共有メモリ並列マシン上の細粒度自動負荷分散方式の評価

畑 澤 宏 善†

並列計算機上で大規模プログラムを効率良く実行するためには，負荷分散の方式が問題となる．本論文では，共有メモリ並列マシン上の細粒度自動負荷分散方式について並列推論マシン PIM/p 上の KL1 処理系を使って評価した．本処理系では，クラスタ内の8台の要素プロセッサを対象に自動負荷分散機構を実装している．ベンチマークプログラムを使って，負荷分散機構の性能を分析した結果，次の問題点が見つかった．それは，データの局所性を考慮しない動的負荷分散では共有バスの負荷が高まり，性能低下の原因となること．また，自動負荷分散を行うことでプロセッサ数を増すと見込み計算が増加することなどである．これらの問題を解決するために，自動負荷分散方式についての次の二つの改良方式を検討し，評価した．すなわち，(1)サスペンド状態から実行可能状態に復帰したプロセスを復帰させたプロセッサ上で実行する改良と，(2)自プロセッサの低優先度プロセスよりも他プロセッサの高優先度プロセスを優先して実行する改良である．この結果，自動負荷分散機構によって，静的プロセス割り付けを行った場合に匹敵する台数効果が得られた．

Evaluation of Fine-Grain Automatic Load Balancing Method on the Shared Memory Parallel Machine

Hiroyoshi Hatazawa†

We evaluate the load balancing method for shared memory parallel machines using the KL1 system on the parallel inference machine, PIM/p. The automatic load balancing method was implemented for eight processing elements of PIM/p. We find that the low data locality causes the large bus traffic, and that the inefficient load balancing increases the speculative calculation. In order to solve these problems, we present following two improvements. (1) When a suspended process is resumed by a processor, it will be executed on the processor, not on the one which made the suspension. (2) When there are no current priority processes in a processor's queue, the processor takes other processor's current priority process before taking its own low priority processes. By applying these improvements for the automatic load balancing method, we can get as high performance as the case where we locate the processes statically on the processors.

1. はじめに

ICOT では，並列論理型言語 KL1 を実行する並列推論マシン PIM (Parallel Inference Machine) の開発を行っている．PIM にはアーキテクチャの異なる複数のモデルがあり，PIM/p[1),2)] は，8台の要素プロセッサ (PE) が並列キャッシュメモリを介してメモリ共有結合されたクラスタを，ネットワークで結合する二階層構成をとっている．PIM/p の KL1 処理系[3)] では，KL1 プログラムに内在する高い並列性を引き出すと同時にプログラマの負担を軽減するために，クラスタ内の自動負荷分散機構を実装した．

マルチプロセッサシステムにおける動的負荷分散方式については，以下のような研究がなされてきた．プログラム中に負荷分散単位の処理を陽に指示して動的負荷分散を行う方法では，最短経路問題や OR 並列型探索問題などについていくつかの方式が示されている[4),5)]．一連の研究として，文献 6) では，世代の概念を取り入れた拡散型動的負荷分散方式によって，隣接する疎結合プロセッサ間の負荷を均等化する方式を提案している．これらの方式では，負荷分散の粒度を適当な大きさに保つことができるが，対象とするプログラムについての知識が不可欠で，また効率良い負荷分散を行うためにパラメタの調整を行う必要がある．

これに対し，共有メモリによって密結合されたマルチプロセッサシステムでは，プロセッサ間の通信コストが比較的小さいため，より細粒度の動的負荷分散を効率良く実現できると考えられる．文献 7) では，GHC 処理系におけるエキストラ・キューを使った動

† (財)新世代コンピュータ技術開発機構
Institute for New Generation Computer Technology

的負荷分散方式の有効性が示唆された．共有メモリ型マルチプロセッサ上に Prolog や Parlog を実装する方式としては，Andorra-I[8] や Parallel Parlog[9),10] が提案されている．Andorra-I では AND/OR 並列を，Parallel Parlog では AND 並列を扱い，自動負荷分散を行っている．

クラスタ内のプロセッサが共有メモリによって密結合されている PIM/p では，動的負荷分散の機能をクラスタ内自動負荷分散機構として KL1 処理系に組み込んだ．これによって，プログラム中に負荷分散の指示を書かずに，KL1 の AND 並列性に基づく並列実行を可能にした．しかし，負荷分散の性能は大きくばらついており，プログラムの性質に依存すると考えられる．

そこで本研究では，並列推論マシン PIM/p 上の KL1 処理系を用いて，細粒度動的負荷分散の問題点やそれに対する改良方式について検討する．始めに，ベンチマークプログラムを用いて負荷分散状況を分析する．特にハードウェアの視点からは，負荷分散と共有バスを介したプロセッサ間通信との関係についても調べ，データの局所性を考慮しない動的負荷分散ではプロセッサ間通信が増大し，システムの性能が落ちることを示す．また，プロセッサ間通信を減らすために，負荷分散を陽に指定した場合についても調べる．これらを元に，自動負荷分散方式についての改良を提案し，評価する．その結果，単純なプログラムに関しては，自動負荷分散によって静的にプロセスを割り付けた場合と匹敵する性能が得られることを示す．

2. 自動負荷分散機構の実装

PIM/p の KL1 処理系では，細粒度のプロセスであるゴール単位で負荷分散を行う．ゴールはゴールレコードと呼ばれる構造体で実現される．KL1 のゴールには，ソースプログラム中でプラグマを付加することによって，実行優先度や実行プロセッサを指定することができる．ただし，実行プロセッサを指定されたゴールは自動負荷分散の対象外となるので，以下では特に触れない．

各 PE は優先度別のゴールキュー（プライオリティキュー）をもっており，このうち現在実行中のゴールと同じ優先度のゴールキューは，参照を容易にするためカレントプライオリティキューとして保持している．これらのキューは，エンキュー／デキュー操作によって後入れ先出し（LIFO）方式で管理するが，カレントプライオリティキューにはゴール数に制限を設ける．これらのキューは PE ごとのローカルな領域に保持され，アクセス時に排他制御を要しない．

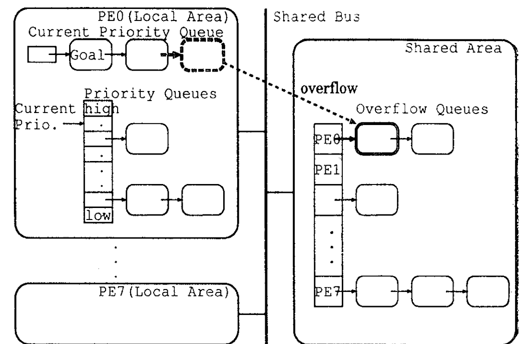

図 1 PIM/p のゴールキュー
Fig. 1 Goal queues in PIM/p.

さらに各 PE ごとにオーバーフローキューがあり，カレントプライオリティキューに一定数以上のゴールが溜ると，最も古いゴールからオーバーフローキューに移される（**図 1**）．オーバーフローキューは共有エリアに保持され，アクセス時に排他制御を必要とする．アクセス順序は LIFO 方式で，他 PE からもゴールをデキューすることができる．この方式は，文献 7）の方式を改良したもので，PE ごとにオーバーフローキューを持つことによって，エンキュー／デキュー時の排他制御によるキューへのアクセス競合を起こりにくくしている．このような利点については，文献 9）でも確認されている．なお，オーバーフローキューにはゴール数の制限はない．

本処理系においてゴールを PE 間で送受するのは，次の二つの場合である．

［get-G］ 実行可能なゴールを持たない PE は，アイドル状態となりクラスタ内他 PE のオーバーフローキューから，ゴールを取り出す（**図 2**）．これによって自動負荷分散を行う．この時に，どの PE のゴールキューから優先してゴールを取り出すかの戦略は性能に影響があると思われるが，今回の実装では，隣の PE（自 PE+1 番）から順にオーバーフローキューにゴールがあるかどうかを調べる方式をとった．

［throw-G］ 実行中断（サスペンド）状態にあるゴールは，中断の要因となった未定義変数の具体化によって実行可能状態になる（リジュームされる）．ゴールをリジュームさせた PE がサスペンドさせた PE と異なる場合，そのゴールはサスペンド元の PE に投げ返される（**図 3**）．この方式は，サスペンド元 PE のキャッ

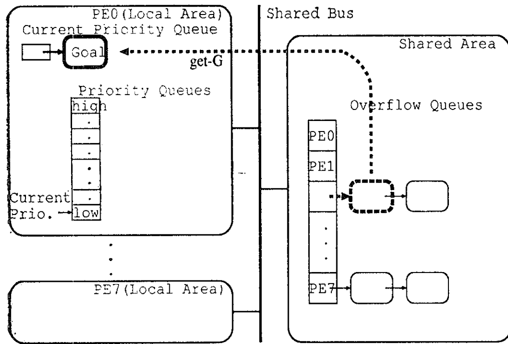

図 2 get-G 解説図
Fig. 2 Schematic explanation for get-G.

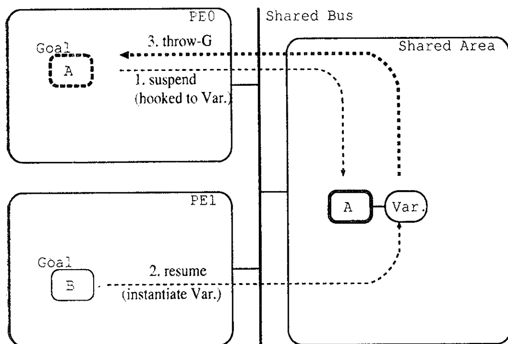

図 3 throw-G 解説図
Fig. 3 Schematic explanation for throw-G.

シュにはゴールを実行するための命令コードやデータが乗っていると期待して採用された．

以下では，これら二つのケースを get-G，throw-G と略記する．

また，デキュー時の各キューの優先順位は，

1. プライオリティキュー内の高優先度ゴール
2. カレントプライオリティキュー
3. 自 PE のオーバーフローキュー
4. プライオリティキュー内の低優先度ゴール
5. 他 PE のオーバーフローキュー (get-G)

としている．

この結果，実行するゴールの優先度が変化した場合には，新たな優先度のプライオリティキューをカレントプライオリティキューとする．新たな高優先度ゴールが発生した際に，オーバーフローキューは空でなくてもそのままにされる．この場合も LIFO 管理を行うので，後からエンキューされた高優先度ゴールが先に取り出されて実行される．

3. 自動負荷分散方式の問題点

負荷分散方式の評価のために次のようなベンチマークプログラムを用いる：

bestpath メッセージ交換によって 100×100 の 2 次元メッシュ上で最短経路問題を解く[4]．

life ライフゲームのシミュレーション．オリジナルと異なり隣接する 4 方向のノードの状態によって状態遷移を行う．

zebra 全解探索型の制約充足問題[11]．文献 8) に Prolog 版では AND 並列性は低いが OR 並列性が比較的高いことが示されている．

mastermind 全解探索型の数当てパズル[11]．文献 9)，10) では Parlog 版の評価で 8 PE で 7.5〜8.0 倍の台数効果が得られている．

これらを使った評価結果を**表 1** に示した．ここで，**台数効果**は 8 PE で実行した時の 1 PE の場合に対する速度比，**バス使用率**はプログラム実行中に共有バスが使用されていた時間の割合，**サスペンド率**，**get-G**，**throw-G 発生率**は 1 リダクションあたりの発生頻度である．なお，台数効果の算出にあたっては，並列実行を意図して記述されたプログラムを 1 PE で実行して基準とした．完全に逐次向けに記述されたプログラ

表 1 各 KL1 プログラムの負荷分散状況
Table 1 Evaluation of the load balancing method for each KL1 benchmarks.

プログラム	PE 数	リダクション数 (×1000)	台数効果	バス使用率 (%)	サスペンド率 (%)	throw-G 発生率(%)	get-G 発生率(%)
bestpath	1	3,056		4.0	6.0		
	8	4,913	1.33	51.7	10.5	5.8	1.5
life	1	360		7.9	57.1		
	8	360	2.81	70.4	62.2	52.4	2.3
zebra	1	404		1.9	0.0		
	8	404	4.30	49.3	0.3	0.2	5.5
mastermind	1	1,737		0.9	0.0		
	8	1,737	7.53	14.0	0.0	0.0	0.2

ムより 1PE 性能は多少劣るが，以下の議論には影響しない．

KL1 ゴールの実行状況から各プログラムの挙動を分析したところ，以下のような問題点が見つかった．

［見込み計算の増加］ KL1 プログラムでゴールに優先度を指定して枝刈りを行う場合，PE ごとにプライオリティキューを持っているため，クラスタ内全体としては優先度が厳密に守られない．1PE では枝刈りの対象となる計算を，8PE では実行してしまう．この場合，PE 数の増加に応じてリダクション数も増加する．表1では，bestpath がその例である．

［サスペンドの増加］ KL1 プログラムを複数 PE で実行した場合，自動負荷分散によってゴールが並列に実行され，1PE ではサスペンドしなかったゴールがサスペンドすることがある．これは，データを生成するゴールよりも消費するゴールが先に走ってしまうためである．表1に示したように，一般に 8PE で実行した場合の方が 1PE の場合よりサスペンド率が高い．

［アイドル状態切替オーバヘッド］ PE がアイドル状態に移行する際には，それまで実行していたゴールの環境の退避や終了判定処理を伴う．シミュレータによる命令トレースの結果，この処理は約 150 クロックかかり，表1に示したプログラムの1リダクション（約 150～500 クロック）と同程度であった．get-G 発生率がその切替頻度の指標になり，zebra では，これが台数効果を落とす要因になっている．

［ゴール送受オーバヘッド］ throw-G 操作は，ゴール送信元から送信先 PE への通信や送信先 PE でのゴール受信処理のために，約 200 クロックを要する．throw-G 発生率の高い life では，これが台数効果を落とす要因になっている．

PE 数が増すとこれらの原因で実行命令数が増大して，台数効果が低下する．

さらに，表1にあげたプログラムのうち台数効果の低い life について，ハードウェアの性能診断機能を使って挙動を分析した．その結果，8PE で実行した場合の性能低下の要因は，**表2**のとおりであった．この表に示したように，1命令を実行するのに要したクロック数（以下 CPI と略す）の増加が，台数効果低下の最大の原因になっている．

図4に，このプログラムを 1PE で実行した場合と，8PE で実行した場合の CPI 増加分（CPI が命令実行分の 1.0 を越える部分）の内訳を示す．このグラフより，CPI 増加の大半は，並列キャッシュの無効化ペナルティ（複数の PE で共有されている可能性のあるキャッシュブロックへの書き込みに伴い，他 PE の該当ブロックを無効化するのに要した時間）とキャッシュミスによるものであることが分かる．

この現象は，以下のように説明される．life プログラムでは，**図5**に示したように，ストリームを使ってメッシュ状に接続されたノード間でデータをやりとりする．あるノード X の $(i+1)$ 世代目の状態 X_{i+1} は，X_i および隣接するノードの状態 N_i, E_i, S_i, W_i を参照して決定され，その次の世代の計算のために隣接する各ノードに送られる．本自動負荷分散方式では，

表 2 life プログラムの性能低下の要因
Table 2 Causes of performance declining in the life program.

アイドル時間	3.0%
実行命令数の増加	21.6%
CPI（1命令実行当たりのクロック数）の増加	75.4%

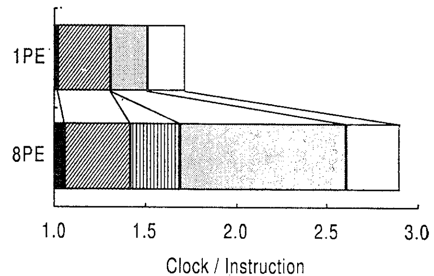

図 4 life プログラムの CPI 増加の内訳
Fig. 4 Details of the CPI increasing in the life program.

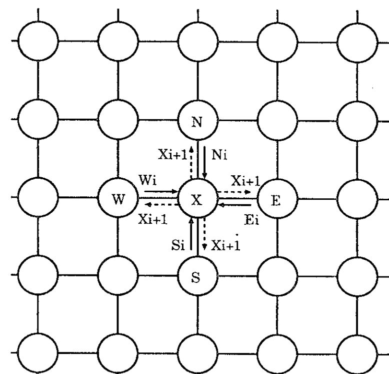

図 5 life プログラム解説図
Fig. 5 Processes in the life program.

最初のゴールが0番 PE で実行されるほかはすべて get-G による自動負荷分散でゴールが分配されるため，各ノードに対応したゴールの PE への割り付けは，ほぼランダムになる．このため，ストリームによるゴール間の通信が PE 間の通信になりやすい．PIM/p では，更新無効化方式の並列キャッシュメモリを採用しているため，PE を跨いだゴール間通信はキャッシュメモリの無効化を多発させ，結果的にキャッシュヒット率を悪化させる．

5章では，ここで述べた PE 数の増加に伴うオーバヘッドに注目して，自動負荷分散方式の改良案を検討する．また，改良による PE 間通信の増減に関しても評価を行う．

4. 静的解析を行った場合の負荷分散

この章では，動的負荷分散の問題を離れ，プログラムの静的解析ができる場合に，プログラム中に負荷分散や実行優先度の記述を明示的に行うことによって，どの程度の台数効果が得られるかを調べる．

本章では，3章で取り上げたベンチマークのうち，life を使って評価を行う．life に注目する理由は，プロセス構成が単純なため評価しやすく，メッシュ状に接続されたノード間でストリーム通信を行うため，ストリーム通信を多用する KL1 プログラムとして典型的な結果が得られると期待されるためである．

ストリーム通信の問題として，メッセージ待ちのためのサスペンド／リジュームの多発，ゴール間の通信が PE 間に跨る場合の共有バスの負荷増大とデータ局所性の低下などが考えられる．これらはいずれも3章で負荷分散上の問題となった事項である．

4.1 静的プロセス割り付け

ストリーム通信を多用する KL1 プログラムを並列実行して台数効果を上げるためには，データ局所性を増してプロセス間の通信がなるべく PE 間に出ないようにするべきである．そこで以下のようなプロセス割り付けを考える．

[静的割付] コンパイル時(あるいはプログラム時)の解析によって，頻繁に通信を行うプロセス同士を生成時に同じ PE に割り付ける．

life において，プログラム中で静的にプロセッサを指定した場合の評価を行った．life プログラムで生成するメッシュ状に接続されたノードを 2×4 の8ブロックに分割し，ブロックごとに PE へ割り付けることによって PE 間の通信を少なくした．結果を**表3**の

表 3 静的解析による改良の評価
Table 3 Evaluation of the improvements using the static analysis.

プログラム 改良項目	実行時間 (msec)	台数 効果	バス使用率 (%)	サスペンド率 (%)
life				
改良前	4,117	2.81	70.4	62.2
静的割付	2,335	4.83	48.2	63.7
サスペンド減	4,550	2.56	65.2	39.2
静的割付 ＋サスペンド減	2,441	4.69	54.5	22.0

「静的割付」に示す．ここで，**台数効果**は同様の改良を施して 1PE および 8PE を使用した時の実行速度比である．この結果，共有バス使用率が減少して，台数効果が向上した．また，実行速度も大きく向上した．

このような割り付けができれば，理想的な台数効果を得られると期待できるが，一般に知識処理等の不定型な問題ではこのような解析を，コンパイル時（あるいはプログラム時）に行うことは難しいと思われる．

4.2 サスペンド回数の削減

ストリーム通信のメッセージ待ちなど，KL1 における同期機構の実現手段であるサスペンド／リジューム方式の改善手段には，大きく二つに分けて，サスペンド／リジューム時のコスト削減と，サスペンド回数そのものの削減とが考えられる．

コスト削減は，処理系のチューニングによって行われるもので，5.1 節で触れる．

本節では，サスペンド回数の削減について検討する．サスペンド回数を減らすためには，一般にはプログラムの静的解析が必要であり，その結果を利用してコンパイル時にゴール実行順を操作するとか，文献 12) や 13) に述べられているようなプログラム変換（ソースコードレベルでサスペンドが減少するように書き換える）を行う．

特に KL1 では，ゴール実行優先度のプラグマ指定によって，ゴールの実行順序を制御することが可能である．そこで，実行優先度の指定を利用して以下の実験を行った．

[サスペンド減] life プログラムにおいて，1世代ごとに実行優先度を下げる指定を行い，ある世代の計算が全ノードで終るまで次の世代の実行を始めないようにして負荷分散状況を調べる．

結果を，表3の「サスペンド減」に示す．サスペンド率が約2/3に下がったにも関わらず，台数効果，実行時間とも改良前より悪くなっている．これは，優先度

を指定したゴール操作が通常のサスペンド／リジューム処理を含めたゴール操作に比べて重いことを示唆している．また，サスペンド率がそれほど下がらなかったのは，PE ごとにプライオリティキューを持っているためにクラスタ内全体としての実行順序制御ができなかったためと考えられる．

また，前述の静的割付と優先度指定を同時に行った場合の結果を，「静的割付＋サスペンド減」に示す．この場合にはサスペンド率は改良前の 1/3 程度になるが，台数効果および 8 PE での実行時間は「静的割付」の場合に比べ若干劣る結果となった．これも，優先度を指定したゴール操作が重いことを示しており，この点は今後改良すべき課題である．

このような優先度の指定によるサスペンド回数の減少は，直接的には負荷分散に結び付かないが，動的負荷分散によってサスペンド率が増えることを考えると，間接的に台数効果の向上に寄与すると思われる．

5. 動的負荷分散方式の改良

この章では，2 章に示した自動負荷分散機構の throw-G と get-G という二つの PE 間ゴール送受メカニズムに注目して動的負荷分散の改良方式を提案し，評価する．

4 章では，プログラムの静的解析が可能な場合についてそれを利用して台数効果を高めることができることを示したが，一般にはそのような静的解析は難しい．そこで，本章に述べるような改良によって自動負荷分散でどの程度の台数効果を得られるかを 4 章の結果と比較する．

本章の評価には 3 章で表 1 に取り上げたすべてのベンチマークプログラムを用いた．

5.1 リジューム処理の改良

本処理系では 3 章に述べたように，新しく生成されたプロセスの PE への割り付けが get-G によってほぼランダムになってしまう．そこで，ストリーム通信に伴うサスペンド／リジューム機構に注目して，データの局所性を高める工夫を検討した．現状ではリジュームしたゴールを実行するのはサスペンド元の PE である（図 3）が，これをリジュームした PE にすることで，中断要因の変数を具体化したデータをサスペンド元に送らなくても済むようになる．そこで次の改良を行った．

[throw-G 廃止] サスペンドしているゴールが中断の要因となった未定義変数の具体化によってリジュームされた時に，このゴールをリジュームした PE 上で実行する．

表 4 動的負荷分散改良の評価
Table 4 Evaluation of the improvements in the dynamic load balancing method.

プログラム 改良項目	リダクション数 (×1000)	実行時間 (msec)	台数効果	バス使用率 (%)
bestpath				
改良前	4,913	29,776	1.33	51.7
throw-G 廃止	4,761	23,461	1.69	60.0
get-G 改良	3,282	13,175	3.03	57.8
throw-G 廃止 ＋get-G 改良	3,140	9,829	4.06	56.6
life				
改良前	360	4,117	2.81	70.4
throw-G 廃止	360	2,752	4.20	63.6
get-G 改良	360	3,892	2.97	72.5
throw-G 廃止 ＋get-G 改良	360	2,300	5.03	63.4
zebra				
改良前	404	2,043	4.30	49.3
throw-G 廃止	404	2,164	4.06	47.3
get-G 改良	405	1,925	4.57	57.3
throw-G 廃止 ＋get-G 改良	405	2,074	4.24	60.0
mastermind				
改良前	1,737	1,828	7.53	14.0
throw-G 廃止	1,737	1,813	7.59	13.5
get-G 改良	1,737	1,770	7.80	10.3
throw-G 廃止 ＋get-G 改良	1,737	1,770	7.80	10.4

評価結果を，**表 4** および**図 6** の「throw-G 廃止」に示した．図 6 では，改良前の 1 PE 実行速度を 1.0 としたが改良後も 1 PE 実行速度は変わらないため，複数 PE での実行速度比はそのまま台数効果と見ることができる．

life の結果に見られるような大幅な性能向上の理由は次のように考えられる．従来の処理系では，オブジェクトコードの局所性を優先していたが，「throw-G 廃止」では，オブジェクトコードの局所性よりも，ゴール間で通信するデータと，ゴールレコードの局所性が優先される．life のように，すべてのノードにおいて同じコードを実行している場合には，「throw-G 廃止」によってもコードの局所性は損なわれないため，台数効果を高めることができる．

その他のベンチマークの結果，bestpath でも，「throw-G 廃止」を施すことによって性能が向上した．この例では，データの局所性の向上に加えて見込み計算の減少によって，実行速度が向上したと考えられる．

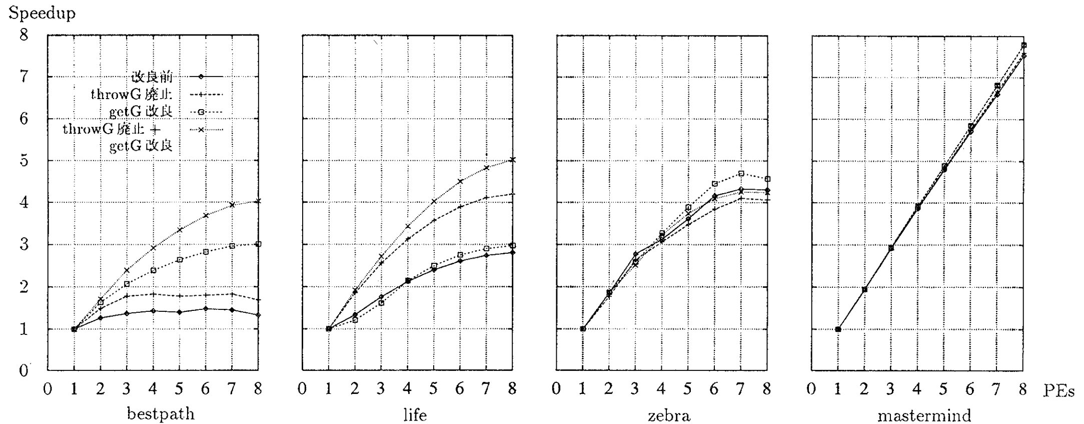

図 6 改良結果の実行速度比（改良前の 1PE 速度を 1.0 とした）
Fig. 6 Speedups by the improvements in the dynamic load balancing method.

この改良方式が throw-G オーバヘッドをなくすことから容易に想像されるように，プロセス間で頻繁にデータの交換を行い，複数 PE での実行時にサスペンドを起こす頻度が大きいプログラムほど，本改良の効果が大きいことが確認できた．

この方式の利点をまとめると：

- プロセス生成時の PE への割り付けがランダムでも，動的な処理によってデータの所局性を高められる．
- 3章で述べた throw-G オーバヘッドを 0 にする．これによって，複数 PE で実行した場合の実行命令数の増大を抑えることができる．
- 4.2 節で言及したリジューム時のコスト削減につながる．

しかし，この方式には次のような問題もある．

ストリーム通信型以外のプログラムで，例えば，ある一つのデータで多数のプロセスが一斉同期をとるような場合には，リジュームを行った 1PE にゴールが溜り過ぎる．この場合，その他の PE ではアイドル状態になりやすく，get-G に伴うアイドル状態切替えオーバヘッドが性能を低下させる．zebra の場合は，表1に示したように，もともと throw-G 発生率が小さく，get-G 発生率の方が大きかった．そのため「throw-G 廃止」の効果よりも get-G 多発の影響が大きく，実行速度が落ちたものと考えられる．しかし次節で述べる改良によって，この逆効果を小さくすることができる．

また，「throw-G 廃止」を行った結果，life では 1 世代を計算するごとに，各ノードの処理を行う PE が変わっていくという問題も生じた．このため，ストリーム通信のデータが PE 間に跨ることが少なくなっても，ゴール自体が PE 間を移動するために共有バスの負荷を十分に下げることができない．これが，表3の「静的割付」とのバス使用率の違いに現れている．これについては，負荷分散とプロセスの寿命などを合わせて考慮しなければならないと考える．

5.2 負荷分散タイミングの改良

従来のゴールキュー管理方式では，自 PE のプライオリティキュー内に実行可能ゴールが存在しないアイドル状態になって初めて，他 PE のオーバーフローゴールキューからゴールを取り出していた．この方式では，他 PE のゴールを実行する際には必ず一旦アイドル状態になるため，3章に述べたアイドル状態切替オーバヘッドが性能低下の要因となっていた．

そこで適切なタイミングで負荷分散を行うことで，アイドル状態になる頻度を小さくするため，次の改良を行った．

[get-G 改良] ゴールキューからゴールを取り出す順序を次のように変更する．

1. プライオリティキュー内の高優先度ゴール
2. カレントプライオリティキュー
3. 自 PE のオーバーフローキュー
4. 他 PE のオーバーフローキュー
5. プライオリティキュー内の低優先度ゴール

従来の方式（2章）とは4と5が逆になっている．4の場合にデキューする PE の選択方式は2章に述べたとおりである．ただし，他 PE から取ってきたゴールの優先度がそれまで自 PE で実行していたゴールの優先度よりも低い場合には，それを一旦自 PE のプライオリティキューに格納し，改めてプライオリティキ

ューからその時点で一番優先度の高いゴールを取り出す．

この改良の評価結果を，表4および図6の「get-G 改良」に示した．

bestpath で台数効果が大幅に向上しているのは，「get-G 改良」によって，見込み計算の増大を抑えることができたためである．すなわち，自 PE の保持する優先度の低いゴールよりも他 PE の持つ優先度の高いゴールを先に実行することによって，クラスタ内全体として優先度がより厳密に守られる．その結果，優先度指定による枝刈りの実行を効果的に行えるようになったのである．

他のプログラムについても実行速度で数%程度の向上が見られた．

この方式の利点をまとめると：

- アイドル状態にならないうちに，他 PE からゴールを取ってくることができるので，get-G がアイドル状態切替えオーバーヘッドを伴わなくなった．
- クラスタ内での優先度が従来よりも厳密に守られるために，優先度を使った枝刈りを行う場合に見込み計算が減少する．

また，表4および図6の「throw-G 廃止＋get-G 改良」には，この章で述べた二つの改良を同時に施した場合の結果を示した．この場合，3章で取り上げた問題点のうち，「throw-G 廃止」によってゴール送受オーバヘッドを小さくし，「get-G 改良」によって見込み計算の増加のアイドル状態切替えオーバヘッドを抑えることができた．また，サスペンドの増加に対しても，「throw-G 廃止」によってリジューム時のコストを小さくした．その結果，life, bestpath では，改良前に比べて台数効果，実行速度とも大幅な性能向上を実現できた．一方 zebra では，「throw-G 廃止」による性能低下と「get-G 改良」による向上がほぼ相殺し，改良前とそれほど変わらない結果を得た．mastermind では，もともと throw-G がほとんど発生していなかったために，「get-G 改良」による分だけ性能が向上した．

特に life においては，これら二つの改良によって，静的プロセス割り付けを行った場合と匹敵する性能を得ることができた．

6. おわりに

並列推論マシン PIM/p 上で KL1 処理系の自動負荷分散機構を評価した．その結果，従来の負荷分散方式には，3章に示したような問題のあることが分かった．さらにハード側の分析から，データの局所性を考慮しない細粒度の動的負荷分散は，共有バスの負荷を高め，システムの性能を落とすことを示した．

life プログラムのように静的解析が容易な問題に対しては，負荷分散を陽に指定することで，プロセッサ間の通信を減らし，台数効果を高めることができる．しかし，PIM の目的とするような大規模知識処理プログラムにおいては静的解析を行うことは容易ではなく，動的あるいは自動負荷分散の効率を高めることが求められる．

本論文では，サスペンドしているプロセスのリジューム時の扱いと負荷分散を行うタイミングについての改良方式を提案した．これらによって，life プログラムでは自動負荷分散でも静的プロセス割り付けに匹敵する性能が得られ，他のプログラムでも改良前に比べて性能の向上が見られた．これらの改良は，共有メモリを介したマルチプロセッサシステムの負荷分散にとって有効なものである．

また，文献14) では，ハードウェアに固有のマシン命令を利用することによって，並列キャッシュメモリの無効化ペナルティを削減できることを示した．このようなハードウェアによるサポートも効率的な並列処理を行うために重要である．

本研究においては，PIM/p の1クラスタ (8 PE) に注目して，共有メモリ並列システム上での負荷分散について検討した．しかし，PIM/p の上位階層では 64 クラスタがハイパーキューブネットワークによって接続されている．今後は，より大きなプログラムを対象に，負荷分散の改善手法を探るとともに，ネットワークで接続された並列システム上での負荷分散方式についても検討していきたい．

謝辞 本研究の機会を頂いた ICOT 第1研究部近山隆部長に感謝します．また，本研究の対象となった自動負荷分散方式の基本的な実装を行い，有益な助言を頂いたシャープ(株)技術本部情報技術研究所の今井明氏，評価のための測定を行い，有意義な討論を頂いた富士通(株)パーソナルシステム事業部 新井進氏ならびに ICOT，富士通，富士通 SSL の PIM グループ諸氏に感謝します．

参考文献

1) 服部 彰ほか：並列型推論マシン PIM/p のアーキテクチャ，情報処理学会論文誌，Vol. 30, No. 12, pp. 1584-1592 (1989).
2) Kumon, K. *et al.*: Architecture and Implementation of PIM/p, *Proc. of International Conference on the Fifth Generation Computer Systems*, pp. 414-424 (1992).
3) ICOT 第1研究室編：VPIM 処理方式解説書，TM-1044, ICOT (1991).
4) Wada, K. and Ichiyoshi, N.: A Study of Mapping of Locally Massage Exchanging Algorithms on a Loosely-Coupled Multiprocessor, TR-587, ICOT (1989).
5) Furuichi, M., Taki, K. and Ichiyoshi, N.: A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI, *Proc. 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 50-59 (1990).
6) 佐藤令子ほか：疎結合型マルチプロセッサ上の拡散型動的負荷分散方式—LLS-G 方式—，情報処理学会論文誌，Vol. 35, No. 4, pp. 571-580 (1994).
7) 安里 彰ほか：GHC 処理系における負荷分散方式の検討，信学技報，Vol. 88, No. 156, CPSY 88-46, pp. 49-54 (1988).
8) Costa, V. S., Warren, D. H. D. and Yang, R.: The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model, *Logic Programming: Proc. of the Eighth International Confernce*, pp. 825-839, MIT Press (1991).
9) Crammond, J.: Scheduling and Variable Assignment in the Parallel Parlog Implementation, *Proc. of the North American Conference on Logic Programming*, pp. 642-657, MIT Press (1990).
10) Crammond, J.: The Abstract Machine and Implementation of Parallel Parlog, *New Generation Computing*, Vol. 10, No. 4, pp. 385-422 (1992).
11) Tick, E.: *Parallel Logic Progamming*, MIT Press, Cambridge, MA (1991).
12) 瀧 和男，市吉伸行：マルチ PSI における並列処理とその評価—小粒度高並列オブジェクトモデルに基づくパラダイムについて—，電子情報通信学会論文誌，Vol. J 75-D-I, No. 8, pp. 723-739 (1992).
13) 久門耕一，平田圭二：FGHC プログラムにおけるプロセスとメッセージの交換—実行スレッドに着目した効率改善手法，日本ソフトウェア科学会第9回大会，A 2-3 (1992).
14) 畑澤宏善，新井 進：並列推論マシン PIM/p におけるクラスタ内自動負荷分散方式の評価，並列処理シンポジウム JSPP '93 論文集，pp. 355-362 (1993).

(平成5年9月14日受付)
(平成6年6月20日採録)

畑澤 宏善（正会員）

1965年生．1987年京都大学理学部卒業．1989年同大学院理学研究科地球物理学専攻修士課程修了．同年(株)富士通ソーシアルサイエンスラボラトリ入社．第五世代コンピュータの基本ソフトウェアの開発に従事．1993年より(財)新世代コンピュータ技術開発機構に出向．並列処理，特に負荷分散の実装方式に興味を持つ．