HOPE: 階層的計算省略に基づく耐障害性を備えた並列実行モデル

八杉 昌宏　村岡 大輔　平石 拓　馬谷 誠二　江本 健斗

我々は耐障害性を備えた並列実行モデルとして，HOPE モデルを提唱している．HOPE モデルは，既存の並列実行モデルが「複数ワーカで仕事を分担」（＋分割統治におけるワークスティール等）に基づくのとは全く逆に，完全冗長実行からの階層的計算省略に基づく．HOPE 言語は C 言語（命令型逐次言語）をベースとしており，HOPE プラグラマは，プログラム中の逐次実行順序が任意でよい部分とそれぞれの計算結果へのアクセス方法を指定すればよい．各ワーカはタスク生成や待ち合わせは行わず，分割統治の各階層を計画しておいた順序にて逐次実行する．計画は全ワーカで事前共有しておき，計算の一部を省略するのに必要な部分的計算結果を適切に交換する．これにより，複数ワーカ並列実行による速度向上と弱点を持たない耐障害性が，探索問題などの不規則な場合を含む広い範囲の応用プログラムで実現できる．

1　はじめに

大規模並列計算システム上で動作させる（不規則アプリケーションを含めた）高性能・高信頼アプリケーションでは，高生産性，スケーラビリティ，優れた負荷分散，耐障害性はいずれも重要である．

負荷分散のためには，ワークスティール・フレームワーク [10] [3] [12] [13] [8] [1] [9] [5] が提案されている．これらは，分割統治アルゴリズムに適しており，各ワーカは必要に応じて，呼び出し木／分割統治のルート付近で，比較的大きなタスクを盗むことで，トータルでのワークスティールのコストを削減している．

中でも遅延タスク生成（Lazy Task Creation; LTC）[10] は，多数の論理スレッドを生成（spawn）しておき，最古優先のワークスティール戦略をとる優れた実装方式といえる．LTC では新しく spawn された論理スレッドが先にスケジュールされ，古い論理スレッドの残りの計算（継続）は，他のワーカによりスティール可能となる．遊休状態となったワーカ（thief）は，スティール対象のワーカ（victim）を通常ランダムに選択する．MIT Cilk [3] もこの方式を用いている．

Hiraishi らは論理スレッドを用いないバックトラックに基づくワークスティール・フレームワークとして Tascell [5] を提案している．ワーカはタスク要求を受け取るまで基本的には逐次計算を行う．要求に応じて，一時的バックトラックで最古のタスク生成可能状態を復元して本物のタスクを生成する．Tascell は分散メモリ環境にも対応している．

しかし，ワークスティール・フレームワークには，チェックポイント方式などで耐障害性をサポートするのが難しいという問題がある．victim はスティールされるタスクを保存しておくことで，thief が停止したとしても自ら同じタスクを実行可能であるが，分割統治のルート付近では，その仕事量は非常に大きいう

---

* HOPE: A Fault-Tolerant Parallel Execution Model Based on Hierarchical Omission.

Masahiro Yasugi, 九州工業大学, Kyushu Institute of Technology.

Daisuke Muraoka, 九州工業大学 (現在，SmartNews, Inc.), Kyushu Institute of Technology (Presently with SmartNews, Inc.).

Tasuku Hiraishi, 京都大学, Kyoto University.

Seiji Umatani, 神奈川大学, Kanagawa University.

Kento Emoto, 九州工業大学, Kyushu Institute of Technology.

え，ルートを担当するワーカは耐障害性における弱点といえる．

我々は耐障害性を備えた並列実行モデルとして，HOPE モデル [15] を提唱している．HOPE モデルは，既存の並列実行モデルが「複数ワーカで仕事を分担」に基づくのとは全く逆に，完全冗長実行からの階層的計算省略に基づく．

本発表では，文献 [15][†1] の内容に基づき，HOPE モデル，HOPE 言語と実装の概要を紹介するとともに，いくつかの拡張と補足と展望を述べる．

HOPE モデルはまったく新しい発想に基づいており，そのパラダイムは，既存の並列実行モデルの専門家であればあるほど，想定を超えたものに感じられるようである．例えば，HOPE ワーカは：

- 仕事全体のうち，割り当てられた（固定された）部分のみを持つのでは**ない**.
- 遅いワーカを待た**ない**.
- 固定の通信パターンを待た**ない**.
- タスクやスレッドを生成（spawn）し**ない**.
- タスクやスレッドをスティールし**ない**.
- 結果を待た**ない**.
- 結果をブロードキャストし**ない**.
- （停止した部分を）修復・再実行し**ない**.
- 実行状態を保存したり復元したり巻き戻したりし**ない**.

HOPE 言語は C 言語（命令型逐次言語）をベースとした高生産性言語であり，HOPE プラグラマは，プログラム中の逐次実行順序が任意でよい部分とそれぞれの計算結果へのアクセス方法や扱い方を指定すればよい．

各ワーカはタスク生成や待ち合わせは行わず，分割統治の各階層を計画しておいた順序にて逐次実行する．計画は全ワーカで事前共有しておき，計算の一部を省略するのに必要な部分的計算結果を適切に交換する．これにより，複数ワーカ並列実行による速度向上と弱点を持たない耐障害性が，探索問題などの不規則な場合を含む広い範囲の応用プログラムで実現できる．

†1 https://doi.org/10.1145/3337821.3337899

```
int fib(int n) {          // int fib_i(int n) {
  if (n <= 2) return 1;   //   if (n <= 2) return 1_i;
  int r0, r1;             //   int r0, r1;
  r0 = fib(n - 1);        //   r0 = fib_{2i}(n - 1);
  r1 = fib(n - 2);        //   r1 = fib_{2i+1}(n - 2);
  return r0 + r1;         //   return r0 + r1;
}                         // }
```

図 1　C program for Fibonacci. The right-hand-side comment is shown to easily identify recursive subcomputations with i.

2　並列実行と耐障害性

耐障害性を実現する標準的な方法には冗長実行がある．冗長実行する複数プロセスのうち，少なくとも 1 プロセスが実行を継続できれば他のプロセスがすべて停止したとしても計算全体の結果は得られる．

不規則な分割統治アプリケーションとして，フィボナッチ数列の第 n 項を再帰的に（**遅く**）求める例を用いる．この例に実際的な意味はないが，並列言語の評価にはよく (例えば，文献 [3] [8] [10] [12] [13] [5] で) 用いられる．

図 1 に示した例では，右側コメント部分に再帰的に定まる，階層的計算の一部（subcomputation）を添え字 i で簡単に識別する方法を示した．図 2 には $\texttt{fib}_1\texttt{(51)}$ に含まれる階層的計算の木を示す．例えば，s_1 は s_2 と s_3 を含み，s_2 は s_4 と s_5 を含む．

最初にワーカ W_0 が s_1 を開始し，ワーカ W_2 が s_3 を W_0（s_1 を持つ）からスティールしたとする．同様に，W_1 は s_5 を W_0（$s_1 - s_3$ を持つ）からスティールし，W_3 は s_7 を W_2 (s_3 を持つ) からスティールしたとすると，4 ワーカとも仕事を持っている状況とできる．

ここで，W_3 による s_7 のスティール時に，W_2 は s_7 をある種の軽量なチェックポイントとして保存できたとする．これにより，W_3 が停止したとしても W_2 は保存した s_7 をリスタートできる．この方式は多少の耐障害性を持つが，W_2 が停止した場合は，分割統治の大きな部分 s_3 をリスタートする必要があるうえ，ルートを担当する W_0 は弱点（a single point of failure）であるため十分な障害耐性を持つとは言え

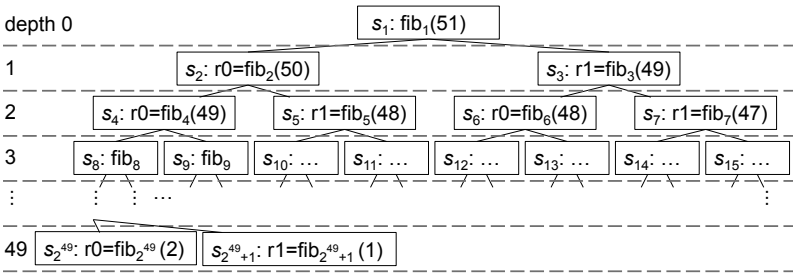

図 2　Recursive subcomputations in fib$_1$(51) in Fig. 1.

ない．

3　階層的計算省略に基づく並列実行モデル HOPE

図 1 と 2 の例を用いて，4 ワーカとも fib$_1$(51) 全体（s_1）を冗長実行する場合を考える．その際，s_2 と s_3 の実行順序は計算全体の結果には影響しない．このように，単一のプログラムを別の順序で実行するような場合を **SPMO** (single program, multiple order) 並列実行と呼ぶ．これにより，回復のための特別な処理をしなくても，1 ワーカでも生き残っている限り実行を続けられる耐障害性が得られる．

ここでは，W_0, W_1, W_2, W_3 は s_1 をそれぞれ s_2s_3, s_2s_3, s_3s_2, s_3s_2 の順序で実行するとする．例えば，W_0 と W_1 は深さ 1 で先に s_2 (i が偶数の s_i) を実行する．これを深さ 0 における **0-優先**（*0-first*）と呼ぶ．同様に，W_2 と W_3 は深さ 0 における **1-優先**（*1-first*）を用いる．

W_0, W_1, W_2, W_3 は s_1 を，それぞれ深さ 2 で，$s_4s_5s_6s_7$, $s_5s_4s_7s_6$, $s_6s_7s_4s_5$, $s_7s_6s_5s_4$ の順序で実行するとする．つまり，W_0 と W_2 は深さ 1 における **0-優先**，W_1 と W_3 は深さ 1 における **1-優先**とする．

深さ 3 以上の詳細を無視すると，上記の計画的順序により，図 3 のように部分的計算結果を交換することで階層的計算のいくつかの部分を省略し，並列実行による速度向上を得ることができる．どのワーカも同じ結果を得ることから，遅いワーカを待つ必要はなく，最速のワーカの結果を用いればよい．

深さ 3 を無視しない場合，W_0, W_1, W_2, W_3 は s_4 をそれぞれ s_8s_9 (0-first), s_8s_9 (0-first), s_9s_8 (1-first), s_9s_8 (1-first) と実行する以外の，より洗練された方式として，図 3 にもあるように，W_0, W_1, W_2, W_3 は s_4 をそれぞれ s_8s_9 (0-first), s_9s_8 (1-first), s_9s_8 (1-first), s_8s_9 (0-first) と実行することができる．さらに，階層的計算の各部の仕事量のばらつきを考えると，図 3 とは別パターンで適応的な通信と，階層的計算省略がなされることになる．

より一般的に階層的計算における計算の一部を特定して，部分的計算結果の交換を行うためには，**可変長アドレス**を用いることにする．例えば，図 4 の場合，“@”, “@1”, “@01”, “@12” といった可変長アドレスで階層的計算の一部を特定できる．可変長アドレスは Cilk Plus [7] における *pedigrees* に類似したものともいえる．

4　HOPE 言語

4.1　概要

HOPE 言語は，C 言語（命令型逐次言語）を HOPE ディレクティブで拡張した言語として設計している．HOPE プラグラマは，プログラム中の逐次実行順序が任意でよい部分とそれぞれの計算結果へのアクセス方法や扱い方を（階層的に）指定すればよい．

図 5 には，`do_two` ディレクティブで直後の 2 つの `omissible` 文が，任意の順序で実行可能であることを示している．

`omissible` ディレクティブは，直後の文の結果がワーカとしてすぐに利用可能な（locally available な）状況なら，その文の実行を省略してよいことを示す．また，ここで，文の実行は冪等（idempotent）でなくてはならない．つまり，複数回実行しても等価な結

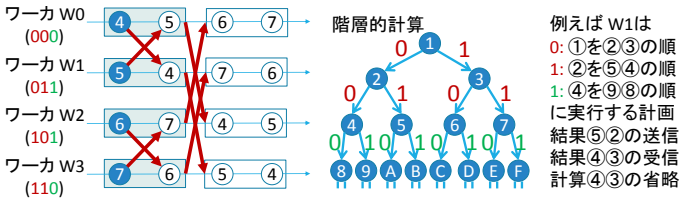

図 3　階層的計算省略に基づく並列実行モデル HOPE [15] における実行例を単純化（深さ 2 までと）したもの。実行主体は 4 ワーカ。記載例は、ワーカ W1 の実行順（計画）、結果送信・受信、計算省略。

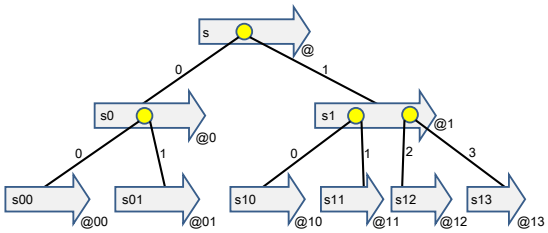

図 4　Hierarchical “multiple-order” computation and variable-length addresses (pedigrees). The circle indicates a nondeterministic execution order for a pair of subcomputations. When “s” at the root address “@” contains a pair of subcomputations {s0, s1}, s0 is at @0 and s1 is at @1. When s0 contains subcomputations {s00, s01}, s01 is at variable-length address @01; when s1 contains two pairs of subcomputations {s10, s11} and {s12, s13}, s12 is at variable-length address @12.

果が冗長に得られるものとする．

`result` 節は結果の場所（*lvalue* もしくは配列記法 [7]）を指定する．図 5 では `r0` や `r1` を指定している．なお，これらは代入の左辺であるとは限らない．

実用的なアプリケーションのために複雑な部分的計算結果を扱う場合には，`result` 節よりも汎用的な `reader`/`writer` 節 [17] を，通常の（脱）直列化技法を用いて利用するものとする．

`do_two` ディレクティブと同様に，HOPE `for` ディレクティブは，`omissible` 文を反復の本体とする反復処理において，すべての反復（間）の実行順序が任意であることを示す．反復全体は，HOPE コンパイラにより，2 分木状の階層的計算に変換される．

ここで，文献 [15] への補足として，図 6 に，バックトラック探索を用いた n クィーンパズルの全解探索の HOPE プログラムを示す．

`search` 関数の再帰呼び出しの前後で，作業空間 `nq_workspace` において副作用によりクィーン駒を設置して斜めの効きのデータも更新したり，再帰呼び出し後には除去（バックトラック）したりしているが，SPMO 並列実行においては各ワーカは固有の作業空間で逐次計算するため，特に問題ない．Tascell [5] では，タスク生成前の一時的バックトラックにおける undo（駒の除去）と，タスク生成後の redo（駒を再設置）を `dynamic_wind` 構文で指定するが，このような構文は HOPE には必要ない．

図 6 のプログラムでは，解の個数を計算結果とするが，HOPE では個数の部分和はワーカ内で副作用で「足し込む」ことはできない．このため，再帰呼び出し単位での部分和は配列の要素として求め，配列要素の総和（リダクション）は HOPE `for` の後に別途，実行している．Tascell [5] では，スティール時に新しいタスクで「足し込む」先を単位元として準備することが可能であり，一度配列の要素として求める必要はない．HOPE でリダクションを指定するための言語設計 [18] を進めているが，実装は今後の課題である．

4.2　拡張

文献 [15] と並行して HOPE のアプリケーションの開発を進める中で，リスト構造を扱う場合 [16] などの一部のアプリケーションでは計算省略時の仮の結果が必要なことが分かってきた．

```
int fib(int n) {
  if (n <= 2) return 1;
  int r0, r1;
  #pragma hope do_two // may execute in arbitrary order
  #pragma hope omissible result(r0, 0) // r0 is the partial result
  r0 = fib(n - 1); // may exchange r0 for omission
  #pragma hope omissible result(r1, 0) // r1 is the partial result
  r1 = fib(n - 2); // may exchange r1 for omission
  return r0 + r1;
}
```

図 5　HOPE program for Fibonacci.

結果が available であればそのデータを用いればよいが，より上位の階層の計算結果のみ available な場合も仮の値を用いて計算を省略しておくことができる．仮の結果やそれに基づくマージ結果は，最終的には分割統治のより上位の階層で正しい計算結果に置き換えられる．

ただし，上位階層の計算を行う前に，現在の階層で結果のマージを安全に行うためには仮の結果が必要であった．そこで `result` 節では仮の結果も指定できるものとする．例えば，図 5 の `result` 節では，`r0` や `r1` の指定とともに省略時の仮の値として `0` も指定している．

複雑な部分的計算結果を扱うための `reader`/`writer` 節 [17] についても，省略時に仮の結果を設定する方法として `omitter` 節を追加で持てるようにする．

5　HOPE 言語の実装

ここでは HOPE 言語の実装の概要を紹介する．詳細は文献 [15] を参照されたい．

5.1 節で紹介する基本的実装をベースとして，スケーラビリティの向上やオーバヘッド削減のための様々な工夫を行うことで，耐障害性を持ちながらも，耐障害性を持たない Tascell [5] の実装 [11] に匹敵するような高い性能が得られている [15].

5.1　基本的実装

現在のところ，複雑な実装を避けるため，各ワーカには OS の**プロセス**を用いている．

各ワーカは，それぞれに固有のメッセージ媒介システム（MMS）を従えている．各 MMS は，従事先ワーカと同じプロセス内に位置する．分散されたプロセスに対応して MMS も分散されており，これらの MMS は互いに通信相手となることで連携する．各 MMS は，**メッセージ**の従事先ワーカ用ストレージとなるとともに，メッセージの自動的な交換を担当する．各メッセージには，階層的計算のある部分に対して，その可変長アドレスならびに計算結果が含まれる．各メッセージは，どのワーカが（冗長に／ idempotent に）見出したかによらない不変・普遍な事柄であり，任意のワーカや MMS により交換できる．

可変長アドレスの集合からは共通プレフィックス部分を共通ノードとする木構造を考えることができるため，各 MMS は，可変長アドレスをキーとした**トライ木**（trie）としてメッセージを貯える．ワーカは，階層的計算の**現在関心のある**部分の可変長アドレスを**カーソル**を用いて指定する．ワーカは，階層的計算のその部分に関して，従えている MMS から／に対して／において，貯えられる部分的計算結果の読み出し／書き込み／存在確認を行える．

カーソルの動きに合わせてトライ木は拡張・縮小される．特にカーソルは**参照**としても働き，MMS 内で重み付き参照カウント方式を用いたメモリ領域の割り当て・解放 [19] を可能とする．階層の上位部分の計算結果が存在する場合，それより下位のトライ木のノードは，参照可能でない限り葉の側から削除されるようにする．

各 MMS はさらに，それぞれに固有の**通信通知層**（communication and notification layer; CNL）を従えている．CNL は MMS 間の**システムメッセージ**を担う．各 MMS はシステムメッセージのハンドラを

```
struct nq_workspace
{
  int a[20];
  int lb[40];
  int rb[40];
};
int nqueens(int n, int k, int ix, int iy,
            struct nq_workspace *ws)
{
  int m[20];
  int i;
  #pragma hope for // may execute in arbitrary order
  for (i = ix; i < iy; ++i)
    // m[i] is the partial result
    #pragma hope omissible result(m[i], 0)
    {
      int ai = ws->a[i];
      if (!(ws->lb[n-1+k-ai] || ws->rb[ai+k]))
        if (k == n - 1)
          m[i] = 1;
        else
          {
            ws->lb[n-1-ai+k] = 1;
            ws->rb[ai+k] = 1;
            ws->a[i] = ws->a[k];
            ws->a[k] = ai;
            // recursive call
            m[i] = nqueens(n, k+1, k+1, n, ws);
            ws->lb[n-1-ai+k] = 0;
            ws->rb[ai+k] = 0;
            ai = ws->a[k];
            ws->a[k] = ws->a[i];
            ws->a[i] = ai;
          }
      else
        m[i] = 0;
    }
  int s = 0;
  for (i = ix; i < iy; ++i)
    s += m[i];
  return s;
}
```

図 6　HOPE による探索プログラム

CNL に登録しておく．CNL も 従事先 MMS と同じプロセスに位置する．現在 CNL は，MPI と POSIX threads を用いて実装されている．CNL は，POSIX threads のスレッドとして，外部とシステムメッセージを通信するためのスレッドと，登録されたハンドラを呼び出すためのスレッドを用いている．ブロック状態が複数の MMS に波及していくのを防ぐため，CNL では MPI の長時間のブロッキング通信は用いず，MMS 間が疎結合となるように注意している．

各プロセス内部では，ワーカスレッドによる MMS の API の呼び出しと，CNL のスレッドによる MMS が登録したハンドラの呼び出しは，同時となることがあるため，POSIX threads による相互排他を適切に適用する．木のルートから葉に向かう半順序でのみ相互排他を用いることでデッドロックを避けるようにしている．また，MMS API として長時間ブロックするような API は提供されない．

HOPE コンパイラは，HOPE プログラムをコンパイルする際，MMS API の呼び出しを埋め込むようにする．（オーバヘッド削減のために複数の実行モードをうまく利用する以前の）基本的な実装しては，階層的計算の各部分（例えば，図 2 における s_2, s_4, s_8, ..., s_{249}）の開始時と終了時に，コンパイル後のプログラムを実行する各ワーカは，従えている MMS において，その部分の結果が現在貯えられていて直ちに利用可能（locally available）かを（存在確認のための API により）確認する．終了時に MMS に結果が存在しなければ，（書き込みのための API により）計算結果の書き込みを行う．開始時に MMS に結果が存在すれば，（読み出しのための API により）計算結果を読み出し，計算を省略する．

MMS に階層の上位部分の計算結果のみが存在する場合も，計算結果を読み出しをせずに，計算を省略しておき，後に，上位部分の正しい計算結果で置き換えることができる．

ワーカらが SPMO 並列実行で用いる実行順序の計画が適切で，MMS がうまく連携してメッセージを交換でき，冗長な計算を削減できれば，以上の HOPE の基本的実装は，HOPE モデルの基本的考え方を体現化することになる．

5.2　スケーラビリティの向上：事前共有計画

スケーラビリティの向上のため，ワーカそれぞれの実行順序を適切に計画し，事前共有しておくことで．以下のような工夫（詳細は文献 [15] を参照）を行う．

- MMS 間の結果（メッセージ）交換については，**ほぼ 1 対 1 対応**での適切な通信先の逆算（図 3 のように）
- 新規参入するワーカへの適切な**追い付き用情報**（catch-up information）の伝達（追い付き用情報は一連のメッセージからなる）
- 新規参入前の追い付き用情報の**プリフェッチ**

- **過度な協調の抑制**

これらの実現には事前共有計画全体のアレンジが重要となる．事前共有計画全体はワーカ数から，各ワーカの実行順序の計画はワーカ番号（ランク）から定まるように設計している．ワーカの実行順序の計画は，図 2 にあるような深さ（depth）それぞれで 0-優先か 1-優先かとして定めている．つまり，深さに関するビット列である．また，設計方針として，周期的なビット列としている．

図 3 の例では，周期 3 のビット列であり，この場合は 3 つ目（深さ 2）のビットは最初の 2 ビット（深さ 0 と深さ 1）の排他的論理和となっている．

図 3 の計画は，表 1 のように表現できる．表左端を深さ 0 と読む．深さ 0 と深さ 1 では 1 行のみに 1 が立っている一方，深さ 2 では 2 行ともに 1 が立っている．表上段を 0 行目と呼ぶことにすると，図 3 のような 4 ワーカ（2 ビットのワーカ番号）においては，表 0 行目をワーカ番号の上位ビット，表 1 行目をワーカ番号の下位ビットとする．これにより，表 1 は，深さ 0 と深さ 1 ではワーカ番号の上位ビットと下位ビットをそれぞれ使い，深さ 2 では 0 行目と 1 行目の両方で 1 が立っていることから，ワーカ番号の両方のビットの排他的論理和を使うと読む．

実行順序の計画は，計算が完了した部分から完了していない部分に完了した部分を優先していたワーカが参入してくることを考えると，参加者が増えても優先順序が 2 分することが望ましい．そのためには周期を長くしておくことも重要であるため，ハミング符号 [4] を応用することで，ワーカ数の $\log_2$ よりも長い周期を実現している．Hamming(15, 11) コードに基づくと 16 ワーカ（4 ビットのワーカ番号）において周期 4 ではなく周期 15 とできる．16 ワーカの場合の例の詳細は文献 [15] を参照していただきたいが，（表 1 の 2 ビットを 4 ビットに増やした場合の）表 2 を補足として示しておく．

文献 [15] に含められなかった工夫の候補として，より確実に結果（メッセージ）を伝搬させるために，

- 再送機能
- 近傍拡散機能

もテストしていた．文献 [15] での評価では，再送機能は off，近傍拡散機能は on となっていたが，その後の（「京」の運用停止前の）調査で近傍拡散機能は off とすべきという結果が得られた．

5.3 オーバヘッド削減：実行モードの使い分け

5.1 節で述べた基本的実装では，各ワーカは階層的計算の各部分の開始時と終了時に，その部分的計算結果の存在確認を行うとしているため，(100 倍規模の）膨大なオーバヘッドとなる．

文献 [15] に示すように，オーバヘッド削減のため，実際の処理系では，複数の実行モードを設けて適切に使い分けるとともに，計算状態操作機構 [6] [14] [20] [21] による動的な実行モード切替も採用している．

実行モードとしては次を用いている．

- **確認モード**（check mode）：基本的実装の通り，開始時と終了時に，その部分的計算結果の存在確認（とそれに伴う適切な読み出し／書き込みと計算省略）を行う．階層的計算で現在計算している部分（の可変長アドレス）に合わせて，カーソルを移動しておく．
- **非確認モード**（non-check mode）：開始時と終了時に部分的計算結果の存在確認を行わない．また，可変長アドレスを指定するためのカーソルの移動を行わない．これにより，計算省略はなされない一方で，オーバヘッドは大幅に削減される．確認モードで実行中，分割統治の階層のより下位の部分（葉により近い側）の計算を始める際に，その部分を計算しているワーカが自分一人だけとなったときには，この階層より下位については非確認モードで実行するようにする．分割統治計算が上位の階層に戻るときには確認モードに戻る．階層的計算のある部分の計算に何人のワーカが参加しているかの見積もりは MMS に問い合わせて知ることができる．MMS は事前共有計画と，現在のカーソル付近のメッセージの存在状況から各ワーカが計算している部分を，近くについてはかなり正確に，遠くについては概略（上位の階層としてどの部分か）で推測できる．
- **集中モード**（work-first mode）：HOPE ディレクティブを無視した C 言語での実行と同様に実

表 1　2 bits with exended 1 bit like Hamming(3, 1) codes.

1	0	1
0	1	1

表 2　4 bits with exended 11 bits like Hamming(15, 11) codes.

1	0	0	0	1	1	1	1	0	0	0	0	1	1	1
0	1	0	0	1	0	1	1	1	0	1	1	1	0	0
0	0	1	0	1	1	0	1	1	1	0	1	0	1	0
0	0	0	1	1	1	1	0	1	1	1	0	0	0	1

行する．すなわち，開始時と終了時に部分的計算結果の存在確認を行わない．また，可変長アドレスを指定するためのカーソルの移動を行わない．また，プログラム本来の順序で実行する．これにより数倍程度のオーバヘッドが削減される．非確認モードであっても，計画に従った順序で計算をすすめるために計画参照と順序選択処理，判断に伴う分岐予測ミスなどがあるためである．ある階層より下位の部分を非確認モードで実行し始めてから 10 段分下位の部分（葉により近い側）の計算を始める際に，この階層より下位については集中モードで実行するようにする．分割統治計算が上位の階層に戻るときには非確認モードに戻る．

さらに計算状態操作機構により，これらの実行モードは動的に切り替えられる．（すでに実行が開始されている）非確認モードから，確認モードへの切り替えは，MMS からの通知（ポーリングで検出）に基づく．MMS はある部分の計算への参加ワーカが増えて，一人ではなくなったときにこれを通知する．（すでに実行が開始されている）集中モードから非確認モードへの切り替えは，確認モードと集中モードにはさまれている非確認モードの段数が 10 段を下回っているときに行う．

確認モードへの切り替えの際は，非確認モードとして遅延していたカーソルの移動に加えて，非確認モードとして遅延していた部分的計算結果を書き出しのために呼び出し元（階層の上位部分）にある変数（図 5 であれば何段か前の呼び出しの `r0` や `r1`）などのデータにアクセスできる必要がある．計算状態操作機構 [6] [14] [20] [21] により，そのようなアクセスを安全に，また，通常の実行時オーバヘッドをあまりかけずに行える．

HOPE `for` ディレクティブが指定された反復処理において，反復全体は，HOPE コンパイラにより，2 分木状の階層的計算に変換される．具体的な 2 分木の形状については文献 [15] で述べていないが，2 べき分解 [18] と呼ぶこととした手法を用いている．

まず反復回数が 2 べきであれば，その均等な 2 分木は自明である．任意の反復回数については，これを 2 進数で表現したときに，1 が立っている部分（それぞれが 2 べき）のリストであると考える．このリストで MSB から LSB に向かう順を，2 分木のうち右下に順に降りていく形と認識し，各左下がリストの要素となるようにできる．(cons セルの car に 2 べき，cdr に残りのリストというのと同様.）ただし，最後の右下の部分木はリストの最終要素（2 べき）としておく．この木の上の移動は，スタックを用いなくても繰り返し回数を 2 進数で保持できるだけのビット列（を表す整数）があればよい．

また，各 2 べきの均等な 2 分木については，0-優先，1-優先による順序の選択のためのビット列と，「反復番号」を 2 進数で表したときのビット列の各桁が対応している．このため以下のような工夫が可能である．

- 確認モード：基本的実装の通り．
- 非確認モード：「単純にインクリメントする反復番号」と順序計画（の一部）との間での，ビット列としてのビット毎排他的論理和を用いることで，「計画通りの順序の場合の反復番号」が得られる．

- 集中モード：「単純にインクリメントする反復番号」のまま実行する．

6 評価

様々な工夫をした HOPE 実装の評価は，文献 [15] に示されているが，結果（メッセージ）**近傍拡散機能**を off にしたところ，性能の改善が見られたので，補足評価結果として図 7 に示しておく．

文献 [15] での評価で見られた 1 ノード 7 ワーカ実行から 2 ノード 14 ワーカ実行を見たときの並列効率の段差状の低下が解消されていることがわかる．この性能低下が発生していた原因としては，近傍拡散機能によるメッセージがさらなる近傍拡散のトリガーとなりオーバヘッドが増幅したこと，近傍拡散機能により結果伝搬を確実にすることの利点はあまりなかったことなどが考えられる．

7 HOPE アプリケーション

HOPE モデル [15] では，分割統治の階層的計算は，idempotent でなくてはならない．また，Tascell [5] などと違い，省略可能な計算結果を上位階層でマージしていくためには，1 つのマージ先に「マージしていく」形をとることができない．つまり，マージするまでは計算結果を分けておく必要がある．

idempotent の制約として，乱数生成が同一であってほしいという点がある．これについては，準乱数の形であるが，どのような計算順序としても，ディレクティブ無（本来の）計算順序の場合の乱数列と一致するように，乱数列をスキップしたり，後戻りしたりできる方式の研究を進めている．

また，マージするまでは計算結果を分けておく必要がある場合，計算結果がサイズが大きく，その初期化やマージのコストが大きい場合にそのコストが問題となる．例えばヒストグラムの分割統治型生成アプリケーション [16] では，通常の配列表現のサイズは大きい．この場合もリスト構造を併用するといったプログラミング技法 [16] を用いることで HOPE でもうまく扱えると考えられる．

データ分散に基づいて並列計算が分散される方式と比較すると，HOPE モデルでは，完全冗長実行からの階層的計算省略に基づくため，各ワーカが全データを必要とし得ることから，大容量メモリを必要とすると考えられるかもしれない．しかし，ワークスティール方式でも，分割統治型計算のどの部分を担当するかは動的に変わり得るため，それに備えると全データを必要とし得るという点に変わりはない．つまり，全データを必要とし得るため，扱うデータが大きい場合に問題となるのは，動的負荷分散に共通した問題であり，HOPE モデル固有の問題ではない．

むしろ，HOPE モデルの SPMO 実行では，実行順序が事前計画に基づくため，大きなデータを保持するための外部記憶や分散記憶に対してプリフェッチやブロック化のような手法を適用しやすいと考えられる．

8 関連研究／展望

HOPE モデルに直結する関連研究については文献 [15] を参照いただきたい．ここでは関連する研究背景や展望について述べる．

耐障害性における重要技術には不揮発性メモリの利用がある．不揮発性メモリの利用により，電源断に対する障害耐性を高めることができる．しかし，一貫性の面などから，不揮発性メモリの安全な利用は必ずしも容易ではない．HOPE モデルは「可変長アドレス k で定まる部分の計算結果は v である」という不変・普遍な事柄（メッセージと呼ぶ）を発見・複製・流通・記憶・利用・削除していくことで，最終的な計算結果を求めるまでの時間を短縮する手法ともいえる．不揮発性メモリを任意のメモリアクセスとしてではなく，メッセージ単位で利用する場合は一貫性を保ちやすくなることが考えられる．

近年は，GPU などのアクセラレータの利用が重要となっている．これについては，階層的計算の葉に近い部分でアクセラレータを用いる分には特に問題はないと考えている．

HOPE 実装でノード内共有メモリを複数ワーカのスレッドを含む形で活かすことも計画している．スレッド並列については，言語仕様としての仕様の更新があったり，メモリモデルなどの複雑な問題もあり，今後重要となる研究といえる．

近年は，仮想環境を利用したシステムも普及してい

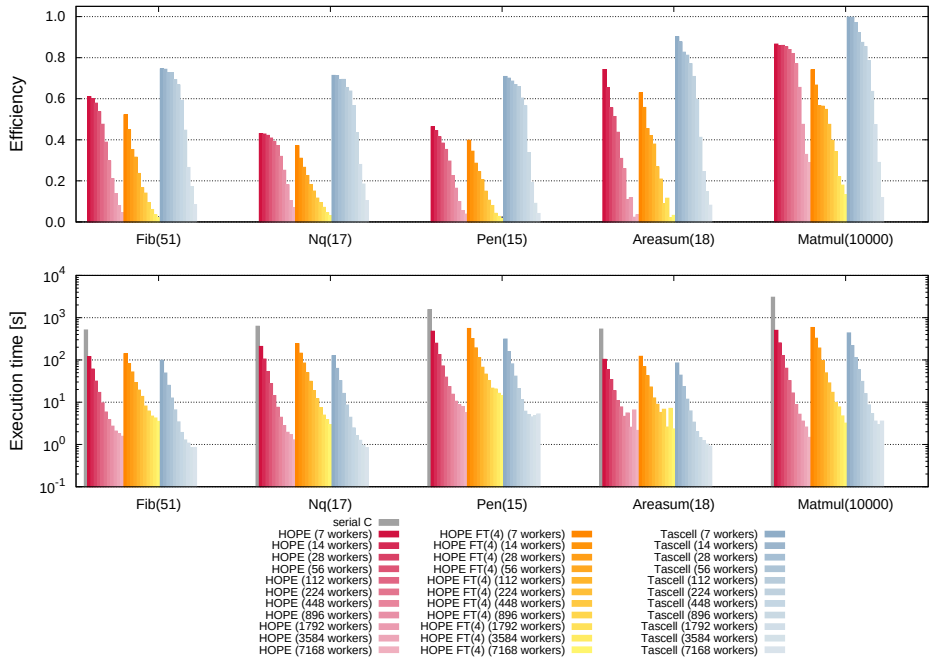

図 7　Efficiency (upper half) and execution times (lower half) of parallel systems using multiple workers on the K computer, in addition to execution times of serial C programs (lower half). Efficiency is defined as S/n_w, where S is the speedup relative to a sequential C program and n_w is the number of workers. (Efficiency $= 1$ means an ideal speedup.) HOPE uses 1 to 1024 nodes each of which uses 7 worker processes. Tascell uses 1 to 1024 nodes each of which uses a single process with 7 worker threads. In HOPE FT(4), one out of four workers are intentionally stopped with fault injections.

る．仮想環境で HOPE モデルがうまく働くのかも重要といえる．本来の計算よりも，計算省略するための処理が仮想化の影響を受けやすいと考えられ，十分な計算省略が難しくなる場合が懸念される．

HOPE モデルでは事前共有計画全体のアレンジが重要であると述べたが，その良さの追求においては，どのようなバリエーションが考えられ，どのような指標を重視し，どのような問題を避けるべきで，どこまでの改善が見込めるか，不連続な改善しかないか，などにおいて不明な点が多い．本研究では伝統的なハミング符号 [4] を異なる視点から応用しているが，符号理論も近年，進歩しており，それらを応用できるとよい．

処理系の良さとしては高性能かどうか以外に，障害耐性を評価したい．例えば，ワークスティール方式ではルートを担当するワーカに障害が発生したときの影響が大きいはずである．このため，障害を模擬することで，重要なワーカにおける障害発生のケースを意図的に含めて障害耐性を評価を行う妥当な方法 [22] について研究を進めている．

MPI を用いる大規模並列計算システムにおいて，本当のあるいは意図的な障害に対する HOPE 実装の耐障害性を確認するには，MPI プロセス集合の一部に障害が発生しても残りの MPI プロセスでは通信や計算が可能な範囲で持続できることが重要である．このためには，耐障害性 MPI（[2] 等）や User Level Failure Mitigation (ULFM) が実際に利用できることが望ましい．

HOPE 実装では，計算状態操作機構を用いて動的に実行モードを切り替えている．計算状態操作機構を利用して呼び出し元の変数の値にアクセスするため，切り替え処理の正しさは自明ではない．動的実行モード切替があっても HOPE 言語のプログラムの計算結果に影響しないことの証明（あるいは証明技法）なども今後必要な研究と考えている．

9 おわりに

本発表では，文献 [15] の内容に基づき，HOPE モデル，HOPE 言語と実装の概要を紹介するとともに，いくつかの拡張と補足，展望（今後の課題）を述べた．

謝辞 本研究の一部は JSPS 科研費 JP19H04087 と JP17K00099 の助成を受けたものである．結果の一部は，理化学研究所のスーパーコンピュータ「京」を利用して得られたものである．本研究の一部は，「京」のプログラミング環境の高度化に関する研究開発，に関する理化学研究所 計算科学研究センターとの共同研究による．

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, *SIGPLAN Not.*, Vol. 40, No. 10(2005), pp. 519–538.

[2] Fagg, G. E. and Dongarra, J.: FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World, *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, London, UK, UK, Springer-Verlag, 2000, pp. 346–353.

[3] Frigo, M., Leiserson, C. E., and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices (PLDI '98)*, Vol. 33, No. 5(1998), pp. 212–223.

[4] Hamming, R.: Error Detecting and Error Correcting Codes, *Bell System Techincal Journal*, Vol. 29, No. 2(1950), pp. 147–160.

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

[6] Hiraishi, T., Yasugi, M., and Yuasa, T.: A Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol. 2(2006), pp. 262–279. (IPSJ Transaction on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50-67.).

[7] Intel Corporation: Introducing Intel® Cilk™ Plus: Extensions to simplify task and data parallelism, `http://www.cilkplus.org`.

[8] Intel Corporation: *Intel Threading Building Block Tutorial*, 2007. `http://threadingbuildingblocks.org/`.

[9] Michael, M. M., Vechev, M. T., and Saraswat, V. A.: Idempotent Work Stealing, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*, February 2009, pp. 45–54.

[10] Mohr, E., Kranz, D. A., and Halstead, Jr., R. H.: 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.

[11] Muraoka, D., Yasugi, M., Hiraishi, T., and Umatani, S.: Evaluation of an MPI-Based Implementation of the Tascell Task-Parallel Language on Massively Parallel Systems, *Proceedings of the 45th International Conference on Parallel Processing Workshops (ICPPW 2016) (Ninth International Workshop on Parallel Programming Models and Systems Software for High-End Computing P2S2 2016, held in conjunction with ICPP 2016)*, August 2016, pp. 161–170.

[12] Taura, K., Tabata, K., and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP'99)*, May 1999, pp. 60–71.

[13] Umatani, S., Yasugi, M., Komiya, T., and Yuasa, T.: Pursuing Laziness for Efficient Implementation of Modern Multithreaded Languages, *Proceedings of the Fifth International Symposium on High Performance Computing*, Lecture Notes in Computer Science, No. 2858, October 2003, pp. 174–188.

[14] Yasugi, M., Hiraishi, T., and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proceedings of the Fifteenth International Conference on Compiler Construction (CC2006)*, Lecture Notes in Computer Science, No. 3923, Springer-Verlag, 2006, pp. 170–184.

[15] Yasugi, M., Muraoka, D., Hiraishi, T., Umatani, S., and Emoto, K.: HOPE: A Parallel Execution Model Based on Hierarchical Omission, *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*, August 2019, pp. 77:1–77:11.

[16] 佐多育斗, 八杉昌宏, 平石拓, 馬谷誠二: 分割統治型総和の部分的計算結果を効率よく利用する方式の研究, 情報処理学会第 121 回プログラミング研究会, November 2018.

[17] 橋本孝太, 八杉昌宏, 平石拓, 馬谷誠二: 汎用送受

信に対応した HOPE コンパイラの研究, 情報処理学会第 125 回プログラミング研究会 (SWoPP2019), July 2019.
[18] 重本孝太, 八杉昌宏, 平石拓, 馬谷誠二: HOPE コンパイラのプロトタイプ実装, 情報処理学会第 115 回プログラミング研究会 (SWoPP2017), July 2017.
[19] 諏訪将大, 八杉昌宏, 平石拓, 馬谷誠二: 分散進捗管理のためのメッセージ媒介システムにおける 不要メッセージ削除機能, 情報処理学会第 105 回プログラミング研究会 (SWoPP2015), August 2015.
[20] 八杉昌宏, 平石拓, 篠原丈成, 湯淺太一: L-Closure：高性能・高信頼プログラミング言語の実装向け言語機構, 情報処理学会論文誌：プログラミング, Vol. 11, No. SIG 1 (PRO 13)(2008), pp. 118–132.
[21] 田附正充, 八杉昌宏, 平石拓, 馬谷誠二: L-Closure の呼び出しコストの削減, 情報処理学会論文誌　プログラミング, Vol. 6, No. 2(2013), pp. 13–32.
[22] 西牟禮亮, 八杉昌宏, 平石拓, 馬谷誠二: ワーカの重要度を考慮した並列実行フレームワークの障害耐性評価手法の検討, 情報処理学会第 125 回プログラミング研究会 (SWoPP2019), July 2019.