HITACHI SR16000/M1 チューニング連載講座

２．単体ノード性能チューニング

片桐　孝洋

東京大学情報基盤センター　准教授

１．はじめに

本稿では、HITACHI SR16000/M1（以降、SR16K と記載）で特徴的な単体性能チューニング技法、すなわち、逐次プログラムの最適化と、ノード内でのスレッド並列（OpenMP やコンパイラによる自動並列化）の最適化について解説します。

紙面の都合から網羅的なチューニング手法の説明は割愛します。チューニング技法について網羅的に知りたい場合は、日立社が東大ユーザ向けに公開しているチューニングマニュアル[1]がありますので、そちらをご覧ください。また、最適化や並列処理の基本となる知識については、著者による記事[2]に詳細が記載されています。この記事は、東京大学情報基盤センタースーパーコンピューティング部門の HP に掲載されていますので、そちらをご覧ください。

本稿では、SR16K の高いスレッド並列実行の性能を最大限に引き出すチューニング手法のうち、スレッドジョブとメモリの割り当ての明示的な指定方法、および、ループ分割とループ融合について説明します。

２．SR16K のハードウェア特徴：プログラミングの観点から

２．１　物理コア・論理コア・SMT 機能

SR16K の最大の特徴は、１ノード上に物理的に 32 コア（**物理コア**とよぶ）からなる高スレッド並列の並列計算機であることです。1 つの物理コアに、東大センターの運用では最大で 2 つのスレッドを割り当てることができるハードウェア構成であるので、最大 64 スレッド並列（**論理コア**でのスレッド実行、もしくは **SMT (Simultaneous Multi-Threading)**機能による実行）が可能であることです。したがって、OpenMP やコンパイラの自動並列化を適用したプログラムは、64 スレッドの並列実行ができます。64 並列もの高スレッド並列性を活用できるプログラムを作成することが、高効率な１ノード内でのスレッド並列実行をするために重要になります。

２．２　SR16K のノード内構成

図１に、SR16K のノード内構成をのせます。図１から、SR16K のノードは、４つの**チップ**（もしくは、**ソケット**とよぶことがあります）で構成されています。１チップは、８個の物理コアから構成されています。チップ間は、インターコネクトにより結線されています。チップ間の通信性能は、チップ内の通信性能に比べて性能が低いため、チップ間は疎結合、チップ内は密結合、になっているといえます。

ノード内のメモリは物理的には離れていますが結線されており、共有メモリを構成します。ただし、チップに物理的に近いメモリと、物理的に遠いメモリがあるため、各チップは物理的に近いメモリへのアクセス（ローカルメモリ・アクセス）は高速ですが、物理的に遠いメモリ

へのアクセス（リモートメモリ・アクセス）は低速になります。このように、共有メモリ内のメモリアクセス速度が、ローカルメモリとリモートメモリで異なる構成を **NUMA(Non Uniform Memory Access)構成**と呼びます。近年の計算機のメモリは、NUMA 構成であることが多いです。

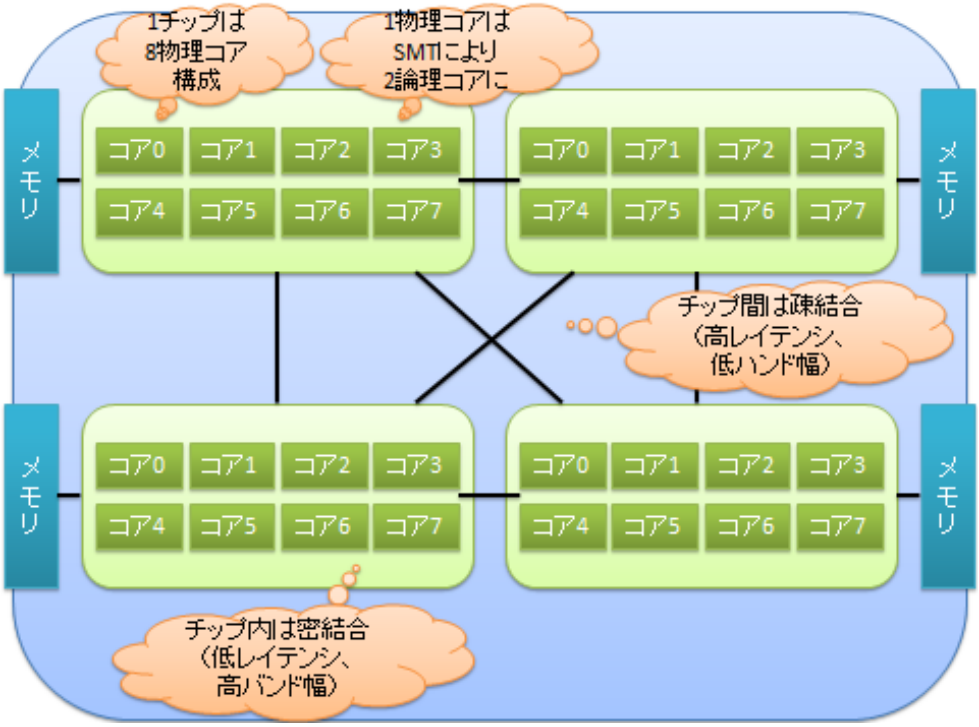

図１　SR16K のノード内構成

NUMA 構成のメモリをもつ計算機では､スレッド実行やMPI などでのプロセス実行をするとき、スレッドおよびプロセスのジョブを、どの物理コアに割り当てるか、および、割り当てたジョブか、どのメモリに配列確保するのかが、性能に影響を及ぼします。したがって、ユーザはジョブ割り当てやメモリ割り当ての方法を明示的に指定することが、一つのチューニング方式となります。

ここで重要なことは、ユーザが明示的に割り当てを指定しない場合は、SR16K をはじめとする多くの NUMA 構成の計算機では、メモリ割り当ては**インターリーブ**になることに注意すべきです。インターリーブとは、ある一つの配列を確保する場合、OS がメモリ管理する単位（ページサイズという）ごとに、循環するように、それぞれのローカルメモリに割り当てられる方式です。したがって配列は、SR16K では４つあるローカルメモリに、ページサイズごとに循環的に配置される点に注意してください[1]。

２．３　SR16K のノード内キャッシュメモリ構成

キャッシュメモリの構成を考慮することも重要です。図２に、SR16K のチップ内のキャッシ

[1] インターリーブでのメモリ割り当ては、配列がランダムにアクセスされる場合は有効といえます。一方、スレッドごとにデータアクセスが局所化されているプログラムは、インターリーブでの割り当ては不向きといえます。この場合は、後述の方法により、ユーザが直接ローカルメモリに全て配列を確保するように指定する必要があります。

ュメモリ構成を示します。

図２では、SR16K のキャッシュメモリは、各チップ上に L1、L2、L3 と 3 階層のキャッシュがあることがわかります。L1 と L2 は物理コアごとに別になっていますが、L3 は 8 個の物理コアで共有になっています。したがって、L3 はスレッド数に依存し、コア辺りのキャッシュ容量が変化します。また、コアごとに個別になっている L1、L2 キャッシュも、SMT 実行をする／しないでキャッシュ容量が変化します。表 1 に、キャッシュの物理容量を記載します。

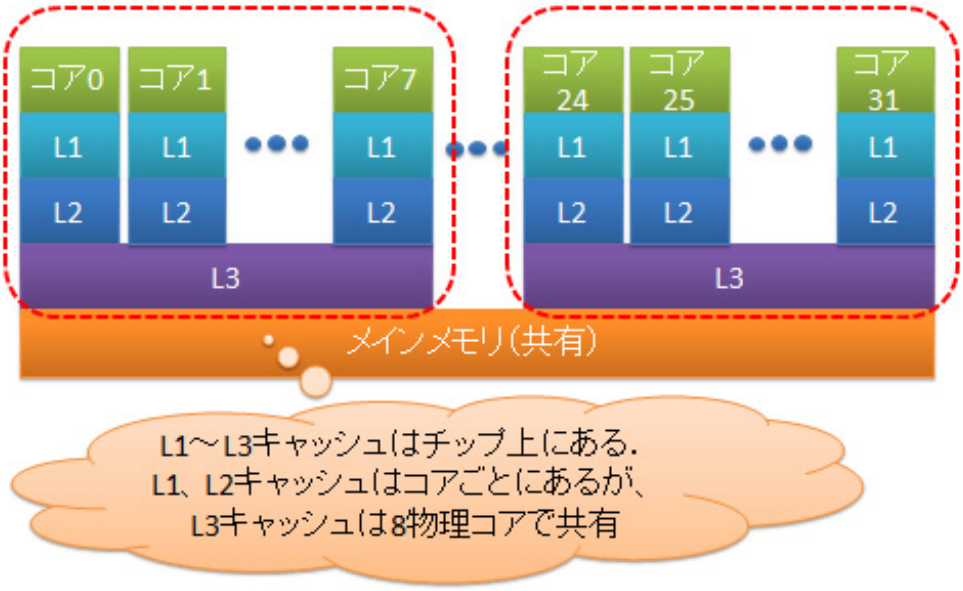

図２　SR16K のキャッシュメモリ構成

表１　SR16K のキャッシュメモリ容量（１チップ内）

キャッシュメモリの種類	物理コア	論理コア（SMT 実行時）
L1 キャッシュ	32KB/コア	16KB／コア
L2 キャッシュ	256KB/コア	128KB／コア
L3 キャッシュ（共有）	4MB／コア（8 コア実行時） （全体 32MB）	2MB/コア（16 コア実行時） （全体 32MB）

表１から、物理コア実行、論理コア（SMT）実行に依存し、キャッシュ容量が変化することに再度注意してください。また L3 キャッシュは共有のため、チップ内の 8 物理コア（16 論理コア）を使い切った時の容量が表１のものです。このサイズより小さいスレッド実行では、L3 容量はスレッド数に応じてスレッド当たりの容量が大きくなるので、性能に影響を及ぼします。

これらの３階層のキャッシュサイズからユーザが考えるべきことは、キャッシュサイズ以下の配列アクセスはキャッシュ上にデータがロードされているため高速アクセスができるので処理が高速化されることです。

SR16K は３階層キャッシュがあるためデータ移動に影響します。初期状態では、配列データ

はメインメモリ上に収納されています。配列をアクセスすると、メインメモリ上のデータはL2キャッシュとL1キャッシュを経由し、レジスタに収納されます。このとき、L2キャッシュ、L1キャッシュにもデータが収納されます。次に同じデータがアクセスされると、L1キャッシュ、もしくは、L2キャッシュからデータをレジスタに転送することができます。新たなメインメモリ上のデータがアクセスされ、かつL1もしくはL2キャッシュの容量があふれると、データは消去されます。このとき、L2キャッシュ上のデータは、L3キャッシュが空いている場合にはL3キャッシュに退避されます。今後同じデータがアクセスされる場合、メインメモリではなく、L1～L3キャッシュからデータ転送されるので、高速化されます。より上位のキャッシュにデータがあれば、より高速にレジスタまでデータが転送されるので、高速化につながります。

３．プロセスおよびメモリの割り当て方法の指定

３．１　スレッド割り当て制御

図１で説明したように、SR16K ではスレッドおよびプロセスのメモリ割り当て方式が、プログラムの配列アクセスパターンに依存し性能に影響を及ぼす可能性があります。そこで、まず初めに行うべきチューニングとして、ジョブの割り当て方式の明示的な指定があります。これは、環境変数の設定をジョブスクリプトで行い実現します。表２に、スレッド割り当て制御の環境変数の説明を載せます。

表２　スレッド割り当て制御の環境変数

環境変数	説明
HF_PRUNST_BIND=1 (0にすると非バインド)	スレッドを論理コアに固定し割り当て（バインド）する。0を指定すると、OSのジョブ割り当てポリシに従い、適当に割り当てられる。
HF_BINDPROC_NUM=0 (論理コア0からバインドする場合)	物理コアにバインドする際に、先頭となる論理コア番号を指定する。指定した論理コア番号から連続し、循環するように割り当てられる。
HF_BINDPROC_STRIDE=2 (ストライド幅2の例)	物理コアにバインドする際の、ストライド幅を指定する。ストライド幅２にすると、物理コアあたり１スレッドを割り当てられる。

特に、ストライド幅の設定(HF_BINDPROC_STRIDE=2)を指定し、HF_PRUNST_BIND=1を指定する方式を試す必要があります。というのはSR16KではSMT機能がサポートされているので、１物理コアあたり2スレッド実行となりますが、2スレッドを割り当てる場合、１論理コアあたりのキャッシュサイズが小さくなります。このHF_BINDPROC_STRIDE=2指定により、1物理コアあたり１スレッド割り当てが保証されますので、１論理コアあたりのキャッシュサイズの増加が望めます。さらに、メインメモリからキャッシュまでの物理結線の転送容量が、１物理コアあたり１スレッドに限定することで、スレッド当たりの転送容量が増加されます。その結果、メインメモリのアクセス頻度が高い＜メモリ負荷の高い＞プログラムでは、高速化される可能性があります。

３．２　メモリ割り当て制御

表３に、メモリ割り当て制御のための環境変数を載せます。特に、MEMORY_AFFINITY=MCM の効果は大きいと思われますので、まず設定すべき環境変数といえます。

一方、スタックサイズを指定する HF_PRUNST_STACKSIZE は、コンパイラによる自動並列化を行い、64 スレッド実行することで多くのスタック容量が必要とされる場合など、スタックサイズ増加の指定をしないと動作しないことがあります。状況に応じてご確認ください。

表３　メモリ割り当て制御のための環境変数

環境変数	説明
HF_PRUNST_STACKSIZE=65536 (64MB の例)	スタックサイズを KB 単位で指定する。
MEMORY_AFFINITY=MCM	各スレッドが使用するメモリをローカルメモリに確保する。 デフォルトはインターリーブで確保される。

３．３　利用方法

表２、表３の環境変数は、ジョブスクリプトに記載した上で、キューへジョブを投入する必要があります。利用しているシェルに依存して記述方法が違いますので、各自確認してください。たとえば、bash で設定する場合は、図３のようになります。

```
#!/bin/bash
#@$-q  parallel
#@$-N  1
#@$-lM 170GB
#@$-lT 00:10:00
export HF_BINDPROC_STRIDE=2
export HF_PRUNST_BIND=1
export MEMORY_AFFINITY=MCM
export OMP_NUM_THREADS=32    ←32 スレッド実行時の指定 (OpenMP 利用時)
./a.out > out_T32.txt
```

図３　スレッド制御の環境変数の利用例（bash）

４．ループ分割とループ融合

SR16K は論理コアが１ノードあたり 64 並列もある高並列スレッド実行の計算機です。したがって、OpenMP のスレッド並列化やコンパイラでの自動スレッド並列化にかかわらず、高いスレッド並列化が可能な逐次プログラムの書き方でないと、１ノード上のスレッド実行で高性能化は望めません。そこで本節では、ループ分割とループ融合について紹介することにします。

SR16K はベクトル計算機ではなく、スカラ計算機（キャッシュ計算機）に分類されます。したがって、単にベクトル長を長くする書き方では、一般のキャッシュ計算機と同様に、高速化に限度があります。キャッシュ計算機に向くブロック化手法は SR16K でも効果的であることに

注意してください。また、レジスタを有効活用するためのループアンローリングも有効です。これらの技法については、一般的なチューニング解説書、日立社チューニングマニュアル[1]、もしくは、著者による解説記事[2]などをご参考ください。

４．１　ループ分割

まず初めに、ループ分割について説明します。このチューニング手法は、ループ中に多数の配列アクセスや式が書かれているとき、レジスタがあふれることにより、メインメモリにデータを書き戻すコード（スピルコード）が生成されるのを防ぐことによる高速化手法です。レジスタ上のデータの有効活用により、高速化が達成されることを目指すチューニング技法です。

余談になりますが著者の経験では、ベクトル計算機を使っていたユーザは、ループ内に多数の式を記載するプログラムを書く傾向があるように思います。この理由は、ベクトル計算機はメインメモリからレジスタへの転送能力がキャッシュ計算機に比べて極めて高いので、ループ中に多数の式を記載してもレジスタあふれが生じなく、かつ高性能を実現できることに起因します。しかしその副作用として、このようなコードをそのままキャッシュ計算機で実行するとスピルコードが生成されやすくなります。結果として、性能劣化を生じます。

ここでは、以下のコードを考えます。

```
  DO K = 1, NZ
    DO J = 1, NY
      DO I = 1, NX
        SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
        SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
        SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
        SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
        SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
        SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
      ENDDO
    ENDDO
  ENDDO
```

上記コードでは、ループ中に配列アクセスが多いため、スピルコードが生成される可能性があります。ここで、このコードでは最内側のＩループについて式を２分割しても、演算結果に違いがない性質を利用し、以下のように書きなおすことができます。これを、**Ｉループに対するループ分割**と呼びます。

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
        SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
        SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
```

```
        SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
    ENDDO
    DO I = 1, NX
        SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
        SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
        SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO
```

一方、このループでは、J ループ、および K ループに対しても、それぞれ独立にループ分割できます。特に、K ループに対してループ分割すると、２つの独立した３重ループが形成されます。これを以下に示します。

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
        SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
        SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
        SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO

DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
        SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
        SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
        SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO
```

このように完全にループが分割されると、場合により、コンパイラの各種最適化が適用されるようになって、高速化される可能性があります[2]。

[2] この例題のように簡単な式が書かれているループの場合は、コンパイラが自動でループ分割を行い最適化することが多いと思います。しかし、ループ中の式が多くなり複雑になると、コンパイラによるデータ依存解析ができなくなって、ループ分割がされなくなりコンパイラによる最適化も限定されます。つまるところ、逐次コードもシンプルに記載するのが、コンパイラ

４．２　ループ融合

前節のループ分割は主に、レジスタへのデータ移動を最小にすることで高速化を狙うチューニング技法でした。ここでは、外側のループ長を長くすることで、高スレッド並列化（OpenMP 並列化やコンパイラによる自動スレッド並列化）環境での高速化を狙うチューニング技法であるループ融合を紹介します。

前節と同じ以下のコードに、ループ融合を施すことを考えます。

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
      SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
      SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
      SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
      SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
      SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
      SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
  ENDDO
ENDDO
```

上記ループは３重ループなので、ループ融合の可能性は、（１）K、J、I ループすべてをまとめて１重ループ化する方法；（２）J、I ループをまとめて２重ループ化する方法、の２つがあります。

まず、（１）の K、J、I ループすべてをまとめて１重ループ化するループ融合を示します。これは、以下になります。

```
DO KK = 1, NZ*NY*NX
      K = (KK-1)/(NY*NX) + 1
      J = mod((KK-1)/NX,NY) + 1
      I = mod(KK-1,NX) + 1
      SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
      SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
      SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
      SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
      SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
      SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
ENDDO
```

上記ループの特徴は、ループ融合され、新しく作られた KK ループ長は NZ＊NY＊NX の長さと

による高速化の鍵となります。

なり、もとのKループ長であるNZより格段に長くなることです。これは、OpenMPなどでの高スレッド並列化する場合、大変有利となります。

たとえば、NX=XY=NZ＝10のときを考えます。このとき、外側ループをOpenMPでスレッド並列化すると、元のループではループ長が10しかないので、10スレッドまでしか並列化できません。一方、１重ループ化したときのループ長は1000ですので、SR16Kのような64スレッド並列化を行っても、性能向上が期待できます。

このループの欠点は、K、J、Iの変数の値が関数（mod関数）で写像されるため、K、J、Iで参照される配列のメインメモリからの呼び出しに関し、コンパイラによるデータの読み出し最適化（データプリフェッチ）が阻害されることです。その結果、データ読み出し時間の増大を招き、結果として、実行時間が遅くなります。

まとめると､データ読み出し時間は増えるが､スレッド並列化による速度向上が望める場合、上記のコードは高速になります。

次に、（２）のJ、Iループをまとめて２重ループ化するループ融合の例を紹介します。これは、以下になります。

```
DO KK = 1, NZ*NY
    K = (KK-1)/NY + 1
    J = mod(KK-1,NY) + 1
    DO I = 1, NX
      SXX (I,J,K) = ( SXX (I,J,K) + (RL(I) + RM(I)*DX(I,J,K))*DT )*QG(I)
      SYY (I,J,K) = ( SYY (I,J,K) + (RL(I) + RM(I)*DY(I,J,K))*DT )*QG(I)
      SZZ (I,J,K) = ( SZZ (I,J,K) + (RL(I) + RM(I)*DZ(I,J,K))*DT )*QG(I)
      SXY (I,J,K) = ( SXY (I,J,K) + (RM(I)*(DDX(I,J,K)+DDY(I,J,K)))*DT )*QG(I)
      SXZ (I,J,K) = ( SXZ (I,J,K) + (RM(I)*(DDX(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
      SYZ (I,J,K) = ( SYZ (I,J,K) + (RM(I)*(DDY(I,J,K)+DDZ(I,J,K)))*DT )*QG(I)
    ENDDO
ENDDO
```

上記コードでは、最外側のKKループの長さがNZ＊NYの長さになるので、高並列スレッド実行に向いたコードになります。かつ、最内のIループは連続アクセスなため、コンパイラによるデータプリフェッチも阻害しません。結果として、ループ長を長く保つスレッド並列化に向いたループになり、かつ、コンパイラによるデータ読み出し最適化も阻害しないコードとなります。結果として、最も効果的なループになる可能性があります。

４．３　コンパイラ専用ディレクティブの利用

以上説明した、ループ分割やループ融合は、内部に書かれる式が単純な場合はコンパイラが自動で行いますが、複雑になると手で書き直さないと効果がないことがあります。

手でコードを書き直すのが面倒な場合、コンパイラによっては、ユーザが直接、指定するループに指示を与えることで、自動でループ分割やループ融合のコード変換を行ってくれる場合があります。この方式の欠点は、計算機環境が変わり、使っているコンパイラが利用できなく

なると、この最適化が行えなくなることです。

このような問題を解決するため、まだ研究段階ですが、チューニング専用言語で特定の指示を与えることで、Fortran90 言語、もしくは C 言語のコードを自動生成し、コンパイラの種類に依存せずループ分割とループ融合を行える計算機言語とプリプロセッサを開発しているプロジェクトがあります。この詳細は、著者らによる *ppOpen-HPC* プロジェクト[3]での *ppOpen-AT* の開発をご参照ください。

日立最適化 Fortran90 コンパイラを利用する場合、以下の指示行（ディレクティブ）により、ループ分割とループ融合を行うことが出来ます[4]。

まずループ分割ですが、以下のディレクティブをループ内の文中に記載することで、その場所でループ分割をすることができます。

```
*soption DISTRIBUTE_POINT
```

次にループ融合ですが、以下のディレクティブをループの先頭に記載することで、ループ融合を行うことが出来ます。depth は、ループ融合を行う場合のループの深さを指定します。

```
*soption LOOPFUSE〔(depth)〕
```

なお、ループによっては、内部に書かれている式に依存関係があり、ループ融合やループ分割ができないことがあります。

コンパイラによる上記の適用結果を見るためには、最適化の適用結果を報告するコンパイラオプション“-loglist”をつけてコンパイルしたあと、自動生成されるファイル“<ファイル名>.log”を見てください。上記以外の最適化についても、ループごとに最適化適用結果の報告がされます。チューニング時に役に立つオプションといえます。

５．コードチューニング事例

ここでは、ある有限差分法の実シミュレーションコードに現れる主要カーネルの１つに、ループ分割とループ融合を施した場合の性能評価結果を載せます。

このプログラムでは、最外ループを OpenMP によりスレッド並列化しています。以下に、元のプログラムを以下に載せます。

```
!$omp parallel do private(k,j,i,STMP1,STMP2,STMP3,STMP4,RLD,RMD,RM2D,
!$omp&            RMAXYD,RMAXZD,RMAYZD,RLTHETAD,QGD,NUM_THREAD)
      DO K = 1, NZ
           NUM_THREAD = omp_get_thread_num() + 1
      DO J = 1, NY
      DO I = 1, NX
           STMP1 = 1.0/RIG(I,J,K)
           STMP2 = 1.0/RIG(I+1,J,K)
           STMP4 = 1.0/RIG(I,J,K+1)
```

```
            STMP3 = STMP1 + STMP2
            RLD(I,NUM_THREAD)  = LAM (I,J,K)
            RMD(I,NUM_THREAD)  = RIG (I,J,K)
            RM2D(I,NUM_THREAD) = RMD(I,NUM_THREAD) + RMD(I,NUM_THREAD)
            RMAXYD(I,NUM_THREAD) = 4.0/(STMP3 + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
            RMAXZD(I,NUM_THREAD) = 4.0/(STMP3 + STMP4 + 1.0/RIG(I+1,J,K+1))
            RMAYZD(I,NUM_THREAD) = 4.0/(STMP3 + STMP4 + 1.0/RIG(I,J+1,K+1))
```

<ループ分割ポイント１>

```
            RLTHETAD(I,NUM_THREAD) = (DXVX(I,J,K)+DYVY(I,J,K)+
                DZVZ(I,J,K))*RLD(I,NUM_THREAD)
            QGD(I,NUM_THREAD)  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
```

<ループ分割ポイント 2>

```
            SXX (I,J,K) = ( SXX (I,J,K) + (RLTHETAD(I,NUM_THREAD) +
                RM2D(I,NUM_THREAD) *DXVX(I,J,K))*DT ) *QGD(I,NUM_THREAD)
            SYY (I,J,K) = ( SYY (I,J,K) + (RLTHETAD(I,NUM_THREAD) +
                RM2D(I,NUM_THREAD) *DXVX(I,J,K))*DT )*QGD(I,NUM_THREAD)
            SZZ (I,J,K) = ( SZZ (I,J,K) + (RLTHETAD(I,NUM_THREAD) +
                RM2D(I,NUM_THREAD)* DZVZ(I,J,K))*DT )*QGD(I,NUM_THREAD)
```

<ループ分割ポイント 3>

```
            SXY (I,J,K) = ( SXY (I,J,K) + (RMAXYD(I,NUM_THREAD)*
              (DXVY(I,J,K)+DYVX(I,J,K)))* DT )* QGD(I,NUM_THREAD)
            SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZD(I,NUM_THREAD)*
              (DXVZ(I,J,K)+DZVX(I,J,K)))* DT )*QGD(I,NUM_THREAD)
            SYZ (I,J,K) = ( SYZ (I,J,K) + (RMAYZD(I,NUM_THREAD)*
              (DYVZ(I,J,K)+DZVY(I,J,K)))* DT )*QGD(I,NUM_THREAD)
        END DO
        END DO
        END DO
!$ omp end parallel
```

上記のプログラムにおいて、**<ループ分割ポイント x>**とあるのは、この分割ポイントの組合せでループ分割を行うことを意味しています。ただし、上記の分割点は最適な点ではなく、ユーザが直観的に有効と思える分割点を示しています。

この元プログラムに対して､ループ分割とループ融合を行った以下の 13 種類のコードの実行時間を、SR16K の 1 ノード（64 スレッド実行）を用いて性能評価しました。

以下に、13 種類のプログラムの概略を載せます。

- #1： 元の３重ループ
- #2～#5： #1 のコードに対する、ループ分割ポイントによるループ分割
- #6： #1 のコードのループ融合(K と J ループ、２重ループになる)
- #7： #1 のコードのループ融合（K と J と I ループ、１重ループになる）

● #8～#13： #2～#5 のループに対するループ融合（K と J ループ、２重ループになる）

問題サイズは、NX=33、NY=32、NZ=16 です。

図４に、ジョブおよびメモリ割りあて指定を明示的にしない場合の、各実行時間を載せます。

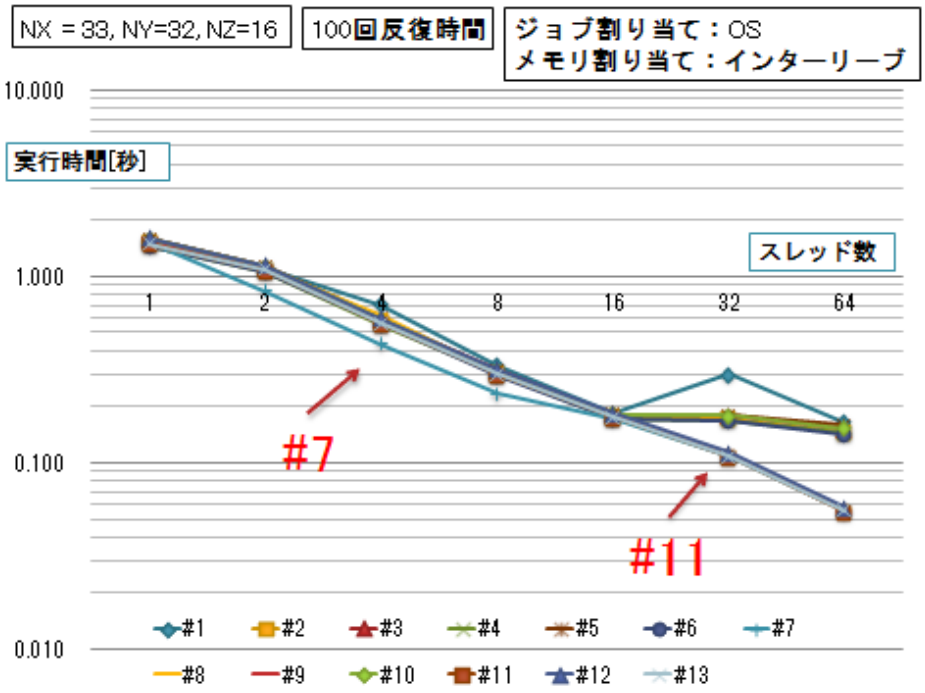

図４　各コードの実行時間[秒]。ジョブおよびメモリ割り当て指定なし。

図４から、８スレッド実行までは＃７のコードが高速でした。これは、３重ループを１重ループ化したループ融合コードです。32 スレッド実行から、＃１～＃５までのコードは、性能向上が望めなくなりました。この理由は、OpenMP で並列化されるのは最外ループの K ですが、ループ長が NZ=16 のため、16 スレッドまでしか並列性がないことによります。

一方、ループ融合したコード＃６～＃１３までは、32 スレッド以上でも高速化されます。16 スレッド以上で高速となるのは＃１１のコードですが、＃７～＃１３のうち、＃１０を除くコードは、ほぼ実行時間が同じで有意な差はありませんでした。

図５に、ジョブおよびメモリ割りあて指定をした場合（HF_BINDPROC_STRIDE=2, HF_PRUNST_BIND=1, MEMORY_AFFINITY=MCM）の実行時間を載せます。

図５から、図４と比較すると各プログラムについて実行時間の違いが無くなり、実行時間が均一になる傾向があります。これが明示的にジョブ割り当てとメモリ割り当てを行った効果といえます。

また図５では、16 スレッドまでは＃４のコード（<ループ分割ポイント 3>で分割をしたコード）が高速です。32 スレッドを超えると、＃１３のコードが最高速となりますが、＃７～＃１３のうち、＃１０を除くコードは実行時間がほぼ同じであり、有意な差はありませんでした。

ジョブおよびメモリ割り当てを明示的に行う場合と、行わない場合で高速となる実装の実行時間を比較することは興味深いです。そこで、ジョブおよびメモリ割り当てを明示的に行う場合については＃７と＃１１のコード、明示的に行わない場合については＃４と＃１３のコードの実行時間をまとめたものを表４に載せます。

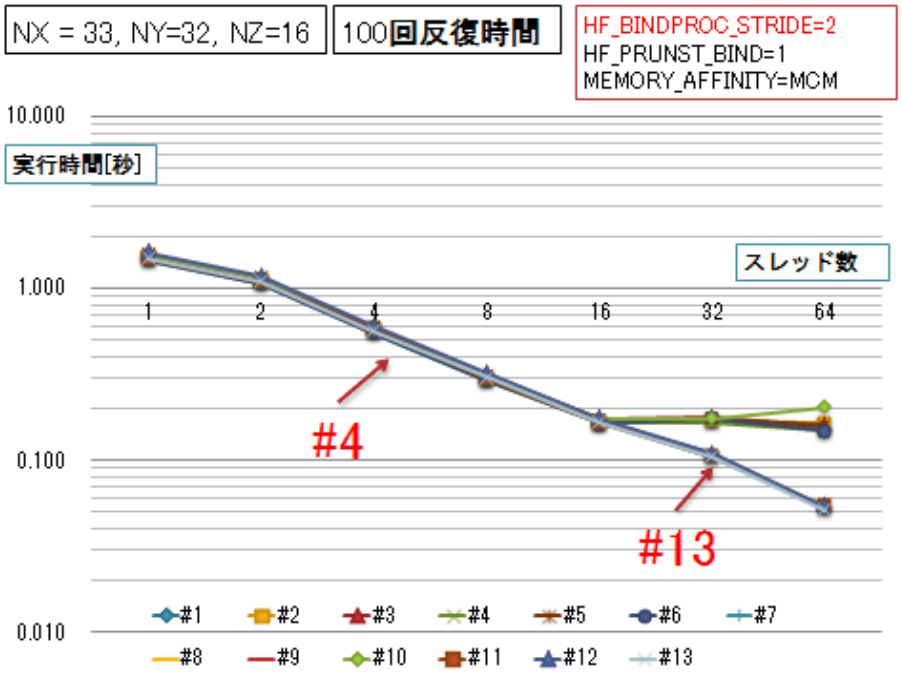

図５　各コードの実行時間[秒]。ジョブおよびメモリ割り当て指定あり。

表４　ジョブおよびメモリの明示的な割り当てあり／なしで
高速となる実装の実行時間[秒]。

#Threads	1	2	4	8	16	32	64
#7	1.526	0.838	0.437	0.235	0.174	0.109	0.055
#11	1.499	1.084	0.563	0.297	0.175	0.108	0.055
#4	1.465	1.075	0.554	0.294	0.165	0.167	0.148
#13	1.500	1.101	0.566	0.305	0.168	0.106	0.051

HF_BINDPROC_STRIDE=2
HF_PRUNST_BIND=1
MEMORY_AFFINITY=MCM

表４から、この問題サイズの場合は、全体を通して＃７の実装がほぼ最適といえます。＃７は元のコードをループ融合して１重ループ化したものですので、SR16K にとっては最外ループ長を長くするチューニングが、高スレッド並列実行で有効となることを意味しています。

一方、１スレッド実行では、＃４の場所でループ分割をする実装が高速です。この場合の元ループの実行時間は 1.55 秒ですので、＃４の逐次最適化で約 6%高速化できます。

さいごに、図６に、元のコードに対する、＃７のコードの速度向上を載せます。

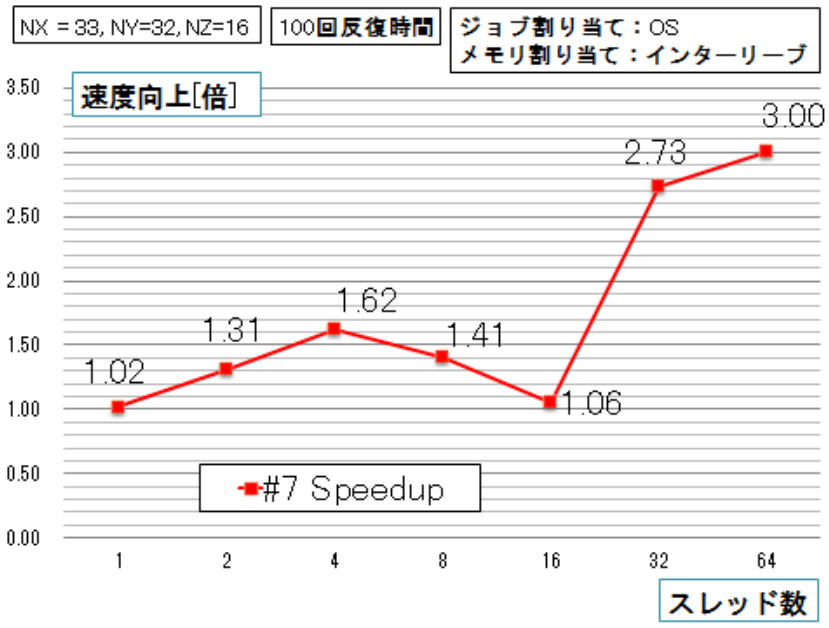

図６　元のコードに対する＃７コードの速度向上

図６から、スレッド数が 16 を超えると元のループは並列性を抽出できなくなるため、＃７のコードの＃１コードに対する速度向上率は大きくなります。64 スレッド実行時では、約３倍の速度向上を達成できます。SR16K におけるスレッド並列化において、ループ融合によるチューニングは無視できない技術であるといえます。

６．おわりに

本稿では、SR16K における、単体ノード性能チューニング手法⌐ ≈ ™ ≡　　⇔ ╕ ⇔ √ ⁹ SR16K では、ノードあたり物理的に 32 コアを有するのですが、SMT 機　⌐ ╠ ╢　　◖ ▪ ─　　⌐ ╠ ╡ ⁸ １ノードあたり 64 スレッド実行まで可能となる＜高スレッド　　⌐ ⌂ ╡ ╕ Δ ⁹

このような高スレッド計算機環境においては、（１）明示的なスレッドのコアへの割り当て指定、（２）明示的なローカルメモリへの配列確保の指定、が性能向上のために必要となります。また、OpenMP やコンパイラの自動並列化によるスレッド並列化においては、最外側ループのループ長をできるだけ長くする目的で、ループ融合技術が重要になると予想されます。それに加えて、レジスタを有効活用するためのループ分割も効果的である可能性があります。

本稿では説明を割愛しましたが、SR16K はキャッシュ計算機のため、単にベクトル長を長くするだけではなく、ある程度のサイズでループを分割して、＜ブロック化＞する技術も重要になります。これらのことを考慮して、性能チューニングをしていく必要があります。

性能チューニングのためには、プログラムの性能プロファイルを取り、それをもとに性能解析をすることが必須です。SR16K では、日立 Fotran90 コンパイラと連結した“pmpr”コマンドによる性能解析が可能です。このコマンドの利用方法は、日立製作所によるチューニングマニュアル[1]をご覧ください。

最後になりますが、本稿が少しでも皆様のプログラムの性能向上の参考になるのであれば、まことに幸甚です。

参 考 文 献

[1] (株) 日立製作所：スーパテクニカルサーバ HITACHI SR16000 モデル M1 チューニングマニュアル (2011)

https://yayoi-man.cc.u-tokyo.ac.jp/manual-j/index.html

[2]片桐孝洋：T2K オープンスパコン（東大）チューニング連載講座：高性能プログラミング（I）入門編，スーパーコンピューティングニュース，東京大学情報基盤センタースーパーコンピューティング部門，Vol.10，No.4（2008 年 7 月）

http://www.cc.u-tokyo.ac.jp/support/press/news/

[3]ppOpen-HPC プロジェクト HP

http://ppopenhpc.cc.u-tokyo.ac.jp/

[4] (株)日立製作所：SR16000 最適化 FORTRAN90　使用の手引、2011 年 1 月 (3000-3-C51)

以上