TR-0796

PIMOS負荷バランスユーティリティ マニュアル

古市　昌一（三菱）

August, 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

PIMOS

負荷バランスユーティリティマニュアル

スタック分割動的負荷バランサー (STB)

マルチレベル動的負荷バランサー (MLB)

三菱電機株式会社
情報電子研究所
古市　昌一
furuichi@isl.melco.co.jp

1992年7月

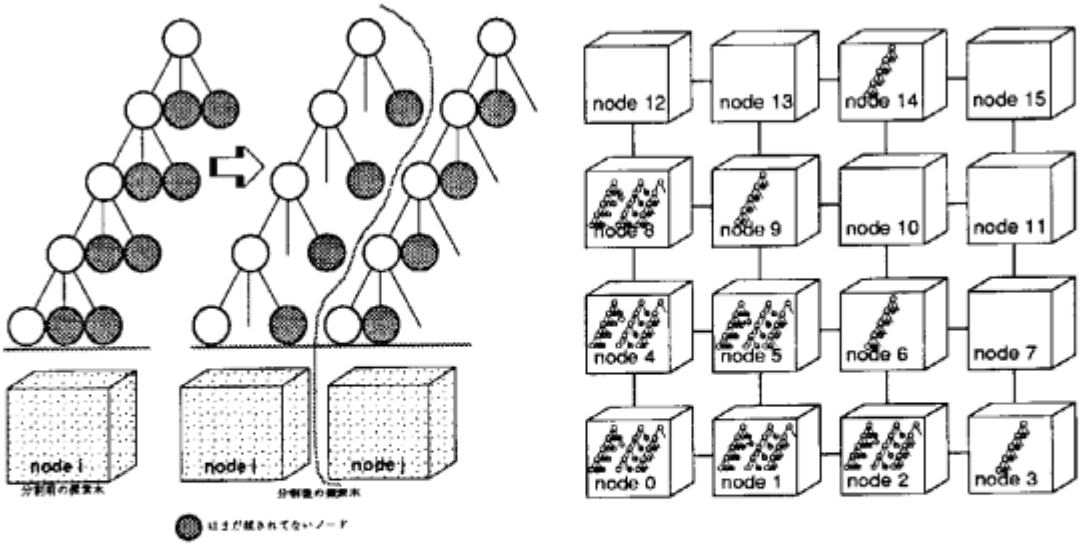
Load Balancer

はじめに

本説明書は、OR 並列型探索問題の並列処理に適した 2 種類の動的負荷バランスユーティリティの使用方法を記述したものである。ここで OR 並列型探索問題とは、一つの探索問題を複数の独立した部分探索木の探索問題として並列に実行できる問題のことで、言いかえると、「問題を互いに独立性の高い部分問題に分割して並列実行できる」タイプの問題である。例えば *n queens* や詰め込みパズルは OR 並列型探索問題であり、本ユーティリティを適用して効率良く動的負荷バランスを行うことができる。なお、部分問題同士は完全に独立である必要はなく、少々の通信或いは同期を伴うものもこの範疇に含める。しかし、負荷の均等化のためには充分な数の部分問題に分割できることが望ましい。

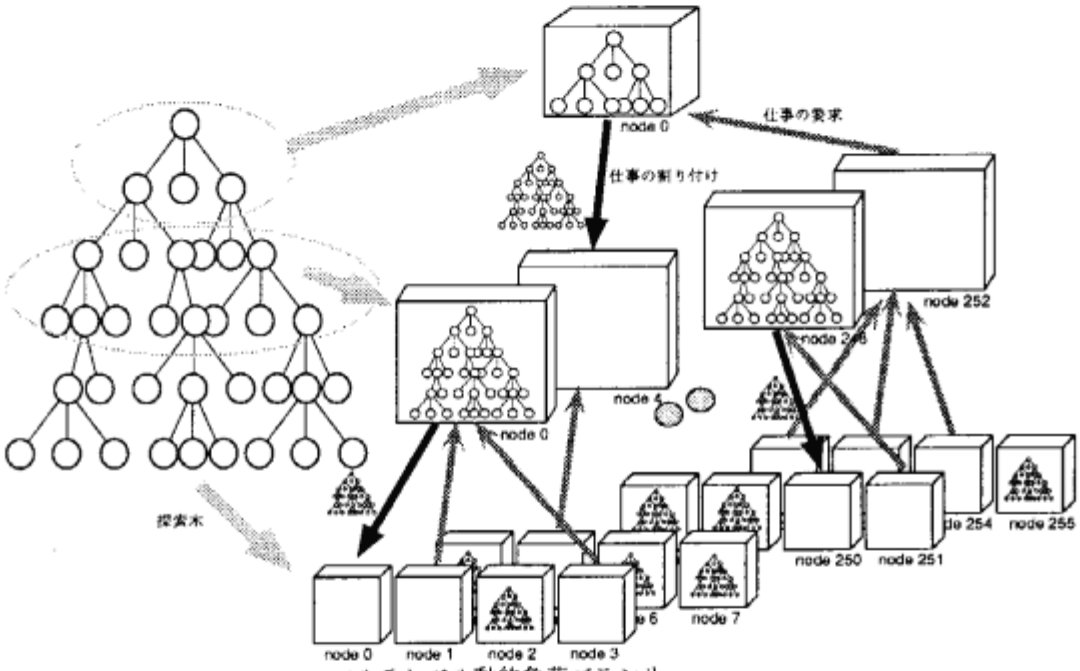
ここで説明する 2 つの負荷バランサーは、いずれも OR 並列型探索問題に適用するものであるが、使い方は次のように異なる。

スタック分割動的負荷バランサー: STB STB では、プロセッサが暇になると自動的に他のプロセッサから仕事 (データ) を取ってきて負荷のバランスを行う。ユーザ側は、STB に対して特定の規則に従って問題記述をプログラムとして与える。具体的には、探索問題の探索木をスタックで表現するものとして、ユーザはスタックからデータを一つ取り出し、そのデータを処理して部分解データをスタックに戻す手続きを STB に教えておく。更に、データを処理した結果が最終解の場合には、その解を集める手順を STB に教えておく。従って、ユーザはプログラム中にプラグマを付加することなく、プロセッサ内での処理をプログラムとして与えてやるのみで、本ユーティリティによって負荷のバランスは自動的に行われる。



スタック分割動的負荷バランサー
(スタックの分割と、プロセッサへの動的マッピング)

マルチレベル動的負荷バランサー: **MLB** MLB では、現在実行すべき仕事がなくて暇にしているプロセッサを検出する機能を提供する。ユーザ側は、プログラム中に負荷分散するゴール部分に負荷分散プラグマ **@node(NodeNo)** を付加しておき、実行時に MLB から得られた暇なプロセッサに対してゴールを割り付けする。従って、ユーザはプロセッサへの割り付けアルゴリズムを考えなくても、プラグマを付ける場所に注意し、本ユーティリティによって検出される暇なプロセッサに割り付けを行なうことにより、負荷のバランスは自動的に行なわれる。



マルチレベル動的負荷バランサー
(部分問題への分割と、プロセッサへの動的マッピング)

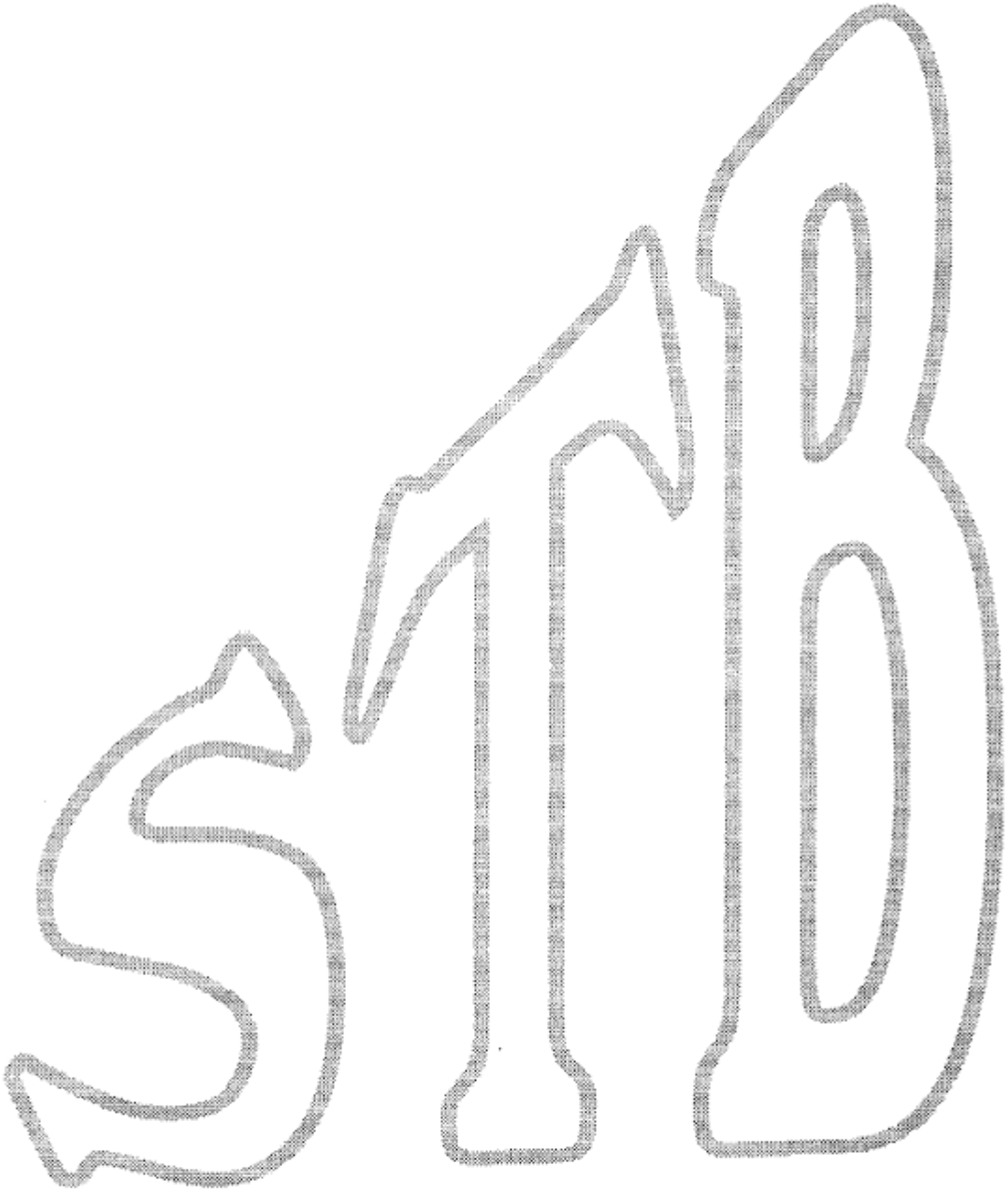
本使用説明書では、全体を 2 部に分けて 第 I 部 でスタック分割動的負荷分散方式 (STB)、第 II 部でマルチレベル動的負荷分散方式 (MLB)、の説明を行う。以下第 1 章では STB 方式の基本的な解説を行い、第 2 章で本ユーティリティの使用方法を説明す。また第 3 章では簡単な使用例を示し、第 4 章で N クイーン問題に適用したプログラム例を示す。続いて第 5 章では MLB 方式の基本的な解説を行い、第 6 章で本ユーティリティの使用方法を説明す。また第 7 章では簡単な使用例を示し、第 8 章で N クイーン問題に適用したプログラム例を示す。

なお、STB 方式に関しては [4]、 MLB 方式に関しては [2, 3] 、 に詳しく説明されているので参照のこと。

目次

第Ⅰ部

スタック分割動的負荷バランサー



第 1 章

スタック分割動的負荷分散方式

本章では、スタック分割動的負荷分散方式の基本的な考え方と、その実現方式について簡単に説明する。なお、更に詳細を知りたい場合には、文献 [4] を参照のこと。

1.1 STB の基本的な考え方

STB は、深さ優先探索アルゴリズム等スタックを用いた問題の並列処理に適した負荷分散方式である。ここで探索木はスタックで表現し、スタック中には木の各ノードをデータとして保持する。プログラムは、このスタック中からデータを一個取り出し、探索を一段深めた結果のノードデータを新たにスタックに格納する。リーフに達して解が得られた場合には、そのデータはスタックには格納せずに外部にストリームを通じて報告する。

STB は、スタックからデータを取り出して処理するプロセス（ユーザプログラム）をあらかじめ各プロセッサに一つずつマッピングしておき、スタックの分割を自動的に行ってプロセッサ間の負荷の均等化を自動的に行う。従って、ユーザは単一プロセッサ内処理のプログラムのみを記述すれば良い。

1.2 スタックの分割とプロセッサへのマッピング

図 1.2 は、STB でスタックの分割がどのように行われるかを示している。図中、左側の探索木は分割前の状態を表している。○は既に探索したノードを表しており、●はまだ探索していないノードを示している。また、カットオフとはその探索レベルより深い部分の探索は同一プロセッサ内で行う様子を表している。これは、あまり小さな粒度の仕事は別プロセッサにマッピングしない方が良いからであるが、このカットオフは STB では自動的には行わないので、カットオフが必要な場合には、ノードデータ中に探索レベルを保持させ、ユーザプログラムの中でカットオフの制御を行うこと。

図 1.2 は、分割されたスタックがどのように行われるかを示している。初期状態では、各プロセッサ上のスタックは全て空である。次に、1 つのプロセッサ上のスタックに初期データが入れられると、実行が開始される。すると、その時自分のプロセッサ上のスタック上のデータが空のプロセッサは、他のプロセッサに対してデータを要求するメッセージを送る。データが得られた場合にはそのデータをスタックに入れ、得られなかった場合には別のプロセッサに対して要求する。

どのプロセッサに対して要求するかは種々の方法が考えられるが、本 STB では乱数で表されるプロセッサに対して要求する。

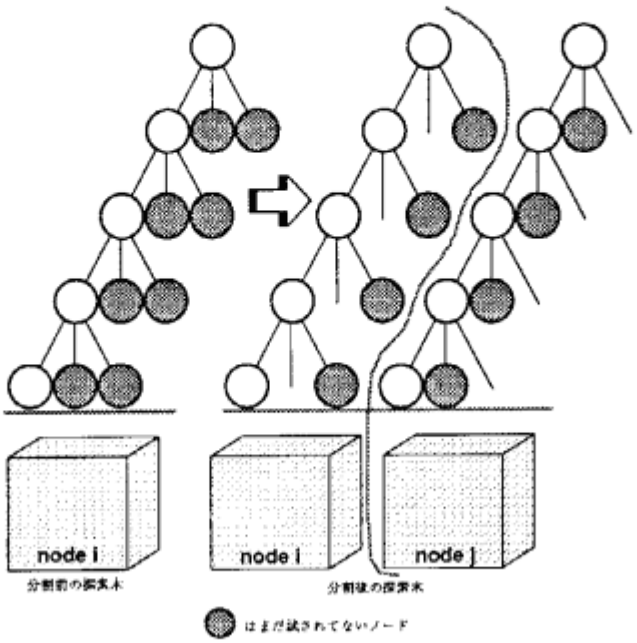


図 1.1: スタックの分割

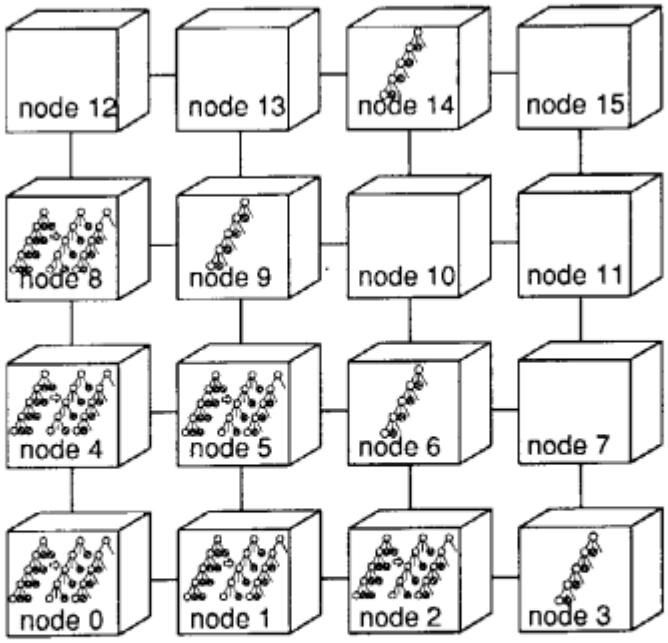


図 1.2: 分割したスタックのプロセッサへのマッピング

第 2 章

STB の使い方

2.1 ユーザプログラムとのインタフェース

STB を使うには、モジュール stb_solver 中で次の 2 つの述語をユーザが定義する必要がある。いずれの述語も、STB システム中から呼ばれる述語であり、この仕様通りに定義すること。

expand(Problem,^SubProblems,^Solutions)

本述語は、STB システムがスタック中からノードデータ Problem を一つ取り出し、これをパラメタとして呼び出す、サブルーチン的に用いられる述語である。ユーザは、Problem に対して処理を行い、次のノードデータ SP0,SP1,... をストーリーム SubProblems に返す。新たなノードデータをスタックに格納する処理は、STB が行う。また、Problem に対して処理を行った結果答えが得られた場合には、答え ANS0,ANS1,... を ストリーム Solutions に返す。

combine(In,Out)

本述語は、expand の処理によって同一プロセッサ内で得られた全ての答えをストリーム In に集め、それに対してユーザが 述語 combine で定義された処理を行い、その結果をストリーム Out に流すためのプロセスを定義した述語である。言い換えると、プロセッサ内で得られた全ての解に対してフィルターをかける処理を行う。

以下に 述語 combine の定義例を示す。

得られた解の総和を計算する

```
combine(In,Out):- true   | sum(In,0,Out).
sum([],    S,Out):- true | Out=[S].
sum([X|Xs],S,Out):- true | sum(Xs,^(S+X),Out).
```

得られた解をマージする

```
combine(In,Out):- true | merge(In,Out).
```

図 2.1 には、4 台のプロセッサ上で STB を利用した際のプロセス構造を示す。各プロセッサ上には STB プロセスが常駐し、これはスタックの管理、他のプロセッサ上の STB プロセスとのスタックのやりとりを行う。また、STB プロセスはスタックよりデータを取り出すたびに 述語 expand を呼び、その結果をスタックに戻すあるいは解を combine プロセスに受け渡す処理を行う。全プロセッサの combine プロセスの出力は、ストリーム Results にマージされる。

なお、使用する際にはまず STB プログラムをロードし、次にユーザプログラムをコンパイルし、その後に STB プログラムと再リンクすること。

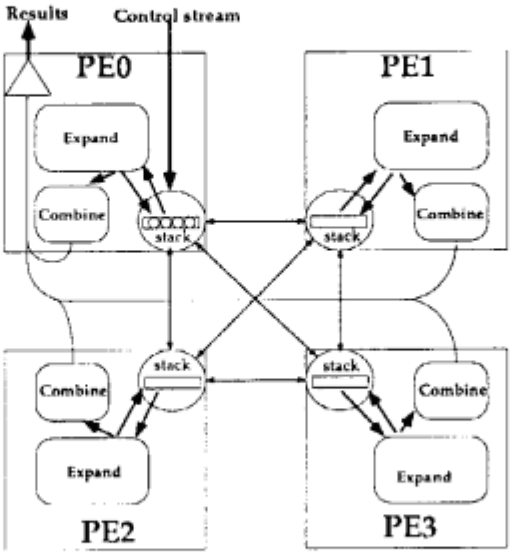


図 2.1: STB のプロセス構造

2.2 起動方法

ストリームの獲得

起動するためには、まずバランサーを起動して制御ストリームを得る。制御ストリームの獲得方法には次の2種類がある。

stb:create(^BL,^Results)

STB への制御ストリームと解が得られるストリームが得られる。この場合、全てのプロセッサ（ノード）を用いて負荷分散が行われる。

stb:create(NofNodes,^BL,^Results)

NofNodes は使用するプロセッサ（ノード）の数であり、整数で指定する。本述語が実行されたプロセッサから数えて NofNodes 台のプロセッサが用いられる。プロセッサ番号の上限を越えた場合には、サイクリックに0番のプロセッサより用いられる。

実行の開始

実行の開始は、STB のストリーム BL にスタックに格納する初期データを流すことによって行われる。

start(InitData)

InitData は、スタックに格納された後ユーザ定義述語 expand に渡されるデータであり、どんなデータタイプでも構わない。ただ、負荷バランス時にプロセッサ間で受け渡されるデータなので、通信量を減らすためにあまり複雑な構造体データは用いずに、単純な構造体データを用いるのが望ましい。

評価データの収集

STB の起動後実行の開始前には、プロセッサ間に全結合のストリームネットワークを張る。次のメッセージを BL に流すことによって、ネットワークの初期化に要した時間 (ミリ秒) を得ることができる。

netgen_time(Time)

実行の終了

STB 方式では、実行の終了は全てのプロセッサ上のスタック中のデータが空になったことで検出できる。STB では、ショートサーキット法 ([1] 参照のこと) を用いて、全てのノードデータにショートサーキットフラグを自動的につけて終了を判定している。従って、ユーザからみると解が返ってくるストリーム Results が閉じられたことによって実行の終了は判定できる。

第 3 章

STB の基本的な使用例

ここでは、STB の基本的な使用例を示す。

STB を使うためには、モジュール stb をあらかじめコンパイルあるいはロードしておく。また、ユーザはモジュール stb_solver を作成してその中に二つの述語を定義しておかねばならない。

3.1 モジュール stb_solver の例

スタックから取り出されたデータ **Problem** をモジュール **my_expand** の述語 **expand/3** の呼び出しによって処理するプログラムは次のように記述する。

```
:- module stb_solver.
:- with_macro pimos.
expand(Problem,SubProblems,Solutions) :- true |
        my_expand:expand(Problem,SubProblems,Solutions).
combine(In,Out) :- true | merge(In,Out).
```

次にこのプログラムをコンパイルした後に、再リンクしてモジュール stb と本モジュール stb_solver をリンクする。このプログラムの実行は次のように行う。この例では、スタックに格納する初期データが整数の 100 であるとする。

```
?- stb:create(BL,Results),BL=[start(100)].
Results=[1,2,3,4,5....]
```

第 4 章

STB の N クイーン問題への応用例

ここでは、STB を N クイーン問題に適用して負荷分散制御した例を示す。N クイーン問題自身の説明は、KL1 講習会の初級編テキスト [1] を参照のこと。

4.1 負荷分散を行わない場合の N クイーンプログラム

まず、負荷分散を行わないときのプログラムを示す。

```
:- module queens.
:- with_macro pimos.
:- public queens/2.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% queens(N,R)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
queens(N,R) :- N>0 |                                        % (1)
        merge(R0,R),                                        % (2)
        queens(N,N,[],R0).                                  % (3)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% queens(N,I,B,R)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
queens(N,I,B,R) :- I>0  |                                   % (4)
        next_queen(N,I,N,B,R).                              % (5)
queens(_,0,B,R) :- true | R=[B].                            % (6)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% next_queen(N,I,J,B,R)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
next_queen(N,I,J,B,R) :- J>0 |                              % (7)
        R={R0,R1},                                          % (8)
        try(N,I,J,B,R0),                                    % (9)
        next_queen(N,I,~(J-1),B,R1).                        % (10)
next_queen(_,_,0,_,R) :- true | R=[].                       % (11)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% try(N,I,J,B,R)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
try(N,I,J,B,R) :- true |                                    % (12)
        check(B,J,1,Res),                                   % (13)
        if_succeeded(N,I,J,B,R,Res).                        % (14)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% if_succeeded(N,I,J,B,R,YesNo)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if_succeeded(N,I,J,B,R,yes) :- true |                       % (15)
        queens(N,~(I-1),[J|B],R).                           % (16)
if_succeeded(_,_,_,_,R,no) :- true | R=[].                  % (17)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% check(B,J,D,YesNo)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
check([K|_],K,_,Res) :- true | Res=no.                    % (18)
check([K|_],J,D,Res) :- J=:=K+D | Res=no.                 % (19)
check([K|_],J,D,Res) :- J=:=K-D | Res=no.                 % (20)
otherwise.                                                % (21)
check([_|B],J,D,Res) :- true |                            % (22)
        check(B,J,~(D+1),Res).                            % (23)
check([],_,_,Res) :- true | Res=yes.                      % (24)
```

4.2 STB で負荷分散を行った N クイーン

STB を利用する際には、先の負荷分散を行わない例の中で探索プロセスを生成する部分を、プロセスを生成するかわりにデータを生成するように仕様を変更すればよい。

```
:- module stb_solver.
:- with_macro pimos.
:- public expand/3, combine/2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% expand(Problem,SubProblems,Results)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
expand({N,I,B},SubProblems,Results):- I>0 |               % (1)
        Results=[],                                       % (2)
        next_queen(N,I,N,B,SubProblems).                  % (3)
expand({N,I,B},SubProblems,Results):- I=0 |               % (4)
        SubProblems=[],Results= [B].                      % (5)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% combine(In,Out)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
combine(In,Out):- true | merge(In,Out).                   % (6)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% next_queen(N,I,J,B,S)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
next_queen(N,I,J,B,S) :- J>0 |                            % (7)
        S={S0,S1},                                        % (8)
        try(N,I,J,B,S0),                                  % (9)
        next_queen(N,I,~(J-1),B,S1).                      % (10)
next_queen(_,_,J,_,S) :- J=0  | S=[].                     % (11)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% try(N,I,J,B,S)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
try(N,I,J,B,S) :- true |                                  % (12)
        check(B,J,1,Res),                                 % (13)
        if_succeeded(N,I,J,B,S,Res).                      % (14)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% if_succeeded(N,I,J,B,S,YesNo)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if_succeeded(N,I,J,B,S,yes) :- true |                     % (15)
        S=[{N,~(I-1),[J|B]}].                             % (16)
if_succeeded(_,_,_,_,S,no) :- true | S=[].                % (17)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% check(B,J,D,YesNo)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
check([K|_],K,_,Res) :- true | Res=no.                    % (18)
check([K|_],J,D,Res) :- J=:=K+D | Res=no.                 % (19)
check([K|_],J,D,Res) :- J=:=K-D | Res=no.                 % (20)
otherwise.                                                % (21)
check([_|B],J,D,Res) :- true |                            % (22)
        check(B,J,~(D+1),Res).                            % (23)
check([],_,_,Res) :- true | Res=yes.                      % (24)
```

ここで、負荷分散を行わないプログラムから変更した部分は次のとおりである。まず、(1) 行から (6) 行までを STB とのインタフェースを合わせるために **expand**、 **combine** に変更し、再帰呼び出しを行っていた部分を単純な述語呼び出しに変更した。また、(16) 行の部分では、再帰呼びだしを行っていた部分を、 **SubProblems** にデータを返すように変更した。

なお、このプログラムの実行方法は、次の通りである。

```
?- stb:create(BL,Result),BL=[start({8,8,[]})].
BL=[[6,4,2,....],[....],...]
```

第 II 部

マルチレベル動的負荷バランサー

第 5 章

マルチレベル動的負荷分散方式

本章では、マルチレベル動的負荷分散方式の基本的な考え方と、その実現方式について簡単に説明する。なお、更に詳細を知りたい場合には、文献 [2, 3] を参照のこと。

5.1 MLB の基本的な考え方

MLB では、その時に実行すべき仕事がなくて暇にしているプロセッサを検出するメカニズムを提供する。ユーザはこれを利用してプログラム中でゴールに負荷分散プラグマ **@node(NodeNo)** を付加し、プロセッサ間の負荷の均等化は動的に MLB が行う。MLB が扱う問題のモデルは次の通りである。

(1) 問題を部分問題に分割し、部分問題をプロセッサに割り付ける。
この部分は同一のプロセッサ（マスタープロセッサ）上で実行される。以後サブタスクジェネレータと呼び、生成された部分問題はサブタスクと呼ぶ。

(2) 割り付けられた部分問題（サブタスク）を実行する。
一旦プロセッサに割り付けられたサブタスクは同一プロセッサ上で実行される。

ここに示したように、1 台のプロセッサがサブタスクの生成を行ない、一旦プロセッサにサブタスクを割り付けた後には同一プロセッサ上で行なうような負荷分散を、ここでは単純な負荷分散方式と呼ぶ。

図 5.1には、探索問題を単純の負荷分散方式を用いて実行する様子を示している。探索問題の場合は、探索を一段深める毎に部分探索木ができるが、この部分探索木以下の探索をサブタスクとここでは呼ぶ。図中、大きな三角形は OR 木で表される探索空間を示す。縦棒で塗り潰された小さな三角形はサブタスクの生成を行なうサブタスクジェネレータを示し、探索の深さがあるレベル (負荷分散 (開始) レベル) に達するまでマスタープロセッサ上で実行される。円は生成されたサブタスクを示し、その大きさが粒度を示す。従って、サブタスクの粒度は負荷分散レベルを深くするに従って小さくなる。ただし、粒度にはばらつきがあるものと仮定する。各サブタスクは、各プロセッサの負荷がバランスするようにプロセッサに割り付ける。

ここで、サブタスクの粒度には互いに反する 2 つの条件が要求される。一方では多くの PE を全て忙しく働かせるためにサブタスクの数は多くなければならない。他方、負荷分散の手間と比較して充分粒度を大きくしなければ、負荷分散のためのオーバヘッドにより性能が抑えられる。例えば, 図 5.1(a) の場合は負荷分散のオーバヘッドは小さいが、サブタスクの数が少なく粒度のばらつきにより負荷はバランスされない。また、図 5.1(c) の場合はサブタスクの数が多過ぎて、サブタスクの供給がボトルネックとなる。従って良い台数効果を得るためには、図 5.1(b) に示されるように、チューニングによって最適な負荷分散レベルを見つけられるようにプログラムを設計するのが望ましい。

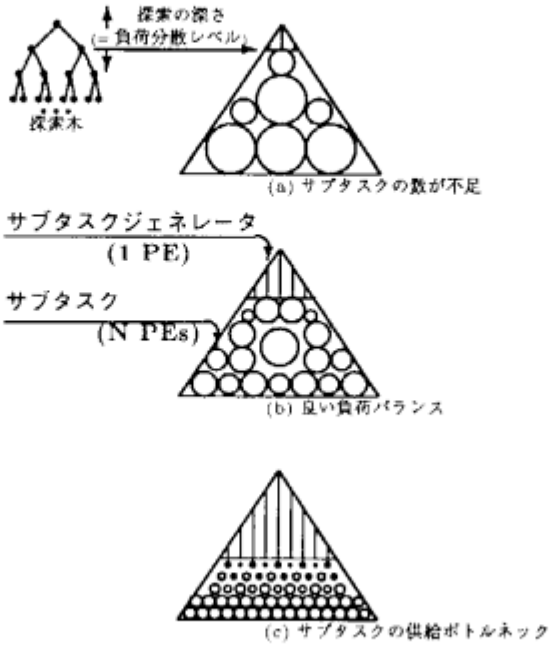


図 5.1: サブタスクの生成

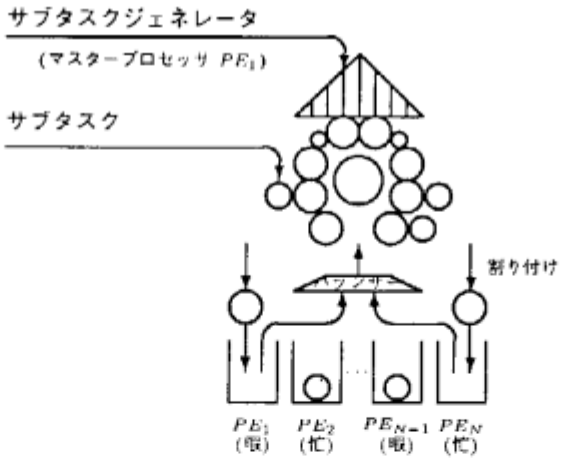


図 5.2: 単純な動的負荷バランス

5.2 単純な動的負荷バランス

次に、各プロセッサの負荷がバランスするようにサブタスクをプロセッサに割り付ける基本的な方式について説明する。プロセッサへの割り付けは、現在暇なプロセッサに対して行なわれる。暇なプロセッサの検出は、あるプロセッサが暇になるとマスタープロセッサに対してメッセージを送ることによって行なわれる。この時、暇なプロセッサからのメッセージを受け取ってユーザプログラムに対して暇なプロセッサを知らせるマネージャープロセスを、負荷バランスマネージャあるいは負荷バランサーと呼び、これはマスタープロセッサ上に置かれる。ユーザプログラムでは、サブタスクを生成するとまず負荷バランサーに対してストリーム通信で等い合わせを行なうことによって現在暇なプロセッサ番号を

得、このプロセッサ対してサブタスクを割り付ける。図 5.2に単純な動的負荷バランス方式の構成図を示す。

5.3 マルチレベル動的負荷バランス

先に示した単純な動的負荷バランス方式では、プロセッサの台数が増えた時にはサブタスクの供給がボトルネックになるという問題点がある。すなわち、プロセッサの台数が増えるに従って単位時間当たりに解かれるサブタスクの数が、単位時間当たりに生成及び供給するサブタスクの数を越えたときに、サブタスクの供給がボトルネックとなる。このような場合には、サブタスクの生成を行う部分を階層的に行い、供給を行うプロセッサの台数を増やせば良い 。図 5.3には、２段階でサブタスクの生成を行う例を示してある。

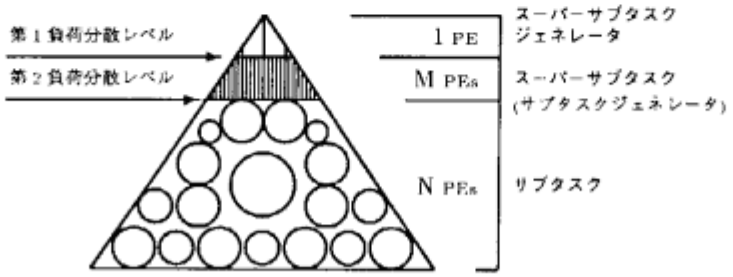


図 5.3: 二段階のサブタスク生成

サブタスクの生成を二段階で行なうのにともない、本負荷バランスユーティリティでは N 台の物理プロセッサ (PE) を論理的に M 個のプロセッサグループ (PG) にグループ化し、階層的に負荷バランスを行なう。ここで、プロセッサグループの内特定のプロセッサはグループマスタープロセッサと呼ぶ。従って、M 台のグループマスタープロセッサがあるものとする。

まずマスタープロセッサ上で行なわれる一段目のサブタスク生成では、生成したスーパーサブタスクを次々と暇なグループマスタープロセッサに割り付ける。この時、暇なグループマスタープロセッサを管理する負荷バランサーはマスタープロセッサ上に置かれる。

グループマスタープロセッサ上で行なわれる二段目のサブタスク生成では、生成したサブタスクを次々とグループ内の暇なプロセッサに割り付ける。この時、グループ内の暇なプロセッサを管理する負荷バランサーはグループマスタープロセッサ上に置かれる。この様子は図 5.4 に示されている。なお、一段目のサブタスク生成及び割り付けが終了した後、グループ内のプロセッサで暇になったプロセッサは、まだサブタスクの生成及び割り付けを行っている他のプロセッサグループに動的にマージされる（グループマージ）。

ここで、プロセッサを論理的に M 個のグループにグループ化する際、本負荷バランスユーティリティでは M はあらかじめユーザがグループ分割ファクター **SpF** として与えることができる。例えば、６４プロセッサのシステムに対して **SpF** を４とした場合にはプロセッサグループの個数は４個となり、二段目の負荷バランスは １６プロセッサ $(\frac{4}{64})$ で構成されるプロセッサグループ内でローカルに行われる。

MLB では、更に多段階の負荷バランスを行なうことができる。この場合２段階の場合と同様のことを再帰的に行うものであり、段数を深めるにつれてプロセッサグループの大きさはグループ分割ファクターによって次第に小さくなる。例えば、６４プロセッサのシステムに対して **SpF** を４とした場合には、二段目の負荷バランスで行なわれるプロセッサグループの大きさは１６プロセッサ、三段目のプロセッサグループの大きさは４プロセッサとなり、以後段数を増やした場合にはプロセッサグループの大きさは１プロセッサとなる。なお、ここで示したプロセッサグループの大きさは初期状態のものであり、サブタスクの生成及び割り付けの終了したグループができた場合には、以後グループマージメカニズムによって次第にグループは大きくなる。

5.3. マルチレベル動的負荷バランス

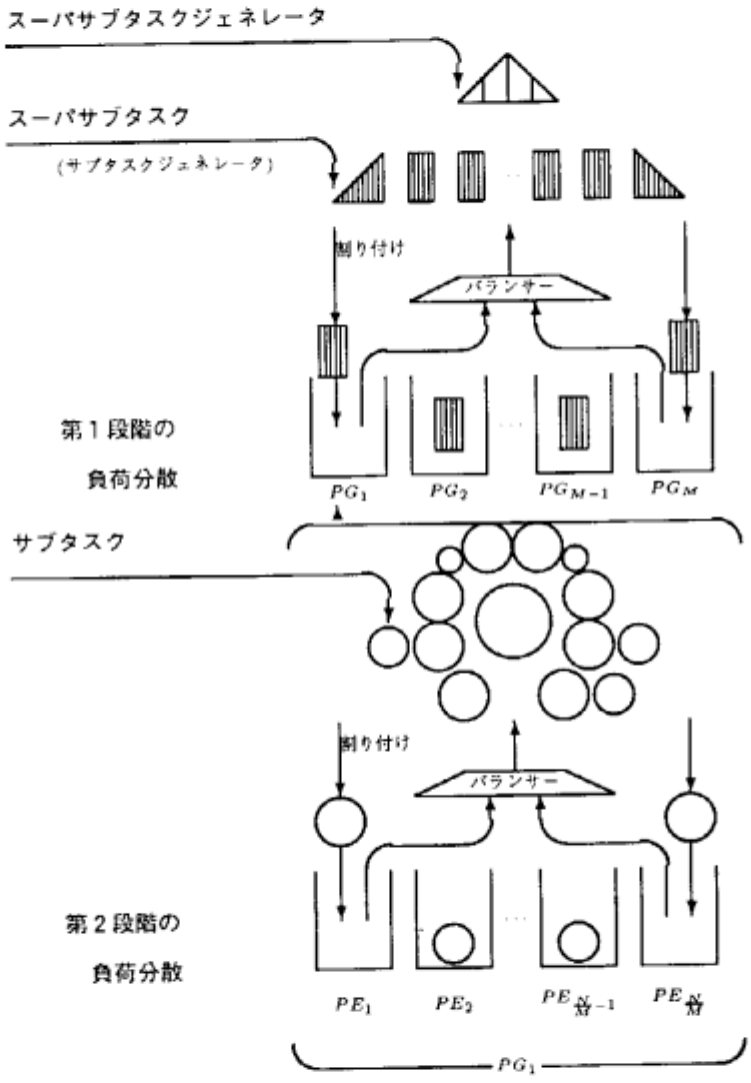


図 5.4: 2 レベルの動的負荷バランス

第 6 章

MLB の使用方法

まずモジュールの述語呼び出しを行なうと、呼び出したプロセッサ上に負荷バランサープロセスが生成され、負荷バランサーストリーム ^BL （以後 BL あるいはバランサーストリームと呼ぶ）を得る。その後、1 レベル負荷バランスを行う場合と 2 レベル或いはマルチレベル負荷バランスを行う場合とでは、使用法が次のように異なる。

1 レベル負荷バランス
: BL に対して get(^Pe) メッセージを流すことによって現在暇なプロセッサ番号を得、Pe に対してゴールを割り付ける。

2 レベル負荷バランス
: BL に対して get(^Pe,^NewBL) メッセージを流すと現在暇なプロセッサ Pe をグループマスタプロセッサとして、その上に新しい負荷バランサープロセスを生成する。ゴールはグループマスタプロセッサに割り付け、以後グループ内の動的負荷分散は NewBL に対して get(^Pe) メッセージを流して 1 レベル負荷バランスの場合と同様に行なう。

マルチレベル負荷バランス
: 更にマルチレベル負荷バランスを行なう場合は、2 レベル負荷バランスの時に NewBL に対して get(^Pe) メッセージを流す代わりに get(^Pe,^NewBL) メッセージを流すことによって行なう。このようにネストすることによって何段の負荷バランスでも行なうことができるが、プロセッサ台数に応じてプロセッサグループを構成するプロセッサが 1 台程度になるレベルでネストを打ち切るのが効率上望ましい。

6.1 ストリームの獲得方法

ストリームの獲得方法には次の 2 種類がある。それぞれの違いは、負荷分散するプロセッサの台数あるいは番号、マルチレベル負荷分散する際のグループ分割ファクターの指定方法が異なる点である。

multi_level_load_balancer:create(Parameters,^BL)

使用するプロセッサ或いはグループ分割ファクター等を Parameters で指定するとバランサーストリーム BL が得られる。Parameters はリスト形式で、各要素はプロセッサ指定を行う PEs パラメタとグループ分割ファクターを指定する SpF パラメタからなる。各パラメタを与える順番はいずれでも良く、また省略した場合にはデフォルト値が設定される。

グループ分割ファクター SpF

グループ分割ファクターパラメタは、次の様な項形式で与える。ここで、SpF は整数で与える。省略した場合は、デフォルトとして 4 が与えられる。

splitting_factor(SpF) あるいは spf(SpF)

プロセッサ指定パラメタ PEs

プロセッサ指定パラメタは、次の様な項形式で与える。省略した場合は、デフォルトとして全プロセッサが与えられる。

processor(PEs) あるいは pes(PEs)

PEs は、リスト形式でプロセッサ番号を与える。プロセッサ番号の指定は、次のような表記が可能である。また、これらは混在可能である。

表記	説明	例
整数	整数で与えたプロセッサ番号	0,1,3
整数 A-整数 B	整数 A で与えたプロセッサ番号から 整数 B で与えたプロセッサ番号まで全て	0-15, 24-63
整数 A (*)	整数 A で与えたプロセッサ番号から 最後のプロセッサまで全て	0-(*), 32-(*)
整数 A:整数 B	整数 A で与えたプロセッサ番号から 整数 B で与えた台数のプロセッサ	0:16, 24:32

Parameters の指定例

- **[processor([0,1,2,3]),splitting_factor(2)]**
  プロセッサ 0,1,2,3 を指定、グループ分割ファクターは 2。
- **[spf(4),pes([0-3,4:4,8,9,10,12- *])]**
  16 プロセッサのシステムであれば、プロセッサ 1 1 以外全てのプロセッサを指定、グループ分割ファクターは 4。

multi_level_load_balancer:create(^BL)

Parameters の指定を省略した呼びだし方法で、全プロセッサが指定され、グループ分割ファクターはデフォルト値 4 が与えられる。

6.2 暇なプロセッサ番号の獲得方法

BL ストリームに対して次のようなメッセージを流すことによって行なう。

get(^PeNo)
一段階の負荷バランスで暇なプロセッサ番号を得る場合、或いは多段階負荷バランスでプロセッサグループ内の暇なプロセッサ番号を得る場合、負荷バランサーのストリーム **BL** に対してこのメッセージを流すと、暇なプロセッサが検出された時点で **PeNo** にプロセッサ番号が得られる。負荷分散プラグマ **@node(PeNo)** を用いて、暇なプロセッサにゴールを割り付ける。

get(^PeNo,^NewBL)
多段階の負荷バランスで暇なプロセッサ上に新たな負荷バランサーを作る場合、負荷バランサーのストリーム **BL** に対してこのメッセージを流すと、暇なプロセッサが検出された時点で **PeNo** にプロセッサ番号が得られる。その時、このプロセッサ **PeNo** 上には新たな負荷バランサーが作られて **NewBL** が得られる。負荷分散プラグマ **@node(PeNo)** を用いて、暇なプロセッサにゴールを割り付け、その先の負荷バランサーは **NewBL** を用いる。

第 7 章

基本的な使用例

7.1 1 レベル負荷バランスの例

整数 N で与えられた個数ゴール fork を投げるプログラムの例を示す。

```
go(N):- true |                                  % (1)
        multi_level_load_balancer:create(BL),   % (2)
        distribute(N,BL).                       % (3)
distribute(0,BL):- true | BL=[].                % (4)
otherwise.                                      % (5)
distribute(C,BL):- true |                       % (6)
        BL={BL1,BL2},                           % (7)
        BL1=[get(PeNo)],                        % (8)
        fork_ext@node(PeNo).                    % (9)
        distribute(^(C-1),BL2).                 % (10)
fork_ext:- true | fork@priority($,-100).        % (11)
```

(2) で負荷バランサーを生成する。負荷バランサーの使用を終了する時には、(4) のように **BL** に対して [] を流す。(7) では **BL** を 2 つのストリームに分割し、その一方に対して (8) で暇なプロセッサを得るメッセージを流している。(9) では、(8) で得られた暇なプロセッサに対してゴール **fork_ext** を割り付けている。なお、(11) では負荷分散したゴールの優先度を下げているが、これは部分問題を生成して負荷分散する作業は、部分問題の実行よりも優先的に実行した方が効率的であるからである。詳しくは、KL1 講習会テキストの初級編に説明されている。

7.2 2 レベル負荷バランスの例

先の例では 1 台のプロセッサが N 個のゴール fork を投げていたが、一段階目の負荷分散では 10 個のゴール distribute を投げ、その先二段階目の負荷分散で、ゴール spawn はそれぞれ $\frac{N}{10}$ あるいは N *mod* 10 個のゴール fork を投げるプログラムの例を示す。トータルで N 個のゴール fork が投げられる。

```
go(N):- true |                                  % (1)
        multi_level_load_balancer:create(BL),   % (2)
        distribute(N,^(N/10),BL).               % (3)
distribute(C,No,BL):- C=<No |                   % (4)
        BL=[get(PeNo,NewBL)],                   % (5)
        spawn(C,NewBL)@node(PeNo).              % (6)
distribute(C,No,BL):- C>No  |                   % (7)
        BL={BL1,BL2},                           % (8)
        BL1=[get(PeNo,NewBL)],                  % (9)
        spawn(No,NewBL)@node(PeNo),             % (10)
        distribute(^(C-No),No,BL2).             % (11)
```

7.2. 2レベル負荷バランスの例

```
  spawn(0,BL):- true | BL=[].                          % (12)
otherwise.                                             % (13)
  spawn(C,BL):- true |                                 % (14)
          BL={BL1,BL2},                                % (15)
          BL1=[get(PeNo)],                             % (16)
          fork_ext@node(PeNo).                         % (17)
          spawn(~(C-1),BL2).                           % (18)
  fork_ext :- true | fork@priority($,-100).            % (19)
```

(4) から (11) の部分は一段目の負荷分散を行ない、(12) から (18) で二段目の負荷分散を行なっている。(10) では、二段目の負荷分散を行なうゴール spawn に新たな負荷バランサー **NewBL** を渡している点に注意すること。

第 8 章

N クイーン問題への応用例

ここでは、本負荷バランスユーティリティをNクイーン問題に適用して負荷分散制御した例を示す。問題の説明は KL1 講習会の初級編テキスト [1] を参照のこと。

8.1 負荷分散を行わないNクイーン

負荷分散を行わない場合のNクイーンプログラムは、第 4.1 章を参照のこと。

8.2 1レベル負荷分散を行ったNクイーン

1レベル負荷分散を行ったNクイーンのプログラム例を示す。なお、負荷分散を行わない時と同じ述語 check/4 に関しては省略してある。負荷分散の方針は、D に指定した探索レベルまでは同一プロセッサ上で実行し、それ以下の探索は別プロセッサで実行するものである。

```
:- module queens.
:- public queens/3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% queens(N,R,D)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
queens(N,R,D) :- N>0,D>=0 |                          % (1)
        multi_level_load_balancer:create(BL),         % (2)
        merge(R0,R),                                  % (3)
        queens(N,N,[],R0,D,BL).                       % (4)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% queens(N,I,B,R,D,BL)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
queens(N,I,B,R,D,BL) :- I>0  |                        % (5)
        next_queen(N,I,N,B,R,D,BL).                   % (6)
queens(_,0,B,R,_,BL) :- true | R=[B],BL=[].           % (7)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% next_queen(N,I,J,B,R,D,BL)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
next_queen(N,I,J,B,R,D,BL) :- J>0,D=0 |               % (8)
        BL={BL0,BL1},                                 % (9)
        R={R0,R1},                                    % (10)
        BL0=[get(Pe)|BL2],                            % (11)
        try_ext(N,I,J,B,R0,D,BL2)@node(Pe),           % (12)
        next_queen(N,I,~(J-1),B,R1,D,BL1).            % (13)
next_queen(N,I,J,B,R,D,BL) :- J>0,D>0 |               % (14)
        BL={BL0,BL1},                                 % (15)
        R={R0,R1},                                    % (16)
        try(N,I,J,B,R0,D,BL0),                        % (17)
        next_queen(N,I,~(J-1),B,R1,D,BL1).            % (18)
```

```
next_queen(N,I,J,B,R,D,BL) :- J>0,D<0 |                    % (19)
        R={R0,R1},                                         % (20)
        try(N,I,J,B,R0,D,_),                               % (21)
        next_queen(N,I,~(J-1),B,R1,D,BL).                  % (22)
next_queen(_,_,0,_,R,D,BL) :- true | R=[],BL=[].           % (23)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% try(N,I,J,B,R,D,BL)
% try_ext(N,I,J,B,R,D,BL)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
try(N,I,J,B,R,D,BL) :- true |                              % (24)
        check(B,J,1,Res),                                  % (25)
        if_succeeded(N,I,J,B,R,D,BL,Res).                  % (26)
try_ext(N,I,J,B,R,D,BL) :- true |                          % (27)
        merge(NewR,R),                                     % (28)
        try(N,I,J,B,NewR,D,BL)@priority($,-100).           % (29)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% if_succeeded(N,I,J,B,R,D,BL,YesNo)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if_succeeded(N,I,J,B,R,D,BL,yes) :- true |                 % (30)
        queens(N,~(I-1),[J|B],R,~(D-1),BL).                % (31)
if_succeeded(_,_,_,_,R,_,BL,no) :- true | R=[],BL=[].      % (32)
```

ここで、(12)で負荷分散を行なう時には try_ext を呼び、(17)で負荷分散をしない時には try を呼び出している。これは、(29)でわかるように負荷分散したゴールの優先度は下げる方が効率的であるからという理由と、(28)のように新たに解を集めるマージプロセスを生成したほうが効率的であるからである。Nクイーン問題のように解の個数が多い場合には、特定の一台のプロセッサ上のみにマージプロセスを作った場合よりも、ゴールを投げたプロセッサ上にはそれぞれマージプロセスを作った方が効率が良いことがわかっている。

8.3 多段階負荷分散を行ったNクイーン

2レベル負荷分散を行ったNクイーンのプログラム例を示す。なお、1レベル負荷分散を行ったNクイーンと同じ述語 try/7 、 try_ext/7 及び if_succeeded/8 に関しては省略してある。負荷分散の方針は、Dで指定した探索レベルまでに達するまで次々と多段階に負荷分散を行い、指定したレベルに達した後の探索は同一プロセッサ上で行うものである。

```
:- module queens.
:- public queens/3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% queens(N,R,D)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
queens(N,R,D) :- N>0,D>=0 |                                % (1)
        multi_level_load_balancer:create(BL),              % (2)
        merge(R0,R),                                       % (3)
        queens(N,N,[],R0,D,BL).                            % (4)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% queens(N,I,B,R,D,BL)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
queens(N,I,B,R,D,BL) :- I>0  |                             % (5)
        next_queen(N,I,N,B,R,D,BL).                        % (6)
queens(_,0,B,R,_,BL) :- true | R=[B],BL=[].                % (7)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% next_queen(N,I,J,B,R,D,BL)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
next_queen(N,I,J,B,R,D,BL) :- J>0,D>0 |                    % (8)
        R={R0,R1},                                         % (9)
```

```
        BL=[get(Pe,NewBL)|BL1],                                  % (10)
        try_ext(N,I,J,B,R0,D,NewBL)@node(Pe),                    % (11)
        next_queen(N,I,~(J-1),B,R1,D,BL1).                       % (12)
next_queen(N,I,J,B,R,D,BL) :- J>0,D=0 |                          % (13)
        BL={BL0,BL1},                                            % (14)
        R={R0,R1},                                               % (15)
        BL0=[get(Pe)|BL2],                                       % (16)
        try_ext(N,I,J,B,R0,D,BL2)@node(Pe),                      % (17)
        next_queen(N,I,~(J-1),B,R1,D,BL1).                       % (18)
next_queen(N,I,J,B,R,D,BL) :- J>0,D<0 |                          % (19)
        R={R0,R1},                                               % (20)
        try(N,I,J,B,R0,D,_),                                     % (21)
        next_queen(N,I,~(J-1),B,R1,D,BL).                        % (22)
next_queen(_,_,0,_,R,D,BL) :- true | R=[],BL=[].                 % (23)
```

参考文献

[1] 新世代コンピュータ技術開発機構. "KL1 プログラミング入門編／初級編／中級編" July 1989.

[2] M. Furuichi, K. Taki, and N. Ichiyoshi "A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI" In *Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 50-59, Mar. 1990

[3] 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". *Proceedings of KL1 Programming Workshop '90*, pages 1–9, Mar. 1990.

[4] 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". *Proceedings of KL1 Programming Workshop '91*, pages 51–58, Mar. 1990.