

動的負荷分散のための並列木探索 (DTS) アルゴリズムの 拡張

— 並列計算機 AP1000 でのライブラリ化 —

大野 和彦[†], 森 真一郎[†], 中島 浩[†], 富田 眞治[†]

[†] : 京都大学 工学部

疎結合並列計算機上で木探索を行なう際に問題となるプロセッサ間の木構造管理や動的負荷分散機能の実現方法として、並列木探索 (DTS) アルゴリズムを基にした拡張並列木探索 (EDTS) アルゴリズムを提案する。本アルゴリズムは様々な木探索問題や動的負荷分散方式を実現可能であり、これをライブラリ化することにより、木探索問題一般を対象とした言語環境として利用できる。並列計算機 AP1000 上での実装と性能評価の結果、このライブラリが実行速度・台数拡張性ともに良好であることが示された。

Extension of Distributed Tree Search for Dynamic Load Balancing and Its Implementation as a Library on AP1000

Kazuhiko OHNO[†], Shin-ichiro MORI[†], Hiroshi NAKASHIMA[†], Shinji TOMITA[†]

[†] : Department of Information Science
Faculty of Engineering, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto 606-01 Japan

E-mail: {ohno, moris, nakasima, tomita}@kuis.kyoto-u.ac.jp

For programmers of tree searching applications on loosely-coupled multiprocessors, it is troublesome to manage the tree structure spread across processors, and to balance the load of processors dynamically. As a solution of these problems, we propose EDTS, an extension of Distributed Tree Search algorithm. Various tree search applications and dynamic load balancing schemes can be realized with this algorithm, and the implementation as a library is useful as a programming environment for general tree search problems. The result of implementation and evaluation on multiprocessor AP1000 shows good performance and scalability of the library.

1 はじめに

疎結合並列計算機上でのプログラム作成では、アルゴリズムの並列化や通信・同期処理、適切な負荷分散などを実現しなければならない。このため、一般のユーザが効率のよいプログラムを作成するには、言語処理系による並列化の支援が不可欠である。

データ構造として配列を扱うプログラムでは、コンパイル時に分割方法を決定する自動並列化コンパイラがある程度の成果を挙げている。これに対し木やグラフといったデータ構造を扱うプログラムでは、実行時にデータ構造が動的に変化するため静的なデータ分割を行なうことは困難である。このため、木探索の並列化に関しては動的に負荷分散を行なう方式が試みられ、その幾つかでは良好な結果が得られている [2, 3]。しかし、これらの方法は特定のアプリケーションに組み込んだ形で動的負荷分散アルゴリズムの評価を行なったものであり、一般ユーザがアプリケーション作成の際に利用可能な形にはなっていない。

本稿では並列木探索 (DTS:Distributed Tree Search) アルゴリズム [1] を基に、汎用性・台数拡張性に優れた拡張並列木探索 (EDTS:Extended Distributed Tree Search) アルゴリズムを提案する。さらに EDTS アルゴリズムを用いて木構造の管理や動的負荷分散機能を実現する汎用の並列木探索ライブラリを、疎結合並列計算機 AP1000 上に実装した結果について述べる。

以下第 2 章では EDTS アルゴリズムについて、第 3 章では同アルゴリズムの AP1000 上におけるライブラリ化について述べる。ついで第 4 章ではこのライブラリを用いた 2 種類の負荷バランサー関数の実現について、また第 5 章では具体的な応用例に基づく性能評価について、それぞれ述べる。最後に第 6 章においてまとめを行なう。

2 拡張並列木探索 (EDTS) アルゴリズム

一般的な木探索問題は、次のように定義できる。

1. 親ノードが与えられた場合に、なんらかのアルゴリズムによって幾つかの子ノードが生成される。これを木またはノードの展開と呼ぶ。
2. 一定の条件下で終端ノードが現れ、そのノードから下は展開されない。
3. 特定の終端ノードが発見されるか、すべてのノードが展開された時点で探索は終了する。

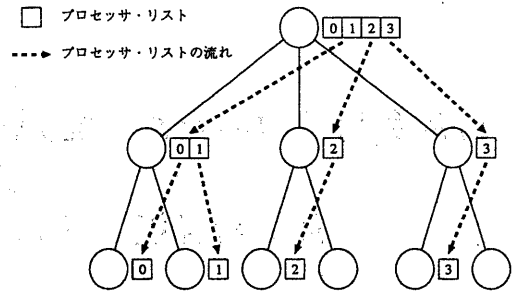


図 1: 木の展開

これを疎結合並列計算機で解く場合には、複数のノードを複数のプロセッサ上で同時に展開していくことにより探索の並列実行を行なうことになる。この場合問題となるのは、実行時に木が展開されていく過程において各プロセッサの仕事量を均等に調整することと、複数プロセッサ上に分散したノード間の接続状態を保持してそれらのノード間で必要な情報伝達を行なうことである。

2.1 並列木探索 (DTS) アルゴリズム

並列木探索 (DTS:Distributed Tree Search) アルゴリズム [1] では、木の各ノード毎にそれをルートとする部分木を探索するための並列プロセスが割り付けられる。また各ノードには利用可能なプロセッサの番号の並び (以下プロセス・リストと呼ぶ) が割り付けられ、これを用いて以下のように探索処理が行なわれる。

1. 木の展開 (図 1)

プロセスは次の処理を行なう。

- (a) 自身のノードを展開し、子ノードを生成する。
- (b) 自身に割り当てられたプロセス・リストを子ノードの個数以下に分割し、それぞれを子ノードに割り当てる。
- (c) プロセス・リストが割り当てられた子ノードには、プロセスのコピーを生成する (以下子プロセスと呼ぶ)。
- (d) 実行を中断 (ブロック) し、子ノードの探索終了を待つ。

子ノードに割り当てられた子プロセスは、それぞれが同様に子孫ノードを生成し、展開を続けていく。ただし、終端ノードに割り当てられたプロセスは子プロセスを生成せず、直ちに 2. に移る。

2. 探索の終了

子プロセスの探索がすべて終了したプロセス

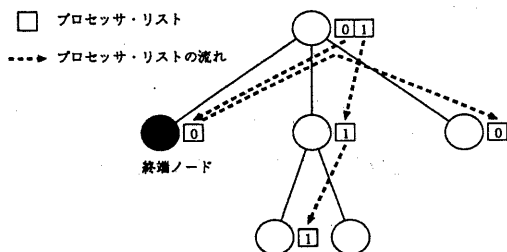


図 2: 子プロセスからの通信

または終端ノードのプロセスは、自身の親ノードに割り当てられたプロセス (以下親プロセスと呼ぶ) に探索結果 (探索の成否、ミニマックス値など) および割り当てられたプロセッサ・リストを送信 (返却) して終了する。

3. 子プロセスからの通信 (図 2)

ブロックしているプロセスが子プロセスからの通信を受けとった時は次の処理を行なう。

- (a) 探索未終了の子ノードが残っていないければ 2. に移る。
- (b) 子プロセスより返却されたプロセッサ・リストを探索未終了の子ノード個数以下に分割し、各々をそれらのすべてであるいは一部に、以下のように割り当てる。
 - i. 割当対象となった子ノードに子プロセスを既に割り付けていれば、それによりたいしてプロセッサ・リストを送信 (追加) する。
 - ii. 割当対象となった子ノードに子プロセスを割り付けていなければ、1. と同様に子プロセスを生成する。

4. 親プロセスからの通信

ブロックしているプロセスが親プロセスからの通信を受けとった時は、親プロセスより追加されたプロセッサ・リストを 3b と同様にして子プロセスに割り当てる。

ルート・ノードにプロセスを生成し、利用可能な全プロセッサをプロセッサ・リストとして割り当てることによって探索が開始される。ルート・ノードのプロセスが終了した時点で、探索は終了する。

物理的にはノードのデータおよびプロセスは、そのノードに割り当てられたプロセッサ・リストの先頭プロセッサ上に存在する (図 3)。プロセッサ・リストはプロセッサに対する仕事割当の権利を示し、親ノードのプロセスは自身に割り当てられたプロセッサ・リストに含まれるプロセッサに対して、仕事 (子ノード) を割り当てることできる。具体的には、分割した各プロセッサ・リストの先頭

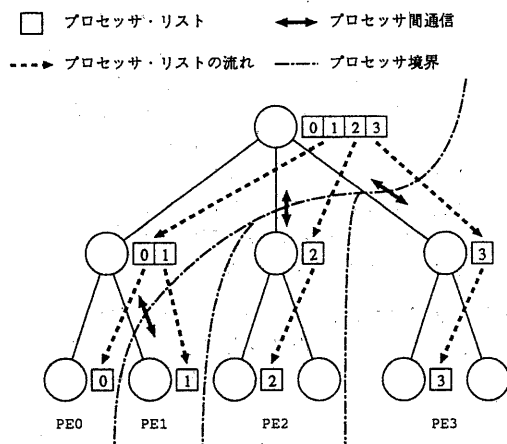


図 3: 木の物理的配置

プロセッサに、ノードのデータおよびプロセッサ・リストを送信し、子プロセスを生成する。

なお、親ノードが存在するプロセッサ自身も分割されたプロセッサ・リストの一つにおいて先頭プロセッサとなっているから、このプロセッサ上では子プロセスの一つが生成されるのと入れ違いに親プロセスがブロックする。したがって、分割が進むと最終的にプロセッサ・リストはそのプロセスが存在するプロセッサのみとなり、これ以降は同一プロセッサ上に次々に子プロセスを生成することによって逐次の深さ優先探索が行なわれる。

2.2 DTS アルゴリズムの拡張

DTS アルゴリズムは動的負荷分散を行ないながら複数プロセッサ上に木構造を展開していく機構を提供しているに過ぎず、具体的な木の展開方法や負荷分散戦略は規定していない。このため様々な木探索アルゴリズムや負荷分散戦略に対応でき、これをライブラリ化した場合、木探索全般への利用が期待できる。また、仕事の割当を行なうプロセッサが固定されていないので、適切な負荷分散戦略を用いればボトルネックとなるプロセッサは生じず、プロセッサ台数の拡張性も期待できる。

しかし本アルゴリズムで行なわれるプロセッサ・リストの分割・割当による動的負荷分散機構は、一般的な負荷分散戦略を実現する上で以下のような問題点がある。

1. 探索未終了の子ノードがあれば返却・追加されたプロセッサは必ずそれらに割り当てられるため、残りの仕事量が少ない時は分割により粒度が小さくなり効率が低下する。
2. 自身が存在するプロセッサは必ず子プロセス

のいずれかに使用されるため、木はプロセッサ間で縦方向に分割される。木のどの部分に重点的にプロセッサ・リストを投入するかにより分割の割合は変更できるが、木を横方向に分割するような負荷分散戦略は実現できない。

3. 未割当のプロセッサ・リストは直ちに割当を決定されなければならないため、他のノードの探索結果が得られるまで仕事割りつけを遅らせるようなアルゴリズムは実現できない。

そこで2.1節で示したアルゴリズムのステップ1, 3, 4における、未割当のプロセッサ・リストが発生したときの分割・割当方法を、以下のように変更する。

1. プロセッサ・リストを任意の個数に分割する。
2. 分割された各リストについて、次のいずれかを選択する。
 - (a) 親ノードに返却する。
 - (b) 探索未終了の子ノードのいずれかに割り当てる。
 - (c) そのノード、あるいはそのノードが存在するプロセッサが保持する。

2c については次に分割・割当を行なう時に、
1. で分割されたリストの一つとして扱う。

これにより、前記したDTS アルゴリズムの問題点はそれぞれ次のように解決できる。

1. 探索未終了の子ノードが残されていても残りの仕事量が少ない時は、親ノードにプロセッサ・リストの一部を返却することにより、より粒度の大きい問題にプロセッサを振り向けることができる。
2. 探索が一定の深さに到達したら自身が存在するプロセッサを含むプロセッサ・リストを親ノードに返却することにより、特定のプロセッサが特定の深さを探索することができ、木の横方向分割が可能になる。
3. プロセッサ・リストの一部をしばらく保持することで、そのプロセッサへの仕事割りつけを遅らせることができ、一部ノードの探索結果により枝刈りを行なうまでプロセッサを待たせておくようなアルゴリズムが可能になる。

また、変更後のアルゴリズムにおいて変更前のアルゴリズムは表現可能であるから、前者はもとのDTS アルゴリズムの拡張になっているといえる。そこでこのアルゴリズムを拡張並列木探索(EDTS:Extended Distributed Tree Search) アルゴリズムと呼ぶことにする。

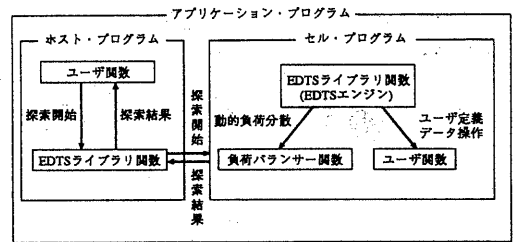


図 4: アプリケーション・プログラムの構成

3 ライブラリの実装

3.1 AP1000 のプログラミング環境

AP1000 は富士通研究所で開発された疎結合並列計算機であり、ホスト計算機と最大1024のセル(プロセッシング・エレメント)から構成される[4]。ユーザはホストおよびセル用のプログラムを作成し、ホスト・プログラム中でセル上にタスク生成を行なうことにより、セル・プログラムを各セル上で並列実行させることができる。言語はFORTRANとCが用意され、通信・同期などの機能はライブラリ関数として提供されている[5]。

3.2 EDTS アルゴリズムのライブラリ化

このAP1000上のC言語環境において、2章で述べたEDTS アルゴリズムを、汎用の並列木探索ライブラリとして設計・実装した。以後これをEDTSライブラリと呼ぶ。

ライブラリ関数は複数プロセッサ間での木構造の展開と保持、動的負荷分散などの機能を実現する。実際に特定の木探索問題を解くためにはこれに加え、ノードや探索結果の内容、子ノードの作成方法などその問題固有のデータやアルゴリズムの定義が必要となる。本ライブラリを用いたアプリケーションの作成とは、ユーザがこれら进行处理するための関数を記述することである。以後これらの関数をユーザ関数、ユーザ関数に処理される問題固有のデータをユーザ定義データと呼ぶ。ユーザ関数をライブラリ関数とリンクすることにより、完成した並列木探索アプリケーションとなる(図4)。

3.2.1 ホスト・プログラム

本ライブラリを用いた並列木探索アプリケーション・プログラムでは、実際の木探索は後述するEDTS エンジンによりセル上で実行される。ホスト・プログラム用のライブラリは、このEDTS エンジンを生成・起動・終了させる関数からなる。

ホスト用のユーザ関数はユーザ定義データを初期化し、ライブラリ関数を用いて EDTS エンジンによる並列木探索を行ない、探索結果のユーザ定義データを出力する。

3.2.2 セル・プログラム

セル・プログラム用のライブラリは、EDTS アルゴリズムを実現する EDTS エンジンが 1 つの関数として用意されている。セル用のユーザ関数はこの EDTS エンジンより適時呼び出され、アプリケーションに依存するデータやアルゴリズムの処理を行なう。

EDTS エンジン EDTS アルゴリズムは 2.1 節で述べたように多数のプロセス生成により実現されるが、今回は実行効率などを考え、各セル上で EDTS エンジンと呼ばれる一種のオートマトンを 1 個ずつ動作させることで実現した。これにより 2.1 節でのプロセス間の通信は、プロセッサ間通信または 1 プロセッサ上における EDTS エンジン内での処理となる。また AP1000 ではプロセッサをセルと呼んでいるため、以後プロセッサ・リストはセル・リストと呼ぶことにする。

EDTS アルゴリズムでは多数のプロセスが生成されるが、すべて同一プロセスのコピーであり、機能は同じである。また、1 プロセッサ上で同時に動作するプロセスは高々 1 個であり、その間他のプロセスはブロックしている。

EDTS エンジンでは、動作中のプロセス番号に相当するものを内部状態として保持し、1 プロセッサ上の全プロセスの機能を果たす。各ノード間の接続状態や割り当てられたセル・リストなどは、構造体のリスト構造として保持される。ノード間通信は、EDTS エンジンが送信ノードの状態において送信データを作成し、このリスト構造を辿って受信ノードの状態に遷移し、これを受信データとして処理することで実現される。受信ノードが他のプロセッサ上にある時には、プロセッサ間通信によりデータを実際に送信する。受信側プロセッサの EDTS エンジンは受信ノードの状態に遷移し、これを処理する。

負荷バランサー関数 EDTS エンジン内で未割当のセル・リストが発生すると、ユーザが用意した負荷バランサー関数を呼び出し、割当を決定する。この関数の機能は引数として渡されたセル・リストを分割して任意個数のセル・リストを作り、各々について、親ノードに返却/子ノードの 1 つに割当/次にこの関数が呼び出されるまで保持、のいずれを行なうかを決定することである。この決定アル

ゴリズムを記述することにより、様々な負荷分散戦略を実現できる。

ユーザ関数 EDTS エンジンではユーザ定義データを操作する必要が生じた時、ユーザ関数を呼び出す。ユーザ関数には次のようなものがあり、引数として渡されたノード番号のユーザ定義データに対して処理を行なうため、木の構造やプロセッサ間の分割状態などを考慮して記述する必要がある。

- **子ノード展開関数**

指定されたノードのユーザ定義データより子ノードを生成し、それらのユーザ定義データを作成する。並列探索を可能にするため、ある親ノードに対するすべての子ノードはこの関数により一度に生成され、その後負荷バランサー関数により適時プロセッサに割り付けられる。また、終端ノードであるかの判定も、ここで行なわれる。

- **探索結果作成関数**

終端ノードが発見された時、この関数により探索結果として親ノードに返すユーザ定義データを作成する。また結果の伝達過程でも、途中の各ノードでデータの更新・追加が可能である。

4 負荷バランサーの実現

4.1 均等分割動的負荷分散 (UDB) 方式の実現

全探索問題などでは、探索順序に関わりなく最終的にすべてのノードを探索する。したがって木の形状に大きな偏りがないと仮定した場合、セルを均等に配分すれば各々の仕事量も均等となり、動的負荷分散処理を行なう回数が最小ですむと考えられる。

そこでセル・リストを子ノードに均等配分していく動的負荷分散アルゴリズムを考える。このとき終端ノードに近い子ノードを分割することにより頻繁に動的負荷分散が必要になるのを防ぐため、一定の深さ (LC) 以下ではセル・リストの分割を行わない機構を設ける。また $\alpha\beta$ 探索など、既に探索を終了した部分の結果を利用して枝刈りを行なうアルゴリズム用に、最初に一定の深さ (UC) まで逐次探索を行なう機構を用意し枝刈り効率を高められるようにする。

以上述べた方式を均等分割動的負荷分散 (UDB: Uniformly Processor Divided Dynamic Load Balancing) 方式と名付け、負荷バランサー関数として実装した (図 5a)。アルゴリズムの詳細を以下に示す。

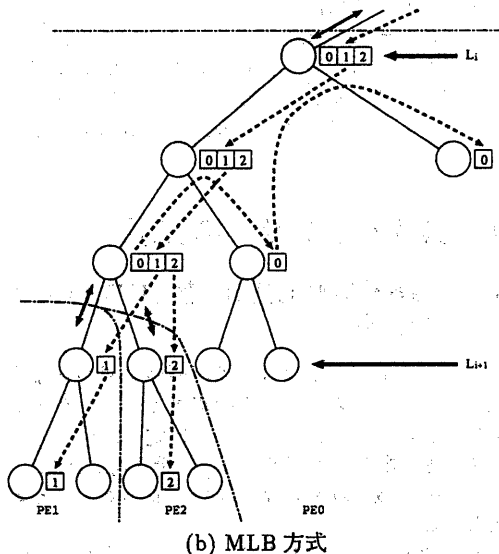
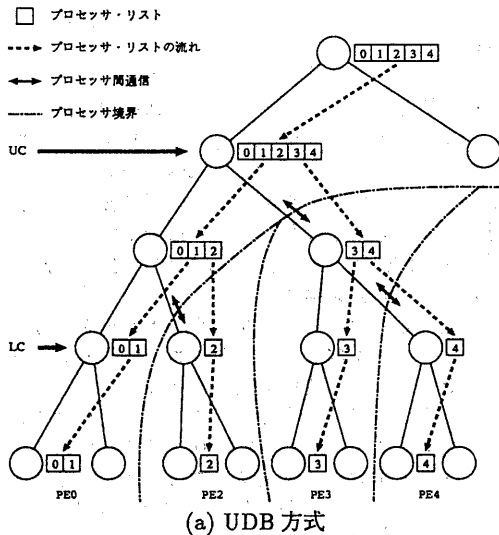


図 5: 負荷バランサーの動作

1. 深さが UC 未満では、セル・リストの分割を行なわない。
2. 深さが UC 以上、 LC 未満では、
 - (a) セル・リストが未割当の子ノードがあれば、それらに均等分割する。
 - (b) a で該当子ノードがなく、セル・リストが割当済みで同一プロセッサ上に存在する子ノードがあれば、それらに均等分割する。
 - (c) a, b で該当子ノードがなく、セル・リストが割当済みで他のプロセッサ上に存在する子ノードがあれば、それらに均等分割

する。

3. 深さが LC 以上では、プロセッサ・リストの分割および追加を行なわない。すなわち、逐次の深さ優先探索を行なう。

4.2 マルチレベル動的負荷分散 (MLB) 方式の実現

マルチレベル動的負荷分散 (MLB: Multi-Level Dynamic Load Balancing) 方式は、古市らにより提案された動的負荷分散方式である [2]。この方式ではプロセッサをグループ化し、プロセッサ・グループ間とグループ内でそれぞれ負荷分散を行なうことにより、階層的に負荷の均一化を行なう。この結果、仕事の割付を行なうプロセッサが多階層をなすため特定のプロセッサがボトルネックにならず、高い台数拡張性を持つことが報告されている。

MLB 方式は木を深さにより横方向分割するため、DTS アルゴリズムでは実現できなかったが、EDTS ライブラリでは、次に述べるアルゴリズムで負荷バランサー関数として実装できた (図 5b)。

n レベルの負荷分散を行なうとする。セルをグループ $G_0, \dots, G_n$ に分け、 G_i に含まれるセルは深さ L_i から $L_{i+1}-1$ までの探索を担当するものとする (初期化)。

セル $C_j \in G_i$ は次のようにしてセル・リストの割当を行なう。

1. $L_i \leq \text{深さ} < L_{i+1}$ のとき

- (a) C_j を含むセル・リストは分割せず、未探索の子ノードの 1 つに割り当てる。該当ノードがなければ親ノードに返却する。
- (b) C_j を含まないセル・リストは分割せず、 C_j を含むセル・リストを割当済みの子ノードの 1 つに割り当てる。該当ノードがなければ親ノードに返却する。ただし深さ $= L_i$ で該当ノードがない場合、および該当ノードが C_j によってこれから展開される場所である場合には、このセル・リストは C_j によって保持される。

2. 深さ $= L_{i+1}$ のとき

- (a) C_j を含むセル・リストは、 C_j のみを 1 つのセル・リストとして親ノードに返却する。残りは 2b に従う。
- (b) C_j を含まないセル・リストは、 G_{i+1} に含まれるセルを 1 個ずつ含むように分割し、未探索の子ノードにそれぞれ割り当てる。1b で保持されていたセル・リストも、ここで未探索の子ノードに割り当てられる。

1a, 2a により C_j は深さ L_i から $L_{i+1}-1$ までのみを探索し、2b により深さ L_{i+1} のノードを G_{i+1} に含まれるセルに割り当てる。これらは同様に深さ L_{i+1} から $L_{i+2}-1$ までを探索した後、1a により自身のセル・リストを C_j に返却し、1b によって次の仕事を割り当てられる。

古市らの実装では、セル $C_j \in G_i$ も暇になった時には L_{i+1} より下の探索を行なうが、今回の実装ではそこまで実現していない。このためパラメータが不適切な時は仕事をしない時間が多くなり急激な性能低下を起こす危険があるが、適切なパラメータを与えた場合の性能は差がないと思われる。

5 性能評価

5.1 評価用アプリケーション

3章で述べたライブラリを使用し、以下の2種類のアプリケーションを作成した。

詰め込みパズル (全解探索) 詰め込みパズルは、様々な形をしたピースを長方形のケースに詰め込むパズルであり、古市らによる MLB 方式の評価 [2] に使用されている。

ここでは、 5×8 の大きさをもつケースに、各々4つの 1×1 の正方形からなる面積4の10個のピースを詰め込む場合を考え、そのすべての方法を求める全解探索問題として、プログラムを作成した。

オセロの着手決定問題 ($\alpha\beta$ 探索) ゲーム木探索の例としてオセロを取り上げ、26手まで進んだ局面において6手先までをミニマックス法で評価し最良着手を決定するプログラムを作成した。

また、 $\alpha\beta$ 法により枝刈りを行なっているが、EDTS ライブラリではすべての子ノードを1度に生成してしまうので、探索の必要がなくなった子ノードのうちセル・リスト未割当のものを削除する方法をとっている。このため複数のセルで探索を行なった場合は、枝刈りが行なわれる前にセル・リストを割り当てられたノードについては枝刈りが行なわれない。

5.2 計測項目

各アプリケーションと負荷バランサーの組合せにおいて、パラメータを変えながら以下の項目を測定した。

実行時間 ホストよりルート・ノードがセルに送られてから、探索が終了するまでの絶対時間 (秒) を測定した。

台数効果 1台での実行時間/ n 台での実行時間で

表 1: 詰め込みパズルにおける実行結果

台数	UDB		MLB	
	実行時間	台数効果	実行時間	台数効果
1	33.61	1.00	31.98	1.00
4	8.27	4.07	10.85	2.95
16	2.23	15.08	2.85	11.24
64	1.05	31.89	0.67	47.59
256	0.39	85.74	0.30	107.32
512	0.35	96.85	0.24	134.38

表 2: オセロ着手決定における実行結果

台数	UDB		MLB	
	実行時間	台数効果	実行時間	台数効果
1	75.18	1.00	29.21	1.00
4	28.66	2.62	11.15	2.62
16	16.20	4.64	11.61	2.52
64	6.96	10.80	4.50	6.50
256	3.28	22.92	1.66	17.61
512	1.98	38.01	0.99	29.41

台数	UDB		MLB	
	割付ノード	枝刈り効率	割付ノード	枝刈り効率
1	82293	100.00	82293	100.00
4	124571	66.06	93510	88.00
16	280129	29.38	333358	24.69
64	406731	20.23	430821	19.10
256	466450	17.64	878673	9.37
512	556097	14.80	911919	9.02

あり、何倍速くなったかを示す。

割付ノード数 オセロの着手決定問題において、枝刈りにより削除されずセル・リストを割り当てられた子ノードの総数を集計した。

枝刈り効率 1台での割付ノード数/ n 台での割付ノード数であり、逐次の $\alpha\beta$ 探索と比較して枝刈りがどれだけ有効に行なわれたかを示す。

5.3 計測結果と考察

測定の結果、実行時間の最も短かったものを表1, 2 に挙げた。また、台数効果をグラフ化したものを、図6に示した。

詰め込みパズル UDB, MLB の両方式ともある程度の台数効果が現れており、特に後者については64台までは線形に近い台数効果が得られている。

MLB 方式では最適の場合 UDB 方式よりも良好な速度向上率が得られたが、パラメータによる変化が激しく、64台で最悪2倍程度の場合があった。これは4.2節で述べたように各レベルの仕事量とセル台数のバランスが悪いと暇なセルができてしまうためと考えられる。また、256台以上のセルを用いた場合に効率の大幅な低下が見られるが、生成ノード数が約100万であったことと実行時間から1セル当りの仕事量が少なすぎたと考えられる。

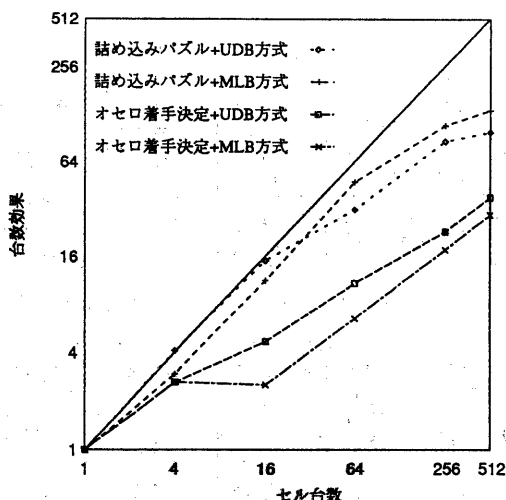


図 6: 台数効果

実際、生成ノード数が約3億となる問題ではセル256台と512台での実行時間がそれぞれ31.52秒と17.01秒であり、後者は前者の1.85倍の速度向上が得られた。このことから大きな問題では、3桁のセル台数でもほぼ線形の台数効果が期待できる。

オセロの着手決定問題 UDB、MLBの両方式ともに、詰め込みパズルに比べると台数効果はあまり得られず、64台でそれぞれ10.8、6.5倍程度であった。これは $\alpha\beta$ 探索が逐次性を持ち、並列探索を行なった場合は本来刈られるはずの枝まで探索されることによるものと考えられる。

MLB方式では詰め込みパズルと逆にUDB方式よりも結果が悪く、台数効果の伸びが不均一である。これはMLB方式が $\alpha\beta$ 探索に向かないというよりも、木の深さが6と浅かったため適切なパラメータが存在しなかったためであろう。これに関しては、より大きな問題で評価し直す必要がある。

なお、既にセル・リストを割り当てられ探索を開始しているノードに対しても探索の必要がなくなったという情報をノード間通信で送るようにすれば、これらのノードに対しても枝刈りを行なうことができる。しかしこの場合は、そのための通信時間と削減される探索時間とのトレードオフになり、よりよい結果が得られるかどうかは今後の検証課題である。

6 おわりに

以上、DTSアルゴリズムを拡張したEDTSアルゴリズム、およびそのライブラリ化について述べ

た。EDTSアルゴリズムは木構造の管理や動的負荷分散機能を実現し、様々な並列木探索問題や動的負荷分散戦略に適用できる。

本アルゴリズムを並列計算機AP1000上にライブラリとして実装し、複数の木探索アプリケーションおよび負荷バランサー関数による性能評価を行なったところ、DTSアルゴリズムでは実現できなかったMLB方式を詰め込みパズルの全解探索と組み合わせた場合で、64台で47倍という結果が得られ、汎用性・台数拡張性に優れたライブラリとして利用できることが示された。

このライブラリを実用的なものにするためには、汎用的かつ高性能な負荷バランサーの実現が欠かせない。今回実現した2種類では、UDB方式はパラメータにあまり敏感でなくある程度速度向上が簡単に得られる。一方MLB方式ではより高性能が得られるものの適切なパラメータ設定が困難である。また両者ともに逐次性が強い $\alpha\beta$ 探索に対しては、線形に近い台数効果は得られるものの、さほど大きな速度向上は得られなかった。

今後の課題としては、高性能かつパラメータ設定の容易な負荷分散方式の実現、あるいはMLB方式などにおけるパラメータ設定支援システムの構築などが挙げられる。

謝辞

並列計算機AP1000の実行環境を提供していた(株)富士通研究所、ならびに日頃より御討論いただく京都大学工学部情報工学教室 富田研究室の諸氏に感謝致します。

参考文献

- [1] Chris Ferguson and Richard E.Korf, "Distributed Tree Search and its Application to Alpha-Beta Pruning", *Proc. 7th National Conf. on Artificial Intelligence*, pp.128-132, Aug. 1988.
- [2] 古市 昌一, 瀧 和男, 市吉 伸行, "疎結合並列計算機上でのOR並列問題に適した動的負荷分散方式とその評価", 情報処理学会計算機アーキテクチャ研究会, pp.73-81, Nov. 1989.
- [3] 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", *CPSY91-8*, pp.33-40, Jul. 1991.
- [4] 石畑宏明, 堀江健志 "MIMDアーキテクチャとそのアルゴリズム - 1", 並列アルゴリズムと並列アーキテクチャ理論と実際, pp.71-84 Jun. 1992.
- [5] 富士通株式会社, "AP1000 プログラム開発手引書 (1) C言語インタフェース", Nov.1989.