並列分枝限定法における耐故障アルゴリズムの評価

川 上 健 太[†] 合 田 憲 人[†]

計算機クラスタやグリッドで大規模な処理を行った場合，計算ノードで障害が発生すると計算全体が停止してしまう恐れがあり，並列計算における耐故障性は非常に重要なテーマである．本稿では，マスタ・ワーカモデルを用いた並列分枝限定法における耐故障性を実現するための手法であるタスクの多重化による手法とワーカの監視による手法の性能評価について述べる．タスクの多重化による手法には，実行時間を削減するために新たに不要な処理の中断機構を実装した．本性能評価の結果，中断機構により，実行時間が大幅に短縮されることが確認された．また，両手法ともに高い耐高負荷及び耐障害性を有することが確認された．

Evaluation of Fault-Tolerant Parallel Branch and Bound Algorithms

Kenta Kawakami[†] and Kento Aida[†]

Fault-tolerance is one of crucial issues for large-scale computing environments, such as a PC cluster or the Grid, where a fault on one computer might stop the whole computation. This paper evaluates two fault-tolerant algorithms, the task replication algorithm and the worker monitoring algorithm, for a parallel branch and bound method parallelized by the master-worker paradigm. For the task replication algorithm, the performance of the mechanism to avoid redundant computation is also evaluated. The results showed that this mechanism significantly reduced computation time and that both fault-tolerant algorithms effectively work to improve fault-tolerance and load-tolerance.

1. はじめに

分枝限定法は最適化問題の解法としてオペレーションズリサーチ，制御工学，マルチプロセッサスケジューリング等，様々な工学問題を解くために用いられている手法である．一般に最適化問題は対象とする問題が大規模になると非常に大きな計算時間を要するため，分枝限定法を並列化することにより処理の高速化を行う事例が報告されている[2)]．

計算機クラスタやグリッドで大規模な処理を行った場合，計算ノードで障害が発生すると計算全体が停止してしまう恐れがあり，並列計算における耐故障性は非常に重要なテーマである．並列分枝限定法においても耐故障性を付加するための手法として，タスクを多重実行する方法が提案されている[1)]．しかし，この手法を使用した場合，多重度（同じ処理を実行させるワーカの台数）を増加させると実行時間も増大してしまうという問題がある等，並列分枝限定法における耐故障アルゴリズムにはまだ解決すべき問題が多い．

本稿では，マスタ-ワーカモデルを用いた並列分枝限定法における耐故障性を向上させる手法について議論する．具体的には，まずタスクを多重化する手法について，実行時間のオーバヘッドを削減するために，マスタの不要な処理を中断させる機構を考案し，性能評価を行う．次に本稿では，ワーカの監視，即ち計算に参加しているワーカが定期的に処理状況の報告を行い，一定時間報告が行われないマシンを障害が発生したと判断する手法を考案し，性能評価を行う．

これら 2 つの手法を並列分枝限定法に適用させてPC クラスタ上で実装し，実行時間の評価を行った結果，タスクの多重化手法における不要な処理の中断機構は，プログラムの実行時間を大幅に短縮させることが確認された．また，タスクの多重化による手法およびワーカの監視手法は，ともに高い耐故障／耐高負荷性能を有することが確認された．両手法を比較すると，タスクの多重化による手法は，ワーカ数が十分に多い場合は有効であるが，ワーカ数が少ない場合は冗長処理のため，ワーカの監視による手法の方が高い性能を示すことが確認された．

† 東京工業大学
Tokyo Institute of Technology

2. マスタ・ワーカ方式を用いた並列分枝限定法

分枝限定法では，親問題の解の探索範囲を分割して子問題を生成し，各子問題ごとに目的関数の下界値，上界値，解を計算するという操作を繰り返すことにより最適解を探索する．マスタ・ワーカ方式で分枝限定法を並列実行する場合，マスタが解の探索範囲である探索ツリーを管理し，探索ツリー上の子問題を複数のワーカに割り当てることにより並列処理を実現する．マスタがワーカにタスクを割り当てる方法の一つに最良下界値優先探索がある．これは，探索ツリー上の各節点の下界値を優先度とし，優先度の高い節点から順に処理を行うというものである．

3. 耐故障アルゴリズム

本節では，本稿で議論する２つの耐故障アルゴリズムについて説明する．これらは，タスクの多重化による手法と，ワーカの監視による手法からなる．

3.1 タスクの多重化による手法

タスクの多重化による手法では，予め設定しておいた多重度（同じ処理を実行させるワーカの台数）のリストに従って，マスタが同一のタスクを複数のワーカに多重に割り当てる．これにより，任意のワーカに障害が発生した場合でも，そのワーカが処理していたタスクは他ワーカ上で実行されているため，正しい計算結果を得ることができる．1) では，タスクの多重度を決定する際に下界値を使用していた．これは，下界値が小さい接点の方が最適解の存在する可能性が高いため，より重要であると予測されるからである．本稿でも優先度として下界値を使用する．マスタは図１の様に優先度によってソートされたタスクのキューを管理し，多重度リストの値に応じて複数のワーカに同一のタスクを割り当てる．このうち，最も早く計算の終了したタスクの結果がワーカからマスタに通知された時点で，タスクはキューから削除される．

タスクの多重化による手法は，多重度リストの値が増大するに従って複数のワーカが同一タスクの計算を行うため，実行時間も増大してしまう．そのため，本稿では新たに不要な処理の中断機構を導入する．本機構では，任意のワーカ上でタスクが正常終了した場合は，同じタスクを実行している他のワーカ上の処理は不要となるためそれらを中断する．

3.2 ワーカの監視による手法

ワーカの監視による手法では，タスクを多重化しない代わりにマスタがワーカの処理状況を常に監視し，ワーカの計算状況が全く変化しない場合は，それらに

タスク番号	4	8	5	6	7
優先度	300	250	200	100	80
多重度	3	1	1	1	1
ワーカ	1	2	3	4	5
実行タスク	4	4	4	5	8

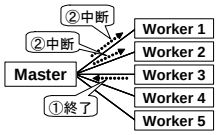

図 1 タスクの多重化による手法 (ワーカ 1,2,3 の中で，ワーカ 3 の処理が最初に終了した場合，ワーカ 1,2 の処理は中断される)

障害が発生していると推測する．本稿では，その様な障害が推測されるワーカを，障害が疑われるという意味でサスペクト (suspect：容疑者) と呼ぶ．各ワーカには障害の可能性を示すパラメータ $suspect\_cnt$ を設定し，その値があらかじめ設定しておいた閾値を越えた場合は，マスタはそのワーカをサスペクトと判断する．

マスタ・ワーカ型の並列分枝限定法において，ワーカの処理の大部分を占めるのは，分枝操作で生成された子問題の評価値 (下界値，上界値等) を計算する処理である．即ち，ワーカに割り当てられるタスクは子問題の処理の集合とみなすことができる．そこで，以降ではタスクの処理は二等分する事が可能であるとし，ワーカが前半の処理を行っている状態を状態 1，後半の処理を行っている状態を状態 2 と呼ぶ．つまり，分枝操作によって生成された子問題の数を n(n：偶数) とすると，最初の $n/2$ 個の子問題の処理が状態 1，後半の $n/2$ 個の処理が状態 2 となる．各ワーカは，状態 1 から状態 2 に移行する際に，状態が変化した旨をマスタに通知する．

マスタは，各々のワーカにタスクを割り当てる際にその時点での全てのワーカの状態を表 $status\_list$ に記録しておく．任意のワーカ上でのタスク処理が正常終了した場合，$status\_list$ を参照し，そのタスクの処理が開始された時点の全ワーカの処理状況と現在の全ワーカの処理状況とを比較し，状態が全く変化していないワーカの $suspect\_cnt$ を 1 増やす．この操作によって $suspect\_cnt$ があらかじめ設定しておいた閾値を越えた場合は，そのワーカをサスペクトとし，マスタはそのワーカに割り当てられていたタスクを他のアイドルワーカに割り当てる．本手法では，あるワーカが一つのタスクを終了した時間で，その半分の処理すら完了していないワーカはサスペクトの候補となる．

実際にサスペクトが検出される例を図 2 を使用して説明する．図 2 の (a) はワーカ 1 がタスク 9 の前半の処理を開始した時点での各ワーカの処理状況である．マスタはこれを $status\_list$ に記録しておく．(b) はワーカ 1 がタスク 9 の後半の処理を開始した時点の状況，(c) はワーカ 1 がタスク 9 の後半の処理を終了した時点の状況である．マスタはワーカ 1 からタス

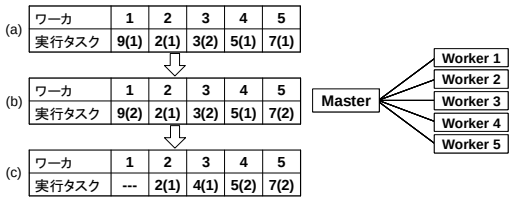

図 2 サスペクトが検出される例（実行タスクの括弧内の数字は処理の状態を意味する）

ク 9 の処理結果が返って来た時点で，現在の状況 (c) とワーカ 1 にタスク 9 を与える前の状況 (a) とを比較し，状況が全く変化していないワーカの *suspect_cnt* を 1 増やす．この例では，ワーカ 2 の状態が全く変化していないので，ワーカ 2 の *suspect_cnt* を 1 増やす．もし，この操作によってワーカ 2 の *suspect_cnt* が閾値を越えた場合は，マスタはワーカ 2 が行っているタスクであるタスク 2 をワーカ 1 に処理させる．これによって，全てのタスクは必ず最後まで実行されることが保証される．

この手法はサスペクトが検出された場合にのみタスクの多重化を行うため，先程のタスクの多重化と比較して，複数ワーカ上での冗長処理は大幅に削減されると期待される．また，サスペクトは障害が発生した場合だけでなく，ワーカ同士の処理能力の差が 2 倍以上である場合に検出されるため，性能の低いワーカ上の処理がボトルネックとなることを防ぐことも出来る．

ワーカの監視による手法は，ワーカのタスクの処理時間に閾値を設けて障害を検出する方法と類似している．しかし，閾値を設けた場合，ある時点で全てのワーカの処理能力が低下した場合に全てのワーカが障害と判断されてしまう可能性がある．一方，本手法では各ワーカの処理時間を比較して障害を検出するため，その様な状況により柔軟に対応する事が出来る．

今回は，ワーカのタスクを 2 等分することで障害の検出を行ったが，n 等分 ($n > 2$) することでより精密な制御を行うことが可能である．何故なら，ワーカがマスタに報告する間隔が短い程，障害は早期に検出されるからである．

4. 実 装 方 法

本節では Ninf[3] により実装される 2 つの耐故障アルゴリズムの詳細について述べる．Ninf は，クライアントマシンからサーバマシン上の計算ルーチン (Ninf_executable) の実行を依頼する RPC を実現するためのライブラリである．ここで，クライアントから Ninf_executable の呼び出し処理は NinfCall と呼ばれる API を通して行われる．本稿が対象とするアプリケーションの Ninf による実装[2] では，Ninf のクライアントプログラムがマスタ，サーバプログラムがワーカに相当しており，マスタからワーカへのタスク（子問題）割り当ては，NinfCall により実現される．

4.1 タスクの多重化による手法

タスクの多重化による手法の詳細なアルゴリズムを以下に示す．

[**マスタの処理**]

(1) 多重度リスト等の初期パラメータを読み込む．
(2) 親問題から子問題を生成しそれをタスクキューに入れる．
(3) アイドルワーカが存在する場合は，以下の手順でアイドルワーカにタスクを割り当てる．
  (a) キュー内部のタスクを優先度に応じてソートする．
  (b) キューの先頭のタスクを選択する．
  (c) 選択したタスクの実行度 (そのタスクを実行しているワーカの台数) と多重度とを比較し，実行度が多重度よりも小さい場合は，アイドルワーカにそのタスクを割り当て，その実行度を 1 増加させる．そうでない場合は，キューの次のタスクを選択し，(3-c) を再度実行する．
(4) ワーカからの通信待ち状態へと移行する．
(5) ワーカから計算終了が通知された場合，子問題計算結果を回収し，そのタスクをタスクキューから削除する．また，そのタスクの実行度が 2 以上である場合は，同じタスクを実行している他のワーカの処理を中断させる．
(6) 子問題計算結果をもとに終了判定を行う．
(7) 終了していない場合 (3) に戻り，終了条件を満たすまで繰り返す．

[**ワーカの処理**]

いずれの場合も，タスクを処理している最中にマスタから中断命令が来た場合は処理を中断する．

(1) マスタから割り当てられた子問題に対し，分枝操作を行って子問題を生成し，生成された子問題の上界値と下界値および解を計算する．
(2) 限定操作を行い，残りの子問題と暫定値をマスタへ返し，待機状態となる．

4.2 ワーカの監視による手法

ワーカの監視による手法の詳細なアルゴリズムを以下に示す．

[**マスタの処理**]

(1) 初期パラメータを読み込む．
(2) 親問題から子問題を生成しそれをタスクキュー

に入れる．

(3) アイドルワーカが存在する場合は，以下の手順でアイドルワーカにタスクを割り当てる．
  (a) キュー内部のタスクを優先度に応じてソートする．
  (b) キューの先頭のタスクを選択する．
  (c) 選択したタスクが，他のワーカで実行されていないか，またはサスペクトによって実行されている場合は，他の全てのワーカの処理状況を *status_list* に記録した後に，そのタスクをアイドルワーカに割り当てる．そうでない場合は，キューの次のタスクを選択し (3-c) を再度実行する．

(4) ワーカからの通信待ち状態へと移行する．

(5) ワーカから状態 2 へと移行する旨が通知された場合，*status_list* 内の該当するワーカの情報を更新する．

(6) ワーカから計算終了が通知された場合，そのワーカの *suspect_cnt* を 0 とする．そのワーカから子問題計算結果を回収し，そのタスクをタスクキューから削除する．*status_list* を参照し，そのタスクの処理が開始された時点の全ワーカの処理状況と現在の全ワーカの処理状況とを比較し，状態が全く変化していないワーカの *suspect_cnt* を 1 増やす．この操作によって *suspect_cnt* が設定しておいた閾値を越えた場合は，そのワーカをサスペクトとする．

(7) 子問題計算結果をもとに終了判定を行う．

(8) 終了していない場合 (3) に戻り，終了条件を満たすまで繰り返す．

[ワーカの処理]

(1) マスタから割り当てられた子問題に対し，分枝操作を行って子問題を生成する．

(2) 生成された n 個の子問題の中で，$n/2$ 個の子問題の上界値と下界値および解を計算する．

(3) 処理の半分が終了したことをマスタに通知する．

(4) 生成された n 個の子問題の中で，残りの $n/2$ 個の子問題の上界値と下界値および解を計算する．

(5) 限定操作を行い，残りの子問題と暫定値をマスタへ返し，待機状態となる．

ワーカからマスタへ処理状況を伝達する際には，Ninf の callback 機能を使用した[3]．また，今回は *suspect_cnt* の閾値を 1 とした．

5. アルゴリズムの評価

本節では，前節までで説明した 2 つの手法の PC クラスタ上での性能評価結果を報告する．表 1 に性能評価に用いた PC クラスタの性能を示す．表 1 のノード A 上でマスタプロセスを実行し，ノード B 上でワーカプロセスを実行した．本評価のベンチマーク問題としては，BMI 固有値問題[4] を用いた．

表 1 PC クラスタのノードの仕様

nodeA (1 nodes)	Intel Xeon 2.4GHz ×2CPU, Memory 512MB, Linux 2.4.20
nodeB (16 nodes)	Pentium III 1.4GHz ×2CPU, Memory 512MB, Linux 2.4.10

表 2 各手法の実装による実行時間の変化 (多重度リストの値は全て 1 とした．また，括弧内は実装前の実行時間を 1 とした場合の実行時間の比である．)

ワーカ数	実行時間 [sec]		
	実装前	タスクの多重化	ワーカの監視
1	1217	1220(1.002)	1241(1.020)
2	612	614(1.003)	638(1.042)
4	311	312(1.003)	326(1.048)
8	162	162(1.000)	172(1.062)
16	87	87(1.000)	95(1.092)
24	62	63(1.016)	74(1.194)
32	52	52(1.000)	66(1.269)

5.1 基本性能の評価

本実験では，各手法を実装したことによる実行時間のオーバヘッドを計測するために，障害が発生しない条件下での実装前のプログラムと各手法を実装した後のプログラムの実行時間の比較を行った．タスクの多重化手法における多重度リストの要素を全て 1 とした場合の実行時間を表 2 に示す．多重度リストの値を全て 1 としたのは，タスクの多重化機構を導入することによる純粋なオーバヘッドを計測するためである．また，多重度リストを変化させた場合の実行時間のグラフを図 3 に示す．

表 2 から，タスクの多重化を導入することによる実行時間のオーバヘッドは殆ど無いことがわかる．一方，図 3 から，多重度リストの値を増加させた場合は実行時間も増加することがわかる．これは，多重度が増加すると冗長な処理も増加するためである[1]．表 2 より，ワーカの監視による手法ではサーバの台数が増加するに従って，実行時間のオーバヘッドは増加した．ワーカの監視による手法では，ワーカは割り当てられたタスクの処理が半分終了する度に，マスタにそれを通知する．即ち，ワーカの台数が増加するにつれてマスタ-ワーカ間の通信回数も増加するために，実行時間も増大するのである．両手法を比較すると，ワーカ数が比較的少ない場合は，タスクの多重化による手法では不要な処理の中断を行う場合でも冗長な処理が残るため，

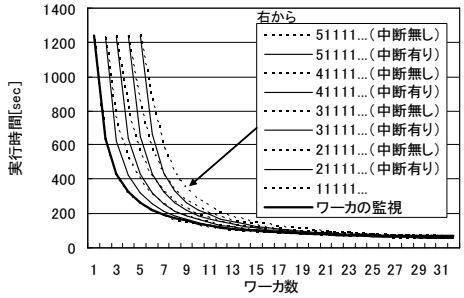

図 3 多重度リストを変化させた場合の実行時間の変化
(凡例の数字は多重度リストの値を意味する)

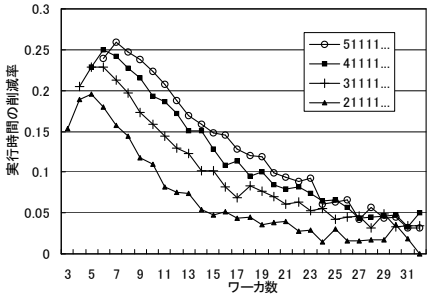

図 4 不要な処理の中断による実行時間の削減率
(凡例の数字は多重度リストの値を意味する)

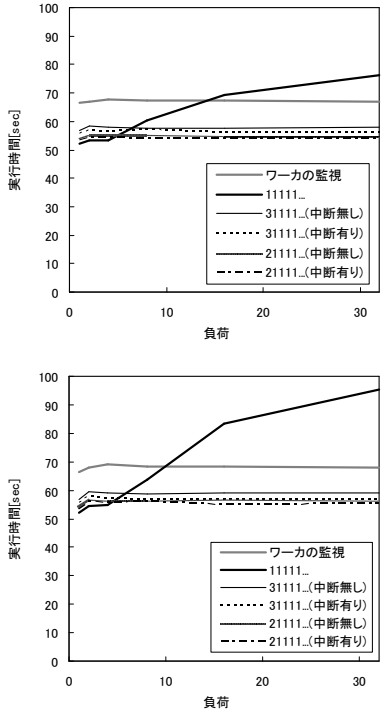

図 5 高負荷なワーカによる実行時間の変化 (上段は負荷を与えたワーカが 1 台の場合，下段は 2 台の場合．また，凡例の数字は多重度リストの値を意味する)

ワーカの監視による手法がよりよい性能を示すことがわかる．一方，ワーカ数が十分に多い場合は，ワーカの監視による手法では通信オーバヘッドが大きいためタスクの多重化による手法がよい性能を示した．

5.2 不要な処理の中断機構の評価

タスクの多重化による手法において，不要な処理の中断を行うことによる実行時間の削減率を図 4 に示す．削減率は以下の式で計算した．

削減率=(A-B)/A

(A：中断無しの実行時間，B：中断有りの実行時間)

図 4 から，多重度リストの値が増加するに従って，削減率も増加していくことがわかる．これは，多重度の増加は冗長な処理の増大を招くため，不要な処理の削減によって得られる効果も大きくなるためである．その一方で，多重度リストの値に関わらず，ワーカ数が増加するにつれて削減率は低下した．これは，ワーカの総数が増加すると，処理全体に占める冗長な処理の割合が相対的に低下するためである．この結果から，不要な処理の中断は，同一タスクを実行しているワーカの割合がワーカの総数に対して高い場合に効果的であるといえる．

5.3 耐高負荷性の評価

本実験では，高負荷なワーカが存在する場合の実行時間の変化を計測した．ワーカの負荷は 2(性能が 1/2 に低下) から 32(性能が 1/32 に低下) まで変化させ，各ケースについて 10 回の平均実行時間を計測した．負荷を与えるワーカに関しては，割り当てられたタスクの処理を m 回繰り返すことで，負荷が m である状態をシミュレートした．全ワーカ数は 32 とし，負荷を掛けるワーカの台数は 1，2 とした．また，タスクの多重化手法における多重度リストは 1111...，2111...，3111 の 3 種類を用いた．負荷を与えたワーカ上の負荷と実行時間の関係を図 5 に示す．

タスクの多重化による手法では，高負荷なワーカの存在は実行時間に大きな影響を与えないことが報告されており[1)]，今回の実験でも同様の結果が得られた．ワーカの監視による手法についても，高負荷なワーカによる実行時間への影響を抑える効果があることが確認された．これは，高負荷なワーカはマスタからサスペクトとみなされ，他のワーカによってその処理が代行されるからである．

5.4 耐故障性の評価

本実験では，計算を行っている最中に，ワーカの処理が停止した場合の実行時間の変化を計測した．停止させるワーカは，最初にタスクを受け取った時点で無限ループに入らせることで計算を停止させた．全ワーカ数は 32 とし，停止させるワーカの台数は 1，2，4，8，

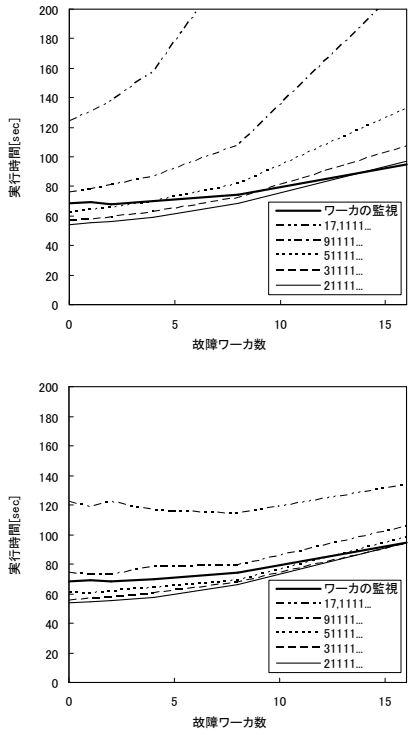

図 6 故障ワーカの存在による実行時間の変化 (上段はタスクの多重化手法において不要な処理の中断をしない場合，下段は中断した場合．また，凡例の数字は多重度リストの値を意味する)

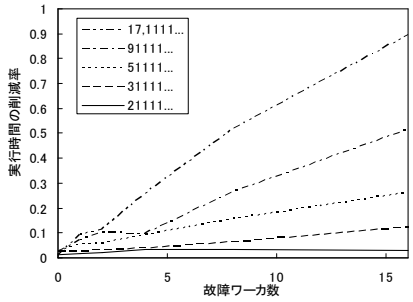

図 7 故障ワーカの存在による削減率の変化
(凡例の数字は多重度リストの値を意味する)

16 と変化させた．各ケースについて 10 回の平均実行時間を計測した．タスクの多重化手法における多重度リストは 2111...，3111...，5111...，9111...，17 111... の 5 種類を用いた．

故障ワーカ数毎の実行時間の変化を図 6 に示す．また，故障ワーカ数毎の削減率の変化を図 7 に示す．図 7 から，故障ワーカ数が増加する程，タスクを多重化させた場合の不要な処理の中断効果が大きいことがわかる．特に，多重度の値が大きい程この傾向は顕著となった．これは，故障ワーカ数が増加した場合，計算に参加しているワーカ数が減少するために，多重度リストの値が相対的に大きくなるためである．

図 6 から，ワーカの監視による手法も高い耐故障性を有することが確認された．また，別途行った実験で 32 台中 n 台のワーカに障害が発生した場合と，障害が発生せずに $32-n$ 台のワーカで実行した場合の実行時間を比較したところ，両者の差は最大でも 3% であった．

6. ま と め

本稿では，マスタ・ワーカモデルを用いた並列分枝限定法における耐故障性を実現するための 2 つのアルゴリズムの性能を評価した．1 つめの手法であるタスクの多重化による手法には，新たに不要な処理の中断機構を導入した．この操作によって，実行時間が大幅に削減されることが確認された．もう 1 つの手法であるワーカの監視による手法に関しては，高負荷なワーカや障害が発生したワーカが存在しても，安定して処理が実行されることが確認された．両手法を比較すると，タスクの多重化による手法は，ワーカ数が十分に多い場合は有効であるが，ワーカ数が少ない場合は冗長処理のため，ワーカの監視による手法の方が高い性能を示すことが確認された．

タスクの多重化による手法に関しては，多重度を処理の状況に応じて変化させることで，実行時間をより削減できると考えられる．ワーカの監視による手法に関しては，サスペクトと判断するタイミング等を変化させて評価を行う予定である．

参 考 文 献

1) 久保田和人, 仲瀬明彦:耐故障／耐高負荷を考慮した並列分枝限定法と基本性能の評価，情報処理学会論文誌, Vol.45, No. SIG11(ACS 7), pp. 171-181(2004).
2) Kento Aida, Yoshiaki Futakata, Shinji Hara, "High-performance Parallel and Distributed Computing for the BMI Eigenvalue Problem" Proc. 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002), Apr. 2002
3) Satoshi Matsuoka, Hidemoto Nakada, Mitsuhisa Sato, Satoshi Sekiguchi, "Design issues of Network Enabled Server Systems for the Grid" Grid Computing – GRID 2000, Springer-Verlag, LNCS 1971, pp.4-17
4) H. Fujioka and K. Hoshijima. Bounds for the bmi eigenvalue problem. Trans. of the Society of Instrument and Control Engineers, 33(7):616.621,1997.