バックトラックに基づく動的負荷分散の明示的スタックによる実現

寺元 悠馬　平石 拓　八杉 昌宏

木探索プログラムは，探索関数を再帰的に呼び出して記述されるほか，再帰を用いず，未探索ノードを明示的スタックで管理して記述されることもある．タスク並列言語はこうした木探索プログラムの並列化に有効であるが，その多くは再帰呼び出しベースの記述を前提とするため，明示的スタック方式のプログラムを並列化するには再帰呼び出しベースへの全面的な書き換えを要する．一時的バックトラックに基づく独特な動的負荷分散により高い性能を実現するタスク並列言語 Tascell も，この種の問題を抱える．本研究では，明示的スタックによる探索の記述スタイルを保ったまま，すなわち再帰呼び出しベースに書き換えることなく Tascell の動的負荷分散機構を利用可能にする言語拡張を提案する．状態復元を入れ子関数で行う方式と，それを排してインライン展開する方式の 2 種類の実装を行った．N 女王問題とペントミノパズルで評価し，明示的スタック方式でも再帰版と同等の並列性能を実現できることを示した．

1　はじめに

多数の計算ノードを用いた並列処理は，探索問題をはじめとする大規模な計算を高速に解くために必要不可欠な手法となっている．プロセッサあたりのコア数やクラスタを構成するノード数は増加の一途にあり，こうした分散・高並列環境でアプリケーションの性能を十分に引き出すには，環境に適した並列実装が求められる．そのための並列プログラミング言語・ライブラリの研究開発が活発に行われており，探索問題に代表される不規則な計算に適したアプローチの一つがタスク並列処理である．

タスク並列処理では，タスクの生成・管理に伴うオーバヘッドをいかに抑えるかが課題となる．広く用いられている手法の一つが Lazy Task Creation（以下，LTC）[7] であり，Cilk [3] をはじめとする多くのタスク並列言語で採用されているが，ワーカ間で共有するタスクプールの管理コストに加え，タスク生成のたびに探索状態を保持するワークスペースを確保・初期化するコストも残る．一時的バックトラックに基づくタスク並列言語 Tascell [5] は，これに代わるアプローチである．Tascell ワーカは，他のワーカからタスク要求を受けない限り逐次実行を続けるが，要求を受けるとタスク生成が可能な最古の地点まで一時的にバックトラックし，そこでタスクを生成するという独特な実行機構を持つ．これにより，これらの問題が解消され，Tascell は特に木探索プログラムの並列化において，LTC ベースの Cilk を上回る性能を達成している [5].

木探索プログラムは，再帰呼び出しを用いて記述されることもあるが，ごみ集めや SAT ソルバをはじめとして，再帰呼び出しを用いず未探索ノードを明示的スタックで管理して記述されることも多い．しかし，従来の Tascell で木探索プログラムを並列化するには，探索の 1 ステップごとに探索関数を再帰的に呼び出すプログラミングスタイルを採らなければならない．そのため，後者のスタイルで書かれたプログ

An Explicit-Stack Implementation of Backtracking-Based Dynamic Load Balancing

Yuma Teramoto, 京都橘大学, Dept. Kyoto Tachibana University.
Tasuku Hiraishi, 京都橘大学, Dept. Kyoto Tachibana University.
Masahiro Yasugi, 九州工業大学, dept. Kyushu Institute of Technology.

```
1  int nqueens(int n, int k)
2  {
3    int s = 0;
4    for (int i = k; i < n; i++) {
5      int ai = a[i];
6      int li = n-1-ai+k, ri = ai+k;
7      if (lb[li] || rb[ri]) continue;
8      if (k == n-1) { s++; continue; }
9      /* place a queen at (row k, column ai) */
10     lb[li] = 1; rb[ri] = 1; swap(a[i], a[k]);
11     /* search the next row recursively */
12     s += nqueens(n, k+1);
13     /* remove the queen (backtrack) */
14     lb[li] = 0; rb[ri] = 0; swap(a[i], a[k]);
15   }
16   return s;
17 }
```

図 1　N 女王問題の全解探索の再帰呼び出しによる実装

ラムを Tascell で並列化するには，まず再帰呼び出しベースへ全面的に書き直す必要があり，プログラマにとって大きな障壁となる．この制約は Tascell に固有のものではない．Cilk [3]，OpenCilk [11]，OpenMP の task 構文 [9]，X10 [1] などの fork-join 型のタスク並列言語も，関数呼び出しによる再帰的な分割には自然に対応する一方，明示的スタックによる探索を並列化するには同様に再帰ベースへの書き換えを要する．

例えば，図 1 と図 2 は，いずれも N 女王問題の全解探索を C 言語で記述したものであり，同じ探索木を同じ順序で辿って同じ解の数を返す．両者はともに，配列 `a` に列の順列を保持し，`a[0,k)` を第 0 行から第 $k-1$ 行に置いたクイーンの列とする．配列 `lb` と `rb` は 2 方向の対角線が使用済みかどうかを表すフラグであり，`swap(`x`,`y`)` は x と y を交換するマクロである．

両者で異なるのは，探索状態の管理方法である．図 1 の再帰版は，第 k 行の探索を関数呼び出し `nqueens(n,k+1)` で表し，第 k 行のループ変数 `i` と，復帰時に盤面を巻き戻すために必要な `li`，`ri` を，呼び出し元の関数フレーム，すなわちコールスタックに残す．バックトラックは関数からの復帰そのものであり，戻った先で `lb`，`rb`，`a` が元に戻される．

一方，図 2 のスタック版は，これらを行（探索の深さ）ごとに 1 つのスロットを持つ配列 `stk` に明示的に保持する．各スロットは，その行の探索ループの残り区間 `[i,i1)` と，その行から次の行へ進む際に置いたクイーンを元に戻すための情報（`qi`，`li`，`ri`）を兼ねる．探索は `while (k >= 0)` の単一のループで進み，現在の行の候補が尽きたらスロットに保存された情報をもとに 1 つ上の行まで戻って盤面を巻き戻す．再帰版でコールスタックが暗黙に担っていた役割を，スタック版では `stk` が明示的に担っていることになる．

```
1  struct level {
2    int i, i1;       /* remaining columns [i, i1) to try */
3    int qi, li, ri;  /* how to undo the queen placed here */
4  };
5
6  int nqueens(int n)
7  {
8    struct level stk[MAXN];
9    int k = 0, s = 0;
10   stk[0].i = 0; stk[0].i1 = n;          /* the first row */
11
12   while (k >= 0) {
13     struct level *f = &stk[k];
14     if (f->i >= f->i1) {
15       if (k == 0) break;                /* whole search done
          */
16       /* subtree done: back up and remove the queen */
17       k--; f = &stk[k];
18       lb[f->li] = 0; rb[f->ri] = 0; swap(a[f->qi], a[k]);
19       continue;
20     }
21     int i = f->i++;                     /* the next column */
22     int ai = a[i];
23     int li = n-1-ai+k, ri = ai+k;
24     if (lb[li] || rb[ri]) continue;
25     if (k == n-1) { s++; continue; }
26     /* place a queen at (row k, column ai) */
27     lb[li] = 1; rb[ri] = 1; swap(a[i], a[k]);
28     /* save how to undo */
29     f->qi = i; f->li = li; f->ri = ri;
30     /* push the loop of the next row */
31     k++; stk[k].i = k; stk[k].i1 = n;
32   }
33   return s;
34 }
```

図 2　N 女王問題の全解探索の明示的スタックによる実装

Tascell は，各探索ステップに対応する関数フレー

ム内にクロージャを生成することで一時的バックトラックを実現しており，並列プログラムは再帰呼び出しベースで記述する必要がある．そのため，図 2 のようなスタック版のプログラムを Tascell で並列化するには，まず再帰版へ全面的に書き直す必要があり，プログラマにとって大きな障壁となる．

本研究では，明示的スタックに探索情報を保持して再帰呼び出しを行わないプログラミングスタイルを採りつつ，Tascell の特徴である一時的バックトラックに基づく並列化を可能にする手法を提案する．提案手法では，プログラマが探索状態を表すフレームを明示的スタック上に積むように記述し，タスク要求時にはスタックを走査して分割可能な地点を直接特定することで，従来の Tascell のように各探索ステップでクロージャを生成することなく，一時的バックトラックに基づくタスク生成を実現する．

本研究の貢献は以下の通りである．

- 明示的スタックによる記述スタイルを保ったまま，Tascell の特徴である一時的バックトラックに基づく並列化を可能にする言語拡張を提案した．
- 提案した言語拡張について，状態復元を入れ子関数で行う方式と，それを排してインライン展開する方式の 2 種類のコンパイラ実装を行った．
- N 女王問題，ペントミノパズル，および巡回セールスマン問題を用いた性能評価により，明示的スタック方式でも再帰版と同等の並列性能を実現できることを示した．

以下，2 章でタスク並列言語 Tascell の機構を概説し，3 章で提案する言語拡張について述べる．4 章で 2 種類の実装を述べ，5 章で評価結果を示す．最後に，6 章で関連研究，7 章でまとめと今後の課題を述べる．

2 タスク並列言語 Tascell

Tascell は，一時的バックトラックに基づく動的負荷分散を特徴とするタスク並列プログラミング言語であり，C 言語の拡張として実装されている [5]．Tascell は S 式構文と C 言語の意味論を持つ基盤言語 SC-0 上に，変換規則として実装されており [4]，本稿の Tascell コード例も S 式で示す．C 言語ベースの構文と S 式は表記上の違いに過ぎず，本質的な差はない．

図 3 に，N 女王問題の全解探索を Tascell で記述したタスク構造体と探索関数 `nqueens` の本体を示す．`nqueens` は第 `k` 行にクイーンを置く列を選ぶ探索であり，第 `k` 行の候補列にわたる並列ループを `do-many` で表現する．ループ本体では，クイーンが置けるとき `dynamic-wind` によって盤面の状態を変更し（*redo* 処理）（23–27 行目），その状態のまま次の行 `k+1` を再帰的に探索し（28–29 行目），復帰時に状態を元に戻す（*undo* 処理）（30–35 行目）．タスク要求がない通常時は，この再帰によって単一のワーカが探索木を深さ優先で逐次に辿る．

Tascell ワーカは最初に複数のタスクを生成せず逐次実行を行い，他のワーカからタスク要求を受け取ったときのみ一時的にバックトラックを行い，タスクを生成する．図 4 に，このプログラムを並列実行したときのワーカ動作を表す．この図が示す通り，ワーカがタスク要求を受け取ると，以下のように動作する．

1. 最古の並列 for までバックトラックを行う．
2. 新しいタスクを生成する．
3. バックトラックから復帰する．
4. 自身のタスクを再開する．

ワーカがタスク要求を受け取ると，Tascell はコールスタックを根に向かって遡り，最も外側（最古）の `do-many` を探す（図 4 の 1）．このバックトラックの過程で各 `dynamic-wind` の `:after` が実行され，盤面の状態が分割対象の `do-many` の時点まで巻き戻される．そこで残りの反復範囲 `[i1, i2)` を分割し，`handles` 節の `:put` で切り出したタスクの入力を設定して他ワーカへ渡す（同 2）．その後，`:before` を再適用しながら元の探索地点まで復帰し（同 3），自身の探索を再開する（同 4）．分割したタスクの結果は，そのタスクの完了時に `:get` で集約される．このように，最も古いタスク分割点に遡ることにより，粒度の大きいタスクを生成することができる．この一時的バックトラックは，Tascell コンパイラによる C 言語への変換によって実現されている [5]．C は標準では *stack walk*，すなわち現在のフレームより外側（下位）のフレームに置かれた変数へのアクセス手段を提供しない．そこで Tascell コンパイラは，各探索ステップに対応する関数フレームごとに GCC の**入れ子関数**（ネ

```
;; the task structure for nqueens
(def (task nqueens)
  (def r int)               ;; output: number of solutions
  (def n int)               ;; problem size
  (def k int)               ;; row to search
  (def i1 int)              ;; range: [i1, i2)
  (def i2 int)
  (def a (array int MAXN))  ;; used columns
  (def lb (array int MAXN)) ;; diagonal usage 1
  (def rb (array int MAXN))) ;; diagonal usage 2

;; nqueens: place a queen on row k (columns a[ix..iy))
(def (nqueens n k ix iy tsk)
    (wfn int int int int int (ptr (struct nqueens)))
  (def s int 0)
  (do-many for i from ix to iy
    (def ai int (aref tsk->a i))
    ;; can we place a queen at (row k, column ai)?
    (if (not (or (aref tsk->lb (+ n -1 (- ai) k))
                 (aref tsk->rb (+ ai k))))
      (if (== k (- n 1)) (inc s)  ;; a solution
        (dynamic-wind
          (:before        ;; place a queen (update board)
            (= (aref tsk->lb (+ n -1 (- ai) k)) 1)
            (= (aref tsk->rb (+ ai k)) 1)
            (= (aref tsk->a i) (aref tsk->a k))
            (= (aref tsk->a k) ai))
          (:body          ;; search the next row recursively
            (+= s (nqueens n (+ k 1) (+ k 1) n tsk)))
          (:after         ;; remove the queen (backtrack)
            (= (aref tsk->lb (+ n -1 (- ai) k)) 0)
            (= (aref tsk->rb (+ ai k)) 0)
            (= ai (aref tsk->a k))
            (= (aref tsk->a k) (aref tsk->a i))
            (= (aref tsk->a i) ai)))))
    (handles nqueens
      (:put from i1 to i2     ;; hand off the range [i1, i2)
        (= this (mref tsk)) (= this.k k)
        (= this.i1 i1) (= this.i2 i2))
      (:get (+= s this.r))))    ;; collect a partial result
  (return s))
```

図 3　N 女王問題の全解探索の Tascell プログラム

ストした関数）を生成し，1 つ外側のフレームへの関数ポインタを引数として渡してこれらを連鎖させることで，stack walk を代替する．たとえば，あるフレームが持つ局所変数を外側から参照する処理は，図 5 のように，各フレームが自身の局所変数を返すか，あるいは 1 つ外側のフレームへ処理を委譲するかを分岐する入れ子関数として実装される．

タスク要求を検知したワーカは，最も内側のフレームからこの連鎖を順に呼び出しながら遡ることで，分割可能な最も外側の `do-many` まで一時的にバックトラックできる．

3　提案する言語拡張

本章では，本研究で提案する言語拡張について述べる．以下，3.1 節で本言語拡張の設計方針を述べ，3.2 節で具体的な言語仕様を示し，3.3 節で N 女王問題への適用例を示す．

3.1　設計方針

本言語拡張では，深さ優先探索における残りの仕事を，C のコールスタックではなく，Tascell が言語機能として提供する専用の明示的スタックで表現する．タスク要求時には，コールスタックを遡る代わりに**明示的スタックを走査**することで，従来の Tascell がコールスタックを遡って特定するのと同じく，最も外側の分割可能なループを特定する．そこで必要なフレームの書き換えを行い，タスクを要求したワーカのためのタスクオブジェクトを生成する（このタスクオブジェクトの生成処理自体は従来の Tascell と同様である）．これにより，各探索ステップでクロージャを生成することなく，一時的バックトラックに基づくタスク生成を実現する．

明示的スタックには，*P-frame* と *D-frame* の 2 種類のフレームを push する．P-frame は `do-many` の並列ループ 1 段に対応するフレームであり，残りのループ範囲を現在の反復位置 `i0` と終端 `i1` の 2 つの整数として保持し，分割可能な地点の対象となる．*D-frame* は，再帰呼び出し実装であれば関数フレームが暗黙に保持していた，1 回の状態変更を元に戻すためのデータ（undo 情報）を明示的に保持するフレームである．後述の `tcell-push` による D-frame の push 時に，フック処理を定義できる．このデータの内容（フィールド）とフック処理はいずれもアプリケーションに依存し，`tcell-push` 呼び出し時にユーザが宣言する．これにより，アプリケーションに依存する状態変更・復元処理，すなわち一時的バックトラックとそこからの復帰時の undo/redo 処理を，フ

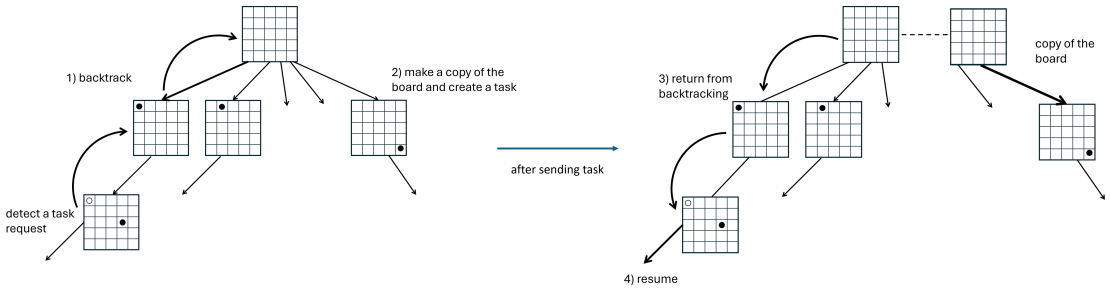

図 4　Tascell の動作

```
1  int f(int (*outer)(int), int n) {
2    int pc = 0;
3    int probe(int k) {        /* nested function (closure) */
4      if (k == 0) return pc; /* this frame's local variable */
5      return outer(k - 1);   /* delegate to the outer frame */
6    }
7    ...
8    s1 = f(probe, n - 1);     /* pass the closure down */
9    ...
10 }
```

図 5　入れ子関数による stack walk の実現

レームワーク側の仕組みとして実現する．この明示的スタック上で計算を実行中のワーカがワークスティール要求を受けると，以下のように動作する．

1. スタックを走査し，もっとも底に近い分割可能な P-frame を特定する．
2. スタックの先頭から，特定した P-frame までの間にある D-frame について，フックとして定義されている *undo* 処理を，先頭に近いものから順に実行する．
3. 特定した P-frame のループの一部に相当するタスクを生成し，タスクを要求したワーカに送信する．自身は残りの反復のみを実行するよう，P-frame の内容を書き換える．
4. 先ほど *undo* 処理を実行した D-frame について，*redo* 処理を *undo* 処理とは逆順に実行する．

これにより，従来の Tascell の `dynamic-wind` に相当する undo/redo 処理やタスク生成を実現する．以下では，そのために導入する言語仕様を述べる．

3.2　言語仕様

本節では，本言語拡張で追加するコンストラクトについて説明する．

3.2.1　do-stack 文

`do-stack` 文の構文は以下の通りである．

(`do-stack` *stk*
  block-item$^*_{body}$
  (`handles` *task-id*
    (`:put from` *put-from* `to` *put-to* *block-item*$^*_{put}$)
    (`:get` *block-item*$^*_{get}$)))

stk は初期化済みのスタックであり，このスタックが空でない間 *block-item*$^*_{body}$ を繰り返し実行する．各反復の開始時に他ワーカからのタスク要求を検知し，要求があれば 3.1 節で述べた一時的バックトラック，タスク生成，一時的バックトラックからの復帰の一連の処理を *stk* 上で実行する．

`handles` 節は切り出すタスク *task-id* の受け渡しを記述するものであり，従来の Tascell にも相当するものがある．*block-item*$^*_{put}$ はタスク生成時に実行され，分割された範囲 [*put-from*，*put-to*) を実行するタスクのオブジェクトを初期化する．*block-item*$^*_{get}$ は *stk* のループが完了した時点で実行され，ループ実行中に生成・送信したタスクの実行結果を集約する．*block-item*$^*_{put}$ と *block-item*$^*_{get}$ の中では，キーワード `this` を用いてタスクオブジェクトを参照できる．*block-item*$^*_{put}$ では `this` のフィールドに値を代入してタスクを初期化し，*block-item*$^*_{get}$ では `this` のフィールドを参照してタスクの結果を得る．

3.2.2　tcell-push 文

`tcell-push` 文の構文は以下の通りである．

```
(tcell-push stk (id*)
  (:before block-item*before)
  (:after  block-item*after))
```

`tcell-push` 文は *stk* に D-frame を push する．再帰版 Tascell における関数の再帰呼び出しと `dynamic-wind` に相当する．各 *id* は，この `tcell-push` 文を囲むスコープに存在する局所変数を指し，push 時点でのその値が D-frame へ保存される．D-frame に保存されるフィールドの型は，参照した局所変数の型と同一である．$block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$ はいずれも，実行の直前に D-frame に保存された値を対応する局所変数へ書き戻す．$block\text{-}item^*_{before}$ は，この D-frame の push 時（すなわちこの `tcell-push` 文の実行時）と一時的バックトラックからの復帰時に実行される．$block\text{-}item^*_{after}$ は，一時的バックトラック時と，後述の `tcell-proceed` によるこの D-frame の pop 時に実行される．

3.2.3 tcell-push-loop 文

`tcell-push-loop` 文の構文は以下の通りである．

```
(tcell-push-loop stk from to)
```

`tcell-push-loop` 文は，新たな P-frame を push する．従来の Tascell の `do-many` に対応する．*from* と *to* は整数であり，反復範囲を表す半開区間 [*from*, *to*) の下端と上端である．ここで積まれた P-frame がタスク分割の候補となる．

3.2.4 tcell-proceed 文

`tcell-proceed` 文の構文は以下の通りである．

```
(tcell-proceed stk)
```

`tcell-proceed` は，次の手続きを実行する．

1. *stk* のトップが D-frame である間，それを pop して $block\text{-}item^*_{after}$ を実行する．
2. トップの P-frame について，その半開区間の下端を 1 増やす．
3. 下端が上端より小さければ，ここで終了する．
4. 下端と上端が一致していれば（区間を使い切っていれば），その P-frame 自身を pop し，1. に戻る．

3.2.5 tcell-loop-i0 式

`tcell-loop-i0` 式の構文は以下の通りである．

```
(tcell-loop-i0)
```

この式は現在もっともトップに近い P-frame の半開区間の下端（ループ変数の値に相当）を返す．P-frame のフィールドを直接読むことは許されず，必ずこの構文を用いる．

3.3 N 女王問題への適用例

図 6 に，ここまで述べた言語仕様を用いて N 女王問題の全解探索を提案手法で記述した例を示す．

タスク構造体 `nqueens`（2–10 行目）は，出力 `r`，問題サイズ `n`，探索対象の行 `k`，反復範囲の下端・上端 `i0`，`i1`，および盤面の状態 `a`・`lb`・`rb` を保持する．探索関数 `nqueens`（13 行目以降）がワーカ関数であり，その本体は `stk` 上の `do-stack` 文（18 行目以降）である．*stk* は空の状態で作成したのち，関数の引数 `i0`，`i1` を反復範囲とする P-frame を `tcell-push-loop` 文で 1 つだけ push して初期化する（16–17 行目）．各反復では，`tcell-loop-i0` 式で現在の反復位置を読み（19 行目），そこにクイーンを配置できるか判定する（23–24 行目）．最終行であれば解を 1 つ数える（26 行目）．そうでなければ，`tcell-push` 文で D-frame を push する（29 行目）．この D-frame は，巻き戻しに必要な対角線の位置 `lb-index`・`rb-index` と，配置した列・行 `i`・`k` を保持し（30 行目），盤面の変更を $block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$ として記述する（31–39 行目）．`tcell-push-loop` 文で新たな P-frame を push して次の行へ進む（41 行目）．配置できない場合や，最終行で解を数えた後は，`tcell-proceed` 文でループを進める（43 行目）．`tcell-proceed` 文が D-frame を pop する際には，`tcell-push` 文で定義した $block\text{-}item^*_{after}$，すなわちクイーンを取り除く処理が実行される（36–39 行目）．`handles` 節（44–48 行目）は，スティール成立時のタスクオブジェクトの初期化（45–47 行目）と，結果が返ったときの集約（48 行目）を記述する．

ループの実行中に他ワーカからのタスク要求を検知すると，3.1 節で述べた手順に従い，*stk* を走査してもっとも底に近い分割可能な P-frame を特定し，そこまでの D-frame について $block\text{-}item^*_{after}$ を実行して

```
1  ;; the task structure for nqueens
2  (def (task nqueens)
3    (def r int)                ;; output: number of solutions
4    (def n int)                ;; problem size
5    (def k int)                 ;; row to search
6    (def i0 int)                  ;; range: [i0, i1)
7    (def i1 int)
8    (def a (array int MAXN))     ;; used columns
9    (def lb (array int MAXN))    ;; diagonal usage 1
10   (def rb (array int MAXN)))   ;; diagonal usage 2
11
12 ;; nqueens: explore rows [k, n) using an explicit stack
13 (def (nqueens n k i0 i1 tsk)
14     (wfn int int int int int (ptr (struct nqueens)))
15   (def s int 0)
16   (def stk tcell-stack (tcell-stack-init))
17   (tcell-push-loop stk i0 i1)
18   (do-stack stk
19     (def i int (tcell-loop-i0))
20     (def ai int (aref tsk->a i))
21     (def lb-index int (+ n -1 (- ai) k))
22     (def rb-index int (+ ai k))
23     (if (not (or (aref tsk->lb lb-index)
24                  (aref tsk->rb rb-index)))
25       (if (== k (- n 1))
26           (inc s)        ;; a solution
27           (begin
28             ;; dynamic-wind = D-frame
29             (tcell-push stk
30               (lb-index rb-index i k)
31               (:before        ;; place (redo)
32                 (= (aref tsk->lb lb-index) 1)
33                 (= (aref tsk->rb rb-index) 1)
34                 (swap (aref tsk->a i) (aref tsk->a k))
35                 (inc k))
36               (:after        ;; remove (undo)
37                 (= (aref tsk->lb lb-index) 0)
38                 (= (aref tsk->rb rb-index) 0)
39                 (swap (aref tsk->a i) (aref tsk->a k))))
40             ;; deepen one level (new P-frame)
41             (tcell-push-loop stk k n)
42             (continue))))
43     (tcell-proceed stk)      ;; advance / pop
44     (handles nqueens
45       (:put from lo to hi   ;; build stolen task
46         (= this (mref tsk)) (= this.k k)
47         (= this.i0 lo) (= this.i1 hi))
48       (:get (+= s this.r))))      ;; collect result
49   (return s))
```

図 6　提案する言語拡張を用いた N 女王問題の全解探索の記述

盤面を巻き戻す(36–39 行目)．その P-frame の反復範囲の一部を $block\text{-}item^*_{put}$ によってタスクオブジェクトへ切り出し(45–47 行目)，その後 $block\text{-}item^*_{before}$ を実行しながら盤面を元の状態へ復帰させ(31–35 行目)，自身の探索を再開する．

4　実装

本言語拡張は，C 言語への変換として実装されている従来の Tascell コンパイラを拡張する形で実装した．`do-stack` 文は，基本的には `while` ループへ変換できる．ループ先頭でタスク要求の有無を検査する処理を挿入すればよい．`tcell-push-loop` 文や `tcell-proceed` 文，`tcell-loop-i0` 式など，他の追加構文の多くの部分も同様に素朴な変換で実装できる．

問題となるのは，一時的バックトラックおよびそこからの復帰の途上，および `tcell-proceed` の実行中に，`tcell-push` 文に記述された $block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$ をどう実行するか，また，タスク生成時に `handles` 節内の $block\text{-}item^*_{put}$ をどう実行するかである．これらの実行には周囲の局所変数へのアクセスが必要であり，単純にトップレベル関数として切り出すことができない．本研究では，従来の Tascell と同様に入れ子関数を用いる方式（4.1 節）と，コードを複製することで実現する方式（4.2 節）の 2 種類の実装を行った．さらに，本拡張言語による典型的な実行では P-frame の値の書き換えが頻繁に発生するため，これに伴うメモリアクセスを削減する最適化も行った（4.3 節）．

4.1　入れ子関数による実装

第一の実装は，従来の Tascell と同様，入れ子関数を用いて $block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$，$block\text{-}item^*_{put}$ を呼び出す方式である．各 `tcell-push` は，D-frame を push する箇所でその場に $block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$ それぞれに対応する入れ子関数を 1 つずつ定義し，D-frame にそれらの関数ポインタを保持させる．入れ子関数は周囲のスコープを暗黙に捕捉するため，$block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$ が参照する局所変数（id^* で列挙したもの）にそのままアクセスできる．以下に，`tcell-push` 文の変換の要点を疑

似コードで示す.

```
{
  void before (struct d_frame *self) {
    lb_index = self->lb_index; rb_index = self->rb_index;
    i = self->i; k = self->k;
    block-item*_before
  }
  void after (struct d_frame *self) {
    lb_index = self->lb_index; rb_index = self->rb_index;
    i = self->i; k = self->k;
    block-item*_after
  }
  struct d_frame *f = push_d_frame(stk);
  f->before = before; f->after = after;
  f->lb_index = lb_index; f->rb_index = rb_index;
  f->i = i; f->k = k;
  before(f);   block-item*_before をその場で実行
}
```

一時的バックトラックおよびそこからの復帰では，スタック上の D-frame を順に辿りながら，各 D-frame が保持する関数ポインタを介して $block\text{-}item^*_{after}$，$block\text{-}item^*_{before}$ を呼び出す．`tcell-proceed` が D-frame を pop する際も，同じ関数ポインタ経由で $block\text{-}item^*_{after}$ を呼び出す．`handles` 節の $block\text{-}item^*_{put}$ も同じ理由で入れ子関数として定義し，スティール成立時にその場で呼び出す．

この方式は，Tascell が従来用いてきたクロージャ生成の枠組みを踏襲し，外側のバックトラック連鎖にも参加できる一方，GCC 拡張に依存し，入れ子関数の生成・維持に伴うコストが生じる．

4.2 インライン展開による実装

第二の実装は，入れ子関数を排し，$block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$，$block\text{-}item^*_{put}$ の本体を必要な箇所へ複製することで実現する方式である．関数内に出現する `tcell-push` を出現順に採番し，各 D-frame にはこの番号のみを保持させる．$block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$ の本体は入れ子関数として切り出さず，この番号による `switch` 分岐の 1 ケースとして，元の関数の中に複製して直接記述する．複製先は，push 時点にその場でインライン実行される箇所と，undo 走査・redo 走査それぞれの `switch` 分岐の計 3 箇所である．これにより，$block\text{-}item^*_{before}$，$block\text{-}item^*_{after}$ は周囲の局所変数を（入れ子関数を介さず）直接参照できる．以下に，`tcell-push` 文の変換の要点を疑似コードで示す.

```
/* push: tcell−push #0 */
push_d_frame(stk, /*frame_no=*/0, lb_index, rb_index, i, k
    );
block-item*_before                  /* その場でインライン実行 */

...

/* undo走査（トップに近いものから）*/
switch (frame_no) { case 0: block-item*_after  break; ... }

/* redo走査（1 つ下から順に）*/
switch (frame_no) { case 0: block-item*_before break; ... }
```

D-frame は関数ポインタを持つ必要がなく，GCC 拡張に依存しない標準的な C 言語の範囲で記述できる．ただし，$block\text{-}item^*_{before}$・$block\text{-}item^*_{after}$ の本体は複数箇所に複製されるため，そこから参照できる局所変数は，id^* で列挙したものと，`tcell-push` 文を囲む関数の引数のようにそれより外側で元から宣言済みの変数に限られる．

本方式が適用できるのは，`tcell-push` 等が `do-stack` 文と同一の関数内に記述されているなど，一定の条件を満たす場合に限られる．条件を満たさない場合，処理系は 4.1 節で述べた入れ子関数による実装にフォールバックする．この判定も処理系がプログラムを走査して自動的に行うため，ユーザはいずれの場合も同一の記述でよい．

4.3 ループ変数のキャッシュ

本拡張言語による典型的な実行では，`tcell-loop-i0` 式で読むループ変数 `i0`（および終端 `i1`）が毎反復更新される．これをトップ P-frame 上のフィールドとして毎回読み書きすると，反復のたびにメモリアクセスが発生する．そこで，`i0`，`i1` をトップ P-frame へ都度書き戻すのではなくローカル変数へキャッシュしてから探索本体を実行する最適化を行う．フィールドアクセスを局所変数へのアクセスに置き換えることで，レジスタアクセスとなることを期待するものである．

```
/* キャッシュなし:  毎反復 P-frame を読み書きする */
while (...) {
  p_frame *top = top_p_frame(stk);
```

表 1　評価環境

	Camphor 3 (システム A)
CPU	Intel Xeon CPU Max 9480 1.9GHz 56-core×2
メモリ	HBM2e Memory 128GB (3.2TB/s)
OS	Red Hat Linux 8.9
コンパイラ	gcc 8.5.0 (-O3 最適化，入れ子関数の実装は GCC の trampoline を利用)
ワーカ	pthread_create で生成 (PTHREAD_SCOPE_SYSTEM スケジューリング)

```
4   if (top->i0 >= top->i1) { ... }
5   int i = top->i0++;
6   ...
7 }
8
9 /* キャッシュあり:  トップ P-frame が変わるまでレジスタに保持する
      */
10 int i0 = top->i0, i1 = top->i1;
11 while (...) {
12   if (i0 >= i1) {
13     top->i0 = i0;                          /*
         pop する前に書き戻す */
14     ...
15     i0 = new_top->i0; i1 = new_top->i1;  /* 新しいトップから読
         み直す */
16     continue;
17   }
18   int i = i0++;
19   ...
20 }
```

5　性能評価

5.1　評価方法

4 章で示した各実装を評価するために，京都大学学術情報メディアセンターのスーパーコンピュータ Camphor 3 (システム A) の 1 ノードを用いて性能評価を行った．評価に用いた環境を表 1 にまとめる．本評価では，次の 3 つのベンチマークを用い，従来の再帰呼び出しベースの実装を行ったプログラム，および逐次 C プログラムと性能比較を行った．

Nqueens(16)：16×16 の盤面に 16 個のクイーンを，同じ行・列・対角線上に 2 つ以上置かれないように配置する問題の全解探索．行ごとに置ける列を順に試すバックトラック法で探索する．

Pentomino(14)：5 マスからなる複数種類のピース（ペントミノ）を，盤面に重なりなく敷き詰める問題の全解探索．標準的なペントミノパズルは 12 種類のピースを用いるが，本評価では文献 [5] の評価にならい，2 種類のピースと盤面を拡張した 14 ピース版を用いた．盤面のマスを左上から順に埋めていき，そのマスを含むピースを置けるかどうかを判定するバックトラック法で探索する．

TSP(6x6)：巡回セールスマン問題．辺コストの異なる 2 次元トーラス 6×6 格子上で，全頂点をちょうど 1 度ずつ訪れて出発点に戻る経路のうち，総コストが最小のものを求める．単純な深さ優先探索による総当たり法で探索する．

各実装について 1，2，4，8，16，32，64，112 ワーカで実行時間を測定した．各データ点は 3 回測定の中央値である．測定のばらつき（各点 3 回の最大値と最小値の差を最小値で割った値）は，Nqueens で平均 0.88%，Pentomino で平均 1.07%と小さい．TSP は実行時間が 5 時間に及ぶこともあり平均 2.85%（最大 13.39%）とやや大きいため，最小値ではなく中央値を採ることで外れ値の混入を避けている．

表 2　比較対象の実装

名称	内容
逐次 C	逐次 C プログラム（並列化なし）
再帰	再帰版（従来 Tascell）
do-stack	提案拡張，入れ子関数 do_stack_bk による実装
do-stack-nc	同上，ループ変数キャッシュ無効
nonest	提案拡張，インライン展開による実装
nonest-nc	同上，ループ変数キャッシュ無効

比較対象は表 2 に示す並列 5 実装と，基準となる逐次 C 版である．提案手法の 2 つの実装それぞれについて，4 章で述べたループ変数のキャッシュを無効化した版（-nc）を用意し，キャッシュがもたらす効果と，入れ子関数の有無がもたらす効果とを分離して評価できるようにした．逐次 C 版はタスク分割機構を一切含まない C プログラムであり，1 ワーカでのみ測定する．

5.2　並列性能

各実装の 1 ワーカ，112 ワーカの実行時間および再帰版に対する相対時間を表 3 に，ワーカ数を変化させたときの再帰版 1 ワーカに対する性能向上を図 7–9 に示す．

表 3　各実装の実行時間と，同ワーカ数での再帰版に対する相対時間

	Nqueens(16)		Pentomino(14)		TSP(6x6)	
	1W	112W	1W	112W	1W	112W
再帰	84.9 (1.00)	0.914 (1.00)	75.6 (1.00)	0.876 (1.00)	446.2 (1.00)	4.792 (1.00)
do-stack	105.3 (1.24)	1.175 (1.29)	106.1 (1.40)	1.145 (1.31)	588.6 (1.32)	6.448 (1.35)
do-stack-nc	107.8 (1.27)	1.181 (1.29)	111.5 (1.47)	1.197 (1.37)	606.2 (1.36)	6.461 (1.35)
nonest	89.5 (1.05)	1.021 (1.12)	96.6 (1.28)	1.018 (1.16)	540.5 (1.21)	5.819 (1.21)
nonest-nc	114.1 (1.34)	1.201 (1.31)	108.2 (1.43)	1.132 (1.29)	596.3 (1.34)	6.357 (1.33)

各欄は実行時間 [s]，括弧内は同ワーカ数における再帰版に対する相対時間．

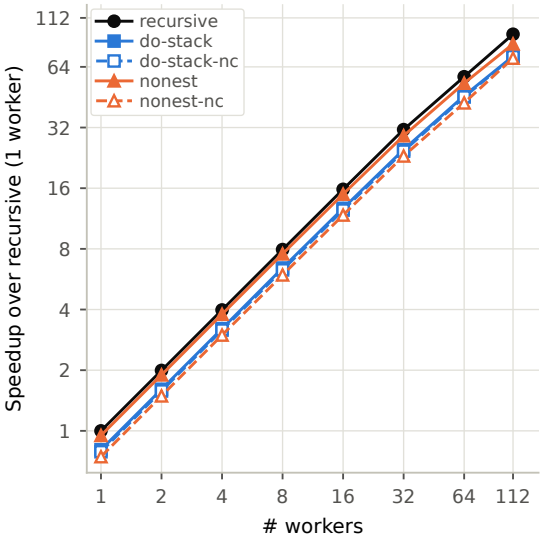

図 7　Nqueens(16) の並列性能

いずれのベンチマークも，5 実装は全ワーカ数域でほぼ同等のスケーラビリティを示す．112 ワーカでの再帰版 1 ワーカに対するスピードアップは，Nqueens で 70.7〜92.9 倍，Pentomino で 63.2〜86.3 倍，TSP で 69.1〜93.1 倍である．すなわち明示的スタック化は逐次性能を悪化させるものの，スケーラビリティを損なわない．

唯一の例外は Pentomino であり，図 8 では明示的スタック版の曲線が高並列域で再帰版へわずかに近づく．実際，再帰版に対する相対時間は Nqueens と TSP ではワーカ数によらずほぼ一定であるのに対し，Pentomino では `do-stack` が 1 ワーカの 1.40 倍から 112 ワーカの 1.31 倍へと縮む（表 3）．ただし逐次

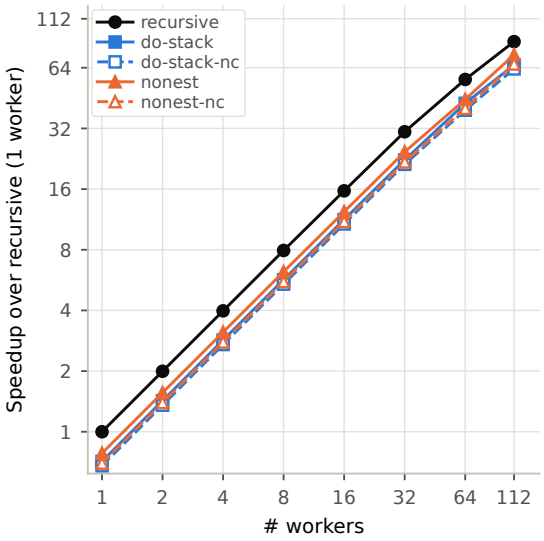

図 8　Pentomino(14) の並列性能

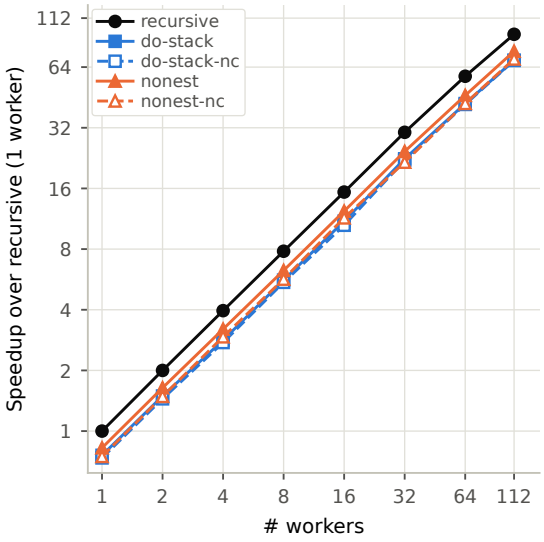

図 9　TSP(6x6) の並列性能

オーバーヘッドが上回るため，112 ワーカでも絶対時間の逆転には至らない．

その結果，絶対時間では全ワーカ数で再帰版が最速であり，明示的スタック版が逆転する点はない．ただし逐次オーバーヘッドが提案手法のうち最小の `nonest` は，全ベンチマークの全ワーカ数で再帰版に次いで速く，絶対性能でも再帰版とおおむね同等の性能を実現できた．以上より，5 実装の性能差の要因は

表 4　逐次 C 版との比較

	Nqueens(16)		Pentomino(14)		TSP(6x6)	
	1W	112W	1W	112W	1W	112W
再帰	1.19	78.1	1.05	82.4	1.27	73.5
do-stack	1.48	60.7	1.47	63.0	1.67	54.6
do-stack-nc	1.51	60.4	1.54	60.3	1.72	54.5
nonest	1.25	69.8	1.34	70.9	1.54	60.5
nonest-nc	1.60	59.4	1.50	63.8	1.69	55.4

1W 欄は相対実行時間（逐次 C 版=1），112W 欄はスピードアップ.

並列スケーリングではなく逐次性能に集約される．そこで次節では，タスク並列機構を一切含まない逐次 C プログラムを基準として，その逐次オーバーヘッドの内訳を見る．

5.3　逐次性能との比較

タスク分割機構を一切含まない逐次 C プログラムを基準とした比較を表 4 に示す．1 ワーカの列は各実装が逐次実行に課すオーバーヘッドを，112 ワーカの列は 1 コアの逐次 C に対して 112 コアで何倍を達成できたかを表す．逐次 C 版を 1.00 としたとき，1 ワーカ実行時間は再帰版が 1.19 倍（Nqueens），1.05 倍（Pentomino），1.27 倍（TSP），`nonest` が 1.25 倍，1.34 倍，1.54 倍，`do-stack` が 1.48 倍，1.47 倍，1.67 倍である．

ここから 2 段のオーバーヘッドが分離できる．第 1 段は Tascell 化のコスト，すなわち逐次 C プログラムをタスク分割可能な形に書き直すことによる劣化で，4.8%（Pentomino）から 26.7%（TSP）である．第 2 段は明示的スタック化のコスト，すなわち呼び出しスタックを明示的なデータ構造へ置き換えることによる劣化で，再帰版に対し `nonest` で 5.4%（Nqueens），27.8%（Pentomino），21.1%（TSP），`do-stack` で 24.1%，40.3%，31.9%である．第 2 段が第 1 段を上回るベンチマークが多いが，最良の組み合わせでも逐次 C の 1.05〜1.27 倍，提案手法のうち速い `nonest` でも 1.25〜1.54 倍に収まっており，明示的スタック方式に動的負荷分散を導入する代償はこの範囲にとどまる．

112 ワーカの列に示すとおり，逐次 C 版に対するスピードアップは再帰版が 73.5〜82.4 倍，`nonest` が 60.5〜70.9 倍，`do-stack` が 54.6〜63.0 倍である．各実装が自身の 1 ワーカに対して達成する 77〜86%という効率と，逐次 C に対する 54.5〜82.4 倍というこの値との差が，Tascell 化と明示的スタック化の逐次オーバーヘッドそのものである．

表 5　ループ変数キャッシュ無効化による劣化率

	Nqueens(16)	Pentomino(14)	TSP(6x6)
do-stack	1.024	1.051	1.030
nonest	1.275	1.120	1.103

1 ワーカ．-nc 版/キャッシュ有効版の実行時間比.

実装間では，全ベンチマークで `nonest` が `do-stack` より速く，その差は 15.1%（Nqueens），8.9%（Pentomino），8.2%（TSP）である．入れ子関数が生成するトランポリンと，`before`/`after` の間接呼び出しとを除去した効果であり，差は Nqueens で最大となる．Nqueens は 1 反復あたりの仕事が小さく，フレーム操作の占める比率が高いためと考えられる．この差の内訳については 5.4 節で改めて述べる．なお，逐次オーバーヘッドが Pentomino で Nqueens より一貫して大きいのは，Pentomino が多層ループを用いるため，各深さで積むフレーム数が Nqueens（単層ループ）より多く，その固定コストがより重いためである．

5.4　ループ変数キャッシュの効果

キャッシュを無効化した`-nc` 版との比較から，ループ変数のキャッシュがもたらす効果と入れ子関数の有無がもたらす効果とを分離できる．表 5 に，1 ワーカにおけるキャッシュ無効化による実行時間の劣化率を示す．

キャッシュの効果は入れ子関数の有無によって大きく異なる．`do-stack` では劣化率が 2〜5%にとどまるのに対し，`nonest` では 10〜28%に達する．これは，`do-stack` ではキャッシュ変数が入れ子関数 `do_stack_bk` から参照されるため，GCC が static chain 経由のアクセスに備えて必ずスタックフレーム上に配置し，キャッシュしてもメモリアクセスのままとなるからである．入れ子関数を持たない `nonest` では，キャッシュ変数がレジスタに割り当てられる．

このことから，逐次性能における `nonest` の優位の実体はキャッシュのレジスタ化であると言える．1 ワーカにおける `nonest`/`do-stack` の実行時間比は，両者

ともキャッシュ有効のとき 0.850（Nqueens），0.911（Pentomino），0.918（TSP）と `nonest` が明確に速いのに対し，両者ともキャッシュ無効にすると 1.058，0.971，0.984 となり，優位はほぼ消失する（Nqueens では逆転する）．さらに，キャッシュを無効にした `nonest-nc` は，入れ子関数を持つ `do-stack`（キャッシュ有効）よりも遅い（Nqueens で 114.1 秒対 105.3 秒，Pentomino で 108.2 秒対 106.1 秒，TSP で 596.3 秒対 588.6 秒）．したがって，`nonest` の利得は入れ子関数（トランポリン）の除去そのものではなく，それによりキャッシュがレジスタに載るようになることが主因であり，入れ子関数の維持コスト単独の寄与は上記の比（0.971〜1.058）から見て数%程度と考えられる．このことは,「入れ子関数の除去」と「ループ変数のキャッシュ」が対で意味を持つことを示している．実際，`nonest-nc` は Nqueens と Pentomino の 2 ベンチマークで 5 実装中もっとも遅く，`nonest` 化による利得をキャッシュ無効化が食い潰している．

6 関連研究

特定のアプリケーションに限れば，明示的スタックを並列に走査する手法は古くから作り込まれてきた．大規模共有メモリ機向け並列マークスイープごみ集め [2] は，各プロセッサが持つマークスタック（明示的スタック）をプロセッサ間で交換して動的に負荷分散する具体例であり，本研究が扱う問題設定が実在することを示している．上野らの軽量スレッド上の並行・並列ごみ集め [12] も，タスク並列基盤の上にワーカ間のワークスティーリングを構成している．

明示的スタックやワークリストを直接並列化する汎用の手法としては，Galois [6] [10] [8] が最も近い．Galois は不規則計算を operator formulation で表現し，ワークリストから取り出した複数の active node を投機的に並列実行して，neighborhood の重なりを実行時に検出し，衝突すればロールバックする．しかし，独自の計算モデルとライブラリの語彙への書き換えを避けられないうえ，本研究が対象とするバックトラック探索では，盤面のような共有状態を undo/redo しながら深さ優先で辿るため，並行に進む枝どうしの neighborhood が重なりやすく，投機による並列化は本質的に不利である．提案手法は投機を用いず，分割地点までの巻き戻しと再適用によって切り出すタスクの文脈を確定させるため，衝突検出もロールバックも要さない．

7 まとめと今後の課題

本研究では，明示的スタックによる木探索の記述スタイルを保ったまま，すなわち再帰呼び出しベースへ書き換えることなく，Tascell の一時的バックトラックに基づく動的負荷分散を利用可能にする言語拡張を提案した．状態復元を入れ子関数で行う `do-stack` 版と，それをインライン展開する `nonest` 版の 2 種類を実装し，Nqueens，Pentomino，TSP の 3 つのベンチマークを用いて，逐次 C プログラムを基準に評価した．その結果，明示的スタック方式でも再帰版と同等の並列性能が得られること，具体的には全ベンチマークで 5 実装がほぼ同等のスケーラビリティを示すことを確認した．唯一 Pentomino では，高並列域で明示的スタック版の曲線が再帰版へわずかに近づく傾向が見られた．代償となる逐次オーバーヘッドは，逐次 C に対して再帰版が 1.05〜1.27 倍であるのに対し，提案手法のうち速い `nonest` 版で 1.25〜1.54 倍であり，全ベンチマークの全ワーカ数で再帰版に次いで速く，絶対性能でも再帰版とおおむね同等の性能を実現できた．また入れ子関数を排した `nonest` 版が，ループ変数キャッシュのレジスタ化により逐次実行性能で `do-stack` 版を 8〜15%上回ることを示した．

今後の課題として，まず，ごみ集めや SAT ソルバなど，既存の実用的な明示的スタックベースの木探索アルゴリズムに本言語拡張を適用し，その性能と記述性を評価することが挙げられる．性能評価は単一ノード上で行ったが，分散メモリ環境における評価も今後行いたい．

参 考 文 献

[1] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing, *ACM SIGPLAN Notices*, Vol. 40, No. 10(2005), pp. 519–538.

[2] Endo, T., Taura, K., and Yonezawa, A.: A Scalable Mark-Sweep Garbage Collector on Large-Scale

Shared-Memory Machines, *Proc. 1997 ACM/IEEE Conference on Supercomputing (SC'97)*, 1997.

[3] Frigo, M., Leiserson, C. E., and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, 1998, pp. 212–223.

[4] Hiraishi, T.: *Transformation-based Implementation of S-expression Based C Languages*, PhD Thesis, Kyoto University, 2008.

[5] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T.: Backtracking-based Load Balancing, *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*, Raleigh, NC, USA, 2009, pp. 55–64.

[6] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P.: Optimistic Parallelism Requires Abstractions, *Proc. 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 211–222.

[7] Mohr, E., Kranz, D. A., and Robert H. Halstead, J.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3(1991), pp. 264–280.

[8] Nguyen, D., Lenharth, A., and Pingali, K.: A Lightweight Infrastructure for Graph Analytics, *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 456–471.

[9] OpenMP Architecture Review Board: *OpenMP Application Program Interface Version 4.5*, 2015.

[10] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X.: The Tao of Parallelism in Algorithms, *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 12–25.

[11] Schardl, T. B. and Lee, I.-T. A.: OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code, *Proc. 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2023)*, 2023, pp. 189–203.

[12] Ueno, K. and Ohori, A.: Concurrent and Parallel Garbage Collection for Lightweight Threads on Multicore Processors, *Proc. 2022 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2022, pp. 29–42.