社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－ARC－157　(30)
2004－HPC－ 97　(30)
2004／3／3

耐故障／耐高負荷を考慮した並列分枝限定法

久保田　和人[†]　　仲　瀬　明　彦[†]

{kazuto, nakase}@isl.rdc.toshiba.co.jp

(株) 東芝 研究開発センター[†]

概要

マスタ・ワーカモデルを用いた並列分枝限定法に対して，ワーカージョブに優先度を与え優先度に応じてジョブを多重実行することで，耐故障／耐高負荷性能を付加した．PC クラスタを用いたシミュレーション実験により，高負荷なワーカの処理がボトルネックとなり全体の処理時間が増大する問題に対して，本手法が有効に機能する場面を確認した．また，あらかじめ想定した台数のワーカのダウンに対して，多重度を適切に設定することにより処理の停止が回避できることを確認した．

Parallel Branch and Bound Method with Fault and Load Tolerance

KAZUTO KUBOTA[†] and AKIHIKO NAKASE[†]

Research and Development Center Toshiba Cooporation[†]

Abstract

Fault tolerant and load tolerant feature is embedded to parallel branch and bound method. Our method is based on a master-worker programming model. A priority is assigned to each worker job, and high priority jobs are executed redundantly. Experimental results on a PC cluster show its fault and load tolerant feature. In some cases when heavy load workers exist, our method can reduce the total execution time.

1. はじめに

分枝限定法は，最適化問題を解く手法として広く使われている．計算機クラスタやグリッド[1] のような計算機環境の登場に伴って，より大規模な問題が高速に解けるようになり，活用の場面は益々広がりを見せつつある．しかし，計算機規模の大規模化や広域分散化はシステム全体の安定性に対して不利に働く．特定の計算機ノードが故障すると並列計算が終了しなくなる可能性があり，また，特定のノードのレスポンスが何らかの理由で遅延すると，その処理がボトルネックとなって全体の処理時間が著しく伸びる可能性がある．このような状況下でも安定して動作する分枝限定法を提供できればアプリケーションユーザの受けられる恩恵は多大なものとなる．以下では，故障とレスポンスの遅延を合わせて障害と呼ぶことにする．レスポンスを遅延させる要因は様々であるが，ここでは高負荷という言葉で代表させることにする．

計算に耐障害性を持たせる方法は大きく分けて三つある．(1) 物理的に冗長性を持たせる方法[2] は，ハードウェア自体を多重化させる方法であり，一つのハードウェアがダウンしても他のハードウェアが処理を継続することで計算のダウンを防ぐ．(2) ミドルウェアあるいは OS に耐障害性を持たせる方法[3] は，故障によって消失した処理を何らかのソフトウェア手段によって復旧する方法である．例えば，処理のスナップショットをとり，正常なハードウェア上で処理を再開する．(3) アプリケーションそのものに冗長性を持たせる方法[4][5] は，OS やミドルウェアに頼らずにアプリケーションだけで耐故障に備えるというものである．なお，これら三つの手法は排他な手法ではなく，お互いに組み合わせることで耐障害性を高めることができる．

本稿では，アプリケーションレベルで分枝限定法に対障害性を持たせる方法について述べる．並列分枝限定法はマスタ／ワーカ型モデルを仮定し，ワーカ処理を多重化することで障害に備える．しかし，全てのワーカ処理を多重化すると処理数が膨大になるので，処理に優先度を付け優先度の高いもののみ多重度を上げて実行するというアプローチをとる．多重化によるメリットは，(1) 故障でジョブが消失しても処理全体が停止しないことや，(2) 多重化したジョブうちの最も早く返ってきた結果を採用することで処理全体の遅延を防げる可能性があることなどである．本稿では処

理の詳細を述べた後に，耐高負荷，耐故障性能の評価を行う．

2. 分枝限定法と最良優先探索

分枝限定法は最適解を求める解法の一つである．解空間全体を木で表し，ルートから葉までのパスが一つの解と見なされる．例えば，最適化問題の一つであるナップザック問題は，重さ w と価値 v を持つ n 個の荷物が存在したときに，重さの合計が W 以下で価値の総和が最大となる荷物の選び方を求める問題である．分枝限定法を用いてナップザック問題を解く場合は，木の深さ d の節点を d 番目の荷物を入れるか入れないかに対応させる．木全体では，全ての荷物を入れるか入れないかという全ての場合が尽くされていることになる．この中で価値が最も高くなる荷物の入れ方，すなわち，その入れ方を示すルートから葉へのパスが求めるナップザック問題の解となる．

解の探索はルートから葉方向へ木を生成しながら行う．節点を分岐させて新たな節点を作る操作を分枝操作と呼ぶ．これは元の問題を小さな部分問題に分けることに相当する．以降，本稿では最適化問題を解くに際して，最も評価値の高くなる解を求めているものとする．個々の節点において，その節点から下に木を広げていった場合，到達の可能性がある最大の評価値を計算することができる．この値を上界値と呼ぶ．また，個々の節点において，少なくとも達成可能である評価値を計算できる．この値を下界値と呼ぶ．ある節点の上界値が他の節点の下界値を下回った時，その節点からの探索では最適解に到達できないことがわかる．従って，その節点からの分枝操作を中止する．この処理を限定操作と呼ぶ．分枝限定法では，無用な解の探索を打ち切ることで解の探索空間を限定している．

一般に木を探索する手順としては，深さ優先探索や幅優先探索などがある．これらの探索では，分枝操作をあらかじめ決められた順序で行う．一方，これとは別に個々の節点に対して探索の優先度を求め優先度の高い順に節点の処理を行う方法がある．この方法は最良優先探索と呼ばれる．これは，解が存在する可能性が高い方向の探索を優先して行うことで，早く最適解に到達することを目指すものである．優先度を決める方法として，各節点における下界値を優先度として用いる方法がある (最良下界探索と呼ばれる)．これは，下界値の小さい節点は大きい節点より最適解に到達する可能性が高いであろうという考え方に基づくものである．

並列分枝限定法への耐障害機能の付加にあたっては，冗長に実行されるワーカージョブの数を限定するために，この優先度を利用する．

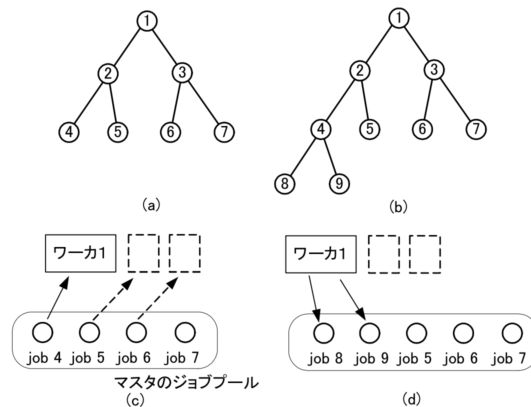

図 1 並列分枝限定法におけるジョブの割り当て．
Fig. 1 Parallel Branch and Bound method.

3. 並列分枝限定法のへの耐障害機能の付加

マスタ／ワーカ型の並列分枝限定法では，マスタがジョブプールを管理しジョブプール内には子問題 (節点) が格納される．マスタは一つ以上の子問題をジョブとしてまとめワーカに投げる．ワーカはジョブ内の子問題に対して分枝操作を行い，新たに発生した子問題および下界値をマスタに返す．この様子を図 1 に示す．図 1(a) の節点 4〜7 がこれから解かれようとしている子問題であり，個々の子問題が一つのジョブであるとする．マスタのジョブプールに格納された job 4 は，空きワーカ 1 に投げられる．ワーカ 1 は分枝操作を行う．簡単のため，ワーカは 1 回のみの分枝操作を行うものとする．ワーカ 1 は二つの節点を生成しマスタへと返す．マスタは二つの節点 8, 9 をジョブプールへと格納する (図 1(d))．この時の全体の木の様子は図 1(b) のようになる．

耐障害機能を付加するにあたって，各節点に探索の優先度を付ける．ここでは単純に下界値を優先度とする．図 1(a) の木に優先度をつけた木を図 2(a) に示す．マスタ中のジョブ管理は，図 2(c) に示すような拡張されたジョブプールと多重度リストで行う．拡張されたジョブプール (以降，単にジョブプールと呼ぶ) は，個々のジョブについてジョブ番号，優先度，実行度を保持し，優先度の順番にソートされる (下界値の低いものほど優先度が高い)．実行度にはそのジョブを実行しているワーカの台数が保持される．多重度リストにはジョブプール中のジョブの順位ごとの多重度の上限が格納される．同じジョブでもジョブプール内の順位が変動することで多重度が変化することに注意されたい．ジョブの実行は，実行度が多重度に満たないジョブを空きワーカに投げることで行う．ジョブプールからのジョブ削除は，該当するジョブを実行するワーカがマスタに結果を返した時点で行う．ジョブプール中

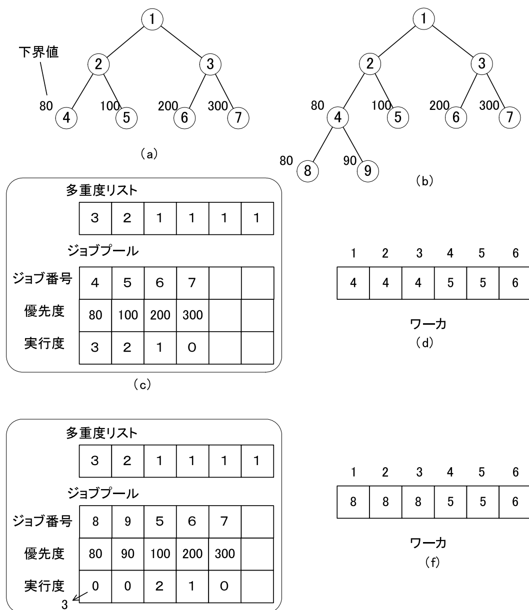

図 2 並列分枝限定法への耐障害機能の付加.

Fig. 2 Fault and load torelant Parallel Branch and Bound Method.

のジョブが無くなった時点で処理の終了となる.

図 2(c) は図 2(a) の 4〜7 のノードを実行しているときのジョブプールの様子である．多重度リストには，優先度の高い順に 3, 2, 1, 1, ... が与えられているものとする．ジョブ 4 が 3 台，ジョブ 5 が 2 台，ジョブ 6 が 1 台のワーカでそれぞれ実行されている．これらのジョブを実行しているワーカの様子を図 2(d) に示す．図 2(d) において，ワーカ 1 が処理を終了しマスタに新たなジョブ 8, 9 を返したとする．それぞれの下界値を 80, 90 とする．マスタ内のジョブプールからジョブ 4 が削除され，新たにジョブ 8, 9 が挿入される (図 2(e))．ここでジョブ 8 を 3 重に実行したとするとジョブ 8 の実行度は 0 から 3 となり，この時のワーカの様子は図 2(f) のようになる．また，この時の木の様子は図 2(b) のようになる．

耐障害機能の付加は，並列分枝限定法に二つの利益を生む可能性がある．一つ目は，高負荷なワーカが発生しても全体の計算性能を著しく低下させない効果である．高負荷なワーカが発生し，そのワーカに優先度の高いジョブが割り当てられてしまうと，有効であろう探索が停滞してしまい下界値の更新が行われず早い段階での枝刈りが行われなくなる可能性がある．耐障害機能を付加することで，有効な方向の探索は多重化され，停止，停滞しずらくなるので計算性能の著しい低下を回避できる可能性がある．二つ目は，耐故障効果である．優先度が低く多重化されていないジョブであっても，他の価値が高いジョブが終了してジョブプール内での順位が上がると多重化されて実行されるようになるので，多重度リストの先頭要素の値未満のワーカが故障しても計算は最後まで行われることになる.

4. 評価プログラムの実装

以下，本稿では耐障害機能を付加した並列分枝限定法の基本性能について評価していく．評価にあたってはナップザック問題を並列に解くコードを利用して様々なケースをシミュレーションした．ここではその詳細を述べていく.

先に述べたように，マスタ／ワーカモデルで並列分枝限定法を解く場合，各節点の分枝操作がワーカのジョブとなる．単純に各節点の一回の分枝操作を 1 ジョブとする方法もあるが，これでは粒度が細かすぎるため，いくつかの節点をまとめ，一つの節点に対する分枝操作も複数回行う．今回用いたプログラムでは，ワーカが 1 回のジョブで分枝操作を行う上限回数 (*branch_limit*) を設け，分枝操作がこの回数を超えた場合はその時点で残っている節点 (子問題) と下界値をマスタに返送する.

1 ジョブに含まれる節点数は *unit_number* という変数で管理する．節点は一時的にスタックに保持され，そこから *unit_number* 個だけ切り出されてジョブ化される．スタック中の節点が *unit_number* に満たない場合は，半分の節点をジョブ化する.

スタック中の節点のジョブ化は，ワーカにジョブを投げるタイミングで行う．まず，スタックからジョブを作成し，ジョブを管理するジョブプールに仮投入する．ジョブプールの中から評価関数を用いて実行するジョブを選択する．評価関数については後述する．選択されたジョブがスタックから作成された仮ジョブだった場合は，仮ジョブを正式にジョブプールに登録しスタックから該当する節点を削除する．そうでなかった場合は仮ジョブを破棄しスタックは元の状態のままにしておく．これによって，なるべくジョブに含まれる節点を *unit_umber* に近づける.

ジョブに含まれる情報は，ジョブ番号，優先度，実行度および節点数である．ジョブの優先度は，含まれる節点の下界値の最小値としている．ジョブプール内のジョブは優先度でソートされる．実行するジョブは，実行度が多重度リストの値より小さいものの中から選択される．選択のポリシーとしては，(1) 優先度が高いものを選ぶ，(2) 実行度の小さいものを選ぶなどが考えられる．ここでは，仮ジョブの優先度が一番高い場合は仮ジョブを、そうでない場合は、ジョブプール中で実行度がなるべく小さく，ついで優先度が高いジョブを選択している．ジョブはマスタがその時点で持つグローバルな下界値とともにワーカに投げられ、実行度が 1 加算される。ワーカからジョブ (節点群) と下界値が返送された場合，節点群はマスタのスタックに登録され下界値が更新される.

ワーカには，ジョブの処理時間をコントロールする

機能を組み込んでいる．具体的には，分枝操作の途中で wait loop を入れ見掛け上の処理時間を遅くする機能である．これによって，ワーカに負荷が発生している状況を擬似的に作り出すことができる．

処理の全体の流れをまとめると以下のようになる．

マスタプログラム

- Step 1. スタックに初期ジョブを置く
- Step 2. ワーカのメッセージが届いていれば
  - Step 2-1. ワーカを空きワーカリストに登録
  - Step 2-2. 終了したジョブをプールから削除
  - Step 2-3. 返された節点をスタックに登録
  - Step 2-4. 下界値の更新
- Step 3. スタック中の節点から仮ジョブを生成．ジョブプール中のジョブと仮ジョブの中から実行度が多重度に満たないジョブを選び，その中から評価関数を用いて実行すべき一つのジョブを選択する．
- Step 4. 選択したジョブが仮ジョブならば仮ジョブをジョブプールに正式登録．スタックから該当するジョブを削除．そうでなければ仮ジョブを破棄．
- Step 5. 選択したジョブを空きワーカで実行．実行度を 1 加算．
- Step 6. ジョブプールが空なら終了．そうでなければ Step 2. へ．

ワーカプログラム

- Step 1. マスタからジョブを受信
- Step 2. 節点をスタックに登録
- Step 3. スタック中の節点が空か，ループ回数が *branch_limit* に到達するまで以下の処理を繰り返す．
  - Step 3-1. 節点を取り出し分枝操作を実行．下界値の更新．新たに生成された節点をスタックに戻す．
  - Step 3-2. 擬似的な負荷を与えるための wait loop.
- Step 4. スタック中の節点と下界値をマスタに返送．

本プログラムで制御できるパラメータと目的を表 1 に示す．一般に *unit_number* を小さくするとジョブ内の節点の数が減り粒度は下がる．しかし，仮にジョブ内に含まれる節点が一つでもそこから生成される部分木は巨大になる可能性があるため，粒度を決めるパラメータとしては間接的と言える．一方，*branch_limit* はワーカ内での分枝操作に上限を与えるので，粒度を決定するパラメータとしては直接的である．また，マスタワーカモデルでは粒度が下がれば負荷分散は均等化される方向に進む．

5. 評価実験と考察

前節で説明したプログラムのパラメータを変化させて，本手法の挙動，性能を計っていく．ここで評価する項目は以下の二項目である．

表 1 プログラムのパラメータとその目的
Table 1 Program parameters and their purposes.

unit_number	(粒度制御，負荷分散)
branch_limit	粒度制御，(負荷分散)
多重度リスト	多重度の制御

表 2 PC クラスタのノードの仕様
Table 2 PC cluster specification.

台数	16 台 (32 CPU)
Node CPU	Dual Pentium-III 800 MHz
Node memory	1 GB
Node HDD	60 GB
Node NIC	100 BASE-T Ethernet
Switch	Cisco Catalyst 3500

- 高負荷なワーカが存在する場合の性能
- ワーカに故障が発生した場合の耐故障性能

実験は 16 台構成 32CPU の PC クラスタを用いて行った．諸元を表 2 に示す．プログラムは MPI を用いて c 言語で記述した．

ナップザック問題は，問題の性質に応じて挙動が大きく異なるため，ここでは 10 種類の問題を人工的に生成し，その平均的な挙動を観察することにした．作成したデータは，荷物の重さの最大格差を 2 倍とし，価値/重さを 1%から 4%まで変動させ，荷物の数を 100 から 250 まで変えたデータの中から，1 台での処理時間が 10 秒以上，60 秒未満のものを 10 個選択したものである．これらは，枝刈りは発生するが探索にはある程度の時間を要する問題である．

評価に先立って予備実験を行い，以降の実験で用いる *unit_number*(ジョブ中の節点数) と *branch_limit*(ジョブの粒度) をそれぞれ 100, 100000 とした。この値は，8 PE, 16 PE で実行した際に、今回利用したデータの平均的処理時間を一番短くできるものである．

5.1 高負荷な PE が存在する場合の性能

高負荷で処理が遅延する PE が存在した場合の本手法の性能を調べた．PE 数は 16, 32 とし，負荷を与える PE 数は 1，2 とした．負荷は 1(負荷なし) から 32(性能が 1/32 に低下) まで変化させ，2 台に負荷を与える場合は同一の負荷とした．多重度リストは 1111..., 2111..., 3111..., 2211... の四種類を用いた．図 3 に負荷を横軸に，処理時間を縦軸にとったグラフを示す．

耐障害性を考慮しないと (1111... のケース)，負荷が増大するにしたがって処理時間が伸びていくことがわかる．負荷を与える PE 数が 2 の方が，この傾向が顕著である．一方，耐故障性を考慮した手法では処理時間の変化はさほどなく，高負荷な PE が存在しても動作が安定している．耐障害性を考慮しない場合に処理時間が伸びるのは，負荷の高い PE が良い解を生む節点 (クリティカルパス) を抱えてしまうと，その間に他の速いノードが無駄な方向の探索を行ってしまうためである．

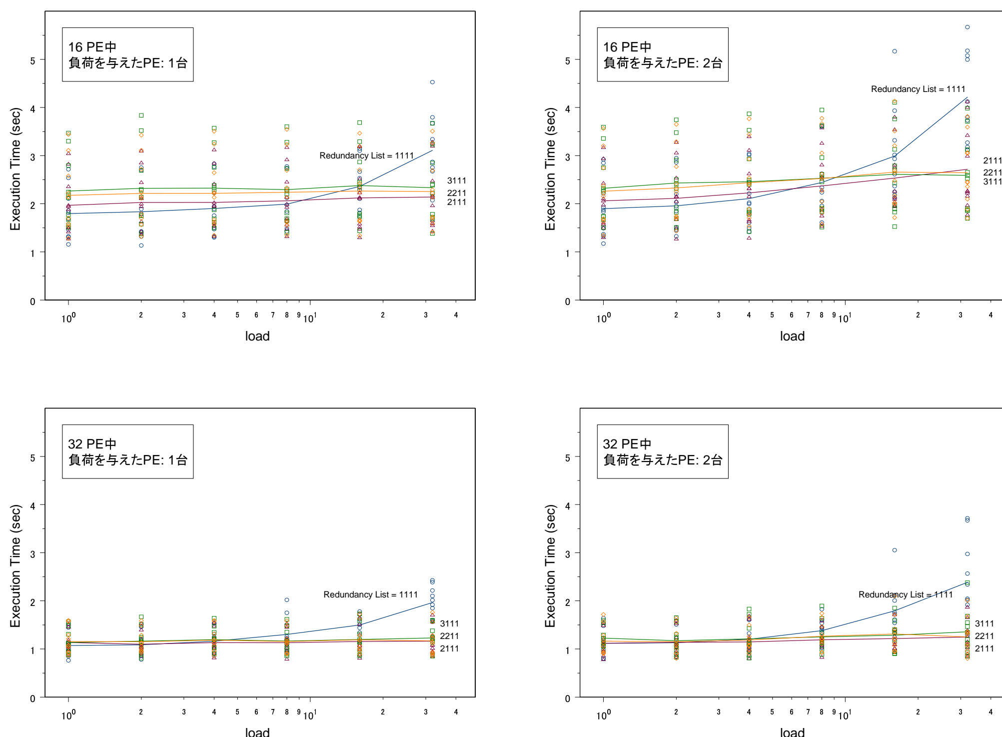

図 3 高負荷 PE 存在時の性能.
Fig. 3 Load Torelance Performance.

2 PE に負荷を与えた場合，冗長な PE 数が 2 の場合だけでなく，1 でも性能がほぼ維持できている．これは，クリティカルな計算が 2 台の負荷の高い PE に 2 重化して割り当てられるケースが確率的に少ないためだと考えられる．耐障害性を考慮することにより処理が高速になるケースは，16PE の場合は負荷が 10 程度以上，32PE の場合は負荷が 4 程度以上である．32PE の場合は耐障害性を考慮することによるペナルティがほぼゼロなので，本手法を適用することによる処理時間面での不利益はない．

5.2 耐故障性能

ワーカが障害でダウンする場合を想定して，ワーカ故障時に探索が最後まで終了するか否かを調べる実験を行った．故障ワーカを 1, 2, 4, 8, 16 と変化させ，多重度リストの内容は 1111...(耐障害なし), 2111..., 3111..., 5111..., 9111..., 17 111... と変化させた．それぞれの組み合わせについて先に利用した 10 種類のデータをそれぞれ 10 回づつ計 100 回実行し，処理が最後まで終了した回数をカウントした．利用 PE 台数は 32 台である．表 3 に結果を示す．表中の○は理論的にプログラムは正しく実行されることが期待され，実際にも 100 回正しく終了した組み合わせで，表中の×は理論的にプログラムは停止し結果も全てハングアップしてしまった組み合わせである．プログラムが期待通りの耐故障動作を行っていることが確認できる．また，数字が入っている部分に関しては，原理的にプログラムの終了が保証できなくても，実際には処理が正しく終了する場合があることを示している．例えば，32 PE 中の 8 PE が故障する場合を仮定したとき，多重度リストを 5111...(冗長な PE 数が 4) とすれば，ぼぼ，プログラムの停止を回避できていることがわかる．これは，8 台の故障 PE 存在しても，一つのジョブに故障 PE ばかり 5 台割り当てられる確率はかなり小さいことを示している．この理論的確率は，全 PE 数，故障 PE 数，多重度リストの値から計算できる．この値はプログラムの動作が確率的に保証できれば良い問題に対

表 3 耐故障性能の評価
Table 3 Evaluation of Fault Tolerance.

故障 PE 数/全 PE 数	1/32	2/32	4/32	8/32	16/32
1111...	×	×	×	×	×
2111...	○	99	81	39	10
3111...	○	○	98	85	24
5111...	○	○	○	100	87
9111...	○	○	○	○	100
17 111...	○	○	○	○	○

○：理論，計測ともにプログラムが実行完了
×：理論，計測ともにプログラムは実行停止

して有用なパラメータとなる．

6. 今後の課題

6.1 不要ジョブ，過剰に多重化されたジョブの処理

ジョブが多重化されていた場合，最初の答えが返ってきた時点で多重化されて実行された残りのジョブは不要になる．この時，積極的に不要ジョブを kill すれば，空きワーカを増やすことができる．また，新しいジョブがジョブプールに投入された時点で，多重化して実行されていたジョブの優先度の順位が下がり，実行度が多重度リストの値を下回るケースが生じる．これは，ジョブが過剰に多重化されていると見なすことができ，過剰分を kill することで先のケースと同様に空きワーカを増やすことができる．空きワーカを増やすことは探索の範囲を広げ高速化に結び付く反面，ワーカを kill する処理はオーバーヘッドを伴う可能性がある．これらの処理を付加することの有効性について検証していきたい．

6.2 開発環境の整備

既存の分枝限定法のコードに本手法を適用する場合，ジョブキュー管理に関するコードの追加もしくは変更が必要となる．この作業はさほど煩雑ではないが，この処理がミドルウェア化，API 化されていればユーザのプログラム開発コストは大きく削減される．今回評価に使ったコードから分枝限定法本体の部分と耐障害性のために付け加えたジョブキューやワーカの管理部分のコードを切り分け，耐障害性を持つ分枝限定法記述用ミドルウェア，API を構築したいと考えている．

6.3 実用環境での利用，評価に向けて

本手法が有効に活用される場面は，不安定さの大きい広域分散環境にあると考えられる．今回の評価は，手法の基本性能を知りたいという動機から，単純な負荷，故障モデルについて PC クラスタ上で評価を行った．今後は，より実践に近い広域分散環境をモデリングした評価，あるいは広域分散環境上そのものでの評価を行っていきたい．現在，広域分散環境上で RPC (Remoto Procedure Call) ベースのプログラミング環境やマスタ／ワーカ処理を実現するミドルウェアやシステムが開発されている[6)7)8)]．本手法とこのようなミドルウェア，システムとの融合についても考えていきたい．また，Condor[3)] に代表される耐故障をミドルウェアレベルで実現するシステムとの比較評価，あるいは組み合わせについても考えて行きたい．

7. まとめ

本稿ではマスタ／ワーカモデルにもとづく並列分枝限定法に耐故障／耐高負荷性能を持たせる手法について述べ，その基本性能の評価を行った．手法のポイントは二点ある．一点目は，ジョブを多重化して実行し，ジョブに優先度を付けて多重度を制御していることである．これによりワーカの故障に備えられる他，負荷の高いワーカが存在した際にも計算性能の著しい低下を防ぐことができる．二点目は，ジョブの終了をもってジョブプールからジョブを削除する枠組みを採用していることである．ジョブの優先度の相対的順位が動的に変化するので，優先度が低く 1 ワーカで実行されて停滞／停止しているジョブも，いずれは多重化されて実行されるようになる．

実験では，本手法の基本的な挙動について示した後，耐高負荷機能や耐故障機能が働いていることを示した．今後は，今後の課題で述べた事項に取り組み，本手法のさらなる評価および実用化を図っていきたい．

謝辞

本研究を進めるにあたり貴重なご意見をいただいた，産業技術総合研究所の田中良夫研究員，中田秀基研究員，首藤一幸研究員に深く感謝いたします．

参考文献

1) Ian Foster and Carl Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1988.
2) 南谷崇, フォールトトレラントコンピュータ, オーム社, 1991.
3) Condor Project Homepage. http://www.cs.wisc.edu/condor/.
4) Adriana Iamnitchi and Ian Foster, "A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems," Proc. of the International Conference on Parallel Processing 2000, pp.4–14, 2000.
5) Yaohang Li and Michael Mascagni, "Improving Performance via Computational Replication on a Large-Scale Computational Grid," Proc. of CCGrid 2003, pp.442-448, May 2003
6) 中田秀基, 田中良夫, 松岡聡, 関口智嗣, "GridRPC を用いたタスクファーミング API の試作", 情処研報, 2003-HPC-96, pp.61–66, 2003.
7) 首藤一幸, 大西丈治, 田中良夫, 関口智嗣, " 計算機資源の流通および集約のための P2P ミドルウェア", 情処研報, プログラミング研究会, PRO45-1, pp.1–14.
8) 秋山智宏, 中田秀基, 松岡聡, 関口智嗣, "グリッド環境に適した並列組み合わせ最適化システム jPoP における分枝限定法の実装", SPA 2003, 2003.
9) SETI@Home: Search for Extraterrestrial Intelligence at Home. http://setiathome.ssl.berkeley.edu/.