ＨＰＦの概要

核融合科学研究所
シミュレーション科学研究部
坂上仁志
sakagami.hitoshi@nifs.ac.jp

アウトライン

- JAHPF/HPFPC
- 並列プログラミング
  - HPFと他の手法との比較
- HPFの概要
  - 特徴
  - 利点
  - よく使う指示文
- まとめ

JAHPF

- HPF合同検討会 (Japan Association for HPF)
  - http://www.hpfpc.org/jahpf/
  - 1997年1月29日に発足し，活動を開始した．
- HPFをベースとし，以下の3条件を満たす並列言語仕様を確立する．
  - 標準性：多用なプラットフォーム上で動作
  - 適用性：科学技術計算スキームの実装
  - 高速性：ハードウェア性能の活用
- HPF/JA仕様を策定した．

HPFPC

- HPF推進評議会 (HPF Promoting Consortium)
  - http://www.hpfpc.org/
  - JAHPFを発展解消し，2001年7月9日に設立総会を開催して，任意団体としての活動を開始した．
- 活動の中心を拡張言語仕様の開発から，実用アプリケーションのHPF化促進，実環境でのHPFの評価にフォーカスする．
  - HPF利用を支援するために，会員所有のコードをHPF化するための個別相談や講習会を行う．

コードの並列化

- 実験結果と直接比較しえる大規模のシミュレーションがしたいので，分散メモリ型並列計算機が前提になる．
  - コードを並列化しなければならない．
- コードを並列化するためには，本来の物理モデル／アルゴリズムのプログラミング以外に，並列化のためのプログラミングが必要である．
  - 並列プログラミングにおけるデファクトスタンダードは，MPIである．

MPI並列プログラミング

- 開発途上のコードをMPIでプログラミングすることは大きなストレスとなる．
  - 新しい物理モデルの導入，新しいアルゴリズムの採用によるコードの改変が頻繁にある．
  - 単純なデバック出力でさえも，大きな労力が必要である．
  - バグの原因が，そもそものプログラム／アルゴリズムにあったのか，MPI化によって混入したのかの判断が難しい．
  - 唯でさえ，大規模な分散メモリ型並列計算機の性能を引き出すことは難しい．

本当にMPIで十分か？

- 計算科学ユーザ (≠計算機科学ユーザ) にとって, MPIによる並列プログラミングはあまりにも煩雑である.
  - 並列プログラミングは, 本業ではない.
  - 可能なら完全自動並列化が望ましいが. . .
- MPIしかなければ, 多くの計算科学ユーザは分散メモリ型並列計算機を本来のHPCのために使わないであろう.
  - 複数のパラメータランを同時に実行する環境として並列計算機を見る.

なぜHPFか？

- 比較的簡単にプログラミングできて, そこそこの並列性能が得られればよい.
  - プロダクションラン用コードなら, がんばってMPI化を一回すればよいのだが. . .
- OpenMPは, 共有メモリ型並列計算機でしか使えない.
  - 分散メモリ型並列計算機では使えない.
  - PCクラスタでも使えない.
- 現在, 他に選択肢がない.
  - DARPAはHigh Productivity Computing Systemプロジェクトを進めているが, まだ成果なし.

他の並列プログラミング手法

- OpenMP系列
  - OpenMP+ccNUMA
  - OpenMP+Software Distributed Shared Memory
  - Cluster OpenMP
- 並列言語
  - Co-Array Fortran
  - Unified Parallel C
  - Chapel, X10, ~~fortress~~

OpenMP系列

- 分散メモリを共有メモリのように見せるので，プログラミングは楽だが．．．
- 超高速なネットワークが必須である．
  - ハードウェアが非常に高価になる．
- リモートメモリへのアクセスが発生すると，性能が急激に低下する．
  - アクセスのローカリティを保証できない．
  - first touchだけでは，無理がある．
- **大規模並列での性能は，期待できない．**

Co-Array Fortran(Unified Parallel C)

- データのプロセッサ上への分散配置だけでなく，データ転送を明示的に記述しなければならない．
  - MPIとプログラミングの手間は，同じ？
- Fortranコンパイラでは，エラーになる．
- 2005年にFortran2008の標準としていったん採用されたが，最近，標準実装から削除する動きが活発である．
  - なぜFortran標準になったのか，理解不能と言う人が大勢いる．

サンプルプログラム

```
real :: r[*]      ! Scalar co-array
real :: x(n)[*] ! Array  co-array
! Co-arrays always have assumed co-size

real :: t         ! Local scalar
integer :: p      ! Local scalar

! Remote array references
! MPI_GET communication
t = r[p]
x(:) = x(:)[p]

! Reference without [] is to local part
! MPI_PUT communication
x(:)[p] = r
```

Chapel

- Cascade High-Productivity Language
- アドレス空間はグローバルであり，通信はコンパイラが自動的に生成する．
- データ並列，タスク並列を抽象化できる記述方法を提供する．
  - localeとdomain
- 考え方は，ほとんどHPFと同じ？？？
  - ただし，HPFとは逆に，OpenMPと同様にデータ分散より処理分担を優先している．

サンプルプログラム

```
var N: integer = 1000;
var A, B: [1..N] float;

// forall specify parallel execution
forall i in 2..n-1 do
   A(i) = B(i-1) + B(i+1);
```

```
var N: integer = 1000;
var CompGrid: [1..N] locale;
var D: domain(2) distributed(Block(2), CompGrid);
var A, B: [1..N] float;

forall i in D on B(i) do
   A(i) = B(i);
```

並列プログラミング手法の比較

- 並列プログラミングで大事な三つのポイントについて，それぞれの手法を比較する．

	データの分散	処理の分担	通信の生成
MPI			
OpenMP+DSM	※分散共有メモリ		※分散共有メモリ
Cluster OpenMP			※分散共有メモリ
HPF			
CAF(UPC)	※Co-Array		※代入文
Chapel			※グローバルメモリ

自動(不要)

自動＋
手動の最適化

手動(指示行)

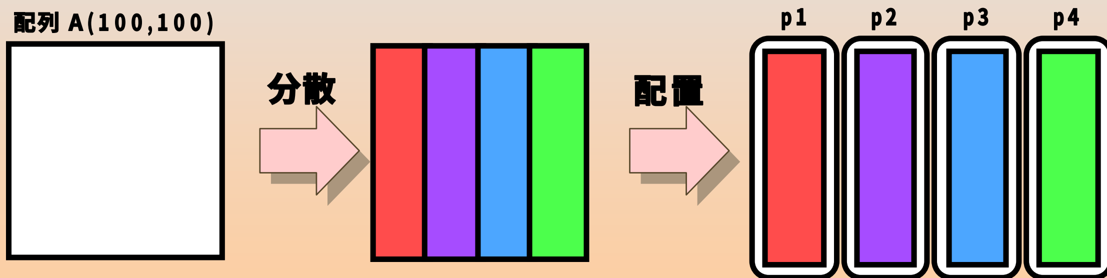
手動

HPFの特徴

- データ並列のプログラミング
  - 単一スレッド，単一の制御流れ
  - グローバルな名前空間，データ空間の共有
  - 緩やかな同期
- 優れた可搬性
  - 高い抽象度により，異なるアーキテクチャでもソースプログラムの互換性がある．
  - HPF指示文は，通常のFortranではコメント行として扱われる．

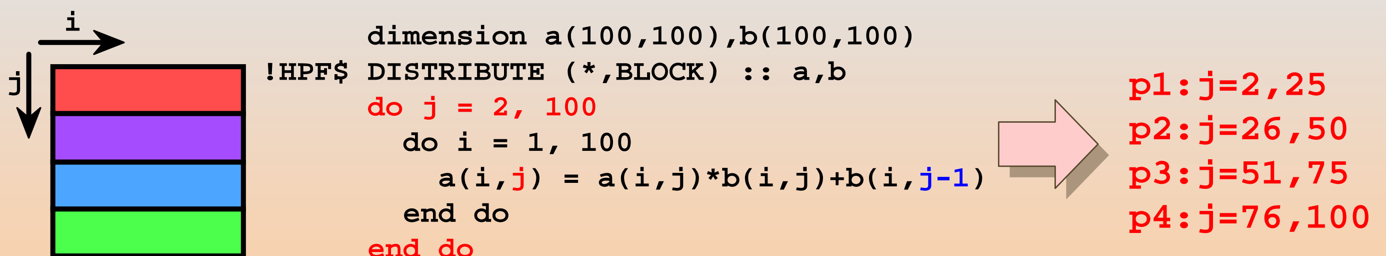
HPFによる並列化#1

- ユーザは, Fortran配列をどのように複数のプロセッサ上へ分散して配置するかだけを明示的に指定する.（データ分散）



HPFによる並列化#2

- 更新データを保持するプロセッサが処理を行うように並列化する.（処理分担）
  - Owner Computes Rule




```
       dimension a(100,100),b(100,100)
!HPF$ DISTRIBUTE (*,BLOCK) :: a,b
       do j = 2, 100
         do i = 1, 100
           a(i,j) = a(i,j)*b(i,j)+b(i,j-1)
         end do
       end do
```

- 必要なデータ転送は, コンパイラが自動で面倒を見る.

ＨＰＦの利点

- HPFは通常のFortranプログラムに指示文を挿入するだけである．
  - プログラミングが容易である．
  - 通常のFortranではコメント行として扱われるので，逐次プログラムとして互換性がある．
- HPF指示文を増やすことで，段階的に並列化ができる．
  - ある程度の満足できる並列性能が得られれば，並列化の作業をそこで止めればよい．
  - 問題があった場合，前の段階にすぐに戻れる．

プログラムの比較

```
parameter(n=100)
real a(n), b(n)
read(*,*) a, b
aipd = 0.0
do i = 1, n
  aipd = aipd + a(i) * b(i)
end do
write(*,*) 'aipd = ', aipd
stop
end
```

MPI

```
parameter(n=100)
real a(n), b(n)
call MPI_INIT ( ierr )
call MPI_COMM_SIZE ( MPI_COMM_WORLD, np, ierr )
call MPI_COMM_RANK ( MPI_COMM_WORLD, id, ierr )
if( id .eq. 0 ) then
  read(*,*) a, b
  do i = 1, np-1
    call MPI_SEND ( a, ...
    call MPI_SEND ( b, ...
  end do
else
  call MPI_RECV ( a, ...
  call MPI_RECV ( b, ...
end if
is = ( n / np ) * id + 1
ie = ( n / np ) * ( id + 1 )
aipdt = 0.0
do i = is, ie
  aipdt = aipdt + a(i) * b(i)
end do
call MPI_REDUCE ( aipdt, aipd, ...
if( id .eq. 0 ) write(*,*) 'aipd = ', aipd
call MPI_FINALIZE ( ierr )
stop
end
```

HPF

```
      parameter(n=100)
      real a(n), b(n)
!HPF$ PROCESSORS proc(number_of_processors())
!HPF$ DISTRIBUTE (BLOCK) ONTO proc :: a,b
      read(*,*) a, b
      aipd = 0.0
!HPF$ INDEPENDENT, REDUCTION(+:aipd)
      do i = 1, n
        aipd = aipd + a(i) * b(i)
      end do
      write(*,*) 'aipd = ', aipd
      stop
      end
```

HPFの機能

- HPF 2.0基本機能
  - 単純なデータの分散とループ処理の分担
  - 外部手続の並列呼び出し
- HPF 2.0公認拡張機能
  - 複雑なデータ分散
  - タスク並列実行
- HPF/JA 1.0拡張機能
  - 集約演算の拡張
  - 明示的な通信の最適化

従来のHPFコンパイラ

- 従来のHPFコンパイラは，仕様制限，解析能力，並列性能に大きな問題があった．
  - 欧米でのHPF衰退の大きな原因となった．
- HPFは，PCクラスタでしか使えなかった．
  - 実用アプリは，高ベクトル化されていることが多いので，実計算を考えるとPCクラスタでの並列実行よりベクトル型スパコン1台での実行の方が遙かに効率が良い．
  - 実用アプリをHPF化する元気が出ない．

ＳＸ用ＨＰＦコンパイラ

- 従来のコンパイラより大幅な性能向上
- トータルで得られる大きな計算能力
- HPF普及のためにはPCクラスタ上で動作する無料のコンパイラは必要であろう.
  - NEC製のHPFコンパイラ
    - SX版とほぼ同じ高い能力
    - ライセンス上の問題で, NEC製PCのみで利用可能
  - 富士通製のHPFコンパイラ (fhpf)
    - 開発中のためやや低い実装レベル
    - 標準的なPCクラスタ環境で制約なしに利用可能

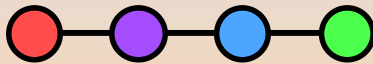
ＨＰＦ指示文

- 抽象プロセッサの定義
- 配列の分散, 整列
  - 1次元配列, 2次元配列, . . .
  - BLOCK, CYCLIC, GEN_BLOCK
- 並列実行の指示
  - ループ, 変数, 演算
- ループ処理分担の指示
- 袖領域の定義と通信
- 不必要な通信の抑制

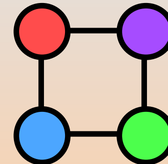
抽象プロセッサの定義

- 抽象プロセッサの名前，次元およびサイズを定義する．

```
!HPF$ PROCESSORS linear(4)
```



```
!HPF$ PROCESSORS mesh(2,2)
```

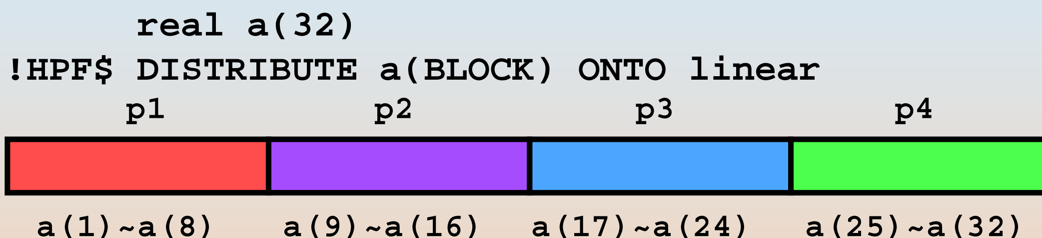


- ただし，抽象プロセッサから物理プロセッサへのマッピングは，処理系に依存する．

1次元配列の分散

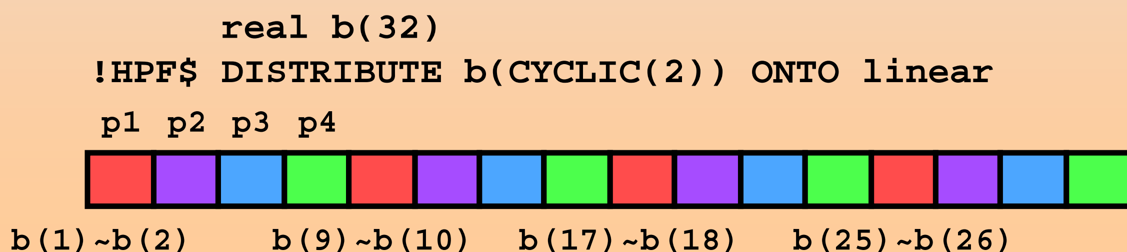
- BLOCK分散

```
      real a(32)
!HPF$ DISTRIBUTE a(BLOCK) ONTO linear
```



- CYCLIC分散

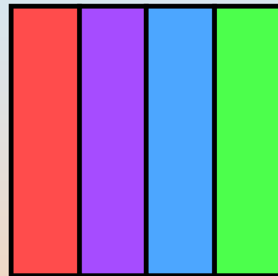
```
      real b(32)
!HPF$ DISTRIBUTE b(CYCLIC(2)) ONTO linear
```



2次元配列の分散

```
 real a(8,8)
!HPF$ DISTRIBUTE a(TYPE,TYPE) ONTO linear
```

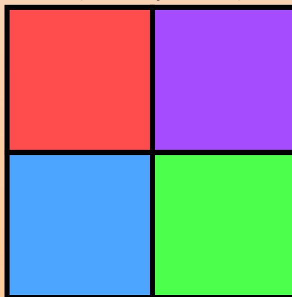
a(BLOCK,*)



a(*,CYCLIC)

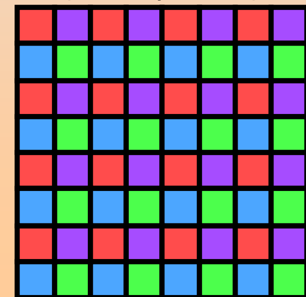
```
 real a(8,8)
!HPF$ DISTRIBUTE a(TYPE,TYPE) ONTO mesh
```

a(BLOCK,BLOCK)



a(BLOCK,CYCLIC)

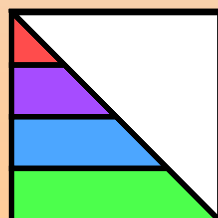
a(CYCLIC,CYCLIC)

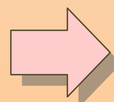


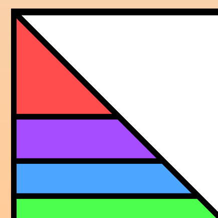
不均等分散

- 3角行列の演算等では，配列を均等に分散すると各プロセッサの計算負荷が不均一になり，全体的な効率が悪くなる．
- GEN_BLOCKを用いた不均等分散で，各プロセッサの計算量を均等にする．

```
      real a(256,256)
      integer,parameter::m(4)=(/128,53,41,34/)
!HPF$ DISTRIBUTE a(*,GEN_BLOCK(m)) ONTO linear
```



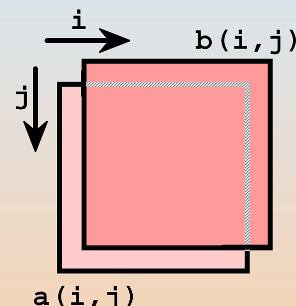




配列の整列

- 以下のような場合，同じ分散を指定すると境界でデータ転送が発生する．

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



- 複数の配列間の相互関係を指定してデータ転送が発生しないように調整する．

```
!HPF$ ALIGN a(i,j) WITH b(i+1,j-1)
```

並列実行の指示（ループ）

- コンパイラだけでは判断できない場合にループを並列に実行しても問題のないことを明示する．
  - *VDIR NODEPと同じ

```
!HPF$ INDEPENDENT
      do i = 1, 100
        a(ind(i)) = a(ind(i)) + b(i)
      end do
```

  - ind(i)に重なりがあった場合，並列実行すると回帰参照により正しい実行結果が得られないが，コンパイラはind(i)に重なりがあるかどうか判断できないのでループを並列化しない．
  - そこで，ind(i)に重なりがない場合，コンパイラにループの並列化を指示する．

並列実行の指示 (変数)

- ループの各繰返しで一時的に使われる変数があると厳密な意味でのINDEPENDENTにはならない.

```
!HPF$ INDEPENDENT, NEW(tmp)
      do i = 1, 100
        tmp  = a(i) + b(i)
        c(i) = tmp * c(i)
      end do
```

- あるプロセッサがtmpに値を代入後, 参照する前に別のプロセッサがtmpの値を更新するかもしれない.
- しかし, その変数に対してNEW宣言をすると, 繰返し毎に新しい実体を割り当てるため, ループが並列化される.
- ただし, スカラ変数の場合には, コンパイラが自動的に判断することが多い.

並列実行の指示 (演算)

- ループ中に総和等の集約演算があると, その演算結果を求める変数に対してはNEW宣言ができず, ループはINDEPENDENTにはならない. そこで, 各プロセッサが部分的な結果を求め, 最後に全プロセッサで最終的な結果を求める特別な指示をする.

```
      s = 0.0
!HPF$ INDEPENDENT, REDUCTION(+:s)
      do i = 1, 100
        s = s + a(i) * b(i)
      end do
```

- 総和計算であることを明示する.

ループ処理分担の指示

- コンパイラのループ処理分担が不適切な場合，データ転送が多発して並列効率が低下するので，分担の仕方を指示する．

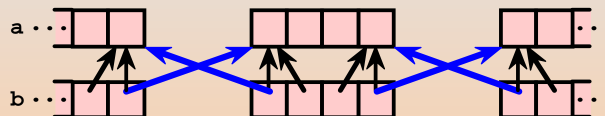
```
      real a(100), b(100), c(100), d(100)
!HPF$ DISTRIBUTE(BLOCK) ONTO linear::a,b,c,d
      do i = 1, 99
!HPF$ ON HOME(b(i+1))
        a(i) = b(i+1) + c(i+1) + d(i+1)
      end do
```

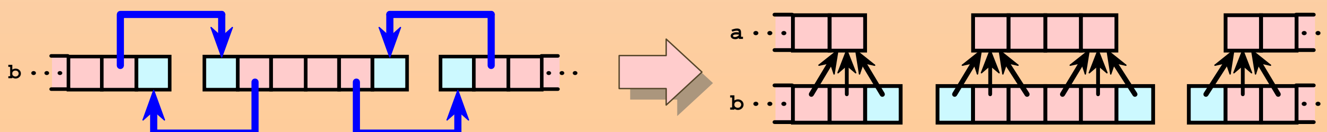
  – この場合，配列(i)を保持するプロセッサより配列(i+1)を保持するプロセッサで実行する方が，データ転送量が少ない．

袖領域の定義とその通信

- 分散配列に袖領域を定義して，その部分を一括通信すると効率が良い．

```
      real a(100), b(100)
!HPF$ DISTRIBUTE(BLOCK) ONTO linear::a,b
!HPF$ SHADOW b(1)
      do i = 1, 100
         b(i) = …
      end do
!HPF$ REFLECT b
!HPF$ INDEPENDENT
      do i = 2, 99
        a(i) = b(i-1) + b(i) + b(i+1)
      end do
```





不必要な通信の抑制

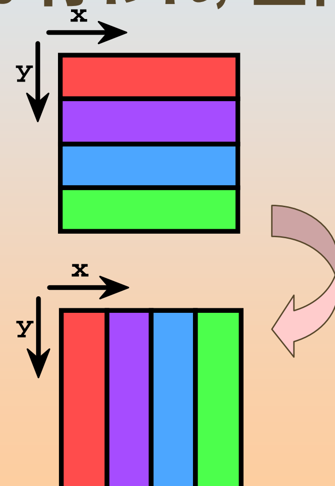
- コンパイラの解析能力が不足している場合や解析できない場合には，不必要な通信を行ってしまうかもしれない．そこで，データ転送の必要がないことをユーザが明示的に指示し，無駄な通信を抑制する．

```
!HPF$ REFLECT b
!HPF$ INDEPENDENT
      do i = 2, 99
!HPF$ ON HOME(a(i)), LOCAL
        a(i) = b(i-1) + b(i) + b(i+1)
      end do
      …
      call sub ( b )
      …
!HPF$ INDEPENDENT
      do i = 1, 99
!HPF$ ON HOME(c(i)), LOCAL
        c(i) = ( b(i) + b(i+1) ) / 2.0
      end do
```

動的な分散の変更

- 配列に対する最適な分散が異なる場合，分散を動的に変更すると一括して通信が行われ，全体的な効率が良い．

```
      dimension a(256,256)
!HPF$ DISTRIBUTE a(*,BLOCK)
      call subx ( a )
      call suby ( a )
      …
      subroutine subx ( a )
      dimension a(256,256)
!HPF$ DISTRIBUTE a(*,BLOCK)
      …
      subroutine suby ( a )
      dimension a(256,256)
!HPF$ DISTRIBUTE a(BLOCK,*)
```



– サブルーチンの呼出し時に自動的に変更される．

HPFによるプログラミング

1. プログラムにおいて並列化する部分を決定する.
   – 10%のサブルーチンがCPU時間の90%を費やす.
2. その部分において，並列化のためのデータ転送が最も少いデータ分散方法を決定する.
   – PROCESSORS, DISTRIBUTE指示文
3. コンパイラが自動並列化できないループについて明示的に並列化を指示する.
   – INDEPENDENT指示文
4. 各種最適化を行う.
   – ループ処理分担の追加補足のための指示
   – データ転送を最適化するための指示

まとめ＃１

- HPFでは，指示文を順次増やすことで，ステップバイステップに並列化やチューニングができる.
  – 並列化の手間を考慮して，得られた並列性能に納得できれば，並列化をそこで止めればよい.
  – 問題があった場合，すぐに前の段階に戻れる.
- とりあえず，手軽にやってみる！
  – 最初から苦労してMPIでプログラミングするより，まず，HPFによる並列化を試してみる価値は，十分にある！

まとめ＃2

- 規則的な構造の問題なら，HPFでも十分に並列性能が得られることが多い．
  - プログラムの構造から，できるだけ通信が起こらないデータ分散を考える．
- 並列化できても性能が出ない場合がある．
  - 原因の追及は，やや難しい．
    - コンパイラの詳細メッセージ
  - 原因がわかれば，比較的容易に解決できる場合が多い．
  - HPFPCに相談していただければ，お手伝いできます．

まとめ＃3

- 不規則な構造を持つ問題は，HPFでの並列化が難しいことが多い．
  - 一般に，MPIでも並列化は難しいが. . .
- 圧縮された疎行列形式を用いている場合は，GEN_BLOCKで対応できる．
- FEMのように不規則構造が静的な場合は，HPFのNEC独自拡張であるHALOを用いれば，対応できる．