Asynchronous Progress Control による計算と通信のオーバーラップ：パイプライン型共役勾配法への適用事例

堀越 将司(インテル)
中島 研吾(東京大学情報基盤センター)

1. はじめに

本稿は 2019 年 5 月 30～31 日に実施された「大規模 HPC チャレンジ（Oakforest-PACS)」の結果をまとめたものである。本研究では，スケーラビリティの高い手法として注目されているパイプライン型アルゴリズムに基づく共役勾配法（Conjugate Gradient（CG） based on Pipelined Algorithms）〔1〕の Oakforest-PACS システム（最先端共同 HPC 基盤施設）〔2〕上での実装に関する検討を実施した。本稿では，まず，対象とするアプリケーション，アルゴリズムの概要について，先行研究〔3〕の結果も交えて紹介し，6 章以降で，Oakforest-PACS における実行手法と結果について紹介する。

有限要素法に代表される偏微分方程式の数値解法において，最も計算時間を要するプロセスは大規模な疎行列を係数行列とする連立一次方程式の求解であり，その最適化に向けて様々な試みがなされてきた．連立一次方程式の求解には共役勾配法（Conjugate Gradient，CG）に代表されるクリロフ部分空間法（反復法）が広く使用されているが，大規模並列計算においては通信によるオーバーヘッドが顕著となる。図 1 は連立一次方程式 Ax=b を前処理付き共役勾配法で解くアルゴリズムである：

```
1: r0:= b-Ax0; u0:= M^-1 r0; p0:= u0
2: for i=0 … do
3:     s:= Api
4:     α:= (ri,ui)/(s,pi)
5:     xi+1:= xi + αpi
6:     ri+1:= ri- αs
7:     ui+1:= M^-1 ri+1
8:     β:= (ri+1,ui+1)/(ri,ui)
9:     pi+1:= ui+1 + βpi
10: end do
```

図 1 前処理付き共役勾配法（Preconditioned Conjugate Gradient，PCG）のアルゴリズム（Alg.-1），Ax=b，M：前処理行列，赤字：疎行列ベクトル積（SpMV, Spars Matrix Vector Multiply），青字：内積，緑字：前処理

分散メモリ型並列計算機上で並列計算を実施する場合は：

- 疎行列ベクトル積
- 内積
- 前処理

で通信が発生する可能性がある。前処理手法によって，通信パターンは異なるが，疎行列ベクトル積では隣接プロセスとの 1 対 1 通信（Point-to-Point Communication），内積では全プロセス

との集合通信（Collective Communication）が発生する。MPI を使用する場合は，前者は MPI_Isend や MPI_Irecv，後者は MPI_Allreduce のような関数が使用される。

通信によるオーバーヘッドを削減するために，通信回避・削減型アルゴリズム（Communication Avoiding/Reducing Algorithm）の研究開発が盛んに進められている。Matrix Powers Kernel〔4〕に基づく *s*-step 法〔5〕では，*s*（*s*>1）反復分の通信を 1 反復で済ませる方法であり：

- 内積実施回数削減による集団通信オーバーヘッド削減
- 並列分散データの Halo 領域を大きくとって，*s* 回分の行列ベクトル積を，通信をしないで連続して実施する

手法である。一般に *s* が大きくなると冗長計算が増加するとともに不安定性が増す。また *s*-step 法は，適用可能な前処理手法が限定されるという問題点がある。

また，陽解法で広く使用されている，通信と計算のオーバラップ〔6〕の手法を疎行列ベクトル積に適用することも可能であるが，通信オーバーヘッドの削減効果をあげることは困難である。

本研究で扱うパイプライン型共役勾配法（Pipelined CG Method）〔1〕は，*s*-step 法の元になったアルゴリズムに基づき，漸化式の適用によって，図 1 に示したオリジナルの前処理付き CG 法のアルゴリズムが変わらないように計算の順序を変更する手法である。内積の直後に疎行列ベクトル積，前処理などの計算量が多く，かつ直前で実施した内積の結果を使わないような処理を実行するように計算順序が変更される。Pipelined 法では，これに MPI-3 でサポートされている MPI_Iallreduce 等の非同期集団通信（Asynchronous Collective Communication）〔7，8〕を組み合わせることによって，集団通信と疎行列ベクトル積，前処理等の演算をオーバーラップすることができ，通信によるオーバーヘッドの隠蔽が可能となる。

疎行列ベクトル積は，限定された数の隣接プロセスとの通信であるが，集団通信は全プロセスに対する通信であり，大規模並列計算機システムにおいてノード数が増加すると，オーバーヘッドはより顕著となる。図 1 に示すオリジナルの前処理付き共役勾配法では，内積 1 回の通信量はスカラー変数 1 つ分であり，いわゆるレイテンシの効果が大きい。

〔3〕では〔1〕に示されたパイプライン型共役勾配法を，三次元有限要素法構造解析コードへ適用し，Reedbush-U（RBU）（東京大学情報基盤センター）〔9〕，Oakforest-PACS（OFP）〔7〕を使用して実施した性能評価事例を紹介している。以下，対象アプリケーション，数値アルゴリズム，計算機環境，問題設定，計算結果について述べる。

2. 対象アプリケーション：GeoFEM ベンチマーク

本研究で対象としているのは，GeoFEM プロジェクト〔10〕で開発された並列有限要素法アプリケーションを元に整備した性能評価のためのベンチマークプログラム「GeoFEM／Cube」である。本ベンチマークは，均質場における三次元弾性静解析問題（Cube 型モデル（図 2））に関する並列前処理付き反復法による疎行列ソルバーの実行時性能を様々な条件下で計測するものである。要素タイプは三次元一次六面体要素（tri-linear）であり，各要素 8 つの節点を有している。

三次元弾性問題では 1 節点あたり 3 つの自由度があるため，これらを 1 つのブロックとして取り扱っている（図 3）。係数行列はこのブロック型の特性を利用したブロック CRS 形式（Compressed Row Storage）によって格納されている。

プログラムは全て OpenMP ディレクティヴを含む Fortran90 および MPI で記述されている。GeoFEM で採用されている局所分散データ構造〔10〕を使用しており，領域分割された各領域に MPI プロセスが割り当てられる。MPI，OpenMP，Hybrid（OpenMP＋MPI）の全ての環境で稼動する。

三次元弾性静解析問題では係数行列が対称正定な疎行列となることから，前処理を施した共役勾配法（Conjugate Gradient, CG）法によって連立一次方程式を解いている。前処理手法としては，各 MPI プロセスの扱う領域に対して，局所化されたブロックヤコビ型 Symmetric Gauss Seidel（SGS）法を適用している（図 4）〔10,11〕。

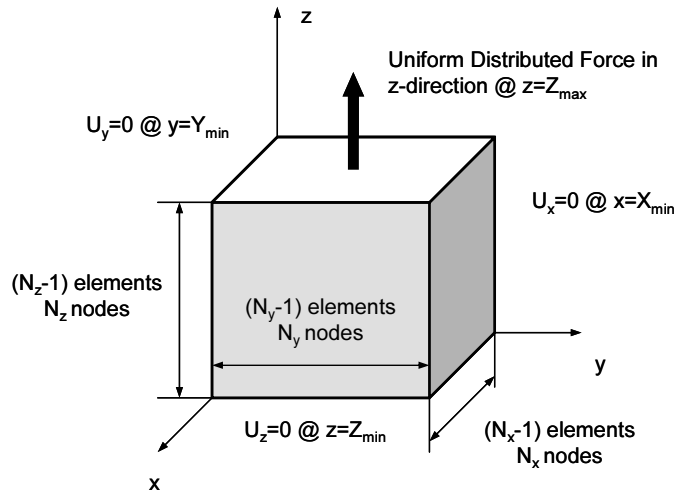

図 2　GeoFEM／Cube の解析対象（Cube モデル）

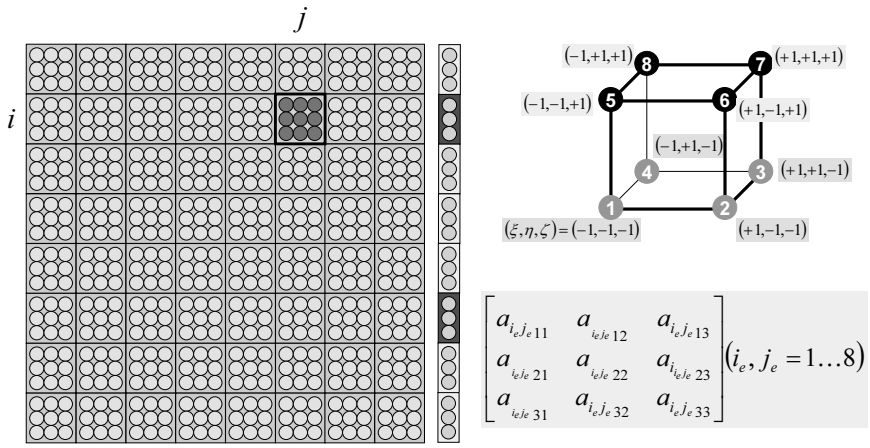

図 3　8 節点六面体要素，1 節点 3 自由度の処理

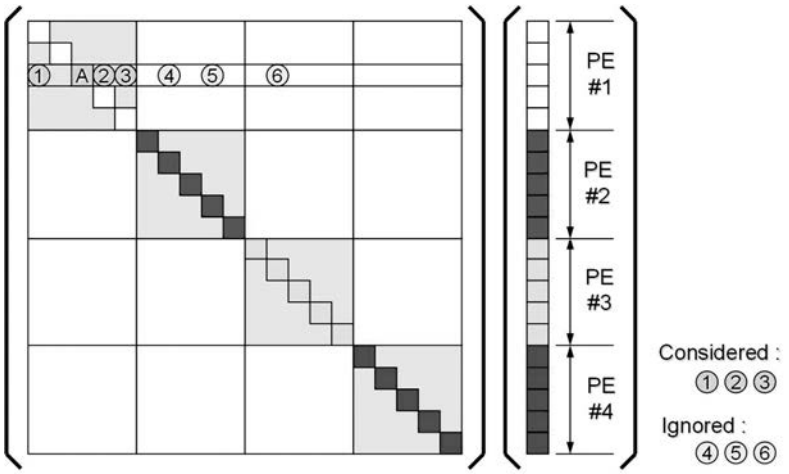

図 4　ブロックヤコビ型局所 SGS 前処理〔10，11〕

SGS 前処理は，前進代入，後退代入のプロセスでメモリへの書き込みと参照が同時に生じ，データ依存性が発生する可能がある。これを回避するための方法として色づけ（coloring）によるリオーダリング（reordering）が広く使用されている〔12，13〕。お互いに依存性を持たない要素群を同じ色に色づけすることによって，色内での並列処理が可能となる。本研究では，並列性に優れたマルチカラー法（Multicoloring, MC）とより安定した収束を示す Reverse Cuthill-McKee（RCM）法を組み合わせ，RCM 法に Cyclic マルチカラー法（Cyclic Multicoloring，CM）を適用した CM-RCM(k)法を使用している〔11,12〕（一部のケースでは RCM 法を使用している）。図 5 は CM-RCM(k)法による並び替え例である。ここでは，4 色に色分けされており（CM-RCM(4)），たとえば，RCM の第 1，第 5，第 9，第 13 組の要素群が CM-RCM(k)法の第 1 色に分類される。各色には 16 の要素が含まれる。CM-RCM(k)法における色数は，各色内の要素が依存性を持たない程度に大きい必要がある。

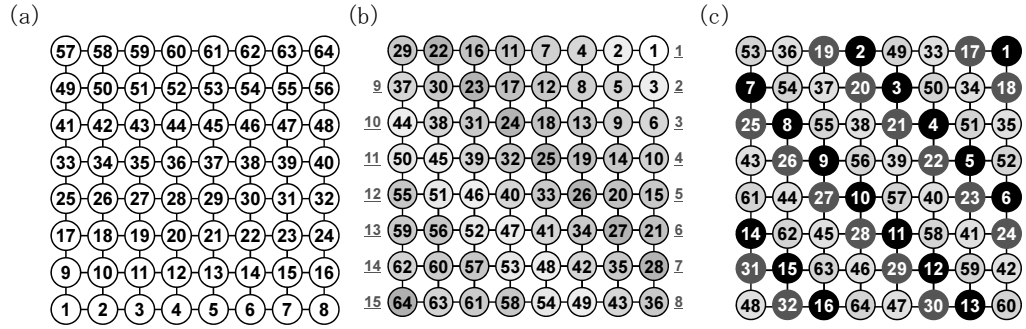

図 5　CM-RCM(k)法による色づけとリオーダリング，(a) 元のグラフ，(b) RCM 法によるリオーダリング（赤字はレベルセット番号），(c) CM-RCM(k)法による再リオーダリング（4 色：CM-RCM(4)），各色内の要素数は 16 でバランス

局所 SGS 前処理法を安定化させる手法として，領域間オーバーラップ領域に加法型 Schwartz 領域分割法（Additive Schwartz Domain Decomposition：ASDD）〔13〕を適用している。ASDD の手順は以下のとおりである：

(1) 共役勾配法の前処理の計算を実施する（図 1 参照）：

$$u = M^{-1} r$$

M：前処理行列，r,z：ベクトル，である。

前処理計算 M^{-1} は SGS 法の前進・後退代入に相当する

(2) 全体領域が図 6 に示すように Ω_1 および Ω_2 の 2 領域に分かれているものと仮定すると前処理（前進・後退代入）は各領域において，以下のように局所的に実施される。

$$u_{\Omega_1} = M_{\Omega_1}^{-1} r_{\Omega_1}, \quad u_{\Omega_2} = M_{\Omega_2}^{-1} r_{\Omega_2}$$

(3) 局所的な前処理を実行したのちに，領域間オーバーラップ領域 Γ_1 および Γ_2 において以下の計算を実施する（図 6（b））：

$$u_{\Omega_1}^{n} = u_{\Omega_1}^{n-1} + M_{\Omega_1}^{-1}(r_{\Omega_1} - M_{\Omega_1} u_{\Omega_1}^{n-1} - M_{\Gamma_1} u_{\Gamma_1}^{n-1})$$

$$u_{\Omega_2}^{n} = u_{\Omega_2}^{n-1} + M_{\Omega_2}^{-1}(r_{\Omega_2} - M_{\Omega_2} u_{\Omega_2}^{n-1} - M_{\Gamma_2} u_{\Gamma_2}^{n-1})$$

ここで n は ASDD の繰り返し数である。

(4) (2) および (3) のプロセスを収束まで繰り返す。

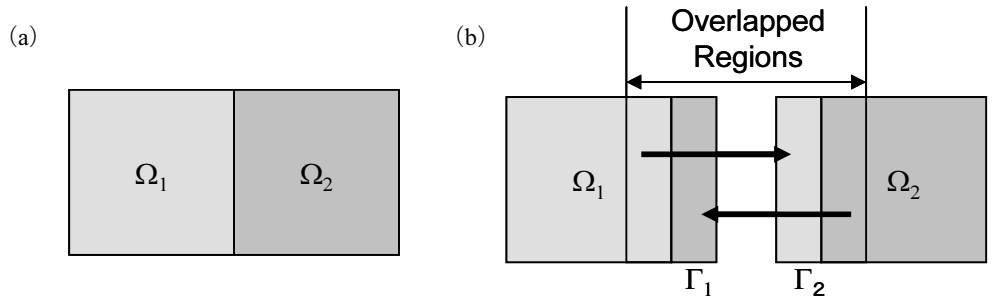

図 6　二領域間における加法型 Schwartz 領域分割法（Additive Schwartz Domain Decomposition, ASDD）処理の概要〔13〕：(a) 局所処理，(b) 大域的修正

3. パイプライン型共役勾配法

本研究では，以下の 4 種類のアルゴリズムを使用する：

- オリジナル前処理付き共役勾配法（Alg.-1）（図 1）
- Chronopoulos/Gear アルゴリズム（Alg.-2）〔1,5〕
- パイプライン型アルゴリズム（Alg.-3）〔1〕
- Gropp アルゴリズム（Alg.-4）〔15〕

Chronopoulos/Gear アルゴリズム（Alg.-2）は s-step 法〔2〕において s=1 としたもので，結果として，図 7 に示すようなアルゴリズムが得られる。このアルゴリズムの特徴は，γ_i, δという内積処理を連続して実施できることにある（図 7 の 10 行目，11 行目）。したがって，実装上は MPI_Allreduce を 1 回呼ぶだけで済むため，全 MPI プロセスが関わる集合通信の回数を 1 回削減することができる：

```
1: r0:= b-Ax0; u0:= M^-1 r0; w0:= Au0
2: α0:= (r0,u0)/(w0,u0); β0:=0; γ0:= (r0,u0)
3: for i=0 … do
4:     pi:= ui + βi pi-1
5:     si:= wi + βi si-1
6:     xi+1:= xi + αi pi
7:     ri+1:= ri- αi si
8:     ui+1:= M^-1 ri+1
9:     wi+1:= Aui+1
10:    γi+1:= (ri+1,ui+1)
11:    δ:= (wi+1,ui+1)
12:    βi+1:= γi+1/ γi
13:    αi+1:= γi+1/ ( δ -  βi+1 γi+1/αi)
14: end do
```

図 7　Chronopoulos/Gear アルゴリズム（Alg.-2）〔1,5〕

Alg.-2 を漸化式を用いて変形すると図 8 に示すアルゴリズムが得られる。このアルゴリズムはパイプライン型 Chronopoulos/Gear アルゴリズム（前処理無し）である。

このアルゴリズムの特徴は，Alg.-2 と同様に ・$_i$，・・という内積処理を連続して実施できるだけでなく（図 8 の 3 行目，4 行目），更にこれらの内積の値は 5 行目の疎行列ベクトル積では使用されず，6 行目以降で使用されるということである。この部分に，MPI-3 でサポートされている非同期集団通信機能〔7,8〕を適用すれば，内積計算のための集合通信と疎行列ベクトル積をオーバーラップさせることが可能となる。

```
1: r0:= b-Ax0; w0:= Au0
2: for i=0 … do
3:     γi:= (ri,ri)
4:     δ:= (wi,ri)
5:     qi+1:= Awi+1
6:     if i>0 then
7:     βi:= γi/ γi-1; αi:= γi/ ( δ -  βi γi/αi-1)
8:     else
9:     βi:= 0; αi:= γi/δ
10:    end if
11:    zi:= qi + βi zi-1
12:    si:= wi + βi si-1
13:    pi:= ri + βi pi-1
14:    xi+1:= xi + αi pi
15:    ri+1:= ri - αi si
16:    wi+1:= wi - αi zi
17: end for
```

図 8　パイプライン型 Chronopoulos/Gear アルゴリズム（前処理無し）〔1〕

更に，これを漸化式を用いて変形することによって，図 9 に示すパイライン型前処理付き共役勾配法のアルゴリズム（Alg.-3）〔1〕が得られる：

```
1: r0:= b-Ax0; u0:= M^-1 r0; w0:= Au0
2: for i=0 … do
3:     γi:= (ri,ui)
4:     δ:= (wi,ui)
5:     mi:= M^-1 wi
6:     ni:= Ami
7:     if i>0 then
8:     βi:= γi/γi-1; αi:= γi/(δ - βi γi/αi-1)
9:     else
10:    βi:= 0; αi:= γi/δ
11:    end if
12:    zi:= ni + βi zi-1
13:    qi:= mi + βi qi-1
14:    si:= wi + βi si-1
15:    pi:= ui + βi pi-1
16:    xi+1:= xi + αi pi
17:    ri+1:= ri - αi si
18:    ui+1:= ui - αi qi
19:    wi+1:= wi - αi zi
20: end for
```

図 9　パイプライン型（前処理付き）共役勾配法のアルゴリズム（Alg.-3）〔1〕

このアルゴリズムも Alg.-2，図 8 に示すアルゴリズムの特性を継承しており，内積（3 行目，4 行目）の後に前処理，疎行列ベクトル積の計算があり，その後初めて内積の値を使用するため，集団通信と計算のオーバーラップが可能である。この他〔1〕で紹介されている Gropp のアルゴリズム（Alg.-4）〔15〕は，Alg.-3 と良く似たアルゴリズムであるが，δの導出法が異なっており，計算量も若干少なくなっている（図 10）。

```
1: r0:= b-Ax0; u0:= M^-1 r0; p0:= u0; s0:= Ap0;
γ0:= (r0,u0)
2: for i=0 … do
3:     δ:= (pi,si)
4:     qi:= M^-1 si
5:     αi:= γi/δ
6:     xi+1:= xi + αi pi
7:     ri+1:= ri - αi si
8:     ui+1:= ui - αi qi
9:     γi+1:= (ri+1,ui+1)
10:    wi+1:= Aui+1
11:    βi+1:= γi+1/γi
12:    pi+1:= ui+1 + βi+1 pi
13:    si+1:= wi+1 + βi+1 si
14: end for
```

図 10　Gropp アルゴリズム（Alg.-4）〔1,15〕

図 11 は Gropp アルゴリズム（図 10）の 3〜5 行目，Fortran，OpenMP，MPI で実装した例である。内積計算用集団通信関数として MPI_Allreduce ではなく MPI-3 でサポートされている非同期集団通信関数 MPI_Iallreduce を呼んでいる。内積の計算結果（δ）を使用する直前に MPI_Wait を呼んで同期しているため，集団通信と前処理計算部（4 行目）をオーバーラップすることが可能である。

表 1 は各アルゴリズムの計算量である。DAXPY はベクトルの定数倍の加減（例：$x_i = x_i + \alpha_i p_i$）である。Alg.3，Alg.4 はオリジナルの手法と比較して，計算量が増えているが，SpMV

や前処理と比較すると計算量は少なく，無視できる。

Alg.1～Alg.4 はアルゴリズム的には同じ計算内容であるが，計算順序が変わっているため，丸め誤差の伝播の仕方が異なっている。従って，悪条件問題の場合，アルゴリズムによって収束の履歴が異なる場合がある。〔1〕では，そのような事例が実際にあるため，50 反復に一回残差ベクトル（r_i）の値を $r_i = b - Ax_i$ によって補正している。本研究で対象としている問題では，そのような補正は不要で，各アルゴリズム全く同じ履歴，同じ回数で収束した。

```
!C> 3: δ:= (p_i,s_i)
      DL0= 0.d0
!$omp parallel do private(i) reduction(+:DL0)
      do i= 1, 3*N
        DL0= DL0 + P(i)*S(i)
      enddo
      call MPI_Iallreduce                              &
     &    (DL0, Delta, 1, MPI_DOUBLE_PRECISION,   &
     &     MPI_SUM, MPI_COMM_WORLD, request, err)

!C> 4: q_i:= M^-1 s_i
   （前処理：省略）

   call MPI_Wait (request, status, err)

!C> 5: α_i:= γ_i/δ
   Alpha= Gamma / Delta
```

図 11　Gropp アルゴリズムの実装例（非同期集団通信と計算のオーバーラップ）

表 1　各アルゴリズムの計算量（SpMV：疎行列ベクトル積，DAXPY：ベクトル定数倍加減），内積の（+1）は残差ベクトルのノルム計算

Alg.		SpMV	前処理	内積	DAXPY
1	Original CG	1	1	2+1	3
2	Chronopoulos/ Gear	1	1	2+1	4
3	Pipelined CG	1	1	2+1	8
4	Gropp	1	1	2+1	5

4. 先行研究における計算結果（Reedbush-U）

先行研究〔3〕では東京大学情報基盤センターの Reedbush-U システム〔9〕を使用した。Reedbush-U の計算ノードは，CPU として Intel Xeon E5-2695v4 （Broadwell-EP）を 2 ソケット搭載している。CPU の各コアには，AVX2 命令に対応したベクトルユニットが 2 基ずつ搭載されている。これらの計算ノード 420 ノードは，InfiniBand-EDR によりフルバイセクションバンド幅を持つ Fat-tree 網で接続されている。Intel Compiler，Intel Parallel Studio に含まれる Intel MPI を使用した。表 2 は各ソケットの概要である。

表 2　Reedbush-U 各ソケットの概要

CPU	Intel Xeon E5-2695 v4　(Broadwell-EP)
動作周波数（GHz）	2.10
コア数	18
理論演算性能（GFLOPS）	604.8
主記憶容量（GB）	128
メモリバンド幅性能（GB/sec, Stream Triad）	65.5
インタコネクト	Mellanox InfiniBand EDR 4x(100 Gbps) Full-bisection BW Fat-tree

Reedbush-U（RBU）を使用した計算の概要は表 3 の通りである。1 ソケットの 18 コアのうち 16 コアを使用し，最小 2 ノード（4 ソケット，64 コア）から最大 384 ノード（768 ソケット，12,288 コア）までの Strong Scaling 性能を評価した。

表 3　Reedbush-U における計算の概要

問題規模	Small：256×128×144 節点（14,155,776 DOF） Medium：256×128×288 節点（28,311,552DOF）
利用ノード数	最小 2 ノード，4 ソケット 最大 384 ノード，768 ソケット 1 ソケットあたり 16 コア使用，最大 12,288 コア Small : コア当り最小問題サイズ＝1,152 DOF/core Medium：同＝2,304 DOF/core
並列プログラミングモデル	Flat MPI：1 コアあたり 1MPI プロセス Hybrid : 1 ソケットあたり 1MPI プロセス，16 スレッド
ノード内リオーダリング法	RCM
ASDD 反復数	2 回
CG 法収束条件	$\|r\|/\|b\| = \|b-Ax\|/\|b\| < 1.0\times10^{-8}$

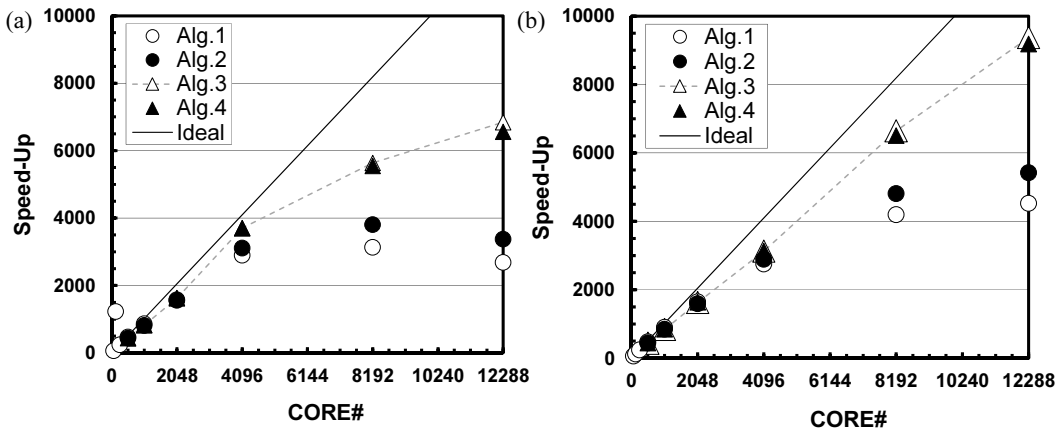

図 12 並列 CG 法の計算性能（Hybrid），Flat MPI　2 ノード（4 ソケット，64 コア）（Alg.1）の性能を 64.0 としたときの速度向上率（a）Small，（b）Medium

ブロックヤコビ型局所 SGS 前処理を適用しているため，コア数が増加すると反復回数が増加する傾向があるものの，ASDD（加法型 Schwartz 領域分割法）の効果によって Hybrid の場合は

64 コアから 12,288 コアまで 10%程度の反復回数増加に留まっている〔3〕。図 12 は Hybrid の 384 ノード，12,288 コアまでの速度向上率である。Flat MPI　2 ノード（4 ソケット，64 コア）の Alg.1 の性能を 64.0 としている。Small，Medium のいずれの場合にも Alg.1 の速度向上率が低く，Alg.2 がややそれより良く，Alg.3，Alg.4 は更に良く，両者の速度向上率はほぼ同じである。Medium での Alg.3，Alg.4 の速度向上率は 12,888 コアで 9,000 を超えており，約 75%の並列化効率が得られている。

図 13 は Alg.3，Alg.4 において非同期集団通信関数（MPI_Iallreduce）を使った場合（IAR）と通常の MPI_Allreduce を使った場合（AR）と Alg.1 を比較したものである。非同期集団通信関数を使用しない場合は，Alg.1 とほぼ同じ挙動を示していることがわかり，非同期集団通信機能の適用により，通信と計算のオーバーラップを実現し，大幅な性能向上が得られていることがわかる。

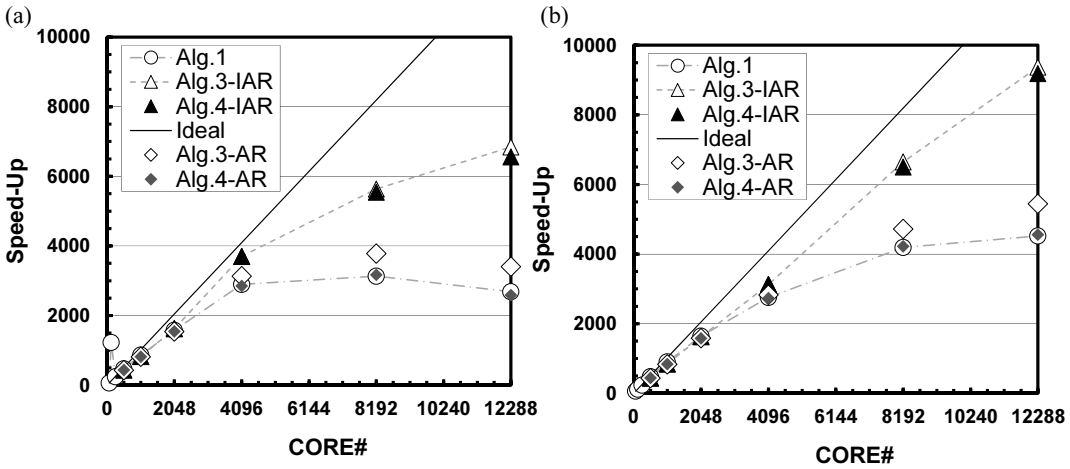

図 13　並列 CG 法の計算性能（Hybrid），Flat MPI　2 ノード（4 ソケット，64 コア）（Alg.1）の性能を 64.0 としたときの速度向上率（a）Small，（b）Medium, IAR：MPI_Iallreduce 使用，AR：MPI_Reduce 使用

5. Oakforest-PACS における検討

先行研究〔3〕では，更に Oakforest-PACS での検討も実施しているが，128 ノード 8,192 コアの実行において，非同期集団通信関数（MPI_Iallreduce）を使用した実装と同期通信関数（MPI_Allreduce）を用いた実装とで顕著な差が見られず，Oakforest-PACS のノード間インターコネクト（Omni-Path Architecture）が CPU への負荷依存の大きい On load タイプのインターコネクトであることが原因の一つとして考えられる。Reedbus-U では InfiniBand EDR（Off load タイプ）および Xeon CPU が使用されているが，Oakforest-PACS の CPU（Intel Xeon/Phi，Knights Landing）が Reedbush-U の CPU（Intel Xeon）と比較して，低速なコアであることも考えられる。CPU 依存の大きな On load のインターコネクトを低速なコアの集合であるメニーコアアーキテクチャで処理しているため，通信レイテンシは Xeon と比較すると CPU 依存がより大きくなる。

Oakforest-PACS については，2016 年 10 月の運用開始当初からシステムや MPI の最適化が行われてきたが〔16〕，非同期通信への最適化は Oakforest-PACS 上でこれまでおこわなれていなかった。MPICH に hand-off モデル〔17〕が新たに採用され，効率的な非同期通信（計算と通信のオ

ーバーラップ）が可能となった。Intel MPI は，バージョン 2019 よりこの hand-off モデルをベースとした Asynchronous progress thread を実装している。本研究では Intel MPI 2019 の新機能である，Asynchronous progress thread を Oakforest-PACS 上で検証した結果を紹介する。

表 4 は，GeoFEM の 128 ノード（8,192 コア）から 2048 ノード（131,072 コア）までの Strong Scaling における，実行時間中の集団通信（MPI_Allreduce）の割合である。ノード数の増大により，集団通信の割合が大きくなっており，これが大規模時の性能ボトルネックとなっている。

表 4 GeoFEM における集団通信の割合

Core#	CG loop [sec]	Allreduce [sec]	Allreduce Ratio [%]
8192	15.34	2.26	14.75
16384	9.89	1.56	15.73
32768	7.29	1.89	25.94
65536	5.93	2.32	39.15
131072	6.53	3.88	59.50

本研究で使用した Oakforest-PACS の計算ノードは Intel Xeon Phi 7250 (Codename: Knights Landing または KNL)1 ソケットにより構成されるメニーコアシステムである。各コアには AVX512 命令に対応した SIMD 演算ユニットが 2 基ずつ搭載されている。これらのコア 2 つで L2 キャッシュを共有し，1 つのタイルを構成している。タイルは，メモリコントローラや PCI-Ex コントロール等と併せて 1 チップ内で 2 次元メッシュ上にせず属されている。

KNL の CPU 上には，MCDRAM と呼ばれる 3 次元積層メモリが搭載され，DDR4 メモリと比較し 5 倍以上のメモリバンド幅を持つ。MCDRAM は，メモリして使用可能な Flat モードとキャッシュとして利用可能な Cache モードがある。本研究では Flat モードを使用し，numactl により MCDRAM への明示的な割付を行った。

表 5　計算環境の概要

システム名	Oakforest-PACS（OFP）
CPU	Intel Xeon Phi 7250 (Knights Landing)
動作周波数（GHz）	1.40
コア数（最大有効スレッド数）	68 (272)
理論演算性能（GFLOPS）	3,046.4
主記憶容量（GB）	MCDRAM: 16，DDR4: 96
メモリ性能（GB/sec.，STREAM Triad）	MCDRAM: 490，DDR4: 84.5
インターコネクト	Intel OmniPath Architecture (100Gbps) Full-bisection BW fat-tree

OFP では OS ジッタの影響を最小化するため，タイマ割り込みを受け付けないコアを指定する，full tickless の設定を行っている。タイマ割り込みを受け付けるコアは 0 に設定されているが，L2 キャッシュを共有するコア 1 も同様に避けて割当をするようにし，性能改善を図っている。

プログラムは Fortran90 で記述してあり，Intel Compiler 2019 Update1 および Intel MPI 2019 Update1 を使用して，ノード内並列には OpenMP を使用している。表 5 に計算機環境の概要を示

す。計算ノード間インターコネクトは Intel OmniPath Architecture を用いた 100Gnbps リンクであり，フルバイセクションバンド幅を持つ fat-tree 網で構成されている。

6. Asynchronous Progress Thread 設定

Intel MPI2019 から利用可能となった Asynchronous Progress Thread の利用方法を説明する。実行アプリケーションがスタティックリンクを用いている場合，バージョン 2019 にて再コンパイルが必要となるが，ダイナミックリンクの場合は再コンパイルを必ずしも必要しない。その場合，実行時にロードされる libmpifort.so.12 等の共有ライブラリをバージョン 2019 に指定するだけで良い。

Asynchronous Progress Thread を効率よく使用するためには，MPI-3 の非同期通信をアプリケーションに実装する必要があるが，性能を出すためには通信専用にコアを設ける必要がある。実際はアプリケーションや通信パターン，サイズに依るところがあるが，必ずしも通信コアを必要とするわけではない。本研究で使用したアプリケーションでは通信用コアを用意しなければ，顕著な性能向上は見られなかった。コアの割付については後述する。

OFP におけるジョブスクリプトについて説明する。図 14 はノードあたり MPI 8 プロセス，OpenMP 8 スレッドを使用したハイブリッド並列のスクリプトである。

```
#1
source /work/opt/local/cores/intel/impi/2019.1.144/intel64/bin/mpivars.sh release_mt

#2
export I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=0-3,68-71,136-139,204-207

#3
export KMP_HW_SUBSET=1T; export OMP_NUM_THREADS=8; export I_MPI_PIN_DOMAIN=32

#4
export I_MPI_FABRICS=shm:ofi

#5
export I_MPI_ASYNC_PROGRESS=on

#6
export I_MPI_ASYNC_PROGRESS_PIN="0,1,66,67,68,69,134,135”

#7
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -m 1 ./your_job.out
```

図 14 各ノード MPI 8 プロセス, OpenMP 8 スレッドを使用したハイブリッド並列のスクリプト

まず，#1 で環境の読み込みを行う。OFP の module load では mpivars.sh release が設定されるため，Asynchronous Progress Thread は有効とならない。そのため，明示的に release_mt の環境を指定する必要がある。#2 では，ジッタコアを避けるために設定をしている。I_MPI_PIN_PROCESSOR_EXCLUDE_LIST で指定した論理コア（この例では先頭の物理コア 2 つ）をアプリケーションの実行から排除する。#3 では OpenMP 周りの設定を行っており，各コア 1 論理スレッドの使用（export KMP_HW_SUBSET=1T），スレッド数の指定 (OMP_NUM_THREADS=8)，プロセスあたりの OpenMP ドメイン(I_MPI_PIN_DOMAIN)の設定を行っている。I_MPI_PIN_DOMAIN は，1 つの CPU を 8 つのプロセスでパーティションする

ために，256 の論理コア（64 物理コア）を 8 で割った 32 を指定する。#4 はインターコネクトの設定である，バージョン 2018 までは OFP 上では shm:tmi が使用されていたが，バージョン 2019 では shm:ofi が必要となる。#5 は，Asynchronous Progress Thread を有効にする環境変数である。#6 において，非同期通信に使用するスレッドの割付を明示的に指定する。明示的に使用しない場合，デフォルトでは，論理コアの後方から（OFP では 217 番の論理コアから）割付が行われる。ここでは，先頭 2 論理コア(0,1)および後方 2 論理コア(66, 67)，そしてそれらのコアの 2 番目の論理スレッド(68-69,. 134-135)を割り付け，計 8 つの論理コアを指定している。この例では 8 つの MPI プロセスが通信するため，8 つの論理コアを割付ける必要がある。最後に#7 にてアプリケーションを実行する。umactl -m 1 にて明示的に MCDRAM メモリへの割付を行っている。Asyncronous Progress Thread の論理コアへの割付はいくつかの方法が考えられる。表 6 に，論理コアへのマッピングをまとめた。

表 6 通信スレッド(Asynchronous Progress Thread)の割付

Mapping	I_MPI_ASYNC_PROGRESS_PIN
1: Non async prog. thread	N/A
2: default	(271 to 264)
3: Last tile	66,67,134,135,202,203,270,271
4: 3rd HT of MPI process	138,146,154,162,170,178,186,192
5: 4th HT of MPI process	206,214,222,230,238,246,254,262
6: First tile	0,1,68,69,136,137,204,205
7: First and last tiles	0,1,68,69,66,67,134,135

1:Non async prog. thread は I_MPI_ASYNC_PROGRESS=off とした時のものであり，I_MPI_ASYNC_PROGRESS_PIN は実質的に意味を持たない。2:default は I_MPI_ASYNC_PROGRESS=on とした時の既定値であり，I_MPI_ASYNC_PROGRESS_PIN を指定しなければ，論理コアの最後尾から必要な数の論理コアが自動で選択される（本ケースでは最後尾より 8 論理コア）。3:から 7:は I_MPI_ASYNC_PROGRESS=on とした時に明示的に論理コアを Asynchronous Progress Thread へ割り付けた場合を示している。3:Last tile は，計算に使用されていない最後のタイル（物理コア 66-67）を通信スレッドに割り当てる。4:3rd HT of MPI process は，MPI プロセスが起動される物理コア上の 3 番目の論理スレッドに割り当てる。5: 4th HT of MPI process も同様に，MPI プロセスが起動される物理コア上の 4 番目の論理スレッドに割り当てる。4:および 5:では，計算に使用される物理コア内で計算と通信のスレッドが同時に立ち上がる。6:First tile は，計算に使用されていない最後のタイル（物理コア 0-1）を通信スレッドに割り当てる。3:および 6:のマッピングでは，物理コアあたり 4 つすべての論理スレッドを通信スレッドに割り当てることを意味する。最後の 7:First and last times は，最初と最後のタイル(物理コア 0-1, 66-67)を通信スレッドに割り当てるが，物理コアが 4 つあるので，物理コアあたり 2 つの通信スレッドを割り当てることになる。これら 7 つのマッピングパターンで非同期通信スレッドの割当の効果を検証した。

図 15 に例として，マッピング 7:の場合の計算スレッドと通信スレッドの CPU 上の配置を図示する。

68 core KNL							MPI Progress Thread				
HT0	0	1	2	3	4	...	10	...	65	66	67
HT1	68	69	70	71	72	...	78	...	133	134	135
HT2	136	137	138	139	140	...	146	...	201	202	203
HT3	204	205	206	207	208	...	214	...	269	270	271

Computation Thread

図 15　計算スレッドと通信スレッドの KNL 上での割付

7. 計算結果（Oakforest-PACS）

1 ソケットの 68 コアのうち 64 コアを使用し，残りの 4 コアに OS サービスや通信スレッドである Asynchronous Progress Thread を割り当てている。Asynchronous Progress Thread の効果を検証するために，スレッドのマッピングを変えた検証を行った。また，MPI_Isend/Irecv の効果を見るために 32 ノードで行った検証，MPI_Iallreduce の効果を見るための 4,096 ノードまでのスケーリング検証の結果を議論する。図 1，7，9，10 で紹介した Alg.1～Alg.4 について検討を実施した。

Asynchronous Progress Thread のマッピングは表 6 に示した 7 通りが考えられ，データキャッシュの共有の面では MPI プロセスが起動する物理コア内の使用されていない論理コアを使用するのが良いと考えられる（4:および 5:）。一方で，物理コア内のリソース共有を避ける面では 3:, 6:, 7:といった全く使われていない物理コア上に割付を行う方法が考えられる。問題サイズは，3,145,728 DOFs (128x128x64)を使用し，32 ノードにて実行した結果が図 16 である。1:の Asynchronous Progress Thread を使用しない（I_MPI_ASYNC_PROGRESS=off）を 1 として相対性能を示している。

Alg 1-AR, Alg. 2-AR, Alg-3-AR, Alg-4-AR は非同期集団通信関数（MPI_Iallreduce）を使用していないが，隣接ランク間ののりしろ通信に MPI_Isend/Irecv を使用しており，まずこれらの性能を見ることにより，Asynchronous Progress Thread を使用するモードにて P2P 通信の隠蔽効果を確認することができる。実際に効果が確認できるのは，7:の場合のみである。7:における隣接間通信での効果については後述する。

次に Alg.3，Alg.4 において非同期集団通信関数（MPI_Iallreduce）を使った場合（IAR）と通常の MPI_Allreduce を使った場合（AR）の比較をすることで，非同期集団通信が計算とオーバーラップが効率よく行われているか見ることができる。使用しない物理コアを割り当てる，3:と 7:の場合のみ性能向上が見られ，またより物理リソース(物理コア)を割り当てた 7:の場合に最も効果が大きい。以上により，隣接間通信や大規模での集団通信の効果を検証するマッピングを 7:とすることにした。

全体の傾向を見ると，I_MPI_ASYNC_PROGRESS=on のみを設定し通信コアの割付を規定値で使う場合(2:)では性能向上は見られず，明示的な割付が必要である。また，MPI プロセスが起動する物理コア上の論理コアに割り当てる方法(4 および 5)も効果が見られず性能が劣化している。キャッシュ上でのデータ共有の利点よりも，リソースを追加する利点の効果が大きいことがわかった。3:と 6:を比較すると，最初のタイル上に通信スレッドを割り付けるよりも，最後のスレッドに割り付けるほうが高速である。これは，最初のタイルでは OS サービスが起動していることに依るものと考えられる。ノードあたり 2 MPI プロセス以下では
最後のタイルに 4 つの通信スレッドすべてを割り付ける方法が良いと考えられる。以上をまとめると，通信用スレッドは

- 計算に使用されていない物理コアから割付
- 物理コアあたり 2 論理スレッドまで
- タイル 0 は上記条件が満たせるならば使わない

という方法が良いと考えられる。

次に，MPI_Isend/Irecv の効果を詳細に見るために比較的少ないノード数（32 ノード）で行った検証について述べる。GeoFEM は，少ないノードにおいては，MPI_Allreduce のコストは大きくなく MPI_Isend/Irecv のコストがある程度見えてくる。そのため，本検証では MPI_Iallreduce を使用しない Alg 1-AR, Alg. 2-AR, Alg-3-AR, Alg-4-AR を比較する。これらのアルゴリズムはすべて隣接間のりしろ通信に MPI_Isend/Irecv を使用している。Asynchronous Progress Thread の ON/OFF の切り替えによって，MPI_Isend/Irecv と計算とのオーバーラップが行われるかを検証した。問題サイズは，3,145,728 DOFs (128x128x64)を使用した，これは通信スレッドのマッピングを選定する検証で用いたモデルと同じサイズである。

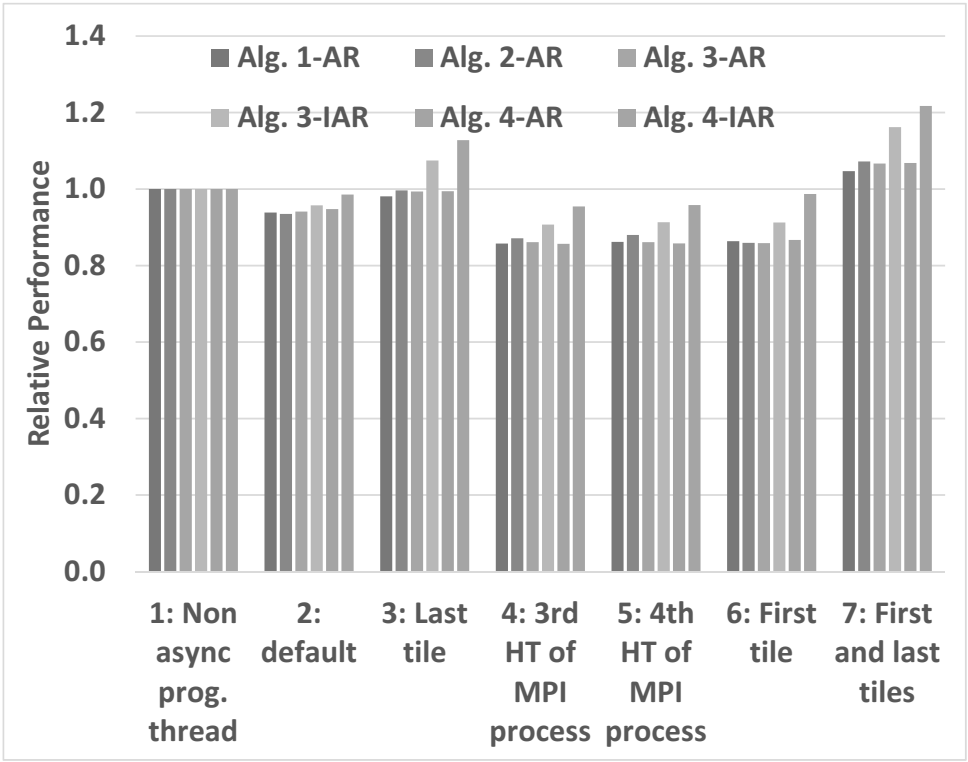

図 16 Asynchronous Progress Thread の割付による性能の変化

図 17 に各アリゴリズムでの相対実行時間を比較した。Alg. 1-AR の性能を 1 に基準をとっている。すべてのアルゴリズムにて計算と通信の隠蔽が確認できる（4.5%から 6.7%） 。これは，Asynchronous Progress Thread の設定により，通信バッファーへの詰め込み（計算）とのりしろ通信がオーバーラップしていることを示している。通信するメッセージサイズとオーバーラップさせる計算，その他の計算分との割合にもよるため，また問題サイズや実行ノード数によるところ

が大きいが，計算ノード数を少なくすることで非同期通信の効果を確認することができた。

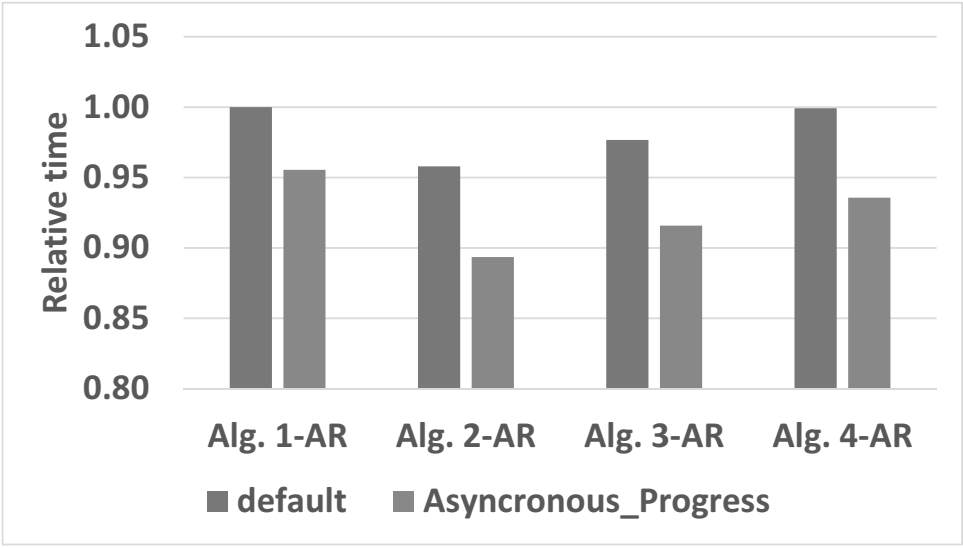

図 17　Asynchronous Progress Thread による MPI_Isend/Irecv の隠蔽 (32 ノード)

続いて，大規模ノード時における MPI_Iallreduce の検証を試みた。最小 128 ノード（1 ソケット，8,192 コア）から最大 4096 ノード（4096 ソケット，262,144 コア）までの Strong Scaling 性能を評価した．問題サイズは，100,663,296 DOF (512×256×256)を使用した。

図 18 に結果を示す。ブロックヤコビ型局所 SGS 前処理を適用しているため，コア数が増加すると反復回数が増加する傾向にあるため，反復回数あたりの性能を図示している。
Alg. 1-AR の 128 ノード 8,192 コアの性能を 8,192 として基準としている。どのアルゴリズムも 512 ノード(32,768 コア)あたりからスケーラビリティの劣化が見て取れるが，これはノード数増加に依る MPI_Allreduce のコスト増大が支配的であるものよる。Alg.3，Alg.4 において非同期集団通信関数（MPI_Iallreduce）を使った場合（IAR）と通常の MPI_Allreduce を使った場合（AR）と Alg.1 を比較したものである．非同期集団通信関数を使用しない場合は，Alg.1 とほぼ同じ挙動を示していることがわかり，非同期集団通信機能の適用により，通信と計算のオーバーラップを実現し，大幅な性能向上が得られていることがわかる．

表 7 に Alg. 3-AR と Alg.3-IAR, Alg. 4-AR と Alg.4-IAR を比較した場合の性能向上をまとめた。Alg.3 では最大 19%，Alg. 4 では 38%の効果が得られており，大幅なスケーラビリティの向上が測られた。ノード数が増えると向上率が大きくなる傾向が見て取れる。ノード数が増えると，MPI_Allreduce のコストが大きくなるために非同期通信の MPI_Iallreduce の隠蔽効果がより現れていると考えられる。

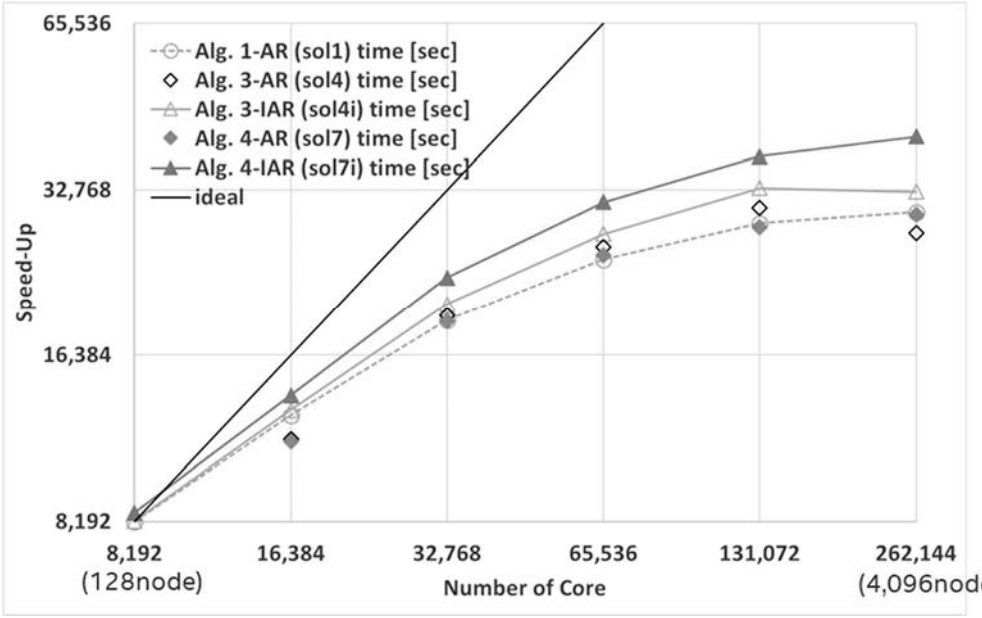

図 18　Asynchronous Progress Thread による MPI_Iallreduce の隠蔽 (128 ノード～4,096 ノード)

表 7　Asynchronous Progress Thread による効果 (128 ノード～4,096 ノード)

Core/Speedup	Alg. 3	Alg. 4
8,192	1.03	1.06
16,384	1.13	1.21
32,768	1.05	1.20
65,536	1.06	1.25
131,072	1.08	1.34
262,144	1.19	1.38

8. まとめと今後の展望

これまで MPI-3 の非同期集団通信関数の使用に効果が観測できなかった OFP において，Intel MPI 2019 の Asynchronous Progress Thread を用いることにより，計算と通信のオーバーラップが可能となり，少ノード数では MPI_Isend/Irecv，大規模ノード数では MPI_IAllreduce の隠蔽に成功した。特に後者の非同期集団通信では，4,096 ノードという大規模ノード数において 38%もの性能向上を得ることができた。

Asynchronous Progress Thread を使用し、性能向上を得るためには通信専用のコアを計算コアとは別途用意する必要がある。MPI 8 プロセス x OpenMP 8 スレッドでは、通信コアは 8 論理コアあれば良いが、例えば Flat MPI で 64 プロセス使用する場合、物理コアは 4 つしか余っていないために効率の良い通信のオーバーラップをすることができない。計算コアに何コアを使用して、通信に何コアを割り当てるかはアプリケーションに依るところが大きいが、より効率の良い MPI 実装やコアのマッピングが今後の課題である。

参考文献

[1] Ghysels, P., and Vanroose, W., Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm, Parallel Computing 40, 224-238, 2014

[2] 最先端共同 HPC 基盤施設（JCAHPC），http://jcahpc.jp/

[3] 塙敏博，中島研吾，大島聡史，星野哲也，伊田明弘，パイプライン型共役勾配法の性能評価，情報処理学会研究報告（2016-HPC-157（6））, 2016

[4] Demmel, J., Hoemmen, M., Marghoob, M., and Yelick, K., Avoiding Communication in Sparse Matrix Computations, IEEE Proceedings of 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2008), 2008

[5] Chronopoulos, A.T., and Gear, C.W., s-step iterative methods for symmetric linear systems, Journal of Computational and Applied Mathematics 25-2, 153-168, 1989

[6] Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A., and Matsuoka, S., Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer, ACM/IEEE Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11) , 2011

[7] MPI: A Message-Passing Interface Standard Version 3.0, Message Passing Interface Forum,

[8] http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012

[9] 東京大学情報基盤センター（スーパーコンピューティング部門），http://www.cc.u-tokyo.ac.jp/

[10] Nakajima, K., Parallel Iterative Solvers of GeoFEM with Selective Blocking Preconditioning for Nonlinear Contact Problems on the Earth Simulator, ACM/IEEE Proceedings of SC2003, 2003

[11] 中島研吾，片桐孝洋，マルチコアプロセッサにおけるリオーダリング付き非構造格子向け前処理付反復法の性能，情報処理学会研究報告（2009-HPC-120（6）），2009

[12] 中島研吾，拡張型 Sliced-ELL 行列格納手法に基づくメニィコア向け疎行列ソルバー，情報処理学会研究報告（2014-HPC-147（3）），2014

[13] Washio, T., Maruyama, K., Osoda, T., Shimizu, F., and Doi, S., Efficient implementations of block sparse matrix operations on shared memory vector machines. Proceedings of The 4th International Conference on Supercomputing in Nuclear Applications (SNA2000) ， 2000

[14] Smith, B., Bjϕrstad, P. and Gropp, W., Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations, Cambridge University Press ， 1996

[15] Gropp, W., Update on libraries for Blue Waters, 2010 http://jointlab.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf>.

[16] Masashi Horikoshi, Larry Meadows, Tom Elken, Pradeep Sivakumar, Edward Mascarenhas, James Erwin, Dmitry Durnov, Alexander Sannikov, Toshihiro Hanawa, and Taisuke Boku. 2018. Scaling collectives on large clusters using Intel(R) architecture processors and fabric. In Proceedings of Workshops of HPC Asia (HPC Asia '18). ACM, New York, NY, USA, 59-62. DOI: https://doi.org/10.1145/3176364.3176373

[17] Abdelhalim Amer, Charles Archer, Michael Blocksome, Chongxiao Cao, Michael Chuvelev, Hajime Fujita, Maria Garzaran, Yanfei Guo, Jeff R. Hammond, Shintaro Iwasaki, Kenneth J. Raffenetti, Mikhail Shiryaev, Min Si, Kenjiro Taura, Sagar Thapaliya, and Pavan Balaji. 2019. Software combining to mitigate multithreaded MPI contention. In Proceedings of the ACM International Conference on Supercomputing (ICS '19). ACM, New York, NY, USA, 367-379. DOI: https://doi.org/10.1145/3330345.3330378