MPIまたはCUDAを用いた将棋評価関数学習プログラムの並列化

2009/06/30

目次

1. まえがき・・・・・・・・・・・・・・・・・・・・・ 3
2. 計算方法・・・・・・・・・・・・・・・・・・・・・ 4
3. MPIを用いた並列化・・・・・・・・・・ 6
4. CUDAを用いた並列化・・・・・・・11
5. 計算結果・・・・・・・・・・・・・・・・・・・・・20
6. まとめ・・・・・・・・・・・・・・・・・・・・・・・・24

1. まえがき

〇目的
将棋の評価関数を棋譜から学習するボナンザメソッドの簡易版を作成し、それをMPIまたはCUDAを用いて並列化し、計算時間を短縮することを目的とする。

〇評価関数
評価関数は駒割のみを考え、静止探索を考えない。従って、評価関数は盤面81個と持ち駒7*2個の合計95個の和である。
歩の価値=1を固定とする。特徴ベクトルは、香～飛、と～龍までの合計12個の駒の価値である。

○MPI
MPI(Message Passing Interface)は共有メモリーと分散メモリーの混合環境で、プログラムを並列化するものである。
本プログラムは並列度が高いので、容易に並列化することができる。

○CUDA
NVIDIA製のビデオカードを持つPCで、多数のコアによる並列計算を行うものである。
C言語に少しの拡張を加えたCUDAと、通常のC/C++の混合により開発する。
メモリー構造が複雑なので、性能が出せるようにアルゴリズムを設計することが重要である。

2. 計算方法

次式で定義される特性関数を最小化する。

$$J(\boldsymbol{v}) = \sum_{i=0}^{N-1} \sum_{m=0}^{M_i-1} T\left(\left(f(p_{i,m}, \boldsymbol{v}) - f(p_{i,0}, \boldsymbol{v})\right)\sigma_i\right) \tag{1}$$

記号の意味は以下の通りである。

p_i ： 棋譜の局面(親局面、 合計N個)
M_i ： 局面 p_i の合法手の数(平均80)
$f(p_{i,m}, \boldsymbol{v})$ ： 局面 p_i を合法手mで進めた局面の評価関数 、m＝0は棋譜の手
$\boldsymbol{v}$ ： 特徴ベクトル(L次元)
$\sigma_i = +1$ (先手番), -1 (後手番)

関数Tは次式で定義されるsigmoid関数である。

$$T(x) = \frac{1}{1+\exp(-ax)} \tag{2}$$

特徴ベクトルvに適当な初期値を与えて、以下の反復計算により、特性関数Jを最小化する。

$$v_l^{new} = v_l^{old} - h * sign\left(\frac{\partial J(\boldsymbol{v})}{\partial v_l}\right) \quad (l = 0, \dots, L-1) \qquad (3)$$

$$sign\left(\frac{\partial J(\boldsymbol{v})}{\partial v_l}\right) = sign\left(J(\boldsymbol{v} + \Delta \hat{v}_l) - J(\boldsymbol{v})\right) \qquad (4)$$

$\left(\Delta \hat{v}_l\text{：第l成分だけ正の微小量を持つベクトル}\right)$

ここで、変動量hは反復回数ごとに小さくする。

演算回数は「反復回数 * N * 80 * L」と評価できる

3. MPIを用いた並列化

計算は以下のような5重ループになる。右の数字は標準的なループ長である。

(1)収束反復計算	10～10^2
(2)特徴ベクトル次元に関するループ	10～10^4 (=L)
(3)棋譜局面に関するループ	10^3～10^6 (=N)
(4)合法手に関するループ	10^2 (= M_i)
(5)評価関数計算	10^2～10^4

(1)は本質的に逐次計算であるから並列計算はできない。
(2)～(5)についてはすべて並列計算可能である。
MPIは並列処理のオーバーヘッドが比較的大きいので、上位に近い方が効率がよい。
ここでは、プログラミングの容易さも考慮して、ループ(2)を並列処理する。

表1がMPIで並列化した反復計算部のソースコードである。
特徴ベクトルに関するループを分割して並列処理している。
表中の赤い部分が並列化のために書き直したところである。10数行の変更で並列化できる。呼び出すMPI関数はMPI_Allgatherの一箇所である。

#ifdef MPI～#endifの部分を削除すると、通常の逐次コードになる。

Windowsでのコンパイル・実行コマンドは以下のようになる。

```
> cl -Ox -DMPI -Fea.exe *.c mpi.lib
> mpiexec -n 4 a.exe
```
（共有メモリーのとき、数字はコア数）

（注）`mpiexec.exe`が環境変数`PATH`に、`mpi.h`が環境変数`INCLUDE`に、`mpi.lib`が環境変数`LIB`に含まれていることが必要。

表1 MPI ソースコード(反復計算部)

```
// 反復計算
void iter(int nproc, int mproc, const int iparam[], const float rparam[], const float fv[])
{
    float fv_old[NFV], fv_dif[NFV], fv_new[NFV];      // NFV : 特徴ベクトルの次元L

    // パラメータ
    const int itermax = iparam[6];       // 反復回数
    float h = rparam[0];                 // 特徴ベクトル変化量の初期値
    const float hfactor = rparam[1];     // 特徴ベクトル変化量の縮小因子
    const float eps = rparam[2];         // 特徴ベクトルの差分

    // 特徴ベクトルのインデックス
    // 逐次版では nlength = NFV, nstart = 0
    int nlength = (NFV % nproc == 0) ? (NFV / nproc) : (NFV / nproc + 1);
    int nstart = mproc * nlength;

#ifdef MPI
    // MPI用変数
    float *send = malloc(nlength * sizeof(float));
    float *recv = malloc(nlength * nproc * sizeof(float));
#endif

    // 特徴ベクトル初期値
    for (i = 0; i < NFV; i++) {
        fv_old[i] = fv[i];
    }

    // 反復計算
    for (i = 0; i < itermax; i++) {
        // 中心値
        // 並列計算のときは全プロセッサーで重複計算する
        j0 = jcalc(fv_old, iparam);
```

```
        // 特徴ベクトルに関するループ
        for (n = nstart; n < MIN(nstart + nlength, NFV); n++) {
            // 特徴ベクトルの第n成分を変化させる
            for (k = 0; k < NFV; k++) {
                fv_dif[k] = fv_old[k] + ((k == n) ? eps : 0);
            }
            // 特性関数計算
            j1 = jcalc(fv_dif, iparam);
            // 特徴ベクトル更新、特性関数の増減から判断する
            fv_new[n] = fv_old[n] - ((j1 > j0) ? +h : -h);
#ifdef MPI
            // 送信用バッファに保存する
            send[n - nstart] = fv_new[n];
#endif
        }
#ifdef MPI
        // 各プロセッサーで計算された特徴ベクトル値を全プロセッサーで共有する
        MPI_Allgather(send, nlength, MPI_FLOAT, fv_new, nlength, MPI_FLOAT, MPI_COMM_WORLD);
#endif
        // 次回の特徴ベクトル
        for (n = 0; n < NFV; n++) {
            fv_old[n] = fv_new[n];
        }

        // 差分幅更新
        h *= hfactor;
    }
#ifdef MPI
    free(send);
    free(recv);
#endif
}
```

表2 MPI版の計算時間

プロセッサー数	計算時間[秒]	速度比
1	1313.7	1.00
2	709.4	1.85
3	507.2	2.59
4	407.6	3.22

●動作環境：
CPU： Xeon E5430 (2.66GHz)
OS：Windows Vista 64ビット
コンパイラ：Visual C++ 15.0(64ビット)
MPICH2 1.1 (64ビット)

●計算条件：
棋譜数=2000
手数=40-79
局面数=80000弱
評価局面数=6,315,907
反復回数=50

●考察：
・プロセッサー数が多くなると計算時間が短縮されることがわかる。
・本プログラムでは中心値(式(4)の右辺第2項)を全プロセッサーで重複計算しているので、プロセッサー数Pのときの

速度比 = (L + 1) / (L/P + 1)

となり、これにL=12, P=4を代入すると、

速度比 = (12 + 1) / (12/4 + 1) = 3.25

となり、表2の3.22に近いので、ほぼ理想的に並列計算が行われていることがわかる。
Lが十分大きくなると速度比はPに近づく。
・1プロセッサーのときのNPSは6.316M * 50 * (L + 1) / 1313.7 = 3.1[MNPS]

4. CUDAを用いた並列化

CUDAではMPI版で用いたコードそのまま並列化しても性能は出ないので、以下のように変更する。
前章のループ(3)(4)を合体してgridとし、ループ(5)をblockとして並列計算を行う。
式(1)の子局面$p_{i,m}$を予め計算しp_kとおくと、式(1)の二重和は式(5)の一重和になる。
ここで、n(k)は子局面kに対応する棋譜の指し手(以下本譜と呼ぶ)で進めた子局面の番号、σ_kは子局面kに対応する親局面の手番である。
n(k)とσ_kは子局面p_kを作成するとき同時に作成しておく。
そのとき、最初に本譜で子局面を作成し、続いて全合法手(その中に本譜が一つ含まれる)の子局面を作成する。そうすると、どの合法手が本譜と一致するか調べる必要がない。本譜は式(5)に定数項T(0)=1/2を加えるだけであり、式(3)からわかるようにJの傾きのみが必要なので、Jに定数を加えても結果は同じである。従って、本譜は重複して計算してもかまわない。
以上から、子局面の総数Kは式(6)となる。
以下では、式(5)を並列処理により高速に計算することを考える。

$$J(\boldsymbol{v})=\sum_{k=0}^{K-1} T\left(\left(f(p_k,\boldsymbol{v})-f(p_{n(k)},\boldsymbol{v})\right)\sigma_k\right) \tag{5}$$

$$K=\sum_{i=0}^{N-1}(M_i+1) \tag{6}$$

局面データ

局面データは盤面81個と先後の持ち駒の数7*2個の合計95個である。
型はunsigned charでよい。
なお、global memory coalescingを考慮して、局面データの数は16の倍数である96個としたほうがいい。（一つダミーデータを加える）
従って、一つの局面は96バイトから成る。
これ以外に、n(k),σ_k,評価関数fが必要であるから、合計96+4+4+4=108バイトになる。
使用できるglobal memoryは512MB～1GBからOS分を引いたものであるから、これを仮に900MBとすると、子局面の上限数は900*10^6 / 108 = 8.33*10^6 個である。合法手の数を80とすると、これは約100000個の親局面を意味する。
これより大きな親局面を計算するには何らかの方法で分割処理することが必要であるが、ここではそのようなケースは考えない。

用語：
host memory : メインメモリー(CPU)
device memory　または global memory : ビデオメモリー(GPU)
shared memory : マルチスレッドで共有する高速なメモリー(GPU)
host コード : CPUで処理するコード
device または kernel コード : GPUで処理するコード
core : 1スレッドを処理する単位
block : スレッドの集合(上限512)
grid : blockの集合(上限は事実上なし)
threadIdx,blockDim,blockIdx,gridDim : CUDAの予約語(.x,.y,.zを持つ構造体)

並列処理

式(5)の計算を以下の二つのステップに分けて並列処理する。(表3)

(1) 子局面の評価関数の計算(表4)
一つの子局面を一つのblockに割り当て、評価関数をマルチスレッドで並列計算する。
ここでは、評価関数は駒割のみであるから、96スレッドで計算できる。(注1)
gridサイズは子局面の数Kである。
得られたK個の評価関数値をglobal memoryに格納する。

具体的な処理内容は以下の通りである。
1) 局面の各要素をglobal memoryからregisterにコピーする
2) 各要素の価値を計算してshared memoryに格納する
3) shared memoryの和をreduction操作(注2)で計算し、結果をglobal memoryに格納する。
以上で、時間のかかるglobal memoryへのアクセスは最初と最後の計2回ですむ。

(2) 特性関数Jの計算(表5)
式(5)の和を並列計算する。項数Kはスレッド数の上限512を超えるので、K/512個のblockに分割する。各blockは512スレッドで部分和を計算する(要reduction操作(注2))。
最後に部分和の和を計算する。

計算時間は(1)が大部分を占め、(2)の割合は小さい。

(注1)コアの数は通常96より大きいが、実行時に全部のコアがフル稼働するようにgridが分配されるので、性能は低下しない。
(注2)reductionとはマルチスレッドで配列の和などを計算することで、N個の和を$\log_2 N$回の演算で計算することができる。CUDAのサンプルプログラムreduction参考。

表3 CUDA版並列計算コード(jcalc.cu)

```
// 特性関数Jの計算
// fv : 特徴ベクトル
// 関数値 : 特性関数値の平均
float jcalc(const float fv[])
{
    // (1)評価関数の計算
    // copy host to device
    cudaMemcpy(d_fv, fv, NFV * sizeof(float), cudaMemcpyHostToDevice);
    // kernel
    dim3 grid(nleaf);
    dim3 block(96);
    evaluate_gpu<<<grid, block>>>(d_kyo_db, d_fv, d_fval, nleaf);

    // (2)特性関数の和を計算する
    // 部分和を計算する
    int ncore = 512;
    int nblock = (nleaf + ncore - 1) / ncore;
    sumf<<<nblock, ncore, ncore * sizeof(float)>>>
        (nleaf, ncore, d_fval, d_adr0_db, d_sente_db, d_subarray);
    // copy device to host
    cudaMemcpy(subarray, d_subarray, subarray_size, cudaMemcpyDeviceToHost);
    // 和,floatでは桁落ちが発生する
    double sum = 0;
    for (int i = 0; i < nblock; i++) {
        sum += subarray[i];
    }

    // 特性関数値の平均を返す
    return (sum / nleaf);
}
```

表4 CUDA版評価関数計算コード(evaluate_gpu.cu)

```
__global__
void evaluate_gpu(unsigned char kyo[], const float fv[], float fval[], int maxsize)
{
    // shared memory
    __shared__ float s[KYOSIZE];

    // スレッド番号
    int tid = threadIdx.x;
    // ブロック番号
    int bid = blockIdx.x + (blockIdx.y * gridDim.x);
    // 配列上限チェック(局面数がblock数の倍数とは限らないため)
    if (bid >= maxsize) return;
    // 局面要素
    unsigned char uc = kyo[tid + (bid * KYOSIZE)];

    float v = 0.0f;

    // 盤面
    if (tid < 81) {
        if (idx[uc] >= 0) {
            v = (uc & 0x10) ? fv[idx[uc]] : -fv[idx[uc]];
        }
        else if (idx[uc] == -1) {
            v = (uc & 0x10) ? +1.0f : -1.0f;
        }
    }
    // 駒台
    else if (tid == 81) {
        v = uc;
    }
    else if (tid < 88) {
        v = fv[tid - 82] * uc;
    }
```

一番重い処理がマルチスレッドのネックになるのでフローの最初に持ってくる

```
    else if (tid == 88) {
        v = -uc;
    }
    else if (tid < 95) {
        v = -fv[tid - 89] * uc;
    }

    // shared memoryに格納
    s[tid] = v;

    // スレッド数=96を利用してreduction操作をinline展開する
    // 項数が64以下のときは、
    // 項数の半分がwarpSize(=32)以下になり同期__syncthreads()が不要になり少し速くなる
    __syncthreads(); // 最初は必要
    if (tid < 48) {s[tid] += s[tid + 48]; __syncthreads();}
    if (tid < 24)  s[tid] += s[tid + 24];
    if (tid < 12)  s[tid] += s[tid + 12];
    if (tid <  6)  s[tid] += s[tid +  6];
    if (tid <  3)  s[tid] += s[tid +  3];
    if (tid == 0) {
        fval[bid] = s[0] + s[1] + s[2];
    }
}

// 特徴ベクトルのインデックス
__constant__
const char idx[] = {
// 空(16個)
   -2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
// 空 歩 香 桂 銀 金 角 飛 玉 と 杏 圭 全 金 馬 龍
    0, -1,  0,  1,  2,  3,  4,  5, -2,  6,  7,  8,  9, -2, 10, 11, // 先手
    0, -1,  0,  1,  2,  3,  4,  5, -2,  6,  7,  8,  9, -2, 10, 11  // 後手
};
```

表5 CUDA版特性関数部分和計算コード

```
// 部分和の計算
__global__
void sumf(int nleaf, int ncore, float *fval, int *adr0_db, int *sente_db, float *ary)
{
    extern __shared__ float s[];

    int id = threadIdx.x + (blockIdx.x * blockDim.x);

    s[threadIdx.x] = (id < nleaf)
     ? sigmoid_gpu(sente_db[id] ?
              (fval[id] - fval[adr0_db[id]]) : -(fval[id] - fval[adr0_db[id]]))
     : 0.0f;

    reduction_sum(threadIdx.x, ncore, s, &ary[blockIdx.x]);
}

// sigmoid関数(GPU)
__device__
float sigmoid_gpu(float x)
{
    return 1.0f / (1.0f + expf(-3.0f * x));
}
```

コンパイル・実行

添付ソースコードのコンパイル法は以下の通りである。

```
> cl -c -Ox *.c
> nvcc.exe -O2 -o a.exe *.obj main.cu iter.cu jcalc.cu
```

ただし、`Toolkit\bin`が環境変数`PATH`に、`Toolkit\include`と`SDK\common\inc`が環境変数`INCLUDE`に、`Toolkit\lib`と`SDK\common\lib`が環境変数`LIB`に含まれていることが必要である。

プログラムの実行は以下の通りである。

```
> a.exe gpu        (GPUで実行するとき)
> a.exe cpu        (CPUで実行するとき)
```

計算が正常に行われると、以下のように表示される。3項目以降は、香～龍の価値である。

```
(0,100,1) (40,60,1) 126417 22[MB]
  0 0.376548 3.00 3.00 3.00 3.00 3.00 3.00 3.00 3.00 3.00 3.00 3.00 3.00
  1 0.375239 2.70 3.30 3.30 2.70 2.70 2.70 3.30 3.30 2.70 2.70 3.30 3.30
  2 0.374835 2.41 3.59 3.59 2.41 2.41 2.41 3.59 3.59 2.41 2.41 3.59 3.59
  3 0.374846 2.13 3.87 3.87 2.13 2.13 2.13 3.87 3.87 2.13 2.13 3.87 3.87
  4 0.374677 2.40 4.15 3.60 1.85 2.40 2.40 4.15 4.15 1.85 2.40 3.60 3.60
  5 0.374675 2.67 4.41 3.33 2.12 2.13 2.13 4.41 4.41 1.59 2.13 3.33 3.87
  6 0.374624 2.41 4.67 3.59 2.38 2.39 2.39 4.67 4.67 1.84 1.88 3.59 3.61
  7 0.374699 2.66 4.42 3.34 2.13 2.14 2.14 4.92 4.42 2.09 2.13 3.84 3.86
  8 0.374605 2.42 4.66 3.10 2.37 2.39 2.39 5.16 4.66 1.85 1.88 3.60 3.61
  9 0.374605 2.65 4.90 2.86 2.13 2.15 2.62 4.93 4.43 2.09 1.65 3.83 3.85
 10 0.374602 2.88 4.67 2.64 2.36 2.38 2.39 5.16 4.66 2.32 1.88 3.61 4.08
time[sec] = 5.023
```

表6 CUDA版の計算時間

ハードウェア	計算時間[秒]	速度比
CPU	1456.6	1.00
GPU	125.0	11.66

●動作環境:
GPU : NVIDIA GeForce GTX 260 (216コア,896MB)
CPU : Xeon E5430 (2.66GHz)
OS : Windows Vista 64ビット
CUDA 2.2 (32ビット)
Visual C++ 15.0(32ビット)

●計算条件:
棋譜数=2000
手数=40-79
局面数=80000弱
評価局面数=6,315,907
反復回数=50

●考察
・GPUはCPUの約12倍高速である。
・CPU版とは対応する処理をすべてCPUで行うように書いたコードである。
計算結果の検証と計算時間の評価に用いる。
（表2の1プロセッサーとほぼ同じ計算時間である）
・評価関数の項数がさらに増えると速度比が向上すると予想される。
・CUDA2.2(64ビット)はVC++64ビットを認識しないので、CUDA2.2(32ビット)とVC++32ビットの組み合わせで開発した。

5. 計算結果

表2=表6と同じ条件で反復回数を100回に変えて計算を行い、特徴ベクトルの各要素の収束状況をプロットしたものが図1、図2である。
特徴ベクトルの初期値は図1はすべて3、図2はすべて5である。（歩の価値=1固定）

計算条件:
棋譜数=2000
手数=40-79
局面数=80000弱
評価局面数=6,315,907
反復回数=100

その他の計算条件は以下の通りである。
式(2)のa=3
式(3)のhの初期値=0.3
式(4)のΔv=0.05
hの縮小因子=0.97　　(0.97^100=0.048)

表7は特徴ベクトルの各要素の収束値である。

図1、図2と表7から、初期値が悪いと多くの反復回数が必要となるが、結果的にはほぼ同じ値に収束することがわかる。
（特徴ベクトルの次元が大きくなると、反復計算にはより工夫が必要になる）

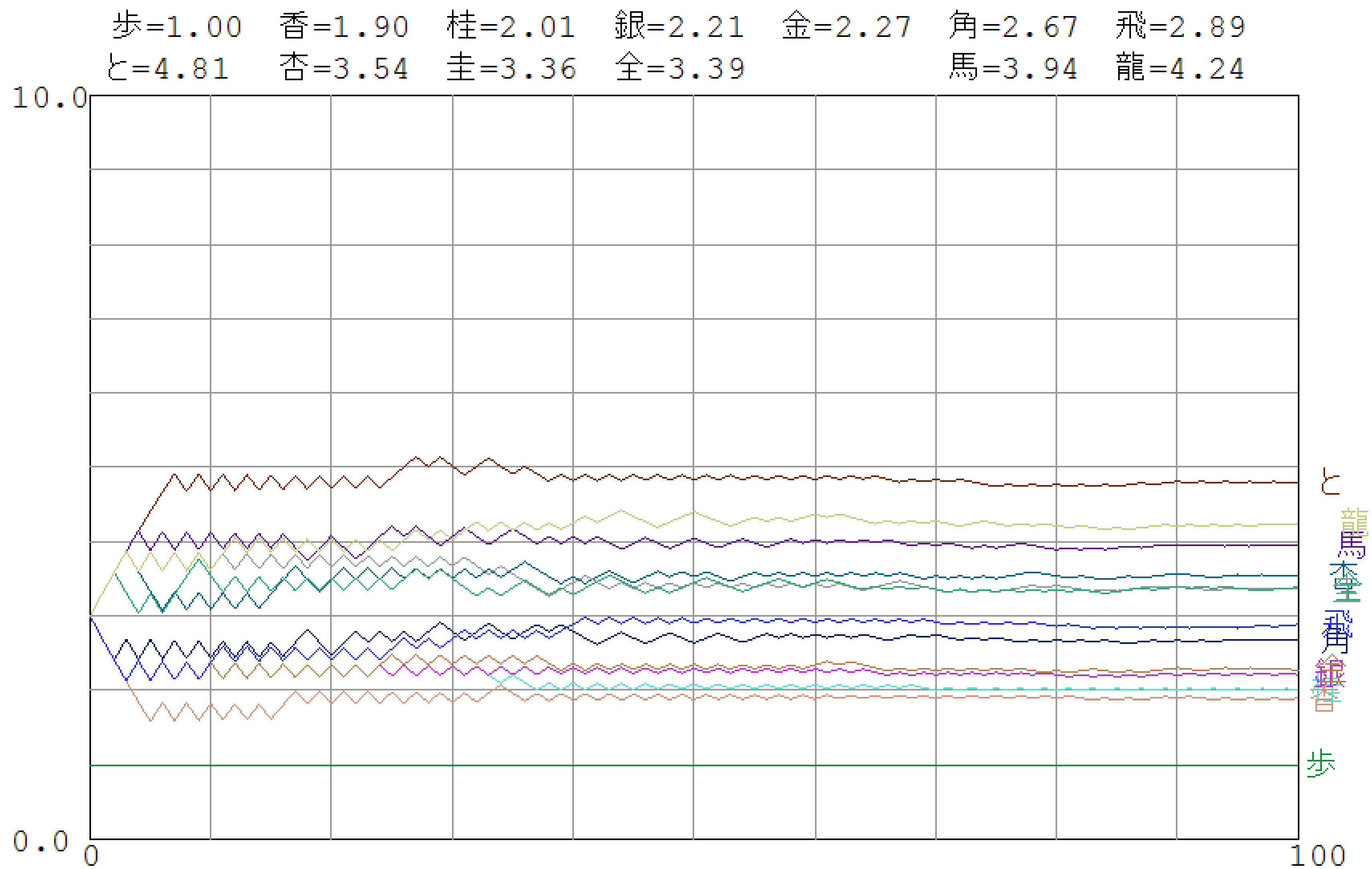

図1 特徴ベクトル要素の収束状況(初期値=3)

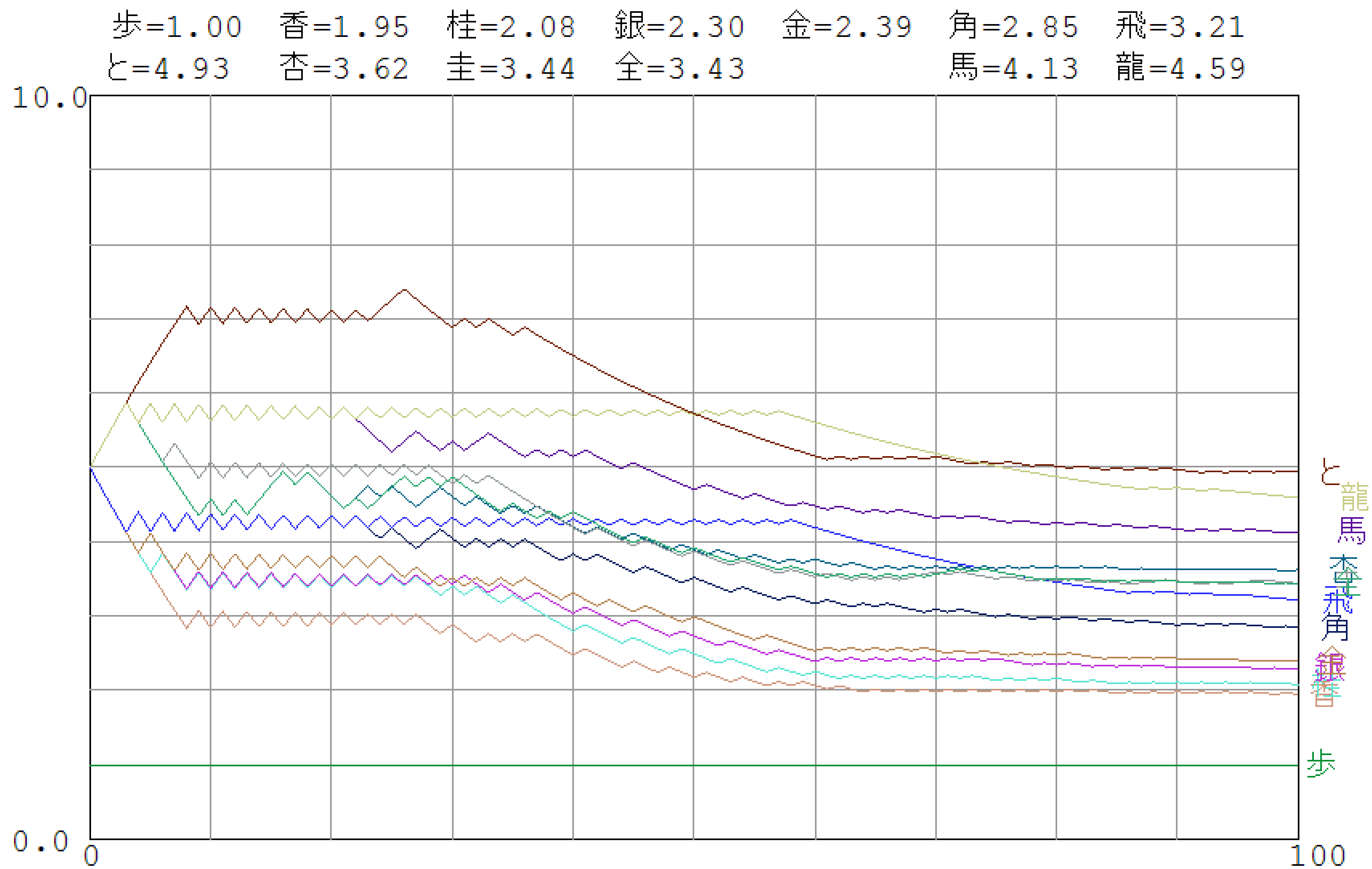

図2 特徴ベクトル要素の収束状況（初期値=5）

表7 駒の価値の収束値

駒	初期値=300	初期値=500
歩	100固定	100固定
香	190	195
桂	201	208
銀	221	230
金	227	239
角	267	285
飛	289	321
と	481	493
成香	354	362
成桂	336	344
成銀	339	343
馬	394	413
龍	424	459

●考察
・初期値が違っていてもほぼ同じ値に収束する
・“と”の価値が高い。本計算方法では、一手進めるだけで静止探索を行っていないためと思われる。実戦では、歩が成れるときは成ることが多いので、このような評価関数では、”と”の価値を実体以上に高いと考えてしまう。

6. まとめ

将棋の評価関数を学習するプログラムをMPIまたはCUDAを用いて並列化した。

MPIでは、簡単な変更でプロセッサー数にほぼ比例するプログラムができることを示した。

CUDAでは、データ構造を変えることによって、高速化できることを示した。GPU/CPUの速度比は約12倍となった。
今後の課題は以下の通りである。
・より高速化する。
・評価関数の項目を増やす。
・特徴ベクトルの次元を増やす。
・静止探索を実装する。
・より多数の局面が計算できるように局面データベースを分割する。

将棋の思考プログラムでは、局面をその都度deviceに転送する必要があるので、本方法を直ちに適用することはできない。