最小限のコンパイラサポートによる細粒度マルチスレッディング——効率的なマルチスレッド言語を実装するためのコスト効率の良い方法

田浦 健次朗† 米澤 明憲†

マルチスレッド言語（細粒度スレッドの動的な生成を提供する言語）の実行モデルを C の 1 スタックによる実行モデル上で実現することは難しい．そのため，これまでの効率的なマルチスレッド言語の研究では，独自のフレームフォーマットやフレーム割当て方式を，専用のコンパイラとともに実装してきた．この論文はそれに代わって，既存の逐次 C コンパイラを最大限に有効利用する，費用対効果の高い実装方式 StackThreads について述べる．StackThreads では，スレッドの生成は単なる C の手続き呼び出しによって行われる．これによりスレッド生成の性能は非常に高くなる．それに加え，ある手続きが実行を中断したときに，その手続きのコンテクスト（大体，その手続きのスタックフレームに相当する）をヒープにコピーし，後に再開するための機構を提供する．StackThreads によって，コンパイラの作成者は，言語の逐次構文を C の対応する構文に変換し，並列構文を実現するために単に StackThreads の機能を呼び出す，という単純なコンパイル方法をとることができるようになる．

Fine-grain Multithreading with Minimal Compiler Support ——A Cost Effective Approach to Implementing Efficient Multithreading Languages

Kenjiro Taura† and Akinori Yonezawa†

It is not easy to map the execution model of multithreading languages (languages which support fine-grain dynamic thread creation) onto the single stack execution model of C. Consequently, previous work on efficient multithreading uses elaborate frame formats and allocation strategy, with compilers customized for them. This paper presents an alternative cost-effective implementation strategy for multithreading languages which can maximally exploit current sequential C compilers. In StackThreads, a thread creation is done just by a C procedure call, maximizing thread creation performance. When a procedure suspends an execution, the context of the procedure, which is roughly a stack frame of the procedure, is saved into heap and resumed later. With StackThreads, the compiler writer can straightforwardly translate sequential constructs of the source language into corresponding C statements or expressions, while using StackThreads primitives as a blackbox mechanism which switches execution between C procedures.

1. 導 入

多くの並列言語は動的なスレッド生成をサポートしている．言語コンストラクトの例としては，future[16),32)]，並列オブジェクト間非同期メッセージ送受信[1),7),33)]，fork-join[21)]，並列ブロックとループ[6),8)]，暗黙的並列性[3),22)] などがある．動的なスレッド生成をともなう並列言語（以後マルチスレッド言語と呼ぶ）の実装は，良い逐次性能を犠牲にすることなしに，効率的なマルチスレッドを実現できなくてはならない．

本論文はマルチスレッド言語の高効率な実装法に対する有効な選択肢を提供する．その機構は StackThreads と呼ばれる**ライブラリ**によって提供される．伝統的なスレッドライブラリとは異なり，StackThreads はマルチスレッド言語の性能に対する要求——小さい生成オーバヘッド——を満たしている．StackThreads では新しいスレッドを生成して実行することは，パラメータ渡しを含め，数命令しか要しない．実際，C 関数 f の本体を実行する新しいスレッドを始めるには，単に手続き f を呼び出せばよい（実装されている言語構文によっては追加のパラメータがつけられる）．スレッドが中断したときのために，(1) C の手続きのコンテクストを保存してその **caller** を再開する，(2) 保

† 東京大学
University of Tokyo

存されたコンテクストから中断したスレッドを後に再開する，ための機構が提供される．これらまたは似たような機構が，独自のフレーム管理プロトコルを仮定して実装されているこれまでのマルチスレッド言語の実装とは異なり，StackThreads はこれらを，4 章で述べるいくつかの条件を満たすすべての C コードに対して達成する．つまり生成された C コードには，スレッドが実行するコードと，スレッドが中断するところで StackThreads ライブラリへの呼び出しが含まれるだけであり，明示的にフレームを管理したり，スレッドを切り替えるために明示的に変数を退避させるなどのコードは含まれない．これはコンパイラ開発の手間を削減し，さらには C コンパイラによる最適化をより有効に活用できるようにする．

本論文は以降次のように構成される．2 章はこれまでの研究を概観する．3 章は StackThreads の機構が依存しているスタックフレームの配置と C 手続きのコード生成の慣例をまとめる．4 章は我々の機構がどのように働くかを詳しく述べる．5 章では性能を報告し，6 章でまとめと結論を述べる．

2. 関 連 研 究

この章はこれまでの高効率マルチスレッディング方式の概観をする．伝統的な CPU の上に実装された方式のみに焦点を当て，マルチスレッドアーキテクチャの上に実装されたものについては議論しない．それらのマルチスレッディング方式はすべて独自のフレーム管理と手続き結合慣例を設計しているか，一般的なマルチスレッド機構なしで実装可能なように並行モデルを制限しているかである．**表 1** にこれらの仕事をおおよその年代順に並べ，サポートされている並行モデルとコード生成方法を付記する．**並行モデル**の欄は，もし一般的なマルチスレッドを実装していたら**一般**であり，そうでなければ**制限**と書いてある．**一般**マルチスレッドモデルとは，スレッド間の同期の向きなどに制約がなく，少なくともすべてのスレッドは，それが唯一の実行可能スレッドになったとき，スケジューリングされることを保証しているモデルである☆．**コード生成方法**の欄には，もしそれらがアセンブリ言語を生成する場合は**アセンブリ**と，C コードを生成するが，フレーム管理やコンテクスト切替えのコード列を埋め込むのがコンパイラの責任であるならば**アセンブリ的 C** と，もし単に C のスタックフレーム管理の上で走るだけならば**単純な C** と書いてある．

表 1 これまでのマルチスレッディングの高効率実装のリスト
Table 1 A list of existing efficient multithreading systems.

	並行モデル	コード生成方式	対象言語
SML/NJ [9]	一般	アセンブリ	SML/NJ [2]
LTC [12),18)]	一般	アセンブリ	Multilisp [16]
TAM [10]	一般	アセンブリ	Id [22]
ABCL [27]	一般	アセンブリ的 C	ABCL
Leapfrogging [31]	制限	単純な C	Multilisp
Olden [26]	制限	アセンブリ	Olden
Concert [25]	一般	アセンブリ的 C	CA [7]
			ICC++ [8]
Lazy Threads [14]	一般	アセンブリ	Id
Lazy RPC [11]	制限	単純な C	ParSubC [11]
Schematic [24]	一般	アセンブリ的 C	Schematic [29]
Cilk [13]	制限	アセンブリ的 C	Cilk [5]

最も特筆すべきことに，**単純な C** コード生成を採用している方法，leapfrogging [31]，Lazy RPC [11] は，一般的な並行モデルを提供していない．StackThreads を他と区別する特徴は，それが一般のマルチスレッドモデルを実装していながら単純な C コード生成を許していることである．

紙面の都合上，関連研究についてはこれ以上詳しく述べない．それらの研究については，文献 34) などで詳しく述べられている．

3. C 手続きの手続き結合慣例

3.1 概 観

本節では，StackThreads の詳細を理解するために必要な C 手続きの手続き結合方式とコード生成慣例を手短かに述べる．特に，手続きの復帰の機構の詳細——手続きがどのように caller に帰り，caller がどのように実行を継続するのか——を理解することは，普通と違うやり方で手続きのコンテクストを保存/復帰するための基本である．

3.2 スタックフレームの配置

図 1 は C 手続きの典型的なスタックフレームの配置を示している．図はスタックが下（小さなアドレスの方向）に伸びるという仮定のもとに，手続き f のスタックフレームと，その caller P を図解している．**スタックポインタ（SP）**はスタックの最も低いアドレスを指し，**フレームポインタ（FP）**は f のどこか，慣例によって定められている場所をさしている☆☆．ス

☆ 我々の研究を含む多くの研究は，より強い意味での公平性（fairness）を保証しておらず，単にランタイムシステムやコンパイラによってスレッド間にまったく依存関係が追加されないことを保証しているだけである．

☆☆ すべての手続き FP を持つとは限らないような慣例も存在するが，以降の議論には影響しない．

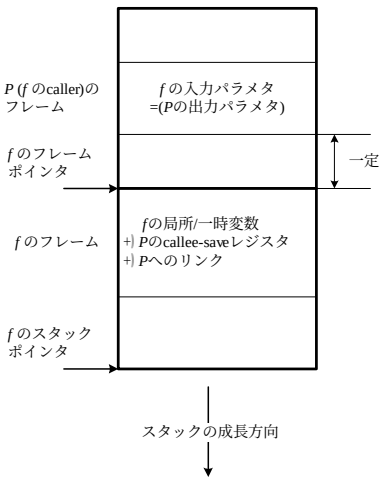

図 1 C 手続きの典型的なスタック配置
Fig. 1 A typical stack layout for C procedures.

タックフレーム f は以下のものを保持する．

- f の局所/一時変数
- P の callee-save レジスタ
- caller へのリンク（P へのポインタと戻り番地）

レジスタに割り当てられていない入力パラメータは caller のフレームに格納されるので，caller は呼ばれる手続きのフレーム配置について知る必要はない．入力パラメータが配置される領域の，callee のフレームからのオフセットはすべての手続きを通じて一定である．よって callee は caller のフレーム配置を知ることなく入力パラメータにアクセスできる．手続きが復帰するとき SP と FP を回復するのは callee の責任である．

3.3 レジスタ使用慣例

レジスタは各 CPU 上で定まる慣例により 2 つに分類される．1 つは caller が手続きをまたがって保存されることを仮定している *callee-save* **レジスタ**であり，もう 1 つは caller が手続き呼び出しをまたがると破壊されることを仮定している *caller-save* **レジスタ**である☆．手続きが正しいレジスタの状態で正しい場所に復帰するために，手続きのスタックフレームは戻り番地，親のフレームポインタ，その手続きが使用する callee-save レジスタを保存する．手続きが復帰するときには，FP，SP，callee-save レジスタの値を回復し，戻り番地にジャンプする．その後 caller は，FP，SP，callee-save レジスタが元のままの値を持っており，他のレジスタは持っていないことを仮定して実行を続ける．

Callee-save レジスタの存在は我々のスレッド実装の中の最も問題となる部分である．なぜなら手続きのコンテクストが未知の数のフレームやレジスタにわたって存在するかもしれないからである．手続き f が 4 つの callee-save レジスタ $\{A, B, C, D\}$ を使用しているとし，f は $\{A, B\}$ を使用する手続き g を呼び出し，さらに g が今度は callee-save レジスタ $\{B, C\}$ を使う手続き h を呼び出すとしよう．f に関連するコンテクストは何だろうか? h がアクティブな（つまり，そのフレームがスタックの頭にある）ときは，A と B は g のスタックフレームに保存され，C は h のスタックフレームに保存され，そして，D はまだレジスタの上にある! 明らかに，この状態から f のコンテクストを保存するためには，それらがどこにあるかが分からなくてはならない．

どの callee-save レジスタが手続きによって使われていて，どこにそれらが保存されているかの情報はスタックフレーム中には存在しない．もしそれがあったとしても，その情報を解釈しながらそれらを回復することはコンテクスト切替えを非常に遅くする．代わりに，我々が callee-save レジスタを扱う方式は，C コンパイラのコード生成方式に関するある仮定に依存している．その仮定とは，手続きは callee-save レジスタを手続きの**入り口**（*prologue*）ですべて一度に保存し，**出口**（*epilogue*）ですべて一度に回復するというものである．言い換えれば，手続きは実行が進むに従ってそれらを徐々に保存する，ということをしない．その仮定は GNU C コンパイラを含む，我々が知るすべての C コンパイラであてはまる．実際いくつかの手続き呼び出し慣例で[17] は，callee-save レジスタが入り口で保存されることを明示的に述べている．この仮定は，スレッドを中断させ，一貫した状態で親を再開させる，*epilogue code threading* と呼ばれる，4.3 節で説明される興味深い技法を有効にする．

3.4 まとめ：コンテクストはどこにあるのか?

手続き I がアクティブでない（つまり，そのスタックフレームがスタックの頂上にない）とき，I に関連づけられたコンテクストは次からなる．

- I の局所/一時変数
- I の入力パラメータ
- I の SP と FP
- I の callee-save レジスタ

局所/一時変数は I のフレーム中にあり，入力パラメータは I の caller のフレーム中にある．FP は I の直

☆ SP と FP も手続き呼び出しをまたがって保存されることが仮定されている．しかし，我々の目的のために，我々はそれらを特別なレジスタと見なし，他の通常の callee-save レジスタとは区別する．

接の子のフレームの中に保存される．最後に，callee-save レジスタの位置を特定するのは前述したように難しい．次の章では最初の 3 つをどうやって見つけ，必要な callee-save レジスタをどう獲得するかについてさらに詳しく述べる．

4. StackThreads：枠組みと実装

4.1 概　　観

StackThreads の実行方式の基本的な考え方は単純で，すでに別の場所で著者らによって発表されている[28]．手続き f の評価をする新しいスレッドを fork するときには f を単なる逐次呼び出しで呼び出す．もし f が中断したときには，そのコンテクストをスタックからヒープに移動し，スタックを巻き戻すことで caller を再開することができる．たとえ callee が中断しても caller は再スケジュールされうるので，これで結果的にスレッドを生成したということができる☆．もし f が後に再スケジュールされるときには，コンテクストはスタックの上に回復され，制御は中断した点に移る．以下で，スタックフレームのどの部分とどのレジスタが保存，復帰されなければならなくて，それらをどのように伝統的な C のスタックフレームから正しく獲得するのかについて述べる．

4.2 C 手続き呼び出しによってスレッドを生成する

C の手続き f の本体を評価する新しいスレッドを fork したいとしよう．そのスレッドの親は通常の C の手続き呼び出しとまったく同じ方式でパラメータを渡して単に f を呼び出す．もし f がうまく実行を中断せずに終了したならば，結果の値は手続き呼び出しの返り値として得られる．スレッドの生成と関数呼び出しの唯一の違いは，スレッドの生成では，たとえ f が中断しても，制御が caller に戻ることがあるという点である．その場合 f の返り値は当然ながらまだ定義されていない．f がひとたび中断したあとは，f の caller は，もはや f の終了後すぐにスケジュールされるとは限らないので，f が C の `return` 文によって結果の値を返すことは意味を持たない．ゆえに，一般にスレッドは caller に対して，自分が終了したのか中断したのかを教え，もし中断したならば 2 つのスレッドが以後返り値を通信する場所を教えることがしばしば必要になる．特に，いわゆる逐次呼び出しは，もしその呼び出しが中断する可能性があるならば，この種のプロトコルを使用して実装されなければならない．StackThreads は，適切なプロトコルはしばしば言語独立のものであり，時には不必要なものであるという観察に基づき，これに対するいかなる固定したプロトコルも定義していない．そのプロトコルは，たとえば，すべてのメソッド起動が非同期メッセージによってなされ，結果の値が別の非同期メッセージで渡されるピュアな Actor に基づく言語では不必要である．

4.3 Epilogue Code Threading によってスレッドを中断する

StackThreads はスレッドがヒープにコンテクストを保存して中断する，具体的には，現在のフレームのすぐ下のフレーム（そのスレッドの**現在の親**）に制御を移す方法を提供する．現在の親とは，そのスレッドが初めて中断したときにはもともとの caller（作成者），そうでなければそのスレッドを最も最近再開させたスレッドである．

スレッド P が f を fork し，今 f は自身の実行を中断して P を再開したいとする．f はまず StackThreads が提供する関数 `alloc_ctxt` を呼ぶことによって適切な大きさの，コンテクストを格納する領域を確保し，その後，実際のコンテクスト切替えを起動するために，やはり StackThreads で提供される関数 `switch_to_parent` を呼ぶ．このときパラメータとして確保された領域へのポインタを渡す．`switch_to_parent` はその領域に f のコンテクストを格納し，f の現在の親を再開する．f が後に再開されるときには，あたかも `switch_to_parent` が正常に復帰したかのように制御が f に戻る．通常コンテクスト切替えのコードは，中断したスレッドのコンテクストに対し，言語依存の操作（たとえばそのコンテクストへのポインタを，あとで再開するためにどこかへ保存するなど）を行う必要があるため，コンテクストのための領域確保と実際のコンテクスト切替えは分離されている．典型的なコンテクスト切替えで行われることは，まず領域を確保し，それへのポインタを必要な言語構文を実装するためにどこかに保存し，そして `switch_to_parent` を呼び出すことである．またこのインタフェースは，1 つの手続きが複数回中断するときに，コンテクスト保存のために同じ領域を再利用することも可能にしている．

さしあたり簡単のために，f が `switch_to_parent` を**直接**呼び出し，ゆえに `switch_to_parent` は中断したスレッドのフレームが現在のフレームのすぐ下にあることを知っていると仮定しよう．`switch_to_parent` が f によって呼ばれた直後のスタックフレームとコントロールフローが**図 2** に図解されている．太線は手続きの prologue と epilogue のコード列を示す．後にあたかも

☆ 2 章でも述べたが，スレッドの公平性は無視している．

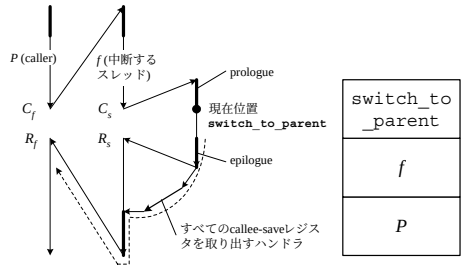

図 2 P が f を fork し，f が自分を中断するために `switch_to_parent` を呼び出したときの制御フローおよびスタックフレームの配置．制御は現在 *the current position* で示される点にあり，P を再開させようとしている．

Fig. 2 Control flow and stack frames, after P forked f, which then called `switch_to_parent` to suspend itself. Control is now at position indicated by *the current position*, and P is about to resume.

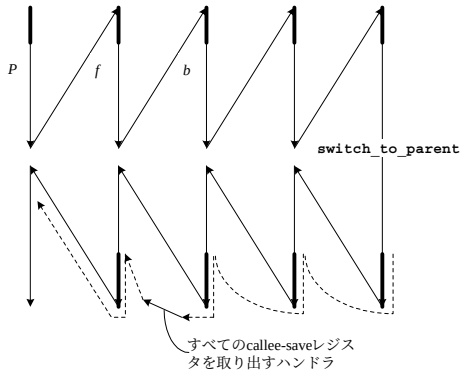

図 3 F の現在の親を再開するときの制御パス（一般的なケース）

Fig. 3 Control flow to resume F's current parent (general case).

`switch_to_parent` が f に復帰したかのように f を再開するためには，我々は，(1) f が `switch_to_parent` を呼んだ点（図中では C_s）での f の状態を獲得，保存し，(2) P が f を呼び出した点（図中では C_f）におけるスタック/フレームポインタ，callee-save レジスタの値とともに f の戻り番地（図中では R_f）に制御を移動しなければならない．

f の状態はフレーム内の局所/一時変数と f の入力パラメータ，callee-save レジスタからなる．局所/一時変数を保存することは，フレーム f 内の局所変数保存領域の最高と最低のアドレスを必要とし，入力パラメータについてはその大きさを必要とする（その領域の FP からのオフセットはすべての手続きを通じて定数である）．Callee-save レジスタを保存することはより複雑である．どの callee-save レジスタを `switch_to_parent` が破壊するのかが分からないため☆，C_s における callee-save レジスタを獲得する唯一の実際的な方法は**実際に `switch_to_parent` の epilogue コードを走らせ**，そこでの callee-save レジスタの値を獲得することである．これは，`switch_to_parent` の epilogue を走り終わったあとに特別なハンドラルーチンにジャンプするように `switch_to_parent` の戻り番地を修正することで実現している．そのハンドラルーチンはレジスタ使用慣例で定められた**すべての** callee-save レジスタを保存し，f の epilogue コードにジャンプする．その後 f の epilogue コードは f が使用した callee-save レジスタを復帰し，P に戻る．f の callee-save レジスタを保存し，P を再開する制御パスは図 2 の点線によって示されている．制御パスは，f の手続きの本体の残りの部分が省略されていることを除けば通常の制御パスと同等であることに注意してほしい．

我々はこれまで `switch_to_parent` が f から直接呼ばれることを仮定してきた．この仮定を緩め，`switch_to_parent` が f から呼ばれている手続きから間接的に呼び出されているような一般的な場合を考える．StackThreads は言語実装者のために `switch_to_parent` が間接的に呼ばれることを許している．仮にもしそれが中断する手続きから直接にしか呼び出せないとすると，あらゆる同期地点で，継続するか中断するかを決定するコード列はつねにその関数内に埋め込まなくてはならない．そのような埋め込まれたコード列はときに非常に長くなることがあるため，スレッドが小さいオーバヘッドで継続できる場合のコードのみを埋め込み，それ以外の場合は分離した手続きとして記述できるようにしたい．そのためには `switch_to_parent` が，中断する手続きから何段かの手続き呼び出しを経て間接的に呼び出されることを許す必要がある．

一般化した場合を**図 3** に示す．P が f を fork したとし，f は関数 b を呼んだとし，b は最終的に f が中断しなければならないという決定をしたとする．この場合の StackThreads によって提供される意味は，後に f が再開するときに，あたかも b が f へ復帰したかのように f を再開するというもので，**`switch_to_parent` から b までの計算は単に破棄される**．つまり保存されるのは f のフレームだけであり，`switch_to_parent` から b に至るまでのフレームは単に棄てられる．b 以

☆ 実際には，生成されるアセンブリコードを見るか，それらを最初からアセンブリで書くことによって，どの callee-save レジスタを `switch_to_parent` が使用しているのかを知ることができる．しかし一般には，`switch_to_parent` は f から間接的に呼ばれ，その場合は `switch_to_parent` によって破壊された callee-save レジスタのみを回復するだけでは十分ではない．

外のすべてのフレームは，その epilogue 列を実行するとその caller の epilogue コードにジャンプするように，そして b のフレームはすべての callee-save レジスタを保存するハンドラへジャンプするようにと，戻り番地が書き替えられる．switch_to_parent から P へ至る制御の流れは図 2 の点線に示すように呼び出しの連鎖のすべての epilogue コード列の連なりである．

4.4 呼び出しの鎖の再構成によって中断したスレッドを再開する

スレッド A が，中断しているスレッド f の実行を再開する条件を満たしたとする．スレッド A は restart_thread(c) を呼ぶことによって f を再開することができる．ここで c は switch_to_parent によって満たされた，ヒープ上のコンテクストである．基本的な操作は次のようになる．スタックの頭に f の局所変数と入力パラメータの領域を作り，f の callee-save レジスタを回復し，SP と FP を新しいフレームの場所にセットし，再開地点にジャンプする．このとき，f がやがて終了するか再び中断したときには，f が正しい callee-save レジスタの状態とともに，正しく restart_thread に復帰するように注意を払わなければならない．我々はここで f の途中にジャンプしようとしているので，通常の方法で f を呼び出したときに実行される，f の正規の prologue 列を実行しない．したがって，もしも何も注意を払わずに f を再開させたとすると，f が epilogue コード列を実行したときには，restart_thread に復帰するのではなく，f の**もともとの** caller へ復帰してしまう．この問題に対する我々の対処は，2 つの部分からなっている．

- f が正しい FP と SP の値とともに restart_thread へ復帰するように，restart_thread は，f のフレーム内で caller へのリンクを保持しているのスロットを上書きする．より具体的にいえば，caller の戻り番地と，FP を保存しているスロットを上書きする．
- restart_thread は，f を再開する前にその CPU の慣例で定まるすべての callee-save レジスタを保存し，f が復帰したあと，自分でそれらの callee-save レジスタを回復する．

これにより，f はあたかも restart_thread から呼び出されたかのように振る舞うことになる．

まとめると，中断したスレッドを再開するために次のステップを踏む．(1) f のためにスタックフレームを確保する．(2) 局所変数と入力パラメータをスタックにコピーする．(3) f が restart_thread に復帰できるように，親へのリンク（戻り番地と親の FP）を restart_thread へのリンクに置き換える．(4) すべての callee-save レジスタを保存する．(5) f が中断したときに保存された callee-save レジスタを回復する．(6) f の再開地点にジャンプする．f はやがて無効な callee-save レジスタの状態で restart_thread に復帰するので，restart_thread は f が復帰した直後に (4) で保存された callee-save レジスタをすべて回復する．f を再開し，最終的に f が restart_thread に復帰するまでの制御パスを**図 4** に示す．

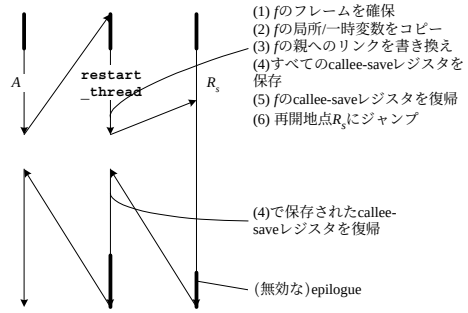

図 4 中断したスレッド f を回復する制御パス
Fig. 4 Control flow to resume a once suspended thread f.

4.5 制限と議論

StackThreads は C で可能なプログラミングスタイルをできるだけ多くサポートすることを目指しているが，いくつかの制限も存在する．

まず，StackThreads は基本的にスタックフレームが移動可能である事実，より正確にいえば，スタックに確保されたデータは FP および SP に相対的にアクセス可能であることに依存している．この性質はもちろんすべての C プログラムとコンパイラが満たしているわけではない．特に StackThreads は，スタックに確保されたデータのアドレスを取り出し，そのアドレスがコンテクスト切替えをまたがって有効なデータを保持しつづけることを仮定することを禁止している．さらに悪いことに，たとえ手続きが明示的にスタックデータのアドレスを取り出さなくても，最適化器が汎用レジスタにそのアドレスをキャッシュし，手続き全体にわたってそのアドレスを使用することがありうる．これは典型的には集合データ（配列もしくはスタックに確保された構造体）に対して行われる．総合すると，StackThreads はスタック上にどんな集合型データを確保することも奨励していない．我々はごみ集めをサポートする（すべてのデータをヒープに確保する）言語については，これが致命的な制限でないことを信じている．我々は StackThreads をごみ集めをサポート

する並列オブジェクト指向言語 ABCL/f を実装するために実際に利用した．

第 2 に，StackThreads は，コンテクスト切替えのコストを小さくするために生きている変数の情報を利用できない．手続きのすべての局所変数を保持している領域全体を保存/回復している．これは潜在的には問題であるが，5 章の性能評価は他の要素が支配的であることを示している．

5. 性能評価

5.1 マイクロベンチマーク

StackThreads では，スレッド生成は単に手続き呼び出しである．ゆえに，オーバヘッドは，ターゲットマシンにおける手続き呼び出しに必要なものすべてであり，それはレジスタの保存とパラメータの準備に支配されている．ゆえに，興味深い数字は中断と再開のコストである．`switch_to_parent` が直接中断したスレッドから呼ばれた場合の，中断と再開のコストを分解して**表 2**，**表 3** に示す．数字は SPARC の命令数で与えられている．オーバヘッドは手続きの局所変数の個数（l），手続きの入力パラメータ（p），慣例で定まる callee-save レジスタの数（r）に依存している．中断のコストは，新しいコンテクストを確保するか，この手続きの以前中断したときのコンテクストを再利用できるかどうかに依存する．

SPARC のレジスタ使用慣例（$r = 14$）において典型的な手続きとして $l = 16$，$p = 3$ を仮定すると，中断コストは 267 命令（新しいコンテクストを確保するとき）もしくは 178 命令（コンテクストを再利用するとき）であり，再開のコストは 191 命令である．局所変数とパラメータをコピーすることが，中断においては総命令数の 3 分の 1 を数え，再開においては半分を数える．再開については，callee-save レジスタの保存と回復が総命令数の 4 分の 1 を占める．スレッドが中断，再開を繰り返す簡単なベンチマークプログラムでは，中断と再開の合計に要する時間は，150 MHz HyperSparc 上で 2.31 μs である．これは Plevyak らによって報告された結果[25] に匹敵する．

4.5 節ですでに議論されているように，我々の方式は生きている変数の情報をまったく無視しており，すべての局所変数とパラメータをコピーしている．この潜在的な制限要因にもかかわらず，実際的にはそれらは大きな問題ではないことをこれらの結果は示している．StackThreads において生きている変数の情報を利用することは多くても中断コストの 3 分の 1，再開コストの半分を節約するだけである．実際の影響は詳細に依存するものの，これは切替えに少々のコストを払っても低オーバヘッドの生成を優先するどんなマルチスレッディング方式にもあてはまる．

コンテクスト切替えのほとんどすべての命令がライブラリで共有されていることも指摘しておきたい．ベンチマークプログラムでは，中断のために埋め込まれたコード列は 9 命令のみであり，すべての他の命令はすべてのコンテクスト切替えの場所で共有されるか，またはどのみち必要なもの（たとえば epilogue のコード列）である．

表 2 命令数による中断コストの分解．パラメータ l，p，r は局所/一時変数の個数，入力引数の個数，callee-save レジスタの個数をそれぞれ表している．
Table 2 Instruction counts of a suspension. Parameters l, p, and r refer to the number of local/temporary variables, incoming arguments, and callee-save registers, respectively.

分類	詳細	命令数
1. コンテクストの割当て	1-1. コンテクストの大きさの計算	33
	1-2. MALLOC	28
	1-3. 初期化	27
2. 親に制御を移す	2-1. 戻り番地の変更	22
	2-2. 局所/一時変数のコピー	$16 + 3.25l$
	2-3. パラメータのコピー	$10 + 5p$
	2-4. Callee-save レジスタの領域の設定	11
	2-5. Epilogue の実行	6
3. Callee-save レジスタの保存	3-1. Callee-save レジスタを保存	$7 + r$
	3-2. スレッドの Epilogue の実行	11（スレッドに依存）
その他		15
合計（含コンテクスト割当て）		$186+3.25l + 5p + r$
合計（除コンテクスト割当て）		$98 + 3.25l + 5p + r$

表 3 命令数による再開コストの分解．パラメータ l，p，r の意味は，表 2 と同様
Table 3 Instructions counts of a resumption. See Table 2 for the definitions of parameters l, p, and r.

項目	命令数
1. 初期化	21
2. 局所/一時変数のコピー	$16 + 3.25l$
3. パラメータのコピー	$10 + 5p$
4. Caller へのリンクを設定	13
5. Callee-save レジスタのスタックへのコピー	$1 + 2r$
6. コンテクストを解放すべきかの検査	6
7. Callee-save レジスタの入替え	$1 + 2r$
合計	$68+3.25l+ 5p + 4r$

5.2 アプリケーションによる評価

5.2.1 設　　定

スレッド生成やコンテクスト切替えのオーバヘッドが実際のアプリケーション実行時間に対して課す影響を見るために，我々は以下の 3 つのアプリケーションの性能を測定した．

BH： Barnes-Hut N 体シミュレーション[4),15)]．並列プログラムにおいては，すべてのプロセッサが，BH-tree と呼ばれるデータ構造を共有する．粒子はプロセッサ間に分割され，各プロセッサは割り当てられた各粒子にかかる力を順に計算する．以下では力の計算部分だけを測定している．

RNA： 組合せ探索と枝刈りによる並列 RNA2 次構造予測[19)]．再帰呼び出しがある深さに達するまで並列に再起呼び出しをすることで並列性が抽出される．この並列に呼び出す深さの限界は，ユーザが実行時に与える．この深さの最適な値は入力および実行時の枝刈りの状況に依存し，コンパイル時には予測できない．

CKY： CKY アルゴリズムによる文脈自由文法（CFG）構文解析[20),23)]．長さ n の入力文が与えられると，$0 \leq i < j \leq n$ を満たす各 (i, j) の組に対し，集合 $S_{i,j}$ を計算する．各 $S_{i,j}$ に対して 1 つスレッドを割り当てて計算を行う（つまり合計 $1/2\, n(n+1)$ のスレッドが作られる）．$S_{i,j}$ を計算するスレッドは，$i < k < j$ を満たす各 k に対して，$S_{i,k}$ と $S_{k,j}$ の値を必要とする．もし，この必要なデータを使おうとするときにそれらがまだ計算されていなければ，そのスレッドは中断する．

各アプリケーションに対して，純粋な逐次プログラムを C++によって書き，それらに StackThreads プリミティブを付加したプログラム（以下 StackThreads プログラム）と比較した．StackThreads プログラムも 1 台のプロセッサ（HyperSparc 150 Mhz）上で実行する．より詳しくは，

- StackThreds プログラムにおいては実際の並列・分散実行であれば必要なところで新しいスレッドが fork され，同期のための条件が検査される．
- StackThreds プログラムにおいては，中断するかもしれない手続きの各呼び出し地点で（たとえそれが通常の逐次呼び出しであっても），新しいスレッドが fork される．

最後の項目は，StackThreads が中断するかもしれない手続きに対する逐次的な呼び出しを直接提供していないために必要になる．そのような呼び出しは，スレッド生成とスレッド間の明示的な同期および通信を使って実現される．

アプリケーションは GNU C++（g++）を使って，最大の最適化オプションをつけてコンパイルした．そして以下の 3 種類のプログラムを比べた．

SQ： C++による純粋な逐次実行．マルチスレッドによるオーバヘッドはない．

FK： SQ に，上で述べたように StackThreads プリミティブを加えたもの．スレッド生成と同期条件の検査のためのオーバヘッドが加わる．同期は実際にはけっして失敗しない．そのため，制御の流れは SQ のものと同一である．

SW： スレッドを同期地点で中断させて，実際の分散記憶計算機上での実行を模擬したもの．

各アプリケーションが実際にどこで fork や中断を生ずるかについては詳しくは文献 30) を参照されたい．

5.2.2 実行オーバヘッド

図 5 は FK 版および SW 版の（SQ 版に対する相対）実行時間を示している．FK は並列プログラムを 1 台のプロセッサで実行したときに現れるオーバヘッドを評価している．オーバヘッドの源は，スレッド生成，同期条件の検査，および同期のためのデータ構造の生成である．SW は実際の分散記憶並列計算機における実行において各プロセッサ上で生ずる，スレッド生成，同期，および切替えのオーバヘッドを示している（通信などのオーバヘッドは考慮されない）．すべてのベンチマークで，FK 版のオーバヘッドは 15%以内，SW 版のオーバヘッドは 30%以内であった．

表 4 は各版の実行時間（それぞれ T_{SQ}，T_{FK}，および T_{SW} と呼ぶ），fork の数（F），同期の数（S），および中断した回数（B）を示している．言い換えれば，S は潜在的な中断地点の数であり，B は実際の中断によるスレッド切替えの回数である．実行時間は HyperSparc 150 Mhz のプロセッサにおいて，ms 単位での時間である．

$(T_{SW} - T_{FK})/B$ は，1 回の切替えあたりの大体の

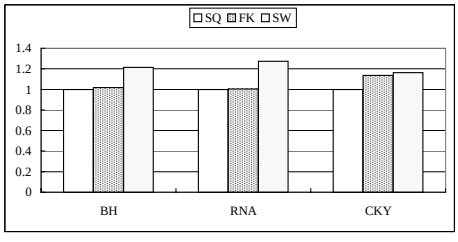

図 5 各版の逐次版の実行時間を 1 としたときの実行時間
Fig. 5 Elapsed time of each version (relative to the sequential version).

表 4 各版の ms 単位の実行時間（T_{SQ}，T_{FK}，and T_{SW}），fork の実行数，潜在的な中断地点数，および実際の中断数（それぞれ，F，S，および B）
Table 4 Elapsed time in milliseconds (T_{SQ}, T_{FK}, and T_{SW}) and the dynamic counts of forks (F), potential blocking points (S), and actual blocking points (B).

アプリケーション	T_{SQ}	T_{FK}	T_{SW}	fork 回数（F）	同期回数（S）	中断回数（B）
BH	5,224	5,321	6,338	1,298,124	1,323,554	68,360
RNA	1,095	1,101	1,395	160,411	177,341	33,861
CKY	5,803	6,602	6,746	14,560	941,690	28,524

時間を与える．それはアプリケーションによって異なるが，5 μs から 15 μs 程度の値である．

6. まとめと結論

StackThreads は効率的なマルチスレッド言語を実装するための実際的なアプローチを提供する．それは非常に低いオーバヘッドのスレッド生成と，普通に書かれた C の手続きとの間の効率的なスレッド切替えをサポートしている．効率的なマルチスレッディングに関する以前までの方式とは異なり，それはコード生成器との広範な連携を必要とせず，それゆえに，コンパイラの設計，実装者は StackThreads の並列/並行プリミティブの低オーバヘッドのマルチスレッディングのサポートを受ける一方で，言語の逐次のコンストラクトを直接 C に対応させることができる．性能測定の結果は SPARC における中断コストが今までに知られた最も良い結果[25] に匹敵することを示している．彼らの結果は我々のものとは異なり，コンパイラの広範な連携を必要としている．

参 考 文 献

1) Agha, G.A.: *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts (1986).
2) Appel, A.W.: *Compiling with Continuations*, Cambridge University Press (1992).
3) Arvind, Heller, S.K. and Nikhil, R.S.: Programming generality and parallel computers, Technical Report, Massachusetts Instituted of Technology, Cambridge (1988).
4) Barnes, J. and Hut, P.: A hierarchical $O(N \log N)$ force-calculation algorithm, *Nature*, Vol.324, pp.446–449 (1986).
5) Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H. and Zhou, Y.: Cilk: An efficient multithreaded runtime system, *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP*), pp.207–216 (1995).
6) Chandy, K.M. and Kesselman, C.: CC++: A declarative concurrent object-oriented programming notation, *Research Directions in Concurrent Object-Oriented Programming*, chapter 11, pp.281–313, The MIT Press (1993).
7) Chien, A.A.: *Concurrent Aggregates* (*CA*), MIT Press (1991).
8) Chien, A.A., Reddy, U.S., Plevyak, J. and Dolby, J.: ICC++ – A C++ dialect for high performance parallel computing, *Proc. 2nd International Symposium on Object Technologies for Advanced Software* (1996).
9) Cooper, E.C.: Adding threads to standard ML, Technical Report 90-186, Carnegie Mellon University, Pittsburgh (Dec. 1990).
10) Culler, D.E., Sah, A., Schauser, K.E., von Eicken, T. and Wawrzynek, J.: Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine, *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.166–175 (1991).
11) Feeley, M.: Lazy remote procedure call and its implementation in a parallel variant of C, *Proc. International Workshop on Parallel Symbolic Languages and Systems*, Vol.1068, Lecture Notes in Computer Science, pp.3–21, Springer-Verlag (1995).
12) Feeley, M.: An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors, PhD Thesis, Brandeis University (1993).
13) Frigo, M., Leiserson, C.E. and Randall, K.H.: The implementation of the Cilk-5 multithreaded language, *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (*PLDI*) (1998).
14) Goldstein, S.C., Schauser, K.E. and Culler, D.: Enabling primitives for compiling parallel languages, *Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers* (1995).
15) Grama, A.Y., Kumar, V. and Sameh, A.: Scalable parallel formulation of the Barnes-Hut method for n-body simulations, *Proc. Supercomputing '94*, pp.439–448 (1994).
16) Halstead, Jr., R.H.: Multilisp: A language for

concurrent symbolic computation, *ACM Trans. Programming Languages and Systems*, Vol.7, No.4, pp.501–538 (Apr. 1985).
17) Kane, G. and Heinrich, J.: *MIPS RISC Architecture*, Prentice Hall (1992).
18) Mohr, E., Kranz, D.A. and Halstead, Jr., R.H.: Lazy Task Creation: A techinque for increasing the granularity of parallel programs, *IEEE Trans. on Parallel and Distributed Systems*, Vol.2, No.3, pp.264–280 (July 1991).
19) Nakaya, A., Yamamoto, K. and Yonezawa, A.: RNA secondary structure prediction using highly parallel computers, *Comput. Applic. Biosci.* (*CABIOS*), Vol.11 (1995).
20) Nijholt, A.: Parallel approaches to context-free language parsing, *Parallel Natural Language Processing*, pp.135–167, Ablex Publishing Corporation (1994).
21) Nikhil, R.S.: Parallel symbolic computing in *Cid*, Ito, T., Halstead, Jr., R.H. and Queinnec, C. (Eds.), *Proc. International Workshop on Parallel Symbolic Languages and Systems*, Vol.1068, Lecture Notes in Computer Science, pp.217–242, Springer-Verlag (1995).
22) Nikhil, R.S. and Arvind: Id: A language with implicit parallelism, Technical Report, Massachusetts Instituted of Technology, Cambridge (1990).
23) Ninomiya, T., Taura, K., Torisawa, K. and Tsujii, J.: A scalable implementation of parallel CKY algorithm in concurrent object-oriented language ABCL/f, *Proc. JSSST Workshop on Object-Oriented Computing* (*WOOC*) (1997). (in Japanese).
24) Oyama, Y., Taura, K. and Yonezawa, A.: An efficient compilation framework for languages based on concurrent process calculus, *Proc. Europar '97*, Vol.1300, Lecture Notes in Computer Science, pp.546–553 (1997).
25) Plevyak, J., Karamcheti, V., Zhang, X. and Chien, A.A.: A hybrid execution model for fine-grained languages on distributed memory multicomputers, *Supercomputing '95* (1995).
26) Rogers, A., Carlisle, M., Reppy, J. and Hendren, L.: Supporting dynamic data structures on distributed memory machines, *ACM Trans. Programming Languages and Systems*, Vol.17, No.2, pp.233–263 (1995).
27) Taura, K., Matsuoka, S. and Yonezawa, A.: An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers, *Proc. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (*PPoPP*), pp.218–228 (1993).
28) Taura, K., Matsuoka, S. and Yonezawa, A.: *StackThreads*: An abstract machine for scheduling fine-grain threads on stock CPUs, *Joint Symposium on Parallel Processing*, pp.25–32 (1994).
29) Taura, K. and Yonezawa, A.: Schematic: A concurrent object-oriented extension to scheme, *Proc. Workshop on Object-Based Parallel and Distributed Computation*, Vol.1107, Lecture Notes in Computer Science, pp.59–82, Springer-Verlag (1996).
30) Taura, K. and Yonezawa, A.: Fine-grain multithreading with minimal compiler support–a cost effective approach to implementing efficient multithreading languages, *Proc. 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '97*), pp.320–333 (1997).
31) Wagner, D.B. and Calder, B.G.: Leapfrogging: A portable technique for implementing efficient futures, *Proc. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (*PPoPP*), pp.208–217 (1993).
32) Weihl, W., Brewer, E., Colbrook, A., Dellarocas, C., Hsieh, W., Joseph, A., Waldspurger, C. and Wang, P.: PRELUDE: A system for portable parallel software, Technical Report MIT/LCS/TR-519, Laboratory for Computer Science, Massachusetts Institute of Technology (1991).
33) Yonezawa, A., Briot, J.-P. and Shibayama, E.: Object-oriented concurrent programming in ABCL/1, *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA '86*), pp.258–268 (1986).
34) 田浦健次朗：細粒度マルチスレッディングのための言語処理系技術（(I) および (II)），コンピュータソフトウェア，Vol.16, No.2, pp.1–19 および No.3, pp.9–28 (1999).

(平成 11 年 8 月 31 日受付)
(平成 11 年 12 月 2 日採録)

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・分散ソフトウエアアーキテクチャ等に興味を持つ．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．1992〜1996 年ドイツ国立情報処理研究所（GMD）科学顧問，ACM TOPLAS 副編集長，IEEE Parallel & Distributed Technology および Computer 編集委員等を歴任，元日本ソフトウェア科学会理事長．