の小さな(64KW ~ 128KW)実メモリ上で使われていたプログラムや数値計算法のうちには、メモリー・ハイラーキーをもつシステム用に再検討する必要があるものが多いと思われる。もちろん何ら特別の工夫をしなくとも大きなメモリの便へののが仮想記憶の特徴ではあるが、大きなメモリを“効率よく”使うにはそれだけの工夫がいるということである。

この意味から仮想記憶を含むメモリー・ハイラーキー系に關しては今後次のような研究が望まれます。

- (1) 仮想記憶・バッファメモリに適した数値計算法およびプログラミング技法の開拓。
- (2) 従来のライブラリー・プログラム・コンパイラの改良。
- (3) 大きなメモリ上でのプログラムのふるまいの実態調査や分析, それをもとづくモデル化
- (4) 新しいスワッピング・アルゴリズムの工夫。これには上記(3)の成果が資料となる。
- (5) スワッピング用ハードウェアの改良やドラムのICメモリ化。
- (6) クラスタリングなどによるプログラム配値の最適化。
- (7) 仮想記憶系の性能の定量的な評価。[これには仮想記憶のあかばでプログラミングの手術が省けるというピーポル・パフォーマンスの要素が入るのではなかなか難しい。]

最後に本稿を手とめるに当り, 資料の提供およびさまざまなご教示を頂いた日立製作所の大西勲氏, 大不尚氏, 村田健部氏らに深く感謝したい。

参考文献

1. 石田: 仮想記憶(1)(2)(3), bit, vol.5, No.11 ~ No.13