実環境向け並列言語実装技法の仮想環境における有効性の調査

與田 拓磨　八杉 昌宏　平石 拓　光来 健一

並列言語ならびにその処理系は，並列環境において効率良い並列計算を実現するために重要な役割を果たす．並列言語には，Cilk 言語，Tascell 言語，HOPE 言語などがあり，その処理系では，ワークスティールによる実行時負荷分散（Cilk，Tascell）や，階層的計算省略による耐障害性（HOPE）が実現されている．様々な並列言語実装技法があり，1 つの選択肢としてビジーウェイトがある．これは，Tascell のワークスティールの仕事待ちに用いた場合の有効性が実環境において確認されている．近年は仮想環境における効率良い並列計算も必要であり，本研究では上記のような実環境向けの高速化手法が仮想環境ではどのような効果を持つかについて調査を行う．仮想環境における実験では，仮想化ソフトウェアとして Xen を用いる．

1　はじめに

並列言語には Cilk 言語 [3]，Tascell 言語 [4]，HOPE 言語 [6] などがあり，その処理系では，ワークスティールによる実行時負荷分散（Cilk，Tascell）や，階層的計算省略による耐障害性（HOPE）が実現されている．

従来の Tascell では，ワークスティール時のタスク要求後のタスク返信待ちにおいて一時的にブロック状態としている．そのためタスクの返信が来た場合すぐに処理へと移行できない場合があり，並列計算の性能が低下する場合がある．そこで，並列言語実装技法の 1 つであるビジーウェイトをワークスティールのタスク返信待ちに用いた．結果として実環境でのビジーウェイトの有効性が確認されている．

近年は仮想環境における効率の良い並列計算も重要であるが，仮想環境上では実環境向け高速化手法が実環境とは異なる効果をもたらす可能性がある．

本研究では実環境向け高速化手法の 1 つであるビジーウェイトが仮想環境ではどのような効果を持つかについて調査を行う．

本論文の構成は以下の通りである．第 2 章では本研究で対象とする並列言語である Tascell の概要を説明する．第 3 章では Tascell におけるワークスティールの概要を説明した後，タスク返信待ちの，条件変数を用いる既存の実装手法とビジーウェイトを用いる手法の違いを述べる．第 4 章では実環境での評価を行う．第 5 章では仮想環境の概要と本研究の仮想環境の構築方法について述べる．第 6 章では仮想環境での評価を行う．第 7 章では PAUSE 命令を利用した場合について，仮想環境上で効果が期待できる PAUSE-Loop Exiting を考慮した評価を行う．第 8 章では関連研究の紹介を行う．第 9 章では今後の課題について述べる．

---

* Investigating How Virtual Environments Affect the Effectiveness of Pallarel Language Implementation Techniques for Real Environments.

Takuma Yoda, 九州工業大学大学院情報工学府, Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology.
Masahiro Yasugi, Kenichi Kourai, 九州工業大学大学院情報工学研究院, Dept. of Computer Science and Networks, Kyushu Institute of Technology..
Tasuku Hiraishi, 京都大学学術情報メディアセンター, Academic Center for Computing and Media Studies, Kyoto University.

2 並列言語 Tascell

Tascell の概要について，文献 [4] [7] に基づいて述べる．Tascell はワークスティールにより並列計算を実現している拡張 C 言語である．素朴な並列化の方法としては，各ワーカがその時々の状態に基づいてタスク生成するかどうかを判断することが考えられる．すなわち，各ワーカが最外ループのすべての反復を自分で計算するか反復の一部をタスクとして生成するかを，何らかの基準（タスクワーカが存在するときのみタスクを生成する，など）で判断する．効率の良い負荷分散のためには，各ワーカは計算の初期段階で適切な数のタスクを生成しておき，その後はずっと（計算終盤の微調整を除き）生成を行わないような判断基準が最善である．

Tascell では一時的バックトラックを行うことで，タスクの遅延分割を行なっている．すなわち，ワーカは，常に最初は「タスクを生成しない」ことを選択するが，他のワーカからタスク要求を受けると，過去の選択を変更したかのようにタスクを生成する．Cilk [3] における LTC と呼ばれる手法に対して，論理スレッドを一切生成しないので，タスクキュー管理コストが発生しないなどの優位点を持つ．

3 Tascell の実環境向け高速化手法

3.1 Tascell のワークスティール

Tascell のワークスティールについて [4] をもとに説明する．Tascell ワーカは自身のタスクスタックが空になると，thief ワーカとしてタスクを実行中のワーカ（victim）にタスク要求メッセージを送信する．要求を受けたワーカは一時的バックトラックにより最古のタスク生成可能状態を復元した後，タスクを生成して thief ワーカに送信する．Tascell では一時的バックトラックによって，より仕事量の多いタスクを生成して渡すことができる．タスクを受け取った thief ワーカはそのタスクを実行し，結果を victim ワーカに返信する．

3.2 条件変数による返信待ちの実装

タスク要求を送信した thief ワーカは，その返信が返ってくるまで待機する必要があるが，従来の実装ではこの待機を条件変数を用いて実現していた．すなわち，ワーカに対応する OS スレッドはタスク要求メッセージを送信した後，`pthread_cond_wait` によりブロック状態となる．タスク要求に対する返信が届くと，`pthread_cond_signal` あるいは `pthread_cond_broadcast` によりこのワーカスレッドを再開させる．

この方式では，返信待ちのワーカにより計算資源が浪費されることがない一方，ブロック状態への移行やブロックの解除に時間を要してしまうという問題点がある．

3.3 ビジーウェイトによる返信待ちの実装

ビジーウェイトとは共有メモリに同期用の変数を設けて，そのフラグの状態を繰り返し確認することにより同期待ちを実現するという並列処理における実装技法の 1 つである．条件変数を用いる代わりにビジーウェイトでタスク要求に対する返信待ちを行うようにすることで，ワーカスレッドは返信待ちの際に計算資源を消費するようになる一方，タスク要求に対する返信を受け取った後に処理を再開するまでの時間を短縮できることが期待できる．

実装には図 1 に示す `pthread_cond_busywait` マクロを使用した．このマクロはロック `PMUT` を解放した状態で式 `EXP` が偽になるまで繰り返し評価し，偽になれば `PMUT` を再び獲得するという処理を行うものである．

このマクロを用いたビジーウェイトによるタスク返信待ちの処理のコードを図 2 に示す．なお，このコードは説明のために一部簡略化してあり，また，比較のために条件変数を用いた処理コードも図に示している（`#ifdef` ディレクティブの`#else` 節）．ビジーウェイトを用いた実装では，ワーカスレッドは `pthread_cond_wait` を呼ぶ代わりに，`pthread_cond_busywait` を用いてビジーウェイトを継続するための条件式の値を繰り返し評価する．ここで，`volatile` 型修飾子を持つ型への型キャストを

```
#define pthread_cond_busywait(EXP,PMUT) \
do {                                    \
  pthread_mutex_unlock (( PMUT ));      \
  while (( EXP ));                      \
  pthread_mutex_lock (( PMUT ));        \
} while (0)
```

図 1　pthread-cond-busywait マクロ

用いることで，コンパイラの最適化による条件式の繰り返し評価の省略や実行順序の入れ替えを防いでいることに注意する．

pthread_cond_busywait 内での条件式の評価はロック thr->mut を解放した状態で行っているため，pthread_cond_busywait を抜けたとしても，最後の評価の後に条件式の真偽が変化している可能性を排除できない †1．そのため，ワーカスレッドは pthread_cond_busywait を抜けた後 while ループの先頭に戻り，thr->mut を再獲得した状態で条件式の値を再確認するようにしている．

なお，図 2 中の thr->sub は，ワーカが他のワーカに送信して結果待ち状態になっているタスクの存在の有無を示している．そのようなタスクが存在する場合，タスク要求を送信したワーカは，その要求に対する返信があったとき（tx->stat が TASK_ALLOCATED 以外になったとき）だけでなく，結果待ちのタスクの結果を受け取ったとき（thr->sub->stat が TASK_HOME_DONE になったとき）にも返信待ちを抜けることになっている．

4　実環境での効果

4.1　評価方法

並列言語実装技法の 1 つであるビジーウェイトを Tascell のワークスティールのタスク返信待ちに適用した場合の実環境上での効果を評価する．実環境として用いた並列実行環境を表 1 に示す．

ベンチマークプログラムには文献 [8] と同様の，n 番目のフィボナッチ数を再帰的に計算する（fib(n)），

†1　条件式の評価による確認がデータレースとなっているのも意図的である．ここで，データレースの影響は条件式の真偽値にのみ現れると仮定している．

表 1　並列実行環境（実環境）

	Linux PC Server
OS	Ubuntu 16.04.3 LTS 64bit
CPU	Intel Xeon E5-2630 v3 2.40GHz 8 コア × 2
メモリ	64GB
コンパイラ	Tascell コンパイラ，GCC 5.4.0
ハイパースレッディング	無効

$n \times n$ の行列の LU 分解を cache-oblivious な再帰アルゴリズムで計算する（lu(n)），n ピースのペントミノパズル（$n > 12$ は追加のピースと拡張ボードを使用する）の全解探索を行う（pen(n)），2 つの n 要素の配列間の全要素ペア（(a_i, b_j) for all $0 \leq i, j < n$）について比較演算を実行するプログラム（cmp(n)），$(2n+1)^3$ 個の同一質量の質点からある点にかかる総引力を計算するプログラム（grav(n)）を用い，上記プログラムを実環境上 16 ワーカで実行し，その実行時間を調べることにより評価を行った．

4.2　評価結果

実環境上での評価結果を図 3 に示す．アプリケーションを 22 回実行し，その中央値で評価を行った．それぞれのアプリケーションで pthread_cond_wait（ビジーウェイト無し）の結果を 1 として busywait（ビジーウェイト有り）の比率を求め，棒グラフとして表している．各棒グラフの上部に実際の実行時間 [秒] を表記している． 全てのアプリケーションでビジーウェイトを適用した方が実行時間が短くなり，特に fib(44) にて 1.9%程度，lu(4000) にて 4.2%程度実行時間が短縮された．

ここまで並列言語実装技法のひとつであるビジーウェイトの実環境上での Tascell ワークスティールにおける有効性を示した．

5　仮想環境

5.1　概要

仮想環境における仮想マシン（VM）では，CPU は物理的な CPU コア（物理 CPU）を仮想化した仮想的な CPU（仮想 CPU）として提供されている．1 台の実環境マシンに対して 1 台の仮想マシンであれば，仮想 CPU 1 つにつき 1 つの物理 CPU が割り当

```
while (1) {
  if (thr->sub) {
    if (!(tx->stat == TASK_ALLOCATED && thr->sub->stat != TASK_HOME_DONE))
      break; // 返信到着確認（または到着待ちの必要がなくなった）
  } else {
    if (!(tx->stat == TASK_ALLOCATED)) {
      break; // 返信到着確認
    }
  }

#ifdef BUSYWAIT
  /* ビジーウェイトを用いる実装 */
  if (thr->sub) {
    pthread_cond_busywait (*(volatile enum task_stat*) (&tx->stat) == TASK_ALLOCATED
         && *(volatile enum task_home_stat*) (&thr->sub->stat) != TASK_HOME_DONE,
         &thr->mut);
  } else {
    pthread_cond_busywait (*(volatile enum task_stat*) (&tx->stat) == TASK_ALLOCATED,
         &thr->mut);
#else
    /* 条件変数を用いる実装 */
    pthread_cond_wait (&thr->cond, &thr->mut);
#endif
  }
} // while の先頭に戻り，thr->mut のロック獲得した状態で再確認
```

図 2　タスク要求に対する返信待ちの実装

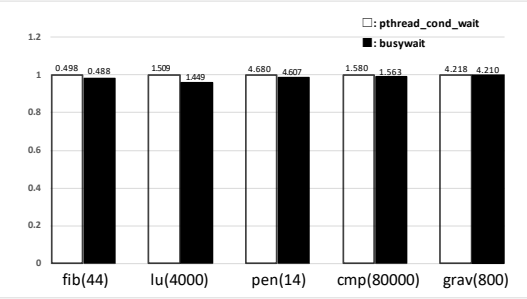

図 3　実環境上での相対実行時間

てられる．しかし仮想マシンを利用する場合，運用上の理由で 1 台の実環境マシンに複数台の仮想マシンを配置することがある．そういった場合，仮想 CPU に対し物理 CPU が不足することがある．

本研究では様々な場合を想定し，16 コアの（表 1 の CPU とメモリを持ち，ハイパースレッディングは無効化している）実環境マシン上に仮想 CPU 数 16 の VM を複数台構築して評価を行う．

5.2　構築方法

主な仮想環境の構築方法にはホスト型とハイパーバイザ型がある．ホスト型は OS 上で仮想化ソフトウェアを利用して VM を動かすもので VMware Player や Oracle VirtualBox などがある．ハイパーバイザー型では直接サーバに VM を立ち上げるもので Xen [1] などがある．ホスト型の構築では簡単に VM を構築することができるなどの利点があり，ハイパーバイザー型の構築では OS を介さずに VM を構築するためハードウェアのリソースを直接管理できるなどの利点がある．

本研究では仮想化ソフトウェアとして，ハイパーバイザ型の構築方法を持つ Xen 4.6 を用い（表 2），ハイパーバイザをハードウェア上で直接実行した．その上でドメイン 0 と呼ばれる制御用仮想マシンを実行し，ドメイン U と呼ばれる仮想マシンを用いて実験を行った．構築したドメイン U の設定を表 3 に示す．

表 2　使用したハイパーバイザ

	ハイパーバイザ
バージョン	Xen 4.6

表 3　ドメイン U（仮想環境 1 台分）の設定

	仮想環境
仮想マシン上の OS	Ubuntu 16.04.5 LTS 64bit
仮想化方式	HVM
仮想 CPU	16
メモリ	1.6GB

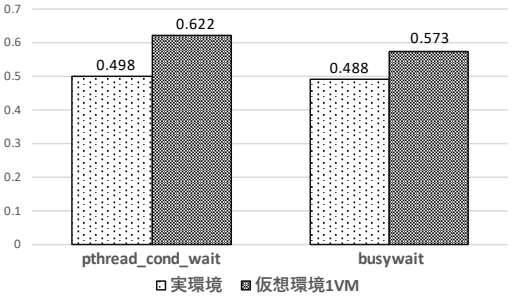

図 4　実環境と仮想環境 1VM での fib(44) の実行時間

6　仮想環境での効果

6.1　評価方法

Tascell アプリケーションの 1 つで，フィボナッチ数を求める fib を利用し，実行時間を計測することにより評価を行う．仮想マシン（表 3）を複数台同時に起動し，parallel ssh を利用してそれらに同時にアプリケーションを実行させてその実行時間を計測する．ビジーウェイト適用前後の fib(44) による性能比較のため，はじめに実環境と VM 数が 1 台の場合を比較して評価を行い，その後 VM 数（「16 仮想 CPU で 16 ワーカ実行する VM」の数）を 1, 2, 4, 8, 16 台と変化させ，それぞれの場合について，平均実行時間（全 VM 実行時間合計/VM 数）からの換算実行時間（平均実行時間/VM 数）を求めて評価を行った．

6.2　評価結果

実環境と仮想環境 VM1 台で fib(44) を 22 回実行して，実行時間の中央値を図 4 に示す．実環境上に比べて仮想環境 1VM の方が実行時間が長くなるが，これは仮想環境での実行時はドメイン U が 1 台であってもドメイン U だけでなくドメイン 0 を並行実行していることや，それを含めた仮想化全般のオーバーヘッドが原因ではないかと考えられる．

pthread_cond_wait（ビジーウェイト無し）の場合と busywait（ビジーウェイト有り）の実行時間の短縮割合を比較した場合，実環境では 1.9%程度，仮想環境 1VM では 7.9%程度短縮されている．

pthread_cond_wait の場合タスク要求の返信を待つワーカは通知が来るまでブロックするが，仮想環境の方が実環境に比べて通知によってワーカ起こされるまでに余計に時間がかかる．そのため仮想環境の方がビジーウェイトを適用することによる実行時間の短縮が大きくなったのではないかと考えられる．

busywait の場合も，仮想環境では実環境より実行時間が長くなっているが，これは，ビジーウェイトを，すべてのウェイトに対しては適用していないため，つまり，条件変数でウェイトする場合が残っていることもその要因として考えられる．具体的にはタスク要求の返信待ちでビジーウェイトを適用しているものの，同時に結果待ちとしていたサブタスクの結果が先に受け取れた場合は，タスク要求については後で必ず none が返信されることが確定しているため，none が返信されることのウェイトをすぐには行わず，返信待ち数の記録のみで，次のタスク要求直前まで遅延しており，遅延後のウェイトは条件変数でのウェイトのままになっている．残っている条件変数でのウェイトにもビジーウェイトを適用し，`pthread_cond_broadcast` も削除する場合の調査は今後の課題である．

VM 数を 1, 2, 4, 8, 16 台と変化させた場合の評価結果を図 5 に示す．fib(44) を 22 回実行して換算実行時間を求め，その中央値を棒グラフで示し，四分位数をプロットしている．VM 数が 1 の場合には，上の評価結果にもあるようにビジーウェイトの適用により実行時間が 7.9%程度短縮された．それに対し VM 数が 2, 4, 8, 16 の場合にはビジーウェイトの適用により実行時間が増加する結果となった．また VM 数

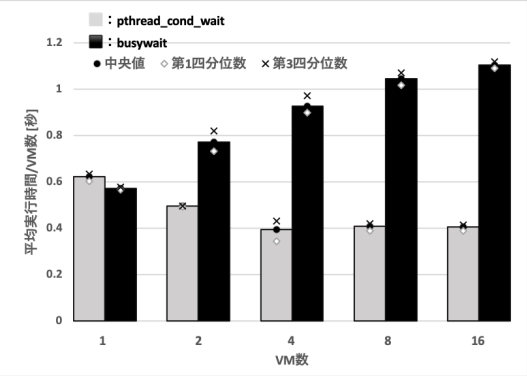

図 5　VM 数を変えたときの換算実行時間

が増えるほど実行時間が増加する．

これは返信が得られるまでの確認の繰り返しにより，有限の計算資源が消費されてしまうためであり，VM 数が増えるほど消費される割合が増えていくためだと推測される．

pthread_cond_wait の場合，VM 数が増えるほど実行時間が短縮されるが，これは仮想 CPU をオーバーコミットしたことにより効率的に物理 CPU が利用されたり，特に終盤でのワークスティールの増加が他の VM の処理があったために抑制されたりしたのではないかと考えられる．

7　PAUSE 命令の効果

7.1　概要

マルチスレッドを使用する並列プログラムを正しく機能させるには，何らかの同期操作が必要であり．通常の同期操作は，共用の同期変数と，それをチェックする「スピン・ウェイト」ループで行う．現在，大部分の Intel プロセッサ [2] ではスピン・ループに関するパフォーマンス問題に対処するための命令として PAUSE 命令を提供している．同期変数をあまりに頻繁にチェックすると，不必要にシステム・リソースを消費し，プロセッサがループを抜けるまでにかなり大きなペナルティを受けることとなる．PAUSE 命令をループに入れることによって，ループにわずかな遅延が入り，システム・リソースの使用量が少なくなり，スピン・ウェイト中の電力消費量が大きく減少する効果が得られる．

```
#define pthread_cond_busywait(EXP,PMUT) \
do {                                    \
  pthread_mutex_unlock (( PMUT ));      \
  while (( EXP )) xcc_pause();          \
  pthread_mutex_lock (( PMUT ));        \
} while (0)
```

図 6　PAUSE 命令を追加した pthread-cond-busywait マクロ

仮想環境上でタスクを要求した後の返信をビジーウェイトして待つ場合，実行時間がビジーウェイトしない場合に比べ長くなっているが，ビジーウェイトの繰り返しに PAUSE 命令を用いることで上記と同様の効果が得られる可能性がある．PAUSE 命令を仮想環境で利用するにあたって，Intel の仮想化対応プロセッサには Pause Loop Exiting（PLE） [5] という機能がある．PLE を利用することにより PAUSE 命令が連続的に使用された場合，ハイパーバイザに制御を戻すことができる．PLE には PLE_Gap と PLE_Window という 2 つの設定値がある．

- PLE_Gap：PAUSE 命令が連続していると認識するまでの最大時間
- PLE_Window：PAUSE 命令が連続して実行される最大時間

PLE_Gap の設定値以下の間隔で連続して PAUSE 命令が実行され，その実行が PLE_Window の設定値以上の時間連続すると，PLE によるハイパーバイザへの exit が発生する．

7.2　評価方法

ビジーウェイトマクロのループ部分に PAUSE 命令を追加した図 6 として，先ほどの評価と同様に，アプリケーション fib を利用して評価を行う．

- `pthread_cond_wait`+ PLE（gap = 128, window = 4096）
- ビジーウェイト PAUSE 命令無+ PLE（gap = 128, window = 4096）
- ビジーウェイト PAUSE 命令有+ PLE（gap = 128, window = 4096）
- ビジーウェイト PAUSE 命令有+ PLE（gap =

512, window = 1024）
の 4 つ場合で実行する．上 3 つの PLE の値はデフォルトの値であり，PAUSE 命令有りの場合には PLE が確実に動くような PLE の値も設定している．VM 数（「16 仮想 CPU で 16 ワーカ実行する VM」の数）を 1, 2, 4, 8, 16 台と変化させて，平均実行時間（全 VM 実行時間合計/VM 数）からの換算実行時間（平均実行時間/VM 数）を求めて評価を行った．

7.3 評価結果

評価結果を図 7 に示す．fib(44) を 22 回実行し，換算実行時間を求めて中央値を棒グラフで示している．

デフォルト PLE+PAUSE 無の場合と，確実な PLE+PAUSE 有の場合の実行時間を比較すると，VM が 1 台の場合は実行時間にほぼ差が無い結果（デフォルト PLE+PAUSE 無で 0.573[s]，確実な PLE+PAUSE 有で 0.570[s]）となり，VM が 2 台以上の場合は確実な PLE+PAUSE 有の場合の方が実行時間が短くなり，VM が 2 台の場合には 8.0%程度実行時間が短縮されている．pthread_cond_wait の場合と確実な PLE+PAUSE 有の場合を比較すると，VM が 1 台のときののみ確実な PLE+PAUSE 有の方が実行時間が短く，それ以外は pthread_cond_wait の場合の方が実行時間が短くなる結果となった．

PAUSE 命令と確実な PLE 設定を組み合わせることによって，VM が複数台のときのビジーウェイトによるリソースの余計な消費を多少抑えられたが，pthread_cond_wait と比較すると，実行時間が長くなっている．その理由としては，exit して他の仮想 CPU に切り替えたとしても，切り替え先の仮想 CPU も結局ビジーウェイトしている場合がある，といった点や，exit によって最も切り替えたい先は thief と同じ VM にいる victim ワーカであるが，VM 数が増えることで順番が回って来にくい点，などが考えられる．

図 7 の評価では，図 6 による PAUSE 命令と確実な PLE 設定（デフォルトとは異なる設定）の両方を用いていたため，それぞれ単独の効果を，図 7 で性能比が最も大きい VM 数 2 の場合について調査した．55 回の実行に関して換算実行時間の中央値（並びに第 1 四分位数，第 3 四分位数）を求めた：

- pthred_cond_wait+デフォルト PLE:0.479（0.466，0.494）
- PAUSE 無+デフォルト PLE : 0.784（0.753，0.839）
- PAUSE 有+デフォルト PLE : 0.780（0.737，0.829）
- pthread_cond_wait+確実な PLE : 0.488（0.481，0.495）
- PAUSE 無+確実な PLE : 0.734（0.688，0.764）
- PAUSE 有+確実な PLE : 0.709（0.655，0.760）

同じ結果を図 8 にも示す．

タスク要求の返信待ちのビジーウェイトで PAUSE 命令を用いない場合であっても，デフォルト PLE から確実な PLE に設定変更することで，換算実行時間の中央値が 0.050 秒程度も短縮されている．この理由としては，タスク要求の返信待ちのビジーウェイト以外でも例えば `pthread_mutex_lock` やゲスト OS 内部で PAUSE 命令が用いられていること，その部分に対して確実な PLE の効果があったこと，が考えられる．また，タスク要求の返信待ちのビジーウェイトで PAUSE 命令を用いていた場合は，デフォルト PLE から確実な PLE に設定変更することで，0.071 秒程度とさらに大きく中央値が短縮されている．

確実な PLE では，PLE_Gap を 128 から 512 に増やすことでキャッシュミスなどで数百サイクル程度のペナルティがあっても対象から外れにくくなり，PLE_Window を 4096 から 1024 に減らすことで exit するまでの資源消費が低減されている．

また，PAUSE 命令無から有にプログラムを変更することの単独の効果は，デフォルト PLE では換算実行時間の中央値で 0.004 秒程度の短縮と小さい．確実な PLE の場合は，0.025 秒程度の短縮と，PLE の設定変更ほどではないが，一定の短縮効果が見られる．ただ，確実な PLE の場合に，換算実行時間の第 3 四分位数では PAUSE 命令無と有の差はほとんどなく，55 回の実行間の「ばらつき」が大きいといえる．

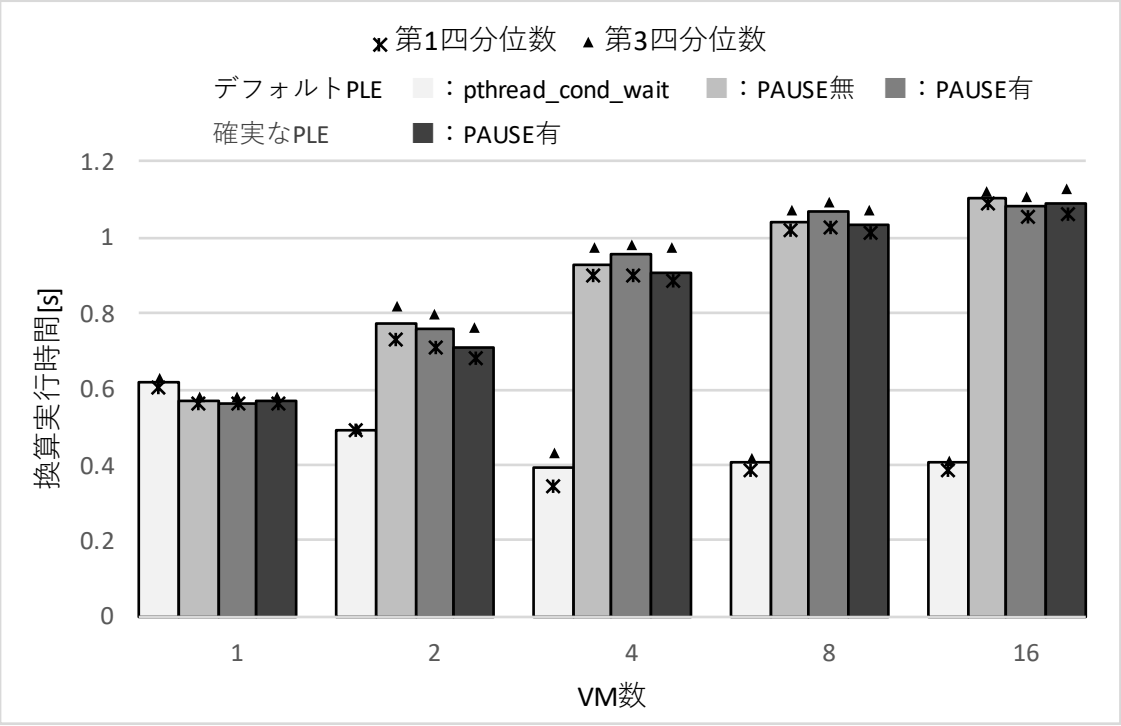

図 7　換算実行時間

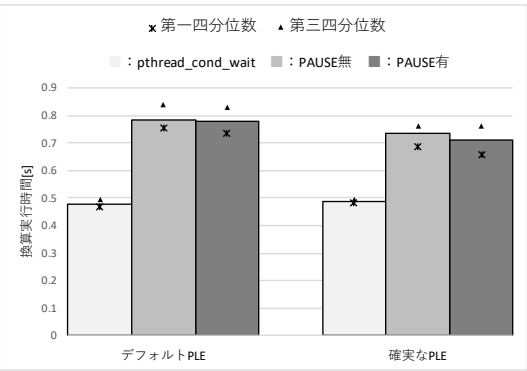

図 8　VM 数 2 の場合の比較

8　関連研究

8.1　仮想環境を考慮した要求駆動型負荷分散

CPU の物理コア数を超えて仮想 CPU を運用する場合，物理 CPU を時分割で割り当てるため，いずれかの仮想 CPU は休止状態となっている時間が存在する．そのような状況では Tascell を用いた並列計算において並列処理の性能が低下する恐れがある．この問題への対応を試みる研究 [9] がある．

研究 [9] では，本研究で「pthread_cond_wait によるタスク要求待ち」とした実装（本研究での調査では性能低下は見られず，むしろオーバコミットの場合にはスループットが向上する）をベースとして，各ワーカがすぐにスティールできるタスクを一定数用意しておくことにより，スティールされる側の関与なし（一方の CPU 資源の利用）でスティールできる機会を増やす手法が提案されている．

8.2　VM が利用可能な CPU 数の変化に対応した並列アプリケーション実行の最適化

VM 内で並列アプリケーションを動かす時に生じる，物理 CPU の不足や並列アプリケーションの性能が割り当てる CPU の減少以上に低下する問題の改善を試みた研究として pCPU-Est [10] がある．物理 CPU が不足する状況として，複数の VM 間での物理 CPU の共有，VM が利用できる CPU 使用率の制限，

VM に割り当てる物理 CPU を減らす，が想定されている．この研究では実行時情報に基づいて VM の仮想 CPU 数を動的に最適化することで並列アプリケーションの性能を改善する手法が提案されている．

9 まとめ

並列言語実装技法の 1 つであるビジーウェイトを並列言語とその処理系である Tascell におけるワークスティールの返信待ちに適用したところその有効性を確認した．同様に仮想環境上で VM 数を変動させながら評価を行ったところ，VM 数が 1 の場合を除いてビジーウェイトを適用した場合の方が既存の手法に比べて実行時間が長くなる結果となった．ビジーウェイトの適用によって仮想環境での実行時間が長くなることを少しでも回避するため，PAUSE 命令をビジーウェイトの繰り返しに用いて，PLE によってハイパーバイザに exit させる場合の評価を行ってみたところ，単にビジーウェイトする場合に比べて実行時間が短縮されたが，既存の pthread_cond_wait する場合に比べると実行時間は長くなった．

今後の課題として，PAUSE 命令に関する調査では共通の PLE の設定での評価をいくつか増やす必要がある．現在の PAUSE 命令により exit する仕様では exit 先によって PAUSE 命令による恩恵が得られにくい場合がある．そのため，exit した仮想 CPU の優先度をしばらく下げて exit 先として選ばれにくくするような設定項目があれば，より PAUSE 命令を有効に使えるのではないかと考える．

謝辞 本研究の一部は JSPS 科研費 JP19H04087 および JP17K00099 の助成を受けたものである

参 考 文 献

[1] : Xen Project Hypervisor, https://wiki.xenproject.org/wiki/.

[2] : インテル Pentium4 プロセッサおよびインテル Xeon プロセッサにおけるスピン・ループの使用, https://www.intel.co.jp/content/dam/www/public/ijkk/jp/ja/documents/developer/w_spinlock_j.pdf.

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

[4] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T.: Backtracking-based Load Balancing, New York, NY, USA, ACM, 2009, pp. 55–64.

[5] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, 2016.

[6] 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.

[7] 平石拓, 河野卓矢, 八杉昌宏, 馬谷誠二, 湯淺太一: バックトラックに基づく負荷分散の高並列環境における評価, 情報処理学会研究報告. *[ハイパフォーマンスコンピューティング]*, Vol. 2010, No. 25(2010), pp. 1–11.

[8] 平石拓, 八杉昌宏, 馬谷誠二: 動的負荷分散フレームワーク Tascell の広域分散およびメニーコア環境における評価, 先進的計算基盤システムシンポジウム論文集, Vol. 2011, May 2011, pp. 55–63.

[9] 良本海, 八杉昌宏, 平石拓, 馬谷誠二: 仮想環境を考慮した要求駆動型負荷分散, 日本ソフトウェア科学会第 *34* 回大会講演論文集, September 2017.

[10] 高山都句子, 光来健一: VM が利用可能な CPU 数の変化に対応した並列アプ リケーション実行の最適化, 日本ソフトウェア科学会第 33 回大会講演論文集, September 2016.