# MPI 化支援関数群 Mutton の実装と性能評価

曽我 隆 [1)], 内田 孝紀 [2)], 伊達 進 [1)]

1) 大阪大学

2) 九州大学

soga.takashi.cmc@osaka-u.ac.jp

# Implementation and Performance Evaluation of the MPI Utility Function Suite Mutton

Takashi Soga[1)], Takanori Uchida[2)], Susumu Date[1)]

1) The University of Osaka.

2) Kyushu University.

**概要**

クラスタ型の高性能計算機（High Performance Computing，以下 HPC）システムを用いた数値シミュレーションの実行には分散メモリ並列化が必要となる．今日の大規模数値シミュレーションでは，分散メモリ並列化に MPI を用いる．通常，MPI 化に際しては，並列化を阻害する依存関係が存在せず，並列化を行なっても結果に大きな差異を生じないことの保障や，どのように計算領域（あるいは処理）を分割するのか，どのように分割した領域間のデータを通信するかを設計・実装する必要がある．しかし，計算科学に携わる研究者が MPI 化に多大なる時間を割くことは現実的ではなく，MPI 化作業を外注する場合がある．外注し，MPI 化されたコードは可読性が損なわれる場合が多く，コードの維持管理を困難にする．本研究では，利用者の MPI 化作業を軽減する MPI 化支援関数群 **Mutton** を提案する．**Mutton** は，MPI 化に必要である計算領域の分割や境界要素の交換などの煩雑な操作を代行する関数およびスレッド操作を用いる演算と通信のオーバーラップ手法や，演算と並行してファイル I/O を行うアシスタント・プロセス手法などのシミュレーションの実行時間の削減を目的とする最適化手法を適用するための関数とフレームワークで構成される．評価では，**Mutton** を用いて風車ウエイク現象をシミュレーションする実コードを MPI 化し，その効果を検証する．また，**Mutton** に含まれる静的スレッド・オーバーラップ手法やアシスタント・プロセス手法がシミュレーション時間を短縮することを明らかにする．

## 1 はじめに

数値シミュレーションは理論と実験を補完する第三の科学的手法として，流体力学・材料科学・地球科学・生物医学などの多岐にわたる分野で利用されている [1][2]. 数値シミュレーションには，実験が困難な環境・現象の再現を可能にすること [3] や，試作制作や実験に要するコストを削減する [4] などの利点がある．また，近年，数値シミュレーションの応用範囲の広がりだけでなく，高度化（高精度化・大規模化）に対する要求が高くなっており，実行に必要な計算資源量（必要なメモリ容量や演算性能）が増加している．大容量のメモリ空間を必要とする場合，単一の HPC システムだけでは確保できないため，複数の計算ノードを高速なネットワークで接続したクラスタシステムを用いる必要がある．このような複数の計算ノードが必要な数値シミュレーションプログラムは，分散メモリ並列化を実装する必要がある．

現在，多くの数値シミュレーションプログラムは MPI（Message Passing Interface）ライブラリ [5] を用いて分散メモリ並列化を実現している．MPI 化する場合，プログラムの並列性の保証や，計算領域（または処理）の分割方法，MPI が生成するプロセス間のデータの通信方法を検討・設計・実装する必要がある．計算科学の研究者を対象とした調査 [6] では，38% の人が少なくとも 20% 以上の研究時間をソフトウェア開発に費やし，また，その 45% の人がソフトウェア開発の負担が増大していると回答している．計算科学分野の研究者は自身の研究領域に対する研究が主であることから，MPI 化のような直接研究に関係しない作業に割ける工数は限られてしまい，MPI 化作業を外注する場合がある．しかし，外注した MPI 化コードは大きく書き換えられ可読性が損なわれる場合があり，コードの維持管理を困難にする．そのため，研究者が

プログラムコードに新たな機能の追加や処理内容を変更したい場合は，MPI 化前のコードを修正することになる．そのため，研究者自身でコードの維持管理を実現するには，コードの可動性を出来るだけ損なわない MPI 化が必要である．

## 2　MPI 化の要点

利用者が MPI 化を行う際，以下を利用者自身で決定する必要がある．

- 並列性の保証
- 計算領域または処理の分割方法の決定
- 並列化対象のループの始点と終点の値の修正
- プロセス間のデータ通信方法の検討

```
 5       n=1000
 6       sum=0.0d0
 7       do i=1,n
 8         sum=sum+dble(i)
 9       enddo
10       write(6,*) "Result = ",sum
```

図 1　MPI 化前の総和プログラム

図 1 に MPI 化前の総和プログラムの例を示す．プログラムは 1 から 1000 までの整数を実数型に変換して足し込み，結果を標準出力する Fortran プログラムである．

```
 7       call MPI_INIT(ierr)
 8       call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
 9       call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
10       n=1000
11       ist=((n-1)/nprocs+1)*myrank+1
12       ied=((n-1)/nprocs+1)*(myrank+1)
13       sum=0.0d0
14       do i=ist,ied
15         sum=sum+dble(i)
16       enddo
17       call MPI_REDUCE(sum,sum2,1,MPI_REAL8,MPI_SUM,0,
18      +                MPI_COMM_WORLD,ierr)
19       if(myrank.eq.0) write(6,*) "Result = ",sum2
20       call MPI_FINALIZE(ierr)
```

図 2　MPI 化後の総和プログラム

図 2 に MPI 化後の総和プログラムの例を示す．まず MPI の初期化処理を行い，総プロセス数と自プロセス番号を取得する（7-9 行）．次に各プロセスに割り当てる処理量（ループの始点と終点）を決める（11-12 行）．この例ではループ数 1000 がプロセス数で割り切れることを前提としており，割り切れない場合を考慮する場合は余り分の扱い処理の追加が必要である．各プロセスに割り当てた総和演算が終了すると，集団通信を用いてランク 0 番（マスタープロセス）に結果を集計する（17-18 行）．結果の出力はランク 0 番のみが実行するように操作し，最後に MPI の終了化を行う．このように僅か 6 行程度のプログラムを MPI 化するために，以上のような操作を追加する必要がある．

### 2.1　領域分割

MPI が生成するプロセス（プロセス数は実行時に指定するが，プログラムで固定する場合もある）に計算領域または処理を割り当てる必要がある．計算領域を分割する場合，領域をプロセスに均等に割り当てるのではなく，計算量（計算時間）が均等になるように割り当てる必要があり，領域内に計算量の不均衡がある場合は格子サイズで調整する，あるいは，割り当てるループ数をプロセス毎に変更する必要がある．また，一次元分割（配列や領域を 1 方向だけに切って分割する方法）と多次元分割（複数の方向に切って分割する方法）のどちらを採用するかでも分割方法が異なる．一次元分割は連続するメモリ空間で分割することが可能であるため，隣接間のデータ通信を容易に実装できるが，通信量が多くなるデメリットがある．多次元分割は並列化対象の方向（ループ）が複数あるため，並列数を多く設定できるメリットもあるが，連続しないメモリ領域のデータ通信が発生するため，作業配列を設けるなど処理が煩雑になる．

図 3 に計算領域を均等に分割する例を示す．この例ではプロセス数は 4 であり，Y 方向と Z 方向の二次元分割を行う．Y 方向と Z 方向のループ長が 100 と仮定すると各プロセスは

- プロセス 0 の Y 方向は 1-50、Z 方向は 1-50
- プロセス 1 の Y 方向は 51-100、Z 方向は 1-50
- プロセス 2 の Y 方向は 1-50、Z 方向は 51-100
- プロセス 3 の Y 方向は 51-100、Z 方向は 51-100

にようにループを割り当てる処理が必要になる．

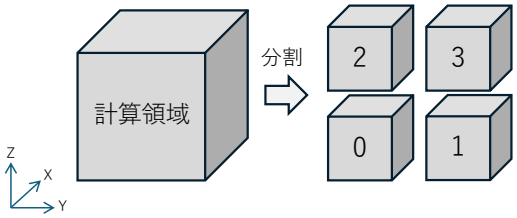


図 3　領域分割の例

### 2.2 隣接間通信

また, 数値流体解析 (Computational Fluid Dynamics：CFD）では a(i)+a(i-1) のような差分計算を多用する．差分計算が隣接するプロセスの持つデータを参照する場合（例えば a(i-1) のデータが隣接プロセスにある），隣接プロセスからのデータ通信が必要になる．図 4 に隣接するプロセスからのデータ通信の例を示す．プロセス番号 0 はプロセス 1，2，3 番と隣接するため，それぞれの境界要素を参照する．① のような Z 方向のプロセスからの通信は必要なデータが連続領域に格納されており，一括で通信することが可能である．しかし ② のような Y 方向のプロセスとの通信は非連続領域に格納されており，作業配列に一旦格納して通信する必要がある．また通信後には元の配列に戻す処理も必要である．また ③ のように対角の位置にあるプロセスからもデータの通信が発生する．このように，利用者は隣接する領域を担当するプロセス番号を把握し，②，③，④ のデータ通信処理を記述する必要がある．

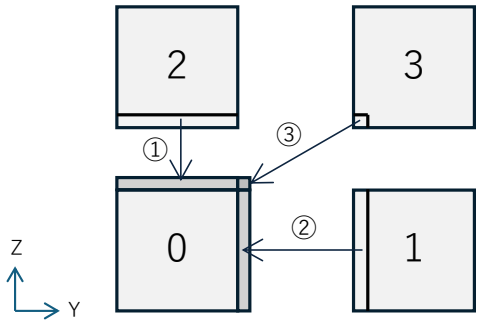


図 4　境界要素の通信の例

### 2.3 並列化を支援するツール

分散メモリ並列化を計算科学の研究者が MPI ライブラリを用いて MPI 化するのは 2.1 節や 2.2 節のような処理を自身で実装する必要がある．研究者の MPI 化作業の負担が大きいことから，並列化を支援するツールが提案されている．

分散メモリ並列化を指示文の挿入程度のプログラムの改変で可能にするツールとして，XMP（XcalableMP）[7] や HPF（High Performance Fortran）[8] がある．こられのツールを用いることで 2.1 節や 2.2 節のような処理が自動で実装される．しかし，ハードウェアの多様性への対応の遅れや性能に対する不満の声が挙がっている．

また CFD の並列化フレームワークとして，AMReX（Block-structured Adaptive Mesh Refinement (AMR) Framework for Exascale Computing）[9] が開発されている．本フレームワークを用いると AMR 付き格子管理や通信処理を代替してくれるので，研究者は数値スキームの実装に集中することが可能になる．しかし，AMR 格子を前提とするため単一格子 CFD モデルではオーバースペックである．また C++ 言語ベースであるため，Fortran 言語で記述されたプログラムに適用するのは難しい課題がある．

分散メモリ並列化のための並列オブジェクト・ランタイムシステム Charm++ [10] が米国イリノイ大学 Urbana-Champaign 校で開発されている．このシステムは MPI 化するのではなく，研究者がタスク（Chare）を定義して，ランタイムがマッピングや通信などの最適化を行うものである．研究者が通信や同期処理を細かく管理しなくても高い並列性能を得られる利点があるが，Chare という独自モデルを理解する必要がある．

本研究が提案する MPI 化支援関数群 **Mutton**（**M**PI **u**tili**t**y funct**on** suite developed at the University of Osaka）は，研究者のプログラムの可読性をできるだけ損なわず，研究者自身で MPI 化を容易に実装することを支援するものである．本報告では **Mutton** の概要と実装方法，性能評価の一例を述べる．

## 3 Mutton

**Mutton** の研究開発と検証は CFD シミュレーションの実プログラムを用いて実施している．そのため，まず CFD シミュレーションプログラムの MPI 化に必要な関数を用意している．また，現在は Fortran プログラムのインタフェースのみを提供しているが，C 言語インタフェースを整備する計画がある．本章で **Mutton** を使用する例を紹介する．

### 3.1 基本操作

現在，**Mutton** は以下の関数を用意している．

(1) MPI の初期処理・終了処理
(2) 計算領域分割（二次元分割まで）
(3) データの配布・収集
(4) 境界要素の交換
(5) 残差の集計

MPI ライブラリを直接使用せず，**Mutton** の関数のみで MPI 化することも可能であるが，処理の一部分

にのみ **Mutton** を適用することも可能である．

領域分割の方法を例に **Mutton** の利用方法を説明する．最初に，図 5 のように，利用者は MODULE 文 mpi_global を定義する必要があり，プログラム文・サブルーチン文の直後に use 文で引用する．(1) の MPI の初期処理を行う関数 inimpi を call すると，変数 nprocs には総プロセス数，変数 myrank には自プロセスのランク番号が格納される．

```
      module mpi_global
      integer :: nprocs,myrank
      integer :: ly,ry,lz,rz
      end module
```

図 5　MODULE 文 mpi_global の内容

図 6 に (2) の計算領域分割を行う関数 setuploop の呼び出し方法を示す．関数 setuploop に与える情報は，isty が Y 方向のループの始点，iedy が終点，istz が Z 方向のループの始点，iedz が終点の値である．また，Y 方向の分割数を divy，Z 方向の分割数を divz で与える．本関数では，Y 方向・Z 方向とも分割数でできるだけ均等に分割する．もしループ数が分割数で割り切れない場合は，先頭プロセスから順に余り分を加算していく．各プロセスの Y 方向の始点は配列 jst に，終点は配列 jed に格納される．各プロセスの Z 方向の始点は配列 kst に，終点は配列 ked に格納される．配列 jst，jed，kst，ked は予め (0:nprocs-1) でアロケートしておく．関数 setuploop を使用すると変数 ly，ry，lz，rz に隣接プロセスのランク番号が格納される．変数 ly と ry には Y 方向の前後，変数 lz と rz には Z 方向の前後の位置のプロセスである．隣接プロセスが存在しない場合は「-1」を格納する．(4) の境界要素の交換を実装する際に，変数 ly，ry，lz，rz の値を参照して境界要素の交換の有無を確認する．

```
      call setuploop(isty,iedy,istz,iedz,divy,divz,
     +               jst,jed,kst,ked,ierr)
```

図 6　関数 setuploop の引数

### 3.2　静的スレッド・オーバーラップ手法

スレッド・オーバーラップ手法とは，OpenMP [11] のスレッド操作を用いて演算処理と通信処理のオーバーラップを行う方法である．この方法では，マスタースレッドに通信処理を割り当て，通信処理が終了したら通信に依存しない領域の計算に参加する．また，その他のスレッドには通信に依存しない計算処理のみを割り当てる．OpenMP の schedule 節を用いた動的スケジューリングは OpenMP のオーバーヘッドの影響を受けることから，本研究では研究者が **Mutton** のループ分割関数 divloop を用いて，各スレッドに割り当てる演算量を決定する静的スレッド・オーバーラップ [12][?] を採用している．

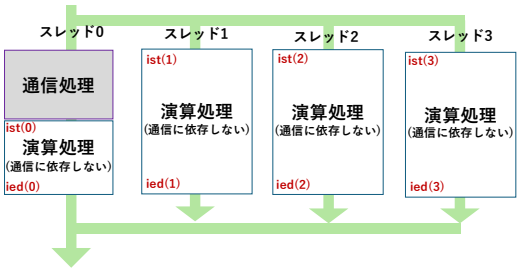


図 7　静的スレッド・オーバーラップ手法のイメージ

図 7 に静的スレッド・オーバーラップ手法のイメージ図を示す．マススタースレッド（スレッド 0 番）は通信処理を実行した後，演算処理に参加する（マススタースレッドへの演算処理の割り当て量は通信処理時間を考慮して他のスレッドと終了時間を合わせるように決定する）．他のスレッド（この例ではスレッド 1 から 3 番）には演算処理のみを割り当てる．各スレッドの演算処理のループの開始位置（配列 ist）と終了位置（配列 ied）は関数 divloop で設定する．関数 divloop の第一引数値により，マススタースレッドに割り当てる演算量を調整することが可能である．

### 3.3　アシスタント・プロセス手法

CFD のようなシミュレーションプログラムは大量の数値結果を視覚的に把握するために数値結果の可視化を行う．そのため，計算時間ステップ毎に数値結果をファイル出力する必要がある．ファイル出力時間の削減手法に MPI I/O があるが，バイナリ出力にしか対応していないなどの制約があることから，マスタープロセスに数値結果を集約してファイル出力することが多い．このファイル出力の時間を隠蔽するため，本研究では演算処理を行うプロセスとは別のプロセスを生成し，そのプロセスに数値結果を集約して演算処理と同時にファイル出力を実行するアシスタント・プロセス手法を採用した [13]．

図 8 にアシスタント・プロセスのためのコミュニケータ分割イメージを示す．アシスタント・プロセスは全体のコミュニケータ MPI_COMM_WORLD では最後のランク番号になる．**Mutton** にはコミュニ

ケータの分割やシミュレーションを担当する各ランクから数値結果をアシスタント・プロセスに集約する関数を用意しており，利用者のコード修正に負荷を軽減する．また，ファイルの出力だけでなく，アシスタント・プロセスに数値結果を集約して gnuplot のような可視化アプリケーションを実行してリアルタイム可視化を実現することも可能である．

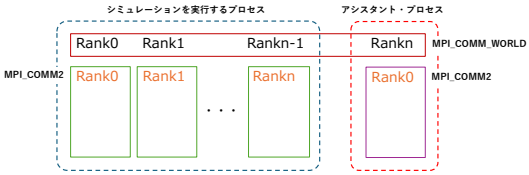


図 8　コミュニケータ分割のイメージ

## 4　性能評価

評価環境は大阪大学 D3 センターの大型計算機システム SQUID (**S**upercomputer for **Q**uest to **U**nsolved **I**nterdisciplinary **D**atascience) [14] の汎用 CPU ノード群とベクトルノード群を利用して実施した．表 1 に評価環境の諸元を示す．また評価プログラムは九州大学で開発された風車ウエイク現象シミュレーションコード [15] を使用した．

表 1　評価環境の諸元

| | |
|---|---|
| 汎用 CPU ノード群 | プロセッサ：Intel Xeon Platinum 8368 (Icelake / 2.40 GHz 38 コア) 2 基.<br>主記憶容量：256GB |
| ベクトルノード群 | VH(Vector Host)：<br>プロセッサ：AMD EPYC 7402P (2.80 GHz 24 コア) 1 基.<br>主記憶容量：128GB<br>VE(Vector Engine)：<br>プロセッサ SX-Aurora TSUBASA. Type20A(10 コア) 8 基.<br>主記憶容量：48GB |
| ストレージ | DDN EXAScaler(Lustre) HDD：20 PB |

### 4.1　並列効果の検証

風車ウエイク現象シミュレーションコードを **Mutton** の関数のみを用いて MPI 化した．検証は SQUID のベクトルノード群を使用した．本コードの実行結果の検証は，圧力値を求める Poisson 式を反復法で計算するループの残差の値で行うが，8VE を用いて 8 プロセス（プロセス内は 10 スレッド）で 5000 時間ステップ実行した結果を逐次実行と比較して問題ない誤差範囲であることを確認した．図 9 に 5000 時間ステップ実行時の時間ステップループの実行時間を示す．逐次実行は 1VE（10 スレッド）の結果である．8 プロセス（プロセス内 10 スレッド）は 552 秒となり，3.16 倍の並列効果，アムダール則により並列化率は 97.2% であった．

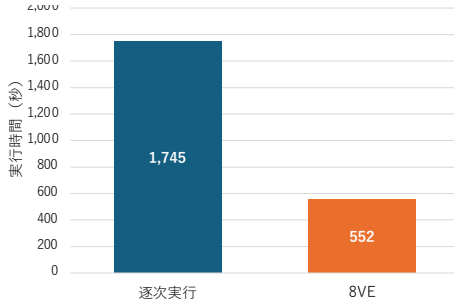


図 9　並列効果の検証結果

### 4.2　静的スレッド・オーバーラップの効果

風車ウエイク現象シミュレーションの圧力値を求める Poisson 式を反復法で計算するループの境界要素（隣接プロセスから参照される領域）を先に計算して，境界要素の交換のための通信処理と境界要素以外の領域の計算をオーバーラップするように修正している．静的スレッド・オーバーラップを適用するにあたり，演算と通信の処理時間の差が小さいことからマスタースレッドには演算処理を割り当てないように関数 divloop を設定した．

図 10 に静的スレッド・オーバーラップ手法のベクトルノード群の結果を示す．VE を 8 基使用し，VE あたり 1 プロセス，10 スレッドで実行した．結果は 5,000 時間ステップ実行時の Poisson 式の反復法ループの演算時間と境界要素の交換のための通信時間である．オリジナルは通信処理と演算処理をオーバーラップさせていない結果であり，通信時間と演算時間がほぼ等しい．スレッド・オーバーラップは静的スレッド・オーバーラップ手法を適用した結果である．マスタースレッドを通信専用に割り当てているため，9 スレッドで演算している．スレッド並列化対象のループ長が 80 であるため，オリジナルはスレッドあたり 8 ループを実行．それに対してスレッド・オーバーラップはスレッドあたり 9 ループ（最も多いスレッド）を実行しているため，演算時間が延びていると考えられる．しかし通信時間の隠蔽の効果は大きく，275 秒から 175 秒と 100 秒時間を短縮した．

図 11 は静的スレッド・オーバーラップ手法の汎用 CPU ノード群の結果である．4 ノード使用しており，各ノードに 2 プロセス（ソケットあたり 1 プロセス），プロセスあたり 32 スレッドで実行した．ベク

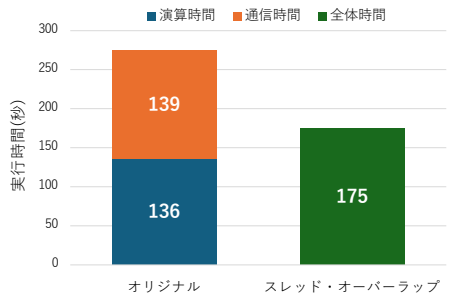


図 10　ベクトルノード群の結果

トルノード群の実行と条件を合わせているが，こちらは 866 秒の演算と通信処理の時間から約 244 秒短縮した．

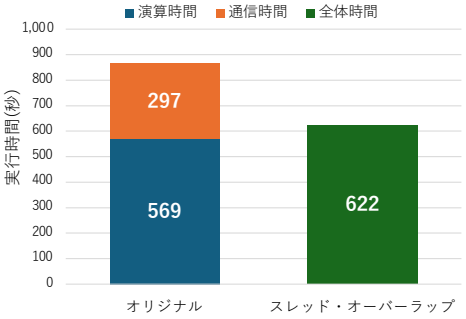


図 11　汎用 CPU ノード群の結果

以上の結果から，静的スレッド・オーバーラップ手法の通信時間の隠蔽効果は大きいことが示された．また，静的スレッド・オーバーラップ手法の実装（各スレッドの計算量の割当て）も **Mutton** のループ分割関数 divloop を用いることにより，容易に実現可能である．

### 4.3　アシスタント・プロセスの効果

風車ウエイク現象シミュレーションでは 800 時間ステップごとに可視化のため XZ 平面と XY 平面の速度 U,V,W と温度 T の値をファイルに出力している．格子サイズが（3101, 211, 161）であるので，1 回あたりの XY 平面のファイル出力量は約 16 MByte，XZ 平面のファイル出力量も約 16 MByte である．5,000 時間ステップでは計 6 回のファイル出力処理が行われる．作業配列を用いるなどの最適化の結果，ベクトルノード群のファイル出力の時間は 5.88 秒であった．

図 12 にアシスタント・プロセス手法のベクトルノード群の結果を示す．VE を 8 基使用し，VE あたり 1 プロセス，10 スレッドで実行した．アシスタント・プロセスは VH 上に生成している．シミュレーションを担当する VE 上のプロセスから数値結果を VH 上のアシスタント・プロセスに転送し，アシスタント・プロセスがファイル出力している間に VE 上のプロセスは次の処理を実行することでファイル出力の時間を隠蔽することが可能となり，5.86 秒から 0.01 秒に短縮した．

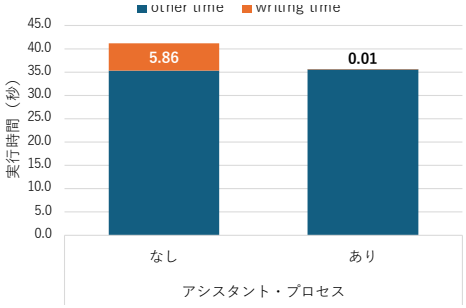


図 12　ベクトルノード群の結果

図 13 はアシスタント・プロセス手法の汎用 CPU ノード群の結果である．4 ノード使用しており，各ノードに 2 プロセス（ソケットあたり 1 プロセス），プロセスあたり 32 スレッドで実行した．汎用 CPU ノード群ではアシスタント・プロセスも汎用 CPU ノード群上に生成している．3.91 秒のファイル出力時間が 0.03 秒に短縮することを確認した．

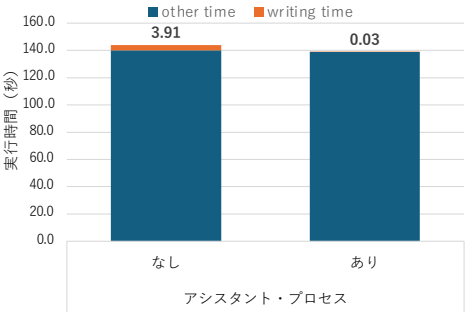


図 13　汎用 CPU ノード群の結果

このように **Mutton** が提供するアシスタント・プロセス用の関数を利用することにより，ファイル出力時間を隠蔽する効果が得られることを確認した．

## 5　まとめ

本研究は計算科学の研究者の MPI 化作業を支援するため，関数群 **Mutton** を整備・公開している．

**Mutton** には MPI 化に必要な領域分割や隣接するプロセス間の境界要素の交換を行う関数等を用意している．また通信時間を隠蔽する手法の静的スレッド・オーバーラップ手法を実装するためのループ分割関数や，ファイル出力時間を隠蔽するためのアシスタント・プロセスを実装するための関数など，シミュレーション時間を削減するための関数や実装のためのフレームワークを提供している．

現状は数値流体解析シミュレーションプログラムに特化した関数のみ，Fortran インタフェースのみであるが，今後はたの分野のプログラムコードの MPI 化を支援する関数や C 言語インタフェースを用意する計画である．

## 6 謝辞

本研究は大阪大学 D3 センターの大型計算機システム SQUID を利用している．

## 参考文献

[1] Tao Luan, "A Comprehensive Review of Simulation Technology: Development, Methods, Applications, Challenges and Future Trends," International Journal of Emerging Technologies and Advanced Applications, 1(5):9-14, June 2024. DOI: 10.62677/IJETAA.2405119.

[2] Juan M. Durán, "Computer Simulations in Science and Engineering: Concepts – Practices – Perspectives," The Frontiers Collection, Springer, Cham, 2018. DOI:https://doi.org/10.1007/978-3-319-90882-3.

[3] K. Ahmadi and S. Ismail, "Review of potential advantages and pitfalls of numerical simulation of self-excited vibrations," URI:https://bibliotekanauki.pl/articles/99795.

[4] Qifei Gu, Huichao Wu, Xue Sui, Xiaodan Zhang, Yongchao Liu, Wei Feng, Rui Zhou, and Shouying Du, "Leveraging Numerical Simulation Technology to Advance Drug Preparation: A Comprehensive Review of Application Scenarios and Cases," Pharmaceutics, vol. 16, no. 10, article 1304, 2024. DOI:vol. 16, no. 10, article 1304, 2024.

[5] MPI Forum, "https://www.mpi-forum.org".

[6] C. Storey, L. Baskerville, "Computational Science: A Field of Inquiry for Design Science Research," Proceedings of the 55th Hawaii International Conference on System Sciences, 2022 年.

[7] XcalableMP, "https://xcalablemp.org/ja/index.html".

[8] 高性能 Fortran 推進協議会, "https://site.hpfpc.org".

[9] AMReX-Codes, "https://amrex-codes.github.io/amrex/".

[10] Charm++, "https://charmplusplus.org".

[11] OpenMP.org, "http://openmp.org/wp/".

[12] T Soga, K Yamaguchi, R Mathur, O Watanabe, A Musa, R Egawa, H Kobayashi, "Effects of Using a Memory Stalled Core for Handling MPI Communication Overlapping in the SOR Solver on SX-ACE and SX-Aurora TSUBASA," Supercomputing Frontiers and Innovations, 7 (4), 4-15, Dec.2020, DOI: 10.14529/jsfi200401.

[13] 曽我隆, 内田孝紀, 伊達進, "アシスタント・プロセスによるファイル出力時間の隠蔽," 第 33 回マルチメディア通信と分散処理ワークショップ (DPSWS 2025), 2025.

[14] 大阪大学 D3 センター 大型計算機システム SQUID, "https://www.hpc.cmc.osaka-u.ac.jp/squid/".

[15] T. Uchida, and Y. Gagnon, "Effects of continuously changing inlet wind direction on near-to-far wake characteristics behind wind turbines over flat terrain," Journal of Wind Engineering and Industrial Aerodynamics, Volume 220, January 2022, 104869, DOI:https://doi.org/10.1016/j.jweia.2021.104869