OpenMPの基礎と簡単な並列計算法

2012年8月24日

京都大学 学術情報メディアセンター
牛島 省

はじめに

- 初めて OpenMP による並列化を試みる方を対象
- OpenMP (ver.2.5) を使う programming の初歩
  (細かいチューニング技法には触れない)
- 使用言語 : Fortran90/95
- その利用例 (京都大学のシステムを中心に)

What is OpenMP?

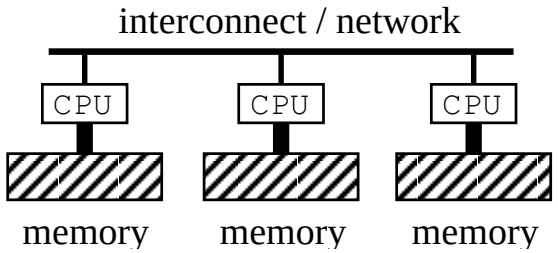
- The OpenMP ARB is the non-profit corporation that owns the OpenMP brand, oversees the OpenMP specification...
  http://openmp.org/wp/, OpenMP3.1 (July, 2011)
- OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs.
- 共有メモリ環境における並列演算,
  言語ではなく，指示文，関数，環境変数に対する仕様

参考書籍

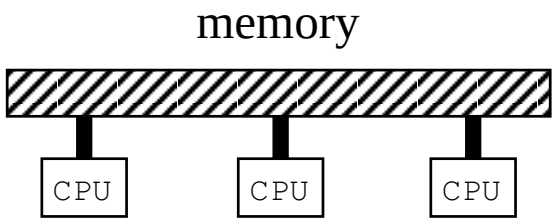
- **Using OpenMP, by Barbara Chapman, 2007, MIT Press.**
- **Parallel Programming in OpenMP (Paperback), by Rohit Chandra, etc., 2000, Morgan Kaufmann.**
- **Parallel Programming in C with MPI and OpenMP, by Michael J. Quinn, 2003, McGraw Hill.**
- **OpenMP 入門 マルチコア CPU 時代の並列プログラミング, 北山 洋幸, 2009, 秀和システム.**
- **C/C++プログラマーのための OpenMP 並列プログラミング, 菅原 清文, 2009, カットシステム.**
- **OpenMP による並列プログラミングと数値計算法, 牛島 省, 2006, 丸善. (Fortran, 丸善 HP より download 可, ver.2.5)**

共有・分散メモリシステム

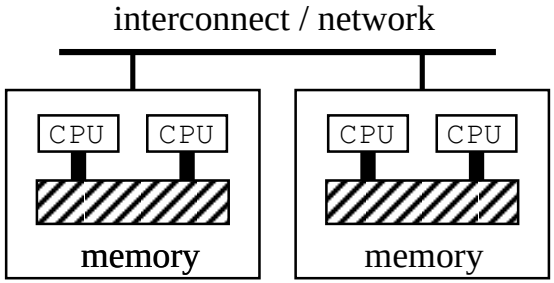
- 典型的なシステムの例 (図中の 1 CPU=1 core と考えて下さい)



(a) distributed mem. sys.



(b) shared mem. sys.

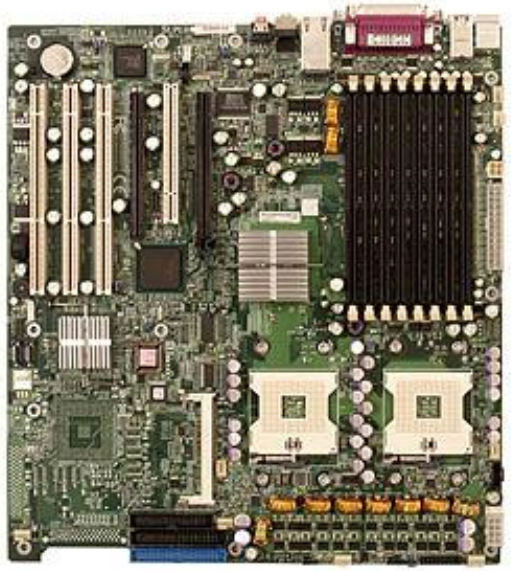


(c) Distributed shared mem. sys.

- NUMA (non-uniform memory access) : network でメモリ共有
  (アクセス速度数倍異なる、data locality が重要)

最も簡単な共有メモリシステムの一例 (7,8 年前)

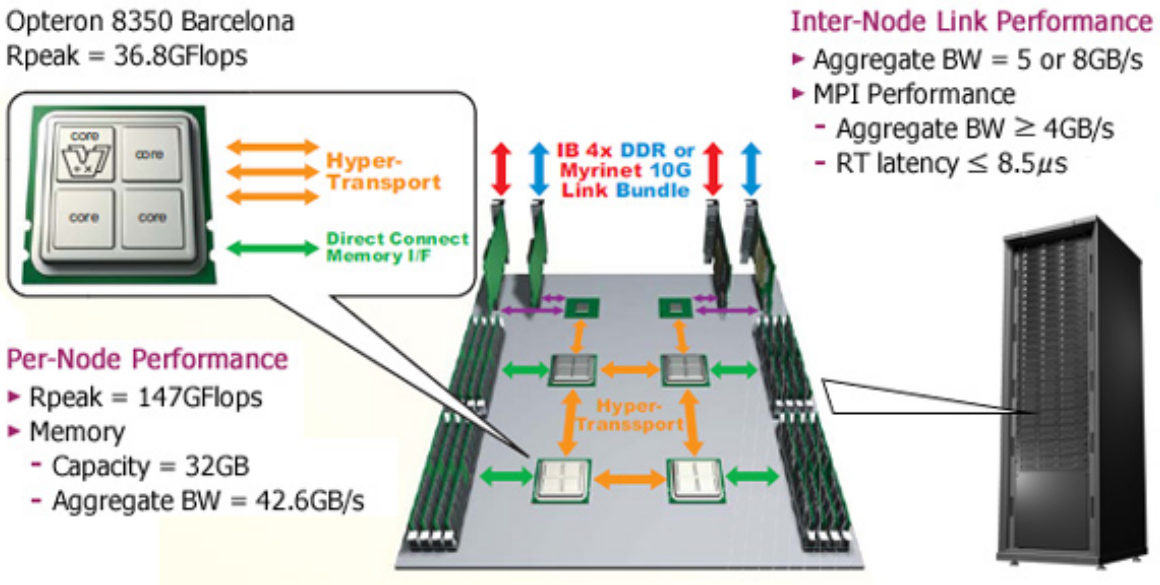
(a) outlook



(b) mother board

most simple SMS (dual CPU PC)

最近のシステムの例（その１）



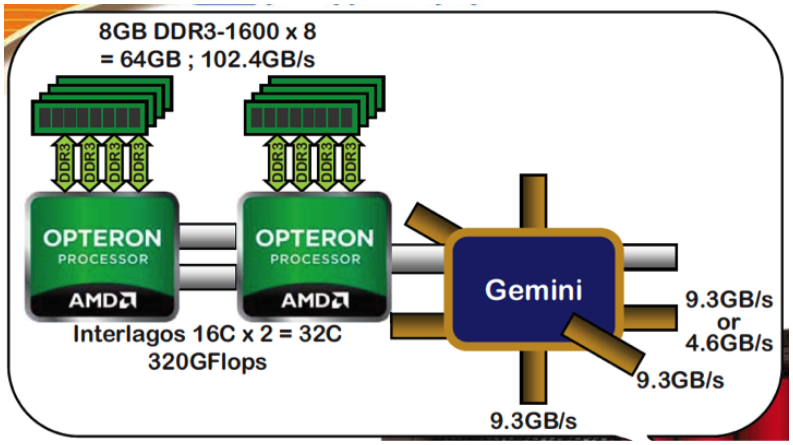
HX600 (Thin) in Kyoto Univ.

4 cores × 4 processors × 416 nodes

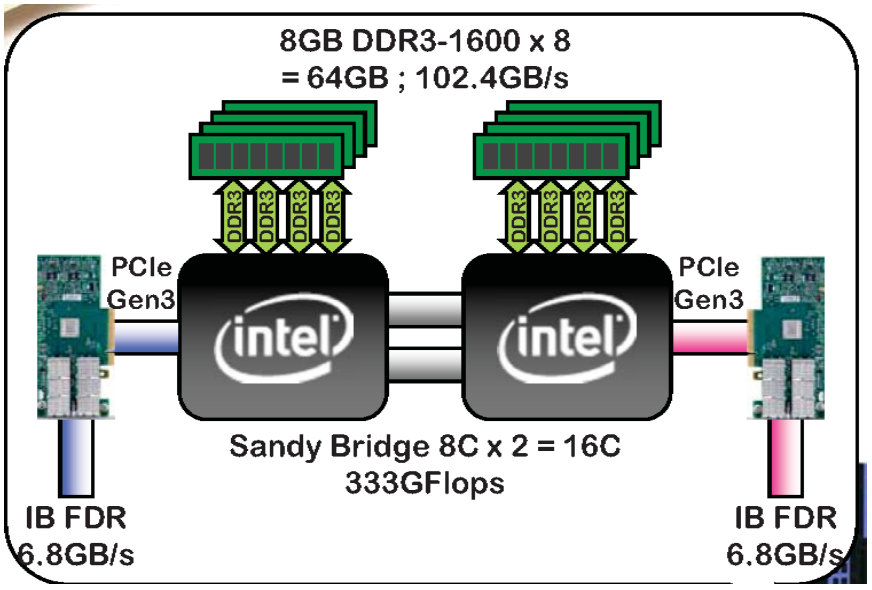
= 6,656 cores (61.2 TFLOPS)

- 京大の旧システム (until 2012/3)

最近のシステムの例（その２）



(a) 京大 Sys.A (32 Cores × 940 Nodes)



(b) 京大 Sys.B (16 Cores × 601 Nodes)

- 京大の新システム (since 2012/5)
- 現在ではマルチコアコアプロセッサが一般に使われているので、
  数倍から十数倍程度の高速化は OpenMP で簡単に行える

計算機と並列化手法のまとめ

計算機	分散メモリシステム	共有メモリシステム
並列処理の形態	プロセス並列	スレッド並列
プログラミングツール	MPI	OpenMP (+ MPI)

- 分散メモリ環境では、MPI による通信が必要
- 共有メモリ環境では、OpenMP による並列化が可能
- 共有メモリ環境では、MPI による並列化も可能
- 分散共有メモリ (DSM) 環境では、MPI と OpenMP を
  組み合わせたハイブリッド並列が有効な場合もある

MPI か OpenMP か? (両方か?)

- 利用できる計算機環境は?
- OpenMP では，逐次計算プログラムから
  並列演算プログラムへの書き換えが簡単
- 書き加える部分が少なく、説明用 program が
  スライド１枚に収まりやすい！ (MPI だと...)
- 簡単に (小規模な) 並列演算を行うには，OpenMP
- 大規模な並列演算を行うには，MPI
- MPI と OpenMP の両方を組み合わせること
  (hybrid 並列) も可能

OpenMP による並列計算に必要なもの

- OpenMP による並列演算を行うために，何が必要か?

  1) マルチコアプロセッサ or 共有メモリシステム
     → OK

  2) OpenMP 仕様に compliant なコンパイラ
     → OK (無ければ gfortran 4.2 or later)

  3) OpenMP 仕様で書かれたソースプログラム
     → 逐次計算 program を若干の変更すれば OK

OpenMP program の書き方入門

- **serial code**

```
program hello
  write(*, *) 'hello'
end program hello
```

↓　指示文 (directives) や関数 (routines) を入れる

- **parallelized code**

```
program hello
  !$ use omp_lib   ! declaration to use OMP module (F90)
  !$ call omp_set_num_threads(3) ! OMP routine
  !$OMP parallel   ! directive
  write(*, *) 'hello'
  !$OMP end parallel  ! directive
end program hello
```

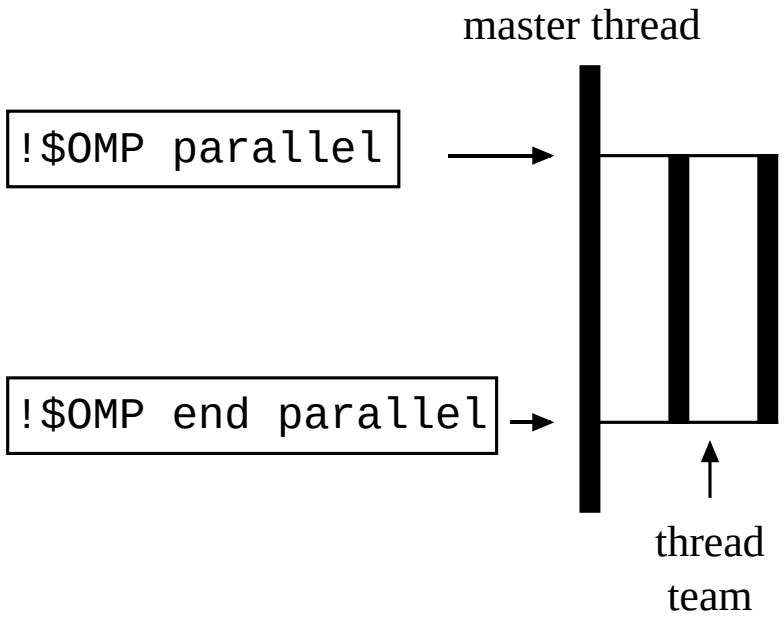
compile と指示文

- compile 時に option を付ける
  例:

```
gfortran -fopenmp source.f90 # gnu fortran (free)
ifort -openmp source.f90 # Intel fortran
```

- 指示文 (directive) : !$OMP で始まる文
- 条件付き compiling : 行頭に !$がある文
- Fortran では, “!” で始まる実行文は,
  コメント文として無視される (逐次 compile の場合)

parallel region と fork-join model



- 計算は逐次処理 (thread 数=1) として start
- “thread team” が “parallel directive” で形成 (fork)
- “thread team” が “end parallel directive” で消滅 (join)
- これを fork-join model という
- 並列計算は，“parallel region” 内で行われる

共有 (shared) 変数と private 変数の利用例

```
program shared_private
  !$ use omp_lib
  integer :: is = -1, ip = 100
  write(*, *) 'init val: is, ip = ', is, ip
  !$ call omp_set_num_threads(2) ! OpenMP routine
  !$OMP parallel private(ip)     ! <- private clause(節)
    !$ ip = omp_get_thread_num() ! OpenMP routine
    write(*, *) 'p-region: is, ip = ', is, ip
  !$OMP end parallel
  write(*, *) 's-region: is, ip = ', is, ip
end program shared_private
```

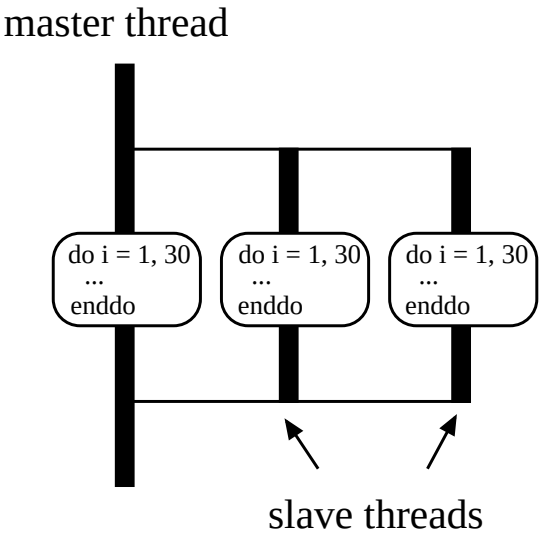
↓

```
 init val: is, ip =   -1     100
 p-region: is, ip =   -1       0
 p-region: is, ip =   -1       1
 s-region: is, ip =   -1     100
```

共有 (shared) 変数と private 変数の性質

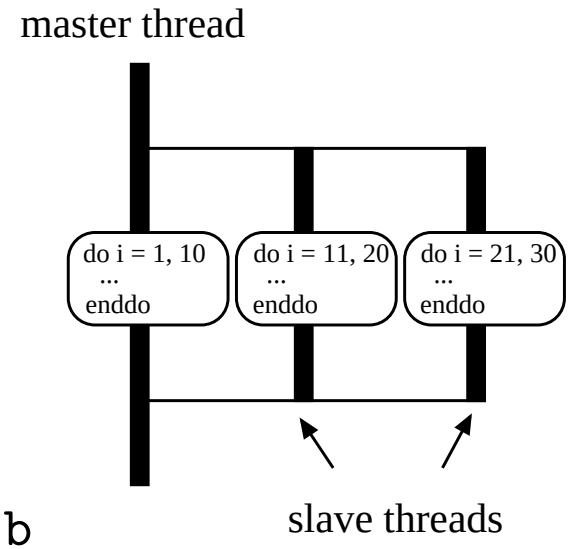
- private 変数 = accessible from only one thread
- 共有 (shared) 変数 = accessible from all threads
- default は “共有変数”
  (exceptions: loop counter, etc.)
- fistprivate 節や lastprivate 節の利用

Parallel region と loop 演算




```
program ws1
  !$ use omp_lib
  !$ call omp_set_num_threads(3)
  !$OMP parallel     ! start parallel region
  do i = 1, 30
     ...
  enddo
  !$OMP end parallel ! end parallel region
end program ws1
```

Work sharing (do directive) による並列化




```
program ws2
  !$ use omp_lib
  !$ call omp_set_num_threads(3)
  !$OMP parallel      ! start parallel region
    !$OMP do          ! <- work sharing
    do i = 1, 30
       ...
    enddo
    !$OMP end do      ! <- optional (省略可)
  !$OMP end parallel ! end parallel region
end program ws2
```

Work sharing 構文

- parallel region 内で，スレッドが行う演算を制御
- loop 構文は，その直下の do loop にのみ有効
  (多重 loop では，2 番目以降の loop は並列化されない)
- loop 構文では，!$OMP end do または enddo で同期
  (!$OMP end do 省略時には enddo で暗黙の同期)
- 同期を解除したいときには，nowait 節を使う
- loop 内の演算のスレッドへの割当には schedule 節を
  利用可 (eg. 三角行列とベクトルの積の計算など)
- loop 構文の他に，sections 構文，single 構文，
  workshare 構文がある

loop 構文と single 構文の使用例

```
program loop
  !$ use omp_lib
  integer i, j, a(2, 2), n
  !$ call omp_set_num_threads(2)
  !$OMP parallel
    !$OMP single
    read(*, *) n      ! performed by single thread
    !$OMP end single ! implied barrier
    !$OMP do
    do j = 1, n       ! this loop is parallelized
       do i = 1, n    ! this is NOT parallelized
          a(i, j) = 1
       enddo
    enddo             ! implied barrier
    !$OMP enddo       ! <- this can be omitted (optional)
  !$OMP end parallel
  write(*, *) a(:, :)
end program loop
```

reduction 演算 (合計を求める演算)

```
...
integer :: sum = 0, i
!$ call omp_set_num_threads(2)
!$OMP parallel
  !$OMP do
  do i = 1, 2
     sum = sum + i  ! wrong, data race occurs !!!
  enddo
!$OMP end parallel
...
```

↓ “reduction clause(節)” is needed to prevent data race

```
...
!$OMP parallel
  !$OMP do reduction(+ : sum) ! reduction clause
  do i = 1, 2
     sum = sum + i  ! OK, no data race occurs
  enddo
!$OMP end parallel
...
```

reduction 節

- reduction(オペレータ : 変数名リスト)
- オペレータは，+，−，∗ の他，いくつかのものがある
- 複数の変数を記述するときには，カンマで区切る
- 演算順序の変化
  round-off error のため，結果が異なる場合あり

thread 演算の同期と制御 (critical 構文)

```
...
sum = 0
!$ call omp_set_num_threads(3)
!$OMP parallel private(i, psum) shared(sum)
  ! thread 内の和 psum を求める
  psum = 0
  !$OMP do
  do i = 1, 100
     psum = psum + i
  enddo
  ! 全体の和 sum を求める
  !$OMP critical ! critical region では全 thread が逐次計算
    sum = sum + psum
  !$OMP end critical
!$OMP end parallel
write(*, *) 'sum = ', sum
...
```

thread 演算の同期と制御 (barrier 指示文)

```
...
!$OMP parallel private(priv_info) shared(eps)
do itr = 1, nitr
   ! (ここで収束計算等が行われ，誤差 eps が計算される)
   if (eps < threshold) then ! 誤差としきい値の比較
      !$ write(*, *) 'info = ', priv_info
      !$OMP barrier     ! barrier 指示文 (すべてのスレッドが
      stop 'converged' ! 出力を行ってから終了する)
   endif
   ...
   !$OMP barrier  ! barrier 指示文 (反復計算の同期を取る)
enddo
!$OMP end parallel
...
```

同期と制御を行うその他の方法

- ordered 構文 = 逐次計算の順序どおりの処理
- master 構文 = master thread のみが処理を担当
- atomic 指示文 = 単一の命令文に対する critical 処理
- その他

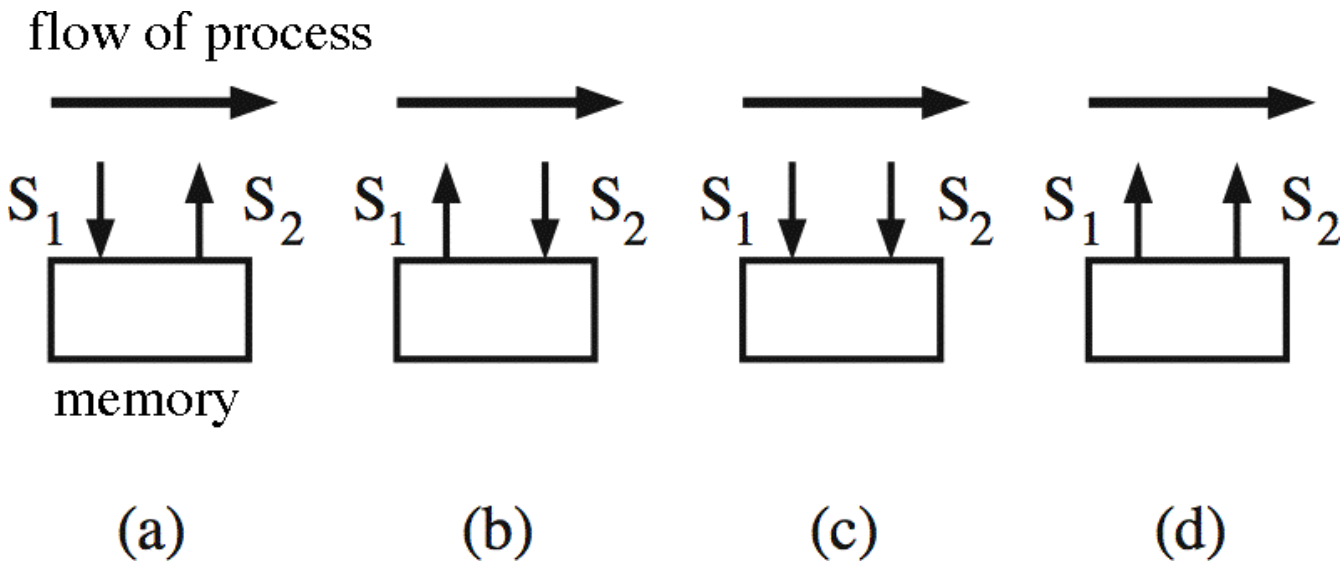
結合 worksharing 構文

```
...
!$OMP parallel  ! これと
  !$OMP do      ! これをまとめて記述
  do i = 1, 100
    ...
  enddo
!$OMP end parallel
...
```

↓

```
...
!$OMP parallel do  ! まとめて書いたもの
  do i = 1, 100
    ...
  enddo
!$OMP end parallel do ! 省略可
...
```

Data dependencies (データ依存性)



(a) フロー依存性, (b) 反依存性
(c) 出力依存性, (d) 依存性なし

- data race とも言われる

データ依存性のある演算例 (フロー依存性)

```
...
do i = 2, 3
   x(i) = x(i) + x(i - 1)
enddo
...
```

↓

```
x(2) = x(2) + x(1) ! S1
x(3) = x(3) + x(2) ! S2
```

- このままでは並列化できない!

データ依存性のある演算例 (反依存性)

```
...
do i = 1, n - 1 ! n は 3 以上
   a(i) = a(i) + a(i+1)
enddo
...
```

↓ 並列化

```
!$OMP parallel do
do i = 1, n - 1
   a2(i) = a(i+1)
enddo
!$OMP parallel do
do i = 1, n - 1
   a(i) = a(i) + a2(i)
enddo
```

- このように，copy を作っておく，という手があるが...

データ依存性のある演算例 (出力依存性)

```
do i = 2, n - 1
   d = f(i-1) + f(i+1)
   fn(i) = f(i) + d
enddo
e = d * dt ! 最後に書き込まれた d を使う演算
...
```

↓

```
!$OMP parallel do lastprivate(d)
do i = 2, n - 1
   d = f(i-1) + f(i+1)
   fn(i) = f(i) + d
enddo
e = d * dt
...
```

Quiz

- Is it possible to parallelize the following codes
  by simply putting do-directives ?

```
do i = 1, n
   z(i) = a * x(i) + y(i)
enddo

do i = 2, n
   y(i) = x(i) - x(i - 1)
enddo

do i = 1, n
   x(i) = a * x(i) + y(i)
enddo
```

Quiz (2)：行列ベクトル積

- $\boldsymbol{y} = A\boldsymbol{x}$ の計算
- Is it OK?

```
!$OMP parallel
!$OMP do
do i = 1, n
   y(i) = 0.0d0
   do j = 1, n
      y(i) = y(i) + a(i, j) * x(j)
   enddo
enddo
!$OMP end parallel
```

Ans. (2) : 行列ベクトル積

- reduction 演算が行われる内側の loop は
  並列化されないので，大丈夫 !!

データ依存性を除去して並列化しよう!

```
do j = 2, n
   do i = 1, n
      x(i, j) = x(i, j) + x(i, j - 1) ! data race in j-dir.
   enddo
enddo
```

↓

```
!$OMP parallel
!$OMP do
do j = 2, n
   do i = 1, n
      x(i, j) = x(i, j) + x(i, j - 1) ! これは誤り !!
   enddo
enddo
!$OMP end parallel
```

方策 (その 1)

```
do j = 2, n
   !$OMP parallel ! parallelize inner loop
   !$OMP do
   do i = 1, n
      x(i, j) = x(i, j) + x(i, j - 1)
   enddo
   !$OMP end parallel
enddo
```

- OK，ですが，フォーク・ジョインの負荷が...

方策 (その 2)

```
!$OMP parallel
!$OMP do
do i = 1, n      ! exchange i- and j-loops
   do j = 2, n
      x(i, j) = x(i, j) + x(i, j - 1)
   enddo
enddo
!$OMP end parallel
```

- ... 推奨しませんが (メモリアクセスが悪化)...

方策 (その 3)

is, ie の設定は済ませておく

```
...
!$OMP parallel private(id) ! SPMD-like code
id = omp_get_thread_num() + 1
do j = 2, n
   do i = is(id), ie(id)
      x(i, j) = x(i, j) + x(i, j - 1)
   enddo
enddo
!$OMP end parallel
```

- やや技巧的?

OpenMP プログラムのさらなるチューニング

- first touch (data locality)
- キャッシュ競合 (false sharing) の回避
- 負荷バランスの均等化
- プロファイラの利用

応用計算１：緩和計算 or 拡散方程式の差分式

- 拡散方程式の差分式
- あるいは、GS, SOR 法などの緩和計算 (omg = 加速係数)

```
...
do j = 2, n - 1
   do i = 2, n - 1
      rhs = 0.25d0 * (f(i-1, j  ) + f(i+1, j  ) &
                    + f(i  , j-1) + f(i  , j+1))
      f(i, j) = f(i, j) + omg * (rhs - f(i, j))
   enddo
enddo
...
```

- データ依存性のため、演算過程が重要な場合には、
  このままでは並列化できない
- 更新前後の値を異なる配列に格納しておけば
  OpenMP で並列化可能 (GS → Jacobi)

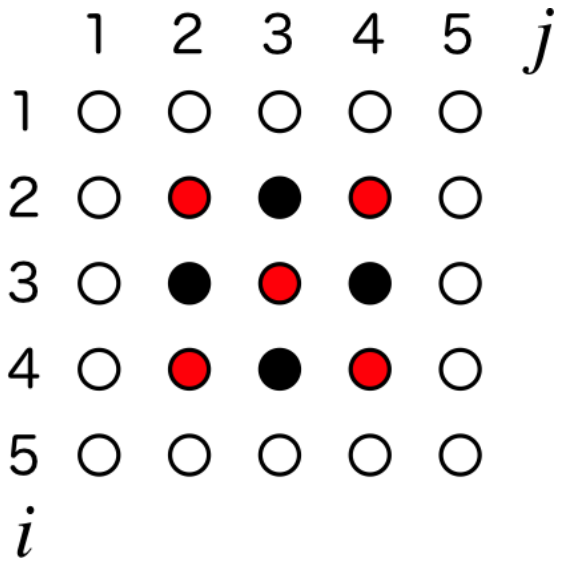
odd-even (red-black) 法

- 効率的な演算法 (逐次計算でも高速化)

```
!$OMP parallel private(rhs)
do itr = 1, nitr
   !$OMP do
   do j = 2, n - 1
      do i = 2 + mod(j  , 2), n - 1, 2
         rhs = 0.25d0 * (f(i-1, j  ) + f(i+1, j  ) &
                       + f(i  , j-1) + f(i  , j+1))
         f(i, j) = f(i, j) + omg * (rhs - f(i, j))
      enddo
   enddo
   !$OMP do
   do j = 2, n - 1
      do i = 2 + mod(j+1, 2), n - 1, 2
         rhs = 0.25d0 * (f(i-1, j  ) + f(i+1, j  ) &
                       + f(i  , j-1) + f(i  , j+1))
         f(i, j) = f(i, j) + omg * (rhs - f(i, j))
      enddo
   enddo
   ...
```

odd-even (red-black) 法の演算手順

- ２つの loop で、下記の赤、黒格子の値が順に計算される
- 各 loop 内の演算が OpenMP により並列化される



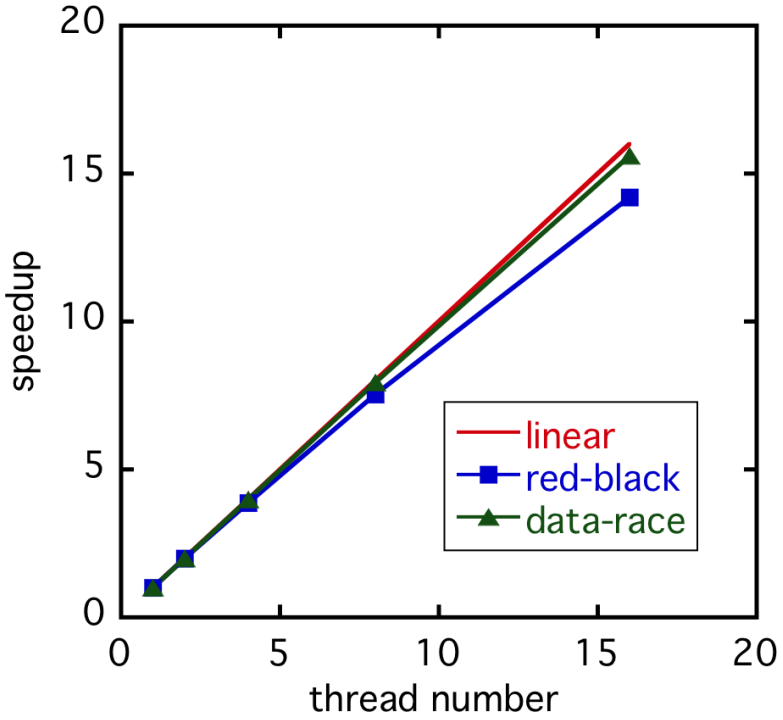
- 赤が計算されてから、その値を使って黒が計算されるので、
  厳密には依存性は除去されていない

データ依存性を無視して収束解を得る方法

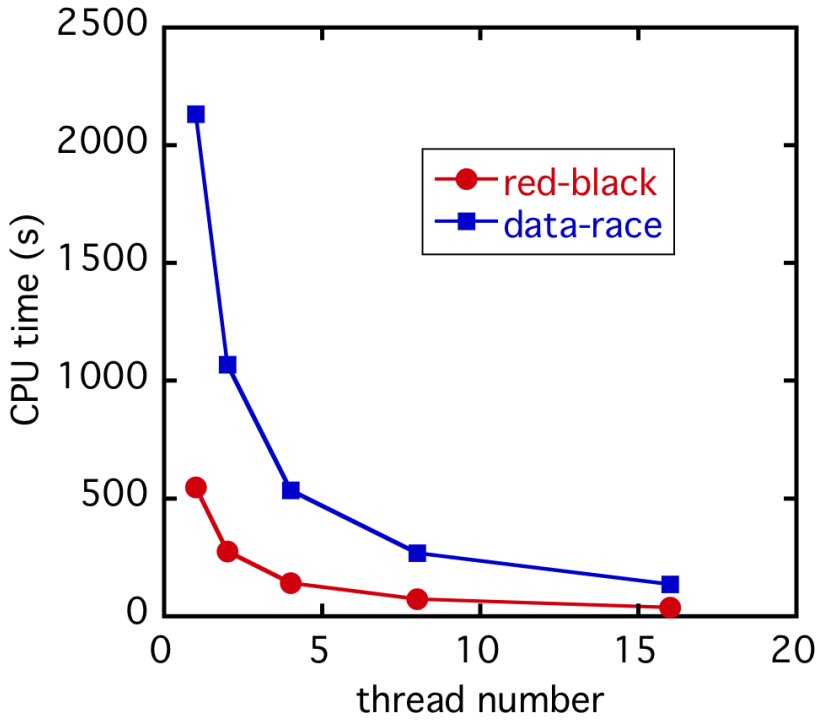
```
!$OMP parallel private(rhs)
do itr = 1, nitr
   !$OMP do
   do j = 2, n - 1
      do i = 2, n - 1
         rhs = 0.25d0 * (f(i-1, j  ) + f(i+1, j  ) &
                       + f(i  , j-1) + f(i  , j+1))
         f(i, j) = f(i, j) + omg * (rhs - f(i, j))
      enddo
   enddo
   ...
```

- 途中結果は依存性を除いた手法と異なるが、
  最終結果は同様になる

OpenMP を利用する緩和計算の例 (sys.B)



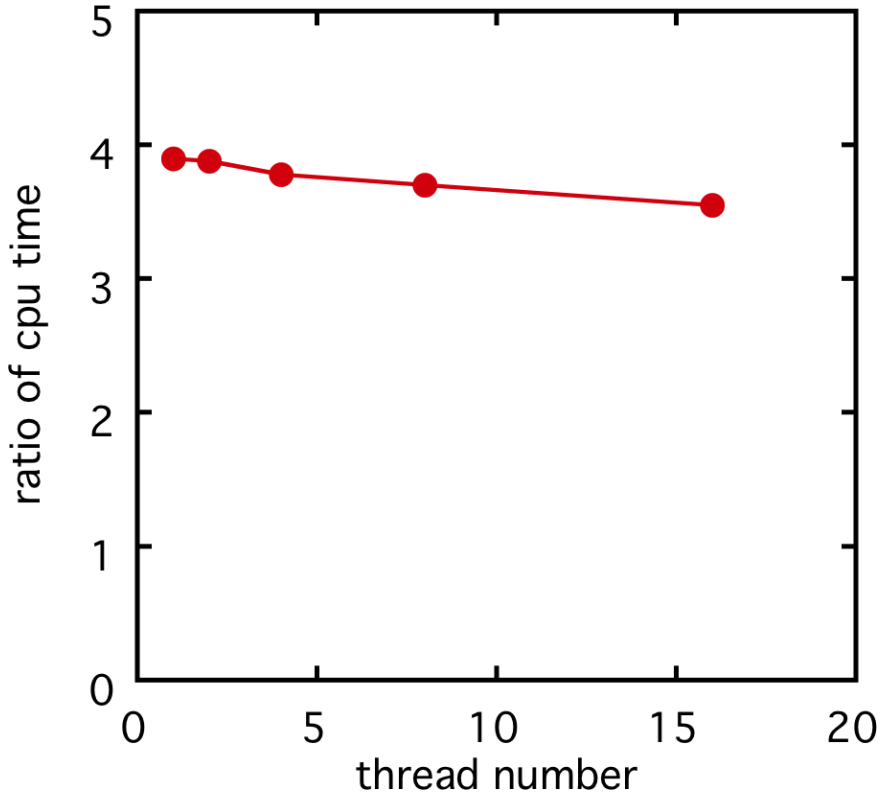
(a) スピードアップ



(b) 計算時間

- 格子数 $1,000 \times 1,000$、残差しきい値 1.0×10^{-10}、omg=1.5
- 反復回数は両手法でほぼ同じ (約 170,000 回)

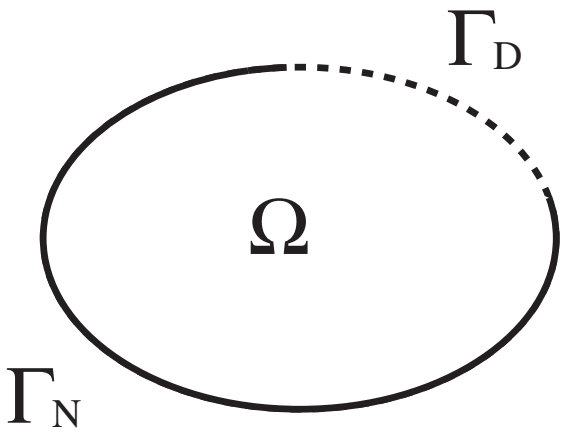
odd-even と依存性を無視した計算時間の比



- thread 数にかかわらず、上記の計算条件では 4 倍弱程度高速

応用計算２：FEM によるラプラス方程式の計算

- ２次元領域における境界値問題



$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0$$

$$\phi = \phi_D \quad [\text{on } \Gamma_D] \,, \qquad \frac{\partial\phi}{\partial n} = u_n \quad [\text{on } \Gamma_N]$$

離散化と代表的な計算方法

- 重み付き残差法 (ガラーキン法)、三角形 1 次要素
- 各要素に対する関係式 (ϕ_m は節点上の変数，Γ_D 含む)

$$[K]_e \begin{bmatrix} \phi_i \\ \phi_j \\ \phi_k \end{bmatrix} - [B]_e \begin{bmatrix} U_i \\ U_j \\ U_k \end{bmatrix} = \mathbf{0}$$

- 全体を構成する $A\boldsymbol{x} = \boldsymbol{b}$ から数値解が得られる
- CG 系解法で必要な行列ベクトル積 $A\boldsymbol{x}$ の代表的な計算方法
  - 各要素の関係式を重ね合わせた A を使って計算 (CRS 法)
  - A を求めず、要素ベースで計算し、節点の和 (EBE 法)
  - A を求めず、節点ベースで順に計算していく

element-by-element 法 (要素ベースの計算法)

- EBE 法による要素ベースの行列ベクトル積 $A\boldsymbol{x}$ の計算部分

```
...
do ne = 1, nelm  ! nelm は全要素数
   do nt = 1, 3  ! 三角形要素の各頂点にわたるループ
      nd = ind(ne, nt)  ! 節点番号 nd を取得
      s(nd) = s(nd) &   ! s は A と x の積を格納するベクトル
            + a(1, nt, ne) * x(inode(1, ne)) &
            + a(2, nt, ne) * x(inode(2, ne)) &
            + a(3, nt, ne) * x(inode(3, ne))
   enddo
enddo
...
```

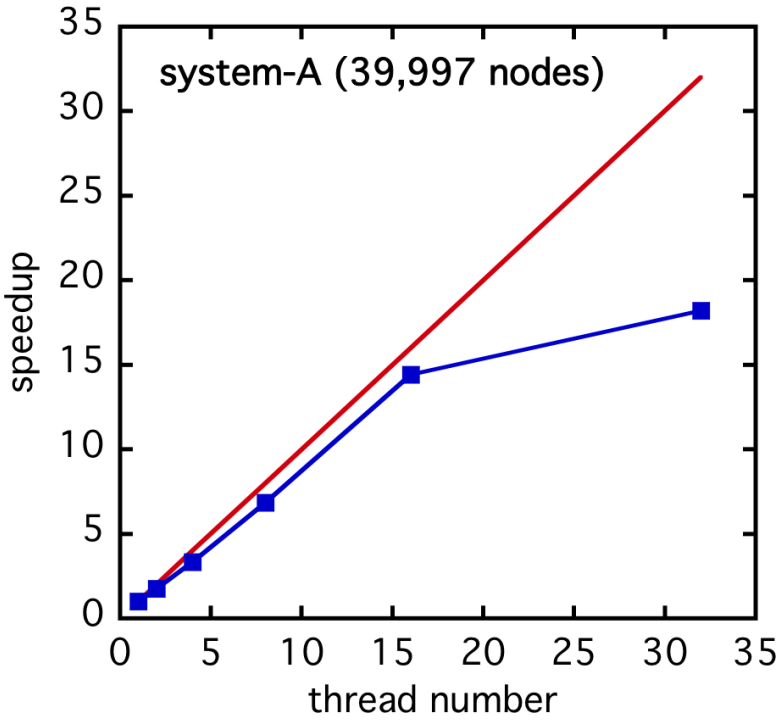
- 全体マトリクスが不要、CG 法系の解法と組み合わせて利用
- 分散メモリシステム +MPI で使われる場合がある
- 共有メモリ (OpenMP) ではデータ依存性のため、
  指示文を挿入して並列化することはできない

節点ベースの解法と OpenMP による並列化

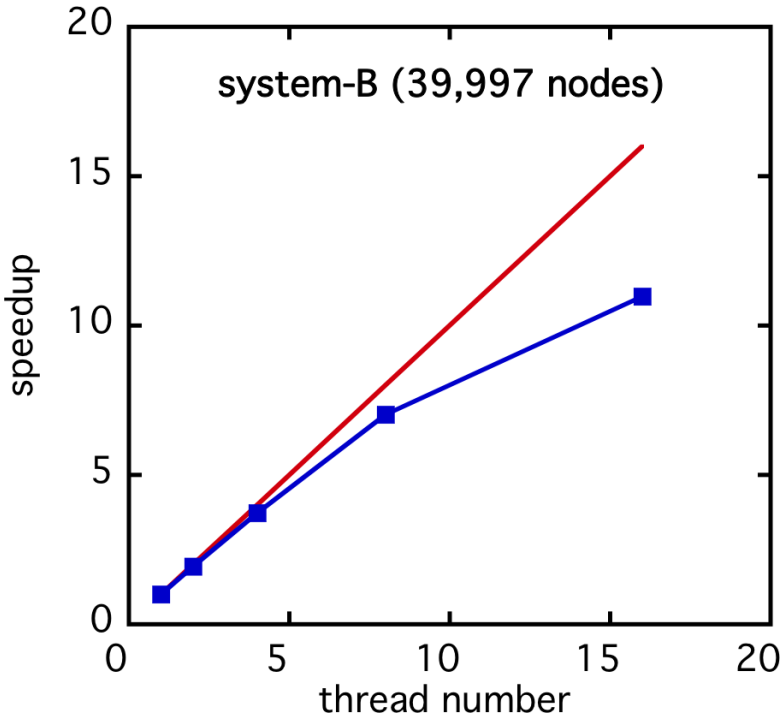
- 節点のループを OpenMP により並列化して $A\boldsymbol{x}$ を計算

```
...
!$OMP parallel private(ne, nt, ni) ! パラレル構文開始
!$OMP do
do nd = 1, node_cmp    ! node_cmp は計算対象となる総節点数
   ni = inode_cmp(nd) ! 計算対象となる節点番号 ni を取得
   s(ni) = 0.0d0
   do i = 1, elm_no(ni) !elm_no(nd) は節点 nd を含む総要素数
      ne = elm_id(i, ni) !要素番号 ne を取得
      nt = elm_tr(i, ni) !三角形の頂点の番号 nt(=1-3) を取得
      s(ni) = s(ni) &     !s は A と x の積を格納するベクトル
            + a(1, nt, ne) * x(inode(1, ne)) &
            + a(2, nt, ne) * x(inode(2, ne)) &
            + a(3, nt, ne) * x(inode(3, ne))
   enddo
enddo
...
```

OpenMP+節点ベースFEMの計算例



(a) system A (Cray compiler)



(b) system B (Intel compiler)

- 前処理無し CG 法を利用
- 上記の例では節点数が少ない (約 4 万節点)

応用計算３：OpenMP による DFT の計算

- 離散フーリエ変換 (DFT: discrete Fourier transform)

$$\tilde{a}_k = \frac{1}{N}\sum_{j=0}^{N-1} a_j e^{-ikx_j} = \frac{1}{N}\sum_{j=0}^{N-1} \omega_N^{kj} a_j \qquad (k = -\frac{N}{2}, \cdots, \frac{N}{2} - 1)$$

  ω_N は回転因子と呼ばれることもある

- DFT の逆変換

$$a_j = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \tilde{a}_k e^{ikx_j} = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \omega_N^{-kj} \tilde{a}_k \qquad (j = 0, \cdots, N-1)$$

- 行列ベクトル積 $A\boldsymbol{x}$ (N^2 回のかけ算) に相当

thread 数による DFT 演算の分解

- データ数 N がスレッド数 P の倍数である場合を対象
- 下記のように剰余に基づきデータを P 個のグループに分ける

$$\begin{aligned}
\mathrm{mod}(k,P) = 0 \quad &: \quad k = 0,\ P,\ \cdots,\ N-P \\
&\vdots \\
\mathrm{mod}(k,P) = r \quad &: \quad k = r,\ P+r,\ \cdots,\ N-P+r \\
&\vdots \\
\mathrm{mod}(k,P) = P-1 \quad &: \quad k = P-1,\ 2P-1,\ \cdots,\ N-1
\end{aligned}$$

- 各グループのデータ数は N/P
  P 個のグループの演算を P 個のスレッドにそれぞれ割り当てる

thread に割り当てる演算

- DFT の一般的な演算 A_k

$$A_k = \sum_{j=0}^{N-1} \omega_N^{kj} a_j \qquad (k = 0, \cdots, N-1)$$

- これを以下のように表す $(k' = 0, \cdots, N/P-1;\ r = 0, \cdots, P-1)$

$$A_k = A_{Pk'+r} = \sum_{j=0}^{N-1} \omega_N^{(Pk'+r)j} a_j = \sum_{j=0}^{N-1} \omega_{N/P}^{k'j} \omega_N^{rj} a_j$$

$$= \cdots = \sum_{j=0}^{N/P-1} \omega_{N/P}^{k'j} \left[\omega_N^{rj} \sum_{m=0}^{P-1} \omega_P^{rm} a_{j+mN/P} \right] \equiv \sum_{j=0}^{N/P-1} \omega_{N/P}^{k'j}\ d_{j,r}$$

- $r = 0, \cdots, P-1$ のデータを各スレッドに割り当て、
  各スレッドは担当分のデータに対して FFT を行う

1 次元 Burgers 方程式

- 1 次元 Burgers 方程式, $u(x,t)$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad \nu = \text{const.} > 0$$

- 初期・境界条件 :

$$0 \le x \le 2\pi, \quad t \ge 0$$

$$u(x,0) = \sin x \quad \text{(初期条件)}$$

周期境界条件

- Cole-Hopf 変換により理論解が得られる

DFT を利用する近似式と残差

- 近似解を $u(x,t)$ とおく

$$u(x,t) = \sum_{m=-\frac{N}{2}}^{\frac{N}{2}-1} \tilde{u}_m(t)\ e^{imx}, \quad i = \text{虚数単位}$$

- Burgers 方程式に代入すると、残差 $R(x,t)$ は

$$R(x,t) = \sum_{m=-\frac{N}{2}}^{N-1} \frac{d\tilde{u}_m}{dt}\ e^{imx} + \sum_{r=-\frac{N}{2}}^{\frac{N}{2}-1} \sum_{s=-\frac{N}{2}}^{\frac{N}{2}-1} ir\tilde{u}_r\tilde{u}_s\ e^{ix(r+s)}$$

$$+ \nu \sum_{m=-\frac{N}{2}}^{\frac{N}{2}-1} m^2\tilde{u}_m\ e^{imx}$$

重み付き残差法

- 重み関数 e^{-ikx} を利用して、次式が成り立つように解を求める

$$\int_0^{2\pi} R(x,t)\ e^{-ikx}\ dx = 0$$

- 残差 $R(x,t)$ を上式に代入して次式を得る

$$\frac{d\tilde{u}_k}{dt} + \sum_{\substack{r,s=-\frac{N}{2} \\ r+s=k}}^{\frac{N}{2}-1} ir\tilde{u}_r\tilde{u}_s + \nu k^2\tilde{u}_k = 0$$

$\left(\right.$ 直交性を利用, $\int_0^{2\pi}\ e^{i(j-k)x}\ dx = 2\pi\delta_{j-k,0}.\ \left.\right)$

常微分方程式の離散化

- 通常 Runge-Kutta 等が用いられるが、
  ここでは簡単に Euler explicit を利用

$$\tilde{u}_k^{n+1} = (1 - \nu k^2 \Delta t)\tilde{u}_k^n - C_k^n \Delta t$$

  C_k = 2nd term on LHS.

- $\tilde{v}_r = ir\tilde{u}_r$, $\tilde{w}_s = \tilde{u}_s$ とおくと、
  非線形項は次のかけ算の和となる (N^2 回の乗算) :

$$C_k = \sum_{\substack{r,s=-\frac{N}{2} \\ r+s=k}}^{\frac{N}{2}-1} \tilde{v}_r \tilde{w}_s \qquad (k = -\frac{N}{2}, \cdots, \frac{N}{2} - 1)$$

C_k の効率的な計算（１）

- 逆 DFT により、$\tilde{v}_r$ と $\tilde{w}_s$ は、次のように変換される

$$v_j = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \tilde{v}_k e^{ikx_j} \ , \qquad w_j = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \tilde{w}_k e^{ikx_j}$$

ここに、$j = 0, \cdots, N-1$

- $v_j w_j$ に DFT を行うと、

$$\tilde{z}_k = \frac{1}{N} \sum_{j=0}^{N-1} v_j w_j e^{-ikx_j} = \sum_{\substack{r,s=-\frac{N}{2} \\ r+s=k}}^{\frac{N}{2}-1} \tilde{v}_r \tilde{w}_s + \sum_{\substack{r,s=-\frac{N}{2} \\ r+s=k\pm N}}^{\frac{N}{2}-1} \tilde{v}_r \tilde{w}_s$$

due to $\sum_{j=0}^{N-1} e^{ikx_j} = N \ (k = 0, \pm N, \pm 2N, \cdots), \ \ 0 \ (\text{otherwise}).$

C_k の効率的な計算（２）

- 前ページの RHS 第 1 項は C_k に相当するので、

$$C_k = \tilde{z}_k - \sum_{\substack{r,s=-\frac{N}{2} \\ r+s=k\pm N}}^{\frac{N}{2}-1} \tilde{v}_r \tilde{w}_s$$

- $\tilde{z}_k \leftarrow$ order of N multiplications and DFT
- 上式の RHS 第 2 項は aliasing error 項といわれる
- Pseudo-spectral method (neglecting 2nd term)

スペクトル法の outline

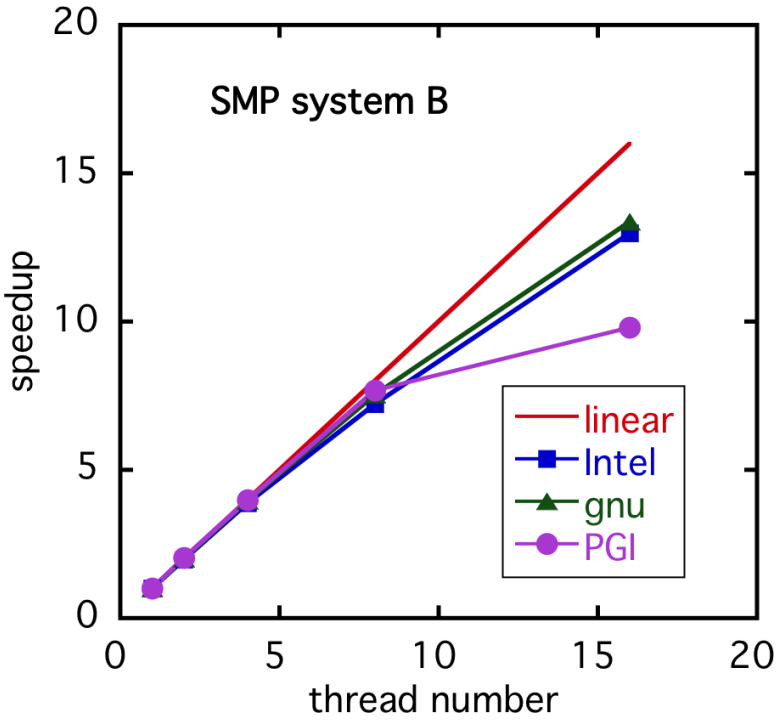
- In summary, in a spectral method,
  - Initial cond. : DFT for u to $\tilde{u}$
  - Unsteady comp. in W.N.S.
    1) IDFT for $\tilde{v}_r$ and $\tilde{w}_s$
    2) DFT for $v_j w_j$
    3) time integration for $\tilde{u}_k$ (EE, RK etc.)
  - Final procedure : IDFT for $\tilde{u}$ to u

京大 System B の各種コンパイラ

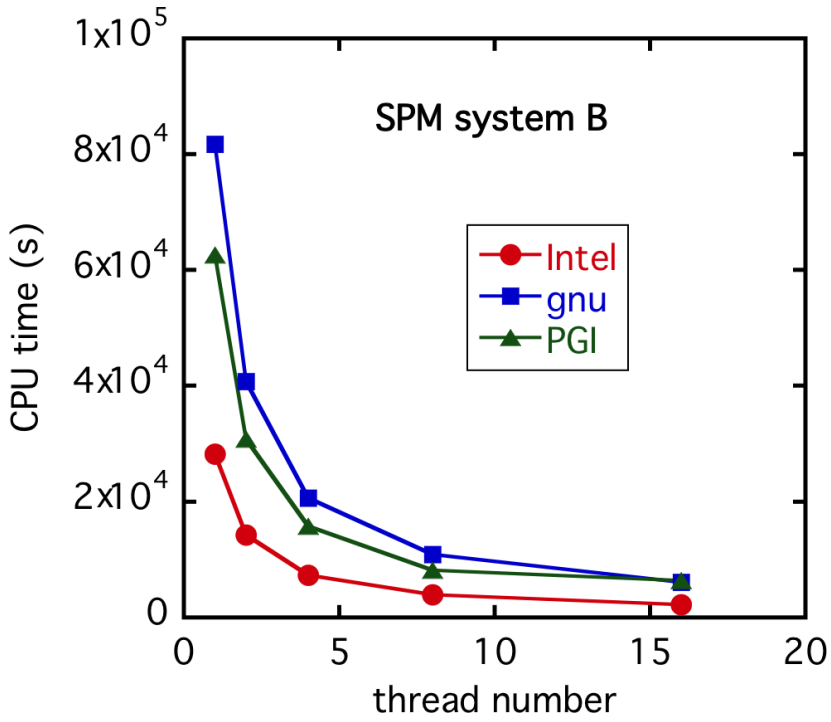
コンパイラ	コマンド	OpenMP	MPI
Intel 12.1	ifort	-openmp	対応
PGI 12.3	pgf95	-mp	非対応
GNU 4.4.6	gfortran	-fopenmp	非対応

- system B に login した時点で Intel, GNU は利用可能
- PGI コンパイラ使用前に module を load する必要有り：
  % module load pgi
- MPI を使用する場合は、Intel コンパイラのみ利用可能

OpenMP+再帰処理FFT (system B)



(a) speedup



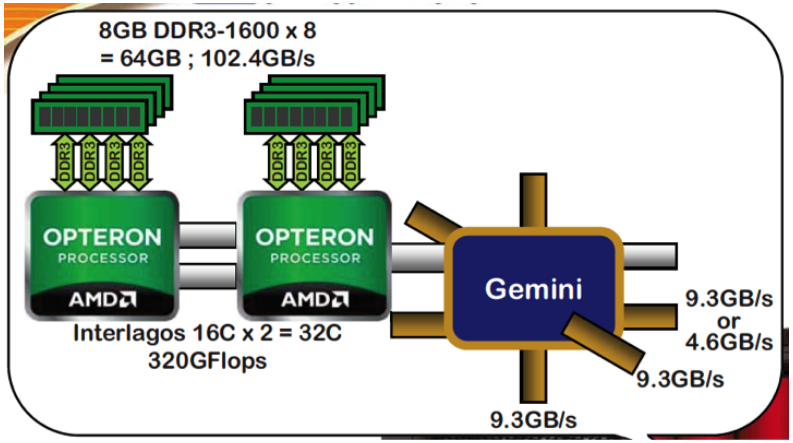
(b) CPU time

- スペクトル法による 1D-Burgers 方程式の計算 (6,400,000 格子)
- intel compiler が高速 (ifort -openmp -fast)

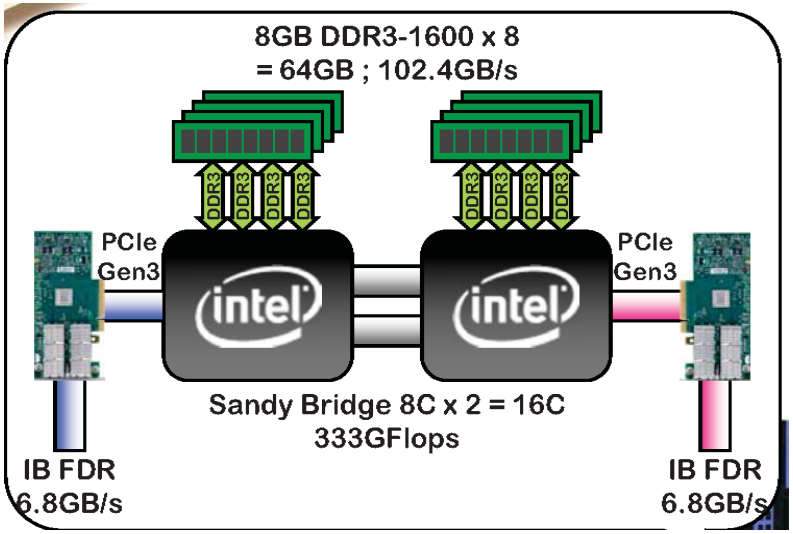
まとめ

- OpenMP により，共有メモリ system 上で，逐次計算 program を 簡単に 並列化可能
- programming 時には，データ依存性に注意
- 大規模な高速演算を行う場合には MPI が better だが、MPI+OpenMP という手もある

京大で募集中のサービス (2012/8/31 締切り)



(a) Sys.A (32 Cores × 940 Nodes)



(b) Sys.B (16 Cores × 601 Nodes)

1) グループコースの追加募集 (sys. A, B and C)

2) 若手研究者支援・プログラム高度化・大規模計算支援

3) 先端的大規模計算サービス (民間企業の方を対象に
利用負担金の半額をセンターが補助)