Vol. 40 No. SIG 1 (PRO 2) 情報処理学会論文誌 Feb. 1999

明示的なタスク配置指定が可能な遅延タスク生成に基づく動的負荷分散方法

外山 純生† 大山 恵弘†
田浦 健次朗† 米澤 明憲†

本論文では，遅延タスク生成（Lazy Task Creation，以下 LTC）による動的負荷分散を行う言語上に，明示的に計算位置を指定できる機構を導入する．LTC は共有メモリ並列計算機上の，効率的なタスクスケジューラとして知られており，主な利点は空間使用量が小さいこと，および，オーバヘッドが少ないことである．しかし，メモリアクセスコストが一様でない場合，しばしば，データの配置によってタスクの配置を決める必要が生ずる．このような環境では，任意の遊休状態のプロセッサが任意のスレッドを実行する単純な LTC 方式では不十分である．この論文では，明示的なタスク位置指定プリミティブを，LTC に基くスケジューラに導入する際の問題点を述べ，LTC の利点を失わずに，それを実装する方法を示す．我々の実装では，プロセッサがタスク位置を指定したタスク生成を行ったときに，生成されたタスクだけでなく，そのプロセッサ内のタスクスタック中の全てのタスクを，指定されたプロセッサに移動する．指定されたプロセッサは移動したタスクを直ちに実行し，そのプロセッサ内のタスクスタック中のタスクを，全て元のプロセッサに返す．この**スタック交換**がどのように LTC の利点を保ったまま，明示的なタスク配置の合理的なセマンティクスを実現しているかを示す．我々は，明示的なタスク位置指定プリミティブを，並列オブジェクト指向言語 Schematic に実装し，分散共有メモリ計算機 Origin 2000 にて性能評価を行った．その結果 (1) そのプリミティブを加えたことによるオーバヘッドは 2–10%に抑えられ，(2) 2 次キャッシュに入りきらない程巨大なデータをアクセスするプログラムにおいて，15–18%の速度向上がみられた．

Enabling Explicit Task Placement on Lazy Task Creation

Sumio Toyama,† Yoshihiro Oyama,† Kenjiro Taura†
and Akinori Yonezawa†

This paper introduces an explicit task placement primitive to dynamic load balancing method based on Lazy Task Creation (LTC). LTC is known as an efficient task scheduler on shared-memory parallel computer, whose main advantages are its space efficiency and small overhead. When memory access cost is not uniform, however, we often need to place tasks according to data locations. On such environments, the simple LTC in which any idle processor steals any task will not work very well. In this paper, we discuss issues in introducing an explicit task placement primitive into LTC-based scheduler and show how to implement it without losing the benefits of LTC. In our implementation, when a processor encounters a task creation with an explicit placement, the processor not only migrates the thread just created, but also migrates *all* the tasks in its task stack to the target processor. The target processor immediately continues the migrated task and, in return, gives all the tasks in its task stack to the source processor. We show how does this "stack swapping" operation preserves all the benefits of LTC, while implementing a reasonable semantics of the explicit task placement primitive. We implemented the primitive in concurrent object-oriented language Schematic and evaluated it on Origin 2000, a distributed shared-memory computer. Experimental results indicate that (1) the primitive adds a small (2–10%) overhead to programs that do not use it, and (2) it improves programs that accesses large data that do not fit in the second-level cache by 15–18%.

1. はじめに

遅延タスク生成[6)]（Lazy Task Creation，以下 LTC と略す）は，共有メモリ並列計算機における，効率的な動的負荷分散の実装として知られており．具体的に

† 東京大学大学院理学系研究科 情報科学専攻
Department of Information Science, Graduate School of Science, University of Tokyo

は次のような利点が挙げられる．

貪欲なスケジューリング LTC では，実行可能な仕事が PE（Processing Element）の数以下の場合には必ず実行される．そのため，実行可能な仕事があるにもかかわらず，遊休状態の PE が出現することがなく，効率的で均等な負荷分散をすることができる．

低廉なオーバヘッド 仕事が無くなった PE が出現した時になって初めて，負荷分散用の計算が生成される．そのため，仕事が移動する回数が少なく，仕事の生成や受け渡しのオーバヘッドが少ないことが知られている．

少ない必要な記憶領域 スレッドの生成者を「親」とし，生成されたスレッドを「子」として，計算を木構造で表すと，深さ優先に実行される．これにより，非常に沢山のスレッドを生成しても，必要な記憶領域が劇的に増大しないことを保証している．

一方，Origin 2000 のように，共有メモリ機構を備えてはいるが，リモートメモリとローカルメモリのアクセスコストが異なる環境も存在する．このような環境では，データの配置によって，実行方式に応じてタスクの位置を変えられる機構が必要である．

プログラマが明示的に計算位置を指定できる機構（以降「計算位置指定機構」と呼ぶ）を，LTC による動的負荷分散を採用している言語に，単純に導入しようとすると，次のような問題が発生し，先に述べた LTC の利点が失われてしまう．

不均等な負荷分散 計算位置を指定されたスレッドは，指定された PE でしか実行できない．すると，実行可能な仕事が位置指定されたものしか無い場合に，遊休状態の PE があっても，それらの仕事を実行できない事態が生ずる．このため，LTC の利点である均等な負荷分散が出来なくなり，性能低下を招く恐れがある．

オーバヘッドの増大 単純な計算位置指定機構の実装では，計算位置を指定されたスレッドが生成される度に仕事に移動が発生してしまう．これにより，LTC の「仕事の移動回数が少ない」という特徴が失なわれてしまい，計算位置を指定されたスレッドが沢山生成される場合には，オーバヘッドが増大する．

必要な記憶領域の劇的な増大 2) では，実行中の計算を木構造で表したときに，常に「全ての葉のスレッドには PE が割り当てられている」という性質（busy-leaves property）が成り立つ時に，P 個の PE で並列実行した時に必要な記憶容量 S_P は，S_1 を同じプログラムを逐次実行した時に必要な記憶容量☆とすると，$S_1 P$ 以下となることが証明されている．ところが，単純な実装では，計算位置を指定されたスレッドが生成された時に，既に指定された PE に仕事がある場合には，そのスレッドが「葉」であったとしても，指定された PE が遊休状態になるまで封鎖されてしまい，busy-leaves property を満さなくなる．そのため，スレッド管理のための記憶領域が劇的に増大する恐れがある．

本稿では，計算位置が指定されたスレッドの生成時に，依頼主の PE と依頼先の PE のスタックを入れ換えることにより，LTC の特徴を維持しつつ，計算位置指定機構を導入する方法（これを**スタック交換方式**と名付ける）を示す．このようにスタックを入れ換えることによって，依頼主と依頼先が，両方とも「葉」のスレッドを実行している時にも，どちらのスレッドも封鎖されることなく実行されるので，busy-leaves property を満たすことができる．また，我々の実装では，計算位置指定機構は，プログラマによる処理系のヒントという位置付けなので，プログラマがむやみに計算位置を指定しても効率の良い実行ができるように考慮されている．具体的には，

- 計算位置が指定されたスレッドでも，粒度の小さいものは，指定は無視され，同じ PE のまま実行される．
- 遊休状態の PE は，計算位置の指定されたスレッドで，かつ，まだ実行されていないスレッドを盗んで実行することもある

という実装になっているので，オーバヘッドの増大や，不均等な負荷分散を防ぐことができる．

以下，2 章では LTC の負荷分散の方法について述べ，3 章で単純な計算位置指定機構の実装方法と，そのような実装をした時に生ずる問題点について詳しく述べる．さらに，4 章で我々のスタック交換方式の実装について述べ，5 章で，Schematic 上での実験結果から，我々の実装の有効性について検証する．6 章で関連する研究について触れ，最後に 7 章で結論と今後の課題を述べる．

2. Lazy Task Creation

Lazy Task Creation（LTC）は，効率的なマルチスレッディングの実現方式[6]である．本章では，並

☆ 実行の各瞬間における，開始されて終了していないスレッドが必要な記憶容量の最大値．詳しくは 2) を参照．

```
(define (sum-up tree)
  (if (leaf? tree)
      (leaf-value tree)
      (let ((c (future (sum-up (left tree)))))
        (+ (touch c) (sum-up (right tree))))))
```

図 1 二分木の葉の総和を並列に求める Schematic プログラム
Fig. 1 Schematic program to sum the leaves of a binary tree in parallel.

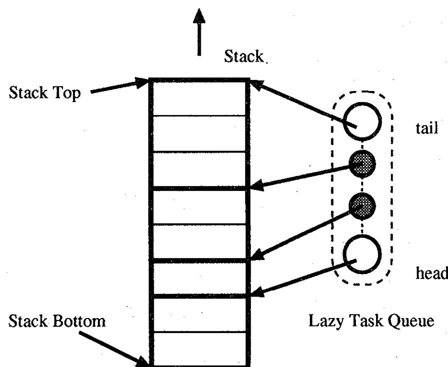


図 2 Lazy Task Queue の構造
Fig. 2 Structure of Lazy Task Queue.

列オブジェクト指向言語 Schematic を例に，LTC の実装について説明し，最後に LTC の特徴について述べる．

2.1 Lazy Task Creation と Schematic

並列オブジェクト指向言語 Schematic[8] は，共有メモリ並列計算機 Sun Ultra Enterprise 10000，および分散共有メモリ並列計算機 Origin 2000 上に実装されており，負荷分散の方式として LTC を採用している [7]．Schematic では並列プリミティブとして，**future** と **touch** が用意されている．**図 1** は，Schematic における，並列に二分木の葉の総和を求めるプログラム例である．

(**future** *E*) は，*E* を非同期に実行し，**チャネル**を返値とする．チャネルには，*E* の実行が終わると，その返値が格納される．

(**touch** *c*) は，チャネル *c* から値を得ようとする．もしも，まだチャネルに値が格納されていない場合は，値が格納されるまで封鎖される．

LTC では，各 PE に Lazy Task Queue（LTQ）というデータ構造を持たせて，それによって仕事を管理する（**図 2**）．LTQ は，自 PE からはスタックとして扱われ，新しく関数が呼ばれた際に，次のように動作する．

通常の関数呼び出し　（future ではない）通常の関数呼び出しが行われると，その関数の継続がスタックに積まれる．図 2 では，このようにして積まれた継続のスタックの境界は細い実線で描かれている．

future 呼び出し　future を使った関数呼び出しが発生した場合，その PE は future の本体を実行し，future の継続をスタックに積む．図 2 では，このようにして積まれた継続のスタックの境界は，太い実線で描かれている．

一方，LTQ は，他の PE からはキューとして扱われ，キューの先頭はスタックの底を指し，末尾はスタックの先頭を指す．なお，図 2 で示されているように，キューの各要素は，図の太い実線を単位にして格納されている．

遊休状態にある PE は，適当な PE に仕事を渡すように要求し，要求が受理されると，その PE の LTQ の先頭（スタックの底）から仕事を 1 つ受け取って実行する．この時に仕事が取り出される単位は図 2 で太い実線に囲まれた部分である．

すなわち，future 呼び出しと，通常の関数呼び出しは，「そこを境界として仕事を盗めるか」という点を除いては，全く変わらない．

2.2 LTC の特徴

2.2.1 貪欲なスケジューリング

LTC では，実行可能な仕事がある限り，遊休状態の PE は必ず仕事を盗んで実行する．すなわち，実行可能な仕事が n 個，PE の数が P 個だとすると，

(1)　$n > P$ なら P 個
(2)　$n \leq P$ なら n 個

の仕事が，常に PE に割り当てられている，という性質がある．したがって，最適な負荷分散方法についてプログラマが熟知している必要はなく，並列性を抽出したい所に全て future を付けておけば，LTC が均等に負荷分散を行なってくれ，結果的に効率の良い実行ができる．

2.2.2 低廉なオーバヘッド

図 1 の(sum-up *tree*) を 4PE で実行した時の様子は，**図 3** のようになる．この図において，各ノードはひとつの sum-up 関数を表しており，future によっ

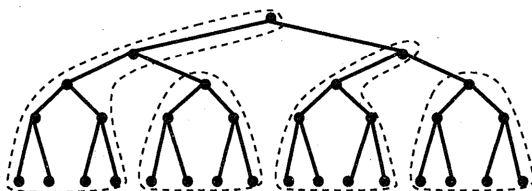


図 3 LTC における(sum-up *tree*) の 4PE での実行状態
Fig. 3 LTC execution of (sum-up *tree*) on four PEs.

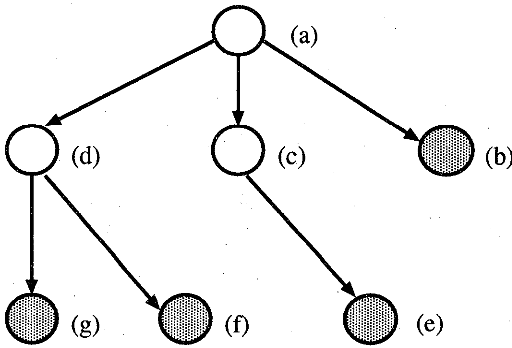


図 4 計算の木構造の例
Fig. 4 Example of tree structure of execution.

て生成された仕事が左の子で，通常の関数呼び出しによって生成された仕事が右の子である．また，破線で囲まれた部分が1つのPEによって実行されている．

2.1 節で説明したように，**sum-up** の実行が始まり，**future** 呼び出しが行われると，**future** の本体は自PEで実行され，**future** の継続は幅優先に他のPEに盗まれ，最終的に4つのPEが全て，**sum-up** を実行している状態になる．

この状態になると，遊休状態のPEが現れない限り，**future** によってスタックに積まれた継続は他のPEが盗まなくなり，結局自PEが継続を実行するようになる．LTC では，継続が実際に盗まれる時になってはじめて負荷分散のための計算が生成されるので，この状態になってからは，逐次実行したときとほぼ同じオーバヘッドしかかからなくなり，効率の良い実行ができる．

2.2.3 合理的なメモリ使用量

(**sum-up** *tree*) のようなプログラムは，計算が終了するまでに，*tree* の深さ d に対して指数関数的な回数の**future** 呼び出しが発生するので，**future** 呼び出しの度に，**future** の本体を実行せずに継続を先に実行するようなスケジューリングを行なうと，スレッド管理に必要なメモリ領域が足りなくなってしまう．LTC ではそのようなことは起こらないのだが，2) では，必要なメモリ領域が，ある基準以下に抑えられるための本質的な条件を明らかにしている．

この条件について説明するために，今迄にも出てきた計算の木構造について，もう少し詳しく述べる．図4は，例えば次のようなプログラム☆

```
(define (f)
  (let* ((c1 (future (f)))
         (c2 (future (f)))
         (c3 (future (f))))
    (touch c1)
    (touch c2)
    (touch c3)))
```

を実行したときの，実行中のある瞬間の図である．この図において，すべてのノードは関数**(f)** を表しており，**future** によって**(f)** を呼ぶと，子供が生成される．すなわち，(a) は**future** によって，(b)・(c)・(d) を生成したことになる．

このような図において，**leaf** を「まだ子供を産んでいないノード」と定義する．すなわち，図4においては，灰色で塗り潰されているノードが leaf となる．なお，これはあくまでも実行中のある瞬間の話であるので，ある瞬間 leaf だったものが，子供を産んで leaf でなくなることは有り得ることに注意されたい．さらに，

全てのleaf に，常にプロセッサが割り当てられている

という性質を**busy-leaves property** と定義する．

以上の定義のもとで，2) では，動的負荷分散のアルゴリズムが，busy-leaves property を満たすならば，

$S_P \leq PS_1$

が成り立つことが証明されている．ここで，P はPEの数を表し，S_n は，n 台のPEで実行した時に，動的負荷分散に必要な記憶容量を表している．すなわち，busy-leaves property を満たす負荷分散アルゴリズムは，最大でも，そのプログラムを逐次に動かすのに必要な記憶容量のPE台数倍の記憶容量しか必要としない，ということである．

この busy-leaves property は，LTC に基いていても，全てのプログラムについて満たされるものではないが，

あらゆる同期（封鎖を伴う同期）は祖先が子の終了を待つ同期にかぎられる

という制限（**strict computation**）を満たしたプログラムなら，busy-leaves property を満たすことが知られている [2]．なお，この制限は，Schematic では，**future** の返値のチャネルを，あらゆる関数全体の返値にできるが，他の関数呼び出しの引数として渡すことはできない，ということに相当する☆☆．

☆ これは，あくまでも図4の説明のためのものであり，無限に再帰し続けるだけの，意味のないプログラムである．

☆☆ Schematic では，**future** の返値であるチャネルは first-class なので，この制限を破ったプログラミングも可能である．

3. 単純な計算位置指定機構の実装と問題点

ここでは，我々が仮定している計算位置指定機構の「単純な」実装の詳細を述べたあとに，この実装の問題点について考察する．なお，この単純な方法も，4章で述べる方法と共に，Schematic に実装されている．

3.1 実装の詳細

まず，新たに，各 PE に位置指定された計算を格納しておくためのキュー（ここでは**future** キューと呼ぶ）を用意しておく．

各 PE は，自分のスタックが空になり，仕事が無くなると，自分の future キューが空かどうか調べ，空ではなかったら，そこから取り出して実行する．もしも future キューが空だった場合は，従来の LTC と同様に，他の PE へ負荷分散要求を出す．

次に，計算位置指定機構の実装のため，**future** を拡張した

(**future** *E* :on *loc*)

という式を導入する．この式は，2章で説明した:on の無い**future** と同様に，*E* を非同期に実行して，*E* の返値が格納されるチャネルを全体の返値として持つが，*E* を PE 番号 *loc* で実行するように指示する．

(**future** *E* :on *loc*) は

(1) *E* の関数閉包
(2) 返値が格納されるべきチャネル
(3) *E* に渡す引数群

をデータ構造に格納し，*loc* に指定された PE の future キューへ投入する．**future** を実行した PE は，キューへの投入が終わると (**future** ... :on ...) の継続を実行する．

3.2 問題点

3.2.1 厳格過ぎる:on のセマンティクス

ここで述べた実装と:on のセマンティクスでは，:on を付けた式は，必ず指定された PE で実行される．一見すると，このセマンティクスはプログラマにとって単純明快に思えるが，プログラマが「どの程度まで:on を付けると速くなるのか」熟知した上で，最適な量だけ:on を付けないと，次に挙げるような理由で，性能が悪化する恐れがある．

不均等な負荷分散 このセマンティクスだと:on によって生成されたスレッドは絶対に他の PE によって実行されない．そのため，1つの PE へ位置を指定したスレッド生成を集中させると，本来は負荷分散できる場合でも，残りの PE は遊休状態のまま待たされることになり，性能低下につながってしまう．

```
(define (sum-up tree)
  (if (leaf? tree)
      (leaf-value tree)
      (let* ((l (left tree))
             (c (future (sum-up l)
                        :on (whereis l))))
        (+ (touch c) (sum-up (right tree))))))
```

図 5 (future ... :on ...) を使った，二分木の葉の総和を求める Schematic プログラム
Fig. 5 Schematic program to sum the leaves of a binary tree by using (future ... :on ...) expression.

オーバヘッドの増大 非常に小さい粒度のスレッドに:on を付けると，:on による性能向上が，スレッド移動（future キューへの出し入れ）のオーバヘッドで隠蔽されてしまい，逆に性能低下につながる恐れがある．

これらは，プログラマが，:on を用いたスレッドを均等に負荷分散し，かつ，粒度の大きなスレッドにしか:on を付けないようにすれば回避できる問題であるが，プログラマに多大な負担を要求することになる．

3.2.2 空間使用効率の悪化

計算位置指定機構の単純な実装で特に問題なのは，計算が (**future** ... :on ...) に到達したときに，**future** が実行されずにキューへ投入され，かつ，**投入された仕事がすぐに実行されるとは限らない点**である．:on の無い**future** の場合は，その PE で**future** の本体が実行され，**future** の継続の方が，即座には実行されずにスタックへ積まれるのであった．

このことに注意して，**図 5** がどのように実行されるのか見てみよう．なお，(**whereis** *l*) は，*l* が確保されている PE 番号を返すプリミティブである．

sum-up は，**future** 呼び出しの度にスレッドを生成して，future キューに投入し，自分はそのまま継続の実行を続けることになるので，**図 6** のような実行状態になる．なお，各ノードに付いている数字は，その時点での木の深さであり破線部に付いている斜体の値は，その破線部で，何個の**future** 呼び出しがあるかを表している．

仮に，4つの PE で，この関数が実行されて，淡い色の枠で囲まれている部分の計算が PE に割り当てられたとしよう．このとき，割り当てられた PE は，通常の関数呼び出しによってノードを右へ1つ下る度に**future** で子供を生成し，それを葉に達するまで繰り返すことになる．すなわち，(**sum-up** *tree*) の呼び出しで，最初に割り当てらた4つの PE が生成する子供の数を $c(n)$ とすると，

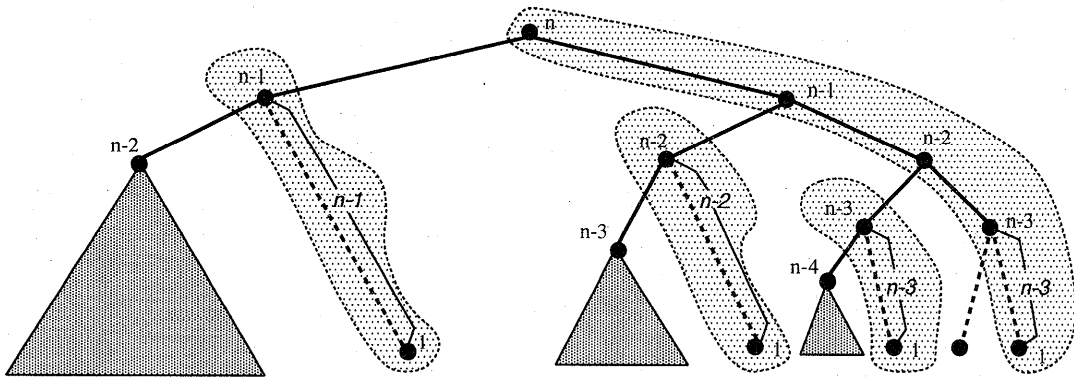


図 6 位置指定機構を単純に実装した時の(sum-up *tree*) の実行の様子
Fig. 6 Execution of (sum-up *tree*) by a trivial implementation.

$$c(n) = (n-1) + (n-2) + (n-3) + (n-3) = 4n-9$$

となり，$c(n)$ 個の子供は，全て実行されずに future キューに投入されることになる．$c(n)$ 個の子供が生成されてから，4 つの PE が遊休状態になると，最悪の場合，深さ $n-2$ のノードにおける**sum-up**が実行される代わりに future キューから 4 つの仕事が取り出されるので，延べ $c(n)-3+c(n-2)=8n-29$ 個の仕事が future キューに貯まることになる．これは，例えば，深さ 30 の木に対して**sum-up** を 4 つの PE で実行した時に，8 つのスレッドの計算が終了するまでに，211 個のスレッドが future キューに眠っていることを表している．実際，10PE で深さ 30 の木を**sum-up** すると，図 1 のコードでは，すぐに正常終了するが，図 5 のコードでは，各 PE に 10MB のヒープを確保しても，メモリが溢れて，計算を終えることができない．

次に，この実装方式を busy-leaves property の観点で考察してみる．ある時点で，計算が(**future** *E* **:on** ...) に到達した時に，*E* は必ず leaf となるので，busy-leaves property を満たすためには，*E* はすぐに実行されなくてはならない．しかし，指定された PE が既に忙しい場合には *E* は実行されずに封鎖されてしまうので，busy-leaves property は満たされない．

4. スタック交換方式の実装

前章で，単純な方法での計算位置指定機構の実装では，いくつかの問題があることを述べた．本章では，これらの問題を解決する我々の実装を説明する．

4.1 新しい:on のセマンティクス

3.2.1 節で指摘したように，厳格過ぎる:on のセマンティクスにはいくつか問題がある．そこで:on を，あくまでも計算位置のヒントとして位置づけ，セマンティクスを次のように柔軟なものに変更する．

- 実行時に，同じ計算位置を指定された関数が，同時に 2 つ以上実行されている場合は，必ず 1 つの関数は，指定通りの PE で実行され，残りの関数は，他の PE で実行される．
- ある PE *loc* で，位置指定されている関数の実行が終了したときに，他のいくつかの PE で，*loc* で計算するように指示されている関数が実行されているならば，その内の 1 つは，指定通りの PE（ここでは *loc*）で，実行の続きが行われる．

簡潔に言うと，原則として，実行の終始にわたって，

> 全ての PE *loc* に対して，*loc* で実行するように指定されている計算が複数存在している場合に，そのうちの 1 つは，必ず *loc* で実行されている．

という性質が，常に満たされているということである．ただし，以下の場合は，この性質よりも以下の規則が優先される．

- ある閾値以下の時間で終わってしまう関数は，**:on** を付けても無視される．
- 遊休状態の PE がいる場合には，位置の指定がなされていても，その遊休状態の PE で実行されることがある．

このようにすることで，非常に短い時間で実行が終わる関数は，仕事の移動のオーバヘッドなしに，(**:on** は無視されたまま）実行されるので，オーバヘッドを軽減することができ，負荷も均等に分散できる．しかし，分散共有メモリ計算機において，あるオブジェクトを必ず特定の PE で作りたい時など，**:on** が無視されては困る状況も存在する．この問題については，「そのプリミティブ以降は必ず指定された PE で実行せよ」という意味の(**move**) プリミティブを用意し，プログラマに指定させることで回避している．

4.2 スタック交換方式のアイディア

3.2 節で，busy-leaves property が，単純な実装では満たされない原因は，計算が`(future` E `:on` *loc*`)` に到達した時に，*loc* の PE が忙しい場合には，leaf である E が実行されずに封鎖されてしまうからだと述べた．つまり，単純な実装を，busy-leaves property を満たすように変更するには，E を実行するスレッドが生成されたら，すぐにその実行を始めるようなスケジューリングを行えば良い．しかし，E のスレッドが生成された瞬間に，*loc* の PE が，今計算している仕事 E' を封鎖して E を実行することにしても，何の解決にもならない．何故なら，E' も leaf であった場合，その計算が封鎖されてはいけないからである．

そこで，我々は，常に E も，E' も封鎖されずに実行させるために，`(future` E `:on` *loc*`)` に到達したら，E を実行している PE のスタックと，E' を実行している PE のスタックを**入れ換える**ことで，お互いの仕事を交換する方法を示す．

このようにすることで,「実行されているノード」の状態は全く変更せずに,「ノードを実行している PE」だけを変更できるので，busy-leaves property を満たすことができ，strict computation のもとでは，必要な記憶領域を PS_1 以下に抑えることができる．

4.3 実装の詳細

4.1 節で説明したセマンティクスを満たすためには，

今実行している関数は，本来どの PE で実行してほしい関数なのか

を，実行時に管理する必要がある．そこで，Schematic コンパイラは，暗黙のうちに，関数引数に「PE 番号」を追加する.「PE 番号」には，実際に PE に振られている番号の他に “any” という値が渡され得る．引数として番号が渡された時は，4.1 節のセマンティクスにしたがって，その関数を渡された PE 番号で実行し，“any” が渡された時は，従来の LTC の方式に基いて，適当な PE に負荷分散される．実際に「PE 番号」として，関数に渡す引数は，以下のような規則で決定される．

- 一番最初に呼ばれる関数には “any” が渡される．
- `(future` E `:on` *loc*`)` の E を呼ぶときには，*loc* が渡される．
- 上記以外の関数を呼ぶ時には，呼ぶ関数が「PE 番号」として渡された引数を，そのまま呼ばれる関数に渡す．

この規則の意味するところは，`(future` E `:on` *loc*`)` が呼ばれない限り，すべての関数呼び出しには “any” が渡され，一旦`(future` E `:on` *loc*`)` が呼ばれると，E の中で呼ばれる関数には，E の内部で`(future ... :on ...)` を呼ばない限り,（通常の関数呼び出しか，`:on` の無い`future` 呼び出しかに関係なく）*loc* が渡されるということである．

このようにして渡された「PE 番号」は，関数閉包から取り出せるようになっているので，スタックに積まれている関数閉包から，実行すべき PE 番号を得ることができる．

さて，各 PE は，スタックから関数閉包を取り出して実行する時に，取り出した関数閉包の「PE 番号」を見，それが自分以外の（“any” を除く）PE を表していたら，定期的に，**スタック交換要求**を，取り出した「PE 番号」に対して発行する．

一方，スタック交換要求に応えるため，各 PE は，やはり定期的に自分にスタック交換要求が来ているか調べている．もしも自分に交換要求が来ていた場合は，自分が今実行している関数閉包の「PE 番号」を調べ，それが自分以外の PE を表していたら，要求を受け入れ，スタックを交換する．そうでは無い場合は，自分も位置指定された計算をしていることになるので，交換要求は単に無視する．

このように，交換要求やその確認のために，定期的にポーリングを行うことによって,「常に`:on` で指定された関数のどれかは，その PE で実行されている」という，4.1 節で述べたセマンティクスを満たすことができる☆．

なお，スタックの交換は，交換したい PE 同士が，スタックポインタなど，いくつかのレジスタの値を交換するだけで実現できる．具体的に交換する情報は，

(1) 関数閉包
(2) 各 PE が持つスタックポインタ
(3) 返値が格納されるべきチャネル
(4) その関数閉包が必要とする引数群

と，(2) 以外は 3.1 節で説明した，`(future ... :on ...)` の実行の際に future キューに格納する情報と同じである．

この方式により，3.2 節で指摘した問題が解決でき，オーバヘッドが少なく，空間使用効率の良い計算位置指定機構が実現できる．ただ，この方式では，仕事の移動時にスタックを交換しているため，ヒープ領域がローカルになる代わりに，スタック領域はリモートメモリになってしまうという欠点がある．この欠点が，どの程度性能に影響するかについては，まだ詳細な検

☆ ここまで，ポーリングの間隔を「定期的」と表現していたが，実際には新しい関数閉包に入る度に，その先頭でポーリングを行っている．

討をしていないが，スタック領域はヒープ領域と違って，（仕事の受け渡しなどの特殊な場合を除いては）1つのPEからしかアクセスされないので，比較的キャッシュにヒットしやすく，影響は少ないと考えている．

5. 実　　験

4章で，我々の実装について詳細を述べ，その実装で，LTCと同じ空間使用効率が実現できることを述べたが，計算位置指定のためのオーバヘッドが，指定することによって得られる速度向上よりも大きい場合は意味がない．ここで期待している速度向上は「リモートメモリアクセスよりもローカルメモリアクセスの方が速い」ことを前提にしているが，リモートメモリへのアクセスでも，実際にはキャッシュの影響によって，ローカルメモリへのアクセスよりも高速なことがある．

そこで，スタック交換方式による計算位置指定機構の有効性を検証するため，次の2点に関して実験を行なう．

まず1つめは，アクセスされるデータサイズと速度向上の関係を計測する実験である．アクセスされるデータサイズがキャッシュの容量を大幅に上回る場合には，キャッシュによるリモートアクセスの高速化の効果は殆ど期待できない．一方，データサイズがキャッシュに全て収まってしまう場合には，リモートアクセス遅延は殆ど問題にならないであろう．この実験では，単に大きなオブジェクトを作って読み込むだけといった単純なテストプログラムを用いて計測する（詳細は後で述べる）．

2つめの実験では，現実に存在するようなアクセスパターンで，我々の方式による位置指定の効果がどれほどになるかを検証するため，実用的なアプリケーション（ここではN体問題を解くプログラムを用いる．詳細は後述.）に **:on** を付けた場合と付けなかった場合の時間を計測する．

以降，本章では，まず実験に用いた計算機環境を述べたあとに，2つの実験について詳細を述べる．

5.1 計算機環境

計算機環境としては，分散共有メモリ並列計算機である Origin 2000（MIPS R10000 195MHz × 128PE, 18GB Main Memory, 32KB L1 data cache, 4MB L2 cache）のうち，30個のPEを用いて実験した．Origin 2000は，ハードウェアで分散共有メモリを実現しているが，ローカルメモリとリモートメモリでアクセス時間に差があるのが特徴である．**表1**に，5)から抜粋した，メモリやキャッシュへのアクセス遅延を挙げる．この表によると4つを超えるPEがある場合は，平均してローカルメモリの2倍程度，リモートメモリへのアクセスに時間が掛かることが分かる．

表1 Origin 2000 のアクセス遅延
Table 1 Origin 2000 latencies.

Memory level	Latency (ns)
L1 cache	5.1
L2 cache	56.4
local memory	310
4P remote memory	540
8P avg. remote memory	707
16P avg. remote memory	726
32P avg. remote memory	773
64P avg. remote memory	867
128P avg. remote memory	945

また，各ノードにおけるメモリへのバンド幅は 0.78GB/sec[5] であり，ノード間の二分（bisection）バンド幅は128PEの場合で20GB/secである．

計算位置指定機構の実装は，共有メモリを前提に実装されている並列オブジェクト指向言語Schematicに対して行った．

5.2 データサイズと速度向上の関係

5.2.1 プログラムの概略

次の2つのプログラムを用意して実験を行った．

ReadVector ある要素数のvector（4Bytes/要素）をPEと同じ数（30個）用意して，同数（30個）のスレッドがそれぞれのvectorの全要素を読むプログラム．1PEに持たせるvectorの要素数を，500,000要素（約2MB・2次キャッシュの半分）から4,000,000要素（約16MB・2次キャッシュの4倍）まで，500,000要素間隔で変化させて測定を行う．

BinTree あるノード数（32Bytes/ノード）の二分木をPEと同数用意して，同数のスレッドが，それぞれの二分木の全ノードの値の総和を求めるプログラム．1PEに持たせるノード数を，100,000ノード（約3.2MB）から500,000（約16MB）まで，50,000ノード間隔で変化させて測定を行う．

さらに，それぞれのプログラムについて，次の3つの計測条件を用意した．

交換なし 計算位置指定機構（スタック交換）の実装がされていない処理系で実行する．計算位置の指定はできないため，30個のオブジェクト（vectorや二分木）がどのPEに作られるかも，それらのオブジェクトを読むPEがどれかも不定である．

交換あり指定なし 計算位置指定機構が実装されている処理系で実行するが，プログラムには計算位置の指定を使っていない．従って，「交換なし」と挙

動は同じになるが，処理系に計算位置指定機構が実装されているため,「交換なし」に比べて，余計なポーリングなどを行っている.

交換あり指定あり　計算位置指定機構が実装されている処理系で実行し，かつ，プログラムにも計算位置の指定を行なっている．この場合は，30 個のオブジェクトはそれぞれ別々の PE で作られ，それらのオブジェクトを読む PE は，オブジェクトが作られた PE となる．従って，オブジェクトを読む部分のメモリアクセスは，全てローカルメモリに対するアクセスになる.

これら 6 種類のプログラムで,「30 個のスレッドがオブジェクトを読んで全てが終了するまで」をそれぞれ 100 回繰り返し，その時間を計測した．最初にオブジェクトを作る部分は計測に入っていない．また,「全てが終了するまで」の時間を計測しているので，実行が一番遅いスレッドが，計測時間を決定することも指摘しておく.

したがって,「交換あり指定あり」の計測環境では，全てのスレッドがローカルメモリへのアクセスになるので，全てのスレッドのメモリアクセスが高速に行われ，一番遅いスレッドと一番速いスレッドの速度差は，殆どないと思われる．一方，それ以外の計測環境では，運悪く，遠い PE にあるリモートメモリをアクセスするスレッドが，一番遅いスレッドとなり，計測時間が伸びることになる.

しかし，実際には，リモートアクセスでも，キャッシュにヒットした場合は，小さい遅延で値を読むことができるので，上の考察はキャッシュの影響を無視した場合にしか適用できない．すなわち，キャッシュに入りきらないほど大きなオブジェクトを読む場合は，キャッシュの影響を無視できるため,「交換あり指定あり」が「交換なし」よりも高速になることが期待できるが，キャッシュに全て入ってしまうような小さいオブジェクトでは，スタック交換のためのオーバヘッドが顕在化して，両者の差は縮まるか，逆転するものと考えられる.

なお，ReadVector については，1 つのループの中で vector の複数要素を読むことで，ループのオーバヘッド軽減をはかっている.

5.2.2　計測結果と考察

それぞれのプログラムの実行結果を，ReadVector は図 **7** に，BinTree は図 **8** に示す．なお，それぞれの図について，縦軸は実行時間の比（「交換なし」に掛かった時間を 1 とする）を，横軸はオブジェクトサイズ（ベクトルの要素数/二分木のノード数）を表わしており，3 本の折れ線は先に述べた 3 つの計測条件を表している.

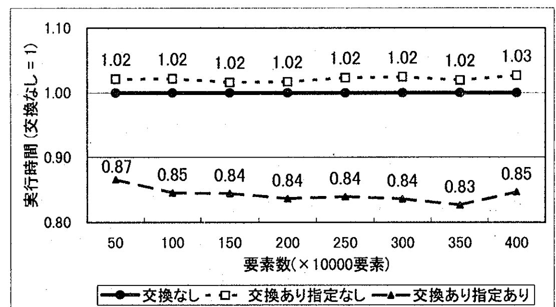


図 7　ReadVector の計測結果
Fig. 7　Elapsed time for ReadVector.

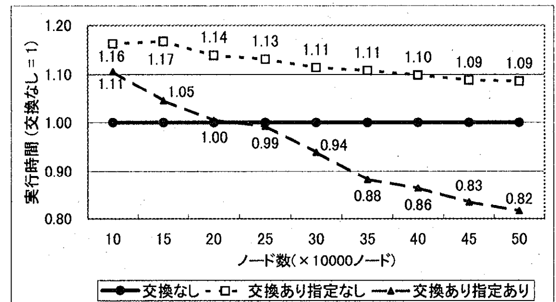


図 8　BinTree の計測結果
Fig. 8　Elapsed time for BinTree.

これらのグラフから，

(1)　2 つの図に共通して，ある程度オブジェクトサイズが大きところでは,「交換なし」よりも「交換あり指定なし」の性能が低下し,「交換あり指定あり」の性能が向上している.

(2)　BinTree では，オブジェクトサイズが小さくなるにつれて,（1）で述べた性能差が縮まっていき，2 次キャッシュの容量以下になると，ほとんど性能差が無くなるか,「交換あり指定あり」の方が「交換なし」よりも遅くなっている.

という特徴を読み取ることができる.

特徴（1）として挙げた「交換なし」に対する「交換あり指定なし」の性能低下はポーリングのオーバヘッドを示しており，どちらのプログラムの場合でも数%以下に抑えられている．また「交換あり指定あり」の性能向上は，プログラマが位置指定をすることにより，位置指定のできないものよりもどれだけ性能が向上するかを表している．この実験結果では，十分大きなオブジェクトサイズにおいては，ポーリングのオーバヘッドを打ち消し，さらに 15〜18%の性能向上が認められ，我々の方法が有効であることが分かる.

特徴（2）は，リモートオブジェクトがキャッシュに

のっている場合には，リモートオブジェクトへのアクセスが高速化されるため，位置指定機構の効果がほとんど無くなることを表している．すなわち，データサイズがキャッシュに収まる程度の大きさならば:onを付けることは無意味だということになる．しかし，そのような場合でも，位置を指定することによって，なにもしない場合よりも性能が著しく低下することは無い☆ことも結果から読み取れるので，プログラマは，リモートオブジェクトのサイズを気にすることなく，:onを付けておけば，性能向上の可能性を得ることができることが分かる．この特徴はBinTreeにしか表れていないが，何故ReadVectorで表れないのかはまだ分かっていない．

なお，表1によると，リモートメモリとローカルメモリのアクセス速度は2倍以上違うことになっているが，今回の実験における速度向上は，最高でも数十%に抑えられている．この原因としては，Schematicでは，メモリアクセスだけではなく，他の操作（スレッドの作成・終了待ちや，オブジェクトへのタグ付け・ポーリングなど）を行っていることが考えられる．

Cで，単純なループによるメモリアクセスを行った場合には，ローカルメモリを読む場合とリモートメモリを読む場合で，約2倍の差が出ることは確認済みである．

5.3 現実的なアプリケーションにおける効果

5.3.1 プログラムの概略

ここでは，現実的なアクセスパターンを持つアプリケーションとして，N体問題を解くプログラムを用いて実験を行う．アルゴリズムはBurnes-Hut法[1]を用い，八分木を生成するフェーズでは質点を並列に（木に）投入し，加速度も質点に関して並列に計算する．木を生成するフェーズでは，:onの無い`future`を用いて並列に行い，加速度の計算の際に必要な木のトラバースでは，:onを用いて，そのノードのあるPEへ移動して実行するようにプログラムを記述した．

このプログラムを，:onが無視される「交換なし」の処理系と:onが有効な「交換あり指定あり」の処理系のもとで実行し，その実行時間を計測した．以下の結果は，質点数を10,000として，1ステップ（全質点の加速度を求め，位置と速度を更新するまで）の時間を計測したものである．

5.3.2 計測結果と考察

プログラムの実行結果を図9に示す．ごみ集め（GC）

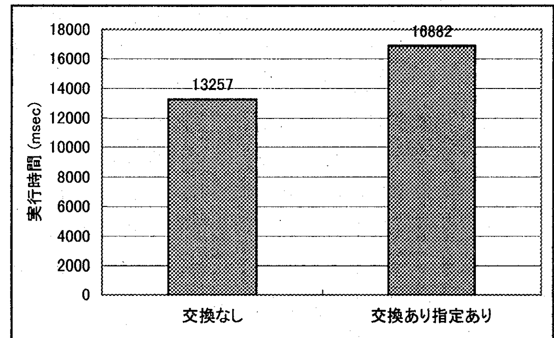


図9 N体問題の計測結果
Fig. 9 Elapsed time for N-body problem.

の時間は含まれていない．これによると「交換あり指定あり」の方が「交換なし」に比べて，27%ほど性能が低下してしまっている．

現時点では詳細な検討はしていないが，性能低下の主な原因は，:onされた計算の粒度が中途半端に小さいからであると考えている．細粒度な計算による性能低下に対しては，4.1節で触れたように，:onされた仕事を即座に移動せずに，ある閾値以下の時間でその仕事が終了しなかったときにはじめて仕事の移動をすることで解決を図っているが，この方法では「ある閾値以下の時間で終了しなかった仕事」は，それ以降も十分長い時間が掛かる仕事であると仮定している．したがって，その閾値よりも少しだけ長い時間がかかる仕事が多いときには，オーバヘッドが顕在化して，性能が悪化してしまう．

この問題は，閾値をどこに設定しても必ず起こる問題で，プログラムの傾向によって閾値を動的に設定するなど，他の方法で解決しなければならないと考えている．この問題の解決と，このプログラムでの性能低下の詳細な検討は我々の今後の課題である．

6. 関連研究

6)によって提案されたLazy Task Creation（LTC）は，オーバヘッドが低廉なことと，空間使用効率が良いことを特徴とする，細粒度スレッド生成を支援する動的負荷分散の方法である．本研究では，このLTCに，LTCの持つ利点を保存しつつ，計算位置指定機構を導入している．

これらのLTCの利点は，経験的にしか知られていなかったが，2)により，必要な記憶容量が劇的に大きくならないために，動的負荷分散アルゴリズムが満たすべき性質（busy-leaves property）が明らかにされた．我々は，この研究成果を踏まえて，計算位置指定機構を導入してもbusy-leaves propertyを保存する

☆「交換なし」よりも「交換あり指定あり」の方が遅くなっている部分についても，「交換あり指定なし」によって払うオーバヘッドよりは小さい性能低下なので，問題ないと考えている．

方法として，今回のスタック交換方式を実装した．

LTC に基いた動的負荷分散を採用している言語には，Cilk[2] や Gambit[3] がある．Cilk は,「スレッドを作った関数と，その祖先しか，そのスレッドの終了を待てない」という strict computation 以外の同期構造を許さないことで，必要な記憶容量の上限を保証している．Schematic はそのような制限が無いので，必要な記憶容量について，常に上限を保証することはできないが，Cilk と同じ同期構造しか使用しない場合は，Cilk と同様に，必要な記憶容量の上限が保証される．

Gambit は，Schematic と同様に，PE によってアクセスコストの異なる NUMA（Non-Uniform Memory Access）アーキテクチャの計算機上に実装されているが，我々のような計算位置を指定できる機構はない．

4) は，並列オブジェクト指向言語 ICC++ を Origin 2000 上に実装している．一つの分散共有メモリ空間を全 PE が直接読み書きする実行形式と，共有メモリ空間を PE 数に分割し，各 PE が自分に割り当てられたメモリ空間のみを直接読み書きし，他 PE のメモリの操作をメッセージ送受信で行う実行形式の性能を比較している．彼らは，ヒープ分割でワーキングセットが縮小することによるキャッシュ効率の向上に主に注目しており，我々と異なり，Origin 2000 の特徴であるメモリ局所性の効率への影響に関する考察はない．また，4) では，他 PE への仕事の依頼は，オブジェクトの位置情報をもとにランタイムがメッセージ送受信の形で行う．彼らの方式は本質的には本論文で述べた単純な計算位置指定機構と同じであり，仕事依頼用の記憶領域の拡大を抑える境界が存在しない．

7. 結論と今後の課題

我々は，LTC による動的負荷分散を行う並列言語上に，スレッド管理のための記憶領域が小さく抑えられる利点を保持しつつ，計算位置を指定できる機構を実現する方法として，位置指定スレッド生成時にスタックを交換する方式を示し，並列オブジェクト指向言語 Schematic に実装した．さらに，Origin 2000 での実験の結果，オブジェクトサイズが十分大きな場合では，我々の方式で位置指定実行したものは，従来の方式よりも 15〜18%高速化しており，我々の方式の有効性が確かめられた．

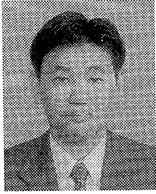
しかし N 体問題を解くプログラムでは 3 割程度の性能低下が見られた．この性能低下の原因の詳細な解明と，その解決は，我々の今後の課題である．また，今回の実験ではリモートメモリへのアクセス遅延が比較的小さい Origin 2000 を用いたが，もっとアクセス遅延の大きい計算機においても計測を行ない，我々の方式の有効性を検討したいと考えている．

謝辞 Origin 2000 の利用に際し，多大な便宜を図っていただいた，東京大学医科学研究所の中谷明弘氏に感謝致します．また，遠藤敏夫氏・山本泰宇氏をはじめとする，東京大学大学院理学系研究科の米澤研究室・小林研究室の方々には，様々な協力をしていただきました．

参考文献

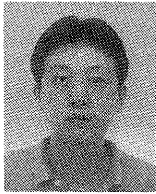
1) Barnes, J. and Hut, P.: A Hierarchical Force-Calculation Algorithm, *Nature*, Vol. 324, No. 6096, pp. 446–449 (1986).
2) Blumofe, R. D. and Leiserson, C. E.: Scheduling Multithreaded Computations by Work Stealing, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 356–368 (1994).
3) Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proceedings of Parallel Symbolic Computing: Languages, Systems, and Applications* (Halstead, R. and Ito, T.(eds.)), Lecture Notes in Computer Science, Vol. 748, Cambridge, MA, USA, Springer-Verlag, pp. 94–107 (1993).
4) Ganguly, B.: Concurrent Object-Oriented Programming on Large Scaled Shared Memory Architectures: A Structured Adaptive Mesh Refinement Method in ICC++, Master's thesis, University of Illinois (1994).
5) Laudon, J. and Lenoski, D.: The SGI Origin: A ccNUMA Highly Scalable Server, *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 241–251 (1997).
6) Mohr, E., Kranz, D. A. and Halstead Jr., R.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280 (1991).
7) Oyama, Y., Taura, K., Endo, T. and Yonezawa, A.: An Implementation and Performance Evaluation of Language with Fine-Grain Thread Creation on Shared Memory Parallel Computer, *Proceedings of PDCS '98*, Las Vegas, Nevada, USA, IASTED (1998).
8) Taura, K. and Yonezawa, A.: Schematic: A Concurrent Object-Oriented Extension to Scheme, *Proceedings of Workshop on Object-Based Parallel and Distributed Computation*, Lecture Notes in Computer Science, Vol. 1107, Springer-Verlag, pp. 59–82 (1996).

(平成10年 7 月24日受付)
(平成10年10月10日採録)



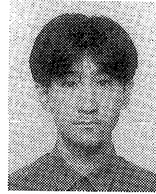
外山 純生

1975年生．1998年東京大学理学部情報科学科卒業．現在東京大学大学院理学系研究科情報科学専攻修士課程に在学中．主に並列プログラミング言語の研究に従事．



大山 恵弘（学生会員）

1996年東京大学理学部情報科学科卒業．1998年東京大学大学院理学系研究科情報科学専攻修士課程終了．現在同大学院博士課程在学中．主に並列プログラミング言語の研究に従事．



田浦健次朗（正会員）

1969年生．1996年より東京大学大学院理学系研究科助手．1997年東京大学大学院より理学博士取得．並列プログラミング言語の設計，実装に関する研究に従事．

米澤 明憲（正会員）

1947年生．1977年 Ph. D. in Computer Science（MIT）．1989年より東京大学理学部情報科学科教授．超並列・分散ソフトウェアアーキテクチャ，などに興味を持つ．共著書「算法表現論」,「モデルと表現」（岩波書店），編著書「ABCL: An Object-Oriented Concurrent System」（MIT Press）等がある．1992年–1996年ドイツ国立情報処理研究所（GMD）科学顧問，ACM Transaction on Programming Languages and Systems 副編集長，IEEE Parallel & Distributed Technology および Computer 編集委員などを歴任，元日本ソフトウェア科学会理事長．