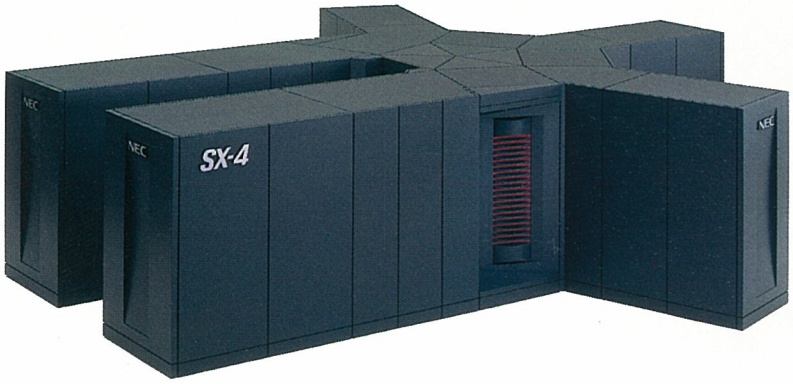
高速化推進研究活動報告



東北大学大型計算機センター

March 2001

目　　次

1　高速化推進研究活動報告の刊行にあたって

センター長　根元　義章

東北大学大型計算機センターは、1969年に、大学の教員、その他の研究者が学術研究等のために利用する全国共同利用施設として設置され、30年が経過した。本センターは、わが国の計算機の揺籃期に、期待をもって誕生し、時代とともに進展をとげ、学術研究の発展に大きく貢献してきた。これも文部科学省をはじめ、関係各位のご支援、ご協力の賜物であり、厚く感謝申し上げる。

本センターの目的は、「最先端かつ世界最大級のコンピュータシステムの導入と利用環境の構築」ということに尽きる。計算機の揺籃期においては、大型計算機センター、計算機メーカ、利用者が一体となって、コンピュータのハードウェア、ソフトウェア、プログラミング言語、アプリケーション等を開発した。また、これら最先端の機器や技術を使いこなすために、上記３者が協力して利用環境を整備し、さらにわかりやすいマニュアルの作成やプログラミング相談、講習会等で利用技術等の普及に努めてきた。その後も現在にいたるまで、種々の情報技術やネットワーク技術の進展にあわせて、上記３者の協力による情報技術の開発が進められてきている。

さて、スーパーコンピュータは、コンピュータ上で模擬実験等を行うシミュレーションに用いられている。スーパーコンピュータの演算処理能力は、普通のコンピュータにくらべ3桁　以上の能力を有している。本センターが1998年1月に導入したスーパーコンピュータSX-4/128H4はベクトル型スーパーコンピュータとしては、当時世界最大の演算処理能力を有しており、メモリー共有機構やベクトル処理機構、コンパイラの自動並列化や自動ベクトル化等で世界的にも優れた特長を有していた。しかし、大学の利用者は、より規模の大きい３次元シミュレーションを行うことを切望しており、計算方法やプログラミングを工夫することが必要であった。そこで導入決定直後の1997年9月より、利用者、本センター、日本電気(株)の３者により高速化推進のための研究会を立ち上げ、約３年間にわたって共同研究を進めてきた。本報告はその成果をまとめたものである。本共同研究で、本センターの技術系職員を中心とするスタッフが中心的役割を果たしたことを特に書き記しておきたい。

最後に、本研究会に積極的に参加し、目覚しい成果を挙げていただいた、本センター利用者、日本電気(株)側スタッフならびに日本電気株式会社、本センター側スタッフに厚くお礼を申し上げる。

2　高速化推進研究活動報告

研究開発部長　牧野　正三

2．1　はじめに

現在、科学は微小化、高精度化、大規模化へと発展しており、従来のように実際に実験装置を作成し、実験条件を種々変えながら結果を見ていくという手法は、費用と時間の面で極めて困難になってきている。スーパーコンピュータ上で模擬実験を行うシミュレーションでは、実際に装置を作成して実験を行わなくても、種々の条件を変えた場合の実験結果を理論に基づいて計算でき、装置作成や実験にかかる時間や費用を大幅に節約することができる。また、電子レベルや分子レベルでの現象の解明、超高温での現象の解明、稼動中のシステムにおける現象の解明等の研究では、いずれも実際に現象を観測することは大変な困難を伴う。このような場合には、スーパーコンピュータによるシミュレーションが、現象の出現機構を解明することができる唯一の方法である。このようにシミュレーションは、理論、実験とともに、先端科学を支える３本柱の一角を成している。

先端科学の研究を使命とする大学にとって、スーパーコンピュータによるシミュレーションはいまやなくてはならないものであり、シミュレーションの高速化や高精度化が今後の研究にとって必要不可欠なものになっている。

2．2　シミュレーションとコンピュータの処理能力

我々の世界が縦ー横ー高さをもつ立体的な３次元の世界であるのと同様、シミュレーションにおいても図２．１に示すように模擬したい物体(ここでは立体)を縦ー横ー高さの３次元空間で扱う(３次元モデルという)。しかし、３次元モデルでは計算量が膨大になるため、コンピュータの処理能力が低い場合は、次元を落としたモデルで近似し、計算量の減少が図られる。たとえば、２次元モデルでは、縦ー横の２つの次元のみを扱い高さは影響しないと仮定する。１次元モデルでは、横のみを扱い縦と高さは影響しないと仮定して計算量を少なくする。さらに１次元モデル、２次元モデル、３次元モデルそれぞれにおいても、コンピュータの処理能力が低い場合は、精度を落としたモデルで近似し、計算量の減少が図られる。

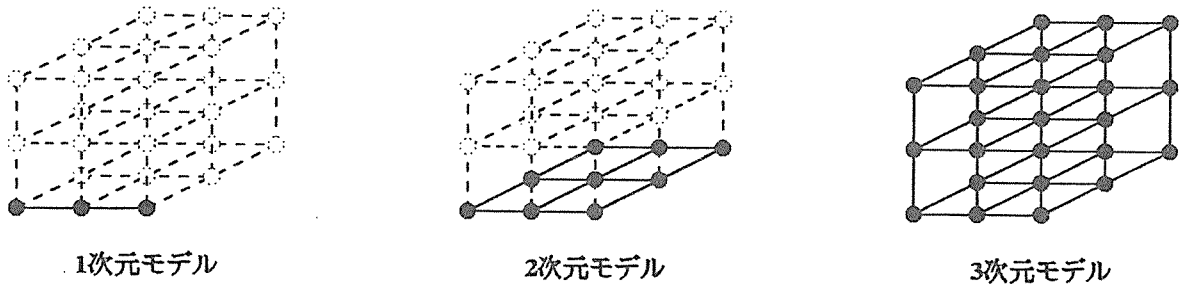


図２．１　シミュレーションモデル

例えば一番精度の良い大規模モデルの計算量が巨大な場合には、精度を落とした中規模モデル、小規模モデルにして計算量を少なくする。こうして、シミュレーションはその時々のコンピュータの処理能力に合わせて、１次元から２次元、２次元から３次元へと、また小規模

モデルから中規模モデルを経て大規模モデルへと進んできている。

シミュレーションを行う際の計算方法には種々あるが、代表的なものは図２．１に示すように対象を格子状に区切り、格子点ごとに計算を行っていく方法である。図２．１の例では格子点の数は、３次元モデルでは 3×3×3=27、２次元モデルでは 3×3=9、１次元モデルでは３となる。計算量はこの格子点の数の２乗から５乗に比例する。仮に２乗としても１次元モデルから２次元モデル、あるいは２次元モデルから３次元モデルに拡張する際の計算量は９倍増加する。また３次元モデルにおいて精度を２倍(5×5×5=125)にすると計算量は21倍増加し、３次元モデルにおいて精度を上げるにはこれまで以上に演算処理能力が必要である。現在、新しい計算方法の利用やプログラムの高速化のための種々の工夫によって、図２．２に示すように、分野によってようやく小規模３次元モデルによるシミュレーションが可能な状態になっている。しかし、今後の中規模や大規模の３次元モデルによるシミュレーションではこれまで以上に計算量の増加は大きく、高速化のためのアルゴリズムや工夫が一層必要である。

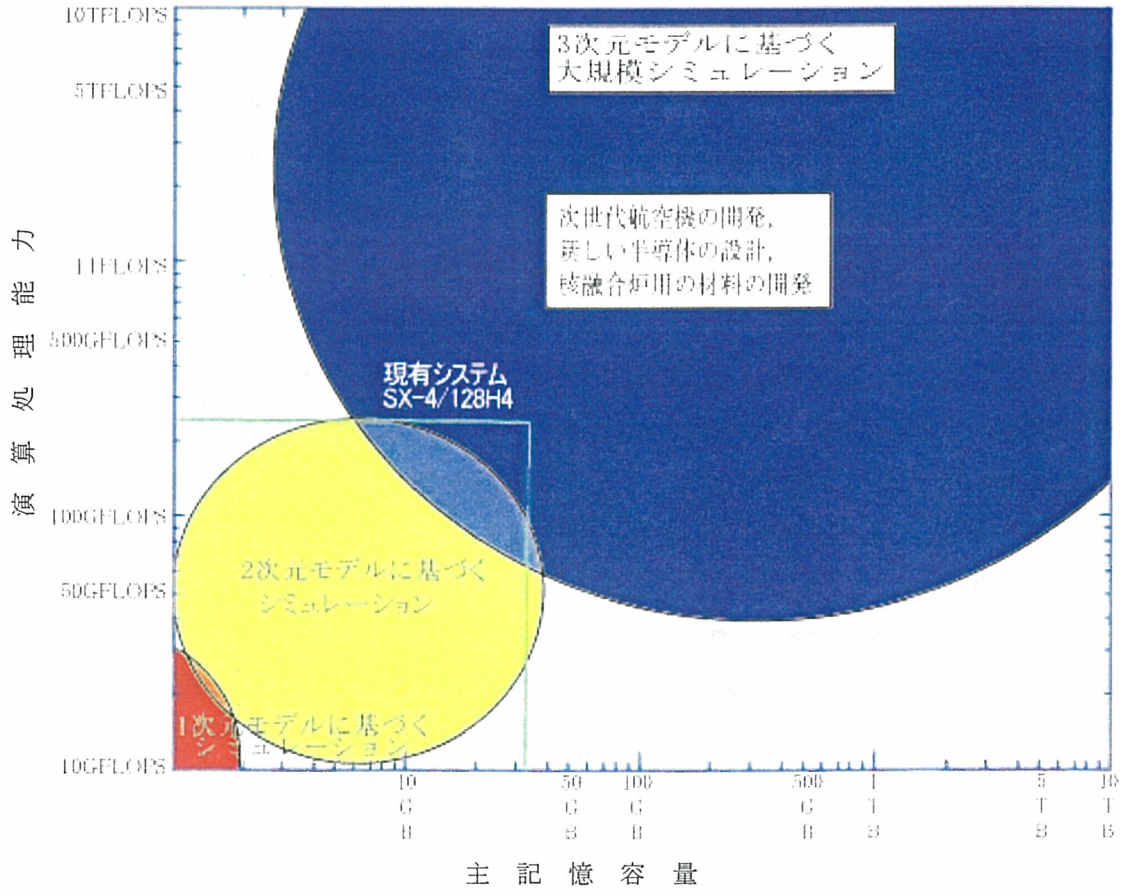


図２．２　シミュレーションモデルと演算処理能力、主記憶容量の関係

２．３　スーパーコンピュータの計算方式

高速演算するためにスーパーコンピュータで採用されている計算方式を通常のコンピュータとの比較で示す。

(1) **通常のコンピュータ** 通常のコンピュータのCPUは、荷物運び作業に例えると、図２．３に示すように一人の作業員が１回に１個の荷物を運んでいることに相当する。

(2) **ベクトル型スーパーコンピュータ** ベクトル型スーパーコンピュータのCPUは、図２．４に示すように流れ作業的な処理をすることによって高速化することが特長である。この図では、ベクトル型スーパーコンピュータのCPU内の処理の流れを、荷物運び作業に例えている。荷物が最初の作業員に手渡され、さらに次の作業員に手渡されると同時に、新しい荷物が最初の作業員に手渡される。このようにすると、本来は運ぶのに長時間かかる作業であっても、多くの作業員で分担すると非常に高速に荷物運びが可能になる。特に各作業員の作業時間が等しくかつ大量の荷物を運ぶ場合には後述する並列スーパーコンピュータに比べて高速化の効果が顕著に現れる。しかし、荷物の大きさや形、重さが変って各作業員の作業時間の均等性や連続性が失われると作業効率が落ちる。

図２．３　通常のコンピュータの処理の流れ

図２．４　ベクトル型スーパーコンピュータの処理の流れ

(3) **並列型スーパーコンピュータ** 並列型スーパーコンピュータのCPUは、基本的には通常のコンピュータと同じCPUである。図２.５に示すように作業員一人が一つのCPUに対応し、作業員それぞれは１回に１個の荷物しか運べない。そこで多数の作業員が同時に荷物を運ぶことで高速化を図る。それぞれの作業員が独立に荷物を運ぶことができる場合は、荷物の形や大きさ、重さが違うことによって個々の作業員の作業時間が変っても周りには影響を与えないが、他の作業員の仕事が終了するのを待たなければいけない場合には効率が落ちる。また大規模行列の演算のように作業の均等性や連続性がある場合は、ベクトル型スーパーコンピュータに比べ性能がかなり落ちる。

(4) **ベクトル並列型スーパーコンピュータ** ベクトル並列型スーパーコンピュータは図２．６に示すように、ベクトル型と並列型のスーパーコンピュータを融合したものである。横１列の作業員がベクトル型スーパーコンピュータのCPUに相当し、縦３列が３個のベクトル型CPUに対応する。前述したように３次元シミュレーションには、膨大な演算処理能力が必要である。そこでベクトル型スーパーコンピュータのCPUを並列に多数個並べるという方式が開発された。プログラミングに留意すると、ベクトル型と並列型の長所を併せ持つことができる。現有のSX-4/128H4は、ベクトル並列型スーパーコンピュータである。

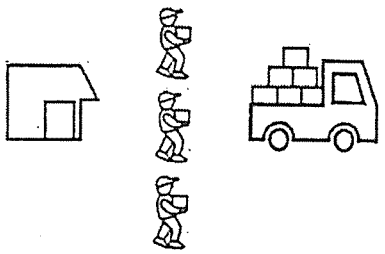


図２．５　並列型スーパーコンピュータの処理の流れ

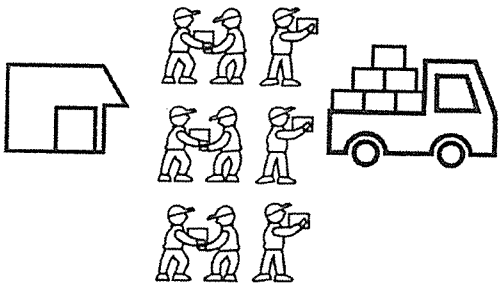


図２．６　ベクトル並列型スーパーコンピュータの処理の流れ

２．４　東北大学大型計算機センターにおけるスーパーコンピュータの変遷

東北大学大型計算機センターは 1986 年の SX-1(日本電気製、0.57GFLOPS)から現在の SX-4/128H4(日本電気製、128CPU、32GB、256GFLOPS、1998 年１月更新)に至るまで、常に世界最高クラスのスーパーコンピュータを導入し、これら最先端の学術研究を強力に支援・推進してきた。その結果、表２.１に示すように、シミュレーションも SX-1 の小規模１次元モデルから SX-4 の小規模の３次元モデルへと発展してきている。

表２．１　本センターのベクトル型スーパーコンピュータの変遷

導入年	型式	性能(GFLOPS)	記憶容量	シミュレーションモデル
1986 年	SX-1	0.57	256MB	小規模１次元モデル
1990 年	SX-2N	1.14	256MB	中規模１次元モデル
1994 年	SX-3/44R	25.6	4GB(4000MB)	中・大規模２次元モデル
1998 年	SX-4/128H4	256	32GB	小規模３次元モデル

現有のSX-4/128H4では、２GFLOPSのベクトル型CPUを32個集めて１ノードとし、４ノードで１システムとしている。利用者は、自動並列化と自動ベクトル化を利用することによって、MPIなどの知識を有さなくても、１ノードの演算処理能力(64GFLOPS：１秒間に640億回の浮動小数点演算が可能)をフルに利用することができる。

２．５　シミュレーションの高速化推進研究

1998 年１月に導入した現有の SX-4/128H4 は本センターとして初のベクトル並列型スーパーコンピュータである。本システムは、利用者の有する大部分のプログラムに対して、FORTRAN の自動並列化・自動ベクトル化により、その特長である共有メモリーとベクトル機構をフルに活用できる。しかし、より規模の大きい３次元シミュレーションを実現する

ためには、自動並列化・自動ベクトル化に加えて、プログラムをより高速化する必要があった。このため、SX-4/128H4 の導入決定直後の 1997 年 9 月より、利用者、本センター、日本電気(株)の３者によりシミュレーションの高速化推進のための研究会を立ち上げ、高速化のための作業を開始した。表２．２に本プロジェクトに参加した本センター側スタッフと日本電気側スタッフを示す。シミュレーションプログラムの高速化を推進するには、利用者のプログラムで使われている計算アルゴリズム、データ構造、さらには研究目的や研究内容まで熟知する必要がある。そのためセンター側スタッフ、日本電気側スタッフが長時間をかけて利用者の研究目的や研究内容を理解し、シミュレーションに適したアルゴリズムやプログラミング方法、SX-4/128H4 のアーキテクチャを考慮したデータ構造等について改善策を提案した。

表２．２　東北大学と日本電気(株)参加者

東北大学	日本電気（株）
研究開発部 牧野　正三、岡部　公起 システム管理掛 伊藤　英一、大泉　健治 システム運用掛 小野　敏、佐藤　多佳子	第一コンピュータソフトウエア事業部 片山　博、橋本　ユキ子 スーパーコンピュータ販売推進本部 津和　義昭、小久保　達信 第二公共システム開発事業部 渡辺　好幸、神山　典、熊沢　浩市

２．６　高速化推進研究会の成果

研究会の成果の詳細は次章以降で述べる。この研究会で得られた成果の概要を以下に挙げる。

(1) ３年間で 27 本の大規模プログラムの高速化を行った。その中には、オリジナルのプログラムに比べ最大 347 倍の高速化が達成されたものもある。その結果、利用者の研究活動のリードタイムを大幅に短縮できた。

(2) 本センターにとって、利用者プログラムの高速化は混雑の著しいスーパーコンピュータの有効活用につながった。

(3) 日本電気株式会社にとって、本研究会活動で得られた成果を、FORTRAN コンパイラの製品へ反映させることができ、ベクトル化、並列化の自動化技術を向上することができた。

3　高速化推進の必要性

3.1 SX-4の性能

SX-4/128H4システムは128個のCPUをもつ、「スケーラブル・パラレル・スーパーコンピュータ」であり、その理論最大演算性能はCPU単体で2GFLOPS、システム全体では256GFLOPSに達する。

この超高速演算性能を実現しているハードウェアの特長としては以下のものがある。

・CPUあたり32本のパイプライン演算器が同時動作できる強力なベクトル演算ユニット

・32台のCPUを共有メモリに接続するマルチプロセッサ構成のノード

・4台のノードを超高速インタフェースHIPPIにより接続するクラスタ構成

こうしたSX-4のハードウェア性能を十分に引き出すためには、

・個々のCPUの中で効率的に計算を行う、**ベクトル化**

・複数のCPUを効率的に使用する、**並列化**

が大変重要となる。

SXシステムのFortran90コンパイラであるFORTRAN90/SXは、SX-4のハードウェア性能を十分に引き出すための、高度な自動ベクトル化、自動並列化機能を備えている。

しかし、こうしたコンパイラによるベクトル化、並列化の能力には限度があるため、常に最適な実行プログラムが生成できるわけではない。SX-4のハードウェアの特性に合わせたプログラミングにおける工夫を行うことによって、性能を大きく改善できることがある。

3.2　ベクトル化による高速化

通常の演算命令（スカラ命令）は、一度に一組のデータに対する演算処理を行う。これに対して、ベクトル命令は、複数の組のデータに対する演算処理を一つの命令で一度に行うことができる。

ループ中で計算される行列の要素など、規則的に並んだ配列データ(ベクトルデータ)に対してベクトル命令を適用することを自動ベクトル化と呼び、ベクトル化することによって高速な演算が可能となる。

ベクトル化によるプログラムの実行性能の向上では、ベクトル化率の向上、ベクトル長の向上、メモリアクセスの改善の3点が重要になる。

1）ベクトル化率

プログラムをスカラ命令だけで実行させた場合の実行時間に占める、ベクトル命令で実行可能な部分の時間の割合をベクトル化率と呼ぶ。

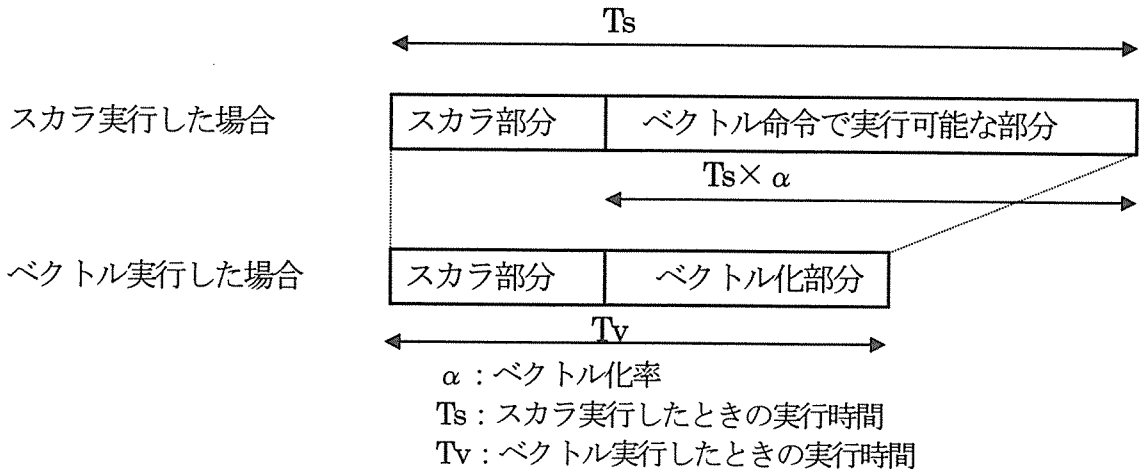


図３．１　ベクトル化率

プログラムをベクトル化することにより実行性能が向上するが、プログラムの一部だけをベクトル化しても、ベクトル化の効果はあまり期待できない。たとえば、ベクトル化部分の実行時間が非常に小さくても、ベクトル化率が 50%程度では、高々2 倍の性能にしかならない。

ベクトル化率と、プログラム全体の実行時間の理想的な性能向上率の関係を表すのがアムダールの法則である。

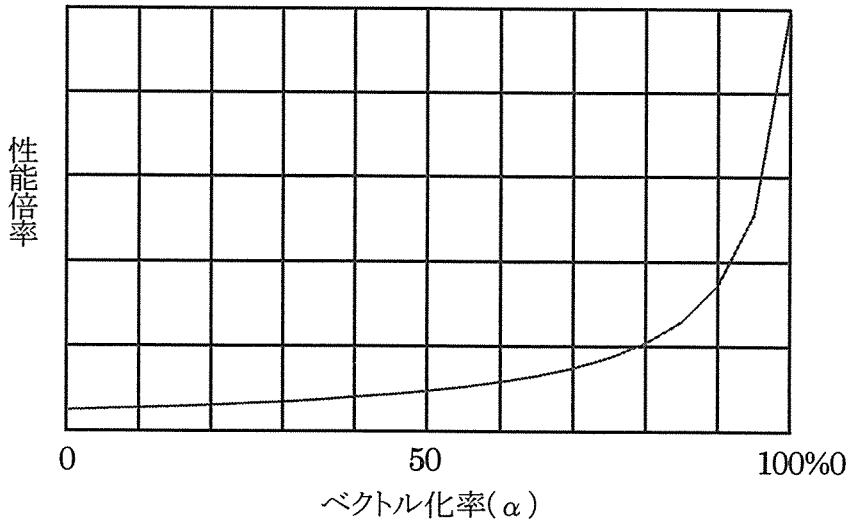


図３．２　アムダールの法則

このグラフを見るとわかるとおり、一般にはベクトル化率が 95%以上ないと、ベクトル化による大きな性能向上は期待できない。

FORTRAN90/SX では、プログラム中の **DO** ループ、**IF GOTO** ループ、配列式をベクトル化の対象としている。

それらがベクトル化できるためは、以下の条件を満たしている必要がある。

(a) ループまたは配列式内にベクトル化可能な文、型、演算がある。

(b) ループ中の変数や配列が定義・引用される順序がベクトル化によって変化しない。

CALL 文、入出力文などの文、文字型や4倍精度実数型、構造型などの型、文字比較、文字代入、ポインタ代入などは (a)の条件を満たさないのでベクトル化されない。

また、図3.3のように、前の繰り返し回で定義された値を後の繰り返し回で参照するようなループは、(b)の条件を満たさないため、ベクトル化されない。

```
DO I=3,N
   X(I)=X(I-2)*2.0+Y(I)
END DO
```

図3．3　ベクトル化できないループ

プログラムを高速化するためには、指示行の挿入や、プログラムの書き換えにより、ベクトル化できない部分を減らし、ベクトル化率をできるかぎり 100%に近づけていくことが重要となる。

2）ベクトル長

個々のベクトル命令は、演算処理に入る前にある程度の準備処理が必要となる。この準備処理に要する時間を立ち上がり時間と呼ぶ。即ち、実際のベクトル演算に要する時間が余りにも小さいと、立ち上がり時間の影響が大きくなり、ベクトル化による高速化が行えない。

ベクトル化した場合とベクトル化しない場合とで実行時間が等しくなるループの繰り返し数(ループ長)を交差ループ長と呼び、立ち上がり時間と交差ループ長には、図3.4に示す関係がある。

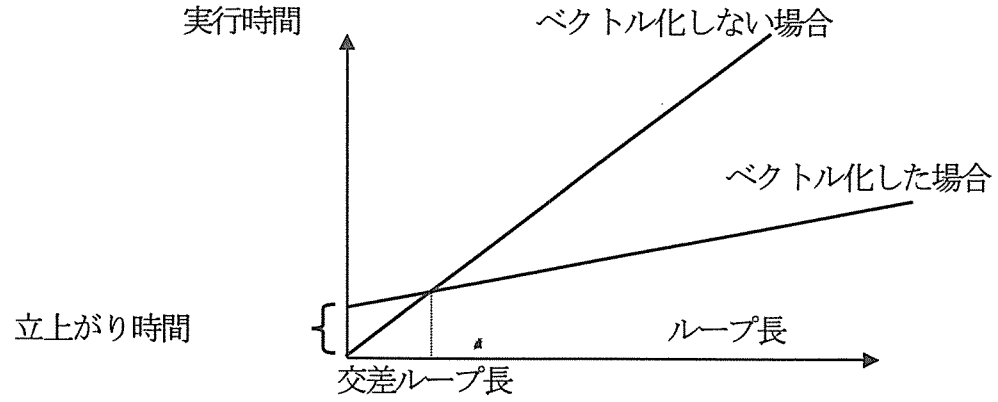


図3．4　立上がり時間と交差ループ長

この図からわかるように、ループ長をできるだけ長くした方が、ベクトル化による高速化の効果が大きくなる。したがって、できるかぎり大きなループ長をもつループでベクトル化を行うことが重要である。

3）メモリーアクセスの改善

SX-4 では、主記憶とベクトルレジスタとの間のデータの転送を高速に行うために、主記憶を 32 個の独立したモジュール（これをメモリバンク・グループと呼ぶ）に分割して、別々のグループに対して、並列に読み出しまたは書き込みができるようにしている。しかし、同一のメモリバンク・グループに対しては、同時には１つの読み出しまたは書き込みしかできないため、連続して同じグループのデータをアクセスすると性能が低下する。この状態をバンクコンフリクト呼ぶ。

たとえば、一つの配列を 32 要素飛び(又は 32 の倍数要素飛び)にアクセスすると、バンクコンフリクトが発生する。これを防ぐためには、配列宣言を奇数になるように変更するか、ループを入れ換えて、配列のアクセスが連続するように変更するなどの工夫が必要となる。

3.3　並列処理における高速化

並列処理では、1 つの仕事をいくつかの小さな仕事に分割し、それを複数のタスク(CPU)で並列に実行することにより、仕事の終了までの経過時間を短縮する。

ここで注意しなければならないのは、並列処理では、CPU 時間が削減されるわけではなく、逆に、仕事を各タスクで並列実行させるための処理（オーバーヘッド）が必要となり、CPU 時間は増加することである。

CPU 時間と経過時間の関係は、以下のようになる。

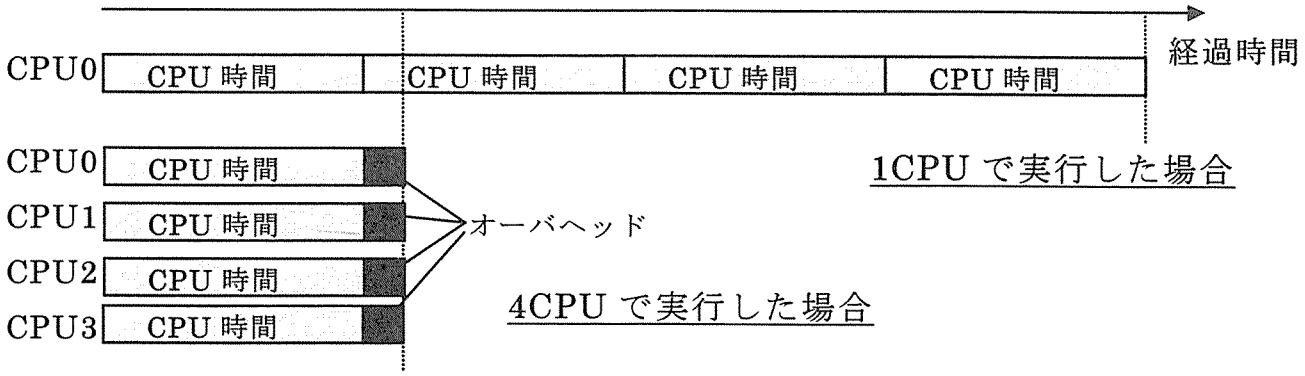


図３．５　並列化におけるＣＰＵ 時間と経過時間

この並列処理のオーバヘッド時間のため、並列に実行される仕事量（粒度と呼ぶ）が十分に大きくなければ、並列化の効果は期待できない。当然であるが、並列処理のオーバヘッド時間よりも並列実行される部分の実行時間の方が小さければ、並列化したことにより、実行時間（経過時間）がかえって多くなってしまうことになる。これは、高速なベクトル演算を使用することにより CPU 時間が短縮され、同時に経過時間も短縮されるため、ほとんど全ての場合に性能向上が図れるベクトル化と大きく異なるところである。

並列化により性能を向上させるためには、並列化率の向上と並列化効率の向上が重要となる。

1）並列化率

並列化率とは、あるプログラムを 1CPU で実行した場合の実行時間に対する、並列実行可能な部分の実行時間の割合を示したものである。

Ts：シングル実行したときの実行時間
Tp：並列実行したときの実行時間
α：並列化率
n：並列実行した CPU 台数

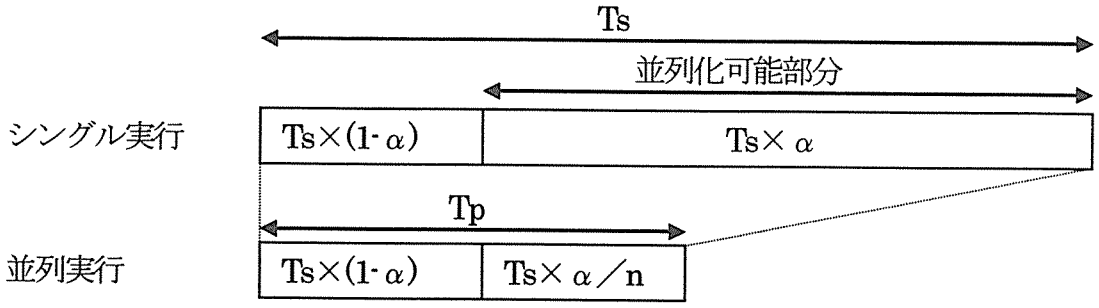


図３．６　並列化率（α）

並列化でも、ベクトル化と同様にアムダールの法則が成り立つ。

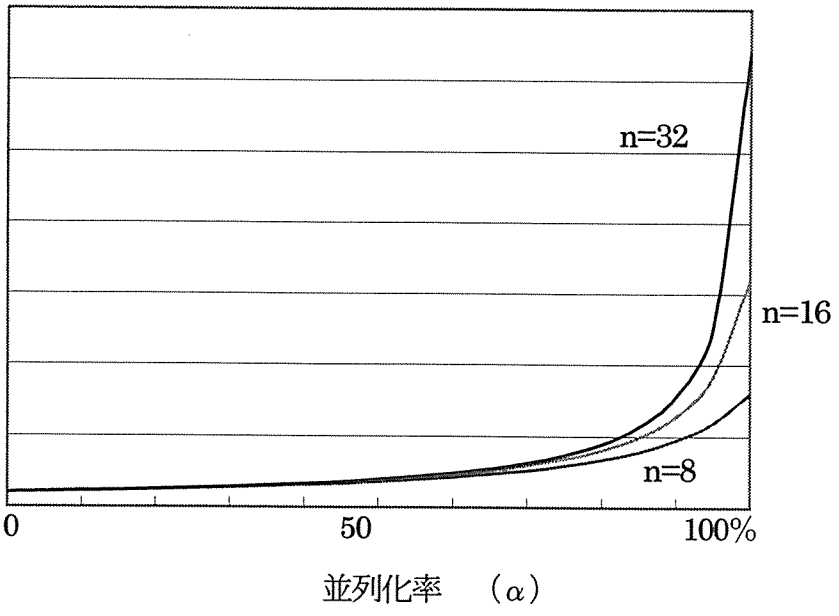


図３．７　並列化におけるアムダールの法則

したがって、並列化によって性能を向上するためには、並列化可能部分をいかに増やし並列化率を上げるかが大きな鍵となる。

FORTRAN90/SX の自動並列化機能は、プログラムを解析し、十分性能が向上するだけの粒度をもっているか、DO ループの繰り返しを並列実行しても結果不正になるような文を含んではいないかなどを調査し、可能な場合は、並列化できるようにループやデータの定義を書き直して１ノード内の 32CPU を用いた並列実行プログラムを生成する。

ループが並列化できるためは、以下の条件を満たしている必要がある。

(a) ループ内にCALL文や入出力文などの並列化できない文がない。

(b) ループ中の変数や配列がループの繰り返しにまたがって定義・引用されていない。

並列化率を上げるためには、コンパイラが自動並列化できないループに対して、プログラムを書き変えたり、並列化指示行を挿入したりして、並列化を促進していくことが必要である。

2）並列化効率

並列化効率を左右する大きな要因としては、次のようなものが考えられる。

・並列化されているループの（ベクトル化後の）実行時間（粒度）は十分に大きいか

・並列実行されるループの仕事量（負荷バランス）は各タスクに対して均一か

前に説明したとおり、並列処理されるループが十分な仕事量をもっていなければ、並列処理のオーバーヘッドの割合が大きくなり、並列化効率は低くなってしまう。したがって、同じ多重ループを並列化する場合でも、より演算量が多くなる繰り返しに着目して並列化ができるように注意を払うことが必要である。

SX-4では、内側ループはベクトル化により高速化を行い、なるべく外側のループを並列化により高速化するようにループの入れ換えを行うなどの工夫が必要となる。

一方、いくら並列化率が高く、並列の粒度が大きく、並列オーバーヘッドの割合が低くても、個々のタスクの負荷バランスが悪いと、経過時間は最も負荷の高いタスクによって決まってしまう。したがって、各タスクの仕事量をできるだけ均等にする必要がある。

FORTRAN90/SXでは、ループの並列実行方法は以下の2つの方法がある。

for： ループを指定した数に分割する。

by： ループを指定した数の繰り返しをもつループに分割する。

並列化されるループの特性に合わせ、これらを使い分けることにより、ループの処理ができるだけ均等に各タスク上で処理されるように分割方法を工夫することが必要である。

3.4 高速化に対する取り組み

ＮＥＣでは高速化推進として、システムサポート部門内にベクトル化・並列化の専門グループを準備し推進してきている。

本グループはユーザーからの高速化要求に対応して、コンパイラの開発部門及びハードウエア開発部門との連携をとり、高速化の最善施策を検討し、具体的解決を実施してきている。

また、同時に科学技術計算ライブラリの開発部門との連携により、ライブラリ適用での支援、さらには必要に応じてベクトル化・並列化での個別支援を受ける体制で推進しており、高速化の最適解を目指している。

4　高速化技法

他では実行できないような大規模・長時間ジョブの実行を可能とするため、本センターではスーパーコンピュータ SX-4 と並列サーバ Exemplar/X を設置している。この２つのコンピュータは共に高性能であり、また、コンパイラも高度の最適化機能を有しているので、通常はコンパイラのもつ自動ベクトル化および自動並列化の機能で十分な高速化が期待できる。しかし、数千時間にも及ぶプログラムの実行となると、SX-4 の p32 クラス（32CPU を用いた並列処理）を用いても１週間以上を要することになり、さらなる高速化の必要性は極めて高い。

一口に高速化といっても、SX-4 はベクトル並列型のスーパーコンピュータのためベクトル化を中心とした高速化、並列サーバはスカラ並列型コンピュータのためキャッシュを意識した高速化、というように、それぞれのコンピュータの特徴を活かした高速化が必要になってくる。ここでは、特に事例の多かった SX-4 について、代表的な高速化の技法（チューニング）を述べる。

4.1　プログラムの動作の分析

SX-4 が高性能のスーパーコンピュータとはいっても、ベクトル化率が低い場合は並列サーバと同程度の性能であるため、そのまま並列化しても大きな性能向上比は得られない。前述の図３.２からも明らかなように、ベクトル化による性能向上の効果が現れるのはベクトル化率が 90%を超えるあたりからで、95%以上では一層顕著となるので、ベクトル化率を少しでも高くすることが SX-4 での高速化の最大のポイントになる。そのためには、プログラムの現状を分析し、どのようにチューニングを行っていくかを決めなければならない。このとき、大きな手掛かりとなるのが、プログラム動作情報とサブルーチン動作情報である。

（1）プログラム動作情報

プログラム動作情報は、プログラムの実行終了時に図４.１の形式で出力される。これには、プログラムの実行に関わるいくつかの情報が含まれているが、なかでも重要なのが図に示したベクトル演算率、平均ベクトル長、バンクコンフリクトの３つである。

①ベクトル演算率（V.Op.Ratio）

ベクトル化率の正確な測定ができないため、その近似値として用いている。100%に近いのが理想で、98%程度であれば一応満足といえる。図のベクトル演算率は 86%と低く、ベクトル化されていない DO ループのあることが分かる。

②平均ベクトル長（VLEN）

ベクトル化された DO ループで実行されるベクトル命令の平均の長さを表す。SX-4 のベクトル命令は一度に 256 要素を扱うので、この値は最大で 256 となる。たとえば、ループの長さが 257 であれば、平均ベクトル長は 128.5 である。図の平均ベクトル長は 10.96 と非常に小さいので、ベクトル化による高速化はあまり期待できないことになり（図３.４参照）、DO ループの入れ替え等を行って、ループ長を拡大する必要のあることが分かる。

③バンクコンフリクト（Bank）

バンクコンフリクト（3.2参照）発生によるメモリアクセスのロス時間であり、値が小さい程メモリアクセスの効率が良いことになる。図の56秒は全体のCPU時間（User Time）からみれば小さいが、ベクトル演算時間（Vector Time）との比較でみれば約24%を占めていることになり、バンクコンフリクト削減の必要があることが分かる。

```
***** Program Information *****
Real Time (sec)   :     1335.462466
User Time (sec)   :     1285.002113     CPU時間
Sys  Time (sec)   :        3.370989
Vector Time (sec) :      236.964672
Inst. Count       :    34053385834.
V. Inst. Count    :    12614258618.
V. Element Count  :   138255230171.
FLOP Count        :    43120320074.
MOPS              :      156.220226
MFLOPS            :       48.723409
VLEN              :       10.960234     平均ベクトル長
V. Op. Ratio (%)  :       86.574900     ベクトル演算率
Memory size (MB)  :      938.478310
MIPS              :       26.500689
I-Cache (sec)     :       10.053183
O-Cache (sec)     :       23.042072
Bank (sec)        :       56.153429     バンクコンフリクト
```

図4.1　プログラム動作情報

（2）サブルーチン動作情報

プログラム動作情報の分析では、全体的な問題点は分かっても、それがプログラムのどの部分に起因しているかはわからない。また、CPU 時間の少ない部分のチューニングを行っても、性能向上にはほとんど寄与しないので、CPU 時間の大きいサブルーチンに限定したチューニングを行う必要もある。そのためには、図4.2に示すサブルーチン動作情報を採取し、分析する。プログラム動作情報がプログラム全体の情報であるのに対し、文字通り個々のサブルーチンの情報を知ることができる。サブルーチン動作情報はオプション「-ftrace」を指定してコンパイルすることにより、プログラムの実行終了時に出力される。

```
sxf90 -ftrace -R5 source.f
```

図より、サブルーチンsub01は、呼び出し回数（FREQUENCY）が100回、CPU時間（EXCLUSIVE TIME）は743秒で全体の57.8%を占め、呼び出し1回あたりのCPU時間（AVER.TIME ）は7432 ミリ秒である。ベクトル演算率（V.OP.RATIO）の90.61%は十分とはいえず、平均ベクトル長（AVER. V.LEN）の65も小さい。また、バンクコンフリクト（BANK CONF）も53秒あり、バンクコンフリクトはこのsub01に集中していることが分かる。したがって、サブルーチンsub01はベクトル化率の向上、ループ長の拡大、バンク

コンフリクトの削減の必要があることになる。sub03 のベクトル演算率 67%、平均ベクトル長 39 も同様に改善する必要がある。

PROG. UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER. TIME [msec]	V. OP RATIO	AVER. V. LEN	BANK CONF
sub01	100	743.266(57.8)	7432.660	90.61	65.0	53.477
sub03	100	436.418(34.0)	4364.176	67.57	39.4	2.676
sub04	10	90.539(7.0)	9053.895	95.51	239.9	0.000
main_	1	10.028(0.8)	1002.887	96.00	243.0	0.000
.	.		.			

図４．２　サブルーチン動作情報（抜粋）

４.２　ベクトル化のチューニング

プログラム動作情報とサブルーチン動作情報の分析の結果、チューニングの方針が決まったら、問題ありとしたサブルーチンを調べて原因のループの改善を行う。ここでは、代表的な例として、ベクトル化指示行によるベクトル化の促進、ループの入れ替えによるループ長の拡大、バンクコンフリクトの削減について述べる。

(1)ベクトル化指示行によるベクトル化

ベクトル化の条件（３.２参照）を満たさないループの他に、条件に適合しない可能性のあるループもベクトル化されない。その典型的な例が図４.３に示すリストベクトルである。ループがベクトル化されているか否かは、オプション「-R5」をつけてコンパイルしたときに得られる編集リストで確かめることができる（サフィックスが.L のファイルに出力される)。図のように、do 文の左方に付加された記号が＋で始まるループはベクトル化されていないことを示す（ベクトル化されたときはVで始まる)。

図で、配列ｙは別の配列 ix の内容にもとづいて引用・定義が行われているが、ループの指標ｉが１からｎまで動く間に、ix(i)が同じ値をとる場合は、異なるｉで同一の配列要素 y(ix(i))を指すことになり（これを「依存関係がある」という)、ベクトル化すると不正な結果をもたらすからである。したがって、ix(i)がすべて異なる値をとる場合は、異なる繰り返しで同一の要素を使うことはない、すなわち、依存関係はないので、ベクトル化しても正しい結果が得られることになり、ベクトル化指示行「!cdir nodep」を用いてベクトル化することができる。

```
        !cdir nodep                    ベクトル化指示行（カラム１から記述）
+------>    do i=1,n
|             y(ix(i))=y(ix(i))+a(i)*b(i)     配列ｙの指標が間接的
+------      end do                            →リストベクトルという
```

図４．３　ベクトル化されないループ

（2）ループの入れ替えによるループ長の拡大

DOループが多層構造になっている場合、ベクトル化されるのは一番内側のループである。このため、そのループの繰り返し数、すなわち、ループ長が小さいと、ベクトル命令の立ち上がり時間が支配的になり、ベクトル化の効果が低くなる（図3.4参照）。図4.4の(a)は、ベクトル化の対象である内側のjのループのループ長は10であるため、ベクトル化による高速化はほとんど期待できないことになる。この例では、同図(b)のようにループ長の小さいjのループを外側に、ループ長が2000である外側のiのループを内側に入れ替えることで、ベクトル化される内側ループのループ長を拡大でき、高速化を図ることができる。

```
do i=1,2000                    |  do j=1,10
  x=sqrt(b(i))                 |    do i=1,2000
  do j=1,10                    |      x=sqrt(b(i))
    c(i,j)=a(i,j)+x*j          |      c(i,j)=a(i,j)+x*j
  enddo                        |    enddo
enddo                          |  enddo
```

（a）元のループ　　（b）入れ替え後のループ

図4.4　ループの入れ替え

（3）バンクコンフリクトの削減

バンクコンフリクトは、ベクトル化されたループにおいて、①2次元以上の配列の1次元目の寸法が2のベキ乗、かつ、②配列の2次元目以降をループで使用、のときに発生する。図4.5がその例で、配列dの1次元目が256で、ループではd(k,i)として2次元目を使用しているのが原因である。上の2つの条件が同時に成立しないときにはバンクコンフリクトは発生しないので、図の右側に示したように配列の1次元目を+1してd(ns+1,ns)と変更し、1次元目の寸法を2のベキ乗でないようにすることで、この部分のバンクコンフリクトを削減できる。

```
parameter ( ns=256 )
real*4 a(ns),x(ns),y(ns),d(ns,ns)      →   d(ns+1,ns)
do i=1,ns                                  配列dの1次元目を奇数にし
  y(i)=x(i)+a(i)*d(k,i)                    バンクコンフリクトを削減
enddo
```

図4.5　バンクコンフリクトの発生するループ

4.3　並列化のチューニング

1つのプログラムを複数のCPUで同時に実行する並列処理は、オプション「-P auto」を付けてコンパイルすることで利用でき、並列化の条件（3.3参照）を満たしているDOループは、自動的に並列実行される。並列化した時の性能向上比は、図3.7のように並列化率に左右される。ベクトル化率と同様、並列化率の厳密な測定はできないが、アムダールの法

則のTp として並列処理時のプログラム動作情報（図４．６）の１台目の CPU 時間（Conc.Time(>=1)）を使うことによって、近似的にはつぎのように推定できる。ここで、n は並列処理で使用するCPUの数、Tsは並列化しないときのCPU時間（プログラム動作情報のUser Time）である。

$$並列化率（\alpha）=\frac{n}{n-1}\times\left(1-\frac{Tp}{Ts}\right)$$

```
***** Program Information *****
Real Time (sec)      : 1570.275969
User Time (sec)      : 6217.915341      CPU時間（合計）
     ...             :    ...
VLEN                 :  242.552757
V.Op.Ratio(%)        :   99.643774
Max Concurrent Proc. :            4.    CPU数
 Conc.Time(>=1)(sec) : 2567.214718      １台目のCPU時間Tp
 Conc.Time(>=2)(sec) : 2058.110054      ２台目のCPU時間
 Conc.Time(>=3)(sec) : 1058.094943      ３台目    〃
 Conc.Time(>=4)(sec) :  536.357927      ４台目    〃
               ...          :    ...
```

図４．６　並列処理時のプログラム動作情報

いま、並列化しないときのCPU時間Tsを5900秒とすると、図４．６の例はCPU数(Max Concurrent Proc.) ４のときの１台目のCPU時間Conc.Time(>=1)が2567秒なので、並列化率は約58%であり、性能向上比も2.29倍である。この並列化率では、仮にCPU数を無限に増やしたとしても、性能向上比は最大2.38倍にしかならなず、CPU台数を増やしても意味がないことになる。ベクトル化のときと同様、並列化による性能向上の効果が現れるのも、並列化率が90%を超えるあたりからなので、高速化のためには並列化率を少しでも高めることが必要である。

以下では，自動並列化を妨げる２つのケースについて、その原因と並列化促進のための並列化指示行について述べる。

（１）依存関係

並列化の条件（３．３参照）を満たさないループの他に、その可能性のあるループも並列化されない。その１つが、図４．７に示すループで、前に述べた編集リストで並列化の有無を知ることができる。図で、内側のｉのループはVが付きベクトル化されているが、外側のｊのループは＋が付いているので並列化されていない（並列化されたときはPが付く）。

原因は、並列化の対象となる外側ループの指標ｊが１からnyまで動く間に、iy(j)が同じ値をとる場合は、異なるｊで同一の配列要素a(i, iy(j))を指すことになり、不正な結果となるからである（ベクトル化のときと同様、依存関係があるという）。すなわち、ｊで並列化し

たとき、異なるｊを受け持つ別々のCPUが同一の配列要素への書き込みを行うことになり、不正な結果を招くからである。

したがって、iy(j)がすべて異なる値をとる場合は依存関係はなく、並列化しても正しい結果が得られるので、図４．７のように並列化指示行「!cdir nosync」を用いて並列化することができる。

```
              !cdir nosync          (並列化指示行、カラム１から記述)
+-------------->    do j=1,ny
|V------------->      do i=1,nx
||                      a(i,iy(j))=b(i,j)+c(i,j)
|V-------------       enddo
+-------------       enddo
```

図４．７　並列化されないループ

（２）強制並列化

図４．８に示すループは、サブルーチン呼び出しをするcall文を内部に含んでいるために、並列化の条件を満たさず、並列化されない。サブルーチンsub1さらにはそこから呼び出されるサブルーチンにおいて、並列化の対象である外側ループの指標ｊが１からｎまで動く間に、複数のCPUが配列ｙや他のcommon宣言をした配列等の同じ要素を使うかもしれない、すなわち、依存関係があるかもしれない、というのがその理由である。

したがって、異なるｊで同一の配列要素を使うことがない場合は、正しい結果が得られるので、並列化指示行「!cdir parallel do」を用いて、強制的な並列化を行うことができる。ここで、並列を受け持つCPU相互間で変数ｚを共有してはいけないので、図４．８では「private(z)」でその旨を指示している。

なお、外側ループｊに関しては同一の配列要素を使う可能性があるが、内側のｉのループではその可能性がないという場合は、並列化指示行を内側のdo文の直前に置いて、内側ループを並列化することもできる。

```
!cdir parallel do private(z)
      do j=1,n
        do i=1,n
          z=a(j)+b(j)
          call sub1(z,y(1,i,j))
        end do
      end do
```

図４．８　強制並列化の指示行

5 高速化推進研究活動の成果

スーパーコンピュータをSX-2NからSX-3/44Rに更新したとき、ジョブ1件あたりの処理量は性能に比例して約20倍に増大した。このため、SX-4への更新の際も、1件あたりの処理量はSX3/44Rの10倍以上に増大し、大規模化、長時間化するものと推測した。そこで、SX-4の運用形態の設計に関しては、つぎの2つのことを重視した。1つは、SX-4は自動並列化機能を持っているので、1ノード32台のCPUを1つのジョブで占有して使用できるように、並列処理を運用の中心においた。もう一つは、大規模・長時間ジョブでは実行時間の予測が困難なため、CPU 時間の制限を廃止するものとした。これにより、従来では不可能であった大規模・長時間ジョブの実行がSX-4では可能となった。実際、2000年度には、ジョブ1件あたりの処理量はSX-3/44Rの10倍になり、また、100時間以上のジョブで占めるCPU時間の割合は全体の約75%に達している。

このような大規模・長時間ジョブにおいては、32 台の CPU を使っても数日を要することもあり、ベクトル化率と並列化率を極限まで高めておくことが必要である。そのためには、スーパーコンピュータのハードウェアやコンパイラについての深い知識が要求される。また、自動ベクトル化、自動並列化の機能強化には利用者プログラムに積極的に関わっていく必要もある。そこで、本センターでは高速化を専門に受け持つ担当者を置き、利用者プログラムの高速化を支援することにした。

以下では、いままでに高速化を行ったプログラムのうち主なものについて概略を述べる。また、この高速化推進研究活動で得られた成果を付録に示す。

1997年(10月～12月)の研究活動

○ プログラム1

このプログラムは、実行時間の約20%を占めるFFTが、スーパーコンピュータ用にチューニングしたものではなかった。そこで、FFTを科学技術計算ライブラリASL/SXに置き換えることにより高速化を図った。また、依存関係の可能性があるために、ベクトル化および並列化できないループについて詳細に調べた結果、共に依存関係が無いことが分り、指示行を用いてベクトル化および並列化を行った。

○ プログラム2

このプログラムは、FFTが実行時間の約25%を占めており、かつ1次元のFFTをdoループ内で繰り返し呼び出していた。そこで、科学技術計算ライブラリASL/SXの多重1次元FFTライブラリに置き換えることにより、この部分について約2.5倍程度の高速化を実現した。また、乱数発生組込関数をループ内の複数の箇所で使用しているためベクトル化されていなかったが、乱数を別ループで作成し、ループ本体ではそれを引用するように

書き換えを行い、双方のベクトル化を行った。さらに、依存関係があるためベクトル化も並列化もできなかったループを、作業配列を用いることにより並列化した（ベクトル化をすると巨大な作業配列が必要なるため、並列化を選択した)。

表　5.1　性能向上比（1）

プログラム番号	対オリジナル比	
	ベクトル	8並列
1	1.2倍	7.5倍
2	2.6	58.6 (32並列)

1998年の研究活動

○ プログラム1

このプログラムは、ベクトル化率が低いため並列サーバで実行していた。ベクトル化率が低かったのは、ループの中にサブルーチン呼び出しがあり、その先のサブルーチンにはdoループが無いためベクトル化の対象になるループが存在しないというのが原因であった。そこで、プログラムの構造を変更し、doループをサブルーチン側へ移すことによりベクトル化を行った。また、サブルーチン呼び出しを含んでいるため自動並列化されていないループを指示行を用いて並列化した。

○ プログラム2

このプログラムは、元々ベクトル演算率および並列化率が高かったので、性能向上には結びつかなかった。しかし、コンパイラによるループ交換のために、内側ループのループ長が短くなっているものがあり、指示行を用いてループ交換を抑止することで性能を若干改善した。なお、この件に関してはその後のコンパイラの機能強化により、現在は解決している。

○ プログラム3

このプログラムは、バンクコンフリクトが実行時間の20%程度を占めていた。原因はベクトル化されていたループにおいて、配列の1次元目の寸法が2のべき乗で、かつ2次元目を使っていたためである。そこで、配列の1次元目の寸法を1増やすことでバンクコンフリクトを削減した。また、依存関係があるため並列化されないループを分割し、ループ本体部分の並列化を行った。

○ プログラム４

このプログラムは、実行時間の大部分を占めるループが依存関係のためベクトル化されていなかった。そこで、依存関係のある部分を別ループで計算するように書き換えを行い、ループ本体部分をベクトル化した。また、doループ内でサブルーチン呼出しを多数回行っていたので、doループをサブルーチン側へ移しキャッシュミス時間を削減した。

○ プログラム５

このプログラムは、実行時の30%程度を占めるループにおいて乱数発生組込み関数を複数箇所で使用していたためベクトル化されていなかった。そこで、乱数を別ループで作成し、ループ本体ではそれを引用するようにプログラムを書き換え、双方のベクトル化を行った。さらに、依存関係があるため並列化されていないループを分割し、ループ本体部分の並列化を行った。

○ プログラム６

このプログラムは、実行時間の大きい二重のループにおいて外側および内側のループ長が共に小さかった。そこで、一部書き換えることによりループの一重化を行い、ループ長を大きくした。また、ループ長の小さい内側ループと外側ループとを入れ替えることにより平均ベクトル長を大きくした。さらに、並列処理用サブルーチンPTFORKを使用することによる並列化が可能であったので、その書き換えを行い並列化した。

○ プログラム７

このプログラムは、２つのループで同じ計算をしている部分があったので、その融合を図り、演算数を削減した。また、ループ内で式の変形を行うことにより、演算数の削減も行った。さらに、並列処理用サブルーチンPTFORKを使用することにより並列化が可能であったので、その書き換えを行い並列化した。

○ プログラム８

このプログラムは、ベクトル演算率が99%を超えており、それ以上のベクトル化は困難であった。しかし、バンクコンフリクトが発生していたので、配列の１次元目を奇数に変更することでバンクコンフリクトを削減した。なお、並列処理については依存関係があるため並列化は不可能であった。

表　5.2　性能向上比（2）

プログラム番号	対オリジナル比	
	ベクトル	8並列
1	22.3 倍	347.6倍 (32 並列)
2	1.0	1.0
3	1.5	4.8
4	342.8	—
5	1.4	3.0
6	1.3	5.9
7	2.0	9.0
8	1.1	—

1999 年の研究活動

○ プログラム１

このプログラムは、実行時間の大きい二重の do ループにおいて、外側ループのアンロールによる演算数の大幅な削減が可能であったのでその書き換えを行った。また、バンクコンフリクトが実行時間の 20%を占めていたので、配列の 1 次元目を奇数に変更することによりバンクコンフリクトを削減した。さらに、自動並列化の条件を満たしていたのでコンパイルオプション「-P auto」 を用いることにより自動並列化を行った。

○ プログラム２

このプログラムは、 6.4GB のメモリを使用していたが、それらはほとんどが作業用の配列であった。そのため複数ループを統合することにより、メモリを 1.8GB にした。また、統合の結果、相互に共通する部分の演算が削減できた。また、write 文がループ内にあったので、並びによる write 文に変更し入出力時間を大幅に短縮した。さらに、依存関係の可能性があるため並列化できていなかったループについて詳細に調べた結果、依存関係のないことが判明したので、指示行を挿入することにより並列化した。

○ プログラム３

このプログラムは、ループ長の大きい内側ループがコンパイラによるループ交換のため外側ループになっていた。そこで、指示行 NOLOOPCHG を用いループ交換を抑止した。また、バンクコンフリクト時間が大きかったが、複素数の演算がその原因であった。そこで、実行時間の大きいループを実行する前に実部と虚部に分け、ループ本体では別々に計算し、その後、複素数に合成することによりバンクコンフリクト時間を低減した。

○ プログラム４

このプログラムは、バンクコンフリクトが発生していたため配列の１次元目を奇数に変更することでバンクコンフリクトを削減した。また、実行時間の大きい複数のループにおいてループ長がサブルーチンの引数で渡されていたので、parameter 文を用いてループ長を明示的に指定した。その結果、ループ長の小さい内側ループの自動的なループ展開が行われループ長の大きい外側ループでのベクトル化が可能となった。また、これらのサブルーチンを呼出しているループ内では依存関係の無いことが判明したので、指示行 PARALLEL DO を用いて並列化した。

○ プログラム５

このプログラムは、実行時間の大きい三重の do ループにおいて、最内側のループ長は５、中間のループ長は２であった。そこで、この２つのループを展開した形に書き換えてループを一重化し、ループ長の大きい最外側のループでベクトル化を行った。また、サブルーチン呼出しを含む do ループについては、インライン展開することによりベクトル化を行った。

○ プログラム６

このプログラムは、乱数発生組込み関数を複数箇所で使用していたためベクトル化されていなかった。そこで、乱数を別ループで作成し、ループ本体ではそれを引用することにより双方のベクトル化を行った。また、依存関係があるためベクトル化不可のループについては、内部に存在する多数の if 文の順序を変更することにより実質的な演算数を削減した。

○ プログラム７

このプログラムは、元々ベクトル演算率 、並列化率が高く、大きな性能向上は得られなかった。ただし、FFT を科学技術計算ライブラリ ASL/SX に置換すること、および開発直後の ASL/SX 並列版の固有値ライブラリに置換することにより 1.2 倍の性能向上を得た。

○ プログラム８

このプログラムは SX-4 で実行していたが、ベクトル演算率も低く平均ベクトル長も小さかった。そこで、並列サーバへの移植をするため、do ループ交換によるキャッシュの利用率向上、ループ内でのサブルーチン呼出しのインライン展開を行った。また、粒度の大きいループについて指示行を用いて並列化を行った。さらに、Fortran とＣのサブルーチン間でのデータ受け渡しをファイル経由で行っていたが、配列の行と列を入れ替えた後、直接配列で受け渡すように書き換え、入出力の時間を短縮した。

表　5.3　性能向上比（3）

プログラム番号	対オリジナル比	
	ベクトル	8並列
1	2.1 倍	15.1 倍
2	1.7	6.7
3	4.4	—
4	10.1	73.2
5	4.0	—
6	2.0	—
7	1.2	—
8	10.5	—

2000年の研究活動

○ プログラム1

このプログラムは、実行時間の大きいループがリストベクトルを使用するためベクトル化されていなかった。プログラムを詳細に調べた結果、ループ内の演算数は多く、また、異なる繰り返しで同一の配列要素を使う頻度が少なかったので、新機能として提供されたコンパイルオプション「-Wf-pvctl listvec 」を用いてそのループをベクトル化した。さらに、作業ファイルの入出力を多用していたため、拡張記憶装置 XMU を使用することで、入出力の時間を大幅に短縮した。なお、並列化のチューニングについては「6高速化の事例」を参照されたい。

○ プログラム2

このプログラムは、二重の do ループにおいて内側ループはベクトル化されていたが、漸化式でありループ長も小さかった。一方、外側のループはループ長が大きかったので、プログラムを書き換えループの交換を行った。その結果、新たに内側となったループのループ長は大きく、また、内側ループの漸化式も解消された。さらに、ループの入れ替えに伴なう一部の式の変形によって演算数が削減された。

○ プログラム3

このプログラムは、実行時間の大きい三重の do ループにおいて、最内側のループ長は5、中間のループ長は3であった。そこで、この2つのループを展開した形に書き換えてループを一重化し、ループ長の大きい最外側のループでベクトル化を行った。

なお、並列化については、依存関係が存在するため並列化は不可能であった。

○ プログラム4

このプログラムは、科学技術計算ライブラリ ASL/SX を使用して、複素行列を係数行列とする連立 1 次方程式を解いていたが、 複素数引数型でライブラリを使用していたためバンクコンフリクトが全体の 20%を占めていた。そこで、引用前に実部と虚部に分けて実数引数型のライブラリを引用するように変更しバンクコンフリクトを削減した。

○ プログラム5

このプログラムは、バンクコンフリクトが実行時間の 10%を占めていたため、配列の 1 次元目の寸法を奇数にしバンクコンフリクトを削減した。また、3.5GB の配列を common 領域に取っていなかったため、並列処理では 2CPU しか使用できなかった。原因は、SX-4 でのメモリ制御が common 領域以外は各 CPU ごとに個別に確保するためであり、このプログラムの場合 3 並列以上では物理メモリ量を超えるためである。そこで、その配列を common 領域 に取ることによりこの問題を解消した。

○ プログラム6

このプログラムは、ベクトル演算率が 99%以上であり、並列化の効率も良かった。しかし、ベクトル化された do ループにおいて、ネストされた if 文の一番内側にループ本体があり、かつ条件の成立は 0.1%程度であった。このため、if 文の部分とループ本体の部分にループを分割し、本体部分は条件が成立するときだけベクトル化しないで実行し、高速化した。

○ プログラム7

このプログラムはベクトル演算率が 99%を超えており、ベクトル演算率のそれ以上の向上は困難と思われたが、外側ループのアンロールや内側ループのループ融合を行うことにより 10%程度の性能向上を実現した。なお、並列化については粒度が小さいため効果が得られなかった。

○ プログラム8

このプログラムは、実行時間の大きいループが依存関係のためベクトル化されていなかった。そのループを詳細に調べたところ、ループの前半部分には if 文とそれに伴なう式があり、その部分はベクトル化可能であることが判明した。そこで、ループを分割し、後半は条件が成立するときのみ、ベクトル化しないで実行するように変更した。また、6GB を超えるメモリを使用していたが、プログラムを詳細に調べた結果、計算順序を変更することによりメモリ量を半減することができた。

○ プログラム９

このプログラムは、図 5.1(a)に示すようにｓに依存関係があるためベクトル化されていないループがあった。そこで、これを同図(b)のようにｓを前もって計算する新たなループを作り、ループ本体ではそれを引用するように、書き換えることで双方のループのベクトル化を行った。

```
s=y
do   i=1,n
     sm1(i)=s*.....
     s=x(i)*a+....
     sm2(i)=s*...
 enddo
```

(a) 元のループ

```
ss(1)=y
do   j=2,n+1
        ss(j)=x(j)*a+....
   enddo
  do   i=1,n
     sm1(i)=ss(i)*.....
      sm2(i)=ss(i+1)*.....
enddo
```

(b)書き換え後のループ

図 5.1 依存関係のためベクトル化できないループ

表　5.4　性能向上比（４）

プログラム番号	対オリジナル比	
	ベクトル	８並列
1	1.6倍	5.8倍
2	1.6	—
3	1.8	—
4	1.2	—
5	—	11.4
6	8.0	—
7	2.3	—
8	1.7	—
9	1.5	—

6　高速化の事例

SX-4 の共有メモリ型かつベクトル並列型の特徴を活かすべく、コンパイラには自動ベクトル化と自動並列化の機能が備わっており、通常はこの機能だけで十分な高速化が期待できる。しかし、中には自動ベクトル化および自動並列化の条件を満たさず、SX-4 の性能を活かしきれていないプログラムもある。この場合、ベクトル化および並列化の指示行を用いることで解決できることもあるが、特に並列化の促進を図るときにはプログラム全体にわたる調査を行い、その論理構造を詳細に調べてチューニングを行うこともしばしばである。以下では、その一例について述べる。

6.1　プログラムの解析

チューニングを進めるにあたり、サブルーチン動作情報を採取してプログラムの動作状況を確認した。サブルーチン動作情報は、オプション「-ftrace」を付けてコンパイルしたプログラムの実行が終了した時に、表6.1に示す形式で出力される。これには、サブルーチン単位の FREQUENCY（実行回数）、EXCLUSIVE TIME（実行時間）とその割合（%）、V.OP.RATIO（ベクトル演算率）、AVER.V.LEN（平均ベクトル長）、BANK CONF（バンクコンフリクト時間）等の情報が含まれているので、サブルーチンごとの問題点を把握することができる。

採取した情報を分析した結果、ベクトル性能について以下の事が分かり、本プログラムの単一 CPU での性能をこれ以上向上させることは難しいと判断し、並列処理による高速化を図ることとした。

(1) 実行時間の大きいサブルーチン hsnnl と sibrot のバンクコンフリクトは、複素数演算部分で発生しているため、バンクコンフリクトを回避することは困難である。

(2) サブルーチン ysymch の平均ベクトル長が小さいのは、実行時間の大部分を占めるループのループ長が 40 前後のためであり、外側ループとの入れ替えはできない。

表6.1　サブルーチン動作情報

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	I-CACHE MISS	O-CACHE MISS	BANK CONF
hsnnl	25095	2794.732(38.9)	111.366	2751.5	1361.9	99.38	251.2	13.8280	0.9927	595.6785
gsum	2	852.688(11.9)	426343.831	2429.6	1354.7	99.60	256.0	0.2870	0.0754	0.0251
sibrot	24744	750.774(10.5)	30.342	3028.9	1575.6	99.72	251.1	4.6233	0.2639	206.2873
thdbg	88218	552.690(7.7)	6.265	3116.3	1183.5	99.63	249.4	1.4379	0.1190	0.2483
thdfg	65772	477.101(6.6)	7.254	3153.9	1195.0	99.64	249.3	1.2404	0.1044	0.2279
fftxyz	218829	339.351(4.7)	1.551	1463.1	0.0	99.54	256.0	0.2934	0.0818	97.7244
ysymch	663	250.706(3.5)	378.139	637.5	116.7	92.01	48.2	0.0898	1.0773	0.0000
crgnl	51	242.280(3.4)	4750.596	1788.4	350.2	99.84	255.4	0.1550	0.0414	57.5760
:										

6.2　並列化のチューニング

表6.1のサブルーチン動作情報を元に、実行時間の大きいサブルーチンについて並列化の検討を行った。

(1) サブルーチン hsnnl

プロファイラ（注）による2CPUでの並列化状況を表6.2に示す。実行時間の大きいループが自動並列化されており、表6.2によると、-micro1で示される1台目のCPU時間47.5062ミリ秒と-micro2で示される2台目のCPU時間47.6062ミリ秒が同程度である。すなわちロードバランスが良いので、これ以上のチューニングの必要はないとした。

（注）ここでは、並列処理のロードバランスをプロファイラで調べているが、2001年4月からは、並列処理時も「-ftrace」が使用できるようになり、サブルーチン動作情報の一部としてロードバランスに関する情報が出力されるようになっている。

表6.2　サブルーチン hsnnl の並列化状況

%Res.	T/M	Micro(busy-wait)	CPU	#Calls	msec/call	Name
0.0	0.00	(37.366)	2386.94	50192	0.0000	hsnnl_
24.3		1192.21(20.138)		25096	47.5062	-micro1
24.4		1194.72(17.228)		25096	47.6062	-micro2

(2) サブルーチン gsum

このサブルーチンも、実行時間の大きいループが自動並列化されている。表6.3のプロファイラの結果によると、2CPU使用時のロードバランスが多少悪いものの、大きな問題ではないとした。

表6.3　サブルーチン gsum の並列化状況

%Res.	T/M	Micro(busy-wait)	CPU	#Calls	msec/call	Name
0.0	0.00	(0.050)	250.77	4	0.	gsum$1_
2.8		137.57(0.000)		2	68787.	-micro1
2.3		113.20(0.050)		2	56600.	-micro2

(3)サブルーチン sibrot

図6.1に示すサブルーチンの主要部分において、外側のループ（732行目）が並列化されていない。原因は、外側ループの繰り返しにおいて、一つ前の繰り返しの結果を用いて配列cl1、hcl1、scl1への加算を行っている、すなわち、依存関係が存在するためである。また、ループ内にはサブルーチン呼び出しもある。そこで、734行目と736行目のループの粒度が共に大きいことから、並列化指示行innerを用いて、内側ループの並列化を行った。なお、ループ長（図ではng2の値）が小さい、あるいは、ループ内の演算数が少ない場合は、内側

ループを並列化すると、ベクトル化のみのときよりも遅くなるので注意が必要である。

```
732: +------>          do 100 jb=1,nbnd
733: |                    ctemp=(0.d0,0.d0)
734: |V----->             do 30 ig=1,ng2
735: |V-----       30      ctemp=ctemp+dconjg(sp(ig,jb))*cl1(ig)
736: |V----->             do 50 ig=1,ng2
737: | |                    cl1 (ig)=cl1 (ig)-ctemp*p (ig,jb)
738: | |                    hcl1(ig)=hcl1(ig)-ctemp*hp(ig,jb)
739: | |                    scl1(ig)=scl1(ig)-ctemp*sp(ig,jb)
740: |V-----       50      continue
741: |                    ctemp=-ctemp
742: |                    call ctrns(pbcl1,pby(1,1,1,jb),ctemp,npbc,npbc)
743: +------     100 continue
```

図６.１　サブルーチン sibrot の主要部分

(4)サブルーチン ysymch

図６.２に示すサブルーチンの主要部分において、外側のループ（214 行目）が並列化されていない。原因は、230 行目で定義されている配列 rho の添え字が、別の配列 nwk の内容によって決まる、すなわち、依存関係が存在する可能性があるためである。そこで、k1、k2、k3 の値を調べたところ、外側ループの異なる繰り返しで同じ値の組が現れ、実際に依存関係の存在することが判明したので、並列化することはできなかった。また、内側のループに関しては演算数が少なく、また平均のループ長も 40 前後で、粒度は小さい。そこで、並列化は行わず、このままベクトル化のみで実行することとした。

```
214: +------>          do 10 i=1,nrcell
215: |                    s=0.0d0
216: |                    fac=1.0d0/dble(nwk(i,4))
217: |V----->             do 20 k=1,nwk(i,4)
218: | |                    is=is+1
219: | |                    k1=nwk(is,1)
220: | |                    k2=nwk(is,2)
221: | |                    k3=nwk(is,3)
222: | |                    s=s+rho(k1,k2,k3)
223: |V-----       20 continue
224: |                    s=s*fac
225: |V----->             do 30 k=1,nwk(i,4)
226: | |                    ik=ik+1
227: | |                    k1=nwk(ik,1)
228: | |                    k2=nwk(ik,2)
229: | |                    k3=nwk(ik,3)
230: | |                    rho(k1,k2,k3)=s
231: |V-----       30 continue
232: +------     10 continue
```

図６.２　サブルーチン ysymch の主要部分

(5)サブルーチン crgn1

実行時間の大きい外側のループが並列化されるが、表6.4のプロファイラの結果によると、ロードバランスが悪い。原因は、並列化される外側のループの実行回数が多いときでも2回しかないためである。そこで、内側ループの粒度が大きいことに着目してその並列化を検討したが、依存関係があることが分かり、並列化を断念した。

表6.4　サブルーチン crgn1 の並列化状況

%Res.	T/M	Micro(busy-wait)	CPU	#Calls	msec/call	Name
0.0	0.00		235.24	5406	0.000	crgn1$1_
4.6		226.03		2703	83.622	-micro1
0.2		9.21		2703	3.407	-micro2

(6)FFT 部分

本プログラムは、NCARL FFTPACK を使用して、三次元複素フーリエ変換を行っている。FFT を行っている部分のサブルーチンは約 40 個あり、そのほとんどが一重ループのため、効率的な並列化を行うのは難しい。また、この部分の実行時間は全体の実行時間の約 27%にも達している。そこで、科学技術計算ライブラリ ASL/SX への置換を図るべく、これらすべてのサブルーチンを詳細に調査した結果、ASL/SX の並列機能版への置換が可能であることが判明した。

ASL への置換に際し、この部分で発生していたバンクコンフリクト時間も同時に削減するために、複素数引数型の引用から、実部と虚部に分けて引用する実数引数型に変更した。そのため、元のプログラムと同じ名前のサブルーチンを作成し、その中で ASL/SX の並列機能版三次元複素フーリエ変換（実数引数型）を call し、変換終了後に実部と虚部から複素数に戻した。また、ASL/SX では、データ数が変わらない限り、2 回目以降は「初期値化後の変換」のライブラリを引用することで、因数分解や三角関数テーブルの作成が不要となり、実行時間の短縮につながった。

以上の書き替え作業実施後の 2CPU でのプロファイラの結果を、表6.5に示す。ロードバランスが多少悪いものの、並列化できていることが分かる。

表6.5　並列化チューニング後の 2CPU 実行結果

Seconds				Seconds				Thread/Macro[tid]
%Res.	Res.	T/M	Micro	%CPU	CPU	CPUcum.	Wait	-micro[n]
0.0	4340.98	0.02		0.0	0.02	0.02	0.00	Root1
100.0	-	-	4340.96	53.7	4340.96	4340.98	0.00	-micro1
86.3	-	-	3745.40	46.3	3745.40	8086.38	43.44	-micro2

6.3　並列性能の検討

プログラム全体の性能向上の結果を表６.６に示す（並列効果の算定を１台目のCPU時間Conc.Time(>=1)で比較)。本プログラムは、ベクトル化のチューニング後のベクトル演算率は99.43%、ベクトル長241であり、コストの上位を占めるサブルーチンの性能は、SX-4のピーク性能の70%前後に達していた。そのため、バンクコンフリクトが発生していたものの、並列化に注力したチューニングを実施することとし、８CPU使用時に3.6倍の性能向上を果たした。

表６.６　性能向上比

	Conc. Time(>= 1) (sec)	ベクトル演算率(%)	メモリ (MB)	性能向上比
1CPU	7,241.3	99.4	514.0	1.0
2CPU	4,385.4	98.7	591.0	1.7
4CPU	2,774.9	98.6	689.0	2.6
8CPU	2,022.6	98.4	885.0	3.6

並列化のチューニングにおいて、高いコストのループに依存関係があり、残念ながら十分な並列効果を得るには至らなかった。しかしながら、25%以上のコストを占める三次元フーリエ変換部分を、SX-4用に最適化された科学技術計算ライブラリASL/SXの並列機能版に置き換えることができ、成果を得ることができた。ただし、プログラムが複素数引数型のフーリエ変換を行っていたため、実部と虚部を分けた実数引数型の引用への変更に伴う、複素数から実数、またフーリエ変換後に実数から複素数への変換処理が必要になった。プログラム全体を、実部と虚部に分けて計算するように書き換えると、バンクコンフリクトが減少し、さらなる性能向上を望むことが可能と考えるが、残念ながらプログラムの規模が大きいため、今回はプログラム全体の書き換えは行わなかった。

付　　録

1　南部健一、近藤修司：
「マグネトロン放電プラズマのダイナミックス」
SENAC　Vol.31,No 4 (1998.10)

2　稲岡毅、小久保達信、岡部公起、小野敏：
「ベクトル化と並列化でどれだけ速くなるか？
—微粒子電子励起の計算への応用—」
SENAC　Vol.32,No3・4 (1999.11)

3　小野敏、稲岡毅、小久保達信、伊藤英一、大泉健治、岡部公起：
「スーパーコンピュータ SX-4 の高速化支援について」
全国共同利用大型計算機センター研究開発論文集 No.21

マグネトロン放電プラズマのダイナミックス

東北大学流体科学研究所　　南部　健一
東北大学大学院工学研究科　　近藤　修司

概　要

大型計算機センターのスーパーコンピュータSX-4の大規模な利用により，3次元マグネトロンプラズマがある条件下ではカオス的変動を伴う定常状態をとることが初めて明らかにされた．スーパーコンピューティングが実験を越える日はそう遠くはない．

1．はじめに

近年，携帯電話，パソコン，ワープロなどの電子機械の国内生産額の伸びは急速であり，1996年にはついに我が国産業のトップの座を占めていた自動車等の輸送機械の生産額を追い抜くまでになった．電子機械の主要部品は電子デバイスである．デバイスの作成はプラズマを用いて行なわれる．すなわちシリコンウェハ上に薄い膜を堆積させるCVDやスパッタリングの技術と，堆積した膜を微細加工するエッチング技術が主なものであるが，これらはすべてプラズマを用いて行なわれる．そこでこれらのプラズマを用いた材料処理の技術はプラズマプロセシングと総称されている．プラズマプロセシングは，いまや我が国の基幹産業に成長した電子工業を支える基盤技術である．

半導体チップ，半導体製造用プラズマ装置，コンピュータ等の電子機器の国際市場では，日米が常に激しいシェア争いを展開しており，より高性能の製品を開発するために両国政府も産業界もプラズマプロセシングの研究推進に力を注いでいる．例えば米国政府主導の研究組合SEMATECHが10年ほど前に発足し，これが着実に力を付けて来たことから，日本企業のシェアは低下して来ている．日本政府も数年前に研究組合ASET（超先端電子技術開発機構）を発足させたが，米国優位の現状は変わっていない．

現状を打開するためには，これまでのような実験を主体にした試行錯誤的研究開発方針では無理であろう．すなわち目前に控えた直径300mmの大口径ウェハ，線幅0.2μm以下の超微細加工の時代には，そのような方針では開発費と開発時間がうなぎ登りとなり，企業は疲弊してしまうことになる．一見遠回りに思えても，シミュレーションによりプラズマ装置の最適設計を行なうのが近道である．実際，かつて航空機の開発や原子力開発ではシミュレーションが重要な役割を果たして来た．また我が国は，日本電気，日立，富士通の3社がスーパーコンピュータを供給するという世界に類のない計算機資源に恵まれた国である．

それではなぜプロセシングプラズマの研究にシミュレーションがさほど用いられていないのか，またシミュレーション技法の研究がそれほど進んでいないのか．答えは簡単である．プラズマ装置の中で起こっている物理・化学現象は実に複雑であり，例えば電子と分子C_4F_8が衝突したときC_4F_8がどのように分解するかは，ほとんど分かっていない．C_4F_8は酸化膜エッチングに用いられる最も重要なガスである．さらに数学的な難問もある．すなわち最近のプラズマプロセシングではガス圧力が低下しプラズマ密度が上昇して来ている．このため，プラズマ装置内の電子やイオンを，流体としてではなく粒子として扱わなければならない．つまり学問的に十分な成果の蓄積が有る流体力学は通用せず，ボルツマン方程式やフォッカー・プランク方程式を基にして現象を解析する必要がある．さらに3次元非定常電磁場に対するマクスウェル方程式も解かなければならない．それも1回や2回ではなく10万回は解く必要がある．プロセシングプラズマの性質は電子を抜きにしては語れない．電子は電磁場の変化に敏感に反応し，また電磁場は電子の運動に依存する．すなわち電子をタイムステップΔtだけ動かす度に，電磁場を解析する必要がある．10万ステップの計算はスーパーコンピュータにとっても負担になる．

困難は山積している．しかしだからこそ魅力的な研究対象である．ジャンボジェット機の揚力や抗力は，流体力学の基礎式であるナビエ・ストークス式を，現在のスーパーコンピュータをフルに利用して解けば求められる．風洞実験と一致するデータが今や計算によって得られている．このような計算は大変ではあるがやればできる段階に来ている．しかしプロセシングプラズマのシミュレーション研究は，本質的に挑戦的な様相を帯びている．この大魚を料理するにはどうしても4本の包丁がいる．電子衝突包丁，表面反応包丁，数学包丁，SX-4包丁である．どの包丁も最高の切れ味を持っているよう，いつも磨いていなければならない．これは料理人にとって楽しいことである．

本稿では我々の包丁で料理できた3次元マグネトロン放電プラズマについて述べる．マグネトロンスパッタは，液晶ディスプレイ用パネルのほか，自動車やビルの窓ガラスに機能性薄膜を付着させるのに用いられている．このプラズマの構造をSX-4を利用して調べているうちに，奇妙なことが分かった．すなわちある条件下でプラズマはカオス的変動を伴う準安定状態を示した．この現象に焦点を当てたい．

2. 平板マグネトロンスパッタリングの原理

スパッタリング現象を理解するために，銅板の表面にアルミニウムの薄膜を作成する場合について考えよう．まず銅板とアルミ板をある距離（5～10cm）離して平行に配置し，これを真空容器の中に置く．容器の中にアルゴンガスを流し込み，同時に真空排気をして容器内のガス圧力（1～10mTorr）を一定に保つ．つぎに，銅板（陽極）をアースし，アルミ板（陰極）に負の電圧（-1000～-300V）を印加する．

圧力が低いのでこのままではプラズマは点火しない．図1に示すように，アルミ板の背後に永久磁石のN極とS極を陸上競技場のトラックのへりを作るような形に配置する．この磁石による磁場によって，電子は陰極の真上でトラックを周回するように走るようになる．この運動は$E \times B$ドリフトと呼ばれている．実は，電子は競技者のようにトラックに沿って真直ぐ走るのではなく，トラックの幅一杯に左右に蛇行しながら周回する．あたかも酔っぱらいがトラックを走っているように電子は走る．この蛇行を伴う周回運動によって，電子とAr原子が衝突する確率は著しく増加し，時には

$$\mathrm{e} + \mathrm{Ar} \rightarrow \mathrm{e} + \mathrm{Ar}^{+} + \mathrm{e}$$

によってAr原子は電離し，イオンAr^{+}が生成される．もし磁場Bがなければ，電子は陰極から陽極へ真直ぐ走るだけであり，走る距離が短いので電離衝突は起こりにくい．マグネトロン放電は，低ガス圧力でもプラズマを維持する目的で発明された放電方式である．

さて電子衝突により発生したイオンAr^{+}は電場により加速され，数百eVのエネルギーでAl板（陰極）に激突する．陰極はイオンの標的になるので，ターゲットとも呼ばれる．このときアルミ板の表面近くの原子がはじき出される（スパッタされる）．すなわち

$$\mathrm{Ar}^{+} + \mathrm{Al\,(solid)} \rightarrow \mathrm{Ar} + \mathrm{Al} + \mathrm{e}$$

ここにeはAl原子と同時に放出される2次電子である．Al原子はガス原子Arと衝突しながら対向する銅板の方向に拡散し，ついには銅基板（陽極）上に体積して薄膜を形成する．2次電子は$E \times B$ドリフト中に電離衝突を起こし新たなイオンAr^{+}を生成する．すなわちスパッタされたAl原子と同時に放出される2次電子によって，放電が維持される．以上がマグネトロンスパッタリングによる薄膜作成の原理である．

3．成膜予測研究のシナリオ

以上の原理は古くから知られており，この原理を用いたスパッタリング装置も実用化されている．しかし近年，形成された薄膜の性質に対する要求が非常に厳しくなって来た．例えば，膜厚の一様性と膜結晶組織の均一性に対する要求である．これらの要求を満たすためには，陽極面にやってくるAl原子のフラックス分布（どこに何個のAl原子が飛来するか）とエネルギー分布（どのようなエネルギーを持ったものがどれだけ飛来するか）をシミュレーションによって予測してから，スパッタリング装置の設計を行なわなければならない．この予測を行うには，次の3つの研究が必要である．

（A）プラズマ解析
（B）スパッタリング解析
（C）スパッタ原子流解析

まず（A）のプラズマ解析によって，陰極（ターゲット）面の各部に入射するイオンのフラックスと速度分布関数が分かる．この2つの量が分かると，(B) のスパッタリング解析によって放出されるスパッタ原子（Al）のターゲット面でのフラックスと速度分布関数が分かる．この（B）の結果を初期条件として，(C) のスパッタ原子流解析を行なえば，陽極（基板）面上でのスパッタ原子のフラックス分布とエネルギー分布関数が分かる．したがってこの2つの量から膜厚分布が予測でき，かつ膜結晶組織の均一性についてもある程度の推測が可能となる．

上記3つの研究のうち，(A) は最も難度が高く，我々がSX-4を用いて得た研究成果が世界最初の本格的なものである．応用物理学会主催の第3回国際反応性プラズマ会議における招待講演で成果を発表したところ，シミュレーションはここまで来たのかという驚きの入り混じった反響があった．オーストラリア国立大学プラズマ研究所のBoswell教授から，オーストラリアで開催予定の国際会議に招待するのでこの研究をもう一度講演して欲しいと頼まれたが，修士論文の発表日程とぶつかり引き受けられなかった．

シミュレーションを，実験家が羨ましがるような完成度の高いものにしようという考えで，これまで研究してきた．実験データは物理現象の真実を与える．実験に比肩しうるシミュレーションを行なうためには，妥当性不明の仮定を排し，また近似を用いるにしても，常に近似に起因する誤差が最終結果に影響を及ぼさないように監視している必要がある．我々はまず，電子－アルゴン原子衝突の断面積セットとして，最も信頼性の高いものを捜し，林真氏の未発表データを使うことにした．それでも心配なので，同氏のデータを用いて電子のドリフト速度，拡散係数，電離係数を換算電界の広い範囲にわたって計算し，Phelps教授提供の最新の実験データと比較して見た．実験との一致は完璧であった．林氏の断面積セットは，弾性衝突および電離衝突のほか，25種の電子励起衝突を含む．これらの合計27種の衝突事象を従来のモンテカルロ法で扱うと計算能率が非常に悪いことが分かったので，ただ1つの乱数で衝突事象を決定できる新しい方法を考え出した．この方法を作るさい，最近まで南部が行なっていたボルツマン方程式の確率解法に関する研究がヒントになった．つぎはAr^+とArの衝突の取り扱いであるが，これまで発表されているものは電荷交換の取り扱いが近似にすぎ，満足できなかった．そこでAr^+のドリフト速度の実験データを再現できる新しい衝突モデルを構築した．最後の難問は3次元電場の解析であった．空間電荷の分布はPIC法（Particle-in-Cell method）で求めればよいのだが，差分形の3次元ポアソン方程式をいかに高速に解くかが問題であった．多重格子SOR法も検討したが，マグネトロン放電ではプラズマの大部分がトラック状磁場に捕捉され陰極付近にのみ局在することから，電場に垂直なx，y方向には周期条件を課し高速フーリエ変換（FFT）を適用した．これは非常にうまく行った．またSX-4用にNECが手を加えた同社提供のFFTのライブラリーはCRAY

C916のFFTより高速なことも，我々を喜ばせた．スーパーコンピュータの製造会社は，汎用の数学ライブラリーIMSLを提供するだけではユーザーに文句を言われるだろう．自社のコンピュータのアーキテクチャに適合するようチューニングし高速化したライブラリーを提供すべきである．さて電場に平行なz方向にはThomas法を用いた．この部分はベクトル化が困難であり，計算時間全体のかなりの部分を占めていた．しかし大型計算機センターの岡部先生のご指導により並列化に成功し，かなり処理速度が速くなった．

いままで（A）のプラズマ解析について述べてきた．次は（B）のスパッタリング解析であるが，この分野の研究は，核融合炉炉壁の高速中性子によるスパッタリングに関連してかなり進んでおり，この分野の第一人者である山村教授（岡山理科大学）のデータを使えばこと足りる．ただしターゲット表面の薄い酸化膜をイオンで打ち破りながら下地の金属をスパッタする場合のデータがまだない．このようなデータは，酸素を含む反応性スパッタリングの研究に欠かせない．

最後は（C）のスパッタ原子流の解析であるが，この解析に必要なテスト粒子モンテカルロ法は，大分以前に我々のグループがすでに開発した．この方法を用いればシミュレーションから求めた膜厚分布が実験値と一致することも確かめてある．

4．マグネトロンプラズマの動的構造

円管の中に色水を流して色水の様子を見てみよう．流量が少ないうちは静かな流れ（層流）になっているが，流量をあげて行くと色水の流れは乱れ活発な乱流になる．我々の身の周りの流れはほとんど乱流である．乱流は空間時間的にカオス的変動を伴うが，大きな時間スケールにわたって平均してしまうと定常流れである．マグネトロンプラズマにも静かなプラズマと乱れたプラズマがあることが，SX-4を用いた大規模計算によって分かって来た．この研究を国際会議で最初に発表したところ，マグネトロンスパッタリングの実験をしていたドイツのある研究者が，コーヒーブレイクのとき私のもとに来てこんなことを言った．「マグネトロンプラズマの実験をしていると，トラック状磁場のあちこちで光のスポットが蛍のように点滅する．点滅する時間間隔は不規則だし，点滅する位置も一定していない．この現象はあなたの理論計算で説明できそうだ」と．SENAC本号の表紙を見ていただきたい．陰極に平行な平面における電子密度を表わしているが，トラック状磁場の数箇所に赤いピークが見える．すなわち電子密度が急上昇しているスポットがある．これは，スポット附近で電離衝突が多いことを示している．電離と電子励起のしきい値エネルギーにさほど大きな差はないので，スポット附近では励起衝突も多いであろう．励起原子の失活により，赤いピーク附近で発光が起こると考えられる．また表紙の絵から発光スポットの位置が空間的にも時間的にも変化することも分かる．

マグネトロンプラズマの動的構造を見ておこう．計算領域を$256 \times 128 \times 250$の

格子点に分割した．電極間距離が50mm，ガス圧力が5mTorr，印加電圧が−300V，磁場の強さが125Gの場合を考える．座標系のとり方を図1に示す．陰極に垂直にz軸をとる．原点は陰極面の中央に位置する．系がカオス的定常状態に入った時刻を$t=0$とし，$0.114\mu s\,(=\tau)$ごとに物理量をサンプリングした．図2は陰極の中央面$(x=0)$における電位ϕの分布を示す．3次元のシースが形成されている．電位の時間変動は小さく，ほとんど見えない．図3は同じ中央面における電子密度の分布である．トラック状磁場の強さがトラックの上下で対称になっているのに，電子密度はyの正負で対称になっていない．また電子密度の分布が時間とともに変化することも分かる．分布は2つの山から成るが，右の山の尾根形状の時間変化を図4に示す．分布は連続的に変化するものの，無周期の変化でありカオス的である．尾根の高さはほぼz=9mmでピークをとるが，z=9mmの平面における電子密度の分布を示したものが表紙絵である．

SX-4の能力は相当なものである．しかし既存の物理モデルや解析手法をそのまま用いたのでは，SX-4でも1000時間や2000時間の計算時間が必要になろう．これは研究者が頭を使わずスーパーコンピュータを酷使する研究方法である．得られた結果がよほど素晴らしいものでないかぎり，このような論文は評価されない．まず研究者がこれまでにない高能率で高精度な手法を開発し，これを用いてさらにSX-4を限界まで使用するなら，SX-4も納得してくれよう．

なおここでは静かなマグネトロンプラズマのデータはすべて省略した．

5. おわりに

プロセシングプラズマのシミュレーション研究は，我が国の電子工業にとって今後ますます重要性を増すであろう．研究は次の点を念頭に置いて進めなければならない．

（イ）微細パターン底部の帯電によるダメージとノッチングを防止する．

（ロ）高アスペクト比のエッチングを高速で行なう．

（ハ）プラズマ発生機構を単純化し，プラズマ装置の製造コストを下げる．

（イ）の目的のためには，負イオンを含むプラズマを用い，外部入力や基板バイアスを時間変調する必要がある．（ロ）のためには，低ガス圧力で高プラズマ密度を実現しなければならない．したがってプラズマの発生も従来の平行平板高周波放電方式から，主力は誘導結合や波動励起または表面波を用いた方式に移行しつつある．（ハ）も重要であり，例えば強い外部磁場を必要とする電子サイクロトロン共鳴装置は，プラズマ発生原理は優れているものの装置コストが高く，市場で苦戦を強いられている．

学問的には低ガス圧力で高密度プラズマを実現する必要性から，いわゆる流体力学的取り扱いは破綻するので，次のように考えて行かなければならない．

(i) ガス分子や中性ラジカルの流れは，ボルツマン方程式の確率解法であるDSMC法で扱う．

(ii) 電子やイオンは，電磁場のマクスウェル方程式と荷電粒子に対するボルツマン方程式の連立解法であるPIC/MC法で扱う．

(iii) 高密度プラズマでは荷電粒子間の衝突が効いてくる．これはフォッカー・プランク方程式の解法である南部の集積散乱角法で扱う．

以上の取り扱いによる計算負荷は，従来の流体モデルによるものに比べて桁違いに大きい．しかし流体モデルが成立しない以上，他に道はない．複雑な現象を記述する高精度でしかも簡単な物理モデルを構築し，これらに合った計算スキームを開発したのちSX-4の性能を最大限に引き出せば，不可能な研究などないであろう．ベールに覆われた真理の姿はいつも美しい．その姿を見たいという一心は純粋である．

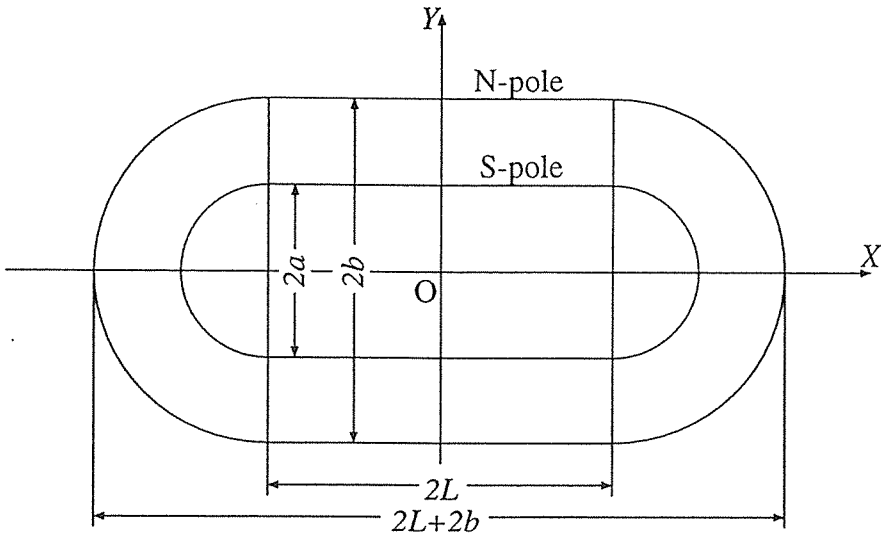


図 1 陰極（ターゲット）背面に設置した永久磁石の配置.
($a = 50\,\mathrm{mm}$, $b = L = 100\,\mathrm{mm}$).

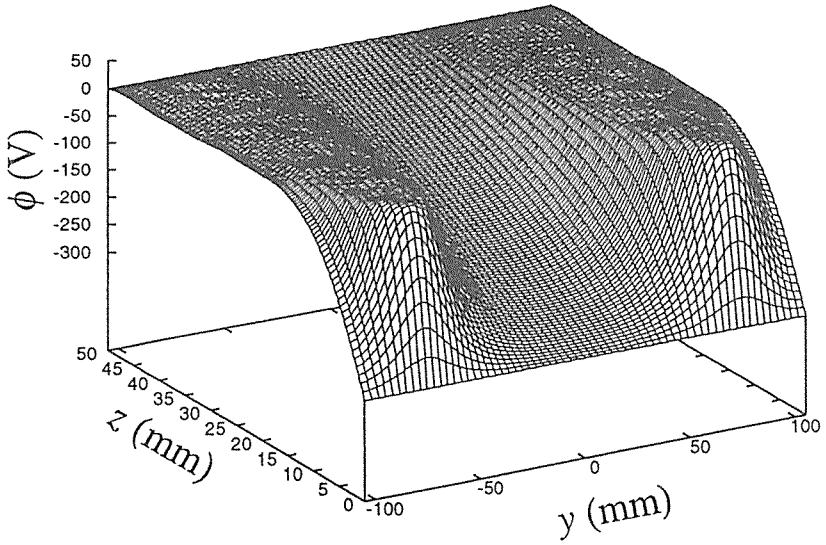


図 2 yz面（$x = 0$）における空間電位 ϕ.

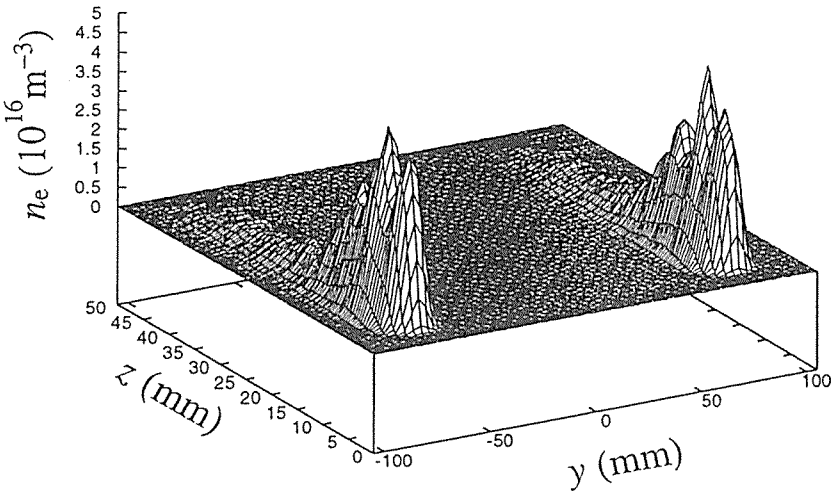


図 3 yz面（$x = 0$）における電子数密度 n_e.
(a) $t = 0$

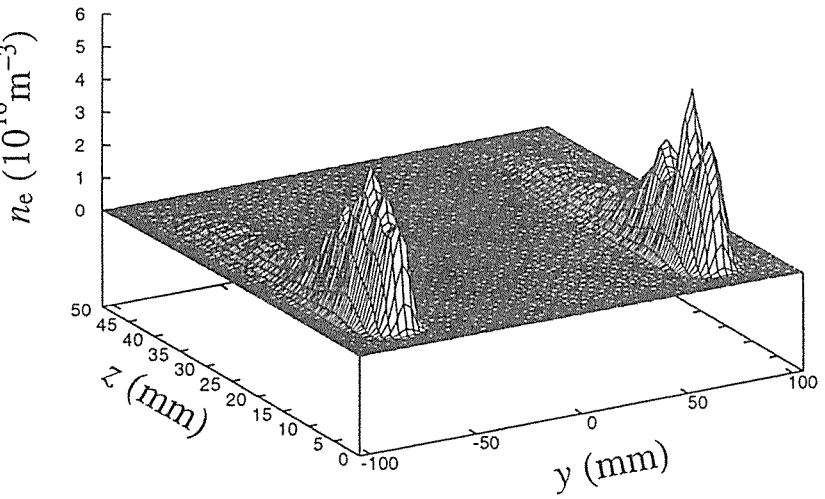


図 3 yz面（$x = 0$）における電子数密度 n_e.
(b) $t = 2\tau$

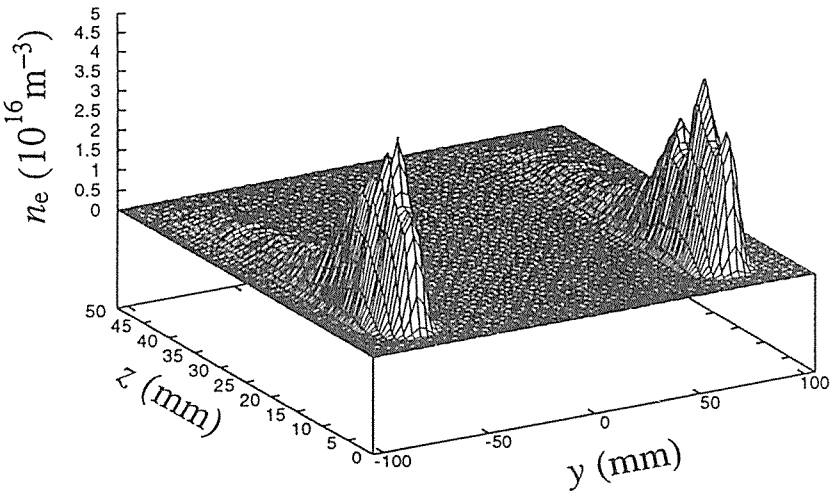


図 3 yz面（$x = 0$）における電子数密度 n_e.
(c) $t = 4\tau$

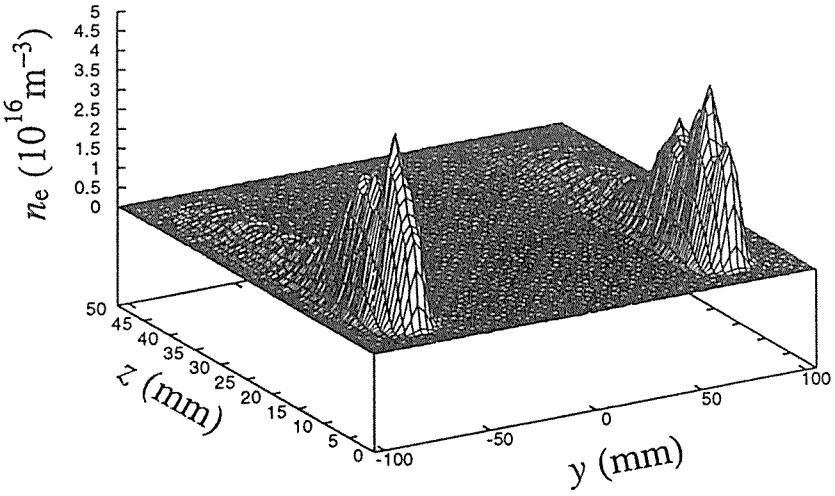


図 3 yz面（$x = 0$）における電子数密度 n_e.
(d) $t = 6\tau$

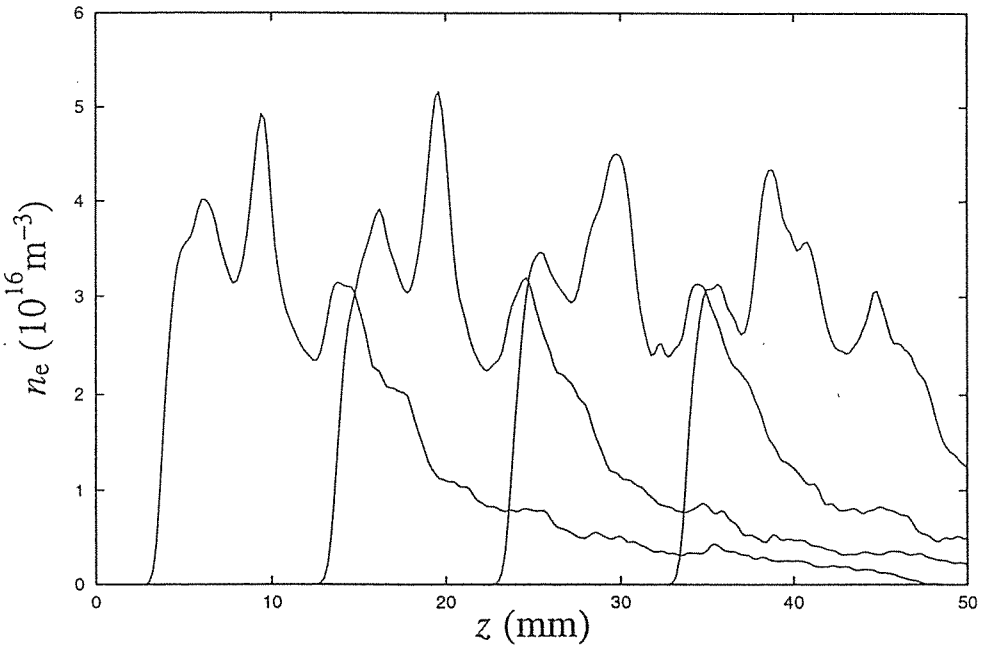


図 4 尾根（$y = 75.5$mm）における電子数密度 n_e.
左からそれぞれ時刻 $t = 0, 2\tau, 4\tau, 6\tau$ におけるもの.
z座標の原点は時刻の増加とともに 10mm ずつシフトして描いてある.

ベクトル化と並列化でどれだけ速くなるか？
－微粒子電子励起の計算への応用－

岩手大学工学部材料物性工学科　稲岡 毅
日本電気第1コンピュータ事業本部　小久保 達信
東北大学大型計算機センター　岡部 公起，小野 敏

1. はじめに

最近，数多くの大学や研究機関に並列型のスーパーコンピュータが導入され，東北大学大型計算機センターでも128台のCPUを有するシステムSX-4が稼働しています．このようなコンピュータでは，ベクトル化とともに並列化を有効に活用することにより，著しい高速化が達成できるのですが，並列化については，有効な活用方法がまだ研究されている段階にあります．

この記事では，微粒子電子励起の計算を例にとり，ベクトル化と並列化をどのように実施して，どの程度の高速化が達成できたかを報告します．コンピュータを使って数値計算を行う際，多くの場合，まず解析的に定式化を行い，得られた方程式をプログラム化します．定式化により得られた方程式を忠実に数値計算するために作られたプログラムも，ベクトル化と並列化という別の視点から検討してみると，いろいろと改良点が見つかります．このような改良を加えて，ベクトル化と並列化の両方が十分に機能したときは，顕著な高速化が実現されます．

2. 微粒子電子励起の計算

最初に，微粒子伝導電子系の励起の計算について述べます．微粒子電子系は表面効果，量子サイズ効果が顕著に現れる興味深い系であり，その電子物性のサイズ依存性を研究する究極的目的の1つは，系のサイズが原子・分子からバルクへと大きくなるに連れて，電子的性質がどのように移り変わっていくかを明らかにすることです．この目的のためには，小さいサイズからなるべく大きいサイズまで幅広いサイズ領域で計算する必要があるのですが，サイズが大きくなっていくと，計算規模，とりわけ演算時間が急激に増加してしまいます．電子系の基底状態の計算と比べてずっと多くの演算時間を要する電子励起の計算の場合，この問題は一層深刻になります．このため，どのくらい大きいサイズまで計算できるかは，計算機の性能と使い方に大きく依存することになります．微粒子伝導電子系の励起モードは，微視的には複数の電子遷移過程から構成されています．電子遷移過程のレベルで視た電子励起（双極子表面プラズモン）の内部構造が，幅広いサイズ領域でどのように移り変わっていくかについては，文献[1]をご覧下さい．このような電子励起を調べるためには，角振動数ωで振動し，球面調和関数$Y_{LM}(\theta,\phi)$の角度分布を持つ外部ポテンシャルが，微粒子の伝導電子系に加えられたときに起こる動的応答を計算します．計算内容の詳細は文献[1]に譲りますが，計算の中心部分は，動径座標ρを変数とするポテンシャル$V(\rho)$に関する次のような積分方程式を数値的に解くことです．なお，以下の表式に現れる物理量は，すでに無次元化されています．

$$V(\rho) = \rho^L + \int_0^1 d\rho' \left[\int_0^1 d\rho'' K(\rho, \rho'') \chi(\rho'', \rho')\right] V(\rho') \qquad (1)$$

ここで，核 $K(\rho, \rho')$，応答関数 $\chi(\rho, \rho')$は次のように与えられます．

$$K(\rho, \rho') = (\rho')^2 \left[(\rho_<)^L / (\rho_>)^{L+1} - \rho^L (\rho')^L \right] \qquad (2)$$

$$\chi(\rho, \rho') = \sum_{n,l}' \sum_{l'} A(l, l', L)(\rho')^2 \frac{j_l(\alpha_{ln}\rho)\, j_l(\alpha_{ln}\rho')}{\{j_{l+1}(\alpha_{ln})\}^2}$$

$$\times \left[G_{l'}(\rho, \rho'; \alpha_{ln}^2 + \omega) + G_{l'}^*(\rho, \rho'; \alpha_{ln}^2 - \omega) \right] \qquad (3)$$

(2) 式中の $\rho_>$，$\rho_<$は，それぞれ ρ，ρ' の大きい方，小さい方を表します．(3) 式の $\chi(\rho, \rho')$は電子系の動的応答を記述する関数であり，励起を構成する電子遷移過程からの寄与の和で与えられます．この式の中の n (=1,2,3,…)，l あるいは l' (=0,1,2,…)はそれぞれ1電子状態を指定するための動径量子数，角運動量量子数を表し，Σ' は占有状態について和を取ることを意味します．$A(l, l', L)$は角運動量選択則を含む係数であり，ウィグナーの $3j$ 記号を含む形で書かれます．ここでは，$L = 2$，すなわち4重極励起の場合を考えることにしますと，選択則 $l' = l \geq 1$，あるいは，$l' - l = \pm 2$ を満足するとき以外は，$A(l, l', L) = 0$ になります．従って，(3)式中の l' についての和は簡単に実行でき，上記の選択則を満足するごく限られた項のみが残ります．関数 $j_l(x)$は l次の球ベッセル関数を，α_{ln} は $j_l(x)$の n 番目の正の零点を表します．グリーン関数 $G_l(\rho, \rho'; E)$は，Eの符号に応じて，球ベッセル関数 $j_l(x)$と $n_l(x)$，あるいは変形球ベッセル関数 $i_l(x)$と $k_l(x)$を用いて表すことができます．式(1)は $V(\rho)$に関する線形積分方程式なので，積分範囲に分点を取り，各分点における Vの値が未知数であると考えると，この積分方程式は連立1次方程式に帰着されます．

3. プログラム原版の構成

最初のプログラムは，ベクトル化や並列化を特に意識しないで，定式化により得られた方程式を忠実に数値計算するもので，メインプログラムと7つのサブルーチンより成ります(表4. 1参照)．このうち，最も多くの演算時間を要するのは，応答関数を計算するサブルーチン SUB1, GLPET, GLNET で，$\chi(\rho, \rho')$を計算するために，ρと ρ' の値の組に対して，量子数 lと n についての和を計算しています．主要部分はつぎのような構造で，これを原版と呼ぶことにします．

ここで，NRH は ρの分点の数，M は ρ，ρ' の値の組を指定する指標です．また，LとNのループは，量子数 lと nについての和を計算するためのループであり，その中のGLPETとGLNET

は，それぞれ $E>0$, $E<0$ の場合の $G_l(\rho,\rho';E)$を計算するためのサブルーチンです．また，引数 RH, RHP はそれぞれ ρ, ρ' を，VGLPERとVGLNERは $G_l(\rho,\rho';E)$の実部，VGLPEI と VGLNEI は虚部を表します．

[原版]

```
SUBROUTINE SUB1(…, NRH, ACHIR, ACHII)
DIMENSION ACHIR(NRH*NRH), ACHII(NRH*NRH)
……
do M=1, NRH*NRH
  SMCHR=0.0
  SMCHI=0.0
  do L=0, LOMX
    do N=1, NOEL(L)
      ……
      call GLPET(…, RH, RHP, VGLPER, VGLPEI)  …スカラ変数
      ……
      call GLNET(…, RH, RHP, VGLNER, VGLNEI)  …スカラ変数
      ……
      SMCHR=SMCHR+(GLPET, GLNET の結果)
      SMCHI=SMCHI+(GLPET, GLNET の結果)
    enddo
  enddo
  ACHIR(M)=ACHIR(M)+(SMCHR を利用)
  ACHII(M)=ACHII(M)+(SMCHI を利用)
enddo
……
END
SUBROUTINE GLPET(…, RH, RHP, VGLPER, VGLPEI)
REAL RH, RHP                                  …入力 (スカラ変数)
REAL VGLPER, VGLPEI                           …出力 (スカラ変数)
……
END
SUBROUTINE GLNET(…, RH, RHP, VGLNER, VGLNEI)
REAL RH, RHP                                  …入力 (スカラ変数)
REAL VGLNER, VGLNEI                           …出力 (スカラ変数)
……
END
```

4. 高速化の方針

まず，高速化の方針を決めるために，原版の動作状況を確認することにします．SX-4 の f90 コンパイラには高速化を支援するための機能があり，オプション-ftraceを指定してコンパイルすると，プログラム実行後にサブルーチン単位での動作情報を採取することができます．

この機能を用いて採取したのが，表4.1に示す原版のサブルーチン動作情報です．サブルーチン単位の，FREQUENCY(実行回数)，EXCLUSIVE TIME(実行時間)とその割合(%)，V.OP.RATIO(ベクトル演算率)，AVER.V.LEN(平均ベクトル長)，BANK CONF(バンクコンフリクト)等の情報を得ることができます．表4.1より，つぎのことが分かります．

(1) 実行時間は GLPET に集中している(90.5%)．

(2) GLPET と GLNET の実行回数(呼び出される回数)が非常に多い．

(3) GLPET と GLNET のベクトル演算率は0であり，ベクトル化されていない．

(4) バンクコンフリクトは発生していない．

表4.1 原版のサブルーチン動作情報(抜粋)

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	BANK CONF
glpet	222265625	26783.77(90.5)	57.0	32.0	0.00	0.0	0.00
glnet	18359375	1800.60(6.1)	53.7	32.4	0.00	0.0	0.00
sub1	2	820.79(2.8)	36.8	10.4	55.09	3.4	0.09
sub4	2	94.77(0.3)	616.0	174.1	97.03	32.9	0.12
sub2	2	49.84(0.2)	1047.5	363.2	98.96	124.1	1.26
main_	1	30.50(0.1)	292.5	161.6	79.06	196.3	0.00
sub3	2	0.07(0.0)	1102.0	321.1	98.92	105.5	0.00
initalp	1	0.00(0.0)	20.5	0.0	0.00	0.0	0.00

原版のプログラムは，ρとρ'の値の各組に対して量子数lとnについて計算してから和をとっていますが，微粒子のサイズが小さいときは，これでも，性能上特に問題にはなりません．しかし，微粒子のサイズが大きくなっていくと，ρの分点の数とLとNで和を取らなければならない項の数が多くなり，特に分点の数については，GLPETとGLNETの呼び出し回数がρの分点の数の2乗に比例して増加することになります．これが，上の(1)と(2)の原因と考えられます．また，(3)については，GLPETとGLNETが1組の分点ρ，ρ'だけについての計算をしているために，ベクトル化の対象になっていないことが原因と考えられます．

このようなことに注目して，つぎの高速化の方針をたてました．

(ⅰ) ループL，Nとループ Mを入れ替えて，ループ Mの方が内側になるようにする．

(ⅱ) ループ MをGLPETとGLNETの中に移し，すべての分点の計算を1回の呼び出しで行うようにする．これによって，サブルーチンの呼び出し回数を減らす．

(ⅲ) GLPETとGLNETの中でループ Mをベクトル化する．

(ⅳ) 量子数lとnについては，独立に計算できるため，新たに外側になったループL，Nについて並列化を行う．

次章以降で，このような対策(チューニング)を順次講じていったとき，演算時間がどのように短縮されていくかを示します．

5. ベクトル化による高速化

5. 1 ループMの移動とSUB1，GLPET，GLNETのベクトル化

SUB1の中で，スカラー変数であったSMCHR，SMCHIをNRH*NRHの大きさのワーク配列とすることで，ループL，Nとループ Mを入れ替え，ループ Mを内側に移動できます．また，GLPETとGLNETのサブルーチン内部で使用している中間変数(スカラ)をNRH*NRHの大きさの配列とすることで，ループ Mを最内側のループにすることができます．これを高速化1版とします．具体的には，次のようにプログラムコードを書き換えました(GLNETはGLPETと同様の手法で書き換えましたので，以下では省略します)．この時点の性能情報を表5.1に示します．

[高速化１版]

```
      SUBROUTINE SUB1(…,NRH,ACHIR,ACHII)
      DIMENSION ACHIR(NRH*NRH),ACHII(NRH*NRH)
      DIMENSION SMCHR(NRH*NRH),SMCHI(NRH*NRH)      配列として宣言
      DIMENSION RH(NRH*NRH),RHP(NRH*NRH)                 〃
      DIMENSION VGLPER(NRH*NRH),VGLPEI(NRH*NRH)          〃
      DIMENSION VGLNER(NRH*NRH),VGLNEI(NRH*NRH)          〃
      ……
      do M=1, NRH*NRH
        SMCHR(M)=0.0
        SMCHI(M)=0.0
      enddo
      do L=0,LOMX
        do N=1,NOEL(L)
          ……
          call GLPET(…,NRH,RH,RHP,VGLPER,VGLPEI)    …配列
          ……
          call GLNET(…,NRH,RH,RHP,VGLNER,VGLNEI)    …配列
          ……
          do M=1,NRH*NRH
            SMCHR(M)=SMCHR(M)+(GLPET,GLNET の結果(配列))
            SMCHI(M)=SMCHI(M)+(GLPET,GLNET の結果(配列))
          enddo
        enddo
      enddo
      do M=1,NRH*NRH
        ACHIR(M)=ACHIR(M)+(SMCHR(M)を利用)
        ACHII(M)=ACHII(M)+(SMCHI(M)を利用)
      enddo
      ……
      END
      SUBROUTINE GLPET(…,NRH,RH,RHP,VGLPER,VGLPEI)
      DIMENSION RH(NRH*NRH),RHP(NRH*NRH)              入力(配列)
      DIMENSION VGLPER(NRH*NRH),VGLPEI(NRH*NRH)       出力(配列)
      DIMENSION xxx(NRH*NRH),…        中間情報を保存するための配列(大きさ NRH*NRH)
      ……
      do M=1,NRH*NRH                  中間変数を配列としたためベクトル化可能
        xxx(M) = RH(M),RHP(M),...     中間情報を配列へ
        ……
      enddo
      ……
      do M=1,NRH*NRH
        VGLPER(M) = xxx(M),...        中間情報は配列から
        VGLPEI(M) = xxx(M),...        中間情報は配列から
      enddo
      ……
      END
```

表 5.1 高速化1版のサブルーチン動作情報(抜粋)

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN
glpet	569	2484.17(90.8)	2721.2	908.0	99.85	256.0
glnet	47	89.10(3.3)	2571.3	895.1	99.82	255.4
sub4	2	84.94(3.1)	682.9	196.7	98.22	34.1
sub2	2	50.25(1.8)	1038.9	360.2	98.96	124.1
main_	1	17.34(0.6)	565.4	289.8	82.06	187.5
sub1	2	10.01(0.4)	1982.0	808.4	99.94	256.0
sub3	2	0.05(0.0)	1113.9	428.8	98.95	93.1
initalp	1	0.00(0.0)	20.1	0.0	0.00	0.0

★★★★★高速化効果(1)★★★★★

SUB1の実行時間が820秒から10秒に大きく性能向上しました．また，GLPET，GLNETの中にループMを移動することで，GLPETとGLNETの呼び出し回数が劇的に減少しました．さらに，中間変数を配列とすることでGLPET，GLNETをベクトル化できました．この結果，実行時間については，GLPETが26,783秒から2,484秒に，GLNETが1,800秒から89秒と，大幅な性能向上を得ることができました．しかし，ワーク配列を利用したことにより，メモリが約42MBから約430MBに増加しました．

5．2 ワーク領域の削減，さらなるベクトルチューニング

次にワーク配列の削減を実施します．高速化1版では，中間配列の大きさを単純にループ長としていましたが，ループ長を256単位に短く刻むことを行います．こうすることでNRH*NRHの大きさの配列を，256の大きさの配列にすることができます．これで所要メモリを削減できます．

ここで問題となるのは，ループ長が短くなることによるベクトル性能の劣化です．単純にループを短く刻んだ場合，ベクトル演算器を立ち上げるためのオーバーヘッドの割合が多くなり，性能が劣化します．しかし，ループ長がベクトルレジスタ長と同じであれば，コンパイラの最適化により，データをベクトルレジスタに載せたまま演算することが可能で，この条件に合えば性能が向上します．ワーク配列削減の変更で，すべての最内側ループのループ長をベクトルレジスタ長と強制的に一致させています．この結果，副次的ですがループ長を短くしたマイナス要因を抑えて，コンパイラによる最適化の効果が加わり，性能劣化を超えて性能向上が期待できます．このチューニングを実施した結果を高速化2版とします．具体的には，次のようにプログラムコードを書き換えました．

[高速化2版]

```
      SUBROUTINE GLPET(…,NRH,RH,RHP,VGLPER,VGLPEI)
      DIMENSION RH(NRH*NRH),RHP(NRH*NRH)            入力(配列)
      DIMENSION VGLPER(NRH*NRH),VGLPEI(NRH*NRH)     出力(配列)
      parameter(MVL=256)
      dimension xxx(0:MVL-1),…        中間情報を保存する配列(大きさ256)
      do MM=1, NRH*NRH, MVL           ループを256飛びに刻む
        M0=min(NRH*NRH-MM,MVL-1)
!cdir nodep,shortloop
        do M=MM,MM+M0                 ループ長は256
          xxx(M) = RH(M),RHP(M)
          ……
        enddo
        ……
!cdir nodep,shortloop
        do M=MM,MM+M0                 ループ長は256
          VGLPER(M) = xxx(M),...
          VGLPEI(M) = xxx(M),...
        enddo
      enddo
      ……
      END
```

表 5.2 高速化2版のサブルーチン動作情報(抜粋)

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN
glpet	569	2058.37(89.7)	2882.9	1196.8	99.90	256.0
sub4	2	84.90(3.7)	683.3	196.8	98.22	34.1
glnet	47	73.51(3.2)	2823.7	1232.0	99.90	255.4
sub2	2	49.50(2.2)	1054.5	365.6	98.96	124.1
main_	1	17.32(0.8)	566.1	290.1	82.06	187.5
sub1	2	10.34(0.5)	1920.8	783.1	99.85	256.0
sub3	2	0.05(0.0)	1107.9	426.5	98.95	93.1
initalp	1	0.00(0.0)	20.5	0.0	0.00	0.0

★★★★★高速化効果(2)★★★★★

ループ長を 256 に短く刻むことで, NRH*NRH の大きさのワーク配列を 256 の大きさのワーク配列とすることができました. この結果, 使用メモリが約430MBから約70MBに削減されました.

また, ループ長が256(ベクトルレジスタ長)であることを利用し, コンパイラに対するベクトル指示行 !cdir shortloop を指定することで, 副次的ですがベクトルレジスタの有効活用が図られ性能が向上しました. この結果, GLPET の実行時間が 2,484 秒から 2,058 秒となりました(ベクトル化指示行の詳細は文献[2]を参照してください).

5.3 GLPET:条件計算で真率を利用したチューニング

GLPETの計算において, 条件計算で理論上真率が10%程度の部分があります. この部分をベクトル化するため, 高速化2版では, マスク演算を使いました. このため, 偽であってもマスク演算が実行されます. 刻んだループがすべて偽であった場合, 演算しないようにコードを変更し, 演算数の削減を行います. このチューニングを実施した結果を高速化3版とします. 具体的には, 次のようにプログラムコードを書き換えました.

[高速化2版の GLPET 主要部分]

```
!cdir nodep,shortloop
      do M=MM,MM+M0
        IF(XXL(M).LT.XCJ1) then          <-- マスクの設定
          flag1(M)=.true.
        else
          flag1(M)=.false.
        endif
      enddo

!cdir nodep,shortloop
      do M=MM,MM+M0                      <-- マスク演算を実行
        if(flag1(M)) then
          ......                         <-- GLPET 主要部分, 演算多数
        endif
      enddo
```

[高速化3版の GLPET 主要部分]

```
      run_t_1=-1
      run_f_1=-1
!cdir nodep,shortloop
      do M=MM,MM+M0
        IF(XXL(M).LT.XCJ1) then      <-- マスクの設定
          flag1(M)=.true.
          run_t_1=run_t_1+1
        else
          flag1(M)=.false.
          run_f_1=run_f_1+1
        endif
      enddo

      run_1=-1                       <-- 真と偽が交じっていると判断
      if(run_f_1.eq.M0) run_1=0      <-- 偽のみと判断
      if(run_t_1.eq.M0) run_1=1      <-- 真のみと判断

      if(run_1.eq.-1) then           <-- 真と偽が交じっている場合:
!cdir nodep,shortloop                    マスク演算を実行
        do M=MM,MM+M0
          if(flag1(M)) then
          ......                     <-- GLPET 主要部分，演算多数
          endif
        enddo
        ......
      else if(run_1.eq.1) then       <-- 真のみ:通常のベクトル演算を実行
!cdir nodep,shortloop
        do M=MM,MM+M0
          ......                     <-- GLPET 主要部分，演算多数
        enddo
        ......
      endif                          <-- 偽のみ:演算を実行しない
```

表 5.3 高速化3版のサブルーチン動作情報(抜粋)

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN
glpet	569	1087.52(82.1)	2731.5	1165.4	99.74	256.0
sub4	2	84.97(6.4)	682.7	196.7	98.22	34.1
glnet	47	73.45(5.5)	2826.3	1233.1	99.90	255.4
sub2	2	50.11(3.8)	1041.7	361.2	98.96	124.1
main_	1	17.41(1.3)	563.2	288.6	82.06	187.5
sub1	2	10.35(0.8)	1919.5	782.5	99.85	256.0
sub3	2	0.05(0.0)	1114.2	428.9	98.95	93.1
initalp	1	0.00(0.0)	20.2	0.0	0.00	0.0

★★★★★高速化効果(3)★★★★★

条件計算で真率を利用したコード変更により，GLPET の実行時間が 2,058 秒から 1,087 秒に性能向上しました.

5．4 ベクトルチューニングによる性能情報

1CPUでの性能情報をまとめます．ベクトルチューニング後，ベクトル演算率は 7%から 99%に改善され，平均ベクトル長は 18.7 から 226.6 へと長くなり，演算性能は 32.5MFLOPS から 1062 MFLOPS へと向上しました．使用メモリは 42MB から 69MB へと増加しています．

表 5.4 ベクトル実行での性能情報

	ＣＰＵ時間	ベクトル演算率	平均ベクトル長	MFLOPS	使用メモリ
原版：	29580.3 sec	7.44 %	18.7	32.5	42.0 MB
高速化１版：	2735.9 sec	99.81 %	241.4	871.1	430.0 MB
高速化２版：	2294.0 sec	99.85 %	239.6	1134.2	69.0 MB
高速化３版：	1323.9 sec	99.66 %	226.6	1062.0	69.1 MB

6. 並列化による高速化

ベクトル化のチューニングを行った後の高速化3版では，サブルーチン SUB1 は次のようになっています．

[高速化3版のサブルーチン SUB1]

```
      SUBROUTINE SUB1(…,NRH,ACHIR,ACHII)
      DIMENSION ACHIR(NRH*NRH),ACHII(NRH*NRH)
      DIMENSION SMCHR(NRH*NRH),SMCHI(NRH*NRH)      配列として宣言
      DIMENSION RH(NRH*NRH),RHP(NRH*NRH)             〃
      DIMENSION VGLPER(NRH*NRH),VGLPEI(NRH*NRH)      〃
      DIMENSION VGLNER(NRH*NRH),VGLNEI(NRH*NRH)      〃
      ……
      do M=1, NRH*NRH
        SMCHR(M)=0.0
        SMCHI(M)=0.0
      enddo
      do L=0,LOMX
        do N=1,NOEL(L)
          ……
          call GLPET(…,NRH,RH,RHP,VGLPER,VGLPEI)   …配列
          ……
          call GLNET(…,NRH,RH,RHP,VGLNER,VGLNEI)   …配列
          ……
          do M=1,NRH*NRH
            SMCHR(M)=SMCHR(M)+(GLPET,GLNET の結果(配列))
            SMCHI(M)=SMCHI(M)+(GLPET,GLNET の結果(配列))
          enddo
        enddo
      enddo
      do M=1,NRH*NRH
        ACHIR(M)=ACHIR(M)+(SMCHR(M)を利用)
        ACHII(M)=ACHII(M)+(SMCHI(M)を利用)
      enddo
      ……
      END
```

「4. 高速化の方針」で書いたように，量子数 l と n についての和は独立に計算できるため並列化の対象として利用します．この方針にしたがって，外側のループ L, N について並列化を行います．ここで外側のループ L, N で並列化するときの問題点は，つぎのようになります．

(1) L 単独，N単独のループではループ長が短いため並列台数(最大 32)に届きません．そこでループ L, N を融合し，ループ長を長くする方法をとります．

(2) SMCHR, SMCHI はループ L, N で総和計算となります．そこで，並列に実行されるタスクごとに得られる部分和(ローカル)を，後でグローバル領域に集める(総和)ように工夫する必要があります．

[並列化版]

```
      SUBROUTINE SUB1(…,NRH,ACHIR,ACHII)          <--引数はグローバル領域
      DIMENSION ACHIR(NRH*NRH),ACHII(NRH*NRH)
      DIMENSION SMCHR(NRH*NRH),SMCHI(NRH*NRH)     <--サブルーチン内での配列は
      DIMENSION RH(NRH*NRH),RHP(NRH*NRH)              ローカル領域
      DIMENSION VGLPER(NRH*NRH),VGLPEI(NRH*NRH)
      DIMENSION VGLNER(NRH*NRH),VGLNEI(NRH*NRH)
      dimension ln_l(NLT*(LLT+1))
      dimension ln_n(NLT*(LLT+1))
      save ln_l,ln_n,ln_max       <--配列を共有するために save 文を用い,
                                      グローバル領域とします.
!cdir serial                      <--ループ L, N を融合するための処理.
      ln_max=0                        並列動作時, 複数のタスクで同時に実行しないように
      do L=0,LOMX                     serial セクションで区切ります.
        do N=1,NOEL(L)                ln_l, ln_n にループ融合したときのLとNに関する
          ln_max=ln_max+1             指標情報が入ります.
          ln_l(ln_max)=L
          ln_n(ln_max)=N
        enddo                         ln_l, ln_n をグローバル領域としたため
      enddo                           すべてのタスクから参照できます.
!cdir end serial
      do M=1,NRH*NRH              <--ローカル配列の初期化のための処理.
        SMCHR(M)=0.0                  SMCHR, SMCHI はローカル配列です.
        SMCHI(M)=0.0                  このローカルな配列は, すべてのタスクで
      enddo                           初期化されます.
!cdir pardo                       <--並列化の指示
      do ln_loop=1,ln_max             融合されたループ L, N, ここで並列化されます.
         l=ln_l(ln_loop)              ln_loop を指標にして各タスクに分割されます.
         n=ln_n(ln_loop)
         ……
         CALL GLPET(……)           <--並列化された各タスクから呼出されます.
         ……
         CALL GLNET(……)
         ……
         do M=1,NRH*NRH           <--並列化された各タスクでのローカルの総和
           SMCHR(M)=SMCHR(M)+(GLPET,GLNET の結果)
           SMCHI(M)=SMCHI(M)+(GLPET,GLNET の結果)
         enddo
         ……
     enddo
!cdir serial                      <--グローバルの総和の準備(初期化)
      do M=1,NRH*NRH                  ACHIR,ACHII はグローバル領域です.
         ACHIR(M)=0.0                 複数のタスクで同時に実行しないように
         ACHII(M)=0.0                 serial セクションで区切ります.
      enddo
!cdir end serial
!cdir critical                    <--グローバルの総和計算, critical セクション
      do M=1,NRH*NRH                  により, タスク間で排他的に実行されます.
         ACHIR(M)=ACHIR(M)+(SMCHR(M)を利用)
         ACHII(M)=ACHII(M)+(SMCHI(M)を利用)
      enddo
!cdir end critical
      ……
      END
```

この変更を施した結果を並列化版とします．具体的には，前ページのようにプログラムコードを書き換えました．総和計算を並列化するために，ローカル領域(SMCHR,SMCHI)で部分和を取った後，グローバル領域(ACHIR,ACHII)に排他的(critical 処理)に集めるように計算方法を変更しています(並列化指示行の詳細は文献[3]を参照してください)．

表6 並列実行での性能情報(並列化効果)

	Real Time (sec)	User Time (sec)	MFLOPS (conc.)	ベクトル演算率(%)	メモリ (MB)	並列化率 (%)	SPEED UP
1CPU	1323.9	1323.2	1062.0	99.66	69.1	-	1.0
8CPU	194.5	1328.2	7235.7	99.64	275.0	97.3	6.8
16CPU	131.6	1369.2	11535.8	99.62	510.0	96.0	10.1
32CPU	85.1	1508.8	16613.9	99.54	980.0	96.5	15.5

(注) 並列化率：1CPU の実行時間と比較してアムダール則から計算
SPEED UP：1CPU の実行時間と比較した性能向上度

7. おわりに

微粒子電子励起の計算を例にとり，プログラムの高速化を図るために，どのようにしてベクトル化と並列化のチューニングを行ったかを紹介しました．原版のプログラムでは，29,580 秒であった演算時間が，まずベクトル化のチューニングにより 1,324 秒になり，22 倍もの高速化が実現できました．さらに，32CPU を用いた並列化ではわずか 85 秒になり，原版に対し実に 347 倍も高速になりました．

この結果は，ベクトル化と並列化の両方が十分に機能したときのスーパーコンピュータの素晴らしい性能を如実に示しています．高速化によってユーザ自身が効率良く研究を進めることができるのは勿論ですが，年度末などは多数のジョブがシステムに滞留することを考えますと，高速化のための努力は多数のユーザのジョブを円滑に処理するためにも必要であると言えます．この報告により，高速化に熱意を感じるようになっていただければ，著者一同これに過ぎる喜びはありません．

参考文献

[1] T. Inaoka, Journal of the Physical Society of Japan Vol. 66, No. 12, 1997, pp. 3908-3921.
[2] FORTRAN90/SX プログラミングの手引，日本電気㈱
[3] FORTRAN90/SX 並列処理機能利用の手引，日本電気㈱

スーパーコンピュータ SX-4 の高速化支援について

小野敏[*1]　稲岡毅[*2]　小久保達信[*3]
伊藤英一[*1]　大泉健治[*1]　岡部公起[*1]

[*1]東北大学大型計算機センター、[*2]岩手大学、[*3]日本電気株式会社

1.　はじめに

東北大学大型計算機センター(以下、本センターと略す)では、共有メモリ型のベクトル並列型スーパーコンピュタ SX-4/128H4(以下、SX-4 と略す)を運用している[1]。SX-4 は、4 つのノードから構成されており、1 ノードあたり 32 台の CPU を持っている。本センターでは、この 128 台という CPU を効率よく運用するために、並列処理を中心とした運用を行っており、利用者も積極的に並列処理を使用している[2]。特に、大規模・長時間ジョブほどこの傾向が強く、ベクトル化率ならびに並列化率も高い。しかし、ベクトル化率あるいは並列化率が低く、SX-4 の性能を十分に活かしきれていないプログラムもある。このようなプログラムは、SX-4 の持っている自動ベクトル化機能、自動並列化機能だけでは対応できず、高速化のためには、ベクトル化/並列化指示行の利用やプログラムの書き換えが必要となる例が多い。そこで、センターでは高速化を専門に受け持つ担当者を置き、ベクトル化率ならびに並列化率の向上を図り、利用者プログラムの高速化を支援している。

本稿では、スーパーコンピュータのジョブ処理状況、センターでの高速化支援の体制、ベクトル化ならびに並列化のチューニング例について述べる。

2.　ジョブ処理状況

SX-4 の運用を開始した平成 10 年 1 月から平成 11 年 6 月までのジョブ処理状況について述べる。ジョブクラスは、s(非並列)、p8(8 並列)、p16(16 並列)、p32(32 並列)であり、会話型処理についてはここでは述べない。

表 2.1 はジョブの CPU 時間が 1、10、100 時間未満とそれ以上についての分布を示したものである。CPU 時間が 1 時間未満のジョブの件数の割合は 80.1%となっているが、それらの CPU 時間は全体の 3.9%にすぎない。一方、100 時間以上の長時間ジョブにおいては、件数は 1.2%にすぎないが、CPU 時間は全体の 67.8%を占めており、大規模計算に使用されていることが分かる。

表 2.2 は各ジョブクラスごとの処理件数と CPU 時間の割合を示したものである。s クラスは処理件数で 89.9%、CPU 時間で 14.2%を占めている。p8 クラスの処理件数 5.2%は、p16 と p32 クラスの合計件数より多く、中規模程度の計算に利用されているものと考えられる。p16、p32 クラスでは処理件数がそれぞれ 2.5%、2.4%であるのに対して、CPU 時間は 24.1%、51.8%となっており、この 2 つのジョブクラスの CPU 時間が全体の 75%以上を占めている。p16、p32 クラスが大規模・長時間ジョブに積極的に利用されていることが窺える。

表 2.3 は p32 クラスにおける並列化効率を 3 ヶ月ごとに示したものである。並列化効率は、総 CPU 時間／(最長 CPU 時間*使用 CPU 台数)で求めている。平成 10 年 10 月～12 月以外は、並列化効率 70%以上のジョブが常に 50%を超えている。通常、ベクトル化率・並列化率の向上には時間のかかることが多く、数週間を要することもある。平成 10 年 10 月～12 月に並列化効率の低いジョブの割合が大きいのは、繁忙期に入り利用者がチューニングに時間を割く余裕がなかったためであると思われ、今年度は、しっかりとした対応が必要であると考えている。

表 2.1　ジョブ処理件数の分布

ジョブの CPU 時間	処理件数	CPU 時間
1 時間未満	80.1%	3.9%
10 時間未満	14.9	6.0
100 時間未満	3.8	22.3
100 時間以上	1.2	67.8

表 2.2　ジョブクラスごとの処理状況

クラス名	処理件数	CPU 時間
s	89.9%	14.2%
p8	5.2	9.9
p16	2.5	24.1
p32	2.4	51.8

表 2.3 ジョブクラス p32 における並列化効率

	平成 10 年 1 月～3 月	4 月～6 月	7 月～9 月	10 月～12 月	平成 11 年 1 月～3 月	4 月～6 月
10% 未満	11%	12%	7%	4%	18%	3%
10%～	4	4	1	2	2	8
20%～	7	5	2	8	11	7
30%～	3	2	2	21	4	7
40%～	6	3	4	9	1	9
50%～	7	6	7	11	1	7
60%～	7	7	6	5	4	7
70%～	18	10	15	7	6	14
80%～	14	8	24	9	10	22
90%～	23	43	32	24	43	16

3. 高速化支援体制

本センターでは、他では実行できないような大規模・長時間ジョブを実行してもらうために、SX-4 の運用開始とともに CPU 時間の制限を廃止した。これにより、表 2.1 からも分かるように、10 時間以上のジョブで占める CPU 時間は 90%を超えており、CPU 時間の制限廃止の効果が現れている。また、表 2.2 からは、p8、p16、p32 の並列処理のクラスで処理される CPU 時間の合計が 85%を超え、並列処理が長時間ジョブで積極的に利用されていることが分かる。しかし、表 2.3 に示すように、並列化効率の低いプログラムもあり、このようなプログラムはベクトル化率も低いことが多い。そこで、自動ベクトル化、自動並列化機能だけでは対応しきれないプログラムについては、通常のプログラム相談とは別に、高速化支援を専門に受け持つ担当者を置き、利用者の支援を行ってきた。また、これとは別に統計情報を常時監視することによって、チューニングの必要な利用者には高速化の必要性を促している。

プログラムの高速化においては、指示行の利用や簡単なプログラムの書き換えで高速化できることも多いが、書き換えが広範囲に渡る場合や計算方法の変更が必要な場合は、プログラム作成者の協力も必要となる。また、このようなチューニングは、コンパイラの強化につながることもあり、メーカも積極的に参加している。このように、利用者、センター、メーカが三者一体となってプログラムの高速化を行っている。代表的なプログラムの性能改善例を表 3.1 に示す。なお、性能向上比は、相談を受けた時点に対してのものである。

表 3.1 プログラム性能改善例

プログラム	主な改善点	性能向上比
1	作業配列使用によるベクトル化 ループをサブルーチン内へ移動しベクトル化 ループ一重化による並列数の増大	347.6
2	最適化処理の抑止 並列化指示行挿入による並列化	11.8
3	ライブラリ置換(FFT) 並列化指示行挿入による並列化	3.5
4	ライブラリ置換(FFT) 作業配列使用により乱数生成組込関数を含むループのベクトル化 作業配列を使用、依存関係の除去による並列化	28.5
5	作業配列使用、指示行挿入によるベクトル化 並列化指示行挿入による並列化	5.0
6	組込関数の適用(誤差関数) 作業配列使用によるベクトル化	342.8
7	作業配列使用によるベクトル化 配列宣言の変更、依存関係の除去による並列化	3.0
8	ループの一重化、入替えによるループ長の増大 マクロタスク使用による並列化	5.9
9	アルゴリズム改善による演算数の削減 マクロタスク使用による並列化	9.0
10	ライブラリ置換(逆行列) ループをサブルーチン内へ移動しベクトル化 マクロタスク使用による並列化	54.4

4. ベクトル化のチューニング

ベクトル化のチューニング例として、表 3.1 のプログラム 1 について述べる。

4.1 プログラム原版

このプログラムは、ベクトル化や並列化を特に意識しない、定式化により得られた方程式を忠実に数値計算するもので、メインプログラムと7つのサブルーチンより成っている(図 4.1)。SX-4 の f90 コンパイラのオプション-ftrace を指定すると、図 4.1 に示すようにプログラム実行後にサブルーチン単位での動作情報を採取することができる。左から順に、サブルーチン名、実行回数、CPU 時間とその割合、ベクトル演算率、平均ベクトル長、バンクコンフリクトである。演算時間を要するのは、サブルーチン GLPET、GLNET、SUB1 であり、主要部分の構造を図 4.2 に示す。

```
PROG.UNIT  FREQUENCY    EXCLUSIVE          V.OP     AVER.     BANK
                        TIME[sec](  %  )   RATIO    V.LEN     CONF
glpet      222265625    26783.77(90.5)      0.00      0.0     0.00
glnet       18359375     1800.60( 6.1)      0.00      0.0     0.00
sub1               2      820.79( 2.8)     55.09      3.4     0.09
sub4               2       94.77( 0.3)     97.03     32.9     0.12
sub2               2       49.84( 0.2)     98.96    124.1     1.26
main               1       30.50( 0.1)     79.06    196.3     0.00
sub3               2        0.07( 0.0)     98.92    105.5     0.00
initalp            1        0.00( 0.0)      0.00      0.0     0.00
```

図 4.1 原版のサブルーチン動作情報(抜粋)

```
SUBROUTINE SUB1(…, NRH, ACHIR, ACHII)
REAL*8 ACHIR(NRH*NRH), ACHII(NRH*NRH)
REAL*8 SMCHR, SMCHI, RH, RHP, VGLPER, VGLPEI, VGLNER, VGLNEI
……
do M=1, NRH*NRH
    SMCHR=0.0
    SMCHI=0.0
    do L=0, LOMX
        do N=1, NOEL(L)
            ……
            call GLPET(…, RH, RHP, VGLPER, VGLPEI)    ! スカラ変数
            ……
            call GLNET(…, RH, RHP, VGLNER, VGLNEI)    ! スカラ変数
            ……
            SMCHR=SMCHR+(GLPET, GLNET の結果)
            SMCHI=SMCHI+(GLPET, GLNET の結果)
        enddo
    enddo
    ACHIR(M)=ACHIR(M)+(SMCHR を利用)
    ACHII(M)=ACHII(M)+(SMCHI を利用)
enddo
……
END
SUBROUTINE GLPET(…, RH, RHP, VGLPER, VGLPEI)
REAL*8 RH, RHP                                  ! 入力 (スカラ変数)
REAL*8 VGLPER, VGLPEI                           ! 出力 (スカラ変数)
REAL*8 xxx, xxl, . . .                          ! 中間変数
……
END
SUBROUTINE GLNET(…, RH, RHP, VGLNER, VGLNEI)
REAL*8 RH, RHP                                  ! 入力 (スカラ変数)
REAL*8 VGLNER, VGLNEI                           ! 出力 (スカラ変数)
REAL*8 xxx, xxl, . . .
……
END
```

図 4.2 プログラム原版

4.2 ループ交換とベクトル化

原版では、ループ構造がM、L、Nとなっているが、L、N、Mの構造にしても物理的に問題ないのでループの入れ替えを行う。

(1)SUB1 の中のスカラー変数 SMCHR、SMCHI を配列とする。
(2)ループ L、N とループ M を入れ替え、ループ M を内側にする。
(3)ループ M を GLPET、GLNET の中に移す。
(4)GLPET、GLNET 内部で使用している中間変数(スカラー)を配列とする。
(5)GLPET、GLNET をループ M でベクトル化する。
これを高速化1版とし図 4.3 に示す(GLNET は GLPET と同様の手法で書き換えたので、以下では省略)。
図 4.4 は、高速化 1 版の動作情報で次のように性能が向上した。
(1)GLPET はベクトル演算率が 99.90%となり実行時間は 2,058 秒に減少した。
(2)GLNET はベクトル演算率が 98.22%となり実行時間は 85 秒に減少した。
(3)SUB1 はベクトル演算率が 99.85%となり実行時間は 10 秒に減少した。

```
      SUBROUTINE SUB1(…, NRH, ACHIR, ACHII)
      REAL*8 ACHIR(NRH*NRH), ACHII(NRH*NRH)
      REAL*8 SMCHR(NRH*NRH), SMCHI(NRH*NRH)              ! 配列として宣言
      REAL*8 RH(NRH*NRH), RHP(NRH*NRH)                   !      〃
      REAL*8 VGLPER(NRH*NRH), VGLPEI(NRH*NRH)            !      〃
      ……
      do M=1, NRH*NRH
         SMCHR(M)=0.0
         SMCHI(M)=0.0
      enddo
      do L=0, LOMX
         do N=1, NOEL(L)
            ……
            call GLPET(…, NRH, RH, RHP, VGLPER, VGLPEI)  ! 配列
            ……
            call GLNET(…, NRH, RH, RHP, VGLNER, VGLNEI)  ! 配列
            ……
            do M=1, NRH*NRH
               SMCHR(M)=SMCHR(M)+(GLPET, GLNET の結果(配列))
               SMCHI(M)=SMCHI(M)+(GLPET, GLNET の結果(配列))
            enddo
         enddo
      enddo
      do M=1, NRH*NRH
         ACHIR(M)=ACHIR(M)+(SMCHR(M) を利用)
         ACHII(M)=ACHII(M)+(SMCHI(M) を利用)
      enddo
      ……
      END
      SUBROUTINE GLPET(…, NRH, RH, RHP, VGLPER, VGLPEI)
      REAL*8 RH(NRH*NRH), RHP(NRH*NRH)                   ! 入力(配列)
      REAL*8 VGLPER(NRH*NRH), VGLPEI(NRH*NRH)            ! 出力(配列)
      parameter(MVL=256)
      REAL*8 xxx(0:MVL-1), xxl(0:MVL-1), ...             ! 中間情報用作業配列(大きさ 256)
      do MM=1, NRH*NRH, MVL                              ! ループを 256 飛びに刻む
         M0=min(NRH*NRH-MM, MVL-1)                       ! (作業配列削減の為)
!cdir nodep, shortloop                                  ! 中間変数を配列にしたため
                                                        ! ベクトル化可能
         do M=MM, MM+M0                                 ! ループ長は 256
            xxx(M) = RH(M), RHP(M), ..                  ! 中間情報を配列へ
            ……
         enddo
         ……
!cdir nodep, shortloop
         do M=MM, MM+M0                                 ! ループ長は 256
            IF(xxl(M).LT.XCJ1) THEN
               ……                                       ! GLPET の主要部分、演算多数
               VGLPER(M) = xxx(M), ...                  ! 中間情報は配列から
               VGLPEI(M) = xxx(M), ...                  ! 中間情報は配列から
            ENDIF
         enddo
      enddo
      END
```

図 4.3 高速化 1 版

```
PROG.UNIT      FREQUENCY     EXCLUSIVE          V.OP     AVER.
                           TIME[sec](  %  )    RATIO    V.LEN
glpet            569       2058.37( 89.7)     99.90    256.0
sub4               2         84.90(  3.7)     98.22     34.1
glnet             47         73.51(  3.2)     99.90    255.4
sub2               2         49.50(  2.2)     98.96    124.1
main_              1         17.32(  0.8)     82.06    187.5
sub1               2         10.34(  0.5)     99.85    256.0
sub3               2          0.05(  0.0)     98.95     93.1
initalp            1          0.00(  0.0)      0.00      0.0
```

図 4.4　高速化 1 版のサブルーチン動作情報(抜粋)

4.3　条件計算で真率を利用したチューニング

GLPET のベクトル化したループ M の計算の中で、IF 文による条件計算がある。そこで、条件が成立する割合を調べたところ、真率が10%程度であった。このため、最初にマスクの設定を行い、それに従って真と偽が交ざった場合、真だけの場合、偽だけの場合についてそれぞれループを実行するようにした。ただし、図 4.3 からも明らかなように、偽だけの場合は何もしない。このチューニングを実施した結果を高速化2版とし図 4.5 に示す。また、この時の動作情報を図 4.6 に示す。この結果 GLPET の実行時間は、1,087 秒に減少した。

```
        runT1=-1
        runF1=-1
!cdir nodep,shortloop
        do M=MM,MM+MO
            IF(xx1(M).LT.XCJ1) then              ! マスクの設定
                flag1(M)=.true.
                runT1=runT1+1
            else
                flag1(M)=.false.
                runF1=runF1+1
            endif
        enddo
        runflag=-1
        if(runF1.eq.MO) runflag=0                ! 偽のみと判断
        if(runT1.eq.MO) runflag=1                ! 真のみと判断

        if(runflag.eq.-1) then                   ! 真と偽が交じっている場合
!cdir nodep,shortloop
            do M=MM,MM+MO
                if(flag1(M)) then
                    ......                       ! GLPET 主要部分，演算多数
                endif
            enddo
            ......
        else if(runflag.eq.1) then               ! 真のみ:通常のベクトル演算を実行
!cdir nodep,shortloop
            do M=MM,MM+MO
                ......                           !  GLPET 主要部分，演算多数
            enddo
            ......
        endif                                    ! 偽のみ:演算を実行しない
```

図 4.5　高速化2版の GLPET 主要部分

```
PROG.UNIT FREQUENCY     EXCLUSIVE          V.OP     AVER.
                        TIME[sec](  %  )    RATIO    V.LEN
glpet          569     1087.52( 82.1)     99.74    256.0
sub4             2       84.97(  6.4)     98.22     34.1
glnet           47       73.45(  5.5)     99.90    255.4
sub2             2       50.11(  3.8)     98.96    124.1
main_            1       17.41(  1.3)     82.06    187.5
sub1             2       10.35(  0.8)     99.85    256.0
sub3             2        0.05(  0.0)     98.95     93.1
initalp          1        0.00(  0.0)      0.00      0.0
```

図 4.6 高速化2版のサブルーチン動作情報(抜粋)

5. 並列化のチューニング

サブルーチンSUB1でのループ交換とベクトル化、サブルーチンGLPET、GLNETへのループの移動とベクトル化、および、真率を利用したチューニングを行い、表5.1に示すように原版に対して22倍の高速化が実現できた。

表5.1 ベクトル化による性能情報

	CPU時間 (sec)	ベクトル 演算率(%)	平均ベクトル長	使用メモリ (MB)	性能向上比
原版：	29580.3	7.44	18.7	42.0	1.0
高速化1版：	2735.9	99.81	241.4	69.1	10.8
高速化2版：	1323.9	99.66	226.6	69.1	22.3

さらなる高速化のためには、ベクトルチューニングは限界であるため、並列化を行うこととした。

(1)新たに外側になったループL、Nについては、独立に計算できるため並列化が可能である。

(2)L単独、N単独のループではループ長が短いため並列台数(最大32)に届かない。そこでループL、Nを融合しループ長を長くする。

(3)SMCHR、SMCHIはループL、Nで総和計算となる。そこで総和計算を並列化するために、ローカル領域(SMCHR,SMCHI)で部分和を取った後、グローバル領域(ACHIR,ACHII)に排他的(critical処理)にめるようにした。

この変更を施した結果を並列化版とし、図5.1に示す。32CPUを用いた並列化では、原版に対して347倍の性能向上となった。並列実行での性能情報を表5.2に示す。

表5.2 並列実行での性能情報(並列化効果)

	Real Time (sec)	CPU時間 (sec)	ベクトル 演算率(%)	メモリ (MB)	並列化率 (%)	性能向上比
1CPU	1323.9	1323.2	99.66	69.1	-	1.0
8CPU	194.5	1328.2	99.64	275.0	97.3	6.8
16CPU	131.6	1369.2	99.62	510.0	96.0	10.1
32CPU	85.1	1508.8	99.54	980.0	96.5	15.5

(注) 並列化率 :1CPUの実行時間と比較してアムダール則から計算
性能向上比:1CPUの実行時間と比較した性能向上度

6. おわりに

スーパーコンピュータSX-4では、並列処理を中心とした運用を行っており、利用者も長時間ジョブで積極的に並列処理を利用している。SX-4は共有メモリ型のベクトル並列型スーパーコンピュータであり、自動並列化機能を有しているが、中には並列化効率が低くチューニングの必要なプログラムもある。そこで、センターでは高速化支援体制を設けて、利用者、センター、メーカが一体となりベクトル化率ならびに並列化率の向上を図ってきた。その代表的な例が、本稿で述べたプログラムの高速化であり、原版に対して、ベクトルチューニングで22倍、並列チューニングで347倍の性能向上が得られた。

SX-4の128台のCPUを最大限に使用するために、ベクトル化率ならびに並列化率の向上は不可欠である。利用者の一部は毎年入れ替わり、新しいプログラムも作成される。そのため、この高速化支援体制の継続は、必要なことである。それとともに、高速化に対する認識を、利用者に高めてもらうよう努力していきたいと考えている。

参考文献

[1]伊藤英一、熊谷紀子、花岡勝太郎、大泉健治、佐藤倫子、小野敏:
「スーパーコンピュータSX-4の使い方」東北大学大型計算機センター広報 Vol.30 1997-11

[2]伊藤英一、大泉健治、小野敏、高橋洋一、岡部公起:
「スーパーコンピュータSX-4/128H4のシステム構築について」全国共同利用大型計算機センター研究開発論文集No.20

```
        SUBROUTINE SUB1(…, NRH, ACHIR, ACHII)    ! 引数はグローバル領域
        REAL*8 ACHIR(NRH*NRH), ACHII(NRH*NRH)    ! サブルーチン内での
        REAL*8 SMCHR(NRH*NRH), SMCHI(NRH*NRH)    ! 配列はローカル領域
        REAL*8 RH(NRH*NRH), RHP(NRH*NRH)
        REAL*8 VGLPER(NRH*NRH), VGLPEI(NRH*NRH)
        integer ln_l(NLT*(LLT+1))
        integer ln_n(NLT*(LLT+1))
        save ln_l, ln_n, ln_max                  ! 配列を共有できるように save 文を利用
!cdir serial                                     ! ループ L, N を融合するための処理並列動作時
        ln_max=0                                 ! 複数のタスクで同時に実行しないように serial
        do L=0, LOMX                             ! セクションで区切る。ln_l, ln_n にループ融合した
            do N=1, NOEL(L)                      ! ときの情報(リスト)が入る。ln_l, ln_n をグローバル
                ln_max=ln_max+1                  ! 領域としたためすべてのタスクから参照できる
                ln_l(ln_max)=L
                ln_n(ln_max)=N
            enddo
        enddo
!cdir end serial
        do M=1, NRH*NRH                          ! ローカル配列の初期化のための処理
            SMCHR(M)=0.0                         ! SMCHR, SMCHI はローカルな配列
            SMCHI(M)=0.0                         ! このローカルな配列が, すべてのタスクで
        enddo                                    ! 初期化される
!cdir pardo                                      ! 並列化の指示
        do ln_loop=1, ln_max                     ! 融合されたループ L, N, ここで並列化される
            l=ln_l(ln_loop)                      !  ln_loop を指標にして各タスクに分割される
            n=ln_n(ln_loop)
            ……
            CALL GLPET(……)                       ! 並列化された各タスクから呼出される
            ……
            CALL GLNET(……)
            ……
            do M=1, NRH*NRH                      ! 並列化された各タスクでのローカルの総和
                SMCHR(M)=SMCHR(M)+(GLPET, GLNET の結果)
                SMCHI(M)=SMCHI(M)+(GLPET, GLNET の結果)
            enddo
            ……
        enddo
!cdir serial                                     ! グローバルの総和の準備(初期化)
        do M=1, NRH*NRH                          ! ACHIR,ACHII はグローバル領域
            ACHIR(M)=0.0                         ! 複数のタスクで同時に実行しないように
            ACHII(M)=0.0                         ! serial セクションで区切る
        enddo
!cdir end serial
!cdir critical                                   ! グローバルの総和計算, critical セクション
        do M=1, NRH*NRH                          ! により, タスク間で排他的に実行される
            ACHIR(M)=ACHIR(M)+(SMCHR(M) を利用)
            ACHII(M)=ACHII(M)+(SMCHI(M) を利用)
        enddo
!cdir end critical
        ……
    END
```

図 5.1　並列化版

平成 13 年 3 月発行

編集・発行　東北大学大型計算機センター
仙台市青葉区荒巻字青葉
〒980-8578