TR-0734

Iterative-Deepening A アルゴリズムの並列化と並列推論マシン PIM/m 上の性能評価

和田　正寛、六沢　一昭（沖）
市吉　伸行

January, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Iterative-Deepening A* アルゴリズムの並列化と並列推論マシン PIM/m 上の性能評価

和田 正寛 *† 市吉 伸行 ‡ 六沢 一昭 §
(財) 新世代コンピュータ技術開発機構 ¶ 沖電気工業 (株)
第 7 研究室 総合システム研究所

E-mail: {mwada, ichiyoshi}@icot.or.jp, rokusawa@okilab.oki.co.jp

概要

探索問題は多くの問題に含まれる問題であり、様々な探索アルゴリズムが知られている。その一つに Iterative-Deepening A* [1] がある。Iterative-Deepening A* とは、探索深さを徐々に深くしながら深さ優先探索を繰り返し、探索の際にゴールに近いと思われる枝を優先するというアルゴリズムである。この探索手法を並列実行によって高速化する実験を行った。並列化の実現方法としては、探索木のある段階までは逐次に探索を行い、それから先の部分木は並列に探索するという方式を用いている。この方式では、逐次探索から並列探索に移る段階を決定するのが難しい問題となる。今回の実験では、Iterative-Deepening A* アルゴリズムにおいて動的に得られる閾値を利用して、並列探索に移る段階を自動的に決定する枠組を作り、台数効果の検討を行った。

この探索アルゴリズムを並列論理型言語 KL1 [2] で記述し、15 パズル問題を PIM/m [3][4] 上で解いてみたところ、プロセッサ 256 台構成の PIM/m で最大 232 倍の効果を得ることに成功した。

A parallel iterative-deepening A* algorithm was implemented in the parallel logic language KL1 and evaluated on the parallel inference machine PIM/m using the Fifteen Puzzle as a simple problem. The multi-level dynamic load balancer allocates the subtrees of the highly unbalanced search tree to the processors. The heuristic threshold values were used to determine the load distribution points. A maximum of 232-fold speedup was attained with 256 processors. The dependency of speedup on problem sizes and problem instances is discussed.

An Implementation of Parallel Iterative-Deepening A and Its Evaluation on the Parallel Inference Machine PIM/m

†Masahiro WADA
‡Nobuyuki ICHIYOSHI
§Kazuaki ROKUSAWA
¶Institute for New Generation Computer Technology

1 Iterative-Deepening A*

探索問題を解くために用いられるアルゴリズムには様々なものがあるが、中でもよく知られているのは深さ優先探索と幅優先探索である。しかしこれら二つの探索戦略には、それぞれ下記のような問題点が存在する。

- 深さ優先探索の問題点
  1. 選択された枝が無限に伸びている場合、他の枝に解が存在しても 発見することができない。
  2. 解が見つかっても、それが最適解である保証がない。
- 幅優先探索の問題点
  1. 探索した節点をすべて記憶しておかなければならないため、 記憶のための資源を大量に消費する。

深さ優先／幅優先の探索戦略を組み合わせ、問題点を克服したのが Depth-First Iterative-Deepening (以後 ID と略す) である。この探索戦略のアルゴリズムは以下のようになる。

1: Depth=0 とする。
2: 出発節点から深さ Depth までの範囲内で深さ優先探索を行う。
3: 解が見つかれば終了。
4: 解が見つからなかったなら、Depth を 1 増やし、その回の探索過程をすべて忘れる。
5: 2: に戻る。

この戦略を従えば、たとえ枝が無限に伸びていたとしても解が存在すれば必ず発見でき、また最初に発見される解は必ず最も浅い所にある解の一つである。そして探索動作としては深さ優先探索を繰り返すだけであるから、記憶のための資源消費は深さ優先探索で消費される

のと同じ程度ですむ。以上により前述の問題点は回避される。

このアルゴリズムによると、Depth=dで解が見つかる場合、Depth=$d-1$までの探索は無駄であり、それが実行時間の増大をもたらすことが懸念される。しかし探索空間は指数関数的に増大していき、最終回の探索が最も大きな時間を占めることから重大な問題にはならない [1]。

IDのアルゴリズムでは、探索深さを限った深さ優先探索を何度も繰り返し、繰り返しの度に深さを1段ずつ増加させる。しかし、実際には、よりゴールに近いであろうと思われる枝をより深くまで探索するという戦略を取った方が効率がいいと思われる。そこで枝刈りを行う為に、ヒューリスティック探索のA*アルゴリズム [5] を取り入れる。A*アルゴリズムとは、ゴールにより近いと思われる枝を優先して探索するという戦略である。ゴールまでの距離の目安としては、そのノードに達するまでに要したコストgと、そのノードからゴールにいたるまでのコストの見積もり値hの和fを用いる。ただし、hの値をゴールまでの最短コストより大きく見積もってはならない。hの値を見積もり過ぎた場合は得られる解が最適解でない可能性があるが、A*アルゴリズムでは得られる解が最適解であることが保証されている。

A*を取り入れたID(以後IDA*と略す)の具体的なアルゴリズムとしては、以下のようになる。

1: 閾値 Threshold=0 とする。
2: 出発節点から、
Threshold $\leq f$ である間、
深さ優先探索を行う。
3: 解が見つかれば
その中から最小のコストのものを選んで終了。
4: 解が見つからなければ、深さ優先探索時に
Threshold を越えたf値を集め、
その中から最小値を選ぶ。
5: その最小値を新たな Threshold の値と
設定する。
6: 2: に戻る。

2 IDA*アルゴリズムの並列化と負荷分散

並列実行の方式としては、探索木の部分木を独立に並列探索する事で実現している。具体的には、探索途中で生成された状態のうちいくつかを選んで、その先の探索を他のプロセッサで実行することによって並列動作が行われる。

この方式では、閾値の収集が処理のボトルネックになりやすい。これを回避するために、探索木における自分の親に結果を戻すごとに、子供同士の閾値を比較してもっとも小さいものだけを返すようにしている

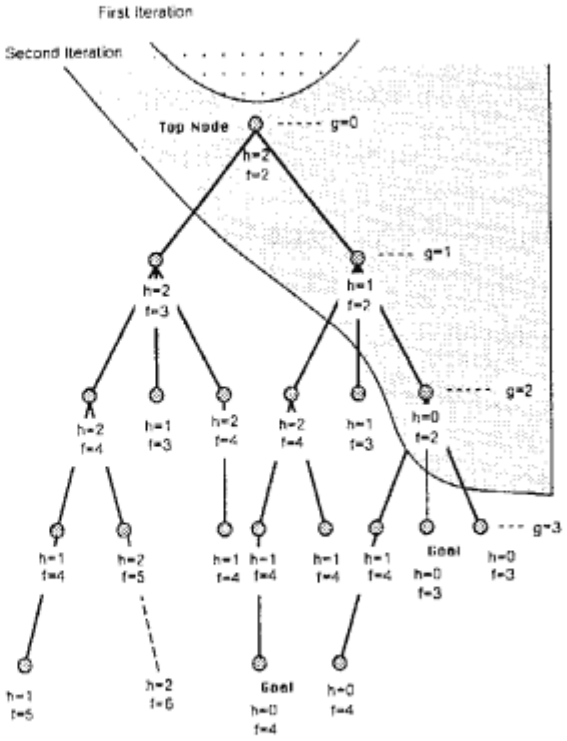

図 1: 探索木とIDA*探索戦略

負荷分散を実現する方法としては、マルチレベル動的負荷バランサ [6] (以下、負荷バランサと呼ぶ) を利用している。

負荷バランサは、全てのプロセッサに稼働状態報告用のプロセスを投げ、マスタプロセッサがその報告を集めて管理している。プログラムが処理を割り当てるためのプロセッサを要求すると、マスタプロセッサはその時点で暇なプロセッサを与えるという動作を行う。このような構造であると、マスタプロセッサに要求が集中することによる処理のボトルネックが発生するおそれがあるが、負荷バランサではプロセッサをグループ化し、グループごとにサブマスタプロセッサを割り当てて負荷割り付けを階層的にする事によってその問題を回避している。階層を1段だけ設ける負荷分散を1レベル負荷分散、複数段設けるものをマルチレベル負荷分散と呼んでいる。

また、割り当てた負荷が不均等であると、プロセッサグループの中に早く作業を終えるものと遅くまで動き続けるものとが出てくる。このような状態は並列効果を低下させる原因となる。負荷バランサは作業の終了したプロセッサグループを作業中のプロセッサグループに合流させて、仕事を分担させるようにしている。このしくみをグループマージと呼んでいる。

IDA*アルゴリズムのプログラムからこの負荷バランサを呼び出す事によって負荷分散を行っている。

ここで問題となるのが、他のプロセッサに部分木の探索を割り当てるタイミングをどのように指定するかという事である。負荷バランサを利用した実験としては詰

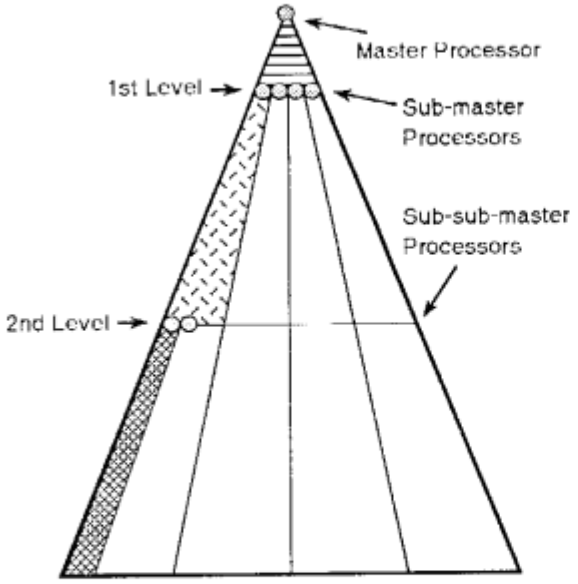

図 2: マルチレベル負荷分散

め込みパズル (ペントミノ) 問題が文献 [6] で紹介されている 。詰め込みパズルとは、ある矩形領域内に様々な形をしたピースを詰め込んでいくという問題である。このような問題では、探索空間の深さはピースの個数によって決定される。よって上記の実験では、問題を解いていく上での探索深さに注目して負荷分散を実行している。プログラムとしては、探索を進めていく間、探索深さを数えておき、ある深さに達した時点で負荷分散を実行するという構造である。この戦略は、探索問題に一般的に適用出来る手法である。

一般に、負荷分散を実行する探索深さを決定するのは容易ではない。深さが浅過ぎると使用出来るプロセッサの台数に比べて割り当てる部分問題が少な過ぎるという問題が発生し、逆に深さが深過ぎると負荷分散を実行するまでに 1 台のプロセッサで処理すべき問題が大きくなり過ぎる。詰め込みパズル問題では探索木の最大探索深さはあらかじめ得る事が可能であるので、それを目安に負荷分散深さを検討する事が出来たが、探索問題一般にそのような手法が使えるとは限らず、よって負荷分散深さを決定するのはさらに困難になる。

またIDA* アルゴリズムの場合、閾値による打ち切りのために探索木が非常にアンバランスになり、深さを基準にして負荷分散をする事は適切でないと予想される。

IDA* アルゴリズムは、探索を繰り返すごとに次の探索で用いるべき閾値を返す。この値は探索における探索面を示している。負荷分散を行うのは探索面に沿った方が効率的であると考えられる。動的に情報を得やすいという利点もある事から、閾値による負荷分散を実験する。

具体的に、繰り返し探索回数が $I = 1, 2, 3, \cdots$ と進むごとに閾値が $T = 0, t_2, t_3, \cdots t_l$ となる場合を考えると、探索回数が I である時、f 値が $t_2, \cdots t_l$ となる節点に達するごとに負荷分散を実行するという方式である。これをヒューリスティック関数値による負荷バランス方式と名付ける。

3 15 パズル問題

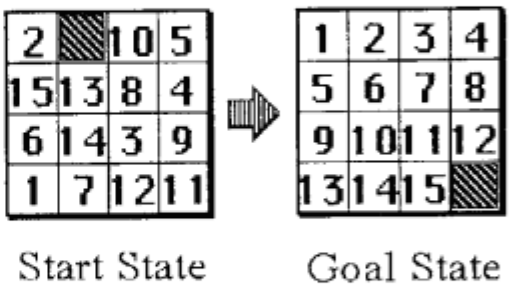

図 3: 15 パズル問題

実験問題としては 15 パズル問題を用いた。15 パズル問題とは、4 × 4 の桝目の上に 1 から 15 まで番号付けられたピースが乗っており、一か所残った空き桝にピースを順次動かし、ピースの配置を目標の配置に並べ替えるという問題である。

15 パズルでは、初期状態や目標状態を変える事によって探索木はまったく違ったものとなる。今回実験に用いた初期状態について、表 1 にその特徴を記す。(今回の実験では目標状態は固定している。)

表 1: 実験に用いた 15 パズルの問題状態の特徴

問題	探索深さ	全状態数	
[A]	43	262,018	無作為に選んだ状態
[B]	43	295,215	[A] と深さが同じ状態
[C]	44	295,252	[A] からピースを一つ動かした状態
[D]	37	1,526,582	[A] と比較して深さは深くないが繰り返し探索回数が多い状態
[E]	41	5,885,471	[A] よりも全状態数を大きくした状態
[F]	42	1,167,104	[A] と深さは同程度で全状態数を大きくした状態
[G]	43	1,298,207	[F] からピースを一つ動かした状態

表中の全状態数とは、繰り返し探索の最終回における探索木の節点の個数であり、問題空間の大きさの目安となる。

4 実験と考察

4.1 探索深さによる負荷分散

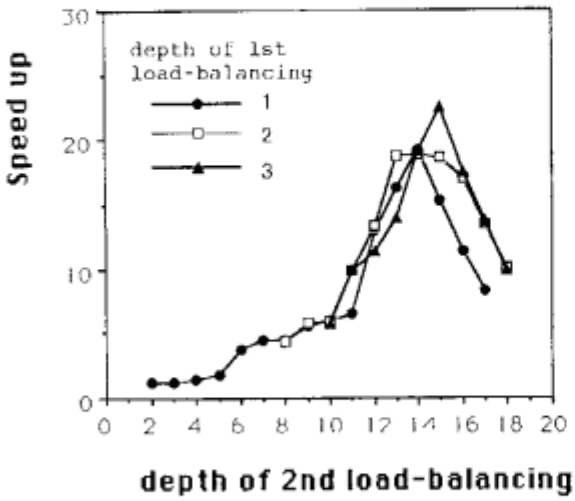

図 4: 探索深さによる 2 レベル負荷分散

問題 [A] を用いて探索深さによる負荷バランス方式の実験を、プロセッサ 64 台構成で行った。すべての探索深さに対して 1 レベル負荷分散による台数効果を測定したところ、最大で 12 倍の台数効果しか達成できなかった。

2 レベル負荷分散の場合、探索深さの組により負荷分散レベルが作られるので可能な組み合わせは増大する。図 4 に示す組み合わせで台数効果を測定してみたが、最大で 23 倍にとどまり、高い台数効果を得る組み合わせを発見する手掛りは得られなかった。
(なお、この実験はマルチ PSI[7] [8] で実験を行った。)

4.2 ヒューリスティック関数値による負荷分散

ヒューリスティック関数値による負荷分散方式の実験を行ったところ、台数効果は表 2 のような結果となった。

プロセッサを 256 台用いる事により、[E] において 232 倍の台数効果を得ている一方で、[A], [B], [C] の台数効果は低い値にとどまっている。プロセッサ 1 台での実行時間を見てみると [A], [B], [C] はいずれも実行時間が短い。また表 1 における [A], [B], [C] の全状態数をみても他に比べて小さい事がわかる。これらの事より、[A], [B], [C] は問題空間が小さいことが台数効果の低さの原因であると推測される。

また、使用プロセッサ数を変えて台数効果を測定した結果を図 5 に示す。

表 2: PIM/m プロセッサ 256 台による台数効果

問題	1PE での実行時間 (秒)	256PE での実行時間 (秒)	台数効果 (倍)
[A]	390.4	8.4	46.5
[B]	423.9	5.7	74.4
[C]	437.5	7.4	59.1
[D]	2,542.3	12.7	200.2
[E]	9,463.5	40.7	232.5
[F]	1,753.2	11.8	148.6
[G]	1,949.2	11.7	166.6

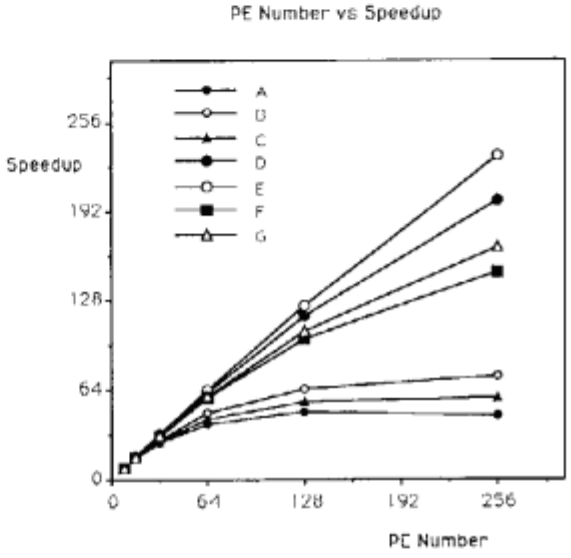

図 5: 使用プロセッサ数を変えて測定した台数効果

4.3 問題空間の大きさと台数効果の関係の分析

問題空間の大きさと台数効果の関係をより詳しく分析するために、繰り返し探索ごとの台数効果を調べる。表 2 における台数効果の値は、15 パズルを解き始めてから解が求まるまでの時間全体を用いて算出している。IDA* の探索動作では、繰り返し探索ごとに探索木の大きさが大きくなる。そこで、繰り返し探索ごとに台数効果を計測することによって、問題空間そのものの大きさと台数効果の関係を得ることが出来る。

例えば問題 [A] の場合、繰り返し探索ごとの問題空間の大きさ (プロセッサ 1 台での実行時間で表現する) と台数効果の値は表 3 のようになる。(1 回目の繰り返し探索は非常に実行時間が短いのでデータとしては省いてある。) 表 2 における [A] の台数効果は 46.5 倍と、実験結果の中で最も小さい値になっているが、表 3 を見ると最終回の繰り返し探索において 109.5 倍という値が出ていることがわかる。

表 3: 問題 [A] における繰り返し探索ごとの台数効果

繰り返し探索回数	1PE での実行時間 (秒)	256PE での実行時間 (秒)	台数効果 (倍)
2 回目	0.2	1.9	0.1
3 回目	1.3	2.5	0.5
4 回目	9.2	1.3	6.9
5 回目	56.2	1.4	40.1
6 回目	323.4	3.0	109.5

また、すべての問題における繰り返し探索の最終回の台数効果の値を表 4に示す。

表 4: 最終回の繰り返し探索における台数効果

問題	1PE での実行時間 (秒)	256PE での実行時間 (秒)	台数効果 (倍)
[A]	323.4	3.0	109.5
[B]	359.9	2.4	148.6
[C]	362.2	2.5	146.0
[D]	1,952.3	8.3	234.8
[E]	7,438.1	30.0	247.9
[F]	1,440.3	6.8	210.8
[G]	1,604.1	7.1	225.6

問題空間が大きくなるに従って台数効果が向上しているのが表 3, 表 4からも明らかである。この分析を全問題状態に対して行い、結果をグラフに表したのが図 6である。

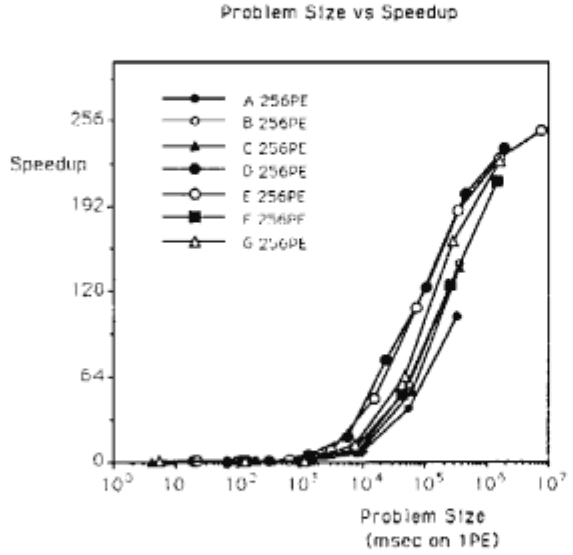

図 6: 繰り返し探索ごとの台数効果

4.4 実験結果のモデル化

図 6のグラフでは縦軸に台数効果 U_p を取っているが、使用プロセッサ台数 p を変えて実験結果を比較するために、台数効果 U_p の値をプロセッサ台数 p で割った値 (以後、効率 E と表す) を用いる。

図 6のグラフは、シグモイド関数 (hyperbolic tangent) の形をしている。そこで、プロセッサ 1 台での実行時間を W 、プロセッサ台数 p での実行時間を W_p 、効率 $E = 0.5$ でのグラフの傾きを α 、 $E = 0.5$ となる時の実行時間を w とすると、図 6のグラフは

$$E = \frac{1}{2}(1 + \tanh 2\alpha(S - s)) = \frac{1}{1 + e^{-2\xi}} \qquad (1)$$

(ただし、$S = \log W, s = \log w, \xi = 2\alpha(S - s)$)
によってよく近似できる。

ここで、

$$E = \frac{U_p}{p} = \frac{W}{W_p * p} \qquad (2)$$

であることから、

$$\begin{aligned} W_p &= \frac{W}{p}(1 + e^{-2\xi}) \\ &= \frac{W}{p}(1 + e^{-4\alpha(S-s)}) \\ &= \frac{W}{p}(1 + (\frac{w}{W})^{4\alpha}) \\ &= \frac{W}{p} + \frac{1}{p}(w^{4\alpha}W^{1-4\alpha}) \end{aligned}$$

となる。

この式における $\frac{W}{p}$ は、大きさ W の問題をプロセッサを p 台用いて解いた時に、負荷が均一に分散され、通信オーバヘッド等が無い、理想的な並列実行時間そのものであると考えることが出来る。よって W_p の残り時間は、並列探索を行うためのオーバヘッド O である。

$$O = \frac{1}{p}(w^{4\alpha}W^{1-4\alpha}) \qquad (3)$$

問題 [E] を用いて、プロセッサ台数を変えて効率を測定した結果を図 7に示す。図 7では、 $E = 0.5$ における傾き α はほぼ一定 ($\alpha \approx 0.18$) であるが、プロセッサ台数を増やすにつれてグラフが右に移動していく。これはプロセッサ台数を増やした場合に、増やす前と同じ効率を得るためには、より大きな問題空間が必要とされることを示している。

また $\alpha \approx 0.18$ より、プロセッサ台数 p を固定した場合は $O \propto W^{0.28}$ であり、問題空間の大きさ W を大きくするとオーバヘッド O も大きくなるが、問題空間の大きさに対するオーバヘッドの比率 $\frac{O}{W}$ は減少することがわかる。

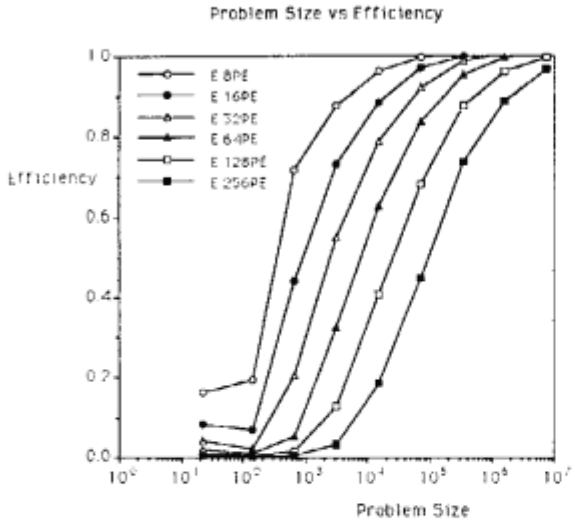

図 7: PE 数の変化に対する効率の変化 (問題 [E])

4.5 並列実行時間中のオーバヘッド時間

並列実行時間 W_p には並列実行時間そのものとオーバヘッド時間が含まれていることが式 (3) からも明らかになった。

ここで、256 台のプロセッサの実行中の稼働状態を調べる。図 8は、問題 [E] の繰り返し探索の最終回における 256 台のプロセッサの稼働状態を、横軸に経過時間、縦軸に全プロセッサの稼働状態の平均を表したグラフである。(横軸の 1 目盛りは 2 秒であることに注意。)

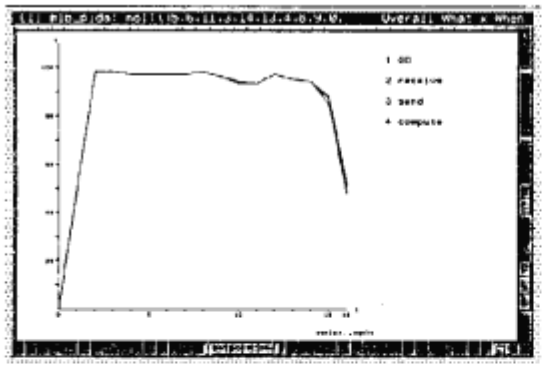

図 8: プロセッサの稼働状況 (問題 [E])

グラフでは計算時間とともにメッセージ通信のための時間とガベージコレクションにかかった時間が表されているが、稼働状態のほとんどが計算時間であることがわかる。このグラフによるとプロセッサの稼働状態は、最初の約 4 秒間は 0% から 100% に増加していき、その後は小さな変動はあるがほぼ 100% の状態を保ち、最後は急速に低下している。このグラフの傾向から、以下の現象が考えられる。

1. 初めに稼働状態が 0% から高くなっていくのは、処理が全プロセッサに分配されていく過程である。
2. 途中状態は、全プロセッサの稼働状態がほぼ 100% であることを示している。途中の小さな変動は、部分問題の不均一やそれに伴うグループマージ等が原因であると考えられる。
3. 最後に急速に低下しているのは、全部のプロセッサにおいてほぼ一斉に処理が終了していることを示す。

これらの事から、問題 [E] の最終回の繰り返し探索では、全プロセッサに負荷を分配するために必要な時間に起因するオーバヘッドがオーバヘッド時間の大部分を占めていることがわかる。

一方、図 9は問題 [F] の最終回の繰り返し探索における結果である。このグラフでは、全プロセッサに負荷がいきわった後の稼働状態は、80% 程度にとどまっている。これは負荷の不均一によるグループマージが頻繁におこり、そのための処理がボトルネックになっていると推測される。

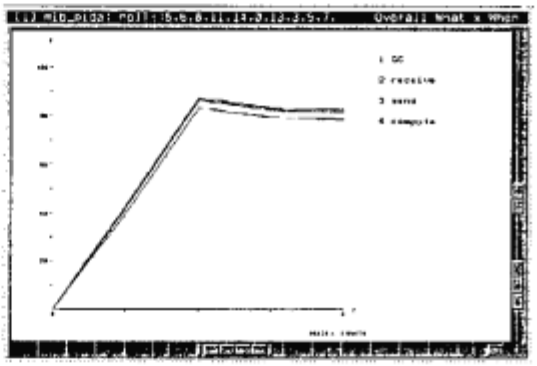

図 9: プロセッサの稼働状況 (問題 [F])

以上のことからオーバヘッド時間には、初めに負荷を全プロセッサに分配するためのオーバヘッドと、負荷の不均一を原因とする探索途中でのオーバヘッドの 2 種類が存在していること、また問題によってそれぞれの相対的大きさが異なっていることがわかった。

4.6 同じ問題空間サイズでの効率のばらつき

効率の特性はモデル化によって説明出来たが、図 6において問題状態の違いにより、問題空間が同じであっても台数効果に違いが生じている現象は説明出来ない。例えば、問題空間の大きさが約 $6 * 10^4$msec のところを見ると、初期状態 [E] の台数効果は 128 倍近く出ているのに比べ、問題 [A] の値は 40 倍程度にとどまっている。

ここで、各問題状態における探索の繰り返し回数を調べてみる。すなわち今あげた例では、問題空間の大きさが $6 * 10^4$msec の時点では問題 [E] は 7 回目の探索を行っているが、状態 [A] は 5 回目の探索を行っている。この繰り返し回数の違いは、そのままマルチレベル負荷分散のレベル数に反映される。(7 回目の探索では 6 レ

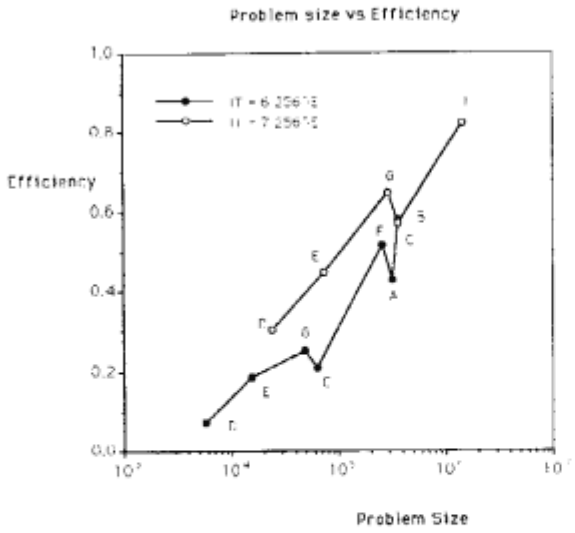

図 10: 負荷分散レベルを固定した場合の問題空間と効率の関係

ベル、5回目の探索では4レベルの負荷分散が実行される。)問題空間の大きさが同じであっても、負荷分散レベルが違うために台数効果に差が生じているのではないかと推測される。

そこで、繰り返し回数が同じであるデータを集めて問題空間と効率の関係を調べたのが図10である。

図10から、問題空間の大きさが同じであっても探索の繰り返し回数が違っていると効率に差が出る傾向が読みとれる。すなわち、問題空間が同じであっても、負荷分散レベル数が異なることが原因となって効率の値に差が生じ、図6のような結果となったと言える。

この負荷分散レベル数を適切に設定することにより効率をより良くする事が可能と考えられる。しかし今回用いた負荷分散戦略では、負荷分散レベル数は自動的に設定されるパラメータであるから調整は不可能である。この負荷分散レベル数がどの程度適切に設定されているかが、今回実験を行った負荷分散戦略の性能を評価することになるであろう。

負荷分散レベル数を固定した場合も、問題空間の大きさと効率との関係のグラフはシグモイド関数の形になることが期待される。しかし図10によると、結果にばらつきが見られる。これは探索木の構造に基づく負荷分散の効率の違いが影響しているのではないかと推測される。

5 おわりに

今回実験したIDA*アルゴリズムの並列化方式と負荷分散戦略は、256台のプロセッサを用いて最大232倍の高速化を達成する事が出来た。また、実行過程において繰り返される探索動作を分析することにより、問題空間の大きさと効率の関係を知ることが出来た。さらに今回用いた負荷分散戦略では、探索過程で自動的に決定される負荷分散レベルが性能に影響していることが明らかになった。

今後は、並列化におけるオーバヘッド要因のさらなる分析によって、負荷分散の改良への手掛りを得たいと考えている。

謝辞

本研究において多くの助言をいただいた(財)新世代コンピュータ技術開発機構第7研究室の木村宏一氏、三菱電機株式会社情報電子研究所の古市昌一氏に深く感謝いたします。

参考文献

[1] Richard E. Korf: "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence* Vol.27 No.1, pp.97–109 (1985).

[2] K. Ueda and T. Chikayama: "Design of the Kernel Language for the Parallel Inference Machine", *The Computer Journal* Vol.33 No.6, pp. 494–500 (1990).

[3] 中島 浩, 武田 保孝, 中島 克人: "PIM/m 要素プロセッサのアーキテクチャ", 並列処理シンポジウム JSPP'90, pp. 145–151 (1990).

[4] H. Nakashima, K. Nakajima S. Kondoh, Y. Takeda and K. Masuda: "Architecture and Implementation of PIM/m", *Proc. Conf. Fifth Generation Computer Systems 1992.*

[5] Nils J. Nilsson: "Principles of Artificial Intelligence", *Morgan Kaufmann Publishers, Inc.*, pp. 76–81 (1980).

[6] M. Furuichi, K. Taki and N. Ichiyoshi: "A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI", *Proc. Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp. 50–59 (1990).

[7] Kazuo Taki: "The Parallel Software Research and Development Tool: Multi-PSI System", *Programming of Future Generation Computers*, K. Fuchi and M. Nivat (eds.), Elsevier Science Publishers, pp. 411–426 (1988).

[8] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama: "Distributed Implementation of KL1 on the Multi-PSI/V2", *Proc. 6th International Conference on Logic Programming*, pp. 436–451 (1989).