プログラミング 6－4
（1996. 3. 26）

データ駆動の手法を採り入れたデータパラレル・コンパイル方式

大谷 浩司* 安村 通晃†
*慶應義塾大学湘南藤沢メディアセンター
†慶應義塾大学環境情報学部
連絡先:〒252 藤沢市遠藤 5322 慶應義塾大学環境情報学部安村研究室
キーワード: 超並列, Fortran, コンパイラ, SPMD

あらまし

分散メモリ型の並列計算機上の並列言語コンパイラの生成コードには、データパラレルの並列実行を中心としたデータパラレル方式と、汎用の並列実行を行なうデータ駆動方式などがある。データパラレル方式は、データアクセスの簡単なループの場合には非常に効率が良いが、そうでない場合には性能が著しく低下する。そこで、データパラレル方式を基本として、複雑なループの場合にデータ駆動方式の手法を用いて性能を向上させるデータ駆動データパラレル方式を提案する。本稿では、方式について説明したのち実際のマシンで実行してその有効性を示す。

Data Driven and Data Parallel Combined Compiling Method

Koji Ohtani* Michiaki Yasumura†
*Keio University Shonan Fujisawa Mediacenter
†Keio University Institute of Environmental Information
Address:Keio University Yasumura Lab. 5322 Endo Fujisawa shi.
Keyword:Massively parallel, Fortran, Compiler, SPMD

Abstract

One of the implementation methods of parallel programing languages for distributed memory parallel machines is the Data-Parallel Method which parallelizes do-loops containing data parallelism. Another implementation method is the Data Driven method which exploits general parallelism in programs. Data parallel method generates very effecient code for regular access loop. But, for irregular access loops it can only generate ineffecient code. We propose the Data Driven Data Parallel method which is based on the Data Parallel method but uses Data Driven technique for irregular loops. In this paper we describe Data Driven Data Parallel method and demonstrate its usefuleness by presenting performance results on a real parallel machine.

1 はじめに

HPF(High Performance Fortran)、Fortran-D[1]、Vienna Fortran、Distributed Fortran90などの分散メモリ型の超並列計算機上の並列Fortranの実装では、データに関する並列性に注目したデータパラレル方式が多く採用されている。データパラレル方式では、配列の要素を複数のプロセッサに割り当て配列の各要素に対しほとんど同一の演算を行なうDOループを並列に実行する。演算のためのデータは必要に応じて通信する。一方、マルチスレッドを用いたデータ駆動方式のコード生成の方法も提案され主に関数型言語やデータフロー型言語の実装に使用されている。この方式では、プログラムを複数のスレッドに分割し、必要なデータが揃ったスレッドから実行可能となる。

データ駆動方式では、動的にスレッドの起動やスケジューリング等を行なうために、非常に柔軟な並列動作を行なうことができ、データパラレルだけでなくコントロールパラレルなどプログラムに内在する多様な並列性を引き出すことができる。反面、メモリ管理やスケジューリングなどのオーバヘッドが大きい、データやプログラムのアクセスの局所性が失われキャッシュなどの性能が低下するという欠点がある。

データパラレル方式では、コンパイラの解析により、静的なスケジューリング、静的なメモリ管理およびメッセージ・ベクトル化などの通信の最適化を行なうことにより、効率の良い実行を行なうことができる。しかし、コンパイラが解析困難なデータアクセスが生じた場合には、並列化が不可能であるか、可能であっても性能が大きく低下する。

科学計算では、データパラレルの並列性が大きくFortranの実装では効率の良さによりデータパラレル方式が採用されている。しかし、科学計算においても疎行列や不規則な構造のシュミレーションなどで、配列の添字が複雑になり、従来のデータパラレル方式ではうまく対応できない。そこで、本稿ではデータパラレル方式を基本として、データ駆動の方式を採り入れることにより、従来の方式では困難な場合の実行性能を改善することを提案する。この方式をデータ駆動・データパラレル方式と呼ぶことにする。また、データ駆動方式は、オーバヘッドが多いと考えられているが、使用を限定し最適化を行なうことにより従来のデータパラレル方式に劣らない効率を実現できることを示す。

```
do i=1,N-1
  a(i) = b(ib(i+1)) + c(ic(i+1))
end do
```

図 1: 添字が複雑なループ

以下では、まずデータ駆動データパラレル方式の概要を説明し、次に実際の計算機に実装する方法を考える。さらに、最適化について述べた後、本方式を実装したプロトタイプ・コンパイラの生成したコードと、従来に方式によるコードの性能を比較する。

2 データ駆動データパラレル

2.1 従来の方式

図1のプログラムを見てみよう。ib、icは大きさがNの整数の配列でBLOCK分割されているとする。b、cの添字の値は実行時になるまで不明であり、通信のパターンは複雑になり実行時になるまで決定できない。しかし、ループの各イテレーションで配列の参照の際に通信を行なうのは効率的でない。そこで、このような添字が複雑になったループに対しては、ループに入るまえに実行時に添字を解析し通信を行なうinspector/executor方式が提案されている。[2][3] この方式では、まずinspectorにおいて添字を解析し、配列の要素の通信のパターン(communication scheduleと呼ばれる)を作成する。次にexecutorでは、作成されたscheduleに基づいて配列の要素の通信、ループ内演算の実行を行なう。

inspector/executor方式には、以下のような問題がある。

- イテレーション間に依存があるループには適用できない。
- inspector自身の処理がかなり多い。
- 最終的な通信のパターンを求めるために行なう通信が、大量に必要となる。
- 添字がさらに複雑な場合、各プロセッサpが参照する配列の添字の値の集合を求めるのにさらに別のinpector/executorが必要になる。

2.2 データ駆動・データパラレル方式

複雑な添字を持つループを並列化した場合、通信のスケジュールを作成する処理やメッセージを組み立てたり、メッセージからデータを取り出したりする処理に多くの時間を要する。また、inspector/executorのような手法では、送信と受信のフェーズが明確に分離しているために、受信時に必要メッセージが到着していなければ、プロセッサ資源が無駄になる。複雑なアクセスを含むループでは、各フェーズに要する時間は各プロセッサで異なるために、このような無駄が生じることも多い。従って、実行時間を向上させるためには、このような処理も積極的に通信レイテンシの隠蔽や並列度の上昇に利用することが重要になる。

また、従来の方法ではコンパイル時に所有プロセッサと必要とするプロセッサが決定できないデータを移動するのに、データの受取側から要求する立場をとっている。配列要素 a(i) を所有するプロセッサを owner(a(i)) と表すことにすると、この方法では、ひとつの b(ib(i+1)) のデータを演算を実行するプロセッサ owner(a(i)) に渡すためには、owner(ib(i+1)) − > owner(a(i) − > owner(b(ib(i+1)) − > owner(a(i)) の最悪 3 回の通信が必要になる。一方、データの所有者側に立ったアプローチをとった場合、自分の持つデータを次々と渡して行くという方法を使用すれば owner(ib(i+1)) − > owner(b(ib(i+1)) − > owner(a(i)) と通信を 2 回に減らすことができる。しかし、後者のアプローチでは、中間のプロセッサはメッセージをいくつ受け取るかは分からない。メッセージの数が分からなければ、静的に処理をスケジューリングすることはできない。そのため、この様なメッセージの処理は動的に起動する必要がある。

そこで、以上の問題点を解決するためにデータ駆動データパラレル方式を提案する。

ループを以下のような処理を含むスレッドに分解する。ここで、スレッドとは途中で受信や同期を必要としない一連のコードのことである。必ずしもスレッド毎にスタックなどの実行コンテクストを持つ必要はない。

(1) メッセージ・ベクトル化を施した後の複数のイテレーション

(2) 配列の添字を求めるコード。複数のスレッドに分かれることもある。

(3) 必要なデータを準備してメッセージに組み立てるコード

(4) 受信したデータを所定のアドレスに書き込むコード

スレッドを実行するためには、開始時に参照するすべてのデータが揃っている必要がある。参照するすべてのデータが揃っているスレッドを実行可能スレッド、ループ開始時に実行可能であることがコンパイル時に分かっているスレッドを常時実行可能スレッドと呼ぶことにする。

メッセージは、スレッドに対して送信される。ただし、同一プロセッサに対するメッセージは、実際には組み立てられず、直接に渡される。メッセージ・ベクトル化でまとめられたメッセージは、本方式においても単一のメッセージとして扱う。あるスレッドの処理が終了すれば、実行可能になることがコンパイル時に分かっているスレッドは、前のスレッドと融合してひとつのスレッドにする。逆に、処理時間が長過ぎるスレッドは並列性を損なう可能性があるので、場合によっては分割する。

スレッドの実行は、メッセージの到着以外では中断されないものとする。

これらのスレッドを以下の手順で実行する。

(1) 常時実行可能スレッドのうちコード中に送信を含むものを実行

(2) 残りの常時実行可能スレッドを実行

(3) 割り当てイテレーションをすべて終了するまで実行可能になったスレッドを実行

手順 (3) では、メッセージで届けられたデータによって次々とスレッドが実行可能になり、データ駆動により処理が進む。手順 (2) では、手順 (1) で行なわれた通信とオーバラップさせることにより通信レイテンシの隠蔽を行なう。

この方式では、データ駆動の手法を採用したことによって以下のような利点がある。

(1) 送信先を決定する演算やアドレス演算、メッセージの組み立てのような処理も通信とオーバラップさせることができる。— 通信レイテンシの隠蔽

(2) inspector に相当する処理と executor に相当する処理をオーバラップさせることができる。— 並列度の上昇

(3) 通信のパターンの自由度が高く、通信回数の少ない方法を選ぶことができる。— 通信量の削減

(4) イテレーション間に依存があっても適用できる。— 並列化の向上

一方、データパラレル方式を基本とし、データ駆動の処理をデータパラレルのループの処理に限定することによって以下の様な利点が生じる。

(1) アクティベーション・フレームのような特別な構造や記憶領域を必要としないために、メモリ管理のオーバヘッドが小さい。

(2) スレッドの制限のためにスケジューリングの処理が非常に簡単で軽い。

(3) ループ内でスレッドをまとめてあるために、データアクセスに局所性があり、キャッシュが有効である。

(5) 従来の方法で使用できるメッセージ・ベクトル化などの最適化を利用できる。

(4) 簡単なループや、ループ以外では余分なオーバヘッドがない。

2.3 適用例

では、次に図 11のプログラムに適用してみよう。

イテレーションの本体を実行するプロセッサを演算プロセッサと呼ぶことにする。

inspector/executor 方式と異なり、配列 b、c の添字の値 ib(i+1)、ic(i+1) は演算プロセッサに集めることをせず、b(ib(i+1))、c(ic(i+1)) の要素の所有プロセッサに直接送ることとして、通信回数を減らすことにする。

このプログラムでは、スレッドは以下のコードになる。

(t1) ib(i+1) の値を b(ib(i+1)) の所有プロセッサに送信するコード

(t2) ic(i+1) の値を c(ic(i+1)) の所有プロセッサに送信するコード

(t3) 添字の値 ib(i+1) を受信して、b(ib(i+1)) を演算プロセッサに送信するコード

(t4) 添字の値 ic(i+1) を受信して、c(ic(i+1)) を演算プロセッサに送信するコード

(t5) b(ib(i+1)) を受信するコード

(t6) c(ic(i+1)) を受信するコード

(t7) 担当するイテレーション本体の演算を行なうコード

t1,t2 が常時実行可能スレッドであり、このスレッドの実行から処理を開始する。

3 実装

本章では、本方式を富士通 (株) 社製の高並列計算機 AP1000[4] 上で適用する。今回は、試験的な評価のための実装であり AP1000 上の OS CellOS 1.3.1 で提供されているライブラリのみを使用して、C 言語で実装しアセンブラコードなどは使用しない。

AP1000 ではメッセージの到着でコードを起動することはできない。そのため、スレッドのスケジューリングは主に受信のポーリングと switch 文で行なう。

割り当てイテレーションの終了の検出は、そのためのカウンタを設け最初に割り当てイテレーション数に初期化し、イテレーションが終了する毎に減算することにより行なう。

自身のイテレーションが終了していても、他のプロセッサのイテレーションのためのスレッドは起動される可能性がある。スレッドの起動をポーリングと switch 文で行なっているために全プロセッサのイテレーションが終了するまで、ポーリングを続けなければいけない。そのため、全プロセッサのイテレーションの終了を検出する必要がある。ブロックしてしまうバリア同期は使用できないため、AP1000 のステータス機能を使用することにする。

図 1のコードは図 2のようになる。イテレーションの本体 t7 はポーリングの停止後に実行することにする。

4 最適化

さらに、最適化として以下のものを行なう。

4.1 通信回数の削減

通信の時間には、通信データの大きさに関わらない時間がある。この時間が AP1000 の様に比較的大きい場合、少々のオーバヘッドを生じても、送信デー

```
イテレーション・カウンタの設定
ステータスのセット
t1,t2 の実行
while (ステータスの論理和がゼロでない) {
  if (メッセージあり) {
    switch (メッセージの宛先スレッド) {
    case t3: t3 の実行
      break;
    case t4: t4 の実行
      break;
    case t5: t5 の実行
      break;
    case t6: t6 の実行
      break;
    }
  }
}
t7 の実行
```

図 2: 図 1の本方式によるコード

タをブロック化して通信回数を削減するのが得策である。そこで、メッセージは直ちに送信せず、バッファに蓄えてある程度の長さにブロック化して送信する。

4.2 スレッドのインライン展開

スレッド t1 で、ib(i+1) の所有プロセッサと、b(ib(i+1)) の所有プロセッサが同じ場合、通信を経ずにスレッド t3 を実行できる。そこで、この様な場合スケジュールをせずに t3 を t1 の中にインライン展開して、スケジューリングや分岐のオーバヘッドを削減する。この手法は、t2 と t4、t3 と t5、t4 と t6 の間にも適用できる。

4.3 イテレーションの範囲によるスレッドの分解

t1 に t3、t5 をインライン展開したものに注目する。ib(i+1) の後ろの方のブロック境界以外のでは、ib(i+1) の所有プロセッサと演算プロセッサが同一であることはコンパイル時に決定できる。従って、このブロック境界用とそれ以外用にスレッドを分ければ、t1 に埋め込まれた t3 中の ib(i+1) の所有プロセッサと演算プロセッサの同一性を判断するコードが削減できる。この手法は、t2 にも適用できる。

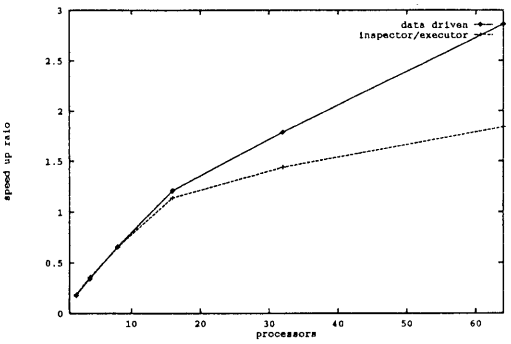

図 3: スピード・アップ

5 性能評価

AP1000 上で図 1を本方式によるコードと、既存の inspector/executor 方式によるコードを実行して速さを評価した。

本方式によるコードは、作成中のプロトタイプ・コンパイラによって生成されたコードである。inspector/executor 方式によるコードは、人手によって作成したコードである。send_set の作成の際には、全対全のプロセッサ通信ではなく CHAOS[3] と同様、要求しているプロセッサの数を数える手法を採用して通信量を削減している。

最初に通信に対する影響を見るため図 1中の N は 6400 と比較的小さく、a、b、c は整数の配列とする。また、ib(i)、ic(i) の値は i とした。通信はラインセンドモードを使用した。

逐次プログラムをひとつのプロセッサで実行した時の性能を 1 とするスピードアップのグラフを図 3 に示す。

グラフ中 inspector/executor とあるのが従来の方式、data driven とあるのが本方式によるコードの結果である。

グラフをみると、絶対的な性能は 64 プロセッサでも 3 以下とあまり良くない。イテレーション中の演算を整数の加算という非常に軽いものにし、データ数を大きくしなかったために、演算に対する通信のオーバヘッドが大きいためであると思われる。

また、AP1000 のライブラリを使う送信のためにかかる時間を測定するとラインセンド・モードで $6.6 + 0.05x\mu sec$ (x は送信バイト数) である。先に述べた

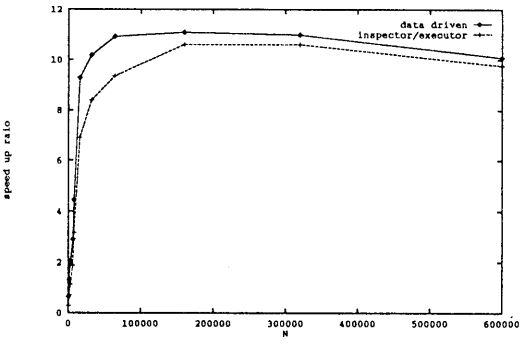

図 4: 要素数とスピード・アップの関係

通信時間 $18+0.05x\mu sec$ と比較すると、$11.4\mu sec$ しか、演算とオーバラップできない。さらに、受信のための時間を測定すると、$22+0.019x\mu sec$ である。送信、受信を合わせて考えると現在のライブラリを使用している限り、通信レイテンシの隠蔽はほとんど期待できない。このため、本方式のレイテンシ隠蔽の効果は出ていない。inspector/executor による方法でも local_iter を使ったレイテンシ隠蔽の効果を狙ったコードが存在するが、この効果はないのでイテレーションを local と nonlocal に分ける処理は純粋なオーバヘッドになってしまう。そのため、評価に使った inspector/executor のコードではこの処理は行なっていない。

従来の方式と、本方式のグラフを比較すると、プロセッサ数が少ない時には本方式は、既存の方式とあまり差がない。しかし、プロセッサ数が増加すると従来の方式の性能は伸び悩むが、本方式ではさらに上昇している。これは、従来の inspector/executor 方式では send_set を作成するのに多くの時間を要するためである。

次に配列の要素数を 640 から 600000 まで変化させた時のスピードアップを見てみる。プロセッサ数は 64 に固定する。図 4にその結果のグラフを示す。図 3と同様、inspector/executor とあるのが従来の方式、data driven とあるのが本方式によるコードの結果である。このグラフを見ると、本方式では、要素数の少ないところで、急勾配で上昇し従来の方式を大きく上回る。そして、要素数があるところまで増加すると、従来の方式に近付く。従来方式の send_set の作成処理に要する時間は、要素数に無関係の部分が多く数が少ないところで影響が大きく出ているためである。

6 結論

本稿では、複雑な配列アクセスを含むループの並列化に対し、データ駆動データパラレル方式を提案し試験的に実装し性能を評価した。通信レイテンシの隠蔽を行なうことができないなどのマシン上の制約のため大幅な高速化を達成することはできなかったが、オーバヘッドの大きな実装法にもかかわらず、従来法を若干上回る性能を実現することができた。しかし、実際の問題に適用するためには、現在の採用しているライブラリによるポーリングと switch 文、ステータスを使用したスケジューリング法はオーバヘッドが大きい。より本格的な実装を行なうとともに、絶対的な性能を引き上げることが課題となる。

参考文献

[1] Hiranandani, S., Kennedy, K. and Tseng, C.W., Compiling Fortran D for MIMD Distributed-Memory Machines, *Communications of the ACM*, Vol.35, No.8, Aug., 1992, pp.66-80

[2] Koelbel, C. and Mehrotra, P., Compiling Global Name-Space Parallel Loops for Distributed Execution, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.4, Oct., 1991, pp.440-451

[3] Ponnusamy, R., Saltz, J., Choudhary A., Hwang, Y.S. and Fox F., Runtime Support and Compilation Methods for User-Specified Irregular Data Distributions, *IEEE Trans. Parallel and Distributed Systems*, Vol.6, No.8, Aug., 1995, pp. 815-831

[4] Ishihata, H., Horie, T. and Shimizu, T., Architecture for the AP1000 Highly Parallel Computer, *Fujitu SCIENTIFIC & TECHNICAL JOURNAL*, Spring 1993, Vol.29, No.1, pp.6-14