Iterative-Deepening A* アルゴリズムのスタック分割動的負荷分散方式による並列化と並列推論マシン PIM/m 上の性能評価

和田 正寛
(財) 新世代コンピュータ技術開発機構

市吉 伸行
(株) 三菱総合研究所

六沢 一昭
沖電気工業 (株)

Iterative-Deepening A* (IDA*) アルゴリズムの並列化を試み、性能を評価した。IDA* アルゴリズムは、探索打切り深さを徐々に深めながら目標節に達するまで深さ優先探索を繰り返す木探索アルゴリズムであり、深さ優先探索と幅優先探索の欠点を補い合うものである。各イテレーションにおいて、探索途中状態を表現するスタックを分割することによって、負荷をプロセッサに分配する。スタックの空になったプロセッサは他のプロセッサに仕事を要求し、要求されたプロセッサは自分のスタックを2分割して半分を分け与えることによって動的に負荷が均等化される（スタック分割動的負荷バランス方式）。
この並列 Iterative-Deepening A* (IDA*) アルゴリズムを並列論理型言語 KL1 で記述し、例題として 15 パズル問題を並列推論マシン PIM/m 上で解いてみたところ、プロセッサ 256 台で最大 192 倍の高速化が得られた。またこの並列化方式の性能を著者らが先に実装した f 値部分木分割方式と比較して分析を行った。

An Implementation of Parallel Iterative-Deepening A* using Stack-splitting Dynamic Load Balancer and Its Evaluation on the Parallel Inference Machine PIM/m

Masahiro WADA
ICOT

Nobuyuki ICHIYOSHI
Mitsubishi Research Institute

Kazuaki ROKUSAWA
Oki Electric Industry

A parallel Iterative-Deepening A* algorithm was implemented and the performance was evaluated. The Iterative-Deepening A* algorithm is a tree search algorithm in which the search depth limit is incrementally deepened until the goal node is reached. In each iteration, the stack representing the search state is split and distributed onto processors. When the stack on a processor becomes empty, the processor demands a new task to other processors in turn, and one of the processors splits its stack into two and gives one to the demanding processor, thus realizing dynamic load balancing (stack-splitting dynamic load balancing).
A parallel Iterative-Deepening A* algorithm was implemented in KL1 implemented and evaluated on the parallel inference machine PIM/m using the the Fifteen Puzzle as a sample problem. A maximum of 192-fold speedup was attained with 256 processors. The performance is compared with the f-value-based tree partitioning scheme the authors implemented.

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。しかし前者は、探索木の枝が無限に伸びている場合は解が存在しても有限時間内に解が発見出来ない可能性があり、さらに最適解を必要とする場合に適当でない。また後者は、途中で生成される節が指数関数的に増大する場合、記憶のための資源を使い果たしてしまって探索が継続出来なくなるという欠点がある。

深さ優先／幅優先の探索戦略を組み合わせ、それぞれの問題点を解消したのが Depth-First Iterative-Deepening (以後ID と略す)[1] である。ID は基本的に深さ優先探索であるが、探索深さに制限を設け、その範囲内に解が見つからなかった場合は深さを深くし、深さ優先探索を最初からやり直すという手法である。具体的なアルゴリズムを以下に示す。

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。

上記のアルゴリズムでは、深さに制限を設け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段ずつ増加させている。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考えられる。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 [2] を取り入れる。A* アルゴリズムとは、解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。ある節から解までの距離の目安としては、その節に達するまでに要したコスト g と、その節から解にいたるまでのコストを見積もった値 h との和 f を用いる。ただし、h の値をその節から解までの最短コストより大きく見積もってはならない。大きく見積もっ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)[1] の具体的なアルゴリズムは以下の通りである。

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 f 値部分木分割方式とマルチレベル動的負荷バランサ

システムの並列化を行う場合、二つの問題を考慮する必要がある。一つは、並列に実行するために「問題をどのように分割するか」という点であり、もう一つは、「分割した問題をどのようにプロセッサに割り付けるか」である。システムを動かす前にそのシステムの挙動が予測出来る場合は、問題分割もプロセッサ割り付けもシステムを動かす前に（静的に）設定することが出来る。しかし予想が出来ない場合は、システムを動かしながらその挙動に応じて（動的に）問題分割及びプロセッサ割り付けを行う必要がある。

文献 [4] は “f 値部分木分割方式” によって問題を分割し、“マルチレベル動的負荷バランサ” を用いて部分問題を割り付ける並列実行方式を述べている。以下、それぞれについて説明する。

f 値部分木分割方式

IDA* アルゴリズムは、探索を繰り返すごとにその時点での閾値を越えた f 値の最小値を返す。この値を利用して部分木分割を実現する方式である。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 $T = 0, t_2, t_3, \cdots t_I$ となる場合を考えると、探索回数が I である時、f 値が $t_2, \cdots t_I$ となる節点で部分木に分割するというアルゴリズムである。

マルチレベル動的負荷バランサ

動的なプロセッサ割り付け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 [3, 7](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付け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処理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。負荷バランサは処理の終了したプロセッサグループを処理中のプロセッサグループに合流させることにより負荷の均一化を図っている。このしくみをグループマージと呼んでいる。

以後、f値部分木分割方式とマルチレベル動的負荷バランサを利用した並列実行方式をftp方式と呼ぶ。

3 スタック分割動的負荷分散方式

今回新たに実験を行ったのは、スタック分割動的負荷分散方式（以後、STB方式と略す）[5, 6, 7]である。STB方式は、各プロセッサ上にスタックを作成する。また各プロセッサ上には、スタックから取り出された節（以後、節データと呼ぶ）が解であるかどうかを調べ、解であった場合は解の報告を行い、解でなかった場合はその節データから探索木における次の深さの節データを生成する（以後、展開すると呼ぶ）という動作を行う問題解決部が置かれる。一つの節データから新たに展開された節データはスタックに積まれる。スタックの一番上にある節データを取りだし、問題解決部によってその節データを展開し、新たな節データをスタックに積む、この一連の動作を繰り返すことによって探索が継続される。

自プロセッサのスタックが空になった場合、他のプロセッサにスタック内の節データを分け与えてくれるように要求を出す。要求を受け取ったプロセッサは、自分のスタックに十分な節データがある場合、節データを分け与える。この方式により問題分割とプロセッサ割り当ての両方が実現出来、並列実行が行われる。

STB方式を用いる場合、探索を行う前に、各プロセッサ上にスタックと問題解決部、さらにスタックを管理して他のプロセッサとメッセージの交換を行うプロセスを生成しなければならない。これらを生成する過程をプロセスネットワーク生成過程と呼ぶ。

STB方式は以上の通りであるが、実際にプログラムを作成し実行する場合、効率の上で以下のような問題が存在する。

節データの要求先の決定

まず、スタックが空になったプロセッサが<u>どのプロセッサに節データを要求するか</u>という問題である。たとえば、スタックに持つ節データが最も多いプロセッサに要求を出す方法が考えられるが、それを実現するには、すべてのプロセッサにおけるスタックの状態を常に管理している必要があり、そのための処理が大きな負荷になってしまう。そこで、とりあえずいずれかのプロセッサに要求を出し、データを分けて貰えなかった場合は他のプロセッサに要求を出すという方式をとる。その場合にも、どのプロセッサに要求を出すかという問題が残る。戦略としては、隣のプロセッサに出す等様々な方式が考えられるが、現在は乱数により要求先を決めている。

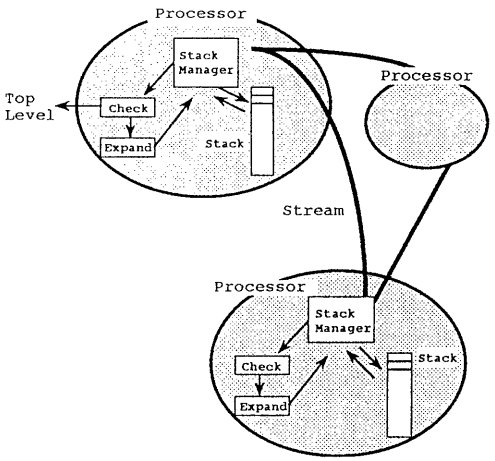

図 1: スタック分割動的負荷分散方式

節データの分け与えの可否の決定

次に、要求を受け取ったプロセッサが、自分自身のスタックに<u>どれだけの節データが有る場合に分け与えを行うか</u>という問題である。スタックに積まれている節データの量は、そのプロセッサに割り当てられた仕事の量と考える事が出来る。一般に、自分自身もあまりたくさん仕事を持っていない場合に仕事を分け与えてしまうと、やがて自分自身の仕事をし尽くしてしまい、今度は要求を出す側にまわってしまう。その結果、プロセッサ間で飛び交う要求の頻度が増加し、要求処理の増大のために全体の効率が低下してしまう。よってスタックにどれだけの節データがあった場合に分け与えを実行するかはSTB方式の効率を向上させる上で重要な問題であるが、現時点では分け与えを実行するデータ量の適切な目安は明らかではない。今回の実験では、スタックに積まれている節データ数がある値以上であるかどうかで分け与えの判定を行った。

節データの分け方

また、要求を分け与える場合、<u>節データをどのように分けるか</u>も効率の上で問題となる。この問題については、分けることによって生じる二つの部分木の持つ計算量をなるべく均等化するという意味から、スタック内の節データを一つおきに取り出すヒューリスティックを用いている[7]。

4 15パズル問題

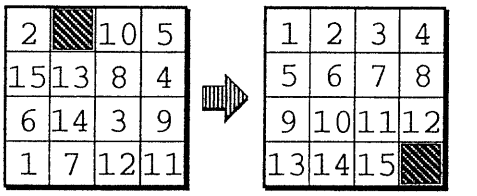

図 2: 15パズル問題

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	繰り返し探索回数	全状態数
[A]	43	6	262,019
[B]	43	6	295,216
[C]	44	7	295,253
[D]	37	10	1,526,583
[E]	41	10	5,885,472
[F]	42	7	1,167,105
[G]	43	8	1,298,208

問題	特徴
[A]	無作為に選んだ状態
[B]	[A] と探索深さが同じ状態
[C]	[A] からピースを一つ動かした状態
[D]	[A] よりも探索深さは浅いが繰り返し探索回数が多い状態
[E]	[A] よりも全状態数を大きくした状態
[F]	[A] と探索深さは同程度で全状態数を大きくした状態
[G]	[F] からピースを一つ動かした状態

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、問題サイズの目安となる。

5 実験と考察

5.1 STB方式による負荷分散

並列推論マシンPIM/m[8, 9]を用いてSTB方式による負荷分散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。

表 2: プロセッサ256台を用いたSTB方式の台数効果

問題	1PEでの実行時間(秒)	256PEでの実行時間(秒)	台数効果(倍)
[A]	411.2	4.8	85.7
[B]	451.3	4.5	100.3
[C]	462.7	4.9	94.4
[D]	2,591.4	17.9	144.8
[E]	9,706.6	50.6	191.8
[F]	1,849.8	12.7	145.7
[G]	2,049.2	13.7	149.6

プロセッサを256台用いる事により、[E]において192倍の台数効果を得る事が出来た。一方で[A]や[C]の台数効果は100倍以下に留まっているが、その原因は問題サイズが小さいためであろう。

表2中の実行時間には、プロセスネットワーク生成のための時間を含んでいない。STB方式そのものの性能を評価する上ではプロセスネットワーク生成時間を省くのは適切でないが、今回はSTB方式における負荷分散実行部のみの評価を行った。

表 3: プロセッサ台数とプロセスネットワーク生成時間

台数	生成時間(秒)
8	0.01
16	0.02
32	0.06
64	0.17
128	0.54
256	1.75

参考までに、現時点でのシステムにおいてプロセスネットワーク生成部に要している時間を表3に示す。プロセスネットワーク生成のための時間は問題にかかわらず一定であり、プロセッサ台数の約1.7乗に比例して増加している。また、現在の仕様では繰り返し探索ごとにプロセスネットワークを作り直す必要があるため、そのために必要な時間は無視できない大きなものとなっている(例えば問題[D]をプロセッサ256台用いて解く場合、探索そのものは144秒で終了する

が、プロセスネットワーク生成時間は 1.75 秒 ×10 回で約 18 秒にもなる。）。ただし、ネットワーク生成の方式を改善することによって、生成時間がプロセッサ台数 p のほぼ $p\log p$ に比例し、256 台で 0.5 秒程度に短縮出来ることがわかっている。

5.2 ftp 方式との性能比較

台数効果の比較

同じ問題を STB 方式を用いて解いた場合と ftp 方式を用いた場合の比較を行う。表 4 に、プロセッサを 1 台用いた場合でのそれぞれの方式の実行時間を示す。

表 4: プロセッサ 1 台での実行時間

問題	STB(秒)	ftp(秒)	比 (STB/ftp)
[A]	411.2	390.4	1.05
[B]	451.3	423.9	1.06
[C]	462.7	437.5	1.06
[D]	2,591.4	2,542.3	1.02
[E]	9,706.6	9,463.5	1.03
[F]	1,849.8	1,753.2	1.06
[G]	2,049.2	1,949.2	1.05

プロセッサ 1 台での実行時間を見てみると、STB 方式と ftp 方式の間に大きな違いはない。このことから、両方式の性能を台数効果で比較する事が有意であると言える。

表 5 は、プロセッサを 256 台用いた場合でのそれぞれの方式の実行時間と台数効果を示している。

表 5: プロセッサ 256 台での実行時間と台数効果

問題	実行時間 (秒)			台数効果 (倍)		
	STB	ftp	比	STB	ftp	比
[A]	4.8	8.4	0.57	85.7	46.5	1.84
[B]	4.5	5.7	0.79	100.3	74.4	1.35
[C]	4.9	7.4	0.66	94.4	59.1	1.60
[D]	17.9	12.7	1.41	144.8	200.2	0.72
[E]	50.6	40.7	1.24	191.8	232.5	0.82
[F]	12.7	11.8	1.08	145.7	148.6	0.98
[G]	13.7	11.7	1.17	149.6	166.6	0.90

台数効果を比較してみると、両方式に明確な違いが見られる。問題 [A] などでは STB 方式の方が良い結果を示しているが、問題 [E] などでは逆に ftp 方式の方が好結果を示している。表 5 から、問題サイズが大きくなると STB 方式よりも ftp 方式の方が台数効果が良くなる傾向が見られる。

効率の比較

次に、使用するプロセッサ台数を変えて実験を行う。台数効果の値は使用するプロセッサの数に依存するため、プロセッサ台数を変えた場合の比較には不適当である。そこで台数効果の値を、使用したプロセッサの数で割った値を用いて比較を行う。この値を効率と呼び、理想的な並列処理においては効率の値は 1 となる。プロセッサ台数と効率の関係を、代表的な問題状態 [A], [E], [F] を用いて測定した結果が図 3 である。

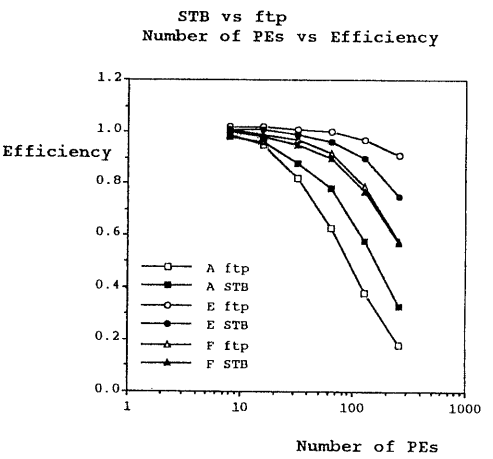

図 3: プロセッサ台数と効率

図 3 によると、いずれの問題もプロセッサ台数の増加につれて効率が低下している。問題 [A] においては ftp 方式の方が効率低下が大きいが、問題 [E] においては STB 方式の方が効率低下が大きい。効率が低下するのは、プロセッサ台数の増加につれて全体の稼働率が低下する事、並列処理のためのオーバヘッドが増加する事等が原因と推測される。また負荷分散方式の違いによる効率の差は、問題サイズと関係あるのではないかと思われる。

問題サイズと台数効果の関係の比較

効率低下の原因の検討を行う前に、問題サイズと台数効果との関係を分析する。前節までの分析においては、実行時間は、問題を解き始めてから解が求まるまでの時間全体を用いていた。IDA* アルゴリズムは、探索領域を拡大しながら探索動作を繰り返すものであるが、各繰り返し探索を単独で取り出してみると、それぞれが独立した探索動作であると見成す事が出来る。さらに繰り返しごとに探索サイズが大きくなっていく事から、より多くの状態に対して分析する事が出来る。

例えば問題 [A] をプロセッサ 256 台用いて解いた場合、繰り返し探索ごとの問題サイズ (探索木の節の総数で表現する) と台数効果の関係は表 6 のようになった。

表 6: STB 方式における繰り返し探索ごとの台数効果 (問題 [A])

繰り返し探索回数	節数	1PE (秒)	256PE (秒)	台数効果
1 回目	1	0.00	0.03	0.0
2 回目	172	0.22	0.16	1.4
3 回目	1,112	1.43	0.30	4.8
4 回目	7,678	9.93	0.57	17.4
5 回目	46,099	59.73	0.94	63.5
6 回目	262,019	339.89	2.86	118.8

問題サイズが大きくなるに従って台数効果が向上しているのが表 6 から明らかである。この分析を全ての問題に対して行い、結果をグラフに表したのが図 4 である。

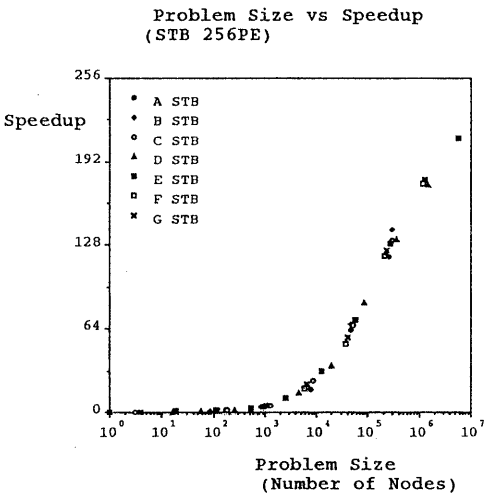

図 4: 問題サイズと台数効果 (STB 方式, 256PE)

ftp 方式による実行では、問題サイズの対数と台数効果との間にはシグモイド関数 (hyperbolic tangent) の関係が見られた [4] が、STB 方式を用いた場合も図 4 に示すように同様の関係が見られる。

図 5 に、ftp 方式を用いた場合の問題サイズと台数効果の関係を示す。ftp 方式では問題サイズと台数効果の組によって示される点の分布に広がりが見られるが、STB 方式ではほとんどすべての点が一本の曲線上に乗っている。

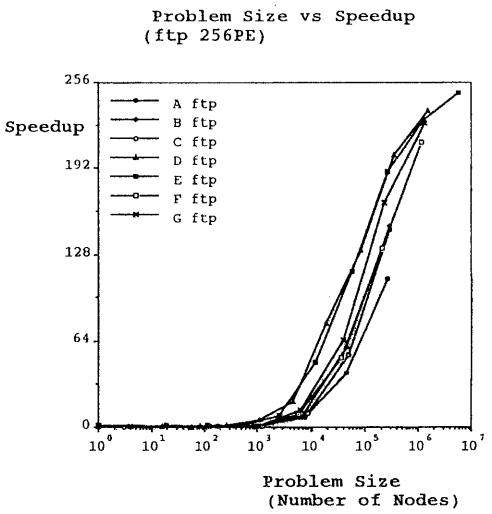

図 5: 問題サイズと台数効果 (ftp 方式, 256PE)

ftp 方式において曲線に幅が生じたのは、問題によって負荷分散レベル数に差が生じ、それが台数効果の差となって現れたことがわかっている [4]。STB 方式においてはそのような現象は起きないため、問題の大きさのみによって台数効果が決まるものと考えられる。

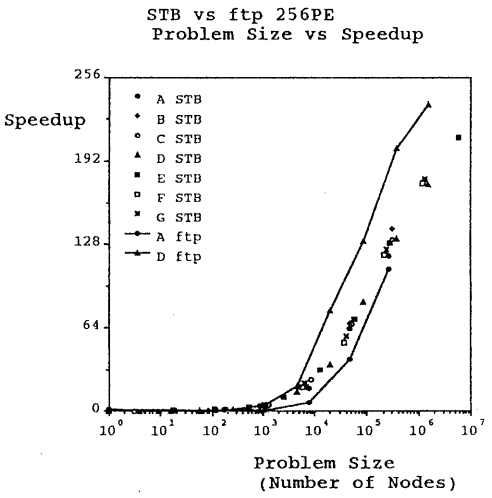

図 6: STB 方式と ftp 方式の台数効果比較 (256PE)

STB 方式での結果を、ftp 方式を用いた場合の最良結果および最悪結果と比較したのが図 6 である。STB 方式は、ftp 方式の最良結果と最悪結果の間にあることがわかる。

プロセッサの稼働状態の比較

同じ台数のプロセッサを使用した場合に、STB 方式と ftp 方式に台数効果の差が生じる原因を、プロセッサの稼働状態から分析する。256 台のプロセッサを使用して問題 [F] を解いた場合のプロセッサの稼働状態を見てみる。

問題 [F] の探索において、最終回の繰り返し探索の実験結果は表 7 のようになる。

表 7: 問題 [F] の最終繰り返し探索における台数効果 (256PE)

STB 方式		ftp 方式	
実行時間	台数効果	実行時間	台数効果
8.68	174.9	6.83	210.8

問題 [F] の最終繰り返し探索は、今回用いた問題の中で問題サイズの大きい部類に入る (問題 [F] の最終繰り返し探索において生成される節数は 1,167,105 である。)。ftp 方式を用いた場合の台数効果はかなり良い値 (効率にして 0.82) が得られているのに比べ、STB 方式では低い値にとどまっている。

探索実行中のプロセッサ稼働状態を図 7 に示す[1]。図 7 は、縦軸に 256 台のプロセッサのある時点での稼働状態を表している (正確には 8 台のプロセッサが 1 組となって表されている。)。各桝目の色が濃いほど、その時点でそのプロセッサが忙しく稼働していることを表す。横軸は時間の経過を示す (一目盛りは 200 ミリ秒)。

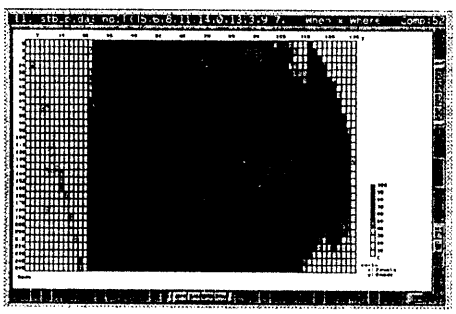

図 7: 問題 [F] の最終繰り返し探索における稼働状態 (STB 方式)

最初の約 2 秒は稼働状態が著しく悪いが、これはプロセスネットワーク生成過程であり、台数効果の測定上は省いているので無視する。探索実行中は、ほとんどのプロセッサが 80% 以上の忙しさで稼働しており、負荷分散状態は悪くはない。

[1]図 7, 図 9 等は、PIMOS ユーティリティの一つである ParaGraph によって得られた図である。

探索実行時間の長さから、探索動作は図 7 における時刻 115 の位置で終了していると考えられる。しかし図 7 では、探索終了後にプロセッサが再び忙しく稼働しはじめ、その後約 2 秒間にプロセッサは順々に動作を終了している。この現象は、プロセスネットワークを消去するための処理であり、負荷は不均一になっているが、この過程も計測時間に含まれていない。

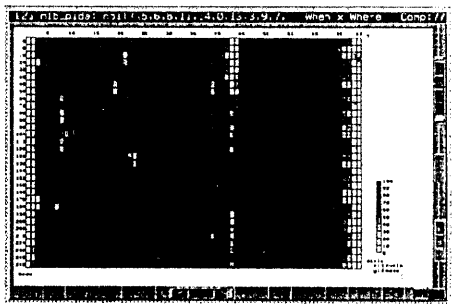

図 8: 問題 [F] の最終繰り返し探索における稼働状態 (ftp 方式)

比較のために、同じ問題を ftp 方式で解いた時のプロセッサ稼働状態を図 8 に示す (一目盛りは 100 ミリ秒)。これらの図から、STB 方式は ftp 方式に比べて負荷はほぼ均一であり、稼働状態も悪くないことがわかる。ゆえに ftp 方式に比べて STB 方式において台数効果が低くなるのは他の要因であると結論される。

プロセッサ間通信の検討

次に、プロセッサが行う処理のうち、通信のための処理がどれだけの割合を占めているかを調べる。図 9 は、横軸は時間の経過を示し、縦軸は各時刻における処理のうち、実計算 (compute) と通信 (send, receive) の占める割合を示している (グラフの値は、全ての使用プロセッサの平均値で表されている。)。グラフは compute, send, receive それぞれの値を積み重ねて表現している。例えば時刻 40 の位置 (実行開始後 4 秒を示す) では、各処理の大きさは compute に約 80%、send に約 3%、receive に約 5% であることがわかる。探索実行中は、処理の約 80% を実計算に、約 10% を通信に費やしていて、ほぼ一定の値を保っている。動作終了前の約 2 秒間に実計算の比率が大きく低下し通信の比率が増大しているが、これはプロセスネットワーク消去過程に起因する現象である。

同じ問題を ftp 方式で解いた時の結果を図 10 に示す。まず実計算に着目すると、比率が 80% から 90% になっており、STB 方式よりも高い値を示している。しかし比率に変動が見られ、特に時刻 40 の位置 (実行開始後 4 秒) では大幅に低下している。これは探索木のアンバランスに起因する負荷のばらつきが原因で

あると推測される。このような現象はSTB方式には見られない。次に通信に着目すると、最大でも約8%でそれ以下に留まっている。確かにSTB方式の方が通信の比率は高いと言える。

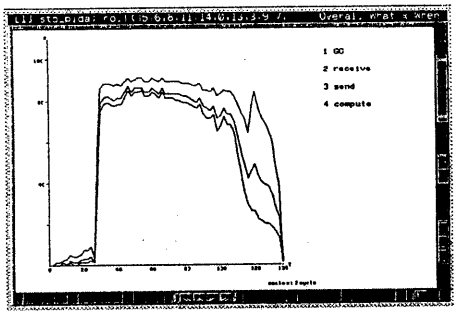

図 9: 問題 [F] の最終繰り返し探索における処理内容 (STB 方式)

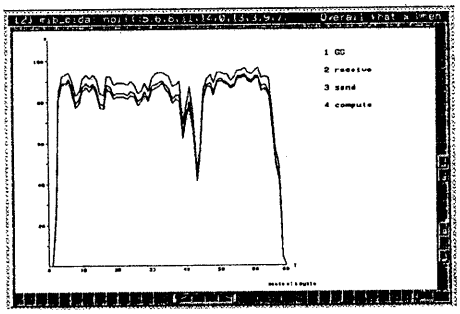

図 10: 問題 [F] の最終繰り返し探索における処理内容 (ftp 方式)

6 おわりに

今回実験したSTB方式では、256台のプロセッサを用いて最大192倍の高速化を達成する事が出来た。また、問題サイズに対して一定の台数効果が得られることがわかった。ftp方式と比べて最高成績は劣るが、これはSTB方式の方がプロセッサ間通信量が多く、それが性能を低下させる原因になっていると考えられる。

今後は、ネットワーク生成部及び消去部の改善によるモジュールとしての性能向上、節データの分け与え戦略の検討を行いたいと考えている。また、STB方式とftp方式のいずれにおいても、問題サイズと台数効果との間には同じ傾向が見られたが、これが一般的な問題においても見られる現象であるかを実験して調べたい。

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", *Aritificial Intelligence* Vol.27 No.1, pp.97–109 (1985).

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

[3] 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).

[4] 和田 正寛, 市吉 伸行, 六沢 一昭: "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", 並列処理シンポジウム JSPP'92, pp. 85–91 (1992).

[5] V. Kumar and V. N. Rao. "Scalable Parallel Formulations of Depth-First Search" In *Parallel Algorithms in Machine Intelligence and Vision*, SpringerVerlag, pp. 1–41 (1990).

[6] 古市昌一, 中島克人, 中島浩, 市吉伸行: "スタック分割動的負荷分散方式とマルチ PSI 上での評価". 1991年 並列/分散/協調処理に関する「大沼」サマー・ワークショプ, コンピュータシステム研究会, pp.33–40 (1991).

[7] 古市 昌一: "負荷バランスユーティリティマニュアル", ICOT Technical Memorandum (1992).

[8] 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).

[9] H. Nakashima, K. Nakajima S. Kondoh, Y. Takeda and K. Masuda: "Architecture and Implementation of PIM/m", *Proc. Conf. Fifth Generation Computer Systems*, pp. 425–435 (1992).