SR8000 の有効利用法

（株）日立製作所

1. はじめに

SR8000 には、複数の RISC マイクロプロセッサーが１つのノード内に複数搭載され、複数のノードが多次元クロスバーネットワークによって接続されています。

１つのノードは、複数のプロセッサー、システム制御部、主記憶から構成されています。

又、各プロセッサーでは、複数の演算パイプラインを保持し、複数命令の並列実行が可能なスーパースカラー方式を実現しています。

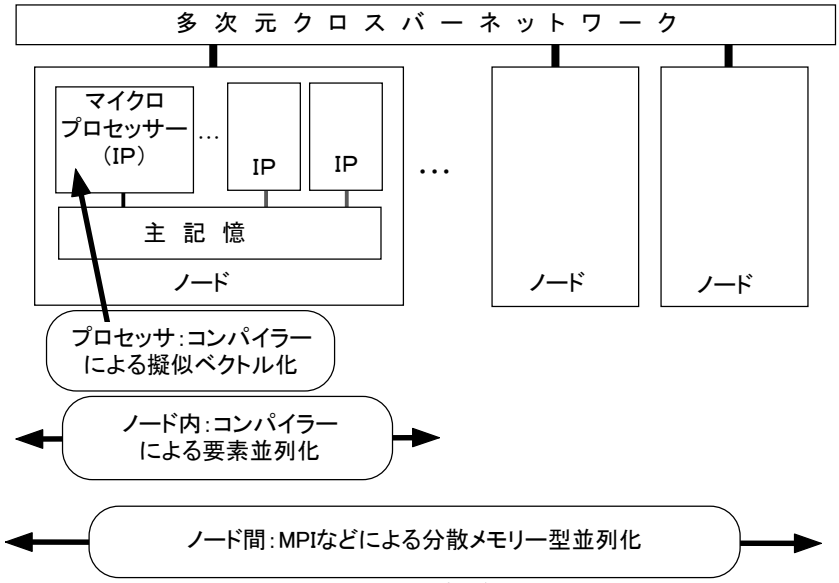


図 1.1　SR8000 概略図

ここでは、ノード間の分散メモリー型並列化の説明は省略し、すべてのユーザーに共通する以下の２点について解説します。

・プロセッサー内での擬似ベクトル処理機構の利用

・ノード内での複数プロセッサー使用による要素並列処理の利用

SR8000 は、擬似ベクトル処理によるメモリーレイテンシーの隠蔽・浮動小数演算性能の向上、及び並列処理が可能な部分での要素並列処理により、高速なプログラム実行が行える計算機です。

ここでは、コンパイルオプションにより擬似ベクトル処理・要素並列処理実行を実現し、得られる性能について示します。

1.1　擬似ベクトル化処理

擬似ベクトル化とは、各々のプロセッサーにおいて、主記憶上にあるデーターに対して擬似的なベクトル処理を行うことです。キャッシュまたはレジスターファイルを対象として、データーを主記憶から先読みすることで、データー転送オーバーヘッド（主記憶レイテンシー）が大きい主記憶上にデーターがある場合でも主記憶レイテンシーを隠蔽し、プログラムの高速処理が可能になります。

具体的なデーターの先読み方法には、以下の 2 種類があります。

・連続アクセスデーターに有効なキャッシュへの先読み

・非連続アクセスデーターに有効な浮動小数点レジスターへの先読み

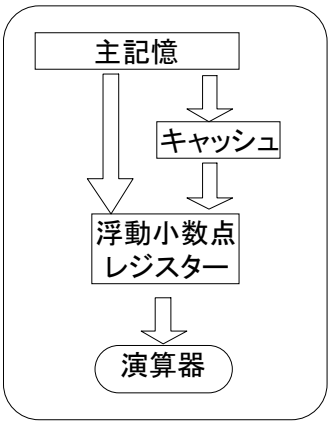


図 1.2　擬似ベクトル処理機構

キャッシュへの先読みとは、主記憶上の連続した領域（ライン）のデーターをあらかじめキャッシュに取り込むものです。DO ループ中のデーターアクセスが連続アドレスである場合に有効です。データーがキャッシュに先読みされている場合、より高速な処理が可能です。

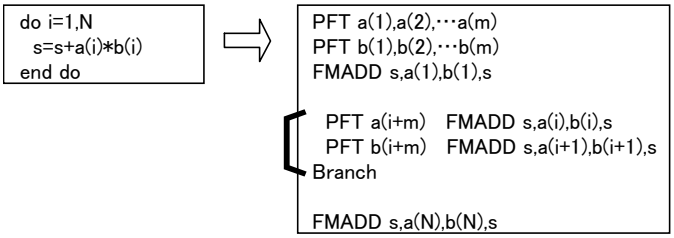


図 1.3　ソースプログラム例とオブジェクトコードの概略

図 1.3 にソースプログラムと、簡略化したオブジェクトコードの概念図を示します。ループ処理に先立って複数個のデーターをあらかじめ先読みしておき（図 1.3 の PFT 命令）、又ループ繰り返し部分でも、データーの先読みを行っています。コンパイラーは、ループ繰り返し部分の実行サイクル数を見積もり、主記憶レイテンシーを考慮して先読みしておくデーターの数（m）を決定します。

浮動小数点レジスターへの先読みとは、後続命令を止めることなく、主記憶上のデーターを浮動小数点レジスターに直接転送することです。SR8000 の各プロセッサーでは、浮動小数点データーを保持する拡張レジスターファイルを備えています。拡張レジスターファイル内のレジスターに対し、最大で拡張レジスター数分、データーの先読みを同時に実行することが可能です。レジスターへの先読みは、DO ループ中のデーターアクセスが非連続アドレスである場合に有効です。あらかじめ十分な数のデーターを取り込んでおくことで、データーの先読みと演算をパイプライン的に並列実行し、主記憶レイテンシーを隠蔽することが可能になります。この時、直接、浮動小数点レジスターに転送されたデーターはキャッシュには

格納されません。

1.2　要素並列化処理

要素並列化とは、1 つのノード内にある複数のプロセッサーが DO ループやスレッド（手続き呼び出しなどの文の集まり）を分担処理することです。分担処理の起動をハードウェアによるプロセッサー起動支援機構を用いて行うことにより、ベクトルプロセッサーの起動に近いオーバーヘッドで実現することが可能です。

要素並列処理を含むプログラム実行では、親プロセッサーで逐次処理部を実行し、親プロセッサー・子プロセッサーで分担処理部を実行します。図 1.4 に要素並列実行の流れを示します。

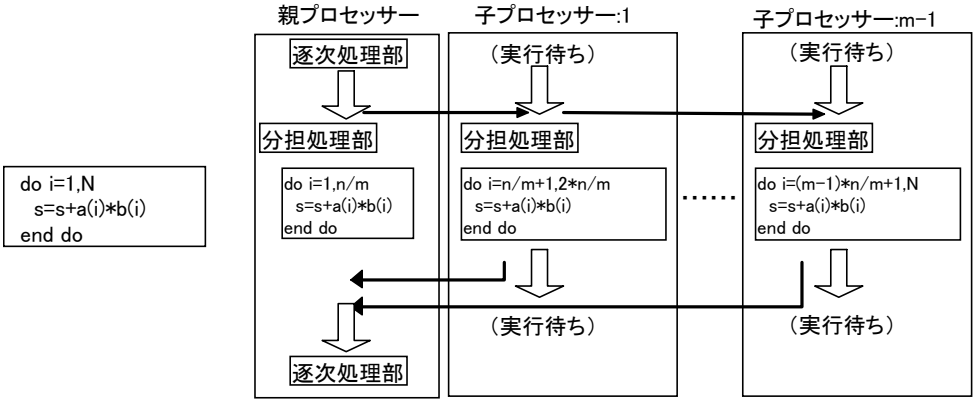


図 1.4　要素並列実行のイメージ

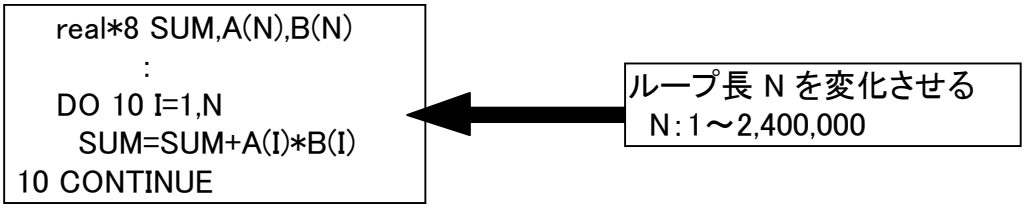
プログラムは、下記の手順により実行されます。

(1) 親プロセッサーは、分担処理部の以前にある逐次処理部を実行します。
　　子プロセッサーは、親プロセッサーによる起動を待ちます。
(2) 親プロセッサーは、ループ長等の分担処理部の実行に必要な情報を、プロセッサー起動支援機構を介して子プロセッサーに通知し、子プロセッサーを起動します。
(3) 子プロセッサー起動後、全プロセッサーは、分担処理部を要素並列実行します。
(4) 全プロセッサーが分担処理部の実行を終了すると、プロセッサー起動支援機構を介して親プロセッサーに分担処理部終了の通知がされます。
(5) 親プロセッサーは、逐次処理部の実行を開始します。
　　子プロセッサーは、再び、親プロセッサーによる起動を待ちます。

2.　コンパイルオプションによる性能の違い

本章では、内積計算プログラムにおいて、擬似ベクトル処理・要素並列処理の利用の有無による性能差について比較します。

設定配列　　：1 次元倍精度実数型配列　　2 個
配列の大きさ・ループ長　：1 ～ 2,400,000



尚、擬似ベクトル処理・要素並列処理の利用の有無は、コンパイル時のオプションで指定します。又、本測定の最適化レベルは S です。

2.1 擬似ベクトル化処理効果

はじめに、擬似ベクトル処理の利用の有無による性能差について比較します。

SR8000 に実装されているキャッシュの大きさは、1 プロセッサー当たり 128KB です。

図 2.1 の DO ループを要素並列化を行わずに 1 プロセッサーで実行する場合に、配列 A,B の全てのデーターがキャッシュに納まる上限の大きさを考えます。N は、配列の大きさ・ループ長を与えるパラメータとします。

$$2(\text{arrays}) \times 8(\text{byte}) \times N(\text{length}) < 128(\text{Kbyte})$$

$$\underline{N(\text{length}) < 8(\text{Kbyte})}$$

つまり、N が 8,000 より小さい時には、配列 A,B のデーターは、全てキャッシュ内に納まります。しかし、N が 8,000 を超えるとキャッシュあふれを発生します。

図 2.2 に擬似ベクトル処理の利用の有無による演算性能をループ長 N を変数として示します。

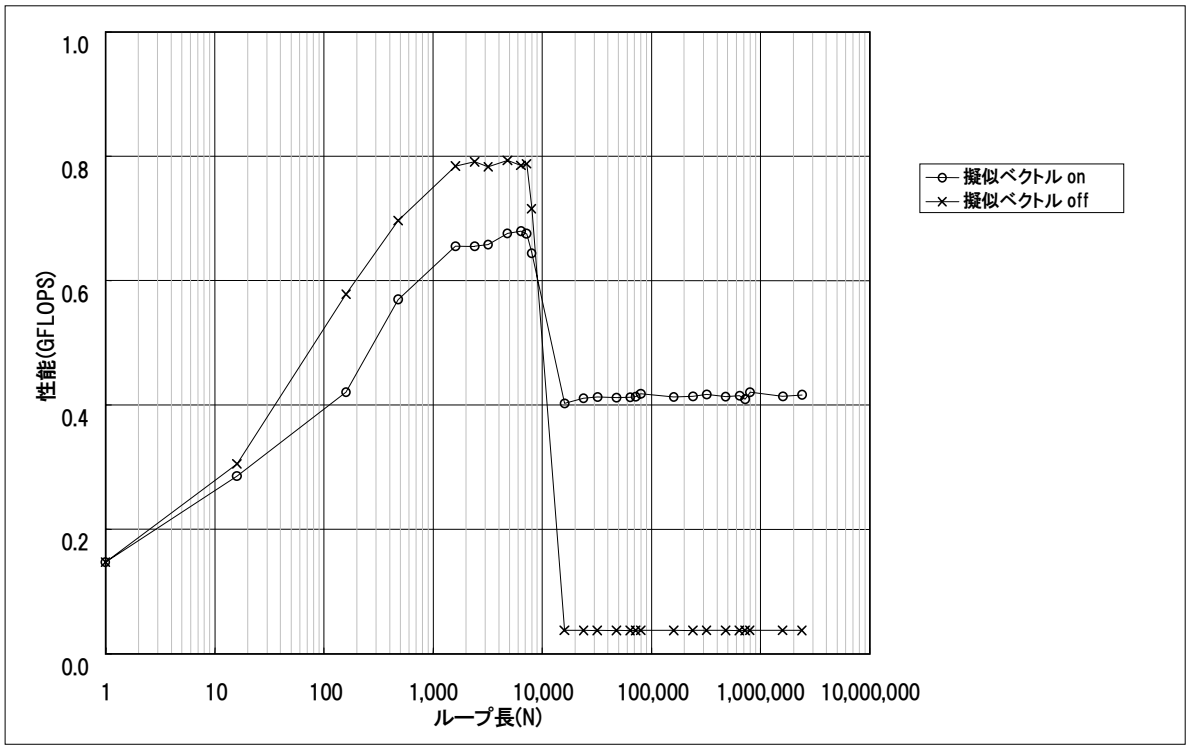


図 2.2　擬似ベクトル化処理効果

擬似ベクトル処理機構を利用しない場合、N=8,000 付近でピーク性能 0.8GFLOPS を示します。しかし、N が 8,000 を超え、キャッシュあふれを発生した時点で演算性能は著しく低下します。一方、擬似ベクトル処理機構を利用した場合、キャッシュあふれによる性能低下が少なく、ループ長が増加した時にも性能は維持されます。

以上のように、擬似ベクトル処理機構の利用により、キャッシュの容量に依存した性能の変動が少なく安定した性能を得ることが可能です。

2.2　要素並列化処理効果

次に、要素並列処理の利用の有無による性能差について比較します。

SR8000 の 1 ノードには、理論ピーク性能 1GFLOPS のプロセッサーが 8 つ搭載されています。

8 プロセッサーで実行する場合に、対象となる DO ループが完全に並列処理できるならば、1 プロセッサーで実行する場合と比較して 8 倍の演算性能が期待できます。又、8 プロセッサーで実行する場合には、使用可能なキャッシュの大きさも 8 倍になります。

$$2(\text{arrays}) \times 8(\text{byte}) \times N(\text{length}) < 128(\text{Kbyte}) \times 8(\text{processor})$$

$$\underline{N(\text{length}) < 64(\text{Kbyte})}$$

つまり、キャッシュあふれを発生する N の大きさも 8 倍であると考えられます。

図 2.3 に要素並列処理の利用の有無による演算性能をループ長 N を変数として示します。

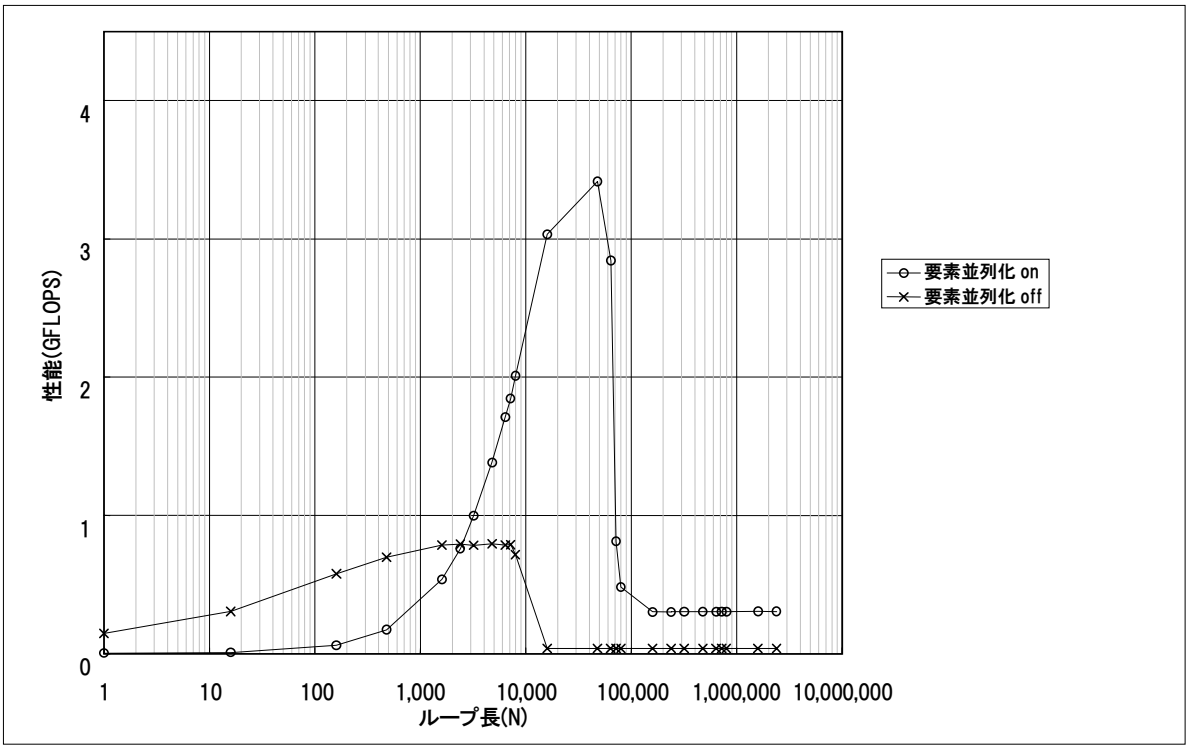


図 2.3　要素並列化処理効果

要素並列処理を利用する実行の場合には、ユーザー設定の変数・配列以外にも並列処理に必要な変数・配列を内部的に保持します。このためキャッシュあふれを発生するループ長が、理論的に考えられる N(length)　<　64(Kbyte) よりも小さくなります。

要素並列処理の利用により、ほぼ 8 倍のピーク性能 3.4GFLOPS を得られます。しかし、要素並列処理を利用した場合にもキャッシュあふれを発生した時点で演算性能は著しく低下します。

2.3　擬似ベクトル処理・要素並列処理の複合効果

図 2.4、図 2.5 に擬似ベクトル処理・要素並列処理の組み合わせにより得られる演算性能を示します。

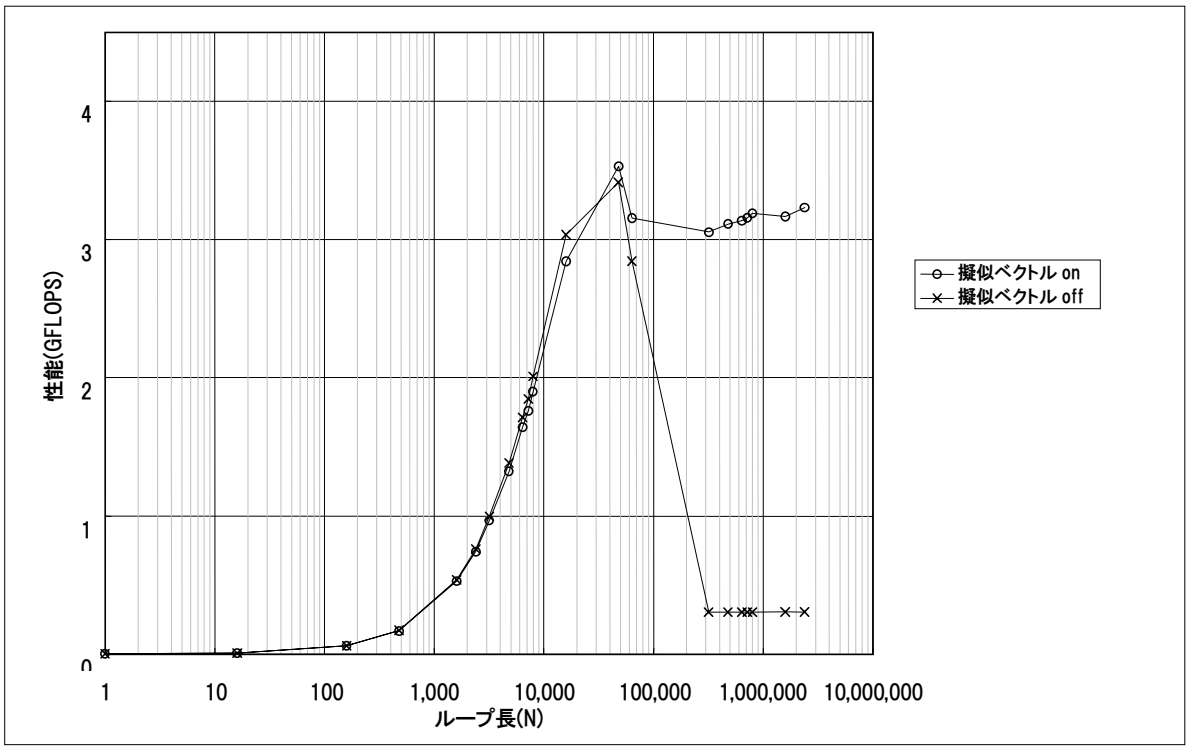


図 2.4　要素並列化処理適用時の擬似ベクトル化処理の有無の比較

図 2.4 には、要素並列処理を利用しますが、擬似ベクトル処理機構を強制的に適用した性能と、適用しない性能を比較しました。

要素並列処理を利用する実行の場合には、データーがキャッシュに納まっている範囲では、擬似ベクトル処理機構の有無で演算性能に大きな差はありません。しかし、擬似ベクトル処理機構を利用しない場合、キャッシュあふれが発生した時点で、著しく性能が低下します。

要素並列処理を利用する実行の場合にも擬似ベクトル処理機構の利用により、キャッシュ容量を意識せず安定した性能を得ることが可能です。

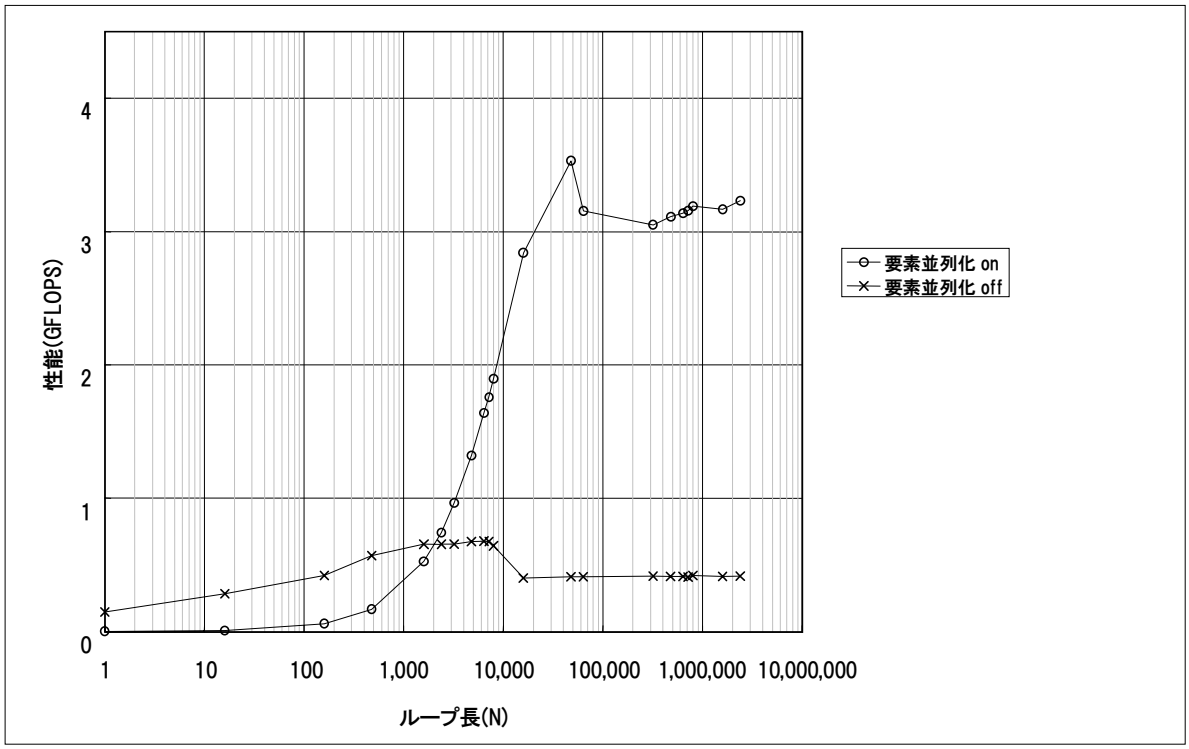


図 2.5　擬似ベクトル化処理適用時の要素並列化処理の有無の比較

図 2.5 には、擬似ベクトル化処理機構を適用しますが、要素並列化処理を強制的に適用した性能と、適用しない性能を比較しました。

擬似ベクトル処理機構を利用する場合、要素並列処理の利用の有無で、使用可能なキャッシュの大きさが異なるためキャッシュあふれを発生するループ長の大きさが異なります。又、要素並列処理の利用の有無に関わらず、安定した性能を得ることが可能です。

擬似ベクトル処理機構を利用する場合にも要素並列処理の利用により、キャッシュあふれ発生以降のループ長の増加についてもほぼ 8 倍の性能が得られます。

2.4　2 次元配列を用いた内積計算

2 次元配列を用いた行列の内積計算プログラムの擬似ベクトル処理・要素並列処理効果を比較します。

設定配列	：2 次元倍精度実数型配列　　3 個
最内側ループ長	：10 ～ 1,000
各配列の大きさ	：（最内側ループ長） - （最内側ループ長）

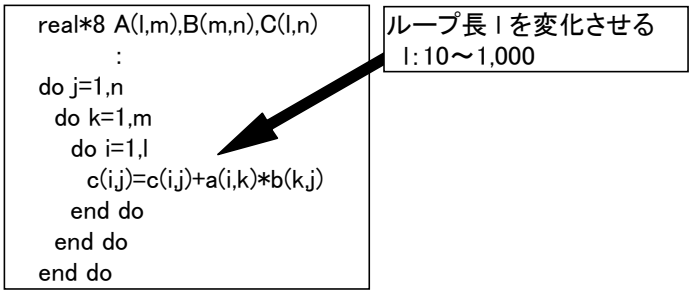


図 2.6　内積計算（2 次元配列）

コンパイルオプションにより、擬似ベクトル処理・要素並列処理の利用を指定します。又、本測定の最適化レベルは S です。

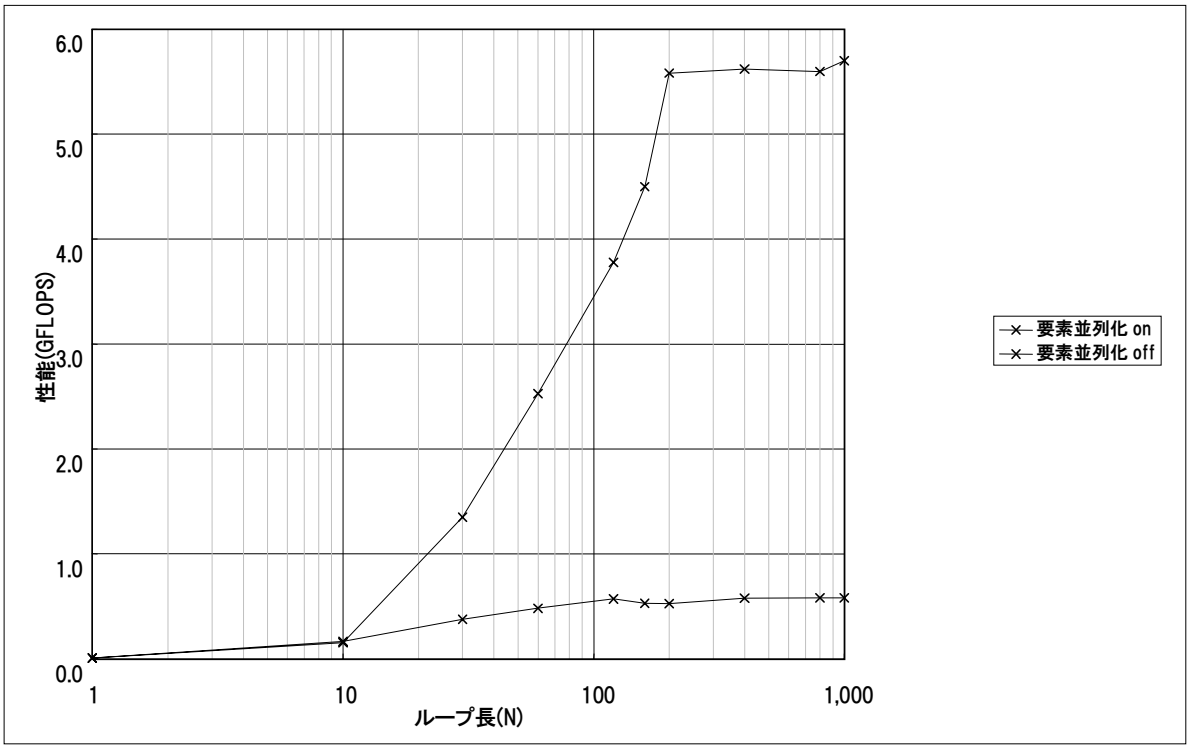


図 2.7　擬似ベクトル化処理機構適用時の要素並列化処理の有無の比較

図 2.7 には、擬似ベクトル化処理機構を適用しますが、要素並列化処理を強制的に適用した性能と、適用しない性能を比較しました。

1 次元配列を用いた場合と同様、ループ長が大きくなるにつれ、要素並列の利用の有無で性能差は大きくなり、本例では、ほぼ 8 倍の性能差が得られます。

3.　コンパイルオプション

基本的に最適化・擬似ベクトル化・要素並列化は、コンパイルオプションの付加によりコンパイラーが自動的に行います。

3.1　性能に関するコンパイルオプション

最適化・擬似ベクトル化・要素並列化を実現するために付加するオプションの例を以下に示します。

表 3.1　コンパイルオプション例

オプション	内　容		
	最適化	擬似ベクトル化	要素並列化
-04	4	off	off
-04 -pvec	4	on	off
-04 -parallel=3 -procnum=8	4	off	on 要素並列化レベル：3 並列実行 プロセッサー数：8
-04 -pvec -parallel=3 -procnum=8	4	on	on 要素並列化レベル：3 並列実行 プロセッサー数：8
-0s (-pvec -parallel=3)	S	on	on 要素並列化レベル：3 並列実行 プロセッサー数：N
-0s -nopvec -procnum=8	S	off	on 要素並列化レベル：3 並列実行 プロセッサー数：8
-0s -noparallel	S	on	off

最適化レベルSSでは、最高性能を得るために実行速度が最も早くなるような最適化を行っています。しかし、この副作用として演算精度が落ち、結果不正を起こす可能性があります。

3.2　-limit/-nolimit オプション

コンパイラーは、指定されたコンパイルオプションに従って、様々な最適化手法を試み、最適なオブジェクトを生成します。

コンパイラーは、より高度な最適化を行う場合、コンパイル時間、コンパイル時に使用するメモリー量にある程度の制約を設けています。制約の範囲内で最適なオブジェクトの生成が不可能な場合、最適化をやめてしまうわけではありませんが、やや最適化レベルの低いオブジェクトが生成される場合があります。

徹底的な最適化を行ったオブジェクトを生成したい時には、-nolimit オプションを指定

して下さい。但し、コンパイル時間は、非常に長くなります。

最適化･擬似ベクトル化･要素並列化によるプログラムの動作確認時には -limit オプションで、実行性能測定時には -nolimit オプションで、オブジェクトを生成することをお勧めします。

3.3 擬似ベクトル化・要素並列化診断メッセージ出力オプション

コンパイルオプションの指定により、コンパイラーが判断した擬似ベクトル化・要素並列化についての診断メッセージを出力することが出来ます。これにより、コンパイラーがどのように擬似ベクトル化・要素並列化したオブジェクトを生成したかを知ることが出来ます。

表 3.2 擬似ベクトル化・要素並列化診断メッセージ

診断メッセージの種類	オプション名	出力メッセージ
擬似ベクトル化診断メッセージ	-pvdiag or -WO,'PVEC(DIAG(1))'	KCHF1700 ～ 1706K KCHF1730 ～ 1739K
要素並列化診断メッセージ	-pardiag=1 or -WO,'MP(DIAG(1))'	KCHF2000 ～ 2014K
詳細な要素並列化診断メッセージ	-pardiag=2 or -WO,'MP(DIAG(2))'	KCHF2030 ～ 2112K

4. 基本的なチューニング

SR8000 は、要素並列化と擬似ベクトルという 2 つの特長を持っています。従って、SR8000 を用いて高い処理性能を得るためのチューニングは、この 2 つの特長を有効に活用することが基本となります。

要素並列化については、まず次の項目に着目し、問題があれば対策します。

- ･主要なループが要素並列化されているか？
- ･それが多重ループならば、外側で要素並列化されているか？
- ･要素並列化されているなら、プロセッサー間で負荷バランスが良いか？

擬似ベクトルについては、まず次の項目に着目し、問題があれば対策します。

- ･主要なループに対して、プリフェッチ、プリロードが適用されているか？
- ･配列データーへのアクセスは、できる限り連続になっているか？
- ･ループ内処理はソフトウェアパイプライン化されているか？

DO ループが要素並列化されているか、擬似ベクトル化されているかを判断するには、コンパイラーがソースプログラムと融合して出力する診断（ログ）メッセージが有益です。

ログメッセージは、コンパイル時にオプション -optlog を追加してコンパイルし、ソースファイル（ここでは foo.f とします）を引数にコマンド loggen を用いることで、foo.log にソースコードにログメッセージが埋め込まれたファイルを得ることができます（図 4.1)。

```
f90 （コンパイルオプション（以下の例題では -0ss 使用））-optlog foo.f
loggen foo.f
```

図 4.1　診断（ログ）メッセージ付きソースコードの作成方法

次の図 4.2 は、要素並列化可能なループの例と、ログツール（loggen コマンド）を使用したときのログメッセージを示しています。

ログメッセージはループ単位に、ループの直前に埋め込まれます。

図 4.2 において、最初のループに対するメッセージ「ループ入口で並列処理　開始」は、最初のループから要素並列処理を開始したこと、2 つ目のループに対するメッセージ「ループ出口で並列処理　終了」は、2 つ目のループ終了後、並列処理を終了するコードを生成したことを意味しています。

メッセージ「並列ループ」は該当ループが要素並列化されたことを示しています。

メッセージ「並列処理を継続」は次のループに対しても要素並列処理を終了させず継続していることを意味します。一般に異なる並列ループの間にはデーターの同期処理のため「ループ入口でバリア出力」というメッセージが出現しますが、本例ではループ間に同期処理が不要であるため「ループ入口のバリアを削除した」というメッセージになっています。

要素並列処理の起動単位は、並列手続き _parallel_func_1_xxx と名づけられた手続きとしてコンパイルされます。ここで、xxx はこのループを含む手続き名、数字 1 は起動単位に自動的につけられる一連の番号です。

```
** ループ入口で並列処理　開始
** 並列手続き名　：_parallel_func_1_xxx
** 並列ループ
**     最内側ループ展開（8 倍）を行った。
**     擬似ベクトル化を適用した。
      DO I=1,N
        A(I)=B(I)+C(I)
      END DO

** 並列処理を継続
** 並列ループ
** --- ループ入口のバリアを削除した ---
** ループ出口で並列処理　終了
**     最内側ループ展開（8 倍）を行った。
**     擬似ベクトル化を適用した。
      DO I=1,M
        X(I)=Y(I)+Z(I)
      END DO
```

図 4.2　ログメッセージの例

図 4.3 はデーター依存関係が原因で要素並列化できない場合の例です。メッセージ「逐次ループ」は本ループが要素並列化されず逐次実行されていることを意味します。次の行にあ

る「 A: ループ運搬依存がある」は要素並列化できない原因を示しています。この例では、配列 A に関してループイタレーション間にまたがるループ運搬依存（前のループイタレーションでの演算結果が必要など）があるため要素並列化できないことを示しています。

```
XX 逐次ループ
XX     A: ループ運搬依存がある
**     最内側ループ展開（16 倍）を行った。
**     擬似ベクトル化を適用した。
       DO I=1,N
         A(I+1)=A(I)+B(I)
       END DO
```

図 4.3　データー依存により要素並列化不可能なループのログメッセージ

以下の章では、ログメッセージを参考にしながら、SR8000 の基本的なチューニングポイントである要素並列化と擬似ベクトル化の有効的な活用方法について、例題を用いて説明します。

なお、以下の例において、ログメッセージは必要なもののみ抜粋した形で示します。

5.　要素並列処理

要素並列処理とは、ノード内の複数プロセッサーを協調動作させ、DO ループのループボディーを並列に処理することです。DO ループが要素並列化可能であるかどうかは、通常はコンパイラーが自動的に判断します。コンパイラーは、イタレーション間でデーター依存がないことが明らかな場合に限り、DO ループを自動要素並列化します。

以下の場合には、コンパイラーによる DO ループの自動要素並列化は行われません。

(1) イタレーション間でデーター依存がある場合
(2) コンパイラーがデーターの依存関係を把握できない場合
(3) ループボディーに手続呼び出しや関数呼び出しが含まれている場合

コンパイラーによる自動要素並列化が行なわれない場合でも、上記 (2)、(3) については、実際にはデーター依存がなく要素並列化可能なケースも存在します。このような場合、以下の方法により要素並列化を行うことができます。

・依存関係の明示（コンパイラーに要素並列化可能性を示唆する。）
・インライン展開（手続き呼び出しや関数呼び出しを要素並列化可能にする。）
・手続き呼び出しや関数呼び出しの別ループ化（それ以外のループボディーを要素並列化する。）
・強制要素並列化（要素並列化箇所を明示する。）

5.1　ループ内に手続き呼び出しが含まれている場合の要素並列化

次の図 5.1 に示すループは、手続き呼び出しを含むため、ログメッセージ「ループ内に並列化対象外の文がある」が示すように自動的には要素並列化されません。

```
XX 逐次ループ
XX  --- ループ内に並列化対象外の文がある  --
      DO I=1,N
        CALL INDEX(I,IM,IP,N)
        A(I)=B(IM)+B(I)+B(IP)
      END DO
```

```
      SUBROUTINE INDEX(K, KM, KP, N)
      common /parm/model
      KM = K-1
      KP = K+1
      if(model.eq.1)then
        if( K .eq. 1) KM=N
        if( K .eq. N) KP=1
      end if
      return
      end
```

図 5.1　手続き呼び出しを含むループ

しかし、図 5.1 の例における手続きの処理内容を見れば、要素間に依存はなく要素並列処理が可能であることがわかります。このような場合、手続きをインライン展開する、または、ループに強制要素並列化ディレクティブ *poption parallel を指示する（図 5.2）ことで要素並列化ができます。

手続き呼び出しを含むループを強制要素並列化する場合、さらに、次の指示が必要です。

手続きを呼び出すループには、強制要素並列化ディレクティブ *poption parallel の他に、手続間を跨って参照される変数に対して、並列化の仕方の指示をします。図 5.1 の例では、変数 IP と IM は手続間を跨って参照されるので、各要素プロセッサーがこれらの変数を固有の領域に割り付ける（ローカル化する）必要があります。この指示には、ディレクティブ *poption tlocal を使います。

さらに、呼び出される手続き側にも指示が必要です。すでに手続きを呼び出すループが並列化されているので、この手続きは並列化できません。従って、図 5.2 に示すように、呼ばれる手続きは要素並列化をしないオプション MP(P(0))、及び手続きのローカル変数の状態に関してコンパイルオプション LANGLVL(SAVE(0))（前回の呼び出しでの値を保持しない）を指定する必要があります。

```
*poption parallel,tlocal(IM,IP)
** 並列ループ
**    IM : TLOCAL 変数
**    IP : TLOCAL 変数

      DO I=1,N
        CALL INDEX(I,IM,IP,N)
        A(I)=B(IM)+B(I)+B(IP)
      END DO
```

```
*poption MP(P(0)) LANGLVL(save(0))          呼ばれる手続きに必要な
      SUBROUTINE INDEX(K, KM, KP, N)          コンパイルオプション
      common /parm/model
      KM = K-1
      KP = K+1
      if(model.eq.1)then
        if( K .eq. 1) KM=N
        if( K .eq. N) KP=1
      end if
      return
      end
```

図 5.2　手続き呼び出しを含むループの強制並列化

手続き呼び出しを含むループを *poption parallel により強制的に並列化した場合、正しい並列化のためにローカル化が必要な変数が、実際にローカル化されていることをログメッセージで確かめて下さい。ただし変数の中でループの制御変数のようにループ 1 回実行ごとに等差数列的に増える変数に対しては特にメッセージは出力されません。

5.2　不明依存がある場合の要素並列化

次の図 5.3 に示すループは 2 重ループですが、ログメッセージを見ると、「A：ループ内に不明依存がある」とあり、外側ループは要素並列化せず、内側のループに対して要素並列化を行っています。実際、変数 J1 と M1 の関係によっては、外側ループの要素並列化により誤演算を引き起こす場合があるからです。

要素並列化は、できる限り大きな単位で行い、並列化の起動・終結に要するオーバーヘッドを少なくすることが、性能向上につながります。図 5.3 のような場合、外側ループで要素並列化が本当に出来ないのか、調べてみることが重要です。

```
XX 逐次ループ
XX    A  :  ループ内に不明依存がある
      DO J=1,M1
** 並列ループ
       DO I=1,N
        A(I, J)=B(I, J)
        A(I,J1+J)=B(I,J1+J)
       END DO
      END DO
```

図 5.3　コンパイラーがデーター依存を解決できない（外側ループ）

図 5.3 の例では、J1=0 または J1 > M1 であることが（ユーザーから見て）明らかな場合、配列 A への A(I,J) と A(I,J1+J) のストアはループイタレーション間で依存がないので、ディレクティブ *poption indep により依存関係を明示することで外側ループの要素並列化が可能になります（図 5.4）。

```
*poption indep(A)
**  並列ループ
**     I : TLOCAL 変数
      DO J=1,M1
       DO I=1,N
        A(I, J)=B(I, J)
        A(I,J1+J)=B(I,J1+J)
       END DO
      END DO
```

図 5.4　依存関係の明示による要素並列化

5.3　要素並列化箇所の指定

一般的に多重ループは、外側で要素並列化した方が効率は良くなりますが、そうでない場合もあります。

```
XX 逐次ループ
      DO J=1,10
** 並列ループ
        DO I=1,1000
          A(I,J)=A(I,J)+B(I,J)
        END DO
      END DO
```

図 5.5　外側ループが短い 2 重ループ

図 5.5 の 2 重ループは外側ループ長が 10 と短いので、これを 8 プロセッサーで要素並列化するより、内側ループを分割した方が効率は良くなります。コンパイラーも自動的に判断して外側ループの要素並列化を抑止していることがログメッセージから分かります。

ただし、実際のプログラムにおいて、このように配列のサイズやループ長がコンパイル時に明らかな場合はほとんどありません。上記の例で、1000 を変数 N で、10 を変数 M で置き換えると、コンパイラーは当然外側ループで要素並列化します。このような場合は、内側ループで要素並列化するように、外側ループの要素並列化抑止のディレクティブ *poption noparallel を指示します（図 5.6)。

```
*poption noparallel
XX 逐次ループ
      DO J=1,M
** 並列ループ
        DO I=1,N
          A(I,J)=A(I,J)+B(I,J)
        END DO
      END DO
```

図 5.6　外側ループの要素並列化抑止（内側ループ強制要素並列化）

5.4　ソースコード変更による要素並列化

次の図 5.7 の例は、図 5.8 に示す 2 重ループを 1 重化したコードです。前半の 2 重ループでリストを作成し、処理 A(M)=A(M-1)+B(M) を 1 重ループで行っています。

コンパイラーは配列 A の依存性が解らないため、この 1 重ループは自動要素並列化できません。ただし、ログメッセージにもあるようにループ分配を行って（ループを 2 つに分割して）M=mlist(L) の部分だけ要素並列化していますが、部分的な並列化はそれほど高速になるものではありません。

```
      Real*8 A(IMX*JMX),B(IMX*JMX)
      Integer mlist(IMX*JMX)

      L=0
      DO J=1,JMX-1
        DO I=2,IMX-1
          L=L+1
          mlist(L)=I+IMX*(J-1)
        END DO
      END DO
      LMX=L

** 並列ループ
**    --- 並列化向けループ分配を行った ---
XX 逐次ループ
XX    A: ループ内に不明依存がある。
      DO L=1,LMX
        M=mlist(L)
        A(M)=A(M-1)+B(M)
      END DO
```

図 5.7　1 重化により自動要素並列化不可になったループ

この例の場合は、図 5.8 に示すオリジナルの 2 重ループに戻せば、外側ループで要素並列化可能になります。

```
      Real*8 A(IMX,JMX),B(IMX,JMX)

** 並列ループ
**    I: TLOCAL 変数

      DO J=1,JMX-1
        DO I=2,IMX-1
          A(I,J)=A(I-1,J)+B(I,J)
        END DO
      END DO
```

図 5.8　要素並列化可能なオリジナルループ

ベクトル機向けのコードでは、多少余計な処理が増えてもループ長を長くした方が効率がよいことから、ループの 1 重化はよく見られる処理です。ただし、SR8000 は外側ループを要素並列化できますから、むしろ余計な処理のないオリジナルのコードの方が、一般的に効率がよくなります。

6. 擬似ベクトル化

擬似ベクトル処理とは、プリフェッチ命令（キャッシュへの先読み）およびプリロード命令（浮動小数点レジスターへの先読み）を用い、主記憶アクセスレイテンシーを隠蔽する命令スケジューリングを行うことです。プリフェッチは、キャッシュ 1 ライン（128 バイト）分の連続データーを、主記憶からプロセッサーの持つキャッシュに転送する命令です。一方、プリロードは、8 バイトまたは 16 バイトのデーターを、主記憶からレジスターに転送する命令です。

最内側ループで添字が変化するアクセス配列に対して、コンパイラーは通常、

(1) 連続アクセスに対してはプリフェッチの適用を、
(2) ストライドアクセスでアドレスの飛び幅が小さいときはプリフェッチの適用を、大きいときはプリロードの適用を、
(3) 離散的データーとなる可能性のあるアクセスに対してはプリロードの適用を

行います。

図 6.1 の例では、配列 xx(i) が連続アクセス、配列 yy(i),ind(1,i),ind(2,i),ind(3,i) がストライドアクセス、配列 zz(ii,jj,kk) が離散的アクセスになります。

コンパイラーは、同一ループイタレーションでの配列間のアクセス連続性についても検出を試み（ind(1,i),ind(2,i),ind(3,i)）のような連続データー）、プリフェッチが有利と判断すればプリフェッチを適用します。

図 6.1 のログメッセージで「擬似ベクトル化を適用した。」とはループ内で配列を主記憶から読み出す際に主記憶レイテンシーの隠蔽コードを生成することに成功したということを意味しています。

```
** 並列ループ
**
**    擬似ベクトル化を適用した。
**
      DO i=1,m
         ii=ind(1,i)
         jj=ind(2,i)
         kk=ind(3,i)
         ww(i)=xx(i)+yy(n,i)+zz(ii,jj,kk)
      END DO
```

図 6.1　メモリー上のデーターのアクセス方式

6.1　連続アクセスへの変換

プリロードは個々のデーターをレジスターに転送しますが、プリフェッチは連続データー

を一括してキャッシュに転送しますので、主記憶データーの転送効率が高くなります。そのため、配列参照はできるだけ主記憶の連続方向に行い、その配列に対してプリフェッチを適用することが望ましいと言えます。連続的なストリームを多くする手段として、ループ交換・抑止の適用があります。

```
**  擬似ベクトル化を適用した。

      DO 30 I=1,N

**  並列ループ
** --- 外側ループ分配を行った。---
** --- ループ交換を行った  ((I,J)->(J,I))---

        DO 10 J=1,M
          A(I,J)=B(I,J)+C(I,J)
   10   continue
** (略)
        DO 20 J=2,M-1
          D(J,I)=E(J,I)*F(J,I)
   20   continue
   30 continue
```

図 6.2　内側ループの 1 つがストライドアクセスになっている 2 重ループ

この例では、内側ループが 2 つ（DO 10,DO 20）ありますが、最初の DO 10 ループの配列アクセスがストライドアクセスになっています。このように判断が簡単な場合で最適化オプションのレベルが高い（-0s,-0ss）場合、ログメッセージが示すように、コンパイラーは自動的に外側ループに関して分配（ループを途中で複数のループに分割）を行い、最初のループに関してはループ交換（内外のループの入れ替え）を行います。そして DO30 に関して擬似ベクトル化を行います。結果的に図6.3のようなソースプログラムを書いた場合と同一で、最内側ループはすべて連続アクセスとなり効率が良くなります。

```
      DO 10 J=1,M
        DO 30 I=1,N
          A(I,J)=B(I,J)+C(I,J)
   30   continue
   10 continue
      DO 300 I=1,N
        DO 20 J=2,M-1
          D(J,I)=E(J,I)*F(J,I)
   20   continue
  300 continue
```

図 6.3　ループ交換により内側ループを連続アクセスにした例

図 6.2 の例の場合、コンパイラーはループ交換をしてもプログラムの意味が変わらないことがわかり、交換することによって性能上のメリットがあると判定しました。しかし、一般

的にはコンパイラーが、交換しても有利とは判断できない（例えばループ内に連続アクセスと非連続アクセスが混在する）場合や、依存関係が解析できずループ交換可能かどうか判断できない場合があります。その時は、ユーザーの判断でソースプログラムを図 6.3 のように書きかえることで性能向上が見込めます。

また、図 6.2 の例では、コンパイラーが自動的に判断してループ交換を行っていますが、外側ループ長が小さい場合は、5.3 で述べたように負荷分散の観点から望ましくないので、ループ交換を抑止するディレクティブ *soption noloopinterchange をループに挿入する必要があります。

6.2　ソフトウェアパイプライン

ソフトウェアパイプラインとは、異なるループのイタレーション（i = i1 と i = i1 + 1）での処理を並列に動作させることで、深いパイプラインを有する演算器のレイテンシーを隠蔽し、命令パイプラインを効率よく動作させるスケジューリング手法です。

コンパイラーは、データーの依存関係を調べた上で、プロセッサーの持つ演算器やレジスターなどの計算資源と各種命令の演算レイテンシーを考慮して命令スケジューリングをおこない、ソフトウェアパイプラインコードを生成します。ただし、以下の場合にはソフトウェアパイプラインは適用されません。

(1) コンパイラーがデーターの依存関係を解析できない場合
　（結果的にこの場合はソフトウェアパイプライン化されても効率的な並列度の高いコードにはなりません（ i = i1 と i = i1 + 1 の命令がオーバーラップされにくい)。）
(2) ループに手続き呼び出しや関数呼び出しが含まれている場合
(3) ループに IF 文が含まれている場合
(4) レジスターが不足した場合

上記 (1) の場合、要素並列化の場合と同様な方法でソフトウェアパイプラインの適用を期待できます。

・依存関係の明示
・依存関係の解消

また、上記 (2) の場合も同様に、以下を適用して、効果が期待できます。

・インライン展開
・数学関数に対する擬似ベクトル化関数の適用
・手続き呼び出しや関数呼び出しの別ループ化

上記 (3) については 6.5、(4) については 6.6 で説明します。

なおログメッセージには、主記憶レイテンシー隠蔽のための擬似ベクトル処理とソフトウェアパイプラインが適用された場合「擬似ベクトル化を適用した」と表示されます。主記憶レイテンシーの隠蔽処理は行われるがソフトウェアパイプラインが適用されない場合は「ソフトウェアパイプライニングを適用しなかった」というメッセージが付加されます。

6.3　依存関係の解消によるソフトウェアパイプラインの適用

次の例のログメッセージを見ると、擬似ベクトル化されソフトウェアパイプラインコード

が生成されているので、性能的には問題がないように思ってしまうかもしれません。

```
      Real*8 A(M),B(M)
      N=M/2
XX  逐次ループ
**    A: ループ内に不明依存がある。
**    B: ループ内に不明依存がある。
**
**    擬似ベクトル化を適用した。

      DO I=1,M/2
        A(I)=A(I+N)+B(I+N)
        B(I)=A(I+N)-B(I+N)
      END DO
```

図 6.4 コンパイラーがデーターの依存関係を把握できない場合

しかしログメッセージの「ループ内に不明依存がある」は、実際は並列度の低いコードとなっていることを示しています。何故なら上記の例では、配列 A と B の依存関係が不明であるため、ロード（A(I+N),B(I+N)）、ストア（A(I)）、ロード（A(I+N),B(I+N)）、ストア（B(I)）の順番がまったく換えられず、ループイタレーション間を跨いでも配置できないからです。

実際には図 6.5 に示すように、更新範囲と参照範囲に重なりはありません。

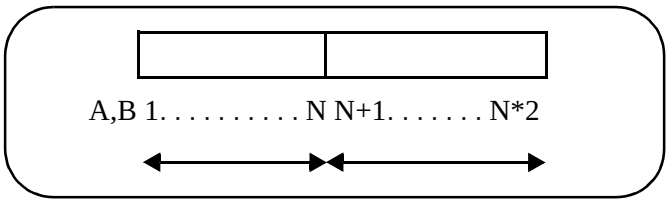


図 6.5　配列 A,B の更新および参照範囲

このことをコンパイラーにディレクティブ *voption indep で指示すると、コードスケジューリングの自由度（上記のロードストア命令の順序関係）が広がり、大きく性能向上します。

ソフトウェアパイプラインのコードスケジューリングはトライアンドエラー処理を成功するまで繰り返すためコンパイル時間がかかる処理です。データーの依存関係が不明で順序関係に拘束がある場合、スケジューリングが難しく、失敗することも多いので、コンパイル時間を消費する割には実行速度は向上しないことがあります。その意味でも、（ユーザーによる）依存関係の明示は重要です。

図 6.6 に依存関係を明示した場合のログメッセージを示します。

```
      Real*8 A(M),B(M)
      N=M/2
*voption indep(A,B)

** 並列ループ
** 擬似ベクトル化を適用した。

      DO I=1,M/2
        A(I)=A(I+N)+B(I+N)
        B(I)=A(I+N)-B(I+N)
      END DO
```

図 6.6　*voption indep ディレクティブにより依存関係を明示した例

6.4　ループ内に数学関数が含まれている場合

ループ中に数学関数が含まれている場合、通常のコンパイラーオプションでは数学関数ライブラリーの呼び出しとなるためソフトウェアパイプラインが適用されません。

```
XX　擬似ベクトル化を適用した。ただしループ中に手続き呼び出しがあるためソフトウェアパイプライニングを適用しなかった。

      DO I=1,N
        A(I)=EXP(B(I)+C(I))
        D(I)=A(I)*D(I)
      END DO
```

図 6.7 ループ内に数学関数が含まれている例

この場合、数学関数部分をベクトルライブラリーとして切り出し、全体をソフトウェアパイプライン化するために、以下のコンパイルオプションまたはディレクティブを設定します。

・コンパイルオプション　　-pvfunc=2　　または

・ディレクティブ　　*voption pvfunc(2)

上記のコンパイルオプションは、デフォルトでは設定されていませんが、ほとんど常に有効であると考えられますので、常に設定するのがよいでしょう。

例えばコンパイルオプション -pvfunc=2 を指定した場合を図 6.7 に示します。

```
**　ベクトルライブラリ化を行った（_hf_pvdexp)。
**　ベクトルライブラリ向けループ分配（3 分割）を行った。
**　擬似ベクトル化を適用した。

      DO i=1,N
        A(i)=EXP(B(i)+C(i))
        D(i)=A(i)*D(i)
      END DO
```

図 6.8 コンパイラーオプション（-pvfunc=2）指定した場合

上記のコンパイラーオプションまたはディレクティブが指定されると、コンパイラーは次のようにループを処理します。

・学関数が含まれる部分と、それ以外の部分にループを分配
（上記のログメッセージで「ループ分配（3 分割)」とある部分）
・数学関数部分にベクトルライブラリー（擬似ベクトル化数学関数）を適用
（上記のログメッセージにおける、_hf_pvdexp は倍精度擬似ベクトル指数関数）
・それ以外の部分にソフトウェアパイラインを適用

図 6.8 のループは、次の図 6.9 のような処理に変換されます。

```
DO i = 1,N
  Work(i)=B(i)+C(i)                ループ①
END DO

Call _hf_pvdexp(Work,A,N)          倍精度擬似ベクトル指数関数

DO i = 1,N
  D(i)=A(i)*D(i)                   ループ②
END DO
```

図 6.9　ループに擬似ベクトル化数学関数を適用するためのコンパイラー内部の変換

図 6.9 のループ①と②にソフトウェアパイプラインが適用され、数学関数に対して擬似ベクトル化された数学関数が適用されます。

なおオプション -pvfunc=1（ディレクティブ　*voption pvfunc(1)）は上記のようなプログラム上にない新たな配列（Work）を導入しない範囲で擬似ベクトル化数学関数を適用するだけで、適用範囲が狭いため、ここではお勧めしません。

現在、擬似ベクトル化数学関数のレパートリーは、以下の倍精度実数関数が提供されています。

除算・べき乗・平方根（SQRT)・指数（EXP)・自然対数（LOG)・常用対数（LOG10)・正弦（SIN)・余弦（COS)・正接（TAN)・逆正弦（ASIN)・余弦（ACOS)・正接（ATAN,ATAN2)・双曲線正弦（SINH)・双曲線余弦（COSH)

図 6.10　擬似ベクトル化数学関数のレパートリー

6.5　IF 文を含むループ

通常の RISC 計算機では、IF 文は条件判定とそれに基づく分岐命令の処理に分解されます。分岐命令はハードウェアによる予測が失敗すると 5 サイクル以上のオーバーヘッドが発生します。さらに、一般にソフトウェアパイプラインコードが生成できません。

IF 文を含むループにソフトウェアパイプラインを適用する方法の一つに、プレディケート最適化の適用があります。

プレディケート最適化（IF 変換）とはプロセッサーの提供するプレディケート機能を利用し、IF の成否にかかわらず IF ブロックおよび ELSE ブロック内の演算をおこない、演算完

了後に IF の成否に応じて演算結果を採択する処理です。

プレディケート最適化をおこなうと、ソフトウェアパイプラインによる高性能化が望める反面、最終的に採択されない演算結果を求めるための（結果として無効な）演算をおこなうことになります。そのため、IF ブロックや ELSE ブロック内の演算量と、IF の成否の確率によっては、この無効な演算のためにかえって性能が低下する場合があります。具体的には、採択確率の低いブロックのサイズが大きかったり、あるいは除算を含んだりする場合には、プレディケート最適化により性能が低下する可能性があります。

明らかにプレディケート最適化が有効になるときは if-then（-else）のブロック内にある演算が少ないときです。例えば MAX 等の条件判定処理を含む小さな関数処理です。

また、逆に性能が低下する可能性があるのは、実行する確率の小さいパスに多くの演算や除算のような処理があるときです。

コンパイルオプション -Oss では、プレディケート最適化適用が仮定されており、それ以外ではプレディケート最適化の抑止が仮定されています。ループの特徴から、明らかにプレディケートを適用（または抑止）した方がよい場合は、以下のオプション、またはディレクティブを指定して下さい。

・プレディケート最適化適用

コンパイルオプション：　-predicate

ディレクティブ：　*soption predicate

・プレディケート最適化抑止

コンパイルオプション　-nopredicate

ディレクティブ：　*soption nopredicate

次の図 6.11 および図 6.12 にプレディケート最適化を適用・抑止した場合のログメッセージを示します。

```
      Integer(kind=4),parameter :: N=100
      Real*8 A(N),B(N),C(N)

XX    擬似ベクトル化を適用した。ただし，ループが条件分岐を含むためソフトウェアパイプラインニングを適用しなかった。

*soption nopredicate

      DO i = 1,N
        If(A(i) > 0.d0) then
          A(i) = B(i)*C(i) + D(i)*E(i) + F(i)
        Endif
      END DO
```

図 6.11　if 文を含むループ（成立確率が低い場合）

```
      Integer(kind=4),parameter :: N=100
      Real*8 A(N),B(N),C(N)

**    IF 変換を適用した。
**    擬似ベクトル化を適用した。

*soption predicate

      DO i = 1,N
        if(A(i) > 0.d0) then
          A(i) = B(i)*C(i) + D(i)*E(i) + F(i)
        Endif
      END DO
```

図 6.12　if 文を含むループ（成立確率が高い場合）

プレディケート最適化の適用・抑止の他に、IF 文の then(-else) 側に除算や数学関数があるときは、IF 文のリスト化により性能が向上する場合があります。

6.6　レジスター不足の解消

擬似ベクトル処理を行うには多くのレジスターを必要とします。特にアクセスする配列数が多い場合にその傾向は顕著です。必要なレジスター数が利用可能なレジスター数を越えると、コンパイラーはソフトウェアパイプラインの適用をあきらめたり、あるいはレジスタースピルを発生させたりするので、性能低下要因となります。

レジスタースピルとは、次のいずれかの処理によりレジスター不足を補う手法です。

(1) ループ内で値が変わらないデーターをスタックに格納しておき、利用時にロードする。
(2) ループ内で値が変わるデーターをいったんスタックに退避し、再利用時にレジスターに回復する。

(1) の場合、ログメッセージは「レジスタが不足したのでループ内にループ不変変数に対して LOAD 命令を生成した」と表示します。(2) の場合は「レジスタが不足したのでループ内に LOAD/STORE 命令を生成した」と表示します。

一般に (1) の場合はそれほど気にする必要はありませんが、(2) の場合はレジスター不足を抑える工夫をしたほうが良いことがあります。

レジスター不足を抑えるには、以下の手法を適用します。

- ループ展開数の低減 （ループボディーを小さくする。）
- ループ分割 （ループボディーを小さくする。）
- ベクトル向け一重化ループの多重ループへの巻き戻し （ループボディーを小さくする。）
- 整数配列の実数化 （汎用レジスターが不足しやすいため、整数配列は使わない。）
- 単精度実数の倍精度化 （単精度実数演算では拡張レジスターを利用できないため、レジスター不足になりやすい。拡張レジスターを利用可能な倍精度実数を用いる。）

次の図 6.13 は、汎用レジスターが不足したため、ソフトウェアパイプラインが適用されなかったループの例です。

```
      Integer(kind=4),parameter :: IMAX=100,JMAX=100
      Real(kind=8),dimension(IMAX,JMAX) ::
     $    A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,
     $    B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,
     $    C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,
     $    D0,D1,D2,D3,D4,D5,D6,D7,D8,D9

** 並列ループ
**   I : TLOCAL 変数

      DO J=1,JMAX

XX  擬似ベクトル化を適用した。ただし，汎用レジスタが不足したためソフトウェア
パイプライニングを適用しなかった。
XX  レジスタが不足したのでループ内にループ不変変数に LOAD を生成した。

        DO I=1,IMAX
          A0(I,J)=B0(I,J)+C0(I,J)*D0(I,J)
          A1(I,J)=B1(I,J)+C1(I,J)*D1(I,J)
          .............
          A9(I,J)=B9(I,J)+C9(I,J)*D9(I,J)
        END DO
      END DO
```

図 6.13　多数のメモリー参照を含むループ

この例は、メモリー参照が多いループです。配列のベースレジスターを保持する汎用レジスターが不足となりソフトウェアパイプラインが適用できていません。この場合、汎用レジスターがスピルしないように、ループを分割します。例えば図 6.14 に示すように、2 つにループを分割するとレジスター不足にはならないことがわかります。

この例のように、分割のために新たなワーク配列を必要としない場合は、一般に性能向上を達成できることは確実です。しかしループ分割のためにワーク配列が必要となる場合は、ワーク配列への余分なメモリー参照とスピルによるスケジューリングオーバーヘッドとのトレードオフで性能が決まるため、どのような場合にも効果があるわけではありません。

```
      Integer(kind=4),parameter :: IMAX=100,JMAX=100
      Real(kind=8),dimension(IMAX,JMAX) ::
     $    A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,
     $    B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,
     $    C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,
     $    D0,D1,D2,D3,D4,D5,D6,D7,D8,D9

** 並列ループ
**   I : TLOCAL 変数

      DO J=1,JMAX

**       擬似ベクトル化を適用した。
         DO I=1,IMAX
           A0(I,J)=B0(I,J)+C0(I,J)*D0(I,J)
           .......
           A4(I,J)=B4(I,J)+C4(I,J)*D4(I,J)
         END DO

** 並列処理を継続
**       擬似ベクトル化を適用した。

         DO I=1,IMAX
           A5(I,J)=B5(I,J)+C5(I,J)*D5(I,J)
           .......
           A9(I,J)=B9(I,J)+C9(I,J)*D9(I,J)
         END DO

      END DO
```

図 6.14　スピルを抑止するためにループ分割した例

7. まとめ

SR8000 を効率的に用いることは、要素並列化処理と擬似ベクトル化処理を有効に活用することです。この２つの機能を利用するかしないかは、コンパイル時のオプションで指定します。目的に応じて、最適なオプションを指定して下さい。

要素並列化処理は、できるだけ大きな単位で行い、かつ、各プロセッサーの負荷がバランスすることが重要です。擬似ベクトル化について重要なことは、ソフトウェアパイプラインが効率良く適用されているかどうかです。ユーザーから見ると自明なことが、コンパイラーには分からず、最適化の効率が低い場合があります。主要な処理については、効率的な要素並列化、擬似ベクトル化をコンパイラーが行っているかログメッセージを参考にチェックし、必要ならば指示を与えてください。

以上