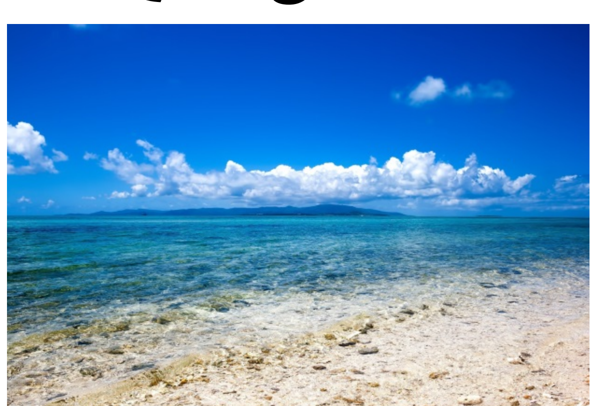
数値予報における超並列計算機利用の実際

気象庁予報部数値予報課
荒波　恒平

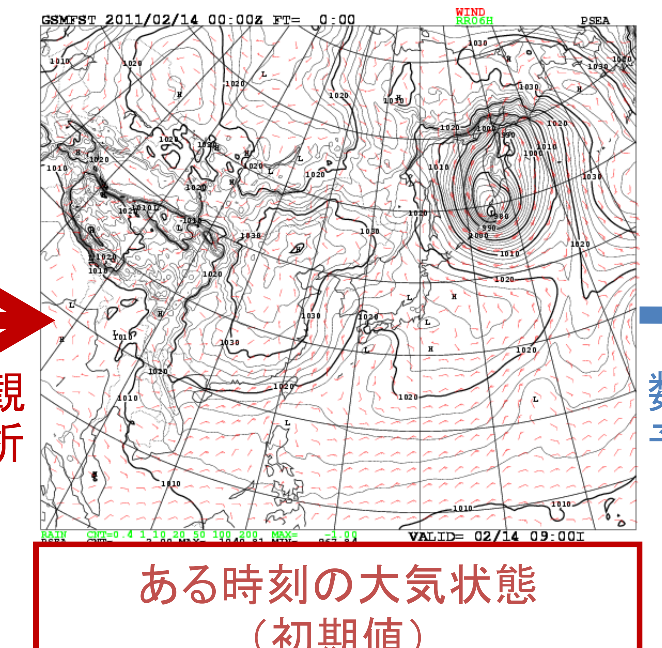
数値予報とは

- 観測値を基に、ある特定の時刻の大気の状態を数値的に解析し
- 流体力学や熱力学などの物理法則に基づいて、その大気状態を時間発展させ、将来の大気の状態を予測すること
- 気象庁の発表する防災気象情報、天気予報の基盤となっている



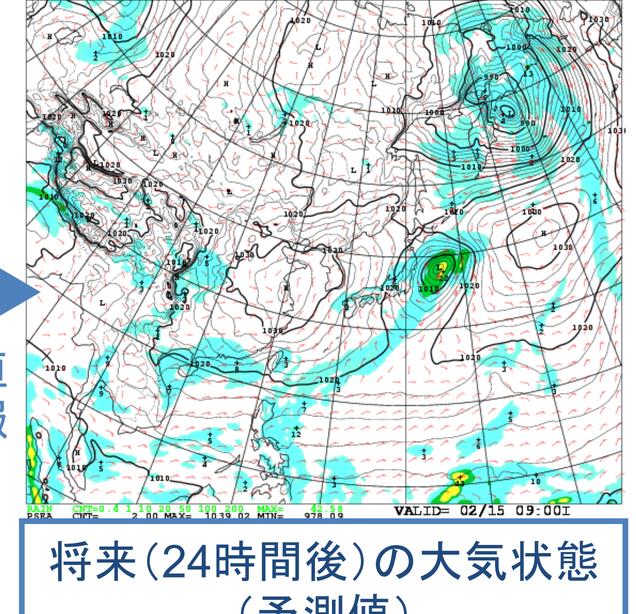
ある時刻の、大気の状態を表す要素（気圧,気温,水蒸気量,風向・風速など）の観測値

客観解析



ある時刻の大気状態（初期値）

数値予報



将来（24時間後）の大気状態（予測値）

実際の数値予報

大気の支配方程式:

非線型項(変数どうしのかけ算、たとえば移流項)を含むため、
一般には**解析的**に解けない → **数値的**に解く必要あり

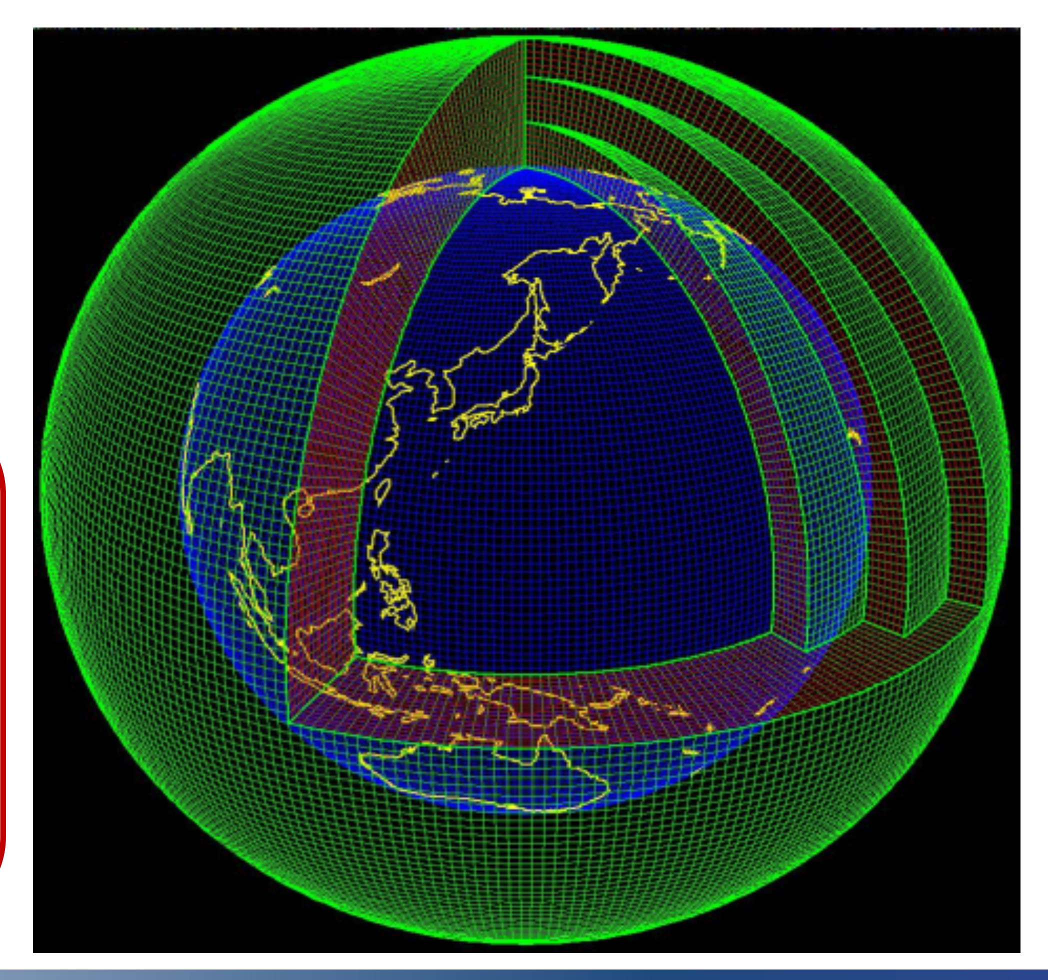
空間的、時間的離散化

数値表現される仮想的な大気
(モデル大気)で計算する

数値予報の主要コンポーネント

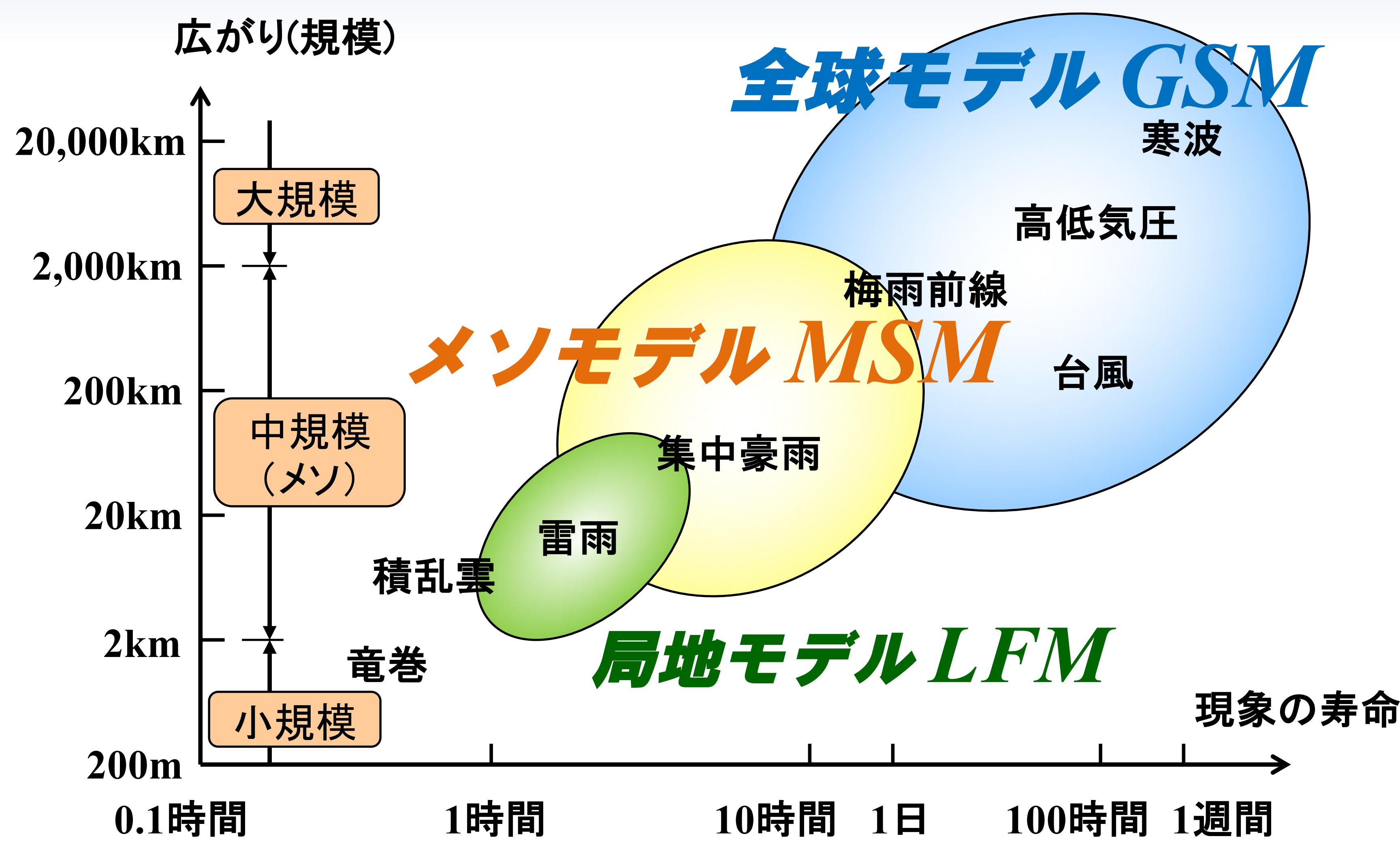
・初期の大気状態を解析
データ同化システム

・将来の大気状態を推定
数値予報モデル

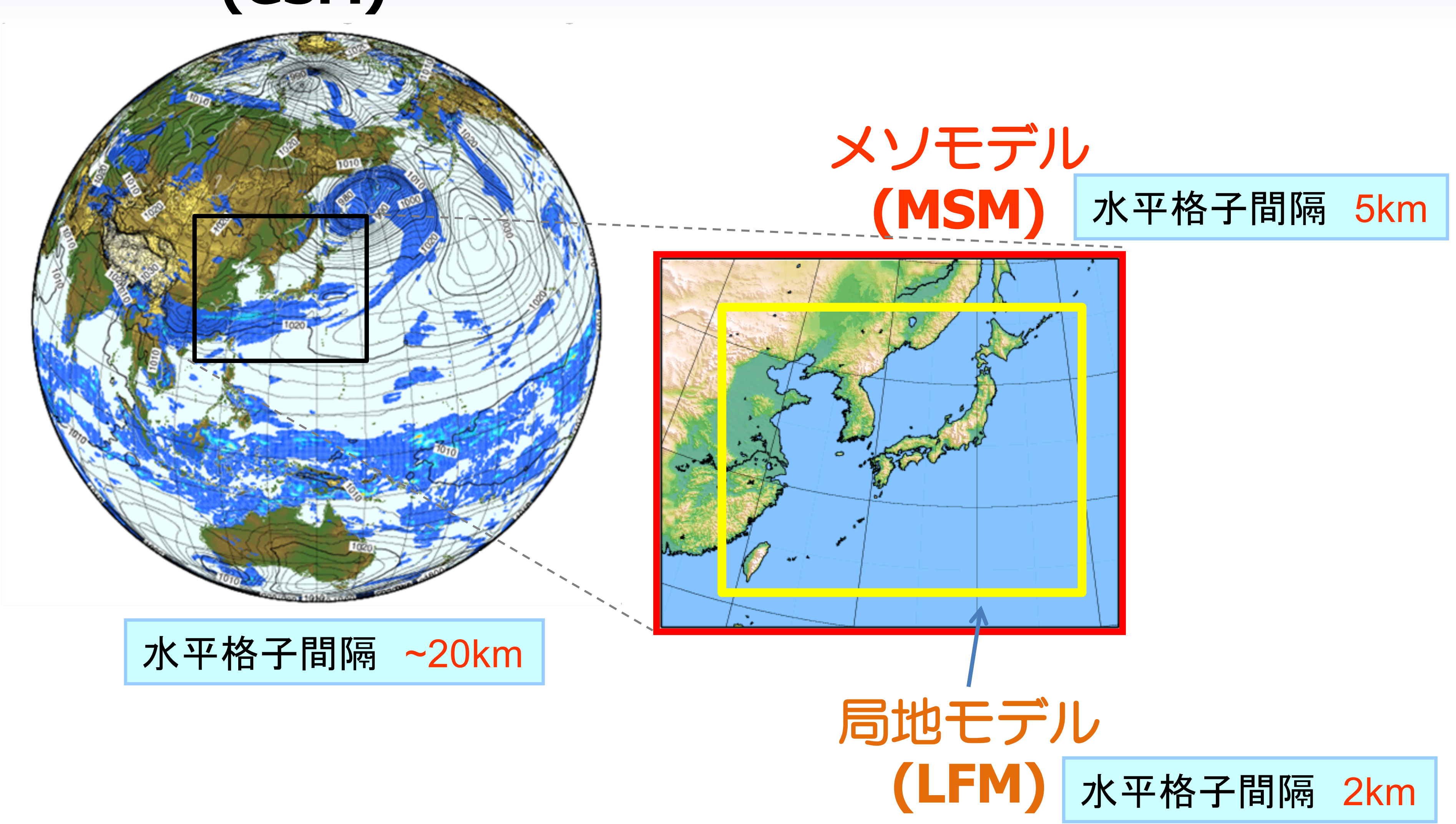


現象と予測可能性

予測可能な時間の限界：予測対象や初期状態によって異なる



気象庁の短期予報向け数値予報モデル



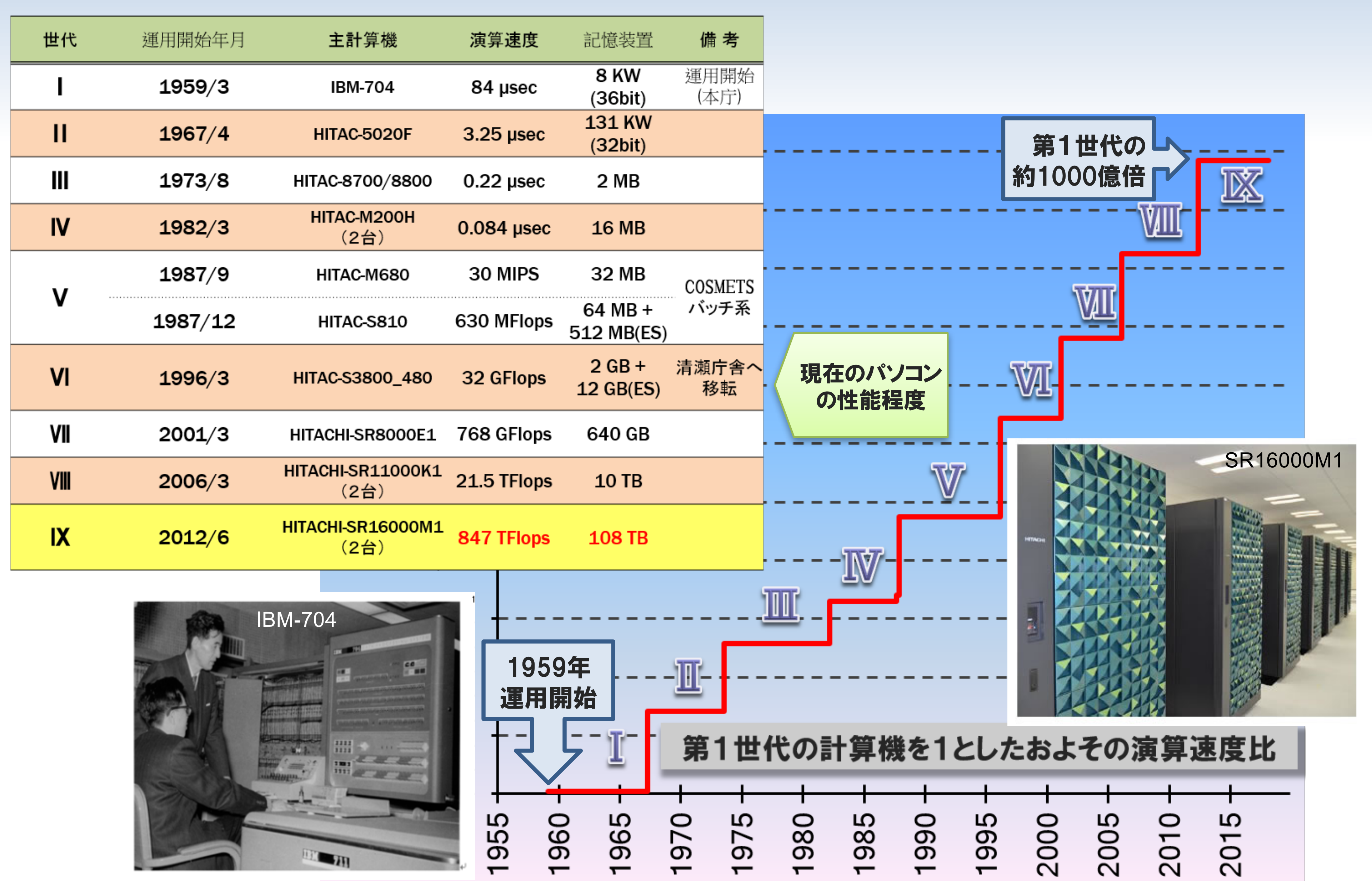
現業数値予報システムの仕様

短期予報用の大気モデルのみ、2016年10月現在

	モデル	領域 水平解像度	鉛直 層数	予報時間 （初期時刻（UTC））	目的
決定論的予報	全球モデル （GSM）	地球全体 約20km	100層	84 hours (*00,06,18*) 264 hours (*12*)	・各種予報・ガイダンスの基礎資料 ・各種数値予報モデルの入力値 （波浪，黄砂，移流拡散など） ・メソモデルの側面境界条件
	メソモデル （MSM）	日本周辺 約5km	50層 (48+2)	39 hours (*00,03,06,09,* *12,15,18,21*)	・各種予報の支援 （防災気象情報など） ・各種数値予報モデルの入力値 （波浪，高潮，火山灰移流拡散など） ・局地モデルの側面境界条件
	局地モデル （LFM）	日本周辺 約2km	58層	9 hours (毎時)	・各種予報の支援 （飛行場予報, 防災気象情報など）
確率論的予報	週間 アンサンブル予報 システム（WEPS）	地球全体 約40km	60層	264 hours (*00,12*) ※各27メンバー ～54メンバー/日	・週間天気予報 ・東南アジア等諸外国の予報支援
	台風 アンサンブル予報 システム（TEPS）	地球全体 約40km	60層	132 hours (*00,06,12,18*) ※25メンバー	・台風進路予報 ※台風発生が予想される時、 台風が存在する時等に実行される

※赤字は現スパコン導入（2012年6月）後に導入・更新された箇所

数値予報に用いる計算機の変遷



世代	運用開始年月	主計算機	演算速度	記憶装置	備 考
Ⅰ	1959/3	IBM-704	84 µsec	8 KW (36bit)	運用開始 (本庁)
Ⅱ	1967/4	HITAC-5020F	3.25 µsec	131 KW (32bit)	
Ⅲ	1973/8	HITAC-8700/8800	0.22 µsec	2 MB	
Ⅳ	1982/3	HITAC-M200H （2台）	0.084 µsec	16 MB	
Ⅴ	1987/9	HITAC-M680	30 MIPS	32 MB	COSMETS バッチ系
	1987/12	HITAC-S810	630 MFlops	64 MB + 512 MB(ES)	
Ⅵ	1996/3	HITAC-S3800_480	32 GFlops	2 GB + 12 GB(ES)	清瀬庁舎へ移転
Ⅶ	2001/3	HITACHI-SR8000E1	768 GFlops	640 GB	
Ⅷ	2006/3	HITACHI-SR11000K1 （2台）	21.5 TFlops	10 TB	
Ⅸ	2012/6	HITACHI-SR16000M1 （2台）	847 TFlops	108 TB	

数値予報モデルの計算量

- 全球モデル
  - 格子数
    - 1億3000万
  - 積分時間間隔
    - 6分40秒（400秒）
  - 計算時間
    - 84時間予報／264時間予報
    - 約20分／約55分
  - 必要な計算機資源
    - 40ノード（1280cores）

- 局地モデル
  - 格子数
    - 1億2000万
  - 積分時間間隔
    - 50/3秒（16.67秒）
  - 計算時間
    - 9時間予報
    - 約20分
  - 必要な計算機資源
    - 80ノード（2560cores）

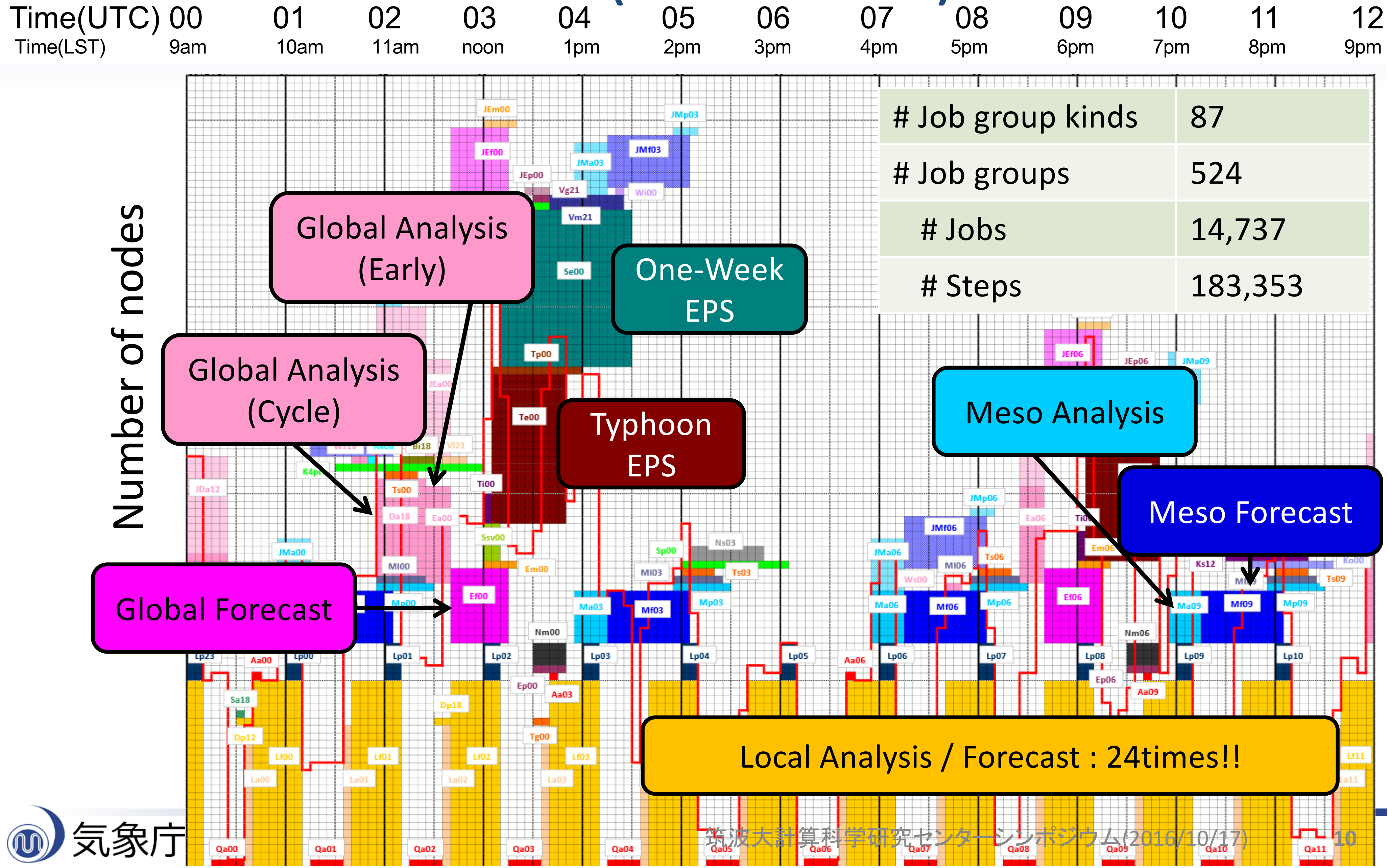
本日はこちらのモデルの話

第9世代数値解析予報システム

- HITACHI SR16000 model M1
  - 2つの独立なサブシステムで構成
    - 主系:現業運用
    - 副系:主系のバックアップ
  - 仕様

ノードあたり	CPU	IBM POWER7 3.83GHz x 4
	cores / CPU	8
	ピーク性能	980.48GFlops
	メモリ容量	128 GB
	L1/L2キャッシュ	32KB/256KB on chip
システム全体	ノード数	864(432x2)
	ピーク性能	847(423.5x2)TFlops
	メモリ容量	108(54x2)TB

現業ジョブのスケジュール
日中 (00-12UTC)



# Job group kinds	87
# Job groups	524
# Jobs	14,737
# Steps	183,353

気象のアプリケーションの特徴

- Fortranで記述
- 鉛直格子間隔<<水平格子間隔
  - 鉛直方向に陰解法
- 伝統的なベクトル計算機では、最内からi (東西), j(南北), k(鉛直)の並び。
  - 鉛直方向に依存性のあるところでも、水平方向でベクトル化
- いわゆるホットスポット(全実行時間の数割を占めるようなループ)がない、または少ない
  - 上位100ルーチン程度に最適化を施す必要。
- 伝統的にメモリバンド幅律速

加えて、現業モデルの特徴

- 多くの業務が同時に実行されている
  - ジョブ毎に資源の割り当てが決まっている
- 決まった時間で毎回計算を終える必要がある。
  - 極端な例：いくら高精度な予報でも、12時間後の予報を24時間後にもらっても意味がない。
  - 最適化を施して計算時間を短縮することで、より高度な手法を導入することが可能
    - 計算機科学の重要性
- 局地モデルでは約20分
  - ノード数はある程度つぎ込める
  - ストロングスケーリング（問題サイズ固定）が重要

次世代非静力学モデルasuca

- 2007年開発開始
- 十分な計算安定性
- 厳密に質量保存を満たす
- スカラー機での計算効率
- 人為的なパラメータの排除
- コードの整理 => 開発および維持・管理のしやすさ
- 科学的な見地に基づいた基礎からのモデル開発

計算機を良く知る

- I／O時間 ＞＞通信時間
- 通信時間＞＞メモリコピー
- メモリコピー＞＞演算時間

- なるべく、速い操作の裏で、遅い操作を行う（オーバーラップ）

- 具体的な例を以下で

メモリコピー＞＞演算時間

asucaのソフトウェアデザイン

- Fortran90で記述、MPI+OpenMPで並列化
- スカラー機で高い計算効率を得る
  - **kij - ordering** `real(8) :: u(nz, nx, ny)`
    - 3次元配列の要素の並びを z (k), x (i), y (j) にする.
      - 最内側に鉛直方向を持ってくる=>キャッシュヒットをあげる。
        - » 局地モデル(nz=58+2)では、鉛直1次元の倍精度の配列であれば、L1キャッシュに66配列、L2キャッシュに528配列保持できる
      - 外側のループ長が大きい=>並列化に有利
        - » 局地モデルでは、1MPIランクあたりnx=52, ny=62くらい
  - **MPI通信回数の削減**
    - 力学過程・物理過程の計算順序を整理し、まとめて通信を行うようにする

	ASUCA	JMA-NHM
Number of calling MPI comm.	72600	138652

Assumption of 1hour forecast of LFM(dx=dy=2km)
JMA-NHM:dt=8, asuca:dt=16

ijk 方式の場合

```
subroutine dynamics
call coriolis(u,v,tend_x,tend_y)
call curvature(u,v,tend_x,tend_y)
…
```

```
subroutine coriolis(u,v,tend_x,tend_y)
real(8),intent(in)    ::u(nx,ny,nz)
real(8),intent(in)    ::v(nx,ny,nz)
real(8),intent(inout) ::tend_x(nx,ny,nz)
real(8),intent(inout) ::tend_y(nx,ny,nz)


!$OMP PARALLEL DO
  do k = 1, nz
  do j = 1, ny
  do i = 1, nx
    tend_x(i,j,k) = tend_x(i,j,k) &
      & + 2 * omg * v(i,j,k) * sin_flat
    tend_y(i,j,k) = tend_y(i,j,k) &
      & + 2 * omg * u(i,j,k) * sin_flat
  end do
  end do
  end do
!$OMP END PARALLEL DO
```

```
subroutine curvature(u,v,tend_x,tend_y)
real(8),intent(in)    ::u(nx,ny,nz)
real(8),intent(in)    ::v(nx,ny,nz)
real(8),intent(inout) ::tend_x(nx,ny,nz)
real(8),intent(inout) ::tend_y(nx,ny,nz)


!$OMP PARALLEL DO
  do k = 1, nz
  do j = 1, ny
  do i = 1, nx
    tend_x(i,j,k) = tend_x(i,j,k) &
      & + u(i,j,k) * m2/m1…
    tend_y(i,j,k) = tend_y(i,j,k) &
      & + v(i,j,k) * m2/m1…
  end do
  end do
  end do
!$OMP END PARALLEL DO
```

kij方式の場合

```
subroutine dynamics
real(8),intent(in)    ::u(nz,nx,ny)
real(8),intent(in)    ::v(nz,nx,ny)
real(8),intent(inout) ::tend_x(nz,nx,ny)
real(8),intent(inout) ::tend_y(nz,nx,ny)

!$OMP PARALLEL DO
  do j = 1, ny
  do i = 1, nx
    call colioris(u(1,i,j), v(1,i,j), &
      &            tend_x(1,i,j),     &
      &            tend_y(1,i,j)      )
    call curvature(u(1,i,j), v(1,i,j), &
      &             tend_x(1,i,j),     &
      &             tend_y(1,i,j)      )
  end do
  end do
!$OMP END PARALLEL DO
```

```
subroutine coriolis(u,v,tend_x,tend_y)
real(8),intent(in)    ::u(nz)
real(8),intent(in)    ::v(nz)
real(8),intent(inout) ::tend_x(nz)
real(8),intent(inout) ::tend_y(nz)
do k = 1, nz
  tend_x(k) = tend_x(k) &
      & + 2 * omg * v(k) * sin_flat
  tend_y(k) = tend_y(k) &
      & + 2 * omg * u(k) * sin_flat
end do
```

```
subroutine curvature(u,v,tend_x,tend_y)
real(8),intent(in)    ::u(nz)
real(8),intent(in)    ::v(nz)
real(8),intent(inout) ::tend_x(nz)
real(8),intent(inout) ::tend_y(nz)
do k = 1, nz
  tend_x(k) = tend_x(k) &
    & + u(k) * m2/m1…
  tend_y(k) = tend_y(k) &
      & + v(k) * m2/m1…
end do
```

- メリット
  - SMP並列の起動回数削減
  - キャッシュヒットの向上
- デメリット
  - ベクトル長が短くなる

通信時間＞＞メモリコピー＞＞演算時間

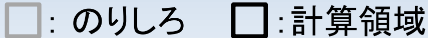
プロセスレベルの並列化

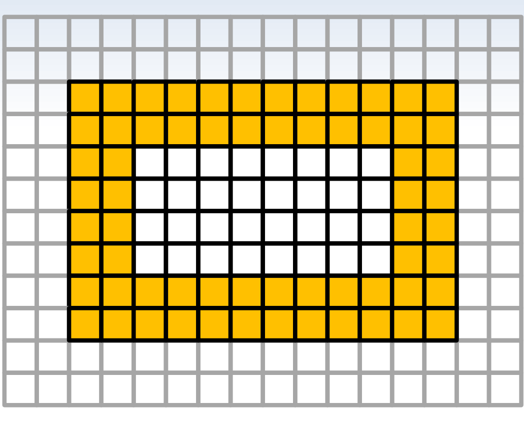
- 2次元領域分割 ＋ I／O専用rank
  - のりしろ、を設ける
- 任意のMPIプロセス数を用いた場合に、結果がbit レベルで一致するようにしている
  - 現業運用では、並列化効率が多少落ちても、割り当て資源内でノードをつぎ込んで短い時間で終わらせる。
  - 開発段階で、現業運用と同じノード数を用いるのは、無駄が多い
  - 異常終了が起きた場合など、ほんのわずかな差が結果を左右することもある
  - 総和演算があると簡単ではない

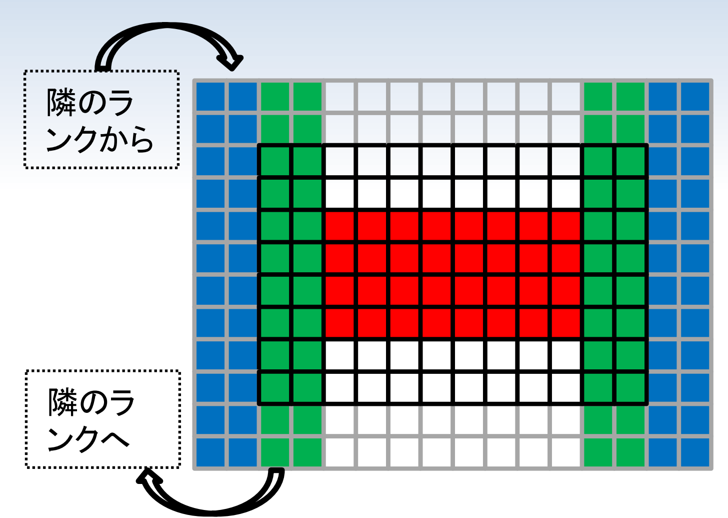
のりしろの通信

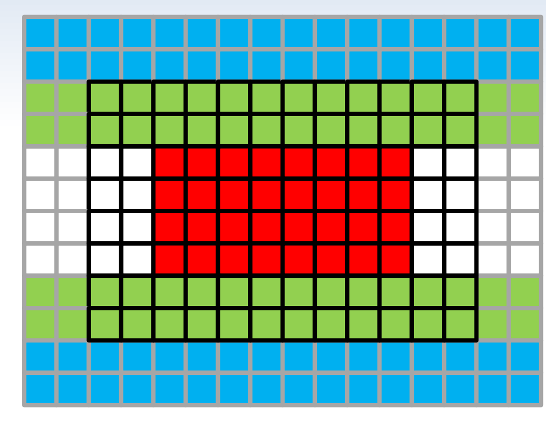
- 複数の要素を通信用のバッファにコピーして、まとめて通信する。
  - 起動回数を減らしてオーバーヘッドを削減
- 重い演算とオーバーラップさせる

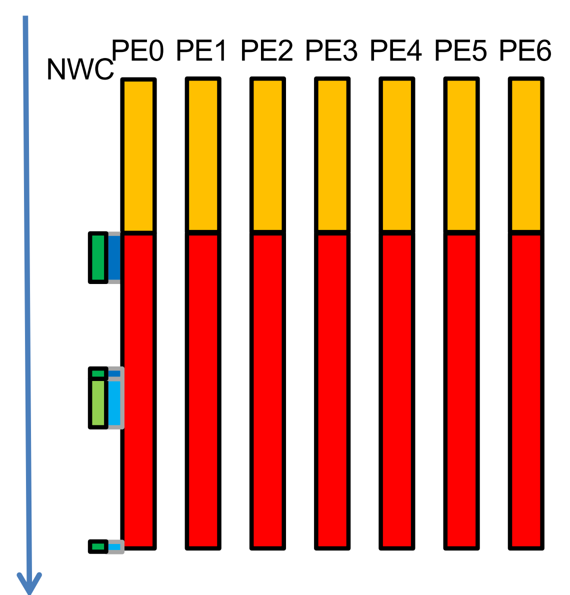
演算とのりしろ交換のオーバーラップ(1)

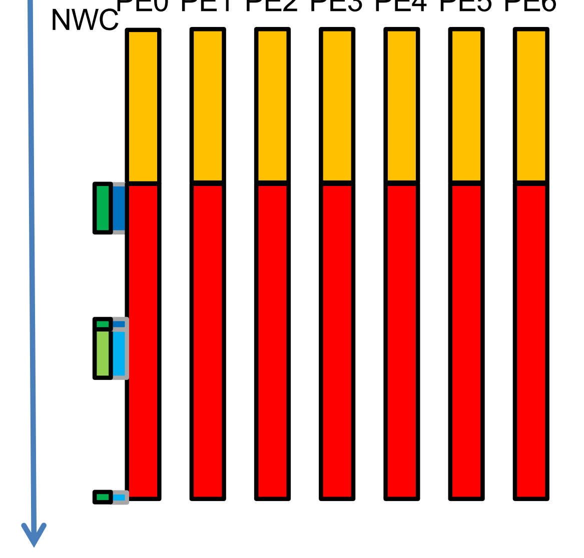












1. □を計算。
2. スレッド0で□と□の通信を起動しつつ、全スレッドを用いて□を計算。
3. スレッド0で□と□について同期を取った上で、□と□の通信を起動。全スレッドでの□の計算は継続。
4. スレッド0で□と□について同期。
5. 演算と通信のバランスにより、右のようなパターンも利用

NWC PE0 PE1 PE2 PE3 PE4 PE5 PE6

NWC PE0 PE1 PE2 PE3 PE4 PE5 PE6

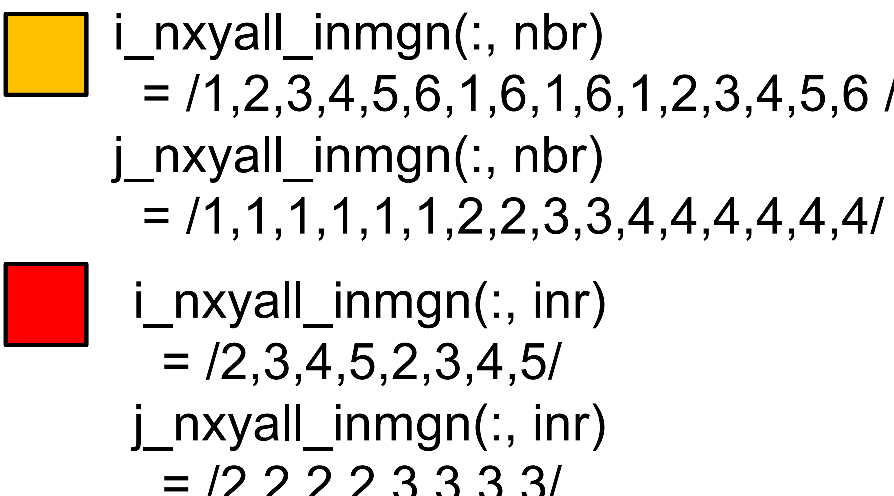
演算とのりしろ交換のオーバーラップ(2)

- なるべく重い演算と通信をオーバーラップさせたい
- 基本は鉛直1次元の設計
  - あるカラム(同一i,j)に対する計算をなるべく多く行いたい
  - （必然的に）あるi,jが与えられたときのコード／サブルーチンコールが増える
  - オーバーラップのために同じコードを2回書きたくない！

演算とのりしろ交換のオーバーラップ(3)

- 簡単のためのりしろが1列の水平2次元の例
- iとjのループを融合した上で、番号を振りなおす（間接アドレス参照）
- 実際には最内側にｋループがあり、アドレスの連続性はｋループで担保

0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5



演算とのりしろ交換のオーバーラップ(4)

- 同じコードを2回書きたくない！

```
do idx = nbr, inr, nti
  if (idx == nbr) then
    ijs = 1
    ije = nxy_ngbrmgn
  else if (idx == inr) then
    ! ijs = -29
    ijs = 0
    ijs = - nxy_innrmgn/6 + 10
    ije = nxy_innrmgn
  endif
!$OMP PARALLEL DO &
!$OMP& PRIVATE(i) &
!$OMP& PRIVATE(j)
  do ij = ijs, ije
    if (ij == 0) then
      call mpi_halo_run( mom_z_v(nz_mn,nx_mn,ny_mn,it_calc) )
    elseif (ij >= 1) then
      j = j_nxy_inmgn(ij, idx)
      i = i_nxy_inmgn(ij, idx)
      call rldamp_run_dens( tend_dens_ptb_v_rk_s(1,i,j), dens_ptb, i, j)
      if ( isw_monitflux .and. rk_counter == 3 .and. rk_counter_s == 3) then
        call monitflux_run_src( tend_dens_ptb_v_rk_s(1,i,j), dt_rk_s, i, j )
      endif
      if ( .not. isw_cyclic ) then
        call lbc_run_rk_short_dens_rldamp(i,j)
      endif
```

境界付近、中央の順に制御

領域の端付近をまわすためのループ

領域の中央部分をまわす＋通信のためのループ

0番ランクの演算が少なくなるよう、開始インデックスをずらす

```
      call dyn_hevi_momz( &
        & mom_z_v(nz_mn,i     ,j     ,it_calc), &
        & mom_x_v(nz_mn,nx_mn,ny_mn,it_calc), &
        & mom_y_v(nz_mn,nx_mn,ny_mn,it_calc), &
        & mom_z_v(nz_mn,nx_mn,ny_mn,it_root), &
        & rmpt_ptb_v(nz_mn,nx_mn,ny_mn,it_tend), &
        & dens_ptb_v(nz_mn,nx_mn,ny_mn,it_tend), &
        & mom_xi_v, mom_yi_v, mom_zi_v, &
        & tend_mom_z_v_rk, tend_rmpt_ptb_v_rk, &
        & tend_dens_ptb_v_rk, &
        & tend_dens_ptb_v_rk_s, i, j)
    endif
  end do
!$OMP END PARALLEL DO
end do
```

・計算本体を徹底的に鉛直1次元化
・先に計算して送るところと、通信しながら計算するところを分けて、インデックスのリストを作成

I／O時間＞＞メモリコピー

現業局地モデルのI／O

- 単に読み書きするだけでも結構時間がかかる
  - 入力データ量
    - 初期値等：46GB
    - 境界値：120GB
  - 出力データ量
    - 全部で80GB（ただしinteger(2)のレベル値）

・計算開始時点
　メトリック:39GB, 初期値12GB,
　リファレンス等5GB, 境界値24GB
・その後計算終了までで96GB

- 単純な実装でやってみる
  - ランク0でデータを読んで計算ランクに配る
  - ランク0に集約して出力

	CPU時間
計算時間	898秒
I／O時間（通信や圧縮を含む。計算ランクでは待ち時間）	764秒

⇒ 全体で20分(1200秒)で終わる必要

入出力の扱い(1)

- 並列計算において
  - 入出力のみを行うプロセス(I/Oプロセス)
  - 計算のみを行うプロセス(計算プロセス)

  を設けて、計算プロセスになるべく待ちが発生しないようにする。

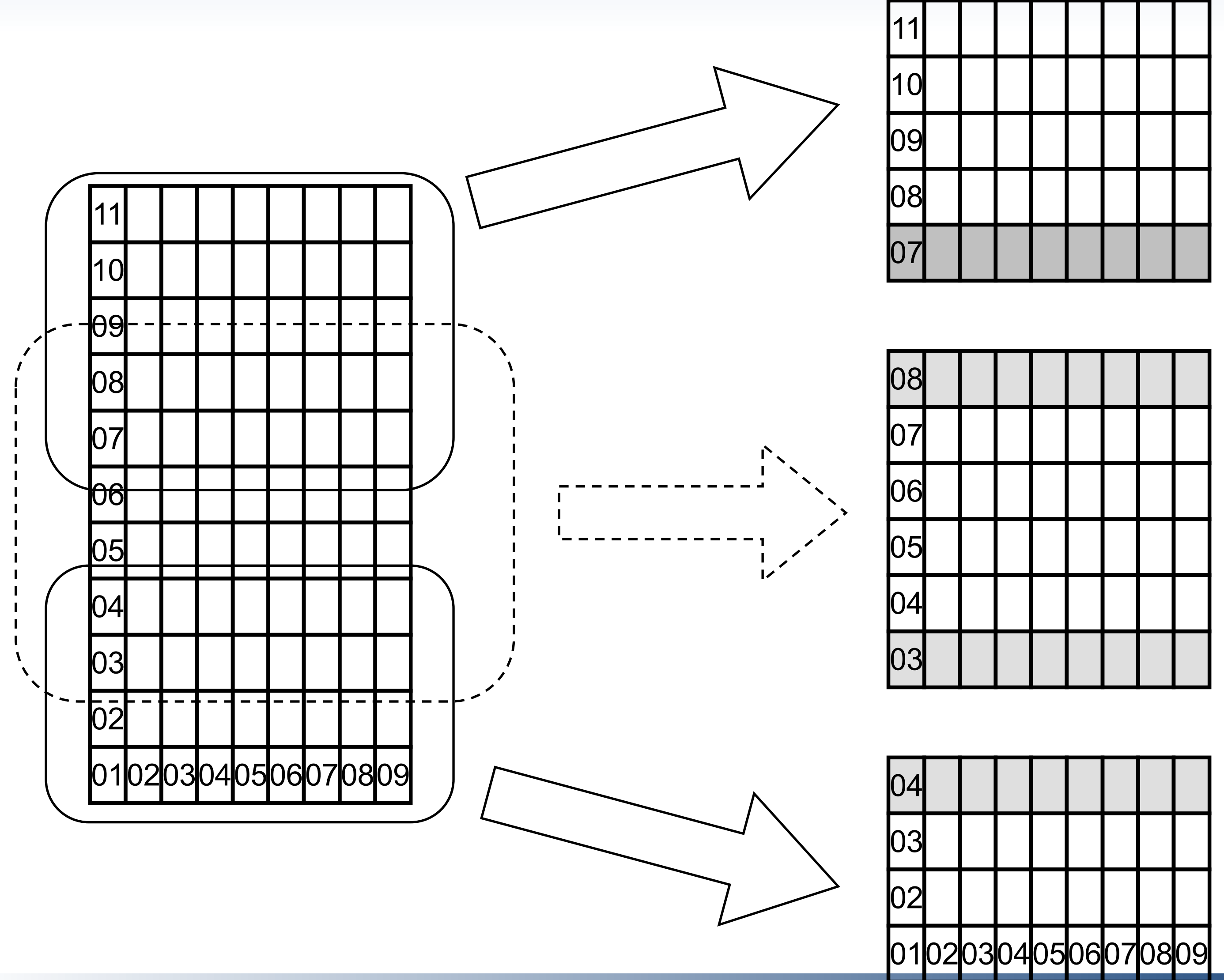
入出力の扱い(2)

- 入力･･･先読み
  - 計算プロセスが計算している間に、先にディスクから読んでメモリに保持しておく(側面境界値など)。
  - 複数のプロセスで手分けして読み込み。
    - 各プロセスが利用できるメモリ量の制限(1rank あたり10GB程度)等があるため。
    - ファイルシステムの性質上、複数のファイルを手分けして読むと高スループット
    - データを計算ランクに配る時間も結構かかるので、その間にデータを読める
  - (計算開始時刻を除く境界値では)計算プロセスでデータが必要になった時点で、既にI/Oプロセスではデータを送り終えている。計算プロセスでも受信が終わっており、同期をとるだけ。

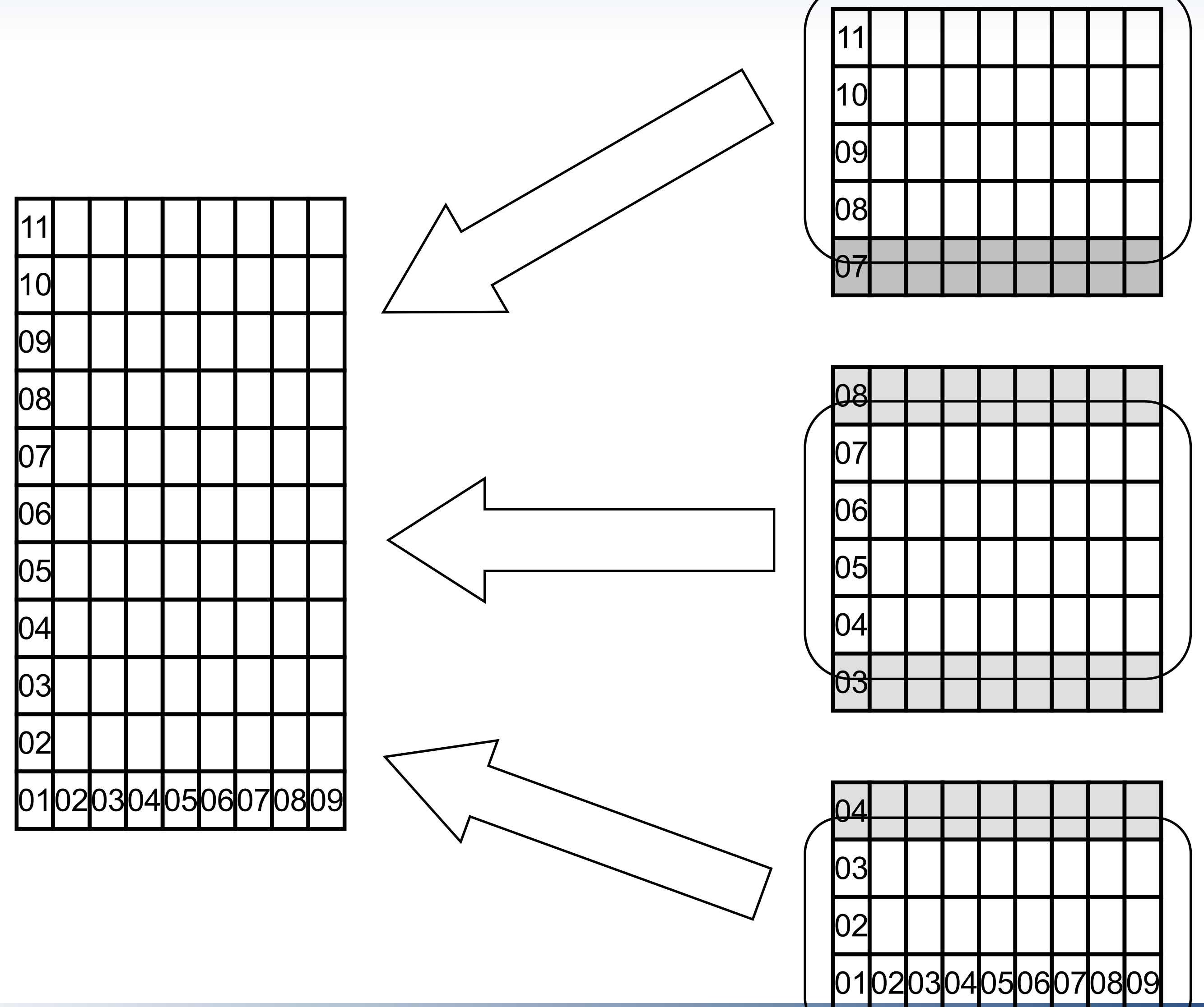
入出力の扱い(3)

- 出力･･･たらいまわし
  - プロダクトとして、領域全体をまとめた形で後続のジョブに引き継ぐ必要があるため、集約して集める必要がある。
  - 計算プロセスは自分が持つバッファに出力データを退避させて通信を起動し後続計算をすぐに再開。
  - I/Oプロセスは自分のバッファに受信して出力する。
  - 出力時刻ごとに出力するI/Oプロセスを切り替える。
    - 出力時間間隔が短いと出力が終わる前に次の出力のタイミングとなってしまい待ちが生じるため。

入力データを計算ランクに配る



出力データをI／Oランクに集める



read性能を簡単な実験により確認

```
program main
  implicit none
  integer(4) :: i
  integer(4) :: ib
  integer(4) :: n = 80
  integer(4) :: nn
  integer(4) :: nxy=1024*1024*128
  real(8), allocatable :: wbuf(:,:)
  include 'mpif.h'
  integer(4) :: ierr, nrank, myrank
  integer(4) :: onefile
  namelist /parameter/onefile
  open(1, file = 'parameter.txt')
  read(1, parameter)
  close(1)
  call mpi_init(ierr)
  call mpi_comm_size(mpi_comm_world, nrank, ierr)
  call mpi_comm_rank(mpi_comm_world, myrank, ierr)
  nn = n / nrank
  allocate(wbuf(nxy,nn)) !1GB * nrank
!$OMP  PARALLEL DO &
!$OMP& SCHEDULE(STATIC,1) &
!$OMP& PRIVATE(ib)
  do i = 1, n
    ib = mod(i-1,nn)+1
    if ( (i-1)/nn == myrank ) then
      write(6,*) "i,ib = ", i,ib, "at rank = ", myrank
      call readbuf(i,wbuf(1,ib),nxy,onefile)
    endif
  end do
!$OMP END PARALLEL DO
  call mpi_barrier(mpi_comm
  call mpi_finalize(ierr)
  stop
end program main
```

```
subroutine readbuf(i,buf,nx,onefile)
  implicit none
  integer(4), intent(in) :: i
  integer(4), intent(in) :: nx
  integer(4), intent(in) :: onefile
  integer(4) :: ip
  real(8), intent(out) :: buf(nx)
  if (onefile == 0) then
    read(10+i) buf
  else
    rewind(10)
    do ip = 1, i-1
      read(10)
    end do
    read(10) buf
  endif
  return
end subroutine readbuf
```

- SR16K 1ノードを占有して実行（全体で80 GB をread ）。

	1MPIx 1SMP	1MPIx 7SMP	8MPIx 1SMP	8MPIx 7SMP
1ファイル	3m26s	-	2m33s	-
80ファイル	2m19s	47s	33s	19s

スループットを出すには・・・
* ファイルは、ある程度の数に分けなければならない。
* 複数のノード（CPU）から読まなければならない。

データを読んで配る部分の改善

- 複数ファイルを、複数CPUから読む必要がある。
  – 読み込み用バッファは、読み込みバイト数×ファイル数
- 1MPIランクで使用できるメモリサイズは上限10GB程度。なるべく送信バッファに割り当てたい。
- 読み込みの単位を面ごととし、読み込んだらすぐに送信バッファにつめる
  – 面ごとなら、1581x1301x8x7 = 115MB程度の読み込み用バッファで済む

```
real:: rbuf(nx0,ny0)

!$OMP PARALLEL DO &
!$OMP& PRIVATE(rbuf)
  do i = 1, nfile
    read(mt(i)) rbuf
    call rbuf2sbuf(rbuf,…)
  end do
!$OMP END PARALLEL DO
```

更なる効率化：ファイルサイズの削減

```
do k = nz_mn, nz_mx
   maxv(k) = maxval(:,:,k)
   minv(k)  = minval(:,:,k)
enddo

do k = nz_mn, nz_mx
   write(mt) maxv(k), minv(k)
end do

do k = nz_mn, nz_mx
   if (maxv(k) /= minv(k)) then
     write(mt) data(:,:,k)
   endif
end do
```

- ある要素を中間ファイルに出力する際、まずは領域内の最大値と最小値の組をファイルに出力する。
- 最大値＝最小値の場合は、領域内全部が同じ値と考えられるので、2次元データの出力はしない。そうでない場合のみデータを出力する。
- 予報変数ではあるが、親モデルに存在しないような要素（e.g あられ）について、全部0になるので実質鉛直層数×２個のデータを読み、それを各計算ランクへ配るだけになる。
- 雨のように、上層では（気温が低いために）全領域で0になる場合も自動的にこの形の“圧縮”がかかる。
- I／Oサイズ、通信サイズが少なくなって効率化。
- 主・副間同期でネットワーク越しにコピーする量が減る。

現業局地モデルと同じスペックでのデータ量

data	変更前(GB)	変更後(GB)
BND_FILES.dir	126.52	110.06
CST_FILES.dir	1.67	1.48
GRID_FILES.dir	52.09	38.99
INIT_FILES.dir	23.31	19.68

３次元データを計算ランクへ配置 具体的な実装

	I／Oランク	計算ランク
ステップ1	・読み込むデータの各層（＝各レコード）のmax, min を読み込み、計算ランクへ送信。	・各層のmax, min を受け取る。ここから受信すべきデータの層の数が計算できる。
ステップ2	・実際にデータを読み込み、送信用バッファにためる。 ・次の要素を呼んだときに、バッファの容量(*)を超えそうになったら、I／Oランクを切り替える。	・I／Oランクからのデータ受信を起動。 ・バッファの容量と大きさに応じて、随時送信元のI／Oランクを切り替え。
ステップ3	・後処理（ファイルを閉じる、I／Oランク切り替え、メモリの開放等）。	・mpi_wait で受信確認の後、実際の配列（風、気温、・・・）に代入。 ・後処理。

(*)バッファの容量：実行時オプションにより、あらかじめ与えられる

側面境界値

- 領域モデルは、限定された領域内の時間積分を行う。
- 側面境界値（境界条件）は、より粗いモデルの計算結果（1時間間隔）を時間内挿して使う。
- 局地モデル（9時間積分）のうち、時刻2－9のデータは、積分開始時点で不要。
  - 計算とオーバーラップさせて、データの読み込み、通信のオーバーヘッドを隠蔽。

T=2h と T=3h の時間内挿で作成

T=2h から T=3h を計算している間に、データの読み込みと通信を行っておく。

T=2h　T=3h　T=4h

境界値を配るコードの実際

- ステップ1（iloop=1)
  - 総層数カウント
  - 各層のmax, min 送信
- ステップ2(iloop=2)
  - 3次元データ通信起動
  - 必要に応じI/Oランク切替
- ステップ3(iloop=3)
  - mpi_wait、配列への代入
  - 時刻の張り替え
- 積分開始時点
  - （1,2,3）×2と、1,2を起動
- T＝1h 以降
  - 3,1,2 を起動

```
if (is_inibnd_file == isw_file_dist) then
  iloops = 1
  iloope = 1
else
  if (l_wait_only) then
    iloops = 3
    iloope = 3
  elseif (l_irecv_only) then
    iloops = 1
    iloope = 2
  else
    iloops = 1
    iloope = 3
  endif
endif

do iloop = iloops, iloope
  call mpi_comm_read_scatter_pre(mt, fname_bnd, is_inibnd_file, iloop)
  call mpi_comm_read_scatter_run(mom_x_v_bnd(nz_mn,nx_mn,ny_mn,idx), &
    &   mt, nz_mn, nz_mx, nx_mn, nx_mx, ny_mn, ny_mx, is_inibnd_file, iloop)
  call mpi_comm_read_scatter_run(mom_y_v_bnd(nz_mn,nx_mn,ny_mn,idx), &
    &   mt, nz_mn, nz_mx, nx_mn, nx_mx, ny_mn, ny_mx, is_inibnd_file, iloop)
  call mpi_comm_read_scatter_run(mom_z_v_bnd(nz_mn,nx_mn,ny_mn,idx), &
    &   mt, nz_mn, nz_mx, nx_mn, nx_mx, ny_mn, ny_mx, is_inibnd_file, iloop)
  call mpi_comm_read_scatter_run(dens_ptb_v_bnd(nz_mn,nx_mn,ny_mn,idx), &
    &   mt, nz_mn, nz_mx, nx_mn, nx_mx, ny_mn, ny_mx, is_inibnd_file, iloop)
  call mpi_comm_read_scatter_run(rmpt_ptb_v_bnd(nz_mn,nx_mn,ny_mn,idx), &
    &   mt, nz_mn, nz_mx, nx_mn, nx_mx, ny_mn, ny_mx, is_inibnd_file, iloop)
  do id = id_qa_s, id_qa_e
    call mpi_comm_read_scatter_run(rqa_v_bnd(nz_mn,nx_mn,ny_mn,id,idx), &
    &   mt, nz_mn, nz_mx, nx_mn, nx_mx, ny_mn, ny_mx, is_inibnd_file, iloop)
  end do
end do

if ((.not. l_irecv_only) .or. is_inibnd_file == isw_file_dist) then
  call mpi_comm_read_scatter_post(mt, is_inibnd_file)
```

同じループを3回呼んで、内部で動作を制御することで、複雑な仕組みを理解しなくても変数の追加／削除が容易に。

初期値／境界値　（＝主要予報変数）を読みこんで計算ランクへ配るサブルーチン

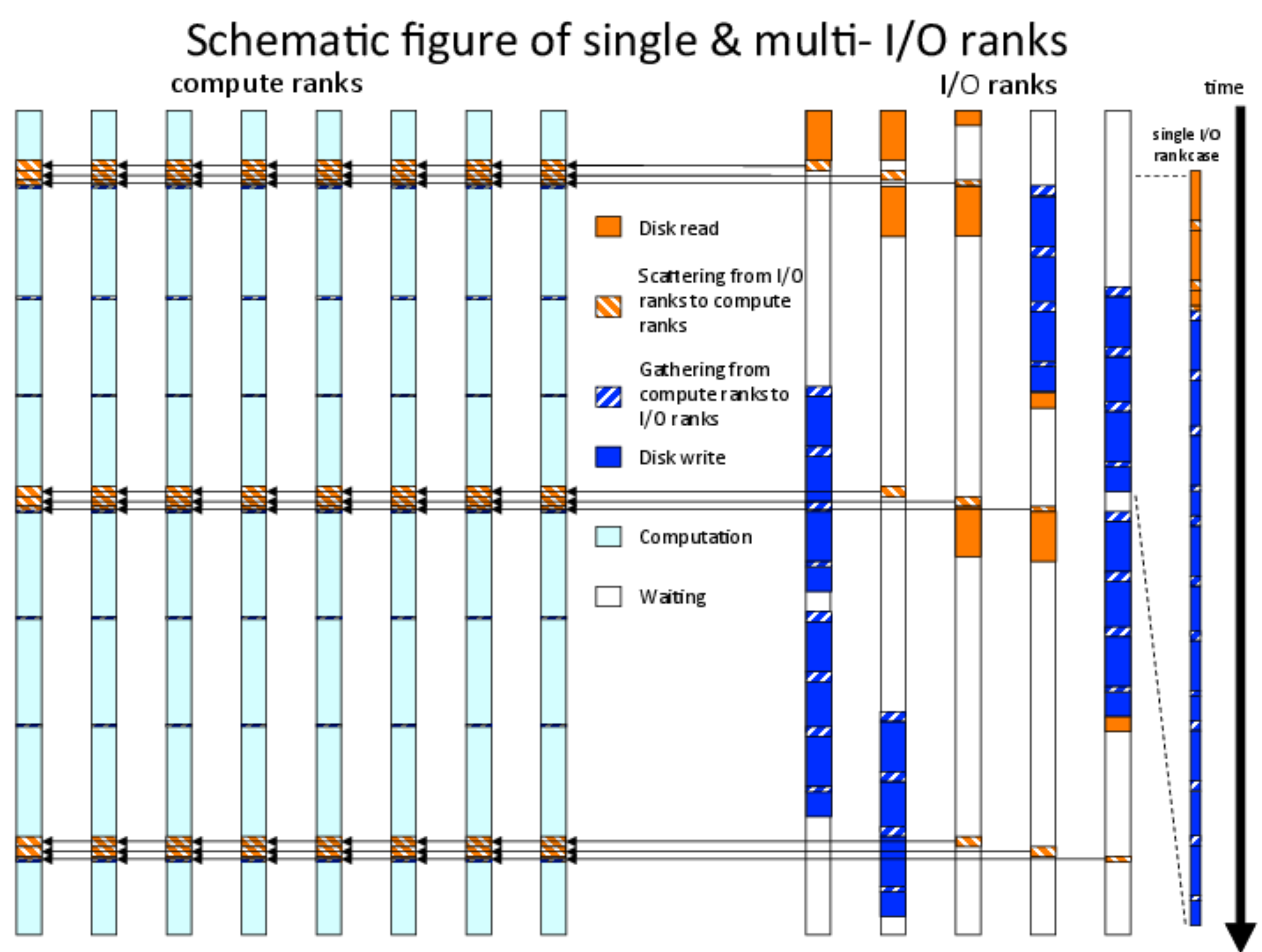
出力データの特徴

- 数値予報ルーチンではNuSDaSという独自形式で出力
  - 最終的には2 byte パックされる
    - 2byte パック : 65532 のレベル値(integer(2))に変換
    - 単精度に変換してから集めて出力（予報変数は倍精度）。
  - 2byteパック後でも、局地モデルでは1時刻あたり8GB程度の出力になる: 全体で80GB以上
    - 単に出力するだけでも結構かかる
  - 入出力ライブラリのAPI (e.g. nusdas_write) を通して出力する
    - 同じ対象時刻のデータ（同じファイルのデータ）を複数のMPIプロセスから効率的に出力することは難しい。
      - メタデータを入出力ライブラリのバッファで保持していたりするはず

出力たらいまわしの実際

- 計算ランク
  - I/Oランクの数だけバッファを用意
  - 出力のタイミングがきたら、その時刻を担当するI/Oランクと同期を取った上で、バッファにつめる。
    - 前時刻の出力が終わっていない場合の上書きを防ぐ
  - バッファ全体を、1度に通信する量（ルーチン局地モデルでは3GB)の単位に切って、タグを付け替えながらmpi_isend を起動。
    - バッファのサイズは、実行時のパラメータとして与える
  - 後続の計算に突入。実質メモリコピーの時間のみ。
- I/Oランク
  - 受けとったデータを並べかえ
  - “出力ライブラリ“を通じて出力。
  - 同じデータを2回通信する必要はない。

I/O 専用MPI rank

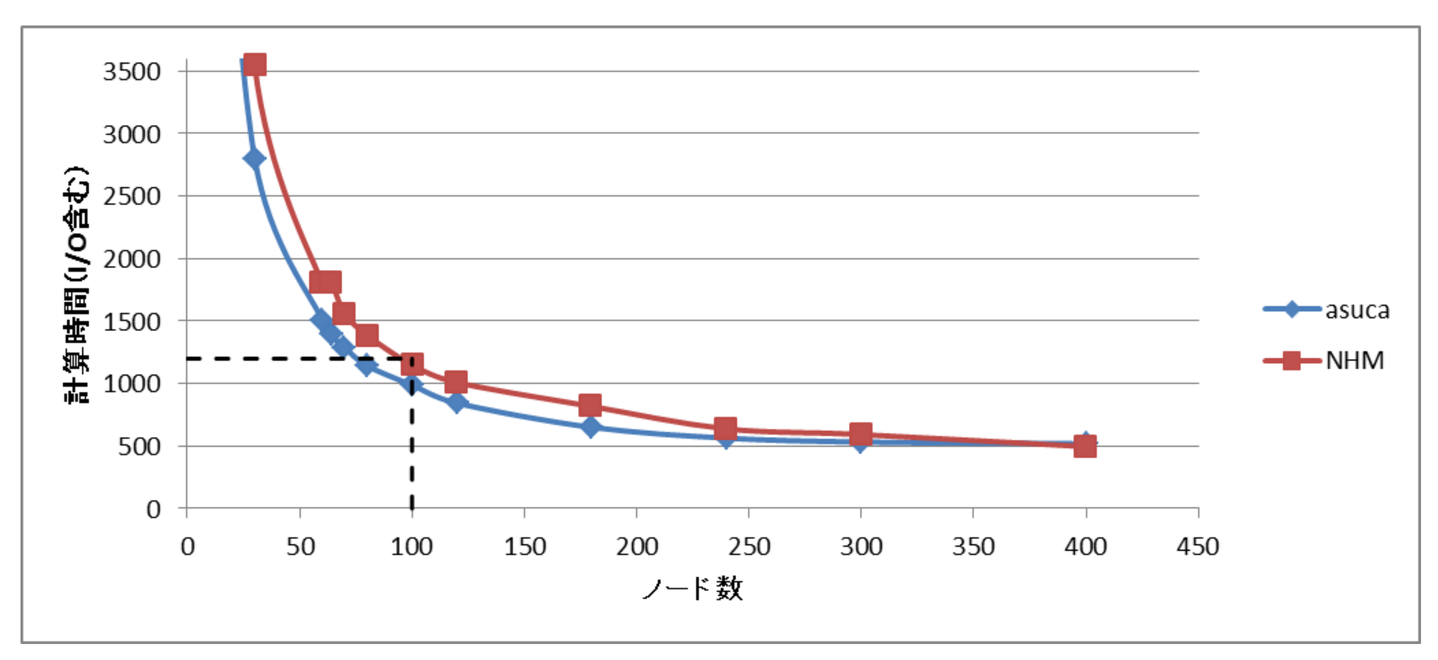


I／O専用ランクの効果

80ノード 30×21 + 0 vs 30 × 21 + 10

	I／Oランクなし	I／Oランク×10利用
計算時間	898秒	865秒
I／O時間（通信や圧縮含む）	764秒	87秒（内訳は以下）
		初期値のデータ読み込み：36秒
		予報終了時刻のデータ出力：51秒

局地モデル現業化時点での ノード数と実行時間の関係



まとめ

- 数値予報
  - 現在の天気予報、防災気象情報作成の基盤
  - 定時的なプロダクト作成のために、計算時間の制約が厳しく、ストロングスケーリングが重要
  - 計算機のつくりを良く知って、I／Oも含めてボトルネックをつぶす
  - 予報精度向上のため、計算機科学が重要な分野の一つになっている

ご清聴ありがとうございました。