KLICにおけるゴール・スケジューリング最適化

伊川雅彦[†]　大野和彦[†]　五島正裕[†]
森眞一郎[†]　中島浩[†]　富田眞治[†]

並列論理型言語KL1では、一つ一つのゴールが並行実行の単位であるため、その細粒度の並行性制御に伴うオーバヘッドが速度低下を引き起こしている。そこで本研究では、逐次実行が最適であるようなシーケンスを並行実行の単位とする最適化手法を提案する。本手法では、KL1プログラムを静的に解析し、半順序の依存関係がなりたつゴールの集合を求める。これを用いて、プログラムをスレッドと呼ぶ逐次実行が最適なゴール系列に分割する。スレッド内は逐次実行を行うように静的にスケジューリングされ、各々のスレッドは並行実行の単位として動的にスケジュールされる。一方、実行時の並列性の低下を最小限にとどめるため、要求されたデータを生成するスレッドを優先的にスケジュールするような動的なスケジューリング機構を導入する。この最適化をICOTで開発されたKL1処理系KLIC上に実装し、簡単なプログラムで性能評価を行ったところ、実行速度で約1.7倍の速度向上が達成された。

The Optimization of Goal Scheduling on KLIC

Masahiko Ikawa,[†] Kazuhiko Ohno,[†] Masahiro Goshima,[†]
Shin-ichiro Mori,[†] Hiroshi Nakashima[†] and Shinji Tomita[†]

In the conventional run-time system for the parallel logic programming language KL1, its fine-grained concurrency control frequently casues unnecessary goal switching which degrades its execution perfromance. We propose an optimization method to reduce the number of goal switching. In this method, we analyse the data dependency among goals to find a group of goals, named *thread*. A thread consists of the goals which are executed in a predefined order and are free from deadlock caused by mutual data dependency. Thus this threading reduces not only the number of goal switching but also the overhead on the switching owing to the static scheduling. On the dynamic scheduling of threads, we give a priority to each thread so that a generator thread of a request data is scheduled prior to other threads. Evaluation results show that we achieved 1.7 fold speedup by this optimization.

1. はじめに

近年、並列計算環境のハードウェア的な基盤が急速に整備されつつあるのに対し、ソフトウェアに関する課題は山積しており、特に高い性能を発揮するプログラムを容易に開発する技術が強く求められている。そこで、KL1をはじめとする並行／並列型のプログラミング言語が期待されている。これらの言語は、プログラムの並行性／並列性の記述を言語の基本的プリミティブとして備えているため、並列処理にともなうバグ混入を最小化できるという特質を持つ。また、並行／並列プロセスのネットワークを自然に記述できるため、いわゆるデータ並列に留まらない多種多様な応用とプログラミング・パラダイムに対する高い包容力も有している。

しかしながら、並列処理の重要な側面である高速性の観点からは、これらの並行／並列型言語の処理システムの性能は十分とは言えない。この性能上の問題点の大きな理由に、言語の本質である並行性制御にともなうオーバヘッドがある。KL1はその端的な例の一つであり、各々のゴールが並行実行の単位であるため、ゴール・スケジューリングの方式がプログラムの性質に適合していないと、本来逐次的に実行するのが最適な部分を並行実行するなど、無駄な処理によるオーバヘッドが著しく大きくなる場合がある。

そこで本研究では、このオーバヘッドを減少させることを目的として、ゴールのスレッド化によりゴールスケジューリングの最適化を行った。

以下第2章では本研究の背景を述べ、3章で従来のKL1処理系のゴールスケジューリングに関する問題点とその解決方法について述べる。また、5章で本論で提案する最適化手法について述べた後、第6章で本手法の性能評価とその考察を行ない、第7章でまとめを行なう。

---

† 京都大学大学院工学研究科

2. 背　　景

2.1 並列論理型言語 KL1$^{1)}$

KL1 は Flat GHC に基づく言語で、以下の形をした節の集合で表される。

$$H :- G_1 \cdots, G_m | B_1 \cdots, B_n.$$

H,G,B は述語であり、それぞれクローズヘッド、ガードゴール、ボディゴールと呼ばれる。述語は '述語名 / 引数の個数' で表記される。

ゴール G が与えられると、ヘッド H とのユニフィケーションおよびガードゴール $G_1, \cdots, G_m$ の実行が行われる。ヘッドとのユニフィケーションと、すべてのガードゴールの実行が成功した場合に、その節が選択され、ゴール G はボディゴール $B_1, \cdots, B_n$ に書き換えられ、それぞれが並列に実行される。また、ガードゴールで未具体化変数に対する参照が行われるとそのゴールは実行を中断する。これをサスペンドと呼ぶ。また、この未具体化変数が具体化されると、このゴールは実行を再開する。これを resume と呼ぶ。また、変数は単一代入であり、書き換えを行うことはできない。

2.2 KL1 処理系 KLIC$^{2)4)}$

KLIC は、ICOT で開発された KL1 の処理系であり、KL1 プログラムをいったん C プログラムに変換し、オブジェクトコードの生成はターゲットマシンの C コンパイラが行う方式をとっている。よって、通信関係などの機種依存部を除き移植性の高い処理系となっている。現在、汎用並列通信ライブラリである PVM$^{3)}$ を用いたワークステーション版 KLIC が稼働中である。

実行プログラムは KL1 プログラムから得られたオブジェクトコードと KLIC のランタイムライブラリをリンクすることで作成される。実行ゴールの選択などのプログラム全体の制御は、カーネルと呼ばれるランタイムルーチンが行う。

また KLIC には、KL1 の基本データの他に generic object と呼ばれるデータ構造が存在する。generic object はデータ領域とデータを操作するメソッドと呼ばれる手続きからなる。generic object には data, consumer, generator の 3 つのタイプがあり、用途に応じて使い分けられる。consumer object と generator object は共に未具体化変数の一種で、それぞれに対して、具体化や参照といったアクションがあった場合に自動的にメソッドが起動するようなオブジェクトである。

3. 従来手法の問題点と最適化手法の概要

3.1 従来のゴールスケジューリングの問題点

従来の KL1 処理系ではゴール間のデータ依存によるデッドロックを避けるため、ゴールを動的にスケジューリングする。この時、ゴール・スケジューリングの方式がプログラムの性質に適合していないと、無駄なスケジューリングによるオーバヘッドが著しく大きくなる場合がある。ここでは、KLIC 以前の KL1 処理系で用いられてきたプロセス指向スケジューリングと KLIC で用いられている resumption first スケジューリングの問題点について述べる。

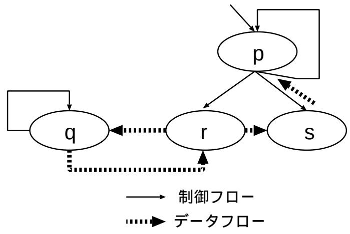


図 1　プログラム例 1

3.1.1 プロセス指向スケジューリング

KLIC 以前の従来 KL1 処理系ではプロセス指向スケジューリングと呼ばれるスケジューリング方式が用いられてきた。この方式は実行を開始したゴールのサブゴールを可能な限り実行することによって、ゴールのスイッチングのコストを減らすことを目的としている。このスケジューリング方式の問題点を図 1のようなプログラム断片を用いて考察する。

このプログラム断片のデータ依存は次の通りである。

> 再帰によるループの各ステップを考え、ループの k 回目で生成されるゴール $goal$ を $goal_k$ と表すと、q_n, r_{n+1}, s_n, p_{n+1} がそれぞれ r_n, q_n, r_n, s_n にデータ依存する。

ループの k 回目で q_k が中断し、p_k がスケジューリングされた状態を考えると、プロセス指向スケジューリングの場合、次のように実行される。

$$\begin{array}{lllll}
p_k & \rightarrow r_k & \rightarrow s_k & \rightarrow p_{k+1} & \rightarrow \\
r_{k+1}(\mathrm{susp}) & \rightarrow s_{k+1}(\mathrm{susp}) & \rightarrow p_{k+2}(\mathrm{susp}) & \rightarrow (\mathrm{rsm}) q_k & \rightarrow \\
q_{k+1}(\mathrm{susp}) & \rightarrow (\mathrm{rsm}) r_{k+1} & \rightarrow (\mathrm{rsm}) s_{k+1} & \rightarrow (\mathrm{rsm}) p_{k+2} & \rightarrow
\end{array}$$

$goal\ (\mathrm{susp})$: $goal$ が中断

$(\mathrm{rsm})\ goal$: $goal$ が再開

このように q と r が 1 回メッセージをやり取りする度に、各々が 1 度ずつ中断してしまう。また、s と次世代の p は r に依存しているにもかかわらず r が中断した場合もスケジュールされてしまうため、再帰ループ 1 回あたり計 4 回のゴール中断が起こってしまう。

このようにプロセス指向スケジューリングでは、データの生成・参照ゴール間の切り替え時にゴール中断が生じるだけでなく、そのデータを間接的に必要とするゴールも多数中断する可能性がある。

3.1.2 resumption first スケジューリング

KLIC でとられているスケジューリングの方式は次の通りである。

- 基本的には左 / 深さ優先のゴールスケジューリングを行う
- ただし、同一化によって実行可能になったゴールは優先的にスケジュールされる

このように KLIC のスケジューリング方式は、resume されたゴールが最も優先的にスケジュールされる

```
main :- p0([int(100)|L1],L2),p1(L1,L2).
p0(L1,L2) :- L1=[int(N)|NL1],N>0
  | send0(NN,L2,NL2),dec0(N,NN),p0(NL1,NL2).
dec0(NI,NO) :- NO:=NI-1.
send0(N,L,NL) :- L=[NewN|NL],NewN = int(N).
p1(L1,L2) :- L2=[int(N)|NL2]
  | send1(NN,L1,NL1),dec1(N,NN),p1(NL1,NL2).
dec1(NI,NO) :- NO:=NI-1.
send1(N,L,NL) :- L=[NewN|NL],NewN = int(N).
```

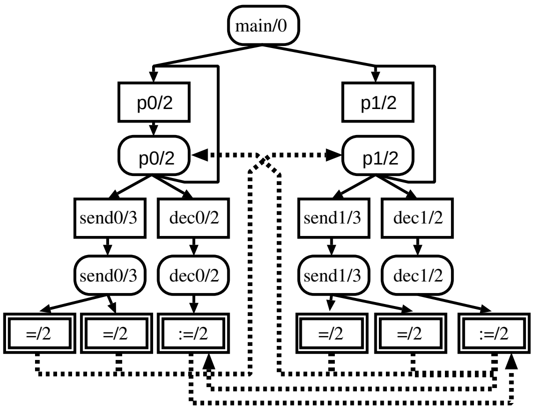


図 2 プログラム例 2

ことから resumption first スケジューリングと呼ばれる。

resumption first 方式を用いて、図 1の例をスケジュールすると、次のように実行される。

$$p_k \rightarrow r_k \rightarrow (\mathrm{rsm})\ q_k \rightarrow q_{k+1}\ (\mathrm{susp}) \rightarrow p_{k+1} \rightarrow \ldots$$

この場合は再帰 1 回あたり 1 回のゴール中断ですみ、プロセス指向スケジューリングよりよい結果が得られている。しかし、ゴールの中断の原因となった未具体化変数が複雑な構造データに具体化される場合には、データが完全なものとなる前に中断ゴールを再開してしまう可能性がある。この場合、再開されたゴールやそのサブゴールがデータ中の未具体化部分を参照し、再び中断してしまう恐れがある。

例えば、図 2の例では、ゴール `p0/2` は `p1/2` 側の 3 つのゴール (=/2,=/2,:=/2) にデータ依存している。よって中断した `p0/2` の実行を再開するには `p1/2` 側の上記 3 つのゴールすべてを実行する必要がある。

しかしながら resumption first scheduling では 3 つのゴールのうち "ゴール `L=New[N|NL]`" が実行されただけで制御が `p0/2` に移ってしまい、その結果 `p0/2` はメッセージに不完全なものを見つけ中断してしまう。この例では "ゴール `NewN=int(N)`" の実行においても同様のことが起こるため、 `p0/2` の 1 ループあたり計 3 回の中断が起こる。一方、これをプロセス指向スケジューリングでスケジュールを行うと、中断回数は 2 回 /1 ループとなる。

このように、 resumption first では構造型データの具体化が不完全である場合、不要な制御移行が生じる可能性がある。この結果、場合によってはプロセス指向スケジューリングより性能を悪化させてしまうことがよくある。

3.2 手法の概要

一般に並行 / 並列型プログラムのスケジューリングでは、逐次実行が最適であるようなシーケンスを発見し、それを逐次実行する最適化が重要である。例えば KLIC などこれまでの KL1 処理系では、プログラムを左 / 深さ優先に実行するのが逐次化の最適シーケンスと見なしている。しかし、プログラムの性質が、このような単純な推定には適合しないことが普通であり、前節でのべたような問題が生じる。

まず、プロセス指向スケジューリングで生じた問題点は中断するゴールにデータ依存するゴールをスケジュールしないようにすることで解決できる。

一方、 resumption first スケジューリングで生じた問題点は、プログラムがデッドロックしない範囲で、できるだけ制御移行を遅らせることで、多くの場合解決できる。

これらの解決方法は、各プログラムに応じた逐次化を行い、その逐次化されたプログラム断片を並列実行の単位とすることで実現が可能である。実際 KL1 プログラムでは、コンパイル時に依存解析を行うことによって、実行順序を静的に決定できるゴールも少なくない。そこで、コンパイル時に依存解析を行い、半順序の依存関係が成り立つゴールの集合を逐次実行が最適なゴールの系列と考える。このゴール系列を本稿ではスレッドと呼ぶ。本研究では、プログラムをスレッドの集まりと考えて、最適化を行なう。

各スレッドはそれぞれ途中でサスペンドすることはあっても、内部のゴール順序が入れ替わることはない。したがって、スレッド中のあるゴールで参照データが未具体化だった場合にはスレッドごとサスペンドして他のスレッドに実行を切り替えても、デッドロックを生じることはない。

また、スレッド内は最適な逐次実行を行うように静的にスケジューリングを行いつつ、さらに、スレッドを動的にスケジューリングすることで、元のプログラムが有する本質的な並行動作を実現し、データの依存関係を保つことが可能になる。

一方、このスレッド化を行うことによってプログラムの粒度が粗くなり、プログラムの並列性が低下するという新たな問題が生じる。スレッド間の応答性をあげることによって、並列性の低下による影響を減らすことが可能である。そこで、本研究では中断したスレッドを再開させるスレッドを優先的にスケジューリング方式を行い、並列性の低下による性能への影響を緩和する。

4. スケジューリング最適化手法

4.1 スレッド化

KL1 のプログラムではゴール間の依存には、次の 2

種類が存在する。

- 制御依存
  ゴール c が実行されるには、その前に c をボディゴールとして生成する親ゴール p が実行されていなければならない。
- データ依存
  あるゴール r が実行されるには、そのゴールで参照される値を具体化するゴール $w_1, \ldots, w_n$ がすべて実行されていなければならない。

現在 KLIC などでは、データ依存を考慮したスケジューリングを行っていない。このため、 r を w_i より先にスケジューリングしてしまうと、値を参照しようとした段階で中断し、 w_i の実行によりその値が具体化されると r の実行を再開する。これに対し、事前にデータ依存解析を行い、 w_i の実行後に r をスケジュールするようにすれば、この中断・再開に伴うオーバヘッドを避けることができる。

次にあるゴール p から複数のボディゴール $q_1, \ldots, q_n$ が生成される場合を考える。このスケジューリング順序を決定する方式として、次の 2 種類が考えられる。

(1) 兄弟ゴール $q_1, \ldots, q_n$ のみを静的スケジューリングの対象とする。
(2) $q_1, \ldots, q_n$ に加え、まだ実行されていない p の兄弟ゴールおよびその子孫もスケジューリング対象とする。

スレッド内のゴール間では中断による実行順序変更は生じないから、方式 1では、 q_i がスケジュールされると他の $q_j (j \neq i)$ は q_i の子孫の実行がすべて完了するまでスケジュールされない。したがって、ゴールの環境をスタックの形で保持することができ、速度効率の良い実装ができるという利点がある。

一方、方式 2では、従兄弟関係にあるようなゴール間でもスケジューリングが行われる。したがって、図 3のような例でも、全体を静的スケジューリング可能になる。このようなプログラムに対しては、方式 1では p と p' を別スレッドにせざるを得ない。

しかし、図 3のような見掛け上の並行プログラムを方式 1で静的スケジューリング可能な形に書き直すことは容易であり、実際にこうした記述がなされることは少ないと考えられる。そこで、本研究ではスタック実装による利点を重視して、方式 1を採用した。

4.1.1 スタック実装による利点

静的なスケジューリング対象を兄弟ゴール間のみとすることで、ゴールの実行環境をスタックを用いた実装が可能になる。

KLIC ではあらゆるデータ構造をヒープ上で管理し、そのヒープ領域がデータで一杯になるとガーベジコレクション (GC) をかけ、不必要となったデータを回収する。この GC にかかるコストは大きく、速度低下を引き起こす大きな要因の一つである。

しかし、ヒープ上におかれるデータのうちゴールの実行環境については、ゴールの実行が終了した時点でそのデータはゴミとなるため、短命であることが多い。一方、本方式では、スレッドの実行環境についてはスタックを用いた実装が可能であるため、短命なゴールの実行環境が引き起こす GC 回数を減少させることが可能になる。

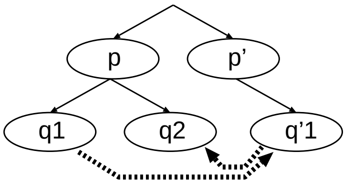


図 3 見掛け上の並行プログラム

4.2 スレッドの実行

本手法ではスレッド実行中に未具体化の入力引数が現れた場合は、その引数を具体化するのは他スレッドのゴールであることが保証されているため、スレッド自体を中断させる。この処理により、プロセス指向スケジューリングで生じた、中断ゴールの出力を待つゴールを無条件にスケジュールしてしまう、という問題点を避けることができる。

例えば、図 1のプログラムでは、図 4.(1) のようにスレッド化される。この結果、ゴール r_k の中断は thread1 全体の中断となるため、 s_k,p_{k+1} はスケジュールされずに $thread0$ に制御が移り、 s_k,p_{k+1} を中断することなく実行が行われる。

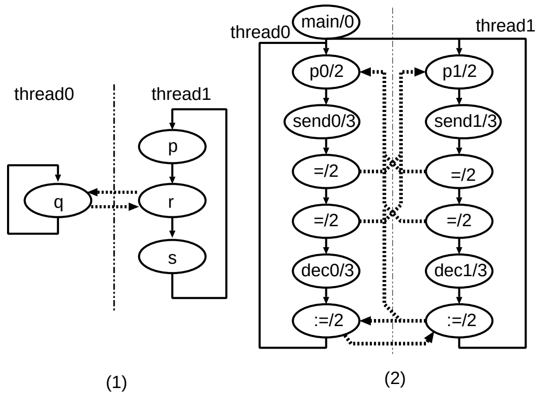
また、本手法ではあるスレッドが実行を開始すると、入力引数がそろっている限り、スレッド内のゴールを逐次的に実行する。これにより、プログラムがデッドロックしない範囲で制御が移行するのを遅らせることができ、 resumption first で起きた問題点を多くの場合避けることができる。

例えば、図 2のプログラムは、図 4.(2) のようにスレッド化される。このスレッド化によって、 `p0/2` の中断後、制御は thread1 に移り、ゴール `p1/2` から `:=/2` までをスレッドを切り替えることなく実行する。このため、 `p0/2` が次にスケジュールされたとき、 `p0/2` の必要とするデータはすべて具体化済みである。よって、 resumption first で起きた `p0/2` での不完全な具体化による中断は防止できている。

4.3 スレッドスケジューリング

実行時には、スレッドを逐次実行すると同時に、必要に応じてスレッド間の動的スケジューリングを行い、本質的な並行動作を実現する。

本手法では基本的にはスレッド単位で resumption first scheduling を行なうようなスケジューリング方針を採用する。この方法の利点としては、現在の KLIC スケジューラをゴール単位のものからスレッド単位のものに置き換えるだけで実装が可能であることや、スレッドの粒度が小さい場合にも従来の KLIC 処理系とほぼ同等



(1) (2)

図 4 プログラム例 1,2のスレッド化

の性能が得られることなどが上げられる。

一方、スレッド化による並列性の低下の問題については、中断スレッドを再開させるスレッドを優先的にスケジュールすることで対応する。

4.3.1 スレッドスケジューリングの最適化

本手法では並列実行の単位が従来のゴールからスレッドというより粒度の粗いものになったため、分散環境で本手法を用いる場合、並列性の低下が大きな問題となる。

例えば、あるノード (N_c) 上のスレッド (T_c) が他のノード (N_p) に対してメッセージの要求を出した場合を考えてみる。この場合 N_c は要求したデータが到着するまで中断するが、このとき N_c に実行可能なスレッドが存在しないと、ノード N_c は idle 状態になってしまい、プログラムの並列性が低下してしまう。特に本最適化の場合、要求メッセージが到着した際に N_p 上で実行されているスレッドが非常に大きくしかも、要求されたデータを作成しない場合、この N_c が idle でいる時間は非常に大きなものとなってしまう。

そこで本最適化では、この並列性の低下を防ぐために、スレッド化に利用したデータ依存情報を再利用し、スレッド (T_{susp}) が要求したデータを作成するスレッドを優先的にスケジューリングするようなスレッドスケジューリングの手法を提案する。

本手法を実現するにあたって、未具体化変数を具体化するゴールの特定が必要になるが、KL1 プログラムではモード解析5) を行なうことによって、ある変数 X の具体値を生成する可能性のあるゴールを特定できる。よってスレッド T_{susp} が未具体化変数 X を他ノードに対し要求した場合、この解析情報により X を具体化する可能性のあるスレッド T_{unify} を特定することが可能である。

次に本手法を実際のプログラムに適用する際に、留意する点として

(1) 複数のデータ生成要求がある場合。
(2) そのゴールを含むスレッドがそのゴールを実行する前に中断してしまう場合。

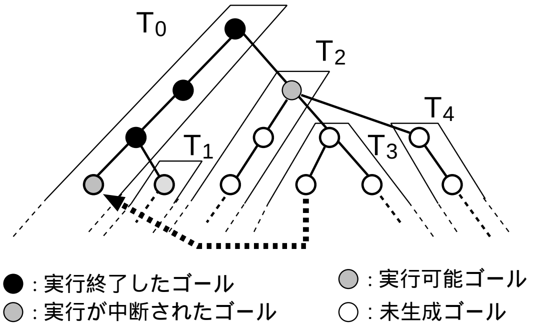


図 5 スレッドの動的スケジューリング

(3) そのゴールを含むスレッドがまだ生成されていない場合。

の 3 点があげられる。

1の場合は、先に到着した要求から順に処理することとし、後から到着した要求は前に要求されたデータが作成するスレッドが中断するか、データが生成されるまで処理されないこととする。

2の場合は、T_{susp} を実行可能にするスレッド T_{unify} が中断した場合、T_{unify} を実行可能にするスレッドを優先的にスケジュールするようにすれば問題はない。

この手法を実現するには、中断スレッド (T_{susp}) と、その中断原因となった未具体化変数を具体化するスレッド (T_{unify}) が同一ノード上にあるかどうかに関わらず、具体化を行うスレッドを優先的にスケジューリングするような手法を用いることで実現が可能である。

3の場合にとるべき方法としては、現在生成されているスレッドから要求データを作成するスレッドまでのスレッドの系列をスケジューリングすることが考えられる。

図 5の例では、スレッド T_0 を再開するスレッドは T_3 であるが、T_3 はまだ作成されておらず、スケジュールすることができない。この場合、本手法では T_1 よりも T_2 を優先的にスケジュールする。また T_2 の実行で作成されるスレッドのうち T_3 を T_4 よりも優先的にスケジュールする。

しかしながら、このようなスレッドのスケジューリングを行うとき、KL1 では実行時に同名のスレッドが複数の実体をもつ可能性があり静的な順序決めをすることは困難である。このため、我々はこのようなスケジュールを動的に実現する機構を導入した。

4.3.2 KLIC における実現

ここでは、中断スレッドを優先的にスケジュールする最適化機構を KLIC 上でどのように実現するかについて述べる。

- スレッドを新たに作成する際に、その初期ゴールの引数を調べ、出力変数 (X) が存在する場合は、その変数を generator object を用いて実現する。この場合の出力変数とはそのゴールもしくはそのゴールのサブゴールが具体化を行うような変数である。

このgenerator objectは新たに生成されたスレッドを内部データとして持ち、参照が起きた場合にスレッドを最高優先度でスケジュールするようなメソッドを持つ。具体的にはこのスレッドを最高優先度のスレッドキューの先頭につなぐことで実現する。この機構により具体化スレッドまたはその先祖のスレッドが最優先にスケジュールされる。

- generator objectに対する参照が起きると、現在実行していたスレッドはキューに戻され、generator objectが持つスレッドが実行を開始する。このときシステムは今どの未具体化変数(この例だとX)を具体化しようとしているかを覚えておき、Xを出力引数に持つスレッドが新たに作られる場合は、そのスレッドを最高優先度キューの先頭につなぐようにする。これにより、必ず最短経路で具体化スレッドがスケジュールされる。
- ある要求データを作成しているときに、別の要求があった場合は先に生成しているデータを作成するスレッドを最優先にスケジュールする。具体的には、あらたに要求されたデータを作成するスレッドは最高優先度のスレッドキューの最後尾にスケジュールすることで実現が可能である。

4.3.3 スレッドスケジューリング方針のまとめ

他ノードに要求されたデータを生成するスレッドは、並列性の低下の影響を最小限にするために、最優先にスケジューリングする必要がある。同様の理由で、要求データを生成するスレッドが中断した場合に、これをresumeするスレッドも最優先のスケジュールが必要である。一方、他ノードからのデータ要求が無い場合の、同一ノード内の中断ゴールをresumeするスレッドは、もし別のスレッドをスケジューリングしても並列性が低下することはないため上記2つの場合に比べて、優先度は低い。

以上のことから本最適化では、通常の左/深さ優先のスケジューリングに加え、

(1) 他ノードから要求されたデータを作るスレッド
(2) 1のスレッドが中断した場合に、それをresumeするスレッド
(3) 2以外の自ノード内の中断ゴールをresumeするスレッド
(4) resumeされたスレッド

を上の優先順位で優先的にスケジュールする。

5. 性能評価

本手法を図2のプログラムに対し適用し、実行時間・GC回数・中断回数のそれぞれを調べた結果を表1に示す。

実装対象はKLIC version 2.002で、ともにSun Ultra 1(167Mhz) 1台による逐次環境でトレース機能をOFFにしたものでの評価である。ただし、中断ゴールを優先的にスケジュールする最適化は未実装である。

表1 hand_shakeの性能評価

	実行時間(sec)	GC	中断回数
通常	21.40	2115	2000000
最適化	12.50	651	1000000

GC回数をみると、1/3に減少しており、スタックを用いたことによりメモリ効率が大幅に良くなっていることがわかる。一方、中断回数も1/2に減少しており、スレッド化による効果がはっきりとでている。これらの効果と、スタックを用いたことによる実行効率の上昇により、本手法を用いたことで、約1.7倍の速度向上が得られている。

6. おわりに

本論ではプログラムのスレッド化による最適化の手法とその性能評価、およびスレッドの最適なスケジューリング手法について述べた。

図2のプログラムでの性能評価の結果、スレッド化により中断回数が減少すること、またスレッドの実行環境の扱いをスタック化することによりメモリ効率および実行効率が向上することが確認できた。

今後の予定としては、大きなプログラムによる現システムの評価や、分散環境における評価、また4.3節で述べた最適化の実装および評価等があげられる。

謝辞　日頃御討論頂く富田研究室の諸氏に感謝致します。また数々の情報と助言を頂いた(財)先端情報技術研究所の方々に感謝致します。

参考文献

1) Takashi Chikayama, "Introduction to KL1", August. 1994.
2) Takashi Chikayama, "KLIC User's Manual", October. 1994.
3) Al Geist,Adam Beguelin,Jack Dongarra,Weicheng Jiang,Robert Manchek,Vaidy Sunderam, "PVM3 User's guide & reference manual", August. 1994.
4) 六沢 一昭, 仲瀬 明彦, 近山 隆, 藤瀬 哲朗, "KLIC分散メモリ処理系の設計と初期評価", JSPP'95,pp.153-160,1995.
5) Kazunori UEDA,Masao MORITA, "Moded Flat GHC and Its Message-Oriented Implementation Technique", New Generation Computing, 13(1994) p3-43.