ICOT Technical Memorandum: TM-0658

TM-0658

マルチPSIにおける並列詰め碁プログラムの実現と評価

沖　廣明(未来技研)，瀧　和男，
清　慎一，古市昌一(一蓮)

December, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

マルチ PSI における並列詰め碁プログラムの実現と評価

沖 廣明
(株) 未来技術研究所
瀧 和男, 清 愼一
(財) 新世代コンピュータ技術開発機構
古市昌一
三菱電機 (株) 情報電子研究所

概要： 詰め碁を解くプログラムを並列コンピュータマルチ PSI 上に論理型言語 KL1 を使って試作した。詰め碁の解探索には、基本的にαβ枝刈り法を用いている。この方法が本来高い逐次性を含んでいるのに対して、並列性を効果的に引き出すことができるよう、いくつかの工夫を試みた。1つはアルゴリズムの見直しであり、もう1つは負荷分散方式の検討である。負荷分散方式については2つの方式を検討した。1つはコンパイル段階で割り振りのアルゴリズムを固定し、実行時の負荷情報は利用しない準静的な負荷分散であり、もう1つは実行時に暇なプロセッサを検出して割り振る、いわゆる動的負荷分散である。
今回はプログラム構造、負荷分散方式について述べるとともに、台数効果、プロセッサの稼働率の測定結果について報告する。

1 はじめに

ICOT の中期計画 (昭和 59 年〜昭和 63 年) のプロジェクトの1つに「棋士システム」がある。このプロジェクトは、人間と同じようにコンピュータに囲碁の対局をさせようとするもので、従来チェスなどでとられている探索一辺倒の方式でなく、知識をうまく利用する方式を目指している。

昭和 60 年にはプロトタイプ「碁世代」[1] が完成し、その後現在まで、知識の拡充や精密化など、多くの改良が続けられてきたが、思うように棋力が伸びていないのが現状である。

この原因の1つにコンピュータパワーの問題がある。「碁世代」は探索一辺倒の方式ではないが、局所的には目的毎にいくつかの探索を、知識のサブモジュールとして利用している。現在、一局 (平均 200 〜 300 手) の消費時間を数十分程度に押さえる為に、知識の量や探索の手数を制限しており、これが棋力の伸び悩みの原因となっている。

この問題を解決する1つの方法として、並列コンピュータを使って、より高速に実行させることが考えられる。今回その実験システムとして、「碁世代」の探索モジュールの1つ「詰め碁」を取り上げ、並列化を試みた。「詰め碁」を選んだ理由としては、

1. 探索は一般にコストが高く、また、「詰め碁」はその中でも特にコストが掛かる。
2. 詰め碁などのゲーム木の探索は高い逐次性を有し、どれくらい並列性が引き出せるかは、まだよくわかっていない。
3. 探索の中でも詰め碁は、短手数の問題から非常に長手数の問題まで幅広く、しかも容易に作成できるので、並列化の実験システムとして向いている。

などがあげられる。

また、この並列詰め碁プログラムは、並列論理型言語 KL1[2, 3] で記述され、並列推論マシンマルチ PSI[4] とそのオペレーティングシステム PIMOS[2] という環境の下で、実行している。

以下の各章で、本来逐次性の高いαβ枝刈り法から効果的に並列性を引き出す為の、並列アルゴリズム及び負荷分散方式について説明し、最後に試作プログラムによる実際の測定結果を示す。

2 詰め碁の定式化

詰め碁は囲碁というゲームの中で生じる部分問題の1つである。従って、詰め碁を定式化する前に囲碁というゲームのルールについて簡単にふれよう。

囲碁のルールの厳密な定義は容易ではなく、また国によって異なるルールも存在するが、原則となるルールは以下のように簡明である。

1. 交互着手
   2人のプレイヤがそれぞれ黒石と白石を持ち、図 2.1 のような 19×19 の格子点の上に石を交互に置いていく（ただし、パスすることによって、石を置かずに相手に着手を譲ることもできる）。
2. 除去
   縦横に繋り合う同色の石の集団の縦横全てが敵の石に取り囲まれた時、囲まれた石は盤上から除去される。図 2.1 の中の図 A の例では×の位置に黒石が置かれると白石は除去され、右側の配置に変わる。
3. 自殺着手禁止
   着手した石が除去される状態になる時は着手できない。ただし、着手によって他の石を除去できる場合

は除く。例えば、図2.1の図Bの×の点に黒石を置くことはできないが、図B'の×の点に置くことはできる。

4. 同形再現着手の禁止
着手によって石を除去する場合、石を除去した盤上の石の配置が1手前に現れた配置と同じになってはいけない。例えば図2.1の中の図Cの左側の図の×の点に黒石を置いて、右の配置になった時、右側の図の×の点に白石を置くことはできない。このような形を劫(コウ)と呼び、このルールによって永遠にゲームが終わらない状態を回避している。

5. 勝敗
プレイヤの両者が連続してパスをした時を終局とし、その時点で盤上に存在する石の数が多い方を勝ちとする。

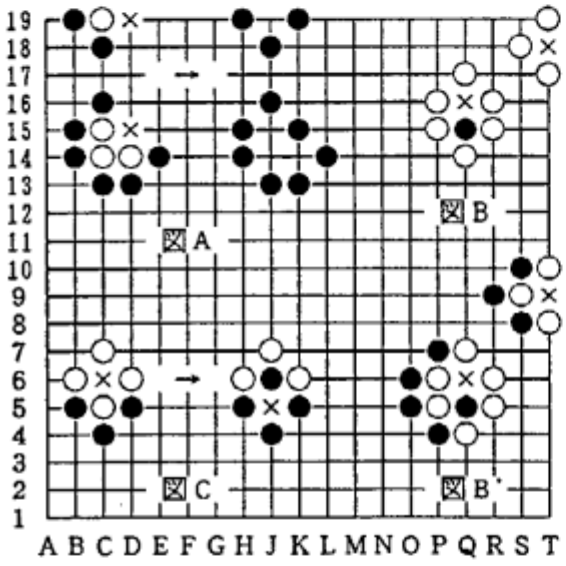


図2.1

次に、詰め碁を定式化しよう。詰め碁は通常、石の配置と手番とから、その局面以降、囲碁のルールに従って黒白双方が最善の着手を繰り返した時、ある囲まれた石の集団が除去されうるか否かの問題である。結果が除去されうる場合を「死に」といい、除去されない場合を「生き」という。図2.2は詰め碁の1例であり、この問題の答は「死に」になる。

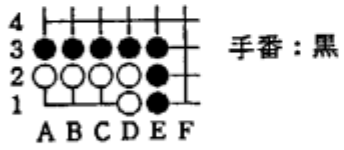


図2.2

人間が通常行っている詰め碁の定式化は以上で充分だが、今回の詰め碁プログラムでは、もう2つの条件を予め与えることにした。1つは「死に」の判定を集団の石全てが除去された時でなく、ある特定の石が除去された時とし、その特定の石の場所を指定することであり、もう1つは黒/白の打てる範囲を指定することである。例えば、図2.3のような問題で、白の石が全て盤上から除去されるまで読まなくても、3Bの石が除去されるかどうかだけで充分だとするのである。また、黒/白の手を考える時も黒に包囲されている内側の点だけで充分とするのである。

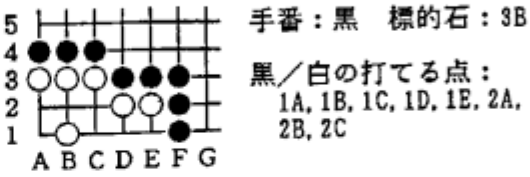


図2.3

このような制限を加えても結果はほとんどの場合変わらないが、このような制限がないと探索手数が急激に増えてしまう。従って、これらは定式化の条件というよりも、詰め碁探索のヒューリスティックといった方がよいかもしれない。実際、多くの問題でそのような標的石の場所や打てる範囲を決めることは簡単で、自動的に求められそうだが、例外的な問題も少なからずあるので、今回は便宜上人間が予め与えることにした。

即ち、今回の詰め碁プログラムは、盤上の石の配置、手番、標的石の位置、黒/白の打てる点の範囲が与えられ、標的石が除去されるか否かを探索によって調べ、その結果を答えるものである。

3 アルファ・ベータ枝刈り法

詰め碁を解く為には、与えられた初期局面にたいして、その手番のあらゆる石の置き方を試みて、その1手進んだそれぞれの局面について今度は敵側の全ての石の置き方を試みる、といった探索が必要である。このような、利害の相反する自分と敵が交互に手を進めていくような問題における解の探索は一般に「ゲーム木の探索」と呼ばれている。

ゲーム木の探索では、図3.1のように終端の局面について生き/死にの結果が決まる。例えば、図3.2のように標的としている石が除去されていれば、その局面は終端になり、結果は「死に」になる。また、図3.3のように攻め側が着手禁止の場所以外打つ手がない局面になれば、その局面も終端になり、結果は「生き」になる。

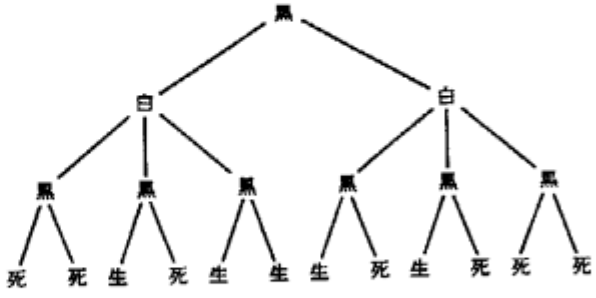


図3.1

非終端の局面の結果は、その子ノードの結果の内、その手番にとって最も有利なものが選ばれる。即ち、黒が白を攻めるような問題では、黒番では「死に」を選ぼうと

するし、白番は「生き」を選ぼうとする。そのようにして、終端からルートに向かって順繰りに結果を決めていくことにより、解が求められる。図3.1の問題の場合、図3.1のように結果は「死に」になることが判明する。

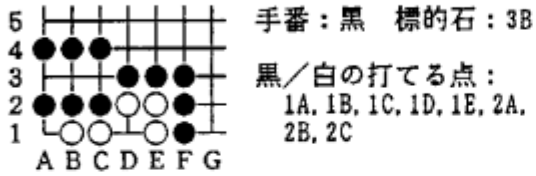


図3.2

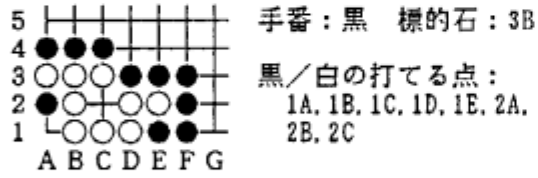


図3.3

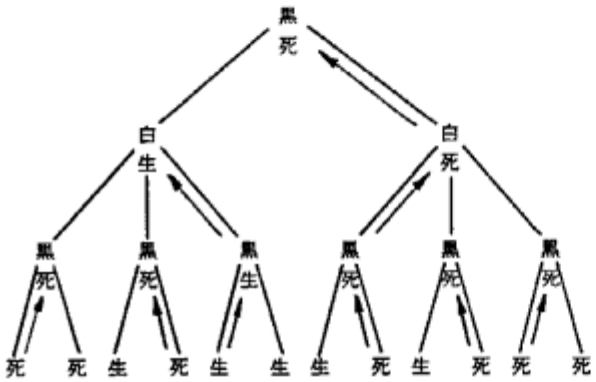


図3.4

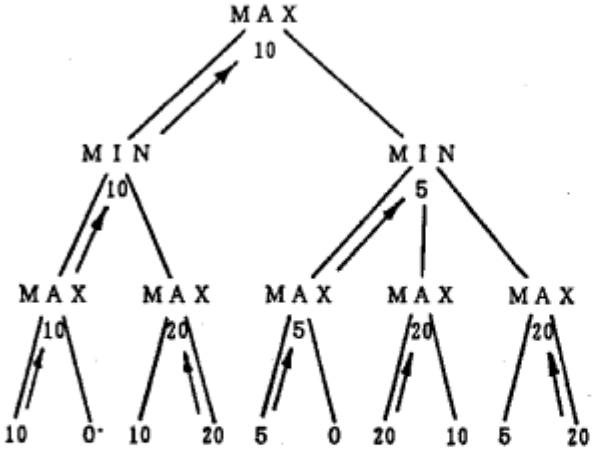


図3.5

詰め碁の場合、局面の結果は、「生き」、「死に」の2値（実際は「コウ」という結果もありうるが、説明を簡単にするため省略した）しかないが、より一般的にいろいろな評価値を取りうるゲームの木を表したものが図3.5である。MAXとMINの二人が交互に打ち、MAXから打ち始める。終端に達した時MAXにとって有利な程、高くなるような評価値を与える。MAXの手番では評価値の最も高いものを選び、MINの手番では評価値の最も低いものを選ぶ。このような方法はミニマックス法[5]と呼ばれる。

このミニマックス法と常に同じ結果で、かつ効率のよいゲーム木の探索法として、αβ枝刈り法[5]が提案されている。αβ枝刈り法は木の左端の枝から縦方向優先で探索が行なわれる。図3.6の例では、D,E,B,F,G,H,C,Aという順に評価値を決めていく。この時、αβ枝刈り法は、BとFとの評価値の関係から、GとH以下の枝は探索しなくともよい(このことを「枝を刈る」という)ことを保証する。このように、αβ枝刈り法はミニマックス法に比べて、アルゴリズムに逐次性は出てくるが、探索する枝の数を少なくできるのが、特徴である。

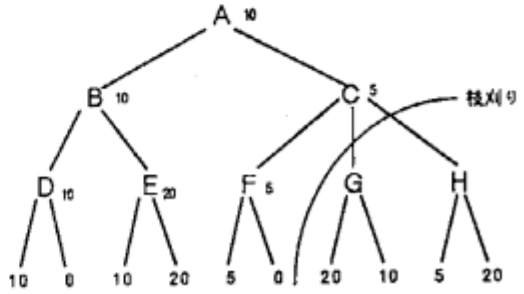


図3.6

また、枝刈りをより効果的にする為には、良い手(良い評価値の局面へ導くような手)をできるだけ左側のノードに配置する必要がある。今回の詰め碁プログラムは、囲碁の「急所」や「手筋」といったヒューリスティックを使って、有望と思われる手をできるだけ木の左側に配置するようにしている。

4　並列アルファ・ベータ探索アルゴリズム

αβ枝刈り法は、左端の枝から縦方向優先で探索を行い、その結果を用いて右側の枝を刈ろうとする為、きわめて逐次性の高いアルゴリズムである。並列コンピュータでゲーム木の探索を行うには、逐次性の高いαβ枝刈り法をたくさんのプロセッサを用いて、いかに並列実行するかという点がポイントとなる。そのために、

1. 並列実行の可能性を高めること
2. 枝刈りの効率はなるべく落とさないこと

といった、2つの相反する要求を同時に満たすことを目指して、並列化の方式を検討した。

基本的な考え方は、並列度を増すために縦型探索ではなくて、探索木を同時並列的に展開していくことである。評価値はあとから終端の側から上がってくるので、各ノードには、下から報告される評価値を兄弟関係にある子ノードに伝えたり、評価値に基づいて枝刈りを実施したりする為のプロセスを配置しておく。またプロセス間はストリームで接続しておく。プロセスの構造を図に

示したものが図4.1である。各プロセスの機能の説明については最後に付しておいた。

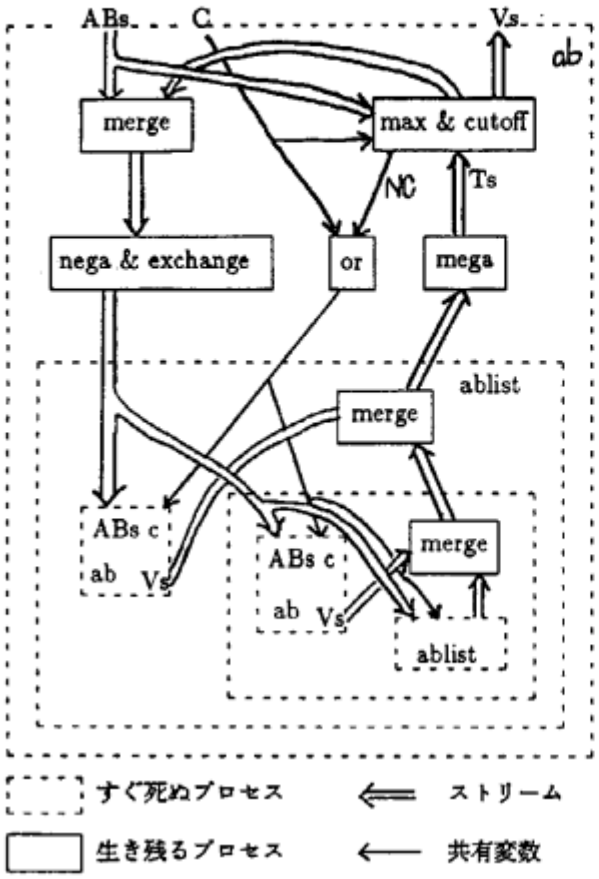


図4.1

しかし、ただ並列に展開していくだけでは終端から評価値が上がってくる以前に探索木をどんどん展開してしまい、枝刈りはほとんど期待できない。木の左側程有望な手になっている特徴をうまくいかせるようにしたのが、図4.2のプライオリティ制御である。すべてのノードについて、常に左側のノードの実行プライオリティが高くなるようにすれば、木を展開する順番は、プロセッサ1台の場合には縦型探索に一致し、複数プロセッサで実行する場合でも木の左の方の枝を優先的に展開することができる。こうしておくと、枝刈り効率もある程度確保される。この実行プライオリティ制御の機能はKL1言語処理系によりサポートされている。

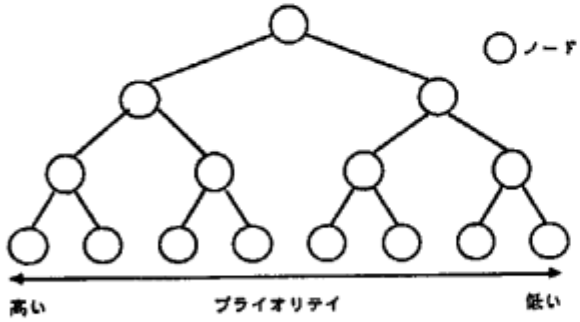


図4.2

5 負荷分散方式

プライオリティ付けされたノードプロセスをすべて適当な別々のプロセッサに分散したのでは、通信のオーバヘッドが大きくなり過ぎる。そこで探索木のある深さに達するとそれ以下の探索は同一プロセッサ内で続けるようにした。ある部分木を1つのプロセッサ内で左側のノード程高いプライオリティで実行することは、縦型探索に一致するので、むしろ積極的に、縦型探索を行う逐次$\alpha\beta$枝刈り法を使うようにした。

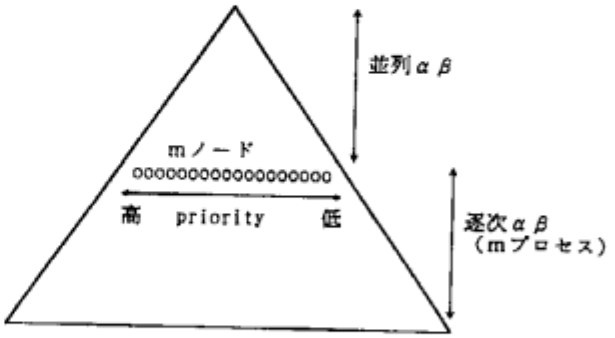


図5.1

図5.1のように、探索木のルートからある深さまでは、並列にノードプロセスを展開し、その深さから下には逐次$\alpha\beta$枝刈り法に切り換えて実行している。そして、これらの部分木の探索を行う逐次$\alpha\beta$枝刈りのプロセスを、負荷分散の単位としている。なお、これらのプロセスには、左側優先の実行プライオリティが付けられている。

また、並列$\alpha\beta$枝刈りのプロセス群はうまい方法があれば複数のプロセッサに分散してもよいが、計算量があまり大きくならないことが見込まれるので、現在は1台のプロセッサで実行させている。

逐次$\alpha\beta$枝刈りを実行するプロセスをどのプロセスに割り振るかという負荷分散の方式として、

1. 準静的負荷分散
2. 動的負荷分散

の2種類を試作した。

準静的負荷分散というのは、逐次$\alpha\beta$枝刈りのプロセスを乱数を用いて各プロセッサに割り付けてしまうものである。1つのプロセッサに複数のプロセスが割り付けられ、プライオリティの高いプロセスから順次実行される。

この準静的負荷分散では、逐次$\alpha\beta$枝刈りプロセスをどんどん割り付けてしまうので、運悪く1つのプロセッサに処理の重いプロセスが集中してしまったり、高プライオリティのプロセスが集中してしまうと、他のプロセッサが遊んでしまい、並列効果が思うように出ないこともありうる。1台あたりにばらまかれる逐次$\alpha\beta$枝刈りのプロセスの数を増やせば増やす程、負荷バランスは良くなると思われるので、逐次$\alpha\beta$枝刈りに切り換える深さを深くすれば稼働率は上がることが期待される。しかし、あまり深くし過ぎると逐次$\alpha\beta$枝刈りのプロセスの個数が増えるので通信のコストが高くなる。両者の兼ね合いで逐次$\alpha\beta$枝刈りに切り換える深さを決める必要がある。

これに対して動的負荷分散方式は、暇なプロセッサにたいして逐次αβ枝刈りのプロセスを割り付けていくという方式である。暇なプロセッサの検出には、各プロセッサに予め最低プライオリティのプロセスを常駐させておき、暇になったら（このプロセスは最低プライオリティであるので、動くこと自体が暇な時となる）、その旨をストリームを介して、報告させるのである。図5.2のように並列αβは左側優先でどんどん一定の深さまで進んだ局面の部分問題を生成していき、その部分問題をその時点の暇なプロセッサに解かせるのである。

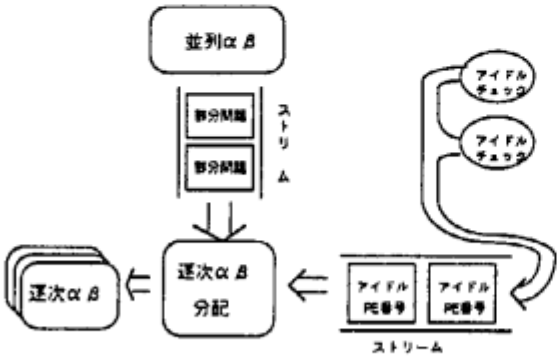


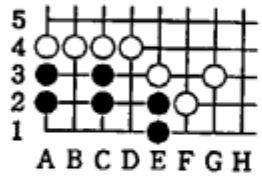
図5.2

1つの逐次αβ枝刈りプロセスが終わって、次の逐次αβ枝刈りプロセスが駆動されるまでのロスタイムは生じるが、負荷バランスは一般に良くなることが期待される。

6 考察

図6.1のような2種類の詰め碁問題について、負荷分散の方式、逐次αβ枝刈りに切り換える深さ及びPE(プロセッサエレメント)台数などのパラメータをいろいろ変えて、解かせてみた。最も性能の良かった深さに対する測定結果を表6.2に示す。

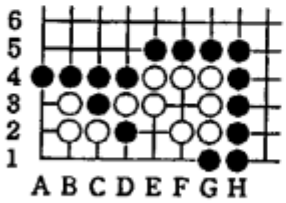
問題1



手番：白　標的石：2C

黒／白の打てる点：
1A, 1B, 1C, 1D, 1E, 2B, 2D, 2E, 3B, 3D, 1F(黒のみ)

問題2



手番：黒　標的石：2B

黒／白の打てる点：
1A, 1B, 1C, 1D, 1E, 1F, 2A, 2D, 2E, 3A, 3F

図6.1

この測定結果から、PE台数と性能向上率との関係を求めたものが、図6.3である。ここで、性能向上率とは、

$$\text{性能向上率} = \frac{\text{1PEにおける処理時間}}{\text{nPEにおける処理時間}}$$

のことで、PE台数を増やすことによって、何倍速くなったかを示す量である。この図からわかるように、64台で性能の比較的良い2つは、PE1台の場合と比べて約16倍速くなっている。そして、その2つとも動的負荷分散方式である。準静的負荷分散よりも、動的負荷分散方式の方が性能が良いように思われるが、現時点ではまだそれ程多くのデータについて調べていないので、結論は下せない。

No.		A	B	C	D
問題		問題1		問題2	
負荷分散		準静的	動的	準静的	動的
深さ		3	2	4	5
プロセッサ台数	1PE	21,257手 3分24秒	21,257手 3分30秒	102,703手 20分	89,536手 18分
	4PE	32,534手 1分44秒	29,937手 1分40秒	132,008手 8分39秒	124,740手 7分23秒
	16PE	60,323手 38秒	29,759手 32秒	331,504手 4分35秒	164,550手 2分16秒
	64PE	70,340手 17秒	32,175手 12秒	591,619手 4分00秒	266,515手 1分06秒

表6.2

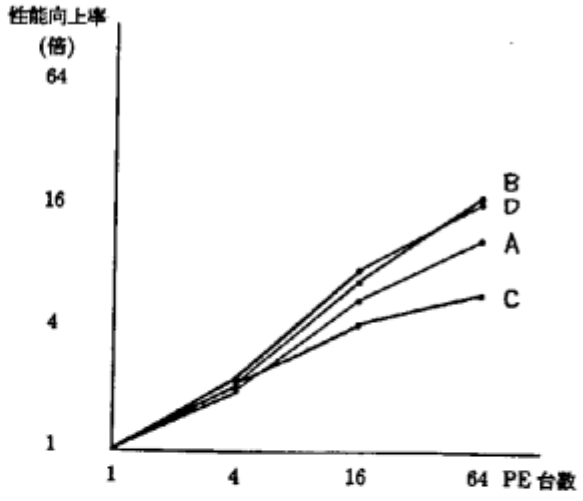


図6.3

PE64台で性能向上率が、16倍程度にしか伸びない原因として、以下の2つがある。

1. 枝刈り効率が下がって仕事が増えている。
2. 稼働率が下がっている。

PE台数と探索に要した手数との関係を示したものが図6.4である。頭打ちになっているものもあるが、一般に

台数が増える程探索に要する手数も増えている。枝刈り効率が下がって無駄な仕事が増えていることが分かる。

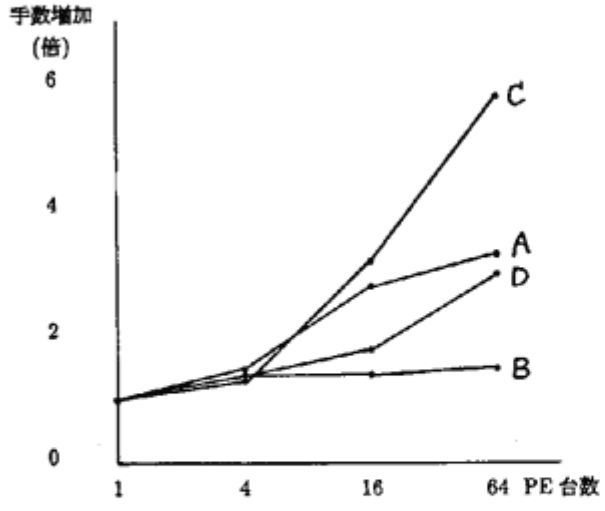


図6.4

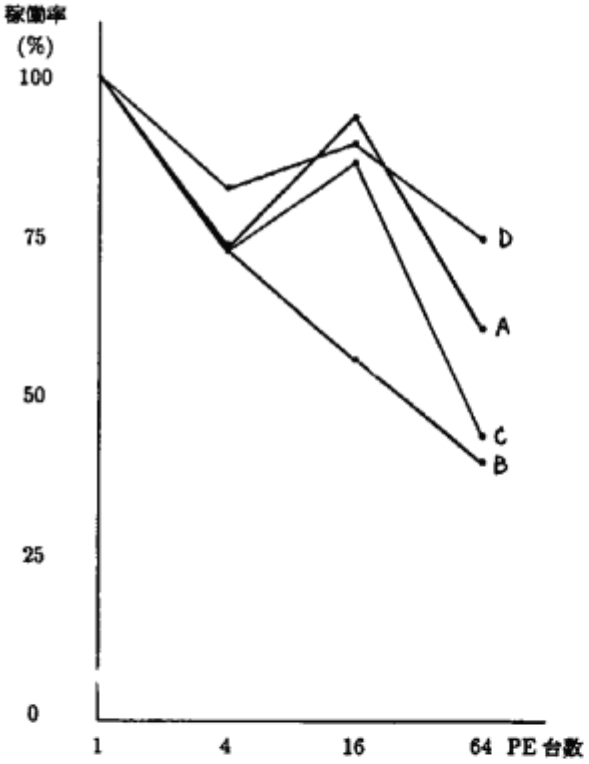


図6.5

次に、稼働率について見てみよう。残念ながら、現在、ある一定時間内のプロセッサの平均稼働率を知る方法はない。しかし、先程の測定値からおおよその見当はつく。探索における仕事量は通常その手数に比例する。通信のオーバヘッドを除けば、1台で探索した場合の1手に掛かる処理時間と複数台で探索した場合の1手に掛かる処理時間とは同じである。従って以下の式で、通信のオーバヘッドを無視できると仮定した時の、全PEに対する平均の稼働率が見積もれる。

$$\text{推定平均稼働率} = \frac{T1 \times Nn}{N1 \times Tn \times n}$$

ただし、

T1：1PEでの処理時間
N1：1PEでの探索手数
Tn：nPEでの処理時間
Nn：nPEでの探索手数
n：PE台数

この推定平均稼働率とPE台数との関係を求めたものが、図6.5である。多少の上下はあるが、一般に台数が増える程、稼働率が下がるようである。

最後に、動的負荷分散方式について、逐次αβ枝刈りに切り換える深さをいろいろ変えてみて、PE64台での性能がどう変化するか求めたグラフが図6.6である。この図からは、逐次αβ枝刈りに切り換える深さを小さくし過ぎても大きくし過ぎても性能向上率は下がる、くらいのことしか言えそうにない。最適な深さは問題によっていろいろ異なるようだが、調べたデータ量が少なく、現在明瞭に分析するまでには至っていない。

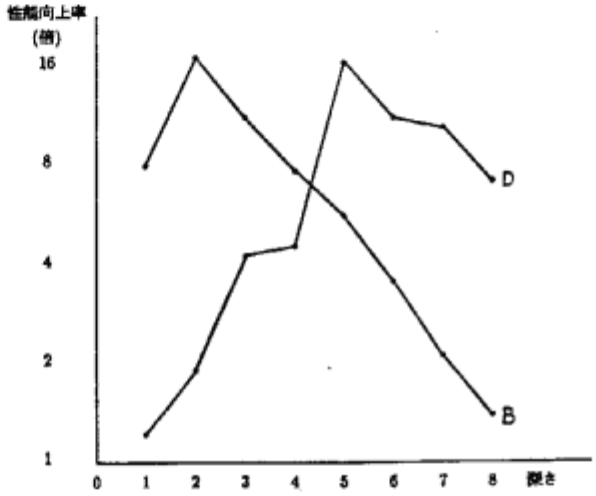


図6.6

7 まとめ

αβ枝刈り法を用いて詰め碁の問題を解くプログラムを並列コンピュータマルチPSI上に試作した。本来逐次性の高いαβ枝刈り法のアルゴリズムから効果的に並列性を引き出す為の、並列アルゴリズム及び負荷分散方式について研究し、その測定結果の一部を速報として報告した。

例題として、初段クラスの問題を2つとりあげ、解かせてみた。どちらの問題もPE64台で最高16倍強の性能を示し、時間の掛かった方でも1分程度で解いた。その時の全プロセッサの平均稼働率は約75%であった。

今回はパラメータとして(並列αβ枝刈りから逐次αβ枝刈りに切り換える)深さを使っている。この値の選び方によって性能向上率は大きく変化すると思われるが、この値をどのように選べばよいかは今後の課題である。

今後の予定として、より多くの詰め碁問題について計測をし、深さ選定方法などの解析を続けるとともに、より性能の高い方法を検討していく。そして、それらの成果を囲碁対局システム「碁世代」の並列化の際の参考にしていくつもりである。

参考文献

[1] 実近憲昭.他：囲碁システム「碁世代の方法」, ICOT 研究速報 TM-618, 1988

[2] T.Chikayama, etc. : Overview of the Parallel Inference Machine Operating System (PIMOS), PROCEEDING OF THE INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS 1988 Vol.1 p.230-251, 1988

[3] K.Ueda. : Guarded horn clauses: A parallel logic programming language with the concept of a guard, ICOT Technical Report TR-208, 1986

[4] S.Uchida, etc.: RESEARCH AND DEVELOPMENT OF THE PARALLEL INFERENCE SYSTEM IN THE INTERMEDIATE STAGE OF THE FGCS PROJECT. PROCEEDING OF THE INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS 1988 Vol.1 P.16-36, 1988

[5] 白井良明, 辻井潤一：人工知能, 岩波講座情報化学-22, 1982

A　並列アルファ・ベータ枝刈りのプロセス

並列$\alpha\beta$枝刈りの各プロセス毎の機能について説明する。

- ab :
  - 最初に木の葉に達したか否かの判定をし、まだなら、max_&_cutoff、or、nega_merge_ex のプロセスを生成し、ablist を起動する。
  - 木の葉に達していれば評価値を Vs に返す。
- ablist :
  - 候補の数だけ新しい ab を生成する。ab の結果は merge して Ts に流す。
  - α値、β値を子の ab に流す時は符号を反転する。
- max_&_cutoff :
  最新のα、β値を保持しており、カットオフ制御も行う。
  - C から打ち切りを受け取ったら、Vs=[ ],NAs=[ ] で終了する。
  - Ts から新しい値を受け取ったら、α値と比較し大きければα値を更新する。
    - この時β値とぶつかったら、カットオフ制御 (NC= 打切り,NAs=[ ],Vs=[β値] で終了) を行う。
    - ぶつからなければ、NAs=[新α値 — 旧 NAs] とする。
  - Ts から [ ] を受け取ったら、カットオフ制御 (NC= 打切り,NAs=[ ],Vs=[最新のα値] で終了) を行う。
  - ABs からα値を受け取ったら、自分のα値と比較し、大きければ更新する。
  - ABs からβ値を受け取ったら、自分のβ値と比較し、小さければ更新する。この時、α値とぶつかったらカットオフ制御 (NC= 打切り,NAs=[ ],Vs=[β値] で終了) を行う。
- or :
  C と NC のどちらかが打切りになった時打切りを出力する。