数値計算レポート No.03
(5月19日出題分)

匂坂岳志 (G99P701-7)

2001年6月2日

1 課題

double 型 で 1000 x 1000 行列の積を計算する C プログラムを作成せよ。
10 以上の組合せで計算して平均時間を可能な限り短縮せよ。

2 成果

2.1 性能

このプログラムは学内第5端末で可能な限り高速に動作するように設計した。ベンチマークテストの結果では、このプログラムが 1000 x 1000 行列の積を計算するためにかかる平均時間は 4.01 秒 である。
プログラムの実行に際しては、実行環境のリソースを可能な限り最大限に有効利用したつもりだが、X Window System や Window Manager を利用しないようにすることもできれば、3秒台も可能であると考えている。
ちなみに第5端末室の構成は以下の通り。

OS	CPU	Clock	Memory
Linux	Pentium III	800MHz	256MB

(注)
Pentium III 800 MHz の L1 Cache は 32KB、L2 Cache は 256KB。
600 MHz 以上のクロックと十分なメモリがあれば、マシン性能によっても結果はそう大差ないようである。

2.2 プログラムソースコード

以下に最高のベンチマークを示したプログラムのソースコードを載せる。

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define ORGSIZE 1000
#define BLKSIZE  125
static double  m[ORGSIZE][ORGSIZE],
               m1[ORGSIZE][ORGSIZE],
               m2[ORGSIZE][ORGSIZE];

void mulmtrx(double [][], double [][], double [][]);

int main(void){

  int i,j,x,y,z,dx,dy,dz,nblk;
  double *tmpx, *tmpy;
  double  sbm[BLKSIZE][BLKSIZE],
         sbm1[BLKSIZE][BLKSIZE],
         sbm2[BLKSIZE][BLKSIZE];
  clock_t start, end;

  /* 初期化 */
  for(i = 0; i < ORGSIZE; i++){
    for(j = 0; j < ORGSIZE; j++){
      m[i][j] = 0;
    }
  }

  srand(time(NULL));
  for(i = 0; i < ORGSIZE; i++){
    for(j = 0; j < ORGSIZE; j++){
      m1[i][j] = (double)rand();
      m2[i][j] = (double)rand();
    }
  }

  nblk = ORGSIZE / BLKSIZE; /* ブロック数 */

  start = clock();

  for(x = 0; x < nblk; x++){
    dx = BLKSIZE * x;
    for(y = 0; y < nblk; y++){
      dy = BLKSIZE * y;

      /* m1 から sbm1 へのコピー */
      for(i = 0; i < BLKSIZE; i++){
        tmpx = (*(sbm1 + i));
        tmpy = ((*(m1 + dx + i)) + dy);
```

```
        (*tmpx) = (*tmpy);
        for(j = 0; j < nblk - 1; j++){
          (*(tmpx++)) = (*(tmpy++));
        }
      }

      for(z = 0; z < nblk; z++){
        dz = BLKSIZE * z;

        /* m2 から sbm2 へのコピー */
        for(i = 0; i < BLKSIZE; i++){
          tmpx = (*(sbm2 + i));
          tmpy = ((*(m2 + dy + i)) + dz);
          (*tmpx) = (*tmpy);
          for(j = 0; j < nblk - 1; j++){
            (*(tmpx++)) = (*(tmpy++));
          }
        }

        /*  sbm1 x sbm2 ブロック行列の乗算 */
        mulmtrx(sbm, sbm1, sbm2);

        /* sbm を m に加える */
        for(i = 0; i < BLKSIZE; i++){
          tmpx = ((*(m + dx + i)) + dz);
          tmpy = (*(sbm + i));
          (*tmpx) += (*tmpy);
          for(j = 0; j < nblk - 1; j++){
          (*(tmpx++)) += (*(tmpy++));
          }
        }
      }
    }
  }

  end = clock();

  printf("TIME = %f\n", (double)(end - start)/CLOCKS_PER_SEC);

  return 0;
}

void mulmtrx(double  sbm[BLKSIZE][BLKSIZE],
             double sbm1[BLKSIZE][BLKSIZE],
             double sbm2[BLKSIZE][BLKSIZE]){

  int i,j,k;
  double tmp1, tmp2, tmp3, tmp4, tmp5;

  for(i = 0; i < BLKSIZE; i++){
    for(j = 0; j < BLKSIZE; j++){
      tmp1 = tmp2 = tmp3 = tmp4 = tmp5 = 0;
      for(k = 0; k < BLKSIZE; k++){
```

```
        tmp1 += sbm1[i][k] * sbm2[k][j];
        k++;
        tmp2 += sbm1[i][k] * sbm2[k][j];
        k++;
        tmp3 += sbm1[i][k] * sbm2[k][j];
        k++;
        tmp4 += sbm1[i][k] * sbm2[k][j];
        k++;
        tmp5 += sbm1[i][k] * sbm2[k][j];
        k++;
      }
      sbm[i][j] = tmp1 + tmp2 + tmp3 + tmp4 + tmp5;
    }
  }
}
```

3　工夫

今回作成したプログラムには以下のような工夫を施して高速化を図った。

3.1　行列のブロック分割

CPU の L1,L2 キャッシュを有効利用するために、1000 x 1000 行列を 64 個の 125 x 125 行列に分割した。

3.2　ブロック行列乗算サブルーチン

行列の要素のコピー、加算もサブルーチンにしてみたが、ブロック行列の乗算を行う部分のみをサブルーチンにしたときにコンパイラによる最適化が最も有効になった。

3.3　メイン関数内での 2 次元配列の分解

メイン関数内で行う行列の要素のコピー、加算は 2 次元配列を利用せずに、全てポインタを利用して記述することで、実メモリ上にある要素を、より高速にアクセスできるようした。

3.4　行列乗算アルゴリズム

実際この計算は CPU キャッシュを利用して行われることを期待しているので、メイン関数内と同様にポインタを利用して 2 次元配列 を分解してしまうと逆に遅くなってしまう。2 次元配列を利用したままで最も高速な処理を可能にするために、ループのアンローリングを利用することになった。

3.5　ループ変数順序最適化

実メモリへのアクセス時間を短縮するために、サブルーチンともにループ変数の順序を最も高速になるようにした。

3.6　コンパイラによる最高最適化

プログラムのコンパイラには gcc(cgcs-2.91.66) を利用し、-O3 オプションを付けて最高の最適化を行った。

3.7　定数の変数化

行列の縦横それぞれのブロック数は、ブロックのサイズを 125 と決定した時点で、必然的に 8 (1000 / 125) と決まるのだが、それをあえてメイン関数内の局所（ローカル）変数とすることで、コンパイラによる最適化が行われるようになった。

3.8　実行環境リソースの最大有効利用

プログラムの実行に際しては、CPU やメモリといったリソースを最大限に利用できるように、並列で実行させるプログラムは可能な限り必要最低限のプログラムのみにした。

4　感想

最終的なプログラムのソースコードを見ただけでは分かってもらえないかも知れないが、「工夫」に挙げたことを実現させる他にも、このプログラムを作成するまでには本当に多くの試行錯誤があった。

またプログラムの調整も佳境に入ると、実行平均時間を 0.01 秒を短縮するのにも必死になった。特に 5 秒の壁を越えるには本当に苦労した。

Department of Information and Computer Science,
School of Science and Engineering, Waseda Univ.
Sagisaka Takeshi< *g99p7017@mse.waseda.ac.jp* >