分散並列シミュレーション入門

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

- 熱伝導方程式
  - 陽的解法スキーム
- 並列計算
  - 並列コンピュータ
  - 並列化の方法
  - 並列プログラミング
- MPIの問題点
  - MPIとHPF
- 他の並列プログラミング手法

熱伝導方程式

- 熱の伝導や乱歩による粒子の拡散を記述する方程式である.
- 例として2次元熱伝導方程式を考える.

$$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

- 伝導(拡散)のしやすさは,拡散係数Dによって決まる.
  - 媒質や物質の状態によって決まる定数である.
  - 例えば, $D_{銅} > D_{鉄}$ である.

陽的解法スキーム

- この方程式を解くために,微分を差分で表現して一番簡単な陽的解法スキームを考える.

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} = D\left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2}\right)$$

  - ここで,nは時間,i,jは空間の離散化メッシュを表す.
    - $t=n\Delta t, x=i\Delta x, y=j\Delta y$
- 計算機では,以下の式を計算すればよい.

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \kappa\left(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} + T_{i,j-1}^{n} - 4T_{i,j}^{n}\right)$$

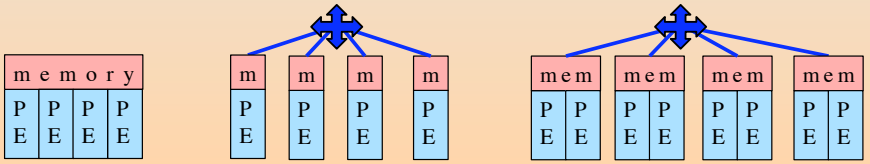
  - ただし,$\Delta x=\Delta y=\Delta, \kappa=D\Delta t/\Delta^2$とした.
  - $\kappa \leqq 1/4$にしなければならない.

なぜ並列計算?

- より大規模,高精度の計算が要求されている.
  - 空間サイズを10倍すると100倍の計算規模になる.
  - 精度を10倍にするためには,Δを1/10にしなければならず,100倍の計算規模になり,かつ,同じ時刻まで計算するためには,10^2倍の時間ステップが必要になる.つまり,計算時間は,10,000倍になる.
- シングルプロセッサの性能が頭打ちである.
  - むしろ,電力消費量を考慮して,性能が低下する傾向にある.(P∝クロック^2)
  - 性能低下をマルチコア化で補う.

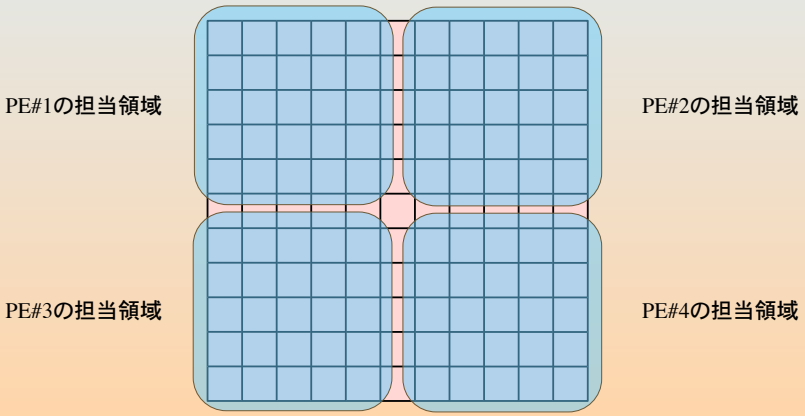
並列計算と並列コンピュータ

- 並列計算とは?
  - n台のコンピュータで同時に計算し,一台当たりの計算量を1/nにする.
- 並列コンピュータのアーキテクチャ
  - 共有メモリ型:複数のPEでメモリを共有
  - 分散メモリ型:独立した(PE+メモリ)をネットワーク接続
  - 分散共有メモリ型:共有メモリ型をネットワーク接続



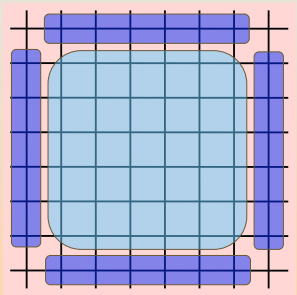
並列化のための空間領域分割

- 空間領域を分割して,それぞれを別々のPEで計算することにより,並列に計算をする.



自担当領域外のデータ参照

- スキームは,(i,j)の値を計算するため(i±1,j)および(i,j±1)のデータを利用するため,自担当領域外の隣接領域のデータを参照する必要がある.



共有メモリ型並列コンピュータと共有並列プログラミング

- PE間でメモリを共有しているため,他PE担当領域のデータを自然に参照できる.
- OpenMPの指示文を挿入することにより,簡単に並列プログラミングができる.
  - 領域を1次元で分割した例を以下に示す.

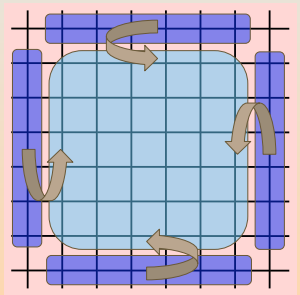
```
!$OMP PARALLEL DO PRIVATE(ix)
      do iy = 2, ly-1
         do ix = 2, lx-1
            tt(ix,iy) = t(ix,iy) + kappa *
     $              ( t(ix+1,iy  ) - 2.0*t(ix,iy) + t(ix-1,iy  )
     $              + t(ix  ,iy+1) - 2.0*t(ix,iy) + t(ix  ,iy-1) )
         end do
       end do
```

共有メモリ型並列コンピュータの限界

- メモリを共有するためのハードウェアの制約により,並列実行できる台数に制限がある.
  - 最大でも16台程度が現実的である.
  - それを超えるためには,高価な特別のハードウェアが必要となる.
- 並列実行する台数を増やしても,全メモリ量が増加するわけではない.
  - より大規模なサイズの計算ができるわけではない.
- 分散(共有)メモリ型コンピュータが必要になる.
  - マルチコア化により分散共有型が今後の主流になる.

分散メモリ型並列コンピュータ

- 他PE担当領域のデータは,そのままでは参照できないため,そのデータを転送し,自メモリ上に持ってこなければならない.

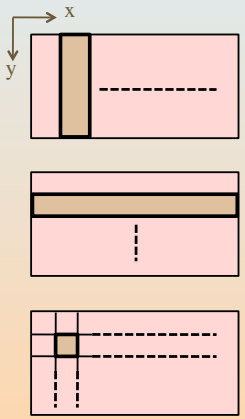


並列コンピュータのネットワーク

- 並列コンピュータのネットワークは,特別なハードウェアを用いて構成されている.
  - メッシュ,ハイパーキューブ,多段クロスバー,...
- このため,PCをスイッチ／ハブで接続しただけのクラスターに比べて性能が良い.
  - 特に,台数が増えると性能の違いが大きくなる.
  - 勿論,その分高価である.
- 一般に,PE#1とPE#2間,PE#3とPE#4間,...は同時にデータ転送可能である.

分割方法と転送データ量

- 領域を分割する方法によって,転送するデータ量が変わる.
- X方向にだけ分割する場合
  - L_y(台数に依存しない)
- Y方向にだけ分割する場合
  - $L_x > L_y$(台数に依存しない)
- 両方向に分割する場合
  - $L_x / n_x + L_y / n_y$
  - L_x=L_yで16台(n_x=n_y=4)の場合,1方向に比べて転送データ量は半分になる.

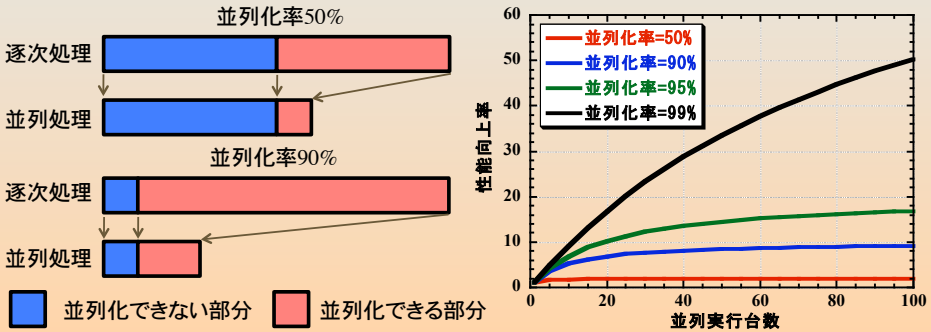


分割方法とデータ転送回数

- 領域を分割する方法によって,時間ステップ当たりのデータ転送回数が変わる.
  - 通信時間=レイテンシー+(データ量／転送速度)
  - 高速なネットワークほど転送回数の影響が大きい.
- X(Y)方向にだけ分割する場合
  - 1回
- 両方向に分割する場合
  - 2回
- 多データ1回と少データ2回の全通信時間
  - どちらが短いかは,ネットワークによる.

並列化率とアムダールの法則

- 計算処理のうち並列処理(並列化)できる割合を並列化率と呼ぶ.
- 例え無限大の台数から構成された超並列コンピュータでも,並列化率が低いと高い性能を発揮できない.

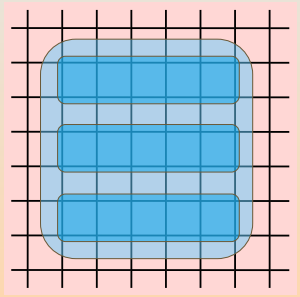


分散並列プログラミング

- プログラム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.
  - プログラムの実行フローを意識した上で,データの転送を明示的にプログラムしなければならない.
- 分散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.
  - Message Passing Interface
  - APIを提供(サブルーチンライブラリ)

分散共有メモリ型並列コンピュータ

- 分散並列化された自担当領域の処理を更に共有並列によって並列化する.
  - 階層並列化



ハイブリッドプログラミング

- 分散並列プログラミングと共有並列プログラミングの両方を用いることをハイブリッドプログラミングと呼ぶ.
- 共有並列プログラミングに用いるOpenMPは容易に扱えるので,実際には,分散並列プログラミングとほとんど同じである.
  - 分散並列化した後,共有並列化を考える.
  - 二重ループの外側を分散並列化し,内側を共有並列化する方法もある.

MPIプログラミングの問題点

- 一般のユーザにとって,MPIでプログラミングすることは,大きなストレスとなる.
  - 単純なデバック出力でさえも,プログラミングに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.
  - ソフトウェアの継承と開発の持続性を考えると,非常に問題がある.
    - Life is too short for MPI
      (T-shirts message@WOMPAT2001)

本当にMPIで十分か?

- 一般のユーザ(≠コンピュータの専門家)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.
  - 本来の解きたい問題に集中できない.
  - 可能なら完全自動並列化が望ましいが…
- もしMPIしかなければ,多くの一般のユーザにとって,分散メモリ型並列コンピュータを本来のHPCのために使いこなすのは,難しい.
  - 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コンピュータを見る!!

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,共有メモリ型並列コンピュータでしか使えない.
  - 分散メモリ型並列コンピュータでは使えない.
  - PCクラスタでも使えない.
- 現在,他に選択肢がない.

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

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
```

```
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
```

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の現状

- 現在活動しているのは,ほぼ日本だけ.
- 残念ながら,この段階から今後広く普及する可能性は低いと言わざるを得ない.
  - 富士通のVPP Fortranと同様に,NECの方言として利用される可能性が高い.
- しかし,並列プログラミングを試すには,非常に便利である.
  - 逐次プログラムとして互換性がある.
  - 並列化の作業は,いつでも止められる.

陽的解法スキームのHPFプログラム

```
      dimension u(lx,ly), uu(lx,ly)
!HPF$ DISTRIBUTE (*,BLOCK) :: u, uu
!HPF$ SHADOW (0,1) :: u
      ...
!HPF$ REFLECT u
!HPF$ INDEPENDENT, NEW(ix,iy)
      do iy = 2, ly-1
!HPF$ ON HOME(uu(:,iy)), LOCAL BEGIN
         do ix = 2, lx-1
            uu(ix,iy) = u(ix,iy) + akap *
     $               ( u(ix+1,iy  ) -2.0*u(ix,iy)+u(ix-1,iy  )
     $               + u(ix  ,iy+1) -2.0*u(ix,iy)+u(ix  ,iy-1) )
         end do
!HPF$ END ON
      end do
!HPF$ INDEPENDENT, NEW(ix,iy)
      do iy = 1, ly
         do ix = 1, lx
            u(ix,iy) = uu(ix,iy)
         end do
      end do
```

他の分散並列プログラミング手法

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

OpenMP系

- 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はなく,データ転送も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

XcalableMP (XMP)

- 二つの並列プログラミングスタイルをサポートする新しい並列言語仕様である.
- グローバルビュー
  - データ並列プログラミングモデルとワークシェアにより,典型的な並列化をサポートする.
  - HPFと同等な機能／考え方である.
- ローカルビュー
  - 個別のノードを意識してプログラミングできるようPGAS機能を提供する.