JAERI-Data/Code
97-007

粒子コードの共有メモリ型ベクトル並列計算機での並列化

1997年3月

大田敏郎・折居茂夫

日本原子力研究所
Japan Atomic Energy Research Institute

本レポートは，日本原子力研究所が不定期に公刊している研究報告書です。

入手の問合わせは，日本原子力研究所研究情報部研究情報課（〒319-11 茨城県那珂郡東海村）あて，お申し越しください。なお，このほかに財団法人原子力弘済会資料センター（〒319-11 茨城県那珂郡東海村日本原子力研究所内）で複写による実費頒布をおこなっております。

This report is issued irregularly.

Inquiries about availability of the reports should be addressed to Research Information Division, Department of Intellectual Resources, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibaraki-ken 319-11, Japan.

© Japan Atomic Energy Research Institute, 1997

編集兼発行　日本原子力研究所
印　　刷　㈱原子力資料サービス

JAERI－Data/Code 97－007

粒子コードの共有メモリ型ベクトル並列計算機での並列化

日本原子力研究所計算科学技術推進センター
大田　敏郎・折居　茂夫

（1997年2月10日受理）

粒子コードを共有メモリ型並列計算機T90の並列化コンパイラを用いて並列化した。
その結果，自動並列化では十分な性能向上が得られなかったが，並列化のために用意されたディレクティブを挿入すること，およびある一つの有効なベクトル並列化手法を使用することによって十分な性能向上を得ることができ，4CPU使用時にオリジナルコード比で約3.2倍の加速率が得られた。
性能向上作業の過程とその効果について共有メモリ型ベクトル計算機の特徴と共に述べる。

日本原子力研究所：〒153　東京都目黒区中目黒2－2－54

Parallelization of a Particle Code on Shared Memory Parallel Vector Computer

Toshio OHTA and Shigeo ORII

Center for Promotion of Computational Science and Engineering
Japan Atomic Energy Research Institute
Nakameguro, Meguro-ku, Tokyo-to

(Received February 10, 1997)

A particle code was parallelized by the parallel compiler on the shared memory vector computer T90.

The auto-parallelization compiler was found not effective for this code. But the works of inserting parallel directives and using a particular method for vectorization and parallelization make this code about 3.2 times faster than the original performance.

The results will be shown with the tuning process and its effect as well as the characteristics of the shared memory parallel computer.

Keywords: Parallel, Vectorization, Autotasking, Shared Memory, Particle Code

目　　次

1. はじめに …… 1
2. 共有メモリ機と分散メモリ機 …… 2
3. 並列計算機 CRAY T90 …… 4
  3.1 ハードウェア …… 4
  3.2 ソフトウェア …… 4
4. 粒子コード GYRO3C の特徴 …… 8
  4.1 物理的背景 …… 8
  4.2 プログラムの概要 …… 8
5. 粒子コード GYRO3C の並列化 …… 10
  5.1 プログラム分析 …… 10
  5.2 コンパイラによる自動並列化 …… 11
  5.3 ディレクティブ挿入による並列化 …… 13
  5.4 阿部の方法の導入 …… 14
6. まとめと課題 …… 18
謝　辞 …… 18
参考文献 …… 19
付　録 …… 20

Contents

1. Introduction 1
2. Shared Memory Machine and Distributed Memory Machine 2
3. Parallel Computer T90 4
  3.1 Hardware 4
  3.2 Software 4
4. Characteristic of Particle Code GYRO3C 8
  4.1 Physical Status 8
  4.2 Brief Review of Program 8
5. Parallelization of Particle Code GYRO3C 10
  5.1 Analysis of Program 10
  5.2 Auto Parallelization by Compiler 11
  5.3 Parallelization by Directive Insertion 13
  5.4 Implementation of Abe's Method 14
6. Conclusions and Problems 18
Acknowledgement 18
References 19
Appendix 20

1.　はじめに

近年、単一プロセッサの処理速度向上が限界に近づいたことから、複数のCPU（中央演算処理装置）を並列に使用して処理速度の向上を図る並列計算機の普及が広がりつつある。

並列計算機におけるひとつの処理単位をノードとすると、ノード間の接続形態で大きく分けて２つに分類することができる。ひとつはノード間で主記憶を共有する共有メモリ型計算機、もうひとつはノードごとに独立した主記憶を持つ分散メモリ型計算機である。

本研究では、共有メモリ型並列計算機での既存プログラムの並列化の一例として、CRAY社のT90シリーズ上でプラズマ物理の粒子コードGYRO3Cを並列化し、自動並列化の現状を調査し、検討を行った。

まず、第2章では、並列計算機のタイプの違いによる特徴について述べた。次に第3章では、今回使用した並列計算機T90シリーズについて述べた。第4章では、粒子コードGYRO3Cの特徴について述べた。第5章では、GYRO3Cに対し行った並列化作業について述べた。最後に、第6章では、議論とまとめを行った。

2. 共有メモリ機と分散メモリ機

並列計算機をCPUとメモリの接続形態を考えて分類すると、共有メモリ型並列機と、分散メモリ型並列機に分けることができる。

共有メモリ型並列計算機は、複数のCPUで単一の主記憶装置を共有する並列計算機で各CPUはメモリバスで接続される。そのため、物理的な配線の限界のため、現状では最大でも32台程度の規模のCPU数までしか実現されていない。

分散メモリ型並列計算機は、各CPUに独立した主記憶装置を備え、各CPUはネットワークで接続される。ネットワークによるCPU間の接続はメモリバスの接続に比較して物理的制限が緩いため、数100から数1000台までのCPUまで実現されている。

共有メモリ機としては、T90、PowerChallengeなどが、分散メモリ機としてはT3E,VPP300,SP2などがあげられる。

自動並列化コンパイラの開発の容易さという観点で考えると、共有メモリ型並列機と分散メモリ型並列機の両者を比較すると前者の方に利点が多く見られる。

最大の相違点は共有メモリ型ではデータを主記憶空間で共有すると基本的にCPU間でデータ通信を行う必要が無くなるため、処理手続きの分割とメモリアクセスタイミングのみに留意すれば良い点が上げられる。

分散メモリ型に対する共有メモリ型の利点としては他にも、メモリ利用効率の良さが上げられる。分散メモリの場合は配列データをノード間で分割するには配列データの分割が必要となり、プログラマの負担が大幅に増加する。そのためにデータ分割を行わないとかなりのデータを全ノードで重複して持つことになり、資源の有効利用という観点からは好ましくない。共有メモリ型の場合は基本的にデータは共有メモリ上にCPU間で共有して持たれるため、メモリ資源を有効に活用することができる。

分散メモリ型では一部の処理系を除いてメッセージパッシングライブラリを用いてノード間で通信を行い、並列処理を行うことが多い。メッセージパッシングライブラリにはpvm,MPIなどが普及しており、それらを用いた並列化プログラムは並列性能を維持したままポータビリティが高いという特徴を持つ。

しかしながら既存コードの並列化を行う際にはコードに大幅な変更を加えなければならず、新規に並列コードを開発する際にもコード中に多くのメッセージパッシングライブラリの呼び出しを多数記述する必要があるため、コードの保守性、見易さは低下し、プログラマの負担が大きくなるという傾向がある。

一方、共有メモリ機で主流であるコンパイラディレクティブによる並列指示を介する並列プログラミング環境ではプログラマの負担は少なく、プログラムの変更点も少ない。

しかし、共有メモリ機の並列プログラミング環境には標準というものが存在せず、また標準化に向けた動きも見られない。C言語を用いた場合にはマルチスレッドを使って並列性能を出すこ

とができるが、科学技術計算で依然として主流を占めるFORTRANではマルチスレッド環境はサポートされていないので利用することができない。

以上が共有メモリ機と分散メモリ機のプログラミング環境の概況である。

3. 並列計算機 CRAY T90

本論文で自動並列化コンパイラの評価に用いたプラットフォームは、日本原子力研究所計算科学技術推進センターが現有する CRAY 社の T90 シリーズの 4CPU モデルである T94 である。以下に T94 の特徴を述べる。

3.1 ハードウェア

CCSE が所有する T94 の特徴を以下に述べる。T94 は 4 個の CPU を持つ共有メモリ型のベクトルコンピュータであり、1CPU 単体の理論ピーク性能は 1.8GFlops であり、システム全体では 7.2GFlops の理論ピーク性能を有する。

各プロセッサは、加算、乗算、逆数近似の 3 種類の 2 重化された浮動小数点演算機、8 種類の 2 重化されたベクトル演算機、4 種類のスカラ演算機、2 種類のアドレス演算機で構成される。

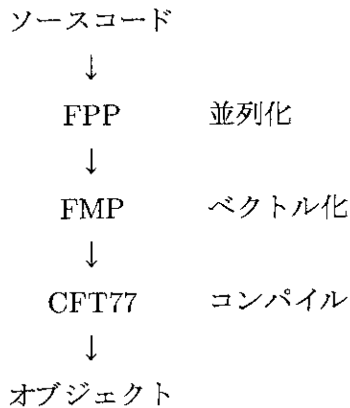
また、インタ・コネクタ機構を eZIF(electrically activated Zero Insertion Force) コネクタの採用により、ワイヤリングを完全に廃し、実装密度を高めたシステムとなっている。そのため、CPU サイクルは 2ns と非常に高速になっている。4 個の CPU で共有される主記憶容量は 128Mw(1GB) で、システムメモリー 18Mw を除いた 110Mw がユーザー空間として使用可能である。

各プロセッサとメインメモリは並列メモリポートにより結合され、メモリバンド幅は 100GB/sec 以上である。

また、CPU 間のデータの相互参照を、共有メモリを利用した方法に比較してより高速に実行可能な共有レジスタによる高速プロセッサ間通信をサポートしており、メインメモリを介さないオーバーヘッドの少ないプロセッサ間通信を可能としている。

3.2 ソフトウェア

T94 には並列コンパイラが CF77 と CF90 の 2 種類が用意されている。CF77 は FORTRAN77 規格に則ったコンパイラであり、並列化指示行の解釈をプリプロセッサを介して実現している。CF77 で用意されているプリプロセッサは FPP,FMP の 2 つで、FPP はソースを解析して自動的に並列化ディレクティブを挿入し、FMP は同様にベクトル化ディレクティブを挿入するプリプロセッサである。FORTRAN77 コンパイラ本体は CFT77 というコマンドである。FPP,FMP の両者を使用したときの処理の流れは



となる。

CF90 は Fortran90 規格に則ったコンパイラで、並列化部分もコンパイラ自体に内蔵されている。現状では両者は平行して開発されているが、今後は CF77 は CF90 に吸収されるものと思われる。本研究では、並列化過程の解析が容易な CF77 を使用して並列化の評価を行った。

3.2.1 並列化パラダイム

CF77 ではプログラムの並列化はプログラム中に並列化ディレクティブを挿入することによって行う。

並列化対象は主に DO ループ単位となり、各並列実行パートは並列タスクとして分割され、実行時に動的に解釈されながら並列実行される。CRAY ではこのことをオートタスキングと呼んでいる。オートタスキングによる並列化の具体的な記述方法については概要については参考文献 [1], 詳細については [2] に詳しい。

以下に代表的なオートタスキングによる並列化ディレクティブを紹介する。

まず、オートタスキングによって DO ループを並列化指示した例を示す。cmic$は FPP に対する並列化指示行であることを表す。shared,private は、それぞれ変数を共有メモリ空間上に確保するか私有領域に確保するかの指定である。

```
cmic$ do all shared(x,y,z) private(i)
      do i=1,n
          x(i)=y(i)+z(i)
      enddo
```

オートタスキングでのタスク分割の方針は、デフォルトでは 1 ループ単位で全てのループを並列タスクに分割するが、タスクの分割単位はディレクティブにより、反復回数、処理のブロックの単位で分割指定可能である。例えば、

```
cmic$ do all chunksize(10)
      do loop
      enddo
```

この例では対象とするDOループを10回のループに分割してタスク分割している。

なお、ユーザーが記述できる並列化ディレクティブ中の変数の指定方法には autoscope という方法も存在する。autoscope 指定は変数の shared,private 判定を FPP の次段のプリプロセッサである FMP に任せることを意味しており、指定すべき変数の数が多くなると記述をシンプルにすることができる。しかし、必ずしも autoscope 指定で適切な判定がなされるわけではないのでパフォーマンスを重視する場合にはこの指定には注意が必要である。

以下の例では変数 i、配列 xx についてはユーザーが指示し、残りの変数については FMP に任せている。

```
cmic$ do all private(i),shared(xx),autoscope
      do i=1,n
        xx(i) = ....
      enddo
```

DO ALL ディレクティブを用いた並列化の場合、DOループ毎にタスクの生成、解放が繰り返されるため、オーバーヘッドが大きくなる。そこで、parallel ディレクティブを用い、複数のDOループをまとめて並列化対象としてオーバーヘッドの削減を図ることができる。以下にその例を示す。

```
cmic$ parallel                        ; 並列タスク生成
cmic$ do parallel autoscope
      do i=1,n
        xx = ....
      enddo
cmic$ do parallel autoscope
      do j=1,n
        yy = ....
      enddo
cmic$ end parallel                    ; 並列タスク解放
```

do all ディレクティブでは、1つめのDOループが終了した時点で並列タスクが解放され、2つめのDOループの始まりで再び生成されるのに対し、上記の parallel ディレクティブを用いることで、タスク制御のオーバーヘッドを削減することができる。

以上で代表的なオートタスキングのディレクティブ紹介を終わる。

なお、並列化ディレクティブはそれを理解しないコンパイラにとってはコメントとして解釈されるために、並列化されたコードのポータビリティに影響しない。

並列化指示はユーザーがディレクティブを手作業で挿入するほかにも FPP にソースコードを解析させ、並列化可能なDOループを自動的に並列化することも可能となっている。FPP は CF77 からコンパイラオプションによって呼び出すことができる。

FPPによって挿入される並列化ディレクティブには変数のshared,private指定も含まれる。sharedは変数は共有メモリ空間上に静的に確保され、private変数は各CPUの持つヒープ空間上に動的に確保される。並列処理に必要な変数をshared,privateにどのように判断するかというと、sharedは

- 読み出しのみの変数
- ループ・インデックスにより指標される配列
- 読み出された後に書き込まれる変数

であり、privateは

- 書き込まれた後に読み出される変数

である。

このことからわかるように、FPPによって挿入される並列化ディレクティブはプログラムの並列実行に最低限必要な変数のみをスタック領域に確保し実行するため、極力メモリ資源の有効利用を考えた指定方法であると言える。

3.2.2 並列実行環境

T90シリーズでの並列実行環境は、分散メモリ並列機に良く見られるような1CPUが1並列処理単位で独占されるような環境とは違い、常にマルチユーザー環境で実行され、マシン全体を独占使用した場合でも各タスクがディスパッチャによってどのCPUに割り当てられるかは一意には決まっていない。したがって、システムの混雑時には限られたCPUを多くのタスクで取り合う事になり、並列実行時の経過時間が大幅に大きくなる。

この特徴は、システムの効率的な運用という観点からは、常に全てのCPUが有効に利用されるため非常に好ましいが、研究対象として並列性能を評価する際には不具合となって現れる。

本研究では、実行時間を計測する際には一時的にマシンを独占使用し、可能な限り外的要因が入らないように留意して計測を行ったが、システムに常駐するデーモン類は動作しているため、ある程度のばらつきが生じた。

4. 粒子コード GYRO3C の特徴

4.1 物理的背景

GYRO3C [3] は、主にトカマク中のプラズマの挙動の時間変化をシミュレートする計算コードである。

計算手法は基本的には粒子-メッシュタイプの粒子コードであり、PIC(Particle in Cell Method)法 [4] を基にし、基礎方程式としてジャイロ運動論的方程式を用いていることを特徴としている。そのため、時間ステップ幅を細かく取る必要性が緩和され、より現実に近い時間スケールの問題を解くことが可能となっている。

並列プログラミングという観点からこのコードをとらえると、原理的に粒子系は良好な並列性を有しており、良好な並列性能を得ることが期待できる。

4.2 プログラムの概要

Fig. 4.2に、GYRO3C のメインルーチンの時間ループ内の流れを示す。プログラムの基本的な流れはサブルーチン SRC によって各粒子の物理量がメッシュ系に投影し、それを FFT（高速フーリエ変換）によって k 空間に変換し、必要な電磁場計算を行ってから real 空間に逆変換を行い、その電磁場をもとに運動方程式を解いて粒子を移動させて次の時間ループに至るという流れである。

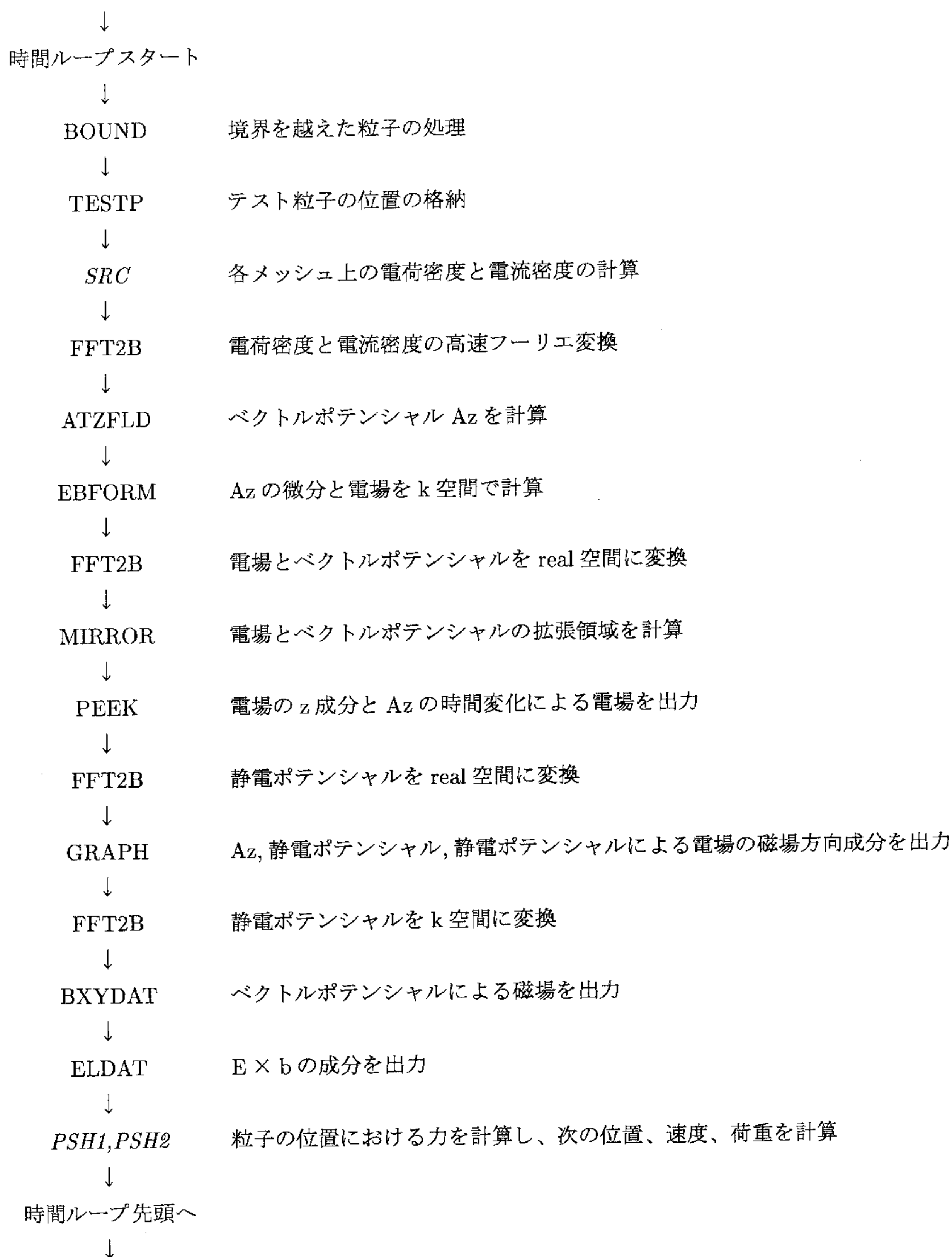


Fig. 4.2 Flow chart of main routine

5. 粒子コード GYRO3C の並列化

この章では実際に行った作業とその効果について段階的に示す。

5.1 プログラム分析

まず、プログラムの性能向上を図る前に、一切手を加えない状態での時間コスト分布を調査し、性能向上の基準とする。Table. 5.1 はプログラム分析ツール kpx [5]（付録 2 に kpx についての説明を記す）に含まれる時間コスト分布アナライザ ktool を用いて調査した GYRO3C コードの主要ルーチンのコスト分布である。なお、問題のサイズはメッシュ系が 32x32x16、粒子系が 64x64x16=65536 である。

Table 5.1 Time cost distribution of original GYRO3C code

RANK	PROGRAM NAME	TIMES	TIME(SEC)	RATE(%)
1	SRC	6000	103.27	31.87
2	PSH1	2997	67.89	20.95
3	PSH2	3000	67.42	20.80
4	FFT2B	40011	23.60	7.28
5	USINM	288088	10.43	3.22
6	CFTN	40011	9.92	3.06
7	ATZFLD	1999	6.16	1.90
8	FSINT	72022	5.51	1.70
9	BOUND	6000	4.41	1.36
10	MIRROR	14000	4.05	1.25
			323.25	

ここでは ktool の、プログラムの構造単位の先頭と最後に実時間を返すルーチンを挿入し、プログラム終了後にコスト分布を集計する機能を用いた。

なお、コンパイルオプションには、-Zv（自動ベクトル化指定）を使用し、サブルーチン CFTN はベクトル化のためのワーク領域が不足していたので-Wf"-o aggress"を使用した。また、サブルーチン FFT2B ではローカル変数をグローバルに確保する必要があったため、-Wf"-o static"を使用した。

粒子系のルーチンである SRC,PSH1,PSH2 が上位を占め、以下、高速フーリエ変換 (FFT) の

ルーチンが続いている。今回はこのうち、粒子系の上位3ルーチンを並列化対象とすることとした。

5.2 コンパイラによる自動並列化

CF77ではコンパイルオプション(-Zp)を指定するとプリプロセッサFPPにより、並列ディレクティブ挿入が行われる。これは、ユーザーが並列ディレクティブの知識を全く持たずに並列性能が得られる特徴を有する。しかしながら、コンパイラはプログラムの物理的、数学的特徴を把握できるわけではないので、完全にコード依存型の並列化となり、必ずしも良好な並列性能が得られるわけではない。

CF77では並列ディレクティブ挿入指示オプションを-ZPと変化させることにより、並列化ディレクティブが挿入された後のソースコードを得ることができる。GYRO3Cの全ルーチンに対して-ZPオプションを適用して中間ソースコードを得たところ、PSH1,PSH2の2つのルーチンに対して並列化ディレクティブが挿入されていることを確認できた。以下にサブルーチンPSH1に挿入された並列化指示行の一部を示す。

```
CMIC@ DO ALL IF (NLENGL .LT. NLENG) SHARED(NDIV, NLENG, NLAST, ZOUT, XXO
CMIC@1    , BPAR, RMAJOR, WREV, FACTOR, CMRT, DT, STZI, AVCON, VKX, VKY,
CMIC@2    VT2I, NO, Z, X, NCXPP, IR, Y, NCYPP, JR, NCZ, KR, ELX, ELY, ELZ
CMIC@3    , AZ, AZX, AZY, AZZ, PPAR, VPAR, AMYU, ACONA, PPOLD, VX, VY, VZ
CMIC@4    , XOLD, YOLD, ZOLD, WGHT, DELW, WOLD) PRIVATE(NLENGL, NCON,
CMIC@5    JFST, JJ, I, LX, LXR, LY, LYR, LZ, LZR, DX, DY, DZ, DX1, DY1,
CMIC@6    DZ1, ALF1, ALF2, ALF3, ALF4, ALF5, ALF6, ALF7, ALF8, EXT, EYT,
CMIC@7    EZT, AZT, AZXT, AZYT, AZZT, XXT, BTOT, BTOTI, BTOTO, BXT, BYT,
CMIC@8    AKAPX, GRDBX, WREV1, VXET, VYET, VYBDT, VPARAT, VPARA2, VYCDT,
CMIC@9    PTEMP, X1, Y1, Z1, SS1, SS2, SS3, WTMP, FCT01)
```

CMIC@は並列化ディレクティブの始まりを示す。ユーザーが自分で並列化ディレクティブを挿入する場合の指示命令はcmic$であるが、CMIC@はFPPによって挿入された並列化ディレクティブであることを示す。

DO ALLは直後のDOループに対する並列処理指示を示す。

IFはある条件を満たす場合にのみ並列実行することを意味し、その決定は実行時になされる。

SHARED,PRIVATEはそれぞれ、変数の私有、共有を意味し、列挙した変数の取り扱いを示す。

DOループ中に存在する全ての変数の共有、私有がここで指示されている。

なお、DOループが具体的にどのように並列処理に適するように分割されるかは、コンパイラ内部で処理されるため、ユーザーには知る手段が無い。

以上のPSH1,PSH2の2ルーチンが自動並列化された場合の時間コスト分布をTable. 5.2に示す。

PSH1,PSH2の2ルーチンはほぼ同じ内容なのでほぼ同じ傾向が見られるが、両者共に1CPU使用時比で4CPU使用時に約1.6倍の加速率を得た。あまり高い加速率ではないが、高い加速率が得られない理由としては

Table 5.2 Time cost of parallelized code by -Zp option

	total	psh1	psh2
original code	323.25	67.89	67.42
parallel code	273.33	42.43	41.78

- 並列実行すべきかどうかの実行時判断
- 並列実行への遷移に伴うオーバーヘッド
- 共有メモリへの複数の CPU からのアクセス競合

などが考えられる。

FPP によって挿入されたディレクティブには `IF (NLENGL .LT. NLENG)` という条件文があるが、この条件判断は実行時になされる。そのため並列対象の DO ループを実行する度に条件判断がなされるためタスク生成時のオーバーヘッドとなる。

オートタスキングによる並列実行は DO ループの先頭でタスクを生成し、並列実行が終了すると並列タスクを開放し、1CPU による逐次処理へと遷移する。このため、タスク生成時に `private` 変数のヒープ領域上への確保が必要となるため、オーバーヘッドとなって性能低下の原因となる。

並列性能が伸びない原因を調べるためには、以下に述べる並列実行時間とタスク生成時間を切り分ける方法である程度推測可能である。

並列化対象となる DO ループが以下のようなとき、

```
cmic$ do all autoscope
      do 1 i=1,n
        処理
    1 continue
```

その DO ループを以下のように展開して二重化する。

```
cmic$ do all autoscope
      do 2 i=1,n1
        do 1 j=1,n2
          処理
    1   continue
    2 continue
```

そして、その並列化した DO ループ全体を ktool で時間測定する。ktool で得られた内側の DO ループにかかった時間は並列実行に費やされた時間であり、外側の DO ループに費やす時間から

内側の DO ループに費やされた時間を引いた時間は並列タスク生成にかかるオーバヘッドであると考えられる。

一般には、時間測定のためのシステムコールをループ 1 の前後に自分で挿入すると、通常はそのシステムコールのオーバーヘッドがループ 2 の実行時間に含まれてしまい、並列処理立ち上げのオーバーヘッドとの切り分けができなくなるが、kpx では計測のためのオーバーヘッドを測定して計算終了後に差し引いて評価するため切り分けが可能となる。

以上に述べた方法で PSH1,PSH2 の性能低下の原因を調査した結果、並列実行部の多くを並列オーバーヘッドが費やしていることが分かった。これは実際には private に確保する必要の無い変数を private 宣言してしまっていることに対することによって生じていると考えられる。

5.3 ディレクティブ挿入による並列化

プリプロッセッサによるディレクティブ挿入は、プログラマへの負担が非常に少ない反面、グローバルな解析を行わないため、得られる並列化効率はそれほど高くならない。並列化可能かどうかの判断をループ内の変数の依存関係のみを材料としているため、潜在的に並列化可能な DO ループは並列化の対象とならないためである。

そこで、手作業により時間コスト分布の上位 3 ルーチンのひとつであるサブルーチン SRC に並列化ディレクティブを挿入し、並列化を実施した。

サブールチン PSH1,PSH2 については、第 5.2 節で調査したように、性能低下が、不必要な private 宣言によって引き起こされている可能性が高いため、使用頻度が少なく、アクセス競合が起きにくい変数を shared 宣言に変更して結果を計測しながら変数の shared,private 宣言を調整し、並列性能の向上を図った。

Table. 5.3に得られた性能向上結果を示す。

Table 5.3 Time cost of parallelized code by directive (4CPU)

	total	src	psh1	psh2
original code	323.25	103.27	67.89	67.42
parallel code(-Zp)	273.33	103.55	42.43	41.78
parallel code(parallel directive)	225.75	92.73	23.22	23.59

サブルーチン SRC については、1CPU による逐次実行に対して 4CPU による並列実行ではコンパイルオプション-Zp による自動並列化では全く並列化が行われなかったのに対し、ディレクティブを挿入した結果約 1.1 倍の加速率を得ることができ、僅かではあるが並列化による高速化の効果があった。

ここで SRC の並列性能が伸びなかった原因を、前節で説明した DO ループを 2 重化して計測する方法で調査した結果、並列実行のためのオーバーヘッドよりも実際に並列処理を行っている時

間の方が遥かに大きいということがわかった。これは shared 指定を行う必要がある加算対象の二次元配列へのアクセス競合が原因である可能性が非常に高い。

アクセス競合が原因となる性能低下は

```
cmic$ parallel private(l1) shared(l)
cmic$ do parallel
      do loop 2
        l = .....
      enddo
cmic$ do parallel
      do loop 1
        l = l1
      enddo
cmic$ end parallel
```

のようにして一時的に private 化した配列へのアクセスにし、後に整合性を取る方法で回避が可能である。

しかし、この場合加算対象の二次元配列が非常に巨大なため、この配列を各 CPU の管理するヒープ領域上に確保して private 化する手法はメモリ資源の観点から現実的ではなく、上記の最適化は不可能であることから今回はこの方法による性能向上は見送った。

PSH1,PSH2 の 2ルーチンにおいては、変数の指定を変数の過剰な private 指定を shared 指定に調整することで FPP による自動並列化と比較して約 1.8 倍程度性能を向上させることができた。

5.4 阿部の方法の導入

サブルーチン SRC は、粒子の物理量をメッシュ上に投影するルーチンであるが、

```
      do 1 i=1,n
        ll(list(i)) = ll(list(i)) + x(i)
    1 continue
```

のような、依存性を持つ可能性がある加算が行われているため、配列 list のうち、一組みでも同じ値を持つものがあると本来はベクトル化は不可能である。

オリジナルの GYRO3C では Heron,Adam の方法 [6] でルーチン SRC のベクトル化を可能にしている。しかしこの方法はベクトル化による高速化のメリットを得られる反面、物理量を保存する配列に比較してはるかに巨大な、本来計算上必要無いワーク配列を使用するために、扱うことのできる問題のサイズが著しく制限されてしまうというデメリットも有する。

そこで、本研究では SRC ルーチンに参考文献 [7] に紹介されているベクトル化手法（以下、これを阿部の方法 [付録 1] と呼ぶ）をインプリメントし、さらなる高速化を図った。阿部の方法もワーク配列を必要とするが、サイズは小さく、メモリ容量的に問題にならない。

阿部の方法を使用するにはいくつかの一般には保証されていないベクトル機の性質を利用する。その条件は

- ベクトルレジスタからデータがストアされるとき、同じアドレスを持ったデータは最後に出現したものが有効となること。
- 1ベクトルレジスタの長さが既知であること。

である。この条件はT90シリーズでは仕様として保証されていないものの、テストプログラムを用いて検証した結果、正常に動作することがわかったので阿部の方法を適用した。

阿部の方法を実現するため、以下に示すようなコーディングを行った。

```
      do 1 i=1,nmax
        ll(i)=0                        ; 阿部の方法のためのリストの初期化
    1 continue
      nt=0                             ; アドレスの重なりのカウンタ
c
cmic$ do all private (l) shared (g1x,gze) autoscope
      do 4 j=1,nmax,nlengo             ; nlengo=128(T90 のベクトル長)
cdir$ ivdep
        do 2 l=j,min(j+nlengo-1,nmax)
          処理
          g1x(...)= g1x(...) + alf
          gze(...)= gze(...) + blf
          lla=lx+(ly-1)*ncx+(lz-1)*ncx*ncy
          k(...)=lla
          ll(...)=l
    2   continue
c
        do 3 l=j,min(j+nlengo-1,nmax)
          if(ll(listp(l)).ne.l) then
            nt=nt+1
            kk(nt)=l                   ; これを k として再計算を繰り返す
          endif
    3   continue
    4 continue
c
   do while(nt.gt.0)                ; ここから反映されなかった部分の再計算
      nn=0                          ;1 回目とほぼ同様だが
      do j=1,nt,nleng0
```

```
cdir$ ivdep
        do jj=j,min(j+nlengo-1,nt)
          l=kk(jj)                    ; ここで1回目に得たリストを利用している
          処理
          g1x(...)= g1x(...) + alf
          gze(...)= gze(...) + blf
          k(l)=lla
          ll(lla)=l
        enddo
cdir$ ivdep
        do jj=j,min(j+nleng0-1,nt)
          l=kk(jj)                    ; ここでも同様
          if(ll(k(l).ne.l) then
            nn=nn+1
            kk(nn)=1
          endif
        enddo
      enddo
      nt=nn
    enddo
```

簡単のために nmax=3 とする。最初のループ1は阿部の方法のためのリストの初期化である。このリストに計算に反映されなかった代入を記録する。

次のループ2では j=1,2,3 について計算がなされるが、ここで粒子位置 k(1),k(2),k(3) がいずれも1の時、ll(1),ll(1),ll(1) と3回 ll の同じアドレスに1が代入されるが、先に述べた仕様により最後の要素である ll(k(3)) のみ計算結果が反映される。

次のループ3では先のループ2で反映されなかった計算結果の粒子番号を kk の中に記録する。そして kk を k として do while ループの処理を同様に処理を nt=0 となるまで繰り返すことにより、計算を終了する。

このアルゴリズムでは nt の値が増加すると無駄な計算を繰り返すことになり、性能が低下するが、粒子の位置が十分にランダムであれば高速にベクトル計算が可能となる。

なお、ここに示したリストで cmic$で指定した DO ループが今回並列化された DO ループである。

Table. 5.4 に、サブルーチン SRC に阿部の方法を適用した場合の単体の性能および並列性能を比較した結果を示す。

阿部の方法は、リストのアドレスの重なりが頻繁に現れる場合には計算量が大幅に増加するために、その効果は小さくなるが、GYRO3C の場合は初期状態で乱数を用いてアドレスが重ならないように工夫して有るので非常に良い速度向上 (約 2.3 倍) が得られていることがわかる。

Table 5.4 Time cost of Abe's Method

	total	src
original SRC(single)	323.25	103.27
original SRC(4CPU)	225.75	92.73
Abe's SRC(1CPU)	254.35	44.33
Abe's SRC(4CPU)	166.83	32.78

並列化による速度向上も約 1.4 倍得られており、オリジナルコードに用いられている方法よりは並列化に向いているベクトル並列化手法と言える。ゆえに SRC としてはオリジナルと比較して最終的に 4CPU 使用時に約 3.15 倍の高速化が達成され、コード全体でオリジナルコード比で約 1.9 倍の速度向上を得た。

6. まとめと課題

今回並列化対象としたGYRO3Cでは、現状ではディレクティブ挿入まで含めた、全自動の並列化ではあまり大きな加速率を得ることができなかった。

しかし、変数のアクセス効率を考慮して、ユーザーが明示的にディレクティブを介して変数の扱いを指定すると並列化効率はかなり改善された。

現状ではT90においては自動並列化コンパイラは必ずしも有効ではない場合があることがわかったが、ユーザーが並列ディレクティブを挿入することにより、かなり並列性能を改善可能である。

プログラマの作業量の面ではコードの大幅な書き換えが前提となるメッセージパッシングによる分散メモリ型の並列計算機上の並列プログラミング環境に対する、共有メモリ型の並列計算機のメリットが示唆された。

自動ベクトル化コンパイラの普及によりベクトルコンピュータが大きく普及したのと同様、並列コンピューティングの普及を考えると自動並列化コンパイラに期待される役目は依然として大きく、今後の研究によるコンパイラの高性能化が待たれる。

また、オートタスキングの並列化は基本的にDOループの分割を代表とする細粒度並列処理をターゲットとしていて、並列実行粒度は小さい。この場合、単体実行部から並列実行部への切り替えの際のオーバーヘッドが多数のCPUを使用したときには無視できない性能低下の要因となるため、サブルーチン間の依存解析を含めたさらに並列処理粒度の大きな並列化プログラミング環境の整備が待たれる。

謝　　　辞

日本原子力研究所 計算科学技術推進センター 並列処理支援技術開発グループの相川 裕史グループリーダーには研究を進める上で数々の多大な助言をいただき、感謝します。また、炉心プラズマ研究部 プラズマ理論研究室の徳田伸二氏にはGYRO3Cのコード提供を頂くとともに助言をいただいたことを感謝します。

6.　まとめと課題

今回並列化対象としたGYRO3Cでは、現状ではディレクティブ挿入まで含めた、全自動の並列化ではあまり大きな加速率を得ることができなかった。

しかし、変数のアクセス効率を考慮して、ユーザーが明示的にディレクティブを介して変数の扱いを指定すると並列化効率はかなり改善された。

現状ではT90においては自動並列化コンパイラは必ずしも有効ではない場合があることがわかったが、ユーザーが並列ディレクティブを挿入することにより、かなり並列性能を改善可能である。

プログラマの作業量の面ではコードの大幅な書き換えが前提となるメッセージパッシングによる分散メモリ型の並列計算機上の並列プログラミング環境に対する、共有メモリ型の並列計算機のメリットが示唆された。

自動ベクトル化コンパイラの普及によりベクトルコンピュータが大きく普及したのと同様、並列コンピューティングの普及を考えると自動並列化コンパイラに期待される役目は依然として大きく、今後の研究によるコンパイラの高性能化が待たれる。

また、オートタスキングの並列化は基本的にDOループの分割を代表とする細粒度並列処理をターゲットとしていて、並列実行粒度は小さい。この場合、単体実行部から並列実行部への切り替えの際のオーバーヘッドが多数のCPUを使用したときには無視できない性能低下の要因となるため、サブルーチン間の依存解析を含めたさらに並列処理粒度の大きな並列化プログラミング環境の整備が待たれる。

謝　　　　辞

日本原子力研究所 計算科学技術推進センター 並列処理支援技術開発グループの相川 裕史グループリーダーには研究を進める上で数々の多大な助言をいただき、感謝します。また、炉心プラズマ研究部 プラズマ理論研究室の徳田伸二氏にはGYRO3Cのコード提供を頂くとともに助言をいただいたことを感謝します。

参 考 文 献

[1] オートタスキング入門 JSJ-0002, 日本クレイ（株）

[2] CF77 コンパイル・システム Volume 4:並列処理 JSG-3074 5.0, 日本クレイ（株）

[3] N.Naitou, K.Tokuda, W.W.Lee and R.D.Sydora, Physics of Plasmas 2, 4257, 1995

[4] C.K.Birdsall and A.B.Langdon, Plasma Physics via Computer Simulation, McGraw Hill Book Company ,1985

[5] 松山雄二・折居茂夫・大田敏郎・久米悦雄・相川裕史, プログラム並列化支援解析ツール kpx, JAERI-Tech 97−017

[6] A.Heron and J.C.Adam, Particle Code Optimization on Vector Computers, Journal of Computational Physics 85, 281-301, 1989

[7] Y.Abe, Present Status of Computer Simulation at IPP, CIENCE AND APPLICATIONS (Supercomputing88), 72-80, 1988

[8] Tuning Guide to Parallel Vector Applications SG-2182 1.3, Cray Research,inc.

付録1 阿部の方法によるベクトル化

阿部の方法によるベクトル化の原理を説明する。

例として以下のようなDOループ1のベクトル化を考える。

```
      paramter(nmax=8)
      real*8 a(8),list(nmax)
      data a /8*0.0/
      data x /1,0,2.0,3.0,4.0,5.0,6.0,7.0,8.0/
      data list /1,2,3,1,5,1,7,8/
      do 1 i=1,nmax
        a(list(i))=a(list(i))+x(i)
    1 continue
```

DOループ1は配列aに関して依存関係を含む可能性があるため、ベクトル化を指示すると正しい結果が得られない。

ここでDOループ1を強制的にベクトル化すると配列aへの加算は以下のようなイメージで行われる。

```
a(1) = a(1)+1.0                     ; (a)
  a(2) = a(2)+2.0
    a(3) = a(3)+3.0
      a(1) = a(1)+4.0               ; (b)
        a(5) = a(5)+5.0
          a(1) = a(1)+6.0           ; (c)
            a(7) = a(7)+7.0
              a(8) = a(8)+1.0
```

ここで重要なのは仕様として保証はされていないが、加算(a),(b),(c)のうち、有効となるのは最後の(c)となることである。したがって、このループが終了した時点では`a(1)=6.0`であり、加算(a),(b)は結果に反映されない。

阿部の方法ではこの(a),(b)に対して加算が行われなかったことをワーク配列に記録しておき、全ての計算結果が正しく反映されるまで反復してベクトル加算を繰り返すことで正しい結果を高速に得る。1回目のループに対しては(a),(b)の加算は無駄な計算であるが、残りの要素に対してはベクトル化の効果を得られるので演算コストを全体で考えると無駄な計算部分のロスを逆転して高速に演算することができる場合があることを利用した方法である。

この例の場合、ループ1をスカラー演算で実行した場合スカラー加算が8回行われるのに対し、阿部の方法で実行するとベクトル加算が3回で済むことになる。現実に阿部の方法を適用する場

合にはこの差がはるかに大きい場合に適用するので高速化の効果が期待できる。

ただし、ベクトルレジスタ中のリストが頻繁に同じアドレスを指して無駄な計算が繰り返し行われる場合にはベクトル演算の有利な点が損なわれるため、計算効率の低下を招き、阿部の方法による効果は期待できない事に注意されたい。

最後に、この方法はベクトル計算機の保証されていない仕様を利用して実現しているため、どのベクトル計算機でも適用可能かは保証されない。使用にあたってはテストプログラムを用いて十分な検証を行うことが必要であることを付け加えておく。

付録2 プログラム並列化支援解析ツール kpx

既存コードの並列化を行う際には、まず計算コストの高い部分を調査し、並列化対象部分を決定することが重要である。

そのためにプログラムの性能分析を行うツールを用いることが一般的だが、プログラムの構造単位のコスト分布を調査するツールは各処理系によって独自に用意されているのが現状で、全てのシステムによって共通で使用できるツールは存在しない。

また、用意されていても時間計測のための動作原理やオーバーヘッドなどの細かい情報が公開されていないため、必ずしも使い勝手の良いものとはいえない。

例えば、T90シリーズが採用しているUNICOSでは、profview,perfview,jumpview,atexpert [8]などの可視化を伴った性能分析ツール群が用意されているが、機能、性能面共に充実しているものの、内部仕様は公開されておらず、計測結果に対する客観的な判断が難しい。

そこで本研究では、現在計算科学技術推進センターで開発中の（1997年1月現在βテスト中）性能評価ツールkpxを使用し、主にサブルーチンやDOループ単位の性能測定を行うktoolを用いて時間コスト分布を調査した。

kpxの特徴は、独自開発であるためソースが付属しており、動作原理も公開されているため、得られたデータに対して客観的な判断がしやすい。そのためどの処理系で使用してもほぼ同様の基準で計測が可能である。

kpxの時間計測の方針は、絶対時間を得るシステムコールをプログラム単位の先頭と最後に挿入し、差を取ることで行っている。kpx自身のオーバーヘッドは記録しているので集計時に差し引かれ、よりプログラム本来の持つコスト分布に近い結果を得ることができる。

また、新たな処理系にkpxを移植する場合にも経過時間を得るシステムコールを処理系によって変更するだけなので作業は用意である。