Fortranプログラム
～チューニングの「いろは」～

高性能Fortran推進協議会
核融合科学研究所
坂上仁志

チューニングは重要か?

- 実行時間10秒のプログラムが,10倍速くなって1秒で計算が終わるようになっても,あまりうれしくない.
- でも,一週間かかる計算が2日で終わるようになると,研究の進展度合いが大きく変わる!
  - 大規模で長時間かかる計算を行うプログラムほどチューニングは重要である.
- しかし,完成したプログラムをチューニングするのは大変なので,最初から「計算が速い」プログラムを書くことが基本である.
  - ただし,プログラムの可読性を損なわないことも,プログラムの保守を考えれば重要である.

コンパイラの最適化

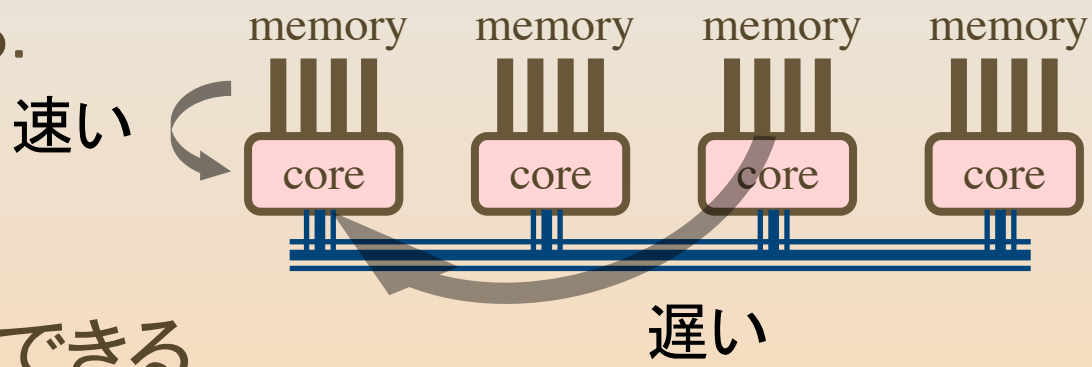
- まずは,ソースを修正しなくてもできることをやる.
- コンパイラによる最適化
  - 一般には,レベル1,2,3が指定できる.(-O1, -O2, -O3)
  - その上で特別なオプションが指定できることもある.(-Ofast)
  - 最適化すると計算順序が変わるため,実行結果が異なることが多いので,十分に確認する.
    - (a + b) + c ≠ a + (b + c)
    - 最適化レベル0(-O0,最適化しない)の結果と比べると良い.
  - 最適化レベル0では正しく計算できても,最適化すると計算結果が不正になる場合がある.
    - 最適化レベル0では,コンパイラが全変数をクリアしてくれたりするので,未初期化変数(ゼロクリアを忘れた)や未定義変数(変数名をタイプミスした)の参照が原因であることが多い.

コンパイラの自動並列化

- コア数が多いコンピュータで有効である.
- コンパイラが,自動的にスレッド並列化してくれる.
  - 前述の最適化と組み合わせて利用することもできる.
  - 並列化すると,逆に遅くなる場合もある.
- 同様に計算順序が変わるため,実行結果が異なることが多いので,十分に確認する.
  - 総和計算,最大値を持つインデックス計算
- ただし,並列化の能力は,一般に,コンパイラによって大きく異なるため,計算性能がコンパイラに依存性する.
  - あるコンピュータでは速く計算できるが,別のコンピュータでは計算が遅い.
- スレッド並列化するときは,OpenMPの利用を考える.
  - 指示行を挿入するだけでスレッド並列化ができる.

アフィニティ(近接性)

- コアが遠くのメモリをアクセスすると実行が遅くなるため,コアとメモリの結び付きを適切に指定すると,実行が速くなる場合が多い.
  - 指定しないと適当に割り振られる.
  - メモリサイズに注意する.



- numactlコマンドで指定できる.

```
% numactl --membind=2 --cpunodebind=2 ./a.out < input.dat > output.list
```

プロファイリング

- 一般に20%のサブルーチンが80%のCPU時間を使うと言われているので,計算が重たい部分,いわゆるホットスポットからチューニングすると,作業効率が良い.
- そこで,どのサブルーチンがどのくらいのCPU時間を使っているのかを調べる.
  - これを一般に,プロファイリングと呼ぶ.
- 手順は以下になる.
  1. gfortran なら -pg オプションを付けて再コンパイルする.
  2. 実行すると gmon.out というファイルが作成される.
  3. % gprof ロードモジュール名 gmon.out とするとプロファイル情報が出力される.

gprofの出力例

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 30.84     23.63    23.63      795    29.72    29.72  c1pef_
 10.51     31.68     8.05      795    10.13    10.13  c1fdi_
 10.22     39.51     7.83      795     9.85     9.85  c1fde_
  7.92     45.58     6.07      795     7.64     7.64  c1pif_
  7.83     51.58     6.00      794     7.56     7.56  c1fj_
  7.57     57.38     5.80      794     7.30     7.31  c1piv_
...
index % time    self  children    called     name
-----------------------------------------------
                0.01    0.00       1/795         s1init_ [24]
                6.06    0.00     794/795         main [1]
[5]      7.9    6.07    0.00     795         c1pif_ [5]
                0.00    0.00    1590/38250       fdebug_ [30]
-----------------------------------------------
                6.00    0.00     794/794         main [1]
[6]      7.8    6.00    0.00     794         c1fj_ [6]
                0.00    0.00     794/797         c1flsr_ [34]
-----------------------------------------------
```

「計算が速い」プログラム

- アーキテクチャによって異なるが,基本的には,
  - 人が読みやすいプログラムを書く.
    - コンパイラにも理解しやすく,最適化が容易になる.
  - メモリが連続アクセスされるように配列宣言／ループを書く.
  - ループ中では,IF文や外部関数／サブルーチンの呼び出しをできるだけ避ける.
  - チューニングされたライブラリを使う.
- ポインタは使わない.
  - 配列の形状がコンパイル時にわからないので,コンパイラの最適化が難しく,効率良く処理できない.
  - 性能 vs. 便利さ

「計算が速い」プログラム

- DOのインデックス変数は，コンパイラの最適化を阻害するので，
  - ループ外で参照しない．
  - implicit noneだからといってmodule/useしない．
  - インデックス変数から誘導される変数も上記と同様に扱う．

```
do i = 1, n
  ij = i * 2 - 1
  ii = i * 2
  a(i) = b(ij)**2 + b(ii)**2
end do
```

```
do i = 1, n
  ...
  if( ... ) exit
  ...
end do
ilast = i
```

```
do i = 1, n
  ...
  if( ... ) then
    ilast = i
    exit
  end if
  ...
end do
```

```
module doindex
integer :: i, j, k
end module
subroutine sub1
use doindex
implicit none
do k =
  do j =
    do i =
      ...
subroutine sub2
use doindex
implicit none
do i =
  ...
```

```
subroutine sub1
implicit none
integer :: i, j, k
do k =
  do j =
    do i =
      ...
subroutine sub2
implicit none
integer :: i
do i =
  ...
```

「計算が速い」プログラム

- 加減乗除
  - 加減<乗≪除
  - ただし,2のべき乗の除算は,高速なシフト演算になることがある.
- べき乗
  - 整数べきと実数べきは違う.
  - **0.5よりSQRTを使う.
- 不変式は事前に定義する.
- 同一式は同じように書く.
  - コンパイラが最適化しやすい.
  - 可読性が高い.

```
do i = 1, n
  ene(i) = ene(i)/me/c**2.0
end do
```

```
invmec2 = 1.0/(me*c**2)
do i = 1, n
  ene(i) = ene(i) * invmec2
end do
```

```
do i = 1, n
  a(i) = a(i)*c(i)*d
  b(i) = d*b(i)*c(i)
end do
```

```
do i = 1, n
  a(i) = a(i) * (c(i)*d)
  b(i) = b(i) * (c(i)*d)
end do
```

```
do i = 1, n
  tmp = c(i) * d
  a(i) = a(i) * tmp
  b(i) = b(i) * tmp
end do
```

再計算 vs. メモリサイズ

- 中間変数は再計算するのか,保存して再利用するのかを考える.
  - メモリサイズが許す限り,できるだけ保存して再利用する方が,多くの場合に計算は速い.

```
do iy = 1, ny
  do ix = 1, nx
    cs = sqrt(gamma*p(ix,iy)/r(ix,iy))
    a(ix,iy) = ... / cs
  end do
end do
...
do iy = 1, ny
  do ix = 1, nx
    cs = sqrt(gamma*p(ix,iy)/r(ix,iy))
    b(ix,iy) = ... / cs
  end do
end do
```

```
do iy = 1, ny
  do ix = 1, nx
    invcs(ix,iy) = 1.0 / &
      sqrt(gamma*p(ix,iy)/r(ix,iy))
  end do
end do
do iy = 1, ny
  do ix = 1, nx
    a(ix,iy) = ... * invcs(ix,iy)
  end do
end do
...
do iy = 1, ny
  do ix = 1, nx
    b(ix,iy) = ... * invcs(ix,iy)
  end do
end do
```

ループアンローリング

- DOループを展開して,ループ内の実行文を増やす.
  - ループオーバーヘッド,ロード／ストアが削減できる.
  - パイプライニングによる演算の効率化が期待できる.
  - コンパイラの最適化で自動的にアンロールされることもある.
  - ただし,アンロールの段数に対して十分な数のレジスターが必要である.

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

```
do j = 1, n, 4
  do i = 1, n
    a(i) = a(i) + b(i)*c(i,j)
    a(i) = a(i) + b(i)*c(i,j+1)
    a(i) = a(i) + b(i)*c(i,j+2)
    a(i) = a(i) + b(i)*c(i,j+3)
  end do
end do
```

*ループ回数がアンロール段数で割りきれないときに注意！

インライン展開

- 外部関数やサブルーチンを導入することはプログラムの可読性向上に役立つが,ループ内で繰り返し呼び出す場合は,コーリングオーバーヘッドが大きく,かつ,コンパイラの最適化がかかりにくい.
- 外部関数／サブルーチンを呼びだし側に展開する.
  - ループアンローリングと併せて使うと効果が大きい.

```
do j = 1, n
  do i = 1, n
    ds(i,j) = funs(a(i,j),b(i,j),c(i,j))
    dc(i,j) = func(a(i,j),b(i,j),c(i,j))
  end do
end do
```

```
function funs(a, b, c)
funs = a*1.234 + sqrt(b) + sin(c)
return
end

function func(a, b, c)
func = a*6.789 + sqrt(b) + cos(c)
return
end
```

gather/scatterによる連続アクセス

- メモリへの連続アクセスにならない場合は,一時配列を用いて連続アクセスに変換する.
  - gather/scatterのコスト vs. 計算の効率化
  - SIMD化すると効果が大きい.

```
do i = 1, n
  a(i) = a(i) + b(in(i)) + c(in(i))
  d(in(i)) = d(in(i)) + b(in(i)) * c(in(i))
end do
```

```
do i = 1, n
  b$(i) = b(in(i))
  c$(i) = c(in(i))
  d$(i) = d(in(i))
end do
do i = 1, n
  a(i) = a(i) + b$(i) + c$(i)
  d$(i) = d$(i) + b$(i) * c$(i)
end do
do i = 1, n
  d(in(i)) = d$(i)
end do
```

```
do i = 1, n
  if( icnd(i) .eq. 0 ) &
    a(i) = a(i) + b(i) * c(i)
end do
```

```
i$ = 0
do i = 1, n
  if( icnd(i) .eq. 0 ) then
    i$ = i$ + 1
    a$(i$) = a(i)
    b$(i$) = b(i)
    c$(i$) = c(i)
    in$(i$) = i
  end if
end do
do i = 1, i$
  a$(i) = a$(i) + b$(i) * c$(i)
end do
do i = 1, i$
  a(in$(i)) = a$(i)
end do
```

部分配列のアクセス

- 部分配列を使うとメモリへの連続アクセスにならない場合があるので,gather/scatterと同様に一時配列を用いて連続アクセスに変換する.
  - コンパイラが自動的にパック／アンパックする場合でも,無駄な処理を削除できるかもしれない.

```
dimension :: a(100,100), b(100,100)
call sub1( a(1:10,51:60), b(1:10,51:60) )
call sub2( a(1:10,51:60), b(1:10,51:60) )
```

```
dimension :: a(100,100), b(100,100)
dimension :: a$(10,10), b$(10,10)
a$(:,:) = a(1:10,51:60)
b$(:,:) = b(1:10,51:60)
call sub1( a$, b$ )
call sub2( a$, b$ )
!a(1:10,51:60) = a$(:,:)
!b(1:10,51:60) = b$(:,:)
```

ループの分割

- 最適なループ構造が変数によって異なる場合は,それぞれが最適になるようにループを分割する.

```
do j = 1, nj
  do i = 1, ni
    a(i,j) = a(i,j) + b(i,j)
    ta(j,i) = ta(j,i) + tb(j,i)
  end do
end do
```

```
do j = 1, nj
  do i = 1, ni
    a(i,j) = a(i,j) + b(i,j)
  end do
end do
do i = 1, ni
  do j = 1, nj
    ta(j,i) = ta(j,i) + tb(j,i)
  end do
end do
```

```
do i = i, ni
  c(i) = ...
  do j = 1, nj
    a(i,j) = a(i,j) + b(i,j) * c(i)
  end do
end do
```

```
do i = i, ni
  c(i) = ...
end do
do j = 1, nj
  do i = 1, ni
!   c(i) = ...
    a(i,j) = a(i,j) + b(i,j) * c(i)
  end do
end do
```

ループの融合

- 最適なループ構造が同じ場合は,ループを分けないで単一のループにする.
  - ループアンローリングと同様なループオーバーヘッド,ロード／ストアの削減が期待できる.

```
do i = 1, n
  a(i) = a(i) + b(i) * c(i)
end do
do i = 1, n
  if( a(i) .lt. 0.0 ) a(i) = 0.0
end do
```

```
do i = 1, n
  a(i) = a(i) + b(i) * c(i)
  if( a(i) .lt. 0.0 ) a(i) = 0.0
end do
```

```
do i = 1, n
  tmp = a(i) + b(i) * c(i)
  if( tmp .lt. 0.0 ) tmp = 0.0
  a(i) = tmp
end do
```

ループの境界処理

- ループの上下限である境界について,特別な処理が必要な場合は,ループの外に出す.
  - IF文の処理コストが削減される.
  - 各種のコンパイラによる最適化が阻害されなくなる.

```
do i = 1, n
  if( i .eq. 1 ) then
    a(i) = ( b(i) + b(i+1) ) / 2.0
  else if( i .eq. n ) then
    a(i) = ( b(i-1) + b(i) ) / 2.0
  else
    a(i) = ( b(i-1)+b(i)+b(i+1) ) &
         / 3.0
  end if
end do
```

```
inv3 = 1.0 / 3.0
a(1) = ( b(1) + b(2) ) * 0.5
do i = 2, n-1
  a(i) = ( b(i-1)+b(i)+b(i+1) ) &
       * inv3
end do
a(n) = ( b(n-1) + b(n) ) * 0.5
```

ループ内の例外処理

- ループ内の例外処理をIF文で判断している場合は,なるべくIF文を除去する.
  - ループの境界処理と同様な効果が期待できる.

```
do i = 1, n
  a(i) = a(i) + b(i) * c(i)
  if( a(i).lt.0.0 ) a(i)= 0.0
end do
```

```
do i = 1, n
  a(i) = max( 0.0, &
              a(i)+b(i)*c(i) )
end do
```

```
do i = 1, n
  if( icnd .eq. 0 ) then
    a(i) = a(i) + b(i)*c(i)
  else
    a(i) = d(i)
  end if
end do
```

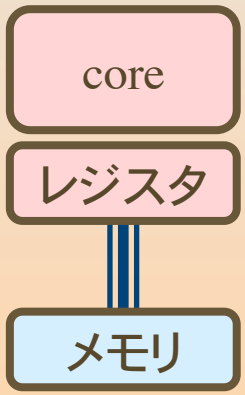
```
if( icnd .eq. 0 ) then
  do i = 1, n
    a(i) = a(i) + b(i)*c(i)
  end do
else
  do i = 1, n
    a(i) = d(i)
  end do
end if
```

```
do j = 1, n
  do i = 1, n
    if( i .ne. j ) then
      a(i,j) = b(i,j) * c(i,j)
    else
      a(i,j) = 1.0
    end if
  end do
end do
```

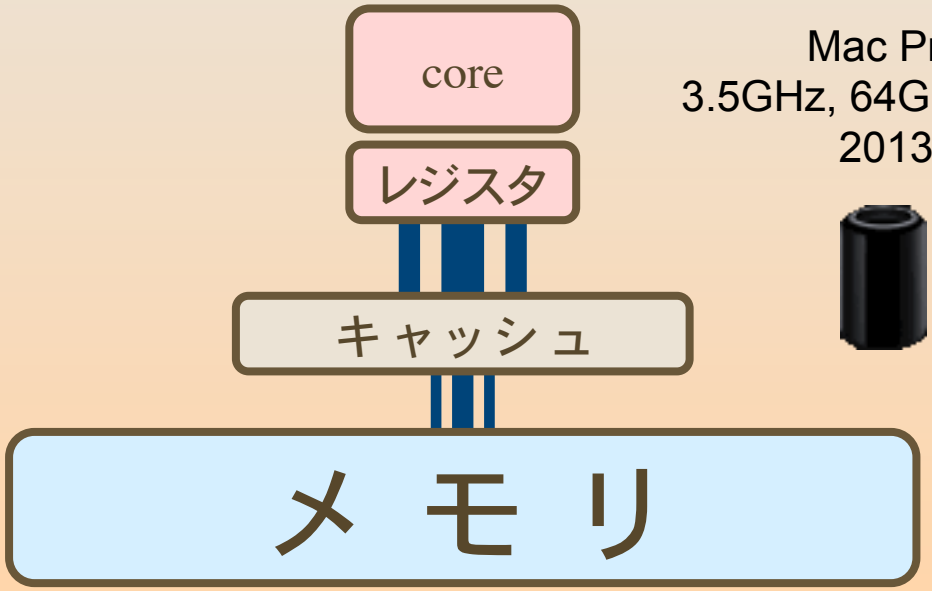
```
do j = 1, n
  do i = 1, n
    a(i,j) = b(i,j) * c(i,j)
  end do
end do
do i = 1, n
  a(i,i) = 1.0
end do
```

メモリの階層構造

- 演算能力とメモリ容量は,劇的に増大したが,レジスタとメモリ間のデータ転送速度は,技術的およびコスト的にそこまで増大させることはできなかった.
  - 演算器(レジスタ)がデータ待ちをするため,著しく計算効率が低下した.
- そこで,メモリ容量こそ少ないが,レジスタへのデータ転送速度はそこそこ速いバッファメモリ(=キャッシュ)を導入した.
  - 実際には,キャッシュも階層構造になっている.(L1, L2, L3 cache)
  - 低速・大容量 vs. 高速・低容量



Macintosh SE
8MHz, 4MB, no cache
1987



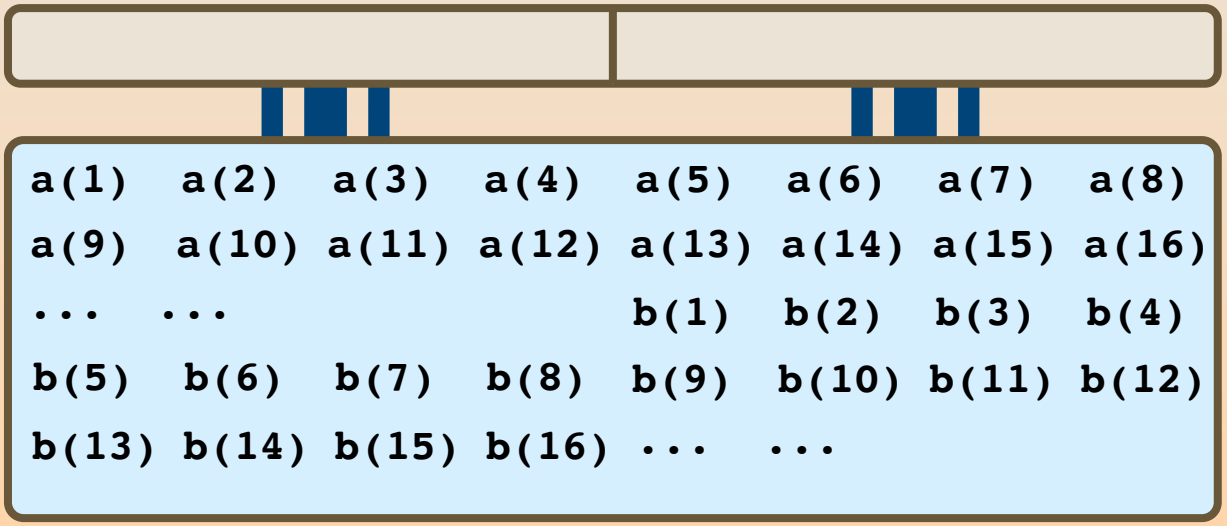
Mac Pro
3.5GHz, 64GB, 10MB
2013

キャッシュの動作

- キャッシュ上にデータがない(キャッシュミス)とメモリから転送する.
- メモリとキャッシュは,独立した複数の経路(キャッシュライン)で接続されており,決まった大きさ(ラインサイズ)のデータが一度に転送される.
  - 異なった経路では,並行して転送できる.(インターリーブ)
  - 一般にライン当たり複数の領域を持つ.
- キャッシュが一杯になると,利用率の低いデータを選んで,必要ならメモリに追い出してから,上書きする.

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

キャッシュライン数:2
ラインサイズ:16B
ライン当たり:一つ

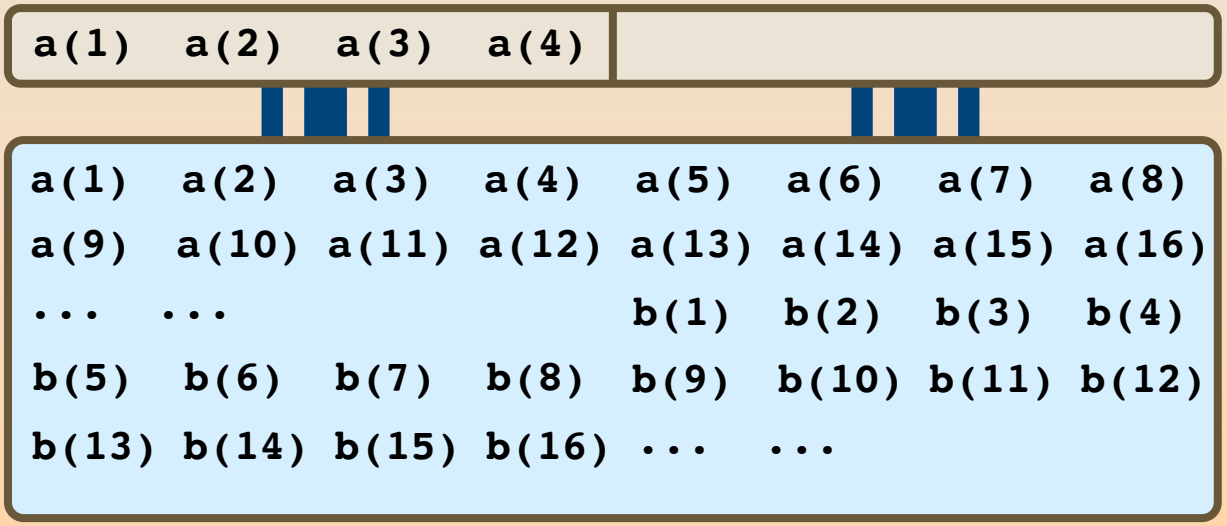


キャッシュの動作

- キャッシュ上にデータがない(キャッシュミス)とメモリから転送する.
- メモリとキャッシュは,独立した複数の経路(キャッシュライン)で接続されており,決まった大きさ(ラインサイズ)のデータが一度に転送される.
  - 異なった経路では,並行して転送できる.(インターリーブ)
  - 一般にライン当たり複数の領域を持つ.
- キャッシュが一杯になると,利用率の低いデータを選んで,必要ならメモリに追い出してから,上書きする.

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

キャッシュライン数:2
ラインサイズ:16B
ライン当たり:一つ



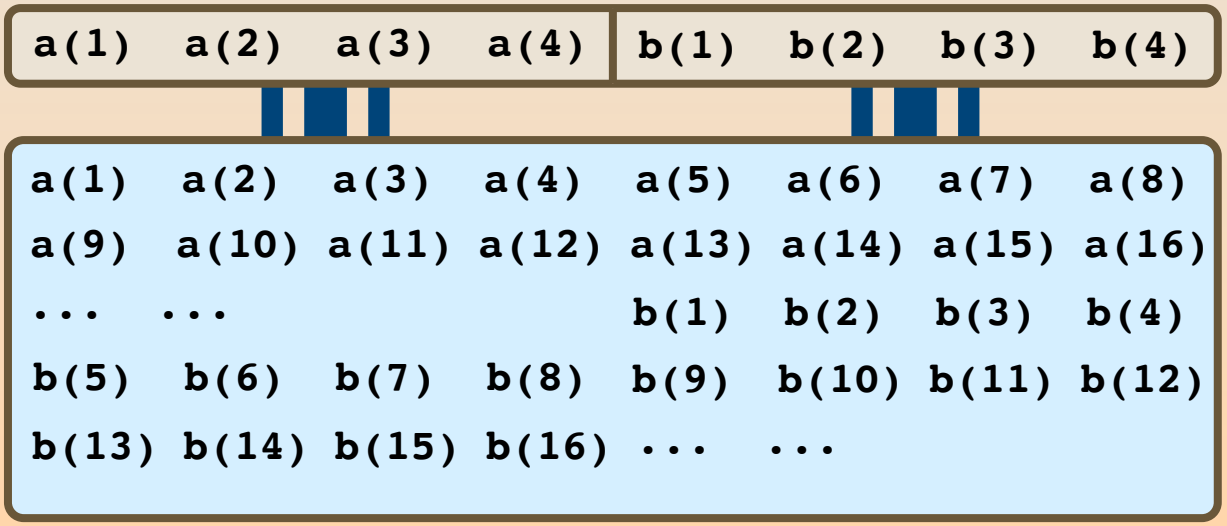
キャッシュの動作

- キャッシュ上にデータがない(キャッシュミス)とメモリから転送する.
- メモリとキャッシュは,独立した複数の経路(キャッシュライン)で接続されており,決まった大きさ(ラインサイズ)のデータが一度に転送される.
  - 異なった経路では,並行して転送できる.(インターリーブ)
  - 一般にライン当たり複数の領域を持つ.
- キャッシュが一杯になると,利用率の低いデータを選んで,必要ならメモリに追い出してから,上書きする.

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

キャッシュライン数:2
ラインサイズ:16B
ライン当たり:一つ

*実際には, aとbは同時に転送される

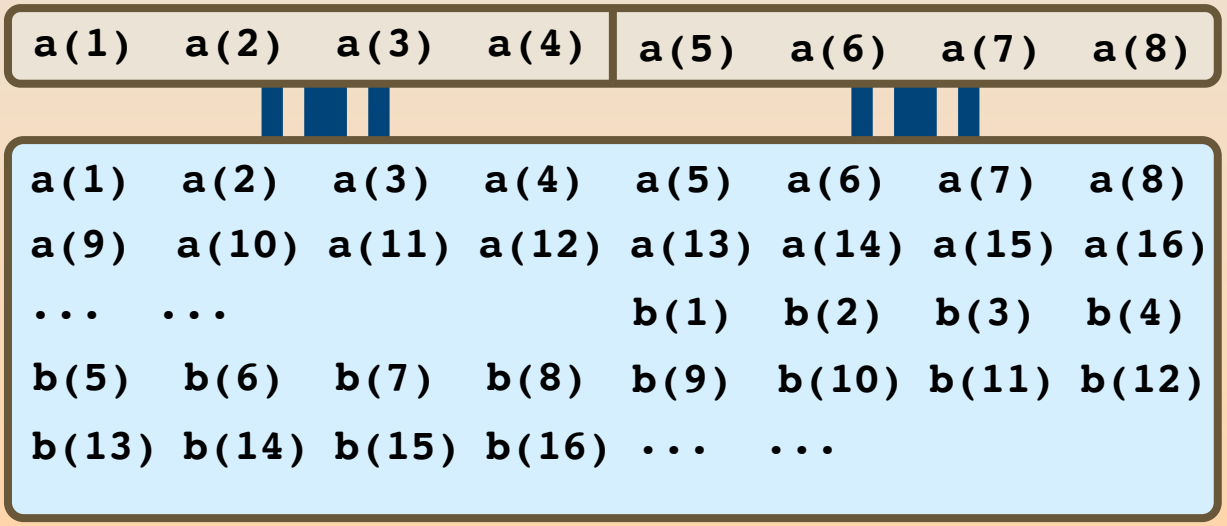


キャッシュの動作

- キャッシュ上にデータがない(キャッシュミス)とメモリから転送する.
- メモリとキャッシュは,独立した複数の経路(キャッシュライン)で接続されており,決まった大きさ(ラインサイズ)のデータが一度に転送される.
  - 異なった経路では,並行して転送できる.(インターリーブ)
  - 一般にライン当たり複数の領域を持つ.
- キャッシュが一杯になると,利用率の低いデータを選んで,必要ならメモリに追い出してから,上書きする.

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

キャッシュライン数:2
ラインサイズ:16B
ライン当たり:一つ

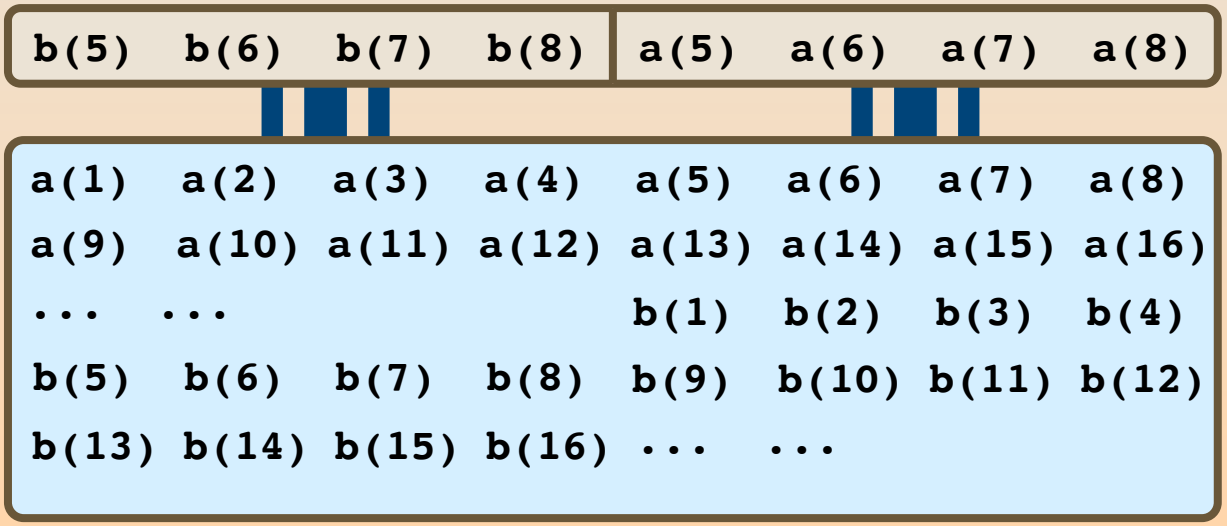


キャッシュの動作

- キャッシュ上にデータがない(キャッシュミス)とメモリから転送する.
- メモリとキャッシュは,独立した複数の経路(キャッシュライン)で接続されており,決まった大きさ(ラインサイズ)のデータが一度に転送される.
  - 異なった経路では,並行して転送できる.(インターリーブ)
  - 一般にライン当たり複数の領域を持つ.
- キャッシュが一杯になると,利用率の低いデータを選んで,必要ならメモリに追い出してから,上書きする.

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

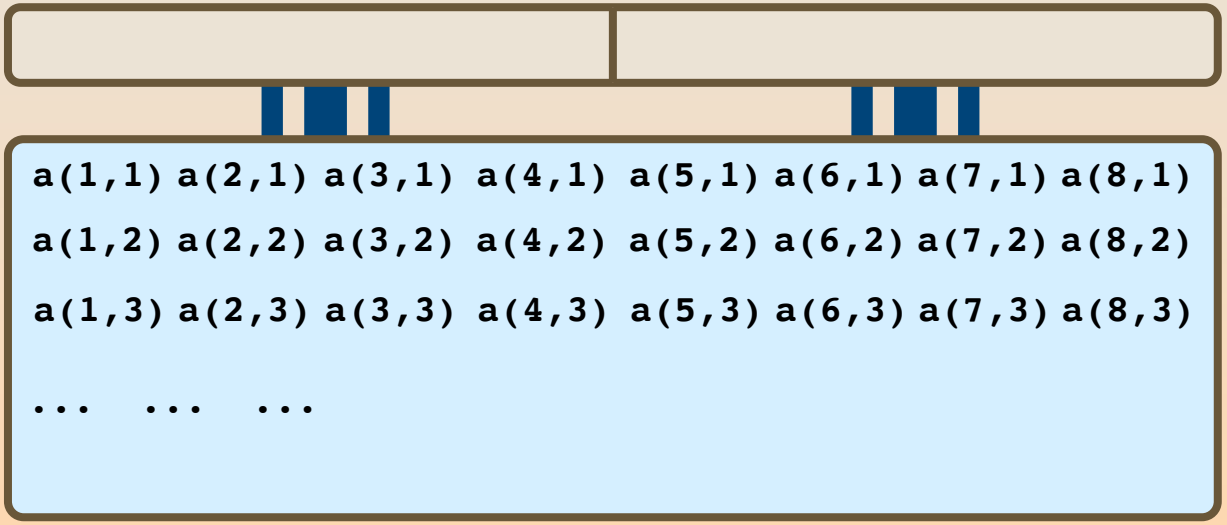
キャッシュライン数:2
ラインサイズ:16B
ライン当たり:一つ



キャッシュチューニング

- なるべくキャッシュミスが起こらないように,キャッシュ上のデータだけで計算できるようにする.
  - トイプログラムと実際のプログラムでは,チューニングの結果が異なることが多いので,基本的にはトライアル&エラーになる.

```
dimension a(8,8)
do j = 1, 8
  do i = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```

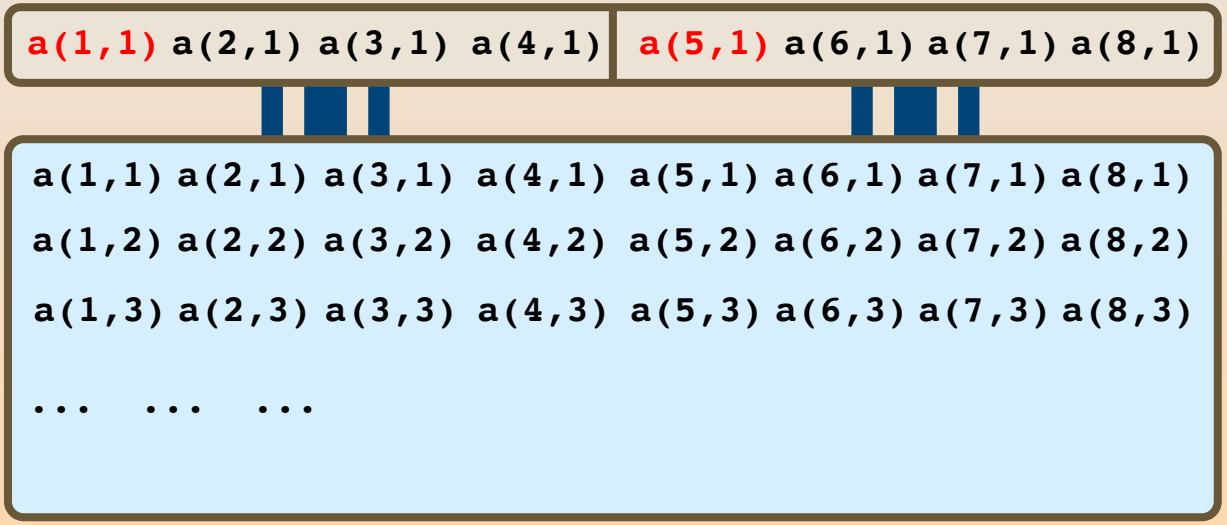


キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないように,キャッシュ上のデータだけで計算できるようにする.
  - トイプログラムと実際のプログラムでは,チューニングの結果が異なることが多いので,基本的にはトライアル&エラーになる.

```
dimension a(8,8)
do j = 1, 8
  do i = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```

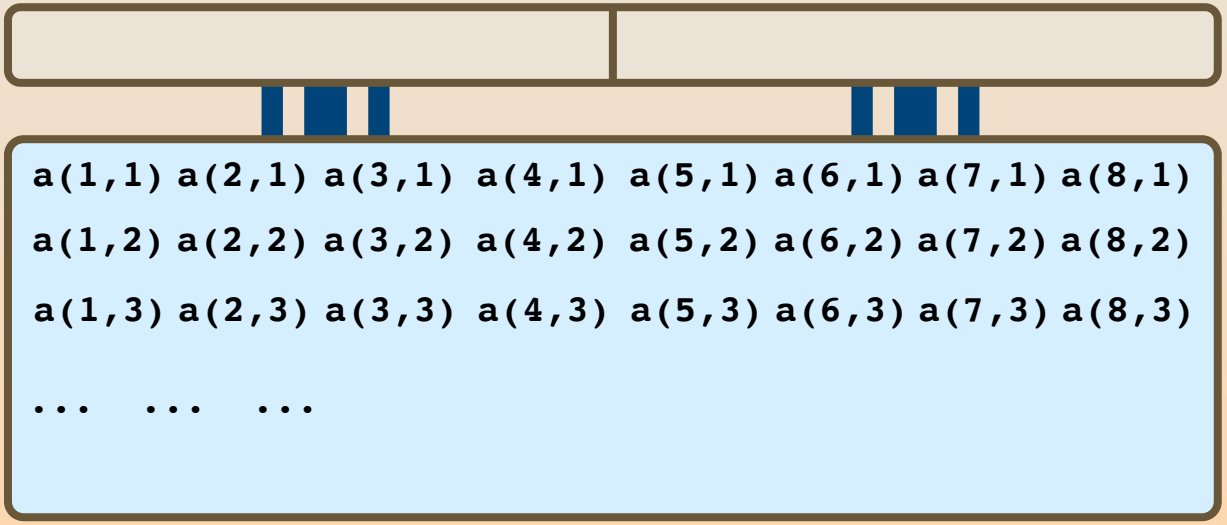


キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないように,キャッシュ上のデータだけで計算できるようにする.
  - トイプログラムと実際のプログラムでは,チューニングの結果が異なることが多いので,基本的にはトライアル&エラーになる.

```
dimension a(8,8)
do i = 1, 8
  do j = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```

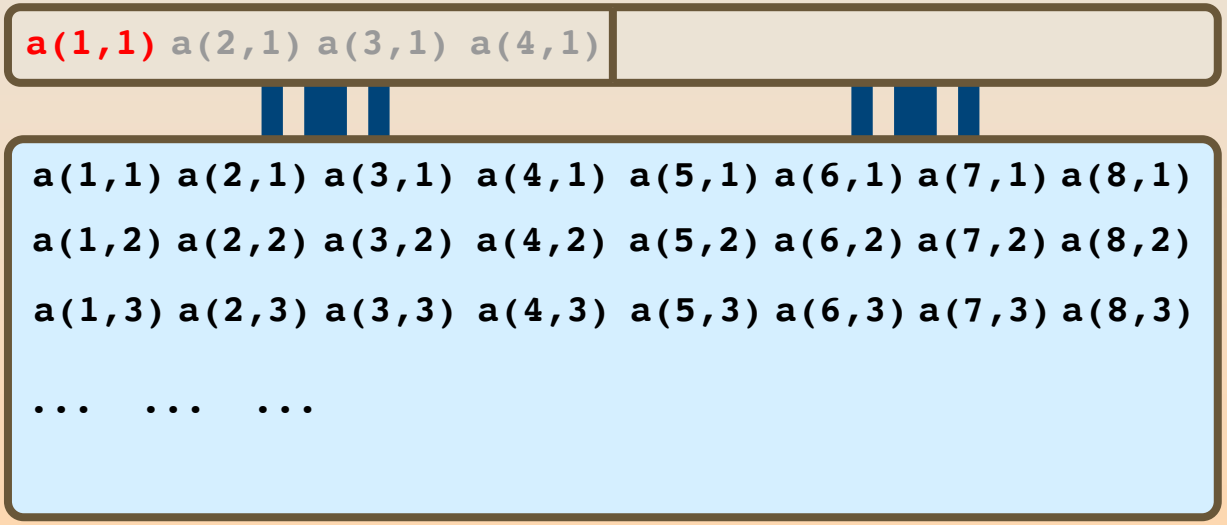


キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないように,キャッシュ上のデータだけで計算できるようにする.
  - トイプログラムと実際のプログラムでは,チューニングの結果が異なることが多いので,基本的にはトライアル&エラーになる.

```
dimension a(8,8)
do i = 1, 8
  do j = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```

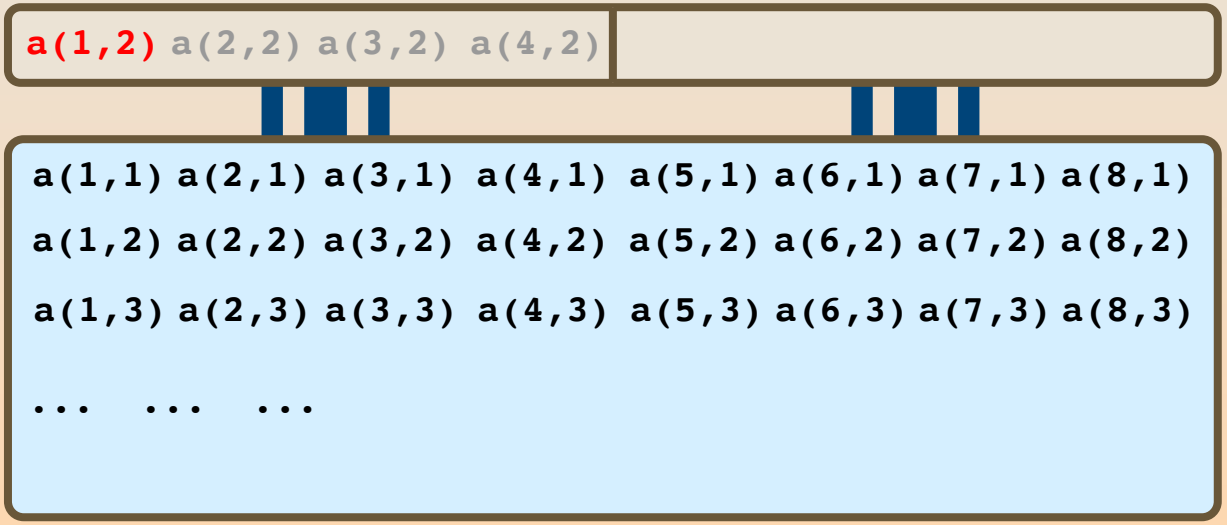


キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないように,キャッシュ上のデータだけで計算できるようにする.
  - トイプログラムと実際のプログラムでは,チューニングの結果が異なることが多いので,基本的にはトライアル&エラーになる.

```
dimension a(8,8)
do i = 1, 8
  do j = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```

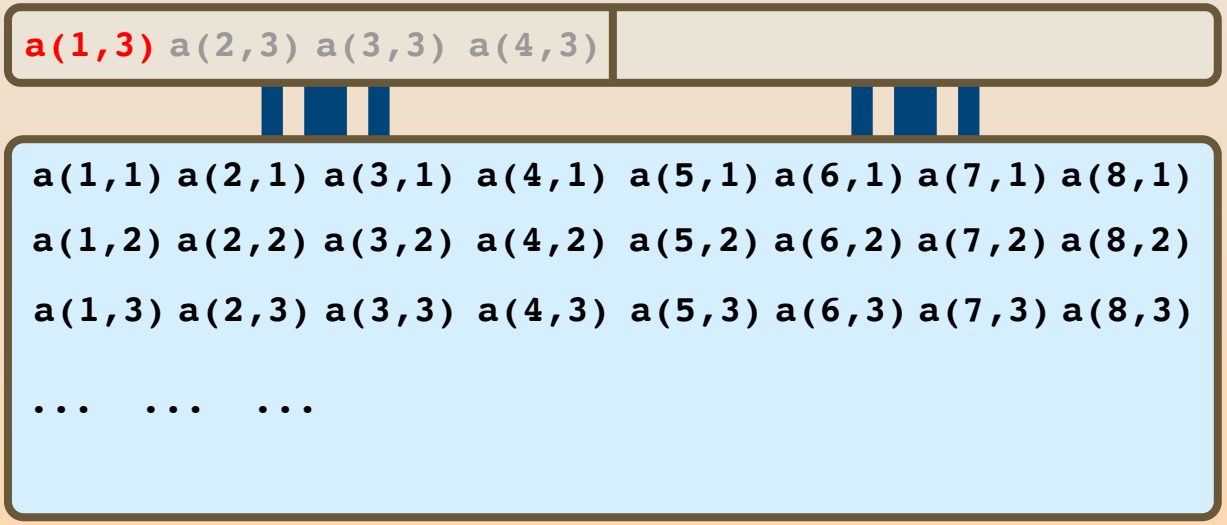


キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないように,キャッシュ上のデータだけで計算できるようにする.
  - トイプログラムと実際のプログラムでは,チューニングの結果が異なることが多いので,基本的にはトライアル&エラーになる.

```
dimension a(8,8)
do i = 1, 8
  do j = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```



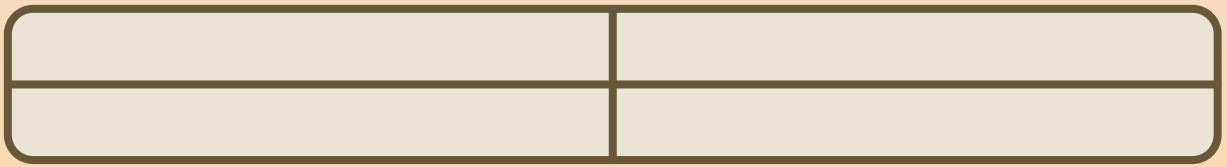
キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



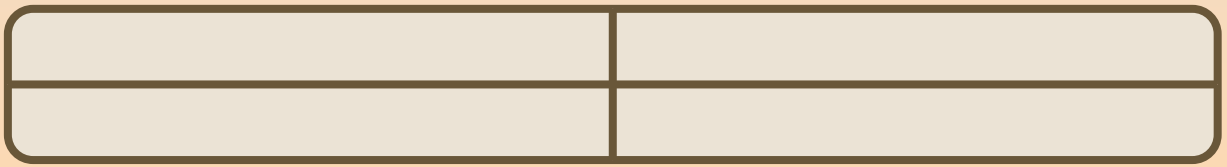
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



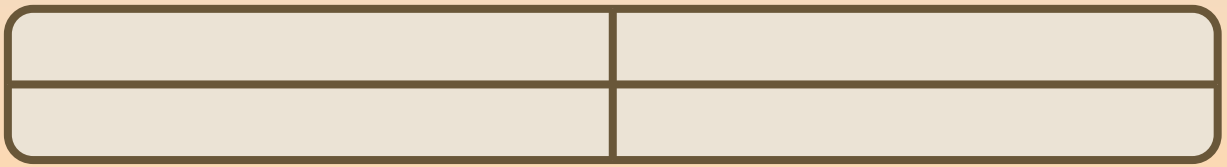
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



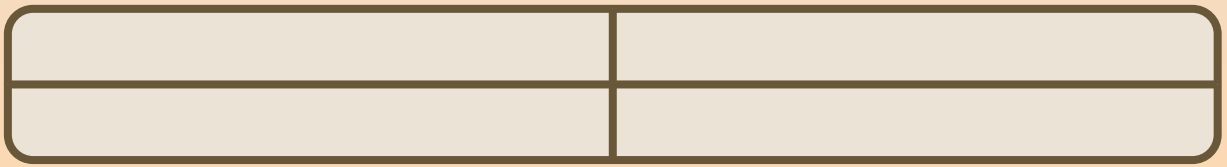
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



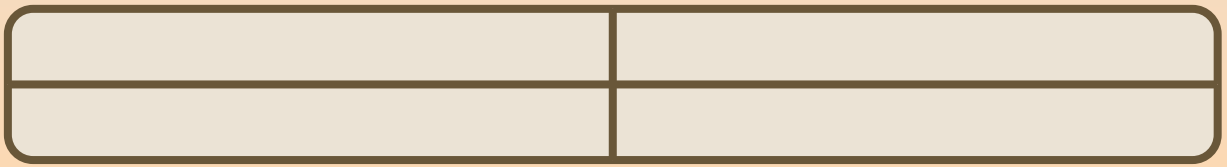
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



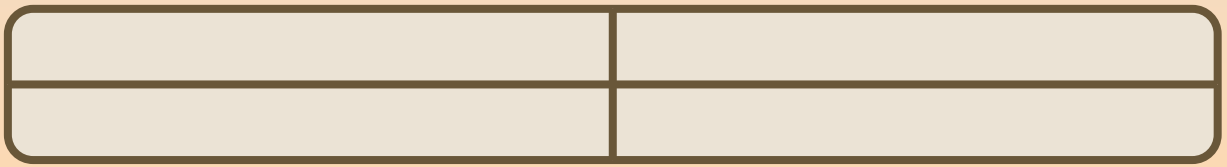
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



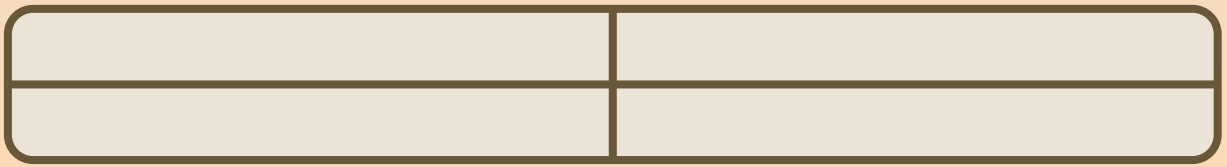
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



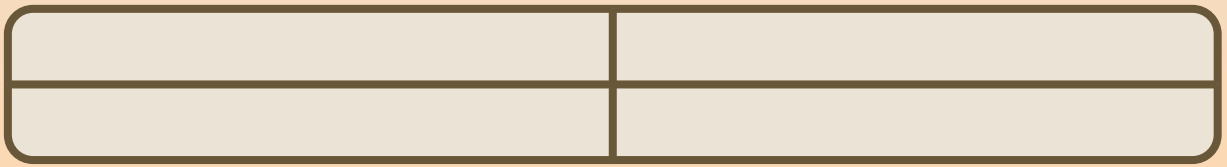
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



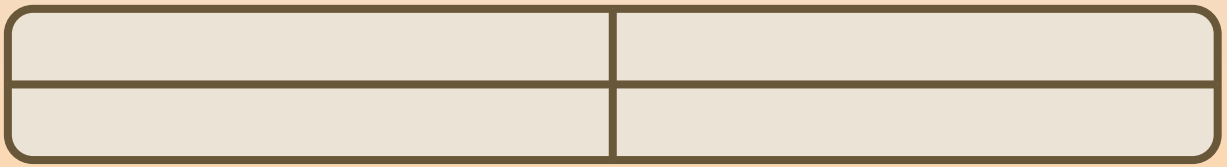
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ

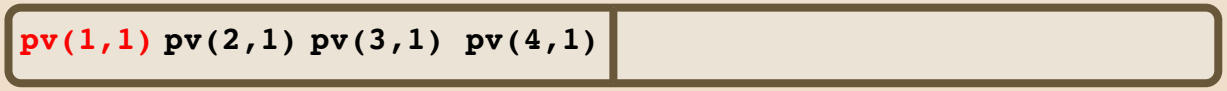


キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

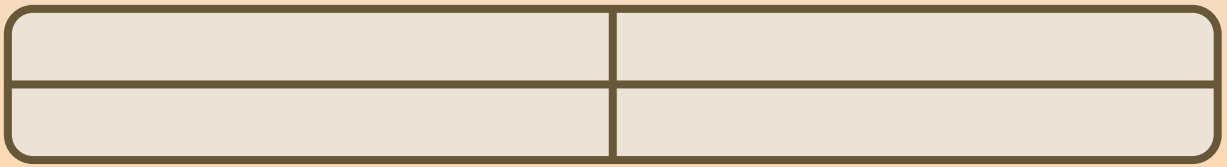
Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```



キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ

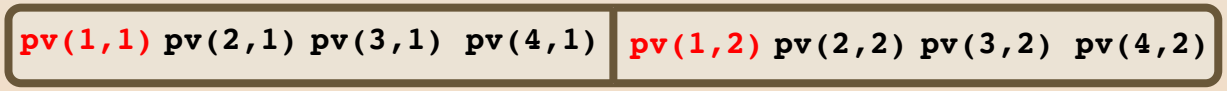


キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

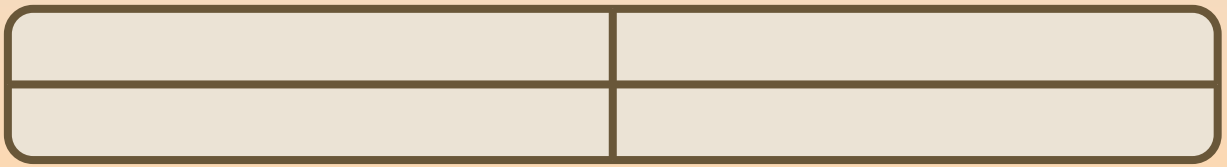
Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```



キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ

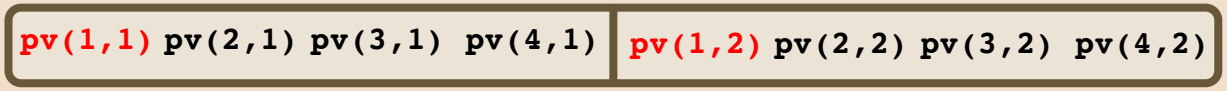


キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

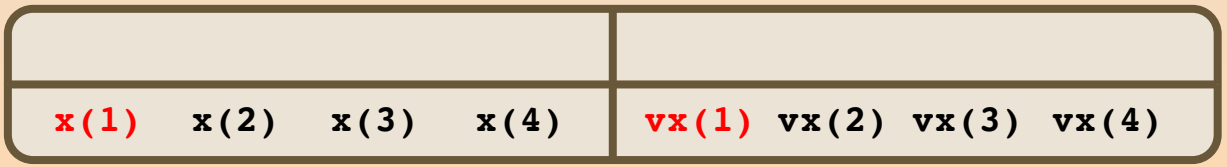
Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```



キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1)	pv(2,1)	pv(3,1)	pv(4,1)	pv(1,2)	pv(2,2)	pv(3,2)	pv(4,2)

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ

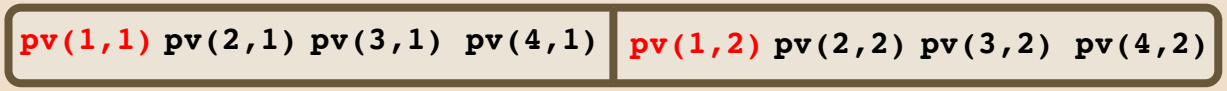
y(1)	y(2)	y(3)	y(4)	vy(1)	vy(2)	vy(3)	vy(4)
x(1)	x(2)	x(3)	x(4)	vx(1)	vx(2)	vx(3)	vx(4)

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

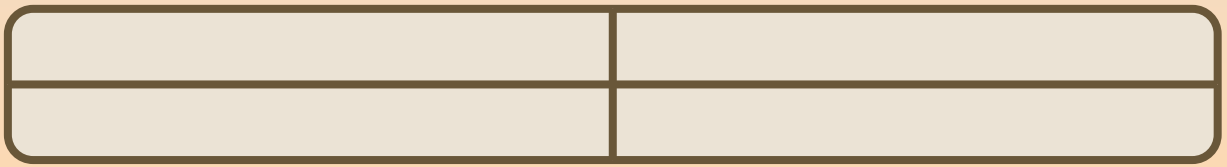
Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```



キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ

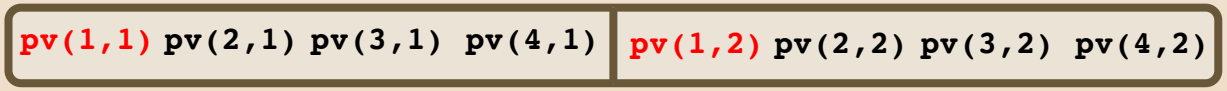


キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```



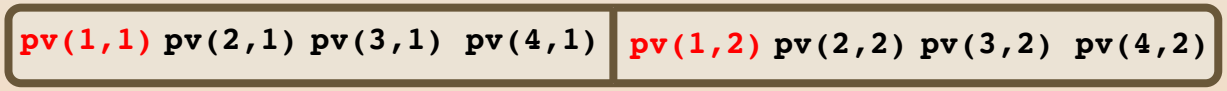
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```



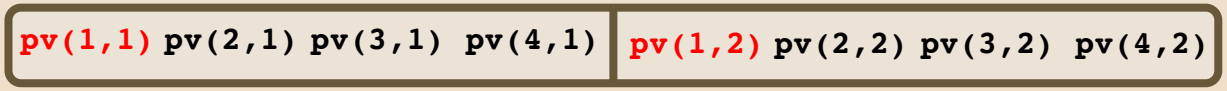
キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ

キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

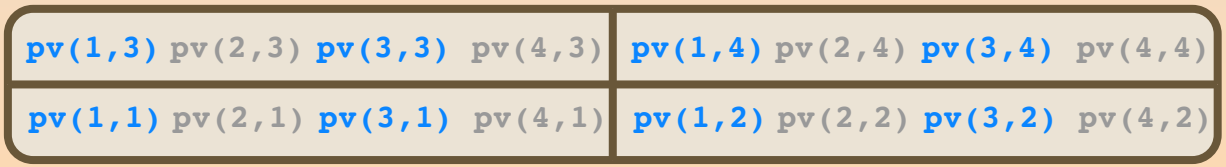
Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
! x(i)=pv(1,i),vx(i)=pv(2,i)
! y(i)=pv(3,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```



キャッシュライン数：2，ラインサイズ：16B，ライン当たり：一つ



キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

Array of Structureの書き方

- 2次元配列を使った書き方は，わかりにくい．
  - 構造体を使う
  - プリプロセッサ機能を使う

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

```
parameter( N=10000000 )
type particle
  real :: x, vx, y, vy
end type
type(particle) :: pv(N)
do i = 1, N
   pv(i)%x = pv(i)%x + pv(i)%vx
   pv(i)%y = pv(i)%y + pv(i)%vy
end do
```

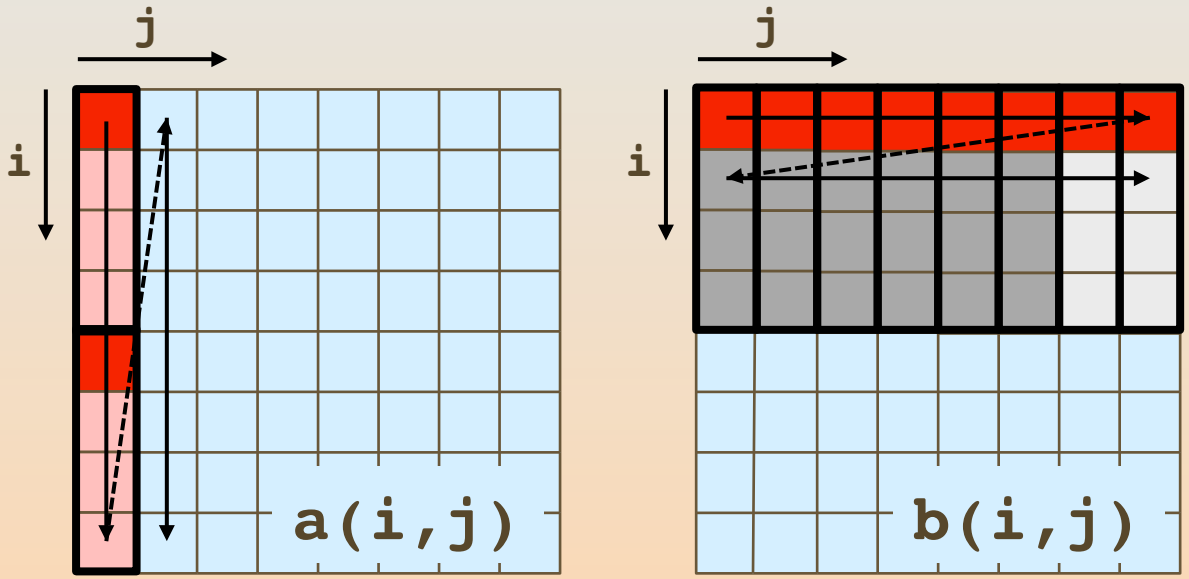
```
parameter( N=10000000 )
#define x(i) pv(1,i)
#define y(i) pv(3,i)
#define vx(i) pv(2,i)
#define vy(i) pv(4,i)
real pv(4,N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
#define x(i) pv(i)%x
#define y(i) pv(i)%y
#define vx(i) pv(i)%vx
#define vy(i) pv(i)%vy
type particle
  real :: x, vx, y, vy
end type
type(particle) :: pv(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

2次元配列の葛藤

- ループ順序を変えても,どちらかの2次元配列が連続アクセスにならない場合がある.

```
do j = 1, n
  do i = 1, n
    a(i,j) = a(i,j) + b(j,i)
  end do
end do
```

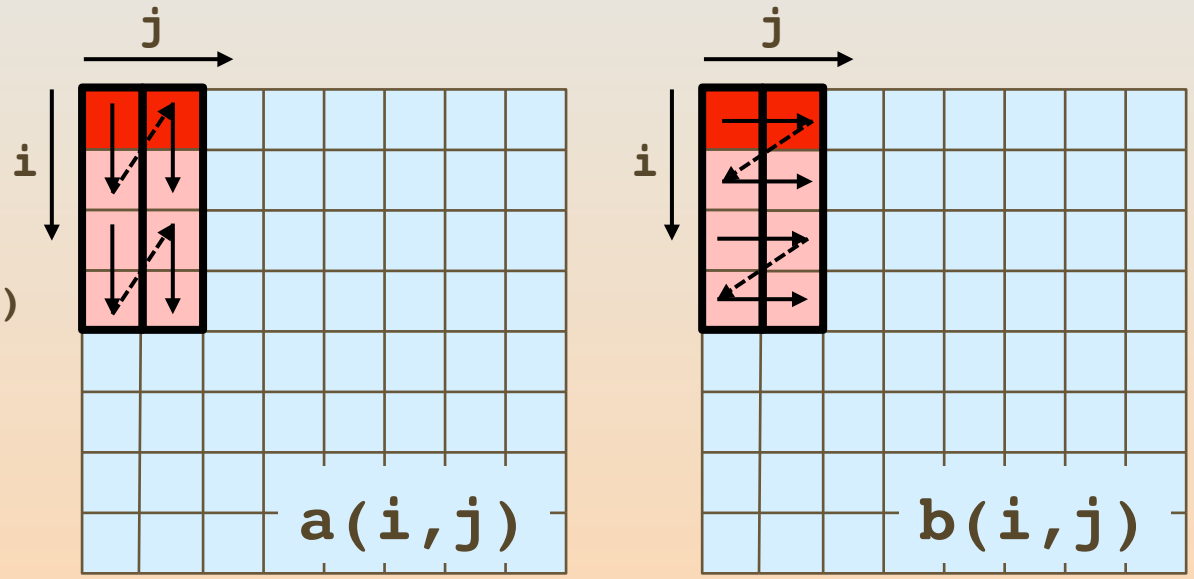


キャッシュライン数：2，ラインサイズ：16B，ライン当たり：二つ

2次元配列のブロック化

- 2次元配列をブロック化することにより,キャッシュミスを削減できる.
  - 最適なブロックの大きさは,キャッシュ容量,キャッシュライン数,ラインサイズ,ライン当たりの個数によって異なる.

```
lb = 2
do jb = 1, n, lb
  do ib = 1, n, lb
    do j = jb, jb+lb-1
      do i = ib, ib+lb-1
        a(i,j) = a(i,j) + b(j,i)
      end do
    end do
  end do
end do
```



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

*ループ回数がブロック数で割りきれないときに注意！

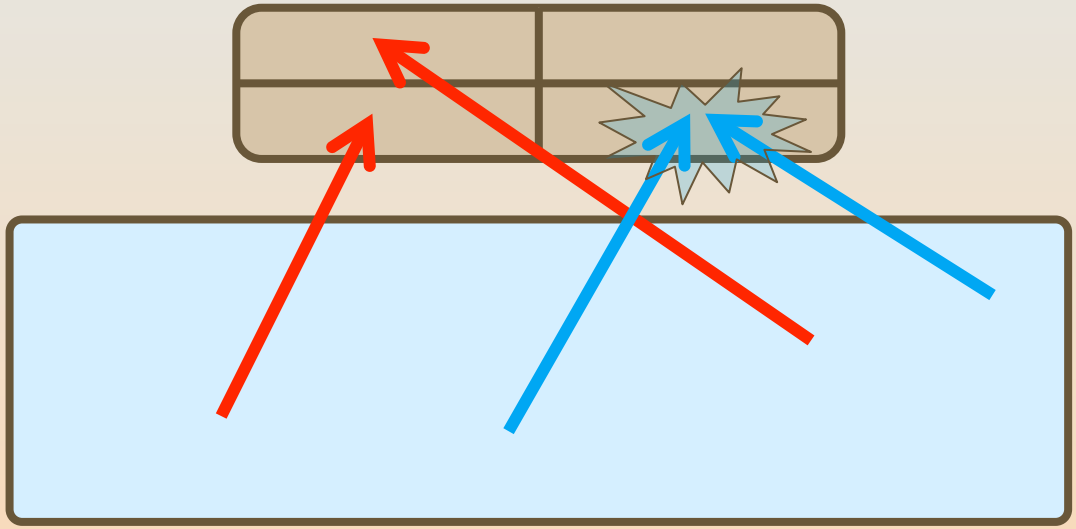
キャッシュコンフリクト

- 連続アクセスしている場合でも,キャッシュミスが起こる場合がある.
  - ループ内に変数が多い場合は,すべての変数がキャッシュに載らないかもしれない.
  - キャッシュラインが衝突する場合は,同時にキャッシュに載らない.
  - 起こるか起こらないかは,キャッシュ容量,キャッシュライン数等による.

```
do i = 1, n
  a(i) = a(i)+b(i)+c(i)+d(i)
end do
```

⇩ ただし，ループオーバーヘッドとの兼ね合い.

```
do i = 1, n
  a(i) = a(i)+b(i)
end do
do i = 1, n
  a(i) = a(i)+c(i)
end do
do i = 1, n
  a(i) = a(i)+d(i)
end do
```



```
dimension a(1024,1024), b(1024,1024)
```

⇩ ラインサイズだけずらす.

```
dimension a(1028,1024), b(1028,1024)
```

Formatted vs. Unformatted IO

- 書式付きファイルの入出力は,非常に遅いので,プログラム間でのデータの受け渡しには,書式なしファイルを用いる.
- ただし,異なるアーキテクチャーのコンピュータ間でデータを受け渡すときは,以下に注意すること.
  - Big Endian (SPARC, POWER) vs. Little Endian (Intel)
  - 4バイトヘッダー vs. 8バイトヘッダー(超大レコード)
  - バイトカウント vs. ワードカウント(現在では,ほとんどない)
  - ほとんどの場合,コンパイラオプションや環境変数で対応できる.