MPIを用いた将棋思考プログラムの並列化

2008/11/29

目次

日付	改定内容
2008/11/29	加筆訂正、文献追加
2008/11/25	初版

1. まえがき

- 目的

  ・将棋思考プログラムをMPI(Message Passing Interface)を用いて並列化する

- 手法

  ・αβ探索を並列化する

  ・探索木を管理する一つのmasterと実際に探索を行う複数のslaveから成るmaster-slave(master-worker)方式

  ・ルート局面を分割する（それ以外も可）

- 動作環境

  ・同一プログラムが共有メモリー(マルチコア、マルチCPU)と分散メモリー(PCクラスタ)で実行可能

2. セットアップ

- MPICH2ダウンロード

  以下のサイトから、”Win IA32 Binary”(32ビット)または”Win X86_64 Binary”(64ビット)をダウンロードしてインストールする(選択はすべて既定値でよい)

  http://www.mcs.anl.gov/research/projects/mpich2/

- タスクマネージャでsmpd.exe(プロセスマネージャ)が動いていることを確認する
- 環境変数の設定

  INCLUDEに”C:\Program Files\MPICH2\include”を追加

  LIBに”C:\Program Files\MPICH2\lib”を追加

  PATHに”C:\Program Files\MPICH2\bin”を追加

- 開発言語：C

3. プログラムの説明

(1)全般

・プログラム形式はSPMD(同一プログラムでプロセッサー毎に異なる処理を行う)
・並列版と逐次版を同一コードで記述すると、プログラムの管理が楽になる。
・並列版にはコンパイルオプション/DMPIをつけ、並列関係の処理を
#ifdef MPI ～ #endif　内に記述する。
・プロセッサー数(=Nproc)と自分のプロセッサー番号(=Mproc)は最初に決定したらプログラムを通して変わらず、各所で呼び出すのでグローバル変数にしたほうが便利。
・自分が何者であるか(master/slaveの別)は自分のプロセッサー番号から判断する。Mproc=0ならmaster、Mproc>0ならslave。
・探索関係の変数(局面データなど)はグローバル変数ではなく、引数で渡してデータの流れを常時監視した方が便利。
・MPIの関数(MPI_で始まる)は一つのファイル(以下ではcomm.c)にまとめると管理しやすい。この中だけでmpi.hをincludeする。
・使用するMPI関数は必須関数(MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Finalize)以外はMPI_Send(ブロッキング送信),MPI_Recv(ブロッキング受信)のみで実装できる。
・探索中の通信はmasterと個々のslave間だけで行い、slave同士は通信しない。探索量が予め予測できないので、全プロセッサーの同期が必要な集団通信は使用しない。
・コードの並列化で行う作業は、本プログラムを使用すればαβ探索部の形式的な変更で済み、評価関数計算部の変更は必要ない。

(2)masterとslaveの処理内容

masterは送信前にflag(1個の整数)を送信し、次に送るデータの意味を知らせる(受信側が通信データのサイズを予め知る必要があるため)。
その後、masterとslaveは以下の処理を行う。

flag=0:
- master:プログラムを終了させる
- slave:メインプログラムに戻り、プログラムを終了させる

flag=1:
- master:ルート局面を送信する
- slave:受信後、思考前に必要な処理を行い、局面または指し手を待つ

flag=2:(ルートで分割する場合は省略可)
- master:指し手を分割する親局面を送信する
- slave:受信後、指し手を待つ

flag=3:
- master:手が空いているslave(=探索結果を返してきたslave)に次の指し手と探索条件(深さ、αβ値等)を送信し、いずれかのslaveからの受信(探索結果)を待つ
- slave:受信した指し手を進めた子局面の探索を行い、探索結果(評価値等)をmasterに送信する(CPU時間のほとんどはこの間に消費される)

(3)制限時間の管理

制限時間の管理はmasterまたはslaveのどちらで行うことも可能

・masterで行う場合：
masterは指し手を送る前に常に制限時間をチェックし、制限時間が来たら、全slaveに探索終了を知らせる、slaveは現在の手の探索を終えた時点で探索終了を知る。実装は比較的容易だが時間ラグが発生する。

・slaveで行う場合：
初めに親局面を送信するときに、制限時間も合わせて送信する。
slaveは探索中に定期的に制限時間をチェックし、制限時間が来たら探索を終了する。実装はやや面倒だが時間ラグが少ない。

(4)コア数とプロセス数

・masterはCPUをほとんど消費しないので、全コア数をNcoreとすると、Ncore+1個のプロセスを起動する。slaveの数がNcoreに等しい。

(5)プロセッサー番号とその意味

プロセッサー番号	0	1以上
MPIでの呼び方	ルートプロセッサー	特になし (非ルートプロセッサー)
MPI-1での入出力機能	入出力可	入出力不可
本プログラムでの役割	master	slave

※ルートプロセッサーと探索のルート局面は無関係であることに注意

表1 メインプログラム(spara.c)

```
#include "spara.h"

int main(int argc, char **argv)
{
    // プロセッサー数と自分のランクを逐次版の既定値で初期化しておく
    Nproc = 1;
    Mproc = 0;

    // 並列版のプロセッサー数と自分のランクを取得する(逐次版では何もしない)
    mpi_init(argc, argv, &Nproc, &Mproc);

    // 準備作業はここで行う(全プロセッサー共通)
    //setup();

    // 外部ファイル(定跡等)を読み込むときはここで行う
    // ルートで読み込み、全プロセッサーにMPI_Bcastでbroadcastする
    //if (!Mproc) read_db();
    //broadcast_db();

    // 本処理
    if (!Mproc) {
            // 逐次版または並列版のルート(=master):入出力処理に入る
            io_usi();
    }
    else {
            // 並列版のslave:探索に入る
            slave();
    }

    // 並列版終了処理(逐次版では何もしない)
    mpi_close();

    return 0;
}
```

表2 思考開始関数(sikou.c)

```
// 思考開始
// 逐次版または並列版のmaster
int sikou(int ban[], int dai[], int sente)
{
    int proc;
    int depth, depth_rest, alpha, beta;
    int value;

    // すべてのslaveに思考開始を送信(逐次版では何もしない)
    for (proc = 1; proc < Nproc; proc++) {
        // 思考開始flag(=1)送信
        send_flag(proc, 1);
        // 局面送信
        send_kyokumen(proc, ban, dai, sente);
        // 必要ならここで思考条件(制限時間等)を送信する
        //send_control(proc, TODO);
    }

    // 思考前の準備(各変数の初期化等)
    setup_sikou();

    // 思考開始:αβ探索(通常は反復深化のループになる)
    depth = 0;
    depth_rest = Depth;  // 探索の深さ
    alpha = -INF;
    beta  = +INF;
    value = alpha_beta(depth, depth_rest, alpha, beta, ban, dai, sente)
          * (sente ? +1 : -1);

    return value;
}
```

表3(1) αβ法(alpha_beta.c)

```
// alpha_beta.c
#include "spara.h"

// αβ法
// 関数値    : 評価関数値
// depth      : 現在の深さ(ルートは0)
// depth_rest : 残り深さ(末端は0)
int alpha_beta(int depth, int depth_rest, int alpha, int beta,
    int ban[], int dai[], int sente)
{
    // 末端:評価関数を計算して戻る
    if (depth_rest == 0) {
        return evaluate(ban, dai) * (sente ? +1 : -1);
    }

    // 指し手生成
    tenum = generate_moves(te, ban, dai, sente);

    // 探索順初期化
    // 探索順をソートするときは変数inumを適当に設定する(注意)
    for (i = 0; i < tenum; i++) {
        inum[i] = i;
    }
```

表3(2)

```
    // 並列版のmaster、ここを変更すると分割する深さを変えることができる(注意)
    if ((Nproc > 1) && !Mproc && (depth == 0)) {
        return master(tenum, te, inum, ban, dai, sente,
            depth, depth_rest, alpha, beta);
    }

    // 逐次版または並列版のslave
    else {
        // 指し手に関するループ
        for (i = 0; i < tenum; i++) {
            // 一手進める
            make_move(ban, dai, &sente, te[inum[i]]);
            // 次の深さに進む
            value = -alpha_beta(depth + 1, depth_rest - 1, -beta, -alpha,
                ban, dai, sente);
            // 一手戻す
            unmake_move(ban, dai, &sente, te[inum[i]]);
            // β-cut
            if (value >= beta) {
                return beta;
            }
            // 評価関数の評価
            if (value > alpha) {
                // α更新
                alpha = value;
                // 最善手更新、info出力(ルート)等、エンジンに合わせて適用に編集する(注意)
            }
        }
    }

    return alpha;
}
```

表4(1) master関数(master.c)

```
// master.c
// 並列版のmaster
// αβ探索、MPI関数は直接呼ばない、複雑な処理はすべて本関数に集約されている
int master(int tenum, int tebuf[][5], int inum[], int ban[], int dai[], int sente,
           int depth, int depth_rest, int alpha, int beta)
{
    int done[MAXTE];      // 手番号が計算を終えたか(N/Y=0/1)
    int numdone;          // 処理を終えたプロセッサー数
    int beta_cut = 0;          // β-cutが発生済みか(N/Y=0/1)

    // 終了プロセッサー数初期化
    numdone = 0;
    // 終了flag初期化
    for (i = 0; i < tenum; i++) {
        done[i] = 0;
    }
    // 現在の局面を全slaveに送信する
    for (proc = 1; proc < Nproc; proc++) {
        // flag送信
        send_flag(proc, 2);
        // データ送信
        send_kyokumen(proc, ban, dai, sente);
    }
    // 最初に読む手と現在の探索条件を送信する
    // (手の数はプロセッサー数より大きいことを仮定している)
    for (proc = 1; proc < Nproc; proc++) {
        // flag送信
        send_flag(proc, 3);
        // データ送信
        m = inum[proc - 1];       // 指し手番号(注意)
        send_te(proc, tebuf[m], depth, depth_rest, alpha, beta, Nnode);
        // 計算済みflagを立てる
        done[m] = 1;
    }
```

表4(2)

```
// すべてのプロセッサーが探索を終えるまで続ける
while (numdone < Nproc - 1) {

    // slaveから探索結果を受信する、プロセッサー番号(src)は返信に使うので保存しておく
    src = recv_value(&value, te, &node);

    // β-cut済みのとき
    if (beta_cut) {
        // 終了済みプロセッサー数++
        numdone++;
        // 次のslaveからの受信を待つ
        continue;
    }

    // β-cut
    if (value >= beta) {
        // β-cut済みflagを立てる
        beta_cut = 1;
        // 終了済みプロセッサー数++
        numdone++;
        // 次のslaveからの受信を待つ
        continue;
    }

    // 評価関数の評価
    if (value > alpha) {
        // α更新
        alpha = value;
        // 最善手更新、info出力(ルート)等、エンジンに合わせて適用に編集する(注意)
    }
```

表4(3)

```
        // 次の未計算の手番号を取得する(すべて計算したときは-1)
        m = -1;
        for (i = 0; i < tenum; i++) {
            if (!done[inum[i]]) {
                m = inum[i];
                break;
            }
        }

        // まだ手が残っている:送信してきたプロセッサーに次の手を送信する
        if (m >= 0) {
            // flag送信
            send_flag(src, 3);
            // データ送信
            send_te(src, tebuf[m], depth, depth_rest, alpha, beta, Nnode);
            // 計算済みflagを立てる
            done[m] = 1;
        }
        // すべての手が計算済み:送信してきたプロセッサーには何も送らず、終了済みプロセッサー数++
        else {
            numdone++;
        }
    }


    return beta_cut ? beta : alpha;
}
```

表5 slave関数(slave.c)

```
// 並列版のslave, MPI関数は直接呼ばない
void slave(void)
{
    do {
        // flag受信
        flag = recv_flag();
        // flagで処理を分類
        if (flag == 1) {
            // ルート局面を受信
            recv_kyokumen(ban, dai, &sente);
            // 必要ならここで思考条件(制限時間等)を受信する、sikou.c参考
            //recv_control(TODO);
            // 思考前の処理
            setup_sikou();
        }
        else if (flag == 2) {
            // 現在の局面を受信
            recv_kyokumen(ban, dai, &sente);
        }
        else if (flag == 3) {
            // 指し手と探索条件を受信
            recv_te(te, &depth, &depth_rest, &alpha, &beta, &node);
            // 受信した手で局面を進める
            make_move(ban, dai, &sente, te);
            // 探索を実行する
            value = -alpha_beta(depth + 1, depth_rest - 1, -beta, -alpha,
                ban, dai, sente);
            // 局面を戻す
            unmake_move(ban, dai, &sente, te);
            // 探索結果を送信する
            send_value(value, te, (Nnode - node));
        }
    } while (flag);       // flag = 0 で終了
}
```

表6(1) MPI関数(comm.c)

```
// comm.c
// 並列版の通信ルーチン、MPI関数はすべてここに集める
// master : プロセッサー番号=0(ルート)
// slave : プロセッサー番号=1,2...
#ifdef MPI
#include "mpi.h"
#endif

// MPI初期化
void mpi_init(int argc, char **argv, int *nproc, int *mproc)
{
#ifdef MPI
    // 引数
    MPI_Init(&argc, &argv);
    // プロセッサー数とランク番号
    MPI_Comm_size(MPI_COMM_WORLD, nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, mproc);
#endif
}

// MPI終了
void mpi_close(void)
{
#ifdef MPI
    int proc;
    // masterからすべてのslaveに終了フラグ(=0)を送信する
    if (!Mproc) {
        for (proc = 1; proc < Nproc; proc++) {
            send_flag(proc, 0);
        }
    }
    // 終了処理
    MPI_Finalize();
#endif
}
```

表6(2)

```
// masterがslaveに送信する
// データ:flag(1個の整数)
void send_flag(int dst, int flag)
{
#ifdef MPI
    MPI_Send(&flag, 1, MPI_INT, dst, 0, MPI_COMM_WORLD);
#endif
}

// slaveがmasterから受信する
// データ:flag(1個の整数)
int recv_flag(void)
{
#ifdef MPI
    int flag;
    MPI_Status status;

    MPI_Recv(&flag, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    return flag;
#endif
}
```

表6(3)

```
// masterがslaveに送信する
// データ:現在の局面(引数はエンジンに応じて修正する)
void send_kyokumen(int dst, int ban[], int dai[], int sente)
{
#ifdef MPI
     int i, id;
     int send[BANSIZE + KOMASIZE + 1];
     id = 0;
     for (i = 0; i < BANSIZE; i++) {
          send[id++] = ban[i];
     }
     for (i = 0; i < KOMASIZE; i++) {
          send[id++] = dai[i];
     }
     send[id++] = sente;
     MPI_Send(send, (BANSIZE + KOMASIZE + 1), MPI_INT, dst, 0, MPI_COMM_WORLD);
#endif
}

// slaveがmasterから受信する
// データ:send_kyokumenと同じ
void recv_kyokumen(int ban[], int dai[], int *sente)
{
#ifdef MPI
     int i, id;
     int recv[BANSIZE + KOMASIZE + 1];
     MPI_Status status;
     MPI_Recv(recv, (BANSIZE + KOMASIZE + 1), MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
     id = 0;
     for (i = 0; i < BANSIZE; i++) {
          ban[i] = recv[id++];
     }
     for (i = 0; i < KOMASIZE; i++) {
          dai[i] = recv[id++];
     }
     *sente = recv[id++];
#endif
}
```

表6(4)

```
// masterがslaveに送信する
// データ:手、現在の深さ、残り深さ、α、β、ノード数(debug用)(引数はエンジンに応じて修正する)
void send_te(int dst, int te[], int depth, int depth_rest, int alpha, int beta, int node)
{
#ifdef MPI
     int i, id;
     int send[10];

     id = 0;
     for (i = 0; i < 5; i++) {
          send[id++] = te[i];
     }
     send[id++] = depth;
     send[id++] = depth_rest;
     send[id++] = alpha;
     send[id++] = beta;
     send[id++] = node;
     MPI_Send(send, 10, MPI_INT, dst, 0, MPI_COMM_WORLD);
#endif
}

// slaveがmasterから受信する
// データ:send_teと同じ
void recv_te(int te[], int *depth, int *depth_rest, int *alpha, int *beta, int *node)
{
#ifdef MPI
     int i, id;
     int recv[10];
     MPI_Status status;

     MPI_Recv(recv, 10, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
     id = 0;
     for (i = 0; i < 5; i++) {
          te[i] = recv[id++];
     }
     *depth      = recv[id++];
     *depth_rest = recv[id++];
     *alpha      = recv[id++];
     *beta       = recv[id++];
     *node       = recv[id++];
#endif
}
```

表6(5)

```
// slaveがmasterに送信する
// データ:評価関数値、指し手、ノード数増分(debug用)(引数はエンジンに応じて修正する)
void send_value(int value, int te[], int node)
{
#ifdef MPI
     int i, id;
     int send[7];

     id = 0;
     send[id++] = value;
     send[id++] = node;
     for (i = 0; i < 5; i++) {
          send[id++] = te[i];
     }
     MPI_Send(send, 7, MPI_INT, 0, 0, MPI_COMM_WORLD);
#endif
}

// masterがslaveから受信する
// データ:send_valueと同じ
// 返り値:slave番号
int recv_value(int *value, int te[], int *node)
{
#ifdef MPI
     int i, id;
     int recv[7];
     MPI_Status status;

     MPI_Recv(recv, 7, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
     id = 0;
     *value = recv[id++];
     *node  = recv[id++];
     for (i = 0; i < 5; i++) {
          te[i] = recv[id++];
     }

     return status.MPI_SOURCE;
#endif
}
```

4. コンパイルと実行

(1) コンパイル・リンク

○並列版
```
> cl /DMPI /Ox /Fespara.exe *.c mpi.lib
```

○逐次版
```
> cl /Ox /Fespara.exe *.c
```

実行プログラムspara.exeができる

(2) 実行

○並列版
```
> mpiexec -n 5 spara.exe  (4コア・共有メモリーのとき)
```

○逐次版
```
> spara.exe
```

(注1) 並列版と逐次版の指し手は一般に一致するが、評価値が同じ手が複数あるときは不定になる。
(注2) USI対応のGUIとして「将棋所」「プチ将棋」などがあるが、上のように並列版の実行には引数が必要になり、現在、引数に対応しているのは「プチ将棋」のみ。
(注3) MPICH2はTCP/IPの通信 (ポート番号8676) を使うので、最初に実行するとき、ファイアウォールの警告が出るので許可する。
(注4) ログオン後、最初に実行するとき、アカウント (通常、単にEnterでよい) とログオンパスワードの入力が必要になる。 (GUIから起動する前に本作業を行っておくこと)

5. 添付プログラムの仕様

添付プログラムはMPIによる並列化を説明するためのもので、対局プログラムとして必要最低限の機能しか持っていない。
その仕様は以下の通り。

・MPIによる並列化
・USIプロトコル対応、標準入出力
・αβ探索
・全幅探索
・評価関数は駒割のみ
・定跡なし
・探索深さを指定する(深さ固定)
・ノード数を出力する(デバッグ用)
・CSA形式の局面を読み込む(USI拡張、デバッグ用)
・ソースコードサイズ：約2000行

6. 計算結果

〇計算内容:
プロの棋譜を4局取り上げ、並列計算のコア数と計算時間・ノード数の関係を調べる。ここで、ノード数は訪れた節点(βカットされなかった節点)の数とする。

〇プログラム:
前章の機能に加えて、評価関数の精密化といくつかの高速化技術を用いている。

〇計算機環境:
・CPU : Intel Xeon E5430 (4コア) 2.66GHz
・OS : Windows Vista (64ビット)
・コンパイラ : Microsoft Visual C++ 15.0 (64ビット)
・MPICH2 : 1.0.6p1 (64ビット)

〇計算条件:
特に断らない限り以下の条件で計算を行った。
・探索深さ=9
・手数40手目から90手目までの51局面を探索
・4コア(従ってプロセス数=5)

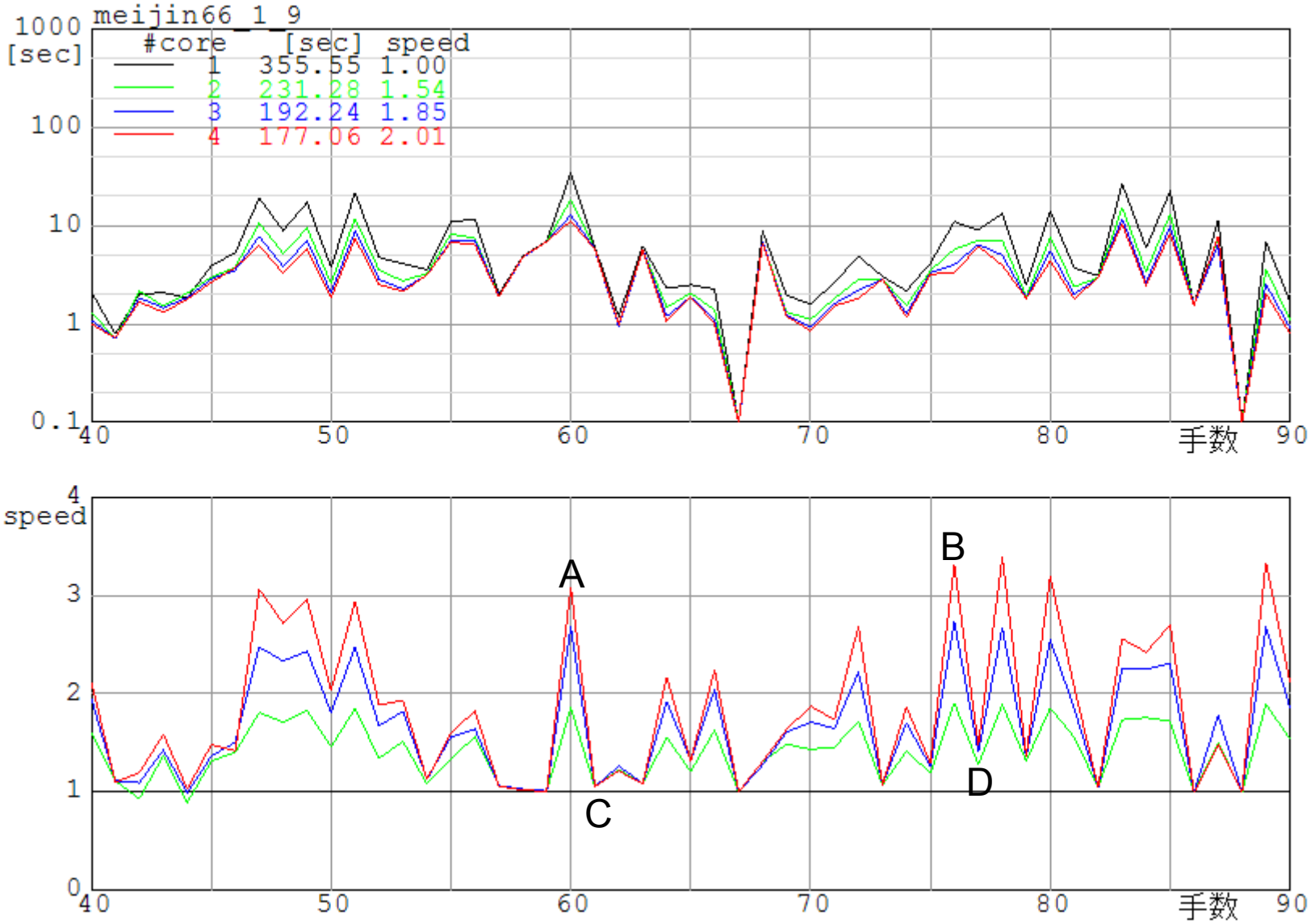


図1 計算時間と速度比(66期名人戦第1局, 2008/04/08-09)

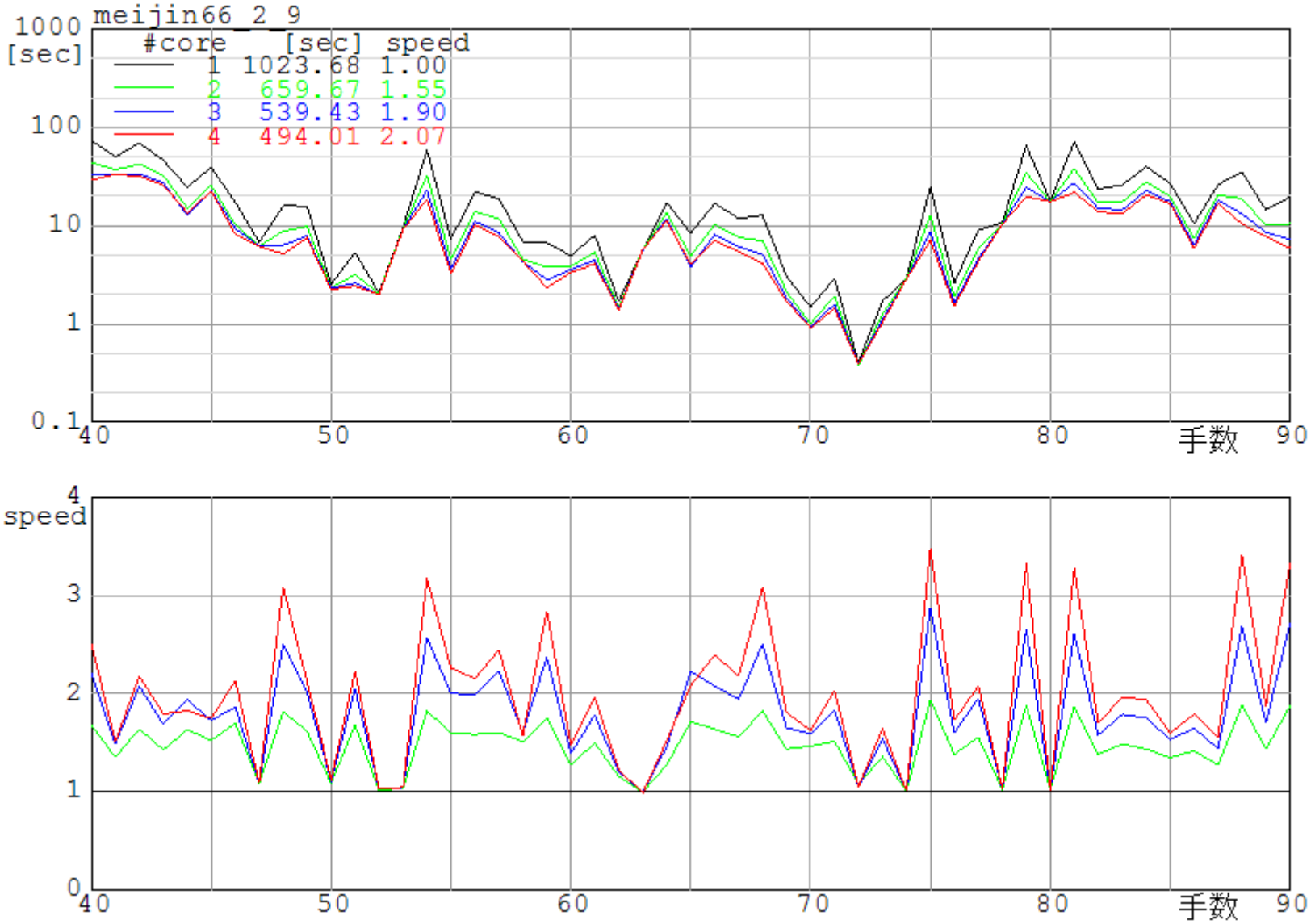


図2 計算時間と速度比(66期名人戦第2局, 2008/04/22-23)

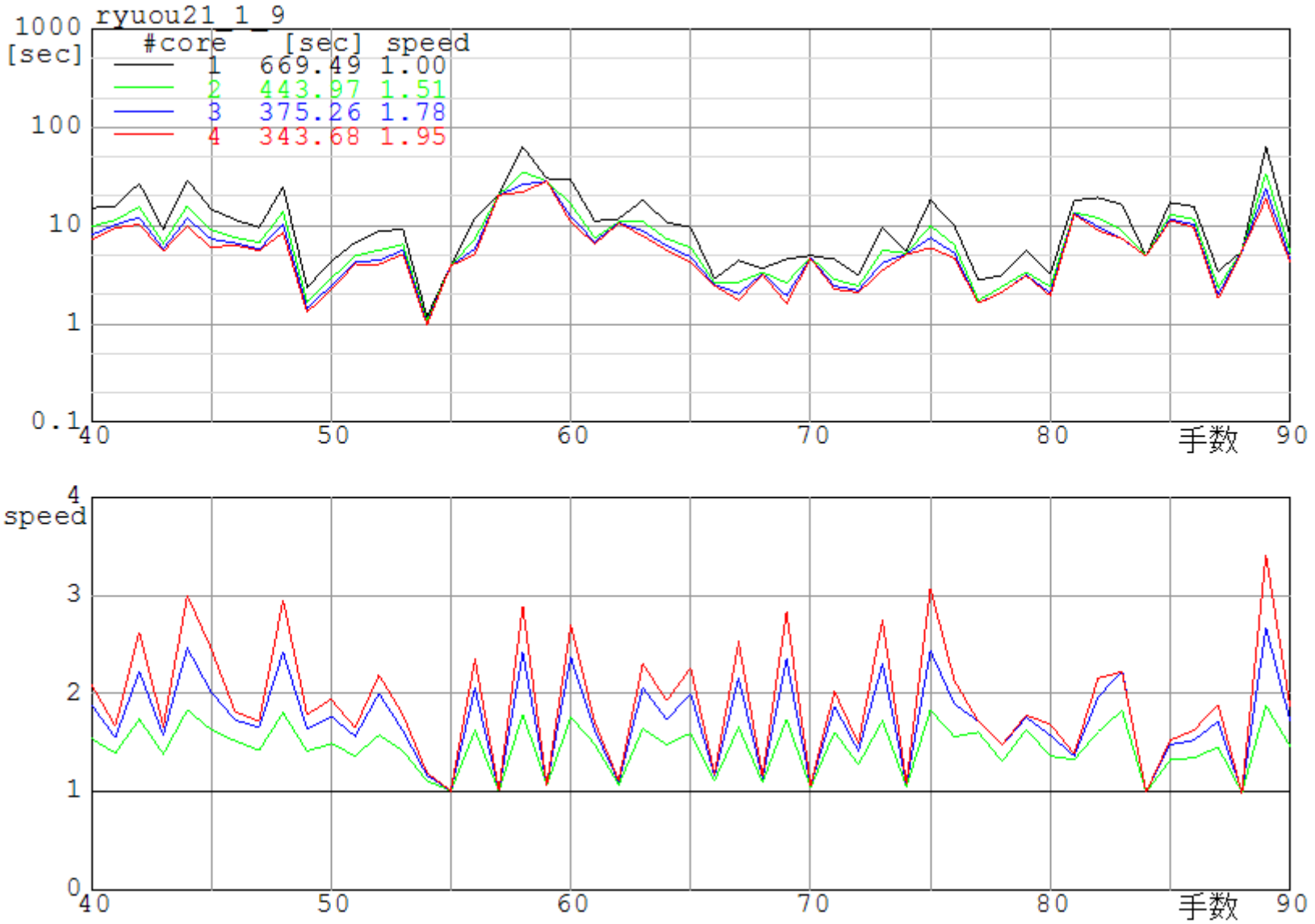


図3 計算時間と速度比(21期竜王戦第1局, 2008/10/18-19)

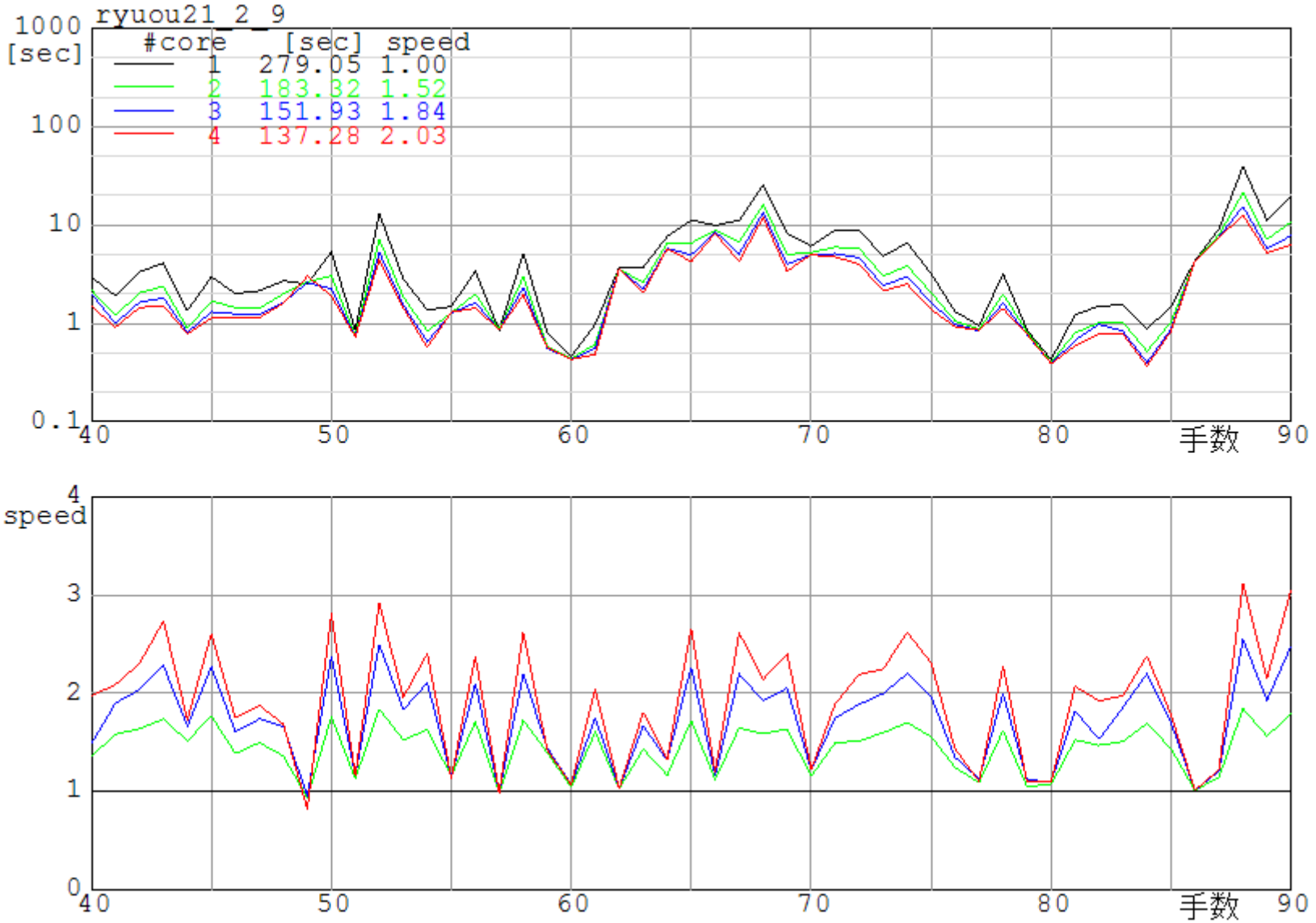
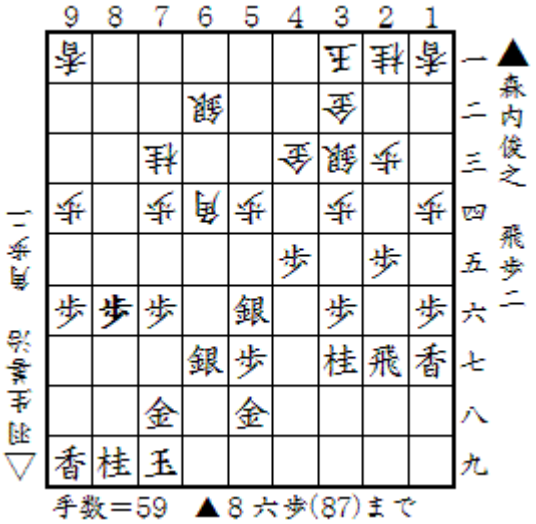


図4 計算時間と速度比(21期竜王戦第2局, 2008/10/30-31)

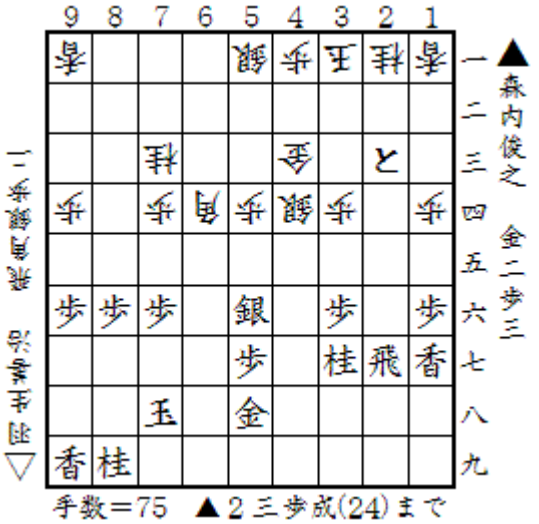
表7 計算時間と速度比のまとめ
(数字は51局面の計算時間の和(単位:秒)、()内は1コアとの比)

コア数	図1	図2	図3	図4	合計
1	355(1.00)	1023(1.00)	669(1.00)	279(1.00)	2326(1.00)
2	231(1.54)	659(1.55)	443(1.51)	183(1.52)	1516(1.53)
3	192(1.85)	539(1.90)	375(1.78)	151(1.84)	1257(1.85)
4	177(2.01)	494(2.07)	343(1.95)	137(2.03)	1151(2.02)

表7より、速度比は多くの局面での平均をとると棋譜によらずほぼ一定で、1コアのときと比べておよそ、2コアで1.53倍、3コアで1.85倍、4コアで2.02倍となる。
しかし、図1-図4からわかるように、速度比は局面によって大きく変動する。
図1における、速度比の大きい局面A,Bおよび速度比の小さい局面C,Dは図5,図6の通りである。
これより、速度比の大きい局面は候補手が多い局面、速度比の小さい局面は候補手が少ない局面であることがわかる。
前者では、探索量の多い指し手が多いために、並列計算での指し手の分割が効率よく機能するためである。
一般に、候補手が多い局面の高速化がより望ましいので、実効的な速度比は表7の結果より高いと言える。

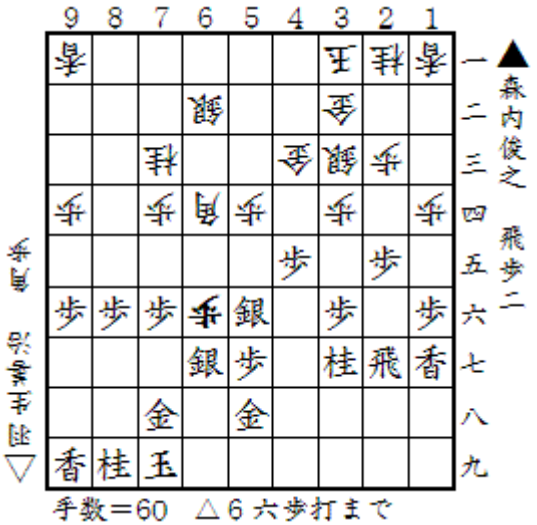


(1)図1のA(候補手：△２二玉、△３八角等)

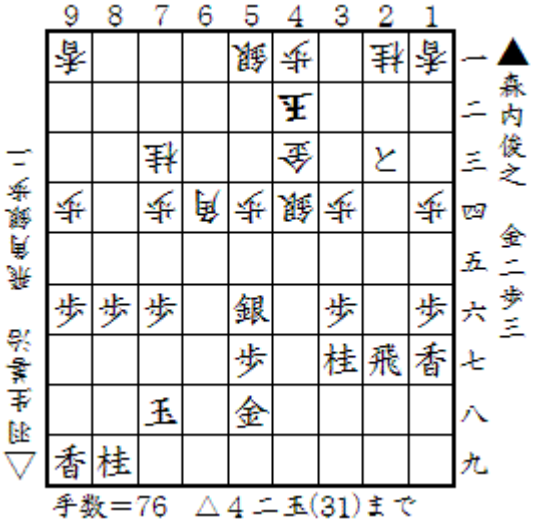


(2)図1のB(候補手：△８七銀、△４二玉等)

図5 並列化による速度比が大きい局面(候補手が多い)



(1)図1のC(候補手：▲６一飛)



(2)図1のD(候補手：▲６三金)

図6 並列化による速度比が小さい局面(候補手が少ない)

換算速度比

実装されているコア数よりもコア数が多いときの速度比を予測するには、計算時間よりもノード数に注目したほうがよい。
一般に、コア数が増えるとαβ探索の効率が低下し、探索ノード数が増える。
ここで、次式で換算速度比を定義する。

Nコアでの換算速度比 = N * (1コアでのノード数) / (Nコアでのノード数)

これは、並列計算に伴う通信関係のオーバヘッドや負荷の不均等がないと仮定したときの、速度比の上限である。

図7にコア数=1/2/4/8/16のときの探索ノード数と換算速度比を示す。棋譜は図2と同じである。図2、図7からわかるように、実際の計算時間から求めた速度比と換算速度比は同じ傾向を示す。これから、コア数が多いときの速度比の上限を換算速度比から予測することができる。

表8は図1-図4のケースに対する、コア数=2/4/8/16の換算速度比である。
コア数=2/4のときは、実速度比と換算速度比の差は小さくことがわかる。

図7と表8からわかるように、コア数が8,16と増えると探索ノード数が急に増大し、速度比の上昇が小さくなる。ただし、既に述べたように、候補手が多い局面では、速度比が大きく、図7より、16コアで6倍を超える局面も少なくない。

表8 換算速度比(()内は表7の実速度比)

コア数	図1	図2	図3	図4	平均
1	1.00	1.00	1.00	1.00	1.00
2	1.61	1.57	1.54	1.57	1.57(1.53)
4	2.31	2.20	2.11	2.22	2.21(2.02)
8	2.97	2.80	2.59	2.85	2.80
16	3.51	3.25	2.96	3.34	3.26

●課題
コア数が多いときの速度比をさらに向上させるには、並列探索アルゴリズムの改良が必要である。
特に、初手の有力候補手が少ないときは、初手を分割することは並列効率が悪い。
表3(2)のように、分割する深さを変えることが可能なので、初手の有力候補手が少ないときは分割する深さを1とすると、速度比が向上するケースがあることは確認できる。
しかし、その場合、通信回数が増えること、深さ1以上ではβカットのために枝の数が少ないこと、深さ1も有力候補手が少ない場合があるなどの理由で逆に遅くなる場合も少なくない。
結局、さらに並列探索の効率を上げるには、探索の状況に応じて動的に指し手を分割することが必要になるが、これは難しい。

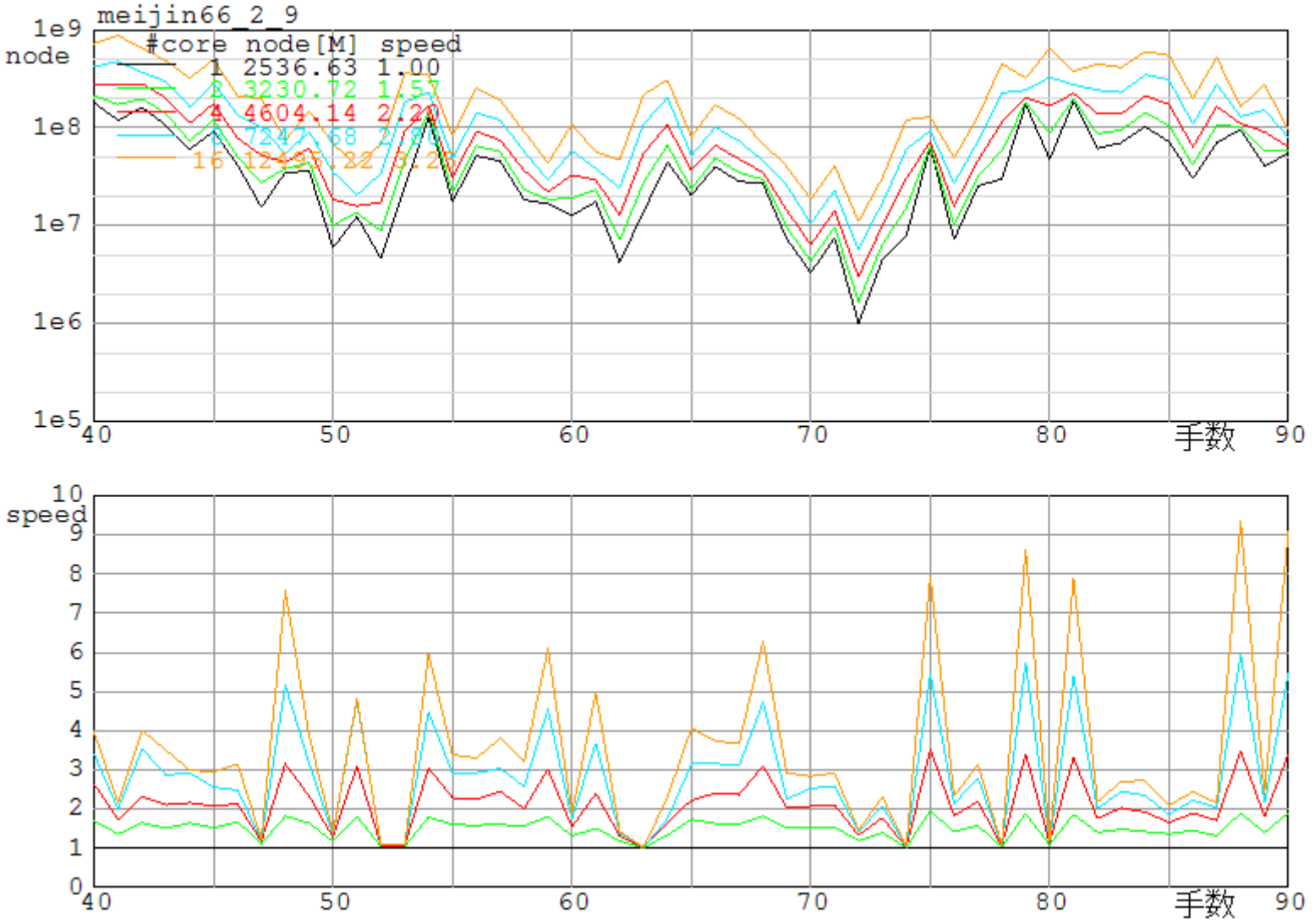


図7 探索ノード数と換算速度比(1/2/4/8/16コア、図2と同条件)

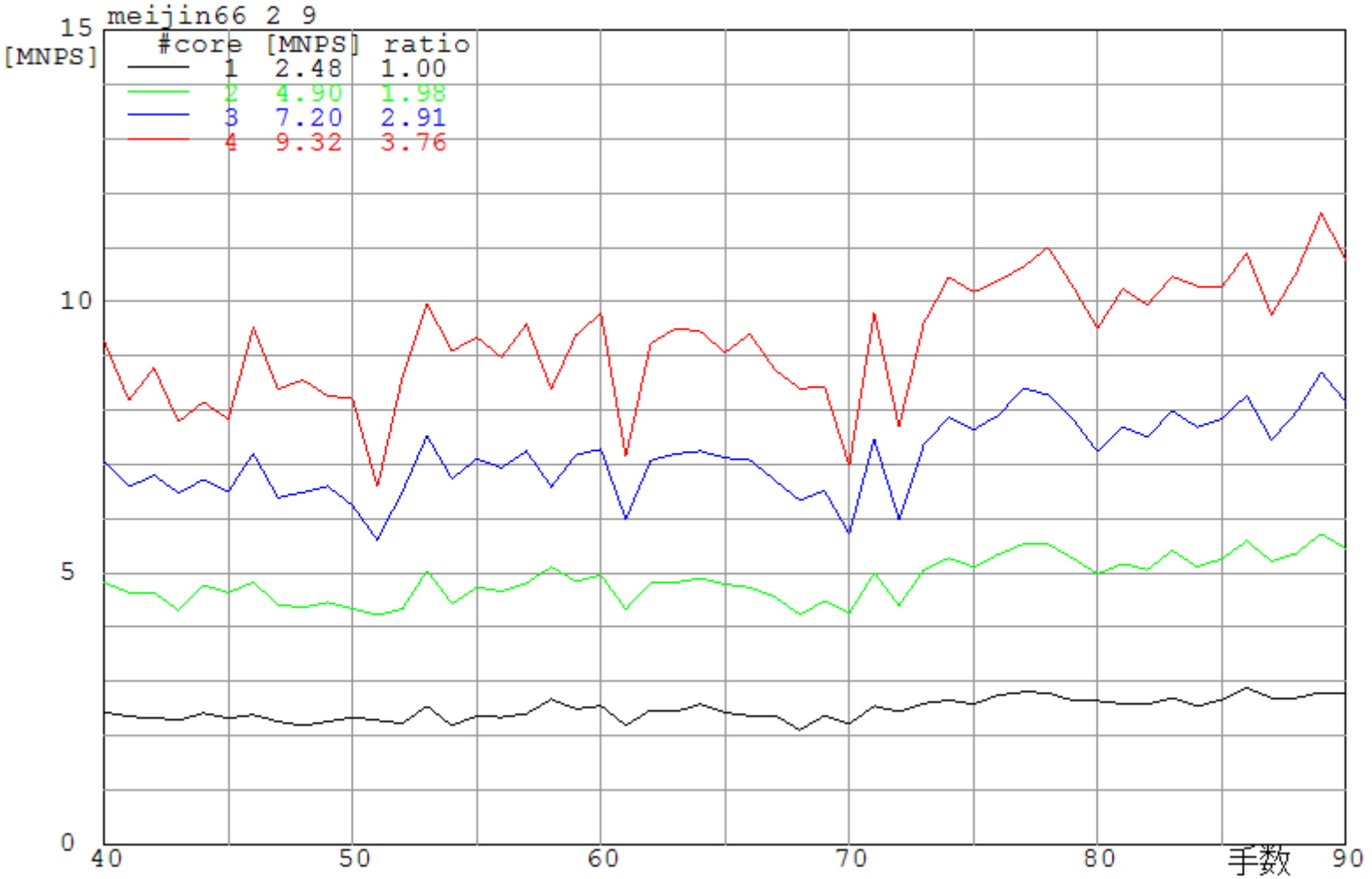


図8 NPSとコア数の関係(1/2/3/4コア、図2と同条件)

図8はNPS(1秒あたりの探索ノード数)とコア数の関係である。
NPSは局面によって大きく変化せず、1コアのとき平均2.5[MNPS]であり、コア数が増えるとほぼコア数に比例することがわかる。

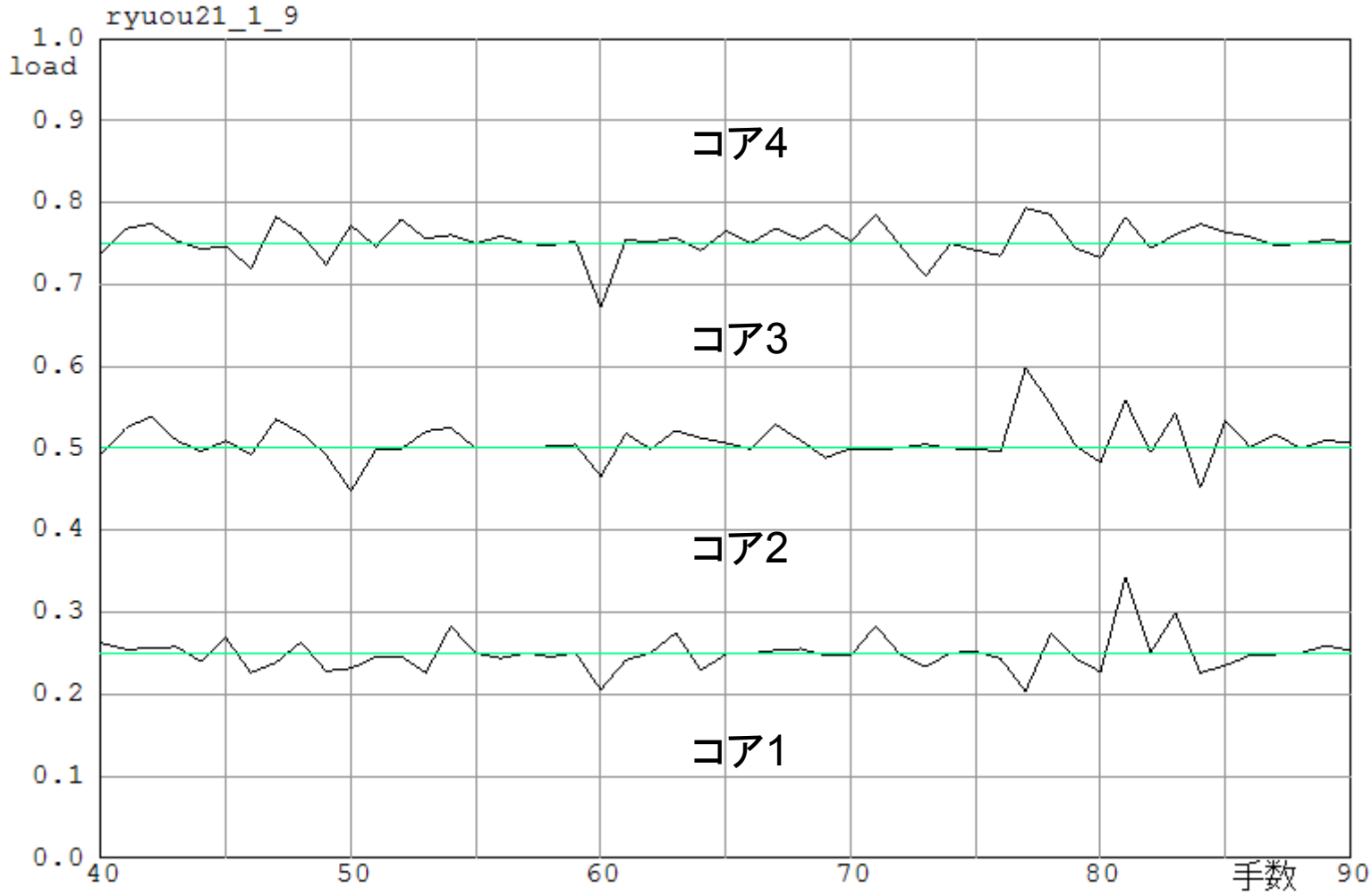


図9 各コアの負荷量(4コアのとき、図3と同条件)

図9は4コアのときの、各コアごとの探索ノード数の内訳である。
緑線はコア数で等分割したものである。
図より、各コアの探索ノード数(負荷)の不均等は小さいことがわかる。

7. 分散メモリー

MPIで並列化したプログラムはネットワークに接続された複数のPC(分散メモリー環境)で並列計算することができる。

●セットアップ:
1)複数のPCをハブを介してネットワーク接続する、ギガビットEthernetでよい。
2)[ネットワーク]で互いのコンピュータが認識できるように設定する。
3)並列版実行プログラム(ここではspara.exe)をすべてのPCの同じ絶対pathに保存する。(外部ファイル(定跡等)は転送する必要はない:ファイルの読み書きをするのはルートプロセッサーのみ)
4)MPICH2のsmpd.exeを用いて、すべてのPCでコマンド ”smpd.exe -install” を実行する。(タスクマネージャでsmpd.exeが動いていることを確認する)

●実行
並列計算を行うには以下のコマンドを実行する。
> mpiexec -hosts 2 pc1 2 pc2 1 spara.exe
(pc1(ルートPC)で2プロセス、pc2で1プロセス実行するとき)

●計算結果と考察
計算結果は表9の通りである。2台のPCで計算したときの速度比(=1.18)が低い。
通信回数は表6からわかるように一つの指し手について3回(send_flag, send_te, send_value)である。
指し手の数を平均100、通信遅延(latency)を大きく見積もって100μsecとすると[3]、通信遅延の合計は
3 * 100 * 100e-6 = 0.03sec
となり、本ケースでの平均的な思考時間10secに比べて十分小さい。
また通信するデータ量は各回数十バイト以下であり、バンド幅(数十MB/秒)に比べると十分小さい。
以上から、分散メモリーでも速度向上は十分期待できるはずであるが、そのような結果は得られなかった。原因については、さらに検討が必要である。

表9 分散メモリーでの計算時間と速度比

PC構成	総計算時間[秒]	速度比
pc1X1コア	336	1.00
pc2X2コア	227	1.48
pc1X1コア+pc2X1コア	283	1.18

計算条件:
・図1と同じ棋譜、40-90までの51局面
・pc1 : Xeon E5430 2.66GHz (4コア)
・pc2 : Core2 Duo 2.60GHz (2コア)

文献

[1]P.パチェコ(秋葉博訳)「MPI並列プログラミング」培風館、2001
[2]青山幸也「並列プログラミング入門MPI版」 http://accc.riken.jp/HPC/training/text.html
[3]石川裕、他「Linuxで並列処理をしよう 第2版」共立出版、2007
[4]William Gropp, et.al “Using MPI, Portable Parallel Programming With the Message-Passing Interface, 2nd ed.”, MIT Press, 1999.
[5]ウィリアム・グロップ他(畑崎隆雄訳)「実践MPI-2、メッセージパッシング・インタフェースの上級者向け機能」ピアソン・エデュケーション、2002