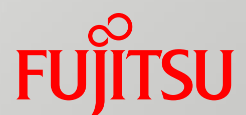
Oakbridge-CX チューニングマニュアル

1.0版
2019年6月
富士通株式会社

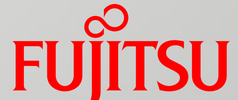
Copyright 2019 FUJITSU LIMITED

本書の内容



本書は、スーパーコンピュータシステム向けの情報を
記載しています。
プログラムはFortranを前提としています。

Copyright 2019 FUJITSU LIMITED



1. チューニングの考え方

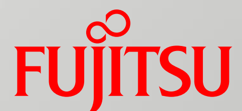
1.1 プロファイリングとチューニング

1.2 チューニングを行うための準備

1.3 スカラチューニングとスレッド並列化

Copyright 2019 FUJITSU LIMITED

1.1 プロファイリングとチューニング



プロファイリング情報(VTune)や、タイマーによる測定結果等を用いて高コスト部を特定する

改善方法検討する(VTune、コンパイル時の最適化レポートによる解析)

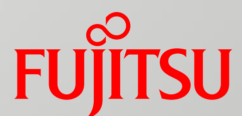
ソースコード変更
コンパイルオプションの選択

効果の確認

プログラムを改善して実行時間を短縮することをチューニングといいます。プロファイリング情報等を使用し、高コスト処理を特定し、必要な対策を施すことで、実行時間の短縮を図ります。

Copyright 2019 FUJITSU LIMITED

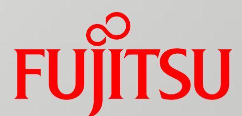
1.2 チューニングを行うための準備



目的	使用ツール／関数等
ホットスポットの特定	ツールを使用して、プログラムの高コスト部を特定する。 ・VTune Amplifier ・Adviser
任意の区間測定（時間）	プログラムの任意の区間を測定するために、タイマー関数をプログラムに組み込み測定を行う。 ・MPI_Wtime ・system_clock
最適化状況の確認	どのような最適化が行われたか、コンパイル時にオプションを指定することで、最適化レポートが出力される。 ・ -qopt-report=1～5 数字が大きいほど、詳細な最適化情報を出力 ・ -qopt-report-annotate ソースリストに最適化情報が付加される。annotという拡張子のファイルが出来る。

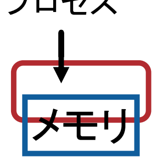
Copyright 2019 FUJITSU LIMITED

1.3 スカラチューニングとスレッド並列化



	内　容
逐次 性能向上	**スカラチューニング** SIMD化、キャッシュメモリの 有効利用等
並列化	**スレッド並列化** (1) 自動並列化 (2) OpenMP並列化
	プロセス並列化　MPI, etc

逐次(非並列)処理

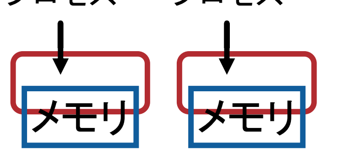


スレッド並列化
各並列処理が同じメモリを参照

プロセス
スレッド
メモリ

ノード内のみ可能(ノード間は不可)

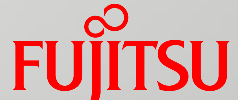
プロセス並列化
各並列処理は独立したメモリを参照



プロセス間の
通信により
計算を進める

ノード内，ノード間ともに可能

Copyright 2019 FUJITSU LIMITED

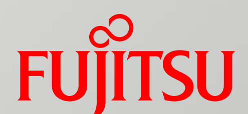


2. スカラチューニング

2.1 メモリの連続アクセス
2.2 キャッシュの仕組み
2.3 チューニング例

Copyright 2019 FUJITSU LIMITED

2.1 メモリの連続アクセス (1/3)



多次元配列のメモリ格納方式

Fortran : 列優先で格納

```
double precision a(3,4)
```

a(1,1)	a(1,2)	a(1,3)	a(1,4)
a(2,1)	a(2,2)	a(2,3)	a(2,4)
a(3,1)	a(3,2)	a(3,3)	a(3,4)

メモリ上では

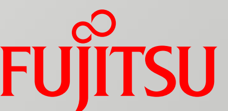
a(1,1)
a(2,1)
a(3,1)
a(1,2)
a(2,2)
⋮
a(2,4)
a(3,4)

メモリを<u>連続にアクセス</u>する ⇒ 高速

<u>左側</u>の添字が先に動くようにする

Copyright 2019 FUJITSU LIMITED

2.1 メモリの連続アクセス (2/3)



○ 連続アクセス

```
do j = 1, n
  do i = 1, n
    a(i,j)=a(i,j)+b(i,j)*c
  enddo
enddo
```

△ ストライドアクセス (幅: n)

```
do i = 1, n
  do j = 1, n
    a(i,j)=a(i,j)+b(i,j)*c
  enddo
enddo
```

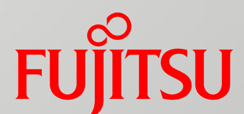
可能であるなら，連続アクセスするように変更

(1) ループの順序の入れ替え
(2) 配列次元の入れ替え

コンパイラが最適化の一環として，ループの順序の入れ替えを行う場合があります．
その状況は，コンパイラの翻訳時メッセージ等にて確認できます．

Copyright 2019 FUJITSU LIMITED

2.1 メモリの連続アクセス (3/3)



C言語のメモリ格納方式
Fortranとは正反対

C：行優先で格納

```
double a[3][4];
```

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

メモリ上では

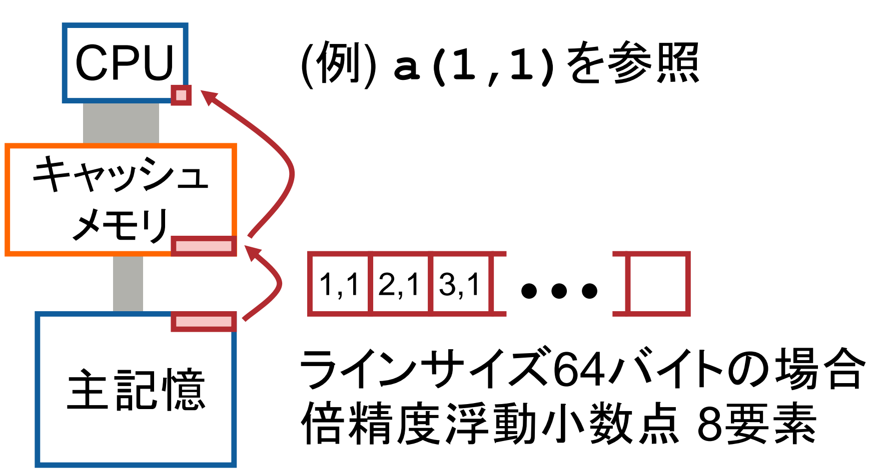
a[0][0]
a[0][1]
a[0][2]
a[0][3]
a[1][0]
⋮
a[2][2]
a[2][3]

Copyright 2019 FUJITSU LIMITED

FUJITSU

キャッシュメモリと主記憶間のデータの移動

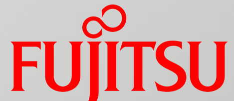
⇒決まった大きさの連続領域単位(キャッシュライン)



○ 連続アクセス
　キャッシュラインの再利用

△ ストライドアクセス
　キャッシュライン中に
　すぐには使わないデータ
　(使う時にはすでに
　キャッシュにない可能性)

Copyright 2019 FUJITSU LIMITED



2.3 チューニング例

2.3.1 連続アクセス

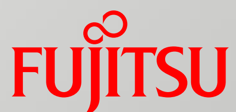
2.3.2 作業配列の次元縮小

2.3.3 IF文の除去

2.3.4 ループ依存関係

Copyright 2019 FUJITSU LIMITED

2.3.1 連続アクセス



変更前

```
do i = 1, NMAX
  do j = 1, NMAX
    do k = 1, NMAX
      X(i,j,k) = A(i,j,k) + B(i,j,k)*C
    enddo
  enddo
enddo
```

変更後

```
do k = 1, NMAX
  do j = 1, NMAX
    do i = 1, NMAX
      X(i,j,k) = A(i,j,k) + B(i,j,k)*C
    enddo
  enddo
enddo
```

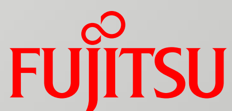
メモリアクセスがストライドから連続になり、
キャッシュラインが有効に活用される。

上記例のような単純なループであれば、コンパイラが自動的に最適化を実施します。
コンパイル時に以下のメッセージが出力されていれば、最適なアクセスに自動的に変更されています。

「remark #25444: Loopnest Interchanged: (1 2 3) --> (3 2 1)」

Copyright 2019 FUJITSU LIMITED

2.3.2 作業配列の次元縮小



変更前

```
!---- Loop1 -----
do k = 1, NMAX
  do j = 1, NMAX
    do i = 1, NMAX
      X(i,j,k) = A(i,j,k) + B(i,j,k)*C
      temp(i,j,k) = X(i,j,k)*X(i,j,k)
      temp2(i,j,k)= D(i,j,k)*C
    enddo
  enddo
enddo

!---- Loop2 -----
do k = 1, NMAX
  do j = 1, NMAX
    do i = 1, NMAX
      Y(i,j,k)=(temp(i,j,k)+D(i,j,k))
&              *temp2(i,j,k)/B(i,j,k)+ E(j,k)
    enddo
  enddo
enddo
```

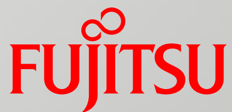
変更後

```
!---- Loop1 -----
do k = 1, NMAX
  do j = 1, NMAX
    do i = 1, NMAX
      X(i,j,k) = A(i,j,k) + B(i,j,k)*C
      temp = X(i,j,k)*X(i,j,k)
      temp2= D(i,j,k)*C
      !---- Loop2 -----
      Y(i,j,k)=(temp+D(i,j,k))*
              * temp2 / B(i,j,k)+E(j,k)
    enddo
  enddo
enddo
```

temp，temp2を配列から変数に変更する。
temp，temp2は、ループ中で一時的に使用しており、全ての値を保持する必要は無い。

Copyright 2019 FUJITSU LIMITED

2.3.3 IF文の除去



変更前

```
do i = 1, nx-1
  if(i .eq. 1) then
    do k = 1, nz
      do j = 2, ny-1
        A(i,j,k) = A(i,j,k) * 0.5d0
      enddo
    enddo
  endif

  do k = 1, nz
    do j = 2, ny-1
      B(i,j,k) = func(A(i+1,j,k),A(i,j,k))
    enddo
  enddo
enddo
```

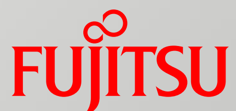
変更後

```
i = 1

do k = 1, nz
  do j = 2, ny-1
    A(i,j,k) = A(i,j,k) * 0.5d0
  enddo
enddo

do k = 1, nz
  do j = 2, ny-1
    do i = 1, nx-1
      B(i,j,k) = func(A(i+1,j,k),A(i,j,k))
    enddo
  enddo
enddo
```

最外のループを最内に移動する。
不要な条件判定を削減でき、かつ、
メモリの連続化もできる。

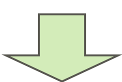
Copyright 2019 FUJITSU LIMITED

2.3.4 データ依存関係



■ 定義引用関係が不明な場合

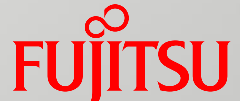
```
do j=1,n2
   do i=1,n1
      a( la(i), j ) = a( lx(i), j ) / b(i, j)
   end do
end do
リマーク #15344: ループ はベクトル化されませんでした: ベクトル依存関係がベクトル化を妨げています。
```



```
do j=1,n2
!DIR$ IVDEP
   do i=1,n1
      a( la(i), j ) = a( lx(i), j ) / b(i, j)
   end do
end do
リマーク #15300: ループがベクトル化されました。
```

配列aのデータ依存関係が不明なため、ベクトル化が適用されていません。
定義と参照に依存性がない場合は、IVDEP宣言子を指定し、依存性がないこと
を明示することで最適化が適用されます。

Copyright 2019 FUJITSU LIMITED

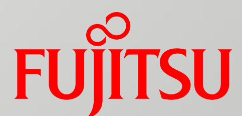


3. スレッドチューニング

3.1 スレッド並列化の考え方
3.2 スレッド並列における性能阻害要因
3.3 OpenMPによる並列化
3.4 外側のループで並列化
3.5 並列化のオーバヘッド
3.6 チューニング例

Copyright 2019 FUJITSU LIMITED

3.1 スレッド並列化の考え方



スレッド並列化：doループの繰り返しを分割

```
do i = 1, 1000
   a(i)=a(i)+b(i)*c
enddo
```

スレッド0

```
do i1 = 1, 500
   a(i1)=a(i1)+b(i1)*c
enddo
```

スレッド1

```
do i2 = 501, 1000
   a(i2)=a(i2)+b(i2)*c
enddo
```

スレッド並列化の実現方法

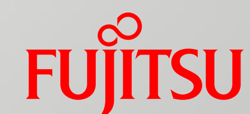
(1) コンパイラによる自動並列化

(2) OpenMP指示文による
明示的な並列化

- 回帰参照のあるループは並列化困難
- write/read文やユーザ定義ルーチンを含むループは，自動並列化されない

Copyright 2019 FUJITSU LIMITED

3.2 スレッド並列における性能阻害要因

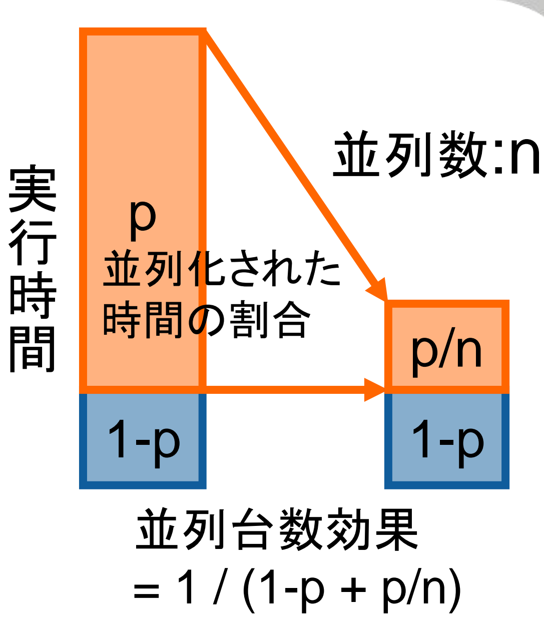


- アムダールの法則

  コストの高い部分が並列化されていないと，並列化の効果は小さい

  ⇒コンパイラの翻訳時メッセージより

  スレッド並列化状況の確認

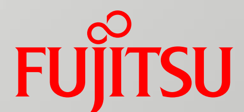


- 並列化のオーバヘッド ➡ 後述
- キャッシュメモリのfalse-sharing
- メモリアクセスネック

  ⇒スカラチューニングによる，キャッシュの有効利用

Copyright 2019 FUJITSU LIMITED

3.3 OpenMP指示文による並列化



指示文(directive)をソースに追加

- Fortranのコメントの形式
- コンパイルオプション ⇒ 意味を持つ

OpenMP 指示文
「ループ並列化を指示」

```
!$OMP PARALLEL DO 指示節
      do j = 1, ny
        do i = 1, nx
          . . .
        enddo

        do i = 2, nx-1
          . . .
        enddo
      enddo
```

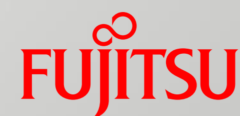
OpenMP並列化の注意点 (1)

- 指示文は強制的に並列化する
  コンパイラは，(回帰参照等の)
  意味をチェックしない

OpenMPについての詳細
http://www.openmp.org/
OpenMP Application Program Interface

Copyright 2019 FUJITSU LIMITED

3.4 外側のループで並列化



多重ループの並列化:

原則, より外側のループを並列化する

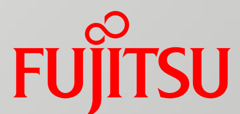
スレッド並列化するループ

```
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      a(i,j,k) = ~
    enddo
  enddo
enddo
```

並列性があるのにもかかわらず, 自動並列化が,
より外側ループで並列化しない場合 (ループ構造が複雑のため)
⇒ OpenMP指示文よる明示的な並列化 を利用

Copyright 2019 FUJITSU LIMITED

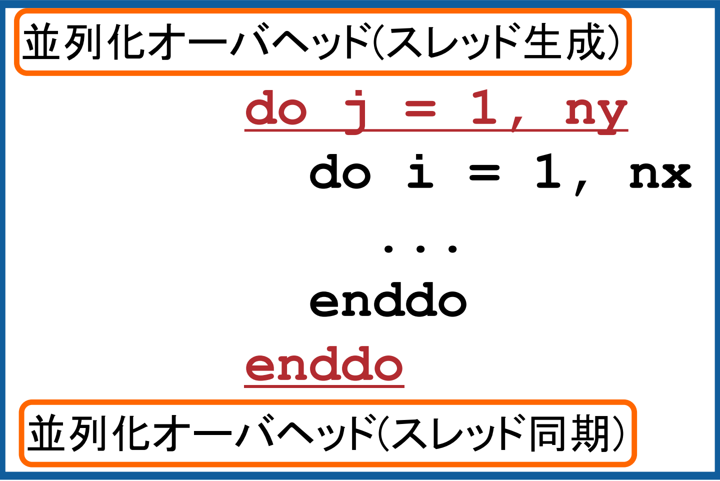
3.5 並列化のオーバヘッド



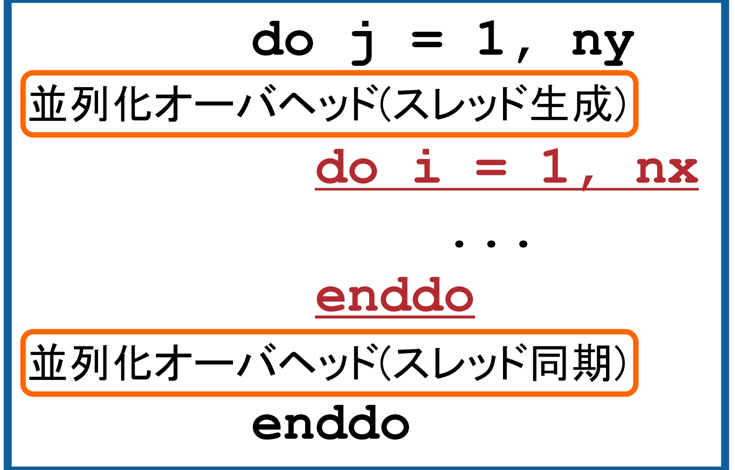
並列化のオーバヘッド：

並列化ループの開始/終了時に必要な処理

外側ループjで並列化

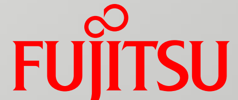


内側ループiで並列化



ループjが回転するたびに，
並列化オーバヘッドがかかる

Copyright 2019 FUJITSU LIMITED



3.6 チューニング例

3.6.1 スレッド間ロードインバランスの改善1

3.6.2 スレッド間ロードインバランスの改善2

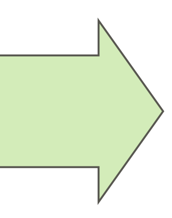
Copyright 2019 FUJITSU LIMITED

3.6.1 スレッド間ロードインバランスの改善1

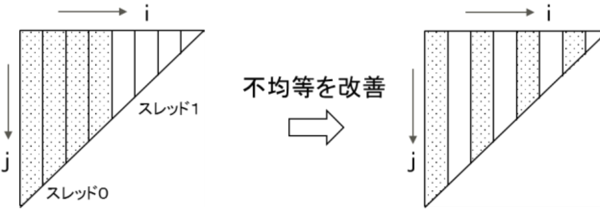
FUJITSU

■ 三角ループの場合

```
subroutine sub(a,b,c,n)
  integer*8 i,j,n
  real*8 a(n+1,n),b(n+1,n),c(n+1,n)
!$omp parallel do
  do j=1,n
    do i=j,n
      a(i,j)=b(i,j)+c(i,j)
    enddo
  enddo
end
```



```
subroutine sub(a,b,c,n)
  integer*8 i,j,n
  real*8 a(n+1,n),b(n+1,n),c(n+1,n)
!$omp parallel do schedule(static,1)
  do j=1,n
    do i=j,n
      a(i,j)=b(i,j)+c(i,j)
    enddo
  enddo
end
```



内側ループのループ長が外側ループの制御変数によって決まるループをブロック分割で並列化すると、ロードインバランスが発生します。
ループを細かくサイクリックに各スレッドに割り当てることで、処理量を均一化します。

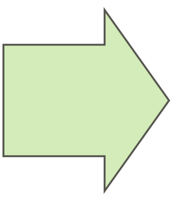
Copyright 2019 FUJITSU LIMITED

3.6.2 スレッド間ロードインバランスの改善2

FUJITSU

■ 処理量が不規則な場合

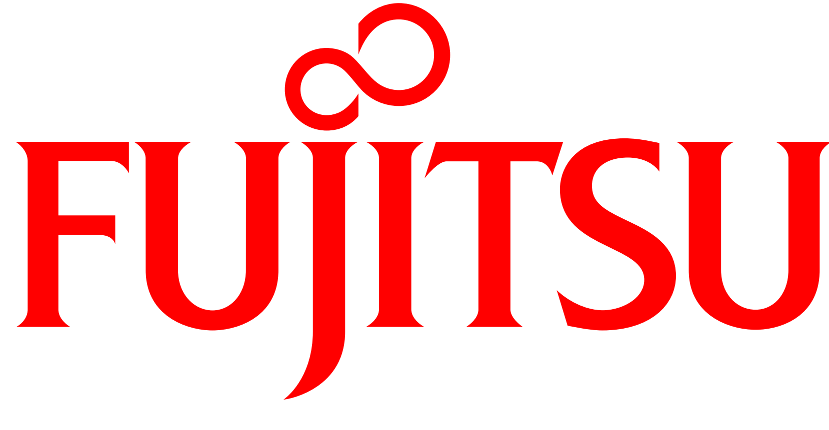
```
subroutine sub(a,b,s,n,m,nn,kt)
  real*8 a(m,n),b(m,n)
  real s
  integer kt(nn)
!$omp parallel do schedule(static,1)
   do k=1,nn
     if( kt(k) .ge. 0 ) then
       do j=1,n
         do i=1,m
           a(i,j) = a(i,j)*b(i,j)*s
         enddo
       enddo
     endif
   enddo
end subroutine sub
```



```
subroutine sub(a,b,s,n,m,nn,kt)
  real*8 a(m,n),b(m,n)
  real s
  integer kt(nn)
!$omp parallel do schedule(dynamic,1)
   do k=1,nn
     if( kt(k) .ge. 0 ) then
       do j=1,n
         do i=1,m
           a(i,j) = a(i,j)*b(i,j)*s
         enddo
       enddo
     endif
   enddo
end subroutine sub
```

IF構文があり、毎回の処理量が異なるようなループを並列処理した場合、ロードインバランスが発生することがあります。このような場合、スケジュールをdynamicにすることで、先に終了したスレッドが次の処理を行えるようになり、ロードインバランスが改善します。

Copyright 2019 FUJITSU LIMITED



shaping tomorrow with you