チューニング入門

2011年度冬学期

中島研吾

科学技術計算Ⅱ（並列有限要素法）（4820-1028）

- チューニングとは
- ベクトルプロセッサとスカラープロセッサ
- チューニング例：スカラープロセッサ
- メモリ性能，T2K, numactl

チューニングとは

- チューニングの目的
  - 計算時間の削減
  - 最適化
- チューニングの実施方法
  - アルゴリズムの変更
  - ハードウェアに応じた修正，最適化
  - もちろん計算結果に影響を及ぼすようなものであってはならない
    - またはその効果を承知していなければならない

「チューニング」との向き合い方・・・

- そもそもどういうタイミングでチューニングをやるのか?
  - プログラムができてしまってから，チューニングをするのは（誰がやっても）大変な作業
- 最初に作るときから，ある程度，チューニングに関することを考慮するべきである
  - いくつかの心得
- 「わかりやすい」，「読みやすい」プログラムを作ること
  - バグの出にくいプログラムを作る・・・ということでもある
- チューニングされたライブラリを使う
- 良い並列プログラム＝良いシリアルプログラム
- チューニング⇒高速⇒研究サイクルの効率化・・・

チューニング：参考文献

- スカラープロセッサ
  - 寒川「RISC超高速化プログラミング技法」，共立出版，1995.
  - Dowd（久良知訳）「ハイ・パフォーマンス・コンピューティング-RISCワークステーションで最高のパフォーマンスを引き出すための方法」，トムソン，1994.
  - Goedecker, Hoisie “Performance Optimization for Numerically Intensive Codes”, SIAM, 2001.
- 自動チューニング
  - 片桐「ソフトウェア自動チューニング」，慧文社，2004.
- ベクトルプロセッサ
  - 長島，田中「スーパーコンピュータ」，オーム社，1992.
  - 奥田，中島「並列有限要素解析」，培風館，2004.

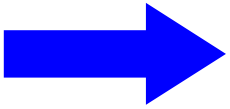
「チューニング」の心得

- メモリアクセスパターンに配慮
- 最内側ループでサブルーチン，関数コールは避ける。
  - 最近のコンパイラでは「インライン展開」が可能であるが，うまく働かない場合もある。
  - IF文もできるだけ避ける。
- 多重DOループを避ける。
- 除算，組み込み関数の多用を避ける。
- 無駄な計算はなるべく排除する。
  - 記憶できるところは記憶しておく。
  - メモリ容量との兼ね合い。
- **H/W，コンパイラ依存性は残念ながら大きい**
  - **オプション，ディレクティヴ：実際に試して速い手法を採用するべき**
  - **今日の話は一般的なもの**

例：多重DOループ

- 1つのDOループに到達するごとに，ループカウンタの初期設定というオーバーヘッドが生じる。
  - 例えば，下左の例では，最内ループに1,000,000回到達するため1,000,000回のオーバーヘッドが生じる。
  - 右のように展開してしまった方が良い。

```
real*8 AMAT(3,1000000)
. . .
do j= 1, 1000000
  do i= 1, 3
    A(i,j)= A(i,j) + 1.0
  enddo
enddo
. . .
```



```
real*8 AMAT(3,1000000)
. . .
do j= 1, 1000000
  A(1,j)= A(1,j) + 1.0
  A(2,j)= A(2,j) + 1.0
  A(3,j)= A(3,j) + 1.0
enddo
. . .
```

簡単に実施できること:パフォーマンス測定

- 「time」コマンド
- 「timer」サブルーチンによる測定
- 「プロファイリング（profiling）」ツールの使用
  - ホットスポットの特定
  - gprof（UNIX）
  - 他にも処理系，コンパイラに応じたプロファイラがある
    - pgprof: PGIコンパイラ
    - Vtune: Intel
    - T2Kも日立製の性能モニタがある（2010年11月より公開）

ファイルコピー：FORTRANのみ

簡単なプログラムなので，自分で作ってください

```
>$ cd <$T-fem2>

FORTRAN
>$ cp /home/t00000/fem2/s3.tar .
>$ tar xvf s3.tar

直下に「/mpi/S3」というディレクトリができている。
<$T-fem2>/mpi/S3を<$T-S3>と呼ぶ。
```

gprof on T2K（日立コンパイラ）

- サブルーチンごとにCPU時間，呼び出し回数が表示される。
  - どのサブルーチンから何回コールされ，どのサブルーチンを何回コールしたかが表示される。
- 「gmon.out」が生成される。

```
$> cd <$T-S3>

$> f90 –Oss -noparallel -G test.f
$> ./a.out
$> ls gmon.out
   gmon.out

$> gprof > out.lst
```

gprof の出力例(最初の部分)
コンパイルオプション -O3の場合

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 97.30      0.36     0.36        1   360.00   360.00  cg_
  2.70      0.37     0.01        1    10.00   370.00  MAIN_
  0.00      0.37     0.00    38424     0.00     0.00  csearch_
  0.00      0.37     0.00     6400     0.00     0.00  jacobi_
```

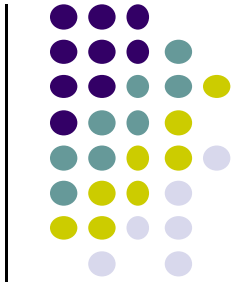
gprof on ECCS2008

- サブルーチンごとにCPU時間，呼び出し回数が表示される。
  - どのサブルーチンから何回コールされ，どのサブルーチンを何回コールしたかが表示される。
- 「gmon.out」が生成される。

```
$> g95 -O3 -pg test.f
$> ./a.out
$> ls gmon.out
   gmon.out

$> gprof > out.lst
```

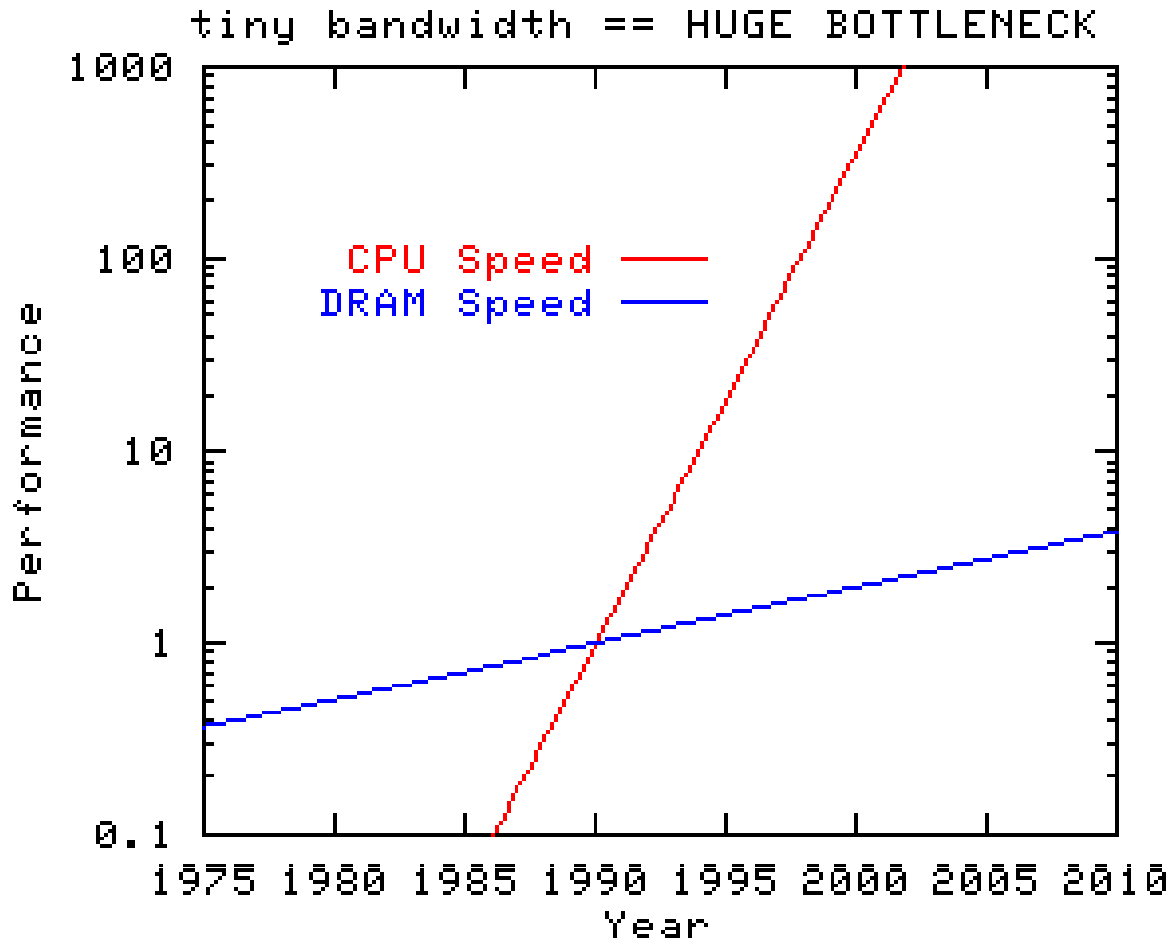
- チューニングとは
- **ベクトルプロセッサとスカラープロセッサ**
- チューニング例：スカラープロセッサ
- メモリ性能，T2K, numactl

スカラー／ベクトルプロセッサ



- スカラープロセッサ
  - クロック数とメモリバンド幅のギャップ
    - 改善はされつつある　→　マルチコア，メニーコア
  - 低い対ピーク性能比
    - 例：IBM Power-3，Power-4，FEM型アプリケーション → 5-8 %
- ベクトルプロセッサ
  - 高い対ピーク性能比
    - 例：地球シミュレータ，FEM型アプリケーション → >35 %
  - そのためには・・・
    - ベクトルプロセッサ用チューニング
    - 充分長いベクトル長（問題サイズ）
  - 比較的単純な問題に適している

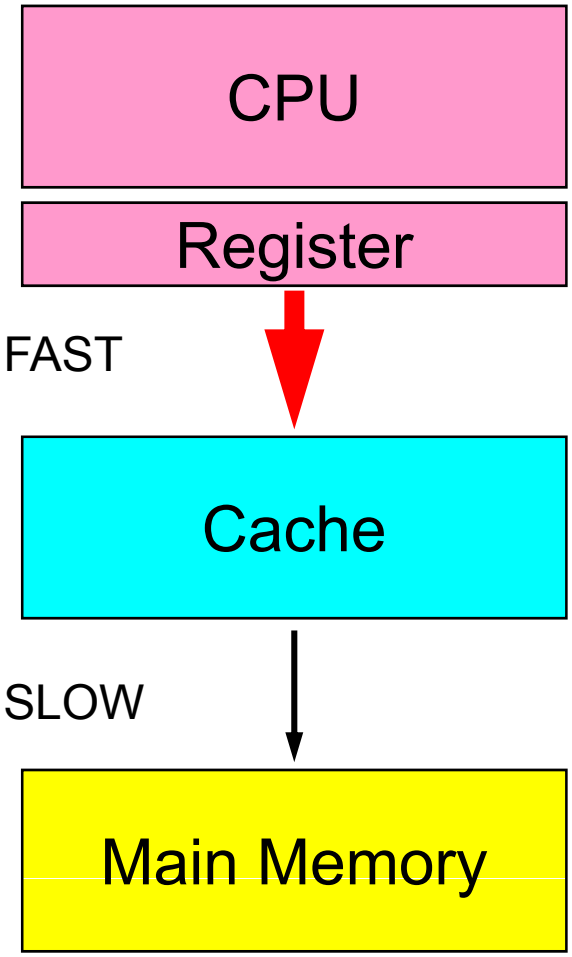
マイクロプロセッサの動向
CPU性能，メモリバンド幅のギャップ



http://www.streambench.org/

スカラープロセッサ

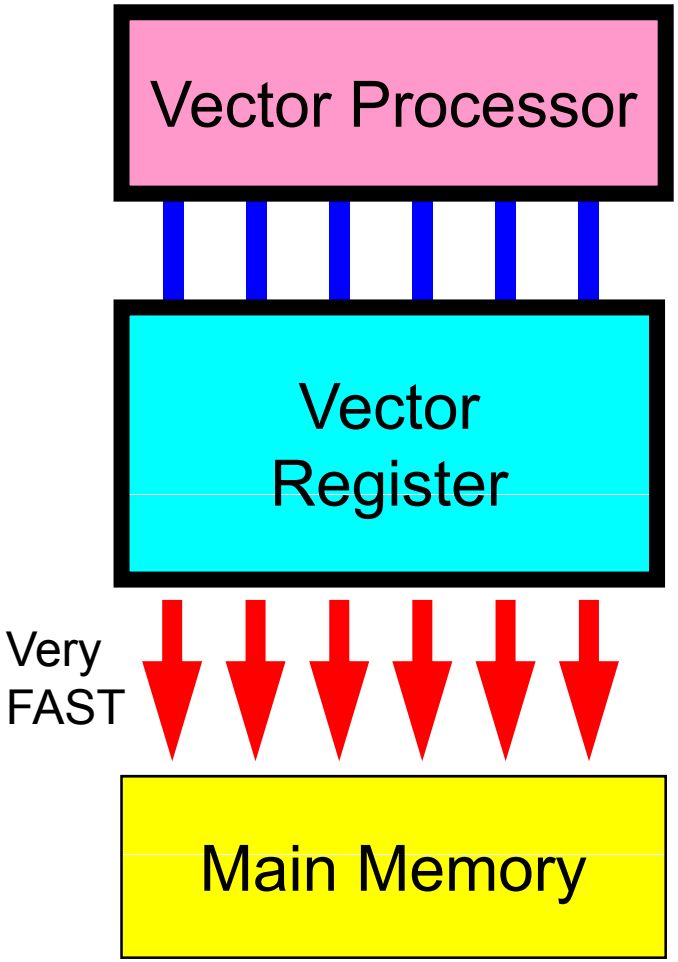
CPU–キャッシュ–メモリの階層構造



小容量 (MB)
高価
大きい (数億以上のトランジスタ)
命令キャッシュ（プログラム），データキャッシュ
多階層：L1,L2,L3

大容量 (GB)
廉価

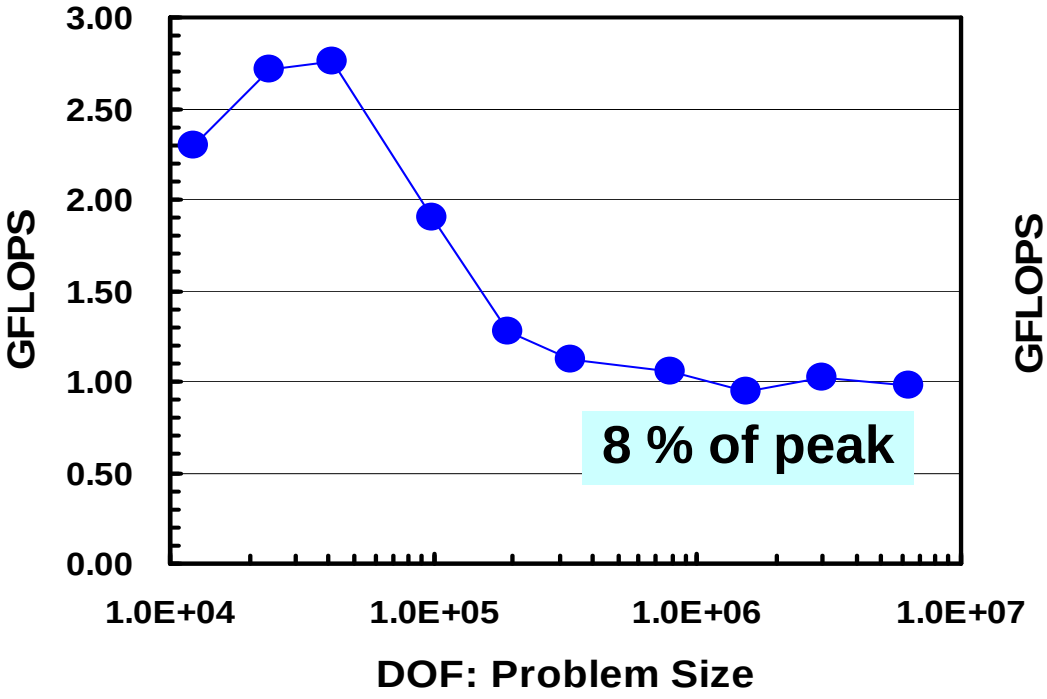
ベクトルプロセッサ
ベクトルレジスタと高速メモリ

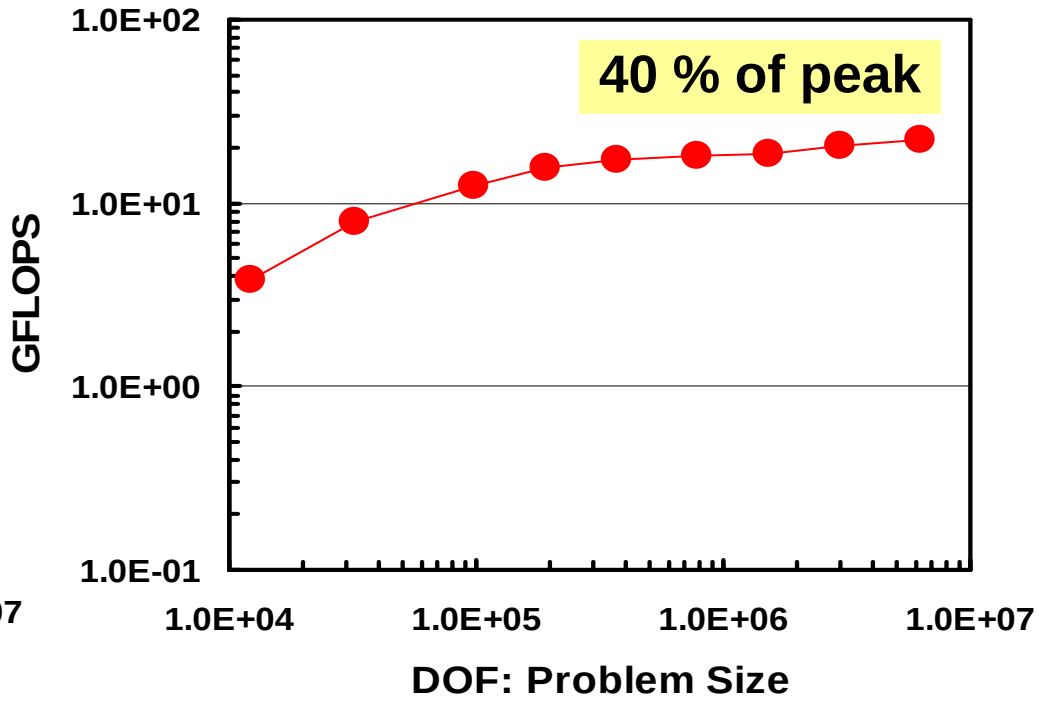


- 単純構造のDOループの並列処理
- 単純，大規模な演算に適している

```
do i= 1, N
  A(i)= B(i) + C(i)
enddo
```

典型的な挙動

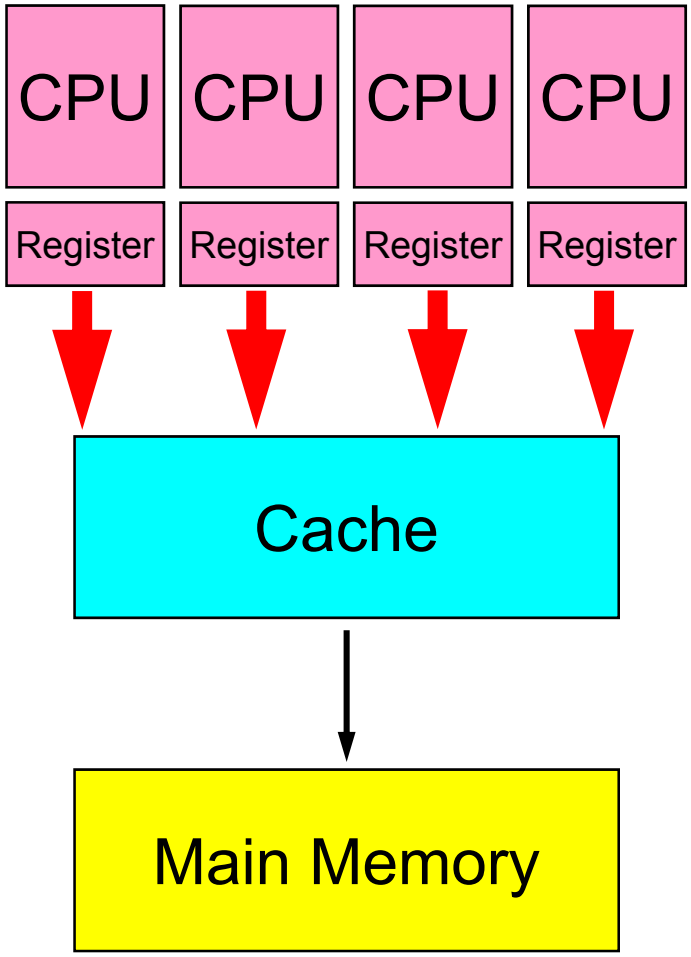


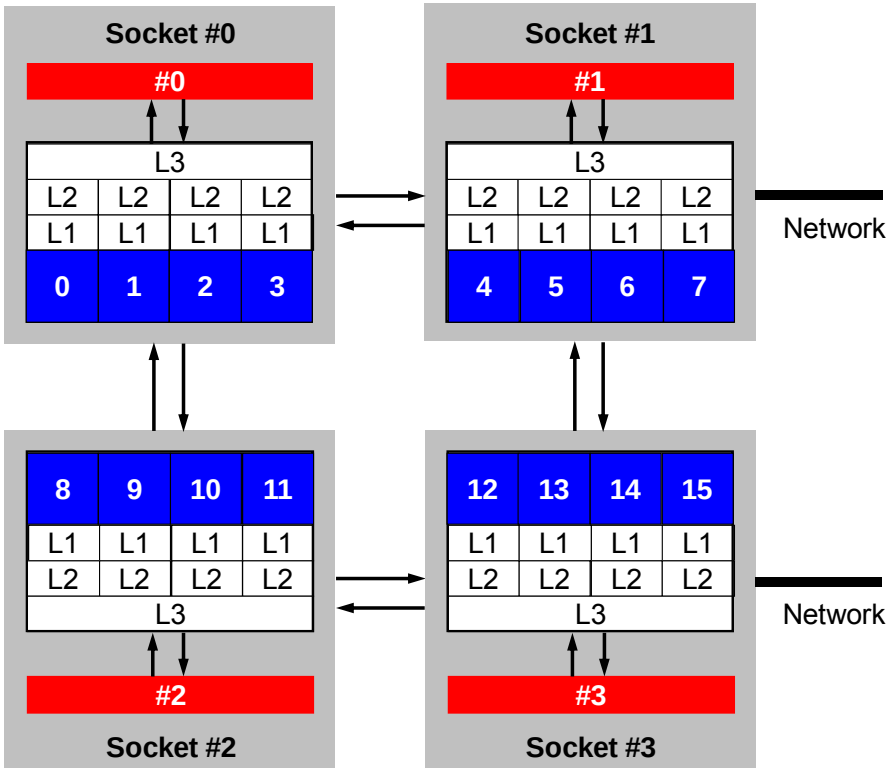


IBM-SP3:
問題サイズが小さい場合はキャッシュの影響のため性能が良い

Earth Simulator:
大規模な問題ほどベクトル長が長くなり，性能が高い

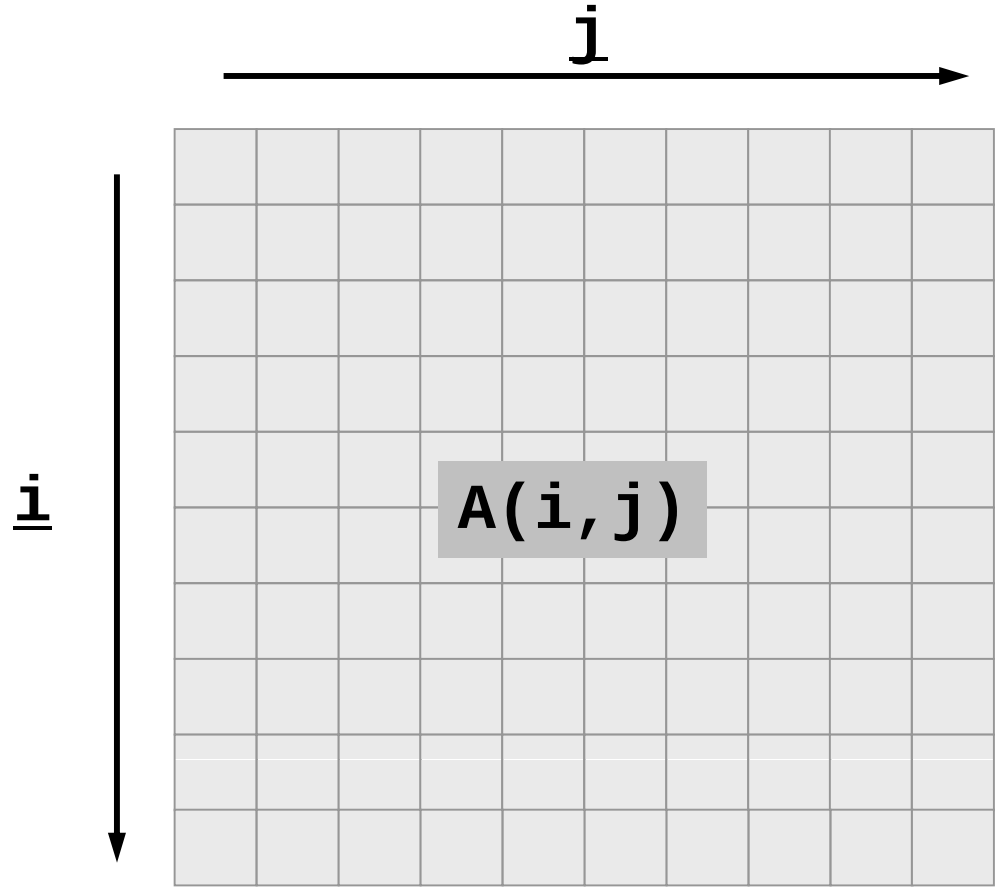
マルチコア，メニーコア
複数のコアでメモリ，キャッシュを共有
→競合，性能低下





プロセッサに応じたチューニング

- メモリ参照の最適化･･･に尽きる



プロセッサに応じたチューニング（続き）

- ベクトルプロセッサ
  - ループ長を大きくとる。
- スカラープロセッサ
  - キャッシュを有効利用，細切れにしたデータを扱う。
  - PCクラスタなどでは，キャッシュサイズを大きくできない一方で，メモリレイテンシ（立ち上がりオーバーヘッド）の減少，メモリバンド幅の増加の傾向。
- 共通事項
  - メモリアクセスの連続性
  - 局所性
  - 計算順序の変更によって計算結果が変わる可能性について注意すること

- チューニングとは
- ベクトルプロセッサとスカラープロセッサ
- **チューニング例：スカラープロセッサ**
- メモリ性能，T2K, numactl

スカラープロセッサの代表的なチューニング

- ループアンローリング
  - ループのオーバーヘッドの削減
  - ロード・ストアの削減
- ブロック化
  - キャッシュミスの削減

BLAS: Basic Linear Algebra Subprograms

- ベクトル，（密）行列の基本的な演算に関するライブラリAPI

- レベル1：ベクトル演算（内積，DAXPY）
- レベル2：（行列×ベクトル）積
- レベル3：（行列×行列）積

- LINPACK等で利用
  - DGEMM：行列×行列（レベル3）

Scalar

ループアンローリング

ロード・ストアの削減（1/4）

- ループ処理に対する演算の割合が増す。

<$T-S3>/t2.f

```
N= 10000

do j= 1, N
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
  enddo
enddo

do j= 1, N-1, 2
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
    A(i)= A(i) + B(i)*C(i,j+1)
  enddo
enddo

do j= 1, N-3, 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)
  enddo
enddo
```

```
$> cd <$T-S3>
$> mpif90 -O3 t2.f

$> <modify “go.sh”>

$> qsub go.sh （1プロセス）

     5.084839E-01
     3.539491E-01
     3.044438E-01
```

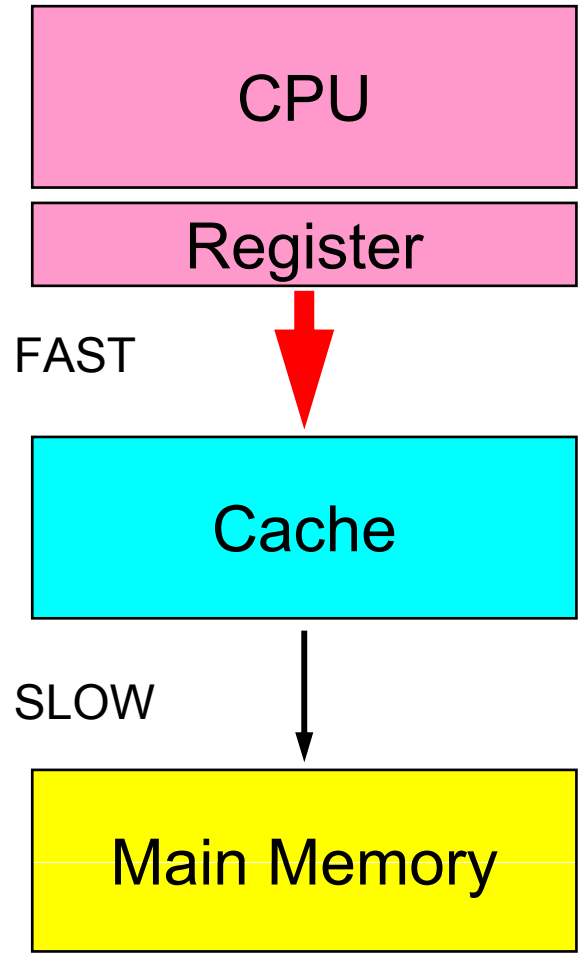
ジョブスクリプト:go.sh 1コア用

```
#@$-r S3-KN
#@$-q lecture
#@$-N 1
#@$-J T1
#@$-e err
#@$-o aaa.lst
#@$-lM 28GB
#@$-lT 00:05:00
#@$-s /bin/sh
#@$

cd $PBS_O_WORKDIR
mpirun numactl --localalloc ./a.out
```

ループアンローリング

ロード・ストアの削減（2/4）

- ロード：メモリ⇒キャッシュ⇒レジスタ
- ストア：ロードの逆
- ロード・ストアが少ないほど効率良い



ループアンローリング

ロード・ストアの削減（3/4）

```
do j= 1, N
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
   ストア  ロード   ロード ロード
  enddo
enddo
```

- A(i,j)に対して，各ループでロード・ストアが発生する。

Scalar

ループアンローリング

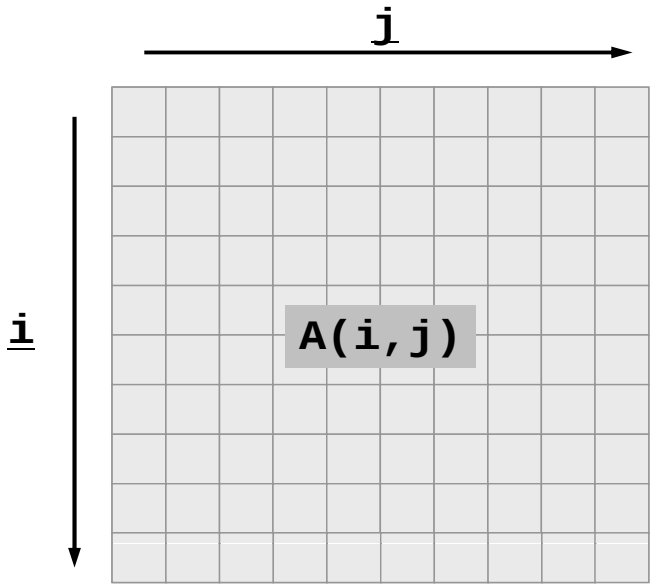
ロード・ストアの削減（4/4）

```
do j= 1, N-3, 4
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
          ロード  ロード ロード
    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)
    ストア
  enddo
enddo
```

- 同一ループ内で複数回表れている場合には，最初にロード，最後にストアが実施され，その間レジスターに保持される。
- 計算順序に注意。

ループ入替によるメモリ参照最適化(1/2)

```
TYPE-A
do i= 1, N
  do j= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo

TYPE-B
do j= 1, N
  do i= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```



- FORTRANでは，A(i,j)のアドレスは，A(1,1), A(2,1), A(3,1),…, A(N,1), A(1,2), A(2,2),…, A(1,N), A(2,N),…, A(N,N)のように並んでいる
  - Cは逆：A[0][0], A[0][1], A[0][2], …, A[N-1][0], A[N-1][1],…,A[N-1][N-1]
- この順番にアクセスしないと効率悪い(ベクトル，スカラーに共通)。

ループ入替によるメモリ参照最適化（2/2）

<$T-S3>/2d-1.f

```
TYPE-A
do i= 1, N
  do j= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo

TYPE-B
do j= 1, N
  do i= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

```
TYPE-A
for (j=0; j<N; j++){
  for (i=0; i<N; i++){
    A[i][j]= A[i][j] + B[i][j];
  }
}

TYPE-B
for (i=0; i<N; i++){
  for (j=0; j<N; j++){
    A[i][j]= A[i][j] + B[i][j];
  }
}
```

```
$> cd <$T-S3>
$> mpif90 -O3 2d-1.f
$> qsub go.sh
### N ###      512
WORSE      4.142785E-02
BETTER     1.610041E-03
### N ###     1024
WORSE      1.714511E-01
BETTER     6.197929E-03
### N ###     1536
WORSE      3.920140E-01
BETTER     1.383591E-02
### N ###     2048
WORSE      8.177490E-01
BETTER     2.445412E-02
### N ###     2560
WORSE      1.093015E+00
BETTER     3.825879E-02
### N ###     3072
WORSE      1.571926E+00
BETTER     5.488586E-02
### N ###     3584
WORSE      2.034765E+00
BETTER     7.465911E-02
### N ###     4096
WORSE      3.404434E+00
BETTER     9.754300E-02
```

ループ入替によるメモリ参照最適化(2/2)
-Ossでコンパイルすると差異が無くなる

```
TYPE-A
do i= 1, N
  do j= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo

TYPE-B
do j= 1, N
  do i= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

```
TYPE-A
for (j=0; j<N; j++){
  for (i=0; i<N; i++){
    A[i][j]= A[i][j] + B[i][j];
  }
}

TYPE-B
for (i=0; i<N; i++){
  for (j=0; j<N; j++){
    A[i][j]= A[i][j] + B[i][j];
  }
}
```

```
$> cd <$T-S3>
$> mpif90 -Oss –noparallel 2d-1.f
$> qsub go.sh
### N ###      512
WORSE      1.232147E-03
BETTER     1.234055E-03
### N ###     1024
WORSE      5.063057E-03
BETTER     5.065203E-03
### N ###     1536
WORSE      1.131892E-02
BETTER     1.131606E-02
### N ###     2048
WORSE      2.012396E-02
BETTER     2.005601E-02
### N ###     2560
WORSE      3.110313E-02
BETTER     3.127599E-02
### N ###     3072
WORSE      4.489899E-02
BETTER     4.506397E-02
### N ###     3584
WORSE      6.102514E-02
BETTER     6.143689E-02
### N ###     4096
WORSE      7.983208E-02
BETTER     7.957292E-02
```

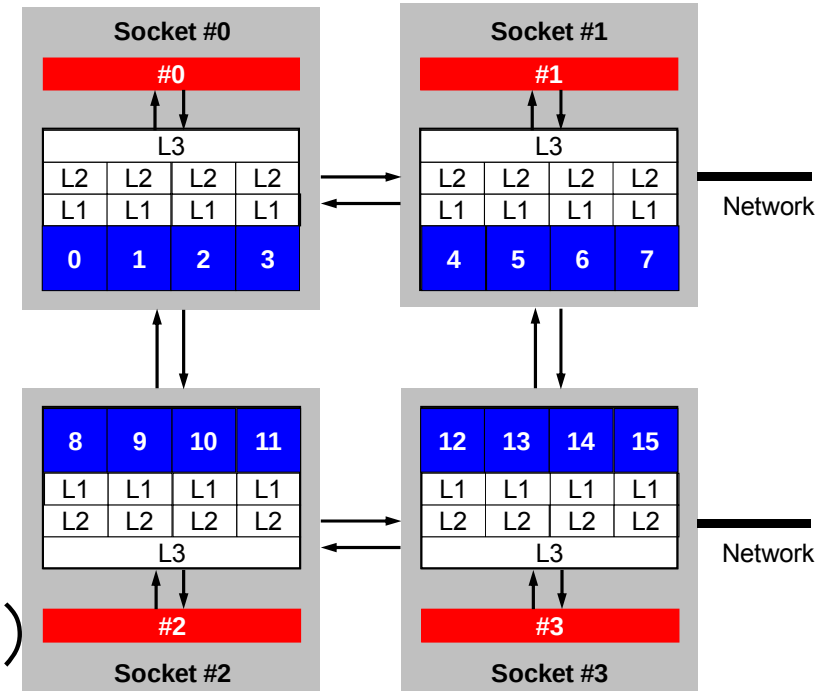
ブロック化によるキャッシュミス削減（1/7）

```
do i= 1, NN
  do j= 1, NN
    A(j,i)= A(j,i) + B(i,j)
  enddo
enddo
```

- 計算の処理の都合でこのような順番で計算せざるを得ない場合。

キャッシュの有効利用

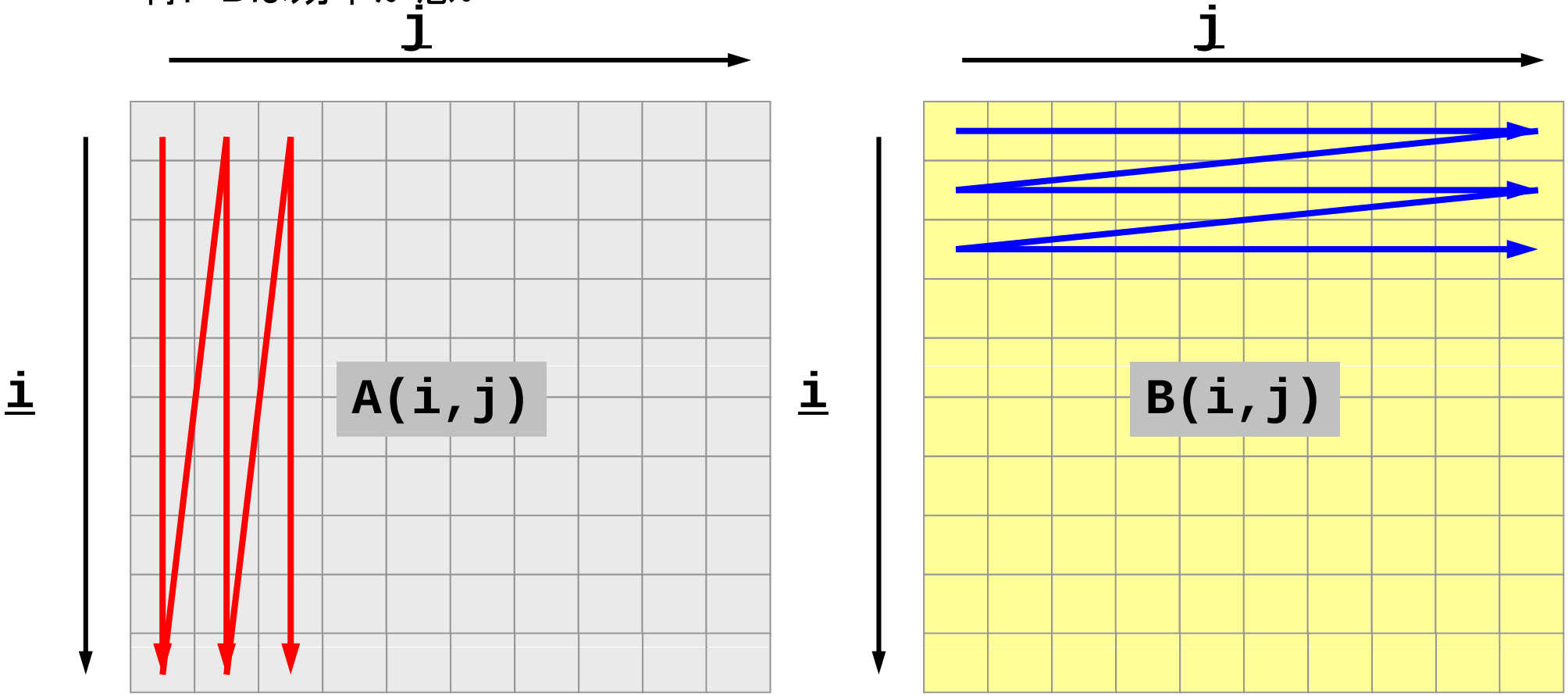
- キャッシュ：T2K
  - L1:64KB（命令，データ各），L2:512KB, L3:2,048KB
  - 実際は64byteの細かいキャッシュラインに分かれている。
  - キャッシュライン単位でメモリへのリクエストが行われる。



- TLB（Translate Lookaside Buffer）
  - アドレス変換バッファ
    - 仮想アドレスから実アドレスへの変換機能
  - TLB用のキャッシュ
    - 通常128×8 kbyte程度：リンク時に可変
  - 「キャッシュ」が有効に使われていればTLBミスも起こりにくい

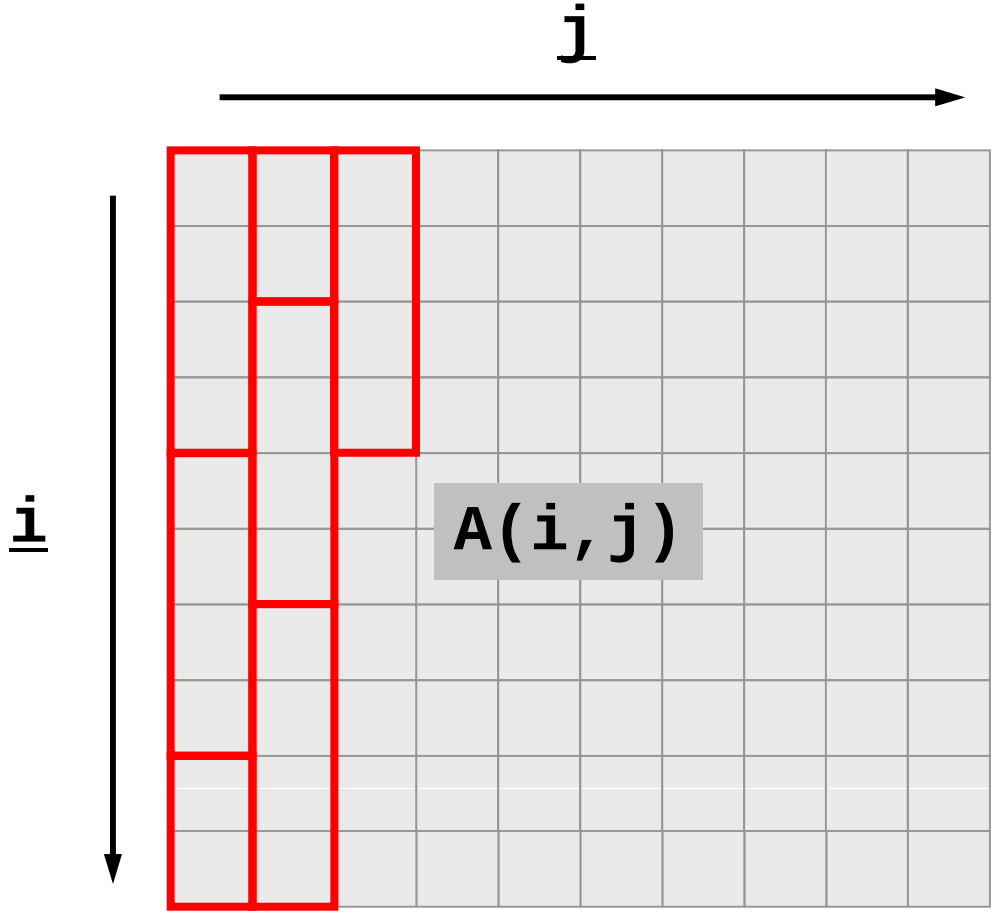
ブロック化によるキャッシュミス削減（2/7）

- A, Bでメモリアクセスパターンが相反
  - 特にBは効率が悪い



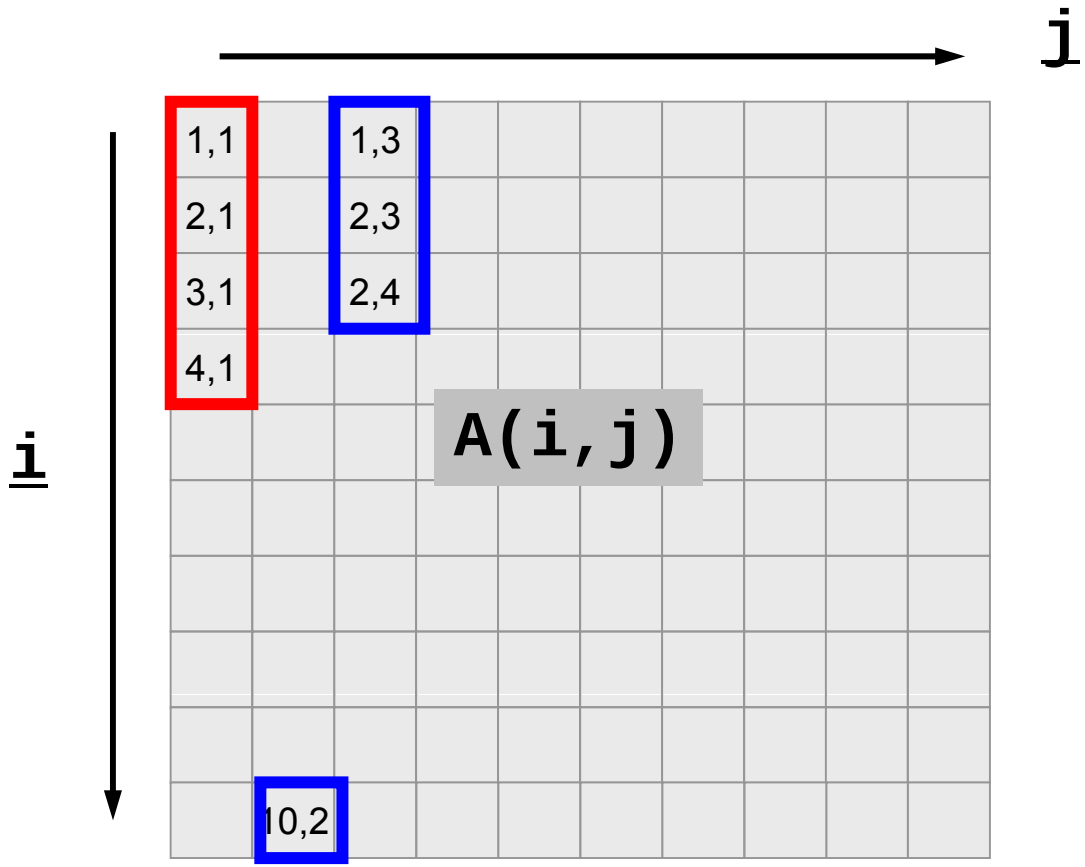
ブロック化によるキャッシュミス削減（3/7）

- 例えば，キャッシュラインサイズを4ワードとすると配列の値は以下のようにキャッシュに転送される。



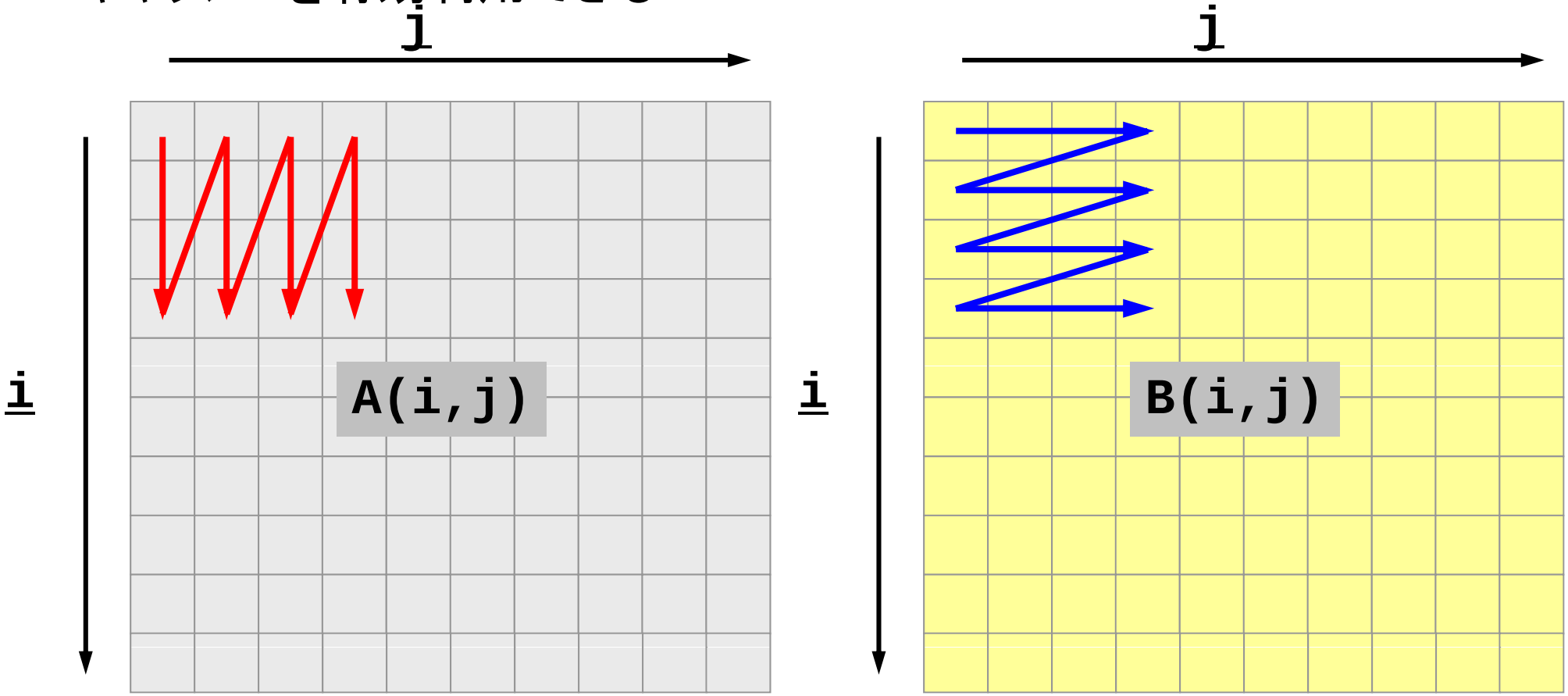
ブロック化によるキャッシュミス削減（4/7）

- したがって，A(1,1)をアクセスしたら，A(1,1), A(2,1), A(3,1), A(4,1)が，A(10,2)をアクセスしたらA(10,2), A(1,3), A(2,3), A(3,3)がそれぞれキャッシュ上にあるということになる。



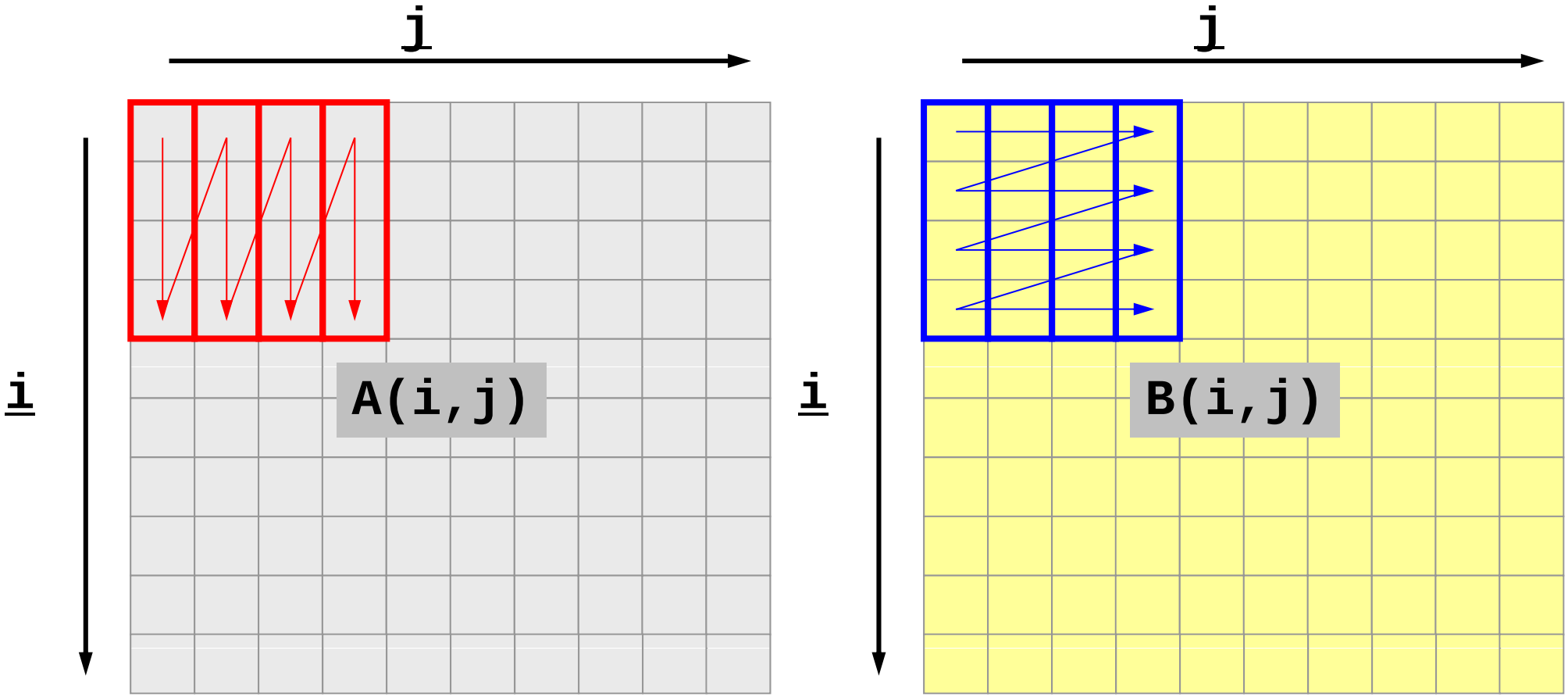
ブロック化によるキャッシュミス削減（5/7）

- したがって，以下のようなブロック型のパターンでアクセスすれば，キャッシュを有効利用できる



ブロック化によるキャッシュミス削減（6/7）

- □，□で囲んだ部分はキャッシュに載っている。



ブロック化によるキャッシュミス削減（7/7）

- 2×2ブロック

<$T-S3>/2d-2.f

```
do i= 1, NN
  do j= 1, NN
    A(j,i)= A(j,i) + B(i,j)
  enddo
enddo


do i= 1, NN-1, 2
  do j= 1, NN-1, 2
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j  ,i+1)= A(j  ,i+1) + B(i+1,j  )
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo

do i= 1, NN-1, 2
  do j= 1, NN/2, 2
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j  ,i+1)= A(j  ,i+1) + B(i+1,j  )
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo

do i= 1, NN-1, 2
  do j= NN/2+1, NN-1, 2
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j  ,i+1)= A(j  ,i+1) + B(i+1,j  )
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo
```

ループ分割もTLBミス，
キャッシュミス削減に
有効と言われている

```
$> cd <$T-S3>
$> mpif90 -O3 2d-2.f
$> qsub go.sh

### N ###    1024
BASIC    3.483510E-02
2x2      2.019596E-02
2x2-b    1.354480E-02
### N ###    1536
BASIC    6.352186E-02
2x2      4.553390E-02
2x2-b    3.850698E-02

...

### N ###    3072
BASIC    3.217320E-01
2x2      1.907461E-01
2x2-b    1.831641E-01
### N ###    3584
BASIC    3.632019E-01
2x2      2.601080E-01
2x2-b    2.465279E-01
### N ###    4096
BASIC    2.066922E+00
2x2      1.040879E+00
2x2-b    1.040897E+00
```

チューニング：まとめ

- スカラープロセッサ
- 密行列：BLAS

- 本講義で主として扱う疎行列の場合はもっと難しい（研究途上の課題）
  - しかし，基本的な考え方は変わらない
  - メモリアクセスの効率化

- 2d-2.f
  - コンパイルオプションを変えてみよ（-Oss -noparallel）

疎行列と密行列

```
do i= 1, N
  Y(i)= D(i)*X(i)
  do k= index(i-1)+1, index(i)
    Y(i)= Y(i) + AMAT(k)*X(item(k))
  enddo
enddo
```

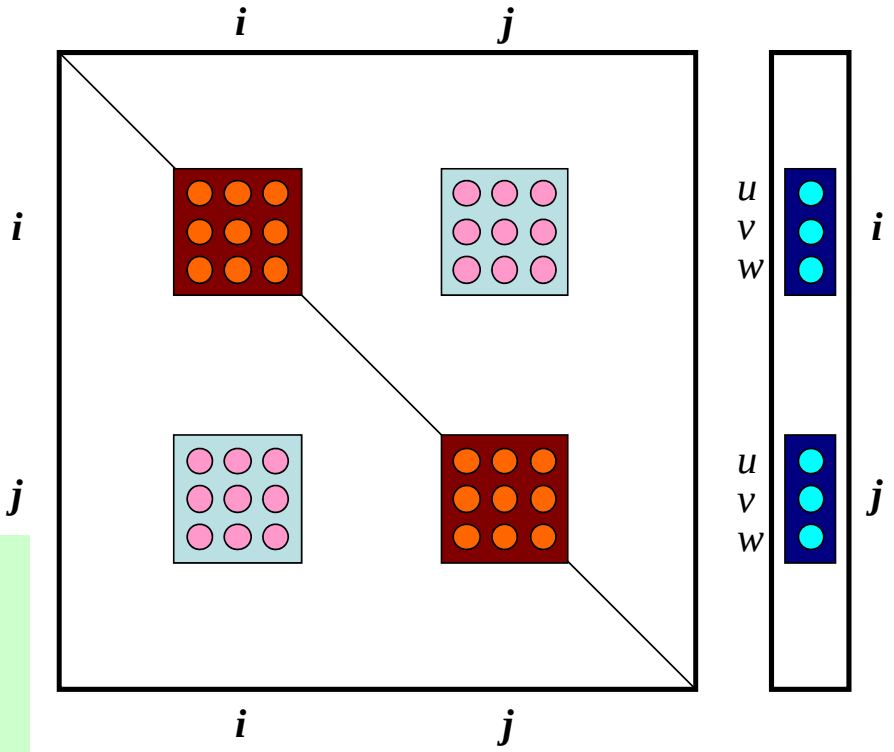
```
do j= 1, N
  do i= 1, N
    Y(j)= Y(j) + A(i,j)*X(i)
  enddo
enddo
```

- 右辺のX
  - 密行列：メモリ上で連続しているのでキャッシュを上手く使える
  - 疎行列：メモリ上で連続している保証が無いので，そのたびにメモリアクセスが必要になる場合がある
    - 疎行列計算の性能はメモリの性能が支配：memory-bound
- 疎行列における最も有効な対策：ブロック化
  - 並び替え：ブロック性の抽出
  - 物理的特徴の利用：1要素，1節点に複数自由度

Scalar

疎行列におけるブロック化

- 三次元弾性問題，有限要素法
  - 間接参照（メモリに負担）と計算の比が小さくなる
  - ベクトル，スカラー共に効く：2倍以上の性能
    - 連続領域，キャッシュに載る

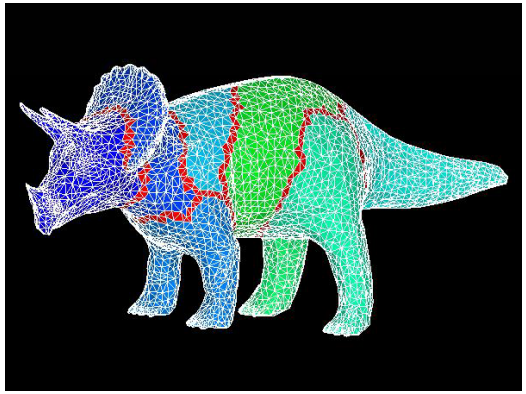



```
do i= 1, N
  X1= X(3*i-2)
  X2= X(3*i-1)
  X3= X(3*i)
  Y(3*i-2)= D(9*i-8)*X1+D(9*i-7)*X2+D(9*i-6)*X3
  Y(3*i-1)= D(9*i-5)*X1+D(9*i-4)*X2+D(9*i-3)*X3
  Y(3*I  )= D(9*i-2)*X1+D(9*i-1)*X2+D(9*I  )*X3
  do k= index(i-1)+1, index(i)
    kk= item(k)
    X1= X(3*kk-2)     u
    X2= X(3*kk-1)     v
    X3= X(3*kk)       w
    Y(3*i-2)= Y(3*i-2)+AMAT(9*k-8)*X1+AMAT(9*k-7)*X2 &
                                     +AMAT(9*k-6)*X3
    Y(3*i-1)= Y(3*i-1)+AMAT(9*k-5)*X1+AMAT(9*k-4)*X2 &
                                     +AMAT(9*k-3)*X3
    Y(3*I  )= Y(3*I  )+AMAT(9*k-2)*X1+AMAT(9*k-1)*X2 &
                                     +AMAT(9*k  )*X3
  enddo
enddo
```

```
do i= 1, 3*N
  Y(i)= D(i)*X(i)
  do k= index(i-1)+1, index(i)
    kk= item(k)
    Y(i)= Y(i) + AMAT(k)*X(kk)
  enddo
enddo
```

チューニング：まとめのちょっと余談

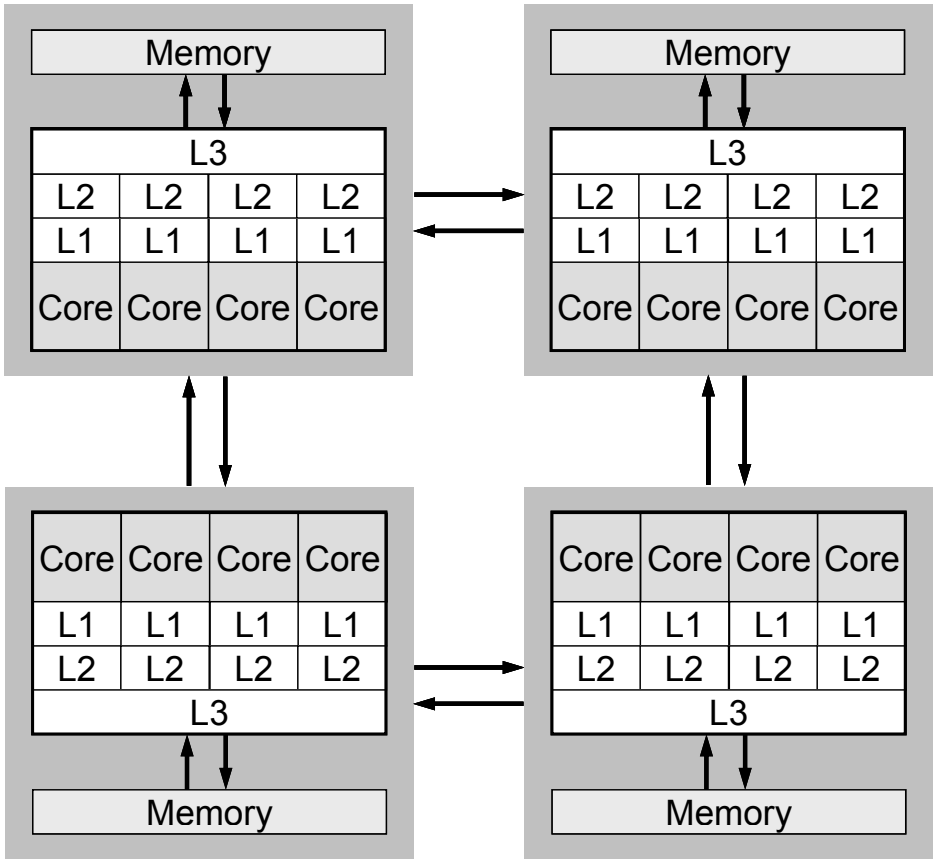
- 疎行列
  - データの並び方によってチューニングの戦略が変わる
  - データの並び方によってプログラムが変わる場合もある
- 密行列
  - 規則性
  - マシンパラメータ，問題サイズで性能は決まる
    - 他にもコンパイルオプションの影響などもある
    - 自動化（自動チューニング）の余地がある
  - ライブラリ
    - ATLAS（自動チューニング）
      - http://math-atlas.sourceforge.net/
    - GoToBLAS（手動チューニング）
      - 後藤和茂氏（現Microsoft）



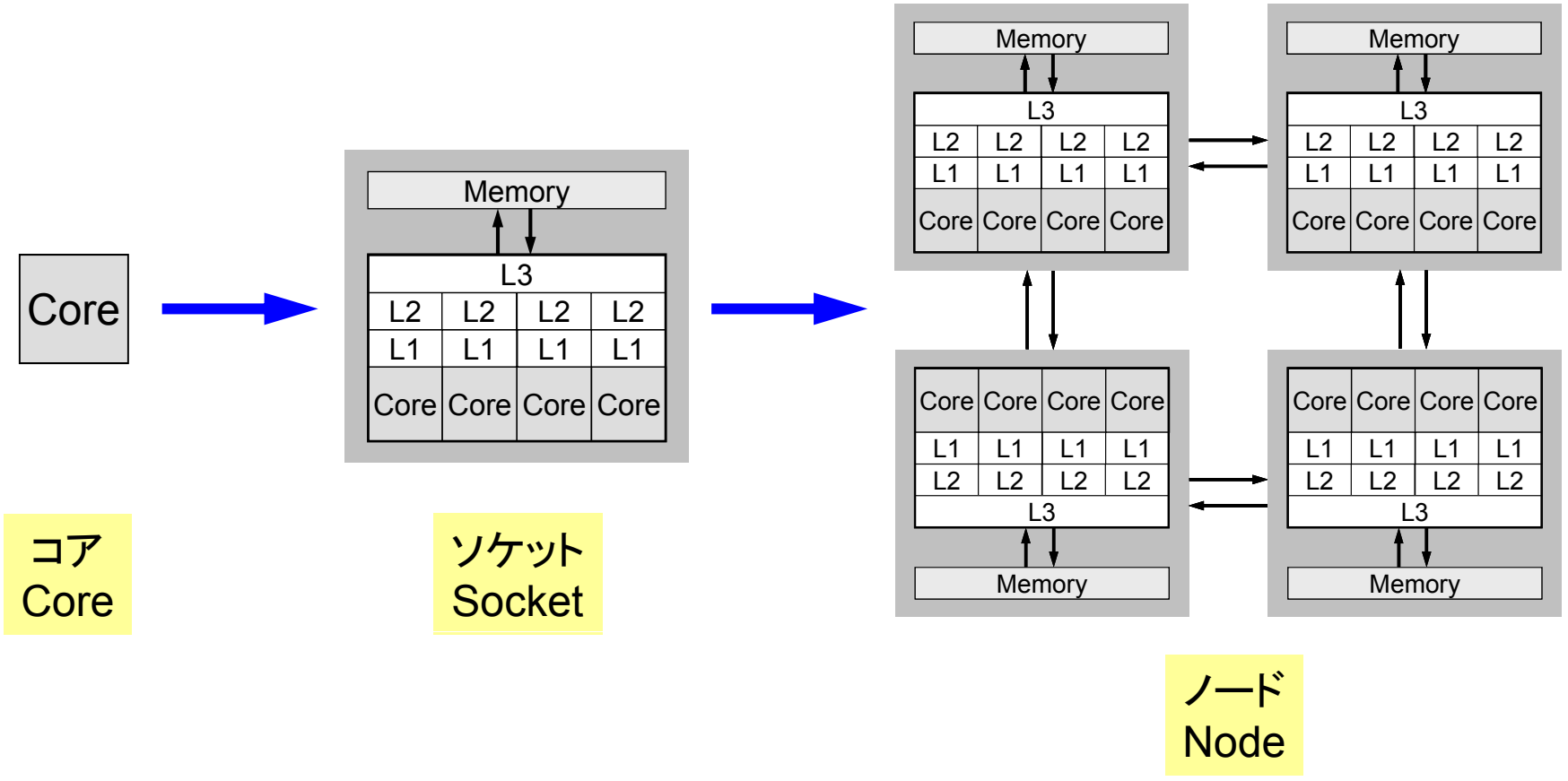
- チューニングとは
- ベクトルプロセッサとスカラープロセッサ
- チューニング例：スカラープロセッサ
- **メモリ性能，T2K, numactl**

Quad Core Opteron: ccNUMA Arch.

- AMD Quad Core Opteron 2.3GHz
  - Quad Coreのソケット×4 ⇒ 1ノード（16コア）
- 各ソケットがローカルにメモリを持っている
  - NUMA：Non-Uniform Memory Access
    - ローカルのメモリをアクセスして計算するようなプログラミング，データ配置，実行時制御（numactl）が必要
  - cc： cache-coherent
    - キャッシュ同期：Thread並列



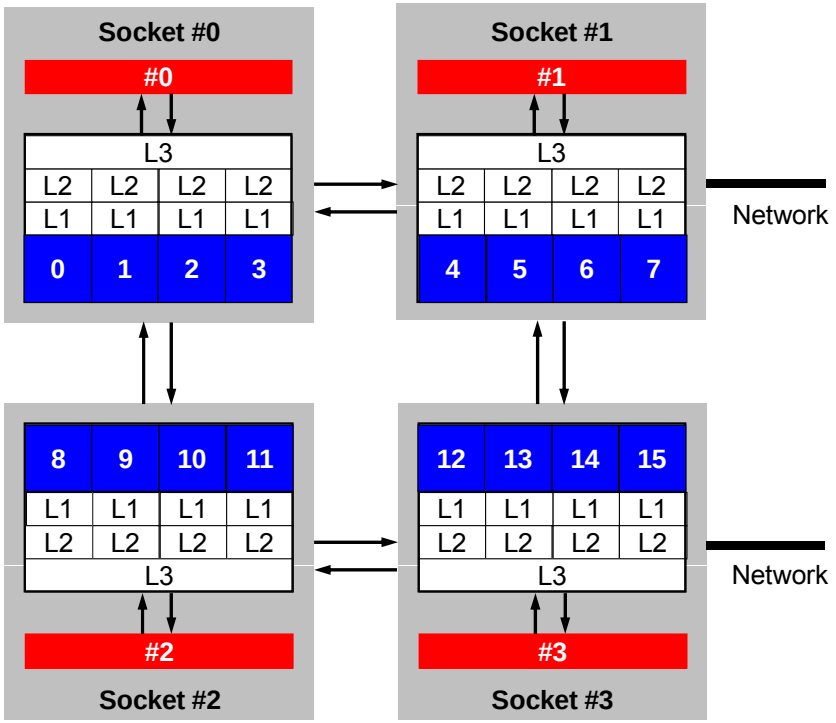
ccNUMA Architecture



numactl

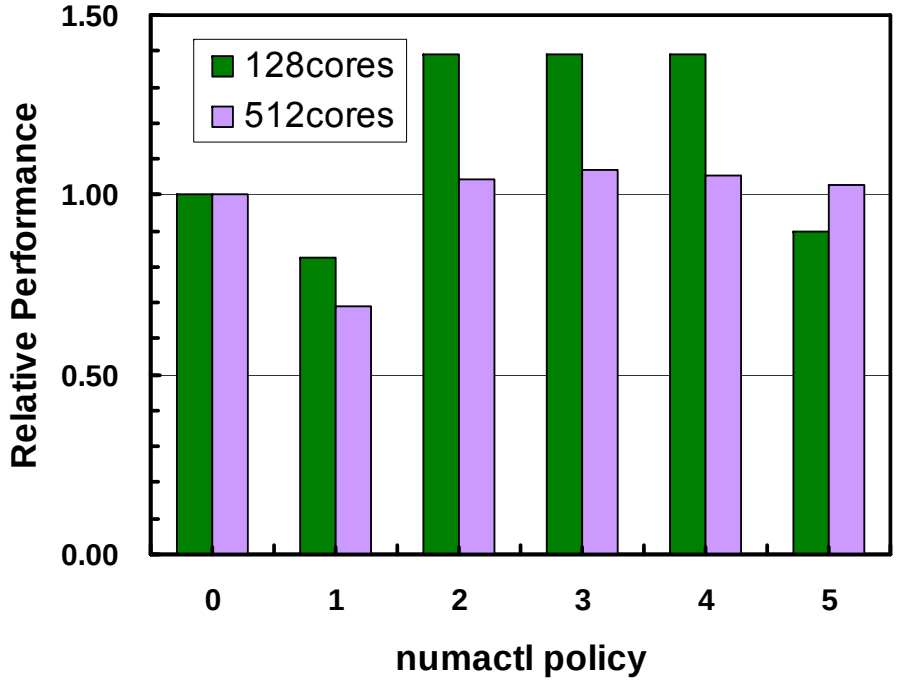
```
>$ numactl --show

policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15    コア
cpubind: 0 1 2 3                                      ソケット
nodebind: 0 1 2 3                                     ソケット
membind: 0 1 2 3                                      メモリ
```

- NUMA(Non Uniform Memory Access) 向けのメモリ割付のためのコマンドライン：Linuxでサポート
- 使うコアとメモリの関係指定：ローカルなメモリ上のデータを使うのが得策



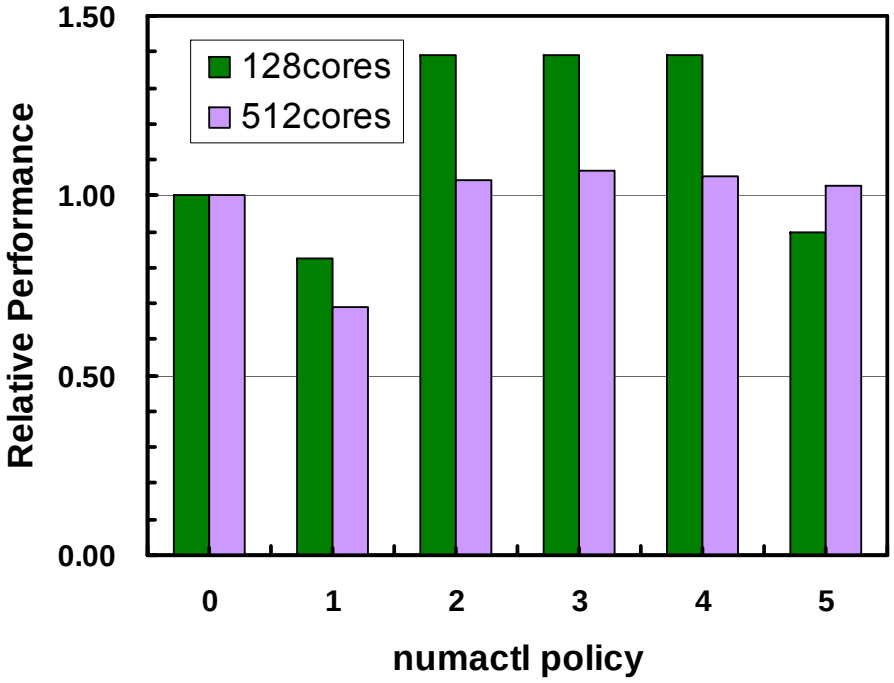
numactlの影響

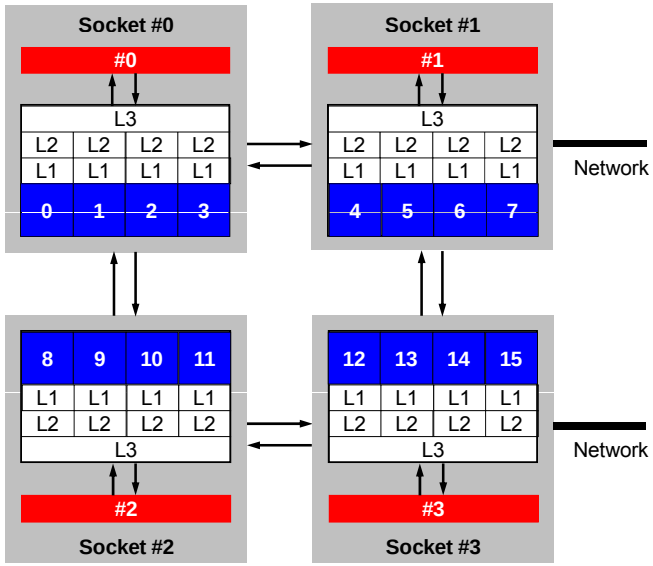
- T2K，有限要素法アプリケーション，Strong Scaling，Flat MPI
- 128コア：1コアあたりの問題サイズは512コアの4倍
  - メモリへの負担大
- Policy=0：何も指定しない場合
- 相対性能：大きいほど良い—128ノードで影響大




```
#@$-r HID-org
#@$-q h08nkl32
#@$-N 24
#@$-J T16
#@$-e err
#@$-o x384-40-1-a.lst
#@$-lM 27GB
#@$-lE 03:00:00
#@$-s /bin/sh
#@$
cd $PBS_O_WORKDIR
mpirun ./numarun.sh ./sol
exit
```

numarun.shの中身





Policy:1
```
#!/bin/bash
MYRANK=$MXMPI_ID MPIのプロセス番号
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --interleave=all $@
```

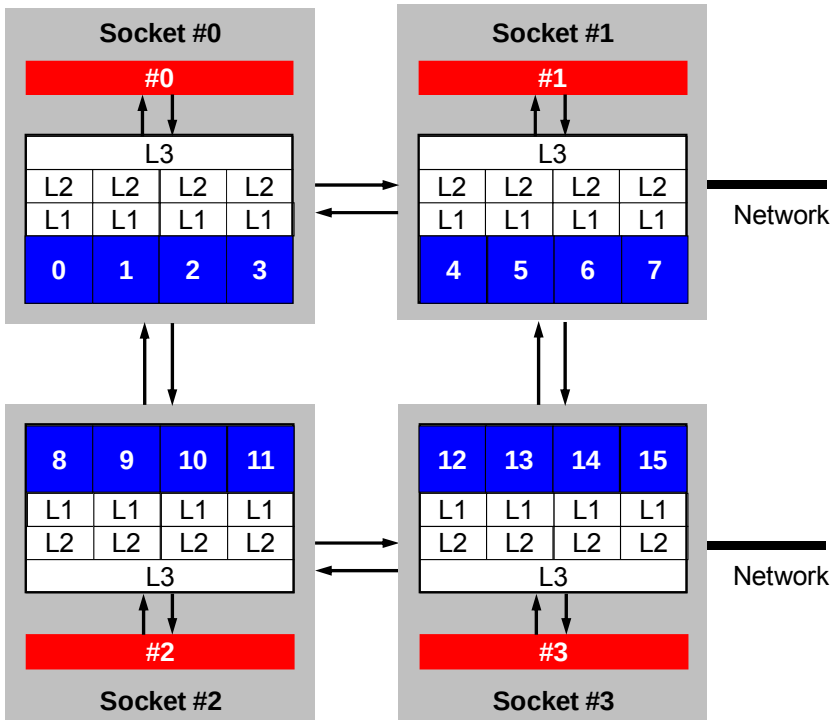
Policy:2
```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --interleave=$SOCKET $@
```

Policy:3
```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --membind=$SOCKET $@
```

Policy:4
```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --localalloc $@
```

Policy:5
```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --localalloc $@
```

Policy-3の割り当て

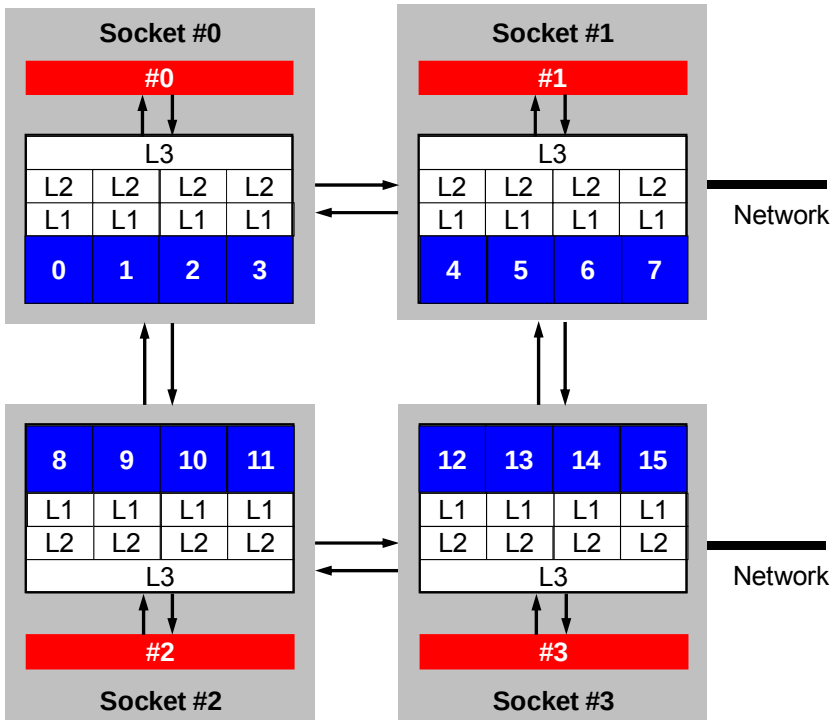


Policy:3

```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --membind=$SOCKET $@
```

MPI process	Socket	Memory	Core
0	0	0	
1	0	0	
2	0	0	
3	0	0	
4	1	1	
5	1	1	
6	1	1	
7	1	1	
8	2	2	
9	2	2	
10	2	2	
11	2	2	
12	3	3	
13	3	3	
14	3	3	
15	3	3	

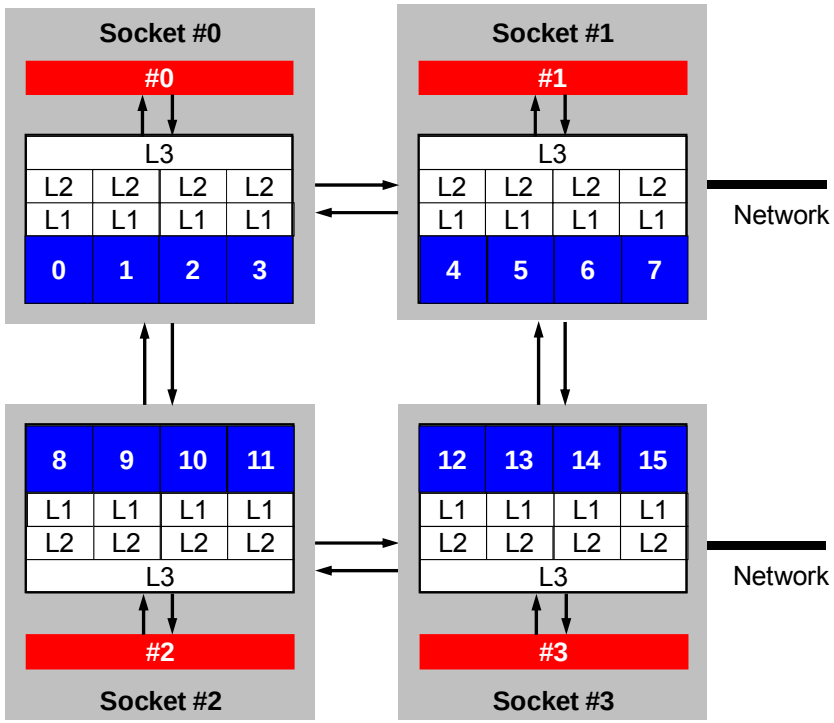
Cyclicな割り付けも可能




```
#!/bin/bash
MYRANK=$MXMPI_ID
SOCKET=$(expr $MYRANK % 4)
numactl --cpunodebind=$SOCKET --membind=$SOCKET $@
```

MPI process	Socket	Memory	Core
0	0	0	
1	1	1	
2	2	2	
3	3	3	
4	0	0	
5	1	1	
6	2	2	
7	3	3	
8	0	0	
9	1	1	
10	2	2	
11	3	3	
12	0	0	
13	1	1	
14	2	2	
15	3	3	

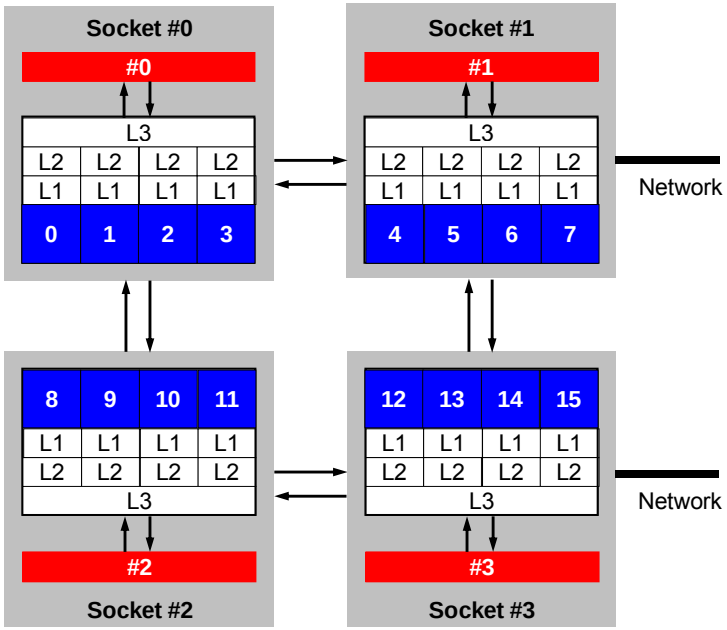
コアを指定することもできる




```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
CORE=$(expr $MYRANK % 16)
numactl --physcpubind=$CORE --membind=$SOCKET $@
```

MPI process	Socket	Memory	Core
0		0	0
1		0	1
2		0	2
3		0	3
4		1	4
5		1	5
6		1	6
7		1	7
8		2	8
9		2	9
10		2	10
11		2	11
12		3	12
13		3	13
14		3	14
15		3	15

8コア使う場合はどうする？



MPI process	Socket	Memory	Core
0		0	0
1		0	2
2		1	4
3		1	6
4		2	8
5		2	10
6		3	12
7		3	14

MPI process	Socket	Memory	Core
0		0	0
1		0	1
2		0	2
3		0	3
4		1	4
5		1	5
6		1	6
7		1	7

MPI process	Socket	Memory	Core
0		0	0
1		0	1
2		1	4
3		1	5
4		2	8
5		2	9
6		3	12
7		3	13

STREAM ベンチマーク

http://www.streambench.org/

- メモリバンド幅を測定するベンチマーク
  - Copy, Scale, Add, Triad（DAXPYと同じ）

```
-------------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------
 Number of processors =           16
 Array size =    2000000
 Offset     =          0
 The total memory requirement is     732.4 MB (     45.8MB/task)
 You are running each test  10 times
 --
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
 -----------------------------------------------------
 -----------------------------------------------------
Function     Rate (MB/s)  Avg time   Min time  Max time
Copy:      18334.1898      0.0280      0.0279      0.0280
Scale:     18035.1690      0.0284      0.0284      0.0285
Add:       18649.4455      0.0412      0.0412      0.0413
Triad:     19603.8455      0.0394      0.0392      0.0398
```

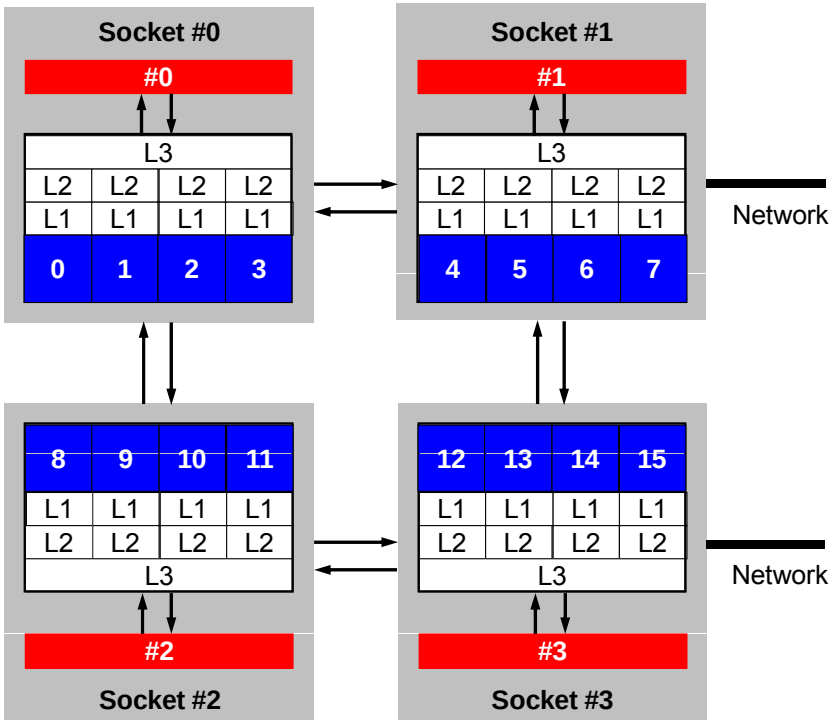
ファイルコピー：今回はFORTRANのみ

```
>$ cd <$T-fem2>

FORTRAN
>$ cp /home/t00000/fem2/stream.tar .
>$ tar xvf stream.tar

直下に「/stream」というディレクトリができている。
<$T-fem2>/streamを<$T-st>と呼ぶ。

>$ cd <$T-st>
>$ mpif90 –Oss –noparallel stream.f –o stream

今回はMPI版だが，OpenMP版もある
```

set-a0.sh：Policy-5相当

- プロセスが割り当てられたソケットのローカルメモリを使う
- ソケット番号は基本的にサイクリック（round-robin的）に割り当てられるが，異なる場合もある（実行ごとに性能が変動）

```
#@$-r stream
#@$-q lecture
#@$-N 1
#@$-J T16
#@$-e err
#@$-o a0-16-1.lst
#@$-lM 28GB
#@$-lT 00:05:00
#@$

cd $PBS_O_WORKDIR

mpirun numactl --localalloc ./stream
exit
```



set-b1/b2.sh：Policy-3,4相当

- 各プロセスをソケット0番から順番に埋まるように割り当て，各ソケットのローカルメモリを使う
- b1とb2ではあまり差は無い

set-b1.sh

```
#@$-r stream
#@$-q lecture
#@$-N 1
#@$-J T8
#@$-e err
#@$-o b1-08-1.lst
#@$-lM 28GB
#@$-lT 00:05:00
#@$

cd $PBS_O_WORKDIR

mpirun ./b1.sh ./stream
exit
```

b1.sh：Policy-3相当

```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --membind=$SOCKET $@
```

b2.sh：Policy-4相当

```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --localalloc $@
```

set-x.sh

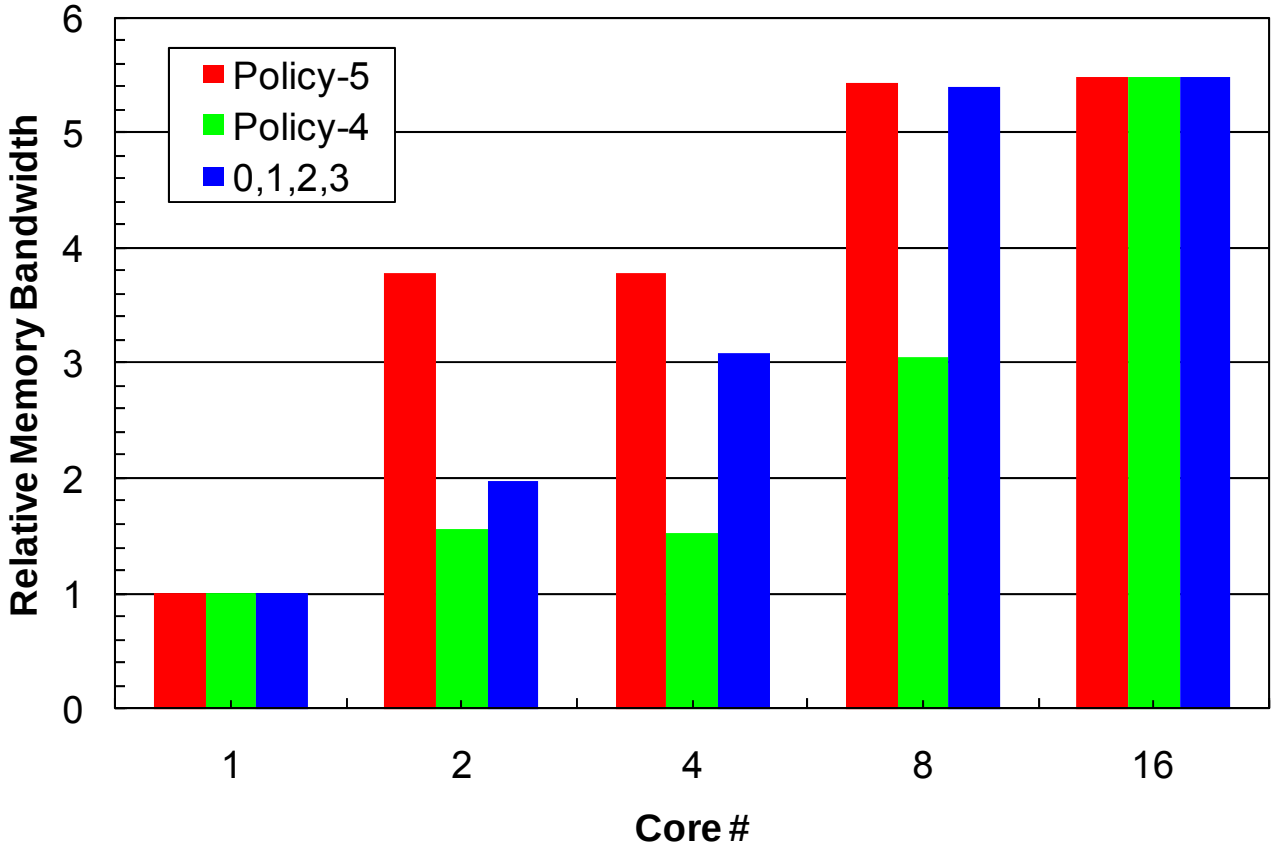
- 各プロセスが割り当てられたソケットのメモリを使う
- T2:0,1, T4:0,1,2,3
- T8:0,0,1,1,2,2,3,3
- T16:0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3

```
#@$-r stream
#@$-q lecture
#@$-N 1
#@$-J T16
#@$-e err
#@$-o x1-16-1.lst
#@$-lM 28GB
#@$-lT 00:05:00
#@$

cd $PBS_O_WORKDIR
```

```
mpirun numactl --cpunodebind=0,1,2,3 --localalloc ./stream
exit
```

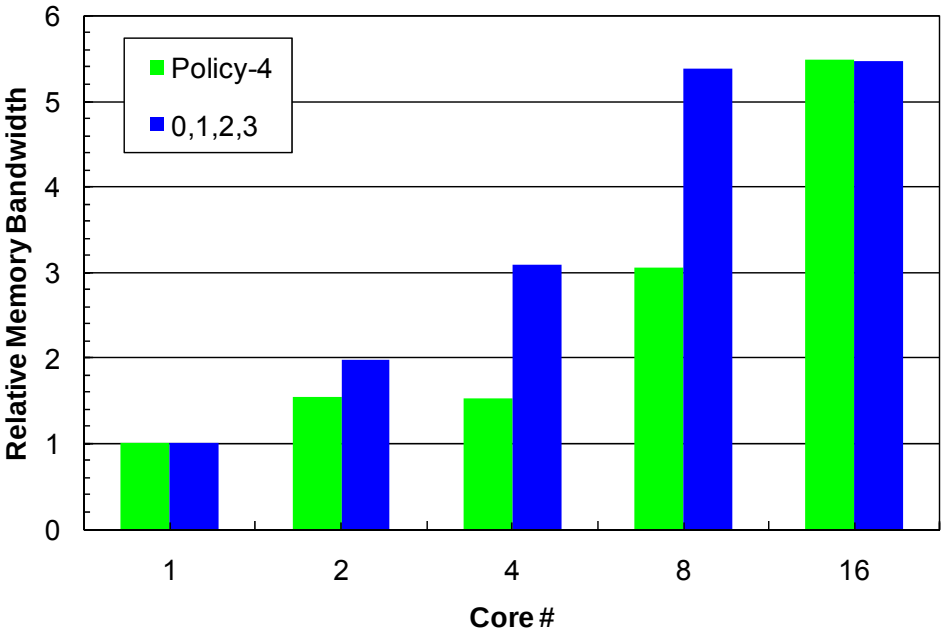
Triadの結果：Policy-5@T1の性能を1.00

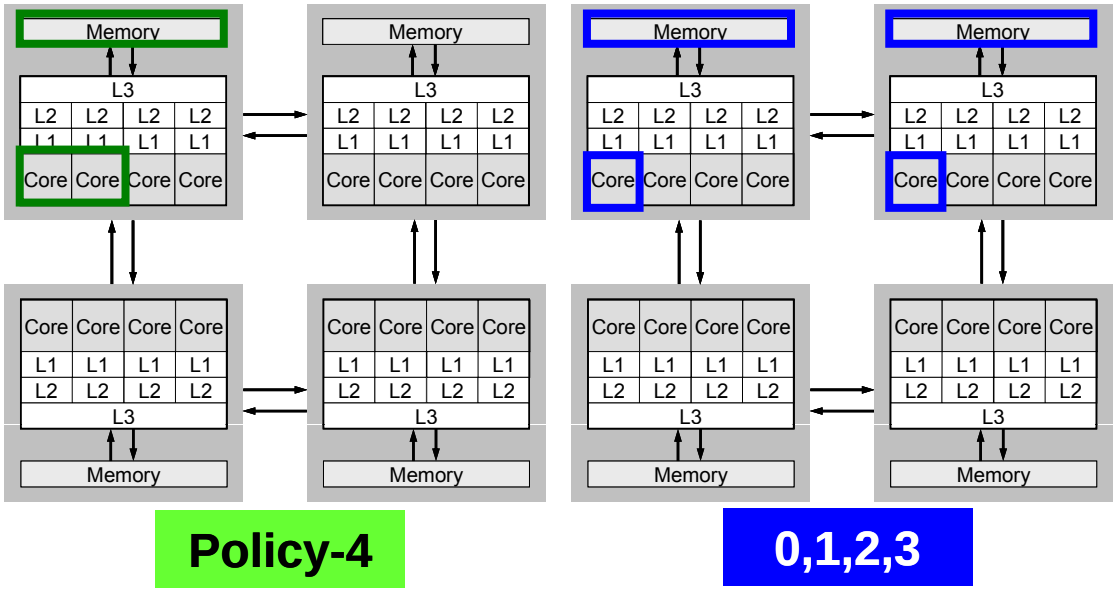
- Policy-5: set-a0，Policy-4: set-b2
- 0,1,2,3:　set-x
- 16コア使った場合，メモリの性能は5倍強にしかなっていない



Triadの性能：T2

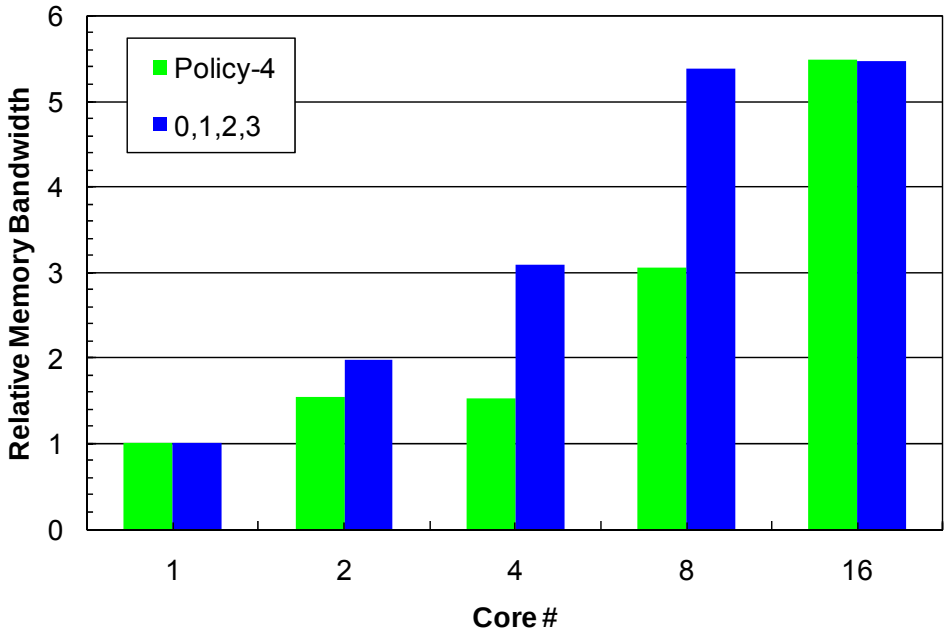
- Policy-4
  - 1コアのときの2倍までは行かないが性能は向上
  - 2コアで1つのメモリを共有するため，多少の競合が生じている
- 0,1,2,3
  - ほぼ100%近い性能向上
  - 各コアでローカルメモリを占有

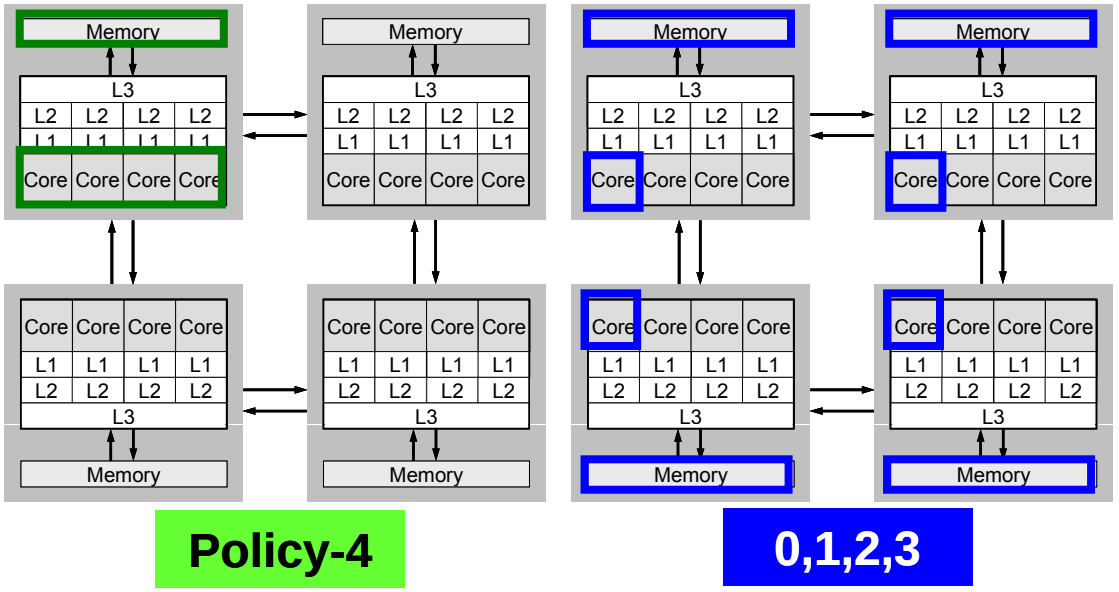




Triadの性能：T4

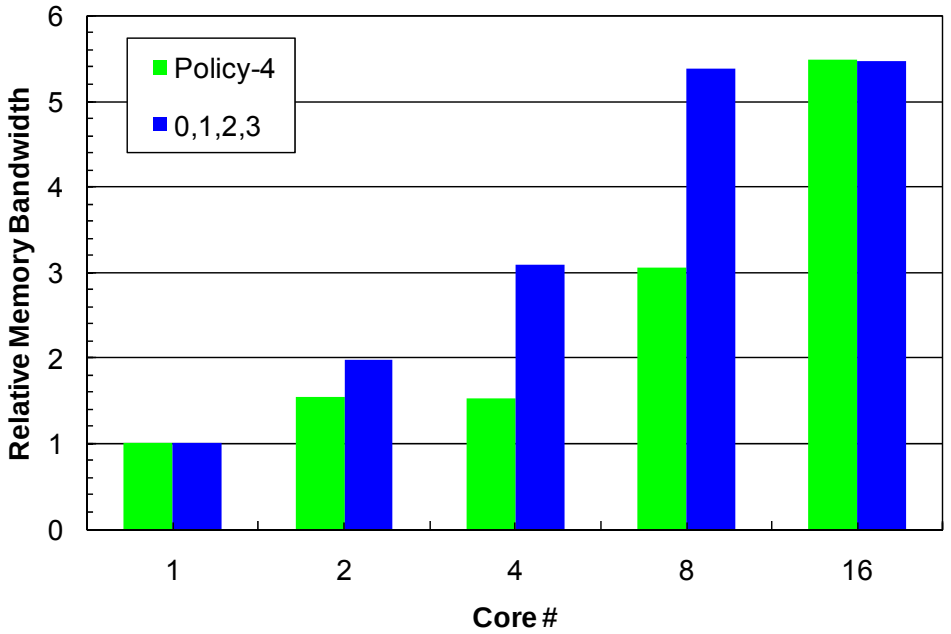
- Policy-4
  - T2のときとほとんど変わらない，つまり1コア当たりのメモリ性能は半分に落ちている
  - 飽和状態
- 0,1,2,3
  - T1の4倍までは行かないが順調にスケール
  - Cache-Coherency
  - 通信の影響もあり

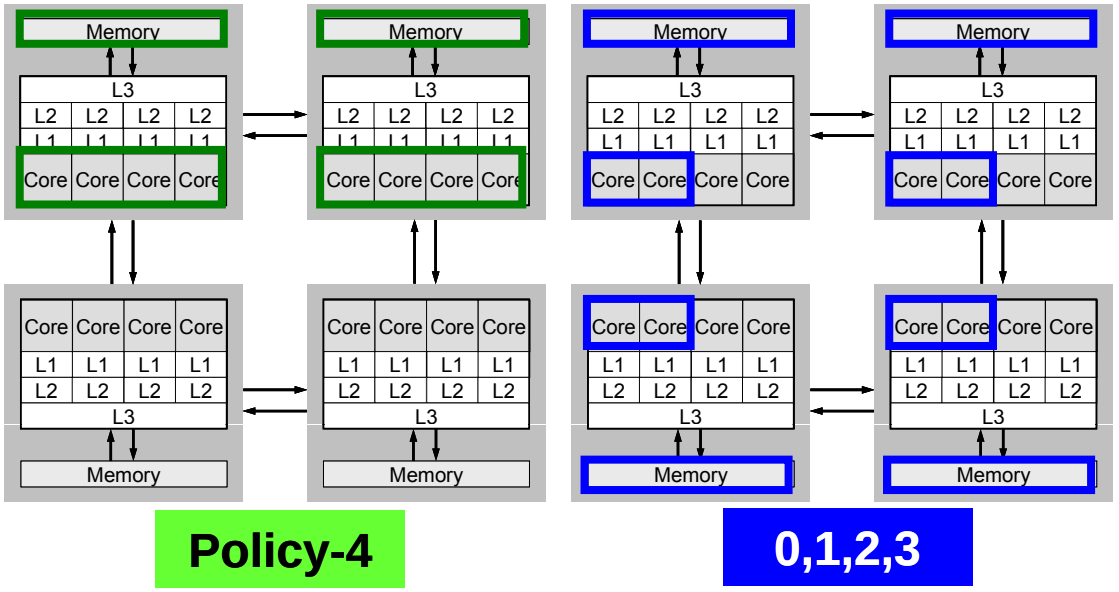




Triadの性能：T8，T16

- Policy-4
  - T8
    - T4の約2倍弱
  - T16
    - T8の約2倍弱
- 0,1,2,3
  - T8
    - T4の約2倍弱
  - T16
    - T8とほぼ同じ
    - Policy-4におけるT2→T4と同じ





T2Kの性能

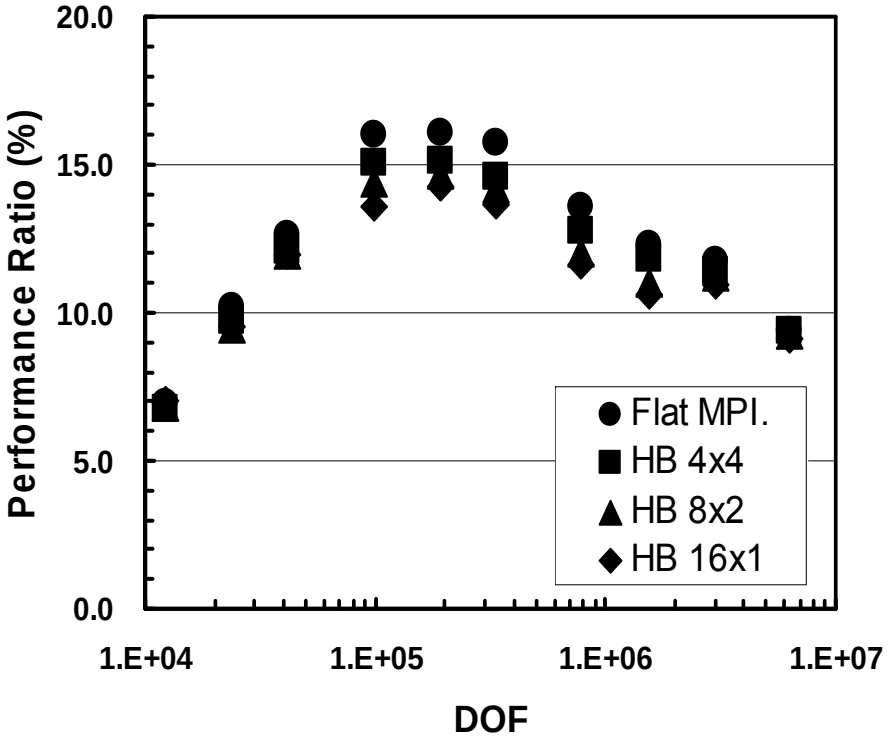
- 実用上はノード内の性能について議論することは余り意味がないのだが・・・
  - 1ノードが最低単位

- メモリの性能（の悪さ）に充分注意する必要がある
  - Strong Scaling（同じ問題をコア数を変えて解く）の場合，基準を1ノードあるいは1ソケット（Policy-2,3,4のようにする）にする

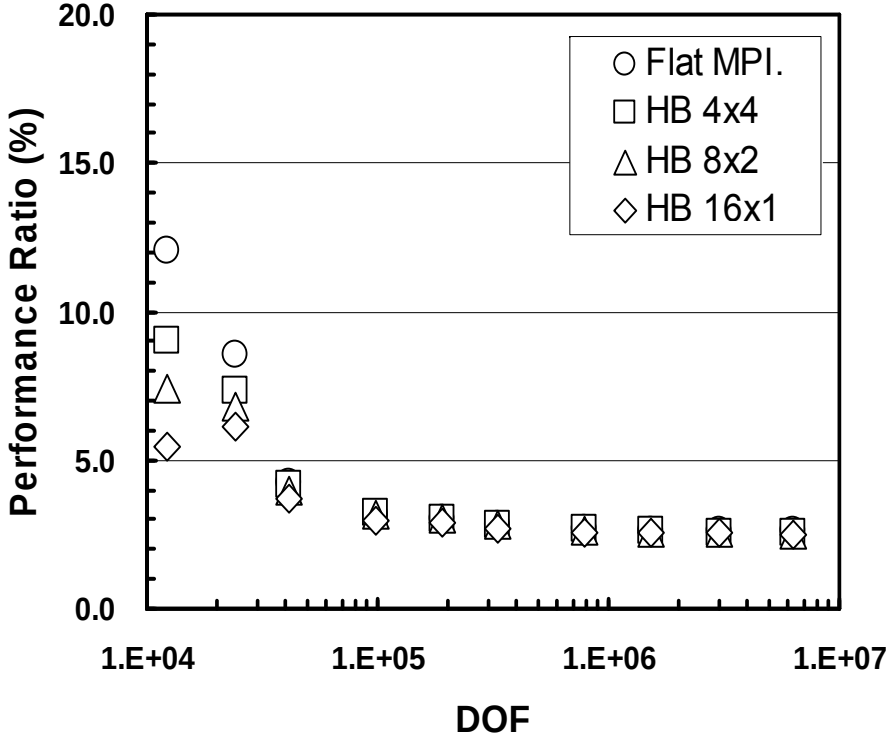
疎行列ソルバーの性能：三次元弾性問題

ICCG法，T2K・SR11000　1ノード：メモリバンド幅が効く

Hitachi SR11000/J2
Power 5+ 2.3GHz x 16
147.2 GFLOPS/node
100 GB/s for STREAM/Triad
L3 cache: 18MB/core



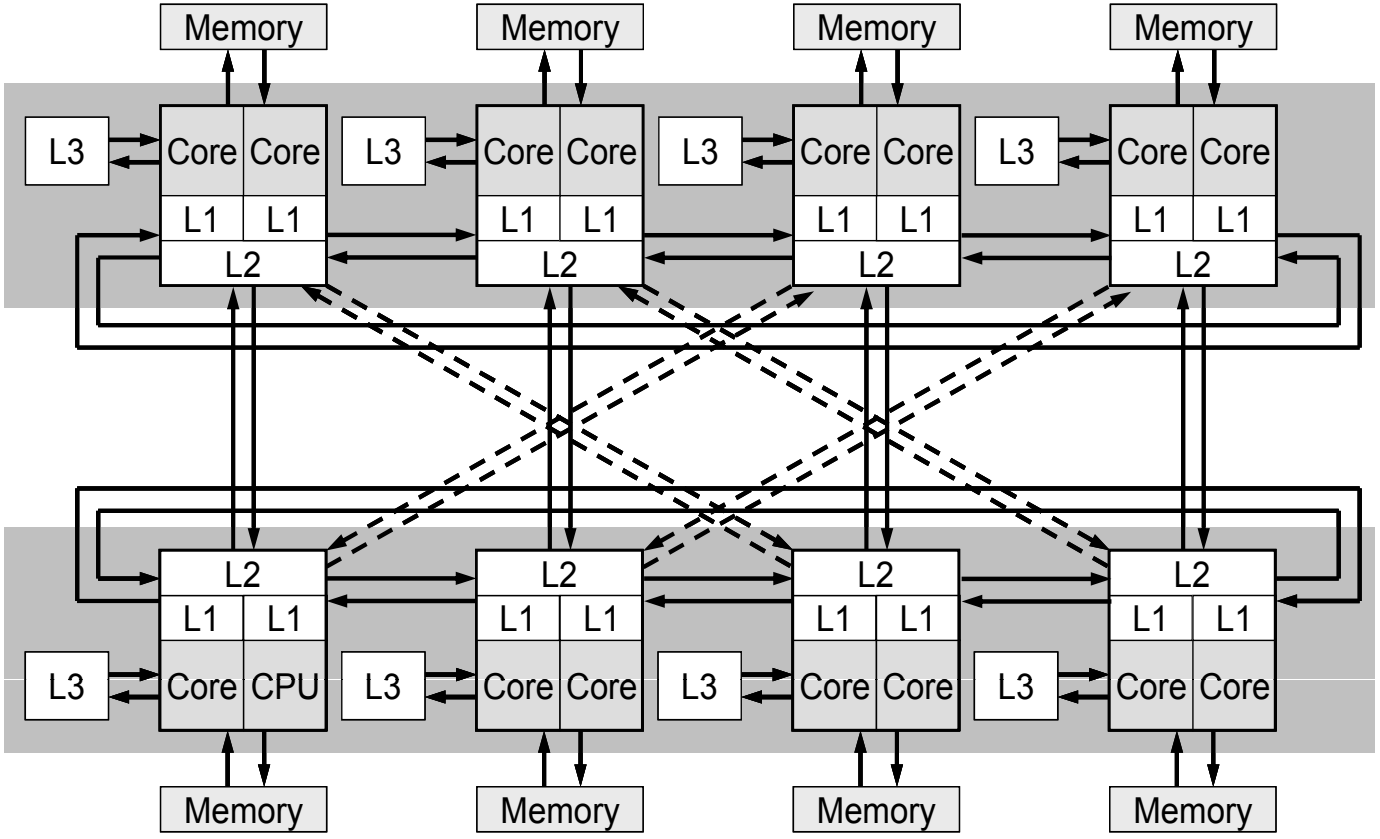
T2K/Tokyo
Opteron 2.3GHz x 16
147.2 GFLOPS/node
20 GB/s for STREAM/Triad
L3 cache: 0.5MB/core



Hitachi SR11000/J2

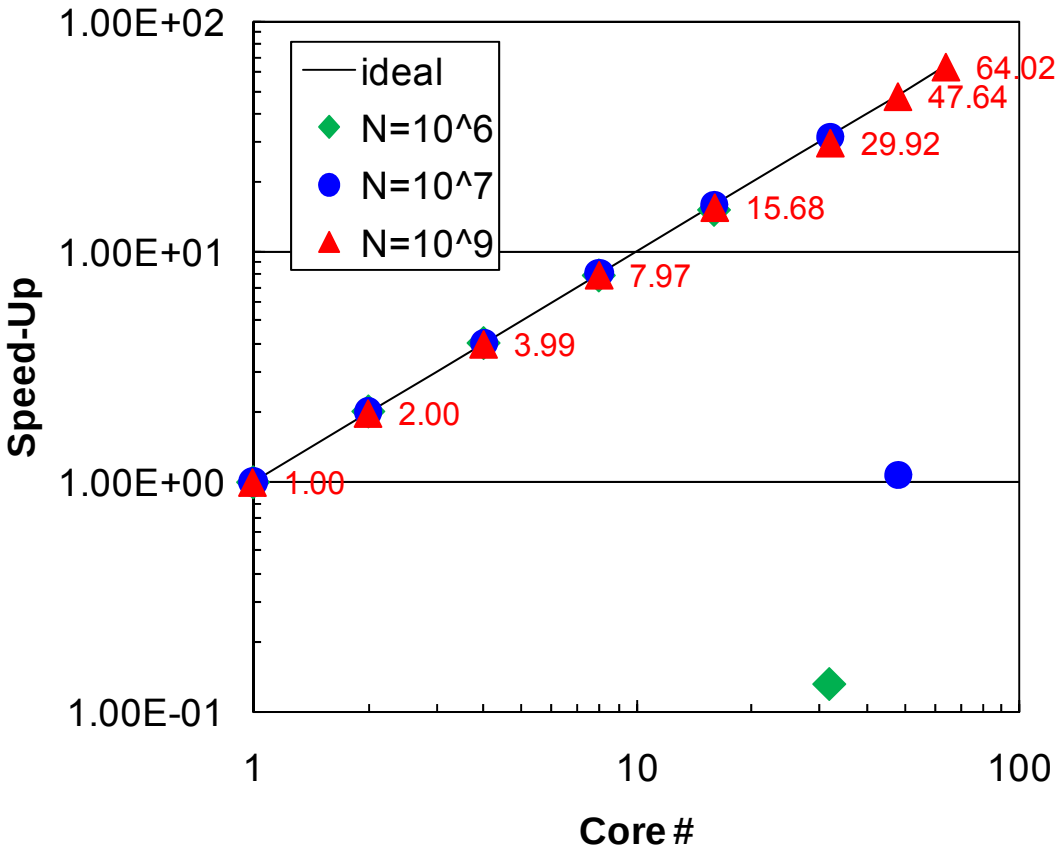
- IBM POWER5+ 2.3GHz (9.2GFLOPS)
- Dual Core×4 ⇒「MCM（Multi Core Module）」×2 ⇒ 1node（16cores）
- based on NUMA architectures, but small latency of memory, huge cache

青木，中村，助川，齋藤，深川，中川，五百木（2005）
スーパーテクニカルサーバーSR11000 モデルJ1のノードアーキテクチュアと性能評価
情報処理学会論文誌：コンピューティングシステム 45-SIG12（ACS11），27-36 より作成



S1-3：T2K（東大）における並列効果

- ◆：N=10^6, ●：10^7, ▲：10^9
- ―：理想値
- 1コアにおける計測結果（sec.）からそれぞれ算出




```
#@$-r test
#@$-q lecture
#@$-N 1
#@$-J T4
#@$-e err
#@$-o hello.lst
#@$-lM 28GB
#@$-lT 00:05:00
#@$

cd $PBS_O_WORKDIR
mpirun numactl --localalloc ./a.out
```

- 赤字部分の影響
- 1ノードより多いと変動あり
- メモリに負担のかからない計算