ハイパフォーマンス
コンピューティング 79−11
（1999. 12. 3）

実行時再コンパイルによる並列プログラムのメモリ割り付け最適化

村井 均　荒木 拓也　松浦 健一郎　末広 謙二　妹尾 義樹
新情報処理開発機構 並列分散システム NEC 研究室

概要

本稿では、実行時に再コンパイルを行うことにより、並列プログラムのメモリ割り付けを最適化する手法を示す。コンパイル時には定まらない様々な要因のために、コンパイラの最適化には限界がある。そのような問題に対し、本手法は、実行時情報の採取, 再コンパイル判定, 再コンパイル実行, オブジェクト置換をプログラム実行時に行うことによって、最適な実行プログラムを得ることを目的とする。本手法は、プログラムサイズと適用範囲において従来手法より優れている。NEC Cenju-4 上で、実行時再コンパイルによるメモリ割り付けの最適化を適用した結果、1割程度の速度向上が得られた。

Memory Allocation Optimization for Parallel Programs by Dynamic Recompilation

Hitoshi MURAI, Takuya ARAKI, Ken-ichiro MATSUURA, Kenji SUEHIRO and Yoshiki SEO
Parallel and Distributed Systems NEC Laboratory, Real World Computing Partnership

Abstract

This paper describes an optimization technique for parallel programs by dynamic recompilation. Since it may not be possible to get information needed to determine the applicability and usefulness of a program transformation at compile-time, the optimizations of traditional compilers are limited. Our run-time system gathers program information, determines whether recompilation is necessary, executes recompilation and replaces the current object code with the newly generated one at run-time, to get the optimal code. The advantage of our technique is that any optimization can be applied without increasing program size so much. Evaluation on NEC Cenju-4 shows that memory allocation optimization by dynamic recompilation results in about 10% speedup as compared to the static optimization.

1 はじめに

あるプログラム変形の適用可能性や有効性はコンパイル時には判定できない場合があるため、コンパイラの行う最適化には限界がある。

特に、異機種の計算機から構成される異機種並列分散システム [1, 2] では、実行時のシステム環境が様々に変化し得るため、静的に最適なコードを得ることは難しい。

この問題に対し、我々の提案する実行時再コンパイル方式は、

1. 実行時情報の採取
2. 再コンパイル判定
3. 再コンパイル実行
4. オブジェクト置換

を実行時に行うことによって、対象プログラムの最適な実行プログラムを得ることを目的とする。

本稿では、特に、実行時再コンパイルによって並列プログラムのメモリ割り付けを最適化する手法について述べる。並列化コンパイラが、各プロセッサにデータを割り付ける方法としては、メモリサイズを優先する方法と速度を優先する方法の2種類が考えられる。各データに対する最適な割り付け方法は、実行時に利用可能なメモリサイズに依存するため、コンパイル時に決定することはできないが、実行時再コンパイルを用いれば、最適なメモリ割り付けを達成することが可能となる。

我々は、言語処理系技術をベースとするシームレスな異機種並列分散システム向けプログラミング環境の構築に取り組んでおり [3]、本稿で

提案する手法はこのプログラミング環境上に実装される。

以下、2章で従来手法について、3章で我々の提案する実行時再コンパイル手法について述べる。続いて、4章で評価結果を示した後、5章で結論を述べる。

2 従来手法

あるプログラム部分に対し、最適化方法の適用可能性や有効性をコンパイル時に判定できない場合がある。このような場合、通常のコンパイラは、「安全な」コードを生成しようとするため、プログラムの効率は低下する。このような場合においても、効率の良いプログラムを生成するために、次のような手法が提案されている。

- 条件最適化 (マルチバージョン)

  コンパイル時に最適なコンパイル方法を決定することが不可能な場合、起こり得る全ての場合の各々に対して最適なオブジェクトを用意しておき、実行時にそれらの中から最適なものを条件分岐によって選択して実行する手法である [4]。条件最適化によるベクトル化の例を図1に示す。

  この手法は、実行時に条件式の評価と条件分岐以外のオーバヘッドが必要ないことが特長であるが、多用するとプログラムサイズが非常に大きくなるという問題がある。

```
do i=1 to n
  a(i+k) = a(i) + b(i)
end do
```

↓ 条件最適化 (ベクトル化)

```
if (k >= n) then
  /* ベクトルコード */
  a(1+k:n+k) = a(1:n) + b(1:n)
else
  /* スカラコード */
  do i=1 to n
    a(i+k) = a(i) + b(i)
  end do
end if
```

図 1: 条件最適化の例

- 適応的変形

  複数の異なる最適化の結果を表現できるようなコードを生成する手法である。生成されたコードは、実行時情報に基づいて選択される方法で最適化されたコードとして振る舞う [5]。適応的変形によるループ融合の例を図2に示す。

```
do i=1 to n
  s1(i)
end do
do j=1 to n
  s2(j)
end do
```

↓ 適応的変形 (ループ融合)

```
    条件P (融合可能か) を設定
l1: while i <= n
      s1(i); i++
      if P then goto l2
    end while
l2: while j <= n
      s2(j); j++
      if P then goto l1
    end while
```

図 2: 適応的変形の例

また、ループのストリップマイニングやタイリングのような、パラメータ (ストリップマイニング長, タイルのサイズ) を持つ最適化手法において、パラメータの値を変数として変形を行い、実行時に最適なパラメータを求める、といった方法も適応的変形の一種と考えられる。

この手法は特定の最適化手法にしか適用できない。また、複数の最適化に対して適用するのは困難である。

- 部分的評価によるコード生成

  最適化方法に関わる変数の値がコンパイル時に確定しない場合、それらの値を確定しないまま不完全なオブジェクトを生成しておき、実行時に変数の値が確定したときに完全なオブジェクトを生成する。ロード・ストアの削減, ループアンローリング, 条件分岐の削除などに適用できる [6]。部分的評価によるループアンローリングの例を図3に示す。

  適応的変形と同様、この手法も特定の最適化手法にしか適用できず、複数の最適化に対して適用するのは困難である。

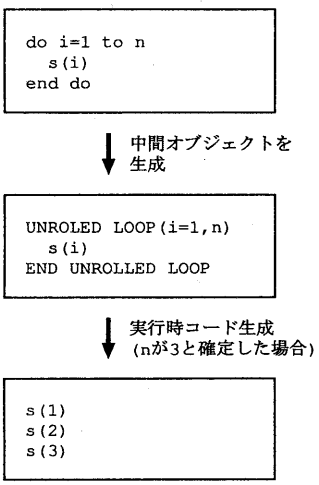

図 3: 部分的評価によるコード生成の例

3 実行時再コンパイル

3.1 実行時再コンパイルによる最適化

実行時再コンパイルによる最適化は、次の各ステップを実行時に行うことを特徴とする。

1. 実行時情報の採取
2. 再コンパイル判定
3. 再コンパイル実行
4. オブジェクト置換

実行時再コンパイルによる最適化を用いることで、従来方式に比べて以下のような利点が得られる。

- プログラムサイズ
  条件最適化方式のようにプログラムサイズが巨大になることはなく、適応的変形や部分的評価によるコード生成と同程度である。
- 適用対象
  あらゆる最適化手法に適用することができる。また、再コンパイルは手続き全体を対象とできるため、手続き全体に影響を及ぼすような変形も扱うことができる。

一方で、実行時情報の採取やオブジェクト置換のためのオーバヘッドが生じる。実行時再コンパイルによる最適化が特に有効に機能するのは、例えば以下のような最適化を行う場合である。

- メモリ割り付け
  3.2節で詳しく述べる。
- 手続きのクローニング
  手続き引数の取り得る値や分散状態の範囲が有限であることが確定すれば、それらの値や分散状態に特化した手続きをクローニング処理によって生成することができる。
- タスクスケジューリング
  処理系が動的スケジューリングの機構を持たない場合にも、システムの状態や各タスクの負荷を考慮して、実行時に最適なスケジュールを決定することができる。

3.2 実行時再コンパイルによるメモリ割り付けの最適化

本節では、実行時再コンパイルによる最適化の例として、分散メモリ型並列計算機における並列プログラムのメモリ割り付けについて述べる。

並列化コンパイラは、ソースプログラムに現れるデータ (配列) を、各プロセッサのローカルメモリ上に分割・配置する。このとき、宣言された配列の領域 (グローバル領域) のうち、各プロセッサに分割・配置されたローカルな領域だけが割り付けられる (SHRUNK 方式と呼ぶ。図 4(a))。ローカル領域上の配列要素のアドレスを得るには、実行時にグローバル領域上の添字からローカル領域上の添字への変換処理が必要になる。この添字変換処理は非常にコストが大きく、プログラム実行時間を増大させる原因になる。また、配列の分割・配置の方法を変更する際には、変更後の配列の形状に合う領域を動的に割り付けたり、元の領域から割り付けた領域へ値をコピーしたりする処理が必要になる。

一方、HPF/JA 仕様 [7] の定義する「フル SHADOW」のように、全プロセッサが配列の全体に等しい領域を保持し、そのうちのローカルな部分のみを用いる、という方法も考えられる (NOSHRUNK 方式と呼ぶ。図 4(b))。この方法では、添字変換や領域の動的割り付けなどの処理は不要になるが、SHRUNK 方式に比べ

ると、メモリの使用効率が悪く、巨大な配列を扱うプログラムに適用することはできない。

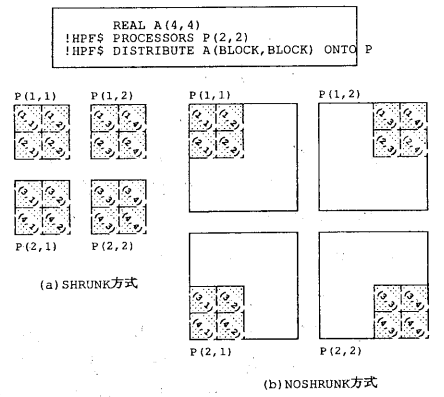

図 4: SHRUNK 方式と NOSHRUNK 方式

配列に対してメモリ割り付け方法を個別に決めることができる場合、対象の計算機システムの備えるメモリの範囲内でできるだけ多くの配列を NOSHRUNK 方式で割り付ける方がよい。このようなメモリ割り付け最適化問題は次のように定式化できる。

メモリ割り付け最適化問題　プログラム中で使われる配列 $A_i(i = 1,2,\cdots,n)$ を SHRUNK 方式および NOSHRUNK 方式で割り付ける場合に必要となるメモリサイズをそれぞれ m_i および M_i、配列 A_i の重み (A_i を SHRUNK 方式で割り付けたときに生じるオーバヘッドの合計) を w_i とし、$I \equiv \{i = 1,2,\cdots n\}$ としたとき、

$$\sum_{i \in I - I_{ns}} m_i + \sum_{i \in I_{ns}} M_i \leq M_{all}$$

の元で、

$$W \equiv \sum_{i \in I_{ns}} w_i$$

を最大にするような $I_{ns}(\subseteq I)$ を求める。ただし、M_{all} は利用可能なメモリサイズである。

この問題は、ナップザック問題に帰着できる。ナップザック問題は NP であるが、多項式時間内で近似解を得る方法は多く知られている [8]。

利用可能なメモリサイズ M_{all} は実行環境によって様々に変化し得るため、最適なメモリ割り付け方法 I_{ns} をコンパイル時に決定することはできないが、実行時再コンパイルを用いれば、次のように実行時にメモリ割り付けを最適化することが可能となる。

1. すべての配列を SHRUNK 方式で割り付けるようにコンパイルしておく。
2. 実行時に利用可能なメモリサイズ M_{all} が得られれば、それに基づいて最適なメモリ割り付け I_{ns} を計算する。
3. 実行時再コンパイルによって、I_{ns} を実現するオブジェクトを生成する。

ただし、実際には、最適解 I_{ns} を得ることは困難なので、例えば greedy アルゴリズムで得た近似解を用いる。また、正確な値を得ることが難しい w_i は、w_i におおよそ比例する A_i のアクセス回数で代用する。

3.3　実行時再コンパイル機能の実現

実行時再コンパイル機能は、現在開発中である異機種並列分散システム向けプログラミング環境の機能の一つである [3]。このプログラミング環境は次の 3 つの構成要素から成る。

- シームレス並列分散ライブラリ
  異機種並列分散システムにおいて、シームレスな通信を実現する。
- シームレス並列分散化コンパイラ
  異機種並列分散システム向けの独自拡張を含む HPF 言語を処理する。配列に対するメモリ割り付け方法を個別に扱う機能を有する。
- シームレス並列分散化支援ツール
  GUI ベースのプログラミング支援ツールであり、逐次 Fortran プログラムの並列分散化を支援する。シームレス並列分散コンパイラと連携してプログラム実行時の各種情報を採取するためのフレームワークが実装されている。

実行時再コンパイルの各ステップは、これらの構成要素の提供する種々の機能を利用して実現される。以下、実行時再コンパイルの各ステップの実現方法について順に説明する。

プログラム生成　シームレス並列分散化コンパイラ (以下コンパイラ) は入力プログラムを解析し、実行時再コンパイルによる最適化の対象部分，最適化の種類，実行時再コンパイルを起動する条件を求める。対象部分はユーザが指定す

ることも可能である。実行時再コンパイルの対象手続きは、図5に示すように、中継ルーチン dcaller を介して呼ばれるように変形される。また、採取すべき実行時情報をシームレス並列分散化支援ツール(以下ツール)に指示する。

実行時情報の採取　ツールは、コンパイラの指示に従って、プログラム実行時に、再コンパイル判定および最適化に必要な実行時情報を採取する。

再コンパイル判定　実行中のプログラムは、採取された実行時情報より、実行時再コンパイルが必要か否かを判定する。

再コンパイル実行　再コンパイルが必要と判断されると、ツールはコンパイラを起動する。コンパイラは、ツールから受け取った実行時情報を元に最適化されたオブジェクトを生成する。

オブジェクト置換　実行時再コンパイルによる最適化を実現するためには、再コンパイルを行うだけではなく、プログラムの実行を停止することなく、オブジェクトの一部を置換する必要がある。

中継ルーチン dcaller は、対象手続きを呼ぶ前に、その手続きが再コンパイルされているか否かをチェックする。再コンパイルされていた場合は、オブジェクトの置換を実行する(図5)。今回は、GNU の BFD ライブラリを用いて、実行時再コンパイルの対象とする手続きを含むオブジェクトファイルをロードし、手続きのポインタを置換する、という方法をとっている。他に、ダイナミックリンクの機能を利用した実装も考えられる。

以上のステップのうち、実行時情報の採取, 実行時再コンパイル判定, オブジェクト置換は、プログラムの実行に対してオーバヘッドとなる。したがって、実行時再コンパイルによる最適化は、性能に及ぼす影響の大きい、時間発展ループなどの何度も繰り返される処理において特に有効である。再コンパイル処理は、実際にはプログラムの実行とオーバラップして行われるので、プログラムのオーバヘッドにはならない。

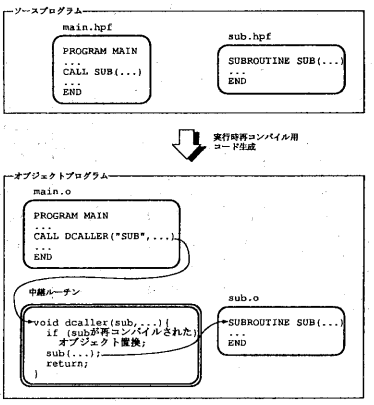

図 5: 実行時再コンパイル用コード

4　評価

実行時再コンパイルによるメモリ割り付けの最適化を、NEC Cenju-4 上で評価した。ターゲットプログラムは、NAS Parallel Benchmarks(NPB) 1.0[9] の SP(クラス A) である。コンパイルは、上記のシームレス並列分散コンパイラおよび FORTRAN77/UX による。

実行時情報の採取および再コンパイル判定の機能は未実装であるため、400 回転の時間発展ループのうち100回目の繰り返しにおいて固定的にオブジェクトの置換を実行するようにした。したがって、実行時情報の採取に要する時間は今回の評価結果には含まれていない。また、再コンパイル処理に要する時間はプログラムの実行とオーバラップされるものとして、評価結果に含めていない。メモリ割り付け最適化の対象は、時間発展ループ内から繰り返し呼び出される手続き etasweep, ztasweep,xisweep のローカル配列 a,b,c,d,e (65×65×64 の REAL*8 型 3 次元配列) とした。

その結果を表 1および図6に示す。値は NPB で定められた測定時間 (400 回転に要した時間) である。「再コンパイル」は実行時再コンパイルによる最適化を適用したときの結果を、「SHRUNK」「NOSHRUNK」はすべての配列をそれぞれ SHRUNK 方式と NOSHRUNK 方式で割り付けたときの結果である。

また、実行時再コンパイル機能を実現するためのオーバヘッド (中継ルーチン dcaller の呼び出しやオブジェクト置換のための BFD ライブラリ呼び出しなど) は、400 回転で 7 秒程度

表 1: 実行時間とメモリサイズ

PE数	再コンパイル		SHRUNK		NOSHRUNK	
	実行時間 (sec)	メモリ (MB)	実行時間 (sec)	メモリ (MB)	実行時間 (sec)	メモリ (MB)
1	4054.6	48	4643.8	48	2671.9	48
2	2178.2	29	2476.1	24	1445.9	48
4	1194.6	19.5	1321.5	12	785.2	48
8	650.1	14.8	702.4	6	420.3	48
16	370.2	12.4	395.2	3	224.7	48
32	218.6	11.2	225.8	1.5	124.0	48

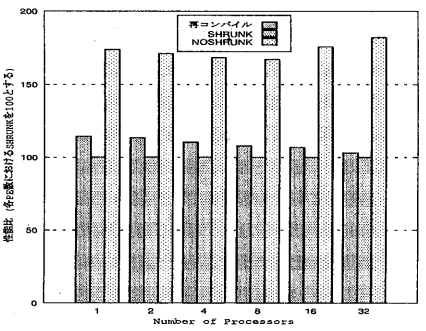

図 6: 性能比

(32 並列時で全実行時間の約 3%) であった。

表 1および図 6より、実行時に再コンパイルを行って一部のデータのメモリ割り付け方法を変更することで、1 割程度の速度向上が得られたことがわかる。このとき、必要なメモリサイズは増加するが、NOSHRUNK 方式の場合に比べると小さく抑えられている。また、再コンパイル機能を実現するためのオーバヘッドは 32 並列時で全実行時間の約 3% とわずかであった。以上より、提案した手法を用いれば、実行時にメモリ割り付けを最適化することで、プログラムの性能を向上させることが可能であるといえる。

5 まとめ

実行時に再コンパイルを行うことで、プログラムの最適化を行う手法について述べた。特に、並列プログラムのメモリ割り付けの最適化に対して提案した手法を適用し、その有効性を確認した。

今後は、プログラム生成からオブジェクト置換までの全てのステップを自動的に実行する機能を、異機種並列分散システム向けプログラミング環境上に実現するとともに、メモリ割り付け以外の最適化も扱えるよう機能拡張を行い、提案手法の有効性の検証を進める予定である。

参考文献

[1] 草野ほか, 異機種並列分散処理による高性能計算機システムの構想, SWoPP97, 信学技報 CPSY97-63, pp.91-96, 1997.

[2] 片山ほか, 高性能並列分散システムの設計, 1998 RWC シンポジウム, RWC Technical Report (TR-98001), pp.41-46, 1998.

[3] 末広ほか, 異機種並列分散システム向けプログラミング環境, SWoPP'99, 情処研報 Vol.99, No.66, pp.77-82, 1999.

[4] M. Byler *et al.*, Multiple Version Loops, International Conf. on Parallel Processing, pages 312-318, August 1987.

[5] R. Gupta *et al.*, Adaptive Loop Transformations for Scientific Programs, IEEE Symposium on Parallel and Distributed Processing, pages 368-375, San Antonio, Texas, October 1995.

[6] J. Auslander *et al.*, Fast, Effective Dynamic Compilation, Conference on Programming Language Design and Implementation, May 1996.

[7] High Performance Fortran Forum, 財団法人高度情報科学技術研究機構, High Performance Fortran 2.0 公式マニュアル, pp.311-381, シュプリンガー・フェアラーク東京, 1999, `http://www.tokyo.rist.or.jp/jahpf/spec/jahpf-j.html`.

[8] S. Sahni, Approximate algorithms for the 0/1 knapsack problem, Journal of the ACM, vol. 22, no. 1, pp. 115-124, January 1975.

[9] D. Bailey *et al.*, THE NAS PARALLEL BENCHMARKS, RNR Technical Report RNR-94-007, Mar 1994.