Vol.2016-HPC-153 No.7
2016/3/1

ステンシル計算における効率的な Halo 通信・計算モデルの開発

深沢圭一郎[†1†4]　森江善之[†2†4]　曽我武史[†3†4]　高見利也[†2†4]　南里豪志[†2†4]

概要：ステンシル計算は計算を行う点の周辺データを利用して計算を進めるため，並列化に伴いいわゆる Halo（袖）通信が発生する．この Halo 通信時間をいかに減らすかがステンシル計算の並列計算性能向上につながる．我々は Halo スレッドを導入し，「計算」と「通信が必要な計算と通信」を分け，並列に計算を行うことで，並列計算性能を向上させた．この「通信が必要な計算と通信」は通信が終わらなければ，計算を行えない領域であり，非効率なままであった．本研究ではある Halo 通信が終われば，そこに関連する計算を行う Halo 通信・計算モデルを開発し，実際にステンシル計算である電磁流体（MHD）シミュレーションに導入した．これを利用した性能評価では Halo スレッドでの計算時間が減ることが確認された．

キーワード：袖通信，ステンシル計算，並列計算

Development of Effective Halo Communication and Calculation Model on Stencil Computation

KEIICHIRO FUKAZAWA[†1†4]　YOSHIYUKI MORIE[†2†4]
TAKESHI SOGA[†3†4]　TOSHIYA TAKAMI[†2†4]　TAKESHI NANRI[†2†4]

Abstract: The stencil computation requires the neighboring data to proceed the calculation. Thus the Halo communication is needed in parallel computation. It is important for the parallel scalability of stencil computation to decrease the Halo communication time. To achieve the high scalability, we have introduced the Halo thread and divided the simulation code into “calculation part” (regular thread) and “communication and calculation related the communication part” (Halo thread). However, “communication and calculation related the communication part” has not been effective execution yet. In this study we have developed the Halo communication functions which perform the Halo communication and related calculation effectively and introduced the MHD simulation code. As the results we have obtained good performances and confirmed the decrease of elapse time in the Halo thread.

Keywords: Halo communication, Stencil computation, Parallel computing

1. はじめに

エクサスケールの計算機システムは 300 万ノード以上の構成が想像されているが，我々が開発している惑星磁気圏を解く電磁流体（MHD）シミュレーションコードでは最大で京コンピュータの 3 万ノード程度で性能評価を行っており，weak scaling でスケーラビリティが約 10%劣化している．このような問題に対してハードウェア側から，通信専用コアの導入や，ノード間通信性能向上といった開発が進められており，ミドルウェアの部分では新しい通信ライブラリや MPI 自体の性能向上も議論されている．一方，アプリケーション側からは計算と通信をオーバーラップさせる手法が開発されており，通信にかかる時間をできる限り隠蔽しようとする努力がされている．例えば，*Sur* らは RDMA ベースのオーバーラップ手法を提案しており[1]，核融合分野では特定のネットワークハードウェア上だが，PGAS を用いた計算と通信のオーバーラップを *Preissl* らが提案している[2]．最近では *Idomura* らが MPI_isend/irecv における通信 progress を効率的に実行でき，通信終了後に通信スレッドも計算スレッドに参加する手法を提案している[3]．

MHD シミュレーションは流体シミュレーションの 1 種であり，stencil 計算である．stencil 計算では並列化に伴い Halo 領域と呼ばれる各プロセスにある袖領域をプロセス間で通信する必要が出てくる．そこで我々は Halo 通信とその通信結果が必要な計算を専用スレッド（Halo スレッド）にまかせる手法を開発し，MHD シミュレーションコードに導入した[4]．この手法では，通信と依存関係のある計算をすべて Halo スレッドに担当させることで，計算スレッドに通信に伴う同期を行わせる必要がない．この性能評価から，weak scaling における高いスケーラビリティを達成し，strong Scaling においても高いスケーラビリティを達成することができた．この手法では，Halo スレッドが担当する“通信と計算”自体は既存の手法と変わりが無く最適化の余地があった．

†1 京都大学学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University
†2 九州大学情報基盤研究開発センター
Research Institute for Information Technology, Kyushu University
†3 九州先端科学技術研究所
Institute of Systems, Information Technologies and Nanotechnologies
†4 CREST, JST
JST, CREST

そこで本研究では，Halo スレッドが担当する“通信と計算”を効率的に行うために Halo 通信用の関数群を開発し，MHD シミュレーションコードに導入した結果を評価する．

2. Simulation Model

電磁流体力学（MHD）シミュレーションは以下の MHD 方程式を解くことで，プラズマの振る舞いを調べている．

$$\begin{aligned}\frac{\partial \rho}{\partial t} &= -\nabla\cdot(\mathbf{v}\rho)\\ \frac{\partial \mathbf{v}}{\partial t} &= -(\mathbf{v}\cdot\nabla)\mathbf{v} - \frac{1}{\rho}\nabla p + \frac{1}{\rho}\mathbf{J}\times\mathbf{B}\\ \frac{\partial p}{\partial t} &= -(\mathbf{v}\cdot\nabla)p - \gamma p\nabla\cdot\mathbf{v}\\ \frac{\partial \mathbf{B}}{\partial t} &= \nabla\times(\mathbf{v}\times\mathbf{B})\end{aligned} \quad (1)$$

上から，連続の式，運動方程式，圧力変化の式（エネルギーの式），最後が磁場の誘導方程式となる[5]．簡単に言えば，電磁場を考慮した流体力学方程式と呼べる．詳しい導出方法は参考文献を参照されたい[6]．MHD 方程式は Vlasov 方程式から求められるが，いわゆる流体の方程式に電磁場を考慮した方程式になっている．惑星磁気圏といった巨大な構造を調べる際に利用されており，問題サイズとしてはエクサスケールにおいても weak scaling が続くと想定されている．

MHD 方程式を解く数値計算法としては，*Ogino* らによって開発された Modified Leap Frog（MLF）法[7，8]という差分法を使用する．これは最初の 1 回を two step Lax-Wendroff 法で解き，続く （l － 1）回を Leap Frog 法で解き，その一連の手続きを繰り返す．この中で，1 タイムステップ進めるために 2 段階に分けた計算を採用しているため，後述の Halo スレッドを導入した場合の MHD 計算フローチャートにおいて，計算が 2 段階になっている（図 1）．

3. Halo 通信・計算モデル

3.1 Halo Thread

我々のMHDシミュレーションは直交格子を利用しており，いわゆるstencil計算である．計算手法は前述の通り，差分法で解いており，並列化には領域分割を利用し，通信は基本的にはHalo通信のみである[8]．今までの研究では，通信専用スレッドを立て，非同期通信を行い，袖領域の通信を行うことが多い[1, 2]．この場合，計算スレッドが減少し，全体の計算性能が下がるため，通信時間が全実行時間の半分を占めるような場合を除き，一般には全体の計算コストは上がってしまう．そのため，*Idomura*らは通信終了後にスレッドのダイナミックスケジューリングにより計算に通信スレッドを参加させる工夫がされている[3]．我々は，それらの手法とは異なり，通信終了後に通信結果を必要とする計算までを通信を行ったスレッド自身（Haloスレッド）に行わせる．これにより，計算スレッドで行われている計算と通信スレッドで行われる通信と計算を完全にオーバーラップさせることができる．さらにこの手法であれば，計算終了後の1回だけ同期を取れば良い．この手法は更新される配列，更新に利用する配列が異なっていれば，安全に実行できる．

Haloスレッドを導入した場合のフローチャートを図1に示す．MLF法では前述のように2段計算（1st+2nd MHD calc.）で1タイムステップを進める．差分計算を行うために必要なHalo領域のデータを，これまでは計算開始前にHalo通信を行い送受信していた．Haloスレッドを導入した場合は，計算スレッドは通信をケアする必要が無いため，独立して計算ができる．このようにHaloスレッドと計算スレッドを分けると，計算担当スレッドが減る分，計算性能が下がり，通信時間が隠蔽できても全体的には計算性能が下がる場合もある．またMICのようなメニーコアの場合は一つのコアの性能が低いために，Haloスレッドを1コアだけではなく，2コア以上に設定することも考えられる．

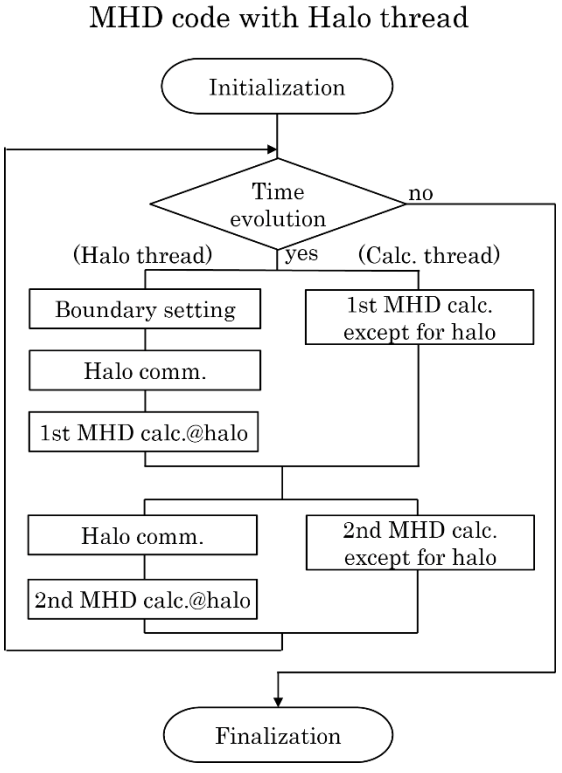

図 1　Halo Thread を導入した MHD 計算フローチャート
Figure 1　Flowchart of MHD code with Halo thread.

3.2 Halo 関数

Halo スレッドで実行される Halo 通信は Stencil 計算の場合，決まった相手と決まった量を通信する場合が多い（AMR などは除く）．また Halo スレッドに限らないが，通信が終わったところから計算を進めていけば，通信による性能劣化が最小限に抑えられることが考えられる．しかし

ながら，複数次元の領域分割より並列化を行った Stencil 計算では，通信の回数を削減するために，通信順序を固定して Halo 通信を行うことが多い[9]. そこで本研究では，Halo 通信に必要な各種パラメータを前もって登録し，Halo 領域にある面，線と点データを効率的に送受信できる下記の関数を作成した.

- halo_init : Halo 通信の初期設定を行う．本関数は，与えられたプロセス分割の次元数及び，次元毎のプロセス数に応じて，自動的に自プロセスの論理座標を割り当てる．また, halo 通信の対象となる行列(配列)と，その次元数と次元毎の要素数及び，論理分割次元に配列のどの次元が相当するかを指示することにより，行列を halo 通信の対象として登録する．halo_init 関数は通信に参加する全プロセスが通信の対象であるとみなして論理座標割り当てを行う.
- halo_isend : 袖領域の送信を行う．本関数は，指示したハンドルに登録されている自プロセスの配列内の指示した範囲から（パッキングも行う），指示した方向の隣接プロセスへの通信を開始する．本関数は通信完了を待つことなく完了する．通信の完了は，halo_wait または，halo_test 関数により知ることができる.
- halo_irecv : 袖領域の受信を行う．本関数は，指示した方向の隣接プロセスから，指示したハンドルに登録されている自プロセスの配列に内の指示した範囲へ（アンパッキングも行う）の通信を開始する．本関数は通信完了を待つことなく完了する. 通信の完了は, halo_wait または，halo_test 関数により知ることができる．通信完了前に受信領域を書き換えた場合には，データの内容は不定になる.
- halo_wait : 本関数は，指示したリクエスタに対応する通信が完了するまで待つ．halo_isend 関数に対応した halo_wait 関数の完了後は，通信領域を書き換えても書き換え前のデータが受信側に送られていることが保障される．また，halo_irecv 関数に対応した halo_wait 関数の完了後に通信領域から読み出されるデータは，送信側から送られてきたデータであることを保障する.
- halo_finalize : 本関数は，halo_init で登録された内容を解放して，halo 通信を終える.

引数など詳細については，付録 A を参照されたい．この関数群は C 言語と現在は MPI ライブラリで実装されているが，ACP ライブラリ[10]で実装することも可能である．大規模数値計算に利用者の多い Fortran から利用できるように wrapper が用意されており, 本研究では Fortran から Halo 関数を利用している．この Halo 関数群を利用すると，ランク配置などは halo_init で行うため, 典型的な Stencil 計算であれば，MPI を陽に呼ばずに並列計算を実装することができる.

3.3 Implementation

Halo スレッドと Halo 関数の MHD シミュレーションコードへの実装例は図 2 のようになる（Fortran+OpenMP 利用）．Halo スレッド（スレッド番号は 0）はまず Halo 関数により通信を行い，その後 Halo 領域を計算している．その一方で，他のスレッドは計算だけを行っている．実装自体は現状でも非常にシンプルである．また計算と通信をオーバーラップするように実装した例が図 3 になる．ここでは通信を先に呼び，通信が終わったところから計算を行っている.

```
  call halo_init(f) ! Halo Initilization
!
!----Time evoluation---!
  do time = 1, 1000
!
!----Thread setting----!
!$OMP PARALLEL PRIVATE(myid,mylid,ks,ke,ii)
    myid = omp_get_thread_num()
    nthreads = omp_get_num_threads() - 1
    mylid = myid - 1
    kmod = mod(nzz-2, nthreads)
    kdiv = floor(real((nzz-2)/nthreads))
!
    if (kmod > mylid) then
      ks = mylid * (kdiv + 1) + 1
      ke = ks + kdiv
    else if (kmod == mylid) then
      ks = mylid * (kdiv + 1) + 1
      ke = ks + kdiv - 1
    else
      ks = mylid * kdiv + kmod + 1
      ke = ks + kdiv - 1
    end if
!
!----Halo thread----!
    if(myid == 0) then
      call boundary(f)   ! boundary setting
      do l = 1, 26
        call halo_irecv(f) ! Halo recieve
        call halo_isend(f) ! Halo send
        call halo_wait ! for receive
        call halo_wait ! for send
!
        do k = zs(l), ze(l)
          call mhd_calc(f) ! MHD calc. at Halo
        end do
!
      end do
!----Calc thread----!
    else
      do k = ks+1, ke-1
        call mhd_calc(f) ! MHD calc.
      end do
    end if
!
!$OMP END PARALLEL
.
.
.
  end do
```

図 2　Halo 関数の MHD コードへの実装例
Figure 2　Implementation of Halo function to MHD code.

```
!----Halo thread----!
    if(myid == 0) then
      call boundary(f)   ! boundary setting
      do l = 1, 26
        call halo_irecv(f) ! Halo recieve
        call halo_isend(f) ! Halo send
      end do
!
      do l = 1, 26
        call halo_wait ! for receive
        do k = zs(l), ze(l)
          call mhd_calc(f) ! MHD calc. at Halo
        end do
      end do
!
      do l = 1, 26
         call halo_wait ! for send
      end do
!----Calc thread----!
    else
      do k = ks+1, ke-1
        call mhd_calc(f) ! MHD calc.
      end do
    end if
```

図 3　通信と計算をオーバーラップさせた場合の Halo 関数の実装例

Figure 3　Implementation of Halo function with overlapping the calculation and communication.

4. Performance Measurements

本研究では，以前の研究[4]で Halo スレッド導入効果があった結果に Halo 関数を導入し，その効果を調べる．利用する計算機システムは，九州大学情報基盤研究開発センターの Fujitsu PRIMEHPC FX10（以下，FX10），と京都大学学術情報メディアセンターの CRAY XC30（以下，XC30）の 2 種類である．各計算機システムの情報は表 1 に掲載している．それぞれ CPU，コンパイラ，インターコネクトなどが異なるシステムである．

表 1　FX10，XC30 のシステム構成

Table 1　System of FX10 and XC30

システム名	FX10	XC30
CPU/node	SPARC64 IXfx (1.848GHz, 16cores)	Xeon E5-2695v3 (2.3 GHz, 14cores) ×2
DRAM	DDR3-1333 32GB	DDR4-2133 64GB
ノード数	768	416
Interconnect	Tofu Interconect (双方向 5GB/s)	Aries (片方向 15.7GB/s)
OS	XTCOS	Cray Compute Node Linux
Compiler	Fujitsu Fortran Compiler ver. 1.2.1-09	Cray Compiler ver. 8.3.9
Compiler option	-Kfast,openmp	-O3 -h omp
MPI	Fujitsu MPI ver. 1.2.1-09	Cray MPT (MPI) ver. 7.1.3

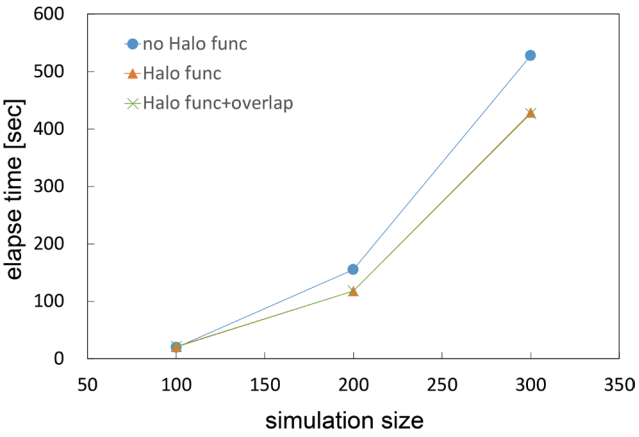

図 4　FX10 における Halo 関数を利用しない場合，Halo 関数を導入した場合とオーバーラップをさせた場合の測定結果

Figure 4　Performance measurement of no Halo function, Halo function, and Halo function with overlap technique on FX10.

計測に利用したプロセスは16MPI並列（2×2×4の3次元領域分割）で，計算サイズは各プロセスに100^3，200^3，300^3，の3次元グリッドを割り当てた．スレッド数はFX10が8スレッド，XC30では7スレッドを利用した．

図4にFX10におけるHalo関数を利用しない場合（no Halo func），Halo関数を導入した場合（Halo func），さらにオーバーラップをさせた場合（Halo func+overlap）の計算サイズの違いによる測定結果を載せる．100^3の計測結果では，ほぼHalo関数導入効果が見えないが，約2秒の差があり，10%の性能劣化となっている．200^3, 300^3では明らかな違いが出ており，それぞれ24%と19%の性能向上となっている．オーバーラップ導入効果は100^3と200^3の場合には出ておらず，300^3の場合に約1%の性能向上が見えた．

図5にXC30を利用した計測結果を載せる．書式は図4と同じである．XC3の場合はFX10ほど性能に差が出ていない．100^3ではHalo関数を導入すると49%性能劣化，200^3では4%の劣化，300^3では逆に5%の性能向上となっている．一方でオーバーラップを加えた場合は，100^3で20%の性能劣化になり，200^3では7%の性能向上が見られた．300^3ではオーバーラップの効果は見えなかった．

Haloスレッド導入した場合の計算時間は，max（Haloスレッドでの計算時間，計算スレッドの計算時間）で決まる．このため，Haloスレッドでの計算時間が元々計算スレッドでの計算時間に比べて小さい場合には，総計算時間に影響が無い．この場合，Halo関数導入した際に性能が変わらない．これに関連し，オーバーラップ導入効果が出る場合はHaloスレッドでの計算時間が計算スレッドの計算時間より大きい場合に効いてくると考えられる．また，計算サイズが小さい場合（100^3）には通信量に比べて，Halo関数のオーバーヘッド大きく出ていると考えられる．

表 2　計算サイズ変化時の Halo スレッドと計算スレッドの経過時間．表中で H thread は Halo thread，C thread は Calc thread を表す．

Table 2 Elapse time of Halo and calculation threads with the variation of calculation size. H thread is Halo thread and C thread is Calc thread in the table.

System	FX10			XC30		
grid 数/process	100^3	200^3	300^3	100^3	200^3	300^3
1st elapse at H thread [s]	0.1420	0.5426	1.6211	0.0692	0.3338	1.1519
1st elapse at C thread [s]	0.1025	0.9183	3.1557	0.0443	0.3373	1.2536
2nd elapse at H thread [s]	0.1509	0.6342	2.0524	0.0876	0.4067	1.4413
2nd elapse at C thread [s]	0.1172	1.0586	3.5420	0.0568	0.4743	1.9107
Overlapped sampling time [s]	21.6959	118.1713	427.1017	9.3762	53.6411	202.5846
No overlapped Sampling time [s]	20.0566	117.9559	428.8563	11.6829	59.9288	202.6182

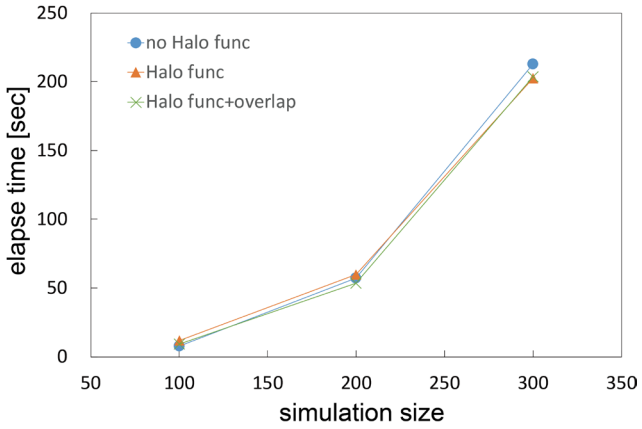

図 5　XC30 における Halo 関数を利用しない場合，Halo 関数を導入した場合とオーバーラップをさせた場合の測定結果

Figure 5　Performance measurement of no Halo function, Halo function, and Halo function with overlap technique on XC30.

表2にオーバーラップ有りの場合のHaloスレッドと計算スレッドの経過時間を載せる．Haloスレッドの計算時間が計算スレッドの計算時間より少ない場合はオーバーラップの効果はあまり見えておらず，逆の場合にオーバーラップによる性能向上が見えている．

5. まとめ

本研究では宇宙プラズマを扱う MHD シミュレーションという stencil 計算に対して，いわゆる袖領域である Halo 領域での処理を専門に担当する Halo スレッドを導入し，そのスレッドでの効率的な通信関数を開発し，その効果を調べた．Halo スレッドは通信スレッドと異なり，Halo 領域の通信だけでなく，その領域での計算を担当することで計算スレッドとの同期が必要無いこと，また，計算も担当することで 1 スレッド分計算スレッドが減り計算性能が下がる影響を抑えていることである．

以前の結果より，Halo スレッド導入はスレッド数と計算サイズのバランスが重要ではあるが，基本的には weak scaling であればスケーラビリティは劣化せず，strong scaling においても通信時間が計算スレッドの計算時間より大きくならない条件であれば，スケーラビリティは劣化しない結果が得られている．Halo スレッドは計算スレッドから独立しているので，Halo スレッドのみに対して，通信や計算の最適化が行える．本研究ではこの Halo スレッドに対して，Halo 関数群を新しく開発し，その性能を評価した．Halo 関数群は Halo 通信を効率的に行うための関数であり，通信と計算のオーバーラップにも対応しやすい構造である．

FX10 と XC30 において Halo 関数の効果を調べたところ，計算サイズが小さい場合は効果が出ず，サイズが大きい場合に効果が出やすかった．計算サイズが小さい場合は，通信時間が短くなるため，Halo 関数で通信を行う際のオーバーヘッドが現れやすいためと考えられる．Halo 関数導入効果が出る場合は，Halo スレッドでの計算時間が計算スレッドの計算時間より大きい場合と考えられる．このため，今まで Halo スレッドを導入して性能が出ない場合（Halo スレッドの計算時間が長い場合）に，Halo 関数，特にオーバーラップさせると効果がある．これにより Halo スレッドの効果がある条件を広げることが可能となった．

本研究で開発した Halo 関数群や Fortran で利用するための wrapper は ACE Project で公開する予定である．これらを利用した MPI を明示的に呼ばずに Stencil で並列計算可能なフレームワークも準備する予定である．

謝辞　本研究の計算結果は九州大学情報基盤研究開発センターと京都大学学術情報メディアセンターの計算機システムを利用して得られた．本研究は JST，CREST の研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」の研究課題「省メモリ技術と動的最適化技術によるスケーラブル通信 ライブラリの開発」の支援を受けている．

参考文献

[1] Sur S, Jin HW, Chai L and Panda DK, RDMA read based rendezvous protocol for MPI over Infiniband: Design alternatives and benefits. In: ACM SIGPLAN symposium on principles and practice of parallel programming, (PPOPP 2006) (ed J Torrellas and S Chatterjee), New York, USA, 29-31 March 2006, pp. 32-39. New York: ACM Press.
[2] Preissl R, Wichmann N, Long B, Shalf J, Ethier S and Koniges A, Multithreaded global address space communication techniques for gyrokinetic fusion applications on ultra-scale platforms. In: 2011 international conference for high performance computing, networking, storage and analysis (SC '11), Seattle, USA, 14-17 November 2011. New York: ACM Press.
[3] Idomura, Y., Nakata, M., Yamada, S., Machida, M., Imamura, T., Watanabe, T., Nunami, M., Inoue, H., Tsutsumi, S., Miyoshi, I., Shida, N.: Communication-overlap techniques for improved strong scaling of gyrokinetic Eulerian code beyond 100k cores on the K-computer. Int. J. High Perform. Comput. Appl. 28, 73-86, 2013.
[4] 深沢圭一郎，森江善之，曽我武史，高見利也，南里豪志，エクサスケールコンピューティングに向けた Halo スレッドの電磁流体シミュレーションに対する効果，情報処理学会研究報告, 2015-HPC-151(23), 1-6, 2015.
[5] Chang, C. L. and Lee, R. C. T.: Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.
[6] R. O. Dendy,『Plasma Dynamics』, Oxford University Press, 1990.
[7] T. Ogino, R. J. Walker, M. Ashour-Abdalla, A global magnetohydrodynamic simulation of the magnetopause when the interplanetary magnetic field is northward, IEEE Trans. Plasma Sci.20, 817.828, 1992.
[8] Fukazawa, K., T. Ogino, and R.J. Walker, "The Configuration and Dynamics of the Jovian Magnetosphere", J. Geophys. Res., 111, A10207, 2006.
[9] 青山幸也，並列プログラミング虎の巻 MPI 版.
[10] ACE Project
http://ace-project.kyushu-u.ac.jp/main/jp/index.html

付録

付録 A.1 Halo 関数

int halo_init(int ldim, int *lsz, int *l2m, int mdim, int *msz, double *mtx, int *lc, struct halo_hnd_t *hnd)

・機能 : halo 通信初期化関数
・引数
ldim : halo 通信に用いる並列プロセスの論理分割次元数．整数．全プロセス同じ値を指定する．
lsz : プロセス分割における次元毎のプロセス数．ldim 数個の整数からなる一次元配列, lsz の全要素の乗数が袖通信に参加する全並列プロセス数になる．全プロセス同じ値を指定する．
l2m : 論理プロセス分割次元軸と処理行列の次元軸との対応表．ldim 数個の整数からなる一次元配列で，配列の要素番号が論理分割次元軸番号を示し，各配列要素の整数値が行列の次元軸番号となる．
mdim : 行列の次元数．整数．
msz : 行列の次元毎の大きさ．mdim 数個の整数からなる一次元配列．
mtx : 袖領域を含む行列を表す mdim 次元配列の先頭アドレス．
lc : 呼び出したプロセスに割り当てられた論理座標，ldim 数個の整数配列．
hnd : この呼び出しで初期設定された通信の識別子．
・返り値
0 : 正常終了
それ以外 : エラー(次元数)

int halo_isend(int dir, int *area, struct halo_hnd_t *hnd, struct halo_req_t *req)

・機能 : 袖領域送信関数
・引数
dir : 送信先となる隣接プロセスの方向，enum 型整数 HL_xyz の書式で指定する．x,y,z には方向を示す文字を入れる(I->+1,N->0,D->-1).
area : 送信する範囲．一次元整数配列．
hnd : 送信対象のハンドル．halo_init 関数の hnd 変数値．
req : 通信確認用の識別子．構造体ポインタ．
・返り値
0 : 正常終了
それ以外 : エラー(ハンドル，方向，領域，下位関数)

int halo_irecv(int dir, int *area, struct halo_hnd_t *hnd, struct halo_req_t *req)

・機能 : 袖領域受信関数
・引数
dir : 送信元となる隣接プロセスの方向，enum 型整数 HL_xyz の書式で指定する．x,y,z には方向を示す文字を入れる(I->+1,N->0,D->-1).
area : 受信する範囲．一次元整数配列．
hnd : 受信対象のハンドル．halo_init 関数の hnd 変数値．
req : 通信確認用の識別子．構造体ポインタ．
・返り値
0 : 正常終了
それ以外 : エラー(ハンドル，方向，領域，下位関数)

int halo_wait(struct halo_req_t *req)

・機能 : 通信完了待ち関数
・引数
req : 通信確認用の識別子．構造体ポインタ．
・返り値
0 : 正常終了
それ以外 : エラー(ハンドル，方向，領域，下位関数)

void halo_finalize(struct halo_hnd_t *hnd);

・機能 : halo 通信終了関数
・引数
hnd : halo_init で設定された通信の識別子.